View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by University of Waterloo's Institutional Repository

In-Memory Storage for Labeled Tree-Structured Data

by

Gelin Zhou

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2017

© Gelin Zhou 2017

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner	Johannes Fischer Professor
Supervisor	J. Ian Munro Professor
Co-Supervisor	Meng He Assistant Professor
Internal Examiner	Gordon V. Cormack Professor
Internal Examiner	Therese Biedl Professor
Internal-external Examiner	Gordon B. Agnew Associate Professor

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

In this thesis, we design in-memory data structures for labeled and weights trees, so that various types of path queries or operations can be supported with efficient query time. We assume the word RAM model with word size w, which permits random accesses to w-bit memory cells. Our data structures are space-efficient and many of them are even succinct. These succinct data structures occupy space close to the information theoretic lower bounds of the input trees within lower order terms.

First, we study the problems of supporting various path queries over weighted trees. A path counting query asks for the number of nodes on a query path whose weights lie within a query range, while a path reporting query requires to report these nodes. A path median query asks for the median weight on a path between two given nodes, and a path selection query returns the k-th smallest weight. We design succinct data structures to support path counting queries in $O(\lg \sigma / \lg \lg n + 1)$ time, path reporting queries in $O((occ + 1)(\lg \sigma / \lg \lg n + 1))$ time, and path median and path selection queries in $O(\lg \sigma / \lg \lg \sigma)$ time, where n is the size of the input tree, the weights of nodes are drawn from $[1..\sigma]$ and occ is the size of the output. Our results not only greatly improve the best known data structures [31, 75, 65], but also match the lower bounds for path counting, median and selection queries [86, 87, 71] when $\sigma = \Omega(n/\text{polylog}(n))$.

Second, we study the problem of representing labeled ordinal trees succinctly. Our new representations support a much broader collection of operations than previous work. In our approach, labels of nodes are stored in a preorder label sequence, which can be compressed using any succinct representation of strings that supports `access`, `rank` and `select` operations. Thus, we present a framework for succinct representations of labeled ordinal trees that is able to handle large alphabets. This answers an open problem presented by Geary et al. [54], which asks for representations of labeled ordinal trees that remain space-efficient for large alphabets. We further extend our work and present the first succinct representations for dynamic labeled ordinal trees that support several label-based operations including finding the level ancestor with a given label.

Third, we study the problems of supporting path minimum and semigroup path sum queries. In the path minimum problem, we preprocess a tree on n weighted nodes, such that given an arbitrary path, the node with the smallest weight along this path can be located. We design novel succinct indices for this problem under the indexing model, for which weights of nodes are read-only and can be accessed with ranks of nodes in the preorder traversal sequence of the input tree. One of our index structures supports queries in $O(\boldsymbol{\alpha}(m, n))$ time, and occupies $O(m)$ bits of space in addition to the space required for the input tree, where m is an integer greater than or equal to n and $\boldsymbol{\alpha}(m, n)$ is the inverse-Ackermann function. Following the same approach, we also develop succinct data structures for semigroup path sum queries, for which a query asks for the sum of weights along a given query path. Then, using the succinct indices for path minimum queries, we achieve three different time-space tradeoffs for path reporting queries.

Finally, we study the problems of supporting various path queries in dynamic settings. We propose the first non-trivial linear-space solution that supports path reporting in $O((\lg n/\lg\lg n)^2 + occ\lg n/\lg\lg n))$ query time, where n is the size of the input tree and occ is the output size, and the insertion and deletion of a node of an arbitrary degree in $O(\lg^{2+\epsilon} n)$ amortized time, for any constant $\epsilon \in (0, 1)$. Obvious solutions based on directly dynamizing solutions to the static version of this problem all require $\Omega((\lg n/\lg\lg n)^2)$ time for each node reported. We also design data structures that support path counting and path reporting queries in $O((\lg n/\lg\lg n)^2)$ time, and insertions and deletions in $O((\lg n/\lg\lg n)^2)$ amortized time. This matches the best known results for dynamic two-dimensional range counting [62] and range selection [63], which can be viewed as special cases of path counting and path selection.

Acknowledgements

First of all, I would like to express my sincere gratitude to my supervisor, Professor J. Ian Munro, for his continuous guidance and support throughout my graduate programs. It has been an honor and great experience to work with and learn from Professor Munro during my masters and doctoral studies. I decided to take an industry job and move to San Francisco 3.5 years ago. Professor Munro fully supported my choice and suggested me to continue with my PhD program as a part-time student. This thesis would not be completed without his support and encouragement.

My sincere thanks also go to my co-supervisor, Professor Meng He, for the much effort he has spent on proofreading my thesis and the papers we collaborated. He has provided a lot of helpful and insightful feedback to greatly improve the quality of my thesis. I can hardly thank him enough for doing that.

Besides my supervisor and co-supervisor, I would like to thank the rest of my thesis committee, Professor Gordon V. Cormack, Professor Therese Biedl, Professor Gordon B. Agnew, and Professor Johannes Fischer for spending time to review my thesis and attend my defense. The comments they provided have further improved the presentation of my thesis. I also want to thank Professor Joseph Cheriyan for serving as the defense chair. Big thanks to our department's administrative coordinators, Wendy Rush and Margaret Towell, for the much help they have provided me.

During my graduate studies, Professor Timothy M. Chan taught me a lot about geometric data structures, and the discussions and the collaborations with him were always interesting and informative. Before I moving to San Francisco, I was fortunate to work with Professor Venkatesh Raman and Professor Moshe Lewenstein. I learned a lot from the discussions with them.

When I was preparing for the thesis defense, I was told that Professor Alejandro López-Ortiz is fighting with a fatal disease. He is a reader of my Master's thesis and a committee member of my PhD Comprehensive-II exam. He is such a nice person and probably a friend of every graduate student of the Algorithms and Complexity Group. I would like to thank him for all his help and I wish him well.

I thank my office mates for the fruitful discussions and for all the fun we have had during the years I have spent at Waterloo: Francisco Claude, Reza Dorrigiv, Robert Fraser, Shahin Kamali, Daniela Maftulea, Patrick K. Nicholson, Alejandro Salinger, and Diego Seco. I also would like to thank Eric Y. Chen, Arash Farzan, Alexander Golynski, and Derek Phillips for sharing their industrial experience with me. Special thanks to Marilyn Miller, Professor Munro's wife, for her kindness, her help, and the delicious desserts she made. I still remember the advice for buying suits she gave to me before my Master's convocation.

Last but not least, I would like to thank my wife, Corona Luo, for her love and her support.

Dedication

This is dedicated to all the bit manipulations that make this world a better place.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In many modern computer applications, the majority of data is stored in internal memory to speed up the access and to guarantee low response time. The research on how to organize data in internal memory, i.e., in-memory data structures dates back to the early years of computer science. Researchers have focused on minimizing the amount of space occupied by the data structures as well as reducing query, update and construction times for the data structures. As data grows rapidly in volume, there has been a trend of designing space-efficient and *succinct data structures*, where the term succinct data structure was proposed by Jacobson [69] and refers to the data structures that use space close to the information-theoretic lower bounds.

As fundamental structures in computer science, trees are widely used in modeling and representing various types of data in numerous computer applications. In many cases, objects are represented by nodes and their properties are characterized by weights or labels assigned to nodes. In modern computers, many file systems can be visualized as *ordinal trees* (a rooted tree of arbitrary degree in which children are ordered) in which each node represents a file or a directory, and is annotated with metadata (e.g., size, creation time) that describes the file or directory being represented. In compilers, source code is transformed into a parsing tree, which represents the syntactic structure of the source code according to the programming language. Later, the parsing tree will be traversed to facilitate code analysis and optimization, and additional properties (e.g., cacheability, whether it depends on blocking I/Os) will be stored on each node to characterize the snippet it represents. As pointed out by Leighton [76], routing in a general network graph can be done using only the edges of a spanning tree of the graph. A *distance labeling scheme* assigns labels to the nodes, so that, given the labels of any two nodes in the network graph, the distance between these two nodes in the spanning tree of the graph can be determined.

Researchers have studied the problems of supporting *path queries*, that is, the schemes to preprocess a weighted tree such that various functions over the weights of nodes on a given query path can be computed efficiently. These queries include path minimum [2, 31, 73, 72, 21, 38], path counting [31, 65, 85], path reporting [65, 85], path median and selection [75, 65, 85], path least-frequent element, path α-minority and path top-k color queries [41].

Another topic of research is to represent ordinal trees with satellite data in the form of a label from a large alphabet. Much of this is motivated by the needs of large text-dominated databases that store and manipulate XML documents, which can be essentially modeled as ordinal trees in which each node is assigned a tag drawn from a tag set.

In this thesis, we design in-memory data structures for labeled and weighted trees, so that various types of path queries or operations can be supported with efficient query time. Our data structures are space-efficient and many of them are even succinct, which are suitable for handling large tree-structured data sets in internal storage. We further consider how to support insertions or deletions of nodes.

The investigation of data structure problems is done from a purely theoretical perspective in this thesis. Our objective is to optimize the asymptotic time and space costs, and some of our improvements achieve lower costs only when the input size is considerably large. Engineering our data structures is an interesting topic but beyond the scope of this thesis.

1.1 Organization of the Thesis

The rest of the thesis is organized as follows.

Chapter 2 summarizes the background knowledge of the research area. We first describe the models of computation and the notation used throughout the thesis. Then we review the succinct data structures for bit vectors and sequences of symbols, especially the wavelet trees and the solutions to the ball-inheritance problem. These data structures will be key building blocks for many results in this thesis. We also introduce many concepts of ordinal trees and the related data structure techniques, which will be applied extensively in later chapters.

Chapter 3 studies the problem of supporting various path queries over weighted trees. A path counting query asks for the number of the nodes on a query path whose weights are in a query range, while a path reporting query requires to report these nodes. A path median query asks for the median weight on a path between two given nodes, and a path selection query returns the k-th smallest weight. Let T denote an input tree on n nodes whose weights are drawn from $[1..\sigma]$. We design succinct data structures to encode T using $nH(W_T) + 2n + o(n \lg \sigma)$ bits of space, such that we can support path counting queries in $O(\lg \sigma / \lg \lg n + 1)$ time, path reporting queries in $O((occ + 1)(\lg \sigma / \lg \lg n + 1))$ time, and path median and path selection queries in $O(\lg \sigma / \lg \lg \sigma)$ time, where $H(W_T)$ is the entropy of the multiset of the weights of the nodes in T and *occ* is the size of the output. Our results not only greatly improve the best known data structures [31, 75, 65], but also match the lower bounds for path counting, median and selection queries [86, 87, 71] when $\sigma = \Omega(n/\text{polylog}(n))$. This chapter is based on part of the joint work with Meng He and J. Ian Munro [66].

Chapter 4 considers succinct representations of labeled ordinal trees that support a rich set of operations. Our new representations support a much broader collection of operations than previous work. In our approach, labels of nodes are stored in a *preorder label sequence*, which can be compressed using any succinct representation of strings that supports `access`, `rank` and `select` operations. Thus, we present a framework for succinct representations of labeled ordinal trees that is able to handle large alphabets. This answers an open problem presented by Geary et al. [54], which asks for representations of labeled ordinal trees that remain space-efficient for large alphabets. We further extend our work and present the first succinct representations for dynamic labeled ordinal trees that support several label-based operations including finding the level ancestor with a given label. This chapter is based on part of the joint work with Meng He and J. Ian Munro [68].

In Chapter 5, we deal with path minimum and semigroup path sum queries, and revisit path reporting queries. In the path minimum problem, we preprocess a tree on n weighted nodes, such that given an arbitrary path, the node with the smallest weight along this path can be located. We design novel succinct indices for this problem under the *indexing model*, for which weights of nodes are read-only and can be accessed with ranks of nodes in the preorder traversal sequence of the input tree. We present

- an index within $O(m)$ bits of additional space that supports queries in $O(\boldsymbol{\alpha}(m, n))$ time and $O(\boldsymbol{\alpha}(m, n))$ accesses to the weights of nodes, for any integer $m \geq n$; and
- an index within $2n + o(n)$ bits of additional space that supports queries in $O(\boldsymbol{\alpha}(n))$ time and $O(\boldsymbol{\alpha}(n))$ accesses to the weights of nodes.

Here $\boldsymbol{\alpha}(m, n)$ is the inverse-Ackermann function, and $\boldsymbol{\alpha}(n) = \boldsymbol{\alpha}(n, n)$. These indices give us the first succinct data structures for the path minimum problem. Following the same approach, we also develop succinct data structures for semigroup path sum queries, for which a query asks for the sum of weights along a given query path. One of our data structures requires $n \lg \sigma + 2n + o(n \lg \sigma)$ bits of space and $O(\boldsymbol{\alpha}(n))$ query time, where σ

is the size of the semigroup.

In the same chapter, using the succinct indices for path minimum queries, we achieve three different time-space tradeoffs for path reporting by designing

- an $O(n)$-word data structure with $O(\lg^\epsilon n + occ \cdot \lg^\epsilon n)$ query time;
- an $O(n \lg\lg n)$-word data structure with $O(\lg\lg n + occ \cdot \lg\lg n)$ query time; and
- an $O(n \lg^\epsilon n)$-word data structure with $O(\lg\lg n + occ)$ query time.

Here *occ* is the number of nodes reported and ϵ is an arbitrary constant between 0 and 1. These tradeoffs match the state of the art of two-dimensional orthogonal range reporting queries [27], which can be treated as a special case of path reporting queries. When the number of distinct weights is much smaller than n, we further improve both the query time and the space cost of these three results. This chapter is based on part of the joint work with Timothy M. Chan, Meng He and J. Ian Munro [26].

Chapter 6 considers the problem of supporting various path queries in dynamic settings. We propose the first non-trivial linear-space solution that supports path reporting in $O((\lg n/\lg\lg n)^2 + occ \lg n/\lg\lg n))$ query time, where n is the size of the input tree and *occ* is the output size, and the insertion and deletion of a node of an arbitrary degree in $O(\lg^{2+\epsilon} n)$ amortized time, for any constant $\epsilon \in (0, 1)$. Obvious solutions based on directly dynamizing solutions to the static version of this problem all require $\Omega((\lg n/\lg\lg n)^2)$ time for each node reported, and thus our data structure supports queries much faster. We also design data structures that support path counting and path reporting queries in $O((\lg n/\lg\lg n)^2)$ time, and insertions and deletions in $O((\lg n/\lg\lg n)^2)$ amortized time. This matches the best known results for dynamic two-dimensional range counting [62] and range selection [63], which can be viewed as special cases of path counting and path selection. This chapter is based on part of the joint work with Meng He and J. Ian Munro [67].

Chapter 7 provides a summary, conclusions, and some suggestions for future work.

Chapter 2

Preliminaries

2.1 Models of Computation

With the development of modern computing devices, researchers have proposed several models of computation to abstract a computer from its architectural details, so that data structure problems can be attacked more readily. There exist several theoretical models to characterize how a computer operates, each with some advantages and limitations. In the following, we are going to briefly summarize three models that will be adopted in this thesis.

Pointer Machine Model [95]. In a pointer machine[1], a data structure is represented as a directed graph in which each node stores a constant number of data values and contains a constant number of pointers to other nodes. An algorithm under this model is permitted to access a node by following a pointer to the node, to compare two values or two pointers, and to create new nodes, data values and pointers. In previous formulations [23], algorithms can perform arithmetic operations on data values but not on pointers. The time cost of an algorithm under this model is measured by the number of accesses to nodes and the number of operations performed, while the space cost is measured by the number of nodes in the directed graph.

Word RAM Model [51]. As one of its major drawbacks, a pointer machine cannot support random access memory (RAM), which is implemented in modern computer hardware. Under the word RAM model with word size w, data is stored in an infinite array of memory cells with addresses 0, 1, 2, . . ., where each cell consists of w bits and is referred to as a *word*. A set of unit cost operations are permitted to be performed, which include

[1] A pointer machine has occasionally been called a *pointer algorithm* in previous work [13].

read and write operations to memory cells in addition to arithmetic and bitwise operations on w-bit integers. The time cost of an algorithm is the number of these operations used, while the space cost is the maximum index of a word that is accessed by the algorithm. We also use the number of bits to measure the space cost. When we say a data structure occupies s bits of space, the implicitly meaning is that the space cost is $\lceil s/w \rceil$ words.

There are several variants of word RAM models with different operation sets, which prohibit unit cost multiplications and integer divisions [59], or even allow non-standard word operations [29]. However, we note that the operation set we permit is supported efficiently in most modern computer architectures and widely adopted in literature. Thus we will refer to our variant of word RAM as the standard word RAM model.

Cell Probe Model [98]. This model is similar to the word RAM model, except that there is no charge for operations on words. The running time under this model is measured by the number of accesses to memory cells only. As this model is stronger than the other two models, lower bounds obtained under the cell probe model also apply to the pointer machine model and the word RAM model. Unfortunately, it is difficult to prove meaningful cell probe lower bounds, even for simple data structure problems.

2.2 Notation

We use $\log_2 x$ to denote the base-2 logarithm of x and $\lg x$ to denote $\lceil \log_2 x \rceil$. We also use $\lg^y x$ to denote $(\lg x)^y$, or the y-th power of the logarithm of x. The iterated logarithm of x, or the log star x, is written as $\lg^* x$ and defined to be the number of times $\log_2$ must be applied before the result becomes less than or equal to 1.

For integers $a \leq b$, we use $[a..b]$ to denote the set $\{a, a+1, \ldots, b\}$.

The *Shannon entropy* of a data stream is given by the formula $H = -\sum_i p_i \log_2 p_i$, where p_i is the probability of symbol i showing up in the stream and $0 \log_2 0$ is assumed to be 0. For a fixed sequence $S[1..n]$ over the alphabet $\Sigma = [1..\sigma]$, we substitute n_i/n for p_i and obtain the *empirical entropy*, where n_i is the number of occurrences of symbol i in S. The concept of empirical entropies has been borrowed to analyze the Burrows-Wheeler transform [77]. The zeroth order empirical entropy of the sequence S is defined as

$$H_0(S) = -\sum_{i=1}^{\sigma} \frac{n_i}{n} \log_2 \big(\frac{n_i}{n}\big).$$

Note that $H_0(S)$ is less than or equal to $\log_2 \sigma$. Intuitively, the value of $nH_0(S)$ represents the output size of an ideal compressor, which uses $-\log_2 \big(\frac{n_i}{n}\big)$ bits to encode symbol i.

The compression ratio can be further improved if the codeword for each symbol is selected based on the k symbols preceding it. For any k-symbol word $W \in \Sigma^k$, we denote by S_W the subsequence of S that contains the symbols following W in S. Note that the length of S_W is equal to the number of occurrences of W in S, or one less if W is a suffix of S. The k-th order empirical entropy of S is defined as

$$H_k(S) = \frac{1}{n} \sum_{W \in \Sigma^k} |S_W| H_0(S_W).$$

The value of $nH_k(S)$ serves as a lower bound to the compression one can achieve using codes that depend on the k most recently seen symbols. It is not surprising that the value of $H_k(S)$ decreases with k.

2.3 Bit Vectors

A bit vector that supports `rank` and `select` operations is a key structure for many succinct data structures and the research work in this thesis. The positions in a bit vector of length n are numbered from 1 to n. Given a bit vector $B[1..n]$ and $\alpha \in \{0, 1\}$, we consider the following operations:

- $\texttt{access}(B, i)$: returns the i-th bit, $B[i]$, in B;
- $\texttt{rank}_\alpha(B, i)$: returns the number of α-bits in $B[1..i]$;
- $\texttt{select}_\alpha(B, i)$: returns the position of the i-th α-bit in B.

We omit the parameter B when it is clear from the context. The following lemma summarizes the support for succinct bit vectors, where part (a) is from Clark and Munro [36], while part (b) is from Raman et al. [92].

Lemma 2.3.1. *A bit vector $B[1..n]$ with m 1-bits can be represented in either (a) $n + o(n)$ bits, or (b) $\lg \binom{n}{m} + O(n \lg\lg n / \lg n)$ bits, to support* `access`*,* $\texttt{rank}_\alpha$ *and* $\texttt{select}_\alpha$ *in $O(1)$ time.*

Researchers have further considered dynamic bit vectors, for which $\texttt{insert}_\alpha(B, i)$ inserts an α-bit between $B[i - 1]$ and $B[i]$, while $\texttt{delete}(B, i)$ deletes $B[i]$. The support for these updates, which has been obtained by He and Munro [61] and Navarro and Sadakane [82], is addressed in the following lemma.

Lemma 2.3.2 ([61, 82])**.** *A dynamic bit vector $B[1..n]$ can be encoded in $n+O(n \lg\lg n / \lg n)$ bits to support* `access`*,* $\texttt{rank}_\alpha$*,* $\texttt{select}_\alpha$*,* $\texttt{insert}_\alpha$ *and* `delete` *in $O(\lg n / \lg\lg n)$ time.*

2.4 Sequences

Bit vectors can be generalized to sequences of symbols that are drawn from an alphabet $\Sigma = [1..\sigma]$. Given a sequence $S[1..n]$ and symbol $\alpha \in \Sigma$, we consider the following operations:

- $\texttt{access}(S, i)$: returns the i-th symbol, $S[i]$, in S;
- $\texttt{rank}_\alpha(S, i)$: counts the occurrences of symbol α's in $S[1..i]$, i.e., the cardinality of $\{j|S[j] = \alpha \text{ and } 1 \leq j \leq i\}$;
- $\texttt{select}_\alpha(S, i)$: returns the position of the i-th α in S, i.e., the smallest j so that $\texttt{rank}_\alpha(S, j) = i$;
- $\texttt{substr}(S, i, j)$: returns the substring $S[i..j]$;

and, in particular, for dynamic sequences:

- $\texttt{insert}_\alpha(S, i)$: inserts symbol α between $S[i-1]$ and $S[i]$;
- $\texttt{delete}(S, i)$: deletes the symbol at the i-th position of S.

As in Section 2.3, the parameter S will be omitted when it is clear from the context.

For static sequences, Belazzougui and Navarro [12] presented the following data structures to support $\texttt{access}$, $\texttt{rank}_\alpha$ and $\texttt{select}_\alpha$ operations:

Lemma 2.4.1. *Let S be a sequence of length n over an alphabet $\Sigma = [1..\sigma]$. Under the standard word RAM with word size $w = \Omega(\lg n)$,*

(a) for $\sigma = w^{O(1)}$, S can be encoded using $nH_0(S) + o(n)$ bits of space to support $\texttt{access}$, $\texttt{rank}_\alpha$ *and* $\texttt{select}_\alpha$ *in $O(1)$ time;*

(b) for $\sigma \leq n$, S can be encoded using $nH_0(S) + o(nH_0(S)) + o(n)$ bits of space to support $\texttt{access}$ *in $O(1)$ time,* $\texttt{rank}_\alpha$ *in $O(\lg \frac{\lg \sigma}{\lg w})$ time, and* $\texttt{select}_\alpha$ *in $O(f(n, \sigma))$ time, given any function $f(n, \sigma)$ that satisfies $f(n, \sigma) = \omega(1)$ and $f(n, \sigma) = o(\lg \frac{\lg \sigma}{\lg w})$;*

(c) for $\sigma \leq n$ and $\lg \sigma = \omega(\lg w)$, and for any $k = o(\log_\sigma n)$, S can be encoded using $nH_k(S) + o(n \lg \sigma)$ bits to support $\texttt{access}$ *in $O(1)$ time,* $\texttt{rank}_\alpha$ *in $O(\lg \frac{\lg \sigma}{\lg w})$ time, and* $\texttt{select}_\alpha$ *in $O(f(n, \sigma))$ time, given any function $f(n, \sigma)$ that satisfies $f(n, \sigma) = \omega(1)$ and $f(n, \sigma) = o(\lg \frac{\lg \sigma}{\lg w})$.*

Specifically, the support for retrieving a substring has been addressed by Ferragina and Venturini [46].

Lemma 2.4.2. *A sequence $S[1..n]$ over an alphabet $\Sigma = [1..\sigma]$ can be compressed into $nH_k(S) + O(\frac{n(k\lg\sigma+\lg\lg n)}{\log_\sigma n})$ bits of space to support* $\mathtt{substr}(i,j)$ *in $O((j-i+1)/\log_\sigma n)$ time.*

Very recently, the problem of supporting dynamic sequence has been extensively studied in the seminal works of Navarro and Nekrich [81] and Munro and Nekrich [79], which make novel use of dynamic fractional cascading [34].

Lemma 2.4.3. *A sequence $S[1..n]$ over an alphabet $\Sigma = [1..\sigma]$ can be compressed into $nH_k(S)+o(n\lg\sigma)$ bits of space, for any $k = o(\log_\sigma n)$, to support* $\mathtt{access}$, $\mathtt{rank}_\alpha$, $\mathtt{select}_\alpha$, $\mathtt{insert}_\alpha$ *and* $\mathtt{delete}$ *in $O(\lg n/\lg\lg n)$ time, and support* $\mathtt{substr}(i,j)$ *in $O(\lg n/\lg\lg n + (j-i+1)/\log_\sigma n)$ time.*

2.4.1 Wavelet Trees and the Ball-Inheritance Problem

As a refinement of the data structure of Chazelle for range searching problems [32], the *wavelet tree* was invented in 2003 by Grossi, Gupta and Vitter [57] to represent a static sequence of symbols succinctly and support $\mathtt{access}$, $\mathtt{rank}_\alpha$ and $\mathtt{select}_\alpha$ operations. This data structure provides an intuitive way of decomposing an alphabet. We will borrow the idea to handle path queries in Chapters 3 and 5, and we will use Ferragina, Manzini, Mäkinen and Navarro's generalized version of wavelet trees [45].

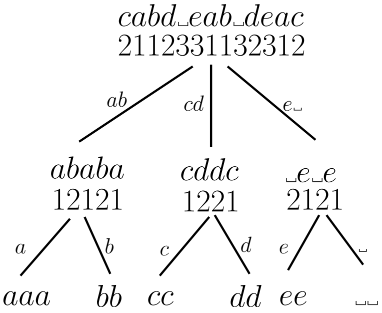


Figure 2.1: $S = cabd\text{␣}eab\text{␣}deac$ and $\Sigma = \{a, b, c, d, e, \text{␣}\}$. We set $f = 3$ and list $\widetilde{S}_v$ sequences on internal nodes.

A generalized wavelet tree for a sequence $S[1..n]$ over an alphabet Σ is constructed by recursively splitting the alphabet into $f = \lceil \lg^\epsilon n \rceil$ subsets of almost equal sizes. Each node v in that tree is associated with a subset of labels Σ_v, and a subsequence S_v of S that consists of the positions whose labels are in Σ_v. In particular, the only root node is associated with Σ, and the leaf nodes are each associated with a single label. At each non-leaf node v, a sequence $\widetilde{S}_v$ of symbols drawn from $[1..f]$ is created according to S_v. Formally, suppose that the children of v are $v_1, v_2, \ldots, v_f$, for $1 \leq i \leq |S_v|$, $\widetilde{S}_v[i] = \alpha$ if and only if $S_v[i] \in \Sigma_{v_\alpha}$. See Figure 2.1 for an example.

The levels of the generalized wavelet tree are numbered $1, 2, \ldots, h$ from top to bottom; clearly $h = \Theta(\lg n / \lg\lg n)$. For $\ell = 1, 2, \ldots, h-1$, the sequences constructed for the nodes at level ℓ are concatenated from left to right. The concatenated sequence, S_ℓ, is stored using Lemma 2.4.4 with $\Sigma = [1..f]$, so that $\texttt{rank}_\alpha$ and $\texttt{select}_\alpha$ operations on S_ℓ can be supported in $O(1)$ time.

Lemma 2.4.4. *[45, Theorem 3.1] Let S_ℓ be a sequence of length n over an alphabet $\Sigma = [1..\sigma]$, where $1 \leq \sigma \leq \sqrt{n}$. Under the standard word RAM with word size $w = \Omega(\lg n)$, S_ℓ can be represented in $nH_0(S_\ell) + O(\sigma(n \lg\lg n)/\log_\sigma n)$ bits, where $H_0(S_\ell)$ is the zeroth order empirical entropy of S_ℓ, to support* `access`*,* $\texttt{rank}_\alpha$ *and* $\texttt{select}_\alpha$ *in $O(1)$ time, and support* `substr` *in $O((j-i+1)/\log_\sigma n)$ time.*

Note that the last operation is not explicitly mentioned in the original theorem. However, in the proof of the original theorem, S_ℓ is divided into blocks of size $\lfloor \frac{1}{2} \log_\sigma n \rfloor$, and the entry of table E that corresponds to some block G can be found in constant time. We can explicitly store in each entry the content of the block that corresponds to this entry. Thus the content of any block can be retrieved in constant time. It is easy to verify that the extra space cost is $o(n)$ bits.

When σ is sufficiently small, say $\sigma = O(\lg^\epsilon n)$ for some constant $0 < \epsilon < 1$, the second term in the space cost is bounded above by $O(\sigma(n \lg\lg n)/\log_\sigma n) = O(n(\lg\lg n)^2/\lg^{1-\epsilon} n) = o(n)$ bits.

The $\texttt{rank}_\alpha$ operations can be further extended to $\texttt{count}_\beta$ operations, where $\texttt{count}_\beta(S, i)$ is defined to be the number of symbols in $S[1..i]$ that are no greater than β. Bose, He, Maheshwari and Morin presented the following lemma:

Lemma 2.4.5 ([18, Lemma 3])**.** *Let S_ℓ be a sequence of length n over an alphabet $\Sigma = [1..\sigma]$, where $\sigma = O(\lg^\epsilon n)$ for some constant $0 < \epsilon < 1$. Provided that any $\lceil \lg^\lambda n \rceil$ consecutive integers in S_ℓ can be retrieved in $O(1)$ time for some constant $\lambda \in (\epsilon, 1)$, S_ℓ can be indexed using $o(n)$ bits of additional space and $o(n)$ construction time to support* $\texttt{count}_\beta$ *in $O(1)$ time.*

Proof. Here we only analyze the construction time, which was not explicitly mentioned in the work of Bose et al. [18]. In their approach, the sequence S_ℓ is divided into blocks of size $\lceil \lg^2 n \rceil$, and each block is divided into subblocks of size $\lceil \lg^\lambda n \rceil$. They maintain the following auxiliary data structures:

- A two-dimensional array $A[1..n/\lceil \lg^2 n \rceil, 1..\sigma]$ in which $A[i,j]$ is the number of integers in $S_\ell[1..i\lceil \lg^2 n \rceil]$ that are no greater than j;
- A two-dimensional array $B[1..n/\lceil \lg^\lambda n \rceil, 1..\sigma]$ in which $B[i,j]$ is the number of integers in $S_\ell[i'..i\lceil \lg^\lambda n \rceil]$ that are no greater than j, where i' is the starting position of the block that contains $S_\ell[i\lceil \lg^\lambda n \rceil]$;
- A lookup table C that stores for every possible subblock, every integer i in $[1..\lceil \lg^\lambda n \rceil]$, and every integer j in $[1..\lceil \lg^\epsilon n \rceil]$, the number of integers at the first i positions of the subblock that are no greater than j.

The table C can be constructed in $O(n^\delta \lg^{\lambda+\epsilon} n) = o(n)$ time for some $\delta \in (\lambda, 1)$, as there are only $O(n^\delta)$ different subblocks. Under our assumption that $\sigma = O(\lg^\epsilon n)$ for some constant $0 < \epsilon < 1$, the array B can be computed in $O(n/\lceil \lg^\lambda n \rceil \times \sigma) = o(n)$ time using lookup table C and recurrence. In a similar way, the array A can be computed in $O(n/\lceil \lg^2 n \rceil \times \sigma) = o(n)$ time. □

Each position in S_v corresponds to a position in S. Chan, Larsen, and Pătraşcu [27] studied the following *ball-inheritance* problem: given an arbitrary position in some S_v, find the corresponding position i in S and the symbol $S[i]$. The support for the ball-inheritance problem is addressed in Lemma 2.4.6. Note that the original solution of Chan, Larsen, and Pătraşcu was developed for binary wavelet trees. However, their approach and conclusion can be directly extended to generalized wavelet trees.

Lemma 2.4.6. *Let $S[1..n]$ be a sequence of symbols that are drawn from $[1..\sigma]$. Given a generalized wavelet tree of S, one can build auxiliary data structures for the ball-inheritance problem with $O(n \lg n \cdot \mathtt{s}(\sigma))$ bits of space and $O(\mathtt{t}(\sigma))$ query time, where (a) $\mathtt{s}(\sigma) = O(1)$ and $\mathtt{t}(\sigma) = O(\lg^\epsilon \sigma)$; (b) $\mathtt{s}(\sigma) = O(\lg\lg\sigma)$ and $\mathtt{t}(\sigma) = O(\lg\lg\sigma)$; or (c) $\mathtt{s}(\sigma) = O(\lg^\epsilon \sigma)$ and $\mathtt{t}(\sigma) = O(1)$.*

2.5 Ordinal Trees

An ordinal tree is a rooted tree in which siblings are ordered from left to right. The *preorder traversal sequence* of an ordinal tree T, which will be referred to as the *preorder* *sequence* of T in the rest of this thesis, is a sequence that contains each node of T exactly once. The sequence starts with the root r of T, and followed by the left-to-right ordered concatenation of the preorder sequences of the subtrees rooted at the children of r.

An ordinal forest F is a left-to-right ordered list of ordinal trees. The depth, children, parent, and ancestors of a node v in F are the same as those of v in the ordinal tree containing v. The preorder sequence of F is the left-to-right ordered concatenation of the preorder sequences of the ordinal trees in F.

An ordinal forest F can be viewed as an ordinal tree rooted at a non-removable dummy node of which the children are the roots of ordinal trees in F. To be consistent, the dummy node is not taken into account for the preorder sequence, or the depth of any node in F. We will make use of both views of ordinal forests interchangeably.

Any connected subgraph of an ordinal tree T is also an ordinal tree, which will be referred to as a cover element in the context of tree covering (Section 2.5.2), or as a cluster in the context of topological partitions (Section 2.5.4). Following the notation of He et al. [64], the *preorder segments* of a connected subgraph are defined to be the maximal contiguous subsequences of nodes in the preorder sequence of T that are in the same subgraph.

Another useful order in traversing the nodes of an ordinal tree is the *depth-first unary degree sequence* (DFUDS) order as defined by Benoit et al. [15]. Given an ordinal tree T, we view its actual root r as the only child of an added dummy root r'. Starting with the dummy root r', we first visit all its children from left to right, and then recurse on the subtrees rooted at these children in the same order. We illustrate the concepts of ordinal trees, preorder sequences, preorder segments and DFUDS order in Figure 2.2.

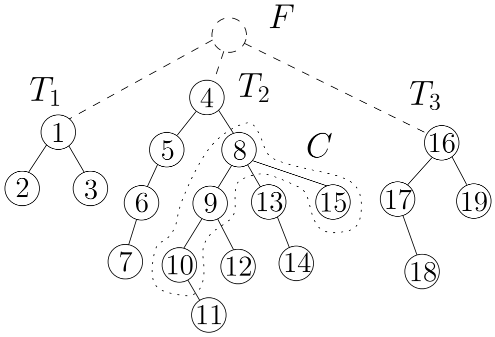


Figure 2.2: An ordinal forest F that contains 3 ordinal trees T_1, T_2 and T_3. The dummy root is represented by a dashed circle and the edges to its children are represented by dashed lines. The number on each non-dummy node is its position in the preorder sequence of F. The connected subgraph C, which is enclosed by a dotted splinegon, contains two preorder segments [8..10] and [15..15]. The nodes of T_2 in DFUDS order are $4, 5, 8, 6, 7, 9, 13, 15, 10, 12, 11, 14$.

As storage cost is a key factor in many computer applications, researchers have considered succinct representations of ordinal trees. An ordinal tree on n nodes needs to be encoded within space close to the information theoretic minimum, which is $2n - O(\lg n)$ bits as there are $\binom{2n}{n}/(n+1)$ different ordinal trees. In addition, the encoding should support various operations and updates efficiently. In Sections 2.5.1 and 2.5.2, we will review *balanced parentheses* and *tree covering* in the context of representing ordinal trees succinctly.

Section 2.5.3 describes *tree extraction*, which is inspired by *tree edit distance*. In Section 2.5.4, we will discuss *restricted topological partitions* and the underlying *directed topology tree*.

2.5.1 Balanced Parentheses

In the work of Munro and Raman on succinct ordinal trees [80], the input ordinal tree on n nodes is represented as a sequence, $P[1..2n]$, of balanced parentheses, so that each node in the input tree corresponds to a matching pair of parentheses. See Figures 5.6 and 6.5 for examples. The sequence P can be stored as a bit vector, as the opening and closing parentheses are encoded as 1-bits and 0-bits, respectively. Munro and Raman defined two operations, $\texttt{findopen}(P, x)$ and $\texttt{findclose}(P, x)$, that return the positions of the corresponding opening and closing parentheses for a given node x, respectively. The operation $\texttt{rank}_\alpha(P, i)$, which returns the number of α-bits in $P[1..i]$, can also be defined.

Navarro and Sadakane [82] further considered insertions and deletions of nodes, which were reduced to insertions and deletions of matching parenthesis pairs. Their data structure stored the underlying balanced parenthesis sequence of a dynamic ordinal tree on n nodes, using $2n + o(n)$ bits of space, such that the following operations can be performed in $O(\lg n/ \lg\lg n)$ worst-case time: (1) given the preorder rank of x, retrieving $\texttt{findopen}(P, x)$ and $\texttt{findclose}(P, x)$; (2) for any $1 \le i \le 2n$ and $\alpha \in \{0, 1\}$, computing $\texttt{rank}_\alpha(P, i)$; and (3) supporting insertions/deletions of nodes, which are denoted by $\texttt{node\_insert}$ and $\texttt{node\_delete}$.

2.5.2 Tree Covering

In this section, we briefly summarize the tree-covering based representation of ordinal trees [54, 64, 42, 43]. Later, in Section 3.4, we make use of the representation directly to reduce the space cost of our data structures. In Section 3.5.1, we extend the representation to support more powerful operations such as $\texttt{node\_count}_\beta$ and $\texttt{node\_summarize}$. We review only the notation and techniques related to our requirements.

The tree-covering based representation was proposed by Geary, Raman and Raman [54] to represent an ordinal tree succinctly to support navigational operations. Let T be an ordinal tree on n nodes. A tree cover of T with a given parameter M is essentially a set of $O(n/M)$ *cover elements*, each being a connected subtree of size $O(M)$. These subtrees, being either disjoint or joined at the common root, cover all the nodes in T.

Geary et al. [54] proposed an algorithm to cover T with *mini-trees* or *tier-1 subtrees* for $M = \max\{\lceil (\lg n)^4 \rceil, 2\}$. Again, they apply the algorithm to cover each mini-tree with *micro-trees* or *tier-2 subtrees* for $M' = \max\{\lceil \lg n/24 \rceil, 2\}$. For $k = 1, 2$, the nodes that are roots of the tier-k subtrees are called *tier-k roots*. Note that a tier-k root can be the root node of multiple tier-k subtrees, and a tier-1 root must also be a tier-2 root.

To use tree-covering based representations to support more operations, He et al. [64, Definition 4.22] proposed the notion of tier-k *preorder segments*, i.e., maximal substrings of nodes in the preorder traversal sequence that are in the same mini-tree (tier-1 segments) or micro-tree (tier-2 segments). Farzan and Munro [42] further modified the tree-covering algorithm. Their algorithm produces a tree cover such that nodes in a cover element are distributed into a constant number of preorder segments, as proved by Farzan et al. [43]. These results are summarized in the following lemma:

Lemma 2.5.1 ([42, Theorem 1] and [43, Lemma 2])**.** *Let T be an ordinal tree on n nodes. For a fixed parameter M, the tree decomposition algorithm in [42] covers the nodes in T by $\Theta(n/M)$ cover elements of size at most $2M$, all of which are pairwise disjoint other than their root nodes. In addition, nodes in one cover element are distributed into a constant number of preorder segments.*

The techniques in [54, 64, 42, 43] encode an unlabeled ordinal tree on n node in $2n+o(n)$ bits to support in constant time a set of operations related to nodes, tier-k subtrees and tier-k roots. In these representations, nodes are identified by their preorder ranks. Given an arbitrary node x, we can compute its depth, and find its i-th (lowest) ancestor. Given two nodes x and y, we can compute their lowest common ancestor (LCA) [1, 14].

For $k = 1$ or 2, the tier-k subtrees are ordered by the preorder ranks of their root nodes (breaking ties with the preorder ranks of arbitrary non-root nodes) and specified by their ranks. The following operations can be performed in constant time: For each tier-k subtree, we can find its root node, compute the tier-k preorder segments whose nodes belong to this subtree, and select the i-th node in preorder that belongs to this subtree. For each micro-tree, we can compute the encoding of its structure. For each node that is not a tier-k root, we can find the tier-k subtree to which it belongs, and its relative preorder rank in this tier-k subtree.

Similarly, for $k = 1$ or 2, the tier-k roots are ordered in preorder and specified by their ranks. Let r_i^1/r_i^2 denote the i-th tier-1/tier-2 root. Given an arbitrary node x, we can compute its rank if x is a tier-k root, or determine that x is not a tier-k root. Conversely, given the rank of a tier-k root, we can compute its preorder rank. We summarize these operations in the following lemma:

Lemma 2.5.2. *Let T be an unlabeled ordinal tree on n nodes. T can be represented in $2n + o(n)$ bits and $O(n)$ construction time such that the operations described in the three previous paragraphs can be supported in $O(1)$ time.*

Geary et al. [54] further considered labeled ordinal trees. Their results are addressed in Lemma 2.5.3. We list only a small subset of the operations supported by Geary et al.'s [54] succinct representation of labeled ordinal trees. For simplicity, a node is said to be an *α-node* if its label is α, and an α-node is an *α-ancestor* of its descendants. In addition, we assume that nodes x and y are contained in T, $\alpha \in \Sigma$, a node precedes itself in preorder, and a node is its own 0-th ancestor.

Lemma 2.5.3. *Let T be an ordinal tree on n nodes, each having a label from an alphabet Σ of size σ. Then T can be represented using $O(n)$ construction time and $n(\lg \sigma + 2) + O(\sigma n \lg\lg\lg n / \lg\lg n)$ bits of space, so that the following operations can be supported in $O(1)$ time.*

- `pre_rank`(T, x): *returns the number of nodes that precede x in preorder;*
- `pre_rank`$_\alpha(T, x)$: *returns the number of α-nodes that precede x in preorder;*
- `pre_select`(T, i): *returns the i-th node in preorder;*
- `pre_select`$_\alpha(T, i)$: *returns the i-th α-node in preorder;*
- `depth`(T, x): *returns the number of ancestors of x;*
- `depth`$_\alpha(T, x)$: *returns the number of α-ancestors of x;*
- `parent`(T, x): *returns the parent of x;*
- `level_anc`(T, x, i): *returns the i-th lowest ancestor of x;*
- `level_anc`$_\alpha(T, x, i)$: *returns the i-th lowest α-ancestor of x;*
- `lowest_anc`$_\alpha(T, x)$: *returns the lowest α-ancestor of x if such an α-ancestor exists, otherwise returns* `NULL`*;*

- `LCA`(T, x, y): *returns the lowest common ancestor of x and y.*

It should be drawn to the reader's attention that, due to the assumption that nodes are denoted by their preorder ranks, `pre_rank`(T, x) and `pre_select`(T, i) are essentially identity functions. We preserve these operations for the following reason: Later on, we will consider the preorder ranks of nodes in trees extracted from the given tree (like $T_{a,b}$'s in Section 3.4 and T_ℓ's in Section 3.5), and we may use these operations to explicitly specify which tree we refer to when we mention the preorder rank of a certain node.

2.5.3 Tree Extraction

The technique of tree extraction is introduced in the work of He et al. [65] and the author's Master's thesis [100], and will be further used in this thesis. We revisit the deletion operation of tree edit distance [16]. Unlike the original definition, here we can delete any node from an ordinal forest, even if it is the root of an ordinal tree. Suppose we want to delete a node u from an ordinal forest F and u is contained in some ordinal tree T of F. Here are two cases for the deletion of u, which are illustrated in Figure 2.3.

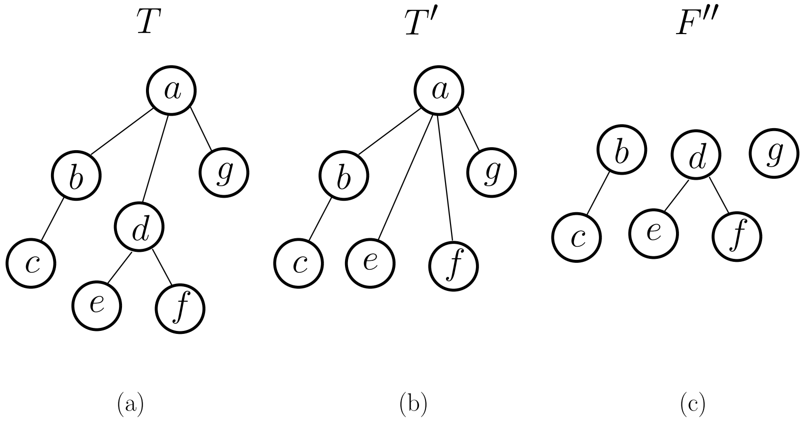


Figure 2.3: (a) An ordinal tree T on nodes that are identified by the letters. (b) The ordinal tree T' obtained by deleting a non-root node d from T. (c) The ordinal forest F'' obtained by deleting the root node a from T.

- Case 1: When u is not the root node of T and has a parent node v, we insert u's children in place of u into the list of children of v, preserving the original left-to-right order.

- Case 2: When u is the root node of T, we insert the subtrees rooted at u's children in place of T into the list of ordinal trees of F, preserving the original left-to-right order.

The deletion operation preserves the ancestor-descendant relationship, and the relative positions in preorder among the remaining nodes.

Let F be an ordinal forest and let $V(F)$ be the set of nodes in F. For any set $X \subseteq V(F)$, we denote by F_X (or T_X if F_X contains only a single tree) the ordinal forest obtained by deleting from F all the nodes of $V(F) - X$, where the nodes are deleted from bottom to top. (In fact, the order of deletion does not matter.) F_X or T_X is called the X-*extraction* of F. See Figure 2.4 for an illustration.

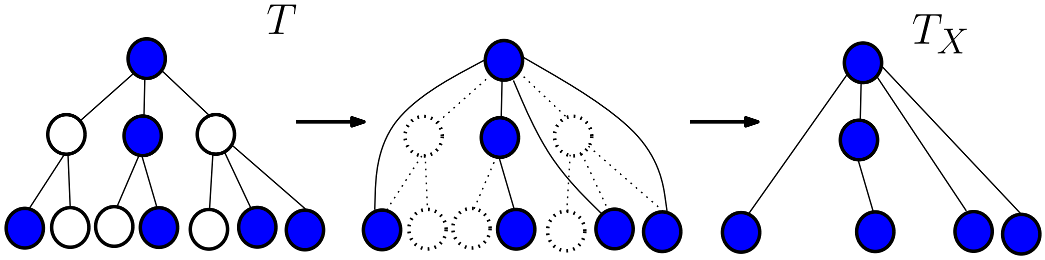


Figure 2.4: An example of T_X, i.e., the X-extraction of T, for which X is set of all dark nodes.

After the extraction, each node in X still occurs in the resulting forest F_X, though it may be moved to a different position. The occurrences of this node in F and F_X are viewed as distinct nodes, which are said to *correspond* to each other. Thus we obtain a natural one-to-one correspondence between the nodes in X and the nodes in F_X. To clarify notation, we denote by u_X a node in F_X if and only if the node in F that corresponds to u_X is denoted by u.

Lemma 2.5.4 captures some essential properties of tree extraction.

Lemma 2.5.4 ([65, 100]). *For any two sets of nodes $X, X' \subseteq V(T)$ that contain the root of T, the nodes in $X \cap X'$ have the same relative positions in the preorder and the postorder traversal sequences of T_X and $T_{X'}$.*

Proof. Let T be an arbitrary ordinal tree, and let T' be the ordinal tree after deleting a non-root node x from T. By the definition of the deletion operation of tree edit distance, the preorder/postorder traversal sequence of T' is the same as the preorder/postorder traversal sequence of T with x being removed from the sequence.

Let $Y = X \cap X'$. Since T_X is obtained by deleting nodes from T, by applying the above proposition multiple times, we can show that the nodes in Y have the same relative positions in the preorder/postorder traversal sequences of T and T_X. A similar claim holds for T and $T_{X'}$. Thus the lemma holds. □

2.5.4 Restricted Topological Partitions

Topological partitions and restricted topological partitions have found applications in computing the k smallest spanning trees of a graph [48, 49], and in dynamic maintenance of minimum spanning trees and connectivity information [48], 2-edge-connectivity information [49], and a set of rooted trees that support link-cut operations [50]. In this thesis, we follow the definitions and notation of restricted topological partitions and directed topology trees [50, 49].

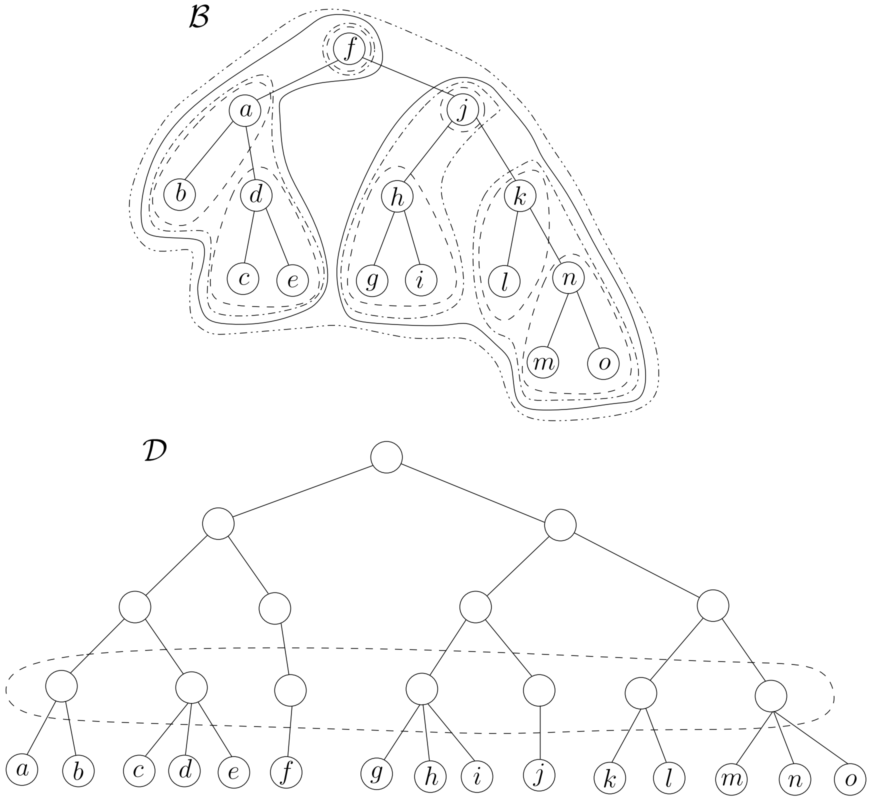


Figure 2.5: A binary tree $\mathcal{B}$, its restricted multilevel partition with $s = 3$, and the corresponding directed topology tree $\mathcal{D}$. Here the base clusters are enclosed by dashed splinegons.

Let $\mathcal{B}$ be a rooted binary tree. A *cluster* with respect to $\mathcal{B}$ is a subset of nodes whose induced subgraph forms a connected component. The *external degree* of a cluster is the number of edges that have exactly one endpoint in the cluster. These endpoints are referred to as the *boundary nodes* of the cluster. Note that the root of $\mathcal{B}$ does not have to be a boundary node. For two disjoint clusters C_1 and C_2, C_1 is said to be a *child cluster* of C_2 if C_1 contains a node whose parent is contained in C_2. A *restricted partition of order s* of $\mathcal{B}$ is defined to be a partition that satisfies the following conditions:

- each cluster has external degree at most 3 and at most 2 child clusters;
- each cluster has at most s nodes;
- each cluster that has 2 child clusters contains only one node;
- no two adjacent clusters can be combined without breaking the above conditions.

Frederickson gave a linear-time algorithm that can create a restricted partition of order s for a given binary tree on n nodes, and proved that the number of clusters is $\Theta(\lceil n/s \rceil)$.

Frederickson further defined a *restricted multilevel partition* of a binary tree $\mathcal{B}$ consisting of a set of h partitions of the nodes. A restricted multilevel partition of order s can be computed recursively as follows: The clusters at level 0, which are called *base clusters*, are obtained by computing a restricted partition of order s of $\mathcal{B}$. Then, to compute the level-ℓ clusters for any level $\ell > 0$, view each cluster at level $\ell - 1$ as a node, make two of these nodes adjacent if their corresponding clusters are adjacent, and then compute a restricted partition of order 2 of the resulting tree. This recursion stops when the partition contains only one cluster containing all the nodes, which is the level-h cluster.

By the properties of restricted partitions, a level-ℓ cluster with 2 child clusters contains exactly one level-$(\ell-1)$ cluster, which also has two child clusters at level $\ell-1$. By induction, this level-ℓ cluster contains only a single node of the binary tree $\mathcal{B}$. Therefore, any cluster in the restricted multilevel partition either has at most one child cluster or contains only one node of $\mathcal{B}$. We follow the notation of He et al. [64] and define the preorder segments of a cluster to be the maximal contiguous subsequences of nodes in the preorder sequence of $\mathcal{B}$ that are contained in the same cluster. Thus Frederickson's approach guarantees that each cluster contains at most 2 boundary nodes and at most 2 preorder segments.

A *directed topology tree* $\mathcal{D}$ is further defined for a restricted multilevel partition of a binary tree $\mathcal{B}$, as illustrated in Figure 2.5. $\mathcal{D}$ contains $h + 1$ levels. A node of $\mathcal{D}$ at level ℓ represents a level-ℓ cluster of the hierarchy, and, if $\ell > 0$, its children represent the clusters at the lower level that partition this level-ℓ cluster. Additional links are maintained between each pair of adjacent nodes at the same level of $\mathcal{D}$. Frederickson proved the following lemma about the number of levels.

Lemma 2.5.5 (Theorem 2.3 in [49]). *For $\ell > 0$, the number of clusters at level ℓ is at most $5/6$ the number of clusters at level $\ell - 1$. The value of h is bounded above by $O(\lg n)$.*

Frederickson [49, 50] further used topology trees to maintain a dynamic forest of binary trees, to support two operations: `link` which combines two trees in the forest into one by adding an edge between the root of one binary tree and an arbitrary given node of the other that has less than two children, and `cut` which breaks one tree into two by removing an arbitrary given edge. The following lemma summaries a special case of their results to be used in our solutions, in which we say that a cluster is *modified* during updates if it is deleted or created during this update, its nodes or edges have been changed or an edge with one endpoint in the cluster has been inserted or deleted:

Lemma 2.5.6 ([49, 50]). *The topology trees of the binary trees in a given forest F on n nodes can be maintained in $O(s+\lg n)$ time for each `link` and `cut`, where s is the maximum size of base clusters. Furthermore, each `link` or `cut` modifies $O(1)$ clusters at any level of the topology trees maintained for the two binary trees updated by this operation, and once a cluster is modified, the clusters represented by the ancestors of its corresponding node in the topology tree are all modified. These topology trees have $\Theta(f + n/s)$ nodes in total, where f is the current number of trees in F, and occupy $O(S + (f + n/s)\lg n)$ bits in total, where S is the total space required to store the tree structures of base clusters.*

Chapter 3

Static Succinct Data Structures for Path Queries

3.1 Introduction

In this chapter, we design data structures to maintain a weighted tree on n nodes such that we can support several path queries. We consider path counting, path reporting, path median and path selection queries, where the first two varieties of queries are also called path searching queries. The formal definitions of these queries are listed below. For simplicity, let $P_{u,v}$ denote the set of nodes on the path from u to v, where u and v are nodes in the given tree. Likewise, let $R_{p,q}$ denote the set of nodes in the given tree whose weights are in the range $[p, q]$.

- Path Counting Query: Given two nodes u and v, and a range $[p, q]$, return the cardinality of $P_{u,v} \cap R_{p,q}$;
- Path Reporting Query: Given two nodes u and v, and a range $[p, q]$, return the nodes in $P_{u,v} \cap R_{p,q}$;
- Path Selection Query: Given two nodes u and v, and an integer $1 \leq k \leq |P_{u,v}|$, return the k-th smallest weight in the multiset of the weights of the nodes in $P_{u,v}$. If k is fixed to be $\lceil |P_{u,v}|/2 \rceil$, then path selection queries become path median queries.

When the given tree is a path, the above queries become two-dimensional orthogonal range counting and range reporting queries, range median, and range selection queries, respectively. Thus the path queries we consider generalize these fundamental queries to weighted trees.

We represent the input tree as an ordinal (ordered) tree. This does not significantly impact the space cost, which is dominated by storing the weights of nodes. In addition, we assume that the weights are drawn from $[1..\sigma]$, or rank space. Thus a query range is an integral one, denoted by $[p..q]$ in the rest of this chapter. We later analyze time and space costs in terms of n and σ. Unless otherwise specified, our discussions are based on the standard word RAM model with word size $w = \Omega(\lg n)$; we also consider the pointer machine model and the cell probe model.

