Chapter 3

STACKS AND QUEUES

3.1 FUNDAMENTALS

Two of the more common data objects found in computer algorithms are stacks and queues. They arise so often that we will discuss them separately before moving on to more complex objects. Both these data objects are special cases of the more general data object, an ordered list which we considered in the previous chapter. Recall that $A = (a_1, a_2, \ldots, a_n)$, is an ordered list of $n \geqq 0$ elements. The a_i are referred to as atoms or elements which are taken 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*, while all deletions take place at the other end, the *front*. Given a stack $S = (a_1, \ldots, a_n)$, we say that a_1 is the *bottomost* element and element a_i is on *top* of element a_{i-1}, $1 < i \leq n$. When viewed as a queue with a_n as the rear element one says that a_{i+1} is behind a_i, $1 \leq i < n$.

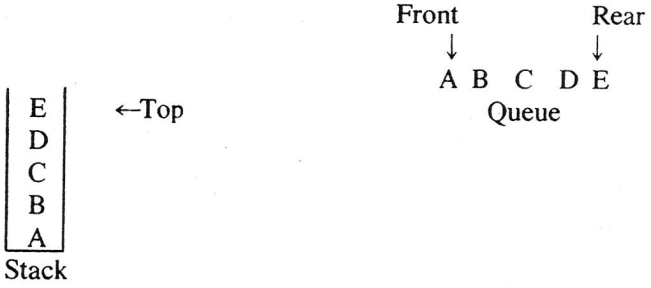

Figure 3.1.

The restrictions on a stack imply that if the elements A, B, C, D, E are added to the 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 will be 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 restrictions on a queue require that the first element which is inserted into the queue will be the first one to be removed. Thus A is the first letter to be removed, and queues are known as *F*irst *I*n *F*irst *O*ut (FIFO) lists. Note that the data object queue as defined here need not necessarily correspond to the mathematical concept of queue in which the insert/delete rules may be different.

One natural example of stacks which arises in computer programming is the processing of procedure calls and their terminations. Suppose we have four procedures as below:

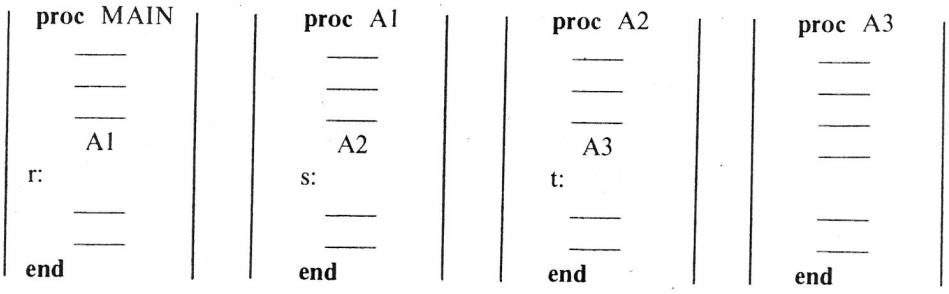

Figure 3.2 Sequence of subroutine calls.

The MAIN procedure invokes procedure *A*1. On completion of *A*1 execution of MAIN will resume at location *r*. The address *r* is passed to *A*1 which saves it in some location for later processing. *A*1 then invokes *A*2 which in turn invokes *A*3. In each case the invoking procedure passes the return address to the invoked procedure. If we examine the memory while *A*3 is computing there will be an implicit stack which looks like

$$(q,r,s,t).$$

The first entry, *q*, is the address to which MAIN returns control. This list operates as a stack since the returns will be made in the reverse order of the invocations. Thus *t* is removed before *s*, *s* before *r* and *r* before *q*. Equivalently, this means that *A*3 must finish processing before *A*2, *A*2 before *A*1, and *A*1 before MAIN. This list of return addresses need not be maintained in consecutive locations. For each procedure there is usually a single location associated with the machine code which is used to retain the return address. This can be severely limiting in the case of recursive and re-entrant procedures, since every time we invoke a procedure the new return address wipes out the old one. For example, if we inserted a call to *A*1 within procedure *A*3 expecting the return to be at location *u*, then at execution time the stack would become (q,u,s,t) and the return address *r* would be lost. When recursion is allowed, it is no longer adequate to reserve one location for the return address of each procedure. Since returns are made

in the reverse order of calls, an elegant and natural solution to this procedure return problem is afforded through the explicit use of a stack of return addresses. Whenever a return is made, it is to the top address in the stack. Implementing recursion using a stack is discussed in Section 4.9.

