3

STACKS, QUEUES AND LINKED LISTS

Unit Structure:

3.1 Abstract Data Types (ADTS)
3.2 The List Abstract Data Type
3.3 The Stack ADT
3.4 The Queue ADT

3.1 ABSTRACT DATA TYPES (ADTS):

One of the basic rules concerning programming is that no routine should ever exceed a page. This is accomplished by breaking the program down into *modules*. Each module is a logical unit and does a specific job. Its size is kept small by calling other modules. Modularity has several advantages.

(a) it is much easier to debug small routines than large routines.

(b) it is easier for several people to work on a modular program simultaneously.

(c) a well-written modular program places certain dependencies in only one routine, making changes easier.

For instance, if output needs to be written in a certain format, it is certainly important to have one routine to do this. If printing statements are scattered throughout the program, it will take considerably longer to make modifications.

The idea that global variables and side effects are bad is directly attributable to the idea that modularity is good.

An *abstract data type* (ADT) is a set of operations. Abstract data types are mathematical abstractions; nowhere in an ADT's definition is there any mention of *how* the set of operations is implemented. This can be viewed as an extension of modular design.

Objects such as lists, sets, and graphs, along with their operations, can be viewed as abstract data types, just as integers, reals, and booleans are data types. Integers, reals, and booleans have operations associated with them, and so do abstract data

types. For the set ADT, we might have such operations as *union, intersection, size*, and *complement*. Alternately, we might only want the two operations *union* and *find,* which would define a different ADT on the set.

The basic idea is that the implementation of these operations is written once in the program, and any other part of the program that needs to perform an operation on the ADT can do so by calling the appropriate function. If for some reason implementation details need to change, it should be easy to do so by merely changing the routines that perform the ADT operations. This change, in a perfect world, would be completely transparent to the rest of the program.

There is no rule telling us which operations must be supported for each ADT; this is a design decision. Error handling and tie breaking (where appropriate) are also generally up to the program designer. The three data structures that we will study in this chapter are primary examples of ADTs. We will see how each can be implemented in several ways, but if they are done correctly, the programs that use them will not need to know which implementation was used.

3.2 THE LIST ABSTRACT DATA TYPE:

We will deal with a general list of the form $a_1, a_2, a_3, \ldots, a_n$. We say that the size of this list is n. We will call the special list of size 0 a *null list*.

For any list except the null list, we say that a_{i+1} follows (or succeeds) a_i ($i < n$) and that a_{i-1} precedes a_i ($i > 1$). The first element of the list is a_1, and the last element is a_n. We will not define the predecessor of a_1 or the successor of a_n. The *position* of element a_i in a list is i. Throughout this discussion, we will assume, to simplify matters, that the elements in the list are integers, but in general, arbitrarily complex elements are allowed.

Associated with these "definitions" is a set of operations that we would like to perform on the list ADT. Some popular operations are

(a) *print_list* and *make_null*, which do the obvious things;

(b) *find*, which returns the position of the first occurrence of a key;

(c) *insert* and *delete*, which generally insert and delete some key from some position in the list; and

(d) *find_kth*, which returns the element in some position (specified as an argument).

If the list is 34, 12, 52, 16, 12, then *find*(52) might return 3; *insert*(*x*,3) might make the list into 34, 12, 52, *x*, 16, 12 (if we insert

after the position given); and *delete*(3) might turn that list into 34, 12, *x*, 16, 12.

Of course, the interpretation of what is appropriate for a function is entirely up to the programmer, as is the handling of special cases (for example, what does *find*(1) return above?). We could also add operations such as *next* and *previous*, which would take a position as argument and return the position of the successor and predecessor, respectively.

3.2.1. Simple Array Implementation of Lists

Obviously all of these instructions can be implemented just by using an array. Even if the array is dynamically allocated, an estimate of the maximum size of the list is required. Usually this requires a high over-estimate, which wastes considerable space. This could be a serious limitation, especially if there are many lists of unknown size.

An array implementation allows *print_list* and *find* to be carried out in linear time, which is as good as can be expected, and the *find_kth* operation takes constant time. However, insertion and deletion are expensive. For example, inserting at position 0 (which amounts to making a new first element) requires first pushing the entire array down one spot to make room, whereas deleting the first element requires shifting all the elements in the list up one, so the worst case of these operations is $O(n)$. On average, half the list needs to be moved for either operation, so linear time is still required. Merely building a list by n successive inserts would require quadratic time.

Because the running time for insertions and deletions is so slow and the list size must be known in advance, simple arrays are generally not used to implement lists.

3.2.2. Linked Lists

In order to avoid the linear cost of insertion and deletion, we need to ensure that the list is not stored contiguously, since otherwise entire parts of the list will need to be moved. Figure 3.1 shows the general idea of a *linked list*.

The linked list consists of a series of structures, which are not necessarily adjacent in memory. Each structure contains the element and a pointer to a structure containing its successor. We call this the *next* pointer. The last cell's *next* pointer points to *; this value is defined by C and cannot be confused with another pointer. ANSI C specifies that* is zero.

Recall that a pointer variable is just a variable that contains the address where some other data is stored. Thus, if *p* is declared to be a pointer to a structure, then the value stored in *p* is interpreted as the location, in main memory, where a structure can be found. A field of that structure can be accessed by *p->field_name*, where *field_name* is the name of the field we wish to examine. Figure 3.2 shows the actual representation of the list in Figure 3.1. The list contains five structures, which happen to reside in memory locations 1000, 800, 712, 992, and 692 respectively. The *next* pointer in the first structure has the value 800, which provides the indication of where the second structure is. The other structures each have a pointer that serves a similar purpose. Of course, in order to access this list, we need to know where the first cell can be found. A pointer variable can be used for this purpose. It is important to remember that a pointer is just a number. For the rest of this chapter, we will draw pointers with arrows, because they are more illustrative.

