Part V Advanced Data Structures

Introduction

This part returns to studying data structures that support operations on dynamic sets, but at a more advanced level than Part III. One of the chapters, for example, makes extensive use of the amortized analysis techniques from Chapter 16.

Chapter 17 shows how to augment red-black trees—adding additional information in each node—to support dynamic-set operations in addition to those covered in Chapters 12 and 13. The first example augments red-black trees to dynamically maintain order statistics for a set of keys. Another example augments them in a different way to maintain intervals of real numbers. Chapter 17 includes a theorem giving sufficient conditions for when a red-black tree can be augmented while maintaining the $O(\lg n)$ running times for insertion and deletion.

Chapter 18 presents B-trees, which are balanced search trees specifically designed to be stored on disks. Since disks operate much more slowly than random-access memory, B-tree performance depends not only on how much computing time the dynamic-set operations consume but also on how many disk accesses they perform. For each B-tree operation, the number of disk accesses increases with the height of the B-tree, but B-tree operations keep the height low.

Chapter 19 examines data structures for disjoint sets. Starting with a universe of n elements, each initially in its own singleton set, the operation UNION unites two sets. At all times, the n elements are partitioned into disjoint sets, even as calls to the UNION operation change the members of a set dynamically. The query FIND-SET identifies the unique set that contains a given element at the moment. Representing each set as a simple rooted tree yields surprisingly fast operations: a sequence of m operations runs in $O(m\,\alpha(n))$ time, where $\alpha(n)$ is an incredibly slowly growing function—$\alpha(n)$ is at most 4 in any conceivable application. The amortized analysis that proves this time bound is as complex as the data structure is simple.

The topics covered in this part are by no means the only examples of "advanced" data structures. Other advanced data structures include the following:

- ***Fibonacci heaps*** [156] implement mergeable heaps (see Problem 10-2 on page 268) with the operations INSERT, MINIMUM, and UNION taking only $O(1)$ actual and amortized time, and the operations EXTRACT-MIN and DELETE taking $O(\lg n)$ amortized time. The most significant advantage of these data structures, however, is that DECREASE-KEY takes only $O(1)$ amortized time. ***Strict Fibonacci heaps*** [73], developed later, made all of these time bounds actual. Because the DECREASE-KEY operation takes constant amortized time, (strict) Fibonacci heaps constitute key components of some of the asymptotically fastest algorithms to date for graph problems.
- ***Dynamic trees*** [415, 429] maintain a forest of disjoint rooted trees. Each edge in each tree has a real-valued cost. Dynamic trees support queries to find parents, roots, edge costs, and the minimum edge cost on a simple path from a node up to a root. Trees may be manipulated by cutting edges, updating all edge costs on a simple path from a node up to a root, linking a root into another tree, and making a node the root of the tree it appears in. One implementation of dynamic trees gives an $O(\lg n)$ amortized time bound for each operation, while a more complicated implementation yields $O(\lg n)$ worst-case time bounds. Dynamic trees are used in some of the asymptotically fastest network-flow algorithms.
- ***Splay trees*** [418, 429] are a form of binary search tree on which the standard search-tree operations run in $O(\lg n)$ amortized time. One application of splay trees simplifies dynamic trees.
- ***Persistent*** data structures allow queries, and sometimes updates as well, on past versions of a data structure. For example, linked data structures can be made persistent with only a small time and space cost [126]. Problem 13-1 gives a simple example of a persistent dynamic set.
- Several data structures allow a faster implementation of dictionary operations (INSERT, DELETE, and SEARCH) for a restricted universe of keys. By taking advantage of these restrictions, they are able to achieve better worst-case asymptotic running times than comparison-based data structures. If the keys are unique integers drawn from the set $\{0, 1, 2, \ldots, u - 1\}$, where u is an exact power of 2, then a recursive data structure known as a ***van Emde Boas tree*** [440, 441] supports each of the operations SEARCH, INSERT, DELETE, MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR in $O(\lg \lg u)$ time. ***Fusion trees*** [157] were the first data structure to allow faster dictionary operations when the universe is restricted to integers, implementing these operations in $O(\lg n / \lg \lg n)$ time. Several subsequent data structures, including ***exponential search trees*** [17], have also given improved bounds on some or all of

the dictionary operations and are mentioned in the chapter notes throughout this book.

- ***Dynamic graph data structures*** support various queries while allowing the structure of a graph to change through operations that insert or delete vertices or edges. Examples of the queries that they support include vertex connectivity [214], edge connectivity, minimum spanning trees [213], biconnectivity, and transitive closure [212].

Chapter notes throughout this book mention additional data structures.

17 Augmenting Data Structures

Some solutions require no more than a "textbook" data structure—such as a doubly linked list, a hash table, or a binary search tree—but many others require a dash of creativity. Rarely will you need to create an entirely new type of data structure, though. More often, you can augment a textbook data structure by storing additional information in it. You can then program new operations for the data structure to support your application. Augmenting a data structure is not always straightforward, however, since the added information must be updated and maintained by the ordinary operations on the data structure.

This chapter discusses two data structures based on red-black trees that are augmented with additional information. Section 17.1 describes a data structure that supports general order-statistic operations on a dynamic set: quickly finding the ith smallest number or the rank of a given element. Section 17.2 abstracts the process of augmenting a data structure and provides a theorem that you can use when augmenting red-black trees. Section 17.3 uses this theorem to help design a data structure for maintaining a dynamic set of intervals, such as time intervals. You can use this data structure to quickly find an interval that overlaps a given query interval.

17.1 Dynamic order statistics

Chapter 9 introduced the notion of an order statistic. Specifically, the ith order statistic of a set of n elements, where $i \in \{1, 2, \ldots, n\}$, is simply the element in the set with the ith smallest key. In Chapter 9, you saw how to determine any order statistic in $O(n)$ time from an unordered set. This section shows how to modify red-black trees so that you can determine any order statistic for a dynamic set in $O(\lg n)$ time and also compute the ***rank*** of an element—its position in the linear order of the set—in $O(\lg n)$ time.

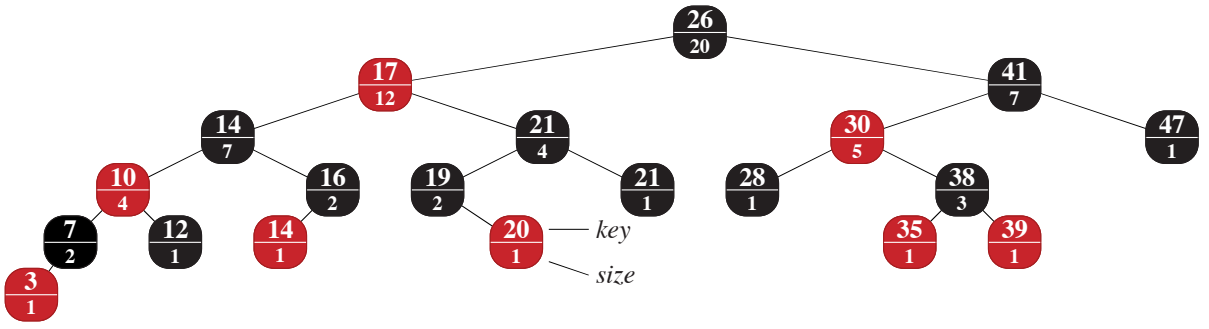


Figure 17.1 An order-statistic tree, which is an augmented red-black tree. In addition to its usual attributes, each node x has an attribute $x.size$, which is the number of nodes, other than the sentinel, in the subtree rooted at x.

Figure 17.1 shows a data structure that can support fast order-statistic operations. An ***order-statistic tree*** T is simply a red-black tree with additional information stored in each node. Each node x contains the usual red-black tree attributes $x.key$, $x.color$, $x.p$, $x.left$, and $x.right$, along with a new attribute, $x.size$. This attribute contains the number of internal nodes in the subtree rooted at x (including x itself, but not including any sentinels), that is, the size of the subtree. If we define the sentinel's size to be 0—that is, we set $T.nil.size$ to be 0—then we have the identity

$$x.size = x.left.size + x.right.size + 1 .$$

Keys need not be distinct in an order-statistic tree. For example, the tree in Figure 17.1 has two keys with value 14 and two keys with value 21. When equal keys are present, the above notion of rank is not well defined. We remove this ambiguity for an order-statistic tree by defining the rank of an element as the position at which it would be printed in an inorder walk of the tree. In Figure 17.1, for example, the key 14 stored in a black node has rank 5, and the key 14 stored in a red node has rank 6.

Retrieving the element with a given rank

Before we show how to maintain the size information during insertion and deletion, let's see how to implement two order-statistic queries that use this additional information. We begin with an operation that retrieves the element with a given rank. The procedure OS-SELECT(x, i) on the following page returns a pointer to the node containing the ith smallest key in the subtree rooted at x. To find the node with the ith smallest key in an order-statistic tree T, call OS-SELECT$(T.root, i)$.

Here is how OS-SELECT works. Line 1 computes r, the rank of node x within the subtree rooted at x. The value of $x.left.size$ is the number of nodes that come

```
OS-SELECT(x, i)
1  r = x.left.size + 1   // rank of x within the subtree rooted at x
2  if i == r
3       return x
4  elseif i < r
5       return OS-SELECT(x.left, i)
6  else return OS-SELECT(x.right, i − r)
```

before x in an inorder tree walk of the subtree rooted at x. Thus, $x.left.size + 1$ is the rank of x within the subtree rooted at x. If $i = r$, then node x is the ith smallest element, and so line 3 returns x. If $i < r$, then the ith smallest element resides in x's left subtree, and therefore, line 5 recurses on $x.left$. If $i > r$, then the ith smallest element resides in x's right subtree. Since the subtree rooted at x contains r elements that come before x's right subtree in an inorder tree walk, the ith smallest element in the subtree rooted at x is the $(i - r)$th smallest element in the subtree rooted at $x.right$. Line 6 determines this element recursively.

As an example of how OS-SELECT operates, consider a search for the 17th smallest element in the order-statistic tree of Figure 17.1. The search starts with x as the root, whose key is 26, and with $i = 17$. Since the size of 26's left subtree is 12, its rank is 13. Thus, the node with rank 17 is the $17 - 13 = 4$th smallest element in 26's right subtree. In the recursive call, x is the node with key 41, and $i = 4$. Since the size of 41's left subtree is 5, its rank within its subtree is 6. Therefore, the node with rank 4 is the 4th smallest element in 41's left subtree. In the recursive call, x is the node with key 30, and its rank within its subtree is 2. The procedure recurses once again to find the $4 - 2 = 2$nd smallest element in the subtree rooted at the node with key 38. Its left subtree has size 1, which means it is the second smallest element. Thus, the procedure returns a pointer to the node with key 38.

Because each recursive call goes down one level in the order-statistic tree, the total time for OS-SELECT is at worst proportional to the height of the tree. Since the tree is a red-black tree, its height is $O(\lg n)$, where n is the number of nodes. Thus, the running time of OS-SELECT is $O(\lg n)$ for a dynamic set of n elements.

Determining the rank of an element

Given a pointer to a node x in an order-statistic tree T, the procedure OS-RANK on the facing page returns the position of x in the linear order determined by an inorder tree walk of T.

```
OS-RANK(T, x)
1  r = x.left.size + 1          // rank of x within the subtree rooted at x
2  y = x                        // root of subtree being examined
3  while y ≠ T.root
4      if y == y.p.right                  // if root of a right subtree ...
5          r = r + y.p.left.size + 1      // ... add in parent and its left subtree
6      y = y.p                            // move y toward the root
7  return r
```

The OS-RANK procedure works as follows. You can think of node x's rank as the number of nodes preceding x in an inorder tree walk, plus 1 for x itself. OS-RANK maintains the following loop invariant:

> At the start of each iteration of the **while** loop of lines 3–6, r is the rank of $x.key$ in the subtree rooted at node y.

We use this loop invariant to show that OS-RANK works correctly as follows:

Initialization: Prior to the first iteration, line 1 sets r to be the rank of $x.key$ within the subtree rooted at x. Setting $y = x$ in line 2 makes the invariant true the first time the test in line 3 executes.

Maintenance: At the end of each iteration of the **while** loop, line 6 sets $y = y.p$. Thus, we must show that if r is the rank of $x.key$ in the subtree rooted at y at the start of the loop body, then r is the rank of $x.key$ in the subtree rooted at $y.p$ at the end of the loop body. In each iteration of the **while** loop, consider the subtree rooted at $y.p$. The value of r already includes the number of nodes in the subtree rooted at node y that precede x in an inorder walk, and so the procedure must add the nodes in the subtree rooted at y's sibling that precede x in an inorder walk, plus 1 for $y.p$ if it, too, precedes x. If y is a left child, then neither $y.p$ nor any node in $y.p$'s right subtree precedes x, and so OS-RANK leaves r alone. Otherwise, y is a right child and all the nodes in $y.p$'s left subtree precede x, as does $y.p$ itself. In this case, line 5 adds $y.p.left.size + 1$ to the current value of r.

Termination: Because each iteration of the loop moves y toward the root and the loop terminates when $y = T.root$, the loop eventually terminates. Moreover, the subtree rooted at y is the entire tree. Thus, the value of r is the rank of $x.key$ in the entire tree.

As an example, when OS-RANK runs on the order-statistic tree of Figure 17.1 to find the rank of the node with key 38, the following sequence of values of $y.key$ and r occurs at the top of the **while** loop:

iteration	$y.key$	r
1	38	2
2	30	4
3	41	4
4	26	17

The procedure returns the rank 17.

Since each iteration of the **while** loop takes $O(1)$ time, and y goes up one level in the tree with each iteration, the running time of OS-RANK is at worst proportional to the height of the tree: $O(\lg n)$ on an n-node order-statistic tree.

Maintaining subtree sizes

Given the *size* attribute in each node, OS-SELECT and OS-RANK can quickly compute order-statistic information. But if the basic modifying operations on red-black trees cannot efficiently maintain the *size* attribute, our work will have been for naught. Let's see how to maintain subtree sizes for both insertion and deletion without affecting the asymptotic running time of either operation.

Recall from Section 13.3 that insertion into a red-black tree consists of two phases. The first phase goes down the tree from the root, inserting the new node as a child of an existing node. The second phase goes up the tree, changing colors and performing rotations to maintain the red-black properties.

To maintain the subtree sizes in the first phase, simply increment $x.size$ for each node x on the simple path traversed from the root down toward the leaves. The new node added gets a *size* of 1. Since there are $O(\lg n)$ nodes on the traversed path, the additional cost of maintaining the *size* attributes is $O(\lg n)$.

In the second phase, the only structural changes to the underlying red-black tree are caused by rotations, of which there are at most two. Moreover, a rotation is a local operation: only two nodes have their *size* attributes invalidated. The link around which the rotation is performed is incident on these two nodes. Referring to the code for LEFT-ROTATE(T, x) on page 336, add the following lines:

```
13  y.size = x.size
14  x.size = x.left.size + x.right.size + 1
```

Figure 17.2 illustrates how the attributes are updated. The change to RIGHT-ROTATE is symmetric.

Since inserting into a red-black tree requires at most two rotations, updating the *size* attributes in the second phase costs only $O(1)$ additional time. Thus, the total time for insertion into an n-node order-statistic tree is $O(\lg n)$, which is asymptotically the same as for an ordinary red-black tree.

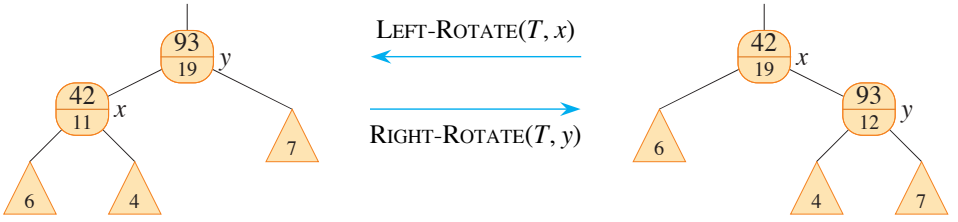


Figure 17.2 Updating subtree sizes during rotations. The updates are local, requiring only the *size* information stored in x, y, and the roots of the subtrees shown as triangles.

Deletion from a red-black tree also consists of two phases: the first operates on the underlying search tree, and the second causes at most three rotations and otherwise performs no structural changes. (See Section 13.4.) The first phase removes one node z from the tree and could move at most two other nodes within the tree (nodes y and x in Figure 12.4 on page 323). To update the subtree sizes, simply traverse a simple path from the lowest node that moves (starting from its original position within the tree) up to the root, decrementing the *size* attribute of each node on the path. Since this path has length $O(\lg n)$ in an n-node red-black tree, the additional time spent maintaining *size* attributes in the first phase is $O(\lg n)$. For the $O(1)$ rotations in the second phase of deletion, handle them in the same manner as for insertion. Thus, both insertion and deletion, including maintaining the *size* attributes, take $O(\lg n)$ time for an n-node order-statistic tree.

Exercises

17.1-1

Show how OS-SELECT$(T.root, 10)$ operates on the red-black tree T shown in Figure 17.1.

17.1-2

Show how OS-RANK(T, x) operates on the red-black tree T shown in Figure 17.1 and the node x with $x.key = 35$.

17.1-3

Write a nonrecursive version of OS-SELECT.

17.1-4

Write a procedure OS-KEY-RANK(T, k) that takes an order-statistic tree T and a key k and returns the rank of k in the dynamic set represented by T. Assume that the keys of T are distinct.

17.1-5
Given an element x in an n-node order-statistic tree and a natural number i, show how to determine the ith successor of x in the linear order of the tree in $O(\lg n)$ time.

17.1-6
The procedures OS-SELECT and OS-RANK use the *size* attribute of a node only to compute a rank. Suppose that you store in each node its rank in the subtree of which it is the root instead of the *size* attribute. Show how to maintain this information during insertion and deletion. (Remember that these two operations can cause rotations.)

17.1-7
Show how to use an order-statistic tree to count the number of inversions (see Problem 2-4 on page 47) in an array of n distinct elements in $O(n \lg n)$ time.

★ ***17.1-8***
Consider n chords on a circle, each defined by its endpoints. Describe an $O(n \lg n)$-time algorithm to determine the number of pairs of chords that intersect inside the circle. (For example, if the n chords are all diameters that meet at the center, then the answer is $\binom{n}{2}$.) Assume that no two chords share an endpoint.

17.2 How to augment a data structure

The process of augmenting a basic data structure to support additional functionality occurs quite frequently in algorithm design. We'll use it again in the next section to design a data structure that supports operations on intervals. This section examines the steps involved in such augmentation. It includes a useful theorem that allows you to augment red-black trees easily in many cases.

You can break the process of augmenting a data structure into four steps:

1. Choose an underlying data structure.
2. Determine additional information to maintain in the underlying data structure.
3. Verify that you can maintain the additional information for the basic modifying operations on the underlying data structure.
4. Develop new operations.

