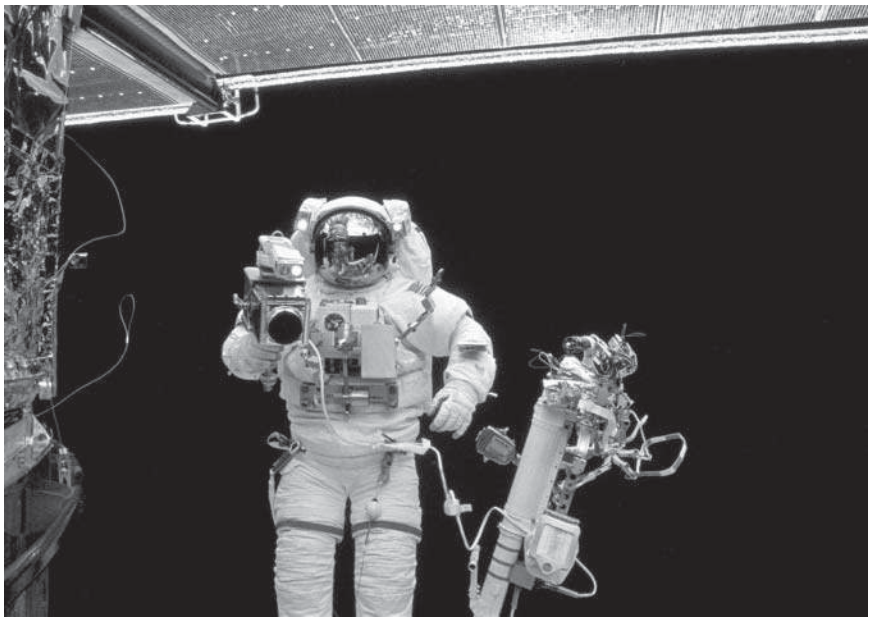
Chapter 2 Basic Data Structures



An astronaut recording a video of a Hubble Space Telescope servicing mission in 1997. U.S. government image. NASA.

Contents

The Internet is designed to route information in discrete packets, which are at most 1500 bytes in length. Because of this design, any time a video stream is transmitted on the Internet, it must be subdivided into packets and these packets must each be individually routed to their destination. In the case of a stored video file, it is desirable to send packets using the Transmission Control Protocol (TCP), since this protocol guarantees that all the packets will arrive at their destination in the correct order. Nevertheless, because of vagaries and errors, the time it takes for these packets to arrive at their destination can be highly variable. Thus, we need a way of "smoothing out" these variations in order to avoid long pauses if someone wants to watch a video at the same time it is being transmitted.

This smoothing is typically achieved is by using a ***buffer***, which is a portion of computer memory that is used to temporarily store items, as they are being produced by one computational process and consumed by another. For example, in this case of video packets arriving via TCP, the networking process is producing the packets and the playback process is consuming them. Thus, ignoring any rewinding, this ***producer-consumer model*** is enforcing a first-in, first-out (FIFO) protocol for the packets, in that the consumer process is always retrieving the packet that has been in the buffer the longest. (See Figure 2.1.)

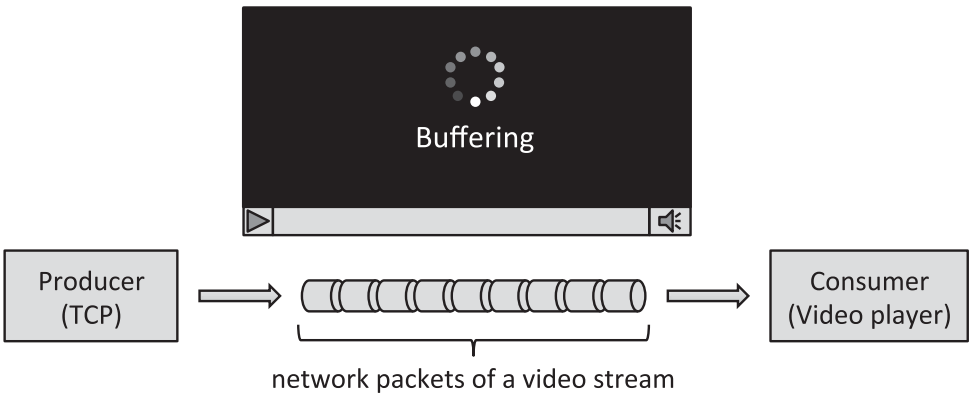


Figure 2.1: Using a buffer to smooth video transmission over the Internet.

Network packets are produced by the TCP process, which inserts packets in the correct order into the buffer, but does so at a variable rate. Packets are consumed by the video player process, which removes packets from the buffer at a constant speed according to the video standard it is using. The buffer enforces a first-in, first-out (FIFO) protocol and, if it is big enough, it should smooth out the packet production and consumption so that packets can be processed at the average transmission rate without annoying pauses. In this chapter, we explore how to implement such a buffer using a ***queue*** data structure, which itself can be implemented with either an array or a linked list. We also study several other basic data structures, including stacks, lists, and trees, along with applications of these structures.

2.1 Stacks and Queues

2.1.1 Stacks

A ***stack*** is a container of objects that are inserted and removed according to the ***last-in first-out*** (***LIFO***) principle. Objects can be inserted into a stack at any time, but only the most-recently inserted (that is, "last") object can be removed at any time. The name "stack" is derived from the metaphor of a stack of plates in a spring-loaded cafeteria plate dispenser. In this case, the fundamental operations involve the "pushing" and "popping" of plates on the stack.

Example 2.1: *Internet web browsers store the addresses of recently visited sites on a stack. Each time a user visits a new site, that site's address is "pushed" onto the stack of addresses. The browser then allows the user to "pop" back to previously visited sites using the "back" button.*

Viewed abstractly, a ***stack***, S, is a container that supports the following two methods:

push(o): Insert object o at the top of the stack.

pop(): Remove from the stack and return the top object on the stack, that is, the most recently inserted element still in the stack; an error occurs if the stack is empty.

A Simple Array-Based Implementation

A stack is easily implemented with an N-element array S, with elements stored from $S[0]$ to $S[t]$, where t is an integer that gives the index of the top element in S. Note that one of the important details of such an implementation is that we must specify some maximum size N for our stack, say, $N = 1,000$. (See Figure 2.2.)

Figure 2.2: Implementing a stack with an array S. The top element is in cell $S[t]$.

If we use the convention that arrays begin at index 0, then we would initialize t to -1 (for an initially empty stack), and we can use this value to also test if a stack is empty. In addition, we can use this variable to determine the number of elements in a stack ($t+1$). In this array-based implementation of a stack, we also must signal an error condition that arises if we try to insert a new element and the array S is full. Given this error condition, we can then implement the main methods of a stack as described in Algorithm 2.3.

Algorithm push(o):
    if $t + 1 = N$ **then**
        return that a stack-full error has occurred
    $t \leftarrow t + 1$
    $S[t] \leftarrow o$
    return

Algorithm pop():
    if $t < 0$ **then**
        return that a stack-empty error has occurred
    $e \leftarrow S[t]$
    $S[t] \leftarrow$ **null**
    $t \leftarrow t - 1$
    return e

Algorithm 2.3: Implementing a stack with an array.

Turning to the analysis of this array-based implementation of a stack, it should be clear that each of the stack methods, push and pop, runs in a constant amount of time. This is because they only involve a constant number of simple arithmetic operations, comparisons, and assignment statements. That is, in this array-based implementation of the stack, each method runs in $O(1)$ time.

The array implementation of a stack is both simple and efficient, and is widely used in a variety of computing applications. Nevertheless, this implementation has one negative aspect; it must assume a fixed upper bound N on the ultimate size of the stack. An application may actually need much less space than this, in which case we would be wasting memory. Alternatively, an application may need more space than this, in which case our stack implementation may "crash" the application with an error as soon as it tries to push its $(N + 1)$st object on the stack. Thus, even with its simplicity and efficiency, the array-based stack implementation is not always ideal.

Fortunately, there is another implementation, which is to use a linked list, discussed later in this chapter. Such an implementation would not have a size limitation (other than that imposed by the total amount of memory on our computer) and it would use an amount of space proportional to the actual number of elements stored in the stack. Alternatively, we could also use an extendable table, as discussed in Section 1.4.2, which could grow in size as the stack grows. In cases where we have a good estimate on the number of items needing to go in the stack, however, the array-based implementation is hard to beat, because they achieve $O(1)$-time performance for the push and pop operations. Stacks serve a vital role in a number of computing applications, so it is helpful to have a fast stack implementation, such as the simple array-based implementation.

Using Stacks for Procedure Calls and Recursion

Stacks have an important application in the runtime environments of programming languages, such as C, C++, Java, and Python. Each thread in a running program written in one of these languages has a private stack, called the ***method stack***, which is used to keep track of local variables and other important information on methods, as they are invoked during execution. (See Figure 2.4.)

More specifically, during the execution of a program thread, the runtime environment maintains a stack whose elements are descriptors of the currently active (that is, nonterminated) invocations of methods. These descriptors are called ***frames***. A frame for some invocation of method cool stores the current values of the local variables and parameters of method cool, as well as information on the method that called cool and on what needs to be returned to this method.

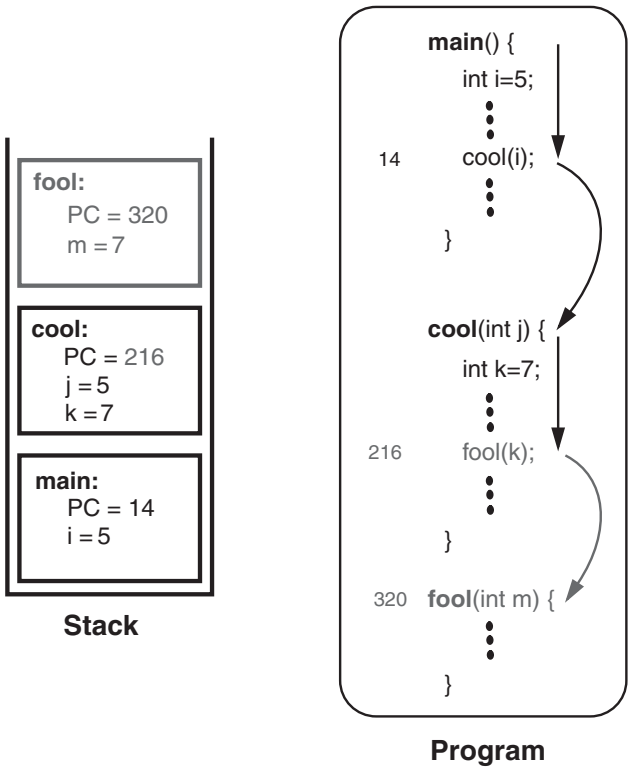


Figure 2.4: An example of a method stack: Method fool has just been called by method cool, which itself was previously called by method main. Note the values of the program counter, parameters, and local variables stored in the stack frames. When the invocation of method fool terminates, the invocation of method cool will resume its execution at instruction 217, which is obtained by incrementing the value of the program counter stored in the stack frame.

