Università degli Studi di Pisa

Dipartimento di Informatica
Dottorato di Ricerca in Informatica

Ph.D. Thesis

Ranked Queries in Index Data Structures

Iwona Bialynicka-Birula

Supervisor
Roberto Grossi

October 15, 2008

Abstract

A ranked query is a query which returns the top-ranking elements of a set, sorted by rank, where the rank corresponds to some sort of preference function defined on the items of the set. This thesis investigates the problem of adding rank query capabilities to several index data structures on top of their existing functionality. Among the data structures investigated are suffix trees, range trees, and hierarchical data structures. We explore the problem of additionally specifying rank when querying these data structures. So, for example in the case of suffix trees, we would like to obtain not all of the occurrences of a substring in a text or in a set of documents, but to obtain only the most preferable results. What is most important, the efficiency of such a query must be proportional to the number of preferable results and not all of the occurrences, which can be too many to process efficiently.

First, we introduce the concept of rank-sensitive data structures. Rank-sensitive data structures are defined through an analogy to output-sensitive data structures. Output-sensitive data structures are capable of reporting the items satisfying an online query in time linear to the number of items returned plus a sublinear function of the number of items stored. Rank-sensitive data structures are additionally given a ranking of the items and just the top k best-ranking items are reported at query time, sorted in rank order. The query must remain linear only with respect to the number of items returned, which this time is not a function of the elements satisfying the query, but the parameter k given at query time. We explore several ways of adding rank-sensitivity to different data structures and the different trade-offs which this incurs.

Adding rank to an index query can be viewed as adding an additional dimension to the indexed data set. Therefore, ranked queries can be viewed as multi-dimensional range queries, with the notable difference that we additionally want to maintain an ordering along one of the dimensions. Most range data structures do not maintain such an order, with the exception of the Cartesian tree. The Cartesian tree has multiple applications in range searching an other fields, but is rarely used

for indexing due to its rigid structure which makes it difficult to use with dynamic content. The second part of this work deals with overcoming this rigidness and describes the first efficient dynamic version of the Cartesian tree.

Acknowledgments

I would like to thank everybody at the University of Pisa, especially my advisor Professor Roberto Grossi, Professor Andrea Maggiolo Schettini, and Professor Pierpaolo Degano, who made my studies at the University of Pisa and this work possible. I would also like to thank Professor Anna Karlin for granting me guest access to the courses at the University of Washington. Last but not least, I would like to thank my family (Iwo Białynicki-Birula, Zofia Białynicka-Birula, Piotr Białynicki-Birula, Stefania Kube) and my husband (Vittorio Bertocci) without whose continuing faith and support this work would not have been possible.

Contents

List of Tables

List of Figures

Introduction

In recent years we have been literally overwhelmed by the electronic data available in fields ranging from information retrieval, through text processing and computational geometry to computational biology. Making sense of an ever-increasing torrent of data is becoming more and more of a problem.

An obvious example of this phenomenon is the web search engine. Web search engines are designed to answer queries which return all documents containing a given phrase, or phrases, from the set of all documents available on the World Wide Web. Unless the query phrase is extremely specific, which is usually not the case, it potentially returns a very large number of results. This number is typically so large, that neither the search engine itself, nor the initiator of the query can have any chance of processing these results in a reasonable amount of time.

In the early days of the Internet, web search engines did indeed attempt to return all of the results of a web search query, but as the Web exploded in size, it became quickly apparent that the paradigm of the web query needs to be refined. This is when the next generation of search engines came into being: search engines, which do not only return the documents containing a query word, but returned these results *sorted according to rank*. We will not delve into the subtleties of defining such a rank here. It is only important to note that this rank tries to reflect a user preference on the results, so that higher-ranked results are more likely to be the results that the user would like to consider first.

The key point to ranked queries is the fact that these queries do not really return all of the results of the query, but only a small subset of the highest-ranked ones. This way not only the user is not overwhelmed by a large amount of data, but also the engine has a chance of answering the query efficiently.

Ranked queries have proved so far the most successful remedy for the enormity of online data, but document retrieval is not the only field in which the amount of data is growing rapidly and its processing is a growing concern. Let us take, for example, computational biology. Biological databases containing various sequences

(for example genetic information expressed as sequences of nucleotides, or protein structures expressed as sequences of amino acids) are growing exponentially as the mechanisms used to obtain this data are enhanced from year to year.

Biological databases are but one example of an increasingly unmanageable information source. Numerous other examples exist, from geographical databases to the results of demographical marketing studies. The simple keyword search used for web queries does not always suffice when extracting information from these data stores. Sometimes there is need to employ other types of queries, such as full-text search or some kind of range-query (possibly multi-dimensional). The problem with such queries is that they only work efficiently if the number of results returned is small, which is not always the case.

This brings us to the motivation for this work. Full-text search and range queries [15] are among the most common methods for retrieving various types of data. Suffix trees [87] and range trees [14] are very popular and thoroughly studied index data structures used for efficiently answering such queries. These structures are also output-sensitive, which means that the main component in the complexity of the query time is linear with respect to the number of items returned. This linear component is indeed the best one can do if we actually want to return all of the results of the query. But, as with the web search example, this can be of little use if the number of items returned is huge, because then not only does the query take long to process, but the result is unmanageable to the user.

However, as with web queries, very often we do have some sort of additional preference regarding the items returned by the query. It may even be much more naturally definable than the rank of a web page. In the case of a full-text search we may want to simply consider the results in the order of their appearance. Or, if the full-text search spans a number of documents, we may have a preference regarding the containing documents. If the data represents graphical objects in two dimensions, we may want to consider them according to a third dimension (the z-order). When querying various databases, we may want to access records in the order of their physical location (to minimize disk seek time) or according to a time order (for example in the case of news items).

This thesis investigates the problem of adding rank query functionality to a class of output-sensitive index data structures, among which the foremost are suffix tree and range trees. The first part of the work surveys related work. It also provides an overview of the two structures which we work on making rank-sensitive — suffix trees and range trees.

The second part introduces the concept of a *rank-sensitive data structure* and describes several different implementations of such structures. In particular, it explores how rank-sensitivity can be added to suffix trees and range trees. The first approach to adding rank sensitivity to suffix trees is explored both in a theoretical and experimental setting. The second approach is more general framework which can be applied to any structure in the family considered.

The last part of the thesis deals with a related problem, which is the dynamization of Cartesian trees [85], a data structure which intrinsically organizes elements according to rank in addition to another criterion. We show how Cartesian trees, previously only studied in the static context, can be made dynamic. This is an important first step in potentially improving rank-sensitive structure performance and could also lead to the solution of other problems.

Part I

Background

Chapter 1

State of the Art

The problem we are investigating is that of turning an output-sensitive data structure, such as a suffix tree or a range tree, into something even more powerful — a rank-sensitive data structure, which additionally has a rank associated with each element it stores. While the output-sensitive data structure returns a set Q of results in response to a query, the rank-sensitive counterpart takes an additional query parameter k and returns a subset of Q of size k containing the k highest ranked elements of Q, sorted by rank. Moreover, the linear component of the query time is not proportional to $|Q|$ as in the case of output-sensitive data structures, but to k.

While ranking itself has been the subject of intense theoretical investigation in the context of search engines [55, 57, 70, 20], we could not find any explicit study pertaining to ranking in the context of data structures.

The only known data structure which can actually be considered rank-sensitive according to our definition is the inverted index [93, 90]. An inverted index is a structure used in most search engines for indexing large bodies of documents. The documents are first tokenized to create sets of words contained in each document. This mapping of documents to words is then inverted to create a list of containing documents for each encountered word. This makes it easy to efficiently return all documents containing a given word. Moreover, the inverted lists of documents can be sorted according to the rank of the documents, which makes it possible to efficiently return the best-ranking results.

Inverted indexes, however, do not solve our general problem due mainly to the initial tokenization process. This process can only be applied when a natural tokenization method exists, which is true in the case of natural language documents, but not so in the case of, for example, biological databases. It also causes the loss of proximity information between the words in the document. This makes the inverted

index much less efficient for answering more complicated queries, such as substring search.

An indirect form of ranking can be found in the (dynamic) rectangular intersection with priorities [53]. This work presents a structure for answering rectangle stabbing queries which returns the highest priority (or best-ranking) rectangle, which is a special case of a rank-sensitive query, with $k = 1$.

A somewhat related problem is the document listing problem described in [65]. Muthukrishnan's structure builds on the suffix tree [87] to answer substring queries. However, it additionally takes into account that the source underlying the indexed text is not a single document, but a collection of documents. It then returns not all of the occurrences of the pattern in the text, but all of the documents containing the pattern. Moreover, the structure is output-sensitive, which means that the linear component is proportional to the number of containing documents and not to the total number of occurrences as in the case of the regular suffix tree. In some cases, the latter can be a much smaller number and be a much more meaningful result to the user.

For a certain class of data structures, such as suffix trees, we can formulate the ranking problem as a geometric problem. Note that the results of a query on a suffix tree correspond to a range of its leaves. We can therefore associate two coordinates with each leaf e of the tree — one, $pos(e)$, corresponding to its inorder position in the tree and the other, $rank(e)$ corresponding to its rank. Let the range e_i to e_j be the result of one query. The rank-sensitive version of the query takes parameter k and is effectively a three-sided or $1\frac{1}{2}$-dimensional query on $pos(e_i) \dots pos(e_j)$ along the x-axis, and $0 \dots rank(e')$ along the y-axis, where e' is the kth entry in rank order such that $pos(e_i) \leq pos(e') \leq pos(e_j)$.

Priority search trees [59] and Cartesian trees [85] are among the prominent data structures supporting these queries, but do not provide items in sorted order (they can end up with half of the items unsorted during their traversal). Since we can identify the aforementioned e' by a variation of [41], in $O(k)$ time, we can retrieve the top k best-ranking items in $O(\log n + k)$ time in *unsorted order*. Improvements to get optimal $O(k)$ time can be made using scaling [44] and persistent data structures [32, 37, 54, 52]. Priority search trees are described in detail in Section 1.1.

What if we adopt the above solution in a *real-time* setting? Think of a server that provides items in rank order on the fly, or any other similar real-time application in which guaranteed response time is mandatory. Given a query, the above solution and its variants can only start listing the first items after $O(t(n) + k \log k)$ time,

where the $t(n)$ component is the search time and $k \log k$ accounts for the reporting time of the output-sensitive query and the time to sort the items reported according to rank. In contrast, rank-sensitive data structures work in real-time. After $O(t(n))$ time, they provide each subsequent item in $O(1)$ worst-case time according to the rank order (i.e. the qth item in rank order is listed after $c_1 t(n) + c_2 q$ steps, for $1 \leq q \leq k$ and constants $c_1, c_2 > 0$).

Persistent data structures Persistent data structures [32, 37, 54, 52] are dynamic structures which provide access not only to their latest versions, but also to all of their intermediate versions, as opposed to ephemeral data structures which provide access only to their most recent version. Persistent data structures which allow access to all of their versions, but allow the modification only of the most recent version are called partially persistent data structures. Fully persistent data structures are those in which each intermediate version can be both accessed an modified. There exists also the notion of confluently persistent data structures which refers to fully persistent data structures which support merge operations involving more than one intermediate version.

In [32], the authors present a general technique for making any pointer-based data structure partially persistent with only a constant overhead on time and space complexity and logarithmic access time to any version with respect to the number of versions. This technique can be used to solve the static version of our problem in the following way. Let us use $e_0, e_1, \ldots, e_{n-1}$ to denote the list of leaves of a suffix tree, as in the preceding paragraph. The entries e_i for increasing i are sorted according to the structure of the tree and a range e_i to e_j is the result of a query.

Using the partially persistent version of the linked list, we create $O(\log n)$ lists of lengths $2^0, 2^1, \ldots, 2^{\lfloor \log n \rfloor}$. The first version of the list of length 2^r contains elements $e_0, e_1, \ldots, e_{2^r-1}$ sorted in rank order. The second version does not contain e_0, but instead contains e_{2^r} and is still sorted according to rank — and so on: each subsequent version is obtained from the previous by removing one element of the list and adding another, so that version i contains elements $e_{i-1}, e_i, \ldots, e_{2^r+i-2}$ in rank order. Each subsequent list version adds only a constant overhead to the persistent structure, because removing and adding elements in a linked list modifies only a constant number of nodes. Since the structure is persistent, we can access any of the versions in logarithmic time with respect to the number of versions, so in effect we can get any subset of $e_0, e_1, \ldots, e_{n-1}$ whose length is a power of two sorted according to rank.

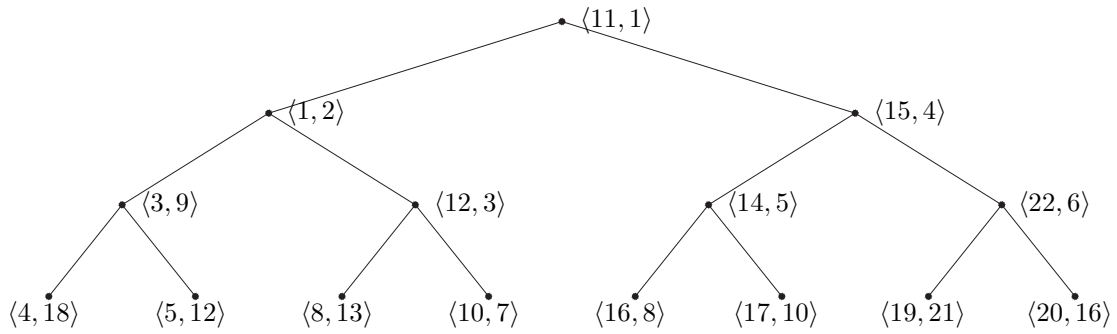

Figure 1.1: A sample priority search tree.

In order to retrieve any subset of leaves e_i to e_j sorted according to rank, we compute the largest r such that $2^r \leq j - i$. Among the versions of the sublists for 2^r entries, we take the one starting at e_i and the one ending in e_j. Merging these two lists on the fly for k steps solves our problem. This solution uses $O(n \log n)$ space and the query time is $O(\log n + k)$. It only works in the static setting since a single change in $e_0, e_1, \ldots, e_{n-1}$ can affect $\Theta(n)$ linked list versions in the worst case.

1.1 Priority search trees

As mentioned in the preceding section, priority search trees solve a problem which is somewhat similar to our rank problem — that of answering three-sided queries (see Definition 1.1.1). The structure was first introduced by McCreight in [59].

Definition 1.1.1 *Given a set D of order pairs $\langle x, y\rangle \in D$ and values x_0, x_1, y_1, a three-sided query returns all pairs $\langle x, y\rangle \in D$ such that $x_0 \leq x \leq x_1$ and $y \leq y_1$.*

A priority search tree (PST) for a set D of points $\langle x, y\rangle$ is a balanced binary tree in which each tree node can be mapped to one of the points in D. Additionally, a pivot value $P(v)$ is associated with each node v of the tree. The following conditions hold:

- For any two points $\langle x_1, y_1\rangle \in D$ and $\langle x_2, y_2\rangle \in D$, if the node associated with $\langle x_1, y_1\rangle$ is an ancestor of the node associated with $\langle x_2, y_2\rangle$, then $y_1 \leq y_2$.
- For any two points $\langle x_1, y_1\rangle \in D$ and $\langle x_2, y_2\rangle \in D$, if the node associated with $\langle x_1, y_1\rangle$ belongs to the left subtree of some node v and the node associated with $\langle x_2, y_2\rangle$ belongs to the right subtree of v, then $x_1 \leq P(v)$ and $x_2 > P(v)$.

Operation	Complexity
PST construction	$O(n \log n)$
Single element insertion	$O(\log n)$
Single element deletion	$O(\log n)$
Three-sided query (k is the number of items reported)	$O(\log n + k)$

Table 1.1: Operations supported with the use of a priority search tree and their corresponding complexities.

- Priority search trees are balanced (i.e. their height is $O(\log n)$ where $n = |D|$).

See Figure 1.1 for an example of a priority search tree.

1.1.1 Complexity

A priority search tree storing n items occupies $O(n)$ space. The supported operations and their complexities are summarized in Table 1.1. Note that with respect to the three-sided query, priority search trees are output-sensitive. Insertions and deletions can be performed on a dynamic priority search tree, which is a variant of the structure covered in detail in Section 1.1.4.

1.1.2 Construction

A priority search tree PST(D) for the set D of points $\langle x, y \rangle$ can be constructed according to the following recursive definition:

- If $D = \emptyset$, then PST(D) is an empty tree.
- If $D \neq \emptyset$, then:
  - Let $\langle x', y' \rangle$ be a point in D such that for each $\langle x, y \rangle \in D$, $y' \leq y$.
  - Let $D' = D - \langle x', y' \rangle$.
  - Let x'' be the median of the x coordinates of the points in D'.
  - Let D_L be the set of $\langle x_L, y_L \rangle \in D'$ such that $x_L \leq x''$.
  - Let D_R be the set of $\langle x_R, y_R \rangle \in D'$ such that $x_R > x''$.
  - The root of PST(D) is a node v associated with point $\langle x', y' \rangle$. $P(v) = x''$. The left child of the root is PST(D_L) and the right child of the root is PST(D_R).

Note that this construction produces a balanced tree as long as the x coordinates of the points in D are distinct (an assumption which can always be made, because in the case of repeating values we may introduce an additional distinction). The construction takes $O(n \log n)$ time, where $n = |D|$.

1.1.3 Three-sided query

A three-sided query (see definition 1.1.1) on a priority search tree, given parameters x_0, x_1, y_1 and a priority search tree PST(D) is performed as follows.

1. If the tree is empty, return an empty set.
2. If the tree is not empty, then:
   (a) Let v be the root of the tree and let $\langle x', y' \rangle$ be its associated point and let $P(v)$ be its associated pivot point.
   (b) Let PST(D_L) and PST(D_R) be the left and right subtrees of v respectively.
   (c) If $y' > y_1$, return an empty set.
   (d) If $y' \geq y_1$, then:
      i. If $x_0 \leq x' \leq x_1$, report $\langle x', y' \rangle$.
      ii. If $x_0 \leq P(v)$, recursively search PST(D_L).
      iii. If $x_1 > P(v)$, recursively search PST(D_R).

It is clear that this algorithm reports only the points satisfying the query. Parts of the tree omitted in Step 2c of the algorithm do not contain relevant points on account of the heap property of the tree. Parts of the tree omitted in steps 2(d)ii and 2(d)iii of the algorithm do not contain relevant points on account of the ordering of the points in the tree with respect to the pivot point. Therefore, the algorithm lists exactly the points satisfying the query.

One can base the complexity analysis of the query algorithm on categorizing the points visited by the algorithm. It is clear that each point is visited at most once. Each visited point falls into one of the three categories:

1. Points which are reported.
2. Points which are not reported, because the condition in Point 2(d)i fails.

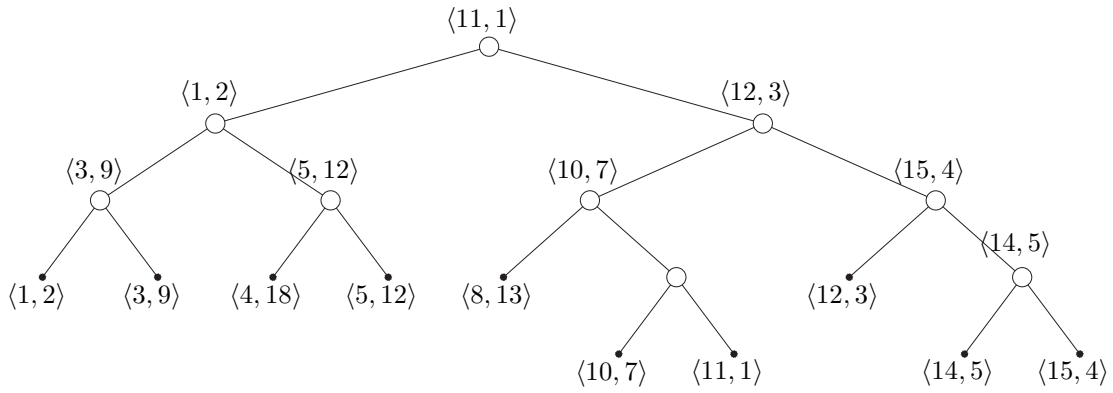

Figure 1.2: A sample dynamic priority search tree.

3. Points which are not reported, because the condition in Point 2c fails.

There are k points in category number 1, where k is the number of items satisfying the query.

On each level of the tree there at most two nodes which fall into category number 2. This is because on each level the inorder order of the nodes corresponds to the order of the x coordinates of their corresponding points. Due to Step 2(d)ii of the algorithm, if we visit a point in this category which is to the left of the x_0–x_1 range, then we will not visit points to the left of this point. Analogically, due to Step 2(d)iii of the algorithm, if we visit a point in this category which is to the right of the x_0–x_1 range, then we will not visit points to the right of this point. Therefore, there are at most $O(\log n)$ points in this category, because the tree is balanced and has logarithmic height.

As for category number 3, if a point falls into it, then its parent must fall into categories 1 or 2. This is because after Step 2c of the algorithm we do not visit the child nodes of the node in question. Therefore, there at most as many points in this category as there are points in the remaining two categories.

Overall, the complexity of the algorithm is $O(\log n + k)$ where n is the size of the tree and k is the number of items satisfying the query.

1.1.4 Dynamic priority search trees

Dynamic priority search trees are slightly different from their static counterparts. In a dynamic priority search tree, each point is associated with a leaf of the tree and may also be associated with one of the internal nodes. Therefore, a tree storing n

points has $2n - 1$ nodes. The inorder order of the leaves corresponds to the x order of the points. Each internal node v is associated with the point with the smallest y among the points associated with the leaves descending from v which is not already associated with a parent of v (such a point may not exist, in which case v is not associated with any point). See Figure 1.2 for an example of a dynamic priority search tree.

The tree is balanced using the balancing mechanism of the red-black tree [26].

The dynamic version of the priority search tree has the same heap property with respect to the y coordinate and ordering with respect to the x coordinate (points stored in the left subtree of a node all have the x coordinate smaller than the items stored in its right subtree), so the three-sided query can be performed in the same way.

Dynamic priority search tree construction Dynamic priority search trees can be constructed in a bottom-up fashion. First, the points are sorted according the their x coordinate to create the leaves of the tree. Pairs of consecutive leaves are joined to create the lowest level of internal nodes. Each internal node is associated with the point with smaller y among the two leaves. In each subsequent step internal nodes are a joined to create a higher level of internal nodes. When two internal nodes are joined, one of the points stored in the two nodes being joined is moved to the parent, leaving the lower node "free". At this point, a point from a lower internal node is moved up, and so on until the bottom is reached or no point remains to move up (in which case the node remains empty).

Sorting the elements takes $O(n \log n)$ time. Then, there are $O(n)$ joins which may cause nodes moving up, but those are limited by the height of the tree, hence $O(\log n)$ time. Overall, the construction takes $O(n \log n)$ time as with static priority search trees.

Insertion Insertion into the dynamic PST requires a number of steps. First, the new leaf needs to be added. The location of the new leaf can easily be found using the tree itself, because the tree is a balanced search tree with respect to the leaves, so it is enough to follow a path from the root according to the usual binary search tree rules. Once a leaf is reached, it is replaced with an internal node with this leaf as one of its children and the new leaf as the other child.

Following the insertion of the leaf, the internal node values need to be updated. This stage is accomplished by "pushing down" the new node along a path which

starts at the root of the tree. The y value of the point associated with each internal node considered is compared to the y value of the point being pushed down. If the y value of the pushed point is greater, then the algorithm continues down according to the x coordinate of the pushed point. If the y value is smaller, then internal node is associated with the pushed point. After this, the algorithm takes the old point associated with this internal node and continues down, pushing down this old point until the bottom of the tree or an empty internal node is reached.

After each insertion, balance is maintained by re-balancing the underlying red-black tree using the standard red-black tree rotations [26]. Note that these rotations by definition maintain the binary search tree order on the x coordinates of the nodes. As for the heap order, this can again be corrected using the pushdown operations.

Since the rotations and pushdown operations all pertain to just one path in tree, the overall insertion time is bound by $O(\log n)$.

Deletion Similarly to insertions, deletions require removing the associated leaf, updating the values associated with the internal nodes, and possibly re-balancing the tree.

Again the associated leaf can be found and removed by using the binary tree properties of the PST. Internal nodes need to updated on the path from the leaf to internal node associated with the deleted point. This can be done bottom-up using the same algorithm as is used in the construction of the tree.

Finally, rotations may be required in order to re-balance the tree. These can be performed just like the rotations resulting from the insertions, that is, by coupling them with any necessary pushdown operation required to maintain the heap order of the tree.

1.1.5 Priority search trees and ranked queries

As mentioned earlier in this chapter, we can use priority search trees to solve a problem similar to our ranking problem, although the resulting structure is not fully rank-sensitive. Let us take a data structure, such as a suffix tree, such that the result of the query is a sublist of a list $e_0 \dots e_{n-1}$ — in the case of the suffix tree it is a sublist of the leaves of the tree ordered according to their (inorder) position in the tree.

If we treat the positions of the elements in this list $pos(e)$ as x coordinates and ranks of the elements $rank(e)$ as y coordinates, then the top-k query for the range

of elements e_i to e_j is actually a three-sided query with $pos(e_i)$ and $pos(e_j)$ as the x bounds and $rank(e')$ as the y bound, where e' is the kth entry in rank order such that $pos(e_i) \leq pos(e') \leq pos(e_j)$.

We can identify the element e' by a variation of [41], in $O(k)$ time. However, the three-sided query returns items in unsorted order, so sorting them according to rank takes another $O(k \log k)$ time. Therefore the overall query time is $O(\log n + k \log k)$, so the priority tree-based structure for answering ranked queries offers only superlinear query complexity with respect k and hence is not rank-sensitive.

1.2 Cartesian trees

Cartesian trees have been introduced 25 years ago by Vuillemin [84, 85]. Due to their unique properties (cf. [44]), they have found numerous applications in priority queue implementations [40], randomized searching [73], range searching [74], range maximum queries [44], least common ancestor queries [12, 48], integer sorting [6], string algorithms [34] and memory management [77], to name a few.

1.2.1 Definition

A Cartesian tree T is a binary tree, in which each node is associated with a pair of values. We may view these values as points $\langle x, y \rangle$ in the Cartesian plane. The nodes of T satisfy the following conditions:

1. The order on the x-coordinates of the points matches the *inorder* of the nodes in T.

2. The order on the y-coordinates of the points complies with the *heap* order on the nodes in T.

1.2.2 Construction

From the definition of the Cartesian tree, we may deduce the following recursive algorithm for building it, given a set of points to store in its nodes:

1. Take the point $\langle \bar{x}, \bar{y} \rangle$ with the maximum y-value in the set (y-values can be viewed as a priority measure). This point has to be the root of the tree, otherwise the heap condition (Condition 2) would be violated.

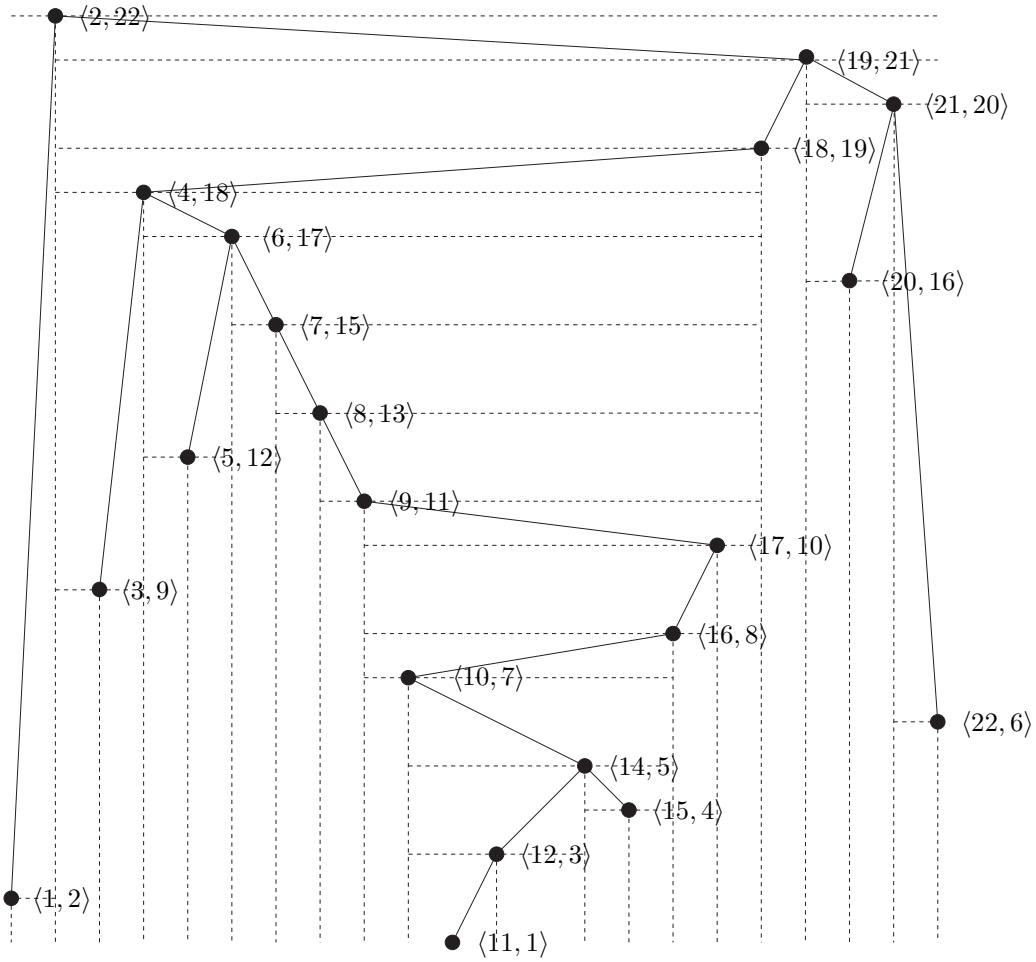

Figure 1.3: An example of a Cartesian tree. The nodes storing points $\langle x, y\rangle$ are located at coordinates (x, y) in the Cartesian plane. The edges $e = (\langle x_L, y_L\rangle, \langle x_R, y_R\rangle)$ are depicted as lines connecting the two coordinates (x_L, y_L) and (x_R, y_R). The tree induces a subdivision of the Cartesian plane, which is illustrated by dotted lines.

2. The x-value of the root, $\bar{x}$, induces a partition of the remaining elements into sets $L = \{\langle x, y\rangle : x < \bar{x}\}$ and $R = \{\langle x, y\rangle : x > \bar{x}\}$. Set L must constitute the left subtree of $\langle \bar{x}, \bar{y}\rangle$ and set R must constitute the the right subtree, in order to maintain Condition 1. So the roots of the Cartesian trees obtained from L and R are the left and right children of the root $\langle \bar{x}, \bar{y}\rangle$, respectively.

Example 1.2.1 Figure 1.3 shows a sample Cartesian tree containing a set of 22 points.

This construction takes $O(n \log n)$ time. If we have the points in the tree already sorted according to their first coordinate, we can construct the tree left to right in linear time in the following fashion. Suppose C_i is the Cartesian tree containing points $\langle x_0, y_0\rangle \ldots \langle x_i, y_i\rangle$. We construct C_{i+1} by adding $\langle x_{i+1}, y_{i+1}\rangle$ somewhere on the rightmost path of C_i. We do it by traversing the rightmost path from the most recently added node upwards to find the place to add $\langle x_{i+1}, y_{i+1}\rangle$. The key observation is that each node can only join the rightmost path once and leave it also just once, so even if in one step we traverse i steps of the rightmost path, we are shrinking the rightmost path by i elements, so the total construction time is $O(n)$.

Up to now Cartesian trees have only been studied in the static setting. In Part III we explore how to make this structure dynamic.

