Dipartimento di Informatica
Università di Pisa

New Combinatorial Properties and Algorithms for AVL Trees

Mahdi Amani

Under Supervision of
Linda Pagli, Anna Bernasconi

3 May 2016

Abstract

In this thesis, new properties of AVL trees and a new partitioning of binary search trees named *core partitioning scheme* are discussed, this scheme is applied to three binary search trees namely AVL trees, weight-balanced trees, and plain binary search trees.

We introduce the core partitioning scheme, which maintains a balanced search tree as a dynamic collection of complete balanced binary trees called cores. Using this technique we achieve the same theoretical efficiency of modern cache-oblivious data structures by using classic data structures such as weight-balanced trees or height balanced trees (*e.g.* AVL trees). We preserve the original topology and algorithms of the given balanced search tree using a simple post-processing with guaranteed performance to completely rebuild the changed cores (possibly all of them) after each update. Using our core partitioning scheme, we simultaneously achieve good memory allocation, space-efficient representation, and cache-obliviousness. We also apply this scheme to arbitrary binary search trees which can be unbalanced and we produce a new data structure, called Cache-Oblivious General Balanced Tree (COG-tree).

Using our scheme, searching a key requires $O(\log_B n)$ block transfers and $O(\log n)$ comparisons in the external-memory and in the cache-oblivious model. These complexities are theoretically efficient. Interestingly, the core partition for weight-balanced trees and COG-tree can be maintained with amortized $O(\log_B n)$ block transfers per update, whereas maintaining the core partition for AVL trees requires more than a poly-logarithmic amortized cost.

Studying the properties of these trees also lead us to some other new properties of AVL trees and trees with bounded degree, namely, we present and study *gaps* in AVL trees and we prove Tarjan *et al.* 's conjecture on the number of rotations in a sequence of deletions and insertions.

Keywords: AVL trees, Weight-balanced tree, External-memory model, Cache-oblivious model, Core partitioning scheme, COG-tree, Gap, AVL rotation.

Dedication

"Thesis topics come and go, but the adventurous styling of the journey is forever."

I may not be a religious man, but it is hard to escape the feeling that, underneath all the problems you face in your routine life, there is something of the divine in the solutions you find. I've felt God carrying me when I was stuck and completely lost.

This dissertation is dedicated to him and to my wife who shared her patience and support through the *hard life* of a young researcher.

To the compassionate one and to Eli

Acknowledgments

This work could not have been achieved without the help, support, and encouragement of many people. My heartfelt thanks is owed to many people who made this journey possible. My special thanks and appreciation to Prof. Linda Pagli, Prof. Anna Bernasconi, and Prof. Roberto Grossi not only for their help and encouragement throughout my studies, but also for their compassion and empathetic understanding and touching my life during these years that I had the great honor to work with them. I am grateful as well to Prof. Pierpaolo Degano for coordinating and for all the support that he has given me throughout my academic life in Pisa.

I thank the members of my dissertation committee for their contribution and their good-natured support, I also thank the other members of department of computer science, university of Pisa, for all good memories, help and all the things they taught me. I also thank Robert Tarjan for his time, support and encouragement to prove their conjecture on the number of rotations in AVL trees.

Last but not least, I sincerely and wholeheartedly express my gratitude to all my family members specially my parents and my wonderful wife for their patience and support.

Contents

List of Figures

Chapter 1

Introduction

Trees are one of the most important basic and simple data structures for organizing information in computer science, and have found many applications such as database [75, 57], pattern recognition [57], decision table programming [57], analysis of algorithms [57], string matching [57], switching theory [102], computational geometry [34], image processing [91, 99], and even in the theoretical design of circuits required for VLSI [102]. Trees are also widely used for showing the organization of real world data such family trees, taxonomies, and modeling of the connections between neurons of the brain in computational neuroscience [25, 26].

Many balanced search trees have been designed for their usage in main memory, with optimal asymptotical complexity in terms of CPU time and number of performed comparisons, such as AVL trees [1], red-black trees [17], weight-balanced trees [73], and 2-3 trees [43], just to name the pioneering ones. Unfortunately, they use non linear space and they perform poorly when cache performance is taken into account or large data sets are stored in external memory.

1.1 Memory Hierarchy and Memory Models

In this thesis, we adopt external-memory model [2] and cache-oblivious model [39, 81] to evaluate I/O complexities. The memory hierarchies of modern computers are composed of several levels of memories, that starting from the caches, have increasing access time and capacity. The design of data structures and algorithms must now take care of this situation and try to efficiently amortize the cost of memory accesses by transferring blocks of contiguous data from one level to another. The CPU have access to a relatively small but fast pool of solid-state storage space, *the main memory*; it could also communicate with other, slower but potentially larger storage spaces, *the*

external memory. The memory hierarchies of modern computers are composed of several levels of memories start from *the caches*. Caches have very small access time and capacity comparing to main memory and external memory. From cache to main memory, then to external memory, access time and capacity increase significantly.

In external-memory model [2], the computer has access to a large external memory in which all of the data resides. This memory is divided into memory blocks each containing B words, and B is known. The computer also has limited internal memory on which it can perform computations. Transferring a block between internal memory and external memory takes constant time. Computations performed within the internal memory are free; they take no time at all and that is because of the fact that external memory is so much slower than random access memory [70]. We assume that each external memory access (called an I/O operation or just I/O) transmits one page of B elements.

Traditional databases are designed to reduce the number of disk accesses, since accessing data on the disk is orders of magnitude more expensive than accessing data in main memory. With data sets becoming resident in main memory, the new performance bottleneck is the latency in accessing data from the main memory [49]. Therefore, we also adopt the cache-oblivious model [39, 81] to evaluate the I/O complexity, here called cache complexity. The cache-oblivious model is a simple and elegant model introduced in [39, 81] which allows to consider only a two-level hierarchy, but proves results for a hierarchy composed of an unknown number of levels. In this model, memory has blocks of size B, where B is an *unknown* parameter and a cache-oblivious algorithm is completely *unaware* of the value of B used by the underlying system.

1.2 Main Results

We propose a general method to store the nodes of balanced search trees and obtain provably good space-efficient external-memory/cache-oblivious data structures. The proposed scheme hinges on the decomposition of a balanced search tree into a set of disjoint cores: a *core* is a complete balanced binary tree that appears as a portion of the balanced tree. A core of height h has $2^h - 1$ nodes when the height of a node is the number of nodes on the longest simple downward path from that node to a leaf [57]. Our method is **not** invasive, as it does not change the original algorithms. It just requires a post-processing procedure after each update to maintain the cores. The nodes of a core are stored in a chunk of consecutive memory cells. Hence, the core partition adds a memory layout for the nodes of a balanced tree but does not interfere with the original algorithms for the

tree.

For a given binary search tree T with size n and height H, for a parameter $h^*(T)$ (that depends on the type of the given balanced tree), our recursive scheme requires that the first $h^*(T)$ levels of the nodes in the given balanced tree are full, thus they form a core. It conceptually removes these nodes and applies recursively this process to the resulting bottom subtrees. The recursion ends when the subtree size is below a threshold r^* to be specified, we call such a (possibly empty) terminal subtree, a *terminal-core*. As a result, the given balanced tree is decomposed into cores, which are central to our findings. We call this technique, *core partitioning scheme*, which maintains a balanced search tree as a dynamic collection of complete balanced binary trees (cores).

We obtain a *successful core partition* when the cores found along any root-to-leaf path of the balanced tree are of doubly exponentially decreasing size, with $O(1)$ of them being of size smaller than r^*. We show that for any binary search tree with such a successful core partition, we obtain a space-efficient external-memory/cache-oblivious layout to dynamically maintain the structure and their keys. Using the external-memory/cache-oblivious models [2, 81], it takes $\Theta(n/B)$ blocks of memory of size B to store the keys with extra $O(n)$ bits space needed for the external pointers to the cores and the terminal-cores, note that representing the structure of a balanced binary tree using $O(n)$ bits is also another efficient bound independently achieved by the core partitioning scheme. Searching a key requires $O(\log_B n)$ block transfers and $O(\log n)$ comparisons in the external-memory model, and the amortized cost of update varies with the specifications of the balanced binary tree. As case studies, we apply the core partitioning scheme on weight-balanced trees [73] and AVL trees [1]. Interestingly, the core partition for weight-balanced trees can be maintained with amortized $O(\log_B n)$ block transfers and amortized $O(\log n)$ time complexity per update, whereas maintaining the core partition for AVL trees requires super polylogarithmic amortized cost. We prove this result providing a 'new lower bound' on the subtree size of the rotated nodes in AVL trees.

We present core partitioning scheme as a general approach for making different classic and well-studied balanced binary search trees efficient and applicable in external-memory/cache-oblivious models and compatible to the modern search data structures, thus making our method of independent interest. More precisely, similarly to our case studies, a core partitioning scheme can be applied to other types of balanced binary search trees. For any type of balanced binary search trees, if one can prove that they admit a successful core partition, all of the core partition properties such as external-memory efficiency, cache-obliviousness, linear space, and $O(\log_B n)$ search

cost would be instantly achieved, more importantly, the original structure of that binary search tree will *always* be preserved. However, the update cost varies depending on the class of binary search tree.

An example of the benefit of our technique is that by preserving the original structure of the given binary search tree, we can reuse the vast knowledge on balanced search trees to provide a repertoire of space-efficient external-memory and cache-oblivious data structures which are competitive with modern search data structures that are purposely designed for these models (e.g. [20, 21, 23, 24, 29]). This opens a number of possibilities that are known for modern search data structures but unknown for several previous balanced trees:

- I/O efficiency and cache-obliviousness can be achieved.
- Dynamic memory management can be handled.
- The space is linear; $O(n)$ 'words' to store the keys with an extra $O(n)$ 'bits' for the external pointers to the cores and the terminal-cores (rather than $\Omega(n \log n)$ bits for the external pointers in the link based presentations).
- Search can be performed in $O(\log_B n)$ I/Os and $O(\log n)$ comparisons.

We emphasize that the above features just require the original algorithms described for the given balanced tree, thus offering simultaneously many features that have been introduced later on different search trees. What we add is the maintenance of our structure for the nodes, and the algorithmic challenge is how to maintain it efficiently. When performing the updates, we proceed as usual, except that we perform a post-processing: we take the topmost core that should be changed because of the update, and recompute the partition from it in a greedy fashion.

The notion of *core partition* introduced above shows how to obtain cache-efficient versions of classical balanced binary search trees such as AVL trees and weight-balanced trees. A natural question is whether the core partition can be applied also to arbitrary binary search trees which can be *unbalanced*. We give a positive answer to this question by presenting a data structure, called *Cache-Oblivious General Balanced Tree* (*COG-tree*).

A binary tree is typically kept balanced by storing at each node some information on the structure of the tree and checking at each update that some constraints on the structure of the tree are maintained. This information must be dynamically updated after insertions and deletions. A different approach let the tree assume any shape as long as its height is logarithmic. In this way there is no need of storing and checking the balance information, but it is sufficient to check whether

the maximal possible height has been exceeded. Trees of this kind, called *General Balanced Trees*, introduced by [8] and later rediscovered by [41] under the name of *scapegoat trees*, can be efficiently maintained and require as additional space only that for the pointers. They are restructured with an operation, called *partial rebuilding* that transforms a subtree of the tree in a perfectly balanced tree. The operation is expensive having a cost proportional to the number of nodes of the subtree, but performed rarely hence has a low amortized cost. COG-trees use such partial rebuilding operations with some modifications.

A COG-tree of n nodes has an improved cache complexity of $O(\log_B n)$ amortized block transfers and $O(\log n)$ amortized time for updates, and $O(\log_B n)$ block transfers and $O(\log n)$ time for searches. Same as before, the $O(\log_B n)$ amortized block transfers for update is theoretically efficient. The space occupancy is also linear.

1.3 Other Results on Properties of AVL Trees

Studying the properties of these trees also lead us to some other new properties of AVL trees and trees with bounded degree, namely, we define and study *gaps* and we prove Tarjan *et al.* 's conjecture on the number of rotations in a sequence of deletions and insertions and finally, we generate trees with bounded degree in an specified ordering (A-order).

Gaps in AVL trees are special tree edges such that the height difference between the subtrees rooted at their two endpoints, is equal to 2. Using gaps we prove the *Basic-Theorem* that allows us to express the size of a given AVL tree in terms of the heights of the gaps. The Basic-Theorem can represent any AVL tree (and its subtrees) with a series of powers of 2 of the heights of the gaps. The Basic-Theorem and its corollaries are interesting to characterize the tree size of any AVL tree with a very simple and useful formula. They describe the precise relationship between the size of the tree and the heights of the nodes, also the subtree sizes and the heights of the gaps, and finally they independently describe the relationship between the heights of the nodes and the heights of the gaps. We will also investigate how gaps change (disappear or reappear) in an AVL tree during a sequence of insertions and deletions.

As we know, an insertion in an n-node AVL tree takes at most two rotations, but a deletion in an n-node AVL tree can require $\Theta(\log n)$. A natural question is whether deletions can take many rotations not only in the worst case but in the amortized case as well. A sequence of n successive deletions in an n-node tree takes $O(n)$ rotations [101], but what happens when insertions are intermixed with deletions?

Heaupler, Sen, and Tarjan [48] conjectured that alternating insertions and deletions in an n-node AVL tree can cause each deletion to do $\Omega(\log n)$ rotations, but they provided no construction to justify their claim. We provide such a construction which causes each deletion to do $\Omega(\log n)$ rotations: we show that, for infinitely many n, there is a set E of *expensive* n-node AVL trees with the property that, given any tree in E, deleting a certain leaf and then reinserting it produces a tree in E, with the deletion having performed $\Theta(\log n)$ rotations. One can do an arbitrary number of such expensive deletion-insertion pairs. The difficulty in obtaining such a construction is that, in general, the tree produced by an expensive deletion-insertion pair is not the original tree. Indeed, if the trees in E have odd height h, $2^{\frac{h-1}{2}}$ deletion-insertion pairs are required to reproduce the original tree.

Finally the last result in this thesis is the generation of trees with bounded degree in A-order. Exhaustive generation of certain combinatorial objects has always been of great interest for computer scientists. Designing algorithms to generate combinatorial objects has long fascinated mathematicians and computer scientists as well. Some of the earlier works on the interplay between mathematics and computer science have been devoted to combinatorial algorithms. Because of its many applications in science and engineering, the subject continues to receive much attention.

Studying combinatorial properties of restricted graphs or graphs with configurations has also many applications in various fields such as machine learning and chemoinformatics. Studying combinatorial properties of restricted trees and outerplanar graphs (*e.g.* ordered trees with bounded degree) can be used for many purposes including virtual exploration of chemical universe, reconstruction of molecular structures from their signatures, and the inference of structures of chemical compounds [117, 94, 40, 46, 44, 50, 14]. Therefore, in Chapter 6, we will study the generation of unlabeled ordered trees whose nodes have maximum degree Δ. For the sake of simplicity, we denote such a tree by T^{Δ} *tree*, we also use T_n^{Δ} to denote the class of T^{Δ} trees with n nodes.

Typically, trees are encoded as strings over a given alphabet and then these strings (called codeword) are generated [80]. By choosing a suitable codeword to represent the trees, we can design efficient generation algorithm for these codewords. Any generation algorithm is characterized by the ordering it imposes on the set of objects being generated and by its complexity. The most well-known orderings on trees are A-order and B-order [115] which will be defined in Chapter 2. A-order has been referred to as the most natural ordering on the set of trees. The A-order definition uses global information concerning the tree nodes, where as the B-order definition uses local information. Besides the generation algorithm for trees, ranking and unranking algorithms are also important in

the concept of tree generation [87, 111, 115]. Given a specific order on the set of trees, the rank of a tree (or corresponding sequence) is its position in the exhaustive generated list, and the ranking algorithm computes the rank of a given tree (or corresponding sequence) in this order. The reverse operation of ranking is called unranking; it generates the tree (or sequence) corresponding to a given rank. For this class of trees, besides an efficient algorithm of generation in A-order we present an encoding over 4 letters and size n with two efficient ranking and unranking algorithms. The generation algorithm has $O(n)$ time complexity in the worst case and $O(1)$ in the average case. The ranking and unranking algorithms have $O(n)$ and $O(n \log n)$ time complexity, respectively. The presented ranking and unranking algorithms use a precomputed table of size $O(n^2)$ (assuming Δ is constant).

1.4 Thesis Organization and Overview

In summary, this assertion is organized as follows. Some preliminaries on binary search trees, external-memory model, cache-oblivious model, important search tree data structures, and the concept of exhaustive generation of trees with bounded degree are presented in Chapter 2. In Chapter 3, we propose a general method to store the nodes of balanced search trees (the core partitioning scheme). Then the core partitioning scheme is applied directly to plain binary search trees in Chapter 4. In Chapter 5, we present some new features and properties of AVL trees including the proof of Heaupler, Sen, and Tarjan [48] conjecture that alternating insertions and deletions in an n-node AVL tree can cause each deletion to do $\Omega(\log n)$ rotations. Chapter 6 is dedicated to generation of trees with bounded degree which is a byproduct of our research. Finally, some concluding remarks and suggestions for further research are given in Chapter 7.

Chapter 2

Background

In this chapter, we study some basic concepts of important binary search trees [33, 57, 107, 31, 90], external-memory model [2] and cache-oblivious model [81, 39], the most important external-memory/cache-oblivious data structures [13, 21, 23, 24, 29, 74], and the concept of exhaustive generation of trees with bounded degree [77, 96, 115].

2.1 Trees and Binary Search Trees

Trees are one of the most important basic and simple data structures for organizing information in computer science. Trees have many applications including database generation, decision table programming, analysis of algorithms, string matching [57, 90], switching theory, theoretical VLSI circuit design [102], computational geometry [34], image processing [91, 99], HTML hierarchy structure [16], and maintaining data [75]. Trees are also widely used for showing the organization of real world data such family/geneaology trees [35], taxonomies, and modeling of the connections between dendrites of the brain in computational neuroscience [26]. Also in image processing, particular cases of t-ary trees, quadtrees and octrees, are used for the hierarchical representation of 2 and 3 dimensional images, respectively [91].

There are many notions for trees as well as various notations concerning graphs. We suppose the reader is familiar with basic concept of graph, trees and algorithms. In this section, some definitions and properties of several kinds of trees are presented.

A *rooted tree* is a tree in which one of the nodes is distinguished from the others. The distinguished node is called the *root* of the tree. We often refer to a node of a rooted tree as a node of the tree. In a rooted tree, *degree of a node* is defined as the number of its children and a *leaf* is a

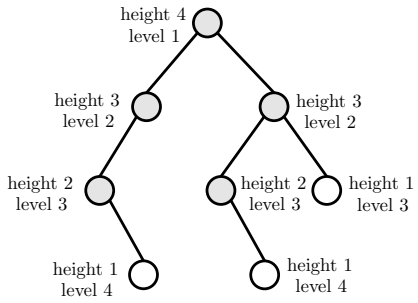


Figure 2.1: An example of the heights and the levels of the nodes in a given tree.

node of degree 0. An *internal node* is a node of degree at least 1. A *labeled tree* is a tree in which each node is given a unique label. The nodes of a labeled tree on n nodes are typically given the labels $1, 2, \ldots, n$.

Consider a node x in a rooted tree T with root r. Any node y on the unique path from r to x is called an *ancestor* of x. If y is an ancestor of x, then x is a *descendant* of y. If y is an ancestor of x and $x \neq y$, then y is a *proper ancestor* of x and x is a *proper descendant* of y. The *subtree* rooted at x is the tree consisting of the descendants of x, rooted at x. The length of the path from the root r to a node x plus one is the *level (depth)* of x in T. The *height of a node* in a tree is the number of nodes on the longest simple downward path from the node to a leaf, and the *height of a tree* is the height of its root [57]. The height of a tree is also equal to the largest level of nodes in the tree. The heights and the levels of the nodes on a tree with height 4 is illustrated in Figure 2.1.

An *ordered tree* or *plane tree* is a rooted tree for which an ordering is specified for the children of each node. This is called a "plane tree" because an ordering of the children is equivalent to an embedding of the tree in the plane, with the root at the top and the children of each node lower than that node. Given an embedding of a rooted tree in the plane, if one fixes a direction of children, say left to right, then an embedding gives an ordering of the children. Conversely, given an ordered tree, and conventionally drawing the root at the top, then the child nodes in an ordered tree can be drawn left-to-right, yielding an essentially unique planar embedding . Figure 2.2 shows an embedding of an ordered rooted tree in the plane with root labeled by 'x', in this figure, the node with gray color are the internal ones and the rest are the leaves.

2.1.1 Isomorphism on rooted trees

Recall that two graphs are isomorphic if there exists a one-to-one correspondence between their node sets which preserves adjacency relations in the graphs. For rooted trees, isomorphism on

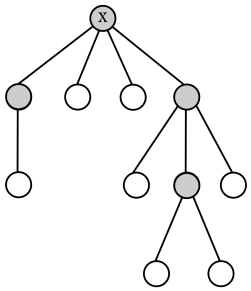


Figure 2.2: An embedding of an ordered rooted tree in the plane on a set of 11 nodes (4 internal nodes and 7 leaves) with root labeled by 'x'.

rooted trees preserves the roots (*i.e.*, roots are mapped to each other) [55]. More precisely, if T and T' are two rooted trees and V, E, r, V', E', r' denote the set of nodes, the set of edges, and the root of T and T', receptively, *isomorphism of rooted trees* T and T' is a bijection between their nodes $f : V \to V'$ such that:

$$f(r) = r' \text{ and } \forall u, v \in V, (u, v) \in E \Leftrightarrow (f(u), f(v)) \in E'.$$

In simple words, two rooted trees are isomorphic if one tree can be obtained from the other by performing any number of *flips* while flip means swapping left and right children of a node. Figure 2.7 shows two isomorphic rooted trees.

2.1.2 Positional Trees

A positional tree is an ordered tree in which the children of a node are labeled with distinct positive integers. The i^{th} child of a node is *absent* if no child is labeled with integer i [33, 107].

t-ary Trees and Binary Trees:

A *t-regular* tree is a rooted tree in which each node has t children. To construct a t-regular tree from a rooted tree, to every node which has $q < t$ children, $t - q$ special nodes are added as its children. These special nodes are called *null pointers (null nodes)*. Clearly, the constructed tree is not unique. An example of a 3-regular is shown in Figure 2.3. A *t-ary* tree is a positional tree in which for every node, all children with labels greater than t are missing. t-ary tree can also be defined as an ordered t-regular tree, in which every internal node has exactly t ordered children (including null pointers). 2-ary trees are also called *binary trees*, while 3-ary trees are sometimes called *ternary trees*. An n-node t-ary tree T is a t-ary tree with n nodes, *i.e.*, $|T| = n$. Clearly, an

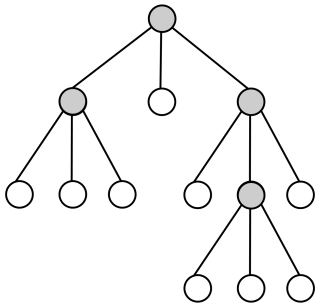


Figure 2.3: An example of a 3-regular tree.

n-node t-ary tree has $(t-1)n+1$ null pointers. The t-ary tree that contains no nodes is called an *empty tree* or *null tree*. Also, a t-ary tree T can be defined recursively as being 'a null pointer' or 'a node together with a sequence $T_1, T_2, \ldots, T_t$ of t-ary trees'. T_i is called a *subtree* of T. So sometime a tree T is shown as $T = T_1, T_2, \ldots, T_t$.

It is well known that **binary trees** with n internal nodes are counted by the n^{th} Catalan number [97]:

$$C_n = \frac{1}{n+1}\binom{2n}{n},$$

and it is also known that the number of **t-ary trees** with n internal nodes is [97, 42]:

$$\frac{1}{tn+1}\binom{tn+1}{n}.$$

Lemma 1 *[107] The maximum number of internal nodes on level i $(i \geq 0)$ of a t-ary tree is t^i.*

A *complete t-ary* tree is a t-ary tree in which all leaves have the same level. For $t = 2$, the t-ary trees are called binary trees, where each node has a *left* and a *right* child. Also, a binary tree is best described recursively. A binary tree T is a structure defined on a finite set of nodes that either:

- contains no node, or
- is composed of three disjoint sets of nodes: a root node, a set of nodes called *left subtree* of T, and a set of nodes called *right subtree* of T, and their roots are called *left child* and *right child* of the root, respectively. Both subtrees are themselves binary trees.

A binary tree is **not** simply an ordered tree in which each node has degree at most 2. For example, in a binary tree, if a node has just one child, the position of the child, whether it is the left child

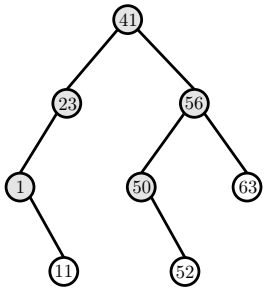


Figure 2.4: In a BST, for every node with key x, the keys in the left (right) subtree are less (greater) than x.

or the right child, matters. In an ordered tree, there is no distinguishing a solo child as being left or right. Sometimes a binary tree T is shown as $T = T_L T_R$, in which T_L is the left subtree and T_R is the right subtree of T. A *full binary tree* is a binary tree in which each node is either a leaf or has degree exactly 2. A *complete balanced binary tree* is a binary tree in which all leaves have the same level. Clearly, a complete balanced binary tree of height h has $2^h - 1$ nodes.

In the following we define binary search trees and we list and introduce some binary search trees which are more important.

2.1.3 Binary Search Trees

Binary search tree (BST) is basically a data structure based on binary trees where each node has a comparable *key* (and an associated *value*) and satisfies the restriction that the key in any node is larger than all the keys in the left subtree and smaller than all the keys in the right subtree (an example of a BST is given in Figure 2.4). This data structure is one of the most common data structure who guarantees to store data in a sorted way. The size of a BST is only limited by the amount of free memory in the operating system. The common properties of binary search trees are as follows.

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtrees are binary search trees.
- Each node can have up to two children.

Generally, the information represented by each node is a record rather than a single data element. However, for sequencing purposes, nodes are compared according to their keys rather

than any part of their associated records. The advantages of binary search trees over other data structures are:

- BST can easily be split for parallel operations.
- BST is mostly fast in search, insertion and deletion operations (depends on how much *balanced* the tree is).
- BSTs are dynamic data structures by nature.
- Its implementation is easier than other data structures.
- BST can find the closest element to some arbitrary key value efficiently. It can also support range queries[1] reasonably fast since it does not search a subtree completely out of the range.

Some of their disadvantages are the followings.

- The shape of the binary search tree totally depends on the order of the insertions, and it can be very *unbalanced*, so that the search operation has the worst case time complexity $O(n)$ (*e.g.* inserting a sorted sequence to an empty tree generates a BST of height n).
- After a long intermixed sequence of random insertion and deletion, the expected height of the tree approaches the square root of the number of keys which grows much faster than $\log n$.

In the decades, researchers have introduced many interesting binary search trees and other tree data structures to keep the tree as balanced as possible, so search, insertion, and deletion operations have the worst case cost $O(\log n)$. In the following we study the most important ones.

AVL Trees

Height-balanced binary trees (hb-trees) have the property that, for every node, the heights of the left and right subtrees differ at most by an integer value Δ [38, 67]. *AVL trees*, the original type of balanced binary search trees were introduced over 50 years ago [1] but still are remarkable for their efficiency. AVL trees are the first family of hb-trees which appeared in the literature, for which $\Delta = 1$. Since the invention of AVL trees in 1962, a wide variety of ways to balanced binary search trees have been proposed. They are mostly based on some particular *rebalancing* algorithms executed after an insertion or a deletion to maintain the tree balanced.

[1] A *range query* is an operation that retrieves all the keys between an upper bound and a lower bound.

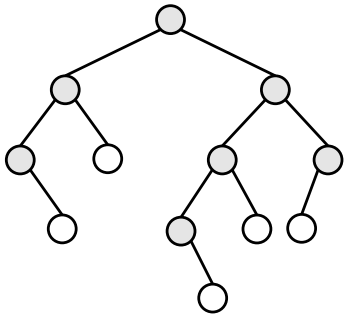


Figure 2.5: An example of AVL tree.

An AVL tree ("Adelson-Velskii and Landis' tree", named after the inventors) is a balanced binary search tree. It was the first data structure of this kind to be invented. In an AVL tree, the heights of the two child subtrees of any node differ by at most one; if at any time they differ by more than one, rebalancing is done to restore this property. Search, insertion, and deletion all take $O(\log n)$ time in both the "average" and "worst cases", where n is the number of nodes in the tree prior to the operation. An example of AVL tree is given in Figure 2.5. Basic operations of an AVL tree involve carrying out the same actions as would be carried out on an unbalanced binary search tree, but modifications (insertions and deletions) are followed by some more operations called *tree rotations*, which help to restore the height balance of the subtrees. Figure 2.6 illustrates a rotation.

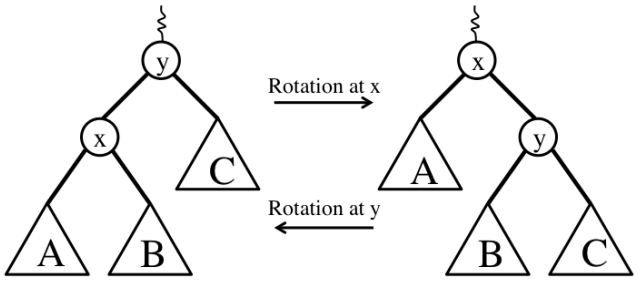


Figure 2.6: Right rotation at node x. Triangles denote subtrees. The inverse operation is a left rotation at y.

The time complexity for the search operation is $O(\log n)$, and the time complexity for the insertion operation is $O(\log n)$ for searching the place where the key must be inserted, plus a constant number of rebalancing operations which take constant time if the tree is maintained by pointer-based data structures. For the deletion operation, the time required is again $O(\log n)$ for search, plus a maximum of $O(\log n)$ rotations on the way back to the root, so the operation can be completed in $O(\log n)$ time. An insertion in an n-node AVL tree takes at most two rotations, but a deletion in an n-node AVL tree can take $\Theta(\log n)$. Heaupler, Sen, and Tarjan [48] conjectured that

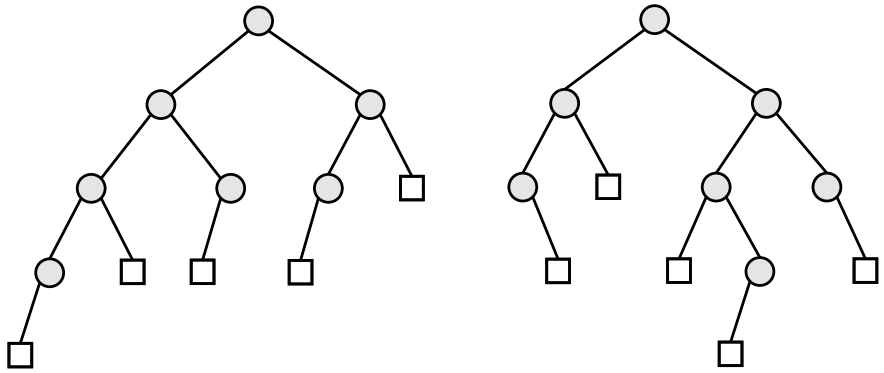


Figure 2.7: A Fibonacci tree of height 5 in the left side and one of its isomorphisms in the right side.

alternating insertions and deletions in an n-node AVL tree can cause each deletion to do $\Omega(\log n)$ rotations, but they provided no construction to justify their claim, and this would be one of our challenges.

Besides AVL trees, many other interesting data structures, such as B-trees [32, 18, 33], red-black trees [17], weight-balanced trees [73], 2-3 trees [43], and (a, b)-trees [54] have been introduced, probably none of them reaching the same appeal. If we look at "vintage" AVL trees with today's eyes, they are indeed pretty modern. The English translation [1] of the Russian paper by Adel′son-Vel′skiĭ and Landis is very close, except some terminology, to the way AVL trees are currently presented in classroom. The rebalancing operations after an insertion are extremely elegant. In the following we study other well known data structures.

Fibonacci Trees *Fibonacci trees* is a beautiful class of binary search trees (see [98]) which represents fully unbalanced AVL trees that in every branch, the height of the left subtree is bigger than the height of the right one. The *Fibonacci tree of height* h has F_h leaves, where F_i shows the i^{th} Fibonacci number (*i.e.*, $F_0 = 0, F_1 = 1, F_i = F_{i-1} + F_{i-2}$). Fibonacci tree is defined recursively as follows [53]. The Fibonacci tree of height h for $h = 0$ is an empty tree and for $h = 1$ is just a single root; If $h \geq 2$, the left subtree of the Fibonacci tree of height h is the Fibonacci tree of height $h - 1$ and the right subtree is the Fibonacci tree of height $h - 2$.

We define a *Fibonacci-isomorphic* tree as an ordered tree which is isomorphic to a Fibonacci tree. Figure 2.7 shows two Fibonacci-isomorphic trees of height 5, the left one is a Fibonacci tree of height 5 and the right one is one of its isomorphisms.

Fact 1 *By definition, a Fibonacci tree of height* h *has* F_h *leaves, also its internal nodes form a*

Fibonacci tree of height $h-1$. Therefore, by a simple induction, the total number of nodes in a Fibonacci tree of height h is $F_h+F_{h-1}+\ldots+F_1$. On the other hand, isomorphic trees have the same number of nodes, hence, for every Fibonacci-isomorphic tree T with height h, $|T| = \sum_{i=1}^{n} F_i = F_{h+2} - 1$.

Note that Fibonacci trees represent the most unbalanced AVL trees and they have $\Theta(\log n)$ height. In [57, p.460] it has been shown that the maximum height of an AVL tree with size n is upper bounded by $\log_{\Phi}(\sqrt{5}(n+2)) - 2 \approx 1.4404 \times \log(n+2) - 0.3277$.

Red-Black Trees

A *red-black* tree is another interesting balanced binary search trees. This structure comes with an extra bit of storage per node which is its color(red or black). Red-black tree remains balanced during a sequence of insertions and deletions by painting each node with one of two colors (these are typically called 'red' and 'black', hence the name of the trees) in such a way that the resulting painted tree satisfies certain properties that don't allow it to become significantly unbalanced. When the tree is modified, the new tree is subsequently rearranged and repainted to restore the coloring properties. The properties are designed in such a way that this rearranging and recoloring can be performed efficiently. Formally, a red-black tree is a binary search tree in which each node has a color (red or black) associated with it with the following properties [17, 43, 33]:

- *Root property*: The root of the red-black tree is black
- *Red property*: The children of a red node are black.
- *Black property*: Every path from a given node to any of its descendant leaves contains the same number of black nodes.

These properties guarantees that the path from the root to the furthest leaf is no more than twice as long as the path from the root to the nearest leaf. The result is that a red-black tree of n internal nodes has height at most $2\log(n+1)$ [33]. The insertion operation for a new node x containing the key k is performed as follows.

- Use BST insertion algorithm to add x to the tree.
- Color the node x red.
- Restore red-black tree properties (if necessary).

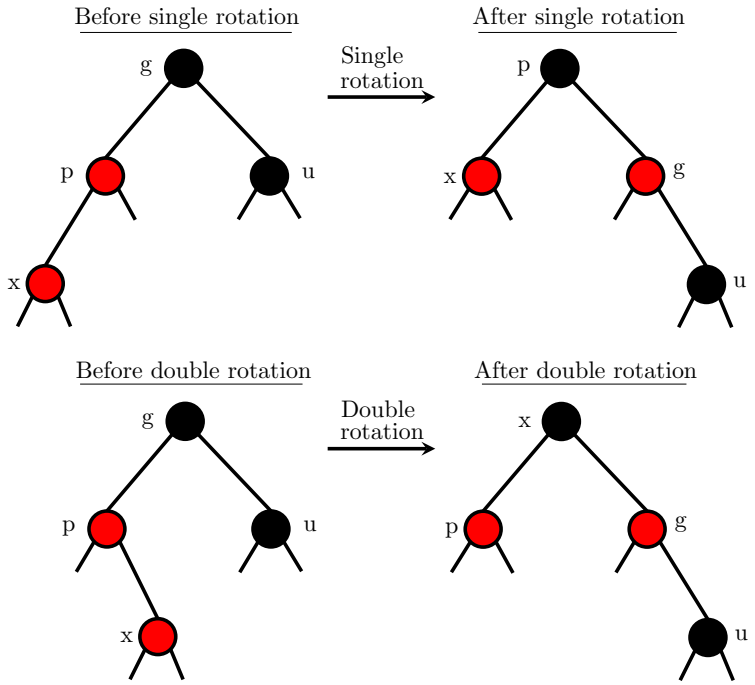


Figure 2.8: Single and double rotations in a red-black tree.