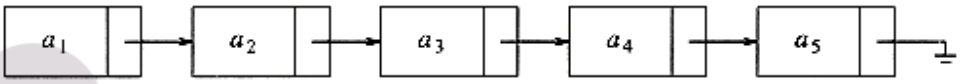

Figure 3.1 A linked list

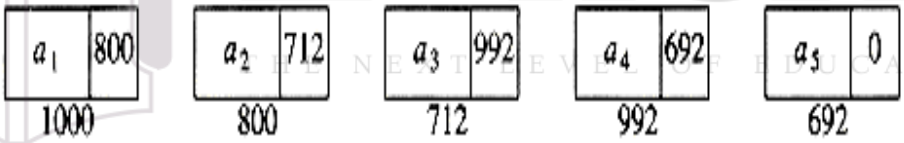

Figure 3.2 Linked list with actual pointer values

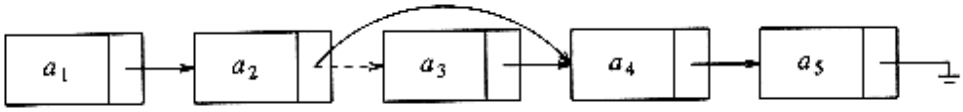

Figure 3.3 Deletion from a linked list

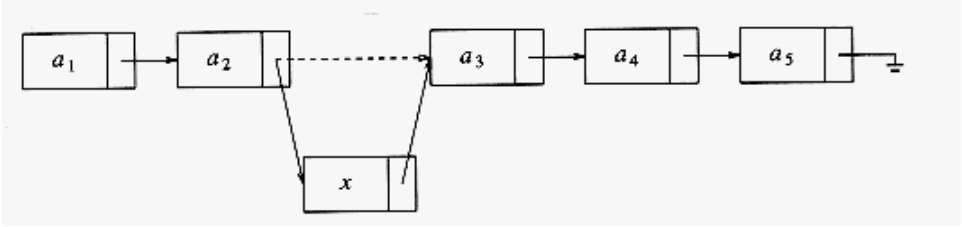

Figure 3.4 Insertion into a linked list

To execute *print_list(L)* or *find(L,key)*, we merely pass a pointer to the first element in the list and then traverse the list by following the *next* pointers. This operation is clearly linear-time, although the constant is likely to be larger than if an array implementation were used. The *find_kth* operation is no longer quite as efficient as an array implementation; *find_kth(L,i)* takes $O(i)$ time and works by traversing down the list in the obvious manner. In practice, this bound is pessimistic, because frequently the calls to *find_kth* are in sorted order (by i). As an example, *find_kth(L,2), find_kth(L,3), find_kth(L,4), find_kth(L,6)* can all be executed in one scan down the list.

The *delete* command can be executed in one pointer change. Figure 3.3 shows the result of deleting the third element in the original list.

The *insert* command requires obtaining a new cell from the system by using an *malloc* call (more on this later) and then executing two pointer maneuvers. The general idea is shown in Figure 3.4. The dashed line represents the old pointer.

3.2.3. Programming Linked Lists

The description above is actually enough to get everything working, but there are several places where you are likely to go wrong. First of all, there is no really obvious way to insert at the front of the list from the definitions given. Second, deleting from the front of the list is a special case, because it changes the start of the list; careless coding will lose the list. A third problem concerns deletion in general. Although the pointer moves above are simple, the deletion algorithm requires us to keep track of the cell *before* the one that we want to delete.

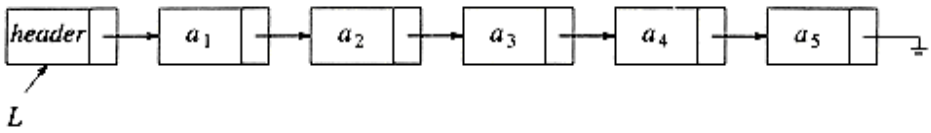

Figure 3.5 Linked list with a header

It turns out that one simple change solves all three problems. We will keep a sentinel node, which is sometimes referred to as a *header* or *dummy* node. This is a common practice, which we will see several times in the future. Our convention will be that the header is in position 0. Figure 3.5 shows a linked list with a header representing the list $a_1, a_2, \ldots, a_5$.

To avoid the problems associated with deletions, we need to write a routine *find_previous*, which will return the position of the predecessor of the cell we wish to delete. If we use a header, then

if we wish to delete the first element in the list, *find_previous* will return the position of the header. The use of a header node is somewhat controversial. Some people argue that avoiding special cases is not sufficient justification for adding fictitious cells; they view the use of header nodes as little more than old-style hacking. Even so, we will use them here, precisely because they allow us to show the basic pointer manipulations without obscuring the code with special cases. Otherwise, whether or not a header should be used is a matter of personal preference.

As examples, we will write about half of the list ADT routines. First, we need our declarations, which are given in **Figure 3.6.**

```
typedef struct node *node_ptr;
struct node
{
        element_type element;
        node_ptr next;
};
typedef node_ptr LIST;
typedef node_ptr position;
```

Figure 3.6 Type declarations for linked lists

The first function that we will write tests for an empty list. When we write code for any data structure that involves pointers, it is always best to draw a picture first. Figure 3.7 shows an empty list;

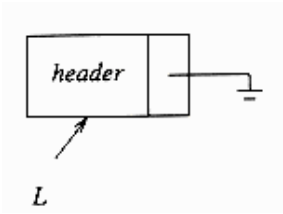

Figure 3.7 Empty list with header

from the figure it is easy to write the function in Figure 3.8.

```
int
is_empty( LIST L )
{
        return( L->next == NULL );
}
```

Figure 3.8

The next function, which is shown in Figure 3.9, tests whether the current element, which by assumption exists, is the last of the list.

```
int
is_last( position p, LIST L )
{
        return( p->next == NULL );
}
```

Figure 3.9 Function to test whether current position is the last in a linked list

The next routine we will write is *find*. *Find*, shown in Figure 3.10, returns the position in the list of some element. Line 2 takes advantage of the fact that the *and* (&&) operation is *short-circuited:* if the first half of the *and* is false, the result is automatically false and the second half is not executed.

```
/* Return position of x in L; NULL if not found */
position
find ( element_type x, LIST L )
{
position p;
/*1*/       p = L->next;
/*2*/       while( (p != NULL) && (p->element != x) )
/*3*/       p = p->next;
/*4*/       return p;
}
```

Figure 3.10 Find routine

Our fourth routine will delete some element *x* in list *L*. We need to decide what to do if *x* occurs more than once or not at all. Our routine deletes the first occurrence of *x* and does nothing if *x* is not in the list. To do this, we find *p*, which is the cell prior to the one containing *x*, via a call to *find_previous*. The code to implement this is shown in Figure 3.11.

```
/* Delete from a list. Cell pointed */
/* to by p->next is wiped out. */
/* Assume that the position is legal. */
/* Assume use of a header node. */
void
delete( element_type x, LIST L )
{
        position p, tmp_cell;
        p = find_previous( x, L );
        if( p->next != NULL )  /* Implicit assumption of header
                                  use */
        {                      /* x is found: delete it */
                tmp_cell = p->next;
                p->next = tmp_cell->next; /* bypass the cell to be
                                             deleted */
                free( tmp_cell );
        }
}
```