1.3 Weight-balanced B-trees

Weight-balanced B-trees [7] are a type of balanced dynamic search tree, which have the unique property the a rebalancing operation on a node v with x descendants happens only every $O(x)$ times. This property is very useful when additional data is associated with each node, the data is $O(x)$ in size and needs to be reorganized when the node is rebalanced. In such a case, due to the property, the cost of rebuilding the associated data can be amortized. This can be exploited in the case of dynamic operations on range trees (see Section 2.2.6). We also use this property in our dynamic version of Cartesian trees (see Section 5.8.1).

1.3.1 B-trees

B-trees [9, 10, 25] are a dynamic search trees with amortized logarithmic query and update time. They are the most commonly used structure in database and file system applications mainly because they behave well in external memory.

Each B-tree node other than the root has between a and $2a-1$ children[1]. The root of a B-tree has between 2 and $2a-1$ children. A node v with x children stores $x-1$ values in such a way as to maintain the search property, that is all of the values stored in the first i children of v and their descendants are less than the i-th value stored in v and the remaining children of v and their descendants store values greater than the i-th value in v.

All of the leaves of a B-tree are at the same level and the tree is logarithmic in height.

Query operations are performed as with binary search trees except more than one value has to be checked in each node. The number of values in each node is limited by the constant a, so the query time is $O(\log n)$.

As with binary trees, the insertion of an item starts with the search for this item ending in a leaf. If the found leaf has less than then maximum allowed number of values, then the value is inserted into the leaf. Otherwise, the leaf is split into two leaves, resulting in the middle value being pushed up into the parent. If the parent size is exceeded then it has to split as well and so on until a node is reached which is not full or the root is split and a new root is formed.

Since each value stored in a node acts as a separator value for the subtrees, deletion is a bit more involved. First, the value to be deleted is swapped out with a value from a child node in such a way as to maintain the search ordering. This continues until a leaf is reached. If the value can be removed from the leaf without invalidating the size of the leaf, then the operation terminates. Otherwise if a neighboring leaf is of more than the minimum size, then the elements are redistributed between the leaves so as to maintain the size requirements. In the case in which both the leaf in question and its neighbors are already of minimum size, the leaf is fused with its neighbor and one value from the parent node. This may leave the parent node with illegal size in which case the process propagates upwards until a node with more than minimum size is reached or the root is removed.

1.3.2 Weight-balanced B-trees

A weight-balanced B-tree is balanced based on the weight of each node rather than its degree. The weight of a node v is defined as the total number of elements stored in the leaves which descend from v (or in v if v itself is a leaf). A weight-balanced B-tree

[1] A more general structure is a weak B-tree or an (a,b)-tree [60] which limits the number of children to a range from a to b, where a and b are any positive numbers such that $2a \leq b$.

with branching parameter a and leaf parameter k has the following properties [7]:

1. All leaves are on the same level and store between k and $2k-1$ elements.

2. The weight $w(v)$ of a node v on level ℓ which is not the root satisfies $\frac{1}{2}a^\ell k < w(v) < 2a^\ell k$, where levels are numbered bottom-up starting from 0 at the leaf level.

3. The weight of the root at level ℓ is less than $2a^\ell k$ and the root has more than one child.

From the above conditions it follows that the nodes of the tree have no more than $4a$ children and all nodes except for the root have at least $a/4$ children and the height of the tree is logarithmic.

Insertion As with the regular B-tree, insertion starts at a leaf and may result in split operations which are propagated upwards until a node of less than maximum size is reached or the root splits and new root is formed. The difference here is that maximum size is determined by the weight and not the degree of a node, hence a node splits if it would otherwise have to exceed the maximum weight of $2a^\ell k - 1$. A node on level ℓ is split at the i-th value, where i is such that the weights of the nodes resulting from the split are within $2a^{\ell-1}k$ of $a^\ell k$. This is always possible due to the weight limit of each of the children.

The following lemma is crucial to our (Section 5.8.1) and other applications of weight-balanced B-trees.

Lemma 1.3.1 ([7]) *After splitting a node v on level ℓ into two nodes, v' and v'', at least $\frac{1}{2}a^\ell k$ insertions have to be performed below v' (or v'') before it splits again. After creating a new root in a tree with n elements, at least $3n$ insertions are performed before it splits again.*

Proof: A node splits when its weight reaches $2a^\ell k$ and results in nodes whose weight is at most $a^\ell k + 2a^{\ell-1}k < \frac{3}{2}a^\ell k$. Therefore the a weight increase of $\frac{1}{2}a\ell k$, and thus so many insertions are needed for the node to split again.

A root is created when its weight (the number of elements in the tree) reaches $2a^{\ell-1}k$ and splits when it reached $2a^\ell k > 8a^{\ell-1}k$ which is four times the size of the tree at the time of the root creation. □

Deletion Deletion in a weight-balanced B-tree can be performed using the *global rebuilding* technique [68]. This means that the deletions are implemented as *weak deletions*: marking an element as deleted, but not removing it from the structure. Once the number of weak-deleted elements reaches a predefined constant fraction of the total number of elements, the entire structure is rebuilt without the weak-deleted elements. This way the cost $O(n \log n)$ of rebuilding the structure is amortized over $O(n)$ deletions, so deletion effectively take $O(\log n)$ amortized time.

Chapter 2

Base data structures

2.1 Suffix trees

A suffix tree is an output-sensitive static data structure used for identifying the occurrences of a substring in a text, though it also has a variety of other uses (see Section 2.1.3).

A suffix tree is a special case of a compacted trie [42] or PATRICIA tree [63].

A trie is a tree used for storing a set of strings over an alphabet. Each edge in the tree is labeled with a character of the alphabet in such a way that all edges from a node to its children are labeled with distinct characters and the set of words obtained by concatenating the characters along each paths from the root to each of the leaves is the set of strings stored in the structure.

Tries can be used for quickly (in time proportional to the size of the input string) determining if a string belongs to a given set, as well as retrieving all strings with a given prefix, so they are widely used in spell-checkers and for auto-completion.

Most often in natural language applications, tries contain a large number of unary nodes, which makes them space-inefficient. PATRICIA trees are modified tries in which series of edges between any two branching nodes are compacted into single edges. Depending on the application, the labels of the compacted edges also have to be compacted somehow. If the PATRICIA tree is only used for verifying if a string belongs to the subset, it is enough to store the first edge label and the length of the original path in the compacted edge. If an application requires reproducing the entire string of characters along the original path, we may take advantage of the fact that every such path corresponds to a substring of the original input and store a reference to this original input in the compacted edge.

A suffix tree is a PATRICIA tree which stores all of the suffixes of a given text.

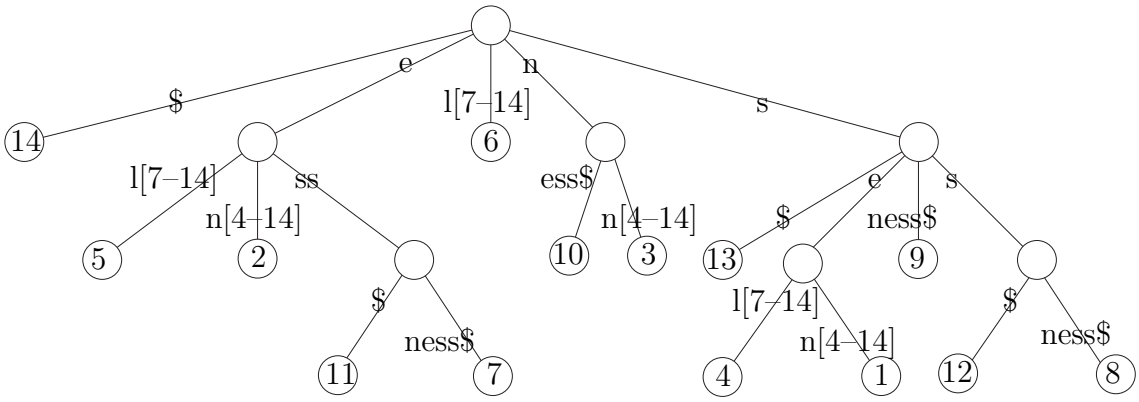

Figure 2.1: Suffix tree for the string *senselessness$*. Longer substrings are represented using start and end positions of the substring in the text. Leaves contain starting positions of their corresponding suffixes.

Since every substring of a text is prefix of some suffix of this text, every internal node of the trie of the suffixes corresponds to a substring of the text. Since the suffix tree is compacted though, some of the edges may correspond to more than one substring. As with the PATRICIA tree, edge labels in a suffix tree may correspond to long substrings of the text, but those can be represented using the start and end indexes of their corresponding substrings, provided the original text is also stored.

In order to maintain a useful one-to-one correspondence of text suffixes and tree leaves, the original text is padded at the end with an out-of-alphabet character, typically denoted \$. Generalized suffix trees [46] are suffix trees constructed for a set of texts by concatenating all of the texts into a single text while using out-of-alphabet characters in between them.

Example 2.1.1 Figure 2.1 shows a suffix tree for the text *senselessness$*. Longer substrings are represented using their start and end positions in the original text.

2.1.1 Complexity

Suffix trees for a text of length n occupy $O(n)$ memory words, provided the word is large enough to address the size of the original text and to store a character of the alphabet. They can be constructed in $O(n)$ time using an online algorithm which processes the text from left to right [80, 82] (see 2.1.2).

Determining if a substring of length m belongs to the indexed text can be performed in $O(m)$ time by using the same algorithm as used for pattern matching in a trie. Listing all of the occurrences of the substring involves locating its corresponding node and then listing all of the leaves descending from this node (provided the text is padded with the out-of-alphabet symbol at the end). Since the tree is compacted and all nodes have at least two children, this task takes $O(m + \ell)$ time, where ℓ is the number of occurrences of the substring in the text.

2.1.2 Construction

There are several other algorithms for constructing suffix trees which have been known earlier (see for example [87, 58, 23]). The Ukkonen algorithm [80, 82] became the most popular, though, because it is the first online one: It processes each character of the input text from left to right and in each step it has constructed the prefix of the text containing all of the characters processed so far. We will describe the Ukkonen algorithm in this section.

Ukkonen suffix tree construction is based on the construction of the corresponding uncompressed trie. Each node in the suffix tree corresponds to a node in the corresponding trie. Since the suffix tree is compressed, however, there are nodes in the trie which do not correspond to nodes in the suffix tree. Following [82] we will refer to the trie nodes as states and refer to those having a counterpart in the suffix tree as explicit states and the rest — implicit states. Implicit states can be understood as locations in the middle of a suffix tree edge. An implicit state can be uniquely defined by the parent of such an edge and the prefix of the edge label, referred to in [82] as the *canonical reference pair*.

The construction operates on an augmented version of the suffix tree being constructed. The augmentation involves adding an *auxiliary state* $\perp$ and *suffix links*. Suffix links are pointers from each node v to the node $f(v)$, where f is the *suffix function*. The suffix function f assigns a node to every node in the tree in the following way: For each node v other than the root, if v corresponds to the string ax for some character a, then $f(v)$ is the node corresponding to x. For v equal to the root, $f(v) = \perp$. Note that the transition function can also be defined for implicit states.

Crucial to the online construction of suffix trees is the introduction of *open edges*. As we mentioned earlier, each edge in the suffix tree is labeled with the start and end indexes of its corresponding substring. Note that this means that each edge

leading to a leaf has the current string length as the second part of its label. If we were to update these labels in each step of the online algorithm, the algorithm would end up quadratic. To address this, the notion of an open edge is introduced. An open edge is an edge leading to a leaf. Since we know that the second part of the label of such an edge is always the last index of the string, we do not store it explicitly, but instead denote it as ∞. This eliminates the need for updating all the edge labels of the open edges explicitly in each step.

The *boundary path* of the suffix tree is the path which follows suffix links from the deepest node (the one corresponding to the entire string) to the root and finally the auxiliary state $\perp$. Adding each subsequent letter to the end of the string on which the suffix tree is based involves updating the nodes on the boundary path. Ukkonen's algorithm is based on the observation that the boundary path can be split into three segments:

1. The first segment contains nodes at the end of edges which need to be extended by one when adding the new character.

2. The middle segment contains nodes at the end of edges which need to be split when adding the new character. The splitting of such an edge results in a new leaf.

3. The last segment contains nodes which already have a transition corresponding to the letter being added.

Due to the open edge concept, nodes belonging to the first segment do not need to be updated at all. Nodes belonging to the last segment do not need to updated either, since they already have the required transition. The only nodes which need to be updated are nodes belonging to the middle segment, but each of these creates a new leaf, so their creation can be amortized by the number of leaves in the tree $O(n)$.

The algorithm keeps track of the beginning of the middle segment, the so called *active point*. Therefore, it is able to traverse and update only the middle segment in each step. The overall complexity of the algorithm is $O(n)$ for the construction of a suffix tree for a text of length n.

2.1.3 Applications

Suffix trees are usually used as static data structures, because changes in the underlying text can potentially change the entire structure of the tree. For example,

a suffix tree for the text $aa...a\$$ is a binary "comb", while changing just the middle letter to form $aa...abaa...a\$$ reduces the height two-fold while increasing the degree of most nodes to three.

Apart from finding the occurrences of a substring in text, suffix trees have multiple other applications [45]. Those range from finding common substring of a set of strings or detecting palindromes [46], to clustering web search results [91]. Suffix trees can also be used for approximate string matching — see for example [81].

Multiple applications of suffix trees can be found in the field of computational biology where the indexed text represents a sequence of amino acids or nucleotides. One such application is genome alignment, a process which determines the similarity between two or more genomes [29, 28, 50]. Another application is signature selection [51]. Signature selection requires finding short subsequences which occur in only one sequence of a set of sequences thus identifying this sequence. Such signatures are used for building microarrays, since the sequences bind to different areas of the microarray allowing one to measure the number of sequences of each type — gene expression. Suffix trees are also used in computational biology to find tandem repeats [47] (consecutive occurrences of the same substring).

2.2 Range trees

Besides suffix trees, range trees are an another versatile output-sensitive data structure which can be made rank-sensitive using the techniques presented in this work (see Chapter 5).

Range trees have been introduced by Bentley [14] in 1980 and since then have been the base for most orthogonal range searching data structures [2].

Orthogonal range search is a special case of geometric range search. Geometric range search deals with searching a set of elements which can be represented as points in a multidimensional space and locating elements which fall within a specified geometric range.

With orthogonal range search the range is a multidimensional rectangle with sides parallel to the axis of the geometric space. This rectangle can be open-ended on some of its sides. Variations of orthogonal range search queries include listing all of the points in the rectangle, determining their number, or determining if any such points exist.

In this work, when speaking of orthogonal range search, we will mean the first variation (see Definition 2.2.1).

Definition 2.2.1 *Orthogonal range search query operates on a set of d-dimensional points* $(x_1, \ldots, x_d)$*. The query specifies a rectangle with corners at* $(a_1, \ldots, a_d)$ *and* $(b_1, \ldots, b_d)$ *and returns the set of points* $(x_1, \ldots, x_d)$ *such that* $a_i \leq x_i \leq b_i$ *for* $i = 1 \ldots d$*.*

2.2.1 Definition

The structure of the range tree can be defined recursively with respect to the dimensionality d of the points stored in the structure.

1-dimensional range trees A range tree in one dimension is a minimum-height binary tree with the elements stored in the leaves ordered from left to right from smallest to largest. The leaves are additionally organized in a doubly linked list. Alternatively, in the static model, the 1-dimensional range tree may also be represented simply as a sorted array of all of the elements.

d-dimensional range trees A d-dimensional range tree stores points from a d-dimensional space. It is a minimum-height binary tree with the points stored in the leaves and ordered from left to right according to the first coordinate. We call this tree the *primary tree* or *primary structure*. Each internal node v contains a $(d-1)$-dimensional range tree of the points stored in the subtree rooted at v projected onto a $(d-1)$-dimensional space by disregarding the first coordinate. We will call these trees the *secondary structures*.

See Figure 2.2 for an example of a 2-dimensional range tree.

2.2.2 Space complexity

A range tree storing n d-dimensional points occupies $O(n \log^{d-1} n)$ space. This can be shown through induction.

A 1-dimensional range tree occupies $O(n \log^{1-1} n) = O(n)$ space, because it is a binary tree with n leaves. Linking the leaves in a list does not induce additional complexity.

In a d-dimensional range tree each node stores a $(d-1)$-dimensional range tree holding all of its descending points. Since the tree is of minimum-height, there are $m = O(\log n)$ levels in the tree. There are $O(2^i)$ nodes at levels i and each of them is an ancestor of $O(\frac{n}{2^i})$ other nodes. Therefore, assuming (inductively) that

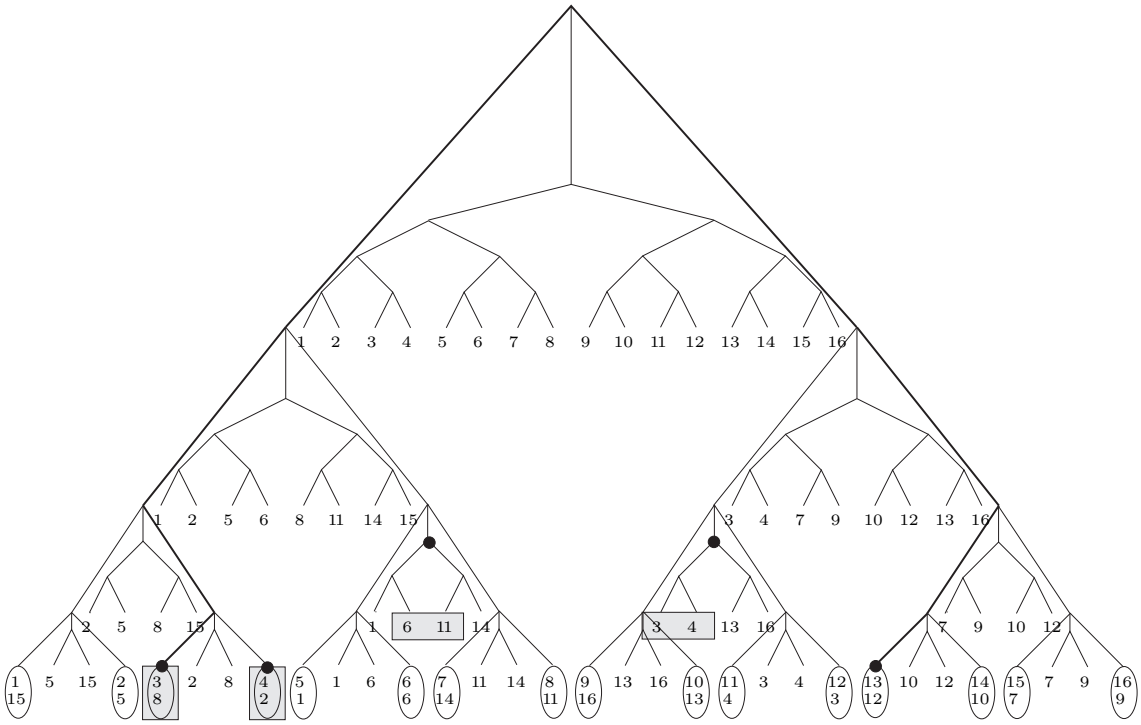

Figure 2.2: Range tree for the set of points $\{(1,15), (2,5), (3,8), (4,2), (5,1), (6,6), (7,14), (8,11), (9,16), (10,13), (11,4), (12,3), (13,12), (14,10), (15,7), (16,9)\}$. For clarity, the first coordinate has been omitted in the second-level tree nodes. The query $1 \leq x \leq 13$, $2 \leq y \leq 11$ is considered. The path of the query is marked using bold edges. The leaves and the roots of the second-level nodes considered for the query are marked with black dots. Finally, the ranges of points reported are indicated using gray rectangles.

a $(d-1)$-dimensional tree occupies $O(n \log^{d-2} n)$ space, the space occupied by the d-dimensional tree holding n points, $S(d,n)$, can be calculated as follows:

$$\begin{aligned}
S(d,n) &= \sum_{i=0}^{\log n} O(2^i) * S(d-1, O(\frac{n}{2^i})) \\
&= \sum_{i=0}^{\log n} O(2^i) * O(\frac{n}{2^i} \log^{d-2} \frac{n}{2^i}) \\
&= O(n) \sum_{i=0}^{\log n} O(\log^{d-2} \frac{n}{2^i}) \\
&= O(n) \sum_{i=0}^{\log n} O(\log^{d-2} n) \\
&= O(n) \log(n) * O(\log^{d-2} n) \\
&= O(n \log^{d-1} n)
\end{aligned}$$

2.2.3 Construction

A range tree can be constructed in optimal time, that is in time linear with respect to its size $O(n \log^{d-1} n)$, provided that the points are already sorted with respect to one coordinate. Again, this can be shown by induction with respect to d.

A 1-dimensional range tree is a binary tree, so it can be constructed in $O(n)$ time provided that the elements are already sorted. Linking the leaves into a linked list does to increase the complexity.

Now let us assume, by inductive assumption, that a $(d-1)$-dimensional range tree can be constructed in time linear with respect to its size. In order to construct a d-dimensional range tree, we need to construct the primary tree which takes $O(n \log n)$ time. We then have to construct the $(d-1)$-dimensional range trees stored in each node. We already know from Section 2.2.2 that the total size of these structures can not exceed $O(n \log^{d-1} n)$ and by inductive assumption their construction time is linear, so the total time of building a tree of two or more dimensions is $O(n \log n + O(n \log^{d-1} n)) = O(n \log^{d-1} n)$.

2.2.4 Orthogonal range search query

1-dimensional range trees Orthogonal range search query in one dimension is simply locating elements within a given range. It can be accomplished using a 1-dimensional range tree by finding one of the endpoints of this range using the binary search tree and then listing all of the elements by traversing the linked list

until the other endpoint is reached. If the range is open-ended on one side, this algorithm still works, provided we start from the closed end of the range. In the case in which the tree is represented as an array, locating elements within a range involves a binary search on the array to locate one endpoint of the range and then listing the consecutive elements in the array until the other endpoint of the range is reached.

d-dimensional range trees Orthogonal range search query in d-dimensions involves reporting all points within the multidimensional rectangle with corners at points $(a_1, \ldots, a_d)$ and $(b_1, \ldots, b_d)$. It is performed as follows: The leaves corresponding to a_1 and b_1, as well as their lowest common ancestor v, are found using binary search. The points in these leaves are reported depending on if they fall in the range or not. Then, the right children of the nodes on the path from the left leaf to v (excluding the leaf and v) and the left children of the nodes on the path from the right leaf to v (again excluding the leaf and v) are considered if they are not part of these paths (see Figure 2.2). All of the leaves descending from these nodes have a first coordinate falling in the desired range. The $(d-1)$-dimensional range trees stored in these nodes are used to recursively report all the points in the range.

Example 2.2.2 Figure 2.2 shows an example of a 2-dimensional range tree containing 16 points and illustrates the process of performing a query on that tree.

Range search query using a d-dimensional range tree containing n points takes $O(\log^d n + \ell)$ time, where ℓ is the number of points in the range. This can be shown by induction with respect to d.

A 1-dimensional range query is reduced to tracing two paths in a minimum-height binary tree which takes $O(\log n)$ time and following ℓ entries in a linked list, which takes $O(\ell)$ time. Overall, the query time is $O(\log n + \ell)$.

A range query on a d-dimensional range tree involves tracing two paths in a minimum-height binary tree ($O(\log n)$ time) and querying $O(\log n)$ $(d-1)$-dimensional range trees. The latter results in reporting ℓ items ($O(\ell)$ time) and a $O(\log n * \log^{d-1} n = \log^d n)$ total search time. The overall query time is $O(\log n + \log^d n + \ell) = O(\log^d n + \ell)$.

For $d \geq 2$, the query time can be improved to $O(\log^{d-1} n + \ell)$ with the use of *fractional cascading* (see next section).

2.2.5 Fractional cascading

Fractional cascading was introduced in [22] as general technique for speeding up various operations in different data structures. It can be applied to data structures which store the same element more than once, such as the range tree. It is based on the idea that instead of searching for the same item more than once, we can link copies of the same item into a list and in effect search for each item only once.

Fractional cascading can be used to improve the query time in 2-dimensional range trees by a factor of $O(\log n)$, which in turn improves the query time by the same factor for any d-dimensional range tree where $d \geq 2$, which follows directly from inductive query time analysis in 2.2.4.

Let us recall that in a 2-dimensional range tree, each node stores a *secondary structure* which is a 1-dimensional range tree. Each of these secondary structures is a list of points ordered by their second coordinate with a balanced binary tree for answering range queries on this list. When a query is performed on the 2-dimensional range tree to return points in the rectangle defined by corners (a_1, a_2) and (b_1, b_2) it follows a logarithmic path in the tree and in each node the range query $a_2 - b_2$ is performed on the secondary structure.

These secondary queries can take $O(\log n)$ time and there are $O(\log n)$ of them, which is why the query time is $O(\log^2 n)$. However, observe that in each node the query range is the same — bounded by a_2 and b_2. Also, each secondary structure holds a subset of the structure stored in the parent node.

Fractional cascading applied to the 2-dimensional range tree augments each secondary structure with pointers. Each element in the secondary structure of node v has two pointers — one to the secondary structure of the left child of v and one to the secondary structure of the right child of v. If v stores the point with second coordinate y, then the pointers both point to elements storing such a y' that y' is the smallest second coordinate stored in the respective structure which is greater or equal to y (see Figure 2.3).

With the fractional cascading pointers, when performing the 2-dimensional range query $(a_1, a_2) - (b_1, b_2)$, it is enough two perform the 1-dimensional range query on the secondary structures only once — in the lowest common ancestor node v (see Section 2.2.4). This can be done either using a binary tree if the last level secondary structures are lists, or a binary search on an array if they are arrays. One can then follow the fractional cascading pointers of elements at the ends of the identified range down the two paths of the query in the main tree. Note that from the query

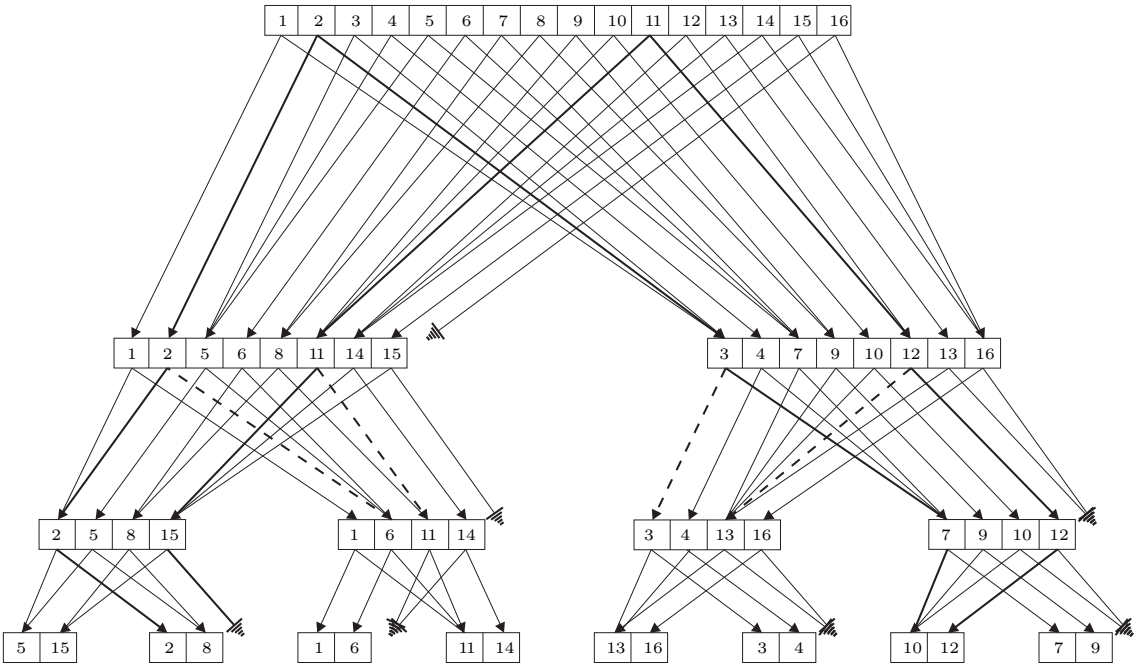

Figure 2.3: The second-level structures of the range tree in Figure 2.2 linked using fractional cascading. For clarity, the first-level nodes are not depicted — they are the same as the ones in Figure 2.2. The bold lines indicate the fractional cascading pointers following the paths of the query in the primary tree. The bold dashed lines indicate the fractional cascading pointers followed from a node on the main query path to its child whose secondary structure needs to be queried.

algorithm described in Section 2.2.4, each node whose secondary structure is queried has a parent located on this path. Therefore, instead of querying these structures, we can use the path of the fractional cascading pointers to obtain the desired range in constant time each time (see Figure 2.3).

Fractional cascading increases the space complexity of the range tree only by a constant factor, because each element of a secondary structure stores two extra pointers. The query time becomes $O(\log n + \log n + \ell) = O(\log n + \ell)$. As mentioned earlier, due to the recursive argument in Section 2.2.4, the query time for d-dimensional range tree with fractional cascading is $O(\log^{d-1} n + \ell)$ for $d \geq 2$.

Example 2.2.3 Figure 2.3 shows the secondary structures from the tree in Example 2.2.2 augmented with fractional cascading pointers. The fractional cascading pointers followed during the query $1 \leq x \leq 13$, $2 \leq y \leq 11$ are indicated.

2.2.6 Dynamic operations on range trees

Range trees can support insertion and deletion operations. In the dynamic case, the primary tree structure is a weight-balanced B-tree (see Section 1.3). Insertion or deletion into the tree involves the insertion or deletion into the main tree and the insertion or deletion into a logarithmic number of secondary structures. As with the previous operations, we can show by induction that this takes $O(\log^d n)$.

Rebalancing operations on a node v require rebuilding the secondary structure associated with v. From Section 2.2.3 we know that this takes time linear with respect to the size of this structure, which in turn is linear with respect to the number of leaves descending from v. Due to the property of the weight-balanced B-tree (see Section 1.3), each rebalancing operation on a node with x descendants happens only every $O(x)$ operations, hence the amortized cost is constant and rebalancing does not increase the amortized complexity of the dynamic operations.

2.2.7 Applications

Range tree structures have numerous applications in many fields. If the elements stored in tree are treated as points, range trees offer an efficient way of performing geometric orthogonal searching. However, the elements can be interpreted as any other kind of data having d scalar attributes, where d is the dimensionality of the tree. For example, the elements can be employee records in a database, for which the salary, position, and seniority is stored. In such a case, range trees can be used

to answer queries of the form "Return all employees holding a managerial position, with a salary between 100000 and 120000, who have been with the company for more than 10 years".

Part II

Making structures rank-sensitive

Chapter 3

Introduction and definitions

Output-sensitive data structures are at the heart of text searching [46], geometric searching [27], database searching [83], and information retrieval in general [8, 90]. They are the result of preprocessing n items (these can be textual data, geometric data, database records, multimedia, or any other kind of data) into $O(n \operatorname{polylog}(n))$ space in such a way, as to allow quickly answering on-line queries in $O(t(n) + \ell)$ time, where $t(n) = o(n)$ is the cost of querying the data structure (typically $t(n) = \operatorname{polylog}(n)$). The term output-sensitive means that the query cost is proportional to ℓ, the number of reported items satisfying the query, assuming that $\ell \leq n$ can be much smaller than n. In literature, a lot of effort has been devoted to minimizing $t(n)$, while the dependency on the *variable* cost ℓ has been considered unavoidable because it depends on the items satisfying the given query and cannot be predicted before querying.

In recent years we have been literally overwhelmed by the electronic data available in fields ranging from information retrieval, through text processing and computational geometry to computational biology. For instance, the number ℓ of items reported by search engines can be so huge as to hinder any reasonable attempt at their post-processing. In other words, n is very large but ℓ is very large too (even if ℓ is much smaller than n). Output-sensitive data structures are too optimistic in a case such as this, and returning all the ℓ items is not the solution to the torrent of information. For instance, the aforementioned search engines return millions of Web pages per query—an amount no human can read and no computer can quickly crawl over for post-processing.

