Data Structures

Lecture Notes for
CS 3110: Design and Analysis of Algorithms

Norbert Zeh

Faculty of Computer Science, Dalhousie University,
6050 University Ave, Halifax, NS B3H 2Y5, Canada

nzeh@cs.dal.ca

July 8, 2014

Contents

Chapter 1
(a, b)-Trees

In this chapter, we discuss a rather elegant tree structure for representing sorted data: (a, b)-trees. It is in spirit the same as a red-black tree or AVL-tree, that is, yet another balanced search tree. However, it is not a *binary* search tree, whose height is kept logarithmic by clever rotations; its rebalancing rules are much more transparent, which is why I hope that you feel more comfortable arguing about this structure than about red-black trees.

The discussion of (a, b)-trees is divided into different subsections. In Section 1.1, we define what (a, b)-trees are and prove a number of useful properties, including that their height is $O(\lg n)$, as long as a and b are constants. Details of how (a, b)-trees are represented using standard programming language constructs are provided in Section 1.2. In Section 1.3, we argue that a number of query operations can be performed in logarithmic time on (a, b)-trees, including searching for an element. Finally, in Section 1.4, we discuss how to insert and delete elements into and from an (a, b)-tree.

1.1 Definition

As binary search trees, (a, b)-trees are rooted trees. For a rooted tree T and a node v in T, we denote the subtree rooted at v by T_v. The nodes in T_v are the ***descendants*** of v; we use $Desc(v)$ to denote this set. The data items stored at the leaves of T_v are denoted by $Items(v)$. The keys of these items are denoted by $Keys(v)$.

Note the fine distinction we make between keys and data items. If there was no such distinction, search trees would be rather useless: If we search for element 14 and simply return it if we find it, we know little more than before. The only additional information we have gained is that element 14 is indeed in our set. So the point is that you should think about the items we store in the dictionary as a record, much like in a database; the key of an item is just one of the pieces of information stored in the record. For example, you may think about implementing Dalhousie's banner system. Then the elements we store in our database—that is, in our search tree—are records storing different pieces of information about each student. When we search for a student's record, we may locate this record, for example, using the student's banner ID as the search key; but the information we are interested in may be the student's transcript, email address, etc. So, by locating the record, we have gained more information than we had before the search.

Having said that there is a distinction between keys and elements, we will use our search tree to store numbers; the key of a number is the number itself. This is to keep the discussion simple. However, you should keep in mind that a data item and its key are usually two different things.

An (a, b)-tree is now defined as follows:

Definition 1.1 *For two integers $2 \leq a < b$, an* **(a, b)-tree** *is a rooted tree with the following properties:*

(AB1) The leaves are all at the same level (distance from the root).

(AB2) The data items are stored at the leaves, sorted from left to right.

(AB3) Every internal node that is not the root has between a and b children.

(AB4) If the root is not a leaf, it has between 2 and b children.

(AB5) Every node v stores a **key** *$key(v)$. For a leaf, $key(v)$ is the key of the data item associated with this leaf. For an internal node, $key(v) = \min(Keys(v))$.*

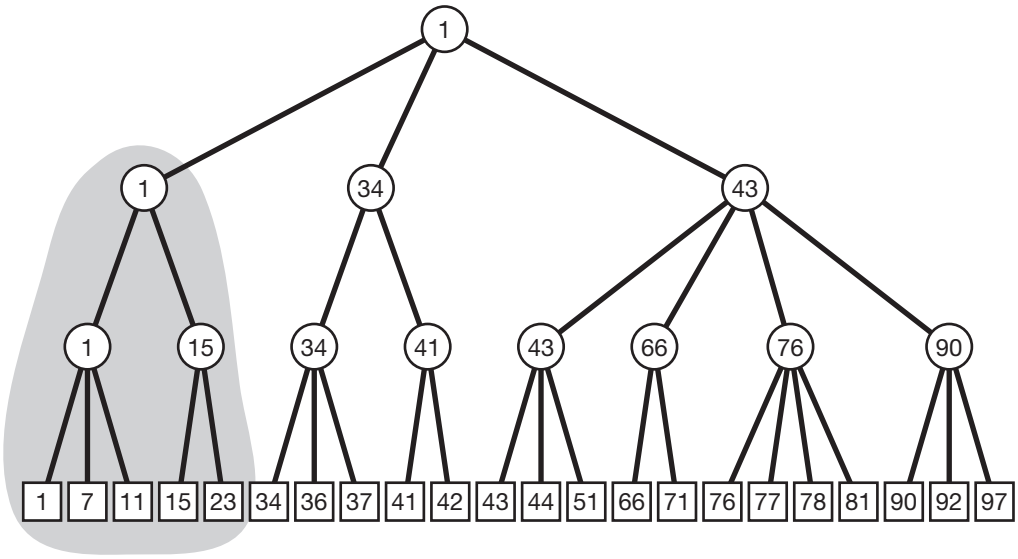


Figure 1.1. A $(2, 4)$-tree.

An example of a $(2, 4)$-tree is shown in Figure 1.1. The first two natural questions we would ask about an (a, b)-tree is what its size is if it stores n data items, and what its height is. For, as in any search tree, a search operation will traverse a path from the root to a leaf; that is, the height has a significant impact on the running time of a search operation on an (a, b)-tree. The following two lemmas give favourable answers to these questions.

Lemma 1.1 *An (a, b)-tree storing n items has height between $\log_b n$ and $\log_a(n/2) + 1$.*

Proof. Assuming that the height of the tree is h, we prove below that the number, n, of data items stored in the tree is between $2a^{h-1}$ and b^h. Using elementary arithmetic, we obtain the desired bounds on h from this claim:

$$n \leq b^h$$
$$\log_b n \leq h$$

and

$$\begin{aligned} n &\geq 2a^{h-1} \\ \frac{n}{2} &\geq a^{h-1} \\ \log_a \frac{n}{2} &\geq h-1 \\ \log_a \frac{n}{2} + 1 &\geq h. \end{aligned}$$

We have to prove the claimed bounds on the number of items as a function of h. First we prove the upper bound:

We use induction on h to prove that there are at most b^h leaves in an (a,b)-tree of height h. An (a,b)-tree of height 0 has only one node, which is at the same time a leaf and the root; that is, this tree has $1 = b^0$ leaves.

So assume that $h > 0$. Let r be the root of the tree, and let $v_1, v_2, \ldots, v_k$ be its children. For $1 \leq i \leq k$, T_{v_i} is an (a,b)-tree of height $h-1$. By the inductive hypothesis, this implies that T_{v_i} has at most b^{h-1} leaves. Every leaf of T is a leaf of some tree T_{v_i}. Hence, T has at most $k \cdot b^{h-1}$ leaves, which is at most $b \cdot b^{h-1} = b^h$, by Properties (AB3) and (AB4).

The proof of the lower bound is split into two parts. First we assume that the root satisfies Condition (AB3), which is stronger than Condition (AB4). Later we remove this assumption. We prove that, under the stronger assumption that the root has Property (AB3), the number of leaves is at least a^h. The proof uses induction on h again. For $h = 0$, the tree has $1 = a^0$ leaves. For $h > 0$, the root has $k \geq a$ children $v_1, v_2, \ldots, v_k$, each of which is the root of an (a,b)-tree of height $h-1$. By the inductive hypothesis, each such subtree has at least a^{h-1} leaves. Thus, the total number of leaves is at least $k \cdot a^{h-1} \geq a^h$.

If the root satisfies only Property (AB4), then a tree of height $h = 0$ has $1 \geq 2a^{-1}$ leaves. The inequality holds because $a \geq 2$. In a tree of height $h > 0$, the root has at least two children, which are the roots of (a,b)-trees of height $h-1$ and whose roots satisfy Property (AB3)! Hence, both subtrees have at least a^{h-1} leaves, which implies that the whole tree has at least $2a^{h-1}$ leaves. This is what we wanted to show.

Lemma 1.2 *An (a,b)-tree storing n items has less than $2n$ nodes.*

Proof. Again, we prove this claim by induction on the height of T. For a tree of height 0, this is true because there is only $n = 1 < 2 = 2n$ node. For a tree of height $h > 0$, the root has $k \geq 2$ children $v_1, v_2, \ldots, v_k$. For each subtree T_{v_i}, let n_i be the number of leaves in the subtree. By the inductive hypothesis, we have $|T_{v_i}| < 2n_i$. Hence,

$$\begin{aligned} |T| &= 1 + \sum_{i=1}^{k} |T_{v_i}| \\ &\leq 1 + \sum_{i=1}^{k} (2n_i - 1) \\ &= (1-k) + 2n \\ &< 2n. \end{aligned}$$

The last inequality holds because $k \geq 2$ and, hence, $1 - k < 0$.

It is not immediately clear that counting the number of nodes is sufficient to determine the space consumption of an (a, b)-tree, because it seems possible that a node has to store up to b pointers to its children; if b is not constant, this requires a non-constant amount of memory per node. In the next section, we discuss how to represent an (a, b)-tree using a constant amount of information per node. Together with Lemma 1.2, this implies that an (a, b)-tree uses linear space.

1.2 Representing (a, b)-Trees

Throughout these lecture notes, we will draw an (a, b)-tree as in Figure 1.1 on page 2. This is legitimate only if we understand perfectly how a tree drawn in this fashion is represented in a standard programming language, such as C. We discuss these representation issues in this section.

The external "handle" on an (a, b)-tree we are given is a pointer to the root node.[1] Every node v is represented using five pieces of information:

- Its ***key*** $key(v)$.
- Its ***degree*** $deg(v)$, which is the number of its children.
- A pointer $p(v)$ to its parent.
- A pointer $left(v)$ to its left sibling.
- A pointer $right(v)$ to its right sibling.
- A pointer $child(v)$ to its leftmost child.

If any of the nodes these pointers are supposed to point to do not exist, the pointers are NIL. For example, for the root of the tree, $p(v) = \text{NIL}$; and for a leaf, $child(v) = \text{NIL}$. Storing the degree of a node explicitly is not strictly necessary, but it makes update operations, which we discuss in Section 1.4, easier.

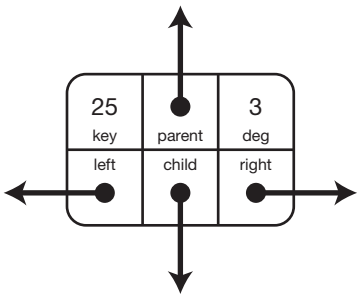


Figure 1.2. An (a, b)-tree node.

In a programming language like C, an (a, b)-tree node would therefore be expressed using a structure as the following:

```
struct ABTreeNode {
    KeyType key;
    int deg;
    struct ABTreeNode *p, *left, *right, *child;
};
```

This is visualized in Figure 1.2; Figure 1.3 shows the representation of part of the tree in Figure 1.1. We will see in the next sections how to perform basic search operations and how to modify (a, b)-trees efficiently using the representation described here.

We conclude this section with the following proposition, which is an immediate consequence of Lemma 1.2 and the fact that every node stores only a constant amount of information.

Proposition 1.1 *An (a, b)-tree storing n data items uses $O(n)$ space.*

[1] There may be other pointers into the tree we may want to store to speed-up certain computations; but a pointer to the root is the minimum information we can count on.

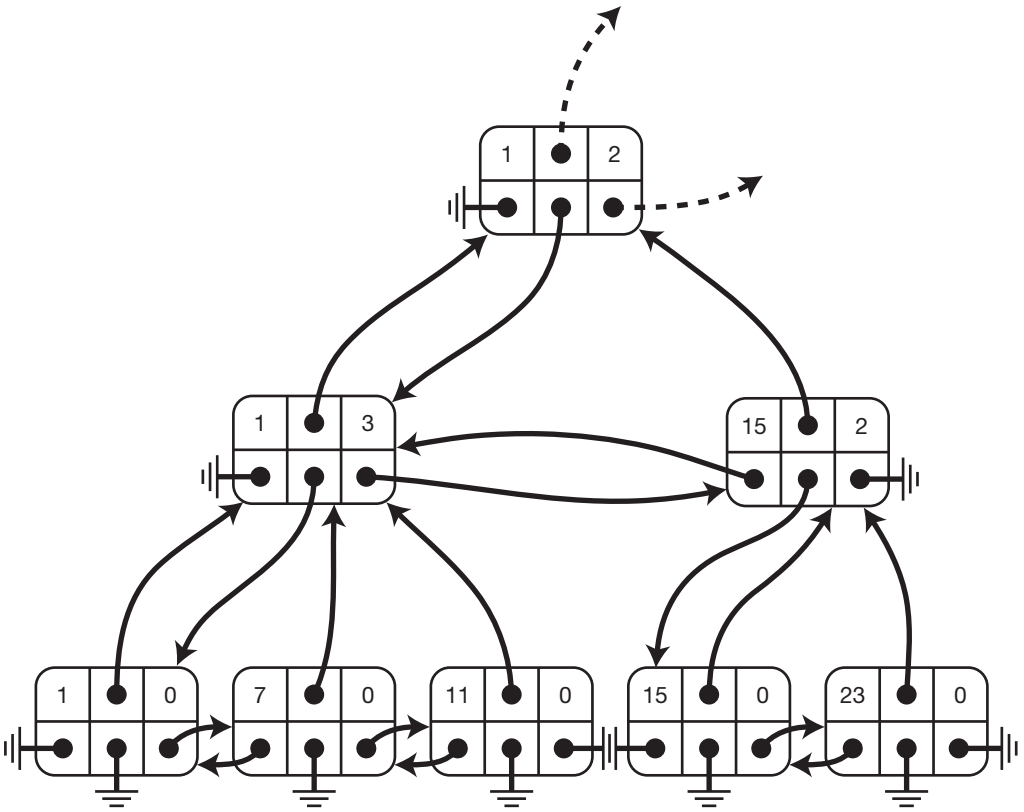


Figure 1.3. The pointer representation of the left subtree of the tree in Figure 1.1. NIL pointers are represented as grounded.

1.3 Searching (a, b)-Trees

An (a, b)-tree is a dictionary structure, whose purpose is to support different types of search queries efficiently. The most fundamental search operation is the FIND operation, which, given a key x, finds an entry with this key in T or reports that no such element exists. However, we often have to answer other types of queries as well. The ones we consider here are MINIMUM, MAXIMUM, PREDECESSOR, and SUCCESSOR queries. The first two return the minimum and maximum elements stored in T. The latter find the predecessor or successor of a given element x in the sorted sequence represented by T. More precisely, these operations return the following results if S is the set of elements currently stored in T:

MINIMUM(T): $\min(S)$

MAXIMUM(T): $\max(S)$

SUCCESSOR(T, x): Let $x_1, x_2, \ldots, x_n$ be the sorted order of the elements in S. If $x = x_i$, for some $i < n$, then SUCCESSOR(T, x) returns element x_{i+1}. If $x = x_n$ or $x \notin S$, then SUCCESSOR(T, x) returns NIL.

PREDECESSOR(T, x): Let $x_1, x_2, \ldots, x_n$ be the sorted order of the elements in S. If $x = x_i$, for some $i > 1$, then PREDECESSOR(T, x) returns element x_{i-1}. If $x = x_1$ or $x \notin S$, then PREDECESSOR(T, x) returns NIL.

We also consider an extension to the FIND operation which, given two keys l and u, finds and returns all data items with keys between l and u. This operation is call RANGE-QUERY. Operations FIND, MINIMUM, MAXIMUM, PREDECESSOR, and

SUCCESSOR are elementary in the sense that they return only a constant amount of information (a data item or NIL). The RANGE-QUERY operation may return anywhere between 0 and n data items, because none or all items may have keys in the given range. Thus, it is only natural that the running time of this operation will depend on the number of reported elements. The idea that the running time of an algorithm may depend on the size of the produced output is quite fundamental in algorithm design and is known as ***output sensitivity***.

1.3.1 The Find Operation

The FIND procedure, shown below, is given a key x as an argument. If an element with this key exists in T, FIND returns this element; otherwise it returns NIL to indicate that no such element exists.

```
FIND(T, x)
1  v ← root(T)
2  while v is not a leaf
3        do w ← child(v)
4           while right(w) ≠ NIL and key(right(w)) ≤ x
5                 do w ← right(w)
6           v ← w
7  if key(v) = x
8     then return v
9     else return NIL
```

Let us consider what the algorithm does when searching for element 77 in the tree in Figure 1.1. This process is illustrated in Figure 1.4. In the following discussion, we consider a node ***visited*** if it is assigned to v at some point during the procedure. We say that a node is ***inspected*** by procedure FIND if its key is examined, but it is never assigned to v.

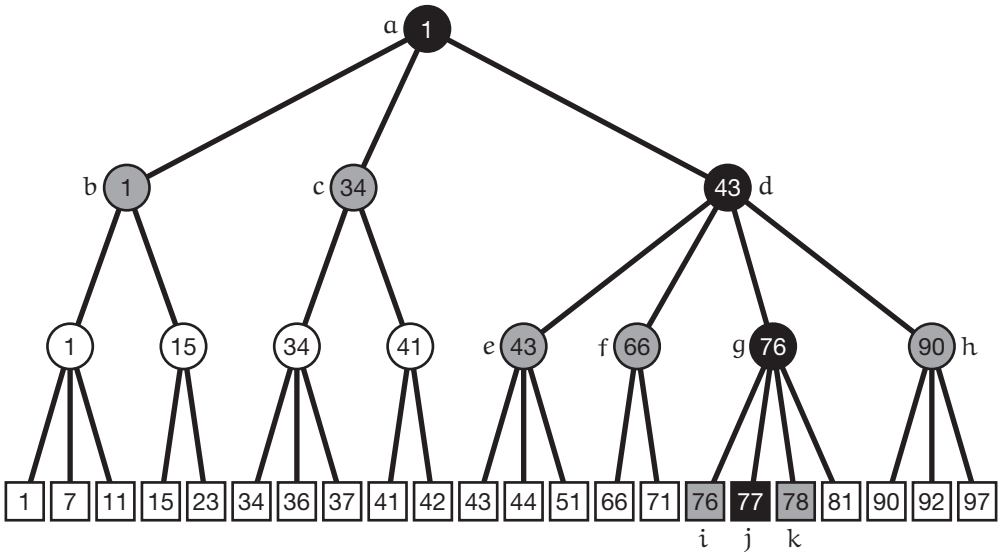


Figure 1.4. Searching for element 77 in the tree of Figure 1.1. Visited nodes are shown in black. Inspected nodes are shown in grey.

Our goal is obviously to locate the leaf j, which stores the element with key 77. In Line 1, we initialize v to point to the root of T, that is, to node a. Node

a is not a leaf; so we enter the outer while-loop and initialize w to point to the leftmost child of v, node b. Node c, which is node b's right sibling, has key 34, which is less than 77; so we enter the inner while-loop. The first iteration updates w to point to node c. Its right neighour, node d, has key 43, which is still less than 77; so we remain in the inner loop for another iteration and update w to point to node d. Node d does not have a right neighbour; so we exit the inner loop and update v to point to w's current value, that is, to node d. Node d is not a leaf; so we enter the next iteration of the outer loop. In Line 3, we initialize w to point to node d's leftmost child, node e. Its right sibling, node f, has key 66, which is less than 77; so we enter the inner loop, whose first iteration makes w point to node f. Its right sibling, node g, has key 76, which is still less than 77; so we execute another iteration of the inner loop, which updates w to point to node g. Node h, node g's right sibling, has key 90, which is greater than 77. So we exit the inner loop and update v to point to node g. Since node g is not a leaf, we enter another iteration of the outer loop, which inspects the keys of nodes i, j, and k. After exiting the inner loop, we update v to point to node j. At this point, v is a leaf, and we exit the outer loop. The test in Line 7 now determines that the key of node j is equal to 77, and we correctly return node j in answer to our query.

So what is the intuition behind the FIND procedure? We can easily formulate loop invariants for both of the loops; these loop invariants capture the intuition and also provide the basis for proving rigorously that the FIND procedure is correct. The loop invariant for the inner loop is the following:

> **Inner loop invariant:** For every child w' of v that is to the left of w, if $x \in \mathit{Keys}(w')$, then $x \in \mathit{Keys}(w)$.

Let us prove that the algorithm maintains the invariant.

Lemma 1.3 *The inner loop of procedure* FIND *maintains the inner loop invariant.*

Proof. We prove the claim by induction on the number of iterations that have already been executed. Before the first iteration, the base case, the invariant is trivially true because there is no child of v to the left of w: w is v's leftmost child. So assume that the claim holds for the current iteration. We have to prove that it holds for the next iteration. In order to do so, all we have to do is prove that $x \in \mathit{Keys}(w)$ implies that $x \in \mathit{Keys}(\mathit{right}(w))$. Indeed, this implies the loop invariant for $w' = w$. It also implies the loop invariant for $w' \neq w$ because, by the induction hypothesis, $x \in \mathit{Keys}(w')$ implies $x \in \mathit{Keys}(w)$ and, hence, by our claim, $x \in \mathit{Keys}(\mathit{right}(w))$. We have to prove our claim.

Assume that $x \in \mathit{Keys}(w)$. Since we execute the inner loop only if $\mathit{right}(w) \neq \text{NIL}$ and $\mathit{key}(\mathit{right}(w)) = \min(\mathit{Keys}(\mathit{right}(w))) \leq x$, there exists an element $y \in \mathit{Keys}(\mathit{right}(w))$ such that $y \leq x$. However, $x \in \mathit{Keys}(w)$, w is to the left of $\mathit{right}(w)$, and the keys are stored in sorted order, left to right, at the leaves of T. Therefore, we have $x \leq y$, that is, $x = y$; and $x \in \mathit{Keys}(\mathit{right}(w))$.

The outer loop maintains the following invariant:

> **Outer loop invariant:** If $x \in T$, then $x \in \mathit{Keys}(v)$.

Obviously, this is what we want to maintain because, once the search has advanced to a node v, it can never reach a node outside of T_v; that is, if we want to be sure to find x, it better be in T_v.

Lemma 1.4 *The outer loop of procedure* FIND *maintains the outer loop invariant.*

Proof. We prove the claim by induction on the number of executed iterations. The base case is the first iteration. Before this iteration, we have $v = \mathit{root}(T)$. Hence, if $x \in T$, we have $x \in \mathit{Keys}(v)$.

Now consider an iteration of the outer loop. By the induction hypothesis, $x \in T$ implies that $x \in T_v$. We have to prove that $x \in \mathit{Keys}(v)$ implies that $x \in \mathit{Keys}(w)$, for the node to which w points in Line 6. So assume that $x \in \mathit{Keys}(v)$. Then $x \in \mathit{Keys}(w')$, for some child w'. If w' is to the left of w, the inner loop invariant implies that $x \in \mathit{Keys}(w)$. If w' is to the right of w, we have $\mathit{key}(w') \leq x$. Since the keys of the children of v are non-decreasing from left to right, this implies that $\mathit{right}(w) \neq \text{NIL}$ and $\mathit{key}(\mathit{right}(w)) \leq x$. But this contradicts the fact that the inner loop has exited with the current value of w, because this value satisfies the loop condition. Hence, w' cannot be to the right of w, and we have $x \in \mathit{Keys}(w)$.

Lemma 1.4 immediately implies the correctness of the algorithm.

Lemma 1.5 *If $x \in T$, then procedure* FIND *returns a node that stores an element with key x. If $x \notin T$, the procedure returns* NIL.

Proof. By the outer loop invariant, $x \in T$ implies that $x \in \mathit{Keys}(v)$, for the leaf v at which the search terminates. But, since v is a leaf, we have $\mathit{Keys}(v) = \{\mathit{key}(v)\}$, that is, $x \in T$ implies that $x = \mathit{key}(v)$. Thus, the test in Line 7 is positive, and we report v.

If $x \notin T$, no matter which leaf we reach, we have $x \neq \mathit{key}(v)$. Thus, the test in Line 7 fails, and we return NIL.

Procedure FIND is not only correct, but also very efficient.

Lemma 1.6 *Procedure* FIND *terminates in $O(b \log_a n)$ time. For constant values of a and b, this is $O(\lg n)$.*

Proof. First we count the number of iterations of the outer loop. At the end of each such iteration, we update v to point to a child of the current node v, that is, v advances one level down the tree in every iteration. By Lemma 1.1, the height of the tree is $O(\log_a n)$, that is, there are $O(\log_a n)$ iterations of the outer loop. Excluding the inner loop, the cost of each iteration of the outer loop is constant. Hence, this part of the algorithm costs $O(\log_a n)$ time.

In the worst case, the inner loop iterates over all children of the current node v. Since there are at most b children, there are $O(b)$ iterations of the inner loop per iteration of the outer loop. Each iteration costs constant time. Hence, the total time spent in the inner loop is at most $O(b \log_a n)$. Summing the costs of the outer loop (excluding the inner loop) and the inner loop, we obtain the desired time bound for the FIND procedure.

1.3.2 Minimum and Maximum

The minimum and maximum elements stored in T are easy to identify because the elements are stored at the leaves, sorted left-to-right by increasing keys. In particular, the minimum element is stored at the leftmost leaf, and the maximum element is stored at the rightmost leaf. Since $child(v)$ points to the leftmost child of v, for every node v, locating the leftmost leaf amounts to following child pointers. This is what the following procedure does:

```
MINIMUM(T)
1  v ← root(T)
2  while v is not a leaf
3        do v ← child(v)
4  return v
```

Similarly, to find the rightmost leaf, we have to go from every node to its rightmost child, starting at the root and finishing when we reach a leaf. This is not quite as easy because a node v does no explicitly store a pointer to its rightmost child. However, by following its $child(v)$ pointer, we reach the leftmost child; then we can follow $right(w)$ pointers to identify the rightmost child of v, from where we continue our search down the tree. The following procedure implements this strategy:

```
MAXIMUM(T)
1  v ← root(T)
2  while v is not a leaf
3        do v ← child(v)
4           while right(v) ≠ NIL
5                 do v ← right(v)
6  return v
```

The correctness of these two procedures is obvious. The MINIMUM procedure spends constant time per level of the tree. By Lemma 1.1, the height of an (a, b)-tree is $O(\log_a n)$. Hence, the MINIMUM procedure takes $O(\log_a n)$ time. The MAXIMUM procedure spends $O(b)$ time per level because it inspects all children of every node on the rightmost path in the tree, and there are up to b children per node. Hence, the MAXIMUM procedure takes $O(b \log_a n)$ time. This is summarized in the following proposition.

Proposition 1.2 *The minimum and maximum elements in an (a, b)-tree can be found in $O(b \log_a n)$ time.*

1.3.3 Predecessor and Successor

Often, it is useful to have a fast procedure that, given an element x in the dictionary, returns the next greater element, which we call x's ***successor***. In other applications, it may be useful to fing the next smaller element, x's ***predecessor***. For example, imagine finding the 10 best earning people in your company that do not belong to the senior management. Assuming that there is a clear separation between the salary ranges of somebody who is part of the senior management and

somebody who is not, we first search for the best-earning person x_0 who earns no more than the maximal salary one can earn as a regular worker bee—this can be achieved through a straightforward modification of the FIND procedure—and then we find x_0's predecessor x_1, then x_1's predecessor x_2, and so on until we have identified the 10 best earning people $x_9, x_8, \ldots, x_0$ below the given salary cap.

Since the leaves of an (a, b)-tree are sorted in left-to-right order, the predecessor of an element stored at a leaf v is stored at the leaf immediately to the left of v; the successor is stored immediately to the right of v. These two leaves are found by the following two procedures:[2]

```
PREDECESSOR(T, v)
1  while v ≠ NIL and left(v) = NIL
2        do v ← p(v)
3  if v = NIL
4     then return NIL
5     else return MAXIMUM(left(v))
```

```
SUCCESSOR(T, v)
1  while v ≠ NIL and right(v) = NIL
2        do v ← p(v)
3  if v = NIL
4     then return NIL
5     else return MINIMUM(right(v))
```

The running times of these procedures are obviously $O(b \log_a n)$ because the while-loop at the beginning traverses a path from a leaf towards the root and, once the loop terminates, we either spend constant time to return NIL or we invoke one of procedures MINIMUM and MAXIMUM, which, by Proposition 1.2, takes $O(b \log_a n)$ time. The correctness is sufficiently non-obvious that we need to prove it.

Consider procedure PREDECESSOR. (The correctness of procedure SUCCESSOR is established in an analogous fashion.) First observe that, in the while-loop, the initial node v—call this node v_0—is the leftmost leaf of the tree T_v rooted at the current node v. Indeed, this is true before the first iteration. In every iteration, we advance from v to $p(v)$ only if $\mathit{left}(v) = \text{NIL}$, that is, if v is the leftmost child of $p(v)$. This implies that the leftmost leaf of T_v—that is, v_0—is also the leftmost leaf of $T_{p(v)}$, and the invariant holds in the next iteration. Once the loop terminates, we have $v = \text{NIL}$ or $\mathit{left}(v) \neq \text{NIL}$. In the former case, we had $v = \mathit{root}(T)$ at the beginning of the last iteration, that is, v_0 is the leftmost leaf of T and therefore has no predecessor; we return NIL. In the latter case, the predecessor of v_0 is the rightmost leaf of $T_{\mathit{left}(v)}$, that is, $\text{MAXIMUM}(\mathit{left}(v))$. This proves

Proposition 1.3 *The predecessor and successor of a node v in an (a, b)-tree can be found in $O(b \log_a n)$ time.*

[2] These two procedures make use of procedures MINIMUM and MAXIMUM that take an (a, b)-tree node instead of an (a, b)-tree as argument. Essentially, these are the same as the MINIMUM and MAXIMUM procedures on page 9, except that the initialization in Line 1 is to be omitted.

1.3.4 Range Searching

The query operations discussed so far are elementary in the sense that they are looking for exactly one element in T, using different search criteria. Sometimes, it is useful to be able to search not for a single element with a given key, but for all elements whose keys lie in a given range $[l, u]$. Quite naturally, such a query is called a ***range query*** with query interval $[l, u]$. There are many ways of approaching this problem. The one we choose here looks for the leftmost element, x, whose key is no less than l and for the rightmost element, y, whose key is no greater than u. We then report all elements between x and y:

```
RANGE-QUERY(v, l, u)
1  if v is a leaf
2     then if l ≤ key(v) ≤ u
3             then output v
4     else  w ← child(v)
5           while right(w) ≠ NIL and key(right(w)) < l
6              do w ← right(w)
7           while w ≠ NIL and key(w) ≤ u
8              do RANGE-QUERY(w, l, u)
9                 w ← right(w)
```

In order to answer a range query on tree T, we invoke procedure RANGE-QUERY with $v = root(T)$. The set of nodes visited by procedure RANGE-QUERY is shown in Figure 1.5.

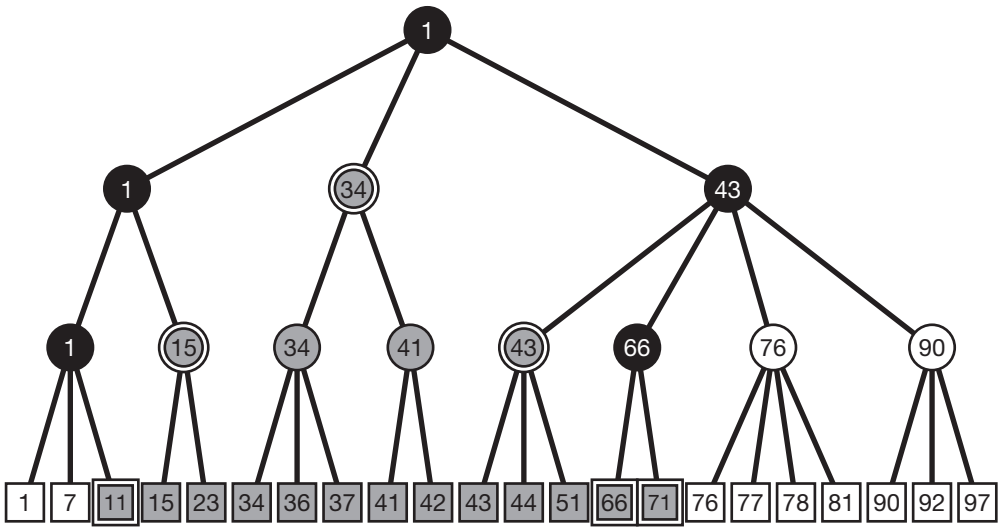


Figure 1.5. The nodes visited by a RANGE-QUERY$(10, 73)$ operation are highlighted in black or grey. Using the terminology in the proof of Lemma 1.8, non-full nodes are black; full nodes are grey; and maximal full nodes have a white border around them.

The following lemma establishes the correctness of procedure RANGE-QUERY.

Lemma 1.7 *Procedure* RANGE-QUERY$(root(T), l, u)$ *finds all items in* T *whose keys lie in the interval* $[l, u]$.

Proof. For a node v, we define its ***height*** to be the number of edges on the shortest path from v to a leaf. We prove by induction on the height of node v that RANGE-QUERY(v, l, u) finds all items in T_v whose keys lie in the interval $[l, u]$.

If the height of v is 0, v is itself a leaf. In this case, we execute Lines 2 and 3, which output v if and only if $key(v) \in [l, u]$. This proves the correctness in the case when v is a leaf. Since we only output a node when we reach it, it also implies that we never report an element whose key is not in $[l, u]$; that is, all elements we do output are in $[l, u]$. So we only have to worry about outputing *all* elements that are in $[l, u]$.

So assume that the height of v is greater than 0, and let $w_1, w_2, \ldots, w_k$ be the children of v. We have to prove that we report all elements in T_v that are in $[l, u]$. Note that $key(w_1) \leq key(w_2) \leq \cdots \leq key(w_k)$. The while-loop in Lines 5 and 6 finds the leftmost child w_i such that $i = k$ or $key(w_{i+1}) \geq l$. Starting at w_i, the while-loop in Lines 7–9 recursively performs range searches on subtrees $T_{w_i}, T_{w_{i+1}}, \ldots, T_{w_j}$, where w_j is the leftmost child such that either $j = k$ or $key(w_{j+1}) > u$. By the induction hypothesis, these range searches find all elements in $T_{w_i}, T_{w_{i+1}}, \ldots, T_{w_j}$ whose keys are in the query interval. Thus, we only have to show that $Keys(w_h) \cap [l, u] = \emptyset$, for $h < i$ or $h > j$.

If $i = 1$, the claim is vacuous for $h < i$. So assume that $i > 1$. Then we have $key(w_{h+1}) < l$ because w_i is the leftmost node with $key(w_{i+1}) \geq l$. Since $key(w_{h+1}) < l$, the ordering of the leaves of T implies that $x < l$, for every $x \in Keys(w_h)$. Thus, the elements in T_{w_h} all lie outside the query interval.

If $j = k$, the claim is vacuous for $h > j$. So assume that $j < k$. Then we have $key(w_h) > u$ because $key(w_{j+1}) > u$ and $key(w_h) \geq key(w_{j+1})$. Since $key(w_h) = \min(Keys(w_h))$, this implies that $x > u$, for every $x \in Keys(w_h)$. Thus, the elements in T_{w_h} all lie outside the query interval. This finishes the proof of the lemma.

Before we state the running time of procedure RANGE-QUERY, let us think what the best running time is we could hope for. First, we should not expect this procedure to be faster than the FIND procedure. For, if it was, the FIND procedure would be useless: we could simulate it by calling RANGE-QUERY$(root(T), x, x)$. Procedure RANGE-QUERY is more general than procedure FIND. The other obvious lower bound on the running time is given by the number of elements we output. If we output only a single element, there is no reason why the procedure should not run in $O(b \log_a n)$ time, the running time of procedure FIND; but in the extreme case, we output *all* n elements in T, and simply listing them takes $\Omega(n)$ time. More generally, if we output t elements in answer to the query, we cannot expect to spend less than t time. Thus, the best running time we can hope for is $O(b \log_a n + t)$, and we prove below that this is indeed the running time of the RANGE-QUERY procedure.