Figure 3.11 Deletion routine for linked lists

The *find_previous* routine is similar to *find* and is shown in Figure 3.12.

```
/* Uses a header. If element is not found, then next field */
/* of returned value is NULL */
position
find_previous( element_type x, LIST L )
{
  position p;
  p = L;
  while( (p->next != NULL) && (p->next->element != x) )
       p = p->next;
  return p;
}
```

Figure 3.12 Find_previous--the find routine for use with delete

The last routine we will write is an insertion routine. We will pass an element to be inserted along with the list *L* and a position

p. Our particular insertion routine will insert an element *after* the position implied by *p*. This decision is arbitrary and meant to show that there are no set rules for what insertion does. It is quite possible to insert the new element into position *p* (which means before the element currently in position *p*), but doing this requires knowledge of the element before position *p.* This could be obtained by a call to *find_previous*. It is thus important to comment what you are doing. This has been done in Figure 3.13.

```
/* Insert (after legal position p).*/
/* Header implementation assumed. */
void
insert( element_type x, LIST L, position p )
{
        position tmp_cell;
    tmp_cell = (position) malloc( sizeof (struct node) );
      if( tmp_cell == NULL )

          fatal_error("Out of space!!!");
            else
          {
          tmp_cell->element = x;
          tmp_cell->next = p->next;
          p->next = tmp_cell;
          }
}
```

Figure 3.13 Insertion routine for linked lists

Notice that we have passed the list to the *insert* and *is_last* routines, even though it was never used. We did this because another implementation might need this information, and so not passing the list would defeat the idea of using ADTs.
We could write additional routines to print a list and to perform the *next* function. These are fairly straightforward. We could also write a routine to implement *previous*.

3.2.4. Common Errors

The most common error that you will get is that your program will crash with a nasty error message from the system, such as "memory access violation" or "segmentation violation." This message usually means that a pointer variable contained a bogus address. One common reason is failure to initialize the variable. For

instance, if line (p = L->next;) in Figure 3.14 is omitted, then *p* is undefined and is not likely to be pointing at a valid part of memory. Another typical error would be line (p->next = tmp_cell;) in Figure 3.13. If *p* is *, then the indirection is illegal. This function knows that* p *is not* , so the routine is OK. Of course, you should comment this so that the routine that calls *insert* will insure this. *Whenever you do an indirection, you must make sure that the pointer is not NULL*. Some C compliers will implicity do this check for you, but this is not part of the C standard. When you port a program from one compiler to another, you may find that it no longer works. This is one of the common reasons why.

```
void
delete_list( LIST L )
{
position p;
        p = L->next;        /* header assumed */
        L->next = NULL;
        while( p != NULL )
            {
            free( p );
            p = p->next;
            }
}
```

Figure 3.14 Incorrect way to delete a list

The second common mistake concerns when and when not to use *malloc* to get a new cell. You must remember that declaring a pointer to a structure does not create the structure but only gives enough space to hold the address where some structure might be. The only way to create a record that is not already declared is to use the *malloc* command. The command *malloc*(*size_p*) has the system create, magically, a new structure and return a pointer to it. If, on the other hand, you want to use a pointer variable to run down a list, there is no need to declare a new structure; in that case the *malloc* command is inappropriate. A type cast is used to make both sides of the assignment operator compatible. The C library provides other variations of *malloc* such as *calloc*.

When things are no longer needed, you can issue a *free* command to inform the system that it may reclaim the space. A consequence of the *free(p)* command is that the address that *p* is pointing to is unchanged, but the data that resides at that address is now undefined.

If you never delete from a linked list, the number of calls to *malloc* should equal the size of the list, plus 1 if a header is used. Any less, and you cannot possibly have a working program. Any

more, and you are wasting space and probably time. Occasionally, if your program uses a lot of space, the system may be unable to satisfy your request for a new cell. In this case a *pointer is returned.*

After a deletion in a linked list, it is usually a good idea to free the cell, especially if there are lots of insertions and deletions intermingled and memory might become a problem. You need to keep a temporary variable set to the cell to be disposed of, because after the pointer moves are finished, you will not have a reference to it. As an example, the code in Figure 3.14 is not the correct way to delete an entire list (although it may work on some systems).

Figure 3.15 shows the correct way to do this. One last warning: *malloc*(*sizeof node_ptr*) is legal, but it doesn't allocate enough space for a structure. It allocates space only for a pointer.

```
void
delete_list( LIST L )
{
position p, tmp;
        p = L->next;   /* header assumed */
        L->next = NULL;
        while( p != NULL )
                {
                tmp = p->next;
                free( p );
                p = tmp;
                }
}
```

Figure 3.15 Correct way to delete a list

3.2.5. Doubly Linked Lists

Sometimes it is convenient to traverse lists backwards. The standard implementation does not help here, but the solution is simple. Merely add an extra field to the data structure, containing a pointer to the previous cell. The cost of this is an extra link, which adds to the space requirement and also doubles the cost of insertions and deletions because there are more pointers to fix. On the other hand, it simplifies deletion, because you no longer have to refer to a key by using a pointer to the previous cell; this information is now at hand. Figure 3.16 shows a doubly linked list.

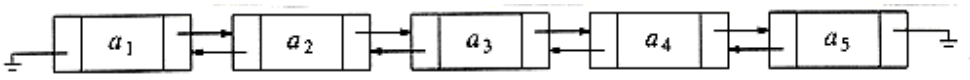

Figure 3.16 A doubly linked list

3.2.5. Doubly Linked Lists

Sometimes it is convenient to traverse lists backwards. The standard implementation does not help here, but the solution is simple. Merely add an extra field to the data structure, containing a pointer to the previous cell. The cost of this is an extra link, which adds to the space requirement and also doubles the cost of insertions and deletions because there are more pointers to fix. On the other hand, it simplifies deletion, because you no longer have to refer to a key by using a pointer to the previous cell; this information is now at hand. Figure 3.16 shows a doubly linked list.

