Chapter 2

ELEMENTARY DATA STRUCTURES

Now that we have examined the fundamental methods we need to express and analyze algorithms, we might feel all set to begin. But, alas, we need to make one last diversion, and that is a discussion of data structures. One of the basic techniques for improving algorithms is to structure the data in such a way that the resulting operations can be efficiently carried out. In this chapter, we review only the most basic and commonly used data structures. Many of these are used in subsequent chapters. We should be familiar with stacks and queues (Section 2.1), binary trees (Section 2.2), and graphs (Section 2.6) and be able to refer to the other structures as needed.

2.1 STACKS AND QUEUES

One of the most common forms of data organization in computer programs is the ordered or linear list, which is often written as $a = (a_1, a_2, \ldots, a_n)$. The a_i's are referred to as *atoms* and they are chosen from some set. The null or empty list has $n = 0$ elements. A *stack* is an ordered list in which all insertions and deletions are made at one end, called the *top*. A *queue* is an ordered list in which all insertions take place at one end, the *rear*, whereas all deletions take place at the other end, the *front*.

The operations of a stack imply that if the elements A, B, C, D, and E are inserted into a stack, in that order, then the first element to be removed (deleted) must be E. Equivalently we say that the last element to be inserted into the stack is the first to be removed. For this reason stacks are sometimes referred to as **L**ast **I**n **F**irst **O**ut (**LIFO**) lists. The operations of a queue require that the first element that is inserted into the queue is the first one to be removed. Thus queues are known as **F**irst **I**n **F**irst **O**ut (**FIFO**) lists. See Figure 2.1 for examples of a stack and a queue each containing the same

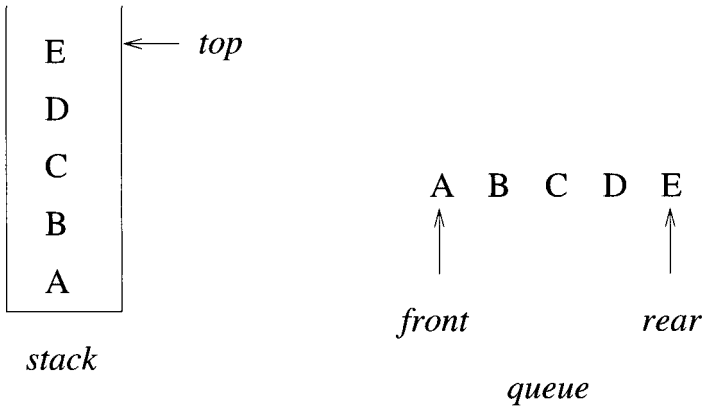

Figure 2.1 Example of a stack and a queue

five elements inserted in the same order. Note that the data object queue as defined here need not correspond to the concept of queue that is studied in queuing theory.

The simplest way to represent a stack is by using a one-dimensional array, say $stack[0 : n-1]$, where n is the maximum number of allowable entries. The first or bottom element in the stack is stored at $stack[0]$, the second at $stack[1]$, and the ith at $stack[i-1]$. Associated with the array is a variable, typically called *top*, which points to the top element in the stack. To test whether the stack is empty, we ask "**if** $(top < 0)$". If not, the topmost element is at $stack[top]$. Checking whether the stack is full can be done by asking "**if** $(top \geq n-1)$". Two more substantial operations are inserting and deleting elements. The corresponding algorithms are Add and Delete (Algorithm 2.1).

Each execution of Add or Delete takes a constant amount of time and is independent of the number of elements in the stack.

Another way to represent a stack is by using links (or pointers). A *node* is a collection of data and link information. A stack can be represented by using nodes with two fields, possibly called *data* and *link*. The data field of each node contains an item in the stack and the corresponding link field points to the node containing the next item in the stack. The link field of the last node is zero, for we assume that all nodes have an address greater than zero. For example, a stack with the items A, B, C, D, and E inserted in that order, looks as in Figure 2.2.

```
1   Algorithm Add(item)
2   // Push an element onto the stack. Return true if successful;
3   // else return false. item is used as an input.
4   {
5       if (top ≥ n − 1) then
6       {
7           write ("Stack is full!"); return false;
8       }
9       else
10      {
11          top := top + 1; stack[top] := item; return true;
12      }
13  }
```

```
1   Algorithm Delete(item)
2   // Pop the top element from the stack. Return true if successful
3   // else return false. item is used as an output.
4   {
5       if (top < 0) then
6       {
7           write ("Stack is empty!"); return false;
8       }
9       else
10      {
11          item := stack[top]; top := top − 1; return true;
12      }
13  }
```