The runtime environment keeps the address of the statement the thread is currently executing in the program in a special register, called the ***program counter***. When a method, cool, invokes another method, fool, the current value of the program counter is recorded in the frame of the current invocation of cool (so the computer will "know" where to return to when method fool is done).

At the top of the method stack is the frame of the ***running method***—that is, the method that currently has control of the execution. The remaining elements of the stack are frames of the ***suspended methods***—that is, methods that have invoked another method and are currently waiting for it to return control to them upon its termination. The order of the elements in the stack corresponds to the chain of invocations of the currently active methods. When a new method is invoked, a frame for this method is pushed onto the stack. When it terminates, its frame is popped from the stack and the computer resumes the processing of the previously suspended method.

The method stack also performs parameter passing to methods. Specifically, many languages, such as C and Java, use the ***call-by-value*** parameter passing protocol using the method stack. This means that the current ***value*** of a variable (or expression) is what is passed as an argument to a called method. In the case of a variable x of a primitive type, such as an int or float, the current value of x is simply the number that is associated with x. When such a value is passed to the called method, it is assigned to a local variable in the called method's frame. (This simple assignment is also illustrated in Figure 2.4.) Note that if the called method changes the value of this local variable, it will ***not*** change the value of the variable in the calling method.

Recursion

One of the benefits of using a stack to implement method invocation is that it allows programs to use ***recursion*** (Section 1.1.4). That is, it allows a method to call itself as a subroutine.

Recall that in using this technique correctly, we must always design a recursive method so that it is guaranteed to terminate at some point (for example, by always making recursive calls for "smaller" instances of the problem and handling the "smallest" instances nonrecursively as special cases). We note that if we design an "infinitely recursive" method, it will not actually run forever. It will instead, at some point, use up all the memory available for the method stack and generate an out-of-memory or stack-overflow error. If we use recursion with care, however, the method stack will implement recursive methods without any trouble. Each call of the same method will be associated with a different frame, complete with its own values for local variables. Recursion can be very powerful, as it often allows us to design simple and efficient programs for fairly difficult problems.

2.1.2 Queues

Another basic data structure is the ***queue***. It is a close "cousin" of the stack, as a queue is a container of objects that are inserted and removed according to the ***first-in first-out*** (***FIFO***) principle, as in the video buffering application discussed at the beginning of this chapter. That is, elements can be inserted at any time, but only the element that has been in the queue the longest can be removed at any time. We usually say that elements enter the queue at the ***rear*** and are removed from the ***front***.

Queue Definition

Viewed abstractly, a queue keeps objects in a sequence, where element access and deletion are restricted to the first element in the sequence, which is called the ***front*** of the queue, and element insertion is restricted to the end of the sequence, which is called the ***rear*** of the queue. Thus, we enforce the rule that items are inserted and removed according to the FIFO principle. A ***queue*** supports the following two fundamental methods:

enqueue(o): Insert object o at the rear of the queue.

dequeue(): Remove and return from the queue the object at the front; an error occurs if the queue is empty.

A Simple Array-Based Implementation

As with a stack, we can implement a queue with an array, but the details are slightly more complicated. In this case, we use an array, Q, with capacity N, for storing its elements. Since the main rule for a queue is that we insert and delete objects according to the FIFO principle, we must decide how we are going to keep track of the front and rear of the queue.

To avoid moving objects once they are placed in Q, we define two variables f and r, which have the following meanings:

- f is an index to the cell of Q storing the first element of the queue (which is the next candidate to be removed by a dequeue operation), unless the queue is empty (in which case $f = r$).
- r is an index to the next available array cell in Q.

Initially, we assign $f = r = 0$, and we indicate that the queue is empty by the condition $f = r$. Now, when we remove an element from the front of the queue, we can simply increment f to index the next cell. Likewise, when we add an element, we can simply increment r to index the next available cell in Q. We have to be a little careful not to overflow the end of the array, however. Consider, for example, what happens if we repeatedly enqueue and dequeue a single element N different

times. We would have $f = r = N$. If we were then to try to insert the element just one more time, we would get an array-out-of-bounds error (since the N valid locations in Q are from $Q[0]$ to $Q[N-1]$), even though there is plenty of room in the queue in this case. To avoid this problem and be able to utilize all of the array Q, we let the f and r indices "wrap around" the end of Q. That is, we now view Q as a "circular array" that goes from $Q[0]$ to $Q[N-1]$ and then immediately back to $Q[0]$ again. (See Figure 2.5.)

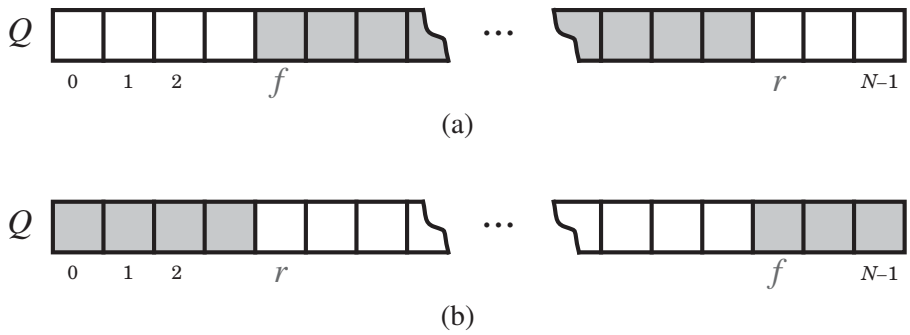


Figure 2.5: Implementing a queue using an array Q in a circular fashion: (a) the "normal" configuration with $f \leq r$; (b) the "wrapped around" configuration with $r < f$. The cells storing queue elements are highlighted.

Implementing this circular view of Q is pretty easy. Each time we increment f or r, we simply compute this increment as "$(f+1) \bmod N$" or "$(r+1) \bmod N$," respectively. Recall here that the operator "$\bmod$" is the ***modulo*** operator, which is computed by taking the remainder after an integral division, so that, if y is nonzero, then

$$x \bmod y = x - \lfloor x/y \rfloor y.$$

Consider now the situation that occurs if we enqueue N objects without dequeuing them. We would have $f = r$, which is the same condition as when the queue is empty. Hence, we would not be able to tell the difference between a full queue and an empty one in this case. Fortunately, this is not a big problem, and a number of ways for dealing with it exist. For example, we can simply insist that Q can never hold more than $N-1$ objects. The above simple rule for handling a full queue takes care of the final problem with our implementation, and leads to the pseudocoded descriptions of the main queue methods given in Algorithm 2.6. Note that we may compute the size of the queue by means of the expression $(N - f + r) \bmod N$, which gives the correct result both in the "normal" configuration (when $f \leq r$) and in the "wrapped around" configuration (when $r < f$).

Algorithm dequeue():
    if $f = r$ **then**
        return an error condition that the queue is empty
    $temp \leftarrow Q[f]$
    $Q[f] \leftarrow$ **null**
    $f \leftarrow (f + 1) \bmod N$
    return $temp$

Algorithm enqueue(o):
    if $(N - f + r) \bmod N = N - 1$ **then**
        return an error condition that the queue is full
    $Q[r] \leftarrow o$
    $r \leftarrow (r + 1) \bmod N$
    return

Algorithm 2.6: Implementing a queue with an array, which is viewed circularly.

Analysis and Applications of Queues

As with our array-based stack implementation, it should be clear that each of the above queue methods based on implementing the queue with an array executes in a constant number of statements involving arithmetic operations, comparisons, and assignments. Thus, each method in this array-based queue implementation runs in $O(1)$ time.

We have already discussed the application of queues to the problem of buffering video as it is streamed on the Internet. There are several other applications of queues, as well. For instance, a queue is an ideal data structure for processing online ticketing requests. Likewise, a queue is typically used to manage a ***printer spooler***, which is a process that manages documents that are sent to be output by a printer.

As with the array-based stack implementation, the only real disadvantage of the array-based queue implementation is that we artificially set the capacity of the queue to be some number N. In a real application, we may actually need more or less queue capacity than this, but if we have a good estimate of the number of elements that will be in the queue at the same time, then the array-based implementation is quite efficient. If we don't have a good estimate, however, then we can implement a queue using a linked list, which is a data structure we discuss in the next section.

2.2 Lists

Stacks and queues store elements according to a linear sequence determined by update operations that act on the "ends" of the sequence. Lists, on the other hand, which we discuss in this section, maintain linear orders while allowing for accesses and updates in the "middle."

2.2.1 Index-Based Lists

Suppose we are given a linear sequence, S, that contains n elements. We can uniquely refer to each element e of S using an integer in the range $[0, n-1]$ that is equal to the number of elements of S that precede e in S. In particular, we define the ***index*** (or ***rank***) of an element e in S to be the number of elements that are before e in S. Hence, the first element in a sequence is at index 0 and the last element is at index $n-1$.

Note that this notion of "index" is different from indices of cells in an array, since array cells are static. This notion of index in a list of elements implies that the index of an element can change depending on whether elements before it are inserted or removed in its list. Such an index in a list is therefore dynamic and depends on the operations that are performed on the list. For instance, if we insert a new element at the beginning of such a list, the index of each of the other elements increases by one. This definition is consistent, for example, with the ArrayList class in Java.

We refer to a linear sequence that supports access to its elements by their indices in this way as an ***index-based list***.

An index-based list, S, storing n elements supports the following methods:

$\mathsf{get}(r)$: Return the element of S with index r; an error condition occurs if $r < 0$ or $r > n-1$.

$\mathsf{set}(r, e)$: Replace with e the element at index r and return it; an error condition occurs if $r < 0$ or $r > n-1$.

$\mathsf{add}(r, e)$: Insert a new element e into S to have index r; an error condition occurs if $r < 0$ or $r > n$.

$\mathsf{remove}(r)$: Remove from S the element at index r; an error condition occurs if $r < 0$ or $r > n-1$.