3.1.1 Previous Work

Path counting. The problem of supporting path counting queries was proposed and studied by Chazelle [31]. In his formulation, weights are assigned to edges instead of nodes, and a query asks for the number of edges on a given path whose weights are in a given range. Chazelle designed a linear space data structure to support queries in $O(\lg n)$ time, which is based on tree partition. He proved that any tree T containing at least two edges can be partitioned into two subtrees that contain at least one-third of the edges of T. Based on this lemma, an emulation dag of the input tree can be constructed, in which each edge corresponds to a canonical path in the tree. Chazelle further showed that any path in the input tree can be partitioned into $O(\lg n)$ canonical paths. He finally obtained the data structure by a generalized range tree, along with a compaction technique in [32]. The bottleneck of the time cost is due to the number of canonical paths in the partitions of query paths, so this approach requires $O(\lg n)$ time for an arbitrary set of weights.

With the technique of tree extraction, He et al. [65] slightly improved Chazelle's results. Their linear space data structure requires $O(\lg \sigma)$ query time when the weights are drawn from a set of σ distinct values.

Path reporting. The path reporting problem was proposed by He et al. [65], who obtained two solutions under the word RAM model: One requires $O(n)$ words of space and $O(\lg \sigma + occ \lg \sigma)$ query time, the other requires $O(n \lg\lg \sigma)$ words of space but only $O(\lg \sigma + occ \lg\lg \sigma)$ query time, where occ is the size of output and σ is the number of distinct weights.

Path median and selection. The path median problem was proposed by Krizanc et al. [75], who presented two solutions for this problem under the pointer machine model. The first one supports queries in $O(\lg n)$ time, and occupies $O(n \lg^2 n)$ space. The second one requires $O(b \lg^3 n / \lg b)$ query time and $O(n \log_b n)$ space, for any $2 \le b \le n$. If b is set to be $n^\epsilon / \lg^2 n$ for some small constant $\epsilon > 0$, then the space cost becomes linear, but the query time is $O(n^\epsilon)$. These are the best known results for the path median problem.

The approach of Krizanc et al. [75] is also based on tree partition. Their data structures maintain a set of subpaths in which the weights of the nodes are sorted, such that any query path can be divided into the disjoint union of two or more subpaths in the set. Thus the answer to the query can be obtained by performing a binary search on these subpaths. The time and space cost is determined by the number of subpaths required to partition a query path.

He et al. [65] significantly improved Krizanc et al.'s results. Under the standard word RAM model, their linear space data structure requires $O(\lg \sigma)$ query time only, where σ is the size of the set of weights.

Two-dimensional orthogonal range searching. As a fundamental problem in computational geometry, the two-dimensional orthogonal range searching problem arises from databases and geographic information system (GIS) applications. In this problem, we maintain a set, N, of points on an $n \times n$ grid, such that the information about the points in R, a given rectangle whose edges are parallel to the axes, can be retrieved efficiently. A range counting query returns the cardinality of $N \cap R$, and a range reporting query returns the points in $N \cap R$.

The time-space tradeoff of the two-dimensional orthogonal range counting problem has been well studied. Pătraşcu [86, 87] proved that any data structure occupying $O(n \cdot w^{O(1)})$ bits of space requires $\Omega(\lg n / \lg w)$ query time under the cell probe model. In fact, the optimal query time can be achieved with linear space [70, 18, 28]. Chan and Pătraşcu [28] also studied the construction time. Their data structure supports queries in $O(\lg n / \lg \lg n)$ time with linear space, and requires only $O(n\sqrt{\lg n})$ time in preprocessing, while the earlier data structures with logarithmic query time require $\Omega(n \lg n)$ construction time.

The two-dimensional orthogonal range reporting problem has also been studied heavily. Let *occ* denote the size of the output. Under the pointer machine model, Chazelle [30] provided a data structure that answers queries in $O(\lg n + occ)$ time using $O(n \lg n / \lg \lg n)$ space. This is optimal due to a lower bound later proved by Chazelle [33], which indicates that any data structure for this problem using $O(\lg^{O(1)} n + occ)$ query time requires $\Omega(n \lg n / \lg \lg n)$ space.

Under the word RAM model, the best time-space tradeoff for this basic problem is still open. By a reduction from the colored predecessor search problem [88], one can show that any data structure using $O(n \lg^{O(1)} n)$ bits of space requires $\Omega(\lg \lg n + occ)$ query time. The best known data structures that achieve the optimal query time are due to Alstrup et al. [3] and Chan et al. [27], both of which require $O(n \lg^\epsilon n)$ words of space for any constant $\epsilon > 0$. There still exist data structures that use less space but more query time. In the recent work of Chan et al. [27], one of their solutions achieves $O(\lg \lg n + occ \lg \lg n)$ query time with $O(n \lg \lg n)$ words of space, and another solution supports queries in $O(\lg^\epsilon n + occ \lg^\epsilon n)$

time using linear space.

Range median and selection. The range median and selection problem was also proposed by Krizanc et al. [75]. In this problem, an unsorted array of n elements is given, and a query asks for the median or the k-th smallest element in a range. To obtain constant query time, the known solutions require near-quadratic space [75, 89, 90]. For linear space data structures, Gfeller and Sanders [55] presented a solution to answer a query in $O(\lg n)$ time. Gagie et al. [53] considered this problem in terms of σ, the number of distinct weights, and designed a data structure based on wavelet trees [32, 57] that supports range selection queries in $O(\lg \sigma)$ time. The best upper bound was achieved by Brodal et al. [24, 23], which requires $O(\lg n / \lg \lg n)$ query time. Jørgensen and Larsen [71] later showed that Brodal et al.'s result is optimal by proving a lower bound of $\Omega(\lg n / \lg w)$ on query time under the cell probe model, provided that data structures for the static range selection problem use $O(n \cdot w^{O(1)})$ bits of space.

Succinct data structures for range queries. For two-dimensional orthogonal range searching queries, the best known result is due to Bose et al. [18]. They presented a data structure that encodes a set of n points on an $n \times n$ grid using $n \lg n + o(n \lg n)$ bits to support range counting queries in $O(\lg n / \lg \lg n)$ time, and range reporting queries in $O((occ + 1) \cdot \lg n / \lg \lg n)$ time, where occ is the size of the output. For range median and range selection queries, Brodal et al. [24, 23] claimed that their data structure achieving $O(\lg n / \lg \lg n)$ query time uses $O(n)$ words of space. A careful analysis reveals that their results are even better than claimed: the space cost of their data structure is only $n \lg n + o(n \lg n)$ bits in rank space, i.e., their structure is succinct.

Succinct representations of static trees. The problem of encoding a static tree succinctly has been studied extensively. For unlabeled trees, a series of succinct representations have been designed [54, 64, 42, 43, 82]. For labeled trees, Geary et al. [54] presented a data structure to encode a tree on n nodes, each having a label drawn from an alphabet of size σ, supporting a set of operations in constant time. The overall space cost is $n(\lg \sigma + 2) + O(\sigma n \lg \lg \lg n / \lg \lg n)$ bits, which is much more than the information-theoretic lower bound of $n \lg \sigma + 2n - O(\lg n)$ bits when $\sigma = \omega(\lg \lg n)$. Ferragina et al. [44] and Barbay et al. [9, 10] designed data structures for labeled trees using space close to the information-theoretic minimum, but supporting a more restricted set of operations. The data structure of Ferragina et al. [44] also supports SubPathSearch, which returns the number of nodes whose upward paths start with a given query string. The most functional succinct representation for labeled ordinal trees is due to He et al. [68], which supports a rich set of operations and preserves succinctness over large alphabets. We will describe this data structure in detail in Chapter 4.

3.1.2 Our Contributions

The primary contributions of this chapter are the succinct data structures for path queries, which are described in Section 3.5 and heavily based on the technique of tree extraction. It should be drawn to the reader's attention that the technique of tree extraction along with the data structures described in Sections 3.2 to 3.4 are from the joint work with Meng He and J. Ian Munro [65] and the author's Master's thesis [100]. They are repeated to help the reader understand Section 3.5.

The technique of tree extraction is inspired by the deletion operation of tree edit distance [16], though its usage here is completely different from computing the edit distance between two labeled ordinal trees. The basic idea of tree extraction is to extract a subset of nodes from an ordinal tree or an ordinal forest, retaining some relative properties among the nodes in this subset. This technique allows us to perform divide-and-conquer approaches, in a space-efficient way, on the set of the weights rather than the structure of the input tree. Our methods in this chapter are completely different from approaches used in [31, 75] that partition tree structures directly. Following the technique of tree extraction, we obtain the following results:

Table 3.1: Our results on the path counting, path reporting, path median and path selection problems, along with previous results. Here $H(W_T)$ is the entropy of the multiset of the weights of the nodes in T. Note that $H(W_T)$ is at most $\lg \sigma$, which is $O(w)$.

Path Query Type	Source	Model	Space	Query Time
Counting	Theorem 3.3.3 [65]	Pointer Machine	$O(n \lg \sigma)$	$O(\lg \sigma)$
	[31]	RAM	$O(n)$ words	$O(\lg n)$
	Theorem 3.4.1 [65]	RAM	$O(n)$ words	$O(\lg \sigma)$
	Theorem 3.5.8	RAM	$nH(W_T) + 2n + o(n \lg \sigma)$ bits	$O(\lg \sigma / \lg \lg n + 1)$
Reporting	Theorem 3.3.3 [65]	Pointer Machine	$O(n \lg \sigma)$	$O(\lg \sigma + occ)$
	Theorem 3.4.1 [65]	RAM	$O(n)$ words	$O(\lg \sigma + occ \lg \sigma)$
	[65]	RAM	$O(n \lg \lg n)$ words	$O(\lg \sigma + occ \lg \lg \sigma)$
	Theorem 3.5.8	RAM	$nH(W_T) + 2n + o(n \lg \sigma)$ bits	$O((occ + 1)(\lg \sigma / \lg \lg n + 1))$
Median / Selection	[75]	Pointer Machine	$O(n \lg^2 n)$	$O(\lg n)$
	[75]	Pointer Machine	$O(n \lg_b n)$	$O(b \lg^3 n / \lg b)$, for $2 \leq b \leq n$
	[75]	Pointer Machine	$O(n)$	$O(n^\epsilon)$
	Theorem 3.3.3 [65]	Pointer Machine	$O(n \lg \sigma)$	$O(\lg \sigma)$
	Theorem 3.4.1 [65]	RAM	$O(n)$ words	$O(\lg \sigma)$
	Theorem 3.5.9	RAM	$nH(W_T) + 2n + o(n \lg \sigma)$ bits	$O(\lg \sigma / \lg \lg \sigma)$

We design succinct data structures for path queries over an ordinal tree T on n nodes, each having a weight drawn from $[1..\sigma]$. Let W_T denote the multiset that consists of the weights of the nodes in T. Our data structures occupy $nH(W_T) + 2n + o(n \lg \sigma)$ bits of space, which is close to the information-theoretic lower bound ignoring lower-order terms, achieving faster query time compared to the best known $O(n \lg n)$-bit data structures [65], which are not succinct. We summarize our results in Table 3.1, along with previous results.

3.1.3 The Organization of This Chapter

The rest of this chapter is organized as follows. In Section 3.2, we prove some properties that are needed when answering path queries using the technique of tree extraction. Then, we describe our data structures in Section 3.3 to Section 3.5. Finally, Section 3.6 concludes this chapter with a summary and some open problems.

3.2 Applying Tree Extraction to Path Queries

Now we apply the technique of tree extraction, which has been described in Section 2.5.3, to path queries. Let T be the given input tree on n nodes, each having a weight drawn from $[1..\sigma]$. For any integral range $[a..b] \subseteq [1..\sigma]$, we define $R_{a,b}$ to be the set of nodes in T that have a weight in $[a..b]$. We denote by $F_{a,b}$ the $R_{a,b}$-extraction of T, which is a forest containing exactly $|R_{a,b}|$ nodes. Note that the nodes in $F_{a,b}$ are not weighted, though they correspond to weighted nodes in T. Thus, $F_{1,\sigma}$ has the same structure as T, but the nodes in $F_{1,\sigma}$ are not weighted. An example of constructing $F_{1,1}$ and $F_{3,4}$ for an weighted ordinal tree with $n = 10$ and $\sigma = 4$ is illustrated in Figure 3.1.

We define more depth and ancestor operators in terms of weights. For any ordinal forest F_X that is extracted from T, and any node u_X in F_X, we define $dep_{a,b}(F_X, u_X)$,

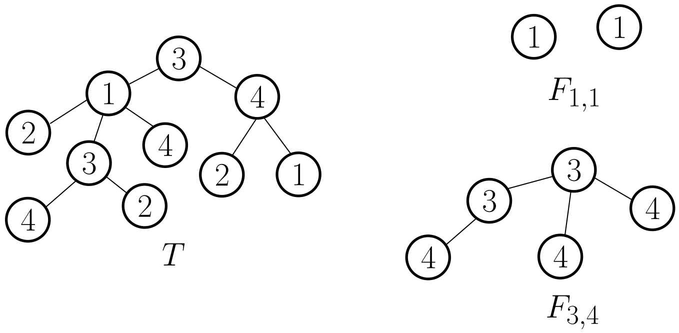


Figure 3.1: An example with $n = 10$ nodes and $\sigma = 4$. The integer on each node of $F_{1,1}$ and $F_{3,4}$ is the weight of its corresponding node in T. We still write down these weights to help readers locate the original nodes.

or the $[a..b]$-*depth* of u_X in F_X, to be the number of ancestors of u_X that correspond to a node in $R_{a,b}$. Likewise, we define $anc_{a,b}(F_X, u_X)$ to be the lowest ancestor of u_X that corresponds to a node in $R_{a,b}$. If no such ancestor exists, then $anc_{a,b}(F_X, u_X)$ is defined to be `dummy`. We further apply these definitions to T. That is, $dep_{a,b}(T, u)$ is equivalent to $dep_{a,b}(F_{1,\sigma}, u_{R_{1,\sigma}})$, and $anc_{a,b}(T, u)$ is equivalent to the node in T that corresponds to $anc_{a,b}(F_{1,\sigma}, u_{R_{1,\sigma}})$, where $u_{R_{1,\sigma}}$ is the node in $F_{1,\sigma}$ that corresponds to u.

For any nodes u and v in T, let $P_{u,v}$ denote the set of nodes on the path from u to v. If u and v are the same node, then $P_{u,v}$ contains this node only. For any node u in T and its ancestor t, we define $A_{u,t}$ to be the set of nodes on the path from u to t, excluding the top node t. Thus, $A_{u,u}$ is a valid but empty set. It is clear that, for any nodes u and v in T, $P_{u,v}$ is the disjoint union of $A_{u,t}$, $A_{v,t}$ and $\{t\}$, where t is the lowest common ancestor of u and v.

Consider how to compute the intersection of $R_{a,b}$ and $P_{u,v}$. Provided that t is the lowest common ancestor of u and v, we have

$$\begin{aligned} R_{a,b} \cap P_{u,v} &= R_{a,b} \cap (A_{u,t} \cup A_{v,t} \cup \{t\}) \\ &= (R_{a,b} \cap A_{u,t}) \cup (R_{a,b} \cap A_{v,t}) \cup (R_{a,b} \cap \{t\}); \qquad (3.1) \end{aligned}$$

and its cardinality is

$$\begin{aligned} |R_{a,b} \cap P_{u,v}| &= |(R_{a,b} \cap A_{u,t}) \cup (R_{a,b} \cap A_{v,t}) \cup (R_{a,b} \cap \{t\})| \\ &= |R_{a,b} \cap A_{u,t}| + |R_{a,b} \cap A_{v,t}| + |R_{a,b} \cap \{t\}| \\ &= dep_{a,b}(T, u) - dep_{a,b}(T, t) + dep_{a,b}(T, v) - dep_{a,b}(T, t) + \mathbf{1}_{R_{a,b}}(t) \\ &= dep_{a,b}(T, u) + dep_{a,b}(T, v) - 2 \cdot dep_{a,b}(T, t) + \mathbf{1}_{R_{a,b}}(t), \qquad (3.2) \end{aligned}$$

where $\mathbf{1}_{R_{a,b}}(t)$ is equal to 1 if the weight of t is between a and b, or equal to 0 if not. In order to compute the cardinality efficiently, we need a fast way to compute $dep_{a,b}(T, u)$'s. A naive solution is to store the value of $dep_{1,b}(T, u)$ for any $1 \le b \le \sigma$ and any node u in T. This method requires $O(\sigma)$ space for each node in T so that the overall space cost would be $O(n\sigma)$. To save space, we prove the follow lemmas with respect to ranges of weights.

Lemma 3.2.1. *For any node $u \in V(T)$, and any two nested ranges $[a..b] \subseteq [a'..b'] \subseteq [1..\sigma]$, the following statements hold:*

(a) $dep_{a,b}(T, u) = dep_{a,b}(F_{a',b'}, v_{R_{a',b'}})$,

(b) $anc_{a,b}(T, u)$ corresponds to node $anc_{a,b}(F_{a',b'}, v_{R_{a',b'}})$ in $F_{a',b'}$,

where $v_{R_{a',b'}}$ is the node in $F_{a',b'}$ that corresponds to $v = anc_{a',b'}(T, u)$.

Proof. Let X and Y denote $R_{a,b}$ and $R_{a',b'}$, respectively; then $F_{a',b'} = F_Y$ and $v_{R_{a',b'}} = v_Y$. We also let X_Y denote the set of nodes in F_Y that correspond to the nodes in X. Thus $dep_{a,b}(F_{a',b'}, v_{R_{a',b'}}) = dep_{X_Y}(F_Y, v_Y)$ and $anc_{a,b}(F_{a',b'}, v_{R_{a',b'}}) = anc_{X_Y}(F_Y, v_Y)$.

For part (a), we first prove that $dep_{a,b}(F, u) = dep_{a,b}(F, v)$. Since $X \subseteq Y$ and v is the lowest ancestor of u that belongs to Y, any ancestor of u that belongs to X must also be an ancestor of v. Thus we have $dep_{a,b}(F, u) \leq dep_{a,b}(F, v)$. Remember that v is an ancestor of u; we therefore have $dep_{a,b}(F, u) = dep_{a,b}(F, v)$.

In addition, observe that deleting a node not in X does not change the $[a..b]$-depth of any other node. By induction on the number of the deletion operations, we can show that, for any node $v \in Y$, $dep_{a,b}(F, v)$ is equal to $dep_{X_Y}(F_Y, v_Y)$. Combining these two equations, we conclude that $dep_{a,b}(F, u) = dep_{X_Y}(F_Y, v_Y)$.

For part (b), we define that $t = anc_{a,b}(F, u)$ and $z_Y = anc_{X_Y}(F_Y, v_Y)$. Our claim clearly holds if $t = \texttt{dummy}$. Otherwise, t must be an ancestor of v in T, and t_Y must be an ancestor of v_Y in F_Y. Since $t \in X$ and $t_Y \in X_Y$, $z_Y = anc_{X_Y}(F_Y, v_Y)$ is not a dummy node. In addition, t_Y must be an ancestor of z_Y in F_Y, and t must be an ancestor of z in T. On the other hand, the depth of z in T cannot be larger than the depth of t, since t is the lowest ancestor of u that belongs to X. We thus conclude that t is equal to z and corresponds to z_Y in F_Y. □

Lemma 3.2.2. *For any node u in T, and any range $[a..b] \subseteq [1..\sigma]$, the following equation holds:*

$$dep_{a,b}(T, u) = dep(F_{a,b}, x_{R_{a,b}}),$$

where $x_{R_{a,b}}$ is the node in $F_{a,b}$ that corresponds to $x = anc_{a,b}(T, u)$.

Proof. By part (a) of Lemma 3.2.1, setting $[a..b] = [a'..b']$. □

Lemma 3.2.3. *For any range $[a..b] \subseteq [1..\sigma]$, the preorder traversal sequence of $F_{a,b}$ corresponds to the sequence of nodes obtained by removing the nodes that are not in $R_{a,b}$ from the preorder traversal sequence of T.*

Proof. By Lemma 2.5.4, setting $X = R_{a,b}$ and $X' = V(T)$. □

3.3 Data Structures under the Pointer Machine Model

In this section, we present our data structure under the pointer machine model. Our basic idea is to build a conceptual range tree on $[1..\sigma]$: Starting with $[1..\sigma]$, we keep splitting each range into two child ranges that differ by at most 1 in length. Formally, provided that $a < b$, the range $[a..b]$ will be split evenly into child ranges $[a_1..b_1]$ and $[a_2..b_2]$, where $a_1 = a$, $b_1 = \lfloor (a+b)/2 \rfloor$, $a_2 = b_1 + 1$ and $b_2 = b$. This procedure stops when $[1..\sigma]$ has been split into σ leaf ranges of length 1, each of which corresponds to a single value in $[1..\sigma]$.

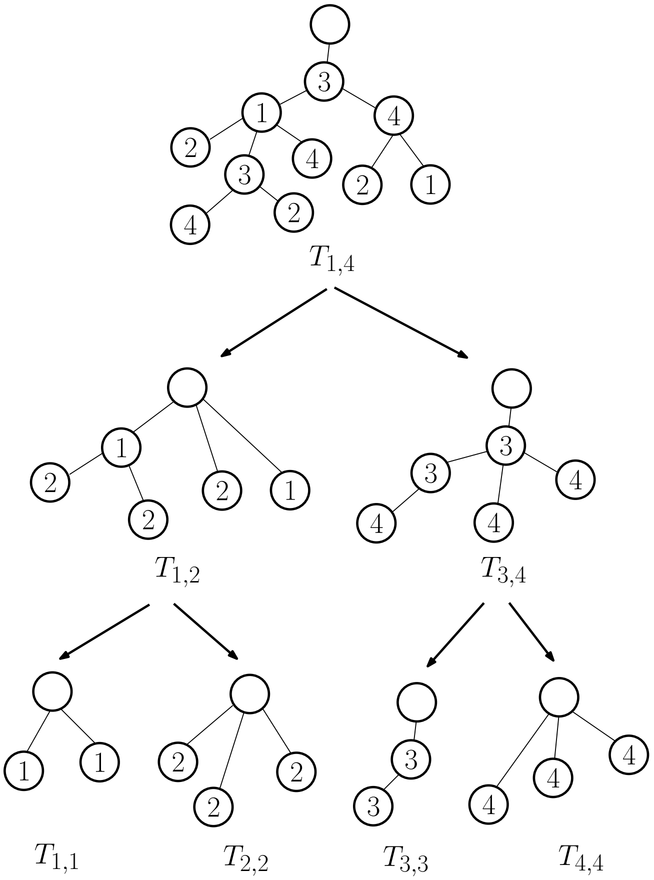


Figure 3.2: An example with $n = 10$ nodes and $\sigma = 4$. The integer on each non-dummy node is the weight of its corresponding node in T. We preserve these weights to help readers locate the corresponding nodes.

For each range $[a..b]$ in the range tree, we construct and store $F_{a,b}$ explicitly. For the sake of convenience, we explicitly add a dummy root to each $F_{a,b}$, and denote by $T_{a,b}$ the new ordinal tree rooted at the dummy node. The dummy root is omitted for the preorder traversal sequence of $T_{a,b}$ and the depths of the nodes in $T_{a,b}$, so $F_{a,b}$ and $T_{a,b}$ can be used interchangeably for the depth and ancestor operators, and Lemmas 3.2.1 to 3.2.3 listed in Section 3.2 still hold for $T_{a,b}$'s. Figure 3.2 gives an example of constructing all the $T_{a,b}$'s for the ordinal tree T in Figure 3.1.

On each node x in $T_{a,b}$ we store the depth of x, and a pointer to the node in T that corresponds to x. For each non-leaf range $[a..b]$, let $[a_1..b_1]$ and $[a_2..b_2]$ be the child ranges of $[a..b]$. For each node x in $T_{a,b}$, we pre-compute and store the value of $dep(T_{a,b}, x)$, and a pointer to the node in T_{a_i,b_i} that corresponds to $anc_{a_i,b_i}(T_{a,b}, x)$ for $i = 1, 2$, which is denoted by $pointer_{a,b}(x, i)$.

We now show how to support path queries using the above data structure.

Algorithm 1 The algorithm for path counting and reporting queries.

```
procedure SEARCH($[a..b], x, y, z, [p..q]$)
                ▷ $x$, $y$ and $z$ correspond to $anc_{a,b}(T, u)$, $anc_{a,b}(T, v)$ and $anc_{a,b}(T, t)$, respectively.
    if $[a..b] \subseteq [p..q]$ then
        if the given query is a path reporting query then
            report all nodes on the path from $x$ to $y$ except $z$;
            report node $t$ if its weight is in $[a..b]$;
        end if
        return $dep(T_{a,b}, x) + dep(T_{a,b}, y) - 2 \cdot dep(T_{a,b}, z) + \mathbf{1}_{R_{a,b}}(t)$;
                ▷ $|P_{u,v} \cap R_{a,b}|$, by Lemma 3.2.2 and Equation 3.2.
    end if
    Let $[a_1..b_1]$ and $[a_2..b_2]$ be the child ranges of $[a..b]$;
    $\delta_i \leftarrow pointer_{a,b}(\delta, i)$ for $\delta = \{x, y, z\}$ and $i = 1, 2$;      ▷ By part (b) of Lemma 3.2.1.
    $count \leftarrow 0$;
    for $i \leftarrow 1, 2$ do
        if $[a_i..b_i] \cap [p..q] \neq \phi$ then
            $count \leftarrow count+$ SEARCH($[a_i..b_i], x_i, y_i, z_i, [p..q]$);
        end if
    end for
    return $count$;
end procedure
```

Lemma 3.3.1. *The data structure described in this section supports path counting queries in $O(\lg \sigma)$ time, and path reporting queries in $O(\lg \sigma + occ)$ time, where occ is the output size of the query.*

Proof. Let u and v be the endpoints of the query path, and let $[p..q]$ be the query range. We first consider how to compute the cardinality of $P_{u,v} \cap R_{p,q}$ for path counting queries. By properties of range trees, each query range $[p..q] \subseteq [1..\sigma]$ can be represented as the union of m disjoint ranges in the range tree, say $[a_1..b_1], \ldots, [a_m..b_m]$, where $m = O(\lg \sigma)$. Because $P_{u,v} \cap R_{p,q} = \bigcup_{1 \leq i \leq m}(P_{u,v} \cap R_{a_i,b_i})$, we need only to compute the cardinality of $P_{u,v} \cap R_{a_i,b_i}$ efficiently, for $1 \leq i \leq m$.

The algorithm is shown in Algorithm 1, where t is the lowest common ancestor of u and v. Provided that $[a..b]$ is a range in the conceptual range tree, x, y and z are nodes in $T_{a,b}$ that correspond to $anc_{a,b}(T,u)$, $anc_{a,b}(T,v)$ and $anc_{a,b}(T,t)$, the procedure $\textsc{Search}([a..b],x,y,z,[p..q])$ returns the cardinality of $P_{u,v} \cap R_{a,b} \cap R_{p,q}$, and reports the nodes in the intersection if the given query is a path reporting query. Please note that the node in T that corresponds to z is not necessarily on the path from u to v. To compute $P_{u,v} \cap R_{p,q}$, we need only to call $\textsc{Search}([1..\sigma],u,v,t,[p..q])$.

Our algorithm accesses the range tree from top to bottom, ending at the ranges completely included in $[p..q]$. Line 8 computes $|P_{u,v} \cap R_{a,b}|$ in $O(1)$ time when $[a..b] \subseteq [p..q]$. In line 12, the algorithm iteratively computes the lowest ancestors of x, y, z for child ranges. Finally, in line 16, the algorithm recurses on child ranges that intersect with $[p..q]$.

For path reporting queries, as shown in lines 4 to 7, our algorithm traverses from x to y, reporting all the nodes on this path except z. Note that for each node being reported, we report the node in T that corresponds to it by following the pointer saved in this node. Finally, we report node t if its weight is in this range. These nodes correspond to the ones in $P_{u,v} \cap R_{a,b}$.

Now we analyze the time cost of Algorithm 1. First of all, the LCA query performed at the very beginning can be supported in constant time and linear space (see [14] for a simple implementation). We thus need only to consider our range tree. For path counting queries, Algorithm 1 accesses $O(\lg \sigma)$ ranges, and it spends constant time on each range. For path reporting queries, Algorithm 1 uses $O(1)$ additional time to report each occurrence. Hence, the query time of path counting is $O(\lg \sigma)$, and the query time of path reporting is $O(\lg \sigma + occ)$, where occ is the output size. □

Lemma 3.3.2. *The data structure described in this section supports path median and selection queries in $O(\lg \sigma)$ time.*

Proof. It suffices to consider path selection queries only. Let u and v be the nodes given in the query, and let k be the rank of the weight to select. Our algorithm for path median and selection queries is shown in Algorithm 2, where t is the lowest common ancestor of u and v. Provided that $[a..b]$ is a range in the conceptual range tree, x, y and z are nodes in $T_{a,b}$ that correspond to $anc_{a,b}(T,u)$, $anc_{a,b}(T,v)$ and $anc_{a,b}(T,t)$, the procedure $\textsc{Select}([a..b],x,y,z,s)$ returns the s-th smallest weight among the weights of the nodes in $R_{a,b} \cap P_{u,v}$. To compute the given query, we only need to call $\textsc{Select}([1..\sigma],u,v,t,k)$.

Now let us analyze the procedure $\textsc{Select}$. If $a = b$, the weight to return must be a. Otherwise, let $[a_1..b_1]$ and $[a_2..b_2]$ be the child ranges of $[a..b]$, where $b_1 < a_2$. The algorithm computes $count = |P_{u,v} \cap R_{a_1,b_1}|$ in line 9, and compares it with s. If s is not larger than $count$, then the algorithm recurses on $[a_1..b_1]$ in line 12; otherwise the algorithm deducts

Algorithm 2 The algorithm for path median and selection queries.

```
 1: procedure SELECT($[a..b], x, y, z, s$)
 2:                      ▷ $x$, $y$ and $z$ correspond to $anc_{a,b}(T,u)$, $anc_{a,b}(T,v)$ and $anc_{a,b}(T,t)$, respectively.
 3:     if $a = b$ then
 4:         return $a$;
 5:     end if
 6:     Let $[a_1..b_1]$ and $[a_2..b_2]$ be the child ranges of $[a..b]$, where $b_1 < a_2$;
 7:     $\delta_i \leftarrow pointer_{a,b}(\delta, i)$ for $\delta = \{x, y, z\}$ and $i = 1, 2$;
 8:                                                      ▷ By part (b) of Lemma 3.2.1.
 9:     $count \leftarrow dep(T_{a_1,b_1}, x_1) + dep(T_{a_1,b_1}, y_1) - 2 \cdot dep(T_{a_1,b_1}, z_1) + \mathbf{1}_{R_{a_1,b_1}}(t)$;
10:                                  ▷ $|P_{u,v} \cap R_{a_1,b_1}|$, by Lemma 3.2.2 and Equation 3.2.
11:     if $s \le count$ then
12:         return SELECT($[a_1..b_1], x_1, y_1, z_1, s$);
13:     else
14:         return SELECT($[a_2..b_2], x_2, y_2, z_2, s - count$);
15:     end if
16: end procedure
```

count from s and recurses on $[a_2..b_2]$ in line 14. Hence Algorithm 2 traverses the range tree from top to bottom, visiting at most $O(\lg \sigma)$ ranges, and finally ending at some range of length 1. It also spends only constant time on each visited range. This algorithm thus uses $O(\lg \sigma)$ time to answer a path median or selection query. □

With Lemmas 3.3.1 and 3.3.2, we can present our result on supporting path queries under the pointer machine model.

Theorem 3.3.3. *Under the pointer machine model, a tree on n weighted nodes can be represented in $O(n \lg \sigma)$ space to support path counting, median and selection queries in $O(\lg \sigma)$ time, and path reporting queries in $O(\lg \sigma + occ)$ time, where the weights are drawn from $[1..\sigma]$, and occ is the output size of the path reporting query.*

Proof. The claim of query time follows from Lemma 3.3.1 and Lemma 3.3.2. It suffices to analyze the space cost of our data structure. For a range $[a..b]$ in the range tree, our data structure uses $O(|R_{a,b}|)$ space to store $T_{a,b}$, the depths of nodes, and the pointers to T and child ranges. Thus the adjunct data structures constructed for each level of the range tree occupy $O(n)$ space in total, and the overall space cost of our data structure is $O(n \lg \sigma)$. □

3.4 Word RAM Data Structures with Reduced Space Cost

In this section we show how to reduce the space cost of the data structure presented in Section 3.3. We adopt the standard word RAM model of computation with word size $w = \Omega(\lg n)$. For path counting, median and selection queries, we achieve $O(\lg \sigma)$ query time with $O(n)$ words of space. For path reporting queries, we require $O(\lg \sigma + occ \lg \sigma)$ query time with $O(n)$ words of space. Note that the data structures for path counting, reporting, median, and selection queries are worse than the succinct data structures presented in Section 3.5. We keep these results in this chapter because they may help readers understand our succinct data structures.

Our starting point for space optimization is the succinct representation of labeled ordinal trees. It is noteworthy that weights and labels both mean integers in this thesis. However, for the sake of clarity and consistency, we resist to use these two terms interchangeably. Integers assigned to nodes of the given input tree are referred to as weights. To support these path queries over the input tree, we construct ordinal trees whose nodes are assigned integers that are computed from weights in certain ways (like $T_{a,b}$'s in this section and T_ℓ's in Section 3.5). These integers are referred to as labels.

As shown in Lemma 2.5.3, there exists a data structure that encodes a tree on n labeled nodes in $O(n)$ bits of space when the number of distinct labels is constant, and supports a set of basic operations in constant time. To make use of this representation, nodes are denoted by the preorder ranks throughout Section 3.4. We now present our linear space data structure.

Theorem 3.4.1. *Under the word RAM model with word size $w = \Omega(\lg n)$, a tree on n weighted nodes can be represented in $O(n)$ words of space to support path counting, median and selection queries in $O(\lg \sigma)$ time, and path reporting queries in $O(\lg \sigma + occ \lg \sigma)$ time, where the weights are drawn from $[1..\sigma]$, and occ is the output size of the path reporting query.*

Proof. For our new data structure, we still build a conceptual range tree as in Section 3.3. Unlike the previous data structure, we store only the succinct representation of $T_{a,b}$ for each range $[a..b]$ in the range tree. We assign a label to each node in $T_{a,b}$ if range $[a..b]$ is not a leaf range. Let $[a_1..b_1]$ and $[a_2..b_2]$ be child ranges of $[a..b]$, where $b_1 < a_2$. We assign label 0 to the dummy root in $T_{a,b}$, though this dummy root is not taken into account for the preorder traversal sequence of $T_{a,b}$. For each non-dummy node x in $T_{a,b}$, we assign it label i if x corresponds to a node in R_{a_i,b_i} for $i = 1$ or 2. By Lemma 2.5.3, the succinct representation for range $[a..b]$ occupies $O(|R_{a,b}|)$ bits. Thus, the space cost of our new data structure is $O(n \lg \sigma / w) = O(n)$ words.

Now consider how to answer path queries. Note that the nodes in the succinct representation can be identified by their ranks in preorder. Let $[a..b]$ be a non-leaf range, and let $[a_1..b_1]$ and $[a_2..b_2]$ be the child ranges of $[a..b]$, where $b_1 < a_2$. Suppose that node x is in $T_{a,b}$, node x_i is in T_{a_i,b_i} for $i = 1$ or 2, and x and x_i correspond to the same node in T. We show that, once the preorder rank of x in $T_{a,b}$ is known, the preorder rank of x_i in T_{a_i,b_i} can be computed in constant time, and vice versa. By the construction of our linear space data structure, T_{a_i,b_i} contains the nodes that correspond to the nodes in $T_{a,b}$ that have a label i. By Corollary 3.2.3, these nodes have the same relative positions in the preorder traversal sequences of T_{a_i,b_i} and $T_{a,b}$. We thus have that

$$\texttt{pre\_rank}_i(T_{a,b}, x) = \texttt{pre\_rank}(T_{a_i,b_i}, x_i),$$

$$x_i = \texttt{pre\_select}(T_{a_i,b_i}, \texttt{pre\_rank}_i(T_{a,b}, x)), \tag{3.3}$$

$$x = \texttt{pre\_select}_i(T_{a,b}, \texttt{pre\_rank}(T_{a_i,b_i}, x_i)). \tag{3.4}$$

By applying these formulas, it takes constant time to convert the preorder ranks of two corresponding nodes between two adjacent levels in the range tree.

Since `depth` is supported over the succinct representation of $T_{a,b}$ for each range $[a..b]$, we need only to consider how to compute $\mathit{pointer}_{a,b}(x, i)$ for child ranges (Line 12 in Algorithm 1 and line 7 in Algorithm 2). For $\delta = \{x, y, z\}$ and $i = 1, 2$, we can compute the node in $T_{a,b}$ that corresponds to $\mathit{pointer}_{a,b}(\delta, i)$ by $\texttt{lowest\_anc}_i(T_{a,b}, \delta)$, and convert it to the actual $\mathit{pointer}_{a,b}(\delta, i)$ using Equation 3.3. If δ has no ancestor with label i in $T_{a,b}$, then $\mathit{pointer}_{a,b}(\delta, i)$ would be the dummy root of T_{a_i,b_i}.

It is more complicated to deal with path reporting queries. Unlike the data structure described in Section 3.3, given a node x in some $T_{a,b}$ that needs to be reported, we cannot directly locate its corresponding node in T by following an appropriate pointer, as we cannot afford to store these pointers. Instead, we find the node in the tree constructed for the parent range of $[a..b]$ that corresponds to x using Equation 3.4, and repeat this process until we reach the root range in the range tree. This procedure takes $O(\lg \sigma)$ time for each node to be reported.

To analyze the query time, we observe that, for path counting, median and selection queries, our linear space data structure still uses constant time on each visited range. Hence, these queries can be answered in $O(\lg \sigma)$ time. For path reporting queries, this data structure requires $O(\lg \sigma)$ additional time for each node to report. Thus, path reporting queries can be answered in $O(\lg \sigma + \mathit{occ} \lg \sigma)$ time, where occ is the output size. □

3.5 Succinct Data Structures with Improved Query Time

In this section, we present the main results in this chapter. The basic idea is to increase the branching factor of the conceptual range tree, pack all $F_{a,b}$'s at the same level into a single labeled ordinal tree, and store it using a succinct representation. However, the representation in Lemma 2.5.3 is succinct only if $\sigma = o(\lg \lg n)$. We have to develop another succinct representation of labeled ordinal trees in Section 3.5.1. Then, using this representation and other auxiliary data structures, we design the succinct data structures for path queries in Sections 3.5.2 and 3.5.3. As in the previous section, we assume that nodes are denoted by their preorder ranks in Section 3.5.

3.5.1 Succinct Ordinal Trees over an Alphabet of Size $O(\lg^{\epsilon} n)$

We describe a succinct data structure to encode an ordinal tree T on n labeled nodes, in which the labels are drawn from $[1..\sigma]$, where $\sigma = O(\lg^{\epsilon} n)$ for some constant $0 < \epsilon < 1$. This succinct representation occupies $n(H_0(PLS_T) + 2) + o(n)$ bits of space to support a set of operations in constant time, where PLS_T is the preorder label sequence of T. Let W_T denote the multiset that consists of the weights of the nodes in T. As the entropy of W_T only depends on the frequencies of the labels, we have $H_0(PLS_T) = H(W_T)$. Thus, the space cost can also be represented as $n(H(W_T) + 2) + o(n)$ bits. However, we use $H_0(PLS_T)$ in this section to facilitate space analysis. The starting points of our succinct representation are Geary et al.'s [54] succinct ordinal tree, and Bose et al.'s [18] data structure for orthogonal range search on a narrow grid.

In the rest of this section, we consider the operations listed below. For the sake of convenience, we call a node x an α-node or an α-ancestor of its descendants if the label of x is α. Also, we assume that nodes x and y are contained in T, α and β are in $[1..\sigma]$, a node precedes itself in preorder, and a node is its own 0-th ancestor.

- $\texttt{pre\_rank}(T, x)$: Return the number of nodes that precede x in preorder;
- $\texttt{pre\_rank}_{\alpha}(T, x)$: Return the number of α-nodes that precede x in preorder;
- $\texttt{pre\_select}(T, i)$: Return the i-th node in preorder;
- $\texttt{pre\_select}_{\alpha}(T, i)$: Return the i-th α-node in preorder;

- $\texttt{pre\_count}_\beta(T, i)$: Return the number of nodes whose preorder ranks are at most i and labels are at most β;
- $\texttt{depth}(T, x)$: Return the number of ancestors of x;
- $\texttt{depth}_\alpha(T, x)$: Return the number of α-ancestors of x;
- $\texttt{LCA}(T, x, y)$: Return the lowest common ancestor of x and y;
- $\texttt{lowest\_anc}_\alpha(T, x)$: Return the lowest α-ancestor of x if such an α-ancestor exists, otherwise return NULL;
- $\texttt{node\_count}_\beta(T, x)$: Return the number of nodes on the path from x to the root of T whose labels are at most β;
- $\texttt{node\_summarize}(T, x, y)$: Given that node y is an ancestor of x, this operation returns σ bits, where the α-th bit is 1 if and only if there exists an α-node on the path from x to y (excluding y), for $1 \leq \alpha \leq \sigma$.

We first compute the mini-micro tree cover of T using Lemma 2.5.1 for $M = \lceil \lg^2 n \rceil$ and $M' = \lceil \lg^\lambda n \rceil$ for some $\max\{\epsilon, \frac{1}{2}\} < \lambda < 1$. It is easy to verify that, with our choice of M and M', the operations in Lemma 2.5.2, which are related to tree nodes, tier-k subtrees and tier-k roots, can still be supported in $O(1)$ time, using $2n + o(n)$ bits of space and $O(n)$ construction time. Let n_1 and n_2 be the numbers of tier-1 roots and tier-2 roots. By Lemma 2.5.1, they are bounded by $O(n/M)$ and $O(n/M')$, respectively. To store the labels, we encode $PLS_T[1..n]$ using Lemma 2.4.4, occupying $nH_0(PLS_T) + o(n)$ bits of space. We defer the analysis of the construction time of PLS_T to the proof of Theorem 3.5.8. These are our main data structures that encode the structure of the tree and the labels of nodes.

To support operations, we design auxiliary data structures that require $o(n)$ bits of additional space and $O(n)$ construction time.

Lemma 3.5.1. *Operations* $\texttt{pre\_rank}$, $\texttt{pre\_select}$, $\texttt{depth}$, $\texttt{LCA}$, $\texttt{pre\_rank}_\alpha$, $\texttt{pre\_select}_\alpha$ *and* $\texttt{pre\_count}_\beta$ *can be supported using* $O(1)$ *query time,* $o(n)$ *bits of additional space and* $o(n)$ *construction time.*

Proof. The first four operations are supported by Lemma 2.5.2 without extra cost. Also, $\texttt{pre\_rank}_\alpha$ and $\texttt{pre\_select}_\alpha$ are naturally supported by Lemma 2.4.4 without extra cost. In addition, the conclusion of Lemma 2.4.4 satisfies the condition of Lemma 2.4.5. Thus $\texttt{pre\_count}_\beta$ can also be supported in $O(1)$ time, using $o(n)$ bits of additional space and $o(n)$ construction time. □

Before describing the support for operations $\texttt{depth}_\alpha$, $\texttt{lowest\_anc}_\alpha$, $\texttt{node\_count}_\beta$ and `node_summarize`, we first define four auxiliary operations related to tier-1 and tier-2 roots. In the definitions below, x is a tier-1 root, and y is a tier-2 root.

- $\texttt{mini\_depth}(T, x)$: Return the number of tier-1 roots on the path from x to the root of T;
- $\texttt{mini\_anc}(T, x, i)$: Return the $(i+1)$-st tier-1 root on the path from x to the root of T, provided that $0 \leq i < \texttt{mini\_depth}(T, x)$;
- $\texttt{micro\_depth}(T, y)$: Return the number of tier-2 roots on the path from y to the root of T;
- $\texttt{micro\_anc}(T, y, i)$: Return the $(i+1)$-st tier-2 root on the path from y to the root of T, provided that $0 \leq i < \texttt{micro\_depth}(T, y)$.

Lemma 3.5.2. `mini_depth`*,* `mini_anc`*,* `micro_depth`*, and* `micro_anc` *can be supported using $O(1)$ query time, $o(n)$ bits of additional space and $O(n)$ construction time.*

Proof. We only show how to support the first two operations. The other operations can be supported in the same way. We compute tree T' by deleting all the nodes other than the tier-1 roots from T, which uses $O(n)$ time. Clearly T' has n_1 nodes. To perform conversions between the nodes in T' and the tier-1 roots in T, we construct and store a bit vector $B_1[1..n]$ in which $B_1[i] = 1$ if and only if the i-th node in preorder of T is a tier-1 root. By Lemma 2.3.1, B_1 can be encoded using $o(n)$ bits and $O(n)$ construction time to support `rank` and `select` in $O(1)$ time. Note that the i-th node in T' corresponds to the tier-1 root of preorder rank j in T if and only if $B_1[j]$ is the i-th 1-bit in B_1. Thus the conversion can be done in constant time using `rank` and `select` on B_1. Applying the techniques in Lemma 2.5.2, we encode T' in $2n_1 + o(n_1) = o(n)$ bits and $O(n_1) = o(n)$ construction time to support `mini_depth` and `mini_anc` directly. The overall space cost is $o(n)$ additional bits, and the overall construction time is $O(n)$. □

Lemma 3.5.3. $\texttt{depth}_\alpha$*,* $\texttt{lowest\_anc}_\alpha$ *and* $\texttt{node\_count}_\beta$ *can be supported using $O(1)$ query time, $o(n)$ bits of additional space and $o(n)$ construction time.*

Proof. We first show how to compute the encoding of a micro tree, which consists of two parts, i.e., the encoding of the structure and the encoding of the labels. The first part can be directly retrieved using Lemma 2.5.2. The computation for the second part requires several lemmas. By Lemmas 2.5.1 and 2.5.2, nodes in this micro tree are distributed into

$O(1)$ tier-2 preorder segments, and the starting and ending positions of all these segments can be found in constant time. Remember that this micro tree has at most $2M' = o(\log_\sigma n)$ nodes; by Lemma 2.4.4, we can obtain the labels of nodes for each of these segments in $O(1)$ time. Concatenating all these labels in preorder, we obtain the encoding of the labels for the entire micro tree. This can be done using a constant number of bitwise operations, since all these labels can be stored in $2M' \times \lceil \lg \sigma \rceil = o(\lg n)$ bits.

Once $\texttt{node\_count}_\beta$ operations are supported, $\texttt{depth}_\alpha$ operations can be computed by the equation $\texttt{depth}_\alpha(T, x) = \texttt{node\_count}_\alpha(T, x) - \texttt{node\_count}_{\alpha-1}(T, x)$. Thus we only show how to support $\texttt{node\_count}_\beta$. $\texttt{lowest\_anc}_\alpha$ can be supported in a similar way.

To support $\texttt{node\_count}_\beta$ operations, we store the following auxiliary data structures:

- A two-dimensional array $A[1..n_1, 1..\sigma]$, in which $A[i, \beta]$ stores the number of nodes on the path from r_i^1 to the root of T whose labels are at most β.

- A two-dimensional array $B[1..n_2, 1..\sigma]$, in which $B[i, \beta]$ stores the number of nodes on the path from r_i^2 to and excluding the root of the mini-tree containing r_i^2 whose labels are at most β.

- A table C that stores for every possible labeled tree p on at most $2M'$ nodes whose labels are drawn from $[1..\sigma]$, every integer i in $[1..2M']$ and every integer β in $[1..\sigma]$, the number of nodes on the path from the i-th node in preorder of p to and excluding the root of p whose labels are at most β.

Now we analyze the space cost of these auxiliary data structures. A occupies $n_1 \times \sigma \times O(\lg n) = O(n/\lg^{1-\epsilon} n) = o(n)$ bits, since $n_1 = O(n/\lg^2 n)$, and each entry can be stored in $O(\lg n)$ bits. B occupies $n_2 \times \sigma \times O(\lg\lg n) = O(n \lg\lg n/\lg^{\lambda-\epsilon} n) = o(n)$ bits, since $n_2 = O(n/\lg^\lambda n)$, and each entry can be stored in $O(\lg\lg n)$ bits. Now let us consider the table C. By Lemma 2.5.1, a micro-tree has no more than $2M'$ nodes. Thus C has at most $2^{4M'} \times \sigma^{2M'} \times 2M' \times \sigma = 2^{2\lceil \lg^\lambda n \rceil (\lg \sigma + 2)} \times 2\lceil \lg^\lambda n \rceil \times \sigma = O(n^{1-\delta})$ entries for some constant $\delta > 0$, each occupying $O(\lg\lg n)$ bits. Therefore, A, B and C occupy $o(n)$ additional bits in total.

The analysis of the construction time is similar to the proof of Lemma 2.4.5. First we precompute the lookup table C by enumeration. Then we fill the array B row by row. For $i = 1, 2, \ldots, n_2$, we set $B[i, \beta] = 0$ if r_i^2 is also a tier-1 root. Otherwise, we compute $r_a^2 = \texttt{micro\_anc}(T, r_i^2, 1)$, and set $B[i, \beta]$ to be $B[a, \beta]$ plus the number of nodes on the path from r_i^2 to and excluding r_a^2 whose labels are at most β, where the latter value can be determined using the lookup table C in $O(1)$ time per entry. Thus the array B can be computed in $O(n_2\sigma) = o(n)$ construction time. The computation of A is similar. The overall construction time is thus $o(n)$.

Let r_c^1 and r_d^2 be the root nodes of the mini-tree and the micro-tree containing x, respectively. By Lemma 2.5.2, they can be found in constant time. To answer operation $\texttt{node\_count}_\beta(T, x)$, we split the path from x to the root of T into three parts: the path from x to and excluding r_d^2, the path from r_d^2 to and excluding r_c^1, and the path from r_c^1 to the root of T. The numbers on the second and the third parts can be accumulated by accessing $B[d, \beta]$ and $A[c, \beta]$, respectively. For the first part, we consider only the case in which x is not a tier-2 root. By Lemma 2.5.2, we are able to compute in constant time the relative preorder rank of x with respect to the micro-tree containing x. Thus we can obtain the count on the first part by a table lookup on C. □

Lemma 3.5.4. `node_summarize` *can be supported using* $O(1)$ *query time,* $o(n)$ *bits of additional space and* $o(n)$ *construction time.*

Proof. To support `node_summarize`, we construct the following auxiliary data structures:

- A two-dimensional array $D[1..n_1, 0..\lceil \lg n \rceil]$, in which $D[i, j]$ stores a bit vector of length σ whose α-th bit is 1 if and only if there exists an α-node on the path from r_i^1 to $\texttt{mini\_anc}(T, r_i^1, 2^j)$.

- A two-dimensional array $E[1..n_2, 0..3\lceil \lg \lg n \rceil]$, in which $E[i, j]$ stores a bit vector of length σ whose α-th bit is 1 if and only if there exists an α-node on the path from r_i^2 to $\texttt{micro\_anc}(T, r_i^2, 2^j)$.

- A table F that stores for every possible labeled tree p on at most $2M'$ nodes whose labels are drawn from $[1..\sigma]$ and for every integer i, j in $[1..2M']$, a bit vector of length σ whose α-th bit is 1 if and only if there exists an α-node on the path from the i-th node of p in preorder to the j-th node in preorder of p.

- All these paths do not include the top nodes.

Now we analyze the space cost. D occupies $n_1 \times (\lceil \lg n \rceil + 1) \times \sigma = O(n / \lg^{1-\epsilon} n) = o(n)$ bits. E occupies $n_2 \times (3\lceil \lg \lg n \rceil + 1) \times \sigma = O(n \lg \lg n / \lg^{\lambda - \epsilon} n) = o(n)$ bits. As in the analysis of C, F has $O(n^{1-\delta})$ entries for some constant $\delta > 0$, each occupying σ bits. Therefore, D, E and F occupy $o(n)$ bits in total. The analysis of the construction time is similar to the proofs of Lemmas 2.4.5 and 3.5.3.

Let r_a^1 and r_b^2 be the roots of the mini-tree and the micro-tree containing x, respectively. Also, let r_c^1 and r_d^2 be the roots of the mini-tree and the micro-tree containing y, respectively.

Suppose that $a \neq c$ (the case in which $a = c$ can be handled similarly). We define the following two nodes:

$$\begin{aligned} r_e^1 &= \texttt{mini\_anc}(T, r_a^1, \texttt{mini\_depth}(T, r_a^1) - \texttt{mini\_depth}(T, r_c^1) - 1), \\ r_f^2 &= \texttt{micro\_anc}(T, r_e^1, \texttt{micro\_depth}(T, r_e^1) - \texttt{micro\_depth}(T, r_d^2) - 1). \end{aligned}$$

To answer $\texttt{node\_summarize}(T, x, y)$, we split the path from x to y into $x \sim r_b^2 \sim r_a^1 \sim r_e^1 \sim r_f^2 \sim y$. We compute for each part a bit vector of length σ whose α-th bit is 1 if and only if there exists an α-node on this part. The answer to the $\texttt{node\_summarize}$ operation is the result of a bitwise OR operation on these bit vectors.

The first part and the last part are paths in the same micro-tree. Thus their bit vectors can be computed by table lookups on F. We only show how to compute the bit vector of the third part, i.e., the path $r_a^1 \sim r_e^1$; the bit vectors of the second and the fourth part can be computed in a similar way. Setting $i = \texttt{mini\_depth}(T, r_a^1) - \texttt{mini\_depth}(T, r_e^1)$, $j = \lfloor \lg i \rfloor$, tier-1 root $r_g^1 = \texttt{mini\_anc}(T, r_a^1, i - 2^j)$, the path from r_a^1 to r_e^1 can be represented as the union of the path from r_a^1 to $\texttt{mini\_anc}(T, r_a^1, 2^j)$ and path from r_g^1 to $\texttt{mini\_anc}(T, r_g^1, 2^j)$. Therefore the vector of this part is equal to the bitwise OR of $D[a, j]$ and $D[g, j]$. □

To analyze the space cost of our data structure, we observe that the main cost is due to the sequence PLS_T and Lemma 2.5.2, which occupy $n(H_0(PLS_T) + 2) + o(n)$ bits in total. The other auxiliary data structures occupy $o(n)$ bits of space only. We thus have the following lemma.

Lemma 3.5.5. *Let T be an ordinal tree on n nodes, each having a label drawn from $[1..\sigma]$, where $\sigma = O(\lg^\epsilon n)$ for some constant $0 < \epsilon < 1$. Then T can be represented in $n(H_0(PLS_T)+2)+o(n)$ bits of space to support the operations listed at the beginning of this section, and the operations described in Lemma 2.5.2 in constant time. This data structure can be constructed in $O(n)$ time plus the preprocessing cost of PLS_T.*

3.5.2 Path Counting and Reporting

We now describe the data structures for general path searching queries. The basic idea is to build a conceptual range tree on $[1..\sigma]$ with the branching factor $f = \lceil \lg^\epsilon n \rceil$ for some constant $0 < \epsilon < 1$. In this range tree, a range $[a..b]$ corresponds to the nodes in T whose weights are in $[a..b]$. These nodes and weights from a to b are said to be contained in the range $[a..b]$. A range $[a..b]$ is said to be empty if $a > b$. It is possible that a non-empty range does not correspond to any node in T. In addition, the length of a range $[a..b]$ is defined to be $b - a + 1$. Empty ranges have non-positive lengths.

We describe this range tree level by level. In the beginning, we have the top level only, which contains the root range $[1..\sigma]$. Starting from the top level, we keep splitting each range in the current lowest level into f child ranges, some of which might be empty. A range $[a..b]$, where $a \leq b$, will be split into f child ranges $[a_1..b_1], [a_2..b_2], \ldots, [a_f..b_f]$ such that, for $a \leq j \leq b$, the nodes that have a weight j are contained in the child range of subscript $\lceil \frac{f(j-a+1)}{b-a+1} \rceil$. After some calculations, we have that

$$a_i = \min\left\{a \leq j \leq b \mid \lceil \frac{f(j-a+1)}{b-a+1} \rceil = i\right\} = \lfloor \frac{(i-1)(b-a+1)}{f} \rfloor + a; \tag{3.5}$$

$$b_i = \max\left\{a \leq j \leq b \mid \lceil \frac{f(j-a+1)}{b-a+1} \rceil = i\right\} = \lfloor \frac{i(b-a+1)}{f} \rfloor + a - 1. \tag{3.6}$$

This procedure stops when all the ranges in the current lowest level have length 1, which form the bottom level of the conceptual range tree.

We list all the levels from top to bottom. The top level is level 1, and the bottom one is level h, where $h = \lceil \log_f \sigma \rceil + 1$ is the height of the conceptual range tree. It is clear that each value in $[1..\sigma]$ occurs in exactly one range at each level, and each leaf range corresponds to a single value in $[1..\sigma]$.

For each level in the range tree other than the bottom one, we create and explicitly store an auxiliary tree. Let T_ℓ denote the auxiliary tree created for level ℓ. To create T_ℓ, we list all the non-empty ranges at level ℓ in increasing order. Let $[a_1..b_1], [a_2..b_2], \cdots, [a_m..b_m]$ be those ranges. We have that $a_1 = 1, b_m = \sigma$, and $b_i = a_{i+1} - 1$ for $i = 1, 2, \ldots, m-1$. Initially, T_ℓ contains only a dummy root r_ℓ. For $i = 1, 2, \cdots, m$, we compute forest F_{a_i,b_i} as described in Section 3.2, and then insert the trees in F_{a_i,b_i} into T_ℓ in the original left-to-right order. That is, for $i = 1, 2, \cdots, m$, from left to right, we append the root node of each tree in F_{a_i,b_i} to the list of children of r_ℓ. Thus, for $\ell = 1, 2, \cdots, h-1$, there exists a one-to-one correspondence between the nodes in T and the non-root nodes in T_ℓ. Note that all the F_{a_i,b_i}'s at level ℓ share the same dummy root. This is slightly different from the description in Section 3.2, where each F_{a_i,b_i} is viewed as an ordinal tree with a non-removable dummy root. Nevertheless, as in Section 3.2, the shared dummy root is not taken into account for the preorder traversal sequence of T_ℓ, or the depths of the nodes in T_ℓ.

