

III Data Structures

Introduction

Sets are as fundamental to computer science as they are to mathematics. Whereas mathematical sets are unchanging, the sets manipulated by algorithms can grow, shrink, or otherwise change over time. We call such sets *dynamic*. The next five chapters present some basic techniques for representing finite dynamic sets and manipulating them on a computer.

Algorithms may require several different types of operations to be performed on sets. For example, many algorithms need only the ability to insert elements into, delete elements from, and test membership in a set. We call a dynamic set that supports these operations a *dictionary*. Other algorithms require more complicated operations. For example, min-priority queues, which Chapter 6 introduced in the context of the heap data structure, support the operations of inserting an element into and extracting the smallest element from a set. The best way to implement a dynamic set depends upon the operations that must be supported.

Elements of a dynamic set

In a typical implementation of a dynamic set, each element is represented by an object whose attributes can be examined and manipulated if we have a pointer to the object. (Section 10.3 discusses the implementation of objects and pointers in programming environments that do not contain them as basic data types.) Some kinds of dynamic sets assume that one of the object's attributes is an identifying *key*. If the keys are all different, we can think of the dynamic set as being a set of key values. The object may contain *satellite data*, which are carried around in other object attributes but are otherwise unused by the set implementation. It may

also have attributes that are manipulated by the set operations; these attributes may contain data or pointers to other objects in the set.

Some dynamic sets presuppose that the keys are drawn from a totally ordered set, such as the real numbers, or the set of all words under the usual alphabetic ordering. A total ordering allows us to define the minimum element of the set, for example, or to speak of the next element larger than a given element in a set.

Operations on dynamic sets

Operations on a dynamic set can be grouped into two categories: *queries*, which simply return information about the set, and *modifying operations*, which change the set. Here is a list of typical operations. Any specific application will usually require only a few of these to be implemented.

SEARCH(S, k)

A query that, given a set S and a key value k , returns a pointer x to an element in S such that $x.key = k$, or NIL if no such element belongs to S .

INSERT(S, x)

A modifying operation that augments the set S with the element pointed to by x . We usually assume that any attributes in element x needed by the set implementation have already been initialized.

DELETE(S, x)

A modifying operation that, given a pointer x to an element in the set S , removes x from S . (Note that this operation takes a pointer to an element x , not a key value.)

MINIMUM(S)

A query on a totally ordered set S that returns a pointer to the element of S with the smallest key.

MAXIMUM(S)

A query on a totally ordered set S that returns a pointer to the element of S with the largest key.

SUCCESSOR(S, x)

A query that, given an element x whose key is from a totally ordered set S , returns a pointer to the next larger element in S , or NIL if x is the maximum element.

PREDECESSOR(S, x)

A query that, given an element x whose key is from a totally ordered set S , returns a pointer to the next smaller element in S , or NIL if x is the minimum element.

In some situations, we can extend the queries `SUCCESSOR` and `PREDECESSOR` so that they apply to sets with nondistinct keys. For a set on n keys, the normal presumption is that a call to `MINIMUM` followed by $n - 1$ calls to `SUCCESSOR` enumerates the elements in the set in sorted order.

We usually measure the time taken to execute a set operation in terms of the size of the set. For example, Chapter 13 describes a data structure that can support any of the operations listed above on a set of size n in time $O(\lg n)$.

Overview of Part III

Chapters 10–14 describe several data structures that we can use to implement dynamic sets; we shall use many of these later to construct efficient algorithms for a variety of problems. We already saw another important data structure—the heap—in Chapter 6.

Chapter 10 presents the essentials of working with simple data structures such as stacks, queues, linked lists, and rooted trees. It also shows how to implement objects and pointers in programming environments that do not support them as primitives. If you have taken an introductory programming course, then much of this material should be familiar to you.

Chapter 11 introduces hash tables, which support the dictionary operations `INSERT`, `DELETE`, and `SEARCH`. In the worst case, hashing requires $\Theta(n)$ time to perform a `SEARCH` operation, but the expected time for hash-table operations is $O(1)$. The analysis of hashing relies on probability, but most of the chapter requires no background in the subject.

Binary search trees, which are covered in Chapter 12, support all the dynamic-set operations listed above. In the worst case, each operation takes $\Theta(n)$ time on a tree with n elements, but on a randomly built binary search tree, the expected time for each operation is $O(\lg n)$. Binary search trees serve as the basis for many other data structures.

Chapter 13 introduces red-black trees, which are a variant of binary search trees. Unlike ordinary binary search trees, red-black trees are guaranteed to perform well: operations take $O(\lg n)$ time in the worst case. A red-black tree is a balanced search tree; Chapter 18 in Part V presents another kind of balanced search tree, called a B-tree. Although the mechanics of red-black trees are somewhat intricate, you can glean most of their properties from the chapter without studying the mechanics in detail. Nevertheless, you probably will find walking through the code to be quite instructive.

In Chapter 14, we show how to augment red-black trees to support operations other than the basic ones listed above. First, we augment them so that we can dynamically maintain order statistics for a set of keys. Then, we augment them in a different way to maintain intervals of real numbers.

10

Elementary Data Structures

In this chapter, we examine the representation of dynamic sets by simple data structures that use pointers. Although we can construct many complex data structures using pointers, we present only the rudimentary ones: stacks, queues, linked lists, and rooted trees. We also show ways to synthesize objects and pointers from arrays.

10.1 Stacks and queues

Stacks and queues are dynamic sets in which the element removed from the set by the DELETE operation is prespecified. In a *stack*, the element deleted from the set is the one most recently inserted: the stack implements a *last-in, first-out*, or *LIFO*, policy. Similarly, in a *queue*, the element deleted is always the one that has been in the set for the longest time: the queue implements a *first-in, first-out*, or *FIFO*, policy. There are several efficient ways to implement stacks and queues on a computer. In this section we show how to use a simple array to implement each.

Stacks

The INSERT operation on a stack is often called PUSH, and the DELETE operation, which does not take an element argument, is often called POP. These names are allusions to physical stacks, such as the spring-loaded stacks of plates used in cafeterias. The order in which plates are popped from the stack is the reverse of the order in which they were pushed onto the stack, since only the top plate is accessible.

As Figure 10.1 shows, we can implement a stack of at most n elements with an array $S[1..n]$. The array has an attribute $S.top$ that indexes the most recently

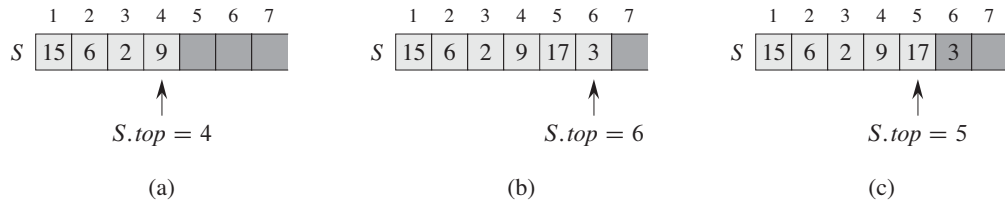


Figure 10.1 An array implementation of a stack S . Stack elements appear only in the lightly shaded positions. (a) Stack S has 4 elements. The top element is 9. (b) Stack S after the calls $PUSH(S, 17)$ and $PUSH(S, 3)$. (c) Stack S after the call $POP(S)$ has returned the element 3, which is the one most recently pushed. Although element 3 still appears in the array, it is no longer in the stack; the top is element 17.

inserted element. The stack consists of elements $S[1 \dots S.top]$, where $S[1]$ is the element at the bottom of the stack and $S[S.top]$ is the element at the top.

When $S.top = 0$, the stack contains no elements and is *empty*. We can test to see whether the stack is empty by query operation `STACK-EMPTY`. If we attempt to pop an empty stack, we say the stack *underflows*, which is normally an error. If $S.top$ exceeds n , the stack *overflows*. (In our pseudocode implementation, we don't worry about stack overflow.)

We can implement each of the stack operations with just a few lines of code:

`STACK-EMPTY(S)`

```

1  if  $S.top == 0$ 
2      return TRUE
3  else return FALSE

```

`PUSH(S, x)`

```

1   $S.top = S.top + 1$ 
2   $S[S.top] = x$ 

```

`POP(S)`

```

1  if STACK-EMPTY( $S$ )
2      error "underflow"
3  else  $S.top = S.top - 1$ 
4      return  $S[S.top + 1]$ 

```

Figure 10.1 shows the effects of the modifying operations `PUSH` and `POP`. Each of the three stack operations takes $O(1)$ time.

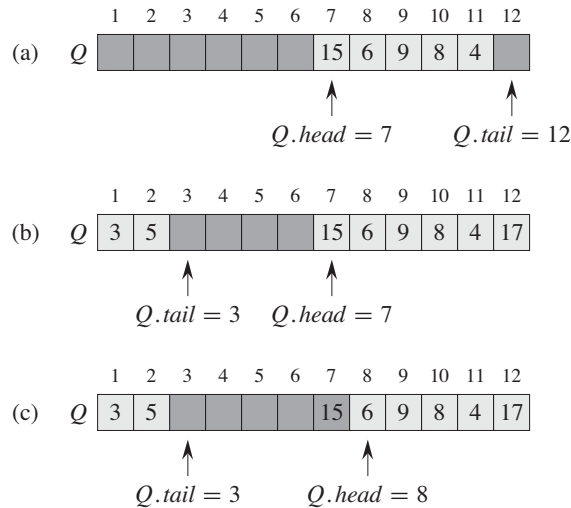


Figure 10.2 A queue implemented using an array $Q[1..12]$. Queue elements appear only in the lightly shaded positions. (a) The queue has 5 elements, in locations $Q[7..11]$. (b) The configuration of the queue after the calls $ENQUEUE(Q, 17)$, $ENQUEUE(Q, 3)$, and $ENQUEUE(Q, 5)$. (c) The configuration of the queue after the call $DEQUEUE(Q)$ returns the key value 15 formerly at the head of the queue. The new head has key 6.

Queues

We call the INSERT operation on a queue **ENQUEUE**, and we call the DELETE operation **DEQUEUE**; like the stack operation **POP**, **DEQUEUE** takes no element argument. The FIFO property of a queue causes it to operate like a line of customers waiting to pay a cashier. The queue has a **head** and a **tail**. When an element is enqueued, it takes its place at the tail of the queue, just as a newly arriving customer takes a place at the end of the line. The element dequeued is always the one at the head of the queue, like the customer at the head of the line who has waited the longest.

Figure 10.2 shows one way to implement a queue of at most $n - 1$ elements using an array $Q[1..n]$. The queue has an attribute $Q.head$ that indexes, or points to, its head. The attribute $Q.tail$ indexes the next location at which a newly arriving element will be inserted into the queue. The elements in the queue reside in locations $Q.head, Q.head + 1, \dots, Q.tail - 1$, where we “wrap around” in the sense that location 1 immediately follows location n in a circular order. When $Q.head = Q.tail$, the queue is empty. Initially, we have $Q.head = Q.tail = 1$. If we attempt to dequeue an element from an empty queue, the queue underflows.

When $Q.head = Q.tail + 1$, the queue is full, and if we attempt to enqueue an element, then the queue overflows.

In our procedures ENQUEUE and DEQUEUE, we have omitted the error checking for underflow and overflow. (Exercise 10.1-4 asks you to supply code that checks for these two error conditions.) The pseudocode assumes that $n = Q.length$.

```

ENQUEUE( $Q, x$ )
1   $Q[Q.tail] = x$ 
2  if  $Q.tail == Q.length$ 
3       $Q.tail = 1$ 
4  else  $Q.tail = Q.tail + 1$ 

DEQUEUE( $Q$ )
1   $x = Q[Q.head]$ 
2  if  $Q.head == Q.length$ 
3       $Q.head = 1$ 
4  else  $Q.head = Q.head + 1$ 
5  return  $x$ 

```

Figure 10.2 shows the effects of the ENQUEUE and DEQUEUE operations. Each operation takes $O(1)$ time.

Exercises

10.1-1

Using Figure 10.1 as a model, illustrate the result of each operation in the sequence $PUSH(S, 4)$, $PUSH(S, 1)$, $PUSH(S, 3)$, $POP(S)$, $PUSH(S, 8)$, and $POP(S)$ on an initially empty stack S stored in array $S[1..6]$.

10.1-2

Explain how to implement two stacks in one array $A[1..n]$ in such a way that neither stack overflows unless the total number of elements in both stacks together is n . The PUSH and POP operations should run in $O(1)$ time.

10.1-3

Using Figure 10.2 as a model, illustrate the result of each operation in the sequence $ENQUEUE(Q, 4)$, $ENQUEUE(Q, 1)$, $ENQUEUE(Q, 3)$, $DEQUEUE(Q)$, $ENQUEUE(Q, 8)$, and $DEQUEUE(Q)$ on an initially empty queue Q stored in array $Q[1..6]$.

10.1-4

Rewrite ENQUEUE and DEQUEUE to detect underflow and overflow of a queue.

10.1-5

Whereas a stack allows insertion and deletion of elements at only one end, and a queue allows insertion at one end and deletion at the other end, a **deque** (double-ended queue) allows insertion and deletion at both ends. Write four $O(1)$ -time procedures to insert elements into and delete elements from both ends of a deque implemented by an array.

10.1-6

Show how to implement a queue using two stacks. Analyze the running time of the queue operations.

10.1-7

Show how to implement a stack using two queues. Analyze the running time of the stack operations.

10.2 Linked lists

A **linked list** is a data structure in which the objects are arranged in a linear order. Unlike an array, however, in which the linear order is determined by the array indices, the order in a linked list is determined by a pointer in each object. Linked lists provide a simple, flexible representation for dynamic sets, supporting (though not necessarily efficiently) all the operations listed on page 230.

As shown in Figure 10.3, each element of a **doubly linked list** L is an object with an attribute *key* and two other pointer attributes: *next* and *prev*. The object may also contain other satellite data. Given an element x in the list, $x.next$ points to its successor in the linked list, and $x.prev$ points to its predecessor. If $x.prev = \text{NIL}$, the element x has no predecessor and is therefore the first element, or **head**, of the list. If $x.next = \text{NIL}$, the element x has no successor and is therefore the last element, or **tail**, of the list. An attribute $L.head$ points to the first element of the list. If $L.head = \text{NIL}$, the list is empty.

A list may have one of several forms. It may be either singly linked or doubly linked, it may be sorted or not, and it may be circular or not. If a list is **singly linked**, we omit the *prev* pointer in each element. If a list is **sorted**, the linear order of the list corresponds to the linear order of keys stored in elements of the list; the minimum element is then the head of the list, and the maximum element is the tail. If the list is **unsorted**, the elements can appear in any order. In a **circular list**, the *prev* pointer of the head of the list points to the tail, and the *next* pointer of the tail of the list points to the head. We can think of a circular list as a ring of

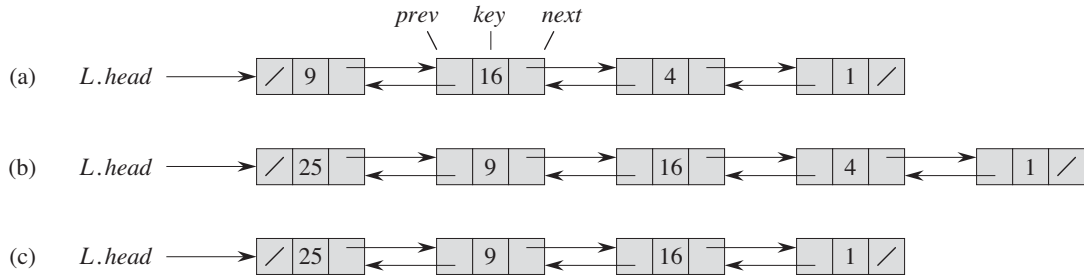


Figure 10.3 (a) A doubly linked list L representing the dynamic set $\{1, 4, 9, 16\}$. Each element in the list is an object with attributes for the key and pointers (shown by arrows) to the next and previous objects. The *next* attribute of the tail and the *prev* attribute of the head are NIL, indicated by a diagonal slash. The attribute $L.head$ points to the head. (b) Following the execution of $LIST-INSERT(L, x)$, where $x.key = 25$, the linked list has a new object with key 25 as the new head. This new object points to the old head with key 9. (c) The result of the subsequent call $LIST-DELETE(L, x)$, where x points to the object with key 4.

elements. In the remainder of this section, we assume that the lists with which we are working are unsorted and doubly linked.

Searching a linked list

The procedure $LIST-SEARCH(L, k)$ finds the first element with key k in list L by a simple linear search, returning a pointer to this element. If no object with key k appears in the list, then the procedure returns NIL. For the linked list in Figure 10.3(a), the call $LIST-SEARCH(L, 4)$ returns a pointer to the third element, and the call $LIST-SEARCH(L, 7)$ returns NIL.

$LIST-SEARCH(L, k)$

```

1   $x = L.head$ 
2  while  $x \neq \text{NIL}$  and  $x.key \neq k$ 
3       $x = x.next$ 
4  return  $x$ 
```

To search a list of n objects, the $LIST-SEARCH$ procedure takes $\Theta(n)$ time in the worst case, since it may have to search the entire list.

Inserting into a linked list

Given an element x whose *key* attribute has already been set, the $LIST-INSERT$ procedure “splices” x onto the front of the linked list, as shown in Figure 10.3(b).

LIST-INSERT(L, x)

```

1   $x.next = L.head$ 
2  if  $L.head \neq \text{NIL}$ 
3       $L.head.prev = x$ 
4   $L.head = x$ 
5   $x.prev = \text{NIL}$ 

```

(Recall that our attribute notation can cascade, so that $L.head.prev$ denotes the *prev* attribute of the object that $L.head$ points to.) The running time for LIST-INSERT on a list of n elements is $O(1)$.

Deleting from a linked list

The procedure LIST-DELETE removes an element x from a linked list L . It must be given a pointer to x , and it then “splices” x out of the list by updating pointers. If we wish to delete an element with a given key, we must first call LIST-SEARCH to retrieve a pointer to the element.

LIST-DELETE(L, x)

```

1  if  $x.prev \neq \text{NIL}$ 
2       $x.prev.next = x.next$ 
3  else  $L.head = x.next$ 
4  if  $x.next \neq \text{NIL}$ 
5       $x.next.prev = x.prev$ 

```

Figure 10.3(c) shows how an element is deleted from a linked list. LIST-DELETE runs in $O(1)$ time, but if we wish to delete an element with a given key, $\Theta(n)$ time is required in the worst case because we must first call LIST-SEARCH to find the element.

Sentinels

The code for LIST-DELETE would be simpler if we could ignore the boundary conditions at the head and tail of the list:

LIST-DELETE'(L, x)

```

1   $x.prev.next = x.next$ 
2   $x.next.prev = x.prev$ 

```

A *sentinel* is a dummy object that allows us to simplify boundary conditions. For example, suppose that we provide with list L an object $L.nil$ that represents NIL

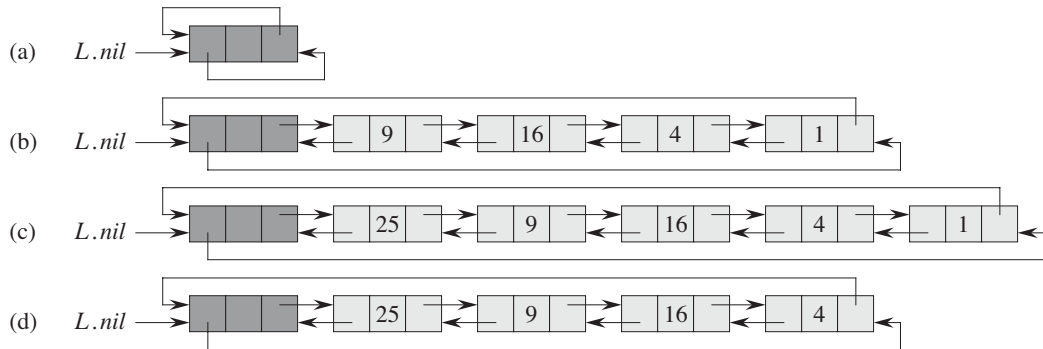


Figure 10.4 A circular, doubly linked list with a sentinel. The sentinel $L.nil$ appears between the head and tail. The attribute $L.head$ is no longer needed, since we can access the head of the list by $L.nil.next$. (a) An empty list. (b) The linked list from Figure 10.3(a), with key 9 at the head and key 1 at the tail. (c) The list after executing $LIST-INSERT'(L, x)$, where $x.key = 25$. The new object becomes the head of the list. (d) The list after deleting the object with key 1. The new tail is the object with key 4.

but has all the attributes of the other objects in the list. Wherever we have a reference to NIL in list code, we replace it by a reference to the sentinel $L.nil$. As shown in Figure 10.4, this change turns a regular doubly linked list into a **circular, doubly linked list with a sentinel**, in which the sentinel $L.nil$ lies between the head and tail. The attribute $L.nil.next$ points to the head of the list, and $L.nil.prev$ points to the tail. Similarly, both the $next$ attribute of the tail and the $prev$ attribute of the head point to $L.nil$. Since $L.nil.next$ points to the head, we can eliminate the attribute $L.head$ altogether, replacing references to it by references to $L.nil.next$. Figure 10.4(a) shows that an empty list consists of just the sentinel, and both $L.nil.next$ and $L.nil.prev$ point to $L.nil$.

The code for LIST-SEARCH remains the same as before, but with the references to NIL and $L.head$ changed as specified above:

```
LIST-SEARCH'(L, k)
1   $x = L.nil.next$ 
2  while  $x \neq L.nil$  and  $x.key \neq k$ 
3       $x = x.next$ 
4  return  $x$ 
```

We use the two-line procedure LIST-DELETE' from before to delete an element from the list. The following procedure inserts an element into the list:

LIST-INSERT'(L, x)

```

1   $x.next = L.nil.next$ 
2   $L.nil.next.prev = x$ 
3   $L.nil.next = x$ 
4   $x.prev = L.nil$ 

```

Figure 10.4 shows the effects of LIST-INSERT' and LIST-DELETE' on a sample list.

Sentinels rarely reduce the asymptotic time bounds of data structure operations, but they can reduce constant factors. The gain from using sentinels within loops is usually a matter of clarity of code rather than speed; the linked list code, for example, becomes simpler when we use sentinels, but we save only $O(1)$ time in the LIST-INSERT' and LIST-DELETE' procedures. In other situations, however, the use of sentinels helps to tighten the code in a loop, thus reducing the coefficient of, say, n or n^2 in the running time.

We should use sentinels judiciously. When there are many small lists, the extra storage used by their sentinels can represent significant wasted memory. In this book, we use sentinels only when they truly simplify the code.

Exercises

10.2-1

Can you implement the dynamic-set operation INSERT on a singly linked list in $O(1)$ time? How about DELETE?

10.2-2

Implement a stack using a singly linked list L . The operations PUSH and POP should still take $O(1)$ time.

10.2-3

Implement a queue by a singly linked list L . The operations ENQUEUE and DEQUEUE should still take $O(1)$ time.

10.2-4

As written, each loop iteration in the LIST-SEARCH' procedure requires two tests: one for $x \neq L.nil$ and one for $x.key \neq k$. Show how to eliminate the test for $x \neq L.nil$ in each iteration.

10.2-5

Implement the dictionary operations INSERT, DELETE, and SEARCH using singly linked, circular lists. What are the running times of your procedures?

10.2-6

The dynamic-set operation UNION takes two disjoint sets S_1 and S_2 as input, and it returns a set $S = S_1 \cup S_2$ consisting of all the elements of S_1 and S_2 . The sets S_1 and S_2 are usually destroyed by the operation. Show how to support UNION in $O(1)$ time using a suitable list data structure.

10.2-7

Give a $\Theta(n)$ -time nonrecursive procedure that reverses a singly linked list of n elements. The procedure should use no more than constant storage beyond that needed for the list itself.

10.2-8 ★

Explain how to implement doubly linked lists using only one pointer value $x.np$ per item instead of the usual two ($next$ and $prev$). Assume that all pointer values can be interpreted as k -bit integers, and define $x.np$ to be $x.np = x.next \text{ XOR } x.prev$, the k -bit “exclusive-or” of $x.next$ and $x.prev$. (The value NIL is represented by 0.) Be sure to describe what information you need to access the head of the list. Show how to implement the SEARCH, INSERT, and DELETE operations on such a list. Also show how to reverse such a list in $O(1)$ time.

10.3 Implementing pointers and objects

How do we implement pointers and objects in languages that do not provide them? In this section, we shall see two ways of implementing linked data structures without an explicit pointer data type. We shall synthesize objects and pointers from arrays and array indices.

A multiple-array representation of objects

We can represent a collection of objects that have the same attributes by using an array for each attribute. As an example, Figure 10.5 shows how we can implement the linked list of Figure 10.3(a) with three arrays. The array *key* holds the values of the keys currently in the dynamic set, and the pointers reside in the arrays *next* and *prev*. For a given array index x , the array entries $key[x]$, $next[x]$, and $prev[x]$ represent an object in the linked list. Under this interpretation, a pointer x is simply a common index into the *key*, *next*, and *prev* arrays.

In Figure 10.3(a), the object with key 4 follows the object with key 16 in the linked list. In Figure 10.5, key 4 appears in $key[2]$, and key 16 appears in $key[5]$, and so $next[5] = 2$ and $prev[2] = 5$. Although the constant NIL appears in the *next*

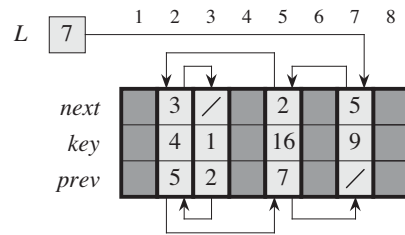


Figure 10.5 The linked list of Figure 10.3(a) represented by the arrays *key*, *next*, and *prev*. Each vertical slice of the arrays represents a single object. Stored pointers correspond to the array indices shown at the top; the arrows show how to interpret them. Lightly shaded object positions contain list elements. The variable *L* keeps the index of the head.

attribute of the tail and the *prev* attribute of the head, we usually use an integer (such as 0 or -1) that cannot possibly represent an actual index into the arrays. A variable *L* holds the index of the head of the list.

A single-array representation of objects

The words in a computer memory are typically addressed by integers from 0 to $M - 1$, where M is a suitably large integer. In many programming languages, an object occupies a contiguous set of locations in the computer memory. A pointer is simply the address of the first memory location of the object, and we can address other memory locations within the object by adding an offset to the pointer.

We can use the same strategy for implementing objects in programming environments that do not provide explicit pointer data types. For example, Figure 10.6 shows how to use a single array *A* to store the linked list from Figures 10.3(a) and 10.5. An object occupies a contiguous subarray $A[j \dots k]$. Each attribute of the object corresponds to an offset in the range from 0 to $k - j$, and a pointer to the object is the index *j*. In Figure 10.6, the offsets corresponding to *key*, *next*, and *prev* are 0, 1, and 2, respectively. To read the value of *i.prev*, given a pointer *i*, we add the value *i* of the pointer to the offset 2, thus reading $A[i + 2]$.

The single-array representation is flexible in that it permits objects of different lengths to be stored in the same array. The problem of managing such a heterogeneous collection of objects is more difficult than the problem of managing a homogeneous collection, where all objects have the same attributes. Since most of the data structures we shall consider are composed of homogeneous elements, it will be sufficient for our purposes to use the multiple-array representation of objects.

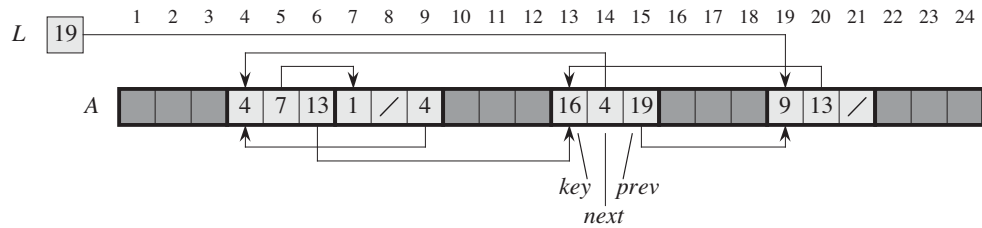


Figure 10.6 The linked list of Figures 10.3(a) and 10.5 represented in a single array *A*. Each list element is an object that occupies a contiguous subarray of length 3 within the array. The three attributes *key*, *next*, and *prev* correspond to the offsets 0, 1, and 2, respectively, within each object. A pointer to an object is the index of the first element of the object. Objects containing list elements are lightly shaded, and arrows show the list ordering.

Allocating and freeing objects

To insert a key into a dynamic set represented by a doubly linked list, we must allocate a pointer to a currently unused object in the linked-list representation. Thus, it is useful to manage the storage of objects not currently used in the linked-list representation so that one can be allocated. In some systems, a **garbage collector** is responsible for determining which objects are unused. Many applications, however, are simple enough that they can bear responsibility for returning an unused object to a storage manager. We shall now explore the problem of allocating and freeing (or deallocating) homogeneous objects using the example of a doubly linked list represented by multiple arrays.

Suppose that the arrays in the multiple-array representation have length m and that at some moment the dynamic set contains $n \leq m$ elements. Then n objects represent elements currently in the dynamic set, and the remaining $m - n$ objects are **free**; the free objects are available to represent elements inserted into the dynamic set in the future.

We keep the free objects in a singly linked list, which we call the **free list**. The free list uses only the *next* array, which stores the *next* pointers within the list. The head of the free list is held in the global variable *free*. When the dynamic set represented by linked list *L* is nonempty, the free list may be intertwined with list *L*, as shown in Figure 10.7. Note that each object in the representation is either in list *L* or in the free list, but not in both.

The free list acts like a stack: the next object allocated is the last one freed. We can use a list implementation of the stack operations PUSH and POP to implement the procedures for allocating and freeing objects, respectively. We assume that the global variable *free* used in the following procedures points to the first element of the free list.

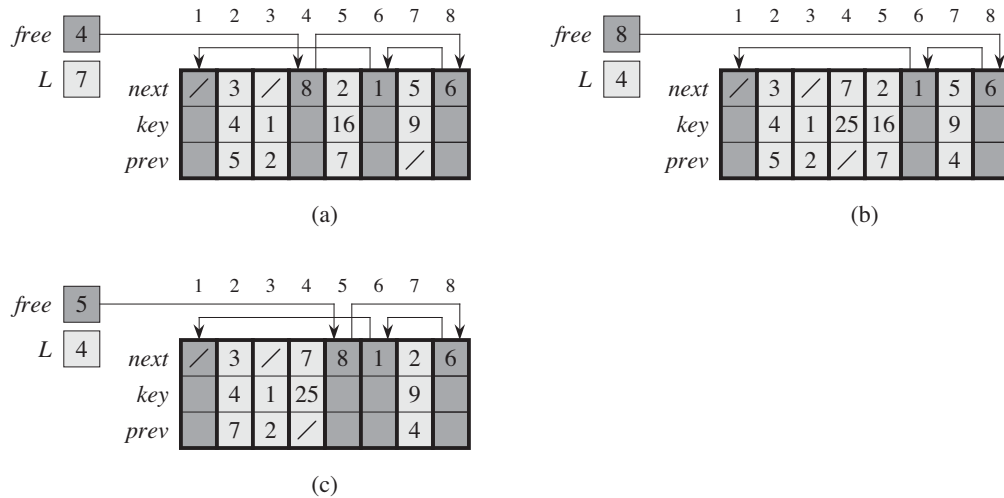


Figure 10.7 The effect of the `ALLOCATE-OBJECT` and `FREE-OBJECT` procedures. (a) The list of Figure 10.5 (lightly shaded) and a free list (heavily shaded). Arrows show the free-list structure. (b) The result of calling `ALLOCATE-OBJECT()` (which returns index 4), setting `key[4]` to 25, and calling `LIST-INSERT(L, 4)`. The new free-list head is object 8, which had been `next[4]` on the free list. (c) After executing `LIST-DELETE(L, 5)`, we call `FREE-OBJECT(5)`. Object 5 becomes the new free-list head, with object 8 following it on the free list.

`ALLOCATE-OBJECT()`

```

1  if free == NIL
2      error "out of space"
3  else x = free
4      free = x.next
5      return x

```

`FREE-OBJECT(x)`

```

1  x.next = free
2  free = x

```

The free list initially contains all n unallocated objects. Once the free list has been exhausted, running the `ALLOCATE-OBJECT` procedure signals an error. We can even service several linked lists with just a single free list. Figure 10.8 shows two linked lists and a free list intertwined through `key`, `next`, and `prev` arrays.

The two procedures run in $O(1)$ time, which makes them quite practical. We can modify them to work for any homogeneous collection of objects by letting any one of the attributes in the object act like a `next` attribute in the free list.

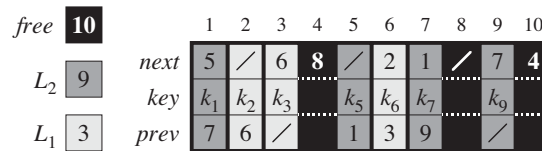


Figure 10.8 Two linked lists, L_1 (lightly shaded) and L_2 (heavily shaded), and a free list (darkened) intertwined.

Exercises

10.3-1

Draw a picture of the sequence $\langle 13, 4, 8, 19, 5, 11 \rangle$ stored as a doubly linked list using the multiple-array representation. Do the same for the single-array representation.

10.3-2

Write the procedures `ALLOCATE-OBJECT` and `FREE-OBJECT` for a homogeneous collection of objects implemented by the single-array representation.

10.3-3

Why don't we need to set or reset the *prev* attributes of objects in the implementation of the `ALLOCATE-OBJECT` and `FREE-OBJECT` procedures?

10.3-4

It is often desirable to keep all elements of a doubly linked list compact in storage, using, for example, the first m index locations in the multiple-array representation. (This is the case in a paged, virtual-memory computing environment.) Explain how to implement the procedures `ALLOCATE-OBJECT` and `FREE-OBJECT` so that the representation is compact. Assume that there are no pointers to elements of the linked list outside the list itself. (*Hint:* Use the array implementation of a stack.)

10.3-5

Let L be a doubly linked list of length n stored in arrays *key*, *prev*, and *next* of length m . Suppose that these arrays are managed by `ALLOCATE-OBJECT` and `FREE-OBJECT` procedures that keep a doubly linked free list F . Suppose further that of the m items, exactly n are on list L and $m - n$ are on the free list. Write a procedure `COMPACTIFY-LIST(L, F)` that, given the list L and the free list F , moves the items in L so that they occupy array positions $1, 2, \dots, n$ and adjusts the free list F so that it remains correct, occupying array positions $n + 1, n + 2, \dots, m$. The running time of your procedure should be $\Theta(n)$, and it should use only a constant amount of extra space. Argue that your procedure is correct.

10.4 Representing rooted trees

The methods for representing lists given in the previous section extend to any homogeneous data structure. In this section, we look specifically at the problem of representing rooted trees by linked data structures. We first look at binary trees, and then we present a method for rooted trees in which nodes can have an arbitrary number of children.

We represent each node of a tree by an object. As with linked lists, we assume that each node contains a *key* attribute. The remaining attributes of interest are pointers to other nodes, and they vary according to the type of tree.

Binary trees

Figure 10.9 shows how we use the attributes *p*, *left*, and *right* to store pointers to the parent, left child, and right child of each node in a binary tree *T*. If $x.p = \text{NIL}$, then *x* is the root. If node *x* has no left child, then $x.\text{left} = \text{NIL}$, and similarly for the right child. The root of the entire tree *T* is pointed to by the attribute *T.root*. If $T.\text{root} = \text{NIL}$, then the tree is empty.

Rooted trees with unbounded branching

We can extend the scheme for representing a binary tree to any class of trees in which the number of children of each node is at most some constant *k*: we replace the *left* and *right* attributes by $\text{child}_1, \text{child}_2, \dots, \text{child}_k$. This scheme no longer works when the number of children of a node is unbounded, since we do not know how many attributes (arrays in the multiple-array representation) to allocate in advance. Moreover, even if the number of children *k* is bounded by a large constant but most nodes have a small number of children, we may waste a lot of memory.

Fortunately, there is a clever scheme to represent trees with arbitrary numbers of children. It has the advantage of using only $O(n)$ space for any *n*-node rooted tree. The **left-child, right-sibling representation** appears in Figure 10.10. As before, each node contains a parent pointer *p*, and *T.root* points to the root of tree *T*. Instead of having a pointer to each of its children, however, each node *x* has only two pointers:

1. *x.left-child* points to the leftmost child of node *x*, and
2. *x.right-sibling* points to the sibling of *x* immediately to its right.

If node *x* has no children, then $x.\text{left-child} = \text{NIL}$, and if node *x* is the rightmost child of its parent, then $x.\text{right-sibling} = \text{NIL}$.

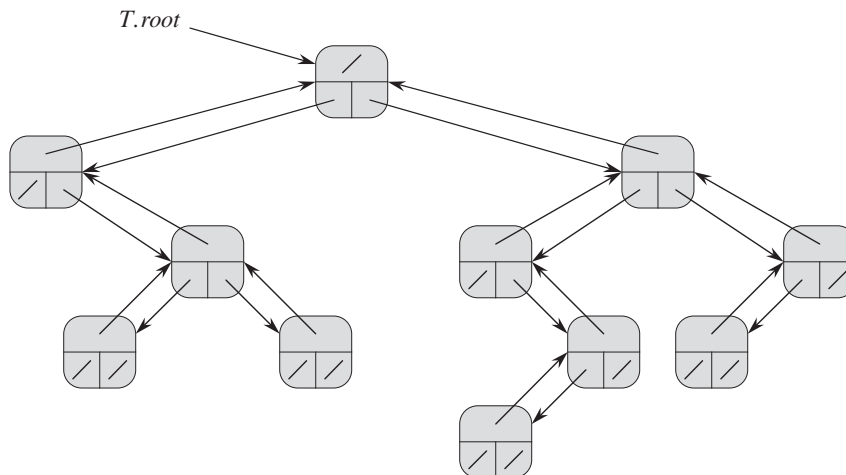
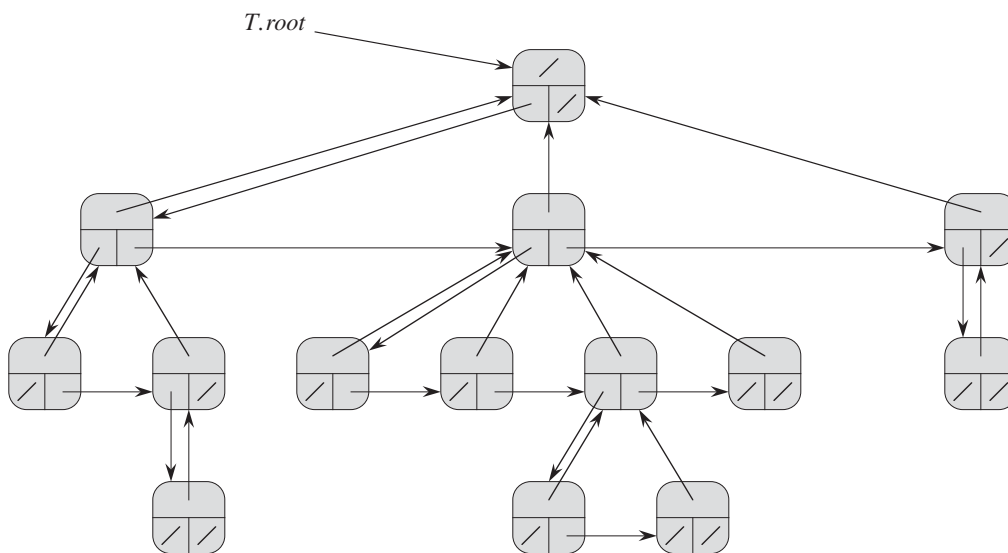


Figure 10.10 The left-child, right-sibling representation of a tree T . Each node x has attributes $x.p$ (top), $x.left-child$ (lower left), and $x.right-sibling$ (lower right). The *key* attributes are not shown.



Other tree representations

We sometimes represent rooted trees in other ways. In Chapter 6, for example, we represented a heap, which is based on a complete binary tree, by a single array plus the index of the last node in the heap. The trees that appear in Chapter 21 are traversed only toward the root, and so only the parent pointers are present; there are no pointers to children. Many other schemes are possible. Which scheme is best depends on the application.

Exercises

10.4-1

Draw the binary tree rooted at index 6 that is represented by the following attributes:

index	key	left	right
1	12	7	3
2	15	8	NIL
3	4	10	NIL
4	10	5	9
5	2	NIL	NIL
6	18	1	4
7	7	NIL	NIL
8	14	6	2
9	21	NIL	NIL
10	5	NIL	NIL

10.4-2

Write an $O(n)$ -time recursive procedure that, given an n -node binary tree, prints out the key of each node in the tree.

10.4-3

Write an $O(n)$ -time nonrecursive procedure that, given an n -node binary tree, prints out the key of each node in the tree. Use a stack as an auxiliary data structure.

10.4-4

Write an $O(n)$ -time procedure that prints all the keys of an arbitrary rooted tree with n nodes, where the tree is stored using the left-child, right-sibling representation.

10.4-5 ★

Write an $O(n)$ -time nonrecursive procedure that, given an n -node binary tree, prints out the key of each node. Use no more than constant extra space outside

of the tree itself and do not modify the tree, even temporarily, during the procedure.

10.4-6 ★

The left-child, right-sibling representation of an arbitrary rooted tree uses three pointers in each node: *left-child*, *right-sibling*, and *parent*. From any node, its parent can be reached and identified in constant time and all its children can be reached and identified in time linear in the number of children. Show how to use only two pointers and one boolean value in each node so that the parent of a node or all of its children can be reached and identified in time linear in the number of children.

Problems

10-1 Comparisons among lists

For each of the four types of lists in the following table, what is the asymptotic worst-case running time for each dynamic-set operation listed?

	unsorted, singly linked	sorted, singly linked	unsorted, doubly linked	sorted, doubly linked
SEARCH(L, k)				
INSERT(L, x)				
DELETE(L, x)				
SUCCESSOR(L, x)				
PREDECESSOR(L, x)				
MINIMUM(L)				
MAXIMUM(L)				

10-2 Mergeable heaps using linked lists

A *mergeable heap* supports the following operations: MAKE-HEAP (which creates an empty mergeable heap), INSERT, MINIMUM, EXTRACT-MIN, and UNION.¹ Show how to implement mergeable heaps using linked lists in each of the following cases. Try to make each operation as efficient as possible. Analyze the running time of each operation in terms of the size of the dynamic set(s) being operated on.

- a. Lists are sorted.
- b. Lists are unsorted.
- c. Lists are unsorted, and dynamic sets to be merged are disjoint.

10-3 Searching a sorted compact list

Exercise 10.3-4 asked how we might maintain an n -element list compactly in the first n positions of an array. We shall assume that all keys are distinct and that the compact list is also sorted, that is, $key[i] < key[next[i]]$ for all $i = 1, 2, \dots, n$ such that $next[i] \neq \text{NIL}$. We will also assume that we have a variable L that contains the index of the first element on the list. Under these assumptions, you will show that we can use the following randomized algorithm to search the list in $O(\sqrt{n})$ expected time.

COMPACT-LIST-SEARCH(L, n, k)

```

1   $i = L$ 
2  while  $i \neq \text{NIL}$  and  $key[i] < k$ 
3       $j = \text{RANDOM}(1, n)$ 
4      if  $key[i] < key[j]$  and  $key[j] \leq k$ 
5           $i = j$ 
6          if  $key[i] == k$ 
7              return  $i$ 
8       $i = next[i]$ 
9  if  $i == \text{NIL}$  or  $key[i] > k$ 
10     return NIL
11 else return  $i$ 
```

If we ignore lines 3–7 of the procedure, we have an ordinary algorithm for searching a sorted linked list, in which index i points to each position of the list in

¹Because we have defined a mergeable heap to support MINIMUM and EXTRACT-MIN, we can also refer to it as a *mergeable min-heap*. Alternatively, if it supported MAXIMUM and EXTRACT-MAX, it would be a *mergeable max-heap*.

turn. The search terminates once the index i “falls off” the end of the list or once $key[i] \geq k$. In the latter case, if $key[i] = k$, clearly we have found a key with the value k . If, however, $key[i] > k$, then we will never find a key with the value k , and so terminating the search was the right thing to do.

Lines 3–7 attempt to skip ahead to a randomly chosen position j . Such a skip benefits us if $key[j]$ is larger than $key[i]$ and no larger than k ; in such a case, j marks a position in the list that i would have to reach during an ordinary list search. Because the list is compact, we know that any choice of j between 1 and n indexes some object in the list rather than a slot on the free list.

Instead of analyzing the performance of COMPACT-LIST-SEARCH directly, we shall analyze a related algorithm, COMPACT-LIST-SEARCH', which executes two separate loops. This algorithm takes an additional parameter t which determines an upper bound on the number of iterations of the first loop.

COMPACT-LIST-SEARCH'(L, n, k, t)

```

1   $i = L$ 
2  for  $q = 1$  to  $t$ 
3       $j = \text{RANDOM}(1, n)$ 
4      if  $key[i] < key[j]$  and  $key[j] \leq k$ 
5           $i = j$ 
6          if  $key[i] == k$ 
7              return  $i$ 
8  while  $i \neq \text{NIL}$  and  $key[i] < k$ 
9       $i = \text{next}[i]$ 
10 if  $i == \text{NIL}$  or  $key[i] > k$ 
11     return  $\text{NIL}$ 
12 else return  $i$ 
```

To compare the execution of the algorithms COMPACT-LIST-SEARCH(L, n, k) and COMPACT-LIST-SEARCH'(L, n, k, t), assume that the sequence of integers returned by the calls of RANDOM($1, n$) is the same for both algorithms.

- a. Suppose that COMPACT-LIST-SEARCH(L, n, k) takes t iterations of the **while** loop of lines 2–8. Argue that COMPACT-LIST-SEARCH'(L, n, k, t) returns the same answer and that the total number of iterations of both the **for** and **while** loops within COMPACT-LIST-SEARCH' is at least t .

In the call COMPACT-LIST-SEARCH'(L, n, k, t), let X_t be the random variable that describes the distance in the linked list (that is, through the chain of *next* pointers) from position i to the desired key k after t iterations of the **for** loop of lines 2–7 have occurred.

- b.* Argue that the expected running time of $\text{COMPACT-LIST-SEARCH}'(L, n, k, t)$ is $O(t + E[X_t])$.
- c.* Show that $E[X_t] \leq \sum_{r=1}^n (1 - r/n)^t$. (*Hint:* Use equation (C.25).)
- d.* Show that $\sum_{r=0}^{n-1} r^t \leq n^{t+1}/(t + 1)$.
- e.* Prove that $E[X_t] \leq n/(t + 1)$.
- f.* Show that $\text{COMPACT-LIST-SEARCH}'(L, n, k, t)$ runs in $O(t + n/t)$ expected time.
- g.* Conclude that $\text{COMPACT-LIST-SEARCH}$ runs in $O(\sqrt{n})$ expected time.
- h.* Why do we assume that all keys are distinct in $\text{COMPACT-LIST-SEARCH}$? Argue that random skips do not necessarily help asymptotically when the list contains repeated key values.

Chapter notes

Aho, Hopcroft, and Ullman [6] and Knuth [209] are excellent references for elementary data structures. Many other texts cover both basic data structures and their implementation in a particular programming language. Examples of these types of textbooks include Goodrich and Tamassia [147], Main [241], Shaffer [311], and Weiss [352, 353, 354]. Gonnet [145] provides experimental data on the performance of many data-structure operations.

The origin of stacks and queues as data structures in computer science is unclear, since corresponding notions already existed in mathematics and paper-based business practices before the introduction of digital computers. Knuth [209] cites A. M. Turing for the development of stacks for subroutine linkage in 1947.

Pointer-based data structures also seem to be a folk invention. According to Knuth, pointers were apparently used in early computers with drum memories. The A-1 language developed by G. M. Hopper in 1951 represented algebraic formulas as binary trees. Knuth credits the IPL-II language, developed in 1956 by A. Newell, J. C. Shaw, and H. A. Simon, for recognizing the importance and promoting the use of pointers. Their IPL-III language, developed in 1957, included explicit stack operations.

Many applications require a dynamic set that supports only the dictionary operations INSERT, SEARCH, and DELETE. For example, a compiler that translates a programming language maintains a symbol table, in which the keys of elements are arbitrary character strings corresponding to identifiers in the language. A hash table is an effective data structure for implementing dictionaries. Although searching for an element in a hash table can take as long as searching for an element in a linked list— $\Theta(n)$ time in the worst case—in practice, hashing performs extremely well. Under reasonable assumptions, the average time to search for an element in a hash table is $O(1)$.

A hash table generalizes the simpler notion of an ordinary array. Directly addressing into an ordinary array makes effective use of our ability to examine an arbitrary position in an array in $O(1)$ time. Section 11.1 discusses direct addressing in more detail. We can take advantage of direct addressing when we can afford to allocate an array that has one position for every possible key.

When the number of keys actually stored is small relative to the total number of possible keys, hash tables become an effective alternative to directly addressing an array, since a hash table typically uses an array of size proportional to the number of keys actually stored. Instead of using the key as an array index directly, the array index is *computed* from the key. Section 11.2 presents the main ideas, focusing on “chaining” as a way to handle “collisions,” in which more than one key maps to the same array index. Section 11.3 describes how we can compute array indices from keys using hash functions. We present and analyze several variations on the basic theme. Section 11.4 looks at “open addressing,” which is another way to deal with collisions. The bottom line is that hashing is an extremely effective and practical technique: the basic dictionary operations require only $O(1)$ time on the average. Section 11.5 explains how “perfect hashing” can support searches in $O(1)$ *worst-case* time, when the set of keys being stored is static (that is, when the set of keys never changes once stored).

11.1 Direct-address tables

Direct addressing is a simple technique that works well when the universe U of keys is reasonably small. Suppose that an application needs a dynamic set in which each element has a key drawn from the universe $U = \{0, 1, \dots, m-1\}$, where m is not too large. We shall assume that no two elements have the same key.

To represent the dynamic set, we use an array, or *direct-address table*, denoted by $T[0..m-1]$, in which each position, or *slot*, corresponds to a key in the universe U . Figure 11.1 illustrates the approach; slot k points to an element in the set with key k . If the set contains no element with key k , then $T[k] = \text{NIL}$.

The dictionary operations are trivial to implement:

DIRECT-ADDRESS-SEARCH(T, k)

1 **return** $T[k]$

DIRECT-ADDRESS-INSERT(T, x)

1 $T[x.\text{key}] = x$

DIRECT-ADDRESS-DELETE(T, x)

1 $T[x.\text{key}] = \text{NIL}$

Each of these operations takes only $O(1)$ time.

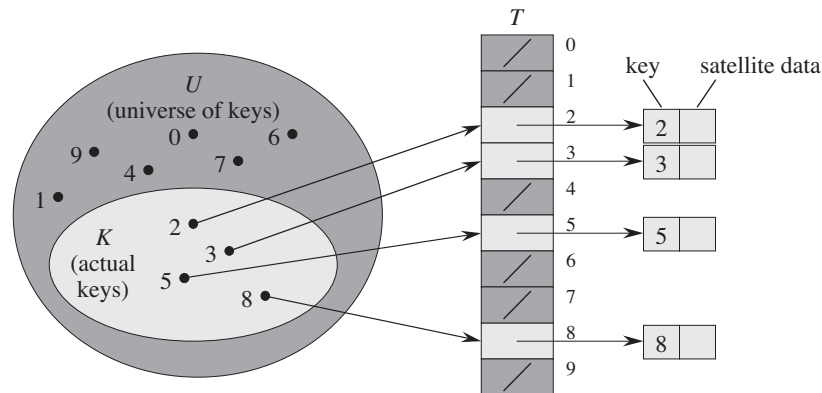


Figure 11.1 How to implement a dynamic set by a direct-address table T . Each key in the universe $U = \{0, 1, \dots, 9\}$ corresponds to an index in the table. The set $K = \{2, 3, 5, 8\}$ of actual keys determines the slots in the table that contain pointers to elements. The other slots, heavily shaded, contain NIL.

For some applications, the direct-address table itself can hold the elements in the dynamic set. That is, rather than storing an element's key and satellite data in an object external to the direct-address table, with a pointer from a slot in the table to the object, we can store the object in the slot itself, thus saving space. We would use a special key within an object to indicate an empty slot. Moreover, it is often unnecessary to store the key of the object, since if we have the index of an object in the table, we have its key. If keys are not stored, however, we must have some way to tell whether the slot is empty.

Exercises

11.1-1

Suppose that a dynamic set S is represented by a direct-address table T of length m . Describe a procedure that finds the maximum element of S . What is the worst-case performance of your procedure?

11.1-2

A **bit vector** is simply an array of bits (0s and 1s). A bit vector of length m takes much less space than an array of m pointers. Describe how to use a bit vector to represent a dynamic set of distinct elements with no satellite data. Dictionary operations should run in $O(1)$ time.

11.1-3

Suggest how to implement a direct-address table in which the keys of stored elements do not need to be distinct and the elements can have satellite data. All three dictionary operations (INSERT, DELETE, and SEARCH) should run in $O(1)$ time. (Don't forget that DELETE takes as an argument a pointer to an object to be deleted, not a key.)

11.1-4 ★

We wish to implement a dictionary by using direct addressing on a *huge* array. At the start, the array entries may contain garbage, and initializing the entire array is impractical because of its size. Describe a scheme for implementing a direct-address dictionary on a huge array. Each stored object should use $O(1)$ space; the operations SEARCH, INSERT, and DELETE should take $O(1)$ time each; and initializing the data structure should take $O(1)$ time. (*Hint:* Use an additional array, treated somewhat like a stack whose size is the number of keys actually stored in the dictionary, to help determine whether a given entry in the huge array is valid or not.)

11.2 Hash tables

The downside of direct addressing is obvious: if the universe U is large, storing a table T of size $|U|$ may be impractical, or even impossible, given the memory available on a typical computer. Furthermore, the set K of keys *actually stored* may be so small relative to U that most of the space allocated for T would be wasted.

When the set K of keys stored in a dictionary is much smaller than the universe U of all possible keys, a hash table requires much less storage than a direct-address table. Specifically, we can reduce the storage requirement to $\Theta(|K|)$ while we maintain the benefit that searching for an element in the hash table still requires only $O(1)$ time. The catch is that this bound is for the *average-case time*, whereas for direct addressing it holds for the *worst-case time*.

With direct addressing, an element with key k is stored in slot k . With hashing, this element is stored in slot $h(k)$; that is, we use a **hash function** h to compute the slot from the key k . Here, h maps the universe U of keys into the slots of a **hash table** $T[0..m-1]$:

$$h : U \rightarrow \{0, 1, \dots, m-1\} ,$$

where the size m of the hash table is typically much less than $|U|$. We say that an element with key k **hashes** to slot $h(k)$; we also say that $h(k)$ is the **hash value** of key k . Figure 11.2 illustrates the basic idea. The hash function reduces the range of array indices and hence the size of the array. Instead of a size of $|U|$, the array can have size m .

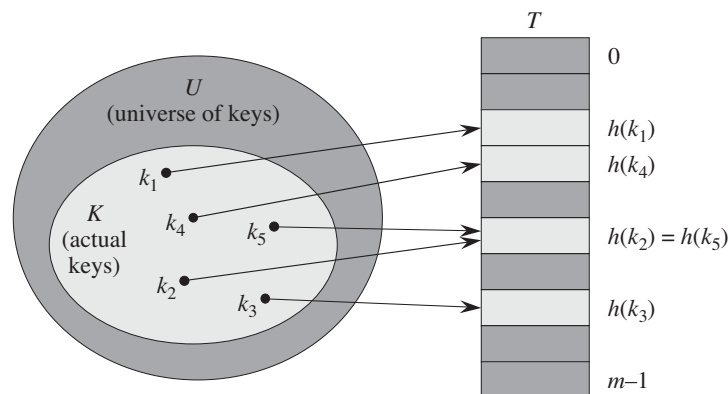


Figure 11.2 Using a hash function h to map keys to hash-table slots. Because keys k_2 and k_5 map to the same slot, they collide.

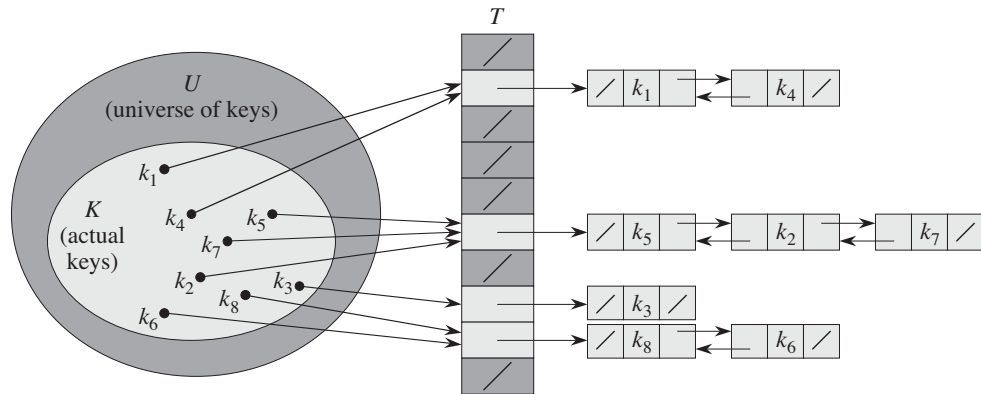


Figure 11.3 Collision resolution by chaining. Each hash-table slot $T[j]$ contains a linked list of all the keys whose hash value is j . For example, $h(k_1) = h(k_4)$ and $h(k_5) = h(k_7) = h(k_2)$. The linked list can be either singly or doubly linked; we show it as doubly linked because deletion is faster that way.

There is one hitch: two keys may hash to the same slot. We call this situation a **collision**. Fortunately, we have effective techniques for resolving the conflict created by collisions.

Of course, the ideal solution would be to avoid collisions altogether. We might try to achieve this goal by choosing a suitable hash function h . One idea is to make h appear to be “random,” thus avoiding collisions or at least minimizing their number. The very term “to hash,” evoking images of random mixing and chopping, captures the spirit of this approach. (Of course, a hash function h must be deterministic in that a given input k should always produce the same output $h(k)$.) Because $|U| > m$, however, there must be at least two keys that have the same hash value; avoiding collisions altogether is therefore impossible. Thus, while a well-designed, “random”-looking hash function can minimize the number of collisions, we still need a method for resolving the collisions that do occur.

The remainder of this section presents the simplest collision resolution technique, called chaining. Section 11.4 introduces an alternative method for resolving collisions, called open addressing.

Collision resolution by chaining

In **chaining**, we place all the elements that hash to the same slot into the same linked list, as Figure 11.3 shows. Slot j contains a pointer to the head of the list of all stored elements that hash to j ; if there are no such elements, slot j contains NIL.

The dictionary operations on a hash table T are easy to implement when collisions are resolved by chaining:

CHAINED-HASH-INSERT(T, x)

1 insert x at the head of list $T[h(x.key)]$

CHAINED-HASH-SEARCH(T, k)

1 search for an element with key k in list $T[h(k)]$

CHAINED-HASH-DELETE(T, x)

1 delete x from the list $T[h(x.key)]$

The worst-case running time for insertion is $O(1)$. The insertion procedure is fast in part because it assumes that the element x being inserted is not already present in the table; if necessary, we can check this assumption (at additional cost) by searching for an element whose key is $x.key$ before we insert. For searching, the worst-case running time is proportional to the length of the list; we shall analyze this operation more closely below. We can delete an element in $O(1)$ time if the lists are doubly linked, as Figure 11.3 depicts. (Note that CHAINED-HASH-DELETE takes as input an element x and not its key k , so that we don't have to search for x first. If the hash table supports deletion, then its linked lists should be doubly linked so that we can delete an item quickly. If the lists were only singly linked, then to delete element x , we would first have to find x in the list $T[h(x.key)]$ so that we could update the *next* attribute of x 's predecessor. With singly linked lists, both deletion and searching would have the same asymptotic running times.)

Analysis of hashing with chaining

How well does hashing with chaining perform? In particular, how long does it take to search for an element with a given key?

Given a hash table T with m slots that stores n elements, we define the **load factor** α for T as n/m , that is, the average number of elements stored in a chain. Our analysis will be in terms of α , which can be less than, equal to, or greater than 1.

The worst-case behavior of hashing with chaining is terrible: all n keys hash to the same slot, creating a list of length n . The worst-case time for searching is thus $\Theta(n)$ plus the time to compute the hash function—no better than if we used one linked list for all the elements. Clearly, we do not use hash tables for their worst-case performance. (Perfect hashing, described in Section 11.5, does provide good worst-case performance when the set of keys is static, however.)

The average-case performance of hashing depends on how well the hash function h distributes the set of keys to be stored among the m slots, on the average.

Section 11.3 discusses these issues, but for now we shall assume that any given element is equally likely to hash into any of the m slots, independently of where any other element has hashed to. We call this the assumption of **simple uniform hashing**.

For $j = 0, 1, \dots, m-1$, let us denote the length of the list $T[j]$ by n_j , so that

$$n = n_0 + n_1 + \dots + n_{m-1}, \quad (11.1)$$

and the expected value of n_j is $E[n_j] = \alpha = n/m$.

We assume that $O(1)$ time suffices to compute the hash value $h(k)$, so that the time required to search for an element with key k depends linearly on the length $n_{h(k)}$ of the list $T[h(k)]$. Setting aside the $O(1)$ time required to compute the hash function and to access slot $h(k)$, let us consider the expected number of elements examined by the search algorithm, that is, the number of elements in the list $T[h(k)]$ that the algorithm checks to see whether any have a key equal to k . We shall consider two cases. In the first, the search is unsuccessful: no element in the table has key k . In the second, the search successfully finds an element with key k .

Theorem 11.1

In a hash table in which collisions are resolved by chaining, an unsuccessful search takes average-case time $\Theta(1 + \alpha)$, under the assumption of simple uniform hashing.

Proof Under the assumption of simple uniform hashing, any key k not already stored in the table is equally likely to hash to any of the m slots. The expected time to search unsuccessfully for a key k is the expected time to search to the end of list $T[h(k)]$, which has expected length $E[n_{h(k)}] = \alpha$. Thus, the expected number of elements examined in an unsuccessful search is α , and the total time required (including the time for computing $h(k)$) is $\Theta(1 + \alpha)$. ■

The situation for a successful search is slightly different, since each list is not equally likely to be searched. Instead, the probability that a list is searched is proportional to the number of elements it contains. Nonetheless, the expected search time still turns out to be $\Theta(1 + \alpha)$.

Theorem 11.2

In a hash table in which collisions are resolved by chaining, a successful search takes average-case time $\Theta(1 + \alpha)$, under the assumption of simple uniform hashing.

Proof We assume that the element being searched for is equally likely to be any of the n elements stored in the table. The number of elements examined during a successful search for an element x is one more than the number of elements that

appear before x in x 's list. Because new elements are placed at the front of the list, elements before x in the list were all inserted after x was inserted. To find the expected number of elements examined, we take the average, over the n elements x in the table, of 1 plus the expected number of elements added to x 's list after x was added to the list. Let x_i denote the i th element inserted into the table, for $i = 1, 2, \dots, n$, and let $k_i = x_i.\text{key}$. For keys k_i and k_j , we define the indicator random variable $X_{ij} = \mathbf{I}\{h(k_i) = h(k_j)\}$. Under the assumption of simple uniform hashing, we have $\Pr\{h(k_i) = h(k_j)\} = 1/m$, and so by Lemma 5.1, $E[X_{ij}] = 1/m$. Thus, the expected number of elements examined in a successful search is

$$\begin{aligned}
 E\left[\frac{1}{n} \sum_{i=1}^n \left(1 + \sum_{j=i+1}^n X_{ij}\right)\right] &= \frac{1}{n} \sum_{i=1}^n \left(1 + \sum_{j=i+1}^n E[X_{ij}]\right) \quad (\text{by linearity of expectation}) \\
 &= \frac{1}{n} \sum_{i=1}^n \left(1 + \sum_{j=i+1}^n \frac{1}{m}\right) \\
 &= 1 + \frac{1}{nm} \sum_{i=1}^n (n-i) \\
 &= 1 + \frac{1}{nm} \left(\sum_{i=1}^n n - \sum_{i=1}^n i\right) \\
 &= 1 + \frac{1}{nm} \left(n^2 - \frac{n(n+1)}{2}\right) \quad (\text{by equation (A.1)}) \\
 &= 1 + \frac{n-1}{2m} \\
 &= 1 + \frac{\alpha}{2} - \frac{\alpha}{2n}.
 \end{aligned}$$

Thus, the total time required for a successful search (including the time for computing the hash function) is $\Theta(2 + \alpha/2 - \alpha/2n) = \Theta(1 + \alpha)$. ■

What does this analysis mean? If the number of hash-table slots is at least proportional to the number of elements in the table, we have $n = O(m)$ and, consequently, $\alpha = n/m = O(m)/m = O(1)$. Thus, searching takes constant time on average. Since insertion takes $O(1)$ worst-case time and deletion takes $O(1)$ worst-case time when the lists are doubly linked, we can support all dictionary operations in $O(1)$ time on average.

Exercises

11.2-1

Suppose we use a hash function h to hash n distinct keys into an array T of length m . Assuming simple uniform hashing, what is the expected number of collisions? More precisely, what is the expected cardinality of $\{\{k, l\} : k \neq l \text{ and } h(k) = h(l)\}$?