Search engines are just one example; many situations arising in large scale searching share a similar problem. But what if we have some preference regarding the items stored in the output-sensitive data structures? Perhaps we do not just want any

occurrences of a string, the items from a geometric range search, or a group of similar genotype sequences, but we want them returned in some order. This order may have nothing to do with the linear order underlying the internal organization of the output-sensitive data structure. More importantly, we might not need all of the elements, but just the top k best-ranking ones. For example, we might want to see the first 100 occurrences (in left-to-right order) of a string in a text, the lowest 5 points lying between two vertical lines, or the 10 longest genotypes similar to that of the fruit fly.

The solution in this case involves assigning an application-dependent *ranking* to the items, so that the top k best-ranking items among the ℓ ones satisfying an online query can be returned *sorted in rank order*. (We assume that $k \leq \ell$ although the general bound is indeed for $\min\{k, \ell\}$.) Note that the overload is significantly reduced when $k \ll \ell$. For example, PageRank [70] is at the heart of the Google search engine, but many other rankings are available for other types of data. Z-order is useful in graphics, since it is the order in which geometrical objects are displayed on the screen [49]. Records in databases can be returned in the order of their physical location (to minimize disk seek time) or according to a time order (e.g. press news). Positions in biological sequences can be ranked according to their biological function and relevance [46]. These are just basic examples, but more can be found in statistics, geographic information systems, etc.

To address this, we define the concept of $r$ank-sensitive data structures as follows.

Definition 3.0.4 *Throughout this work, we will use the term* r*ank to refer to the given linear order defined on the items stored in a structure, which represents a preference regarding these items. We will identify it with a function* $rank$ *which assigns an integer value form the range* $1 \ldots n$ *to each of the* n *items in the structure in such a way that* $rank(x) < rank(y)$ *if and only if the item* x *is better ranking than the item* y*.*

Definition 3.0.5 *A* r*ank-sensitive data structure is the result of preprocessing* n *items into* $O(n \operatorname{polylog}(n))$ *space in such a way, as to allow the answering of online queries in* $O(t(n) + k)$ *time, where* $t(n) = o(n)$ *is the cost of querying the data structure and the parameter* k*, given at query time, is the number of items reported. The* k *reported items are the top* k *(according to rank) items satisfying the query and the items are reported in rank order. The rank is a given linear order defined on the* n *items stored in the structure.*

Chapter 4

Adding rank-sensitivity to suffix trees

4.1 Introduction

From the architecture of suffix trees (see Chapter 2.1), the result set of a query on a suffix tree corresponds to the set of leaves descending from a particular internal node. Therefore, in order to obtain a rank-sensitive version of the suffix tree, we need a way of augment this structure in such a way as to enable the efficient returning of only the top-ranking subset of leaves descending from any given node.

In this chapter, we present a way to augment any tree data structure, given a rank (a linear order) defined on its leaves, to enable the returning of the top k best-ranking leaves descending from node q, in rank order, for any given k and q, in time linearly proportional to k.

This method can be used to create rank-sensitive suffix trees, but can also be used on other tree index structures, which have the property that the query results correspond to the leaves in a subtree. Any kind of trees which represent hierarchies have this property. In computational biology, hierarchical trees are used to express the similarity level between organisms or proteins or DNA structures, or to express the ancestor-descendant relationship between them. A query on such a structure could be something like "return all species from the genus *Escherichia*". Large hierarchical trees also often result from automatic hierarchical clustering [86].

Other databases also often reflect a hierarchy – for example the organizational hierarchy of the employees of a company. The increasingly popular XML format [18] is intrinsically hierarchical and queries on XML documents may involve returning the leaves descending from a node.

4.2 Preliminaries

The base for our data structure is a tree which we will call *the original tree*. The original tree has a specified root, hence the parent-child relationship between its nodes is well defined (the parent of a node is the next node on the path from this node to the root; the root has no parent; a node is a child of its parent). The nodes of such a tree can be divided into internal nodes (nodes with at least one child) and leaves (nodes without children).

From this point on, we will assume that the original tree is a full binary tree — one whose nodes have either two or no children. Any tree can be transformed into a full binary tree (with no more than twice the nodes) in such a way that every subset of leaves previously defined as being the descendants of some node, is still the set of leaves descending from some node in the full binary version of the tree.

Definition 4.2.1 *The depth of a node is defined as the number of edges on the path from this node to the root.*

Rank is defined on the set of leaves of the tree.

4.2.1 Light and heavy edges

Essential to our approach is a partition of the original tree based on the one utilized in [75]. We classify all tree edges into *light edges* and *heavy edges* in such a way that for each internal node v_p and its children v_{c_1} and v_{c_2} the following properties hold.

1. Among the two edges, $(v_p - v_{c_1})$ and $(v_p - v_{c_2})$, one edge is heavy and the other is light.
2. The edge $(v_p - v_{c_1})$ is heavy if and only if the subtree rooted at v_{c_1} contains no less nodes than the subtree rooted at v_{c_2}.

This definition may not be entirely unambiguous (if two sibling nodes are roots of equal-sized subtrees), but a consistent classification can always be established.

Example 4.2.2 Figure 4.1 shows a tree with edges correctly partitioned into light and heavy ones.

Definition 4.2.3 *A heavy path is a path in the tree composed solely of heavy edges.*

Definition 4.2.4 *The light depth of a node is defined as the number of light edges on the path from this node to the root.*

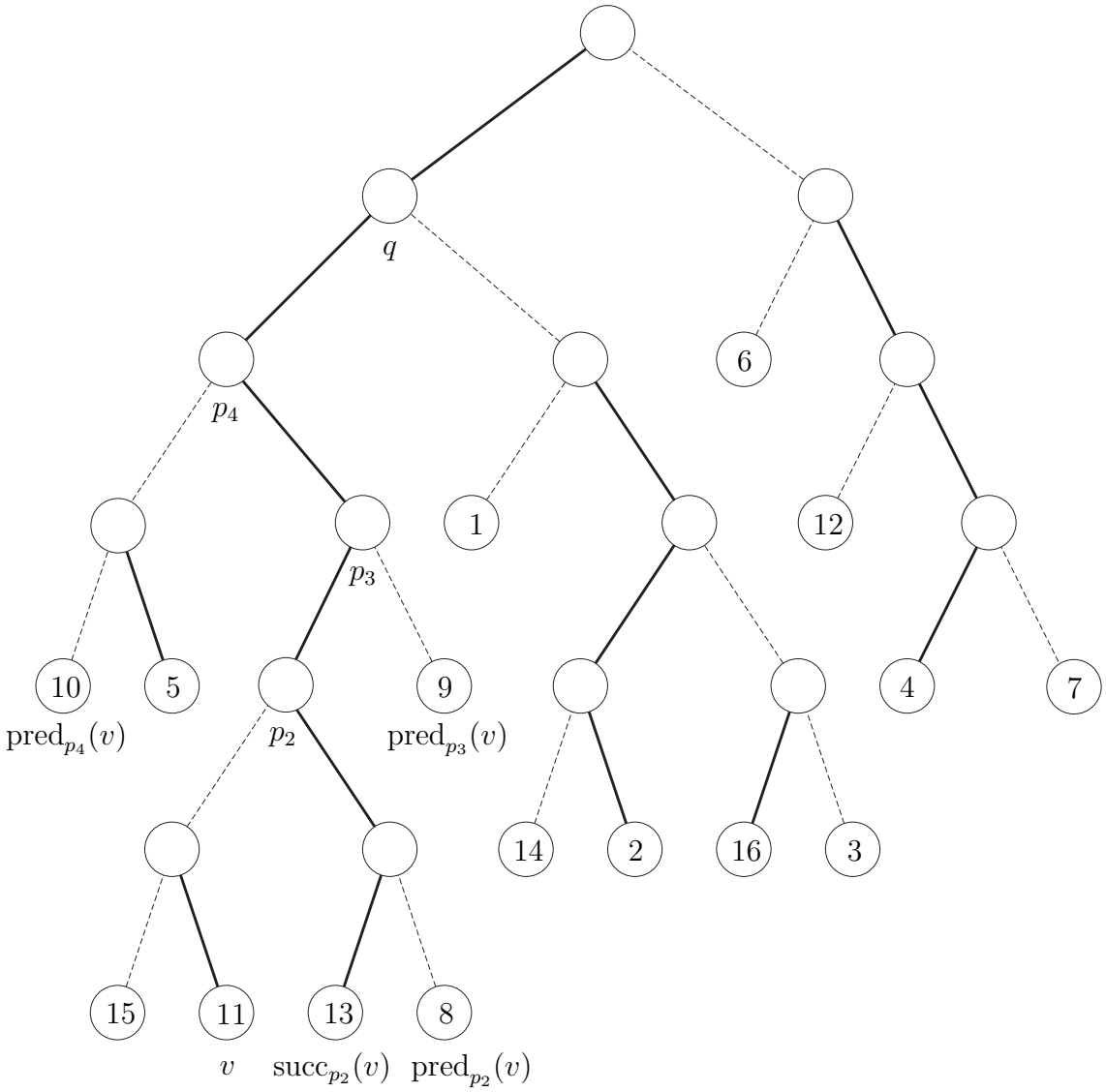

Figure 4.1: A sample tree with the leaf rank indicated inside the leaf symbols. Heavy edges are depicted using solid lines and light edges are depicted using dashed lines. Some nodes have labels below for reference.

4.2.2 Rank-predecessor and rank-successor

Definition 4.2.5 *We use the phrase "rank-predecessor of v at w" ($\mathrm{pred}_w(v)$) to mean the predecessor of the leaf v in rank order in the set of leaves descending from the node w. Analogously, we define "rank-successor of v at w" ($\mathrm{succ}_w(v)$) as the successor of v in rank order in the set of leaves descending from w. If no rank-predecessor or rank-successor exists, we write $\mathrm{pred}_w(v) = \emptyset$ or $\mathrm{succ}_w(v) = \emptyset$.*

Example 4.2.6 See Figure 4.1 for examples of rank-predecessors and of a rank-successor.

4.3 Approach

4.3.1 Naive solution with quadratic space

Let us recall that the problem in question is to quickly list the best-ranking leaves descending from a given node in rank order. The simplest way to achieve this is to preprocess the tree and to store the following information.

1. For each node w — the best-ranking leaf descending from w.

2. For each leaf v and its ancestor w — the rank-successor of v at w.

With the above information accessible in constant time (this can be easily implemented using a set of arrays) a query can be answered in output-sensitive time, by first returning the best-ranking leaf descending from the query node and then for each returned node, looking up its rank-successor at the query node until the desired number of leaves is returned.

Example 4.3.1 A preprocessing of the tree in Figure 4.1 produces a table such as the one in Table 4.1 and stores information about the best-ranking descending leaf for each node. To subsequently list the best-ranking leaves descending from query node q, one can first use the latter information to find out that the leaf of rank 1 should be returned first. Next, one can use the row of the table corresponding to a depth of 1 (since 1 is the depth of query node q) to return each subsequent leaf until the desired number of leaves is returned. The entry for the leaf of rank 1 at depth 1 is the leaf with rank 2 so 2 is returned next. Analogously, leaves of rank 3, 5, 8, 9, 10, 11, 13, 14, 15, 16 are returned next in that order.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	∅
1	2	3	5	6	8	7	12	9	10	11	13	∅	14	15	16	∅
2	2	3	14	7	8	∅	12	9	10	11	13	∅	15	16	∅	∅
3	∅	3	14	7	10	-	∅	9	11	∅	13	∅	15	16	∅	∅
4	-	14	16	∅	∅	-	∅	11	∅	∅	13	-	15	∅	∅	∅
5	-	∅	∅	-	-	-	-	13	-	-	15	-	∅	∅	∅	∅
6	-	-	-	-	-	-	-	∅	-	-	∅	-	∅	-	∅	-

Table 4.1: A table for the quick answering of rank-successor queries computed for the tree in Figure 4.1. A cell in column i and row j identifies the rank-successor of leaf i at the ancestor of i whose depth is j. For clarity, leaves are labeled with their rank.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	2	3	**4**	**5**	**6**	7	**8**	9	10	11	**12**	**13**	14	15	16	∅
1	2	3	**5**	**6**	8	**7**	12	9	10	11	13	∅	**14**	**15**	**16**	∅
2	**2**	3	14	7	**8**	∅	**12**	9	**10**	**11**	13	∅	15	16	∅	∅
3	∅	**3**	**14**	**7**	**10**	-	∅	**9**	**11**	∅	13	∅	15	**16**	∅	∅
4	-	**14**	**16**	∅	∅	-	∅	**11**	∅	∅	**13**	-	**15**	∅	∅	∅
5	-	∅	∅	-	-	-	-	**13**	-	-	**15**	-	∅	∅	∅	∅
6	-	-	-	-	-	-	-	∅	-	-	∅	-	∅	-	∅	-

Table 4.2: Table 4.1 with only the distinct rank-successors of each leaf distinguished.

This naive solution matches our bounds only in the case of a logarithmic-height tree, which is not always the case for suffix trees [31]. In the worst case, the size of the lookup table can be quadratic with respect to the number of leaves in the original tree. The following sections describe how $O(n \log(n))$ space can be achieved in the case of any tree, regardless of its shape.

4.3.2 Naive solution with non-optimal time

The problem with the lookup table described in the previous section is its pessimistic space complexity. Notice, however, that many entries in this table repeat themselves. In particular, the rank-successor of a given leaf often does not change from one ancestor to the next. Hence, let us consider only these rank-successors of a leaf which differ from the rank-successors at the next depth (for an example see Table 4.2). The number of these distinct rank-successors can not exceed $n \log(n)/2$, which will be shown in detail as part of the complexity analysis in Section 4.6.

So, in order to guarantee $O(n \log(n))$ space complexity, we now store only the *distinct* rank-successors as a list for each node (from now on we will use distinct to mean differing from the value corresponding to a larger depth). However, with this change we lose constant-time access to the rank-successor of a leaf at a given height, since retrieving this information now requires traversing the list associated with the leaf and not an array lookup. Using balanced search trees instead of lists could guarantee logarithmic-time access to any sought rank-successors, but no data structure can guarantee the constant-time access which is needed to turn this solution into an output-sensitive one.

The following section shows how to modify this solution to achieve output-sensitive time while maintaining $O(n \log(n))$ space complexity.

4.3.3 Rank-tree solution

This section provides an outline of the solution. For a formal description of the structure and algorithm see sections 4.4 and 4.5.

Solution outline

There are three key tricks we use to transform the solution from the previous section into one with optimal query time. These are enumerated below.

1. Instead of using rank-successor information, we use rank-predecessor information and list the best-ranking occurrences in *reverse* rank order. Thus produced output can be easily reversed back to its correct order with the use of a stack while maintaining optimal time complexity. The reversal of these pointers significantly reduces the number of list elements which need to be considered. We will later show why this is so.

2. The above modification complicates the procedure of finding the first element to return. Let us recall that when answering a query in the correct order one would always start from the best-ranking element and so the starting element could be stored explicitly for each possible query node. Now that the items are returned in reverse order, the first element should be the k-th ranking leaf descending from q and hence it depends not only on the query node q, but also on the number k of items to list. We address this issue by storing the starting element for each possible query node q and for each possible k' which is a power of 2. Storing this information requires $O(n \log(n))$ space, so it does not increase the space complexity of the solution. When answering a query, the number k of elements to return is rounded up to k' — the nearest power of 2 and k' elements are actually considered. Since k' is never more than twice k, this maneuver does not increase the time complexity of the query.

3. The distinct rank-predecessors of each leaf at the different ancestors of this leaf are stored in lists for each leaf. These lists are additionally augmented with pointers which allow constant time access to areas of the list corresponding to different light depths. Since a leaf can have at most a logarithmic light depth, this does not worsen the space complexity. At the same time, it allows us to consider only a part of the list.

Query algorithm

After these modifications, the query algorithm for listing k best-ranking leaves descending from q in rank order can be summarized in the following steps.

- The initial element to list in reverse order is chosen using pre-computed values stored for each query node q and each number k' which is a power of two. This element will be k'-th leaf in rank order descending from q, where k' is the smallest power of two greater than or equal to k.

If k is greater than half the number of leaves descending form the query node, then the worst-ranking descendant is chosen as the starting node.

- Each subsequent leaf to return is found by traversing part of the list of rank-predecessors of the leaf just returned. The part of the list traversed is the part corresponding to the leaf's ancestors which have the same light depth as the query node.

- Since this procedure returns elements in reverse rank order, the elements are pushed on a stack and then popped to produce the actual solution.

Why it works

While it is pretty clear that the space complexity of the structure is $O(n \log(n))$, the time complexity of the algorithm is not at all obvious. There are $2k'$ stack operations, which is of the order of k, since $k' < 2k$, but each procedure of establishing the next leaf to return involves traversing part of a list, which is not a constant operation.

The key part of the analysis is showing that this procedure is indeed of constant time when amortized over all the times it is performed while answering one query. That is to say, even if the length of list to traverse at one point will turn out to be ten elements long, then in ten other cases it will be of length one.

The reason for this phenomenon is that the need for an element of the list to be examined is directly caused by a different leaf descending from the query node. Moreover, this responsible leaf is one of the k' leafs listed during the run of the algorithm and each leaf can be responsible for at most one other. So the total length of list portions to consider is still linearly proportional to k and hence so is the entire query algorithm.

4.4 Ranked tree structure

A ranked tree consists of the *original tree* (a full binary rooted tree with distinguished light and heavy edges), *leaf lists* (LLs), *leaf arrays* (LAs) and *node arrays* (NAs). The latter three data structures are described in detail in the following sections.

The algorithms in Section 4.5 and complexity analysis in Section 4.6 shed light on why these data structures are actually necessary for solving the problem.

4.4.1 Leaf lists

Each leaf v of the original tree has an associated leaf list $\mathrm{LL}(v)$ in the ranked tree. The leaf list of a given leaf contains all of this leaf's distinct rank-predecessor at each node on the path from this leaf to the root.

Formally, the leaf list for a leaf v (assuming v has depth d) may be constructed by performing the following actions.

1. Create a list of pairs

$$\begin{aligned}\langle (d, p_0, \mathrm{pred}_{p_0}(v)),\\ (d-1, p_1, \mathrm{pred}_{p_1}(v)),\\ (d-2, p_2, \mathrm{pred}_{p_2}(v)),\\ \ldots,\\ (0, p_d, \mathrm{pred}_{d}(v))\rangle,\end{aligned}$$

   where $\langle p_0 = v, p_1, p_2, \ldots, p_d = \mathrm{root}\rangle$ is the path from v to the root of the tree.

2. Remove from this list all entries $(d-i, p_i, \mathrm{pred}_{p_i}(v))$, such that $\mathrm{pred}_{p_i}(v) = \mathrm{pred}_{p_{i-1}}(v)$.

Example 4.4.1 The leaf list of leaf v in the example tree in Figure 4.1 would be

$$\begin{aligned}\langle (6, v, \emptyset),\\ (4, p_2, l_8),\\ (3, p_3, l_9),\\ (2, p_4, l_{10}), \rangle\end{aligned}$$

where l_x is the leaf with rank x. Note that there are no entries corresponding to depths 5, 1 and 0, since $\mathrm{pred}_{p_1}(v) = \mathrm{pred}_{p_0}(v) = \emptyset$ and $\mathrm{pred}_{p_6}(v) = \mathrm{pred}_{p_5}(v) = \mathrm{pred}_{p_4}(v) = l_{10}$.

We will use $\mathrm{next}(e)$ to denote the next element after the element e on the list and $\mathrm{depth}(e)$, $\mathrm{node}(e)$, and $\mathrm{leaf}(e)$ to denote the element's first (depth), second (node), and third (leaf) components respectively.

Note that from the point of view of the algorithm, there is no reason to store the second (node) component in the leaf list, because it is never referenced. We introduce it here since it is very useful in the analysis of the correctness and complexity of the system, but an actual implementation would only need to take into account the depth and the leaf components.

4.4.2 Leaf arrays

In addition to the leaf list, each leaf v possesses a leaf array $\mathrm{LA}(v)$. This zero-based array is a set of pointers to the leaf list of this leaf and contains one more element than is the leaf's light depth.

Let us use d_i to denote the depth of the deepest node of light depth i on the path from v to the root. In that case, the i-th element of $\mathrm{LA}(v)$, $\mathrm{LA}(v)[i]$, points to such an entry (j^*, w, u) of $\mathrm{LL}(v)$ that $j^* = \max\{j : j \leq d_i \wedge (\exists l((j, w, u) \in \mathrm{LL}(v)))\}$.

Leaf arrays allow constant access to each of the separate heavy paths to which entries on the leaf list correspond.

Example 4.4.2 Leaf v in Figure 4.1 has a light depth of 1 and hence its leaf array has two items. $\mathrm{LA}(v)[1]$ points to the $(6, v, \emptyset)$ item of $\mathrm{LL}(v)$, since v is the deepest node on the path from v to the root with a light depth of 1. $\mathrm{LA}(v)[0]$ points to $(4, p_2, l_8)$ since p_2 is the deepest node on the path from v to the root with a light depth of 0. If an entry corresponding to p_2 were not present on $\mathrm{LL}(v)$, then $\mathrm{LA}(v)[0]$ would point to the last entry on the list before the place where the p_2 entry would have been.

4.4.3 Node arrays

Each node v of the tree has an associated (zero-based) node array $\mathrm{NA}(v)$. If the subtree rooted in v contains l leaves then $\mathrm{NA}(v)$ contains $\lceil \log_2(l) \rceil + 1$ elements.

For $i < \lceil \log_2(l) \rceil$, the i-th element of $\mathrm{NA}(v)$, $\mathrm{NA}(v)[i]$, is a pointer to the 2^i-th leaf, in rank order, in the set of leaves descending from v. The last element of the array, $\mathrm{NA}(v)[\lceil \log_2(l) \rceil]$ is a pointer to the worst-ranking leaf in the set of leaves descending from v.

Example 4.4.3 Node q in Figure 4.1 is the ancestor of 12 leaves $\{l_1, l_2, l_3, l_5, l_8, l_9, l_{10}, l_{11}, l_{13}, l_{14}, l_{15}, l_{16}\}$, so $\mathrm{NA}(q)$ has $\lceil \log_2(l) \rceil + 1 = 5$ elements: $\mathrm{NA}(q)[0] = l_1$ (l_1 is 2^0-ranking leaf descending from v), analogically $\mathrm{NA}(q)[1] = l_2$, $\mathrm{NA}(q)[2] = l_5$, $\mathrm{NA}(q)[3] = l_{11}$ and $\mathrm{NA}(q)[4] = l_{16}$, because l_{16} is the worst-ranking leaf descending from v.

4.5 Ranked tree algorithms

4.5.1 Top-k query

Input: Node q, positive integer k.

Output: Top k best-ranking leaves, in rank order, descending from q.

1. Let d be the depth of q.
2. Let ld be the light depth of q.
3. Let l be the number of leaves descending from q.
4. If $k > l/2$ then let $k' = l$ and let $v = \mathrm{NA}(q)[\lceil \log_2(l) \rceil]$ else let $k' = 2^{\lceil \log_2(k) \rceil}$ and $v = \mathrm{NA}(q)[\lceil \log_2(k) \rceil]$.
5. Create an empty stack S.
6. Repeat k' times (while $v \neq \emptyset$).
   (a) Push v on the stack S.
   (b) Let $e = \mathrm{LA}(v)[ld]$.
   (c) Repeat while $\mathrm{next}(e) \neq \emptyset$ and $\mathrm{depth}(\mathrm{next}(e)) \geq d$.
      i. Let $e = \mathrm{next}(e)$.
   (d) Let $v = \mathrm{leaf}(e)$.
7. Repeat k times.
   (a) Output $\mathrm{Pop}(S)$.

Lemma 4.5.1 *The first outer loop (Point 6) of the algorithm pushes k' best-ranking leaves descending from q on the stack S in reverse rank order.*

Proof: The fact that the k' rank leaf is pushed first follows directly from the definition of node arrays. It remains to be shown that the leaf pushed on the stack directly after v is $\mathrm{pred}_q(v)$.

Notice that the query node q lies on the path from v to the root, because from the definition of node arrays, v is a node descending from q. The auxiliary pointer e is initialized to $e = \mathrm{LA}(v)[ld]$.

From the definition of leaf lists and arrays, $\mathrm{node}(e)$ is either the deepest node on the path from v to the root with a light depth of ld or it is a node on this path with a light depth greater than ld. Either way, $\mathrm{node}(e)$ can not be less deep than q.

The iterations of the loop in Point 6c can not move $\mathrm{node}(e)$ off of the path from v to the root, because of the way leaf lists are defined. For the same reason, each iteration causes $\mathrm{node}(e)$ to be less deep so the loop must exit eventually.

After exiting, e is such an entry on the leaf list v that depth(e) is the lowest one which does not exceed d (d is the depth of q). This follows from the exit condition of the loop. From the definition of the leaf list, we can conclude that at leaf(e) = pred$_q(v)$ for the value of e on exiting the loop. □

Theorem 4.5.2 *If k is less than the number of leaves descending from q, then the algorithm returns the k best-ranking leaves descending from q in rank order. Otherwise, it returns all of the leaves descending from q in rank order.*

Proof: This follows almost entirely from Lemma 4.5.1. We must only note that $k' \geq \min(k, l)$ (l is the number of leaves descending from q) and that the pushing and subsequent popping of elements on a stack reverses their order. □

Example 4.5.3 Let us consider the example in Figure 4.1. Suppose we want to list the five best-ranking leaves descending from node q ($k = 5$). The initialization phase sets $d = 1$, $ld = 0$, $l = 12$, $k' = 2^{\lceil \log_2(5) \rceil} = 2^3 = 8$ and $v = \text{NA}(q)[\lceil \log_2(5) \rceil] = \text{NA}(q)[3] = v$.

The leaf v is pushed on the stack and LL(v) is considered. As we recall from Sections 4.4.1, this list has the form $\langle (6, v, \emptyset), (4, p_2, l_8), (3, p_3, l_9), (2, p_4, l_{10}) \rangle$.

The light depth of q (ld) is 0 and we know from Section 4.4.2 that LA(v)[0] points to the $(4, p_2, l_8)$ entry of LL(v), so this is the element e is initialized to. The depth of the next element on the list, $(3, p_3, l_9)$, is still greater than d ($3 > 1$), so we set e to point to this next element.

Similarly, we execute another turn of the loop and set $e = (2, p_4, l_{10})$. This is where the process ends, because there are no more elements on the list. At this point, we know that $l_{10} = \text{pred}_q(v)$. If that were not the case, there would be another entry on the list with a depth greater than 2 and smaller or equal to 1 to indicate that the predecessor of v changes at that point on the path.

In the subsequent steps l_{10}, l_9, l_8, l_5, l_3, l_2, and l_1 are pushed on S. Popping $k = 5$ elements returns l_1, l_2, l_3, l_5, l_8, which is the correct answer.

4.6 Complexity

4.6.1 Space complexity

Leaf lists

Each leaf list LL(v) has at least one initial element (depth(v), v, $\emptyset$). If n is the number of leaves in the original tree, then this accounts for exactly n leaf list elements.

Now let us calculate the maximum number of leaf list elements which are not the first elements on the list. These elements will have the form $(\mathrm{depth}(w), w, \mathrm{pred}_w(v))$, where w is an internal node of the tree.

Let l be the son of w such that $(w-l)$ is a light edge and let h be the son of w such that $(w-h)$ is heavy. If the entry $(\mathrm{depth}(w), w, \mathrm{pred}_w(v))$ exists in $\mathrm{LL}(v)$, then either v descends from l and $\mathrm{pred}_w(v)$ descends from h or the other way around — v descends from h and $\mathrm{pred}_w(v)$ descends from l.

If both v and $\mathrm{pred}_w(v)$ descended from the same son of w (let us call this son s), then $\mathrm{pred}_w(v)$ would be equal to $\mathrm{pred}_s(v)$. Since $\mathrm{depth}(s) = \mathrm{depth}(w) + 1$, $\mathrm{LL}(v)$ would not contain the entry $(\mathrm{depth}(w), w, \mathrm{pred}_w(v))$.

From this we can conclude that for each internal node w, there are at most twice as many leaf list entries $(\mathrm{depth}(w), w, \mathrm{pred}_w(v))$ as there are leaves descending from l (twice, because each leaf descending from l can either play the role of v or that of $\mathrm{pred}_w(v)$. This leads to the following recurrence equation for the maximum total length $L(n)$ of leaf lists in a binary tree of n leaves (if m is the number of leaves descending from a son of the root of the tree, then $\min(m, n-m)$ is the number of leaves descending from the son at the light edge).

$$\begin{aligned} L(1) &= 1 \\ L(n) &= \max_{0<m<n} \Big(L(m) + L(n-m) + 2\min(m, n-m) \Big) \end{aligned}$$

Since the maximum always occurs when the subtrees are of equal size, the second equation can be written as $L(n) = L(n/2) + L(n/2) + 2n/2 = 2L(n/2) + n$. It is easy to verify that the solution to this equation is $L(n) = n + n\log_2(n)$, so $L(n) = O(n\log(n))$.

Leaf arrays

The size of the leaf array in each leaf is proportional to its light depth. Since a light depth cannot exceed $\log n$, the total sum of these is again $O(n \log n)$.

Node arrays

The size of the node array of a node is logarithmic with respect to the number of leaves descending from that node, so it is at most $\log n$ and hence the total sum of these arrays is $O(n \log n)$.

4.6.2 Time complexity

Lemma 4.6.1 *The number of executions of the first outer loop (Point 6) of the algorithm (k') is less than twice the number of items returned ($k' < 2k$).*

Proof: In the case that $k > l/2$ and $k' = l$, we have $k > k'/2$, so $k' < 2k$. Otherwise $k' = 2^{\lceil \log_2(k) \rceil}$, so $k' < 2^{log_2(k)+1}$, and so again $k' < 2k$. □

Lemma 4.6.2 *The total number of executions of the internal loop (Point 6c) in one run of the entire algorithm does not exceed k'.*

Proof: The outer loop (Point 6) of the algorithm is executed k' times, and in effect, k' leaves are pushed on the stack S. Let us associate one credit with each leaf pushed on the stack. Each of these credits will be used for paying for an execution of the internal loop (Point 6c) which took place before the leaf was pushed on the stack. We will show that all executions are paid for, and hence there can not be more than k' of them.

Let us use $\text{proj}(v, q)$ to denote the deepest node belonging to the intersection of the path from v to the root and the heavy path containing q (proj stands for the projection of v on the heavy path of q). Since v descends from q, such a node must exist. Moreover, note that $\text{proj}(v, q)$ is uniquely defined for every v and q. When a leaf v is pushed on the stack, its associated credit will be used to pay for an execution of the internal loop which occurred when the leaf $\text{succ}_{\text{proj}(v,q)}(v)$ was on top of the stack.

On the other hand, each time the internal loop is executed with some leaf v' on top of the stack, e assumes a new value, which is an entry in $\text{LL}(v')$ and hence is of the form $(\text{depth}(p_i), p_i, \text{pred}_{p_i}(v'))$. From the boundary conditions of the loop, we know that $\text{depth}(p_i) \geq \text{depth}(q)$. Moreover, e is initialized to equal $e = \text{LA}(v')[\text{lightdepth}(q)]$, so from the definition of leaf arrays, we know that each subsequent leaf list entry after the initial one has a light depth not greater than that of q. Therefore, each of the leaf list entries considered (except possibly the initial one), $(\text{depth}(p_i), p_i, \text{pred}_{p_i}(v'))$, corresponds to a node p_i which is on the heavy path containing q. Moreover, the leaf $\text{pred}_{p_i}(v')$ can not descend from the heavy son of p_i or the entry would not be on the leaf list. So, $p_i = \text{proj}(\text{pred}_{p_i}(v'), q)$. The following observations can be made.

1. If v is on top of the stack, then $\text{pred}_{p_i}(v')$ will eventually be pushed on the stack as well. This follows from the fact that $\text{pred}_{p_i}(v')$ has higher rank than v

(it is its rank-predecessor), that it descends from q (it descends from p_i which descends from q) and the algorithm lists best-ranking leaves descending from q in rank order (Lemma 4.5.1).