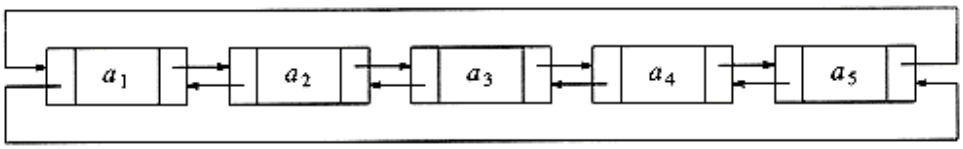

Figure 3.17 A double circularly linked list

3.2.5. Multi-Linked Lists

In a general multi-linked list each node can have any number of pointers to other nodes, and there may or may not be inverses for each pointer.

A university with 40,000 students and 2,500 courses needs to be able to generate two types of reports. The first report lists the class registration for each class, and the second report lists, by student, the classes that each student is registered for.

The obvious implementation might be to use a two-dimensional array. Such an array would have 100 million entries. The average student registers for about three courses, so only 120,000 of these entries, or roughly 0.1 percent, would actually have meaningful data.

What is needed is a list for each class, which contains the students in the class. We also need a list for each student, which contains the classes the student is registered for. Figure 3.18 shows our implementation.

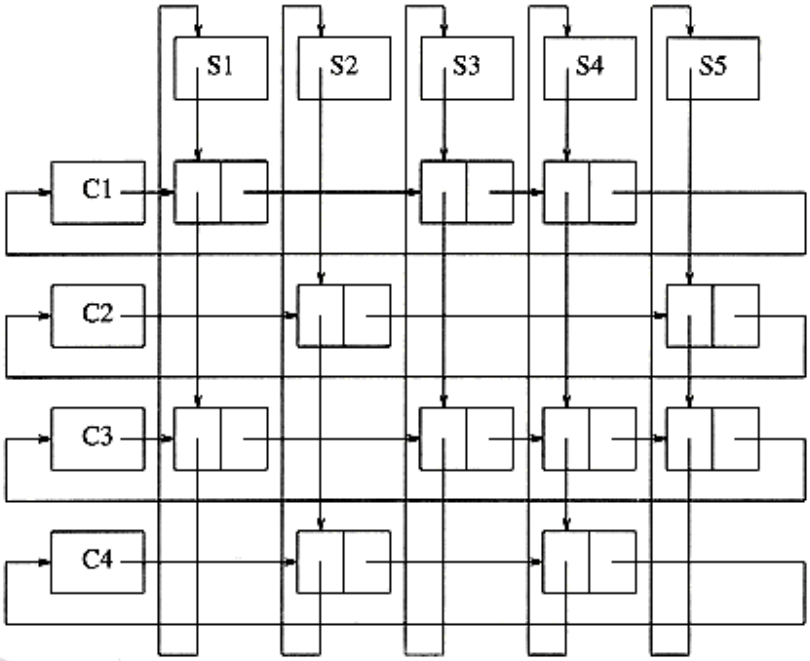

Figure 3.18 Multilinkedlist implementation for registration problem

As the figure shows, we have combined two lists into one. All lists use a header and are circular. To list all of the students in class C3, we start at C3 and traverse its list (by going right). The first cell belongs to student S1. Although there is no explicit information to this effect, this can be determined by following the student's linked list until the header is reached. Once this is done, we return to C3's list (we stored the position we were at in the course list before we traversed the student's list) and find another cell, which can be determined to belong to S3. We can continue and find that S4 and S5 are also in this class. In a similar manner, we can determine, for any student, all of the classes in which the student is registered.

Using a circular list saves space but does so at the expense of time. In the worst case, if the first student was registered for every course, then every entry would need to be examined in order to determine all the course names for that student. Because in this application there are relatively few courses per student and few students per course, this is not likely to happen. If it were suspected that this could cause a problem, then each of the (nonheader) cells could have pointers directly back to the student and class header. This would double the space requirement, but simplify and speed up the implementation.

Check your Progress

1. Write a program to print the elements of a singly linked list.
2. Write a program to swap two adjacent elements by adjusting only the pointers (and not the data) using
   a. singly linked lists,
   b. doubly linked lists
3. Write a function to add two polynomials. Do not destroy the input. Use a linked list implementation. If the polynomials have *m* and *n* terms respectively.

3.3. THE STACK ADT

A stack is a last in, first out (LIFO) abstract data type and data structure. A stack can have any abstract data type as an element, but is characterized by only two fundamental operations: push and pop. The push operation adds to the top of the list, hiding any items already on the stack, or initializing the stack if it is empty. The pop operation removes an item from the top of the list, and returns this value to the caller. A pop either reveals previously concealed items, or results in an empty list.

A stack is a *restricted data structure*, because only a small number of operations are performed on it. The nature of the pop and push operations also means that stack elements have a natural order. Elements are removed from the stack in the reverse order to the order of their addition: therefore, the lower elements are typically those that have been in the list the longest.