Adding red node x clearly will not violate the black property, however, it may violate the root property or the red property. If that is the case, follow the bellow procedure.

- If adding x violate the root property, just recolor the root from red to black and terminate.
- If adding x violate the red property, let p denote the parent of x. Since the addition of x resulted in the red property violation, p is red. Now let g denote the grand parent of x. g is black because it has a red child (p). Now let u be p's sibling (*i.e.*, the uncle of x). For u, we have the following two cases.
  - If u is black or null, by performing a single or a double rotation as shown in Figure 2.8 the tree is rebalanced and the procedure terminates. Observe that after such a rotation, red property is fixed once again.
  - If u is red, we will do a *recoloring* of p, u, and g as shown in Figure 2.9. Recoloring does not affect the black property of a tree, but, it may violate the red property again (between g and g's parent). If that is the case, then we repeat the entire procedure (recursively handle the red property violation) starting at g and g's parent.

Since the deletion operation is similar to the insertion operation (but with more details), we skip that part. The insertion and deletion operations for red-black trees takes $O(\log n)$ time in

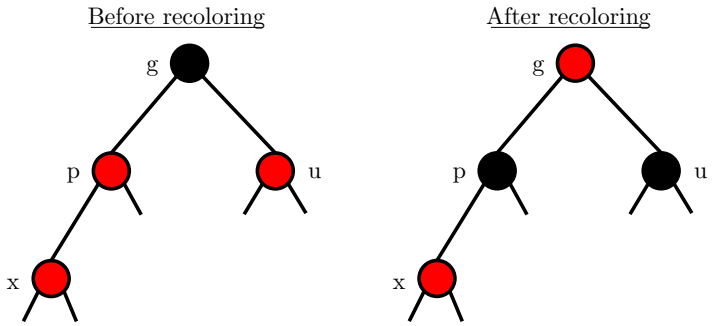


Figure 2.9: Recoloring in a red-black tree.

the worst case [33]. Considering the fact that each black node may have 0, 1 or 2 red children, a red-black tree can be expressed as a B-tree of order 4, where each node can contain between 1 to 3 values and (accordingly) between 2 to 4 child pointers. In such a B-tree, each node will contain only one value matching the value in a black node of the red-black tree, with an optional value before and/or after it in the same node, both matching an equivalent red node of the red-black tree.

Weight-Balanced Trees

In the *weight-balanced tree (WBT)*, the *balance* reflects the relation between the number of nodes in the left and right subtrees and when the balance is disturbed by insertion or deletion operations, rotations are performed to restore it. Specifically, each node stores the size of the subtree rooted at the node, and the sizes of left and right subtrees are kept within some factor of each other. The number of elements in a tree is equal to the size of its root, and the size of the information is exactly the information needed to implement the operations of an order statistic tree. Weight-balanced trees are also called *trees of bounded balance*, or $BB[\alpha]$*-trees* [73].

For a binary tree, the *weight* is the number of null pointers (null nodes), which is equivalent to the number of nodes (*i.e.*, the size) plus one. The weight of a node u is denoted by $w(u)$ and its balance $\beta(u) = w(u.l)/w(u)$ is the ratio between the weight of u's left child and u's weight (note that $w(null) = 1$ by definition of weight) [73].

For a parameter α, where $0 < \alpha \leq 1$, a weight-balanced tree (*a.k.a.* $BB[\alpha]$-tree) is a binary search tree where each node u satisfies $\alpha \leq \beta(u) \leq 1-\alpha$, which is equivalent to say that $\alpha \cdot w(u) \leq w(u.l), w(u.l) \leq (1-\alpha) \cdot w(u)$ for each node u and its two children $u.l$ and $u.r$. Observe that the height of a weight-balanced tree is upper bounded by $\log_{1-\alpha} n = O(\log n)$.

For example, the tree shown in Figure 2.10 is a $BB[\alpha]$-tree for $\alpha = 2/7$ while it is not for

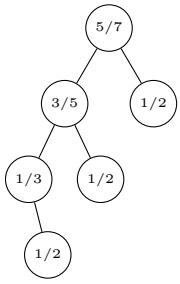


Figure 2.10: $BB[\alpha]$-tree for $\alpha = 2/7$ while it is not for $\alpha = 1/3$.

$\alpha = 1/3$. The value of $\beta(u)$ is reported inside each node u.

As observed by the inventors Nievergelt and Reingold [73], a node of weight 3 should have one child of weight 1, so they assume that $0 < \alpha \leq 1/3$. Moreover, Blum and Mehlhorn [28] show that rebalancing a $BB[\alpha]$-tree with rotations can be done when $2/11 < \alpha \leq 1 - \sqrt{2}/2 = 0.2928\ldots$. When α is strictly inside this interval, they show that there exists $\delta > 0$ depending on α such that an unbalanced node u has balance factor $(1+\delta)\alpha \leq \beta(u) \leq 1 - (1+\delta)\alpha$ after its balance is restored using rotations. Overmars [76, Sect.4.2] shows that rebalancing can be also done with partial rebuilding, and this only requires $0 < \alpha < 1/2$ and obtains a value of $\beta(u)$ close to $1/2$ after restoring the balance of u.

Rank-Balanced Trees

Rank-balanced trees are an extension of AVL trees, where each node x has an integer *rank* $r(x)$ which is proportional to its height. If x is a node with parent $p(x)$, the *rank difference* of x is $r(p(x)) - r(x)$. A node is called an $i - child$ if its rank difference is i, and an $i, j - node$ if its children have rank differences i and j. The initial rank rule is that every node is a $1, 1 - node$ or a $1, 2 - node$. This rule gives exactly the AVL trees. If no deletions occurs, a rank-balanced tree remains an AVL tree; with deletions, $2, 2 - nodes$ will be allowed. The rank and hence the height of a rank-balanced tree is at most $2 \log n$. Considering an initially empty tree and a sequence of m insertions and d deletions ($n = m - d$), it has been shown that the height of the resulting rank-balanced tree is at most $\log_\phi m$ and the total number of rebalancing steps is at most $3m + 6d$, which means $O(1)$ *amortized* rebalancing steps per insertion or deletion. Rank-balanced trees can be rebalanced bottom-up after an insertion or deletion using at most two rotations worst-case [47, 48].

Relaxed AVL Trees (ravls)

Relaxed AVL trees (ravls) are a special class of rank-balanced trees in which the rank difference from child to parent can be non-constant [63, 93]. In this relaxation of AVL trees, rebalancing is done after insertions but not after deletions, however the access time remains logarithmic in the number of insertions. The structure maintains insertion and deletion operations as follows.

To insert a new item into such a tree, use BST insertion algorithm to add its key to the tree. For deletion, first find the item to be deleted (by doing a binary search). If neither child of the item is missing, find either the next item or the previous item, by walking down through left (right) children of the right (left) child of the item, until reaching a node with a missing left (right) child. Then swap the item with the item found. Now the item to be deleted is either a leaf or has one missing child. In the former case, replace it by a missing node; in the latter case, replace it by its non-missing child. If each node has pointers to its children, an access, insertion, or deletion takes $O(h+1)$ time in the worst case, where $h = \log_\phi m$ is the height of the tree and ϕ and m are the golden ratio and the number of insertions, respectively. This structure needs $O(\log\log m)$ bits of balance information per node, or $O(\log\log n)$ with periodic rebuilding, where n is the number of nodes. An insertion takes up to two rotations and constant amortized time.

Randomized Search Trees

Randomized search tree is a data structure for a set X of pairs of *key* and *priority*. Randomized search trees are based on a tree called *treap* which is a rooted binary tree of X that is arranged in *inorder* with respect to the keys and in *heaporder* with respect to the priorities. *Inorder* means that the keys are sorted with respect to inorder traversal and *heaporder* means that the priorities are sorted as a heap (or for any node v the priority of v is greater than priorities of all its ascendants) [92, 11]. In Figure 2.11 an example of treap is shown.

Randomized search trees have an expected cost of $O(\log n)$ for a rotation, when the cost of the rotation is proportional to the subtree size of the rotated node.

Splay Trees

The splay tree presented by Sleator and Tarjan [95] does not require any balance information stored in the nodes. However, the height of a splay tree is not guaranteed to be $O(\log n)$. The logarithmic cost for searching in a splay tree is amortized not worst case. The splay tree is a self-adjusting form of binary search trees. On an n-node splay tree, all the standard search tree operations have

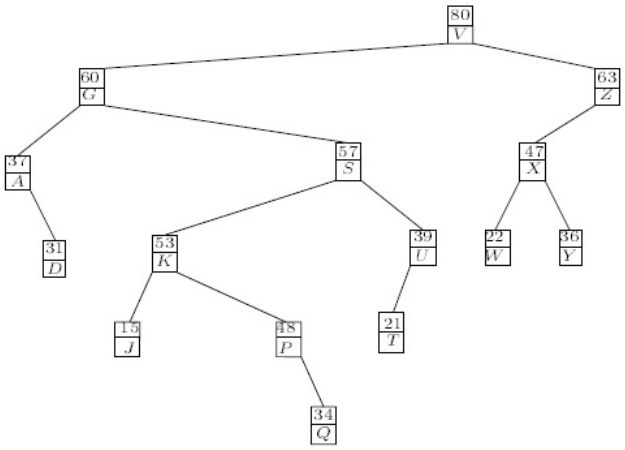


Figure 2.11: An example of Randomized Search Tree, in this picture the numbers are priorities and the alphabets are the keys.

an amortized time bound of $O(\log n)$ per operation. In splay trees a simple heuristic restructuring function called *splaying* is applied whenever the tree is accessed. To *splay* a tree at a node v, repeat the following *splaying step* until v is the root of the tree.

- Case 1 (zig): If $p(v)$ is the tree root, rotate the edge joining v with $p(v)$ (This case terminates splaying the tree).
- Case 2 (zig-zig): If $p(v)$ is not the root and v and $p(v)$ are both left or both right children, rotate the edge joining $p(v)$ with its grandparent $p(p(v))$ and then rotate the edge joining v with $p(v)$.
- Case 3 (zig-zag): If $p(v)$ is not the root and v is a left child and $p(v)$ a right child, or vice versa, rotate the edge joining v with $p(v)$ and then rotate the edge joining v with the new $p(v)$.

Splaying, is similar to move-to-root in that it does rotations bottom-up along the access path and moves the accessed item all the way to the root. But it differs in that it does the rotations in pairs, in an order that depends on the structure of the access path.

General Balanced Trees

Anderson's *general balanced trees* [8] are maintained by *partial rebuilding*, this idea is similar to the scapegoat trees that we will study next and to the technique that we will explain in Chapter 4. For general balanced trees, in order to achieve efficient maintenance of a balanced binary search tree,

no shape restriction other than a logarithmic height is required. The obtained class of trees, general balanced trees, may be maintained at a logarithmic amortized cost with no balance information stored in the nodes (*e.g.* 'colors', 'weights', 'rank', *etc.*). Thus, whenever amortized bounds are sufficient, there is no need for sophisticated balance criteria. The maintenance algorithms use partial rebuilding. The main idea in maintaining a general balanced tree is to let the tree take any shape as long as its height does not exceed $\lceil c \log |T| \rceil$ for some constant $c > 1$. When this criterion is violated, the height can be decreased by partial rebuilding at a low amortized cost. Anderson in [8] proved that the amortized cost incurred by general balanced trees is lower than what has been shown for weight-balanced trees. In general balanced trees, no rebalancing is performed. General balanced trees kept rebalanced using the above partial rebuilding that transforms a subtree of the tree in a perfectly balanced tree. The operation is expensive having a cost proportional to the number of nodes of the subtree, but performed rarely hence has a low amortized cost.

Scapegoat Trees

Scapegoat trees presented by Galperin and Rivest in [41] similarly to general balanced trees use *partial rebuilding* to rebalance and unlike most other balanced-trees, do not require keeping extra data (*e.g.* 'colors', 'weights', 'rank', *etc.*) in the tree nodes. Each node in the tree contains only a key value and pointers to its two children. Associated with the root of the whole tree are the only two extra values needed by the scapegoat scheme: the number of nodes in the whole tree, and the maximum number of nodes in the tree since the tree was last completely rebuilt. In a scapegoat tree a typical rebalancing operation begins at a leaf, and successively examines higher ancestors until a node (*the scapegoat*) is found that is so unbalanced that the entire subtree rooted at the scapegoat can be rebuilt at zero cost, in an amortized sense. Scapegoat trees provides worst-case $O(\log n)$ search time, and $O(\log n)$ amortized insertion and deletion time.

2.1.4 B-tree, 2-3 Tree and (a,b)-trees

B-tree can be considered as a generalization of a binary search tree in which a node can have more than two children and the structure remains always balanced [32]. In B-trees, internal nodes can have any number of children within some pre-defined range. For example, *2-3 trees* are B-trees which any internal node can have only 2 or 3 children [43] and (a, b)-*tree* is a B-tree where each node has at least a and at most b children and $a \leq b/2$ [54]. When data is inserted or removed from a node, its number of child nodes changes. In order to maintain the pre-defined range, internal nodes

may be joined or split. In B-tree nodes are not entirely full so there might be a waste of space, also the structure will be always balanced and it guarantees logarithmic time complexity in both worst case and average case for search, insertion and deletion operations. The range of possible number of children will be optimized regarding to hardware specification to make it practically fast for systems that read and write large blocks of data. This data structures is the oldest data structure with applications external-memory model (which will be introduced later). More information can be found in almost all text books of algorithms and data structures, for example [18, 33, 57].

2.1.5 Tree Representation

To represent a t-ary tree in a computer, the most common but non efficient way is *linked representation*. In this representation, each internal node of tree will have $t + 1$ fields: one *data* and t children fields. Data field is used for holding data (or label) of a node and i^{th} child field points to i^{th} subtree of node. For binary trees, we have two pointer fields in each node, called left child and right child fields. In this representation, no memory is needed for null pointers (null nodes) and all pointers to empty trees are *null*.

The other method of tree representation is when a tree is represented by integer or alphabet sequences. This operation is called *tree encoding*. Basically, the uniqueness of encoding, the length of the encoding, and the capability of constructing the tree from its representation, which is called *decoding*, are essential considerations in the design of the tree encoding schema [68].

2.1.6 Tree Traversal

There are many operations that may be performed on trees. One notion that arises frequently is the idea of *traversing* a tree or *visit* each node in a tree exactly once. A full traversal produces a linear order for the information in a tree. Here, first we define the traversal operations for binary trees and then extend some of them to t-ary trees.

In a binary tree, if we assume that L, V , and R stand for moving left, visiting the node, and moving right, respectively, and if we adopt the convention that we traverse left before right, then the only three traversals will be: LVR, LRV , and VLR. To these traversal types we assign the names *inorder*, *postorder*, and *preorder* respectively. The earliest algorithms represented for tree traversals which mainly use stacks, can be easily written in recursive form. Recursive algorithms for inorder, preorder, and postorder traversals are similar, only the position of visiting the nodes differ due to the corresponding traversal. The inorder, postorder, and preorder traversal algorithms are

```
Procedure InOrder(Current: TreePtr)
begin
    if (Current ≠ NULL) then begin
        InOrder(Current.LeftChild);
        Visit(Current.Data);
        InOrder(Current.RightChild);
    end;
end;
```

Figure 2.12: An inorder traversal algorithm for binary trees.

```
Procedure PostOrder(Current: TreePtr)
begin
    if (Current ≠ NULL) then begin
        PostOrder(Current.LeftChild);
        PostOrder(Current.RightChild);
        Visit(Current.Data);
    end;
end;
```

Figure 2.13: A postorder traversal algorithm for binary trees.

```
Procedure PreOrder(Current: TreePtr)
begin
    if (Current ≠ NULL) then begin
        Visit(Current.Data);
        PreOrder(Current.LeftChild);
        PreOrder(Current.RightChild);
    end;
end;
```

Figure 2.14: A preorder traversal algorithm for binary trees.

presented in Figures 2.12, 2.13 and 2.14, respectively. By using inorder traversal in binary search trees, we are able to list the keys ordered (sorted).

The preorder and postorder traversal can be extended and used for any class of trees, *e.g.*, for t-ary trees in preorder traversal, at first, we visit data field of a node and then traverse the t subtrees of this node one by one. The same procedure can be applied to AVL trees, trees with bounded degree, *etc.*

2.2 External-Memory & Cache-Oblivious Memory Models

In this thesis, we adopt both external-memory model [2] and cache-oblivious model [39, 81] to evaluate I/O complexities. The basic computer systems use a memory hierarchy, the CPU has access to a relatively small but fast pool of solid-state storage space, *the main memory*; it could also communicate with other, slower but potentially larger storage spaces, the *external memory*.

The memory hierarchies of modern computers are composed of several levels of memories starting from the *caches*. Caches have very small access time and capacity comparing to main memory and external memory. From cache to main memory, then to external memory, access time and capacity increases significantly.

Since accessing data in main memory is *expensive* relative to the processor speeds, modern processors make use of processor caches. A processor *cache* is a block of low-latency memory that sits between the processor and main memory, and stores the contents of the most recently accessed memory addresses. Latency in retrieving data from the cache is one to two orders of magnitude smaller than the latency in retrieving data from the main memory [39, 81, 49]. In the following we study external-memory model and cache-oblivious model which describe different layers of memory hierarchy.

2.2.1 External-Memory Model

Accessing an item from external storage is extremely slow. In 2013, as mentioned in [70], the average access time of hard disks was 160000 times slower than random access memories (RAMs) and the average access time of solid state drives was 2500 times slower than random access memory (RAM). These speeds are fairly typical; accessing a random byte from RAM is thousands of times faster than accessing a random byte from a hard disk or solid-state drive. Access time, however, does not tell the whole story. When we access a byte from a hard disk or solid state disk, an entire block of the disk is read.

This is the idea behind the external-memory model of computation, illustrated schematically in Figure 2.15. In this model, the computer has access to a large external memory in which all of the data resides. This memory is divided into memory blocks each containing B words. The computer also has limited internal memory on which it can perform computations. Transferring a block between internal memory and external memory takes constant time. Computations performed within the internal memory are free; they take no time at all. The fact that internal memory computations are free may seem a bit strange, but it simply emphasizes the fact that external memory is so much slower than RAM [70]. We assume that each external memory access (called an I/O operation or just I/O) transmits one page of B elements. We measure the efficiency of an algorithm in terms of the number of I/Os it performs and the number of disk blocks it uses.

External memory data structures have been developed for a wide range of applications, including spatial, temporal, and object oriented databases and geographic information systems [13].

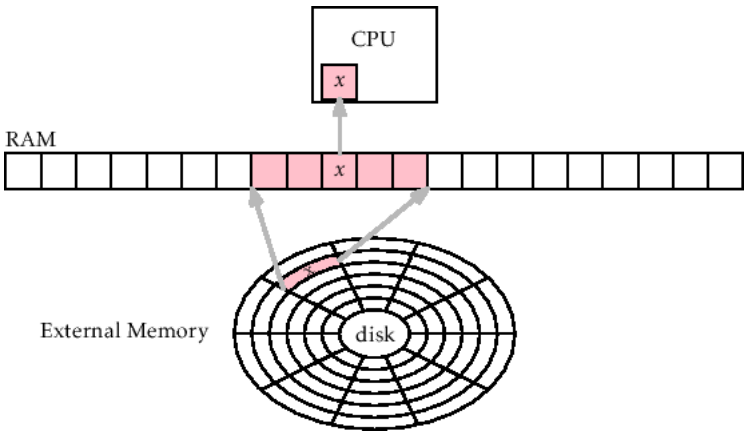


Figure 2.15: The external-memory model.

2.2.2 Cache-Oblivious Memory Model

The memory hierarchies of modern computers are composed of several levels of memories, that starting from the caches, have increasing access time and capacity. Most of today's processor architectures use a hierarchical memory system: a number of caches are placed between the processor and the main memory. Caching has become an increasingly important factor in the practical performance of main-memory data structures. Processor speeds have increased faster than memory speeds, and many applications that previously needed to read data from disk can now fit all of the necessary data in main memory. The relative importance of caching will likely increase in the future [49, 85, 89]. The cache-oblivious model introduced by [39, 81] allows to consider only a two-level hierarchy, but proves results for a hierarchy composed of an unknown number of levels. Cache-oblivious model helps to evaluate the I/O complexity, here called cache complexity and still expressed as number of block transfers of size B. Note that B is now an *unknown* parameter for the block size and a cache-oblivious algorithm is completely unaware of the value of B used by the underlying system.

This model is composed of two parts: the *ideal-cache model* and *cache-oblivious algorithms*. The ideal-cache model has two levels of memory: cache and main memory. The cache contains M locations partitioned into blocks of B contiguous locations each. The main memory can be arbitrarily large. The processing unit can address the locations of the main memory but only the data in cache can be used. If the data needed by the computation is not in cache, a *cache fault (cache miss)* is caused and the corresponding block is transferred from the main memory. The number of processor cache faults has a critical impact on the performance of the system. The

goal is to improve performance by reducing the number of processor cache faults that are incurred during a search operation [49]. When the cache is full, an optimal off-line replacing strategy is used to replace a block with the new one. The cache is fully associative: each block from main memory can be stored anywhere in the cache. An algorithm operating in the ideal-cache model cannot directly manage the transfers of blocks.

There are two types of cache-conscious algorithms; namely, *cache-sensitive (or cache-aware) algorithm*, where the parameters of the caches are assumed to be known to the implementation (*i.e.*, it is another name for external-memory model algorithms) and in contrast, *cache-oblivious algorithms* that attempt to optimize themselves to an unknown memory hierarchy.

In this thesis, we focus on "cache-oblivious model", because the cache-sensitive model is covered by our results in the external-memory model. An algorithm is cache-oblivious if it cannot explicitly use the parameters that are specific to the given memory hierarchy. If the algorithm operates in the ideal-cache model, it cannot be defined in terms of parameters B and M. Being cache-oblivious is an algorithm's strength: since the cache complexity analysis holds for any value of B and M, it holds for any level of a more general, multi-level memory hierarchy, as shown in [39]. The cache-oblivious model can be seen as a "successor" of the RAM model, a successor that incorporates a lot of the new architectural aspects which characterize the real world computing systems in a more refined way.

2.3 Data Structures for External-Memory & Cache-Oblivious Memory Models

Here we study the most important data structures designed for external-memory model or cache-oblivious memory model. These data structures are more complicated than the classic data structures but work more efficiently with real world computers. Recall that for data structures which are comparison based, the optimum bound for search is $O(\log n)$ and for amortized update cost is $O(\log n)$.

T-Trees have been proposed as a better index structure in external memory and main memory database systems. A T-Tree is a balanced binary tree with many elements in a node. Elements in a node contain adjacent key values and are stored in order. Its aim is to balance the space overhead with searching time and cache behavior is not considered [64]. T-Trees put more keys in each node and give the impression of being cache conscious. But if we think of it carefully, we can observe that

for most of the T-Tree nodes, only the two end keys are actually used for comparison. This means that the utilization of each node is low. Since the number of key comparisons is still the same, T-Trees do not provide any better external memory or cache behavior than binary search [84].

Kanellakis *et al.* [56] developed a specific tree[2] occupying optimal $O(n/B)$ blocks in the external-memory model. The structure supports insertions only in $O(log_B n + (log_B^2 n)/B)$ I/Os amortized. A simpler static structure with the same bounds was described by Ramaswamy in [83]. The simplest external-memory model variant of the B-tree is an ordinary *B+-trees* where the node size is chosen to match the size of a block [84]. In B+-trees, in each internal node we store keys and child pointers, but the record pointers are stored on leaf nodes only. Multiple keys are used to search within a node. If we fit each node in a cache line, this means that a cache load can satisfy more than one comparison. So each cache line has a better utilization ratio.

A more advanced version of B+-tree called the Cache-Sensitive B+-tree or CSB+-tree [85] additionally removes pointers from internal nodes by storing the children of a node consecutively in memory. The CSB+-tree has been further optimized using a variety of techniques, such as prefetching, storing only partial keys in nodes, and choosing the node size more carefully [89].

The buffer tree presented in [12] is a well-known example of a general technique for external memory with I/O efficiently. The main idea in the technique is to perform operations on an external (high fanout[3]) tree data structure in a *lazy manner* using main-memory-sized *buffers* associated with internal nodes of the tree. As an example, imagine we are working on a height $O(\log_m n)$ search tree structure with elements stored in the leaves, that is, a structure with fanout $\Theta(m)$ internal nodes and N elements stored in sorted order in n leaves with $\Theta(B)$ elements each, then assign buffers of size $\Theta(m)$ blocks to each of the $O(n/m)$ internal nodes of the structure. When we want to insert a new element, we do not search down the tree for the relevant leaf right away. Instead, we wait until we have collected a block of insertions (or other operations), and then we insert this block into the buffer of the root. When a buffer "runs full" the elements in the buffer are "pushed" one level down to buffers on the next level (this is named *buffer-emptying process*). Deletions or other and perhaps more complicated updates, as well as queries, are basically performed in the same way. Note that as a result of the laziness, we can have several insertions and deletions of the same element in the tree at the same time, and we therefore "time stamp" elements when they are inserted in the root buffer. The laziness also means that queries are batched, since

[2] This data structure also supports diagonal corner queries in $O(log_B n + k/B)$ I/Os (k is the number of reported keys), a *diagonal corner query* is a two sided range query whose corner must lie on the line $x = y$ and whose query region is the quarter plane above and to the left of the corner.

[3] *Fanout* refers to the number of children for an internal node. High fanout means to have more children per internal node.

a query result may be generated (and reported) lazily by several buffer-emptying processes.

The Arge and Vitter weight-balanced B-tree [13] was also presented for external-memory model. The structure uses $O(N/B)$ disk blocks to maintain a set of N intervals such that insertions and deletions can be performed in $O(log_B N)$ I/Os and such that stabbing queries can be answered in $O(log_B N + T/B)$ I/Os, where T denotes the number of points reported.

For cache-oblivious data structures, typical cache optimization techniques include *clustering*, *compression* and *coloring* [84]. Clustering tries to pack, in a cache block, data structure elements that are likely to be accessed successively. Compression tries to remove irrelevant data, thus increases cache block utilization by being able to put more useful elements in a cache block. Coloring maps contemporaneously-accessed elements to non-conflicting regions of the cache [84]. Ladner [62, 61] considered the effects of caches on sorting algorithms and improved performance by restructuring these algorithms to exploit caches. In addition, they constructed a cache-conscious heap structure that clustered and aligned heap elements to cache blocks.

A cache-oblivious layout scheme for fixed-topology trees has been introduced in [22] but it is an open problem to extend it to dynamic trees. Based on this scheme, a new indexing technique called *Cache-Sensitive Search Trees (CSS-trees)* was presented in [89]. The main idea of this technique is to store a directory structure on top of a sorted array. The directory represents a balanced search tree stored itself as an array. Nodes in this search tree are designed to have size matching the cache size of the machine. Therefore, it performs a top-down layout of balanced trees. The partition described in [89] works for any dynamic tree but the pointers are used internally to fix the so-called broken nodes. The authors of [89] report some experimental study to show improvements over traditional trees in practice, but no analysis with provably logarithmic bounds is given for the updates.

The *van Emde Boas (vEB) layout* [81, 29] has many applications in the design of cache-oblivious algorithms including ours (in our scheme will apply vEB layout inside each core for the *unknown* block size B of the cache, see Section 3.2). The vEB layout is a 'static' cache-oblivious data structure that can compactly store an array of a power of two elements, without using any pointers. Given a search tree, where each node has $O(1)$ children, vEB layout describes a mapping from the nodes of the tree to their positions in the memory. Assuming the search tree has height $\Theta(\log n)$, this structure performs search operation in $\Theta(\log_{B+1} n)$ I/O transfers, which is optimal within a constant factor. The basic idea of vEB layout is as follows. Suppose the tree has height h which is a power of two. Conceptually split the tree at the middle level of edges, between nodes of

height $h/2$ and $h/2+1$. This breaks the tree into the top recursive subtree A of height $h/2$, and several bottom recursive subtrees $B_1, B_2, \ldots, B_k$ of height $h/2$. In particular for complete balanced binary trees (all internal nodes have 2 children), the recursive subtrees have size $\sqrt{(n+1)}-1$, and $k = \sqrt{(n+1)}$. We say that these two values are roughly $\sqrt{n}$. The layout of the tree is obtained by recursively laying out each subtree, and combining these layouts in the order $A, B_1, B_2, \ldots, B_k$.

Binary Trees of Small Height [29] were presented by Brodal *et al.* , this cache-oblivious search tree makes use of the 'fast updating of well-balanced trees' [9] implemented by an implicit version of the *van Emde Boas* layout [29, 81]. For a tree of n nodes and block size B, this structure requires $(1+\epsilon)n$ space and performs search operation in the worst case $O(\log_B n)$ block transfers and updates in $O(\log_B{}^2 n/\epsilon B)$ amortized number of block transfers. This structure also allows range queries in $O(\log_B n + k/B)$ block transfers in the worst case, where k is the output size.

The following papers [21, 20, 23] reached optimal bounds and introduced several new data structures on the field.

Bender *et al.* in [20, 23] also presented *Cache-Oblivious B-Trees* in three levels. The top level is a weight-balanced B-tree on $\Theta(n/\log^2 n)$ elements stored according to a vEB layout in a *packed-memory array* (a packed-memory array is an structure for maintaining an ordered collection of n items in an array of size $O(n)$ with the update cost of $O(1+\log_B^2 n)$ amortized memory transfers). The middle level is a collection of $\Theta(n/\log^2 n)$ groups of $\Theta(\log n)$ elements each implemented by a single packed-memory structure, where the representative elements serve as markers between groups. The bottom level is a collection of $\Theta(n/\log n)$ groups of $\Theta(\log n)$ elements each implemented by a packed-memory array if the range query operation is required. Otherwise, the bottom layer is implemented by an unordered collection of groups, where the elements in each group are stored in an arbitrary order within a contiguous region of memory. The update amortized cost of the presented trees are $\Theta(1+\log_{1+B} n)$ when range query is not required or $\Theta(1+\log_{1+B} n+\frac{\log^2 n}{B})$ when search query is required. The space is cn words for a constant $c > 1$.

Bender *et al.* in [21] presented a cache-oblivious data structure called the *exponential structures* for dynamic searching. An *exponential tree* is a tree of $O(\log\log n)$ levels where the degrees of nodes descending from the root level decrease doubly exponentially, *e.g.* as in the series $n^{1/2}, n^{1/4}, n^{1/8}, \ldots, 2$. In the *exponential structure*, internal nodes may have many children and they are called *fat nodes*. The *layer* of a fat node is the number of fat nodes below (*i.e.*, leave fat nodes have level 0). The number of keys stored in a layer i fat node, $i \geq 1$, is in the range $[2^{2^i} - 2^{2^{i-1}}, 2\times 2^{2^i})$, except for the topmost fat node, where the range is given by $[2\times 2^{2^{k-1}}, 2\times 2^{2^k})$,

and k is the layer of the tree. Each layer 0 fat node contains a single item. Loosely speaking, the volumes of the fat nodes square at each successive layer. When updating, if a layer i fat node V acquires $2 \times 2^{2^i}$, it splits as evenly as possible into two subtrees V_1 and V_2 in time $O(|V|)$. When V splits, this adds one to the number of its parent's children. This is accommodated by completely rebuilding the parent. In splitting layer i fat node V into subtrees V_1 and V_2, besides creating V_1 and V_2, all of V's descendant fat nodes are copied into either the portion of the array being used for V_1 and its descendants (or that being used for V_2 and its descendants). Thus, exponential trees achieve search time $O(\log_B n)$ I/Os but increase the space significantly (to $O(n \log^2 n)$). Then, by using buckets of size $\Theta(\log^2 n)$, implemented as two layers of records of size in the range $[\log n, 2 \log n)$, the space can be reduced to $O(n)$ words.

2.4 Exhaustive Generation of Trees with Bounded Degree

The last result in this thesis is the generation of trees with bounded degree in A-order. Therefore, in this section, the basic consideration of the tree generation and the concept of the encoding are discussed, then we introduce the class of trees with bounded degree.

Exhaustive generation of certain combinatorial objects has always been of great interest for computer scientists [77, 96, 115]. In general, the generation of a combinatorial structure problem consists in constructing all possible combinatorial structures of a particular kind in a certain order [60]. For example, a list of all the trees with a given number of nodes n, may be used to test, analyze the complexity, prove the correctness of an algorithm, or for data compression in data communication.

Designing algorithms to generate combinatorial objects has long fascinated mathematicians and computer scientists as well. Some of the earlier works on the interplay between mathematics and computer science have been devoted to combinatorial algorithms. Because of its many applications in science and engineering, the subject continues to receive much attention. In general term, this branch of computer science can be defined as follows. Given a combinatorial object, design an efficient algorithm for generating all the instances of the object. These combinatorial objects could be anything such as graphs, trees, parentheses strings, permutations, combinations, partitions, derangements, *etc.*

Because of the importance of the trees, it is natural to study their properties, and as a result of the existence of numerous applications of trees, algorithms for the generation of the trees have been extensively studied, and many ingenious generation algorithms, for performing this task, have

been discovered [1, 37, 58, 60, 66, 78, 79, 105, 87, 88, 100, 103, 104, 111, 113, 116].

2.4.1 Generation Preliminaries

In most of the trees generation algorithms, a tree is represented by integer or alphabet sequences, and then all possible sequences of this representation are generated. This operation is called *tree encoding*. Basically, the uniqueness of encoding, the length of the encoding, and the capability of constructing the tree from its representation, which is called *decoding*, are essential considerations in the design of the tree encoding schema [80]. By choosing a suitable codeword to represent the trees, we can design an efficient generation algorithm for these codewords.

It is particularly impressive to note that the variation of representations of trees that are possible, such as the *bit strings* [82, 115], the *weight sequences* [77], the *P-sequences* [80], the ℓ-*sequences* [80], the *Ballot sequences* [86], the *Z-sequences* [115], *etc.* In all cases, a one-to-one correspondence is established between the set of trees and the set of certain integer or alphabet sequences; then the set of trees is generated by generating the set of the corresponding sequences.

A-order and B-order

Any generation algorithm is characterized by the ordering it imposes on the set of objects being generated and by its complexity. The most well-known orderings on trees are A-order and B-order [115]. The A-order definition uses global information concerning the tree nodes and appear to be a natural ordering of trees, whereas the B-order definition uses local information. Trees are prominently generated in local order, though natural order and other less useful orders have been addressed to a lesser extent. Up to the present time, the well known tree generation algorithms have utilized B-order, or some other ones, and only a few of them have used A-order. This is perhaps not so surprising if one notes that the generation of trees in A-order is indeed a very difficult task. Here we illustrate these orderings. Let $\prec_A$ and $\prec_B$ denote the A-order and B-order orderings, respectively. Let $\mathbb{T}_n$ be an arbitrary class of trees of size n. For $T, T' \in \mathbb{T}_n$, the most commonly used linear orderings of trees may be defined as follows [104, 103, 115].

Definition 1 *Let T and T' be two ordered trees in $\mathbb{T}_n$, T_i and T'_i show the i^{th} subtrees of T and T', respectively, and $k = max\{deg(T), deg(T')\}$. If $T = T'$, they have the same order, otherwise, we say that T is less than T' in A-order ($T \prec_A T'$), iff*

- $|T| < |T'|$, *or*

- $|T| = |T'|$ *and for some* $1 \leq i \leq k$, $T_j = T'_j$ *for all* $j = 1, 2, \ldots, i-1$ *and* $T_i \prec_A T'_i$,

where $|T|$ *(size of* T*) is usually defined as the number of nodes in the tree* T *and* $deg(T)$ *is defined as the degree of the root of the tree* T*.*

A-order is considered to be the most natural ordering on $\mathbb{T}_n$. From the above definition, it is obvious that the natural order takes into account the size of a tree and hence a global knowledge of trees is compared. This is precisely what makes the generation of most of the trees in the natural ordering non-trivial.

Definition 2 *Let* T *and* T' *be two ordered trees in* $\mathbb{T}_n$, T_i *and* T'_i *show the* i^{th} *subtrees of* T *and* T'*, respectively, and* $k = max\{deg(T), deg(T')\}$*. If* $T = T'$*, they have the same order, otherwise, we say that* T *is less than* T' *in B-order* ($T \prec_B T'$)*, iff*

- $deg(T) < deg(T')$*, or*
- $deg(T) = deg(T')$ *and for some* $1 \leq i \leq k$, $T_j = T'_j$ *for all* $j = 1, 2, \ldots, i-1$*, and* $T_i \prec_B T'_i$*.*

B-order is referred to as *local order*, because in this ordering, we compare the characteristics of the concurrent nodes (whether they are internal nodes or leaves). In other words, it takes a local view of the trees being compared, and the task is easier. This explains why the generation of some trees, such as binary trees or t-ary trees in a local ordering, is popular. One of the advantages for listing trees in the natural order is that the trees of small sizes are listed before the trees of larger sizes. However, no such an advantage is observed in the local order. Furthermore, let $T, T' \in \mathbb{T}_n$; it is possible to have $T \prec_A T'$ and, at the same time, $T' \prec_B T$. Hence, in general, the natural order and the local order list the trees in different orderings.