Since the running time of the RANGE-QUERY procedure depends on the size of the output it produces, not only on the input size, we call this procedure ***output sensitive***. The concept of output sensitivity is an important one: As we have argued above, if the output size is n, we cannot hope for a better running time than $O(n)$. So, taking the non-output-sensitive point of view, we could say that, since we cannot do better than $O(n)$ time in the worst case, we have an optimal algorithm if we can answer range search queries in linear time. This is very easy: just inspect every element to determine whether it falls in the query range. However, if the output size is small, there is no good reason why we should have

to inspect all elements stored in the tree. So an algorithm that runs fast for small output sizes and deteriorates with larger output sizes is much better.

Now let us prove that the running time of procedure RANGE-QUERY is indeed $O(b \log_a n + t)$. The following lemma establishes this.

Lemma 1.8 *Procedure* RANGE-QUERY$(root(T), l, u)$ *takes* $O(b \log_a n + t)$ *time, where* n *is the number of elements in* T *and* t *is the number of elements output in answer to the query.*

Proof. To prove the lemma, we prove the more general claim that, for every node v, invoking procedure RANGE-QUERY(v, l, u) takes $O(b \log_a n_v + t_v)$ time, where n_v is the number of elements stored in T_v, that is, $n_v = |\mathit{Items}(v)|$, and t_v is the number of items in T_v that match the query.

We consider a node w visited if we make an invocation RANGE-QUERY(w, l, u); that is, we consider node v visited, as well as all nodes on which we make recursive calls in Line 8. The cost for visiting a node w depends on whether it is a leaf or an internal node. If w is a leaf, we execute Lines 2 and 3, which clearly takes $O(1)$ time. If w is an internal node, we execute Lines 4–9. The number of iterations of the two while-loops is bounded by the number of children of w, which is at most b. Every iteration, excluding the recursive call to procedure RANGE-QUERY, costs constant time. Hence, the cost for visiting an internal node is $O(b)$.

We call a visited node w ***full*** if we output all elements in T_w; that is, if $t_w = n_w$. To obtain the desired time bound, we prove that we spend $O(t_v)$ time on visiting full nodes and that we visit $O(\log_a n_v)$ non-full nodes, at most two per level of T_v. Since we have just argued that visiting any node takes $O(b)$ time, the total time bound is thus $O(b \log_a n_v + t_v)$, as claimed.

Full nodes: First observe that, for a full node w, all its descendants are also full. Hence, we can identify a set of ***maximal*** full nodes $u_1, u_2, \ldots, u_k$ all of whose descendants are full, but whose parents are not. Visiting the nodes in a single tree T_{u_i} takes $O(t_{u_i})$ time: There are t_{u_i} leaves in T_{u_i} because u_i is full, that is, we output all the elements stored at the leaves of T_{u_i}. Since T_{u_i} is an (a, b)-tree, this implies, by Lemma 1.2, that T_{u_i} has $O(t_{u_i})$ nodes. The cost of visiting any node w in T_{u_i} is $O(1 + \mathit{deg}(w))$. Since every node in T_{u_i}, except u_i, is the child of exactly one node, we obtain that the total cost of visiting all nodes in T_{u_i} is $O(t_{u_i})$. Hence, the cost of visiting all nodes in trees $T_{u_1}, T_{u_2}, \ldots, T_{u_k}$ is $O(\sum_{i=1}^{k} t_{u_i}) = O(t_v)$ because $\sum_{i=1}^{k} t_{u_i} = t_v$.

Non-full nodes: To bound the number of visited non-full nodes, we prove that there are at most two such nodes per level. By Lemma 1.1, the height of T_v is $O(\log_a n_v)$, so that we obtain our claim.

To prove that we visit at most two non-full nodes per level, we make the following observation: For every visited node $w \neq \mathit{root}(T)$, we have $\mathit{key}(w) \leq u$ and $\mathit{right}(w) = \text{NIL}$ or $\mathit{key}(\mathit{right}(w)) \geq l$. Indeed, the first loop, in Lines 5–6 skips over all children of $p(w)$ such that $\mathit{right}(w) \neq \text{NIL}$ and $\mathit{key}(\mathit{right}(w)) < l$. Since we have $\mathit{key}(w_1) \leq \mathit{key}(w_2) \leq \cdots \leq \mathit{key}(w_k)$, where $w_1, w_2, \ldots, w_k$ are the children of v in left-to-right order, this implies that either $\mathit{right}(w) = \text{NIL}$ or $\mathit{key}(\mathit{right}(w)) \geq l$. The second loop, in Lines 7–9 exits as soon as it finds the first node w' with $\mathit{key}(w') > u$. Hence, since we visit w, that is, we invoke procedure RANGE-QUERY(w, l, u) in Line 8, we have $\mathit{key}(w) \leq u$.

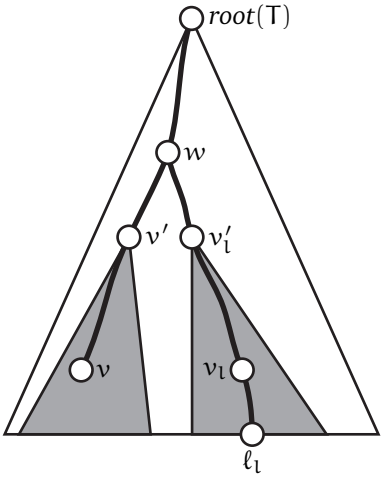


Figure 1.6. Node v cannot be visited.

Given this characterization of visited nodes, we can now prove that every visited non-full node is an ancestor of one of two nodes: If $t_v > 0$, let ℓ_l be the leaf immediately to the left of the leftmost full leaf, and let ℓ_r be the rightmost full leaf. If $t_v = 0$, let ℓ_l be the rightmost leaf of T and let $\ell_r = \ell_l$. We claim that every non-full node we visit is an ancestor of ℓ_l or ℓ_r. Since each of these two nodes has exactly one ancestor per level, this implies our claim that we visit at most two non-full nodes per level.

Now assume for the sake of contradiction that there is a level where we visit a non-full node v that is not an ancestor of ℓ_l or ℓ_r. Let v_l be the ancestor of ℓ_l at this level, and let v_r be the ancestor of ℓ_r at this level. First observe that v cannot be strictly between v_l and v_r because then all leaves that are descendants of v are strictly between ℓ_l and ℓ_r and are therefore full; that is, v would be full in this case. Thus, either v is strictly to the left of v_l or strictly to the right of v_r. In the former case (see Figure 1.6), let w be the lowest common ancestor (LCA) of v and v_l, that is, the node farthest away from the root that is an ancestor of both v and v_l; let v' and v'_l be the children of w such that $v \in T_{v'}$ and $v_l \in T_{v'_l}$. Then $key(right(v')) \leq key(v'_l) \leq key(\ell_l) < l$. Hence, by our characterization of visited nodes, we do not visit v' and, thus, cannot visit v, a contradiction. If v is strictly to the right of v_r, let w be the LCA of v_r and v, and let v'_r and v' be the children of w such that $v_r \in T_{v'_r}$ and $v \in T_{v'}$. Then $key(v') \geq key(right(v'_r)) > u$. The latter inequality follows because ℓ_r is the rightmost leaf with $key(\ell_r) \leq u$; therefore, all leaves in $T_{right(v'_r)}$, which are to the right of ℓ_r, must have a key greater than u. Since $key(v') > u$, we do not visit v' and, hence, we do not visit v either, a contradiction again.

1.4 Updating (a, b)-Trees

What we have established so far is that an (a, b)-tree is a space-efficient (linear-space) dictionary that supports basic queries in the same complexity as a red-black tree; and, arguably, its structure—the reason why the height is bounded and queries are efficient—is much more transparent than is the case for a red-black tree. However, a dictionary is good only if we can update it quickly—we want to be able to add and remove elements to and from the set stored in the dictionary without having to rebuild the entire dictionary, and this should be fast. In standard terms, we want the dictionary to be ***dynamic***, in contrast to a ***static*** dictionary, which has to be rebuilt completely when the data set changes.

1.4.1 Insertion

First we discuss the INSERT operation, which inserts a new element x into the tree. Intuitively—and, in fact, also technically—the process is quite simple (see Figure 1.7): First we locate the rightmost leaf v that stores a key no greater than x.[3] We create a leaf w, which we insert between v and its right sibling, and store x at w. If we are lucky, this finishes the process. If we are unlucky, the insertion of the new leaf w increases the number of children of v's parent to $b + 1$. In this case, we need to spend some extra effort to restore the degree of v's parent.

[3] This is easily done using a slightly modified FIND procedure. In particular, it is easy to verify that procedure FIND(T, x) always reaches the rightmost leaf v that stores a key no greater than x. Now, no matter whether this key is equal to x, we simply return v; that is, the two cases in Lines 7–9 are replaced by a single **return** v statement.

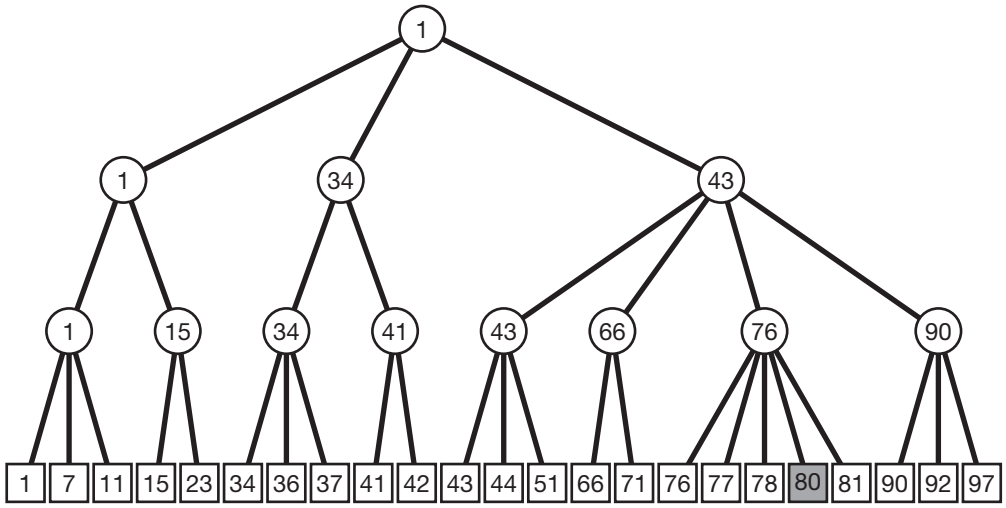


Figure 1.7. The insertion of element 80 into the tree in Figure 1.1 leads to the addition of the grey leaf. This increases the parent's degree to 5, which forces us to rebalance the tree if $a = 2$ and $b = 4$.

The basic process is again quite simple: Let $u = p(v)$. Then we "split" node u into two nodes u and u', that is, we remove the rightmost $\lfloor (b+1)/2 \rfloor$ children of u, make them the children of a new node u', define the key of u' to be equal to the key of the leftmost child of u', and make u' the right sibling of u. See Figure 1.8.

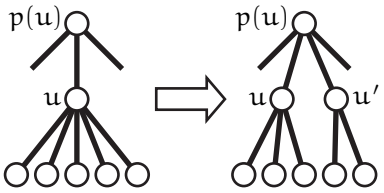


Figure 1.8. The split of a $(2, 4)$-tree node.

As long as $b \geq 2a - 1$, both u and u' will have a number of children between a and b. However, u's parent has now gained a new child, which may increase its number of children to $b + 1$. The fix is easy: we continue this splitting process at u's parent, slowly working our way towards the root until we either reach a situation where splitting the current node x does not increase the degree of x's parent to $b + 1$ or we split the root. If we split the root $root(T)$ into two nodes r and r', we create a new root node r'' and make r and r' the children of r''. This increases the height of the tree by one. This is also the kind of situation why we allow the root of an (a, b)-tree to have a degree less than a. The result of rebalancing the tree in Figure 1.7 is shown in Figure 1.9.

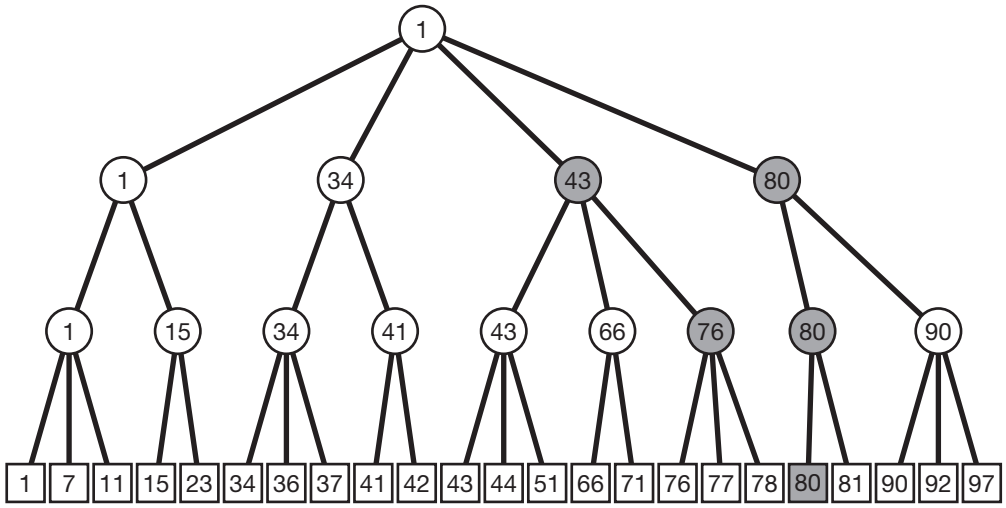


Figure 1.9. The tree obtained after rebalancing the tree in Figure 1.7. The nodes that have been split are shown in grey.

The INSERT procedure, which we have just described informally, is easily formalized, assuming that we have two elementary procedures for creating a new right sibling of a given node and for performing node splits at our disposal. These procedures are provided on the next page.

First, let us establish the correctness of procedure INSERT. We do not have to prove that the search tree property of the tree is maintained because we obviously place the new element in the right spot and update the keys of split nodes correctly: The key of a node v is defined to be the minimum element in T_v. Since the leaves are sorted from left to right, the minimum element in T_v is stored in the subtree T_w rooted at v's leftmost child w, and it is the minimum element in this subtree. Since w's key is correct, we know that $key(w)$ is the minimum element in T_w, that is, the minimum element in T_v, and we correctly assign this value to $key(v)$ in Line 7 of procedure SPLIT.

The following observation is the key to proving that the INSERT procedure also maintains the (a, b)-tree properties:

Observation 1.1 *Procedure* SPLIT(T, v) *produces two nodes of degree between* a *and* b*, provided that the degree of* v *before the split is* $b + 1$*.*

Proof. Procedure SPLIT(T, v) leaves the first $\lceil deg(v)/2 \rceil$ children of v as children of v and makes the last $\lfloor deg(v)/2 \rfloor$ children of v children of the new node w. We have to prove that $a \leq \lfloor deg(v)/2 \rfloor$ and $\lceil deg(v)/2 \rceil \leq b$. But this is easy: We have $\lfloor deg(v)/2 \rfloor = \lfloor (b+1)/2 \rfloor \geq \lfloor 2a/2 \rfloor = a$. Similarly, we have $\lceil deg(v)/2 \rceil = \lceil (b+1)/2 \rceil \leq b/2 + 1 \leq b$ because $b \geq 3$.

Using Observation 1.1, we can now prove the following lemma:

Lemma 1.9 *Procedure* INSERT(T, x) *maintains the* (a, b)*-tree properties of* T*.*

Proof. Let us first get the easy ones out of the way: Obviously, every node we create has an associated key. For the new leaf we create, its key is x. We have already argued above that, for every internal node created by a SPLIT operation, we compute its key correctly. Hence Property (AB5) holds. Properties (AB1) and (AB2) are also satisfied because we make the new leaf a sibling of an existing leaf, store x at this new leaf, and SPLIT operations do not increase the depth of any node in the tree, except when we create a new root; but then the depth of *every* node increases by one.

The interesting properties are Properties (AB3) and (AB4). We use a loop invariant to prove that these two properties are maintained. In particular, we prove that the while-loop in Lines 12–16 of procedure INSERT maintains the following invariant:

> Every non-leaf node has degree between a and b. The only exceptions are the root, whose degree is at least two, and node u, whose degree may be $b + 1$.

This invariant is obviously true before the first iteration of the while-loop because u is the parent of the newly created leaf v, and this is the only node whose degree may have changed. Hence, since all nodes in the tree satisfied Properties (AB3) and (AB4) before the insertion, the only violation may be at node u.

```
INSERT(T, x)
 1  v ← FIND(T, x)
 2  Create a new node w
 3  if x < key(v)
 4     then key(w) ← key(v)
 5          key(v) ← x
 6          y ← v
 7     else key(w) ← x
 8          y ← w
 9  child(w) ← NIL
10  MAKE-SIBLING(T, v, w)          ▷ Make w the right sibling of v.
11  u ← p(w)
12  while u ≠ NIL
13        do key(u) ← key(child(u))
14           if deg(u) > b
15              then SPLIT(T, u)   ▷ Split u into two nodes u and u′.
16           u ← p(u)
17  return y
```

```
MAKE-SIBLING(T, v, w)
 1  u ← p(v)
 2  if u = NIL
 3     then Create a new node u
 4          root(T) ← u
 5          p(u) ← NIL
 6          left(u) ← NIL
 7          right(u) ← NIL
 8          child(u) ← v
 9          deg(u) ← 1
10          p(v) ← u
11  right(w) ← right(v)
12  right(v) ← w
13  left(w) ← v
14  if right(w) ≠ NIL
15     then left(right(w)) ← w
16  deg(u) ← deg(u) + 1
```

```
SPLIT(T, v)
 1  Create a new node w
 2  MAKE-SIBLING(T, v, w)
 3  x ← child(v)
 4  for i ← 1 to ⌈(b + 1)/2⌉
 5        do x ← right(x)
 6  child(w) ← x
 7  key(w) ← key(x)
 8  deg(w) ← ⌊(b + 1)/2⌋
 9  right(left(x)) ← NIL
10  left(x) ← NIL
11  while x ≠ NIL
12        do p(x) ← w
13           x ← right(x)
```

Now consider a given iteration. Assume for now that u is not the root of T. This iteration is executed because $deg(u) > b$, that is, $deg(u) = b + 1$. We then apply procedure SPLIT to node u. By Observation 1.1, this creates two nodes that satisfy the degree bounds. There are no changes to the degrees of any other nodes in the tree, except that u's parent gains a child. This may bring $p(u)$'s degree to $b + 1$. Since we assign $p(u)$ to u at the end of the iteration, it is true before the next iteration that u is the only node that may possibly violate the degree bound.

If u is the root of the tree, then the MAKE-SIBLING operation invoked from the SPLIT operation creates a new root node whose children are u and its newly created sibling. Thus, the result is a root node of degree two, which clearly satisfies Property (AB3).

Since our loop maintains the loop invariant, we can now easily argue that properties (AB3) and (AB4) are restored at the end of procedure INSERT. Indeed, the while-loop may exit because of one of two reasons: either $u = \text{NIL}$ or $deg(u) \leq b$. In either case, there is no violation of Property (AB3) or (AB4) at node u. Since the only possible violation is at node u, there is no violation left. The resulting tree is a valid (a, b)-tree again.

The running time of procedure INSERT is easy to analyze:

Lemma 1.10 *Procedure* INSERT(T, x) *takes* $O(b \log_a n)$ *time.*

Proof. The invocation of procedure FIND in Line 1 takes $O(b \log_a n)$ time, by Lemma 1.6. The cost of Lines 2–11 is clearly $O(1)$. We execute at most one iteration of the while-loop in Lines 12–16 per level of the tree. The cost of each such iteration is bounded by the cost of a SPLIT operation, whose cost we bound by $O(b)$ next. Thus, the total cost of the loop is $O(b \log_a n)$, $O(b)$ for each of the $O(\log_a n)$ levels.

To see that the cost of a SPLIT operation is $O(b)$, we observe that the total number of iterations of the two loops in this procedure is bounded by the number of children of node v, which is at most $b + 1$. Every iteration costs constant time. Hence, the cost of the loops is $O(b)$. Outside of the two loops, we perform a number of constant-time operations and invoke procedure MAKE-SIBLING, which is also easily seen to take constant time. Hence, the total cost of procedure SPLIT is $O(b)$, as claimed.

1.4.2 Deletion

To delete an item, we perform essentially the opposite operations performed by an INSERT operation:

First we locate the leaf v storing the element we want to delete. If the keys of all items are unique, we can achieve this using a FIND operation. If the keys are not unique, then the application usually has another way of uniquely identifying the item to be deleted, often in the form of a direct pointer to the tree node storing this item.

Given node v, we remove it from the list of its parent's children. Again, we have to rebalance the tree only if this leads to a violation of the degree bounds of v's parent. This is the case if either $p(v)$ is not the root of T and $deg(p(v)) < a$ or if $p(v)$ is the root of T and $deg(p(v)) < 2$. If $p(v)$ is the root of T, rebalancing is easy: We remove $p(v)$ from T and make its only child the root of T. If $p(v)$ is not the root, rebalancing is more complicated.

Let us examine our options for restoring the degree constraints of all nodes in the tree after $deg(u)$ becomes $a - 1$, where $u = p(v)$. The natural idea is to take one of its two immediate siblings, the left or right one, and merge u with this sibling—let's call this sibling w. See Figure 1.10. In the best case, $a \leq deg(u) + deg(w) \leq b$. In this case, the merging of u and w produces a node whose degree is within the permissible range. Note that $a \leq deg(u) + deg(w)$ is always true because $deg(u) = a - 1$ and $deg(w) \geq 1$. So the constraint we are worried about is the upper bound. Since $deg(w)$ may already be equal to b before merging u and w, this merge may produce a node whose degree exceeds b. If this happens, we can correct this situation by splitting the merged node again. Indeed, its degree is at least $b + 1$ and at most $a + b - 1$. Thus, the two nodes resulting from the split have degree at least $\lfloor (b+1)/2 \rfloor \geq \lfloor 2a/2 \rfloor = a$ and at most $\lceil (a + b - 1)/2 \rceil \leq \lceil 2b/2 \rceil = b$.

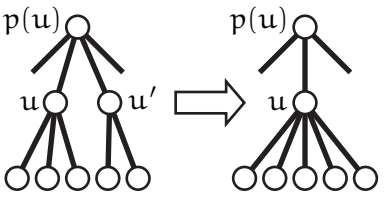


Figure 1.10. Merging two $(2, 4)$-tree nodes.

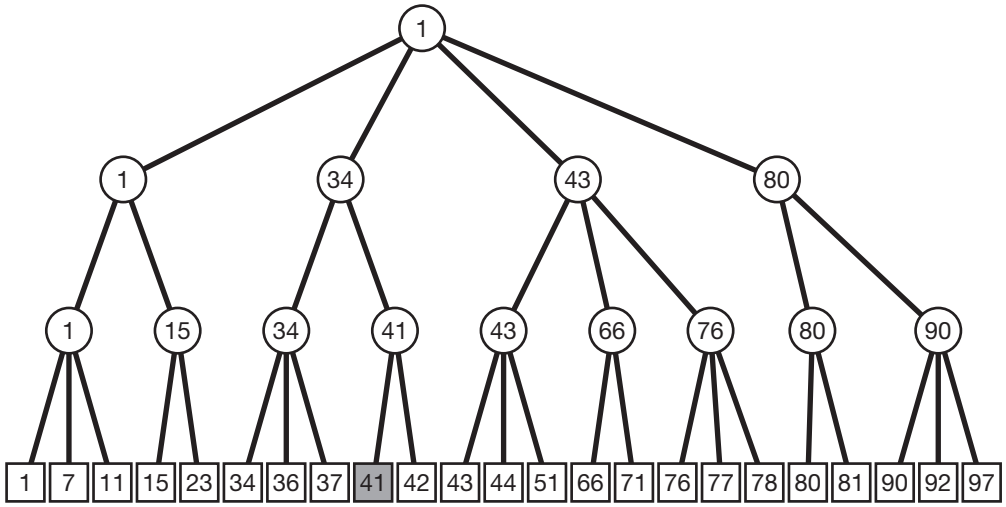
How may this process affect u's parent $p(u)$? If merging u and w produces a node of degree greater than b, we split this node again; that is, we effectively replace u and w with two new children of $p(u)$. Hence, the degree of $p(u)$ does not change, and the rebalancing process terminates. If we do not split the node produced by merging u and w, the degree of $p(u)$ is reduced by one. This may bring its degree down to $a-1$. Our strategy is now the same as before: We merge $p(u)$ with one of its siblings and continue this process towards the root until we either end with a merge followed by a split or we reduce the degree of the root to 1, in which case we remove the root and thereby the only remaining degree violation in the tree. See Figure 1.11 for an illustration of the Delete procedure. The details of the DELETE procedure are shown on page 21. We assume that the node to be deleted is given as a parameter of the procedure because how we determine this node may depend on the particular application.

In Line 2 of procedure DELETE(T, v), we invoke procedure REMOVE-NODE, which deletes node v from the tree, fixing all pointers in and out of v and destroying node v. Then the while-loop in Lines 3–12 walks up the tree, starting at v's parent, and merges nodes whose degree has dropped below a with their siblings. In Lines 7–10, we find a sibling of u. If this sibling is the left sibling of u, we rename u to be this left sibling and w to be the old value of u, to ensure that w is always to the right of u. Then we call procedure FUSE-OR-SHARE to merge u and w, possibly followed by a split.

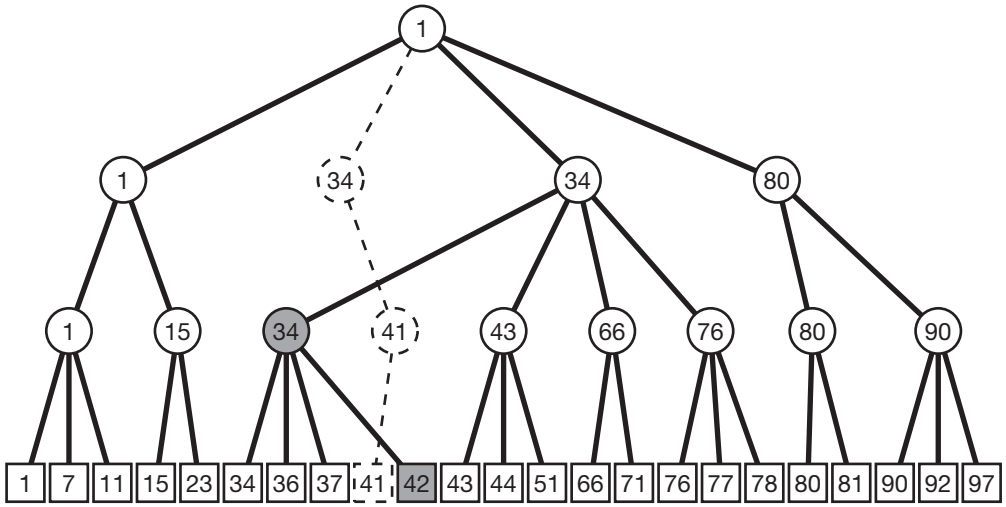
Procedure REMOVE-NODE is rather straightforward. It removes node v from the tree. If v is the root, we are actually removing the last node of the tree. Thus, we set $root(T) =$ NIL in Line 3. If v is not the root, we first decrease the degree of v's parent. Then we test whether v is $p(v)$'s leftmost child and, if so, make $child(p(v))$ point to v's right sibling. We also make the left and right siblings of v point to each other, thereby removing v from the linked list of children of $p(v)$. We finish the procedure by physically deallocating node v.

Procedure FUSE-OR-SHARE first calls procedure FUSE to merge u and w. This may increase u's degree to a value greater than b. If this is the case, we invoke SPLIT(T, u) to split u into two nodes again.

Procedure FUSE, finally, performs the operation shown in Figure 1.10. First, in Lines 1–3, we locate the rightmost child of u. Then, in Lines 4 and 5, we concatenate the list of u's children with the list of w's children. In Line 6, we update u's degree to account for the new children it has gained from w. In Lines 7–10, we update the parent pointers of all children of w to point to u, their new parent. We finish by removing node w from the tree.



(a)



(b)

Figure 1.11. The deletion of element 41 from the tree in Figure (a) leads to the removal of the nodes on the dashed path from the tree in Figure (b). As a result, the parents of the grey nodes change.

```
DELETE(T, v)
 1  u ← p(v)
 2  REMOVE-NODE(T, v)
 3  while u ≠ NIL
 4        do key(u) ← key(child(u))
 5           u′ ← p(u)
 6           if deg(u) < a
 7             then if right(u) = NIL
 8                     then w ← u
 9                          u ← left(u)
10                     else  w ← right(u)
11                  FUSE-OR-SHARE(T, u, w)
12           u ← u′
```

```
REMOVE-NODE(T, v)
 1  u ← p(v)
 2  if u = NIL
 3     then root(T) ← NIL                          ▷ v is the root
 4     else
 5          deg(u) ← deg(u) − 1
 6          if left(v) = NIL
 7            then child(u) ← right(v)             ▷ v is u's leftmost child
 8            else  right(left(v)) ← right(v)
 9          if right(v) ≠ NIL
10            then left(right(v)) ← left(v)
11  Destroy node v
```

```
FUSE-OR-SHARE(T, u, w)
1  FUSE(T, u, w)
2  if deg(u) > b
3     then SPLIT(T, u)
```

```
FUSE(T, u, w)
 1  x ← child(u)
 2  while right(x) ≠ NIL
 3        do x ← right(x)
 4  left(child(w)) ← x
 5  right(x) ← child(w)
 6  deg(u) ← deg(u) + deg(w)
 7  x ← child(w)
 8  while x ≠ NIL
 9        do p(x) ← u
10           x ← right(x)
11  REMOVE-NODE(T, w)
```

Procedures REMOVE-NODE, FUSE-OR-SHARE, and FUSE are helper procedures that ensure that the pointer structure of the (a, b)-tree is updated correctly. Their correctness is obvious, and we leave it to the reader to verify that the running time of procedure FUSE-OR-SHARE is $O(b)$.

The correctness of procedure DELETE follows immediately from the discussion we gave before providing the exact code of this procedure. Its time complexity is stated in the following lemma:

Lemma 1.11 *Procedure* DELETE *takes* $O(b \log_a n)$ *time.*

Proof. We spend constant time in Lines 1 and 2. The cost of each iteration of the while-loop in Lines 3–12 is dominated by the cost of the call to FUSE-OR-SHARE, which is $O(b)$. Since every iteration brings us one level closer to the root, the number of iterations is bounded by the height of the tree which, by Lemma 1.1, is $O(\log_a n)$. Hence, the total cost of the procedure is $O(b \log_a n)$, as claimed.

1.5 Building (a, b)-Trees

An operation we are occasionally interested in is rebuilding an (a, b)-tree from scratch. How quickly can we accomplish this? Obviously, we can do this in $O(bn \log_a n)$ time: Start with an empty (a, b)-tree and insert the elements one by one. This takes $O(b \log_a n)$ time per insertion, $O(bn \log_a n)$ time in total.

But assume that we are given the elements to be inserted in sorted order. Can we then build an (a, b)-tree in linear time? In this section, we discuss a linear-time construction procedure for (a, b)-trees. The central idea is to build the tree bottom-up, that is, starting from the leaves and placing internal nodes on top of them, layer by layer. While this may seem strange at first, on second thought it is quite natural that this strategy should be the road to success:

Consider why an insertion takes $O(b \log_a n)$ time. First we have to locate the leaf (!) where to insert the given element; this search takes $O(b \log_a n)$ time. Then we rebalance bottom-up (!) by performing node splits. By inserting all leaves at the same time, we avoid the searching cost altogether. The batched creation of internal nodes bottom-up is equivalent to performing all node splits corresponding to the performed insertions simultaneously.[4] The procedure that implements the bottom-up construction of an (a, b)-tree is shown on the next page.

The basic strategy is simple: As long as there is more than one node left on the current level, we have to add another level on top of it. During the construction of the next level, as long as there are at least $2b$ nodes on the current level without parent, take the next b nodes, make them children of a new node v, and add v to the next level. Once there are less than $2b$ nodes left, there are two cases: If the number l of remaining nodes is more than b, we make these l nodes children of two new nodes at the next level, distributing them evenly. If $l \leq b$, we make all the remaining nodes children of one node.

Before analyzing the complexity of this procedure, we should ask two questions: Does this produce a valid (a, b)-tree? And, why can't we just form groups of b nodes at every level until we are left with less than b nodes, which we make

[4] In fact, one can prove that the total number of node splits performed by n insertions is $O(n)$. Thus, the real bottleneck is the searching step, which we avoid.

```
BUILD-TREE(A, n)
    ▷ The elements in A are assumed to be sorted.
 1  Create an empty node array N of size n.
    ▷ N holds the nodes of the most recently constructed level.
 2  for i ← 1 to n
 3       do create a new node v
 4          key(v) ← A[i]
 5          child(v) ← NIL
 6          left(v) ← NIL
 7          right(v) ← NIL
 8          p(v) ← NIL
 9          N[i] ← v
10  m ← n    ▷ m is the size of N.
11  while m > 1
12       do j ← 1
13          k ← 0
14          while j ≤ m − 2b + 1
15               do k ← k + 1
16                  N[k] ← ADD-PARENT(N, j, b)
17                  j ← j + b
18          if j < m − b + 1
19            then h ← ⌈(m−j+1)/2⌉
20                  N[k + 1] ← ADD-PARENT(N, j, h)
21                  N[k + 2] ← ADD-PARENT(N, j + h, m − j − h + 1)
22                  k ← k + 2
23            else N[k + 1] ← ADD-PARENT(N, j, m − j + 1)
24                  k ← k + 1
25          m ← j
26  root(T) ← N[1]
27  return T

ADD-PARENT(N, j, h)
 1  Create a new node v
 2  child(v) ← N[j]
 3  key(v) ← key(N[j])
 4  left(v) ← NIL
 5  right(v) ← NIL
 6  p(v) ← NIL
 7  for i ← j to j + h − 2
 8       do p(N[i]) ← v
 9          right(N[i]) ← N[i + 1]
10          left(N[i + 1]) ← N[i]
11  p(N[j + h − 1]) ← v
12  return v
```

children of the final node at the next level? The answer to the first question will shed light on the second question as well.

Lemma 1.12 *Procedure* BUILD-TREE *produces a valid* (a, b)*-tree.*

Proof. Obviously, the leaves of the produced tree T are all at the same level; and the data items are stored at these leaves, sorted from left to right, because A is sorted. Thus, Properties (AB1) and (AB2) are satisfied. Property (AB5) is also satisfied because we choose the key of every non-leaf node to be equal to the key of its leftmost child.

To prove Properties (AB3) and (AB4), we first establish that no node in T has more than b children. We create non-leaf nodes by invoking procedure ADD-PARENT in four different places of the algorithm. When invoking procedure ADD-PARENT in Line 16, the node created by this invocation has exactly b children. When invoking procedure ADD-PARENT in Line 23, we have $j \geq m-b+1$, that is, $m-j+1 \leq b$. Hence, the node created by this invocation has at most b children. When invoking procedure ADD-PARENT in Line 20 or Line 21, the created node has at most $h = \lceil (m-j+1)/2 \rceil$ children. Since $j > m-2b+1$, we have $m-j+1 < 2b$, that is, $h \leq b$. Thus, any non-leaf node we create has at most b children.