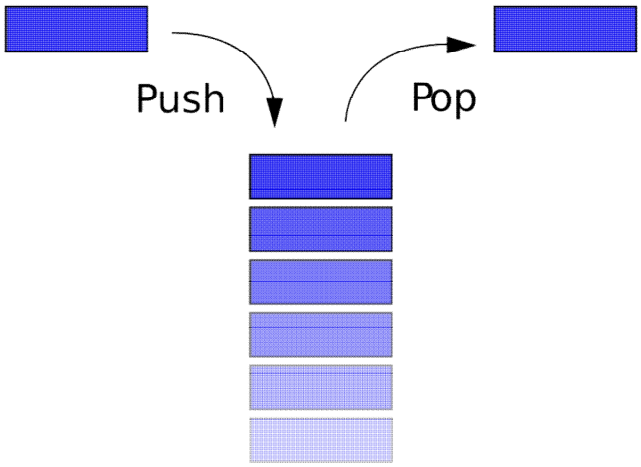

Figure 3.19 Simple representation of stack

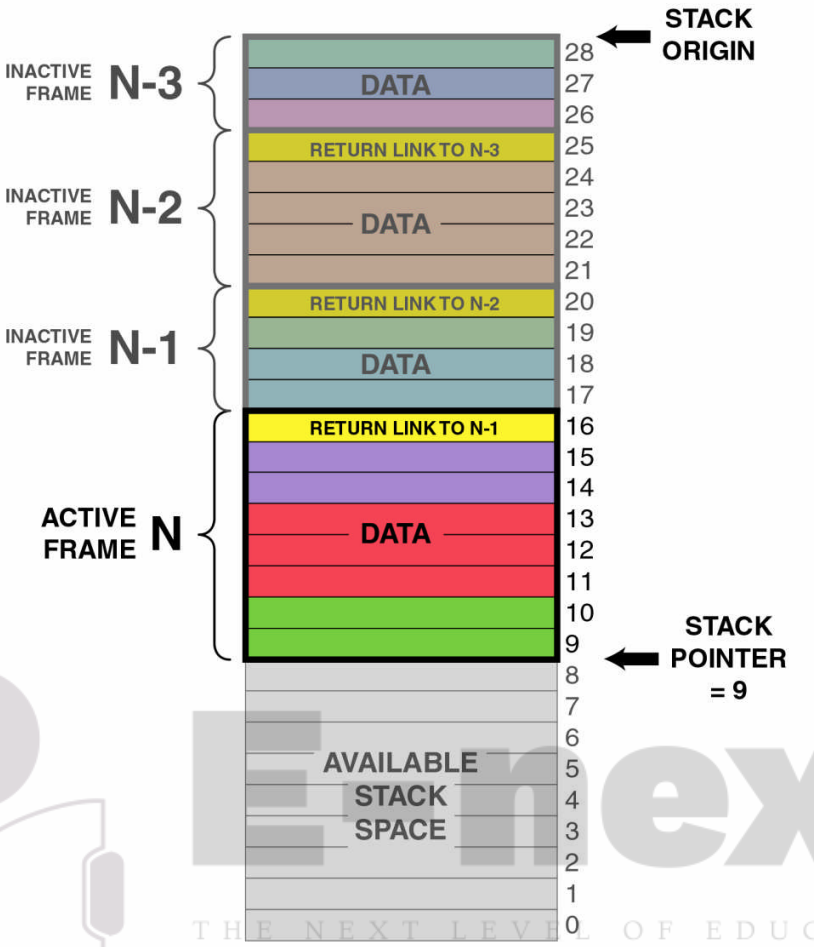

Figure 3.20 Basic Architecture of stack

3.3.1. Stack Model

A *stack* is a list with the restriction that *inserts* and *deletes* can be performed in only one position, namely the end of the list called the *top*. The fundamental operations on a stack are *push*, which is equivalent to an insert, and *pop*, which deletes the most recently inserted element. The most recently inserted element can be examined prior to performing a *pop* by use of the *top* routine. A *pop* or *top* on an empty stack is generally considered an error in the stack ADT. On the other hand, running out of space when performing a *push* is an implementation error but not an ADT error.

Stacks are sometimes known as LIFO (last in, first out) lists. The model depicted in Figure 3.21 signifies only that *pushes* are input operations and *pops* and *tops* are output. The usual operations to make empty stacks and test for emptiness are part of the repertoire, but essentially all that you can do to a stack is *push* and *pop*.

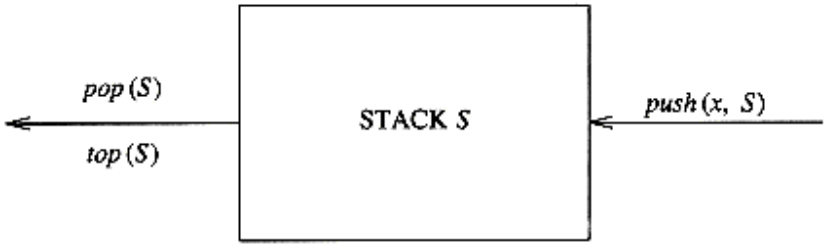

Figure 3.21 Stack model: input to a stack is by push, output is by pop

Figure 3.22 shows an abstract stack after several operations. The general model is that there is some element that is at the top of the stack, and it is the only element that is visible.

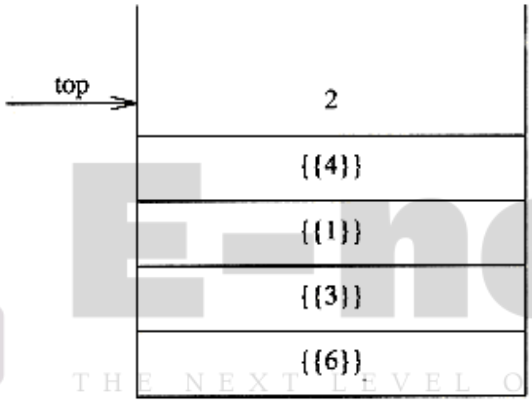

Figure 3.22 Stack model: only the top element is accessible

3.3.2. Implementation of Stacks

Two implementation of stacks are discussed here. Linked list implementation and array implementation.

3.3.2.1 Linked List Implementation of Stacks

The first implementation of a stack uses a singly linked list. We perform a *push* by inserting at the front of the list. We perform a *pop* by deleting the element at the front of the list. A *top* operation merely examines the element at the front of the list, returning its value. Sometimes the *pop* and *top* operations are combined into one. We could use calls to the linked list routines of the previous section, but we will rewrite the stack routines from scratch for the sake of clarity.

First, we give the definitions in Figure 3.23. We implement the stack using a header.

```
typedef struct node *node_ptr;
struct node
{
        element_type element;
        node_ptr next;
};
typedef node_ptr STACK;
```

Figure 3.23 Type declaration for linked list implementation of the stack ADT

Then Figure 3.24 shows that an empty stack is tested for in the same manner as an empty list.

```
int
is_empty( STACK S )
{
        return( S->next == NULL );
}
```

Figure 3.24 Routine to test whether a stack is empty-linked list implementation

Creating an empty stack is also simple. We merely create a header node; *make_null* sets the *next* pointer to *NULL* (see Fig. 3.25).

```
STACK
create_stack( void )
{
        STACK S;
        S = (STACK) malloc( sizeof( struct node ) );
        if( S == NULL )
                fatal_error("Out of space!!!");
        return S;
}
void
make_null( STACK S )
{
        if( S != NULL )
                S->next = NULL;
        else
                error("Must use create_stack first");
}
```

Figure 3.25 Routine to create an empty stack-linked list implementation

The *push* is implemented as an insertion into the front of a linked list, where the front of the list serves as the top of the stack (see Fig. 3.26).

```
void
push( element_type x, STACK S )
{
        node_ptr tmp_cell;
        tmp_cell = (node_ptr) malloc( sizeof ( struct node ) );
        if( tmp_cell == NULL )
                fatal_error("Out of space!!!");
        else
        {
                tmp_cell->element = x;
                tmp_cell->next = S->next;
                S->next = tmp_cell;
        }
}
```