An obvious choice for implementing an index-based list is to use an array A, where $A[i]$ stores (a reference to) the element with index i. We choose the size N of array A to be sufficiently large, and we maintain in an instance variable the actual number $n < N$ of elements in the list. The details of the implementation of the methods of index-based list are reasonably simple. To implement the $\mathsf{get}(r)$

operation, for example, we just return $A[r]$. Implementations of methods add(r, e) and remove(r) are given in Algorithm 2.7.

Algorithm add(r, e):
  if $n = N$ **then**
    return "Array is full."
  if $r < n$ **then**
    for $i \leftarrow n-1, n-2, \ldots, r$ **do**
      $A[i+1] \leftarrow A[i]$ // make room for the new element
  $A[r] \leftarrow e$
  $n \leftarrow n+1$

Algorithm remove(r):
  $e \leftarrow A[r]$ // e is a temporary variable
  if $r < n-1$ **then**
    for $i \leftarrow r, r+1, \ldots, n-2$ **do**
      $A[i] \leftarrow A[i+1]$ // fill in for the removed element
  $n \leftarrow n-1$
  return e

Algorithm 2.7: Methods in an array implementation of an index-based list.

An important (and time-consuming) part of this implementation involves the shifting of elements up or down to keep the occupied cells in the array contiguous. That is, in inserting a new element at rank r we must shift up by 1 the places where all elements previously of index r and higher are stored. Likewise, in removing an element at rank r we must shift down by 1 the places where all elements previously of index $r+1$ and higher are stored. These shifting operations are required to maintain our rule of always storing an element of rank i at index i in A. (See Figure 2.8 and also Exercise C-2.8.)

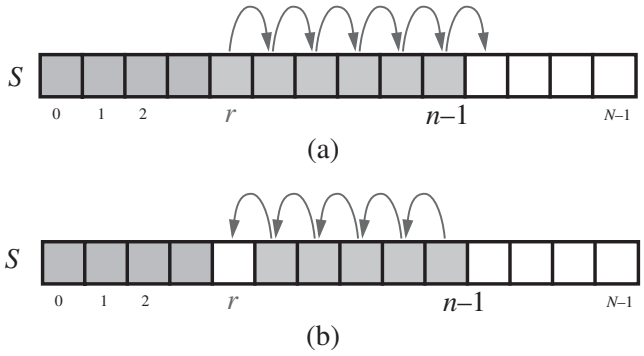


Figure 2.8: Array-based implementation of an index-based list, S: (a) shifting up for an insertion at index r; (b) shifting down for a removal at index r.

Analysis of an Array Implementation of an Index-Based List

Table 2.9 shows the running times of the methods of an index-based list implemented with an array. Note that the insertion and removal methods can take time much longer than $O(1)$. In particular, $\mathsf{add}(r, e)$ runs in time $\Theta(n)$ in the worst case. Indeed, the worst case for this operation occurs when $r = 0$, since all the existing n elements have to be shifted over. A similar argument applies to the method $\mathsf{remove}(r)$, which runs in $O(n)$ time, because we have to shift $n-1$ elements in the worst case ($r = 0$). In fact, assuming that each possible index is equally likely to be passed as an argument to these operations, their average running time is $\Theta(n)$, since we would have to shift $n/2$ elements on average.

Method	Time
$\mathsf{get}(r)$	$O(1)$
$\mathsf{set}(r, e)$	$O(1)$
$\mathsf{add}(r, e)$	$O(n)$
$\mathsf{remove}(r)$	$O(n)$

Table 2.9: Worst-case performance of an index-based list with n elements implemented with an array. The space usage is $O(N)$, where N is the size of the array.

Looking more closely at $\mathsf{add}(r, e)$ and $\mathsf{remove}(r)$, we note that they each run in time $O(n-r+1)$, for only those elements at rank r and higher have to be shifted up or down. Thus, inserting or removing an item at the end of a list, using the methods $\mathsf{add}(n, e)$ and $\mathsf{remove}(n-1)$, respectively take $O(1)$ time each. That is, inserting or removing an element at the end of an index-based list takes constant time, as would inserting or removing an element within a constant number of cells from the end. Still, with the above implementation, inserting or removing an element at the beginning of a list requires shifting every other element by one; hence, it takes $\Theta(n)$ time. Thus, there is an asymmetry to this implementation—updates at the end are fast, whereas updates at the beginning are slow.

Actually, with a little effort, we can produce an array-based implementation of this structure that achieves $O(1)$ time for insertions and removals at index 0, as well as insertions and removals at the end of the list. Achieving this requires that we give up on our rule that an element at list-index i is stored in the array at index i, however, as we would have to use a circular array approach like we used in Section 2.1.2 to implement a queue. We leave the details of this implementation for an exercise (C-2.8). In addition, we note that an index-based list can also be implemented to achieve constant-time amortized insertion and removal operations at the end of a list, by using an extendable table (Section 1.4.2), which, in fact, is the default implementation of the ArrayList class in Java.

2.2.2 Linked Lists

Using an index is not the only way to refer to elements in a list. We could alternatively implement a list S so that each element is stored in a special ***node*** object with references (that is, pointers) to the nodes before and after it in the list. In this case, it could be more natural and efficient to use a node instead of an index to identify where to access and update a list. In this section, we explore such a way of using nodes to represent "places" in a list.

To abstract a way of storing elements in a list, we introduce the concept of ***position*** in a list, which formalizes the intuitive notion of "node" that is storing an element relative to others in the list. In this framework, we view a ***linked list*** as a container of elements that stores each element at a node position and that keeps these positions arranged in a linear order relative to one another. A position is itself an object that supports the following simple method:

element(): Return the element stored at this position.

A position (or node) is always defined ***relatively***, that is, in terms of its neighbors. In a list, a position p will always be "after" some position q and "before" some position s (unless p is the first or last position). A position p, which is associated with some element e in a list S, does not change, even if the rank of e changes in S, unless we explicitly remove e (and, hence, destroy position p). Moreover, the position p does not change even if we replace or swap the element e stored at p with another element. These facts about positions allow us to define a set of position-based list methods that take position objects as parameters and also provide position objects as return values.

Using the concept of position to encapsulate the idea of "node" in a list, we can define a linked list. This structure supports the following methods for a list, S:

first(): Return the position of the first element of S; an error occurs if S is empty.

last(): Return the position of the last element of S; an error occurs if S is empty.

before(p): Return the position of the element of S preceding the one at position p; an error occurs if p is the first position.

after(p): Return the position of the element of S following the one at position p; an error occurs if p is the last position.

The above methods allow us to refer to relative positions in a list, starting at the beginning or end, and to be able to move incrementally up or down the list. As mentioned above, these positions can be thought of as nodes in the list, but note that there are no specific references to pointers or links to previous or next nodes in these methods.

We can also include the following update methods for a linked list.

insertBefore(p, e): Insert a new element e into S before position p in S.

insertAfter(p, e): Insert a new element e into S after position p in S.

remove(p): Remove from S the element at position p.

This approach allows us to view an ordered collection of objects in terms of their places, without worrying about the exact way those places are represented. In addition, this structure, with its built-in notion of position, is useful in a number of settings. For example, a simple text editor embeds the notion of positional insertion and removal, since such editors typically perform all updates relative to a ***cursor***, which represents the current position in a list of characters being edited.

We can use node objects to implement a ***linked list***, so that a great variety of operations, including insertion and removal at various places, can run in $O(1)$ time. A node in a ***singly linked*** list stores in a ***next*** link a reference to the next node in the list. Thus, a singly linked list can only be traversed in one direction—from the head to the tail. A node in a ***doubly linked*** list, on the other hand, stores two references—a ***next*** link, which points to the next node in the list, and a ***prev*** link, which points to the previous node in the list. Therefore, a doubly linked list can be traversed in either direction. Being able to determine the previous and next node from any given node in a list greatly simplifies list implementation; so let us assume we are using such doubly linked nodes to implement a linked list.

To simplify updates and searching, it is convenient to add special nodes at both ends of the list: a ***header*** node just before the head of the list, and a ***trailer*** node just after the tail of the list. These "dummy" or ***sentinel*** nodes do not store any element, but their ubiquitous existence allows us to avoid worrying about special cases for inserting and removing elements at the beginning or end of a list. The header has a valid *next* reference but a null *prev* reference, while the trailer has a valid *prev* reference but a null *next* reference. A doubly linked list with these sentinels is shown in Figure 2.10. Note that a linked list object would simply need to store these two sentinels and a size counter that keeps track of the number of elements (not counting sentinels) in the list.

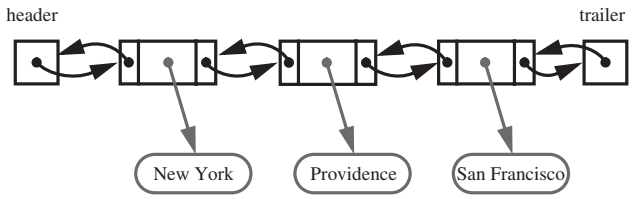


Figure 2.10: A doubly linked list with sentinels, header and trailer, marking the ends of the list. An empty list would have these sentinels pointing to each other.

We can simply make the nodes of the linked list implement the position concept, defining a method element(), which returns the element stored at the node. Thus, the nodes themselves act as positions.

Consider how we might implement the insertAfter(p, e) method, for inserting an element e after position p. We create a new node v to hold the element e, link v into its place in the list, and then update the next and prev references of v's two new neighbors. This method is given in pseudocode in Algorithm 2.11 and is illustrated in Figure 2.12.

Algorithm insertAfter(p, e):
Create a new node v
v.element $\leftarrow e$
v.prev $\leftarrow p$ // link v to its predecessor
v.next $\leftarrow p$.next // link v to its successor
$(p$.next$)$.prev $\leftarrow v$ // link p's old successor to v
p.next $\leftarrow v$ // link p to its new successor, v
return v // the position for the element e

Algorithm 2.11: Inserting an element e after a position p in a linked list.