As with any prescriptive design method, you'll rarely be able to follow the steps precisely in the order given. Most design work contains an element of trial and error, and progress on all steps usually proceeds in parallel. There is no point,

for example, in determining additional information and developing new operations (steps 2 and 4) if you cannot maintain the additional information efficiently. Nevertheless, this four-step method provides a good focus for your efforts in augmenting a data structure, and it is also a good framework for documenting an augmented data structure.

We followed these four steps in Section 17.1 to design order-statistic trees. For step 1, we chose red-black trees as the underlying data structure. Red-black trees seemed like a good starting point because they efficiently support other dynamic-set operations on a total order, such as MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR.

In Step 2, we added the *size* attribute, so that each node x stores the size of the subtree rooted at x. Generally, the additional information makes operations more efficient. For example, it is possible to implement OS-SELECT and OS-RANK using just the keys stored in the tree, but then they would not run in $O(\lg n)$ time. Sometimes, the additional information is pointer information rather than data, as in Exercise 17.2-1.

For step 3, we ensured that insertion and deletion can maintain the *size* attributes while still running in $O(\lg n)$ time. Ideally, you would like to update only a few elements of the data structure in order to maintain the additional information. For example, if each node simply stores its rank in the tree, the OS-SELECT and OS-RANK procedures run quickly, but inserting a new minimum element might cause a change to this information in every node of the tree. Because we chose to store subtree sizes instead, inserting a new element causes information to change in only $O(\lg n)$ nodes.

In Step 4, we developed the operations OS-SELECT and OS-RANK. After all, the need for new operations is why anyone bothers to augment a data structure in the first place. Occasionally, rather than developing new operations, you can use the additional information to expedite existing ones, as in Exercise 17.2-1.

Augmenting red-black trees

When red-black trees underlie an augmented data structure, we can prove that insertion and deletion can always efficiently maintain certain kinds of additional information, thereby simplifying step 3. The proof of the following theorem is similar to the argument from Section 17.1 that we can maintain the *size* attribute for order-statistic trees.

Theorem 17.1 (Augmenting a red-black tree)
Let f be an attribute that augments a red-black tree T of n nodes, and suppose that the value of f for each node x depends only the information in nodes x, $x.left$, and $x.right$ (possibly including $x.left.f$ and $x.right.f$), and that the value of $x.f$ can

be computed from this information in $O(1)$ time. Then, the insertion and deletion operations can maintain the values of f in all nodes of T without asymptotically affecting the $O(\lg n)$ running times of these operations.

Proof The main idea of the proof is that a change to an f attribute in a node x propagates only to ancestors of x in the tree. That is, changing $x.f$ may require $x.p.f$ to be updated, but nothing else; updating $x.p.f$ may require $x.p.p.f$ to be updated, but nothing else; and so on up the tree. After updating $T.root.f$, no other node depends on the new value, and so the process terminates. Since the height of a red-black tree is $O(\lg n)$, changing an f attribute in a node costs $O(\lg n)$ time in updating all nodes that depend on the change.

As we saw in Section 13.3, insertion of a node x into red-black tree T consists of two phases. If the tree T is empty, then the first phase simply makes x be the root of T. If T is not empty, then the first phase inserts x as a child of an existing node. Because we assume that the value of $x.f$ depends only on information in the other attributes of x itself and the information in x's children, and because x's children are both the sentinel $T.nil$, it takes only $O(1)$ time to compute the value of $x.f$. Having computed $x.f$, the change propagates up the tree. Thus, the total time for the first phase of insertion is $O(\lg n)$. During the second phase, the only structural changes to the tree come from rotations. Since only two nodes change in a rotation, but a change to an attribute might need to propagate up to the root, the total time for updating the f attributes is $O(\lg n)$ per rotation. Since the number of rotations during insertion is at most two, the total time for insertion is $O(\lg n)$.

Like insertion, deletion has two phases, as Section 13.4 discusses. In the first phase, changes to the tree occur when a node is deleted, and at most two other nodes could move within the tree. Propagating the updates to f caused by these changes costs at most $O(\lg n)$, since the changes modify the tree locally along a simple path from the lowest changed node to the root. Fixing up the red-black tree during the second phase requires at most three rotations, and each rotation requires at most $O(\lg n)$ time to propagate the updates to f. Thus, like insertion, the total time for deletion is $O(\lg n)$. ■

In many cases, such as maintaining the *size* attributes in order-statistic trees, the cost of updating after a rotation is $O(1)$, rather than the $O(\lg n)$ derived in the proof of Theorem 17.1. Exercise 17.2-3 gives an example.

On the other hand, when an update after a rotation requires a traversal all the way up to the root, it is important that insertion into and deletion from a red-black tree require a constant number of rotations. The chapter notes for Chapter 13 list other schemes for balancing search trees that do not bound the number of rotations per insertion or deletion by a constant. If each operation might require $\Theta(\lg n)$ rota-

tions and each rotation traverses a path up to the root, then a single operation could require $\Theta(\lg^2 n)$ time, rather than the $O(\lg n)$ time bound given by Theorem 17.1.

Exercises

17.2-1

Show, by adding pointers to the nodes, how to support each of the dynamic-set queries MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR in $O(1)$ worst-case time on an augmented order-statistic tree. The asymptotic performance of other operations on order-statistic trees should not be affected.

17.2-2

Can you maintain the black-heights of nodes in a red-black tree as attributes in the nodes of the tree without affecting the asymptotic performance of any of the red-black tree operations? Show how, or argue why not. How about maintaining the depths of nodes?

17.2-3

Let $\otimes$ be an associative binary operator, and let a be an attribute maintained in each node of a red-black tree. Suppose that you want to include in each node x an additional attribute f such that $x.f = x_1.a \otimes x_2.a \otimes \cdots \otimes x_m.a$, where $x_1, x_2, \ldots, x_m$ is the inorder listing of nodes in the subtree rooted at x. Show how to update the f attributes in $O(1)$ time after a rotation. Modify your argument slightly to apply it to the *size* attributes in order-statistic trees.

17.3 Interval trees

This section shows how to augment red-black trees to support operations on dynamic sets of intervals. In this section, we'll assume that intervals are closed. Extending the results to open and half-open intervals is conceptually straightforward. (See page 1157 for definitions of closed, open, and half-open intervals.)

Intervals are convenient for representing events that each occupy a continuous period of time. For example, you could query a database of time intervals to find out which events occurred during a given interval. The data structure in this section provides an efficient means for maintaining such an interval database.

A simple way to represent an interval $[t_1, t_2]$ is as an object i with attributes $i.low = t_1$ (the ***low endpoint***) and $i.high = t_2$ (the ***high endpoint***). We say that intervals i and i' ***overlap*** if $i \cap i' \neq \emptyset$, that is, if $i.low \leq i'.high$ and $i'.low \leq i.high$.

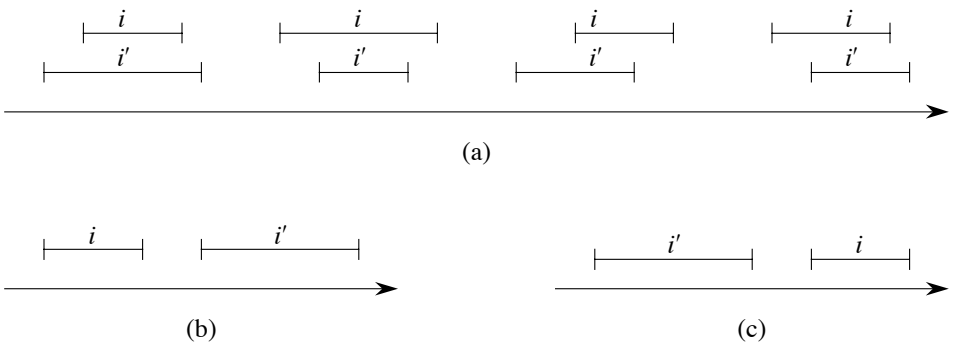


Figure 17.3 The interval trichotomy for two closed intervals i and i'. **(a)** If i and i' overlap, there are four situations, and in each, $i.low \le i'.high$ and $i'.low \le i.high$. **(b)** The intervals do not overlap, and $i.high < i'.low$. **(c)** The intervals do not overlap, and $i'.high < i.low$.

As Figure 17.3 shows, any two intervals i and i' satisfy the ***interval trichotomy***, that is, exactly one of the following three properties holds:

a. i and i' overlap,

b. i is to the left of i' (i.e., $i.high < i'.low$),

c. i is to the right of i' (i.e., $i'.high < i.low$).

An ***interval tree*** is a red-black tree that maintains a dynamic set of elements, with each element x containing an interval $x.int$. Interval trees support the following operations:

INTERVAL-INSERT(T, x) adds the element x, whose *int* attribute is assumed to contain an interval, to the interval tree T.

INTERVAL-DELETE(T, x) removes the element x from the interval tree T.

INTERVAL-SEARCH(T, i) returns a pointer to an element x in the interval tree T such that $x.int$ overlaps interval i, or a pointer to the sentinel $T.nil$ if no such element belongs to the set.

Figure 17.4 shows how an interval tree represents a set of intervals. The four-step method from Section 17.2 will guide our design of an interval tree and the operations that run on it.

Step 1: Underlying data structure

A red-black tree serves as the underlying data structure. Each node x contains an interval $x.int$. The key of x is the low endpoint, $x.int.low$, of the interval. Thus, an inorder tree walk of the data structure lists the intervals in sorted order by low endpoint.

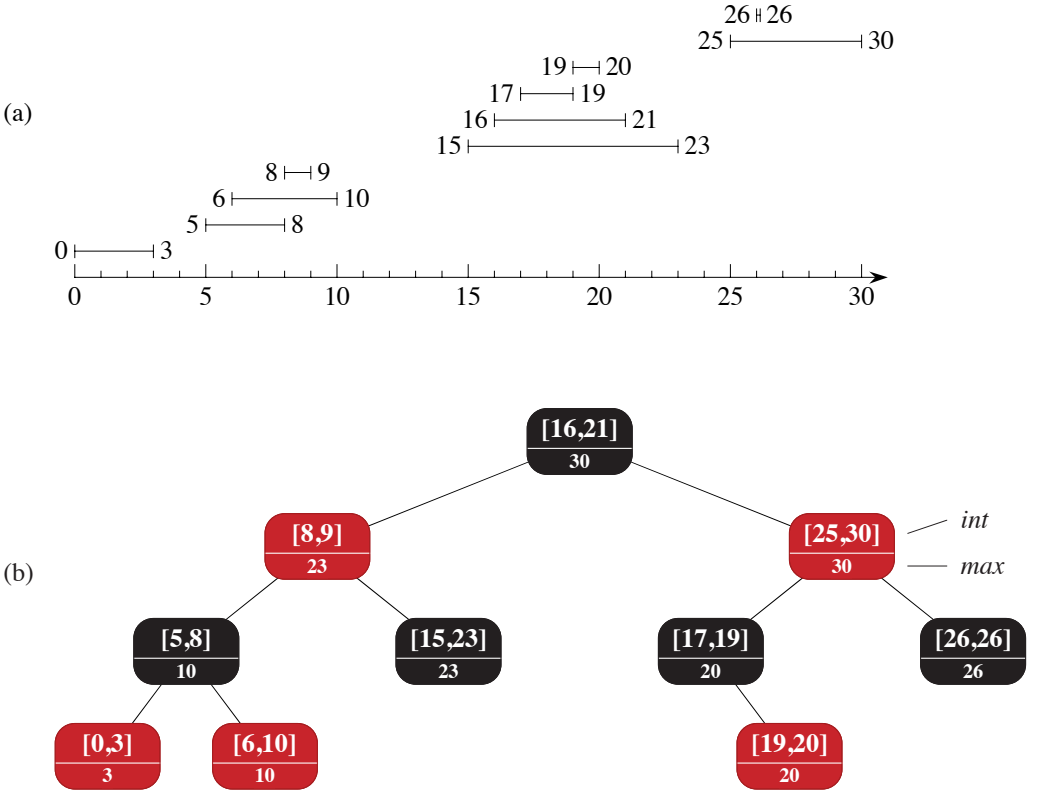


Figure 17.4 An interval tree. **(a)** A set of 10 intervals, shown sorted bottom to top by left endpoint. **(b)** The interval tree that represents them. Each node x contains an interval, shown above the dashed line, and the maximum value of any interval endpoint in the subtree rooted at x, shown below the dashed line. An inorder tree walk of the tree lists the nodes in sorted order by left endpoint.

Step 2: Additional information

In addition to the intervals themselves, each node x contains a value $x.max$, which is the maximum value of any interval endpoint stored in the subtree rooted at x.

Step 3: Maintaining the information

We must verify that insertion and deletion take $O(\lg n)$ time on an interval tree of n nodes. It is simple enough to determine $x.max$ in $O(1)$ time, given interval $x.int$ and the max values of node x's children:

$$x.max = \max\{x.int.high, x.left.max, x.right.max\} \ .$$

Thus, by Theorem 17.1, insertion and deletion run in $O(\lg n)$ time. In fact, you can use either Exercise 17.2-3 or 17.3-1 to show how to update all the *max* attributes that change after a rotation in just $O(1)$ time.

Step 4: Developing new operations

The only new operation is INTERVAL-SEARCH(T, i), which finds a node in tree T whose interval overlaps interval i. If there is no interval in the tree that overlaps i, the procedure returns a pointer to the sentinel $T.nil$.

```
INTERVAL-SEARCH(T, i)
1  x = T.root
2  while x ≠ T.nil and i does not overlap x.int
3      if x.left ≠ T.nil and x.left.max ≥ i.low
4          x = x.left    // overlap in left subtree or no overlap in right subtree
5      else x = x.right  // no overlap in left subtree
6  return x
```

The search for an interval that overlaps i starts at the root of the tree and proceeds downward. It terminates when either it finds an overlapping interval or it reaches the sentinel $T.nil$. Since each iteration of the basic loop takes $O(1)$ time, and since the height of an n-node red-black tree is $O(\lg n)$, the INTERVAL-SEARCH procedure takes $O(\lg n)$ time.

Before we see why INTERVAL-SEARCH is correct, let's examine how it works on the interval tree in Figure 17.4. Let's look for an interval that overlaps the interval $i = [22, 25]$. Begin with x as the root, which contains $[16, 21]$ and does not overlap i. Since $x.left.max = 23$ is greater than $i.low = 22$, the loop continues with x as the left child of the root—the node containing $[8, 9]$, which also does not overlap i. This time, $x.left.max = 10$ is less than $i.low = 22$, and so the loop continues with the right child of x as the new x. Because the interval $[15, 23]$ stored in this node overlaps i, the procedure returns this node.

Now let's try an unsuccessful search, for an interval that overlaps $i = [11, 14]$ in the interval tree of Figure 17.4. Again, begin with x as the root. Since the root's interval $[16, 21]$ does not overlap i, and since $x.left.max = 23$ is greater than $i.low = 11$, go left to the node containing $[8, 9]$. Interval $[8, 9]$ does not overlap i, and $x.left.max = 10$ is less than $i.low = 11$, and so the search goes right. (No interval in the left subtree overlaps i.) Interval $[15, 23]$ does not overlap i, and its left child is $T.nil$, so again the search goes right, the loop terminates, and INTERVAL-SEARCH returns the sentinel $T.nil$.

To see why INTERVAL-SEARCH is correct, we must understand why it suffices to examine a single path from the root. The basic idea is that at any node x, if $x.int$ does not overlap i, the search always proceeds in a safe direction: the search will definitely find an overlapping interval if the tree contains one. The following theorem states this property more precisely.

Theorem 17.2
Any execution of INTERVAL-SEARCH(T, i) either returns a node whose interval overlaps i, or it returns $T.nil$ and the tree T contains no node whose interval overlaps i.

Proof The **while** loop of lines 2–5 terminates when either $x = T.nil$ or i overlaps $x.int$. In the latter case, it is certainly correct to return x. Therefore, we focus on the former case, in which the **while** loop terminates because $x = T.nil$, which is the node that INTERVAL-SEARCH returns.

We'll prove that if the procedure returns $T.nil$, then it did not miss any intervals in T that overlap i. The idea is to show that whether the search goes left in line 4 or right in line 5, it always heads toward a node containing an interval overlapping i, if any such interval exists. In particular, we'll prove that

1. If the search goes left in line 4, then the left subtree of node x contains an interval that overlaps i or the right subtree of x contains no interval that overlaps i. Therefore, even if x's left subtree contains no interval that overlaps i but the search goes left, it does not make a mistake, because x's right subtree does not contain an interval overlapping i, either.
2. If the search goes right in line 5, then the left subtree of x contains no interval that overlaps i. Thus, if the search goes right, it does not make a mistake.

For both cases, we rely on the interval trichotomy. Let's start with the case where the search goes right, whose proof is simpler. By the tests in line 3, we know that $x.left = T.nil$ or $x.left.max < i.low$. If $x.left = T.nil$, then x's left subtree contains no interval that overlaps i, since it contains no intervals at all. Now suppose that $x.left \neq T.nil$, so that we must have $x.left.max < i.low$. Consider any interval i' in x's left subtree. Because $x.left.max$ is the maximum endpoint in x's left subtree, we have $i'.high \leq x.left.max$. Thus, as Figure 17.5(a) shows,

$$\begin{aligned} i'.high &\leq x.left.max \\ &< i.low . \end{aligned}$$

By the interval trichotomy, therefore, intervals i and i' do not overlap, and so x's left subtree contains no interval that overlaps i.

Now we examine the case in which the search goes left. If the left subtree of node x contains an interval that overlaps i, we're done, so let's assume that no node

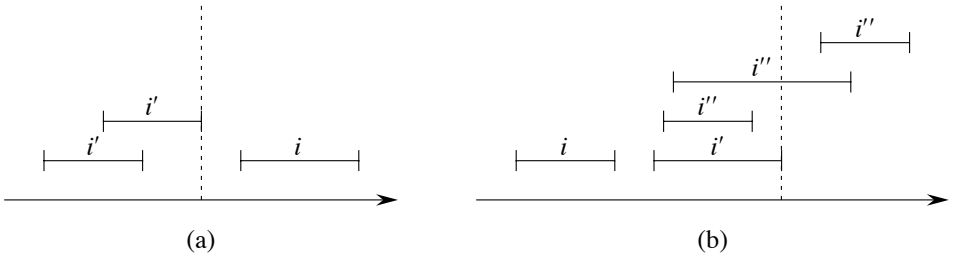


Figure 17.5 Intervals in the proof of Theorem 17.2. The value of $x.left.max$ is shown in each case as a dashed line. **(a)** The search goes right. No interval i' in x's left subtree can overlap i. **(b)** The search goes left. The left subtree of x contains an interval that overlaps i (situation not shown), or x's left subtree contains an interval i' such that $i'.high = x.left.max$. Since i does not overlap i', neither does it overlap any interval i'' in x's right subtree, since $i'.low \leq i''.low$.

in x's left subtree overlaps i. We need to show that in this case, no node in x's right subtree overlaps i, so that going left will not miss any overlaps in x's right subtree. By the tests in line 3, the left subtree of x is not empty and $x.left.max \geq i.low$. By the definition of the *max* attribute, x's left subtree contains some interval i' such that

$$\begin{aligned} i'.high &= x.left.max \\ &\geq i.low , \end{aligned}$$

as illustrated in Figure 17.5(b). Since i' is in x's left subtree, it does not overlap i, and since $i'.high \geq i.low$, the interval trichotomy tells us that $i.high < i'.low$. Now we bring in the property that interval trees are keyed on the low endpoints of intervals. Because i' is in x's left subtree, we have $i'.low \leq x.int.low$. Now consider any interval i'' in x's right subtree, so that $x.int.low \leq i''.low$. Putting inequalities together, we get

$$\begin{aligned} i.high &< i'.low \\ &\leq x.int.low \\ &\leq i''.low . \end{aligned}$$

Because $i.high < i''.low$, the interval trichotomy tells us that i and i'' do not overlap. Since we chose i'' as any interval in x's right subtree, no node in x's right subtree overlaps i. ■

Thus, the INTERVAL-SEARCH procedure works correctly.

