Unit 9 Stack and Queues

Structure:

9.1 Introduction

In computer science, a **stack** is an abstract data type and data structure based on the principle of Last in First out (LIFO). Stacks are used extensively at every level of a modern computer system. For example, a modern PC uses stacks at the architecture level, which are used in the basic design of an operating system for interrupt handling and operating system function calls. The stack is ubiquitous.

Using this analogy a stack is defined as a special type of data structure where items are inserted from one end called top of stack and items are deleted from the same end. Here, the last item inserted will be on top of stack. Since deletion is done from the same end, Last item Inserted is the First item to be deleted out from the stack and so, stack is also called **Last In First Out (LIFO)** data structure.

A queue is a pile in which items are added in one end and removed from the other. In this respect, a queue is like the line of customers waiting to be served by a bank teller. As customers arrive, they join the end of the queue while the teller serves the customer at the head of the queue. As a result, a queue is used when a sequence of activities must be done on a first-come, first-served basis. Queue is a linear list for which all insertions are made at one end of the list; all deletions (and usually all accesses) are made at the other end (**FIFO**).

Objectives:

At the end of this unit, you will be able to explain:

- Stack definition and its operations
- POP and PUSH operation in C
- Various stack applications
- Stack implementation using Arrays and Structure
- Queue, its operations and types.

9.2 Operations on Stack

A Stack is defined formally as a list (a linear data structure) in which all insertion and deletion are made at one end and that end is called the top of the stack. Here, the last item inserted will be on top of stack. Since deletion is done from the same end, Last item Inserted is the first item to be deleted out from the stack and so, stack is also called Last In First Out (LIFO) data structure.

There are three basic operations:

- INSERT / PUSH (Insert an item into the stack)
- DELETE / POP (Delete an item from the stack)
- VIEW / LIST (List the stack elements)

From the definition of stack, it is clear that it is a collection of similar type of items and naturally we can use an array (An array is a collection of similar data types) to hold the items of stack. Since array is used, its size is fixed and that size is called STACK_SIZE.

At the time of push operation, the stack full condition must be checked. If the stack is full, called **"Stack Overflow",** no element can be inserted into the stack. Otherwise, push the element into the stack.

At the time of pop operation, the stack empty condition must be checked. If the stack is empty called, **"Stack Underflow",** no element can be deleted from the stack. Otherwise, pop the element from the stack.

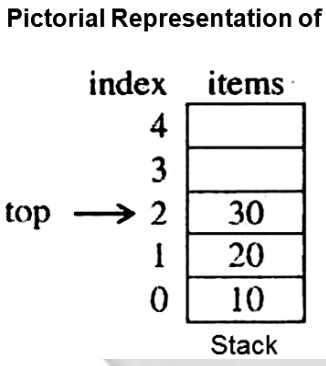


Fig. 9.1: Stack Representation

9.2.1 Push Operation

So, let us assume that 5 items 30, 20, 25, 10 and 40 are to be placed on the stack. The items can be inserted one by one as shown in following figure.

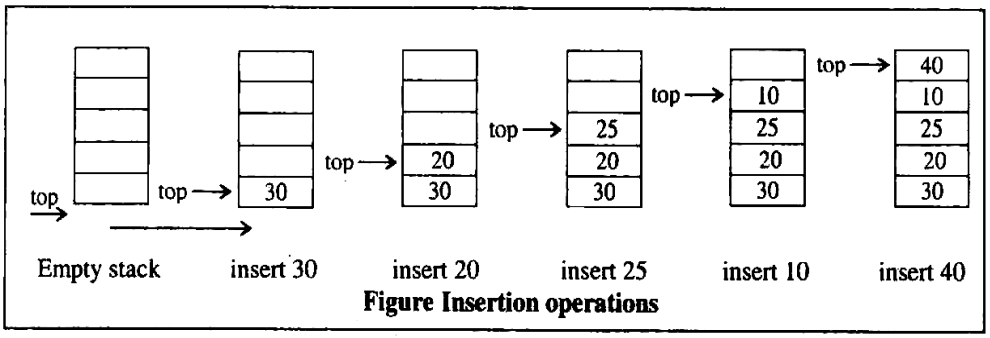


Fig. 9.2: Insertion Operations

After inserting 40 the stack is full. In this situation it is not possible to insert any new item. This situation is called **stack overflow**.

General Algorithm for Stack Operation

TOP is a pointer which moves when PUSH and POP operations are occurred. Initially, the pointer **TOP =-1**, implies no value is present in the stack, that is stack empty. When PUSH operation occurred then **TOP = TOP + 1** and when POP operation occurred then **TOP = TOP - 1**

when **TOP = -1**, implies underflow

TOP = STACK_SIZE -1, implies Overflow

Item is the value to be inserted or deleted

Algorithm for PUSH Operation

PUSH(stack, TOP, item)

Step I : If TOP=STACK_SIZE - 1, then

print “Stack Overflow” and Return

[end of if structure]

Step II: set TOP←TOP+1

Step III: set stack [TOP]← item

Step IV: Return

To design a C function, to start with let us assume that three items are already added to the stack and stack is identified by s as shown in figure a.

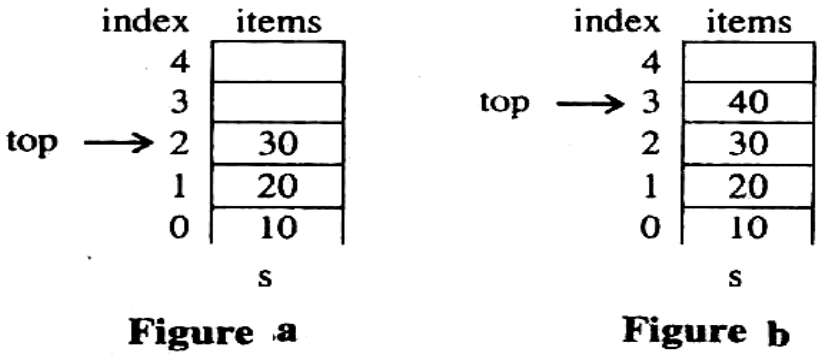


Fig. 9.3

Here, the index top points to 30 which is the topmost item. Here, the value of top is 2. Now, if an item 40 is to be inserted, first increment top by 1 and then insert an item. The corresponding C statements are:

```
top = top + 1;
s[top] = item;
```