2. Once pushed on the stack, $\mathrm{pred}_{p_i}(v')$ will use its credit to pay for the loop execution in question. That is because:

$$\begin{aligned} \mathrm{succ}_{\mathrm{proj}(\mathrm{pred}_{p_i}(v'),q)}(\mathrm{pred}_{p_i}(v')) &= \\ \mathrm{succ}_{p_i}(\mathrm{pred}_{p_i}(v')) &= v'. \end{aligned}$$

It follows from these observations that each execution of the loop with v' on top of the stack which ends with $e = (\mathrm{depth}(p_i), p_i, \mathrm{pred}_{p_i}(v'))$ is paid for by the credit associated with the leaf $\mathrm{pred}_{p_i}(v')$, which will eventually be listed. Therefore, there are at most k' executions of the internal loop (Point 6c). □

Example 4.6.3 Again, let us consider leaf v from Figure 4.1. After v is pushed on the stack, e traverses $\mathrm{LL}(v)$. Initially $e = (4, p_2, l_8)$. After the first execution of the internal loop (Point 6c), e is moved along the list and becomes $(3, p_3, \mathrm{pred}_{p_3}(v) = l_9)$. According to the amortization argument, this loop execution should be paid using the credit associated with l_9.

Indeed, it is clear that l_9 will be pushed on the stack after $v = l_{11}$, since it has a better rank and also descends from q. At the same time $p_3 = \mathrm{proj}(l_9, q)$ and so $v = \mathrm{succ}_{\mathrm{proj}(l_9,q)}(l_9)$. The next loop execution leaves $(2, p_4, \mathrm{pred}_{p_4}(v) = l_{10})$ and hence is paid by the credit associated with l_{10}. Again we can check that l_{10} will be pushed on the stack and that $v = \mathrm{succ}_{\mathrm{proj}(l_{10},q)}(l_{10})$.

Theorem 4.6.4 *The algorithm is output-sensitive, i.e. its running time is $O(k)$.*

Proof: The first external loop is executed $k' < 2k$ times (Lemma 4.6.1) and the second external loop is executed k times (by definition). At the same the internal loop is executed no more than k' times in total. All individual instructions within the loops require constant time under the assumption that all pointers can be stored in a machine word. This yields an overall time complexity proportional to $c_1k' + c_2k + c_3k' < 2c_1k + c_2k + 2c_3k = O(k)$ (c_i are constants).

□

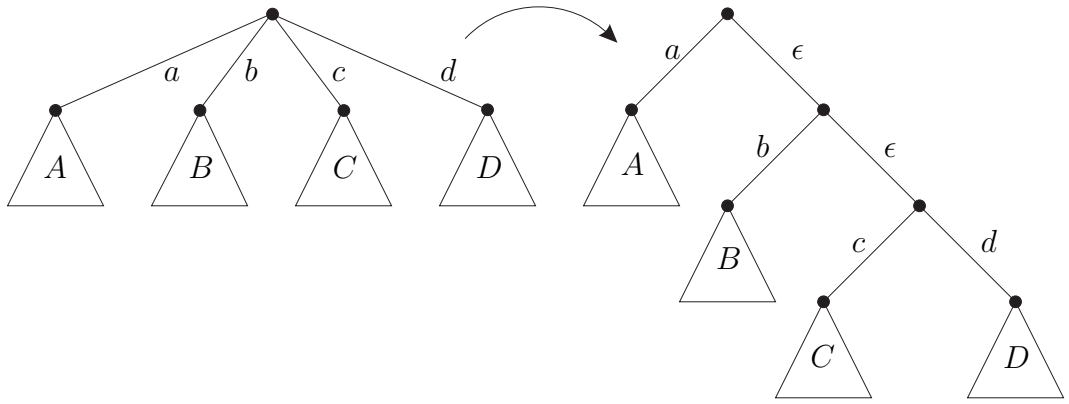

Figure 4.2: Transforming a four degree node into a binary subgraph. Edge labels are indicated with letters a, b, c, d. A special label ϵ is used on the new edges.

4.7 Experimental results

We implemented this algorithm in C++ and compiled using Microsoft Visual C++ 2008 Express Edition with the full optimization (`/Ox`) option. All runs were performed on a 3GHz Core 2 Duo CPU with 4GB RAM running a 64-bit Windows Vista operating system (although the compilation was 32-bit). The query times were measured using the system performance counter (the `QueryPerformanceCounter` function). Function recursion is not used, instead we rely on the stack implementation in the Standard Template Library [76].

As input to the algorithm we used suffix trees created from a text file using an implementation by Zhao [92] which uses the Ukkonen algorithm [80]. The suffix trees, originally of higher degree, were transformed into binary trees by replacing a node with d children with a path of $d-1$ nodes (for $d > 2$) and labeling the edges on the new path with special empty labels (see Figure 4.2). This does not have any significant impact on the suffix tree query algorithm and is only an implementation detail, which is necessary for conforming with the assumption that the original tree is binary.

For the rank function we used the position of the substring in the text, so that a substring occurring earlier in the text would be ranked higher than a substring occurring later.

In order to investigate the behavior of the algorithm given different suffix tree shapes occurring in real life, we tested the algorithm on three types suffix trees built from three types of text files:

- English language text. The text file used was an English language version of Leo Tolstoy's novel "War and Peace" obtained from the Project Gutenberg website (`http://www.gutenberg.org`).
- DNA sequence. The file used was in FASTA [71] format and contained a fragment of the *drosophila melanogaster* genome. It was obtained from the website of the Berkeley Drosophila Genome Project (`http://www.fruitfly.org`).
- Random file. The file used contained a decimal representation of the constant π with a precision of 3000000 decimal digits. It was generated with Wolfram Mathematica 6. Such a file can be considered a uniformly random stream over a 10-digit alphabet as far as the shape of the suffix tree is concerned.

When a dependency on tree size was being considered, smaller files were obtained by taking a prefix of the original file.

The goal of the experiments was to

- Assert the correctness of the solution.
- Provide a real-life usage scenario.
- Investigate the query time as a function of
  - the parameter k,
  - the total number of items satisfying the query,
  - the size of the indexed tree,
  - the shape of the indexed tree.
- Investigate the structure size as a function of
  - the size of the indexed tree,
  - the shape of the indexed tree.

We did not compare the algorithm with existing algorithms offering the same functionality, because we are not aware of any such algorithms. Instead we compared this approach with the what can be achieved using just the original suffix tree.

We found that we can successfully index files of up to approximately 3Mb in length using our algorithm. A memory-optimized implementation would allow for larger file sizes, but our focus was on asymptotic performance and implementation

clarity. Setting up the index structure does consume extra time and memory with respect to using just a simple suffix tree, but it does not create any unmanageable overhead. After the augmented suffix tree structure is set up, top-k queries drastically outperform traditional suffix tree queries in the case in which k is much smaller than the total number of occurrences. In fact, top-k queries show a linear dependency on the parameter k providing empirical proof that the algorithm is indeed rank-sensitive.

The algorithm illustrates how the rank tree structure can be used in the case in which top-k query performance needs to be fast and cannot depend on the total number of items satisfying the query — for example, in the case of real-time applications.

The next section provides some insight on the structure construction algorithm used and the following sections summarize the experiment results.

4.7.1 Structure construction algorithm

The algorithm used to construct the structure can be outlined as follows.

1. Augment the tree with subtree sizes stored in each node
2. Partition edges into light and heavy
3. Augment the tree with depth and light depth stored in each node
4. Set up leaf lists and node arrays in a bottom-up fashion
5. Set up leaf arrays

The first three steps are simple, as each requires a single linear traversal of the tree top-down or bottom.

The only non-trivial part of the construction algorithm is Step 4. We realize it by constructing the structures in a bottom-up fashion. With each node v we associate a temporary sorted (according to rank) array TA of leaves descending from this node. We cannot, however, keep this array for all of the nodes at the same time, or else the total memory requirement would become quadratic, severely reducing the usefulness of the algorithm. Instead, we use the arrays of the child nodes to construct the array of the parent node and to update the appropriate leaf list and node array entries and then we remove the temporary arrays of the child nodes from

memory — that way the total length of the temporary arrays is always equal to the number of leaves, hence is linear with respect to the size of the tree.

Step 4 of the construction can be outlined as follows.

1. For each leaf v:
   (a) Set $\mathrm{NA}(v) = \{v\}$. Node arrays of leaves have just one element.
   (b) Set $\mathrm{TA}(v) = \{v\}$. Temporary arrays of leaves also have just one element.
   (c) Set $\mathrm{LL}(v) = (\mathrm{depth}(v), \emptyset)$. The theoretical description of the algorithm would refer to this entry as $(\mathrm{depth}(v), v, \emptyset)$, but as mentioned earlier, we do not actually have to store the middle component of the triplet. This is just the first the entry of $\mathrm{LL}(v)$, more entries will be added during the next steps of the construction.
2. For each node v with children c_1 and c_2, such that $\mathrm{TA}(c_1)$ and $\mathrm{TA}(c_2)$ are computed, but $\mathrm{TA}(v)$ is not:
   (a) Construct $\mathrm{TA}(v)$ by merging arrays $\mathrm{TA}(c_1)$ and $\mathrm{TA}(c_2)$ while preserving the rank order on the leaf entires.
   (b) For every entry l_1 coming from the child c_1 preceded in $\mathrm{TA}(v)$ by an entry l_2 coming from the child c_2, add the leaf list entry $(\mathrm{depth}(v), l_2)$ at the end of $\mathrm{LL}(l_1)$.
   (c) For every entry l_2 coming from the child c_2 preceded in $\mathrm{TA}(v)$ by an entry l_1 coming from the child c_1, add the leaf list entry $(\mathrm{depth}(v), l_1)$ at the end of $\mathrm{LL}(l_2)$.
   (d) Set up $\mathrm{NA}(v)$ by taking the entries from $\mathrm{TA}(v)$ which are powers of 2 and the last entry of $\mathrm{TA}(v)$ in the case in which the number of leaves descending from v (equal to the size of $\mathrm{TA}(v)$) is not a power of 2.
   (e) Free memory occupied by $\mathrm{TA}(c_1)$ and $\mathrm{TA}(c_2)$.

Step 5 of the construction is again simple and can be realized by traversing the path to the root from each leaf.

4.7.2 Query time results

In order to investigate the top-k query time under different circumstances, we ran the query for different tree shapes, sizes, search substrings, and values of k. The varying

tree shapes were obtained by building suffix trees over the random text, natural language text, and DNA sequence. Varying tree sizes were obtained by taking different prefixes of the files. The search substrings used depended on the tree shape used. For the random decimal file they were strings of consecutive 0's of varying lengths. (Since the file is random, any substring of a given length appears roughly the same number of times.) For the DNA sequence they were strings of consecutive A's of varying lengths. For the natural text, the following strings were consider in order to obtain varying frequencies of occurrence: "e", "n", "d", "he", "the", "of", "you", "here", "they", "head", "house", "night", "heaven", "complacently".

All queries were performed 100 times in a row and the average time was measured. Only the reporting phase (and not the search) was taken into account. Additionally, the whole experiment was performed four times and a minimum query time between the four runs was taken into account in each case in order to minimize the impact of any external disturbances on the query time.

For comparison, we also measured the time to extract all of the occurrences using a standard subtree traversal.

In order to take a representative sample of the results without sampling all possible file lengths and values of k, we ran the queries for file sizes and values of k which were powers of 2.

To ensure that Lemma 4.6.2 holds and that the algorithm is indeed output-sensitive, we also counted the number of the executions of the internal loop (Point 6c) of the query algorithm.

Internal loop executions

In accordance with Lemma 4.6.2, the number of executions of the inner loop of the query algorithm never exceed $2k$ in any of the runs (the largest ratio was found in the language text ≈ 1.9 for 61 executions and $k = 32$). However, it is interesting to observe that the way this ratio varies between 0 and $2k$ distinctly depends on the size as well as shape of the tree (although not on k itself), which has an impact on the overall query time for the different structures.

For both the natural language and random text, the ratio stays in the 1.4–1.6 range growing slightly for larger texts (see Table 4.3). In the case of the DNA sequence, however, the ratio quickly drops to below 1. This suggests that the DNA sequence tree shape is correlated with our measure of rank, so the algorithm has to perform less work. Indeed the DNA sequence contains long sequences of the A

File size	DNA	Random	Language
128	1.54	1.53	1.46
256	1.51	1.55	1.40
512	1.12	1.43	1.42
1024	1.03	1.39	1.45
2048	1.10	1.52	1.45
4096	1.13	1.51	1.44
8192	0.82	1.54	1.49
16384	0.83	1.55	1.51
32768	0.87	1.53	1.54
65536	0.91	1.56	1.56
131072	0.96	1.56	1.56
262144	0.94	1.57	1.57
524288	0.94	1.58	1.58
1048576	0.94	1.59	1.58
2097152	0.95	1.59	1.59

Table 4.3: The average ratio of the number of executions of the inner loop to k depending on the suffix tree size and shape.

character and we were searching for substrings of the form A^i. The occurrences of these substrings would be subsequent positions within the long A chains and would also correspond to subsequent (according to inorder) leaves of the tree. Therefore the rank (for which we use the position in the text) of these occurrences is highly correlated with the inorder position of the corresponding leaves in the tree.

Our algorithm performed faster on a tree exhibiting this correlation much like many sorting algorithms perform faster on a sequence which is already partially sorted.

Query time depending on k

As expected, the overall reporting query time depends mainly on k in a linear way. It also depends on the size and shape of the tree due to the phenomenon discussed in the preceding section, so for the same value of k we get slightly different query time. However, the query time per each element returned across the entire result set fits in a linear range and is limited by 0.000255ms per each item returned. See Table 4.4 and Figure 4.3.

k	Query time in ms		
1	0.0015	–	0.0017
2	0.0015	–	0.0017
4	0.0016	–	0.0017
8	0.0016	–	0.0018
16	0.0019	–	0.0021
32	0.0026	–	0.0035
64	0.0037	–	0.0055
128	0.0065	–	0.0101
256	0.0121	–	0.0225
512	0.0240	–	0.0400
1024	0.0460	–	0.0738
2048	0.0910	–	0.1409
4096	0.1860	–	0.2724
8192	0.4238	–	0.5968
16384	1.2012	–	3.094
32768	5.7078	–	8.1701
65536	13.2288	–	16.7001
131072	28.5475	–	33.2209
262144	59.5983	–	66.5709

Table 4.4: The query time in milliseconds depending on k.

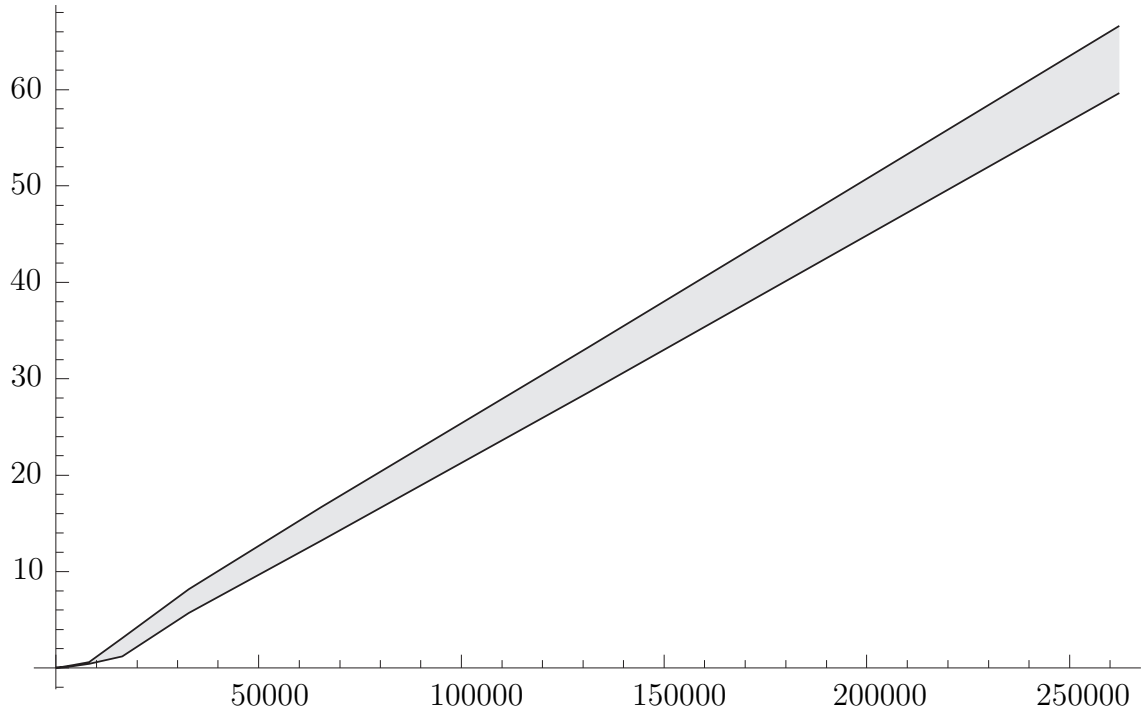

Figure 4.3: The query time in milliseconds depending on k.

Top-k query time as compared to returning all unsorted results

As we mentioned earlier, we also measured the time to report all results using the standard suffix tree method, that is by traversing the relevant subtree (we used the stack implementation in the Standard Template Library [76] to recurse through the subtree). We compare this to running our query with $k = \ell$, where ℓ is the total number of elements matching the query (the number of occurrences of the substring).

Note that our top-k query has the advantage over the standard query in that the results returned are sorted by rank, while the results returned by the standard query are not sorted.

We found that for $\ell < 10000$ our query has very similar running time to the standard query and even often outperforms it for small values of ℓ (see Figure 4.4). It is only for very large values of ℓ (over 100000) that we saw an up to six-fold advantage of the standard query over our top-k query returning all of the results sorted (see Figure 4.5). The relative speed of the top-k query with respect to the standard one did not show a dependence on the size of the tree.

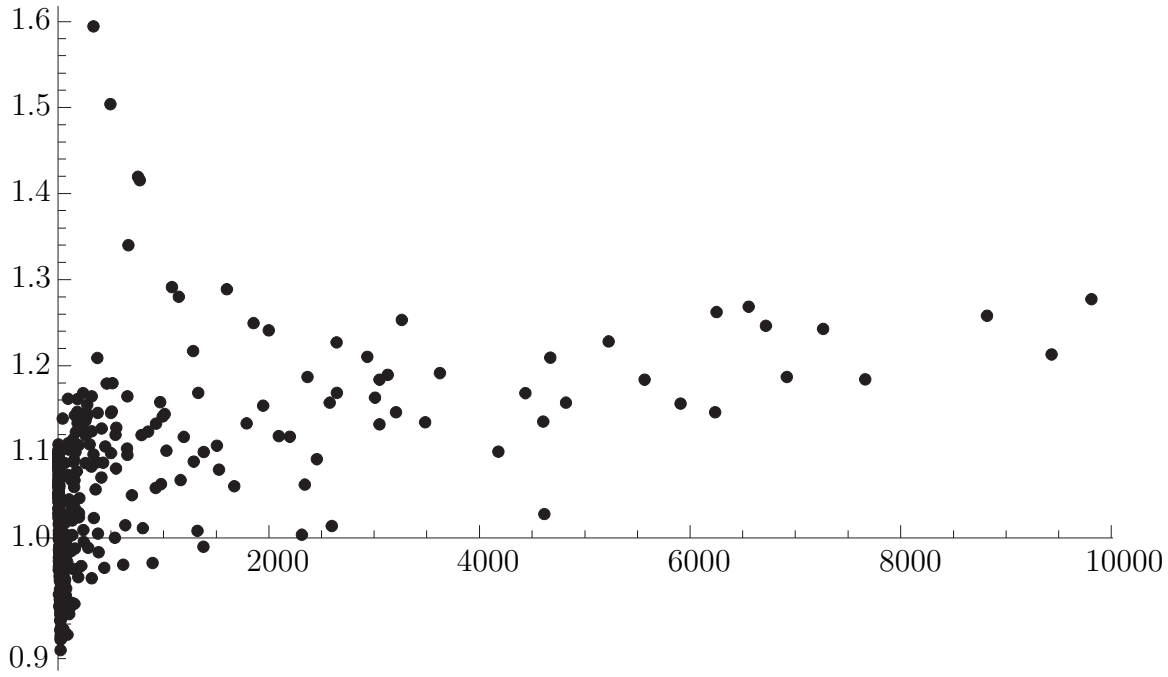

Figure 4.4: Ratio of top-k (sorted) query time to standard suffix tree (unsorted) query time for $k = \ell$, where ℓ (the total number of occurrences) is less than 10000.

Top-k query time as compared to returning all results and sorting them

We also compared results obtained using our structure to obtaining the same result by traversing the suffix tree in the usual way and then sorting the results using the Standard Template Library `list::sort` function. That is, in both cases we output the results of the query into a Standard Template Library `list` structure (implemented as a linked list). Using our structure it was enough to create the list from end to beginning (no change in the complexity) and the result was a sorted list. Using the standard tree traversal, we output the results into the list, but then had to additionally invoke `list::sort` the obtain the rank-sorted list which results from using our structure. In all cases, we ran the tests for $k = \ell$.

The result is that our structure consistently outperforms the standard method in returning rank-sorted results (see Figure 4.6). For very small ℓ there is really no difference between the two and measurement fluctuations prevail in the results, but for ℓ in the hundreds or thousands, our structure is approximately four times faster. For larger ℓ the relative performance of our structure is slightly worse, but still better than the alternative.

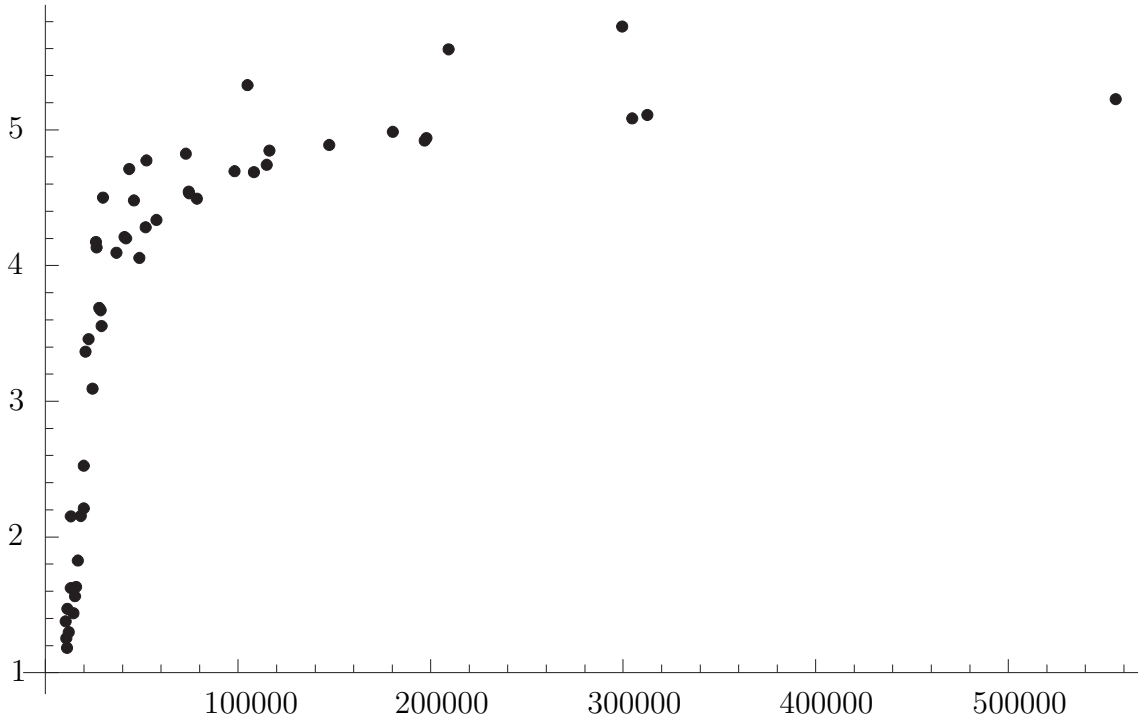

Figure 4.5: Ratio of top-k (sorted) query time to standard suffix tree (unsorted) query time for $k = \ell$, where ℓ (the total number of occurrences) is more than 10000.

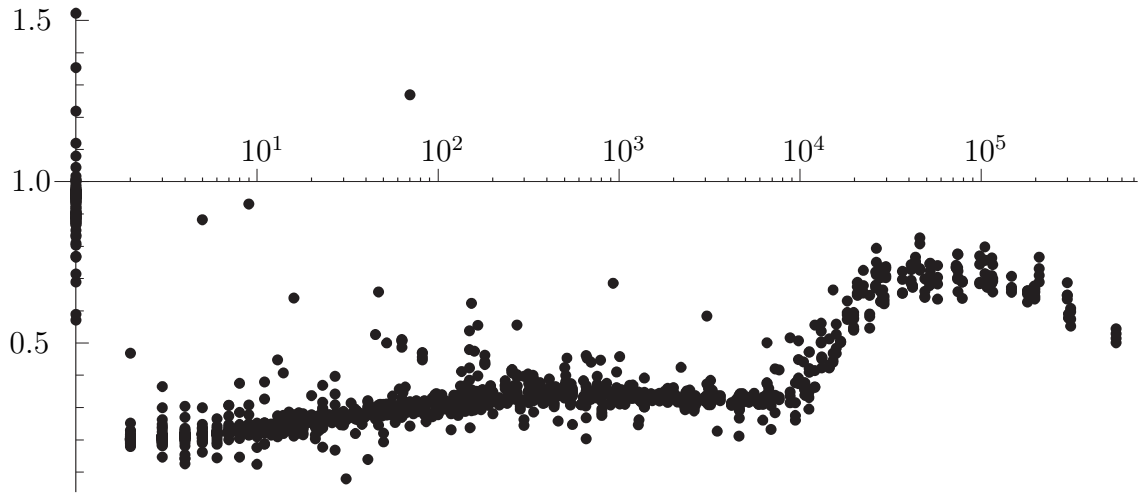

Figure 4.6: Ratio of query time using our structure to obtaining the same result by retrieving all of the items from the suffix tree and sorting them using the Standard Template Library `list::sort` function. Results shown for $k = \ell$ on a logarithmic scale of ℓ for clarity.

	DNA				Random				Language			
Text length	NA	LL	LA	Time	NA	LL	LA	Time	NA	LL	LA	Time
4	12	7	7	0.10	12	7	7	0.10	12	7	7	0.10
8	32	15	15	0.09	30	17	16	0.09	32	15	15	0.09
16	73	35	34	0.11	68	39	36	0.10	73	37	34	0.11
32	157	81	74	0.14	139	99	85	0.14	155	85	75	0.14
64	321	190	162	0.22	288	227	187	0.22	320	187	159	0.23
128	652	464	376	0.39	598	499	405	0.36	625	441	356	0.37
256	1307	1184	864	0.73	1168	1130	897	0.70	1231	1019	797	0.69
512	2463	2417	2127	1.38	2339	2443	1920	1.30	2396	2288	1748	1.32
1024	4792	5319	4529	2.77	4731	5286	4125	2.65	4774	4892	3718	2.49
2048	9365	11860	9833	5.43	9449	11451	8794	5.17	9430	10067	7725	4.97
4096	18477	25659	21935	10.94	18773	24678	18733	10.17	18939	22457	16908	10.25
8192	36663	55592	47094	22.56	37727	52645	39529	21.04	38000	49029	36523	21.49
16384	72963	118791	99154	48.49	75613	112458	83554	45.99	75591	105918	78367	45.95
32768	145564	254230	211109	105.54	150122	238861	176000	98.74	151125	226539	167261	99.56
65536	291091	539672	446908	228.09	301337	503955	369413	210.52	300935	478358	353916	223.66
131072	582004	1137054	942122	479.79	605114	1063767	774289	457.03	600452	1007260	747749	513.68
262144	1180491	2358979	1961077	1055.78	1202964	2243983	1620448	1021.32	1199255	2119975	1577009	1106.11
524288	2592192	4724605	3997045	2369.75	2404242	4707727	3383228	2118.42	2399670	4421080	3304868	2287.09
1048576	5040222	10087227	8442637	5032.81	4838610	9851230	7048631	4483.80	4797763	9267206	6923060	4878.23
2097152	11764144	20066454	16841888	18555.66	9653744	20625087	14666161	9638.04	9596879	19376074	14469231	10434.83

Table 4.5: Structure size and construction time (in milliseconds) depending on the length and type of the indexed text.

4.7.3 Structure size results

We counted the number of leaf list, node array, and leaf array entries which combined constitute our augmented structure. We also measured the time to create the structure. As with the query time, we ran four independent test sets and took the minimum construction time of the four in order to minimize the impact of any external disturbances.

The results are summarized in table 4.5.

We found that for the random and natural language texts, the total node array size, although theoretically bounded by $O(n \log(n))$ is in practice linear with respect to the size of the tree. For the unusual shape of the genetic sequence tree it does, however, grow slightly faster than the tree itself. The total size of the other two structures — leaf lists and leaf arrays — approached in practice the theoretical upper bound of $O(n \log(n))$ for larger tree sizes.

The structure construction time, although theoretically pessimistically quadratic with respect to the size of the tree, in practice turned out mainly proportional to the size of the structure, hence of the order of $O(n \log(n))$. Only for very large trees, did the quadratic component start showing slightly and was evident mainly in the case of the highly irregular genetic sequence tree. Constructing the structure for a tree of 4194303 nodes took roughly 10 seconds for the random and natural language texts, 18 seconds for the genetic sequence.

4.8 An alternative approach using Cartesian trees

4.8.1 Data structure

Another approach to solving the same problem uses *Cartesian trees* [85] in addition to the light and heavy edge partition [75].

Let us define an *index node* as a node which is not connected to its parent with a heavy edge. In particular, the root of the tree is an index node. Note that from the properties of the light and heavy partition [75], a leaf has at most a logarithmic number of index node ancestors.

We associate a Cartesian tree with each index node. The Cartesian tree $C(w)$ for the index node w is constructed from a list of all leaves descending from w. For each such descending leaf v, we note its rank ($rank(v)$) and the distance from w to the first light edge on the path from w to v ($depth_w(v)$).

We then use the pairs $(rank(v), depth_w(v))$ to construct the Cartesian tree for w just like described in [85], except that we use a reversed order on the second coordinate, so that the elements with the largest (and not the smallest) depth end up on top of the Cartesian tree (see Figure 4.7).

Each node v of the Cartesian tree is augmented with a pointer to its nearest ancestor to the left of that node $L(v)$.

Each node u of the tree stores a reference to its nearest index node ancestor (w) and the distance to it $(depth_w(u))$. It also stores a *node array* $(NA(u)[i])$ logarithmic in size with respect to the number of leaves descending from u. The value $NA(u)[i]$ is a reference to a node the Cartesian tree $C(w)$ corresponding to the descendant of u which is number 2^i in rank[1] among all the leaves descending from u.

4.8.2 Query algorithm

Given query node q and a value k, we first locate the nearest indexed node ancestor of q (w) the distance to it $(depth_w(q))$, and the node in $C(w)$ identified by the node array entry $NA(u)[i] = v$, such that $2^i \geq k$. We consider the sequence of nodes produced by following the auxiliary pointers L starting from v, that is $(v, L(v), L(L(v)), \ldots, L^s(v))$, such that $L^s(v)$ does not have an ancestor to the left of it. We then reverse this list to produce $l = (L^s(v), L^{s-1}(v), \ldots, L(v), v)$. We then follow the procedure:

1. For each element t in l
   (a) Traverse the left subtree of t in infix order, but consider only values[2] whose second coordinate is greater or equal to $depth_w(q)$. Return the items visited (their corresponding leaves).
   (b) Return the leaf corresponding to t.