We assign labels to the nodes in T_ℓ. The root of T_ℓ is assigned 1. For each node x in T, we denote by x_ℓ the node at level ℓ that corresponds to x. By the construction of the conceptual range tree, the range containing x at level $\ell + 1$ has to be a child range of the range containing x at level ℓ. We assign a label α to x_ℓ if the range at level $\ell + 1$ is the α-th child range of the range at level ℓ. As the result, we obtain an ordinal tree T_ℓ on $n+1$ nodes, each having a label drawn from $[1..f]$. Since $f = \lceil \lg^\epsilon n \rceil$ satisfies the condition of Lemma 3.5.5, we explicitly store T_ℓ using this lemma.

Equations 3.7 and 3.8 capture the relationship between x_ℓ and $x_{\ell+1}$, for each node x in T and $\ell \in [1..h-1]$. Presuming that node x is contained in a range $[a..b]$ at level ℓ, and contained in a range $[a_\gamma..b_\gamma]$ at level $\ell+1$, which is the γ-th child range of $[a..b]$, we have

$$\begin{aligned} rank &= \texttt{pre\_rank}_\gamma(T_\ell, |R_{1,a-1}|) \\ x_{\ell+1} &= \texttt{pre\_select}(T_{\ell+1}, |R_{1,a_\gamma-1}| + \texttt{pre\_rank}_\gamma(T_\ell, x_\ell) - rank) \qquad (3.7) \\ x_\ell &= \texttt{pre\_select}_\gamma(T_\ell, \texttt{pre\_rank}(T_{\ell+1}, x_{\ell+1}) - |R_{1,a_\gamma-1}| + rank) \qquad (3.8) \end{aligned}$$

With these data structures, we can support path counting and reporting queries.

Algorithm 3 The algorithm for path counting and reporting queries on $A_{u,t}$.

```
procedure SEARCH($[a..b], \ell, c, d, x, z, [p..q]$)
    Let $[a_1..b_1], [a_2..b_2], \cdots, [a_f..b_f]$ be the child ranges of $[a..b]$;
    $count \leftarrow 0$, $[a^*..b^*] = [a..b] \cap [p..q]$;
    $\alpha = \min\{1 \le i \le f \mid a^* \le a_i \le b_i\}$, $\beta = \max\{1 \le i \le f \mid a_i \le b_i \le b^*\}$;
    if $\alpha \le \beta$ then
        $count \leftarrow count + \texttt{node\_count}_{\alpha,\beta}(T_\ell, x) - \texttt{node\_count}_{\alpha,\beta}(T_\ell, z)$;
        if the given query is a path reporting query then
            $summary \leftarrow \texttt{node\_summarize}(T_\ell, x, z)$;
            for each $i \in [\alpha..\beta]$ that the $i$-th bit of $summary$ is 1 do
                report all $i$-nodes on the path from $x$ to $z$ (excluding $z$);
            end for
        end if
    end if
    for $\gamma \in \{\alpha - 1, \beta + 1\}$ do
        if $1 \le \gamma \le f$ and $[a_\gamma..b_\gamma] \cap [a^*..b^*] \ne \phi$ then
            $c_\gamma \leftarrow c + \texttt{pre\_count}_{\gamma-1}(T_\ell, d) - \texttt{pre\_count}_{\gamma-1}(T_\ell, c)$;
            $d_\gamma \leftarrow c + \texttt{pre\_count}_{\gamma}(T_\ell, d) - \texttt{pre\_count}_{\gamma}(T_\ell, c)$;
            $\delta_\gamma \leftarrow$ node in $T_{\ell+1}$ that corresponds to $\texttt{lowest\_anc}_\gamma(T_\ell, \delta)$, for $\delta \in \{x, z\}$;
            $count \leftarrow count+$ SEARCH($[a_\gamma..b_\gamma], \ell+1, c_\gamma, d_\gamma, x_\gamma, z_\gamma, [p..q]$);
        end if
    end for
    return $count$;
end procedure
```

Lemma 3.5.6. *Path counting queries can be supported in $O(\lg \sigma / \lg \lg n + 1)$ time.*

Proof. Let $P_{u,v} \cap R_{p,q}$ be the query. $P_{u,v}$ can be partitioned into $A_{u,t}$, $A_{v,t}$ (see Section 3.2 for the definitions of $A_{u,t}$ and $A_{v,t}$) and $\{t\}$, where t is the lowest common ancestor of u and v. It is trivial to compute $\{t\} \cap R_{p,q}$. We only consider how to compute the cardinality of $A_{u,t} \cap R_{p,q}$. The computation of $A_{v,t} \cap R_{p,q}$ is similar.

Our recursive algorithm for path counting and reporting queries is shown in Algorithm 3. Provided that the range $[a..b]$ is at the ℓ-th level of the conceptual range tree, $c = |R_{1,a-1}|, d = |R_{1,b}|$, and x, z are the nodes in T_ℓ that correspond to $anc_{a,b}(T, u)$ and $anc_{a,b}(T, t)$, the procedure $\textsc{Search}([a..b], \ell, c, d, x, z, [p..q])$ returns the cardinality of $A_{u,t} \cap R_{a,b} \cap R_{p,q}$, and reports the nodes in the intersection if the given query is a path reporting query. To compute $A_{u,t} \cap R_{p,q}$ and its cardinality, we only need to call the procedure $\textsc{Search}([1..\sigma], 1, 0, n, u, t, [p..q])$.

Now let us analyze the procedure $\textsc{Search}$. In line 4, we find the first non-empty child range of $[a..b]$ whose left endpoint is at least a^*, and the last non-empty child range whose right endpoint is at most b^*. Let α and β be the subscripts of these two child ranges, respectively. To compute α (the computation of β is similar), let $\alpha' = \lceil \frac{f(a^*-a+1)}{b-a+1} \rceil$ be the subscript of the child range that contains a^*. α is set to be α' if $a^* \leq a_{\alpha'}$, otherwise α is set to be $\alpha' + 1$. If $\alpha \leq \beta$, we accumulate $|A_{u,t} \cap R_{a_\alpha, b_\beta}|$ in line 6, where $\texttt{node\_count}_{\alpha,\beta}(T, x)$ is defined to be $\texttt{node\_count}_{\beta}(T, x) - \texttt{node\_count}_{\alpha-1}(T, x)$. We still need to compute $A_{u,t} \cap (R_{a^*,b^*} - R_{a_\alpha,b_\beta}) = A_{u,t} \cap R_{a^*,b^*} \cap (R_{a_{\alpha-1},b_{\alpha-1}} \cup R_{a_{\beta+1},b_{\beta+1}})$. In the loop starting at line 14, for $\gamma \in \{\alpha - 1, \beta + 1\}$, we check whether $[a_\gamma..b_\gamma] \cap [a^*..b^*] \neq \phi$. If the intersection is not empty, then we compute $A_{u,t} \cap R_{a^*,b^*} \cap R_{a_\gamma,b_\gamma}$ and accumulate its cardinality by a recursive call in line 19. In lines 16 to 18, we adjust the parameters to satisfy the precondition of the procedure $\textsc{Search}$. We increase c and d as shown in lines 16 and 17, because for $i = 1$ to f, the nodes in T whose weights are in $[a_i..b_i]$ correspond to the nodes in T_ℓ whose labels are i and preorder ranks are in $[c + 1..d]$. In line 18, for $\delta = \{x, z\}$, we set δ_γ to be the node in $T_{\ell+1}$ that corresponds to $\texttt{lowest\_anc}_\gamma(T_\ell, \delta)$. To compute it, we call $\texttt{lowest\_anc}_\gamma(T_\ell, \delta)$, and convert the node returned to its corresponding node in $T_{\ell+1}$ using Equation 3.7. If $\texttt{lowest\_anc}_\gamma(T_\ell, \delta)$ returns $\texttt{NULL}$, then δ_γ is set to be the dummy root of $T_{\ell+1}$.

We now analyze the running time of our algorithm. By the construction of the conceptual range tree, any query range $[p..q]$ will be partitioned into $O(h)$ ranges in the range tree. It takes $O(1)$ time to access a range in the range tree, since, by Lemma 2.5.2 and Lemma 3.5.5, all the operations on T_ℓ take $O(1)$ time. Hence, it takes $O(h) = O(\lg \sigma / \lg \lg n + 1)$ time to compute $|A_{u,t} \cap R_{p,q}|$. The computation of $|A_{v,t} \cap R_{p,q}|$ requires the same amount of time, and the original path counting query can thus be supported in $O(\lg \sigma / \lg \lg n + 1)$ time. □

Lemma 3.5.7. *Path reporting queries can be supported in $O((occ + 1)(\lg \sigma / \lg \lg n + 1))$ time, where occ is the output size.*

Proof. For path reporting query $P_{u,v} \cap R_{p,q}$, as in Lemma 3.5.6, we still let t be the lowest common ancestor of u and v, partition $P_{u,v}$ into $A_{u,t}$, $A_{v,t}$ and $\{t\}$, and only consider how to compute $A_{u,t} \cap R_{p,q}$.

We extend the algorithm for path counting queries from Lemma 3.5.6. In lines 7 to 12 of Algorithm 3, we report all nodes on the path $A_{u,t}$ that have a weight in $[a_\alpha..b_\beta]$. In line 8, we compute *summary*, a bit vector of length f, whose i-th bit is one if and only if $A_{u,t} \cap R_{a_i,b_i} \neq \phi$. Since $f = o(\lg n)$, using word operations, we can enumerate each 1-bit whose index is between α and β in constant time per 1-bit. For each 1-bit, assuming its index is i, we find all the nodes in $A_{u,t} \cap R_{a_i,b_i}$ using `lowest_anc`$_i$ and `parent` operations alternately. Here the `parent` operation, which returns the parent of a given node, is supported in Lemma 3.5.5 (see the part for operations that are irrelevant to labels, i.e., Lemma 2.5.2). At this moment, we only know the preorder ranks of the nodes to report in T_ℓ. We have to convert each of them to the preorder rank in T_1 by using Equation 3.8 $\ell - 1$ times per node.

The analysis of query time is similar to Lemma 3.5.6. First it requires $O(h)$ time for the algorithm to traverse the conceptual range tree from top to bottom. Then the algorithm uses $O(h)$ time to report each node in $A_{u,t} \cap R_{p,q}$. Each 1-bit we enumerate in line 9 corresponds to one or more nodes to report, so the cost of enumeration is subsumed by other parts of the algorithm. Therefore, it requires $O(h + |A_{u,t} \cap R_{p,q}| \cdot h) = O((|A_{u,t} \cap R_{p,q}| + 1)(\lg \sigma / \lg\lg n + 1))$ time to compute $A_{u,t} \cap R_{p,q}$. In a similar way, it requires $O((|A_{v,t} \cap R_{p,q}| + 1)(\lg \sigma / \lg\lg n + 1))$ time to compute $A_{v,t} \cap R_{p,q}$. The overall query time for the original path reporting query is thus $O((occ + 1)(\lg \sigma / \lg\lg n + 1))$, where $occ = |P_{u,v} \cap R_{p,q}|$. □

Theorem 3.5.8. *Let T be an ordinal tree on n nodes, each having a weight drawn from $[1..\sigma]$. Under the word RAM model with word size $w = \Omega(\lg n)$, T can be encoded in (a) $n(H(W_T)+2)+o(n)$ bits when $\sigma = O(\lg^\epsilon n)$ for some constant $0 < \epsilon < 1$, or (b) $nH(W_T)+O(n\lg\sigma/\lg\lg n)$ bits otherwise, supporting path counting queries in $O(\lg\sigma/\lg\lg n + 1)$ time, and path reporting queries in $O((occ+1)(\lg\sigma/\lg\lg n+1))$ time, where $H(W_T)$ is the entropy of the multiset of the weights of the nodes in T, and occ is the output size of the path reporting query. It requires $O(n(\lg\sigma/\lg\lg n + 1))$ preprocessing time to encode T.*

Proof. It is sufficient to analyze the space and processing costs, as the query costs have been justified in Lemmas 3.5.6 and 3.5.7. First of all, by the definition of entropy, we have $H_0(PLS_T) = H(W_T)$. When $\sigma = O(\lg^\epsilon n)$ for some constant $0 < \epsilon < 1$, we reset the branching factor f to be σ such that our conceptual range tree has only two levels. It is sufficient to support path searching queries by storing T_1 using Lemma 3.5.5. The space cost is $n(H(W_T) + 2) + o(n)$ bits in this case. Otherwise, our conceptual range tree has $h = \lceil \log_f \sigma \rceil + 1$ levels. For $\ell = 1, 2, \ldots, h-1$, we explicitly store a labeled ordinal tree T_ℓ

using Lemma 3.5.5. Note that the labels are stored in the preorder label sequences using Lemma 2.4.4. As in the analysis of generalized wavelet trees [45], we claim that all these preorder label sequences occupy $nH_0(PLS_T) + (h-1) \cdot o(n) = nH(W_T) + o(n \lg \sigma / \lg \lg n)$ bits in total. The other parts occupy $(h-1) \cdot O(n) = O(n \lg \sigma / \lg \lg n)$ bits. Therefore, the overall space cost is $nH(W_T) + O(n \lg \sigma / \lg \lg n)$ bits.

Now we analyze the preprocessing cost. First it requires $O(n)$ time to apply tree extraction explicitly and compute the structure of each T_ℓ, and thus requires $O(nh)$ time over all the levels of the conceptual range tree. By Lemma 3.5.5, the cost of encoding each T_ℓ is $O(n)$ time, if we ignore the cost of constructing preorder label sequences. As mentioned in the analysis of the space cost, the preorder label sequences of T_ℓ's over all the levels essentially form a generalized wavelet tree of the preorder label sequence of T, which can be constructed in $O(n \lg \sigma / \sqrt{\lg n})$ time [8]. Therefore, the overall preprocessing cost is $O(nh) = O(n(\lg \sigma / \lg \lg n + 1))$ time. □

3.5.3 Path Median and Path Selection

To support path median and path selection queries, we apply the techniques we have developed in this section to generalize the linear space data structure of Brodal et al. [23] for range selection queries. We still build a conceptual range tree, as shown in Section 3.5.2. The only difference is that the branching factor f is set to be $\lceil \lg^\epsilon \sigma \rceil$ instead of $\lceil \lg^\epsilon n \rceil$, where $0 < \epsilon < 1$ is a fixed constant. We choose such a value for f so that $f \lg f$, which will be an additive term in the overall time cost, would not dominate the query time. It turns out that the height h is $\lceil \log_f \sigma \rceil + 1 = O(\lg \sigma / \lg \lg \sigma)$. For $\ell = 1, 2, \cdots, h-1$, we create and store T_ℓ explicitly using Lemma 3.5.5.

Let s be a non-negative integer in $\lceil \lg n \rceil$ bits. When s can be expressed in fewer than $\lceil \lg n \rceil$ bits, we prepend 0-bits to express s in exactly $\lceil \lg n \rceil$ bits. We divide these $\lceil \lg n \rceil$ bits into sections of size $g = \lfloor w/f \rfloor - 1 = \Omega(\lg n / \lg^\epsilon \sigma)$, where $w = \Omega(\lg n)$ is the word size. The first section contains the first g bits of s, which are the g most significant bits. The second section contains the last five bits of the first section and then the following $g - 5$ bits, and so on. Here we duplicate these 5 bits to cover the approximation error, which is bounded in Inequality 3.13. Thus, s is divided into $\Delta = O(\lg n / g) = O(\lg^\epsilon \sigma)$ sections. Let $sect_p(s)$ denote the p-th section of s.

To support path median and path selection queries, we add auxiliary data structures to each T_ℓ. We store for each T_ℓ a two-dimensional array $X[1..n_1, 1..\Delta]$, where n_1 is the number of tier-1 roots in T_ℓ. For each tier-1 root r_i^1 in T_ℓ, $p = 1, 2, \cdots, \Delta$, and $\beta = 1, 2, \cdots, f$, we prepend an extra bit to each $sect_p(A[r_i^1, \beta])$, concatenate them into a single word (see the proof of Lemma 3.5.3 for the definition of $A[r_i^1, \beta]$), and store the word at $X[i, p]$. Thus the space cost of X is $n_1 \times \Delta \times O(\lg n) = O(n \lg^{\epsilon} \sigma / \lg n) = o(n)$ bits. Clearly X can be constructed in $O(n_1 f) = o(n)$ time.

Theorem 3.5.9. *Let T be an ordinal tree on n nodes, each having a weight drawn from $[1..\sigma]$. Under the word RAM model with word size $w = \Omega(\lg n)$, T can be represented in (a) $n(H(W_T) + 2) + o(n)$ bits when $\sigma = O(1)$, or (b) $nH(W_T) + O(n \lg \sigma / \lg \lg \sigma)$ bits otherwise, supporting path median and path selection queries in $O(\lg \sigma / \lg \lg \sigma)$ time. It requires $O(n(\lg \sigma / \lg \lg n + 1))$ preprocessing time to encode T.*

Proof. We need only show how to answer a query. Let u and v be the endpoints of the query path, and let k be the rank of the weight to select. We still partition $P_{u,v}$ into $A_{u,t}$, $A_{v,t}$ and $\{t\}$, where t is the lowest common ancestor of u and v. Our recursive algorithm is shown in Algorithm 4. Provided that the range $[a..b]$ is at the ℓ-th level of the conceptual range tree, $c = |R_{1,a-1}|, d = |R_{1,b}|$, and x, y, z are the nodes in T_ℓ that correspond to $anc_{a,b}(T, u)$, $anc_{a,b}(T, v)$ and $anc_{a,b}(T, t)$, the procedure $\textsc{Select}([a..b], \ell, c, d, x, y, z, s)$ returns the s-th smallest weight among the weights of nodes in $P_{u,v} \cap R_{a,b}$. To compute the given query, we only need call $\textsc{Select}([1..\sigma], 1, 0, n, u, v, t, k)$.

Algorithm 4 The algorithm for path median and selection queries on $P_{u,v}$.

```
procedure Select($[a..b], \ell, c, d, x, y, z, s$)
    if $a = b$ then
        return $a$;
    end if
    Let $[a_1..b_1], [a_2..b_2], \cdots, [a_f..b_f]$ be the child ranges of $[a..b]$;
    $r^1_{\delta_1} \leftarrow$ the root of the mini-tree in $T_\ell$ that contains $\delta$, for $\delta \in \{x, y, z\}$;
    $q \leftarrow \min\{1 \le p \le \Delta \mid sect_p(|P_{u,v} \cap R_{a,b}|) > 14\}$;
    if such a section exists and $q < \Delta$ then
        compute $index$ as described in the proof of Theorem 3.5.9;
        $\alpha_1 \leftarrow \min\{1 \le \beta \le f \mid index(\beta) \ge sect_q(s) - 3\}$;
        $\alpha_2 \leftarrow \min\{1 \le \beta \le f \mid index(\beta) > sect_q(s) + 4\}$;
        verify if $\gamma = \alpha_1$ or $\gamma = \alpha_2 - 1$; otherwise compute $\gamma$ by binary search;
                                                        ▷ $\gamma$ is defined as Inequality 3.9.
    else
        recover the exact content of $index$ as described;
        $\gamma \leftarrow \min\{1 \le \beta \le f \mid index(\beta) \ge sect_\Delta(s)\}$;
    end if
    $c_\gamma \leftarrow c + \texttt{pre\_count}_{\gamma-1}(T_\ell, d) - \texttt{pre\_count}_{\gamma-1}(T_\ell, c)$;
    $d_\gamma \leftarrow c + \texttt{pre\_count}_{\gamma}(T_\ell, d) - \texttt{pre\_count}_{\gamma}(T_\ell, c)$;
    $\delta_\gamma \leftarrow$ the node in $T_{\ell+1}$ that corresponds to $\texttt{lowest\_anc}_\gamma(T_\ell, \delta)$, for $\delta \in \{x, y, z\}$;
    return Select($[a_\gamma..b_\gamma], \ell + 1, c_\gamma, d_\gamma, x_\gamma, y_\gamma, z_\gamma, s - \texttt{node\_count}_{\gamma-1}(T_\ell, u, v, t)$);
end procedure
```

Now let us analyze the procedure Select. Let the child ranges of $[a..b]$ be $[a_1..b_1]$, $[a_2..b_2]$, …, $[a_f..b_f]$. Our objective is to determine the only child range of $[a..b]$ that contains the s-th smallest weight. Formally, we are looking for $1 \leq \gamma \leq f$ such that

$$\texttt{node\_count}_{\gamma-1}(T_\ell, u, v, t) < s \leq \texttt{node\_count}_{\gamma}(T_\ell, u, v, t), \tag{3.9}$$

where $\texttt{node\_count}_{\gamma}(T_\ell, u, v, t)$ denotes the number of nodes in $P_{u,v}$ whose weights are in $[a..b_\gamma]$, which can be computed by the following equation in constant time:

$$\texttt{node\_count}_{\gamma}(T_\ell, x) + \texttt{node\_count}_{\gamma}(T_\ell, y) - 2 \cdot \texttt{node\_count}_{\gamma}(T_\ell, z) + |\{t\} \cap R_{a,b_\gamma}|. \tag{3.10}$$

We can always compute γ by a binary search in $O(\lg f)$ time. Thus the overall query time would be $h \cdot O(\lg f) = O(\lg \sigma)$, which we cannot afford.

To speed up the selection, we guide the search using X. Without loss of generality, the following discussion is based on a fixed T_ℓ. We define a function $nc(x, \beta) = \texttt{node\_count}_{\beta}(T_\ell, x)$, where x is a node in T_ℓ and β is in $[1..f]$. Note that if x is a tier-1 root, say r_i^1, then $nc(x, \beta) = A[i, \beta]$. In line 6, we compute the roots of the mini-trees in T_ℓ that contain x, y and z, which are denoted by $r_{x_1}^1$, $r_{y_1}^1$ and $r_{z_1}^1$, respectively. In line 7, we compute q, the subscript of the first section of $|P_{u,v} \cap R_{a,b}|$ whose bits represent a number larger than 14. By Equation 3.10, we have $|P_{u,v} \cap R_{a,b}| = \texttt{depth}(T_\ell, x) + \texttt{depth}(T_\ell, y) - 2 \cdot \texttt{depth}(T_\ell, z) + |\{t\} \cap R_{a,b}|$. Using word parallelism, the most significant 1-bit of $|P_{u,v} \cap R_{a,b}|$ and the value of q can be retrieved in constant time.

We will compute, for $\beta = 1, 2, \cdots, f$, an approximate value of the q-th section of $|P_{u,v} \cap R_{a,b_\beta}|$. We will pack these approximate values into a single indexing word, and we expect to determine the value of γ using this word, which is sufficient in most cases. The value of $|P_{u,v} \cap R_{a,b_\beta}|$ differs from $|A_{u,t} \cap R_{a,b_\beta}| + |A_{v,t} \cap R_{a,b_\beta}|$ by at most 1. We make use of $X[x_1]$ and $X[z_1]$ to approximate the q-th section of $|A_{u,t} \cap R_{a,b_\beta}| = nc(x, \beta) - nc(z, \beta)$, for $\beta = 1, 2, \cdots, f$. Approximate values of the q-th section of $|A_{v,t} \cap R_{a,b_\beta}|$ can be computed in a similar way. Putting them together, we obtain approximate values of the q-th section of $|P_{u,v} \cap R_{a,b_\beta}|$.

We first assume that $q < \Delta$. By Lemma 2.5.1, a mini-tree contains at most $2M$ nodes. We thus have that, for $\beta = 1, 2, \cdots, f$, $nc(x, \beta) - nc(z, \beta) - 2M \leq A[x_1, \beta] - A[z_1, \beta] \leq nc(x, \beta) - nc(z, \beta) + 2M$. Since $g = \omega(\lg M)$, the difference between the actual value and the approximation only affects the first $\Delta - 1$ sections by at most 1. More precisely, for $\beta = 1, 2, \cdots, f$ and $p = 1, 2, \cdots, \Delta - 1$,

$$\begin{aligned} \mathit{sect}_p(nc(x, \beta) - nc(z, \beta)) - 1 &\leq \mathit{sect}_p(A[x_1, \beta] - A[z_1, \beta]) \\ &\leq \mathit{sect}_p(nc(x, \beta) - nc(z, \beta)) + 1. \end{aligned} \tag{3.11}$$

We can even compute all the approximate values of $sect_q(A[x_1, \beta] - A[z_1, \beta])$ for $\beta = 1, 2, \cdots, f$ in constant time. We take the subtraction of $X[x_1, q]$ and $X[z_1, q]$. Note that the q-th section of $A[x_1, \beta]$ is not necessarily larger than or equal to the q-th section of $A[z_1, \beta]$. To avoid cascading carries, we set the prepended extra bit of each section in $X[x_1, q]$ to be 1, and set the prepended extra bit of each section in $X[z_1, q]$ to be 0. After the subtraction, we mask out these extra bits. In this way, we obtain an indexing word $index_{u,t,q}$ for $A_{u,t}$ and the q-th sections. This indexing word is the concatenation of approximate values of $sect_q(A[x_1, \beta] - A[z_1, \beta])$ for $\beta = 1, 2, \cdots, f$. Let $index_{u,t,q}(\beta)$ be the value of its β-th section, which is an approximate value for the first β child ranges. It is easy to see that $index_{u,t,q}(\beta)$ is equal to $sect_q(A[x_1, \beta] - A[z_1, \beta])$, or one larger. By Inequality 3.11, we have that, for $\beta = 1, 2, \cdots, f$,

$$sect_q(nc(x, \beta) - nc(z, \beta)) - 1 \leq index_{u,t,q}(\beta) \leq sect_q(nc(x, \beta) - nc(z, \beta)) + 2. \quad (3.12)$$

We can approximate $|A_{v,t} \cap R_{a,b_\beta}|$ in the same way, obtaining another indexing word $index_{v,t,q}$ for $A_{v,t}$ and the q-th sections. Adding $index_{u,t,q}$ to $index_{v,t,q}$, we obtain $index$, the indexing word for the whole path. No carry happens between sections in the addition, since the first bit of $sect_q(|P_{u,v} \cap R_{a,b}|)$ is 0. We also define $index(\beta)$ be the value of its β-th section. By Inequality 3.12, and concerning that the node t is not taken into account for the approximation, we further have that, for $\beta = 1, 2, \cdots, f$,

$$sect_q(|P_{u,v} \cap R_{a,b_\beta}|) - 3 \leq index(\beta) \leq sect_q(|P_{u,v} \cap R_{a,b_\beta}|) + 4. \quad (3.13)$$

In lines 10 and 11, using word parallelism, we compute α_1 and α_2 such that γ must be in $[\alpha_1, \alpha_2)$. In line 12, we verify if $\gamma = \alpha_1$ or $\gamma = \alpha_2 - 1$ by Inequality 3.9 and Equation 3.10. If not, we compute γ by a binary search in $O(\lg f)$ time. Due to lines 10 and 11, the difference between the sections stored in $index(\alpha_1)$ and $index(\alpha_2 - 1)$ is at most 7. Then by the approximation shown in Inequality 3.13, the q-th section of $|P_{u,v} \cap R_{a_\gamma,b_\gamma}|$ is at most 14. Thus, at the $(\ell + 1)$-st level, the value of q will be increased by at least one. Since the value of q is at most Δ, binary searches will be performed at most $\Delta - 1$ times at all the levels. This explains our choice of the value 14 in line 7.

Now consider the case in which $q = \Delta$ (we set q to be Δ if $|P_{u,v} \cap R_{a,b}| \leq 14$). We can recover the exact content of $index$. We do not need to consider carries since this is the last section. The computation is similar to the proof of Lemma 3.5.3. Then we compute γ using word parallelism, as shown in line 16.

After obtaining γ, we adjust the parameters in lines 18 to 20, and recurse on the child range $[a_\gamma..b_\gamma]$. We finally analyze the time cost. The algorithm traverses from the top level to the bottom level, and the operations except binary searches require constant time at each level. In addition, binary searches are performed at most $\Delta - 1$ times, each using

$O(\lg f)$ time. Thus, the overall time cost is $O(h)+\Delta\cdot O(\lg f) = O((\lg\sigma/\lg\lg\sigma)+f\lg f) = O(\lg\sigma/\lg\lg\sigma)$.

The analysis of the space and preprocessing costs of our data structures is similar to the analysis in the proof of Theorem 3.5.8. □

3.6 Discussion

We have obtained new and improved upper bounds for path queries. Let *occ* denote the size of the output of path reporting queries. Under the pointer machine model, our data structures require $O(n\lg\sigma)$ space, supporting path median, selection and counting queries in $O(\lg\sigma)$ time, and path reporting queries in $O(\lg\sigma + occ)$ time. Under the word RAM model, we design succinct data structures that support path counting queries in $O(\lg\sigma/\lg\lg n+1)$ time, path reporting queries in $O((occ+1)(\lg\sigma/\lg\lg n+1))$ time, and path median and path selection queries in $O(\lg\sigma/\lg\lg\sigma)$ time.

When the preliminary version of this chapter was published [66], Patil et al. [85] concurrently designed another succinct data structure based on heavy path decomposition [94, 60] for path queries. This succinct data structure represents a weighted ordinal tree using $n\lg\sigma+6n+o(n\lg\sigma)$ bits of space, supports path counting, path median and path selection queries in $O(\lg\sigma\lg n)$ time, and supports path reporting queries in $O(\lg\sigma\lg n + occ\lg\sigma)$ time. Thus this result is subsumed by ours.

For path reporting queries, our subsequent work in Section 5.5 has further improved the query time while using $O(n\lg\sigma)$ or near $O(n\lg\sigma)$ bits of space.

We finish this chapter with two open problems. Our succinct data structures use more query time to support path median and path selection queries than path counting queries when σ is much smaller than n. Our first open problem is whether there exist succinct data structures for path median and path selection queries with $O(\lg\sigma/\lg\lg n+1)$ query time using $n\lg\sigma+o(n\lg\sigma)$ bits of space.

Our second open problem is about the preprocessing costs of our succinct data structures. In Section 3.5, we have analyzed the preprocessing time. It remains open to determine the space cost in the preprocessing phase, and to further explore the time-space tradeoffs in the construction of our data structures [47].

Chapter 4

Static and Dynamic Succinct Labeled Ordinal Trees

4.1 Introduction

We address the issue of succinct representations of ordinal trees[1] with satellite data in the form of a label from a large alphabet. Much of this is motivated by the needs of large text-dominated databases that store and manipulate XML documents, which can be essentially modeled as ordinal trees in which each node is assigned a tag drawn from a tag set.

Our representations support a much broader collection of operations than previous work [54, 44, 9], particularly those operations that aim at XML-style document retrieval, such as queries written in the XML path language (XPath). Our data structures are succinct, occupying space close to the information-theoretic lower bound, which is essential to systems and applications that deal with very large data sets.

Our approach is based on "tree extraction", that is, constructing subtrees consisting of nodes with appropriate labels (and their parents). The basic idea of tree extraction was described in Section 2.5.3. In the previous work that uses tree extraction [65, 100] and in Chapter 3, this technique was used to answer queries that are generalizations of geometric queries such as range counting from planar point sets to trees. Here we follow a different approach for a different class of operations that originate in the study of text databases. In our data structures, an input tree is split according to labels on nodes, such that we can maintain structural information and labels jointly in a space-efficient way. Previous solutions to the same problem are all based on different approaches [54, 44, 9].

[1] An ordinal tree is a rooted one in which the children of a node are ordered.

In this chapter, we consider the operations listed in Table 4.1, in which preoder and DFUDS order are defined in Section 2.5. The unlabeled versions of these operations have been studied extensively in the data structures community [54, 43, 64, 82, 42]. Here we list only the labeled versions of these operations. The unlabeled versions simply include all nodes. In other words, the support for unlabeled versions can be reduced to the support for labeled versions at no extra space cost by setting the alphabet size to 1. We refer to the labeled versions of operations as α-*operations*. We call a node whose label is α an α-*node* (we define similarly α-*children*, α-*ancestor*, etc). In addition, we define the α-*rank* of node x in a list to be the number of α-nodes up to and including x in the list. Throughout this chapter, n denotes the size of the input tree and σ denotes the size of the alphabet.

Table 4.1: Operations considered in this chapter, which are divided into three groups. Here we give only the definitions of the labeled versions of operations. Geary et al. [54]'s data structures only support the α-operations marked by †. Our static data structures support the α-operations in Groups 1 and 2, while our dynamic data structures support the α-operations in Groups 1 and 3.

Operation	Description	Our Static	Our Dynamic
$\texttt{label}(x)^\dagger$	label of x	✓	✓
$\texttt{parent}_\alpha(x)^\dagger$	closest α-ancestor of x		
$\texttt{depth}_\alpha(x)^\dagger$	α-depth of x, i.e., number of α-nodes from x to the root		
$\texttt{level\_anc}_\alpha(x,i)^\dagger$	α-ancestor y of x satisfying $\texttt{depth}_\alpha(x) - \texttt{depth}_\alpha(y) = i$		
$\texttt{nbdesc}_\alpha(x)^\dagger$	number of α-nodes in the subtree rooted at x		
$\texttt{pre\_rank}_\alpha(x)^\dagger/\texttt{pre\_select}_\alpha(i)^\dagger$	α-rank of x in preorder/i-th α-node in preorder		
$\texttt{post\_rank}_\alpha(x)^\dagger/\texttt{post\_select}_\alpha(i)^\dagger$	α-rank of x in postorder/i-th α-node in postorder		
$\texttt{LCA}_\alpha(x,y)$	lowest common α-ancestor of nodes x and y		
$\texttt{leaf\_lmost}_\alpha(x)/\texttt{leaf\_rmost}_\alpha(x)$	leftmost/rightmost α-leaf in the subtree rooted at x		
$\texttt{leaf\_rank}_\alpha(x)$	number of α-leaves to the left of x in preorder		
$\texttt{leaf\_select}_\alpha(i)$	i-th α-leaf in preorder		
$\texttt{nbleaf}_\alpha(x)$	number of α-leaves in the subtree rooted at node x		
$\texttt{deg}_\alpha(x)^\dagger$	number of α-children of x	✓	×
$\texttt{child\_rank}_\alpha(x)^\dagger$	α-rank of x in the list of children of $\texttt{parent}(x)$		
$\texttt{child\_select}_\alpha(x,i)^\dagger$	i-th α-child of x		
$\texttt{dfuds\_rank}_\alpha(x)/\texttt{dfuds\_select}_\alpha(i)$	α-rank of x in DFUDS order/i-th α-node in DFUDS order		
$\texttt{height}_\alpha(x)$	α-height of x, i.e., maximum number of α-nodes from x to its leaf descendant		
$\texttt{node\_insert}_\alpha(x)$	insert an α-node x as an internal node or a leaf	×	✓
$\texttt{node\_delete}(x)$	delete non-root node x		

Groups 1 and 2 in Table 4.1 are queries, while Group 3 consists of updates. The third and the fourth columns indicate that our static data structures support all these queries (in time log-logarithmic in σ), while our dynamic data structures support the updates and most of these queries (in time sub-logarithmic in each of n and σ). Our structures are succinct, and most of them are able to handle large alphabets to support these α-operations.

The details will follow in Theorems 4.1.1 to 4.1.3 and the discussions of these theorems.

A straightforward approach to representing a labeled ordinal tree is to maintain the structural information and labels separately. The structure is represented using one of the succinct data structures for unlabeled ordinal trees [54, 43, 64, 82, 42], while the labels are stored in a sequence in preorder or DFUDS order. As shown in Barbay et al. [9], their first approach uses preorder label sequences, and supports only `label`, $\texttt{pre\_rank}_\alpha$, $\texttt{pre\_select}_\alpha$ and $\texttt{nbdesc}_\alpha$ efficiently.[2] Their second approach, which uses DFUDS order label sequences, only supports `label`, $\texttt{dfuds\_rank}_\alpha$, $\texttt{dfuds\_select}_\alpha$, $\texttt{deg}_\alpha$, $\texttt{child\_rank}_\alpha$, and $\texttt{child\_select}_\alpha$. Compared to these trivial ideas, our approach *assembles* structural information and labels by a novel use of tree extraction. Thus our framework provides much richer functionality than these basic methods.

Other existing succinct representations of labeled ordinal trees [44, 54] require a small alphabet or support only a restricted subset of queries, and they do not support updates efficiently. Geary et al. [54] presented data structures to support in constant time the α-operations marked by † in Table 4.1 and their unlabeled versions. The overall space cost of their data structures is $n(\lg\sigma + 2) + O(n\sigma \lg\lg\lg n/\lg\lg n)$ bits, which is much more than the information-theoretic lower bound of $n(\lg\sigma + 2) - O(\lg n)$ bits when $\sigma = \Omega(\lg\lg n)$. Ferragina et al. [44] designed data structures for labeled trees that use space close to the information-theoretic lower bound, but supporting an even more restricted set of α-operations. Their xbw-based representation supports only `label`, $\texttt{child\_select}_\alpha$ and $\texttt{deg}_\alpha$.[3]

The approaches of Barbay et al. [9] can be dynamized using succinct representations of dynamic unlabeled ordinal trees [82] and dynamic label sequences [61, 82, 81]. After dynamization, their first approach can support $\texttt{node\_insert}_\alpha$ and $\texttt{node\_delete}_\alpha$ efficiently (see Section 4.4 for the protocol of insertion and deletion). Their second approach can only support insertions and deletions efficiently for low-degree nodes: To insert a node x so that it will become the parent of $d(x)$ existing nodes, or to delete a node x with $d(x)$ children, $\Theta(d(x))$ operations have to be performed on each of the data structures that represent the corresponding unlabeled tree and label sequence. To the best of our knowledge, besides the dynamic variants of Barbay et al.'s [9] approches, there is no succinct data structure for dynamic labeled ordinal trees with both rich functionality and efficient query/update time.

Our results for static and dynamic labeled ordinal trees are summarized in Theo-

[2] It also supports two other operations: the ancestor of x closest to root with a given label and the first descendant of x with a given label in preorder. Both operations can be easily supported using $\texttt{pre\_rank}_\alpha$, $\texttt{pre\_select}_\alpha$, $\texttt{depth}_\alpha$ and $\texttt{level\_anc}_\alpha$, and thus we do not list them in Table 4.1.

[3] Another query supported by their structure is SubPathSearch, which can count or list nodes whose upward paths start with a given query string.

Table 4.2: Our static representations of labeled ordinal trees. Here $f(n, \sigma)$ is any function that satisfies $f(n, \sigma) = \omega(1)$ and $f(n, \sigma) = o(\lg \frac{\lg \sigma}{\lg w})$, and "others in Groups 1 and 2" represents the first two groups of α-operations in Table 4.1 other than `label` and $\texttt{pre\_select}_\alpha$. All the results are obtained under the standard word RAM model with word size $w = \Omega(\lg n)$. Note that $H_0(PLS_T)$ and $H_k(PLS_T)$ are bounded above by $\lg \sigma$.

		Theorem 4.1.2(a)	Theorem 4.1.2(b)	Theorem 4.1.2(c)
Constraints		$\sigma = w^{O(1)}$	$\sigma \leq n$	$\sigma \leq n$, $\lg \sigma = \omega(\lg w)$, and $k = o(\log_\sigma n)$
Space (in bits)		$n(H_0(PLS_T) + 9) + o(n)$	$n(H_0(PLS_T) + 9) + 3\sigma + o(n(H_0(PLS_T) + 1))$	$nH_k(PLS_T) + o(n \lg \sigma)$
Time	$\texttt{label}(x)$	$O(1)$	$O(1)$	$O(1)$
	$\texttt{pre\_select}_\alpha(i)$		$O(f(n, \sigma))$	$O(f(n, \sigma))$
	others in Groups 1/2		$O(\lg \frac{\lg \sigma}{\lg w})$	$O(\lg \frac{\lg \sigma}{\lg w})$

rems 4.1.1 to 4.1.3. As a preliminary result, we improve the succinct representation of labeled ordinal trees of Geary et al. [54]. As shown in Theorem 4.1.1, the improved representation supports more operations while occupying less space, where PLS_T is the preorder label sequence of T, and H_k is the *k-th order empirical entropy* [77], which is bounded above by $\lg \sigma$. However, this data structure is succinct only if the size of alphabet is very small, i.e., $\sigma = o(\lg \lg n)$.

Operations $\texttt{deepest}_\alpha$ and $\texttt{min\_depth}_\alpha$ are auxiliary α-operations used in Sections 4.2.5 and 4.2.7: $\texttt{deepest}_\alpha(i, j)$ returns a node (there could be a tie) with preorder rank in $[i..j]$ that has the maximum α-depth, and $\texttt{min\_depth}_\alpha(i, j)$ returns an α-node with preorder rank in $[i..j]$ that has the minimum depth (i.e., is closest to the root).

Theorem 4.1.1. *Let T be a static ordinal tree on n nodes, each having a label drawn from an alphabet of size $\sigma = o(\lg \lg n)$. Under the word RAM with word size $w = \Omega(\lg n)$, for any $k = o(\log_\sigma n)$, there exists a data structure that encodes T using $n(H_k(PLS_T) + 2) + O(\frac{nk \lg \sigma}{\log_\sigma n}) + O(\frac{n\sigma \lg \lg \lg n}{\lg \lg n})$ bits of space, supporting the α-operations in Groups 1 and 2 of Table 4.1 and their unlabeled versions, plus two additional α-operations $\texttt{deepest}_\alpha$ and $\texttt{min\_depth}_\alpha$, in constant time.*

We summarize the main result of this chapter in Theorem 4.1.2 and Table 4.2. To achieve this result, we present a framework for succinct representations of labeled ordinal trees. As Lemma 4.2.1 will show, bit vectors, preorder label sequences, and unlabeled and 0/1-labeled ordinal trees (i.e., ordinal trees over the alphabet $\{0, 1\}$) are extracted from the original labeled tree in an original way. The method ensures that an α-operation is reduced to a constant number of operations on these simpler data structures that were supported by various approaches in previous work. This time- and space-efficient reduction allows us to handle large alphabets, and to compress labels of nodes into entropy bounds. Thus our framework answers an open problem proposed by Geary et al. [54], which asks for representations of labeled ordinal trees that remain space-efficient for large alphabets.

Theorem 4.1.2. *Let T be a static ordinal tree on n nodes, each having a label drawn from an alphabet $\Sigma = [1..\sigma]$. Under the word RAM with word size $w = \Omega(\lg n)$,*

(a) for $\sigma = w^{O(1)}$, T can be represented using $n(H_0(PLS_T) + 9) + o(n)$ bits of space to support the α-operations in Groups 1 and 2 of Table 4.1 using constant time;

(b) for $\sigma \leq n$, T can be represented using $n(H_0(PLS_T) + 9) + 3\sigma + o(n(H_0(PLS_T) + 1))$ bits of space to support the α-operations in Groups 1 and 2 of Table 4.1 using $O(\lg \frac{\lg \sigma}{\lg w})$ time;

(c) for $\sigma \leq n$ and $\lg \sigma = \omega(\lg w)$, and for any integer $k = o(\log_\sigma n)$, T can be represented using $nH_k(PLS_T) + o(n \lg \sigma)$ bits to support the α-operations in Groups 1 and 2 of Table 4.1 using $O(\lg \frac{\lg \sigma}{\lg w})$ time.

In addition, these data structures support the unlabeled versions of these α-operations in $O(1)$ time.

As shown in Theorem 4.1.3, we further extend our work to the dynamic case. We present the first succinct representations of dynamic labeled ordinal trees that support several α-operations related to ancestors, descendants, and leaves.

Theorem 4.1.3. *Let T be a dynamic ordinal tree on n nodes, each having a label drawn from an alphabet $\Sigma = [1..\sigma]$. Under the word RAM with word size $w = \Omega(\lg n)$, and for any $k = o(\log_\sigma n)$, T can be represented using $n(H_k(PLS_T) + 5) + o(n \lg \sigma)$ bits of space, such that $\texttt{level\_anc}_\alpha$ can be supported in $O(\lg n)$ worst-case time, and the other α-operations listed in Groups 1 and 3 of Table 4.1 can be supported in $O(\frac{\lg n}{\lg \lg n})$ worst-case time. In addition, this data structure supports `level_anc` using $O(\lg n)$ worst-case time, and the unlabeled versions of the other α-operations listed in Groups 1 and 3 of Table 4.1 using $O(\frac{\lg n}{\lg \lg n})$ worst-case time.*

In this chapter, an alphabet Σ is said to be small if its size is $o(\lg \lg n)$; otherwise it is a large alphabet. The data structure in Theorem 4.1.1 is succinct only if the alphabet is small, while the data structures in Theorems 4.1.2 and 4.1.3 are succinct if the alphabet is of size $\omega(1)$. In other words, the latter ones are featured for supporting large alphabets.

The rest of this chapter is organized as follows. In Section 4.2, we describe the construction of our data structures for static trees over large alphabets, i.e., the proof of Theorem 4.1.2, assuming that Theorem 4.1.1 holds. This section contains the most interesting ideas of this chapter for most readers. The rather technical proof of Theorem 4.1.1

is deferred to Section 4.3, where we extend and improve Geary et al.'s [54] representation of labeled ordinal trees for small alphabets. Section 4.4 describes our data structures for dynamic trees, i.e., the proof of Theorem 4.1.3. Finally, the conclusions and open problems are in Section 4.5.

4.2 Static Trees over Large Alphabets : Theorem 4.1.2

We start with the following observation to develop the proof of Theorem 4.1.2: For each possible subscript $\alpha \in \Sigma$, we could support the α-operations in Groups 1 and 2 of Table 4.1 by relabeling T into a 0/1-labeled tree and indexing the relabeled tree, where a node is relabeled 1 if and only if it is an α-node in T. However, we would have to store σ trees that have $n\sigma$ nodes in total if we simply apply this idea for each $\alpha \in \Sigma$, which we could not afford.

Instead of storing all the n nodes for each $\alpha \in \Sigma$, we store only the nodes that are particularly relevant to α, i.e., the α-nodes and their parents, while maintaining the ancestor-descendant relationship between these nodes. We apply tree extraction to summarize such information, where the tree constructed for label α is denoted by T_α.

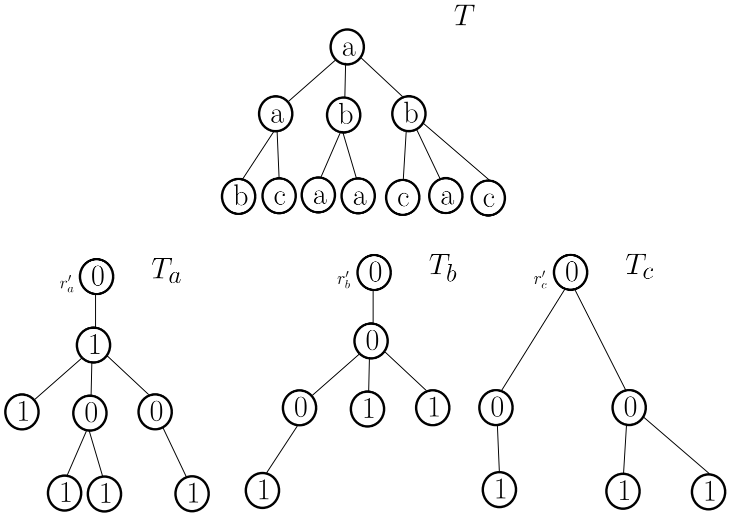


Figure 4.1: An example with 11 nodes and an alphabet Σ of size 3, where we use $\Sigma = \{a, b, c\}$ for clarity. The character on each node in T is its label. The integer on each node in T_α, for each $\alpha \in \Sigma$, indicates whether this node corresponds to an α-node in T.

For $\alpha \in \Sigma$, we create a new root r_α, and make the original root of T be the only child of r_α. The structure of T_α is obtained by computing the X_α-extraction of the augmented tree rooted at r_α, where X_α is the union of r_α, the α-nodes in T, and the parents of the α-nodes. The natural one-to-one mapping between the nodes in X_α and T_α determines the labels of the nodes in T_α. The root of T_α is always labeled 0. A non-root node in T_α is labeled 1 if its corresponding node in T is an α-node, and 0 otherwise. Thus, the number of 1-nodes in T_α is equal to the number of α-nodes in T. Let n_α denote both values. The construction above is illustrated by the example in Figure 4.1.

To clarify notation, the nodes in T are denoted by lowercase letters, while the nodes in T_α are denoted by lowercase letters plus prime symbols. To illustrate the one-to-one mapping, we denote by x' a node in T_α if and only if its corresponding node in T is denoted by x. The root of T_α, which corresponds to r_α, is denoted by r'_α. We show how to convert the corresponding nodes in T and T_α using the preorder label sequence of T in Section 4.2.2.

Since the structure of T_α is different from that of T, we need also store the structure of T and the labels of the nodes in T so that we can perform conversions between the nodes in T and T_α. In addition, to support the α-operations related to leaves, we store for each $\alpha \in \Sigma$ a bit vector $L_\alpha[1..n_\alpha]$ in which the i-th bit is one if and only if the i-th 1-node in preorder of T_α corresponds to a leaf in T.

Following this approach, our succinct representation consists of four components: (a) the structure of T; (b) PLS_T, the preorder label sequence of T; (c) a 0/1-labeled tree T_α for each $\alpha \in \Sigma$; and (d) a bit vector L_α for each $\alpha \in \Sigma$. The unlabeled versions of the α-operations in Groups 1 and 2 of Table 4.1 are directly supported by the data structure that maintains the structure of T. For α-operations, our basic idea is to reduce an α-operation to a constant number of operations on T, PLS_T, T_α's, and L_α's, for which we summarize the previous work in Sections 2.3, 2.4 and 4.2.9.

In the following sections, we describe our algorithms in terms of T, PLS_T, T_α's and L_α's. All of these algorithms require $O(1)$ time and perform $O(1)$ operations on these components. For each operation, we specify the related component as the first parameter. If such a component is not specified in context, then this operation is performed on T. To handle operations with subscript $\alpha \in \Sigma$, our algorithms only access the structure of T, PLS_T, T_α and L_α. A node in T_α is thus denoted by x' instead of x_α for simplicity.

4.2.1 Operations label, pre_rank$_\alpha$, pre_select$_\alpha$ and nbdesc$_\alpha$

By the definitions, we know:

$$\begin{aligned}
\texttt{label}(x) &= \texttt{access}(PLS_T, \texttt{pre\_rank}(x)) \\
\texttt{pre\_rank}_\alpha(x) &= \texttt{rank}_\alpha(PLS_T, \texttt{pre\_rank}(x)) \\
\texttt{pre\_select}_\alpha(i) &= \texttt{pre\_select}(\texttt{select}_\alpha(PLS_T, i))
\end{aligned}$$

We make use of $\texttt{pre\_rank}_\alpha$ and $\texttt{pre\_select}_\alpha$ to find the α-predecessor and the α-successor of node x, which are defined to be the last α-node preceding x and the first α-node succeeding x in preorder (both can be x itself).

We support $\texttt{nbdesc}_\alpha(x)$ in the same way as Barbay et al. [9]. The descendants of x form a consecutive substring in PLS_T, which starts at index $\texttt{pre\_rank}(x)$ and ends at index $\texttt{pre\_rank}(x) + \texttt{nbdesc}(x) - 1$. We can compute the number of α-nodes lying in this range using $\texttt{rank}_\alpha$ on PLS_T. Provided that x has an α-descendant, we can further compute the first and the last α-descendant of node x in preorder, which are the α-successor of x and the α-predecessor of the node that has preorder rank $\texttt{pre\_rank}(x) + \texttt{nbdesc}(x) - 1$, respectively. Let these α-nodes be u and v. For simplicity, we call $[u, v]$ the α-*boundary* of the subtree rooted at x.

4.2.2 Conversion between Nodes in T and T_α

The conversion between nodes in T and the corresponding nodes in T_α plays an important role in supporting α-operations. Given an α-operation on node x in T, our algorithms in this section usually do what follows: They first compute the node y' in T_α that corresponds to x (or the parent of x, the first α-descendant of x, etc), and they retrieve appropriate information with respect to y' by querying the data structures that maintain T_α. Here T_α's will be maintained by the data structures presented in Theorem 4.1.1, since they are 0/1-labeled. If the information retrieved is a node z' in T_α, our algorithms will further compute the node z in T that corresponds to z'.

We first discuss the backward direction of conversion, i.e., computing the node x in T that corresponds to a given node x' in T_α. If x' is a 1-node, then x must be an α-node. By Lemma 2.5.4, we have that $x = \texttt{pre\_select}_\alpha(\texttt{pre\_rank}_1(T_\alpha, x'))$. Otherwise, by the definition of T_α, x must have at least one α-child. In addition, every α-child of x in T must appear as a 1-child of x' in T_α. Using this fact, x can be computed from x' as follows: We first find the first 1-child of x' in T_α, say $y' = \texttt{child\_select}_1(T_\alpha, x', 1)$, and compute the node y in T that corresponds to y' using the case in which the given node is a 1-node. Then x must be the parent of y in T.

The other direction of conversion is more complex. Algorithm 5 shows how to compute the node x' in T_α that corresponds to a given node x in T. This algorithm returns NULL if no such x' exists. If x is an α-node, then x' must exist in T_α. As shown in line 3, applying Lemma 2.5.4 again, we have that $x' = \texttt{pre\_select}_1(T_\alpha, \texttt{pre\_rank}_\alpha(x))$.

The case in which x is not an α-node requires more attention, since the node x' may or may not exist in T_α. We first verify whether x has at least one α-descendant in line 4 using $\texttt{nbdesc}_\alpha$. The node x' does not exist in T_α if x has no α-descendant. Otherwise, we compute the α-boundary, $[u, v]$, of the subtree rooted at x in line 7. We then have two cases, depending on whether x is the lowest common ancestor of u and v. If $x \neq \texttt{LCA}(u, v)$, then y, which is set to be $\texttt{LCA}(u, v)$, must be a descendant of x. All α-descendants of x must also be descendants of y, and, if x has an α-child, then this α-child must be y. Thus, we need only check if y is an α-child of x, as shown in lines 10 to 14. Now suppose $x = \texttt{LCA}(u, v)$. Since the construction of T_α preserves the ancestor-descendant relationship, we claim that x' must be the lowest common ancestor of u' and v' if x' exists in T_α. Thus, we need only compute $z' = \texttt{LCA}(T_\alpha, u', v')$ in line 16, and verify whether z' corresponds to x using the backward direction in lines 17 to 21.

Algorithm 5 Given node x in T, computing the node x' in T_α that corresponds to x

```
 1: procedure CONVERT($T, x, \alpha$)
 2:     if $x$ is an $\alpha$-node then
 3:         return pre_select$_1$($T_\alpha$, pre_rank$_\alpha$($x$));
 4:     else if $x$ has no $\alpha$-descendant then
 5:         return NULL;
 6:     end if
 7:     $[u, v] \leftarrow$ the $\alpha$-boundary of the subtree rooted at $x$;
 8:     if $x \neq$ LCA($u, v$) then
 9:         $y \leftarrow$ LCA($u, v$);
10:         if $y$ is an $\alpha$-child of $x$ then          ▷ check if label($y$) $= \alpha$ and parent($y$) $= x$
11:             return parent($T_\alpha, y'$);
12:         else
13:             return NULL;
14:         end if
15:     else
16:         $z' \leftarrow$ LCA($T_\alpha, u', v'$);
17:         if $z'$ corresponds to $x$ then                                            ▷ $z = x$
18:             return $z'$;
19:         else
20:             return NULL;
21:         end if
22:     end if
23: end procedure
```

We have described how to perform the conversion. In the following sections, we assume that x and x' are both known if one of them is known. The conversion will be done implicitly.

4.2.3 Operations $\texttt{parent}_\alpha$, $\texttt{level\_anc}_\alpha$, $\texttt{LCA}_\alpha$ and $\texttt{depth}_\alpha$

We first show how to compute $\texttt{parent}_\alpha(x)$ in Algorithm 6. The case in which x is an α-node is solved in lines 3 to 4. We simply compute $y' = \texttt{parent}_1(T_\alpha, x')$ and return y. Suppose x is not an α-node. We compute u, the α-predecessor of x in preorder, in line 6. We claim that x has no α-parent if x has no α-predecessor in preorder, since the ancestors of x precede x in preorder. If such u exists, we take a look at $v = \texttt{LCA}(u, x)$ in line 10. We further claim that there is no α-node on the path between v and x (excluding v), because u would not be the α-predecessor if such an α-node existed. In addition, we know that v has at least one α-descendant because of the existence of u. We return v if v is an α-node. Otherwise, we compute the first α-descendant, z, of v. It is clear that there is no α-node on the path between z and v (excluding z). Thus, the α-parent of z, being computed in line 15, must be the α-parent of both v and x.

Algorithm 6 $\texttt{parent}_\alpha(x)$

```
 1: procedure PARENT($T, x, \alpha$)
 2:     if $x$ is an $\alpha$-node then
 3:         $y' \leftarrow \texttt{parent}_1(T_\alpha, x')$;
 4:         return $y$;
 5:     end if
 6:     $u \leftarrow$ the $\alpha$-predecessor of $x$ in preorder of $T$;
 7:     if $x$ has no $\alpha$-predecessor then
 8:         return NULL;
 9:     end if
10:     $v \leftarrow \texttt{LCA}(u, x)$;
11:     if $v$ is an $\alpha$-node then
12:         return $v$;
13:     end if
14:     $z \leftarrow$ the first $\alpha$-descendant of $v$ in preorder;
15:     $y' \leftarrow \texttt{parent}_1(T_\alpha, z')$;
16:     return $y$;
17: end procedure
```

Then we make use of $\texttt{parent}_\alpha(x)$ to support $\texttt{level\_anc}_\alpha(x, i)$: We first compute $y = \texttt{parent}_\alpha(x)$, where y must be an α-node or NULL. We return y if $y = \texttt{NULL}$ or $i = 1$. Otherwise, we compute $z' = \texttt{level\_anc}_1(T_\alpha, y', i-1)$ and return z.