Associated with the object stack there are several operations that are necessary:

CREATE(S) which creates S as an empty stack;
ADD(i,S) which inserts the element i onto the stack S and returns the new stack;
DELETE(S) which removes the top element of stack S and returns the new stack;
TOP(S) which returns the top element of stack S;
ISEMTS(S) which returns true if S is empty else false;

These five functions constitute a working definition of a stack. However we choose to represent a stack, it must be possible to build these operations. But before we do this let us describe formally the structure STACK.

```
    structure STACK (item)
1     declare CREATE( ) → stack
2             ADD(item,stack) → stack
3             DELETE(stack) → stack
4             TOP(stack) → item
5             ISEMTS(stack) → boolean;
6     for all S ∈ stack, i ∈ item let
7         ISEMTS(CREATE)      ::= true
8         ISEMTS(ADD(i,S))    ::= false
9         DELETE(CREATE)      ::= error
10        DELETE(ADD(i,S))    ::= S
11        TOP(CREATE)         ::= error
12        TOP(ADD(i,S))       ::= i
13    end
    end STACK
```

The five functions with their domains and ranges are declared in lines 1 through 5. Lines 6 through 13 are the set of axioms which describe how the functions are related. Lines 10 and 12 are the essential ones which define the last-in-first-out behavior. The above definitions describe an infinite stack for no upper bound on the number of elements is specified. This will be dealt with when we represent this structure in a computer.

The simplest way to represent a stack is by using a one dimensional array, say *stack*[1..n], where n is the maximum number of allowable entries. The first or bottom element in the stack will be stored at *stack*[1], the second at *stack*[2] and the i-th at *stack*[i]. Associated with the array will

be a variable, *top*, which points to the top element in the stack. With this decision made the following implementations result:

```
CREATE(stack) ::= var stack: array[1..n] of items; top: 0..n;
                  top := 0;
ISEMTS(stack) ::= if top = 0 then true
                             else false;
TOP(stack) ::= if top = 0 then error
                          else stack[top];
```

The implementations of these three operations using an array are so short that we needn't make them separate procedures but can just use them directly whenever we need to. The ADD and DELETE operations are only a bit more complex. The corresponding procedures have been written assuming that *stack*, *top*, and *n* are global.

```
procedure add(item : items);
{add item to the global stack stack;
 top is the current top of stack
 and n is its maximum size}
begin
  if top = n then stackfull;
  top := top + 1;
  stack[top] := item;
end; {of add}
```

Program 3.1 Add to a stack

```
procedure delete(var item : items);
{remove top element from the stack stack and put it in item}
begin
  if top = 0 then stackempty;
  item := stack[top];
  top := top - 1;
end; {of delete}
```

Program 3.2 Delete from a stack

Programs 3.1 and 3.2 are so simple that they perhaps need no more explanation. Procedure *delete* actually combines the functions TOP and DELETE, *stackfull* and *stackempty* are procedures which we leave unspecified since they will depend upon the particular application. Often a *stackfull* condition will signal that more storage needs to be allocated

and the program re-run. *Stackempty* is often a meaningful condition. In Section 3.3 we will see a very important computer application of stacks where *stackempty* signals the end of processing.

The correctness of the stack implementation above may be established by showing that in this implementation, the stack axioms of lines 7-12 of the stack structure definition are true. Let us show this for the first three rules. The remainder of the axioms can be shown to hold similarly.

(i) line 7: *ISEMTS(CREATE)* ::= **true**

Since CREATE results in top being initialized to zero, it follows from the implementation of ISEMTS that *ISEMTS(CREATE)* :: = **true**.

(ii) line 8: *ISEMTS(ADD(i,S)* ::= **false**

The value of *top* is changed only in procedures CREATE, *add* and *delete*. CREATE initializes *top* to zero while *add* increments it by 1 so long as *top* is less than *n* (this is necessary because we can implement only a finite stack). *delete* decreases *top* by 1 but never allows its value to become less than zero. Hence, *add*(*i*) either results in an error condition (*stackfull*), or leaves the value of *top* $>$ 0. This then implies that *ISEMTS(ADD(i,s))* ::= **false**.

(iii) line 9: *DELETE(CREATE)* ::= *error*

This follows from the observation that CREATE sets *top* = 0 and the procedure *delete* signals the error condition *stackempty* when *top* = 0.

Queues, like stacks, also arise quite naturally in the computer solution of many problems. Perhaps the most common occurrence of a queue in computer applications is for the scheduling of jobs. In batch processing the jobs are "queued-up" as they are read-in and executed, one after another in the order they were received. This ignores the possible existence of priorities, in which case there will be one queue for each priority.