To establish a lower bound on the degree of every node, we consider the three possibilities again. We have just argued that every node created by an invocation to ADD-PARENT in Line 16 has exactly b children. Since $b \geq a$, this node has at least a children. Any node created by an invocation in Line 20 or Line 21 has degree at least $h' = \lfloor (m-j+1)/2 \rfloor$. However, Lines 20 and 21 are executed only if $j < m-b+1$, that is, $m-j+1 > b$. Hence, $h' \geq b/2 \geq a$, because $b \geq 2a-1$. The crux in the analysis is the invocation in Line 23. If v is the only node created on the current level, then v is the root of T. Since we enter the while-loop only if there are two nodes left without parent, v has at least two children, that is, Property (AB4) is satisfied. If v is not the only node, observe that the node u immediately to the left of v must have been created in Line 16. Hence, immediately before the creation of u, there must have been at least $2b$ nodes left without parent. Exactly b of them are made children of u, which leaves at least b children for v. Hence, in this case v has $b \geq a$ children. This proves that every non-root node satisfies Property (AB3).

Do you see where the proof would have gone wrong if we had formed groups of b nodes and made each group children of the same node, followed by the creation of a final group with possibly less than b nodes that are children of the last node on the next level?

The final lemma in this chapter establishes that procedure BUILD-TREE takes linear time.

Lemma 1.13 *Procedure* BUILD-TREE *takes* $O(n)$ *time.*

Proof. This proof is rather straightforward. Indeed, we observe that the cost of procedure BUILD-TREE is $O(1)$ plus the time spent in the loops in Lines 2–9 and Lines 11–25. Every iteration of the loop in Lines 2–9 costs constant time, and there are n such iterations, one per element in A. Hence, the cost of Lines 2–9 is $O(n)$.

Every iteration of the while-loop in Lines 11–25 costs constant time plus the time spent in the while-loop in Lines 14–17, plus the time spend in invocations to procedure ADD-PARENT. We analyze this cost as follows: Procedure ADD-PARENT costs time $O(h)$, where h is the number of nodes that are made children of the newly created node. Since every node is made the child of exactly one other node, the total cost of all invocations to procedure ADD-PARENT is $O(|T|)$. Similarly, since every iteration of the while-loop in Lines 14–17 creates one new node (by invoking ADD-PARENT), the total number of iterations is bounded by $|T|$; every iteration costs constant time plus the time spent in ADD-PARENT, which we have already accounted for. Hence, the total cost of all iterations of Lines 14–17 is $O(|T|)$ as well. Finally, we observe that the outer loop in Lines 11–25 performs one iteration per level of $|T|$, that is, the number of these iterations is bounded by *height*(T), and their total cost is $O($*height*$(T))$.

Since we have already argued that the tree T produced by procedure BUILD-TREE is is a valid (a, b)-tree, we know, by Lemmas 1.1 and 1.2 that $|T| = O(n)$ and *height*$(T) = O(\log_a n)$. Hence, the total cost of Lines 11–25 is $O(n)$, and the lemma follows.

1.6 Concluding Remarks

With the description of the DELETE operation in the previous section, we have finished the repertoire of elementary dictionary operations on an (a, b)-tree. We have established in this chapter that an (a, b)-tree uses linear space and supports all elementary operations in $O(b \log_a n)$ time, and operation RANGE-QUERY in $O(b \log_a n + t)$ time.

Now, what's a good choice for a and b? There are some applications where choosing non-constant values for a and b is a good idea; but they are beyond the scope of these notes. Throughout these notes, we choose a and b to be some suitable constants, say $a = 2$ and $b = 4$. With this choice of constants, all elementary operations cost $O(\lg n)$ time, and a range query costs $O(\lg n + t)$ time. These are the same bounds as those achieved by a red-black tree; but the (a, b)-tree is simpler.

1.7 Chapter Notes

(a, b)-trees are first described by Huddleston and Mehlhorn (1982). The variant described here differs from the one in (Huddleston and Mehlhorn 1982) in that the variant of Huddleston and Mehlhorn (1982) connects *all* nodes on a level to form a linked list, rather than just the children of each node. This change is useful because it allows fast ***finger searches***: given a leaf v and a key x, find a leaf w that stores an element with key x in time $O(\lg d)$, where d is the number of leaves between v and w. In particular, it implies that predecessor and successor queries can be answered in constant time, a major improvement over the $O(\lg n)$ bound achieved by the PREDECESSOR and SUCCESSOR queries on page 10.

(a, b)-trees are also a generalization of the ubiquitous B-tree (Bayer and McCreight 1972), which is used to store massive data collections on disk. A B-tree is usually a $(B/2, B)$-tree, where B is the number of data items that fit into a disk block. In order to make a B-tree efficient in terms of the number of disk accesses, it is necessary to store the keys of the children of each node at the node itself; but this does not change how the tree works conceptually.

Binary search trees are alternatives to (a, b)-trees. In these trees, logarithmic height is achieved by locally changing the parent-child relationship of nodes in the tree, so-called rotations. The different types of balanced binary trees differ mainly in the rules they apply to decide when and where to rotate. Examples of balanced binary search trees include AVL-trees (Adel'son-Vel'skiĭ and Landis 1962), red-black trees (Bayer 1972; Guibas and Sedgewick 1978), AA-trees (Andersson 1993), and BB[α]-trees (Nievergelt and Reingold 1973). Red-black trees and AA-trees are interesting in the context of (a, b)-trees because they can be seen as binary trees representing $(2, 4)$-trees. In particular, for a red-black tree, we can represent every black node and its red children as a single (a, b)-tree node. Then the black-height property implies that all leaves are at the same height. Since there are at most two red children, each of which has two children, the degree of a node in the resulting tree is between 2 and 4; the tree is a $(2, 4)$-tree.

Chapter 2
Data Structuring

In this chapter, we discuss the ***data structuring*** paradigm. As the name suggests, the idea is to use data structures to solve the problem at hand. In many cases, once we have chosen the right data structure, the algorithms solving even non-trivial problems are surprisingly simple. This is because all the complexity of the solution is hidden in the data structure.

The appeal of the data structuring paradigm is two-fold: First, we achieve modularity of the algorithm, which is good software engineering. If we later develop a better data structure that supports the same operations as our current data structure, only much faster, we do not have to change the algorithm; we only replace one data structure with another one. From the algorithm's point of view, the data structure is a black box. This also keeps our thinking clean because we don't have to worry about building the data structure while thinking about how to use the data structure to solve the problem at hand.

Second, once we have gone to the length of developing a nice general-purpose data structure, we can re-use this data structure to solve a wide variety of problems. Again, this idea of re-using code is an important goal in software engineering.

To illustrate the point, we will discuss three problems in this chapter that can be solved by performing the right sequence of operations on an (a, b)-tree. After all, developing the structure cost us considerable effort. So it would be nice if it could be used for more than just storing the entries in a database. In subsequent chapters, we will see how to extend the set of operations supported by an (a, b)-tree. We say that we ***augment*** the (a, b)-tree data structure. Using these augmented (a, b)-trees, we will be able to solve more problems, thereby providing more examples for the applicability of the data structuring paradigm.

2.1 Orthogonal Line-Segment Intersection

Consider the following problem, which is representative of the type of problems that arise in geographic information systems (GIS): We are given two maps of the land owned by a farmer, one representing the soil type and one representing what kind of crop the farmer plans to grow on different parts of his land. See Figure 2.1.

Not every type of crop grows equally well on every type of soil. We can represent this as a revenue in dollars per acre that we can make by growing a certain type of crop on a certain type of soil. The question we want to answer is how much revenue the farmer can expect, given the current layout of the fields. We determine this by computing a new map that is partitioned into regions each of which represents one crop-soil type combination. In Figure 2.2, for example, the three grey regions represent areas where the farmer grows wheat on humus soil. This operation is known as map overlay.

It turns out that the hardest part of computing the overlay of two maps is finding all intersections between the boundaries of the regions in these two maps.

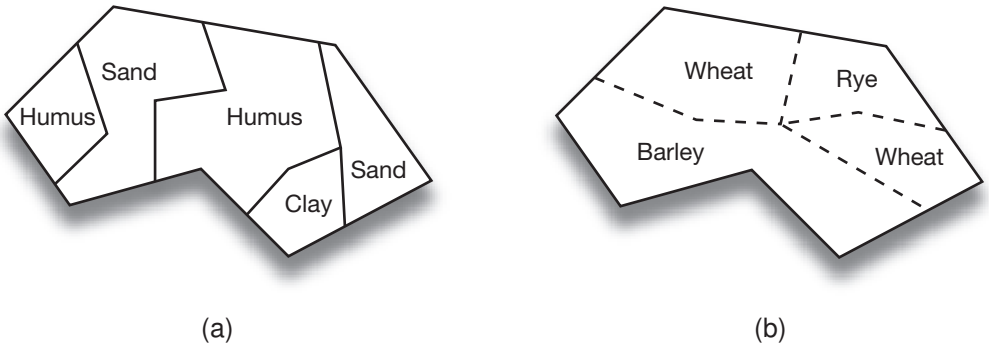


Figure 2.1. Two maps of the same area. (a) The soil type. (b) The type of crop to be grown.

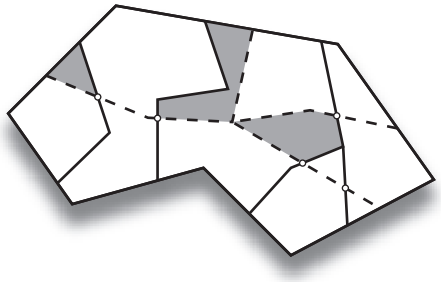


Figure 2.2. The overlay of the two maps in Figure 2.1. The intersections between the region boundaries are marked with white circles.

In GIS, these boundaries are represented as polygonal curves, which consist of straight line segments. So the abstract problem we have is to find all intersections between a set of n straight line segments. In this section, we look at the special case when each segment is either horizontal or vertical. This problem is known as the ***orthogonal line-segment intersection problem***. We study this problem first because it removes a few complications that arise in the general case. In Section 2.3, we study the general problem. So, formally, we want to solve the following problem:

Problem 2.1 (Orthogonal line-segment intersection) *Given a set of vertical line segments,* $V = \langle v_1, v_2, \ldots, v_k \rangle$*, and a set of horizontal line segments,* $H = \langle h_1, h_2, \ldots, h_m \rangle$*, report all pairs* (v_i, h_j) *such that* v_i *and* h_j *intersect.*

Every vertical segment v_i is uniquely described by two y-coordinates $y_i^b \leq y_i^t$ and an x-coordinate x_i; that is, its two endpoints are (x_i, y_i^b) and (x_i, y_i^t). Similarly, we represent a horizontal segment h_j using two x-coordinates $x_j^l \leq x_j^r$ and a y-coordinate y_j.

We define the input size to be $n = k + m$, the number of segments in V and H. We start by observing that this problem has a trivial $O(n^2)$-time solution, which is also optimal in the worst case:

SIMPLE-ORTHOGONAL-LINE-SEGMENT-INTERSECTION(V, H)

```
1  for every segment v_i ∈ V
2      do for every segment h_j ∈ H
3          do if v_i and h_j intersect
4              then output the pair (v_i, h_j)
```

Why is this optimal? Consider the example in Figure 2.3. There are $n^2/4 = \Omega(n^2)$ intersections, and only reporting them requires $\Omega(n^2)$ time, no matter how much time we spend to detect them.

Nevertheless, if there are only few intersections, spending quadratic time to find them seems like a waste. So what is the right running time in this case? Linear time may be too much to hope for. In fact, one can prove that this is impossible, even though we don't do so here. But having an $O(n \lg n)$-time algorithm to find all intersections if there are only few of them seems like a reasonable goal. So, since we are hoping to spend $O(n \lg n)$ time to find and report all intersections if there are only few of them, and we cannot hope to spend less than $\Omega(t)$ time if there are t intersections, we should again aim to develop an output-sensitive algorithm, one whose running time is $O(n \lg n + t)$ time.

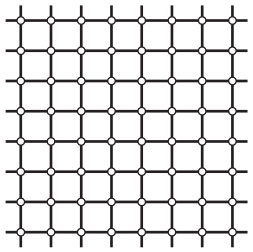


Figure 2.3. An arrangement with $\Omega(n^2)$ intersections.

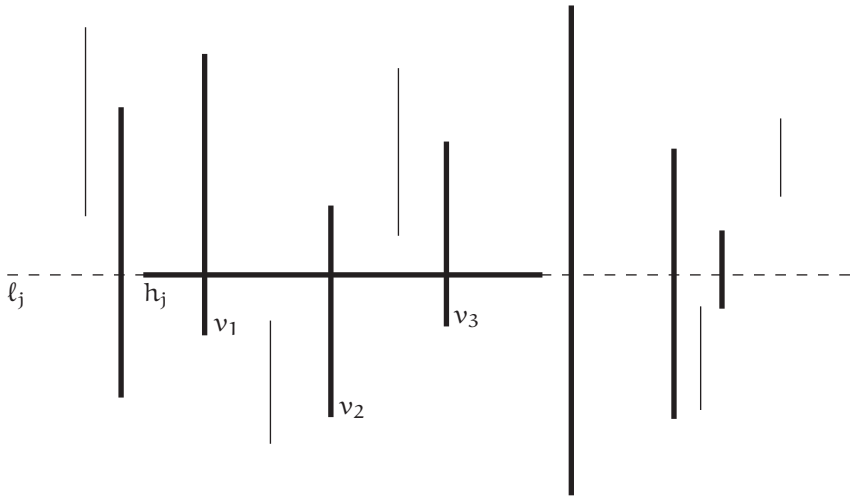


Figure 2.4. The set V_j for segment h_j is shown in bold. From among these segments, only segments v_1, v_2, and v_3 intersect h_j.

To work our way towards a solution, consider for now a single horizontal segment h_j, and let ℓ_j be the horizontal line spanned by h_j. A vertical segment v_i can intersect h_j only if it intersects ℓ_j; that is, it must have one endpoint above ℓ_j and one endpoint below ℓ_j. Let V_j be the set of vertical segments that satisfy this condition. See Figure 2.4. A segment $v_i \in V_j$ intersects h_j if and only if $x_j^l \le x_i \le x_j^r$. This is simply a range query with query interval $[x_j^l, x_j^r]$ over the set of x-coordinates of the segments in V_j. So, if we have an (a, b)-tree T that stores the set V_j, we can find all vertical segments that intersect h_j by invoking procedure RANGE-QUERY(T, x_j^l, x_j^r). The problem is that it is too expensive to build an (a, b)-tree storing the set V_j for each horizontal segment h_j; you can easily verify that this takes $\Omega(n^2)$ time in the worst case, which is no better than our naïve $O(n^2)$-time solution described above.[1]

To overcome this inefficiency, let us see whether we can identify some structure in the problem by considering the horizontal segments in y-sorted order, from bottom to top. Assume w.l.o.g. that $y(h_1) < y(h_2) < \cdots < y(h_m)$. Now choose a segment h_j and consider all vertical segments that *do not* belong to V_j. The segments above ℓ_j cannot belong to any set $V_{j'}$ with $j' < j$ either (see Figure 2.5). Similarly, the segments below ℓ_j cannot belong to any set $V_{j''}$ with $j'' > j$. So if we look at the sets $V_1, V_2, \ldots, V_m$, then we observe that every vertical segment v_i belongs to a range of sets $V_j, V_{j+1}, \ldots, V_{j'}$.

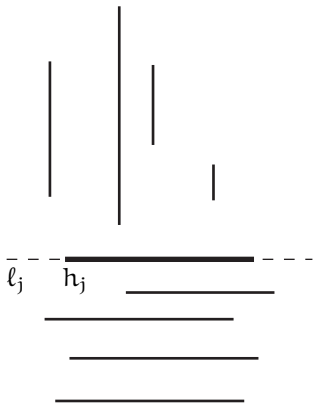


Figure 2.5. Line ℓ_j separates vertical segments above it from horizontal segments below it.

[1] It is in fact worse because it is not faster, but more complicated.

This observation is quite powerful because it allows us to construct all the sets $V_1, V_2, \ldots, V_m$ in turn, while performing only k INSERT and DELETE operations: In order to obtain the set V_{j+1} from the set V_j, we first delete all segments that cross line ℓ_j, but do not cross line ℓ_{j+1}; then we insert all segments that are above ℓ_j, but cross ℓ_{j+1}.

Now our algorithm is essentially complete, at least at a very high level: In order to find all segments intersecting h_1, insert all segments in V_1 into an (a, b)-tree T and then perform a RANGE-QUERY(T, x_1^l, x_1^r) operation. For every subsequent segment h_j, we have the set V_{j-1} stored in T. So we delete all segments in $V_{j-1} \setminus V_j$ from T and insert all segments in $V_j \setminus V_{j-1}$ into T. Now T stores the set V_j, and we perform a RANGE-QUERY(T, x_j^l, x_j^r) operation again to find all vertical segments intersecting h_j.

The procedure we have just invented the hard way is in fact an application of a quite general paradigm for solving geometric problems: the ***plane sweep paradigm***. An algorithm that is based on this paradigm can be thought of as sweeping a horizontal line across the plane, starting at y-coordinate $-\infty$ and proceeding to y-coordinate $+\infty$. The algorithm maintains a ***sweep-line structure***, which represents the interaction of the geometric scene with the sweep line. When the sweep line passes certain ***event points***, it performs certain operations on the sweep-line structure, which may be updates or queries.

In our algorithm, the sweep-line structure is an (a, b)-tree T storing vertical segments. More precisely—this is the interaction we maintain—tree T stores the set of vertical segments intersecting the sweep line. We have two types of event points: When we pass an endpoint p of a vertical segment v_i, we have to update T to ensure that the sweep-line structure continues to represent the set of segments that intersect the sweep line; that is, if p is the bottom endpoint of v_i, we have to insert v_i into T; if it is the top endpoint, we have to delete v_i from T. The second type of event point is the set of y-coordinates of all horizontal segments. When the sweep line passes across a segment h_j, then our invariant that T stores the set of vertical segments intersecting the sweep line implies that T does in fact store the set V_j. Therefore, this is a good time to ask the range query we use to report all vertical segments intersecting h_j. From this discussion, we obtain the following algorithm for the orthogonal line-segment intersection problem:

ORTHOGONAL-LINE-SEGMENT-INTERSECTION(V, H)

```
1  Create an empty (a, b)-tree T to be used as the sweep-line structure.
2  Create a set E of event points.
   ▷ Every event point is a pair (y, v_i) or (y, h_j) that tells us the
   ▷ y-coordinate at which the event occurs and the segment to be
   ▷ processed.
3  Sort the event points in E by their y-coordinates.
4  for every event point e in E, in sorted order
5      do if e = (y, h_j)
6           then RANGE-QUERY(T, x_j^l, x_j^r)
7           else ▷ e = (y, v_i)
8                if y = y_i^b
9                   then INSERT(T, v_i)
10                  else DELETE(T, v_i)
```

The correctness of this procedure follows immediately from our discussion. The running time is stated in the following lemma.

Theorem 2.1 *The running time of procedure* ORTHOGONAL-LINE-SEGMENT-INTERSECTION *is* $O(n \lg n + t)$*, where* n *is the total number of segments and* t *is the number of reported intersections.*

Proof. The creation of an empty (a, b)-tree in Line 1 clearly takes constant time. In order to create the set E in Line 2, all that is required is to sequentially read through V and H and to add two event points per segment in V and one event point per segment in H to E. This takes $\mathcal{O}(n)$ time and adds at most $2n$ event points to E. Consequently, sorting E in Line 3 takes $O(n \lg n)$ time.

The loop in Lines 4–10 runs for at most $2n$ iterations, one per event point. In each iteration, we perform either an insertion, a deletion, or a range query on T. Every insertion or deletion takes $O(\lg n)$ time. The range query corresponding to a segment h_j takes $O(\lg n + t_j)$ time, where t_j is the number of vertical segments intersecting h_j. Hence, the total time spent in the loop is

$$\sum_{i=1}^{k} O(\lg n) + \sum_{j=1}^{m} O(\lg n + t_j) = O(n \lg n) + O\left(\sum_{j=1}^{m} t_j\right) = O(n \lg n + t).$$

Adding the cost of Lines 1–3 to the cost of the loop, we obtain a running time of $O(n \lg n + t)$ for the whole algorithm.

Procedure ORTHOGONAL-LINE-SEGMENT-INTERSECTION is simple and clean; but this was to be expected, given that the (a, b)-tree is a quite powerful data structure. The simplicity, however, is only in the final solution we have obtained; it still required considerable insight to develop the algorithm. This should not be too surprising either. Compare the use of a data structure vs. developing an algorithm that solves the problem with elementary means with driving a car vs. riding a bike. By car, it is much easier and faster to reach one's destination; but driving a car requires more skill. In order to harness the power of data structures, one needs a clear understanding of the strengths and weaknesses of each data structure. This requires experience, and even after years, choosing the right data structure for a problem can require considerable creativity.

2.2 Three-Sided Range Searching

Before addressing the general line-segment intersection problem, let us think about a problem which, on the surface, looks very different from the line-segment intersection problem. Its solution, however, will be very similar to that of the orthogonal line-segment intersection problem, which is why we consider it here. The following is the definition of the problem we want to study:

Problem 2.2 (Three-sided range searching) *Given a set of* n *points in the plane,* $P = \{p_1, p_2, \ldots, p_n\}$*, and a set of* m *three-sided query ranges,* $Q = \{q_1, q_2, \ldots, q_m\}$*, report all pairs* (p_i, q_j) *such that* $p_i \in q_j$*. A three-sided query range* q_j *is described by three coordinates* (x_j^l, x_j^r, y_j^b) *and is the region* $\{p \in \mathbb{R}^2 : x_j^l \leq x_p \leq x_j^r$ *and* $y_p \geq y_j^b\}$*.*

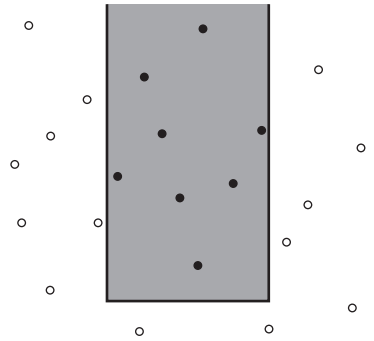


Figure 2.6. A three-sided range query. The set of points to be reported for this query are shown in black. Hollow circles are points that are outside the query range.

If you consider Figure 2.6, then it is clear why we call such a range query three-sided: it is bounded from three sides, the left, the right, and the bottom; but not from the top.

Again, we will solve this problem using the plane-sweep paradigm. We have to figure out what the right event points are and what type of queries to ask. To do so, it helps again to consider a single query q_j. If we were to ask a range query with query range $[x_j^l, x_j^r]$ on an (a, b)-tree storing all points in P, we would be close to an answer: The query would output all points that lie in the x-range of the query; but some of these points may be below the bottom boundary of the query and therefore should not be reported. This suggests that we should answer the range query on an (a, b)-tree storing only the set of points that have y-coordinate at least y_j^b; let us call this set P_j.

And now our machinery gets going again: We observe that, if we sort our queries so that $y_1^b > y_2^b > y_3^b \cdots > y_m^b$, then $P_1 \subseteq P_2 \subseteq \cdots \subseteq P_m$; that is, we can start by constructing P_1, answer query q_1 over this set, add the points in $P_2 \setminus P_1$ to obtain P_2, answer query q_2, and so on.

In the language of the plane-sweep paradigm, our event points are the y-coordinates of the points and the y-coordinates of the bottom boundaries of the queries in Q. For every event point corresponding to a point in P, we insert the point into the (a, b)-tree T. For every event point corresponding to a query q_j, we perform a RANGE-QUERY(T, x_j^l, x_j^r) operation on T. The complete algorithm looks as follows:

THREE-SIDED-RANGE-SEARCHING(P, Q)

```
1  Create an empty (a, b)-tree T to be used as the sweep-line structure.
2  Create the set E of event points.
   ▷ This set contains pairs (y, p_i) or (y, q_j), depending on whether the
   ▷ event point corresponds to a point or a query.
3  Sort the event points in E by their y-coordinates.
4  for every event point e in E, in sorted order
5       do if e = (y, p_i)
6             then INSERT(T, p_i)
7             else ▷ e = (y, q_j)
8                  RANGE-QUERY(T, x_j^l, x_j^r)
```

The correctness of this procedure is obvious, and its complexity is easily bounded by $O((n + m) \lg(n + m) + t)$, where t is the total number of query-point pairs reported. However, we can do a little better: if $m \gg n$, the lg-factor is greater than necessary; we can solve the problem in $O((n+m) \lg n + t)$ time: We divide the set of queries into $\lceil m/n \rceil$ subsets $Q_1, Q_2, \ldots, Q_k$ of size at most n. An invocation of procedure THREE-SIDED-RANGE-SEARCHING with inputs P and Q_i now takes $O(n \lg n + t_i)$ time, where t_i is the output size of the invocation. Since there are $\lceil m/n \rceil < m/n + 1$ invocations, the total complexity of all invocations is

$$O\left(\left(\frac{m}{n} + 1\right) n \lg n + \sum_{i=1}^{k} t_i\right) = O((n + m) \lg n + t).$$

Similarly, if $n \gg m$, we can solve the problem in $O((n + m) \lg m + t)$ time, by partitioning the point set P into $\lceil n/m \rceil$ point sets $P_1, P_2, \ldots, P_k$ of size at most m and invoking procedure THREE-SIDED-RANGE-SEARCHING on all pairs (P_i, Q), for $1 \leq i \leq k$. One such invocation costs $O(m \lg m + t_i)$ time. Hence, using

the same arguments as above, the total running time of all invocations becomes $O((n+m)\lg m + t)$. This proves the following result:

Theorem 2.2 *The three-sided range searching problem can be solved in* $O((n+m)\lg(1+\min(n,m))+t)$ *time.*

We obtain a more interesting and more natural range searching problem when we allow rectangles as query ranges, that is, the query ranges are also bounded from above. These queries are often called four-sided range queries. In Chapter 4, we augment the (a,b)-tree so that it can answer three-sided range queries directly without using the plane-sweep paradigm. By combining this augmented (a,b)-tree with the plane-sweep paradigm, we will be able to answer four-sided range queries in $O((n+m)\lg(1+\min(n,m))+t)$ time. If we are willing to invest $O(n\lg n)$ space, we develop a structure in Chapter 5 that can answer four-sided queries directly, at a cost of $O(\lg n + t)$ time per query.

2.3 General Line-Segment Intersection

We conclude this chapter with a discussion of the general line-segment intersection prolem; that is, now we remove the assumption that every segment is either vertical or horizontal. In fact, we assume now that no segment is horizontal; this can be guaranteed by turning the whole scene by a very small angle if necessary. Since there is no distinction between vertical and horizontal segments any more, we now denote the input to our algorithm as the set $S = \{s_1, s_2, \ldots, s_n\}$; each segment s_i is described by two endpoints $p_i = (x_i, y_i)$ and $q_i = (x_i', y_i')$. We assume w.l.o.g. that, for every segment s_i, $y_i > y_i'$. We also assume that the line segments are in "general position". This term is used very much in computational geometry and is defined to mean that all pathological situations that could make the algorithm fail cannot occur. In our case, we assume that no three segments intersect in a single point; that is, if two segments intersect in a point p, no other segment contains this point. Is this cheating? To some extent, it is; but most of the time, as in the present case, the assumption of general position can be removed by modifying the algorithm to take care of pathological situations. If we were to include these cases in the initial discussion, this would distract from the important ideas. This is why we make this assumption here, to keep things clean.

The general strategy is the same as before: Our sweep-line status maintains the set of segments that intersect the sweep-line. When we pass the bottom endpoint of a segment, we insert it into the sweep-line structure; when we pass its top endpoint, we remove it from the structure. But here is where the trouble starts: First, we have no horizontal segments; so do we ever ask a query on the structure? Second, the segments in the structure may intersect. Thus, there is no well-defined left-to-right order of the segments: below the intersection point, one segment is to the left of the other; above the intersection point, the order is reversed. See Figure 2.7.

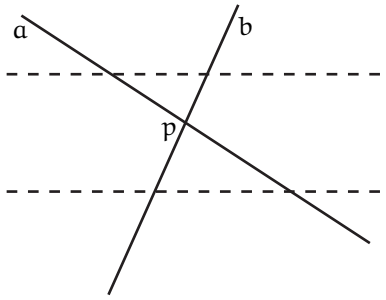


Figure 2.7. Segments a and b change their order when they intersect.

The first "problem" is a pseudo-problem. Nobody forces us to ask range queries on the sweep-line structure. If we can achieve our goal without asking these queries, then why would we want to ask them.

We overcome the second problem by defining a left-to-right order of the segments with respect to the current position of the sweep line. The sweep line intersects all segments in the sweep-line structure. We define the order of the

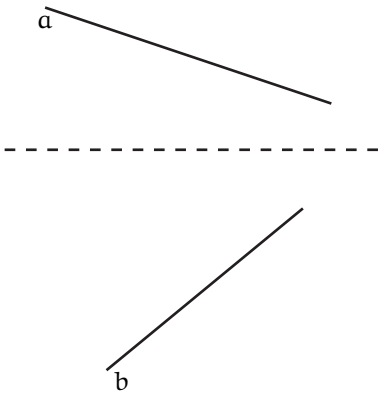


Figure 2.8. Segments a and b are separated by a horizontal line.

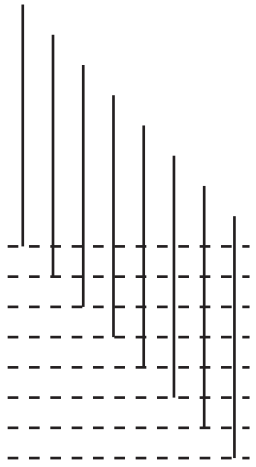


Figure 2.9. The algorithm performs $\Omega(n^2)$ intersection tests; but there are no intersections.

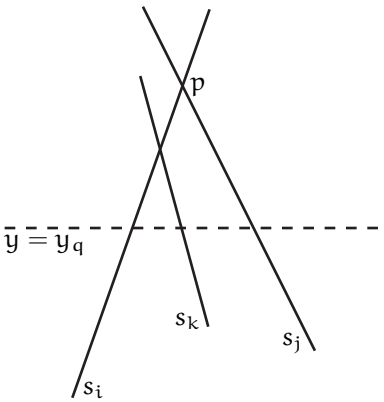


Figure 2.10. Segment s_k must intersect at least one of s_i or s_j.

segments to be the same as the left-to-right order of their intersection points with the sweep line.

This is certainly a valid ordering of the segments in the sweep-line structure; but we have to take care to update this order when it changes. We have observed already that two segments change their order at their intersection point. This seems easy enough to handle: In addition to the top and bottom endpoints of all segments, we add all intersection points to the set of event points, and we update the order of the two intersecting segments at each intersection point. However, after closer inspection, we realize that we have created a chicken-and-egg situation: In order to generate our event schedule, we need to know all the intersection points; but computing the intersection points is the goal of our algorithm.

The way out of the dead-lock is to use a second data structure that allows us to generate the event schedule on the fly, instead of having to know all the event points in advance. In particular, we do away with the presorting of all event points and instead insert them into a priority queue Q. As we identify intersection points, we will insert them as event points into Q, thereby modifying our event schedule.

Now that we know how to generate the event schedule, let us address the central question: How do we detect intersections, in order to report them and insert them into the event schedule? What kind of queries do we ask?

The first observation we make is that two segments that intersect must be stored in the sweep-line structure simultaneouly at some point. Indeed, otherwise, there would exist a horizontal line separating the two segments from each other; see Figure 2.8. So the natural strategy would be to check for intersections between a segment s_i and all other segments in the sweep line structure at the time when the sweep line passes the bottom endpoint of s_i.

This certainly leads to a correct algorithm; but it is not particularly efficient: we may perform $\Omega(n^2)$ intersection tests only to realize that there are no intersections at all. See Figure 2.9 for an illustration.

To avoid performing all those intersection tests in vain, we need a stronger characterization of the set of segments that may potentially intersect. The following lemma provides such a characterization:

Lemma 2.1 *If two segments s_i and s_j intersect, they are stored consecutively in the sweep-line structure at some point before the sweep-line passes their intersection point.*

Proof. Let $E = P \cup I$, where P is the set of segment endpoints and I is the set of intersection points. Let p be the intersection point of s_i and s_j, and let q be the highest point in E with $y_q < y_p$. Between y_q and y_p, the sweep-line status does not change; there are no event points in this range. Now assume for the sake of contradiction that s_i and s_j are not stored consecutively in T. Then there is at least one segment s_k that is stored between s_i and s_j; that is, at y-coordinate y_q, s_k intersects the sweep line between s_i and s_j. Since there is no point in E whose y-coodinate is strictly between y_q and y_p, we know that s_k does not end in the region bounded by s_i, s_j, and the line $y = y_q$; see Figure 2.10. Thus, s_k must intersect at least one of s_i and s_j. Again because E does not contain a point whose y-coordinate is strictly between y_q and y_p, this intersection point must have y-cordinate y_p; that is, s_k contains point p. This contradicts our assumption that no three segments intersect in the same point.

By Lemma 2.1, it is sufficient to test segments for intersection when they become adjacent in the (a, b)-tree. This may happen because of three things: One of the two segments is inserted into the tree because the sweep line passes its bottom endpoint. A segment that was between them is deleted because the sweep line passes its top endpoint. Or one of the two segments intersects one of the previous neighbours of the other segment. Since these three events are already part of our event schedule, it is easy enough to incorporate the intersection tests into our algorithm. There is a small technicality, though. Two segments may become adjacent many times. In Figure 2.11, segments s_1 and s_2 become adjacent three times: when s_2 intersects s_3, when the sweep line passes the top endpoint of s_4, and when s_1 intersects s_5. We do not want to report the same intersection point multiple times, nor do we want to have multiple copies of the same event point in our schedule.

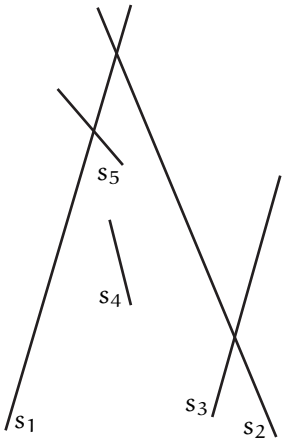


Figure 2.11. Segments s_1 and s_2 become adjacent more than once.

The latter problem we fix by removing intersection points from the event schedule when segments stop being adjacent before the sweep line has passed their intersection point. This is fine because we know, by Lemma 2.1, that the two segments will become adjacent again before they intersect. We fix the former problem by reporting intersections not when they are detected, but at the time when the sweep line passes the intersection point; this can happen only once. So the final algorithm looks as shown on the next page.

Procedure LINE-SEGMENT-INTERSECTION only provides the framework for our algorithm. The priority queue Q provides the event schedule. The loop in Lines 4–12 processes event points one at a time until there are no more event points left. For every event point, we determine what kind of event point it is and then invoke the correct processing procedure. The real work of updating the sweep-line structure T and the event schedule Q is done inside these processing procedures, which look as follows:

Procedure PROCESS-INTERSECTION-POINT is invoked for every intersection point p. This procedure has to achieve three things: report the intersection, switch the order of the two intersecting segments, and detect intersections between segments that have become adjacent in T as a result of the swap.

Line 2 takes care of reporting the intersection between the two segments s_i and s_j that intersect in point p. Lines 3 and 4 perform the swap of segments s_i and s_j. But wait a second. Don't we get exactly the same tree if we first delete s_j and then re-insert it immediately after deleting it? Let us postpone this discussion until a little later because it is connected to a rather fundamental issue with maintaining the sweep-line structure T. Let us assume for now that these lines do indeed achieve the swap. Then s_j now has a new predecessor s_k, and s_i has a new successor s_l in T. We find these two new neighbours in Lines 5 and 6, and we invoke procedure TEST-INTERSECTION on the segment pairs (s_k, s_j) and (s_i, s_l). This procedure, whose pseudo-code is shown on the next page, tests whether its two arguments s_i and s_j intersect and, if so, inserts the intersection point as a new event point into Q.