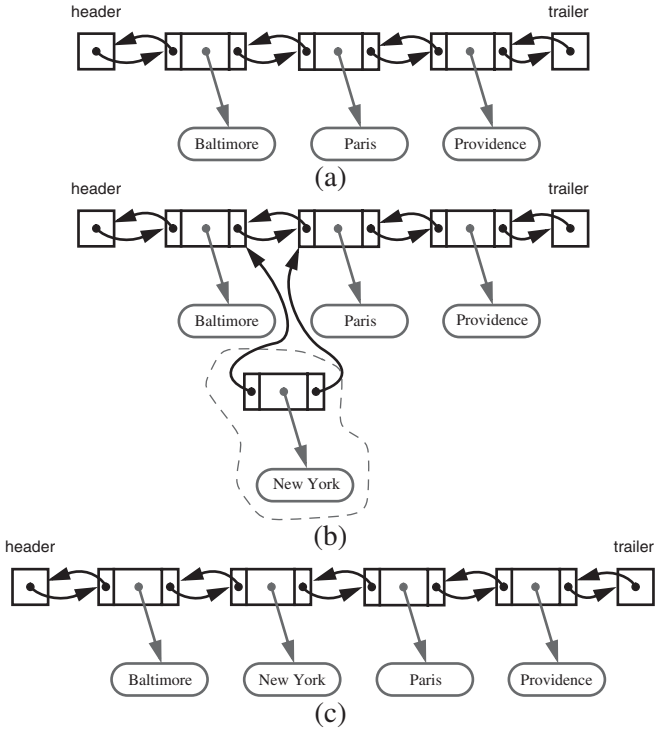


Figure 2.12: Adding a new node after the position for "Baltimore": (a) before the insertion; (b) creating node v and linking it in; (c) after the insertion.

The algorithms for methods insertBefore, insertFirst, and insertLast are similar to that for method insertAfter; we leave their details as an exercise (R-2.3). Next, consider the remove(p) method, which removes the element e stored at position p. To perform this operation, we link the two neighbors of p to refer to one another as new neighbors—linking out p. Note that after p is linked out, no nodes will be pointing to p; hence, a garbage collector can reclaim the space for p. This algorithm is given in Algorithm 2.13 and is illustrated in Figure 2.14. Recalling our use of sentinels, note that this algorithm works even if p is the first, last, or only real position in the list.

Algorithm remove(p):
  $t \leftarrow p$.element // a temporary variable to hold the return value
  (p.prev).next $\leftarrow p$.next // linking out p
  (p.next).prev $\leftarrow p$.prev
  p.prev $\leftarrow$ **null** // invalidating the position p
  p.next $\leftarrow$ **null**
  return t

Algorithm 2.13: Removing an element e stored at a position p in a linked list.

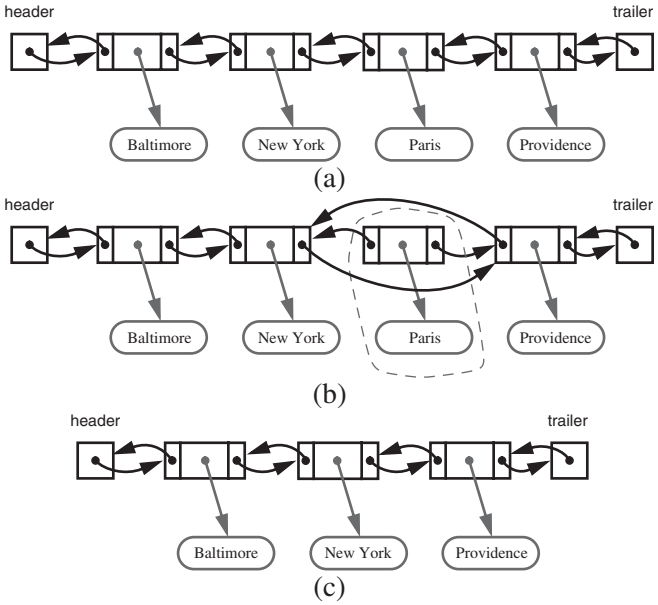


Figure 2.14: Removing the object stored at the position for "Paris": (a) before the removal; (b) linking out the old node; (c) after the removal (and garbage collection).

Analyzing List Implementations

Let us consider the performance of the above node-based linked list implementation. It should not be too difficult to see that all of the methods for a linked list can be implemented to run in $O(1)$ time using a node-based approach. That is, we can perform the methods for a linked list to have running times as shown in Table 2.15.

Method	Time
first()	$O(1)$
last()	$O(1)$
before(p)	$O(1)$
after(p)	$O(1)$
insertBefore(p, e)	$O(1)$
insertAfter(p, e)	$O(1)$
remove(p)	$O(1)$

Table 2.15: Worst-case performance of a node-based linked list with n elements. The space usage is $O(n)$, where n is the number of elements in the list.

Thinking generally about accessing elements by either indices or nodes, we can compare the performance of a linked list to that of an array-based list implementation. If we need to be accessing elements by their ranks or indices, clearly an array-based list is more efficient than a linked list, since the only way to determine the rank of an element e in a linked list is to follow the sequence of pointers from the node storing e to the end or beginning of its list. Thus, using a linked list for index-based operations would require $O(n)$ time for each such operation. Regarding update operations, the linked-list implementation beats the array-based implementation in the position-based update operations, since it can insert or remove elements in the "middle" of a list in constant time, whereas an array-based implementation takes $O(n)$ time (to keep the list contiguous in the array).

Considering space usage, note that an array requires $O(N)$ space, where N is the size of the array (unless we utilize an extendable array), while a doubly linked list uses $O(n)$ space, where n is the number of elements in the sequence. If n is much less than N, this implies that the asymptotic space usage of a linked-list implementation is better than that of a fixed-size array, although there is a small constant factor overhead that is larger for linked lists, since arrays do not need links to maintain the ordering of their cells.

Array-based and linked-list implementations of lists each have advantages and disadvantages, therefore. The correct one for a particular application depends on the operations that are to be performed and the memory space available.

2.3 Trees

Viewed abstractly, a ***tree*** is a data structure that stores elements hierarchically. With the exception of the top element, each element in a tree has a ***parent*** element and zero or more ***children*** elements. A tree is usually visualized by placing elements inside ovals or rectangles, and by drawing the connections between parents and children with straight lines. (See Figure 2.16.) We typically call the top element the ***root*** of the tree, but it is drawn as the highest element, with the other elements being connected below (just the opposite of a botanical tree).

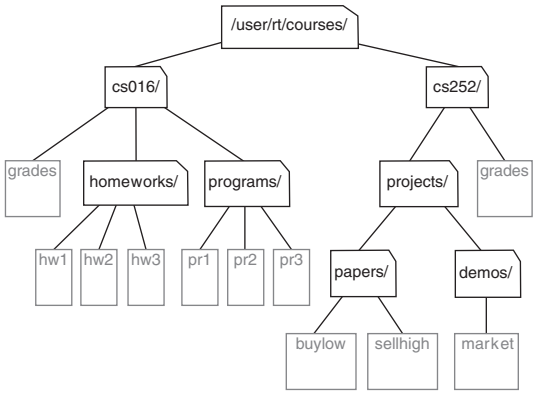


Figure 2.16: A tree representing a portion of a file system.

A ***tree*** T is a set of ***nodes*** storing elements in a ***parent-child*** relationship with the following properties:

- T has a special node r, called the ***root*** of T, with no ***parent*** node.
- Each node v of T different from r has a unique ***parent*** node u.

Note that according to the above definition, a tree cannot be empty, since it must have at least one node, the root. One could also allow the definition to include empty trees, but we adopt the convention that a tree always has a root so as to keep our presentation simple and to avoid having to always deal with the special case of an empty tree in our algorithms.

If node u is the parent of node v, then we say that v is a ***child*** of u. Two nodes that are children of the same parent are ***siblings***. A node is ***external*** if it has no children, and it is ***internal*** if it has one or more children. External nodes are also known as ***leaves***. The ***subtree*** of T ***rooted*** at a node v is the tree consisting of all the descendants of v in T (including v itself). An ***ancestor*** of a node is either the node itself, its parent, or an ancestor of its parent. Conversely, we say that a node v is a ***descendant*** of a node u if u is an ancestor of v.

Example 2.2: *In most operating systems, files are organized hierarchically into nested directories (also called folders), which are presented to the user in the form of a tree. (See Figure 2.16.) More specifically, the internal nodes of the tree are associated with directories and the external nodes are associated with regular files. In Unix-like operating systems, the root of the tree is appropriately called the "root directory," and is represented by the symbol "/." It is the ancestor of all directories and files in such a file system.*

A tree is ***ordered*** if there is a linear ordering defined for the children of each node; that is, we can identify children of a node as being the first, second, third, and so on. Ordered trees typically indicate the linear order relationship existing between siblings by listing them in the correct order.

Example 2.3: *A structured document, such as a book, is hierarchically organized as a tree whose internal nodes are chapters, sections, and subsections, and whose external nodes are paragraphs, tables, figures, the bibliography, and so on. (See Figure 2.17.) We could in fact consider expanding the tree further to show paragraphs consisting of sentences, sentences consisting of words, and words consisting of characters. In any case, such a tree is an example of an ordered tree, because there is a well-defined ordering among the children of each node.*

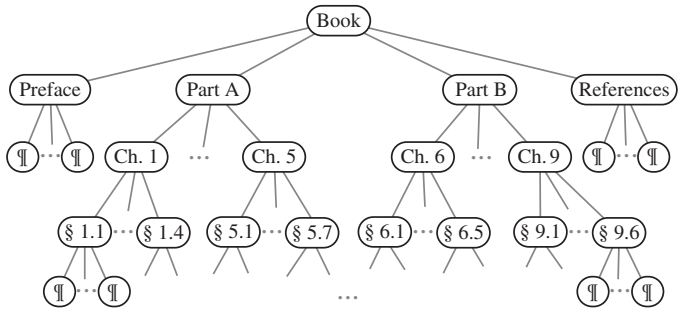


Figure 2.17: A tree associated with a book.

A ***binary tree*** is an ordered tree in which every node has at most two children. A binary tree is ***proper*** if each internal node has two children. For each internal node in a binary tree, we label each child as either being a ***left child*** or a ***right child***. These children are ordered so that a left child comes before a right child. The subtree rooted at a left or right child of an internal node v is called a ***left subtree*** or ***right subtree***, respectively, of v. Of course, even an improper binary tree is still a general tree, with the property that each internal node has at most two children. Binary trees have a number of useful applications, including the following.

Example 2.4: *An arithmetic expression can be represented by a tree whose external nodes are associated with variables or constants, and whose internal nodes are associated with one of the operators $+$, $-$, $\times$, and $/$. (See Figure 2.18.) Each node in such a tree has a value associated with it.*

- *If a node is external, then its value is that of its variable or constant.*
- *If a node is internal, then its value is defined by applying its operation to the values of its children.*

Such an arithmetic expression tree is a proper binary tree, since each of the operators $+$, $-$, $\times$, and $/$ take exactly two operands. Of course, if we were to allow for unary operators, like negation ($-$), as in "$-x$," then we could have an improper binary tree.