These two statements can also be written as

```
s[+ + top] = item
```

But, as we insert an item we must take care of the overflow situation i.e., when top reaches STACK_SIZE-l, stack results in overflow condition and appropriate error message has to be returned as shown below:

```
if (top == STACK_SIZE -1)
{
      printf("Stack overflow\n");
      return;
}
```

Here, STACK_SIZE should be #defined and is called symbolic constant the value of which cannot be modified. If the above condition fails, the item has to be inserted. Now, the C code to insert an item into the stack can be written as

```
if (top == STACK_SIZE -1)
{
   printf("Stack overflow\n");
   return;
 }
 s[ + + top] = item;
```

It is clear from this code that as the item is inserted, the contents of the stack identified by s and top are affected and so they should be passed and used as pointers.

Function to insert an integer item

```
void push(int item,  int *top, int s[])
{
        if (*top == STACK_SIZE -1)
        {
        printf("Stack overflow\n");
        return;
        }
        s[+ +(*top)] = item;  /* Increment top and then insert an item */
}
```

Note: In the above example an item of integer data type is inserted into the stack. The following example illustrates insertion of a character data into a stack.

Function to insert a character item

```
void push(char item, int *top, char s[])
{
        if (*top == ST ACK_SIZE -1)
        {
        printf("Stack overflow\n");
        return;
        }
        s[+ +(*top)] = item; /* Insert an item on the stack */
}
```

9.2.2 POP Operation

When an item is to be deleted, it should be deleted from the top as shown in the following figure.

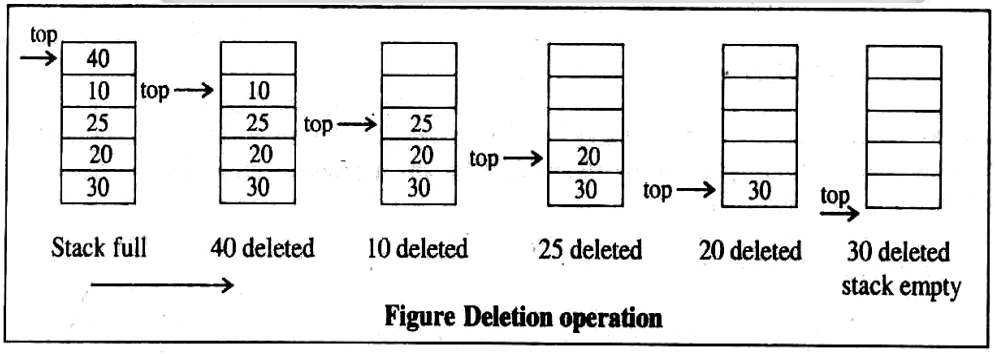


Fig. 9.4: Deletion Operation

The items deleted in order are 40, 10, 25, 20 and 30. Finally, when all items are deleted then TOP points to bottom of stack. When the stack is empty, it is not possible to delete any item and this situation is called **stack underflow**.

Algorithm for POP Operation

POP(stack, TOP, item)

Step I : If TOP=-1, then

print “Stack Underflow”

Return

[end if Structure]

Step II: set item ←stack[TOP]

Step III: TOP ← TOP -1

Step IV: Return

Deleting an element from the stack is called ‘**POP’** operation. This can be achieved by first accessing the top element s[top] and then decrementing top by one as shown below:

item = s[top--];

Each time, the item is deleted, top is decremented and finally, when the stack is empty the top will be -1 and so, it is not possible to delete any item from the stack. The above statement has to be executed only if stack is not empty. Hence, the code to delete an item from stack can be written as

```
if (top == -1)
 {
    return -1; /* Indicates empty stack */
 }
/* Access the item and delete */
 item = s[top--]; .
 return item;
```

As the value of top changes every time the item is deleted, top can be used as a pointer variable. The complete function is shown below. The next example shows how to delete a character item from the stack.

Function to delete an integer item

```
int pop(int *top, int s[ ] )
{
        int     item;
        if (*top == -1)
        {
        return 0;       /* Indicates empty stack */
        }
        item = s[(*top)--];/* Access the item and delete */
        return item; /* Send the item deleted to the calling function */
}
```

Function to delete a character item

```
char pop(int *top, chars[])
{
        char item;
        if(*top= =-1)
        {
        return 0; /* Indicates empty stack */
        }
        item = s[(*top)--];/* Access the item and delete */
        return item; /* Send the item deleted to the calling function */
}
```

9.2.3 Display Items of a Stack

Assume that the stack contains three elements as shown below:

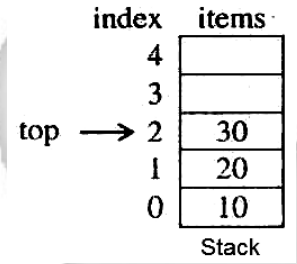


Fig. 9.5

The item 30 is at the top of the stack and item 10 is at the bottom of the stack. Usually, the contents of the stack are displayed from the bottom of the stack till the top of the stack is reached. So, first item to be displayed is 10, next item to be displayed is 20 and final item to be displayed is 30.

So, the code corresponding to this can take the following form

```
        for (i = 0; i <= top; i+ +)
         {
        printf("%d\n", s[i]);
}
```

But, the above statement should not be executed when stack is empty i.e., when top takes the value -1. So, the modified code can be written as shown in the following example.

Function to display the contents of the stack

```
void display(int top, int s[])
{
        int i;
        if(top= = -1)
        {
        printf("Stack is empty\n");
```

```
        return;
        }
        printf("Contents of the stack\n");
        for (i = 0; i <= top; i++)
        {
        printf("%d\n", s[i]);
        }
}
```

9.3 Stack Implementation

The stack can be implemented by using arrays or structures.