Figure 3.26 Routine to push onto a stack-linked list implementation

The *top* is performed by examining the element in the first position of the list (see Fig. 3.27).

```
element_type
top( STACK S )
{
        if( is_empty( S ) )
                error("Empty stack");
        else
                return S->next->element;
}
```

Figure 3.27 Routine to return top element in a stack--linked list implementation

Finally, we implement *pop* as a delete from the front of the list (see Fig. 3.28).

```
void
pop( STACK S )
{
        node_ptr first_cell;
        if( is_empty( S ) )
                error("Empty stack");
        else
        {
                first_cell = S->next;
                S->next = S->next->next;
                free( first_cell );
        }
}
```

Figure 3.28 Routine to pop from a stack--linked list implementation

The drawback of this implementation is that the calls to *malloc* and *free* are expensive, especially in comparison to the pointer manipulation routines. Some of this can be avoided by using a second stack, which is initially empty. When a cell is to be disposed from the first stack, it is merely placed on the second stack. Then, when new cells are needed for the first stack, the second stack is checked first.

One problem that affects the efficiency of implementing stacks is error testing. Our linked list implementation carefully checked for errors. A *pop* on an empty stack or a *push* on a full stack will overflow the array bounds and cause a crash. This is undesirable, but if checks for these conditions were put in the array implementation, they would likely take as much time as the actual stack manipulation. For this reason, it has become a common practice to skip on error checking in the stack routines, except where error handling is crucial (as in operating systems). Although you can probably get away with this in most cases by declaring the stack to be large enough not to overflow and ensuring that routines that use *pop* never attempt to *pop* an empty stack, this can lead to code that barely works at best, especially when programs get large and are written by more than one person or at more than one time. Because stack operations take such fast constant time, it is rare that a significant part of the running time of a program is spent in these routines. This means that it is generally not justifiable to omit error checks. You should always write the error checks; if they are redundant, you can always comment them out if they really cost too much time.

3.3.2.1 Array implementation of Stacks

An alternative implementation avoids pointers and is probably the more popular solution. The only potential hazard with this strategy is that we need to declare an array size ahead of time. Generally this is not a problem, because in typical applications, even if there are quite a few stack operations, the actual number of elements in the stack at any time never gets too large. It is usually easy to declare the array to be large enough without wasting too much space. If this is not possible, then a safe course would be to use a linked list implementation.

If we use an array implementation, the implementation is trivial. Associated with each stack is the top of stack, *tos*, which is -1 for an empty stack (this is how an empty stack is initialized). To push some element *x* onto the stack, we increment *tos* and then set *STACK*[*tos*] = *x*, where *STACK* is the array representing the actual stack. To pop, we set the return value to *STACK*[*tos*] and then decrement *tos*. Of course, since there are potentially several stacks, the *STACK* array and *tos* are part of one structure representing a stack. It is almost always a bad idea to use global variables and fixed names to represent this (or any) data structure, because in most real-life situations there will be more than one stack. When writing your actual code, you should attempt to follow the model as closely as possible, so that no part of your code, except for the stack routines, can attempt to access the array or top-of-stack variable implied by each stack. This is true for *all* ADT operations.

A *STACK* is defined in Figure 3.29 as a pointer to a structure. The structure contains the *top_of_stack* and *stack_size* fields.

```
struct stack_record
{
        unsigned int stack_size;
        int top_of_stack;
        element_type *stack_array;
};
typedef struct stack_record *STACK;
#define EMPTY_TOS (-1) /* Signifies an empty stack */
```

Figure 3.29 STACK definition--array implementaion

Once the maximum size is known, the stack array can be dynamically allocated. Figure 3.30 creates a stack of a given maximum size. Lines 3-5 allocate the stack structure, and lines 6-8 allocate the stack array. Lines 9 and 10 initialize the *top_of_stack* and *stack_size* fields. The stack array does not need to be initialized. The stack is returned at line 11.

```
STACK
create_stack( unsigned int max_elements )
{
STACK S;
/*1*/       if( max_elements < MIN_STACK_SIZE )
/*2*/           error("Stack size is too small");
/*3*/       S = (STACK) malloc(sizeof(struct stack_record) );
/*4*/       if( S == NULL )
/*5*/           fatal_error("Out of space!!!");
/*6*/       S->stack_array = (element_type *)
                        malloc( sizeof( element_type ) * max_elements );
/*7*/       if( S->stack_array == NULL )
/*8*/           fatal_error("Out of space!!!");
/*9*/       S->top_of_stack = EMPTY_TOS;
/*10*/      S->stack_size = max_elements;
/*11*/      return( S );
}
```

Figure 3.30 Stack creation--array implementaion

The routine *dispose_stack* should be written to free the stack structure. This routine first frees the stack array and then the stack structure (See Figure 3.31). Since *create_stack* requires an argument in the array implementation, but not in the linked list implementation, the routine that uses a stack will need to know which implementation is being used unless a dummy parameter is added for the later implementation. Unfortunately, efficiency and software idealism often create conflicts.

```
void
dispose_stack( STACK S )
{
        if( S != NULL )
        {
                free( S->stack_array );
                free( S );
        }
}
```

Figure 3.31 Routine for freeing stack--array implementation

We have assumed that all stacks deal with the same type of element.

We will now rewrite the four stack routines. In true ADT spirit, we will make the function and procedure heading look

identical to the linked list implementation. The routines themselves are very simple and follow the written description exactly (see Figs. 3.32 to 3.36).

Pop is occasionally written as a function that returns the popped element (and alters the stack). Although current thinking suggests that functions should not change their input variables, Figure 3.37 illustrates that this is the most convenient method in C.

```
int
is_empty( STACK S )
{
        return( S->top_of_stack == EMPTY_TOS );
}
```

Figure 3.32 Routine to test whether a stack is empty--array implementation

```
void
make_null( STACK S )
{
        S->top_of_stack = EMPTY_TOS;
}
```

Figure 3.33 Routine to create an empty stack--array implementation


```
void
push( element_type x, STACK S )
{
        if( is_full( S ) )
                error("Full stack");
        else
                S->stack_array[ ++S->top_of_stack ] = x;
}
```

Figure 3.34 Routine to push onto a stack--array implementation

```
element_type
top( STACK S )
{
        if( is_empty( S ) )
                error("Empty stack");
        else
                return S->stack_array[ S->top_of_stack ];
}
```

Figure 3.35 Routine to return top of stack--array implementation

```
void
pop( STACK S )
{
        if( is_empty( S ) )
                error("Empty stack");
        else
                S->top_of_stack--;
}
```

Figure 3.36 Routine to pop from a stack--array implementation

```
element_type
pop( STACK S )
{
        if( is_empty( S ) )
                error("Empty stack");
        else
                return S->stack_array[ S->top_of_stack-- ];
}
```

Figure 3.37 Routine to give top element and pop a stack--array implementation

Check your Progress

1. Write routines to implement two stacks using only one array. Your stack routines should not declare an overflow unless every slot in the array is used.
2. Propose a data structure that supports the stack push and pop operations and a third operation find_min, which returns the smallest element in the data structure, all in O(1) worst case time.
3. Show how to implement three stacks in one array.