Tree Encoding

It is well understood that algorithms for generating trees directly (linked form) are complicated and inefficient due to the need of changing the shape of tree [96]. It is indeed easier to manipulate an alphabet sequence which represent a class of trees, and process alphabet sequences instead of that class of trees as explained in [68]. In this way, trees are encoded as strings over a given alphabet and then these strings (called codeword) are generated. By choosing a suitable codeword to represent the trees, we can design an efficient generation algorithm for these codewords. Here, we explain the primaries of tree encoding on a arbitrary class of trees of size n, named $\mathbb{T}_n$.

In general, an *alphabet sequence* can be defined as follows. Let $\mathbb{S}$ be the set of possible strings on an alphabet set $\sum = \{\delta_1, \delta_2, \ldots, \delta_r\}$, *i.e.*, $\mathbb{S} = \{s | s \in \sum^*\}$, and $\mathbb{S}_n$ is the subset of S with all strings of length n, *i.e.*, $\mathbb{S}_n = \{s | s \in \mathbb{S} \text{ and } |s| = n\}$. If string A belongs to $\mathbb{S}_n$, then A is shown as $A = (a_1, a_2, \ldots, a_n)$, such that each $a_i \in \sum$.

For defining an alphabet sequence corresponding to a tree $T \in \mathbb{T}_n$, the most common procedure is as follows. First an alphabet set $\sum$ (letter or integer) is considered and each node of the tree is labeled with an element of $\sum$ with regard to a specific rule (notice that we speak about labeling only for notational convenience; it is naturally possible to distinguish internal node from external ones without having any label), then the tree is traversed with one traversal procedure (preorder, inorder, or postorder) and each node label is listed in this traversal. The resulting sequence is the corresponding sequence of tree with length n. This function is called *tree encoding* and the sequence generated by it is called *codeword*, or *code sequence*, or *tree sequence*, or simply *encoding*[4]. Let $\sum$ and $\mathbb{S}_n$ be defined as above, then the encoding function is a bijection:

$$encoding : \mathbb{T}_n \rightarrow \mathbb{S}_n.$$

The inverse function of encoding is called *decoding*, and by employing it, we can obtain a tree $T \in \mathbb{T}_n$ corresponding to each code sequence. This function is also a bijection:

$$decoding : \mathbb{S}_n \rightarrow \mathbb{T}_n.$$

A tree sequence $A \in \mathbb{S}_n$ will be called *feasible* if there is a tree $T \in \mathbb{T}_n$ such that $A = encoding(T)$.

In fact, in the encoding or decoding processes, we established a one-to-one correspondence between $\mathbb{T}_n$ trees and tree sequences. Once the correspondence is established, an algorithm can be presented to generate all tree sequences. It should be noted that we can also define an ordering for the set of code sequences $\mathbb{S}_n$. Two such ordering are *lexicographic ordering* and *minimal change ordering* [115, 87]. For two strings $A = (a_1, a_2, \ldots, a_n)$ and $B = (b_1, b_2, \ldots, b_n)$, with A and $B \in \mathbb{S}_n$, the lexicographic ordering (lexicographical order) $\prec_{lex}$ or $\prec_{\ell}$ on $\mathbb{S}_n$ is defined for A and B by the following relation:

$$A \prec_{lex} B \quad \Leftrightarrow \quad \exists j \ (1 \leq j \leq n) \text{ such that } a_1 = b_1, a_2 = b_2, \ldots, a_{j-1} = b_{j-1} \text{ and } a_j < b_j.$$

[4]Note that in general, not necessarily in all the tree encodings, all the nodes are labeled (*e.g.* one may omit the root or the leaves for some trees), but there must exist a one-to-one correspondence between the codewords and the trees, however, for sake of simplicity, we assume the length of the codeword is n.

Ranking and Unranking Algorithms

Besides the generation algorithm for trees, ranking and unranking algorithms are also important in the concept of tree generation [87, 111, 115]. Let us consider an arbitrary class of trees of size n (n nodes) showed by $\mathbb{T}_n$, the elements of this set can be listed based on any defined ordering such as *A-order* or *B-order*. By having $\mathbb{T}_n$ and an ordering, the "position" of tree T in $\mathbb{T}_n$ is called *rank*, the *rank function* determines the rank of T; the inverse operation of ranking is *unranking*, for a position r, the *unrank function* gives the tree T corresponding to this position.

Recall that, the rank function determines the rank of a given tree (*i.e.*, the position of the tree) with respect to the ordering $\prec$. In other words, the rank of a tree is the number of trees that precede this tree in the order $\prec$. Therefore, the rank function will be a bijection;

$$rank : (\mathbb{T}_n, \prec) \rightarrow \{1, 2, \ldots, |\mathbb{T}_n|\},$$

and for a tree $T_i \in \mathbb{T}_n$, we have:

$$rank(T_i) = i.$$

A rank function defines a total ordering on the elements of $\mathbb{T}_n$, by the following relation:

$$\forall\ T_i, T_j \in \mathbb{T}_n, \quad T_i \prec T_j \Leftrightarrow rank(T_i) < rank(T_j),$$

Conversely, there is a unique rank function associated with any total ordering defined on $\mathbb{T}_n$.

If *rank* is a ranking function defined on $\mathbb{T}_n$, then there is a unique *unranking function* associated with the function rank. The function unrank is also a bijection:

$$unrank : \{1, 2, \ldots, |\mathbb{T}_n|\} \rightarrow (\mathbb{T}_n, \prec),$$

and for any $i \in \{1, 2, \ldots, |\mathbb{T}_n|\}$, we have:

$$unrank(i) = T_i.$$

Unrank is the inverse function of the function rank, meaning that if $T \in \mathbb{T}_n$:

$$rank(T) = i \quad \Leftrightarrow \quad unrank(i) = T.$$

Efficient ranking and unranking functions have several potential uses. We mention some of them now. One application is the generation of a "random" tree from the set $\mathbb{T}_n$. This can be done easily by generating a random integer $i \in \{1, 2, \ldots, |\mathbb{T}_n|\}$, and then unranking on i. This algorithm ensures that every element of $\mathbb{T}_n$ is chosen with equal probability of $\frac{1}{|\mathbb{T}_n|}$ (assuming that the random number generator being used is unbiased).

Another use of ranking and unranking is in storing trees in the computer. Instead of storing a tree, which could be complicated, an alternative would be to simply store its rank, which of course is just an integer. If the tree is needed at any time, then it can be recovered by using the unranking algorithm. Also, for example, in traditional tree compression algorithm for encoding the tree to code sequence and decoding the code sequence back to a tree, the ranking and unranking algorithms can be used.

It is particularly impressive to note the variation of representations of trees that are possible, such as the *bit strings* [82, 115], the *weight sequences* [77], the *P-sequences* [80], the *ℓ-sequences* [80], the *Ballot sequences* [86], the *Z-sequences* [115], and *etc.* In all cases, a one-to-one correspondence is established between the set of trees and the set of certain integer or alphabet sequences; then the set of trees is generated by generating the set of corresponding integer sequences.

Many papers have been published earlier in the literature for generating different classes of trees. For example we can mention the generation of binary trees in [80, 104, 112], k-ary trees in [87, 37, 59, 113, 58, 52, 69, 111], rooted trees in [71, 27, 108], trees with n nodes and m leaves in [78], neuronal trees in [79, 103], and AVL trees in [66]. On the other hand, many papers have thoroughly investigated basic combinatorial features of chemical trees [44, 46, 45, 36, 30, 65, 109].

2.4.2 Trees with Bounded Degree

Studying combinatorial properties of restricted graphs, or graphs with configurations, has many applications in various fields such as machine learning and chemoinformatics. Studying combinatorial properties of restricted trees and outerplanar graphs (*e.g.* ordered trees with bounded degree) can be used for many purposes including virtual exploration of chemical universe, reconstruction of molecular structures from their signatures, and the inference of structures of chemical compounds [117, 94, 40, 46, 44, 50, 14].

In Chapter 6, we study the generation, ranking and unranking of unlabeled ordered trees whose nodes have maximum degree Δ, denoted by T^{Δ} *trees*, we also use T_n^{Δ} to denote the class of T^{Δ} trees with n nodes. *Chemical trees* are the most similar trees to T^{Δ} trees. Chemical trees are the

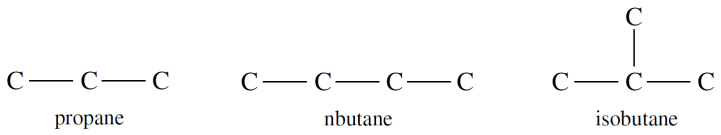


Figure 2.16: Left: C_3H_8 propane, middle and right: C_4H_{10} butanes.

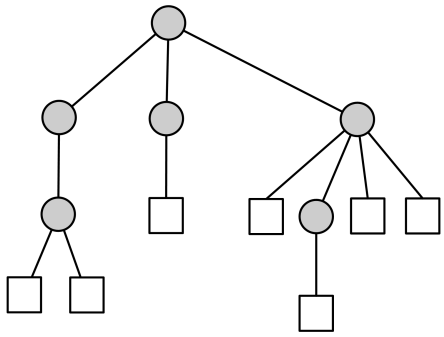


Figure 2.17: A T^{Δ} tree with 12 nodes (for any $\Delta \geq 4$).

graph representations of *alkanes*, or more precisely, the carbon atom skeleton of the molecules of alkanes [44, 46, 45, 36, 30, 65].

The alkane molecular family is partitioned into *classes of homologous molecules*, that is molecules with the same numbers of carbonium and hydrogen atoms; the n^{th} class of alkane molecular family is characterized by the formula C_nH_{2n+2}, $n = 1, 2, ...$ [14] with the same numbers of carbonium and hydrogen atoms. They are usually represented by indicating the carbonium atoms and their links, omitting to represent hydrogen atoms [14], therefore, all the nodes would have the same label; carbon (*i.e.*, the tree is unlabeled), as shown in Figure 2.16 for $n = 3$ and $n = 4$. A *chemical tree* is defined as a tree in which no node has degree greater than 4 [44, 46, 45, 36, 30, 65], chemical trees are also considered to be unlabeled [45, 36, 30, 65]. Therefore, T^{Δ} trees can be considered as a generalization of chemical trees to unlabeled ordered trees whose nodes have maximum degree Δ instead of 4.

Formally, a T^{Δ} tree T is defined as a finite set of nodes such that T has a root r, and if T has more than one node, r is connected to $j \leq \Delta$ subtrees $T_1, T_2, \ldots, T_j$, each one of them is also recursively a T^{Δ} tree and by T_n^{Δ} we represent the class of T^{Δ} trees with n nodes. An example of a T^{Δ} tree is shown in Figure 2.17.

2.4.3 Related Works to Trees with Bounded Degree

More related to our work, in [50] a coding for chemical trees without the generation algorithm, and in [14] the enumeration of chemical trees and in [40, 94] the enumeration of tree-like chemical graphs have been presented. Hendrickson and Parks in [51] investigated the enumeration and the generation of carbon skeletons which can have cycles and are not necessarily trees. The work most related to our research is an algorithm for the generation of certain classes of trees such as chemical trees in [15] with no ranking or unranking algorithm. In that paper, all chemical trees with n nodes are generated from the complete set of chemical trees with $n-1$ nodes, unfortunately, redundant generations are also possible, hence the generation algorithm is not efficient.

The problem of enumeration of ordered trees (without any bounds on the degrees of the nodes) with fixed number of leaves was studied in [114], however no generation algorithm were presented. A generation algorithm for different ordered trees (with no bounds on the degrees of the nodes) was presented in [115]. In [117], a generation algorithm with constant average delay time but with no ranking or unranking algorithms was given for all unrooted trees of n nodes and a diameter at least d such that the degree of each vertex with distance k from the center of the tree is bounded by a given function. In [110] all unrooted unlabeled trees have been generated in constant average time with no ranking or unranking algorithms. Nakano and Uno in [72] gave an algorithm to generate all rooted unordered trees with exactly n nodes and diameter d in constant delay time. Therefore, up to now, to our knowledge, neither efficient generation algorithm, nor any ranking or unranking algorithms are known for either 'chemical trees' or 'ordered trees with bounded degree'.

2.5 Summary

In this chapter, some basic concepts of binary search trees [33, 57, 107, 31, 90], external-memory model [2] and cache-oblivious model [81, 39], external-memory/cache-oblivious data structures [13, 21, 23, 24, 29, 74], and the concept of exhaustive generation of trees and trees with bounded degree [77, 96, 115] with previous works [50, 14, 40, 94, 114, 117, 110, 72] were presented.

Chapter 3

Core Partitioning Scheme

We propose a general method to store the nodes of balanced search trees and to obtain provably efficient external-memory/cache-oblivious data structures. The proposed scheme hinges on decomposition of a balanced search tree into a set of disjoint cores: a *core* is a complete balanced binary tree (of height h and with $2^h - 1$ nodes) that appears as a portion of the balanced tree. Our method is *not* invasive, as it does not change the original algorithms. It just requires an efficient post-processing after each update to maintain the cores. The nodes of a core are stored in a chunk of consecutive memory cells. Hence, the core partition adds a memory layout for the nodes of a balanced tree without interfering with the original algorithms for the tree. Simultaneously, we achieve good memory allocation, space-efficient representation, and efficient time and I/O complexities for both external-memory and cache-oblivious memory models compatible to modern search data structures designed purposely for these models. The advantages and disadvantages of the main result of this chapter has been presented in [3].

In this chapter, in Section 3.1, we introduce the basic idea of cores in binary search trees and its preliminary definitions and properties, then we define the *core partitioning scheme* in Section 3.2. After that, in Section 3.3 we discuss how to obtain efficient external-memory/cache-oblivious results including linear space and $O(\log_B n)$ I/Os and $O(\log n)$ comparisons for search operation. Finally, as case studies, we show that the core partitioning scheme can be applied to weight-balanced trees with the amortized update cost of $O(\log_B n)$ I/Os in Section 3.4 and to AVL trees with more than a polylogarithmic amortized cost of updates in Section 3.5.

3.1 Core Partitioning Preliminaries

For a given binary search tree T of size n and height H, for a parameter $h^*(T)$ (that depends on the type of balanced tree), our recursive scheme requires that the first $h^*(T)$ levels of nodes in the given balanced tree are full, thus they form a core. It conceptually removes these nodes and applies recursively this process to the resulting bottom subtrees. The recursion ends when the subtree size is below a threshold r^* to be specified, we call such a (possibly empty) terminal subtree, a *terminal-core*. As a result, the given balanced tree is decomposed into cores, which are central to our findings. We obtain a *successful core partition* when the cores found along any root-to-leaf path of the balanced tree are of doubly exponentially decreasing size, with $O(1)$ of them being of size smaller than r^*.

For a given binary search tree T of size n, generally, $h^*(T)$ is a function of $|T|$ or of $h(T)$ (or of both) and r^* is a function of n (the size of the entire tree) or of B (the block size), therefore, we can instead denote them as $h^*(|T|, h(T))$ and $r^*(n, B)$. In this thesis, for the sake of simplicity, we denote them by h^* and r^*, respectively.

We show that for any binary search tree with such a successful core partition, we obtain a space-efficient external-memory/cache-oblivious layout to dynamically maintain the structure and their keys. Using the external-memory/cache-oblivious models [2, 81], it takes $\Theta(n/B)$ blocks of memory of size B to store the keys with extra $O(n)$ bits space needed for the external pointers to the cores and the terminal-cores. Note that representing the structure of a balanced binary tree using $O(n)$ bits is also another efficient bound independently achieved by the core partitioning scheme. Searching a key requires $O(\log_B n)$ block transfers and $O(\log n)$ comparisons in the external-memory and the cache-oblivious memory models, and the amortized cost of update varies with the specifications of the balanced binary tree.

We present the core partitioning scheme as a general approach for making different classic and well-studied balanced binary search trees efficiently applicable in external-memory/cache-oblivious models and compatible to the modern search data structures, thus making our method of independent interest. More precisely, similarly to our case studies, a core partitioning scheme can be applied to other types of balanced binary search trees. For any type of balanced binary search trees, if one can prove that they admit a successful core partition, all of the core partition properties such as external-memory efficiency, cache-obliviousness, linear space for the keys and $O(n)$ bits for external pointers to the cores and the terminal-cores, and $O(\log_B n)$ search cost would be instantly achieved. More importantly, the original structure of that binary search tree

will *always* be preserved. However, the update cost varies depending on the class of binary search trees.

An example of the benefit of our technique is that, by preserving the original structure of the given binary search tree, we can reuse the vast knowledge on balanced search trees to provide a repertoire of space-efficient external-memory and cache-oblivious data structures which are competitive with modern search data structures that are purposely designed for these models (e.g. [20, 21, 23, 24, 29]). This opens a number of possibilities that are known for modern search data structures but unknown for several previous balanced trees:

- I/O efficiency and cache-obliviousness can be achieved for a tree of n nodes, as explained in Section 3.3.

- Dynamic memory management can be easily handled by allocating a common contiguous memory chunk for all the keys of each core, since each core contains a number of keys that is a power of two (minus one). This alleviates memory fragmentation.

- The total space is $O(n)$ 'words' to store the keys with an extra $O(n)$ 'bits' for the external pointers to the cores and the terminal-cores.

- Search can be performed in $O(\log_B n)$ I/Os and $O(\log n)$ comparisons.

Thus, these 'classic' search data structures can be dynamized as efficiently as the ones specifically designed for external memory and cache-oblivious memory models.

We emphasize that the above features just require the original algorithms described for the given balanced tree and what we add is the maintenance of our structure for the nodes, and the algorithmic challenge is how to maintain it efficiently. When performing the updates, we proceed as usual, except that, we perform a post-processing: loosely speaking, we take the topmost core that must 'change' because of the update, and we recompute the partition from it in a greedy fashion.

When comparing our results to previous work, we observe that it is folklore to prove that cores can be found in some data structures as mentioned in Chapter 2 but they have never been used before in the literature to make *classic* data structures efficient in external-memory/cache-oblivious models. We think that the contribution of our work is to show how to exploit the core partition to turn some classic balanced search trees into competitive external-memory/cache-oblivious data structures that have guaranteed bounds, using a general technique.

We adopt the external-memory model [2] introduced in Chapter 2 to evaluate the I/O complexity, where B is the block size of the data transfers between main and external memory, and the I/O complexity accounts for the number of block transfers performed during the computation. We also adopt the cache-oblivious model [81, 39] presented in Chapter 2 to evaluate the I/O complexity, here called *cache complexity*. Recall that in this model, B is an *unknown* parameter for the block size and a cache-oblivious algorithm is completely *unaware* of the value of B used by the underlying system: this is a strength as it can thus show good performances on a multilevel memory hierarchy without knowing the cache size or the size of the block transfer [81, 39].

3.2 Core Partitioning Scheme

For an arbitrary binary tree with n nodes, the *level* of a node is the number of the nodes in the path to the root (the root is on level 1), we say that level i in T is *full*, if it contains all the 2^{i-1} nodes, we adopt the standard terminology [57], where the *height of a node* in a tree is the number of nodes on the longest downward path from the node to a leaf, and the *height of a tree* is the height of its root.

3.2.1 Core Partitioning

We say that a binary tree has a *core* of height h^*, if its topmost h^* levels form a complete balanced binary tree. We are interested in the families of binary search trees for which each subtree has a core of guaranteed height. Later we will observe that for external-memory model, the nodes of a core can be stored in blocks of size of multiples of B, and for cache-oblivious memory model they can be stored in a chunk of consecutive memory cells (using van Emde Boas (vEB) layout [29, 81], this structure performs search operation in $\Theta(\log_B n)$ I/O transfers, which is optimal within a constant factor). The existence of such a core in balanced binary search trees is highly expected as they are 'balanced', however, in Sections 3.4 and 3.5 we prove it for weight-balanced trees and AVL trees.

Consider a binary search tree T with n nodes and any two given integer parameters $h^* \geq 1$ and $r^* \geq 1$, such that each nonempty subtree of T of size larger than r^* has a core of height h^*, where as mentioned before, generally h^* is a function of the subtree size or height and r^* is a function of the size of the entire tree or B (the block size). The recursive scheme consists of the following steps.

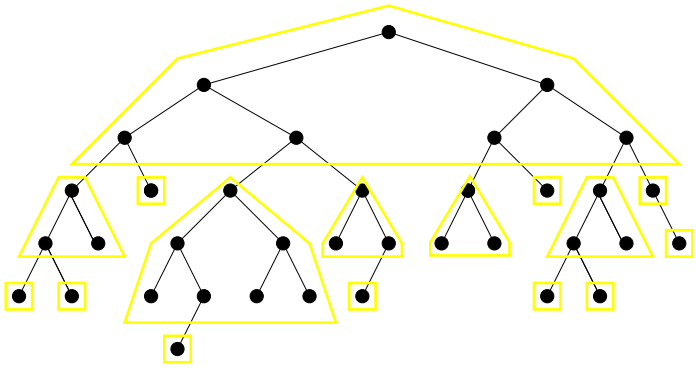


Figure 3.1: Decomposition of a binary search tree into its cores.

1. Conceptually remove the topmost core of height h^* (made up of the topmost h^* levels), which is a complete binary tree of $2^{h^*} - 1$ nodes.

2. Recursively perform the core removal of the bottom subtrees thus obtained, where each of the bottom subtrees can potentially have different height or size.

3. Stop the recursion for a possibly empty subtree (terminal-core) when its size is less than or equal to r^*.

The case for $h^* = 1$ and $r^* = 1$ returns the trivial partition of the tree T into its individual nodes and is of little interest. But other choices of h^* and r^* are more interesting to investigate: a binary search tree T can be seen as conceptually decomposed into a collection of complete binary trees, *i.e.*, the cores, where each core is the top tree that is obtained from the recursive scheme applied to its subtree, plus the subtrees of size less than or equal to r^*. Two cores are linked together if and only if there is one node in one of the two cores that is linked to a node in the other core, where one of the two nodes is the root of the core and the other is a leaf of the other core. Figure 3.1 illustrate core partitioning on a small binary search tree.

In the following, for $n > 1$, when we consider any root-to-leaf path, we let $C_1, C_2, \ldots, C_{t-1}, B_t$ be the subtrees thus traversed, here C_1 is the core containing the root of the tree, B_t is the (possibly empty) terminal-core of size less than or equal to r^* at the end of the path, and $C_2, \ldots, C_{t-1}$ are the cores traversed when going from C_1 downward to B_t. We say that core C_i is *at level* i to indicate that the path from the root of the tree to any descendant of C_i (nodes in C_i included) must traverse $C_1, C_2, \ldots, C_i$. We also denote by h_i^* the height of C_i, namely, $|C_i| = 2^{h_i^*} - 1$.

Definition 3 (successful core partition) *We say that our recursive scheme with parameters h^* and r^* is a* successful core partition *if both conditions below are satisfied.*

1. *There exists a positive constant $\gamma < 1$ such that for any sequence $C_1, C_2, \ldots, C_{t-1}, B_t$ traversed by a root-to-leaf path, the cores are of doubly exponentially decreasing size in γ, namely, there is an integer constant $c \geq 1$ such that $h^*_i \leq \gamma h^*_{i-c}$ (for $c < i \leq t-1$)*[1].

2. *For any sequence $C_1, C_2, \ldots, C_{t-1}, B_t$ traversed along a root-to-leaf path, there are $O(1)$ cores C_i of small size $|C_i| < r^*$.*

Our definition of cores resembles what happens in van Emde Boas trees [81, 29] and exponential trees [10, 21] in that the cores found along a root-to-leaf path have doubly exponentially decreasing sizes, except a constant number of them. In the rest, we will prove that for any given binary tree who has a successful core partition (*e.g.* weight-balanced trees and AVL trees), the external-memory/cache-oblivious properties and the space-efficiency hold.

Lemma 2 *For any binary tree of size n, a* **successful** *core partition with parameters h^* and r^* correctly terminates producing terminal-cores at the bottom of the tree and $O(n/r^*)$ cores, with $O\left(\log \frac{\log(n+1)}{\log(r^*+1)}\right) = O(\log \log n)$ cores traversed in any root-to-leaf path.*

Proof: Since $h^* \geq 1$, by definition of our recursive scheme, the algorithm eventually terminates when the subtree size is $\leq r^*$. Also, there are overall $O(n/r^*)$ cores generated by the scheme as we prove next. Note that, for any core C, its size ($|C|$) can be less than r^* even though, its subtree (the subtree rooted at the root of C) has size $\geq r^*$ by our recursive scheme. We observe that there exist at most $n/r^* + 1$ cores C of size $|C| \geq r^*$ since the sum of their sizes cannot exceed n. Hence, to count the total number of the cores, let us conceptually remove every such a core C of size $|C| \geq r^*$. Now consider the topmost remaining cores (with size $\leq r^*$), they are obtained by *disjoint* subtrees of size $\geq r^* + 1$ (by our recursive scheme), therefore their number can not exceed $O(n/r^*)$ either. Now repeat the latter and conceptually remove them, again the topmost remaining cores (with size $\leq r^*$) have the same property (obtained by disjoint subtrees of size $\geq r^* + 1$) and their number can not exceed $O(n/r^*)$. Repeat this until all the cores are removed. By Definition 3.2 this iteration can not be repeated more than $O(1)$ times, otherwise, it is equivalent to have more than $O(1)$ cores with size $\leq r^*$ in a root-to-leaf path. Therefore, the total number of cores is $O(n/r^*)$.

On the other hand, for t (the number of cores traversed in a root-to-leaf path), when using the inequality of Definition 3.1, by induction, we can prove that $h^*_i \leq \gamma^k h^*_{i-kc}$, where k is the largest

[1] We will show that $\gamma = 2/3$ and $c = 2$ for AVL trees, and $\gamma = (\log_{2/\alpha}(1-\alpha) + 1)$ and $c = 1$ for weight-balanced $BB[\alpha]$.

integer such that $i - kc \geq 1$. Also $h^*_{i-kc} \leq \log(n+1)$ as $|C_{i-kc}| = 2^{h^*_{i-kc}} - 1 \leq n$. Therefore, $h^*_i \leq \gamma^{\frac{i-1}{c}} \log(n+1)$. Let j be the largest i such that $|C_i| \geq r^*$. Observe that $h^*_j \geq \log(r^*+1)$, and that $t = j + O(1)$ by Definition 3.2. Hence $\gamma^{\frac{j-1}{c}} \log(n+1) \geq \log(r^*+1)$, which implies that $j = O(\log(\log(n+1)/\log(r^*+1)))$, and so does t. □

3.2.2 Memory Management

Given a balanced binary search tree with successful core partition, here we explain the basics of its memory management. Let us assume $r^* = \Theta(\log n)$, store each core C_i using the *implicit* vEB layout for its keys into an array without requiring internal pointers (note that, all levels are full in a core). These elements are the keys in the nodes of C, so that it takes $1 + \log_B |C|$ block transfers to implicitly traverse C during a search path [29]. We only keep the pointers from the nodes in the last level of C_i to the roots of the "children" cores. We can also store the keys of each small subtree (terminal-core) of size $n_0 \leq r^*$ in an array of n_0 entries. The simplest way to store these cores is as follows: the arrays for the cores and the terminal-cores are stored in two large segments $\mathcal{C}$ and $\mathcal{S}$ of adjacent memory cells, respectively, in decreasing order of size one after the other. The arrays for the cores are kept in $\mathcal{C}$ while the arrays for the terminal-cores are kept in $\mathcal{S}$. Note that the wasted space is minimal in this way, since we have to store, for each size, how many arrays are of that size.

Fact 2 *Consider a core C, the inorder traversal of all the nodes in C and traversing C during a search path requires $O(|C|/B)$ I/Os and $1 + \log_B |C|$ I/Os, respectively.*

3.2.3 Maintaining the Core Partition

A natural question is how to handle updates, namely, insertions and deletions. Note that during a sequence of insertions, various changes to some cores may occur; namely, a core may need to increase its size because of the increase in its subtree size, or a core may need to change its content because of involving in a rotation operation. Given a node z that is the root of the topmost core that changes size or content, observe that *locally* rebuilding the core partition scheme on z and its descendants does not change the *global* core partition obtained from the root of the whole tree. We exploit this locality to update the core partition of a binary search tree and we define a new reconstructing operation (called *repartition* on node u) by means of the following greedy algorithm for a node u.

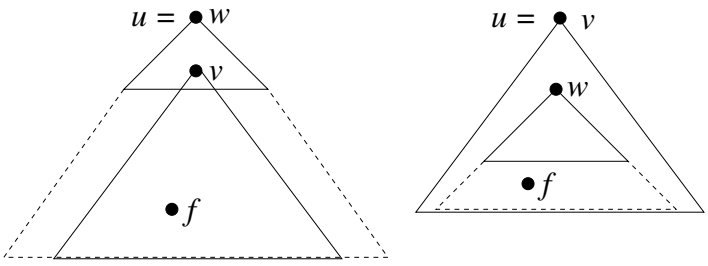


Figure 3.2: Left: when w is higher than v, so $u = w$. Right: when w is lower than v, so $u = v$.

- `repartition`(u):

  (1) rebuild the core C that contains u, and

  (2) find $v_1, \ldots, v_k$, the topmost descendants of u that are *not* in C, and locally recompute the core partition for each node v_i if it is needed ($i = 1, \ldots, k$).

We proceed as usual by inserting a new node (typically a leaf) f and finding its ancestor v (if it exists) that has to be restructured. We also find the topmost ancestor w (if it exists) of f such that w is the root of a core that changes size because of the insertion of f. If neither v nor w exist, return; otherwise, let u be the topmost between v and w (as shown in Figure 3.2), and perform `repartition`(u).

As for deletions, if the physical deletion is actually made, we proceed as in the insertion, locating the topmost node u and performing `repartition`(u). Another possibility is to avoid to use `repartition`. We simply mark the searched key as logically deleted, and remove that mark if the key is inserted again. We periodically rebuild the tree when the number of these marked keys is a constant fraction of the total number of keys.

To analyze `repartition`(u), we need to focus on the following three main events, note that these three events cover the definition of `repartition`(u), also for each event, its cost is discussed based on the Fact 2.

- core resize: if w exists and $u = w$, this accounts for (1) and (2) in `repartition`(u), with a cost proportional to the size of the full subtree rooted at the core's root.

- core rescan: if v exists and $u = v$, this accounts for (1) in `repartition`(u), with a cost proportional to the size of the core containing the critical node v.

- subtree rescan: if v exists and $u = v$, this accounts for (2) in `repartition`(u), with a cost proportional to the size of the subtree rooted at the critical node v.

Note that core resize only occurs when $u = w$ (Figure 3.2: Left) while core rescan and subtree rescan occur when $u = v$ (Figure 3.2: Right). Also the case $u = w = v$ is feasible and all the three events happen in this case: however core resize is chosen as representative since its cost dominates that of core rescan plus subtree rescan. If logical deletions are performed, there can be actually a fourth event. It happens because of rebuilding the binary tree when the number of the deleted keys is a constant fraction of the total number of keys, whose amortized complexity can be analyzed in a traditional way, and thus it is not discussed here (or put into another way, the inserted keys pay also for their possible deletion in the future).

Lemma 3 *When* `repartition`(u) *is applied to a node* u*, let* g *be the size of the core containing* u *(if core rescan occurs in case* $u = v$*) and* s *be either the size of the subtree rooted at* u *(if subtree rescan occurs in case* $u = v$*) or the size of the subtree rooted at the core's root (if core resize occurs in case* $u = w$*). Then, the cost is* $O(g + s)$ *time and* $O((g + s)/\min\{r^*, B\})$ *block transfers, where* B *is the block size.*

Proof: Note that by definition, s equals to the number of the nodes which their entire subtrees need to update, and g equals to the number of the nodes in the core containing u (if core rescan occurs) which they may need to update too, besides those nodes, no other nodes of the tree changes (because of the locality of the cores). Therefore, an $O(g + s)$-time algorithm can rebuild the core partition. Moreover, the number of cores is $O(s/r^*)$ by Lemma 2, and scanning them takes so many block transfers plus $O(g/B + s/B)$. □

3.3 Applications

Let $\mathbb{T}$ be a class of balanced binary search trees with successful core partition. Suppose that the search time is $O(\log n)$ for $\mathbb{T}$ and update requires a 'rebalancing' operation when the balance factor of $\mathbb{T}$ is violated. In this section, we analyze the core partitioning scheme for $\mathbb{T}$. In particular, we obtain space-efficient external-memory and cache-oblivious search trees.

3.3.1 External-Memory Search Trees

For the class $\mathbb{T}$, set $r^* = max\{\log n, B\}$ and obtain a *B-tree-like* data structure for external memory [19]. More precisely, the complete balanced binary tree represented by each core C_i can be stored in blocks of size of multiples of B, so that it takes $O(1 + h_i^*/\log B)$ I/Os to traverse C_i (e.g. see [106]). Moreover, the sibling subtrees of size at most r^* for which the recursion stops,

are packed together in a greedy fashion from left to right: namely, while the next subtree can be stored in the current block, pack it in the block; otherwise, open a new block.

Theorem 1 *Let $\mathbb{T}$ be a class of balanced binary search trees with successful core partition and parameters $h^* = \Theta(\log |T|)$ and $r^* = B$, where B is the block transfer size for the external memory, we can store its nodes in blocks of size B, so that $O(n/B)$ blocks are occupied and any search path from the root to a node requires $O(\log_B n)$ I/Os and $O(\log n)$ comparisons.*

Proof: We observe that the number of occupied blocks is $O(n/B)$ by Lemma 2 and since there are $O(\log \log n)$ different sizes of memory chunks, each of length a power of two, it is not difficult to keep these n nodes in $O(n)$ contiguous memory cells. This guarantees that $O(n/B)$ blocks are occupied for any given block size B. As for the search cost, $O(\log n)$ comparisons derive from the standard analysis of $\mathbb{T}$. Consider the cores $C_1, C_2, \ldots, C_{t-1}$ traversed in a root-to-leaf path of the tree, and let B_t be the (possibly empty) terminal-core of size at most r^* at the end of the path. We just need to follow external-memory references when moving from one core to another core or to B_t. Thus the I/O complexity is $O(1 + h_i^*/\log B)$ per core C_i plus one I/O to access B_t. This gives a total I/O cost of $O(t + \sum_{i=1}^{t-1} h_i^*/\log B) = O\left(\log \log_B n + (1/\log B)\sum_{i=1}^{t-1} h_i^*\right) = O(\log_B n)$. □

3.3.2 Cache-Oblivious Search Trees

We fix $r^* = \lceil \log n \rceil$ and $h^* = \lceil \kappa \log |T| \rceil$ (where $\kappa \leq 1$ is a constant factor) and employ the following memory layout of the nodes. However, for the sake of computation, we simply consider $r^* = \log n$ and $h^* = \kappa \log |T|$. We store the subtrees of size at most r^* in a contiguous memory chunk. We then store the complete binary tree inside each core C_i in a contiguous memory chunk using the *vEB layout* [81, 29], so that it takes $O(1 + h_i^*/\log B)$ block transfers to traverse C_i during a search path. This suffices to obtain cache-oblivious bounds.

Theorem 2 *Given a core partition for a tree $T \in \mathbb{T}$ of size n (big enough) and parameters $h^* = \kappa \log |T|$ and $r^* = \log n$, a memory layout can be used where subtrees and cores are each stored in a contiguous memory chunk, each core using the vEB layout, so that $O(n/B)$ blocks are occupied and any search path from the root to a node requires $O(\log_B n + \log(\frac{\log(B+1)}{\log(\log n+1)}))$ block transfers and $O(\log n)$ comparisons.*

Proof: Since there are $O(\log \log n)$ different sizes of memory chunks, each of length a power of two, it is not difficult to keep these n nodes in $O(n)$ contiguous memory cells. This guarantees that $O(n/B)$ blocks are occupied for any block size B. The bound of $O(\log n)$ comparisons derives

from $\mathbb{T}$'s properties. As for the cache complexity, consider the cores $C_1, C_2, \ldots, C_{t-1}$ traversed in a root-to-leaf path of T, and the terminal-core B_t of size at most $r^* = \log n$ at the end of the path. Let C_k be the smallest core among $C_1, C_2, \ldots, C_{t-1}$ (*i.e.*, largest k) such that $|C_k| \geq B$ and $1 \leq k \leq t-1$. Then, the cache complexity of traversing $C_1, C_2, \ldots, C_k$ is $O(\log_B n)$ as we just saw before. The extra cost is given by traversing $C_{k+1}, \ldots, C_{t-1}, B_t$, namely $O(t-k)$ block transfers. Note that $|C_{k+1}| < B$ by definition of k and, hence, $|T_{k+1}| \leq B^{O(1)}$ by our choice of h^*. Using Lemma 2 on T_{k+1} with at most $B^{O(1)}$ nodes, we obtain $t-k = O(\log(\log(|T_{k+1}|+1)/\log(r^*+1))) = O(\log \frac{\log(B+1)}{\log(\log n+1)})$. □

The $O(\log_B n)$ Block Transfer Solution for Search Operation

In Theorem 2, the $\log(\frac{\log(B+1)}{\log(\log n+1)})$ term in the number of block transfers of search operation can be avoided by keeping any core C and all its descendant nodes in a contiguous portion of memory[2]. This may increase the space but we will see how to handle this problem.