Algorithm 2.1 Operations on a stack

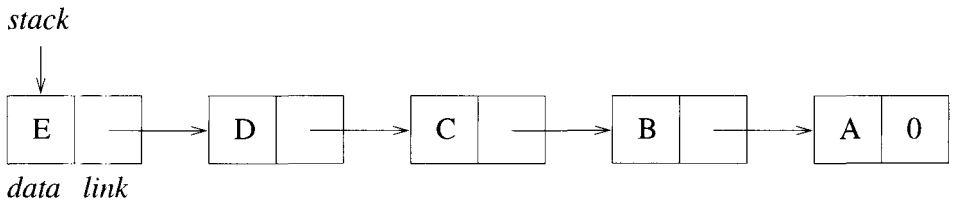

Figure 2.2 Example of a five-element, linked stack

```
// Type is the type of data.
node =record
{
    Type data; node *link;
}

1   Algorithm Add(item)
2   {
3       // Get a new node.
4       temp := new node;
5       if (temp ≠ 0) then
6       {
7           (temp → data) := item; (temp → link) := top;
8           top := temp; return true;
9       }
10      else
11      {
12          write ("Out of space!");
13          return false;
14      }
15  }

1   Algorithm Delete(item)
2   {
3       if (top = 0) then
4       {
5           write ("Stack is empty!");
6           return false;
7       }
8       else
9       {
10          item := (top → data); temp := top;
11          top := (top → link);
12          delete temp; return true;
13      }
14  }
```

Algorithm 2.2 Link representation of a stack

The variable *top* points to the topmost node (the last item inserted) in the list. The empty stack is represented by setting $top := 0$. Because of the way the links are pointing, insertion and deletion are easy to accomplish. See Algorithm 2.2.

In the case of Add, the statement *temp* := **new** *node*; assigns to the variable *temp* the address of an available node. If no more nodes exist, it returns 0. If a node exists, we store appropriate values into the two fields of the node. Then the variable *top* is updated to point to the new top element of the list. Finally, **true** is returned. If no more space exists, it prints an error message and returns **false**. Refering to Delete, if the stack is empty, then trying to delete an item produces the error message "Stack is empty!" and **false** is returned. Otherwise the top element is stored as the value of the variable *item*, a pointer to the first node is saved, and *top* is updated to point to the next node. The deleted node is returned for future use and finally **true** is returned.

The use of links to represent a stack requires more storage than the sequential array $stack[0 : n-1]$ does. However, there is greater flexibility when using links, for many structures can simultaneously use the same pool of available space. Most importantly the times for insertion and deletion using either representation are independent of the size of the stack.

An efficient queue representation can be obtained by taking an array $q[0 : n-1]$ and treating it as if it were circular. Elements are inserted by increasing the variable *rear* to the next free position. When $rear = n-1$, the next element is entered at $q[0]$ in case that spot is free. The variable $front$ always points one position counterclockwise from the first element in the queue. The variable $front = rear$ if and only if the queue is empty and we initially set $front := rear := 0$. Figure 2.3 illustrates two of the possible configurations for a circular queue containing the four elements J1 to J4 with $n > 4$.

To insert an element, it is necessary to move *rear* one position clockwise. This can be done using the code

if $(rear = n-1)$ **then** $rear := 0$;
else $rear := rear + 1$;

A more elegant way to do this is to use the built-in modulo operator which computes remainders. Before doing an insert, we increase the rear pointer by saying $rear := (rear + 1)$ **mod** n;. Similarly, it is necessary to move $front$ one position clockwise each time a deletion is made. An examination of Algorithm 2.3(a) and (b) shows that by treating the array circularly, addition and deletion for queues can be carried out in a fixed amount of time or $O(1)$.

One surprising feature in these two algorithms is that the test for queue full in AddQ and the test for queue empty in DeleteQ are the same. In the

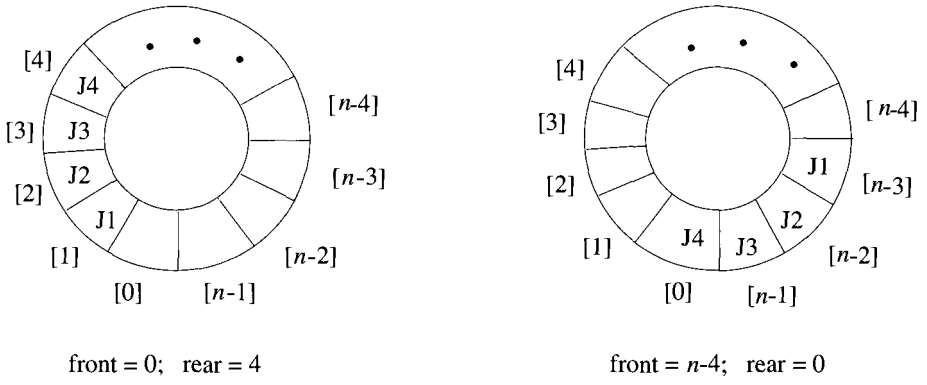

Figure 2.3 Circular queue of capacity $n - 1$ containing four elements J1, J2, J3, and J4