$\texttt{LCA}_\alpha$ and $\texttt{depth}_\alpha$ can also be easily supported using $\texttt{parent}_\alpha$. $\texttt{LCA}_\alpha(x, y)$ is equal to $\texttt{LCA}(x, y)$ if the lowest common ancestor of x and y is an α-node; otherwise it is equal to $\texttt{parent}_\alpha(\texttt{LCA}(x, y))$. To compute $\texttt{depth}_\alpha(x)$, let $y = x$ if x is an α-node, or $y = \texttt{parent}_\alpha(x)$ if x is not. It is then clear that $\texttt{depth}_\alpha(x) = \texttt{depth}_\alpha(y) = \texttt{depth}_1(T_\alpha, y')$, since each α-ancestor of y in T corresponds to a 1-ancestor of y' in T_α.

4.2.4 Operations $\texttt{child\_rank}_\alpha$, $\texttt{child\_select}_\alpha$ and $\texttt{deg}_\alpha$

We can support $\texttt{child\_select}_\alpha(x, i)$ and $\texttt{deg}_\alpha(x)$ using the techniques shown in Section 4.2.2. We first try to find x', the node in T_α that corresponds to x. If such x' does not exist, then x must have no α-child. Thus, we return `NULL` for $\texttt{child\_select}_\alpha(x, i)$ and return 0 for $\texttt{deg}_\alpha(x)$. Otherwise, we compute $y' = \texttt{child\_select}_1(T_\alpha, x', i)$ and return y for $\texttt{child\_select}_\alpha(x, i)$, as well as return $\texttt{deg}_1(T_\alpha, x')$ for $\texttt{deg}_\alpha(x)$.

The algorithm to support $\texttt{child\_rank}_\alpha(x)$ is shown as Algorithm 7. The case in which x is an α-node is easy to handle, as shown in line 3. We consider now the case in which the label of x is not α. In lines 5 to 8, we compute the node, u, that is the parent of x, and verify if u has an α-child using $\texttt{deg}_\alpha$. We return 0 if u has no α-child. Otherwise, we compute the α-predecessor, v, of x in preorder. If such v does not exist, or v is not a proper descendant of u, then x has no α-sibling preceding it and we can return 0, since a sibling preceding x occurs before x in preorder. Suppose v exists as a proper descendant of u. We can find u' and v', the nodes in T_α that correspond to u and v, since u has an α-child and v is an α-node. In addition, we find the child, y', of u' that is an ancestor of v'. Observe that each α-child of u in T that precedes x corresponds to a 1-child of u' in T_α that precedes y'. Otherwise, v would not be the α-predecessor. Finally, we return $\texttt{child\_rank}_1(T_\alpha, y')$ as the answer.

Algorithm 7 $\texttt{child\_rank}_\alpha(x)$

```
 1: procedure CHILD_RANK($T, x, \alpha$)
 2:     if $x$ is an $\alpha$-node then
 3:         return $\texttt{child\_rank}_1(T_\alpha, x')$;
 4:     end if
 5:     $u \leftarrow \texttt{parent}(x)$;
 6:     if $u$ has no $\alpha$-child then                    ▷ check if $\texttt{deg}_\alpha(u) = 0$
 7:         return 0;
 8:     end if
 9:     $v \leftarrow$ the $\alpha$-predecessor of $x$ in preorder;
10:     if $x$ has no $\alpha$-predecessor or $\texttt{pre\_rank}(v) \leq \texttt{pre\_rank}(u)$ then
11:         return 0;
12:     end if
13:     compute $u'$ and $v'$, the nodes in $T_\alpha$ that correspond to $u$ and $v$;
14:     $y' \leftarrow$ the child of $u'$ that is an ancestor of $v'$;
15:                  ▷ $y' = \texttt{level\_anc}(T_a, v', \texttt{depth}(T_a, v') - \texttt{depth}(T_a, u') - 1)$
16:     return $\texttt{child\_rank}_1(T_\alpha, y')$;
17: end procedure
```

4.2.5 Operation $\texttt{height}_\alpha$

To compute $\texttt{height}_\alpha(x)$, we need find node y among the descendants of x such that $\texttt{depth}_\alpha(y)$ is maximized. If x has no α-descendant, then $\texttt{height}_\alpha(x) = 0$. Suppose x has at least one α-descendant. We compute $[u, v]$, the α-boundary of the subtree rooted at x. All α-descendants of x occur between u and v in preorder of T. The α-depth of a node that does not have label α is equal to the α-depth of its closest α-ancestor. Therefore, there exists an α-descendant of x whose α-depth is the largest among all descendants of x. Hence, we consider only the α-descendants of x in the search for y. Remember that $\texttt{depth}_\alpha(z) = \texttt{depth}_1(T_\alpha, z')$ for any α-node z in T. Thus the α-descendant that has the largest α-depth can be computed in T_α instead of in T. Let $i = \texttt{pre\_rank}_1(T_\alpha, u')$ and $j = \texttt{pre\_rank}_1(T_\alpha, v')$. We have that $y' = \texttt{deepest}_1(T_\alpha, i, j)$. Finally, we return $\texttt{depth}_\alpha(y)$ as the answer if x is the root, or $\texttt{depth}_\alpha(y) - \texttt{depth}_\alpha(\texttt{parent}(x))$ otherwise.

4.2.6 Operations $\texttt{post\_rank}_\alpha$ and $\texttt{post\_select}_\alpha$

Operation $\texttt{post\_rank}_\alpha$ relies on several operations supported in Section 4.2.1 and Section 4.2.3. We address their relationship in the following equation: $\texttt{post\_rank}_\alpha(x) = \texttt{pre\_rank}_\alpha(x) - \texttt{depth}_\alpha(x) + \texttt{nbdesc}_\alpha(x)$. This equation holds for any node in T.

By Lemma 2.5.4, we have $\texttt{post\_rank}_\alpha(x) = \texttt{post\_rank}_1(T_\alpha, x')$ for any α-node x in T. Hence, to compute $\texttt{post\_select}_\alpha(i)$, we simply compute $x' = \texttt{post\_select}_1(T_\alpha, i)$, and return x.

4.2.7 Operations $\texttt{dfuds\_rank}_\alpha$ and $\texttt{dfuds\_select}_\alpha$

The following formula shows how to compute $\texttt{dfuds\_rank}_\alpha(x)$ for a non-root node x in T, which is based on a similar formula in [11] that deals with the unlabeled version:

$$\begin{aligned}\texttt{dfuds\_rank}_\alpha(x) &= \texttt{child\_rank}_\alpha(x) + \texttt{pre\_rank}_\alpha(\texttt{parent}(x)) \\ &+ \sum_{y \in \texttt{Anc}(x)\backslash r} \big(\texttt{deg}_\alpha(\texttt{parent}(y)) - \texttt{child\_rank}_\alpha(y)\big),\end{aligned} \tag{4.1}$$

where $\texttt{Anc}(x)$ is the set of ancestors of x (excluding x itself), and r is the root of T. This formula reduces the computation of $\texttt{dfuds\_rank}_\alpha(x)$ to the computation of

$$\sum_{y \in \texttt{Anc}(x)\backslash r} (\texttt{deg}_\alpha(\texttt{parent}(y)) - \texttt{child\_rank}_\alpha(y)), \tag{4.2}$$

which is denoted by $S_\alpha(x)$. Thus, Equation 4.1 can be simplified as

$$\texttt{dfuds\_rank}_\alpha(x) \;=\; \texttt{child\_rank}_\alpha(x) + \texttt{pre\_rank}_\alpha(\texttt{parent}(x)) + S_\alpha(x). \qquad (4.3)$$

For each $\alpha \in \Sigma$, we have similar formulas on T_α. For any non-root node x' in T_α, we define $S_1(T_\alpha, x') = \sum_{y' \in \texttt{Anc}(T_\alpha, x') \setminus r'_\alpha} (\texttt{deg}_1(T_\alpha, \texttt{parent}(T_\alpha, y')) - \texttt{child\_rank}_1(T_\alpha, y'))$, where $\texttt{Anc}(T_\alpha, x')$ is the set of ancestors of x' in T_α (excluding x' itself), and r'_α is the root of T_α. We then have

$$\begin{aligned}\texttt{dfuds\_rank}_1(T_\alpha, x') &= \texttt{child\_rank}_1(T_\alpha, x') + \texttt{pre\_rank}_1(T_\alpha, \texttt{parent}(T_\alpha, x')) \\ &\quad + \sum_{y' \in \texttt{Anc}(T_\alpha, x') \setminus r'_\alpha} (\texttt{deg}_1(T_\alpha, \texttt{parent}(T_\alpha, y')) - \texttt{child\_rank}_1(T_\alpha, y')) \\ &= \texttt{child\_rank}_1(T_\alpha, x') + \texttt{pre\_rank}_1(T_\alpha, \texttt{parent}(T_\alpha, x')) + S_1(T_\alpha, x'). \qquad (4.4)\end{aligned}$$

Hence, for any non-root node x' in T_α, we can easily compute $S_1(T_\alpha, x')$ by subtracting the values of $\texttt{child\_rank}_1(T_\alpha, x')$ and $\texttt{pre\_rank}_1(T_\alpha, \texttt{parent}(T_\alpha, x'))$ from $\texttt{dfuds\_rank}_1(T_\alpha, x')$.

We now prove that, for any non-root node x' in T_α, $S_1(T_\alpha, x') = S_\alpha(x)$. For x in T and $y \in \texttt{Anc}(x)$, let $f(x, y)$ denote the child of y that is an ancestor of x. Expanding $S_\alpha(x)$, we have that

$$\begin{aligned}S_\alpha(x) &= \sum_{y \in \texttt{Anc}(x) \setminus r} \big(\texttt{deg}_\alpha(\texttt{parent}(y)) - \texttt{child\_rank}_\alpha(y)\big) \\ &= \sum_{y \in \texttt{Anc}(x) \setminus \texttt{parent}(x)} \big(\texttt{deg}_\alpha(y) - \texttt{child\_rank}_\alpha(f(x, y))\big) \qquad (4.5)\end{aligned}$$

Similarly, for x' in T_α and $y' \in \texttt{Anc}(T_\alpha, x')$, we denote by $g(x', y')$ the child of y' that is an ancestor of x'. Thus we have the following equation for $S_1(T_\alpha, x')$.

$$\begin{aligned}S_1(T_\alpha, x') &= \sum_{y' \in \texttt{Anc}(T_\alpha, x') \setminus r'_\alpha} \big(\texttt{deg}_1(T_\alpha, \texttt{parent}(T_\alpha, y')) - \texttt{child\_rank}_1(T_\alpha, y')\big) \\ &= \sum_{y' \in \texttt{Anc}(T_\alpha, x') \setminus \texttt{parent}(T_\alpha, x')} \big(\texttt{deg}_1(T_\alpha, y') - \texttt{child\_rank}_1(T_\alpha, g(x', y'))\big) \qquad (4.6)\end{aligned}$$

On the right-hand side of Equation 4.5, a term contributes to the summation only if $\texttt{deg}_\alpha(y) > 0$. In this case, y must have a corresponding node y' in T_α. Thus it suffices to show that the following identity holds: $\texttt{deg}_\alpha(y) - \texttt{child\_rank}_\alpha(f(x, y)) = \texttt{deg}_1(T_\alpha, y') - \texttt{child\_rank}_1(T_\alpha, g(x', y'))$. It is clear that $\texttt{deg}_\alpha(y) = \texttt{deg}_1(T_\alpha, y')$. Let $u = f(x, y)$, and $v' = g(x', y')$. By the construction of T_α, we have that u is an ancestor of v. Thus, any 1-child of y' occurs before v' if and only if the α-child of y that corresponds to this 1-child occurs before u. Also, if u is an α-node, then $v = u$. Hence, we conclude that $\texttt{child\_rank}_\alpha(f(x,y)) = \texttt{child\_rank}_1(T_\alpha, g(x',y'))$. This completes the proof.

We thus have $\texttt{dfuds\_rank}_\alpha(x) = \texttt{dfuds\_rank}_1(T_\alpha, x')$ for any α-node x in T. This claim directly gives us a simple algorithm to compute $\texttt{dfuds\_select}_\alpha(i)$. We need only compute $y' = \texttt{dfuds\_select}_1(T_\alpha, i)$, and return y.

Now consider how to support $\texttt{dfuds\_rank}_\alpha(x)$. As described above, this task is easy if x has a corresponding node x' in T_α. Thus we focus on the case in which such x' does not exist, i.e., x is neither an α-node nor the parent of an α-node. The algorithm to compute $S_\alpha(x)$ for such x is shown as Algorithm 8. The basic idea is to find node y, the lowest ancestor of x that has a corresponding node in T_α. All but a constant number of nodes between y and x contribute nothing to $S_\alpha(x)$. Thus, $S_\alpha(x)$ can be computed from $S_\alpha(y)$ using a constant number of $\texttt{deg}_\alpha$ and $\texttt{child\_rank}_\alpha$ operations, as shown in lines 17 to 21.

At the beginning of the algorithm, we ensure that there exists at least one α-node in T. In lines 5 and 9, we find the α-predecessor of x, and the first α-node in preorder

Algorithm 8 Computing $S_\alpha(x)$ when x' does not occur in T_α

```
 1: procedure COMPUTE_S($T, x, \alpha$)
 2:     if no $\alpha$-node exists in $T$ then
 3:         return 0;
 4:     end if
 5:     $u \leftarrow$ the $\alpha$-predecessor of $x$ in preorder;
 6:     if no such $u$ exists then
 7:         $u \leftarrow$ the root of $T$;
 8:     end if
 9:     $v \leftarrow$ the first $\alpha$-node that occurs after the last descendant of $x$;
10:     if no such $v$ exists then
11:         $v \leftarrow$ the root of $T$;
12:     end if
13:     $a \leftarrow \texttt{LCA}(u, x), b \leftarrow \texttt{LCA}(v, x)$;
14:     $c \leftarrow$ the deeper one of $a$ and $b$;
15:     $d \leftarrow$ the $\alpha$-descendant of $c$ that minimizes $\texttt{depth}(T_\alpha, d')$;
16:     $y' \leftarrow$ the lowest ancestor of $d'$ satisfying that $\texttt{depth}(y) \leq \texttt{depth}(c)$;
17:     $ans \leftarrow S_1(T_\alpha, y') + \texttt{deg}_\alpha(\texttt{parent}(y)) - \texttt{child\_rank}_\alpha(y)$;
18:     if $\texttt{parent}(x) \neq y$ then
19:         $z \leftarrow$ the child of $y$ in $T$ that is an ancestor of $x$;
20:         $ans \leftarrow ans + \texttt{deg}_\alpha(y) - \texttt{child\_rank}_\alpha(z)$;
21:     end if
22:     return $ans$;
23: end procedure
```

that occurs after all descendants of x, call them u and v, respectively. We set u or v to be the root of T if such an α-node does not exist. Then, in line 13, we compute the lowest common ancestor, a, of u and x, and the lowest common ancestor, b, of v and x. Like the analysis in Section 4.2.3, we claim that there is no node between a and x that is an α-node, or has an α-child preceding x in preorder. Also, there is no node between b and x that has an α-child succeeding x in preorder. Thus, provided that c is the deeper one of a and b, nodes between c and x (excluding c) do not belong to X_α.

Node c, which might not have a corresponding node in T_α, is very close to our objective, i.e., node y. We observe that c has at least one α-descendant. In line 15, we find node d, the α-descendant of c that minimizes $\texttt{depth}(T_\alpha, d')$, using $\texttt{min\_depth}_1$ on T_α. Note that the α-descendants of c are the i-th to the j-th α-nodes in preorder of T, where $i = j - \texttt{nbdesc}_\alpha(c) + 1$ and $j = \texttt{pre\_rank}_\alpha(\texttt{pre\_rank}(c) + \texttt{nbdesc}(c) - 1)$. These α-nodes correspond to the i-th to the j-th 1-nodes in T_α, so we have $d' = \texttt{min\_depth}_1(T_\alpha, i, j)$.

Then, y' can be found by checking at most three nodes in T_α, which are d', $d'_1 = \texttt{parent}(T_\alpha, d')$ and $d'_2 = \texttt{parent}(T_\alpha, d'_1)$. Suppose none of these nodes corresponds to c or an ancestor of c in T. In this case, d'_1 and d'_2 must be 0-nodes, and d'_2 must have an α-child, which is an α-descendant of c and corresponds to a 1-node in T_α that is less deep than d'. That contradicts our assumption on d'. Hence we conclude that y' must be among d', d'_1 and d'_2. This completes our algorithm.

4.2.8 The α-operations Related to Leaves

For each T_α, we maintain a bit vector $L_\alpha[1..n_\alpha]$ in which the i-th bit is one if and only if the i-th 1-node in preorder of T_α corresponds to a leaf in T, where n_α is the number of α-nodes in T. The α-operations related to leaves can be supported using $\texttt{rank}/\texttt{select}$ operations on L_α. To compute $\texttt{leaf\_select}_\alpha(i)$, we find the i-th 1-bit in L_α, and return the α-node in T that corresponds to this 1-bit. To compute $\texttt{leaf\_rank}_\alpha(x)$, we need only compute the number of 1-bits in $L_\alpha[1..i]$, where $i = \texttt{pre\_rank}_\alpha(x)$.

Now consider $\texttt{nbleaf}_\alpha(x)$, $\texttt{leaf\_lmost}_\alpha(x)$ and $\texttt{leaf\_rmost}_\alpha(x)$. The case in which x has no α-descendant is trivial: $\texttt{nbleaf}_\alpha(x)$ returns 0, while the other two operations return NULL. Otherwise, we compute the α-boundary, $[u, v]$, of the subtree rooted at x. We further compute $i = \texttt{pre\_rank}_1(T_\alpha, u')$ and $j = \texttt{pre\_rank}_1(T_\alpha, v')$. For $\texttt{nbleaf}_\alpha(x)$, we need only count the number of 1-bits in $L_\alpha[i..j]$. For $\texttt{leaf\_lmost}_\alpha(x)/\texttt{leaf\_rmost}_\alpha(x)$, we compute the first/the last 1-bit $L_\alpha[i..j]$ using $\texttt{rank}_1$ and $\texttt{select}_1$ on L_α, and return the α-node in T that corresponds to this bit.

4.2.9 Completing the Proof of Theorem 4.1.2

In the current state, we have σ 0/1-labeled trees and σ bit vectors. To reduce redundancy, we merge T_α's into a single tree $\mathcal{T}$, and merge L_α's into a single bit vector $\mathcal{L}$. We list the characters in Σ as $1, 2, \cdots, \sigma$. Initially, $\mathcal{T}$ contains a root node $\mathcal{R}$ only, on which the label is 0. Then, for $\alpha = 1$ to σ, we append r'_α, the root of T_α, to the list of children of $\mathcal{R}$. Let n_α be the number of α-nodes in T. For $\alpha \in \Sigma$, T_α has at most $2n_\alpha + 1$ nodes, since each α-node adds a 1-node and at most one 0-node into T_α. In addition, the T_α that corresponds to the label of the root of T has at most $2n_\alpha$ nodes, since the root does not add a 0-node to T_α. Hence, $\mathcal{T}$ has at most $2n + \sigma$ nodes in total. By the construction of $\mathcal{T}$, we have $r'_\alpha = \texttt{child\_select}(\mathcal{R}, \alpha)$ for $\alpha \in \Sigma$, the preorder/postorder traversal sequence of T_α occurs as a substring in the preorder/postorder traversal sequence of $\mathcal{T}$, and the DFUDS traversal sequence of T_α with r'_α removed occurs as a substring in the DFUDS traversal sequence of $\mathcal{T}$.

In addition, we append L_α to $\mathcal{L}$, which is initially empty, for $\alpha = 1$ to σ. The length of $\mathcal{L}$ is clearly n. It is not hard to verify that the reductions described in early sections can still be performed on the merged tree $\mathcal{T}$ and the merged bit vector $\mathcal{L}$. For example, we have $\texttt{dfuds\_rank}_\alpha(x) = \texttt{dfuds\_rank}_1(\mathcal{T}, x') - \texttt{pre\_rank}_1(\mathcal{T}, r'_\alpha)$, for α-node x in T, and x' in T_α that corresponds to x.

The following Assembling Lemma generalizes the discussion.

Lemma 4.2.1 (Assembling Lemma)**.** *Let T be an ordinal tree on n nodes, each having a label drawn from an alphabet $\Sigma = [1..\sigma]$. Suppose that there exist*

- *a data structure $\mathcal{D}_1$ that represents a unlabeled ordinal tree on n nodes using $\mathcal{S}_1(n)$ bits and supports the unlabeled versions of the α-operations in Groups 1 and 2 of Table 4.1;*
- *a data structure $\mathcal{D}_2$ that represents a string S using $\mathcal{S}_2(S)$ bits and supports $\texttt{rank}_\alpha$ and $\texttt{select}_\alpha$ for $\alpha \in \Sigma$;*
- *a data structure $\mathcal{D}_3$ that represents a 0/1-labeled ordinal tree on n nodes using $\mathcal{S}_3(n)$ bits and supports the α-operations in Groups 1 and 2 of Table 4.1 and their unlabeled versions, plus two additional α-operations $\texttt{deepest}_\alpha$ and $\texttt{min\_depth}_\alpha$;*
- *and a data structure $\mathcal{D}_4$ that represents a bit vector of length n using $\mathcal{S}_4(n)$ bits and supports $\texttt{rank}_\alpha$ and $\texttt{select}_\alpha$ for $\alpha \in \{0, 1\}$.*

Then there exists a data structure that encodes T using $\mathcal{S}_1(n) + \mathcal{S}_2(PLS_T) + \mathcal{S}_3(2n + \sigma) + \mathcal{S}_4(n)$ bits of space, supporting the α-operations in Groups 1 and 2 of Table 4.1 and their *unlabeled versions using a constant number of operations mentioned above on $\mathcal{D}_1$, $\mathcal{D}_2$, $\mathcal{D}_3$ and $\mathcal{D}_4$.*

Proof. We maintain the structure of T, PLS_T, $\mathcal{T}$, and $\mathcal{L}$ using $\mathcal{D}_1$, $\mathcal{D}_2$, $\mathcal{D}_3$, and $\mathcal{D}_4$, respectively. The reductions for the α-operations are shown in Sections 4.2.1 to 4.2.8, while their unlabeled versions are supported by $\mathcal{D}_1$ directly. The overall space cost is $\mathcal{S}_1(n) + \mathcal{S}_2(PLS_T) + \mathcal{S}_3(2n + \sigma) + \mathcal{S}_4(n)$ bits. □

It is notable that $\mathcal{D}_3$ need not support the α-operations related to leaves. With Lemmas 2.4.1, 2.3.1 and 4.2.2 the proof of Theorem 4.1.2 follows.

Lemma 4.2.2 ([43, 64, 82, 42]). *Let T be an ordinal tree on n nodes. T can be represented using $2n + o(n)$ bits such that the unlabeled versions of the α-operations in Groups 1 and 2 of Table 4.1 can be supported in constant time.*

Proof of Theorem 4.1.2. We obtain the conclusion by applying Lemma 4.2.2, one variant of Lemma 2.4.1 (a,b,c), Theorem 4.1.1, and Lemma 2.3.1 for $\mathcal{D}_1$, $\mathcal{D}_2$, $\mathcal{D}_3$, and $\mathcal{D}_4$, respectively. Here we only analyze variant (b) of Theorem 4.1.2; the other two variants can be analyzed similarly.

For variant (b), $\mathcal{D}_1$, $\mathcal{D}_2$, and $\mathcal{D}_4$ occupy $2n + o(n)$ bits, $nH_0(PLS_T) + o(nH_0(PLS_T)) + o(n)$ bits, and $n + o(n)$ bits of space, respectively. $\mathcal{T}$ consists of $2n + \sigma$ nodes, and hence $\mathcal{D}_3$ uses $6n + 3\sigma + o(n)$ bits of space. Summing up these costs, the overall space cost is up to $n(H_0(PLS_T) + 9) + 3\sigma + o(n(H_0(PLS_T) + 1))$ bits.

Most α-operations listed in Groups 1 and 2 of Table 4.1 require $O(\lg \frac{\lg \sigma}{\lg w})$ query time, because they rely on both $\texttt{rank}_\alpha$ and $\texttt{select}_\alpha$ operations of PLS_T. There are two exceptions: $\texttt{label}(x)$ and $\texttt{pre\_select}_\alpha(i)$. They correspond to $\texttt{access}$ and $\texttt{select}_\alpha$ on PLS_T, and only use $O(1)$ and $f(n, \sigma)$ query time, respectively, where $f(n, \sigma)$ is any function that satisfies $f(n, \sigma) = \omega(1)$ and $f(n, \sigma) = o(\lg \frac{\lg \sigma}{\lg w})$. □

4.3 Static Trees over Small Alphabets : Theorem 4.1.1

Now we pay off our technical debt and prove Theorem 4.1.1. The basic idea is to extend and improve Geary et al.'s [54] representation of labeled ordinal trees. As shown in Lemma 4.3.1, this representation is succinct only if the size of alphabet is small, i.e., $\sigma = o(\lg \lg n)$.

Lemma 4.3.1 ([54])**.** *Let T be an ordinal tree on n nodes, each having a label drawn from an alphabet Σ of size $\sigma = o(\lg\lg n)$. Then T can be encoded using $n(\lg\sigma+2)+O(\frac{n\sigma\lg\lg\lg n}{\lg\lg n})$ bits of space such that the α-operations marked by † in Table 4.1 and their unlabeled versions can be supported in constant time.*

Geary et al.'s approach is based on a two-level tree-covering decomposition. They proposed an algorithm to decompose T into $O(\frac{n}{M})$ *mini-trees*, each having at most $3M-4$ nodes, for $M = \max\{\lceil\lg^4 n\rceil, 2\}$. They further applied this algorithm to each mini-tree, decomposing each mini-tree into $O(\frac{M}{M'})$ *micro-trees* of sizes at most $3M'-4$, for $M' = \max\{\lceil\frac{\lg n}{24\lg\sigma}\rceil, 2\}$. Thus, any micro-tree can be encoded in $\frac{3}{4}\lg n$ bits, and any query in a single micro tree can be answered in constant time by table lookup using $o(n^{1-\epsilon})$ additional bits for some constant $\epsilon > 0$. The encodings of micro-trees are stored implicitly, occupying $n(\lg\sigma+2)+O(\frac{n}{M'})$ bits of space in total.

Geary et al. supported operations `pre/post_rank`, `pre/post_select`, `depth`, `parent`, `level_anc`, `deg`, `child_rank`, `child_select` and `nbdesc` using auxiliary data structures. These auxiliary structures occupy $O(\frac{n\lg\lg\lg n}{\lg\lg n})$ bits of space in total. To support the labeled versions of these operations, they duplicated the auxiliary data structures for each label $\alpha\in\Sigma$. Thus the overall space cost is $n(\lg\sigma+2)+O(\frac{n\sigma\lg\lg\lg n}{\lg\lg n})$ bits. Note that the term $O(\frac{n\sigma\lg\lg\lg n}{\lg\lg n})$ becomes $o(n\lg\sigma)$ when $\sigma = o(\lg\lg n)$.

He et al. [64] considered more operations such as `height`, `LCA`, `leaf_lmost`, `leaf_rmost`, `leaf_rank`, `leaf_select`, `nbleaf`, `dfuds_rank` and `dfuds_select`. Based on the same tree-covering decomposition, He et al. showed that one could support these operations using constant time and $O(\frac{n(\lg\lg n)^2}{\lg n})$ bits of additional space. It is not hard to verify that the labeled versions of these newer operations can also be supported by duplicating the auxiliary data structures.

We need support two more α-operations, $\texttt{deepest}_\alpha$ and $\texttt{min\_depth}_\alpha$. In fact, the first one, $\texttt{deepest}_\alpha(i,j)$, has been supported by the auxiliary data structures for $\texttt{height}_\alpha$, which can find the node with preorder rank in $[i..j]$ that has the maximum α-depth, for arbitrary $1\le i\le j\le n$. The second one, $\texttt{min\_depth}_\alpha(i,j)$, can also be supported with a slight modification to the auxiliary data structures for $\texttt{height}_\alpha$. We duplicate the array E used in He et al.'s work [64] for each $\alpha\in\Sigma$. We store in $E_\alpha[i]$ the τ_3-name of the α-node, $e_{\alpha,i}$, with minimum depth among the nodes between z_i and z_{i+1} (including z_i but excluding z_{i+1}) in preorder, where we have $O(\frac{n}{M'})$ z_i's that are specified by tree covering. The changes to array M and table A_1 are similar.

Finally, we improve the space cost of this extended representation. We obtain mini-trees and micro-trees using Farzan et al.'s [42] tree decomposition algorithm instead. This algorithm further ensures that nodes in a single micro-tree occupy $O(1)$ substrings in the preorder traversal sequence of T. Therefore, to store the encoding of a micro-tree, we can store the labels on the nodes in the preorder label sequence of T, which is compressed using $nH_k(PLS_T) + O(\frac{n(k \lg \sigma + \lg \lg n)}{\log_\sigma n})$ bits of space with Lemma 2.4.2.

The encoding of a micro-tree can still be retrieved in constant time, since its size is at most $3M' - 4$, which is $O(\log_\sigma n)$. This completes the proof of Theorem 4.1.1.

4.4 Dynamic Trees Supporting Level-Ancestor Operations : Theorem 4.1.3

In this section, we prove Theorem 4.1.3 by extending the main technique for static trees to the dynamic case. Our succinct representation for dynamic trees also consists of four components: (a) the structure of T; (b) PLS_T, the preorder label sequence of T; (c) an unlabeled tree T_α for each $\alpha \in \Sigma$; and (d) a bit vector L_α for each $\alpha \in \Sigma$. The previous work for these components will be summarized in Lemmas 2.4.3 and 4.4.2.

The construction of T_α and L_α is different in this scenario in order to facilitate update operations. For each $\alpha \in \Sigma$, we still add a new root r_α to T, and make T_α be the X_α-extraction of the augmented tree rooted at r_α. Unlike the construction described in Section 4.2, here X_α contains r_α and the α-nodes in T only, and we do not assign labels to the nodes in T_α. Thus T_α is a unlabeled ordinal tree on $n_\alpha + 1$ nodes, where n_α is the number of α-nodes in T. $L_\alpha[1..n_\alpha]$ is a bit vector in which the i-th bit is 1 if and only if the i-th non-root node in preorder of T_α corresponds to a leaf in T. Finally, we still merge T_α's into a single tree $\mathcal{T}$, and merge L_α's into a single bit vector $\mathcal{L}$. Clearly $\mathcal{T}$ contains $n + \sigma + 1$ nodes, and $\mathcal{L}$ is of length n. Keep in mind that any node x in T corresponds to exactly one node in $\mathcal{T}$, and exactly one element in PLS_T and $\mathcal{L}$.

The unlabeled versions of the α-operations in Group 1 of Table 4.1 can be directly supported by the data structures that maintain the structure of T. Like Section 4.2, any given α-operation will still be reduced to a constant number of operations on T, PLS_T, T_α's, and L_α's that were supported in previous work. For each operation, we specify the component on which it performs as the first parameter unless this component is T. We also denote by x' the only node in $\mathcal{T}$ that corresponds to x.

Operations $\texttt{label}$, $\texttt{pre\_rank}_\alpha$, $\texttt{pre\_select}_\alpha$ and $\texttt{nbdesc}_\alpha$ can still be supported as described in Section 4.2.1. Provided that x is an α-node, the conversion between x and x' can be simplified as follows:

$$\begin{aligned} x' &= \texttt{pre\_select}(T_\alpha, \texttt{pre\_rank}_\alpha(x) + 1) \\ x &= \texttt{pre\_select}_\alpha(\texttt{pre\_rank}(T_\alpha, x') - 1) \end{aligned}$$

Following the conversion, the other α-operations in Group 1 of Table 4.1 can be supported by the algorithms described in Sections 4.2.3, 4.2.6, and 4.2.8, with small changes.

As an example, we describe how to compute $\texttt{parent}_\alpha(x)$ in detail, which is similar to Algorithm 6. The case in which x is an α-node can be simply solved by computing $y' = \texttt{parent}(T_\alpha, x')$ and returning y. Suppose x is not an α-node. We still compute u as the α-predecessor of x in preorder. Following the same arguments as Section 4.2.3, we have that x has no α-parent if u does not exist. If such u does exist, we further claim that there is no α-node on the path from x to and excluding v, where $v = \texttt{LCA}(u, x)$. In addition, we know that v has at least one α-descendant because of the existence of u. We return v if v is an α-node. Otherwise, we compute the first α-descendant, z, of v. Clearly there is no α-node on the path from v to and excluding z. Thus, the α-parent of z must be the α-parent of both v and x.

The remaining part is to support updates, for which we assume the same updating protocol as Navarro and Sadakane [82]. This protocal allows us to insert a new leaf, a new root, or a new internal node that will become the parent of consecutive children of an existing node. The location of the new node is specified by an existing node y and its descendants u and v. This means that, after the insertion, x will be a child of y and an ancestor of the nodes whose preorder rank is between $\texttt{pre\_rank}(u)$ to $\texttt{pre\_rank}(v)$. Note that $y = \texttt{NULL}$ if x will be the root, and $u = v = \texttt{NULL}$ if x will be a leaf. Similarly, it is allowed to delete a leaf node, an internal node, or the root node when it has only zero or one child. Once an internal node is deleted, its children will become children of its parent. It should be noted that these updates preserve the ancestor-descendant and preorder/postorder relationships among the remaining nodes in T and T_α.

To insert a node x with label α, we first consider how to insert its correponding node, x', into T_α. Let y and z be the nodes that will be the parent and the α-parent of x after the insertion, respectively. Note that y is given to locate x, and z is equal to either y or $\texttt{parent}_\alpha(y)$. Also, we compute a and b, the α-successor of u and the α-predecessor of v, respectively. Thus, x' is inserted into T_α as a child of z' and an ancestor of the nodes from a' to b'. This is performed over tree $\mathcal{T}$, and the support for required operartions will be given in Lemma 4.4.2. Then, we insert x into the data structures that maintain the structure of T, which will be provided in Lemma 4.4.2. After that, we add the corresponding element into PLS_T by performing $\texttt{insert}_\alpha(PLS_T, \texttt{pre\_rank}(x))$. Finally, we perform $\texttt{insert}_1(L_\alpha, \texttt{pre\_rank}_\alpha(T, x))$ if x is a leaf node, or $\texttt{insert}_0(L_\alpha, \texttt{pre\_rank}_\alpha(T, x))$ otherwise.

To delete a node x, we first retrieve its label $\alpha = \texttt{label}(x)$. Thus x corresponds to the $\texttt{pre\_rank}(x)$-th element in PLS_T, the $\texttt{pre\_rank}_\alpha(x)$-th element in L_α, and the $(\texttt{pre\_rank}_\alpha(x) + 1)$-st node in preorder of T_α. We simply delete these elements or nodes, and finally remove x from the structure of T.

Generalizing the discussion above, we obtain the dynamic version of the Assembling Lemma.

Lemma 4.4.1 (Dynamic Assembling Lemma). *Let T be a dynamic ordinal tree on n nodes, each having a label drawn from an alphabet $\Sigma = [1..\sigma]$. Suppose that there exist*

- *a data structure $\mathcal{D}_1$ that represents an unlabeled ordinal tree on n nodes using $\mathcal{S}_1(n)$ bits and supports the unlabeled versions of the α-operations in Groups 1 and 3 of Table 4.1;*
- *a data structure $\mathcal{D}_2$ that represents a string S using $\mathcal{S}_2(S)$ bits and supports* `access`*,* $\texttt{rank}_\alpha$*,* $\texttt{select}_\alpha$*,* $\texttt{insert}_\alpha$ *and* `delete` *for $\alpha \in \Sigma$;*
- *and a data structure $\mathcal{D}_3$ that represents a bit vector of length n using $\mathcal{S}_3(n)$ bits and supports* `access`*,* $\texttt{rank}_\alpha$*,* $\texttt{select}_\alpha$*,* $\texttt{insert}_\alpha$ *and* `delete` *for $\alpha \in \{0, 1\}$.*

Then there exists a data structure that encodes T using $\mathcal{S}_1(n) + \mathcal{S}_2(PLS_T) + \mathcal{S}_1(n + \sigma + 1) + \mathcal{S}_3(n)$ bits of space, supporting the α-operations in Groups 1 and 3 of Table 4.1 and their unlabeled versions using a constant number of operations mentioned above on $\mathcal{D}_1$, $\mathcal{D}_2$, and $\mathcal{D}_3$.

Proof. We maintain the structure of T, PLS_T, $\mathcal{T}$, and $\mathcal{L}$ using $\mathcal{D}_1$, $\mathcal{D}_2$, $\mathcal{D}_1$, and $\mathcal{D}_3$, respectively. The reductions for the α-operations are shown in Sections 4.2.1 to 4.2.3, 4.2.6, and 4.2.8. The unlabeled versions of these α-operations are supported by $\mathcal{D}_1$ directly. The overall space cost is $\mathcal{S}_1(n) + \mathcal{S}_2(PLS_T) + \mathcal{S}_1(n + \sigma) + \mathcal{S}_3(n)$ bits. □

The proof of Theorem 4.1.3 follows with Lemmas 2.4.3 and 4.4.2.

Lemma 4.4.2 ([82]). *Let T be an ordinal tree on n nodes. Then T can be represented using $2n + o(n)$ bits of space such that* `level_anc` *can be supported in $O(\lg n)$ worst-case time, and the unlabeled versions of the other α-operations in Groups 1 and 3 of Table 4.1 can be supported in $O(\frac{\lg n}{\lg \lg n})$ worst-case time.*

Proof of Theorem 4.1.3. Applying Lemma 4.4.2 for $\mathcal{D}_1$, and Lemma 2.4.3 for $\mathcal{D}_2$ and $\mathcal{D}_3$, respectively, we obtain the conclusion. □

4.5 Discussion

We have presented a simple and novel framework for succinct representations of labeled ordinal trees, with which we have obtained new static and dynamic data structures. Our static data structures provide richer functionality than previous ones [54, 44, 9]. More importantly, our static data structures presented in Theorem 4.1.2 support the α-operations in Groups 1 and 2 of Table 4.1 in time log-logarithmic in σ over large alphabets. This answers an open problem proposed by Geary et al. [54], which asks for representations of labeled ordinal trees that remain space-efficient for large alphabets .

Our dynamic structures support the α-operations in Groups 1 and 3 of Table 4.1 in time sub-logarithmic in each of n and σ, most of which are in worst-case. Though not as functional as the static ones, our dynamic data structures for labeled ordinal trees are the first ones with efficient query and update time.

As shown in Lemmas 4.2.1 and 4.4.1, our data structures in Theorems 4.2 and 4.1.3 are ready-to-assemble. One can simply replace the auxiliary data structures for bit vectors, preorder label sequences, unlabeled and 0/1-labeled ordinal trees with improved results. Thus any improvement on these simpler data structures immediately improves our representations of labeled ordinal trees.

In addition, the bit vector $\mathcal{L}$ can be dropped to save $n + o(n)$ bits of space if the α-operations related to leaves are not required.

We end this chapter with two open problems. The first one is to reduce the coefficient of linear terms in the space cost, which is important when the empirical entropy of the preorder label sequence is small. The second question is to support more α-operations in dynamic representations, especially for $\texttt{deg}_\alpha$, $\texttt{child\_rank}_\alpha$, and $\texttt{child\_select}_\alpha$.

Very recently, Tsur [96] provided an almost complete answer to the first open question. He showed that, (a) for $\sigma = w^{O(1)}$, one can encode a labeled ordinal tree T using $nH_0(PLS_T) + 2n + o(n)$ bits of space, supporting the α-operations marked by † in Table 4.1 using $f(n, \sigma)$ time; and (b) for $\sigma \leq n$, one can encode a labeled ordinal tree T using $nH_0(PLS_T) + 2n + o(nH_0(PLS_T))$ bits of space, supporting $\texttt{label}$ using $O(1)$ time, $\texttt{pre\_rank}_\alpha$ using $f(n, \sigma)$ time, and the rest of the α-operations marked by † in Table 4.1 using $O(\lg \frac{\lg \sigma}{\lg w})$ time. Here T is of size n, labels are drawn from $\Sigma = [1..\sigma]$, and $f(n, \sigma)$ is a function satisfying $f(n, \sigma) = \omega(1)$ and $f(n, \sigma) = o(\lg \frac{\lg \sigma}{\lg w})$. Tsur's data structures cannot support $\texttt{height}_\alpha$, $\texttt{dfuds\_rank}_\alpha$ or $\texttt{dfuds\_select}_\alpha$, and need $n + o(n)$ bits of additional space to support the α-operations related to leaves. His approach is based on ours but with intensive changes.

Chapter 5

Static Succinct Indices for Path Minimum, with Applications

5.1 Introduction

In this chapter, we first consider the path minimum (path maximum) problem and then the more general semigroup path sum problem.

- Path minimum (maximum): Given nodes u and v, return the minimum (maximum) node along the path from u to v, i.e., the node along the path whose weight is the minimum (maximum) one;

- Semigroup path sum: Given nodes u and v, return the sum of weights along the path from u to v, where the weights of nodes are drawn from a semigroup.

We design novel succinct data structures for these two types of path queries. Then we revisit the problem of supporting path reporting queries.

- Path reporting: Given nodes u and v along with a two-sided query range, report the nodes along the path from u to v whose weights are in the query range.

The indexing structures for path minimum queries will play a central role in our approach to path reporting queries.

When the input tree is a single path, path minimum and semigroup path sum queries become range minimum [47, 38] and semigroup range sum [99] queries, respectively. In addition, as mentioned in Chapter 3, path reporting queries become two-dimensional orthogonal range reporting queries [27]. The path queries we consider generalize these fundamental range queries to weighted trees.

In this chapter, we represent the input tree as an ordinal one, and use ϵ to denote a constant in $(0, 1)$. Unless otherwise specified, the underlying model of computation is the standard word RAM model with word size $w = \Omega(\lg n)$.

To present our results, we assume the following definition for the Ackermann function. For integers $\ell \geq 0$ and $i > 1$, we have

$$A_\ell(i) = \begin{cases} i+1 & \text{if } \ell = 0, \\ A_{\ell-1}^{(i+1)}(i+13) & \text{if } \ell > 0, \end{cases}$$

where $A_{\ell-1}^{(0)}(i) = i$ and $A_{\ell-1}^{(i)}(j) = A_{\ell-1}(A_{\ell-1}^{(i-1)}(j))$ for $i \geq 1$. This is growing faster than the one defined by Cormen et al. [37]. Let $\boldsymbol{\alpha}(m, n)$ be the smallest L such that $A_L(\lfloor m/n \rfloor) > n$, and $\boldsymbol{\alpha}(n)$ be $\boldsymbol{\alpha}(n, n)$. Here $\boldsymbol{\alpha}(m, n)$ and $\boldsymbol{\alpha}(n)$ are both referred to as the inverse-Ackermann functions, and are of the same order as the ones defined by Cormen et al. [37].

5.1.1 Path Minimum

The *minimum spanning tree verification* problem asks whether a given spanning tree is minimum with respect to a graph with weighted edges. This problem can be regarded as a special offline case of the path minimum problem, for which all the queries are processed in a single batch. Under the word RAM model [73], this problem can be solved using $O(n + m)$ comparisons and linear overhead, where n and m are the numbers of nodes and edges, respectively. See [25, 35, 40, 74] for other results under different models. The online path minimum problem requires slightly more comparisons. As shown by Pettie [91], $\Omega(q \cdot \boldsymbol{\alpha}(q, n) + n)$ comparisons are necessary to serve q queries over a tree of size n.

Data structures for the path minimum problem have been heavily studied. An early result presented by Alon and Schieber [2] requires $O(n)$ words of space and $O(\boldsymbol{\alpha}(n))$ query time. Since then, several solutions using $O(n)$ words, i.e., $O(n \lg n)$ bits, with constant query time have been designed under the word RAM model [5, 21, 31, 38, 72]. Chazelle [31] and Demaine et al. [38] generalized Cartesian trees [97] to weighted trees and used them to support path minimum queries. Alstrup and Holm [5] and Brodal et al. [21] made use of macro-micro decomposition in designing their data structures. The solution of Kaplan and Shafrir [72] is based on Gabow's recursive decomposition of trees [52].

In this chapter we present lower and upper bounds for path minimum queries. In

Lemma 5.2.1 we show that $\Omega(n \lg n)$ bits of space are necessary to encode the answers to path minimum queries over a tree of size n. This distinguishes path minimum queries from range minimum queries in terms of space cost, for which $2n$ bits are sufficient to encode all answers over an array of size n [47].

We adopt the *indexing model* (also called the *systematic model*) [10, 22, 19] in designing new data structures for path minimum queries. Applying this model to weighted trees, we assume that weights of nodes are represented in an arbitrary given form; the only requirement is that the representation supports access to the weight of a node given its preorder rank, i.e., the rank of the node in the preorder traversal sequence of the weighted tree. Auxiliary data structures called *indices* are then constructed, and query algorithms use indices and the access operator provided for the raw data. This model is theoretically important and its variants are frequently used to prove lower bounds [39, 78, 56]. In addition, the indexing model is also of practical importance as it addresses cases in which the (large) raw data are stored in slower external memory or even remotely, while the (smaller) indices could be stored in memory or locally. The space of an index is called *additional space*. Note that the lower bound in the previous paragraph is proved under the encoding model, and thus does not apply to the indexing model.

The following theorem presents our indices for path minimum.

Theorem 5.1.1. *An ordinal tree on n weighted nodes can be indexed (a) using $O(m)$ bits of space and $O(m)$ construction time to support path minimum queries in $O(\boldsymbol{\alpha}(m,n))$ time and $O(\boldsymbol{\alpha}(m,n))$ accesses to the weights of nodes, for any integer $m \geq n$; or (b) using $2n + o(n)$ bits of space and $O(n)$ construction time to support path minimum queries in $O(\boldsymbol{\alpha}(n))$ time and $O(\boldsymbol{\alpha}(n))$ accesses to the weights of nodes.*

To better understand variant (a) of this result, we discuss the time and space costs for the following possible values of m. When $m = n$, then we have an index of $O(n)$ bits that supports path minimum queries in $O(\boldsymbol{\alpha}(n))$ time. When $m = \Theta(n(\lg^*)^*n)$, for example, then it is well-known that $\boldsymbol{\alpha}(m,n) = O(1)$, and thus we have an index of $O(n(\lg^*)^*n)$ bits that supports path minimum queries in $O(1)$ time[1]. Combining the above index with a trivial encoding of node weights, we obtain data structures for path minimum queries with $O(1)$ query time and almost linear bits of additional space. Previous solutions [5, 21, 31, 38, 72] to the same problem with constant query time occupy $\Omega(n \lg n)$ bits of space in addition to the space required for the input tree.

Taking the construction time into account, variant (a) with $m = \max\{q, n\}$ gives us a data structure that answers q path minimum queries in $O(q \cdot \boldsymbol{\alpha}(q,n) + \max\{q,n\}) =$

[1] The function $(\lg^*)^*$ is the number of times $\lg^*$ must be iteratively applied before the result becomes less than or equal to 1. See Nivasch's discussions [84] for more details.

$O(q \cdot \boldsymbol{\alpha}(q,n) + n)$ time, which matches the lower bound of Pettie [91].

Finally, variant (b) gives us the first succinct data structure for path minimum queries, which occupies an amount of space that is close to the information-theoretic lower bound of storing a weighted tree. Let σ denote the number of distinct weights. With a little extra work, we can even represent a weighted tree using $n \lg \sigma + 2n + o(n)$ bits only, i.e., within $o(n)$ additive term of the information-theoretic lower bound, to support queries in $O(\boldsymbol{\alpha}(n))$ time.

5.1.2 Semigroup Path Sum

Generalizing the semigroup range sum queries on linear lists [99], the problem of supporting semigroup path sum queries has been considered by Alon and Schieber [2] and Chazelle [31]. Alon and Schieber designed a data structure with $O(n)$ words of space and construction time that supports semigroup path sum queries in $O(\boldsymbol{\alpha}(n))$ time. Unlike Alon and Schieber's and our formulation, Chazelle considered trees on weighted edges instead of weighted nodes. However, it is not hard to see that these two formulations are equivalent. Chazelle further showed that, for any $m \geq n$, one could obtain a word RAM data structure with $O(\boldsymbol{\alpha}(m,n))$ query time in addition to $O(m)$ construction time and words of space. The solution of Chazelle is optimal, as established in the lower bound of Yao [99].

Our data structures for semigroup path sum queries are summarized in the following theorem.

Theorem 5.1.2. *Let T be an ordinal tree on n nodes, each having a weight drawn from a semigroup of size σ. Then T can be stored (a) using $m \lg \sigma + 2n + o(n)$ bits of space and $O(m)$ construction time to support semigroup path sum queries in $O(\boldsymbol{\alpha}(m,n))$ time, for some constant $c > 1$ and any integer $m \geq cn$; or (b) using $n \lg \sigma + 2n + o(n \lg \sigma)$ bits of space and $O(n)$ construction time to support semigroup path sum queries in $O(\boldsymbol{\alpha}(n))$ time.*

Variant (a) matches the data structures of Chazelle [31], and our approach can be further used to achieve variant (b), which is the first succinct data structure with near-constant query time for the semigroup path sum problem. Since path minimum queries are special cases of semigroup path sum queries, the data structures described in Theorem 5.1.2 can be directly used for path minimum queries at no extra cost. However, these structures cannot achieve both linear space and constant query time.

5.1.3 Path Reporting

Path reporting queries have been studied in Chapter 3. As stated in Theorem 3.5.8, we have designed a succinct data structure based on tree extraction. This structure, requiring $O((\lg\sigma/\lg\lg n+1)\cdot(1+occ))$ query time and $nH(W_T)+2n+o(n\lg\sigma)$ bits of space, where $H(W_T)$ is the entropy of the multiset of the weights of the nodes in the input tree T, is the best previously known linear space solution. Concurrently, Patil et al. [85] designed a succinct structure based on heavy path decomposition [94, 60]. Their structure requires $n\lg\sigma+6n+o(n\lg\sigma)$ bits and $O(\lg\sigma\lg n+occ\cdot\lg\sigma)$ query time.

In this chapter, we design three new data structures for path reporting queries. These results are included in this chapter, because they are heavily relying on the succinct indices for path minimum queries which we will describe in Section 5.2.2.

Theorem 5.1.3. *An ordinal tree on n nodes whose weights are drawn from a set of σ distinct weights can be represented using $O(n\lg\sigma\cdot\mathtt{s}(\sigma))$ bits of space, so that path reporting queries can be supported using $O(\min\{\lg\lg\sigma+\mathtt{t}(\sigma),\lg\sigma/\lg\lg n+1\}+occ\cdot\min\{\mathtt{t}(\sigma),\lg\sigma/\lg\lg n+1\})$ query time, where occ is the size of output, ϵ is an arbitrary positive constant, and $\mathtt{s}(\sigma)$ and $\mathtt{t}(\sigma)$ are: (a) $\mathtt{s}(\sigma)=O(1)$ and $\mathtt{t}(\sigma)=O(\lg^\epsilon\sigma)$; (b) $\mathtt{s}(\sigma)=O(\lg\lg\sigma)$ and $\mathtt{t}(\sigma)=O(\lg\lg\sigma)$; or (c) $\mathtt{s}(\sigma)=O(\lg^\epsilon\sigma)$ and $\mathtt{t}(\sigma)=O(1)$.*

These results completely subsume almost all previous results; the only exceptions are the succinct data structures for this problem designed in previous work, whose query times are worse than our linear-space solution. Furthermore, our data structures match the state of the art of 2D range reporting queries [27] when $\sigma=n$, and have better performance when σ is much less than n. We compare our results with previous work on path reporting and two-dimensional orthogonal range reporting in Table 5.1.

5.1.4 An Overview of the Chapter

The rest of this chapter is organized as follows. In Sections 5.2 and 5.3, we design novel succinct data structures for the path minimum problem and the semigroup path sum problem. Unlike previous succinct tree structures [54, 85, 64, 42] and the structure described in Lemma 3.5.5, our approach is based on Frederickson's restricted multilevel partitions described in Section 2.5.4, which transform the input tree into a binary tree and further recursively decompose it into a hierarchy of clusters with constant external degrees and logarithmically many levels. The hierarchy is referred to as a directed topology tree. Our main strategy of constructing query-answering structures is to recursively divide the set

Table 5.1: Our data structures for path reporting queries, along with previous results on path reporting and two-dimensional orthogonal range reporting (which are marked by †). All of these results assume the standard word RAM model with word size $w = \Omega(\lg n)$. Here $H(W_T)$ is the entropy of the multiset of the weights of the nodes in T. Note that $H(W_T)$ is at most $\lg \sigma$, which is $O(w)$.

Source	Space	Query Time
Theorem 3.4.1	$O(n)$ words	$O(\lg \sigma \cdot (1 + occ))$
Theorem 3.5.8	$nH(W_T) + 2n + o(n \lg \sigma)$ bits	$O((\lg \sigma / \lg \lg n + 1) \cdot (1 + occ))$
[85]	$n \lg \sigma + 6n + o(n \lg \sigma)$ bits	$O(\lg \sigma \lg n + occ \lg \sigma)$
[18]†	$n \lg \sigma + o(n \lg \sigma)$ bits	$O((\lg \sigma / \lg \lg n + 1) \cdot (1 + occ))$
[27]†	$O(n)$ words	$O(\lg \lg n + \lg^{\epsilon} \sigma + occ \lg^{\epsilon} \sigma)$
[27]†	$O(n \lg \lg \sigma)$ words	$O(\lg \lg n + occ \lg \lg \sigma)$
[27]†	$O(n \lg^{\epsilon} \sigma)$ words	$O(\lg \lg n + occ)$
Theorem 5.1.3	$O(n \lg \sigma)$ bits	$O(\min\{\lg^{\epsilon} \sigma, \lg \sigma / \lg \lg n + 1\} \cdot (1 + occ))$
Theorem 5.1.3	$O(n \lg \sigma \lg \lg \sigma)$ bits	$O(\min\{\lg \lg \sigma, \lg \sigma / \lg \lg n + 1\} \cdot (1 + occ))$
Theorem 5.1.3	$O(n \lg^{1+\epsilon} \sigma)$ bits	$O(\min\{\lg \lg \sigma, \lg \sigma / \lg \lg n + 1\} + occ)$

of levels of hierarchy into multiple subsets of levels; with a carefully-defined variant of the query problem which takes levels in the hierarchy as parameters, the query over the entire structure can be answered by conquering the subproblems local to the subsets of levels. Solutions to special cases of the query problem are also designed, so that we can present the time and space costs of our solution using recursive formulas. Then, by carefully constructing number series and using them in the division of levels into subsets, we can prove that our structures achieve the tradeoff presented in Theorems 5.1.1 to 5.1.2 using the inverse-Ackermann function. This approach is novel and exciting in the design of succinct data structures, and it does not directly use standard techniques for word RAM at all.

The above strategy would not achieve the desired space bound without a succinct data structure that supports navigation in the input tree, the binary tree that it is transformed into and the clusters in the directed topology tree. In Section 5.4, we design such a structure occupying only $2n + o(n)$ bits, which is of independent interest.

In Sections 5.5 and 5.6, to design solutions to path reporting, we follow the general framework described in Chapter 3 to extract subtrees based on the partitions of the entire weight range, and make use of a conceptual structure that borrows ideas from the classical range tree. One strategy of achieving improved results is to further reduce path reporting into queries in which the weight ranges are one-sided, which allows us to apply our succinct index for path minimum queries to achieve improved tradeoffs where query time and space cost match the previous best bounds for two-dimensional orthogonal range reporting as listed in Table 5.1. We further apply a tree covering strategy to reduce the space cost for the case in which the number of distinct weights is much smaller than n, and hence prove Theorem 5.1.3.

Finally, we end this chapter with some open problems in Section 5.7.

5.2 Path Minimum Queries

5.2.1 A Lower Bound under the Encoding Model

We first give a simple lower bound for path minimum queries under the *encoding model*, i.e., the least number of bits required to encode the answers to all possible queries.

Lemma 5.2.1. *In the worst case, $\Omega(n \lg n)$ bits are required to encode the answers to all possible path minimum queries over a tree on n weighted nodes.*

Proof. Consider a tree T with $n_L = \Theta(n)$ leaves. We assign the smallest n_L distinct weights to these leaves, and assign larger weights to the other nodes. It follows that the smallest weight on any path from a leaf to another must appear at one of its endpoints. The order of the weights assigned to leaf nodes, which requires $\lg(n_L!) = \Omega(n \lg n)$ bits to encode, can be fully recovered using path minimum queries. Therefore, $\Omega(n \lg n)$ bits are necessary to encode the answers to path minimum queries over T. □

While the lower bound of Pettie [91] focuses on the overall processing time, Lemma 5.2.1 provides a separation between path minimum and range minimum in terms of space: $\Omega(n \lg n)$ bits are required to encode path minimum queries over a tree on n weighted nodes, while range minimum over an array of length n can always be encoded in $2n$ bits [47].