Example 4.8.1 Figures 4.8 and 4.9 show two examples of queries, for query node q_1 and $k = 4$ and for query node q_2 and $k = 8$. In both cases, w is identified as the index node and the distances of the query node to it are noted ($depth_w(q_1) = 2$ and $depth_w(q_2) = 1$). Next the node arrays are accessed to produce the lists. In case of $(q_1, 4)$ the list is $l = ((8, 5), (9, 3), (10, 2))$ and in case of $(q_2, 8)$ it is $l = ((8, 5), (11, 4))$

[1] Actually, the last value points to the lowest ranking element, like in the version of the structure described in the previous section, but this is just an implementation detail.

[2] We can do that, because the tree is a heap with respect to the second coordinate, so we are just considering some top part of it.

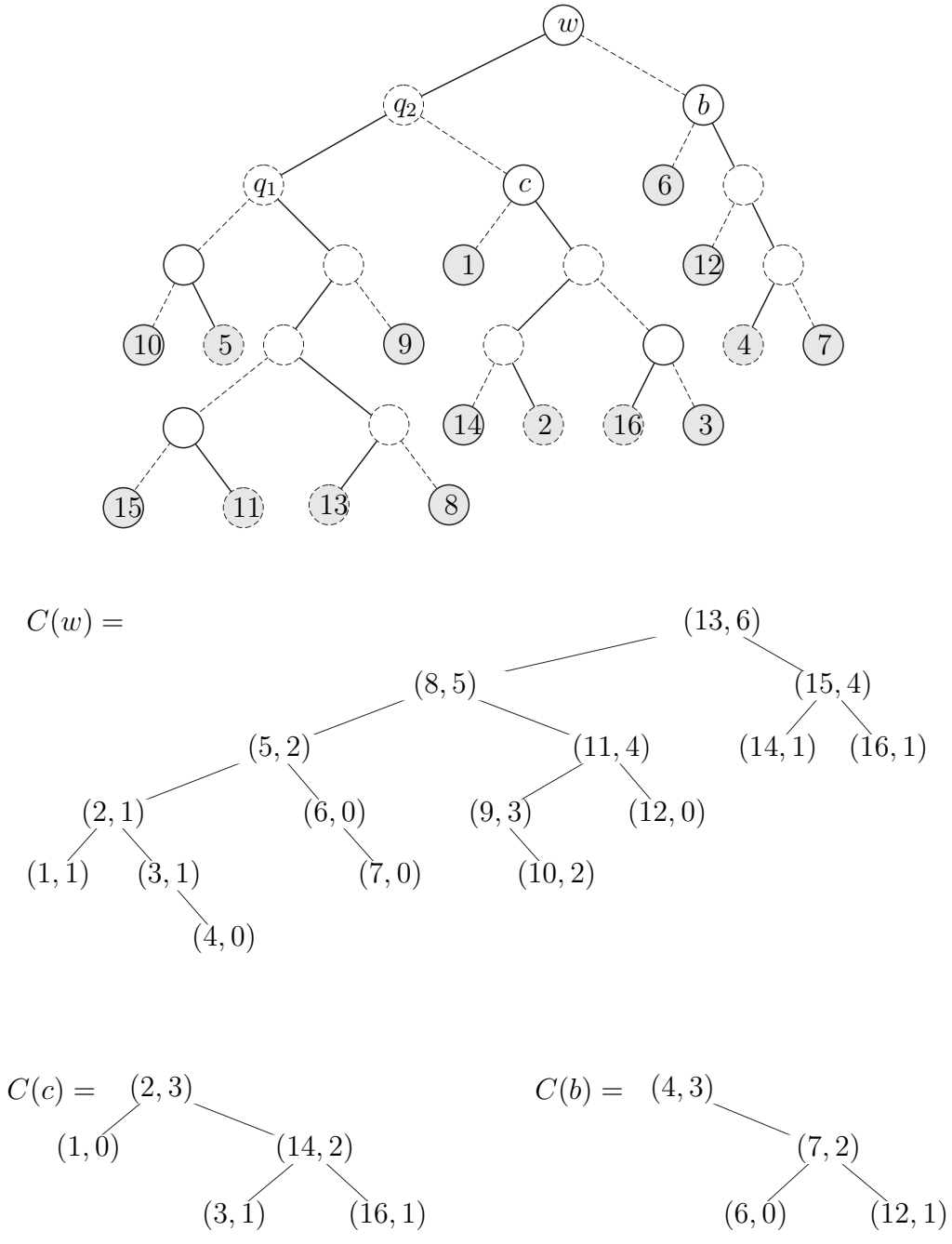

Figure 4.7: A sample tree with the rank order indicated inside the leaves. Light and heavy edges are indicated using dashed and solid lines respectively. Leaves are marked gray. Index nodes are marked using solid outlines. The nontrivial (containing more than two nodes) Cartesian trees are included.

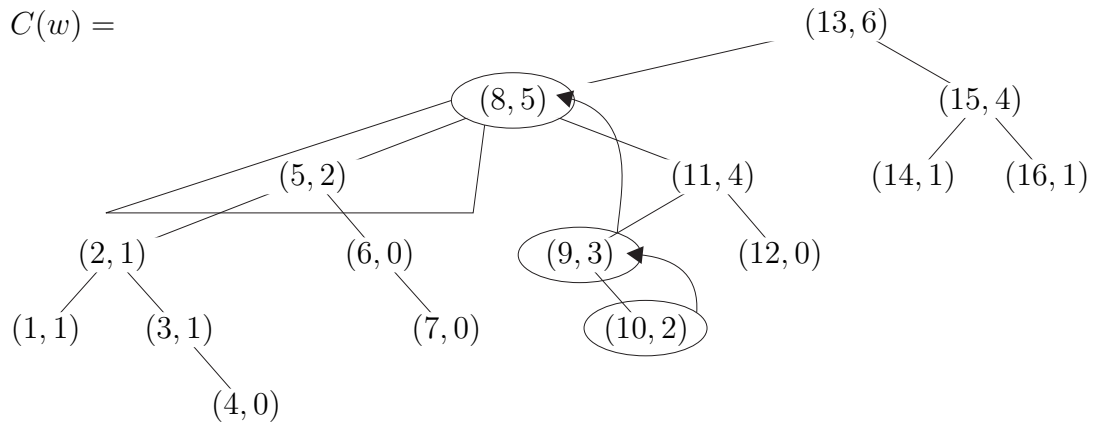

Figure 4.8: An example query. Query node is q_1, $k = 4 = 2^2$. The index node of q_1 is w, $depth_w(q_1) = 2$, $NA(q1)[2] = (10, 2)$, $L(10, 2) = (9, 3)$, $L(9, 3) = (8, 5)$, $L(8, 5) = NULL$.

(we label the nodes by their corresponding pair of coordinates). Next all the elements in the lists are considered together with their left subtrees. The subtrees are cut off at the appropriate depths (2 and 1).

4.8.3 Discussion

With points of the form (x, y) stored as a Cartesian tree C, the query reduces to the following: *Return the first k (according to the x coordinate) elements of C, whose y coordinate is grater or equal to some d.* An infix traversal of C guarantees returning items according to the x coordinate (rank) and its heap structure allows considering all items above a given depth. So if the query would be to return *all* leaves descending from a node in rank order, there would be no problem — we could just traverse the Cartesian tree using an appropriate cut-off value and that would be it. The problem is the additional, *vertical* cut-off due to k. During the infix traversal, while going "down", we visit elements which will be listed later. With the k limit, these items may never end up listed, so the algorithm would seize to be output-sensitive. See, for example, how in the first query example we are able to skip the node $(11, 4)$ — this node would be considered in a regular traversal of the tree and would keep the algorithm from being output-sensitive (there could be many such nodes). That is why we need the sequence of nodes l — to identify exactly all the nodes visited during the traversal and no more. Unfortunately, this list depends both on q and k.

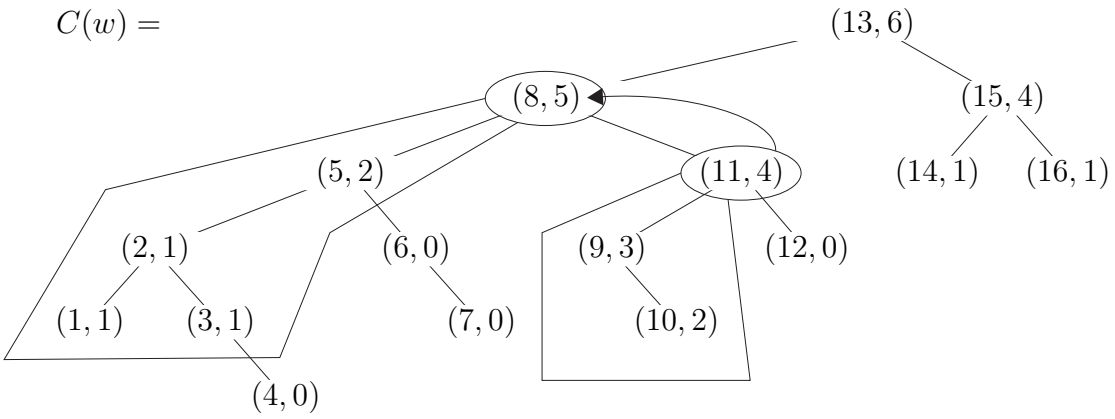

Figure 4.9: An example query. Query node is q_2, $k = 8 = 2^3$. The index node of q_2 is w, $depth_w(q_2) = 1$, $NA(q2)[3] = (11, 4)$, $L(11, 4) = (8, 5)$, $L(8, 5) = NULL$.

Note: The L pointers in the tree correspond to (some of) the leaf list pointers in the original solution and the "first pointer" in the three-pointer version. The other pointers are made obsolete by the fact that we now traverse all the little subtrees. Sadly, there is still the need to "know where to start", so we still need the old node arrays.

The main problem with the dynamization of this structure is inserting items into the Cartesian trees while additionally maintaining the the L pointers in these trees. For a study of dynamic Cartesian trees, see Part III.

Chapter 5

Rank-sensitivity — a general framework

5.1 Introduction and definitions

In this chapter we present a framework for adding rank-sensitivity to a class of output-sensitive data structures. The class in question contains data structures, in which the items are ordered in such a way, that the items satisfying a query form $O(\text{polylog}(n))$ intervals of consecutive entries each, where n is the number of items stored in the structure.

For example suffix trees store items in such a way that the result of a query corresponds to a set of leaves descending from one of the internal nodes. If we consider the order on the leaves defined by the tree structure, then the result of a query corresponds to a single interval of consecutive leaves.

One-dimensional range trees again have this property, that the items returned by a query correspond to a single interval of consecutive leaves, if we consider the tree-induced order on the leaves. For higher dimensions of range trees, rather than one interval, we will have a number of intervals of the order of $O(\text{polylog}(n))$, where n is the number of items in the range tree. These intervals are, however, always disjoint.

We provide a framework for transforming such structures into rank-sensitive data structures (see Definitions 3.0.5 and 3.0.4). Let $s(n)$ be the number of items (including their copies) stored in any such data structure D. Let $O(t(n) + \ell)$ be its query time and let $|D|$ be the number of memory words of space it occupies, each word composed of $O(\log n)$ bits. We obtain a rank-sensitive data structure D', with $O(t(n) + k)$ query time, increasing the space to $|D'| = |D| + O(s(n) \log^{\epsilon} n)$ memory

words, for any positive constant $\epsilon < 1$.

We allow for changing *rank* on the fly during the lifetime of the data structure D', with ranking values in the range from 0 to $O(n)$. In this case, query time becomes $O(t(n)+k)$ plus $O(\log n/\log\log n)$ per interval and each change in the ranking takes $O(\log n)$ time per item copy.

Our solution operates in *real time* as we discuss later.

When D allows for insert and delete operations on the set of items, we obtain an additive cost of $O(\log n/\log\log n)$ time per query operation and $O(\log n)$ time per update operation in D'. The space occupancy is $|D'| = |D|+O(s(n)\log n/\log\log n)$ memory words. Whether $|D'| = |D|+(s(n))$ space is attainable, is an open problem. The preprocessing cost of D' is dominated by sorting the items according to *rank*, plus the preprocessing cost of D.

In order to achieve these bounds, we base our solution on a general scheme borrowed from previous work on two-dimensional range trees [27], adapting it to our case so as to improve time bounds with respect to the best known results for two-dimensional range searching [64] (since our problem is simpler). At the heart of our solution is an extended version of the Q-heap [43], which we call the *multi-Q-heaps*, described in Section 5.3.

As far as we know, previous work on range searching cannot be easily adapted to achieve our bounds. For example, a structure for range searching on the grid has been given by Overmars [69]. It takes $O(n\log n)$ space and the query time is $O(\sqrt{\log n}+k)$ for arbitrary four-sided range queries. However, this solution solves a different problem than the one we are solving for the following reasons. First, the items returned are not sorted. Second, what is specified in the query is the largest (smallest) rank to be returned and not the number k of items to return. One does not follow from the other in any trivial way. These observations hold also for other range trees, e.g. [5], and for priority search trees [27] which handle arbitrary values and three-sided queries.

5.2 The static case and its dynamization

Our starting point is a well-known scheme adopted for two-dimensional range trees [27]. Following the global rebuilding technique described in [68], we can restrict our attention to values of n in the range $0\ldots O(N)$ where $n = \Theta(N)$. Consequently, we use lookup tables tailored for N, so that when the value of N must double or halve, we also rebuild these tables in $o(N)$ time. Our word size is $O(\log N)$. As can be

seen from [68], time bounds can be made worst-case.

We recall that the interval is taken from the list of items $L = e_0, e_1, \ldots, e_{n-1}$, indicating with $pos(e_i)$ the dynamic position of e_i in L (but we do not keep *pos* explicitly) and with $rank(e_i)$ its rank value in $0 \ldots O(N)$. We use a special rank value $+\infty$ that is larger than the other rank values; multiple copies of $+\infty$ are each different from the other (and take $O(\log N)$ bits each).

5.2.1 Static case on a single interval

Structure

We employ a weight-balanced B-tree W [7] as the skeleton structure. At the moment, suppose that W has degree exactly two in the internal nodes and that the n items in L are stored in the leaves of W, assuming that each leaf stores a single item. For each node $u \in W$, let $R(u)$ denote the explicit sorted list of the items in the leaves descending from u, according to *rank* order (see Figure 5.1). If u_0 and u_1 are the two children of u, we have that $R(u)$ is the merge of $R(u_0)$ and $R(u_1)$.

Using a technique described by Chazelle [21], we can use 0s and 1s to mark the entries in $R(u)$ that originate, respectively, from $R(u_0)$ and $R(u_1)$. We obtain $B(u)$, a bit string of $|R(u)|$ bits, totalizing $O(n \log n)$ bits, hence $O(n)$ words of memory, for the entire W (see [21]).

We also implement fractional cascading [89]. We maintain two bi-directional pointers, f_0, f_1, for each element $e \in R(u)$—with f_0 pointing to e's predecessor in $R(u_0)$ and f_1 to e's predecessor in $R(u_1)$ in rank order—for its two children u_0 and u_1. In particular, exactly one of these pointers will always point to the next-level copy of e.

Query algorithm

Rank query works similarly to the query performed on range trees (see [27] and Chapter 2.2). Given entries e_i and e_j in L, we locate their leaves in W, say v_i and v_j. We find their least common ancestor w in W (the case $v_i = v_j$ is trivial). On the path from w to v_i, we traverse $O(\log n)$ internal nodes. If during this traversal, we go from node u to its left child u_0, we consider the list $R(u_1)$, where u_1 is the right child of u. Analogously, on the path from w to v_j, if we go from node u to its right child u_1, we consider list $R(u_0)$ for its left child. In all other cases, we skip the nodes (including w and its two children). Clearly, we include v_i and v_j if needed.

At this point, we reduce the rank-sensitive query for v_i and v_j to the problem of selecting the top k best-ranking items from $O(\log n)$ rank-sorted lists $R()$, containing integers in $0 \ldots O(N)$ (see next Section).

Following Chazelle's approach, we do not explicitly store the lists $R()$, but keep only the bit strings $B()$ (see Figure 5.2) and the additional machinery for translating bits in $B()$ into entries in $R()$, which occupies $O(n \log^{\epsilon} n)$ words of memory, for any positive constant $\epsilon < 1$. (See Lemma 2 in Section 4 of [21].) As a result, we can retrieve the sorted items of lists $R()$ using Chazelle's approach.

5.2.2 Polylog intervals in the dynamic case

Structure

In the general case, we are left with the problem of selecting the top k best-ranking items from $O(\text{polylog}(n))$ rank-sorted *dynamic* lists $R()$, containing integers in $0 \ldots O(N)$. We cannot use Chazelle's machinery in the dynamic setting. We maintain the degree b of the nodes in the weight-balanced B-tree W, such that $(\beta/4) \log n / \log\log n \leq b \leq (4\beta) \log n / \log\log n$, for a suitable constant in $0 < \beta < 1$. As a result from [7], the height of the tree is $O(\log n / \log b) = O(\log n / \log\log n)$. We also explicitly store the lists $R()$, totalizing $O(n)$ words per level of W, and thus yielding $O(n \log n / \log\log n)$ words of memory. Note that the cost of splitting/merging a node $u \in W$ along with $R(u)$ can be deamortized [7].

To enable the efficient updating of all the lists $R()$, we use a variation of dynamic fractional cascading [22, 62] described in [72], which performs efficiently on graphs of a non-constant degree. Fractional cascading does not increase the overall space complexity. At the same time, for a given element e of list $R(u)$, it allows locating the predecessor (in rank order) of e in $R(u')$ when u' is a child or parent of u. This locating is performed in time $O(\log b + \log\log n)$ which amounts to $O(\log\log n)$ under our assumption concerning b, the degree of the tree.

Let us consider a single interval identified by a rank query. It is described by two leaves v_i and v_j, along with their least common ancestor $w \in W$. However, we encounter $O(\log n / \log\log n)$ lists $R()$ in each node u along the path from w to either v_i or v_j. For any such node u, we must consider the lists for u's siblings either to its left or its right. So we have to merge $O((\log n / \log\log n)^2)$ lists on the fly. But we can only afford $O(\log n / \log\log n)$ time.

We solve this multi-way merging problem by introducing *multi-Q-heaps* in Section 5.3, extending Q-heaps [43]. A multi-Q-heap stores $O(\log n / \log\log n)$ items

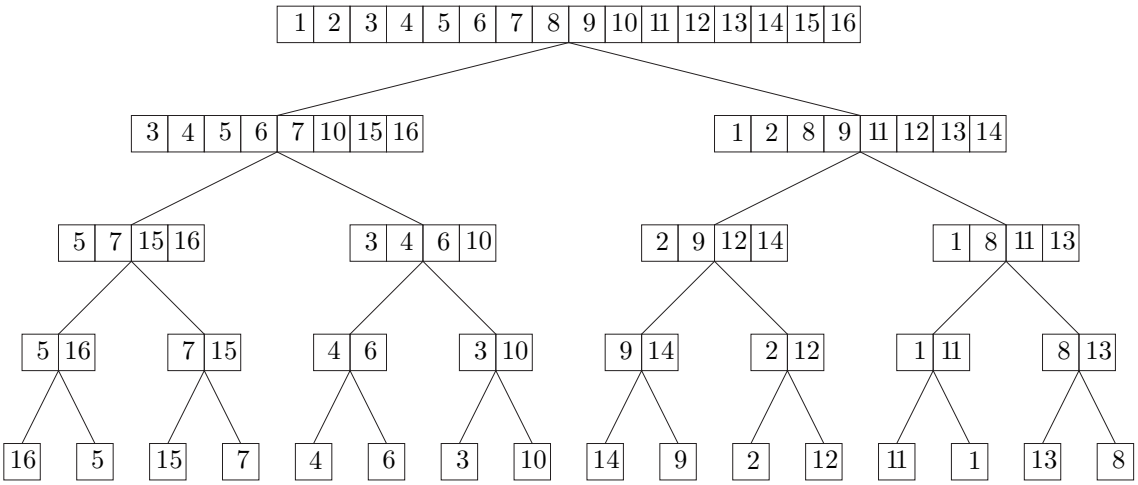

Figure 5.1: An overview of the tree structure employed. The actual structure used is a weight-balanced B-tree. Each internal node stores a list of items stored in the leaves descending from it, sorted by rank.

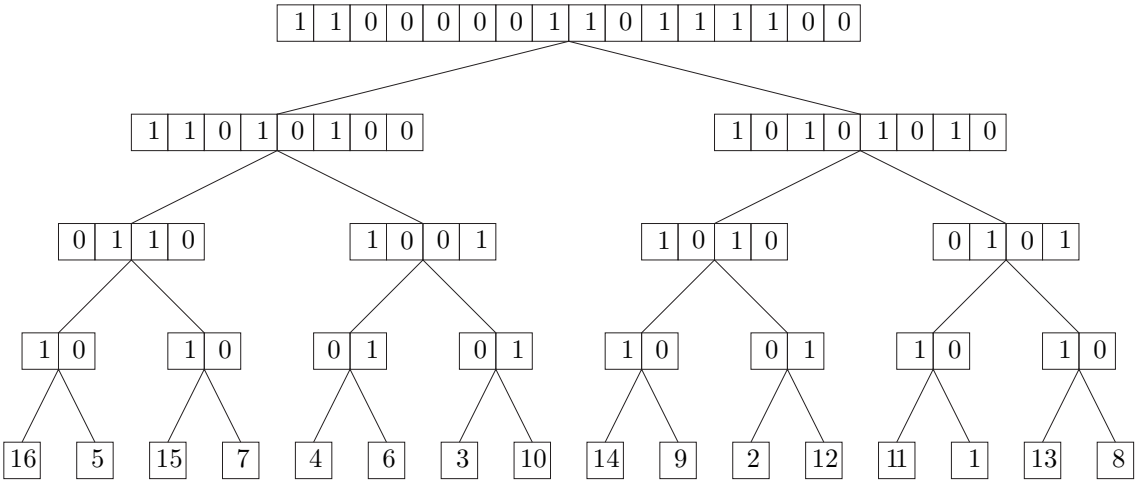

Figure 5.2: Tree from Figure 5.1, except that this time a spacial compression scheme is used which rather than storing the lists of descending leaves in the internal nodes, stores only the information about which subtree (left or right) each element belongs to. Chazelle [21] provides a mechanism for translating these bit values back into their original values.

from a bounded universe $0 \dots O(N)$, and performs constant-time search, insertion, deletion, and find-min operations. In particular, searching and finding can be *restricted* to any *subset* of its entries, still in $O(1)$ time. Each instance of a multi-Q-heap requires just $O(1)$ words of memory. These instances share common lookup tables occupying $o(N)$ memory words. We refer the reader to Theorem 5.3.1 in Section 5.3 for more details.

We employ our multi-Q-heap for the rank values in each node $u \in W$. This does not change the overall space occupancy, since it adds $O(n)$ words, but it allows us to handle rank queries in each node u in $O(1)$ time per item as follows. Let $d = \alpha \log N / \log\log N$ be the maximum number of items that can be stored in a multi-Q-heap (see Theorem 5.3.1). We divide the lists $R()$ associated with u's children into d clusters of d lists each. For each cluster, we repeat the task recursively, with a constant number of levels and $O(\text{polylog}(n)/d)$ multi-Q-heaps. We organize these multi-Q-heaps in a hierarchical pipeline of constant depth. For the sake of discussion, let's assume that we have just depth 2. We employ a (first-level) multi-Q-heap, initially storing d items, which are the minimum entry for each list in the cluster. We employ further d (second-level) multi-Q-heap of d entries each, in which we store a copy of the minimum element of each cluster.

Query algorithm

To select the top k best-ranking leaves, we extract the k smallest entries from the lists by using the above multi-Q-heaps: We first find the minimum entry, x, in one of the second-level multi-Q-heaps, and identify the corresponding first-level multi-Q-heap. From this, we identify the list containing x. We take the entry, y, following x in its list. We extract x from the first-level multi-Q-heap and insert y. Let z be the new minimum thus resulting in the first level. We extract x from the suitable second-level multi-Q-heap and insert z. By repeating this task k times, we return the k leaves in rank-sensitive fashion.

This does not yet solve our problem. Consider the path from, say, v_i to its ancestor w. We have $O(\log n / \log\log n)$ lists for each node along the path. Fortunately, our multi-Q-heaps allow us to handle any subset of these lists, in constant time. The net result is that we need to use just $O(\log n / \log\log n)$ multi-Q-heaps for the entire path. For each node u in the path, the find-min operation is limited to the lists corresponding to a subset of u's sibling at its right. They form a contiguous range, which we can easily manage with multi-Q-heaps. Hence, we can apply the above

2-level organization, in which we have $O(\log n/\log\log n)$ multi-Q-heaps in the path from v_i to w in the second level. (An analogous approach is for the path from v_j to w.) In this way, we can perform a multi-way merging on the fly for finding the least k keys in sorted rank order, in $O(k + \log n/\log\log n)$ time.

Note that the bound is real-time as claimed. In the case of polylog intervals, we use an additional multi-Q-heap hierarchical organization (of constant depth) to merge the items resulting from processing each interval separately.

Handling modifications

We now describe how to handle rank changes of entries in L, as well as insertions and deletions in L.

Changing the rank of entry e_i, say in leaf $v_i \in W$ is performed in a top-down fashion. It affects the nodes on the path from the root of W to v_i.

The list $R(u)$ for each node u along this path contains a copy of e_i but whose rank no longer complies with the ordering of the list. This element is extracted from the list and inserted into the correct place on this list. Both the element itself and the new correct place can be located in the list associated with the root in $O(\log n)$ time.

Next, using the fractional cascading structure, we can relocate the copy of e_i in the list for the next node in the downward path to v_i, having already done it in the current node. This takes $O(\log\log n)$ time per node, thus yielding $O(\log n)$ total time to relocate the copy of e_i in all the lists of the path.

As for the insertions in L (and also in W), they follow the approach in [7]; moreover, the input item e has its $rank(e)$ value, in the range $0 \ldots O(N)$, inserted into the lists $R()$ of the ancestor nodes as described above.

Deletions are simply implemented as weak, changing the rank value of deleted items to $+\infty$. When their number is sufficiently large, we apply rebuilding as in [27].

If the original data structure contains multiple copies of the same item (as in the case of a range tree) then the update in the rank-sensitive structure is applied separately to the individual copies.

We obtain the following result. Let D be an output-sensitive data structure for n items, where the ℓ items satisfying a query on D form $O(\text{polylog}(n))$ intervals of consecutive entries. Let $O(t(n) + \ell)$ be its query time and $s(n)$ be the number of items (including their copies) stored in D.

Theorem 5.2.1 *We can transform D into a static rank-sensitive data structure D', where query time is $O(t(n) + k)$ for any given k, thus reporting the top k best-ranking items among the ℓ listed by D. We increase the space by an additional term of $O(s(n) \log^{\epsilon} n)$ memory words of space, each of $O(\log n)$ bits, for any positive constant $\epsilon < 1$. For the dynamic version of D and D', we allow for changing the ranking of the items, with ranking values in $0 \ldots O(n)$. In this case, query time becomes $O(t(n)+k)$ plus $O(\log n / \log \log n)$ per interval. Each change in the ranking and each insertion/deletion of an item take $O(\log n)$ time for each item copy stored in the original data structure. The additional term in space occupancy increases to $O(s(n) \log n / \log \log n)$.*

5.3 Multi-Q-heaps

The multi-Q-heap is a relative of the Q-heap [43]. Q-heaps provide a way to represent a sub-logarithmic set of elements in the universe $[N] = 0 \ldots O(N)$, so that such operations as inserting, deleting or finding the smallest element can be executed in $O(1)$ time in the worst case. The price to pay for the speed is the need to set up and store lookup tables in $o(N)$ time and space. These tables, however, need only to be computed once as a bootstrap cost and can be shared among any number of Q-heap instances. So at the price of $o(N)$ (charged to the preprocessing cost), we obtain a very efficient mechanism for operating on small sets.

Our multi-Q-heap is functionally more powerful than Q-heap, as it allows performing operations on any *subset* of d common elements, where $d \leq \alpha \log N / \log \log N$ for a suitable positive constant $\alpha < 1$. Naturally, this could be emulated by maintaining Q-heaps for all the different subsets of the elements, but that solution would be exponential in d (for each instance!), while our multi-Q-heap representation requires two or three memory words and still supports constant-time operations.

Our implementation based on lookup tables is quite simple and does not make use of multiplications or special instructions (see [38, 78] for a thorough discussion of this topic). Here we describe a simpler version which deals with contiguous subsets of elements (ranges) rather than arbitrary subsets, however it can be easily extended to deal also with arbitrary subsets. The supported operations are as follows:

- Create a heap for a given list of elements.
- Find the minimum element within a given range.

- Find an element within a given range of items.
- Update the element at a given position.

In the rest of the section, we prove the following result.

Theorem 5.3.1 *There exists a constant $\alpha < 1$ such that d distinct integers in $0 \ldots O(N)$ (where $d \leq \alpha \log N / \log\log N$) can be maintained in a multi-Q-heap supporting search, insert, delete, and find-min operations in constant time per operation in the worst case, with $O(d)$ words of space. The multi-Q-heap requires a set of precomputed lookup tables taking $o(N)$ construction time and space.*

5.3.1 High-level implementation

The d elements are integers from $[N]$. We can refer to their binary representations of $w = \lceil \log[N] \rceil$ bits each. These strings can be used to build a compacted trie on binary strings of length w. However, instead of labeling the leaves of the compact trie with the strings (elements) they correspond to, we keep just the trie shape and the skip values contained in its internal nodes, like in [4, 35]. We store the d elements and their satellite data in a separate table. To provide a connection between the trie and the values, we store a permutation which describes the relation between the order of elements in the trie and the order in which they are stored in the table.

When searching for an element, we first perform a blind search on the trie [4, 35]. Next we access the table corresponding to the found element and we compare it with the sought one. Note that this way we only access the table of values in one place, while the rest of the search is performed on the trie. With an assumption about the maximum number of elements stored in the multi-Q-heap, we can encode both the trie and the permutation as two single memory words. The operations are then performed on these encodings and only the relevant entries in the value table are accessed, which guarantees constant time. The operations on the encoded structures are realized using lookup tables and bit operations.

To support multi-Q-heap operations, we store a single structure containing all the elements. We implement all the extended operations so as to consider only the given subset of the elements while maintaining constant time. We assume a word size of $w = O(\log N)$ bits. We use d to refer to the number of items stored in the multi-Q-heap. We assume $d \leq \alpha \log N / \log\log N$ for some suitable constant $\alpha < 1$. We use $x_0, x_1, \ldots, x_{d-1}$ to refer to the list of items stored in the multi-Q-heap. For

our case, the order defined by the indexes is relevant (when using multi-Q-heaps in the nodes of the weight-balanced B-tree of Section 5.2).

5.3.2 Multi-Q-heap: Representation

The multi-Q-heap can be represented as a triplet (S, τ, σ), where S is the array of elements stored in the structure, τ is the encoding of the compact trie and σ is an encoding of the permutation. The array S stores the elements $x_0, x_1, \ldots, x_{d-1}$ in that order and their satellite data. Each element occupies a word of space. We do not consider satellite data for the sake of discussion.

The encoding of the trie, τ, can be defined in the following fashion. First, let us encode the shape of the binary tree of which it consists. This tree is binary, with no unary nodes and edges implicitly labeled with either 0 or 1. We can encode it by traversing the tree in inorder (visiting first 0 edges and then 1 edges) and outputting the labels of the edges traversed. This encoding can be decoded unambiguously and requires $4d - 4$ bits, since each edge is traversed twice and there are $2d - 2$ edges in the trie.

Next, we encode the skip values. The internal nodes (in which the skip values are stored) are ordered according to their inorder which leads to an ordered list of skip values. Each skip value is stored in $\lceil \log w \rceil$ bits, so the encoding of the list takes $(d-1)\lceil \log w \rceil$ bits. For a suitable value of α the complete encoding of the trie does not exceed $1/4 \log N$ bits and hence can be stored in one word of memory.