9.3.1 Stack Implementation Using Arrays

In the previous sections, we have seen how the stacks can be implemented using arrays. The complete program to perform various stack operations is below.

Example:

Q1) C Program to implement the stack using arrays

[File Name: u9q1.c]

```
#include <stdio.h>
#include <process.h>
#define STACK_SIZE 5
void main( )
{
/******** Function Prototype ********/
void push(int,int *, int *);
int  pop(int *, int *);
void display(int, int *);
/***********************************/
```

```
int  top;              /* Points to top of the stack */
int  s[10];     /* Holds the stack items */
int  tem;      /* Item to be inserted or deleted item */
int  choice;    /* user choice for push, pop and display */
top = -1;          /* Stack is empty to start with */
for (;;)           /*Infinite loop*/
{
clrscr();
printf("\t\t STACK OPERATION\n");
printf("\t\t ~~~~~~~~~~~~~~~\n");
printf("\t\t 1: Push\n");
printf("\t\t 2: Pop\n");
printf("\t\t 3: Display\n");
printf("\t\t 4: Exit\n\n\n");
printf("\t\t Enter the choice : "); scanf("%d", &choice);
switch(choice)
{
case 1:                    // push operation
        printf("\n\nEnter the item to be inserted : "); scanf("%d", &item);
        push(item, &top, s);
        continue;
case 2:                    // pop operation
        item = pop(&top, s);
        if(item == 0)
                printf("Stack is empty\n");
        else
                printf("POP Successful, Deleted  Item = %d\n", item);
        break;
```

```
case 3:                 // display
        display(top, s);
        break;
case 4:                 // exit from program
    exit(0);
default:                // invalid input
        printf("Invalid Input - Try Again");
}
getch();
}  /* End of For loop */
}  /*************************end of main() ******************************/
/************************** push function **************************/
void push(int item, int *top, int s[])
{
if (*top == STACK_SIZE - 1)
{
  printf("Stack overflow\n");
  getch();   return;
}
s[++(*top)] = item;  /* Increment top and then insert an item */
}
/****************************** end of push() *************************/
int pop(int *top, int s[ ] )
{
        int item;
        if (*top == -1)
        {
                return 0;         /* Indicates empty stack */
        }
```

```
   item = s[(*top)--];/* Access the item and delete */
   return item; /* Send the item deleted to the calling function */
}
/****************************** end of pop() *************************/
void display(int top, int s[])
{
        int i;
        if(top == -1)
        {
                printf("Stack is empty\n");
                return;
        }
        printf("\n\n\n\t\t  Contents of the STACK\n\n");
        for (i = top; i >= 0; i--)
        {
                printf("\t\t\t|%5d |\n", s[i]);
        }
        printf("\t\t\t--------");
 } /********************** end of display() ****************************/
```

9.3.2 Stack Implementation Using Structures

So far we have seen how a stack is implemented using an array and various applications of stacks. In this approach whenever a function push() is called, we have to pass three parameters namely item, top and S, where item is the element to be pushed, top is an integer value which is the index of the top most element in the array S. But, as the number of parameters increases, the overhead of programming also increases and efficiency decreases.

In such cases, we group all related items under a common name using a structure and pass structures as parameters, which eventually reduces the burden and increases the efficiency. In our stack implementation, instead of

passing two parameters top and S, we can pass only one parameter if we use a structure. So, a stack can be declared as a structure containing two objects viz., an array to store the elements of the stack and an integer indicating the position of the topmost element in the array.

Example:

Q2) program to simulate the stack operations using structures [File Name: u9q2.c]

```
#include<stdio.h>
#include <process.h>
#define STACK_SIZE 5
struct stack   // structure declaration
{
        int items[STACK_SIZE];
        int top;
};
typedef struct stack STACK;
void push(int, STACK *);
int  is_empty(STACK *);
int  is_full(STACK *);
int  pop(STACK *);
void display(STACK);
void main()
{
   int item; /* Item to be inserted */
   int choice; /* Push, pop, display or quit */
   STACK s; /* To store items */
   s.top = -1; /* Stack is empty initially */
   for (;;)        /*Infinite loop*/
   {
```

```
clrscr();
printf("\t\t  STACK Operation\n");
printf("\t\t  ~~~~~~~~~~~~~~~\n\n");
printf("\t\t  1: Push\n");
printf("\t\t  2: Pop\n");
printf("\t\t  3: Disply\n");
printf("\t\t  4: Exit\n\n");
printf("\t\t Enter the choice : ");
scanf("%d", &choice);
switch(choice)
{
case 1:     // push into stack
   printf("\n  Enter the item to be inserted : "); scanf("%d", &item);
   push(item, & s);
   continue;
case 2:    // pop  from stack
   item = pop(&s);
   if (item != 0)
   {
         printf("\n\n\t\t Pop Successful , Item deleted = %d", item);
   }
   break;
case 3:      // display
   display(s);
   break;
case 4:      // exit from program
   exit(0);
default:      // display
   printf("\n\n\t\t Invalid Input - Try Again");
```

```
            break;
          }  // end of switch
    getch();
    }
}
/***************************************************/
int is_empty(STACK *s)
{
if (s->top == -1)
  return -1; /* Stack empty */
else
  return 0;                  /* Stack is not empty */
}
/***************************************************/
int is_full(STACK *s)
{
if ( s->top == STACK_SIZE  - 1) return 1; /* Stack is full */
return 0; /* Stack is not full */
 }
/***************************************************/
void push(int item, STACK *s)
{
   if ( is_full(s) )
   {
         printf("Stack Overflow\n");
         return;
   }
         s->top++; /* Update top to point to next item */
         s->items[s->top] = item; /* Insert the item into the stack*/
```

```
}
/*********************************************/
int pop(STACK *s)
{
int item;
  if ( is_empty(s) )
  {
    printf("Stack Underflow\n");
    return 0;
  }
 item = s->items[s->top]; /* Access the top element */
 s->top--; /* Update the pointer to point to previous item */
 return item; /* Return the top item to the calling function */
}
/***************************/
void display(STACK s)
{
          int i;
          if (is_empty(&s))
            {
                  printf("\n Stack is empty\n");
                   return ;
             }
                  printf("\n\n\t\t  The contents of the stack\n\n\n");
                  for (i=s.top; i>=0; i--)
                  {
                    printf("\t\t  |%5d |\n ", s.items[i]);
                  }
                    printf("\t\t  --------");
```

```
        return;
}
/*******************************/
```

9.4 Applications of Stack

A stack is very useful in situations when data have to be stored and then retrieved in the reverse order. Some applications of stack are listed below:

Function Calls:

Stacks play in nested function calls. When the main program calls a function named F(), a stack frame for F() gets pushed on top of the stack frame for main. If F() calls another function G(), a new stack frame for G() is pushed on top of the frame for F(). When G() finishes its processing and returns, its frame gets popped off the stack, restoring F() to the top of the stack.