Let ζ be a constant such that for every tree $T \in \mathbb{T}$ with size n, the number of the cores in every root-to-leaf path is less than or equal to $\zeta \log\log n$ (by Lemma 2 we know that there is such a ζ). As shown in Figure 3.3, let τ be a subtree with its core C and $\tau_1, \tau_2, \ldots, \tau_k$ be the topmost subtrees below C, here we define the level and the layer of a core. The *level* of C (denoted by $lev(C)$) is the number of the cores above and the *layer* of C (denoted by $lay(C)$) is $\lceil \zeta \log\log n \rceil - lev(C)$.

To ensure that C and all its descendant nodes fit in a contiguous portion of memory, we assign sufficient space for all C's subtrees in a recursive manner. Let us assume $lay(C) = i$ and the total number of the nodes in C's subtree is m (C and all its descendants), we assign $\Theta(4^i m)$ space to C and all its descendants. Therefore, since the maximum value of the layer of a core is $\lceil \zeta \log\log n \rceil$, this will increase the total space for T to $O(n4^{\lceil \zeta \log\log n \rceil}) = O(n \log^{2\zeta} n) = O(n \,\mathrm{polylog}(n))$ words (if each key occupies one word). We will later see how to decrease this space to $O(n)$ words.

During a sequence of insertions, let C be a core who is being newly created or its memory space needed to be reallocated (*e.g.* by a rebalancing operation above), let τ denote the subtree rooted at the root of C, $|\tau| = m$, let $\tau_1, \tau_2, \ldots, \tau_k$ be the topmost subtrees below C (for $1 \leq i \leq k$, $|\tau_i| = m_i$). Let $SP(\tau)$ denote the space needed for τ in this memory reallocation. $SP(\tau) = 4^{lay(C)}m$ space to C and its descendant cores will be assigned as follows. We allocate a memory of size $|C|$ to the nodes inside core C, followed by the space needed for each τ_i ($1 \leq i \leq k$) in a recursive manner (equals to $\sum_{i=1}^{k} SP(\tau_i)$), followed by a large amount of remaining free space (equals to

[2]Keeping the nodes of a subtree in a contiguous portion of the memory is a general well known approach for cache-oblivious algorithms [20, 21], however, technically, it differs from one paper to another and to ours.

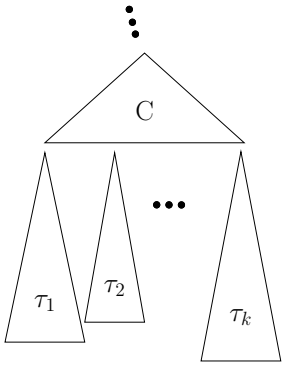


Figure 3.3: Subtree τ with its core C and $\tau_1, \tau_2, \ldots, \tau_k$ the topmost subtrees below C.

$4^{lay(C)}m - |C| - \sum_{i=1}^{k} SP(\tau_i)$), we denote this free space by *spare-space*.

During a sequence of insertions, let *space-overflow (SOF)* denote the event that C and all its descendant cores can not fit anymore in to the space allocated before, in this case we release the whole space and reallocate a new space of size $SP(\tau) = 4^{lay(\tau)}|\tau|$ from the spare-space of its parent core[3], then, if its parent core has a new SOF because its spare-space is not sufficient, we repeat this procedure for its ancestor cores until we reach a core without SOF or we reach the root. In general we have the following three main events resulting in a new memory allocation (*memory reallocation*) of a core C and all its descendant cores (descendant nodes).

1. A rebalancing operation occurs at the root of C or above. The amortized cost of this event depends on the properties of $\mathbb{T}$ (See our case studies in Sections 3.4 and 3.5 for more details).

2. A SOF happens. The total amortized cost of this event will be discussed in Theorem 4.

3. $\lceil \zeta \log\log n \rceil$ increases by one. Note that the layer of every cores is defined based on $\lceil \zeta \log\log n \rceil$, therefore, an increase in $\lceil \zeta \log\log n \rceil$ requires a memory reallocation of the entire tree. This event happens rarely during a sequence of n insertions (at most $\lceil \zeta \log\log n \rceil$ times) and the amortized cost will be computed in Theorem 5.

Fact 3 *Let C be a core and τ be the subtree rooted at the root of C, in the memory allocation of τ, considering the extra space we allocate to guarantee the cache-obliviousness, at most $4^{lay(\tau)}$ words are allocated for any 'real' node $v \in V(\tau)$ (v represents a key).*

Proof: By the memory allocation explained above, when a memory reallocation is required we assign $4^{lay(\tau)}|\tau|$ words and $|\tau|$ is the number of the keys. Later, new keys can be also added, this is equivalent to say that at any moment, for any key we have assigned at most $4^{lay(\tau)}$ words. □

[3]Note that $|\tau|$ has been increased since the last time we reallocated memory to τ.

Lemma 4 *Whenever a SOF occurs on τ, the size of τ is at least twice its size on the previous memory reallocation (caused by a SOF, or a rebalancing operation above, or a $\lceil \zeta \log\log n \rceil$ increase).*

Proof: We use induction on the number of the nodes. Let us assume that on the t^{th} insertion, a new SOF occurs on the memory allocation of τ and on the t_0^{th} insertion ($t_0 < t$) the last memory reallocation was performed on τ. Since τ changes by inserting new keys, we use $\tau(t)$ and $\tau(t_0)$ to denote it on the t^{th} and t_0^{th} insertions, respectively. Note that $\tau(t)$ and $\tau(t_0)$ have the same layer and the same core size $|C|$, but their overall sizes differ. Let $\ell = lay(\tau(t)) = lay(\tau(t_0))$, $m = |\tau(t)|$, and $m_0 = |\tau(t_0)|$, and $\tau_1(t), \tau_2(t), \ldots \tau_k(t)$ denote the subtrees below C on $\tau(t)$, and $\tau_1(t_0), \tau_2(t_0), \ldots \tau_k(t_0)$ denote the subtrees below C on $\tau(t_0)$. We will prove that $m \geq 2m_0$.

On the t^{th} insertion when there is no more enough free space in $\tau(t)$'s allocated memory, let $\mathcal{S}$ be the total amount of space needed for C and $\tau_i(t)$s $1 \leq i \leq k$ (since we have a SOF we know that one $\tau_i(t)$ can not fit in the current spare-space). By induction hypothesis, any time a τ_i has a SOF, its size at least doubles since the last time, this guarantees that its size grows faster than an exponential function of power of 2. Therefore, $\mathcal{S}$ is strictly greater than half of the entire space previously allocated to τ (because of SOF and that exponential function property). Hence, $\mathcal{S} > \frac{1}{2} SP(\tau(t_0)) = \frac{1}{2} 4^{\ell} m_0 = 4^{\ell-1} 2m_0$.

On the other hand, in τ's current memory allocation, for every node (representing a key) in C we have exactly one word and for every node in $\tau_i(t)$s ($1 \leq i \leq k$) we occupy no more than $4^{lay(\tau_i)} = 4^{\ell-1}$ words (by Fact 3 and the fact that τ_i is one level below τ so $lay(\tau_i) \leq \ell - 1$). Therefore, the total number of nodes in $\tau(t)$ is greater than $\frac{\mathcal{S}}{4^{\ell-1}} > 2m_0$. Hence, the proof is complete. □

Theorem 3 *On this memory reallocation, if C is a core and τ is a subtree rooted at the root of C with $|\tau| = m$, the search operation costs $O(\log_B m)$ I/Os in the cache-oblivious model.*

Proof: By definition, the space allocated to τ is $O(m \operatorname{polylog} m)$, also every core and its descendant cores are packed in a contiguous portion of memory, therefore, the search cost in the cache-oblivious model is:

$O(\log_B(m \operatorname{polylog} m)) = O(\log_B m + \log_B(\operatorname{polylog} m)) = O(\log_B m)$ I/Os. □

The Linear Space Solution

Here, we study the space and we prove that in this new memory allocation, the space needed to store the keys can be reduced to $\Theta(n)$ words (if each key occupies one word), using an extra $O(n)$

'bits' space for the external pointers to the cores and the terminal-cores.

Lemma 5 *The space needed to store the keys in this scheme can be reduced to* $\Theta(n)$ *words.*

Proof: To decrease the space to $\Theta(n)$, we use buckets of size $\Theta(\log^{2\zeta} n)$ as explained by Bender *et al.* in [21]. Then we store the buckets separately from the cores of the remaining tree. Each bucket is implemented as $O(1)$ layers of records of size $\Theta(\log n)$ (*e.g.* in the range $[\log n, 2\log n)$). These records are very small comparing to n, so we have a variety of possibilities to implement them, we can implement them as B-trees same as [21] or we can even implement them with sorted arrays in a contiguous portion of memory, thus each record can be sequentially scanned in $\Theta(\log n)/B) = \Theta(\log_B n)$ I/Os, hence, $O(1) \times \Theta(\log_B n) = O(\log_B n)$ I/Os to search a bucket, therefore, it will not change the search complexity.

The space to store these buckets will be $\Theta(\frac{n}{\log^{2\zeta} n} \times \log^{2\zeta} n) = \Theta(n)$. On the other hand, this bucketing reduces the number of the nodes in the main tree to $\overline{n} = \Theta(\frac{n}{\log^{2\zeta} n})$ which implies that the space needed for the core partition reduces to:

$$\Theta(\overline{n} \log^{2\zeta} \overline{n}) = \Theta(\frac{n}{\log^{2\zeta} n} \times \log(\frac{n}{\log^{2\zeta} n})^{2\zeta}) = \Theta(n).$$

□

Lemma 6 *The total space needed for all the pointers to the cores and the terminal-cores is* $O(n)$ *bits.*

Proof: As explained in Section 3.2.2, we only keep the pointers from the nodes in the last level of each core to the roots of its "children" cores, therefore, for every core and terminal-core we have one pointer. On the other hand, by using the above buckets, clearly the number of terminal-cores is $O(\frac{n}{\log^{2\zeta} n})$ and as proved in Lemma 2, the total number of the cores is $O(n/r^*) = O(\frac{n}{\log n})$, if $r^* = \Omega(\log n)$, therefore, the total space needed for the pointers (pointing to the cores and the terminal-cores) will be upper bounded by: $O(\frac{n}{\log^{2\zeta} n} + \frac{n}{\log n}) \times \log n = O(n)$ bits. □

Theorem 4 *The total amortized cost of SOF is* $\Theta(\log \log n)$.

Proof: Since the space is now $\Theta(n)$, for any core C and subtree τ rooted at the root of C, when a memory reallocation occurs on τ, the cost of the memory reallocation is $\Theta(|\tau|)$ operations and $\Theta(|\tau|/B)$ I/Os. On the other hand, by Lemma 4, when a new SOF occurs on τ, between the last memory allocation of τ and this SOF, we have $\Theta(|\tau|)$ fresh insertions. Therefore, by assigning

$\Theta(\log\log n)$ credits to every newly inserted node (key), these credits can cover the total cost of future SOF events. Hence, the total amortized cost of SOF is $\Theta(\log\log n)$. □

Theorem 5 *As discussed before, an increase of* $\lceil \zeta \log\log n \rceil$ *results to a memory reallocation of the entire tree. The total amortized cost of the memory reallocations of this event is* $O(\log\log n)$.

Proof: $\lceil \zeta \log\log n \rceil$ increases at most $\zeta \log\log n$ times during a sequence of n insertions, clearly the amortized cost is upper bounded by $\frac{n\zeta \log\log n}{n} = O(\log\log n)$. □

If one is interested to perform range queries using core partitioning scheme, the above bucketing also allows us to apply some range query techniques inside our buckets. For example, we can apply the technique Bender *et al.* used in [23, Section 3. for *bottom levels* of *ordered B-trees*] inside our buckets and perform search queries in $O(\log_B n + k/B)$ block transfers in the worst case (k is the number of reported keys) with $O(\log_B n + \frac{\log^2 n}{B})$ I/Os amortized update cost.

3.4 Case Study 1: Weight-Balanced Trees

In this section and in the next section, as case studies we apply core partitioning scheme on weight-balanced trees and height balanced trees (AVL trees), respectively. As mentioned before, if one can prove that a balanced binary tree has a successful core partition, then the linear space, $O(n)$ bits space for the pointers to the cores and the terminal-cores, efficient external-memory, and efficient cache-obliviousness will be immediately obtained. However, since every balanced binary tree has a different approach for rebalancing, the cost of update would differ, therefore, for both of our case studies, not only we prove that they have a successful core partition, but also, we compute their amortized cost of update. Here, we show that weight-balanced trees have a successful core partition, then we show that the amortized cost of update is $O(\log n)$ and $O(\log_B n)$ I/Os which is efficient.

Recall from the definition of weight-balanced trees given in Chapter 2, for a binary tree, the weight is the number of *null* nodes (null pointers), which is equivalently the number of nodes (*i.e.*, the size) plus one. The weight of a node u is denoted by $w(u)$ and its balance $\beta(u) = w(u.l)/w(u)$ is the ratio between the weight of u's left child and u's weight (note that $w(null) = 1$ by definition of weight), $u.l$ and $u.r$ denote the left child and the right child of u, respectively.

For a parameter α, where $0 < \alpha \leq 1$, a weight-balanced tree (a.k.a. $BB[\alpha]$-tree) is a binary search tree where each node u satisfies $\alpha \leq \beta(u) \leq 1-\alpha$, which is equivalent to say that $\alpha \cdot w(u) \leq w(u.l), w(u.l) \leq (1-\alpha) \cdot w(u)$ for each node u and its two children $u.l$ and $u.r$.

For example, the tree shown in Figure 2.10 is a $BB[\alpha]$-tree for $\alpha = 2/7$ while it is not for $\alpha = 1/3$.

As observed by the inventors Nievergelt and Reingold [73], a node of weight 3 should have one child of weight 1, so they assume that $0 < \alpha \leq 1/3$. Moreover, Blum and Mehlhorn [28] show that rebalancing a $BB[\alpha]$-tree with rotations can be done when $2/11 < \alpha \leq 1 - \sqrt{2}/2 = 0.2928\ldots$. When α is strictly inside this interval, they show that there exists $\delta > 0$ depending on α such that an unbalanced node u has balance factor $(1+\delta)\alpha \leq \beta(u) \leq 1 - (1+\delta)\alpha$ after its balance is restored using rotations. Overmars [76, Sect.4.2] shows that rebalancing can be also done with partial rebuilding, and this only requires $0 < \alpha < 1/2$ and obtains a value of $\beta(u)$ close to $1/2$ after restoring the balance of u. In both cases, the two properties are important in the amortized complexity for the following reason, as proved in [28, 76].

Lemma 7 *For weight-balanced trees, given a node u, the number of updates between two consecutive rebalancing operations on u is $\Omega(w(u))$ [28, 76].*

3.4.1 Cores in Weight-Balanced Trees

The height of a $BB[\alpha]$-tree of size n is $H \leq \log_{1/(1-\alpha)}(n+1)$. Indeed, its root r has weight $w(r) = n+1$ and the deepest leaf f has weight $w(f) = 2$. Along the path from r to f, the weight of each node is at most $1-\alpha$ times the weight of its parent. Hence, by a simple induction, we have that $2 = w(f) \leq w(r) \cdot (1-\alpha)^{H-1} = (n+1) \cdot (1-\alpha)^{H-1}$. Thus, $H \leq \log_{1/(1-\alpha)}(n+1)/2 + 1 < \log_{1/(1-\alpha)}(n+1)$ as $1/(1-\alpha) < 2$. For a simplified notation, we ignore roundings when using α and logarithms. We use the following facts to obtain cores (Section 3.2.1).

Fact 4 *For a $BB[\alpha]$-tree of n nodes, the nodes on its topmost $\log_{1/\alpha}(n+1)$ levels form a complete balanced binary tree.*

Proof: Consider a shortest path from the root r of a $BB[\alpha]$-tree of n nodes to a null (*i.e.*, external) node x. Let ℓ be the number of nodes (including x itself) along this path: since $w(r) = n+1$ and $w(x) = 1$, we obtain that x should have weight at least $(n+1)\alpha^{\ell-1}$ by a simple induction on ℓ. Hence $(n+1)\alpha^{\ell-1} \leq w(x) = 1$. Thus the topmost $\ell - 1 \geq \log_{1/\alpha}(n+1)$ levels do not contain null nodes, and form a complete balanced binary tree. □

Remark 1 As for the core partition and its dynamic maintenance, we set $h^* = \log_{2/\alpha}(|T|+1)$ and observe that it forms a core by Fact 4 as $h^* \leq \log_{1/\alpha}(|T|+1)$ (Our choice of h^* will be clear when discussing the amortized analysis in Section 3.4.2). We also guarantee that $h^* \geq 1$, which

means $|T| \geq 2/\alpha - 1$, by fixing $r^* \geq 2/\alpha - 1$. In this way, when $|T| \geq r^*$, a core of height h^* surely exists by Fact 4, and when $|T| < r^*$ we stop the recursion on the subtree T (see Section 3.2.1). The term $f(\alpha) = 2/\alpha - 1$ is a decreasing function for increasing α, where $0 < \alpha < 1/2$, and it tends to $+\infty$ for $\alpha \to 0$. If we restrict to the range $2/11 < \alpha < 1/2$, we cover the interesting cases in the literature, and we have that $10 > f(\alpha) > 3$. Hence, it is always safe to choose $r^* \geq 10$ for $2/11 < \alpha < 1/2$.

We now show that we obtain a core partition (Definition 3) using the scheme described in Section 3.2.

Lemma 8 *For any $BB[\alpha]$-tree of size n with $2/11 < \alpha < 1/2$, the scheme of Section 3.2 with $h^* = \log_{2/\alpha}(|T|+1)$ and $r^* \geq 10$ successfully creates a core partition with $\gamma = (\log_{2/\alpha}(1-\alpha)+1) < 1$ and $c = 1$, where a core C_i at level i has size $|C_i| < (n+1)^{\frac{\gamma^i}{\log 2/\alpha}}$.*

Proof: We prove that the conditions of Definition 3 are met, using the following notation. Consider a tree T of size n and its topmost core C of height h^*. Also, consider a node u in $T \setminus C$ such that u's parent is in C. Let T_u denote the subtree rooted at u and C_u be the topmost core of T_u, where we denote the height of C_u by h^*_u (Note that C and C_u are consecutive in any path from the root to u or any of its descendants). We have that $\alpha^{h^*}(n+1) \leq |T_u| + 1 \leq (1-\alpha)^{h^*}(n+1)$ for the balance property of $BB[\alpha]$-trees.

The condition of Definition 3.1 is met as $h^*_u < \gamma h^*$ with $\gamma = (\log_{2/\alpha}(1-\alpha)+1) < 1$. Indeed, since $h^*_u = \log_{2/\alpha}(|T_u|+1) \leq \log_{2/\alpha}((1-\alpha)^{h^*}(n+1))$, we can rewrite the latter inequality as $h^*_u \leq h^* \log_{2/\alpha}(1-\alpha) + \log_{2/\alpha}(n+1)$. Replacing the last addend by h^*, we obtain $h^*_u \leq h^*(\log_{2/\alpha}(1-\alpha)+1)$, where $\gamma = (\log_{2/\alpha}(1-\alpha)+1) < 1$ as $1-\alpha < 1 < 2/\alpha$.

The condition of Definition 3.2 holds as, for any sequence of cores $C_1, C_2, \ldots, C_{t-1}, B_t$ traversed by a root-to-leaf path, there are $O(1)$ cores C_i's of size $|C_i| < r^*$. To see why, we first observe that $|C_i| \leq |C_{i-1}|$ for $2 \leq i \leq t-1$ by construction. Thus, let us consider C_{t-1}. If its size is greater than or equal to r^*, we have nothing else to prove. Otherwise, observe that the subtree T_{t-1} of which C_{t-1} is the topmost core, has size $|T_{t-1}| \geq r^*+1$ by the recursive scheme, and the height of C_{t-1} is $h^*_{t-1} = \log_{2/\alpha}(|T_{t-1}|+1) \geq \log_{2/\alpha}(r^*+2)$. Hence, $(r^*+2)^{1/\log(2/\alpha)} - 1 \leq 2^{h^*_{t-1}} - 1 = |C_{t-1}| < r^*$. Since $h^*_{t-2} > \gamma^{-1} \cdot h^*_{t-1}$ by Definition 3.1, an immediate induction on $j = 1, 2, \ldots$ gives that $|C_{t-1-j}| \geq |C_{t-1}|^{\gamma^{-j}} \geq ((r^*+2)^{1/\log(2/\alpha)} - 1)^{\gamma^{-j}}$ by transitivity.

Finding the largest j such that $((r^*+2)^{1/\log(2/\alpha)}-1)^{\gamma^{-j}} < r^*$ gives an upper bound on the maximum number of cores having size $< r^*$ in $C_1, C_2, \ldots, C_{t-1}, B_t$. From $((r^*+2)^{1/\log(2/\alpha)}-1)^{\gamma^{-j}} < r^*$,

we have $\gamma^{-j} < \log_{((r^*+2)^{1/\log(2/\alpha)}-1)} r^* = O(\log(2/\alpha))$, therefore, $j = O(\log_{1/\gamma} \log(2/\alpha)) = O(1)$, thus proving the condition. Finally, the claim on the size of the core at level i easily follows from the above discussion. □

Thereby, from Section 3.3, we have:

Corollary 1 *In the external-memory model, given a core partition for a $BB[\alpha]$-tree of size n with $2/11 < \alpha < 1/2$ and parameters $h^* = \log_{2/\alpha}(|T|+1)$ and $r^* = B$, where $B \geq 10$ is the block transfer size for the external memory, we can store its nodes in blocks of size B using the approach explained in Section 3.3.1, so that $O(n/B)$ blocks are occupied and any search path from the root to a node requires $O(\log_B n)$ I/Os and $O(\log n)$ comparisons.*

Corollary 2 *In the cache-oblivious memory model, given a core partition for a $BB[\alpha]$-tree of size $n \geq 1024$ with $2/11 < \alpha < 1/2$ and parameters $h^* = \log_{2/\alpha}(|T|+1)$ and $r^* = \log n$, we can use the memory layout explained in Section 3.3.2, so that $O(n/B)$ blocks are occupied and any search path from the root to a node requires $O(\log_B n)$ block transfers and $O(\log n)$ comparisons .*

3.4.2 Amortized Analysis for Repartitioning

We show that the size of a core is smaller than the size of its bottom subtrees. This is important to amortize the cost of core rescanevents.

Fact 5 *Consider a core C in a $BB[\alpha]$-tree T with $2/11 < \alpha < 1/2$ and parameters $h^* = \log_{2/\alpha}(|T|+1)$ and $r^* \geq 10$. Let z be a node in $T \setminus C$ such that z's parent is in C, and let T_z be the subtree rooted in z. Then, $|C| \leq |T_z|$.*

Proof: Let n be the size of T, which is rooted at the topmost node of the core C. Recalling that $|C| = 2^{h^*} - 1$, it suffices to prove that $2^{h^*} \leq |T_z| + 1$. Note that z is h^* levels below the root of T. Hence, we have that $w(z) = |T_z| + 1 \geq \alpha^{h^*}(n+1)$ by definition of balance in $BB[\alpha]$ trees. We show that $2^{h^*} \leq \alpha^{h^*}(n+1)$ to prove our claim. By taking the logarithms, we obtain $h^* \leq h^* \log\alpha + \log(n+1)$, namely, $h^*(1 - \log\alpha) \leq \log(n+1)$. By replacing h^* with $\log_{2/\alpha}(n+1)$, we obtain the inequality $\log_{2/\alpha}(n+1) \cdot (1 - \log\alpha) \leq \log(n+1)$, which is true since $\log 2/\alpha = 1 - \log\alpha$. □

Now we show how to amortize the cost of `repartition`(u) stated in Lemma 3, and focus on the three main events listed in Section 3.2.3. Let T_u be the subtree rooted at u, C_u be the core containing u, and T be the subtree having C_u as topmost core (so $T_u \subseteq T$).

- core resize: Let n_0 be the size of T when the last core resize occurred for C_u and n_1 be the size of T for the current core resize of C_u. Since the size changed, the height changed by 1 (for every increase or decrease of 1 in the height, we have one core resize), so $|\log_{2/\alpha}(n_0+1)-\log_{2/\alpha}(n_1+1)| \geq 1$. This implies that $|n_0-n_1| = \Omega(|T|)$, and thus so many fresh update operations below u can cover the cost.

- core rescan: By Fact 5, the size of C_u is upper bounded by that of a subtree of T_u, and so $|C_u| \leq |T_u|$, which means that this cost is absorbed by subtree rescan.

- subtree rescan: By Lemma 7, we can charge the $O(|T_u|)$ cost to $\Omega(w(u))$ fresh update operations below u as $w(u) = |T_u| + 1$.

From the discussion above, for each update operation, we can charge $O(\log n)$ credits for the running time and $O((\log n)/\min\{r^*, B\})$ credits for the cache complexity, with $O(1)$ credits (respectively, $O(1/\min\{r^*, B\})$ credits) to be used for each ancestor as illustrated above. Therefore, we have the following result.

Theorem 6 *For any $BB[\alpha]$-tree of size n with $2/11 < \alpha < 1/2$, its core partition with parameters $h^* = \log_{2/\alpha}(|T|+1)$ and $r^* \geq 10$ can be dynamically maintained with an amortized cost of $O(\log n)$ time and $O((\log n)/\min\{r^*, B\})$ block transfers per update operation.*

3.5 Case Study 2: AVL Trees

Height-balanced binary trees have the property that, for every node, the heights of the left and right subtrees differ at most by an integer value Δ [38, 67]. An AVL tree is the first data structure of this kind to be invented. In this section, as the second case study, we prove that AVL trees (as the most well known height-balanced binary trees) have also a successful core partition, with the same bounds in external-memory/cache-oblivious models, however, maintaining the core partition for AVL trees is more expensive as we show at the end of this section.

3.5.1 Cores in AVL Trees

We exploit the following folklore to define the cores in AVL trees.

Fact 6 *Consider an AVL tree of height H. Then, the nodes on its topmost $\lceil H/2 \rceil$ levels form a complete balanced binary tree.*

Proof: By induction on H. For $H = 1$ and $H = 2$, the property trivially holds as there is a single node in the topmost $\lceil H/2 \rceil$ levels. For the inductive step on $H \geq 3$, we assume that the property is true for any AVL tree of height $< H$, and we then use this assumption to prove the statement for height H. We consider two cases for the given AVL tree T.

Let H be even. Consider the left and right subtrees of T. Their height is at least $H - 2$, thus by the induction hypothesis, they are complete at least in the topmost $\lceil (H-2)/2 \rceil = H/2 - 1$ levels. Thus, considering the additional level of the root of T, we have that T is complete at least in the topmost $H/2 = \lceil H/2 \rceil$ levels.

Let now H be odd. The height of the left and right subtrees of T is at least $H - 2$, and by the induction hypothesis, they are complete at least in the topmost $\lceil (H-2)/2 \rceil = (H-1)/2$ levels. Thus, considering the additional level of the root of T, we have that T is complete at least in the topmost $(H-1)/2 + 1 = (H+1)/2 = \lceil H/2 \rceil$ levels. □

We fix $h^* = \lceil \log \sqrt{|T|} \rceil$, where as before, $|T|$ is the size of the AVL (sub)tree, and thus the top core has size $2^{h^*} - 1 < 2\sqrt{n}$. We also fix $r^* = 1$ for the sake of discussion, but other choices can be done. Note that the choice of $h^* = \lceil \log \sqrt{|T|} \rceil$ guarantees that the top tree is a core.

Fact 7 *For any AVL tree of height H with $n > 1$ nodes, the topmost $\lceil \log \sqrt{n} \rceil$ levels of nodes form a complete balanced binary tree.*

Proof: The claim immediately follows from Fact 6: we have $\lceil \log \sqrt{n} \rceil = \lceil \frac{1}{2} \log n \rceil \leq \lceil \frac{1}{2} \log(n+1) \rceil \leq \lceil H/2 \rceil$ as $H \geq \log(n+1)$. □

Resembling what happens in Section 3.4, we want to prove that the recursive scheme provides a core partition as stated in Definition 3. However, here we fix $c = 2$, meaning that the cores are exponentially decreasing by taking *every other core* in the root-to-leaf path, as shown next. (This is not true for $c = 1$, as it can be checked when the left subtree of the root is a complete balanced binary tree of height $H - 1$ and the right subtree is a Fibonacci tree of height $H - 2$.)

Lemma 9 *For any AVL tree with $n > 1$ node, the recursive scheme of Section 3.2 with $h^* = \lceil \log \sqrt{|T|} \rceil$ and $r^* \geq 1$ successfully creates a core partition with $\gamma = 2/3$ and $c = 2$, where a core C_i at level i has size $|C_i| < (2\sqrt{n})^{(\frac{2}{3})^{\lfloor (i-1)/2 \rfloor}}$.*

The proof of Lemma 9 relies on the following properties of cores in the AVL tree (see Definition 3.1).

Lemma 10 *Let $C_1, C_2, \ldots, C_{t-1}$ be the cores traversed in any root-to-leaf path for $t > 1$. Then, the following properties hold:*

- $|C_2| \leq |C_1| < 2\sqrt{n}$*;*
- $|C_i| < |C_{i-2}|^{\frac{2}{3}}$ *for* $|C_{i-2}| > 1$ *and* $3 \leq i \leq t-1$*.*

Proof: We know that $|C_1| < 2\sqrt{n}$ by our choice of h^*, for the topmost core, $h^* = \lceil \log \sqrt{n} \rceil$ when $n > 1$; if C_2 exists, it cannot have more descendants than C_1, so it is $|C_2| \leq |C_1|$ by construction. In general, note that $|C_i| \leq |C_{i-1}|$ for $3 \leq i \leq t-1$ where $3 \leq i \leq t-1$: Let T_i denote the subtree rooted at the topmost node of C_i. As $|T_i| < |T_{i-1}|$, it follows that $|C_i| = 2^{\lceil \log \sqrt{|T_i|} \rceil} - 1 \leq 2^{\lceil \log \sqrt{|T_{i-1}|} \rceil} - 1 = |C_{i-1}|$.

We now prove that $|C_i| < |C_{i-2}|^{\frac{2}{3}}$ for $|C_{i-2}| > 1$ and $3 \leq i \leq t-1$. Let $h_i^* = \lceil \log \sqrt{|T_i|} \rceil$ denote the height of core C_i, and H_i denote the height of subtree T_i. First suppose that

$$h_{i-1}^* \geq h_i^* + 1 \tag{3.1}$$

holds. (We will show later how to deal when (3.1) does not hold.) Also, observe that our choice of h_i^* and (the proof of) Fact 7 imply that

$$H_i \geq 2h_i^* - 1 \tag{3.2}$$

Since $H_{i-2} = h_{i-2}^* + h_{i-1}^* + H_i$, we can use (3.1) and (3.2) to bound the height of T_{i-2} as

$$H_{i-2} \geq h_{i-2}^* + 3h_i^* \tag{3.3}$$

We are ready to prove that $|C_i| < |C_{i-2}|^{\frac{2}{3}}$ for $|C_{i-2}| > 1$. By contradiction, suppose $|C_i| \geq |C_{i-2}|^{\frac{2}{3}}$. This is equivalent to say that $2^{\lceil \log \sqrt{|T_i|} \rceil} - 1 \geq (2^{\lceil \log \sqrt{|T_{i-2}|} \rceil} - 1)^{\frac{2}{3}}$. Since $(x-1)^{\frac{2}{3}} \geq x^{\frac{2}{3}} - 1$ for $x \geq 1$, we obtain that $2^{\lceil \log \sqrt{|T_i|} \rceil} \geq (2^{\lceil \log \sqrt{|T_{i-2}|} \rceil})^{\frac{2}{3}}$ and so $\lceil \log \sqrt{|T_i|} \rceil \geq \frac{2}{3} \times \lceil \log \sqrt{|T_{i-2}|} \rceil$. That is,

$$h_i^* \geq \frac{2}{3} \times \lceil \log \sqrt{|T_{i-2}|} \rceil \tag{3.4}$$

Recalling from [57, p.460] that $H_{i-2} \leq 1.4404 \times \log(|T_{i-2}| + 2) - 0.3277$, we obtain from (3.3) and (3.4) that $1.4404 \times \log(|T_{i-2}| + 2) - 0.3277 \geq h_{i-2}^* + 3h_i^* \geq \lceil \log \sqrt{|T_{i-2}|} \rceil + 2 \times \lceil \log \sqrt{|T_{i-2}|} \rceil \geq 3 \times \log \sqrt{|T_{i-2}|}$. But we have a contradiction for $|T_{i-2}| \geq 8$, since the inequality $1.4404 \times \log(|T_{i-2}| + 2) - 0.3277 \geq \frac{3}{2} \times \log |T_{i-2}|$ does not hold in this cases. Hence, we can conclude that $|C_i| < |C_{i-2}|^{\frac{2}{3}}$

for $|T_{i-2}| \geq 8$. As for $|T_{i-2}| < 8$, there are a small finite number of cases to examine, so we directly prove for them that $|C_i| < |C_{i-2}|^{\frac{2}{3}}$ when $|C_{i-2}| > 1$: we inspect them case by case as at the end of this proof.

It remains to discuss when the inequality in (3.1) does not hold (whereas $h^*_{i-1} \geq h^*_i$ is always true). Its purpose is to guarantee that (3.3) holds. However, since $H_{i-2} = h^*_{i-2} + H_{i-1}$, we observe that $H_{i-1} \geq 3h^*_i$ also implies that (3.3) holds, and so we are done. Consequently, it suffices to discuss the case when $h^*_{i-1} = h^*_i$ and $H_{i-1} \leq 3h^*_i - 1$. In the following, we use h as the shorthand for both h^*_i and h^*_{i-1}, since they are equal. Also, observe that $H_{i-1} = h^*_{i-1} + H_i = h + H_i \geq 3h - 1$ by (3.2), thus implying that $H_{i-1} = 3h - 1$. This gives a precise scenario: both C_{i-1} and C_i are of height h, the height of T_{i-1} is $3h - 1$, and the height of T_i is $2h - 1$ (so it extends by further $h - 1$ levels below C_i).

We prove that it not possible to have $h > 3$ in this scenario. For the given $h = \lceil \log \sqrt{x} \rceil$, we observe by a simple induction that the feasible range of values for x are $2^{2(h-1)} < x \leq 2^{2h}$. Hence, $2^{2(h-1)} < |T_i| < |T_{i-1}| \leq 2^{2h}$. Also, the latter two trees cannot have less nodes than the Fibonacci trees of their same height, $|T_i| \geq F_{2h+1} - 1$ and $|T_{i-1}| \geq F_{3h+1} - 1$, formulated in terms of Fibonacci numbers (recalling that the Fibonacci tree of height k has $F_{k+2} - 1$ nodes [57, p.460]).