Procedure PROCESS-BOTTOM-ENDPOINT first inserts the new segment s_i into T. Then it identifies the predecessor s_j and successor s_k of s_i. If neither of them is NIL, then s_j and s_k were adjacent before the insertion of s_i. If they intersect, their intersection is scheduled as an intersection point in Q, and, as discussed earlier, this intersection point should for now be deleted from the event schedule. This is what we do in Line 5. Finally, the insertion of s_i has created two new pairs of adjacent segments: (s_j, s_i) and (s_i, s_k). We invoke procedure TEST-INTERSECTION

```
LINE-SEGMENT-INTERSECTION(S)
 1  Create an empty (a, b)-tree T to serve as the sweep-line structure.
 2  Create an empty priority queue Q to maintain the event schedule.
 3  Insert the endpoints of all segments in S into Q.
 4  while Q is not empty
 5       do p ← DELETE-MIN(Q)
 6          if p is an intersection point
 7             then PROCESS-INTERSECTION-POINT(T, Q, p)
 8             else ▷ p is the top or bottom endpoint of a segment s_i
 9                  if p = p_i
10                     then PROCESS-BOTTOM-ENDPOINT(T, Q, s_i)
11                     else ▷ p = q_i
12                          PROCESS-TOP-ENDPOINT(T, Q, s_i)
```

```
PROCESS-INTERSECTION-POINT(T, Q, p)
1  Let s_i and s_j be the two segments that intersect in p,
   with s_i preceding s_j in T.
2  Report the intersection between s_i and s_j.
3  DELETE(T, s_j)                    ▷ Delete s_j to the right of s_i.
4  INSERT(T, s_j)                    ▷ Insert s_j to the left of s_i.
5  s_k ← PREDECESSOR(T, s_j)
6  s_l ← SUCCESSOR(T, s_i)
7  TEST-INTERSECTION(T, Q, s_k, s_j)
8  TEST-INTERSECTION(T, Q, s_i, s_l)
```

```
PROCESS-BOTTOM-ENDPOINT(T, Q, s_i)
1  INSERT(T, s_i)
2  s_j ← PREDECESSOR(T, s_i)
3  s_k ← SUCCESSOR(T, s_i)
4  if s_j ≠ NIL, s_k ≠ NIL, and s_j and s_k intersect in a point q
5     then DELETE(Q, q)
6  TEST-INTERSECTION(T, Q, s_j, s_i)
7  TEST-INTERSECTION(T, Q, s_i, s_k)
```

```
PROCESS-TOP-ENDPOINT(T, Q, s_i)
1  s_j ← PREDECESSOR(T, s_i)
2  s_k ← SUCCESSOR(T, s_i)
3  DELETE(T, s_i)
4  TEST-INTERSECTION(T, Q, s_j, s_k)
```

```
TEST-INTERSECTION(T, Q, s_i, s_j)
1  if s_i ≠ NIL, s_j ≠ NIL, and s_i and s_j intersect in a point q
2     then INSERT(Q, q)              ▷ Store pointers to s_i and s_j with q.
```

on these two pairs to ensure that potential intersections are inserted into the event schedule.

Procedure PROCESS-TOP-ENDPOINT identifies the two neighbours of s_i—s_j and s_k—and then deletes s_i. After the deletion of s_i, s_j and s_k become adjacent. So we invoke procedure TEST-INTERSECTION on them again to detect the possible intersection between them. Note that we do not have to delete any event points for intersections between s_j and s_i or between s_i and s_k. This is so because, if these segments do indeed intersect, the intersection points precede the top endpoint of s_i in the y-order and, therefore, have been removed from the event schedule (and processed) already.

We have argued that we process every event point correctly. So let us return to the rather suspicious-looking Lines 3 and 4, which are supposed to achieve a swap of s_i and s_j. The reason why these two lines do indeed achieve a swap, at least if we fill in the right details, is connected to the question how we search for a segment in T. Recall our discussion of (a, b)-trees in Chapter 1. In this chapter, we conveniently assumed that all keys are numbers, that is, we implicitly assumed that all keys are drawn from a total order. (Recall that a total order is a set where any two elements are comparable; that is, a comparison tells us that one element is smaller than the other or that the two elements are equal.) Now we store segments in T, and the set of segments is not a total order: at certain y-coordinates one segment is to the left of the other, and at other y-coordinates the order is reversed. We worked around this problem by defining the left-to-right order as the order in which segments intersect the sweep line. Thus, in order to be able to search for the correct location of a segment in T, we have to provide the y-coordinate of the sweep line to the FIND procedure. And this is where the magic happens in Lines 3 and 4. Segment s_j was to the right of s_i just below the current sweep line; segments s_i and s_j intersect the sweep line in the same point, namely their intersection point; and above the current sweep line, segment s_j is to the left of s_i. Hence, if we provide a y-coordinate minimally above the sweep line to the INSERT procedure in Line 4, it will insert s_j to the left of s_i, as desired.

This concludes the discussion of our line segment intersection algorithm. Next we prove its correctness and analyze its running time. First the correctness proof:

Lemma 2.2 *Procedure* LINE-SEGMENT-INTERSECTION *reports all intersections between segments in* S.

Proof. First observe that we do not report intersections that do not exist. This is true because we report intersections at intersection points in the event schedule. These intersection points are in the event schedule because of an intersection at exactly this point that was detected by procedure TEST-INTERSECTION.

The fact that we do not fail to report any intersection points that exist follows almost immediately from Lemma 2.1. Indeed, the proof of this lemma establishes that two intersecting segments s_i and s_j are adjacent in T immediately before the sweep line passes their intersection point q. Every time s_i and s_j become adjacent, we insert q into Q. Since s_i and s_j are adjacent in T immediately before the sweep line passes q, we do not delete q from Q between the last time s_i and s_j became adjacent and the time when the sweep line passes q. Thus, q is retrieved from the event schedule at this point, and the intersection is reported.

Now let us analyze the running time.

Lemma 2.3 *The running time of procedure* LINE-SEGMENT-INTERSECTION *is* $O((n+t)\lg n)$.

Proof. It is easy to see that the first two lines of procedure LINE-SEGMENT-INTERSECTION take constant time. In Line 3, we perform $2n$ INSERT operations on a binary heap, one per segment endpoint. This takes $O(n \lg n)$ time. The cost of every iteration of the while-loop in Lines 4–12 is dominated by the call to procedure DELETE-MIN in Line 5 and by the invocation of one of the three event point processing procedures in Line 7, 10, or 12. The DELETE-MIN procedure takes $O(\lg n)$ time. Each of the event point processing procedures performs at most four standard operations on T and at most three updates of the priority queue Q, inside procedure TEST-INTERSECTION or in Line 5 of procedure PROCESS-BOTTOM-ENDPOINT. Thus, each event point processing procedure takes $O(\lg n)$ time, and one iteration of the while-loop in procedure LINE-SEGMENT-INTERSECTION takes $O(\lg n)$ time. The number of iterations is equal to the number of processed event points, which is $2n + t$: $2n$ segment endpoints and t intersection points. Thus, the total cost of the while-loop is $O((n+t)\lg n)$. Since the preprocessing before the loop takes $O(n \lg n)$ time, the total time of the procedure is $O((n+t)\lg n)$.

Note that this running time is worse than the $O(n \lg n + t)$ time bound we were able to achieve for the orthogonal case. It is also worse than the naïve $O(n^2)$ time algorithm if $t = \omega(n^2/\lg n)$. An $O(n \lg n + t)$ time solution can be obtained; but significantly more subtle ideas are needed.

2.4 Chapter Notes

The orthogonal line segment intersection algorithm is a special case of the red-blue line segment intersection problem, where one is given a set B of blue segments and a set R of red segments so that no two blue segments and no two red segments intersect; the goal is to find all intersections between red and blue segments. An algorithm that solves this problem in $O(n \lg n + t)$ time was proposed by Mairson and Stolfi (1988). The general line segment intersection algorithm is due to Bentley and Ottmann (1979). The trick of removing event points defined by segments that are no longer adjacent in the sweep-line structure is due to Pach and Sharir (1991), which also guarantees that the space used by the algorithm is $O(n)$. Solutions to the general line segment intersection problem that run in $O(n \lg n + t)$ time have been obtained by Chazelle and Edelsbrunner (1988, Chazelle and Edelsbrunner (1992), Clarkson and Shor (1989), Mulmuley (1988), and Balaban (1995). This list represents a progression in the sense that Chazelle and Edelsbrunner's algorithm requires $O(n + t)$ space. The algorithms of Clarkson and Shor, and Mulmuley reduce the space to $O(n)$, but make use of randomization. Balaban's algorithm finally ended the quest by requiring $O(n)$ space and being deterministic. There is a large number of other geometric problems that can be solved using the plane-sweep paradigm. Excellent introductory texts to the area of computational geometry are the text books of Preparata and Shamos (1985) and of de Berg, van Kreveld, Overmars, and Schwarzkopf (1997, 2000).

Chapter 3
Dynamic Order Statistics

In class, we have already discussed how to find the **k-*th order statistic***, that is, the k-th smallest element among a set S of n numbers in linear time. Now consider the scenario when the set S is changing; that is, we insert new numbers into S and remove numbers from S. Whenever we update the set S, it seems costly to spend linear time to recompute the order statistic we are interested in. So we want to store the set S in a data structure T that can answer $\textsc{Select}(T, k)$ queries—that is, queries of the type "Find the k-th order statistic in the current set."—quickly and that can be updated quickly as we insert elements into S or remove elements from S. A closely related query we also want the structure to answer quickly is a ***rank*** query: Given an element x, count the number of elements in S that are less than x, and add one. Intuitively, this gives us the position where element x would be stored if we were to sort the elements in S and then insert x into S in the leftmost position where x can be stored while keeping $S \cup \{x\}$ sorted.

After giving a formal definition of the problem we want to solve, which we do in Section 3.1, we will motivate this problem by discussing a few problems where having a data structure that allows us to compute the rank of any element quickly helps us to design fast algorithms, using the data structuring paradigm. Section 3.2 is dedicated to this. The rest of the chapter then deals with developing a data structure that solves the dynamic order statistics problem.

3.1 Definition of the Problem

Before describing a data structure for the problem we have already described informally, let us give a formal definition of the problem. For a set S and an element x, we define the ***rank*** of x to be

$$\text{rank}(x) = 1 + |\{y \in S : y < x\}|.$$

The **k-*th order statistic*** in S is the element $x \in S$ with rank k.

Our goal is to store S in a data structure that supports the following operations in $O(\lg n)$ time:

$\textsc{Insert}(S, x)$**:** Add element x to S.

$\textsc{Delete}(S, x)$**:** Remove element x from S.

$\textsc{Rank}(S, x)$**:** Determine the rank of element x in S. (Note that x may or may not be in S.)

$\textsc{Select}(S, k)$**:** Determine the k-th order statistic of S.

3.2 Counting Problems

We say that a problem is a ***counting problem*** if we are given a set of elements, S, and we want to count the number of elements in S that satisfy a certain condition.

Determining the rank of an element x is just one kind of counting problem: we want to count the elements in S that are less than x. In this section, we discuss a number of more interesting counting problems, which can all be reduced to the dynamic order statistics problem.

3.2.1 Counting Line-Segment Intersections

In Sections 2.1 and 2.3, we studied the problem of reporting all intersections between n line segments. In some cases, we may not be interested in outputting all the intersections, but just in counting how many intersections there are. In this section, we try to obtain an efficient algorithm for this problem in the case when the line segments are orthogonal. We do not study the general case because it requires insights beyond the scope of this course. Similar to the definition of the orthogonal line segment intersection problem (Problem 2.1 on page 28), the problem we want to solve is the following:

Problem 3.1 (Orthogonal line-segment intersection counting) *Given a set of vertical line segments, $V = \langle v_1, v_2, \ldots, v_k \rangle$, and a set of horizontal line segments, $H = \langle h_1, h_2, \ldots, h_m \rangle$, compute the number of pairs (v_i, h_j) such that v_i and h_j intersect.*

Clearly, we could just use the algorithm from Section 2.1 to solve this problem in $O(n \lg n + t)$ time. But this is not very efficient in general. In the case of the line segment intersection problem, our goal was to report all intersections. Thus, if there are t intersections, we had no choice but to spend $\Omega(t)$ time to report them; that is, our running time was lower-bounded by the output size. In the counting version of the problem, the output size is 1: we are looking for one single number, the number of intersections. Thus, there is no reason why the algorithm should not take $O(n \lg n)$ time, no matter how many intersections there are. However, the line segment intersection algorithm from Section 2.1 takes $O(n \lg n + t)$ time, no matter whether we report the intersections or only count them.

Nevertheless, the algorithm is a good starting point. In fact, it almost gives us the final solution to the intersection counting problem. Consider the line segment intersection algorithm from page 30 again. We argued in the proof of Theorem 2.1 that the running time of procedure ORTHOGONAL-LINE-SEGMENT-INTERSECTION is $O(n \lg n)$, except for the cost of the range queries we ask in Line 6. We ask n of these queries, and their total cost is $O(n \lg n + t)$. In particular, a single query for a horizontal segment h_j costs $O(\lg n + t_j)$ time, where t_j is the number of segments intersecting h_j.

The reason why a single range query costs $O(\lg n + t_j)$ time is that the query has to traverse the whole set of vertical segments whose x-coordinates are in the range $[x_j^l, x_j^r]$. Now we do not want to report them, but only count them. To simplify the problem, let us assume "general position" again. In this case, what we mean is that no two segment endpoints have the same x-coordinate. Then the number of segments that intersect h_j is the number of segments in V_j that have an x-coordinate less than x_j^r, but greater than x_j^l. The number of segments in V_j with x-coordinate less than x_j^r equals $\mathit{rank}_{V_j}(x_j^r) - 1$. Since this figure includes the number of segments with x-coordinate less than x_j^l, we need to subtract their number, which is $\mathit{rank}_{V_j}(x_j^l) - 1$. Hence, we have

$$t_j = \mathit{rank}_{V_j}(x_j^r) - \mathit{rank}_{V_j}(x_j^l).$$

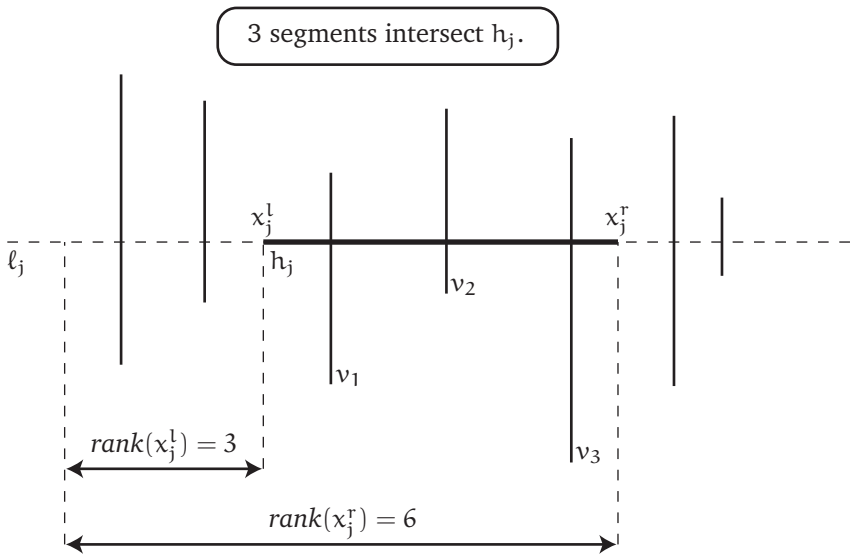


Figure 3.1. The number of segments intersecting h_j is equal to the difference of the ranks of the x-coordinates of the endpoints of h_j in V_j.

See Figure 3.1. This suggests the following algorithm:

ORTHOGONAL-LINE-SEGMENT-INTERSECTION-COUNTING(V, H)

```
 1  count ← 0
    ▷ No intersections found so far.
 2  Create an empty dynamic order statistics structure T to be used as
    the sweep-line structure.
 3  Create a set E of event points.
    ▷ Every event point is a pair (y, v_i) or (y, h_j) that tells us the
    ▷ y-coordinate at which the event occurs and the segment to be
    ▷ processed.
 4  Sort the event points in E by their y-coordinates.
 5  for every event point e in E, in sorted order
 6       do if e = (y, h_j)
 7            then count ← count + RANK(T, x_j^r) − RANK(T, x_j^l)
 8            else ▷ e = (y, v_i)
 9                 if y = y_i^b
10                   then INSERT(T, v_i)
11                   else DELETE(T, v_i)
12  return count
```

Let us prove that the algorithm works and that it takes the desired $O(n \lg n)$ time:

Theorem 3.1 *Using procedure* ORTHOGONAL-LINE-SEGMENT-INTERSECTION-COUNTING*, the orthogonal line segment intersection counting problem can be solved in* $O(n \lg n)$ *time.*

Proof. The running time follows immediately from our discussion: We have argued that procedure ORTHOGONAL-LINE-SEGMENT-INTERSECTION takes $O(n \lg n)$ time, excluding the cost of the n range queries. In procedure ORTHOGONAL-LINE-SEGMENT-INTERSECTION-COUNTING, we have replaced every range query with

two rank queries plus a constant amount of additional computation. Hence, the total cost of Line 7 over all iterations of the loop is $O(n \lg n)$. Since the rest of the algorithm also costs $O(n \lg n)$ time, the total running time of the algorithm is $O(n \lg n)$.

To prove that we report the correct number of intersections, we have to argue that we count the number of intersections correctly for every individual horizontal segment h_j. Note that we ask a query on T at the time when T stores exactly the set V_j of horizontal segments intersecting line ℓ_j. $\textsc{Rank}(T, x_j^l)$ now returns one more than the number of segments that intersect ℓ_j to the left of coordinate x_j^l; $\textsc{Rank}(T, x_j^r)$ returns one more than the number of segments that intersect ℓ_j to the left of coordinate x_j^l. Therefore, $\textsc{Rank}(T, x_j^r) - \textsc{Rank}(T, x_j^l)$ equals the number of segments that intersect ℓ_j to the left of x_j^r, but not to the left of x_j^l. But this is exactly the number of segments that intersect h_j; so we count the number of intersections correctly.

3.2.2 Orthogonal Range Counting and Dominance Counting

Just as the intersection counting problem we studied in Section 3.2.1 is the counting version of the intersection reporting problem, we can define a counting version of the range searching problem. But now let us be a little more ambitious and try to answer 4-sided range counting queries; that is, every query range is now an axis-parallel rectangle, and we want to count the points that are in the rectangle. Here's the definition of the problem:

Problem 3.2 (Orthogonal range counting) *Given a set of n points in the plane, $P = \{p_1, p_2, \ldots, p_n\}$, and a set of m axis-parallel query rectangles, $Q = \{q_1, q_2, \ldots, q_m\}$, report pairs $(q_1, t_1), (q_2, t_2), \ldots, (q_m, t_m)$, where, for all $1 \le i \le m$, t_i is the number of points in P contained in q_i. An axis-parallel rectangle is defined as the region $\{p \in \mathbb{R}^2 : x^l \le x_p \le x^r$ and $y^b \le y_p \le y^t\}$.*

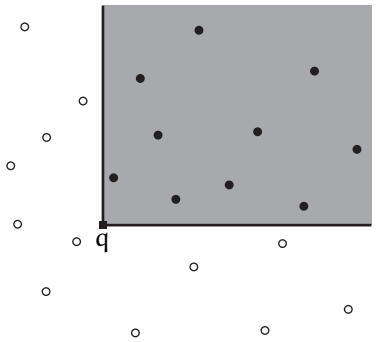


Figure 3.2. The set of points dominating point q is shown in black.

In the orthogonal range counting problem, we have a more restrictive set of query ranges than in the 3-sided case, which makes the problem harder. In order to develop a potential solution, let us first try to solve a simpler problem.

We say that a point p ***dominates*** another point q if

$$x_p \ge x_q \text{ and } y_p \ge y_q;$$

that is, point p is in the north-east quadrant of q (see Figure 3.2). The set of points that dominate a point q is simply the set of points that lie in a 2-sided query range bounded from the left by the line $x = x_q$ and from the bottom by the line $y = y_q$. Now let us consider the following problem:

Problem 3.3 (Dominance counting) *Given a set of n points in the plane, $P = \{p_1, p_2, \ldots, p_n\}$, and a set of m query points, $Q = \{q_1, q_2, \ldots, q_m\}$, report pairs $(q_1, t_1), (q_2, t_2), \ldots, (q_m, t_m)$, where, for all $1 \le i \le m$, t_i is the number of points in P that dominate point q_i.*

Before developing a solution for the dominance counting problem, let us prove the following interesting lemma:

Lemma 3.1 *If there exists an algorithm that solves the dominance counting problem in $T(n, m)$ time, then the orthogonal range counting problem can be solved in $O(m + T(n, m))$ time.*

The proof of this lemma is given by the following algorithm, whose running time is obviously $O(m + T(n, m))$:

ORTHOGONAL-RANGE-COUNTING(P, Q)

```
1  Create a set Q' of points containing all four corners of each query
   rectangle in Q.
2  A' ← DOMINANCE-COUNTING(P, Q')
3  Generate a sequence A of pairs (q, 0), for q ∈ Q.
4  for every answer (q', t') in A'
5       do Let q be the query rectangle such that q' is a corner of q.
6          Let (q, t_q) be the current answer stored in A for q
7          if q' is the bottom-left or top-right corner of q
8            then t'_q ← t_q + t'
9            else t'_q ← t_q − t'
10         Replace (q, t_q) with (q, t'_q) in A.
11 return A
```

If we denote by $\mathrm{dom}_P(p)$ the number of points in P that dominate p, then the algorithm computes for every query range $q \in Q$ the answer

$$\mathrm{dom}_P(a) + \mathrm{dom}_P(d) - \mathrm{dom}_P(b) - \mathrm{dom}_P(c),$$

where a, b, c, and d are the bottom-left, bottom-right, top-left, and top-right corners of q, respectively (see Figure 3.3). The following lemma shows that this produces the correct answer.

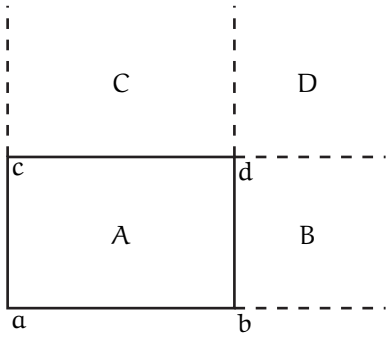


Figure 3.3. Reduction of orthogonal range counting queries to four dominance counting queries.

Lemma 3.2 *For any range query q with bottom-left corner a, bottom-right corner b, top-left corner c, and top-right corner d, $t_q = \mathrm{dom}_P(a) + \mathrm{dom}_P(d) - \mathrm{dom}_P(b) - \mathrm{dom}_P(c)$.*

Proof. For a region R, we denote the number of points in P that lie in this region by $|R|$. Referring to Figure 3.3, we observe that

$$\begin{aligned} t_q &= |A| \\ &= (|A| + |B| + |C| + |D|) - (|B| + |D|) - (|C| + |D|) + |D| \\ &= \mathrm{dom}_P(a) - \mathrm{dom}_P(b) - \mathrm{dom}_P(c) + \mathrm{dom}_P(d). \end{aligned}$$

By Lemma 3.1, it suffices to develop a fast solution to the dominance counting problem. In particular, we want to have an algorithm that takes $O((n+m)\lg(1+\min(n, m)))$ time. To obtain this algorithm, we employ a plane sweep again: by answering rank queries at the right time during this sweep, we are able to determine the number of points in P, t_q, dominating each query point q.

So assume that our sweep-line status is a dynamic order statistics structure T that stores the points in P and allows rank queries based on their x-coordinates. Such a rank query tells us how many points in P are to the left of the query point. Since we know the total number of points in P, we can also determine the

number of points that are to the right of the query point. So how do we count only the points that dominate a query point q_i? We ask a rank query on the set P_i of points in P that are above point q_i and then subtract the answer from the total number of points in P_i. The strategy should be pretty clear now: We sweep a sweep line from top to bottom, maintaining the set of points above the sweep line in a dynamic order statistics structure T and also maintaining the count of the number of points above the sweep line. When the sweep line passes a point in P, we insert the point into T and increase the point count by one. When the sweep line passes a point in Q, we ask a rank query on T and subtract the answer from the current point count. The pseudo-code looks as follows:

```
DOMINANCE-COUNTING(P, Q)
 1  Create an empty order statistics structure T.
 2  c ← 0
 3  S ← P ∪ Q
 4  Sort S by decreasing y-coordinates
 5  for every point p in S, in sorted order
 6       do if p ∈ P
 7            then INSERT(T, p)
 8                 c ← c + 1
 9            else ▷ p ∈ Q
10                 t_p ← c − RANK(T, p)
11                 Add the pair (p, t_p) to the output.
```

The correctness of this procedure follows from our discussion. The running time is $O((n+m)\lg(n+m))$. In particular, the sorting step in Line 4 costs as much. The loop in Lines 5–11 has $n+m$ iterations. In every iteration, we perform a constant amount of work plus an INSERT or RANK operation on T. Since T stores at most n points, the total cost of the loop is therefore $O((n+m)\lg n)$. This is dominated by the sorting cost.

While this is quite good, it is not the best we can do. In particular, if we are to ask only a single dominance query, we should not expect to spend $O(n \lg n)$ time. Reading through the list of points in P and counting those that dominate the query point would take only $O(n)$ time. Conversely, if we have only one point, and we want to determine for every query point whether this single point dominates it, we can easily do so in $O(n)$ time. So what we want is an algorithm that is adaptive to these two situations. We want to replace the $\lg(n+m)$ factor with $\lg(1+\min(n,m))$. This is rather easy:

If $m > n$, we divide the query set into $\lceil m/n \rceil$ sets of size at most n. Then we process one query set at a time to answer the queries in this set. Since a single query set has size at most n, we can answer the queries in this set in $O(n \lg n)$ time. Summing this over all query sets, the total running time is

$$\begin{aligned} O\left(\left\lceil \frac{m}{n} \right\rceil n \lg n\right) &\le O\left(\left(1+\frac{m}{n}\right) n \lg n\right) \\ &= O((n+m)\lg n). \end{aligned}$$

If $m < n$, we divide the point set into $\lceil n/m \rceil$ sets of size at most m and apply the same trick the other way around, thereby obtaining an $O((n+m)\lg m)$ query time. This proves the following result:

Theorem 3.2 *The dominance counting and 4-sided range counting problems can be solved in* $O((n+m)\lg(1+\min(m,n)))$ *time, where* n *is the number of points and* m *is the number of queries.*

3.3 The Order Statistics Tree

The previous section demonstrates that having a structure for the dynamic order statistics problem is useful. The rest of this chapter is devoted to developing such a structure. Let us focus on developing a data structure that supports INSERT, DELETE, and RANK operations efficiently. Once we have such a data structure, we will see that we can support SELECT queries just as efficiently as RANK queries.

3.3.1 Range Queries?

We start by observing that an (a,b)-tree does not seem too far away from the correct solution to the problem: It supports INSERT and DELETE operations in the desired $O(\lg n)$ time. As for rank queries, we observe that such a query is by definition equivalent to counting all elements in S that are less than the given query element x. If we assume that $x \in S$, then the elements in S that are less than x are exactly those that are stored at leaves to the left of the leaf storing x.

So how could we count these leaves? A first attempt would be to answer a range query with query range $(-\infty, x)$ on T; but instead of reporting the elements in the query range, we count them. While this certainly produces the correct answer, it is generally rather inefficient: for each of the rightmost $n/2$ elements, the range query visits at least $n/2$ leaves and therefore takes linear time, a long shot from the $O(\lg n)$ time bound we are aiming for.

3.3.2 An Augmented (a,b)-Tree

If we hope to overcome this linear-time threshold, we cannot count the elements smaller than x individually; we have to gather aggregate information and treat whole groups of elements as one entity if we can verify that all elements in such a group are less than x. This grouping idea is at the core of tree-like search structures and therefore should be easy to realize using an (a,b)-tree. In particular, any search tree T, binary or not, representing a sorted sequence S can be seen as a hierarchical partition of S into smaller subsets, which are treated as atomic entities by different query operations. The root of the tree represents the whole set, any other node in the tree represents the set of all elements stored in its subtree. A FIND operation, for example, can now be seen as "zooming in" until it finds the element it is looking for. Consider an (a,b)-tree. Then all we know initially is that the element may or may not be stored in T. We try to decide this question by considering the root. Since this does not give us enough information, we need to zoom in one level and consider the children of the root. Now observe that the search keys stored at the children of the root give us enough information to determine that all these subtrees, except one, do not contain the given query element. Thus, there is no need to zoom into these any more—we treat them as atomic entities. We only zoom into the subtree that may possibly contain the element we are looking for, by recursing on it.

Now let us return to the order statistics problem. We just observed that, in order to determine the rank of an element x, it suffices to count the elements in T

that are less than x. In other words, we need to count all the elements in subtrees to the left of the search path traversed by query x. Since we want to achieve the same complexity as for a FIND operation, at least within constant factors, we need to be able to count the number of elements in a subtree without zooming into it. The most natural idea to achieve this is to have every node in T *store* the number of elements stored in its subtree; that is, we augment every node v in T to store a $count(v)$ field, which is always equal to the number of leaves in T_v (see Figure 3.4). With this information, a rank query now looks as easy as this, where the initial invocation is RANK$(root(T), x)$:

```
RANK(v, x)
 1  if child(v) = NIL
 2     then
 3          if x > key(v)
 4             then return 2
 5             else return 1
 6     else w ← child(v)
 7          ▷ c is the number of elements less than x in T_v
            c ← 0
 8          while right(w) ≠ NIL and key(right(w)) < x
 9               do c ← c + count(w)
10                  w ← right(w)
11          c ← c + RANK(w, x)
12          return c
```

This procedure is a recursive version of the FIND procedure on page 6, except that Line 9 has been added. Since this increases the cost of every iteration of the while-loop in Lines 8–10 by only a constant factor, the cost is still $O(\lg n)$. Figure 3.4 illustrates the operation of procedure RANK with argument $x = 77$. As procedure FIND, it traverses the black path to leaf i; but at every visited node, it sums the count values of all children that are to the left of the path from the root to leaf i. When it reaches leaf i, it adds 1 to the current sum of these count values because $x \le key(i)$, that is, the leaf itself should not be counted as being less than x. The next lemma shows that the query procedure is correct.

Lemma 3.3 *Procedure* RANK(v, x) *computes the rank of element* x *in* T_v.

Proof. We prove this lemma by induction on the height h of tree T_v. If $h = 0$, then T_v consists of node v, which is a leaf. In this case, we return 1 or 2, depending on whether or not $x > key(v)$. This is obviously the correct answer.

If $h > 0$, let $w_1, w_2, \ldots, w_k$ be the children of v, ordered from left to right. Let w_i be the leftmost child such that either $right(w_i) = \text{NIL}$ or $key(right(w_i)) \ge x$. Then all elements in trees $T_{w_1}, T_{w_2}, \ldots, T_{w_{i-1}}$ are less than x and all elements in trees $T_{w_{i+1}}, T_{w_{i+2}}, \ldots, T_{w_k}$ are no less than x. Thus, we have

$$rank_v(x) = \sum_{j=1}^{i-1} count(w_j) + rank_{w_i}(x),$$

where $rank_u(x)$ denotes the rank of element x w.r.t. the set $Keys(u)$. This is exactly what the invocation RANK(v, x) computes. Thus, it produces the correct answer.

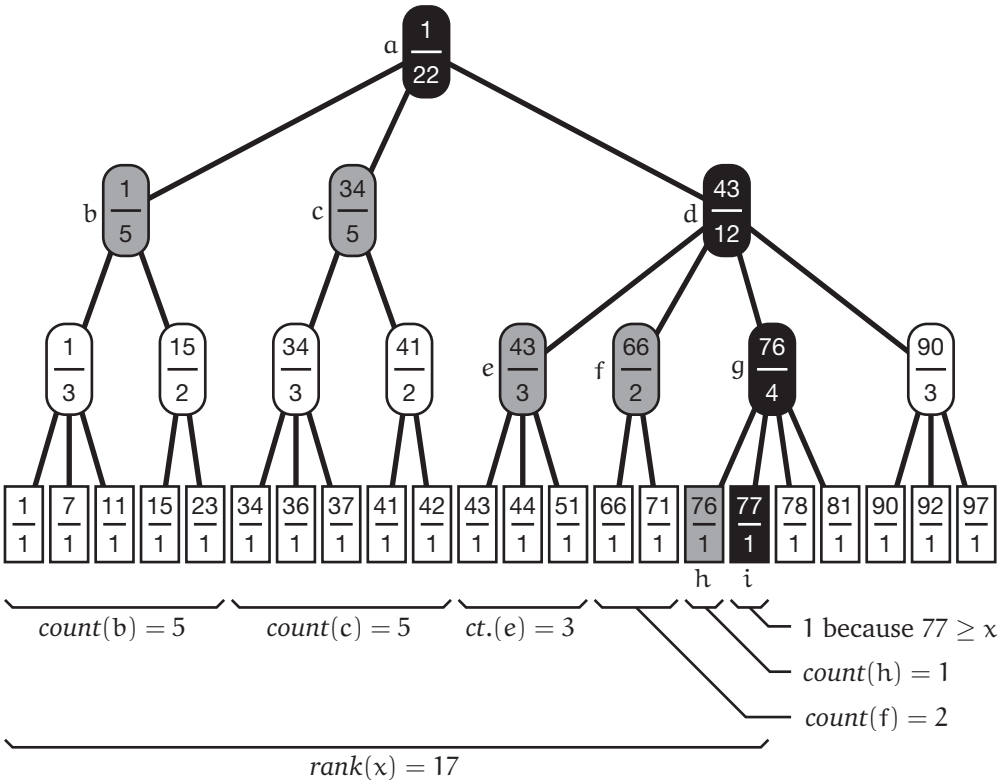


Figure 3.4. An order statistics tree. The top number in every node v is its key, $key(v)$; the bottom number is the number of leaves in its subtree, $count(v)$. The colouring of the nodes shows how procedure RANK answers a rank query with key $x = 77$. It traverses the path of black nodes, sums the count values of the grey nodes and then adds one because x is not greater than the key of the leaf i. Thus, the returned value is 17, which is the correct rank of element 77 in the current set.

3.3.3 Updates

We have argued that storing the number, $count(v)$, of elements in each subtree T_v with v provides us with sufficient information to answer rank queries. However, we want to design a dynamic structure, that is, one that can be updated efficiently after inserting or deleting a new element into or from the set represented by the data structure. We have to argue that we can update the information in our order statistics tree efficiently.

The argument we will apply to do so follows a standard pattern that will recur in the discussion of other augmented data structures in subsequent chapters: Our base structure is an (a, b)-tree. Insertions and deletions into and from an (a, b)-tree are implemented using the following five primitives:

- The FIND operation
- Addition of a new leaf
- Deletion of a leaf
- Node split
- Node fusion

In particular, an insertion uses the FIND operation to locate the place in the data structure where to insert the new element and then adds a new leaf l at this

location. If this increases the degree of l's parent beyond b, the tree is rebalanced using node splits. A deletion removes the leaf l that stores the element to be deleted. If this decreases the degree of l's parent below a, the tree is rebalanced using node fusions, possibly finishing with a node split.