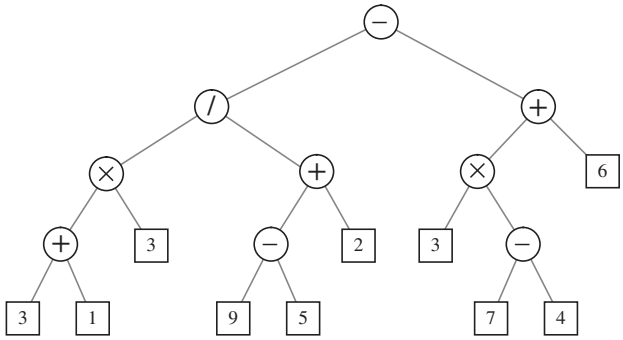


Figure 2.18: A binary tree representing an arithmetic expression. This tree represents the expression $((((3 + 1) \times 3)/((9 - 5) + 2)) - ((3 \times (7 - 4)) + 6))$. The value associated with the internal node labeled "/" is 2.

2.3.1 A Tree Definition

Viewed abstractly, a tree stores elements at positions, which, as with positions in a list, are defined relative to neighboring positions. The ***positions*** in a tree are its ***nodes***, and neighboring positions satisfy the parent-child relationships that define a valid tree. Therefore, we use the terms "position" and "node" interchangeably for trees. As with a list position, a position object for a tree supports the element() method, which returns the object at this position. The real power of node positions in a tree, however, comes from the following ***accessor methods*** for a tree:

root(): Return the root of the tree.

parent(v): Return the parent of node v; an error occurs if v is root.

children(v): Return a set containing the children of node v.

If a tree T is ordered, then the children(v) operation returns the children of v in order. If v is an external node, then children(v) is an empty set.

In addition, we also include the following ***query methods***:

isInternal(v): Test whether node v is internal.

isExternal(v): Test whether node v is external.

isRoot(v): Test whether node v is the root.

There are also a number of methods a tree should support that are not necessarily related to its tree structure. Such ***generic methods*** include the following:

size(): Return the number of nodes in the tree.

elements(): Return a set containing all the elements stored at nodes of the tree.

positions(): Return a set containing all the nodes of the tree.

swapElements(v, w): Swap the elements stored at the nodes v and w.

replaceElement(v, e): Replace with e and return the element stored at node v.

We do not define any specialized update methods for a tree here. Instead, let us reserve the potential to define different tree update methods in conjunction with specific tree applications.

2.3.2 Tree Traversal

In this section, we present algorithms for performing some important ***traversal*** operations on a tree.

Assumptions

In order to analyze the running time of tree-based algorithms, we make the following assumptions on the running times of various methods for a tree:

- The accessor methods root() and parent(v) take $O(1)$ time.
- The query methods isInternal(v), isExternal(v), and isRoot(v) take $O(1)$ time, as well.
- The accessor method children(v) takes $O(c_v)$ time, where c_v is the number of children of v.
- The generic methods swapElements(v, w) and replaceElement(v, e) take $O(1)$ time.
- The generic methods elements() and positions(), which return sets, take $O(n)$ time, where n is the number of nodes in the tree.

In Section 2.3.4, we present data structures for trees that satisfy the above assumptions. Before we describe how to implement a tree using a concrete data structure, however, let us describe how we can use the methods for an abstract tree structure to solve some interesting problems for trees.

Depth and Height

Let v be a node of a tree T. The ***depth*** of v is the number of ancestors of v, excluding v itself. Note that this definition implies that the depth of the root of T is 0. The depth of a node v can also be recursively defined as follows:

- If v is the root, then the depth of v is 0.
- Otherwise, the depth of v is one plus the depth of the parent of v.

Based on the above definition, the recursive algorithm depth, shown in Algorithm 2.19, computes the depth of a node v of T by calling itself recursively on the parent of v, and adding 1 to the value returned.

Algorithm depth(T, v):
  if T.isRoot(v) **then**
    return 0
  else
    return $1 +$ depth$(T, T$.parent$(v))$

Algorithm 2.19: Algorithm depth for computing the depth of a node v in a tree T.

The running time of algorithm depth(T, v) is $O(1 + d_v)$, where d_v denotes the depth of the node v in the tree T, because the algorithm performs a constant-time recursive step for each ancestor of v. Thus, in the worst case, the depth algorithm runs in $O(n)$ time, where n is the total number of nodes in the tree T, since some nodes may have nearly this depth in T. Although such a running time is a function of the input size, it is more accurate to characterize the running time in terms of the parameter d_v, since this will often be much smaller than n.

The ***height*** of a tree T is equal to the maximum depth of an external node of T. While this definition is correct, it does not lead to an efficient algorithm. Indeed, if we were to apply the above depth-finding algorithm to each node in the tree T, we would derive an $O(n^2)$-time algorithm to compute the height of T. We can do much better, however, using the following recursive definition of the ***height*** of a node v in a tree T:

- If v is an external node, then the height of v is 0.
- Otherwise, the height of v is one plus the maximum height of a child of v.

The ***height*** of a tree T is the height of the root of T.

Algorithm height, shown in Algorithm 2.20 computes the height of tree T in an efficient manner by using the above recursive definition of height. The algorithm is expressed by a recursive method height(T, v) that computes the height of the subtree of T rooted at a node v. The height of tree T is obtained by calling height$(T, T.\text{root}())$.

Algorithm height(T, v):
  if T.isExternal(v) **then**
    return 0
  else
    $h = 0$
    for each $w \in T$.children(v) **do**
      $h = \max(h, \text{height}(T, w))$
    return $1 + h$

Algorithm 2.20: Algorithm height for computing the height of the subtree of tree T rooted at a node v.

The height algorithm is recursive, and if it is initially called on the root of T, it will eventually be called once on each node of T. Thus, we can determine the running time of this method by an amortization argument where we first determine the amount of time spent at each node (on the nonrecursive part), and then sum this time bound over all the nodes. The computation of a set returned by children(v) takes $O(c_v)$ time, where c_v denotes the number of children of node v. Also, the **for** loop has c_v iterations, and each iteration of the loop takes $O(1)$ time plus the time for the recursive call on a child of v. Thus, the algorithm height spends $O(1 + c_v)$ time at each node v, and its running time is $O(\sum_{v \in T}(1 + c_v))$. In order to complete the analysis, we make use of the following property.

Theorem 2.5: *Let T be a tree with n nodes, and let c_v denote the number of children of a node v of T. Then*

$$\sum_{v \in T} c_v = n - 1.$$

Proof: Each node of T, with the exception of the root, is a child of another node, and thus contributes one unit to the summation $\sum_{v \in T} c_v$. ■

By Theorem 2.5, the running time of Algorithm height when called on the root of T is $O(n)$, where n is the number of nodes of T.

A ***traversal*** of a tree T is a systematic way of accessing, or "visiting," all the nodes of T. We next present basic traversal schemes for trees, called preorder and postorder traversals.

Preorder Traversal

In a ***preorder*** traversal of a tree T, the root of T is visited first and then the subtrees rooted at its children are traversed recursively. If the tree is ordered, then the subtrees are traversed according to the order of the children. The specific action associated with the "visit" of a node v depends on the application of this traversal, and could involve anything from incrementing a counter to performing some complex computation for v. The pseudocode for the preorder traversal of the subtree rooted at a node v is shown in Algorithm 2.21. We initially call this routine as preorder$(T, T.\mathsf{root}())$.

Algorithm preorder(T, v):
  perform the "visit" action for node v
  for each child w of v **do**
    recursively traverse the subtree rooted at w by calling preorder(T, w)

Algorithm 2.21: Algorithm preorder.

The preorder traversal algorithm is useful for producing a linear ordering of the nodes of a tree where parents must always come before their children in the ordering. Such orderings have several different applications; we explore a simple instance of such an application in the next example.

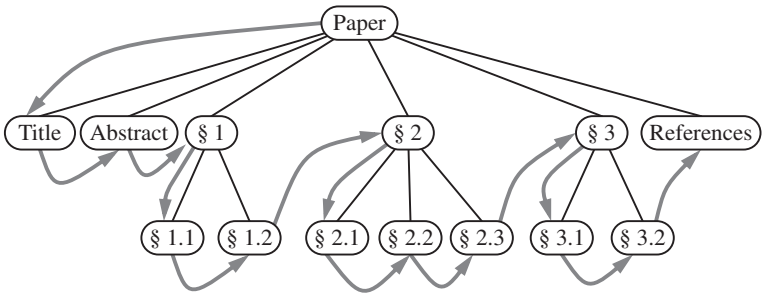


Figure 2.22: Preorder traversal of an ordered tree.

Example 2.6: *The preorder traversal of the tree associated with a document, as in Example 2.3, examines an entire document sequentially, from beginning to end. If the external nodes are removed before the traversal, then the traversal examines the table of contents of the document. (See Figure 2.22.)*

The analysis of preorder traversal is actually similar to that of algorithm height given above. At each node v, the nonrecursive part of the preorder traversal algorithm requires time $O(1 + c_v)$, where c_v is the number of children of v. Thus, by Theorem 2.5, the overall running time of the preorder traversal of T is $O(n)$.

Postorder Traversal

Another important tree traversal algorithm is the ***postorder*** traversal. This algorithm can be viewed as the opposite of the preorder traversal, because it recursively traverses the subtrees rooted at the children of the root first, and then visits the root. It is similar to the preorder traversal, however, in that we use it to solve a particular problem by specializing an action associated with the "visit" of a node v. Still, as with the preorder traversal, if the tree is ordered, we make recursive calls for the children of a node v according to their specified order. Pseudo-code for the postorder traversal is given in Algorithm 2.23.

Algorithm postorder(T, v):
  for each child w of v **do**
    recursively traverse the subtree rooted at w by calling postorder(T, w)
  perform the "visit" action for node v

Algorithm 2.23: Method postorder.

The name of the postorder traversal comes from the fact that this traversal method will visit a node v after it has visited all the other nodes in the subtree rooted at v. (See Figure 2.24.)

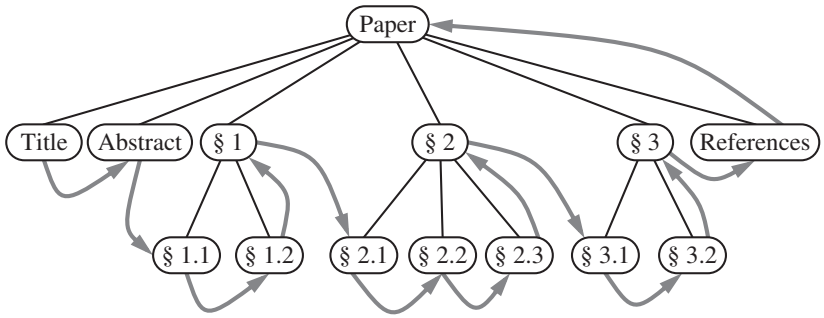


Figure 2.24: Postorder traversal of the ordered tree of Figure 2.22.