3.4. THE QUEUE ADT

A *queue* is a particular kind of collection in which the entities in the collection are kept in order and the principal (or only) operations on the collection are the addition of entities to the rear terminal position and removal of entities from the front terminal position. This makes the queue a First-In-First-Out (FIFO) data structure. In a FIFO data structure, the first element added to the queue will be the first one to be removed. This is equivalent to the requirement that whenever an element is added, all elements that were added before have to be removed before the new element can be invoked. A queue is an example of a linear data structure.

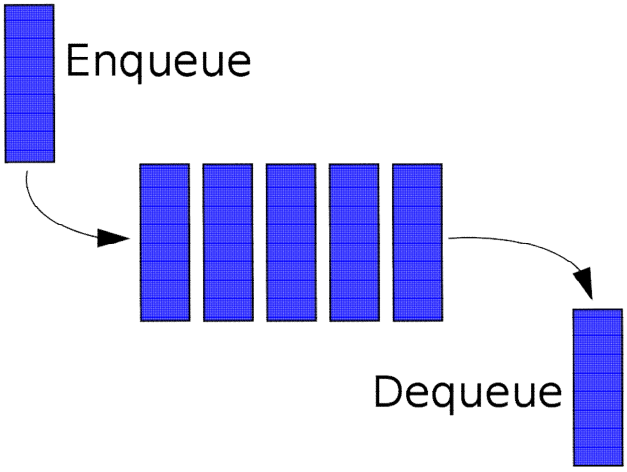

Fig. 3.38 Simple representation of a queue

Like stacks, *queues* are lists. With a queue, however, insertion is done at one end, whereas deletion is performed at the other end.

Queues provide services in computer science, transport and operations research where various entities such as data, objects, persons, or events are stored and held to be processed later. In these contexts, the queue performs the function of a buffer.

Queues are common in computer programs, where they are implemented as data structures coupled with access routines, as an abstract data structure or in object-oriented languages as classes. Common implementations are circular buffers and linked lists.

3.4.1. Queue Model

The basic operations on a queue are *enqueue,* which inserts an element at the end of the list (called the rear), and *dequeue,* which deletes (and returns) the element at the start of the list (known as the front). Figure 3.39 shows the abstract model of a queue.

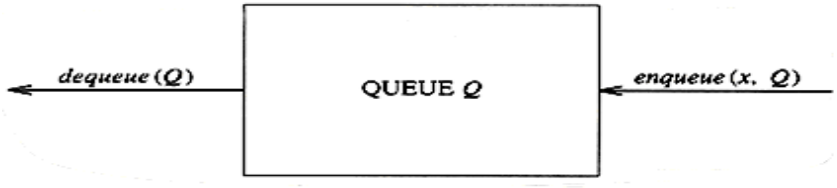

Figure 3.39 Model of a queue

3.3.2. Implementation of Queues

Two implementation of queue are discussed here. Linked list implementation and array implementation.

3.3.2.1 Linked List implementation

```
typedef struct list
{
    int data;
    struct list *next;
}node;
```

Figure 3.40 Definitions for Linked list implementation of queue

```
node *insert(node **f,node **r)
{
    node *ln;
    ln=(node *)malloc(sizeof(node));
    printf("enter the value:");
    scanf("%d",&ln->data);
    ln->next=NULL;
    (*r)->next=(node *)malloc(sizeof(node));
    if(*f==NULL)
    {
        *f=ln;
        *r=ln;
    }
    else
    {
        (*r)->next=ln;
        *r=ln;
    }
}
```

Figure 3.41 Inserting in a queue

```
display(node                                                    *ln)
{
    node                                                        *tmp;
    while(ln!=NULL)
    {
        printf("%d->",ln->data);
        ln=ln->next;
    }
}
```

Figure 3.41 Displaying a queue

```
delete(node                    **f,node                         **r)
{
    node                                                        *tmp;
    tmp=*f;
    if(*f==NULL)
    {
        printf("queue                                           empty");
    }
    else
    {
        (*f)=(*f)->next;
        free(tmp);
    }
}
```


Figure 3.42 Deleting from a queue

3.3.2.2 Array Implementation of Queues

For each queue data structure, we keep an array, *QUEUE*[], and the positions *q_front* and *q_rear,* which represent the ends of the queue. We also keep track of the number of elements that are actually in the queue, *q_size*. All this information is part of one structure, and as usual, except for the queue routines themselves, no routine should ever access these directly. The following figure shows a queue in some intermediate state. By the way, the cells that are blanks have undefined values in them. In particular, the first two cells have elements that used to be in the queue.

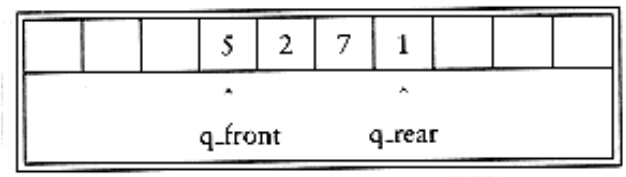

Figure 3.43 Sample queue

The operations should be clear. To *enqueue* an element *x*, we increment *q_size* and *q_rear*, then set *QUEUE*[*q_rear*] = *x*. To *dequeue* an element, we set the return value to *QUEUE*[*q_front*], decrement *q_size*, and then increment *q_front*. Other strategies are possible (this is discussed later). We will comment on checking for errors presently.

There is one potential problem with this implementation. After 10 enqueues, the queue appears to be full, since *q_front* is now 10, and the next *enqueue* would be in a nonexistent position. However, there might only be a few elements in the queue, because several elements may have already been dequeued. Queues, like stacks, frequently stay small even in the presence of a lot of operations.

The simple solution is that whenever *q_front* or *q_rear* gets to the end of the array, it is wrapped around to the beginning. The following figure shows the queue during some operations. This is known as a *circular array* implementation.