Exercises

17.3-1

Write pseudocode for LEFT-ROTATE that operates on nodes in an interval tree and updates all the *max* attributes that change in $O(1)$ time.

17.3-2

Describe an efficient algorithm that, given an interval i, returns an interval overlapping i that has the minimum low endpoint, or $T.nil$ if no such interval exists.

17.3-3

Given an interval tree T and an interval i, describe how to list all intervals in T that overlap i in $O(\min\{n, k \lg n\})$ time, where k is the number of intervals in the output list. (*Hint:* One simple method makes several queries, modifying the tree between queries. A slightly more complicated method does not modify the tree.)

17.3-4

Suggest modifications to the interval-tree procedures to support the new operation INTERVAL-SEARCH-EXACTLY(T, i), where T is an interval tree and i is an interval. The operation should return a pointer to a node x in T such that $x.int.low = i.low$ and $x.int.high = i.high$, or $T.nil$ if T contains no such node. All operations, including INTERVAL-SEARCH-EXACTLY, should run in $O(\lg n)$ time on an n-node interval tree.

17.3-5

Show how to maintain a dynamic set Q of numbers that supports the operation MIN-GAP, which gives the absolute value of the difference of the two closest numbers in Q. For example, if we have $Q = \{1, 5, 9, 15, 18, 22\}$, then MIN-GAP$(Q)$ returns 3, since 15 and 18 are the two closest numbers in Q. Make the operations INSERT, DELETE, SEARCH, and MIN-GAP as efficient as possible, and analyze their running times.

★ ### 17.3-6

VLSI databases commonly represent an integrated circuit as a list of rectangles. Assume that each rectangle is rectilinearly oriented (sides parallel to the x- and y-axes), so that each rectangle is represented by four values: its minimum and maximum x- and y-coordinates. Give an $O(n \lg n)$-time algorithm to decide whether a set of n rectangles so represented contains two rectangles that overlap. Your algorithm need not report all intersecting pairs, but it must report that an overlap exists if one rectangle entirely covers another, even if the boundary lines do not intersect. (*Hint:* Move a "sweep" line across the set of rectangles.)

Problems

17-1 Point of maximum overlap

You wish to keep track of a ***point of maximum overlap*** in a set of intervals—a point with the largest number of intervals in the set that overlap it.

a. Show that there is always a point of maximum overlap that is an endpoint of one of the intervals.

b. Design a data structure that efficiently supports the operations INTERVAL-INSERT, INTERVAL-DELETE, and FIND-POM, which returns a point of maximum overlap. (*Hint:* Keep a red-black tree of all the endpoints. Associate a value of $+1$ with each left endpoint, and associate a value of -1 with each right endpoint. Augment each node of the tree with some extra information to maintain the point of maximum overlap.)

17-2 Josephus permutation

We define the ***Josephus problem*** as follows. A group of n people form a circle, and we are given a positive integer $m \leq n$. Beginning with a designated first person, proceed around the circle, removing every mth person. After each person is removed, counting continues around the circle that remains. This process continues until nobody remains in the circle. The order in which the people are removed from the circle defines the ***(n, m)-Josephus permutation*** of the integers $1, 2, \ldots, n$. For example, the $(7, 3)$-Josephus permutation is $\langle 3, 6, 2, 7, 5, 1, 4 \rangle$.

a. Suppose that m is a constant. Describe an $O(n)$-time algorithm that, given an integer n, outputs the (n, m)-Josephus permutation.

b. Suppose that m is not necessarily a constant. Describe an $O(n \lg n)$-time algorithm that, given integers n and m, outputs the (n, m)-Josephus permutation.

Chapter notes

In their book, Preparata and Shamos [364] describe several of the interval trees that appear in the literature, citing work by H. Edelsbrunner (1980) and E. M. McCreight (1981). The book details an interval tree that, given a static database of n intervals, allows us to enumerate all k intervals that overlap a given query interval in $O(k + \lg n)$ time.

18 B-Trees

B-trees are balanced search trees designed to work well on disk drives or other direct-access secondary storage devices. B-trees are similar to red-black trees (Chapter 13), but they are better at minimizing the number of operations that access disks. (We often say just "disk" instead of "disk drive.") Many database systems use B-trees, or variants of B-trees, to store information.

B-trees differ from red-black trees in that B-tree nodes may have many children, from a few to thousands. That is, the "branching factor" of a B-tree can be quite large, although it usually depends on characteristics of the disk drive used. B-trees are similar to red-black trees in that every n-node B-tree has height $O(\lg n)$, so that B-trees can implement many dynamic-set operations in $O(\lg n)$ time. But a B-tree has a larger branching factor than a red-black tree, so the base of the logarithm that expresses its height is larger, and hence its height can be considerably lower.

B-trees generalize binary search trees in a natural manner. Figure 18.1 shows a simple B-tree. If an internal B-tree node x contains $x.n$ keys, then x has $x.n + 1$ children. The keys in node x serve as dividing points separating the range of keys handled by x into $x.n + 1$ subranges, each handled by one child of x. A search for a key in a B-tree makes an $(x.n + 1)$-way decision based on comparisons with the $x.n$ keys stored at node x. An internal node contains pointers to its children, but a leaf node does not.

Section 18.1 gives a precise definition of B-trees and proves that the height of a B-tree grows only logarithmically with the number of nodes it contains. Section 18.2 describes how to search for a key and insert a key into a B-tree, and Section 18.3 discusses deletion. Before proceeding, however, we need to ask why we evaluate data structures designed to work on a disk drive differently from data structures designed to work in main random-access memory.

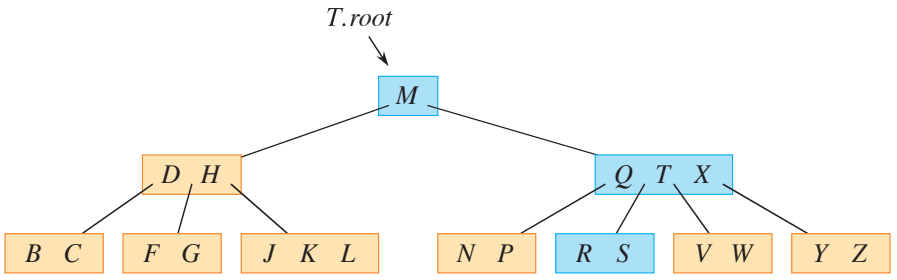


Figure 18.1 A B-tree whose keys are the consonants of English. An internal node x containing $x.n$ keys has $x.n + 1$ children. All leaves are at the same depth in the tree. The blue nodes are examined in a search for the letter R.

Data structures on secondary storage

Computer systems take advantage of various technologies that provide memory capacity. The ***main memory*** of a computer system normally consists of silicon memory chips. This technology is typically more than an order of magnitude more expensive per bit stored than magnetic storage technology, such as tapes or disk drives. Most computer systems also have ***secondary storage*** based on solid-state drives (SSDs) or magnetic disk drives. The amount of such secondary storage often exceeds the amount of primary memory by one to two orders of magnitude. SSDs have faster access times than magnetic disk drives, which are mechanical devices. In recent years, SSD capacities have increased while their prices have decreased. Magnetic disk drives typically have much higher capacities than SSDs, and they remain a more cost-effective means for storing massive amounts of information. Disk drives that store several terabytes[1] can be found for under $100.

Figure 18.2 shows a typical disk drive. The drive consists of one or more ***platters***, which rotate at a constant speed around a common ***spindle***. A magnetizable material covers the surface of each platter. The drive reads and writes each platter by a ***head*** at the end of an ***arm***. The arms can move their heads toward or away from the spindle. The surface that passes underneath a given head when it is stationary is called a ***track***.

Although disk drives are cheaper and have higher capacity than main memory, they are much, much slower because they have moving mechanical parts. The mechanical motion has two components: platter rotation and arm movement. As of this writing, commodity disk drives rotate at speeds of 5400–15,000 revolutions per minute (RPM). Typical speeds are 15,000 RPM in server-grade drives, 7200 RPM

[1] When specifying disk capacities, one terabyte is one trillion bytes, rather than 2^{40} bytes.

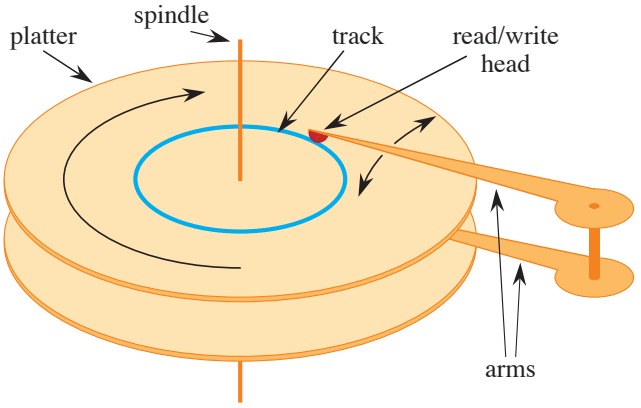


Figure 18.2 A typical magnetic disk drive. It consists of one or more platters covered with a magnetizable material (two platters are shown here) that rotate around a spindle. Each platter is read and written with a head, shown in red, at the end of an arm. Arms rotate around a common pivot axis. A track, drawn in blue, is the surface that passes beneath the read/write head when the head is stationary.

in drives for desktops, and 5400 RPM in drives for laptops. Although 7200 RPM may seem fast, one rotation takes 8.33 milliseconds, which is over 5 orders of magnitude longer than the 50 nanosecond access times (more or less) commonly found for main memory. In other words, if a computer waits a full rotation for a particular item to come under the read/write head, it could access main memory more than 100,000 times during that span. The average wait is only half a rotation, but still, the difference in access times for main memory compared with disk drives is enormous. Moving the arms also takes some time. As of this writing, average access times for commodity disk drives are around 4 milliseconds.

In order to amortize the time spent waiting for mechanical movements, also known as ***latency***, disk drives access not just one item but several at a time. Information is divided into a number of equal-sized ***blocks*** of bits that appear consecutively within tracks, and each disk read or write is of one or more entire blocks.[2] Typical disk drives have block sizes running from 512 to 4096 bytes. Once the read/write head is positioned correctly and the platter has rotated to the beginning of the desired block, reading or writing a magnetic disk drive is entirely electronic (aside from the rotation of the platter), and the disk drive can quickly read or write large amounts of data.

---

[2] SSDs also exhibit greater latency than main memory and access data in blocks.

Often, accessing a block of information and reading it from a disk drive takes longer than processing all the information read. For this reason, in this chapter we'll look separately at the two principal components of the running time:

- the number of disk accesses, and
- the CPU (computing) time.

We measure the number of disk accesses in terms of the number of blocks of information that need to be read from or written to the disk drive. Although disk-access time is not constant—it depends on the distance between the current track and the desired track and also on the initial rotational position of the platters—the number of blocks read or written provides a good first-order approximation of the total time spent accessing the disk drive.

In a typical B-tree application, the amount of data handled is so large that all the data do not fit into main memory at once. The B-tree algorithms copy selected blocks from disk into main memory as needed and write back onto disk the blocks that have changed. B-tree algorithms keep only a constant number of blocks in main memory at any time, and thus the size of main memory does not limit the size of B-trees that can be handled.

B-tree procedures need to be able to read information from disk into main memory and write information from main memory to disk. Consider some object x. If x is currently in the computer's main memory, then the code can refer to the attributes of x as usual: $x.key$, for example. If x resides on disk, however, then the procedure must perform the operation DISK-READ(x) to read the block containing object x into main memory before it can refer to x's attributes. (Assume that if x is already in main memory, then DISK-READ(x) requires no disk accesses: it is a "no-op.") Similarly, procedures call DISK-WRITE(x) to save any changes that have been made to the attributes of object x by writing to disk the block containing x. Thus, the typical pattern for working with an object is as follows:

```
x = a pointer to some object
DISK-READ(x)
operations that access and/or modify the attributes of x
DISK-WRITE(x)            // omitted if no attributes of x were changed
other operations that access but do not modify attributes of x
```

The system can keep only a limited number of blocks in main memory at any one time. Our B-tree algorithms assume that the system automatically flushes from main memory blocks that are no longer in use.

Since in most systems the running time of a B-tree algorithm depends primarily on the number of DISK-READ and DISK-WRITE operations it performs, we

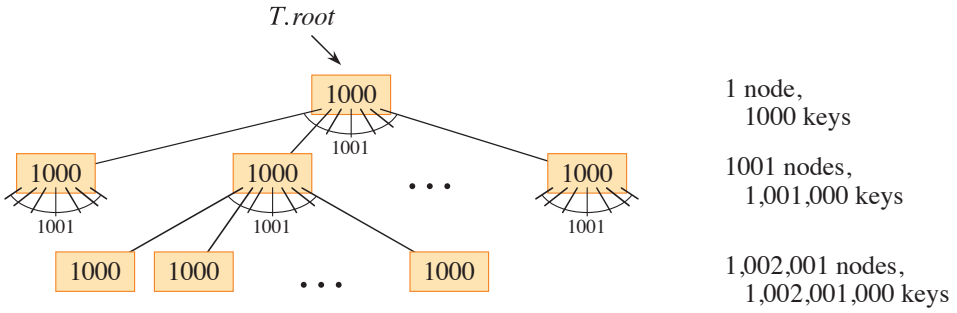


Figure 18.3 A B-tree of height 2 containing over one billion keys. Shown inside each node x is $x.n$, the number of keys in x. Each internal node and leaf contains 1000 keys. This B-tree has 1001 nodes at depth 1 and over one million leaves at depth 2.

typically want each of these operations to read or write as much information as possible. Thus, a B-tree node is usually as large as a whole disk block, and this size limits the number of children a B-tree node can have.

Large B-trees stored on disk drives often have branching factors between 50 and 2000, depending on the size of a key relative to the size of a block. A large branching factor dramatically reduces both the height of the tree and the number of disk accesses required to find any key. Figure 18.3 shows a B-tree with a branching factor of 1001 and height 2 that can store over one billion keys. Nevertheless, if the root node is kept permanently in main memory, at most two disk accesses suffice to find any key in this tree.

18.1 Definition of B-trees

To keep things simple, let's assume, as we have for binary search trees and red-black trees, that any satellite information associated with a key resides in the same node as the key. In practice, you might actually store with each key just a pointer to another disk block containing the satellite information for that key. The pseudocode in this chapter implicitly assumes that the satellite information associated with a key, or the pointer to such satellite information, travels with the key whenever the key is moved from node to node. A common variant on a B-tree, known as a ***B*$^+$*-tree***, stores all the satellite information in the leaves and stores only keys and child pointers in the internal nodes, thus maximizing the branching factor of the internal nodes.

A ***B-tree*** T is a rooted tree with root $T.root$ having the following properties:

1. Every node x has the following attributes:
   a. $x.n$, the number of keys currently stored in node x,
   b. the $x.n$ keys themselves, $x.key_1, x.key_2, \ldots, x.key_{x.n}$, stored in monotonically increasing order, so that $x.key_1 \le x.key_2 \le \cdots \le x.key_{x.n}$,
   c. $x.leaf$, a boolean value that is TRUE if x is a leaf and FALSE if x is an internal node.
2. Each internal node x also contains $x.n + 1$ pointers $x.c_1, x.c_2, \ldots, x.c_{x.n+1}$ to its children. Leaf nodes have no children, and so their c_i attributes are undefined.
3. The keys $x.key_i$ separate the ranges of keys stored in each subtree: if k_i is any key stored in the subtree with root $x.c_i$, then
   $$k_1 \le x.key_1 \le k_2 \le x.key_2 \le \cdots \le x.key_{x.n} \le k_{x.n+1} .$$
4. All leaves have the same depth, which is the tree's height h.
5. Nodes have lower and upper bounds on the number of keys they can contain, expressed in terms of a fixed integer $t \ge 2$ called the ***minimum degree*** of the B-tree:
   a. Every node other than the root must have at least $t - 1$ keys. Every internal node other than the root thus has at least t children. If the tree is nonempty, the root must have at least one key.
   b. Every node may contain at most $2t - 1$ keys. Therefore, an internal node may have at most $2t$ children. We say that a node is ***full*** if it contains exactly $2t - 1$ keys.[3]

The simplest B-tree occurs when $t = 2$. Every internal node then has either 2, 3, or 4 children, and it is a ***2-3-4 tree***. In practice, however, much larger values of t yield B-trees with smaller height.

The height of a B-tree

The number of disk accesses required for most operations on a B-tree is proportional to the height of the B-tree. The following theorem bounds the worst-case height of a B-tree.

---

[3] Another common variant on a B-tree, known as a ***B*-tree***, requires each internal node to be at least 2/3 full, rather than at least half full, as a B-tree requires.

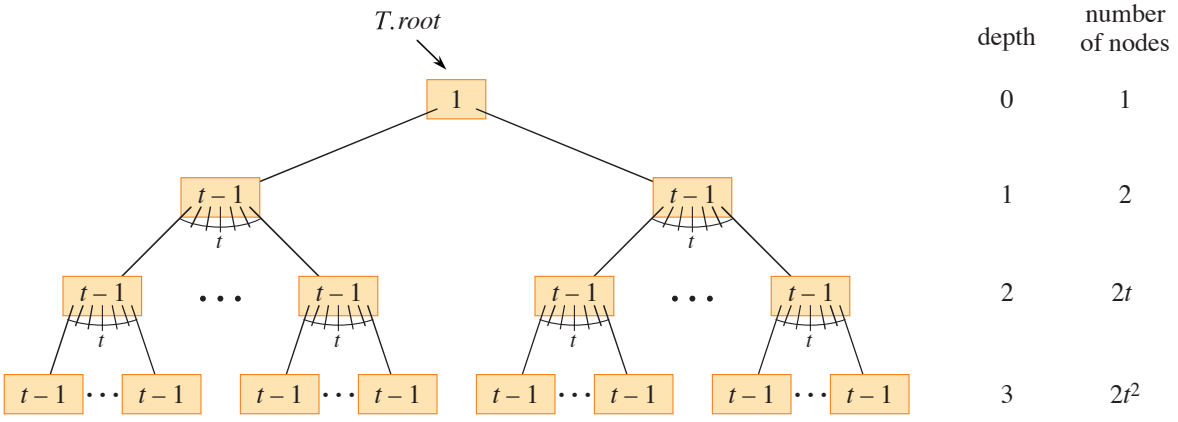


Figure 18.4 A B-tree of height 3 containing a minimum possible number of keys. Shown inside each node x is $x.n$.