The permutation σ reflects the array order $x_0, x_1, \ldots, x_{d-1}$ with respect to the order of these elements sorted by their values (which is the same as the inorder of the corresponding leaves in the trie). There are $d!$ possible permutations, so we choose α so that $\log d! < 1/4 \log N$ and the encoding on the permutation fits in one word of memory. We use the encoding described in [66], which takes linear time to rank and unrank a permutation, hence to encode and decode it.

5.3.3 Multi-Q-heap: Supported operations

Init

The *Init* operation sets up all the lookup tables required for implementing the multi-Q-heap. It needs to be performed only once. See Section 5.3.4 for details concerning the lookup tables. These lookup tables are used in the implementations of the operations described below. If invoked multiple times, only the first is effective.

Create

The *Create* operation takes the array S of values $x_0, x_1, \ldots, x_{d-1}$ and sets up the structures τ and σ. It takes the time required to construct the compact trie for d elements, hence $O(d)$.

Findmin

The function *Findmin* returns the smallest element among the elements $x_i, \ldots, x_j$ stored in the multi-Q-heap. We implement it using the lookup table *Subheap* and *Index*. We use $Subheap[\tau, \sigma, i, j]$ to obtain τ' and σ', the structure for elements $x_i, \ldots, x_j$. We then use $Index[\sigma', 1]$ to obtain the array index of the smallest element in the range.

Search

The function *Search* searches the subset of elements $x_i, \ldots, x_j$ stored in the multi-Q-heap and returns the index of the element in the multi-Q-heap which is smallest among those not smaller than y, where $y \in [N]$ can be any value. As previously, we use $Subheap[\tau, \sigma, i, j]$ to obtain τ' and σ', the subheap for elements $x_i, \ldots, x_j$. We then search the reduced trie for x', the first half (bitwise) of x, by looking up $u = Top[\tau', x']$. Next, using $LDescendant[\tau', u]$, we identify one of the strings descending from u and compare this string with x' to compute their longest common prefix length *lcp*.

This computation can be done in constant time with another lookup table, which is standard and is not described. If $lcp < 1/2 \log N$, then $LDescendant[\tau', u]$ identifies the sought element. If $lcp = 1/2 \log N$, we continue the search in the bottom part of the trie by setting $u = Top[\tau', x', u]$. Also here $LDescendant[\tau', u]$ provides the answer.

Update

The *Update* operation replaces the element x_r in the array S with y, where $y \in [N]$ can be any value. It updates τ and σ accordingly.

We first simulate the search for y in τ, as described in the previous paragraph to find the rank i of y among $x_0, \ldots x_d$ and use this together with the table $UpdatePermutation[\sigma, r, i]$ to produce the updated permutation. We then use values obtained during the simulated blind search for y in τ to obtain values needed to access the *UpdateTrie* table. During the search we find the node u at which the search

for y ends (in the second half of the trie in the case the search gets that far) and the *lcp* obtained by comparing its leftmost descendant with y. We use $Ancestor[\tau, u, lcp]$ for identifying the node whose parent edge is to be split for inserting. The *lcp* is the skip value the parameter c depends on the bit at position $lcp + 1$ of y. With this information, we access *UpdateTrie* to obtain the encoding of the updated trie.

5.3.4 Multi-Q-heap: Lookup tables

This section describes the lookup tables required to perform the operations described in the previous section. The number of tables can be reduced, but at the expense of the clarity of the implementation description.

Index

The *Index* table provides a way for obtaining the array index of an element given the inorder position of its corresponding leaf in the trie (let us call this the trie position). It contains the appropriate array index entry for every possible permutation and trie position.

The space occupancy is $2^{1/4 \log N} \times d \times \log d = N^{1/4} \times d \times \log d = o(N)$.

Inverse index

The $Index^{-1}$ table is the inverse of *Index* in the sense that it provides a way of obtaining a trie position from an index, by containing a position entry for every possible permutation and index.

The space occupancy is the same as for *Index*.

Subheap

The *Subheap* table provides a means of obtaining a new subheap structure, (S, τ', σ'), from a given one (S, τ, σ). The new subheap structure uses the same array S, but takes into account only the subset $x_i, \ldots, x_j$ of its items. Note that only τ, σ, i, and j are needed to determine τ' and σ' and not the values stored in S.

The new trie τ' is obtained from the old trie τ by removing leaves not corresponding to $x_i, \ldots, x_j$ (these can be identified using σ). The new permutation σ' is obtained from the old one σ by extracting all the elements with values $i, \ldots, j$ and moving them to the beginning of the permutation (without changing their relative order) so that they now correspond to the appropriate $j - i + 1$ leaves of the reduced trie.

The space occupancy of *Subheap* is $2^{1/4 \log N} \times 2^{1/4 \log N} \times d \times d \times 1/4 \log N \times 1/4 \log N = N^{1/2} \times d^2 \times (1/4 \log N)^2 = o(N)$.

Top and Bottom

The *Top* and *Bottom* tables allow searching for a value in the trie. The searching for a value must be divided into two stages, because a table which in one dimension is indexed with a full value, one of $O(N)$ possible, would occupy too much space. We therefore set up two tables: *Top* for searching for the first $1/2 \log N$ bits of the value and *Bottom* for the remaining.

The table *Top* contains entries for every possible trie τ and x', the first $1/2 \log N$ bits of some sought value x. The value in the table specifies the node of τ (with nodes specified by their inorder position) at which the blind search [4, 35] for x' (starting from the root of the trie) ends.

The table *Bottom* contains entries for every possible trie τ, x'' (the second $1/2 \log N$ bits of some sought value x) and an internal node of the trie v. The value in the table specifies the node of τ at which the blind search [4, 35] for x'' ends, but in this case the blind search starts from v instead of from the root of the trie.

The space occupancy of *Top* is $2^{1/4 \log N} \times 2^{1/2 \log N} \times \log d = N^{3/4} \times \log d = o(N)$ and the space occupancy of *Bottom* is $2^{1/4 \log N} \times 2^{1/2 \log N} \times d \times \log d = N^{3/4} \times d \times \log d = o(N)$.

UpdateTrie

The *UpdateTrie* table specifies a new multi-Q-heap and permutation which is created from a given one by removing the leaf number i from τ and inserting instead a new leaf. The new leaf is the c child of a node inserted on the edge leading to u. This new node has skip value s.

The space occupancy is $2^{1/4 \log N} \times d \times 2 \times 2^{\log \log N} \times d \times 1/4 \log N \times 1/4 \log N = N^{1/4} \times d^2 \times 1/8 \log^3 N = o(N)$.

UpdatePermutation

The *UpdatePermutation* table specifies the permutation obtained from σ if the element with index r is removed and an element ranking i among the original elements of the multi-Q-heap is inserted in its place.

The space occupancy is $2^{1/4 \log N} \times d \times d \times 1/4 \log N = N^{1/4} \times d^2 \times 1/4 \log N = o(N)$.

LDescendant

The *LDescendant* table specifies the leftmost descending leaf of node u in τ.

Its space occupancy is $2^{1/4 \log N} \times d \times \log d = N^{1/4} \times d \times \log d = o(N)$.

Ancestor

The *Ancestor* table specifies the shallowest ancestor of u having a skip value equal to or greater than s.

The space occupancy is $2^{1/4 \log N} \times d \times 2^{\log \log N} \times \log d = N^{1/4} \times d \times \log N \times \log d = o(N)$

5.3.5 General case

In order to implement a general multi-Q-heap which handles arbitrary subsets rather than just ranges, we need to encode a permutation π in a single word since $x_0, \ldots, x_{d-1}$ can be further permuted due to the insertions and deletions. An arbitrary subset is represented by a bit mask that replaces the two small integers i and j delimiting a range. The sizes of the lookup tables in Section 5.3.4 increase but still remain $o(N)$.

5.4 Conclusions

In this chapter we presented a general framework for adding rank-sensitivity to a class of output-sensitive data structures, such as suffix trees or range trees.

We showed how such a structure can be augmented to answer rank-sensitive queries, that is to return the top k highest-ranking query results. Our rank-sensitive query achieves reporting time proportional to the query parameter k rather than to the total number of items satisfying the query.

We showed results for both the static and dynamic model. In the static model, we achieved rank-sensitivity at the cost of increasing the space complexity by a factor of $O(\log^{\epsilon} n)$ for any positive constant $\epsilon < 1$, while in the dynamic model the space complexity is increased by a factor of $O(\log n / \log \log n)$.

Part III

Dynamic Cartesian trees

5.5 Introduction

Cartesian trees are described in Section 1.2. Note that the recursive construction of the structure does not contain any ambiguities, which means that the set of points uniquely determines the shape of the Cartesian tree[1]. The points induce a rigid subdivision of the Cartesian plane, because each point divides the space below it into two halves and no tree edge can cross this dividing vertical line.

This "rigidness" is exploited in applications but does not leave room for any balancing operations. The tree height can even be linear with respect to the number of elements it stores, which leads to very high update time in the worst case. For this reason, Cartesian trees have only been used as static data structures or considered in a stochastic setting. To our knowledge, the amortized cost of modifying a Cartesian tree has not been studied.

In this part of the work, we study the amortized cost of update operations on the Cartesian tree and present the first dynamic version of the Cartesian tree. Our solution supports insertions and week deletions in amortized logarithmic time per operation. We do not maintain an equivalent representation of the Cartesian tree, but provide its actual form, so that the tree structure can be exploited between each operation, as needed, regardless of the particular application.

5.5.1 Background

In 1978 Francon et al. [40] introduced a priority queue structure called the pagoda, which shares some features of the Cartesian tree. The term Cartesian tree itself was introduced by Vuillemin [85] in a work aiming to illustrate the usefulness of such geometrical objects in various algorithms involving sorting and searching. Since then, Cartesian trees have appeared in a number of different of settings and have found numerous applications.

An important and heavily exploited feature of Cartesian trees is that they provide a parallel between the range maximum query (RMQ) and the least common ancestor (LCA) problems (see for example [12]). It is easy to check that if $\langle \bar{x}, \bar{y} \rangle = \text{LCA}(\langle x_L, y_L \rangle, \langle x_R, y_R \rangle)$ for some $x_L < x_R$ then $\bar{y} = \text{RMQ}(x_L, x_R)$ is the maximum y value among the nodes whose x values fall between x_L and x_R. This fact is the basis for the realization of an optimal static structure supporting range maximum queries [12].

[1] Throughout this paper we assume that all of the points have distinct coordinates.

Applications of Cartesian trees can also be found it the domain of text algorithms. Besides the fact that the aforementioned RMQ and LCA algorithms, which are based on Cartesian trees, play an important role in many string algorithms (see for example [34]), Cartesian trees also have a meaning in themselves. In particular, they provide a connection between two important structures used in string algorithms: the suffix tree [87] and the least common prefix (LCP) array. A suffix tree for a given text can be seen as a Cartesian tree of index-value pairs of items in the LCP array for this text (provided neighboring nodes of the Cartesian tree with the same y value are joined).

Cartesian trees are also known as *treaps* [73] when the priorities (y values) of the nodes are assigned randomly with a uniform distribution and the tree is used as a binary search tree (for the x values). The random distribution of y values guarantees good average dynamic behavior of the tree as the height is expected to stay $O(\log n)$ in this case. It is important to note at this point that the logarithmic expected update time is indeed achieved only under a random uniform distribution (with x values independent from y). In fact, a thorough study of the average behavior of Cartesian trees Devroye [30] shows that if there is a dependence between the x and y values, then the expected height of the tree (and hence the time to perform update operations) is $O(\sqrt{n})$ or even $O(n)$ in some cases.

Cartesian trees are also used in dynamic memory management. Stephenson [77] introduces a storage allocation scheme which uses Cartesian trees to store available blocks according to their physical location (x coordinate) and size (y coordinate). This approach is called Fast Fits and is implemented in some operating systems, such as SunOS 4.1.

Shi and JáJá [74] use Cartesian trees for range reporting. Cartesian trees in themselves can be used for dominance reporting in a straightforward way, much like many other similar structures. Note, however, that only Cartesian trees are capable of reporting items according to the order defined by one of the coordinates, which may be very useful in some applications.

Priority search trees [59] somewhat resemble Cartesian trees. The main difference between the two is that the partition into subtrees in Cartesian trees is determined by the x coordinate of the root, while in priority trees it is chosen so as to maintain balance in the tree. This balance makes priority search trees an efficient tool for answering range search queries. However, the price to pay for the balancing is the loss of the x order present in the Cartesian tree and so priority search trees can not always be used where Cartesian trees can. For example, note that the items

returned by a range query on priority search trees returns items in an order which matches neither the x nor y order of the nodes, while a query on Cartesian trees returns items in x order, as this order matches that of the tree traversal.

As far as we know, the amortized complexity of updating the Cartesian tree is currently unknown, except for the fact that a Cartesian tree can be built from scratch in $O(n \log n)$ time, or even in $O(n)$ time when the points are already given in sorted order [44].

5.5.2 Motivation

We are interested in studying Cartesian trees in a dynamic setting, under insertions and deletions of arbitrary points. Before discussing the algorithmic issues, we motivate our study by observing that Cartesian trees are difficult to update and no polylogarithmic update bounds are currently known. There exist similar structures with good update time, such as priority search trees [59], but those have weaker topological properties in the Cartesian plane and can not always be used in place of Cartesian trees. Due to the difficulty of the updates, the use of Cartesian trees has been confined to the static case.

Along with the least common ancestor query (LCA), Cartesian trees can be used to answer range maximum queries (RMQ) in constant time. Range maximum queries are a generalization of priority-queue queries in which the find-min operation is restricted to ranges of values. The implementation of LCA itself in [12] uses Cartesian trees for a reduction from general trees to RMQ, while the dynamic version of LCA [24] does not use Cartesian trees. Hopefully, the efficient dynamization of the Cartesian trees can make a first significant step in finding a suitable array of applications in a dynamic setting, such as solving the dynamic version of the constant-time RMQ (in logarithmic update time). This can stimulate further research, such as extending the dynamic constant-time LCA to treat cut and link operations among trees (in logarithmic update time).

From an algorithmic point of view, designing the update algorithms for a Cartesian tree T and analyzing their amortized complexity is challenging and non-trivial. The aforementioned rigidness of T is an obstacle when updating the structure since it can lead to a very unbalanced shape (uniquely defined by the points in T). We do not simply maintain an equivalent representation of T. We obtain the actual structure of T as it results from the standard $O(n)$-time insertion algorithm. Under these assumptions, we obtain an amortized update cost of $O(\log n)$ time.

The requirement of maintaining the actual Cartesian tree between each operation does *not* permit the efficient amortization of a sequence of interleaved insertions and deletions of points in T. Using the configuration shown in Figure 5.3, it is not difficult to create a sequence of updates that requires changing $O(n)$ edges per operation. However, we study the results of a sequence of pure insertions or pure deletions and show that the worst-case scenario from Figure 5.3 can not happen repeatedly for such a sequence.

5.5.3 Our results

In order to illustrate our findings, let us split the cost of inserting a new point $\langle \bar{x}, \bar{y} \rangle$ into the Cartesian tree T. We evaluate the *searching cost* as the time complexity of traversing T and locating the place in which to insert $\langle \bar{x}, \bar{y} \rangle$, as well as locating the edges which should be modified. We then account for the *restructuring cost* as the time complexity of actually changing the structure of T as a result of the insertion of $\langle \bar{x}, \bar{y} \rangle$. (Looking at Figure 5.3, we can realize that the searching cost does not amortize because of T's traversal.)

We analyze the behavior of T in a combinatorial setting, and prove that the restructuring cost of insertions is $O(1 + \mathscr{H}(T)/n)$ time, where $\mathscr{H}(T) = O(n \log n)$ is an entropy-related measure for the partial order encoded by T, as described in Section 5.7. The key observation is that the worst-case situation depicted in Figure 5.3 cannot happen repeatedly when subsequent insertions are performed in the same neighborhood.

This analysis based on $\mathscr{H}(T)$ may be interesting in itself, since it could also be useful when analyzing the average height of random treaps and other heap-based data structures. A random choice of y values maximizes the entropy at the root, and recursively at its two children, giving rise to an almost balanced tree structure.

We also show that the search cost can be reduced to $O(\log n)$ time. The latter requires *locating* the elements to update and actually performing the update. Moreover, most of these updates are of a special kind, a fact which will be vital to the next part of our work, where we show a reduction to a constrained problem on intervals.

Here, weak deletions (logically marking nodes as deleted and periodically rebuilding T) can be amortized when coupled with insertions, at the price of having a constant fraction of nodes in T marked as deleted. We can maintain T under insertions and weak deletions in $O(\log n)$ amortized time per operation, using $O(n)$

space. (Handling both insertions and deletions cannot be amortized well otherwise.) As previously mentioned, the amortized cost of restructuring T is $O(1 + \mathscr{H}(T)/n)$ time per operation.

We employ the *companion interval tree* in our solution, which is based on the interval tree [33] implemented using a weight-balanced B-tree [7]. We take advantage of the special properties of the constrained problem and the results of our analysis to provide algorithms that match the amortized analysis.

Previous work already exploited a constrained version of stabbing queries on dynamic sets of intervals. [36] and [79] consider interval endpoints which belong to a bounded universe, while ours belong to a unbounded universe under the comparison model. [53] devises a special solution for nested intervals (as in our case) and the stabbing query identifies the max-priority interval using the max-cost operation in dynamic trees.

In our case, however, we need a more constrained query, namely, reporting all the stabbed intervals having priority below an arbitrary threshold, which cannot be handled by max-cost. [1] presents an improvement over the work of [53], but this still cannot help in our queries. Furthermore, all the aforementioned solutions cannot guarantee an important property that allows us to obtain logarithmic bounds. Our intervals can shrink $O(n \log n)$ times. The above solutions only allow a shrink operation to be implemented as a deletion followed by an insertion, at a non-constant cost. This is not sufficient in our case, so we prove combinatorial properties of our intervals, which guarantee that each shrink operation requires just $O(1)$ amortized time and does not require deleting and reinserting the interval.

5.6 Preliminaries

We consider a set of points — ordered pairs $\langle x, y \rangle$ drawn from an unbounded universe. We assume a total order defined on the first as well as the second coordinate of the points. We assume that the points have distinct coordinates. We define a Cartesian tree for the set of points as follows:

1. The root stores the point $\langle \bar{x}, \bar{y} \rangle$ with the maximum y-value in the set.

2. The x-value of the root, $\bar{x}$, induces a partition of the remaining elements into sets $L = \{\langle x, y \rangle : x < \bar{x}\}$ and $R = \{\langle x, y \rangle : x > \bar{x}\}$. The roots of the Cartesian trees obtained from L and R are the left and right children of the root $\langle \bar{x}, \bar{y} \rangle$, respectively.

We identify T with the set of points it stores in its nodes and hence write $\langle x, y\rangle \in T$ for a point in T. Since the set of points *uniquely* determines the Cartesian tree, this is not misleading.

We write $e = (\langle x_L, y_L\rangle, \langle x_R, y_R\rangle)$ to denote an edge in the tree.

All logarithms are to the base of 2.

5.6.1 Properties of Cartesian trees

As follows from the recursive definition, a rigid subdivision of the Cartesian plane is induced by T (see Figure 1.3), because each point divides the space below it into two halves and no tree edge can cross this dividing vertical line. Also this space subdivision is determined unambiguously by the set of points in T. We shall exploit the properties of this space subdivision in the rest of the paper, so it is useful to express it by means of the following fact:

Fact 5.6.1 *Let T be a Cartesian tree. Let $\langle x, y\rangle \in T$ be a node in the tree and let $e = (\langle x_L, y_L\rangle, \langle x_R, y_R\rangle)$ be and edge in T ($x_L < x_R$). If $y > \min(y_L, y_R)$ then either $x < x_L$ or $x > x_R$. In other words, no edge can cross the vertical line originating from any node and extending down.*

From the above fact, we can deduce the following properties of Cartesian tree edges:

Fact 5.6.2 *Let $e = (\langle x_L, y_L\rangle, \langle x_R, y_R\rangle)$ and $e' = (\langle x'_L, y'_L\rangle, \langle x'_R, y'_R\rangle)$ be edges in Cartesian tree T and suppose that $x_L < x'_L$. In this case, either $x'_R < x_R$ (the projections of the edges onto the x-axis are nested) or $x_R \leq x'_L$ (the projections of the edges onto the x-axis do not overlap or overlap only by point $x_R = x'_L$).*

Fact 5.6.3 *Let $e = (\langle x_L, y_L\rangle, \langle x_R, y_R\rangle)$ and $e' = (\langle x'_L, y'_L\rangle, \langle x'_R, y'_R\rangle)$ be edges in Cartesian tree T, if the projection of e' is nested within the projection of e, then $y'_i \leq y_j$ for $i, j \in \{L, R\}$.*

Fact 5.6.4 *Let $e = (\langle x_L, y_L\rangle, \langle x_R, y_R\rangle)$ and $e' = (\langle x'_L, y'_L\rangle, \langle x'_R, y'_R\rangle)$ be edges in Cartesian tree T, if the projections of the edges onto the x-axis are nested, then e and e' lie on the same path in T.*

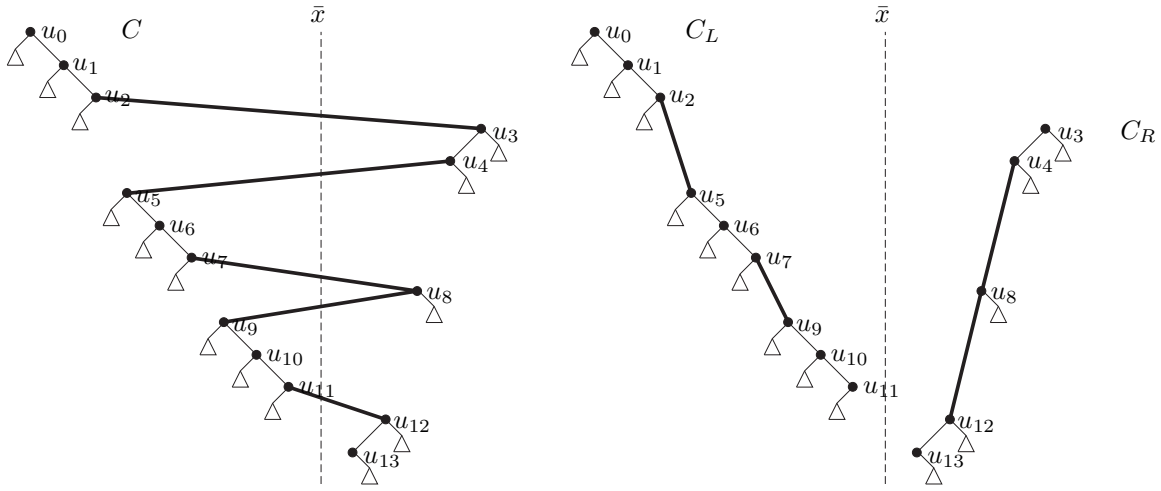

Figure 5.3: The outcome of operation $split(C, \bar{x})$ (from left to right) resulting from an insertion, and that of $merge(C_L, C_R)$ (from right to left) resulting from a deletion; one is the other's inverse. Note that the path revealed in the illustration can be proportional in length to the overall tree size in which case an insert or delete operation affects $O(n)$ edges of a tree of size n.

5.6.2 Dynamic operations on Cartesian trees

The main dynamic operations on a Cartesian tree are those of inserting and deleting points. We review these operations as they will be the basis for the analysis in the rest of the paper. It is useful to see insertions and deletions as based on the more basic operations of splitting and merging Cartesian trees.

Split

We define $split(C, \bar{x})$ for a Cartesian tree C and a value $\bar{x}$ as the operation returning $C_L = \{\langle x, y\rangle \in C : x < \bar{x}\}$ and $C_R = \{\langle x, y\rangle \in C : x > \bar{x}\}$ (see Figure 5.3). In other words, this operation splits the Cartesian tree C with a vertical line at $\bar{x}$. Nodes to the left of this line end up in tree C_L and nodes to the right of it end up in C_R. Edges not crossed by the line are not affected by the split operation and are present in either C_L or C_R (depending on if they are left or right of the line). We study what happens to the edges crossed by the vertical line at $\bar{x}$ as a result of the split operation. These edges are marked in bold in Figure 5.3 and we refer to them as affected edges.

Notice that the affected edges are each one below the other, because they are all

crossed by the vertical line $\bar{x}$. Hence, from facts 5.6.2 and 5.6.4, they are nested and are all part of the same path. We label the nodes along this path $u_0, u_1, \ldots u_s$, as in Figure 5.3.

The affected edges, when removed, partition C into a number of subtrees. We can order these subtrees according to the y order of their roots. Every second subtree belongs to C_L and the remaining belong to C_R. In C, the affected edges connect these subtrees according to the y order of their roots. These edges are no longer present in C_L and C_R. However, new edges in C_L and C_R connect the subtrees within the C_L tree as well as within C_R separately, still according to the y order of the roots.

For reasons which will become apparent later, we prefer not to view the splitting operation in terms of removed and inserted edges, but rather it terms of edge transformations. Notice that for each (but one) affected edge $e = (u_i, u_{i+1})$, there is an edge $e' = (u_i, u_j)$, where $j > i+1$ and u_j is the root of the next subtree after the one containing u_i on the same side of the $\bar{x}$ line as u_i. The only edge which does not have such a counterpart is the lowest of the affected edges (indeed C_L and C_R have one less edge that C). So each but the lowest edge affected by a split operation can be considered transformed into its corresponding edge in C_L or C_R. We call such a transformation "shrinking", because it shrinks the projection of an edge onto the x-axis.

Example 5.6.5 For example, in Figure 5.3, edge (u_2, u_3) shrinks and becomes (u_2, u_5), (u_4, u_5) becomes (u_4, u_8), (u_7, u_8) becomes (u_7, u_9), (u_8, u_9) becomes (u_8, u_{12}), and the lowest affected edge (u_{11}, u_{12}) is deleted.

A degenerate case of *split*$(C, \bar{x})$ occurs when $\bar{x}$ is either smaller or greater than all the x values of the nodes in C. In that case C_L is empty and $C_R = C$ or the other way around. In this case the resulting tree has the same number of edges as the original, rather than having one less, and no parts of the tree are affected by the split. Most of the following arguments, however, hold also for this extreme case, so we will not make a distinction between the two cases, unless stated otherwise.

We use k to denote the number of edges which shrink due to *split*$(C, \bar{x})$. If the number of affected edges is one or zero then $k = 0$. Otherwise k is one less than the number of affected edges, since the last affected edge gets deleted and not shrunk.

Fact 5.6.6 *Edges affected by a split operation are all nested before the split and are disjoint (or overlap by at most a single point) after the split.*

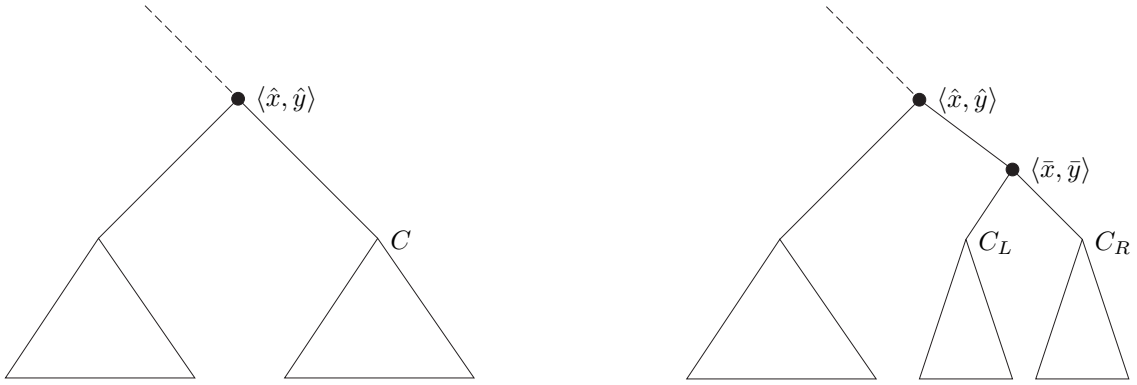

Figure 5.4: An illustration of an insert operation where $\langle \bar{x}, \bar{y} \rangle$ is inserted as the right child of $\langle \hat{x}, \hat{y} \rangle$.

Insertion

Let us consider $T' = T \cup \{\langle \bar{x}, \bar{y} \rangle\}$, where T is a Cartesian tree and $\langle \bar{x}, \bar{y} \rangle$ is a new point to insert into T. If $\bar{y} > y$ for each $\langle x, y \rangle \in T$, then the new point becomes the root of T' and its two children are T_L and T_R, respectively, obtained by invoking $split(T, \bar{x})$. In all other cases $\langle \bar{x}, \bar{y} \rangle$ has a parent in T'. Let us denote this parent $\langle \hat{x}, \hat{y} \rangle$.

We can find $\langle \hat{x}, \hat{y} \rangle$ in the following way. Let $\langle x_L, y_L \rangle \in T$ be the rightmost point such that $x_L < \bar{x}$ and $y_L > \bar{y}$ and let $\langle x_R, y_R \rangle \in T$ be the leftmost point such that $x_R > \bar{x}$ and $y_R > \bar{y}$. If there are points in T above $\langle \bar{x}, \bar{y} \rangle$, at least one of these two points must exist. The parent of $\langle \bar{x}, \bar{y} \rangle$ in T' is the lower of these two points (i.e., the one with the minimum between y_L and y_R). This follows almost directly from the recursive construction in Section 5.6.

In the case that $\langle \hat{x}, \hat{y} \rangle = \langle x_L, y_L \rangle$, let C be the right subtree of $\langle \hat{x}, \hat{y} \rangle$ in T and in the other case let it be the left subtree. Then C_L and C_R obtained by invoking $split(C, \bar{x})$ become the children of $\langle \bar{x}, \bar{y} \rangle$ in T'. See Figure 5.4 for an illustration.

Note that the edges affected by an insertion are those connecting the new node $\langle \bar{x}, \bar{y} \rangle$ to $\langle \hat{x}, \hat{y} \rangle$ and to C_L and C_R, the edge which connected $\langle \hat{x}, \hat{y} \rangle$ to C, and the edges affected by the split of C. Of these, only the edges affected by the split are not constant in number.

Example 5.6.7 Figure 5.5 shows an insertion applied to the sample tree from Figure 1.3. The affected and transformed edges are indicated.

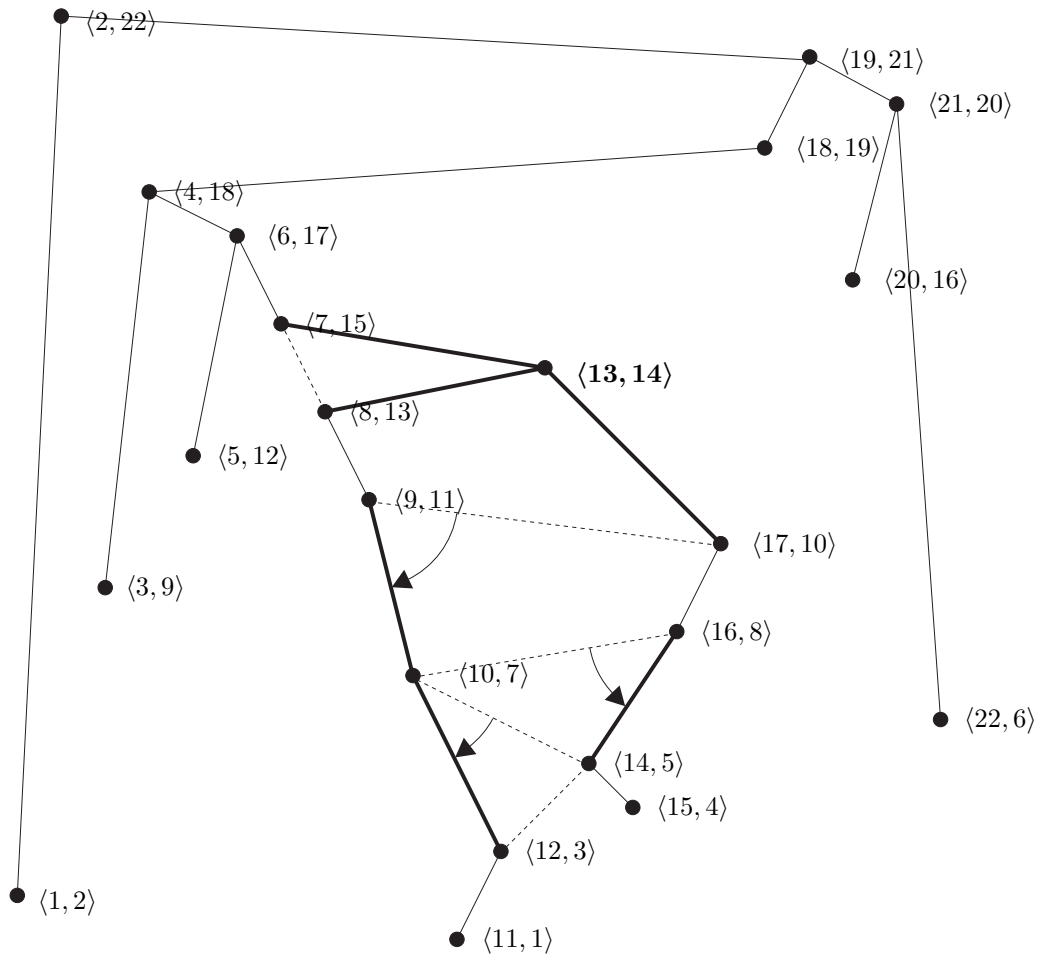

Figure 5.5: The transformation of the Cartesian tree from Figure 1.3 caused by the insertion of point $\langle 13, 14\rangle$. New edges are marked in bold and edges no longer present in the new tree are dashed. Arrows indicate old edges shrinking and becoming new edges.