11.2-2

Demonstrate what happens when we insert the keys 5, 28, 19, 15, 20, 33, 12, 17, 10 into a hash table with collisions resolved by chaining. Let the table have 9 slots, and let the hash function be $h(k) = k \bmod 9$.

11.2-3

Professor Marley hypothesizes that he can obtain substantial performance gains by modifying the chaining scheme to keep each list in sorted order. How does the professor's modification affect the running time for successful searches, unsuccessful searches, insertions, and deletions?

11.2-4

Suggest how to allocate and deallocate storage for elements within the hash table itself by linking all unused slots into a free list. Assume that one slot can store a flag and either one element plus a pointer or two pointers. All dictionary and free-list operations should run in $O(1)$ expected time. Does the free list need to be doubly linked, or does a singly linked free list suffice?

11.2-5

Suppose that we are storing a set of n keys into a hash table of size m . Show that if the keys are drawn from a universe U with $|U| > nm$, then U has a subset of size n consisting of keys that all hash to the same slot, so that the worst-case searching time for hashing with chaining is $\Theta(n)$.

11.2-6

Suppose we have stored n keys in a hash table of size m , with collisions resolved by chaining, and that we know the length of each chain, including the length L of the longest chain. Describe a procedure that selects a key uniformly at random from among the keys in the hash table and returns it in expected time $O(L \cdot (1 + 1/\alpha))$.

11.3 Hash functions

In this section, we discuss some issues regarding the design of good hash functions and then present three schemes for their creation. Two of the schemes, hashing by division and hashing by multiplication, are heuristic in nature, whereas the third scheme, universal hashing, uses randomization to provide provably good performance.

What makes a good hash function?

A good hash function satisfies (approximately) the assumption of simple uniform hashing: each key is equally likely to hash to any of the m slots, independently of where any other key has hashed to. Unfortunately, we typically have no way to check this condition, since we rarely know the probability distribution from which the keys are drawn. Moreover, the keys might not be drawn independently.

Occasionally we do know the distribution. For example, if we know that the keys are random real numbers k independently and uniformly distributed in the range $0 \leq k < 1$, then the hash function

$$h(k) = \lfloor km \rfloor$$

satisfies the condition of simple uniform hashing.

In practice, we can often employ heuristic techniques to create a hash function that performs well. Qualitative information about the distribution of keys may be useful in this design process. For example, consider a compiler's symbol table, in which the keys are character strings representing identifiers in a program. Closely related symbols, such as `pt` and `pts`, often occur in the same program. A good hash function would minimize the chance that such variants hash to the same slot.

A good approach derives the hash value in a way that we expect to be independent of any patterns that might exist in the data. For example, the “division method” (discussed in Section 11.3.1) computes the hash value as the remainder when the key is divided by a specified prime number. This method frequently gives good results, assuming that we choose a prime number that is unrelated to any patterns in the distribution of keys.

Finally, we note that some applications of hash functions might require stronger properties than are provided by simple uniform hashing. For example, we might want keys that are “close” in some sense to yield hash values that are far apart. (This property is especially desirable when we are using linear probing, defined in Section 11.4.) Universal hashing, described in Section 11.3.3, often provides the desired properties.

Interpreting keys as natural numbers

Most hash functions assume that the universe of keys is the set $\mathbb{N} = \{0, 1, 2, \dots\}$ of natural numbers. Thus, if the keys are not natural numbers, we find a way to interpret them as natural numbers. For example, we can interpret a character string as an integer expressed in suitable radix notation. Thus, we might interpret the identifier `pt` as the pair of decimal integers (112, 116), since `p` = 112 and `t` = 116 in the ASCII character set; then, expressed as a radix-128 integer, `pt` becomes $(112 \cdot 128) + 116 = 14452$. In the context of a given application, we can usually devise some such method for interpreting each key as a (possibly large) natural number. In what follows, we assume that the keys are natural numbers.

11.3.1 The division method

In the *division method* for creating hash functions, we map a key k into one of m slots by taking the remainder of k divided by m . That is, the hash function is

$$h(k) = k \bmod m .$$

For example, if the hash table has size $m = 12$ and the key is $k = 100$, then $h(k) = 4$. Since it requires only a single division operation, hashing by division is quite fast.

When using the division method, we usually avoid certain values of m . For example, m should not be a power of 2, since if $m = 2^p$, then $h(k)$ is just the p lowest-order bits of k . Unless we know that all low-order p -bit patterns are equally likely, we are better off designing the hash function to depend on all the bits of the key. As Exercise 11.3-3 asks you to show, choosing $m = 2^p - 1$ when k is a character string interpreted in radix 2^p may be a poor choice, because permuting the characters of k does not change its hash value.

A prime not too close to an exact power of 2 is often a good choice for m . For example, suppose we wish to allocate a hash table, with collisions resolved by chaining, to hold roughly $n = 2000$ character strings, where a character has 8 bits. We don't mind examining an average of 3 elements in an unsuccessful search, and so we allocate a hash table of size $m = 701$. We could choose $m = 701$ because it is a prime near $2000/3$ but not near any power of 2. Treating each key k as an integer, our hash function would be

$$h(k) = k \bmod 701 .$$

11.3.2 The multiplication method

The *multiplication method* for creating hash functions operates in two steps. First, we multiply the key k by a constant A in the range $0 < A < 1$ and extract the

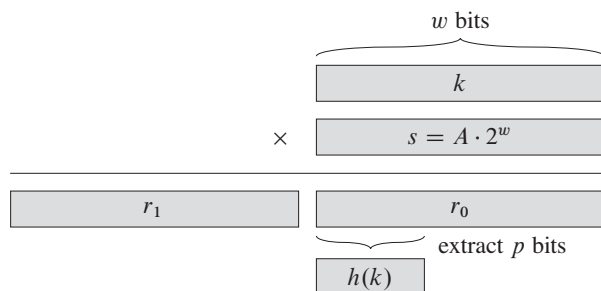


Figure 11.4 The multiplication method of hashing. The w -bit representation of the key k is multiplied by the w -bit value $s = A \cdot 2^w$. The p highest-order bits of the lower w -bit half of the product form the desired hash value $h(k)$.

fractional part of kA . Then, we multiply this value by m and take the floor of the result. In short, the hash function is

$$h(k) = \lfloor m (kA \bmod 1) \rfloor ,$$

where “ $kA \bmod 1$ ” means the fractional part of kA , that is, $kA - \lfloor kA \rfloor$.

An advantage of the multiplication method is that the value of m is not critical. We typically choose it to be a power of 2 ($m = 2^p$ for some integer p), since we can then easily implement the function on most computers as follows. Suppose that the word size of the machine is w bits and that k fits into a single word. We restrict A to be a fraction of the form $s/2^w$, where s is an integer in the range $0 < s < 2^w$. Referring to Figure 11.4, we first multiply k by the w -bit integer $s = A \cdot 2^w$. The result is a $2w$ -bit value $r_1 2^w + r_0$, where r_1 is the high-order word of the product and r_0 is the low-order word of the product. The desired p -bit hash value consists of the p most significant bits of r_0 .

Although this method works with any value of the constant A , it works better with some values than with others. The optimal choice depends on the characteristics of the data being hashed. Knuth [211] suggests that

$$A \approx (\sqrt{5} - 1)/2 = 0.6180339887 \dots \quad (11.2)$$

is likely to work reasonably well.

As an example, suppose we have $k = 123456$, $p = 14$, $m = 2^{14} = 16384$, and $w = 32$. Adapting Knuth’s suggestion, we choose A to be the fraction of the form $s/2^{32}$ that is closest to $(\sqrt{5} - 1)/2$, so that $A = 2654435769/2^{32}$. Then $k \cdot s = 327706022297664 = (76300 \cdot 2^{32}) + 17612864$, and so $r_1 = 76300$ and $r_0 = 17612864$. The 14 most significant bits of r_0 yield the value $h(k) = 67$.

★ 11.3.3 Universal hashing

If a malicious adversary chooses the keys to be hashed by some fixed hash function, then the adversary can choose n keys that all hash to the same slot, yielding an average retrieval time of $\Theta(n)$. Any fixed hash function is vulnerable to such terrible worst-case behavior; the only effective way to improve the situation is to choose the hash function *randomly* in a way that is *independent* of the keys that are actually going to be stored. This approach, called **universal hashing**, can yield provably good performance on average, no matter which keys the adversary chooses.

In universal hashing, at the beginning of execution we select the hash function at random from a carefully designed class of functions. As in the case of quicksort, randomization guarantees that no single input will always evoke worst-case behavior. Because we randomly select the hash function, the algorithm can behave differently on each execution, even for the same input, guaranteeing good average-case performance for any input. Returning to the example of a compiler's symbol table, we find that the programmer's choice of identifiers cannot now cause consistently poor hashing performance. Poor performance occurs only when the compiler chooses a random hash function that causes the set of identifiers to hash poorly, but the probability of this situation occurring is small and is the same for any set of identifiers of the same size.

Let \mathcal{H} be a finite collection of hash functions that map a given universe U of keys into the range $\{0, 1, \dots, m-1\}$. Such a collection is said to be **universal** if for each pair of distinct keys $k, l \in U$, the number of hash functions $h \in \mathcal{H}$ for which $h(k) = h(l)$ is at most $|\mathcal{H}|/m$. In other words, with a hash function randomly chosen from \mathcal{H} , the chance of a collision between distinct keys k and l is no more than the chance $1/m$ of a collision if $h(k)$ and $h(l)$ were randomly and independently chosen from the set $\{0, 1, \dots, m-1\}$.

The following theorem shows that a universal class of hash functions gives good average-case behavior. Recall that n_i denotes the length of list $T[i]$.

Theorem 11.3

Suppose that a hash function h is chosen randomly from a universal collection of hash functions and has been used to hash n keys into a table T of size m , using chaining to resolve collisions. If key k is not in the table, then the expected length $E[n_{h(k)}]$ of the list that key k hashes to is at most the load factor $\alpha = n/m$. If key k is in the table, then the expected length $E[n_{h(k)}]$ of the list containing key k is at most $1 + \alpha$.

Proof We note that the expectations here are over the choice of the hash function and do not depend on any assumptions about the distribution of the keys. For each pair k and l of distinct keys, define the indicator random variable

$X_{kl} = \mathbb{I}\{h(k) = h(l)\}$. Since by the definition of a universal collection of hash functions, a single pair of keys collides with probability at most $1/m$, we have $\Pr\{h(k) = h(l)\} \leq 1/m$. By Lemma 5.1, therefore, we have $\mathbb{E}[X_{kl}] \leq 1/m$.

Next we define, for each key k , the random variable Y_k that equals the number of keys other than k that hash to the same slot as k , so that

$$Y_k = \sum_{\substack{l \in T \\ l \neq k}} X_{kl}.$$

Thus we have

$$\begin{aligned} \mathbb{E}[Y_k] &= \mathbb{E}\left[\sum_{\substack{l \in T \\ l \neq k}} X_{kl}\right] \\ &= \sum_{\substack{l \in T \\ l \neq k}} \mathbb{E}[X_{kl}] \quad (\text{by linearity of expectation}) \\ &\leq \sum_{\substack{l \in T \\ l \neq k}} \frac{1}{m}. \end{aligned}$$

The remainder of the proof depends on whether key k is in table T .

- If $k \notin T$, then $n_{h(k)} = Y_k$ and $|\{l : l \in T \text{ and } l \neq k\}| = n$. Thus $\mathbb{E}[n_{h(k)}] = \mathbb{E}[Y_k] \leq n/m = \alpha$.
- If $k \in T$, then because key k appears in list $T[h(k)]$ and the count Y_k does not include key k , we have $n_{h(k)} = Y_k + 1$ and $|\{l : l \in T \text{ and } l \neq k\}| = n - 1$. Thus $\mathbb{E}[n_{h(k)}] = \mathbb{E}[Y_k] + 1 \leq (n - 1)/m + 1 = 1 + \alpha - 1/m < 1 + \alpha$. ■

The following corollary says universal hashing provides the desired payoff: it has now become impossible for an adversary to pick a sequence of operations that forces the worst-case running time. By cleverly randomizing the choice of hash function at run time, we guarantee that we can process every sequence of operations with a good average-case running time.

Corollary 11.4

Using universal hashing and collision resolution by chaining in an initially empty table with m slots, it takes expected time $\Theta(n)$ to handle any sequence of n INSERT, SEARCH, and DELETE operations containing $O(m)$ INSERT operations.

Proof Since the number of insertions is $O(m)$, we have $n = O(m)$ and so $\alpha = O(1)$. The INSERT and DELETE operations take constant time and, by Theorem 11.3, the expected time for each SEARCH operation is $O(1)$. By linearity of

expectation, therefore, the expected time for the entire sequence of n operations is $O(n)$. Since each operation takes $\Omega(1)$ time, the $\Theta(n)$ bound follows. ■

Designing a universal class of hash functions

It is quite easy to design a universal class of hash functions, as a little number theory will help us prove. You may wish to consult Chapter 31 first if you are unfamiliar with number theory.

We begin by choosing a prime number p large enough so that every possible key k is in the range 0 to $p - 1$, inclusive. Let \mathbb{Z}_p denote the set $\{0, 1, \dots, p - 1\}$, and let \mathbb{Z}_p^* denote the set $\{1, 2, \dots, p - 1\}$. Since p is prime, we can solve equations modulo p with the methods given in Chapter 31. Because we assume that the size of the universe of keys is greater than the number of slots in the hash table, we have $p > m$.

We now define the hash function h_{ab} for any $a \in \mathbb{Z}_p^*$ and any $b \in \mathbb{Z}_p$ using a linear transformation followed by reductions modulo p and then modulo m :

$$h_{ab}(k) = ((ak + b) \bmod p) \bmod m. \quad (11.3)$$

For example, with $p = 17$ and $m = 6$, we have $h_{3,4}(8) = 5$. The family of all such hash functions is

$$\mathcal{H}_{pm} = \{h_{ab} : a \in \mathbb{Z}_p^* \text{ and } b \in \mathbb{Z}_p\}. \quad (11.4)$$

Each hash function h_{ab} maps \mathbb{Z}_p to \mathbb{Z}_m . This class of hash functions has the nice property that the size m of the output range is arbitrary—not necessarily prime—a feature which we shall use in Section 11.5. Since we have $p - 1$ choices for a and p choices for b , the collection \mathcal{H}_{pm} contains $p(p - 1)$ hash functions.

Theorem 11.5

The class \mathcal{H}_{pm} of hash functions defined by equations (11.3) and (11.4) is universal.

Proof Consider two distinct keys k and l from \mathbb{Z}_p , so that $k \neq l$. For a given hash function h_{ab} we let

$$r = (ak + b) \bmod p,$$

$$s = (al + b) \bmod p.$$

We first note that $r \neq s$. Why? Observe that

$$r - s \equiv a(k - l) \pmod{p}.$$

It follows that $r \neq s$ because p is prime and both a and $(k - l)$ are nonzero modulo p , and so their product must also be nonzero modulo p by Theorem 31.6. Therefore, when computing any $h_{ab} \in \mathcal{H}_{pm}$, distinct inputs k and l map to distinct

values r and s modulo p ; there are no collisions yet at the “mod p level.” Moreover, each of the possible $p(p-1)$ choices for the pair (a, b) with $a \neq 0$ yields a *different* resulting pair (r, s) with $r \neq s$, since we can solve for a and b given r and s :

$$\begin{aligned} a &= ((r - s)((k - l)^{-1} \bmod p)) \bmod p, \\ b &= (r - ak) \bmod p, \end{aligned}$$

where $((k - l)^{-1} \bmod p)$ denotes the unique multiplicative inverse, modulo p , of $k - l$. Since there are only $p(p - 1)$ possible pairs (r, s) with $r \neq s$, there is a one-to-one correspondence between pairs (a, b) with $a \neq 0$ and pairs (r, s) with $r \neq s$. Thus, for any given pair of inputs k and l , if we pick (a, b) uniformly at random from $\mathbb{Z}_p^* \times \mathbb{Z}_p$, the resulting pair (r, s) is equally likely to be any pair of distinct values modulo p .

Therefore, the probability that distinct keys k and l collide is equal to the probability that $r \equiv s \pmod{m}$ when r and s are randomly chosen as distinct values modulo p . For a given value of r , of the $p - 1$ possible remaining values for s , the number of values s such that $s \neq r$ and $s \equiv r \pmod{m}$ is at most

$$\begin{aligned} \lceil p/m \rceil - 1 &\leq ((p + m - 1)/m) - 1 \quad (\text{by inequality (3.6)}) \\ &= (p - 1)/m. \end{aligned}$$

The probability that s collides with r when reduced modulo m is at most $((p - 1)/m)/(p - 1) = 1/m$.

Therefore, for any pair of distinct values $k, l \in \mathbb{Z}_p$,

$$\Pr \{h_{ab}(k) = h_{ab}(l)\} \leq 1/m,$$

so that \mathcal{H}_{pm} is indeed universal. ■

Exercises

11.3-1

Suppose we wish to search a linked list of length n , where each element contains a key k along with a hash value $h(k)$. Each key is a long character string. How might we take advantage of the hash values when searching the list for an element with a given key?

11.3-2

Suppose that we hash a string of r characters into m slots by treating it as a radix-128 number and then using the division method. We can easily represent the number m as a 32-bit computer word, but the string of r characters, treated as a radix-128 number, takes many words. How can we apply the division method to compute the hash value of the character string without using more than a constant number of words of storage outside the string itself?

11.3-3

Consider a version of the division method in which $h(k) = k \bmod m$, where $m = 2^p - 1$ and k is a character string interpreted in radix 2^p . Show that if we can derive string x from string y by permuting its characters, then x and y hash to the same value. Give an example of an application in which this property would be undesirable in a hash function.

11.3-4

Consider a hash table of size $m = 1000$ and a corresponding hash function $h(k) = \lfloor m(kA \bmod 1) \rfloor$ for $A = (\sqrt{5} - 1)/2$. Compute the locations to which the keys 61, 62, 63, 64, and 65 are mapped.

11.3-5 ★

Define a family \mathcal{H} of hash functions from a finite set U to a finite set B to be **ϵ -universal** if for all pairs of distinct elements k and l in U ,

$$\Pr \{h(k) = h(l)\} \leq \epsilon ,$$

where the probability is over the choice of the hash function h drawn at random from the family \mathcal{H} . Show that an ϵ -universal family of hash functions must have

$$\epsilon \geq \frac{1}{|B|} - \frac{1}{|U|} .$$

11.3-6 ★

Let U be the set of n -tuples of values drawn from \mathbb{Z}_p , and let $B = \mathbb{Z}_p$, where p is prime. Define the hash function $h_b : U \rightarrow B$ for $b \in \mathbb{Z}_p$ on an input n -tuple $\langle a_0, a_1, \dots, a_{n-1} \rangle$ from U as

$$h_b(\langle a_0, a_1, \dots, a_{n-1} \rangle) = \left(\sum_{j=0}^{n-1} a_j b^j \right) \bmod p ,$$

and let $\mathcal{H} = \{h_b : b \in \mathbb{Z}_p\}$. Argue that \mathcal{H} is $((n-1)/p)$ -universal according to the definition of ϵ -universal in Exercise 11.3-5. (*Hint*: See Exercise 31.4-4.)

11.4 Open addressing

In **open addressing**, all elements occupy the hash table itself. That is, each table entry contains either an element of the dynamic set or NIL. When searching for an element, we systematically examine table slots until either we find the desired element or we have ascertained that the element is not in the table. No lists and

no elements are stored outside the table, unlike in chaining. Thus, in open addressing, the hash table can “fill up” so that no further insertions can be made; one consequence is that the load factor α can never exceed 1.

Of course, we could store the linked lists for chaining inside the hash table, in the otherwise unused hash-table slots (see Exercise 11.2-4), but the advantage of open addressing is that it avoids pointers altogether. Instead of following pointers, we *compute* the sequence of slots to be examined. The extra memory freed by not storing pointers provides the hash table with a larger number of slots for the same amount of memory, potentially yielding fewer collisions and faster retrieval.

To perform insertion using open addressing, we successively examine, or **probe**, the hash table until we find an empty slot in which to put the key. Instead of being fixed in the order $0, 1, \dots, m-1$ (which requires $\Theta(n)$ search time), the sequence of positions probed *depends upon the key being inserted*. To determine which slots to probe, we extend the hash function to include the probe number (starting from 0) as a second input. Thus, the hash function becomes

$$h : U \times \{0, 1, \dots, m-1\} \rightarrow \{0, 1, \dots, m-1\} .$$

With open addressing, we require that for every key k , the **probe sequence**

$$\langle h(k, 0), h(k, 1), \dots, h(k, m-1) \rangle$$

be a permutation of $\{0, 1, \dots, m-1\}$, so that every hash-table position is eventually considered as a slot for a new key as the table fills up. In the following pseudocode, we assume that the elements in the hash table T are keys with no satellite information; the key k is identical to the element containing key k . Each slot contains either a key or NIL (if the slot is empty). The HASH-INSERT procedure takes as input a hash table T and a key k . It either returns the slot number where it stores key k or flags an error because the hash table is already full.

HASH-INSERT(T, k)

```

1   $i = 0$ 
2  repeat
3       $j = h(k, i)$ 
4      if  $T[j] == \text{NIL}$ 
5           $T[j] = k$ 
6          return  $j$ 
7      else  $i = i + 1$ 
8  until  $i == m$ 
9  error “hash table overflow”
```

The algorithm for searching for key k probes the same sequence of slots that the insertion algorithm examined when key k was inserted. Therefore, the search can

terminate (unsuccessfully) when it finds an empty slot, since k would have been inserted there and not later in its probe sequence. (This argument assumes that keys are not deleted from the hash table.) The procedure **HASH-SEARCH** takes as input a hash table T and a key k , returning j if it finds that slot j contains key k , or **NIL** if key k is not present in table T .

HASH-SEARCH(T, k)

```

1   $i = 0$ 
2  repeat
3       $j = h(k, i)$ 
4      if  $T[j] == k$ 
5          return  $j$ 
6       $i = i + 1$ 
7  until  $T[j] == \text{NIL}$  or  $i == m$ 
8  return NIL
```

Deletion from an open-address hash table is difficult. When we delete a key from slot i , we cannot simply mark that slot as empty by storing **NIL** in it. If we did, we might be unable to retrieve any key k during whose insertion we had probed slot i and found it occupied. We can solve this problem by marking the slot, storing in it the special value **DELETED** instead of **NIL**. We would then modify the procedure **HASH-INSERT** to treat such a slot as if it were empty so that we can insert a new key there. We do not need to modify **HASH-SEARCH**, since it will pass over **DELETED** values while searching. When we use the special value **DELETED**, however, search times no longer depend on the load factor α , and for this reason chaining is more commonly selected as a collision resolution technique when keys must be deleted.

In our analysis, we assume *uniform hashing*: the probe sequence of each key is equally likely to be any of the $m!$ permutations of $\{0, 1, \dots, m - 1\}$. Uniform hashing generalizes the notion of simple uniform hashing defined earlier to a hash function that produces not just a single number, but a whole probe sequence. True uniform hashing is difficult to implement, however, and in practice suitable approximations (such as double hashing, defined below) are used.

We will examine three commonly used techniques to compute the probe sequences required for open addressing: linear probing, quadratic probing, and double hashing. These techniques all guarantee that $\langle h(k, 0), h(k, 1), \dots, h(k, m - 1) \rangle$ is a permutation of $\{0, 1, \dots, m - 1\}$ for each key k . None of these techniques fulfills the assumption of uniform hashing, however, since none of them is capable of generating more than m^2 different probe sequences (instead of the $m!$ that uniform hashing requires). Double hashing has the greatest number of probe sequences and, as one might expect, seems to give the best results.

Linear probing

Given an ordinary hash function $h' : U \rightarrow \{0, 1, \dots, m-1\}$, which we refer to as an **auxiliary hash function**, the method of **linear probing** uses the hash function

$$h(k, i) = (h'(k) + i) \bmod m$$

for $i = 0, 1, \dots, m-1$. Given key k , we first probe $T[h'(k)]$, i.e., the slot given by the auxiliary hash function. We next probe slot $T[h'(k) + 1]$, and so on up to slot $T[m-1]$. Then we wrap around to slots $T[0], T[1], \dots$ until we finally probe slot $T[h'(k) - 1]$. Because the initial probe determines the entire probe sequence, there are only m distinct probe sequences.

Linear probing is easy to implement, but it suffers from a problem known as **primary clustering**. Long runs of occupied slots build up, increasing the average search time. Clusters arise because an empty slot preceded by i full slots gets filled next with probability $(i+1)/m$. Long runs of occupied slots tend to get longer, and the average search time increases.

Quadratic probing

Quadratic probing uses a hash function of the form

$$h(k, i) = (h'(k) + c_1i + c_2i^2) \bmod m, \quad (11.5)$$

where h' is an auxiliary hash function, c_1 and c_2 are positive auxiliary constants, and $i = 0, 1, \dots, m-1$. The initial position probed is $T[h'(k)]$; later positions probed are offset by amounts that depend in a quadratic manner on the probe number i . This method works much better than linear probing, but to make full use of the hash table, the values of c_1 , c_2 , and m are constrained. Problem 11-3 shows one way to select these parameters. Also, if two keys have the same initial probe position, then their probe sequences are the same, since $h(k_1, 0) = h(k_2, 0)$ implies $h(k_1, i) = h(k_2, i)$. This property leads to a milder form of clustering, called **secondary clustering**. As in linear probing, the initial probe determines the entire sequence, and so only m distinct probe sequences are used.

Double hashing

Double hashing offers one of the best methods available for open addressing because the permutations produced have many of the characteristics of randomly chosen permutations. **Double hashing** uses a hash function of the form

$$h(k, i) = (h_1(k) + ih_2(k)) \bmod m,$$

where both h_1 and h_2 are auxiliary hash functions. The initial probe goes to position $T[h_1(k)]$; successive probe positions are offset from previous positions by the

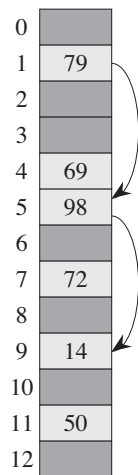


Figure 11.5 Insertion by double hashing. Here we have a hash table of size 13 with $h_1(k) = k \bmod 13$ and $h_2(k) = 1 + (k \bmod 11)$. Since $14 \equiv 1 \pmod{13}$ and $14 \equiv 3 \pmod{11}$, we insert the key 14 into empty slot 9, after examining slots 1 and 5 and finding them to be occupied.

amount $h_2(k)$, modulo m . Thus, unlike the case of linear or quadratic probing, the probe sequence here depends in two ways upon the key k , since the initial probe position, the offset, or both, may vary. Figure 11.5 gives an example of insertion by double hashing.

The value $h_2(k)$ must be relatively prime to the hash-table size m for the entire hash table to be searched. (See Exercise 11.4-4.) A convenient way to ensure this condition is to let m be a power of 2 and to design h_2 so that it always produces an odd number. Another way is to let m be prime and to design h_2 so that it always returns a positive integer less than m . For example, we could choose m prime and let

$$\begin{aligned} h_1(k) &= k \bmod m, \\ h_2(k) &= 1 + (k \bmod m'), \end{aligned}$$

where m' is chosen to be slightly less than m (say, $m - 1$). For example, if $k = 123456$, $m = 701$, and $m' = 700$, we have $h_1(k) = 80$ and $h_2(k) = 257$, so that we first probe position 80, and then we examine every 257th slot (modulo m) until we find the key or have examined every slot.

When m is prime or a power of 2, double hashing improves over linear or quadratic probing in that $\Theta(m^2)$ probe sequences are used, rather than $\Theta(m)$, since each possible $(h_1(k), h_2(k))$ pair yields a distinct probe sequence. As a result, for

such values of m , the performance of double hashing appears to be very close to the performance of the “ideal” scheme of uniform hashing.

Although values of m other than primes or powers of 2 could in principle be used with double hashing, in practice it becomes more difficult to efficiently generate $h_2(k)$ in a way that ensures that it is relatively prime to m , in part because the relative density $\phi(m)/m$ of such numbers may be small (see equation (31.24)).

Analysis of open-address hashing

As in our analysis of chaining, we express our analysis of open addressing in terms of the load factor $\alpha = n/m$ of the hash table. Of course, with open addressing, at most one element occupies each slot, and thus $n \leq m$, which implies $\alpha \leq 1$.

We assume that we are using uniform hashing. In this idealized scheme, the probe sequence $\langle h(k, 0), h(k, 1), \dots, h(k, m-1) \rangle$ used to insert or search for each key k is equally likely to be any permutation of $\langle 0, 1, \dots, m-1 \rangle$. Of course, a given key has a unique fixed probe sequence associated with it; what we mean here is that, considering the probability distribution on the space of keys and the operation of the hash function on the keys, each possible probe sequence is equally likely.

We now analyze the expected number of probes for hashing with open addressing under the assumption of uniform hashing, beginning with an analysis of the number of probes made in an unsuccessful search.

Theorem 11.6

Given an open-address hash table with load factor $\alpha = n/m < 1$, the expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$, assuming uniform hashing.

Proof In an unsuccessful search, every probe but the last accesses an occupied slot that does not contain the desired key, and the last slot probed is empty. Let us define the random variable X to be the number of probes made in an unsuccessful search, and let us also define the event A_i , for $i = 1, 2, \dots$, to be the event that an i th probe occurs and it is to an occupied slot. Then the event $\{X \geq i\}$ is the intersection of events $A_1 \cap A_2 \cap \dots \cap A_{i-1}$. We will bound $\Pr\{X \geq i\}$ by bounding $\Pr\{A_1 \cap A_2 \cap \dots \cap A_{i-1}\}$. By Exercise C.2-5,

$$\Pr\{A_1 \cap A_2 \cap \dots \cap A_{i-1}\} = \Pr\{A_1\} \cdot \Pr\{A_2 \mid A_1\} \cdot \Pr\{A_3 \mid A_1 \cap A_2\} \cdots \Pr\{A_{i-1} \mid A_1 \cap A_2 \cap \dots \cap A_{i-2}\}.$$

Since there are n elements and m slots, $\Pr\{A_1\} = n/m$. For $j > 1$, the probability that there is a j th probe and it is to an occupied slot, given that the first $j-1$ probes were to occupied slots, is $(n-j+1)/(m-j+1)$. This probability follows

because we would be finding one of the remaining $(n - (j - 1))$ elements in one of the $(m - (j - 1))$ unexamined slots, and by the assumption of uniform hashing, the probability is the ratio of these quantities. Observing that $n < m$ implies that $(n - j)/(m - j) \leq n/m$ for all j such that $0 \leq j < m$, we have for all i such that $1 \leq i \leq m$,

$$\begin{aligned} \Pr\{X \geq i\} &= \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdots \frac{n-i+2}{m-i+2} \\ &\leq \left(\frac{n}{m}\right)^{i-1} \\ &= \alpha^{i-1}. \end{aligned}$$

Now, we use equation (C.25) to bound the expected number of probes:

$$\begin{aligned} E[X] &= \sum_{i=1}^{\infty} \Pr\{X \geq i\} \\ &\leq \sum_{i=1}^{\infty} \alpha^{i-1} \\ &= \sum_{i=0}^{\infty} \alpha^i \\ &= \frac{1}{1-\alpha}. \end{aligned}$$

■

This bound of $1/(1-\alpha) = 1 + \alpha + \alpha^2 + \alpha^3 + \cdots$ has an intuitive interpretation. We always make the first probe. With probability approximately α , the first probe finds an occupied slot, so that we need to probe a second time. With probability approximately α^2 , the first two slots are occupied so that we make a third probe, and so on.

If α is a constant, Theorem 11.6 predicts that an unsuccessful search runs in $O(1)$ time. For example, if the hash table is half full, the average number of probes in an unsuccessful search is at most $1/(1 - .5) = 2$. If it is 90 percent full, the average number of probes is at most $1/(1 - .9) = 10$.

Theorem 11.6 gives us the performance of the HASH-INSERT procedure almost immediately.

Corollary 11.7

Inserting an element into an open-address hash table with load factor α requires at most $1/(1 - \alpha)$ probes on average, assuming uniform hashing.

Proof An element is inserted only if there is room in the table, and thus $\alpha < 1$. Inserting a key requires an unsuccessful search followed by placing the key into the first empty slot found. Thus, the expected number of probes is at most $1/(1-\alpha)$. ■

We have to do a little more work to compute the expected number of probes for a successful search.

Theorem 11.8

Given an open-address hash table with load factor $\alpha < 1$, the expected number of probes in a successful search is at most

$$\frac{1}{\alpha} \ln \frac{1}{1-\alpha},$$

assuming uniform hashing and assuming that each key in the table is equally likely to be searched for.

Proof A search for a key k reproduces the same probe sequence as when the element with key k was inserted. By Corollary 11.7, if k was the $(i+1)$ st key inserted into the hash table, the expected number of probes made in a search for k is at most $1/(1-i/m) = m/(m-i)$. Averaging over all n keys in the hash table gives us the expected number of probes in a successful search:

$$\begin{aligned} \frac{1}{n} \sum_{i=0}^{n-1} \frac{m}{m-i} &= \frac{m}{n} \sum_{i=0}^{n-1} \frac{1}{m-i} \\ &= \frac{1}{\alpha} \sum_{k=m-n+1}^m \frac{1}{k} \\ &\leq \frac{1}{\alpha} \int_{m-n}^m (1/x) dx \quad (\text{by inequality (A.12)}) \\ &= \frac{1}{\alpha} \ln \frac{m}{m-n} \\ &= \frac{1}{\alpha} \ln \frac{1}{1-\alpha}. \quad \blacksquare \end{aligned}$$

If the hash table is half full, the expected number of probes in a successful search is less than 1.387. If the hash table is 90 percent full, the expected number of probes is less than 2.559.

Exercises**11.4-1**

Consider inserting the keys 10, 22, 31, 4, 15, 28, 17, 88, 59 into a hash table of length $m = 11$ using open addressing with the auxiliary hash function $h'(k) = k$. Illustrate the result of inserting these keys using linear probing, using quadratic probing with $c_1 = 1$ and $c_2 = 3$, and using double hashing with $h_1(k) = k$ and $h_2(k) = 1 + (k \bmod (m - 1))$.

11.4-2

Write pseudocode for HASH-DELETE as outlined in the text, and modify HASH-INSERT to handle the special value DELETED.

11.4-3

Consider an open-address hash table with uniform hashing. Give upper bounds on the expected number of probes in an unsuccessful search and on the expected number of probes in a successful search when the load factor is $3/4$ and when it is $7/8$.

11.4-4 ★

Suppose that we use double hashing to resolve collisions—that is, we use the hash function $h(k, i) = (h_1(k) + ih_2(k)) \bmod m$. Show that if m and $h_2(k)$ have greatest common divisor $d \geq 1$ for some key k , then an unsuccessful search for key k examines $(1/d)$ th of the hash table before returning to slot $h_1(k)$. Thus, when $d = 1$, so that m and $h_2(k)$ are relatively prime, the search may examine the entire hash table. (*Hint:* See Chapter 31.)

11.4-5 ★

Consider an open-address hash table with a load factor α . Find the nonzero value α for which the expected number of probes in an unsuccessful search equals twice the expected number of probes in a successful search. Use the upper bounds given by Theorems 11.6 and 11.8 for these expected numbers of probes.

★ 11.5 Perfect hashing

Although hashing is often a good choice for its excellent average-case performance, hashing can also provide excellent *worst-case* performance when the set of keys is *static*: once the keys are stored in the table, the set of keys never changes. Some applications naturally have static sets of keys: consider the set of reserved words in a programming language, or the set of file names on a CD-ROM. We

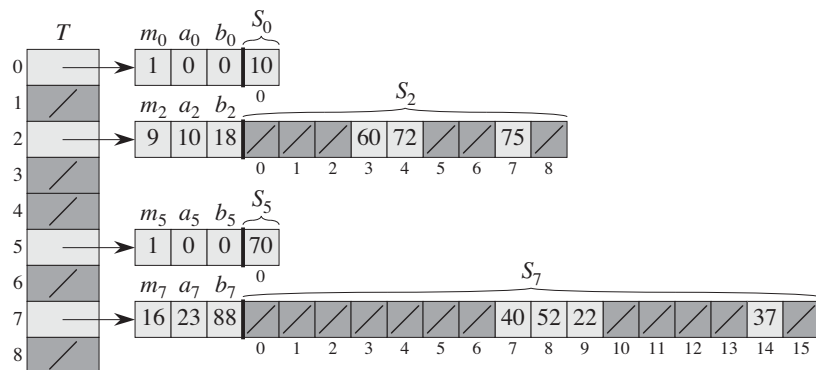


Figure 11.6 Using perfect hashing to store the set $K = \{10, 22, 37, 40, 52, 60, 70, 72, 75\}$. The outer hash function is $h(k) = ((ak + b) \bmod p) \bmod m$, where $a = 3$, $b = 42$, $p = 101$, and $m = 9$. For example, $h(75) = 2$, and so key 75 hashes to slot 2 of table T . A secondary hash table S_j stores all keys hashing to slot j . The size of hash table S_j is $m_j = n_j^2$, and the associated hash function is $h_j(k) = ((a_j k + b_j) \bmod p) \bmod m_j$. Since $h_2(75) = 7$, key 75 is stored in slot 7 of secondary hash table S_2 . No collisions occur in any of the secondary hash tables, and so searching takes constant time in the worst case.

call a hashing technique **perfect hashing** if $O(1)$ memory accesses are required to perform a search in the worst case.

To create a perfect hashing scheme, we use two levels of hashing, with universal hashing at each level. Figure 11.6 illustrates the approach.

The first level is essentially the same as for hashing with chaining: we hash the n keys into m slots using a hash function h carefully selected from a family of universal hash functions.

Instead of making a linked list of the keys hashing to slot j , however, we use a small **secondary hash table** S_j with an associated hash function h_j . By choosing the hash functions h_j carefully, we can guarantee that there are no collisions at the secondary level.

In order to guarantee that there are no collisions at the secondary level, however, we will need to let the size m_j of hash table S_j be the square of the number n_j of keys hashing to slot j . Although you might think that the quadratic dependence of m_j on n_j may seem likely to cause the overall storage requirement to be excessive, we shall show that by choosing the first-level hash function well, we can limit the expected total amount of space used to $O(n)$.

We use hash functions chosen from the universal classes of hash functions of Section 11.3.3. The first-level hash function comes from the class \mathcal{H}_{pm} , where as in Section 11.3.3, p is a prime number greater than any key value. Those keys

hashing to slot j are re-hashed into a secondary hash table S_j of size m_j using a hash function h_j chosen from the class \mathcal{H}_{p,m_j} .¹

We shall proceed in two steps. First, we shall determine how to ensure that the secondary tables have no collisions. Second, we shall show that the expected amount of memory used overall—for the primary hash table and all the secondary hash tables—is $O(n)$.

Theorem 11.9

Suppose that we store n keys in a hash table of size $m = n^2$ using a hash function h randomly chosen from a universal class of hash functions. Then, the probability is less than $1/2$ that there are any collisions.

Proof There are $\binom{n}{2}$ pairs of keys that may collide; each pair collides with probability $1/m$ if h is chosen at random from a universal family \mathcal{H} of hash functions. Let X be a random variable that counts the number of collisions. When $m = n^2$, the expected number of collisions is

$$\begin{aligned} E[X] &= \binom{n}{2} \cdot \frac{1}{n^2} \\ &= \frac{n^2 - n}{2} \cdot \frac{1}{n^2} \\ &< 1/2. \end{aligned}$$

(This analysis is similar to the analysis of the birthday paradox in Section 5.4.1.) Applying Markov's inequality (C.30), $\Pr\{X \geq t\} \leq E[X]/t$, with $t = 1$, completes the proof. ■

In the situation described in Theorem 11.9, where $m = n^2$, it follows that a hash function h chosen at random from \mathcal{H} is more likely than not to have *no* collisions. Given the set K of n keys to be hashed (remember that K is static), it is thus easy to find a collision-free hash function h with a few random trials.

When n is large, however, a hash table of size $m = n^2$ is excessive. Therefore, we adopt the two-level hashing approach, and we use the approach of Theorem 11.9 only to hash the entries within each slot. We use an outer, or first-level, hash function h to hash the keys into $m = n$ slots. Then, if n_j keys hash to slot j , we use a secondary hash table S_j of size $m_j = n_j^2$ to provide collision-free constant-time lookup.

¹When $n_j = m_j = 1$, we don't really need a hash function for slot j ; when we choose a hash function $h_{ab}(k) = ((ak + b) \bmod p) \bmod m_j$ for such a slot, we just use $a = b = 0$.

We now turn to the issue of ensuring that the overall memory used is $O(n)$. Since the size m_j of the j th secondary hash table grows quadratically with the number n_j of keys stored, we run the risk that the overall amount of storage could be excessive.

If the first-level table size is $m = n$, then the amount of memory used is $O(n)$ for the primary hash table, for the storage of the sizes m_j of the secondary hash tables, and for the storage of the parameters a_j and b_j defining the secondary hash functions h_j drawn from the class \mathcal{H}_{p,m_j} of Section 11.3.3 (except when $n_j = 1$ and we use $a = b = 0$). The following theorem and a corollary provide a bound on the expected combined sizes of all the secondary hash tables. A second corollary bounds the probability that the combined size of all the secondary hash tables is superlinear (actually, that it equals or exceeds $4n$).

Theorem 11.10

Suppose that we store n keys in a hash table of size $m = n$ using a hash function h randomly chosen from a universal class of hash functions. Then, we have

$$\mathbb{E} \left[\sum_{j=0}^{m-1} n_j^2 \right] < 2n ,$$

where n_j is the number of keys hashing to slot j .

Proof We start with the following identity, which holds for any nonnegative integer a :

$$a^2 = a + 2 \binom{a}{2} . \tag{11.6}$$

We have

$$\begin{aligned} \mathbb{E} \left[\sum_{j=0}^{m-1} n_j^2 \right] &= \mathbb{E} \left[\sum_{j=0}^{m-1} \left(n_j + 2 \binom{n_j}{2} \right) \right] && \text{(by equation (11.6))} \\ &= \mathbb{E} \left[\sum_{j=0}^{m-1} n_j \right] + 2 \mathbb{E} \left[\sum_{j=0}^{m-1} \binom{n_j}{2} \right] && \text{(by linearity of expectation)} \\ &= \mathbb{E} [n] + 2 \mathbb{E} \left[\sum_{j=0}^{m-1} \binom{n_j}{2} \right] && \text{(by equation (11.1))} \end{aligned}$$

$$= n + 2 \mathbb{E} \left[\sum_{j=0}^{m-1} \binom{n_j}{2} \right] \quad (\text{since } n \text{ is not a random variable}) .$$

To evaluate the summation $\sum_{j=0}^{m-1} \binom{n_j}{2}$, we observe that it is just the total number of pairs of keys in the hash table that collide. By the properties of universal hashing, the expected value of this summation is at most

$$\begin{aligned} \binom{n}{2} \frac{1}{m} &= \frac{n(n-1)}{2m} \\ &= \frac{n-1}{2} , \end{aligned}$$

since $m = n$. Thus,

$$\begin{aligned} \mathbb{E} \left[\sum_{j=0}^{m-1} n_j^2 \right] &\leq n + 2 \frac{n-1}{2} \\ &= 2n - 1 \\ &< 2n . \end{aligned}$$

■

Corollary 11.11

Suppose that we store n keys in a hash table of size $m = n$ using a hash function h randomly chosen from a universal class of hash functions, and we set the size of each secondary hash table to $m_j = n_j^2$ for $j = 0, 1, \dots, m-1$. Then, the expected amount of storage required for all secondary hash tables in a perfect hashing scheme is less than $2n$.

Proof Since $m_j = n_j^2$ for $j = 0, 1, \dots, m-1$, Theorem 11.10 gives

$$\begin{aligned} \mathbb{E} \left[\sum_{j=0}^{m-1} m_j \right] &= \mathbb{E} \left[\sum_{j=0}^{m-1} n_j^2 \right] \\ &< 2n , \end{aligned} \tag{11.7}$$

which completes the proof. ■

Corollary 11.12

Suppose that we store n keys in a hash table of size $m = n$ using a hash function h randomly chosen from a universal class of hash functions, and we set the size of each secondary hash table to $m_j = n_j^2$ for $j = 0, 1, \dots, m-1$. Then, the probability is less than $1/2$ that the total storage used for secondary hash tables equals or exceeds $4n$.

Proof Again we apply Markov's inequality (C.30), $\Pr\{X \geq t\} \leq E[X]/t$, this time to inequality (11.7), with $X = \sum_{j=0}^{m-1} m_j$ and $t = 4n$:

$$\begin{aligned} \Pr\left\{\sum_{j=0}^{m-1} m_j \geq 4n\right\} &\leq \frac{E\left[\sum_{j=0}^{m-1} m_j\right]}{4n} \\ &< \frac{2n}{4n} \\ &= 1/2. \end{aligned}$$

■

From Corollary 11.12, we see that if we test a few randomly chosen hash functions from the universal family, we will quickly find one that uses a reasonable amount of storage.

Exercises

11.5-1 ★

Suppose that we insert n keys into a hash table of size m using open addressing and uniform hashing. Let $p(n, m)$ be the probability that no collisions occur. Show that $p(n, m) \leq e^{-n(n-1)/2m}$. (*Hint*: See equation (3.12).) Argue that when n exceeds \sqrt{m} , the probability of avoiding collisions goes rapidly to zero.

Problems

11-1 Longest-probe bound for hashing

Suppose that we use an open-addressed hash table of size m to store $n \leq m/2$ items.

- Assuming uniform hashing, show that for $i = 1, 2, \dots, n$, the probability is at most 2^{-k} that the i th insertion requires strictly more than k probes.
- Show that for $i = 1, 2, \dots, n$, the probability is $O(1/n^2)$ that the i th insertion requires more than $2 \lg n$ probes.

Let the random variable X_i denote the number of probes required by the i th insertion. You have shown in part (b) that $\Pr\{X_i > 2 \lg n\} = O(1/n^2)$. Let the random variable $X = \max_{1 \leq i \leq n} X_i$ denote the maximum number of probes required by any of the n insertions.

- Show that $\Pr\{X > 2 \lg n\} = O(1/n)$.
- Show that the expected length $E[X]$ of the longest probe sequence is $O(\lg n)$.

11-2 Slot-size bound for chaining

Suppose that we have a hash table with n slots, with collisions resolved by chaining, and suppose that n keys are inserted into the table. Each key is equally likely to be hashed to each slot. Let M be the maximum number of keys in any slot after all the keys have been inserted. Your mission is to prove an $O(\lg n / \lg \lg n)$ upper bound on $E[M]$, the expected value of M .

- a. Argue that the probability Q_k that exactly k keys hash to a particular slot is given by

$$Q_k = \left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{n-k} \binom{n}{k}.$$

- b. Let P_k be the probability that $M = k$, that is, the probability that the slot containing the most keys contains k keys. Show that $P_k \leq n Q_k$.
- c. Use Stirling's approximation, equation (3.18), to show that $Q_k < e^k / k^k$.
- d. Show that there exists a constant $c > 1$ such that $Q_{k_0} < 1/n^3$ for $k_0 = c \lg n / \lg \lg n$. Conclude that $P_k < 1/n^2$ for $k \geq k_0 = c \lg n / \lg \lg n$.

- e. Argue that

$$E[M] \leq \Pr \left\{ M > \frac{c \lg n}{\lg \lg n} \right\} \cdot n + \Pr \left\{ M \leq \frac{c \lg n}{\lg \lg n} \right\} \cdot \frac{c \lg n}{\lg \lg n}.$$

Conclude that $E[M] = O(\lg n / \lg \lg n)$.

11-3 Quadratic probing

Suppose that we are given a key k to search for in a hash table with positions $0, 1, \dots, m-1$, and suppose that we have a hash function h mapping the key space into the set $\{0, 1, \dots, m-1\}$. The search scheme is as follows:

1. Compute the value $j = h(k)$, and set $i = 0$.
2. Probe in position j for the desired key k . If you find it, or if this position is empty, terminate the search.
3. Set $i = i + 1$. If i now equals m , the table is full, so terminate the search. Otherwise, set $j = (i + j) \bmod m$, and return to step 2.

Assume that m is a power of 2.

- a. Show that this scheme is an instance of the general “quadratic probing” scheme by exhibiting the appropriate constants c_1 and c_2 for equation (11.5).
- b. Prove that this algorithm examines every table position in the worst case.

11-4 Hashing and authentication

Let \mathcal{H} be a class of hash functions in which each hash function $h \in \mathcal{H}$ maps the universe U of keys to $\{0, 1, \dots, m-1\}$. We say that \mathcal{H} is **k -universal** if, for every fixed sequence of k distinct keys $\langle x^{(1)}, x^{(2)}, \dots, x^{(k)} \rangle$ and for any h chosen at random from \mathcal{H} , the sequence $\langle h(x^{(1)}), h(x^{(2)}), \dots, h(x^{(k)}) \rangle$ is equally likely to be any of the m^k sequences of length k with elements drawn from $\{0, 1, \dots, m-1\}$.

- a. Show that if the family \mathcal{H} of hash functions is 2-universal, then it is universal.
- b. Suppose that the universe U is the set of n -tuples of values drawn from $\mathbb{Z}_p = \{0, 1, \dots, p-1\}$, where p is prime. Consider an element $x = \langle x_0, x_1, \dots, x_{n-1} \rangle \in U$. For any n -tuple $a = \langle a_0, a_1, \dots, a_{n-1} \rangle \in U$, define the hash function h_a by

$$h_a(x) = \left(\sum_{j=0}^{n-1} a_j x_j \right) \bmod p.$$

Let $\mathcal{H} = \{h_a\}$. Show that \mathcal{H} is universal, but not 2-universal. (*Hint*: Find a key for which all hash functions in \mathcal{H} produce the same value.)

- c. Suppose that we modify \mathcal{H} slightly from part (b): for any $a \in U$ and for any $b \in \mathbb{Z}_p$, define

$$h'_{ab}(x) = \left(\sum_{j=0}^{n-1} a_j x_j + b \right) \bmod p$$

and $\mathcal{H}' = \{h'_{ab}\}$. Argue that \mathcal{H}' is 2-universal. (*Hint*: Consider fixed n -tuples $x \in U$ and $y \in U$, with $x_i \neq y_i$ for some i . What happens to $h'_{ab}(x)$ and $h'_{ab}(y)$ as a_i and b range over \mathbb{Z}_p ?)

- d. Suppose that Alice and Bob secretly agree on a hash function h from a 2-universal family \mathcal{H} of hash functions. Each $h \in \mathcal{H}$ maps from a universe of keys U to \mathbb{Z}_p , where p is prime. Later, Alice sends a message m to Bob over the Internet, where $m \in U$. She authenticates this message to Bob by also sending an authentication tag $t = h(m)$, and Bob checks that the pair (m, t) he receives indeed satisfies $t = h(m)$. Suppose that an adversary intercepts (m, t) en route and tries to fool Bob by replacing the pair (m, t) with a different pair (m', t') . Argue that the probability that the adversary succeeds in fooling Bob into accepting (m', t') is at most $1/p$, no matter how much computing power the adversary has, and even if the adversary knows the family \mathcal{H} of hash functions used.

Chapter notes

Knuth [211] and Gonnet [145] are excellent references for the analysis of hashing algorithms. Knuth credits H. P. Luhn (1953) for inventing hash tables, along with the chaining method for resolving collisions. At about the same time, G. M. Amdahl originated the idea of open addressing.

Carter and Wegman introduced the notion of universal classes of hash functions in 1979 [58].

Fredman, Komlós, and Szemerédi [112] developed the perfect hashing scheme for static sets presented in Section 11.5. An extension of their method to dynamic sets, handling insertions and deletions in amortized expected time $O(1)$, has been given by Dietzfelbinger et al. [86].

12 Binary Search Trees

The search tree data structure supports many dynamic-set operations, including SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, and DELETE. Thus, we can use a search tree both as a dictionary and as a priority queue.

Basic operations on a binary search tree take time proportional to the height of the tree. For a complete binary tree with n nodes, such operations run in $\Theta(\lg n)$ worst-case time. If the tree is a linear chain of n nodes, however, the same operations take $\Theta(n)$ worst-case time. We shall see in Section 12.4 that the expected height of a randomly built binary search tree is $O(\lg n)$, so that basic dynamic-set operations on such a tree take $\Theta(\lg n)$ time on average.

In practice, we can't always guarantee that binary search trees are built randomly, but we can design variations of binary search trees with good guaranteed worst-case performance on basic operations. Chapter 13 presents one such variation, red-black trees, which have height $O(\lg n)$. Chapter 18 introduces B-trees, which are particularly good for maintaining databases on secondary (disk) storage.

After presenting the basic properties of binary search trees, the following sections show how to walk a binary search tree to print its values in sorted order, how to search for a value in a binary search tree, how to find the minimum or maximum element, how to find the predecessor or successor of an element, and how to insert into or delete from a binary search tree. The basic mathematical properties of trees appear in Appendix B.

12.1 What is a binary search tree?

A binary search tree is organized, as the name suggests, in a binary tree, as shown in Figure 12.1. We can represent such a tree by a linked data structure in which each node is an object. In addition to a *key* and satellite data, each node contains attributes *left*, *right*, and *p* that point to the nodes corresponding to its left child,

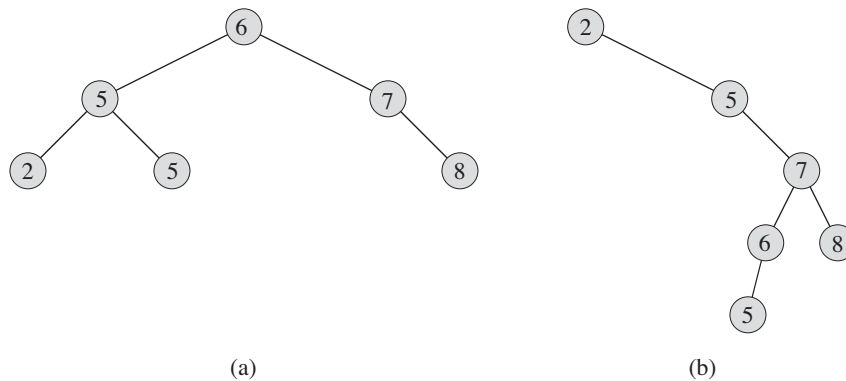


Figure 12.1 Binary search trees. For any node x , the keys in the left subtree of x are at most $x.key$, and the keys in the right subtree of x are at least $x.key$. Different binary search trees can represent the same set of values. The worst-case running time for most search-tree operations is proportional to the height of the tree. **(a)** A binary search tree on 6 nodes with height 2. **(b)** A less efficient binary search tree with height 4 that contains the same keys.

its right child, and its parent, respectively. If a child or the parent is missing, the appropriate attribute contains the value NIL. The root node is the only node in the tree whose parent is NIL.

The keys in a binary search tree are always stored in such a way as to satisfy the **binary-search-tree property**:

Let x be a node in a binary search tree. If y is a node in the left subtree of x , then $y.key \leq x.key$. If y is a node in the right subtree of x , then $y.key \geq x.key$.

Thus, in Figure 12.1(a), the key of the root is 6, the keys 2, 5, and 5 in its left subtree are no larger than 6, and the keys 7 and 8 in its right subtree are no smaller than 6. The same property holds for every node in the tree. For example, the key 5 in the root's left child is no smaller than the key 2 in that node's left subtree and no larger than the key 5 in the right subtree.

The binary-search-tree property allows us to print out all the keys in a binary search tree in sorted order by a simple recursive algorithm, called an **inorder tree walk**. This algorithm is so named because it prints the key of the root of a subtree between printing the values in its left subtree and printing those in its right subtree. (Similarly, a **preorder tree walk** prints the root before the values in either subtree, and a **postorder tree walk** prints the root after the values in its subtrees.) To use the following procedure to print all the elements in a binary search tree T , we call INORDER-TREE-WALK($T.root$).

INORDER-TREE-WALK(x)

```

1  if  $x \neq \text{NIL}$ 
2      INORDER-TREE-WALK( $x.\text{left}$ )
3      print  $x.\text{key}$ 
4      INORDER-TREE-WALK( $x.\text{right}$ )

```

As an example, the inorder tree walk prints the keys in each of the two binary search trees from Figure 12.1 in the order 2, 5, 5, 6, 7, 8. The correctness of the algorithm follows by induction directly from the binary-search-tree property.

It takes $\Theta(n)$ time to walk an n -node binary search tree, since after the initial call, the procedure calls itself recursively exactly twice for each node in the tree—once for its left child and once for its right child. The following theorem gives a formal proof that it takes linear time to perform an inorder tree walk.

Theorem 12.1

If x is the root of an n -node subtree, then the call INORDER-TREE-WALK(x) takes $\Theta(n)$ time.

Proof Let $T(n)$ denote the time taken by INORDER-TREE-WALK when it is called on the root of an n -node subtree. Since INORDER-TREE-WALK visits all n nodes of the subtree, we have $T(n) = \Omega(n)$. It remains to show that $T(n) = O(n)$.

Since INORDER-TREE-WALK takes a small, constant amount of time on an empty subtree (for the test $x \neq \text{NIL}$), we have $T(0) = c$ for some constant $c > 0$.

For $n > 0$, suppose that INORDER-TREE-WALK is called on a node x whose left subtree has k nodes and whose right subtree has $n - k - 1$ nodes. The time to perform INORDER-TREE-WALK(x) is bounded by $T(n) \leq T(k) + T(n - k - 1) + d$ for some constant $d > 0$ that reflects an upper bound on the time to execute the body of INORDER-TREE-WALK(x), exclusive of the time spent in recursive calls.

We use the substitution method to show that $T(n) = O(n)$ by proving that $T(n) \leq (c + d)n + c$. For $n = 0$, we have $(c + d) \cdot 0 + c = c = T(0)$. For $n > 0$, we have

$$\begin{aligned}
 T(n) &\leq T(k) + T(n - k - 1) + d \\
 &= ((c + d)k + c) + ((c + d)(n - k - 1) + c) + d \\
 &= (c + d)n + c - (c + d) + c + d \\
 &= (c + d)n + c,
 \end{aligned}$$

which completes the proof. ■

Exercises

12.1-1

For the set of $\{1, 4, 5, 10, 16, 17, 21\}$ of keys, draw binary search trees of heights 2, 3, 4, 5, and 6.

12.1-2

What is the difference between the binary-search-tree property and the min-heap property (see page 153)? Can the min-heap property be used to print out the keys of an n -node tree in sorted order in $O(n)$ time? Show how, or explain why not.

12.1-3

Give a nonrecursive algorithm that performs an inorder tree walk. (*Hint:* An easy solution uses a stack as an auxiliary data structure. A more complicated, but elegant, solution uses no stack but assumes that we can test two pointers for equality.)

12.1-4

Give recursive algorithms that perform preorder and postorder tree walks in $\Theta(n)$ time on a tree of n nodes.

12.1-5

Argue that since sorting n elements takes $\Omega(n \lg n)$ time in the worst case in the comparison model, any comparison-based algorithm for constructing a binary search tree from an arbitrary list of n elements takes $\Omega(n \lg n)$ time in the worst case.

12.2 Querying a binary search tree

We often need to search for a key stored in a binary search tree. Besides the SEARCH operation, binary search trees can support such queries as MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR. In this section, we shall examine these operations and show how to support each one in time $O(h)$ on any binary search tree of height h .

Searching

We use the following procedure to search for a node with a given key in a binary search tree. Given a pointer to the root of the tree and a key k , TREE-SEARCH returns a pointer to a node with key k if one exists; otherwise, it returns NIL.

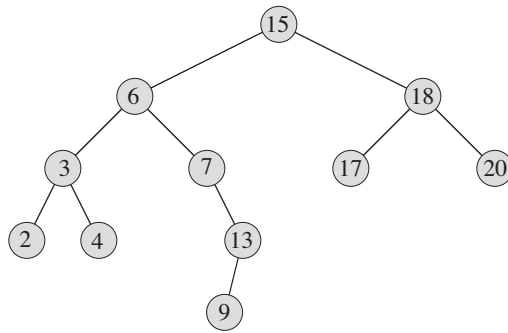


Figure 12.2 Queries on a binary search tree. To search for the key 13 in the tree, we follow the path $15 \rightarrow 6 \rightarrow 7 \rightarrow 13$ from the root. The minimum key in the tree is 2, which is found by following *left* pointers from the root. The maximum key 20 is found by following *right* pointers from the root. The successor of the node with key 15 is the node with key 17, since it is the minimum key in the right subtree of 15. The node with key 13 has no right subtree, and thus its successor is its lowest ancestor whose left child is also an ancestor. In this case, the node with key 15 is its successor.

TREE-SEARCH(x, k)

```

1  if  $x == \text{NIL}$  or  $k == x.\text{key}$ 
2      return  $x$ 
3  if  $k < x.\text{key}$ 
4      return TREE-SEARCH( $x.\text{left}, k$ )
5  else return TREE-SEARCH( $x.\text{right}, k$ )
  
```

The procedure begins its search at the root and traces a simple path downward in the tree, as shown in Figure 12.2. For each node x it encounters, it compares the key k with $x.\text{key}$. If the two keys are equal, the search terminates. If k is smaller than $x.\text{key}$, the search continues in the left subtree of x , since the binary-search-tree property implies that k could not be stored in the right subtree. Symmetrically, if k is larger than $x.\text{key}$, the search continues in the right subtree. The nodes encountered during the recursion form a simple path downward from the root of the tree, and thus the running time of TREE-SEARCH is $O(h)$, where h is the height of the tree.

We can rewrite this procedure in an iterative fashion by “unrolling” the recursion into a **while** loop. On most computers, the iterative version is more efficient.

ITERATIVE-TREE-SEARCH(x, k)

```

1  while  $x \neq \text{NIL}$  and  $k \neq x.\text{key}$ 
2      if  $k < x.\text{key}$ 
3           $x = x.\text{left}$ 
4      else  $x = x.\text{right}$ 
5  return  $x$ 

```

Minimum and maximum

We can always find an element in a binary search tree whose key is a minimum by following *left* child pointers from the root until we encounter a NIL, as shown in Figure 12.2. The following procedure returns a pointer to the minimum element in the subtree rooted at a given node x , which we assume to be non-NIL:

TREE-MINIMUM(x)

```

1  while  $x.\text{left} \neq \text{NIL}$ 
2       $x = x.\text{left}$ 
3  return  $x$ 

```

The binary-search-tree property guarantees that TREE-MINIMUM is correct. If a node x has no left subtree, then since every key in the right subtree of x is at least as large as $x.\text{key}$, the minimum key in the subtree rooted at x is $x.\text{key}$. If node x has a left subtree, then since no key in the right subtree is smaller than $x.\text{key}$ and every key in the left subtree is not larger than $x.\text{key}$, the minimum key in the subtree rooted at x resides in the subtree rooted at $x.\text{left}$.

The pseudocode for TREE-MAXIMUM is symmetric:

TREE-MAXIMUM(x)

```

1  while  $x.\text{right} \neq \text{NIL}$ 
2       $x = x.\text{right}$ 
3  return  $x$ 

```

Both of these procedures run in $O(h)$ time on a tree of height h since, as in TREE-SEARCH, the sequence of nodes encountered forms a simple path downward from the root.

Successor and predecessor

Given a node in a binary search tree, sometimes we need to find its successor in the sorted order determined by an inorder tree walk. If all keys are distinct, the

successor of a node x is the node with the smallest key greater than $x.key$. The structure of a binary search tree allows us to determine the successor of a node without ever comparing keys. The following procedure returns the successor of a node x in a binary search tree if it exists, and NIL if x has the largest key in the tree:

```

TREE-SUCCESSOR( $x$ )
1  if  $x.right \neq \text{NIL}$ 
2      return TREE-MINIMUM( $x.right$ )
3   $y = x.p$ 
4  while  $y \neq \text{NIL}$  and  $x == y.right$ 
5       $x = y$ 
6       $y = y.p$ 
7  return  $y$ 

```

We break the code for TREE-SUCCESSOR into two cases. If the right subtree of node x is nonempty, then the successor of x is just the leftmost node in x 's right subtree, which we find in line 2 by calling TREE-MINIMUM($x.right$). For example, the successor of the node with key 15 in Figure 12.2 is the node with key 17.

On the other hand, as Exercise 12.2-6 asks you to show, if the right subtree of node x is empty and x has a successor y , then y is the lowest ancestor of x whose left child is also an ancestor of x . In Figure 12.2, the successor of the node with key 13 is the node with key 15. To find y , we simply go up the tree from x until we encounter a node that is the left child of its parent; lines 3–7 of TREE-SUCCESSOR handle this case.

The running time of TREE-SUCCESSOR on a tree of height h is $O(h)$, since we either follow a simple path up the tree or follow a simple path down the tree. The procedure TREE-PREDECESSOR, which is symmetric to TREE-SUCCESSOR, also runs in time $O(h)$.

Even if keys are not distinct, we define the successor and predecessor of any node x as the node returned by calls made to TREE-SUCCESSOR(x) and TREE-PREDECESSOR(x), respectively.

In summary, we have proved the following theorem.