case of AddQ, however, when $front = rear$, there is actually one space free, $q[rear]$, since the first element in the queue is not at $q[front]$ but is one position clockwise from this point. However, if we insert an item there, then we cannot distinguish between the cases full and empty, since this insertion leaves $front = rear$. To avoid this, we signal queue full and permit a maximum of $n - 1$ rather than n elements to be in the queue at any time. One way to use all n positions is to use another variable, tag, to distinguish between the two situations; that is, $tag = 0$ if and only if the queue is empty. This however slows down the two algorithms. Since the AddQ and DeleteQ algorithms are used many times in any problem involving queues, the loss of one queue position is more than made up by the reduction in computing time.

Another way to represent a queue is by using links. Figure 2.4 shows a queue with the four elements A, B, C, and D entered in that order. As with the linked stack example, each node of the queue is composed of the two fields *data* and *link*. A queue is pointed at by two variables, $front$ and $rear$. Deletions are made from the front, and insertions at the rear. Variable $front = 0$ signals an empty queue. The procedures for insertion and deletion in linked queues are left as exercises.

EXERCISES

1. Write algorithms for AddQ and DeleteQ, assuming the queue is represented as a linked list.

```
1   Algorithm AddQ(item)
2   // Insert item in the circular queue stored in q[0 : n − 1].
3   // rear points to the last item, and front is one
4   // position counterclockwise from the first item in q.
5   {
6       rear := (rear + 1) mod n; // Advance rear clockwise.
7       if (front = rear) then
8       {
9           write ("Queue is full!");
10          if (front = 0) then rear := n − 1;
11          else rear := rear − 1;
12          // Move rear one position counterclockwise.
13          return false;
14      }
15      else
16      {
17          q[rear] := item; // Insert new item.
18          return true;
19      }
20  }
```

(a) Addition of an element

```
1   Algorithm DeleteQ(item)
2   // Removes and returns the front element of the queue q[0 : n − 1].
3   {
4       if (front = rear) then
5       {
6           write ("Queue is empty!");
7           return false;
8       }
9       else
10      {
11          front := (front + 1) mod n; // Advance front clockwise.
12          item := q[front]; // Set item to front of queue.
13          return true;
14      }
15  }
```

(b) Deletion of an element

Algorithm 2.3 Basic queue operations

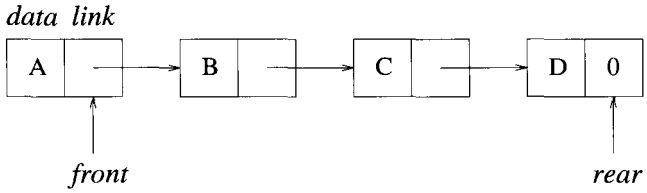

Figure 2.4 A linked queue with four elements

2. A linear list is being maintained circularly in an array $c[0 : n-1]$ with f and r set up as for circular queues.
   (a) Obtain a formula in terms of f, r, and n for the number of elements in the list.
   (b) Write an algorithm to delete the kth element in the list.
   (c) Write an algorithm to insert an element y immediately after the kth element.

   What is the time complexity of your algorithms for parts (b) and (c)?

3. Let $X = (x_1, \ldots, x_n)$ and $Y = (y_1, \ldots, y_m)$ be two linked lists. Write an algorithm to merge the two lists to obtain the linked list $Z = (x_1, y_1, x_2, y_2, \ldots, x_m, y_m, x_{m+1}, \ldots, x_n)$ if $m \leq n$ or $Z = (x_1, y_1, x_2, y_2, \ldots, x_n, y_n, y_{n+1}, \ldots, y_m)$ if $m > n$.

4. A double-ended queue (deque) is a linear list for which insertions and deletions can occur at either end. Show how to represent a deque in a one-dimensional array and write algorithms that insert and delete at either end.

5. Consider the hypothetical data object $X2$. The object $X2$ is a linear list with the restriction that although additions to the list can be made at either end, deletions can be made from one end only. Design a linked list representation for $X2$. Specify initial and boundary conditions for your representation.

2.2 TREES

Definition 2.1 [Tree] A *tree* is a finite set of one or more nodes such that there is a specially designated node called the *root* and the remaining nodes are partitioned into $n \geq 0$ disjoint sets $T_1, \ldots, T_n$, where each of these sets is a tree. The sets $T_1, \ldots, T_n$ are called the *subtrees* of the root. □