In case of recursion a system stack is maintained to keep the track of function calls.

Large number Arithmetic:

As another example, consider adding very large numbers. Suppose we want to add 353,120,457,764,910,452,008,700 and 234,765,000, 129,654,080,277. First of all, note that it would be difficult to represent the numbers as integer variables or other, as they cannot hold such large values. The problem can be solved by treating the numbers as strings of numerals, storing them on two stacks, and then performing addition by popping numbers from the stacks.

9.5 Polish Notation

Stacks are useful in evaluation of arithmetic expressions. The process of writing the operators on an expression either before operands or after operands or in between them is called **Polish Notation**, in honour of its

discoverer, the Polish Mathematician Jan Luksiewiez. The fundamental property is that the order in which the operations are to be performed is completely determined by the positions of the operators and operands is the expression. Accordingly, one never needs parenthesis when writing expression in Polish notation.

There are 3 types of notations for expressions. The standard form is known as the **infix** form. The other two are **postfix** and **prefix** forms.

Infix: operator is between operands [A + B]
Postfix: operator follows operands [A B +]
Prefix: operator precedes operands [+ A B]

The name came according to the position of the operator.

Example 1 (Infix to Postfix):

a + b * c [**Infix form**]
(precedence of * is higher than of +)
a + (b * c) convert the multiplication
a + (b c *) convert the addition
a (b c *) + Remove parentheses
a b c * + [**Postfix form**]

Note: There is no need of parentheses in postfix forms.

Example 2 (Infix to Prefix):

(A + B) * C **Infix form**
(+ A B) * C Convert the addition
* (+ A B) C Convert multiplication
* + A B C [**Prefix form**]

Note: There is no need of parenthesis anywhere

9.5.1 Infix to Postfix Conversion

Algorithm

Input : Infix(Q)

Output : Postfix(P)

Polish(Q,P)

Step I :	Push “(“ on to the stack and add “)” at the end of Q.
Step II:	Scan Q from left to right and repeat Step III to Step IV for each element of Q until the stack is empty.
Step III:	If an operand is encountered add it to P.
Step IV:	It a left parenthesis “(“ is encountered push it on to Stack.
Step V:	If an operator is encountered then: A Repeatedly pop from Stack and add to P of each operator which has the same precedence as or higher precedence than <operator> b) Add <operator> to Stack [End of If Statement]
Step Vi:	If the right parenthesis is encountered then:- Repeatedly pop from Stack and add to P of each operator until a left parenthesis is encountered. b) Remove left parenthesis from Stack. [End of If Statement] [End of Step II Loop]
Step VII:	Exit

Example:

Step By Step conversion from Infix to Postfix Notation

Q ≡ A + (B * C - (D / E ^ F) * G) * H [Infix Notation]

P ≡ A B C * D E F ^ / G * - H * + [Postfix Notation]

Input:

Q ≡ A + (B * C - (D / E ^ F) * G) * H [Infix Notation]

Scan Symbols	Stack	Expression P
A	(	A
+	(+	A
(	(+(	A
B	(+(	AB
*	(+(*	AB
C	(+(*	ABC
-	(+(-	ABC*
(	(+(-(	ABC*
D	(+(-(	ABC*D
/	(+(-(/	ABC*D
E	(+(-(/	ABC*DE
^	(+(-(/^	ABC*DE
F	(+(-(/^	ABC*DEF
)	(+(-	ABC*DEF^/
*	(+(-*	ABC*DEF^/
G	(+(-*	ABC*DEF^/G
)	(+	ABC*DEF^/G*-
*	(+*	ABC*DEF^/G*-
H	(+*	ABC*DEF^/G*-H
)	Empty	ABC*DEF^/G*-H*+

Output: P ≡ A B C * D E F ^ / G * - H * +

9.5.2 Postfix Evaluation

Algorithm:

This algorithm evaluates a postfix expression (P)

Step I :	Add a right parenthesis ")" at the end of P
Step II:	Scan P from left to right and repeat Step III to Step IV for each element of P until the right parenthesis is encountered.
Step III:	If an operand is encountered then put it into the stack.
Step IV:	If an operator is encountered then:- a) Remove two top elements from Stack where A is the top and B is the next top element b) Evaluate B <operator> A and push into Stack [End of If Statement] [End of Step II Loop]
Step V:	Set value = top element of Stack
Step VI:	Exit

Example:

Postfix Evaluation:

Infix Expression : Q = 5 * (6 + 2) - (12 / 4)

Postfix Expression : P = 5 6 2 + * 12 4 1 - **)**

Scan Character	Stack
5	5
6	6
2	5 6 2
+	8
*	40
12	12
4	40 12 4
/	3
-	37
)	Stop

Result: 37

9.6 Operations on Queues

A queue is defined as a special type of data structure where elements are inserted from one end and elements are deleted from the other end. The end from where the elements are inserted is called '**rear end '(r)** and the end from where elements are deleted is called '**front end'(f)**. In a queue always elements are inserted from the rear end and elements are deleted from the front end. The operation is called First In First Out (**FIFO**). The pictorial representation of the queue is shown below.