We are now ready to state a necessary condition that excludes the cases for $h > 3$. The quantity $2^{2h} - (F_{3h+1} - 1)$ represents an upper bound on the number of nodes that can be added to $|T_{i-1}|$ without increasing its height $3h - 1$. Then T_i cannot contain too many nodes, namely, $|T_i| \leq (F_{2h+1} - 1) + [2^{2h} - (F_{3h+1} - 1)]$: starting from the minimal number $F_{2h+1} - 1$ of nodes for its height $2h - 1$, we cannot add more than $2^{2h} - (F_{3h+1} - 1)$ nodes since T_i is a subtree of T_{i-1}. Also, $2^{2(h-1)} < |T_i|$ as previously discussed. Putting all together, we obtain that $2^{2(h-1)} < |T_i| \leq (F_{2h+1} - 1) + 2^{2h} - (F_{3h+1} - 1)$, producing the necessary condition $2^{2(h-1)} < (F_{2h+1} - 1) + 2^{2h} - (F_{3h+1} - 1)$, which can be equivalently stated as

$$\frac{3}{4} \times 4^h - F_{3h+1} + F_{2h+1} > 0 \tag{3.5}$$

Note that the condition in (3.5) is satisfied only when $h \leq 3$. There are just a small finite number of cases for $h \leq 3$, so we directly prove for them that $|C_i| < |C_{i-2}|^{\frac{2}{3}}$ for $|C_{i-2}| > 1$ as follows. Here by direct case inspection that $|C_i| < |C_{i-2}|^{\frac{2}{3}}$ when $|C_{i-2}| > 1$ and $|T_{i-2}| < 8$. Note that $|C_{i-2}|$ is a power of two minus 1, so the latter conditions and the choice of h^*_{i-2} imply that $|C_{i-2}| = 3$. Given this, the only feasible choices are $|T_{i-2}| \in [5 \dots 7]$, and thus we have a very small number of feasible situations. Indeed, $|T_{i-1}| \leq |T_{i-2}| - |C_{i-2}| \leq 4$. Hence, $h^*_{i-1} = 1$ and $|C_{i-1}| = 1$. This

immediately implies that $|C_i| \leq |C_{i-1}| = 1 < 3^{\frac{2}{3}} = |C_{i-2}|^{\frac{2}{3}}$, thus proving the first claim.

We also prove here by direct case inspection for $1 \leq h \leq 3$ that $|C_i| < |C_{i-2}|^{\frac{2}{3}}$ for $|C_{i-2}| > 1$ when $h = h^*_{i-1} = h^*_i$ and $H_{i-1} = 3h - 1$. Recall that $|C_{i-2}| > 1$ is equivalent to $|C_{i-2}| \geq 3$ and so $h^*_{i-2} \geq 2$, thus proving the second claim. Case $h = 1$. Simply put, $|C_i| = 1 < 3^{\frac{2}{3}} \leq |C_{i-2}|^{\frac{2}{3}}$. Case $h = 2$. Since $H_{i-2} = h^*_{i-2} + H_{i-1} = h^*_{i-2} + 3h - 1 \geq 7$, the subtree T_{i-2} cannot have less nodes than those (33) of the Fibonacci tree of height 7, so $|T_{i-2}| \geq 33$. This implies that $|C_{i-2}| = 2^{\lceil \log \sqrt{|T_{i-2}|} \rceil} - 1 \geq 2^{\lceil \log \sqrt{33} \rceil} - 1 = 7$. Thus, $|C_i| = 2^h - 1 = 3 < 7^{\frac{2}{3}} \leq |C_{i-2}|^{\frac{2}{3}}$. Case $h = 3$. We first prove that $h^*_{i-2} \geq 4$. Since $h^*_{i-2} \geq h^*_{i-1} = h$, we have $H_{i-2} = h^*_{i-2} + 3h - 1 \geq 4h - 1 = 11$. The subtree T_{i-2} cannot have less nodes than those (232) of the Fibonacci tree of height 11, so $|T_{i-2}| \geq 232$ and $h^*_{i-2} = \lceil \log \sqrt{|T_{i-2}|} \rceil \geq \lceil \log \sqrt{232} \rceil = 4$. Next, we give a better bound on the height of T_{i-2} as $H_{i-2} = h^*_{i-2} + 3h - 1 \geq 12$. The Fibonacci tree of height 12 has 376 nodes, and so $|C_{i-2}| = 2^{\lceil \log \sqrt{|T_{i-2}|} \rceil} - 1 \geq 2^{\lceil \log \sqrt{376} \rceil} - 1 = 31$. Thus, $|C_i| = 2^h - 1 = 7 < 31^{\frac{2}{3}} \leq |C_{i-2}|^{\frac{2}{3}}$. □

We also need to prove that there are few small cores (see Definition 3.2). This follows the same path as we did at the end of the proof of Lemma 8, thus showing that AVL admits a core partition with $\gamma = 2/3$ and $c = 2$. Thus, let us consider C_{t-1} when its size is $< r^*$, and observe that the subtree T_{t-1} of which C_{t-1} is the topmost core, has size $|T_{t-1}| \geq r^* + 1$, and the height of C_{t-1} is $h^*_{t-1} = \lceil (1/2) \log_2 |T_{t-1}| \rceil \geq (1/2) \log_2 (r^* + 1)$. Hence, $\sqrt{r^* + 1} - 1 \leq |C_{t-1}| < r^*$. An immediate induction on $j = 1, 2, \ldots$ gives that $|C_{t-1-2j}| \geq |C_{t-1}|^{(3/2)^j} \geq (\sqrt{r^* + 1} - 1)^{(3/2)^j}$ by transitivity.

Finding the largest j such that $(\sqrt{r^* + 1} - 1)^{(3/2)^j} < r^*$ gives an upper bound on the maximum number of cores having size $< r^*$ in $C_1, C_2, \ldots, C_{t-1}, B_t$. From $(\sqrt{r^* + 1} - 1)^{(3/2)^j} < r^*$, we have $(3/2)^j < \log_{(\sqrt{r^*+1}-1)} r^*$, therefore, $j = O(\log_{3/2} \log_{(\sqrt{r^*+1}-1)} r^* = O(1)$, thus proving the condition in Definition 3.2. Therefore, the external-memory model and cache-oblivious memory model of AVL trees as explained in Section 3.3 are available with the same bounds. In the following, we study its amortized cost of update.

3.5.2 Amortized Analysis for Repartitioning

We prove that an amortized (poly)logarithmic cost cannot be achieved for maintaining a core partition of AVL trees, contrarily to the case of weight-balanced trees as discussed in Section 3.4.2. For any $n \geq 2$, we can produce a sequence of n insertions into an initially empty AVL tree with $\Omega(n)$ rotations. The cost of these operations is dominated by the `repartition` operations. In particular, the total cost of the corresponding subtree rescans is a lower bound for the amortized cost of the sequence of n insertions. Thus we prove that the latter cost alone prevents from

obtaining a poly-logarithmic amortized cost.

Lemma 11 *Given any AVL tree of height h, its height can be increased by one with at most F_{h+2} insertions*

Proof: By induction on h, the base case is a unary node of height 1, and thus its height becomes 2 by a $F_1 = 1$ insertion that replaces one of the missing child by a leaf. For the inductive case, suppose that the height $k < h$ of an AVL tree can be increased with at most F_{k+2} insertions. Let x be the root of the AVL tree of height h, and observe that x's children either have the same height or their heights differs by one. If x has two children of same height $k = h - 1$, we can increase the height of one of them by induction, and thus this increases the height of the AVL tree by one with $F_{k+2} < F_{h+2}$ insertions. If x has one child y of height $h - 1$ and another one z of height $h - 2$, we first perform F_h insertions into the subtree rooted at z to increase its height by one and then perform F_{h+1} insertions into the subtree rooted at y to increase its height by one (using inductive hypothesis twice with $k = h - 2$ and $k = h - 1$). These insertions are at most $F_h + F_{h+1} = F_{h+2}$ in number, and increase the height of the AVL by one. □

Theorem 7 *The amortization cost for* subtree rescan *is $\Omega((\frac{2}{\phi})^{\log n})$, where $\phi = \frac{1+\sqrt{5}}{2} < 2$ is the golden ratio, and thus* subtree rescans *for AVL trees cannot be amortized in poly-logarithmic time.*

Proof: We provide a counterexample for a tree T of height $h + 1$ whose left subtree is a complete balanced binary tree of height h, named $\mathcal{B}$, and the right subtree (right sibling of $\mathcal{B}$) is an arbitrary AVL tree of height h. Let $P(\mathcal{B})$ and $\mathcal{B}^{sib}$ denote the parent of $\mathcal{B}$ (initially the root of the tree) and the right sibling of $\mathcal{B}$. We apply Lemma 11 "twice" to $\mathcal{B}^{sib}$ to increase its height by 2: first we increase its height from h to $h + 1$ by at most F_{h+2} insertions, then we increase its height from $h + 1$ to $h + 2$ by at most another F_{h+3} insertions. This makes its height $h + 2$, which in turn causes a rotation on the tree to make it balanced. Because of the rotation, $P(\mathcal{B})$ and $\mathcal{B}^{sib}$ change and move one level below. By definition, now $P(\mathcal{B})$ and $\mathcal{B}^{sib}$ denote to new parent and sibling of $\mathcal{B}$, thus, $P(\mathcal{B})$ and $\mathcal{B}^{sib}$ will be again of height $h + 1$ and h, respectively.

We repeatedly apply Lemma 11 "twice" to $\mathcal{B}^{sib}$ to increase its height by 2 (from h to $h + 2$). Each time this height increases, it causes a rotation on $P(\mathcal{B})$ and produces new $P(\mathcal{B})$ and $\mathcal{B}^{sib}$ of height $h + 1$ and h. In each rotation, $\mathcal{B}$ is involved in the subtree rescan, so the cost of the rotation is at least 2^h. The number of insertions needed to generate this rotation at each iteration is at most $F_{h+2} + F_{h+3} = F_{h+4}$. If we let $n' = 2^h$ and do $\frac{n'}{F_{h+4}}$ iterations, the total number of insertions

is $n = O(\frac{n'}{F_{h+4}} \times F_{h+4}) = O(n')$. But the total cost of subtree rescan is $\Omega(n' \times \frac{n'}{F_{h+4}}) = \Omega(n(\frac{2}{\phi})^{\log n})$ where ϕ is the golden ratio. □

3.6 Summary

In this chapter, we presented the *core partitioning scheme*, which maintains a balanced search tree as a dynamic collection of complete balanced binary trees called *cores*, we preserve the original topology and algorithms of the given balanced search tree using a simple post-processing with guaranteed performance to completely rebuild the changed cores (possibly all of them) after each update. By applying core partitioning scheme on a given balanced binary search tree, if it is a *successful core partition*, simultaneously dynamic memory allocation, cache-obliviousness, and efficient I/O complexities in external-memory and cache-oblivious models are provided occupying $O(n/B)$ blocks of memory. Amortized cost for update depends on the type of the given balanced binary search tree since they differ on the rebalancing operations they use.

As case studies we applied core partitioning to weight-balanced trees and height-balanced trees (AVL trees). We had shown that they have successful core partition, thus simultaneously achieve good memory allocation, space-efficient representation, and cache-obliviousness. For AVL trees, the logarithmic amortization of insertion/deletion is impossible (*i.e.*, AVL trees require super poly-logarithmic cost by a lower bound on the subtree size of the rotated nodes), while weight-balanced trees can be maintained with a logarithmic cost.

Chapter 4

Core Partitioning Directly on Plain Binary Trees

We introduced the notion of core partition in Chapter 3 to show how to obtain cache-efficient versions of classic balanced binary search trees such as AVL trees and weight-balanced trees. Looking at weight-balanced tree in Chapter 3 that are kept balanced using local rebuilding to "simulate" rotations (see Section 3.4), we observe that a subtree is rescanned for two reasons.

- Its root u is unbalanced and we perform local rebuilding.
- Its top core needs core resize and we have to maintain the core partition without changing the underlying topology.

This seems an interesting challenge to investigate: what if we use the core partition on plain binary search trees? In other words, what if we maintain the tree balanced just by using core partitioning without extra rebalancing operations such as rotations? An objection is that they do not have a core large enough. However, we can use an "aggressive" version of core resize, so that when we maintain the core partition, we also transform the subtree in a perfectly balanced tree as in Overmars' partial rebuilding [76, Sect.4.2].

Now let us reformulate the challenge: take an empty plain binary search tree and, whenever core resize happens, transform the subtree in a perfectly balanced tree. We only operate the above, no rebalancing is performed.

It can be easily observed that we can get $O(\log^2 n)$ height using only the aggressive version of core resize. However, this is not so interesting, as the same bound can be obtained with the

logarithmic rebuilding method using a logarithmic number of sorted arrays. Can we get $O(\log n)$? In this chapter we give a positive answer to this question.

We also adopt the external-memory model [2] and cache-oblivious model [39, 81] explained in Chapter 2 to evaluate the I/O complexity. Note that B is an *unknown* parameter of the block size for cache-oblivious model. We introduce a new data structure, called *Cache-Oblivious General Balanced Tree* (*COG-tree*) which guarantees logarithmic search time and logarithmic amortized insert time. We use *cores* as explained in Chapter 3, so we keep the first levels of each node of the tree in the form of a complete balanced binary tree. As we will see this property can be also maintained after an insertion operation in a very simple way and in logarithmic amortized time for the proper values of the number of full levels. The latter property allows us to efficiently use the new structure for cache-oblivious model.

Our proposal exhibits good performances that is a COG-tree of n nodes requires $O(\log_B n)$ I/Os amortized for updates, and $O(\log_B n)$ I/Os in the worst case for searches. In addition, it can be laid out in $O(n)$ space, external pointers in our data structures occupies only $O(n)$ bits in total. These complexities are theoretically optimal, and our structure compares optimally with respect to the previous ones. It obtains the same optimal results with respect to Bender *et al.* 's Cache-Oblivious B-Trees in [20, 23] and Bender *et al.* 's exponential structures in [21].

In this chapter, we start with the preliminaries of COG-trees in Section 4.1, the definition of COG-trees is given in Section 4.2, we describe the basics of its memory management in Section 4.3, then we show how to maintain COG-trees in the external-memory and the cache-oblivious memory models in Sections 4.4 and 4.5. Finally in Section 4.6 we compute the amortization cost of the update.

4.1 Preliminaries and Notation

As mentioned in Chapter 2, most binary search trees require storing data (*e.g.* 'colors', 'weights', 'rank', *etc.*) on the nodes of the tree and checking at each update that some constraints on the structure of the tree are maintained. This information must be dynamically updated after insertions and deletions. A different approach is to let the tree have any shape as long as its height is logarithmic. In this way, there is no need of storing and checking the balance information, but it is sufficient to check whether the maximal possible height has been exceeded. Trees of this kind, called *General Balanced Trees*, introduced by [8] and later rediscovered by [41] under the name of *scapegoat trees*, can be efficiently maintained and require as additional space only

that for the pointers. They are restructured with an operation, called *partial rebuilding*, that transforms a subtree of the tree in a perfectly balanced tree. The operation is expensive having a cost proportional to the number of the nodes of the subtree, but performs rarely hence has a low amortized cost.

Our *Cache-Oblivious General Balanced Tree* (*COG-tree*) data structure is a smooth extension of general balanced trees [8, 41] and guarantees the same logarithmic search time and logarithmic amortized insertion time in the comparison model as the general balanced trees, however, our data structures also performs efficiently in the external-memory/cache-oblivious models. Our structure, uses cores as presented in Chapter 3 (keeps the first levels of each node of the tree in the form of a complete balanced binary search tree) and we maintain this property after the insertion operation in a very simple way and in logarithmic amortized time for the proper values of the number of full levels. The latter property allows us to efficiently use the new structure for cache-oblivious model.

For a given node u, we use T_u to denote the subtree of T rooted at u, $s(u) = |T_u|$ the subtree size, and $h(u)$ the height of T_u. A perfectly balanced tree of n nodes satisfies the property that its height is the minimal possible, namely, its height is equal to $\lceil \log(n+1) \rceil$.

In the core partitioning scheme introduced in Chapter 3, we say that T has a *core* of height h^*, if its topmost h^* levels are full. In Chapter 3, in Figure 3.1 an example of core partition was given. Recall from Chapter 3, if every nonempty subtree of size larger than r^* has a core of height h^*, where h^* is a function of the size or the height (or both) and r^* is a function of the size of the entire tree or the block size, we say that T has a *successful core partition* if it satisfies the conditions below.

1. Any root-to-leaf path in T traverses cores of doubly exponentially decreasing size.
2. Only a constant number of the above cores are of small size less than or equal to r^*.

In the rest of the chapter, we present an algorithm on plain binary search tree so that it guarantees having a successful core partition in logarithmic amortized cost.

4.2 Definition of COG-Tree

In this section, we study whether also *plain* binary search trees can benefit of the idea of the core partition, we explore how to employ the core partition idea to make these trees balanced and cache-oblivious. In the following, we start with defining two simple invariants for our new structure and we show that both of them are required to obtain efficient results. To show that, we use a

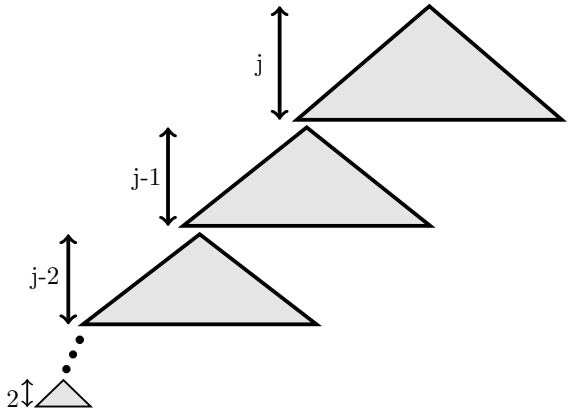


Figure 4.1: Q_j, a tree with core-fullness invariant and height $\Theta(\log^2 |Q_j|)$.

simple fact that the resulting tree must have logarithmic height and cores of "sufficiently large" heights in a way that cache-obliviousness can be achieved, however, we later simplify them to a single invariant satisfying both.

Although, the minimal height of binary search tree T rooted at u is $\lceil \log(s(u)+1) \rceil$, we can show that a perfectly balanced tree of $s(u)$ nodes is full up to its first $\lceil \log(s(u)+2) \rceil - 1$ levels (by an induction on the size). We define $h_{\min}(u) = \lceil \log(s(u)+2) \rceil$ as an asymptotic minimal height of T so that we can enforce cores in T by requiring the following condition to be maintained by partial rebuilding.

{Core-fullness invariant} If the tree is non-empty, the first $h_{\min}(u) - 1$ levels of T are full. Also the same property holds for all the non-empty subtrees below the $(h_{\min}(u) - 1)^{th}$ level of T, in a recursive manner[1].

Unfortunately, the condition alone does not guarantee to obtain a structure with efficient core partition. In particular, the height of the tree can be more than logarithmic, as shown next.

For a given positive integer j, let Q_j be the tree constructed as follows. Q_j is built starting from a complete balanced binary tree of height j, where we replace one of the null pointers at the bottom of the tree with another complete balanced binary tree of height $j-1$; again one null pointer of this second tree is replaced with another complete balanced binary tree of height $j-2$, and so on till we reach a binary tree of height 2. This tree is shown in Figure 4.1.

Lemma 12 *Q_j satisfies the core-fullness invariant but it has a height of $\Theta(\log^2 |Q_j|)$.*

[1]Roughly speaking, this condition can be considered as an "aggressive" version of the core resize operation.

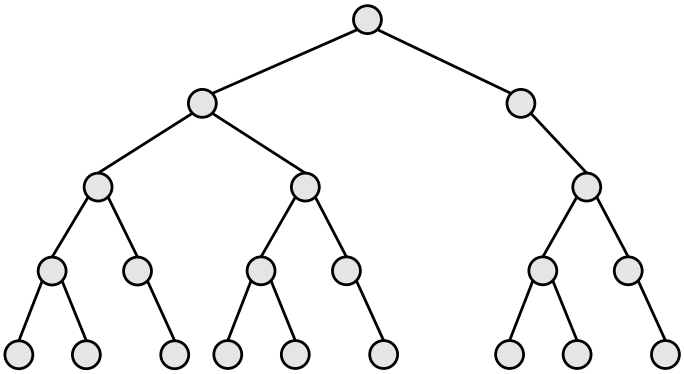


Figure 4.2: An example of P_2.

Proof: The overall size of this tree is given by $|Q_j| = (2^j-1)+(2^{j-1}-1)+(2^{j-2}-1)+....+(2^2-1) = 2^{j+1}-2-(j-1) = 2^{j+1}-j-1$. Therefore, $h_{\min}(u) = \lceil \log(2^{j+1}-j-1+2) \rceil = j+1$, and $h_{\min}(u)-1 = j$, and by construction, the first j levels are full, this also holds recursively for the rest of the tree. Therefore, Q_j satisfies the core-fullness invariant.

On the other hand, the overall height of Q_j is given by $h = j + (j-1) + (j-2) + \ldots + 2 = j(j+1)/2 - 1 = \Theta(j^2) = \Theta(\log^2 |Q_j|)$. □

We can enforce to have a logarithmic height in T by requiring the following condition to be maintained by partial rebuilding.

{Height invariant} For every node u in T, $h(u) \leq c\, h_{\min}(u)$ for some constant $c \geq 1$.

Note that the above condition has been employed several times in different forms, including for general balanced trees [8] or scapegoat trees [41]. However, it is not sufficient alone to get an efficient core partition as shown next. Indeed, the height invariant does not necessarily imply the presence of the cores of sufficiently large height, as the following counterexamples show. The first counterexample is simply given by a tree composed of a root with a complete balanced binary tree as left subtree and null as right subtree. In this case, it is easy to observe that the height invariant is satisfied, but there is no core involving the root. As a more general counterexample, we construct a tree with the following structure. We start from a complete balanced binary tree of height 3, and we arbitrary remove one leaf. Let P_1 denote this tree. Starting from P_1, we recursively construct a tree P_{i+1} substituting each of the three leaves of P_1 with one subtree P_i. Figure 4.2 illustrates an example of P_2.

Lemma 13 *P_i satisfies the height invariant for $c \geq 2$, but it cannot be partitioned into sufficiently large cores so that cache-obliviousness can be obtained.*

Proof: By construction, it is obvious that, for any i, P_i does not contain cores with height greater than three. Now, consider any node u in P_i. To show that the height invariant is satisfied, we need to prove that the height $h(u)$ of the subtree T_u rooted at u is less or equal to $2h_{\min}(u)$. Observe that u is either the root or one of the children of the root of a $P_{\lfloor h(u)/2 \rfloor}$ tree. In both cases, $|T_u| \geq |P_{\lfloor h(u)/2 \rfloor - 1}| + 1$. Thus, $h_{\min}(u) \geq \log |P_{\lfloor h(u)/2 \rfloor - 1}|$. On the other hand, it is easy to observe that $|P_i| = 3|P_{i-1}| + 3$, and therefore $|P_i| = 3^i + \frac{3^{i+1}-1}{2} - 1$.

Hence,

$$\begin{aligned} h_{\min}(u) &\geq \log |P_{\lfloor h(u)/2 \rfloor - 1}| \\ &\geq \log(3^{\lfloor h(u)/2 \rfloor - 1} + \frac{3^{\lfloor h(u)/2 \rfloor} - 1}{2} - 1) \\ &\geq h(u)/2\,. \end{aligned}$$

Therefore, $h(u) \leq 2h_{\min}(u)$ and the height invariant is satisfied. □

We conclude that both core-fullness invariant and height invariant are necessary for our data structure to be efficient in the cache-oblivious memory model, however, to have a simpler data structure, in the following, we present a condition called *fullness invariant*, then we show that it satisfies both the core-fullness and the height invariants.

{Fullness invariant} The first $h_{\min}(u) - 1$ levels of every node u in T are full.

The only difference between the fullness invariant and the core-fullness invariant is that in the first one, the first $h_{\min}(u) - 1$ levels of **every** node in T are full. Clearly the fullness invariant is stronger and it satisfies the core-fullness invariant. In Lemma 16, we prove that it also satisfies the height invariant. Therefore, it is a much simpler candidate for us to build our data structure upon on (however, it can be shown that the same results are also achievable if one is interested to apply the core-fullness and the height invariants, instead).

Lemma 14 *For any node v in any binary search tree T,*

$$2^{h_{min}(v)-1} - 2 < s(v) \leq 2^{h_{min}(v)} - 2.$$

Proof: Recall that by definition of $h_{min}(v)$, $h_{min}(v) = \lceil \log(s(v)+2) \rceil$. Therefore,

$$\log(s(v)+2) \leq h_{min}(v) < \log(s(v)+2) + 1$$
$$h_{min}(v) - 1 < \log(s(v)+2) \leq h_{min}(v)$$
$$2^{h_{min}(v)-1} - 2 < s(v) \leq 2^{h_{min}(v)} - 2\,.$$

□

Corollary 3 *Let v be a node in tree T with fullness invariant, the number of the nodes in the first $h_{min}(v) - 1$ levels of T_v is greater than or equal to $s(v)/2$.*

Proof: The number of the nodes in the first $h_{min}(v) - 1$ levels is $2^{h_{min}(v)-1} - 1$. On the other hand, by the previous lemma, $2^{h_{min}(v)} - 2 \geq s(v)$, therefore, $2^{h_{min}(v)-1} - 1 \geq s(v)/2$. □

Lemma 15 *In a tree T with fullness invariant, if f is a leaf and $(f = u_0), u_1, u_2, \ldots, (u_\ell = root)$ is the path from f to the root, for $0 \leq i \leq \ell - 4$,*

$$h_{min}(u_i) \leq h_{min}(u_{i+4}) - 1.$$

Proof: Observe that $h_{min}(u_i)$ is a nondecreasing function (when i is increasing), since $s(u_i)$ is an increasing function. Now consider $T_{u_{i+1}}$, its first $h_{min}(u_{i+1}) - 1$ levels are full. The number of the nodes in $T_{u_{i+1}}$'s first $h_{min}(u_{i+1}) - 1$ levels is $2^{h_{min}(u_{i+1})-1} - 1$ which $2^{h_{min}(u_{i+1})-2} - 1$ nodes are in T_{u_i} and $2^{h_{min}(u_{i+1})-2} - 1$ nodes are in $T_{sib(u_i)}$ (where $sib(v)$ denotes the sibling of node v), hence, $2^{h_{min}(u_{i+1})-2}$ nodes (including u_{i+1}) are completely **disjoint** from the nodes in T_{u_i}. Similarly, for $j = 2, 3, \text{and } 4$, for $T_{u_{i+j}}$, its first $h_{min}(u_{i+j}) - 1$ levels are full and $2^{h_{min}(u_{i+j})-2}$ of the nodes in those levels are completely disjoint from $T_{u_{i+(j-1)}}$. Therefore,

$$\begin{aligned}
s(u_{i+4}) &\geq s(u_i) + 2^{h_{min}(u_{i+1})-2} + 2^{h_{min}(u_{i+2})-2} + 2^{h_{min}(u_{i+3})-2} + 2^{h_{min}(u_{i+4})-2} \\
&\geq s(u_i) + 4 \times 2^{h_{min}(u_i)-2} \quad (\text{since } h_{min}(u_i) \text{ is a nondecreasing function}) \\
&\geq s(u_i) + 2^{h_{min}(u_i)} \\
&\geq 2s(u_i) + 2 \quad (\text{by the definition of } h_{min}(u_i))\,.
\end{aligned}$$

On the other hand,

$$\begin{aligned} h_{min}(u_{i+4}) &= \lceil \log(s(u_{i+4}) + 2) \rceil \\ &\geq \lceil \log(2s(u_i) + 2 + 2) \rceil \\ &\geq \lceil \log 2 + \log(s(u_i) + 2) \rceil \\ &\geq h_{min}(u_i) + 1 \,. \end{aligned}$$

Hence the proof is complete. □

Lemma 16 *The fullness invariant satisfies the height invariant for $c \geq 4$.*

Proof: Let u be a node in tree T with fullness invariant, to show that its height is upper bounded by $4h_{min}(u)$, let f be the deepest leaf of T_u and $(f = u_0), u_1, u_2, \ldots, (u_\ell = u)$ be the path from f to u. By Lemma 15, $h_{min}(u_i) \leq h_{min}(u_{i+4}) - 1$, for $0 \leq i \leq \ell - 4$. Therefore, since $h_{min}(f) = \lceil \log(1 + 2) \rceil = 2$, $\ell \leq 4(h_{min}(u) - 1)$, so $h(u) = \ell + 1 \leq 4h_{min}(u)$, hence, the height invariant is satisfied for $c \geq 4$. □

Fortunately, the fullness invariant guarantees the existence of a core partition as we will observe in the following theorem.

Theorem 8 *If a binary tree T satisfies the fullness invariant, it also admits a successful core partition with $r^* \geq 1$ and $h^* = h_{\min}(u) - 1$ for the core rooted at u.*

Proof: We prove that condition 1 of the core partition in Section 4.1 holds. Consider any two cores C and C' such that C' is a child of C. Let h and h' denote the heights of C and C', respectively. Observe that $|C| \geq |C'|$ and $|C'| = 2^h - 1$ and $|C'| = 2^{h'} - 1$. As shown in Figure 4.3, let $u_0, u_1, u_2, \ldots, u_\ell$ show the path from the root of C' to the root of C (u_0 is the root of C' and u_ℓ is the root of C). Since $h \geq h'$, $\ell \geq h'$. On the other hand, in Lemma 15, we had shown that $h_{min}(u_{i+4}) \geq h_{min}(u_i) + 1$, for $0 \leq i \leq \ell - 4$. Therefore, $h_{min}(u_\ell) \geq h_{min}(u_0) + h'/4$. By the definition of h^*, $h = h_{min}(u_\ell) - 1$ and $h' = h_{min}(u_0) - 1$. Hence,

$$\begin{aligned} h &= h_{min}(u_\ell) - 1 \\ &\geq h_{min}(u_0) + h'/4 - 1 \\ &\geq h' + h'/4 \geq \frac{5}{4} h' \,. \end{aligned}$$

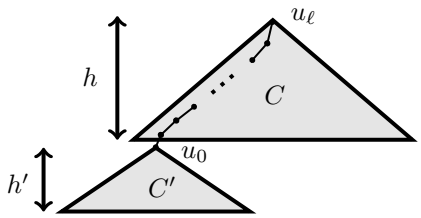


Figure 4.3: Two consecutive cores C and C' in a search path.

Therefore, $|C'|+1 \leq (|C|+1)^{\frac{4}{5}}$, so the cores are doubly exponentially decreasing in a root-to-leaf path.

To show that also condition 2 holds, in any root-to-leaf path, let $C_1, C_2, \ldots, C_{t-1}, B_t$ be the traversed cores and the terminal-core at the bottom of the search. Let us consider C_{t-1} when its size is less than or equal to r^*, and observe that the subtree T_{t-1} of which C_{t-1} is the topmost core, has size $|T_{t-1}| > r^*$, so $|C_{t-1}|+1 > 2^{\lceil \log(r^*+2)\rceil - 1} > 2^{\log(r^*)-1} > \frac{1}{2}r^*$. By an immediate induction on $|C|+1 \geq (|C'|+1)^{\frac{5}{4}}$, we observe that $|C_{t-1-j}|+1 \geq (|C_{t-1}|+1)^{(\frac{5}{4})^j} \geq (\frac{r^*}{2})^{(\frac{5}{4})^j}$, for $j = 1, 2, \ldots, t-2$.

Finding the largest j such that $(\frac{r^*}{2})^{(\frac{5}{4})^j} < r^*$ gives an upper bound on the maximum number of cores having size $< r^*$ in $C_1, C_2, \ldots, C_{t-1}, B_t$. Observe that such a j is $O(1)$, thus, proving the condition 2. □

We define a *Primitive General Balanced Tree* (*PGB-tree*) as a binary search tree that satisfies the fullness invariant. We are interested to apply the core partitioning scheme on PGB-tree in order to obtain an efficient data structure for the external-memory/cache-oblivious models. We start by setting parameters $h^* = h_{\min} - 1$ and $r^* = \log n$ and we call such a data structure a *Primitive Cache-Oblivious General Balanced Tree* (*PCOG-tree*).

The following lemma shows a very interesting fact about PCOG-trees that if the probability of searching a key is uniformly distributed for all the keys inside the tree, in average, during a search, just 2 cores are visited (average path length is 2 cores).

Lemma 17 *In a PCOG-tree T, the average number of the cores traversed to search for a node is less than or equal to 2 if the probability of searching for each node of T is uniformly distributed.*

Proof: Let $n = |T|$, suppose that we perform n search operations to visit all nodes of T, let $|E(T)|$ denote the total number of the cores traversed during these search operations divided by n (*i.e.*, the average number of the cores traversed).

We use induction on n to prove that $|E(T)| \leq 2$, for small values of n, it is immediate. Assume that for any tree enjoying fullness invariant and size less than n, the lemma holds. Let C denote the topmost core (topmost $h_{min}(r) - 1$ levels) of T with root r, since $|C| = 2^{h_{min}(r)-1} - 1$, by Corollary 3, we have $|C| \geq \frac{n}{2}$.

Now let $T_1, T_2, \ldots, T_{2^{h_{min}(r)}}$ denote the $2^{h_{min}(r)}$ subtrees below C and $n_i = |T_i|$ for $1 \leq i \leq 2^{h_{min}(r)}$ (also $\sum_{i=1}^{h_{min}(r)} n_i = n - |C|$). By induction hypothesis, we have $E(T_i) \leq 2$ for $1 \leq i \leq 2^{h_{min}(r)}$. On the other hand, for all $|C|$ nodes in C, we traverse just one core to visit them, and for the all the other nodes in $|T_i|$ $(1 \leq i \leq 2^{h_{min}(r)})$ in average, we traverse less than or equal to $2 + 1$ cores (by induction hypothesis and also the fact that they are all below core C). Therefore,

$$\begin{aligned} E(T) &= \frac{|C| \times 1 + (\sum_{i=1}^{h_{min}(r)} n_i) \times (3)}{n} \\ &\leq \frac{|C| \times 1 + (n - |C|) \times (3)}{n} \\ &\leq 3 - \frac{2|C|}{n}\,. \end{aligned}$$

Finally, since $|C| \geq \frac{n}{2}$, $-\frac{2|C|}{n} \leq -1$. Thereby, $E(T) \leq 2$ and the lemma holds. □

Although in the same manner to Chapter 3, PCOG-trees can be improved to an efficient data structure (with linear space and logarithmic search time) in the cache-oblivious memory model, in the following, we suggest a slightly modified version of PCOG-tree so that we can achieve the same efficient bounds with a much simpler approach.

Here, we make a small modification to the definition of terminal-cores (the subtree of size $< r^*$ at the bottom of the tree), we present a new parameter called $\overline{r^*} = \lceil \log^{\alpha} n \rceil$, for $\alpha \geq 1$, and we force terminal-cores to have size $\Theta(\overline{r^*})$ (*i.e.*, terminal-core sizes must be between $\frac{1}{2}\overline{r^*}$ and $4\overline{r^*}$) instead of $< r^*$. This changes the algorithm slightly, but it has a great impact on the simplicity and on the efficiency, for example, it merges the two concepts of r^* and buckets presented in Chapter 3 in the definition of $\overline{r^*}$. Using this small modification on the core partitioning scheme, we obtain the following data structure. We emphasize that n denotes the size of the **entire** given PGB-tree while $s(v)$ or $|T_v|$ denote the size of a subtree rooted at node v.

A *Cache-Oblivious General Balanced Tree* (*COG-tree*) is a PGB-trees (a tree with fullness invariant) with parameters $h^* = h_{\min} - \lceil \log \log^{\alpha} n \rceil = h_{\min} - \lceil \alpha \log \log n \rceil$ and $\overline{r^*} = \lceil \log^{\alpha} n \rceil$ such that every terminal-core is forced to have size of $\Theta(\overline{r^*})$ (*i.e.*, in between $\frac{1}{2}\overline{r^*}$ and $4\overline{r^*}$)[2].

[2]We chose a smaller h^* than in PCOG-trees so we have left some nodes for the terminal-cores to be of size $\Theta(\overline{r^*})$.

For the sake of simplicity, we ignore roundings in the definition of h^* and $\overline{r^*}$ when possible. The mentioned modification on the terminal-cores in a COG-tree, does not interfere with other concepts or definitions in the core partitioning scheme, *e.g.* the definition of successful core partition will be the same as before using $\overline{r^*}$ instead of r^*.

Theorem 9 *A COG-tree satisfies the following properties (successful core partition).*

1. *Any root-to-leaf path in T traverses cores of doubly exponentially decreasing size.*

2. *Only a constant number of the above cores are of small size less than or equal to $\overline{r^*}$.*

Proof: To show that condition 1 holds, in a similar manner to the first part of the proof of Theorem 8, in a root-to-leaf search path, let C and C' be two consecutive cores with heights h, h', respectively. As shown in Figure 4.3, let $u_0, u_1, u_2, \ldots, u_\ell$ show the path from the root of C' to the root of C (u_0 is the root of C' and u_ℓ is the root of C). Since $h \geq h'$, $\ell \geq h'$. On the other hand, from Lemma 15, $h_{min}(u_{i+4}) \geq h_{min}(u_i) + 1$, for $0 \leq i \leq \ell - 4$. Therefore, $h_{min}(u_\ell) \geq h_{min}(u_0) + h'/4$, also by the definition of h^*, $h = h_{min}(u_\ell) - \lceil \alpha \log\log n \rceil$ and $h' = h_{min}(u_0) - \lceil \alpha \log\log n \rceil$ (Recall that n is the size of the entire tree). Putting all together,

$$\begin{aligned} h &= h_{min}(u_\ell) - \lceil \alpha \log\log n \rceil \\ &\geq h_{min}(u_0) + h'/4 - \lceil \alpha \log\log n \rceil \\ &\geq h_{min}(u_0) - \lceil \alpha \log\log n \rceil + h'/4 \geq \frac{5}{4} h' \,. \end{aligned}$$

Therefore, $|C'| + 1 \leq (|C| + 1)^{\frac{4}{5}}$, hence, cores are doubly exponentially decreasing in a root-to-leaf path. The condition 2 also holds similarly to the second part of the proof given for Theorem 8. □

Corollary 4 *In a COG-tree with root r, the number of the the cores traversed in any root-to-leaf path is upper bounded by* $3.1063 \log\log n$.