The analysis of the running time of a postorder traversal is analogous to that of a preorder traversal. The total time spent in the nonrecursive portions of the algorithm is proportional to the time spent visiting the children of each node in the tree. Thus, a postorder traversal of a tree T with n nodes takes $O(n)$ time, assuming that visiting each node takes $O(1)$ time. That is, the postorder traversal runs in linear time.

The postorder traversal method is useful for solving problems where we wish to compute some property for each node v in a tree, but computing that property for v requires that we have already computed that same property for v's children.

2.3.3 Binary Trees

One kind of tree that is of particular interest is the binary tree. As we mentioned in Section 2.3, a proper ***binary tree*** is an ordered tree in which each internal node has exactly two children. We make the convention that, unless otherwise stated, binary trees are assumed to be proper. Note that our convention for binary trees is made without loss of generality, for we can easily convert any improper binary tree into a proper one, as we explore in Exercise C-2.16. Even without such a conversion, we can consider an improper binary tree as proper, simply by viewing missing external nodes as "null nodes" or place holders that still count as nodes.

Viewed abstractly, a binary tree is a specialization of a tree that supports three additional accessor methods:

leftChild(v): Return the left child of v; an error condition occurs if v is an external node.

rightChild(v): Return the right child of v; an error condition occurs if v is an external node.

sibling(v): Return the sibling of node v; an error condition occurs if v is the root.

Note that these methods must have additional error conditions if we are dealing with improper binary trees. For example, in an improper binary tree, an internal node may not have the left child or right child. We do not include here any methods for updating a binary tree, for such methods can be created as required in the context of specific needs.

Properties of Binary Trees

We denote the set of all nodes of a tree T at the same depth d as the ***level*** d of T. In a binary tree, level 0 has one node (the root), level 1 has at most two nodes (the children of the root), level 2 has at most four nodes, and so on. (See Figure 2.25.) In general, level d has at most 2^d nodes, which implies the following theorem (whose proof is left to Exercise R-2.6).

Theorem 2.7: *Let T be a proper binary tree with n nodes, and let h denote the height of T. Then T has the following properties:*

1. *The number of external nodes in T is at least $h + 1$ and at most 2^h.*
2. *The number of internal nodes in T is at least h and at most $2^h - 1$.*
3. *The total number of nodes in T is at least $2h + 1$ and at most $2^{h+1} - 1$.*
4. *The height of T is at least $\log(n + 1) - 1$ and at most $(n - 1)/2$, that is, $\log(n + 1) - 1 \leq h \leq (n - 1)/2$.*

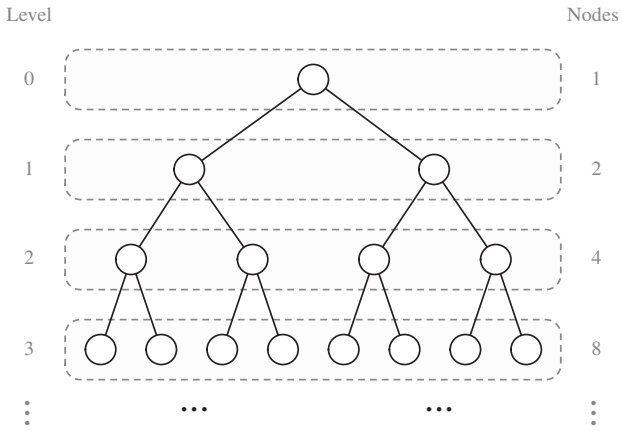


Figure 2.25: Maximum number of nodes in the levels of a binary tree.

In addition, we also have the following.

Theorem 2.8: *In a proper binary tree T, the number of external nodes is 1 more than the number of internal nodes.*

Proof: The proof is by induction. If T itself has only one node v, then v is external, and the proposition clearly holds. Otherwise, we remove from T an (arbitrary) external node w and its parent v, which is an internal node. If v has a parent u, then we reconnect u with the former sibling z of w, as shown in Figure 2.26. This operation, which we call removeAboveExternal(w), removes one internal node and one external node, and it leaves the tree being a proper binary tree. Thus, by the inductive hypothesis, the number of external nodes in this tree is one more than the number of internal nodes. Since we removed one internal and one external node to reduce T to this smaller tree, this same property must hold for T. ■

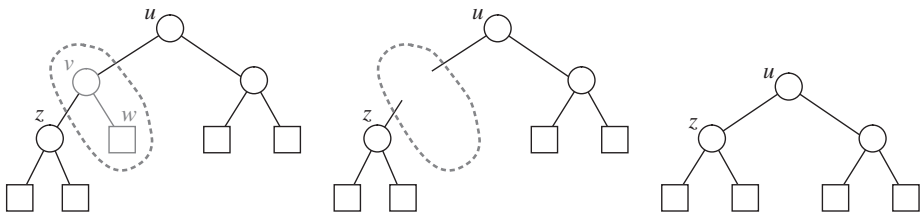


Figure 2.26: Operation removeAboveExternal(w), which removes an external node and its parent node, used in the justification of Theorem 2.8.

Note that the above relationship does not hold, in general, for nonbinary trees.

In subsequent chapters, we explore some important applications of the above facts. Before we can discuss such applications, however, we should first understand more about how binary trees are traversed and represented.

Traversals of a Binary Tree

As with general trees, computations performed on binary trees often involve tree traversals. In this section, we present binary tree traversal algorithms. As for running times, in addition to the assumptions on the running time for tree methods made in Section 2.3.2, we assume that, for a binary tree, the children(v) operation takes $O(1)$ time, because each node has either zero or two children. Likewise, we assume that methods leftChild(v), rightChild(v), and sibling(v) each take $O(1)$ time.

Preorder Traversal of a Binary Tree

Since any binary tree can also be viewed as a general tree, the preorder traversal for general trees (Code Fragment 2.21) can be applied to any binary tree. We can simplify the pseudocode in the case of a binary tree traversal, however, as we show in Algorithm 2.27.

Algorithm binaryPreorder(T, v):
  perform the "visit" action for node v
  if v is an internal node **then**
    binaryPreorder(T, T.leftChild(v)) // recursively traverse left subtree
    binaryPreorder(T, T.rightChild(v)) // recursively traverse right subtree

Algorithm 2.27: Algorithm binaryPreorder that performs the preorder traversal of the subtree of a binary tree T rooted at node v.

Postorder Traversal of a Binary Tree

Analogously, the postorder traversal for general trees (Algorithm 2.23) can be specialized for binary trees, as shown in Algorithm 2.28.

Algorithm binaryPostorder(T, v):
  if v is an internal node **then**
    binaryPostorder(T, T.leftChild(v)) // recursively traverse left subtree
    binaryPostorder(T, T.rightChild(v)) // recursively traverse right subtree
  perform the "visit" action for the node v

Algorithm 2.28: Algorithm binaryPostorder for performing the postorder traversal of the subtree of a binary tree T rooted at v.

Interestingly, the specialization of the general preorder and postorder traversal methods to binary trees suggests a third traversal in a binary tree that is different from both the preorder and postorder traversals.

Inorder Traversal of a Binary Tree

An additional traversal method for a binary tree is the ***inorder*** traversal. In this traversal, we visit a node between the recursive traversals of its left and right subtrees, as shown in Algorithm 2.29.

Algorithm inorder(T, v):
  if v is an internal node **then**
    inorder$(T, T.\text{leftChild}(v))$ // recursively traverse left subtree
  perform the "visit" action for node v
  if v is an internal node **then**
    inorder$(T, T.\text{rightChild}(v))$ // recursively traverse right subtree

Algorithm 2.29: Algorithm inorder for performing the inorder traversal of the subtree of a binary tree T rooted at a node v.

The inorder traversal of a binary tree T can be informally viewed as visiting the nodes of T "from left to right." Indeed, for every node v, the inorder traversal visits v after all the nodes in the left subtree of v and before all the nodes in the right subtree of v. (See Figure 2.30.)

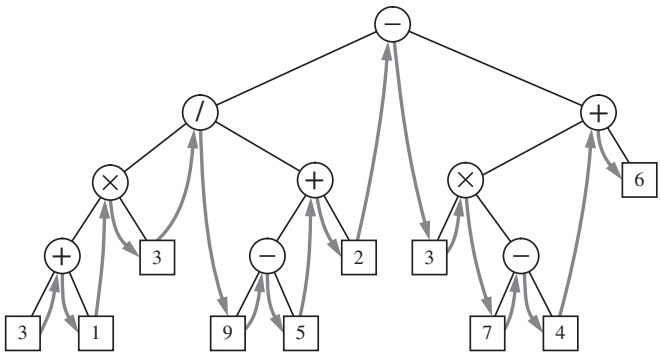


Figure 2.30: Inorder traversal of a binary tree.

A Unified Tree Traversal Framework

Each traversal visits the nodes of a tree in a certain order and is guaranteed to visit each node exactly once. We can unify the tree-traversal algorithms given above into a single framework, however, by relaxing the requirement that each node be visited exactly once. The resulting traversal method is called the ***Euler tour traversal***, which we study next. The advantage of this traversal is that it allows for more general kinds of algorithms to be expressed easily.

The Euler Tour Traversal of a Binary Tree

The Euler tour traversal of a binary tree T can be informally defined as a "walk" around T, where we start by going from the root toward its left child, viewing the edges of T as being "walls" that we always keep to our left. (See Figure 2.31.) Each node v of T is encountered three times by the Euler tour:

- "On the left" (before the Euler tour of v's left subtree)
- "From below" (between the Euler tours of v's two subtrees)
- "On the right" (after the Euler tour of v's right subtree).

If v is external, then these three "visits" actually all happen at the same time.