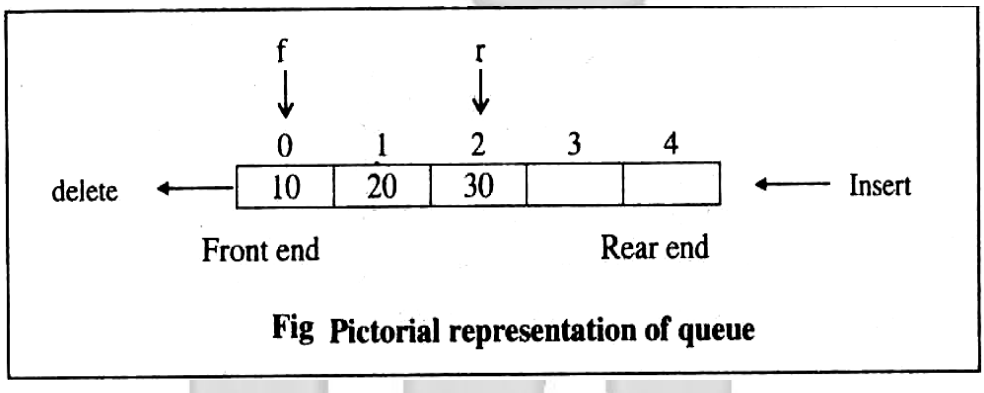


Fig. 9.6: Pictorial Representation of Queue

This data structure is useful in time-sharing systems where many user jobs will be waiting in the system queue for processing. These jobs may request the services of CPU, main memory or external devices such as printer etc. All these jobs will be given a fixed time for processing and are allowed to use one after the other.

Like Stack, there are three basic operations

a) **PUSH** (Insert an item into a Queue)
b) **POP** (Delete an item from a Queue)
c) **VIEW** (Display items from a Queue)

PUSH Operation:

At the time of PUSH operation it checks the REAR pointer, if it is greater than QUEUESIZE then shows "**Queue Overflow**" message and exits.

Initially Front=0, Rear= -1

Push(Queue, Front, Rear, QUEUESIZE, Item)

Step I :	if Rear = QUEUESIZE -1 then Print "**Queue Overflow**" and Return [End of if Structure]
Step II:	Rear ← Rear +1
Step III:	Queue[Rear] ← item
Step IV :	Exit

Algorithm (POP Operation):

Data deletion from the queue is called POP. At the time of POP operation it checks the FRONT pointer, if it is greater than REAR then shows "**Queue Underflow**" message and exits.

Initially Front=0, Rear= -1

Pop(Queue, Front, Rear, QUEUESIZE, Item)

Step I :	if Front > Rear then Print "**Queue Underflow**" and Return [End of if Structure]
Step II:	Front ← Front +1
Step III:	Exit

9.7 Different Types of Queues

The different types of queues are:

- Ordinary queue
- Double-ended queue

- Circular queue
- Priority queue

9.7.1 Ordinary queue

The ordinary queue and its algorithm has already been discussed. We can insert data from one end and delete from the other end. The main disadvantage of that type of queue is that though some situation appeared when space is available but we can't insert data (Queue Overflow).

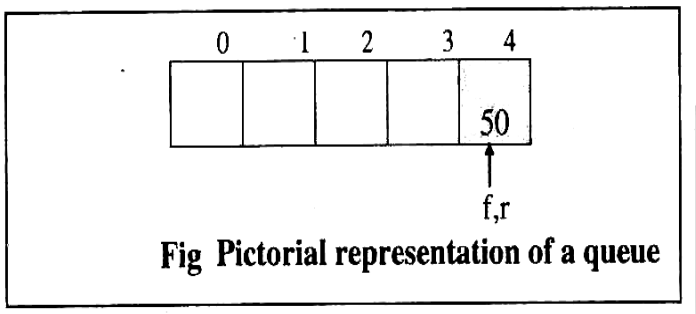


Fig. 9.7: Pictorial Representation of a Queue

A queue can be represented using an array as shown in figure 9.7. The operations that can be performed on these queues are:

a) PUSH ,Insert an item at the rear end
b) POP, Delete an item at the front end
c) VIEW, Display the contents of the queue

Let us discuss how these operations can be designed and implemented.

a) PUSH, Insert at the rear end

Consider a queue, with QUEUE_SIZE as 5 and assume 4 items are present as shown in figure 9.8a below:

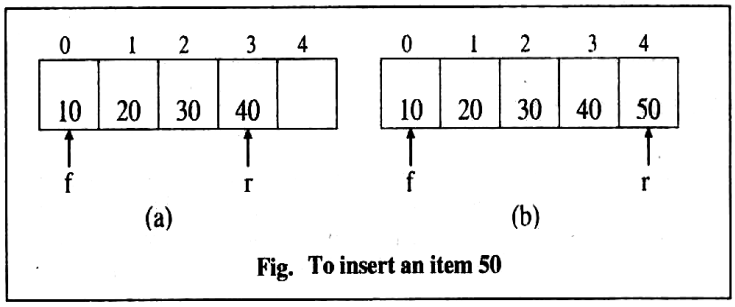


Fig. 9.8: To Insert an Item 50

Here, the two variables f and r are used to access the elements located at the front end and rear end respectively. It is clear from this figure that at most 5 elements can be inserted into the queue. Any new item should be to be inserted to the right of item 40 i.e., at q[4]. It is possible if we increment r by 1 so as to point to next location and then insert the item 50. The queue after inserting an item 50 is shown in figure 9.8b.

b) POP, Delete from the front end

The first item-'to be deleted from the queue is the item, which is at the front end of the queue. It is clear from the queue shown in figure 9.9 (a) that the first item to be deleted is 10. Once this item is deleted, next item i.e., 20 in the queue will be the first element and the resulting queue is of the form shown in figure 9.9(b).