Proof: Clearly the height of the topmost core is upper bounded by $\log n$, and the heights of the cores are decreasing exponentially with factor $\frac{4}{5}$, therefore, the number of the the cores traversed in any root-to-leaf path is upper bounded by $\log_{\frac{5}{4}} \log n = \frac{1}{\log 5/4} \log\log n < 3.1063 \log\log n$ □

Finally, it should be mentioned that in general, the behavior of the PCOG-trees and the COG-trees are similar, in the rest of the chapter, we focus only on COG-trees.

4.3 Memory Management

Here we explain the basics of the memory management of a COG-tree. The implementation of a COG-tree T exploits its core partition and it is an easy programming task. Same as the memory management of the core partitioning scheme in Section 3.2.2, given a core C of COG-tree T, we observe that it contains $|C| = 2^{h^*} - 1$ keys for some h^*, and $|C| + 1$ external pointers to its "children" cores. We can thus allocate an array of size $2|C| = 2^{h^*+1}$ entries, and fill it with the entries from C using the implicit vEB layout, so no internal pointers among C's nodes are stored. It takes $1 + \log_B |C|$ block transfers to implicitly traverse core C during a search path [29] and it takes $O(|C|/B)$ block transfers to visit (*e.g.* inorder traversal) all the nodes in core C.

Each terminal-cores of size $\Theta(\overline{r^*} = \log^\alpha n)$ is implemented as $\lceil \alpha \rceil$ layers of records of size $\Theta(\log n)$. These records are very small comparing to n, so we have a variety of possibilities to implement them, we can implement them as B-trees same as the buckets implemented in [21] or we can even implement them with sorted arrays in a contiguous portion of memory, thus each record can be sequentially scanned in $\Theta((\log n)/B) = \Theta(\log_B n)$ I/Os, hence, $\lceil \alpha \rceil \times \Theta(\log_B n) = O(\log_B n)$ I/Os to search a terminal-core, therefore, it will not change the search complexity as later we will see the search complexity is also $O(\log_B n)$.

Fact 8 *Consider a node v in a core C, and let m be the number of all the nodes descending from v (including itself) that are inside C. Then, the inorder traversal of these nodes in C requires the time needed to search v plus $O(m/B + 1)$ block transfers.*

Proof: To visit all the nodes descending from v that are inside C, we first need to search v, then since C is implemented by vEB layout, the number of block transfers to read v's descendants inside C is $O(m/B + 1)$I/Os by [29]. □

4.4 Maintaining a COG-Tree

In this section, we explain how to maintain a COG-tree as a dynamic data structure. To build a COG-tree, we can simply start from an empty tree or we can initially apply COG-tree on a given (possibly unbalanced) binary search tree[3], and then we perform a sequence of insertions and deletions maintaining the COG-tree properties, namely, it must *always* maintain the fullness invariant and the core partitioning scheme.

[3]In the case of applying COG-tree on a given (possibly unbalanced) binary search tree, we initially reconstruct the entire tree to a perfectly balanced binary search tree (last level may not be full) and we build a COG-tree on top of it.

4.4.1 Deletions

Similar to Chapter 3, for the deletion, we simply mark the searched key as logically deleted, and remove that mark if the key is inserted again. We periodically rebuild the entire tree when the number of these marked keys is a constant fraction of the total number of keys. This amortized complexity can be analyzed in a traditional way, and thus it is not discussed here.

4.4.2 Insertions

We therefore focus on the insertions. When a new key is inserted in COG-tree T, a new leaf f is created. After that, we have to maintain the fullness invariant described in Section 4.2, as well as the core partitioning scheme.

Recall that for any node u, $h_{min}(u) = \lceil \log(s(u) + 2) \rceil$. When f is inserted, for any ancestor z of f whose value $h_{min}(z)$ increases by one (we call this event *minimum-height-increase* of z), the fullness invariant can be violated. Note that because of inserting f, $s(z)$ increases by one, therefore, $h_{min}(u) = \lceil \log(s(u) + 2) \rceil$ either does not change or it increases by one (minimum-height-increase of z). To preserve the fullness invariant we then proceed as follows. We take the topmost ancestor u of f which has a minimum-height-increase (if there is any), and apply the operation called balance(u) which

(a) replaces T_u by a perfectly balanced tree T'_u storing the same set of keys, and

(b) updates the core partition.

Task (b) is performed as follows. Let C be the core containing u: we replace the entries for $C \cap T_u$ in the array storing C with the topmost $|C \cap T_u|$ entries from T'_u; observing that the number of these entries is a power of 2 minus 1, they correspond to the topmost full levels, let us say the first t levels of T'_u. The remaining entries in T'_u, which are on levels greater than t, are stored in cores using a simple greedy top-down approach.

We emphasis that a minimum-height-increase of an ancestor z of the new inserted key does not necessarily violate the fullness invariant, however, to maintain the core partitioning scheme, we perform balance(u). As stated in the next lemma, the rebalancing operation balance(u) preserves the fullness invariant in the given COG-tree. The amortized cost of balance(u) will be discussed in Section 4.6.

Lemma 18 *Operation* balance(u) *preserves the fullness invariant.*

Proof: For a given node v, let $e(v)$ denote the topmost null pointer (empty node) in T_v and let ℓ_v be the length of the path between v and $e(v)$. When operation balance(u) is performed, T_u simply becomes balanced by filling some null pointers (empty nodes) closer to node u with other nodes further from u. This means that in general, balance(u) does not decrease ℓ_z for any ancestor z of node u.

Now by contradiction, suppose that after performing balance(u), there is an ancestor z of u that violates the fullness invariant. Before performing balance(u), T_z was full up to $h_{min}(z) - 1$ level, so ℓ_z was greater than or equal to $h_{min}(z) - 1$, also z does not have a minimum-height-increase, because by definition u is the topmost node with minimum-height-increase. On the other hand, the fact that after balance(u), z violates the fullness invariant, implies that $\ell_z \leq h_{min}(z) - 2$. This means that balance(u) caused a decrease in ℓ_z, which is in contradiction with the previous result. □

There is also another simple event we need to consider for the maintenance of COG-trees. We need to reconstruct the entire COG-tree when $\lceil \alpha \log\log n \rceil\}$ or $\lceil \log^{\alpha} n \rceil$ increases by one, because these values are used in the definition of h^* and $\overline{r^*}$, hence, a change on these values results to reconstruct the entire tree. However, this event occurs rarely and every time it occurs the size of the entire tree increases by a constant factor, hence, amortized cost is linear.

4.5 Applications

In this section, we study COG-trees in the external-memory and the cache-oblivious memory models.

Lemma 19 *The number of the cores and the terminal-cores in a COG-tree of size n is $O(\frac{n}{\log^{\alpha} n})$.*

Proof: We observe that there exist at most $O(\frac{n}{\log^{\alpha} n})$ terminal-cores since the sum of their sizes cannot exceed n, also the number of the cores is upper bounded by the number of the terminal-cores (they form a t-regular tree with cores as internal nodes and terminal-cores as leaves), therefore, the number of the cores is also $O(\frac{n}{\log^{\alpha} n})$. □

Corollary 5 *Having $\alpha \geq 1$, the total space for the external pointers (pointers to the cores and the terminal-cores) is $O(\frac{n}{\log^{\alpha} n}) \times O(\log(\frac{n}{\log^{\alpha} n})) = O(n)$ bits. This is an efficient bound for the space needed for the pointers in a COG-tree.*

4.5.1 External-Memory Search Trees

In a similar manner to Section 3.3.2, by setting $\overline{r^*} = max\{\log^\alpha n, B\}$, we obtain a *B-tree-like* data structure for external memory [19]. More precisely, the complete balanced binary tree represented by each core C_i can be stored in contiguous portions of the memory of size of multiples of B, so that it takes $O(1 + h_i^*/\log B)$ I/Os to traverse C_i (e.g. see [106]).

Theorem 10 *In the external-memory model with block size B, a COG-tree of size n can be stored using $O(n/\overline{r^*})$ cores and occupying $O(n/B)$ blocks in total. Any search path from the root to a node requires $O(\log_B n)$ I/Os and $O(\log n)$ comparisons.*

Proof: The proof is similar to the proof of Theorem 1. □

4.5.2 Cache-Oblivious Search Trees

The benefit of COG-trees is that not only we can easily apply vEB layout inside each core, but also in the cache-oblivious memory model, to obtain the efficient search time of $O(\log_B n)$ using linear space, we do not need the buckets as we used to in Chapter 3. Note that now terminal-cores have a guaranteed size of $\Theta(\overline{r^*})$, therefore behaving similarly to the buckets of Chapter 3.

For a COG-tree of size n, for a subtree T_v, for the sake of computations, we ignore roundings in the formulas of r^* and h^*; we consider $\overline{r^*} = \log^\alpha n$ and $h^* = h_{min}(v) - \alpha \log\log n$, where $h_{min}(v) = \lceil \log(s(v) + 2) \rceil$.

For a COG-tree, to obtain efficient cache-oblivious search time, same as Chapter 3, we store the complete binary tree inside each core C using the *vEB layout* [81, 29], we also keep any core C and all its descendant cores in a contiguous portion of memory as we explain next. We store terminal-cores separately in another contiguous portion of the memory using $\lceil \alpha \rceil$ layers of records of size $\Theta(\log n)$ for each terminal-core. Therefore, the remaining cores form a tree of size $\overline{n} = O(\frac{n}{\log^\alpha n})$. Let $\overline{T}$ denote the tree of these remaining core (ignoring terminal-cores) and let $s'(v)$ denote the number of the nodes in $\overline{T}_v$ (the subtree of node v in $\overline{T}$).

By Corollary 4, the number of the cores traversed in any root-to-leaf path is upper bounded by $3.1063 \log\log n$. We define the level and the layer of a core similarly as before. For a core C rooted at v, the *level* of C (denoted by $lev(C)$) is the number of cores above and the *layer* of C (denoted by $lay(C)$) is $\lceil 3.1063 \log\log n \rceil - lev(C)$ and we use the same recursive scheme as Section 3.3.2 to assign $\Theta(4^{lay(C)} s'(v))$ space to C and all its descendant cores. Observe that in a similar manner to Theorems 4 and 5, the total amortized cost of maintaining such a structure in the event of SOF

or increase in $\lceil 3.1063 \log\log n \rceil$ is $O(\log\log n)$.

Theorem 11 *For $\alpha \geq 2 \times 3.1063 = 6.2126$, a COG-tree of size n with parameters $\overline{r^*} = \log^{\alpha} n$ and $h^* = h_{\min} - \alpha \log\log n$, can be stored in the cache-oblivious memory model, so that $O(n/B)$ blocks are occupied and any search path from the root to a node requires $O(\log_B n)$ I/Os and $O(\log n)$ comparisons.*

Proof: The search is $O(\log_B n)$ similarly to the Theorem 3. Also as explained before, the total space for the terminal-cores is $O(n/B)$ blocks. On the other hand, the total number of blocks occupied to store the cores is upper bounded by $O(\frac{1}{B}\overline{n}4^{\lceil 3.1063 \log\log n \rceil}) = O(\frac{1}{B}\overline{n}\log^{2\times 3.1063} n) = O(\frac{n \log^{6.2126} n}{B \log^{\alpha} n})$ which for $\alpha \geq 6.2126$, it is upper bounded by $O(n/B)$. The bound of $O(\log n)$ comparisons derives from the height invariant which is satisfied by the fullness invariant. □

4.6 Amortized Analysis

In this section, we study the amortized cost of operation balance(u) in a sequence of insertions. Recall from Section 4.4.2, after insertion of a new leaf f, operation balance(u) is performed on the topmost ancestor u of f which has a minimum-height-increase (if there is any) with the following two tasks.

(a) replaces T_u by a perfectly balanced tree T'_u storing the same set of keys, and

(b) updates the core partition.

Lemma 20 *Operation* balance(u) *can be performed in $O(s(u))$ time and $O(s(u)/B + \frac{s(u)}{\log^{\alpha} n})$ block transfers in the cache-oblivious model.*

Proof: Suppose that we want to perform an inorder traversal of the subtree T_u, here, we discuss its cache complexity. Let C be the core containing u, and observe that the inorder traversal of $C \cap T_u$ requires a linear number of blocks, $O(|C \cap T_u|/B + 1)$, by Fact 8. For the rest of the cores in T_u, we use a bottom-up induction on the cores traversed by the inorder traversal. Let C' be one of the cores below C that are traversed in T_u, and let $d(C')$ be the number of "children" cores of C'. By induction hypothesis, it takes $O(|C'|/B + 1)$ block transfers to read and traverse all the nodes in C'. We should also add $d(C')$ block transfers that are needed to access its children. Hence, the overall cache complexity of the inorder traversal is

$$O(|C \cap T_u|/B + \sum_{C' \in T_u} (|C'|/B + d(C'))) = O(s(u)/B + \sum_{C' \in T_u} d(C')),$$

where $\sum_{C' \in T_u} d(C') = O(\frac{s(u)}{\log^\alpha n})$ as there are so many cores and terminal-cores in T_u by Lemma 19. As a result, we can produce the sorted sequence of keys in T_u with $O(s(u)/B + \frac{s(u)}{\log^\alpha n})$ block transfers. After that, it is a standard computation to build the perfectly balanced tree T'_u with $O(s(u)/B)$ block transfers, thus completing task (a). As for task (b), we can observe that the cache complexity follows the same route as that for the inorder traversal, thus giving a total cost of $O(s(u)/B + \frac{s(u)}{\log^\alpha n})$ block transfers. □

Fact 9 *After inserting a new leaf f, if node u is the topmost ancestor of f with minimum-height-increase (if there is any), after performing operation balance(u) on u, T_u becomes a complete balanced binary search tree with $s(u) = 2^k - 1$ nodes, for some integer k. Also for any node v in T_u, T_v is a complete balanced binary search tree of size $s(v) = 2^{k'} - 1$, for some integer k', and $h_{\min}(v) = \lceil \log(s(v) + 2) \rceil = k' + 1 = \log(s(v) + 1) + 1$.*

Lemma 21 *Consider an insertion of a new leaf f in a COG-tree T, and suppose that a minimum-height-increase happens in an ancestor of f. Let v be the topmost such an ancestor. Let $m = s(v)$ be the size of the subtree T_v rooted at v and let T'_v denote the subtree rooted at v after the last rebalancing operation on v due to a previous minimum-height-increase on v or above. Finally, let $m' = |T'_v|$ and $h'_{\min}(v) = \lceil \log(m'+2) \rceil$. We have $m - m' = \Omega(m)$. i.e., $\Omega(m)$ new keys are inserted as descendants of v since the last minimum-height-increase on v or above.*

Proof: By the definition of minimum-height-increase, $h_{\min}(v) = h'_{\min}(v)+1$ implying that $\lceil \log(m+2) \rceil = \lceil \log(m'+2) \rceil + 1$. By Fact 9, $m+1$ and $m'+1$ are powers of 2. Thus, $\log(m+1)+1 = \log(m'+1)+2$, implying $m+1 = 2(m'+1)$, and finally $m - m' = \Omega(m)$. □

Theorem 12 *The amortized cost of operation* balance(u) *in COG-trees is $O(\log n)$ time and $O(\log_B n)$ block transfers.*

Proof: Lemma 21 implies that the cost of operation balance(u) in Lemma 20 can be spread out among $O(s(u))$ fresh insertions. As a result, the amortized cost is $O(1)$ time and $O(\frac{1}{B} + \frac{1}{\log^\alpha n})$ block transfers per new entry in the subtree T_u. Since each new entry is an inserted leaf f at some time, and f is involved as a fresh entry in $O(\log n)$ ancestors, we can charge f with $O(\log n)$ time and $O(\frac{\log n}{B} + \frac{\log n}{\log^\alpha n}) = O(\log_B n)$ block transfers. Hence, the amortized cost of operation balance(u) in COG-trees is $O(\log n)$ time and $O(\log_B n)$ block transfers. □

4.7 Summary

In this chapter, we applied the core partitioning scheme introduced in Chapter 3 to arbitrary binary search trees which can be 'unbalanced'. We then introduced a new data structure called Cache-Oblivious General Balanced Tree (COG-tree). The COG-tree of n nodes has an improved cache complexity of $O(\log_B n)$ amortized block transfers and $O(\log n)$ amortized time for updates. Search operation takes $O(\log_B n)$ block transfers and $O(\log n)$ comparisons. The space occupancy is $O(n)$ extra bits besides the space needed to store the keys alone.

Chapter 5

Other Properties of AVL Trees

In this chapter, we present some new features and properties of AVL trees. In Section 5.1, we define *gaps* as special edges in AVL trees, such that the height difference between the subtrees rooted at two endpoints of a gap is equal to 2. Using this definition, we present the *Basic-Theorem* which illustrates how the size of an AVL tree (and its subtrees) can be represented by a series of powers of 2 of the heights of the gaps. Basic-Theorem characterizes the tree size of any AVL tree with a very simple formula. We also investigate that how gaps change during a sequence of insertions and deletions. We have presented this results at the conference Combinatorics 2014 [4].

In Section 5.2, we answer to the question whether deletions can take $\Omega(\log n)$ rotations not only in the worst case, but also in the amortized case as well, when insertions are intermixed with deletions. Heaupler, Sen, and Tarjan [48] conjectured that alternating insertions and deletions in an n-node AVL tree can cause each deletion to do $\Omega(\log n)$ rotations. We provide a construction which makes each deletion to do $\Omega(\log n)$ rotations. Recently, this work has been published in the Journal Information Processing Letters [5].

5.1 GAP

Recall that for a given node v in an AVL tree T, we use the following notations.

- $|T|$ and $V(T)$ denote the size of tree T and its set of nodes, respectively,
- T_v denotes the subtree rooted in v and $key(v)$ denotes its key,
- $p(v)$ and $child(v)$ denote parent and child of v, respectively, and v_r and v_l denote the right and the left child of v, respectively,

- $h(v)$ denotes the height of T_v,
- $lev(v)$ denotes the number of the nodes in the path from v to the root.

Definition 4 *The balance factor of v (also denoted by $b(v)$) is the difference in the heights of its two subtrees $(h(v_r) - h(v_l))$. The balance factor of the nodes of an AVL tree may take one of the values -1, 0, +1. A node is* balanced *(or* unbalanced*) if its balance factor is* 0 *(or* ± 1*).*

Definition 5 *For any pair of nodes v and w, which $v = p(w)$, the edge between v and w is called* **gap** *iff the height difference between v and w is equal to 2. If there is a gap g between v and w, we say v has a gap child, and v and w are called parent and child of this gap.*

For gap g by $p(g)$, $child(g)$, and $h(g)$ we denote the parent, the child, and the height of g respectively, where we define $h(g) = h(child(g))$. We also use $GAP(T)$ to denote the set of all the gaps in a tree T.

Fact 10 *For any given node v in an AVL tree T:*

- *There is at most one gap child for v between v and v_l or v and v_r.*
- *Leaves have no gap children.*

According to the standard algorithms of AVL trees described in [57], two main operations which can change AVL tree's structure are deletion and insertion. We are going to study gap-properties for an AVL tree, in general and during these these two operations.

5.1.1 General Properties of Gaps

The following theorem expresses the size of a given AVL tree with height H in terms of the powers of 2 of the heights of the gaps.

Theorem 13 Basic-Theorem:

$$|T| = n = 2^H - 1 - \sum_{g \in GAP(T)} 2^{h(g)}.$$

Proof: By induction on H. For $H = 0$ (empty tree) and $H = 1$ (one-node tree), the theorem trivially holds. For the inductive step on $H \geq 2$, we assume that the theorem holds for any AVL tree of height less than H, then we use this assumption to prove the statement for height H. Let

T_l and T_r denote the left and right subtree of the root of the given AVL tree T, respectively. We consider two cases for T.

First suppose that the height of T_l and T_r are equal to $H-1$. Then, $GAP(T) = GAP(T_l) \cup GAP(T_r)$ as the two edges between the root of T and the roots of T_l and T_r are not gaps, and the theorem easily follows using the induction hypothesis,

$$\begin{aligned} |T| &= |T_l| + |T_r| + 1 = 2^{H-1} - 1 - \sum_{g \in GAP(T_l)} 2^{h(g)} + 2^{H-1} - 1 - \sum_{g \in GAP(T_r)} 2^{h(g)} + 1 \\ &= 2^H - 1 - \sum_{g \in GAP(T)} 2^{h(g)} . \end{aligned}$$

Now suppose that two subtrees have different heights $H-1$ and $H-2$. Then the set of the gaps of T contains all gaps in T_l and T_r, plus the new gap g' given by the edge between the root of T and the root of the subtree of height $H-2$. Therefore, using the induction hypothesis and the fact that $h(g') = H-2$, we have:

$$\begin{aligned} |T| &= |T_l| + |T_r| + 1 = 2^{H-1} + 2^{H-2} - 1 - \sum_{g \in GAP(T_l)} 2^{h(g)} - \sum_{g \in GAP(T_r)} 2^{h(g)} \\ &= 2^H - 2^{H-2} - 1 - \sum_{g \in GAP(T_l)} 2^{h(g)} - \sum_{g \in GAP(T_r)} 2^{h(g)} \\ &= 2^H - 1 - 2^{h(g')} - \sum_{g \in GAP(T_l)} 2^{h(g)} - \sum_{g \in GAP(T_r)} 2^{h(g)} = 2^H - 1 - \sum_{g \in GAP(T)} 2^{h(g)} . \end{aligned}$$

□

To show how powerful this theorem is, the following corollary describes the precise relationship between the size of the entire tree (n), the heights of the nodes, the subtree sizes, and the heights of the gaps in a given AVL tree.

Corollary 6 *For a gap g let us define $lev(g)$ as the number of the nodes above g in the path from g to the root (i.e., the number of node-ancestors of g), note that the level of a gap is equal to the level of its parent node, then:*

$$\sum_{u \in V(T)} (2^{h(u)} - |T_u|) - \sum_{g \in GAP(T)} lev(g) 2^{h(g)} = \boldsymbol{n}.$$

Proof: By Theorem 13 (Basic-Theorem) we know that for any node u, $2^{h(u)} - \sum_{g \in GAP(T_u)} 2^{h(g)} -$

$|T_u| = 1$. Therefore, by summing up of this formula over all nodes we have:

$$\sum_{u \in V(T)} 1 = n = \sum_{u \in V(T)} \{2^{h(u)} - |T_u| - \sum_{g \in GAP(T_u)} 2^{h(g)}\}.$$

On the other hand, for any gap g, for any ancestor u of g, $g \in GAP(T_u)$ and vice versa. Therefore, there are exactly $lev(g)$ nodes u which $g \in GAP(T_u)$. So, we claim that:

$$\sum_{u \in V(T)} \sum_{g \in GAP(T_u)} 2^{h(g)} = \sum_{g \in GAP(T)} lev(g) 2^{h(g)},$$

Therefore,

$$n = \sum_{u \in V(T)} (2^{h(u)} - |T_u|) - \sum_{u \in V(T)} \sum_{g \in GAP(T_u)} 2^{h(g)},$$

$$n = \sum_{u \in V(T)} (2^{h(u)} - |T_u|) - \sum_{g \in GAP(T)} lev(g) 2^{h(g)}.$$

□

Corollary 7 *The powers of 2 of the heights of the nodes and the gaps are related by the following upper bounds.*

$$\sum_{u \in V(T)} (2^{h(u)}) - \sum_{g \in GAP(T)} lev(g) 2^{h(g)} \leq n + nH = \Theta(n \log n).$$

Proof: Immediately by using Corollary 6 and the fact that $\sum_{u \in V(T)} (|T_u|)$ is the same as the total internal path length which is upper bounded by $nH \leq \Theta(n \log n)$. □

5.1.2 Gaps in Insertions and Deletions

In this section, we study how gaps change during deletion and insertion operations of AVL trees. According to the standard insertion algorithm of AVL trees described in [57], after the insertion of a new node, three different situations can occur, namely:

1. absorption;
2. rotation at the critical node (single or double);
3. height increase.

For the insertion of a new node v into an AVL tree T, let v_k denote the root of T, $v_k, v_{k-1}, \ldots, v_0$ be the insertion path of $key(v)$ and i be the maximum index such that $b(v_i) = b(v_{i-1}) = \ldots =$

$b(v_0) = 0$, recall that $b(v)$ is the balance factor of node v. Hence, v_{i+1} (if any) is called the *critical node*, and $v_i, \ldots, v_0$ is called the *critical path*. The length of the critical path is $l_v = i$. In the case of height increase of T, $i = k$ and there exists no critical node.

First, we consider insertion operation, as we know there are three possibilities (absorption, rotation, and height increase), when one of these cases occurs, gaps inside the subtree of the critical node will change, some will disappear and some will be created as studied in the following theorem.

Theorem 14 *In an insertion of a new node v with critical path of length l_v and critical node u:*

- *In the case of a "height increase" (increasing the height of the entire tree), a sequence of gaps with heights from 0 to l_v will be created as shown in Figure 5.1.b.*
- *In the case of "absorption" one gap of height l_v will disappear (will be removed) and a sequence of gaps with heights from 0 to $l_v - 1$ will be created, see Figure 5.1.a.*
- *In the case of "rotation" (single or double) one gap of height $l_v - 1$ will disappear (will be removed) and a sequence of gaps with heights from 0 to $l_v - 2$ will be created, see Figure 5.2, and 5.3.*

Proof: As illustrated in Figures 5.1.a, 5.2 and 5.3, in the cases of absorption and rotation, there should be always a gap g before performing the insertion whose parent is u; obviously this gap disappears after absorption/rotation. The only case remaining is the height increase; in this case, as illustrated in Figure 5.1.b, there is no critical node and there is no gap to disappear. In all the cases, the sequences of created gaps are shown in Figures 5.1, 5.2, and 5.3. □

Definition 6 *For a given gap g,* 'consuming' *g means that this gap has disappeared from the tree, either by a rotation or an absorption, and a sequence of gaps as mentioned before, has been* 'generated'.

Theorem 15 *A gap g or a sequence of gaps can be generated either by a height increase or by consuming a gap above, as shown in Figures 5.1, 5.2, and 5.3.*

Proof: Notice that except the case of the height increase, to generate a gap or a sequence of gaps, one gap above should be consumed. □

Recall that, unlike insertion, deletion of a node can violate AVL tree condition **at every level** in the AVL tree. According to the standard deletion algorithm of AVL trees described in [57], after

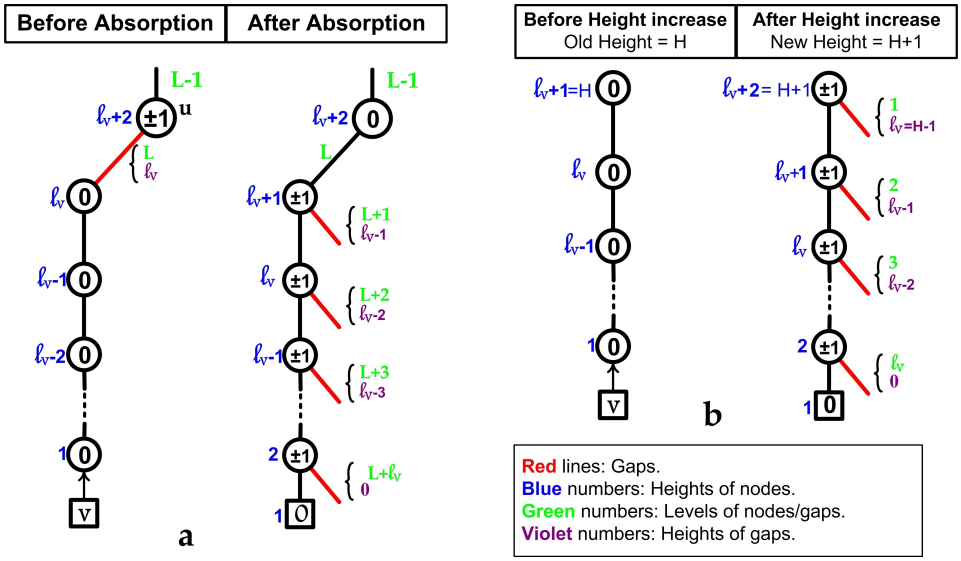


Figure 5.1: Gaps before and after absorption(a) and height increase(b).

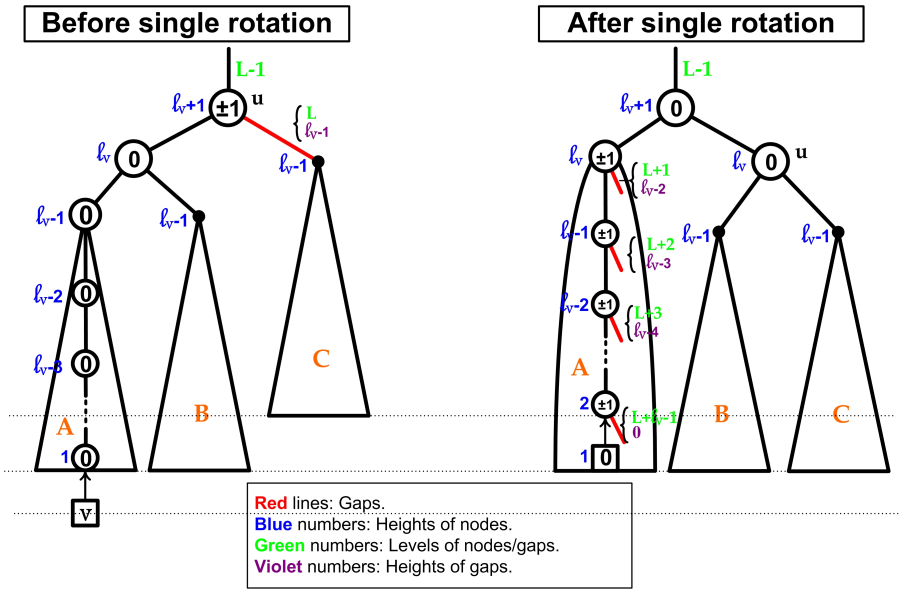


Figure 5.2: Gaps before and after single rotation.

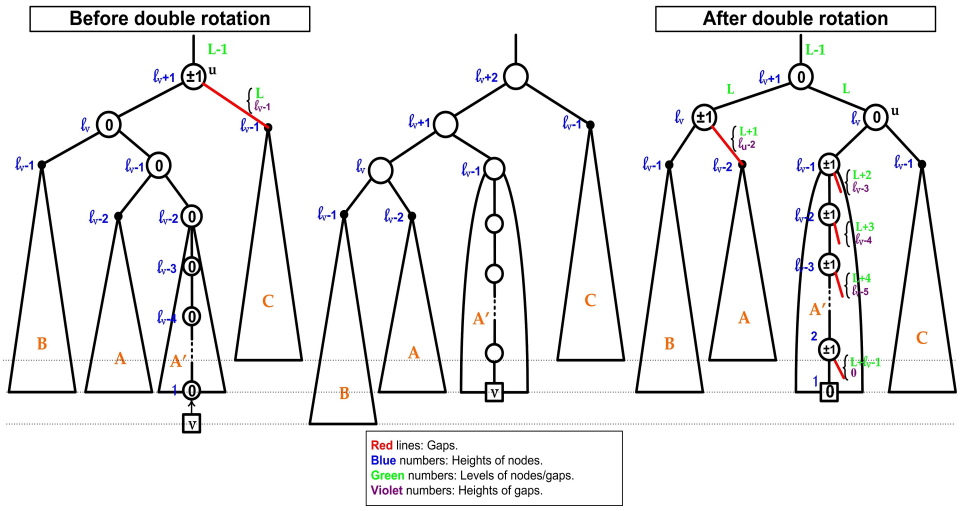


Figure 5.3: Gaps before and after double rotation.

deletion of a node v, three different situations based on the number of children of v can occur, namely:

1. v has 0 children: v will be deleted and nodes' heights in the path from v to root may change and they may need to rebalance.

2. v has 1 child: v will be deleted, its child should be connected to its parent and nodes' heights in the path from v to root may change and they may need to rebalance.

3. v has 2 children: we should find v's *successor* and replace it with v and remove the successor, therefore, nodes' heights in the path from the successor to the root may change and they may need to rebalance.

For the deletion of a node v from an AVL tree T, let $v = v_0, v_1, \ldots, v_i = u$ be the maximum path made of gaps starting from $v = v_0$ going upward (this path can be empty). As it has been shown in Figure 5.4, in both cases of "0 child" or "1 child", after the deletion, all the gaps of this path will be consumed (disappeared) and the edge $(u, p(u))$ can become a gap or not, depending on the balance factor of $p(u)$ before deletion (if $b(p(u)) = 0$ then $(u, p(u))$ will be a gap after deletion). In case of "1 child", another gap will be generated as the child of v. In the case of "2 children" since we replace v with its successor and we remove the successor, we will have similar cases of "0 child" or "1 child" but for the successor. So there is no need to study this case separately.

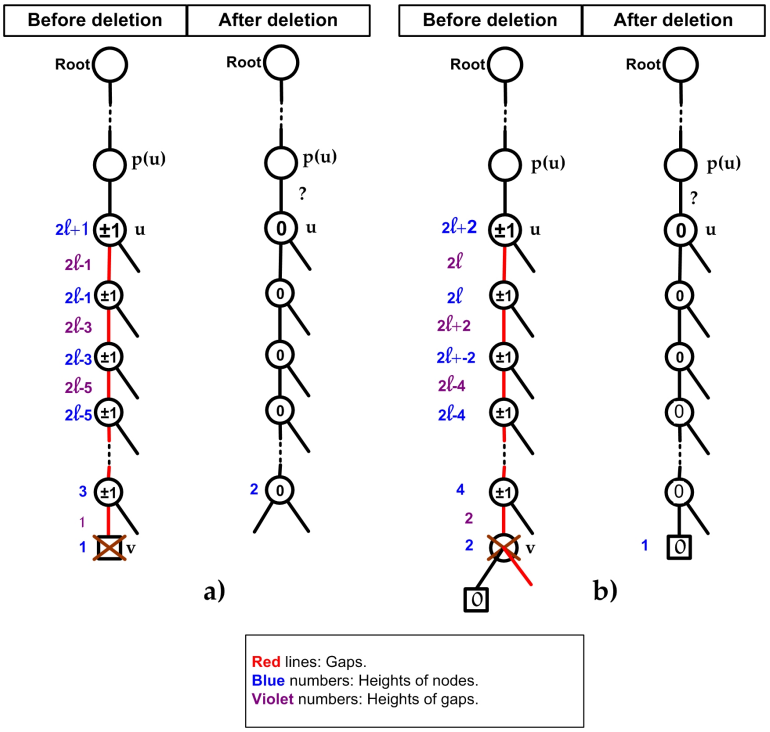


Figure 5.4: Gaps before and after deletions in case of no children of deleted node(a) or one child(b).

5.2 Amortized Rotation Cost in AVL Trees

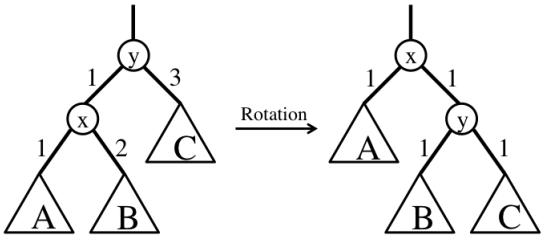
In this section, we study Heaupler, Sen, and Tarjan's conjecture in [48] that alternating insertions and deletions in an n-node AVL tree can cause each deletion to do $\Omega(\log n)$ rotations. We use partially the balance framework of Haeupler, Sen, and Tarjan [48]. A node in a binary tree is *binary*, *unary*, or a *leaf* if it has two, one, or no children, respectively. Recall that the *height* of a node in a tree is the number of the nodes on the longest simple downward path from the node to a leaf. By convention, a null pointer has height 0. The height of a tree is the height of its root. Recall that we denote the parent of a node x by $p(x)$. The *height difference* of a child x is $h(p(x)) - h(x)$. A child of height difference i is an *i-child*; a node whose children have height differences i and j with $i \leq j$ is an *i, j node*[1].