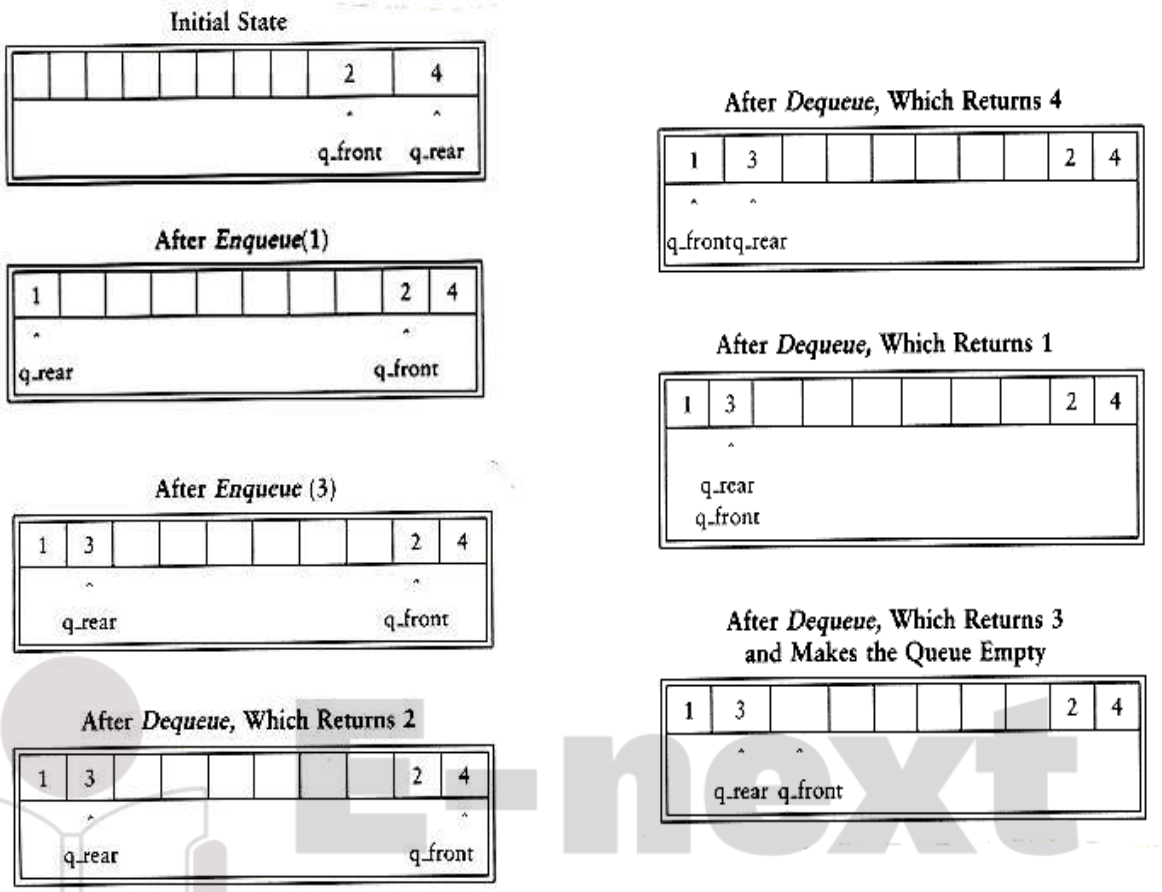

Figure 3.44 Circular array implementation of queue

```
struct queue_record
{
        unsigned int q_max_size;  /* Maximum # of elements */
        /* until Q is full */
        unsigned int q_front;
        unsigned int q_rear;
        unsigned int q_size;      /* Current # of elements in Q */
        element_type *q_array;
};
typedef struct queue_record * QUEUE;
```

Figure 3.45 Type declarations for queue--array implementation

```
int
is_empty( QUEUE Q )
{
        return( Q->q_size == 0 );
}
```

Figure 3.45 Routine to test whether a queue is empty-array implementation

```
void
make_null ( QUEUE Q )
{
        Q->q_size = 0;
        Q->q_front = 1;
        Q->q_rear = 0;
}
```

Figure 3.46 Routine to make an empty queue-array implementation

```
unsigned int
succ( unsigned int value, QUEUE Q )
{
        if( ++value == Q->q_max_size )
                value = 0;
        return value;
}
void
enqueue( element_type x, QUEUE Q )
{
        if( is_full( Q ) )
                error("Full queue");
        else
        {
                Q->q_size++;
                Q->q_rear = succ( Q->q_rear, Q );
                Q->q_array[ Q->q_rear ] = x;
        }
}
```

Figure 3.47 Routine to enqueue-array implementation

Check your Progress

1. Write the routines to implement queues using
   1. linked lists
   2. arrays
2. A *deque* is a data structure consisting of a list of items, on which the following operations are possible:

   push(x,d): Insert item *x* on the front end of deque *d.*

   pop(d): Remove the front item from deque *d* and return it.

   inject(x,d): Insert item *x* on the rear end of deque *d.*

   eject(d): Remove the rear item from deque *d* and return it.
   Write routines to support the deque that take *O*(1) time per operation.

Let us sum up

This chapter describes the concept of ADTs and illustrates the concept with three of the most common abstract data types. The primary objective is to separate the implementation of the abstract data types from their function. The program must know what the operations do, but it is actually better off not knowing how it is done.

Lists, stacks, and queues are perhaps the three fundamental data structures in all of computer science, and their use is documented through a host of examples. In particular, we saw how stacks are used to keep track of procedure and function calls and how recursion is actually implemented. This is important to understand, not just because it makes procedural languages possible, but because knowing how recursion is implemented removes a good deal of the mystery that surrounds its use. Although recursion is very powerful, it is not an entirely free operation; misuse and abuse of recursion can result in programs crashing.

References:

1. Data structure – A Pseudocode Approach with C – Richard F Gilberg Behrouz A. Forouzan, Thomson
2. Schaum's Outlines Data structure Seymour Lipschutz Tata McGraw Hill 2nd Edition
3. Data structures & Program Design in C Robert Kruse, C. L.Tondo, Bruce Leung Pearson
4. "Data structure using C" AM Tanenbaum, Y Langsam & M J Augustein, Prentice Hall India

Question Pattern:

1. What are abstract data types? Explain.
2. Explain the concept of stack in detail.
3. Explain the concept of queue in detail.