UNIT 3 STACKS AND QUEUES

Structure

3.0 INTRODUCTION

In the previous unit we discussed linear LISTS and their implementations. Lists may be modeled in many different types of data structures. We have been concentrating on structuring data in order to insert, delete, or access items arbitrarily. Actually, it turns out that for many applications, it suffices to consider various (rather stringent) restrictions on how the data structure is accessed. Such restrictions are beneficial in two ways: first, they can alleviate the need for the program using the data structure to be concerned with its details (for example, keeping track of links to or indices of items); second, they allow simpler and more flexible implementations, since fewer operations need be supported. Two of such data structures are the focus of this Unit. These are Stacks and Queues. These are two special cases of linear LISTS.

Stacks and Queues are very useful in computer science. Stacks are used in compilers in passing an expression by recursion; in memory management in operating system; etc. Queues find their use in CPU scheduling, in printer spooling, in message queuing in computer network etc. The List of application of Stacks and Queues in real life is enormous. In this Unit we first define both the structures. Subsequently we shall discuss their operations and implementation. At the end we shall take up some of the example applications of Stacks and Queues.

3.1 OBJECTIVES

At the end of this Unit, you shall be able to:

- differentiate between Stack and Queue structures;
- state the usefulness of these two data structures; and
- implement program that reply on Stacks and Queues by using either arrays when the size of Stack or Queue is bounded in advance, using dynamic storage allocation when the size is not known.

3.2 DEFINING STACK AND QUEUE

A Stack is a linear data structure in which data is inserted and deleted at one end (same end) i.e. data is stored and retrieved in a Last In First Out (LEFO) order. The most recently arrived data object is the first one to depart from a Stack. A Stack operates somewhat like a busy executive's "in" box: work piles up on a Stack, and whenever the executive is ready to do some work, he takes it off the top. This might mean that something gets stuck in the bottom of the Stack for some time, but a good executive would presumably manage to get the Stack emptied periodically. It turns out that sometimes a computer program is naturally organised in this way, postponing some tasks while doing others, and thus pushdown Stacks appear as the fundamental data structure for many algorithms. We may draw a Stack in any one of the forms as given in Figure 1.

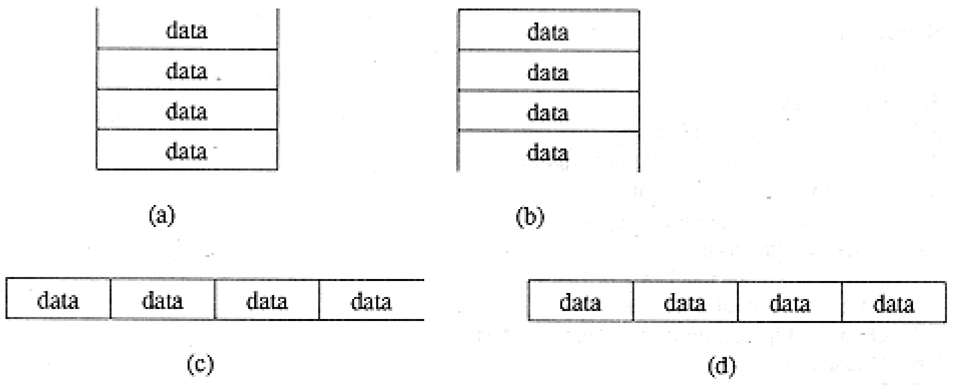


Figure: 1

Each one of the above have one open and one closed end. The data movement (i.e. storage and retrieval) takes place only at the open end, i.e. data is stored and retrieved in last in first out (LIFO) order. We generally use the form given in Figure 1 (a), having the open end in up direction.

The open end of the Stack is called the top of the Stack. The store and retrieval operations for a Stack are called push and pop respectively. Figure 2 shows how a sample Stack evolves through the series of push and pop operations represented by the sequence:

A * S A M * P * L * E S * T * * * A * C K * *

Each letter in this list means "push" (the letter); each asterisk means "POP". We will see a great many applications of Stacks in the Unit.