Theorem 12.2

We can implement the dynamic-set operations SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR so that each one runs in $O(h)$ time on a binary search tree of height h . ■

Exercises**12.2-1**

Suppose that we have numbers between 1 and 1000 in a binary search tree, and we want to search for the number 363. Which of the following sequences could *not* be the sequence of nodes examined?

- a.* 2, 252, 401, 398, 330, 344, 397, 363.
- b.* 924, 220, 911, 244, 898, 258, 362, 363.
- c.* 925, 202, 911, 240, 912, 245, 363.
- d.* 2, 399, 387, 219, 266, 382, 381, 278, 363.
- e.* 935, 278, 347, 621, 299, 392, 358, 363.

12.2-2

Write recursive versions of TREE-MINIMUM and TREE-MAXIMUM.

12.2-3

Write the TREE-PREDECESSOR procedure.

12.2-4

Professor Bunyan thinks he has discovered a remarkable property of binary search trees. Suppose that the search for key k in a binary search tree ends up in a leaf. Consider three sets: A , the keys to the left of the search path; B , the keys on the search path; and C , the keys to the right of the search path. Professor Bunyan claims that any three keys $a \in A$, $b \in B$, and $c \in C$ must satisfy $a \leq b \leq c$. Give a smallest possible counterexample to the professor's claim.

12.2-5

Show that if a node in a binary search tree has two children, then its successor has no left child and its predecessor has no right child.

12.2-6

Consider a binary search tree T whose keys are distinct. Show that if the right subtree of a node x in T is empty and x has a successor y , then y is the lowest ancestor of x whose left child is also an ancestor of x . (Recall that every node is its own ancestor.)

12.2-7

An alternative method of performing an inorder tree walk of an n -node binary search tree finds the minimum element in the tree by calling TREE-MINIMUM and then making $n - 1$ calls to TREE-SUCCESSOR. Prove that this algorithm runs in $\Theta(n)$ time.

12.2-8

Prove that no matter what node we start at in a height- h binary search tree, k successive calls to TREE-SUCCESSOR take $O(k + h)$ time.

12.2-9

Let T be a binary search tree whose keys are distinct, let x be a leaf node, and let y be its parent. Show that $y.key$ is either the smallest key in T larger than $x.key$ or the largest key in T smaller than $x.key$.

12.3 Insertion and deletion

The operations of insertion and deletion cause the dynamic set represented by a binary search tree to change. The data structure must be modified to reflect this change, but in such a way that the binary-search-tree property continues to hold. As we shall see, modifying the tree to insert a new element is relatively straightforward, but handling deletion is somewhat more intricate.

Insertion

To insert a new value v into a binary search tree T , we use the procedure TREE-INSERT. The procedure takes a node z for which $z.key = v$, $z.left = \text{NIL}$, and $z.right = \text{NIL}$. It modifies T and some of the attributes of z in such a way that it inserts z into an appropriate position in the tree.

TREE-INSERT(T, z)

```

1   $y = \text{NIL}$ 
2   $x = T.root$ 
3  while  $x \neq \text{NIL}$ 
4       $y = x$ 
5      if  $z.key < x.key$ 
6           $x = x.left$ 
7      else  $x = x.right$ 
8   $z.p = y$ 
9  if  $y == \text{NIL}$ 
10      $T.root = z$       // tree  $T$  was empty
11 elseif  $z.key < y.key$ 
12      $y.left = z$ 
13 else  $y.right = z$ 
```

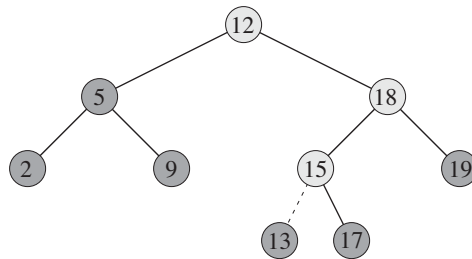


Figure 12.3 Inserting an item with key 13 into a binary search tree. Lightly shaded nodes indicate the simple path from the root down to the position where the item is inserted. The dashed line indicates the link in the tree that is added to insert the item.

Figure 12.3 shows how TREE-INSERT works. Just like the procedures TREE-SEARCH and ITERATIVE-TREE-SEARCH, TREE-INSERT begins at the root of the tree and the pointer x traces a simple path downward looking for a NIL to replace with the input item z . The procedure maintains the *trailing pointer* y as the parent of x . After initialization, the **while** loop in lines 3–7 causes these two pointers to move down the tree, going left or right depending on the comparison of $z.key$ with $x.key$, until x becomes NIL. This NIL occupies the position where we wish to place the input item z . We need the trailing pointer y , because by the time we find the NIL where z belongs, the search has proceeded one step beyond the node that needs to be changed. Lines 8–13 set the pointers that cause z to be inserted.

Like the other primitive operations on search trees, the procedure TREE-INSERT runs in $O(h)$ time on a tree of height h .

Deletion

The overall strategy for deleting a node z from a binary search tree T has three basic cases but, as we shall see, one of the cases is a bit tricky.

- If z has no children, then we simply remove it by modifying its parent to replace z with NIL as its child.
- If z has just one child, then we elevate that child to take z 's position in the tree by modifying z 's parent to replace z by z 's child.
- If z has two children, then we find z 's successor y —which must be in z 's right subtree—and have y take z 's position in the tree. The rest of z 's original right subtree becomes y 's new right subtree, and z 's left subtree becomes y 's new left subtree. This case is the tricky one because, as we shall see, it matters whether y is z 's right child.

The procedure for deleting a given node z from a binary search tree T takes as arguments pointers to T and z . It organizes its cases a bit differently from the three cases outlined previously by considering the four cases shown in Figure 12.4.

- If z has no left child (part (a) of the figure), then we replace z by its right child, which may or may not be NIL. When z 's right child is NIL, this case deals with the situation in which z has no children. When z 's right child is non-NIL, this case handles the situation in which z has just one child, which is its right child.
- If z has just one child, which is its left child (part (b) of the figure), then we replace z by its left child.
- Otherwise, z has both a left and a right child. We find z 's successor y , which lies in z 's right subtree and has no left child (see Exercise 12.2-5). We want to splice y out of its current location and have it replace z in the tree.
 - If y is z 's right child (part (c)), then we replace z by y , leaving y 's right child alone.
 - Otherwise, y lies within z 's right subtree but is not z 's right child (part (d)). In this case, we first replace y by its own right child, and then we replace z by y .

In order to move subtrees around within the binary search tree, we define a subroutine TRANSPLANT, which replaces one subtree as a child of its parent with another subtree. When TRANSPLANT replaces the subtree rooted at node u with the subtree rooted at node v , node u 's parent becomes node v 's parent, and u 's parent ends up having v as its appropriate child.

TRANSPLANT(T, u, v)

```

1  if  $u.p == \text{NIL}$ 
2       $T.\text{root} = v$ 
3  elseif  $u == u.p.\text{left}$ 
4       $u.p.\text{left} = v$ 
5  else  $u.p.\text{right} = v$ 
6  if  $v \neq \text{NIL}$ 
7       $v.p = u.p$ 
```

Lines 1–2 handle the case in which u is the root of T . Otherwise, u is either a left child or a right child of its parent. Lines 3–4 take care of updating $u.p.\text{left}$ if u is a left child, and line 5 updates $u.p.\text{right}$ if u is a right child. We allow v to be NIL, and lines 6–7 update $v.p$ if v is non-NIL. Note that TRANSPLANT does not attempt to update $v.\text{left}$ and $v.\text{right}$; doing so, or not doing so, is the responsibility of TRANSPLANT's caller.

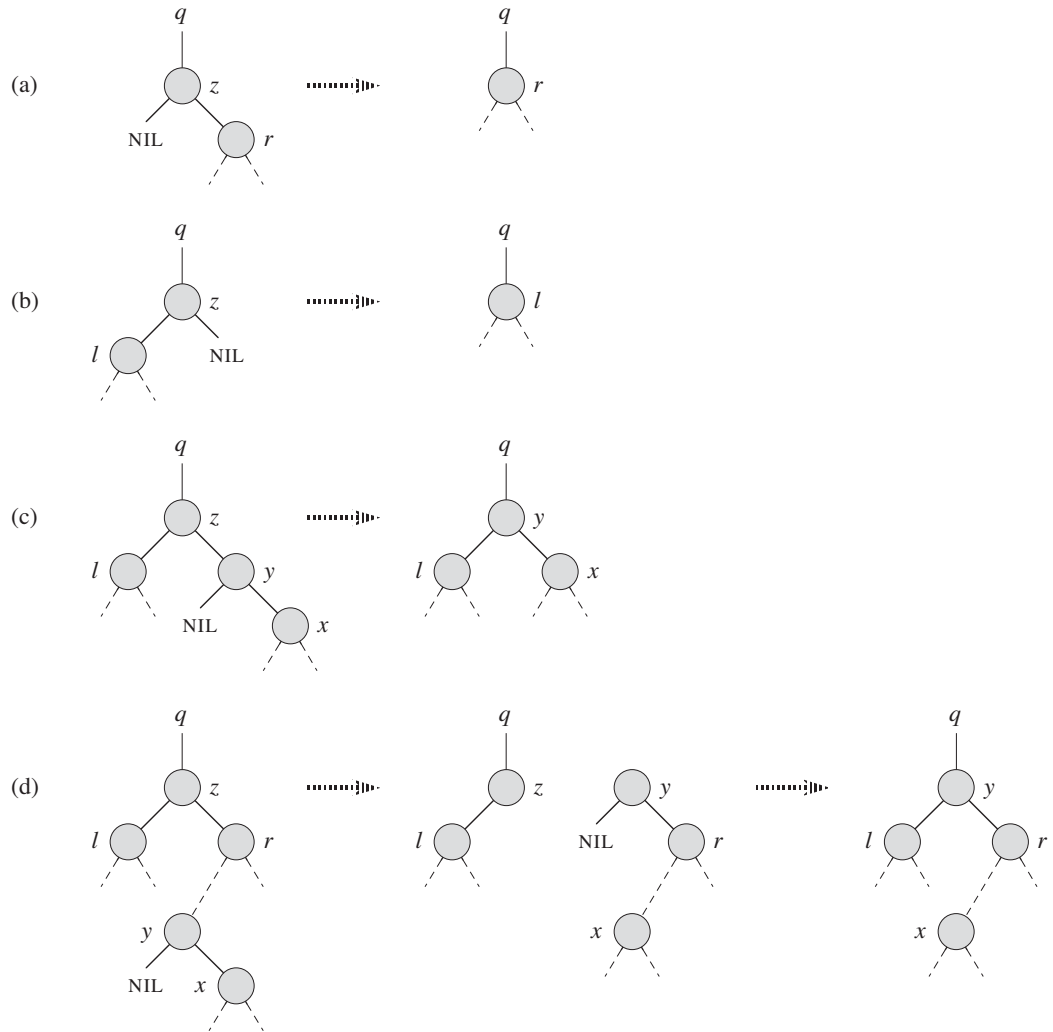


Figure 12.4 Deleting a node z from a binary search tree. Node z may be the root, a left child of node q , or a right child of q . **(a)** Node z has no left child. We replace z by its right child r , which may or may not be NIL. **(b)** Node z has a left child l but no right child. We replace z by l . **(c)** Node z has two children; its left child is node l , its right child is its successor y , and y 's right child is node x . We replace z by y , updating y 's left child to become l , but leaving x as y 's right child. **(d)** Node z has two children (left child l and right child r), and its successor $y \neq r$ lies within the subtree rooted at r . We replace y by its own right child x , and we set y to be r 's parent. Then, we set y to be q 's child and the parent of l .

With the TRANSPLANT procedure in hand, here is the procedure that deletes node z from binary search tree T :

```

TREE-DELETE( $T, z$ )
1  if  $z.left == \text{NIL}$ 
2      TRANSPLANT( $T, z, z.right$ )
3  elseif  $z.right == \text{NIL}$ 
4      TRANSPLANT( $T, z, z.left$ )
5  else  $y = \text{TREE-MINIMUM}(z.right)$ 
6      if  $y.p \neq z$ 
7          TRANSPLANT( $T, y, y.right$ )
8           $y.right = z.right$ 
9           $y.right.p = y$ 
10     TRANSPLANT( $T, z, y$ )
11      $y.left = z.left$ 
12      $y.left.p = y$ 

```

The TREE-DELETE procedure executes the four cases as follows. Lines 1–2 handle the case in which node z has no left child, and lines 3–4 handle the case in which z has a left child but no right child. Lines 5–12 deal with the remaining two cases, in which z has two children. Line 5 finds node y , which is the successor of z . Because z has a nonempty right subtree, its successor must be the node in that subtree with the smallest key; hence the call to TREE-MINIMUM($z.right$). As we noted before, y has no left child. We want to splice y out of its current location, and it should replace z in the tree. If y is z 's right child, then lines 10–12 replace z as a child of its parent by y and replace y 's left child by z 's left child. If y is not z 's left child, lines 7–9 replace y as a child of its parent by y 's right child and turn z 's right child into y 's right child, and then lines 10–12 replace z as a child of its parent by y and replace y 's left child by z 's left child.

Each line of TREE-DELETE, including the calls to TRANSPLANT, takes constant time, except for the call to TREE-MINIMUM in line 5. Thus, TREE-DELETE runs in $O(h)$ time on a tree of height h .

In summary, we have proved the following theorem.

Theorem 12.3

We can implement the dynamic-set operations INSERT and DELETE so that each one runs in $O(h)$ time on a binary search tree of height h . ■

Exercises

12.3-1

Give a recursive version of the TREE-INSERT procedure.

12.3-2

Suppose that we construct a binary search tree by repeatedly inserting distinct values into the tree. Argue that the number of nodes examined in searching for a value in the tree is one plus the number of nodes examined when the value was first inserted into the tree.

12.3-3

We can sort a given set of n numbers by first building a binary search tree containing these numbers (using TREE-INSERT repeatedly to insert the numbers one by one) and then printing the numbers by an inorder tree walk. What are the worst-case and best-case running times for this sorting algorithm?

12.3-4

Is the operation of deletion “commutative” in the sense that deleting x and then y from a binary search tree leaves the same tree as deleting y and then x ? Argue why it is or give a counterexample.

12.3-5

Suppose that instead of each node x keeping the attribute $x.p$, pointing to x 's parent, it keeps $x.succ$, pointing to x 's successor. Give pseudocode for SEARCH, INSERT, and DELETE on a binary search tree T using this representation. These procedures should operate in time $O(h)$, where h is the height of the tree T . (*Hint:* You may wish to implement a subroutine that returns the parent of a node.)

12.3-6

When node z in TREE-DELETE has two children, we could choose node y as its predecessor rather than its successor. What other changes to TREE-DELETE would be necessary if we did so? Some have argued that a fair strategy, giving equal priority to predecessor and successor, yields better empirical performance. How might TREE-DELETE be changed to implement such a fair strategy?

★ 12.4 Randomly built binary search trees

We have shown that each of the basic operations on a binary search tree runs in $O(h)$ time, where h is the height of the tree. The height of a binary search

tree varies, however, as items are inserted and deleted. If, for example, the n items are inserted in strictly increasing order, the tree will be a chain with height $n - 1$. On the other hand, Exercise B.5-4 shows that $h \geq \lfloor \lg n \rfloor$. As with quicksort, we can show that the behavior of the average case is much closer to the best case than to the worst case.

Unfortunately, little is known about the average height of a binary search tree when both insertion and deletion are used to create it. When the tree is created by insertion alone, the analysis becomes more tractable. Let us therefore define a **randomly built binary search tree** on n keys as one that arises from inserting the keys in random order into an initially empty tree, where each of the $n!$ permutations of the input keys is equally likely. (Exercise 12.4-3 asks you to show that this notion is different from assuming that every binary search tree on n keys is equally likely.) In this section, we shall prove the following theorem.

Theorem 12.4

The expected height of a randomly built binary search tree on n distinct keys is $O(\lg n)$.

Proof We start by defining three random variables that help measure the height of a randomly built binary search tree. We denote the height of a randomly built binary search on n keys by X_n , and we define the **exponential height** $Y_n = 2^{X_n}$. When we build a binary search tree on n keys, we choose one key as that of the root, and we let R_n denote the random variable that holds this key's **rank** within the set of n keys; that is, R_n holds the position that this key would occupy if the set of keys were sorted. The value of R_n is equally likely to be any element of the set $\{1, 2, \dots, n\}$. If $R_n = i$, then the left subtree of the root is a randomly built binary search tree on $i - 1$ keys, and the right subtree is a randomly built binary search tree on $n - i$ keys. Because the height of a binary tree is 1 more than the larger of the heights of the two subtrees of the root, the exponential height of a binary tree is twice the larger of the exponential heights of the two subtrees of the root. If we know that $R_n = i$, it follows that

$$Y_n = 2 \cdot \max(Y_{i-1}, Y_{n-i}) .$$

As base cases, we have that $Y_1 = 1$, because the exponential height of a tree with 1 node is $2^0 = 1$ and, for convenience, we define $Y_0 = 0$.

Next, define indicator random variables $Z_{n,1}, Z_{n,2}, \dots, Z_{n,n}$, where

$$Z_{n,i} = \mathbf{I}\{R_n = i\} .$$

Because R_n is equally likely to be any element of $\{1, 2, \dots, n\}$, it follows that $\Pr\{R_n = i\} = 1/n$ for $i = 1, 2, \dots, n$, and hence, by Lemma 5.1, we have

$$\mathbf{E}[Z_{n,i}] = 1/n , \tag{12.1}$$

for $i = 1, 2, \dots, n$. Because exactly one value of $Z_{n,i}$ is 1 and all others are 0, we also have

$$Y_n = \sum_{i=1}^n Z_{n,i} (2 \cdot \max(Y_{i-1}, Y_{n-i})) .$$

We shall show that $E[Y_n]$ is polynomial in n , which will ultimately imply that $E[X_n] = O(\lg n)$.

We claim that the indicator random variable $Z_{n,i} = I\{R_n = i\}$ is independent of the values of Y_{i-1} and Y_{n-i} . Having chosen $R_n = i$, the left subtree (whose exponential height is Y_{i-1}) is randomly built on the $i - 1$ keys whose ranks are less than i . This subtree is just like any other randomly built binary search tree on $i - 1$ keys. Other than the number of keys it contains, this subtree's structure is not affected at all by the choice of $R_n = i$, and hence the random variables Y_{i-1} and $Z_{n,i}$ are independent. Likewise, the right subtree, whose exponential height is Y_{n-i} , is randomly built on the $n - i$ keys whose ranks are greater than i . Its structure is independent of the value of R_n , and so the random variables Y_{n-i} and $Z_{n,i}$ are independent. Hence, we have

$$\begin{aligned} E[Y_n] &= E \left[\sum_{i=1}^n Z_{n,i} (2 \cdot \max(Y_{i-1}, Y_{n-i})) \right] \\ &= \sum_{i=1}^n E[Z_{n,i} (2 \cdot \max(Y_{i-1}, Y_{n-i}))] \quad (\text{by linearity of expectation}) \\ &= \sum_{i=1}^n E[Z_{n,i}] E[2 \cdot \max(Y_{i-1}, Y_{n-i})] \quad (\text{by independence}) \\ &= \sum_{i=1}^n \frac{1}{n} \cdot E[2 \cdot \max(Y_{i-1}, Y_{n-i})] \quad (\text{by equation (12.1)}) \\ &= \frac{2}{n} \sum_{i=1}^n E[\max(Y_{i-1}, Y_{n-i})] \quad (\text{by equation (C.22)}) \\ &\leq \frac{2}{n} \sum_{i=1}^n (E[Y_{i-1}] + E[Y_{n-i}]) \quad (\text{by Exercise C.3-4}) . \end{aligned}$$

Since each term $E[Y_0], E[Y_1], \dots, E[Y_{n-1}]$ appears twice in the last summation, once as $E[Y_{i-1}]$ and once as $E[Y_{n-i}]$, we have the recurrence

$$E[Y_n] \leq \frac{4}{n} \sum_{i=0}^{n-1} E[Y_i] . \tag{12.2}$$

Using the substitution method, we shall show that for all positive integers n , the recurrence (12.2) has the solution

$$E[Y_n] \leq \frac{1}{4} \binom{n+3}{3}.$$

In doing so, we shall use the identity

$$\sum_{i=0}^{n-1} \binom{i+3}{3} = \binom{n+3}{4}. \quad (12.3)$$

(Exercise 12.4-1 asks you to prove this identity.)

For the base cases, we note that the bounds $0 = Y_0 = E[Y_0] \leq (1/4)\binom{3}{3} = 1/4$ and $1 = Y_1 = E[Y_1] \leq (1/4)\binom{4}{3} = 1$ hold. For the inductive case, we have that

$$\begin{aligned} E[Y_n] &\leq \frac{4}{n} \sum_{i=0}^{n-1} E[Y_i] \\ &\leq \frac{4}{n} \sum_{i=0}^{n-1} \frac{1}{4} \binom{i+3}{3} \quad (\text{by the inductive hypothesis}) \\ &= \frac{1}{n} \sum_{i=0}^{n-1} \binom{i+3}{3} \\ &= \frac{1}{n} \binom{n+3}{4} \quad (\text{by equation (12.3)}) \\ &= \frac{1}{n} \cdot \frac{(n+3)!}{4! (n-1)!} \\ &= \frac{1}{4} \cdot \frac{(n+3)!}{3! n!} \\ &= \frac{1}{4} \binom{n+3}{3}. \end{aligned}$$

We have bounded $E[Y_n]$, but our ultimate goal is to bound $E[X_n]$. As Exercise 12.4-4 asks you to show, the function $f(x) = 2^x$ is convex (see page 1199). Therefore, we can employ Jensen's inequality (C.26), which says that

$$\begin{aligned} 2^{E[X_n]} &\leq E[2^{X_n}] \\ &= E[Y_n], \end{aligned}$$

as follows:

$$2^{E[X_n]} \leq \frac{1}{4} \binom{n+3}{3}$$

$$\begin{aligned}
&= \frac{1}{4} \cdot \frac{(n+3)(n+2)(n+1)}{6} \\
&= \frac{n^3 + 6n^2 + 11n + 6}{24}.
\end{aligned}$$

Taking logarithms of both sides gives $E[X_n] = O(\lg n)$. ■

Exercises

12.4-1

Prove equation (12.3).

12.4-2

Describe a binary search tree on n nodes such that the average depth of a node in the tree is $\Theta(\lg n)$ but the height of the tree is $\omega(\lg n)$. Give an asymptotic upper bound on the height of an n -node binary search tree in which the average depth of a node is $\Theta(\lg n)$.

12.4-3

Show that the notion of a randomly chosen binary search tree on n keys, where each binary search tree of n keys is equally likely to be chosen, is different from the notion of a randomly built binary search tree given in this section. (*Hint*: List the possibilities when $n = 3$.)

12.4-4

Show that the function $f(x) = 2^x$ is convex.

12.4-5 ★

Consider RANDOMIZED-QUICKSORT operating on a sequence of n distinct input numbers. Prove that for any constant $k > 0$, all but $O(1/n^k)$ of the $n!$ input permutations yield an $O(n \lg n)$ running time.

Problems

12-1 Binary search trees with equal keys

Equal keys pose a problem for the implementation of binary search trees.

- a. What is the asymptotic performance of TREE-INSERT when used to insert n items with identical keys into an initially empty binary search tree?

We propose to improve TREE-INSERT by testing before line 5 to determine whether $z.key = x.key$ and by testing before line 11 to determine whether $z.key = y.key$.

If equality holds, we implement one of the following strategies. For each strategy, find the asymptotic performance of inserting n items with identical keys into an initially empty binary search tree. (The strategies are described for line 5, in which we compare the keys of z and x . Substitute y for x to arrive at the strategies for line 11.)

- b.* Keep a boolean flag $x.b$ at node x , and set x to either $x.left$ or $x.right$ based on the value of $x.b$, which alternates between FALSE and TRUE each time we visit x while inserting a node with the same key as x .
- c.* Keep a list of nodes with equal keys at x , and insert z into the list.
- d.* Randomly set x to either $x.left$ or $x.right$. (Give the worst-case performance and informally derive the expected running time.)

12-2 Radix trees

Given two strings $a = a_0a_1 \dots a_p$ and $b = b_0b_1 \dots b_q$, where each a_i and each b_j is in some ordered set of characters, we say that string a is **lexicographically less than** string b if either

1. there exists an integer j , where $0 \leq j \leq \min(p, q)$, such that $a_i = b_i$ for all $i = 0, 1, \dots, j-1$ and $a_j < b_j$, or
2. $p < q$ and $a_i = b_i$ for all $i = 0, 1, \dots, p$.

For example, if a and b are bit strings, then $10100 < 10110$ by rule 1 (letting $j = 3$) and $10100 < 101000$ by rule 2. This ordering is similar to that used in English-language dictionaries.

The **radix tree** data structure shown in Figure 12.5 stores the bit strings 1011, 10, 011, 100, and 0. When searching for a key $a = a_0a_1 \dots a_p$, we go left at a node of depth i if $a_i = 0$ and right if $a_i = 1$. Let S be a set of distinct bit strings whose lengths sum to n . Show how to use a radix tree to sort S lexicographically in $\Theta(n)$ time. For the example in Figure 12.5, the output of the sort should be the sequence 0, 011, 10, 100, 1011.

12-3 Average node depth in a randomly built binary search tree

In this problem, we prove that the average depth of a node in a randomly built binary search tree with n nodes is $O(\lg n)$. Although this result is weaker than that of Theorem 12.4, the technique we shall use reveals a surprising similarity between the building of a binary search tree and the execution of RANDOMIZED-QUICKSORT from Section 7.3.

We define the **total path length** $P(T)$ of a binary tree T as the sum, over all nodes x in T , of the depth of node x , which we denote by $d(x, T)$.

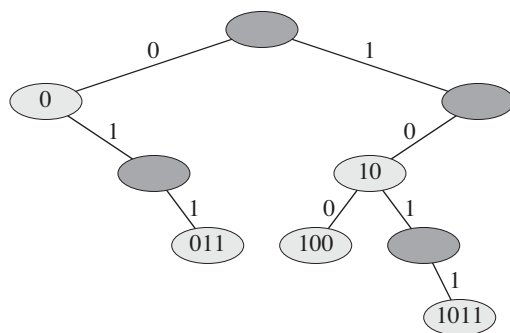


Figure 12.5 A radix tree storing the bit strings 1011, 10, 011, 100, and 0. We can determine each node's key by traversing the simple path from the root to that node. There is no need, therefore, to store the keys in the nodes; the keys appear here for illustrative purposes only. Nodes are heavily shaded if the keys corresponding to them are not in the tree; such nodes are present only to establish a path to other nodes.

a. Argue that the average depth of a node in T is

$$\frac{1}{n} \sum_{x \in T} d(x, T) = \frac{1}{n} P(T) .$$

Thus, we wish to show that the expected value of $P(T)$ is $O(n \lg n)$.

b. Let T_L and T_R denote the left and right subtrees of tree T , respectively. Argue that if T has n nodes, then

$$P(T) = P(T_L) + P(T_R) + n - 1 .$$

c. Let $P(n)$ denote the average total path length of a randomly built binary search tree with n nodes. Show that

$$P(n) = \frac{1}{n} \sum_{i=0}^{n-1} (P(i) + P(n-i-1) + n-1) .$$

d. Show how to rewrite $P(n)$ as

$$P(n) = \frac{2}{n} \sum_{k=1}^{n-1} P(k) + \Theta(n) .$$

e. Recalling the alternative analysis of the randomized version of quicksort given in Problem 7-3, conclude that $P(n) = O(n \lg n)$.

At each recursive invocation of quicksort, we choose a random pivot element to partition the set of elements being sorted. Each node of a binary search tree partitions the set of elements that fall into the subtree rooted at that node.

- f.* Describe an implementation of quicksort in which the comparisons to sort a set of elements are exactly the same as the comparisons to insert the elements into a binary search tree. (The order in which comparisons are made may differ, but the same comparisons must occur.)

12-4 Number of different binary trees

Let b_n denote the number of different binary trees with n nodes. In this problem, you will find a formula for b_n , as well as an asymptotic estimate.

- a.* Show that $b_0 = 1$ and that, for $n \geq 1$,

$$b_n = \sum_{k=0}^{n-1} b_k b_{n-1-k} .$$

- b.* Referring to Problem 4-4 for the definition of a generating function, let $B(x)$ be the generating function

$$B(x) = \sum_{n=0}^{\infty} b_n x^n .$$

Show that $B(x) = xB(x)^2 + 1$, and hence one way to express $B(x)$ in closed form is

$$B(x) = \frac{1}{2x} (1 - \sqrt{1 - 4x}) .$$

The **Taylor expansion** of $f(x)$ around the point $x = a$ is given by

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(a)}{k!} (x - a)^k ,$$

where $f^{(k)}(x)$ is the k th derivative of f evaluated at x .

- c.* Show that

$$b_n = \frac{1}{n+1} \binom{2n}{n}$$

(the n th **Catalan number**) by using the Taylor expansion of $\sqrt{1-4x}$ around $x = 0$. (If you wish, instead of using the Taylor expansion, you may use the generalization of the binomial expansion (C.4) to nonintegral exponents n , where for any real number n and for any integer k , we interpret $\binom{n}{k}$ to be $n(n-1)\cdots(n-k+1)/k!$ if $k \geq 0$, and 0 otherwise.)

d. Show that

$$b_n = \frac{4^n}{\sqrt{\pi n^{3/2}}} (1 + O(1/n)) .$$

Chapter notes

Knuth [211] contains a good discussion of simple binary search trees as well as many variations. Binary search trees seem to have been independently discovered by a number of people in the late 1950s. Radix trees are often called “tries,” which comes from the middle letters in the word *retrieval*. Knuth [211] also discusses them.

Many texts, including the first two editions of this book, have a somewhat simpler method of deleting a node from a binary search tree when both of its children are present. Instead of replacing node z by its successor y , we delete node y but copy its key and satellite data into node z . The downside of this approach is that the node actually deleted might not be the node passed to the delete procedure. If other components of a program maintain pointers to nodes in the tree, they could mistakenly end up with “stale” pointers to nodes that have been deleted. Although the deletion method presented in this edition of this book is a bit more complicated, it guarantees that a call to delete node z deletes node z and only node z .

Section 15.5 will show how to construct an optimal binary search tree when we know the search frequencies before constructing the tree. That is, given the frequencies of searching for each key and the frequencies of searching for values that fall between keys in the tree, we construct a binary search tree for which a set of searches that follows these frequencies examines the minimum number of nodes.

The proof in Section 12.4 that bounds the expected height of a randomly built binary search tree is due to Aslam [24]. Martínez and Roura [243] give randomized algorithms for insertion into and deletion from binary search trees in which the result of either operation is a random binary search tree. Their definition of a random binary search tree differs—only slightly—from that of a randomly built binary search tree in this chapter, however.

13 Red-Black Trees

Chapter 12 showed that a binary search tree of height h can support any of the basic dynamic-set operations—such as SEARCH, PREDECESSOR, SUCCESSOR, MINIMUM, MAXIMUM, INSERT, and DELETE—in $O(h)$ time. Thus, the set operations are fast if the height of the search tree is small. If its height is large, however, the set operations may run no faster than with a linked list. Red-black trees are one of many search-tree schemes that are “balanced” in order to guarantee that basic dynamic-set operations take $O(\lg n)$ time in the worst case.

13.1 Properties of red-black trees

A *red-black tree* is a binary search tree with one extra bit of storage per node: its *color*, which can be either RED or BLACK. By constraining the node colors on any simple path from the root to a leaf, red-black trees ensure that no such path is more than twice as long as any other, so that the tree is approximately *balanced*.

Each node of the tree now contains the attributes *color*, *key*, *left*, *right*, and *p*. If a child or the parent of a node does not exist, the corresponding pointer attribute of the node contains the value NIL. We shall regard these NILs as being pointers to leaves (external nodes) of the binary search tree and the normal, key-bearing nodes as being internal nodes of the tree.

A red-black tree is a binary tree that satisfies the following *red-black properties*:

1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

Figure 13.1(a) shows an example of a red-black tree.

As a matter of convenience in dealing with boundary conditions in red-black tree code, we use a single sentinel to represent NIL (see page 238). For a red-black tree T , the sentinel $T.nil$ is an object with the same attributes as an ordinary node in the tree. Its *color* attribute is BLACK, and its other attributes—*p*, *left*, *right*, and *key*—can take on arbitrary values. As Figure 13.1(b) shows, all pointers to NIL are replaced by pointers to the sentinel $T.nil$.

We use the sentinel so that we can treat a NIL child of a node x as an ordinary node whose parent is x . Although we instead could add a distinct sentinel node for each NIL in the tree, so that the parent of each NIL is well defined, that approach would waste space. Instead, we use the one sentinel $T.nil$ to represent all the NILs—all leaves and the root's parent. The values of the attributes *p*, *left*, *right*, and *key* of the sentinel are immaterial, although we may set them during the course of a procedure for our convenience.

We generally confine our interest to the internal nodes of a red-black tree, since they hold the key values. In the remainder of this chapter, we omit the leaves when we draw red-black trees, as shown in Figure 13.1(c).

We call the number of black nodes on any simple path from, but not including, a node x down to a leaf the **black-height** of the node, denoted $bh(x)$. By property 5, the notion of black-height is well defined, since all descending simple paths from the node have the same number of black nodes. We define the black-height of a red-black tree to be the black-height of its root.

The following lemma shows why red-black trees make good search trees.

Lemma 13.1

A red-black tree with n internal nodes has height at most $2 \lg(n + 1)$.

Proof We start by showing that the subtree rooted at any node x contains at least $2^{bh(x)} - 1$ internal nodes. We prove this claim by induction on the height of x . If the height of x is 0, then x must be a leaf ($T.nil$), and the subtree rooted at x indeed contains at least $2^{bh(x)} - 1 = 2^0 - 1 = 0$ internal nodes. For the inductive step, consider a node x that has positive height and is an internal node with two children. Each child has a black-height of either $bh(x)$ or $bh(x) - 1$, depending on whether its color is red or black, respectively. Since the height of a child of x is less than the height of x itself, we can apply the inductive hypothesis to conclude that each child has at least $2^{bh(x)-1} - 1$ internal nodes. Thus, the subtree rooted at x contains at least $(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$ internal nodes, which proves the claim.

To complete the proof of the lemma, let h be the height of the tree. According to property 4, at least half the nodes on any simple path from the root to a leaf, not

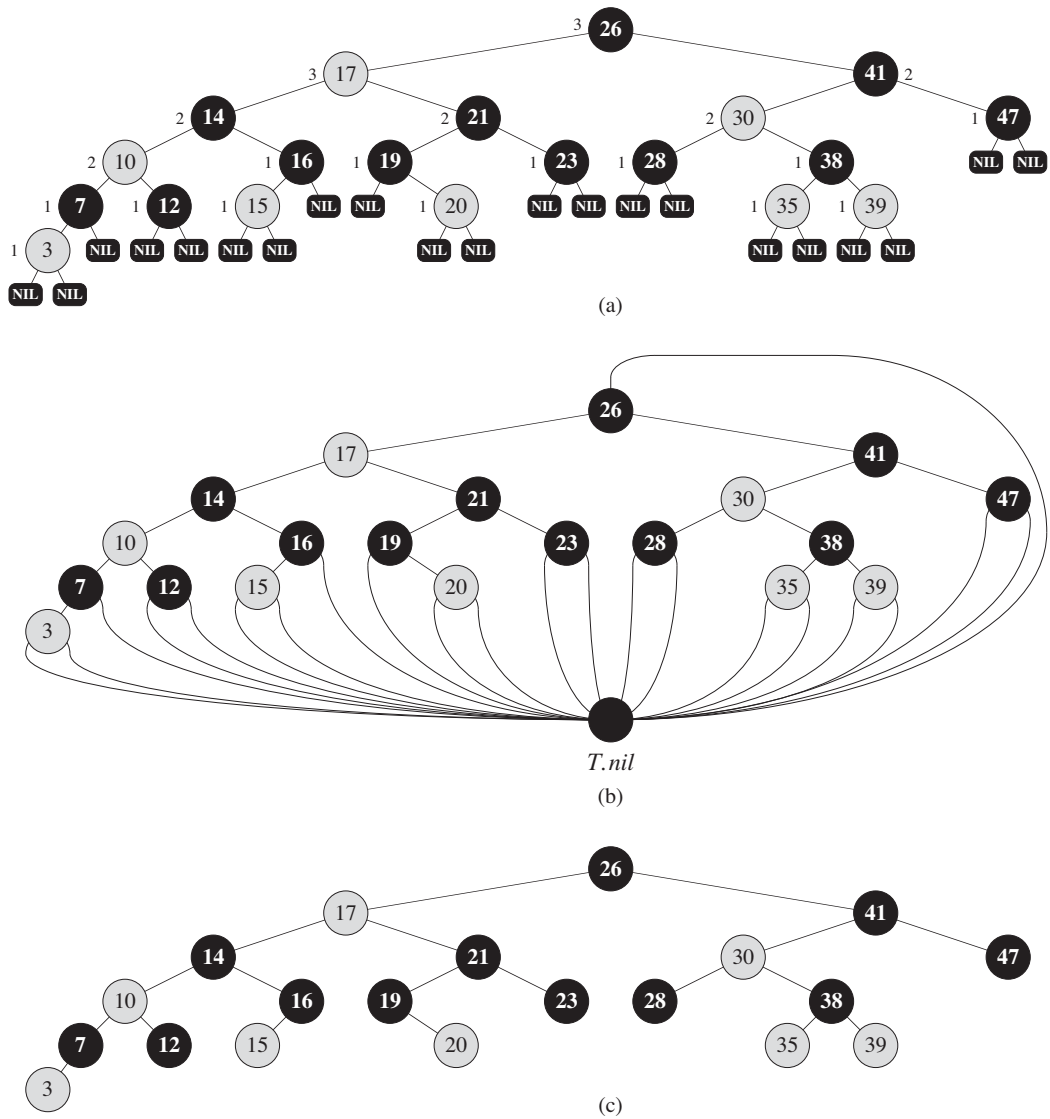


Figure 13.1 A red-black tree with black nodes darkened and red nodes shaded. Every node in a red-black tree is either red or black, the children of a red node are both black, and every simple path from a node to a descendant leaf contains the same number of black nodes. (a) Every leaf, shown as a NIL, is black. Each non-NIL node is marked with its black-height; NILs have black-height 0. (b) The same red-black tree but with each NIL replaced by the single sentinel $T.nil$, which is always black, and with black-heights omitted. The root's parent is also the sentinel. (c) The same red-black tree but with leaves and the root's parent omitted entirely. We shall use this drawing style in the remainder of this chapter.

including the root, must be black. Consequently, the black-height of the root must be at least $h/2$; thus,

$$n \geq 2^{h/2} - 1.$$

Moving the 1 to the left-hand side and taking logarithms on both sides yields $\lg(n + 1) \geq h/2$, or $h \leq 2\lg(n + 1)$. ■

As an immediate consequence of this lemma, we can implement the dynamic-set operations SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR in $O(\lg n)$ time on red-black trees, since each can run in $O(h)$ time on a binary search tree of height h (as shown in Chapter 12) and any red-black tree on n nodes is a binary search tree with height $O(\lg n)$. (Of course, references to NIL in the algorithms of Chapter 12 would have to be replaced by $T.nil$.) Although the algorithms TREE-INSERT and TREE-DELETE from Chapter 12 run in $O(\lg n)$ time when given a red-black tree as input, they do not directly support the dynamic-set operations INSERT and DELETE, since they do not guarantee that the modified binary search tree will be a red-black tree. We shall see in Sections 13.3 and 13.4, however, how to support these two operations in $O(\lg n)$ time.

Exercises

13.1-1

In the style of Figure 13.1(a), draw the complete binary search tree of height 3 on the keys $\{1, 2, \dots, 15\}$. Add the NIL leaves and color the nodes in three different ways such that the black-heights of the resulting red-black trees are 2, 3, and 4.

13.1-2

Draw the red-black tree that results after TREE-INSERT is called on the tree in Figure 13.1 with key 36. If the inserted node is colored red, is the resulting tree a red-black tree? What if it is colored black?

13.1-3

Let us define a **relaxed red-black tree** as a binary search tree that satisfies red-black properties 1, 3, 4, and 5. In other words, the root may be either red or black. Consider a relaxed red-black tree T whose root is red. If we color the root of T black but make no other changes to T , is the resulting tree a red-black tree?

13.1-4

Suppose that we “absorb” every red node in a red-black tree into its black parent, so that the children of the red node become children of the black parent. (Ignore what happens to the keys.) What are the possible degrees of a black node after all

its red children are absorbed? What can you say about the depths of the leaves of the resulting tree?

13.1-5

Show that the longest simple path from a node x in a red-black tree to a descendant leaf has length at most twice that of the shortest simple path from node x to a descendant leaf.

13.1-6

What is the largest possible number of internal nodes in a red-black tree with black-height k ? What is the smallest possible number?

13.1-7

Describe a red-black tree on n keys that realizes the largest possible ratio of red internal nodes to black internal nodes. What is this ratio? What tree has the smallest possible ratio, and what is the ratio?

13.2 Rotations

The search-tree operations TREE-INSERT and TREE-DELETE, when run on a red-black tree with n keys, take $O(\lg n)$ time. Because they modify the tree, the result may violate the red-black properties enumerated in Section 13.1. To restore these properties, we must change the colors of some of the nodes in the tree and also change the pointer structure.

We change the pointer structure through *rotation*, which is a local operation in a search tree that preserves the binary-search-tree property. Figure 13.2 shows the two kinds of rotations: left rotations and right rotations. When we do a left rotation on a node x , we assume that its right child y is not $T.nil$; x may be any node in the tree whose right child is not $T.nil$. The left rotation “pivots” around the link from x to y . It makes y the new root of the subtree, with x as y ’s left child and y ’s left child as x ’s right child.

The pseudocode for LEFT-ROTATE assumes that $x.right \neq T.nil$ and that the root’s parent is $T.nil$.

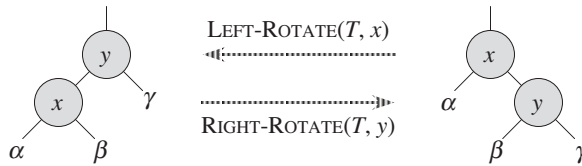


Figure 13.2 The rotation operations on a binary search tree. The operation $\text{LEFT-ROTATE}(T, x)$ transforms the configuration of the two nodes on the right into the configuration on the left by changing a constant number of pointers. The inverse operation $\text{RIGHT-ROTATE}(T, y)$ transforms the configuration on the left into the configuration on the right. The letters α , β , and γ represent arbitrary subtrees. A rotation operation preserves the binary-search-tree property: the keys in α precede $x.\text{key}$, which precedes the keys in β , which precedes $y.\text{key}$, which precedes the keys in γ .

$\text{LEFT-ROTATE}(T, x)$

```

1  y = x.right           // set y
2  x.right = y.left      // turn y's left subtree into x's right subtree
3  if y.left ≠ T.nil
4      y.left.p = x
5  y.p = x.p             // link x's parent to y
6  if x.p == T.nil
7      T.root = y
8  elseif x == x.p.left
9      x.p.left = y
10 else x.p.right = y
11 y.left = x            // put x on y's left
12 x.p = y

```

Figure 13.3 shows an example of how LEFT-ROTATE modifies a binary search tree. The code for RIGHT-ROTATE is symmetric. Both LEFT-ROTATE and RIGHT-ROTATE run in $O(1)$ time. Only pointers are changed by a rotation; all other attributes in a node remain the same.

Exercises

13.2-1

Write pseudocode for RIGHT-ROTATE .

13.2-2

Argue that in every n -node binary search tree, there are exactly $n - 1$ possible rotations.

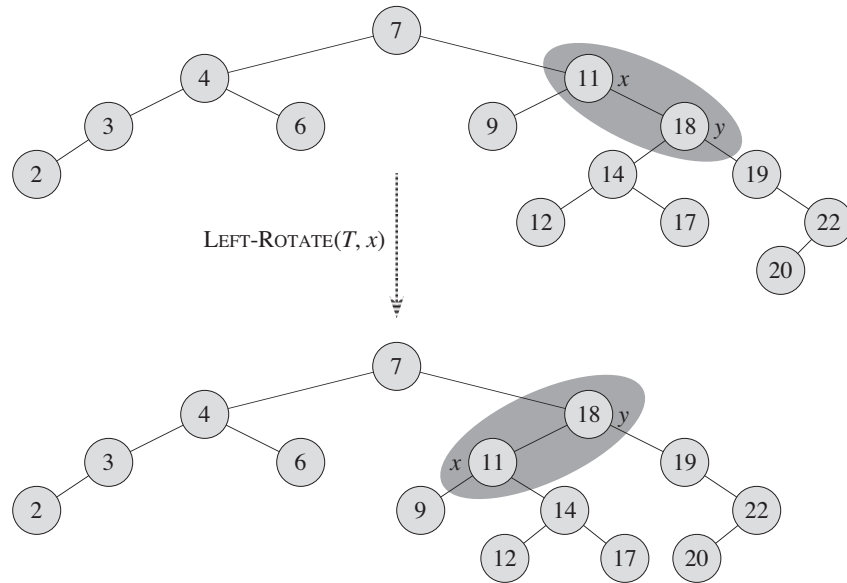


Figure 13.3 An example of how the procedure $\text{LEFT-ROTATE}(T, x)$ modifies a binary search tree. Inorder tree walks of the input tree and the modified tree produce the same listing of key values.

13.2-3

Let a , b , and c be arbitrary nodes in subtrees α , β , and γ , respectively, in the left tree of Figure 13.2. How do the depths of a , b , and c change when a left rotation is performed on node x in the figure?

13.2-4

Show that any arbitrary n -node binary search tree can be transformed into any other arbitrary n -node binary search tree using $O(n)$ rotations. (*Hint*: First show that at most $n - 1$ right rotations suffice to transform the tree into a right-going chain.)

13.2-5 ★

We say that a binary search tree T_1 can be **right-converted** to binary search tree T_2 if it is possible to obtain T_2 from T_1 via a series of calls to RIGHT-ROTATE . Give an example of two trees T_1 and T_2 such that T_1 cannot be right-converted to T_2 . Then, show that if a tree T_1 can be right-converted to T_2 , it can be right-converted using $O(n^2)$ calls to RIGHT-ROTATE .

13.3 Insertion

We can insert a node into an n -node red-black tree in $O(\lg n)$ time. To do so, we use a slightly modified version of the TREE-INSERT procedure (Section 12.3) to insert node z into the tree T as if it were an ordinary binary search tree, and then we color z red. (Exercise 13.3-1 asks you to explain why we choose to make node z red rather than black.) To guarantee that the red-black properties are preserved, we then call an auxiliary procedure RB-INSERT-FIXUP to recolor nodes and perform rotations. The call RB-INSERT(T, z) inserts node z , whose *key* is assumed to have already been filled in, into the red-black tree T .

```

RB-INSERT( $T, z$ )
1   $y = T.nil$ 
2   $x = T.root$ 
3  while  $x \neq T.nil$ 
4       $y = x$ 
5      if  $z.key < x.key$ 
6           $x = x.left$ 
7      else  $x = x.right$ 
8   $z.p = y$ 
9  if  $y == T.nil$ 
10      $T.root = z$ 
11 elseif  $z.key < y.key$ 
12      $y.left = z$ 
13 else  $y.right = z$ 
14  $z.left = T.nil$ 
15  $z.right = T.nil$ 
16  $z.color = RED$ 
17 RB-INSERT-FIXUP( $T, z$ )

```

The procedures TREE-INSERT and RB-INSERT differ in four ways. First, all instances of NIL in TREE-INSERT are replaced by $T.nil$. Second, we set $z.left$ and $z.right$ to $T.nil$ in lines 14–15 of RB-INSERT, in order to maintain the proper tree structure. Third, we color z red in line 16. Fourth, because coloring z red may cause a violation of one of the red-black properties, we call RB-INSERT-FIXUP(T, z) in line 17 of RB-INSERT to restore the red-black properties.

RB-INSERT-FIXUP(T, z)

```

1  while  $z.p.color == \text{RED}$ 
2      if  $z.p == z.p.p.left$ 
3           $y = z.p.p.right$ 
4          if  $y.color == \text{RED}$ 
5               $z.p.color = \text{BLACK}$                                 // case 1
6               $y.color = \text{BLACK}$                                 // case 1
7               $z.p.p.color = \text{RED}$                                 // case 1
8               $z = z.p.p$                                         // case 1
9          else if  $z == z.p.right$ 
10              $z = z.p$                                           // case 2
11             LEFT-ROTATE( $T, z$ )                                // case 2
12              $z.p.color = \text{BLACK}$                                 // case 3
13              $z.p.p.color = \text{RED}$                                 // case 3
14             RIGHT-ROTATE( $T, z.p.p$ )                            // case 3
15      else (same as then clause
              with “right” and “left” exchanged)
16   $T.root.color = \text{BLACK}$ 

```

To understand how RB-INSERT-FIXUP works, we shall break our examination of the code into three major steps. First, we shall determine what violations of the red-black properties are introduced in RB-INSERT when node z is inserted and colored red. Second, we shall examine the overall goal of the **while** loop in lines 1–15. Finally, we shall explore each of the three cases¹ within the **while** loop’s body and see how they accomplish the goal. Figure 13.4 shows how RB-INSERT-FIXUP operates on a sample red-black tree.

Which of the red-black properties might be violated upon the call to RB-INSERT-FIXUP? Property 1 certainly continues to hold, as does property 3, since both children of the newly inserted red node are the sentinel $T.nil$. Property 5, which says that the number of black nodes is the same on every simple path from a given node, is satisfied as well, because node z replaces the (black) sentinel, and node z is red with sentinel children. Thus, the only properties that might be violated are property 2, which requires the root to be black, and property 4, which says that a red node cannot have a red child. Both possible violations are due to z being colored red. Property 2 is violated if z is the root, and property 4 is violated if z ’s parent is red. Figure 13.4(a) shows a violation of property 4 after the node z has been inserted.

¹Case 2 falls through into case 3, and so these two cases are not mutually exclusive.

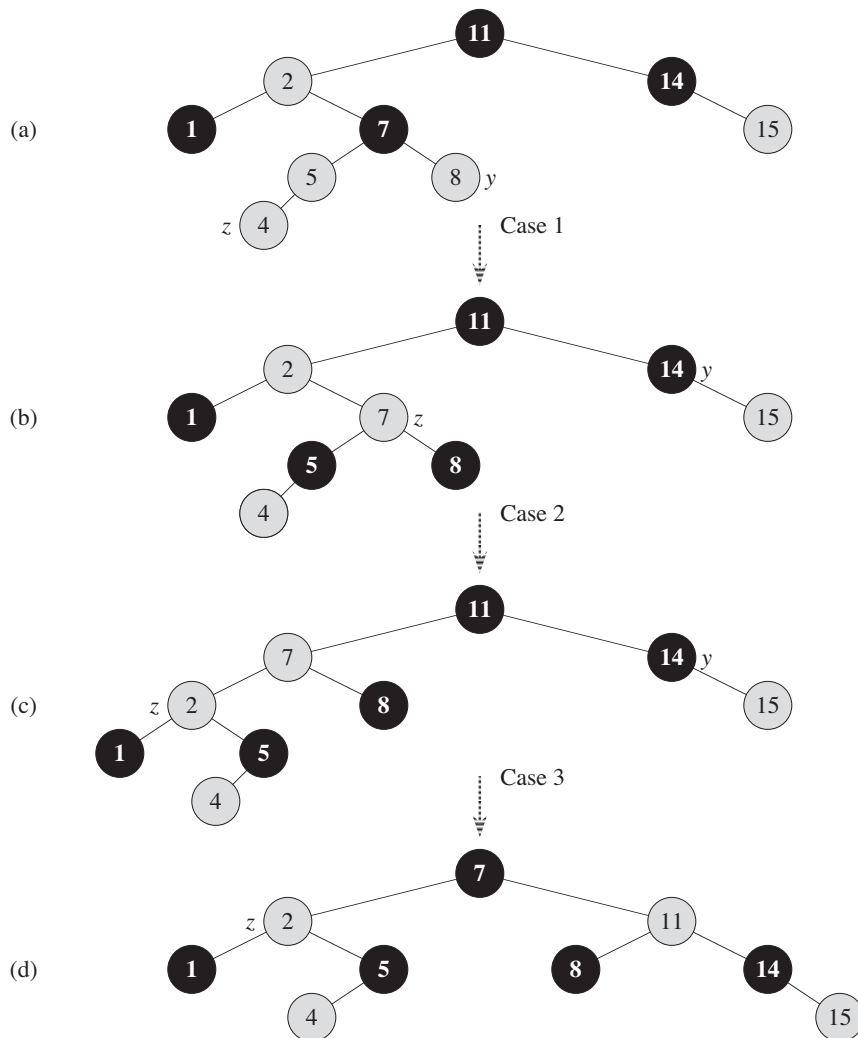


Figure 13.4 The operation of RB-INSERT-FIXUP. **(a)** A node z after insertion. Because both z and its parent $z.p$ are red, a violation of property 4 occurs. Since z 's uncle y is red, case 1 in the code applies. We recolor nodes and move the pointer z up the tree, resulting in the tree shown in **(b)**. Once again, z and its parent are both red, but z 's uncle y is black. Since z is the right child of $z.p$, case 2 applies. We perform a left rotation, and the tree that results is shown in **(c)**. Now, z is the left child of its parent, and case 3 applies. Recoloring and right rotation yield the tree in **(d)**, which is a legal red-black tree.

The **while** loop in lines 1–15 maintains the following three-part invariant at the start of each iteration of the loop:

- a. Node z is red.
- b. If $z.p$ is the root, then $z.p$ is black.
- c. If the tree violates any of the red-black properties, then it violates at most one of them, and the violation is of either property 2 or property 4. If the tree violates property 2, it is because z is the root and is red. If the tree violates property 4, it is because both z and $z.p$ are red.

Part (c), which deals with violations of red-black properties, is more central to showing that RB-INSERT-FIXUP restores the red-black properties than parts (a) and (b), which we use along the way to understand situations in the code. Because we'll be focusing on node z and nodes near it in the tree, it helps to know from part (a) that z is red. We shall use part (b) to show that the node $z.p.p$ exists when we reference it in lines 2, 3, 7, 8, 13, and 14.

Recall that we need to show that a loop invariant is true prior to the first iteration of the loop, that each iteration maintains the loop invariant, and that the loop invariant gives us a useful property at loop termination.

We start with the initialization and termination arguments. Then, as we examine how the body of the loop works in more detail, we shall argue that the loop maintains the invariant upon each iteration. Along the way, we shall also demonstrate that each iteration of the loop has two possible outcomes: either the pointer z moves up the tree, or we perform some rotations and then the loop terminates.

Initialization: Prior to the first iteration of the loop, we started with a red-black tree with no violations, and we added a red node z . We show that each part of the invariant holds at the time RB-INSERT-FIXUP is called:

- a. When RB-INSERT-FIXUP is called, z is the red node that was added.
- b. If $z.p$ is the root, then $z.p$ started out black and did not change prior to the call of RB-INSERT-FIXUP.
- c. We have already seen that properties 1, 3, and 5 hold when RB-INSERT-FIXUP is called.

If the tree violates property 2, then the red root must be the newly added node z , which is the only internal node in the tree. Because the parent and both children of z are the sentinel, which is black, the tree does not also violate property 4. Thus, this violation of property 2 is the only violation of red-black properties in the entire tree.

If the tree violates property 4, then, because the children of node z are black sentinels and the tree had no other violations prior to z being added, the

violation must be because both z and $z.p$ are red. Moreover, the tree violates no other red-black properties.

Termination: When the loop terminates, it does so because $z.p$ is black. (If z is the root, then $z.p$ is the sentinel $T.nil$, which is black.) Thus, the tree does not violate property 4 at loop termination. By the loop invariant, the only property that might fail to hold is property 2. Line 16 restores this property, too, so that when RB-INSERT-FIXUP terminates, all the red-black properties hold.

Maintenance: We actually need to consider six cases in the **while** loop, but three of them are symmetric to the other three, depending on whether line 2 determines z 's parent $z.p$ to be a left child or a right child of z 's grandparent $z.p.p$. We have given the code only for the situation in which $z.p$ is a left child. The node $z.p.p$ exists, since by part (b) of the loop invariant, if $z.p$ is the root, then $z.p$ is black. Since we enter a loop iteration only if $z.p$ is red, we know that $z.p$ cannot be the root. Hence, $z.p.p$ exists.

We distinguish case 1 from cases 2 and 3 by the color of z 's parent's sibling, or "uncle." Line 3 makes y point to z 's uncle $z.p.p.right$, and line 4 tests y 's color. If y is red, then we execute case 1. Otherwise, control passes to cases 2 and 3. In all three cases, z 's grandparent $z.p.p$ is black, since its parent $z.p$ is red, and property 4 is violated only between z and $z.p$.

Case 1: z 's uncle y is red

Figure 13.5 shows the situation for case 1 (lines 5–8), which occurs when both $z.p$ and y are red. Because $z.p.p$ is black, we can color both $z.p$ and y black, thereby fixing the problem of z and $z.p$ both being red, and we can color $z.p.p$ red, thereby maintaining property 5. We then repeat the **while** loop with $z.p.p$ as the new node z . The pointer z moves up two levels in the tree.

Now, we show that case 1 maintains the loop invariant at the start of the next iteration. We use z to denote node z in the current iteration, and $z' = z.p.p$ to denote the node that will be called node z at the test in line 1 upon the next iteration.

- Because this iteration colors $z.p.p$ red, node z' is red at the start of the next iteration.
- The node $z'.p$ is $z.p.p.p$ in this iteration, and the color of this node does not change. If this node is the root, it was black prior to this iteration, and it remains black at the start of the next iteration.
- We have already argued that case 1 maintains property 5, and it does not introduce a violation of properties 1 or 3.

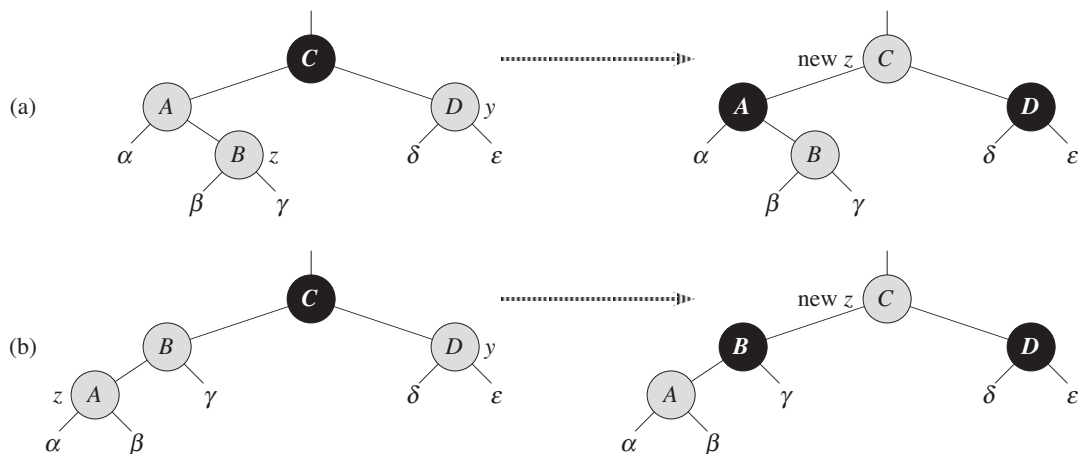


Figure 13.5 Case 1 of the procedure RB-INSERT-FIXUP. Property 4 is violated, since z and its parent $z.p$ are both red. We take the same action whether (a) z is a right child or (b) z is a left child. Each of the subtrees α , β , γ , δ , and ϵ has a black root, and each has the same black-height. The code for case 1 changes the colors of some nodes, preserving property 5: all downward simple paths from a node to a leaf have the same number of blacks. The **while** loop continues with node z 's grandparent $z.p.p$ as the new z . Any violation of property 4 can now occur only between the new z , which is red, and its parent, if it is red as well.

If node z' is the root at the start of the next iteration, then case 1 corrected the lone violation of property 4 in this iteration. Since z' is red and it is the root, property 2 becomes the only one that is violated, and this violation is due to z' .

If node z' is not the root at the start of the next iteration, then case 1 has not created a violation of property 2. Case 1 corrected the lone violation of property 4 that existed at the start of this iteration. It then made z' red and left $z'.p$ alone. If $z'.p$ was black, there is no violation of property 4. If $z'.p$ was red, coloring z' red created one violation of property 4 between z' and $z'.p$.

Case 2: z 's uncle y is black and z is a right child

Case 3: z 's uncle y is black and z is a left child

In cases 2 and 3, the color of z 's uncle y is black. We distinguish the two cases according to whether z is a right or left child of $z.p$. Lines 10–11 constitute case 2, which is shown in Figure 13.6 together with case 3. In case 2, node z is a right child of its parent. We immediately use a left rotation to transform the situation into case 3 (lines 12–14), in which node z is a left child. Because

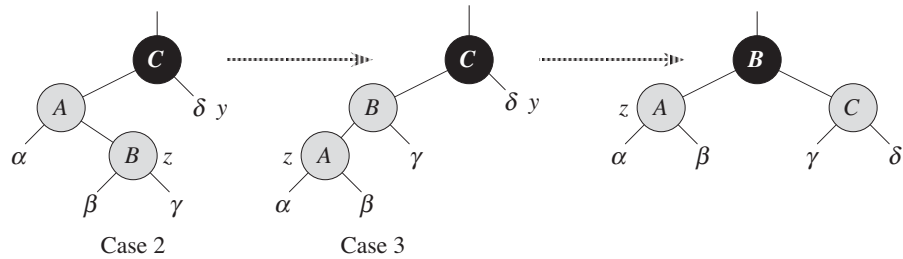


Figure 13.6 Cases 2 and 3 of the procedure RB-INSERT-FIXUP. As in case 1, property 4 is violated in either case 2 or case 3 because z and its parent $z.p$ are both red. Each of the subtrees α , β , γ , and δ has a black root (α , β , and γ from property 4, and δ because otherwise we would be in case 1), and each has the same black-height. We transform case 2 into case 3 by a left rotation, which preserves property 5: all downward simple paths from a node to a leaf have the same number of blacks. Case 3 causes some color changes and a right rotation, which also preserve property 5. The **while** loop then terminates, because property 4 is satisfied: there are no longer two red nodes in a row.

both z and $z.p$ are red, the rotation affects neither the black-height of nodes nor property 5. Whether we enter case 3 directly or through case 2, z 's uncle y is black, since otherwise we would have executed case 1. Additionally, the node $z.p.p$ exists, since we have argued that this node existed at the time that lines 2 and 3 were executed, and after moving z up one level in line 10 and then down one level in line 11, the identity of $z.p.p$ remains unchanged. In case 3, we execute some color changes and a right rotation, which preserve property 5, and then, since we no longer have two red nodes in a row, we are done. The **while** loop does not iterate another time, since $z.p$ is now black.

We now show that cases 2 and 3 maintain the loop invariant. (As we have just argued, $z.p$ will be black upon the next test in line 1, and the loop body will not execute again.)

- Case 2 makes z point to $z.p$, which is red. No further change to z or its color occurs in cases 2 and 3.
- Case 3 makes $z.p$ black, so that if $z.p$ is the root at the start of the next iteration, it is black.
- As in case 1, properties 1, 3, and 5 are maintained in cases 2 and 3.

Since node z is not the root in cases 2 and 3, we know that there is no violation of property 2. Cases 2 and 3 do not introduce a violation of property 2, since the only node that is made red becomes a child of a black node by the rotation in case 3.

Cases 2 and 3 correct the lone violation of property 4, and they do not introduce another violation.

Having shown that each iteration of the loop maintains the invariant, we have shown that RB-INSERT-FIXUP correctly restores the red-black properties.

Analysis

What is the running time of RB-INSERT? Since the height of a red-black tree on n nodes is $O(\lg n)$, lines 1–16 of RB-INSERT take $O(\lg n)$ time. In RB-INSERT-FIXUP, the **while** loop repeats only if case 1 occurs, and then the pointer z moves two levels up the tree. The total number of times the **while** loop can be executed is therefore $O(\lg n)$. Thus, RB-INSERT takes a total of $O(\lg n)$ time. Moreover, it never performs more than two rotations, since the **while** loop terminates if case 2 or case 3 is executed.

Exercises

13.3-1

In line 16 of RB-INSERT, we set the color of the newly inserted node z to red. Observe that if we had chosen to set z 's color to black, then property 4 of a red-black tree would not be violated. Why didn't we choose to set z 's color to black?

13.3-2

Show the red-black trees that result after successively inserting the keys 41, 38, 31, 12, 19, 8 into an initially empty red-black tree.

13.3-3

Suppose that the black-height of each of the subtrees $\alpha, \beta, \gamma, \delta, \varepsilon$ in Figures 13.5 and 13.6 is k . Label each node in each figure with its black-height to verify that the indicated transformation preserves property 5.

13.3-4

Professor Teach is concerned that RB-INSERT-FIXUP might set $T.nil.color$ to RED, in which case the test in line 1 would not cause the loop to terminate when z is the root. Show that the professor's concern is unfounded by arguing that RB-INSERT-FIXUP never sets $T.nil.color$ to RED.

13.3-5

Consider a red-black tree formed by inserting n nodes with RB-INSERT. Argue that if $n > 1$, the tree has at least one red node.