Recall that using the above definition, an *AVL tree* is a binary tree satisfying the following *height-rule*: every node is 1,1 or 1,2. Since null pointers have height 0, every leaf in an AVL tree is 1,1 and has height 1, and every unary node is 1,2 and has height 2. Also, by definition, for every 2-child node, the edge to its parent is a gap.

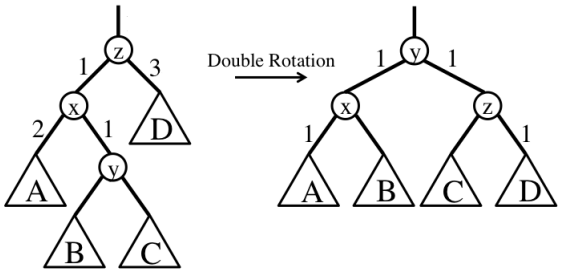
Recall that AVL trees grow by leaf insertions and shrink by deletions of leaves and unary

[1]In [48], *i-child* and *i, j node* were defined based on the *ranks* of the nodes which are equal to the heights minus one except possibly during rebalancing, here we used a similar but slightly different definition.

nodes. To add a leaf to an AVL tree, replace a missing node by the new leaf of height 1. If this violates the height-rule by having a 1,3 node, then we need to rebalance the tree by applying the appropriate single/double rotation as shown in Figure 5.5. On the other hand, to delete a leaf in an AVL tree, remove and replace it by a null pointer; to delete a unary node, replace it by its only child. Similarly, such a deletion can violate the height-rule by producing a 1,3 node. In this case, rebalance the tree by applying the appropriate case in Figure 5.6 until there is no violation. Each application of a case in Figure 5.6 either restores the height-rule or creates a new violation at the parent of the previously violating node. Whereas, each rotation case in insertion terminates rebalancing, the rotation cases in deletion can be non-terminating.



(a) Single rotation to rebalance after insertion



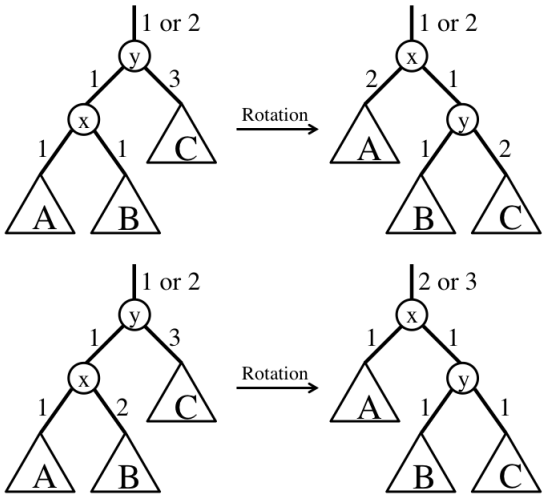
(b) Double rotation to rebalance after insertion

Figure 5.5: Rebalancing cases after insertion. Numbers next to edges are height differences.

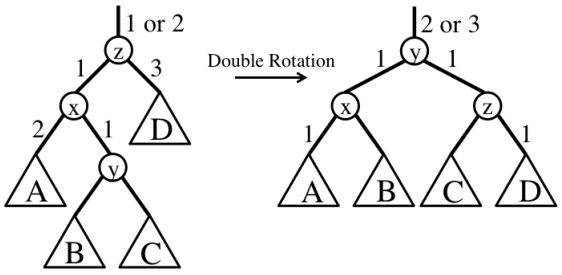
In order to obtain an initial tree in our expensive set E, we must build it from an empty tree. Thus the first step in our construction is to show that any n-node AVL tree can be built from an empty tree by doing n insertions (see Theorem 16). Although this result is easy to prove, we have not seen it in the literature.

Theorem 16 *Any n-node AVL tree can be built from an empty tree by doing T insertions, each of these insertion does only absorption or height increase (no rotation).*

Proof: Let T be a non-empty AVL tree. The *truncation* $\underline{T}$ of T is obtained by deleting all the leaves of T and decreasing the height of each remaining node by 1. We prove by induction on the height h of T that we can convert its truncation $\underline{T}$ into T by inserting the leaves deleted from T to



(a) Single rotation to rebalance after deletion



(b) Double rotation to rebalance after deletion

Figure 5.6: Rebalancing cases after deletion. Numbers next to edges are height differences.

form $\underline{T}$, in an order such that no insertion needs a rotation. The theorem then follows by induction on the height of the desired tree.

For $h = 1$ or $h = 2$, the hypothesis is trivial. Suppose $h \geq 3$ and the result holds for any AVL tree with height less than h. Let T be an AVL tree of height h. Tree T consists of a root x and left and right subtrees T_l and T_r, both of which are AVL trees. The truncation $\underline{T}$ of T consists of root x, now of height $h-1$, and left and right subtrees $\underline{T_l}$ and $\underline{T_r}$. Both T_l and T_r have height $h-1$ or $h-2$, and at least one of them has height $h-1$. Without loss of generality, suppose T_r has height $h-1$.

In the left subtree of $\underline{T}$, do the sequence of insertions that converts $\underline{T_l}$ into T_l. Then, in the right subtree of the resulting tree, do the sequence of insertions that converts $\underline{T_r}$ into T_r. T_l has height either $h-1$ or $h-2$. If T_l has height $h-1$, then the insertion into $\underline{T_l}$ that increases the root height by 1, when done in $\underline{T}$, also increases the root height of $\underline{T}$ by 1, from $h-1$ to h, this results in increasing the height difference of the right child of the root from 1 to 2 but having no other effect on the right subtree of the root. Thus, after all the insertions into the left subtree, the tree consists of root x, now of height h, left subtree T_l, and right subtree $\underline{T_r}$ of height $h-2$. The subsequent insertions into the right subtree will convert it into T_r (and changing its root's height difference to 1) without affecting the rest of the tree, producing T as the final tree.

On the other hand, if T_l has height $h-2$, then the insertions into the left subtree of $\underline{T}$ will convert the left subtree into T_l with increasing the height of the root of the left subtree from $h-3$ to $h-2$ but having no effect on the root or the right subtree. The subsequent insertions will convert the right subtree into T_r. Among these insertions, the one that increases the height of the root of the right subtree from $h-2$ to $h-1$ will also increase the height of x from $h-1$ to h, thereby, converting the root of the left subtree from a 1-child to a 2-child but having no other effect on the left subtree. Thus the final tree is T. □

5.2.1 Expensive AVL Trees

Our expensive trees have odd height. We define the set E of expensive trees recursively. Initially, set E contains the empty tree of height 0 and the one-node tree of height 1. Now, if A, B, and C are AVL trees which B has height $h-1$ and A and C have height h, and $A \in E$ and $B \in E$, then the two trees of height $h+2$ shown in Figure 5.7 are in E. The tree of type L in Figure 5.7 contains a root x of height $h+2$ and a left child y of the root of height $h+1$, and has A, B, and C as the left and right subtrees of y and the right subtree of x, respectively. The tree of type R in

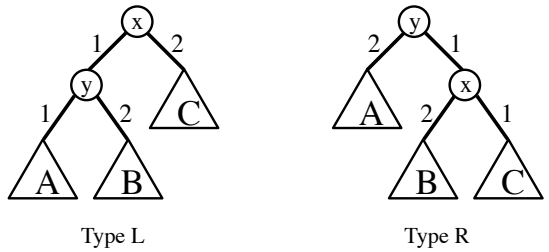


Figure 5.7: Recursive definition of E. Numbers on edges are height differences. The two trees shown are in E if A and C are in E with height h and B is an AVL tree with height $h-1$.

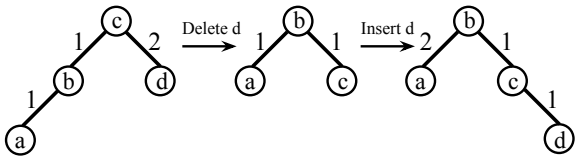


Figure 5.8: Deletion and insertion of the shallow leaf in a type-L tree of height 3.

Figure 5.7 is similar except that x is the right child of y and A, B, and C are the left subtree of y and the left and right subtrees of x, respectively.

If T is a tree in E, its shallow leaf is the leaf z such that all nodes on the path from z to the root, except the root itself, are 2-children (the edges on the path are gaps). A straightforward proof by induction shows that the shallow leaf exists and is unique.

Theorem 17 *If T is a tree in E of odd height h, deletion of its shallow leaf takes $\frac{h-1}{2}$ single rotations and produces a tree of height $h-1$. Reinsertion of the deleted leaf produces a tree of height h that is in E.*

Proof: We prove the theorem by induction on h. In the one-node tree of height 1, the shallow leaf is the only node. Its deletion takes no rotations and produces the empty tree; its reinsertion reproduces the original tree. For $h = 3$, there is exactly one tree in E of type L and one of type R. As shown in Figure 5.8, rebalancing after deletion of the shallow leaf in the type-L tree takes one rotation and produces a tree of height 2, and reinsertion produces the type-R tree. Symmetrically, deletion of the shallow leaf in the type-R tree takes one rotation and produces a tree of height 2, and reinsertion produces the type-L tree.

Suppose that the theorem is true for odd height h. Let T be a tree of height $h+2$ and type L in E (the argument is symmetric for a tree of type R). Let x be the root, y the left child of x, and A, B, and C the left and right subtrees of y and the right subtree of x, respectively (see the first tree in Figure 5.9). The shallow leaf of C is also the shallow leaf of T. By the induction hypothesis, its

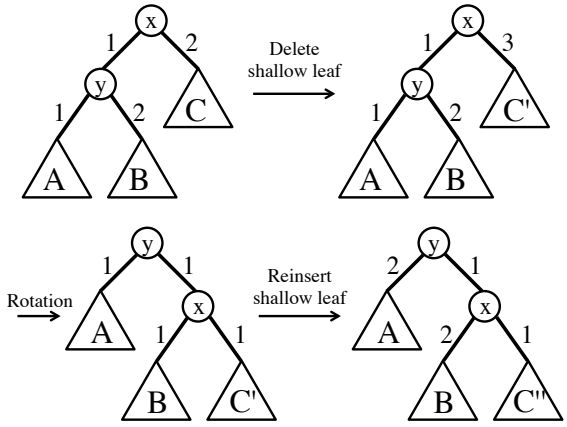


Figure 5.9: Deletion and insertion of the shallow leaf in a type-L tree of height $h+2$.

deletion in C does $(h-1)/2$ rotations and converts C into a tree C' of height $h-1$. In T, deletion of the shallow leaf converts the right subtree of x into C', making the root of C' a 3-child (see the second tree in Figure 5.9). This causes one more single rotation, for a total of $\frac{h-1}{2}+1 = \frac{(h+2)-1}{2}$ rotations, and produces the tree T' (shown as the third tree in Figure 5.9), of height $h+1$, with 1,1 root y whose right child x is also 1,1. By the induction hypothesis, reinsertion of the deleted leaf into C' converts C' into a tree C'' in E of height h. In T', the same reinsertion converts the right subtree of T' into C'' and produces the tree T'' in Figure 5.9, which is a tree in E of type R. □

Corollary 8 *The proof of Theorem 17 implies that if one starts with a tree T in E of odd height h and does $2^{(h-1)/2}$ deletion-reinsertion pairs, the final tree will be T.*

Corollary 9 *For infinitely many n, there is a sequence of $3n$ intermixed insertions and deletions on an initially empty AVL tree that takes $\Theta(n \log n)$ rotations.*

Proof: Let T be any tree in E. If T has n nodes, its height is $\Theta(\log n)$ since it is an AVL tree [1]. Apply Theorem 16 to build T in n insertions. Then repeat the following pair of operations n times: delete the shallow leaf; reinsert the deleted leaf. By Theorem 17, the total number of rotations will be $\Theta(n \log n)$. □

5.3 Summary

In this chapter, we presented a new way for studying AVL trees: the *gaps*, then we proved a new set of theorems and lemmas for finding the subtree size of any node of an AVL tree with

respect to the heights and the structure of the gaps, also we proved Heaupler, Sen, and Tarjan's conjecture that alternating insertions and deletions in an n-node AVL tree can cause each deletion to do $\Omega(\log n)$ rotations. To do this, we provided a construction which causes each deletion to do $\Omega(\log n)$ rotations: we show that, for infinitely many n, there is a set E of *expensive* n-node AVL trees with the property that, given any tree in E, deleting a certain leaf and then reinserting it produces a tree in E, with the deletions having done $\Theta(\log n)$ rotations. In general, the tree produced by an expensive deletion-insertion pair is not the original tree. Indeed, for the trees in E with odd height h, $2^{(h-1)/2}$ deletion-insertion pairs are required to reproduce the original tree.

Chapter 6

Generation of Trees with Bounded Degree

Studying combinatorial properties of restricted graphs or graphs with configurations has many applications in various fields of computer science. In this chapter, as a byproduct of our research, we study unlabeled ordered trees whose nodes have maximum degree Δ and we present a new encoding with the respective generation, ranking, and unranking algorithms. This work has been presented in DCM 2015 [6]. We also have presented another result of such ranking and unranking algorithms in [7].

A *labeled tree* is a tree in where to each node is given a unique label. A *rooted tree* is a tree in which one of the nodes is distinguished from the others as the *root*. An *ordered tree* or *plane tree* is a rooted tree for which an ordering is specified for the children of each node. We denote *unlabeled ordered trees whose nodes have maximum degree* Δ by T^{Δ} *trees*, we also use T_n^{Δ} to denote the class of T^{Δ} trees with n nodes. Formally, a T^{Δ} tree T is defined as a finite set of nodes such that T has a root r, and if T has more than one node, r is connected to $j \leq \Delta$ subtrees $T_1, T_2, \ldots, T_j$, each one of them is also recursively a T^{Δ} tree, by T_n^{Δ} we represent the class of T^{Δ} trees with n nodes. An example of a T^{Δ} tree is shown in Figure 2.17.

As mentioned in Chapter 2, although many papers have been published earlier in the literature for generating different classes of trees, few of them were related to the trees with bounded degree, and to our knowledge, no ranking or unranking are known for ordered trees with bounded degree, while a generation algorithm for this class already exists [15] where all such trees with n nodes are generated from the complete set of trees with $n-1$ nodes. Unfortunately, redundant generations

are also possible, hence, the generation algorithm is not efficient.

In Section 6.1, we present a new encoding for T_n^Δ trees. The size of our encoding is n while the alphabet size is 4. We also present a new generation algorithm with constant average time and $O(n)$ worst case time in Section 6.2. In this algorithm, the trees are generated in A-order. Ranking and unranking algorithms are also designed in Section 6.3 with $O(n)$ and $O(n \log n)$ time complexities, respectively. The presented ranking and unranking algorithms need a precomputation of size and time $O(n^2)$ (assuming Δ is constant).

6.1 The Encoding Schema

As mentioned earlier, in most of the tree generation algorithms, a tree is represented by an integer or an alphabet sequence called *codeword*, hence all possible sequences of this representation are generated. In general, on any class of trees, we can define a variety of orderings for the set of the trees. Classical orderings on trees are *A-order* and *B-order* which are defined as follows [104, 103, 115].

Definition 7 *Let T and T' be two ordered trees in T^Δ and $k = max\{deg(T), deg(T')\}$, if $T = T'$, they have the same order, otherwise, we say that T is less than T' in A-order ($T \prec_A T'$), iff*

- *$|T| < |T'|$, or*
- *$|T| = |T'|$ and for some $1 \leq i \leq k$, $T_j = T'_j$ for all $j = 1, 2, \ldots, i-1$ and $T_i \prec_A T'_i$;*

where $|T|$ is the number of nodes in T and $deg(T)$ is the degree of the root of T.

Definition 8 *Let T and T' be two ordered trees in T^Δ and $k = max\{deg(T), deg(T')\}$, if $T = T'$, they have the same order, otherwise, we say that T is less than T' in B-order ($T \prec_B T'$), iff*

- *$deg(T) < deg(T')$, or*
- *$deg(T) = deg(T')$ and for some $1 \leq i \leq k$, $T_j = T'_j$ for all $j = 1, 2, \ldots, i-1$ and $T_i \prec_B T'_i$.*

Our generation algorithm, given in the Section 6.2, produces the sequences corresponding to T_n^Δ trees in A-order. For a given tree $T \in T_n^\Delta$, the *generation algorithm* generates all the successor trees of T in T_n^Δ, the position of tree T in T_n^Δ is called *rank*, the *rank function* determines the rank of T; the inverse operation of ranking is *unranking*. These functions can be easily employed in any random generation of T_n^Δ trees.

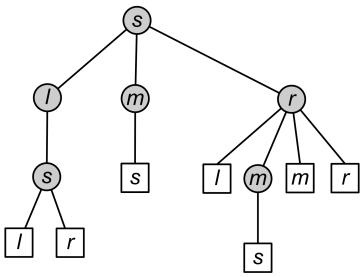


Figure 6.1: An example of a tree $T \in T_n^{\Delta}$ (for $\Delta \geq 4$). Its codeword is "$s\ell s\ell rmsr\ell msmr$".

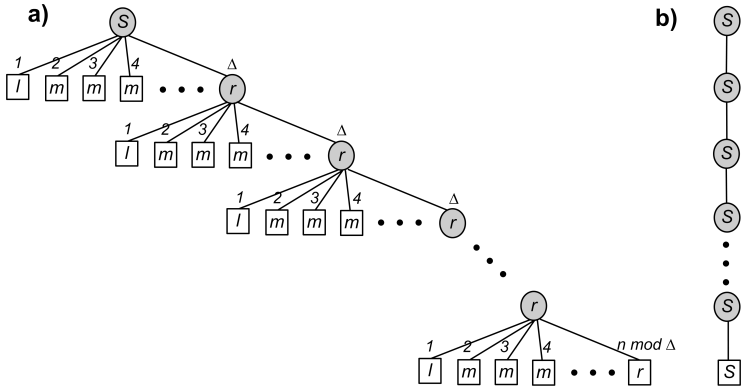


Figure 6.2: a) The first T_n^{Δ} tree in A-order. b) The last T_n^{Δ} tree in A-order.

The main point in generating trees is to choose a suitable encoding to represent them, and generate their corresponding codewords. Regarding the properties of T_n^{Δ}, we present our new encoding. For any T_n^{Δ} tree T, the encoding over 4 letters $\{s, \ell, m, r\}$ is defined as follows. The root of T is labeled by s, and for any internal node, if it has only one child, that child is labeled by s, otherwise the leftmost child is labeled by ℓ, and the rightmost child is labeled by r, and the children between the leftmost and the rightmost children (if exist) are all labeled by m. Nodes are labeled in the same way for any internal node in each level recursively, and by a pre-order traversal of T, the codeword will be obtained. This labeling is illustrated in Figure 6.1. Note that the 4-letters alphabet codeword corresponding to the first and last T_n^{Δ} trees in A-order are respectively "$s\ell m^{\Delta-2}r\ell m^{\Delta-2}r \ldots \ell m^{(n \bmod \Delta)-2}r$" and "$s^n$" which are shown in Figure 6.2-a and Figure 6.2-b. In Theorem 18 we will prove the validity of this encoding for T_n^{Δ} trees, *i.e.*, every T_n^{Δ} tree has such a codeword and two different T_n^{Δ} trees can never have the same codewords.

Definition 9 *Suppose that $\{s, \ell, m, r\}^*$ is the set of all sequences with alphabet of s, m, ℓ, r and let A be a proper subset of $\{s, \ell, m, r\}^*$, then we call the set A a* CodeSet$^{\Delta}$ *iff A satisfies the following properties:*

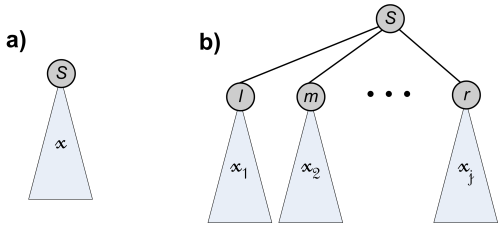


Figure 6.3: T^{Δ} trees encoded by $C = sx$ and $C = s\ell x_1 m x_2 \dots m x_{j-1} r x_j$.

1. $\epsilon \in A$ (ϵ *is a string of length* 0),

2. $\forall x \in A : sx \in A$,

3. $\forall\ x_1, x_2, \dots, x_i \in A$, *and* $2 \leq i \leq \Delta$: $\ell x_1 m x_2 m x_3 \dots m x_{i-1} r x_i \in A$.

Now we show that a valid codeword is obtained by the concatenation of the character s and each element of CodeSet$^{\Delta}$.

Theorem 18 *Let A be the "CodeSet$^{\Delta}$" and δ be a string such that $\delta \in A$ and C be a codeword obtained by the concatenation of the character s and δ (we show it by $s\delta$). There is a one-to-one correspondence between C and a unique T^{Δ} tree.*

Proof: It can be proved by induction on the length of C. Initially for a codeword of length equal to 1, the proof is trivial. Assume that any codeword obtained in the above manner with length less than n encodes a unique T^{Δ} tree. For a given codeword with length n, because of that concatenation of s and δ, we have:

1. $C = sx$, such that $x \in A$, or

2. $C = s\ell x_1 m x_2 \dots m x_{j-1} r x_j$, such that $x_i \in A$, $\forall 1 \leq i \leq j \leq \Delta$.

For the first case by induction hypothesis, x is a valid codeword of a T^{Δ} tree T; therefore, sx is another codeword corresponding to a T^{Δ} tree by adding a new root to the top of T. This tree is shown in Figure 6.3-a. For the second case, by induction hypothesis and that concatenation of s and δ, each sx_i for $1 \leq i \leq j$ is a valid codeword for a T^{Δ} tree, therefore with replacement of 's with ℓ in sx_1' and 's with m in sx_i for $2 \leq i \leq j-1$' and finally 's with r in sx_j' we can produce $\ell x_1, m x_2, \dots m x_{j-1}, r x_j$ codewords. Now they all are subtrees of a T^{Δ} tree whose codeword is $C = s\ell x_1 m x_2 \dots m x_{j-1} r x_j$ (add a new root and connect it to each one of them). This tree is shown in Figure 6.3-b. □

For a T_n^Δ tree, this encoding needs only 4 alphabet letters and has length n. This encoding is simple and powerful, so it can be used for many other applications besides the generation algorithm. In the next section, we use it to generate T^Δ trees in A-order.

6.2 The Generation Algorithm

In this section, we present an algorithm that generates the successor sequence of a given codeword of a T_n^Δ tree in A-order. For generating the successor of a given codeword C corresponding to a T_n^Δ tree T, the codeword C is scanned from right to left. Scanning the codeword C from right to left, corresponds to a reverse pre-order traversal of T. First we describe how this algorithm works directly on T, then we present the algorithm for generating the successor of C. For generating the successor of a given T_n^Δ tree T we traverse the tree in reverse pre-order as follows.

1. Let v be the last node of T in the pre-order traversal.
2. If v doesn't have any brothers, then
   - repeat $\{v =$ parent of $v\}$
     until v has at least one brother or v is the root of tree T.
   - If $v =$ root, then the tree is the last tree in A-order and there is no successor.
3. If v has at least one brother (obviously it has to be a left brother), delete one node from the subtree of v and insert this node into its left brother's subtree, then rebuild both subtrees (each one as a first tree with corresponding nodes in A-order).

To see that the above procedure gives the successor, it is sufficient to notice that its main principal (in step 2) is to find the last node v (in the pre-order traversal) with a left brother, so that in step 3, it updates the tree by moving one node from subtree of v to its brother and rebuilding both subtrees as the first corresponding trees in A-order. This approach is based on the definition of A-order trees and it is easy to observe that it generates the successor tree.

The pseudo-code of this algorithm for codewords corresponding to T_n^Δ trees is presented in Figure 6.4. In this algorithm, C is a global array of characters holding the codeword (the algorithm generates the successor sequence of this codeword), n shows the size of the codeword (the number of nodes of the tree corresponded to C), $STsize$ is a variable contains the size of the subtree rooted by node corresponded to $C[i]$ and $SNum$ holds the number of consecutive visited s characters. This algorithm also calls two functions $updateChildren(i, ChNum)$ presented in Figure 6.5, and

```
Function AOrder-Next(n : integer);
var i, Current, STsize, SNum: integer; finished, RDeleted: boolean;
begin
    Current := n; STSize := 0; RDeleted := false; finished := false;
    while ( (C[Current] = 's') & (Current ≥ 1) ) do
        STSize + +; Current − −;
    if (Current = 0) then return ('no successor');
    while (not finished) do
    begin
        STSize + +;
        switch C[Current] of
            case 'r':
                i := Current − 1; SNum := 0;
                while (C[i] = 's') do
                    SNum := SNum + 1; i − −;
                if (C[i] = 'r') then begin
                    updateBrothers ( Current , STSize);
                    Current := i; STSize := SNum;
                end;
                if ( (C[i] = 'm') or (C[i] = 'ℓ') ) then begin
                    if (STSize = 1) then RDeleted := true;
                    if (STSize > 1) then begin
                        STSize − −; updateBrothers(Current + 1, STSize);
                        Current := i; STSize := SNum + 1;
                    end;
                end;
            case 'm':
                if (RDeleted = true) then C[Current] := 'r';
                updateChildren( Current + 1, STSize − 1); finished:= true;
            case 'ℓ':
                if (RDeleted = true) then C[Current] := 's';
                updateChildren( Current + 1, STSize − 1); finished := true;
    end;
end;
```

Figure 6.4: Algorithm for generating the successor codeword for T_n^{Δ} trees in A-order.

$updateBrothers(i, ChNum)$ presented in Figure 6.6. The procedure $updateChildren(i, ChNum)$ regenerates the codeword corresponding to the children of an updated node and the procedure $updateBrothers(i, ChNum)$ also regenerates the codeword corresponding to the brothers of a node with regard to the maximum degree Δ for each node. In these algorithms, C is a global array of characters holding the codeword, i is the position of the current node in the array C, $ChNum$ is the number of children/brothers of $C[i]$ to regenerate the corresponding codeword and $NChild$ is a global array which $NChild[i]$ holds the number of left brothers of node corresponding to $C[i]$ plus one.

In Theorem 19, we prove that this generation algorithm has a worst case time of $O(n)$ and a constant average time.

```
procedure updateChildren( i, ChNum: integer);
begin
    while (ChNum > 0) do begin
        if ChNum = 1 then begin
            C[i] := 's'; NChild[i] := 1; i++; ChNum--;
        end;
        if ChNum > 1 then begin
            C[i] := 'ℓ'; NChild[i] := 1; i++; ChNum--;
            while ( (NChild[i] < (Δ − 1)) & (ChNum > 1) ) do begin
                C[i] := 'm'; NChild[i] := NChild[i − 1] + 1; i++; ChNum--;
            end;
            C[i] := 'r'; NChild[i] := NChild[i − 1] + 1; i++; ChNum--;
        end
    end;
end;
```

Figure 6.5: Algorithm for updating the children.

```
Procedure updateBrothers( i, ChNum: integer);
begin
    if ChNum = 1 then begin
        C[i] := 'r'; NChild[i] := NChild[i − 1]; ChNum--;
    end;
    if ChNum > 1 then begin
        C[i] := 'm'; ChNum--; i++;
        while ( (NChild[i] < (Δ − 1) ) & (ChNum > 1) ) do begin
            C[i] := 'm'; NChild[i] := NChild[i − 1] + 1; i++; ChNum--;
        end;
        C[i] := 'r'; NChild[i] := NChild[i − 1] + 1;
        i++; ChNum--; updateChildren(i, ChNum);
    end;
end;
```

Figure 6.6: Algorithm for updating the neighbors.

Let $S^{n,\Delta}$ be the number of T_n^Δ trees and $S^{n,\Delta,d}$ be the number of T_n^Δ trees which its root has **maximum** degree $d \leq \Delta$. Note that $S^{0,\Delta} = S^{0,\Delta,d} = S^{1,\Delta} = S^{1,\Delta,d} = 1$.

Lemma 22 *There is a constant value $\gamma > 1$, such that $S^{n+1,\Delta} \leq \gamma S^{n,\Delta} - \gamma$.*

Proof: We use induction on n. Observe that for small values of n it is trivial. Let us assume $S^{n+1,\Delta} \leq \gamma S^{n,\Delta} - \gamma$ for any $n \leq m$. Let T be a T_{m+1}^Δ tree and T_1 be its first subtree. Clearly, $T \backslash T_1$ (tree obtained by removing T_1 entirely from T) is a $T_{m-|T_1|}^\Delta$ tree which its root has **maximum** degree $\Delta - 1$. Therefore,

$$S^{m,\Delta} = \sum_{i=1}^{m-1} (S^{i,\Delta} \times S^{m-i,\Delta,\Delta-1}) \,. \tag{6.1}$$

Therefore,

$$\begin{aligned}
S^{m+1,\Delta} &= \sum_{i=1}^{m}(S^{i,\Delta}S^{m+1-i,\Delta,\Delta-1}) \\
&= S^{m,\Delta,\Delta-1} + \sum_{i=2}^{m}(S^{i,\Delta}S^{m+1-i,\Delta,\Delta-1}) \\
&= S^{m,\Delta,\Delta-1} + \sum_{i=2}^{m}(\gamma S^{i-1,\Delta}S^{m+1-i,\Delta,\Delta-1}) - \gamma\sum_{i=2}^{m}S^{m+1-i,\Delta,\Delta-1} \quad \text{(induction hypothesis)} \\
&= S^{m,\Delta,\Delta-1} - \gamma\sum_{i=2}^{m}S^{m+1-i,\Delta,\Delta-1} + \gamma\sum_{j=1}^{m-1}(S^{j+1-1,\Delta}S^{m+1-(j+1),\Delta,\Delta-1}) \quad (\text{for } j=i-1) \\
&\leq -\gamma + \gamma\sum_{j=1}^{m-1}(S^{j,\Delta}S^{m-j,\Delta,\Delta-1}) \\
&\leq \gamma S^{m,\Delta} - \gamma \quad \text{(by Equation 6.1)}.
\end{aligned}$$

Hence the induction is complete. □

Theorem 19 *The algorithm Next presented in Figure 6.4 has a worst case time complexity of $O(n)$ and an average time complexity of $O(1)$.*

Proof: The worst case time complexity of this algorithm is $O(n)$ because the sequence is scanned just once. For computing the average time, it should be noted that during the scanning process, every time we visit the characters m or ℓ, the algorithm will terminate, so we define $S_i^{n,\Delta}$ as the number of codewords of T_n^Δ trees whose the last character m or ℓ has distance i from the end, recall that $S^{n,\Delta}$ denotes the total number of T_n^Δ trees. Obviously we have:

$$S^{n,\Delta} = \sum_{i=1}^{n} S_i^{n,\Delta}. \tag{6.2}$$

We define H_n as the average time of generating all codewords of T_n^Δ trees,

$$\begin{aligned}
H_n &\leq (k/S^{n,\Delta})\textstyle\sum_{i=1}^{n} iS_i^{n,\Delta}, \\
&\leq (k/S^{n,\Delta})\textstyle\sum_{j=1}^{n}\sum_{i=j}^{n} S_i^{n,\Delta}.
\end{aligned}$$

Where k is a constant value. On the other hand, consider that for $S_j^{n+1,\Delta}$ we have two cases, in the first case, the last character m or ℓ is a leaf and in the second one, it is not. Therefore, $S_j^{n+1,\Delta}$ is greater than or equal to just the first case, and in that case by removing the node corresponding to the 'last character m or ℓ of the codeword', the remaining tree will have a corresponding codeword belongs to exactly one of $S_k^{n,\Delta}$ cases, for $j \leq k \leq n$. By substituting k and i we have:

$$S_j^{n+1,\Delta} \geq \sum_{i=j}^{n} S_i^{n,\Delta}.$$

Therefore, for H_n we have:

$$H_n \leq (k/S^{n,\Delta}) \sum_{j=1}^{n} S_j^{n+1,\Delta},$$

then by using Equation 6.2,

$$H_n \leq kS^{n+1,\Delta}/S^{n,\Delta}.$$

Finally, by Lemma 22, $H_n \leq k(\gamma S^{n,\Delta} - \gamma)/S^{n,\Delta} = O(1)$. $H_n \leq kO(1) = O(1)$. □

It should be mentioned that this constant average time complexity is without considering the input or the output time.

6.3 Ranking and Unranking Algorithms

By designing a generation algorithm in a specific order, the ranking algorithm is desired. In this section, ranking and unranking algorithms for these trees in A-order will be given. Ranking and unranking algorithms usually use a precomputed table of the number of a subclass of given trees with some specified properties to achieve efficient time complexities; these precomputations will be done only once and stored in a table for further use. Recall that $S^{n,\Delta}$ denotes the number of T_n^{Δ} trees. Let $S_{m,d}^{n,\Delta}$ be the number of T_n^{Δ} trees whose first subtree has **exactly** m nodes and its root has maximum degree d and $D_{m,d}^{n,\Delta}$ be the number of T_n^{Δ} trees whose first subtree has **at most** m nodes and its root has maximum degree d.

Theorem 20

- $D_{m,d}^{n,\Delta} = \sum_{i=1}^{m} S_{i,d}^{n,\Delta}$,
- $S^{n,\Delta} = \sum_{i=1}^{n-1} S_{i,\Delta}^{n,\Delta}$.

Proof: The proof is trivial. □

Theorem 21

$$S_{m,d}^{n,\Delta} = S_{m,1}^{m+1,\Delta} \times \sum_{i=1}^{n-m-1} (S_{i,d-1}^{n-m,\Delta}).$$

Proof: Let T be a T_n^{Δ} tree whose first subtree has exactly m nodes and its root has maximum degree d; by the definition and as shown in Figure 6.7 the number of the possible cases for the first

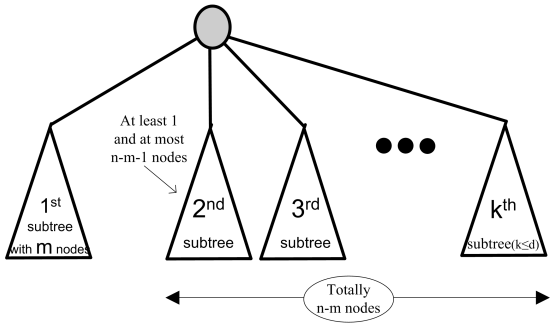


Figure 6.7: T_n^{Δ} tree whose first subtree has exactly m nodes and its root has maximum degree d.

subtree is $S_{m,1}^{m+1,\Delta}$ and the number of cases for the other parts of the tree is: $\sum_{i=1}^{n-m-1}(S_{i,d-1}^{n-m,\Delta})$. So:

$$S_{m,d}^{n,\Delta} = S_{m,1}^{m+1,\Delta} \times \sum_{i=1}^{n-m-1}(S_{i,d-1}^{n-m,\Delta}).$$

□

Now, let T be a T_n^{Δ} tree whose subtrees are defined by $T_1, T_2, \ldots, T_k$ and for $1 \leq i \leq k \leq \Delta$: $|T_i| = n_i$ and $\sum_{i=1}^{k} n_i = n-1$. For computing the rank of T, we have to enumerate the number of trees generated before T. Let *Rank(T,n)* be the rank of T. The number of T^{Δ} trees whose first subtree is smaller than T_1 is equal to:

$$\sum_{i=1}^{n_1-1} S_{i,\Delta}^{n,\Delta} + (Rank(T_1, n_1) - 1) \times \sum_{i=1}^{n-n_1} S_{i,\Delta-1}^{n-n_1,\Delta},$$

and the number of T^{Δ} trees whose first subtree is equal to T_1 but the second subtree is smaller than T_2 is equal to:

$$\sum_{i=1}^{n_2-1} S_{i,\Delta-1}^{n-n_1,\Delta} + (Rank(T_2, n_2) - 1) \times \sum_{i=1}^{n-n_1-n_2} S_{i,\Delta-2}^{n-n_1-n_2,\Delta}.$$

Similarly, the number of T^{Δ} trees whose first $(j-1)$ subtrees are equal to $T_1, T_2, \ldots, T_{j-1}$ and the j^{th} subtree is smaller than T_j is equal to:

$$\sum_{i=1}^{n_j-1} S_{i,\Delta-j+1}^{(n-\sum_{\ell=1}^{j-1} n_\ell),\Delta} + (Rank(T_j, n_j) - 1) \times \sum_{i=1}^{n-\sum_{\ell=1}^{j} n_\ell} S_{i,\Delta-j}^{n-\sum_{\ell=1}^{j} n_\ell,\Delta}.$$

Therefore, regarding enumerations explained above, for given tree $T \in T_n^{\Delta}$ whose subtrees are defined by $T_1, T_2, \ldots, T_k$, we can write:

$$\begin{aligned} Rank(T,1) &= 1, \\ Rank(T,n) &= 1 + \sum_{j=1}^{k} (\sum_{i=1}^{n_j - 1} S_{i,\Delta-j+1}^{(n-\sum_{\ell=1}^{j-1} n_\ell),\Delta} + \\ &+ (Rank(T_j, n_j) - 1) \sum_{i=1}^{n-\sum_{\ell=1}^{j} n_\ell} S_{i,\Delta-j}^{(n-\sum_{\ell=1}^{j} n_\ell),\Delta}). \end{aligned}$$

Hence, from Theorem 20, by using $D_{m,d}^{n,\Delta} = \sum_{i=1}^{m} S_{i,d}^{n,\Delta}$, we have:

$$\begin{aligned} Rank(T,1) &= 1, \\ Rank(T,n) &= 1 + \sum_{j=1}^{k} (D_{(n_j-1),(\Delta-j+1)}^{(n-\sum_{\ell=1}^{j-1} n_\ell),\Delta} + \\ &+ (Rank(T_j, n_j) - 1) D_{(n-\sum_{\ell=1}^{j} n_\ell),(\Delta-j)}^{(n-\sum_{\ell=1}^{j} n_\ell),\Delta}). \end{aligned}$$

To achieve the most efficient time for ranking and unranking algorithms, we need to precompute $D_{m,d}^{n,\Delta}$ and store it for further use. Assuming Δ is constant, to store $D_{m,d}^{n,\Delta}$ values, a 3-dimensional table denoted by $D[n, m, d]$ is enough, this table will have a size of $O(n \times n \times \Delta) = O(n^2)$ and can be computed using Theorems 20 and 21 with time complexity of $O(n \times n \times \Delta) = O(n^2)$. Note that for ranking and unranking of trees without a simple structure, it is expected to have a quadratic or even cubic precomputation [66].