Theorem 18.1

If $n \geq 1$, then for any n-key B-tree T of height h and minimum degree $t \geq 2$,

$$h \leq \log_t \frac{n+1}{2} .$$

Proof By definition, the root of a nonempty B-tree T contains at least one key, and all other nodes contain at least $t - 1$ keys. Let h be the height of T. Then T contains at least 2 nodes at depth 1, at least $2t$ nodes at depth 2, at least $2t^2$ nodes at depth 3, and so on, until at depth h, it has at least $2t^{h-1}$ nodes. Figure 18.4 illustrates such a tree for $h = 3$. The number n of keys therefore satisfies the inequality

$$\begin{aligned} n &\geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} \\ &= 1 + 2(t-1)\left(\frac{t^h - 1}{t-1}\right) \quad \text{(by equation (A.6) on page 1142)} \\ &= 2t^h - 1 , \end{aligned}$$

so that $t^h \leq (n+1)/2$. Taking base-t logarithms of both sides proves the theorem. ■

You can see the power of B-trees as compared with red-black trees. Although the height of the tree grows as $O(\log n)$ in both cases (recall that t is a constant), for B-trees the base of the logarithm can be many times larger. Thus, B-trees save

a factor of about $\lg t$ over red-black trees in the number of nodes examined for most tree operations. Because examining an arbitrary node in a tree usually entails accessing the disk, B-trees avoid a substantial number of disk accesses.

Exercises

18.1-1
Why isn't a minimum degree of $t = 1$ allowed?

18.1-2
For what values of t is the tree of Figure 18.1 a legal B-tree?

18.1-3
Show all legal B-trees of minimum degree 2 that store the keys $1, 2, 3, 4, 5$.

18.1-4
As a function of the minimum degree t, what is the maximum number of keys that can be stored in a B-tree of height h?

18.1-5
Describe the data structure that results if each black node in a red-black tree absorbs its red children, incorporating their children with its own.

18.2 Basic operations on B-trees

This section presents the details of the operations B-TREE-SEARCH, B-TREE-CREATE, and B-TREE-INSERT. These procedures observe two conventions:

- The root of the B-tree is always in main memory, so that no procedure ever needs to perform a DISK-READ on the root. If any changes to the root node occur, however, then DISK-WRITE must be called on the root.
- Any nodes that are passed as parameters must already have had a DISK-READ operation performed on them.

The procedures are all "one-pass" algorithms that proceed downward from the root of the tree, without having to back up.

Searching a B-tree

Searching a B-tree is much like searching a binary search tree, except that instead of making a binary, or "two-way," branching decision at each node, the search

makes a multiway branching decision according to the number of the node's children. More precisely, at each internal node x, the search makes an $(x.n + 1)$-way branching decision.

The procedure B-TREE-SEARCH generalizes the TREE-SEARCH procedure defined for binary search trees on page 316. It takes as input a pointer to the root node x of a subtree and a key k to be searched for in that subtree. The top-level call is thus of the form B-TREE-SEARCH$(T.root, k)$. If k is in the B-tree, then B-TREE-SEARCH returns the ordered pair (y, i) consisting of a node y and an index i such that $y.key_i = k$. Otherwise, the procedure returns NIL.

```
B-TREE-SEARCH(x, k)
1  i = 1
2  while i ≤ x.n and k > x.key_i
3       i = i + 1
4  if i ≤ x.n and k == x.key_i
5       return (x, i)
6  elseif x.leaf
7       return NIL
8  else DISK-READ(x.c_i)
9       return B-TREE-SEARCH(x.c_i, k)
```

Using a linear-search procedure, lines 1–3 of B-TREE-SEARCH find the smallest index i such that $k \le x.key_i$, or else they set i to $x.n + 1$. Lines 4–5 check to see whether the search has discovered the key, returning if it has. Otherwise, if x is a leaf, then line 7 terminates the search unsuccessfully, and if x is an internal node, lines 8–9 recurse to search the appropriate subtree of x, after performing the necessary DISK-READ on that child. Figure 18.1 illustrates the operation of B-TREE-SEARCH. The blue nodes are those examined during a search for the key R.

As in the TREE-SEARCH procedure for binary search trees, the nodes encountered during the recursion form a simple path downward from the root of the tree. The B-TREE-SEARCH procedure therefore accesses $O(h) = O(\log_t n)$ disk blocks, where h is the height of the B-tree and n is the number of keys in the B-tree. Since $x.n < 2t$, the **while** loop of lines 2–3 takes $O(t)$ time within each node, and the total CPU time is $O(th) = O(t \log_t n)$.

Creating an empty B-tree

To build a B-tree T, first use the B-TREE-CREATE procedure on the next page to create an empty root node and then call the B-TREE-INSERT procedure on

page 508 to add new keys. Both of these procedures use an auxiliary procedure ALLOCATE-NODE, whose pseudocode we omit and which allocates one disk block to be used as a new node in $O(1)$ time. A node created by ALLOCATE-NODE requires no DISK-READ, since there is as yet no useful information stored on the disk for that node. B-TREE-CREATE requires $O(1)$ disk operations and $O(1)$ CPU time.

```
B-TREE-CREATE(T)
1  x = ALLOCATE-NODE()
2  x.leaf = TRUE
3  x.n = 0
4  DISK-WRITE(x)
5  T.root = x
```

Inserting a key into a B-tree

Inserting a key into a B-tree is significantly more complicated than inserting a key into a binary search tree. As with binary search trees, you search for the leaf position at which to insert the new key. With a B-tree, however, you cannot simply create a new leaf node and insert it, as the resulting tree would fail to be a valid B-tree. Instead, you insert the new key into an existing leaf node. Since you cannot insert a key into a leaf node that is full, you need an operation that ***splits*** a full node y (having $2t - 1$ keys) around its ***median key*** $y.key_t$ into two nodes having only $t - 1$ keys each. The median key moves up into y's parent to identify the dividing point between the two new trees. But if y's parent is also full, you must split it before you can insert the new key, and thus you could end up splitting full nodes all the way up the tree.

To avoid having to go back up the tree, just split every full node you encounter as you go down the tree. In this way, whenever you need to split a full node, you are assured that its parent is not full. Inserting a key into a B-tree then requires only a single pass down the tree from the root to a leaf.

Splitting a node in a B-tree

The procedure B-TREE-SPLIT-CHILD on the facing page takes as input a *nonfull* internal node x (assumed to reside in main memory) and an index i such that $x.c_i$ (also assumed to reside in main memory) is a *full* child of x. The procedure splits this child in two and adjusts x so that it has an additional child. To split a full root, you first need to make the root a child of a new empty root node, so that you can

use B-TREE-SPLIT-CHILD. The tree thus grows in height by 1: splitting is the only means by which the tree grows taller.

```
B-TREE-SPLIT-CHILD(x, i)
 1  y = x.c_i                          // full node to split
 2  z = ALLOCATE-NODE()                // z will take half of y
 3  z.leaf = y.leaf
 4  z.n = t − 1
 5  for j = 1 to t − 1                 // z gets y's greatest keys ...
 6      z.key_j = y.key_{j+t}
 7  if not y.leaf
 8      for j = 1 to t                 // ... and its corresponding children
 9          z.c_j = y.c_{j+t}
10  y.n = t − 1                        // y keeps t − 1 keys
11  for j = x.n + 1 downto i + 1       // shift x's children to the right ...
12      x.c_{j+1} = x.c_j
13  x.c_{i+1} = z                      // ... to make room for z as a child
14  for j = x.n downto i               // shift the corresponding keys in x
15      x.key_{j+1} = x.key_j
16  x.key_i = y.key_t                  // insert y's median key
17  x.n = x.n + 1                      // x has gained a child
18  DISK-WRITE(y)
19  DISK-WRITE(z)
20  DISK-WRITE(x)
```

Figure 18.5 illustrates how a node splits. B-TREE-SPLIT-CHILD splits the full node $y = x.c_i$ about its median key (S in the figure), which moves up into y's parent node x. Those keys in y that are greater than the median key move into a new node z, which becomes a new child of x.

B-TREE-SPLIT-CHILD works by straightforward cutting and pasting. Node x is the parent of the node y being split, which is x's ith child (set in line 1). Node y originally has $2t$ children and $2t - 1$ keys, but splitting reduces y to t children and $t - 1$ keys. The t largest children and $t - 1$ keys of node y move over to node z, which becomes a new child of x, positioned just after y in x's table of children. The median key of y moves up to become the key in node x that separates the pointers to nodes y and z.

Lines 2–9 create node z and give it the largest $t - 1$ keys and, if y and z are internal nodes, the corresponding t children of y. Line 10 adjusts the key count for y. Then, lines 11–17 shift keys and child pointers in x to the right in order to make room for x's new child, insert z as a new child of x, move the median key

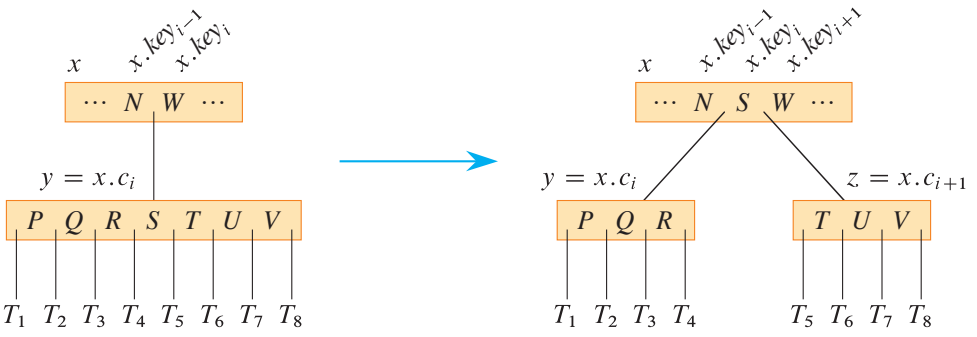


Figure 18.5 Splitting a node with $t = 4$. Node $y = x.c_i$ splits into two nodes, y and z, and the median key S of y moves up into y's parent.

from y up to x in order to separate y from z, and adjust x's key count. Lines 18–20 write out all modified disk blocks. The CPU time used by B-TREE-SPLIT-CHILD is $\Theta(t)$, due to the **for** loops in lines 5–6 and 8–9. (The **for** loops in lines 11–12 and 14–15 also run for $O(t)$ iterations.) The procedure performs $O(1)$ disk operations.

Inserting a key into a B-tree in a single pass down the tree

Inserting a key k into a B-tree T of height h requires just a single pass down the tree and $O(h)$ disk accesses. The CPU time required is $O(th) = O(t \log_t n)$. The B-TREE-INSERT procedure uses B-TREE-SPLIT-CHILD to guarantee that the recursion never descends to a full node. If the root is full, B-TREE-INSERT splits it by calling the procedure B-TREE-SPLIT-ROOT on the facing page.

```
B-TREE-INSERT(T, k)
1  r = T.root
2  if r.n == 2t − 1
3       s = B-TREE-SPLIT-ROOT(T)
4       B-TREE-INSERT-NONFULL(s, k)
5  else B-TREE-INSERT-NONFULL(r, k)
```

B-TREE-INSERT works as follows. If the root is full, then line 3 calls B-TREE-SPLIT-ROOT in line 3 to split it. A new node s (with two children) becomes the root and is returned by B-TREE-SPLIT-ROOT. Splitting the root, illustrated in Figure 18.6, is the only way to increase the height of a B-tree. Unlike a binary search tree, a B-tree increases in height at the top instead of at the bottom. Regardless of whether the root split, B-TREE-INSERT finishes by calling B-TREE-INSERT-NONFULL to insert key k into the tree rooted at the nonfull root node,

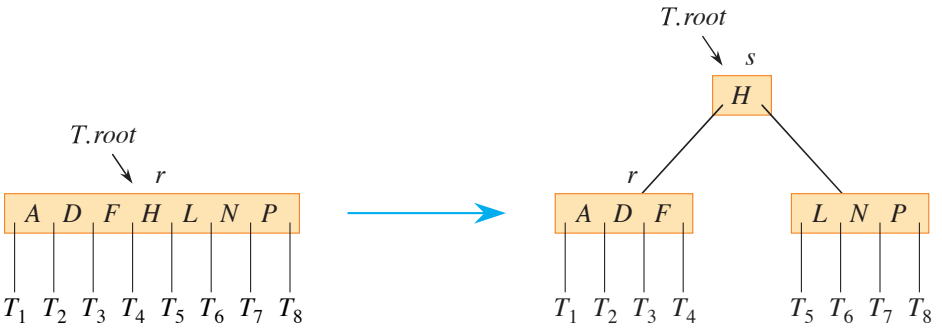


Figure 18.6 Splitting the root with $t = 4$. Root node r splits in two, and a new root node s is created. The new root contains the median key of r and has the two halves of r as children. The B-tree grows in height by one when the root is split. A B-tree's height increases only when the root splits.

which is either the new root (the call in line 4) or the original root (the call in line 5).

```
B-TREE-SPLIT-ROOT(T)
1  s = ALLOCATE-NODE()
2  s.leaf = FALSE
3  s.n = 0
4  s.c_1 = T.root
5  T.root = s
6  B-TREE-SPLIT-CHILD(s, 1)
7  return s
```

The auxiliary procedure B-TREE-INSERT-NONFULL on page 511 inserts key k into node x, which is assumed to be nonfull when the procedure is called. B-TREE-INSERT-NONFULL recurses as necessary down the tree, at all times guaranteeing that the node to which it recurses is not full by calling B-TREE-SPLIT-CHILD as necessary. The operation of B-TREE-INSERT and the recursive operation of B-TREE-INSERT-NONFULL guarantee that this assumption is true.

Figure 18.7 illustrates the various cases of how B-TREE-INSERT-NONFULL inserts a key into a B-tree. Lines 3–8 handle the case in which x is a leaf node by inserting key k into x, shifting to the right all keys in x that are greater than k. If x is not a leaf node, then k should go into the appropriate leaf node in the subtree rooted at internal node x. Lines 9–11 determine the child $x.c_i$ to which the recursion descends. Line 13 detects whether the recursion would descend to a full child, in which case line 14 calls B-TREE-SPLIT-CHILD to split that child into two non-

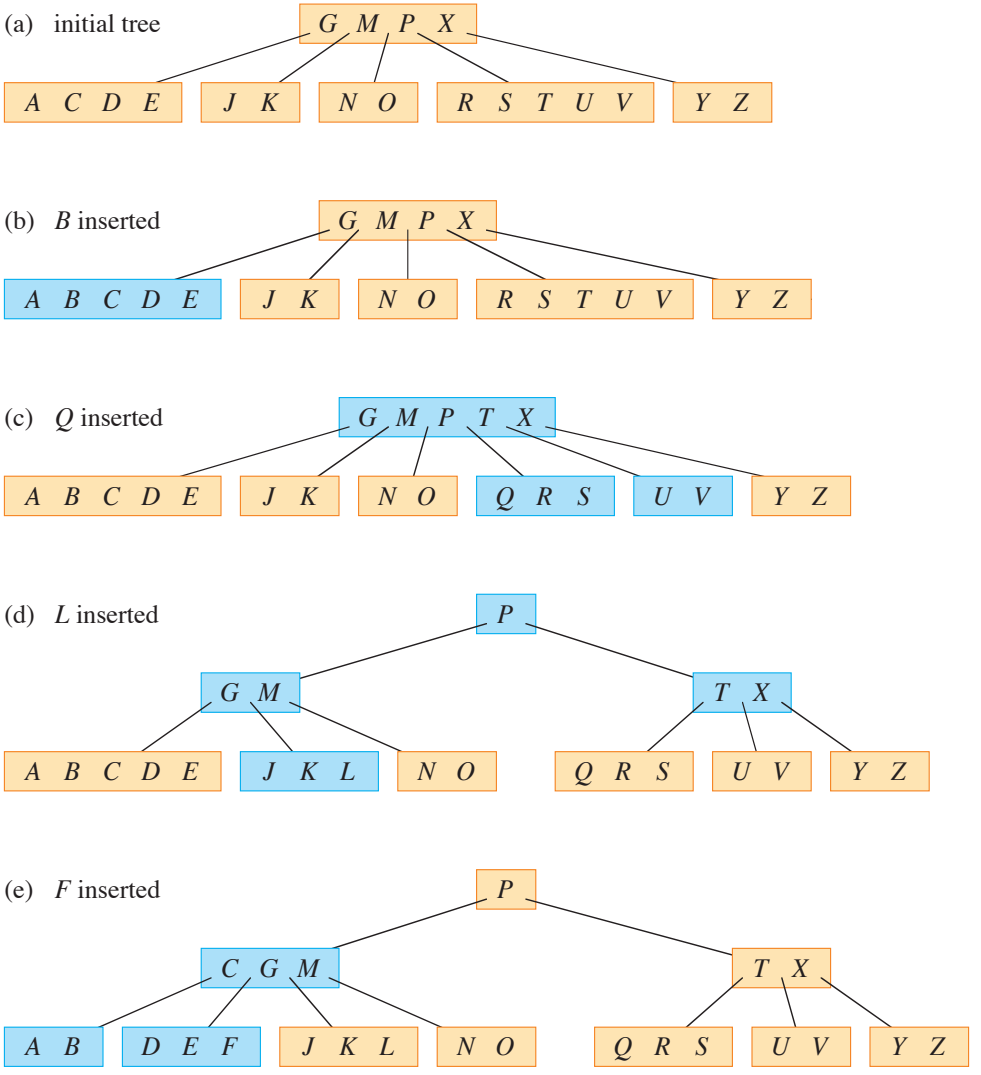


Figure 18.7 Inserting keys into a B-tree. The minimum degree t for this B-tree is 3, so that a node can hold at most 5 keys. Blue nodes are modified by the insertion process. **(a)** The initial tree for this example. **(b)** The result of inserting B into the initial tree. This case is a simple insertion into a leaf node. **(c)** The result of inserting Q into the previous tree. The node $RSTUV$ splits into two nodes containing RS and UV, the key T moves up to the root, and Q is inserted in the leftmost of the two halves (the RS node). **(d)** The result of inserting L into the previous tree. The root splits right away, since it is full, and the B-tree grows in height by one. Then L is inserted into the leaf containing JK. **(e)** The result of inserting F into the previous tree. The node $ABCDE$ splits before F is inserted into the rightmost of the two halves (the DE node).

```
B-TREE-INSERT-NONFULL(x, k)
 1  i = x.n
 2  if x.leaf                                  // inserting into a leaf?
 3      while i ≥ 1 and k < x.key_i            // shift keys in x to make room for k
 4          x.key_{i+1} = x.key_i
 5          i = i − 1
 6      x.key_{i+1} = k                        // insert key k in x
 7      x.n = x.n + 1                          // now x has 1 more key
 8      DISK-WRITE(x)
 9  else while i ≥ 1 and k < x.key_i           // find the child where k belongs
10          i = i − 1
11      i = i + 1
12      DISK-READ(x.c_i)
13      if x.c_i.n == 2t − 1                   // split the child if it's full
14          B-TREE-SPLIT-CHILD(x, i)
15          if k > x.key_i                     // does k go into x.c_i or x.c_{i+1}?
16              i = i + 1
17      B-TREE-INSERT-NONFULL(x.c_i, k)
```

full children, and lines 15–16 determine which of the two children is the correct one to descend to. (Note that DISK-READ$(x.c_i)$ is not needed after line 16 increments i, since the recursion descends in this case to a child that was just created by B-TREE-SPLIT-CHILD.) The net effect of lines 13–16 is thus to guarantee that the procedure never recurses to a full node. Line 17 then recurses to insert k into the appropriate subtree.