As mentioned earlier, when we talk of queues we talk about two distinct ends: the front and the rear. Additions to the queue take place at the rear. Deletions are made from the front. So, if a job is submitted for execution, it joins at the rear of the job queue. The job at the front of the queue is the next one to be executed. A minimal set of useful operations on a queue includes the following:

CREATEQ(Q)	which creates Q as an empty queue;
ADDQ(i,Q)	which adds the element i to the rear of a queue and returns the new queue;
DELETEQ(Q)	which removes the front element from the queue Q and returns the resulting queue;
FRONT(Q)	which returns the front element of Q;
ISEMTQ(Q)	which returns true if Q is empty else false.

A complete specification of this data structure is

```
    structure QUEUE (item)
 1    declare CREATEQ( ) → queue
 2            ADDQ(item,queue) → queue
 3            DELETEQ(queue) → queue
 4            FRONT(queue) → item
 5            ISEMTQ(queue) → boolean;
 6    for all Q ∈ queue, i ∈ item let
 7        ISEMTQ(CREATEQ)      ::= true
 8        ISEMTQ(ADD(i,Q))     ::= false
 9        DELETEQ(CREATEQ)     ::= error
10        DELETEQ(ADDQ(i,Q))   ::=
11          if ISEMTQ(Q) then CREATEQ
12                       else ADDQ(i, DELETEQ(Q))
13        FRONT(CREATEQ)       ::= error
14        FRONT(ADDQ(i,Q))     ::=
15          if ISEMTQ(Q) then i else FRONT(Q)
16    end
17  end QUEUE
```

The axiom of lines 10-12 shows that deletions are made from the front of the queue.

The representation of a finite queue in sequential locations is somewhat more difficult than a stack. In addition to a one dimensional array $q[1..n]$, we need two variables, *front* and *rear*. The conventions we shall adopt for these two variables are that *front* is always 1 less than the actual front of the queue and *rear* always points to the last element in the queue. Thus, *front* = *rear* if and only if there are no elements in the queue. The initial condition then is *front* = *rear* = 0. With these conventions, let us try an example by inserting and deleting jobs, J_i, from a job queue.

		Q[1]	[2]	[3]	[4]	[5]	[6]	[7]	... Remarks
front	rear								
0	0		queue		empty				Initial
0	1	J1							Job 1 joins Q
0	2	J1	J2						Job 2 joins Q
0	3	J1	J2	J3					Job 3 joins Q
1	3		J2	J3					Job 1 leaves Q
1	4		J2	J3	J4				Job 4 joins Q
2	4			J3	J4				Job 2 leaves Q

With this scheme, the following implementation of the CREATEQ, ISEMTQ, and FRONT operations results for a queue with capacity *n*:

```
CREATEQ(q)  ::= var q: array[1..n] of items; front, rear: 0..n;
                front := 0; rear := 0;
ISEMTQ(q)   ::= if front = rear then true
                                else false;
FRONT(q)    ::= if ISEMTQ(q) then error
                                else q[front + 1];
```

The procedures for ADDQ and DELETEQ are given as Programs 3.3 and 3.4.

```
procedure addq (item : items);
{add item to the queue q}
begin
  if rear = n then queuefull
              else begin
                     rear := rear + 1;
                     q[rear] := item;
                   end;
end; {of addq}
```

Program 3.3 Add to a queue

```
procedure deleteq (var item : items);
{delete from the front of q and put into item}
begin
  if front = rear then queueempty
                  else begin
                         front := front + 1;
                         item := q[front];
                       end;
end; {of deleteq}
```

Program 3.4 Delete from a queue

The correctness of this implementation may be established in a manner akin to that used for stacks. With this set up, notice that unless the front regularly catches up with the rear and both pointers are reset to zero, then the *queuefull* signal does not necessarily imply that there are n elements in the queue. That is, the queue will gradually move to the right. One obvious thing to do when *queuefull* is signaled is to move the entire queue to the left so that the first element is again at $q[1]$ and $front = 0$. This is time consuming, especially when there are many elements in the queue at the time of the *queuefull* signal.

Let us look at an example which shows what could happen, in the worst case, if each time the queue becomes full we choose to move the entire queue left so that it starts at $q[1]$. To begin, assume there are n elements J_1, ..., J_n in the queue and we next receive alternate requests to delete and add elements. Each time a new element is added, the entire queue of $n-1$ elements is moved left.

front	*rear*	*q*[1]	[2]	[3]	.....	[n]	next operation
0	n	J_1	J_2	J_3	.....	J_n	initial state
1	n		J_2	J_3	.....	J_n	delete J_1
0	n	J_2	J_3	J_4	.....	J_{n+1}	add J_{n+1} (jobs J_2 through J_n are moved)
1	n		J_3	J_4	.....	J_{n+1}	delete J_2
0	n	J_3	J_4	J_5	.....	J_{n+2}	add J_{n+2}

Figure 3.3.