13.3-6

Suggest how to implement RB-INSERT efficiently if the representation for red-black trees includes no storage for parent pointers.

13.4 Deletion

Like the other basic operations on an n -node red-black tree, deletion of a node takes time $O(\lg n)$. Deleting a node from a red-black tree is a bit more complicated than inserting a node.

The procedure for deleting a node from a red-black tree is based on the TREE-DELETE procedure (Section 12.3). First, we need to customize the TRANSPLANT subroutine that TREE-DELETE calls so that it applies to a red-black tree:

RB-TRANSPLANT(T, u, v)

```

1  if  $u.p == T.nil$ 
2       $T.root = v$ 
3  elseif  $u == u.p.left$ 
4       $u.p.left = v$ 
5  else  $u.p.right = v$ 
6   $v.p = u.p$ 
```

The procedure RB-TRANSPLANT differs from TRANSPLANT in two ways. First, line 1 references the sentinel $T.nil$ instead of NIL. Second, the assignment to $v.p$ in line 6 occurs unconditionally: we can assign to $v.p$ even if v points to the sentinel. In fact, we shall exploit the ability to assign to $v.p$ when $v = T.nil$.

The procedure RB-DELETE is like the TREE-DELETE procedure, but with additional lines of pseudocode. Some of the additional lines keep track of a node y that might cause violations of the red-black properties. When we want to delete node z and z has fewer than two children, then z is removed from the tree, and we want y to be z . When z has two children, then y should be z 's successor, and y moves into z 's position in the tree. We also remember y 's color before it is removed from or moved within the tree, and we keep track of the node x that moves into y 's original position in the tree, because node x might also cause violations of the red-black properties. After deleting node z , RB-DELETE calls an auxiliary procedure RB-DELETE-FIXUP, which changes colors and performs rotations to restore the red-black properties.


```

RB-DELETE( $T, z$ )
1   $y = z$ 
2   $y\text{-original-color} = y.\text{color}$ 
3  if  $z.\text{left} == T.\text{nil}$ 
4       $x = z.\text{right}$ 
5      RB-TRANSPLANT( $T, z, z.\text{right}$ )
6  elseif  $z.\text{right} == T.\text{nil}$ 
7       $x = z.\text{left}$ 
8      RB-TRANSPLANT( $T, z, z.\text{left}$ )
9  else  $y = \text{TREE-MINIMUM}(z.\text{right})$ 
10      $y\text{-original-color} = y.\text{color}$ 
11      $x = y.\text{right}$ 
12     if  $y.p == z$ 
13          $x.p = y$ 
14     else RB-TRANSPLANT( $T, y, y.\text{right}$ )
15          $y.\text{right} = z.\text{right}$ 
16          $y.\text{right}.p = y$ 
17     RB-TRANSPLANT( $T, z, y$ )
18      $y.\text{left} = z.\text{left}$ 
19      $y.\text{left}.p = y$ 
20      $y.\text{color} = z.\text{color}$ 
21     if  $y\text{-original-color} == \text{BLACK}$ 
22         RB-DELETE-FIXUP( $T, x$ )

```

Although RB-DELETE contains almost twice as many lines of pseudocode as TREE-DELETE, the two procedures have the same basic structure. You can find each line of TREE-DELETE within RB-DELETE (with the changes of replacing NIL by $T.\text{nil}$ and replacing calls to TRANSPLANT by calls to RB-TRANSPLANT), executed under the same conditions.

Here are the other differences between the two procedures:

- We maintain node y as the node either removed from the tree or moved within the tree. Line 1 sets y to point to node z when z has fewer than two children and is therefore removed. When z has two children, line 9 sets y to point to z 's successor, just as in TREE-DELETE, and y will move into z 's position in the tree.
- Because node y 's color might change, the variable $y\text{-original-color}$ stores y 's color before any changes occur. Lines 2 and 10 set this variable immediately after assignments to y . When z has two children, then $y \neq z$ and node y moves into node z 's original position in the red-black tree; line 20 gives y the same color as z . We need to save y 's original color in order to test it at the

end of RB-DELETE; if it was black, then removing or moving y could cause violations of the red-black properties.

- As discussed, we keep track of the node x that moves into node y 's original position. The assignments in lines 4, 7, and 11 set x to point to either y 's only child or, if y has no children, the sentinel $T.nil$. (Recall from Section 12.3 that y has no left child.)
- Since node x moves into node y 's original position, the attribute $x.p$ is always set to point to the original position in the tree of y 's parent, even if x is, in fact, the sentinel $T.nil$. Unless z is y 's original parent (which occurs only when z has two children and its successor y is z 's right child), the assignment to $x.p$ takes place in line 6 of RB-TRANSPLANT. (Observe that when RB-TRANSPLANT is called in lines 5, 8, or 14, the second parameter passed is the same as x .)

When y 's original parent is z , however, we do not want $x.p$ to point to y 's original parent, since we are removing that node from the tree. Because node y will move up to take z 's position in the tree, setting $x.p$ to y in line 13 causes $x.p$ to point to the original position of y 's parent, even if $x = T.nil$.

- Finally, if node y was black, we might have introduced one or more violations of the red-black properties, and so we call RB-DELETE-FIXUP in line 22 to restore the red-black properties. If y was red, the red-black properties still hold when y is removed or moved, for the following reasons:

1. No black-heights in the tree have changed.
2. No red nodes have been made adjacent. Because y takes z 's place in the tree, along with z 's color, we cannot have two adjacent red nodes at y 's new position in the tree. In addition, if y was not z 's right child, then y 's original right child x replaces y in the tree. If y is red, then x must be black, and so replacing y by x cannot cause two red nodes to become adjacent.
3. Since y could not have been the root if it was red, the root remains black.

If node y was black, three problems may arise, which the call of RB-DELETE-FIXUP will remedy. First, if y had been the root and a red child of y becomes the new root, we have violated property 2. Second, if both x and $x.p$ are red, then we have violated property 4. Third, moving y within the tree causes any simple path that previously contained y to have one fewer black node. Thus, property 5 is now violated by any ancestor of y in the tree. We can correct the violation of property 5 by saying that node x , now occupying y 's original position, has an "extra" black. That is, if we add 1 to the count of black nodes on any simple path that contains x , then under this interpretation, property 5 holds. When we remove or move the black node y , we "push" its blackness onto node x . The problem is that now node x is neither red nor black, thereby violating property 1. Instead,

node x is either “doubly black” or “red-and-black,” and it contributes either 2 or 1, respectively, to the count of black nodes on simple paths containing x . The *color* attribute of x will still be either RED (if x is red-and-black) or BLACK (if x is doubly black). In other words, the extra black on a node is reflected in x ’s pointing to the node rather than in the *color* attribute.

We can now see the procedure RB-DELETE-FIXUP and examine how it restores the red-black properties to the search tree.

RB-DELETE-FIXUP(T, x)

```

1  while  $x \neq T.root$  and  $x.color == BLACK$ 
2      if  $x == x.p.left$ 
3           $w = x.p.right$ 
4          if  $w.color == RED$ 
5               $w.color = BLACK$                                 // case 1
6               $x.p.color = RED$                                 // case 1
7              LEFT-ROTATE( $T, x.p$ )                            // case 1
8               $w = x.p.right$                                     // case 1
9          if  $w.left.color == BLACK$  and  $w.right.color == BLACK$ 
10              $w.color = RED$                                     // case 2
11              $x = x.p$                                           // case 2
12         else if  $w.right.color == BLACK$ 
13              $w.left.color = BLACK$                             // case 3
14              $w.color = RED$                                     // case 3
15             RIGHT-ROTATE( $T, w$ )                                // case 3
16              $w = x.p.right$                                     // case 3
17              $w.color = x.p.color$                               // case 4
18              $x.p.color = BLACK$                                 // case 4
19              $w.right.color = BLACK$                             // case 4
20             LEFT-ROTATE( $T, x.p$ )                                // case 4
21              $x = T.root$                                         // case 4
22         else (same as then clause with “right” and “left” exchanged)
23      $x.color = BLACK$ 

```

The procedure RB-DELETE-FIXUP restores properties 1, 2, and 4. Exercises 13.4-1 and 13.4-2 ask you to show that the procedure restores properties 2 and 4, and so in the remainder of this section, we shall focus on property 1. The goal of the **while** loop in lines 1–22 is to move the extra black up the tree until

1. x points to a red-and-black node, in which case we color x (singly) black in line 23;
2. x points to the root, in which case we simply “remove” the extra black; or
3. having performed suitable rotations and recolorings, we exit the loop.

Within the **while** loop, x always points to a nonroot doubly black node. We determine in line 2 whether x is a left child or a right child of its parent $x.p$. (We have given the code for the situation in which x is a left child; the situation in which x is a right child—line 22—is symmetric.) We maintain a pointer w to the sibling of x . Since node x is doubly black, node w cannot be $T.nil$, because otherwise, the number of blacks on the simple path from $x.p$ to the (singly black) leaf w would be smaller than the number on the simple path from $x.p$ to x .

The four cases² in the code appear in Figure 13.7. Before examining each case in detail, let's look more generally at how we can verify that the transformation in each of the cases preserves property 5. The key idea is that in each case, the transformation applied preserves the number of black nodes (including x 's extra black) from (and including) the root of the subtree shown to each of the subtrees $\alpha, \beta, \dots, \zeta$. Thus, if property 5 holds prior to the transformation, it continues to hold afterward. For example, in Figure 13.7(a), which illustrates case 1, the number of black nodes from the root to either subtree α or β is 3, both before and after the transformation. (Again, remember that node x adds an extra black.) Similarly, the number of black nodes from the root to any of γ, δ, ϵ , and ζ is 2, both before and after the transformation. In Figure 13.7(b), the counting must involve the value c of the *color* attribute of the root of the subtree shown, which can be either RED or BLACK. If we define $\text{count}(\text{RED}) = 0$ and $\text{count}(\text{BLACK}) = 1$, then the number of black nodes from the root to α is $2 + \text{count}(c)$, both before and after the transformation. In this case, after the transformation, the new node x has *color* attribute c , but this node is really either red-and-black (if $c = \text{RED}$) or doubly black (if $c = \text{BLACK}$). You can verify the other cases similarly (see Exercise 13.4-5).

Case 1: x 's sibling w is red

Case 1 (lines 5–8 of RB-DELETE-FIXUP and Figure 13.7(a)) occurs when node w , the sibling of node x , is red. Since w must have black children, we can switch the colors of w and $x.p$ and then perform a left-rotation on $x.p$ without violating any of the red-black properties. The new sibling of x , which is one of w 's children prior to the rotation, is now black, and thus we have converted case 1 into case 2, 3, or 4.

Cases 2, 3, and 4 occur when node w is black; they are distinguished by the colors of w 's children.

²As in RB-INSERT-FIXUP, the cases in RB-DELETE-FIXUP are not mutually exclusive.

Case 2: x 's sibling w is black, and both of w 's children are black

In case 2 (lines 10–11 of RB-DELETE-FIXUP and Figure 13.7(b)), both of w 's children are black. Since w is also black, we take one black off both x and w , leaving x with only one black and leaving w red. To compensate for removing one black from x and w , we would like to add an extra black to $x.p$, which was originally either red or black. We do so by repeating the **while** loop with $x.p$ as the new node x . Observe that if we enter case 2 through case 1, the new node x is red-and-black, since the original $x.p$ was red. Hence, the value c of the *color* attribute of the new node x is RED, and the loop terminates when it tests the loop condition. We then color the new node x (singly) black in line 23.

Case 3: x 's sibling w is black, w 's left child is red, and w 's right child is black

Case 3 (lines 13–16 and Figure 13.7(c)) occurs when w is black, its left child is red, and its right child is black. We can switch the colors of w and its left child $w.left$ and then perform a right rotation on w without violating any of the red-black properties. The new sibling w of x is now a black node with a red right child, and thus we have transformed case 3 into case 4.

Case 4: x 's sibling w is black, and w 's right child is red

Case 4 (lines 17–21 and Figure 13.7(d)) occurs when node x 's sibling w is black and w 's right child is red. By making some color changes and performing a left rotation on $x.p$, we can remove the extra black on x , making it singly black, without violating any of the red-black properties. Setting x to be the root causes the **while** loop to terminate when it tests the loop condition.

Analysis

What is the running time of RB-DELETE? Since the height of a red-black tree of n nodes is $O(\lg n)$, the total cost of the procedure without the call to RB-DELETE-FIXUP takes $O(\lg n)$ time. Within RB-DELETE-FIXUP, each of cases 1, 3, and 4 lead to termination after performing a constant number of color changes and at most three rotations. Case 2 is the only case in which the **while** loop can be repeated, and then the pointer x moves up the tree at most $O(\lg n)$ times, performing no rotations. Thus, the procedure RB-DELETE-FIXUP takes $O(\lg n)$ time and performs at most three rotations, and the overall time for RB-DELETE is therefore also $O(\lg n)$.

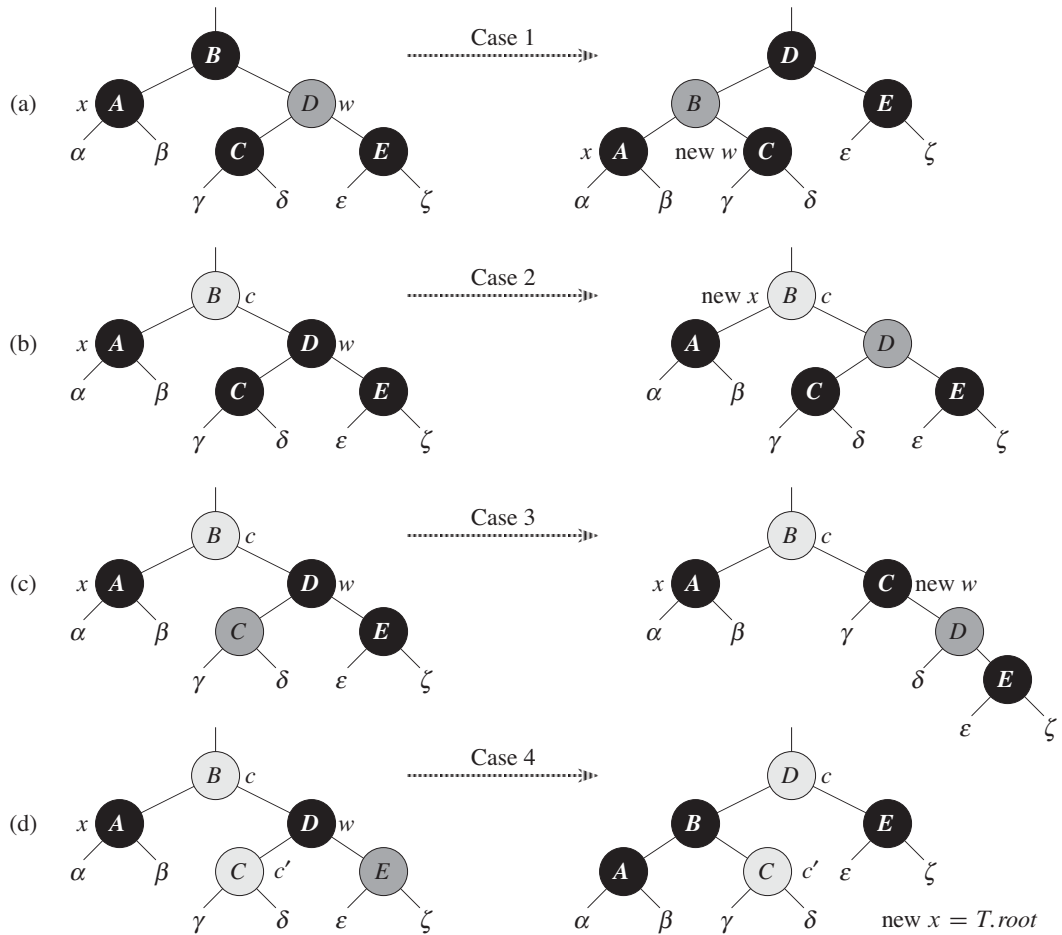


Figure 13.7 The cases in the **while** loop of the procedure **RB-DELETE-FIXUP**. Darkened nodes have *color* attributes **BLACK**, heavily shaded nodes have *color* attributes **RED**, and lightly shaded nodes have *color* attributes represented by c and c' , which may be either **RED** or **BLACK**. The letters $\alpha, \beta, \dots, \zeta$ represent arbitrary subtrees. Each case transforms the configuration on the left into the configuration on the right by changing some colors and/or performing a rotation. Any node pointed to by x has an extra black and is either doubly black or red-and-black. Only case 2 causes the loop to repeat. **(a)** Case 1 is transformed to case 2, 3, or 4 by exchanging the colors of nodes **B** and **D** and performing a left rotation. **(b)** In case 2, the extra black represented by the pointer x moves up the tree by coloring node **D** red and setting x to point to node **B**. If we enter case 2 through case 1, the **while** loop terminates because the new node x is red-and-black, and therefore the value c of its *color* attribute is **RED**. **(c)** Case 3 is transformed to case 4 by exchanging the colors of nodes **C** and **D** and performing a right rotation. **(d)** Case 4 removes the extra black represented by x by changing some colors and performing a left rotation (without violating the red-black properties), and then the loop terminates.

Exercises

13.4-1

Argue that after executing RB-DELETE-FIXUP, the root of the tree must be black.

13.4-2

Argue that if in RB-DELETE both x and $x.p$ are red, then property 4 is restored by the call to RB-DELETE-FIXUP(T, x).

13.4-3

In Exercise 13.3-2, you found the red-black tree that results from successively inserting the keys 41, 38, 31, 12, 19, 8 into an initially empty tree. Now show the red-black trees that result from the successive deletion of the keys in the order 8, 12, 19, 31, 38, 41.

13.4-4

In which lines of the code for RB-DELETE-FIXUP might we examine or modify the sentinel $T.nil$?

13.4-5

In each of the cases of Figure 13.7, give the count of black nodes from the root of the subtree shown to each of the subtrees $\alpha, \beta, \dots, \zeta$, and verify that each count remains the same after the transformation. When a node has a *color* attribute c or c' , use the notation $\text{count}(c)$ or $\text{count}(c')$ symbolically in your count.

13.4-6

Professors Skelton and Baron are concerned that at the start of case 1 of RB-DELETE-FIXUP, the node $x.p$ might not be black. If the professors are correct, then lines 5–6 are wrong. Show that $x.p$ must be black at the start of case 1, so that the professors have nothing to worry about.

13.4-7

Suppose that a node x is inserted into a red-black tree with RB-INSERT and then is immediately deleted with RB-DELETE. Is the resulting red-black tree the same as the initial red-black tree? Justify your answer.

Problems

13-1 Persistent dynamic sets

During the course of an algorithm, we sometimes find that we need to maintain past versions of a dynamic set as it is updated. We call such a set *persistent*. One way to implement a persistent set is to copy the entire set whenever it is modified, but this approach can slow down a program and also consume much space. Sometimes, we can do much better.

Consider a persistent set S with the operations INSERT, DELETE, and SEARCH, which we implement using binary search trees as shown in Figure 13.8(a). We maintain a separate root for every version of the set. In order to insert the key 5 into the set, we create a new node with key 5. This node becomes the left child of a new node with key 7, since we cannot modify the existing node with key 7. Similarly, the new node with key 7 becomes the left child of a new node with key 8 whose right child is the existing node with key 10. The new node with key 8 becomes, in turn, the right child of a new root r' with key 4 whose left child is the existing node with key 3. We thus copy only part of the tree and share some of the nodes with the original tree, as shown in Figure 13.8(b).

Assume that each tree node has the attributes *key*, *left*, and *right* but no parent. (See also Exercise 13.3-6.)

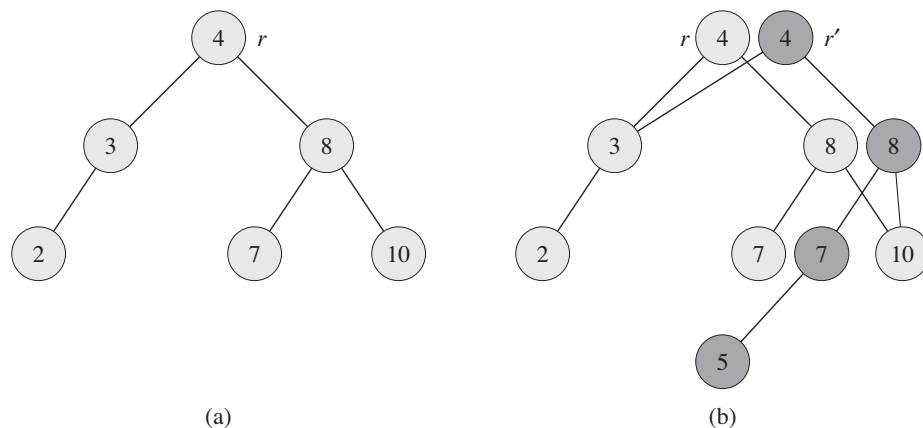


Figure 13.8 (a) A binary search tree with keys 2, 3, 4, 7, 8, 10. (b) The persistent binary search tree that results from the insertion of key 5. The most recent version of the set consists of the nodes reachable from the root r' , and the previous version consists of the nodes reachable from r . Heavily shaded nodes are added when key 5 is inserted.

- a. For a general persistent binary search tree, identify the nodes that we need to change to insert a key k or delete a node y .
- b. Write a procedure PERSISTENT-TREE-INSERT that, given a persistent tree T and a key k to insert, returns a new persistent tree T' that is the result of inserting k into T .
- c. If the height of the persistent binary search tree T is h , what are the time and space requirements of your implementation of PERSISTENT-TREE-INSERT? (The space requirement is proportional to the number of new nodes allocated.)
- d. Suppose that we had included the parent attribute in each node. In this case, PERSISTENT-TREE-INSERT would need to perform additional copying. Prove that PERSISTENT-TREE-INSERT would then require $\Omega(n)$ time and space, where n is the number of nodes in the tree.
- e. Show how to use red-black trees to guarantee that the worst-case running time and space are $O(\lg n)$ per insertion or deletion.

13-2 Join operation on red-black trees

The *join* operation takes two dynamic sets S_1 and S_2 and an element x such that for any $x_1 \in S_1$ and $x_2 \in S_2$, we have $x_1.key \leq x.key \leq x_2.key$. It returns a set $S = S_1 \cup \{x\} \cup S_2$. In this problem, we investigate how to implement the join operation on red-black trees.

- a. Given a red-black tree T , let us store its black-height as the new attribute $T.bh$. Argue that RB-INSERT and RB-DELETE can maintain the bh attribute without requiring extra storage in the nodes of the tree and without increasing the asymptotic running times. Show that while descending through T , we can determine the black-height of each node we visit in $O(1)$ time per node visited.

We wish to implement the operation RB-JOIN(T_1, x, T_2), which destroys T_1 and T_2 and returns a red-black tree $T = T_1 \cup \{x\} \cup T_2$. Let n be the total number of nodes in T_1 and T_2 .

- b. Assume that $T_1.bh \geq T_2.bh$. Describe an $O(\lg n)$ -time algorithm that finds a black node y in T_1 with the largest key from among those nodes whose black-height is $T_2.bh$.
- c. Let T_y be the subtree rooted at y . Describe how $T_y \cup \{x\} \cup T_2$ can replace T_y in $O(1)$ time without destroying the binary-search-tree property.
- d. What color should we make x so that red-black properties 1, 3, and 5 are maintained? Describe how to enforce properties 2 and 4 in $O(\lg n)$ time.

- e. Argue that no generality is lost by making the assumption in part (b). Describe the symmetric situation that arises when $T_1.bh \leq T_2.bh$.
- f. Argue that the running time of RB-JOIN is $O(\lg n)$.

13-3 AVL trees

An **AVL tree** is a binary search tree that is **height balanced**: for each node x , the heights of the left and right subtrees of x differ by at most 1. To implement an AVL tree, we maintain an extra attribute in each node: $x.h$ is the height of node x . As for any other binary search tree T , we assume that $T.root$ points to the root node.

- a. Prove that an AVL tree with n nodes has height $O(\lg n)$. (*Hint*: Prove that an AVL tree of height h has at least F_h nodes, where F_h is the h th Fibonacci number.)
- b. To insert into an AVL tree, we first place a node into the appropriate place in binary search tree order. Afterward, the tree might no longer be height balanced. Specifically, the heights of the left and right children of some node might differ by 2. Describe a procedure **BALANCE**(x), which takes a subtree rooted at x whose left and right children are height balanced and have heights that differ by at most 2, i.e., $|x.right.h - x.left.h| \leq 2$, and alters the subtree rooted at x to be height balanced. (*Hint*: Use rotations.)
- c. Using part (b), describe a recursive procedure **AVL-INSERT**(x, z) that takes a node x within an AVL tree and a newly created node z (whose key has already been filled in), and adds z to the subtree rooted at x , maintaining the property that x is the root of an AVL tree. As in **TREE-INSERT** from Section 12.3, assume that $z.key$ has already been filled in and that $z.left = \text{NIL}$ and $z.right = \text{NIL}$; also assume that $z.h = 0$. Thus, to insert the node z into the AVL tree T , we call **AVL-INSERT**($T.root, z$).
- d. Show that **AVL-INSERT**, run on an n -node AVL tree, takes $O(\lg n)$ time and performs $O(1)$ rotations.

13-4 Treaps

If we insert a set of n items into a binary search tree, the resulting tree may be horribly unbalanced, leading to long search times. As we saw in Section 12.4, however, randomly built binary search trees tend to be balanced. Therefore, one strategy that, on average, builds a balanced tree for a fixed set of items would be to randomly permute the items and then insert them in that order into the tree.

What if we do not have all the items at once? If we receive the items one at a time, can we still randomly build a binary search tree out of them?

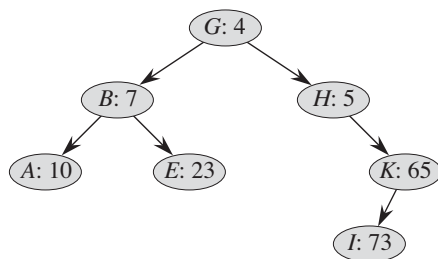


Figure 13.9 A treap. Each node x is labeled with $x.key: x.priority$. For example, the root has key G and priority 4.

We will examine a data structure that answers this question in the affirmative. A *treap* is a binary search tree with a modified way of ordering the nodes. Figure 13.9 shows an example. As usual, each node x in the tree has a key value $x.key$. In addition, we assign $x.priority$, which is a random number chosen independently for each node. We assume that all priorities are distinct and also that all keys are distinct. The nodes of the treap are ordered so that the keys obey the binary-search-tree property and the priorities obey the min-heap order property:

- If v is a left child of u , then $v.key < u.key$.
- If v is a right child of u , then $v.key > u.key$.
- If v is a child of u , then $v.priority > u.priority$.

(This combination of properties is why the tree is called a “treap”: it has features of both a binary search tree and a heap.)

It helps to think of treaps in the following way. Suppose that we insert nodes x_1, x_2, \dots, x_n , with associated keys, into a treap. Then the resulting treap is the tree that would have been formed if the nodes had been inserted into a normal binary search tree in the order given by their (randomly chosen) priorities, i.e., $x_i.priority < x_j.priority$ means that we had inserted x_i before x_j .

- a. Show that given a set of nodes x_1, x_2, \dots, x_n , with associated keys and priorities, all distinct, the treap associated with these nodes is unique.
- b. Show that the expected height of a treap is $\Theta(\lg n)$, and hence the expected time to search for a value in the treap is $\Theta(\lg n)$.

Let us see how to insert a new node into an existing treap. The first thing we do is assign to the new node a random priority. Then we call the insertion algorithm, which we call TREAP-INSERT, whose operation is illustrated in Figure 13.10.

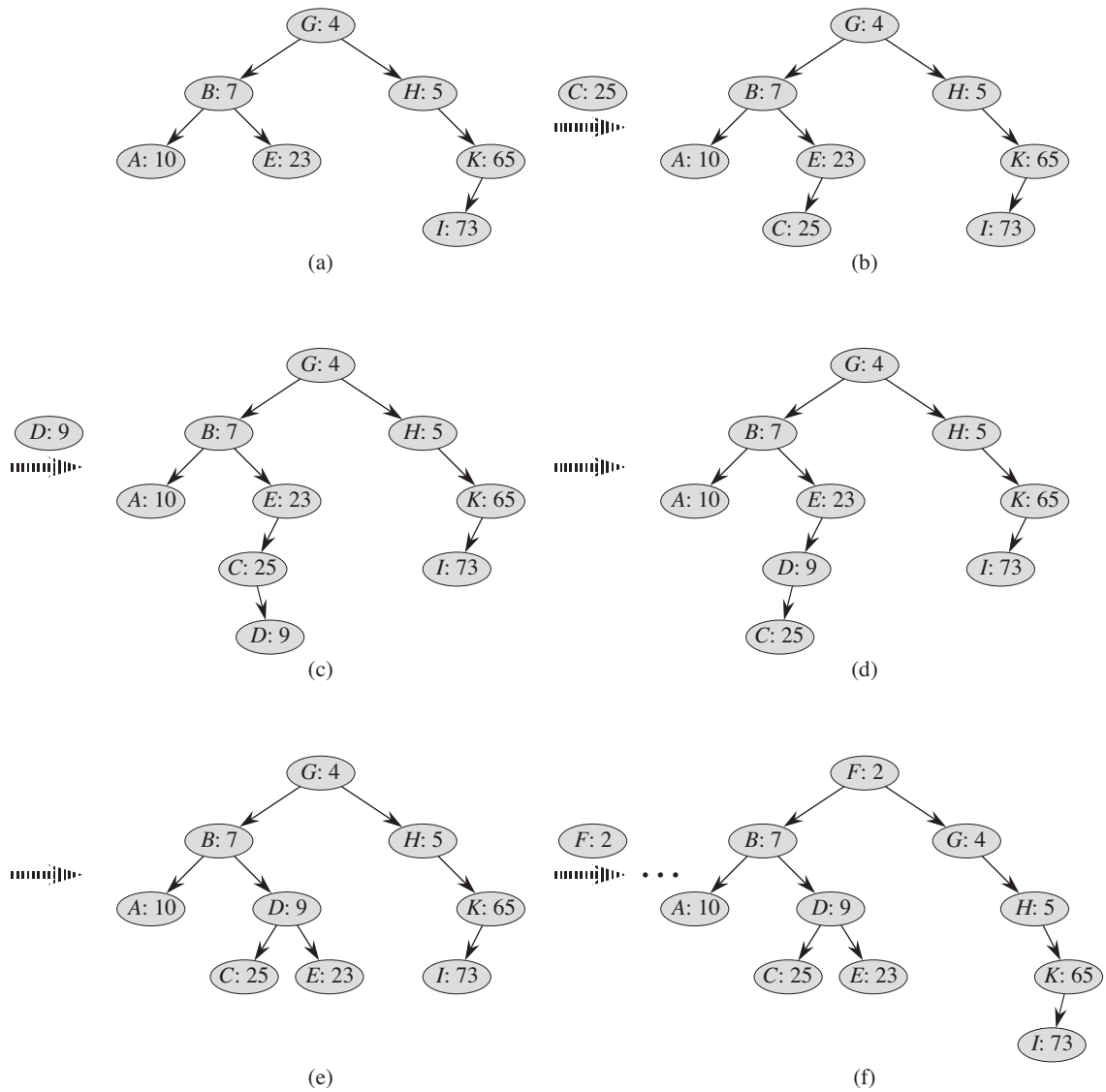


Figure 13.10 The operation of TREAP-INSERT. (a) The original treap, prior to insertion. (b) The treap after inserting a node with key *C* and priority 25. (c)–(d) Intermediate stages when inserting a node with key *D* and priority 9. (e) The treap after the insertion of parts (c) and (d) is done. (f) The treap after inserting a node with key *F* and priority 2.

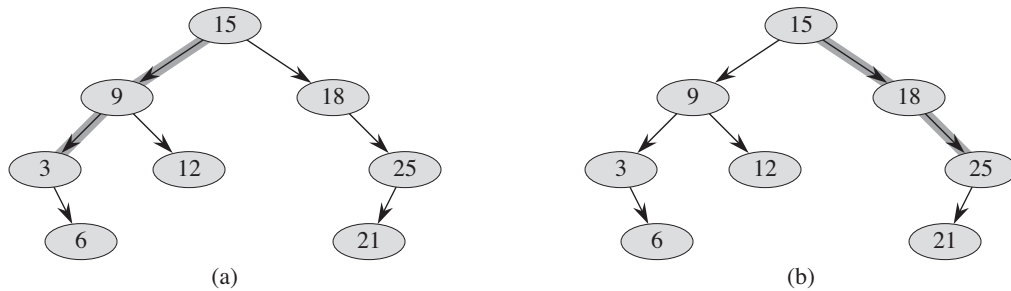


Figure 13.11 Spines of a binary search tree. The left spine is shaded in (a), and the right spine is shaded in (b).

- c. Explain how TREAP-INSERT works. Explain the idea in English and give pseudocode. (*Hint:* Execute the usual binary-search-tree insertion procedure and then perform rotations to restore the min-heap order property.)
- d. Show that the expected running time of TREAP-INSERT is $\Theta(\lg n)$.

TREAP-INSERT performs a search and then a sequence of rotations. Although these two operations have the same expected running time, they have different costs in practice. A search reads information from the treap without modifying it. In contrast, a rotation changes parent and child pointers within the treap. On most computers, read operations are much faster than write operations. Thus we would like TREAP-INSERT to perform few rotations. We will show that the expected number of rotations performed is bounded by a constant.

In order to do so, we will need some definitions, which Figure 13.11 depicts. The *left spine* of a binary search tree T is the simple path from the root to the node with the smallest key. In other words, the left spine is the simple path from the root that consists of only left edges. Symmetrically, the *right spine* of T is the simple path from the root consisting of only right edges. The *length* of a spine is the number of nodes it contains.

- e. Consider the treap T immediately after TREAP-INSERT has inserted node x . Let C be the length of the right spine of the left subtree of x . Let D be the length of the left spine of the right subtree of x . Prove that the total number of rotations that were performed during the insertion of x is equal to $C + D$.

We will now calculate the expected values of C and D . Without loss of generality, we assume that the keys are $1, 2, \dots, n$, since we are comparing them only to one another.

For nodes x and y in treap T , where $y \neq x$, let $k = x.key$ and $i = y.key$. We define indicator random variables

$$X_{ik} = I\{y \text{ is in the right spine of the left subtree of } x\}.$$

f. Show that $X_{ik} = 1$ if and only if $y.priority > x.priority$, $y.key < x.key$, and, for every z such that $y.key < z.key < x.key$, we have $y.priority < z.priority$.

g. Show that

$$\begin{aligned} \Pr\{X_{ik} = 1\} &= \frac{(k-i-1)!}{(k-i+1)!} \\ &= \frac{1}{(k-i+1)(k-i)}. \end{aligned}$$

h. Show that

$$\begin{aligned} E[C] &= \sum_{j=1}^{k-1} \frac{1}{j(j+1)} \\ &= 1 - \frac{1}{k}. \end{aligned}$$

i. Use a symmetry argument to show that

$$E[D] = 1 - \frac{1}{n-k+1}.$$

j. Conclude that the expected number of rotations performed when inserting a node into a treap is less than 2.

Chapter notes

The idea of balancing a search tree is due to Adel'son-Vel'skiĭ and Landis [2], who introduced a class of balanced search trees called “AVL trees” in 1962, described in Problem 13-3. Another class of search trees, called “2-3 trees,” was introduced by J. E. Hopcroft (unpublished) in 1970. A 2-3 tree maintains balance by manipulating the degrees of nodes in the tree. Chapter 18 covers a generalization of 2-3 trees introduced by Bayer and McCreight [35], called “B-trees.”

Red-black trees were invented by Bayer [34] under the name “symmetric binary B-trees.” Guibas and Sedgwick [155] studied their properties at length and introduced the red/black color convention. Andersson [15] gives a simpler-to-code

variant of red-black trees. Weiss [351] calls this variant AA-trees. An AA-tree is similar to a red-black tree except that left children may never be red.

Treaps, the subject of Problem 13-4, were proposed by Seidel and Aragon [309]. They are the default implementation of a dictionary in LEDA [253], which is a well-implemented collection of data structures and algorithms.

There are many other variations on balanced binary trees, including weight-balanced trees [264], k -neighbor trees [245], and scapegoat trees [127]. Perhaps the most intriguing are the “splay trees” introduced by Sleator and Tarjan [320], which are “self-adjusting.” (See Tarjan [330] for a good description of splay trees.) Splay trees maintain balance without any explicit balance condition such as color. Instead, “splay operations” (which involve rotations) are performed within the tree every time an access is made. The amortized cost (see Chapter 17) of each operation on an n -node tree is $O(\lg n)$.

Skip lists [286] provide an alternative to balanced binary trees. A skip list is a linked list that is augmented with a number of additional pointers. Each dictionary operation runs in expected time $O(\lg n)$ on a skip list of n items.

14 Augmenting Data Structures

Some engineering situations 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. Only in rare situations will you need to create an entirely new type of data structure, though. More often, it will suffice to augment a textbook data structure by storing additional information in it. You can then program new operations for the data structure to support the desired 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 that we construct by augmenting red-black trees. Section 14.1 describes a data structure that supports general order-statistic operations on a dynamic set. We can then quickly find the i th smallest number in a set or the rank of a given element in the total ordering of the set. Section 14.2 abstracts the process of augmenting a data structure and provides a theorem that can simplify the process of augmenting red-black trees. Section 14.3 uses this theorem to help design a data structure for maintaining a dynamic set of intervals, such as time intervals. Given a query interval, we can then quickly find an interval in the set that overlaps it.

14.1 Dynamic order statistics

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

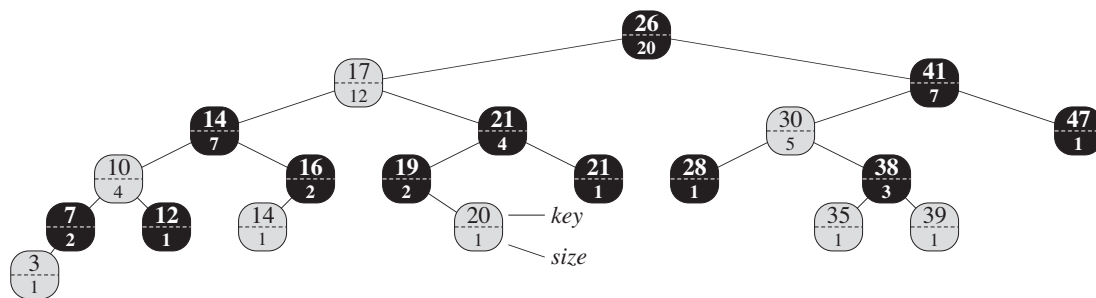


Figure 14.1 An order-statistic tree, which is an augmented red-black tree. Shaded nodes are red, and darkened nodes are black. In addition to its usual attributes, each node x has an attribute $x.size$, which is the number of nodes, other than the sentinel, in the subtree rooted at x .

Figure 14.1 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. Besides the usual red-black tree attributes $x.key$, $x.color$, $x.p$, $x.left$, and $x.right$ in a node x , we have another attribute, $x.size$. This attribute contains the number of (internal) nodes in the subtree rooted at x (including x itself), 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

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

We do not require keys to be distinct in an order-statistic tree. (For example, the tree in Figure 14.1 has two keys with value 14 and two keys with value 21.) In the presence of equal keys, 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 14.1, 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 an element with a given rank

Before we show how to maintain this size information during insertion and deletion, let us examine the implementation of two order-statistic queries that use this additional information. We begin with an operation that retrieves an element with a given rank. The procedure $OS-SELECT(x, i)$ returns a pointer to the node containing the i th smallest key in the subtree rooted at x . To find the node with the i th smallest key in an order-statistic tree T , we call $OS-SELECT(T.root, i)$.

```

OS-SELECT( $x, i$ )
1   $r = x.\text{left.size} + 1$ 
2  if  $i == r$ 
3      return  $x$ 
4  elseif  $i < r$ 
5      return OS-SELECT( $x.\text{left}, i$ )
6  else return OS-SELECT( $x.\text{right}, i - r$ )

```

In line 1 of OS-SELECT, we compute r , the rank of node x within the subtree rooted at x . The value of $x.\text{left.size}$ is the number of nodes that come before x in an inorder tree walk of the subtree rooted at x . Thus, $x.\text{left.size} + 1$ is the rank of x within the subtree rooted at x . If $i = r$, then node x is the i th smallest element, and so we return x in line 3. If $i < r$, then the i th smallest element resides in x 's left subtree, and so we recurse on $x.\text{left}$ in line 5. If $i > r$, then the i th 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 i th smallest element in the subtree rooted at x is the $(i - r)$ th smallest element in the subtree rooted at $x.\text{right}$. Line 6 determines this element recursively.

To see how OS-SELECT operates, consider a search for the 17th smallest element in the order-statistic tree of Figure 14.1. We begin 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, we know that the node with rank 17 is the $17 - 13 = 4$ th smallest element in 26's right subtree. After 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. Thus, we know that the node with rank 4 is the 4th smallest element in 41's left subtree. After the recursive call, x is the node with key 30, and its rank within its subtree is 2. Thus, we recurse once again to find the $4 - 2 = 2$ nd smallest element in the subtree rooted at the node with key 38. We now find that 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 returns the position of x in the linear order determined by an inorder tree walk of T .

OS-RANK(T, x)

```

1   $r = x.left.size + 1$ 
2   $y = x$ 
3  while  $y \neq T.root$ 
4      if  $y == y.p.right$ 
5           $r = r + y.p.left.size + 1$ 
6       $y = y.p$ 
7  return  $r$ 

```

The procedure works as follows. We 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, we set $y = y.p$. 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, we consider the subtree rooted at $y.p$. We have already counted the number of nodes in the subtree rooted at node y that precede x in an inorder walk, and so we 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 we leave 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. Thus, in line 5, we add $y.p.left.size + 1$ to the current value of r .

Termination: The loop terminates when $y = T.root$, so that 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 we run OS-RANK on the order-statistic tree of Figure 14.1 to find the rank of the node with key 38, we get the following sequence of values of $y.key$ and r 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 unless we can efficiently maintain these attributes within the basic modifying operations on red-black trees, our work will have been for naught. We shall now show how to maintain subtree sizes for both insertion and deletion without affecting the asymptotic running time of either operation.

We noted in 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, we 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) in Section 13.2, we add the following lines:

```

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

```

Figure 14.2 illustrates how the attributes are updated. The change to RIGHT-ROTATE is symmetric.

Since at most two rotations are performed during insertion into a red-black tree, we spend only $O(1)$ additional time updating *size* attributes in the second phase. 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.

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 either removes one node y from the tree or moves upward it within the tree. To update the subtree sizes, we simply traverse a simple path from node y (starting from its original position within the tree) up to the root, decrementing the *size*

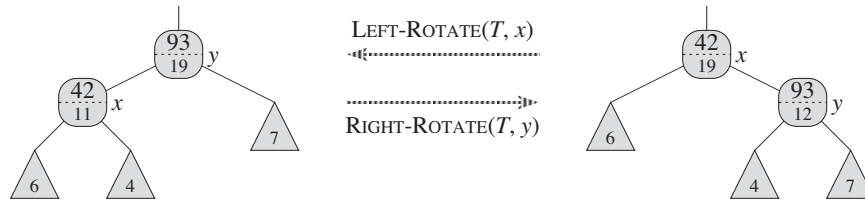


Figure 14.2 Updating subtree sizes during rotations. The link around which we rotate is incident on the two nodes whose *size* attributes need to be updated. The updates are local, requiring only the *size* information stored in x , y , and the roots of the subtrees shown as triangles.

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 $O(\lg n)$. We handle the $O(1)$ rotations in the second phase of deletion 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

14.1-1

Show how $\text{OS-SELECT}(T.\text{root}, 10)$ operates on the red-black tree T of Figure 14.1.

14.1-2

Show how $\text{OS-RANK}(T, x)$ operates on the red-black tree T of Figure 14.1 and the node x with $x.\text{key} = 35$.

14.1-3

Write a nonrecursive version of OS-SELECT .

14.1-4

Write a recursive procedure $\text{OS-KEY-RANK}(T, k)$ that takes as input 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.

14.1-5

Given an element x in an n -node order-statistic tree and a natural number i , how can we determine the i th successor of x in the linear order of the tree in $O(\lg n)$ time?

14.1-6

Observe that whenever we reference the *size* attribute of a node in either OS-SELECT or OS-RANK, we use it only to compute a rank. Accordingly, suppose we store in each node its rank in the subtree of which it is the root. Show how to maintain this information during insertion and deletion. (Remember that these two operations can cause rotations.)

14.1-7

Show how to use an order-statistic tree to count the number of inversions (see Problem 2-4) in an array of size n in time $O(n \lg n)$.

14.1-8 ★

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 correct answer is $\binom{n}{2}$.) Assume that no two chords share an endpoint.

14.2 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 shall use it again in the next section to design a data structure that supports operations on intervals. In this section, we examine the steps involved in such augmentation. We shall also prove a theorem that allows us to augment red-black trees easily in many cases.

We 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 we 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 should not blindly follow the steps 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 we will not be able to 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 way to organize the documentation of an augmented data structure.

We followed these steps in Section 14.1 to design our order-statistic trees. For step 1, we chose red-black trees as the underlying data structure. A clue to the suitability of red-black trees comes from their efficient support of other dynamic-set operations on a total order, such as `MINIMUM`, `MAXIMUM`, `SUCCESSOR`, and `PREDECESSOR`.

For step 2, we added the *size* attribute, in which each node x stores the size of the subtree rooted at x . Generally, the additional information makes operations more efficient. For example, we could have implemented `OS-SELECT` and `OS-RANK` using just the keys stored in the tree, but they would not have run in $O(\lg n)$ time. Sometimes, the additional information is pointer information rather than data, as in Exercise 14.2-1.

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

For step 4, we developed the operations `OS-SELECT` and `OS-RANK`. After all, the need for new operations is why we bother to augment a data structure in the first place. Occasionally, rather than developing new operations, we use the additional information to expedite existing ones, as in Exercise 14.2-1.

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 making step 3 very easy. The proof of the following theorem is similar to the argument from Section 14.1 that we can maintain the *size* attribute for order-statistic trees.

Theorem 14.1 (*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 on only the information in nodes x , $x.left$, and $x.right$, possibly including $x.left.f$ and $x.right.f$. Then, we can maintain the values of f in all nodes of T during insertion and deletion without asymptotically affecting the $O(\lg n)$ performance 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 re-

quire $x.p.f$ 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. Once we have updated $T.root.f$, no other node will depend 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.

Insertion of a node x into T consists of two phases. (See Section 13.3.) The first phase inserts x as a child of an existing node $x.p$. We can compute the value of $x.f$ in $O(1)$ time since, by supposition, it depends only on information in the other attributes of x itself and the information in x 's children, but x 's children are both the sentinel $T.nil$. Once we have 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, 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. (See Section 13.4.) In the first phase, changes to the tree occur when the deleted node is removed from the tree. If the deleted node had two children at the time, then its successor moves into the position of the deleted node. Propagating the updates to f caused by these changes costs at most $O(\lg n)$, since the changes modify the tree locally. 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 14.1. Exercise 14.2-3 gives an example.

Exercises

14.2-1

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.

14.2-2

Can we 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?

14.2-3 ★

Let \otimes be an associative binary operator, and let a be an attribute maintained in each node of a red-black tree. Suppose that we 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, \dots, 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.

14.2-4 ★

We wish to augment red-black trees with an operation $\text{RB-ENUMERATE}(x, a, b)$ that outputs all the keys k such that $a \leq k \leq b$ in a red-black tree rooted at x . Describe how to implement RB-ENUMERATE in $\Theta(m + \lg n)$ time, where m is the number of keys that are output and n is the number of internal nodes in the tree. (*Hint:* You do not need to add new attributes to the red-black tree.)

14.3 Interval trees

In this section, we shall augment red-black trees to support operations on dynamic sets of intervals. A **closed interval** is an ordered pair of real numbers $[t_1, t_2]$, with $t_1 \leq t_2$. The interval $[t_1, t_2]$ represents the set $\{t \in \mathbb{R} : t_1 \leq t \leq t_2\}$. **Open** and **half-open** intervals omit both or one of the endpoints from the set, respectively. In this section, we shall assume that intervals are closed; extending the results to open and half-open intervals is conceptually straightforward.

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

We can represent an interval $[t_1, t_2]$ as an object i , with attributes $i.\text{low} = t_1$ (the **low endpoint**) and $i.\text{high} = t_2$ (the **high endpoint**). We say that intervals i and i' **overlap** if $i \cap i' \neq \emptyset$, that is, if $i.\text{low} \leq i'.\text{high}$ and $i'.\text{low} \leq i.\text{high}$. As Figure 14.3 shows, any two intervals i and i' satisfy the **interval trichotomy**; that is, exactly one of the following three properties holds:

- i and i' overlap,
- i is to the left of i' (i.e., $i.\text{high} < i'.\text{low}$),
- i is to the right of i' (i.e., $i'.\text{high} < i.\text{low}$).

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

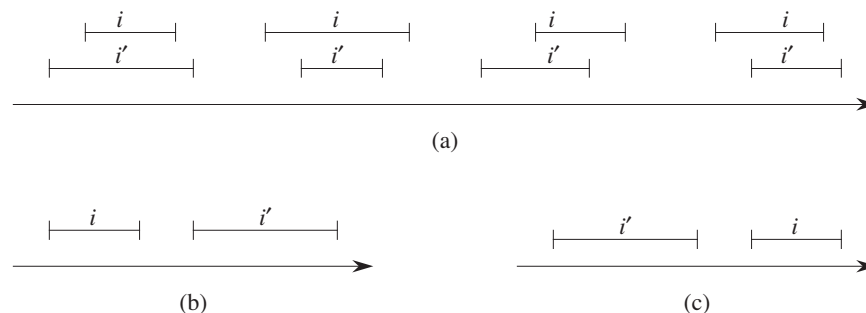


Figure 14.3 The interval trichotomy for two closed intervals i and i' . (a) If i and i' overlap, there are four situations; in each, $i.\text{low} \leq i'.\text{high}$ and $i'.\text{low} \leq i.\text{high}$. (b) The intervals do not overlap, and $i.\text{high} < i'.\text{low}$. (c) The intervals do not overlap, and $i'.\text{high} < i.\text{low}$.

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 $x.\text{int}$ overlaps interval i , or a pointer to the sentinel $T.\text{nil}$ if no such element is in the set.

Figure 14.4 shows how an interval tree represents a set of intervals. We shall track the four-step method from Section 14.2 as we review the design of an interval tree and the operations that run on it.

Step 1: Underlying data structure

We choose a red-black tree in which each node x contains an interval $x.\text{int}$ and the key of x is the low endpoint, $x.\text{int}.\text{low}$, of the interval. Thus, an inorder tree walk of the data structure lists the intervals in sorted order by low endpoint.

Step 2: Additional information

In addition to the intervals themselves, each node x contains a value $x.\text{max}$, 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. We can determine $x.\text{max}$ given interval $x.\text{int}$ and the *max* values of node x 's children:

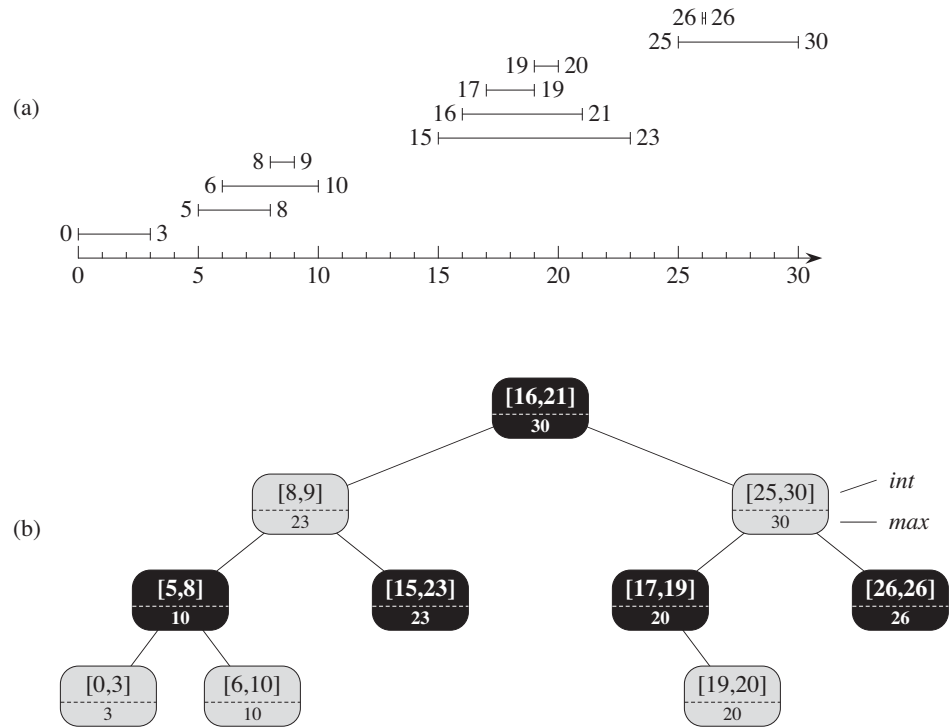


Figure 14.4 An interval tree. (a) A set of 10 intervals, shown sorted bottom to top by left endpoint. (b) The interval tree that represents them. Each node x contains an interval, shown above the dashed line, and the maximum value of any interval endpoint in the subtree rooted at x , shown below the dashed line. An inorder tree walk of the tree lists the nodes in sorted order by left endpoint.

$$x.max = \max(x.int.high, x.left.max, x.right.max) .$$

Thus, by Theorem 14.1, insertion and deletion run in $O(\lg n)$ time. In fact, we can update the *max* attributes after a rotation in $O(1)$ time, as Exercises 14.2-3 and 14.3-1 show.

Step 4: Developing new operations

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

INTERVAL-SEARCH(T, i)

```

1   $x = T.root$ 
2  while  $x \neq T.nil$  and  $i$  does not overlap  $x.int$ 
3      if  $x.left \neq T.nil$  and  $x.left.max \geq i.low$ 
4           $x = x.left$ 
5      else  $x = x.right$ 
6  return  $x$ 

```

The search for an interval that overlaps i starts with x at the root of the tree and proceeds downward. It terminates when either it finds an overlapping interval or x points to 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 14.4. Suppose we wish to find an interval that overlaps the interval $i = [22, 25]$. We 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.

As an example of an unsuccessful search, suppose we wish to find an interval that overlaps $i = [11, 14]$ in the interval tree of Figure 14.4. We once 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 $i.low = 11$, we 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 we go right. (Note that no interval in the left subtree overlaps i .) Interval $[15, 23]$ does not overlap i , and its left child is $T.nil$, so again we go right, the loop terminates, and we return 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 14.2

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 .

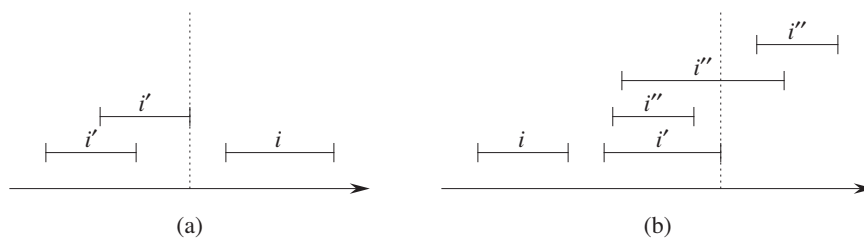


Figure 14.5 Intervals in the proof of Theorem 14.2. The value of $x.left.max$ is shown in each case as a dashed line. **(a)** The search goes right. No interval i' in x 's left subtree can overlap i . **(b)** The search goes left. The left subtree of x contains an interval that overlaps i (situation 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 $i'.low \leq i''.low$.

Proof The **while** loop of lines 2–5 terminates either when $x = T.nil$ 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 $x = T.nil$.

We use the following invariant for the **while** loop of lines 2–5:

If tree T contains an interval that overlaps i , then the subtree rooted at x contains such an interval.

We use this loop invariant as follows:

Initialization: Prior to the first iteration, line 1 sets x to be the root of T , so that the invariant holds.

Maintenance: Each iteration of the **while** loop executes either line 4 or line 5. We shall show that both cases maintain the loop invariant.

If line 5 is executed, then because of the branch condition in line 3, we have $x.left = T.nil$, or $x.left.max < i.low$. If $x.left = T.nil$, the subtree rooted at $x.left$ clearly contains no interval that overlaps i , and so setting x to $x.right$ maintains the invariant. Suppose, therefore, that $x.left \neq T.nil$ and $x.left.max < i.low$. As Figure 14.5(a) shows, for each interval i' in x 's left subtree, we have

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

By the interval trichotomy, therefore, i' and i do not overlap. Thus, the left subtree of x contains no intervals that overlap i , so that setting x to $x.right$ maintains the invariant.

If, on the other hand, line 4 is executed, then we will show that the contrapositive of the loop invariant holds. That is, if the subtree rooted at $x.left$ contains no interval overlapping i , then no interval anywhere in the tree overlaps i . Since line 4 is executed, then because of the branch condition in line 3, we have $x.left.max \geq i.low$. Moreover, by definition of the *max* attribute, x 's left subtree must contain some interval i' such that

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

(Figure 14.5(b) illustrates the situation.) Since i and i' do not overlap, and since it is not true that $i'.high < i.low$, it follows by the interval trichotomy that $i.high < i'.low$. Interval trees are keyed on the low endpoints of intervals, and thus the search-tree property implies that for any interval i'' in x 's right subtree,

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

By the interval trichotomy, i and i'' do not overlap. We conclude that whether or not any interval in x 's left subtree overlaps i , setting x to $x.left$ maintains the invariant.

Termination: If the loop terminates when $x = T.nil$, then the subtree rooted at x contains no interval overlapping i . The contrapositive of the loop invariant implies that T contains no interval that overlaps i . Hence it is correct to return $x = T.nil$. ■

Thus, the INTERVAL-SEARCH procedure works correctly.

Exercises

14.3-1

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

14.3-2

Rewrite the code for INTERVAL-SEARCH so that it works properly when all intervals are open.

14.3-3

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.

14.3-4

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.)

14.3-5

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.

14.3-6

Show how to maintain a dynamic set Q of numbers that supports the operation MIN-GAP, which gives the magnitude of the difference of the two closest numbers in Q . For example, if $Q = \{1, 5, 9, 15, 18, 22\}$, then MIN-GAP(Q) returns $18 - 15 = 3$, 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.

14.3-7 ★

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 we represent a rectangle by its minimum and maximum x - and y -coordinates. Give an $O(n \lg n)$ -time algorithm to decide whether or not 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
14-1 Point of maximum overlap

Suppose that we 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 will always be a point of maximum overlap that is an endpoint of one of the segments.

- 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.)

14-2 Josephus permutation

We define the **Josephus problem** as follows. Suppose that n people form a circle and that we are given a positive integer $m \leq n$. Beginning with a designated first person, we proceed around the circle, removing every m th person. After each person is removed, counting continues around the circle that remains. This process continues until we have removed all n people. The order in which the people are removed from the circle defines the **(n, m) -Josephus permutation** of the integers $1, 2, \dots, n$. 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 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 [282] 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.

IV Advanced Design and Analysis Techniques

Introduction

This part covers three important techniques used in designing and analyzing efficient algorithms: dynamic programming (Chapter 15), greedy algorithms (Chapter 16), and amortized analysis (Chapter 17). Earlier parts have presented other widely applicable techniques, such as divide-and-conquer, randomization, and how to solve recurrences. The techniques in this part are somewhat more sophisticated, but they help us to attack many computational problems. The themes introduced in this part will recur later in this book.

Dynamic programming typically applies to optimization problems in which we make a set of choices in order to arrive at an optimal solution. As we make each choice, subproblems of the same form often arise. Dynamic programming is effective when a given subproblem may arise from more than one partial set of choices; the key technique is to store the solution to each such subproblem in case it should reappear. Chapter 15 shows how this simple idea can sometimes transform exponential-time algorithms into polynomial-time algorithms.

Like dynamic-programming algorithms, greedy algorithms typically apply to optimization problems in which we make a set of choices in order to arrive at an optimal solution. The idea of a greedy algorithm is to make each choice in a locally optimal manner. A simple example is coin-changing: to minimize the number of U.S. coins needed to make change for a given amount, we can repeatedly select the largest-denomination coin that is not larger than the amount that remains. A greedy approach provides an optimal solution for many such problems much more quickly than would a dynamic-programming approach. We cannot always easily tell whether a greedy approach will be effective, however. Chapter 16 introduces

matroid theory, which provides a mathematical basis that can help us to show that a greedy algorithm yields an optimal solution.

We use amortized analysis to analyze certain algorithms that perform a sequence of similar operations. Instead of bounding the cost of the sequence of operations by bounding the actual cost of each operation separately, an amortized analysis provides a bound on the actual cost of the entire sequence. One advantage of this approach is that although some operations might be expensive, many others might be cheap. In other words, many of the operations might run in well under the worst-case time. Amortized analysis is not just an analysis tool, however; it is also a way of thinking about the design of algorithms, since the design of an algorithm and the analysis of its running time are often closely intertwined. Chapter 17 introduces three ways to perform an amortized analysis of an algorithm.

15 Dynamic Programming

Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems. (“Programming” in this context refers to a tabular method, not to writing computer code.) As we saw in Chapters 2 and 4, divide-and-conquer algorithms partition the problem into disjoint subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem. In contrast, dynamic programming applies when the subproblems overlap—that is, when subproblems share subsubproblems. In this context, a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common subsubproblems. A dynamic-programming algorithm solves each subsubproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time it solves each subsubproblem.

We typically apply dynamic programming to *optimization problems*. Such problems can have many possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value. We call such a solution *an* optimal solution to the problem, as opposed to *the* optimal solution, since there may be several solutions that achieve the optimal value.

When developing a dynamic-programming algorithm, we follow a sequence of four steps:

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.

Steps 1–3 form the basis of a dynamic-programming solution to a problem. If we need only the value of an optimal solution, and not the solution itself, then we can omit step 4. When we do perform step 4, we sometimes maintain additional information during step 3 so that we can easily construct an optimal solution.

The sections that follow use the dynamic-programming method to solve some optimization problems. Section 15.1 examines the problem of cutting a rod into

rods of smaller length in way that maximizes their total value. Section 15.2 asks how we can multiply a chain of matrices while performing the fewest total scalar multiplications. Given these examples of dynamic programming, Section 15.3 discusses two key characteristics that a problem must have for dynamic programming to be a viable solution technique. Section 15.4 then shows how to find the longest common subsequence of two sequences via dynamic programming. Finally, Section 15.5 uses dynamic programming to construct binary search trees that are optimal, given a known distribution of keys to be looked up.

15.1 Rod cutting

Our first example uses dynamic programming to solve a simple problem in deciding where to cut steel rods. Serling Enterprises buys long steel rods and cuts them into shorter rods, which it then sells. Each cut is free. The management of Serling Enterprises wants to know the best way to cut up the rods.

We assume that we know, for $i = 1, 2, \dots$, the price p_i in dollars that Serling Enterprises charges for a rod of length i inches. Rod lengths are always an integral number of inches. Figure 15.1 gives a sample price table.

The **rod-cutting problem** is the following. Given a rod of length n inches and a table of prices p_i for $i = 1, 2, \dots, n$, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces. Note that if the price p_n for a rod of length n is large enough, an optimal solution may require no cutting at all.

Consider the case when $n = 4$. Figure 15.2 shows all the ways to cut up a rod of 4 inches in length, including the way with no cuts at all. We see that cutting a 4-inch rod into two 2-inch pieces produces revenue $p_2 + p_2 = 5 + 5 = 10$, which is optimal.

We can cut up a rod of length n in 2^{n-1} different ways, since we have an independent option of cutting, or not cutting, at distance i inches from the left end,

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Figure 15.1 A sample price table for rods. Each rod of length i inches earns the company p_i dollars of revenue.