For a B-tree of height h, B-TREE-INSERT performs $O(h)$ disk accesses, since only $O(1)$ DISK-READ and DISK-WRITE operations occur at each level of the tree. The total CPU time used is $O(t)$ in each level of the tree, or $O(th) = O(t \log_t n)$ overall. Since B-TREE-INSERT-NONFULL is tail-recursive, you can instead implement it with a **while** loop, thereby demonstrating that the number of blocks that need to be in main memory at any time is $O(1)$.

Exercises

18.2-1

Show the results of inserting the keys

$F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E$

in order into an empty B-tree with minimum degree 2. Draw only the configurations of the tree just before some node must split, and also draw the final configuration.

18.2-2

Explain under what circumstances, if any, redundant DISK-READ or DISK-WRITE operations occur during the course of executing a call to B-TREE-INSERT. (A redundant DISK-READ is a DISK-READ for a block that is already in memory. A redundant DISK-WRITE writes to disk a block of information that is identical to what is already stored there.)

18.2-3

Professor Bunyan asserts that the B-TREE-INSERT procedure always results in a B-tree with the minimum possible height. Show that the professor is mistaken by proving that with $t = 2$ and the set of keys $\{1, 2, \ldots, 15\}$, there is no insertion sequence that results in a B-tree with the minimum possible height.

★ ***18.2-4***

If you insert the keys $\{1, 2, \ldots, n\}$ into an empty B-tree with minimum degree 2, how many nodes does the final B-tree have?

18.2-5

Since leaf nodes require no pointers to children, they could conceivably use a different (larger) t value than internal nodes for the same disk block size. Show how to modify the procedures for creating and inserting into a B-tree to handle this variation.

18.2-6

Suppose that you implement B-TREE-SEARCH to use binary search rather than linear search within each node. Show that this change makes the required CPU time $O(\lg n)$, independent of how t might be chosen as a function of n.

18.2-7

Suppose that disk hardware allows you to choose the size of a disk block arbitrarily, but that the time it takes to read the disk block is $a + bt$, where a and b are specified constants and t is the minimum degree for a B-tree using blocks of the selected size. Describe how to choose t so as to minimize (approximately) the B-tree search time. Suggest an optimal value of t for the case in which $a = 5$ milliseconds and $b = 10$ microseconds.

18.3 Deleting a key from a B-tree

Deletion from a B-tree is analogous to insertion but a little more complicated, because you can delete a key from any node—not just a leaf—and when you delete a key from an internal node, you must rearrange the node's children. As in insertion, you must guard against deletion producing a tree whose structure violates the B-tree properties. Just as a node should not get too big due to insertion, a node must not get too small during deletion (except that the root is allowed to have fewer than the minimum number $t - 1$ of keys). And just as a simple insertion algorithm might have to back up if a node on the path to where the key is to be inserted is full, a simple approach to deletion might have to back up if a node (other than the root) along the path to where the key is to be deleted has the minimum number of keys.

The procedure B-TREE-DELETE deletes the key k from the subtree rooted at x. Unlike the procedures TREE-DELETE on page 325 and RB-DELETE on page 348, which are given the node to delete—presumably as the result of a prior search—B-TREE-DELETE combines the search for key k with the deletion process. Why do we combine search and deletion in B-TREE-DELETE? Just as B-TREE-INSERT prevents any node from becoming overfull (having more than $2t - 1$ keys) while making a single pass down the tree, B-TREE-DELETE prevents any node from becoming underfull (having fewer than $t - 1$ keys) while also making a single pass down the tree, searching for and ultimately deleting the key.

To prevent any node from becoming underfull, the design of B-TREE-DELETE guarantees that whenever it calls itself recursively on a node x, the number of keys in x is at least the minimum degree t at the time of the call. (Although the root may have fewer than t keys and a recursive call may be made *from* the root, no recursive call is made *on* the root.) This condition requires one more key than the minimum required by the usual B-tree conditions, and so a key might have to be moved from x into one of its child nodes (still leaving x with at least the minimum $t - 1$ keys) before a recursive call is made on that child, thus allowing deletion to occur in one downward pass without having to traverse back up the tree.

We describe how the procedure B-TREE-DELETE(T, k) deletes a key k from a B-tree T instead of presenting detailed pseudocode. We examine three cases, illustrated in Figure 18.8. The cases are for when the search arrives at a leaf, at an internal node containing key k, and at an internal node not containing key k. As mentioned above, in all three cases node x has at least t keys (with the possible exception of when x is the root). Cases 2 and 3—when x is an internal node—guarantee this property as the recursion descends through the B-tree.

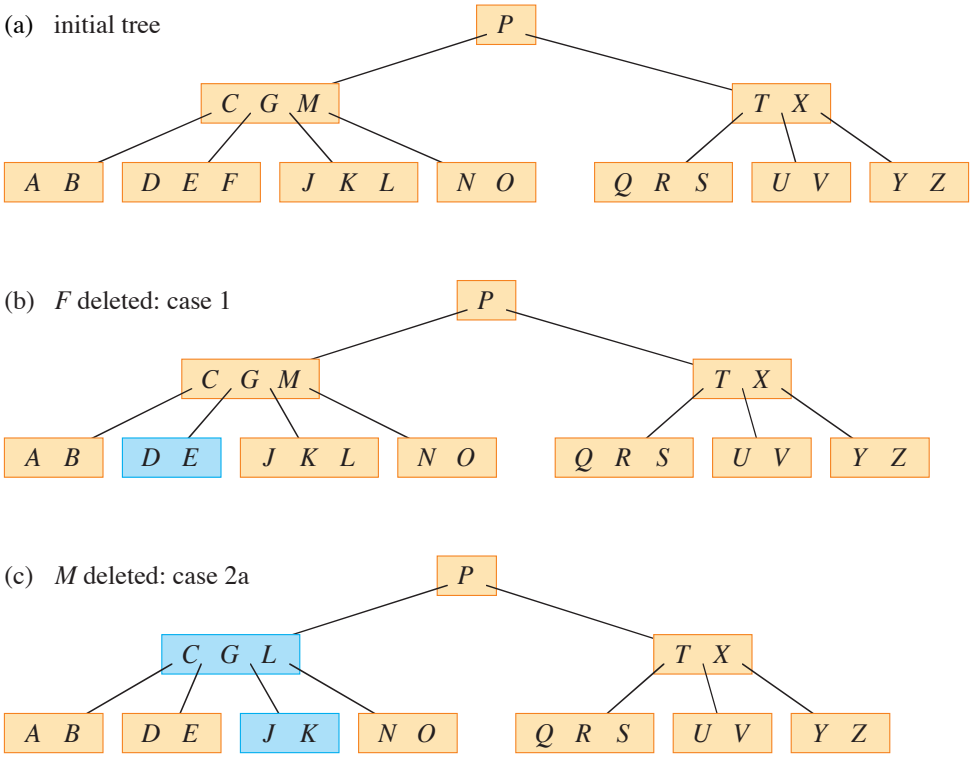


Figure 18.8 Deleting keys from a B-tree. The minimum degree for this B-tree is $t = 3$, so that, other than the root, every node must have at least 2 keys. Blue nodes are those that are modified by the deletion process. **(a)** The B-tree of Figure 18.7(e). **(b)** Deletion of F, which is case 1: simple deletion from a leaf when all nodes visited during the search (other than the root) have at least $t = 3$ keys. **(c)** Deletion of M, which is case 2a: the predecessor L of M moves up to take M's position.

Case 1: *The search arrives at a leaf node x.* If x contains key k, then delete k from x. If x does not contain key k, then k was not in the B-tree and nothing else needs to be done.

Case 2: *The search arrives at an internal node x that contains key k.* Let $k = x.key_i$. One of the following three cases applies, depending on the number of keys in $x.c_i$ (the child of x that precedes k) and $x.c_{i+1}$ (the child of x that follows k).

Case 2a: *$x.c_i$ has at least t keys.* Find the predecessor k' of k in the subtree rooted at $x.c_i$. Recursively delete k' from $x.c_i$, and replace k by k' in x. (Key k' can be found and deleted in a single downward pass.)

Case 2b: *$x.c_i$ has $t - 1$ keys and $x.c_{i+1}$ has at least t keys.* This case is symmetric to case 2a. Find the successor k' of k in the subtree rooted at $x.c_{i+1}$.

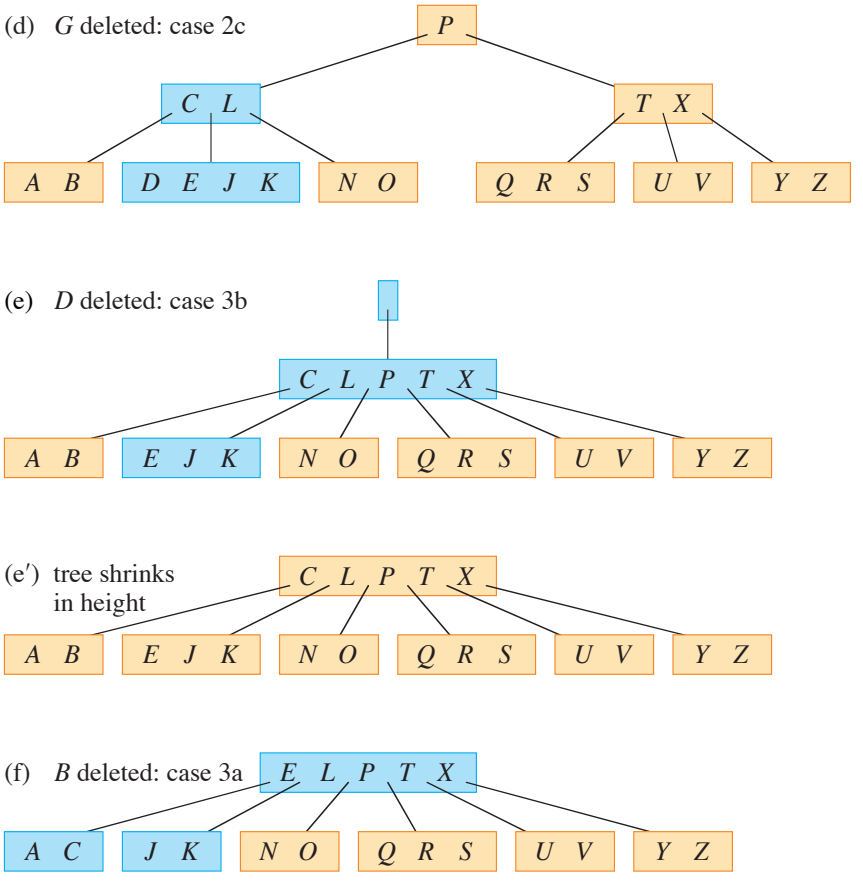


Figure 18.8, continued **(d)** Deletion of G, which is case 2c: push G down to make node $DEGJK$ and then delete G from this leaf (case 1). **(e)** Deletion of D, which is case 3b: since the recursion cannot descend to node CL because it has only 2 keys, push P down and merge it with CL and TX to form $CLPTX$. Then delete D from a leaf (case 1). **(e′)** After (e), delete the empty root. The tree shrinks in height by 1. **(f)** Deletion of B, which is case 3a: C moves to fill B's position and E moves to fill C's position.

Recursively delete k' from $x.c_{i+1}$, and replace k by k' in x. (Again, finding and deleting k' can be done in a single downward pass.)

Case 2c: *Both $x.c_i$ and $x.c_{i+1}$ have $t-1$ keys.* Merge k and all of $x.c_{i+1}$ into $x.c_i$, so that x loses both k and the pointer to $x.c_{i+1}$, and $x.c_i$ now contains $2t-1$ keys. Then free $x.c_{i+1}$ and recursively delete k from $x.c_i$.

Case 3: *The search arrives at an internal node x that does not contain key k.* Continue searching down the tree while ensuring that each node visited has at least t keys. To do so, determine the root $x.c_i$ of the appropriate subtree that must contain k, if k is in the tree at all. If $x.c_i$ has only $t-1$ keys, execute

case 3a or 3b as necessary to guarantee descending to a node containing at least t keys. Then finish by recursing on the appropriate child of x.

Case 3a: *$x.c_i$ has only $t-1$ keys but has an immediate sibling with at least t keys.* Give $x.c_i$ an extra key by moving a key from x down into $x.c_i$, moving a key from $x.c_i$'s immediate left or right sibling up into x, and moving the appropriate child pointer from the sibling into $x.c_i$.

Case 3b: *$x.c_i$ and each of $x.c_i$'s immediate siblings have $t-1$ keys.* (It is possible for $x.c_i$ to have either one or two siblings.) Merge $x.c_i$ with one sibling, which involves moving a key from x down into the new merged node to become the median key for that node.

In cases 2c and 3b, if node x is the root, it could end up having no keys. When this situation occurs, then x is deleted, and x's only child $x.c_1$ becomes the new root of the tree. This action decreases the height of the tree by one and preserves the property that the root of the tree contains at least one key (unless the tree is empty).

Since most of the keys in a B-tree are in the leaves, deletion operations often end up deleting keys from leaves. The B-TREE-DELETE procedure then acts in one downward pass through the tree, without having to back up. When deleting a key in an internal node x, however, the procedure might make a downward pass through the tree to find the key's predecessor or successor and then return to node x to replace the key with its predecessor or successor (cases 2a and 2b). Returning to node x does not require a traversal through all the levels between x and the node containing the predecessor or successor, however, since the procedure can just keep a pointer to x and the key position within x and put the predecessor or successor key directly there.

Although this procedure seems complicated, it involves only $O(h)$ disk operations for a B-tree of height h, since only $O(1)$ calls to DISK-READ and DISK-WRITE are made between recursive invocations of the procedure. The CPU time required is $O(th) = O(t \log_t n)$.

Exercises

18.3-1

Show the results of deleting C, P, and V, in order, from the tree of Figure 18.8(f).

18.3-2

Write pseudocode for B-TREE-DELETE.

Problems

18-1 Stacks on secondary storage

Consider implementing a stack in a computer that has a relatively small amount of fast primary memory and a relatively large amount of slower disk storage. The operations PUSH and POP work on single-word values. The stack can grow to be much larger than can fit in memory, and thus most of it must be stored on disk.

A simple, but inefficient, stack implementation keeps the entire stack on disk. Maintain in memory a stack pointer, which is the disk address of the top element on the stack. Indexing block numbers and word offsets within blocks from 0, if the pointer has value p, the top element is the $(p \bmod m)$th word on block $\lfloor p/m \rfloor$ of the disk, where m is the number of words per block.

To implement the PUSH operation, increment the stack pointer, read the appropriate block into memory from disk, copy the element to be pushed to the appropriate word on the block, and write the block back to disk. A POP operation is similar. Read in the appropriate block from disk, save the top of the stack, decrement the stack pointer, and return the saved value. You need not write back the block, since it was not modified, and the word in the block that contained the popped value is ignored.

As in the analyses of B-tree operations, two costs matter: the total number of disk accesses and the total CPU time. A disk access also incurs a cost in CPU time. In particular, any disk access to a block of m words incurs charges of one disk access and $\Theta(m)$ CPU time.

a. Asymptotically, what is the worst-case number of disk accesses for n stack operations using this simple implementation? What is the CPU time for n stack operations? Express your answer in terms of m and n for this and subsequent parts.

Now consider a stack implementation in which you keep one block of the stack in memory. (You also maintain a small amount of memory to record which block is currently in memory.) You can perform a stack operation only if the relevant disk block resides in memory. If necessary, you can write the block currently in memory to the disk and read the new block from the disk into memory. If the relevant disk block is already in memory, then no disk accesses are required.

b. What is the worst-case number of disk accesses required for n PUSH operations? What is the CPU time?

c. What is the worst-case number of disk accesses required for n stack operations? What is the CPU time?

Suppose that you now implement the stack by keeping two blocks in memory (in addition to a small number of words for bookkeeping).

d. Describe how to manage the stack blocks so that the amortized number of disk accesses for any stack operation is $O(1/m)$ and the amortized CPU time for any stack operation is $O(1)$.

18-2 Joining and splitting 2-3-4 trees

The ***join*** operation takes two dynamic sets S' and S'' and an element x such that $x'.key < x.key < x''.key$ for any $x' \in S'$ and $x'' \in S''$. It returns a set $S = S' \cup \{x\} \cup S''$. The ***split*** operation is like an "inverse" join: given a dynamic set S and an element $x \in S$, it creates a set S' that consists of all elements in $S - \{x\}$ whose keys are less than $x.key$ and another set S'' that consists of all elements in $S - \{x\}$ whose keys are greater than $x.key$. This problem investigates how to implement these operations on 2-3-4 trees (B-trees with $t = 2$). Assume for convenience that elements consist only of keys and that all key values are distinct.

a. Show how to maintain, for every node x of a 2-3-4 tree, the height of the subtree rooted at x as an attribute $x.height$. Make sure that your implementation does not affect the asymptotic running times of searching, insertion, and deletion.

b. Show how to implement the join operation. Given two 2-3-4 trees T' and T'' and a key k, the join operation should run in $O(1 + |h' - h''|)$ time, where h' and h'' are the heights of T' and T'', respectively.

c. Consider the simple path p from the root of a 2-3-4 tree T to a given key k, the set S' of keys in T that are less than k, and the set S'' of keys in T that are greater than k. Show that p breaks S' into a set of trees $\{T'_0, T'_1, \ldots, T'_m\}$ and a set of keys $\{k'_1, k'_2, \ldots, k'_m\}$ such that $y < k'_i < z$ for $i = 1, 2, \ldots, m$ and any keys $y \in T'_{i-1}$ and $z \in T'_i$. What is the relationship between the heights of T'_{i-1} and T'_i? Describe how p breaks S'' into sets of trees and keys.

d. Show how to implement the split operation on T. Use the join operation to assemble the keys in S' into a single 2-3-4 tree T' and the keys in S'' into a single 2-3-4 tree T''. The running time of the split operation should be $O(\lg n)$, where n is the number of keys in T. (*Hint:* The costs for joining should telescope.)