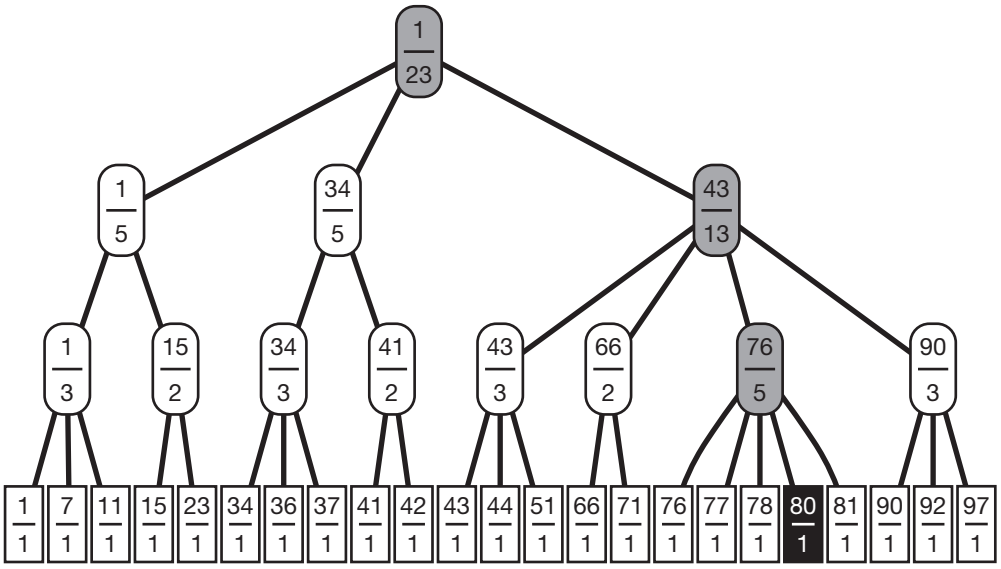
In order to analyze the cost of updating our data structure, we have to argue that we can perform each of these primitives efficiently, that is, that we can maintain the added information efficiently.

Locating an element x. Since the underlying structure of our order statistics tree is a standard (a, b)-tree, it stores all the necessary information to perform a search for a given element x. Hence, as before, we can perform a FIND(T, x) operation in $O(\lg n)$ time. Since this does not change the structure of T, all the information in T remains valid; so we do not have to update anything.

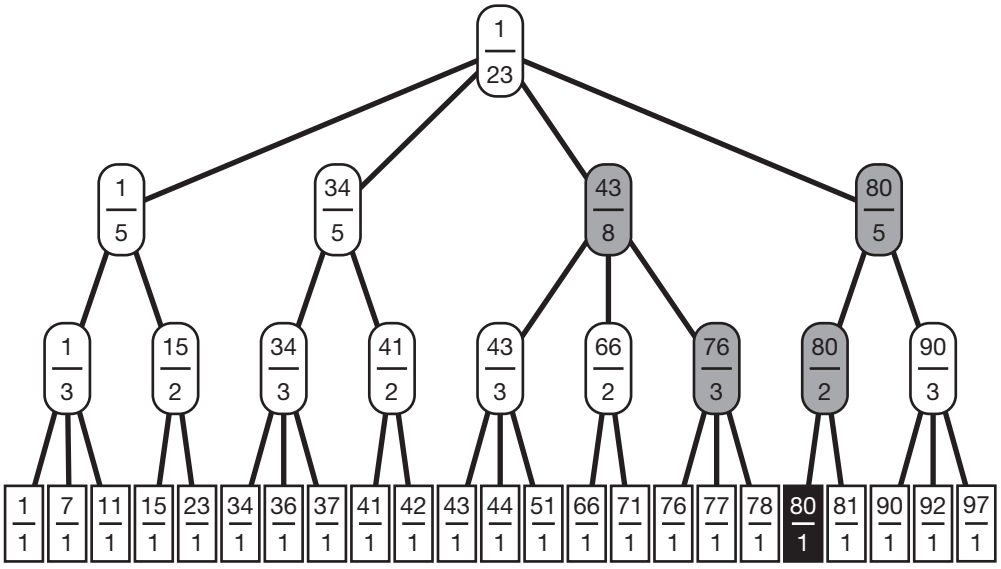
Adding a leaf. The actual addition of the new leaf l is implemented as in a standard (a, b)-tree, which takes $O(1)$ time. However, this addition changes the number of elements in certain subtrees T_v of T. The question is whether we can efficiently identify the nodes v for which this is true, whether there are not too many of them, and whether we can update the information for each of these nodes efficiently.

Obviously, the answer to all three questions is yes: The only subtrees T_v that now store a different number of elements than before are subtrees rooted at ancestors of l (see Figure 3.5(a)). Each such node v gains a new descendant leaf, namely l; that is, $count(v)$ has to be increased by one. Since node l has $O(\lg n)$ ancestors, updating their counts takes $O(\lg n)$ time, by following parent pointers to traverse the path from l to the root of T and increasing $count(v)$ for every node along this path. In more detail, the insertion procedure for an order statistics tree looks as follows:

```
OS-TREE-INSERT(T, x)
 1  v ← FIND(T, x)
 2  Create a new node w
 3  if x < key(v)
 4     then key(w) ← key(v)
 5          key(v) ← x
 6          y ← v
 7     else key(w) ← x
 8          y ← w
 9          child(w) ← NIL
10          count(w) ← 1
11          MAKE-SIBLING(T, v, w)
12          u ← p(w)
13          while u ≠ NIL
14               do key(u) ← key(child(u))
15                  count(u) ← count(u) + 1
16                  if deg(u) > b
17                    then OS-TREE-NODE-SPLIT(T, u)
18                  u ← p(u)
19          return y
```



(a)



(b)

Figure 3.5. (a) The tree produced by inserting element 80 into the tree shown in Figure 3.4. The insertion creates the black leaf storing element 80. The ancestors of this leaf are shown in grey. Their *count* fields are incremented by one, compared to Figure 3.4. (b) The tree obtained by rebalancing the tree in Figure (a) using node splits. The nodes produced by splits are shown in grey.

The only differences to the INSERT procedure for a standard (a, b)-tree are the addition of Line 10, which sets the leaf count of the new leaf to 1, and the addition of Line 15, which increases the leaf count of each ancestor of the new leaf visited by the loop in Lines 13–18 by one.

Deleting a leaf. When a leaf l is deleted, the opposite to a leaf insertion happens (see Figure 3.6(a)): every ancestor v of l loses one descendant leaf, that is, $\mathit{count}(v)$ needs to be decreased by one. Before performing the actual deletion, which takes $O(1)$ time, we can follow parent pointers to traverse the path from l to the root of T and update $\mathit{count}(v)$, for each ancestor v of l. Thus, deleting a leaf takes $O(\lg n)$ time. The details of the deletion procedure are shown below:

```
OS-TREE-DELETE(T, v)
 1  u ← p(v)
 2  REMOVE-NODE(T, v)
 3  while u ≠ NIL
 4        do key(u) ← key(child(u))
 5           count(u) ← count(u) − 1
 6           u′ ← p(u)
 7           if deg(u) < a
 8              then if right(u) = NIL
 9                      then w ← u
10                           u ← left(u)
11                      else  w ← right(u)
12                   FUSE-OR-SHARE(T, u, w)
13           u ← u′
```

As in procedure OS-TREE-INSERT, the only change to a standard (a, b)-tree deletion is the addition of Line 5, which decrements the leaf count of each visited ancestor of the deleted leaf.

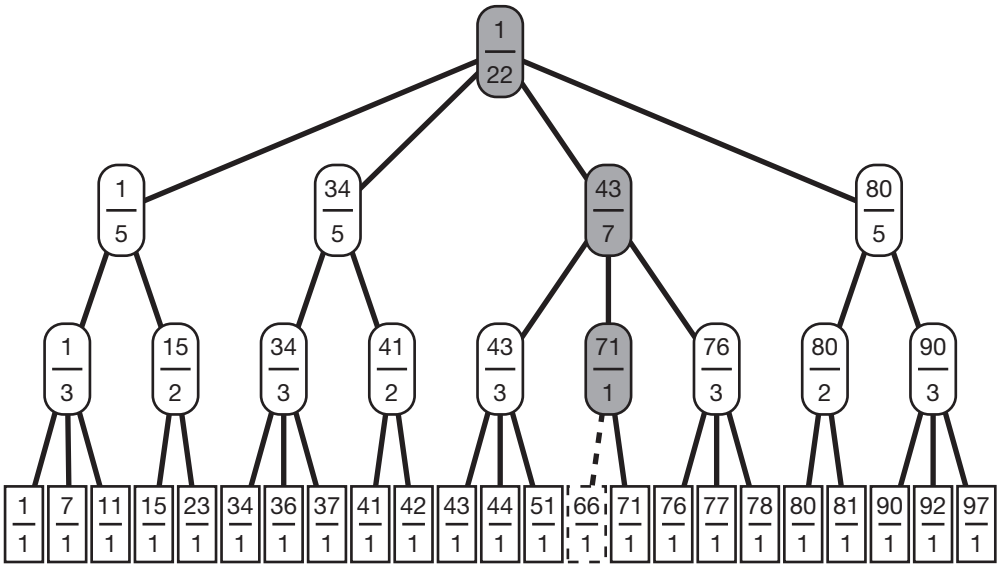
Splitting a node. When a node v is split into two nodes v' and v'', the number of elements in any subtree T_w, $w \notin \{v, v', v''\}$ does not change. Hence, we only have to worry about computing $\mathit{count}(v')$ and $\mathit{count}(v'')$ correctly. Since every element stored in $T_{v'}$ must in fact be stored in a subtree T_w, where w is a child of v', we can compute

$$\mathit{count}(v') = \sum_{i=1}^{k} \mathit{count}(w'_i)$$

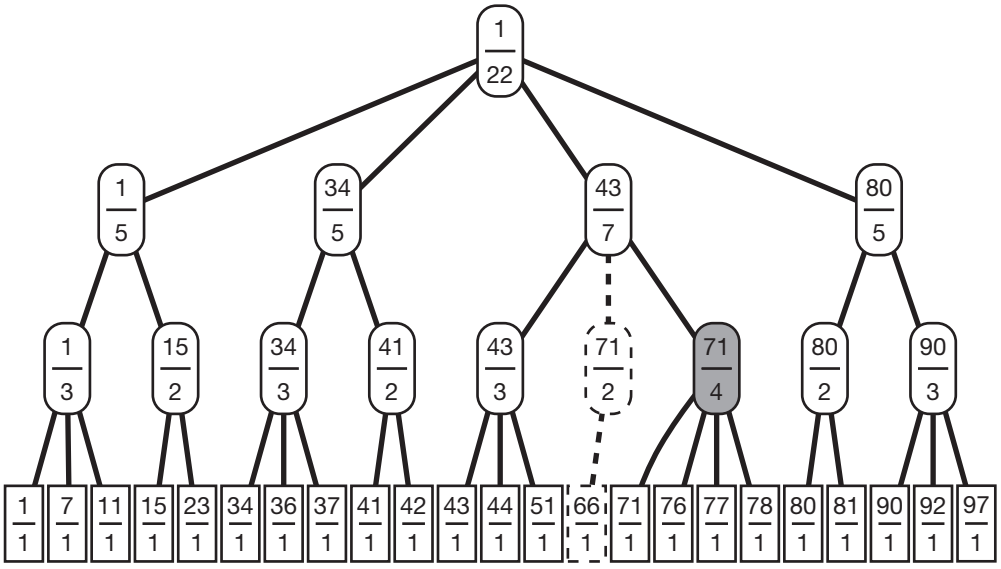
and

$$\mathit{count}(v'') = \mathit{count}(v) - \mathit{count}(v'),$$

where $w'_1, w'_2, \ldots, w'_k$ are the children of v'. In other words, we can traverse the lists of children of v' and sum up their *count* values to compute $\mathit{count}(v')$; the descendant leaves of v'' are exactly those descendant leaves of v that are not descendant leaves of v', which is reflected in the above formula for $\mathit{count}(v'')$. This computation takes time linear in the number of children of v'. Since every child of v' or v'' was previously a child of v, the total number of children of v'



(a)



(b)

Figure 3.6. (a) The tree produced by deleting element 66 from the tree shown in Figure 3.5(b). The *count* fields of all ancestors of the deleted leaf are decreased by one to account for the loss of this leaf. (b) The tree obtained by rebalancing the tree in Figure (a) using node fusions. The internal node with key 66 is fused into its right sibling, shown in grey, whose key and *count* field change as a result.

and v'' is $b + 1$; that is, a node split takes $O(b)$ time. Figure 3.5(b) shows the tree obtained from the tree in Figure 3.5(a) by rebalancing using node splits. The following is the code for performing a node split in an order statistics tree:

OS-TREE-NODE-SPLIT(T, v)

```
 1  Create a new node w
 2  MAKE-SIBLING(T, v, w)
 3  x ← child(v)
 4  c ← 0
 5  for i ← 1 to ⌈(b + 1)/2⌉
 6       do c ← c + count(x)
 7          x ← right(x)
 8  child(w) ← x
 9  key(w) ← key(x)
10  deg(w) ← ⌊(b + 1)/2⌋
11  count(w) ← count(v) − c
12  count(v) ← c
13  right(left(x)) ← NIL
14  left(x) ← NIL
15  while x ≠ NIL
16       do p(x) ← w
17          x ← right(x)
```

Fusing two nodes. A node fusion does the opposite of a node split: it fuses two nodes v' and v'' into one node v. Again, for any node $w \notin \{v, v', v''\}$, the number of elements in T_w does not change. Hence, we only have to compute $count(v)$. However, every descendant leaf of v was previously a descendant leaf of v' or v'', and every descendant leaf of v' or v'' does become a descendant leaf of v. Therefore, we can compute $count(v)$ as

$$count(v) = count(v') + count(v'').$$

This adds only $O(1)$ to the $O(b)$ time a node fusion in a regular (a, b)-tree takes. Figure 3.6(b) shows the tree obtained from the tree in Figure 3.6(a) by rebalancing using node fusions. The code for performing a node fusion looks as follows:

OS-TREE-NODE-FUSE(T, u, w)

```
 1  x ← child(u)
 2  while right(x) ≠ NIL
 3       do x ← right(x)
 4  left(child(w)) ← x
 5  right(x) ← child(w)
 6  deg(u) ← deg(u) + deg(w)
 7  count(u) ← count(u) + count(w)
 8  x ← child(w)
 9  while x ≠ NIL
10       do p(x) ← u
11          x ← right(x)
12  REMOVE-NODE(T, w)
```

We have now established that all the five primitives that are used by (a, b)-tree insertions and deletions to update the tree can be modified so that the additional

count field of every node can be maintained correctly; and we have argued that this can be done at the expense of increasing the cost of each of these primitives by at most a constant factor. Hence, we obtain the following lemma.

Lemma 3.4 *An order statistics tree supports insertions and deletions in* $O(\lg n)$ *time.*

Proof. An insertion performs a FIND operation, then adds a new leaf, and performs at most one node split per level of the tree. The FIND operation takes $O(\lg n)$ time. Adding the leaf takes constant time. A node split takes $O(b) = O(1)$ time, and the height of the tree is $O(\lg n)$; that is, the total cost of all node splits triggered by an insertion is $O(\lg n)$. Summing these costs, we obtain the claimed insertion cost.

A deletion deletes a leaf, performs at most one node fusion per level and at most one node split. Deleting the leaf takes constant time. A node fusion costs $O(b)$ time, as does the node split. The number of levels of the tree is $O(\lg n)$. Hence, the total cost of all node fusions and splits is $O(\lg n)$, which is also the total cost of the whole deletion.

3.3.4 Select Queries

In the discussion so far, we have ignored the second type of query we want to support: SELECT queries, that is, given a parameter k, report the k-th order statistic in the current set. We conclude this chapter by arguing that the subtree sizes stored at the nodes of T are sufficient to answer this type of query as well.

Recall that the k-th order statistic is the element with rank k. Also recall that, for a given element $x \in S$, its rank is one more than the number of elements stored to the left of the path in T from the root to the leaf storing x. So, in order to answer a SELECT(T, k) query, we have to locate the leaf l of T such that $k - 1$ elements are stored to the left of the path from the root of T to l.

Now consider the root r of T, and let $w_1, w_2, \ldots, w_h$ be the children of r, from left to right. If l is a descendant of w_i, for some i, then all descendant leaves of $w_1, w_2, \ldots, w_{i-1}$ are to the left of the path from r to l, all descendant leaves of $w_{i+1}, w_{i+2}, \ldots, w_h$ are to the right of this path, and at least one descendant of w_i, namely l itself, is not to the left of the path from r to l. Thus, in order for l to be a descendant of w_i, we must have

$$\sum_{j=1}^{i-1} \mathit{count}(w_j) < k$$

and

$$\sum_{j=1}^{i} \mathit{count}(w_j) \geq k.$$

Once we have identified the child w_i of r that satisfies this condition, we know that $k' = \sum_{j=1}^{i-1} \mathit{count}(w_j)$ nodes are to the left of the path from the r to any descendant of w_i, including l. In particular, all descendants of $w_1, w_2, \ldots, w_{i-1}$ are to the left of this path. Hence, l is the leaf of T_{w_i} which, in addition to these k' nodes has $k-k'-1$ leaves of T_{w_i} to its left. This immediately leads to the following recursive procedure to find l. The initial call is with arguments $v = \mathit{root}(T)$ and k.

```
SELECT(v, k)
1  if v is a leaf
2     then return v
3  w ← child(v)
4  k′ ← count(w)
5  while k′ < k
6        do w ← right(w)
7           k′ ← k′ + count(w)
8  return SELECT(w, k − k′ + count(w) − 1)
```

The correctness of this procedure follows from our discussion above. The running time is $O(\lg n)$: Every recursive call brings us one step farther away from the root; that is, we can make at most $O(\lg n)$ recursive calls. The running time of every recursive call is dominated by the cost of the while-loop in Lines 5–7. Every iteration of this loop costs constant time, and there is at most one iteration per child of v, at most b in total. This proves the following lemma.

Lemma 3.5 *A dynamic order statistics tree can answer* SELECT *queries in* $O(\lg n)$ *time.*

In summary, we have the following theorem.

Theorem 3.3 *A dynamic order statistics tree of a set S can be maintained under insertions and deletions and supports* RANK *and* SELECT *queries. The cost of each operation is* $O(\lg n)$*. The space usage of the tree is linear.*

3.4 Chapter Notes

The dynamic order statistics tree discussed in this chapter is an adaptation of the red-black tree based dynamic order statistics tree described by Cormen, Leiserson, Rivest, and Stein (2001). The intersection counting algorithm from Section 3.2.1 is a straightforward adaptation of the intersection reporting algorithm from Section 2.1. The hereditary segment tree of Chazelle, Edelsbrunner, Guibas, and Sharir (1994) can be used to solve the more general red-blue line segment intersection counting problem in $O(n \lg n)$ time. For counting the intersections among n arbitrary line segments, $O(n^{4/3} \lg^{O(1)} n)$-time algorithms have been proposed by Agarwal (1990) and Chazelle (1993). The reduction of 4-sided range counting to dominance counting is folklore. The dominance counting algorithm discussed here is based on the plane-sweep paradigm. For a thorough discussion of other geometric problems solvable using this paradigm see the textbooks by Preparata and Shamos (1985) and de Berg, van Kreveld, Overmars, and Schwarzkopf (1997, 2000).

Chapter 4
Priority Search Trees

The previous chapter has given an introduction to the general procedure of augmenting a data structure. The idea that led to the augmentation was rather transparent, as was the analysis of the running times of the different operations. In this chapter, we will develop more subtle ideas, both in terms of the extra information to be stored at a node and in terms of the analysis of the running times. The result will be a data structure that allows us to answer three-sided range queries over a dynamically changing point set and *interval overlap queries*. During the design phase, we will focus on solving the three-sided range search problem. In order to solve the interval overlap problem, we will show that it is equivalent to a two-sided range search problem. In other words, we simply re-interpret the given data and the given problem. No algorithmic changes are required. This approach is instructive in at least two ways: First, it illustrates the power of the data structuring paradigm: Once we have an efficient data structure, we can often solve more than one problem using this data structure. The second lesson we should learn from this reduction is that it helps (not only when designing algorithms, but when solving any kind of problem) to look at the problem from many different perspectives, some of them not the most obvious ones. Any trick is allowed, as long as it allows us to solve the problem at hand.

In Section 4.1, we define the two problems we want to solve in this chapter. In Section 4.2, we discuss the shortcomings of (a, b)-trees when trying to answer three-sided range queries. This will lead us to the definition of a *priority search tree*, which is the structure we will use to answer these queries. In Section 4.3, we discuss the query procedure. In Section 4.4, we provide procedures for adding and deleting points to and from the priority search tree. In Section 4.5, we show how to use a priority search tree to answer interval overlap queries. The update bounds we achieve in Section 4.4 are not quite the ones we hope to obtain. In Section 4.6, we discuss two methods to achieve the desired update bounds. In the course of this discussion, we introduce the concepts of amortization and weight-balancing.

4.1 Three-Sided Range Searching and Interval Overlap Queries

The two problems we want to solve are of a geometric nature. The first one is the good old three-sided range search problem. Now, however, we want to study it in a dynamic setting, that is, we want to represent the point set S using a data structure that can be updated quickly as points are added to or deleted from S, and we want the data structure to answer three-sided range queries over the current point set quickly. Formally, we want to solve the following problem:

Problem 4.1 (Three-sided range searching) *Given a set of n points S in the plane, store it in a linear-size data structure T that supports the following operations:*

INSERT(T, p)**:** *Add the point p to the set S.*

DELETE(T, p)**:** *Remove the point p from S.*

THREE-SIDED-RANGE-QUERY(T, x^l, x^r, y^b)**:** *Output all points $p \in S$ that satisfy $x^l \leq x_p \leq x^r$ and $y_p \geq y^b$.*

Procedures INSERT *and* DELETE *should take $O(\lg n)$ time. Procedure* THREE-SIDED-RANGE-QUERY *should take $O(\lg n + t)$ time, where t is the number of points reported in answer to the query.*

We assume that no two points in S have the same x- or y-coordinates. The structure we develop can easily be adapted to remove this assumption; but making it simplifies the presentation significantly.

Problem 4.1 is closely related to the more elementary dictionary problem solved by (a, b)-trees. There we want to maintain a set of keys—in other words, points in one-dimensional space—and we want to be able to insert points into the set, delete points from the set, and report points whose values lie in a given query interval. The only change we make now is that the elements have a second dimension and that we want to output only elements that are above a certain threshold y^b in this dimension. The space and query bounds we are asking for are the same as for the dictionary problem.

The second problem we want to solve is seemingly completely unrelated; but, as we will see, it is not.

Problem 4.2 (Interval overlap) *Given a set S of n intervals of the form $[a, b]$, store these intervals in a linear-space data structure that supports the following operations:*

INSERT$(T, [l, r])$**:** *Add the interval $[l, r]$ to the set S.*

DELETE$(T, [l, r])$**:** *Remove the interval $[l, r]$ from S.*

OVERLAP$(T, [l, r])$**:** *Output all intervals $[l', r'] \in S$ that overlap interval $[l, r]$, that is, such that $[l', r'] \cap [l, r] \neq \emptyset$.*

Procedures INSERT *and* DELETE *should take $O(\lg n)$ time. Procedure* OVERLAP *should take $O(\lg n + t)$ time, where t is the number of intervals reported in answer to the query.*

Figure 4.1 illustrates an overlap query. on a set of intervals. The bold intervals overlap the query interval shown as a grey slab and are to be reported in answer to the query. The thin intervals do not overlap the query interval and should not be reported.

4.2 Priority Search Trees

In this section, we follow the basic idea of augmenting data structures in that we set out with the goal of using an (a, b)-tree to solve the three-sided range

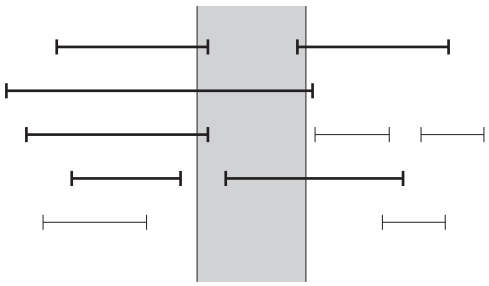


Figure 4.1. An interval overlap query.

searching problem. This seems reasonable because, as discussed in the previous section, an (a, b)-tree supports insertions, deletions, and range queries in the desired time bounds, except that range queries ignore the second dimension. So, by using an (a, b)-tree, we start with a data structure that almost does what we want. Next we identify the problems we encounter when trying to answer three-sided range queries on an (a, b)-tree, and we explore ideas that allow us to overcome these problems, which will produce the priority search tree structure.

4.2.1 Answering Range Queries on an (a, b)-Tree

An (a, b)-tree allows us to answer a range query only on the x-coordinates: Store all the points using their x-coordinates as their keys. Then a standard RANGE-QUERY operation on an (a, b)-tree finds all t points in a given x-range $[x^l, x^r]$ in $O(\lg n + t)$ time. For an illustration, see Figure 4.2. The query outputs all black points; but only the square ones actually lie in the query range.

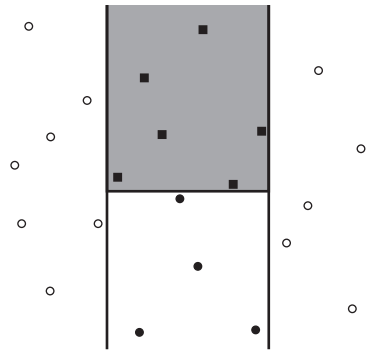


Figure 4.2. A three-sided range query and the corresponding standard range query.

The obvious idea is to adapt the RANGE-QUERY operation so that it takes the lower bound y^b on the y-coordinates as an additional argument; then, before outputting a point, it checks whether the point is above this lower bound; if not, the point is not output. This obviously produces the correct answer because now a point is output if and only if it is in the x-range $[x^l, x^r]$ and above y-coordinate y^b. Unfortunately, the time bound for this procedure is not the one we desire. Do you see why? (Stop reading for a minute and think before you continue.)

Consider Figure 4.3. In this figure, the x-range of the query contains all points stored in T; but none of them is above the bottom y-coordinate of the query, that is, the output size t is in fact zero. Since all points are in the x-range, our modified range-search procedure would inspect all points and spend $\Omega(n)$ time to do so. This is much more than $O(\lg n + t) = O(\lg n)$.

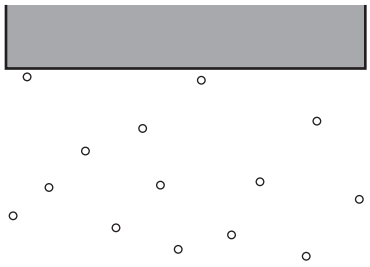


Figure 4.3. The query inspects all points, but outputs none.

4.2.2 Searching by x and y

Now let us consider the query operation in more detail (see Figure 4.4). We can think of the query procedure as traversing two paths P_l and P_r, leading to the leftmost and rightmost points in the x-range, respectively, and then visiting all nodes between these two paths. Traversing paths P_l and P_r costs $O(\lg n)$ time, which is well within our desired query bound. The problem is that there may be many nodes between P_l and P_r, even though we may output only few points in answer to our query. So we cannot afford to visit all nodes between P_l and P_r, unless we have enough output to pay for it—we have to find a way to reduce the number of visited nodes to a number that is proportional to the output size.

In order to figure out a way to do this, let us look at the query from yet another angle. We can partition the set of visited nodes into two sets: those on P_l and P_r

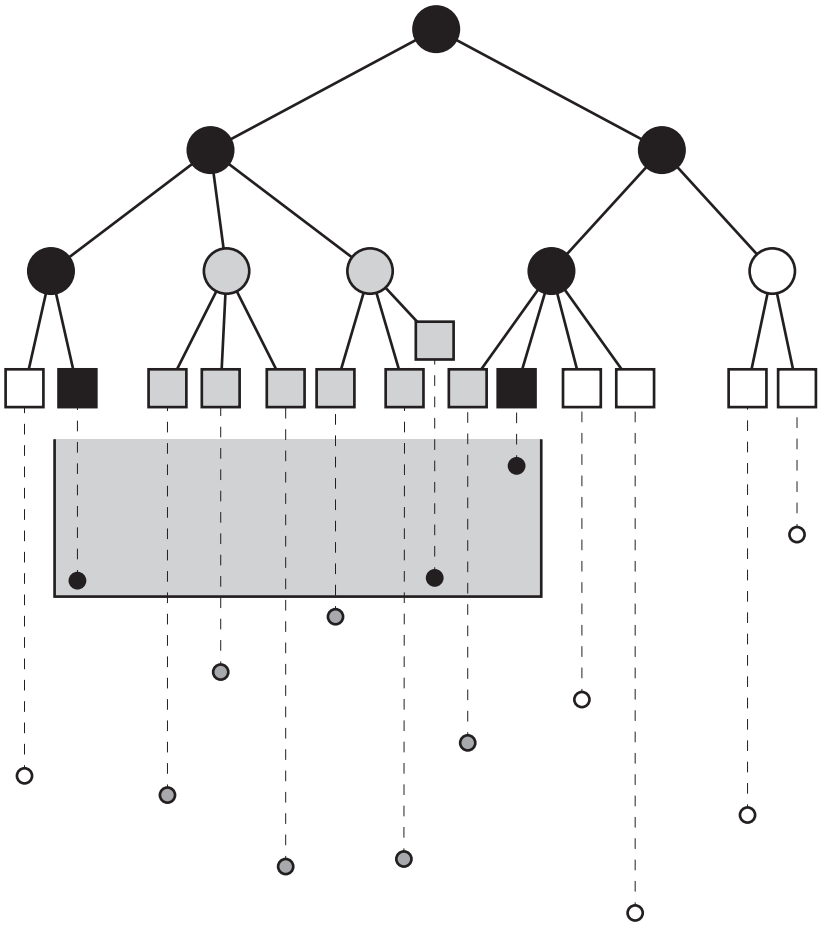


Figure 4.4. The modified range search procedure correctly outputs the black points; but, to do so, it inspects all black and grey points because they are in the x-range of the query. The traversal of the black paths is not problematic. The costly part is visiting the grey nodes.

and those between these two paths. The nodes between P_l and P_r define maximal subtrees $T_1, T_2, \ldots, T_k$ whose roots are children of nodes on P_l or P_r. Since these subtrees are between P_l and P_r, and the leaves of P_l and P_r are respectively the leftmost and rightmost points in the x-range of the query, all points stored in a subtree T_i are guaranteed to be in the x-range of the query. Hence, we need a way to efficiently search among these points based on their y-coordinates.

The y-search we need to perform is far from being a general y-search. Rather we want to report *all* points above our given y-coordinate y^b. Let S_i be the set of points stored in T_i. We can solve the problem of reporting all points in S_i that are above the line $y = y^b$ using many different and mostly simple data structures: If we have the points stored in an array, sorted by decreasing y-coordinates, all we have to do is scan the array and output points until we find the first point with y-coordinate less than y^b. This takes $O(1 + t_i)$ time, where t_i is the number of points in S_i we output.

Another useful data structure is a max-priority queue on the y-coordinates of the points. We can report all the points by performing DELETE-MAX operations until we remove the first point with y-coordinate less than y^b. We perform $t_i + 1$ DELETE-MAX operations, each of which costs $O(\lg n)$ time. Thus, the total cost is $O((t_i + 1) \lg n)$.

There are two things that are not ideal about the priority-queue-based approach: The query bound is $O((t_i + 1) \lg n)$, which is by a factor of $\lg n$ worse than the bound achieved by the array-based solution. And we actually remove all the points we report, which is not what we intended to do. Sure, we could put them back in the structure; but this seems like an ugly solution. In spite of these shortcomings of the priority-queue-based solution, this is the one we will use and which we will refine to overcome its shortcomings.

But why don't we use the array-based solution? The reason is that it would require $O(n \lg n)$ space and is hard to update. Indeed, we have to be prepared to ask a y-query on any subtree of T; that is, we have to store a y-sorted array with every node of T, where node v stores the points in the set S_v of points stored in T_v. Thus, every point p is stored in $O(\lg n)$ lists, namely the ones corresponding to ancestors of the leaf storing p, and the total space bound is $O(n \lg n)$. The priority-queue-based solution has the same shortcoming; but we will see how to overcome this difficulty by turning T into a priority queue that allows us to answer queries on subsets of its nodes.

4.2.3 Using a Priority Queue for y-Searching

Assume for now that we use a binary max-heap H_i as the priority queue storing the points in S_i. Can we reduce the cost of identifying all points above the bottom boundary y^b to $O(1 + t_i)$, while at the same time not modifying the heap at all? To do so, we should have to use the information provided by the heap property. The first obvious test we should perform is whether the root element has a y-coordinate of at least y^b. If not, no point can have a y-coordinate of at least y^b because the root stores the point with maximal y-coordinate; that is, we will have determined in $O(1) = O(1 + t_i)$ time that $t_i = 0$. If, on the other hand, the root element has a y-coordinate of at least b, we output this element and have to inspect the two children of the root. Now the subtrees rooted at these two children are themselves binary heaps. So the same strategy can be applied recursively to these two subtrees to determine which of their nodes should be visited. Thus, our y-search procedure looks as follows:

```
Y-QUERY(v, y^b)
1  p ← key(v)        ▷ The key in this case is a point.
2  if y_p ≥ y^b
3     then output point p
4          if left(v) ≠ NIL
5             then Y-QUERY(left(v), b)
6          if right(v) ≠ NIL
7             then Y-QUERY(right(v), b)
```

This procedure is illustrated in Figure 4.5. The correctness of the procedure follows from our argument above. To bound the running time of the procedure, we observe that every invocation of procedure Y-QUERY, excluding recursive calls, costs constant time. Thus, we obtain the desired running time of $O(1 + t_i)$ if we can argue that we visit only $O(1 + t_i)$ nodes. In the following lemma, we use t_v to denote the number of points in H_v with y-coordinate greater than y^b, where H_v is the subtree of H_i rooted at v. In particular, $t_{root(H_i)} = t_i$.

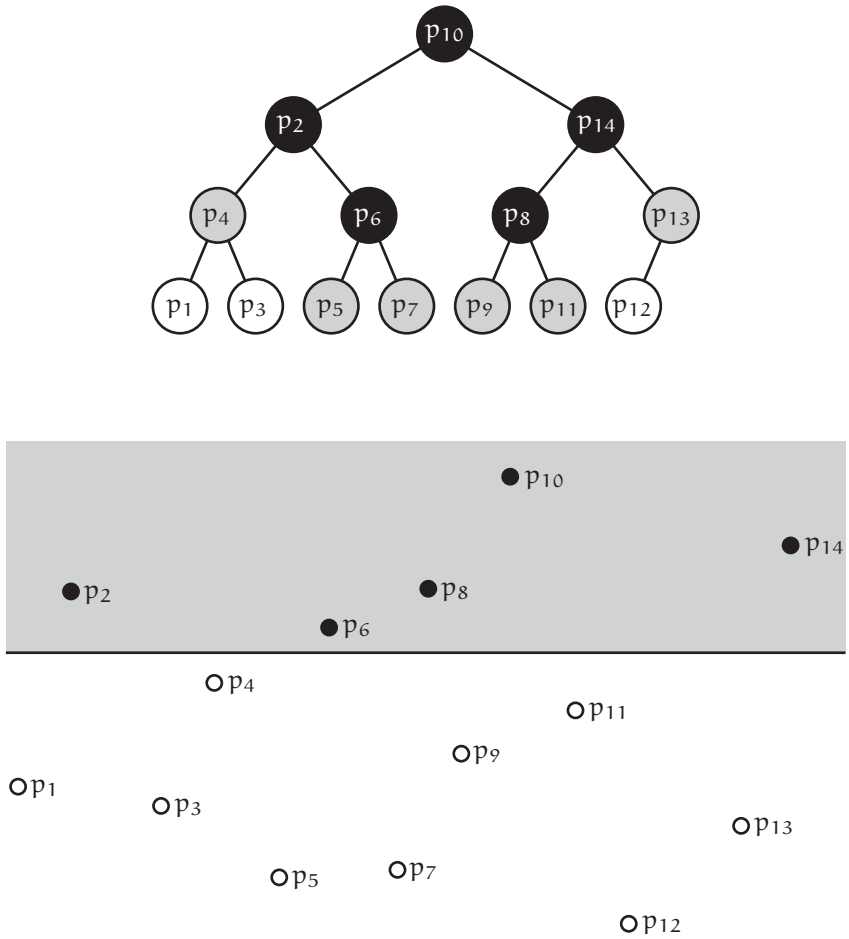


Figure 4.5. The y-search procedure reports only points stored in a subtree of the whole tree whose root is the root of the tree. This tree is shown in black. The nodes that are inspected without outputting any points are adjacent to nodes in the black subtree. They are shown in grey.