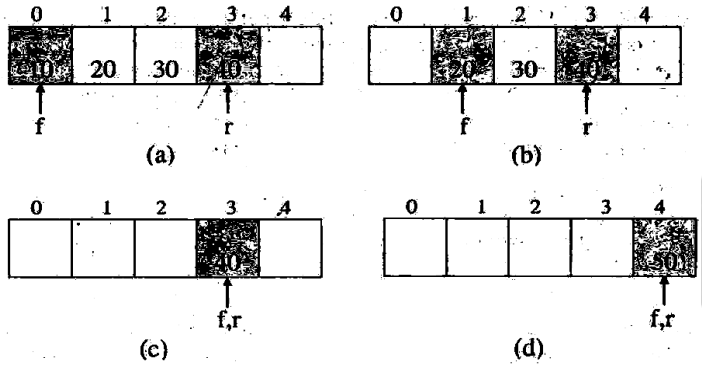


Fig. 9.9: To Delete an Item from quene

So, the variable f should point to 20 indicating that 20 is at the front of the queue. This can be achieved by accessing the item 10 first and then incrementing the variable f.

c) VIEW, Display queue contents

The contents of queue can be displayed only if queue is not empty. If queue is empty an appropriate message is displayed.

Example:

Q3) The C program to implement different operations on an ordinary queue. [File Name: u9q3.c]

```
#include <stdio.h>
#include <process.h>
#define QUEUE_SIZE 5
void main()
{
  void insert_rear(int, int*, int*);
  void delete_front(int *, int *, int *);
  void display(int *, int, int);
  int choice, item, f, r, q[10];
  /* Queue is empty */
  f = 0; /* Front end of queue*/
  r = -1; /* Rear end of queue*/
  for (;;)
  {
  clrscr();
 printf("\t\t\t  Ordinary Queue Operation\n\n");
 printf("\t\t\t  1 .... Push / Insert\n");
 printf("\t\t\t  2 .... Pop  / Delete\n");
 printf("\t\t\t  3 .... View / Display\n");
 printf("\t\t\t  4 .... Exit\n\n\n");
```

```
 printf("\t\t\t  Enter the choice : "); scanf("%d", &choice);
 switch (choice)
 {
  case 1:  // push into the queue
                    printf("Enter the item to be inserted : "); scanf("%d", &item);
                    insert_rear(item, q, &r);
                    continue;
                   case 2:    // pop from the queue
                    delete_front(q, &f, &r);
                    break;
                   case 3:     // display queue
                    display(q, f, r);
                    break;
                   case 4:
                    exit(0);
                   default:
                     printf("\t\t\tInvalid Input - Try Again");
                   }       // end of switch
    getch();
   }// end of for
}    // end of main
/*******************/
void insert_rear(int item, int q[], int *r)
{
                    if ( qfull(*r) ) /* Is queue full ? */
                    {
                       printf("\t\t\tQueue overflow\n");
                       return;
                     }
```

```
                /* Queue is not full */
                 q[++(*r)] = item;  /* Update rear pointer and insert a item */
}
/*------------------------------------------------------------- */
void delete_front(int q[], int *f, int *r)
{
                if ( qempty(*f, *r) )
                 {
                    printf("\t\t\tQueue underflow\n");
                    return;
                }
                printf(" Pop Successfull, element deleted = %d ", q[(*f)++]);
                  if(*f> *r)
                        {
                            *f=0,*r=-1;
                         }
}
/*********************************************************/
void display(int q[], int f, int r)
{
    int i;
    if ( qempty(f,r) )
    {
                printf("Queue is empty\n");
                 return;
            }
                printf("\t\t\t Queue Container\n\n");
                for(i=f;i<=r; i++)
                        printf("\t\t\t| %5d |\n",q[i]);
```

```
}
/*********************************************/
int qempty(int f, int r)
 {
  return ( f > r) ? 1 : 0;
/* returns true if queue is empty otherwise returns false */
 }
/*********************************************/
 int qfull(int r)
 {              /* returns true if queue is full otherwise false */
                  return ( r == QUEUE_SIZE-1 ) ? 1: 0;
 }
```

9.7.2 Disadvantage of Ordinary Queue

Consider the queue shown in figure 9.10. This situation does arise when 5 elements, say 10,20,30,40 and 50, are inserted and then first four items are deleted.

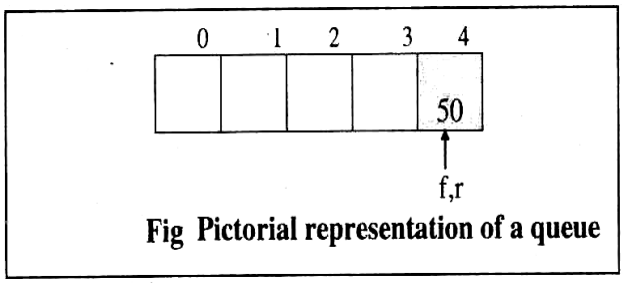


Fig. 9.10: Pictorial Representation of a Queue

If we try to insert an element say 60, since r has the value QUEUE_SIZE-1 (where QUEUE_SIZE is the maximum number of elements that can be stored in a queue), we get an overflow condition and it is not possible to insert any element. Even though queue is not full, in this case, it is not possible to insert any item into the queue. This disadvantage can be

overcome if we use the circular queue, which will be discussed in Circular queue.

9.7.3 Double-ended Queue (Deque)

Another type of queue called double-ended queue also called Deque is discussed in this section. Deque is a special type of data structure in which insertions and deletions will be done either at the front end or at the rear end of the queue. The operations that can be performed on deques are

- Insert an item from front end
- Insert an item from rear end
- Delete an item from front end
- Delete an item from rear end
- Display the contents of queue

The three operations insert rear, delete front and display and the associated operations to check for an underflow and overflow of queue have already been discussed in ‘ordinary queue’. In this section, other two operations i.e., insert an item at the front end and delete an item from the rear end are discused.