Chapter notes

Knuth [261], Aho, Hopcroft, and Ullman [5], and Sedgewick and Wayne [402] give further discussions of balanced-tree schemes and B-trees. Comer [99] provides a comprehensive survey of B-trees. Guibas and Sedgewick [202] discuss the relationships among various kinds of balanced-tree schemes, including red-black trees and 2-3-4 trees.

In 1970, J. E. Hopcroft invented 2-3 trees, a precursor to B-trees and 2-3-4 trees, in which every internal node has either two or three children. Bayer and McCreight [39] introduced B-trees in 1972 with no explanation of their choice of name.

Bender, Demaine, and Farach-Colton [47] studied how to make B-trees perform well in the presence of memory-hierarchy effects. Their ***cache-oblivious*** algorithms work efficiently without explicitly knowing the data transfer sizes within the memory hierarchy.

19 Data Structures for Disjoint Sets

Some applications involve grouping n distinct elements into a collection of disjoint sets—sets with no elements in common. These applications often need to perform two operations in particular: finding the unique set that contains a given element and uniting two sets. This chapter explores methods for maintaining a data structure that supports these operations.

Section 19.1 describes the operations supported by a disjoint-set data structure and presents a simple application. Section 19.2 looks at a simple linked-list implementation for disjoint sets. Section 19.3 presents a more efficient representation using rooted trees. The running time using the tree representation is theoretically superlinear, but for all practical purposes it is linear. Section 19.4 defines and discusses a very quickly growing function and its very slowly growing inverse, which appears in the running time of operations on the tree-based implementation, and then, by a complex amortized analysis, proves an upper bound on the running time that is just barely superlinear.

19.1 Disjoint-set operations

A ***disjoint-set data structure*** maintains a collection $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$ of disjoint dynamic sets. To identify each set, choose a ***representative***, which is some member of the set. In some applications, it doesn't matter which member is used as the representative; it matters only that if you ask for the representative of a dynamic set twice without modifying the set between the requests, you get the same answer both times. Other applications may require a prespecified rule for choosing the representative, such as choosing the smallest member in the set (for a set whose elements can be ordered).

As in the other dynamic-set implementations we have studied, each element of a set is represented by an object. Letting x denote an object, we'll see how to support the following operations:

MAKE-SET(x), where x does not already belong to some other set, creates a new set whose only member (and thus representative) is x.

UNION(x, y) unites two disjoint, dynamic sets that contain x and y, say S_x and S_y, into a new set that is the union of these two sets. The representative of the resulting set is any member of $S_x \cup S_y$, although many implementations of UNION specifically choose the representative of either S_x or S_y as the new representative. Since the sets in the collection must at all times be disjoint, the UNION operation destroys sets S_x and S_y, removing them from the collection $\mathcal{S}$. In practice, implementations often absorb the elements of one of the sets into the other set.

FIND-SET(x) returns a pointer to the representative of the unique set containing x.

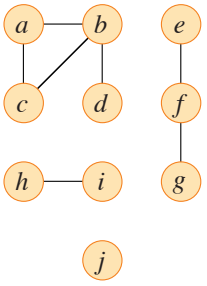
Throughout this chapter, we'll analyze the running times of disjoint-set data structures in terms of two parameters: n, the number of MAKE-SET operations, and m, the total number of MAKE-SET, UNION, and FIND-SET operations. Because the total number of operations m includes the n MAKE-SET operations, $m \geq n$. The first n operations are always MAKE-SET operations, so that after the first n operations, the collection consists of n singleton sets. Since the sets are disjoint at all times, each UNION operation reduces the number of sets by 1. After $n - 1$ UNION operations, therefore, only one set remains, and so at most $n - 1$ UNION operations can occur.

An application of disjoint-set data structures

One of the many applications of disjoint-set data structures arises in determining the connected components of an undirected graph (see Section B.4). Figure 19.1(a), for example, shows a graph with four connected components.

The procedure CONNECTED-COMPONENTS on the following page uses the disjoint-set operations to compute the connected components of a graph. Once the CONNECTED-COMPONENTS procedure has preprocessed the graph, the procedure SAME-COMPONENT answers queries about whether two vertices belong to the same connected component. In pseudocode, we denote the set of vertices of a graph G by $G.V$ and the set of edges by $G.E$.

The procedure CONNECTED-COMPONENTS initially places each vertex v in its own set. Then, for each edge (u, v), it unites the sets containing u and v. By Exercise 19.1-2, after all the edges are processed, two vertices belong to the same connected component if and only if the objects corresponding to the vertices belong



(a)

Edge processed	Collection of disjoint sets									
initial sets	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b,d)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(e,f)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e,f\}$		$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(a,c)	$\{a,c\}$	$\{b,d\}$			$\{e,f\}$		$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(h,i)	$\{a,c\}$	$\{b,d\}$			$\{e,f\}$		$\{g\}$	$\{h,i\}$		$\{j\}$
(a,b)	$\{a,b,c,d\}$				$\{e,f\}$		$\{g\}$	$\{h,i\}$		$\{j\}$
(f,g)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$
(b,c)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$

(b)

Figure 19.1 **(a)** A graph with four connected components: $\{a,b,c,d\}$, $\{e,f,g\}$, $\{h,i\}$, and $\{j\}$. **(b)** The collection of disjoint sets after processing each edge.

```
CONNECTED-COMPONENTS(G)
1  for each vertex v ∈ G.V
2      MAKE-SET(v)
3  for each edge (u, v) ∈ G.E
4      if FIND-SET(u) ≠ FIND-SET(v)
5          UNION(u, v)

SAME-COMPONENT(u, v)
1  if FIND-SET(u) == FIND-SET(v)
2      return TRUE
3  else return FALSE
```

to the same set. Thus CONNECTED-COMPONENTS computes sets in such a way that the procedure SAME-COMPONENT can determine whether two vertices are in the same connected component. Figure 19.1(b) illustrates how CONNECTED-COMPONENTS computes the disjoint sets.

In an actual implementation of this connected-components algorithm, the representations of the graph and the disjoint-set data structure would need to reference each other. That is, an object representing a vertex would contain a pointer to the corresponding disjoint-set object, and vice versa. Since these programming details depend on the implementation language, we do not address them further here.

When the edges of the graph are static—not changing over time—depth-first search can compute the connected components faster (see Exercise 20.3-12 on

page 572). Sometimes, however, the edges are added dynamically, with the connected components updated as each edge is added. In this case, the implementation given here can be more efficient than running a new depth-first search for each new edge.

Exercises

19.1-1

The CONNECTED-COMPONENTS procedure is run on the undirected graph $G = (V, E)$, where $V = \{a, b, c, d, e, f, g, h, i, j, k\}$, and the edges of E are processed in the order $(d, i), (f, k), (g, i), (b, g), (a, h), (i, j), (d, k), (b, j), (d, f), (g, j), (a, e)$. List the vertices in each connected component after each iteration of lines 3–5.

19.1-2

Show that after all edges are processed by CONNECTED-COMPONENTS, two vertices belong to the same connected component if and only if they belong to the same set.

19.1-3

During the execution of CONNECTED-COMPONENTS on an undirected graph $G = (V, E)$ with k connected components, how many times is FIND-SET called? How many times is UNION called? Express your answers in terms of $|V|$, $|E|$, and k.

19.2 Linked-list representation of disjoint sets

Figure 19.2(a) shows a simple way to implement a disjoint-set data structure: each set is represented by its own linked list. The object for each set has attributes *head*, pointing to the first object in the list, and *tail*, pointing to the last object. Each object in the list contains a set member, a pointer to the next object in the list, and a pointer back to the set object. Within each linked list, the objects may appear in any order. The representative is the set member in the first object in the list.

With this linked-list representation, both MAKE-SET and FIND-SET require only $O(1)$ time. To carry out MAKE-SET(x), create a new linked list whose only object is x. For FIND-SET(x), just follow the pointer from x back to its set object and then return the member in the object that *head* points to. For example, in Figure 19.2(a), the call FIND-SET(g) returns f.

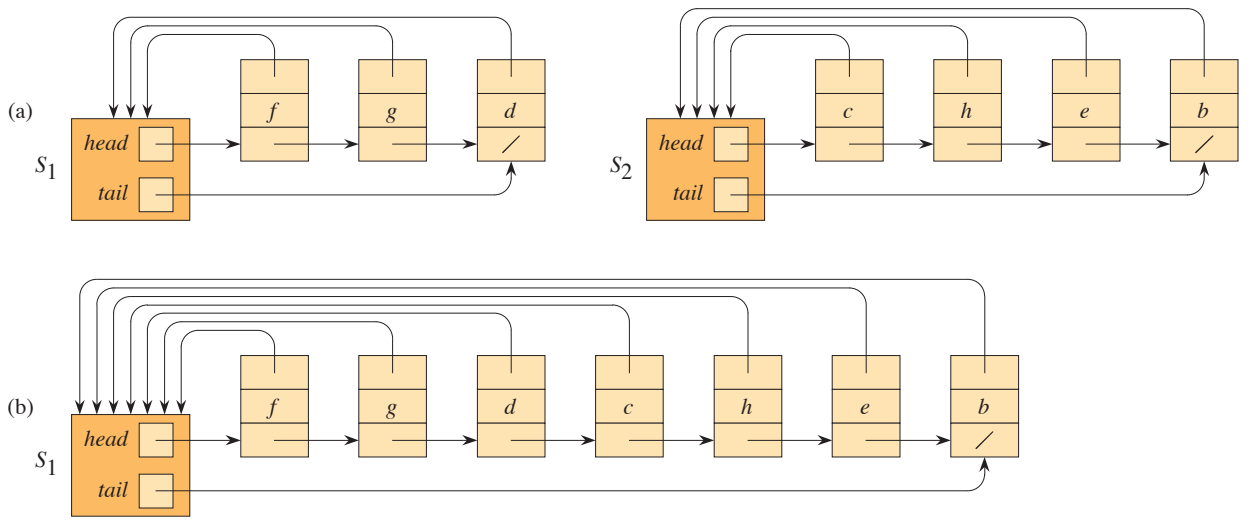


Figure 19.2 **(a)** Linked-list representations of two sets. Set S_1 contains members d, f, and g, with representative f, and set S_2 contains members b, c, e, and h, with representative c. Each object in the list contains a set member, a pointer to the next object in the list, and a pointer back to the set object. Each set object has pointers *head* and *tail* to the first and last objects, respectively. **(b)** The result of UNION(g, e), which appends the linked list containing e to the linked list containing g. The representative of the resulting set is f. The set object for e's list, S_2, is destroyed.

A simple implementation of union

The simplest implementation of the UNION operation using the linked-list set representation takes significantly more time than MAKE-SET or FIND-SET. As Figure 19.2(b) shows, the operation UNION(x, y) appends y's list onto the end of x's list. The representative of x's list becomes the representative of the resulting set. To quickly find where to append y's list, use the *tail* pointer for x's list. Because all members of y's list join x's list, the UNION operation destroys the set object for y's list. The UNION operation is where this implementation pays the price for FIND-SET taking constant time: UNION must also update the pointer to the set object for each object originally on y's list, which takes time linear in the length of y's list. In Figure 19.2, for example, the operation UNION(g, e) causes pointers to be updated in the objects for b, c, e, and h.

In fact, we can construct a sequence of m operations on n objects that requires $\Theta(n^2)$ time. Starting with objects $x_1, x_2, \ldots, x_n$, execute the sequence of n MAKE-SET operations followed by $n-1$ UNION operations shown in Figure 19.3, so that $m = 2n-1$. The n MAKE-SET operations take $\Theta(n)$ time. Because the ith UNION operation updates i objects, the total number of objects updated by all $n-1$ UNION operations forms an arithmetic series:

Operation	Number of objects updated
MAKE-SET(x_1)	1
MAKE-SET(x_2)	1
⋮	⋮
MAKE-SET(x_n)	1
UNION(x_2, x_1)	1
UNION(x_3, x_2)	2
UNION(x_4, x_3)	3
⋮	⋮
UNION(x_n, x_{n-1})	$n-1$

Figure 19.3 A sequence of $2n-1$ operations on n objects that takes $\Theta(n^2)$ time, or $\Theta(n)$ time per operation on average, using the linked-list set representation and the simple implementation of UNION.

$$\sum_{i=1}^{n-1} i = \Theta(n^2) .$$

The total number of operations is $2n-1$, and so each operation on average requires $\Theta(n)$ time. That is, the amortized time of an operation is $\Theta(n)$.

A weighted-union heuristic

In the worst case, the above implementation of UNION requires an average of $\Theta(n)$ time per call, because it might be appending a longer list onto a shorter list, and the procedure must update the pointer to the set object for each member of the longer list. Suppose instead that each list also includes the length of the list (which can be maintained straightforwardly with constant overhead) and that the UNION procedure always appends the shorter list onto the longer, breaking ties arbitrarily. With this simple ***weighted-union heuristic***, a single UNION operation can still take $\Omega(n)$ time if both sets have $\Omega(n)$ members. As the following theorem shows, however, a sequence of m MAKE-SET, UNION, and FIND-SET operations, n of which are MAKE-SET operations, takes $O(m + n \lg n)$ time.

Theorem 19.1
Using the linked-list representation of disjoint sets and the weighted-union heuristic, a sequence of m MAKE-SET, UNION, and FIND-SET operations, n of which are MAKE-SET operations, takes $O(m + n \lg n)$ time.

Proof Because each UNION operation unites two disjoint sets, at most $n-1$ UNION operations occur over all. We now bound the total time taken by these

UNION operations. We start by determining, for each object, an upper bound on the number of times the object's pointer back to its set object is updated. Consider a particular object x. Each time x's pointer is updated, x must have started in the smaller set. The first time x's pointer is updated, therefore, the resulting set must have at least 2 members. Similarly, the next time x's pointer is updated, the resulting set must have had at least 4 members. Continuing on, for any $k \leq n$, after x's pointer has been updated $\lceil \lg k \rceil$ times, the resulting set must have at least k members. Since the largest set has at most n members, each object's pointer is updated at most $\lceil \lg n \rceil$ times over all the UNION operations. Thus the total time spent updating object pointers over all UNION operations is $O(n \lg n)$. We must also account for updating the *tail* pointers and the list lengths, which take only $\Theta(1)$ time per UNION operation. The total time spent in all UNION operations is thus $O(n \lg n)$.

The time for the entire sequence of m operations follows. Each MAKE-SET and FIND-SET operation takes $O(1)$ time, and there are $O(m)$ of them. The total time for the entire sequence is thus $O(m + n \lg n)$. ■

Exercises

19.2-1

Write pseudocode for MAKE-SET, FIND-SET, and UNION using the linked-list representation and the weighted-union heuristic. Make sure to specify the attributes that you assume for set objects and list objects.

19.2-2

Show the data structure that results and the answers returned by the FIND-SET operations in the following program. Use the linked-list representation with the weighted-union heuristic. Assume that if the sets containing x_i and x_j have the same size, then the operation UNION(x_i, x_j) appends x_j's list onto x_i's list.

```
1  for i = 1 to 16
2      MAKE-SET(x_i)
3  for i = 1 to 15 by 2
4      UNION(x_i, x_{i+1})
5  for i = 1 to 13 by 4
6      UNION(x_i, x_{i+2})
7  UNION(x_1, x_5)
8  UNION(x_11, x_13)
9  UNION(x_1, x_10)
10 FIND-SET(x_2)
11 FIND-SET(x_9)
```

19.2-3
Adapt the aggregate proof of Theorem 19.1 to obtain amortized time bounds of $O(1)$ for MAKE-SET and FIND-SET and $O(\lg n)$ for UNION using the linked-list representation and the weighted-union heuristic.

19.2-4
Give a tight asymptotic bound on the running time of the sequence of operations in Figure 19.3 assuming the linked-list representation and the weighted-union heuristic.

19.2-5
Professor Gompers suspects that it might be possible to keep just one pointer in each set object, rather than two (*head* and *tail*), while keeping the number of pointers in each list element at two. Show that the professor's suspicion is well founded by describing how to represent each set by a linked list such that each operation has the same running time as the operations described in this section. Describe also how the operations work. Your scheme should allow for the weighted-union heuristic, with the same effect as described in this section. (*Hint:* Use the tail of a linked list as its set's representative.)

19.2-6
Suggest a simple change to the UNION procedure for the linked-list representation that removes the need to keep the *tail* pointer to the last object in each list. Regardless of whether the weighted-union heuristic is used, your change should not change the asymptotic running time of the UNION procedure. (*Hint:* Rather than appending one list to another, splice them together.)

19.3 Disjoint-set forests

A faster implementation of disjoint sets represents sets by rooted trees, with each node containing one member and each tree representing one set. In a ***disjoint-set forest***, illustrated in Figure 19.4(a), each member points only to its parent. The root of each tree contains the representative and is its own parent. As we'll see, although the straightforward algorithms that use this representation are no faster than ones that use the linked-list representation, two heuristics—"union by rank" and "path compression"—yield an asymptotically optimal disjoint-set data structure.

The three disjoint-set operations have simple implementations. A MAKE-SET operation simply creates a tree with just one node. A FIND-SET operation follows parent pointers until it reaches the root of the tree. The nodes visited on this sim-

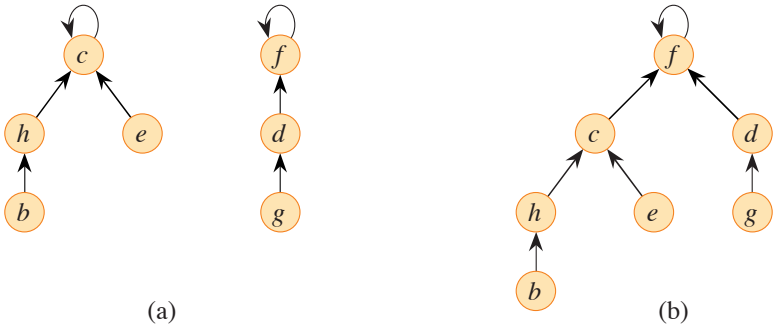


Figure 19.4 A disjoint-set forest. **(a)** Trees representing the two sets of Figure 19.2. The tree on the left represents the set $\{b, c, e, h\}$, with c as the representative, and the tree on the right represents the set $\{d, f, g\}$, with f as the representative. **(b)** The result of UNION(e, g).

ple path toward the root constitute the ***find path***. A UNION operation, shown in Figure 19.4(b), simply causes the root of one tree to point to the root of the other.

Heuristics to improve the running time

So far, disjoint-set forests have not improved on the linked-list implementation. A sequence of $n - 1$ UNION operations could create a tree that is just a linear chain of n nodes. By using two heuristics, however, we can achieve a running time that is almost linear in the total number m of operations.

The first heuristic, ***union by rank***, is similar to the weighted-union heuristic we used with the linked-list representation. The common-sense approach is to make the root of the tree with fewer nodes point to the root of the tree with more nodes. Rather than explicitly keeping track of the size of the subtree rooted at each node, however, we'll adopt an approach that eases the analysis. For each node, maintain a ***rank***, which is an upper bound on the height of the node. Union by rank makes the root with smaller rank point to the root with larger rank during a UNION operation.