5.2.2 Upper Bounds under the Indexing Model

Now we consider the support for path minimum queries. We represent the input tree as an ordinal one, for which the nodes are identified by their preorder ranks. This strategy has no significant impact to the space cost. We will assume the indexing model described in Section 5.1 and develop several novel succinct indices for path minimum queries. In these data structures, the weights of nodes are assumed to be stored separately from the index for queries, and can be accessed with the preorder ranks of nodes. The time cost to answer a given query is measured by the number of accesses to the index and that to node weights.

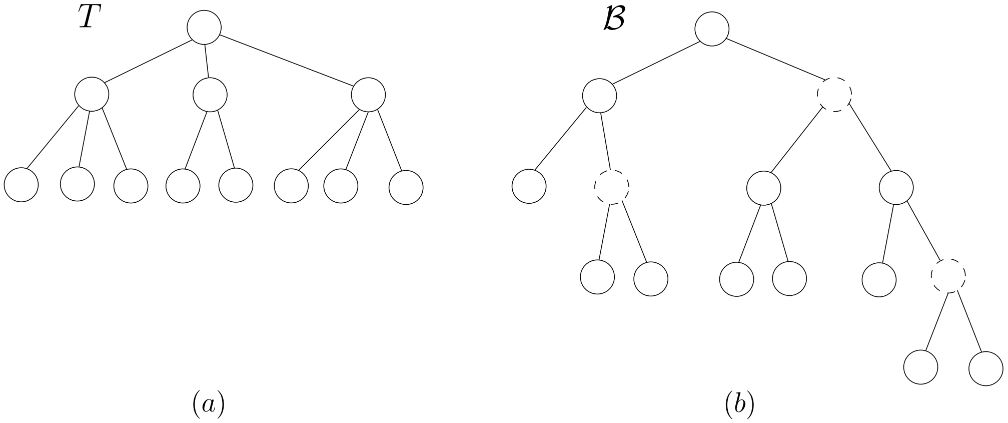


Figure 5.1: An illustration of the binary tree transformation. (a) An input tree T on 12 nodes. (b) The transformed binary tree $\mathcal{B}$, where dummy nodes are represented by dashed circles.

Let T be an input tree on n nodes. Here T is represented as an ordinal one, and its nodes are identified by preorder ranks. As illustrated in Figure 5.1, we transform T into a binary tree, $\mathcal{B}$, of size at most $2n$ as follows (essentially as in the usual way but with added dummy nodes): For each node u with $d > 2$ children, where $v_1, v_2, \ldots, v_d$ are children of u, we add $d - 2$ dummy nodes $x_1, x_2, \ldots, x_{d-2}$. The left and right children of u are set to be v_1 and x_1, respectively. For $1 \leq k < d - 2$, the left and right children of x_k are set to be v_{k+1} and x_{k+1}, respectively. Finally, the left and right children of x_{d-2} are set to be v_{d-1} and v_d, respectively. In this way we have replaced u and its children with a right-leaning binary tree, where the leaf nodes are children of u. This transformation does not change the preorder relationship among the nodes in T. In addition, the set of non-dummy nodes along the path between any two non-dummy nodes remain the same after transformation.

As described in Section 2.5.4, we decompose $\mathcal{B}$ and obtain the directed topology tree $\mathcal{D}$ using a restricted multilevel partition of order $s = 1$. We summarize the properties for this special case in Lemma 5.2.2.

Lemma 5.2.2 ([50]). *A binary tree $\mathcal{B}$ on n nodes can be partitioned into a hierarchy of clusters with $h + 1$ levels for some $h = O(\lg n)$, such that,*

- *the clusters at level 0 each contain a single node, and the only cluster at level h contains all the nodes of $\mathcal{B}$;*

- *for each level $\ell > 0$, each cluster at level ℓ is the disjoint union of at most 2 clusters at level $\ell - 1$;*
- *for each level $\ell = 0, 1, 2, \ldots, h$, there are at most $(5/6)^i n$ clusters of sizes at most 2^ℓ, which form a partition of the nodes in the binary tree;*
- *each cluster is of external degree at most 3 and contains at most two boundary nodes; and*
- *any cluster that has more than one child cluster contains only a single node.*

The leaf nodes of $\mathcal{D}$, which are at level 0, each corresponds to an individual node of the binary tree $\mathcal{B}$. We illustrate these concepts in Figure 5.2.

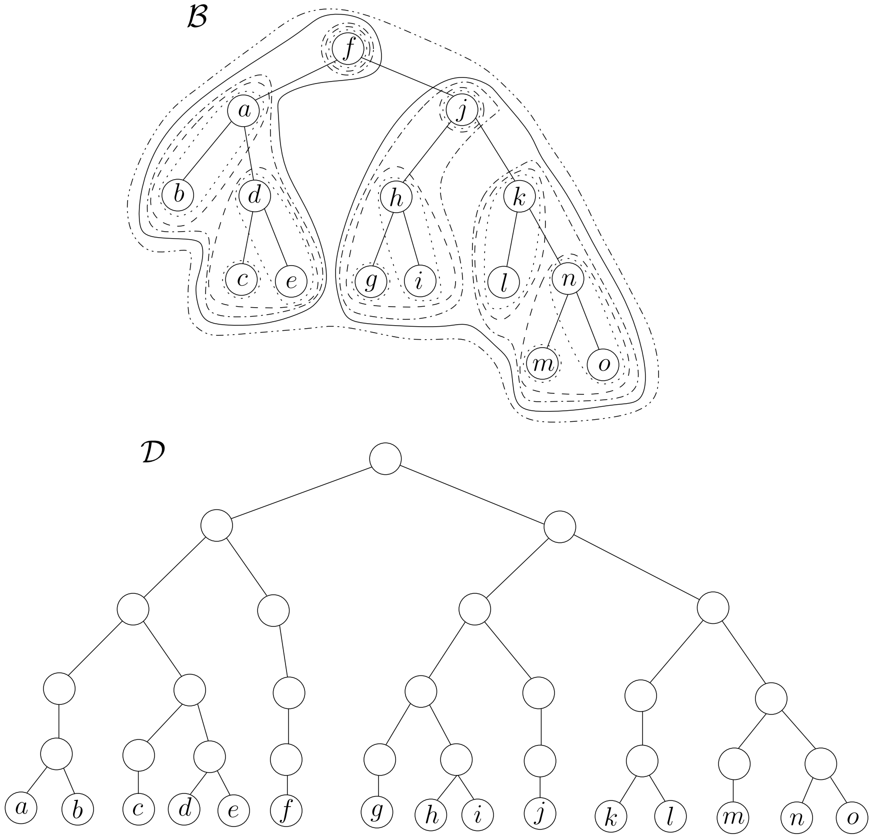


Figure 5.2: The restricted multilevel partitions and the directed topology tree $\mathcal{D}$ for the binary tree $\mathcal{B}$ shown in Figure 5.1(b). The base clusters, which each contain a single node of $\mathcal{B}$, are not explicitly specified in the figure to avoid cluttering.

As T and $\mathcal{B}$ are rooted trees, each cluster contains a node that is the ancestor of all the other nodes in the same cluster. This node is referred to as the *head* of the cluster. Note that the head of a cluster is also a boundary node except for the cluster that includes the root of $\mathcal{B}$. For a cluster that has two boundary nodes, the non-head one is referred to as the *tail* of the cluster. If the head and the tail of a cluster are not adjacent, then the path between but excluding them is said to be the *spine* of the cluster, i.e., the spine is obtained by removing the head and the tail from the path that connects them.

In the directed topology tree $\mathcal{D}$, sibling clusters are ordered by the preorder ranks of their heads. Each cluster C is identified by its *topological rank*, i.e., the preorder rank of the node in $\mathcal{D}$ that represents C. For simplicity, a cluster at level i is called a *level-i cluster*, and its boundary nodes are said to be *level-i boundary nodes*. To facilitate the use of directed topology trees, we define the following operations relevant to nodes, clusters, boundary nodes and spines. The support for these operations is summarized in the following lemma.

Lemma 5.2.3. *Let T be an ordinal tree on n nodes. Then T, the transformed binary tree $\mathcal{B}$, and their directed topology tree $\mathcal{D}$ can be encoded using $O(n)$ construction time and $2n + o(n)$ bits of space, such that the following operations can be supported in $O(1)$ query time. Here x and y are nodes in $\mathcal{B}$, and C is a cluster in $\mathcal{D}$.*

- *conversions between nodes in T and $\mathcal{B}$;*
- `level_cluster`$(\mathcal{D}, i, x)$: *return the level-i cluster that contains node x;*
- `LLC`$(\mathcal{D}, x, y)$: *return the cluster at the lowest level that contains nodes x and y;*
- `cluster_head`$(\mathcal{D}, C)$: *return the head of cluster C;*
- `cluster_tail`$(\mathcal{D}, C)$: *return the tail of cluster C or* `NULL` *if it does not exist;*
- `cluster_spine`$(\mathcal{D}, C)$: *return the endpoints of the spine of cluster C or* `NULL` *if the spine does not exist;*
- `cluster_nn`$(\mathcal{D}, C, x)$: *return the boundary node of C that is the closest to node x, given that x is outside of C;*
- `parent`$(\mathcal{B}, x)$: *return the parent node of x in $\mathcal{B}$;*
- `LCA`$(\mathcal{B}, x, y)$: *return the lowest common ancestor of x and y;*
- `BN_rank`$(\mathcal{B}, i, x)$: *count the level-i boundary nodes that precede x in preorder of $\mathcal{B}$;*
- `BN_select`$(\mathcal{B}, i, j)$: *return the j-th level-i boundary node in preorder of $\mathcal{B}$.*

Next we describe our data structures for path minimum queries. To highlight our key strategy, we defer the proof of Lemma 5.2.3 to Section 5.4. As the conversion between nodes in T and $\mathcal{B}$ can be performed in $O(1)$ time, we assume that the endpoints of query paths and the minimum nodes are both specified by nodes in $\mathcal{B}$. We first consider how to find the minimum node for specific subsets of paths in $\mathcal{B}$. Let h be the highest level of $\mathcal{D}$. The following subproblems are defined in terms of clusters and boundary nodes, for $0 \leq i < j \leq h$.

- $\mathcal{PM}_{i,j}$: find minimum nodes along query paths between two level-i boundary nodes that are contained in the same level-j cluster;
- $\mathcal{PM}'_{i,j}$: find minimum nodes along query paths from a level-i boundary node to a level-j one (which is also a level-i boundary node), where both boundary nodes are contained in the same level-j cluster.

Thus the original problem is $\mathcal{PM}_{0,h}$. If $\mathcal{PM}_{i,j}$ is solved, then $\mathcal{PM}'_{i,j}$ and $\mathcal{PM}_{i',j}$ for $i' > i$ are also naturally solved.

We will select a set of *canonical* paths in $\mathcal{B}$, for which the minimum nodes on all these canonical paths have been precomputed and stored. By the indexing model we adopt, singleton paths are naturally canonical. In our query algorithm, each query path will always be *partitioned* into a set of *canonical subpaths*, so that each node on the query path is contained in exactly one of these canonical subpaths, and the endpoints of the query path are contained in singleton canonical paths. Let $u \sim v$ denote the path from u to v. If node t is on $u \sim v$, then the partition of $u \sim v$ could be obtained by taking the union of the partitions of $u \sim t$ and $v \sim t$, which both contain t in a singleton canonical path.

Let $h_\tau > 0$ be a parameter whose value will be determined later. For each cluster whose level is higher than or equal to h_τ, we explicitly store the minimum node on its spine, i.e., the spine is made canonical. The following lemma addresses the cost incurred.

Lemma 5.2.4. *It requires $O(h_\tau(5/6)^{h_\tau}n)$ bits of additional space and $O(n)$ construction time to make these spines canonical.*

Proof. It requires i bits to store the minimum node on the spine of a level-i cluster, as the cluster contains at most 2^i nodes. As $\mathcal{B}$ has at most $2n$ nodes, there are at most $(5/6)^i \cdot 2n$ level-i clusters. Thus the overall space cost is $\sum_{i=h_\tau}^{h}(i(5/6)^i \cdot 2n) = O(h_\tau(5/6)^{h_\tau}n)$ bits.

The minimum nodes on the spines of level-h_τ clusters can be simply found in $O(n)$ overall time using brute-force search. For $h_\tau < i \leq h$, the spine of a level-i cluster C can be partitioned into singleton paths and spines of level-$(i-1)$ clusters that are contained in C. This requires only $O(1)$ time per cluster, as C is the disjoint union of at most 2 level-$(i-1)$ clusters. Thus the overall construction time is $O(n)+\sum_{i=h_\tau+1}^{h}((5/6)^i \cdot 2n \cdot O(1)) = O(n)$. □

In particular, when $h_\tau = \omega(1)$, the space cost in Lemma 5.2.4 is $o(n)$ bits.

We will solve $\mathcal{PM}_{0,h_\tau}$ using brute-force search, and support $\mathcal{PM}_{h_\tau,h}$ using a novel recursive approach as described below. The base cases of recursion are summarized in Lemmas 5.2.5 to 5.2.7.

Lemma 5.2.5. *$\mathcal{PM}_{0,h_\tau}$ can be solved using $O(2^{h_\tau})$ query time and no extra space.*

Proof. By Lemma 5.2.2, each level-h_τ cluster contains at most 2^{h_τ} nodes. Thus any path of $\mathcal{PM}_{0,h_\tau}$ can be traversed within $O(2^{h_\tau})$ time using `parent` and `LCA` operations. The minimum node on the path can be found during this traversal. □

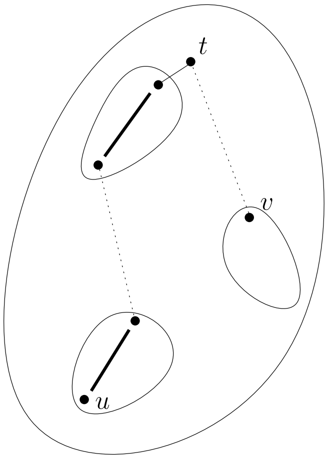
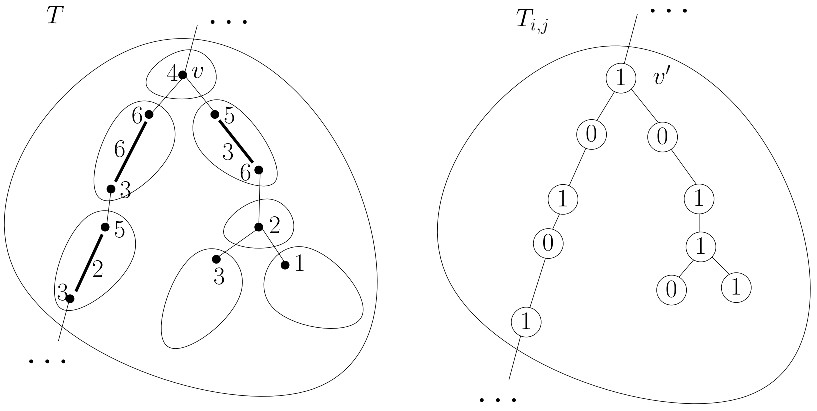


Figure 5.3: An illustration for the proof of Lemma 5.2.6. Here the large splinegon represents a level-j cluster and the small ones represent level-i clusters contained in the level-j cluster. Bold lines represent spines of level-i clusters and dotted lines represent paths.

Lemma 5.2.6. *For every pair of i and j satisfying $h_\tau \le i < j \le h$ and $j - i = O(1)$, $\mathcal{PM}_{i,j}$ can be solved using $O(1)$ query time and no extra space.*

Proof. Let u and v be the endpoints of some given query path of $\mathcal{PM}_{i,j}$. That is, u and v are two level-i boundary nodes that are contained in the same level-j cluster. To partition the path $u \sim v$, we first compute $t = \texttt{LCA}(\mathcal{B}, u, v)$. Node t must also be a level-i boundary node; otherwise the cluster that contains t would have at least two child clusters. As shown in Figure 5.3, we then partition the path $u \sim t$ into a constant number of singleton paths and spines of level-i clusters, which are all canonical. Initially, we set $x = u$ and let C be the level-i cluster that contains x. The following procedure is repeated until x becomes the parent of t. We make use of `cluster_spine`$(\mathcal{D}, C)$ to check whether x is on the spine of C. If x is on the spine, then y is set to be the other endpoint of the spine; otherwise $y = x$. In both cases, we select the path $x \sim y$, which must be canonical, and reset $x = \texttt{parent}(\mathcal{B}, y)$ and $C = \texttt{level\_cluster}(\mathcal{D}, i, x)$.

By Lemma 5.2.2, each level-j cluster is a disjoint union of a constant number of level-i clusters, as $2^{j-i} = 2^{O(1)}$ is a constant. Therefore, the path $u \sim t$ can be partitioned into $O(1)$ canonical subpaths using the procedure described above. The path $v \sim t$ can be partitioned similarly. Taking the union of these two sets of selected canonical paths except for a singleton path that contains t, we determine $O(1)$ canonical paths that the path $u \sim v$ is partitioned into, and thus the minimum node on $u \sim v$. Clearly the algorithm uses only $O(1)$ time. □



(a) (b)

Figure 5.4: An illustration for the proof of Lemma 5.2.7. (a) The large splinegon represents a level-j cluster and the small ones represent level-i clusters contained in the level-j cluster. Bold lines represent spines of level-i clusters. The number alongside a node is its weights, and the one alongside a spine is the minimum weight on the spine. (b) The 01-labeled tree $T_{i,j}$ that corresponds to the cluster head v.

Lemma 5.2.7. *For a fixed pair of i and j satisfying $h_\tau \leq i < j \leq h$, $\mathcal{PM}'_{i,j}$ can be solved using $O(1)$ query time, with an auxiliary data structure requiring $O((5/6)^i n)$ bits of extra space and construction time.*

Proof. In this proof, we will implicitly make each query path of $\mathcal{PM}'_{i,j}$ canonical and store the minimum nodes on these paths in a highly efficient way.

We construct an auxiliary ordinal tree, $T_{i,j}$, using the technique of tree extraction.

The structure of $T_{i,j}$ is obtained by extracting all level-i boundary nodes from $\mathcal{B}$. By Lemma 5.2.2, $T_{i,j}$ consists of $O((5/6)^i n)$ nodes. For convenience, we refer to a node in $T_{i,j}$ as u' iff it corresponds to a level-i boundary node u in $\mathcal{B}$. The conversion between u and u' can be performed in $O(1)$ time using $\texttt{BN\_rank}$ and $\texttt{BN\_select}$.

Next we assign labels from alphabet $\{0, 1\}$ to the nodes of $T_{i,j}$. We only consider the case in which the level-j boundary node is the head of its cluster; the other case can be handled similarly. Let u be any level-i boundary node and let v be the head of $C_0 = \texttt{level\_cluster}(\mathcal{B}, j, u)$, i.e., the level-$j$ cluster that contains u. As in the proof of Lemma 5.2.6, the path from u to v in $\mathcal{B}$ can be partitioned into a sequence of singleton paths and spines of level-i clusters. Let x' be the next node on the path from u' to v'. We assign 1 to u' in $T_{i,j}$ if $u = v$, or the minimum node on $u \sim v$ is smaller than that on $x \sim v$; otherwise we assign 0 to u'. See Figure 5.4 for an example. We represent this labeled tree within $O((5/6)^i n)$ bits of space and $O((5/6)^i n)$ construction time using Lemma 2.5.3.

To find the minimum node between u and v, we need only to find the closest 1-node to u' along the path from u' to v' in $T_{i,j}$. This node can be found in $O(1)$ time by performing $\texttt{level\_anc}_\alpha$ and $\texttt{depth}_\alpha$ operations on $T_{i,j}$. Let x' be the node found. Then the minimum node on $u \sim v$ must be x or appear on the spine of the level-i cluster that contains x, and thus can be retrieved in $O(1)$ time. □

Now we turn to consider general $\mathcal{PM}_{i,j}$, for which we will develop a recursive strategy with multiple iterations. At each iteration, we pick a sequence $i = i_0 < i_1 < i_2 < \cdots < i_k = j$, for which $\mathcal{PM}_{i_0,i_1}, \mathcal{PM}_{i_1,i_2}, \ldots, \mathcal{PM}_{i_{k-1},i_k}$ are assumed to be solved at the previous iteration. By Lemma 5.2.7, we solve $\mathcal{PM}'_{i,i_1}, \mathcal{PM}'_{i,i_2}, \ldots, \mathcal{PM}'_{i,i_k}$ using $O(k(5/6)^i n)$ bits of additional space and construction time.

Let u and v be the endpoints of a query path of $\mathcal{PM}_{i,j}$. That is, u and v are level-i boundary nodes that are contained in the same level-j cluster. As in the proof of Lemma 5.2.6, we still compute $t = \texttt{LCA}(\mathcal{B}, u, v)$ and partition the paths $u \sim t$ and $v \sim t$. For $u \sim t$, we compute $C_0 = \texttt{LLC}(\mathcal{B}, u, t)$, which is the lowest level cluster that contains both u and t. Let i' be the level of C_0. The case in which $i' = i$ can be simply handled by calling $\mathcal{PM}_{i_0,i_1}$. Otherwise, we determine s such that $i_s < i' \leq i_{s+1}$. Here s can be obtained in $O(1)$ time by precomputation for each possible value of i', which requires $O(\lg n)$ time and $O(\lg^2 n)$ bits of space. Let $C_1 = \texttt{level\_cluster}(\mathcal{D}, u, i_s)$, which is the level-$i_s$ cluster that contains u. Let $x = \texttt{cluster\_nn}(\mathcal{D}, C_1, t)$, which is a boundary node of C_1 that is between u and t. Similarly, let C_2 be the level-i_s cluster that contains t and let z be a boundary node of C_2 that is between u and t. By Lemma 5.2.3, x and z can be found in constant time. By Lemma 5.2.7, the paths $u \sim x$ and $z \sim t$ can be partitioned into $O(1)$ canonical paths by querying $\mathcal{PM}'_{i,i_s}$. Finally, the path $x \sim z$ can be partitioned recursively by querying $\mathcal{PM}_{i_s,i_{s+1}}$. See Figure 5.5 for an illustration of partitioning $u \sim t$. On the

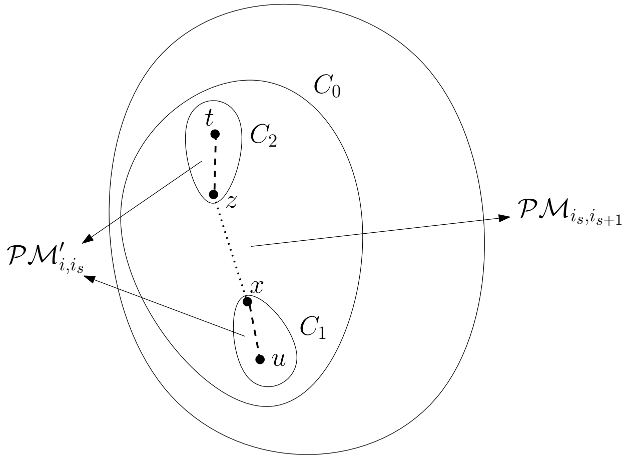


Figure 5.5: An illustration of partitioning $u \sim t$. The outermost splinegon represents the level-i_{s+1} cluster that contains both u and t. The paths $u \sim x$ and $z \sim t$, which are represented by dashed lines, are partitioned by querying $\mathcal{PM}'_{i,i_s}$. The path $x \sim z$, which is represented by a dotted line, is partitioned by querying $\mathcal{PM}_{i_s,i_{s+1}}$.

other hand, the path $v \sim t$ can be partitioned in a similar fashion. Thus the partition of the path $u \sim v$ is obtained.

Summarizing the discussion above, we have the following recurrences. Here $S_\ell(i,j)$ is the space cost and the construction time, and $Q_\ell(i,j)$ is the query time spent at the first ℓ iterations for solving $\mathcal{PM}_{i,j}$. It should be drawn to the reader's attention that the coefficient of $Q_\ell(i_s, i_{s+1})$ is 1 in Equation 5.2, since a top-to-bottom query path requires at most one recursive call to subproblems of the form $\mathcal{PM}_{i_s,i_{s+1}}$.

$$S_{\ell+1}(i,j) = \sum_{s=0}^{k-1} S_\ell(i_s, i_{s+1}) + O(k(5/6)^i n) \tag{5.1}$$

$$Q_{\ell+1}(i,j) = \max_{s=0}^{k-1} Q_\ell(i_s, i_{s+1}) + O(1). \tag{5.2}$$

The desired recursive strategy follows from these recurrences.

Lemma 5.2.8. *Given a fixed value L, there exists a recursive strategy and some constant c such that, for $0 \le \ell \le L$, $S_\ell(i, A_\ell(i)) \le c(6/7)^i n$ and $Q_\ell(i, A_\ell(i)) \le c\ell$.*

Proof. At the 0-th iteration, we set $A_0(i) = i + 1$. This can be used as the base case. By Lemma 5.2.6, $\mathcal{PM}_{i,i+1}$ can be supported using $O(1)$ query time at no extra space cost. Thus the statement holds for $\ell = 0$.

At the (ℓ+1)-st iteration, we pick the sequence $i, i{+}13, A_\ell(i{+}13), A_\ell^{(2)}(i{+}13), \ldots, A_\ell^{(i)}(i{+}13), A_\ell^{(i+1)}(i+13)$. The last term is $A_{\ell+1}(i)$. By Equation 5.1, for some sufficiently large constant c_1:

$$\begin{aligned}
S_{\ell+1}(i, A_{\ell+1}(i)) &\leq \sum_{0\leq j\leq i} S_\ell(A_\ell^{(j)}(i+13), A_\ell^{(j+1)}(i+13)) + O(i(5/6)^i n) \\
&\leq O(i(5/6)^i n) + \sum_{0\leq j\leq i} c_1(6/7)^{A_\ell^{(j)}(i+13)} \cdot n \\
&\leq O(i(5/6)^i n) + \sum_{0\leq j\leq i} c_1(6/7)^{i+13+j} \cdot n \\
&\leq O(i(5/6)^i n) + 7c_1(6/7)^{i+13} \cdot n \ \leq \ c_1(6/7)^i \cdot n.
\end{aligned}$$

This inequality follows because $7(6/7)^{13} \approx 0.9436 < 1$. Similarly, Equation 5.2 implies that, for some sufficiently large constant c_2,

$$Q_{\ell+1}(i, A_{\ell+1}(i)) \leq O(1) + \max_{0\leq j\leq i} Q_\ell(A_\ell^{(j)}(i+13), A_\ell^{(j+1)}(i+13)) \leq O(1) + c_2\ell \leq c_2(\ell+1).$$

The induction thus carries through, and the proof is completed by setting c to be the larger one of c_1 and c_2. □

We finally have Lemmas 5.2.9 and 5.2.10, which cover Theorem 5.1.1.

Lemma 5.2.9. *For $m \geq n$, $\mathcal{PM}_{0,h}$ can be solved using $O(\boldsymbol{\alpha}(m,n))$ query time in addition to $O(m)$ bits of extra space and construction time.*

Proof. Given a parameter $m \geq n$, we set $L = \boldsymbol{\alpha}(m,n)$ and $h_\tau = 0$, and recurse one more iteration. At the final (L+1)-st iteration, we pick the sequence $0, 1, 2, \ldots, \lfloor m/n \rfloor, A_L(\lfloor m/n \rfloor)$. The last term $A_L(\lfloor m/n \rfloor) > n \geq h$. This gives us

$$\begin{aligned}
S_{L+1}(0,h) &\leq S_{L+1}(0, A_L(\lfloor m/n \rfloor)) \\
&\leq S_L(\lfloor m/n \rfloor, A_L(\lfloor m/n \rfloor)) + O(\lfloor m/n \rfloor n) && \text{(Equation 5.1 and Lemma 5.2.6)} \\
&\leq O(m) && \text{(Lemma 5.2.8)}
\end{aligned}$$

and

$$\begin{aligned}
Q_{L+1}(0,h) &\leq Q_{L+1}(0, A_L(\lfloor m/n \rfloor)) \\
&\leq Q_L(\lfloor m/n \rfloor, A_L(\lfloor m/n \rfloor)) + O(1) && \text{(Equation 5.2 and Lemma 5.2.6)} \\
&\leq O(L) && \text{(Lemma 5.2.8)} \\
&= O(\boldsymbol{\alpha}(m,n)).
\end{aligned}$$

Adding Lemmas 5.2.3 and 5.2.4, the overall space cost is $O(m)$ additional bits, the overall construction time is $O(m)$, and the query time is $Q_{L+1}(0,h) = O(\boldsymbol{\alpha}(m,n))$. □

Lemma 5.2.10. *For $\mathtt{r}(n) = (6/7)^{\lg \boldsymbol{\alpha}(n)} \cdot n \in o(n)$, $\mathcal{PM}_{0,h}$ can be solved using $O(\boldsymbol{\alpha}(n))$ query time, $2n + O(\mathtt{r}(n))$ bits of extra space, and $O(n)$ construction time.*

Proof. We choose $L = \boldsymbol{\alpha}(n)$ and $h_\tau = \lceil \lg L \rceil$. Note that $h_\tau = \omega(1)$ and $A_L(h_\tau) \geq h$. Therefore we have $S_L(h_\tau, A_L(h_\tau)) = O((6/7)^{h_\tau} n) = O(\mathtt{r}(n)) = o(n)$, and $Q_L(h_\tau, A_L(h_\tau)) = O(L) = O(\boldsymbol{\alpha}(n))$. By Lemma 5.2.5, $\mathcal{PM}_{0,h_\tau}$ and $\mathcal{PM}'_{0,h_\tau}$ can be solved using $O(2^{h_\tau}) = O(\boldsymbol{\alpha}(n))$ query time at no extra space cost. Adding Lemmas 5.2.3 and 5.2.4, the overall space cost is $2n + O(\mathtt{r}(n))$ additional bits, the overall construction time is $O(n)$, and the query time is $O(\boldsymbol{\alpha}(n))$. □

By constructing the preorder label sequence of T, we further have

Corollary 5.2.11. *Let T be an ordinal tree on n nodes, each having a weight drawn from $[1..\sigma]$. Then T can be represented (a) using $n \lg \sigma + O(m)$ bits of space to support path minimum queries in $O(\boldsymbol{\alpha}(m,n))$ time, for any $m \geq n$; or (b) using $n(\lg \sigma + 2) + o(n)$ bits of space to support path minimum queries in $O(\boldsymbol{\alpha}(n))$ time.*

By directly applying the result of Sadakane and Grossi [93], we can further achieve compression and replace the $n \lg \sigma$ additive term in the space cost of both the results of the above corollary by $nH_k + o(n) \cdot \lg \sigma$ while providing the same support for queries, where $k = o(\log_\sigma n)$ and H_k is the k-th order empirical entropy of the preorder label sequence.

5.3 Semigroup Path Sum Queries

In this section, we generalize the approach of supporting path minimum queries to semigroup path sum queries. As in Section 5.2, we transform the given tree into a binary tree, which is further decomposed using Lemma 5.2.2. We also define the notions of spines, heads and tails in the same manner. Again, our strategy is to make some paths canonical, for which the sum of weights along each canonical path will be precomputed and stored. Naturally, all singleton paths are still canonical. Each query path will be partitioned into disjoint canonical subpaths, and the sum of weights along the whole query path can be obtained by summing up the precomputed sums over these canonical subpaths.

Let $h_\tau > 0$ be a parameter whose value will be determined later. The spines of clusters whose levels are higher than or equal to h_τ are made canonical. As in Section 5.2, we define subproblems $\mathcal{PS}_{i,j}$ and $\mathcal{PS}'_{i,j}$ as follows:

- $\mathcal{PS}_{i,j}$: sum up weights of nodes along query paths between two level-i boundary nodes that are contained in the same level-j cluster;
- $\mathcal{PS}'_{i,j}$: sum up weights of nodes along query paths from a level-i boundary node to a level-j one, where both boundary nodes are contained in the same level-j cluster.

$\mathcal{PS}_{0,h_\tau}$ will be solved using brute-force search, while $\mathcal{PS}_{h_\tau,h}$ will be solved using the recursive approach as described in Section 5.2. In the following, Lemmas 5.3.1 to 5.3.4 are modified from Lemmas 5.2.4 to 5.2.7. For each lemma, we give its proof only if the proof for the corresponding lemma in Section 5.2 cannot be applied directly.

Lemma 5.3.1. *It requires $O((5/6)^{h_\tau} n \lg \sigma)$ bits of additional space and $O(n)$ construction time to make these spines canonical.*

Proof. As the semigroup contains σ elements, it requires $\lg \sigma$ bits to store the sum of weights on the spine of a cluster. Thus the overall space cost is $\sum_{i=h_\tau}^{h}((5/6)^i \cdot 2n \cdot \lg \sigma) = O((5/6)^{h_\tau} n \lg \sigma)$ bits. In particular, when $h_\tau = \omega(1)$, the space cost is $o(n \lg \sigma)$ bits. The construction is the same in Lemma 5.2.4. □

Lemma 5.3.2. *$\mathcal{PS}_{0,h_\tau}$ can be solved using $O(2^{h_\tau})$ query time and no extra space.*

Lemma 5.3.3. *For every pair of i and j satisfying that $h_\tau \leq i < j \leq h$ and $j - i = O(1)$, $\mathcal{PS}_{i,j}$ can be solved using $O(1)$ query time and no extra space.*

Lemma 5.3.4. *For a fixed pair of i and j satisfying that $h_\tau \leq i < j \leq h$, $\mathcal{PS}'_{i,j}$ can be solved using $O(1)$ query time, $O((5/6)^i n \lg \sigma)$ bits of extra space, and $O((5/6)^i n)$ construction time.*

Proof. We make each query path of $\mathcal{PS}'_{i,j}$ canonical and store the sum of weights along each of these paths explicitly. It is easy to see that the space cost is $O((5/6)^i n \lg \sigma)$ extra bits and the construction time is $O((5/6)^i n \lg \sigma)$. □

Following the same recursive strategy in Section 5.2, we have the following recurrences. Here $S_\ell(i,j)$ is the space cost, $P_\ell(i,j)$ the construction time, and $Q_\ell(i,j)$ is the query time spent at the first ℓ iterations for solving $\mathcal{PS}_{i,j}$.

$$S_{\ell+1}(i,j) = \sum_{s=0}^{k-1} S_\ell(i_s, i_{s+1}) + O(k(5/6)^i n \lg \sigma) \tag{5.3}$$

$$P_{\ell+1}(i,j) = \sum_{s=0}^{k-1} P_\ell(i_s, i_{s+1}) + O(k(5/6)^i n) \tag{5.4}$$

$$Q_{\ell+1}(i,j) = \max_{s=0}^{k-1} Q_\ell(i_s, i_{s+1}) + O(1). \tag{5.5}$$

We then have the following key lemma, which is similar to Lemma 5.2.8.

Lemma 5.3.5. *Given a fixed value L, there exists a recursive strategy and some constant c such that, for $0 \leq \ell \leq L$, $S_\ell(i, A_\ell(i)) \leq c(6/7)^i n \lg \sigma$, $P_\ell(i, A_\ell(i)) \leq c(6/7)^i n$, and $Q_\ell(i, A_\ell(i)) \leq c\ell$.*

Finally, we store weights of nodes in the preorder label sequence of T. This requires $n \lg \sigma + o(n)$ bits of space, and the weight of each node can be accessed in $O(1)$ time. The rest of the proof for Theorem 5.1.2 follows from the same strategies of Lemmas 5.2.9 and 5.2.10.

5.4 Encoding Topology Trees: Proof of Lemma 5.2.3

Let T be an ordinal tree on n nodes. As described in Section 5.2, we transform T into a binary tree $\mathcal{B}$, and compute the directed topology tree of $\mathcal{B}$ as $\mathcal{D}$. Let $n_{\mathcal{D}}$ denote the number of nodes in $\mathcal{D}$. By Lemma 5.2.2, we have that $n_{\mathcal{D}} = O(n)$, as there are at most $(5/6)^i \cdot 2n$ level-i clusters. Let $i_1 = \lceil 12 \lg\lg n \rceil$ and $i_2 = \lfloor \lg\lg n \rfloor - 1$. Again by Lemma 5.2.2, there are at most $n_1 = (5/6)^{i_1} \cdot n = O(n/(6/5)^{12 \lg\lg n}) = O(n/\lg^{12\lg(6/5)} n) \in O(n/\lg^3 n)$ level-i_1 clusters, each being of size at most $m_1 = 2^{i_1} \leq 2^{12 \lg\lg n + 1} = 2\lg^{12} n$. Similarly, there are at most $n_2 = (5/6)^{i_2} \cdot n = O(n/\lg^{\lg(6/5)} n) \in O(n/(\lg^{1/5} n))$ level-i_2 clusters, each being of size at most $m_2 = 2^{i_2} \leq 2^{\lg\lg n - 1} = (\lg n)/2$. Clusters at levels i_1 and i_2 are referred to as *mini-clusters* and *micro-clusters*, respectively.

We will precompute several lookup tables that support certain queries for each possible micro-cluster. Note that two clusters are different if the sets of non-dummy nodes are different. There are $O(n^{1-\delta})$ distinct micro-clusters for some $\delta > 0$, since $m_2 \leq (1/2)\lg n$. If each micro-cluster costs $o(n^\delta)$ bits, then the space cost of the lookup table is only $o(n)$ additional bits. We first make use of a lookup table to store the encodings of micro-clusters.

Lemma 5.4.1. *All micro-clusters can be encoded in $2n + o(n)$ bits of space such that given the topological rank of a micro-cluster, its encoding can be retrieved in $O(1)$ time.*

Proof. Note that $\mathcal{B}$ has at most $2n$ nodes. Given a micro-cluster C, we do not store its encoding directly because it could require about $4n$ bits of space for all micro-clusters. Instead, we define X to be the union of non-dummy nodes and dummy boundary nodes of C and store only C_X, where C_X is the X-extraction of C as defined in Section 2.5.3. We also mark the (at most 2) dummy nodes in C_X, which requires $O(\lg m_2) = O(\lg\lg n)$ bits per node. As illustrated in Figure 5.6, we encode C_X as the balanced parentheses

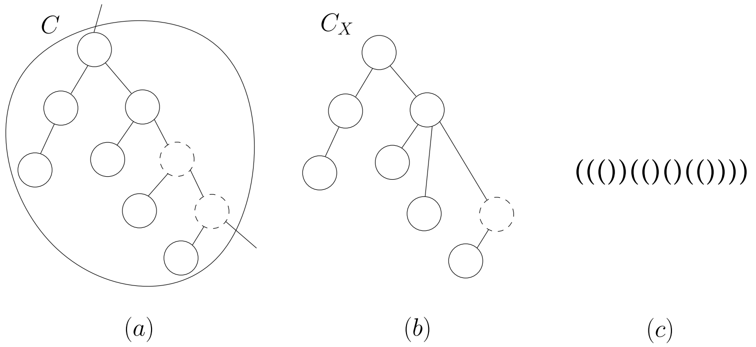


Figure 5.6: An example of encoding micro-clusters. (a) A micro-cluster C in which dummy nodes are represented by dashed circles. (b) The corresponding C_X obtained by preserving non-dummy nodes and dummy boundary nodes. (c) The balanced parentheses for C_X.

described in Section 2.5.1. The overall space cost of encoding C is $2n_C + O(\lg\lg n)$ bits, where n_C is the number of non-dummy nodes in C. We concatenate the above encodings of all micro-clusters ordered by topological rank and store them in a sequence, P, of $n' = 2n + O(n \lg\lg n/(\lg^{1/5} n))$ bits.

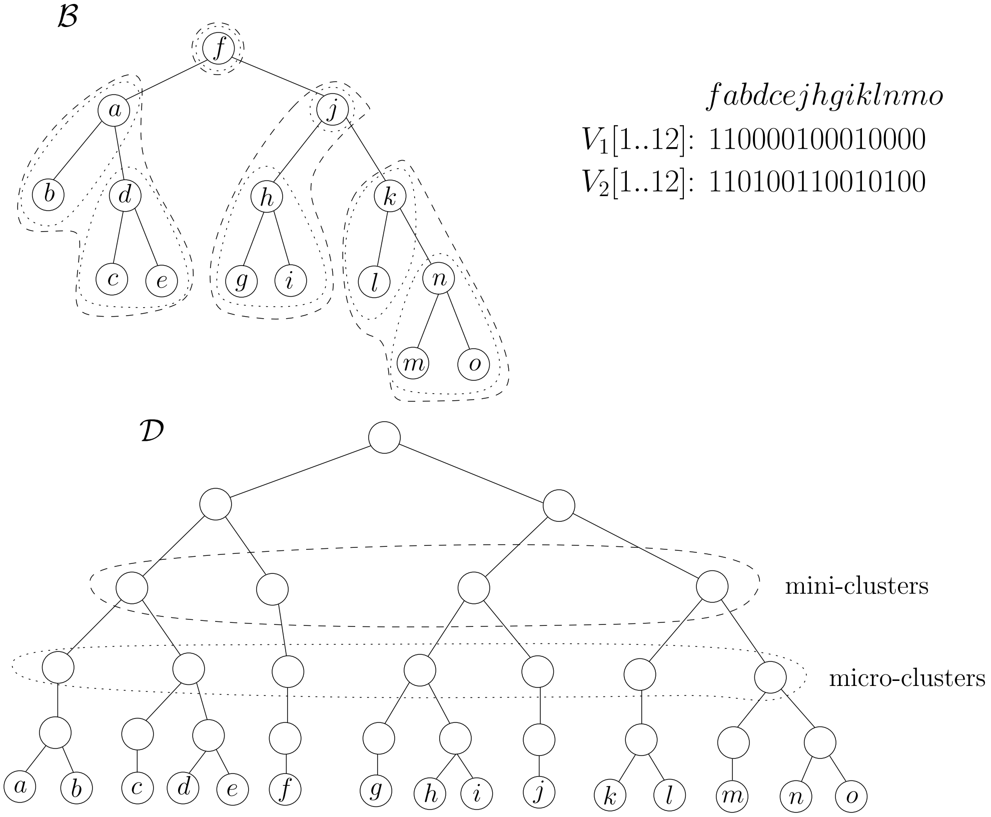
We construct a sparse bit vector, P', of the same length, and set $P'[i]$ to 1 iff $P[i]$ is the first bit of the encoding of a micro-cluster. P' can be represented using Lemma 2.3.1 in $\lg \binom{n'}{n_2} + O(n \lg\lg n/\lg n) = O(n \lg\lg n/(\lg^{1/5} n))$ bits to support $\texttt{rank}_\alpha$ and $\texttt{select}_\alpha$ in constant time. We construct another bit vector $B_0[1..n_{\mathcal{D}}]$, in which $B_0[j] = 1$ iff the cluster with topological rank j is a micro-cluster, which is also encoded using Lemma 2.3.1 in $O(n \lg\lg n/(\lg^{1/5} n))$ bits.

To retrieve the encoding of a cluster, C, whose topological rank is j, we first use B_0 to check if C is a micro-cluster. If this is true, let $r = \texttt{rank}_1(B_0, j)$. Then the encoding of C_X is $P[\texttt{select}_1(P', r)..\texttt{select}_1(P', r+1) - 1]$. To recover C from C_X, we need only to reverse the binary tree transformation described at the beginning of Section 5.2.2. This can be done in $O(1)$ time using a lookup table F_0 of $o(n)$ bits. □

Now we start to consider the support for operations. By Lemma 5.2.2, each node is contained in exactly one cluster at each level. We borrow the terminology from Geary et al.'s work [54] and define the *τ-name* of a node x to be $(\tau_1(x), \tau_2(x), \tau_3(x))$, where $\tau_1(x)$, $\tau_2(x)$ and $\tau_3(x)$ are the topological ranks of the level-i_1, level-i_2, and level-0 clusters that contain x, respectively. Let $i_3 = 0$. Note that, for $k = 1, 2$, $\tau_{k+1}(x)$ is represented as the relative value with respect to the level-i_k cluster that contains x, i.e., the difference between the topological ranks of the level-i_{k+1} and level-i_k clusters that contain x. Thus $\tau_2(x)$ and $\tau_3(x)$ can be encoded in $O(\lg\lg n)$ bits.

As in Lemma 5.2.2, preorder segments are defined to be maximal substrings of nodes in the preorder sequence that are in the same cluster [64, Definition 4.22]. By the same lemma, each cluster contains only one node or has only one child cluster, and thus the nodes of each cluster belong to at most 2 *preorder segments*. These preorder segments and the cluster are said to be *associated with* each other, and the preorder segments of a level-i cluster are called *level-i preorder segments*. For simplicity, level-i_1 and level-i_2 preorder segments are also called *mini-segments* and *micro-segments*, respectively.

In the following proofs, we will precompute several lookup tables that store certain information for each possible micro-cluster C. When we say *the hierarchy of C*, we mean the hierarchy obtained by partitioning C as described in Lemma 5.2.2.



$fabdcejhgiklnmo$

$V_1[1..12]$: 110000100010000

$V_2[1..12]$: 110100110010100

$B_1[1..39]$: 111000000000001000110000000010000000000

$B_2[1..39]$: 000100010000001000010000010001000100000

Figure 5.7: An illustration for the proofs of Lemmas 5.4.2 and 5.4.3. Here mini-clusters and micro-clusters are enclosed by dashed and dotted splinegons, respectively. The bit vectors B_1, B_2, V_1, and V_2 are constructed for the directed topology tree shown in Figure 5.2.

Lemma 5.4.2. *It requires $O(1)$ time and $o(n)$ bits of additional space for the conversion between the preorder rank of a node in T or $\mathcal{B}$ and its τ-name.*

Proof. We only consider the conversion for nodes in $\mathcal{B}$; the other case can be handled similarly. For each level $i \in [i_2..h]$ and each level-i cluster C, we store the following information in $D(C)$: its topological rank, its root node, its boundary nodes, and the starting and ending positions of its associated preorder segments in $\mathcal{B}$. For levels i_1 to h, it requires $O(\lg n)$ bits of space per cluster to store the information directly (the nodes stored in $D(C)$ are encoded as their preorder ranks in $\mathcal{B}$). For each cluster C at levels i_2 to $i_1 - 1$, we store the relative ranks with respect to the mini-cluster C' that contains C. More precisely, we encode the difference between the topological ranks of C and C', and each node stored in $D(C)$ is encoded as i if it is the i-th node in C' in preorder. This requires only $O(\lg\lg n)$ bits per cluster, as each mini-cluster is of size at most $m_1 \in O(\lg^{12} n)$. Physically, for all clusters above level i_2, we use two arrays to store the above information and use two bit vectors to locate the corresponding entry for any given cluster. For levels i_1 to h, we construct a bit vector $B_1[1..n_{\mathcal{D}}]$ in which $B_1[j] = 1$ iff the cluster with topological rank j is at or above level i_1. We construct an array D_1 whose length is equal to the number of clusters at levels i_1 to h; for each cluster C at levels i_1 to $h - 1$, $D(C)$ is stored in the $\texttt{rank}_1(B_1, j)$-th entry of D_1 if the topological rank of C is j. Similar auxiliary data structures D_2 and B_2 are also constructed for clusters at levels i_2 to $i_1 - 1$. In the hierarchy $\mathcal{D}$, there are $O(n_1)$ clusters at levels i_1 to h, and $O(n_2)$ clusters at levels i_2 to $i_1 - 1$. Thus the overall space cost, including the cost of encoding B_1 and B_2 using Lemma 2.3.1, is $O(n_1) \times O(\lg n) + O(n_2) \times O(\lg\lg n) \in o(n)$ bits.

For $k \in \{1, 2\}$, all level-i_k preorder segments form a partition of the preorder traversal sequence of $\mathcal{B}$, and we mark their starting positions in a bit vector V_k. More precisely, we set $V_k[j] = 1$ if and only if the j-th node in preorder of $\mathcal{B}$ is the first node in some level-i_k preorder segment. By Lemma 2.3.1, these two bit vectors can be encoded in $o(n)$ bits of space to support `rank` and `select` in $O(1)$ time. For mini-segment (or micro-segment) s, we store in $E(s)$ the topological rank of its associated mini-cluster (or micro-cluster). It requires $O(n_1) \times O(\lg n) = O(n/\lg^2 n)$ bits to store $E(s)$ directly for all mini-segments. For each micro-segment s, we store the relative topological rank of its associated micro-cluster with respect to the mini-cluster that contains s. This requires only $O(\lg\lg n)$ bits per micro-segment, and $O(n_2) \times O(\lg\lg n) = O(n \lg\lg n/(\lg^{1/5} n))$ bits in total.

Finally, we precompute a lookup table F_1 that stores, for each possible micro-cluster C, the mapping between nodes in C and level-0 clusters in the i_2-level hierarchy of C. These nodes and level-0 clusters are encoded as the relative preorder ranks and topological ranks with respect to C. It is easy to see that F_1 occupies $o(n)$ bits of space, since $m_2 \leq (1/2)\lg n$.

Given a node x in $\mathcal{B}$, we first locate s_1, the mini-segment that contains x, using `rank` operations on V_1. By accessing $E(s_1)$, we can determine C_1, which is the mini-cluster that contains x and the associated mini-segment s_1. Then we locate s_2, the micro-segment that contains x, using V_2. By accessing $E(s_2)$ and $D(C_1)$, we can determine C_2, the micro-cluster that contains x. Finally, by accessing F_1, we can find the level-0 cluster that represents x. That is, we obtain the τ-name of x.

In the other direction, given the τ-name of some node x, we can immediately determine C_1, C_2 and C_3, which are the level-i_1, level-i_2 and level-0 clusters that contain x, respectively. By accessing $D(C_1)$ and $D(C_2)$, we can find the associated preorder segments of C_2. By accessing F_1, we can compute the relative preorder rank of x with respect to C_2. Then we can determine the preorder rank of x in $\mathcal{B}$. □

The conversion between nodes in T and $\mathcal{B}$ directly follows from Lemma 5.4.2. Thus, to answer queries that ask for a node, it suffices to return either its τ-name, or its preorder rank in $\mathcal{B}$ or T. In the remaining part of this section, when we talk about the preorder rank of a node x, we refer to the preorder rank of x in $\mathcal{B}$, unless otherwise specified.

Lemma 5.4.3. *Operations* `cluster_head` *and* `cluster_tail` *can be supported in* $O(1)$ *query time and* $o(n)$ *bits of additional space.*

Proof. We precompute a lookup table F_2 that stores, for each possible micro-cluster C and each cluster C' in the i_2-level hierarchy of C, the root and the boundary nodes of C'. These nodes are encoded as the relative preorder ranks with respect to C. It is clear that F_2 occupies $o(n)$ bits of space.

Recall the bit vectors B_1 and B_2 constructed in the proof of Lemma 5.4.2, as well as the information stored in $D(C)$ for each cluster C whose level is higher than or equal to i_2 in the same proof. Given a cluster C with topological rank j, we first determine if the level of C is at or above i_1 by checking if $B_1[j] = 1$. If the level of C is at or above i_1, then we can retrieve the answers to `cluster_head` and `cluster_tail` directly from $D(C)$. Otherwise, we find the mini-cluster C_1 that contains C by $\texttt{select}_1(B_1, \texttt{rank}_1(B_1, j))$. Then we determine if the level of C is in $[i_2..i_1 - 1]$ by checking if $B_2[j] = 1$. If this is true, then we can obtain the answers using $D(C_1)$ and $D(C)$: To locate the root, c_r, of C (the boundary nodes can be located using the same approach), the relative preorder of c_r in C_1 can be used to locate the preorder segment in C_1 containing c_r. The preorder rank of the first node in this segment, which is stored in $D(C_1)$, can be used to further compute the preorder rank of c_r in constant time. If the level of C is not within the above range, we further find the micro-cluster C_2 that contains C by $\texttt{select}_1(B_2, \texttt{rank}_1(B_2, j))$. By accessing $D(C_1)$, $D(C)$ and the entry in F_2 that corresponds to the encoding of C_2, we can obtain the answers in $O(1)$ time. □

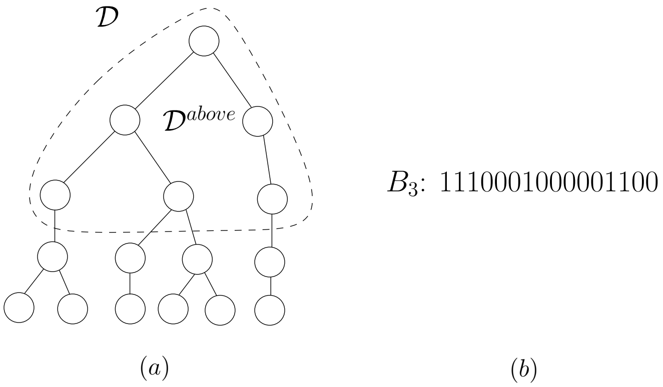


Figure 5.8: An illustration of the support for `level_cluster`. (a) A directed topology tree $\mathcal{D}$ in which the topmost three levels belong to $\mathcal{D}^{above}$. (b) The corresponding bit vector B_3 for $\mathcal{D}$ and $\mathcal{D}^{above}$.

Lemma 5.4.4. *Operations* `level_cluster` *and* `LLC` *can be supported in $O(1)$ query time and $o(n)$ bits of additional space.*

Proof. Let $\mathcal{D}^{above}$ be the topmost $h - i_2 + 1$ levels of $\mathcal{D}$. It is clear that $\mathcal{D}^{above}$ represents the hierarchy for the clusters at levels i_2 to h. We store $\mathcal{D}^{above}$ using Lemma 2.5.2. This requires $O(n_2) = O(n/(\lg^{1/5} n))$ bits of space, since there are at most $O(n_2)$ clusters at these levels. As illustrated in Figure 5.8, we construct a bit vector B_3 in which $B_3[j]$ is 1 iff the cluster with topological rank j is present in $\mathcal{D}^{above}$. By Lemma 2.3.1, B_3 occupies $o(n)$ bits of space and can be used to perform conversions between the topological rank of any given cluster present in $\mathcal{D}^{above}$ and its preorder rank in $\mathcal{D}^{above}$ in constant time.

Let n'_C be the number of nodes in a micro-cluster C, including both dummy and non-dummy ones. We precompute a lookup table F_3 that stores, for each possible micro-cluster C, each level $i \in [0..i_2]$, and each $j \in [1..n'_C]$, the relative topological rank of the level-i cluster in the hierarchy of C that contains the j-th node of C. We also precompute another lookup table F_4 that stores, for each possible micro-cluster C, and each $j_1, j_2 \in [1..n'_C]$, the lowest level cluster in the hierarchy of C that contains the j_1-st and the j_2-nd nodes of C.

We have two cases in computing `level_cluster`$(\mathcal{D}, i, x)$. If $i \leq i_2$, then the answer can be retrieved from F_3 directly. Otherwise, we can determine the micro-cluster that contains x using the τ-name of x, and find the level-i cluster that contains x using `level_anc` operations on $\mathcal{D}^{above}$.

The support for `LLC`$(\mathcal{D}, x, y)$ is similar. We first determine if x and y are in the same micro-cluster using their τ-names. If they are in the same micro-cluster, then the answer can be retrieved from F_4 in $O(1)$ time. Otherwise, we can find the micro-clusters that contain x and y, respectively. Let these micro-clusters be C_1 and C_2. We can compute `LLC` by finding the lowest common ancestor of C_1 and C_2 in $\mathcal{D}^{above}$. □

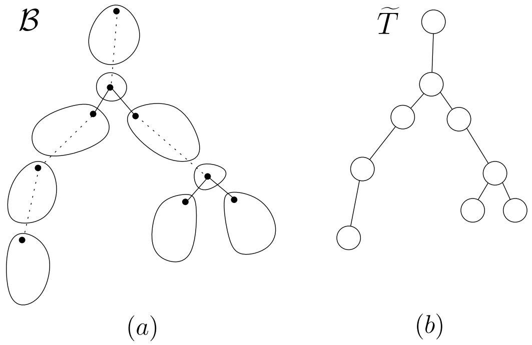


Figure 5.9: An illustration for the proof of Lemma 5.4.5. (a) A binary tree $\mathcal{B}$ in which micro-clusters are represented by splinegons, and their roots are represented by solid circles. (b) The tree $\widetilde{T}$ obtained by extracting roots of micro-clusters in $\mathcal{B}$.

Lemma 5.4.5. *Operation* `LCA` *can be supported in $O(1)$ query time and $o(n)$ bits of additional space.*

Proof. As described in Section 2.5.3, we extract the roots of all micro-clusters from $\mathcal{B}$. As illustrated in Figure 5.9, the extracted tree, $\widetilde{T}$, is maintained using Lemma 2.5.2. For the conversion between nodes in $\widetilde{T}$ and roots of micro-clusters in $\mathcal{B}$, we maintain a bit vector V_3 using Lemma 2.3.1, for which the j-th bit is 1 iff the j-th node of $\mathcal{B}$ is the root of some micro-cluster. In addition, we precompute a lookup table F_5 that stores, for each possible micro-cluster C, and each $j_1, j_2 \in [1..n'_C]$, the lowest common ancestor of the j_1-st and the j_2-nd nodes of C.