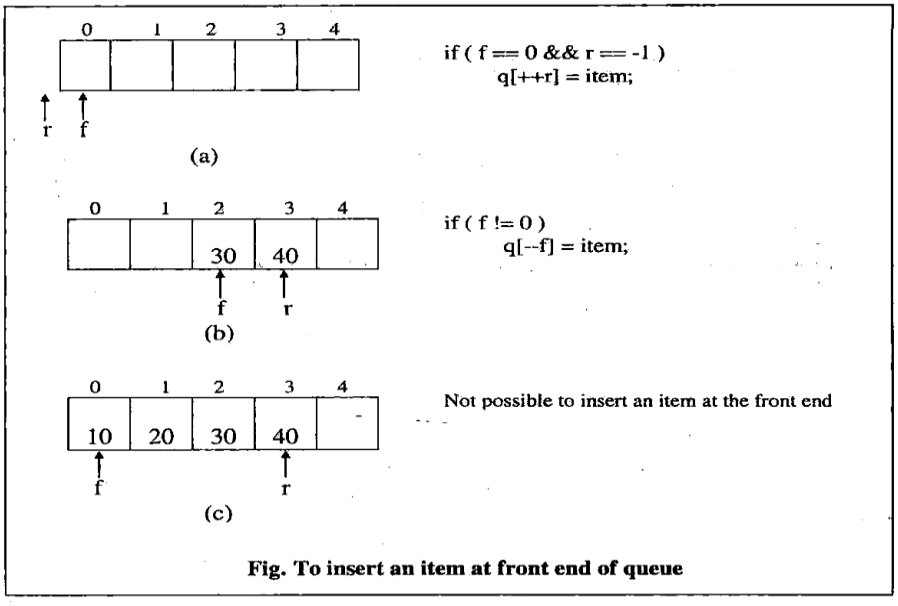


Fig. 9.11: To Insert an Item at front end of Queue

Example:

Q4) C program to implement a double ended queue (dqueue)

[File Name: u9q4.c]

```
#include <stdio.h>
#include <process.h>
#define QUEUE_SIZE 5
void main()
{
  void insert_rear(int,int*, int*);
  void insert_front(int,int*, int*, int *);
  void delete_rear(int *, int *, int *);
  void delete_front(int *, int *, int *);
  void display(int *, int, int);
  int choice, item, f, r, q[10];
  f = 0; /* Front end of queue*/
```

```
r = -1; /* Rear end of queue*/
for (;;)
{
        clrscr();
        printf("\t\t\t  Ordinary Queue Operation\n\n");
        printf("\t\t\t  1 .... Insert Front\n");
        printf("\t\t\t  2 .... Insert Rear\n");
        printf("\t\t\t  3 .... Delete Front\n");
        printf("\t\t\t  4 .... Delete Rear\n");
        printf("\t\t\t  5 .... View / Display\n");
        printf("\t\t\t  6 .... Exit\n\n\n");
        printf("\t\t\t  Enter the choice : "); scanf("%d", &choice);
        switch (choice)
        {
        case 1:
                printf("Enter the item to be inserted : "); scanf("%d",& item);
                insert_front(item, q, &f, &r);
                continue;
        case 2:  // push into the queue
                printf("Enter the item to be inserted : "); scanf("%d",&item);
                insert_rear(item, q, &r);
                continue;
        case 3:    // pop from the queue
                delete_front(q, &f, &r);
                break;
        case 4:
                delete_rear(q, &f, &r);
                 break;
        case 5:     // display queue
```

```
                display(q, f, r);
                break;
        case 6:
                exit(0);
        default:
                 printf("\t\t\tInvalid Input - Try Again");
         }       // end of switch

   getch();
  }// end of for
}    // end of main
/******************/
void insert_rear(int item, int q[], int *r)
{
                if ( qfull(*r) )
                {
                   printf("\t\t\tQueue overflow\n");
                   return;
                 }
            q[++(*r)] = item;
}
void delete_front(int q[], int *f, int *r)
{
                 if ( qempty(*f, *r) )
                  {
                     printf("\t\t\tQueue underflow\n");
                     return;
                  }
                  printf(" Pop Successfull, element deleted = %d ", q[(*f)++]);
```

```
                    if(*f> *r)
                        {
                            *f=0,*r=-1;
                        }
}

void display(int q[], int f, int r)
{
    int i;
    if ( qempty(f, r) )
    {
            printf("Queue is empty\n");
             return;
        }
            printf("\t\t\t Queue Container\n\n");
            for(i=f;i<=r; i++)
                        printf("\t\t\t| %5d |\n",q[i]);
}
        int qempty(int f, int r)
         {
          return ( f > r) ? 1 : 0; /* returns true if queue is empty otherwise
returns false */
         }
 int qfull(int r)
 {      /* returns true if queue is full otherwise false */
            return ( r == QUEUE_SIZE-1 ) ? 1: 0;
 }
void delete_rear(int q[], int *f, int *r)
{
```

```
                if ( qempty(*f, *r) )
                    {
                      printf("Queue underflow\n");
                          return;
                     }
                     printf("The element deleted is %d\n", q[(*r)--]);
                     if (*f > *r)
                          {
                             *f = 0, *r = -1 ;
                          }
}
void insert_front(int item, int q[ ], int *f, int *r)
{
            if( *f== 0 && *r == -1)
                          q[++(*r)] = item;
              else if ( *f != 0)
                          q[--(*f)]=item;
              else
                           printf("Front insertion not possible\n");
}
```

9.7.4 Circular Queue

In an ordinary queue, as an item is inserted, the rear end identified by r is incremented by 1. Once r reaches QUEUE_SIZE -1, we say queue is full. Note that even if some elements are deleted from queue, because the rear end identified by r is still equal to QUEUE_SIZE-l, item cannot be inserted. This disadvantage is overcome by using circular queue. The pictorial representation of a circular queue and its equivalent representation using an array are given in figure 9.12.