Lemma 4.1 *Procedure* Y-QUERY(v, b) *visits at most* $1 + 2t_v$ *nodes.*

Proof. We prove the claim by induction on the number t_v of points in H_v that are output by the procedure. If $t_v = 0$, we do not output the point at v, that is, we do not execute Lines 3–7 and therefore do not make any recursive calls. Hence, the total number of nodes we visit is 1.

So assume that $t_v > 0$ and that the claim holds for any $t < t_v$. If $t_v > 0$, we execute Lines 3–7; that is, we first output $key(v)$ and then recurse on $left(v)$ and $right(v)$. Since $t_v = 1 + t_{left(v)} + t_{right(v)}$, we have $t_{left(v)} < t_v$ and $t_{right(v)} < t_v$. Hence, by the induction hypothesis, the two recursive calls on $H_{left(v)}$ and $H_{right(v)}$ visit at most $1 + 2t_{left(v)}$ and $1 + 2t_{right(v)}$ nodes, respectively. Therefore, the total number of nodes we visit is at most

$$\begin{aligned} 1 + (1 + 2t_{left(v)}) + (1 + 2t_{right(v)}) &= 1 + 2(1 + t_{left(v)} + t_{right(v)}) \\ &= 1 + 2t_v. \end{aligned}$$

Corollary 4.1 *Procedure* Y-QUERY$(root(H_i), y^b)$ *takes* $O(1 + t_i)$ *time.*

What we have achieved so far is to reduce the query bound on a priority queue to $O(1 + t_i)$ time, while not modifying the priority queue in the course of a query. The next subsection addresses the goal of reducing the total space bound to $O(n)$, by combining the search tree T with the priority queues for all its nodes.

4.2.4 Combining Search Tree and Priority Queue

By "combining the search tree and the priority queue" we mean that we want to construct a structure that is at the same time a search tree on the x-coordinates of the points and a max-priority queue on their y-coordinates. If we can achieve this, we can use the following strategy to answer a three-sided range query: First we perform a search on the x-coordinates to identify paths P_l and P_r and thereby the set of maximal subtrees between P_l and P_r. This part of the query procedure uses the fact that the structure is a search tree on the x-coordinates. Then, for each subtree between the two paths, we use the fact that the tree is also a priority queue on the y-coordinates and perform a y-search on the priority queue. The following structure provides such an amalgate of search tree and priority queue:

Definition 4.1 *A* priority search tree T *over a point set* S *in the plane is an* (a, b)*-tree with the following properties:*

(PST1) *There are* n *leaves, each corresponding to one point of* S*. Let* ℓ_p *denote the leaf corresponding to point* p*.*

(PST2) *The leaves are sorted by the* x*-coordinates of their corresponding points, that is, if* $x_p < x_q$*, then* ℓ_p *is to the left of* ℓ_q*.*

(PST3) *Every node* v *of* T *stores at most one point* $\mathit{point}(v)$*.*

(PST4) *Every point* $p \in S$ *is stored at an ancestor of* ℓ_p*.*

(PST5) *If a node* v *stores a point, then so do all its ancestors.*

(PST6) *If* v *is an ancestor of* w*,* $\mathit{point}(v) = p$*, and* $\mathit{point}(w) = q$*, then* $y_p \geq y_q$*.*

Property (PST2) expresses that the tree is a search tree over the x-coordinates of the points in S. Property (PST6) expresses that T is a heap on the y-coordinates of the points in S. Figure 4.6 shows a priority search tree.

Before arguing, in the next section, that this tree structure is useful to answer three-sided range queries, we should ask ourselves whether such a tree can be constructed for *any* point set. As it turns out, this is rather straightforward: We begin by constructing an (a, b)-tree that stores all the points at the leaves, in left-to-right order. Now we inspect the nodes of T by increasing distance from the root, starting with the root itself. For every node v, we consider all the points stored at the leaves of T_v, choose the one with maximal y-coordinate, remove it from its leaf ℓ_p, and store it at v. We remove the point from ℓ_p to ensure that descendants of v exclude p from the candidate set of points to choose from, which ensures that every point is stored only once.

Properties (PST1)–(PST3) are obviously satisfied by the resulting tree. Since we choose the point to be stored at a node v from the points corresponding to its descendant leaves, every point p is stored at an ancestor of ℓ_p, that is Property (PST4) is satisfied. Property (PST5) is satisfied because we fill in the tree from the root towards the leaves. Property (PST6) is satisfied because, for two nodes v

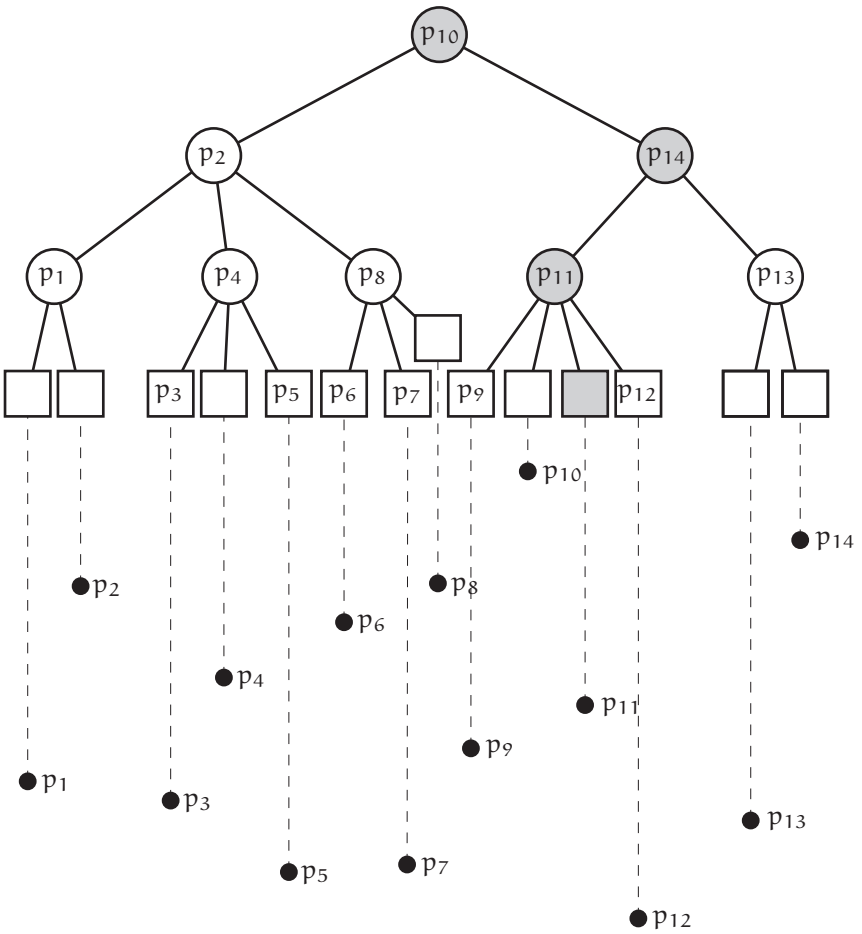


Figure 4.6. An example of a priority search tree.

and w, where v is an ancestor of w, we choose v's point before w's point. Hence, the set of points v can choose from is a superset of the choices w has, that is, the highest point in the set v chooses from cannot be lower than the highest point w can choose. This proves that the resulting tree is indeed a priority search tree.

4.3 Answering Queries

By Properties (PST2) and (PST4), it is still easy to locate a point in T based on its x-coordinate. In particular, if we try to find the leaf corresponding to a given x-coordinate, the standard FIND procedure can be used without modification to find this leaf. The point normally stored at this leaf is now stored somewhere along the path we traverse anyway. In Figure 4.6, for example, point p_{11} can be stored at any node on the grey path, but only there. So all we have to do is modify the FIND procedure so that it also inspects the points stored along the path it traverses, which comes at an increase of the running time by a constant factor.

A three-sided range query now follows the general idea we have already outlined in Section 4.2.4: We locate the leftmost and rightmost points in the x-range of the query. More precisely, we traverse the paths P_l and P_r leading to the leaves corresponding to these two points. For every node v on these two paths, we inspect the point stored at v and output it if it falls in the query range. For every maximal tree T_i between these two paths, we invoke a version of the Y-QUERY

procedure that has been adapted to work on (a, b)-trees. The following is the pseude-code of the THREE-SIDED-RANGE-QUERY procedure:

```
THREE-SIDED-RANGE-QUERY(v, x^l, x^r, y^b)
 1  if point(v) ≠ NIL
 2     then if x^l ≤ x(point(v)) ≤ x^r and y(point(v)) ≥ y^b
 3             then output point(v)
 4          if v is not a leaf
 5             then w ← child(v)
 6                  while right(w) ≠ NIL and key(right(w)) < x^l
 7                     do w ← right(w)
 8                  if right(w) = NIL or key(right(w)) > x^r
 9                    then THREE-SIDED-RANGE-QUERY(w, x^l, x^r, y^b)
10                    else LEFT-HALF-QUERY(w, x^l, y^b)
11                         w ← right(w)
12                         while right(w) ≠ NIL and key(right(w)) ≤ x^r
13                            do Y-QUERY(w, y^b)
14                               w ← right(w)
15                         RIGHT-HALF-QUERY(w, x^r, y^b)
```

For every node v this procedure visits, Line 1 tests whether v stores a point. If not, the whole subtree rooted at v is empty, and we can safely abandon the search. If node v stores a point, Line 2 tests whether this point is in the query range; if so, Line 3 outputs it. If v is a leaf, there is nothing left to do in T_v. Otherwise, we have to decide at which child or children of v to continue the search. Similar to the RANGE-QUERY procedure for regular (a, b)-trees, Lines 5–7 find the leftmost child w that either does not have a right sibling or such that its right sibling has a key greater than x^l. Note in particular that $key(w) < x^l$. If $right(w) = \text{NIL}$ or $key(right(w)) > x^r$, the only subtree possibly storing more points in the query range is the subtree rooted at w. So we continue the search at this node by invoking procedure THREE-SIDED-RANGE-QUERY on w in Line 9. If $right(w) \neq \text{NIL}$ and $key(right(w)) \leq x^r$, we know that every point in T_w has x-coordinate at most x^r. Hence, we can invoke a specialized procedure LEFT-HALF-QUERY on w to report all points in T_w that have x-coordinate at least x^l and y-coordinate at least y^b. This procedure is explained below. For every child w' of v that is to the right of w, we know that all points in $T_{w'}$ have x-coordinate at least x^l because $key(right(w)) \geq x^l$. Thus, as long as $right(w') \neq \text{NIL}$ and $key(right(w')) \leq x^r$, we know that every point in $T_{w'}$ has x-coordinate between x^l and x^r. Hence, we can perform a y-search on $T_{w'}$. In Lines 12–14, we iterate over all these children w' of v until we find the first child w'' such that either $right(w'') = \text{NIL}$ or $key(right(w'')) > x^r$. For w'', we still know that all points in $T_{w''}$ have an x-coordinate of at least x^l; but some of these x-coordinates may be greater than x^r. Hence, Line 15 invokes procedure RIGHT-HALF-QUERY on w'' to identify all points in $T_{w''}$ that have x-coordinates at most x^r and y-coordinates at least y^b. This is similar to procedure LEFT-HALF-QUERY.

The pseudo-code for procedure LEFT-HALF-QUERY is given below. Procedure RIGHT-HALF-QUERY is very similar to this; so we leave it as an exercise to develop the pseudo-code for this procedure.

```
LEFT-HALF-QUERY(v, x^l, y^b)
 1  if point(v) ≠ NIL
 2     then if x(point(v)) ≥ x^l and y(point(v)) ≥ y^b
 3             then output point(v)
 4          if v is not a leaf
 5             then w ← child(v)
 6                  while right(w) ≠ NIL and key(right(w)) < x^l
 7                        do w ← right(w)
 8                  LEFT-HALF-QUERY(w, x^l, y^b)
 9                  w ← right(w)
10                  while w ≠ NIL
11                        do Y-QUERY(w, y^b)
12                           w ← right(w)
```

The reasoning behind procedure LEFT-HALF-QUERY is very similar to that for procedure THREE-SIDED-RANGE-QUERY. Again, we test whether v's subtree is non-empty. If so, we test whether v's point is in the query range and output it. Then, if v is not a leaf, we locate, in Lines 5–7, the leftmost child w where we cannot guarantee that all points in T_w are to the left of the query range. In Line 8, we invoke LEFT-HALF-QUERY recursively on this child. For every child w' to the right of w, all points in $T_{w'}$ have x-coordinates of at least x^l. Hence, in Lines 10–12, we invoke procedure Y-QUERY on each of these children.

Finally, procedure Y-QUERY is easy to adapt so that it works on an (a, b)-tree:

```
Y-QUERY(v, y^b)
1  if point(v) ≠ NIL and y(point(v)) ≥ y^b
2     then output point(v)
3          w ← child(v)
4          while w ≠ NIL
5                do Y-QUERY(w, y^b)
6                   w ← right(w)
```

Figure 4.7 illustrates the operation of these procedures. Now let us prove that procedure THREE-SIDED-RANGE-QUERY(v, x^l, x^r, y^b) reports exactly the points in T_v that fall in the query range. Since the initial invocation is with $v = root(T)$, this implies that we output all the points in S that fall in the query range.

Lemma 4.2 *The invocation* THREE-SIDED-RANGE-QUERY(v, x^l, x^r, y^b) *reports all points in* T_v *that match the query.*

Proof. First observe that we never output a point that is not in the query range. This is obvious for every point that is reported on Line 3 of procedure THREE-SIDED-RANGE-QUERY because we test explicitly whether this point is in the query range. For a point reported inside procedure LEFT-HALF-QUERY, we have argued above that this point must have an x-coordinate of at most x^r. Hence, testing the other two sides is sufficient. Using a similar argument, we observe that every point reported by procedure RIGHT-HALF-QUERY must be in the query range. Finally, whenever we invoke procedure Y-QUERY on a node w, we have argued above that all points in T_w are in the x-range of the query, which implies that testing their y-coordinates before reporting them is sufficient.

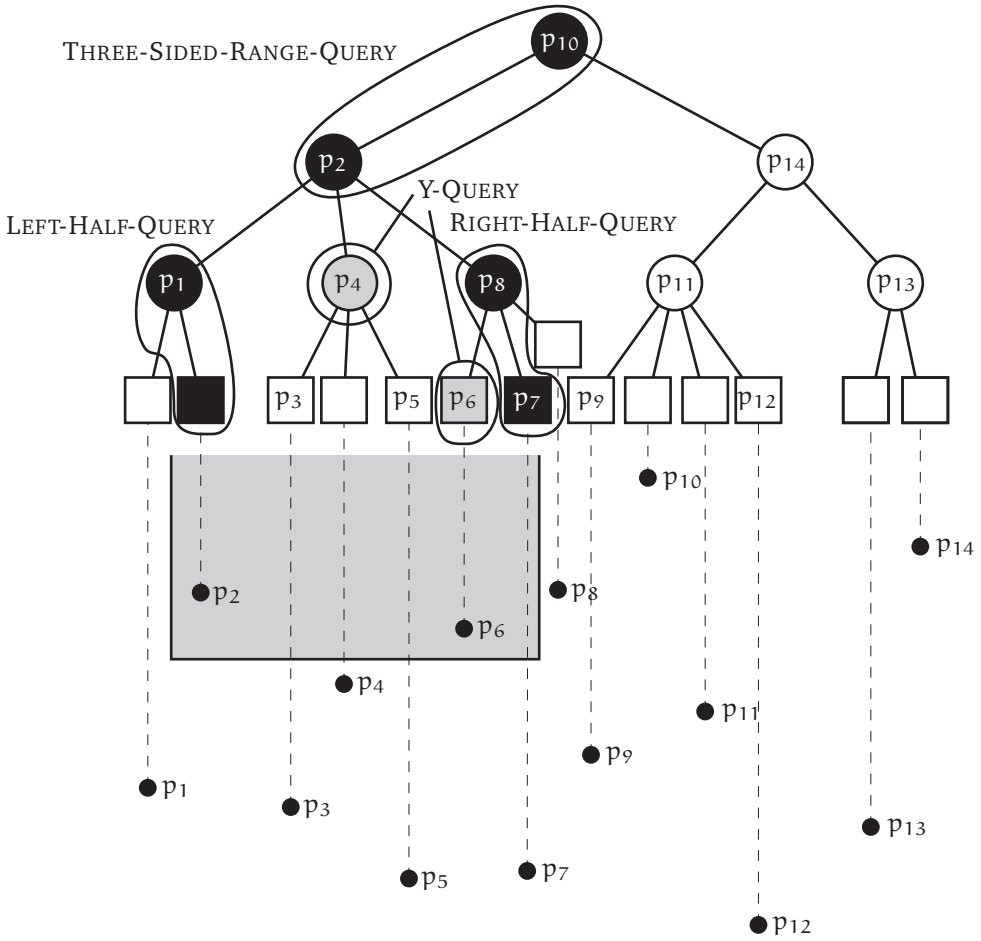


Figure 4.7. Procedure THREE-SIDED-RANGE-QUERY traverses the path containing the two top black nodes. It then invokes procedure LEFT-HALF-QUERY on the node storing point p_1, procedure Y-QUERY on the node storing point p_4, and procedure RIGHT-HALF-QUERY on the node storing point p_8. Procedure LEFT-HALF-QUERY then traverses the left black path. Procedure RIGHT-HALF-QUERY traverses the right black path. Procedure Y-QUERY is invoked on all grey nodes. Note that procedure Y-QUERY is not invoked on the children of the node storing point p_4 because p_4 already isn't in the query range.

By comparing procedures RANGE-QUERY for (a, b)-trees and THREE-SIDED-RANGE-QUERY for the priority search tree, we observe that, if we did not perform the empty subtree test in procedures THREE-SIDED-RANGE-QUERY, LEFT-HALF-QUERY, and RIGHT-HALF-QUERY, nor the test based on y-coordinates in the first line of procedure Y-QUERY, we would visit exactly the same set of nodes visited by procedure RANGE-QUERY. In particular, we conclude that, if we do not report a point, that is, if we do not visit the node v where this point is stored, this is caused by a failure of the test at one of v's ancestors. Call this ancestor u. The only test that can fail in an invocation of one of the procedures THREE-SIDED-RANGE-QUERY, LEFT-HALF-QUERY, and RIGHT-HALF-QUERY is that $\mathit{point}(u) = \text{NIL}$. By Property (PST5), this implies that node v cannot store a point either, a contradiction. So the only reason why we may fail to report point p is because the condition $\mathit{point}(u) \geq y^b$ is not satisfied in an invocation of procedure Y-QUERY on u. However, by Property (PST6), we have $y(\mathit{point}(u)) \geq y(\mathit{point}(v))$ and, hence, $y(\mathit{point}(v)) < y^b$. That is, point p is not reported because it is outside the query range.

Next we prove that procedure THREE-SIDED-RANGE-QUERY has the desired complexity on an (a, b)-tree with $a = O(1)$ and $b = O(1)$. The next lemma gives the desired bound, where t_v denotes the number of points in T_v that match the query.

Lemma 4.3 *The invocation* THREE-SIDED-RANGE-QUERY(v, x^l, x^r, y^b) *has running time* $O(\lg|\mathit{Items}(v)| + t_v)$.

Proof. Since every invocation of procedure THREE-SIDED-RANGE-QUERY, LEFT-HALF-QUERY, RIGHT-HALF-QUERY, or Y-QUERY takes $O(b) = O(1)$ time, it suffices to bound the number of invocations we make. First observe that we make at most one invocation of THREE-SIDED-RANGE-QUERY per level. This is true for the root level. Every subsequent invocation has to be made from an invocation to THREE-SIDED-RANGE-QUERY at the level immediately above. By induction, there is only one such invocation, which itself makes at most one recursive call to THREE-SIDED-RANGE-QUERY at the next lower level. Hence, there is at most one invocation of this procedure on every level. Now consider the highest invocation of procedure LEFT-HALF-QUERY. This invocation is made from within THREE-SIDED-RANGE-QUERY, and THREE-SIDED-RANGE-QUERY makes only one such invocation. Since there is only one invocation of procedure THREE-SIDED-RANGE-QUERY at any level and an invocation of procedure THREE-SIDED-RANGE-QUERY that invokes LEFT-HALF-QUERY does not spawn another invocation of procedure THREE-SIDED-RANGE-QUERY, there is only one top-level invocation of procedure LEFT-HALF-QUERY. Any subsequent invocation of procedure LEFT-HALF-QUERY is made from within an invocation of procedure LEFT-HALF-QUERY at a higher level. Each such invocation makes at most one call to LEFT-HALF-QUERY. Hence, it follows from the same argument as for procedure THREE-SIDED-RANGE-QUERY that there is at most one invocation to procedure LEFT-HALF-QUERY per level. An analogous argument shows that there is at most one invocation to procedure RIGHT-HALF-QUERY per level. This establishes that the cost of all invocations of procedures THREE-SIDED-RANGE-QUERY, LEFT-HALF-QUERY, and RIGHT-HALF-QUERY is $O(\text{height}(T_v)) = O(\lg|\mathit{Items}(v)|)$.

To bound the cost of all invocations to procedure Y-QUERY, we distinguish top-level invocations and recursive invocations. A top-level invocation is one

made from within THREE-SIDED-RANGE-QUERY, LEFT-HALF-QUERY, or RIGHT-HALF-QUERY. Since each such invocation makes at most $b = O(1)$ invocations to procedure Y-QUERY, the number of top-level invocations is $O(\lg|\mathit{Items}(v)|)$. A recursive invocation is one made from within a higher invocation of procedure Y-QUERY. So consider an invocation Y-QUERY(v, y^b) that makes a recursive call Y-QUERY(w, y^b). In order to make this recursive call in Line 5, we have to output a point on Line 2 first. Hence, every recursive invocation of procedure Y-QUERY on a node w can be "charged" to the point reported at its parent v. Since Y-QUERY(v, y^b) spawns at most b recursive calls on the children of v, $\mathit{point}(v)$ is charged for at most $b = O(1)$ recursive calls. Thus, the number of recursive invocations of procedure Y-QUERY is bounded by $O(t_v)$.

In summary, we make at most $O(\lg|\mathit{Items}(v)| + t_v)$ invocations to any of the four procedures that implement THREE-SIDED-RANGE-QUERY, and each such invocation costs constant time. This establishes the claimed time bound.

Since the initial invocation of procedure THREE-SIDED-RANGE-QUERY is on the root of T and $|\mathit{Items}(\mathit{root}(T))| = n$, we obtain

Corollary 4.2 *Procedure* THREE-SIDED-RANGE-QUERY *reports all points in* T *in a given query range in* $O(\lg n + t)$ *time.*

4.4 Updating Priority Search Trees

We have seen that a priority search tree, as defined in Definition 4.1, supports three-sided range queries in $O(\lg n + t)$ time. Next we are concerned with maintaining the information in the tree when adding or deleting a point to or from S.

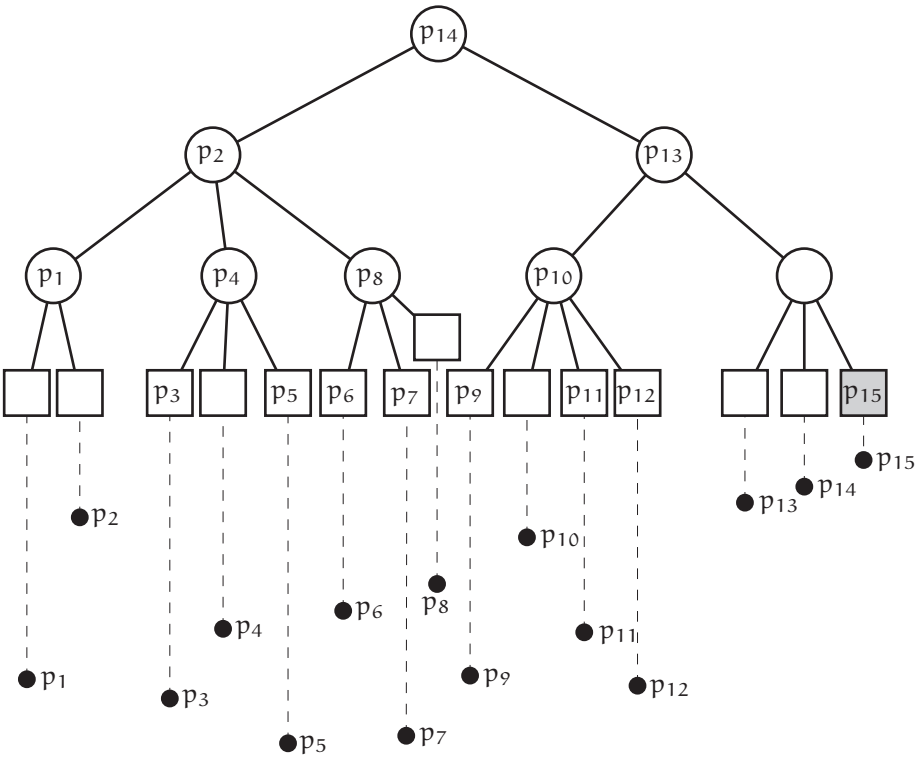
4.4.1 Insertions

When inserting a point into S, our first goal is to maintain the search tree structure. So we perform the insertion as on a normal (a, b)-tree, creating a new leaf ℓ_p in the correct position corresponding to the x-coordinate of point p, and storing p at this leaf. We also may have to split nodes, in order to keep the tree balanced. We worry about node splits later.

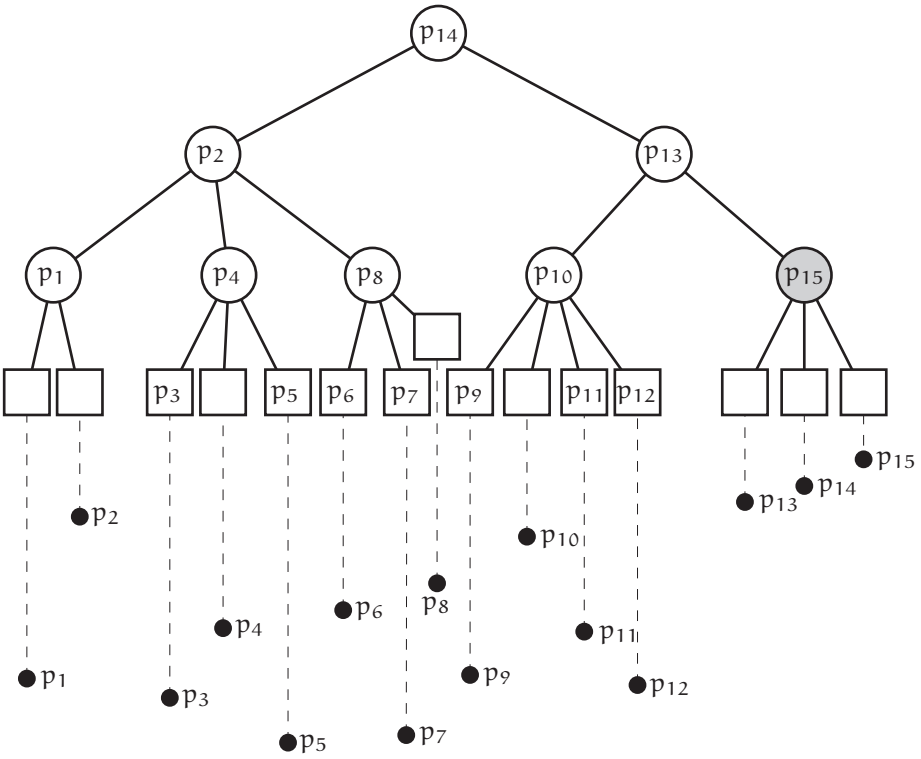
How does the insertion of point p at node ℓ_p affect the properties of tree T? Which properties may be violated? Properties (PST1), (PST2), (PST3), and (PST4) are obviously maintained. However, if v is the parent of ℓ_p and all points from T_v have been propagated to proper ancestors of v, Property (PST5) may be violated; and, obviously, we have no guarantee that p has a lower y-coordinate than any point stored at an ancestor of ℓ_p, that is, Property (PST6) may also be violated. See Figure 4.8a for an illustration. Our goal is therefore to restore Properties (PST5) and (PST6), while maintaining the other four.

Restoring Property (PST5) is easy (see Figure 4.8b): Remove point p from ℓ_p and store it at the highest ancestor of ℓ_p that currently does not store a node. But, if Property (PST6) was violated before, this does not do anything to restore this property. Indeed, any point that was stored at an ancestor of the node storing point p still has this property.

We can come close to a proper procedure for restoring Property (PST6) by remembering how we would do this if T was nothing more than a binary heap. In a binary heap, we would apply the HEAPIFY-UP operation, which swaps point

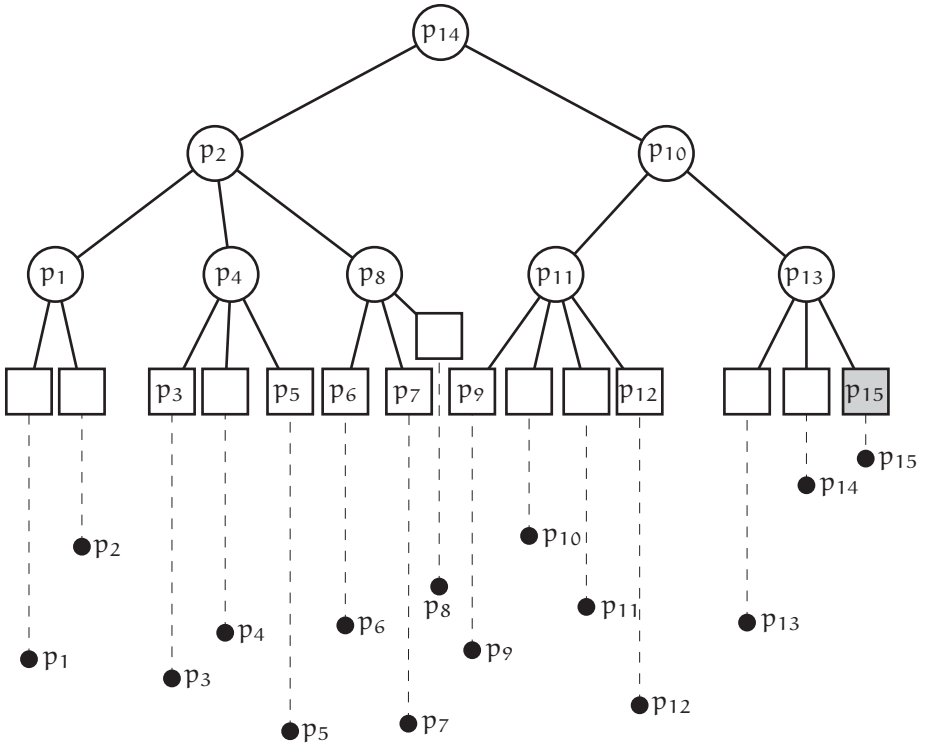


(a)

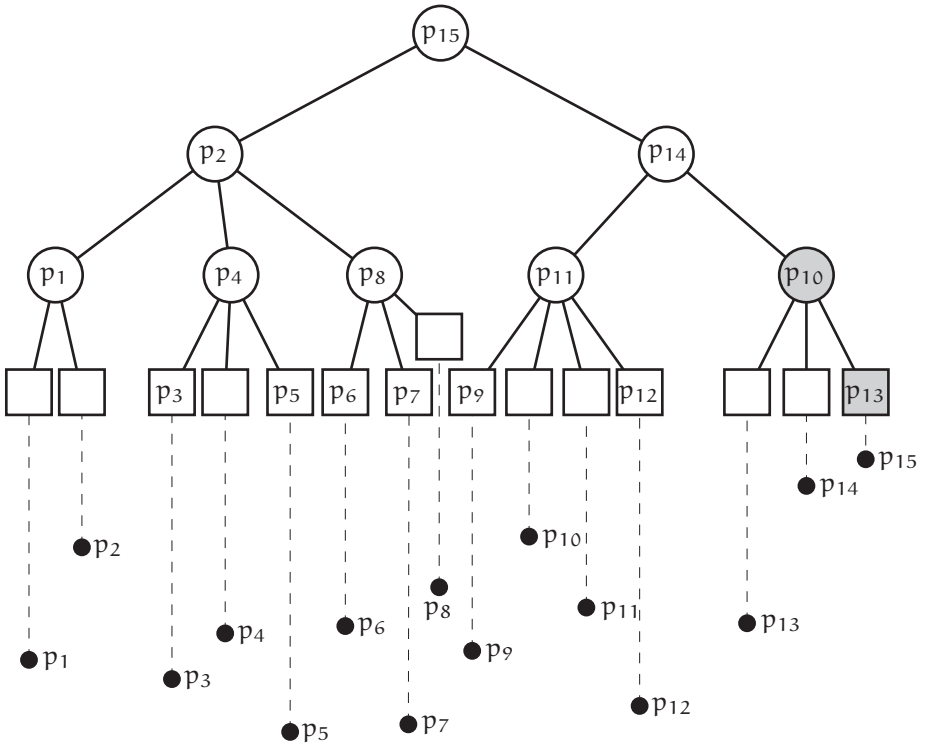


(b)

Figure 4.8. Insertion of a new point, p_{15} into a priority search tree. (a) Standard insertion of p_{15} violates Properties (PST5) and (PST6). (b) Moving p_{15} up restores Property (PST5), but not Property (PST6).



(a)



(b)

Figure 4.9. Insertion of a new point, p_{15} into a priority search tree. (a) Standard insertion of p_{15} violates Property (PST6). (b) Using HEAPIFY-UP to move p_{15} up to the root restores Property (PST6), but leaves points p_{10} and p_{13} at nodes that are not ancestors of their corresponding leaves.

p, currently stored at a node v, with the point q stored at $p(v)$ until we reach a situation where $y_p \leq y_q$. It follows from the discussion of binary heaps that this strategy would restore Property (PST6) in time proportional to the height of T, that is, in $O(\lg n)$ time.

So why can't we use procedure HEAPIFY-UP? The leaf ℓ_q corresponding to the point q currently stored at $p(v)$ may not be a descendant of node v. By swapping p and q, we move q to v; that is, point q would no longer be stored at an ancestor of ℓ_q, a violation of Property (PST4) (see Figure 4.9).