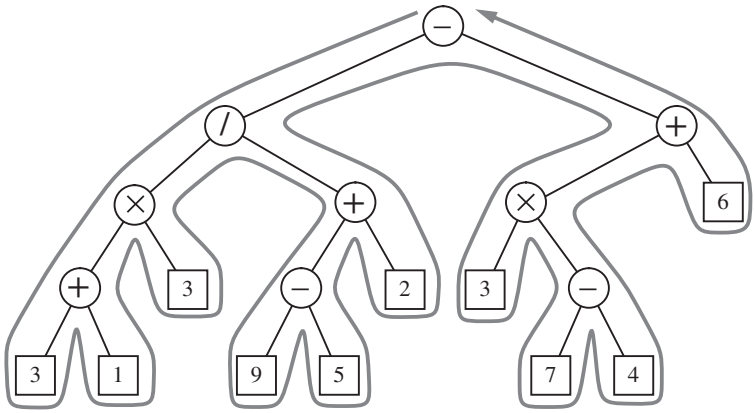


Figure 2.31: Euler tour traversal of a binary tree.

We give pseudocode for the Euler tour of the subtree rooted at a node v in Algorithm 2.32.

Algorithm eulerTour(T, v):
perform the action for visiting node v on the left
if v is an internal node **then**
recursively tour the left subtree of v by calling eulerTour$(T, T.\text{leftChild}(v))$
perform the action for visiting node v from below
if v is an internal node **then**
recursively tour the right subtree of v by calling eulerTour$(T, T.\text{rightChild}(v))$
perform the action for visiting node v on the right

Algorithm 2.32: Algorithm eulerTour for computing the Euler tour traversal of the subtree of a binary tree T rooted at a node v.

The preorder traversal of a binary tree is equivalent to an Euler tour traversal such that each node has an associated "visit" action occur only when it is encountered on the left. Likewise, the inorder and postorder traversals of a binary tree are equivalent to an Euler tour such that each node has an associated "visit" action occur only when it is encountered from below or on the right, respectively.

The Euler tour traversal extends the preorder, inorder, and postorder traversals, but it can also perform other kinds of traversals. For example, suppose we wish to compute the number of descendants of each node v in an n node binary tree T. We start an Euler tour by initializing a counter to 0, and then increment the counter each time we visit a node on the left. To determine the number of descendants of a node v, we compute the difference between the values of the counter when v is visited on the left and when it is visited on the right, and add 1. This simple rule gives us the number of descendants of v, because each node in the subtree rooted at v is counted between v's visit on the left and v's visit on the right. Therefore, we have an $O(n)$-time method for computing the number of descendants of each node in T.

The running time of the Euler tour traversal is easy to analyze, assuming visiting a node takes $O(1)$ time. Namely, in each traversal, we spend a constant amount of time at each node of the tree during the traversal, so the overall running time is $O(n)$ for an n node tree.

Another application of the Euler tour traversal is to print a fully parenthesized arithmetic expression from its expression tree (Example 2.4). The method printExpression, shown in Algorithm 2.33, accomplishes this task by performing the following actions in an Euler tour:

- "On the left" action: if the node is internal, print "("
- "From below" action: print the value or operator stored at the node
- "On the right" action: if the node is internal, print ")."

Algorithm printExpression(T, v):
  if T.isExternal(v) **then**
    print the value stored at v
  else
    print "("
    printExpression$(T, T$.leftChild$(v))$
    print the operator stored at v
    printExpression$(T, T$.rightChild$(v))$
    print ")"

Algorithm 2.33: An algorithm for printing the arithmetic expression associated with the subtree of an arithmetic expression tree T rooted at v.

Having presented these pseudocode examples, we now describe a number of efficient ways of realizing the tree abstraction by concrete data structures, such as arrays and linked structures.

2.3.4 Data Structures for Representing Trees

In this section, we describe concrete data structures for representing trees.

A Linked Structure for Binary Trees

A natural way to implement a binary tree T is to use a ***linked structure***. In this approach we represent each node v of T by an object with references to the element stored at v and the positions associated with the children and parent of v. We show a linked structure representation of a binary tree in Figure 2.34.

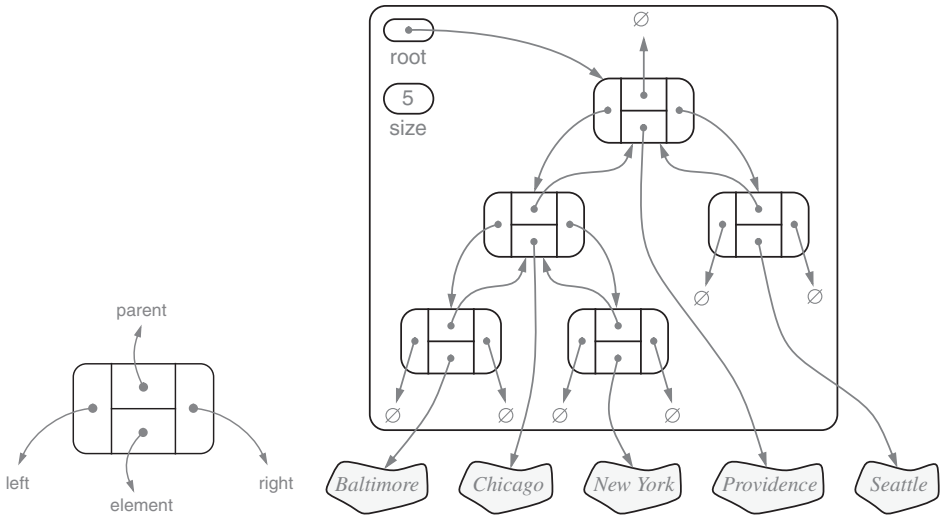


Figure 2.34: An example linked data structure for representing a binary tree: (a) object associated with a node; (b) a structure for a binary tree with five nodes.

If v is the root of T, then the reference to the parent node is null, and if v is an external node, then the references to the children of v are null. If we wish to save space for cases when external nodes are empty, then we can have references to empty external nodes be null. That is, we can allow a reference from an internal node to an external node child to be null. In addition, it is fairly straightforward to implement each of the methods size(), isEmpty(), swapElements(v, w), and replaceElement(v, e) in $O(1)$ time. Moreover, the method positions() can be implemented by performing an inorder traversal, and implementing the method elements() is similar. Thus, methods positions() and elements() take $O(n)$ time each. Considering the space used by this data structure, note that there is a constant-sized object for every node of tree T. Thus, the overall space used is $O(n)$.

A Linked Structure for General Trees

We can extend the linked structure for binary trees to represent general trees. Since there is no limit on the number of children that a node v in a general tree can have, we use a container (for example, a list or array) to store the children of v, instead of using instance variables. This structure is schematically illustrated in Figure 2.35, assuming we implement the container for a node as a list.

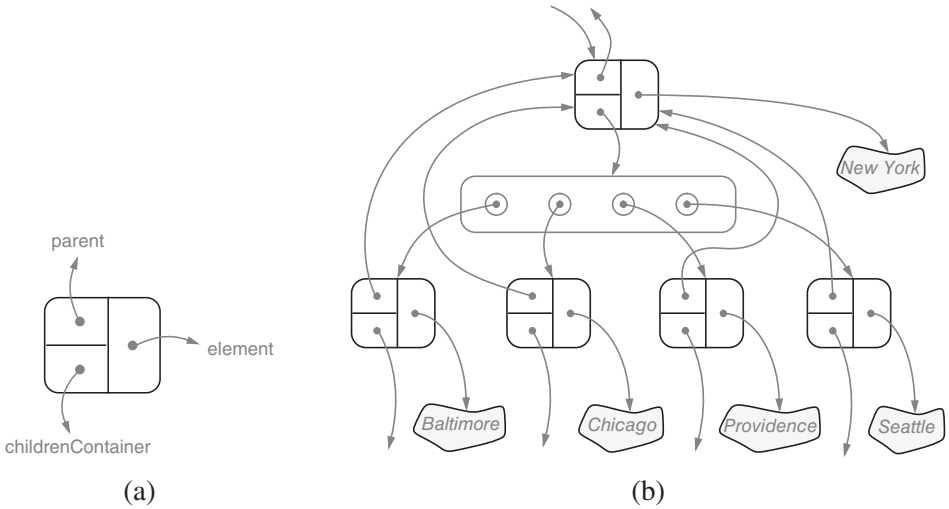


Figure 2.35: The linked structure for a tree: (a) the object associated with a node; (b) the portion of the data structure associated with a node and its children.

We note that the performance of a linked implementation of a tree, shown in Table 2.36, is similar to that of the linked implementation of a binary tree. The main difference is that in the implementation of a tree we use an efficient container, such as a list or array, to store the children of each node v, instead of direct links to exactly two children.

Operation	Time
size, isEmpty	$O(1)$
positions, elements	$O(n)$
swapElements, replaceElement	$O(1)$
root, parent	$O(1)$
children(v)	$O(c_v)$
isInternal, isExternal, isRoot	$O(1)$

Table 2.36: Running times of the methods of an n-node tree implemented with a linked structure. We let c_v denote the number of children of a node v.

2.4 Exercises

Reinforcement

R-2.1 Suppose you are given an array, A, containing n numbers in order. Describe in pseudocode an efficient algorithm for reversing the order of the numbers in A using a single for-loop that indexes through the cells of A, to insert each element into a stack, and then another for-loop that removes the elements from the stack and puts them back into A in reverse order. What is the running time of this algorithm?

R-2.2 Solve the previous exercise using a queue instead of stack. That is, suppose you are given an array, A, containing n numbers in order, as in the previous exercise. Describe in pseudocode an efficient algorithm for reversing the order of the numbers in A using a single for-loop that indexes through the cells of A, to insert each element into a queue, and then another for-loop that removes the elements from the queue and puts them back into A in reverse order. What is the running time of this algorithm?

R-2.3 Describe, using pseudocode, an implementation of the method insertBefore(p, e), for a linked list, assuming the list is implemented using a doubly linked list.

R-2.4 Draw an expression tree that has four external nodes, storing the numbers 1, 5, 6, and 7 (with each number stored one per external node but not necessarily in this order), and has three internal nodes, each storing an operation from the set $\{+, -, \times, /\}$ of binary arithmetic operators, so that the value of the root is 21. The operators are assumed to return rational numbers (not integers), and an operator may be used more than once (but we only store one operator per internal node).

R-2.5 Let T be an ordered tree with more than one node. Is it possible that the preorder traversal of T visits the nodes in the same order as the postorder traversal of T? If so, give an example; otherwise, argue why this cannot occur. Likewise, is it possible that the preorder traversal of T visits the nodes in the reverse order of the postorder traversal of T? If so, give an example; otherwise, argue why this cannot occur.