To compute the rank of a codeword stored in array C, we also need an auxiliary array $N[i]$ which keeps the number of nodes in the subtree whose root is labeled by $C[i]$ and corresponds to n_i in the above formula. This array can be computed by a pre-order traversal or a level first search (DFS) algorithm just before we call the ranking algorithm.

The pseudo-code for ranking algorithm is given in Figure 6.8. In this algorithm, Beg is the variable that shows the positions of the first character in the array C whose rank is being computed (Beg is initially set to 1), and Fin is the variable that returns the position of the last character of C.

Now the time complexity of this algorithm is discussed. Obviously computing the array $N[i]$ takes $O(n)$. Hence we discuss the complexity of ranking algorithm which was given in Figure 6.8.

Theorem 22 *The ranking algorithm has the time complexity of $O(n)$ (with a preprocessing of time*

```
Function Rank( Beg : integer; var Fin: integer) ;
Var R, Point, PointFin, j, Nodes, n: integer;
begin
    n := N[Beg];
    if (n = 1) then begin
        Fin := Beg; return(1)  end;
    else begin
        Point := Beg + 1; R := 0; Nodes := 0; j := 1;
        while ( Nodes < n ) do begin
            R := R + D[n − Nodes, N[Point] − 1, Δ − j + 1]+
                (Rank(Point, PointFin) − 1)×
                D[(n − Nodes − N[Point]), (n − Nodes − N[Point]), Δ − j];
            Nodes := Nodes + N[Point]; j:=j+1;
            Point := PointFin + 1;
        end;
        Fin := Point − 1;
        return( R + 1);
    end;
end
```

Figure 6.8: Ranking algorithm for T_n^Δ trees.

and space $O(n^2)$).

Proof: Let T be a T_n^Δ tree whose subtrees are defined by T_1, T_2, ..., T_k and for $1 \leq i \leq k \leq \Delta$: $|T_i| = n_i$ and $\sum_{i=1}^{k} n_i = n-1$, and let $T(n)$ be the time complexity of ranking algorithm, then we can write:

$$T(n) = T(n_1) + T(n_2) + \ldots + T(n_k) + \alpha k,$$

where α is a constant and αk is the time complexity of the non-recursive parts of the algorithm. By using induction, we prove that if β is a value greater than α then $T(n) \leq \beta n$. We have $T(1) \leq \beta$. We assume $T(m) \leq \beta(m-1)$ for each $m < n$, therefore:

$$T(n) \leq \beta(n_1 - 1) + \beta(n_2 - 1) + \ldots + \beta(n_k - 1) + \alpha k,$$
$$T(n) \leq \beta(n_1 + \ldots + n_k - k) + \alpha k,$$
$$T(n) \leq \beta n - \beta k + \alpha k,$$
$$T(n) \leq \beta n.$$

So the induction is complete and we have $T(n) \leq \beta n = O(n)$. □

Before giving the description of the unranking algorithm we need to define two new operators.

- If a and b are integer numbers then $a\ div^+\ b$ is defined as follows:
  - If $b \nmid a$ then $a\ div^+\ b$ is equal to $(a\ div\ b)$.

  - If $b \mid a$ then $a\ div^+\ b$ is equal to $(a\ div\ b) - 1$.

- If a and b are integer numbers then $a\ mod^+\ b$ is defined as follows:
  - If $b \nmid a$ then $a\ mod^+\ b$ is equal to $(a\ mod\ b)$.
  - If $b \mid a$ then $a\ mod^+\ b$ is equal to b.

For unranking algorithm, we need the values of $S^{n,\Delta}$, these values can be stored in an array of size n, denoted by $S[n]$ (we assume Δ is constant). The unranking algorithm is the reverse approach of the ranking algorithm, the unranking algorithm is given in Figure 6.9. In this algorithm, the rank R is the input, Beg is a variable showing the position of the first character in the global array C and initially is set to 1. The generated codeword will be stored in array C. The variable n is the number of nodes and *Root* stores the character corresponding to the node we consider for the unranking procedure. For the next character we have two possibilities. If the root is r or s then the next character, if exists, will be ℓ or s (based on the number of root's children). If the root is m or ℓ, we have again two possible cases: if all the nodes of the current tree are not produced then the next character is m otherwise the next character will be r.

Theorem 23 *The time complexity of the unranking algorithm is $O(n \log n)$ (with a preprocessing of time and space $O(n^2)$).*

Proof: Let T be a T_n^Δ tree whose subtrees are defined by T_1, T_2, ..., T_k and for $1 \leq i \leq k \leq \Delta$: $|T_i| = n_i$ and $\sum_{i=1}^{k} n_i = n - 1$, and let $T(n)$ be the time complexity of the unranking algorithm. With regards to the unranking algorithm, the time complexity of finding j such that $D[n, j, \Delta - ChildNum + 1] \geq R$ for each T_i of T is $O(\log n_i)$, therefore, we have:

$$T(n) = O(\log n_1 + \log n_2 + \ldots + \log n_k) + T(n_1) + T(n_2) + \ldots + T(n_k).$$

We want to prove that $T(n) = O(n \log n)$. In order to obtain an upper bound for $T(n)$ we do as follows. First we prove this assumption for $k = 2$ then we generalize it. For $k = 2$ we have $T(n) = O(\log(n_1) + \log(n_2)) + T(n_1) + T(n_2)$. Let $n_1 = x$ then we can write the above formula as

$$T(n) = T(x) + T(n - x) + O(\log(x) + \log(n - x)) = T(x) + T(n - x) + C' \log(n).$$

For proving that $T(n) = O(n \log(n))$ we use an induction on n. We assume $T(m) \leq Cm \log(m)$ for all $m \leq n$, thus in $T(n)$ we can substitute

```
Function UnRank ( R, Beg, n: integer; Root: char);
var Point, i, t, ChildNum: integer;
begin
    if ( (n = 0) or (R = 0) ) then return(Beg − 1)
    else begin
        if (n = 1) then begin
            C[Beg] := Root; return(Beg);
        end;
        else begin
            C[Beg] := Root; Point := Beg + 1;
            Root := 'ℓ'; ChildNum := 0;
            while (n > 0) do begin
                ChildNum + +;
                find the smallest i that D[n, i, Δ − ChildNum + 1] ≥ R;
                R := R − D[n, i − 1, Δ − ChildNum + 1];
                if (n − i)= 1 then
                    if (ChildNum = 1) then  Root := 's';
                    else  Root := 'r';
                t := S[n];
                Point := UnRank( (div+(R, t)) + 1, Point, i, Root ) +1;
                R := mod+(R, t);
                n := n − i; Root := 'm';
            end;
            return(Point − 1);
        end;
    end;
end
```

Figure 6.9: Unranking algorithm for T_n^{Δ} trees.

$$T(n) \leq C \times x \log(x) + C \times (n-x) \log(n-x) + C' \log(n).$$

Let $f(x) = C \times x \log(x) + C \times (n-x) \log(n-x)$, now the maximum value of $f(x)$ with respect to x and by considering n as a constant value can be obtained by evaluating the derivation of $f(x)$ which is $f'(x) = C \times \log(x) - C \times \log(n-x)$. Thus if $f'(x) = 0$ we get $x = (n-1)/2$ and by computing $f(1)$, $f(n-2)$ and $f((n-1)/2)$ we have:

$$f(1) = f(n-2) = C \times (n-2) \log(n-2),$$
$$f((n-1)/2) = 2C \times ((n-1)/2) \times \log((n-1)/2) \ < C \times (n-2) \log(n-2).$$

so the maximum value of $f(x)$ is equal to $C \times (n-2) \log(n-2)$ and therefore

$$T(n) \leq C \times (n-2) \log(n-2) + C' \times \log(n).$$

It is enough to assume $C = C'$, then

$$T(n) \leq C \times (n-2) \log(n) + C \times \log(n) \ \leq C \times n \log(n).$$

Now, for generalizing the above proof and proving $T(n) = O(n \log n)$, we should find the maximum of the function $f(n_1, n_2, \ldots, n_k) = \prod_{i=1}^{k} n_i$. By the Lagrange method we prove that the maximum value of $f(n_1, n_2, \ldots, n_k)$ is equal to $(\frac{n}{k})^k$. Then $\frac{\delta f}{\delta k} = (\frac{n}{k})^k(\ln(\frac{n}{k}) - 1) = 0$, and

$$\ln(\tfrac{n}{k}) - 1 = 0,$$
$$\tfrac{n}{k} = e \Rightarrow k = \tfrac{n}{e},$$

so the maximum value of $f(n_1, n_2, \ldots, n_k)$ is equal to $e^{\frac{n}{e}}$. We know that:

$$T(n) = O(\log n_1 + \log n_2 + \ldots + \log n_k) + T(n_1) + T(n_2) + \ldots + T(n_k),$$

so

$$T(n) = O(\log(\textstyle\prod_{i=1}^{k} n_i) + \sum_{i=1}^{k} T(n_i),$$
$$T(n) < O(\log(n^{\frac{n}{e}})) + \textstyle\sum_{i=1}^{k} T(n_i),$$
$$T(n) < O(\tfrac{n}{e} \log e) = O(n) + \textstyle\sum_{i=1}^{k} T(n_i).$$

Finally, by using induction, we assume that for any $m < n$ we have $T(m) < \beta m \log m$, therefore:

$$T(n) = O(n) + \textstyle\sum_{i=1}^{k} T(n_i),$$
$$T(n) < O(n) + \textstyle\sum_{i=1}^{k} \beta O(n_i \log n_i),$$
$$T(n) < O(n) + \beta \log(\textstyle\prod_{i=1}^{k} (n_i^{n_i})),$$
$$T(n) < O(n) + O(\log(n^n)),$$
$$T(n) = O(n \log n).$$

Hence, the proof is complete. □

6.4 Summary

In this chapter, we studied the problem of generation, ranking and unranking of ordered trees of size n and maximum degree Δ; we presented an efficient algorithm for the generation of these trees in A-order with an encoding over 4 letters and size n. Moreover, two efficient ranking and unranking algorithms were designed for this encoding. The generation algorithm has $O(n)$ time complexity in the worst case and $O(1)$ in the average case. The ranking and unranking algorithms have $O(n)$ and $O(n \log n)$ time complexity, respectively. The presented ranking and unranking algorithms use a precomputed table of size $O(n^2)$ (assuming Δ is constant). To our best knowledge, the only previous work on this class of trees was an inefficient generation algorithm [15].

Chapter 7

Conclusions and Discussion

Trees are one of the most important basic and simple data structures for organizing information in computer science. A great amount of research has been done in developing new data structures for organizing data. The memory hierarchies of modern computers are composed of several levels of memories, starting from the caches. Caches have very small access time and capacity comparing to main memory and external memory. From cache to main memory, then to external memory, access time and capacity increases significantly. Two main memory models to evaluate the I/O complexity are external-memory model [2] and cache-oblivious model [39, 81].

In the external-memory model, accessing an item from external storage is extremely slow. Transferring a block between the internal memory and the external memory takes constant time. Computations performed within the internal memory are considered of taking no time at all and this is because the external memory is so much slower than the random access memory [70]. We assume that each external memory access transmits one page of B elements.

Cache-oblivious model [39, 81] allows to consider only a two-level hierarchy, while proving results for a hierarchy composed of an unknown number of levels. In this model, memory has blocks of size B words, which B is an unknown parameter and a cache-oblivious algorithm is completely unaware of the value of B used by the underlying system.

We introduced the core partitioning scheme, which maintains a balanced search tree as a dynamic collection of complete balanced binary trees called cores. Using this technique we achieve the same theoretical efficiency of modern cache-oblivious data structures by using the classic structures such as weight-balanced trees or height balanced trees such as AVL trees. We show that these "classic data structures" can be efficiently used in external-memory/cache-oblivious models. In fact, we

obtain the same optimal results obtained by the data structures purposely designed for external-memory/cache-oblivious models. We preserve the original topology and algorithms of the given balanced search tree using a simple post-processing with guaranteed performance to completely rebuild the changed cores (possibly all of them) after each update. Using our core partitioning scheme, we show how to store balanced trees such as weight-balanced trees and height-balanced trees (AVL trees), so that they simultaneously achieve efficient memory allocation, space-efficient representation, and cache-obliviousness. When performing updates, we show that weight-balanced trees can be maintained with a logarithmic cost, while AVL trees require super poly-logarithmic cost according to a lower bound on the subtree size of the rotated nodes.

The notion of core partition shows how to obtain cache-efficient versions of the classical balanced binary search trees such as AVL trees and weight-balanced trees. A natural question is whether the core partition can be also applied to arbitrary binary search trees which can be unbalanced. We give a positive answer to this question: the resulting data structure, called Cache-Oblivious General Balanced Tree (COG-tree), can be seen as a smooth extension of Anderson's General Balanced Tree to the cache-oblivious model with transfer block size B. Both the COG-trees and the core partitioning on weight-balanced trees occupies linear space and have an improved cache complexity of $O(\log_B n)$ amortized block transfers and $O(\log n)$ amortized time for updates, and $O(\log_B n)$ block transfers and $O(\log n)$ time for the search operation.

We also introduced the gaps in AVL trees. Gaps are special tree edges such that the height difference between the subtrees, rooted at their two endpoints, is equal to 2. We showed how to express the size of a given AVL tree in terms of the heights of the gaps. Using that, the size of any AVL tree can be characterized with a very simple and useful formula and we can describe the precise relationship between 'the size and the heights of the nodes' and 'the subtree sizes and the heights of the gaps', we can also independently describe the relationship between the heights of the nodes and the heights of the gaps. We have also studied gaps' behavior in an AVL tree during a sequence of insertions and deletions. Gaps have been also exploited in some of our other results.

As known, an insertion in an n-node AVL tree takes at most two rotations, but a deletion in an n-node AVL tree can take $\Theta(\log n)$. A natural question is whether deletions can take many rotations not only in the worst case but in the amortized case as well? Heaupler, Sen, and Tarjan's [48] conjectured that alternating insertions and deletions in an n-node AVL tree can cause each deletion to do $\Omega(\log n)$ rotations. We proved that conjuncture is true by providing a construction which causes each deletion to do $\Omega(\log n)$ rotations: we showed that, for infinitely many n, there is a set

E of *expensive* n-node AVL trees with the property that, given any tree in E, deleting a certain leaf and then reinserting it produces a tree in E, with the deletion having done $\Theta(\log n)$ rotations. One can do an arbitrary number of such expensive deletion-insertion pairs. The difficulty in obtaining such a construction is that in general the tree produced by an expensive deletion-insertion pair is not the original tree. Indeed, if the trees in E have odd height h, $2^{(h-1)/2}$ deletion-insertion pairs are required to reproduce the original tree.

Finally as a byproduct of our research, we introduced a new encoding over an alphabet of size 4 for representing unlabeled ordered trees with maximum degree Δ. We use this encoding for generating these trees in A-order with $O(1)$ average time and $O(n)$ worst case time complexity. Due to the given encoding, both ranking and unranking algorithms are also designed taking $O(n)$ and $O(n \log n)$ time complexities (with a precomputation of size and time $O(n^2)$).

For the future works, the main problem would be applying core partitioning scheme on the remaining binary search trees such as red-black trees or 2-3 trees, investigating that if we can obtain efficient results in cache-oblivious/external-memory model.

Bibliography

[1] G. M. Adel'son-Vel-skii and E. Landis, *An algorithm for the organization of information*, Soviet Mathematics Doklady, 3 (1962), pp. 1259–1263.

[2] A. Aggarwal and J. S. Vitter, *The input/output complexity of sorting and related problems*, Communications of the ACM, 31 (1988), pp. 1116–1127.

[3] M. Amani, *A short talk on core partition scheme; advantages and applications.* Paper presented at Workshop on Graph Theory, Algorithms and Applications 3rd Edition, Erice, Italy, 2014.

[4] M. Amani, A. Bernasconi, R. Grossi, and L. Pagli, *Demonstrating avl trees using gaps.* Paper presented at the Combinatorics 2014, Gaeta, Italy, 2014.

[5] M. Amani, K. A. Lai, and R. E. Tarjan, *Amortized rotation cost in avl trees*, Information Processing Letters, 116 (2016), pp. 327–330.

[6] M. Amani and A. Nowzari-Dalini, *Generation, ranking and unranking of ordered trees with degree bounds*, in Proceedings of Eleventh International Workshop on Developments in Computational Models, Cali, Colombia, October 28, 2015, Electronic Proceedings in Theoretical Computer Science, 2015, pp. 31–45.

[7] ——, *Ranking and unranking algorithm for neuronal trees in b-order*, Journal of Physical Sciences, 20 (2015), pp. 19–34.

[8] A. Anderson, *General balanced trees*, Journal of Algorithms, 30 (1999), pp. 1–28.

[9] A. Andersson and T. W. Lai, *Fast updating of well-balanced trees*, in Proceedings of 2nd Scandinavian Workshop on Algorithm Theory, Springer, 1990, pp. 111–121.

[10] A. Andersson and M. Thorup, *Dynamic ordered sets with exponential search trees*, Journal of the ACM, 54 (2007), pp. 13:1–13:40.

[11] C. R. Aragon and R. G. Seidel, *Randomized search trees*, in Proceedings of 30th Annual Symposium on Foundations of Computer Science, IEEE, 1989, pp. 540–545.

[12] L. Arge, *The buffer tree: A technique for designing batched external data structures*, Algorithmica, 37 (2003), pp. 1–24.

[13] L. Arge and J. S. Vitter, *Optimal external memory interval management*, SIAM Journal on Computing, 32 (2003), pp. 1488–1508.

[14] R. Aringhieri, P. Hansen, and F. Malucelli, *Chemical trees enumeration algorithms*, Quarterly Journal of the Belgian, French and Italian Operations Research Societies, 1 (2003), pp. 67–83.

[15] T. S. Balaban, P. A. Filip, and O. Ivanciuc, *Computer generation of acyclic graphs based on local vertex invariants and topological indices. derived canonical labelling and coding of trees and alkanes*, Journal of Mathematical Chemistry, 11 (1992), pp. 79–105.

[16] K. Barksdale and S. Turner, *HTML, JavaScript, and Advanced Internet Technologies*, Course Technology Press, 2005.

[17] R. Bayer, *Symmetric binary B-trees: Data structure and maintenance algorithms*, Acta Informatica, 1 (1972), pp. 290–306.

[18] R. Bayer and E. McCreight, *Organization and maintenance of large ordered indices*, in Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control, ACM, 1970, pp. 107–141.

[19] R. Bayer and E. M. McCreight, *Organization and maintenance of large ordered indexes*, Acta Informatica, 1 (1972), pp. 173–189.

[20] M. Bender, E. D. Demaine, M. Farach-Colton, et al., *Cache-oblivious b-trees*, in Proceedings of 41st Annual Symposium on Foundations of Computer Science, IEEE, 2000, pp. 399–409.

[21] M. A. Bender, R. Cole, and R. Raman, *Exponential structures for efficient cache-oblivious algorithms*, in Proceedings of Automata, Languages and Programming, 29th International Colloquium, 2002, vol. 2380 of Lecture Notes in Computer Science, Springer, 2002, pp. 195–207.

[22] M. A. Bender, E. D. Demaine, and M. Farach-Colton, *Efficient tree layout in a multilevel memory hierarchy*, in Proceedings of Algorithms - ESA 2002, 10th Annual European Symposium, 2002, pp. 165–173.

[23] M. A. Bender, E. D. Demaine, and M. Farach-Colton, *Cache-oblivious b-trees*, SIAM Journal on Computing, 35 (2005), pp. 341–358.

[24] M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. R. Fogel, B. C. Kuszmaul, and J. Nelson, *Cache-oblivious streaming B-trees*, in Proceedings of the 19th Annual ACM Symposium on Parallelism in Algorithms and Architectures, ACM, 2007, pp. 81–92.

[25] S. B. Berger and L. W. Tucker, *Binary tree representation of three-dimensional, reconstructed neuronal trees: a simple, efficient algorithm*, Computer methods and programs in biomedicine, 23 (1986), pp. 231–235.

[26] M. Berry and P. Bradley, *The application of network analysis to the study of branching patterns of large dendritic fields*, Brain research, 109 (1976), pp. 111–132.

[27] T. Beyer and S. M. Hedetniemi, *Constant time generation of rooted trees*, SIAM Journal on Computing, 9 (1980), pp. 706–712.

[28] N. Blum and K. Mehlhorn, *On the average number of rebalancing operations in weight-balanced trees*, Theoretical Computer Science, 11 (1980), pp. 303–320.

[29] G. S. Brodal, R. Fagerberg, and R. Jacob, *Cache oblivious search trees via binary trees of small height*, in Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 6-8, 2002, San Francisco, CA, USA, ACM/SIAM, 2002, pp. 39–48.

[30] G. Caporossi, I. Gutman, and P. Hansen, *Variable neighborhood search for extremal graphs: IV: chemical trees with extremal connectivity index*, Computers & Chemistry, 23 (1999), pp. 469–477.

[31] G. Chartrand and O. R. Oellermann, *Applied and algorithmic graph theory*, McGraw-Hill, 1993.

[32] D. Comer, *Ubiquitous b-tree*, ACM Computing Surveys, 11 (1979), pp. 121–137.

[33] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, The MIT Press, 3 ed., 2009.

[34] M. De Berg, M. Van Kreveld, M. Overmars, and O. C. Schwarzkopf, *Computational geometry*, Springer, 2000.

[35] K. De Queiroz and J. Gauthier, *Phylogeny as a central principle in taxonomy: phylogenetic definitions of taxon names*, Systematic Biology, 39 (1990), pp. 307–322.

[36] A. A. Dobrynin and I. Gutman, *The average wiener index of trees and chemical trees*, Journal of Chemical Information and Computer Sciences, 39 (1999), pp. 679–683.

[37] M. C. Er, *Efficient generation of k-ary trees in natural order*, The Computer Journal, 35 (1992), pp. 306–308.

[38] C. C. Foster, *A generalization of avl trees*, Communications of the ACM, 16 (1973), pp. 513–517.

[39] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandra, *Cache-oblivious algorithms*, in Proceedings of the 40th Annual Symposium on Foundations of Computer Science, 1999, pp. 285–298.

[40] H. Fujiwara, J. Wang, L. Zhao, H. Nagamochi, and T. Akutsu, *Enumerating treelike chemical graphs with given path frequency*, Journal of Chemical Information and Modeling, 48 (2008), pp. 1345–1357.

[41] I. Galperin and R. L. Rivest, *Scapegoat trees*, in Proceedings of the Fourth Annual ACM-SIAM Symposium of Discrete Algorithms, vol. 93, 1993, pp. 165–174.

[42] I. P. Goulden and D. M. Jackson, *Combinatorial Enumeration*, Wiley, New York, 1983.

[43] L. J. Guibas and R. Sedgewick, *A dichromatic framework for balanced trees*, in 19th Annual Symposium on Foundations of Computer Science, IEEE, 1978, pp. 8–21.

[44] I. Gutman, *Graphs and graph polynomials of interest in chemistry*, Springer Berlin Heidelberg, 1987, pp. 177–187.

[45] I. Gutman, P. Hansen, and H. Mélot, *Variable neighborhood search for extremal graphs. 10. comparison of irregularity indices for chemical trees*, Journal of Chemical Information and Modeling, 45 (2005), pp. 222–230.

[46] I. Gutman and O. E. Polansky, *Mathematical concepts in organic chemistry*, Springer Berlin Heidelberg, 1986.

[47] B. Haeupler, S. Sen, and R. E. Tarjan, *Rank-balanced trees*, in Proceedings of Algorithms and Data Structures, 11th International Symposium, vol. 5664 of Lecture Notes in Computer Science, Springer, 2009, pp. 351–362.

[48] B. Haeupler, S. Sen, and R. E. Tarjan, *Rank-balanced trees*, ACM Transactions on Algorithms, 11 (2015), pp. 30:1–30:26.

[49] R. A. Hankins and J. M. Patel, *Effect of node size on the performance of cache-conscious b+-trees*, in Proceedings of ACM SIGMETRICS Performance Evaluation Review, vol. 31, ACM, 2003, pp. 283–294.

[50] P. Hansen, B. Jaumard, C. Lebatteux, and M. Zheng, *Coding chemical trees with the centered n-tuple code*, Journal of Chemical Information and Computer Sciences, 34 (1994), pp. 782–790.

[51] J. B. Hendrickson and C. A. Parks, *Generation and enumeration of carbon skeletons*, Journal of Chemical Information and Computer Sciences, 31 (1991), pp. 101–107.

[52] S. Heubach, N. Y. Li, and T. Mansour, *Staircase tilings and k-catalan structures*, Discrete Mathematics, 308 (2008), pp. 5954–5964.

[53] Y. Horibe, *Notes on fibonacci trees and their optimality*, The Fibonacci Quarterly, 21 (1983), pp. 118–128.

[54] S. Huddleston and K. Mehlhorn, *A new data structure for representing sorted lists*, Acta informatica, 17 (1982), pp. 157–184.

[55] A. Jovanović and D. Danilović, *A new algorithm for solving the tree isomorphism problem*, Computing, 32 (1984), pp. 187–198.

[56] P. C. Kanellakis, S. Ramaswamy, D. E. Vengroff, and J. S. Vitter, *Indexing for data models with constraints and classes*, in Proceedings of the 12th ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems, ACM, 1993, pp. 233–243.

[57] D. E. Knuth, *The art of computer programming: sorting and searching*, vol. 3, Pearson Education, 1998.

[58] J. F. Korsh, *Generating t-ary trees in linked representation*, The Computer Journal, 48 (2005), pp. 488–497.

[59] J. F. Korsh and P. LaFollette, *Loopless generation of gray codes for k-ary trees*, Information processing letters, 70 (1999), pp. 7–11.

[60] D. L. Kreher and D. R. Stinson, *Combinatorial algorithms: generation, enumeration, and search*, Discrete Mathematics and Its Applications, CRC press, 1998.

[61] A. LaMarca and R. Ladner, *The influence of caches on the performance of heaps*, Journal of Experimental Algorithmics (JEA), 1 (1996), p. 4.

[62] A. LaMarca and R. E. Ladner, *The influence of caches on the performance of sorting*, Journal of Algorithms, 31 (1999), pp. 66–104.

[63] K. S. Larsen, *Avl trees with relaxed balance*, Journal of Computer and System Sciences, 61 (2000), pp. 508–522.

[64] T. J. Lehman and M. J. Carey, *Query processing in main memory database management systems*, vol. 15, ACM, 1986.

[65] M. Lepovic and I. Gutman, *A collective property of trees and chemical trees*, Journal of chemical information and computer sciences, 38 (1998), pp. 823–826.

[66] L. Li, *Ranking and unranking of avl-trees*, SIAM Journal on Computing, 15 (1986), pp. 1025–1035.

[67] F. Luccio and L. Pagli, *On the height of height-balanced trees*, Computers, IEEE Transactions on, 100 (1976), pp. 87–90.

[68] E. Mäkinen, *A survey on binary tree codings*, The Computer Journal, 34 (1991), pp. 438–443.

[69] K. Manes, A. Sapounakis, I. Tasoulas, and P. Tsikouras, *Recursive generation of k-ary trees*, Journal of Integer Sequences, 12, article 09.7.7 (2009), pp. 1–18.

[70] P. Morin, *Open Data Structures (in Java)*, 2013. `http://opendatastructures.org/ods-java.pdf`.

[71] S.-I. Nakano and T. Uno, *Efficient generation of rooted trees*, technical report, 2003.

[72] ——, *Constant time generation of trees with specified diameter*, in 30th International Workshop on Graph-Theoretic Concepts in Computer Science, Springer, 2004, pp. 33–45.

[73] J. Nievergelt and E. M. Reingold, *Binary search trees of bounded balance*, SIAM Journal on Computing, 2 (1973), pp. 33–43.

[74] S. Nilsson and G. Karlsson, *Internet programming fast IP routing with LC-tries: Achieving gbit/sec speed in software*, Dr. Dobb's Journal of Software Tools, 23 (1998), pp. 70–75.

[75] O. O. Oluwagbemi, E. F. Adebiyi, S. Fatumo, and A. Dawodu, *Pq trees, consecutive ones problem and applications*, International Journal of Natural and Applied Sciences, 4 (2008), pp. 262–277.

[76] M. H. Overmars, *The design of dynamic data structures*, Springer-Verlag Inc., 1983.

[77] J. M. Pallo, *Enumerating, ranking and unranking binary trees*, The Computer Journal, 29 (1986), pp. 171–175.

[78] ——, *Generating trees with n nodes and m leaves*, International journal of computer mathematics, 21 (1987), pp. 133–144.

[79] ——, *A simple algorithm for generating neuronal dendritic trees*, Computer methods and programs in biomedicine, 33 (1990), pp. 165–169.

[80] J. M. Pallo and R. Racca, *A note on generating binary trees in a-order and b-order*, International Journal of Computer Mathematics, 18 (1985), pp. 27–39.

[81] H. Prokop, *Cache-oblivious algorithms*, Master's thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 1999.

[82] A. Proskurowski and F. Ruskey, *Binary tree gray codes*, Journal of Algorithms, 6 (1985), pp. 225–238.

[83] S. Ramaswamy, *Efficient indexing for constraint and temporal databases*, in Database Theory - ICDT '97: 6th International Conference Delphi, Springer Berlin Heidelberg, 1997, pp. 419–431.

[84] J. Rao and K. A. Ross, *Cache conscious indexing for decision-support in main memory*, in Proceedings of 25th International Conference on Very Large Data Bases, 1998, pp. 78–89.

[85] ——, *Making b+-trees cache conscious in main memory*, in Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, vol. 29, ACM, 2000, pp. 475–486.

[86] D. Rotem and Y. L. Varol, *Generation of binary trees from ballot sequences*, Journal of the ACM, 25 (1978), pp. 396–404.

[87] F. Ruskey, *Generating t-ary trees lexicographically*, SIAM Journal on Computing, 7 (1978), pp. 424–439.

[88] F. Ruskey and T. C. Hu, *Generating binary trees lexicographically*, SIAM Journal on Computing, 6 (1977), pp. 745–758.

[89] R. Saikkonen and E. Soisalon-Soininen, *Cache-sensitive memory layout for binary trees*, in Proceedings of 5th Ifip International Conference On Theoretical Computer Science, Springer, 2008, pp. 241–255.

[90] H. Samet, *The design and analysis of spatial data structures*, vol. 85, Addison-Wesley Reading, 1990.

[91] H. Samet and R. E. Webber, *Hierarchical data structures and algorithms for computer graphics. i. fundamentals*, Computer Graphics and Applications, IEEE, 8 (1988), pp. 48–68.

[92] R. Seidel and C. R. Aragon, *Randomized search trees*, Algorithmica, 16 (1996), pp. 464–497.

[93] S. Sen and R. E. Tarjan, *Deletion without rebalancing in balanced binary trees*, in Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SIAM, 2010, pp. 1490–1499.

[94] M. Shimizu, H. Nagamochi, and T. Akutsu, *Enumerating tree-like chemical graphs with given upper and lower bounds on path frequencies*, BMC Bioinformatics, 12 (2011), pp. 1–9.

[95] D. D. Sleator and R. E. Tarjan, *Self-adjusting binary search trees*, Journal of the ACM, 32 (1985), pp. 652–686.

[96] M. Solomon and R. A. Finkel, *A note on enumerating binary trees*, Journal of the ACM, 27 (1980), pp. 3–5.

[97] R. P. Stanley, *Enumerative combinatorics*, vol. 2, Cambridge University Press, Cambridge, 1999.

[98] P. S. Stevens, *Patterns in Nature*, Little Brown & Company, 1974.

[99] I. P. Stewart, *Quadtrees: Storage and scan conversion*, The Computer Journal, 29 (1986), pp. 60–75.

[100] A. E. Trojanowski, *Ranking and listing algorithms for k-ary trees*, SIAM Journal on Computing, 7 (1978), pp. 492–509.

[101] A. K. Tsakalidis, *Rebalancing operations for deletions in AVL-trees*, RAIRO - Theoretical Informatics and Applications - Informatique Théorique et Applications, 19 (1985), pp. 323–329.

[102] T. Uehara and W. M. v. Cleemput, *Optical layout of cmos functional arrays*, IEEE Transactions on Computers, 30 (1981), pp. 305–312.

[103] V. Vajnovszki, *Listing and random generation of neuronal trees coded by six letters*, The Automation, Computers, and Applied Mathematics, 4 (1995), pp. 29–40.

[104] V. Vajnovszki and J. M. Pallo, *Generating binary trees in a-order from codewords defined on a four-letter alphabet*, Journal of Information and Optimization Sciences, 15 (1994), pp. 345–357.

[105] D. R. van Baronaigien and F. Ruskey, *Generating t-ary trees in a-order*, Information Processing Letters, 27 (1988), pp. 205–213.

[106] J. S. Vitter, *Algorithms and data structures for external memory*, Foundations and Trends in Theoretical Computer Science, 2 (2006), pp. 305–474.

[107] D. B. West, *Introduction to graph theory*, vol. 2, Prentice hall Upper Saddle River, 2001.

[108] H. S. Wilf and N. A. Yoshimura, *Ranking rooted trees, and a graceful application*, Annals of the New York Academy of Sciences, 576 (2006), pp. 633–640.

[109] P. Willett, J. M. Barnard, and G. M. Downs, *Chemical similarity searching*, Journal of chemical information and computer sciences, 38 (1998), pp. 983–996.

[110] R. A. Wright, B. Richmond, A. Odlyzko, and B. D. McKay, *Constant time generation of free trees*, SIAM Journal on Computing, 15 (1986), pp. 540–548.

[111] R.-Y. Wu, J.-M. Chang, and C.-H. Chang, *Ranking and unranking of non-regular trees with a prescribed branching sequence*, Mathematical and Computer Modelling, 53 (2011), pp. 1331–1335.

[112] R.-Y. Wu, J.-M. Chang, and Y.-L. Wang, *A linear time algorithm for binary tree sequences transformation using left-arm and right-arm rotations*, Theoretical Computer Science, 355 (2006), pp. 303–314.

[113] L. Xiang, K. Ushijima, and C. Tang, *On generating k-ary trees in computer representation*, Information processing letters, 77 (2001), pp. 231–238.

[114] K. Yamanaka, Y. Otachi, and S.-i. Nakano, *Efficient enumeration of ordered trees with k leaves*, in WALCOM: Algorithms and Computation, Springer, 2009, pp. 141–150.

[115] S. Zaks, *Lexicographic generation of ordered trees*, Theoretical Computer Science, 10 (1980), pp. 63–82.

[116] D. Zerling, *Generating binary trees using rotations*, Journal of the ACM, 32 (1985), pp. 694–701.

[117] B. Zhuang and H. Nagamochi, *Constant time generation of trees with degree bounds*, in 9th International Symposium on Operations Research and Its Applications, 2010, pp. 183–194.