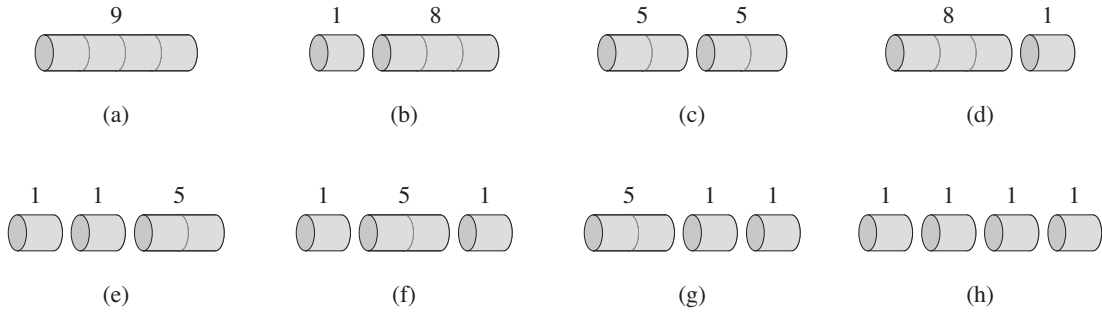


Figure 15.2 The 8 possible ways of cutting up a rod of length 4. Above each piece is the value of that piece, according to the sample price chart of Figure 15.1. The optimal strategy is part (c)—cutting the rod into two pieces of length 2—which has total value 10.

for $i = 1, 2, \dots, n - 1$.¹ We denote a decomposition into pieces using ordinary additive notation, so that $7 = 2 + 2 + 3$ indicates that a rod of length 7 is cut into three pieces—two of length 2 and one of length 3. If an optimal solution cuts the rod into k pieces, for some $1 \leq k \leq n$, then an optimal decomposition

$$n = i_1 + i_2 + \dots + i_k$$

of the rod into pieces of lengths i_1, i_2, \dots, i_k provides maximum corresponding revenue

$$r_n = p_{i_1} + p_{i_2} + \dots + p_{i_k}.$$

For our sample problem, we can determine the optimal revenue figures r_i , for $i = 1, 2, \dots, 10$, by inspection, with the corresponding optimal decompositions

¹If we required the pieces to be cut in order of nondecreasing size, there would be fewer ways to consider. For $n = 4$, we would consider only 5 such ways: parts (a), (b), (c), (e), and (h) in Figure 15.2. The number of ways is called the **partition function**; it is approximately equal to $e^{\pi\sqrt{2n/3}}/4n\sqrt{3}$. This quantity is less than 2^{n-1} , but still much greater than any polynomial in n . We shall not pursue this line of inquiry further, however.

$$\begin{aligned}
r_1 &= 1 && \text{from solution } 1 = 1 \quad (\text{no cuts}) , \\
r_2 &= 5 && \text{from solution } 2 = 2 \quad (\text{no cuts}) , \\
r_3 &= 8 && \text{from solution } 3 = 3 \quad (\text{no cuts}) , \\
r_4 &= 10 && \text{from solution } 4 = 2 + 2 , \\
r_5 &= 13 && \text{from solution } 5 = 2 + 3 , \\
r_6 &= 17 && \text{from solution } 6 = 6 \quad (\text{no cuts}) , \\
r_7 &= 18 && \text{from solution } 7 = 1 + 6 \text{ or } 7 = 2 + 2 + 3 , \\
r_8 &= 22 && \text{from solution } 8 = 2 + 6 , \\
r_9 &= 25 && \text{from solution } 9 = 3 + 6 , \\
r_{10} &= 30 && \text{from solution } 10 = 10 \quad (\text{no cuts}) .
\end{aligned}$$

More generally, we can frame the values r_n for $n \geq 1$ in terms of optimal revenues from shorter rods:

$$r_n = \max (p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \dots, r_{n-1} + r_1) . \quad (15.1)$$

The first argument, p_n , corresponds to making no cuts at all and selling the rod of length n as is. The other $n - 1$ arguments to max correspond to the maximum revenue obtained by making an initial cut of the rod into two pieces of size i and $n - i$, for each $i = 1, 2, \dots, n - 1$, and then optimally cutting up those pieces further, obtaining revenues r_i and r_{n-i} from those two pieces. Since we don't know ahead of time which value of i optimizes revenue, we have to consider all possible values for i and pick the one that maximizes revenue. We also have the option of picking no i at all if we can obtain more revenue by selling the rod uncut.

Note that to solve the original problem of size n , we solve smaller problems of the same type, but of smaller sizes. Once we make the first cut, we may consider the two pieces as independent instances of the rod-cutting problem. The overall optimal solution incorporates optimal solutions to the two related subproblems, maximizing revenue from each of those two pieces. We say that the rod-cutting problem exhibits **optimal substructure**: optimal solutions to a problem incorporate optimal solutions to related subproblems, which we may solve independently.

In a related, but slightly simpler, way to arrange a recursive structure for the rod-cutting problem, we view a decomposition as consisting of a first piece of length i cut off the left-hand end, and then a right-hand remainder of length $n - i$. Only the remainder, and not the first piece, may be further divided. We may view every decomposition of a length- n rod in this way: as a first piece followed by some decomposition of the remainder. When doing so, we can couch the solution with no cuts at all as saying that the first piece has size $i = n$ and revenue p_n and that the remainder has size 0 with corresponding revenue $r_0 = 0$. We thus obtain the following simpler version of equation (15.1):

$$r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i}) . \quad (15.2)$$

In this formulation, an optimal solution embodies the solution to only *one* related subproblem—the remainder—rather than two.

Recursive top-down implementation

The following procedure implements the computation implicit in equation (15.2) in a straightforward, top-down, recursive manner.

```

CUT-ROD( $p, n$ )
1  if  $n == 0$ 
2      return 0
3   $q = -\infty$ 
4  for  $i = 1$  to  $n$ 
5       $q = \max(q, p[i] + \text{CUT-ROD}(p, n - i))$ 
6  return  $q$ 

```

Procedure CUT-ROD takes as input an array $p[1..n]$ of prices and an integer n , and it returns the maximum revenue possible for a rod of length n . If $n = 0$, no revenue is possible, and so CUT-ROD returns 0 in line 2. Line 3 initializes the maximum revenue q to $-\infty$, so that the **for** loop in lines 4–5 correctly computes $q = \max_{1 \leq i \leq n} (p_i + \text{CUT-ROD}(p, n - i))$; line 6 then returns this value. A simple induction on n proves that this answer is equal to the desired answer r_n , using equation (15.2).

If you were to code up CUT-ROD in your favorite programming language and run it on your computer, you would find that once the input size becomes moderately large, your program would take a long time to run. For $n = 40$, you would find that your program takes at least several minutes, and most likely more than an hour. In fact, you would find that each time you increase n by 1, your program's running time would approximately double.

Why is CUT-ROD so inefficient? The problem is that CUT-ROD calls itself recursively over and over again with the same parameter values; it solves the same subproblems repeatedly. Figure 15.3 illustrates what happens for $n = 4$: CUT-ROD(p, n) calls CUT-ROD($p, n - i$) for $i = 1, 2, \dots, n$. Equivalently, CUT-ROD(p, n) calls CUT-ROD(p, j) for each $j = 0, 1, \dots, n - 1$. When this process unfolds recursively, the amount of work done, as a function of n , grows explosively.

To analyze the running time of CUT-ROD, let $T(n)$ denote the total number of calls made to CUT-ROD when called with its second parameter equal to n . This expression equals the number of nodes in a subtree whose root is labeled n in the recursion tree. The count includes the initial call at its root. Thus, $T(0) = 1$ and

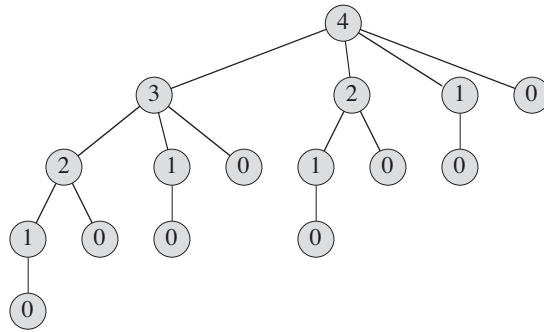


Figure 15.3 The recursion tree showing recursive calls resulting from a call $\text{CUT-ROD}(p, n)$ for $n = 4$. Each node label gives the size n of the corresponding subproblem, so that an edge from a parent with label s to a child with label t corresponds to cutting off an initial piece of size $s - t$ and leaving a remaining subproblem of size t . A path from the root to a leaf corresponds to one of the 2^{n-1} ways of cutting up a rod of length n . In general, this recursion tree has 2^n nodes and 2^{n-1} leaves.

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j) . \quad (15.3)$$

The initial 1 is for the call at the root, and the term $T(j)$ counts the number of calls (including recursive calls) due to the call $\text{CUT-ROD}(p, n - i)$, where $j = n - i$. As Exercise 15.1-1 asks you to show,

$$T(n) = 2^n , \quad (15.4)$$

and so the running time of CUT-ROD is exponential in n .

In retrospect, this exponential running time is not so surprising. CUT-ROD explicitly considers all the 2^{n-1} possible ways of cutting up a rod of length n . The tree of recursive calls has 2^{n-1} leaves, one for each possible way of cutting up the rod. The labels on the simple path from the root to a leaf give the sizes of each remaining right-hand piece before making each cut. That is, the labels give the corresponding cut points, measured from the right-hand end of the rod.

Using dynamic programming for optimal rod cutting

We now show how to convert CUT-ROD into an efficient algorithm, using dynamic programming.

The dynamic-programming method works as follows. Having observed that a naive recursive solution is inefficient because it solves the same subproblems repeatedly, we arrange for each subproblem to be solved only *once*, saving its solution. If we need to refer to this subproblem's solution again later, we can just look it

up, rather than recompute it. Dynamic programming thus uses additional memory to save computation time; it serves an example of a *time-memory trade-off*. The savings may be dramatic: an exponential-time solution may be transformed into a polynomial-time solution. A dynamic-programming approach runs in polynomial time when the number of *distinct* subproblems involved is polynomial in the input size and we can solve each such subproblem in polynomial time.

There are usually two equivalent ways to implement a dynamic-programming approach. We shall illustrate both of them with our rod-cutting example.

The first approach is *top-down with memoization*.² In this approach, we write the procedure recursively in a natural manner, but modified to save the result of each subproblem (usually in an array or hash table). The procedure now first checks to see whether it has previously solved this subproblem. If so, it returns the saved value, saving further computation at this level; if not, the procedure computes the value in the usual manner. We say that the recursive procedure has been *memoized*; it “remembers” what results it has computed previously.

The second approach is the *bottom-up method*. This approach typically depends on some natural notion of the “size” of a subproblem, such that solving any particular subproblem depends only on solving “smaller” subproblems. We sort the subproblems by size and solve them in size order, smallest first. When solving a particular subproblem, we have already solved all of the smaller subproblems its solution depends upon, and we have saved their solutions. We solve each subproblem only once, and when we first see it, we have already solved all of its prerequisite subproblems.

These two approaches yield algorithms with the same asymptotic running time, except in unusual circumstances where the top-down approach does not actually recurse to examine all possible subproblems. The bottom-up approach often has much better constant factors, since it has less overhead for procedure calls.

Here is the the pseudocode for the top-down CUT-ROD procedure, with memoization added:

```
MEMOIZED-CUT-ROD( $p, n$ )
1  let  $r[0 \dots n]$  be a new array
2  for  $i = 0$  to  $n$ 
3       $r[i] = -\infty$ 
4  return MEMOIZED-CUT-ROD-AUX( $p, n, r$ )
```

²This is not a misspelling. The word really is *memoization*, not *memorization*. *Memoization* comes from *memo*, since the technique consists of recording a value so that we can look it up later.

MEMOIZED-CUT-ROD-AUX(p, n, r)

```

1  if  $r[n] \geq 0$ 
2      return  $r[n]$ 
3  if  $n == 0$ 
4       $q = 0$ 
5  else  $q = -\infty$ 
6      for  $i = 1$  to  $n$ 
7           $q = \max(q, p[i] + \text{MEMOIZED-CUT-ROD-AUX}(p, n - i, r))$ 
8   $r[n] = q$ 
9  return  $q$ 

```

Here, the main procedure MEMOIZED-CUT-ROD initializes a new auxiliary array $r[0..n]$ with the value $-\infty$, a convenient choice with which to denote “unknown.” (Known revenue values are always nonnegative.) It then calls its helper routine, MEMOIZED-CUT-ROD-AUX.

The procedure MEMOIZED-CUT-ROD-AUX is just the memoized version of our previous procedure, CUT-ROD. It first checks in line 1 to see whether the desired value is already known and, if it is, then line 2 returns it. Otherwise, lines 3–7 compute the desired value q in the usual manner, line 8 saves it in $r[n]$, and line 9 returns it.

The bottom-up version is even simpler:

BOTTOM-UP-CUT-ROD(p, n)

```

1  let  $r[0..n]$  be a new array
2   $r[0] = 0$ 
3  for  $j = 1$  to  $n$ 
4       $q = -\infty$ 
5      for  $i = 1$  to  $j$ 
6           $q = \max(q, p[i] + r[j - i])$ 
7       $r[j] = q$ 
8  return  $r[n]$ 

```

For the bottom-up dynamic-programming approach, BOTTOM-UP-CUT-ROD uses the natural ordering of the subproblems: a problem of size i is “smaller” than a subproblem of size j if $i < j$. Thus, the procedure solves subproblems of sizes $j = 0, 1, \dots, n$, in that order.

Line 1 of procedure BOTTOM-UP-CUT-ROD creates a new array $r[0..n]$ in which to save the results of the subproblems, and line 2 initializes $r[0]$ to 0, since a rod of length 0 earns no revenue. Lines 3–6 solve each subproblem of size j , for $j = 1, 2, \dots, n$, in order of increasing size. The approach used to solve a problem of a particular size j is the same as that used by CUT-ROD, except that line 6 now

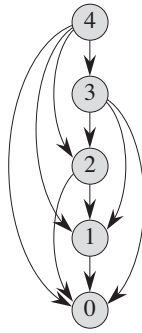


Figure 15.4 The subproblem graph for the rod-cutting problem with $n = 4$. The vertex labels give the sizes of the corresponding subproblems. A directed edge (x, y) indicates that we need a solution to subproblem y when solving subproblem x . This graph is a reduced version of the tree of Figure 15.3, in which all nodes with the same label are collapsed into a single vertex and all edges go from parent to child.

directly references array entry $r[j - i]$ instead of making a recursive call to solve the subproblem of size $j - i$. Line 7 saves in $r[j]$ the solution to the subproblem of size j . Finally, line 8 returns $r[n]$, which equals the optimal value r_n .

The bottom-up and top-down versions have the same asymptotic running time. The running time of procedure BOTTOM-UP-CUT-ROD is $\Theta(n^2)$, due to its doubly-nested loop structure. The number of iterations of its inner **for** loop, in lines 5–6, forms an arithmetic series. The running time of its top-down counterpart, MEMOIZED-CUT-ROD, is also $\Theta(n^2)$, although this running time may be a little harder to see. Because a recursive call to solve a previously solved subproblem returns immediately, MEMOIZED-CUT-ROD solves each subproblem just once. It solves subproblems for sizes $0, 1, \dots, n$. To solve a subproblem of size n , the **for** loop of lines 6–7 iterates n times. Thus, the total number of iterations of this **for** loop, over all recursive calls of MEMOIZED-CUT-ROD, forms an arithmetic series, giving a total of $\Theta(n^2)$ iterations, just like the inner **for** loop of BOTTOM-UP-CUT-ROD. (We actually are using a form of aggregate analysis here. We shall see aggregate analysis in detail in Section 17.1.)

Subproblem graphs

When we think about a dynamic-programming problem, we should understand the set of subproblems involved and how subproblems depend on one another.

The **subproblem graph** for the problem embodies exactly this information. Figure 15.4 shows the subproblem graph for the rod-cutting problem with $n = 4$. It is a directed graph, containing one vertex for each distinct subproblem. The sub-

problem graph has a directed edge from the vertex for subproblem x to the vertex for subproblem y if determining an optimal solution for subproblem x involves directly considering an optimal solution for subproblem y . For example, the subproblem graph contains an edge from x to y if a top-down recursive procedure for solving x directly calls itself to solve y . We can think of the subproblem graph as a “reduced” or “collapsed” version of the recursion tree for the top-down recursive method, in which we coalesce all nodes for the same subproblem into a single vertex and direct all edges from parent to child.

The bottom-up method for dynamic programming considers the vertices of the subproblem graph in such an order that we solve the subproblems y adjacent to a given subproblem x before we solve subproblem x . (Recall from Section B.4 that the adjacency relation is not necessarily symmetric.) Using the terminology from Chapter 22, in a bottom-up dynamic-programming algorithm, we consider the vertices of the subproblem graph in an order that is a “reverse topological sort,” or a “topological sort of the transpose” (see Section 22.4) of the subproblem graph. In other words, no subproblem is considered until all of the subproblems it depends upon have been solved. Similarly, using notions from the same chapter, we can view the top-down method (with memoization) for dynamic programming as a “depth-first search” of the subproblem graph (see Section 22.3).

The size of the subproblem graph $G = (V, E)$ can help us determine the running time of the dynamic programming algorithm. Since we solve each subproblem just once, the running time is the sum of the times needed to solve each subproblem. Typically, the time to compute the solution to a subproblem is proportional to the degree (number of outgoing edges) of the corresponding vertex in the subproblem graph, and the number of subproblems is equal to the number of vertices in the subproblem graph. In this common case, the running time of dynamic programming is linear in the number of vertices and edges.

Reconstructing a solution

Our dynamic-programming solutions to the rod-cutting problem return the value of an optimal solution, but they do not return an actual solution: a list of piece sizes. We can extend the dynamic-programming approach to record not only the optimal *value* computed for each subproblem, but also a *choice* that led to the optimal value. With this information, we can readily print an optimal solution.

Here is an extended version of BOTTOM-UP-CUT-ROD that computes, for each rod size j , not only the maximum revenue r_j , but also s_j , the optimal size of the first piece to cut off:

EXTENDED-BOTTOM-UP-CUT-ROD(p, n)

```

1  let  $r[0..n]$  and  $s[0..n]$  be new arrays
2   $r[0] = 0$ 
3  for  $j = 1$  to  $n$ 
4       $q = -\infty$ 
5      for  $i = 1$  to  $j$ 
6          if  $q < p[i] + r[j - i]$ 
7               $q = p[i] + r[j - i]$ 
8               $s[j] = i$ 
9       $r[j] = q$ 
10 return  $r$  and  $s$ 

```

This procedure is similar to BOTTOM-UP-CUT-ROD, except that it creates the array s in line 1, and it updates $s[j]$ in line 8 to hold the optimal size i of the first piece to cut off when solving a subproblem of size j .

The following procedure takes a price table p and a rod size n , and it calls EXTENDED-BOTTOM-UP-CUT-ROD to compute the array $s[1..n]$ of optimal first-piece sizes and then prints out the complete list of piece sizes in an optimal decomposition of a rod of length n :

PRINT-CUT-ROD-SOLUTION(p, n)

```

1   $(r, s) = \text{EXTENDED-BOTTOM-UP-CUT-ROD}(p, n)$ 
2  while  $n > 0$ 
3      print  $s[n]$ 
4       $n = n - s[n]$ 

```

In our rod-cutting example, the call EXTENDED-BOTTOM-UP-CUT-ROD($p, 10$) would return the following arrays:

i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

A call to PRINT-CUT-ROD-SOLUTION($p, 10$) would print just 10, but a call with $n = 7$ would print the cuts 1 and 6, corresponding to the first optimal decomposition for r_7 given earlier.

Exercises

15.1-1

Show that equation (15.4) follows from equation (15.3) and the initial condition $T(0) = 1$.

15.1-2

Show, by means of a counterexample, that the following “greedy” strategy does not always determine an optimal way to cut rods. Define the *density* of a rod of length i to be p_i/i , that is, its value per inch. The greedy strategy for a rod of length n cuts off a first piece of length i , where $1 \leq i \leq n$, having maximum density. It then continues by applying the greedy strategy to the remaining piece of length $n - i$.

15.1-3

Consider a modification of the rod-cutting problem in which, in addition to a price p_i for each rod, each cut incurs a fixed cost of c . The revenue associated with a solution is now the sum of the prices of the pieces minus the costs of making the cuts. Give a dynamic-programming algorithm to solve this modified problem.

15.1-4

Modify MEMOIZED-CUT-ROD to return not only the value but the actual solution, too.

15.1-5

The Fibonacci numbers are defined by recurrence (3.22). Give an $O(n)$ -time dynamic-programming algorithm to compute the n th Fibonacci number. Draw the subproblem graph. How many vertices and edges are in the graph?

15.2 Matrix-chain multiplication

Our next example of dynamic programming is an algorithm that solves the problem of matrix-chain multiplication. We are given a sequence (chain) $\langle A_1, A_2, \dots, A_n \rangle$ of n matrices to be multiplied, and we wish to compute the product

$$A_1 A_2 \cdots A_n. \quad (15.5)$$

We can evaluate the expression (15.5) using the standard algorithm for multiplying pairs of matrices as a subroutine once we have parenthesized it to resolve all ambiguities in how the matrices are multiplied together. Matrix multiplication is associative, and so all parenthesizations yield the same product. A product of matrices is *fully parenthesized* if it is either a single matrix or the product of two fully parenthesized matrix products, surrounded by parentheses. For example, if the chain of matrices is $\langle A_1, A_2, A_3, A_4 \rangle$, then we can fully parenthesize the product $A_1 A_2 A_3 A_4$ in five distinct ways:

$(A_1(A_2(A_3A_4)))$,
 $(A_1((A_2A_3)A_4))$,
 $((A_1A_2)(A_3A_4))$,
 $((A_1(A_2A_3))A_4)$,
 $((A_1A_2)A_3)A_4)$.

How we parenthesize a chain of matrices can have a dramatic impact on the cost of evaluating the product. Consider first the cost of multiplying two matrices. The standard algorithm is given by the following pseudocode, which generalizes the SQUARE-MATRIX-MULTIPLY procedure from Section 4.2. The attributes *rows* and *columns* are the numbers of rows and columns in a matrix.

MATRIX-MULTIPLY(A, B)

```

1  if  $A.columns \neq B.rows$ 
2      error “incompatible dimensions”
3  else let  $C$  be a new  $A.rows \times B.columns$  matrix
4      for  $i = 1$  to  $A.rows$ 
5          for  $j = 1$  to  $B.columns$ 
6               $c_{ij} = 0$ 
7              for  $k = 1$  to  $A.columns$ 
8                   $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$ 
9  return  $C$ 
  
```

We can multiply two matrices A and B only if they are **compatible**: the number of columns of A must equal the number of rows of B . If A is a $p \times q$ matrix and B is a $q \times r$ matrix, the resulting matrix C is a $p \times r$ matrix. The time to compute C is dominated by the number of scalar multiplications in line 8, which is pqr . In what follows, we shall express costs in terms of the number of scalar multiplications.

To illustrate the different costs incurred by different parenthesizations of a matrix product, consider the problem of a chain $\langle A_1, A_2, A_3 \rangle$ of three matrices. Suppose that the dimensions of the matrices are 10×100 , 100×5 , and 5×50 , respectively. If we multiply according to the parenthesization $((A_1A_2)A_3)$, we perform $10 \cdot 100 \cdot 5 = 5000$ scalar multiplications to compute the 10×5 matrix product A_1A_2 , plus another $10 \cdot 5 \cdot 50 = 2500$ scalar multiplications to multiply this matrix by A_3 , for a total of 7500 scalar multiplications. If instead we multiply according to the parenthesization $(A_1(A_2A_3))$, we perform $100 \cdot 5 \cdot 50 = 25,000$ scalar multiplications to compute the 100×50 matrix product A_2A_3 , plus another $10 \cdot 100 \cdot 50 = 50,000$ scalar multiplications to multiply A_1 by this matrix, for a total of 75,000 scalar multiplications. Thus, computing the product according to the first parenthesization is 10 times faster.

We state the **matrix-chain multiplication problem** as follows: given a chain $\langle A_1, A_2, \dots, A_n \rangle$ of n matrices, where for $i = 1, 2, \dots, n$, matrix A_i has dimension

$p_{i-1} \times p_i$, fully parenthesize the product $A_1 A_2 \cdots A_n$ in a way that minimizes the number of scalar multiplications.

Note that in the matrix-chain multiplication problem, we are not actually multiplying matrices. Our goal is only to determine an order for multiplying matrices that has the lowest cost. Typically, the time invested in determining this optimal order is more than paid for by the time saved later on when actually performing the matrix multiplications (such as performing only 7500 scalar multiplications instead of 75,000).

Counting the number of parenthesizations

Before solving the matrix-chain multiplication problem by dynamic programming, let us convince ourselves that exhaustively checking all possible parenthesizations does not yield an efficient algorithm. Denote the number of alternative parenthesizations of a sequence of n matrices by $P(n)$. When $n = 1$, we have just one matrix and therefore only one way to fully parenthesize the matrix product. When $n \geq 2$, a fully parenthesized matrix product is the product of two fully parenthesized matrix subproducts, and the split between the two subproducts may occur between the k th and $(k + 1)$ st matrices for any $k = 1, 2, \dots, n - 1$. Thus, we obtain the recurrence

$$P(n) = \begin{cases} 1 & \text{if } n = 1, \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2. \end{cases} \quad (15.6)$$

Problem 12-4 asked you to show that the solution to a similar recurrence is the sequence of **Catalan numbers**, which grows as $\Omega(4^n/n^{3/2})$. A simpler exercise (see Exercise 15.2-3) is to show that the solution to the recurrence (15.6) is $\Omega(2^n)$. The number of solutions is thus exponential in n , and the brute-force method of exhaustive search makes for a poor strategy when determining how to optimally parenthesize a matrix chain.

Applying dynamic programming

We shall use the dynamic-programming method to determine how to optimally parenthesize a matrix chain. In so doing, we shall follow the four-step sequence that we stated at the beginning of this chapter:

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution.

4. Construct an optimal solution from computed information.

We shall go through these steps in order, demonstrating clearly how we apply each step to the problem.

Step 1: The structure of an optimal parenthesization

For our first step in the dynamic-programming paradigm, we find the optimal substructure and then use it to construct an optimal solution to the problem from optimal solutions to subproblems. In the matrix-chain multiplication problem, we can perform this step as follows. For convenience, let us adopt the notation $A_{i..j}$, where $i \leq j$, for the matrix that results from evaluating the product $A_i A_{i+1} \cdots A_j$. Observe that if the problem is nontrivial, i.e., $i < j$, then to parenthesize the product $A_i A_{i+1} \cdots A_j$, we must split the product between A_k and A_{k+1} for some integer k in the range $i \leq k < j$. That is, for some value of k , we first compute the matrices $A_{i..k}$ and $A_{k+1..j}$ and then multiply them together to produce the final product $A_{i..j}$. The cost of parenthesizing this way is the cost of computing the matrix $A_{i..k}$, plus the cost of computing $A_{k+1..j}$, plus the cost of multiplying them together.

The optimal substructure of this problem is as follows. Suppose that to optimally parenthesize $A_i A_{i+1} \cdots A_j$, we split the product between A_k and A_{k+1} . Then the way we parenthesize the “prefix” subchain $A_i A_{i+1} \cdots A_k$ within this optimal parenthesization of $A_i A_{i+1} \cdots A_j$ must be an optimal parenthesization of $A_i A_{i+1} \cdots A_k$. Why? If there were a less costly way to parenthesize $A_i A_{i+1} \cdots A_k$, then we could substitute that parenthesization in the optimal parenthesization of $A_i A_{i+1} \cdots A_j$ to produce another way to parenthesize $A_i A_{i+1} \cdots A_j$ whose cost was lower than the optimum: a contradiction. A similar observation holds for how we parenthesize the subchain $A_{k+1} A_{k+2} \cdots A_j$ in the optimal parenthesization of $A_i A_{i+1} \cdots A_j$: it must be an optimal parenthesization of $A_{k+1} A_{k+2} \cdots A_j$.

Now we use our optimal substructure to show that we can construct an optimal solution to the problem from optimal solutions to subproblems. We have seen that any solution to a nontrivial instance of the matrix-chain multiplication problem requires us to split the product, and that any optimal solution contains within it optimal solutions to subproblem instances. Thus, we can build an optimal solution to an instance of the matrix-chain multiplication problem by splitting the problem into two subproblems (optimally parenthesizing $A_i A_{i+1} \cdots A_k$ and $A_{k+1} A_{k+2} \cdots A_j$), finding optimal solutions to subproblem instances, and then combining these optimal subproblem solutions. We must ensure that when we search for the correct place to split the product, we have considered all possible places, so that we are sure of having examined the optimal one.

Step 2: A recursive solution

Next, we define the cost of an optimal solution recursively in terms of the optimal solutions to subproblems. For the matrix-chain multiplication problem, we pick as our subproblems the problems of determining the minimum cost of parenthesizing $A_i A_{i+1} \cdots A_j$ for $1 \leq i \leq j \leq n$. Let $m[i, j]$ be the minimum number of scalar multiplications needed to compute the matrix $A_{i..j}$; for the full problem, the lowest-cost way to compute $A_{1..n}$ would thus be $m[1, n]$.

We can define $m[i, j]$ recursively as follows. If $i = j$, the problem is trivial; the chain consists of just one matrix $A_{i..i} = A_i$, so that no scalar multiplications are necessary to compute the product. Thus, $m[i, i] = 0$ for $i = 1, 2, \dots, n$. To compute $m[i, j]$ when $i < j$, we take advantage of the structure of an optimal solution from step 1. Let us assume that to optimally parenthesize, we split the product $A_i A_{i+1} \cdots A_j$ between A_k and A_{k+1} , where $i \leq k < j$. Then, $m[i, j]$ equals the minimum cost for computing the subproducts $A_{i..k}$ and $A_{k+1..j}$, plus the cost of multiplying these two matrices together. Recalling that each matrix A_i is $p_{i-1} \times p_i$, we see that computing the matrix product $A_{i..k} A_{k+1..j}$ takes $p_{i-1} p_k p_j$ scalar multiplications. Thus, we obtain

$$m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j .$$

This recursive equation assumes that we know the value of k , which we do not. There are only $j - i$ possible values for k , however, namely $k = i, i + 1, \dots, j - 1$. Since the optimal parenthesization must use one of these values for k , we need only check them all to find the best. Thus, our recursive definition for the minimum cost of parenthesizing the product $A_i A_{i+1} \cdots A_j$ becomes

$$m[i, j] = \begin{cases} 0 & \text{if } i = j , \\ \min_{i \leq k < j} \{m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j\} & \text{if } i < j . \end{cases} \quad (15.7)$$

The $m[i, j]$ values give the costs of optimal solutions to subproblems, but they do not provide all the information we need to construct an optimal solution. To help us do so, we define $s[i, j]$ to be a value of k at which we split the product $A_i A_{i+1} \cdots A_j$ in an optimal parenthesization. That is, $s[i, j]$ equals a value k such that $m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$.

Step 3: Computing the optimal costs

At this point, we could easily write a recursive algorithm based on recurrence (15.7) to compute the minimum cost $m[1, n]$ for multiplying $A_1 A_2 \cdots A_n$. As we saw for the rod-cutting problem, and as we shall see in Section 15.3, this recursive algorithm takes exponential time, which is no better than the brute-force method of checking each way of parenthesizing the product.

Observe that we have relatively few distinct subproblems: one subproblem for each choice of i and j satisfying $1 \leq i \leq j \leq n$, or $\binom{n}{2} + n = \Theta(n^2)$ in all. A recursive algorithm may encounter each subproblem many times in different branches of its recursion tree. This property of overlapping subproblems is the second hallmark of when dynamic programming applies (the first hallmark being optimal substructure).

Instead of computing the solution to recurrence (15.7) recursively, we compute the optimal cost by using a tabular, bottom-up approach. (We present the corresponding top-down approach using memoization in Section 15.3.)

We shall implement the tabular, bottom-up method in the procedure MATRIX-CHAIN-ORDER, which appears below. This procedure assumes that matrix A_i has dimensions $p_{i-1} \times p_i$ for $i = 1, 2, \dots, n$. Its input is a sequence $p = \langle p_0, p_1, \dots, p_n \rangle$, where $p.length = n + 1$. The procedure uses an auxiliary table $m[1..n, 1..n]$ for storing the $m[i, j]$ costs and another auxiliary table $s[1..n - 1, 2..n]$ that records which index of k achieved the optimal cost in computing $m[i, j]$. We shall use the table s to construct an optimal solution.

In order to implement the bottom-up approach, we must determine which entries of the table we refer to when computing $m[i, j]$. Equation (15.7) shows that the cost $m[i, j]$ of computing a matrix-chain product of $j - i + 1$ matrices depends only on the costs of computing matrix-chain products of fewer than $j - i + 1$ matrices. That is, for $k = i, i + 1, \dots, j - 1$, the matrix $A_{i..k}$ is a product of $k - i + 1 < j - i + 1$ matrices and the matrix $A_{k+1..j}$ is a product of $j - k < j - i + 1$ matrices. Thus, the algorithm should fill in the table m in a manner that corresponds to solving the parenthesization problem on matrix chains of increasing length. For the subproblem of optimally parenthesizing the chain $A_i A_{i+1} \cdots A_j$, we consider the subproblem size to be the length $j - i + 1$ of the chain.

MATRIX-CHAIN-ORDER(p)

```

1   $n = p.length - 1$ 
2  let  $m[1..n, 1..n]$  and  $s[1..n - 1, 2..n]$  be new tables
3  for  $i = 1$  to  $n$ 
4       $m[i, i] = 0$ 
5  for  $l = 2$  to  $n$            //  $l$  is the chain length
6      for  $i = 1$  to  $n - l + 1$ 
7           $j = i + l - 1$ 
8           $m[i, j] = \infty$ 
9          for  $k = i$  to  $j - 1$ 
10              $q = m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j$ 
11             if  $q < m[i, j]$ 
12                  $m[i, j] = q$ 
13                  $s[i, j] = k$ 
14  return  $m$  and  $s$ 
```

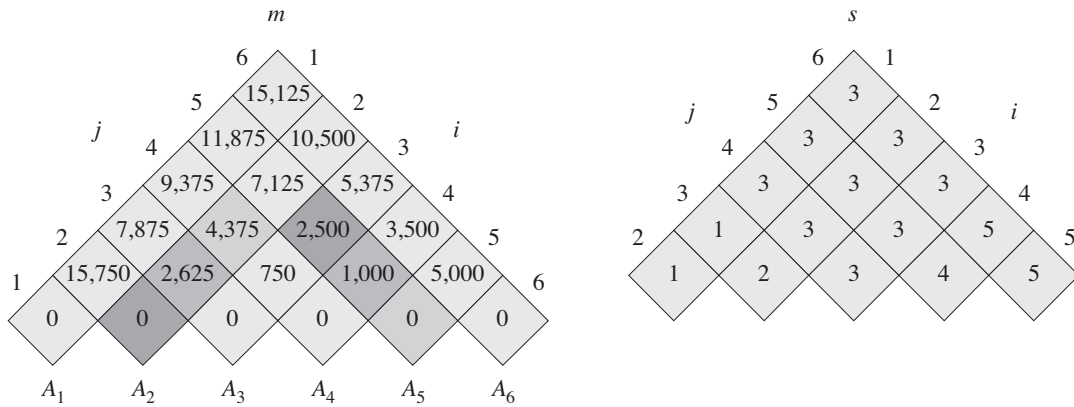


Figure 15.5 The m and s tables computed by MATRIX-CHAIN-ORDER for $n = 6$ and the following matrix dimensions:

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25

The tables are rotated so that the main diagonal runs horizontally. The m table uses only the main diagonal and upper triangle, and the s table uses only the upper triangle. The minimum number of scalar multiplications to multiply the 6 matrices is $m[1, 6] = 15,125$. Of the darker entries, the pairs that have the same shading are taken together in line 10 when computing

$$\begin{aligned}
 m[2, 5] &= \min \begin{cases} m[2, 2] + m[3, 5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13,000, \\ m[2, 3] + m[4, 5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125, \\ m[2, 4] + m[5, 5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11,375 \end{cases} \\
 &= 7125.
 \end{aligned}$$

The algorithm first computes $m[i, i] = 0$ for $i = 1, 2, \dots, n$ (the minimum costs for chains of length 1) in lines 3–4. It then uses recurrence (15.7) to compute $m[i, i + 1]$ for $i = 1, 2, \dots, n - 1$ (the minimum costs for chains of length $l = 2$) during the first execution of the **for** loop in lines 5–13. The second time through the loop, it computes $m[i, i + 2]$ for $i = 1, 2, \dots, n - 2$ (the minimum costs for chains of length $l = 3$), and so forth. At each step, the $m[i, j]$ cost computed in lines 10–13 depends only on table entries $m[i, k]$ and $m[k + 1, j]$ already computed.

Figure 15.5 illustrates this procedure on a chain of $n = 6$ matrices. Since we have defined $m[i, j]$ only for $i \leq j$, only the portion of the table m strictly above the main diagonal is used. The figure shows the table rotated to make the main diagonal run horizontally. The matrix chain is listed along the bottom. Using this layout, we can find the minimum cost $m[i, j]$ for multiplying a subchain $A_i A_{i+1} \cdots A_j$ of matrices at the intersection of lines running northeast from A_i and

northwest from A_j . Each horizontal row in the table contains the entries for matrix chains of the same length. MATRIX-CHAIN-ORDER computes the rows from bottom to top and from left to right within each row. It computes each entry $m[i, j]$ using the products $p_{i-1}p_kp_j$ for $k = i, i + 1, \dots, j - 1$ and all entries southwest and southeast from $m[i, j]$.

A simple inspection of the nested loop structure of MATRIX-CHAIN-ORDER yields a running time of $O(n^3)$ for the algorithm. The loops are nested three deep, and each loop index (l , i , and k) takes on at most $n - 1$ values. Exercise 15.2-5 asks you to show that the running time of this algorithm is in fact also $\Omega(n^3)$. The algorithm requires $\Theta(n^2)$ space to store the m and s tables. Thus, MATRIX-CHAIN-ORDER is much more efficient than the exponential-time method of enumerating all possible parenthesizations and checking each one.

Step 4: Constructing an optimal solution

Although MATRIX-CHAIN-ORDER determines the optimal number of scalar multiplications needed to compute a matrix-chain product, it does not directly show how to multiply the matrices. The table $s[1..n - 1, 2..n]$ gives us the information we need to do so. Each entry $s[i, j]$ records a value of k such that an optimal parenthesization of $A_iA_{i+1}\cdots A_j$ splits the product between A_k and A_{k+1} . Thus, we know that the final matrix multiplication in computing $A_{1..n}$ optimally is $A_{1..s[1,n]}A_{s[1,n]+1..n}$. We can determine the earlier matrix multiplications recursively, since $s[1, s[1, n]]$ determines the last matrix multiplication when computing $A_{1..s[1,n]}$ and $s[s[1, n] + 1, n]$ determines the last matrix multiplication when computing $A_{s[1,n]+1..n}$. The following recursive procedure prints an optimal parenthesization of $\langle A_i, A_{i+1}, \dots, A_j \rangle$, given the s table computed by MATRIX-CHAIN-ORDER and the indices i and j . The initial call PRINT-OPTIMAL-PARENS($s, 1, n$) prints an optimal parenthesization of $\langle A_1, A_2, \dots, A_n \rangle$.

```

PRINT-OPTIMAL-PARENS( $s, i, j$ )
1  if  $i == j$ 
2      print " $A$ " $i$ 
3  else print "("
4      PRINT-OPTIMAL-PARENS( $s, i, s[i, j]$ )
5      PRINT-OPTIMAL-PARENS( $s, s[i, j] + 1, j$ )
6      print ")"

```

In the example of Figure 15.5, the call PRINT-OPTIMAL-PARENS($s, 1, 6$) prints the parenthesization $((A_1(A_2A_3))((A_4A_5)A_6))$.

Exercises**15.2-1**

Find an optimal parenthesization of a matrix-chain product whose sequence of dimensions is $\langle 5, 10, 3, 12, 5, 50, 6 \rangle$.

15.2-2

Give a recursive algorithm `MATRIX-CHAIN-MULTIPLY(A, s, i, j)` that actually performs the optimal matrix-chain multiplication, given the sequence of matrices $\langle A_1, A_2, \dots, A_n \rangle$, the s table computed by `MATRIX-CHAIN-ORDER`, and the indices i and j . (The initial call would be `MATRIX-CHAIN-MULTIPLY($A, s, 1, n$)`.)

15.2-3

Use the substitution method to show that the solution to the recurrence (15.6) is $\Omega(2^n)$.

15.2-4

Describe the subproblem graph for matrix-chain multiplication with an input chain of length n . How many vertices does it have? How many edges does it have, and which edges are they?

15.2-5

Let $R(i, j)$ be the number of times that table entry $m[i, j]$ is referenced while computing other table entries in a call of `MATRIX-CHAIN-ORDER`. Show that the total number of references for the entire table is

$$\sum_{i=1}^n \sum_{j=i}^n R(i, j) = \frac{n^3 - n}{3}.$$

(*Hint:* You may find equation (A.3) useful.)

15.2-6

Show that a full parenthesization of an n -element expression has exactly $n - 1$ pairs of parentheses.

15.3 Elements of dynamic programming

Although we have just worked through two examples of the dynamic-programming method, you might still be wondering just when the method applies. From an engineering perspective, when should we look for a dynamic-programming solution to a problem? In this section, we examine the two key ingredients that an opti-

mization problem must have in order for dynamic programming to apply: optimal substructure and overlapping subproblems. We also revisit and discuss more fully how memoization might help us take advantage of the overlapping-subproblems property in a top-down recursive approach.

Optimal substructure

The first step in solving an optimization problem by dynamic programming is to characterize the structure of an optimal solution. Recall that a problem exhibits **optimal substructure** if an optimal solution to the problem contains within it optimal solutions to subproblems. Whenever a problem exhibits optimal substructure, we have a good clue that dynamic programming might apply. (As Chapter 16 discusses, it also might mean that a greedy strategy applies, however.) In dynamic programming, we build an optimal solution to the problem from optimal solutions to subproblems. Consequently, we must take care to ensure that the range of subproblems we consider includes those used in an optimal solution.

We discovered optimal substructure in both of the problems we have examined in this chapter so far. In Section 15.1, we observed that the optimal way of cutting up a rod of length n (if we make any cuts at all) involves optimally cutting up the two pieces resulting from the first cut. In Section 15.2, we observed that an optimal parenthesization of $A_i A_{i+1} \cdots A_j$ that splits the product between A_k and A_{k+1} contains within it optimal solutions to the problems of parenthesizing $A_i A_{i+1} \cdots A_k$ and $A_{k+1} A_{k+2} \cdots A_j$.

You will find yourself following a common pattern in discovering optimal substructure:

1. You show that a solution to the problem consists of making a choice, such as choosing an initial cut in a rod or choosing an index at which to split the matrix chain. Making this choice leaves one or more subproblems to be solved.
2. You suppose that for a given problem, you are given the choice that leads to an optimal solution. You do not concern yourself yet with how to determine this choice. You just assume that it has been given to you.
3. Given this choice, you determine which subproblems ensue and how to best characterize the resulting space of subproblems.
4. You show that the solutions to the subproblems used within an optimal solution to the problem must themselves be optimal by using a “cut-and-paste” technique. You do so by supposing that each of the subproblem solutions is not optimal and then deriving a contradiction. In particular, by “cutting out” the nonoptimal solution to each subproblem and “pasting in” the optimal one, you show that you can get a better solution to the original problem, thus contradicting your supposition that you already had an optimal solution. If an optimal

solution gives rise to more than one subproblem, they are typically so similar that you can modify the cut-and-paste argument for one to apply to the others with little effort.

To characterize the space of subproblems, a good rule of thumb says to try to keep the space as simple as possible and then expand it as necessary. For example, the space of subproblems that we considered for the rod-cutting problem contained the problems of optimally cutting up a rod of length i for each size i . This subproblem space worked well, and we had no need to try a more general space of subproblems.

Conversely, suppose that we had tried to constrain our subproblem space for matrix-chain multiplication to matrix products of the form $A_1 A_2 \cdots A_j$. As before, an optimal parenthesization must split this product between A_k and A_{k+1} for some $1 \leq k < j$. Unless we could guarantee that k always equals $j - 1$, we would find that we had subproblems of the form $A_1 A_2 \cdots A_k$ and $A_{k+1} A_{k+2} \cdots A_j$, and that the latter subproblem is not of the form $A_1 A_2 \cdots A_j$. For this problem, we needed to allow our subproblems to vary at “both ends,” that is, to allow both i and j to vary in the subproblem $A_i A_{i+1} \cdots A_j$.

Optimal substructure varies across problem domains in two ways:

1. how many subproblems an optimal solution to the original problem uses, and
2. how many choices we have in determining which subproblem(s) to use in an optimal solution.

In the rod-cutting problem, an optimal solution for cutting up a rod of size n uses just one subproblem (of size $n - i$), but we must consider n choices for i in order to determine which one yields an optimal solution. Matrix-chain multiplication for the subchain $A_i A_{i+1} \cdots A_j$ serves as an example with two subproblems and $j - i$ choices. For a given matrix A_k at which we split the product, we have two subproblems—parenthesizing $A_i A_{i+1} \cdots A_k$ and parenthesizing $A_{k+1} A_{k+2} \cdots A_j$ —and we must solve *both* of them optimally. Once we determine the optimal solutions to subproblems, we choose from among $j - i$ candidates for the index k .

Informally, the running time of a dynamic-programming algorithm depends on the product of two factors: the number of subproblems overall and how many choices we look at for each subproblem. In rod cutting, we had $\Theta(n)$ subproblems overall, and at most n choices to examine for each, yielding an $O(n^2)$ running time. Matrix-chain multiplication had $\Theta(n^2)$ subproblems overall, and in each we had at most $n - 1$ choices, giving an $O(n^3)$ running time (actually, a $\Theta(n^3)$ running time, by Exercise 15.2-5).

Usually, the subproblem graph gives an alternative way to perform the same analysis. Each vertex corresponds to a subproblem, and the choices for a sub-

problem are the edges incident to that subproblem. Recall that in rod cutting, the subproblem graph had n vertices and at most n edges per vertex, yielding an $O(n^2)$ running time. For matrix-chain multiplication, if we were to draw the subproblem graph, it would have $\Theta(n^2)$ vertices and each vertex would have degree at most $n - 1$, giving a total of $O(n^3)$ vertices and edges.

Dynamic programming often uses optimal substructure in a bottom-up fashion. That is, we first find optimal solutions to subproblems and, having solved the subproblems, we find an optimal solution to the problem. Finding an optimal solution to the problem entails making a choice among subproblems as to which we will use in solving the problem. The cost of the problem solution is usually the subproblem costs plus a cost that is directly attributable to the choice itself. In rod cutting, for example, first we solved the subproblems of determining optimal ways to cut up rods of length i for $i = 0, 1, \dots, n - 1$, and then we determined which such subproblem yielded an optimal solution for a rod of length n , using equation (15.2). The cost attributable to the choice itself is the term p_i in equation (15.2). In matrix-chain multiplication, we determined optimal parenthesizations of subchains of $A_i A_{i+1} \cdots A_j$, and then we chose the matrix A_k at which to split the product. The cost attributable to the choice itself is the term $p_{i-1} p_k p_j$.

In Chapter 16, we shall examine “greedy algorithms,” which have many similarities to dynamic programming. In particular, problems to which greedy algorithms apply have optimal substructure. One major difference between greedy algorithms and dynamic programming is that instead of first finding optimal solutions to subproblems and then making an informed choice, greedy algorithms first make a “greedy” choice—the choice that looks best at the time—and then solve a resulting subproblem, without bothering to solve all possible related smaller subproblems. Surprisingly, in some cases this strategy works!

Subtleties

You should be careful not to assume that optimal substructure applies when it does not. Consider the following two problems in which we are given a directed graph $G = (V, E)$ and vertices $u, v \in V$.

Unweighted shortest path:³ Find a path from u to v consisting of the fewest edges. Such a path must be simple, since removing a cycle from a path produces a path with fewer edges.

³We use the term “unweighted” to distinguish this problem from that of finding shortest paths with weighted edges, which we shall see in Chapters 24 and 25. We can use the breadth-first search technique of Chapter 22 to solve the unweighted problem.

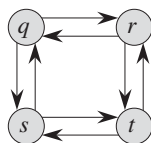


Figure 15.6 A directed graph showing that the problem of finding a longest simple path in an unweighted directed graph does not have optimal substructure. The path $q \rightarrow r \rightarrow t$ is a longest simple path from q to t , but the subpath $q \rightarrow r$ is not a longest simple path from q to r , nor is the subpath $r \rightarrow t$ a longest simple path from r to t .

Unweighted longest simple path: Find a simple path from u to v consisting of the most edges. We need to include the requirement of simplicity because otherwise we can traverse a cycle as many times as we like to create paths with an arbitrarily large number of edges.

The unweighted shortest-path problem exhibits optimal substructure, as follows. Suppose that $u \neq v$, so that the problem is nontrivial. Then, any path p from u to v must contain an intermediate vertex, say w . (Note that w may be u or v .) Thus, we can decompose the path $u \xrightarrow{p} v$ into subpaths $u \xrightarrow{p_1} w \xrightarrow{p_2} v$. Clearly, the number of edges in p equals the number of edges in p_1 plus the number of edges in p_2 . We claim that if p is an optimal (i.e., shortest) path from u to v , then p_1 must be a shortest path from u to w . Why? We use a “cut-and-paste” argument: if there were another path, say p'_1 , from u to w with fewer edges than p_1 , then we could cut out p_1 and paste in p'_1 to produce a path $u \xrightarrow{p'_1} w \xrightarrow{p_2} v$ with fewer edges than p , thus contradicting p ’s optimality. Symmetrically, p_2 must be a shortest path from w to v . Thus, we can find a shortest path from u to v by considering all intermediate vertices w , finding a shortest path from u to w and a shortest path from w to v , and choosing an intermediate vertex w that yields the overall shortest path. In Section 25.2, we use a variant of this observation of optimal substructure to find a shortest path between every pair of vertices on a weighted, directed graph.

You might be tempted to assume that the problem of finding an unweighted longest simple path exhibits optimal substructure as well. After all, if we decompose a longest simple path $u \xrightarrow{p} v$ into subpaths $u \xrightarrow{p_1} w \xrightarrow{p_2} v$, then mustn’t p_1 be a longest simple path from u to w , and mustn’t p_2 be a longest simple path from w to v ? The answer is no! Figure 15.6 supplies an example. Consider the path $q \rightarrow r \rightarrow t$, which is a longest simple path from q to t . Is $q \rightarrow r$ a longest simple path from q to r ? No, for the path $q \rightarrow s \rightarrow t \rightarrow r$ is a simple path that is longer. Is $r \rightarrow t$ a longest simple path from r to t ? No again, for the path $r \rightarrow q \rightarrow s \rightarrow t$ is a simple path that is longer.

This example shows that for longest simple paths, not only does the problem lack optimal substructure, but we cannot necessarily assemble a “legal” solution to the problem from solutions to subproblems. If we combine the longest simple paths $q \rightarrow s \rightarrow t \rightarrow r$ and $r \rightarrow q \rightarrow s \rightarrow t$, we get the path $q \rightarrow s \rightarrow t \rightarrow r \rightarrow q \rightarrow s \rightarrow t$, which is not simple. Indeed, the problem of finding an unweighted longest simple path does not appear to have any sort of optimal substructure. No efficient dynamic-programming algorithm for this problem has ever been found. In fact, this problem is NP-complete, which—as we shall see in Chapter 34—means that we are unlikely to find a way to solve it in polynomial time.

Why is the substructure of a longest simple path so different from that of a shortest path? Although a solution to a problem for both longest and shortest paths uses two subproblems, the subproblems in finding the longest simple path are not *independent*, whereas for shortest paths they are. What do we mean by subproblems being independent? We mean that the solution to one subproblem does not affect the solution to another subproblem of the same problem. For the example of Figure 15.6, we have the problem of finding a longest simple path from q to t with two subproblems: finding longest simple paths from q to r and from r to t . For the first of these subproblems, we choose the path $q \rightarrow s \rightarrow t \rightarrow r$, and so we have also used the vertices s and t . We can no longer use these vertices in the second subproblem, since the combination of the two solutions to subproblems would yield a path that is not simple. If we cannot use vertex t in the second problem, then we cannot solve it at all, since t is required to be on the path that we find, and it is not the vertex at which we are “splicing” together the subproblem solutions (that vertex being r). Because we use vertices s and t in one subproblem solution, we cannot use them in the other subproblem solution. We must use at least one of them to solve the other subproblem, however, and we must use both of them to solve it optimally. Thus, we say that these subproblems are not independent. Looked at another way, using resources in solving one subproblem (those resources being vertices) renders them unavailable for the other subproblem.

Why, then, are the subproblems independent for finding a shortest path? The answer is that by nature, the subproblems do not share resources. We claim that if a vertex w is on a shortest path p from u to v , then we can splice together *any* shortest path $u \stackrel{p_1}{\rightsquigarrow} w$ and *any* shortest path $w \stackrel{p_2}{\rightsquigarrow} v$ to produce a shortest path from u to v . We are assured that, other than w , no vertex can appear in both paths p_1 and p_2 . Why? Suppose that some vertex $x \neq w$ appears in both p_1 and p_2 , so that we can decompose p_1 as $u \stackrel{p_{ux}}{\rightsquigarrow} x \rightsquigarrow w$ and p_2 as $w \rightsquigarrow x \stackrel{p_{xv}}{\rightsquigarrow} v$. By the optimal substructure of this problem, path p has as many edges as p_1 and p_2 together; let’s say that p has e edges. Now let us construct a path $p' = u \stackrel{p_{ux}}{\rightsquigarrow} x \stackrel{p_{xv}}{\rightsquigarrow} v$ from u to v . Because we have excised the paths from x to w and from w to x , each of which contains at least one edge, path p' contains at most $e - 2$ edges, which contradicts

the assumption that p is a shortest path. Thus, we are assured that the subproblems for the shortest-path problem are independent.

Both problems examined in Sections 15.1 and 15.2 have independent subproblems. In matrix-chain multiplication, the subproblems are multiplying subchains $A_i A_{i+1} \cdots A_k$ and $A_{k+1} A_{k+2} \cdots A_j$. These subchains are disjoint, so that no matrix could possibly be included in both of them. In rod cutting, to determine the best way to cut up a rod of length n , we look at the best ways of cutting up rods of length i for $i = 0, 1, \dots, n-1$. Because an optimal solution to the length- n problem includes just one of these subproblem solutions (after we have cut off the first piece), independence of subproblems is not an issue.

Overlapping subproblems

The second ingredient that an optimization problem must have for dynamic programming to apply is that the space of subproblems must be “small” in the sense that a recursive algorithm for the problem solves the same subproblems over and over, rather than always generating new subproblems. Typically, the total number of distinct subproblems is a polynomial in the input size. When a recursive algorithm revisits the same problem repeatedly, we say that the optimization problem has *overlapping subproblems*.⁴ In contrast, a problem for which a divide-and-conquer approach is suitable usually generates brand-new problems at each step of the recursion. Dynamic-programming algorithms typically take advantage of overlapping subproblems by solving each subproblem once and then storing the solution in a table where it can be looked up when needed, using constant time per lookup.

In Section 15.1, we briefly examined how a recursive solution to rod cutting makes exponentially many calls to find solutions of smaller subproblems. Our dynamic-programming solution takes an exponential-time recursive algorithm down to quadratic time.

To illustrate the overlapping-subproblems property in greater detail, let us re-examine the matrix-chain multiplication problem. Referring back to Figure 15.5, observe that MATRIX-CHAIN-ORDER repeatedly looks up the solution to subproblems in lower rows when solving subproblems in higher rows. For example, it references entry $m[3, 4]$ four times: during the computations of $m[2, 4]$, $m[1, 4]$,

⁴It may seem strange that dynamic programming relies on subproblems being both independent and overlapping. Although these requirements may sound contradictory, they describe two different notions, rather than two points on the same axis. Two subproblems of the same problem are independent if they do not share resources. Two subproblems are overlapping if they are really the same subproblem that occurs as a subproblem of different problems.

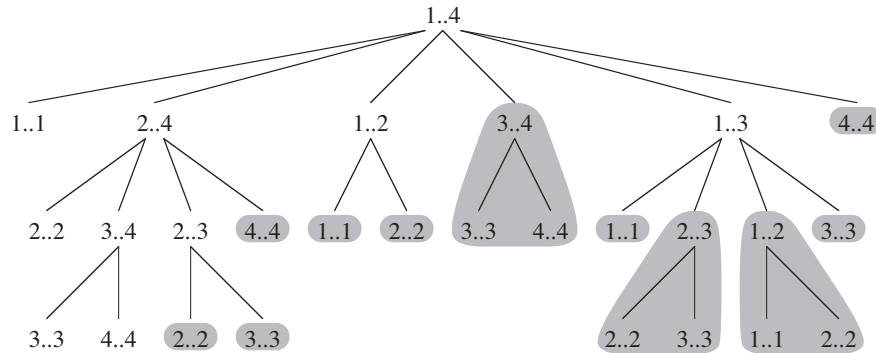


Figure 15.7 The recursion tree for the computation of `RECURSIVE-MATRIX-CHAIN`($p, 1, 4$). Each node contains the parameters i and j . The computations performed in a shaded subtree are replaced by a single table lookup in `MEMOIZED-MATRIX-CHAIN`.

$m[3, 5]$, and $m[3, 6]$. If we were to recompute $m[3, 4]$ each time, rather than just looking it up, the running time would increase dramatically. To see how, consider the following (inefficient) recursive procedure that determines $m[i, j]$, the minimum number of scalar multiplications needed to compute the matrix-chain product $A_{i..j} = A_i A_{i+1} \cdots A_j$. The procedure is based directly on the recurrence (15.7).

`RECURSIVE-MATRIX-CHAIN`(p, i, j)

```

1  if  $i == j$ 
2      return 0
3   $m[i, j] = \infty$ 
4  for  $k = i$  to  $j - 1$ 
5       $q = \text{RECURSIVE-MATRIX-CHAIN}(p, i, k)$ 
        +  $\text{RECURSIVE-MATRIX-CHAIN}(p, k + 1, j)$ 
        +  $p_{i-1} p_k p_j$ 
6      if  $q < m[i, j]$ 
7           $m[i, j] = q$ 
8  return  $m[i, j]$ 
```

Figure 15.7 shows the recursion tree produced by the call `RECURSIVE-MATRIX-CHAIN`($p, 1, 4$). Each node is labeled by the values of the parameters i and j . Observe that some pairs of values occur many times.

In fact, we can show that the time to compute $m[1, n]$ by this recursive procedure is at least exponential in n . Let $T(n)$ denote the time taken by `RECURSIVE-MATRIX-CHAIN` to compute an optimal parenthesization of a chain of n matrices. Because the execution of lines 1–2 and of lines 6–7 each take at least unit time, as

does the multiplication in line 5, inspection of the procedure yields the recurrence

$$\begin{aligned} T(1) &\geq 1, \\ T(n) &\geq 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) \quad \text{for } n > 1. \end{aligned}$$

Noting that for $i = 1, 2, \dots, n-1$, each term $T(i)$ appears once as $T(k)$ and once as $T(n-k)$, and collecting the $n-1$ 1s in the summation together with the 1 out front, we can rewrite the recurrence as

$$T(n) \geq 2 \sum_{i=1}^{n-1} T(i) + n. \quad (15.8)$$

We shall prove that $T(n) = \Omega(2^n)$ using the substitution method. Specifically, we shall show that $T(n) \geq 2^{n-1}$ for all $n \geq 1$. The basis is easy, since $T(1) \geq 1 = 2^0$. Inductively, for $n \geq 2$ we have

$$\begin{aligned} T(n) &\geq 2 \sum_{i=1}^{n-1} 2^{i-1} + n \\ &= 2 \sum_{i=0}^{n-2} 2^i + n \\ &= 2(2^{n-1} - 1) + n \quad (\text{by equation (A.5)}) \\ &= 2^n - 2 + n \\ &\geq 2^{n-1}, \end{aligned}$$

which completes the proof. Thus, the total amount of work performed by the call `RECURSIVE-MATRIX-CHAIN($p, 1, n$)` is at least exponential in n .

Compare this top-down, recursive algorithm (without memoization) with the bottom-up dynamic-programming algorithm. The latter is more efficient because it takes advantage of the overlapping-subproblems property. Matrix-chain multiplication has only $\Theta(n^2)$ distinct subproblems, and the dynamic-programming algorithm solves each exactly once. The recursive algorithm, on the other hand, must again solve each subproblem every time it reappears in the recursion tree. Whenever a recursion tree for the natural recursive solution to a problem contains the same subproblem repeatedly, and the total number of distinct subproblems is small, dynamic programming can improve efficiency, sometimes dramatically.

Reconstructing an optimal solution

As a practical matter, we often store which choice we made in each subproblem in a table so that we do not have to reconstruct this information from the costs that we stored.

For matrix-chain multiplication, the table $s[i, j]$ saves us a significant amount of work when reconstructing an optimal solution. Suppose that we did not maintain the $s[i, j]$ table, having filled in only the table $m[i, j]$ containing optimal subproblem costs. We choose from among $j - i$ possibilities when we determine which subproblems to use in an optimal solution to parenthesizing $A_i A_{i+1} \cdots A_j$, and $j - i$ is not a constant. Therefore, it would take $\Theta(j - i) = \omega(1)$ time to reconstruct which subproblems we chose for a solution to a given problem. By storing in $s[i, j]$ the index of the matrix at which we split the product $A_i A_{i+1} \cdots A_j$, we can reconstruct each choice in $O(1)$ time.

Memoization

As we saw for the rod-cutting problem, there is an alternative approach to dynamic programming that often offers the efficiency of the bottom-up dynamic-programming approach while maintaining a top-down strategy. The idea is to **memoize** the natural, but inefficient, recursive algorithm. As in the bottom-up approach, we maintain a table with subproblem solutions, but the control structure for filling in the table is more like the recursive algorithm.

A memoized recursive algorithm maintains an entry in a table for the solution to each subproblem. Each table entry initially contains a special value to indicate that the entry has yet to be filled in. When the subproblem is first encountered as the recursive algorithm unfolds, its solution is computed and then stored in the table. Each subsequent time that we encounter this subproblem, we simply look up the value stored in the table and return it.⁵

Here is a memoized version of `RECURSIVE-MATRIX-CHAIN`. Note where it resembles the memoized top-down method for the rod-cutting problem.

⁵This approach presupposes that we know the set of all possible subproblem parameters and that we have established the relationship between table positions and subproblems. Another, more general, approach is to memoize by using hashing with the subproblem parameters as keys.

MEMOIZED-MATRIX-CHAIN(p)

```

1   $n = p.length - 1$ 
2  let  $m[1..n, 1..n]$  be a new table
3  for  $i = 1$  to  $n$ 
4      for  $j = i$  to  $n$ 
5           $m[i, j] = \infty$ 
6  return LOOKUP-CHAIN( $m, p, 1, n$ )

```

LOOKUP-CHAIN(m, p, i, j)

```

1  if  $m[i, j] < \infty$ 
2      return  $m[i, j]$ 
3  if  $i == j$ 
4       $m[i, j] = 0$ 
5  else for  $k = i$  to  $j - 1$ 
6       $q = \text{LOOKUP-CHAIN}(m, p, i, k)$ 
         $+ \text{LOOKUP-CHAIN}(m, p, k + 1, j) + p_{i-1}p_kp_j$ 
7      if  $q < m[i, j]$ 
8           $m[i, j] = q$ 
9  return  $m[i, j]$ 

```

The MEMOIZED-MATRIX-CHAIN procedure, like MATRIX-CHAIN-ORDER, maintains a table $m[1..n, 1..n]$ of computed values of $m[i, j]$, the minimum number of scalar multiplications needed to compute the matrix $A_{i..j}$. Each table entry initially contains the value ∞ to indicate that the entry has yet to be filled in. Upon calling LOOKUP-CHAIN(m, p, i, j), if line 1 finds that $m[i, j] < \infty$, then the procedure simply returns the previously computed cost $m[i, j]$ in line 2. Otherwise, the cost is computed as in RECURSIVE-MATRIX-CHAIN, stored in $m[i, j]$, and returned. Thus, LOOKUP-CHAIN(m, p, i, j) always returns the value of $m[i, j]$, but it computes it only upon the first call of LOOKUP-CHAIN with these specific values of i and j .

Figure 15.7 illustrates how MEMOIZED-MATRIX-CHAIN saves time compared with RECURSIVE-MATRIX-CHAIN. Shaded subtrees represent values that it looks up rather than recomputes.

Like the bottom-up dynamic-programming algorithm MATRIX-CHAIN-ORDER, the procedure MEMOIZED-MATRIX-CHAIN runs in $O(n^3)$ time. Line 5 of MEMOIZED-MATRIX-CHAIN executes $\Theta(n^2)$ times. We can categorize the calls of LOOKUP-CHAIN into two types:

1. calls in which $m[i, j] = \infty$, so that lines 3–9 execute, and
2. calls in which $m[i, j] < \infty$, so that LOOKUP-CHAIN simply returns in line 2.

There are $\Theta(n^2)$ calls of the first type, one per table entry. All calls of the second type are made as recursive calls by calls of the first type. Whenever a given call of LOOKUP-CHAIN makes recursive calls, it makes $O(n)$ of them. Therefore, there are $O(n^3)$ calls of the second type in all. Each call of the second type takes $O(1)$ time, and each call of the first type takes $O(n)$ time plus the time spent in its recursive calls. The total time, therefore, is $O(n^3)$. Memoization thus turns an $\Omega(2^n)$ -time algorithm into an $O(n^3)$ -time algorithm.

In summary, we can solve the matrix-chain multiplication problem by either a top-down, memoized dynamic-programming algorithm or a bottom-up dynamic-programming algorithm in $O(n^3)$ time. Both methods take advantage of the overlapping-subproblems property. There are only $\Theta(n^2)$ distinct subproblems in total, and either of these methods computes the solution to each subproblem only once. Without memoization, the natural recursive algorithm runs in exponential time, since solved subproblems are repeatedly solved.

In general practice, if all subproblems must be solved at least once, a bottom-up dynamic-programming algorithm usually outperforms the corresponding top-down memoized algorithm by a constant factor, because the bottom-up algorithm has no overhead for recursion and less overhead for maintaining the table. Moreover, for some problems we can exploit the regular pattern of table accesses in the dynamic-programming algorithm to reduce time or space requirements even further. Alternatively, if some subproblems in the subproblem space need not be solved at all, the memoized solution has the advantage of solving only those subproblems that are definitely required.