Merge and deletion

Merge and deletion can be viewed as the exact opposites of the split and insert operations and they cause edge stretching which is the opposite of edge shrinking.

Fact 5.6.8 *The insertion (deletion) of a point causes $O(1)$ edges in the tree to be inserted or deleted and causes k edges to shrink (stretch), where $0 \leq k \leq n$. Hence, the number of edge modifications is $k + O(1)$ for each inserted or deleted point.*

5.7 Bounding the number of edge modifications

We now focus on the number of edge modifications as expressed by Fact 5.6.8. In this section, we consider insertions only, so we are interested in bounding the number of inserted, deleted and shrunk edges for a sequence of n insertions of points into an initially empty Cartesian tree. Since the number of shrunk edges for each insertion is $k = O(n)$, it may appear that n insertions cause $O(n^2)$ such edge modifications. However, we make the following observation which will eventually lead to a bound which is lower: Inserting a point into a Cartesian tree does not require performing comparisons of the y-coordinates of the nodes in the Cartesian tree (except pertaining to the node being inserted). On the other hand, reversing this operation by deleting the same node *does* require a number of comparisons proportional to the number of affected edges.

In other words, we can fully determine the shape of the Cartesian tree $T' = T \cup \langle \bar{x}, \bar{y} \rangle$ if we know the shape of T and are able to compare $\bar{x}$ and $\bar{y}$ with the values stored in the nodes of T and we do not need to compare the values stored in T with each other. However, in order to determine the shape of $T = T'/\langle x, y \rangle$ from the shape of T', we may need to perform comparisons on y values of points in T'. This suggests that information is *lost* as a result of an insertion and that entropy can serve as a measure of a tree's potential for costly (affecting many edges) insertions.

We now formalize this intuition. A Cartesian tree T induces a partial order on its elements: $\langle x, y \rangle \prec_T \langle x', y' \rangle$ if and only if $\langle x, y \rangle$ is a descendant of $\langle x', y' \rangle$. The intuition behind this definition is that if $\langle x, y \rangle$ descends from $\langle x', y' \rangle$ then we know that $y < y'$ from the heap condition of the tree. In all other cases we can not guess the relative order of y and y' just by looking at the position of these two nodes in the tree (we must perform an actual comparison on the values y and y').

Note that even if y values are drawn from a total order, the order $\prec_T$ is only partial. So we can use any y-ordering of points satisfying $\prec_T$ without violating the

heap condition of the tree. We will say that an ordering is *valid* for T if it satisfies $\prec_T$. Note that unlike the y-ordering, the x-ordering induced by the shape of the tree is always total.

Consider a partially ordered set $\langle P, \prec_P \rangle$ of n distinct items. A *linear extension* of P is a permutation $p_1, p_2, \ldots, p_n$ of the items in P such that if $p_i \prec_P p_j$ is defined, then $1 \leq i < j \leq n$. For example, the linear extensions of the partially ordered set $P = \{\mathtt{A}, \mathtt{T}, \mathtt{G}, \mathtt{C}\}$ where $\mathtt{A} \prec_P \mathtt{T}$ and $\mathtt{G} \prec_P \mathtt{C}$ are ATGC, AGTC, AGCT, GATC, GACT, and GCAT, since A appears before T in these permutations and G before C (see [88]).

In general, calculating the number of linear extensions of a partial order is #P-Complete [19]. In this paper, however, we deal only with partial orders in the form of binary trees in which case the counting of linear extensions is much simpler. (Anyway, we never use the exact number explicitly—only discuss its bounds.)

For a given Cartesian tree T, let $\mathscr{L}(T)$ denote the number of linear extensions of the partial order $\prec_T$ induced by T. We introduce the notion of the *missing entropy* of T as

$$\mathscr{H}(T) = \log \mathscr{L}(T), \tag{5.1}$$

which is the information needed to sort the set of y values in T starting only from the information inferred from the shape of the tree. Since the number of linear extensions cannot exceed the number of permutations, we trivially have $\mathscr{H}(T) \leq \log n! = O(n \log n)$ from order entropy and Stirling's approximation. However, $\mathscr{H}(T)$ can even be zero if the partial order is a total order (this occurs when the Cartesian tree is a single path). We exploit the notion of missing entropy for bounding the number of shrunk edges.

Theorem 5.7.1 *Inserting n points into an initially empty Cartesian tree results in $O(n)$ edge insertions and deletions and $O(n + \mathscr{H}(T))$ edges shrinking, where T is the resulting Cartesian tree.*

In order to demonstrate Theorem 5.7.1, we use the missing entropy of the Cartesian tree given in equation (5.1), as the potential function in our amortized analysis.

The observed loss of information as a result of an insertion $T' = T \cup \{\langle \bar{x}, \bar{y} \rangle\}$ is reflected in the difference between the number of linear extensions of T and of T'. This change can be measured by the ratio $\mathscr{L}(T')/\mathscr{L}(T)$, but it is more convenient to consider its logarithm, which is the change in our potential, $\mathscr{H}(T') - \mathscr{H}(T)$. We recall that this potential is limited by the entropy of a partial order.

The proof goes along the following lines. Consider an insertion $T' = T \cup \{\langle \bar{x}, \bar{y} \rangle\}$ that splits a subtree C of the current Cartesian tree T into C_L and C_R as discussed in Section 5.6.2. By Fact 5.6.8, this operation results in $O(1)$ inserted and deleted edges in T, plus k shrunk edges. We claim that

$$k = O(\mathscr{H}(T') - \mathscr{H}(T)). \tag{5.2}$$

Equation (5.2) holds asymptotically, namely, there exist constants $c_0, k_0 > 0$ such that $k \leq c_0(\mathscr{H}(T') - \mathscr{H}(T))$ for every $k > k_0$. Its proof follows from Lemma 5.7.3 below: Let $l = \lceil (k+2)/2 \rceil$, and take the logarithms as in (5.1), obtaining

$$\log \mathscr{L}(T) + \log \binom{k+2}{l} \leq \log \mathscr{L}(T').$$

Since $k = O(\log \binom{k+2}{l}) = O(\log \mathscr{L}(T') - \log \mathscr{L}(T))$, we obtain the claimed bound in (5.2). It remains to prove Lemmas 5.7.2 and 5.7.3.

Lemma 5.7.2 *Every linear extension of T is also a linear extension of T'.*

Proof: Let us use the terminology of Section 5.6.2: $T' = T \cup \{\langle \bar{x}, \bar{y} \rangle\}$ and the insertion splits a subtree C of T into subtrees C_L and C_R of node $\langle \bar{x}, \bar{y} \rangle$ in T'. Let us use p to denote the parent of $\langle \bar{x}, \bar{y} \rangle$ in T'. Linear extensions of T do not consider the newly inserted point $\langle \bar{x}, \bar{y} \rangle$, so we only consider relations between points of T. We need to show that for any two nodes $a, d \in T$, we have $d \prec_{T'} a \Rightarrow d \prec_T a$ (if a is an ancestor of d in T' then it was also an ancestor of d in T).

Suppose that the premise is true and that a is an ancestor of d in T'. We break the proof up into cases according to the locations of d in T':

Case 1: d is not a descendant of $\langle \bar{x}, \bar{y} \rangle$ in T'. In this case also a can not be a descendant of $\langle \bar{x}, \bar{y} \rangle$ in T' and so both nodes belong to a part of tree not affected by the insertion. So if a is an ancestor of d in T' it must also be and ancestor of d in T.

Case 2: $d \in C_L$ (resp. $d \in C_R$).

Case 2a: a is not a descendant of $\langle \bar{x}, \bar{y} \rangle$ in T'. In this case a must be an ancestor of $\langle \bar{x}, \bar{y} \rangle$ in T' in order to be an ancestor of d in T'. Therefore, a is either p or its ancestor. The node p and its ancestors are not affected by the insertion and so a must also be p or its ancestor in T. At the same time $d \in C$ in T and so a is an ancestor of d.

Case 2b: $a \in C_R$ (resp. $d \in C_L$). This is the case in which a and d belong to different subtrees of $\langle \bar{x}, \bar{y} \rangle$ in T'. But this contradicts the premise that a is an ancestor of d in T'.

Case 2c: $a \in C_L$ (resp. $d \in C_R$). This is the case in which both a and d belong to the same subtree $\langle \bar{x}, \bar{y} \rangle$ in T'. Therefore a and d belong to C in T and end up on the same side of the line at $\bar{x}$ when $split(C, \bar{x})$ is invoked. Let us recall from Section 5.6.2 that the edges affected by $split(C, \bar{x})$ induce a partition of C into subtrees. Both a and d must belong to subtrees on the same side of $\bar{x}$. If they belong to the same subtree then the path connecting them is the same in both T and T'. If they belong to different subtrees then a must lie on the path containing the affected edges in order to be an ancestor of d in T' and d must lie in a subtree whose root (let us call it r) has a smaller y value than a. In this case the path from d to r concatenated with the path from r to a connects d to a. The path from r to a must exist because it is part of the path containing the affected edges. Hence, d is a descendant of a in T.

This concludes the proof of Lemma 5.7.2. □

Lemma 5.7.3 *For each linear extension of T, there are at least $\binom{k+2}{l}$ unique linear extensions of T', where $l = \lceil (k+2)/2 \rceil$. Hence, $\mathscr{L}(T) \times \binom{k+2}{l} \leq \mathscr{L}(T')$.*

Proof: If $k = 0$ then the proof follows directly from Lemma 5.7.2, so we will consider the case where $k > 0$. In this case C_L and C_R are not empty and the split of C causes k edges to shrink.

Let us again use the terminology of Section 5.6.2 and consider the partition of C induced by the affected edges. This partition results in $k+2$ non-empty subtrees. Let us assume without loss of generality (the complimentary case is analogous) that the first subtree belongs to C_L. Let us label the roots of the subtrees $v_1^L, v_1^R, v_2^L, v_2^R, \ldots$ according to their decreasing y values. The last label is either v_l^L or v_l^R depending on the parity of k, where $l = \lceil (k+2)/2 \rceil$.

Now let us consider the partial order $\prec_T$ and some permutation P which is a linear extension of $\prec_T$. Notice that $v_i^R \prec_T v_i^L$ for any i, because all of the subtree roots belong to one path in T (the path containing the affected edges). Now, P is also an extension of $\prec_{T'}$ by Lemma 5.7.2. However, neither $v_i^R \prec_{T'} v_i^L$ nor $v_i^L \prec_{T'} v_i^R$ holds, because v_i^L and v_i^R belong to different subtrees of $\langle \bar{x}, \bar{y} \rangle$ in T' and so there is no ancestor-descendant relationship between them. We can use this property

to produce further unique extensions from P which are extensions of $\prec_{T'}$, but not of $\prec_T$, since P enforces just one of the $\binom{k+2}{l}$ distinct ways of merging $v_1^L, v_2^L, \ldots$ with $v_1^R, v_2^R, \ldots$, while $\prec_{T'}$ does not specify any ordering between the elements of the two sets. In other words, for each linear extension P of $\prec_T$ we can produce $\binom{k+2}{l}$ distinct valid extensions of $\prec_{T'}$ by shuffling the order of subtree roots while maintaining the relative order within the subtrees as well as between the subtrees belonging to C_L and the subtrees belonging to C_R.

□

We now complete the proof of Theorem 5.7.1. Consider an arbitrary sequence of n insertions into an initially empty Cartesian tree, denoted T_0, where $|T_0| = 0$. Let $T_1, T_2, \ldots, T_n$ denote the sequence of resulting Cartesian trees, where T_i is formed from T_{i-1} by the ith insert operation in the sequence, which shrinks k_i edges by Fact 5.6.8, for $1 \leq i \leq n$. Summing up the number of all the shrunk edges, we split the total sum according to the constant k_0 related to equation (5.2), as

$$\sum_{i=1}^{n} k_i = \sum_{i:k_i \leq k_0} k_i + \sum_{i:k_i > k_0} k_i. \tag{5.3}$$

We denote the indexes i such that $k_i > k_0$ in (5.3) by $i_1, i_2, \cdots, i_r$, where $i_1 < i_2 < \cdots < i_r$. Note that $k_{i_j} = O(\mathscr{H}(T_{i_j}) - \mathscr{H}(T_{i_j-1})) = O(\mathscr{H}(T_{i_j}) - \mathscr{H}(T_{i_{j-1}}))$ by equation (5.2) and Lemma 5.7.2, for $1 \leq j \leq r$. Applying these observations to the last term in (5.3) where $i = i_1, i_2, \ldots, i_r$, we obtain the following bound for (5.3):

$$\sum_{i:k_i > k_0} k_i = O\left(\sum_{j=1}^{r} \left(\mathscr{H}(T_{i_j}) - \mathscr{H}(T_{i_{j-1}})\right)\right) = O(\mathscr{H}(T_n)). \tag{5.4}$$

5.8 Implementing the insertions

In this section we show how to exploit the amortized upper bound on the number of edge modifications obtained in Section 5.7. Our aim is to maintain the Cartesian tree under a series of insertions. We do not want to maintain an equivalent data structure, but the actual tree as dictated by the set of points it contains between each insertion.

In order to describe our ideas, we split the cost of inserting a new point $\langle \bar{x}, \bar{y} \rangle$ into Cartesian tree T into a *searching cost* and a *restructuring cost*. The searching cost is the time complexity of traversing the Cartesian tree and locating the place in which to insert $\langle \bar{x}, \bar{y} \rangle$ and locating the edges that should be modified as a result

of the insertion of $\langle \bar{x}, \bar{y} \rangle$. The restructuring cost is the time complexity of actually changing the structure of T as a result of the insertion.

Let T denote the current Cartesian tree, and let k denote the number of edges that are shrunk during the insertion of $\langle \bar{x}, \bar{y} \rangle$ into T (see Fact 5.6.8). The resulting tree is denoted T', where $T' = T \cup \{\langle \bar{x}, \bar{y} \rangle\}$.

The restructuring cost is $O(1 + k)$ time according to our terminology. By Theorem 5.7.1, this can be amortized and becomes $O(1 + \mathscr{H}(T)/n)$, provided that we are able to implement the restructuring in time linearly proportional to the number of edges modified by the insertion.

The searching cost depends on how the search procedure is implemented. Recall that the tree can have linear height and so an algorithm relying on traversing the Cartesian tree would yield an $O(n)$ insertion cost. In order to provide a logarithmic search algorithm, we reduce the maintenance of the edges of T to a special instance of the dynamic maintenance of intervals. This reduction is based on mapping each edge $e = (\langle x, y \rangle, \langle x', y' \rangle)$ of T into its *companion* interval, (x, x'), where (x, x') is a shorthand for the set of coordinates $\hat{x}$ such that $x \leq \hat{x} < x'$. We store the companion intervals in an interval tree [33] as shown in Figure 5.6.

We shall see that insertion, deletion and shrinking of T's edges can be rephrased in terms of equivalent operations on their companion intervals. In the rest of this section, we will exploit the peculiarities of these intervals which are caused by the fact that they are not arbitrary but derived from the "rigid" subdivision of the space induced by T. We obtain the following:

- We obtain a searching cost of $O(\log n + k)$ time using a constrained stabbing query on the companion intervals and some other custom procedures on the companion interval tree.

- We obtain a restructuring cost of $O(\log n + k)$ amortized time by performing the $O(1)$ insertions and the deletions in $O(\log n)$ time and each edge shrink operation in $O(1)$ amortized time on the corresponding intervals. Note that the restructuring cost for the Cartesian tree alone is still $O(1 + k)$. The rest is for the maintenance of the companion intervals.

At this point it becomes clear why we need the concept of shrinking edges. Implementing all of the edge transformations as insertions and deletions into a data structure, such as our companion tree, would give rise to an extra factor of $O(\log n)$, hence $O(\mathscr{H}(T)/n \times \log n) = O(\log^2 n)$ per each point inserted into T. Instead, we

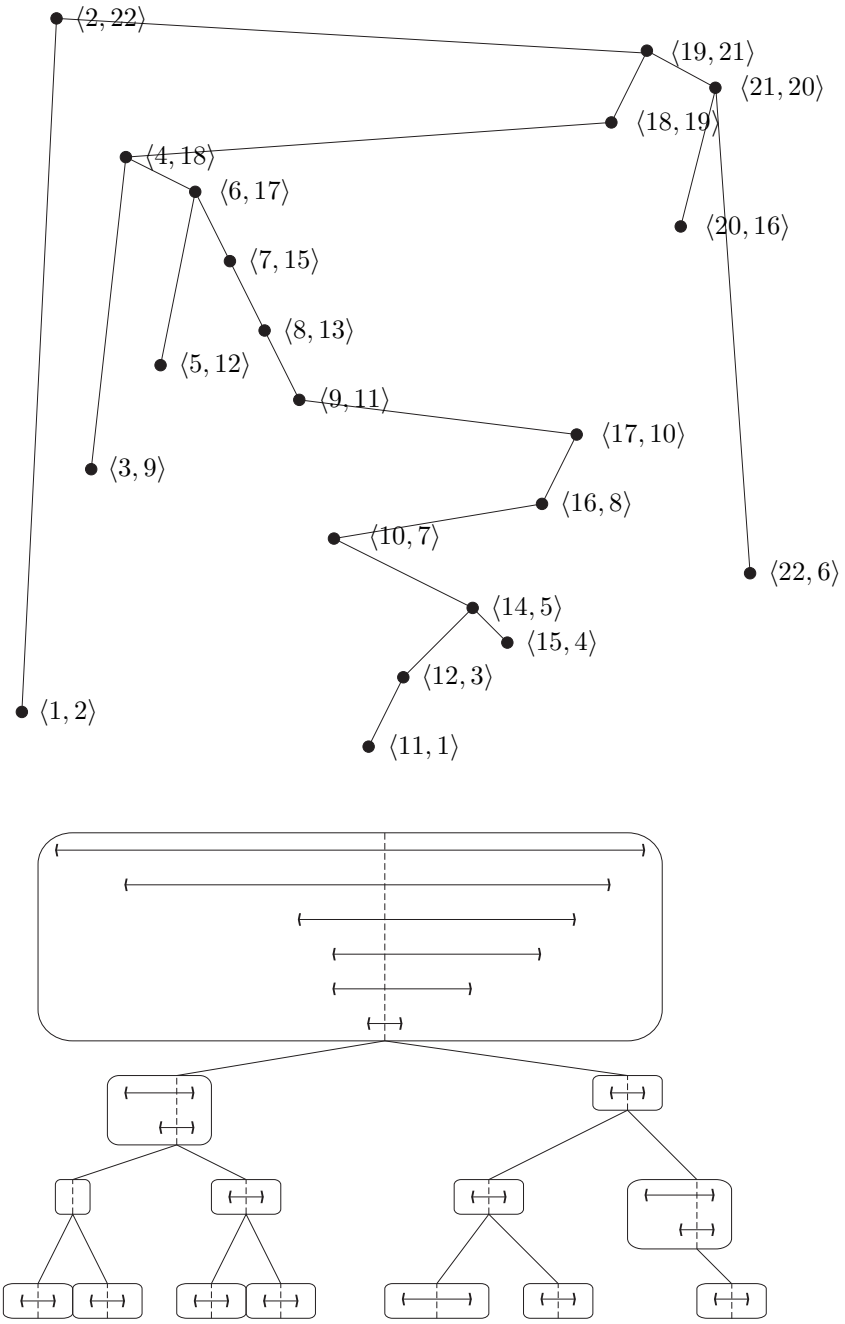

Figure 5.6: The tree from Figure 1.3 and its companion interval tree (below). For clarity, we included a binary interval tree in the illustration, rather than a B-tree.

are able to implement shrinking in $O(1)$ amortized time per shrunk edge and in this way maintain the optimal $O(\log n)$ cost of insertion.

5.8.1 The Companion Interval Tree

Our *companion interval tree*, W, is implemented using Arge and Vitter's weight-balanced B-tree [7] (alternatively, BB[α]-trees [67] can be employed) (see Section 1.3). We actually need a simpler version, without the leaf parameter. Let the weight $w(u)$ of a node u be the number of its descendant leaves. Let us recall that a weight-balanced B-tree W with branching parameter $a > 4$ satisfies the following constraints:

- All the leaves have the same depth and are on level 0.
- An internal node u on level ℓ has weight $(1/2)a^\ell < w(u) < 2a^\ell$.
- The root has at least two children and weight less than $2a^h$, where h is its level.

We fix $a = O(1)$ in our application, so W has height $h = O(\log_a |W|) = O(\log n)$ and each node (except maybe for the root) has between $a/4$ and $4a$ children. We denote the number of children of u by $\deg(u)$. The n leaves of W store elements in sorted order, one element per leaf. Each internal node u contains $d = \deg(u) - 1$ boundaries $b_1, b_2, \ldots, b_d$ chosen from the elements stored in its descendant leaves. In particular, the first child leads to all elements $e \leq b_1$, and the last child to all elements $e > b_d$, while for $2 \leq i \leq d$, the ith child contains all elements $b_{i-1} < e \leq b_i$ in its descendant leaves. Among others, W satisfies the property:

Lemma 5.8.1 ([7]) *After splitting a node u on level ℓ into two nodes, u' and u'', at least $a^\ell/2$ insertions have to be performed below u' (or u'') before it splits again. After creating a new root in a tree with n elements, at least $3n$ insertions are performed before it splits again.*

A weight-balanced B-tree W with n elements supports leaf insertions and deletions in $O(\log n)$ time per operation. Each operation only involves the nodes on the path from the leaf to the root and their children. We do not need to remove amortization or to split nodes lazily in W, since our bounds are amortized anyway.

We use this structure to store our companion intervals. Let $I(T)$ denote the set of companion intervals for the current Cartesian tree T. The leaves of W store the

endpoints of the intervals in $I(T)$. We store companion intervals in the nodes of the tree according to standard interval tree rules. Specifically, each node u contains d secondary lists, $L_1(u)$, $L_2(u)$, ..., $L_d(u)$, where $d = \deg(u) - 1$. For $1 \leq i \leq d$, list $L_i(u)$ is associated with the boundary b_i and stores all intervals $(x, x') \in I$ that contain b_i (i.e., $x \leq b_i < x'$), but are not stored in an ancestor of u.

Since any two intervals in $I(T)$ are either disjoint or one nested within the other (see Fact 5.6.2), every internal node $u \in W$ stores a number of intervals that is bounded by $O(w(u))$, which is crucial to amortize the costs by Lemma 5.8.1. Note that the same interval can be stored in up to d secondary lists of the same node, but not in different nodes, hence, the space occupancy remains linear. We keep these $O(1)$ copies of each interval in a thread.

Properties expressed in facts 5.6.2 and 5.6.3 guarantee that each list $L_i(u)$ stores nested intervals. The order according to which the intervals are nested matches the y order of their corresponding edges (from Fact 5.6.3). We maintain each list $L_i(u)$ of intervals sorted according to this order, from innermost (lowest) to outermost (highest). Each list supports the following operations, where $n_i = |L_i(u)|$ is the number of items in the list:

- Insert the smallest or largest item in $L_i(u)$ in constant time, and any item in $O(\log n_i)$ time.
- Delete any item from $L_i(u)$ in constant time, provided that we have a pointer to its location in $L_i(u)$.
- Perform a (one-dimensional) range query reporting the f items (in sorted order) between two values in $O(\log n_i + f)$ time. In the case of listing the first items, this takes $O(1 + f)$ time.
- Rebuild $L_i(u)$ from scratch in $O(n_i)$ time, provided that items are given in sorted order.

We implement the list $L_i(u)$ using a balanced search tree with constant update time [39, 56], in which we maintain a thread of the items linked in sorted order.[2] It is worth giving some detail on how to maintain the secondary lists when a node $u \in W$ splits into u' and u'' (see Lemma 5.8.1). Let b_i be the median boundary in u. Node

[2]Note that we do not need to use the finger search functionality in [56], which requires non-constant update time, as we can easily keep the minimum dynamically. In practice, we can implement $L_i(u)$ as a skip list.

u' gets boundaries $b_1, \ldots, b_{i-1}$ while u'' gets $b_{i+1}, \ldots, b_d$, along with their secondary lists and child pointers. The boundary b_i is inserted into u's parent. Note that no interval in $L_i(u)$ can belong to a secondary list in u's parent by definition. What remains is to use the threads of the copies of the intervals in $L_i(u)$ for removing these copies from secondary lists in u' and u''. But this takes $O(n_i)$ time, which is $O(1)$ amortized by Lemma 5.8.1.

We will now see how to use the companion interval tree W to implement the insertion of a new point into a Cartesian tree yielding $T' = T \cup \{\langle \bar{x}, \bar{y} \rangle\}$. As it should be clear at this point, we maintain both T and its auxiliary companion tree W. Following the insertion scheme described in Section 5.6.2, we should perform the following *actions*:

1. Find the node $\langle \hat{x}, \hat{y} \rangle$ in T that will become $\langle \bar{x}, \bar{y} \rangle$'s parent in T'.

2. Find the edges to shrink in T (and the one to delete) as a result of the split, which is part of the insert operation.

3. For each of the $O(1)$ edges inserted or removed in the Cartesian tree (see Section 5.6.2), insert or remove its companion interval from W. In particular this regards an edge originating from $\langle \hat{x}, \hat{y} \rangle$ (identified by action 1), the deleted edge identified by action 2, and the new edges connecting $\langle \bar{x}, \bar{y} \rangle$ to other nodes in T'.

4. For each companion interval of the k edges identified by action 2 as edges to shrink, perform the appropriate shrink of an interval in W. Notice that when an interval shrinks, it is relocated downward in W. The interval cannot go upward as a result of shrinking.

Action 3 is a standard operation that takes $O(\log n)$ time in W, so we focus on the remaining. Section 5.8.2 deals with actions 1 and 2 while Section 5.8.3 deals with action 4.

5.8.2 Searching Cost

We exploit the strong connection between the space subdivision induced by T (the dotted lines in Figure 1.3) and the companion intervals. In particular, we exploit the properties described in Section 5.6.1. Recall from Section 5.6.2 that the insertion of $\langle \bar{x}, \bar{y} \rangle$ into T involves finding the node $\langle \hat{x}, \hat{y} \rangle$ which will become the parent of $\langle \bar{x}, \bar{y} \rangle$

(action 1) and locating the edges affected by the split of the subtree of this parent (action 2). We first show how to perform the latter.

The edges affected by the split caused by the insertion of $\langle \bar{x}, \bar{y} \rangle$ into T are the edges in T which cross the vertical line at $\bar{x}$ below $\bar{y}$. Therefore, their companion intervals can be identified by a stabbing query $\bar{x}$ with the additional constraint that the y values of the corresponding edges are below[3] $\bar{y}$. From facts 5.6.2 and 5.6.3 we know that these intervals are nested and that their nesting order corresponds to the y order of their corresponding edges.

Let us recall that a regular, unconstrained, stabbing query traverses a path from the root to the leaf according to the query value $\bar{x}$. In each node an appropriate secondary list is considered and some intervals it contains may be reported. Due to the ordering of the lists, the reported intervals always form a contiguous range at the beginning of the list, which allows their efficient extraction.

We exploit the special properties of the companion intervals to handle the additional constraint in the stabbing query — that the corresponding edges are to fall below $\bar{y}$. Suppose that the search for $\bar{x}$ in W from the root traverses the path $u_h, u_{h-1}, \ldots, u_1, u_0$, where $u_\ell \in W$ is the node on level ℓ (hence, u_h is the root and u_0 is a leaf). Let S denote the set of the k intervals to be identified by the constrained stabbing query.

Lemma 5.8.2 *If for some $1 \leq j \leq h$, a list $L_i(u_j)$ in the node u_j contains intervals crossing the line $\bar{x}$, but not contained in S due to the constraint, then no list $L_{i'}(u_\ell)$ in node u_ℓ above node u_j ($\ell > j$) can contain intervals in S. Moreover, $L_i(u_j) \cap S$ form a contiguous range within $L_i(u_j)$.*

Proof: The proof follows from facts 5.6.2 and 5.6.3. The order of the interval endpoints in each secondary list corresponds to the relative vertical position of the corresponding edges in the Cartesian tree, as already noted previously. Therefore, if a secondary list contains intervals in S, then these intervals form a contiguous range within the list. The upper bound of this range is determined by the coordinate $\bar{x}$ of the stabbing query, just like in the non-constrained version. The lower bound may be determined by the constraint that the vertical position of the corresponding edge must be below $\bar{y}$.

We show by contradiction that if this lower bound does not coincide with the beginning of the list for some list $L_i(u_j)$, then there are no more intervals in S

[3]By "below $\bar{y}$" we mean that at least one of the two endpoints of the edge has a y value which is less than $\bar{y}$.

stored in nodes u_ℓ above u_j ($\ell > j$). Suppose that the contrary holds and that some interval in $L_i(u_j)$ contains $\bar{x}$ and corresponds to an edge e, which is *above* $\bar{y}$, while some other interval stored in $L_{i'}(u_\ell)$ also contains $\bar{x}$, but corresponds to an edge e', which is *below* $\bar{y}$. Since both of these intervals contain $\bar{x}$, they must be nested by Fact 5.6.2. But e is above $\bar{y}$, so it must also be above e', since e' is below $\bar{y}$. From Fact 5.6.3, this means that the interval for e' must be nested within the interval for e. So in particular, the interval for e must contain the same boundary as e' ($b_{i'}$) and be stored in u_ℓ or in its ancestor, according to the rules governing interval trees. Thus, we have reached a contradiction. □

We exploit Lemma 5.8.2 in the search algorithm as follows. We consider the path $u_0, u_1, \ldots, u_{h-1}, u_h$ of nodes visited during the unconstrained stabbing query $\bar{x}$ bottom-up, from a leaf to the root, and the appropriate secondary lists. Each time we check if the edge corresponding to the first interval on the list, $L_{i'}(u_{j'})$, satisfies the constraint of being below $\bar{y}$ (we can do this in $O(1)$ time). As long as it does, we continue as for the unconstrained query. When this condition is not met, we have identified node u_j from Lemma 5.8.2. At this point we use the property that the list order matches the y order of the edges to identify the boundaries of the range to report by Lemma 5.8.2 and we report the f items in this range. This is equivalent to a one-dimensional range query on $L_i(u_j)$ and takes $O(\log |L_i(u_j)| + f)$ time. We then terminate the search, since we know that there are no more intervals in S stored further up along the path to the root by Lemma 5.8.2. Since we examine the first entries in the lists for the other nodes ($\neq u_j$), we obtain a total cost of $O(\log n + k)$ time. This is the same cost as for a standard unconstrained query plus an additional constant cost in each node along the considered path and plus the search cost of the one-dimensional range query.