The second heuristic, ***path compression***, is also quite simple and highly effective. As shown in Figure 19.5, FIND-SET operations use it to make each node on the find path point directly to the root. Path compression does not change any ranks.

Pseudocode for disjoint-set forests

The union-by-rank heuristic requires its implementation to keep track of ranks. With each node x, maintain the integer value $x.rank$, which is an upper bound on the height of x (the number of edges in the longest simple path from a descendant leaf to x). When MAKE-SET creates a singleton set, the single node in the

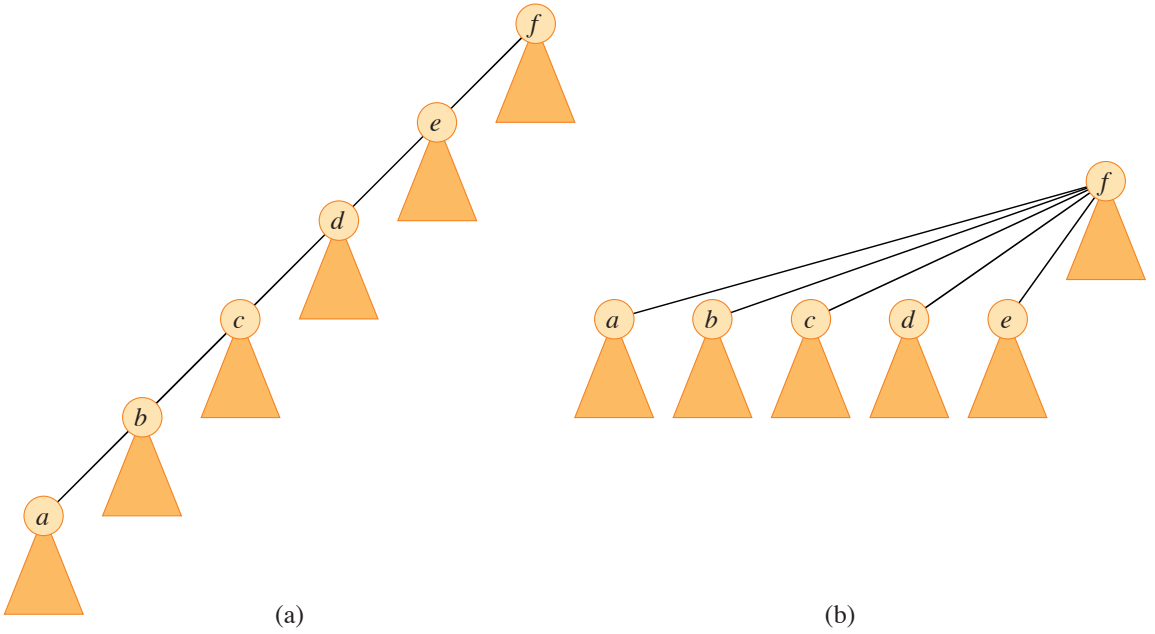


Figure 19.5 Path compression during the operation FIND-SET. Arrows and self-loops at roots are omitted. **(a)** A tree representing a set prior to executing FIND-SET(a). Triangles represent subtrees whose roots are the nodes shown. Each node has a pointer to its parent. **(b)** The same set after executing FIND-SET(a). Each node on the find path now points directly to the root.

corresponding tree has an initial rank of 0. Each FIND-SET operation leaves all ranks unchanged. The UNION operation has two cases, depending on whether the roots of the trees have equal rank. If the roots have unequal ranks, make the root with higher rank the parent of the root with lower rank, but don't change the ranks themselves. If the roots have equal ranks, arbitrarily choose one of the roots as the parent and increment its rank.

Let's put this method into pseudocode, appearing on the next page. The parent of node x is denoted by $x.p$. The LINK procedure, a subroutine called by UNION, takes pointers to two roots as inputs. The FIND-SET procedure with path compression, implemented recursively, turns out to be quite simple.

The FIND-SET procedure is a ***two-pass method***: as it recurses, it makes one pass up the find path to find the root, and as the recursion unwinds, it makes a second pass back down the find path to update each node to point directly to the root. Each call of FIND-SET(x) returns $x.p$ in line 3. If x is the root, then FIND-SET skips line 2 and just returns $x.p$, which is x. In this case the recursion bottoms out. Otherwise, line 2 executes, and the recursive call with parameter $x.p$ returns

```
MAKE-SET(x)
1  x.p = x
2  x.rank = 0

UNION(x, y)
1  LINK(FIND-SET(x), FIND-SET(y))

LINK(x, y)
1  if x.rank > y.rank
2       y.p = x
3  else x.p = y
4       if x.rank == y.rank
5            y.rank = y.rank + 1

FIND-SET(x)
1  if x ≠ x.p                      // not the root?
2       x.p = FIND-SET(x.p)        // the root becomes the parent
3  return x.p                      // return the root
```

a pointer to the root. Line 2 updates node x to point directly to the root, and line 3 returns this pointer.

Effect of the heuristics on the running time

Separately, either union by rank or path compression improves the running time of the operations on disjoint-set forests, and combining the two heuristics yields an even greater improvement. Alone, union by rank yields a running time of $O(m \lg n)$ for a sequence of m operations, n of which are MAKE-SET (see Exercise 19.4-4), and this bound is tight (see Exercise 19.3-3). Although we won't prove it here, for a sequence of n MAKE-SET operations (and hence at most $n - 1$ UNION operations) and f FIND-SET operations, the worst-case running time using only the path-compression heuristic is $\Theta(n + f \cdot (1 + \log_{2+f/n} n))$.

Combining union by rank and path compression gives a worst-case running time of $O(m\alpha(n))$, where $\alpha(n)$ is a *very* slowly growing function, defined in Section 19.4. In any conceivable application of a disjoint-set data structure, $\alpha(n) \leq 4$, and thus, its running time is as good as linear in m for all practical purposes. Mathematically speaking, however, it is superlinear. Section 19.4 proves this $O(m\alpha(n))$ upper bound.

Exercises

19.3-1
Redo Exercise 19.2-2 using a disjoint-set forest with union by rank and path compression. Show the resulting forest with each node including its x_i and rank.

19.3-2
Write a nonrecursive version of FIND-SET with path compression.

19.3-3
Give a sequence of m MAKE-SET, UNION, and FIND-SET operations, n of which are MAKE-SET operations, that takes $\Omega(m \lg n)$ time when using only union by rank and not path compression.

19.3-4
Consider the operation PRINT-SET(x), which is given a node x and prints all the members of x's set, in any order. Show how to add just a single attribute to each node in a disjoint-set forest so that PRINT-SET(x) takes time linear in the number of members of x's set and the asymptotic running times of the other operations are unchanged. Assume that you can print each member of the set in $O(1)$ time.

★ ***19.3-5***
Show that any sequence of m MAKE-SET, FIND-SET, and LINK operations, where all the LINK operations appear before any of the FIND-SET operations, takes only $O(m)$ time when using both path compression and union by rank. You may assume that the arguments to LINK are roots within the disjoint-set forest. What happens in the same situation when using only path compression and not union by rank?

★ 19.4 Analysis of union by rank with path compression

As noted in Section 19.3, the combined union-by-rank and path-compression heuristic runs in $O(m\,\alpha(n))$ time for m disjoint-set operations on n elements. In this section, we'll explore the function α to see just how slowly it grows. Then we'll analyze the running time using the potential method of amortized analysis.

A very quickly growing function and its very slowly growing inverse

For integers $j, k \geq 0$, we define the function $A_k(j)$ as

$$A_k(j) = \begin{cases} j+1 & \text{if } k = 0\,, \\ A_{k-1}^{(j+1)}(j) & \text{if } k \geq 1\,, \end{cases} \tag{19.1}$$

where the expression $A_{k-1}^{(j+1)}(j)$ uses the functional-iteration notation defined in equation (3.30) on page 68. Specifically, equation (3.30) gives $A_{k-1}^{(0)}(j) = j$ and $A_{k-1}^{(i)}(j) = A_{k-1}(A_{k-1}^{(i-1)}(j))$ for $i \geq 1$. We call the parameter k the ***level*** of the function A.

The function $A_k(j)$ strictly increases with both j and k. To see just how quickly this function grows, we first obtain closed-form expressions for $A_1(j)$ and $A_2(j)$.

Lemma 19.2
For any integer $j \geq 1$, we have $A_1(j) = 2j + 1$.

Proof We first use induction on i to show that $A_0^{(i)}(j) = j + i$. For the base case, $A_0^{(0)}(j) = j = j + 0$. For the inductive step, assume that $A_0^{(i-1)}(j) = j + (i-1)$. Then $A_0^{(i)}(j) = A_0(A_0^{(i-1)}(j)) = (j + (i-1)) + 1 = j + i$. Finally, we note that $A_1(j) = A_0^{(j+1)}(j) = j + (j+1) = 2j + 1$. ■

Lemma 19.3
For any integer $j \geq 1$, we have $A_2(j) = 2^{j+1}(j+1) - 1$.

Proof We first use induction on i to show that $A_1^{(i)}(j) = 2^i(j+1) - 1$. For the base case, we have $A_1^{(0)}(j) = j = 2^0(j+1) - 1$. For the inductive step, assume that $A_1^{(i-1)}(j) = 2^{i-1}(j+1) - 1$. Then $A_1^{(i)}(j) = A_1(A_1^{(i-1)}(j)) = A_1(2^{i-1}(j+1) - 1) = 2 \cdot (2^{i-1}(j+1) - 1) + 1 = 2^i(j+1) - 2 + 1 = 2^i(j+1) - 1$. Finally, we note that $A_2(j) = A_1^{(j+1)}(j) = 2^{j+1}(j+1) - 1$. ■

Now we can see how quickly $A_k(j)$ grows by simply examining $A_k(1)$ for levels $k = 0, 1, 2, 3, 4$. From the definition of $A_0(j)$ and the above lemmas, we have $A_0(1) = 1 + 1 = 2$, $A_1(1) = 2 \cdot 1 + 1 = 3$, and $A_2(1) = 2^{1+1} \cdot (1+1) - 1 = 7$. We also have

$$\begin{aligned} A_3(1) &= A_2^{(2)}(1) \\ &= A_2(A_2(1)) \\ &= A_2(7) \\ &= 2^8 \cdot 8 - 1 \\ &= 2^{11} - 1 \\ &= 2047 \end{aligned}$$

and

$$\begin{aligned}
A_4(1) &= A_3^{(2)}(1) \\
&= A_3(A_3(1)) \\
&= A_3(2047) \\
&= A_2^{(2048)}(2047) \\
&\gg A_2(2047) \\
&= 2^{2048} \cdot 2048 - 1 \\
&= 2^{2059} - 1 \\
&> 2^{2056} \\
&= (2^4)^{514} \\
&= 16^{514} \\
&\gg 10^{80} ,
\end{aligned}$$

which is the estimated number of atoms in the observable universe. (The symbol "$\gg$" denotes the "much-greater-than" relation.)

We define the inverse of the function $A_k(n)$, for integer $n \geq 0$, by

$$\alpha(n) = \min \{k : A_k(1) \geq n\} . \tag{19.2}$$

In words, $\alpha(n)$ is the lowest level k for which $A_k(1)$ is at least n. From the above values of $A_k(1)$, we see that

$$\alpha(n) = \begin{cases} 0 & \text{for } 0 \leq n \leq 2 , \\ 1 & \text{for } n = 3 , \\ 2 & \text{for } 4 \leq n \leq 7 , \\ 3 & \text{for } 8 \leq n \leq 2047 , \\ 4 & \text{for } 2048 \leq n \leq A_4(1) . \end{cases}$$

It is only for values of n so large that the term "astronomical" understates them (greater than $A_4(1)$, a huge number) that $\alpha(n) > 4$, and so $\alpha(n) \leq 4$ for all practical purposes.

Properties of ranks

In the remainder of this section, we prove an $O(m\,\alpha(n))$ bound on the running time of the disjoint-set operations with union by rank and path compression. In order to prove this bound, we first prove some simple properties of ranks.

Lemma 19.4

For all nodes x, we have $x.rank \leq x.p.rank$, with strict inequality if $x \neq x.p$ (x is not a root). The value of $x.rank$ is initially 0, increases through time until $x \neq x.p$,

and from then on, $x.rank$ does not change. The value of $x.p.rank$ monotonically increases over time.

Proof The proof is a straightforward induction on the number of operations, using the implementations of MAKE-SET, UNION, and FIND-SET that appear on page 530, and is left as Exercise 19.4-1. ■

Corollary 19.5
On the simple path from any node going up toward a root, node ranks strictly increase. ■

Lemma 19.6
Every node has rank at most $n - 1$.

Proof Each node's rank starts at 0, and it increases only upon LINK operations. Because there are at most $n - 1$ UNION operations, there are also at most $n - 1$ LINK operations. Because each LINK operation either leaves all ranks alone or increases some node's rank by 1, all ranks are at most $n - 1$. ■

Lemma 19.6 provides a weak bound on ranks. In fact, every node has rank at most $\lfloor \lg n \rfloor$ (see Exercise 19.4-2). The looser bound of Lemma 19.6 suffices for our purposes, however.

Proving the time bound

In order to prove the $O(m\,\alpha(n))$ time bound, we'll use the potential method of amortized analysis from Section 16.3. In performing the amortized analysis, it will be convenient to assume that we invoke the LINK operation rather than the UNION operation. That is, since the parameters of the LINK procedure are pointers to two roots, we act as though we perform the appropriate FIND-SET operations separately. The following lemma shows that even if we count the extra FIND-SET operations induced by UNION calls, the asymptotic running time remains unchanged.

Lemma 19.7
Suppose that we convert a sequence S' of m' MAKE-SET, UNION, and FIND-SET operations into a sequence S of m MAKE-SET, LINK, and FIND-SET operations by turning each UNION into two FIND-SET operations followed by one LINK. Then, if sequence S runs in $O(m\,\alpha(n))$ time, sequence S' runs in $O(m'\,\alpha(n))$ time.

Proof Since each UNION operation in sequence S' is converted into three operations in S, we have $m' \leq m \leq 3m'$, so that $m = \Theta(m')$, Thus, an $O(m\,\alpha(n))$

time bound for the converted sequence S implies an $O(m' \alpha(n))$ time bound for the original sequence S'. ■

From now on, we assume that the initial sequence of m' MAKE-SET, UNION, and FIND-SET operations has been converted to a sequence of m MAKE-SET, LINK, and FIND-SET operations. We now prove an $O(m \alpha(n))$ time bound for the converted sequence and appeal to Lemma 19.7 to prove the $O(m' \alpha(n))$ running time of the original sequence of m' operations.

Potential function

The potential function we use assigns a potential $\phi_q(x)$ to each node x in the disjoint-set forest after q operations. For the potential Φ_q of the entire forest after q operations, sum the individual node potentials: $\Phi_q = \sum_x \phi_q(x)$. Because the forest is empty before the first operation, the sum is taken over an empty set, and so $\Phi_0 = 0$. No potential Φ_q is ever negative.

The value of $\phi_q(x)$ depends on whether x is a tree root after the qth operation. If it is, or if $x.rank = 0$, then $\phi_q(x) = \alpha(n) \cdot x.rank$.

Now suppose that after the qth operation, x is not a root and that $x.rank \geq 1$. We need to define two auxiliary functions on x before we can define $\phi_q(x)$. First we define

$$\text{level}(x) = \max \{k : x.p.rank \geq A_k(x.rank)\} \ . \tag{19.3}$$

That is, level(x) is the greatest level k for which A_k, applied to x's rank, is no greater than x's parent's rank.

We claim that

$$0 \leq \text{level}(x) < \alpha(n) \ , \tag{19.4}$$

which we see as follows. We have

$$\begin{aligned} x.p.rank &\geq x.rank + 1 && \text{(by Lemma 19.4 because } x \text{ is not a root)} \\ &= A_0(x.rank) && \text{(by the definition (19.1) of } A_0(j)) \ , \end{aligned}$$

which implies that level$(x) \geq 0$, and

$$\begin{aligned} A_{\alpha(n)}(x.rank) &\geq A_{\alpha(n)}(1) && \text{(because } A_k(j) \text{ is strictly increasing)} \\ &\geq n && \text{(by the definition (19.2) of } \alpha(n)) \\ &> x.\ p.rank && \text{(by Lemma 19.6)} \ , \end{aligned}$$

which implies that level$(x) < \alpha(n)$.

For a given nonroot node x, the value of level(x) monotonically increases over time. Why? Because x is not a root, its rank does not change. The rank of $x.p$

monotonically increases over time, since if $x.p$ is not a root then its rank does not change, and if $x.p$ is a root then its rank can never decrease. Thus, the difference between $x.rank$ and $x.p.rank$ monotonically increases over time. Therefore, the value of k needed for $A_k(x.rank)$ to overtake $x.p.rank$ monotonically increases over time as well.

The second auxiliary function applies when $x.rank \geq 1$:

$$\text{iter}(x) = \max\left\{i : x.p.rank \geq A^{(i)}_{\text{level}(x)}(x.rank)\right\} . \tag{19.5}$$

That is, $\text{iter}(x)$ is the largest number of times we can iteratively apply $A_{\text{level}(x)}$, applied initially to x's rank, before exceeding x's parent's rank.

We claim that when $x.rank \geq 1$, we have

$$1 \leq \text{iter}(x) \leq x.rank , \tag{19.6}$$

which we see as follows. We have

$$\begin{aligned} x.p.rank &\geq A_{\text{level}(x)}(x.rank) && \text{(by the definition (19.3) of level}(x)\text{)} \\ &= A^{(1)}_{\text{level}(x)}(x.rank) && \text{(by the definition (3.30) of functional iteration)} , \end{aligned}$$

which implies that $\text{iter}(x) \geq 1$. We also have

$$\begin{aligned} A^{(x.rank+1)}_{\text{level}(x)}(x.rank) &= A_{\text{level}(x)+1}(x.rank) && \text{(by the definition (19.1) of } A_k(j)\text{)} \\ &> x.p.rank && \text{(by the definition (19.3) of level}(x)\text{)} , \end{aligned}$$

which implies that $\text{iter}(x) \leq x.rank$. Note that because $x.p.rank$ monotonically increases over time, in order for $\text{iter}(x)$ to decrease, $\text{level}(x)$ must increase. As long as $\text{level}(x)$ remains unchanged, $\text{iter}(x)$ must either increase or remain unchanged.

With these auxiliary functions in place, we are ready to define the potential of node x after q operations:

$$\phi_q(x) = \begin{cases} \alpha(n) \cdot x.rank & \text{if } x \text{ is a root or } x.rank = 0 , \\ (\alpha(n) - \text{level}(x)) \cdot x.rank - \text{iter}(x) & \\ & \text{if } x \text{ is not a root and } x.rank \geq 1 . \end{cases} \tag{19.7}$$

We next investigate some useful properties of node potentials.