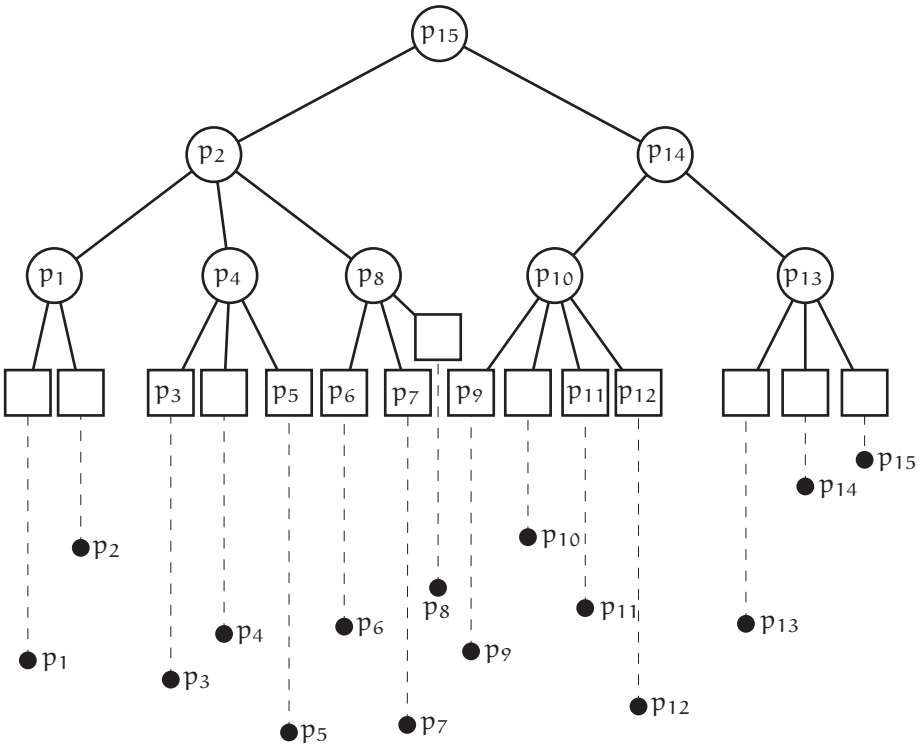


Figure 4.10. Moving point p_{15} to the root and iteratively pushing elements one step down the path to their corresponding leaves restores Property (PST6), while maintaining all other properties.

The net effect of procedure HEAPIFY-UP is to place point p at a node u and to push all the nodes on the path from v to u one position down the path. If we do not try to achieve this goal using swap operations bottom-up, but rather using a replacement procedure top-down, then we have a choice where to push every node, and this allows us to maintain Property (PST4). More precisely, we proceed as follows (see Figure 4.10): First we identify the lowest ancestor u of ℓ_p such that either u is the root or $p(u)$ stores a point q' with $y_{q'} \geq y_p$. Now let q be the point stored at u. Then we replace q with p. If $q \neq \text{NIL}$, let v be the child of u that is an ancestor of ℓ_q. Then we replace the point stored at v with q and continue to propagate this point one level down the tree until we find an empty node where we store the current point, and the propagation process ends. The following two procedures provide the details of this strategy.

```
PST-INSERT(T, p)
1  v ← INSERT(root(T), p)
2  point(v) ← NIL
3  while p(v) ≠ NIL and y(point(p(v))) < y_p
4        do v ← p(v)
5  PST-PUSH(v, p)
```

Procedure PST-INSERT uses the standard (a, b)-tree insertion to insert the new point into T. The newly created leaf is assigned to v. Then in Lines 3–4, we look for the lowest node v such that v is the root or the point stored at $p(v)$ has a greater y-coordinate than p; that is, v is the node where point p is to be stored. Once we have found this node v, we invoke the following PST-PUSH procedure on v:

```
PST-PUSH(v, p)
1  while point(v) ≠ NIL
2        do q ← point(v)
3           point(v) ← p
4           w ← child(v)
5           while right(w) ≠ NIL and x_q ≥ key(right(w))
6                 do w ← right(w)
7           p ← q
8           v ← w
9  point(v) ← p
```

This procedure replaces the point q at the current node v with point p. This forces us to push q one level down the tree. In order to ensure that q is still stored at an ancestor of the leaf ℓ_q corresponding to q, we have to identify the child w of v that is an ancestor of ℓ_q. This child must either be the child w with $key(w) \leq x_q < key(right(w))$ or, if no such child exists, it is the rightmost child of v. We locate this child in Lines 5–6 and then go into the next iteration with $v = w$ and $p = q$. Once the procedure finds a node that does not store any point, it simply stores the point p at this node and exits.

It is clear from our discussion that procedure PST-INSERT produces a valid priority search tree again (apart from the degree constraints of the nodes). The cost of the procedure is $O(\lg n)$: in procedure PST-INSERT itself, we spend constant time per level of the tree T; in procedure PST-PUSH, we spend $O(b) = O(1)$ time per level of T. Hence, we have

Lemma 4.4 *An insertion into a priority search tree (excluding the cost of rebalancing) costs* $O(\lg n)$ *time.*

4.4.2 Deletions

A node deletion has to undo the effect of an insertion. In particular, the leaf ℓ_p corresponding to the point p that is to be deleted has to be removed from T, and point p has to be removed from the node v that stores p. The only property that can be violated by doing this is Property (PST5) (see Figure 4.11a). In order to fix this violation, we have to propagate points up from the descendants of v. Now observe that the highest points among those stored at descendants of v are stored

at the children of v. Thus, in order to maintain the heap property, we have to choose the highest point out of the points stored at v's children and store this point at v. To do so, we have to remove it from the child, w, that currently stores this point; that is, we create a "hole" at w. In order to close this hole, we do the obvious thing: we apply the procedure recursively to w (see Figure 4.11b). Thus, a deletion (excluding rebalancing) quite simply looks as follows:

```
PST-DELETE(T, p)
1  v ← FIND(root(T), p)
2  if point(v) = NIL
3     then u ← p(v)
4          while point(u) ≠ p
5             do u ← p(u)
6          PST-PULL(u)
7  Remove node v from T
```

```
PST-PULL(u)
1  w ← child(u)
2  x ← NIL
3  while w ≠ NIL
4        do if point(w) ≠ NIL and (x = NIL or y(point(w)) > y(point(x)))
5              then x ← w
6  if x ≠ NIL
7     then point(u) ← point(x)
8          PST-PULL(x)
9     else point(u) ← NIL
```

Procedure PST-DELETE first identifies the leaf v corresponding to point p. If point p is stored at v, then, by deleting node v, point p as well as the leaf corresponding to p is deleted by removing v from T, and no further action is required. Otherwise, Lines 3–6 find the ancestor u of v that stores point p, invoke procedure PST-PULL on u, and then remove leaf v.

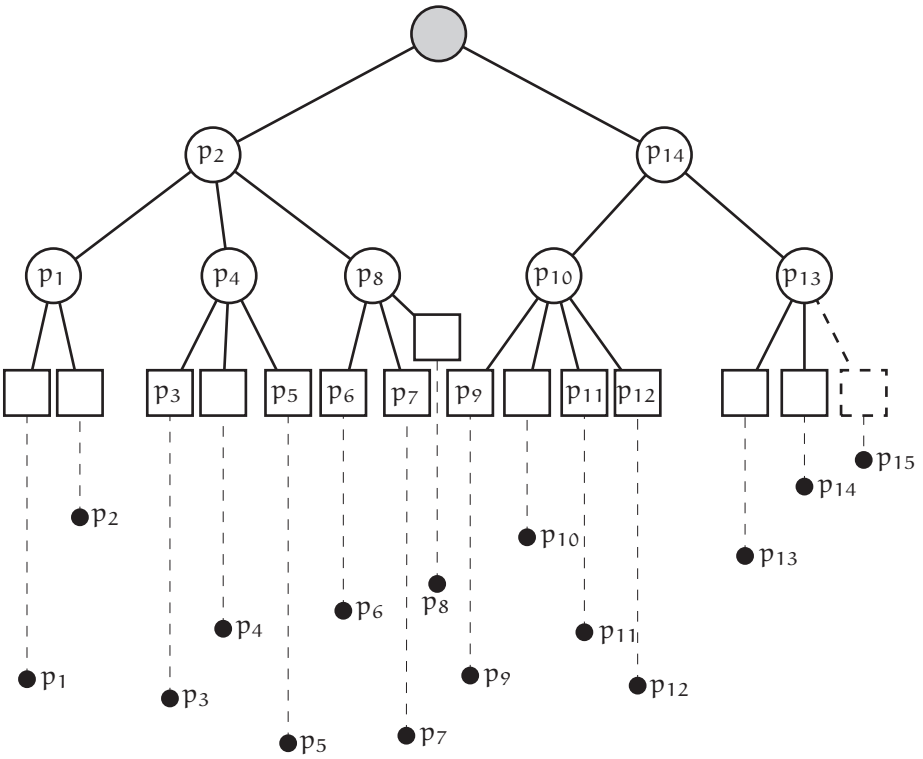
Procedure PST-PULL fills the hole created at u by deleting point p. More precisely, Lines 3–5 identify the child, x, of u that stores the point with maximum y-coordinate among the points stored at u's children, if such a child exists. Otherwise, $x = \text{NIL}$. If $x \neq \text{NIL}$, the point stored at x is moved to u, thereby creating a hole at x. This hole is closed by recursively invoking PST-PULL on x. If $x = \text{NIL}$, the whole subtree below u is empty, and we indicate this by setting $\mathit{point}(u) = \text{NIL}$.

It is easily verified that these updates maintain all priority search tree properties and that the deletion procedure takes $O(\lg n)$ time. Thus, we have established the following

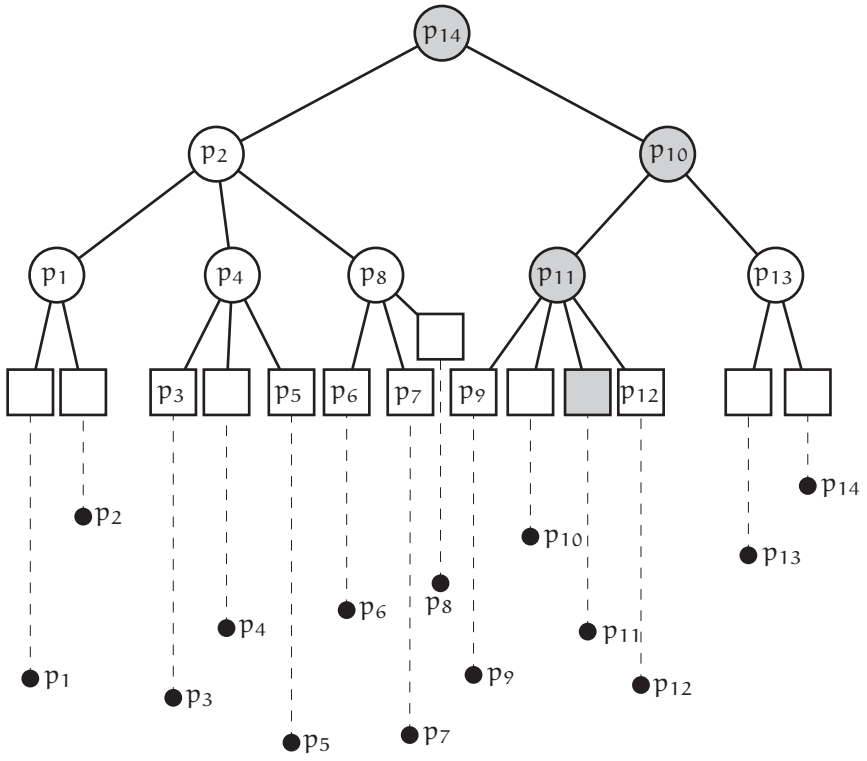
Lemma 4.5 *A deletion from a priority search tree (excluding the cost of rebalancing) costs* $O(\lg n)$ *time.*

4.4.3 Node Splits

Since we may have to rebalance after node insertions and deletions, we also have to worry about how to restore the information stored at all nodes after a node split or node fusion. We discuss node splits first.



(a)



(b)

Figure 4.11. (a) The deletion of point p_{15} from the tree in Figure 4.10 creates a "hole" at the root, violating Property (PST5). (b) Property (PST5) is restored by propagating points along the grey path up the tree.

Consider splitting a node v, and let $w_1, w_2, \ldots, w_k$ be its children, where $k > b$. We split v into two nodes v' and v'' with children $w_1, w_2, \ldots, w_h$ and $w_{h+1}, w_{h+2}, \ldots, w_k$, respectively. We have to answer two questions: Where do we store $\mathit{point}(v)$ (see Figure 4.12)? We will see that this will be one of the nodes v' or v''. Assuming that we store $\mathit{point}(v)$ at v', what do we store at v''?

Assume that we store $\mathit{point}(v)$ at v'. What can go wrong? All ancestors of v' were ancestors of v before the split, and they store the same information as before. Similarly, all descendants of v' were ancestors of v before the split, and they store the same information as before. Hence, by setting $\mathit{point}(v') = \mathit{point}(v)$, we restore Property (PST6). Properties (PST1), (PST2), and (PST3) are also satisfied. The two properties we may violate are Properties (PST4) and (PST5). In particular, the leaf corresponding to point $p = \mathit{point}(v')$ may in fact be a descendant of v''; and node v'' may violate Property (PST5) because its children may store points.

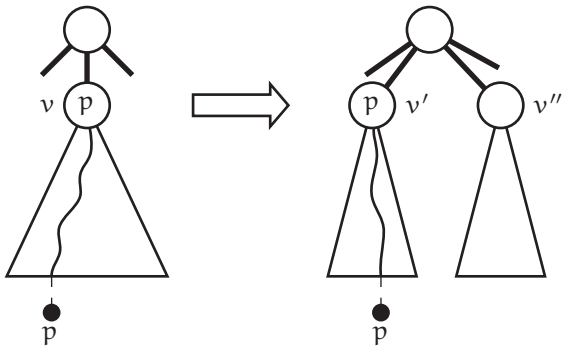


Figure 4.12. A node split in a priority search tree. The point p stored at v before the split is stored at the ancestor, v', of ℓ_p after the split.

If the leaf corresponding to point p is a descendant of v'', we avoid the violation of Property (PST4) by storing p at v'' instead and thereby making node v' violate Property (PST5). Whichever of the two nodes, v' or v'', violates Property (PST5), we can restore the property just like after a deletion, by invoking procedure PST-PULL on the violating node.

Since procedure PST-PULL takes $O(\lg n)$ time, and apart from that, a node split costs $O(b) = O(1)$ time, we have the following

Lemma 4.6 *A node split in a priority search tree costs* $O(\lg n)$ *time.*

Note that a node split in a priority search tree is more expensive than in an (a, b)-tree: in the latter a node split costs only constant time. Since an insertion may trigger $O(\lg n)$ node splits, one per ancestor of the inserted node, we obtain

Corollary 4.3 *An insertion into a priority search tree takes* $O(\lg^2 n)$ *time.*

4.4.4 Node Fusions

When performing a node fusion, we replace two nodes v' and v'' with children $w_1, w_2, \ldots, w_h$ and $w_{h+1}, w_{h+2}, \ldots, w_k$ with a single node v with children $w_1, w_2, \ldots, w_k$. We have to store some point at v, and we have to decide where to store the two points stored at v' and v''. Let us pretend for a minute that we avoid

the whole issue by storing both, $point(v')$ and $point(v'')$, at v. Which properties could be violated?

Properties (PST1), (PST2), (PST4), and (PST5) are maintained. For the first two, this is obvious. For Property (PST4), we observe that v is an ancestor of every leaf that had one of v' and v'' as ancestor before the fusion.

Property (PST3) is obviously violated. So we have to take one of the two points currently stored at v and push it down the tree. The decision which point to push down the tree is guided by the violation of the last property we have not considered yet: Property (PST6). Let $p = point(v')$ and $q = point(v'')$. Then

$$y_p \geq y(point(w_i)), \text{ for } 1 \leq i \leq h \tag{4.1}$$

and

$$y_q \geq y(point(w_i)), \text{ for } h < i \leq k. \tag{4.2}$$

But we have no guarantee that $y_q \geq y(point(w_i))$, for $1 \leq i \leq h$, or that $y_p \geq y(point(w_i))$, for $h < i \leq k$. However, if w.l.o.g. $y_p \geq y_q$, then, by Equation (4.1), we know that $y_p \geq y(point(w_i))$, for $1 \leq i \leq h$, and, by Equation (4.2), we know that $y_p \geq y_q \geq y(point(w_i))$, for $h < i \leq k$. Hence, the heap property is maintained if we store p at v and push q down the tree. We achieve this by setting $point(v) = q$ and then invoking procedure PST-PUSH(v, p). See Figure 4.13.

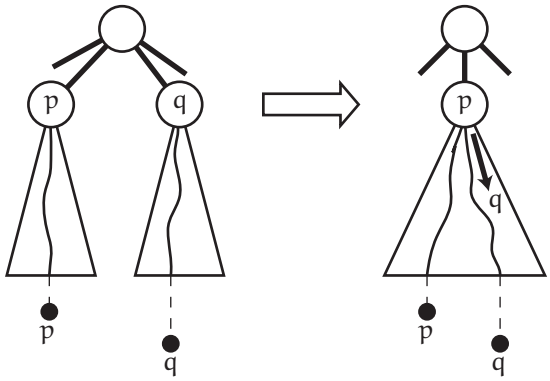


Figure 4.13. A node fusion evicts the lower of the two points stored at the fused nodes and propagates it down the tree.

Since procedure PST-PUSH takes $O(\lg n)$ time and, apart from that, the node fusion takes $O(1)$ time, we have

Lemma 4.7 *A node fusion in a priority search tree takes $O(\lg n)$ time.*

Since a deletion may trigger one node fusion per ancestor of the deleted leaf, possibly followed by a node split, we have

Corollary 4.4 *A deletion from a priority search tree takes $O(\lg^2 n)$ time.*

We summarize the result from this section in the following

Theorem 4.1 *A set of n points in the plane can be stored in a linear-space structure that supports the insertion and deletion of points in $O(\lg^2 n)$ time and three-sided range queries in $O(\lg n + t)$ time.*

4.5 Answering Interval Overlap Queries

Now that we have a data structure that can maintain a set of points in the plane under insertions and deletions and that can answer three-sided range queries, let us return to the second problem we wanted to solve: interval overlap queries. As already said in the introduction to this chapter, we do not want to develop a new data structure for this problem; but we want to interpret this problem in a different light, so we can use a priority search tree to solve this problem.

Since a priority search tree is a structure to store points, we need to interpret the intervals as points in the plane. Obviously, there is nothing that prevents us from doing this because an interval $[a, b]$ is a pair of numbers a and b, which we can readily consider to represent a point with x-coordinate a and y-coordinate b. The more interesting question is what properties such a point (a, b) must have if it is to represent an interval $[a, b]$ that overlaps a query interval $[c, d]$; that is, we want to map the interval $[c, d]$ into a query over the set of points representing the given set of intervals so that a point (a, b) matches the query if and only if interval $[a, b]$ overlaps interval $[c, d]$.

So when do intervals $[a, b]$ and $[c, d]$ overlap? Or, more genenerally, how can they be positioned with respect to each other and what is the distinguishing factor between the configurations where they overlap and the ones where they don't? Figure 4.14 shows all possible configurations. In the overlapping configurations, we have $a \leq d$ and $c \leq b$, while exactly one of these two conditions is violated when $[a, b]$ and $[c, d]$ do not overlap. This allows us to map the query interval $[c, d]$ into a two-sided range query with right boundary d and bottom boundary c. A point (a, b) is in this query range if and only if $a \leq d$ and $c \leq b$, that is, if and only if interval $[a, b]$ overlaps interval $[c, d]$.

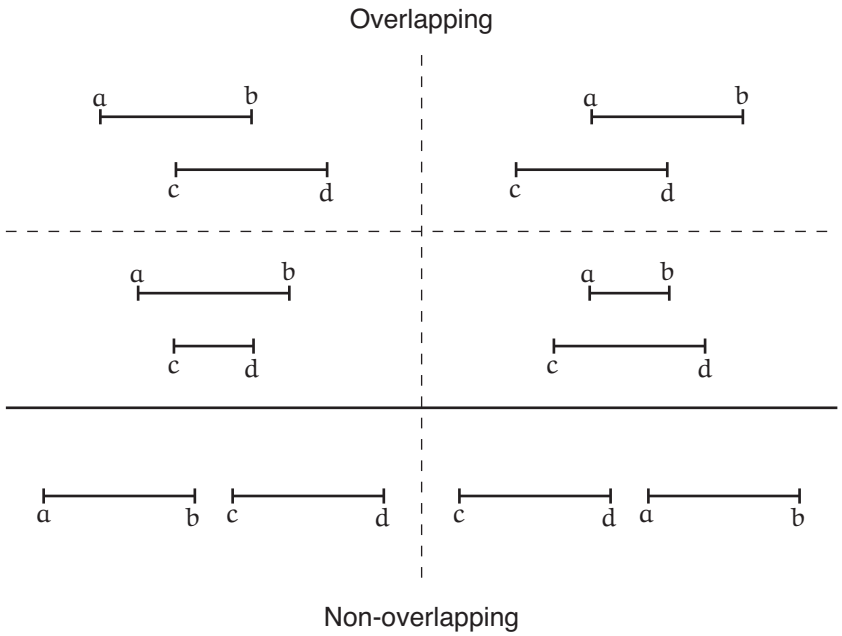


Figure 4.14. The six possible relative positions of two intervals.

A two-sided range query is a special case of a three-sided range query. More precisely, we can consider the query generated by interval $[c, d]$ to be a three-sided range query with left boundary $-\infty$, right boundary d, and bottom boundary c. If we store the point set S in a priority search tree, we can answer this type of

range query in $O(\lg n + t)$ time, that is, we can answer an interval overlap query in $O(\lg n + t)$ time. This leads to the following result.

Theorem 4.2 *A set of n intervals over the real line can be stored in a linear-space structure that supports the insertion and deletion of intervals in $O(\lg^2 n)$ time and interval overlap queries in $O(\lg n + t)$ time.*

4.6 An Improved Update Bound

The results in Theorems 4.1 and 4.2 are not quite what we set out to achieve: updates take $\lg n$ times longer than the desired $O(\lg n)$ time bound. In this section, we will improve the update bounds to the desired $O(\lg n)$ time. In doing so, we will explore two new concepts: ***amortization*** and ***weight-balancing***. When proving amortized time bounds for certain operations on a data structure, we do not make any claims about the worst-case running time of such an operation. In fact, we do not make *any* claim about the running time of an individual operation. Rather, we are interested in a worst-case upper bound on the time a *sequence* of such operations takes. Since we often use a data structure to speed up the computation of an algorithm, it is often irrelevant how long an individual operation takes, as long as the overall running time is low. Often, it is easier to obtain data structures with good amortized bounds on their operations than to guarantee a good worst-case running time.

4.6.1 Weight-Balanced (a, b)-Trees

The balancing conditions we have imposed on (a, b)-trees so far are often called ***degree balancing*** because we ensure that the degree of every node is in a specified range. While this type of balance is sufficient to guarantee a logarithmic height of the tree and to implement all operations we have discussed so far efficiently, it is in fact a rather weak balancing condition. In particular, a rather natural intuition what balancing means is that two subtrees T_v and T_w, where v and w are at the same level in T, should store about the same number of elements. If you consider a degree-balanced (a, b)-tree, it is very well possible that all nodes in the subtree rooted at one child of the root have degree a, and all nodes in the subtree rooted at another child have degree b. Then, if the height of the tree is h, the former subtree has a^{h-1} leaves, that is, stores a^{h-1} elements, while the other subtree stores b^{h-1} elements. The difference between the number of elements is therefore a factor of $(b/a)^{h-1}$, which is polynomial in the number of elements stored in the tree.

Weight balancing is a stronger balance condition on (a, b)-trees, which explicitly enforces our intuition about what balance should mean, implies degree balance, and can be achieved using node splits and fusions, just as degree balancing. In particular, if we define the ***weight***, $weight(v)$, of a node v to be the number of elements in T_v and the ***height*** of a node to be the number of edges on a path from v to a descendant leaf, a weight-balanced (a, b)-tree is defined as follows:

Definition 4.2 *For two integers* a *and* b *with* $2 \leq a$ *and* $16a \leq b$*, let* $\alpha = \sqrt{a}$*,* $\beta = \sqrt{b}$*, and* $\gamma = \alpha\beta$*. A* weight-balanced (a, b)-tree *is a rooted tree with the following properties:*

(WB1) The leaves are all at the same level (distance from the root).

(WB2) The data items are stored at the leaves, sorted from left to right.

(WB3) Every node at height h *has weight at most* $\beta\gamma^h$*.*

(WB4) Every non-root node at height h *has weight at least* $\alpha\gamma^h$*.*

(WB5) The root has at least two children.

(WB6) Every node v *stores a* key *key*(v)*. For a leaf, key*(v) *is the key of the data item associated with this leaf. For an internal node, key*$(v) = \min(\mathit{Keys}(v))$*.*

Let us first prove that weight balance implies degree balance:

Lemma 4.8 *In a weight-balanced* (a, b)*-tree, every node that is neither the root nor a leaf has degree between* a *and* b*. The root has degree between* 2 *and* b*.*

Proof. Consider a node v at height h. By Property (WB3), v has weight at most $\beta\gamma^h$. The children of v are at height $h - 1$ and are not the root of T. Therefore, by Property (WB4), each of them has weight at least $\alpha\gamma^{h-1}$. Thus, the degree of v is at most

$$\begin{aligned}\frac{\beta\gamma^h}{\alpha\gamma^{h-1}} &= \frac{\beta}{\alpha}\gamma \\ &= \frac{\sqrt{b}}{\sqrt{a}}\sqrt{ab} \\ &= b.\end{aligned}$$

The lower bound on the degree of the root is stated explicitly in Property (WB5). For a non-root node v that is not a leaf, the weight of v is at least $\alpha\gamma^h$. Its children have weight at most $\beta\gamma^{h-1}$. Hence, the degree of v is at least

$$\begin{aligned}\frac{\alpha\gamma^h}{\beta\gamma^{h-1}} &= \frac{\alpha}{\beta}\gamma \\ &= \frac{\sqrt{a}}{\sqrt{b}}\sqrt{ab} \\ &= a.\end{aligned}$$

Lemma 4.8, together with Lemma 1.1, implies immediately that, as a regular (a, b)-tree, a weight-balanced (a, b)-tree has logarithmic height.

Corollary 4.5 *A weight-balanced* (a, b)*-tree has height* $O(\log_a n)$*.*

Therefore, all query operations can be implemented in the same complexity on a weight-balanced (a, b)-tree as on a degree-balanced (a, b)-tree.

Next we observe that a weight-balanced (a, b)-tree can be rebalanced using regular node splits and fusions. In fact, we prove the following bound, which is stronger and will be essential for achieving a good amortized update bound in the next subsection:

Lemma 4.9 *A weight-balanced (a, b)-tree can be rebalanced using node splits and fusions. Every node of height h produced by such an operation has weight at most $\frac{3}{4}\beta\gamma^h$. If the produced node is not the root, its weight is at least $\frac{3}{2}\alpha\gamma^h$.*

Proof. First consider a node split. Let v be a node at height h whose weight exceeds $\beta\gamma^h$. A perfect split would split v into two nodes v' and v'' of weight $\mathit{weight}(v)/2$. However, such a split may require placing one of the children of v partially on both sides of the split; let us call this child w. Since we cannot split w, we make it a child of v'. Then v' has weight between $\mathit{weight}(v)/2$ and $\mathit{weight}(v)/2 + \mathit{weight}(w)$. Similarly, node v'' has weight between $\mathit{weight}(v)/2 - \mathit{weight}(w)$ and $\mathit{weight}(v)/2$. However, since w currently satisfies the weight constraints for level $h-1$, $\mathit{weight}(w) \leq \beta\gamma^{h-1}$. Thus, for $u \in \{v', v''\}$, we have

$$\begin{aligned}
\mathit{weight}(u) &\geq \frac{\beta\gamma^h}{2} - \beta\gamma^{h-1} \\
&\geq \beta\gamma^h\left(\frac{1}{2} - \frac{1}{\gamma}\right) \\
&\geq 4\alpha\gamma^h\left(\frac{1}{2} - \frac{1}{8}\right) \\
&= 4\alpha\gamma^h\frac{3}{8} \\
&= \frac{3}{2}\alpha\gamma^h
\end{aligned}$$

and

$$\begin{aligned}
\mathit{weight}(u) &\leq \frac{\beta\gamma^h + 1}{2} + \beta\gamma^{h-1} \\
&\leq \beta\gamma^h\left(\frac{1}{2} + \frac{1}{\gamma}\right) + \frac{1}{2} \\
&\leq \beta\gamma^h\left(\frac{1}{2} + \frac{1}{8}\right) + \frac{1}{2} \\
&= \frac{5}{8}\beta\gamma^h + \frac{1}{2} \\
&\leq \frac{3}{4}\beta\gamma^h.
\end{aligned}$$

For a node fusion, let v' and v'' be the two nodes that are fused, and assume that v' is the node whose weight is too low, that is, $\mathit{weight}(v') < \alpha\gamma^h$. Let v be the node resulting from the fusion. If $\mathit{weight}(v) \leq \frac{3}{4}\beta\gamma^h$, the upper bound on v's weight is satisfied. As for the lower bound, we observe that

$$\begin{aligned}
\mathit{weight}(v) &\geq 2\alpha\gamma^h - 1 \\
&\geq \frac{3}{2}\alpha\gamma^h.
\end{aligned}$$

If $\mathit{weight}(v) > \frac{3}{4}\beta\gamma^h$, we immediately split v into two nodes w' and w''. Since all children of w' and w'' satisfy the weight constraints for level $h-1$, we can apply the same arguments above to show that, for $w \in \{w', w''\}$ and assuming that we split v as evenly as possible,

$$\begin{aligned}\mathit{weight}(w) &\leq \frac{(\alpha+\beta)\gamma^h}{2} + \beta\gamma^{h-1} \\ &\leq \left(\frac{1}{8} + \frac{1}{2} + \frac{1}{\gamma}\right)\beta\gamma^h \\ &\leq \frac{3}{4}\beta\gamma^h.\end{aligned}$$

and

$$\begin{aligned}\mathit{weight}(w) &\geq \frac{(\alpha+\beta)\gamma^h - 1}{2} - \beta\gamma^{h-1} \\ &\geq \left(\frac{1}{8} + \frac{1}{2} - \frac{1}{\gamma}\right)\beta\gamma^h - \frac{1}{2} \\ &\geq \frac{1}{2}\beta\gamma^h - \frac{1}{2} \\ &\geq \frac{3}{2}\alpha\gamma^h.\end{aligned}$$

Hence, in either case, the nodes produced by a fusion satisfy the weight constraints for level h.

The point of Lemma 4.9 is that, after rebalancing a node, a large number of updates below this node are necessary before this node needs to be rebalanced again. The following corollary makes this precise.

Corollary 4.6 *When a node v is produced by a node split or fusion, a least $\alpha\gamma^h/2$ insertions or deletion into or from T_v are necessary before v can trigger another node split or fusion.*

Proof. Since the weight of v is at least $\frac{3}{2}\alpha\gamma^h$ immediately after the split or fusion, at least $\alpha\gamma^h/2$ deletions below v are necessary before v's weight drops below $\alpha\gamma^h$. Similarly, since the weight of v is at most $\frac{3}{4}\beta\gamma^h$ immediately after the split or fusion, at least $\beta\gamma^h/4 \geq \alpha\gamma^h$ insertions below v are necessary before the weight of v exceeds $\beta\gamma^h$.

Note that a node v produced by a onde split or fusion may be involved in another node split or fusion immediately after it is produced, because one of its siblings becomes under- or overfull. However, Corollary 4.6 states that node v cannot be the one that triggers the rebalancing operation.

4.6.2 An Amortized Update Bound

Amortization is a very useful concept. As already said, the idea is not to worry about the running time of individual operations, but about the total cost of a sequence of these operations, which is what is important in many applications. This concept is expressed precisely by the following definition:

Definition 4.3 (Amortization) *Given a data structure $\mathcal{D}$ that supports operations $o_1, o_2, \ldots, o_k$, we say that these operations have amortized cost $T_1(n), T_2(n), \ldots, T_k(n)$ if we can establish the following bound for the cost of any sequence S of n operations: For $1 \leq i \leq k$, let n_i of the operations be of type o_i; in particular, $\sum_{i=1}^{k} n_i = n$. Then the total cost of performing sequence S on an initially empty structure $\mathcal{D}$ is at most $\sum_{i=1}^{k} n_i \cdot T_i(n)$.*

We will now illustrate this concept by proving that a priority search tree based on a weight-balanced (a, b)-tree supports updates in $O(\lg n)$ amortized time. We use what is known as the ***credit method*** or ***savings account method*** to prove this fact. First recall that a single PST-PUSH or PST-PULL operation costs $O(\lg n)$ time. More precisely, there exists a constant c such that such an operation takes at most $c \lg n$ time. We can be a little more precise and observe that every subtree T_v of a weight-balanced (a, b)-tree is itself a weight-balanced (a, b)-tree and, therefore, has height $O(\lg \mathit{weight}(v))$; that is, a PST-PUSH or PST-PULL operation originating at this node costs at most $c \lg \mathit{weight}(v)$ computation steps. Now let us associate computation time with dollars by saying that every computation step costs us \$1. We want to prove that the total cost of n updates is $O(n \lg n)$ dollars.

We create a savings account A_v for every node v of T. Initially, every node has \$0 in its account. Now, for every insertion, when we create a new leaf ℓ, we add c dollars to the savings account of every ancestor of ℓ. This increases the cost of this insertion, but the cost is still $O(\lg n)$: the actual cost of traversing the path to ℓ, plus c dollars for each of the $O(\lg n)$ ancestors of ℓ. For a deletion, we proceed similarly: we add c dollars to the savings account of every ancestor of the deleted leaf ℓ. Again, this adds $O(\lg n)$ to the cost of every deletion. We will now argue that, when a node v is split or triggers a fusion, it has enough money in its savings account to pay for the cost of this operation. In other words, the updates on T_v (which are the cause for this rebalance operation) have already collectively paid for the cost of the rebalance operation. This implies that the total cost of n updates is n times what we pay for a single operation, which is $O(\lg n)$. Hence, the total cost is $O(n \lg n)$, and an individual operation takes $O(\lg n)$ time in the amortized sense.

Lemma 4.10 *When a node v at height h is split or triggers a node fusion, it has at least $c \lg \mathit{weight}(v)$ dollars in its savings account.*

Proof. Consider the time when node v splits or triggers a node fusion. At this time, its weight is either less than $\alpha\gamma^h$ or greater than $\beta\gamma^h$. At the time when node v was produced by a node fusion or a node split, its weight was between $\frac{3}{2}\alpha\gamma^h$ and $\frac{3}{4}\beta\gamma^h$. Thus, we must have performed more than $\frac{1}{2}\alpha\gamma^h$ deletions from T_v or more than $\frac{1}{4}\beta\gamma^h$ insertions into T_v. Each of these operations pays \$4c into v's account A_v. Hence, since $\mathit{weight}(v) = \alpha\gamma^h - 1$ or $\mathit{weight}(v) = \beta\gamma^h + 1$, v must have at least $c \cdot \mathit{weight}(v)$ dollars in A_v. Since $\lg x < x$, for all x, the lemma follows.

Corollary 4.7 *A priority search tree, when implemented using a weight-balanced (a, b)-tree, supports updates in $O(\lg n)$ amortized time and range queries in $O(\lg n + t)$ time in the worst case.*

Proof. By Corollary 4.5, the height of a weight-balanced (a, b)-tree is still $O(\lg n)$. By Lemma 4.8, every node in a weight-balanced (a, b)-tree has constant de-

gree if a and b are constants. Hence, the query bound is the same as in a degree-balanced priority search tree, which is $O(\lg n + t)$, by Corollary 4.2. By Lemma 4.10, the cost of all rebalancing operations is paid for by the extra amount every insertion and deletion pays into the savings accounts of the ancestors of the affected leaf. Since the actual cost of every update is (excluding the rebalancing) is $O(\lg n)$, and every update pays $O(\lg n)$ into the savings accounts of the ancestors of the affected leaf, the amortized cost per update is $O(\lg n)$.