We are left with the problem of locating the node $\langle \hat{x}, \hat{y} \rangle$ in T that will become the parent of the new node $\langle \bar{x}, \bar{y} \rangle$ in T' after its insertion into the tree (action 1). We can assume that this node exists ($\bar{y}$ is not the largest y value in T'), otherwise this step does not need to be performed (it is easy to check which case holds for a given insertion).

We recall from Section 5.6.2 that $\langle \hat{x}, \hat{y} \rangle$ is either the rightmost point in the top-left quadrant delimited by $\langle \bar{x}, \bar{y} \rangle$ or the leftmost point in the top-right quadrant delimited by $\langle \bar{x}, \bar{y} \rangle$, whichever is lower. Such a point can easily be located in logarithmic time, if we maintain an additional data structure for storing points in T, such as a priority tree [59]. However, we can also use our existing companion interval tree to locate

the node in logarithmic node, without the need for any additional data structure. We explain how this can be done in Section 5.8.2.

Lemma 5.8.3 *The searching cost (actions* 1 *and* 2*) using* T *and* W *is* $O(\log n + k)$ *time.*

Locating the insertion point

In order to locate $\langle \hat{x}, \hat{y} \rangle$, we can use some information obtained from executing the constrained stabbing query. Namely, consider the edge e which is the lowest edge stabbed by $\bar{x}$ above $\bar{y}$. If this edge exists then it is easy to identify it during the constrained stabbing query, as it is located in $L_i(u_j)$, adjacent to the range returned by the one-dimensional range query performed on that list. This follows directly from the definition of $L_i(u_j)$.

If e exists then both of its endpoints lie above $\bar{y}$, so they can they be considered candidates for $\langle \hat{x}, \hat{y} \rangle$. No parents of these endpoints can be closer to $\bar{x}$ than the endpoints themselves, or else they would have to lie above e, which contradicts the basic property expressed in Fact 5.6.1. So any other candidate for $\langle \hat{x}, \hat{y} \rangle$ would have to be contained in the subtrees of the endpoints of e. There are no nodes in the rectangle whose corners are $\langle \bar{x}, \bar{y} \rangle$ and the higher endpoint of e, or else they would violate either the rules of tree construction or the definition of e. Therefore, we need to search for $\langle \hat{x}, \hat{y} \rangle$ in the subtree of the lower endpoint of e. Suppose, without loss of generality, that the lower one is the left endpoint. Then, this subtree does not contain any edges which cross $\bar{x}$ above y, due to the definition of e. So we need to search this subtree for the largest x value above $\bar{y}$. For that, we need only search the rightmost branch of this subtree. Note that all of the companion intervals of the edges on this branch must be nested within the companion interval of e. See Figure 5.7 for an illustration.

Now let us consider the case in which e does not exist. In this case, the nodes which have $y > \bar{y}$ must either all lie to the left of $\bar{x}$ or to the right of it, otherwise there would be edges crossing $\bar{x}$ above $\bar{y}$, in particular the lowest one, e. Suppose, without loss of generality, that they all lie to the left. In this case, the problem boils down to finding the node $\langle x, y \rangle$ having $y > \bar{y}$ with the maximum x coordinate. This node is located on the rightmost branch of T.

We can summarize these observations in the following way:

Fact 5.8.4 *The sought parent node* $\langle \hat{x}, \hat{y} \rangle$ *lies on a path of edges in* T*. This path is monotonous with respect to both* x *and* y*. If the edge* e *exists, then this path originates*

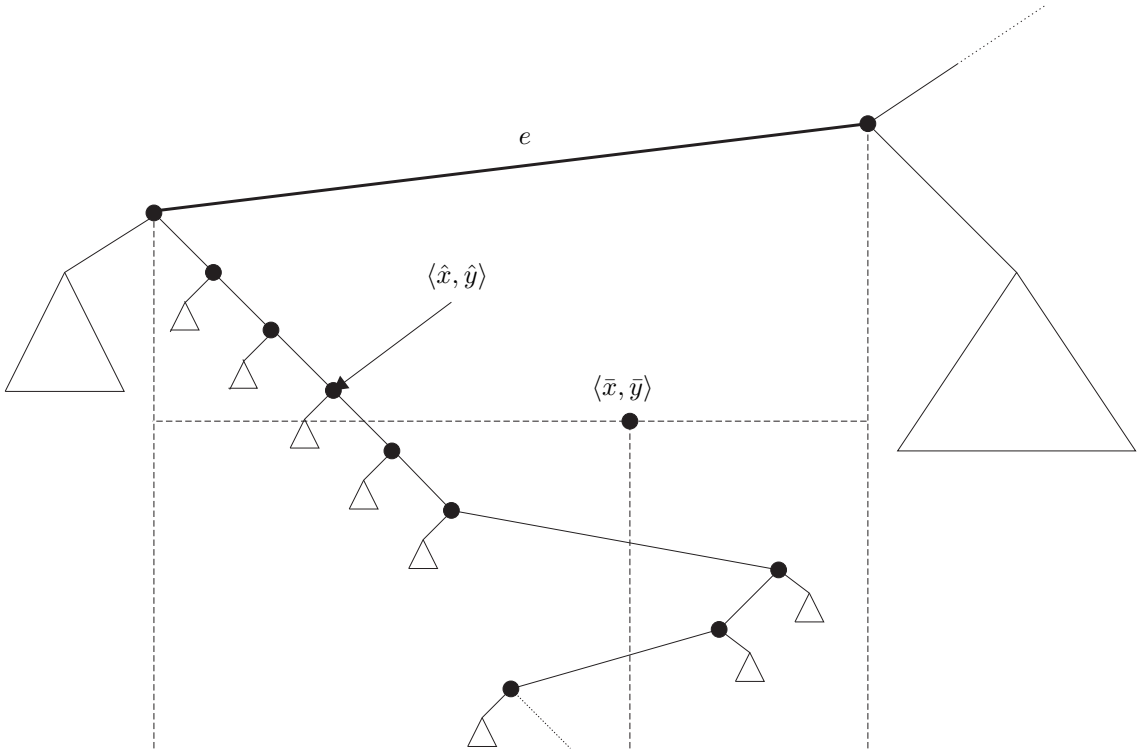

Figure 5.7: The edge e and the rightmost branch of its lower endpoint from which the parent of $\langle \bar{x}, \bar{y} \rangle$, $\langle \hat{x}, \hat{y} \rangle$ must be chosen.

from the lower endpoint of e and contains edges whose companion intervals are nested within the companion interval of e. Otherwise, this path is either the leftmost or the rightmost branch of the tree T. Identifying the edge e as well as verifying its existence can be performed during the constrained stabbing query (action 2), without increasing the complexity of this procedure.

We now show how to use Fact 5.8.4 to find the parent node $\langle \hat{x}, \hat{y} \rangle$, given e.

We first consider the case in which e does not exist. In this case the companion interval of the edge containing $\langle \hat{x}, \hat{y} \rangle$ is located on the leftmost or rightmost branch of T. This interval can not be nested within any other interval, or else it would not belong to such a branch. Therefore, if it is stored in some list $L_i(u)$ then it must be stored as the last element of this list. This observation is crucial to the logarithmic complexity of the procedure, as we only need to spend a constant time examining a list.

We start by examining the last intervals stored in the root of the companion tree. These intervals can not be nested within one another (but the same interval may occur as a last interval in more than one list.) We consider both endpoints of the edges corresponding to the intervals. These edges are a selection of the edges lying within the leftmost and rightmost branch of T, sorted according to x. We consider just the edges belonging to the branch we are interested in. These edges are also sorted according to y. If there is an edge which crosses the line $\bar{y}$ then this is the edge we are looking for — its higher endpoint is $\langle \hat{x}, \hat{y} \rangle$. Otherwise, we use the y values of the other edges on the appropriate branch as search tree keys to navigate down in the companion interval tree along a unique path. (Remember that the y order matches the x order along such a branch, so we can do that.). If all of the edges in the node belong to the wrong branch (leftmost resp. rightmost) then we simply navigate to the most extreme subtree (rightmost resp. leftmost). It may happen that there is no edge in the branch which crosses the line $\bar{y}$. In this case $\langle \hat{x}, \hat{y} \rangle$ is the lower endpoint of the lowest edge on the branch.

In the case in which e exists, we may use a similar procedure, except we start the search from the node u_j containing e. The key observation here is that the sought interval is nested within the interval for e, so it can not be stored above u_j in the interval tree. Therefore, we just need to search u_j and its subtree. Again we can observe that in the descendants of u_j the sought interval must occur as a last interval in its list. That is because we are seeking only intervals directly nested within e (without any intervals nested in between the two) and any intervals

encompassing e as well as the sought edge are stored in u_j or above. In the node u_j itself we consider the items directly preceding e in all the lists that e belongs to. We can do that in constant time, since we have already located e and we maintain a thread of all occurrences of the same edge.

5.8.3 Restructuring Cost

We are left with the implementation of action 4. With reference to the general case depicted in Figure 5.3, we can observe that the shrinking of edges involves a rearrangement at their endpoints. For example, u_2 is detached from u_3 and attached to u_5 and u_4 is detached from u_5 and attached to u_8. In general, it is just an implementation detail how to reconnect the Cartesian tree in $O(1+k)$ time, which becomes its restructuring cost. What we focus on next is how to maintain the companion intervals in W. We need the crucial properties below to perform this task efficiently.

Fact 5.8.5 *Let $(x, x') \in I(T)$ be an interval which shrinks and becomes (x, x''), where $x < x'' < x'$. (Notice that one endpoint always remains unchanged due to shrinking. Here we assume without loss of generality that it is the left endpoint.) Let $u \in W$ be the node whose secondary list(s) contain(s) interval (x, x').*

1. *The shrinking of (x, x') does not introduce any new endpoints to be stored in W. In other words, a leaf of W already stores x''.*
2. *If (x, x'') should be relocated to another node, v, then v is a descendant of u and (x, x'') becomes the first interval in the suitable secondary list(s) of v.*

A similar result holds when the right endpoint remains unchanged and the shrunk interval is (x'', x').

We need property (1) of Lemma 5.8.5 to guarantee that no restructuring of the tree shape of W is needed because of a shrinking. Otherwise, the restructuring cost would increase by a factor of $O(\log n)$. Fortunately, this is not the case and we only need to relocate (x, x') into $O(1)$ secondary lists of W as (x, x'') (or (x'', x')). To this end, we need property (2) of Lemma 5.8.5. Relocation moves the interval downward and requires just $O(1)$ time per node since we need to insert the interval at the beginning of $O(1)$ secondary lists.

Consequently, we perform the following. Let us consider the path of nodes $u_h, u_{h-1}, \ldots, u_1, u_0$ traversed for identifying the edges to shrink, as described in

Section 5.8.2. For each $\ell = h, h-1, \ldots, 1$, if node u_ℓ contains f_ℓ intervals to shrink, say $e_1, e_2, \ldots, e_{f_\ell}$ (already given in order of their left endpoint), we execute the following steps for $j = f_\ell, f_\ell - 1, \ldots, 1$:

1. Let $e_j = (x, x')$ be the interval shrinking. We assume without loss of generality that its left endpoint remains unchanged and the shrinking yields (x, x''). For each secondary list $L_i(u_\ell)$ that contains e_j, let b_i be the boundary associated with the list: if $x'' < b_i$, remove the interval from the list; otherwise leave the intervals as is. Note that this does not change the order inside the list.

2. If at least one copy of (x, x'') remains in the secondary lists of u_ℓ, stop processing the interval.

3. Otherwise, find the descendant v of u_ℓ, which is the new location of (x, x''). Insert (x, x'') into the secondary lists of v as needed, and create a thread of these copies.

The correctness of the method follows from Lemma 5.8.5. As for the complexity, each relocation may cost $O(\log n)$ time, and there are as many as $O(n + \mathscr{H}(T))$ relocations by Theorem 5.7.1, yielding a cost of $O((n+\mathscr{H}(T)) \times \log n) = O(n \log^2 n)$ in the worst case. This is not yet the bound we claimed in the introduction.

We now complete this section by showing that the amortized cost of the relocations in W is $O(n \log n)$ as claimed. We focus on what happens to an interval $e \in W$ and base our analysis on the intuition that the downward relocation path of an interval e is bound by the height of the tree. However, we must also consider that nodes of W may split, so the formal argument uses credits for e. When e is first inserted into W, it is assigned h credits where h is the current height of W. Since $h = O(\log n)$, these credits are paid for by the insertion operation. Moreover, a credit is assigned to e when the node containing it is split in W. According to Lemma 5.8.1, the cost of splitting a node, proportional to the number of intervals it contains, can be amortized. Therefore, also the additional credit assigned to each interval in the node being split, can be amortized. Credits assigned to e are used to pay for the shrinking operations affecting e. It suffices to prove the following lemma:

Lemma 5.8.6 *At any given time, let h be the level of the node u_ℓ currently containing e in some of its secondary lists. Then, the interval e has at least h credits assigned to it.*

Proof: We prove Lemma 5.8.6 by induction with respect to the operations affecting the height of the node storing e or the number of credits assigned to e.

When e is first inserted into W, then it has h credits by definition.

Suppose a relocation of e moves e to the node $u_{\ell'}$ at height $h' = h - \Delta_h$. This operation is paid for by credits associated with e. The operation cost is proportional to Δ_h and e loses Δ_h credits. However, also the height of e decreases by Δ_h, so e now has at least h' credits and the claim holds.

The credit balance of e can also be affected by a split operation. When a node splits, one of its boundaries — and hence also the list of intervals associated with that boundary — is relocated to the parent of the node. This operation may increase the height of an interval contained in the node being split by one. However, it also assigns one credit to each interval contained in the splitting node. So for each interval e contained in node u_ℓ at height h, if e has at least h credits before the split of u_ℓ, then after the split of u_ℓ, the height of e if $h' \leq h + 1$ and the number or credits assigned to e is at least $h + 1$ and so the claim holds. □

Theorem 5.8.7 *Given a Cartesian tree T, we can maintain it under a sequence of n insertions of points using a modified weight balanced B-tree W as an auxiliary data structure, with an amortized cost of $O(\log n)$ time per insertion. The amortized cost of restructuring T is $O(1 + \mathscr{H}(T)/n)$ per insertion, where $\mathscr{H}(T) = O(n \log n)$ is the missing entropy of T.*

5.9 Implementing deletions

In addition to insertions, our structure supports weak deletions, that is marking nodes as deleted. Nodes marked as deleted are present in the Cartesian tree until the overall number of marked nodes in the tree reaches a specified constant fraction α of all of the nodes in the tree. Once that happens, global rebuilding [68] is performed. Global rebuilding means that the Cartesian tree as well as the companion interval tree are rebuilt from scratch using only the nodes which have not been marked as deleted.

The cost of a weak delete operations is $O(\log n)$. It is also inversely proportional to α, but α is a constant. This cost is not used at the time a node is marked as deleted, but at the time of the next global rebuilding. The next global rebuilding occurs after $O(\alpha n)$ weak deletions have occurred and its cost is $O(n \log n)$. This cost includes the cost of building the Cartesian tree, building the companion interval

tree and accounting for the fact that the potential function used for amortizing the insertion cost can decrease even by as much as $O(n \log n)$ as a result of the deletions.

5.10 Conclusions

In this chapter we showed how the rigid Cartesian tree structure can be turned into a dynamic one. The contents of the Cartesian tree uniquely define its shape which renders the structure useful for a variety of applications, but also means that a single update operation can affect the entire structure.

We base the work on the observation that costly insert operations can not happen repeatedly and their cost can be amortized.

We first proved that the number of Cartesian tree elements affected by insertion operations is logarithmic if amortized. We then exploited this fact to create a dynamic version of the Cartesian tree.

The dynamic version of the Cartesian tree is composed of the original tree augmented with a structure which we call the *companion interval tree*. The companion interval tree has the same space complexity as the original tree and enables fast insert operations on the Cartesian tree by providing access to the Cartesian tree elements which are affected by the update.

Our result is the first known dynamic version of the Cartesian tree structure.

Conclusions

In the work we introduced the concept of rank-sensitive data structures — data structures which associate a rank with each of the stored elements and when answering a query return only the highest-ranking results sorted according to rank, where the number of results to return is a parameter given at query time. Such data structures are very much needed and the need for them is increasing still as the datasets we deal with are increasing in size dramatically from year to year and it is more and more difficult to deal with excess of data, especially in such fields as information retrieval and computational biology.

We explored several approaches to designing rank-sensitive data structures while discussing the difficulties and trade-offs this incurs. We presented a way to augment suffix trees in order to make them rank-sensitive. Then, we presented a general framework for adding rank-sensitivity to any data structures which are in the form of a tree and the items satisfying a query are stored in the tree's leaves and form $O(\text{polylog}(n))$ intervals of consecutive (in infix order) leaves (e.g. suffix trees, range trees). We presented both static and dynamic versions of the latter with different spacial and temporal complexities.

Finally we presented the first fully dynamic Cartesian tree structure. Cartesian trees have many applications in priority queue implementations, range searching, range maximum queries, and others and they also intrinsically store items in sorted order according to one dimension and partially sorted in according to the other. Making Cartesian trees dynamic could prove an important first step in improving the presented rank-sensitive data structures and designing new ones, as well as solving other algorithmic problems.

Future work We showed how certain output sensitive structures can be made rank-sensitive. The trade-off was an increased space complexity and in some cases search time. Some of the solutions could not be used in the dynamic setting. Future work should include reducing the time and space overheads incurred by adding rank-

sensitivity, finding better solutions in the dynamic setting, as well as extending the class of structures considered.

We also introduced a dynamic version of the Cartesian tree. This new structure could prove useful in the further work of rank-sensitivity, but could also have other applications which should be investigated. For example, Cartesian trees are key in the $O(1)$ solution to the range maximum query problem, but only in the static setting, so one could investigate if our dynamic version could be used for improving the bounds for range maximum query in dynamic data structures.

Index

Bibliography

[1] Pankaj K. Agarwal, Lars Arge, and Ke Yi. An optimal dynamic interval stabbing-max data structure? In *SODA '05: Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 803–812, Philadelphia, PA, USA, 2005. Society for Industrial and Applied Mathematics.

[2] Pankaj K. Agarwal and Jeff Erickson. Geometric range searching and its relatives. *Advances in Discrete and Computational Geometry*, 23:1–56, 1999.

[3] Alfred V. Aho, John E. Hopcroft, Jeffrey Ullman, J. D. Ullman, and J. E. Hopcroft. *Data Structures and Algorithms*. Addison-Wesley Longman Publishing Co., Inc., 1983.

[4] Miklós Ajtai, Michael L. Fredman, and János Komlós. Hash functions for priority queues. *Information and Control*, 63(3):217–225, December 1984.

[5] Stephen Alstrup, Gerth Stolting Brodal, and Theis Rauhe. New data structures for orthogonal range searching. In IEEE, editor, *41st Annual Symposium on Foundations of Computer Science: proceedings: 12–14 November, 2000, Redondo Beach, California*, pages 198–207. IEEE Computer Society Press, 2000.

[6] Arne Andersson, Torben Hagerup, Stefan Nilsson, and Rajeev Raman. Sorting in linear time? *Journal of Computer and System Sciences*, 57(1):74–93, August 1998.

[7] Lars Arge and Jeffrey S. Vitter. Optimal external memory interval management. *SIAM Journal on Computing*, 32:1488–1508, 2003.

[8] Ricardo A. Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[9] Rudolf Bayer. Binary B-trees for virtual memory. In E. F. Codd and A. L. Dean, editors, *Proceedings of 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control, San Diego, California, November 11-12, 1971*. ACM.

[10] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.

[11] Paul Beame and Faith E. Fich. Optimal bounds for the predecessor problem. In ACM, editor, *Proceedings of the thirty-first annual ACM Symposium on Theory of Computing: Atlanta, Georgia, May 1–4, 1999*, pages 295–304, New York, NY, USA, 1999. ACM Press.

[12] Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In *LATIN '00: Proceedings of the 4th Latin American Symposium on Theoretical Informatics*, pages 88–94, London, UK, 2000. Springer-Verlag.

[13] Michael A. Bender, Martín Farach-Colton, Giridhar Pemmasani, Steven Skiena, and Pavel Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *Journal of Algorithms*, 57(2):75–94, 2005.

[14] Jon Louis Bentley. Multidimensional divide-and-conquer. *Communications of the ACM*, 23(4):214–229, 1980.

[15] Jon Louis Bentley and Jerome H. Friedman. Data structures for range searching. *ACM Computing Surveys*, 11(4):397–409, 1979.

[16] Iwona Bialynicka-Birula and Roberto Grossi. Rank-sensitive data structures. In Mariano P. Consens and Gonzalo Navarro, editors, *SPIRE*, volume 3772 of *Lecture Notes in Computer Science*, pages 79–90. Springer, 2005.

[17] Iwona Bialynicka-Birula and Roberto Grossi. Amortized rigidness in dynamic Cartesian trees. In Bruno Durand and Wolfgang Thomas, editors, *STACS*, volume 3884 of *Lecture Notes in Computer Science*, pages 80–91. Springer, 2006.

[18] Tim Bray, Jean Paoli, C. Michael Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (XML) 1.0 (fourth edition). World Wide Web Consortium, Recommendation REC-xml-20060816, August 2006.

[19] Graham Brightwell and Peter Winkler. Counting linear extensions is #P-complete. In *STOC '91: Proceedings of the twenty-third annual ACM symposium on Theory of computing*, pages 175–181, New York, NY, USA, 1991. ACM Press.

[20] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the Seventh International World-Wide Web Conference*, 1998.

[21] Bernard Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing*, 17(3):427–462, June 1988.

[22] Bernard Chazelle and Leonidas J. Guibas. Fractional cascading: A data structuring technique with geometric applications. In *Proceedings of the 12th Colloquium on Automata, Languages and Programming*, pages 90–100, London, UK, 1985. Springer-Verlag.

[23] M. T. Chen and Joel Seiferas. Efficient and elegant subword-tree construction. In Alberto Apostolico and Zvi Galil, editors, *Combinatorial Algorithms on Words*. Springer-Verlag New York, Inc., 1985.

[24] Richard Cole and Ramesh Hariharan. Dynamic LCA queries on trees. *SIAM Journal on Computing*, 34(4):894–923, 2005.

[25] Douglas Comer. Ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[26] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

[27] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwartzkopf. *Computational Geometry: Algorithms and Applications*. Springer, 1997.

[28] Delcher, Phillippy, Carlton, and Salzberg. Fast algorithms for large-scale genome alignment and comparison. *Nucleic Acids Research*, 30(11):2478–2483, June 2002.

[29] Arthur L. Delcher, Simon Kasif, Robert D. Fleischmann, Jeremy Peterson, Owen White, and Steven L. Salzberg. Alignment of whole genomes. *Nucleic Acids Research*, 27(11):2369–2376, 1999.

[30] Luc Devroye. On random Cartesian trees. *Random Structures and Algorithms*, 5(2):305–328, 1994.

[31] Luc Devroye, Wojciech Szpankowski, and Bonita Rais. A note on the height of suffix trees. *SIAM Journal on Computing*, 21(1):48–53, 1992.

[32] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, February 1989.

[33] Herbert Edelsbrunner. A new approach to rectangle intersections, part I. *International Journal Computer Mathematics*, 13:209–219, 1983.

[34] Martin Farach-Colton, Paolo Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *Journal of the ACM*, 47(6):987–1011, 2000.

[35] Paolo Ferragina and Roberto Grossi. The string B-tree: A new data structure for string search in external memory and its applications. *Journal of the ACM*, 46(2):236–280, 1999.

[36] Paolo Ferragina and S. Muthukrishnan. Efficient dynamic method-lookup for object oriented languages (extended abstract). In *ESA '96: Proceedings of the Fourth Annual European Symposium on Algorithms*, pages 107–120, London, UK, 1996. Springer-Verlag.

[37] Amos Fiat and Haim Kaplan. Making data structures confluently persistent. *Journal of Algorithms*, 48(1):16–58, 2003.

[38] Faith E. Fich. Class notes CSC 2429F: Dynamic data structures, 2003. Department of Computer Science, University of Toronto, Canada.

[39] Rudolf Fleischer. A simple balanced search tree with $O(1)$ worst-case update time. *International Journal of Foundations of Computer Science*, 7(2):137–149, 1996.

[40] Jean Franon, G. Viennot, and Jean Vuillemin. Description and analysis of an efficient priority queue representation. In *FOCS*, pages 1–7, 1978.

[41] Greg N. Frederickson. An optimal algorithm for selection in a min-heap. *Information and Computation*, 104(2):197–214, June 1993.

[42] Edward Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.

[43] Michael L. Fredman and Dan E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *Journal of Computer and System Sciences*, 48(3):533–551, June 1994.

[44] Harold N. Gabow, Jon Louis Bentley, and Robert E. Tarjan. Scaling and related techniques for geometry problems. In *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing*, pages 135–143, Washington, D.C., 1984.

[45] Roberto Grossi and Giuseppe Italiano. Suffix trees and their applications in string algorithms. In *Proceedings of the 1st South American Workshop on String Processing (WSP 1993)*, pages 57–76, September 1993.

[46] Dan Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge University Press, 1997.

[47] Dan Gusfield and Jens Stoye. Linear time algorithms for finding and representing all the tandem repeats in a string. *Journal of Computer and System Sciences*, 69(4):525–546, 2004.

[48] Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, 1984.

[49] Donald Hearn and M. Pauline Baker. *Computer Graphics with OpenGL*. Prentice Hall, third edition, 2003.

[50] Michael Höhl, Stefan Kurtz, and Enno Ohlebusch. Efficient multiple genome alignment. *Bioinformatics*, 18:S312–S320, 2002.

[51] Lars Kaderali and Alexander Schliep. Selecting signature oligonucleotides to identify organisms using dna arrays. *Bioinformatics*, 18(10):1340–1349, October 2002.

[52] Haim Kaplan. Persistent data structures. In Dinesh P. Mehta and Sartaj Sahni, editors, *Handbook on Data Structures and Applications*. CRC Press, 2005.

[53] Haim Kaplan, Eyal Molad, and Robert E. Tarjan. Dynamic rectangular intersection with priorities. In *STOC '03: Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, pages 639–648. ACM Press, 2003.

[54] Kitsios, Makris, Sioutas, Tsakalidis, Tsaknakis, and Vassiliadis. 2-D spatial indexing scheme in optimal time. In *ADBIS: East European Symposium on Advances in Databases and Information Systems*. LNCS, 2000.

[55] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5):604–632, September 1999.

[56] George Lagogiannis, Christos Makris, Yannis Panagis, Spyros Sioutas, and Kostas Tsichlas. New dynamic balanced search trees with worst-case constant update time. *Journal of Automata, Languages and Combinatorics*, 8(4):607–632, 2003.

[57] Ronny Lempel and Shlomo Moran. SALSA: the stochastic approach for link-structure analysis. *ACM Transactions on Information Systems*, 19(2):131–160, 2001.

[58] Edward M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[59] Edward M. McCreight. Priority search trees. *SIAM Journal on Computing*, 14(2):257–276, 1985.

[60] Kurt Mehlhorn. A new data structure for representing sorted lists. In *WG '80: Proceedings of the International Workshop on Graphtheoretic Concepts in Computer Science*, pages 90–112, London, UK, 1981. Springer-Verlag.

[61] Kurt Mehlhorn. *Data structures and algorithms 3: multi-dimensional searching and computational geometry*. Springer-Verlag New York, Inc., New York, NY, USA, 1984.

[62] Kurt Mehlhorn and Stefan Näher. Dynamic fractional cascading. *Algorithmica*, 5(2):215–241, 1990.

[63] Donald R. Morrison. PATRICIA – practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.

[64] Christian Worm Mortensen. Fully-dynamic two dimensional orthogonal range and line segment intersection reporting in logarithmic time. In *Proceedings of the fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA-03)*, pages 618–627, New York, January 2003. ACM Press.

[65] S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *SODA '02: Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 657–666. Society for Industrial and Applied Mathematics, 2002.

[66] Wendy Myrvold and Frank Ruskey. Ranking and unranking permutations in linear time. *Information Processing Letters*, 79(6):281–284, September 2001.

[67] Jürg Nievergelt and Edward M. Reingold. Binary search trees of bounded balance. *SIAM Journal on Computing*, 2:33–43, 1973.

[68] Mark H. Overmars. *The Design of Dynamic Data Structures*, volume 156 of *Lecture Notes in Computer Science*. Springer-Verlag, 1983.

[69] Mark H. Overmars. Efficient data structures for range searching on a grid. *Journal of Algorithms*, 9(2):254–275, 1988.

[70] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. *Technical report, Stanford University, Stanford, CA*, 1998.

[71] W. R. Pearson. Rapid and sensitive sequence comparison with FASTP and FASTA. *Methods Enzymol*, 183:63–98, 1990.

[72] Rajeev Raman. *Eliminating amortization: on data structures with guaranteed response time*. PhD thesis, University of Rochester, Rochester, NY, USA, 1993.

[73] Raimund Seidel and Cecilia R. Aragon. Randomized search trees. *Algorithmica*, 16(4/5):464–497, 1996.

[74] Qingmin Shi and Joseph JáJá. Fast algorithms for 3-D dominance reporting and counting. *International Journal of Foundations of Computer Science*, 15(4):673–684, 2004.

[75] Daniel D. Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983.

[76] Alexander Stepanov and Meng Lee. The standard template library. Technical report, 1994.

[77] C. J. Stephenson. New methods for dynamic storage allocation (fast fits). In *SOSP '83: Proceedings of the ninth ACM symposium on Operating systems principles*, pages 30–32, New York, NY, USA, 1983. ACM Press.

[78] Mikkel Thorup. On AC^0 implementations of fusion trees and atomic heaps. In *Proceedings of the fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA-03)*, pages 699–707, New York, January 2003. ACM Press.

[79] Mikkel Thorup. Space efficient dynamic stabbing with fast queries. In *STOC '03: Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, pages 649–658, New York, NY, USA, 2003. ACM Press.

[80] Esko Ukkonen. Constructing suffix trees on-line in linear time. In *Proceedings of the IFIP 12th World Computer Congress on Algorithms, Software, Architecture - Information Processing '92, Volume 1*, pages 484–492, Amsterdam, The Netherlands, The Netherlands, 1992. North-Holland Publishing Co.

[81] Esko Ukkonen. Approximate string-matching over suffix trees. In *CPM '93: Proceedings of the 4th Annual Symposium on Combinatorial Pattern Matching*, pages 228–242, London, UK, 1993. Springer-Verlag.

[82] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14:249–260, 1995.

[83] Jeffrey Scott Vitter. External memory algorithms and data structures: dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, June 2001.

[84] Jean Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309–315, 1978.

[85] Jean Vuillemin. A unifying look at data structures. *Communications of the ACM*, 23(4):229–239, April 1980.

[86] Joe H. Ward. Hierarchical grouping to optimize an objective function. *Journal of the American Statistical Association*, 58:236–244, 1963.

[87] Peter Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[88] Eric W. Weisstein. Linear extension, 2005. From MathWorld–A Wolfram Web Resource. `http://mathworld.wolfram.com/LinearExtension.html`.

[89] Dan E. Willard and George S. Lueker. Adding range restriction capability to dynamic data structures. *Journal of the ACM*, 32(3):597–617, July 1985.

[90] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing gigabytes (2nd edition): compressing and indexing documents and images*. Morgan Kaufmann Publishers Inc., 1999.

[91] Oren Zamir and Oren Etzioni. Grouper: a dynamic clustering interface to web search results. In *WWW '99: Proceeding of the eighth international conference on World Wide Web*, pages 1361–1374, New York, NY, USA, 1999. Elsevier North-Holland, Inc.

[92] Li Zhao. `http://blog.lizhao.net/2005/01/c-implementation-of-ukkonens-suffix.html`, 2005.

[93] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38(2), 2006.