Exercises

15.3-1

Which is a more efficient way to determine the optimal number of multiplications in a matrix-chain multiplication problem: enumerating all the ways of parenthesizing the product and computing the number of multiplications for each, or running RECURSIVE-MATRIX-CHAIN? Justify your answer.

15.3-2

Draw the recursion tree for the MERGE-SORT procedure from Section 2.3.1 on an array of 16 elements. Explain why memoization fails to speed up a good divide-and-conquer algorithm such as MERGE-SORT.

15.3-3

Consider a variant of the matrix-chain multiplication problem in which the goal is to parenthesize the sequence of matrices so as to maximize, rather than minimize,

the number of scalar multiplications. Does this problem exhibit optimal substructure?

15.3-4

As stated, in dynamic programming we first solve the subproblems and then choose which of them to use in an optimal solution to the problem. Professor Capulet claims that we do not always need to solve all the subproblems in order to find an optimal solution. She suggests that we can find an optimal solution to the matrix-chain multiplication problem by always choosing the matrix A_k at which to split the subproduct $A_i A_{i+1} \cdots A_j$ (by selecting k to minimize the quantity $p_{i-1} p_k p_j$) *before* solving the subproblems. Find an instance of the matrix-chain multiplication problem for which this greedy approach yields a suboptimal solution.

15.3-5

Suppose that in the rod-cutting problem of Section 15.1, we also had limit l_i on the number of pieces of length i that we are allowed to produce, for $i = 1, 2, \dots, n$. Show that the optimal-substructure property described in Section 15.1 no longer holds.

15.3-6

Imagine that you wish to exchange one currency for another. You realize that instead of directly exchanging one currency for another, you might be better off making a series of trades through other currencies, winding up with the currency you want. Suppose that you can trade n different currencies, numbered $1, 2, \dots, n$, where you start with currency 1 and wish to wind up with currency n . You are given, for each pair of currencies i and j , an exchange rate r_{ij} , meaning that if you start with d units of currency i , you can trade for dr_{ij} units of currency j . A sequence of trades may entail a commission, which depends on the number of trades you make. Let c_k be the commission that you are charged when you make k trades. Show that, if $c_k = 0$ for all $k = 1, 2, \dots, n$, then the problem of finding the best sequence of exchanges from currency 1 to currency n exhibits optimal substructure. Then show that if commissions c_k are arbitrary values, then the problem of finding the best sequence of exchanges from currency 1 to currency n does not necessarily exhibit optimal substructure.

15.4 Longest common subsequence

Biological applications often need to compare the DNA of two (or more) different organisms. A strand of DNA consists of a string of molecules called

bases, where the possible bases are adenine, guanine, cytosine, and thymine. Representing each of these bases by its initial letter, we can express a strand of DNA as a string over the finite set $\{A, C, G, T\}$. (See Appendix C for the definition of a string.) For example, the DNA of one organism may be $S_1 = \text{ACCGGTCGAGTGC GCGGAAGCCGGCCGAA}$, and the DNA of another organism may be $S_2 = \text{GTCGTTCGGAATGCCGTTGCTCTGTAAA}$. One reason to compare two strands of DNA is to determine how “similar” the two strands are, as some measure of how closely related the two organisms are. We can, and do, define similarity in many different ways. For example, we can say that two DNA strands are similar if one is a substring of the other. (Chapter 32 explores algorithms to solve this problem.) In our example, neither S_1 nor S_2 is a substring of the other. Alternatively, we could say that two strands are similar if the number of changes needed to turn one into the other is small. (Problem 15-5 looks at this notion.) Yet another way to measure the similarity of strands S_1 and S_2 is by finding a third strand S_3 in which the bases in S_3 appear in each of S_1 and S_2 ; these bases must appear in the same order, but not necessarily consecutively. The longer the strand S_3 we can find, the more similar S_1 and S_2 are. In our example, the longest strand S_3 is $\text{GTCGTCGGAAGCCGGCCGAA}$.

We formalize this last notion of similarity as the longest-common-subsequence problem. A subsequence of a given sequence is just the given sequence with zero or more elements left out. Formally, given a sequence $X = \langle x_1, x_2, \dots, x_m \rangle$, another sequence $Z = \langle z_1, z_2, \dots, z_k \rangle$ is a **subsequence** of X if there exists a strictly increasing sequence $\langle i_1, i_2, \dots, i_k \rangle$ of indices of X such that for all $j = 1, 2, \dots, k$, we have $x_{i_j} = z_j$. For example, $Z = \langle B, C, D, B \rangle$ is a subsequence of $X = \langle A, B, C, B, D, A, B \rangle$ with corresponding index sequence $\langle 2, 3, 5, 7 \rangle$.

Given two sequences X and Y , we say that a sequence Z is a **common subsequence** of X and Y if Z is a subsequence of both X and Y . For example, if $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$, the sequence $\langle B, C, A \rangle$ is a common subsequence of both X and Y . The sequence $\langle B, C, A \rangle$ is not a **longest common subsequence (LCS)** of X and Y , however, since it has length 3 and the sequence $\langle B, C, B, A \rangle$, which is also common to both X and Y , has length 4. The sequence $\langle B, C, B, A \rangle$ is an LCS of X and Y , as is the sequence $\langle B, D, A, B \rangle$, since X and Y have no common subsequence of length 5 or greater.

In the **longest-common-subsequence problem**, we are given two sequences $X = \langle x_1, x_2, \dots, x_m \rangle$ and $Y = \langle y_1, y_2, \dots, y_n \rangle$ and wish to find a maximum-length common subsequence of X and Y . This section shows how to efficiently solve the LCS problem using dynamic programming.

Step 1: Characterizing a longest common subsequence

In a brute-force approach to solving the LCS problem, we would enumerate all subsequences of X and check each subsequence to see whether it is also a subsequence of Y , keeping track of the longest subsequence we find. Each subsequence of X corresponds to a subset of the indices $\{1, 2, \dots, m\}$ of X . Because X has 2^m subsequences, this approach requires exponential time, making it impractical for long sequences.

The LCS problem has an optimal-substructure property, however, as the following theorem shows. As we shall see, the natural classes of subproblems correspond to pairs of “prefixes” of the two input sequences. To be precise, given a sequence $X = \langle x_1, x_2, \dots, x_m \rangle$, we define the i th **prefix** of X , for $i = 0, 1, \dots, m$, as $X_i = \langle x_1, x_2, \dots, x_i \rangle$. For example, if $X = \langle A, B, C, B, D, A, B \rangle$, then $X_4 = \langle A, B, C, B \rangle$ and X_0 is the empty sequence.

Theorem 15.1 (Optimal substructure of an LCS)

Let $X = \langle x_1, x_2, \dots, x_m \rangle$ and $Y = \langle y_1, y_2, \dots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \dots, z_k \rangle$ be any LCS of X and Y .

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1} .
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y .
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1} .

Proof (1) If $z_k \neq x_m$, then we could append $x_m = y_n$ to Z to obtain a common subsequence of X and Y of length $k + 1$, contradicting the supposition that Z is a *longest* common subsequence of X and Y . Thus, we must have $z_k = x_m = y_n$. Now, the prefix Z_{k-1} is a length- $(k - 1)$ common subsequence of X_{m-1} and Y_{n-1} . We wish to show that it is an LCS. Suppose for the purpose of contradiction that there exists a common subsequence W of X_{m-1} and Y_{n-1} with length greater than $k - 1$. Then, appending $x_m = y_n$ to W produces a common subsequence of X and Y whose length is greater than k , which is a contradiction.

(2) If $z_k \neq x_m$, then Z is a common subsequence of X_{m-1} and Y . If there were a common subsequence W of X_{m-1} and Y with length greater than k , then W would also be a common subsequence of X_m and Y , contradicting the assumption that Z is an LCS of X and Y .

(3) The proof is symmetric to (2). ■

The way that Theorem 15.1 characterizes longest common subsequences tells us that an LCS of two sequences contains within it an LCS of prefixes of the two sequences. Thus, the LCS problem has an optimal-substructure property. A recur-

sive solution also has the overlapping-subproblems property, as we shall see in a moment.

Step 2: A recursive solution

Theorem 15.1 implies that we should examine either one or two subproblems when finding an LCS of $X = \langle x_1, x_2, \dots, x_m \rangle$ and $Y = \langle y_1, y_2, \dots, y_n \rangle$. If $x_m = y_n$, we must find an LCS of X_{m-1} and Y_{n-1} . Appending $x_m = y_n$ to this LCS yields an LCS of X and Y . If $x_m \neq y_n$, then we must solve two subproblems: finding an LCS of X_{m-1} and Y and finding an LCS of X and Y_{n-1} . Whichever of these two LCSs is longer is an LCS of X and Y . Because these cases exhaust all possibilities, we know that one of the optimal subproblem solutions must appear within an LCS of X and Y .

We can readily see the overlapping-subproblems property in the LCS problem. To find an LCS of X and Y , we may need to find the LCSs of X and Y_{n-1} and of X_{m-1} and Y . But each of these subproblems has the subsubproblem of finding an LCS of X_{m-1} and Y_{n-1} . Many other subproblems share subsubproblems.

As in the matrix-chain multiplication problem, our recursive solution to the LCS problem involves establishing a recurrence for the value of an optimal solution. Let us define $c[i, j]$ to be the length of an LCS of the sequences X_i and Y_j . If either $i = 0$ or $j = 0$, one of the sequences has length 0, and so the LCS has length 0. The optimal substructure of the LCS problem gives the recursive formula

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i - 1, j - 1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max(c[i, j - 1], c[i - 1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases} \quad (15.9)$$

Observe that in this recursive formulation, a condition in the problem restricts which subproblems we may consider. When $x_i = y_j$, we can and should consider the subproblem of finding an LCS of X_{i-1} and Y_{j-1} . Otherwise, we instead consider the two subproblems of finding an LCS of X_i and Y_{j-1} and of X_{i-1} and Y_j . In the previous dynamic-programming algorithms we have examined—for rod cutting and matrix-chain multiplication—we ruled out no subproblems due to conditions in the problem. Finding an LCS is not the only dynamic-programming algorithm that rules out subproblems based on conditions in the problem. For example, the edit-distance problem (see Problem 15-5) has this characteristic.

Step 3: Computing the length of an LCS

Based on equation (15.9), we could easily write an exponential-time recursive algorithm to compute the length of an LCS of two sequences. Since the LCS problem

has only $\Theta(mn)$ distinct subproblems, however, we can use dynamic programming to compute the solutions bottom up.

Procedure **LCS-LENGTH** takes two sequences $X = \langle x_1, x_2, \dots, x_m \rangle$ and $Y = \langle y_1, y_2, \dots, y_n \rangle$ as inputs. It stores the $c[i, j]$ values in a table $c[0..m, 0..n]$, and it computes the entries in **row-major** order. (That is, the procedure fills in the first row of c from left to right, then the second row, and so on.) The procedure also maintains the table $b[1..m, 1..n]$ to help us construct an optimal solution. Intuitively, $b[i, j]$ points to the table entry corresponding to the optimal subproblem solution chosen when computing $c[i, j]$. The procedure returns the b and c tables; $c[m, n]$ contains the length of an LCS of X and Y .

```

LCS-LENGTH( $X, Y$ )
1   $m = X.length$ 
2   $n = Y.length$ 
3  let  $b[1..m, 1..n]$  and  $c[0..m, 0..n]$  be new tables
4  for  $i = 1$  to  $m$ 
5       $c[i, 0] = 0$ 
6  for  $j = 0$  to  $n$ 
7       $c[0, j] = 0$ 
8  for  $i = 1$  to  $m$ 
9      for  $j = 1$  to  $n$ 
10         if  $x_i == y_j$ 
11              $c[i, j] = c[i - 1, j - 1] + 1$ 
12              $b[i, j] = \nwarrow$ 
13         elseif  $c[i - 1, j] \geq c[i, j - 1]$ 
14              $c[i, j] = c[i - 1, j]$ 
15              $b[i, j] = \uparrow$ 
16         else  $c[i, j] = c[i, j - 1]$ 
17              $b[i, j] = \leftarrow$ 
18  return  $c$  and  $b$ 

```

Figure 15.8 shows the tables produced by **LCS-LENGTH** on the sequences $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$. The running time of the procedure is $\Theta(mn)$, since each table entry takes $\Theta(1)$ time to compute.

Step 4: Constructing an LCS

The b table returned by **LCS-LENGTH** enables us to quickly construct an LCS of $X = \langle x_1, x_2, \dots, x_m \rangle$ and $Y = \langle y_1, y_2, \dots, y_n \rangle$. We simply begin at $b[m, n]$ and trace through the table by following the arrows. Whenever we encounter a “ \nwarrow ” in entry $b[i, j]$, it implies that $x_i = y_j$ is an element of the LCS that **LCS-LENGTH**

		j	0	1	2	3	4	5	6
i	x_i	y_j		B	D	C	A	B	A
0			0	0	0	0	0	0	0
1	A		0	↑	↑	↑	↖	←	↖
2	B		0	↖	1	←	1	↖	←
3	C		0	↑	↑	↑	2	←	2
4	B		0	↖	1	↑	2	↖	←
5	D		0	↑	↖	2	2	↑	3
6	A		0	↑	↑	↑	↖	↑	↖
7	B		0	↖	↑	↑	↑	↖	↑

Figure 15.8 The c and b tables computed by LCS-LENGTH on the sequences $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$. The square in row i and column j contains the value of $c[i, j]$ and the appropriate arrow for the value of $b[i, j]$. The entry 4 in $c[7, 6]$ —the lower right-hand corner of the table—is the length of an LCS $\langle B, C, B, A \rangle$ of X and Y . For $i, j > 0$, entry $c[i, j]$ depends only on whether $x_i = y_j$ and the values in entries $c[i - 1, j]$, $c[i, j - 1]$, and $c[i - 1, j - 1]$, which are computed before $c[i, j]$. To reconstruct the elements of an LCS, follow the $b[i, j]$ arrows from the lower right-hand corner; the sequence is shaded. Each “↖” on the shaded sequence corresponds to an entry (highlighted) for which $x_i = y_j$ is a member of an LCS.

found. With this method, we encounter the elements of this LCS in reverse order. The following recursive procedure prints out an LCS of X and Y in the proper, forward order. The initial call is PRINT-LCS($b, X, X.length, Y.length$).

```

PRINT-LCS( $b, X, i, j$ )
1  if  $i == 0$  or  $j == 0$ 
2      return
3  if  $b[i, j] == \text{“}\nwarrow\text{”}$ 
4      PRINT-LCS( $b, X, i - 1, j - 1$ )
5      print  $x_i$ 
6  elseif  $b[i, j] == \text{“}\uparrow\text{”}$ 
7      PRINT-LCS( $b, X, i - 1, j$ )
8  else PRINT-LCS( $b, X, i, j - 1$ )

```

For the b table in Figure 15.8, this procedure prints $BCBA$. The procedure takes time $O(m + n)$, since it decrements at least one of i and j in each recursive call.

Improving the code

Once you have developed an algorithm, you will often find that you can improve on the time or space it uses. Some changes can simplify the code and improve constant factors but otherwise yield no asymptotic improvement in performance. Others can yield substantial asymptotic savings in time and space.

In the LCS algorithm, for example, we can eliminate the b table altogether. Each $c[i, j]$ entry depends on only three other c table entries: $c[i - 1, j - 1]$, $c[i - 1, j]$, and $c[i, j - 1]$. Given the value of $c[i, j]$, we can determine in $O(1)$ time which of these three values was used to compute $c[i, j]$, without inspecting table b . Thus, we can reconstruct an LCS in $O(m + n)$ time using a procedure similar to PRINT-LCS. (Exercise 15.4-2 asks you to give the pseudocode.) Although we save $\Theta(mn)$ space by this method, the auxiliary space requirement for computing an LCS does not asymptotically decrease, since we need $\Theta(mn)$ space for the c table anyway.

We can, however, reduce the asymptotic space requirements for LCS-LENGTH, since it needs only two rows of table c at a time: the row being computed and the previous row. (In fact, as Exercise 15.4-4 asks you to show, we can use only slightly more than the space for one row of c to compute the length of an LCS.) This improvement works if we need only the length of an LCS; if we need to reconstruct the elements of an LCS, the smaller table does not keep enough information to retrace our steps in $O(m + n)$ time.

Exercises

15.4-1

Determine an LCS of $\langle 1, 0, 0, 1, 0, 1, 0, 1 \rangle$ and $\langle 0, 1, 0, 1, 1, 0, 1, 1, 0 \rangle$.

15.4-2

Give pseudocode to reconstruct an LCS from the completed c table and the original sequences $X = \langle x_1, x_2, \dots, x_m \rangle$ and $Y = \langle y_1, y_2, \dots, y_n \rangle$ in $O(m + n)$ time, without using the b table.

15.4-3

Give a memoized version of LCS-LENGTH that runs in $O(mn)$ time.

15.4-4

Show how to compute the length of an LCS using only $2 \cdot \min(m, n)$ entries in the c table plus $O(1)$ additional space. Then show how to do the same thing, but using $\min(m, n)$ entries plus $O(1)$ additional space.

15.4-5

Give an $O(n^2)$ -time algorithm to find the longest monotonically increasing subsequence of a sequence of n numbers.

15.4-6 ★

Give an $O(n \lg n)$ -time algorithm to find the longest monotonically increasing subsequence of a sequence of n numbers. (*Hint:* Observe that the last element of a candidate subsequence of length i is at least as large as the last element of a candidate subsequence of length $i - 1$. Maintain candidate subsequences by linking them through the input sequence.)

15.5 Optimal binary search trees

Suppose that we are designing a program to translate text from English to French. For each occurrence of each English word in the text, we need to look up its French equivalent. We could perform these lookup operations by building a binary search tree with n English words as keys and their French equivalents as satellite data. Because we will search the tree for each individual word in the text, we want the total time spent searching to be as low as possible. We could ensure an $O(\lg n)$ search time per occurrence by using a red-black tree or any other balanced binary search tree. Words appear with different frequencies, however, and a frequently used word such as *the* may appear far from the root while a rarely used word such as *machicolation* appears near the root. Such an organization would slow down the translation, since the number of nodes visited when searching for a key in a binary search tree equals one plus the depth of the node containing the key. We want words that occur frequently in the text to be placed nearer the root.⁶ Moreover, some words in the text might have no French translation,⁷ and such words would not appear in the binary search tree at all. How do we organize a binary search tree so as to minimize the number of nodes visited in all searches, given that we know how often each word occurs?

What we need is known as an **optimal binary search tree**. Formally, we are given a sequence $K = \langle k_1, k_2, \dots, k_n \rangle$ of n distinct keys in sorted order (so that $k_1 < k_2 < \dots < k_n$), and we wish to build a binary search tree from these keys. For each key k_i , we have a probability p_i that a search will be for k_i . Some searches may be for values not in K , and so we also have $n + 1$ “dummy keys”

⁶If the subject of the text is castle architecture, we might want *machicolation* to appear near the root.

⁷Yes, *machicolation* has a French counterpart: *mâchicoulis*.

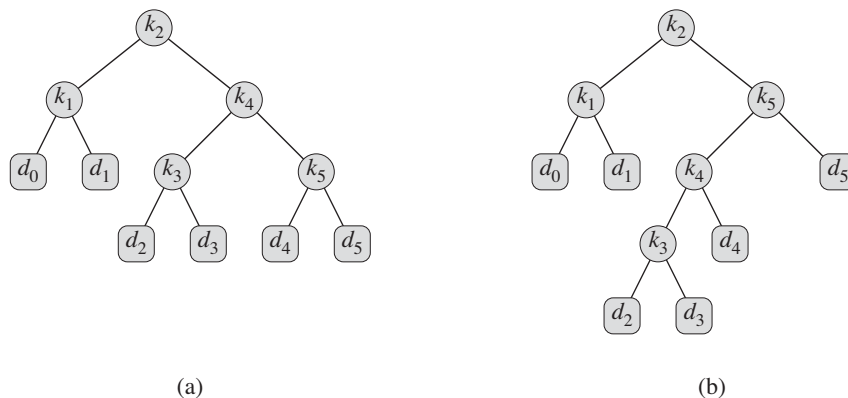


Figure 15.9 Two binary search trees for a set of $n = 5$ keys with the following probabilities:

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

(a) A binary search tree with expected search cost 2.80. (b) A binary search tree with expected search cost 2.75. This tree is optimal.

$d_0, d_1, d_2, \dots, d_n$ representing values not in K . In particular, d_0 represents all values less than k_1 , d_n represents all values greater than k_n , and for $i = 1, 2, \dots, n-1$, the dummy key d_i represents all values between k_i and k_{i+1} . For each dummy key d_i , we have a probability q_i that a search will correspond to d_i . Figure 15.9 shows two binary search trees for a set of $n = 5$ keys. Each key k_i is an internal node, and each dummy key d_i is a leaf. Every search is either successful (finding some key k_i) or unsuccessful (finding some dummy key d_i), and so we have

$$\sum_{i=1}^n p_i + \sum_{i=0}^n q_i = 1. \quad (15.10)$$

Because we have probabilities of searches for each key and each dummy key, we can determine the expected cost of a search in a given binary search tree T . Let us assume that the actual cost of a search equals the number of nodes examined, i.e., the depth of the node found by the search in T , plus 1. Then the expected cost of a search in T is

$$\begin{aligned} E[\text{search cost in } T] &= \sum_{i=1}^n (\text{depth}_T(k_i) + 1) \cdot p_i + \sum_{i=0}^n (\text{depth}_T(d_i) + 1) \cdot q_i \\ &= 1 + \sum_{i=1}^n \text{depth}_T(k_i) \cdot p_i + \sum_{i=0}^n \text{depth}_T(d_i) \cdot q_i, \end{aligned} \quad (15.11)$$

where depth_T denotes a node's depth in the tree T . The last equality follows from equation (15.10). In Figure 15.9(a), we can calculate the expected search cost node by node:

node	depth	probability	contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	2	0.05	0.15
k_4	1	0.10	0.20
k_5	2	0.20	0.60
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	3	0.05	0.20
d_3	3	0.05	0.20
d_4	3	0.05	0.20
d_5	3	0.10	0.40
Total			2.80

For a given set of probabilities, we wish to construct a binary search tree whose expected search cost is smallest. We call such a tree an *optimal binary search tree*. Figure 15.9(b) shows an optimal binary search tree for the probabilities given in the figure caption; its expected cost is 2.75. This example shows that an optimal binary search tree is not necessarily a tree whose overall height is smallest. Nor can we necessarily construct an optimal binary search tree by always putting the key with the greatest probability at the root. Here, key k_5 has the greatest search probability of any key, yet the root of the optimal binary search tree shown is k_2 . (The lowest expected cost of any binary search tree with k_5 at the root is 2.85.)

As with matrix-chain multiplication, exhaustive checking of all possibilities fails to yield an efficient algorithm. We can label the nodes of any n -node binary tree with the keys k_1, k_2, \dots, k_n to construct a binary search tree, and then add in the dummy keys as leaves. In Problem 12-4, we saw that the number of binary trees with n nodes is $\Omega(4^n/n^{3/2})$, and so we would have to examine an exponential number of binary search trees in an exhaustive search. Not surprisingly, we shall solve this problem with dynamic programming.

Step 1: The structure of an optimal binary search tree

To characterize the optimal substructure of optimal binary search trees, we start with an observation about subtrees. Consider any subtree of a binary search tree. It must contain keys in a contiguous range k_i, \dots, k_j , for some $1 \leq i \leq j \leq n$. In addition, a subtree that contains keys k_i, \dots, k_j must also have as its leaves the dummy keys d_{i-1}, \dots, d_j .

Now we can state the optimal substructure: if an optimal binary search tree T has a subtree T' containing keys k_i, \dots, k_j , then this subtree T' must be optimal as

well for the subproblem with keys k_i, \dots, k_j and dummy keys d_{i-1}, \dots, d_j . The usual cut-and-paste argument applies. If there were a subtree T'' whose expected cost is lower than that of T' , then we could cut T' out of T and paste in T'' , resulting in a binary search tree of lower expected cost than T , thus contradicting the optimality of T .

We need to use the optimal substructure to show that we can construct an optimal solution to the problem from optimal solutions to subproblems. Given keys k_i, \dots, k_j , one of these keys, say k_r ($i \leq r \leq j$), is the root of an optimal subtree containing these keys. The left subtree of the root k_r contains the keys k_i, \dots, k_{r-1} (and dummy keys d_{i-1}, \dots, d_{r-1}), and the right subtree contains the keys k_{r+1}, \dots, k_j (and dummy keys d_r, \dots, d_j). As long as we examine all candidate roots k_r , where $i \leq r \leq j$, and we determine all optimal binary search trees containing k_i, \dots, k_{r-1} and those containing k_{r+1}, \dots, k_j , we are guaranteed that we will find an optimal binary search tree.

There is one detail worth noting about “empty” subtrees. Suppose that in a subtree with keys k_i, \dots, k_j , we select k_i as the root. By the above argument, k_i ’s left subtree contains the keys k_i, \dots, k_{i-1} . We interpret this sequence as containing no keys. Bear in mind, however, that subtrees also contain dummy keys. We adopt the convention that a subtree containing keys k_i, \dots, k_{i-1} has no actual keys but does contain the single dummy key d_{i-1} . Symmetrically, if we select k_j as the root, then k_j ’s right subtree contains the keys k_{j+1}, \dots, k_j ; this right subtree contains no actual keys, but it does contain the dummy key d_j .

Step 2: A recursive solution

We are ready to define the value of an optimal solution recursively. We pick our subproblem domain as finding an optimal binary search tree containing the keys k_i, \dots, k_j , where $i \geq 1$, $j \leq n$, and $j \geq i - 1$. (When $j = i - 1$, there are no actual keys; we have just the dummy key d_{i-1} .) Let us define $e[i, j]$ as the expected cost of searching an optimal binary search tree containing the keys k_i, \dots, k_j . Ultimately, we wish to compute $e[1, n]$.

The easy case occurs when $j = i - 1$. Then we have just the dummy key d_{i-1} . The expected search cost is $e[i, i - 1] = q_{i-1}$.

When $j \geq i$, we need to select a root k_r from among k_i, \dots, k_j and then make an optimal binary search tree with keys k_i, \dots, k_{r-1} as its left subtree and an optimal binary search tree with keys k_{r+1}, \dots, k_j as its right subtree. What happens to the expected search cost of a subtree when it becomes a subtree of a node? The depth of each node in the subtree increases by 1. By equation (15.11), the expected search cost of this subtree increases by the sum of all the probabilities in the subtree. For a subtree with keys k_i, \dots, k_j , let us denote this sum of probabilities as

$$w(i, j) = \sum_{l=i}^j p_l + \sum_{l=i-1}^j q_l . \quad (15.12)$$

Thus, if k_r is the root of an optimal subtree containing keys k_i, \dots, k_j , we have

$$e[i, j] = p_r + (e[i, r-1] + w(i, r-1)) + (e[r+1, j] + w(r+1, j)) .$$

Noting that

$$w(i, j) = w(i, r-1) + p_r + w(r+1, j) ,$$

we rewrite $e[i, j]$ as

$$e[i, j] = e[i, r-1] + e[r+1, j] + w(i, j) . \quad (15.13)$$

The recursive equation (15.13) assumes that we know which node k_r to use as the root. We choose the root that gives the lowest expected search cost, giving us our final recursive formulation:

$$e[i, j] = \begin{cases} q_{i-1} & \text{if } j = i - 1 , \\ \min_{i \leq r \leq j} \{e[i, r-1] + e[r+1, j] + w(i, j)\} & \text{if } i \leq j . \end{cases} \quad (15.14)$$

The $e[i, j]$ values give the expected search costs in optimal binary search trees. To help us keep track of the structure of optimal binary search trees, we define $root[i, j]$, for $1 \leq i \leq j \leq n$, to be the index r for which k_r is the root of an optimal binary search tree containing keys k_i, \dots, k_j . Although we will see how to compute the values of $root[i, j]$, we leave the construction of an optimal binary search tree from these values as Exercise 15.5-1.

Step 3: Computing the expected search cost of an optimal binary search tree

At this point, you may have noticed some similarities between our characterizations of optimal binary search trees and matrix-chain multiplication. For both problem domains, our subproblems consist of contiguous index subranges. A direct, recursive implementation of equation (15.14) would be as inefficient as a direct, recursive matrix-chain multiplication algorithm. Instead, we store the $e[i, j]$ values in a table $e[1 \dots n+1, 0 \dots n]$. The first index needs to run to $n+1$ rather than n because in order to have a subtree containing only the dummy key d_n , we need to compute and store $e[n+1, n]$. The second index needs to start from 0 because in order to have a subtree containing only the dummy key d_0 , we need to compute and store $e[1, 0]$. We use only the entries $e[i, j]$ for which $j \geq i-1$. We also use a table $root[i, j]$, for recording the root of the subtree containing keys k_i, \dots, k_j . This table uses only the entries for which $1 \leq i \leq j \leq n$.

We will need one other table for efficiency. Rather than compute the value of $w(i, j)$ from scratch every time we are computing $e[i, j]$ —which would take

$\Theta(j - i)$ additions—we store these values in a table $w[1 \dots n + 1, 0 \dots n]$. For the base case, we compute $w[i, i - 1] = q_{i-1}$ for $1 \leq i \leq n + 1$. For $j \geq i$, we compute

$$w[i, j] = w[i, j - 1] + p_j + q_j. \quad (15.15)$$

Thus, we can compute the $\Theta(n^2)$ values of $w[i, j]$ in $\Theta(1)$ time each.

The pseudocode that follows takes as inputs the probabilities p_1, \dots, p_n and q_0, \dots, q_n and the size n , and it returns the tables e and $root$.

OPTIMAL-BST(p, q, n)

```

1  let  $e[1 \dots n + 1, 0 \dots n]$ ,  $w[1 \dots n + 1, 0 \dots n]$ ,
   and  $root[1 \dots n, 1 \dots n]$  be new tables
2  for  $i = 1$  to  $n + 1$ 
3       $e[i, i - 1] = q_{i-1}$ 
4       $w[i, i - 1] = q_{i-1}$ 
5  for  $l = 1$  to  $n$ 
6      for  $i = 1$  to  $n - l + 1$ 
7           $j = i + l - 1$ 
8           $e[i, j] = \infty$ 
9           $w[i, j] = w[i, j - 1] + p_j + q_j$ 
10         for  $r = i$  to  $j$ 
11              $t = e[i, r - 1] + e[r + 1, j] + w[i, j]$ 
12             if  $t < e[i, j]$ 
13                  $e[i, j] = t$ 
14                  $root[i, j] = r$ 
15  return  $e$  and  $root$ 
```

From the description above and the similarity to the MATRIX-CHAIN-ORDER procedure in Section 15.2, you should find the operation of this procedure to be fairly straightforward. The **for** loop of lines 2–4 initializes the values of $e[i, i - 1]$ and $w[i, i - 1]$. The **for** loop of lines 5–14 then uses the recurrences (15.14) and (15.15) to compute $e[i, j]$ and $w[i, j]$ for all $1 \leq i \leq j \leq n$. In the first iteration, when $l = 1$, the loop computes $e[i, i]$ and $w[i, i]$ for $i = 1, 2, \dots, n$. The second iteration, with $l = 2$, computes $e[i, i + 1]$ and $w[i, i + 1]$ for $i = 1, 2, \dots, n - 1$, and so forth. The innermost **for** loop, in lines 10–14, tries each candidate index r to determine which key k_r to use as the root of an optimal binary search tree containing keys k_i, \dots, k_j . This **for** loop saves the current value of the index r in $root[i, j]$ whenever it finds a better key to use as the root.

Figure 15.10 shows the tables $e[i, j]$, $w[i, j]$, and $root[i, j]$ computed by the procedure OPTIMAL-BST on the key distribution shown in Figure 15.9. As in the matrix-chain multiplication example of Figure 15.5, the tables are rotated to make

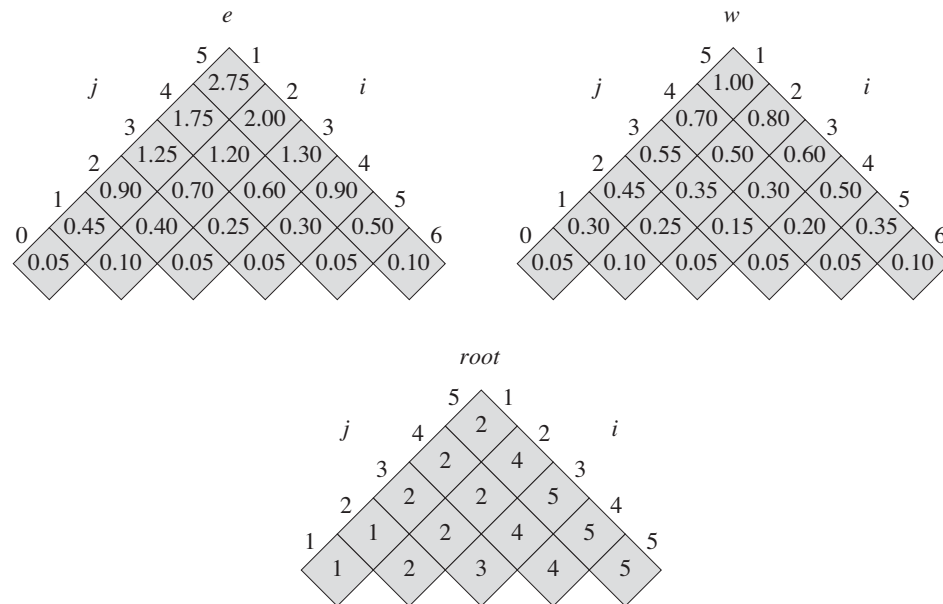


Figure 15.10 The tables $e[i, j]$, $w[i, j]$, and $root[i, j]$ computed by OPTIMAL-BST on the key distribution shown in Figure 15.9. The tables are rotated so that the diagonals run horizontally.

the diagonals run horizontally. OPTIMAL-BST computes the rows from bottom to top and from left to right within each row.

The OPTIMAL-BST procedure takes $\Theta(n^3)$ time, just like MATRIX-CHAIN-ORDER. We can easily see that its running time is $O(n^3)$, since its **for** loops are nested three deep and each loop index takes on at most n values. The loop indices in OPTIMAL-BST do not have exactly the same bounds as those in MATRIX-CHAIN-ORDER, but they are within at most 1 in all directions. Thus, like MATRIX-CHAIN-ORDER, the OPTIMAL-BST procedure takes $\Omega(n^3)$ time.

Exercises

15.5-1

Write pseudocode for the procedure CONSTRUCT-OPTIMAL-BST($root$) which, given the table $root$, outputs the structure of an optimal binary search tree. For the example in Figure 15.10, your procedure should print out the structure

k_2 is the root
 k_1 is the left child of k_2
 d_0 is the left child of k_1
 d_1 is the right child of k_1
 k_5 is the right child of k_2
 k_4 is the left child of k_5
 k_3 is the left child of k_4
 d_2 is the left child of k_3
 d_3 is the right child of k_3
 d_4 is the right child of k_4
 d_5 is the right child of k_5

corresponding to the optimal binary search tree shown in Figure 15.9(b).

15.5-2

Determine the cost and structure of an optimal binary search tree for a set of $n = 7$ keys with the following probabilities:

i	0	1	2	3	4	5	6	7
p_i		0.04	0.06	0.08	0.02	0.10	0.12	0.14
q_i	0.06	0.06	0.06	0.06	0.05	0.05	0.05	0.05

15.5-3

Suppose that instead of maintaining the table $w[i, j]$, we computed the value of $w(i, j)$ directly from equation (15.12) in line 9 of OPTIMAL-BST and used this computed value in line 11. How would this change affect the asymptotic running time of OPTIMAL-BST?

15.5-4 ★

Knuth [212] has shown that there are always roots of optimal subtrees such that $root[i, j - 1] \leq root[i, j] \leq root[i + 1, j]$ for all $1 \leq i < j \leq n$. Use this fact to modify the OPTIMAL-BST procedure to run in $\Theta(n^2)$ time.

Problems

15-1 Longest simple path in a directed acyclic graph

Suppose that we are given a directed acyclic graph $G = (V, E)$ with real-valued edge weights and two distinguished vertices s and t . Describe a dynamic-programming approach for finding a longest weighted simple path from s to t . What does the subproblem graph look like? What is the efficiency of your algorithm?

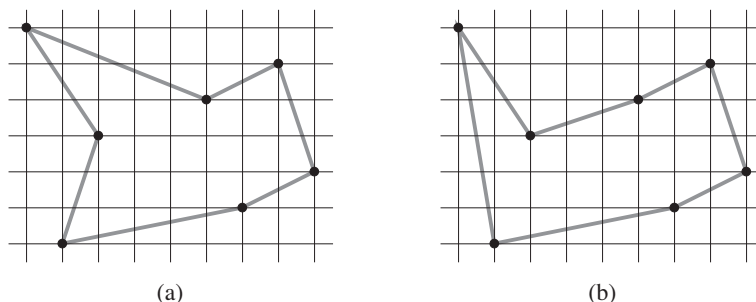


Figure 15.11 Seven points in the plane, shown on a unit grid. **(a)** The shortest closed tour, with length approximately 24.89. This tour is not bitonic. **(b)** The shortest bitonic tour for the same set of points. Its length is approximately 25.58.

15-2 Longest palindrome subsequence

A *palindrome* is a nonempty string over some alphabet that reads the same forward and backward. Examples of palindromes are all strings of length 1, `civic`, `racecar`, and `aibohphobia` (fear of palindromes).

Give an efficient algorithm to find the longest palindrome that is a subsequence of a given input string. For example, given the input `character`, your algorithm should return `carac`. What is the running time of your algorithm?

15-3 Bitonic euclidean traveling-salesman problem

In the *euclidean traveling-salesman problem*, we are given a set of n points in the plane, and we wish to find the shortest closed tour that connects all n points. Figure 15.11(a) shows the solution to a 7-point problem. The general problem is NP-hard, and its solution is therefore believed to require more than polynomial time (see Chapter 34).

J. L. Bentley has suggested that we simplify the problem by restricting our attention to *bitonic tours*, that is, tours that start at the leftmost point, go strictly rightward to the rightmost point, and then go strictly leftward back to the starting point. Figure 15.11(b) shows the shortest bitonic tour of the same 7 points. In this case, a polynomial-time algorithm is possible.

Describe an $O(n^2)$ -time algorithm for determining an optimal bitonic tour. You may assume that no two points have the same x -coordinate and that all operations on real numbers take unit time. (*Hint*: Scan left to right, maintaining optimal possibilities for the two parts of the tour.)

15-4 Printing neatly

Consider the problem of neatly printing a paragraph with a monospaced font (all characters having the same width) on a printer. The input text is a sequence of n

words of lengths l_1, l_2, \dots, l_n , measured in characters. We want to print this paragraph neatly on a number of lines that hold a maximum of M characters each. Our criterion of “neatness” is as follows. If a given line contains words i through j , where $i \leq j$, and we leave exactly one space between words, the number of extra space characters at the end of the line is $M - j + i - \sum_{k=i}^j l_k$, which must be nonnegative so that the words fit on the line. We wish to minimize the sum, over all lines except the last, of the cubes of the numbers of extra space characters at the ends of lines. Give a dynamic-programming algorithm to print a paragraph of n words neatly on a printer. Analyze the running time and space requirements of your algorithm.

15-5 Edit distance

In order to transform one source string of text $x[1..m]$ to a target string $y[1..n]$, we can perform various transformation operations. Our goal is, given x and y , to produce a series of transformations that change x to y . We use an array z —assumed to be large enough to hold all the characters it will need—to hold the intermediate results. Initially, z is empty, and at termination, we should have $z[j] = y[j]$ for $j = 1, 2, \dots, n$. We maintain current indices i into x and j into z , and the operations are allowed to alter z and these indices. Initially, $i = j = 1$. We are required to examine every character in x during the transformation, which means that at the end of the sequence of transformation operations, we must have $i = m + 1$.

We may choose from among six transformation operations:

Copy a character from x to z by setting $z[j] = x[i]$ and then incrementing both i and j . This operation examines $x[i]$.

Replace a character from x by another character c , by setting $z[j] = c$, and then incrementing both i and j . This operation examines $x[i]$.

Delete a character from x by incrementing i but leaving j alone. This operation examines $x[i]$.

Insert the character c into z by setting $z[j] = c$ and then incrementing j , but leaving i alone. This operation examines no characters of x .

Twiddle (i.e., exchange) the next two characters by copying them from x to z but in the opposite order; we do so by setting $z[j] = x[i + 1]$ and $z[j + 1] = x[i]$ and then setting $i = i + 2$ and $j = j + 2$. This operation examines $x[i]$ and $x[i + 1]$.

Kill the remainder of x by setting $i = m + 1$. This operation examines all characters in x that have not yet been examined. This operation, if performed, must be the final operation.

As an example, one way to transform the source string `algorithm` to the target string `altruistic` is to use the following sequence of operations, where the underlined characters are $x[i]$ and $z[j]$ after the operation:

Operation	x	z
<i>initial strings</i>	<u>a</u> lgorithm	—
copy	a <u>l</u> gorithm	a_
copy	al <u>g</u> orithm	al_
replace by t	alg <u>o</u> rithm	alt_
delete	algor <u>i</u> thm	alt_
copy	algor <u>i</u> thm	altr_
insert u	algori <u>h</u> m	altru_
insert i	algori <u>t</u> h	altrui_
insert s	algorit <u>h</u> m	altruish_
twiddle	algorith <u>m</u>	altruist <u>i</u> _
insert c	algorith <u>m</u>	altruistic_
kill	algorithm_	altruistic_

Note that there are several other sequences of transformation operations that transform `algorithm` to `altruistic`.

Each of the transformation operations has an associated cost. The cost of an operation depends on the specific application, but we assume that each operation's cost is a constant that is known to us. We also assume that the individual costs of the copy and replace operations are less than the combined costs of the delete and insert operations; otherwise, the copy and replace operations would not be used. The cost of a given sequence of transformation operations is the sum of the costs of the individual operations in the sequence. For the sequence above, the cost of transforming `algorithm` to `altruistic` is

$$(3 \cdot \text{cost}(\text{copy})) + \text{cost}(\text{replace}) + \text{cost}(\text{delete}) + (4 \cdot \text{cost}(\text{insert})) \\ + \text{cost}(\text{twiddle}) + \text{cost}(\text{kill}) .$$

- a. Given two sequences $x[1..m]$ and $y[1..n]$ and set of transformation-operation costs, the **edit distance** from x to y is the cost of the least expensive operation sequence that transforms x to y . Describe a dynamic-programming algorithm that finds the edit distance from $x[1..m]$ to $y[1..n]$ and prints an optimal operation sequence. Analyze the running time and space requirements of your algorithm.

The edit-distance problem generalizes the problem of aligning two DNA sequences (see, for example, Setubal and Meidanis [310, Section 3.2]). There are several methods for measuring the similarity of two DNA sequences by aligning them. One such method to align two sequences x and y consists of inserting spaces at

arbitrary locations in the two sequences (including at either end) so that the resulting sequences x' and y' have the same length but do not have a space in the same position (i.e., for no position j are both $x'[j]$ and $y'[j]$ a space). Then we assign a “score” to each position. Position j receives a score as follows:

- $+1$ if $x'[j] = y'[j]$ and neither is a space,
- -1 if $x'[j] \neq y'[j]$ and neither is a space,
- -2 if either $x'[j]$ or $y'[j]$ is a space.

The score for the alignment is the sum of the scores of the individual positions. For example, given the sequences $x = \text{GATCGGCAT}$ and $y = \text{CAATGTGAATC}$, one alignment is

```
G  ATCG  GCAT
CAAT GTGAATC
-*****-+++
```

A $+$ under a position indicates a score of $+1$ for that position, a $-$ indicates a score of -1 , and a $*$ indicates a score of -2 , so that this alignment has a total score of $6 \cdot 1 - 2 \cdot 1 - 4 \cdot 2 = -4$.

- b.* Explain how to cast the problem of finding an optimal alignment as an edit distance problem using a subset of the transformation operations copy, replace, delete, insert, twiddle, and kill.

15-6 Planning a company party

Professor Stewart is consulting for the president of a corporation that is planning a company party. The company has a hierarchical structure; that is, the supervisor relation forms a tree rooted at the president. The personnel office has ranked each employee with a conviviality rating, which is a real number. In order to make the party fun for all attendees, the president does not want both an employee and his or her immediate supervisor to attend.

Professor Stewart is given the tree that describes the structure of the corporation, using the left-child, right-sibling representation described in Section 10.4. Each node of the tree holds, in addition to the pointers, the name of an employee and that employee’s conviviality ranking. Describe an algorithm to make up a guest list that maximizes the sum of the conviviality ratings of the guests. Analyze the running time of your algorithm.

15-7 Viterbi algorithm

We can use dynamic programming on a directed graph $G = (V, E)$ for speech recognition. Each edge $(u, v) \in E$ is labeled with a sound $\sigma(u, v)$ from a finite set Σ of sounds. The labeled graph is a formal model of a person speaking

a restricted language. Each path in the graph starting from a distinguished vertex $v_0 \in V$ corresponds to a possible sequence of sounds produced by the model. We define the label of a directed path to be the concatenation of the labels of the edges on that path.

- a. Describe an efficient algorithm that, given an edge-labeled graph G with distinguished vertex v_0 and a sequence $s = \langle \sigma_1, \sigma_2, \dots, \sigma_k \rangle$ of sounds from Σ , returns a path in G that begins at v_0 and has s as its label, if any such path exists. Otherwise, the algorithm should return NO-SUCH-PATH. Analyze the running time of your algorithm. (*Hint:* You may find concepts from Chapter 22 useful.)

Now, suppose that every edge $(u, v) \in E$ has an associated nonnegative probability $p(u, v)$ of traversing the edge (u, v) from vertex u and thus producing the corresponding sound. The sum of the probabilities of the edges leaving any vertex equals 1. The probability of a path is defined to be the product of the probabilities of its edges. We can view the probability of a path beginning at v_0 as the probability that a “random walk” beginning at v_0 will follow the specified path, where we randomly choose which edge to take leaving a vertex u according to the probabilities of the available edges leaving u .

- b. Extend your answer to part (a) so that if a path is returned, it is a *most probable path* starting at v_0 and having label s . Analyze the running time of your algorithm.

15-8 Image compression by seam carving

We are given a color picture consisting of an $m \times n$ array $A[1..m, 1..n]$ of pixels, where each pixel specifies a triple of red, green, and blue (RGB) intensities. Suppose that we wish to compress this picture slightly. Specifically, we wish to remove one pixel from each of the m rows, so that the whole picture becomes one pixel narrower. To avoid disturbing visual effects, however, we require that the pixels removed in two adjacent rows be in the same or adjacent columns; the pixels removed form a “seam” from the top row to the bottom row where successive pixels in the seam are adjacent vertically or diagonally.

- a. Show that the number of such possible seams grows at least exponentially in m , assuming that $n > 1$.
- b. Suppose now that along with each pixel $A[i, j]$, we have calculated a real-valued disruption measure $d[i, j]$, indicating how disruptive it would be to remove pixel $A[i, j]$. Intuitively, the lower a pixel’s disruption measure, the more similar the pixel is to its neighbors. Suppose further that we define the disruption measure of a seam to be the sum of the disruption measures of its pixels.

Give an algorithm to find a seam with the lowest disruption measure. How efficient is your algorithm?

15-9 *Breaking a string*

A certain string-processing language allows a programmer to break a string into two pieces. Because this operation copies the string, it costs n time units to break a string of n characters into two pieces. Suppose a programmer wants to break a string into many pieces. The order in which the breaks occur can affect the total amount of time used. For example, suppose that the programmer wants to break a 20-character string after characters 2, 8, and 10 (numbering the characters in ascending order from the left-hand end, starting from 1). If she programs the breaks to occur in left-to-right order, then the first break costs 20 time units, the second break costs 18 time units (breaking the string from characters 3 to 20 at character 8), and the third break costs 12 time units, totaling 50 time units. If she programs the breaks to occur in right-to-left order, however, then the first break costs 20 time units, the second break costs 10 time units, and the third break costs 8 time units, totaling 38 time units. In yet another order, she could break first at 8 (costing 20), then break the left piece at 2 (costing 8), and finally the right piece at 10 (costing 12), for a total cost of 40.

Design an algorithm that, given the numbers of characters after which to break, determines a least-cost way to sequence those breaks. More formally, given a string S with n characters and an array $L[1..m]$ containing the break points, compute the lowest cost for a sequence of breaks, along with a sequence of breaks that achieves this cost.

15-10 *Planning an investment strategy*

Your knowledge of algorithms helps you obtain an exciting job with the Acme Computer Company, along with a \$10,000 signing bonus. You decide to invest this money with the goal of maximizing your return at the end of 10 years. You decide to use the Amalgamated Investment Company to manage your investments. Amalgamated Investments requires you to observe the following rules. It offers n different investments, numbered 1 through n . In each year j , investment i provides a return rate of r_{ij} . In other words, if you invest d dollars in investment i in year j , then at the end of year j , you have dr_{ij} dollars. The return rates are guaranteed, that is, you are given all the return rates for the next 10 years for each investment. You make investment decisions only once per year. At the end of each year, you can leave the money made in the previous year in the same investments, or you can shift money to other investments, by either shifting money between existing investments or moving money to a new investment. If you do not move your money between two consecutive years, you pay a fee of f_1 dollars, whereas if you switch your money, you pay a fee of f_2 dollars, where $f_2 > f_1$.

- a.* The problem, as stated, allows you to invest your money in multiple investments in each year. Prove that there exists an optimal investment strategy that, in each year, puts all the money into a single investment. (Recall that an optimal investment strategy maximizes the amount of money after 10 years and is not concerned with any other objectives, such as minimizing risk.)
- b.* Prove that the problem of planning your optimal investment strategy exhibits optimal substructure.
- c.* Design an algorithm that plans your optimal investment strategy. What is the running time of your algorithm?
- d.* Suppose that Amalgamated Investments imposed the additional restriction that, at any point, you can have no more than \$15,000 in any one investment. Show that the problem of maximizing your income at the end of 10 years no longer exhibits optimal substructure.

15-11 *Inventory planning*

The Rinky Dink Company makes machines that resurface ice rinks. The demand for such products varies from month to month, and so the company needs to develop a strategy to plan its manufacturing given the fluctuating, but predictable, demand. The company wishes to design a plan for the next n months. For each month i , the company knows the demand d_i , that is, the number of machines that it will sell. Let $D = \sum_{i=1}^n d_i$ be the total demand over the next n months. The company keeps a full-time staff who provide labor to manufacture up to m machines per month. If the company needs to make more than m machines in a given month, it can hire additional, part-time labor, at a cost that works out to c dollars per machine. Furthermore, if, at the end of a month, the company is holding any unsold machines, it must pay inventory costs. The cost for holding j machines is given as a function $h(j)$ for $j = 1, 2, \dots, D$, where $h(j) \geq 0$ for $1 \leq j \leq D$ and $h(j) \leq h(j + 1)$ for $1 \leq j \leq D - 1$.

Give an algorithm that calculates a plan for the company that minimizes its costs while fulfilling all the demand. The running time should be polyomial in n and D .

15-12 *Signing free-agent baseball players*

Suppose that you are the general manager for a major-league baseball team. During the off-season, you need to sign some free-agent players for your team. The team owner has given you a budget of $\$X$ to spend on free agents. You are allowed to spend less than $\$X$ altogether, but the owner will fire you if you spend any more than $\$X$.

You are considering N different positions, and for each position, P free-agent players who play that position are available.⁸ Because you do not want to overload your roster with too many players at any position, for each position you may sign at most one free agent who plays that position. (If you do not sign any players at a particular position, then you plan to stick with the players you already have at that position.)

To determine how valuable a player is going to be, you decide to use a sabermetric statistic⁹ known as “VORP,” or “value over replacement player.” A player with a higher VORP is more valuable than a player with a lower VORP. A player with a higher VORP is not necessarily more expensive to sign than a player with a lower VORP, because factors other than a player’s value determine how much it costs to sign him.

For each available free-agent player, you have three pieces of information:

- the player’s position,
- the amount of money it will cost to sign the player, and
- the player’s VORP.

Devise an algorithm that maximizes the total VORP of the players you sign while spending no more than $\$X$ altogether. You may assume that each player signs for a multiple of \$100,000. Your algorithm should output the total VORP of the players you sign, the total amount of money you spend, and a list of which players you sign. Analyze the running time and space requirement of your algorithm.

Chapter notes

R. Bellman began the systematic study of dynamic programming in 1955. The word “programming,” both here and in linear programming, refers to using a tabular solution method. Although optimization techniques incorporating elements of dynamic programming were known earlier, Bellman provided the area with a solid mathematical basis [37].

⁸Although there are nine positions on a baseball team, N is not necessarily equal to 9 because some general managers have particular ways of thinking about positions. For example, a general manager might consider right-handed pitchers and left-handed pitchers to be separate “positions,” as well as starting pitchers, long relief pitchers (relief pitchers who can pitch several innings), and short relief pitchers (relief pitchers who normally pitch at most only one inning).

⁹*Sabermetrics* is the application of statistical analysis to baseball records. It provides several ways to compare the relative values of individual players.

Galil and Park [125] classify dynamic-programming algorithms according to the size of the table and the number of other table entries each entry depends on. They call a dynamic-programming algorithm tD/eD if its table size is $O(n^t)$ and each entry depends on $O(n^e)$ other entries. For example, the matrix-chain multiplication algorithm in Section 15.2 would be $2D/1D$, and the longest-common-subsequence algorithm in Section 15.4 would be $2D/0D$.

Hu and Shing [182, 183] give an $O(n \lg n)$ -time algorithm for the matrix-chain multiplication problem.

The $O(mn)$ -time algorithm for the longest-common-subsequence problem appears to be a folk algorithm. Knuth [70] posed the question of whether subquadratic algorithms for the LCS problem exist. Masek and Paterson [244] answered this question in the affirmative by giving an algorithm that runs in $O(mn/\lg n)$ time, where $n \leq m$ and the sequences are drawn from a set of bounded size. For the special case in which no element appears more than once in an input sequence, Szymanski [326] shows how to solve the problem in $O((n+m) \lg(n+m))$ time. Many of these results extend to the problem of computing string edit distances (Problem 15-5).

An early paper on variable-length binary encodings by Gilbert and Moore [133] had applications to constructing optimal binary search trees for the case in which all probabilities p_i are 0; this paper contains an $O(n^3)$ -time algorithm. Aho, Hopcroft, and Ullman [5] present the algorithm from Section 15.5. Exercise 15.5-4 is due to Knuth [212]. Hu and Tucker [184] devised an algorithm for the case in which all probabilities p_i are 0 that uses $O(n^2)$ time and $O(n)$ space; subsequently, Knuth [211] reduced the time to $O(n \lg n)$.

Problem 15-8 is due to Avidan and Shamir [27], who have posted on the Web a wonderful video illustrating this image-compression technique.

16 Greedy Algorithms

Algorithms for optimization problems typically go through a sequence of steps, with a set of choices at each step. For many optimization problems, using dynamic programming to determine the best choices is overkill; simpler, more efficient algorithms will do. A *greedy algorithm* always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution. This chapter explores optimization problems for which greedy algorithms provide optimal solutions. Before reading this chapter, you should read about dynamic programming in Chapter 15, particularly Section 15.3.

Greedy algorithms do not always yield optimal solutions, but for many problems they do. We shall first examine, in Section 16.1, a simple but nontrivial problem, the activity-selection problem, for which a greedy algorithm efficiently computes an optimal solution. We shall arrive at the greedy algorithm by first considering a dynamic-programming approach and then showing that we can always make greedy choices to arrive at an optimal solution. Section 16.2 reviews the basic elements of the greedy approach, giving a direct approach for proving greedy algorithms correct. Section 16.3 presents an important application of greedy techniques: designing data-compression (Huffman) codes. In Section 16.4, we investigate some of the theory underlying combinatorial structures called “matroids,” for which a greedy algorithm always produces an optimal solution. Finally, Section 16.5 applies matroids to solve a problem of scheduling unit-time tasks with deadlines and penalties.

The greedy method is quite powerful and works well for a wide range of problems. Later chapters will present many algorithms that we can view as applications of the greedy method, including minimum-spanning-tree algorithms (Chapter 23), Dijkstra’s algorithm for shortest paths from a single source (Chapter 24), and Chvátal’s greedy set-covering heuristic (Chapter 35). Minimum-spanning-tree algorithms furnish a classic example of the greedy method. Although you can read

this chapter and Chapter 23 independently of each other, you might find it useful to read them together.

16.1 An activity-selection problem

Our first example is the problem of scheduling several competing activities that require exclusive use of a common resource, with a goal of selecting a maximum-size set of mutually compatible activities. Suppose we have a set $S = \{a_1, a_2, \dots, a_n\}$ of n proposed **activities** that wish to use a resource, such as a lecture hall, which can serve only one activity at a time. Each activity a_i has a **start time** s_i and a **finish time** f_i , where $0 \leq s_i < f_i < \infty$. If selected, activity a_i takes place during the half-open time interval $[s_i, f_i)$. Activities a_i and a_j are **compatible** if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap. That is, a_i and a_j are compatible if $s_i \geq f_j$ or $s_j \geq f_i$. In the **activity-selection problem**, we wish to select a maximum-size subset of mutually compatible activities. We assume that the activities are sorted in monotonically increasing order of finish time:

$$f_1 \leq f_2 \leq f_3 \leq \dots \leq f_{n-1} \leq f_n. \quad (16.1)$$

(We shall see later the advantage that this assumption provides.) For example, consider the following set S of activities:

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

For this example, the subset $\{a_3, a_9, a_{11}\}$ consists of mutually compatible activities. It is not a maximum subset, however, since the subset $\{a_1, a_4, a_8, a_{11}\}$ is larger. In fact, $\{a_1, a_4, a_8, a_{11}\}$ is a largest subset of mutually compatible activities; another largest subset is $\{a_2, a_4, a_9, a_{11}\}$.

We shall solve this problem in several steps. We start by thinking about a dynamic-programming solution, in which we consider several choices when determining which subproblems to use in an optimal solution. We shall then observe that we need to consider only one choice—the greedy choice—and that when we make the greedy choice, only one subproblem remains. Based on these observations, we shall develop a recursive greedy algorithm to solve the activity-scheduling problem. We shall complete the process of developing a greedy solution by converting the recursive algorithm to an iterative one. Although the steps we shall go through in this section are slightly more involved than is typical when developing a greedy algorithm, they illustrate the relationship between greedy algorithms and dynamic programming.

The optimal substructure of the activity-selection problem

We can easily verify that the activity-selection problem exhibits optimal substructure. Let us denote by S_{ij} the set of activities that start after activity a_i finishes and that finish before activity a_j starts. Suppose that we wish to find a maximum set of mutually compatible activities in S_{ij} , and suppose further that such a maximum set is A_{ij} , which includes some activity a_k . By including a_k in an optimal solution, we are left with two subproblems: finding mutually compatible activities in the set S_{ik} (activities that start after activity a_i finishes and that finish before activity a_k starts) and finding mutually compatible activities in the set S_{kj} (activities that start after activity a_k finishes and that finish before activity a_j starts). Let $A_{ik} = A_{ij} \cap S_{ik}$ and $A_{kj} = A_{ij} \cap S_{kj}$, so that A_{ik} contains the activities in A_{ij} that finish before a_k starts and A_{kj} contains the activities in A_{ij} that start after a_k finishes. Thus, we have $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$, and so the maximum-size set A_{ij} of mutually compatible activities in S_{ij} consists of $|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$ activities.

The usual cut-and-paste argument shows that the optimal solution A_{ij} must also include optimal solutions to the two subproblems for S_{ik} and S_{kj} . If we could find a set A'_{kj} of mutually compatible activities in S_{kj} where $|A'_{kj}| > |A_{kj}|$, then we could use A'_{kj} , rather than A_{kj} , in a solution to the subproblem for S_{ij} . We would have constructed a set of $|A_{ik}| + |A'_{kj}| + 1 > |A_{ik}| + |A_{kj}| + 1 = |A_{ij}|$ mutually compatible activities, which contradicts the assumption that A_{ij} is an optimal solution. A symmetric argument applies to the activities in S_{ik} .

This way of characterizing optimal substructure suggests that we might solve the activity-selection problem by dynamic programming. If we denote the size of an optimal solution for the set S_{ij} by $c[i, j]$, then we would have the recurrence

$$c[i, j] = c[i, k] + c[k, j] + 1.$$

Of course, if we did not know that an optimal solution for the set S_{ij} includes activity a_k , we would have to examine all activities in S_{ij} to find which one to choose, so that

$$c[i, j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset, \\ \max_{a_k \in S_{ij}} \{c[i, k] + c[k, j] + 1\} & \text{if } S_{ij} \neq \emptyset. \end{cases} \quad (16.2)$$

We could then develop a recursive algorithm and memoize it, or we could work bottom-up and fill in table entries as we go along. But we would be overlooking another important characteristic of the activity-selection problem that we can use to great advantage.

Making the greedy choice

What if we could choose an activity to add to our optimal solution without having to first solve all the subproblems? That could save us from having to consider all the choices inherent in recurrence (16.2). In fact, for the activity-selection problem, we need consider only one choice: the greedy choice.

What do we mean by the greedy choice for the activity-selection problem? Intuition suggests that we should choose an activity that leaves the resource available for as many other activities as possible. Now, of the activities we end up choosing, one of them must be the first one to finish. Our intuition tells us, therefore, to choose the activity in S with the earliest finish time, since that would leave the resource available for as many of the activities that follow it as possible. (If more than one activity in S has the earliest finish time, then we can choose any such activity.) In other words, since the activities are sorted in monotonically increasing order by finish time, the greedy choice is activity a_1 . Choosing the first activity to finish is not the only way to think of making a greedy choice for this problem; Exercise 16.1-3 asks you to explore other possibilities.

If we make the greedy choice, we have only one remaining subproblem to solve: finding activities that start after a_1 finishes. Why don't we have to consider activities that finish before a_1 starts? We have that $s_1 < f_1$, and f_1 is the earliest finish time of any activity, and therefore no activity can have a finish time less than or equal to s_1 . Thus, all activities that are compatible with activity a_1 must start after a_1 finishes.

Furthermore, we have already established that the activity-selection problem exhibits optimal substructure. Let $S_k = \{a_i \in S : s_i \geq f_k\}$ be the set of activities that start after activity a_k finishes. If we make the greedy choice of activity a_1 , then S_1 remains as the only subproblem to solve.¹ Optimal substructure tells us that if a_1 is in the optimal solution, then an optimal solution to the original problem consists of activity a_1 and all the activities in an optimal solution to the subproblem S_1 .

One big question remains: is our intuition correct? Is the greedy choice—in which we choose the first activity to finish—always part of some optimal solution? The following theorem shows that it is.

¹We sometimes refer to the sets S_k as subproblems rather than as just sets of activities. It will always be clear from the context whether we are referring to S_k as a set of activities or as a subproblem whose input is that set.

Theorem 16.1

Consider any nonempty subproblem S_k , and let a_m be an activity in S_k with the earliest finish time. Then a_m is included in some maximum-size subset of mutually compatible activities of S_k .

Proof Let A_k be a maximum-size subset of mutually compatible activities in S_k , and let a_j be the activity in A_k with the earliest finish time. If $a_j = a_m$, we are done, since we have shown that a_m is in some maximum-size subset of mutually compatible activities of S_k . If $a_j \neq a_m$, let the set $A'_k = A_k - \{a_j\} \cup \{a_m\}$ be A_k but substituting a_m for a_j . The activities in A'_k are disjoint, which follows because the activities in A_k are disjoint, a_j is the first activity in A_k to finish, and $f_m \leq f_j$. Since $|A'_k| = |A_k|$, we conclude that A'_k is a maximum-size subset of mutually compatible activities of S_k , and it includes a_m . ■

Thus, we see that although we might be able to solve the activity-selection problem with dynamic programming, we don't need to. (Besides, we have not yet examined whether the activity-selection problem even has overlapping subproblems.) Instead, we can repeatedly choose the activity that finishes first, keep only the activities compatible with this activity, and repeat until no activities remain. Moreover, because we always choose the activity with the earliest finish time, the finish times of the activities we choose must strictly increase. We can consider each activity just once overall, in monotonically increasing order of finish times.

An algorithm to solve the activity-selection problem does not need to work bottom-up, like a table-based dynamic-programming algorithm. Instead, it can work top-down, choosing an activity to put into the optimal solution and then solving the subproblem of choosing activities from those that are compatible with those already chosen. Greedy algorithms typically have this top-down design: make a choice and then solve a subproblem, rather than the bottom-up technique of solving subproblems before making a choice.

A recursive greedy algorithm

Now that we have seen how to bypass the dynamic-programming approach and instead use a top-down, greedy algorithm, we can write a straightforward, recursive procedure to solve the activity-selection problem. The procedure `RECURSIVE-ACTIVITY-SELECTOR` takes the start and finish times of the activities, represented as arrays s and f ,² the index k that defines the subproblem S_k it is to solve, and

²Because the pseudocode takes s and f as arrays, it indexes into them with square brackets rather than subscripts.

the size n of the original problem. It returns a maximum-size set of mutually compatible activities in S_k . We assume that the n input activities are already ordered by monotonically increasing finish time, according to equation (16.1). If not, we can sort them into this order in $O(n \lg n)$ time, breaking ties arbitrarily. In order to start, we add the fictitious activity a_0 with $f_0 = 0$, so that subproblem S_0 is the entire set of activities S . The initial call, which solves the entire problem, is `RECURSIVE-ACTIVITY-SELECTOR($s, f, 0, n$)`.

`RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)`

```

1   $m = k + 1$ 
2  while  $m \leq n$  and  $s[m] < f[k]$       // find the first activity in  $S_k$  to finish
3       $m = m + 1$ 
4  if  $m \leq n$ 
5      return  $\{a_m\} \cup \text{RECURSIVE-ACTIVITY-SELECTOR}(s, f, m, n)$ 
6  else return  $\emptyset$ 
```

Figure 16.1 shows the operation of the algorithm. In a given recursive call `RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)`, the **while** loop of lines 2–3 looks for the first activity in S_k to finish. The loop examines $a_{k+1}, a_{k+2}, \dots, a_n$, until it finds the first activity a_m that is compatible with a_k ; such an activity has $s_m \geq f_k$. If the loop terminates because it finds such an activity, line 5 returns the union of $\{a_m\}$ and the maximum-size subset of S_m returned by the recursive call `RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)`. Alternatively, the loop may terminate because $m > n$, in which case we have examined all activities in S_k without finding one that is compatible with a_k . In this case, $S_k = \emptyset$, and so the procedure returns \emptyset in line 6.

Assuming that the activities have already been sorted by finish times, the running time of the call `RECURSIVE-ACTIVITY-SELECTOR($s, f, 0, n$)` is $\Theta(n)$, which we can see as follows. Over all recursive calls, each activity is examined exactly once in the **while** loop test of line 2. In particular, activity a_i is examined in the last call made in which $k < i$.

An iterative greedy algorithm

We easily can convert our recursive procedure to an iterative one. The procedure `RECURSIVE-ACTIVITY-SELECTOR` is almost “tail recursive” (see Problem 7-4): it ends with a recursive call to itself followed by a union operation. It is usually a straightforward task to transform a tail-recursive procedure to an iterative form; in fact, some compilers for certain programming languages perform this task automatically. As written, `RECURSIVE-ACTIVITY-SELECTOR` works for subproblems S_k , i.e., subproblems that consist of the last activities to finish.

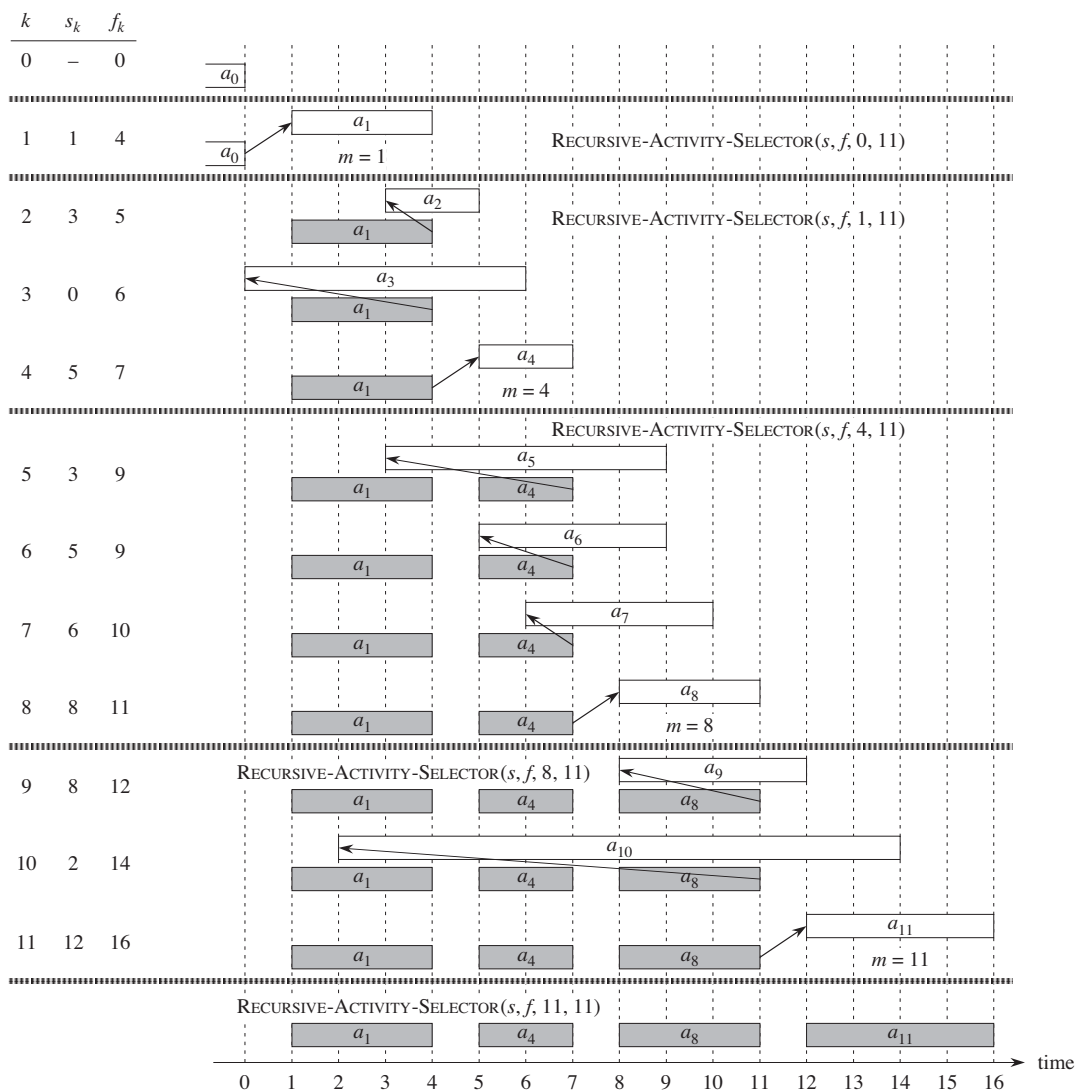


Figure 16.1 The operation of RECURSIVE-ACTIVITY-SELECTOR on the 11 activities given earlier. Activities considered in each recursive call appear between horizontal lines. The fictitious activity a_0 finishes at time 0, and the initial call RECURSIVE-ACTIVITY-SELECTOR($s, f, 0, 11$), selects activity a_1 . In each recursive call, the activities that have already been selected are shaded, and the activity shown in white is being considered. If the starting time of an activity occurs before the finish time of the most recently added activity (the arrow between them points left), it is rejected. Otherwise (the arrow points directly up or to the right), it is selected. The last recursive call, RECURSIVE-ACTIVITY-SELECTOR($s, f, 11, 11$), returns \emptyset . The resulting set of selected activities is $\{a_1, a_4, a_8, a_{11}\}$.

The procedure GREEDY-ACTIVITY-SELECTOR is an iterative version of the procedure RECURSIVE-ACTIVITY-SELECTOR. It also assumes that the input activities are ordered by monotonically increasing finish time. It collects selected activities into a set A and returns this set when it is done.

GREEDY-ACTIVITY-SELECTOR(s, f)

```

1   $n = s.length$ 
2   $A = \{a_1\}$ 
3   $k = 1$ 
4  for  $m = 2$  to  $n$ 
5      if  $s[m] \geq f[k]$ 
6           $A = A \cup \{a_m\}$ 
7           $k = m$ 
8  return  $A$ 
```

The procedure works as follows. The variable k indexes the most recent addition to A , corresponding to the activity a_k in the recursive version. Since we consider the activities in order of monotonically increasing finish time, f_k is always the maximum finish time of any activity in A . That is,

$$f_k = \max \{f_i : a_i \in A\} . \quad (16.3)$$

Lines 2–3 select activity a_1 , initialize A to contain just this activity, and initialize k to index this activity. The **for** loop of lines 4–7 finds the earliest activity in S_k to finish. The loop considers each activity a_m in turn and adds a_m to A if it is compatible with all previously selected activities; such an activity is the earliest in S_k to finish. To see whether activity a_m is compatible with every activity currently in A , it suffices by equation (16.3) to check (in line 5) that its start time s_m is not earlier than the finish time f_k of the activity most recently added to A . If activity a_m is compatible, then lines 6–7 add activity a_m to A and set k to m . The set A returned by the call GREEDY-ACTIVITY-SELECTOR(s, f) is precisely the set returned by the call RECURSIVE-ACTIVITY-SELECTOR($s, f, 0, n$).

Like the recursive version, GREEDY-ACTIVITY-SELECTOR schedules a set of n activities in $\Theta(n)$ time, assuming that the activities were already sorted initially by their finish times.

Exercises

16.1-1

Give a dynamic-programming algorithm for the activity-selection problem, based on recurrence (16.2). Have your algorithm compute the sizes $c[i, j]$ as defined above and also produce the maximum-size subset of mutually compatible activities.

Assume that the inputs have been sorted as in equation (16.1). Compare the running time of your solution to the running time of GREEDY-ACTIVITY-SELECTOR.

16.1-2

Suppose that instead of always selecting the first activity to finish, we instead select the last activity to start that is compatible with all previously selected activities. Describe how this approach is a greedy algorithm, and prove that it yields an optimal solution.

16.1-3

Not just any greedy approach to the activity-selection problem produces a maximum-size set of mutually compatible activities. Give an example to show that the approach of selecting the activity of least duration from among those that are compatible with previously selected activities does not work. Do the same for the approaches of always selecting the compatible activity that overlaps the fewest other remaining activities and always selecting the compatible remaining activity with the earliest start time.

16.1-4

Suppose that we have a set of activities to schedule among a large number of lecture halls, where any activity can take place in any lecture hall. We wish to schedule all the activities using as few lecture halls as possible. Give an efficient greedy algorithm to determine which activity should use which lecture hall.

(This problem is also known as the *interval-graph coloring problem*. We can create an interval graph whose vertices are the given activities and whose edges connect incompatible activities. The smallest number of colors required to color every vertex so that no two adjacent vertices have the same color corresponds to finding the fewest lecture halls needed to schedule all of the given activities.)

16.1-5

Consider a modification to the activity-selection problem in which each activity a_i has, in addition to a start and finish time, a value v_i . The objective is no longer to maximize the number of activities scheduled, but instead to maximize the total value of the activities scheduled. That is, we wish to choose a set A of compatible activities such that $\sum_{a_k \in A} v_k$ is maximized. Give a polynomial-time algorithm for this problem.

16.2 Elements of the greedy strategy

A greedy algorithm obtains an optimal solution to a problem by making a sequence of choices. At each decision point, the algorithm makes choice that seems best at the moment. This heuristic strategy does not always produce an optimal solution, but as we saw in the activity-selection problem, sometimes it does. This section discusses some of the general properties of greedy methods.

The process that we followed in Section 16.1 to develop a greedy algorithm was a bit more involved than is typical. We went through the following steps:

1. Determine the optimal substructure of the problem.
2. Develop a recursive solution. (For the activity-selection problem, we formulated recurrence (16.2), but we bypassed developing a recursive algorithm based on this recurrence.)
3. Show that if we make the greedy choice, then only one subproblem remains.
4. Prove that it is always safe to make the greedy choice. (Steps 3 and 4 can occur in either order.)
5. Develop a recursive algorithm that implements the greedy strategy.
6. Convert the recursive algorithm to an iterative algorithm.

In going through these steps, we saw in great detail the dynamic-programming underpinnings of a greedy algorithm. For example, in the activity-selection problem, we first defined the subproblems S_{ij} , where both i and j varied. We then found that if we always made the greedy choice, we could restrict the subproblems to be of the form S_k .

Alternatively, we could have fashioned our optimal substructure with a greedy choice in mind, so that the choice leaves just one subproblem to solve. In the activity-selection problem, we could have started by dropping the second subscript and defining subproblems of the form S_k . Then, we could have proven that a greedy choice (the first activity a_m to finish in S_k), combined with an optimal solution to the remaining set S_m of compatible activities, yields an optimal solution to S_k . More generally, we design greedy algorithms according to the following sequence of steps:

1. Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
2. Prove that there is always an optimal solution to the original problem that makes the greedy choice, so that the greedy choice is always safe.

3. Demonstrate optimal substructure by showing that, having made the greedy choice, what remains is a subproblem with the property that if we combine an optimal solution to the subproblem with the greedy choice we have made, we arrive at an optimal solution to the original problem.

We shall use this more direct process in later sections of this chapter. Nevertheless, beneath every greedy algorithm, there is almost always a more cumbersome dynamic-programming solution.

How can we tell whether a greedy algorithm will solve a particular optimization problem? No way works all the time, but the greedy-choice property and optimal substructure are the two key ingredients. If we can demonstrate that the problem has these properties, then we are well on the way to developing a greedy algorithm for it.

Greedy-choice property

The first key ingredient is the *greedy-choice property*: we can assemble a globally optimal solution by making locally optimal (greedy) choices. In other words, when we are considering which choice to make, we make the choice that looks best in the current problem, without considering results from subproblems.

Here is where greedy algorithms differ from dynamic programming. In dynamic programming, we make a choice at each step, but the choice usually depends on the solutions to subproblems. Consequently, we typically solve dynamic-programming problems in a bottom-up manner, progressing from smaller subproblems to larger subproblems. (Alternatively, we can solve them top down, but memoizing. Of course, even though the code works top down, we still must solve the subproblems before making a choice.) In a greedy algorithm, we make whatever choice seems best at the moment and then solve the subproblem that remains. The choice made by a greedy algorithm may depend on choices so far, but it cannot depend on any future choices or on the solutions to subproblems. Thus, unlike dynamic programming, which solves the subproblems before making the first choice, a greedy algorithm makes its first choice before solving any subproblems. A dynamic-programming algorithm proceeds bottom up, whereas a greedy strategy usually progresses in a top-down fashion, making one greedy choice after another, reducing each given problem instance to a smaller one.

Of course, we must prove that a greedy choice at each step yields a globally optimal solution. Typically, as in the case of Theorem 16.1, the proof examines a globally optimal solution to some subproblem. It then shows how to modify the solution to substitute the greedy choice for some other choice, resulting in one similar, but smaller, subproblem.

We can usually make the greedy choice more efficiently than when we have to consider a wider set of choices. For example, in the activity-selection problem, as-

suming that we had already sorted the activities in monotonically increasing order of finish times, we needed to examine each activity just once. By preprocessing the input or by using an appropriate data structure (often a priority queue), we often can make greedy choices quickly, thus yielding an efficient algorithm.

Optimal substructure

A problem exhibits *optimal substructure* if an optimal solution to the problem contains within it optimal solutions to subproblems. This property is a key ingredient of assessing the applicability of dynamic programming as well as greedy algorithms. As an example of optimal substructure, recall how we demonstrated in Section 16.1 that if an optimal solution to subproblem S_{ij} includes an activity a_k , then it must also contain optimal solutions to the subproblems S_{ik} and S_{kj} . Given this optimal substructure, we argued that if we knew which activity to use as a_k , we could construct an optimal solution to S_{ij} by selecting a_k along with all activities in optimal solutions to the subproblems S_{ik} and S_{kj} . Based on this observation of optimal substructure, we were able to devise the recurrence (16.2) that described the value of an optimal solution.

We usually use a more direct approach regarding optimal substructure when applying it to greedy algorithms. As mentioned above, we have the luxury of assuming that we arrived at a subproblem by having made the greedy choice in the original problem. All we really need to do is argue that an optimal solution to the subproblem, combined with the greedy choice already made, yields an optimal solution to the original problem. This scheme implicitly uses induction on the subproblems to prove that making the greedy choice at every step produces an optimal solution.

Greedy versus dynamic programming

Because both the greedy and dynamic-programming strategies exploit optimal substructure, you might be tempted to generate a dynamic-programming solution to a problem when a greedy solution suffices or, conversely, you might mistakenly think that a greedy solution works when in fact a dynamic-programming solution is required. To illustrate the subtleties between the two techniques, let us investigate two variants of a classical optimization problem.

The **0-1 knapsack problem** is the following. A thief robbing a store finds n items. The i th item is worth v_i dollars and weighs w_i pounds, where v_i and w_i are integers. The thief wants to take as valuable a load as possible, but he can carry at most W pounds in his knapsack, for some integer W . Which items should he take? (We call this the 0-1 knapsack problem because for each item, the thief must either

take it or leave it behind; he cannot take a fractional amount of an item or take an item more than once.)

In the *fractional knapsack problem*, the setup is the same, but the thief can take fractions of items, rather than having to make a binary (0-1) choice for each item. You can think of an item in the 0-1 knapsack problem as being like a gold ingot and an item in the fractional knapsack problem as more like gold dust.

Both knapsack problems exhibit the optimal-substructure property. For the 0-1 problem, consider the most valuable load that weighs at most W pounds. If we remove item j from this load, the remaining load must be the most valuable load weighing at most $W - w_j$ that the thief can take from the $n - 1$ original items excluding j . For the comparable fractional problem, consider that if we remove a weight w of one item j from the optimal load, the remaining load must be the most valuable load weighing at most $W - w$ that the thief can take from the $n - 1$ original items plus $w_j - w$ pounds of item j .

Although the problems are similar, we can solve the fractional knapsack problem by a greedy strategy, but we cannot solve the 0-1 problem by such a strategy. To solve the fractional problem, we first compute the value per pound v_i/w_i for each item. Obeying a greedy strategy, the thief begins by taking as much as possible of the item with the greatest value per pound. If the supply of that item is exhausted and he can still carry more, he takes as much as possible of the item with the next greatest value per pound, and so forth, until he reaches his weight limit W . Thus, by sorting the items by value per pound, the greedy algorithm runs in $O(n \lg n)$ time. We leave the proof that the fractional knapsack problem has the greedy-choice property as Exercise 16.2-1.

To see that this greedy strategy does not work for the 0-1 knapsack problem, consider the problem instance illustrated in Figure 16.2(a). This example has 3 items and a knapsack that can hold 50 pounds. Item 1 weighs 10 pounds and is worth 60 dollars. Item 2 weighs 20 pounds and is worth 100 dollars. Item 3 weighs 30 pounds and is worth 120 dollars. Thus, the value per pound of item 1 is 6 dollars per pound, which is greater than the value per pound of either item 2 (5 dollars per pound) or item 3 (4 dollars per pound). The greedy strategy, therefore, would take item 1 first. As you can see from the case analysis in Figure 16.2(b), however, the optimal solution takes items 2 and 3, leaving item 1 behind. The two possible solutions that take item 1 are both suboptimal.

For the comparable fractional problem, however, the greedy strategy, which takes item 1 first, does yield an optimal solution, as shown in Figure 16.2(c). Taking item 1 doesn't work in the 0-1 problem because the thief is unable to fill his knapsack to capacity, and the empty space lowers the effective value per pound of his load. In the 0-1 problem, when we consider whether to include an item in the knapsack, we must compare the solution to the subproblem that includes the item with the solution to the subproblem that excludes the item before we can make the

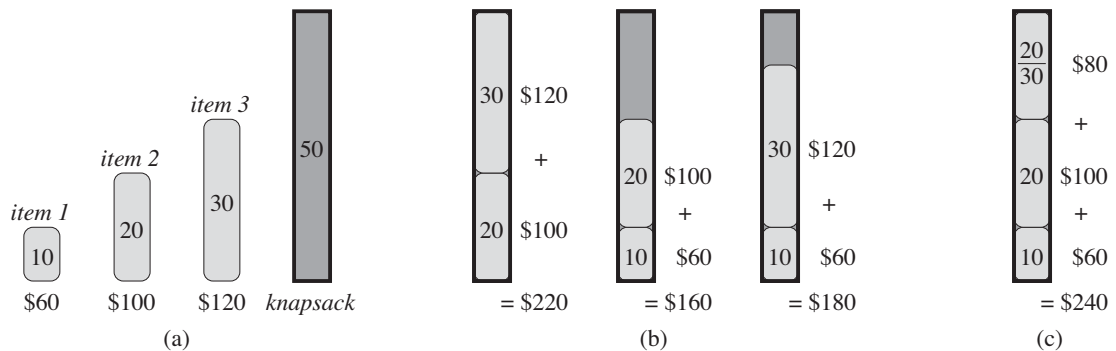


Figure 16.2 An example showing that the greedy strategy does not work for the 0-1 knapsack problem. (a) The thief must select a subset of the three items shown whose weight must not exceed 50 pounds. (b) The optimal subset includes items 2 and 3. Any solution with item 1 is suboptimal, even though item 1 has the greatest value per pound. (c) For the fractional knapsack problem, taking the items in order of greatest value per pound yields an optimal solution.

choice. The problem formulated in this way gives rise to many overlapping sub-problems—a hallmark of dynamic programming, and indeed, as Exercise 16.2-2 asks you to show, we can use dynamic programming to solve the 0-1 problem.

Exercises

16.2-1

Prove that the fractional knapsack problem has the greedy-choice property.

16.2-2

Give a dynamic-programming solution to the 0-1 knapsack problem that runs in $O(nW)$ time, where n is the number of items and W is the maximum weight of items that the thief can put in his knapsack.

16.2-3

Suppose that in a 0-1 knapsack problem, the order of the items when sorted by increasing weight is the same as their order when sorted by decreasing value. Give an efficient algorithm to find an optimal solution to this variant of the knapsack problem, and argue that your algorithm is correct.

16.2-4

Professor Gekko has always dreamed of inline skating across North Dakota. He plans to cross the state on highway U.S. 2, which runs from Grand Forks, on the eastern border with Minnesota, to Williston, near the western border with Montana.

The professor can carry two liters of water, and he can skate m miles before running out of water. (Because North Dakota is relatively flat, the professor does not have to worry about drinking water at a greater rate on uphill sections than on flat or downhill sections.) The professor will start in Grand Forks with two full liters of water. His official North Dakota state map shows all the places along U.S. 2 at which he can refill his water and the distances between these locations.

The professor's goal is to minimize the number of water stops along his route across the state. Give an efficient method by which he can determine which water stops he should make. Prove that your strategy yields an optimal solution, and give its running time.

16.2-5

Describe an efficient algorithm that, given a set $\{x_1, x_2, \dots, x_n\}$ of points on the real line, determines the smallest set of unit-length closed intervals that contains all of the given points. Argue that your algorithm is correct.

16.2-6 ★

Show how to solve the fractional knapsack problem in $O(n)$ time.

16.2-7

Suppose you are given two sets A and B , each containing n positive integers. You can choose to reorder each set however you like. After reordering, let a_i be the i th element of set A , and let b_i be the i th element of set B . You then receive a payoff of $\prod_{i=1}^n a_i^{b_i}$. Give an algorithm that will maximize your payoff. Prove that your algorithm maximizes the payoff, and state its running time.

16.3 Huffman codes

Huffman codes compress data very effectively: savings of 20% to 90% are typical, depending on the characteristics of the data being compressed. We consider the data to be a sequence of characters. Huffman's greedy algorithm uses a table giving how often each character occurs (i.e., its frequency) to build up an optimal way of representing each character as a binary string.

Suppose we have a 100,000-character data file that we wish to store compactly. We observe that the characters in the file occur with the frequencies given by Figure 16.3. That is, only 6 different characters appear, and the character **a** occurs 45,000 times.

We have many options for how to represent such a file of information. Here, we consider the problem of designing a *binary character code* (or *code* for short)

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Figure 16.3 A character-coding problem. A data file of 100,000 characters contains only the characters a–f, with the frequencies indicated. If we assign each character a 3-bit codeword, we can encode the file in 300,000 bits. Using the variable-length code shown, we can encode the file in only 224,000 bits.

in which each character is represented by a unique binary string, which we call a **codeword**. If we use a **fixed-length code**, we need 3 bits to represent 6 characters: a = 000, b = 001, ..., f = 101. This method requires 300,000 bits to code the entire file. Can we do better?

A **variable-length code** can do considerably better than a fixed-length code, by giving frequent characters short codewords and infrequent characters long codewords. Figure 16.3 shows such a code; here the 1-bit string 0 represents a, and the 4-bit string 1100 represents f. This code requires

$$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1,000 = 224,000 \text{ bits}$$

to represent the file, a savings of approximately 25%. In fact, this is an optimal character code for this file, as we shall see.

Prefix codes

We consider here only codes in which no codeword is also a prefix of some other codeword. Such codes are called **prefix codes**.³ Although we won't prove it here, a prefix code can always achieve the optimal data compression among any character code, and so we suffer no loss of generality by restricting our attention to prefix codes.

Encoding is always simple for any binary character code; we just concatenate the codewords representing each character of the file. For example, with the variable-length prefix code of Figure 16.3, we code the 3-character file abc as $0 \cdot 101 \cdot 100 = 0101100$, where “ \cdot ” denotes concatenation.

Prefix codes are desirable because they simplify decoding. Since no codeword is a prefix of any other, the codeword that begins an encoded file is unambiguous. We can simply identify the initial codeword, translate it back to the original char-

³Perhaps “prefix-free codes” would be a better name, but the term “prefix codes” is standard in the literature.

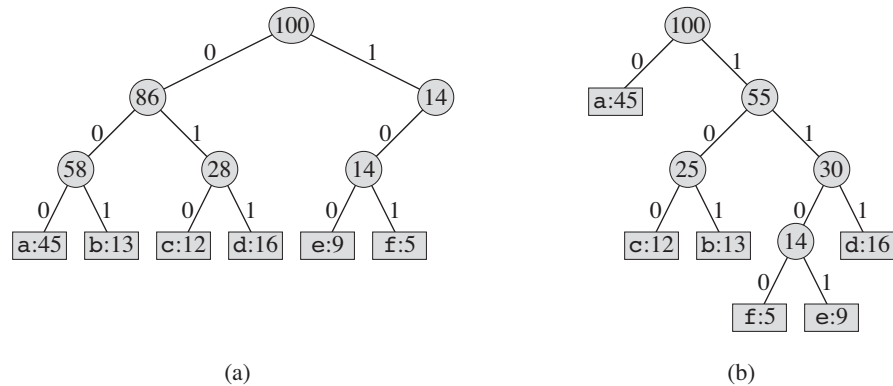


Figure 16.4 Trees corresponding to the coding schemes in Figure 16.3. Each leaf is labeled with a character and its frequency of occurrence. Each internal node is labeled with the sum of the frequencies of the leaves in its subtree. (a) The tree corresponding to the fixed-length code $a = 000, \dots, f = 101$. (b) The tree corresponding to the optimal prefix code $a = 0, b = 101, \dots, f = 1100$.

acter, and repeat the decoding process on the remainder of the encoded file. In our example, the string 001011101 parses uniquely as $0 \cdot 0 \cdot 101 \cdot 1101$, which decodes to **aabe**.

The decoding process needs a convenient representation for the prefix code so that we can easily pick off the initial codeword. A binary tree whose leaves are the given characters provides one such representation. We interpret the binary codeword for a character as the simple path from the root to that character, where 0 means “go to the left child” and 1 means “go to the right child.” Figure 16.4 shows the trees for the two codes of our example. Note that these are not binary search trees, since the leaves need not appear in sorted order and internal nodes do not contain character keys.

An optimal code for a file is always represented by a *full* binary tree, in which every nonleaf node has two children (see Exercise 16.3-2). The fixed-length code in our example is not optimal since its tree, shown in Figure 16.4(a), is not a full binary tree: it contains codewords beginning 10..., but none beginning 11.... Since we can now restrict our attention to full binary trees, we can say that if C is the alphabet from which the characters are drawn and all character frequencies are positive, then the tree for an optimal prefix code has exactly $|C|$ leaves, one for each letter of the alphabet, and exactly $|C| - 1$ internal nodes (see Exercise B.5-3).

Given a tree T corresponding to a prefix code, we can easily compute the number of bits required to encode a file. For each character c in the alphabet C , let the attribute $c.freq$ denote the frequency of c in the file and let $d_T(c)$ denote the depth

of c 's leaf in the tree. Note that $d_T(c)$ is also the length of the codeword for character c . The number of bits required to encode a file is thus

$$B(T) = \sum_{c \in C} c.freq \cdot d_T(c) , \quad (16.4)$$

which we define as the *cost* of the tree T .

Constructing a Huffman code

Huffman invented a greedy algorithm that constructs an optimal prefix code called a **Huffman code**. In line with our observations in Section 16.2, its proof of correctness relies on the greedy-choice property and optimal substructure. Rather than demonstrating that these properties hold and then developing pseudocode, we present the pseudocode first. Doing so will help clarify how the algorithm makes greedy choices.

In the pseudocode that follows, we assume that C is a set of n characters and that each character $c \in C$ is an object with an attribute $c.freq$ giving its frequency. The algorithm builds the tree T corresponding to the optimal code in a bottom-up manner. It begins with a set of $|C|$ leaves and performs a sequence of $|C| - 1$ “merging” operations to create the final tree. The algorithm uses a min-priority queue Q , keyed on the *freq* attribute, to identify the two least-frequent objects to merge together. When we merge two objects, the result is a new object whose frequency is the sum of the frequencies of the two objects that were merged.

HUFFMAN(C)

```

1   $n = |C|$ 
2   $Q = C$ 
3  for  $i = 1$  to  $n - 1$ 
4      allocate a new node  $z$ 
5       $z.left = x = \text{EXTRACT-MIN}(Q)$ 
6       $z.right = y = \text{EXTRACT-MIN}(Q)$ 
7       $z.freq = x.freq + y.freq$ 
8       $\text{INSERT}(Q, z)$ 
9  return  $\text{EXTRACT-MIN}(Q)$     // return the root of the tree
```

For our example, Huffman’s algorithm proceeds as shown in Figure 16.5. Since the alphabet contains 6 letters, the initial queue size is $n = 6$, and 5 merge steps build the tree. The final tree represents the optimal prefix code. The codeword for a letter is the sequence of edge labels on the simple path from the root to the letter.

Line 2 initializes the min-priority queue Q with the characters in C . The **for** loop in lines 3–8 repeatedly extracts the two nodes x and y of lowest frequency

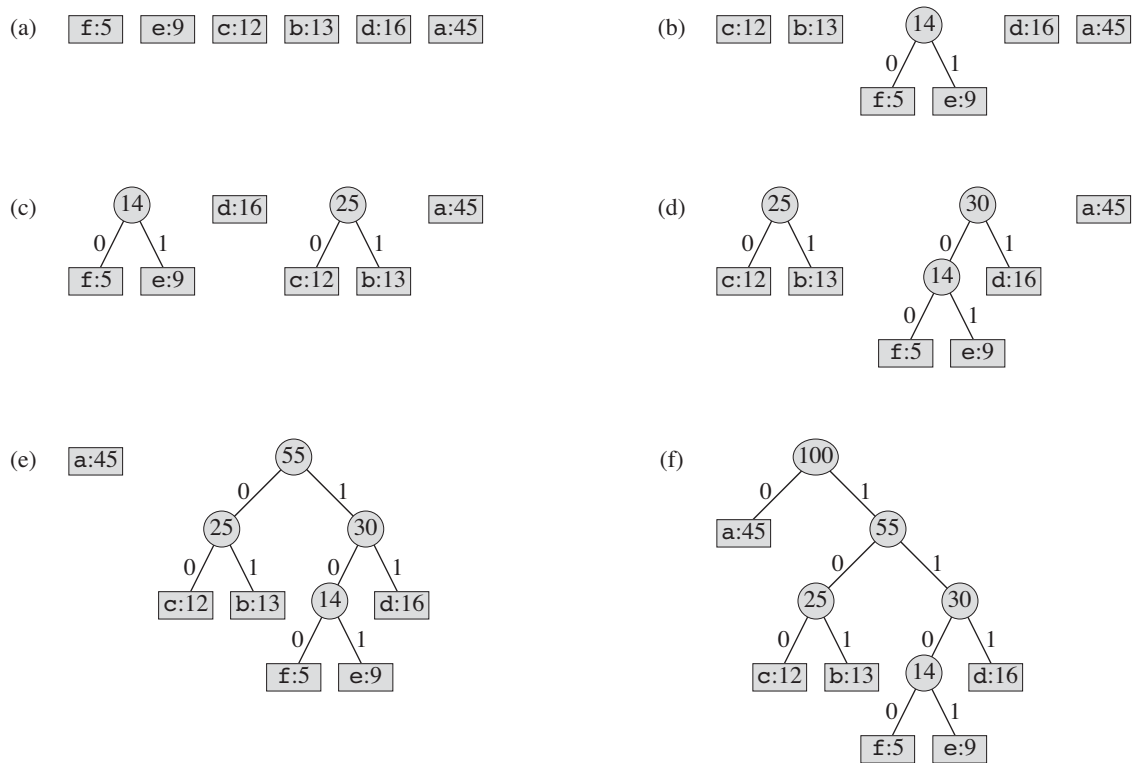


Figure 16.5 The steps of Huffman's algorithm for the frequencies given in Figure 16.3. Each part shows the contents of the queue sorted into increasing order by frequency. At each step, the two trees with lowest frequencies are merged. Leaves are shown as rectangles containing a character and its frequency. Internal nodes are shown as circles containing the sum of the frequencies of their children. An edge connecting an internal node with its children is labeled 0 if it is an edge to a left child and 1 if it is an edge to a right child. The codeword for a letter is the sequence of labels on the edges connecting the root to the leaf for that letter. (a) The initial set of $n = 6$ nodes, one for each letter. (b)–(e) Intermediate stages. (f) The final tree.

from the queue, replacing them in the queue with a new node z representing their merger. The frequency of z is computed as the sum of the frequencies of x and y in line 7. The node z has x as its left child and y as its right child. (This order is arbitrary; switching the left and right child of any node yields a different code of the same cost.) After $n - 1$ mergers, line 9 returns the one node left in the queue, which is the root of the code tree.

Although the algorithm would produce the same result if we were to excise the variables x and y —assigning directly to $z.left$ and $z.right$ in lines 5 and 6, and changing line 7 to $z.freq = z.left.freq + z.right.freq$ —we shall use the node

names x and y in the proof of correctness. Therefore, we find it convenient to leave them in.

To analyze the running time of Huffman's algorithm, we assume that Q is implemented as a binary min-heap (see Chapter 6). For a set C of n characters, we can initialize Q in line 2 in $O(n)$ time using the BUILD-MIN-HEAP procedure discussed in Section 6.3. The **for** loop in lines 3–8 executes exactly $n - 1$ times, and since each heap operation requires time $O(\lg n)$, the loop contributes $O(n \lg n)$ to the running time. Thus, the total running time of HUFFMAN on a set of n characters is $O(n \lg n)$. We can reduce the running time to $O(n \lg \lg n)$ by replacing the binary min-heap with a van Emde Boas tree (see Chapter 20).

Correctness of Huffman's algorithm

To prove that the greedy algorithm HUFFMAN is correct, we show that the problem of determining an optimal prefix code exhibits the greedy-choice and optimal-substructure properties. The next lemma shows that the greedy-choice property holds.

Lemma 16.2

Let C be an alphabet in which each character $c \in C$ has frequency $c.freq$. Let x and y be two characters in C having the lowest frequencies. Then there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.

Proof The idea of the proof is to take the tree T representing an arbitrary optimal prefix code and modify it to make a tree representing another optimal prefix code such that the characters x and y appear as sibling leaves of maximum depth in the new tree. If we can construct such a tree, then the codewords for x and y will have the same length and differ only in the last bit.

Let a and b be two characters that are sibling leaves of maximum depth in T . Without loss of generality, we assume that $a.freq \leq b.freq$ and $x.freq \leq y.freq$. Since $x.freq$ and $y.freq$ are the two lowest leaf frequencies, in order, and $a.freq$ and $b.freq$ are two arbitrary frequencies, in order, we have $x.freq \leq a.freq$ and $y.freq \leq b.freq$.

In the remainder of the proof, it is possible that we could have $x.freq = a.freq$ or $y.freq = b.freq$. However, if we had $x.freq = b.freq$, then we would also have $a.freq = b.freq = x.freq = y.freq$ (see Exercise 16.3-1), and the lemma would be trivially true. Thus, we will assume that $x.freq \neq b.freq$, which means that $x \neq b$.

As Figure 16.6 shows, we exchange the positions in T of a and x to produce a tree T' , and then we exchange the positions in T' of b and y to produce a tree T''

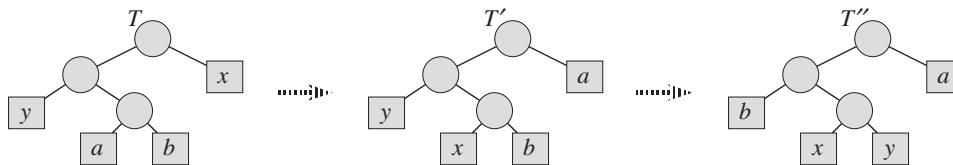


Figure 16.6 An illustration of the key step in the proof of Lemma 16.2. In the optimal tree T , leaves a and b are two siblings of maximum depth. Leaves x and y are the two characters with the lowest frequencies; they appear in arbitrary positions in T . Assuming that $x \neq b$, swapping leaves a and x produces tree T' , and then swapping leaves b and y produces tree T'' . Since each swap does not increase the cost, the resulting tree T'' is also an optimal tree.

in which x and y are sibling leaves of maximum depth. (Note that if $x = b$ but $y \neq a$, then tree T'' does not have x and y as sibling leaves of maximum depth. Because we assume that $x \neq b$, this situation cannot occur.) By equation (16.4), the difference in cost between T and T' is

$$\begin{aligned}
 B(T) - B(T') &= \sum_{c \in C} c.\text{freq} \cdot d_T(c) - \sum_{c \in C} c.\text{freq} \cdot d_{T'}(c) \\
 &= x.\text{freq} \cdot d_T(x) + a.\text{freq} \cdot d_T(a) - x.\text{freq} \cdot d_{T'}(x) - a.\text{freq} \cdot d_{T'}(a) \\
 &= x.\text{freq} \cdot d_T(x) + a.\text{freq} \cdot d_T(a) - x.\text{freq} \cdot d_T(a) - a.\text{freq} \cdot d_T(x) \\
 &= (a.\text{freq} - x.\text{freq})(d_T(a) - d_T(x)) \\
 &\geq 0,
 \end{aligned}$$

because both $a.\text{freq} - x.\text{freq}$ and $d_T(a) - d_T(x)$ are nonnegative. More specifically, $a.\text{freq} - x.\text{freq}$ is nonnegative because x is a minimum-frequency leaf, and $d_T(a) - d_T(x)$ is nonnegative because a is a leaf of maximum depth in T . Similarly, exchanging y and b does not increase the cost, and so $B(T') - B(T'')$ is nonnegative. Therefore, $B(T'') \leq B(T)$, and since T is optimal, we have $B(T) \leq B(T'')$, which implies $B(T'') = B(T)$. Thus, T'' is an optimal tree in which x and y appear as sibling leaves of maximum depth, from which the lemma follows. ■

Lemma 16.2 implies that the process of building up an optimal tree by mergers can, without loss of generality, begin with the greedy choice of merging together those two characters of lowest frequency. Why is this a greedy choice? We can view the cost of a single merger as being the sum of the frequencies of the two items being merged. Exercise 16.3-4 shows that the total cost of the tree constructed equals the sum of the costs of its mergers. Of all possible mergers at each step, HUFFMAN chooses the one that incurs the least cost.

The next lemma shows that the problem of constructing optimal prefix codes has the optimal-substructure property.

Lemma 16.3

Let C be a given alphabet with frequency $c.freq$ defined for each character $c \in C$. Let x and y be two characters in C with minimum frequency. Let C' be the alphabet C with the characters x and y removed and a new character z added, so that $C' = C - \{x, y\} \cup \{z\}$. Define f for C' as for C , except that $z.freq = x.freq + y.freq$. Let T' be any tree representing an optimal prefix code for the alphabet C' . Then the tree T , obtained from T' by replacing the leaf node for z with an internal node having x and y as children, represents an optimal prefix code for the alphabet C .

Proof We first show how to express the cost $B(T)$ of tree T in terms of the cost $B(T')$ of tree T' , by considering the component costs in equation (16.4). For each character $c \in C - \{x, y\}$, we have that $d_T(c) = d_{T'}(c)$, and hence $c.freq \cdot d_T(c) = c.freq \cdot d_{T'}(c)$. Since $d_T(x) = d_T(y) = d_{T'}(z) + 1$, we have

$$\begin{aligned} x.freq \cdot d_T(x) + y.freq \cdot d_T(y) &= (x.freq + y.freq)(d_{T'}(z) + 1) \\ &= z.freq \cdot d_{T'}(z) + (x.freq + y.freq), \end{aligned}$$

from which we conclude that

$$B(T) = B(T') + x.freq + y.freq$$

or, equivalently,

$$B(T') = B(T) - x.freq - y.freq.$$

We now prove the lemma by contradiction. Suppose that T does not represent an optimal prefix code for C . Then there exists an optimal tree T'' such that $B(T'') < B(T)$. Without loss of generality (by Lemma 16.2), T'' has x and y as siblings. Let T''' be the tree T'' with the common parent of x and y replaced by a leaf z with frequency $z.freq = x.freq + y.freq$. Then

$$\begin{aligned} B(T''') &= B(T'') - x.freq - y.freq \\ &< B(T) - x.freq - y.freq \\ &= B(T'), \end{aligned}$$

yielding a contradiction to the assumption that T' represents an optimal prefix code for C' . Thus, T must represent an optimal prefix code for the alphabet C . ■

Theorem 16.4

Procedure HUFFMAN produces an optimal prefix code.

Proof Immediate from Lemmas 16.2 and 16.3. ■

Exercises**16.3-1**

Explain why, in the proof of Lemma 16.2, if $x.freq = b.freq$, then we must have $a.freq = b.freq = x.freq = y.freq$.

16.3-2

Prove that a binary tree that is not full cannot correspond to an optimal prefix code.

16.3-3

What is an optimal Huffman code for the following set of frequencies, based on the first 8 Fibonacci numbers?

a:1 b:1 c:2 d:3 e:5 f:8 g:13 h:21

Can you generalize your answer to find the optimal code when the frequencies are the first n Fibonacci numbers?

16.3-4

Prove that we can also express the total cost of a tree for a code as the sum, over all internal nodes, of the combined frequencies of the two children of the node.

16.3-5

Prove that if we order the characters in an alphabet so that their frequencies are monotonically decreasing, then there exists an optimal code whose codeword lengths are monotonically increasing.

16.3-6

Suppose we have an optimal prefix code on a set $C = \{0, 1, \dots, n-1\}$ of characters and we wish to transmit this code using as few bits as possible. Show how to represent any optimal prefix code on C using only $2n - 1 + n \lceil \lg n \rceil$ bits. (*Hint:* Use $2n - 1$ bits to specify the structure of the tree, as discovered by a walk of the tree.)

16.3-7

Generalize Huffman's algorithm to ternary codewords (i.e., codewords using the symbols 0, 1, and 2), and prove that it yields optimal ternary codes.

16.3-8

Suppose that a data file contains a sequence of 8-bit characters such that all 256 characters are about equally common: the maximum character frequency is less than twice the minimum character frequency. Prove that Huffman coding in this case is no more efficient than using an ordinary 8-bit fixed-length code.

16.3-9

Show that no compression scheme can expect to compress a file of randomly chosen 8-bit characters by even a single bit. (*Hint:* Compare the number of possible files with the number of possible encoded files.)

★ 16.4 Matroids and greedy methods

In this section, we sketch a beautiful theory about greedy algorithms. This theory describes many situations in which the greedy method yields optimal solutions. It involves combinatorial structures known as “matroids.” Although this theory does not cover all cases for which a greedy method applies (for example, it does not cover the activity-selection problem of Section 16.1 or the Huffman-coding problem of Section 16.3), it does cover many cases of practical interest. Furthermore, this theory has been extended to cover many applications; see the notes at the end of this chapter for references.

Matroids

A **matroid** is an ordered pair $M = (S, \mathcal{I})$ satisfying the following conditions.

1. S is a finite set.
2. \mathcal{I} is a nonempty family of subsets of S , called the **independent** subsets of S , such that if $B \in \mathcal{I}$ and $A \subseteq B$, then $A \in \mathcal{I}$. We say that \mathcal{I} is **hereditary** if it satisfies this property. Note that the empty set \emptyset is necessarily a member of \mathcal{I} .
3. If $A \in \mathcal{I}$, $B \in \mathcal{I}$, and $|A| < |B|$, then there exists some element $x \in B - A$ such that $A \cup \{x\} \in \mathcal{I}$. We say that M satisfies the **exchange property**.

The word “matroid” is due to Hassler Whitney. He was studying **matric matroids**, in which the elements of S are the rows of a given matrix and a set of rows is independent if they are linearly independent in the usual sense. As Exercise 16.4-2 asks you to show, this structure defines a matroid.

As another example of matroids, consider the **graphic matroid** $M_G = (S_G, \mathcal{I}_G)$ defined in terms of a given undirected graph $G = (V, E)$ as follows:

- The set S_G is defined to be E , the set of edges of G .
- If A is a subset of E , then $A \in \mathcal{I}_G$ if and only if A is acyclic. That is, a set of edges A is independent if and only if the subgraph $G_A = (V, A)$ forms a forest.

The graphic matroid M_G is closely related to the minimum-spanning-tree problem, which Chapter 23 covers in detail.

Theorem 16.5

If $G = (V, E)$ is an undirected graph, then $M_G = (S_G, \mathcal{I}_G)$ is a matroid.

Proof Clearly, $S_G = E$ is a finite set. Furthermore, \mathcal{I}_G is hereditary, since a subset of a forest is a forest. Putting it another way, removing edges from an acyclic set of edges cannot create cycles.

Thus, it remains to show that M_G satisfies the exchange property. Suppose that $G_A = (V, A)$ and $G_B = (V, B)$ are forests of G and that $|B| > |A|$. That is, A and B are acyclic sets of edges, and B contains more edges than A does.

We claim that a forest $F = (V_F, E_F)$ contains exactly $|V_F| - |E_F|$ trees. To see why, suppose that F consists of t trees, where the i th tree contains v_i vertices and e_i edges. Then, we have

$$\begin{aligned} |E_F| &= \sum_{i=1}^t e_i \\ &= \sum_{i=1}^t (v_i - 1) \quad (\text{by Theorem B.2}) \\ &= \sum_{i=1}^t v_i - t \\ &= |V_F| - t, \end{aligned}$$

which implies that $t = |V_F| - |E_F|$. Thus, forest G_A contains $|V| - |A|$ trees, and forest G_B contains $|V| - |B|$ trees.

Since forest G_B has fewer trees than forest G_A does, forest G_B must contain some tree T whose vertices are in two different trees in forest G_A . Moreover, since T is connected, it must contain an edge (u, v) such that vertices u and v are in different trees in forest G_A . Since the edge (u, v) connects vertices in two different trees in forest G_A , we can add the edge (u, v) to forest G_A without creating a cycle. Therefore, M_G satisfies the exchange property, completing the proof that M_G is a matroid. ■

Given a matroid $M = (S, \mathcal{I})$, we call an element $x \notin A$ an *extension* of $A \in \mathcal{I}$ if we can add x to A while preserving independence; that is, x is an extension of A if $A \cup \{x\} \in \mathcal{I}$. As an example, consider a graphic matroid M_G . If A is an independent set of edges, then edge e is an extension of A if and only if e is not in A and the addition of e to A does not create a cycle.

If A is an independent subset in a matroid M , we say that A is *maximal* if it has no extensions. That is, A is maximal if it is not contained in any larger independent subset of M . The following property is often useful.

Theorem 16.6

All maximal independent subsets in a matroid have the same size.

Proof Suppose to the contrary that A is a maximal independent subset of M and there exists another larger maximal independent subset B of M . Then, the exchange property implies that for some $x \in B - A$, we can extend A to a larger independent set $A \cup \{x\}$, contradicting the assumption that A is maximal. ■

As an illustration of this theorem, consider a graphic matroid M_G for a connected, undirected graph G . Every maximal independent subset of M_G must be a free tree with exactly $|V| - 1$ edges that connects all the vertices of G . Such a tree is called a *spanning tree* of G .

We say that a matroid $M = (S, \mathcal{I})$ is **weighted** if it is associated with a weight function w that assigns a strictly positive weight $w(x)$ to each element $x \in S$. The weight function w extends to subsets of S by summation:

$$w(A) = \sum_{x \in A} w(x)$$

for any $A \subseteq S$. For example, if we let $w(e)$ denote the weight of an edge e in a graphic matroid M_G , then $w(A)$ is the total weight of the edges in edge set A .

Greedy algorithms on a weighted matroid

Many problems for which a greedy approach provides optimal solutions can be formulated in terms of finding a maximum-weight independent subset in a weighted matroid. That is, we are given a weighted matroid $M = (S, \mathcal{I})$, and we wish to find an independent set $A \in \mathcal{I}$ such that $w(A)$ is maximized. We call such a subset that is independent and has maximum possible weight an **optimal** subset of the matroid. Because the weight $w(x)$ of any element $x \in S$ is positive, an optimal subset is always a maximal independent subset—it always helps to make A as large as possible.

For example, in the **minimum-spanning-tree problem**, we are given a connected undirected graph $G = (V, E)$ and a length function w such that $w(e)$ is the (positive) length of edge e . (We use the term “length” here to refer to the original edge weights for the graph, reserving the term “weight” to refer to the weights in the associated matroid.) We wish to find a subset of the edges that connects all of the vertices together and has minimum total length. To view this as a problem of finding an optimal subset of a matroid, consider the weighted matroid M_G with weight function w' , where $w'(e) = w_0 - w(e)$ and w_0 is larger than the maximum length of any edge. In this weighted matroid, all weights are positive and an optimal subset is a spanning tree of minimum total length in the original graph. More specifically, each maximal independent subset A corresponds to a spanning tree

with $|V| - 1$ edges, and since

$$\begin{aligned}
 w'(A) &= \sum_{e \in A} w'(e) \\
 &= \sum_{e \in A} (w_0 - w(e)) \\
 &= (|V| - 1)w_0 - \sum_{e \in A} w(e) \\
 &= (|V| - 1)w_0 - w(A)
 \end{aligned}$$

for any maximal independent subset A , an independent subset that maximizes the quantity $w'(A)$ must minimize $w(A)$. Thus, any algorithm that can find an optimal subset A in an arbitrary matroid can solve the minimum-spanning-tree problem.

Chapter 23 gives algorithms for the minimum-spanning-tree problem, but here we give a greedy algorithm that works for any weighted matroid. The algorithm takes as input a weighted matroid $M = (S, \mathcal{I})$ with an associated positive weight function w , and it returns an optimal subset A . In our pseudocode, we denote the components of M by $M.S$ and $M.\mathcal{I}$ and the weight function by w . The algorithm is greedy because it considers in turn each element $x \in S$, in order of monotonically decreasing weight, and immediately adds it to the set A being accumulated if $A \cup \{x\}$ is independent.

GREEDY(M, w)

```

1   $A = \emptyset$ 
2  sort  $M.S$  into monotonically decreasing order by weight  $w$ 
3  for each  $x \in M.S$ , taken in monotonically decreasing order by weight  $w(x)$ 
4      if  $A \cup \{x\} \in M.\mathcal{I}$ 
5           $A = A \cup \{x\}$ 
6  return  $A$ 

```

Line 4 checks whether adding each element x to A would maintain A as an independent set. If A would remain independent, then line 5 adds x to A . Otherwise, x is discarded. Since the empty set is independent, and since each iteration of the **for** loop maintains A 's independence, the subset A is always independent, by induction. Therefore, GREEDY always returns an independent subset A . We shall see in a moment that A is a subset of maximum possible weight, so that A is an optimal subset.

The running time of GREEDY is easy to analyze. Let n denote $|S|$. The sorting phase of GREEDY takes time $O(n \lg n)$. Line 4 executes exactly n times, once for each element of S . Each execution of line 4 requires a check on whether or not the set $A \cup \{x\}$ is independent. If each such check takes time $O(f(n))$, the entire algorithm runs in time $O(n \lg n + nf(n))$.

We now prove that GREEDY returns an optimal subset.

Lemma 16.7 (Matroids exhibit the greedy-choice property)

Suppose that $M = (S, \mathcal{I})$ is a weighted matroid with weight function w and that S is sorted into monotonically decreasing order by weight. Let x be the first element of S such that $\{x\}$ is independent, if any such x exists. If x exists, then there exists an optimal subset A of S that contains x .

Proof If no such x exists, then the only independent subset is the empty set and the lemma is vacuously true. Otherwise, let B be any nonempty optimal subset. Assume that $x \notin B$; otherwise, letting $A = B$ gives an optimal subset of S that contains x .

No element of B has weight greater than $w(x)$. To see why, observe that $y \in B$ implies that $\{y\}$ is independent, since $B \in \mathcal{I}$ and \mathcal{I} is hereditary. Our choice of x therefore ensures that $w(x) \geq w(y)$ for any $y \in B$.

Construct the set A as follows. Begin with $A = \{x\}$. By the choice of x , set A is independent. Using the exchange property, repeatedly find a new element of B that we can add to A until $|A| = |B|$, while preserving the independence of A . At that point, A and B are the same except that A has x and B has some other element y . That is, $A = B - \{y\} \cup \{x\}$ for some $y \in B$, and so

$$\begin{aligned} w(A) &= w(B) - w(y) + w(x) \\ &\geq w(B). \end{aligned}$$

Because set B is optimal, set A , which contains x , must also be optimal. ■

We next show that if an element is not an option initially, then it cannot be an option later.

Lemma 16.8

Let $M = (S, \mathcal{I})$ be any matroid. If x is an element of S that is an extension of some independent subset A of S , then x is also an extension of \emptyset .

Proof Since x is an extension of A , we have that $A \cup \{x\}$ is independent. Since \mathcal{I} is hereditary, $\{x\}$ must be independent. Thus, x is an extension of \emptyset . ■

Corollary 16.9

Let $M = (S, \mathcal{I})$ be any matroid. If x is an element of S such that x is not an extension of \emptyset , then x is not an extension of any independent subset A of S .

Proof This corollary is simply the contrapositive of Lemma 16.8. ■

Corollary 16.9 says that any element that cannot be used immediately can never be used. Therefore, GREEDY cannot make an error by passing over any initial elements in S that are not an extension of \emptyset , since they can never be used.

Lemma 16.10 (Matroids exhibit the optimal-substructure property)

Let x be the first element of S chosen by GREEDY for the weighted matroid $M = (S, \mathcal{I})$. The remaining problem of finding a maximum-weight independent subset containing x reduces to finding a maximum-weight independent subset of the weighted matroid $M' = (S', \mathcal{I}')$, where

$$\begin{aligned} S' &= \{y \in S : \{x, y\} \in \mathcal{I}\} , \\ \mathcal{I}' &= \{B \subseteq S - \{x\} : B \cup \{x\} \in \mathcal{I}\} , \end{aligned}$$

and the weight function for M' is the weight function for M , restricted to S' . (We call M' the **contraction** of M by the element x .)

Proof If A is any maximum-weight independent subset of M containing x , then $A' = A - \{x\}$ is an independent subset of M' . Conversely, any independent subset A' of M' yields an independent subset $A = A' \cup \{x\}$ of M . Since we have in both cases that $w(A) = w(A') + w(x)$, a maximum-weight solution in M containing x yields a maximum-weight solution in M' , and vice versa. ■

Theorem 16.11 (Correctness of the greedy algorithm on matroids)

If $M = (S, \mathcal{I})$ is a weighted matroid with weight function w , then GREEDY(M, w) returns an optimal subset.

Proof By Corollary 16.9, any elements that GREEDY passes over initially because they are not extensions of \emptyset can be forgotten about, since they can never be useful. Once GREEDY selects the first element x , Lemma 16.7 implies that the algorithm does not err by adding x to A , since there exists an optimal subset containing x . Finally, Lemma 16.10 implies that the remaining problem is one of finding an optimal subset in the matroid M' that is the contraction of M by x . After the procedure GREEDY sets A to $\{x\}$, we can interpret all of its remaining steps as acting in the matroid $M' = (S', \mathcal{I}')$, because B is independent in M' if and only if $B \cup \{x\}$ is independent in M , for all sets $B \in \mathcal{I}'$. Thus, the subsequent operation of GREEDY will find a maximum-weight independent subset for M' , and the overall operation of GREEDY will find a maximum-weight independent subset for M . ■

Exercises

16.4-1

Show that (S, \mathcal{I}_k) is a matroid, where S is any finite set and \mathcal{I}_k is the set of all subsets of S of size at most k , where $k \leq |S|$.

16.4-2 ★

Given an $m \times n$ matrix T over some field (such as the reals), show that (S, \mathcal{I}) is a matroid, where S is the set of columns of T and $A \in \mathcal{I}$ if and only if the columns in A are linearly independent.

16.4-3 ★

Show that if (S, \mathcal{I}) is a matroid, then (S, \mathcal{I}') is a matroid, where

$$\mathcal{I}' = \{A' : S - A' \text{ contains some maximal } A \in \mathcal{I}\}.$$

That is, the maximal independent sets of (S, \mathcal{I}') are just the complements of the maximal independent sets of (S, \mathcal{I}) .

16.4-4 ★

Let S be a finite set and let S_1, S_2, \dots, S_k be a partition of S into nonempty disjoint subsets. Define the structure (S, \mathcal{I}) by the condition that $\mathcal{I} = \{A : |A \cap S_i| \leq 1 \text{ for } i = 1, 2, \dots, k\}$. Show that (S, \mathcal{I}) is a matroid. That is, the set of all sets A that contain at most one member of each subset in the partition determines the independent sets of a matroid.

16.4-5

Show how to transform the weight function of a weighted matroid problem, where the desired optimal solution is a *minimum-weight* maximal independent subset, to make it a standard weighted-matroid problem. Argue carefully that your transformation is correct.

★ 16.5 A task-scheduling problem as a matroid

An interesting problem that we can solve using matroids is the problem of optimally scheduling unit-time tasks on a single processor, where each task has a deadline, along with a penalty paid if the task misses its deadline. The problem looks complicated, but we can solve it in a surprisingly simple manner by casting it as a matroid and using a greedy algorithm.

A **unit-time task** is a job, such as a program to be run on a computer, that requires exactly one unit of time to complete. Given a finite set S of unit-time tasks, a

schedule for S is a permutation of S specifying the order in which to perform these tasks. The first task in the schedule begins at time 0 and finishes at time 1, the second task begins at time 1 and finishes at time 2, and so on.

The problem of **scheduling unit-time tasks with deadlines and penalties for a single processor** has the following inputs:

- a set $S = \{a_1, a_2, \dots, a_n\}$ of n unit-time tasks;
- a set of n integer **deadlines** d_1, d_2, \dots, d_n , such that each d_i satisfies $1 \leq d_i \leq n$ and task a_i is supposed to finish by time d_i ; and
- a set of n nonnegative weights or **penalties** w_1, w_2, \dots, w_n , such that we incur a penalty of w_i if task a_i is not finished by time d_i , and we incur no penalty if a task finishes by its deadline.

We wish to find a schedule for S that minimizes the total penalty incurred for missed deadlines.

Consider a given schedule. We say that a task is **late** in this schedule if it finishes after its deadline. Otherwise, the task is **early** in the schedule. We can always transform an arbitrary schedule into **early-first form**, in which the early tasks precede the late tasks. To see why, note that if some early task a_i follows some late task a_j , then we can switch the positions of a_i and a_j , and a_i will still be early and a_j will still be late.

Furthermore, we claim that we can always transform an arbitrary schedule into **canonical form**, in which the early tasks precede the late tasks and we schedule the early tasks in order of monotonically increasing deadlines. To do so, we put the schedule into early-first form. Then, as long as there exist two early tasks a_i and a_j finishing at respective times k and $k + 1$ in the schedule such that $d_j < d_i$, we swap the positions of a_i and a_j . Since a_j is early before the swap, $k + 1 \leq d_j$. Therefore, $k + 1 < d_i$, and so a_i is still early after the swap. Because task a_j is moved earlier in the schedule, it remains early after the swap.

The search for an optimal schedule thus reduces to finding a set A of tasks that we assign to be early in the optimal schedule. Having determined A , we can create the actual schedule by listing the elements of A in order of monotonically increasing deadlines, then listing the late tasks (i.e., $S - A$) in any order, producing a canonical ordering of the optimal schedule.

We say that a set A of tasks is **independent** if there exists a schedule for these tasks such that no tasks are late. Clearly, the set of early tasks for a schedule forms an independent set of tasks. Let \mathcal{I} denote the set of all independent sets of tasks.

Consider the problem of determining whether a given set A of tasks is independent. For $t = 0, 1, 2, \dots, n$, let $N_t(A)$ denote the number of tasks in A whose deadline is t or earlier. Note that $N_0(A) = 0$ for any set A .

Lemma 16.12

For any set of tasks A , the following statements are equivalent.

1. The set A is independent.
2. For $t = 0, 1, 2, \dots, n$, we have $N_t(A) \leq t$.
3. If the tasks in A are scheduled in order of monotonically increasing deadlines, then no task is late.

Proof To show that (1) implies (2), we prove the contrapositive: if $N_t(A) > t$ for some t , then there is no way to make a schedule with no late tasks for set A , because more than t tasks must finish before time t . Therefore, (1) implies (2). If (2) holds, then (3) must follow: there is no way to “get stuck” when scheduling the tasks in order of monotonically increasing deadlines, since (2) implies that the i th largest deadline is at least i . Finally, (3) trivially implies (1). ■

Using property 2 of Lemma 16.12, we can easily compute whether or not a given set of tasks is independent (see Exercise 16.5-2).

The problem of minimizing the sum of the penalties of the late tasks is the same as the problem of maximizing the sum of the penalties of the early tasks. The following theorem thus ensures that we can use the greedy algorithm to find an independent set A of tasks with the maximum total penalty.

Theorem 16.13

If S is a set of unit-time tasks with deadlines, and \mathcal{I} is the set of all independent sets of tasks, then the corresponding system (S, \mathcal{I}) is a matroid.

Proof Every subset of an independent set of tasks is certainly independent. To prove the exchange property, suppose that B and A are independent sets of tasks and that $|B| > |A|$. Let k be the largest t such that $N_t(B) \leq N_t(A)$. (Such a value of t exists, since $N_0(A) = N_0(B) = 0$.) Since $N_n(B) = |B|$ and $N_n(A) = |A|$, but $|B| > |A|$, we must have that $k < n$ and that $N_j(B) > N_j(A)$ for all j in the range $k + 1 \leq j \leq n$. Therefore, B contains more tasks with deadline $k + 1$ than A does. Let a_i be a task in $B - A$ with deadline $k + 1$. Let $A' = A \cup \{a_i\}$.

We now show that A' must be independent by using property 2 of Lemma 16.12. For $0 \leq t \leq k$, we have $N_t(A') = N_t(A) \leq t$, since A is independent. For $k < t \leq n$, we have $N_t(A') \leq N_t(B) \leq t$, since B is independent. Therefore, A' is independent, completing our proof that (S, \mathcal{I}) is a matroid. ■

By Theorem 16.11, we can use a greedy algorithm to find a maximum-weight independent set of tasks A . We can then create an optimal schedule having the tasks in A as its early tasks. This method is an efficient algorithm for scheduling

	Task						
a_i	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	70	60	50	40	30	20	10

Figure 16.7 An instance of the problem of scheduling unit-time tasks with deadlines and penalties for a single processor.

unit-time tasks with deadlines and penalties for a single processor. The running time is $O(n^2)$ using GREEDY, since each of the $O(n)$ independence checks made by that algorithm takes time $O(n)$ (see Exercise 16.5-2). Problem 16-4 gives a faster implementation.

Figure 16.7 demonstrates an example of the problem of scheduling unit-time tasks with deadlines and penalties for a single processor. In this example, the greedy algorithm selects, in order, tasks a_1, a_2, a_3 , and a_4 , then rejects a_5 (because $N_4(\{a_1, a_2, a_3, a_4, a_5\}) = 5$) and a_6 (because $N_4(\{a_1, a_2, a_3, a_4, a_6\}) = 5$), and finally accepts a_7 . The final optimal schedule is

$$\langle a_2, a_4, a_1, a_3, a_7, a_5, a_6 \rangle,$$

which has a total penalty incurred of $w_5 + w_6 = 50$.

Exercises

16.5-1

Solve the instance of the scheduling problem given in Figure 16.7, but with each penalty w_i replaced by $80 - w_i$.

16.5-2

Show how to use property 2 of Lemma 16.12 to determine in time $O(|A|)$ whether or not a given set A of tasks is independent.

Problems

16-1 Coin changing

Consider the problem of making change for n cents using the fewest number of coins. Assume that each coin's value is an integer.

- Describe a greedy algorithm to make change consisting of quarters, dimes, nickels, and pennies. Prove that your algorithm yields an optimal solution.

- b. Suppose that the available coins are in the denominations that are powers of c , i.e., the denominations are c^0, c^1, \dots, c^k for some integers $c > 1$ and $k \geq 1$. Show that the greedy algorithm always yields an optimal solution.
- c. Give a set of coin denominations for which the greedy algorithm does not yield an optimal solution. Your set should include a penny so that there is a solution for every value of n .
- d. Give an $O(nk)$ -time algorithm that makes change for any set of k different coin denominations, assuming that one of the coins is a penny.