R-2.6 Answer the following questions so as to justify Theorem 2.7.

a. Draw a binary tree with height 7 and maximum number of external nodes.
b. What is the minimum number of external nodes for a binary tree with height h? Justify your answer.
c. What is the maximum number of external nodes for a binary tree with height h? Justify your answer.
d. Let T be a binary tree with height h and n nodes. Show that

$$\log(n+1) - 1 \leq h \leq (n-1)/2.$$

e. For which values of n and h can the above lower and upper bounds on h be attained with equality?

R-2.7 Let T be a binary tree such that all the external nodes have the same depth. Let D_e be the sum of the depths of all the external nodes of T, and let D_i be the sum of the depths of all the internal nodes of T. Find constants a and b such that

$$D_e + 1 = aD_i + bn,$$

where n is the number of nodes of T.

R-2.8 Let T be a binary tree with n nodes, and let p be the level numbering of the nodes of T, so that the root, r, is numbered as $p(r) = 1$, and a node v has left child numbered $2p(v)$ and right child numbered $2p(v) + 1$, if they exist.

a. Show that, for every node v of T, $p(v) \leq 2^{(n+1)/2} - 1$.
b. Show an example of a binary tree with at least five nodes that attains the above upper bound on the maximum value of $p(v)$ for some node v.

Creativity

C-2.1 A ***double-ended queue***, or ***deque***, is a list that allows for insertions and removals at either its head or its tail. Describe a way to implement a deque using a doubly linked list, so that every operation runs in $O(1)$ time.

C-2.2 Suppose that a friend has implemented a deque, as defined in the previous exercise, using a singly linked list, but hasn't given you the details, for example, of whether the links go forward or backward in the list or whether sentinel nodes are used. Nevertheless, show that one of the insertion or removal methods must take $\Omega(n)$ time, where n is the number of elements in the deque.

C-2.3 Describe, in pseudocode, a link-hopping method for finding the middle node of a doubly linked list with header and trailer sentinels, and an odd number of real nodes between them. (Note: This method must only use link hopping; it cannot use a counter.) What is the running time of this method?

C-2.4 Describe how to implement a queue using two stacks, so that the amortized running time for dequeue and enqueue is $O(1)$, assuming that the stacks support constant-time push, pop, and size methods. What is the worst-case running time of the enqueue() and dequeue() methods in this case?

C-2.5 Describe how to implement a stack using two queues. What is the running time of the push() and pop() methods in this case?

C-2.6 Describe a recursive algorithm for enumerating all permutations of the numbers $\{1, 2, \ldots, n\}$. What is the running time of your method?

C-2.7 Show that a stack and a queue can be used to realize any permutation. That is, suppose you are given an empty stack, S, and the numbers, $1, 2, \ldots, n$, in this order, initially stored in a queue, Q. Show how to use only these two structures, and at most a constant number of additional registers, to result in any given permutation, π, of the numbers, $1, 2, \ldots, n$, stored in the Q in the order specified by π. What is the running time of your algorithm?

C-2.8 Describe the structure and pseudocode for an array-based implementation of an index-based list that achieves $O(1)$ time for insertions and removals at index 0, as well as insertions and removals at the end of the list. Your implementation should also provide for a constant-time get method.

C-2.9 Using an array-based list, describe an efficient way of putting a sequence representing a deck of n cards into random order. Use the function randomInt(n), which returns a random number between 0 and $n-1$, inclusive. Your method should guarantee that every possible ordering is equally likely. What is the running time of your method?

C-2.10 Design an algorithm for drawing a binary tree, using quantities computed in a tree traversal.

C-2.11 Design algorithms for the following operations for a node v in a binary tree T:

- preorderNext(v): return the node visited after v in a preorder traversal of T
- inorderNext(v): return the node visited after v in an inorder traversal of T
- postorderNext(v): return the node visited after v in a postorder traversal of T.

What are the worst-case running times of your algorithms?

C-2.12 Give an $O(n)$-time algorithm for computing the depth of all the nodes of a tree T, where n is the number of nodes of T.

C-2.13 The ***balance factor*** of an internal node v of a binary tree is the difference between the heights of the right and left subtrees of v. Show how to specialize the Euler tour traversal to print the balance factors of all the nodes of a binary tree.

C-2.14 Two ordered trees T' and T'' are said to be ***isomorphic*** if one of the following holds:

- Both T' and T'' consist of a single node
- Both T' and T'' have the same number k of subtrees, and the ith subtree of T' is isomorphic to the ith subtree of T'', for $i = 1, \ldots, k$.

Design an algorithm that tests whether two given ordered trees are isomorphic. What is the running time of your algorithm?

C-2.15 Let a visit action in the Euler tour traversal be denoted by a pair (v, a), where v is the visited node and a is one of ***left***, ***below***, or ***right***. Design an algorithm for performing operation tourNext(v, a), which returns the visit action (w, b) following (v, a). What is the worst-case running time of your algorithm?

C-2.16 Show how to represent an improper binary tree by means of a proper one.

C-2.17 Let T be a binary tree with n nodes. Define a ***Roman node*** to be a node v in T, such that the number of descendants in v's left subtree differ from the number of descendants in v's right subtree by at most 5. Describe a linear-time method for finding each node v of T, such that v is not a Roman node, but all of v's descendants are Roman nodes.

C-2.18 Describe in pseudocode a nonrecursive method for performing an Euler tour traversal of a binary tree that runs in linear time and does not use a stack.

Hint: You can tell which visit action to perform at a node by taking note of where you are coming from.

C-2.19 Describe in pseudocode a nonrecursive method for performing an inorder traversal of a binary tree in linear time.

C-2.20 Let T be a binary tree with n nodes. Give a linear-time method that uses the methods of the BinaryTree interface to traverse the nodes of T by increasing values of the level numbering function p given in Exercise R-2.8. This traversal is known as the ***level order traversal***.

C-2.21 The ***path length*** of a tree T is the sum of the depths of all the nodes in T. Describe a linear-time method for computing the path length of a tree T (which is not necessarily binary).

C-2.22 Define the ***internal path length***, $I(T)$, of a tree T to be the sum of the depths of all the internal nodes in T. Likewise, define the ***external path length***, $E(T)$, of a tree T to be the sum of the depths of all the external nodes in T. Show that if T is a binary tree with n internal nodes, then $E(T) = I(T) + 2n$.

Applications

A-2.1 In the children's game "hot potato," a group of n children sit in a circle passing an object, called the "potato," around the circle (say in a clockwise direction). The children continue passing the potato until a leader rings a bell, at which point the child holding the potato must leave the game, and the other children close up the circle. This process is then continued until there is only one child remaining, who is declared the winner. Using a list, describe an efficient method for implementing this game. Suppose the leader always rings the bell immediately after the potato has been passed k times. (Determining the last child remaining in this variation of hot potato is known as the ***Josephus problem***.) What is the running time of your method in terms of n and k, assuming the list is implemented with a doubly linked list? What if the list is implemented with an array?

A-2.2 Suppose you work for a company, iPuritan.com, that has strict rules for when two employees, x and y, may date one another, requiring approval from their lowest-level common supervisor. The employees at iPuritan.com are organized in a tree, T, such that each node in T corresponds to an employee and each employee, z, is considered a supervisor for all of the employees in the subtree of T rooted at z (including z itself). The lowest-level common supervisor for x and y is the employee lowest in the organizational chart, T, that is a supervisor for both x and y. Thus, to find a lowest-level common supervisor for the two employees, x and y, you need to find the ***lowest common ancestor*** (LCA) between the two nodes for x and y, which is the lowest node in T that has both x and y as descendants (where we allow a node to be a descendant of itself). Given the nodes corresponding to the two employees x and y, describe an efficient algorithm for finding the supervisor who may approve whether x and y may date each other, that is, the LCA of x and y in T. What is the running time of your method?

A-2.3 Suppose you work for a company, iPilgrim.com, whose n employees are organized in a tree T, so that each node is associated with an employee and each employee is considered a supervisor for all the employees (including themselves) in his or her subtree in T, as in the previous exercise. Furthermore, suppose that communication in iPilgrim is done the "old fashioned" way, where, for an employee, x, to send a message to an employee, y, x must route this message up to a lowest-level common supervisor of x and y, who then routes this message down to y. The problem is to design an algorithm for finding the length of a longest route that any message must travel in iPilgrim.com. That is, for any node v in T, let d_v denote the depth of v in T. The ***distance*** between two nodes v and w in T is $d_v + d_w - 2d_u$, where u is the LCA u of v and w (as defined in the previous exercise). The ***diameter*** of T is the maximum distance between two nodes in T. Thus, the length of a longest route that any message must travel in iPilgrim.com is equal to the diameter of T. Describe an efficient algorithm for finding the diameter of T. What is the running time of your method?

Chapter Notes

The basic data structures of stacks, queues, and linked lists discussed in this chapter belong to the folklore of computer science. They were first chronicled by Knuth in his seminal book on *Fundamental Algorithms* [129]. In this chapter, we have taken the approach of defining basic data structures first abstractly in terms of their methods and then in terms of concrete implementations. This approach to data structure specification and implementation is an outgrowth of software engineering advances brought on by the object-oriented design approach, and is now considered a standard approach for teaching data structures. We were introduced to this approach to data structure design by the classic books by Aho, Hopcroft, and Ullman on data structures and algorithms [8, 9].

Sequences and lists are pervasive concepts in the C++ Standard Template Library (STL) [163], and they play fundamental roles in Java as well. Lists are also discussed in the book by Arnold and Gosling [14]) and others, including Aho, Hopcroft, and Ullman [9], who introduce the "position" abstraction, and Wood [217], who defines a list abstraction similar to ours. Implementations of sequences via arrays and linked lists are discussed in Knuth's seminal book, *Fundamental Algorithms* [129].

The concept of viewing data structures as containers (and other principles of object-oriented design) can be found in object-oriented design books by Booch [33] and Budd [40]. The concept also exists under the name "collection class" in books by Golberg and Robson [83] and Liskov and Guttag [144]. Our use of the "position" abstraction for tree nodes derives from the "position" and "node" abstractions introduced by Aho, Hopcroft, and Ullman [9]. Discussions of the classic preorder, inorder, and postorder tree traversal methods can be found in Knuth's *Fundamental Algorithms* book [129]. The Euler tour traversal technique comes from the parallel algorithms community, as it is introduced by Tarjan and Vishkin [204] and is discussed by JáJá [110] and by Karp and Ramachandran [124]. The algorithm for drawing a tree is part of the "folklore" of graph drawing algorithms. The reader interested in graph drawing is referred to the handbook edited by Tamassia [203] and the book by Di Battista, Eades, Tamassia and Tollis [55]. The puzzler in Exercise R-2.4 was communicated by Micha Sharir.