To compute $\texttt{LCA}(\mathcal{B}, x, y)$, we first verify if nodes x and y are in the same micro-cluster. If they are both in the same cluster, then their lowest common ancestor can be found by accessing F_5. Otherwise, we determine the micro-clusters that contain x and y. Let u and v be the roots of these micro-clusters, respectively. We perform an `LCA` operation on $\widetilde{T}$ and find the lowest micro-cluster root that is a common ancestor of u and v. This root must be the lowest common ancestor of x and y. □

Lemma 5.4.6. *Operation* `cluster_nn` *can be supported in $O(1)$ query time and no extra space.*

Proof. Let C be a cluster and let x be some node that is outside of C. To compute $\texttt{cluster\_nn}(\mathcal{D}, C, x)$, we first determine if x is in the subtree rooted at the root of C. Let

$y = \texttt{cluster\_head}(\mathcal{D}, C)$. If $\texttt{LCA}(\mathcal{B}, x, y) \neq y$, then x is also outside of the subtree rooted at y, and the closest boundary node to x must be y. Otherwise, the closest boundary node is $z = \texttt{cluster\_tail}(\mathcal{D}, C)$, the tail of the cluster C. □

Lemma 5.4.7. *Operation* `parent` *can be supported in $O(1)$ query time and $o(n)$ bits of additional space.*

Proof. We precompute a lookup table F_6 that stores, for each possible micro-cluster C and each $j \in [1..n'_C]$, the parent of the j-th node of C. Given a node x, we first find the micro-cluster C that contains x. If x is not the root of C, then we can retrieve the parent node of x from F_6 directly. Otherwise, we find the lowest micro-cluster C' above C by finding the parent of the node in $\widetilde{T}$ that represents C. Then the parent of x must be the closest boundary node of x in C', which can be found by `cluster_nn`. □

Lemma 5.4.8. *Operations* `BN_rank` *and* `BN_select` *can be supported in $O(1)$ query time and $o(n)$ bits of additional space.*

Proof. We first construct data structures to support $\texttt{BN\_rank}(\mathcal{B}, i, x)$ and $\texttt{BN\_select}(\mathcal{B}, i, j)$ when $i \in [i_2..h]$. For each $i \in [i_2..h]$, we construct a bit vector I_i, in which $I_i[j] = 1$ iff the j-th node in preorder is a level-i boundary node. Each of these $O(h) = O(\lg n)$ bit vectors is encoded using Lemma 2.3.1. It is clear that to support these two operations in this special case, it is sufficient to perform a single `rank` or `select` operation on I_i, and thus can be performed in constant time. We calculate the space cost in two steps. We first sum up the space cost of all I_i's for $i \in [i_1, h]$. The number of 1-bits in each of these bit vectors is $O(n/\lg^3 n)$, and thus the total cost of these $O(\lg n)$ bit vectors is $O(\lg n) \times O(n \lg\lg n/\lg^3 n) = O(n \lg\lg n/\lg^2 n)$. We then sum up the space occupancy of all I_i's for $i \in [i_2..i_1 - 1]$. As the number of 1 bits in each of these bit vectors is $O(n/\lg^{1/5} n)$, the total cost of these $O(\lg\lg n)$ bit vectors is $O(n(\lg\lg n)^2/\lg^{1/5} n)$ bits, which subsumes the space cost computed in the previous step.

Next we construct data structures to support $\texttt{BN\_rank}(\mathcal{B}, i, x)$ for $i \in [0..i_2 - 1]$. Let n_s and n'_s denote the number of mini-segments and micro-segments, respectively. We construct the following two sets of arrays and one lookup table:

- An array $G_i[1..n_s]$ for each $i \in [0..i_2 - 1]$, in which $G_i[j]$ stores the number of level-i boundary nodes that precede the j-th mini-segment in preorder;
- An array $G'_i[1..n'_s]$ for each $i \in [0..i_2 - 1]$, in which $G'_i[j]$ stores the number of level-i boundary nodes that precede the j-th micro-segment in preorder and also reside in the same mini-segment containing this micro-segment;

- A universal lookup table F_7 that stores, for any possible micro-cluster C (here micro-clusters with the same tree structure but different micro-segments are considered different; recall that there are at most 2 micro-segments in each micro-cluster), any node, x, in C identified by its τ_3-name, and any number $i \in [0..i_2 - 1]$, the number of level-i boundary nodes preceding x in preorder.

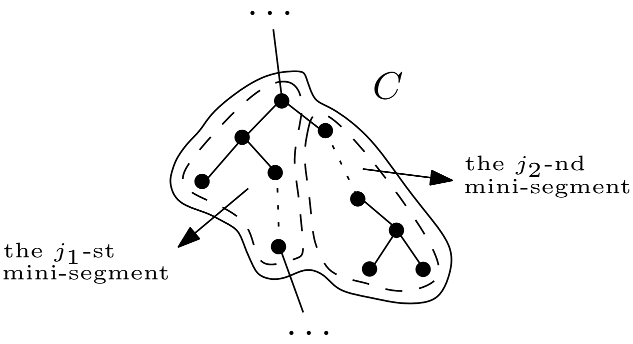


Figure 5.10: An illustration of the support for BN_rank. Here a mini-cluster C with two mini-segments is enclosed by a solid splinegon, and the nodes of these two mini-segments are enclosed by dashed splinegons. We draw only level-i boundary nodes inside C and the edges and paths that connect them, which are represented by solid and dotted lines, respectively. Then we have $G_i[j_1] = k_1$, $G_i[j_1{+}1] = k_1{+}5$, $G_i[j_2] = k_1{+}k_2{+}5$, and $G_i[j_2{+}1] = k_1{+}k_2{+}10$, where k_1 is the number of level-i boundary nodes that precede the head of C in preorder, and k_2 is the number of level-i boundary nodes that are descendants of the tail of C.

We illustrate the definition of G_i's in Figure 5.10. To analyze storage cost, observe that the space costs of all the G'_i's dominate the overall cost, which can be expressed as $O(\lg\lg n) \times O(n \lg\lg n / \lg^{1/5} n) = O(n(\lg\lg n)^2 / \lg^{1/5} n)$ bits.

With these auxiliary structures and the bit vectors V_1 and V_2 constructed in the proof of Lemma 5.4.2, we can support BN_rank$(\mathcal{B}, i, x)$ for $i \in [0..i_2 - 1]$ as follows. We first retrieve $G_i[$rank$(V_1, i)]$ which is the number of level-i boundary nodes preceding the mini-segment, s, containing x, and $G'_i[$rank$(V_2, i)]$ which is the number of level-i boundary nodes inside s that precede the micro-segment, s', containing x. It now suffices to compute the number of level-i boundary nodes preceding x inside s'. This can be computed by first retrieving the encoding of the micro-cluster containing x (its topological rank is $\tau_1(x) + \tau_2(x)$) and then perform a table lookup in F_7.

Finally, to support BN_select$(\mathcal{B}, i, j)$ for $i \in [0..i_2 - 1]$, we construct the following data structures (e_i denotes the number of level-i boundary nodes):

- A bit vector $R_i[1..e_i]$ for each $i \in [0..i_2 - 1]$, in which $R_i[j] = 1$ iff the j-th level-i boundary node in preorder has the smallest preorder rank among all the level-i

boundary nodes contained in the same mini-segment (let t_i denote the number of 1-bits in R_i);

- An array $S_i[1..t_i]$, in which $S_i[j]$ stores the topological rank of the mini-cluster containing the level-i boundary node corresponding to the j-th 1-bit in R_i;
- A bit vector $R'_i[1..e_i]$ for each $i \in [0..i_2 - 1]$, in which $R'_i[j] = 1$ iff the j-th level-i boundary node in preorder has the smallest preorder rank among all the level-i boundary nodes contained in the same micro-segment (let t'_i denote the number of 1-bits in R'_i);
- An array $S'_i[1..t'_i]$, in which $S'_i[j]$ stores a pair: the first item is the relative topological rank of the micro-cluster (relative to the mini-cluster that it resides in) containing the level-i boundary node y corresponding to the j-th 1-bit in R'_i, and an integer in $\{1, 2\}$ indicating which of the up to 2 micro-segments inside this micro-cluster contains y;
- A universal lookup table F_8 that stores, for any possible micro-cluster C, any number $i \in [0..i_2 - 1]$, any number $j \in [1..n_C]$ and any number $k \in \{1, 2\}$, the τ_3-name of the j-th level-i boundary node in the k-th micro-segment of C, or -1 if such a node does not exist.

As $t_i = O(n/\lg^3 n)$ and $t'_i = O(n/\lg^{1/5} n)$ for $i \in [0..i_2 - 1]$, it is easy to show that these structures occupy $O(n(\lg\lg n)^2/\lg^{1/5} n)$ bits. To support $\texttt{BN\_select}(\mathcal{B}, i, j)$ for $i \in [0..i_2 - 1]$, let z denote the answer to be computed. We first use R_i and S_i to locate the mini-cluster containing z, and then use R'_i and S'_i to locate z's micro-cluster C. In this process, we also find out which micro-segment of C contains z, and how many level-i boundary nodes precede z in preorder are in the same micro-segment. A table lookup using F_8 will complete this process. □

5.5 Path Reporting Queries

In this section, we consider the problem of supporting path reporting queries. We represent the input tree T as an ordinal one, which only adds $O(n)$ bits to the overall space cost. The weights of nodes are assumed to be drawn from $[1..\sigma]$. We follow the general strategy described in Section 3.5.2, which makes use of range trees and tree extraction. To achieve new results, we also make novel use of several other data structure techniques, including tree extraction described in Section 2.5.3, the ball-inheritance problem described in Section 2.4.1, and finally the succinct indices developed in Section 5.2.

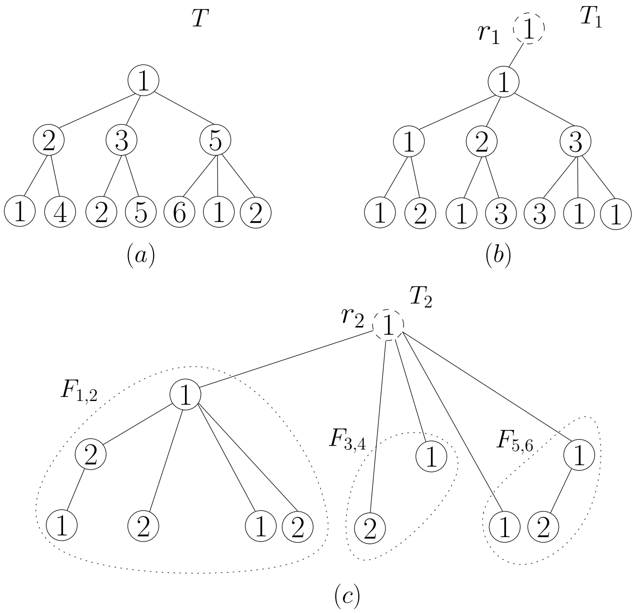


Figure 5.11: (a) An input tree T with $n = 11$ and $\sigma = 6$, for which the conceptual range tree has branching factor $f = 3$. (b) The corresponding tree T_1, where the dummy root r_1 is represented by a dashed circle. (c) The corresponding tree T_2, where the dummy root r_2 is represented by a dashed circle, and $F_{1,2}$, $F_{3,4}$, and $F_{5,6}$ are marked by dotted splinegons.

For completeness, we review the data structures described in Section 3.5.2. We build a conceptual range tree on $[1..\sigma]$ with the branching factor $f = \lceil \lg^{\epsilon} n \rceil$. Starting from the top level, which contains $[1..\sigma]$ initially, we keep splitting each range at the current lowest level into f child ranges of almost equal sizes as specified in Equations 3.5 and 3.6, until we obtain σ leaf ranges that contains a single weight each. This conceptual range tree has $h = \lceil \log_f \sigma \rceil + 1$ levels, which are numbered from top to bottom. The top level is the first level, and the bottom level is the h-th level.

For $\ell = 1, 2, \ldots, h-1$, we create an auxiliary tree T_ℓ for the ℓ-th level, which initially contains a dummy root r_ℓ only. We list the ranges at the ℓ-th level in increasing order of left endpoints. Let $[a_1..b_1], [a_2..b_2], \ldots, [a_m..b_m]$ be these ranges. For $i = 1, 2, \ldots, m$, we construct F_{a_i,b_i} as described in Section 3.2, and add the roots of ordinal trees in F_{a_i,b_i} as children of r_ℓ, preserving the original left-to-right order. Remember that the ranges at each level form a disjoint union of $[1..\sigma]$. Thus there is a one-to-one correspondence between the non-dummy nodes in T_ℓ and the nodes in T.

For each T_ℓ, we assign labels to its nodes. The dummy root is always assigned 1. For each node x in T, we use x_ℓ to denote the node in T_ℓ that corresponds to x. We say that a range $[a..b]$ at the ℓ-th level contains x_ℓ if the weight of x is between a and b. We assign a label α to x_ℓ if the range at the $(\ell+1)$-st level that contains $x_{\ell+1}$ is the α-th child of the range at the ℓ-th level that contains x_ℓ. See Figure 5.11 for an example. T_ℓ is maintained using a succinct representation for labeled ordinal trees over a sublogarithmic alphabet, i.e., Lemma 3.5.5.

When representing each T_ℓ using Lemma 3.5.5, the preorder label sequence of T_ℓ is stored explicitly. Thus the preorder label sequences of all T_ℓ's essentially form the generalized wavelet tree of the preorder label sequence of T. The $\texttt{pre\_rank}_\alpha$, $\texttt{pre\_select}_\alpha$, $\texttt{pre\_count}_\beta$ and $\texttt{lowest\_anc}_\alpha$ operations allow us to traverse up and down this generalized wavelet tree using standard wavelet tree algorithms. More details are given in Section 3.5.2, and the result is summarized in the following lemma:

Lemma 5.5.1. *Given a node x in T_ℓ, where $1 < \ell \leq h-1$, its corresponding node in $T_{\ell-1}$ can be found in $O(1)$ time. Similarly, given a node x in T_ℓ, where $1 \leq \ell < h-1$, its corresponding node in $T_{\ell+1}$ can be found in $O(1)$ time.*

We also store one variant of the auxiliary data structures described in Lemma 2.4.6, which support the ball-inheritance problem using $O(n \lg n \cdot \texttt{s}(\sigma))$ bits of space and $O(\texttt{t}(\sigma))$ query time. Here we have (a) $\texttt{s}(\sigma) = O(1)$ and $\texttt{t}(\sigma) = O(\lg^\epsilon \sigma)$; (b) $\texttt{s}(\sigma) = O(\lg \lg \sigma)$ and $\texttt{t}(\sigma) = O(\lg \lg \sigma)$; or (c) $\texttt{s}(\sigma) = O(\lg^\epsilon \sigma)$ and $\texttt{t}(\sigma) = O(1)$. This implies the following lemma:

Lemma 5.5.2. *Given a node x in T_ℓ, where $1 \leq \ell \leq h-1$, its corresponding node in T can be found using $O(n \lg n \cdot \texttt{s}(\sigma))$ bits of additional space and $O(\texttt{t}(\sigma))$ time.*

Now we describe the details of achieving improved query time. Let u and v denote the endpoints of the query path, and let $[p..q]$ denote the query range. We compute $t = \texttt{LCA}(u, v)$. Let $A_{u,t}$ denote the set of nodes on the path from u to and excluding t. Thus the query path can be decomposed into $A_{u,t}$, $A_{v,t}$, and $\{t\}$. We only consider how to report nodes in $A_{u,t} \cap R_{p,q}$, where $R_{p,q}$, as defined in Section 2.5.3, is the set of nodes in T whose weights are in $[p..q]$.

We find the lowest range in the conceptual range tree that covers the query range $[p..q]$. This range, which is denoted by $[a..b]$, can be computed from the lowest common ancestor of the leaf ranges containing p and q. Let k denote the level that contains $[a..b]$. We then locate the nodes x and z in T_k that correspond to $anc_{a,b}(T, u)$ and $anc_{a,b}(T, t)$, respectively.

Lemma 5.5.3. *The nodes x and z can be found using $O(n \lg \sigma)$ bits of additional space and $O(\min\{\lg \lg n + \texttt{t}(\sigma), \lg \sigma / \lg \lg n + 1\})$ time.*

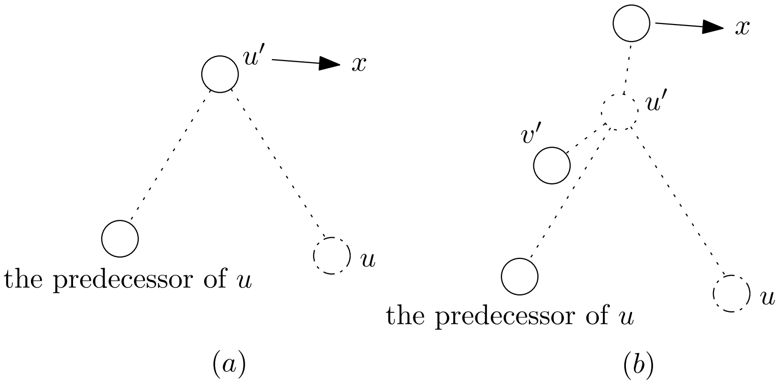


Figure 5.12: An illustration for the proof of Lemma 5.5.3. Normal and dotted circles represent nodes whose weights are in and not in $[a..b]$, respectively. Node u, which could have a weight in $[a..b]$ or not, is represented by a dash dotted circle. (a) The case in which the weight of u' is in $[a..b]$. (b) The case in which the weight of u' is not in $[a..b]$.

Proof. These two nodes can be found using either of the following two approaches. The first approach applies Lemma 5.5.1 repeatedly, which requires $O(k) = O(h) = O(\lg\sigma/\lg\lg n{+}1)$ time.

The second approach is described as follows. For the non-dummy nodes in each $F_{a,b}$, we list the preorder ranks of their corresponding nodes in T as a conceptual array $S_{a,b}$. We maintain $S_{a,b}$ using succinct indices for predecessor search [58], which require $O(\lg\lg n)$ bits per entry and support predecessor and successor queries in $O(\lg\lg n)$ time plus accesses to $O(1)$ entries. These auxiliary indices occupy $O(nh\lg\lg n) = O(n\lg\sigma)$ bits of space over all levels.

As described in Algorithm 6 of Chapter 4, x and z can be found using a constant number of predecessor and successor queries. Here we only describe how to compute x; the computation of z is similar. First we determine, in tree T, the lowest common ancestor, u', of u and the predecessor of u in $S_{a,b}$. If the weight of u' is in $[a..b]$, then, as illustrated in Figure 5.12(a), x corresponds to u' and can be determined by the index of u' in $S_{a,b}$. Otherwise, as illustrated in Figure 5.12(b), we find the successor of u' in $S_{a,b}$ and let the node be v'. The parent of the node in T_k that corresponds to v' will be x.

Summarizing the discussion, the second approach uses $O(\lg\lg n)$ time plus $O(1)$ calls to the ball-inheritance problem. By Lemma 5.5.2, this requires $O(\lg\lg n + \mathtt{t}(\sigma))$ time. Combining these two approaches, the final time cost is $O(\min\{\lg\lg n + \mathtt{t}(\sigma), \lg\sigma/\lg\lg n + 1\})$. □

After determining x and z, we start to report nodes. The query range $[p..q]$ must span more than one child range of $[a..b]$; otherwise $[a..b]$ would not be the lowest range that covers $[p..q]$. Let the child ranges of $[a..b]$ be $[a_1..b_1], [a_2..b_2], \ldots, [a_f..b_f]$, which are listed in increasing order of left endpoints. As described in the proof for Lemma 3.5.6 (see the sentences related to line 4 of Algorithm 3 for details), we can determine in constant time the values of α and β, such that $1 \leq \alpha \leq \beta \leq f$, $[a_\alpha..b_\beta]$ covers $[p..q]$, and $\beta - \alpha$ is minimized. The query range can thus be decomposed into three subranges $[p..b_\alpha]$, $[a_{\alpha+1}..b_{\beta-1}]$ and $[a_\beta..q]$.

The support for the second subrange has been described in Theorem 3.5.8: We first call $\texttt{node\_summarize}(T, x, z)$ and let the result be $\pi[1..f]$. If $\pi[\gamma] = 1$ for $\gamma \in [\alpha+1..\beta-1]$, then we find all nodes whose labels are γ on the path from x to but excluding z by calling $\texttt{lowest\_anc}_\gamma$ repeatedly. Note that each node in the output can be reported by either applying Lemma 5.5.2 repeatedly or Lemma 5.5.1 once, and each 1-bit in $\pi[\alpha+1..\beta-1]$ can be located in constant time using table lookup. Thus it requires $O(\min\{\texttt{t}(\sigma), \lg\sigma/\lg\lg n + 1\})$ time to report a node.

The remaining part is to support the third subrange in the following lemma; the support for the first subrange is similar.

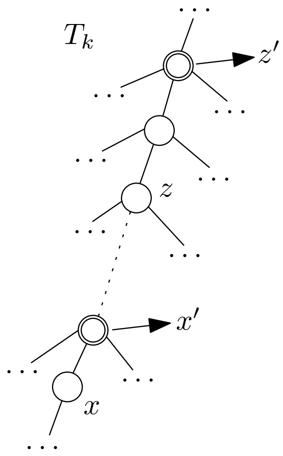


Figure 5.13: The root-to-leaf in T_k that goes through both x and z, where nodes with label β are represented as double circles. The nodes in T_k that correspond to x' and z' can be determined using $\texttt{lowest\_anc}_\beta$ operations.

Lemma 5.5.4. *The nodes in $A_{u,t} \cap R_{a_\beta,q}$ can be reported using $O(n\lg\sigma)$ bits of additional space and $O((|A_{u,t} \cap R_{a_\beta,q}| + 1) \cdot \min\{\texttt{t}(\sigma), \lg\sigma/\lg\lg n + 1\})$ time.*

Proof. We index all T_ℓ's using the first variant of Theorem 5.1.1 with $m = O(n\lg\lg n)$, for which the weight of a node in T_ℓ is defined to be the weight of its corresponding node in T.

Each T_ℓ can be indexed using $O(n \lg\lg n)$ bits of additional space. Hence, these auxiliary data structures occupy $(h-1) \times O(n \lg\lg n) = O(\lg\sigma/\lg\lg n) \times O(n \lg\lg n) = O(n \lg\sigma)$ bits of additional space in total. Path minimum queries over any T_ℓ can be answered with accesses to the weights of $O(1)$ nodes. This requires $O(\mathtt{t}(\sigma))$ time if we use Lemma 5.5.2, or $O(\lg\sigma/\lg\lg n + 1)$ time if we traverse the conceptual range tree level by level using Lemma 5.5.1.

Range $[a_\beta..b_\beta]$ is at the $(k+1)$-st level. Let x' and z' be the nodes in T_{k+1} that correspond to $anc_{a_\beta,b_\beta}(T,u)$ and $anc_{a_\beta,b_\beta}(T,t)$, respectively, as illustrated in Figure 5.13. The nodes x' and z' can be computed from x and z in constant time using $\mathtt{lowest\_anc}_\beta$ operations and Lemma 5.5.1. The nodes in T_{k+1} that correspond to the nodes in $A_{u,t} \cap R_{a_\beta,q}$ must locate on the path from x' to and excluding z'. We make use of path minimum queries to find the node, y', with the minimum weight on this path. This procedure is terminated if the weight of y' is larger than q; otherwise, the node in T that corresponds to y' is reported, and we recurse on two subpaths obtained by splitting the original path at y'. As we perform a path minimum query for each node reported, the query time is $O((|A_{u,t} \cap R_{a_\beta,q}| + 1) \cdot \min\{\mathtt{t}(\sigma), \lg\sigma/\lg\lg n + 1\})$. □

Now we summarize this section in the following theorem.

Theorem 5.5.5. *An ordinal tree on n nodes whose weights are drawn from a set of σ distinct weights can be represented using $O(n \lg n \cdot \mathtt{s}(\sigma))$ bits of space, so that path reporting queries can be supported using $O(\min\{\lg\lg n + \mathtt{t}(\sigma), \lg\sigma/\lg\lg n + 1\} + occ \cdot \min\{\mathtt{t}(\sigma), \lg\sigma/\lg\lg n + 1\})$ query time, where occ is the size of output, ϵ is an arbitrary positive constant, and $\mathtt{s}(\sigma)$ and $\mathtt{t}(\sigma)$ are: (a) $\mathtt{s}(\sigma) = O(1)$ and $\mathtt{t}(\sigma) = O(\lg^\epsilon \sigma)$; (b) $\mathtt{s}(\sigma) = O(\lg\lg\sigma)$ and $\mathtt{t}(\sigma) = O(\lg\lg\sigma)$; or (c) $\mathtt{s}(\sigma) = O(\lg^\epsilon \sigma)$ and $\mathtt{t}(\sigma) = O(1)$.*

Proof. By Lemmas 5.5.3 and 5.5.4, it requires $O(\min\{\lg\lg n + \mathtt{t}(\sigma), \lg\sigma/\lg\lg n + 1\} + (|A_{u,t} \cap R_{p,q}| + 1) \cdot \min\{\mathtt{t}(\sigma), \lg\sigma/\lg\lg n + 1\})$ time to report the nodes in $A_{u,t} \cap R_{p,q}$. The time cost for $A_{v,t} \cap R_{p,q}$ is similar. $\{t\} \cap R_{p,q}$ can be computed in constant time. Summing up these terms, the query time is $O(\min\{\lg\lg n + \mathtt{t}(\sigma), \lg\sigma/\lg\lg n + 1\} + occ \cdot \min\{\mathtt{t}(\sigma), \lg\sigma/\lg\lg n + 1\})$, where *occ* is the size of output. Due to Lemma 5.5.2, the overall space cost is $O(n \lg n \cdot \mathtt{s}(\sigma))$ bits. □

The data structures developed in Theorem 5.5.5 match the state of the art of two-dimensional orthogonal range reporting queries [27] when $\sigma = n$. In Section 5.6, we further refine these data structures for the case in which $\sigma < n$.

5.6 Further Refinements for Range and Path Reporting

Now we further improve the data structures described in Theorem 5.5.5. The refined data structures incur lower cost in terms of both space and time. The space cost is $O(n \lg \sigma \cdot \mathtt{s}(\sigma))$ bits instead of $O(n \lg n \cdot \mathtt{s}(\sigma))$ bits, while the query time is $O(\min\{\lg\lg\sigma + \mathtt{t}(\sigma), \lg\sigma/\lg\lg n + 1\} + occ \cdot \min\{\mathtt{t}(\sigma), \lg\sigma/\lg\lg n + 1\})$ instead of $O(\min\{\lg\lg n + \mathtt{t}(\sigma), \lg\sigma/\lg\lg n + 1\} + occ \cdot \min\{\mathtt{t}(\sigma), \lg\sigma/\lg\lg n + 1\})$, where $\mathtt{s}(\sigma)$ and $\mathtt{t}(\sigma)$ are defined as in Theorem 5.5.5.

To illustrate our idea, we first develop refined data structures for the two-dimensional orthogonal range reporting problem [27]. In this problem, a set of n points on an $n \times n$ grid is given, and a query asks for the points in an axis-aligned rectangle. Here we consider a more general version of this problem, for which points are drawn from an $n \times \sigma$ grid, where $\sigma \leq n$.

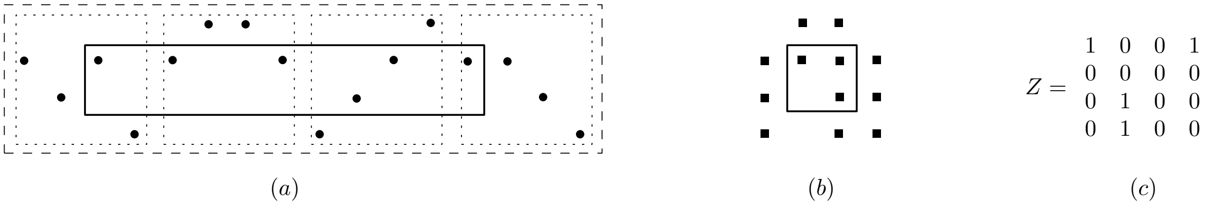


Figure 5.14: An illustration for Theorem 5.6.1. (a) An input point set on a 16×4 grid, which is represented by the dashed rectangle. The dotted rectangles each represent a subgrid and the bold rectangle represents a range query Q. (b) The compressed grid that corresponds to the input point set, where the bold rectangle represents the subquery Q_3. (c) The 01-matrix Z that corresponds to the compressed grid.

Theorem 5.6.1. *A set of n points on an $n \times \sigma$ grid, where $\sigma \leq n$, can be represented using $O(n \lg \sigma \cdot \mathtt{s}(\sigma))$ bits of space, such that range reporting queries can be supported in $O(\min\{\lg\lg\sigma + \mathtt{t}(\sigma), \lg\sigma/\lg\lg n + 1\} + occ \cdot \min\{\mathtt{t}(\sigma), \lg\sigma/\lg\lg n + 1\})$ time, where occ is the size of output, ϵ is an arbitrary positive constant, and $\mathtt{s}(\sigma)$ and $\mathtt{t}(\sigma)$ are: (a) $\mathtt{s}(\sigma) = O(1)$ and $\mathtt{t}(\sigma) = O(\lg^{\epsilon}\sigma)$; (b) $\mathtt{s}(\sigma) = O(\lg\lg\sigma)$ and $\mathtt{t}(\sigma) = O(\lg\lg\sigma)$; or (c) $\mathtt{s}(\sigma) = O(\lg^{\epsilon}\sigma)$ and $\mathtt{t}(\sigma) = O(1)$.*

Proof. As described in the proof of [18, Lemma 7], we can assume that all the given points have distinct x-coordinates. As illustrated in Figure 5.14(a), we partition the $n \times \sigma$ grid into $\lceil n/\sigma \rceil$ subgrids, for which the i-th subgrid spans over $[(i-1)\sigma + 1..i\sigma] \times [1..\sigma]$ for $1 \leq i < \lceil n/\sigma \rceil$, and the last one spans over $[(\lceil n/\sigma - 1 \rceil\sigma + 1)..n] \times [1..\sigma]$. Thus each subgrid except the last contains σ input points, and the last one contains at most σ points.

Let $Q = [x_1..x_2] \times [y_1..y_2]$ be the given query. Thus Q spans over the α-th to the β-th subgrids, where $\alpha = \lceil x_1/\sigma \rceil$ and $\beta = \lceil x_2/\sigma \rceil$. We only consider the cases in which $\alpha < \beta$; the other cases can be handled similarly. Q can be split into three subqueries: Q_1, the intersection with the α-th subgrid; Q_2, the intersection with the β-th subgrid; and finally Q_3, the intersection with $(\alpha+1)$-st to the $(\beta-1)$-st subgrids. These subqueries are supported as follows.

For each subgrid, we build the data structures of Bose et al. [18] and one variant of Chan et al.'s [27] structures, which have been summarized in Table 5.1. These data structures use $O(\sigma \lg \sigma + \sigma \lg \sigma \cdot \mathtt{s}(\sigma)) = O(\sigma \lg \sigma \cdot \mathtt{s}(\sigma))$ bits of space for each subgrid, so the overall space cost is $\lceil n/\sigma \rceil \times O(\sigma \lg \sigma \cdot \mathtt{s}(\sigma)) = O(n \lg \sigma \cdot \mathtt{s}(\sigma))$ bits. In addition, range reporting queries within a subgrid, e.g., Q_1 or Q_2, can be supported using $O(\min\{\lg\lg\sigma + \mathtt{t}(\sigma), \lg\sigma/\lg\lg n + 1\} + occ' \cdot \min\{\mathtt{t}(\sigma), \lg\sigma/\lg\lg n + 1\})$ time, where occ' is the size of answer.

To support Q_3, we transform the input point set from the original grid into a compressed grid of size $\lceil n/\sigma \rceil \times \sigma$, where a hyperpoint corresponds to one or more points in the original point set. More precisely, an input point (x, y), which is contained in the $\lceil x/\sigma \rceil$-th subgrid, is transformed into a *hyperpoint* $(\lceil x/\sigma \rceil, y)$. See Figure 5.14(b) for an illustration. For each hyperpoint in the compressed grid, we explicitly store the input points that correspond to this hyperpoint as a linked list. Since a subgrid consists of at most σ points, each link to a point requires only $O(\lg \sigma)$ bits and the overall space cost of these linked lists is $O(n \lg \sigma)$ bits.

To answer Q_3, we need to find all the hyperpoints contained in $Q'_3 = [\alpha + 1..\beta - 1] \times [y_1..y_2]$. As illustrated in Figure 5.14(c), we construct a 01-matrix $Z[1..\lceil n/\sigma \rceil, 1..\sigma]$ in which $Z[i, j] = 0$ iff there exists a hyperpoint (i, j). We then encode Z using Brodal et al.'s [20] data structure for two-dimensional range minimum queries, which occupies only $O(\lceil n/\sigma \rceil \times \sigma) = O(n)$ bits of space and can return the smallest entry in any given submatrix of Z using $O(1)$ query time. To determine all the hyperpoints contained in Q'_3, we find all 0-entries in $Z[\alpha + 1..\beta - 1, y_1..y_2]$ by repeatedly performing range minimum queries. Initially, we query on $Z[\alpha + 1..\beta - 1, y_1..y_2]$ for any 0-entry. If there exists some entry $Z[i, j] = 0$, then we know that hyperpoint (i, j) is contained in Q'_3. Furthermore, we divide the remaining entries of $Z[\alpha + 1..\beta - 1, y_1..y_2]$ into up to 4 disjoint submatrices and query on them recursively. If no such $Z[i, j]$ exists, then the algorithm terminates and we conclude that there is no more hyperpoint in Q'_3. Observe that we recurse on at most 4 disjoint submatrices only when a hyperpoint is found. Thus, the hyperpoints contained in Q'_3 can be found using $O(1)$ time per hyperpoint. By traversing the linked lists associated to these hyperpoints, the points contained in Q_3 can be returned in $O(1)$ time per point.

Summarizing the discussion, the overall space cost is $O(n \lg \sigma \cdot \mathtt{s}(\sigma))$ bits, and queries can be answered in $O(\min\{\lg\lg\sigma + \mathtt{t}(\sigma), \lg\sigma/\lg\lg n + 1\} + occ \cdot \min\{\mathtt{t}(\sigma), \lg\sigma/\lg\lg n + 1\})$ time, where *occ* is the size of output. □

We finally generalize the approach of Theorem 5.6.1 into weighted trees and complete the proof for Theorem 5.1.3. The cases in which $n = O(\sigma^2)$ have already been handled by Theorem 5.5.5 ($\lg\lg n = O(\lg\lg\sigma)$ in this case). Here we only consider the cases in which $n = \omega(\sigma^2)$.

As in Section 5.2, we make use of Lemma 2.5.1 on T with $M = \lceil\sigma^2\rceil$. Thus we obtain $O(n/M)$ cover elements, each being a subtree of size at most $2M$. The root node of a cover element is called a *cover root*. It should be noted again that a cover root could be the root of multiple cover elements. For simplicity, we denote by s_i the i-th cover root in preorder of T. We define the following auxiliary operations with respect to cover roots. Here x is assumed to be a cover root.

- `cover_rank`(T, x): the number of cover roots preceding x in preorder of T;
- `cover_select`(T, i): s_i, i.e., the i-th cover root in preorder of T;
- `cover_depth`(T, x): the number of cover roots between x and root of T;
- `cover_anc`(T, x, i): the i-th lowest cover root along the path from x to the root of T.

Lemma 5.6.2. *The auxiliary operations* `cover_rank`*,* `cover_select`*,* `cover_depth` *and* `cover_anc` *can be supported using $O(1)$ time and $O(n(\lg M)/M)$ bits of additional space.*

Proof. We store a bit vector $B[1..n]$ to mark cover roots, for which $B[j] = 1$ if the j-th node in preorder of T is a cover root. By Lemma 2.3.1, B can be stored in $O(n(\lg M)/M)$ bits of additional space. We can make use of $\texttt{rank}_\alpha$ and $\texttt{select}_\alpha$ operations to compute the rank of a cover root and select the i-th cover root in preorder, respectively.

In addition, we extract all cover roots from T using tree extraction. This gives us a single ordinal tree T', since the root of T must be a cover root. T' is represented using Lemma 2.5.2, for which we do not store any weight in T'. The overall space cost of storing T' is $O(n/M)$ additional bits, since T' consists of $O(n/M)$ nodes. Since tree extraction preserves ancestor-descendant relationship, `cover_depth` and `cover_anc` can be reduced to `depth` and `level_anc` operations in T', respectively. Therefore, they can be supported in constant time. □

To support path reporting queries, we build the data structures of Theorem 5.5.5 for each cover element, such that queries inside a cover element can be supported using $O(\min\{\lg\lg\sigma + \texttt{t}(\sigma), \lg\sigma/\lg\lg n + 1\} + occ' \cdot \min\{\texttt{t}(\sigma), \lg\sigma/\lg\lg n + 1\})$ time and $O(M\lg\sigma\cdot\texttt{s}(\sigma))$ bits of space, where occ' is the size of output. The space cost over all cover elements is $O(n/M) \times O(M(\lg\sigma)\cdot\texttt{s}(\sigma)) = O(n(\lg\sigma)\cdot\texttt{s}(\sigma))$ bits.

For each cover root s_i, we denote by $\texttt{Path}_i$ the path from s_i to but excluding the root of the lowest cover element that is an ancestor of s_i, i.e., $\texttt{Path}_i$ contains exactly the nodes in A_{s_i,s_j} for $s_j = \texttt{cover\_anc}(T, s_i, 1)$. In particular, $\texttt{Path}_1$ is empty since s_1 is the root node of T. Note that the length of $\texttt{Path}_i$ is bounded above by $O(\sigma^2)$. We store this path in $O(M \lg \sigma)$ bits of space using the data structures of Alstrup et al. [4], such that given a query range of weights, the nodes along this path whose weights are within this range can be reported in $O(occ' + 1)$ time, where occ' is the number of nodes in the query range. The space cost over all cover elements is $O(n/M) \times O(M \lg \sigma) = O(n \lg \sigma)$ bits.

The last auxiliary data structures are $\sigma(\sigma - 1)/2$ ordinal trees, $T^{p,q}$ for $1 \leq p \leq q \leq \sigma$, all of which are of the same structure as T' (defined in the proof of Lemma 5.6.2) but on 01-labeled nodes. For each $T^{p,q}$, we assign 1 to a node if the node corresponds to some cover root s_i and $\texttt{Path}_i$ has a node whose weight is in $[p..q]$; otherwise we assign 0 to this node. See Figure 5.15 for an example. We maintain these 01-labeled trees using the data structures described in Lemma 2.5.3, such that the lowest ancestor whose label is 1 of a given node (i.e., $\texttt{lowest\_anc}_1$) can be found in constant time. Each $T^{p,q}$ requires $O(n/M)$ bits of space, and thus the overall space cost for storing all these labeled trees is $O(\sigma^2) \times O(n/M) = O(n)$ bits.

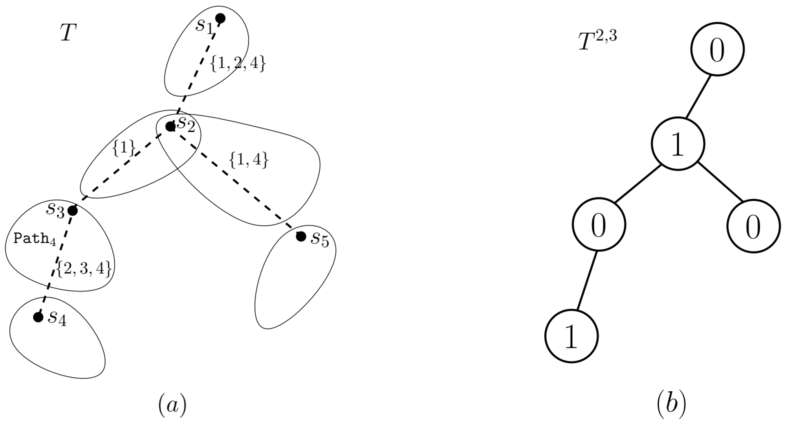


Figure 5.15: (a) An input tree T with $\sigma = 4$ and 6 cover elements. The dashed lines represent $\texttt{Path}_i$'s, and the numbers alongside each $\texttt{Path}_i$ represent the set of weights on this path. (b) The 01-labeled tree $T^{2,3}$.

Finally we consider how to answer a given query. Let u and v denote the endpoints of the query path, and let $[p..q]$ denote the query range. As in Theorem 5.5.5, we only consider the support for $A_{u,t} \cap R_{p,q}$, where t is the lowest common ancestor of u and v, and $A_{u,t}$ is the set of nodes on the path from u to and excluding t.

Let s_a and s_b be the lowest and the highest cover root on the path from u to and excluding t, respectively. The following lemma shows how to compute them.

Lemma 5.6.3. *The nodes s_a and s_b can be computed in $O(1)$ time.*

Proof. The node s_a, which is the lowest cover root on the path from u to t, must be the root of the cover element that contains u if it exists. Following the approach described in Algorithm 6 of Chapter 4, we can locate s_a in constant time. To compute the highest cover root s_b on the path, we first locate s_c, the root of the cover element that contains t. The highest cover root can be expressed as $s_b = \texttt{cover\_anc}(T, s_a, i)$ for $i = \texttt{cover\_depth}(T, s_a) - \texttt{cover\_depth}(T, s_c) - 1$. □

To answer the query, we only consider the case in which $s_a \neq s_b$; the other cases can be handled similarly. Thus $A_{u,t}$ can be decomposed into A_{u,s_a}, A_{s_a,s_b}, and $A_{s_b,t}$. Here A_{u,s_a} is contained in the cover element rooted at s_a, and $A_{s_b,t}$ except node s_b is contained in the cover element rooted at s_b. The nodes along these two subpaths whose weights are in $[p..q]$ can be reported using the data structures described in Theorem 5.5.5, which have already been stored for each cover element. The query time is $O(\min\{\lg\lg\sigma + \texttt{t}(\sigma), \lg\sigma/\lg\lg n + 1\} + (A_{u,s_a} \cup A_{s_b,t}) \cap R_{p,q} \cdot \min\{\texttt{t}(\sigma), \lg\sigma/\lg\lg n + 1\})$.

The subpath from s_a to but excluding s_b can be handled by $T^{p,q}$. Using $\texttt{lowest\_anc}_1$ operations, we can find all cover roots s_i between s_a and s_b such that $\texttt{Path}_i$ has a node whose weight is between p and q, using constant time per s_i. Then, for each of these cover roots s_i, we report all the nodes whose weights are between p and q on $\texttt{Path}_i$, using constant time per node. This is supported by the data structures of Alstrup et al. [4], which have been stored for each $\texttt{Path}_i$.

In sum, the overall space cost is $O(n\lg\sigma\cdot\texttt{s}(\sigma))$ bits, and the query time is $O(\min\{\lg\lg\sigma + \texttt{t}(\sigma), \lg\sigma/\lg\lg n + 1\} + \mathit{occ}\cdot\min\{\texttt{t}(\sigma), \lg\sigma/\lg\lg n + 1\})$, where *occ* is the size of output.

5.7 Discussion

We end this chapter with an open problem about the adaptive encoding complexity of path minimum queries. As shown in Lemma 5.2.1, $\Omega(n\lg n)$ bits are necessary to encode the answers to all possible path minimum queries in the worst cases. However, the family of trees we constructed for this lower bound have $\Theta(n)$ leaves. As another extreme case, this problem becomes the well-known RMQ problem when the input tree is a single path (or has only two leaves), and thus can be encoded in $2n$ bits. It would be interesting to examine the cases in which the number of leaves, n_L, satisfies that $n_L = o(n)$ and $n_L > 2$.

Chapter 6

Dynamic Data Structures for Path Queries

6.1 Introduction

Most previous work on path queries, including our work described in Chapters 3 and 5, focuses on static weighted trees, i.e., the structure and the weights of nodes remain unchanged over time. This assumption is not always realistic and it is highly inefficient to rebuild the whole data structure when handling updates. In this chapter, we consider the problem of maintaining dynamic weighted trees and design linear space data structures that support a variety of path queries in efficient query/update time. We consider path counting, path reporting, path selection and path median queries, as defined in Chapter 3. As mentioned in the same chapter, these path queries generalize two-dimensional range counting, two-dimensional range reporting, range selection and range median queries, respectively.

Without loss of generality, we represent the input tree as an ordinal one. Our data structures allow us to change the weight of an existing node, insert a new node, or delete an existing node. These updates are referred to as `modify_weight`, `node_insert`, and `node_delete`, respectively. For `node_insert` and `node_delete`, we adopt the same powerful updating protocol used by Navarro and Sadakane [82], which enables us to insert or delete a leaf, a root, or an internal node. A newly inserted internal node will become the parent of a set of consecutive children of an existing node, and a deleted root must have only zero or one child before deletion. The deletion of a non-root node is described in Section 2.5.3.

In static ordinal trees, it is natural to identify nodes with their preorder ranks. However, the preorder rank of a node can change over time in dynamic trees. Thus, to specify query paths in our dynamic data structures, nodes are identified by immutable identifiers of sizes $O(\lg n)$ bits such as pointers to these nodes. The identifier of a node is given when a node is created.

Throughout this chapter, we use ϵ to denote an arbitrary constant between 0 and 1. Unless otherwise specified, the underlying model of computation in this chapter is the unit-cost word RAM model with word size $w = \Omega(\lg n)$.

Previous Work. We refer to Sections 3.1.1 and 5.1.1 for a survey on static path counting, path reporting, path selection, path median and path minimum queries. Here we will focus on dynamic data structures.

The dynamic version of the path minimum problem has been studied extensively. Brodal et al. [21] designed a linear space data structure that supports queries and changes to the weight of a node in $O(\lg n/ \lg\lg n)$ time, and handles insertions or deletions of a node with zero or one child in $O(\lg n/ \lg\lg n)$ amortized time. The query time is optimal under the cell probe model provided that the update time is $O(\lg^{O(1)} n)$ [6]. For the more restricted case in which only insertions and deletions of leaves are allowed, queries can be answered in $O(1)$ time and updates can be supported in $O(1)$ amortized time [5, 72, 21].

In the dynamic orthogonal range query problems, a dynamic point set in the plane with real coordinates is maintained such that, given a rectangular query range, certain information over the points contained in the query range can be retrieved efficiently. He and Munro [62] presented a linear-space data structure that supports range counting queries in $O((\lg n/ \lg\lg n)^2)$ time, as well as insertions and deletions in $O((\lg n/ \lg\lg n)^2)$ amortized time. More recently, He et al. [63] designed a dynamic data structure for range selection queries with the same space cost and query/update time. For range reporting queries, Blelloch [17] developed a linear-space data structure with $O(\lg n + occ \lg n/ \lg\lg n)$ query time and $O(\lg n)$ amortized update time, where occ is the size of output. Nekrich [83] designed another $O(n)$-word data structure that require only $O(\lg n + occ \lg^{\epsilon} n)$ query time but $O(\lg^{7/2} n)$ update time, where $\epsilon > 0$ is an arbitrary constant.

Our contributions. We develop efficient dynamic data structures for path counting and path reporting queries, all of which occupy $O(n)$ words. Our data structure supports path counting queries and path selection queries in $O((\lg n/ \lg\lg n)^2)$ time, and handles changes of weights, insertions and deletions in $O((\lg n/ \lg\lg n)^2)$ amortized time. This structure matches the best known results for dynamic range counting [62] and dynamic range selection [63].

For path reporting queries, our data structure requires $O(\lg^{2+\epsilon} n)$ time for updates, but answers queries in $O((\lg n/ \lg\lg n)^2 + occ \cdot \lg n/ \lg\lg n))$ time, where occ is the output size. By slightly sacrificing the update time, our structure significantly improves the query time over the straightforward approaches that dynamize known static data structures [65, 85] or the data structures presented in Theorems 3.5.8 and 5.5.5: One could dynamize the static structure of He et al. [65] by replacing static labeled ordinal trees with dynamic unlabeled trees and dynamic bit vectors, and managing weight ranges using a red-black tree [37]. This leads to an $O(n)$-word data structure with $O((1 + occ) \cdot \lg^2 n / \lg \lg n)$ query time and $O(\lg^2 n / \lg \lg n)$ update time. Alternatively, one could obtain another $O(n)$-word structure with $O(\lg^{2+\epsilon} n + occ \cdot (\lg n / \lg \lg n)^2)$ query time and $O((\lg n / \lg \lg n)^2)$ update time, by dynamizing the improved data structure presented in Theorem 3.5.8 in a similar fashion. It is unclear how to dynamize the structures designed by Patil et al. [85] and the structures presented in Theorem 5.5.5 within linear space.

All of our dynamic structures presented in this chapter are able to handle insertions and deletions of nodes with multiple children, which are not supported in previous dynamic data structures for path queries [5, 72, 21]. To develop our data structures, we employ a variety of techniques including directed topology trees, tree extraction, and balanced parentheses. In particular, for dynamic path reporting, one key strategy is to carefully design transformations on trees that preserve certain properties, such that the idea of dynamic fractional cascading can be adapted to work on multiple datasets in which each set represents tree-structured data. This new approach may be of general interest.

The rest of this chapter is organized as follows. In Section 6.2 we present the main result of this chapter, i.e., our dynamic data structures for path reporting queries. Later, in Section 6.3 we describe how to support path counting queries and path selection queries. Finally in Section 6.4, we end this chapter with open problems.

6.2 Dynamic Path Reporting

To represent a dynamic weighted tree T on n nodes to support path reporting, assume without loss of generality that node weights are distinct. We construct a weight-balanced B-tree [7], W, with leaf parameter 1 and branching factor $d = \lceil \lceil \lg n \rceil^\epsilon \rceil$ for any positive constant ϵ less than $1/5$. When the value of d changes due to updates, as to be shown later, we reconstruct the entire data structure and amortize the cost of rebuilding to updates. By the properties of weight-balanced B-trees, each internal node of W has at least $d/4$ and at most $4d$ children, and the only exception is the root which is allowed to have fewer children. Each leaf of W represents a weight range $[a, b)$, where a and b are weights assigned to nodes of T, and there is no node of T whose weight is strictly between a and b. An internal node of W represents a (contiguous) range which is the union of the ranges represented by its children. The levels of W are numbered $0, 1, 2, \ldots, t$, starting from the leaf level, where

$t = O(\lg n / \lg\lg n)$ denotes the number of the root level. The tree structure of W together with the weight range represented by each node is maintained explicitly.

For each internal node v of W, we conceptually construct a tree $T(v)$ as follows: Let $[a, b)$ denote the weight range represented by v. We construct a tree $T_{[a,b)}$ consisting of nodes of T whose weights are in $[a, b)$ using the tree extraction approach described in Section 2.5.3. For each node x in $T_{[a,b)}$, we then assign an integer label $i \in [1..4d]$ if the weight of x is within the weight range of the i-th child of v. The resulting labeled tree is $T(v)$.

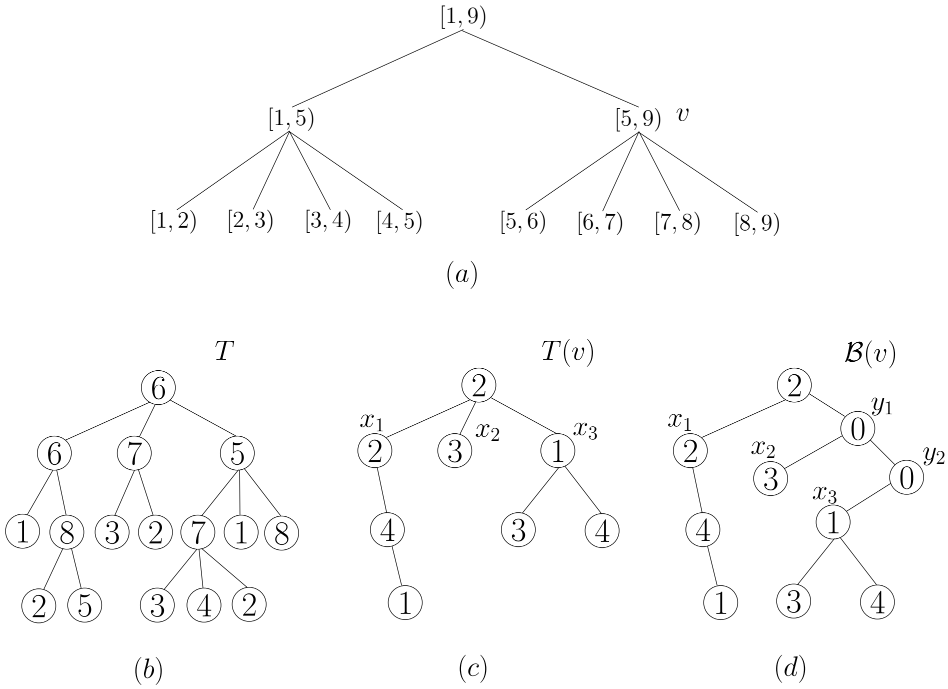


Figure 6.1: (a) A weight-balanced B-tree with branching factor $d = 2$ for $[1, 9)$, where the node v corresponds to $[5, 9)$; (b) an input tree T with $n = 16$ nodes; (c) the labeled tree $T(v)$ extracted from T with respect to the node v; and (d) the transformed binary tree $\mathcal{B}(v)$ for $T(v)$, where two dummy nodes y_1 and y_2 are inserted for siblings x_1, x_2, and x_3.

We do not store each $T(v)$ explicitly. Instead, following the approach described in Section 5.2.2, we transform the tree structure of each $T(v)$ into a binary tree $\mathcal{B}(v)$: For each node x of $T(v)$ with $k > 2$ children denoted by $x_1, x_2, \ldots, x_k$, we add $k - 1$ dummy nodes $y_1, y_2, \ldots, y_{k-1}$. Then, x_1 and y_1 become the left and the right children of x, respectively. For $i = 1, 2, \ldots, k - 2$, the left and the right children of y_i are set to be x_{i+1} and y_{i+1}, respectively. Finally, x_k becomes the left and only child of y_{k-1}. This way x and its children are transformed into a right-leaning binary tree whose leaves are the children of x in $T(v)$. In $\mathcal{B}(v)$, the node corresponding to the dummy root of $T(v)$ is also considered a dummy node, and a node is called an *original node* if it is not a dummy node. We observe that this transformation preserves the preorder and postorder relationships among the original nodes in $T(v)$. Furthermore, the set of original nodes along the path between any two original nodes remains unchanged after transformation. Each original node in $\mathcal{B}(v)$ is associated with its label in $T(v)$, which is an integer in $[1..4d]$, while each dummy node is assigned with label 0. See Figure 6.1 for illustrations.

Let F_ℓ denote the forest containing all the binary trees created for the nodes at the ℓ-th level of W for $\ell > 0$, i.e., $F_\ell = \{\mathcal{B}(v) : v \text{ is a node at the } \ell\text{-th level of } W\}$ for $\ell \in [1..t]$. Thus F_t contains only one binary tree which corresponds to the root of W, and this tree contains all the nodes of the given tree T as original nodes. This allows us to maintain a bidirectional pointer between each node in T and its corresponding original node in F_t.

W and T are stored using standard, pointer-based representations of trees. In the rest of this section, we first present, in Section 6.2.1, a data structure that can be used to maintain a dynamic forest in which each node is assigned a label from an alphabet of sub-logarithmic size, to support a set of operations including *path summary* queries which is to be defined later. This structure is of independent interest and will be used to encode each F_ℓ. We next show, in Section 6.2.2, how to maintain pointers between forests constructed for different levels of W, which will be used to locate appropriate nodes of these forests when answering path reporting queries. Finally we describe how to answer path reporting queries and perform updates in weighted trees in Section 6.2.3.

6.2.1 Representing Dynamic Forests with Small Labels to Support Path Summary Queries

We now describe a data structure which will be used to encode F_ℓ in subsequent subsections. As this structure may be of independent interest, we formally describe the problem its addresses as follows. Let F be a dynamic forest of binary trees on n nodes in total, in which each node is associated with a label from the alphabet $[0..\sigma]$, where $\sigma = O(\lg^\epsilon n)$ for an arbitrary constant $\epsilon \in (0, 1/5)$. Our objective is to maintain F to support `link`, `cut` (see Lemma 2.5.6 for the definitions) and the following operations:

- $\texttt{parent}_\alpha(x)$: return the α-parent of node x, i.e., the lowest ancestor of x that has label α, which can be x itself.
- $\texttt{LCA}(x, y)$: return the lowest common ancestor of two given nodes x and y residing in the same binary tree.

- $\texttt{pre\_succ}_\alpha(x)$: return the α-successor of x in preorder, i.e., the first α-node in preorder that succeeds x (this could be x itself).
- $\texttt{post\_pred}_\alpha(x)$: return the α-predecessor of x in postorder, i.e., the last α-node in postorder that precedes x (this could be x itself).
- $\texttt{summary}(x, y)$: given two nodes x and y residing in the same binary tree, return a bit vector of $\sigma + 1$ bits in which the α-th bit is 1 iff there exists an α-node along the path from x to y. This query is called path summary.
- $\texttt{modify}(x, \alpha)$: change the label of x to α.

First we set $s = \lceil \frac{\lceil \lg n \rceil}{\lg \lceil \lg n \rceil} \rceil$ in Lemma 2.5.6, and use the lemma to maintain the directed topology trees of the binary trees in F. We call each base cluster a *micro-tree*.

Next we mark a subset of levels of the directed topology trees. For $\ell = 0, 1, 2, \ldots$, the ℓ-th *marked level* of a directed topology tree is level $\ell \lfloor 4\epsilon \lg \lg n \rfloor$ of this directed topology tree. Since in a directed topology tree, the restricted partition at each level except level 0 is of order 2, each internal node of the directed topology tree has at most two children. Therefore, for $\ell \geq 1$, each cluster at the ℓ-th marked level contains at most $2^{\lfloor 4\epsilon \lg \lg n \rfloor} \leq 2^{4\epsilon \lg \lg n} = \lg^{4\epsilon} n$ clusters at the $(\ell - 1)$-st marked level. We then define the *macro-tree* for a node at the ℓ-th marked level of a directed topology tree, for $\ell \geq 1$, to be the tree obtained by taking its descendant nodes at the $(\ell - 1)$-st marked level and preserving the adjacency relationships among the corresponding clusters, i.e., we add an edge between two of these descendant nodes if and only if their corresponding clusters are adjacent.