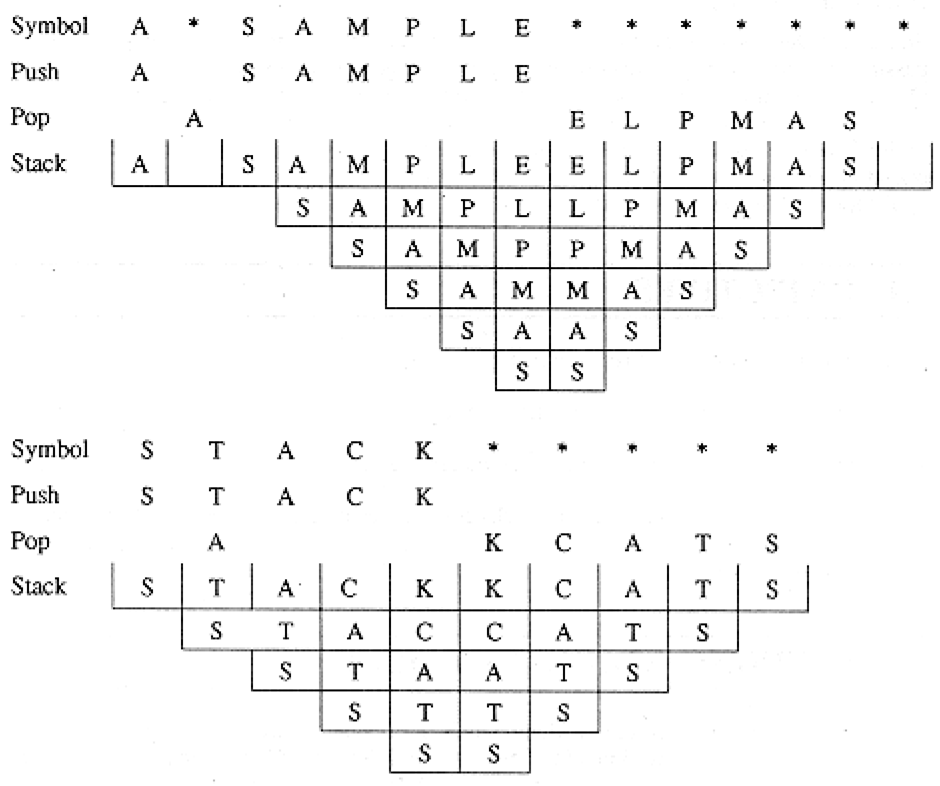


Figure: 2

Another fundamental restricted-access data structure is called the Queue. Again, only two basic operations are involved: one can insert an item into the Queue at the beginning and remove an item from the end. Perhaps our busy executive's "in" box should operate like a Queue, since then work that arrives first would get done first. In a Stack, something can get buried at the bottom, but in a Queue everything is processed in the order received. Queues obey a "First In, First Out (FIFO) discipline. We may draw Queue in any one of the forms as given in Figure 3.

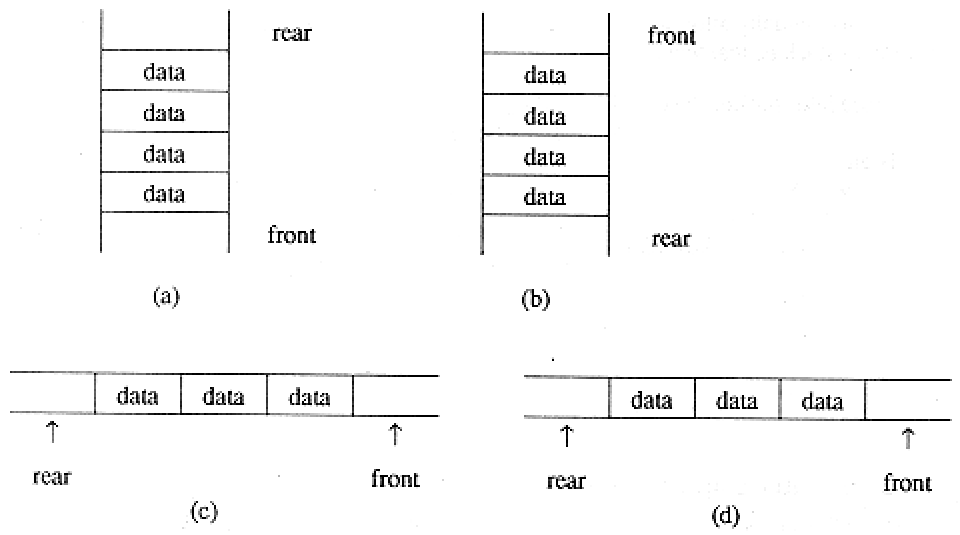


Figure. 3

Queue is marked with two open ends called front and rear.