A more efficient queue representation is obtained by regarding the array $q[1..n]$ as circular. It now becomes more convenient to declare the array as $q[0..n-1]$. When $rear = n - 1$, the next element is entered at $q[0]$ in case that spot is free. Using the same conventions as before, *front* will always point one position counterclockwise from the first element in the queue. Again, $front = rear$ if and only if the queue is empty. Initially we have $front = rear = 1$. Figure 3.4 illustrates some of the possible configurations for a circular queue containing the four elements *J*1-*J*4 with $n > 4$. The assumption of circularity changes the *addq* and *deleteq* procedures slightly. In order to add an element, it will be necessary to move *rear* one position clockwise, i.e.,

```
if rear = n - 1 then rear := 0
                else rear := rear + 1.
```

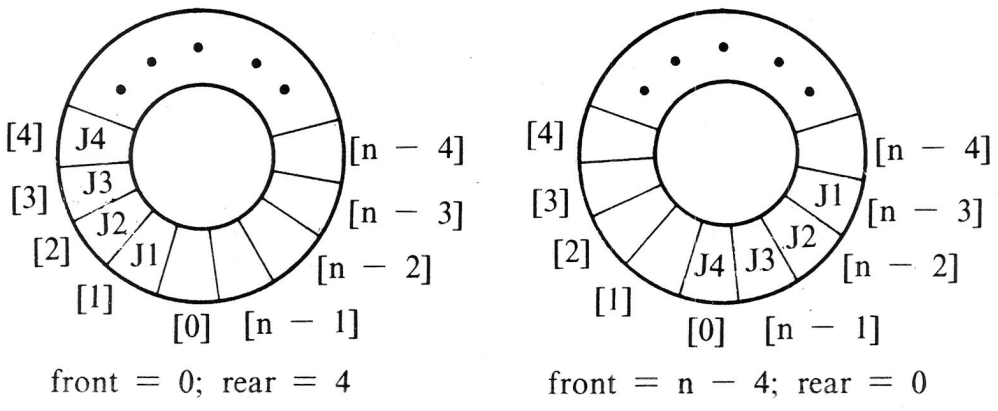

Figure 3.4 Circular queue of n elements and four jobs *J*1, *J*2, *J*3, *J*4.

Using the modulo operator which computes remainders, this is just *rear* := (*rear* + 1)**mod** *n*. Similarly, it will be necessary to move *front* one position clockwise each time a deletion is made. Again, using the modulo operator, this can be accomplished by *front* := (*front* + 1)**mod** *n*. An examination of the algorithms (Programs 3.5 and 3.6) indicates that addition and deletion can now be carried out in a fixed amount of time or $O(1)$.

```
procedure addq (item : items);
{insert item into the circular queue stored in q[0..n - 1]}
begin
  rear := (rear + 1)mod n; {advance rear clockwise}
  if front = rear then queuefull;
  q[rear] := item; {insert}
end; {of addq}
```

Program 3.5 Add to a circular queue

```
procedure deleteq (var item : items);
{remove front element from q and put into item}
begin
  if front = rear then queueempty;
  front := (front + 1)mod n; {advance front clockwise}
  item := q[front];
end; {of deleteq}
```

Program 3.6 Delete from a circular queue

One surprising point in the two algorithms is that the test for queue full in *addq* and the test for queue empty in *deleteq* are the same. In the case of *addq*, however, when *front* = *rear* there is actually one space free, i.e. *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 here, then we will not be able to distinguish between the cases full and empty, since this insertion would leave *front* = *rear*. To avoid this, we signal *queuefull*, thus permitting a maximum of $n - 1$ rather than n elements to be in the queue at any time. One way to use all n positions would be to use another variable, *tag*, to distinguish between the two situations, i.e. *tag* = 0 if and only if the queue is empty. This would, however, slow down the two procedures. Since the *addq* and *deleteq* procedures will be used many times in any problem involving queues, the loss of one queue position will be more than made up for by the reduction in computing time.

The procedures *queuefull* and *queueempty* have been used without explanation, but they are similar to *stackfull* and *stackempty*. Their function will depend on the particular application.

Second Edition

FUNDAMENTALS OF

Data Structures in Pascal

ELLIS HOROWITZ • SARTAJ SAHNI

University of Southern California *University of Minnesota*

COMPUTER SCIENCE PRESS