A macro-tree constructed for some node at the ℓ-th marked level of the directed topology tree is said to be a *tier-ℓ* macro-tree. A tier-ℓ macro-tree is a binary tree with at most $\lg^{4\epsilon} n$ nodes, which each correspond to a cluster at the $(\ell - 1)$-st marked level. Note that $\lg^{4\epsilon} n = O(\lg^{\epsilon'} n)$ for any $\epsilon' \in (4\epsilon, 1)$. A node in a tier-$\ell$ macro-tree is called a *boundary node* if its corresponding cluster contains the endpoint of an edge that has only one endpoint in this tier-ℓ macro-tree. By the properties of restricted multilevel partitions, each macro-tree has at most two boundary nodes, and at most one of them connects to a child cluster, which is referred to as the *lower boundary* of the macro-tree. The root of a macro-tree is its *upper boundary*, if this node is not the root of the entire tree. We define lower and upper boundaries for any cluster in a similar way. For simplicity, we use $\mu[i]$ to denote the i-th node in preorder of some micro-tree or macro-tree μ. We also denote by $C(\mu)$ and $C(\mu[i])$ the clusters that are represented by a macro-tree μ and the i-th node $\mu[i]$ in preorder of μ, respectively.

We construct auxiliary data structures for each micro-tree and macro-tree. Our main idea is to create structures that can fit in $o(\lg n)$ bits (in addition to maintaining pointers such as those that can be used to map macro-tree nodes to macro-trees at the lower marked level), so that we can construct $o(n)$-bit lookup tables to perform operations in each micro-tree or macro-tree. Operations over F are then supported by operating on a constant number of micro-trees and a constant number of macro-trees at each marked level.

The rest of this section will prove the following lemma.

Lemma 6.2.1. *Let F be a dynamic forest of binary trees on n nodes in total, in which each node is associated with a label from the alphabet $[0..\sigma]$, where $\sigma = O(\lg^{\epsilon} n)$ for some constant $\epsilon \in (0, 1/5)$. F can be represented in $O(n \lg\lg n + f \lg n)$ bits to support* $\texttt{parent}_{\alpha}$, LCA, summary, $\texttt{pre\_succ}_{\alpha}$, $\texttt{post\_pred}_{\alpha}$ *and* modify *in $O(\lg n/ \lg\lg n)$ time, and* link *and* cut *in $O(\lg^{1+4\epsilon} n)$ time, where f is the current number of trees in F.*

To prove Lemma 6.2.1, we first describe the auxiliary structures constructed for micro-trees and macro-trees. For each micro-tree μ, we store:

- A representation of the labeled tree μ in the form of a triple $\{|\mu|, T_{\mu}, L_{\mu}\}$, where $|\mu|$ is the number of nodes in μ, T_{μ} is a binary sequence that encodes the structure of μ, and L_{μ} encodes the labels assigned to the nodes of μ in preorder.
- An integer b_{μ} that stores the preorder rank in T_{μ} of the lower boundary of μ.
- A pair (u_{μ}, p_{μ}), in which u_{μ} is a pointer that points to the tier-1 macro-tree containing μ, and p_{μ} is the preorder rank of the node in the tier-1 macro-tree.

For each tier-ℓ macro-tree μ, we also construct $\{|\mu|, T_{\mu}\}$, b_{μ} and, pair (u_{μ}, p_{μ}), with the necessary change that u_{μ} points to the tier-$(\ell+1)$ macro-tree containing μ. In addition, we construct the following:

- An array $S_{\mu}[1..|\mu|]$, in which $S_{\mu}[i] = j$ if $C(\mu[i])$ has j preorder segments. Note that $j \in [1..2]$.
- A two-dimensional array $I_{\mu}[1..|\mu|]$, where $I_{\mu}[i]$ stores a bit vector of length $\sigma + 1$ in which the α-th bit is 1 iff $C(\mu[i])$ contains an α-node on the path from its lower boundary to its root (if $C(\mu[i])$ has no lower boundary, then 0-bits are stored in $I_{\mu}[i]$).
- A two-dimensional array $J_{\mu}[1..|\mu|][1..2]$, where $J_{\mu}[i][j]$ stores a bit vector of length $\sigma + 1$ in which the α-th bit is 1 iff the j-th preorder segment of $C(\mu[i])$ contains an α-node (if this cluster has fewer than j preorder segments, then 0-bits are stored in $J_{\mu}[i][j]$).

- An array $D_\mu[1..|\mu|]$, in which $D_\mu[j]$ stores a pointer to the tier-$(\ell-1)$ macro-tree (or, if $\ell = 1$, the micro-tree) that corresponds to $\mu[j]$.

We then prove the following set of lemmas that will show the correctness of Lemma 6.2.1:

Lemma 6.2.2. *The data structures constructed in this section use $O(n \lg\lg n + f \lg n)$ bits in total, where f is the current number of trees in F.*

Proof. For each micro-tree μ, its size $|\mu|$ and its lower boundary b_μ can be encoded in $O(\lg s) = O(\lg\lg n)$ bits. The pair (u_μ, p_μ) can be encoded in $O(\lg n)$ bits. T_μ can be represented as balanced parentheses [80] using $O(|\mu|)$ bits. L_μ, which stores labels of nodes, occupies $|\mu| \times \lceil \lg(\sigma+1) \rceil = O(|\mu| \lg\lg n)$ bits of space. Thus the space cost of storing the micro-tree μ is $O(\lg n + |\mu| \lg\lg n)$ bits.

The analysis for macro-trees is almost the same — except for the arrays S_μ, I_μ, J_μ and D_μ. S_μ can be stored in $O(|\mu|)$ bits, while I_μ and J_μ can be encoded in $|\mu| \times (\sigma+1) \times 3 = |\mu| \times O(\lg^\epsilon n) \times 3 = O(|\mu| \lg^\epsilon n)$ bits. In addition, D_μ occupies $|\mu| \times O(\lg n) = O(|\mu| \lg n)$ bits of space. Thus the space cost of storing the macro-tree μ is $O(|\mu| \lg n)$ bits.

By Lemma 2.5.6, F is partitioned into $O(f + n/s) = O(f + n \lg\lg n/\lg n)$ disjoint micro-trees. The overall number of nodes of all macro-trees is also $O(f + n \lg\lg n/\lg n)$, as each micro-tree corresponds to a node of a tier-1 macro-tree. Therefore, the space cost for all these micro-trees and macro-trees is $O(f \lg n + n \lg\lg n)$ bits, as the cost for all micro-trees is $O(f + n \lg\lg n/\lg n) \times O(\lg n) + n \times O(\lg\lg n) = O(f \lg n + n \lg\lg n)$ bits, and the cost for all macro-trees is $O(f + n \lg\lg n/\lg n) \times O(\lg n) = O(f \lg n + n \lg\lg n)$ bits. Adding the cost for the directed topology trees, the data structures constructed in this section occupy $O(f \lg n + n \lg\lg n)$ bits of space in total. □

In the following lemmas, we describe how to support the operations defined at the beginning of this subsection. In these algorithms, nodes are identified by a pointer to the micro-tree containing it and its preorder rank inside the micro-tree.

Lemma 6.2.3. *With an $o(n)$-bit universal lookup table, the data structures constructed in this section can support* $\texttt{parent}_\alpha$ *in $O(\lg n/\lg\lg n)$ time.*

Proof. We first extend the definitions of α-ancestors to nodes of macro-trees. Let z be a node in a macro-tree and let z' be its ancestor. Node z' is said to be an α-ancestor of z if $C(z')$ contains an α-ancestor of the root of $C(z)$. We create a lookup table G_1 that stores, for any possible combination of $\{|\mu|, T_\mu\}$ and I_μ, and any $i \in [1..|\mu|]$ and $\alpha \in [0..\sigma]$, the lowest α-ancestor of $\mu[i]$ (or a `NULL` value if such an ancestor does not exist). As a macro-tree contains at most $\lg^{4\epsilon} n$ nodes, $|\mu|$ can be encoded in $O(\lg\lg n)$ bits, T_μ can be encoded in $O(|\mu|) = O(\lg^{4\epsilon} n)$ bits, and I_μ can be encoded in $O(|\mu| \lg^{\epsilon} n) = O(\lg^{5\epsilon} n)$ bits. Thus any combination of $\{|\mu|, T_\mu\}$ and I_μ can be encoded in $O(\lg^{5\epsilon} n) = o(\lg n)$ bits. The lookup table G_1 contains $O(n^\delta)$ entries for some $0 < \delta < 1$, and thus requires only $o(n)$ bits of additional space.

Let x be a node in F and let μ denote the micro-tree that contains x. We now compute $\texttt{parent}_\alpha(x)$. The case in which x has an α-ancestor in μ is simple to handle: we only need to traverse the path from x to the root of μ, which uses $O(s) = O(\lg n/\lg\lg n)$ time.

Otherwise, by following the pointer (u_μ, p_μ), we can find the tier-1 macro-tree, μ_1, that contains x, and the node, x_1, in μ_1 that corresponds to μ. By accessing the lookup table G_1, we can determine in $O(1)$ time whether x_1 has an α-ancestor in μ_1, i.e., whether x has an α-ancestor in $C(\mu_1)$. Repeating this procedure, we can determine the macro-tree at the lowest level that contains both x and an α-ancestor of x. Let μ_k denote this macro-tree, for which k is the tier. During the process of locating μ_k, we are also able to determine the node, x_k, in μ_k that corresponds to a cluster containing x, and the node, y_k, that is the lowest α-ancestor of x_k in μ_k. Note that $y_k \neq x_k$, and there is no α-node between x and $C(y_k)$. Thus the α-parent of x is the lowest α-ancestor of x in $C(y_k)$, which can be found as described below.

By accessing D_{μ_k}, we can determine the tier-$(k-1)$ macro-tree ν_{k-1} that corresponds to y_k, and by accessing $b_{\nu_{k-1}}$, we can further determine the lower boundary z_{k-1} of ν_{k-1}. Then, the lowest node y_{k-1} in ν_{k-1} whose corresponding cluster contains an α-ancestor of x can be found in $O(1)$ time as follows. By accessing $I_{\nu_{k-1}}$, we can determine whether y_{k-1} is equal to z_{k-1}. If this is not the case, we use the lookup table G_1 to determine the lowest α-ancestor of z_{k-1} in ν_{k-1}, which is exactly y_{k-1}.

Repeating this procedure, we can determine the tier-1 macro-tree ν_1 whose corresponding cluster contains the lowest α-ancestor of x in $C(y_k)$, and the lowest node y_1 in ν_1 whose corresponding cluster contains an α-ancestor of x. The answer to $\texttt{parent}_\alpha(x)$ can be thus determined by traversing the micro-tree that corresponds to y_1.

Our algorithm requires $O(\lg n/\lg\lg n)$ time, since it accesses $O(\lg n/\lg\lg n)$ macro-trees and uses $O(1)$ time on each of them. □

Lemma 6.2.4. *With an $o(n)$-bit universal lookup table, the data structures constructed in this section can support* `LCA` *in $O(\lg n/\lg\lg n)$ time.*

Proof. We construct a lookup table G_2 that stores, for any unlabeled binary tree, μ, of size at most s and $i, j \in [1..|\mu|]$, the lowest common ancestor of its i-th and j-th nodes in preorder. As a binary tree of size at most s can be encoded in $O(s) = O(\lg n/\lg\lg n)$ bits, the lookup table G_2 contains $2^{O(s)} \times s \times s = O(n^\delta)$ entries for some $0 < \delta < 1$. Thus G_2

occupies $o(n)$ bits of additional space and can be computed in $o(n)$ time.

Let x and y be two nodes in the same binary tree of F and we now compute $\texttt{LCA}(x, y)$. The case in which x and y are in the same micro-tree is simple. Otherwise, by accessing the pointers (u_μ, p_μ), we can find the macro-tree at the lowest level that contains both x and y. Let μ_k denote this macro-tree, for which k is the tier. We also denote by x_k/y_k the node in μ_k whose corresponding cluster contains x/y.

Using the lookup table G_2, we can determine in $O(1)$ time the lowest common ancestor, z_k, of x_k and y_k in μ_k. Thus we know that the lowest common ancestor of x and y must be contained in the cluster $C(z_k)$. Let μ_{k-1} be the tier-$(k-1)$ macro-tree that corresponds to z_k. We then compute the node, x_{k-1}/y_{k-1}, in μ_{k-1} that is the closest node to x/y as follows. If x_k/y_k is equal to z_k, then x_{k-1}/y_{k-1} is the node in μ_{k-1} whose corresponding cluster contains x/y. Otherwise, x_{k-1}/y_{k-1} is the lower boundary of μ_{k-1}. Again, using the lookup table G_2, we can determine the lowest common ancestor, z_{k-1}, of x_{k-1} and y_{k-1} in μ_{k-1}. We repeat this procedure until μ_0, x_0 and y_0 are determined, where μ_0 is a micro-tree and x_0/y_0 is the closest node of μ_0 to x/y. The lowest common ancestor, z_0, of x_0 and y_0 is also that of x and y, which can be found using another table lookup to G_2.

This algorithm requires $O(\lg n/ \lg\lg n)$ time, as there are $O(\lg n/ \lg\lg n)$ tiers of macro-trees and our algorithm uses $O(1)$ time for each tier. □

Lemma 6.2.5. *With an $o(n)$-bit universal lookup table, the data structures constructed in this section can support* `summary` *in $O(\lg n/ \lg\lg n)$ time.*

Proof. We create a lookup table G_3 that stores, for any valid combination of $\{|\mu|, T_\mu\}$ and I_μ, and any $i, j \in [1..|\mu|]$, a bit vector of length $\sigma + 1$ in which the α-th bit is 1 iff $\mu[i]$ is an ancestor of $\mu[j]$ and there is an α-node on the path from the root of $C(\mu[i])$ to and excluding the root of $C(\mu[j])$. As in previous proofs, the lookup table G_3 occupies $o(n)$ bits of space and requires $o(n)$ preprocessing time.

Let x and y be two nodes in F and let $z = \texttt{LCA}(x, y)$. Without loss of generality, we only consider how to summarize the path from x to z. We decompose this path into a series of subpaths, which are based on the variables defined in the proof of Lemma 6.2.4:

- the subpath from x to the root of the micro-tree ν_0 containing x;
- for $i \in [1..k-1]$, the subpath from the root of $C(\nu_i)$ to and excluding the root of $C(\nu_{i-1})$, where ν_i is the tier-i macro-tree whose corresponding cluster contains x;
- the subpath from the root of $C(x'_k)$ to and excluding the root of $C(x_k)$, where x'_k is the child of z_k in μ_k that is an ancestor of x_k;

- for $i \in [1..k-1]$, the subpath from the root of $C(x'_i)$ to and excluding the root of $C(x'_{i+1})$, where x'_i is the child of z_i in μ_i that is an ancestor of x_i;
- the subpath from x'_0 to z.

The first and the last subpaths, which each are contained in a single micro-tree, can be summarized in $O(s) = O(\lg n/\lg\lg n)$ time. The others can each be summarized in $O(1)$ time by accessing the lookup table G_3 and I_μ. The path can be decomposed using pointers (u_μ, p_μ) and D_μ. With bitwise OR operations, these summaries can be combined using $O(\lg n/\lg\lg n)$ time. Thus we can support `summary` in $O(\lg n/\lg\lg n)$ time and $o(n)$ bits of additional space. □

Lemma 6.2.6. *With an $o(n)$-bit universal lookup table, the data structures constructed in this section can support* `pre_succ`$_\alpha$ *and* `post_pred`$_\alpha$ *in $O(\lg n/\lg\lg n)$ time.*

Proof. We only show how to support `pre_succ`$_\alpha$; the support for `post_pred`$_\alpha$ is similar. We construct a lookup table G_4 that stores, for any valid combination of $\{|\mu|, T_\mu\}$, b_μ and J_μ, and any $i \in [1..|\mu|]$, $j \in \{1,2\}$, a pair (i', j'), where the j'-th preorder segment of $C(\mu[i'])$ contains the first α-node in $C(\mu)$ that succeeds the j-th preorder segment of $C(\mu[i])$. By analysis similar to that in the proof of Lemma 6.2.3, the lookup table G_4 occupies $o(n)$ bits of space.

To compute $z =$ `pre_succ`$_\alpha(x)$, we first find the α-successor y of x in the micro-tree containing x. This can be done by checking the label of each node in the micro-tree, which requires $O(s) = O(\lg n/\lg\lg n)$ time. Note that y can be different from z even if y exists, because z may be located in a micro-tree below the one containing x.

As shown in the proof of Lemma 6.2.3, for $\ell = 1, 2, 3, \ldots$, we determine the tier-ℓ macro-tree μ_ℓ whose corresponding cluster contains x. By accessing b_{μ_ℓ}, we can further determine the node x_ℓ in μ_ℓ and $j_\ell \in \{1,2\}$ so that the j_ℓ-th preorder segment of $C(x_\ell)$ contains x. This requires only $O(1)$ time per marked level. Using the lookup table G_4, for $\ell = 1, 2, 3, \ldots$, we can further find the node y_ℓ of μ_ℓ and $j'_\ell \in \{1,2\}$, so that the j'_ℓ-th preorder segment of $C(y_\ell)$ contains the first α-node in $C(\mu_\ell)$ that succeeds the j_ℓ-th preorder segment of $C(x_\ell)$. The j'_ℓ-th preorder segment of $C(y_\ell)$ is referred to as the tier-ℓ *candidate segment*. In addition, the node y is referred to as the tier-0 candidate segment.

If there does not exist any candidate segment, then we conclude that the targeted node z does not exist, either. Otherwise, we find the first candidate segment in preorder. Let k denote the tier of the candidate segment that precedes all other candidate segments we have checked. The initial value k_0 of k is the lowest tier that has a candidate segment. This candidate segment is the j''_{k_0}-th preorder segment of $C(z_{k_0})$, where (z_{k_0}, j''_{k_0}) is (y_{k_0}, j'_{k_0}).

For $\ell = k_0+1, k_0+2, k_0+3, \ldots$, we find the node z_ℓ in μ_ℓ and $j''_\ell \in \{1,2\}$ so that the j''_ℓ-th preorder segment of $C(z_\ell)$ contains the tier-k candidate segment, where (z_ℓ, j''_ℓ) can be computed from $(z_{\ell-1}, j''_{\ell-1})$ in $O(1)$ time by following the pointer $(u_{\mu_{\ell-1}}, p_{\mu_{\ell-1}})$. If the tier-$\ell$ candidate segment exists and precedes the j''_ℓ-th preorder segment of $C(z_\ell)$, then k is updated to be ℓ and (z_ℓ, j''_ℓ) is reset to be (y_ℓ, j'_ℓ). As before, this procedure requires $O(1)$ time per marked level.

After determining the tier-k candidate segment, the remaining task is to find the first α-node in this segment, which is z. If $k = 0$, then we return $z = y$ as the answer. Otherwise, letting ν_{k-1} be the macro-tree that corresponds to y_k, we can find the node z'_{k-1} of ν_{k-1} and $\widetilde{j}_{k-1} \in \{1,2\}$ so that the $\widetilde{j}_{k-1}$-st preorder segment of $C(z'_{k-1})$ is a prefix of the tier-k candidate segment. By accessing G_4 with the information of ν_{k-1}, z'_{k-1} and $\widetilde{j}_{k-1}$, we can find the node in ν_{k-1} whose corresponding cluster contains z, and the preorder segment z belongs to. Repeating this procedure, we will finally end up with z.

There are $O(\lg n/\lg\lg n)$ tiers of macro-trees and a constant number of operations are performed for each tier. Thus the algorithm uses $O(\lg n/\lg\lg n)$ time and $o(n)$ additional bits of space. □

Lemma 6.2.7. *With $o(n)$-bit universal lookup tables, the data structures constructed in this section support* `modify` *in $O(\lg n/\lg\lg n)$ time, and* `link` *and* `cut` *in $O(\lg^{1+4\epsilon} n)$ time.*

Proof. We construct another lookup table G_5 that stores, for any valid combination of $\{|\mu|, T_\mu\}$, b_μ, S_μ, I_μ and J_μ, the following information:

- the number of the preorder segments of μ;
- a bit vector ρ of length $\sigma+1$ in which the α-th bit is 1 iff $C(\mu)$ contains an α-node on the path from its lower boundary to its root (if μ has no lower boundary, then 0-bits are stored in ρ);
- two bit vectors $\varrho[1..2]$ of length $\sigma+1$ in which the α-th bit of $\varrho[j]$ is 1 iff the j-th preorder segment of $C(\mu)$ contains an α-node (if μ has fewer than j preorder segments, then 0-bits are stored in $\varrho[j]$).

In other words, when μ corresponds to the i-th node of the macro-tree μ' one level above, $S_{\mu'}[i]$, $I_{\mu'}[i]$ and $J_{\mu'}[i]$ can be found in G_5 with the information of μ. By analysis similar to that in the proof of Lemma 6.2.3, the lookup table G_5 occupies $o(n)$ bits of space.

The operation `modify` can be supported as follows. After changing the label of node x, we first update the micro-tree μ and the tier-1 macro-tree μ_1 whose corresponding clusters contain x. It requires $O(1)$ time for μ and $O(\lg n/\lg\lg n)$ time for μ_1, as the entries of S_{μ_1}, I_{μ_1} and J_{μ_1} that correspond to μ can be updated by traversing μ. Then, for $\ell = 2, 3, \ldots$, we determine the tier-ℓ macro-tree whose corresponding cluster contains x, access table G_5 with the information of $\mu_{\ell-1}$, and update the entries of S_{μ_ℓ}, I_{μ_ℓ} and J_{μ_ℓ} that correspond to $\mu_{\ell-1}$. This procedure requires $O(1)$ time per marked level, and the overall update requires $O(\lg n/\lg\lg n)$ time.

To support `link` and `cut`, we first use Lemma 2.5.6 to update the directed topology tree in $O(\lg n)$ time. Then, by the same lemma, we observer that at each marked level of the directed topology tree, only a constant number of macro-trees have been modified. For each modified macro-tree that has not been deleted, we traverse down the directed topology tree to the next marked level below it, to construct the tree structure of the macro-tree in $O(\lg^{4\epsilon} n)$ time, and rebuild the auxiliary structures of this macro-tree in $O(\lg^{4\epsilon} n)$ time. The rebuilding of most of these structures is trivial; for structures such as I_μ and J_μ, we make use of G_5 again to fill the entries one by one. As there are $O(\lg n/\lg\lg n)$ marked levels, the entire process requires $O(\lg^{1+4\epsilon} n/\lg\lg n)$ time. □

6.2.2 Navigation between Levels of W

Now we describe how to navigate between levels of the weight-balanced B-tree W, which was defined at the beginning of Section 6.2. As discussed previously, we use Lemma 6.2.1 to encode each F_ℓ for $\ell > 0$. For each node at the ℓ-th level of W, we store a pointer to the root of its corresponding directed topology tree in F_ℓ. Each tree node in F_ℓ can be uniquely identified by a pointer to the micro-tree containing the node and its preorder rank in the micro-tree. We call the pair of pointer and preorder rank the *local id* of this node in F_ℓ.

Since each node, x, of T appears once in F_ℓ as an original node for each $\ell \in [0..t]$, x has one local id at each level of W. In our algorithm for path reporting, given the local id of x in F_ℓ, we need to find its local id in $F_{\ell-1}$ and $F_{\ell+1}$. It would require superlinear space to store the mappings for each node of each F_ℓ explicitly. Thus, our overall strategy is to precompute, for only a subset of nodes of T, their local ids in $F_{\ell-1}$ and $F_{\ell+1}$. Then, we design an algorithm to compute local ids of other nodes, by making use of the fact that both tree extraction and our way of transforming each $T(v)$ to $\mathcal{B}(v)$ preserve relative preorder among nodes of T. Our approach originates from the idea of dynamic fractional cascading [17, 81], and we further generalize the technique to data sets that are each represented as a forest of ordinal trees.

We now show how to construct auxiliary data structures for each F_ℓ to facilitate the navigation between forests constructed for subsequent levels of W. In F_ℓ, the clusters at the first marked level of the directed topology trees are said to be *mini-trees*. By our discussions in Section 6.2.1, each mini-tree then contains at most $\lg^{4\epsilon} n$ micro-trees, and $O(\lg^{1+4\epsilon} n/\lg\lg n)$ tree nodes. There is a one-to-one correspondence between mini-trees and tier-1 macro-trees, but they are conceptually different: nodes of mini-trees each are a node of F_ℓ, while nodes of tier-1 macro-trees each represent a micro-tree. Thus we continue to store information about each micro-tree or each of its preorder segments in a tier-1 macro-tree, while creating memory blocks for each mini-tree to store the subset of pointers between nodes at different levels of W which are to be described later. We also store a bidirectional pointer between a tier-1 macro-tree and its corresponding mini-tree, and thus we do not distinguish the pointers to a tier-1 macro-tree from a pointer to its corresponding mini-tree, even though technically they store different memory locations. We say a micro-tree is the i-th micro-tree of a mini-tree (or its corresponding tier-1 macro-tree), if this micro-tree is represented by the i-th node of the tier-1 macro-tree corresponding to this mini-tree.

A node in F_ℓ can also be uniquely identified by a pointer to the mini-tree containing the node and its preorder rank in the mini-tree. As local ids, we call the pair of pointer and preorder rank the *regional id* of this node in F_ℓ. The following lemma shows how to convert between regional ids and local ids in $O(1)$ time and sublinear words of additional space.

Lemma 6.2.8. *With $O(n\lg\lg n)$ bits of additional space, one can convert between the local id and the regional id of a node x in F_ℓ using $O(1)$ time.*

Proof. To support the conversion, we construct, for each tier-1 macro-tree μ, an additional two-dimensional array $N_\mu[1..|\mu|][1..2]$, in which $N_\mu[i][j]$ stores the number of nodes of F_ℓ in the j-th preorder segment of the i-th micro-tree of μ, and 0 if this micro-tree has fewer than j preorder segments. Since each preorder segment of a micro tree has $O(\lg n/\lg\lg n)$ nodes, each entry of N_μ can be encoded in $O(\lg\lg n)$ bits. Thus, N_μ uses $O(|\mu|\lg\lg n)$ bits, and all tier-1 macro trees add up to $O(n\lg\lg n)$ bits, which does not change the space bound of Lemma 6.2.1 asymptotically.

We only show how to convert local ids to regional ids; the other direction can be proved similarly. From x's local id, we have a pointer that points to the micro-tree ν containing x, and (u_ν, p_ν) that locates the mini-tree containing ν and x. Thus it suffices to compute x's preorder rank k in this mini-tree from x's preorder rank in ν. We first determine the preorder segment τ of ν that contains x by comparing x's preorder rank with the preorder rank of the lower boundary node stored in b_ν. We compute k as the sum of the following two values: k_1, the number of nodes within τ that precede x in preorder (including x); and k_2, the number of nodes in x's mini-tree that strictly precede the first node of τ in preorder. The value of k_1 is x's preorder rank in ν if τ is the first preorder segment of

ν, or 1 plus the difference between x's preorder rank and b_ν otherwise. To compute k_2 in $O(1)$ time, we construct a universal table G_6 that stores, for each possible tier-1 macro-tree μ, each $i \in [1..\lfloor \lg^{4\epsilon} n \rfloor]$ and each $j \in \{1, 2\}$, the number of nodes of T in the mini-tree corresponding to μ that strictly precedes, in preorder, the first node in the j-th preorder segment of the i-th micro-tree in μ. Here macro-trees with different values in $|\mu|$, T_μ, S_μ, or N_μ are considered different. By analysis similar to that in the proof of Lemma 6.2.3, G_6 occupies $o(n)$ bits. □

Because of the above lemma, we consider local ids and regional ids both as valid identifiers of nodes in F_ℓ in the rest of the chapter, and do not explicitly perform constant-time conversions between them.

Furthermore, we consider the support for $\texttt{parent}_\alpha(\mu, x)$ within any given mini-tree μ, i.e., given a node x, we are interested in finding its α-parent in the same mini-tree if it exists. By simplifying the proof of Lemma 6.2.3, we can easily show that $\texttt{parent}_\alpha$ can be supported in constant time in a given mini-tree, as we need only to use auxiliary structures for two marked levels of directed topology trees instead of all the marked levels. We further consider the following two operations over a mini-tree:

- $\texttt{pre\_rank}_\alpha(\mu, x)$: computes the number of α-nodes preceding x in preorder of μ (including x itself if it is labeled α);
- $\texttt{pre\_select}_\alpha(\mu, i)$: locates the i-th α-node in preorder of μ.

In the above definition, we allow α to be $\bar{0}$, which matches any label that is not 0. By constructing data structures similar to N_μ for each label to record the number of α-nodes in each preorder segment of a micro-tree, it is trivial to modify the proof of Lemma 6.2.8 to support $\texttt{pre\_rank}_\alpha$ and $\texttt{pre\_select}_\alpha$ within each mini-tree in constant time. Thus, we have the following lemma:

Lemma 6.2.9. *With $o(n)$ bits of additional space, $\texttt{parent}_\alpha$, $\texttt{pre\_rank}_\alpha$ and $\texttt{pre\_select}_\alpha$ can be supported in $O(1)$ time over each mini-tree in F_ℓ.*

We next define a set of pointers between nodes of mini-trees at different levels of W, for which a pointer can be regarded as a directed edge from its source node to its destination node. These *inter-level pointers* are defined for each mini-tree μ in any F_ℓ. Let v be the node of W such that $\mathcal{B}(v)$ contains μ. If $\ell < t$, then for each preorder segment of μ, we create an *up pointer* for the first original node, x, of this segment in preorder. This pointer points from x to the original node in $F_{\ell+1}$ that corresponds to the same node of T. Such a pointer essentially links v to its parent. Next, if $\ell > 1$, for each preorder segment of μ and for each label $\alpha \in [1..4d]$, if node y is the first node in this segment in preorder that is labeled α, we store a *down pointer* from y to the original node in $F_{\ell-1}$ that corresponds to the same node of T that y represents. This pointer essentially links v to the α-th child of v. No pointers are created for nodes labeled 0, as they are dummy nodes. So far we have created at most $2(4d + 1) = O(\lg^{\epsilon} n)$ inter-level pointers for each mini-tree, as a mini-tree has up to two preorder segments. Finally, we create a back pointer for each up or down pointer, doubling the total number of inter-level pointers created over all the levels of W.

Remember that the mini-tree μ is in $\mathcal{B}(v)$, which is part of F_{ℓ}. To store inter-level pointers physically, we maintain all the pointers that leave from mini-tree μ in a structure P_{μ}, including up and down pointers created for nodes in μ, and back pointers for some of the up and down pointers created for mini-trees at adjacent levels of W. We further categorize these pointers into at most $4d + 1$ types: A type-0 pointer arrives at a mini-tree in $F_{\ell+1}$, i.e., goes to the level above, and a type-α pointer for $\alpha > 0$ arrives at a mini-tree in $B(v_{\alpha})$, where v_{α} is the α-th child of v. Note that it is possible that an up or down pointer of μ and a back pointer from an adjacent level stored in P_{μ} have the same source, which is a node in μ, and destination, which is a node in $F_{\ell-1}$ or $F_{\ell+1}$. In this case, the back pointer is not stored separately in P_{μ}, and hence each inter-level pointer in P_{μ} can be uniquely identified by its type and the preorder rank of its source node in μ. We also store the preorder rank of the first node of each of the (at most two) preorder segments in μ. The following lemma summarizes how P_{μ} is represented.

Lemma 6.2.10. *P_{μ} can be represented in $O(|P_{\mu}| \lg n)$ bits, where $|P_{\mu}|$ is the current number of inter-level pointers that leave from μ, to support the following operations in $O(1)$ time with $o(n)$-bit universal lookup tables. Here we assume that x is a node that is identified by its preorder rank in μ, and $\alpha \in [0..4d]$ is a type of inter-level pointers.*

- `pointer_pred`(μ, x, α)*: returns the closest preceding node (this could be x itself) in the preorder segment of μ containing x that has a type-α inter-level pointer, as well as the local id of the destination of this pointer;*

- `pointer_insert`(μ, x, α)*: inserts a type-α pointer that leaves from x; and*

- `pointer_delete`(μ, x, α)*: delete the type-α pointer that leaves from x.*

Proof. We represent P_{μ} using the approach of Navarro and Nekrich [81, Section 6.3] with trivial modifications, and we include a proof here for completeness.

First we observe that P_{μ} is polylogarithmic in n. This is because each pointer in P_{μ} is uniquely identified by its type and its source node in μ, and there are $O(\lg^{\epsilon} n)$ types and $O(\lg^{1+4\epsilon} n)$ nodes in μ. We then use a B-tree W_μ with fanout $f_\mu = \Theta(\lg^\epsilon n)$ to store the pointers in P_μ, which has only a constant number of levels. Each inter-level pointer in P_μ is inserted into W_μ using the preorder rank of its source as its key. Each leaf of W_μ stores a constant number of inter-level pointers in P_μ, and the information of each pointer, including its type, the preorder rank of its source in μ and the local id of its destination node, is encoded in $O(\lg n)$ bits. An internal node of W_μ is said to cover the range $[a..b]$ of preorder ranks in μ if all keys between a and b are stored in the leaf descendants of this internal node. Each internal node then stores the size of the range it covers, and, for $0 \leq \alpha \leq 4d$, the number of type-α inter-level pointers stored in the subtree of each child. Thus the information for each internal node can be encoded in $f_\mu \times (4d+1) \times O(\lg\lg n) = O(\lg^{2\epsilon} n \cdot \lg\lg n) = O(\lg^{\epsilon'} n)$ bits for a constant $\epsilon' \in (2\epsilon, 1)$.

Given a node x in μ, we can use the following procedure to look for the the closest node, y, preceding x that has a type-α inter-level pointer. Afterwards, we further compare y with the preorder number of the first node in the preorder segment of μ containing x, to decide whether y is in the same preorder segment. We first descend down the B-tree, which has a constant number of levels to look for the leaf that covers the preorder rank of x. This leaf can be located in constant time if we construct a universal table of $o(n)$ bits and use the $O(\lg^{\epsilon'} n)$-bit information encoded for each internal node to query it. The leaf stores a constant number of inter-level pointers, so in $O(1)$ time, we can either find y and terminate, or find out that y is not stored in the same leaf. In the latter case, we go up the tree, and for each ascendant node we visit, we find out whether it has a left sibling whose subtree stores a type-α pointer using table lookup. Once we find such a left sibling, we look for the last type-α pointer in preorder stored in its subtree by traversing down the tree again. It is clear that we can find y or determine that it does not exist in $O(1)$ time. Thus `pointer_pred` is supported.

For `pointer_insert` and `pointer_delete`, we first locate the appropriate leaf and performing insertion or deletion at the leaf, update the information stored at each of its ancestors in W_μ, and we may merge or split nodes of W_μ if necessary. Since a leaf stores a constant number of inter-level pointers and each internal node can be updated, merged with a sibling, or split in constant time using appropriate universal tables, this also requires $O(1)$ time. □

We now prove the following lemma, which enables us to traverse between different levels of W.

Lemma 6.2.11. *Give the local id of an original node x in F_ℓ, the local id of the original node in $F_{\ell+1}$ (if $\ell < t$) or $F_{\ell-1}$ (if $\ell > 1$) that represents the same node of T can be computed in $O(1)$ time.*

Proof. We first show how to locate the node, y, in $F_{\ell+1}$ that represents the same node of T. Let μ be the mini-tree in F_ℓ that contains x and let τ_0 be the preorder segment containing x in μ. We start by using Lemma 6.2.10 to find $x' = \texttt{pointer\_pred}(\mu, x, 0)$, where x' could be x itself. The destination node, y', of this pointer is also retrieved during the same process, which is a node in $F_{\ell+1}$. Node x' always exists because the first original node of each preorder segment in a mini-tree has an up pointer.

If x' happens to be x itself, then y' is y and the answer is found. If not, we prove that y and y' are in the same preorder segment of a mini-tree in $F_{\ell+1}$ (Figure 6.2 illustrates the objects defined in this proof). Assume to the contrary that y and y' are in two different preorder segments, τ_1 and τ_2, at the mini-tree level, respectively. Note that τ_1 and τ_2 may not be in the same mini-tree, but they must be in the same binary tree, $\mathcal{B}(v)$, in $F_{\ell+1}$. As both tree extraction and our transformation of ordinal trees to binary trees preserve preorder relationship among original nodes, we claim that an original node precedes another one in preorder of $\mathcal{B}(v)$ iff their corresponding nodes in T appear in the same order in preorder. The same reasoning applies to the binary tree, $\mathcal{B}(u)$, that contains x and x'. Furthermore, the original nodes in $\mathcal{B}(u)$ represent a subset of the nodes of T that are represented by the original nodes in $\mathcal{B}(v)$, as u is a child of v in W. Therefore, the relative preorder relationship among original nodes in $\mathcal{B}(u)$ is consistent to that of the corresponding original nodes in $\mathcal{B}(v)$. Since x' strictly precedes x in preorder, y' also strictly precedes y. This implies that the nodes of τ_1 precede the nodes of τ_2 in preorder of $\mathcal{B}(v)$. There cannot be a down pointer from y, for otherwise, there would be a up pointer from x and the presumption $x' \neq x$ would be contradicted. Thus there has to be another node y'' in τ_2 that strictly precedes y in preorder and has a down pointer to some node x'' in $\mathcal{B}(u)$. As y'' is strictly between y and y' in preorder of $\mathcal{B}(v)$, x'' must be strictly between x' and x in preorder of $\mathcal{B}(u)$ and reside in the preorder segment τ_0. As there is a back

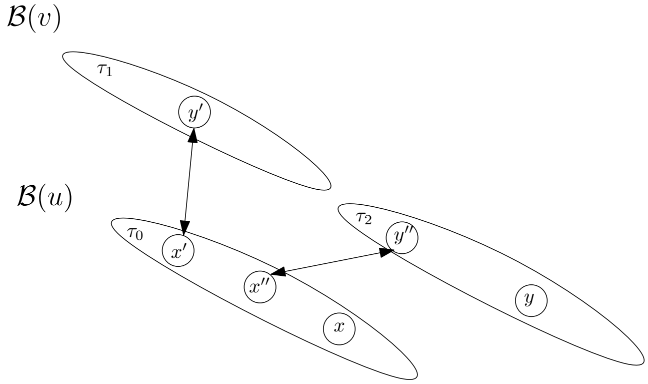


Figure 6.2: A figure for the proof of Lemma 6.2.11.

pointer from x'' to y'', x' would not be the closest predecessor of x in τ_0 that has a type-0 inter-level pointer.

Suppose that u is the α-th node of v. It then follows that the number of α-nodes of $\mathcal{B}(v)$ that are between y' and y in preorder is equal to the number, k, of original nodes between x' and x in $\mathcal{B}(u)$. As the number of original and dummy nodes between x' and x in $\mathcal{B}(u)$ is equal to the difference between the preorder ranks of x' and x, it suffices to compute the number of dummy nodes between them, which can be expressed as $\texttt{pre\_rank}_0(x) - \texttt{pre\_rank}_0(x')$ in $\mathcal{B}(u)$. By Lemma 6.2.9, this can be computed in $O(1)$ time since x and x' reside in the same preorder segment at the mini-tree level. Then, the preorder rank of y can be computed as $\texttt{pre\_select}_\alpha(\texttt{pre\_rank}_\alpha(y') + k)$ in $\mathcal{B}(v)$, which again requires constant time. This gives us the local id of y', and the entire process uses $O(1)$ time.

We then briefly describe how to locate the node, z, in $F_{\ell-1}$ that represents the same node of T as x does. The process is similar to that used to locate y. Recall that μ is the mini-tree in F_ℓ that contains x and τ_0 is the preorder segment containing x in μ. Let β be the label assigned to x. We first find the closest node x^* in τ_0 that precedes x in preorder and has a type-β inter-level pointer. Let z^* be the node in $F_{\ell-1}$ that is the destination of the inter-level pointer that leaves from x^*. Using similar reasoning, we can prove that z^* and z are in the same preorder segment at the mini-tree level, and make use of Lemma 6.2.9 again to locate z in constant time. □

6.2.3 Supporting Path Reporting

We now describe how to support path reporting queries and updates.

Lemma 6.2.12. *The data structures described in this section answer a path reporting query in $O((\lg n/\lg\lg n)^2 + occ \cdot \lg n/\lg\lg n))$ time, where occ is the output size.*

Proof. Let x and y be the endpoints that define the query path, and let $[p, q]$ be the query range of weights. We perform a top-down traversal in W to locate up to two leaves that represent ranges containing p and q, respectively. During this traversal, we visit at most two nodes at each level of W. We further determine each node to visit at the next level using binary search in $O(\lg d) = O(\lg\lg n)$ time, since each node has at most $4d$ children. As there are $O(\lg n/\lg\lg n)$ levels in W, it requires $O(\lg n)$ time to determine the nodes of W to visit.

For each node, v, of W visited during the above top-down traversal, we also determine the original nodes x_v and y_v in $\mathcal{B}(v)$ that represent the lowest ancestors of x and y in T

that are represented by nodes in $\mathcal{B}(v)$, respectively. Note that each node is considered to be its own ancestor. These nodes are located during the top-down traversal as follows. Initially, at the root node r of W, x_r and y_r are the nodes in $\mathcal{B}(r)$ that represent x and y, respectively. Then, given that v is the α-th child of u, we show how to compute x_v and y_v from x_u and y_u, respectively. We only show how to determine x_v; the node y_v can be computed in a similar manner. If x_u is labeled with α, then x_v and x_u represent the same node of T, and we make use of Lemma 6.2.11 to locate x_v in constant time. Otherwise, we first locate x_u's α-parent, $x' = \texttt{parent}_\alpha(x_u)$, in $O(\lg n/ \lg\lg n)$ time using Lemma 6.2.1, and then compute x_v as the node in $\mathcal{B}(v)$ that represents the same node of T as x' does in $O(1)$ time using Lemma 6.2.11. The total time required to locate all these nodes is thus $O((\lg n/ \lg\lg n)^2)$.

For each node, v, of W visited during the traversal, if the range of at least one of v's children is contained entirely in $[p, q]$, then we compute $z_v = \texttt{LCA}(x_v, y_v)$ in $\mathcal{B}(v)$. We also perform a path summary query $\texttt{summary}(x_v, y_v)$ in $\mathcal{B}(v)$, and let V be the bit vector returned by the query. Suppose that the children of v whose ranges are contained in $[p, q]$ are labeled $\beta_1, \beta_1 + 1, \ldots, \beta_2$. As V has $O(\lg^\epsilon n)$ bits, we can retrieve the position of each 1-bit in $V[\beta_1..\beta_2]$ in $O(1)$ time using an $o(n)$-bit universal lookup table. Then for each $\gamma \in [\beta_1..\beta_2]$ such that $V[\gamma] = 1$, we claim that there are γ-nodes along the path from x_v to y_v in $\mathcal{B}(v)$, and these nodes represent a subset of the nodes of T to be reported. Each γ-node from x_v to z_v (including x_v and z_v) can be located using $\texttt{parent}_\gamma$ in $\mathcal{B}(v)$ repeatedly. All these nodes are found when we reach a node whose preorder rank in $\mathcal{B}(v)$ is no greater than that of z_v. For each of these nodes, we apply Lemma 6.2.11 repeatedly to find its local id at the level above until we reach the root level of W, which immediately gives us a node of T to report. The γ-nodes from y_v to z_v (including x_v but excluding z_v) can be handled using the same approach.

We observe that a constant number of LCA and summary operations are performed at each level of W, which require $O((\lg n/ \lg\lg n)^2)$ time in total. Then, each node in the output requires only $O(\lg n/ \lg\lg n)$ time to report: if we always charge each $\texttt{parent}_\alpha$ operation to the last node reported before this operation is performed, then each node is charged a constant number of times, and the process described above, which finds the node of T given its local id in $\mathcal{B}(v)$, requires $O(\lg n/ \lg\lg n)$ time. This completes the proof. □

Lemma 6.2.13. *The data structures described in this section can support* node_insert, node_delete *and* modify_weight *in* $O(\lg^{2+4\epsilon} n)$ *amortized time.*

Proof. We only show how to support node_insert; the other update operations can be handled similarly. Note that update operations may eventually change the value of $\lceil \lg n \rceil$, which in turn affects the values of the branching factor d and the base cluster size. However, this can be handled by rebuilding our data structure using `node_insert`, as rebuilding is only required after a linear number of update operations and thus the cost can be amortized over these operations without changing our time bounds. Hence it suffices to consider the case in which $\lceil \lg n \rceil$ does not change after an insertion.

Suppose that in an insertion operation, we insert a new node h with weight w_h. The new node h is inserted as a child of x, and a set of consecutive children of x between and including child nodes y and z become the children of h after the insertion. Here we consider the general case in which y and z exist and are different nodes; degenerate cases can be handled similarly. In the first step of our insertion algorithm, we insert the weight w_h into W by creating a new leaf for it. This may potentially cause the parent of this new leaf to split. We first consider the case in which a split will not happen, and then show how to handle node splits in W later.

We next perform a top-down traversal of W to fix the structures created for the forest F_ℓ at each level ℓ of W. In our description, when we say node h (or x, etc.) in F_ℓ, we are referring to the original node, either to be inserted to F_ℓ or already existing in F_ℓ, that corresponds to this node in T. At the top level, i.e., the t-th level, of W, the forest F_t contains one single binary tree. As we maintain bidirectional pointers between nodes in T and nodes in F_t, we can immediately locate the nodes x, y and z in F_t. Let $x_1, x_2, \ldots$ be the dummy nodes created for x in F_t, among which x_j and x_k are the dummy nodes

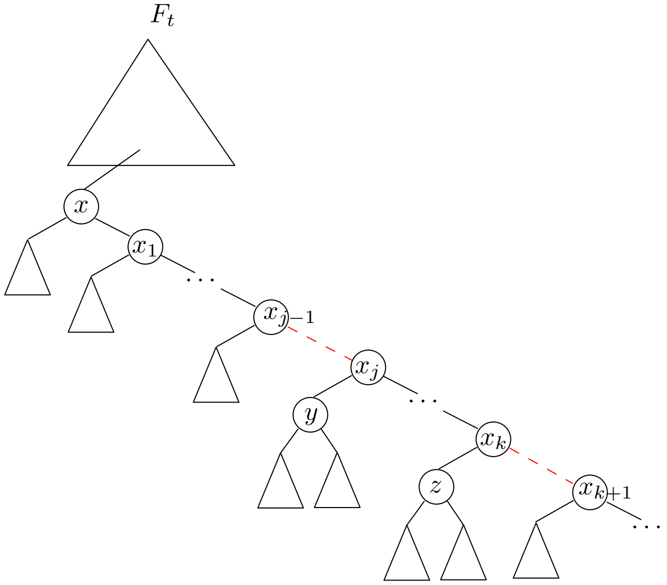


Figure 6.3: Forest F_t (containing a single tree) before `node_insert`. Dashed red lines represent edges to be removed by performing `cut`.

that are parents of y and z, respectively. We then perform a constant number of updates to F_t as follows. First we perform the `cut` operation twice to remove the edge between x_{j-1} and x_j, and the edge between x_k and x_{k+1}. This divide F_t into three trees, which is illustrated in Figure 6.3. We then create a tree on a new node x'_j which is a dummy node, and temporarily include this tree into F_t. Note that creating the directed topology tree and associated auxiliary data structures for a tree on a single node can be trivially done in constant time. We then replace the dummy node x_j by the node h being inserted. This can be done by first performing binary searches in the ranges of the children of the root, r, of W, so that we know the correct label, α, to assign to h. Thus we call `modify` to change the label of x'_j from 0 to α. We then perform `link` to add three edges so that x'_j becomes the right child of x_{j-1}, h becomes the left child of x'_j, and x_{k+1} becomes the right child of x'_j. See Figure 6.4 for the illustration. It is clear that all these operations require $O(\lg^{1+4\epsilon} n)$ time in total.

Let v be the α-th child of r. By the construction of W, h should be part of $\mathcal{B}(v)$ in

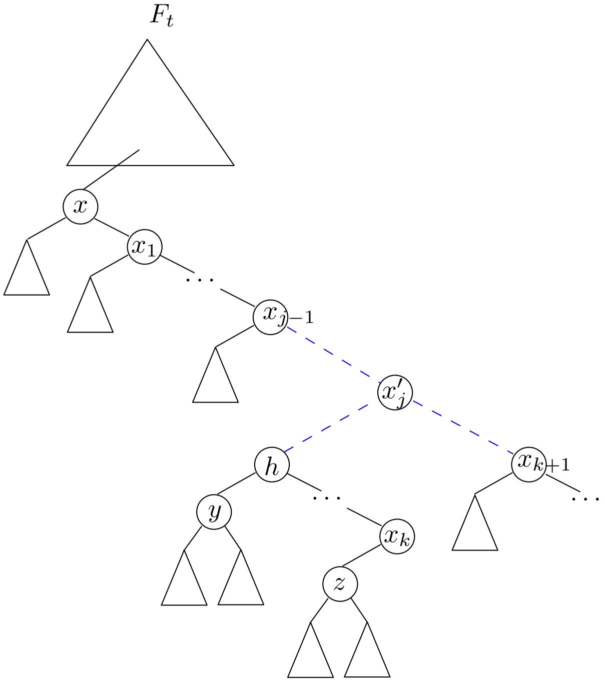


Figure 6.4: Forest F_t after `node_insert`. Dashed blue lines represent edges inserted by performing `link`.

F_{t-1}. To update F_{t-1}, we observe that it suffices to update $\mathcal{B}(v)$ without making changes to any other tree in F_{t-1}. Then, if x is also labeled α in $\mathcal{B}(r)$, we insert h as a child of the original node corresponding to x in $\mathcal{B}(v)$; otherwise, we insert h as a child of the original node of $\mathcal{B}(v)$ that corresponds to the node $x' = \texttt{parent}_{\alpha}(x)$ in $\mathcal{B}(r)$. If x' does not exist, then h is inserted as a child of the dummy root of $\mathcal{B}(v)$. We then observe that h will be inserted to $\mathcal{B}(v)$ as the new parent of the set of children of x, x' or the dummy root (depending on which of the above three cases applies) that are between and including the original nodes in $\mathcal{B}(v)$ that correspond to the nodes $\texttt{pre\_succ}_{\alpha}(y)$ and $\texttt{post\_pred}_{\alpha}(z)$ in $\mathcal{B}(r)$. Thus, in $O(\lg n/\lg\lg n)$ time, we have found where to insert h in F_{t-1}, and by the approach shown in the previous paragraph, we can use `link`, `cut` and `modify` to update $F_{\ell-1}$ in $O(\lg^{1+4\epsilon} n)$ time. This process can then be repeated at each successive level of W. Hence it requires $O(\lg^{2+4\epsilon} n/\lg\lg n)$ time to update all the F_{ℓ}'s.

When updating the F_{ℓ}'s, we also update inter-level pointers. As we only perform `cut`, `link` and `modify` a constant number of times over each F_{ℓ}, by Lemma 2.5.6, in each F_{ℓ} only a constant number of mini-trees will be modified. For each such mini-tree μ except those that are deleted, we reconstruct P_{μ} entirely. We first recompute all the up and down pointers leaving from μ, and the destinations of each of these pointers can be computed in constant time using the inter-level pointers maintained before the insertion. Then, for each P_{ν} that will be deallocated (either because the mini-tree ν is deleted or its content is changed so that a new P_{ν} is being built), we iterate through all the inter-level pointers stored in it. For each of these pointers, we determine whether it is a back pointer of an up or down pointer leaving from an adjacent level, which can be done using labeled `pre_rank` and preorder ranks of the first nodes of each preorder segments in the mini-tree in constant time per pointer. If a pointer fits into the above case, then we locate its source node after the update, and update the corresponding structure for inter-level pointers that leave from the mini-tree containing this source node. Here the source node can be located in constant time if, when mini-trees are being modified, we temporarily maintain pointers between the same node in a newly inserted mini-tree and a mini-tree to be deleted. We also follow the same back pointer to update the inter-level pointer leaving from a mini-tree at an adjacent level that is the reversal of this pointer.

From these discussion, we can see that the time spent on updating inter-level pointers for each level of W is proportional to the maximum number of inter-level pointers that leave a mini-tree. Since each node of a mini-tree can be the source of at most two of these pointers (one to the level above and the other to the level below), we claim that it requires $O(\lg^{1+4\epsilon} n)$ time to update the inter-level pointers for each level, and thus requiring $O(\lg^{2+4\epsilon} n/\lg\lg n)$ time again for all levels of W. One final detail is that the binary tree in F_1 that contains the newly inserted node should be completely rebuilt, as the insertion of this node may cause many nodes in this tree to change labels. This however can be done in $O(\lg^{\epsilon} n)$ time easily, as a binary tree in F_1 has up to $O(d) = O(\lg^{\epsilon} n)$ nodes.

So far we have shown that if no node in W has to split, `node_insert` can be performed in $O(\lg^{2+4\epsilon} n/\lg\lg n)$ time. We now amortize the cost of splitting nodes. Let v be a node at level ℓ of W that is to be split, and let v_1 and v_2 be the two nodes that v is split into. This requires us to split the binary tree $\mathcal{B}(v)$ into two trees $\mathcal{B}(v_1)$ and $\mathcal{B}(v_2)$. The two new trees themselves can be computed by first converting $\mathcal{B}(v)$ back to an ordinal tree containing its original nodes only, and then performing tree extraction to generate two ordinal trees which can in turn be transformed into $\mathcal{B}(v_1)$ and $\mathcal{B}(v_2)$. When updating F_ℓ, however, a tree can only be deleted by cutting out its nodes one-by-one using `cut` and deallocate them, and a new tree can only be created by allocating memory for its nodes one by one (recall that the directed topology tree for a tree on a single node can be created trivially in constant time) and then linking them together using `link`. This requires $O(|\mathcal{B}(v)| \lg^{1+4\epsilon} n)$ time. The affected inter-level pointers can be created or updated in $O(|\mathcal{B}(v)|)$ time, using an approach similar to the one discussed in the previous paragraphs. Thus the total cost is $O(|\mathcal{B}(v)| \lg^{1+4\epsilon} n)$.

This cost can be amortized using the following two properties of weight-balanced B-trees: First, a node at level ℓ has at most $2d^\ell$ leaf descendants. Therefore, $|\mathcal{B}(v)| \leq 2d^\ell$ and thus the total cost is $O(d^\ell \lg^{1+4\epsilon} n)$. Second, after the previous split of a node v at level ℓ, at least $d^\ell/2$ insertions have to be performed below v before v splits again. Therefore, we can amortize the $O(d^\ell \lg^{1+4\epsilon} n)$ total cost over $d^\ell/2$ insertions, which is $O(\lg^{1+4\epsilon} n)$ time per insertion.

Let u be the parent of v. The above split will replace v by two nodes v_1 and v_2. This may potentially affect the labels of a large number of nodes in $\mathcal{B}(u)$, and thus we have to perform `modify` on affected nodes of u, and update some inter-level pointers. This can cost $O(|\mathcal{B}(u)| \lg n/\lg\lg n)$ time, which is $O(d^{\ell+1} \lg n/\lg\lg n)$. When amortizing this cost over the $d^\ell/2$ insertions described in the previous paragraph, each insertion is charged $O(d \lg n/\lg\lg n) = O(\lg^{1+\epsilon} n/\lg\lg n)$ time. Finally, since an `node_insert` can potentially cause up to t nodes in W to split, all the way to the root level, the amortized time cost is $O(\lg^{2+4\epsilon} n/\lg\lg n)$ per insertion. Therefore, each `node_insert` can be performed in $O(\lg^{2+4\epsilon} n)$ amortized time. □

Lemma 6.2.14. *The data structures constructed in this section occupy $O(n)$ words of space.*

Proof. We first bound the total space required to store the inter-level pointers. As each P_μ structure uses $O(\lg n)$ bits or $O(1)$ words per pointer, the space cost in words is linear in the total number of inter-level pointers, which is in turn twice as many as the total number of up and down pointers. Let p_ℓ denote the number of up and down pointers created for F_ℓ. To bound the sum of p_ℓ's over all levels of W, it suffices to bound the total number of mini-trees, because each mini-tree is associated with $O(\lg^\epsilon n)$ up and down pointers. Let

f_ℓ be the number of binary trees in F_ℓ, i.e., the number of nodes at the ℓ-th level of W, and let $n_{\ell,i}$ denote the number of nodes in the i-th binary tree of F_ℓ. Note that $\sum_{i=1}^{f_\ell} n_{\ell,i} \leq 2n$, as the number of dummy nodes created in F_ℓ is at most n. By Lemma 2.5.5, the number of mini-trees in the i-th binary tree is

$$\begin{aligned} O(n_{\ell,i}/s+1)\cdot(5/6)^{4\epsilon\lg\lg n}+1 &< O(n_{\ell,i}/s+1)\cdot(1/2)^{\epsilon\lg\lg n}+1 \\ &\leq O(n_{\ell,i}\cdot\lg\lg n/\lg^{1+\epsilon} n+1). \end{aligned}$$

Then the total number of mini-trees is

$$\sum_{i=1}^{f_\ell} O(n_{\ell,i}\cdot\lg\lg n/\lg^{1+\epsilon} n+1) = O(n\lg\lg n/\lg^{1+\epsilon} n+f_\ell).$$

Therefore, the total number of up and down pointers created for F_ℓ, where $\ell > 1$, is $O(n\lg\lg n/\lg n + f_\ell)$. Note that no inter-level pointers are stored for F_0. As each node in W has between $d/4$ and $4d$ children, we have $f_\ell \leq \frac{n}{(d/4)^\ell}$. Recall that the total number of levels of W is defined to be $t+1 = O(\lg n/\lg\lg n)$. Therefore, the total number of inter-level pointers in our entire data structure is

$$\begin{aligned} O(\sum_{\ell=1}^{t}(n\lg\lg n/\lg n+f_\ell)) &\leq O(n)+O(\sum_{\ell=1}^{\infty}\frac{n}{(d/4)^\ell}) = O(n)+O(\frac{n}{d/4-1}) \\ &= O(n)+O(n/\lg^\epsilon n) = O(n). \end{aligned}$$

Hence inter-level pointers occupy $O(n)$ words of space in total.