3.3 STACK OPERATIONS AND IMPLEMENTATION

Basic operation on stack are as given below:

1. Create a stack.
2. Check whether a stack is empty.
3. Check whether a stack is full.
4. Initialize a stack.
5. Push an element onto a stack (if not full).
6. Pop an element from a stack (if not empty).
7. Read a stack top.
8. Print the entire stack.

Stack (a special case of List) can be implemented as one of the following data structures:

- Array
- Linked List

3.3.1 Array Implementation

The simplest way to represent a stack is by using a one- dimensional array, say Stack [N] with room for N elements. The first element in the stack will be at STACK[0], the second element at STACK[1], and so on. An associated variable TOP points to the top element of stack. Type definition for a sequentially allocated stack is

```
# Define  STACK_SIZE_MAX   100/* maximum stack size* /
Typedef  struct
{ int key;
} element;
element stack [STACK_SIZE_MAX];
int top = -1,/* denotes
an empty stack */
```

To check whether the stack is empty, we just need to check the value of TOP. If it is -1, the function may return 1 else 0.

```
int  stackempty( )
{ If  ( top == -1)
        return 1;
else
        return 0; }
```

Given a sequentially allocated stack, and a value to be pushed, this procedure makes the newtop of stack be that value.

```
void add ( int *top, element item)
{If (*top >= STACK_SIZE_MAX-1)
{ stackoverflow ( );
        return;
}
stack [ ++*top] = item;
}
```

3.3.2 Pointer Implementation

Although this method of allocating storage is adequate for many applications, there are many other applications where the sequential allocation method is inefficient and, therefore, unacceptable.

For such applications we store a stack element in a structure with a pointer to the next lower element on the stack.

```
Type struct stack
        { int data;
        stack_pointer next;
        };
typedef struct stack *stack_pointer;
        int *top;
```

Push, Pop and Top operations involve inserting, deleting and reading item at the top of this list structure.

Let us define push operation as given below:

```
Void    push ( stack_pointer *top, int data)]
{ /* Push an element into Stack */
stack_pointer temp =
(stack_pointer) mallor (size of (stack));
temp → data = data1;
temp → data = * top;
        top = temp;}
```

3.4 STACK APPLICATIONS

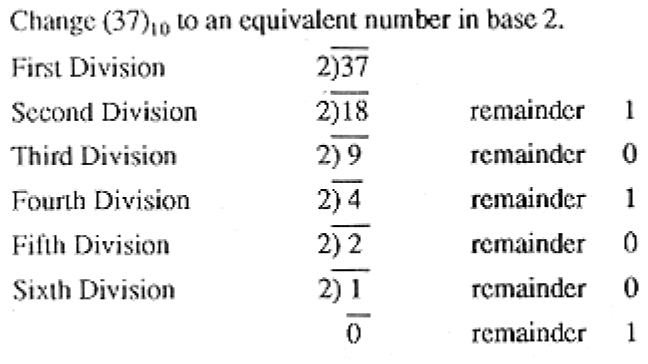
Stacks are simple structures that figure prominently in many algorithms.

Many machines implement basic Stack operations in hardware because they naturally implement function call mechanisms: save the current environment on entry to a procedure by pushing information onto a Stack, resore the environment on exit by using information popped from the Stack. Some calculators and some computing languages base their method of calculation on Stack operations explicitly: every operation pops its arguments from the Stack and returns its results to the Stack. In this section, we shall consider two of the many applications of stacks.

3.4.1 Convert Number Bases by Using Stacks

Changing a number from base 10 to an equivalent number in any other base 0 can be accomplished by repeated division of the number by the new base, until a zero result is obtained, saving the remainder after each division and then writiig out the remainders in reverse order. This process is illustrated by the following example.

Change (37) 10 to an equivalent number in base 2.



Change $(37)_{10}$ to an equivalent number in base 2.