4.7 Chapter Notes

Priority search trees are due to McCreight (1985). When implemented on top of red-black trees rather than (a, b)-trees, a worst-case $O(\lg n)$ update bound can be achieved quite easily. Since red-black trees are nothing but a binary representation of $(2, 4)$-trees, the rules for maintaining balance in a red-black tree based priority search tree can be translated into a $(2, 4)$-tree based priority search tree that supports updates in $O(\lg n)$ time in the worst case. McCreight (1985) is also the one who observed the simple transformation of interval overlap queries into range queries. Arge, Samoladas, and Vitter (1999) discuss a more sophisticated technique (which, however, is of mostly theoretical interest) that can be used to turn the amortized update bound discussed in Section 4.6 into a worst-case bound. Using this technique, amortized update bounds on other data structures can be converted into worst-case bounds as well.

Chapter 5
Range Trees

The final data structure we discuss addresses range searching in higher dimensions. It differs from the data structures we have discussed so far in a number of fundamental ways: First of all, it uses superlinear space; the exact space bound depends on the number of dimensions we are dealing with. Second, it does not support updates, that is, it is *static*: whenever the set of points represented by the data structure changes, we have to completely rebuild the structure. The third difference is that we break with a fundamental design concept, namely that every node of a tree can store only a constant amount of information.

We will not worry too much about the space bound; this is something we are willing to live with. But we should say something about the static nature of the structure and the assumption that a node can store more than a constant amount of data.

What good is a static data structure? Well, if you think about storing a database of historical data, say about all business transactions performed by your company, you may want to analyze this data to discover strengths and weaknesses in your business strategy. The number of queries you want to ask on the data structure may be quite large. So you want to perform queries quickly; you need an efficient data structure. At the same time, since the database contains historical data, it is unlikely to be updated, unless you discover some records in your files that you did not enter into the database when you constructed it. However, this does not happen too often; so rebuilding the data structure when it happens is a price we are willing to pay.

What about the assumption that we can store more than a constant amount of data in a node? Or, conversely, what's so fundamental about the assumption that a node can store only a constant amount of data? Think about how you represent a node in an actual implementation of the data structure. It has to be a struct in C or a class in C++, Java, or any other object-oriented language. For each such structure, you have to specify at compile time how much information the structure stores. No matter how much space you are willing to allocate to a node, it will always be a constant. The way around this constraint is obvious: we can make the node store a pointer (or reference) to a dynamically allocated secondary structure, which may well be of non-constant size. Thus, conceptually, the node stores more than a constant amount of data, because the secondary structure is associated with the node; but it's representation is still of constant size, because it stores nothing more than a pointer to the secondary structure. (Of course, the secondary structure itself uses space proportional to the amount of data it stores; but this is nothing we would have to worry about at compile time.)

Our discussion of the range searching problem will follow the same outline as all the previous chapters. In Section 5.1, we give a formal definition of the problem we want to solve. In Section 5.3, we develop ***range trees*** as the solution to this problem. This time, we use the priority search tree as the starting point of our journey from a known structure to one that solves our problem. This seems natural because we "only" have to support one additional constraint in our queries: the top boundary of the query range.

5.1 Higher-Dimensional Range Searching

We are by now familiar with the range-search problem, having answered 3-sided and 4-sided queries in two-dimensional space using the plane-sweep paradigm and having built a dynamic structure for answering 3-sided range queries. Now we want to build a data structure that generalizes 4-sided queries to higher dimensions, that is, we want to solve the following problem:

Problem 5.1 (Range searching) *Given a set S of n points in d-dimensional space, store them in a space-efficient data structure that allows us to answer range queries efficiently. A range query is specified by the hyperrectangle $q = [l_1, u_1] \times [l_2, u_2] \times \cdots \times [l_d, u_d]$ and asks to report all points $p = (p_1, p_2, \ldots, p_d)$ in S such that $l_i \leq p_i \leq u_i$, for all $1 \leq i \leq d$.*

Figure 5.1 illustrates a range query in three-dimensional space.

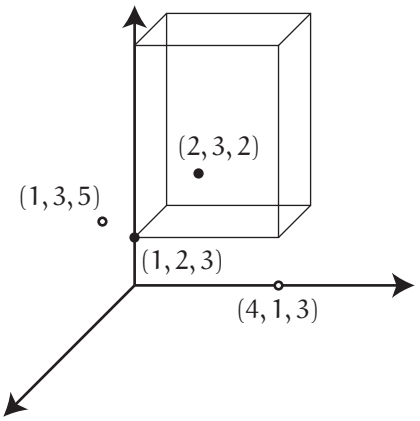


Figure 5.1. A three-dimensional range queries. Points are labelled with their coordinates. The query range is $[1,4] \times [2,6] \times [1,3]$. The black points are inside the query range. The white points are outside the query range.

5.2 Priority Search Trees?

For now, consider the two-dimensional version of the range searching problem; that is, our points are in the plane, and queries are rectangles. Recall how we used (a, b)-trees to answer three-sided range queries: We built an (a, b)-tree on the x-coordinates of the points. A search with the left and right boundaries of a query range identified $O(\lg n)$ parts of the tree that are interesting to the query: the two paths, P_l and P_r, leading to the leftmost and rightmost points in the x-range of the query, respectively, and $O(\lg n)$ maximal subtrees between these two paths. We observed that inspecting all nodes along P_l and P_r costs only $O(\lg n)$ time and, thus, does not pose a major problem. The main problem was the inspection of the subtrees between P_l and P_r. The problem was that the size of these trees may be much larger than the number of points in these trees that fall in the query range. We addressed this problem by turning the tree into a priority queue on the y-coordinates, in addition to maintaining the search tree property on the x-coordinates. The reason why this was possible is that all points stored between P_l and P_r must have x-coordinates in the query range; that is, for these points, the only factor deciding whether they are in the query range are their y-coordinates.

When trying to answer four-sided range queries, we can use the same approach: Our primary structure is an (a, b)-tree on the x-coordinates of the points. When answering queries, we identify paths P_l and P_r as well as the $O(\lg n)$ maximal subtrees between these two paths. For the points stored between these two paths, we know that their x-coordinates are in the query range; so we have to support searching this point set by the y-coordinates of the points. The complication we are dealing with, compared to three-sided queries, is that now we need to answer a full-blown range query on the y-coordinates, as opposed to a half-open query.

5.3 Two-Dimensional Range Trees

5.3.1 Description

When designing priority search trees, before developing the idea of turning every tree T_v itself into a heap structure, we had another idea: for every node v, store all the points in $\mathit{Items}(v)$ in another search structure that allows us to perform y-searches. We discarded this idea because it means that every point is stored $O(\lg n)$ times, which leads to an $O(n \lg n)$ space bound. Now, however, this is exactly what we do. So our structure is a two-level structure that looks as follows: The ***primary structure*** is an (a, b)-tree X over the points in S, sorted by their x-coordinates. Every leaf of X stores the point it corresponds to. Every internal node v of X stores a ***secondary structure*** Y_v, which is another (a, b)-tree over the poins in $\mathit{Items}(v)$, sorted by their y-coordinates. This two-level structure is called a ***two-dimensional range tree*** or ***2D range tree***, for short (see Figure 5.2). We will develop higher-dimensional range trees in the next section.

Note that storing a secondary structure Y_v with every node $v \in X$ is exactly where we conceptually store a non-constant amount of information with every node of X; but this is easily done because v obviously does not have to physically store Y_v; a pointer to the root of Y_v suffices.

Now let us see how good this structure is. In particular, we are interested in three questions:

- How much space does a two-dimensional range tree use?
- How quickly can it answer two-dimensional range queries?
- How quickly can we build it?

Let us start by settling the space question:

Lemma 5.1 *A two-dimensional range tree storing n points uses $O(n \lg n)$ space.*

Proof. The primary tree X is an (a, b)-tree storing n items. Hence, by Proposition 1.1, it uses $O(n)$ space. Each secondary structure Y_v is an (a, b)-tree over the points in $\mathit{Items}(v)$ and, therefore, uses $O(|\mathit{Items}(v)|)$ space. Since we have $|\mathit{Items}(\mathit{root}(X))| = n$, the total size of the secondary structures dominates the size of the primary tree, that is, the total space bound is $O(\sum_{v \in X} |\mathit{Items}(v)|)$. Now we

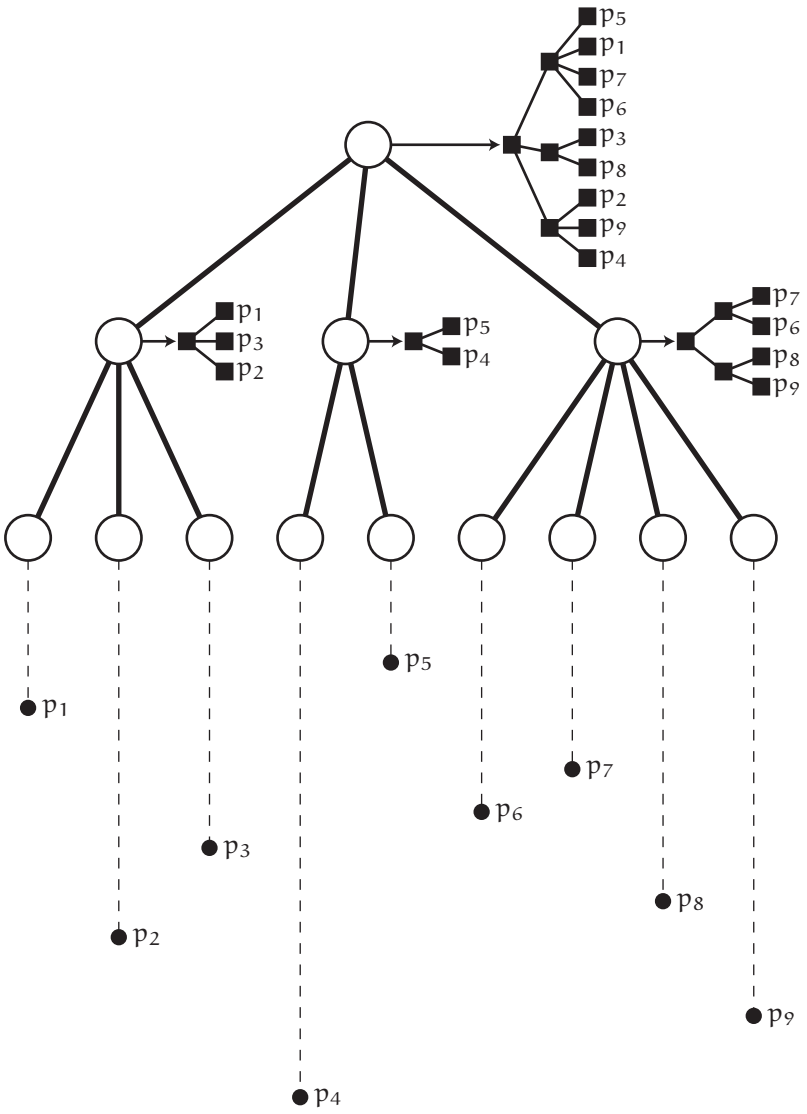


Figure 5.2. A two-dimensional range tree. The primary tree consists of the round nodes and fat edges. Every internal node of the primary tree stores a pointer to a secondary tree, which consists of square nodes and thin edges.

observe that

$$\begin{aligned}\sum_{v \in X} |\mathit{Items}(v)| &= \sum_{l \in \mathit{Leaves}(X)} |\mathit{Ancestors}(l)| \\ &= \sum_{l \in \mathit{Leaves}(X)} O(\lg n) \\ &= O(n \lg n),\end{aligned}$$

where $\mathit{Leaves}(X)$ denotes the set of leaves of X, and $\mathit{Ancestors}(v)$ denotes the set of ancestors of node v in X. The first equality holds because a point p is in $\mathit{Items}(v)$ if and only if $v \in \mathit{Ancestors}(l_p)$, where l_p is the leaf that stores p. The second equality holds because X is an (a, b)-tree over n items and, therefore, by Lemma 1.1, has height $O(\lg n)$. Finally, there is exactly one leaf per stored item, that is, there are n leaves in total. This establishes the third equality, and the lemma follows.

5.3.2 Two-Dimensional Range Queries

Next let us argue that we can answer two-dimensional range queries efficiently. We use the following procedure to answer such a query. The procedure follows the general outline of the THREE-SIDED-RANGE-SEARCH procedure for priority search trees (see page 63). The only difference is that, for every node w such that all elements in $\mathit{Items}(w)$ fall in the x-range of the query, we invoke a standard range query on w's secondary structure Y_w (see Line 12), instead of invoking procedure Y-SEARCH on w. Procedures LEFT-HALF-QUERY and RIGHT-HALF-QUERY need to be adapted in the same way.

```
2D-RANGE-QUERY(v, q)
 1  if v is a leaf
 2     then if point(v) ∈ q
 3             then output point(v)
 4     else w ← child(v)
 5          while right(w) ≠ NIL and key(right(w)) < l1(q)
 6              do w ← right(w)
 7          if right(w) = NIL or key(right(w)) > u1(q)
 8            then 2D-RANGE-QUERY(w, q)
 9            else 2D-LEFT-HALF-QUERY(w, q)
10                 w ← right(w)
11                 while right(w) ≠ NIL and key(right(w)) ≤ u1(q)
12                     do RANGE-QUERY(root(Yw), l2(q), u2(q))
13                        w ← right(w)
14                 2D-RIGHT-HALF-QUERY(w, q)
```

For an illustration of the query procedure, see Figure 5.3. The correctness of this procedure follows directly from our discussion. The next lemma establishes its time bound.

Lemma 5.2 *Procedure* 2D-RANGE-QUERY *takes* $O(\lg^2 n + t)$ *time, where* t *is the number of reported points.*

Proof. Procedures 2D-RANGE-QUERY and THREE-SIDED-RANGE-QUERY are identical, apart from how they deal with nodes whose point sets lie completely inside the x-range of the query. Thus, it follows from the analysis of procedure THREE-SIDED-RANGE-QUERY that we perform $O(\lg n)$ invocations of procedures 2D-RANGE-QUERY, 2D-LEFT-HALF-QUERY, and 2D-RIGHT-HALF-QUERY, one per node on the paths to the leftmost and rightmost leaves in the x-range. Every such invocation costs constant time and makes at most $b = O(1)$ calls to procedure RANGE-QUERY. Hence, we answer only $O(\lg n)$ range queries on secondary structures. Each such range query on a structure Y_w takes $O(\lg n + t_w)$ time, where t_w is the number of points in Y_w we report. Since $t = \sum_w t_w$, the total cost of all range queries is therefore $O(\lg^2 n + t)$.

5.3.3 Building Two-Dimensional Range Trees

Even though we motivate the usefulness of a static range search structure by the fact that, in certain applications, we do not have to update the structure very frequently, it is still desirable to be able to build the structure efficiently. Next, we

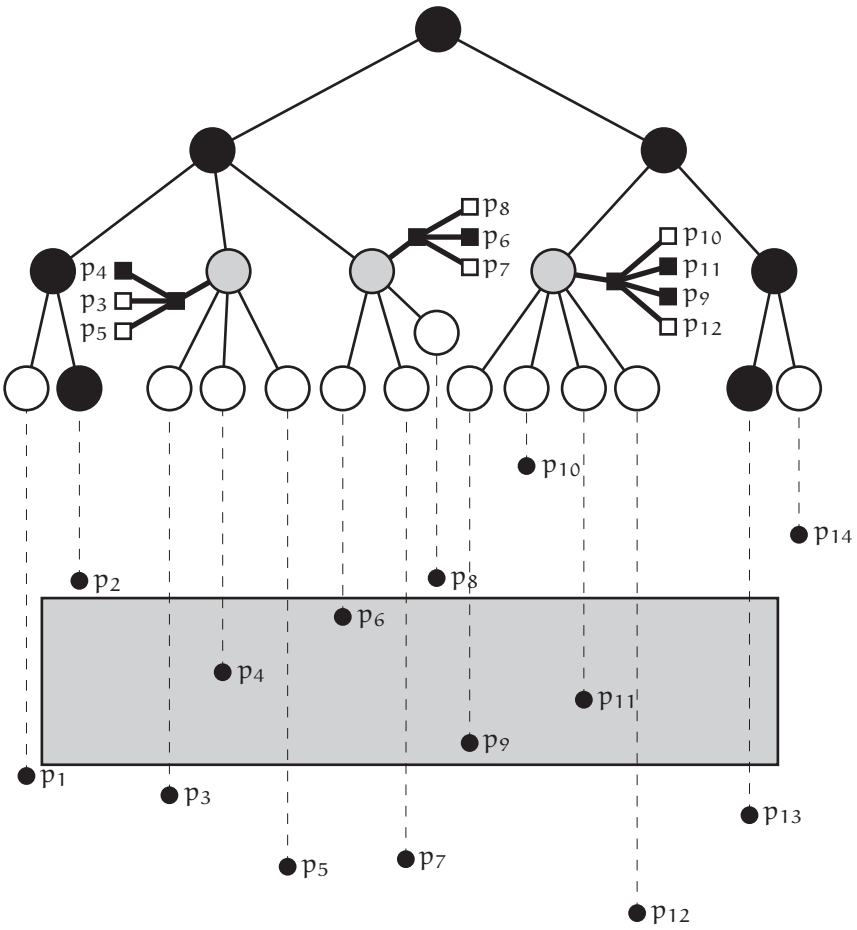


Figure 5.3. When answering a query with the grey query region, procedure 2D-RANGE-QUERY visits the black and grey nodes in the primary and secondary trees. The grey nodes are the ones whose secondary structures are inspected; only these secondary trees are shown. The visited leaves of the secondary structures correspond to the points in the query range: p_4, p_6, p_9, p_{11}.

show that a two-dimensional range tree can be built in the same amount of time as a standard (a, b)-tree, up to constant factors. In order to do so, we have to be somewhat careful. The first idea is to start with an empty structure and insert the points one by one. This certainly allows us to build an (a, b)-tree in $O(n \lg n)$ time. For range trees, we should expect trouble with this approach, given that we have already said that range trees are static structures. So what goes wrong?

The main problem is the updating of secondary structures when performing node splits: Consider a node v that is split into two nodes v' and v''. Before the split, Y_v stores all points in $\mathit{Items}(v)$. After the split, each such item belongs to one of $\mathit{Items}(v')$ or $\mathit{Items}(v'')$. Since Y_v is sorted by y-coordinates, but the split is performed by x-coordinates, we seem to have little choice but to inspect every single item in Y_v and to build $Y_{v'}$ and $Y_{v''}$ from scratch. This is too costly.[1]

So let us avoid the issue of node splitting by building the tree in two phases: In the first phase, we build only the primary tree X. We know how to do this in $O(n \lg n)$ time, either by inserting the points one by one or by sorting the points and then building X in linear time using the BUILD-TREE procedure from page 23.

[1]The interested reader may verify that, by using a weight-balanced primary tree and by being clever about rebuilding secondary structures, as explained below, this rather crude approach to rebalancing nevertheless leads to an $O(\lg n)$ amortized update bound.

At constant cost per node of X, that is, linear cost in total, we can associate an empty secondary structure Y_v with every node v of X. Given the primary tree X, we can now insert the points one by one: For every point $p \in S$, we traverse the path from the root of X to the leaf corresponding to p. At every node v along this path, we insert p into Y_v. What's the cost of this procedure?

Since no secondary structure stores more than n points at any point in time, inserting a point p into a structure Y_v takes $O(\lg n)$ time. Every point is inserted into $O(\lg n)$ secondary structures, one per ancestor of the leaf corresponding to p. Hence, the insertion cost per point is $O(\lg^2 n)$. Since there are n points, the total cost is $O(n \lg^2 n)$, which dominates the cost of building X.

Spending $O(n \lg^2 n)$ to build a 2D range tree is not too bad; but it is a $\lg n$ factor away from the promised $O(n \lg n)$ bound. To eliminate this factor, we have to reduce the insertion cost per point to $O(\lg n)$. Or, looking at it from the point of view of building the secondary structures, we want to build every secondary structure in time linear in its size. Since we know, by Lemma 5.1, that the total size of the secondary structures is $O(n \lg n)$, this implies that we can build all secondary structures in $O(n \lg n)$ time. Since we can build the primary tree in $O(n \lg n)$ time as well, the total time to build the 2D range tree is then $O(n \lg n)$.

Remember how we just avoided the ugly problem with splitting nodes in the primary tree: we exploited the fact that the whole point set S is given in advance; this allowed us to sort the points in S by their x-coordinates and then build X in *linear* time. Of course, the same trick should work for the secondary structures: If we are given $\mathit{Items}(v)$ in y-sorted order, we can build Y_v in $O(|\mathit{Items}(v)|)$ time. The crux is constructing all sets $\mathit{Items}(v)$, $v \in X$, in only $O(n \lg n)$ time. But, by exploiting the fact that the leaves of P store singleton sets and by borrowing some intuition from Merge Sort, this is easily accomplished: we simply build these sets bottom-up; that is, we build $\mathit{Items}(v)$, for a node v, only after we have built the sets associated with its children $w_1, w_2, \ldots, w_k$. Given the sets $\mathit{Items}(w_1), \mathit{Items}(w_2), \ldots, \mathit{Items}(w_k)$, we are left with the problem of merging a constant number ($k \leq b = O(1)$) of sorted lists into one sorted list. This is easily accomplished in linear time. From this discussion, we now obtain the following procedure for building a 2D range tree:

```
BUILD-2D-RANGE-TREE(S)
1  X ← BUILD-TREE(S)
2  BUILD-SECONDARY-STRUCTURES(root(X))

BUILD-SECONDARY-STRUCTURES(v)
   Builds Y_v if v is not a leaf and returns Items(v).
1  if v is a leaf
2     then return ⟨key(v)⟩
3     else w ← child(v)
4          L_v ← ∅
5          while w ≠ NIL
6               do L ← BUILD-SECONDARY-STRUCTURES(w)
7                  L_v ← MERGE(L_v, L)
8          Y_v ← BUILD-TREE(L_v)
9          return L_v
```

The correctness and complexity of this procedure follows from our discussion. Hence, we have the following

Lemma 5.3 *A two-dimensional range tree for n points in the plane can be built in $O(n \lg n)$ time.*

Observe that the merging of lists $\mathit{Items}(w_1), \mathit{Items}(w_2), \ldots, \mathit{Items}(w_k)$ into list $\mathit{Items}(v)$ is done by iteratively merging one list $\mathit{Items}(w_i)$ into the union of all preceding lists. The cost is $O(b|\mathit{Items}(v)|)$, which is $O(|\mathit{Items}(v)|)$ because b is constant. In certain applications, we have good reason to choose b to be a large constant or even not a constant at all. In these applications, a bound of $O(b|\mathit{Items}(v)|)$ is unacceptable. By merging the lists in a binary fashion, very much like Merge Sort, except that we do not start with singleton lists, we can reduce the cost of constructing $\mathit{Items}(v)$ to $O(|\mathit{Items}(v)| \lg b)$.

5.4 Higher-Dimensional Range Trees

Now consider the construction of a two-dimensional range tree a little more carefully. If we consider a regular (a, b)-tree to be a one-dimensional range tree—after all, it allows one-dimensional range queries to be answered optimally—then we can say that a two-dimensional range tree is nothing but a tree of one-dimensional range trees. The dimension on which the primary tree is built is used to decide which points to store in each of the one-dimensional trees. Also, for a query, the primary tree is used to decide which one-dimensional trees to search; but inside a one-dimensional tree, the query is completely oblivious of any constraints in the primary dimension that led to this tree being queried. In other words, the construction has provided a reduction from two-dimensional queries to one-dimensional ones. The question that comes to mind is: Can we repeat this trick to build a d-dimensional range tree from $(d-1)$-dimensional ones?

The answer to this question is obviously yes: The primary tree again gives us a partition of the points into subsets in the first dimension. This partition decides which points to store at every node in the primary tree. The secondary data structure associated with every node in the primary tree is now a $(d-1)$-dimensional range tree over the remaining $d-1$ coordinates. Each such $(d-1)$-dimensional tree is itself a tree of $(d-2)$-dimensional trees, and so on until we have decomposed everything into standard (a, b)-trees.

First, let us analyze the space consumption of a d-dimensional range tree:

Lemma 5.4 *A d-dimensional range tree storing n points uses $O(n \lg^{d-1} n)$ space.*

Proof. We prove this claim by induction on d. For $d = 1$, we have a standard (a, b)-tree, which uses $O(n) = O(n \lg^0 n)$ space. So assume that $d > 1$ and that the lemma holds for $d-1$. Consider the primary tree of the d-dimensional range tree. From the arguments in the proof of Lemma 5.1, it follows that every point is stored in the secondary data structures of $O(\lg n)$ nodes of the primary tree. Let n_v be the number of nodes stored in the secondary structure Y_v of node v. Then, by the induction hypothesis, Y_v uses $O(n_v \lg^{d-2} n_v)$ space. Thus, the total space

usage of all secondary structures is

$$\begin{aligned}\sum_{v\in X}|Y_v| &= \sum_{v\in X} O(n_v \lg^{d-2} n_v)\\ &= \sum_{v\in X} O(n_v \lg^{d-2} n)\\ &= O\left(\left(\sum_{v\in X} n_v\right)\lg^{d-2} n\right)\\ &= O(n \lg n \cdot \lg^{d-2} n)\\ &= O(n \lg^{d-1} n).\end{aligned}$$

Since the primary tree occupies a linear amount of space, the space usage of the secondary structures dominates, and the lemma follows.

In order to answer a d-dimensional range query using a d-dimensional range tree, we proceed as in the two-dimensional case: We perform a search on the primary tree to identify the $O(\lg n)$ maximal subtrees of X all of whose leaves correspond to points that are in the query range in the first dimension. The secondary structure of the root of such a tree stores exactly these points. To distinguish which of these points actually fall in the query range, it suffices to consider the remaining $d-1$ dimensions because we already know that all the points in the secondary structure satisfy the query constraints in the first dimension. Thus, we answer a $(d-1)$-dimensional range query on the secondary structure. Since every query decomposes into $O(\lg n)$ maximal subtrees, we answer $O(\lg n)$ $(d-1)$-dimensional range queries. Thus, we obtain the following lemma:

Lemma 5.5 *A d-dimensional range query on a d-dimensional range tree storing n points takes $O(\lg^d n + t)$ time.*

Proof. Again, we prove the claim by induction. For $d = 1$, we answer a standard range query on an (a,b)-tree. This takes $O(\lg n + t)$ time. So assume that $d > 1$ and that the lemma holds for $d-1$ dimensions. Since we answer $O(\lg n)$ $(d-1)$-dimensional range queries to answer a single d-dimensional range query, the query complexity is

$$\begin{aligned}\sum_v O(\lg^{d-1} n_v + t_v) &= \sum_v O(\lg^{d-1} n_v) + \sum_v O(t_v)\\ &= \sum_v O(\lg^{d-1} n) + O(t)\\ &= O(\lg n \cdot \lg^{d-1} n) + O(t)\\ &= O(\lg^d n + t),\end{aligned}$$

where all sums are over the set of queried nodes. The second equality follows because $n_v \le n$ and the set of points stored in the queried subtrees are disjoint; thus, $t = \sum_v t_v$.

Using a technique known as ***fractional cascading***, the query complexity can be reduced to $O(\lg^{d-1} n + t)$ for $d > 1$. See the notes at the end of the chapter for references.

The final question we should ask about d-dimensional range trees is whether they can be built efficiently. Again, we employ the same strategy as for two-dimensional range trees, that is, we sort the points by their first coordinates and then build the primary tree in linear time. Now we merge point sets in a bottom-up fashion to construct the set L_v to be stored in the secondary structure of each node v. From each such set L_v, we build the secondary structure Y_v associated with node v.

Lemma 5.6 *For $d \geq 2$, a d-dimensional range tree storing n points can be constructed in $O(n \lg^{d-1} n)$ time.*

Proof. Again, the proof is by induction on d. For $d = 2$, Lemma 5.3 states that the tree can be built in $O(n \lg n)$ time. Now assume that $d > 2$ and that the lemma holds for $d-1$ dimensions. To build the primary tree, we sort the points and then apply the BUILD-TREE procedure for standard (a, b)-trees. This takes $O(n \lg n)$ time. The bottom-up merging of point lists takes $O(n \lg n)$ time because every point is involved only in the merging processes at its ancestors. By the induction hypothesis, the cost of building a single secondary structure Y_v is $O(|Y_v| \lg^{d-2} |Y_v|)$. Hence, the total cost of building all secondary structures is

$$\begin{aligned}\sum_{v \in X} O(|Y_v| \lg^{d-2} |Y_v|) &= \sum_{v \in X} O(|Y_v| \lg^{d-2} n) \\ &= O(n \lg n \cdot \lg^{d-2} n) \\ &= O(n \lg^{d-1} n).\end{aligned}$$

This finishes the proof.

5.5 Chapter Notes

Range trees have been discovered independently by Bentley (1979), Lee and Wong (1980), Lueker (1978), and Willard (1979). They are only one type of higher-dimensional search structure. Other, more efficient, structures include k-d-trees (Bentley 1975), R-trees (Guttman 1984; Beckmann, Kriegel, Schneider, and Seeger 1990; Kamel and Faloutsos 1994; Sellis, Roussopoulos, and Faloutsos 1987), PR-trees (Arge, de Berg, Haverkort, and Yi 2004; Arge, de Berg, and Haverkort 2005), and O-trees (Kanth and Singh 1999). Samet (1990a, 1990b) gives a comprehensive survey of spatial data structures and their applications.

In Section 5.4, the technique of fractional cascading was mentioned. This technique can be used to speed up a sequence of searches. The basic idea is to perform one search from scratch and to speed up subsequent searches by using information gained in previous searches. A detailed discussion of this technique is given by Chazelle and Guibas (1986a, 1986b). Lueker (1978) and Willard (1978) were the ones who observed its applicability to orthogonal range searching.

Bibliography

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

Agarwal, P. K. (1990). Partitioning arrangements of lines: II. applications. *Discrete Computational Geometry 5*, 533–573.

Andersson, A. (1993). Balanced search trees made simple. In *Proceedings of the 3rd Workshop on Algorithms and Data Structures*, Volume 709 of *Lecture Notes in Computer Science*, pp. 60–71. Springer-Verlag.

Arge, L., M. de Berg, and H. J. Haverkort (2005). Cache-oblivious R-trees. In *Proceedings of the 21st ACM Symposium on Computational Geometry*, pp. 170–179.

Arge, L., M. de Berg, H. J. Haverkort, and K. Yi (2004). The priority R-tree: A practically efficient and worst-case optimal R-tree. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pp. 347–358.

Arge, L., V. Samoladas, and J. S. Vitter (1999). On two-dimensional indexability and optimal range search indexing. In *Proceedings of the 18th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pp. 346–357.

Balaban, I. J. (1995). An optimal algorithm for finding segment intersections. In *Proceedings of the 11th ACM Symposium on Computational Geometry*, pp. 211–219.

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

Bayer, R. and E. McCreight (1972). Organization of large ordered indexes. *Acta Informatica 1*, 173–189.

Beckmann, N., H.-P. Kriegel, R. Schneider, and B. Seeger (1990). The R*-tree: An effcient and robust access method for points and rectangles. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 322–331.

Bentley, J. L. (1975). Multidimensional binary search trees used for associative searching. *Communications of the ACM 18*, 509–517.

Bentley, J. L. (1979). Decomposable search problems. *Information Processing Letters 8*, 244–251.

Bentley, J. L. and T. A. Ottmann (1979). Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computers C-28*, 643–647.

Chazelle, B. (1993). Cutting hyperplanes for divide-and-conquer. *Discrete Computational Geometry 9*, 145–158.

Chazelle, B. and H. Edelsbrunner (1988). An optimal algorithm for intersecting line segments in the plane. In *Proceedings of the 29th IEEE Symposium on Foundations of Computer Science*, pp. 590–600.

Chazelle, B. and H. Edelsbrunner (1992). An optimal algorithm for intersecting line segments in the plane. *Journal of the ACM 39*, 1–54.

Chazelle, B., H. Edelsbrunner, L. J. Guibas, and M. Sharir (1994). Algorithms for bichromatic line segment problems and polyhedral terrains. *Algorithmica 11*, 116–132.

Chazelle, B. and L. J. Guibas (1986a). Fractional cascading: I. a data structuring technique. *Algorithmica 1*, 133–162.

Chazelle, B. and L. J. Guibas (1986b). Fractional cascading: Ii. applications. *Algorithmica 1*, 163–191.

Clarkson, K. L. and P. W. Shor (1989). Applications of random sampling in computational geometry, II. *Discrete Computational Geometry 4*, 387–421.

Cormen, T. H., C. E. Leiserson, R. L. Rivest, and C. Stein (2001). *Introduction to Algorithms* (second ed.). The MIT Press.

de Berg, M., M. van Kreveld, M. Overmars, and O. Schwarzkopf (1997). *Computational Geometry: Algorithms and Applications* (first ed.). Springer-Verlag.

de Berg, M., M. van Kreveld, M. Overmars, and O. Schwarzkopf (2000). *Computational Geometry: Algorithms and Applications* (second ed.). Springer-Verlag.

Guibas, L. J. and R. Sedgewick (1978). A dichromatic framework for balanced trees. In *Proceedings of the 10th IEEE Symposium on Foundations of Computer Science*, pp. 8–21. IEEE Computer Society.

Guttman, A. (1984). R-trees: A dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pp. 47–57.

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

Kamel, I. and C. Faloutsos (1994). Hilbert R-tree: An improved R-tree using fractals. In *Proceedings of the 20th International Conference on Very Large Databases*, pp. 500–509.

Kanth, K. V. R. and A. K. Singh (1999). Optimal dynamic range searching in non-replicating index structures. In *Proceedings of the 7th International Conference on Database Theory*, Volume 1540 of *Lecture Notes in Computer Science*, pp. 257–276. Springer-Verlag.

Lee, D. T. and C. K. Wong (1980). Quintary trees: a file structure for multidimensional database systems. *ACM Transactions on Database Systems 5*, 339–353.

Lueker, G. S. (1978). A data structure for orthogonal range queries. In *Proceedings of the 19th IEEE Symposium on Foundations of Computer Science*, pp. 28–34.

Mairson, H. G. and J. Stolfi (1988). Reporting and counting intersections between two sets of line segments. In R. A. Earnshaw (Ed.), *Theoretical Foundations of Computer Graphics and CAD*, Volume 40 of *NATO ASI Series F*, pp. 307–325. Springer-Verlag.

McCreight, E. M. (1985). Priority search trees. *SIAM Journal on Computing 14*, 257–276.

Mulmuley, K. (1988). A fast planar partition algorithm, I. In *Proceedings of the 29th IEEE Symposium on Foundations of Computer Science*, pp. 580–589.

Nievergelt, J. and E. M. Reingold (1973). Binary search trees of bounded balance. *SIAM Journal on Computing 2*(1), 33–43.

Pach, J. and M. Sharir (1991). On vertical visibility in arrangements of segments and the queue size in the Bentley-Ottman line sweeping algorithm. *SIAM Journal on Computing 20*, 460–470.

Preparata, F. P. and M. I. Shamos (1985). *Computational Geometry: An Introduction*. Springer-Verlag.

Samet, H. (1990a). *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Reading, MA: Addison-Wesley.

Samet, H. (1990b). *The Design and Analysis of Spatial Data Structures*. Reading, MA: Addison-Wesley.

Sellis, T., N. Roussopoulos, and C. Faloutsos (1987). The R+-tree: A dynamic index for multi-dimensional objects. In *Proceedings of the 13th International Conference on Very Large Databases*, pp. 507–518.

Willard, D. E. (1978). *Predicate-oriented database search algorithms*. Ph. D. thesis, Harvard University, Cambridge, MA.

Willard, D. E. (1979). The super-b-tree algorithm. Technical Report TR-03-79, Harvard University, Cambdridge, MA.