Lemma 19.8

For every node x, and for all operation counts q, we have

$$0 \leq \phi_q(x) \leq \alpha(n) \cdot x.rank .$$

Proof If x is a root or $x.rank = 0$, then $\phi_q(x) = \alpha(n) \cdot x.rank$ by definition. Now suppose that x is not a root and that $x.rank \geq 1$. We can obtain a lower bound on $\phi_q(x)$ by maximizing $\text{level}(x)$ and $\text{iter}(x)$. The bounds (19.4) and (19.6) give $\alpha(n) - \text{level}(x) \geq 1$ and $\text{iter}(x) \leq x.rank$. Thus, we have

$$\begin{aligned}\phi_q(x) &= (\alpha(n) - \text{level}(x)) \cdot x.rank - \text{iter}(x)\\ &\geq x.rank - x.rank\\ &= 0\,.\end{aligned}$$

Similarly, minimizing level(x) and iter(x) provides an upper bound on $\phi_q(x)$. By the bound (19.4), level$(x) \geq 0$, and by the bound (19.6), iter$(x) \geq 1$. Thus, we have

$$\begin{aligned}\phi_q(x) &\leq (\alpha(n) - 0) \cdot x.rank - 1\\ &= \alpha(n) \cdot x.rank - 1\\ &< \alpha(n) \cdot x.rank\,.\end{aligned}$$

■

Corollary 19.9
If node x is not a root and $x.rank > 0$, then $\phi_q(x) < \alpha(n) \cdot x.rank$. ■

Potential changes and amortized costs of operations

We are now ready to examine how the disjoint-set operations affect node potentials. Once we understand how each operation can change the potential, we can determine the amortized costs.

Lemma 19.10
Let x be a node that is not a root, and suppose that the qth operation is either a LINK or a FIND-SET. Then after the qth operation, $\phi_q(x) \leq \phi_{q-1}(x)$. Moreover, if $x.rank \geq 1$ and either level(x) or iter(x) changes due to the qth operation, then $\phi_q(x) \leq \phi_{q-1}(x) - 1$. That is, x's potential cannot increase, and if it has positive rank and either level(x) or iter(x) changes, then x's potential drops by at least 1.

Proof Because x is not a root, the qth operation does not change $x.rank$, and because n does not change after the initial n MAKE-SET operations, $\alpha(n)$ remains unchanged as well. Hence, these components of the formula for x's potential remain the same after the qth operation. If $x.rank = 0$, then $\phi_q(x) = \phi_{q-1}(x) = 0$.

Now assume that $x.rank \geq 1$. Recall that level(x) monotonically increases over time. If the qth operation leaves level(x) unchanged, then iter(x) either increases or remains unchanged. If both level(x) and iter(x) are unchanged, then $\phi_q(x) = \phi_{q-1}(x)$. If level$(x)$ is unchanged and iter(x) increases, then it increases by at least 1, and so $\phi_q(x) \leq \phi_{q-1}(x) - 1$.

Finally, if the qth operation increases level(x), it increases by at least 1, so that the value of the term $(\alpha(n) - \text{level}(x)) \cdot x.rank$ drops by at least $x.rank$. Because level(x) increased, the value of iter(x) might drop, but according to the bound (19.6), the drop is by at most $x.rank - 1$. Thus, the increase in poten-

tial due to the change in $\text{iter}(x)$ is less than the decrease in potential due to the change in $\text{level}(x)$, yielding $\phi_q(x) \leq \phi_{q-1}(x) - 1$. ■

Our final three lemmas show that the amortized cost of each MAKE-SET, LINK, and FIND-SET operation is $O(\alpha(n))$. Recall from equation (16.2) on page 456 that the amortized cost of each operation is its actual cost plus the change in potential due to the operation.

Lemma 19.11
The amortized cost of each MAKE-SET operation is $O(1)$.

Proof Suppose that the qth operation is MAKE-SET(x). This operation creates node x with rank 0, so that $\phi_q(x) = 0$. No other ranks or potentials change, and so $\Phi_q = \Phi_{q-1}$. Noting that the actual cost of the MAKE-SET operation is $O(1)$ completes the proof. ■

Lemma 19.12
The amortized cost of each LINK operation is $O(\alpha(n))$.

Proof Suppose that the qth operation is LINK(x, y). The actual cost of the LINK operation is $O(1)$. Without loss of generality, suppose that the LINK makes y the parent of x.

To determine the change in potential due to the LINK, note that the only nodes whose potentials may change are x, y, and the children of y just prior to the operation. We'll show that the only node whose potential can increase due to the LINK is y, and that its increase is at most $\alpha(n)$:

- By Lemma 19.10, any node that is y's child just before the LINK cannot have its potential increase due to the LINK.
- From the definition (19.7) of $\phi_q(x)$, note that, since x was a root just before the qth operation, $\phi_{q-1}(x) = \alpha(n) \cdot x.rank$ at that time. If $x.rank = 0$, then $\phi_q(x) = \phi_{q-1}(x) = 0$. Otherwise,

  $$\begin{aligned}\phi_q(x) &< \alpha(n) \cdot x.rank \quad \text{(by Corollary 19.9)}\\ &= \phi_{q-1}(x)\ ,\end{aligned}$$

  and so x's potential decreases.
- Because y is a root prior to the LINK, $\phi_{q-1}(y) = \alpha(n) \cdot y.rank$. After the LINK operation, y remains a root, so that y's potential still equals $\alpha(n)$ times its rank after the operation. The LINK operation either leaves y's rank alone or increases y's rank by 1. Therefore, either $\phi_q(y) = \phi_{q-1}(y)$ or $\phi_q(y) = \phi_{q-1}(y) + \alpha(n)$.

The increase in potential due to the LINK operation, therefore, is at most $\alpha(n)$. The amortized cost of the LINK operation is $O(1) + \alpha(n) = O(\alpha(n))$. ■

Lemma 19.13
The amortized cost of each FIND-SET operation is $O(\alpha(n))$.

Proof Suppose that the qth operation is a FIND-SET and that the find path contains s nodes. The actual cost of the FIND-SET operation is $O(s)$. We will show that no node's potential increases due to the FIND-SET and that at least $\max\{0, s - (\alpha(n) + 2)\}$ nodes on the find path have their potential decrease by at least 1.

We first show that no node's potential increases. Lemma 19.10 takes care of all nodes other than the root. If x is the root, then its potential is $\alpha(n) \cdot x.rank$, which does not change due to the FIND-SET operation.

Now we show that at least $\max\{0, s - (\alpha(n) + 2)\}$ nodes have their potential decrease by at least 1. Let x be a node on the find path such that $x.rank > 0$ and x is followed somewhere on the find path by another node y that is not a root, where $\text{level}(y) = \text{level}(x)$ just before the FIND-SET operation. (Node y need not *immediately* follow x on the find path.) All but at most $\alpha(n) + 2$ nodes on the find path satisfy these constraints on x. Those that do not satisfy them are the first node on the find path (if it has rank 0), the last node on the path (i.e., the root), and the last node w on the path for which $\text{level}(w) = k$, for each $k = 0, 1, 2, \ldots, \alpha(n) - 1$.

Consider such a node x. It has positive rank and is followed somewhere on the find path by nonroot node y such that $\text{level}(y) = \text{level}(x)$ before the path compression occurs. We claim that the path compression decreases x's potential by at least 1. To prove this claim, let $k = \text{level}(x) = \text{level}(y)$ and $i = \text{iter}(x)$ before the path compression occurs. Just prior to the path compression caused by the FIND-SET, we have

$$\begin{aligned}
x.p.rank &\geq A_k^{(i)}(x.rank) && \text{(by the definition (19.5) of iter}(x)\text{)}\,, \\
y.p.rank &\geq A_k(y.rank) && \text{(by the definition (19.3) of level}(y)\text{)}\,, \\
y.rank &\geq x.p.rank && \text{(by Corollary 19.5 and because } y \text{ follows } x \text{ on the find path)}\,.
\end{aligned}$$

Putting these inequalities together gives

$$\begin{aligned}
y.p.rank &\geq A_k(y.rank) \\
&\geq A_k(x.p.rank) && \text{(because } A_k(j) \text{ is strictly increasing)} \\
&\geq A_k(A_k^{(i)}(x.rank)) \\
&= A_k^{(i+1)}(x.rank) && \text{(by the definition (3.30) of functional iteration)}\,.
\end{aligned}$$

Because path compression makes x and y have the same parent, after path compression we have $x.p.rank = y.p.rank$. The parent of y might change due to the path compression, but if it does, the rank of y's new parent compared with the rank of y's parent before path compression is either the same or greater. Since $x.rank$ does not change, $x.p.rank = y.p.rank \geq A_k^{(i+1)}(x.rank)$ after path compression. By the definition (19.5) of the iter function, the value of $\text{iter}(x)$ increases from i to at least $i + 1$. By Lemma 19.10, $\phi_q(x) \leq \phi_{q-1}(x) - 1$, so that x's potential decreases by at least 1.

The amortized cost of the FIND-SET operation is the actual cost plus the change in potential. The actual cost is $O(s)$, and we have shown that the total potential decreases by at least $\max\{0, s - (\alpha(n) + 2)\}$. The amortized cost, therefore, is at most $O(s) - (s - (\alpha(n) + 2)) = O(s) - s + O(\alpha(n)) = O(\alpha(n))$, since we can scale up the units of potential to dominate the constant hidden in $O(s)$. (See Exercise 19.4-6.) ■

Putting the preceding lemmas together yields the following theorem.

Theorem 19.14
A sequence of m MAKE-SET, UNION, and FIND-SET operations, n of which are MAKE-SET operations, can be performed on a disjoint-set forest with union by rank and path compression in $O(m\,\alpha(n))$ time.

Proof Immediate from Lemmas 19.7, 19.11, 19.12, and 19.13. ■

Exercises

19.4-1
Prove Lemma 19.4.

19.4-2
Prove that every node has rank at most $\lfloor \lg n \rfloor$.

19.4-3
In light of Exercise 19.4-2, how many bits are necessary to store $x.rank$ for each node x?

19.4-4
Using Exercise 19.4-2, give a simple proof that operations on a disjoint-set forest with union by rank but without path compression run in $O(m \lg n)$ time.

19.4-5

Professor Dante reasons that because node ranks increase strictly along a simple path to the root, node levels must monotonically increase along the path. In other words, if $x.rank > 0$ and $x.p$ is not a root, then $\text{level}(x) \leq \text{level}(x.p)$. Is the professor correct?

19.4-6

The proof of Lemma 19.13 ends with scaling the units of potential to dominate the constant hidden in the $O(s)$ term. To be more precise in the proof, you need to change the definition (19.7) of the potential function to multiply each of the two cases by a constant, say c, that dominates the constant in the $O(s)$ term. How must the rest of the analysis change to accommodate this updated potential function?

★ ***19.4-7***

Consider the function $\alpha'(n) = \min\{k : A_k(1) \geq \lg(n+1)\}$. Show that $\alpha'(n) \leq 3$ for all practical values of n and, using Exercise 19.4-2, show how to modify the potential-function argument to prove that performing a sequence of m MAKE-SET, UNION, and FIND-SET operations, n of which are MAKE-SET operations, on a disjoint-set forest with union by rank and path compression takes $O(m\alpha'(n))$ time.

Problems

19-1 Offline minimum

In the ***offline minimum problem***, you maintain a dynamic set T of elements from the domain $\{1, 2, \ldots, n\}$ under the operations INSERT and EXTRACT-MIN. The input is a sequence S of n INSERT and m EXTRACT-MIN calls, where each key in $\{1, 2, \ldots, n\}$ is inserted exactly once. Your goal is to determine which key is returned by each EXTRACT-MIN call. Specifically, you must fill in an array $extracted[1:m]$, where for $i = 1, 2, \ldots, m$, $extracted[i]$ is the key returned by the ith EXTRACT-MIN call. The problem is "offline" in the sense that you are allowed to process the entire sequence S before determining any of the returned keys.

a. Consider the following instance of the offline minimum problem, in which each operation INSERT(i) is represented by the value of i and each EXTRACT-MIN is represented by the letter E:

4, 8, E, 3, E, 9, 2, 6, E, E, E, 1, 7, E, 5 .

Fill in the correct values in the *extracted* array.

To develop an algorithm for this problem, break the sequence S into homogeneous subsequences. That is, represent S by

$$\mathrm{I}_1, \mathrm{E}, \mathrm{I}_2, \mathrm{E}, \mathrm{I}_3, \ldots, \mathrm{I}_m, \mathrm{E}, \mathrm{I}_{m+1} ,$$

where each E represents a single EXTRACT-MIN call and each I_j represents a (possibly empty) sequence of INSERT calls. For each subsequence I_j, initially place the keys inserted by these operations into a set K_j, which is empty if I_j is empty. Then execute the OFFLINE-MINIMUM procedure.

```
OFFLINE-MINIMUM(m, n)
1  for i = 1 to n
2      determine j such that i ∈ K_j
3      if j ≠ m + 1
4          extracted[j] = i
5          let l be the smallest value greater than j for which set K_l exists
6          K_l = K_j ∪ K_l, destroying K_j
7  return extracted
```

b. Argue that the array *extracted* returned by OFFLINE-MINIMUM is correct.

c. Describe how to implement OFFLINE-MINIMUM efficiently with a disjoint-set data structure. Give as tight a bound as you can on the worst-case running time of your implementation.

19-2 Depth determination

In the ***depth-determination problem***, you maintain a forest $\mathcal{F} = \{T_i\}$ of rooted trees under three operations:

MAKE-TREE(v) creates a tree whose only node is v.

FIND-DEPTH(v) returns the depth of node v within its tree.

GRAFT(r, v) makes node r, which is assumed to be the root of a tree, become the child of node v, which is assumed to be in a different tree from r but may or may not itself be a root.

a. Suppose that you use a tree representation similar to a disjoint-set forest: $v.p$ is the parent of node v, except that $v.p = v$ if v is a root. Suppose further that you implement GRAFT(r, v) by setting $r.p = v$ and FIND-DEPTH(v) by following the find path from v up to the root, returning a count of all nodes other than v encountered. Show that the worst-case running time of a sequence of m MAKE-TREE, FIND-DEPTH, and GRAFT operations is $\Theta(m^2)$.

By using the union-by-rank and path-compression heuristics, you can reduce the worst-case running time. Use the disjoint-set forest $\mathcal{S} = \{S_i\}$, where each set S_i (which is itself a tree) corresponds to a tree T_i in the forest $\mathcal{F}$. The tree structure within a set S_i, however, does not necessarily correspond to that of T_i. In fact, the implementation of S_i does not record the exact parent-child relationships but nevertheless allows you to determine any node's depth in T_i.

The key idea is to maintain in each node v a "pseudodistance" $v.d$, which is defined so that the sum of the pseudodistances along the simple path from v to the root of its set S_i equals the depth of v in T_i. That is, if the simple path from v to its root in S_i is $v_0, v_1, \ldots, v_k$, where $v_0 = v$ and v_k is S_i's root, then the depth of v in T_i is $\sum_{j=0}^{k} v_j.d$.

b. Give an implementation of MAKE-TREE.

c. Show how to modify FIND-SET to implement FIND-DEPTH. Your implementation should perform path compression, and its running time should be linear in the length of the find path. Make sure that your implementation updates pseudodistances correctly.

d. Show how to implement GRAFT(r, v), which combines the sets containing r and v, by modifying the UNION and LINK procedures. Make sure that your implementation updates pseudodistances correctly. Note that the root of a set S_i is not necessarily the root of the corresponding tree T_i.

e. Give a tight bound on the worst-case running time of a sequence of m MAKE-TREE, FIND-DEPTH, and GRAFT operations, n of which are MAKE-TREE operations.

19-3 Tarjan's offline lowest-common-ancestors algorithm

The ***lowest common ancestor*** of two nodes u and v in a rooted tree T is the node w that is an ancestor of both u and v and that has the greatest depth in T. In the ***offline lowest-common-ancestors problem***, you are given a rooted tree T and an arbitrary set $P = \{\{u, v\}\}$ of unordered pairs of nodes in T, and you wish to determine the lowest common ancestor of each pair in P.

To solve the offline lowest-common-ancestors problem, the LCA procedure on the following page performs a tree walk of T with the initial call LCA$(T.root)$. Assume that each node is colored WHITE prior to the walk.

a. Argue that line 10 executes exactly once for each pair $\{u, v\} \in P$.

b. Argue that at the time of the call LCA(u), the number of sets in the disjoint-set data structure equals the depth of u in T.

```
LCA(u)
 1  MAKE-SET(u)
 2  FIND-SET(u).ancestor = u
 3  for each child v of u in T
 4       LCA(v)
 5       UNION(u, v)
 6       FIND-SET(u).ancestor = u
 7  u.color = BLACK
 8  for each node v such that {u, v} ∈ P
 9       if v.color == BLACK
10            print "The lowest common ancestor of"
                  u "and" v "is" FIND-SET(v).ancestor
```

c. Prove that LCA correctly prints the lowest common ancestor of u and v for each pair $\{u, v\} \in P$.

d. Analyze the running time of LCA, assuming that you use the implementation of the disjoint-set data structure in Section 19.3.

Chapter notes

Many of the important results for disjoint-set data structures are due at least in part to R. E. Tarjan. Using aggregate analysis, Tarjan [427, 429] gave the first tight upper bound in terms of the very slowly growing inverse $\hat{\alpha}(m, n)$ of Ackermann's function. (The function $A_k(j)$ given in Section 19.4 is similar to Ackermann's function, and the function $\alpha(n)$ is similar to $\hat{\alpha}(m, n)$. Both $\alpha(n)$ and $\hat{\alpha}(m, n)$ are at most 4 for all conceivable values of m and n.) An upper bound of $O(m \lg^* n)$ was proven earlier by Hopcroft and Ullman [5, 227]. The treatment in Section 19.4 is adapted from a later analysis by Tarjan [431], which is based on an analysis by Kozen [270]. Harfst and Reingold [209] give a potential-based version of Tarjan's earlier bound.

Tarjan and van Leeuwen [432] discuss variants on the path-compression heuristic, including "one-pass methods," which sometimes offer better constant factors in their performance than do two-pass methods. As with Tarjan's earlier analyses of the basic path-compression heuristic, the analyses by Tarjan and van Leeuwen are aggregate. Harfst and Reingold [209] later showed how to make a small change to the potential function to adapt their path-compression analysis to these one-pass variants. Goel et al. [182] prove that linking disjoint-set trees randomly yields the

same asymptotic running time as union by rank. Gabow and Tarjan [166] show that in certain applications, the disjoint-set operations can be made to run in $O(m)$ time.

Tarjan [428] showed that a lower bound of $\Omega(m\,\hat{\alpha}(m,n))$ time is required for operations on any disjoint-set data structure satisfying certain technical conditions. This lower bound was later generalized by Fredman and Saks [155], who showed that in the worst case, $\Omega(m\,\hat{\alpha}(m,n))$ $(\lg n)$-bit words of memory must be accessed.