First Division	$2\overline{)37}$		
Second Division	$2\overline{)18}$	remainder	1
Third Division	$2\overline{)9}$	remainder	0
Fourth Division	$2\overline{)4}$	remainder	1
Fifth Division	$2\overline{)2}$	remainder	0
Sixth Division	$2\overline{)1}$	remainder	0
	$\overline{0}$	remainder	1

The remainders are written out in reverse order giving following result:

(37) 10 = (100101)2

The first remainder is the last to be written out while the last remainder is the first to be written out. We can model this LIFO situation very conveniently by means of a Stack.

An algorithm for the same may be implemented as follows:

```
procedure Change No base;
var item : integer;
begin
        sthead:= nil;
        read(number, base);

while number 0 do begin

        item:= number mod base;

push (sthead, item);
number:= number div base;
end;

PRINTSTACK (stackhead);
writeln
end;
Procedure PRINTSTACK (ptr:link);
begin
        while ptr nildo
begin
        write (ptr . data);
        ptr:= ptr . link
        end;
end.

procedure PUSH (var ptr: link; item: integer);
```

```
var temp: link;
begin
        new (temp);
        temp data:= item;
        temp next:= ptr;
        Ptr:= temp

end;
where
        type link = node;
        node =record
        data integer;
        next link
end;
```

We may also write a POP procedure

```
Procedure POP (var ptr: link; var item: integer);
var temp: link;
begin
        if ptr nil then

begin
        item : = ptr . data;
        temp : = ptr;
        ptr : = ptr . next;

dispose (temp)
end;
end.
```

Now that you have seen how easy it is to design and implement the Stacks, let us look at some of its applications.

3.4.2 Infix to Postfix Conversion

You are already familiar with arithmetic expressions in infix notation. In this notation, a binary operator is placed between its operands. For example

A+B-C
A - (C-D)/(B * D)
A + B * D - E/F

The operations are normally carried out from left to right. We also have procedure rules for evaluating expressions. These have already been dealt in detail in this course (see CS-02, Block .., Unit ..) A typical evaluation sequence for

A * B + C + D * E would be to multiply B and A; adding it to C; saving the result temporarily, say in RESULT., multiplying D and E and add it to the RESULT. Therefore we have followed the sequence as given below:

AB*C+DE*+

This notation is called the postfix notation or reverse polish notation.

In postfix notation, an operator is placed immediately after its operands. Following examples show several arithmetic expressions in both infix and postfix notation.

A-B +C* D
AB-CD*+
A+ (C-D)/(B * D)
ACD-BD*/+
A- C+ D/B *A
AC-DB/A*+

We can convert infix notation to postfix notation by using Stack data structure.

We follow the following mechanism. Suppose we want to convert the infix expression A - B + C * D into postfix notation. We read one character at a time.

Input Character	Process	Stack contents	Output
A	is written to output		A
_	is pushed onto stack	-	
B	is appended to output		AB
+	since it has the same procedure as -, the stack is popped out and + is pushed onto the stack	+	AB -
C	appended to output	+	AB-C
*	since it has higher precedence than top of the stack; it is pushed onto the stack	* + *	AB - C
D	is appended to output	+ +	AB- CD
End	stack is popped out		AB - CD * +

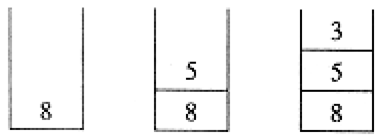
We may write an algorithm as given below:

```
1.    Initialise the stack to be empty.

2     For each character in the infix string
      if operand append to output
      else

            if stack empty or the operator has higher precedence
            than the operator on top of the stack
            then

                  push it onto the stack
            else
                  pop the stack and append it to the output

3.    If input end encountered then loop until stack not empty

      pop the stack and append to the output.
```

We can also use stacks to evaluate expressions in postfix notation. To do this, when an operand is encountered, it is pushed on to the stack, when an operator is encountered, it is applied to the first two operands that are obtained by popping the stack and die result is pushed onto the stack. For example, the postfix expression

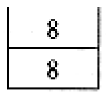
8 5 3 + 9 * + 4 +

is evaluated as follows.