16-2 Scheduling to minimize average completion time

Suppose you are given a set $S = \{a_1, a_2, \dots, a_n\}$ of tasks, where task a_i requires p_i units of processing time to complete, once it has started. You have one computer on which to run these tasks, and the computer can run only one task at a time. Let c_i be the **completion time** of task a_i , that is, the time at which task a_i completes processing. Your goal is to minimize the average completion time, that is, to minimize $(1/n) \sum_{i=1}^n c_i$. For example, suppose there are two tasks, a_1 and a_2 , with $p_1 = 3$ and $p_2 = 5$, and consider the schedule in which a_2 runs first, followed by a_1 . Then $c_2 = 5$, $c_1 = 8$, and the average completion time is $(5 + 8)/2 = 6.5$. If task a_1 runs first, however, then $c_1 = 3$, $c_2 = 8$, and the average completion time is $(3 + 8)/2 = 5.5$.

- a. Give an algorithm that schedules the tasks so as to minimize the average completion time. Each task must run non-preemptively, that is, once task a_i starts, it must run continuously for p_i units of time. Prove that your algorithm minimizes the average completion time, and state the running time of your algorithm.
- b. Suppose now that the tasks are not all available at once. That is, each task cannot start until its **release time** r_i . Suppose also that we allow **preemption**, so that a task can be suspended and restarted at a later time. For example, a task a_i with processing time $p_i = 6$ and release time $r_i = 1$ might start running at time 1 and be preempted at time 4. It might then resume at time 10 but be preempted at time 11, and it might finally resume at time 13 and complete at time 15. Task a_i has run for a total of 6 time units, but its running time has been divided into three pieces. In this scenario, a_i 's completion time is 15. Give an algorithm that schedules the tasks so as to minimize the average completion time in this new scenario. Prove that your algorithm minimizes the average completion time, and state the running time of your algorithm.

16-3 Acyclic subgraphs

- a. The **incidence matrix** for an undirected graph $G = (V, E)$ is a $|V| \times |E|$ matrix M such that $M_{ve} = 1$ if edge e is incident on vertex v , and $M_{ve} = 0$ otherwise. Argue that a set of columns of M is linearly independent over the field of integers modulo 2 if and only if the corresponding set of edges is acyclic. Then, use the result of Exercise 16.4-2 to provide an alternate proof that (E, \mathcal{I}) of part (a) is a matroid.
- b. Suppose that we associate a nonnegative weight $w(e)$ with each edge in an undirected graph $G = (V, E)$. Give an efficient algorithm to find an acyclic subset of E of maximum total weight.
- c. Let $G(V, E)$ be an arbitrary directed graph, and let (E, \mathcal{I}) be defined so that $A \in \mathcal{I}$ if and only if A does not contain any directed cycles. Give an example of a directed graph G such that the associated system (E, \mathcal{I}) is not a matroid. Specify which defining condition for a matroid fails to hold.
- d. The **incidence matrix** for a directed graph $G = (V, E)$ with no self-loops is a $|V| \times |E|$ matrix M such that $M_{ve} = -1$ if edge e leaves vertex v , $M_{ve} = 1$ if edge e enters vertex v , and $M_{ve} = 0$ otherwise. Argue that if a set of columns of M is linearly independent, then the corresponding set of edges does not contain a directed cycle.
- e. Exercise 16.4-2 tells us that the set of linearly independent sets of columns of any matrix M forms a matroid. Explain carefully why the results of parts (d) and (e) are not contradictory. How can there fail to be a perfect correspondence between the notion of a set of edges being acyclic and the notion of the associated set of columns of the incidence matrix being linearly independent?

16-4 Scheduling variations

Consider the following algorithm for the problem from Section 16.5 of scheduling unit-time tasks with deadlines and penalties. Let all n time slots be initially empty, where time slot i is the unit-length slot of time that finishes at time i . We consider the tasks in order of monotonically decreasing penalty. When considering task a_j , if there exists a time slot at or before a_j 's deadline d_j that is still empty, assign a_j to the latest such slot, filling it. If there is no such slot, assign task a_j to the latest of the as yet unfilled slots.

- a. Argue that this algorithm always gives an optimal answer.
- b. Use the fast disjoint-set forest presented in Section 21.3 to implement the algorithm efficiently. Assume that the set of input tasks has already been sorted into

monotonically decreasing order by penalty. Analyze the running time of your implementation.

16-5 Off-line caching

Modern computers use a cache to store a small amount of data in a fast memory. Even though a program may access large amounts of data, by storing a small subset of the main memory in the *cache*—a small but faster memory—overall access time can greatly decrease. When a computer program executes, it makes a sequence $\langle r_1, r_2, \dots, r_n \rangle$ of n memory requests, where each request is for a particular data element. For example, a program that accesses 4 distinct elements $\{a, b, c, d\}$ might make the sequence of requests $\langle d, b, d, b, d, a, c, d, b, a, c, b \rangle$. Let k be the size of the cache. When the cache contains k elements and the program requests the $(k + 1)$ st element, the system must decide, for this and each subsequent request, which k elements to keep in the cache. More precisely, for each request r_i , the cache-management algorithm checks whether element r_i is already in the cache. If it is, then we have a *cache hit*; otherwise, we have a *cache miss*. Upon a cache miss, the system retrieves r_i from the main memory, and the cache-management algorithm must decide whether to keep r_i in the cache. If it decides to keep r_i and the cache already holds k elements, then it must evict one element to make room for r_i . The cache-management algorithm evicts data with the goal of minimizing the number of cache misses over the entire sequence of requests.

Typically, caching is an on-line problem. That is, we have to make decisions about which data to keep in the cache without knowing the future requests. Here, however, we consider the off-line version of this problem, in which we are given in advance the entire sequence of n requests and the cache size k , and we wish to minimize the total number of cache misses.

We can solve this off-line problem by a greedy strategy called *furthest-in-future*, which chooses to evict the item in the cache whose next access in the request sequence comes furthest in the future.

- a. Write pseudocode for a cache manager that uses the furthest-in-future strategy. The input should be a sequence $\langle r_1, r_2, \dots, r_n \rangle$ of requests and a cache size k , and the output should be a sequence of decisions about which data element (if any) to evict upon each request. What is the running time of your algorithm?
- b. Show that the off-line caching problem exhibits optimal substructure.
- c. Prove that furthest-in-future produces the minimum possible number of cache misses.

Chapter notes

Much more material on greedy algorithms and matroids can be found in Lawler [224] and Papadimitriou and Steiglitz [271].

The greedy algorithm first appeared in the combinatorial optimization literature in a 1971 article by Edmonds [101], though the theory of matroids dates back to a 1935 article by Whitney [355].

Our proof of the correctness of the greedy algorithm for the activity-selection problem is based on that of Gavril [131]. The task-scheduling problem is studied in Lawler [224]; Horowitz, Sahni, and Rajasekaran [181]; and Brassard and Bratley [54].

Huffman codes were invented in 1952 [185]; Lelewer and Hirschberg [231] surveys data-compression techniques known as of 1987.

An extension of matroid theory to greedoid theory was pioneered by Korte and Lovász [216, 217, 218, 219], who greatly generalize the theory presented here.

In an *amortized analysis*, we average the time required to perform a sequence of data-structure operations over all the operations performed. With amortized analysis, we can show that the average cost of an operation is small, if we average over a sequence of operations, even though a single operation within the sequence might be expensive. Amortized analysis differs from average-case analysis in that probability is not involved; an amortized analysis guarantees the *average performance of each operation in the worst case*.

The first three sections of this chapter cover the three most common techniques used in amortized analysis. Section 17.1 starts with aggregate analysis, in which we determine an upper bound $T(n)$ on the total cost of a sequence of n operations. The average cost per operation is then $T(n)/n$. We take the average cost as the amortized cost of each operation, so that all operations have the same amortized cost.

Section 17.2 covers the accounting method, in which we determine an amortized cost of each operation. When there is more than one type of operation, each type of operation may have a different amortized cost. The accounting method overcharges some operations early in the sequence, storing the overcharge as “prepaid credit” on specific objects in the data structure. Later in the sequence, the credit pays for operations that are charged less than they actually cost.

Section 17.3 discusses the potential method, which is like the accounting method in that we determine the amortized cost of each operation and may overcharge operations early on to compensate for undercharges later. The potential method maintains the credit as the “potential energy” of the data structure as a whole instead of associating the credit with individual objects within the data structure.

We shall use two examples to examine these three methods. One is a stack with the additional operation `MULTIPOP`, which pops several objects at once. The other is a binary counter that counts up from 0 by means of the single operation `INCREMENT`.

While reading this chapter, bear in mind that the charges assigned during an amortized analysis are for analysis purposes only. They need not—and should not—appear in the code. If, for example, we assign a credit to an object x when using the accounting method, we have no need to assign an appropriate amount to some attribute, such as $x.credit$, in the code.

When we perform an amortized analysis, we often gain insight into a particular data structure, and this insight can help us optimize the design. In Section 17.4, for example, we shall use the potential method to analyze a dynamically expanding and contracting table.

17.1 Aggregate analysis

In *aggregate analysis*, we show that for all n , a sequence of n operations takes *worst-case* time $T(n)$ in total. In the worst case, the average cost, or *amortized cost*, per operation is therefore $T(n)/n$. Note that this amortized cost applies to each operation, even when there are several types of operations in the sequence. The other two methods we shall study in this chapter, the accounting method and the potential method, may assign different amortized costs to different types of operations.

Stack operations

In our first example of aggregate analysis, we analyze stacks that have been augmented with a new operation. Section 10.1 presented the two fundamental stack operations, each of which takes $O(1)$ time:

PUSH(S, x) pushes object x onto stack S .

POP(S) pops the top of stack S and returns the popped object. Calling POP on an empty stack generates an error.

Since each of these operations runs in $O(1)$ time, let us consider the cost of each to be 1. The total cost of a sequence of n PUSH and POP operations is therefore n , and the actual running time for n operations is therefore $\Theta(n)$.

Now we add the stack operation MULTIPOP(S, k), which removes the k top objects of stack S , popping the entire stack if the stack contains fewer than k objects. Of course, we assume that k is positive; otherwise the MULTIPOP operation leaves the stack unchanged. In the following pseudocode, the operation STACK-EMPTY returns TRUE if there are no objects currently on the stack, and FALSE otherwise.

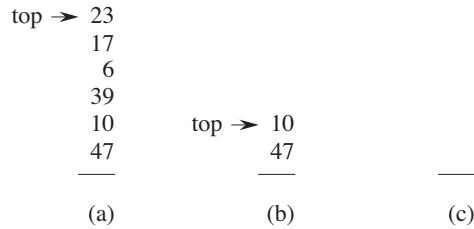


Figure 17.1 The action of MULTIPOP on a stack S , shown initially in (a). The top 4 objects are popped by $\text{MULTIPOP}(S, 4)$, whose result is shown in (b). The next operation is $\text{MULTIPOP}(S, 7)$, which empties the stack—shown in (c)—since there were fewer than 7 objects remaining.

$\text{MULTIPOP}(S, k)$

```

1  while not STACK-EMPTY( $S$ ) and  $k > 0$ 
2      POP( $S$ )
3       $k = k - 1$ 

```

Figure 17.1 shows an example of MULTIPOP .

What is the running time of $\text{MULTIPOP}(S, k)$ on a stack of s objects? The actual running time is linear in the number of POP operations actually executed, and thus we can analyze MULTIPOP in terms of the abstract costs of 1 each for PUSH and POP. The number of iterations of the **while** loop is the number $\min(s, k)$ of objects popped off the stack. Each iteration of the loop makes one call to POP in line 2. Thus, the total cost of MULTIPOP is $\min(s, k)$, and the actual running time is a linear function of this cost.

Let us analyze a sequence of n PUSH, POP, and MULTIPOP operations on an initially empty stack. The worst-case cost of a MULTIPOP operation in the sequence is $O(n)$, since the stack size is at most n . The worst-case time of any stack operation is therefore $O(n)$, and hence a sequence of n operations costs $O(n^2)$, since we may have $O(n)$ MULTIPOP operations costing $O(n)$ each. Although this analysis is correct, the $O(n^2)$ result, which we obtained by considering the worst-case cost of each operation individually, is not tight.

Using aggregate analysis, we can obtain a better upper bound that considers the entire sequence of n operations. In fact, although a single MULTIPOP operation can be expensive, any sequence of n PUSH, POP, and MULTIPOP operations on an initially empty stack can cost at most $O(n)$. Why? We can pop each object from the stack at most once for each time we have pushed it onto the stack. Therefore, the number of times that POP can be called on a nonempty stack, including calls within MULTIPOP , is at most the number of PUSH operations, which is at most n . For any value of n , any sequence of n PUSH, POP, and MULTIPOP operations takes a total of $O(n)$ time. The average cost of an operation is $O(n)/n = O(1)$. In aggregate

analysis, we assign the amortized cost of each operation to be the average cost. In this example, therefore, all three stack operations have an amortized cost of $O(1)$.

We emphasize again that although we have just shown that the average cost, and hence the running time, of a stack operation is $O(1)$, we did not use probabilistic reasoning. We actually showed a *worst-case* bound of $O(n)$ on a sequence of n operations. Dividing this total cost by n yielded the average cost per operation, or the amortized cost.

Incrementing a binary counter

As another example of aggregate analysis, consider the problem of implementing a k -bit binary counter that counts upward from 0. We use an array $A[0 \dots k - 1]$ of bits, where $A.length = k$, as the counter. A binary number x that is stored in the counter has its lowest-order bit in $A[0]$ and its highest-order bit in $A[k - 1]$, so that $x = \sum_{i=0}^{k-1} A[i] \cdot 2^i$. Initially, $x = 0$, and thus $A[i] = 0$ for $i = 0, 1, \dots, k - 1$. To add 1 (modulo 2^k) to the value in the counter, we use the following procedure.

```

INCREMENT( $A$ )
1   $i = 0$ 
2  while  $i < A.length$  and  $A[i] == 1$ 
3       $A[i] = 0$ 
4       $i = i + 1$ 
5  if  $i < A.length$ 
6       $A[i] = 1$ 

```

Figure 17.2 shows what happens to a binary counter as we increment it 16 times, starting with the initial value 0 and ending with the value 16. At the start of each iteration of the **while** loop in lines 2–4, we wish to add a 1 into position i . If $A[i] = 1$, then adding 1 flips the bit to 0 in position i and yields a carry of 1, to be added into position $i + 1$ on the next iteration of the loop. Otherwise, the loop ends, and then, if $i < k$, we know that $A[i] = 0$, so that line 6 adds a 1 into position i , flipping the 0 to a 1. The cost of each INCREMENT operation is linear in the number of bits flipped.

As with the stack example, a cursory analysis yields a bound that is correct but not tight. A single execution of INCREMENT takes time $\Theta(k)$ in the worst case, in which array A contains all 1s. Thus, a sequence of n INCREMENT operations on an initially zero counter takes time $O(nk)$ in the worst case.

We can tighten our analysis to yield a worst-case cost of $O(n)$ for a sequence of n INCREMENT operations by observing that not all bits flip each time INCREMENT is called. As Figure 17.2 shows, $A[0]$ does flip each time INCREMENT is called. The next bit up, $A[1]$, flips only every other time: a sequence of n INCREMENT

Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Total cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

Figure 17.2 An 8-bit binary counter as its value goes from 0 to 16 by a sequence of 16 INCREMENT operations. Bits that flip to achieve the next value are shaded. The running cost for flipping bits is shown at the right. Notice that the total cost is always less than twice the total number of INCREMENT operations.

operations on an initially zero counter causes $A[1]$ to flip $\lfloor n/2 \rfloor$ times. Similarly, bit $A[2]$ flips only every fourth time, or $\lfloor n/4 \rfloor$ times in a sequence of n INCREMENT operations. In general, for $i = 0, 1, \dots, k-1$, bit $A[i]$ flips $\lfloor n/2^i \rfloor$ times in a sequence of n INCREMENT operations on an initially zero counter. For $i \geq k$, bit $A[i]$ does not exist, and so it cannot flip. The total number of flips in the sequence is thus

$$\begin{aligned} \sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor &< n \sum_{i=0}^{\infty} \frac{1}{2^i} \\ &= 2n, \end{aligned}$$

by equation (A.6). The worst-case time for a sequence of n INCREMENT operations on an initially zero counter is therefore $O(n)$. The average cost of each operation, and therefore the amortized cost per operation, is $O(n)/n = O(1)$.

Exercises

17.1-1

If the set of stack operations included a MULTIPUSH operation, which pushes k items onto the stack, would the $O(1)$ bound on the amortized cost of stack operations continue to hold?

17.1-2

Show that if a DECREMENT operation were included in the k -bit counter example, n operations could cost as much as $\Theta(nk)$ time.

17.1-3

Suppose we perform a sequence of n operations on a data structure in which the i th operation costs i if i is an exact power of 2, and 1 otherwise. Use aggregate analysis to determine the amortized cost per operation.

17.2 The accounting method

In the *accounting method* of amortized analysis, we assign differing charges to different operations, with some operations charged more or less than they actually cost. We call the amount we charge an operation its *amortized cost*. When an operation's amortized cost exceeds its actual cost, we assign the difference to specific objects in the data structure as *credit*. Credit can help pay for later operations whose amortized cost is less than their actual cost. Thus, we can view the amortized cost of an operation as being split between its actual cost and credit that is either deposited or used up. Different operations may have different amortized costs. This method differs from aggregate analysis, in which all operations have the same amortized cost.

We must choose the amortized costs of operations carefully. If we want to show that in the worst case the average cost per operation is small by analyzing with amortized costs, we must ensure that the total amortized cost of a sequence of operations provides an upper bound on the total actual cost of the sequence. Moreover, as in aggregate analysis, this relationship must hold for all sequences of operations. If we denote the actual cost of the i th operation by c_i and the amortized cost of the i th operation by \hat{c}_i , we require

$$\sum_{i=1}^n \hat{c}_i \geq \sum_{i=1}^n c_i \quad (17.1)$$

for all sequences of n operations. The total credit stored in the data structure is the difference between the total amortized cost and the total actual cost, or

$\sum_{i=1}^n \hat{c}_i - \sum_{i=1}^n c_i$. By inequality (17.1), the total credit associated with the data structure must be nonnegative at all times. If we ever were to allow the total credit to become negative (the result of undercharging early operations with the promise of repaying the account later on), then the total amortized costs incurred at that time would be below the total actual costs incurred; for the sequence of operations up to that time, the total amortized cost would not be an upper bound on the total actual cost. Thus, we must take care that the total credit in the data structure never becomes negative.

Stack operations

To illustrate the accounting method of amortized analysis, let us return to the stack example. Recall that the actual costs of the operations were

PUSH 1 ,
 POP 1 ,
 MULTIPOP $\min(k, s)$,

where k is the argument supplied to MULTIPOP and s is the stack size when it is called. Let us assign the following amortized costs:

PUSH 2 ,
 POP 0 ,
 MULTIPOP 0 .

Note that the amortized cost of MULTIPOP is a constant (0), whereas the actual cost is variable. Here, all three amortized costs are constant. In general, the amortized costs of the operations under consideration may differ from each other, and they may even differ asymptotically.

We shall now show that we can pay for any sequence of stack operations by charging the amortized costs. Suppose we use a dollar bill to represent each unit of cost. We start with an empty stack. Recall the analogy of Section 10.1 between the stack data structure and a stack of plates in a cafeteria. When we push a plate on the stack, we use 1 dollar to pay the actual cost of the push and are left with a credit of 1 dollar (out of the 2 dollars charged), which we leave on top of the plate. At any point in time, every plate on the stack has a dollar of credit on it.

The dollar stored on the plate serves as prepayment for the cost of popping it from the stack. When we execute a POP operation, we charge the operation nothing and pay its actual cost using the credit stored in the stack. To pop a plate, we take the dollar of credit off the plate and use it to pay the actual cost of the operation. Thus, by charging the PUSH operation a little bit more, we can charge the POP operation nothing.

Moreover, we can also charge MULTIPOP operations nothing. To pop the first plate, we take the dollar of credit off the plate and use it to pay the actual cost of a POP operation. To pop a second plate, we again have a dollar of credit on the plate to pay for the POP operation, and so on. Thus, we have always charged enough up front to pay for MULTIPOP operations. In other words, since each plate on the stack has 1 dollar of credit on it, and the stack always has a nonnegative number of plates, we have ensured that the amount of credit is always nonnegative. Thus, for *any* sequence of n PUSH, POP, and MULTIPOP operations, the total amortized cost is an upper bound on the total actual cost. Since the total amortized cost is $O(n)$, so is the total actual cost.

Incrementing a binary counter

As another illustration of the accounting method, we analyze the INCREMENT operation on a binary counter that starts at zero. As we observed earlier, the running time of this operation is proportional to the number of bits flipped, which we shall use as our cost for this example. Let us once again use a dollar bill to represent each unit of cost (the flipping of a bit in this example).

For the amortized analysis, let us charge an amortized cost of 2 dollars to set a bit to 1. When a bit is set, we use 1 dollar (out of the 2 dollars charged) to pay for the actual setting of the bit, and we place the other dollar on the bit as credit to be used later when we flip the bit back to 0. At any point in time, every 1 in the counter has a dollar of credit on it, and thus we can charge nothing to reset a bit to 0; we just pay for the reset with the dollar bill on the bit.

Now we can determine the amortized cost of INCREMENT. The cost of resetting the bits within the **while** loop is paid for by the dollars on the bits that are reset. The INCREMENT procedure sets at most one bit, in line 6, and therefore the amortized cost of an INCREMENT operation is at most 2 dollars. The number of 1s in the counter never becomes negative, and thus the amount of credit stays nonnegative at all times. Thus, for n INCREMENT operations, the total amortized cost is $O(n)$, which bounds the total actual cost.

Exercises

17.2-1

Suppose we perform a sequence of stack operations on a stack whose size never exceeds k . After every k operations, we make a copy of the entire stack for backup purposes. Show that the cost of n stack operations, including copying the stack, is $O(n)$ by assigning suitable amortized costs to the various stack operations.

17.2-2

Redo Exercise 17.1-3 using an accounting method of analysis.

17.2-3

Suppose we wish not only to increment a counter but also to reset it to zero (i.e., make all bits in it 0). Counting the time to examine or modify a bit as $\Theta(1)$, show how to implement a counter as an array of bits so that any sequence of n INCREMENT and RESET operations takes time $O(n)$ on an initially zero counter. (*Hint*: Keep a pointer to the high-order 1.)

17.3 The potential method

Instead of representing prepaid work as credit stored with specific objects in the data structure, the *potential method* of amortized analysis represents the prepaid work as “potential energy,” or just “potential,” which can be released to pay for future operations. We associate the potential with the data structure as a whole rather than with specific objects within the data structure.

The potential method works as follows. We will perform n operations, starting with an initial data structure D_0 . For each $i = 1, 2, \dots, n$, we let c_i be the actual cost of the i th operation and D_i be the data structure that results after applying the i th operation to data structure D_{i-1} . A *potential function* Φ maps each data structure D_i to a real number $\Phi(D_i)$, which is the *potential* associated with data structure D_i . The *amortized cost* \hat{c}_i of the i th operation with respect to potential function Φ is defined by

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) . \quad (17.2)$$

The amortized cost of each operation is therefore its actual cost plus the change in potential due to the operation. By equation (17.2), the total amortized cost of the n operations is

$$\begin{aligned} \sum_{i=1}^n \hat{c}_i &= \sum_{i=1}^n (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^n c_i + \Phi(D_n) - \Phi(D_0) . \end{aligned} \quad (17.3)$$

The second equality follows from equation (A.9) because the $\Phi(D_i)$ terms telescope.

If we can define a potential function Φ so that $\Phi(D_n) \geq \Phi(D_0)$, then the total amortized cost $\sum_{i=1}^n \hat{c}_i$ gives an upper bound on the total actual cost $\sum_{i=1}^n c_i$.

In practice, we do not always know how many operations might be performed. Therefore, if we require that $\Phi(D_i) \geq \Phi(D_0)$ for all i , then we guarantee, as in the accounting method, that we pay in advance. We usually just define $\Phi(D_0)$ to be 0 and then show that $\Phi(D_i) \geq 0$ for all i . (See Exercise 17.3-1 for an easy way to handle cases in which $\Phi(D_0) \neq 0$.)

Intuitively, if the potential difference $\Phi(D_i) - \Phi(D_{i-1})$ of the i th operation is positive, then the amortized cost \hat{c}_i represents an overcharge to the i th operation, and the potential of the data structure increases. If the potential difference is negative, then the amortized cost represents an undercharge to the i th operation, and the decrease in the potential pays for the actual cost of the operation.

The amortized costs defined by equations (17.2) and (17.3) depend on the choice of the potential function Φ . Different potential functions may yield different amortized costs yet still be upper bounds on the actual costs. We often find trade-offs that we can make in choosing a potential function; the best potential function to use depends on the desired time bounds.

Stack operations

To illustrate the potential method, we return once again to the example of the stack operations PUSH, POP, and MULTIPOP. We define the potential function Φ on a stack to be the number of objects in the stack. For the empty stack D_0 with which we start, we have $\Phi(D_0) = 0$. Since the number of objects in the stack is never negative, the stack D_i that results after the i th operation has nonnegative potential, and thus

$$\begin{aligned}\Phi(D_i) &\geq 0 \\ &= \Phi(D_0) .\end{aligned}$$

The total amortized cost of n operations with respect to Φ therefore represents an upper bound on the actual cost.

Let us now compute the amortized costs of the various stack operations. If the i th operation on a stack containing s objects is a PUSH operation, then the potential difference is

$$\begin{aligned}\Phi(D_i) - \Phi(D_{i-1}) &= (s + 1) - s \\ &= 1 .\end{aligned}$$

By equation (17.2), the amortized cost of this PUSH operation is

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + 1 \\ &= 2 .\end{aligned}$$

Suppose that the i th operation on the stack is $\text{MULTIPOP}(S, k)$, which causes $k' = \min(k, s)$ objects to be popped off the stack. The actual cost of the operation is k' , and the potential difference is

$$\Phi(D_i) - \Phi(D_{i-1}) = -k'.$$

Thus, the amortized cost of the MULTIPOP operation is

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= k' - k' \\ &= 0.\end{aligned}$$

Similarly, the amortized cost of an ordinary POP operation is 0.

The amortized cost of each of the three operations is $O(1)$, and thus the total amortized cost of a sequence of n operations is $O(n)$. Since we have already argued that $\Phi(D_i) \geq \Phi(D_0)$, the total amortized cost of n operations is an upper bound on the total actual cost. The worst-case cost of n operations is therefore $O(n)$.

Incrementing a binary counter

As another example of the potential method, we again look at incrementing a binary counter. This time, we define the potential of the counter after the i th INCREMENT operation to be b_i , the number of 1s in the counter after the i th operation.

Let us compute the amortized cost of an INCREMENT operation. Suppose that the i th INCREMENT operation resets t_i bits. The actual cost of the operation is therefore at most $t_i + 1$, since in addition to resetting t_i bits, it sets at most one bit to 1. If $b_i = 0$, then the i th operation resets all k bits, and so $b_{i-1} = t_i = k$. If $b_i > 0$, then $b_i = b_{i-1} - t_i + 1$. In either case, $b_i \leq b_{i-1} - t_i + 1$, and the potential difference is

$$\begin{aligned}\Phi(D_i) - \Phi(D_{i-1}) &\leq (b_{i-1} - t_i + 1) - b_{i-1} \\ &= 1 - t_i.\end{aligned}$$

The amortized cost is therefore

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &\leq (t_i + 1) + (1 - t_i) \\ &= 2.\end{aligned}$$

If the counter starts at zero, then $\Phi(D_0) = 0$. Since $\Phi(D_i) \geq 0$ for all i , the total amortized cost of a sequence of n INCREMENT operations is an upper bound on the total actual cost, and so the worst-case cost of n INCREMENT operations is $O(n)$.

The potential method gives us an easy way to analyze the counter even when it does not start at zero. The counter starts with b_0 1s, and after n INCREMENT

operations it has b_n 1s, where $0 \leq b_0, b_n \leq k$. (Recall that k is the number of bits in the counter.) We can rewrite equation (17.3) as

$$\sum_{i=1}^n c_i = \sum_{i=1}^n \hat{c}_i - \Phi(D_n) + \Phi(D_0). \quad (17.4)$$

We have $\hat{c}_i \leq 2$ for all $1 \leq i \leq n$. Since $\Phi(D_0) = b_0$ and $\Phi(D_n) = b_n$, the total actual cost of n INCREMENT operations is

$$\begin{aligned} \sum_{i=1}^n c_i &\leq \sum_{i=1}^n 2 - b_n + b_0 \\ &= 2n - b_n + b_0. \end{aligned}$$

Note in particular that since $b_0 \leq k$, as long as $k = O(n)$, the total actual cost is $O(n)$. In other words, if we execute at least $n = \Omega(k)$ INCREMENT operations, the total actual cost is $O(n)$, no matter what initial value the counter contains.

Exercises

17.3-1

Suppose we have a potential function Φ such that $\Phi(D_i) \geq \Phi(D_0)$ for all i , but $\Phi(D_0) \neq 0$. Show that there exists a potential function Φ' such that $\Phi'(D_0) = 0$, $\Phi'(D_i) \geq 0$ for all $i \geq 1$, and the amortized costs using Φ' are the same as the amortized costs using Φ .

17.3-2

Redo Exercise 17.1-3 using a potential method of analysis.

17.3-3

Consider an ordinary binary min-heap data structure with n elements supporting the instructions INSERT and EXTRACT-MIN in $O(\lg n)$ worst-case time. Give a potential function Φ such that the amortized cost of INSERT is $O(\lg n)$ and the amortized cost of EXTRACT-MIN is $O(1)$, and show that it works.

17.3-4

What is the total cost of executing n of the stack operations PUSH, POP, and MULTIPOP, assuming that the stack begins with s_0 objects and finishes with s_n objects?

17.3-5

Suppose that a counter begins at a number with b 1s in its binary representation, rather than at 0. Show that the cost of performing n INCREMENT operations is $O(n)$ if $n = \Omega(b)$. (Do not assume that b is constant.)

17.3-6

Show how to implement a queue with two ordinary stacks (Exercise 10.1-6) so that the amortized cost of each ENQUEUE and each DEQUEUE operation is $O(1)$.

17.3-7

Design a data structure to support the following two operations for a dynamic multiset S of integers, which allows duplicate values:

INSERT(S, x) inserts x into S .

DELETE-LARGER-HALF(S) deletes the largest $\lceil |S|/2 \rceil$ elements from S .

Explain how to implement this data structure so that any sequence of m INSERT and DELETE-LARGER-HALF operations runs in $O(m)$ time. Your implementation should also include a way to output the elements of S in $O(|S|)$ time.

17.4 Dynamic tables

We do not always know in advance how many objects some applications will store in a table. We might allocate space for a table, only to find out later that it is not enough. We must then reallocate the table with a larger size and copy all objects stored in the original table over into the new, larger table. Similarly, if many objects have been deleted from the table, it may be worthwhile to reallocate the table with a smaller size. In this section, we study this problem of dynamically expanding and contracting a table. Using amortized analysis, we shall show that the amortized cost of insertion and deletion is only $O(1)$, even though the actual cost of an operation is large when it triggers an expansion or a contraction. Moreover, we shall see how to guarantee that the unused space in a dynamic table never exceeds a constant fraction of the total space.

We assume that the dynamic table supports the operations TABLE-INSERT and TABLE-DELETE. TABLE-INSERT inserts into the table an item that occupies a single *slot*, that is, a space for one item. Likewise, TABLE-DELETE removes an item from the table, thereby freeing a slot. The details of the data-structuring method used to organize the table are unimportant; we might use a stack (Section 10.1), a heap (Chapter 6), or a hash table (Chapter 11). We might also use an array or collection of arrays to implement object storage, as we did in Section 10.3.

We shall find it convenient to use a concept introduced in our analysis of hashing (Chapter 11). We define the **load factor** $\alpha(T)$ of a nonempty table T to be the number of items stored in the table divided by the size (number of slots) of the table. We assign an empty table (one with no items) size 0, and we define its load factor to be 1. If the load factor of a dynamic table is bounded below by a constant,

the unused space in the table is never more than a constant fraction of the total amount of space.

We start by analyzing a dynamic table in which we only insert items. We then consider the more general case in which we both insert and delete items.

17.4.1 Table expansion

Let us assume that storage for a table is allocated as an array of slots. A table fills up when all slots have been used or, equivalently, when its load factor is 1.¹ In some software environments, upon attempting to insert an item into a full table, the only alternative is to abort with an error. We shall assume, however, that our software environment, like many modern ones, provides a memory-management system that can allocate and free blocks of storage on request. Thus, upon inserting an item into a full table, we can *expand* the table by allocating a new table with more slots than the old table had. Because we always need the table to reside in contiguous memory, we must allocate a new array for the larger table and then copy items from the old table into the new table.

A common heuristic allocates a new table with twice as many slots as the old one. If the only table operations are insertions, then the load factor of the table is always at least $1/2$, and thus the amount of wasted space never exceeds half the total space in the table.

In the following pseudocode, we assume that T is an object representing the table. The attribute $T.table$ contains a pointer to the block of storage representing the table, $T.num$ contains the number of items in the table, and $T.size$ gives the total number of slots in the table. Initially, the table is empty: $T.num = T.size = 0$.

TABLE-INSERT(T, x)

```

1  if  $T.size == 0$ 
2      allocate  $T.table$  with 1 slot
3       $T.size = 1$ 
4  if  $T.num == T.size$ 
5      allocate new-table with  $2 \cdot T.size$  slots
6      insert all items in  $T.table$  into new-table
7      free  $T.table$ 
8       $T.table = \textit{new-table}$ 
9       $T.size = 2 \cdot T.size$ 
10 insert  $x$  into  $T.table$ 
11  $T.num = T.num + 1$ 
```

¹In some situations, such as an open-address hash table, we may wish to consider a table to be full if its load factor equals some constant strictly less than 1. (See Exercise 17.4-1.)

Notice that we have two “insertion” procedures here: the TABLE-INSERT procedure itself and the *elementary insertion* into a table in lines 6 and 10. We can analyze the running time of TABLE-INSERT in terms of the number of elementary insertions by assigning a cost of 1 to each elementary insertion. We assume that the actual running time of TABLE-INSERT is linear in the time to insert individual items, so that the overhead for allocating an initial table in line 2 is constant and the overhead for allocating and freeing storage in lines 5 and 7 is dominated by the cost of transferring items in line 6. We call the event in which lines 5–9 are executed an *expansion*.

Let us analyze a sequence of n TABLE-INSERT operations on an initially empty table. What is the cost c_i of the i th operation? If the current table has room for the new item (or if this is the first operation), then $c_i = 1$, since we need only perform the one elementary insertion in line 10. If the current table is full, however, and an expansion occurs, then $c_i = i$: the cost is 1 for the elementary insertion in line 10 plus $i - 1$ for the items that we must copy from the old table to the new table in line 6. If we perform n operations, the worst-case cost of an operation is $O(n)$, which leads to an upper bound of $O(n^2)$ on the total running time for n operations.

This bound is not tight, because we rarely expand the table in the course of n TABLE-INSERT operations. Specifically, the i th operation causes an expansion only when $i - 1$ is an exact power of 2. The amortized cost of an operation is in fact $O(1)$, as we can show using aggregate analysis. The cost of the i th operation is

$$c_i = \begin{cases} i & \text{if } i - 1 \text{ is an exact power of } 2, \\ 1 & \text{otherwise.} \end{cases}$$

The total cost of n TABLE-INSERT operations is therefore

$$\begin{aligned} \sum_{i=1}^n c_i &\leq n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j \\ &< n + 2n \\ &= 3n, \end{aligned}$$

because at most n operations cost 1 and the costs of the remaining operations form a geometric series. Since the total cost of n TABLE-INSERT operations is bounded by $3n$, the amortized cost of a single operation is at most 3.

By using the accounting method, we can gain some feeling for why the amortized cost of a TABLE-INSERT operation should be 3. Intuitively, each item pays for 3 elementary insertions: inserting itself into the current table, moving itself when the table expands, and moving another item that has already been moved once when the table expands. For example, suppose that the size of the table is m immediately after an expansion. Then the table holds $m/2$ items, and it contains

no credit. We charge 3 dollars for each insertion. The elementary insertion that occurs immediately costs 1 dollar. We place another dollar as credit on the item inserted. We place the third dollar as credit on one of the $m/2$ items already in the table. The table will not fill again until we have inserted another $m/2 - 1$ items, and thus, by the time the table contains m items and is full, we will have placed a dollar on each item to pay to reinsert it during the expansion.

We can use the potential method to analyze a sequence of n TABLE-INSERT operations, and we shall use it in Section 17.4.2 to design a TABLE-DELETE operation that has an $O(1)$ amortized cost as well. We start by defining a potential function Φ that is 0 immediately after an expansion but builds to the table size by the time the table is full, so that we can pay for the next expansion by the potential. The function

$$\Phi(T) = 2 \cdot T.num - T.size \quad (17.5)$$

is one possibility. Immediately after an expansion, we have $T.num = T.size/2$, and thus $\Phi(T) = 0$, as desired. Immediately before an expansion, we have $T.num = T.size$, and thus $\Phi(T) = T.num$, as desired. The initial value of the potential is 0, and since the table is always at least half full, $T.num \geq T.size/2$, which implies that $\Phi(T)$ is always nonnegative. Thus, the sum of the amortized costs of n TABLE-INSERT operations gives an upper bound on the sum of the actual costs.

To analyze the amortized cost of the i th TABLE-INSERT operation, we let num_i denote the number of items stored in the table after the i th operation, $size_i$ denote the total size of the table after the i th operation, and Φ_i denote the potential after the i th operation. Initially, we have $num_0 = 0$, $size_0 = 0$, and $\Phi_0 = 0$.

If the i th TABLE-INSERT operation does not trigger an expansion, then we have $size_i = size_{i-1}$ and the amortized cost of the operation is

$$\begin{aligned} \hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1}) \\ &= 1 + (2 \cdot num_i - size_i) - (2(num_i - 1) - size_i) \\ &= 3. \end{aligned}$$

If the i th operation does trigger an expansion, then we have $size_i = 2 \cdot size_{i-1}$ and $size_{i-1} = num_{i-1} = num_i - 1$, which implies that $size_i = 2 \cdot (num_i - 1)$. Thus, the amortized cost of the operation is

$$\begin{aligned} \hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= num_i + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1}) \\ &= num_i + (2 \cdot num_i - 2 \cdot (num_i - 1)) - (2(num_i - 1) - (num_i - 1)) \\ &= num_i + 2 - (num_i - 1) \\ &= 3. \end{aligned}$$

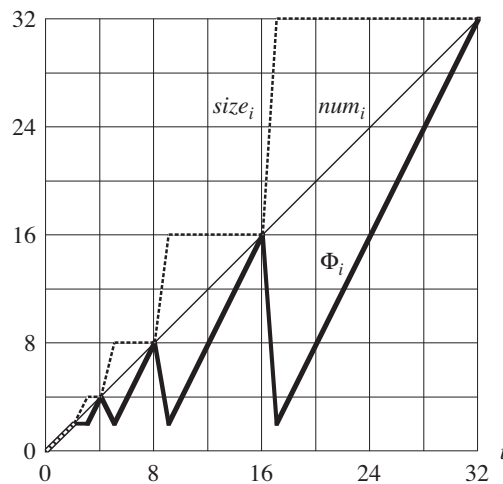


Figure 17.3 The effect of a sequence of n TABLE-INSERT operations on the number num_i of items in the table, the number $size_i$ of slots in the table, and the potential $\Phi_i = 2 \cdot num_i - size_i$, each being measured after the i th operation. The thin line shows num_i , the dashed line shows $size_i$, and the thick line shows Φ_i . Notice that immediately before an expansion, the potential has built up to the number of items in the table, and therefore it can pay for moving all the items to the new table. Afterwards, the potential drops to 0, but it is immediately increased by 2 upon inserting the item that caused the expansion.

Figure 17.3 plots the values of num_i , $size_i$, and Φ_i against i . Notice how the potential builds up to pay for expanding the table.

17.4.2 Table expansion and contraction

To implement a TABLE-DELETE operation, it is simple enough to remove the specified item from the table. In order to limit the amount of wasted space, however, we might wish to **contract** the table when the load factor becomes too small. Table contraction is analogous to table expansion: when the number of items in the table drops too low, we allocate a new, smaller table and then copy the items from the old table into the new one. We can then free the storage for the old table by returning it to the memory-management system. Ideally, we would like to preserve two properties:

- the load factor of the dynamic table is bounded below by a positive constant, and
- the amortized cost of a table operation is bounded above by a constant.

We assume that we measure the cost in terms of elementary insertions and deletions.

You might think that we should double the table size upon inserting an item into a full table and halve the size when deleting an item would cause the table to become less than half full. This strategy would guarantee that the load factor of the table never drops below $1/2$, but unfortunately, it can cause the amortized cost of an operation to be quite large. Consider the following scenario. We perform n operations on a table T , where n is an exact power of 2. The first $n/2$ operations are insertions, which by our previous analysis cost a total of $\Theta(n)$. At the end of this sequence of insertions, $T.num = T.size = n/2$. For the second $n/2$ operations, we perform the following sequence:

insert, delete, delete, insert, insert, delete, delete, insert, insert, . . .

The first insertion causes the table to expand to size n . The two following deletions cause the table to contract back to size $n/2$. Two further insertions cause another expansion, and so forth. The cost of each expansion and contraction is $\Theta(n)$, and there are $\Theta(n)$ of them. Thus, the total cost of the n operations is $\Theta(n^2)$, making the amortized cost of an operation $\Theta(n)$.

The downside of this strategy is obvious: after expanding the table, we do not delete enough items to pay for a contraction. Likewise, after contracting the table, we do not insert enough items to pay for an expansion.

We can improve upon this strategy by allowing the load factor of the table to drop below $1/2$. Specifically, we continue to double the table size upon inserting an item into a full table, but we halve the table size when deleting an item causes the table to become less than $1/4$ full, rather than $1/2$ full as before. The load factor of the table is therefore bounded below by the constant $1/4$.

Intuitively, we would consider a load factor of $1/2$ to be ideal, and the table's potential would then be 0. As the load factor deviates from $1/2$, the potential increases so that by the time we expand or contract the table, the table has garnered sufficient potential to pay for copying all the items into the newly allocated table. Thus, we will need a potential function that has grown to $T.num$ by the time that the load factor has either increased to 1 or decreased to $1/4$. After either expanding or contracting the table, the load factor goes back to $1/2$ and the table's potential reduces back to 0.

We omit the code for TABLE-DELETE, since it is analogous to TABLE-INSERT. For our analysis, we shall assume that whenever the number of items in the table drops to 0, we free the storage for the table. That is, if $T.num = 0$, then $T.size = 0$.

We can now use the potential method to analyze the cost of a sequence of n TABLE-INSERT and TABLE-DELETE operations. We start by defining a potential function Φ that is 0 immediately after an expansion or contraction and builds as the load factor increases to 1 or decreases to $1/4$. Let us denote the load fac-

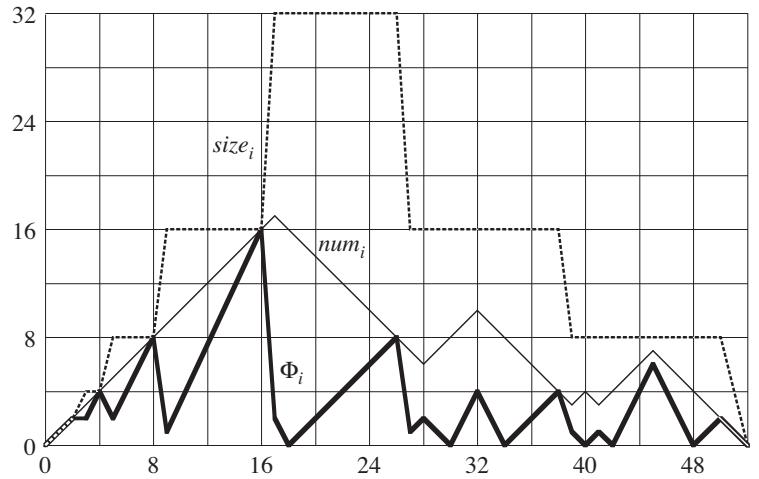


Figure 17.4 The effect of a sequence of n TABLE-INSERT and TABLE-DELETE operations on the number num_i of items in the table, the number $size_i$ of slots in the table, and the potential

$$\Phi_i = \begin{cases} 2 \cdot num_i - size_i & \text{if } \alpha_i \geq 1/2, \\ size_i/2 - num_i & \text{if } \alpha_i < 1/2, \end{cases}$$

each measured after the i th operation. The thin line shows num_i , the dashed line shows $size_i$, and the thick line shows Φ_i . Notice that immediately before an expansion, the potential has built up to the number of items in the table, and therefore it can pay for moving all the items to the new table. Likewise, immediately before a contraction, the potential has built up to the number of items in the table.

tor of a nonempty table T by $\alpha(T) = T.num/T.size$. Since for an empty table, $T.num = T.size = 0$ and $\alpha(T) = 1$, we always have $T.num = \alpha(T) \cdot T.size$, whether the table is empty or not. We shall use as our potential function

$$\Phi(T) = \begin{cases} 2 \cdot T.num - T.size & \text{if } \alpha(T) \geq 1/2, \\ T.size/2 - T.num & \text{if } \alpha(T) < 1/2. \end{cases} \quad (17.6)$$

Observe that the potential of an empty table is 0 and that the potential is never negative. Thus, the total amortized cost of a sequence of operations with respect to Φ provides an upper bound on the actual cost of the sequence.

Before proceeding with a precise analysis, we pause to observe some properties of the potential function, as illustrated in Figure 17.4. Notice that when the load factor is $1/2$, the potential is 0. When the load factor is 1, we have $T.size = T.num$, which implies $\Phi(T) = T.num$, and thus the potential can pay for an expansion if an item is inserted. When the load factor is $1/4$, we have $T.size = 4 \cdot T.num$, which

implies $\Phi(T) = T.num$, and thus the potential can pay for a contraction if an item is deleted.

To analyze a sequence of n TABLE-INSERT and TABLE-DELETE operations, we let c_i denote the actual cost of the i th operation, \hat{c}_i denote its amortized cost with respect to Φ , num_i denote the number of items stored in the table after the i th operation, $size_i$ denote the total size of the table after the i th operation, α_i denote the load factor of the table after the i th operation, and Φ_i denote the potential after the i th operation. Initially, $num_0 = 0$, $size_0 = 0$, $\alpha_0 = 1$, and $\Phi_0 = 0$.

We start with the case in which the i th operation is TABLE-INSERT. The analysis is identical to that for table expansion in Section 17.4.1 if $\alpha_{i-1} \geq 1/2$. Whether the table expands or not, the amortized cost \hat{c}_i of the operation is at most 3. If $\alpha_{i-1} < 1/2$, the table cannot expand as a result of the operation, since the table expands only when $\alpha_{i-1} = 1$. If $\alpha_i < 1/2$ as well, then the amortized cost of the i th operation is

$$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) \\ &= 1 + (size_i/2 - num_i) - (size_i/2 - (num_i - 1)) \\ &= 0.\end{aligned}$$

If $\alpha_{i-1} < 1/2$ but $\alpha_i \geq 1/2$, then

$$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (2 \cdot num_i - size_i) - (size_{i-1}/2 - num_{i-1}) \\ &= 1 + (2(num_{i-1} + 1) - size_{i-1}) - (size_{i-1}/2 - num_{i-1}) \\ &= 3 \cdot num_{i-1} - \frac{3}{2}size_{i-1} + 3 \\ &= 3\alpha_{i-1}size_{i-1} - \frac{3}{2}size_{i-1} + 3 \\ &< \frac{3}{2}size_{i-1} - \frac{3}{2}size_{i-1} + 3 \\ &= 3.\end{aligned}$$

Thus, the amortized cost of a TABLE-INSERT operation is at most 3.

We now turn to the case in which the i th operation is TABLE-DELETE. In this case, $num_i = num_{i-1} - 1$. If $\alpha_{i-1} < 1/2$, then we must consider whether the operation causes the table to contract. If it does not, then $size_i = size_{i-1}$ and the amortized cost of the operation is

$$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) \\ &= 1 + (size_i/2 - num_i) - (size_i/2 - (num_i + 1)) \\ &= 2.\end{aligned}$$

If $\alpha_{i-1} < 1/2$ and the i th operation does trigger a contraction, then the actual cost of the operation is $c_i = \text{num}_i + 1$, since we delete one item and move num_i items. We have $\text{size}_i/2 = \text{size}_{i-1}/4 = \text{num}_{i-1} = \text{num}_i + 1$, and the amortized cost of the operation is

$$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= (\text{num}_i + 1) + (\text{size}_i/2 - \text{num}_i) - (\text{size}_{i-1}/2 - \text{num}_{i-1}) \\ &= (\text{num}_i + 1) + ((\text{num}_i + 1) - \text{num}_i) - ((2 \cdot \text{num}_i + 2) - (\text{num}_i + 1)) \\ &= 1.\end{aligned}$$

When the i th operation is a TABLE-DELETE and $\alpha_{i-1} \geq 1/2$, the amortized cost is also bounded above by a constant. We leave the analysis as Exercise 17.4-2.

In summary, since the amortized cost of each operation is bounded above by a constant, the actual time for any sequence of n operations on a dynamic table is $O(n)$.

Exercises

17.4-1

Suppose that we wish to implement a dynamic, open-address hash table. Why might we consider the table to be full when its load factor reaches some value α that is strictly less than 1? Describe briefly how to make insertion into a dynamic, open-address hash table run in such a way that the expected value of the amortized cost per insertion is $O(1)$. Why is the expected value of the actual cost per insertion not necessarily $O(1)$ for all insertions?

17.4-2

Show that if $\alpha_{i-1} \geq 1/2$ and the i th operation on a dynamic table is TABLE-DELETE, then the amortized cost of the operation with respect to the potential function (17.6) is bounded above by a constant.

17.4-3

Suppose that instead of contracting a table by halving its size when its load factor drops below $1/4$, we contract it by multiplying its size by $2/3$ when its load factor drops below $1/3$. Using the potential function

$$\Phi(T) = |2 \cdot T.\text{num} - T.\text{size}|,$$

show that the amortized cost of a TABLE-DELETE that uses this strategy is bounded above by a constant.

Problems

17-1 Bit-reversed binary counter

Chapter 30 examines an important algorithm called the fast Fourier transform, or FFT. The first step of the FFT algorithm performs a **bit-reversal permutation** on an input array $A[0 \dots n-1]$ whose length is $n = 2^k$ for some nonnegative integer k . This permutation swaps elements whose indices have binary representations that are the reverse of each other.

We can express each index a as a k -bit sequence $\langle a_{k-1}, a_{k-2}, \dots, a_0 \rangle$, where $a = \sum_{i=0}^{k-1} a_i 2^i$. We define

$$\text{rev}_k(\langle a_{k-1}, a_{k-2}, \dots, a_0 \rangle) = \langle a_0, a_1, \dots, a_{k-1} \rangle;$$

thus,

$$\text{rev}_k(a) = \sum_{i=0}^{k-1} a_{k-i-1} 2^i.$$

For example, if $n = 16$ (or, equivalently, $k = 4$), then $\text{rev}_k(3) = 12$, since the 4-bit representation of 3 is 0011, which when reversed gives 1100, the 4-bit representation of 12.

- a. Given a function rev_k that runs in $\Theta(k)$ time, write an algorithm to perform the bit-reversal permutation on an array of length $n = 2^k$ in $O(nk)$ time.

We can use an algorithm based on an amortized analysis to improve the running time of the bit-reversal permutation. We maintain a “bit-reversed counter” and a procedure **BIT-REVERSED-INCREMENT** that, when given a bit-reversed-counter value a , produces $\text{rev}_k(\text{rev}_k(a) + 1)$. If $k = 4$, for example, and the bit-reversed counter starts at 0, then successive calls to **BIT-REVERSED-INCREMENT** produce the sequence

0000, 1000, 0100, 1100, 0010, 1010, $\dots = 0, 8, 4, 12, 2, 10, \dots$.

- b. Assume that the words in your computer store k -bit values and that in unit time, your computer can manipulate the binary values with operations such as shifting left or right by arbitrary amounts, bitwise-AND, bitwise-OR, etc. Describe an implementation of the **BIT-REVERSED-INCREMENT** procedure that allows the bit-reversal permutation on an n -element array to be performed in a total of $O(n)$ time.
- c. Suppose that you can shift a word left or right by only one bit in unit time. Is it still possible to implement an $O(n)$ -time bit-reversal permutation?

17-2 Making binary search dynamic

Binary search of a sorted array takes logarithmic search time, but the time to insert a new element is linear in the size of the array. We can improve the time for insertion by keeping several sorted arrays.

Specifically, suppose that we wish to support SEARCH and INSERT on a set of n elements. Let $k = \lceil \lg(n + 1) \rceil$, and let the binary representation of n be $\langle n_{k-1}, n_{k-2}, \dots, n_0 \rangle$. We have k sorted arrays A_0, A_1, \dots, A_{k-1} , where for $i = 0, 1, \dots, k - 1$, the length of array A_i is 2^i . Each array is either full or empty, depending on whether $n_i = 1$ or $n_i = 0$, respectively. The total number of elements held in all k arrays is therefore $\sum_{i=0}^{k-1} n_i 2^i = n$. Although each individual array is sorted, elements in different arrays bear no particular relationship to each other.

- a. Describe how to perform the SEARCH operation for this data structure. Analyze its worst-case running time.
- b. Describe how to perform the INSERT operation. Analyze its worst-case and amortized running times.
- c. Discuss how to implement DELETE.

17-3 Amortized weight-balanced trees

Consider an ordinary binary search tree augmented by adding to each node x the attribute $x.size$ giving the number of keys stored in the subtree rooted at x . Let α be a constant in the range $1/2 \leq \alpha < 1$. We say that a given node x is **α -balanced** if $x.left.size \leq \alpha \cdot x.size$ and $x.right.size \leq \alpha \cdot x.size$. The tree as a whole is **α -balanced** if every node in the tree is α -balanced. The following amortized approach to maintaining weight-balanced trees was suggested by G. Varghese.

- a. A $1/2$ -balanced tree is, in a sense, as balanced as it can be. Given a node x in an arbitrary binary search tree, show how to rebuild the subtree rooted at x so that it becomes $1/2$ -balanced. Your algorithm should run in time $\Theta(x.size)$, and it can use $O(x.size)$ auxiliary storage.
- b. Show that performing a search in an n -node α -balanced binary search tree takes $O(\lg n)$ worst-case time.

For the remainder of this problem, assume that the constant α is strictly greater than $1/2$. Suppose that we implement INSERT and DELETE as usual for an n -node binary search tree, except that after every such operation, if any node in the tree is no longer α -balanced, then we “rebuild” the subtree rooted at the highest such node in the tree so that it becomes $1/2$ -balanced.

We shall analyze this rebuilding scheme using the potential method. For a node x in a binary search tree T , we define

$$\Delta(x) = |x.\text{left.size} - x.\text{right.size}| ,$$

and we define the potential of T as

$$\Phi(T) = c \sum_{x \in T: \Delta(x) \geq 2} \Delta(x) ,$$

where c is a sufficiently large constant that depends on α .

- c.* Argue that any binary search tree has nonnegative potential and that a $1/2$ -balanced tree has potential 0.
- d.* Suppose that m units of potential can pay for rebuilding an m -node subtree. How large must c be in terms of α in order for it to take $O(1)$ amortized time to rebuild a subtree that is not α -balanced?
- e.* Show that inserting a node into or deleting a node from an n -node α -balanced tree costs $O(\lg n)$ amortized time.

17-4 The cost of restructuring red-black trees

There are four basic operations on red-black trees that perform **structural modifications**: node insertions, node deletions, rotations, and color changes. We have seen that RB-INSERT and RB-DELETE use only $O(1)$ rotations, node insertions, and node deletions to maintain the red-black properties, but they may make many more color changes.

- a.* Describe a legal red-black tree with n nodes such that calling RB-INSERT to add the $(n + 1)$ st node causes $\Omega(\lg n)$ color changes. Then describe a legal red-black tree with n nodes for which calling RB-DELETE on a particular node causes $\Omega(\lg n)$ color changes.

Although the worst-case number of color changes per operation can be logarithmic, we shall prove that any sequence of m RB-INSERT and RB-DELETE operations on an initially empty red-black tree causes $O(m)$ structural modifications in the worst case. Note that we count each color change as a structural modification.

- b.* Some of the cases handled by the main loop of the code of both RB-INSERT-FIXUP and RB-DELETE-FIXUP are **terminating**: once encountered, they cause the loop to terminate after a constant number of additional operations. For each of the cases of RB-INSERT-FIXUP and RB-DELETE-FIXUP, specify which are terminating and which are not. (*Hint*: Look at Figures 13.5, 13.6, and 13.7.)

We shall first analyze the structural modifications when only insertions are performed. Let T be a red-black tree, and define $\Phi(T)$ to be the number of red nodes in T . Assume that 1 unit of potential can pay for the structural modifications performed by any of the three cases of RB-INSERT-FIXUP.

- c. Let T' be the result of applying Case 1 of RB-INSERT-FIXUP to T . Argue that $\Phi(T') = \Phi(T) - 1$.
- d. When we insert a node into a red-black tree using RB-INSERT, we can break the operation into three parts. List the structural modifications and potential changes resulting from lines 1–16 of RB-INSERT, from nonterminating cases of RB-INSERT-FIXUP, and from terminating cases of RB-INSERT-FIXUP.
- e. Using part (d), argue that the amortized number of structural modifications performed by any call of RB-INSERT is $O(1)$.

We now wish to prove that there are $O(m)$ structural modifications when there are both insertions and deletions. Let us define, for each node x ,

$$w(x) = \begin{cases} 0 & \text{if } x \text{ is red ,} \\ 1 & \text{if } x \text{ is black and has no red children ,} \\ 0 & \text{if } x \text{ is black and has one red child ,} \\ 2 & \text{if } x \text{ is black and has two red children .} \end{cases}$$

Now we redefine the potential of a red-black tree T as

$$\Phi(T) = \sum_{x \in T} w(x) ,$$

and let T' be the tree that results from applying any nonterminating case of RB-INSERT-FIXUP or RB-DELETE-FIXUP to T .

- f. Show that $\Phi(T') \leq \Phi(T) - 1$ for all nonterminating cases of RB-INSERT-FIXUP. Argue that the amortized number of structural modifications performed by any call of RB-INSERT-FIXUP is $O(1)$.
- g. Show that $\Phi(T') \leq \Phi(T) - 1$ for all nonterminating cases of RB-DELETE-FIXUP. Argue that the amortized number of structural modifications performed by any call of RB-DELETE-FIXUP is $O(1)$.
- h. Complete the proof that in the worst case, any sequence of m RB-INSERT and RB-DELETE operations performs $O(m)$ structural modifications.

17-5 Competitive analysis of self-organizing lists with move-to-front

A *self-organizing list* is a linked list of n elements, in which each element has a unique key. When we search for an element in the list, we are given a key, and we want to find an element with that key.

A self-organizing list has two important properties:

1. To find an element in the list, given its key, we must traverse the list from the beginning until we encounter the element with the given key. If that element is the k th element from the start of the list, then the cost to find the element is k .
2. We may reorder the list elements after any operation, according to a given rule with a given cost. We may choose any heuristic we like to decide how to reorder the list.

Assume that we start with a given list of n elements, and we are given an access sequence $\sigma = \langle \sigma_1, \sigma_2, \dots, \sigma_m \rangle$ of keys to find, in order. The cost of the sequence is the sum of the costs of the individual accesses in the sequence.

Out of the various possible ways to reorder the list after an operation, this problem focuses on transposing adjacent list elements—switching their positions in the list—with a unit cost for each transpose operation. You will show, by means of a potential function, that a particular heuristic for reordering the list, move-to-front, entails a total cost no worse than 4 times that of any other heuristic for maintaining the list order—even if the other heuristic knows the access sequence in advance! We call this type of analysis a *competitive analysis*.

For a heuristic H and a given initial ordering of the list, denote the access cost of sequence σ by $C_H(\sigma)$. Let m be the number of accesses in σ .

- a. Argue that if heuristic H does not know the access sequence in advance, then the worst-case cost for H on an access sequence σ is $C_H(\sigma) = \Omega(mn)$.

With the *move-to-front* heuristic, immediately after searching for an element x , we move x to the first position on the list (i.e., the front of the list).

Let $\text{rank}_L(x)$ denote the rank of element x in list L , that is, the position of x in list L . For example, if x is the fourth element in L , then $\text{rank}_L(x) = 4$. Let c_i denote the cost of access σ_i using the move-to-front heuristic, which includes the cost of finding the element in the list and the cost of moving it to the front of the list by a series of transpositions of adjacent list elements.

- b. Show that if σ_i accesses element x in list L using the move-to-front heuristic, then $c_i = 2 \cdot \text{rank}_L(x) - 1$.

Now we compare move-to-front with any other heuristic H that processes an access sequence according to the two properties above. Heuristic H may transpose

elements in the list in any way it wants, and it might even know the entire access sequence in advance.

Let L_i be the list after access σ_i using move-to-front, and let L_i^* be the list after access σ_i using heuristic H. We denote the cost of access σ_i by c_i for move-to-front and by c_i^* for heuristic H. Suppose that heuristic H performs t_i^* transpositions during access σ_i .

c. In part (b), you showed that $c_i = 2 \cdot \text{rank}_{L_{i-1}}(x) - 1$. Now show that $c_i^* = \text{rank}_{L_{i-1}^*}(x) + t_i^*$.

We define an ***inversion*** in list L_i as a pair of elements y and z such that y precedes z in L_i and z precedes y in list L_i^* . Suppose that list L_i has q_i inversions after processing the access sequence $\langle \sigma_1, \sigma_2, \dots, \sigma_i \rangle$. Then, we define a potential function Φ that maps L_i to a real number by $\Phi(L_i) = 2q_i$. For example, if L_i has the elements $\langle e, c, a, d, b \rangle$ and L_i^* has the elements $\langle c, a, b, d, e \rangle$, then L_i has 5 inversions $((e, c), (e, a), (e, d), (e, b), (d, b))$, and so $\Phi(L_i) = 10$. Observe that $\Phi(L_i) \geq 0$ for all i and that, if move-to-front and heuristic H start with the same list L_0 , then $\Phi(L_0) = 0$.

d. Argue that a transposition either increases the potential by 2 or decreases the potential by 2.

Suppose that access σ_i finds the element x . To understand how the potential changes due to σ_i , let us partition the elements other than x into four sets, depending on where they are in the lists just before the i th access:

- Set A consists of elements that precede x in both L_{i-1} and L_{i-1}^* .
 - Set B consists of elements that precede x in L_{i-1} and follow x in L_{i-1}^* .
 - Set C consists of elements that follow x in L_{i-1} and precede x in L_{i-1}^* .
 - Set D consists of elements that follow x in both L_{i-1} and L_{i-1}^* .
- e. Argue that $\text{rank}_{L_{i-1}}(x) = |A| + |B| + 1$ and $\text{rank}_{L_{i-1}^*}(x) = |A| + |C| + 1$.

f. Show that access σ_i causes a change in potential of

$$\Phi(L_i) - \Phi(L_{i-1}) \leq 2(|A| - |B| + t_i^*),$$

where, as before, heuristic H performs t_i^* transpositions during access σ_i .

Define the amortized cost \hat{c}_i of access σ_i by $\hat{c}_i = c_i + \Phi(L_i) - \Phi(L_{i-1})$.

g. Show that the amortized cost \hat{c}_i of access σ_i is bounded from above by $4c_i^*$.

h. Conclude that the cost $C_{\text{MTF}}(\sigma)$ of access sequence σ with move-to-front is at most 4 times the cost $C_H(\sigma)$ of σ with any other heuristic H, assuming that both heuristics start with the same list.

Chapter notes

Aho, Hopcroft, and Ullman [5] used aggregate analysis to determine the running time of operations on a disjoint-set forest; we shall analyze this data structure using the potential method in Chapter 21. Tarjan [331] surveys the accounting and potential methods of amortized analysis and presents several applications. He attributes the accounting method to several authors, including M. R. Brown, R. E. Tarjan, S. Huddleston, and K. Mehlhorn. He attributes the potential method to D. D. Sleator. The term “amortized” is due to D. D. Sleator and R. E. Tarjan.

Potential functions are also useful for proving lower bounds for certain types of problems. For each configuration of the problem, we define a potential function that maps the configuration to a real number. Then we determine the potential Φ_{init} of the initial configuration, the potential Φ_{final} of the final configuration, and the maximum change in potential $\Delta\Phi_{\text{max}}$ due to any step. The number of steps must therefore be at least $|\Phi_{\text{final}} - \Phi_{\text{init}}| / |\Delta\Phi_{\text{max}}|$. Examples of potential functions to prove lower bounds in I/O complexity appear in works by Cormen, Sundquist, and Wisniewski [79]; Floyd [107]; and Aggarwal and Vitter [3]. Krumme, Cybenko, and Venkataraman [221] applied potential functions to prove lower bounds on *gossiping*: communicating a unique item from each vertex in a graph to every other vertex.

The move-to-front heuristic from Problem 17-5 works quite well in practice. Moreover, if we recognize that when we find an element, we can splice it out of its position in the list and relocate it to the front of the list in constant time, we can show that the cost of move-to-front is at most twice the cost of any other heuristic including, again, one that knows the entire access sequence in advance.