Next we bound the total space used by auxiliary data structures constructed by micro-trees and macro-trees. Again, these structures are constructed for F_ℓ where $\ell \geq 1$. It is clear that the additional structures we constructed for each tier-1 macro-tree to prove Lemma 6.2.9 will not change the space bound in Lemma 6.2.1. The total space is thus $O(\sum_{\ell=1}^{t}(n\lg\lg n + f_\ell\lg n)) = O(n\lg n)$ bits, which is $O(n)$ words.

The weight-balanced B-tree W occupies $O(n)$ words of space. Finally, all the universal tables constructed in previous proofs use $o(n)$ bits of space in total. Therefore, the total space cost is $O(n)$ words. □

Combining Lemmas 6.2.12 to 6.2.14, and replacing ϵ with $\epsilon/4$, we have our main result:

Theorem 6.2.15. *Under the word RAM model with word size $w = \Omega(\lg n)$, an ordinal tree on n weighted nodes can be stored in $O(n)$ words of space, such that path reporting queries can be answered in $O((\lg n/\lg\lg n)^2 + occ\cdot\lg n/\lg\lg n))$ time, where occ is the output size, and* `modify_weight`, `node_insert` *and* `node_delete` *can be supported in $O(\lg^{2+\epsilon} n)$ amortized time for any constant $\epsilon \in (0,1)$.*

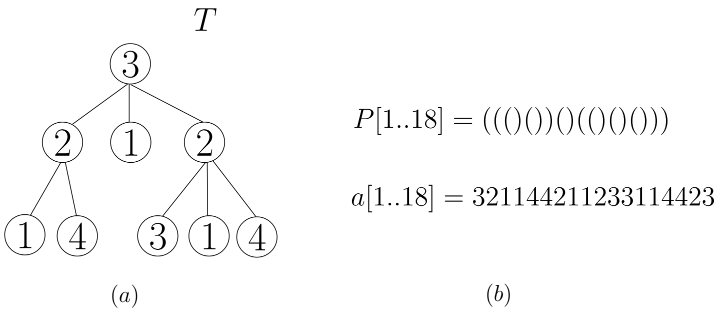
6.3 Dynamic Path Counting and Path Selection

In this section we describe our dynamic data structures for path counting and path selection. Our strategy is similar to He et al.'s approach for dynamic range selection [63]. We include full details for the sake of completeness.

Let T denote the input weighted tree and let n denote the size of T. As described in Section 2.5.1, the structure of T can be represented as a sequence, $P[1..2n]$, of balanced parentheses. We annotate each parenthesis $P[i]$ with a weight $a(i)$, which is equal to the weight of the node that corresponds to $P[i]$, and a score $e(i)$, which is equal to 1 if $P[i]$ is an opening parenthesis, or -1 if $P[i]$ is a closing parenthesis. See Figure 6.5 for an example. To support path counting and path selection queries, we define two types of queries over P:

- $\texttt{par\_range\_sum}(i, j, [p, q])$: given that $1 \le i \le j \le 2n$ and a range $[p, q]$, the *range sum* query returns the sum of scores over $P[i..j]$ whose weights are between p and q;
- $\texttt{par\_range\_sel}(i, j_1, j_2, k)$: given opening parentheses at indices i, j_1, and j_2 that satisfy $1 \le i \le j_1 \le j_2 < \texttt{findclose}(i)$ and an integer $k > 0$, the *two-range selection* query returns the minimal q that satisfies the following inequality

$$\texttt{par\_range\_sum}(i, j_1, (-\infty, q]) + \texttt{par\_range\_sum}(i, j_2, (-\infty, q]) \ge k.$$



$P[1..18] = ((()())()(()()()))$

$a[1..18] = 321144211233114423$

Figure 6.5: (a) An input tree T on 9 weighted nodes. (b) The sequence P of balanced parentheses and the sequence a of weights that correspond to T. The score sequence e can be derived from P easily.

In Lemmas 6.3.1 and 6.3.2, we reduce path counting and path selection queries to these two types of range queries, without adding too much time/space cost.

Lemma 6.3.1. *For any node x and its ancestor z, and any weight range $[p,q]$, the number of nodes on the path from x to z whose weights are in $[p,q]$ is equal to*

$$\texttt{par\_range\_sum}(\texttt{findopen}(P,z), \texttt{findopen}(P,x), [p,q]). \tag{6.1}$$

Proof. Let us consider the substring of P that begins with $\texttt{findopen}(P,z)$ and ends with $\texttt{findopen}(P,x)$, i.e., $P[\texttt{findopen}(P,z)..\texttt{findopen}(P,x)]$. By the properties of balanced parentheses, there are more opening parentheses than closing parentheses in this substring. In addition, each opening parenthesis in this substring has a matching closing parenthesis before $\texttt{findopen}(P,x)$ iff this opening parenthesis corresponds to a node that is not between x and z. Hence, only a node on the path from x to z whose weight is in $[p,q]$ contributes 1 to the value computed in Equation 6.1. □

Lemma 6.3.2. *Within $O(n)$ words of additional space and $O(\lg n)$ additional query/update time, path counting queries and path selection queries over T can be reduced to a constant number of range sum queries and two-range selection queries over $P[1..2n]$, respectively.*

Proof. We make use of an augmented red-black tree [37] to store the identifiers of nodes in preorder. The leaves of this red-black tree each represent a single parenthesis, and the internal nodes each represent a substring of P. Inside each internal node, we maintain two values: the size of the substring this internal node represents and the sum of scores over all parentheses in the substring. Clearly conversions between identifiers and preorder ranks can be supported in $O(\lg n)$ query/update time and $O(n)$ words of space. In addition, we maintain $P[1..2n]$ using Navarro and Sadakane's structure [82]. Thus, given the identifier of a node x, we can obtain the positions of the corresponding opening and closing parentheses for x within the same amount of query/update time and space.

First we show how to reduce path counting queries. Let x and y be the endpoints of a query path, and let $[p,q]$ be a query range. We compute $z = \texttt{LCA}(x,y)$, $i = \texttt{findopen}(P,z)$, $j_1 = \texttt{findopen}(P,x)$, and $j_2 = \texttt{findopen}(P,y)$. By Lemma 6.3.1, the answer to the path counting query is equal to

$$\texttt{par\_range\_sum}(i, j_1, [p,q]) + \texttt{par\_range\_sum}(i, j_2, [p,q]) - \mathbf{1}_{p,q}(z),$$

where $\mathbf{1}_{p,q}(z)$ is 1 if the weight of z is in $[p,q]$, or 0 otherwise. In other words, we decompose the query path from x to y into two top-to-bottoms paths that join at z, count nodes on these top-to-bottoms paths by calling $\texttt{par\_range\_sum}$, and deduct 1 if the weight of z is in the query range.

The reduction for path selection queries is similar. Let x and y be the endpoints of a query path, and let k be the rank of the weight we want to select. We still obtain z, i, j_1, and

j_2 as in the previous paragraph; clearly $P[i]$, $P[j_1]$, and $P[j_2]$ are all opening parentheses. Then we compute $q = \texttt{par\_range\_sel}(i, j_1, j_2, k)$ and $q' = \texttt{par\_range\_sel}(i, j_1, j_2, k+1)$. Thus, the answer to the original path selection query is q if the weight of z is larger than q, or q' otherwise. □

By the reduction, we need only to consider the support for range sum and two-range selection queries.

6.3.1 Supporting Range Sum and Path Counting

The main structure is a weight-balanced B-tree [7], W, with leaf parameter 1 and branching factor $d_1 = \lceil \lg^{\epsilon_1} n \rceil$ for some constant $0 < \epsilon_1 < 1/2$. The tree W stores parentheses at its $2n$ leaves, sorted in non-decreasing order of associated weights. As in Section 6.2, the levels of W are numbered 0, 1, 2, ..., t_1, starting from the leaf level, where $t_1 = O(\lg n / \lg\lg n)$ denotes the number of the root level. Inside each internal node v of W, we store the smallest and largest weights in the subtree, $W(v)$, rooted at the node v. Given $1 \leq i \leq 2n$, one can determine the path from the root of W to the leaf storing $P[i]$ in $O(\lg n)$ time, as it requires $O(\lg\lg n)$ time to perform a binary search over the values stored in the children of an arbitrary internal node.

We also store a dynamic sequence $Y(v)$ for each internal node v in W, which represents all of the parentheses stored in the leaves of $W(v)$, sorted in non-decreasing order of their indices in P. Each parenthesis is represented by a label α, which is the index of the child of node v's subtree in which the parenthesis is contained, and a score e, which is equal to $+1$ if the parenthesis is an opening one and -1 otherwise. Clearly the label is bounded above by $4d_1 = O(\lg^{\epsilon_1} n)$. For $\ell > 0$, we concatenate all the sequences $Y(v)$ for each node v at level ℓ in W into a sequence of length n, denoted by Y_ℓ.

We represent each sequence Y_ℓ using the dynamic data structure of He and Munro [61]. Depending on the context, we denote by Y_ℓ both the sequence and the data structure that represents the sequence. The following operations on the sequence Y_ℓ are supported:

- $\texttt{access}(Y_\ell, i)$: returns the label and the score of the i-th entry in Y_ℓ;
- $\texttt{rank}_\alpha(Y_\ell, i)$: returns the number of occurrences of label α in $Y_\ell[1..i]$;
- $\texttt{range\_sum}(Y_\ell, i_1, i_2, \alpha_1, \alpha_2)$: returns the total score of entries in $Y_\ell[i_1..i_2]$ whose labels are in $[\alpha_1..\alpha_2]$;
- $\texttt{insert}(Y_\ell, i, \alpha, e)$: inserts an entry, which has label α and score e, between $Y_\ell[i-1]$ and $Y_\ell[i]$;

- delete(Y_ℓ, i): deletes $Y_\ell[i]$ from Y_ℓ.

Lemma 6.3.3 (Modified from [61, 63]). *Under the word RAM model with word size $w = \Omega(\lg n)$, a sequence $Y_\ell[1..2n]$ of parentheses whose labels are drawn from an alphabet of size $\sigma = O(\lg^\epsilon n)$ for some constant $\epsilon \in (0, 1)$, and scores are drawn from $\{+1, -1\}$, can be represented using $2n \lg \sigma + o(n \lg \sigma)$ bits to support* access*,* rank*,* range_sum*,* insert *and* delete *in $O(\lg n / \lg \lg n)$ time. Furthermore, a batch of m update operations can be performed in $O(m)$ time on $Y_\ell[i..i + m - 1]$ in which the j-th update operation changes the label of $Y_\ell[i + j - 1]$, provided that $m > 5M/ \lg \sigma$ and $M = \lceil \lceil \lg n \rceil^2 / \lg \lceil \lg n \rceil \rceil$.*

Proof. Note that our formulation of this lemma can be reduced to its original form, where the score of each element is 1 (He and Munro [61] and He et al. [63] did not define the score function at all, but they defined operations that are equivalent to those defined in our chapter when the score of each element is 1). To show their equivalence, we need only to construct another sequence, $Z[1..2n]$, of integers. For $1 \le i \le 2n$, $Z[i]$ is equal to the label of $Y_\ell[i]$ if $Y_\ell[i]$ is an opening parenthesis, or the label of $Y_\ell[i]$ plus σ if $Y_\ell[i]$ is a closing parenthesis. Maintaining the sequence Z using the original form of this lemma as described by He et al. [63], it is easy to reduce the above operations on Y_ℓ to the supported operations on Z. □

We now describe how to support par_range_sum$(i, j, [p, q])$ queries over P. Starting with the t_1-st level, or the root node, r, of W, we compute the first child and the last child of r whose leaf descendants have weights in $[p, q]$, using binary search. Let their indices be α_1 and α_2. We only consider the case in which α_1 and α_2 both exist; the degenerate cases can be handled similarly. Let q' be the smallest weight contained in the α_1-st subtree of r and p' be the largest weight contained in the α_2-nd subtree of r, respectively. Thus we can split the query range $[p, q]$ into $[p, q')$, $[q', p']$, and $(p', q]$. The middle subquery can be computed by calling range_sum$(Y_{t_1}, i, j, \alpha_1, \alpha_2)$. The first subquery can be solved recursively by querying on the $(\alpha_1 - 1)$-st subtree of r, for which i and j are translated to $\texttt{rank}_{\alpha_1 - 1}(Y_{t_1}, i - 1) + 1$ and $\texttt{rank}_{\alpha_1 - 1}(Y_{t_1}, j)$, respectively. The third subquery can be handled similarly by querying on the $(\alpha_2 + 1)$-st subtree of r.

Finally we analyze the query time and space cost in the following lemma.

Lemma 6.3.4. *The data structures described in this section support* par_range_sum *queries and thus path counting queries in $O((\lg n / \lg \lg n)^2)$ time and $O(n)$ words of space.*

Proof. Given a query range, the algorithm described above will split it into $O(t_1) = O(\lg n / \lg \lg n)$ subqueries, each requiring a call to range_sum. By Lemma 6.3.3, the overall query time is $O((\lg n/\lg\lg n)^2)$.

Regarding the space cost, Lemma 6.3.2 and the weight-balanced B-tree W require $O(n)$ words of space. For $1 \leq \ell \leq t_1$, each Y_ℓ occupies $O(n \lg \sigma) = O(n \lg\lg n)$ bits of space. Therefore the overall space cost is $O(n)$ words. □

6.3.2 Space-Efficient Ranking Trees

Following He et al.'s approach [63], we build a space-efficient ranking tree, $R(v)$, on each internal node v of W, which represents all of the parentheses stored in the leaves of $W(v)$, sorted in non-decreasing order of indices. Each space-efficient ranking tree is a weight-balanced B-tree with leaf parameter $\Theta(M/\lg\lceil\lg n\rceil) = \Theta((\lg n/\lg\lg n)^2)$, where M is defined in Lemma 6.3.3, and branching factor $d_2 = \lceil \lg^{\epsilon_2} n \rceil$ for constant $0 < \epsilon_2 < 1 - \epsilon_1$. Thus, the height, t_2, of the ranking tree $R(v)$ is bounded above by $O(\lg n/\lg\lg n)$, and each leaf of $R(v)$ corresponds to a substring of $Y(v)$, which is stored in the dynamic sequence Y_ℓ.

On each internal node, u, of $R(v)$, we maintain a dynamic *searchable partial sums* structure [92] for making branching decisions. Let f_2 be the degree of u in $R(v)$, and, for $1 \leq i \leq f_2$, let p_i be the number of parentheses stored in the subtree rooted at the i-th child of u. We also denote by $T'(u)$ the subtree rooted at u. Given a rank j within the parentheses stored in the leaves of $T'(u)$, this structure can determine in $O(1)$ time the smallest i so that $p_1 + p_2 + \cdots + p_i \geq j$, i.e., the subtree of u that contains the parenthesis whose rank is j. With an $o(n)$-bit precomputed universal lookup table, this structure uses $O(\lg n)$ bits of space per child of u, and requires $O(1)$ update time to handle insertions/deletions of a parenthesis within $T'(u)$.

We also store the matrix structure of Brodal et al. [23] on each internal node u of $R(v)$. We denote by f_1 the degree of v in W, and denote by $W(v_1), \ldots, W(v_{f_1})$ the subtrees rooted at each child of v from left to right. Similarly, $T'(u_1), \ldots, T'(u_{f_2})$ denote the subtrees rooted at each child of u from left to right, where f_2 is the degree of u in $R(v)$ as defined in the previous paragraph. It is clear that $f_1 \leq d_1$ and $f_2 \leq d_2$. The matrix structure associated with node u is a partial sum structure with respect to $W(v_1), \ldots, W(v_{f_1})$ and $T'(u_1), \ldots, T'(u_{f_2})$. More formally, the matrix structure M^u is a $f_1 \times f_2$ matrix, where entry $M^u_{p,q}$ is the summation of scores over the parentheses that are contained in both $\bigcup_{i=1}^{p} W(v_i)$ and $\bigcup_{j=1}^{q} T'(u_j)$. It should be drawn to the reader's attention that the summation of scores is equal to the number of opening parentheses minus the number of closing parentheses, which could result in a positive or negative value. The matrix structure M^u is stored in two ways. The first representation is a standard table, where each entry is stored as two's complement in $O(\lg n)$ bits. In the second representation, we divide each entry into *sections* of $\Theta(\lg^{\epsilon_1} n)$ bits, and number them $s_1, s_2, \ldots, s_g$, where $g = \Theta(\lg^{1-\epsilon_1} n)$ and s_g contains the most significant bits. For technical reasons, consecutive sections have an overlap of $b = \lceil 2 \lg\lg n \rceil$ bits. In the second representation, for each column c, the i-th section of each entry in column c is packed into a single word $w^u_{c,i}$. For similar technical reasons, each section stored in the packed word $w^u_{c,i}$ is padded with two leading zero bits.

We will later show how to support path selection queries using the matrix structure in Section 6.3.3. Here we only summarize its properties in the following lemma.

Lemma 6.3.5 ([23, 63])**.** *The matrix structure M^u for node u in the ranking tree $R(v)$ occupies $O(\lg^{1+\epsilon_1+\epsilon_2} n)$ bits of space, and can be constructed in $o(\lg^{1+\epsilon_1+\epsilon_2} n)$ time. Whenever a parenthesis is inserted into or deleted from $R(v)$, the matrix structure stored on each node along the update path can be updated in $O(1)$ amortized time per node.*

As in He et al.'s work [63], we reduce the space cost of ranking trees by storing the leaves of $R(v)$ implicitly.

Lemma 6.3.6. *Let u be a leaf in $R(v)$ and S be the substring of $Y(v)$ that u represents, where each parenthesis in S is associated with a label drawn from an alphabet of size $\sigma = O(\lg^{\epsilon} n)$ for some constant $0 < \epsilon < 1$, and a score drawn from $\{+1, -1\}$. Using a universal table of size $o(n)$ bits, for any $1 \le z \le |S|$, one can compute an array $C_z = \{c_1, c_2, \ldots, c_\sigma\}$ in $O(\lg n / \lg\lg n)$ time, where $c_\alpha = \texttt{range\_sum}(S, 1, z, 1, \alpha)$ for $1 \le \alpha \le \sigma$.*

Proof. Let ℓ be the level of v; clearly S is a substring of Y_ℓ. Remember that, in the proof of the Lemma 6.3.3, Y_ℓ is stored as a sequence, $Z[1..2n]$, of integers drawn from $[1..2\sigma]$. For $1 \le i \le 2n$, $Z[i]$ is equal to the label of $Y_\ell[i]$ if $Y_\ell[i]$ is an opening parenthesis, or the label of $Y_\ell[i]$ plus σ if $Y_\ell[i]$ is a closing parenthesis. Thus, S also corresponds to a substring, Z', of the sequence Z. Following the same approach of He et al. [63], one can compute an array $C'_z = \{c'_1, c'_2, \ldots, c'_{2\sigma}\}$ using $O(\lg n / \lg\lg n)$ query time and a universal table with $o(n)$ bits of space, where $c'_\beta = \texttt{rank}_\beta(S, z)$ for $1 \le \beta \le 2\sigma$. Finally we obtain C_z by setting $c_\alpha = \sum_{1 \le \beta \le \alpha} (c'_\beta - c'_{\beta+\sigma})$. □

The analysis for the space cost and construction time is summarized in the following lemma.

Lemma 6.3.7. *For any node v in W, its ranking tree $R(v)$ requires $O(\frac{m(\lg\lg n)^2}{\lg^{1-\epsilon_1} n} + w)$ bits of space and $O(m)$ construction time, if $W(v)$ consists of m parentheses and constant-time access to $Y(v)$ is supported.*

Proof. The ranking tree $R(v)$ contains $O(m/d_2 \times (\lg\lg n/\lg n)^2) = O(m(\lg\lg n)^2/\lg^{2+\epsilon_2} n)$ internal nodes. As the searchable partial sums structure and the matrix structure stored for each internal node of $R(v)$ occupies $O(\lg^{1+\epsilon_2} n)$ and $O(\lg^{1+\epsilon_1+\epsilon_2} n)$ bits of space, respectively, the space occupied by all the internal nodes is $O(m(\lg\lg n)^2/\lg^{1-\epsilon_1} n)$ bits. The pointers between the internal nodes can be maintained in $O(\lg n)$ bits per pointer and $O(1)$ amortized time per update using the memory blocking techniques for dynamic data structures (see [61, Appendix J] for an example). The overall space cost is $O(\frac{m(\lg\lg n)^2}{\lg^{1-\epsilon_1} n} + w)$ bits, as we always need to keep a pointer to the root of the ranking tree.

The analysis of the construction time is similar. The number of internal nodes is $O(m(\lg\lg n)^2/\lg^{2+\epsilon_2} n)$, and each internal node requires $O(\lg^{\epsilon_2} n)$ and $o(\lg^{1+\epsilon_1+\epsilon_2} n)$ time to construct. Thus the overall construction time is $O(m)$. □

In particular, as He et al. pointed out, we do not even need to store the ranking tree if $m = O((\lg n/\lg\lg n)^2)$. Instead, by Lemma 6.3.6, we can directly query $Y(v)$ and make a branching decision in $O(\lg n/\lg\lg n)$ time.

6.3.3 Supporting Two-Range Selection and Path Selection

In this section we describe how to make use of ranking trees to support two-range selection queries, and thus path selection queries. First we observe that:

Lemma 6.3.8. *For opening parentheses at indices i and j that satisfy $1 \le i \le j <$ `findclose`(i), the value of* `par_range_sum`$(i, j, (-\infty, q])$ *is increasing with q.*

Proof. Let x and z be the nodes that correspond to the opening parentheses at indices i and j, respectively. By Lemma 6.3.1, `par_range_sum`$(i, j, (-\infty, q])$ is equal to the number of nodes between x and z whose weights are less than or equal to q. Therefore, the value `par_range_sum`$(i, j, (-\infty, q])$ must be increasing with q. □

Lemma 6.3.8 enables us to answer `par_range_sel` queries using `range_sum` operations and binary search, which could imply a basic query algorithm that uses $O(\lg^2 n/\lg\lg n)$ time. For each internal node v in W and i, j_1, j_2 given in the context, we define $s_{v,\gamma} =$ `range_sum`$(Y(v), i, j_1, 1, \gamma)+$`range_sum`$(Y(v), i, j_2, 1, \gamma)$. Given a two-range selection query `par_range_sel`(i, j_1, j_2, k), starting with the root node r of W, we determine the value of γ so that $s_{r,\gamma-1} < k$ and $s_{r,\gamma} \ge k$. Using binary search, this uses $O(\lg f_1) = O(\lg\lg n)$ calls to `range_sum`, which require $O(\lg n)$ time in total. The answer is located in the γ-th subtree of r, which we recurse on after translating i, j_1, and j_2 into `rank`$_\gamma(Y(r), i-1)+1$,

$\mathtt{rank}_\gamma(Y(r), j_1)$, and $\mathtt{rank}_\gamma(Y(r), j_2)$, respectively. For each of these t_1 levels, our query algorithm uses $O(\lg n)$ time, and its overall query time is $O(\lg^2 n/\lg\lg n)$.

To reduce query time, we will avoid performing binary search on each level of W. The main idea is to use word-level parallelism and the second representation of matrix structures stored in internal nodes of ranking trees. For each node v on the root-to-leaf path of W our algorithm will traverse through, we conceptually divide the values of $s_{v,1}, s_{v,2}, \ldots$ into sections in the same way as we divide the entries of the matrix structures into sections. During the traversal from the root of W to a leaf, our algorithm will maintain an index h such that, when node v on the path is being examined, the indices of the most significant non-zero sections of $s_{v,1}, s_{v,2}, \ldots$ are no greater than h. The value of h is equal to g initially, and will be decremented during the traversal. Our algorithm will calculate and examine the h-th sections of $s_{v,1}, s_{v,2}, \ldots$ in a parallel manner. We query $R(v)$ for the path from the root of $R(v)$ to the leaf whose corresponding substring contains $Y(v)[j_1]$. If the search path goes through the c-th child on node u of $R(v)$, then we add $w^u_{c,h}$ into a running total w_1. Similarly, we obtain running totals w_2 and w_0 for j_2 and $i - 1$, respectively. Then we compute $w_1 + w_2$ and set the most significant bit of each section to be 1, and compute $2w_0$ and set the most significant bit of each section to be 0. Subtracting the word representing $2w_0$ from that representing $w_1 + w_2$, and setting the most significant bit of each section to be 0 in the result $\overline{w}$, we finally obtain an approximation of the h-th sections of $s_{v,1}, s_{v,2}, \ldots$ as $\overline{s}_{v,1}, \overline{s}_{v,2}, \ldots$. The computation of $\overline{w}$ requires $O(\lg n/\lg\lg n)$ query time, because, in the second representation of the matrix structures, consecutive sections have an overlap of $b = \lceil 2\lg\lg n\rceil$ bits, and each section is padded with two leading zero bits. For $\beta = 1, 2, \ldots$, the difference between $s_{v,\beta}$ and $\overline{s}_{v,\beta}$ is $O(\lg n/\lg\lg n)$, which is the number of additions and subtractions we used to compute $\overline{w}$.

We then determine the range $[\alpha_1..\alpha_2]$ of children of v such that, for any $\beta \in [\alpha_1..\alpha_2]$, the difference between the h-th section of $s_{v,\beta}$ and that of k is smaller than 2^{b-1}. This can be also done in $O(1)$ time using a universal lookup table of size $o(n)$ bits, since $\overline{w}$ occupies $O(\lg n)$ bits. Here we know that γ must be contained in $[\alpha_1..\alpha_2]$. We can use at most two calls to $\mathtt{range\_sum}$ to determine whether $\gamma = \alpha_1$ or $\gamma = \alpha_2$. If neither of them is true, then we simply determine the value of γ using a binary search. After the binary search, the value of h will be decremented by 1, since the values of s_{v,α_1} and s_{v,α_2} differ by at most 2^b.

Now we can analyze the query time. On each node v between the root and some leaf of W, the algorithm queries the ranking tree $R(v)$ to obtain $\overline{w}$. By Section 6.3.2, this requires only $O(\lg n/\lg\lg n)$ time per level of W, and $O((\lg n/\lg\lg n)^2)$ time in total. Whenever we use binary search to determine γ, the value of h is decremented by 1. Thus we only need to use binary search at most $g = O(\lg^{1-\epsilon_1} n)$ times, and the time cost is up to $O(\lg n) \times g = o((\lg n/\lg\lg n)^2)$. Summing up the discussions, the overall query time is only $O((\lg n/\lg\lg n)^2)$, which improves the initial basic solution by a factor of $\lg\lg n$.

The following lemma summarizes the query time and space cost of supporting path selection queries.

Lemma 6.3.9. *The data structures described in this section support* `par_range_sel` *queries and thus path selection queries in $O((\lg n/\lg\lg n)^2)$ time and $O(n)$ words of space.*

Proof. The query time has been analyzed, and the analysis of the space cost is based on Lemmas 6.3.4 and 6.3.7. By Lemma 6.3.7, for each level of W except the bottom $O(\lg\lg n)$ ones, the ranking trees require $O(\frac{n(\lg\lg n)^2}{\lg^{1-\epsilon_1} n})$ bits of space. In addition, the ranking trees for the bottom $O(\lg\lg n)$ levels require no extra space. Adding the space occupied by the weight-balanced B-trees and dynamic sequences, the overall space cost is $O(n)$ words. □

6.3.4 Handling Updates

Remember that the input tree T is represented as a sequence P of balanced parentheses. To insert a node into T or delete one from T, we need only to insert into P or delete from P a corresponding pair of parentheses. In the rest of this section, we will describe how to insert a single parenthesis into P, which requires us to update the weight-balanced B-tree W, ranking trees $R(v)$'s and sequences Y_ℓ's accordingly. Deletions of parentheses can be handled similarly by following the approach of Brodal et al. [20], which marks nodes that have been deleted, and rebuilds the whole data structure after $\Theta(n)$ updates.

A parenthesis ρ to insert is specified by its weight a, its score e, and its index i in P after insertion. To insert ρ, we first determine the update path π of ρ in W, which starts at the root of W and ends at the leaf that will store ρ after the insertion. This path can be found in $O(\lg n)$ time by performing binary search on each internal node of π. Starting from the root node of W, we update each internal node v on π level by level, for which the ranking tree $R(v)$ and the dynamic sequence $Y(v)$ are also updated accordingly. During the process, some internal node of W may have too many children and will split. For now, we only consider the case in which no node of W will split during the update.

On the root r of W, we first update its node structure, which requires $O(d_1) = O(\lg^{\epsilon_1} n)$ time by the properties of weight-balanced B-trees. We also perform `insert`$(Y(r), i, \alpha, e)$ if the α-th child of r is also on the path π, which requires $O(\lg n/\lg\lg n)$ time by Lemma 6.3.3. Then we insert the parenthesis ρ into the ranking tree $R(r)$, where, after the insertion, ρ's index will be of rank i over the parentheses stored in the leaves of $R(r)$. Starting from the root node of $R(v)$, we update each internal node u on the update path of ρ in $R(v)$ from top to bottom, for which the node structure of u is updated in $O(d_2) = O(\lg^{\epsilon_2} n)$ time, while the searchable partial sums structure and the matrix structure M^u are updated in

$O(1)$ amortized time. Whenever a node u of $R(v)$ splits, we need to rebuild the auxiliary data structures on u's parent and create auxiliary data structures for the nodes that u is split into. By Lemma 6.3.5 and the properties of the searchable partial sums structure, these auxiliary structures require $o(\lg^{1+\epsilon_1+\epsilon_2} n)$ construction time, which is $o(1)$ amortized time per insertion to $R(v)$ according to the properties of weighted-balanced B-trees. After that, we reset $i = \mathtt{rank}_\alpha(Y(r), i)$ and recurse on the α-th child of r. So far our algorithm uses $O(\lg n/\lg\lg n)$ time per level, and $O((\lg n/\lg\lg n)^2)$ update time to insert ρ.

Now we consider how to split node v of W when v has too many children. Without loss of generality, we assume that v has a parent v_p at level ℓ. After v splits, we need to update the ranking tree $R(v_p)$ and the substring $Y(v_p)$ of the sequence Y_ℓ, because the index of v has changed with respect to the other children of v_p. Let m be the number of parentheses stored in $W(v)$. Thus the length of $Y(v_p)$ is bounded above by $O(md_1) = O(m\lg^{\epsilon_1} n)$, and we will later show how to replace $Y(v_p)$ in $O(m\lg^{\epsilon_1} n)$ time. By Lemma 6.3.7, $R(v_p)$ can be reconstructed in $O(m\lg^{\epsilon_1} n)$ time, provided that we have access to the sequence $Y(v_p)$ after the split. By properties of weight-balanced B-trees, at least $m/2$ insertions have to be performed below v since the more recent split of v. By amortizing the reconstruction cost over these insertions, this is $O(\lg^{\epsilon_1} n)$ time. As splits can happen at each level of W, the amortized cost per insertion to W is $O(\lg^{\epsilon_1} n \times \lg n/\lg\lg n)$ time.

The remaining task is to update $Y(v_p)$. Let f denote the degree of v; clearly $f = O(\lg^{\epsilon_1} n)$ and $d \leq f$. Similarly, let f_p be the degree of v_p, and suppose that v is the γ-th child of v_p. We also denote by v_1 and v_2 the nodes which v splits into. After the split, the first d children of v will be assigned to v_1, while the others will be assigned to v_2. We will generate an updated sequence $Y'(v_p)$ for the node v_p after the split, and replace $Y(v_p)$ with $Y'(v_p)$ in Y_ℓ using batched updates.

To generate $Y'(v_p)$, we first retrieve the sequences $Y(v_p)$ and $Y(v)$ by traversing the B-tree structure that stores $Y(v_p)$. This requires $O(m\lg^{\epsilon_1} n + \lg n/\lg\lg n)$ time, as we have to traverse at least one root-to-leaf path of the B-tree. We then scan $Y(v_p)$ from left to right, and append parentheses to $Y'(v_p)$ during the process, where $Y'(v_p)$ is initially empty. When we encounter a parenthesis with label α and score e, we append a parenthesis with the same score and label $\beta \in \{\alpha, \alpha+1\}$ to $Y'(v_p)$. The value of β is α if $\alpha \in \{1, \ldots, \gamma-1\}$, as the parenthesis belongs to some child of v_p that appears before v, while β is $\alpha+1$ if $\alpha \in \{\gamma+1, \ldots, f_p\}$, as the parenthesis belongs to some child of v_p that appears after v. In the case when $\alpha = \gamma$, or the parenthesis belongs to v before the split, we further check the label α' of the corresponding entry of $Y(v)$. The value of β is γ if $\alpha' \in \{1, \ldots, d\}$; otherwise β is $\gamma+1$. In sum, it requires $O(m\lg^{\epsilon_1} n + \lg n/\lg\lg n)$ time to generate $Y'(v_p)$, where the additive $O(\lg n/\lg\lg n)$ term is absorbed in all but a constant number of levels close to the leaves of W. By an analysis similar to that of updating the ranking tree, the amortized cost of the sequence generation algorithm is $O(\lg^{\epsilon_1} n \times \lg n/\lg\lg n)$ time per insertion to

W.

Finally we replace $Y(v_p)$ with $Y'(v_p)$ in Y_ℓ. If the size of $W(v_p)$ is greater than $5M/\lg\sigma$ for $M = \lceil\lceil\lg n\rceil^2/\lg\lceil\lg n\rceil\rceil$, then we make use of the batched updates as described in Lemma 6.3.3, which requires $O(1)$ time for each parenthesis to replace. If the size of $W(v_p)$ is smaller, which only occurs at a constant number of bottom levels of W, we simply replace each parenthesis using `insert` and `delete` operations in $O(\lg n/\lg\lg n)$ time. As the algorithm that generated the sequence $Y'(v_p)$, the amortized cost per insertion to W is $O(\lg^{\epsilon_1} n \times \lg n/\lg\lg n)$ time.

Summarizing the discussions above, and combining Lemmas 6.3.4 and 6.3.9, we present the following theorem:

Theorem 6.3.10. *Under the word RAM model with word size $w = \Omega(\lg n)$, an ordinal tree on n weighted nodes can be stored in $O(n)$ words of space, such that path counting queries can be answered in $O((\lg n/\lg\lg n)^2)$ time, and* `modify_weight`*,* `node_insert` *and* `node_delete` *can be supported in $O((\lg n/\lg\lg n)^2)$ amortized time.*

6.4 Discussion

The major open problem is whether our dynamic data structure for path reporting queries can be further improved. As an example, is it possible to reduce the update time described in Theorem 6.2.15 to $O((\lg n/\lg\lg n)^2)$ while preserving the same query time and space cost? There are two bottlenecks: `cut` and `link` are supported using $\omega(\lg n)$ time in Lemma 6.2.7; the inter-level pointers for each level require $\omega(\lg n)$ update time in the proof of Lemma 6.2.13. Whether the $(\lg n/\lg\lg n)^2$ term in query time can be decreased without sacrificing update time or space cost is another interesting problem.

Chapter 7

Conclusion

In this thesis we have studied several data structure problems for labeled or weighted trees. We have designed in-memory data structures to support various types of path queries and operations using efficient space and query time. In dynamic settings, we have also considered how to handle insertions or deletions of nodes, and modifications to weights of nodes.

In Chapter 3, we design succinct data structures to encode an input tree T using $nH(W_T)+2n+o(n\lg\sigma)$ bits of space, where T consists of n nodes, their weights are drawn from $[1..\sigma]$, and $H(W_T)$ is the entropy of the multiset of the weights of the nodes in T. Our data structures support path counting queries in $O(\lg\sigma/\lg\lg n+1)$ time, path reporting queries in $O((occ+1)(\lg\sigma/\lg\lg n+1))$ time, and path median and path selection queries in $O(\lg\sigma/\lg\lg\sigma)$ time, where occ is the size of the output. Our results not only greatly improve the best known data structures [31, 75, 65], but also match the lower bounds for path counting, median and selection queries [86, 87, 71] when $\sigma=\Omega(n/\text{polylog}(n))$.

In Chapter 4, we design succinct representations of labeled ordinal trees that support a much broader collection of operations than previous work. Our approach presents a framework for succinct representations of labeled ordinal trees that is able to handle large alphabets. This answers an open problem presented by Geary et al. [54], which asks for representations of labeled ordinal trees that remain space-efficient for large alphabets. We further extend our work and present the first succinct representations for dynamic labeled ordinal trees that support several label-based operations including finding the level ancestor with a given label.

In Chapter 5, we design novel succinct indices for path minimum queries. We present

- an index within $O(m)$ bits of additional space that supports queries in $O(\boldsymbol{\alpha}(m,n))$

time and $O(\boldsymbol{\alpha}(m,n))$ accesses to the weights of nodes, for any integer $m \geq n$; and

- an index within $2n + o(n)$ bits of additional space that supports queries in $O(\boldsymbol{\alpha}(n))$ time and $O(\boldsymbol{\alpha}(n))$ accesses to the weights of nodes.

Here $\boldsymbol{\alpha}(m,n)$ is the inverse-Ackermann function, and $\boldsymbol{\alpha}(n) = \boldsymbol{\alpha}(n,n)$. These indices give us the first succinct data structures for the path minimum problem. Following the same approach, we also develop succinct data structures for semigroup path sum queries. One of our data structures requires $n \lg \sigma + 2n + o(n \lg \sigma)$ bits of space and $O(\boldsymbol{\alpha}(n))$ query time, where σ is the size of the semigroup.

In the same chapter, using the succinct indices for path minimum queries, we achieve three different time-space tradeoffs for path reporting by designing

- an $O(n)$-word data structure with $O(\lg^{\epsilon} n + occ \cdot \lg^{\epsilon} n)$ query time;
- an $O(n \lg\lg n)$-word data structure with $O(\lg\lg n + occ \cdot \lg\lg n)$ query time; and
- an $O(n \lg^{\epsilon} n)$-word data structure with $O(\lg\lg n + occ)$ query time.

Here occ is the number of nodes reported and ϵ is an arbitrary constant between 0 and 1. These tradeoffs match the state of the art of two-dimensional orthogonal range reporting queries [27], which can be treated as a special case of path reporting queries. When the number of distinct weights is much smaller than n, we further improve both the query time and the space cost of these three results.

In Chapter 6, we design the first non-trivial linear-space data structure that supports path reporting in $O((\lg n/\lg\lg n)^2 + occ \lg n/\lg\lg n))$ query time, where n is the size of the input tree and occ is the output size, and the insertion and deletion of a node of an arbitrary degree in $O(\lg^{2+\epsilon} n)$ amortized time, for any constant $\epsilon \in (0,1)$. Our data structure supports queries much faster than the obvious solutions that directly dynamize solutions to the static version of this problem, which all require $\Omega((\lg n/\lg\lg n)^2)$ time for each node reported. We also design data structures that support path counting and path reporting queries in $O((\lg n/\lg\lg n)^2)$ time, and insertions or deletions of nodes in $O((\lg n/\lg\lg n)^2)$ amortized time. This matches the best known results for dynamic two-dimensional range counting [62] and range selection [63].

In addition to the specific open problems mentioned in the discussion sections of Chapters 3 to 6, we would like to point out several general directions for future work. First, it would be interesting if more types of path queries can be examined. A recent attempt has been made by Durocher et al. [41] to study path least-frequent element, path α-minority and path top-k color queries. Second, the results in this thesis are obtained under the standard word RAM model, which is for internal memory. When the data set does not fit into the memory, we need to design I/O-efficient algorithms and data structures. It remains open to extend our work to handle data sets in external memory. Finally, proving lower bounds for time-space tradeoffs and query-update tradeoffs is another open research field, especially the lower bounds that capture the differences between data structure problems on trees and paths. Can we find a family of queries that require more time and/or space cost on trees than paths? As described in Lemma 5.2.1, the query that asks for the minimum node is an example.

References

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. On finding lowest common ancestors in trees. *SIAM J. Comput.*, 5(1):115–132, 1976.

[2] Noga Alon and Baruch Schieber. Optimal preprocessing for answering on-line product queries. Technical report, Tel Aviv University, 1987.

[3] Stephen Alstrup, Gerth Stølting Brodal, and Theis Rauhe. New data structures for orthogonal range searching. In *41st Annual Symposium on Foundations of Computer Science, FOCS 2000, 12-14 November 2000, Redondo Beach, California, USA*, pages 198–207, 2000.

[4] Stephen Alstrup, Gerth Stølting Brodal, and Theis Rauhe. Optimal static range reporting in one dimension. In *Proceedings of the 33rd Annual ACM Symposium on Theory of Computing, STOC 2001, Heraklion, Crete, Greece, July 6-8, 2001*, pages 476–482, 2001.

[5] Stephen Alstrup and Jacob Holm. Improved algorithms for finding level ancestors in dynamic trees. In *Automata, Languages and Programming, 27th International Colloquium, ICALP 2000, Geneva, Switzerland, July 9-15, 2000, Proceedings*, pages 73–84, 2000.

[6] Stephen Alstrup, Thore Husfeldt, and Theis Rauhe. Marked ancestor problems. In *39th Annual Symposium on Foundations of Computer Science, FOCS '98, November 8-11, 1998, Palo Alto, California, USA*, pages 534–544, 1998.

[7] Lars Arge and Jeffrey Scott Vitter. Optimal external memory interval management. *SIAM J. Comput.*, 32(6):1488–1508, 2003.

[8] Maxim A. Babenko, Pawel Gawrychowski, Tomasz Kociumaka, and Tatiana A. Starikovskaya. Wavelet trees meet suffix trees. In *Proceedings of the Twenty-Sixth*

Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015, pages 572–591, 2015.

[9] Jérémy Barbay, Alexander Golynski, J. Ian Munro, and S. Srinivasa Rao. Adaptive searching in succinctly encoded binary relations and tree-structured documents. *Theoretical Computer Science*, 387(3):284–297, 2007.

[10] Jérémy Barbay, Meng He, J. Ian Munro, and Srinivasa Rao Satti. Succinct indexes for strings, binary relations and multilabeled trees. *ACM Transactions on Algorithms*, 7(4):52, 2011.

[11] Jérémy Barbay and S. Srinivasa Rao. Succinct encoding for XPath location steps. Technical Report CS-2006-10, University of Waterloo, 2006.

[12] Djamal Belazzougui and Gonzalo Navarro. Optimal lower and upper bounds for representing sequences. *ACM Transactions on Algorithms*, 11(4):31, 2015.

[13] Amir M. Ben-Amram. What is a "pointer machine"? *SIGACT News*, 26(2):88–95, 1995.

[14] Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In *LATIN*, pages 88–94, 2000.

[15] David Benoit, Erik D. Demaine, J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.

[16] Philip Bille. A survey on tree edit distance and related problems. *Theor. Comput. Sci.*, 337(1-3):217–239, 2005.

[17] Guy E. Blelloch. Space-efficient dynamic orthogonal point location, segment intersection, and range reporting. In *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2008, San Francisco, California, USA, January 20-22, 2008*, pages 894–903, 2008.

[18] Prosenjit Bose, Meng He, Anil Maheshwari, and Pat Morin. Succinct orthogonal range search structures on a grid with applications to text indexing. In *Algorithms and Data Structures, 11th International Symposium, WADS 2009, Banff, Canada, August 21-23, 2009. Proceedings*, pages 98–109, 2009.

[19] Karl Bringmann and Kasper Green Larsen. Succinct sampling from discrete distributions. In *Proceedings of the 45th Annual ACM Symposium on Theory of Computing, STOC 2013, Palo Alto, California, USA, June 1-4, 2013*, pages 775–782, 2013.

[20] Gerth Stølting Brodal, Pooya Davoodi, Moshe Lewenstein, Rajeev Raman, and S. Srinivasa Rao. Two dimensional range minimum queries and fibonacci lattices. In *Algorithms - ESA 2012 - 20th Annual European Symposium, Ljubljana, Slovenia, September 10-12, 2012. Proceedings*, pages 217–228, 2012.

[21] Gerth Stølting Brodal, Pooya Davoodi, and S. Srinivasa Rao. Path minima queries in dynamic weighted trees. In *Algorithms and Data Structures - 12th International Symposium, WADS 2011, New York, NY, USA, August 15-17, 2011. Proceedings*, pages 290–301, 2011.

[22] Gerth Stølting Brodal, Pooya Davoodi, and S. Srinivasa Rao. On space efficient two dimensional range minimum data structures. *Algorithmica*, 63(4):815–830, 2012.

[23] Gerth Stølting Brodal, Beat Gfeller, Allan Grønlund Jørgensen, and Peter Sanders. Towards optimal range medians. *Theor. Comput. Sci.*, 412(24):2588–2601, 2011.

[24] Gerth Stølting Brodal and Allan Grønlund Jørgensen. Data structures for range median queries. In *Algorithms and Computation, 20th International Symposium, ISAAC 2009, Honolulu, Hawaii, USA, December 16-18, 2009. Proceedings*, pages 822–831, 2009.

[25] Adam L. Buchsbaum, Loukas Georgiadis, Haim Kaplan, Anne Rogers, Robert Endre Tarjan, and Jeffery Westbrook. Linear-time algorithms for dominators and other path-evaluation problems. *SIAM J. Comput.*, 38(4):1533–1573, 2008.

[26] Timothy M. Chan, Meng He, J. Ian Munro, and Gelin Zhou. Succinct indices for path minimum, with applications to path reporting. In *Algorithms - ESA 2014 - 22th Annual European Symposium, Wroclaw, Poland, September 8-10, 2014. Proceedings*, pages 247–259, 2014.

[27] Timothy M. Chan, Kasper Green Larsen, and Mihai Pătraşcu. Orthogonal range searching on the RAM, revisited. In *Proceedings of the 27th ACM Symposium on Computational Geometry, SoCG 2011, Paris, France, June 13-15, 2011*, pages 1–10, 2011.

[28] Timothy M. Chan and Mihai Pătraşcu. Counting inversions, offline orthogonal range counting, and related problems. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010*, pages 161–173, 2010.

[29] Timothy M. Chan and Bryan T. Wilkinson. Adaptive and approximate orthogonal range counting. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013*, pages 241–251, 2013.

[30] Bernard Chazelle. Filtering search: A new approach to query-answering. *SIAM J. Comput.*, 15(3):703–724, 1986.

[31] Bernard Chazelle. Computing on a free tree via complexity-preserving mappings. *Algorithmica*, 2(1):337–361, 1987.

[32] Bernard Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM J. Comput.*, 17(3):427–462, 1988.

[33] Bernard Chazelle. Lower bounds for orthogonal range searching: I. the reporting case. *J. ACM*, 37(2):200–212, 1990.

[34] Bernard Chazelle and Leonidas J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1(2):133–162, 1986.

[35] Bernard Chazelle and Burton Rosenberg. The complexity of computing partial sums off-line. *Int. J. Comput. Geometry Appl.*, 1(1):33–45, 1991.

[36] David R. Clark and J. Ian Munro. Efficient suffix trees on secondary storage (extended abstract). In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, 28-30 January 1996, Atlanta, Georgia.*, pages 383–391, 1996.

[37] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

[38] Erik D. Demaine, Gad M. Landau, and Oren Weimann. On cartesian trees and range minimum queries. *Algorithmica*, 68(3):610–625, 2014.

[39] Erik D. Demaine and Alejandro López-Ortiz. A linear lower bound on index size for text retrieval. *J. Algorithms*, 48(1):2–15, 2003.

[40] Brandon Dixon, Monika Rauch, and Robert Endre Tarjan. Verification and sensitivity analysis of minimum spanning trees in linear time. *SIAM J. Comput.*, 21(6):1184–1192, 1992.

[41] Stephane Durocher, Rahul Shah, Matthew Skala, and Sharma V. Thankachan. Linear-space data structures for range frequency queries on arrays and trees. *Algorithmica*, 74(1):344–366, 2016.

[42] Arash Farzan and J. Ian Munro. A uniform paradigm to succinctly encode various families of trees. *Algorithmica*, 68(1):16–40, 2014.

[43] Arash Farzan, Rajeev Raman, and S. Srinivasa Rao. Universal succinct representations of trees? In *Automata, Languages and Programming, 36th International Colloquium, ICALP 2009, Rhodes, Greece, July 5-12, 2009, Proceedings, Part I*, pages 451–462, 2009.

[44] Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and S. Muthukrishnan. Compressing and indexing labeled trees, with applications. *Journal of the ACM*, 57(1), 2009.

[45] Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3(2), 2007.

[46] Paolo Ferragina and Rossano Venturini. A simple storage scheme for strings achieving entropy bounds. *Theoretical Computer Science*, 372(1):115–121, 2007.

[47] Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM J. Comput.*, 40(2):465–492, 2011.

[48] Greg N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM J. Comput.*, 14(4):781–798, 1985.

[49] Greg N. Frederickson. Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. *SIAM J. Comput.*, 26(2):484–538, 1997.

[50] Greg N. Frederickson. A data structure for dynamically maintaining rooted trees. *J. Algorithms*, 24(1):37–65, 1997.

[51] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Sci.*, 47(3):424–436, 1993.

[52] Harold N. Gabow. Data structures for weighted matching and nearest common ancestors with linking. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 1990, San Francisco, California, USA, January 22-24, 1990*, pages 434–443, 1990.

[53] Travis Gagie, Gonzalo Navarro, and Simon J. Puglisi. New algorithms on wavelet trees and applications to information retrieval. *Theor. Comput. Sci.*, 426:25–41, 2012.

[54] Richard F. Geary, Rajeev Raman, and Venkatesh Raman. Succinct ordinal trees with level-ancestor queries. *ACM Transactions on Algorithms*, 2(4):510–534, 2006.

[55] Beat Gfeller and Peter Sanders. Towards optimal range medians. In *Automata, Languages and Programming, 36th International Colloquium, ICALP 2009, Rhodes, Greece, July 5-12, 2009, Proceedings, Part I*, pages 475–486, 2009.

[56] Alexander Golynski. Optimal lower bounds for rank and select indexes. *Theor. Comput. Sci.*, 387(3):348–359, 2007.

[57] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 12-14, 2003, Baltimore, Maryland, USA.*, pages 841–850, 2003.

[58] Roberto Grossi, Alessio Orlandi, Rajeev Raman, and S. Srinivasa Rao. More haste, less waste: Lowering the redundancy in fully indexable dictionaries. In *Proceedings of the 29th International Symposium on Theoretical Aspects of Computer Science*, volume 25 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 517–528, Dagstuhl, Germany, 2009. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[59] Torben Hagerup. Sorting and searching on the word RAM. In *STACS 98, 15th Annual Symposium on Theoretical Aspects of Computer Science, Paris, France, February 25-27, 1998, Proceedings*, pages 366–398, 1998.

[60] Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.

[61] Meng He and J. Ian Munro. Succinct representations of dynamic strings. In *Proceedings of String Processing and Information Retrieval - 17th International Symposium*, volume 6393 of *Lecture Notes in Computer Science*, pages 334–346. Springer, 2010.

[62] Meng He and J. Ian Munro. Space efficient data structures for dynamic orthogonal range counting. *Comput. Geom.*, 47(2):268–281, 2014.

[63] Meng He, J. Ian Munro, and Patrick K. Nicholson. Dynamic range selection in linear space. In *Algorithms and Computation - 22nd International Symposium, ISAAC 2011, Yokohama, Japan, December 5-8, 2011. Proceedings*, pages 160–169, 2011.

[64] Meng He, J. Ian Munro, and Srinivasa Rao Satti. Succinct ordinal trees based on tree covering. *ACM Transactions on Algorithms*, 8(4):42, 2012.

[65] Meng He, J. Ian Munro, and Gelin Zhou. Path queries in weighted trees. In *Algorithms and Computation - 22nd International Symposium, ISAAC 2011, Yokohama, Japan, December 5-8, 2011. Proceedings*, pages 140–149, 2011.

[66] Meng He, J. Ian Munro, and Gelin Zhou. Succinct data structures for path queries. In *Algorithms - ESA 2012 - 20th Annual European Symposium, Ljubljana, Slovenia, September 10-12, 2012. Proceedings*, pages 575–586, 2012.

[67] Meng He, J. Ian Munro, and Gelin Zhou. Dynamic path counting and reporting in linear space. In *Algorithms and Computation - 25th International Symposium, ISAAC 2014, Jeonju, Korea, December 15-17, 2014, Proceedings*, pages 565–577, 2014.

[68] Meng He, J. Ian Munro, and Gelin Zhou. A framework for succinct labeled ordinal trees over large alphabets. *Algorithmica*, 70(4):696–717, 2014.

[69] Guy Jacobson. Space-efficient static trees and graphs. In *30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989*, pages 549–554, 1989.

[70] Joseph JáJá, Christian Worm Mortensen, and Qingmin Shi. Space-efficient and fast algorithms for multidimensional dominance reporting and counting. In *ISAAC*, pages 558–568, 2004.

[71] Allan Grønlund Jørgensen and Kasper Green Larsen. Range selection and median: Tight cell probe lower bounds and adaptive data structures. In *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2011, San Francisco, California, USA, January 23-25, 2011*, pages 805–813, 2011.

[72] Haim Kaplan and Nira Shafrir. Path minima in incremental unrooted trees. In *Algorithms - ESA 2008, 16th Annual European Symposium, Karlsruhe, Germany, September 15-17, 2008. Proceedings*, pages 565–576, 2008.

[73] Valerie King. A simpler minimum spanning tree verification algorithm. *Algorithmica*, 18(2):263–270, 1997.

[74] János Komlós. Linear verification for spanning trees. *Combinatorica*, 5(1):57–65, 1985.

[75] Danny Krizanc, Pat Morin, and Michiel H. M. Smid. Range mode and range median queries on lists and trees. *Nord. J. Comput.*, 12(1):1–17, 2005.

[76] Frank Thomson Leighton. Methods for message routing in parallel machines. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing, May 4-6, 1992, Victoria, British Columbia, Canada*, pages 77–96, 1992.

[77] Giovanni Manzini. An analysis of the burrows-wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.

[78] Peter Bro Miltersen. Lower bounds on the size of selection and rank indexes. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005, Vancouver, British Columbia, Canada, January 23-25, 2005*, pages 11–12, 2005.

[79] J. Ian Munro and Yakov Nekrich. Compressed data structures for dynamic sequences. In *Algorithms - ESA 2015 - 23rd Annual European Symposium, Patras, Greece, September 14-16, 2015, Proceedings*, pages 891–902, 2015.

[80] J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses and static trees. *SIAM J. Comput.*, 31(3):762–776, 2001.

[81] Gonzalo Navarro and Yakov Nekrich. Optimal dynamic sequence representations. *SIAM J. Comput.*, 43(5):1781–1806, 2014.

[82] Gonzalo Navarro and Kunihiko Sadakane. Fully functional static and dynamic succinct trees. *ACM Transactions on Algorithms*, 10(3):16:1–16:39, 2014.

[83] Yakov Nekrich. Orthogonal range searching in linear and almost-linear space. *Comput. Geom.*, 42(4):342–351, 2009.

[84] Gabriel Nivasch. Inverse ackermann without pain. http://www.gabrielnivasch.org/fun/inverse-ackermann.

[85] Manish Patil, Rahul Shah, and Sharma V. Thankachan. Succinct representations of weighted trees supporting path queries. *J. Discrete Algorithms*, 17:103–108, 2012.

[86] Mihai Pătraşcu. Lower bounds for 2-dimensional range counting. In *Proceedings of the 39th Annual ACM Symposium on Theory of Computing, San Diego, California, USA, June 11-13, 2007*, pages 40–46, 2007.

[87] Mihai Pătraşcu. Unifying the landscape of cell-probe lower bounds. *SIAM J. Comput.*, 40(3):827–847, 2011.

[88] Mihai Pătraşcu and Mikkel Thorup. Time-space trade-offs for predecessor search. In *Proceedings of the 38th Annual ACM Symposium on Theory of Computing, Seattle, WA, USA, May 21-23, 2006*, pages 232–240, 2006.

[89] Holger Petersen. Improved bounds for range mode and range median queries. In *SOFSEM 2008: Theory and Practice of Computer Science, 34th Conference on Current Trends in Theory and Practice of Computer Science, Nový Smokovec, Slovakia, January 19-25, 2008, Proceedings*, pages 418–423, 2008.

[90] Holger Petersen and Szymon Grabowski. Range mode and range median queries in constant time and sub-quadratic space. *Inf. Process. Lett.*, 109(4):225–228, 2009.

[91] Seth Pettie. An inverse-ackermann type lower bound for online minimum spanning tree verification. *Combinatorica*, 26(2):207–230, 2006.

[92] Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4), 2007.

[93] Kunihiko Sadakane and Roberto Grossi. Squeezing succinct data structures into entropy bounds. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2006, Miami, Florida, USA, January 22-26, 2006*, pages 1230–1239, 2006.

[94] Daniel Dominic Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983.

[95] Robert Endre Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *J. Comput. Syst. Sci.*, 18(2):110–127, 1979.

[96] Dekel Tsur. Succinct representation of labeled trees. *Theor. Comput. Sci.*, 562:320–329, 2015.

[97] Jean Vuillemin. A unifying look at data structures. *Commun. ACM*, 23(4):229–239, 1980.

[98] Andrew Chi-Chih Yao. Should tables be sorted? (extended abstract). In *19th Annual Symposium on Foundations of Computer Science, Ann Arbor, Michigan, USA, 16-18 October 1978*, pages 22–27, 1978.

[99] Andrew Chi-Chih Yao. Space-time tradeoff for answering range queries (extended abstract). In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing, STOC 1982, San Francisco, California, USA, May 5-7, 1982*, pages 128–136, 1982.

[100] Gelin Zhou. Path queries in weighted trees. Master's thesis, Waterloo, ON, Canada, 2012.