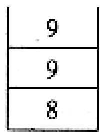


On reading 8, 5 and 3, the stack contents are

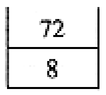
On reading +; 3 and 5 are popped out and are added; the result 8 (3 + 5) is pushed onto stack



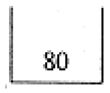
Next 9 is pushe



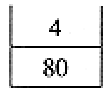
On reading * 9 and 8 are popped and
9 * 8 = 72 is pushed onto the stack



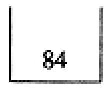
On finding +, 72 and 8 are popped out and
72 + 8 = 80 is pushed onto the stack.



Now 4 is pushed onto the stack.



Finally, a + is read and 4 and 80 are popped, the result
4 + 80 = 84 is pushed onto the stack



End of the string encountered therefore start is popped out and that is the result, i.e. 84.

Exercise

Write a program that converts a legal fully parenthesized infix expression into a postfix expression.

3.5 QUEUES : OPERATIONS AND IMPLEMENTATION

You may encounter another familiar example of a queue in a timesharing computer system where many users share the computer system simultaneously. Similar to stack operations, operations that define a queue are given below:

1. Create a queue
2. Check whether a queue is empty
3. Check whether a queue is full
4. Add item at the rear queue (enqueue)
5. Remove item from front of queue (dequeue)
6. Read the front of queue
7. Print the entire queue

As was done in case of stacks, we can give a vector representation of a queue. We define a queue as a record structure containing the vector and two variables front and rear to denote the present position of its front and rear element.

We may define a Queue as given below:

```
Const max = 100
Struct  q_type
{ elementtype     queue [max];
Int front, rear;
};
```

As an example of this representation of a queue, consider a size 6. Assume that the queue is initially empty. It is required to insert element RED, BLACK and BLUE, followed by delete RED and BLACK and insert GREEN, WHITE AND YELLOW.

Following figure give a trace of the queue contents for this sequence of operations.

RED					
RED	BLACK				
RED	BLACK	BLUE			
	BLACK	BLUE			
		BLUE			
		BLUE	GREEN		
		BLUE	GREEN	WHITE	
		BLUE	GREEN	WHITE	YELLOW

Now if we try to insert ORANGE, an overflow occurs even though the first two cells are free To avoide this drawbacks, we can arrange these elements in a circular fashion with QUEUE [0] following QUEUE [N-1]. It is then called a circular array representation. We may depict a circular queue as given in figure.

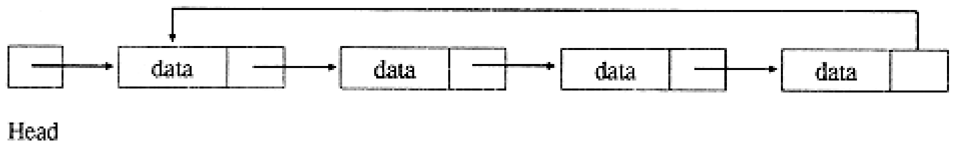


Figure: Circular queue

The insertion and deletion procedure for a circular queue are as given below:

```
Void  Struc qtype qinsert (qtype Q, element type X)
{  int Newrear
        If (q.rear ==max-1)
                Newrear = 0;
        Else
                Newrear = q.r rear +1;
                If ( Newrear == q.r front)
                QUEUE OVERFLOW;
        Else
                {q.rear = Newrear;
                q.queue (rear) = x;
                If q.front == 0
                q.front = q. Rear;
                }
        Return q;
        }
                Stuct qtype qdelete (qtype Q, element type X)
        { If q.rear == 0
                QUEUE UNDER FLOW
        Else
                {  x = q.queue[front];
                If (q.front == q.Rear)
                {
                        front = Rear = 0;
                }
        else
                If q.front = q.Max
                q.front = 1
        else
                ++(q.front);

        }
}
```

Queues are important in simulation models. They serve several roles: as repositives for scheduled events, as holding areas for entities moving through the system.

Similarly we may write procedure for other operations on stacks.

The second approach for implementing queues is by using the dynamic storage allocation through the use of pointers. We can defme a queue consisting of records where each record contains record that comes after it. Therefore we may declare a queue as

```
        Struct  qrec
        { element type data;
        Struct qrec * next;};
Struct qrec *qptr;
```

we must also specify the items at the front and rear of the queue. This may be done by the following declaration.

```
        Struct qtype
        { Struct qrec * front;
                Struct qrec * rear;
        };
        qtype q;
```

This kind of an implementation is a singly linked list implementation of the Queue. Recall the Queue implementation using circular arrays. Similarly we can implement Queues using circular lists. In this list, each node points to the next, and the chain of pointers eventually form a loop back to the first one.

In such a case the declaration becomes simplified as given below:

```
        Struct  qrec
        { element type data;
                Struct qtype next;
        };
        Struct qrec * qtype;
        Struct qrec * q;
```

You must notice that circular list implementation requires special. attention for insertion and deletion in queues with no elements or with just one element

3.6 QUEUE APPLICATION

In a computer network, messages from one to another computer are generally created asynchronously. These messages therefore need to be buffered until the receiving computer is ready for it. These communication buffers make extensive use of Queues by storing these messages in a Queue. Also the messages need to be sent to receiving computer in the same order in which they are created, i.e. FIFO order. Example below implements a communication buffer Queue with messages of type char.

```
define          EMPT        0
define          MAX         10
define          ALWAYS      1
define          RING        "007"
                    enum status {SUCCESS, FULL};
typedef         Char      Data;
typedef         Data      [MAXI;
                    struct q-ptr-def
                    {
                            int q-insert;
                            int q-remove;
                        };
                        typedef struct q-ptr-def Q-PTR;
enum status insert-q                        (Q q,
                                            Q-PTR *q-ptr,
                                            DATA Message)

{
        int new-pos;
        if ((new-pos = q-ptr - q-insert + 1) = MAX)
        new-pos = 0;
```

```
          if (new-pos == q-ptr q-remove)
            return (FULL);
          q-ptr q-insert = new-pos;
          q [q-ptr - q-insert] = message;
            return (SUCCESS);
}

DATA remove-q (Q q,

                    Q-PTR *q-ptr)
}
          DATA message;
if (q-ptr - q-remove = q-ptr - q-insert)
return (EMPT);
if ((q-ptr - q-remove += 1) = MAX)
q-ptr - q-remove = 0;
message = q[q-ptr - q-remove];
return (message);
}
main ( )
{         Q q;
          Q-PTR q-ptr;
          q-ptr.q-insert = 0;
          q-ptr.q-remove = 0;

While (ALWAYS)
}
          printf ("In ENTER\"1\" to Add message in Queue");
          printf ("In ENTER\"2\" to Remove message from Queue")
          printf ("In ENTER\"3\" to Exit In")
          printf ("ENTER MESSAGE TO QUEUE In")
          if (inser-q (q, + q-ptr, get che( )) == FULL
          printf ("In Queue FULL");
          break;
case '2':
          printf  ("Removed message is : %C";
          remove-q (q, q-ptr));
break;

case '3':
          exit ( );
          default:
                    printf (RING);
                    break,
          }
   }
}
```

3.7 PRIORITY QUEUES

Many application involving queues require priority queues rather than the simple FIFO strategy. Each queue element has an associated priority value and the elements are served on a priority basis instead of using the order of arrival. For elements of same priority, the FIFO order is used. For example, in a multi-user system, there will be several programs competing for use of the central processor at one time. The programs have a priority value associated to them and are held in a priority queue. The program with the highest priority is given first use of the central processor.

Scheduling of jobs within a time-sharing system is another application of queues. in such a system many users may request processing at a time and computer time divided among these requests. The simplest approach sets up one queue that store all requests for processing. Computer processes the request at the front of the queue and finished it befor starting on the next. Same approach is also used when several users want to use the same output device, say a printer.

In a time-sharing system, another common approach used is to process a job only for a specified maximum length of time. If the program is fully processed within that time, then the computer goes on to the next process. if the Program is not completely processed within the specified time, the intermediate values are stored and the remaining part of the program is put back on the queue. This approach is useful in handling a mix of long and short jobs.

3.8 SUMMARY

A stack is a list in which retrievals, insertion and deletions can take place at the same position. It follows the last in first out (LIFO) mechanism. Compiler implement recursion by generating code for creating and maintaining an activation stack, i.e. a run time stack that holds the state of each active subprogram. AQ is a list in which retrievals and deletions can take place at one end and insertions occur at another end. It follows first in first out(FIFO) order.

Queues are employed in many situations. The items on a Queues may be vehicle waiting at a crossing, car waiting at the service station, customers in a check-out line at a departmental store, etc.