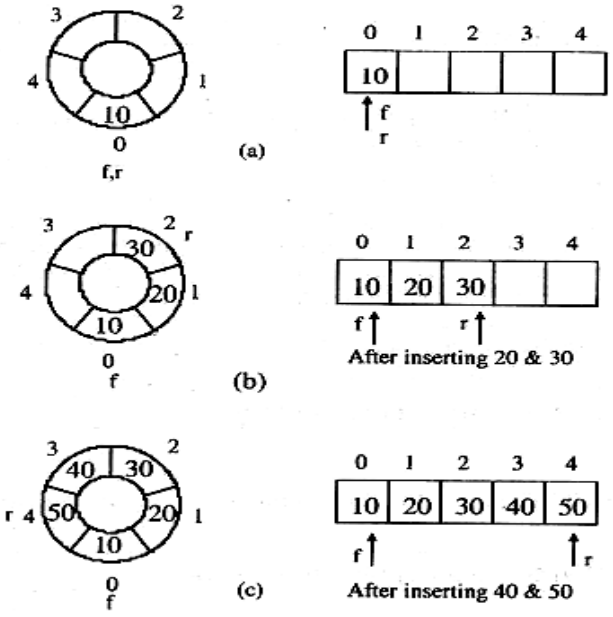
Assume that circular queue contains only one item as shown in figure 9.13(a). In this case, the rear end identified by r is 0 and front end identified by f is also 0. Since rear end is incremented while insertion, just before inserting the first item, the value 'of r should be -1 (Note: An item is inserted only at the rear end and so, only r is incremented by 1 and not f) so that after insertion, f and r points to an item 10. So, naturally, the initial values of f and r should be 0 and -1.

The configuration shown in below figure (b) is obtained after inserting 20 and 30. To insert an item, the rear pointer r has to be incremented first. For this, any of the two statements shown below can be used.

r = r + l or r = (r + 1) %QUEUE_SIZE

Both statements will increment r by 1. But, we prefer the second statement. We see why this method is preferred instead of a simple statement

r= r+1



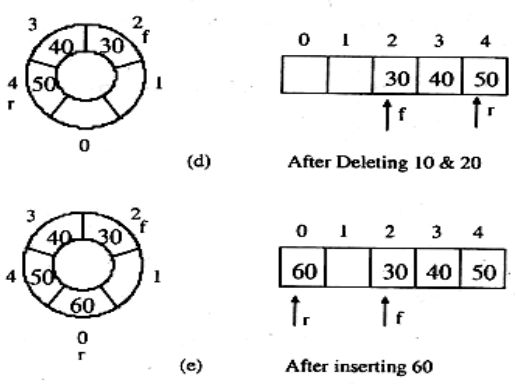


Fig. 9.12: Circular Queue and its Representation

The queue shown in fig.(c) is obtained after inserting 40 and 50. Note that at this point, the queue is full. Suppose we delete two items 10 and 20 one after the other. The resulting queue is shown in figure 9.12 (d).

r= r+1

is used to increment rear pointer, the value of *r* will be 5. But because this is a circular queue r should point to 0. This can be achieved using the statement

r = (r + 1) % QUEUE_SIZE

After execution of the above statement *r* will be 0. If this approach is used to check for overflow or underflow, we use a variable *count* that contains the number of items in the queue at any instant. So as an item is inserted, increment *count* by 1 and as an item is deleted decrement *count* by 1. By this it is ensured that at any point of time, *count* always contains the total number of elements in the queue. So, if queue is empty, *count* will be 0 and if queue is full, *count* will be QUEUE_SIZE. Thus we can easily implement the two functions *qfull()* and *qempty( )* .

Example:

[File Name: u9q5.c]

```
#include <stdio.h>
#include <process.h>
#include <alloc.h>
#include <string.h>
#define QUEUE_SIZE 5
/* function to check for underflow */
int qempty(int count)
{
    return ( count == 0 ) ? 1: 0;
}
/* function to check for overflow */
int qfull(int count)
{
   return ( count == QUEUE_SIZE ) ? 1: 0;
}
/* function to insert an item at the rear end */
void insert_rear(char item[], char q[ ][30], int *r, int *count)
{
          if ( qfull(*count) )
           {
             printf("Overflow of queue\n"); getch();
             return;
            }
             *r = (*r + 1) % QUEUE_SIZE;
               strcpy (q[*r],item);
               *count += 1 ;
}
```

```
/* function to delete an item from the front end */
void delete_front(char q[][30], int *f, int *count)
{
            if ( qempty(*count) )
                 {
                   printf("Underflow of queue\n");  getch();
                   return;
                  }
                  printf (" Pop Successful , deleted element = %s\n",q[*f]);
                  *f = (*f + 1) % QUEUE_SIZE;
                  *count-= 1;
}
/* function to display the contents of the queue */
void display(char q[][30], int f, int count)
{
          int i,j;
           if (qempty (count) )
           {
             printf("Q is empty\n");
              return;
            }
             printf("Contents of queue is\n\n\n");
              i= f;
             for (j = 1;j <= count; j++)
             {
                   printf(" |%10s|\n",q[i]);
                   i = (i + 1) % QUEUE_SIZE;
             }
             printf("\n");
```

```
}
/**********************************/
void main()
{
        int choice,f,r,count;
         char item[20],q[20][30];
         f=0;
          r = -1;
          count = 0; /* queue is empty */
          for (;;)
           {
                clrscr();
                printf("\n\t\t Circular Queue\n");
                printf("\t\t~~~~~~~~~~~~~~~~\n");
                printf("\t\t1: Insert \n");
                printf("\t\t2: Delete\n");
                printf("\t\t3: Display\n");
                printf("\t\t4: Exit\n\n\n\n");
                printf("\t\tEnter the choice : ");
                scanf("%d",&choice);
                switch ( choice )
                {
                        case 1:
                                printf ("Enter the item to be inserted : ");
                                scanf("%s", item);
                                insert_rear(item, q,& r ,&count);
                                continue;
                        case 2:
                                delete_front(q, &f, &count);
```

```
                    break;
                case 3:
                    display(q, f, count);
                    break;
                case 4:
                    exit(0);
                default:
                    printf("\t\tInvalid Input - Try Again");
                } // end of switch
                getch();
            }
}
/*********************************/
```

9.7.5 Priority queue

A Priority Queue is a collection of elements such that each element has been assigned a priority so that the order in which elements are deleted and processed comes from the following rules.

1) An element of higher priority is usually processed before any element of lower priority.
2) Two elements with the same priority are processed according to the order, in which they are added to the queue.

The priority queue obviously fits in well with certain types of tasks. For example, the scheduler in an operating system might use a priority queue to track processes running in the operating system.

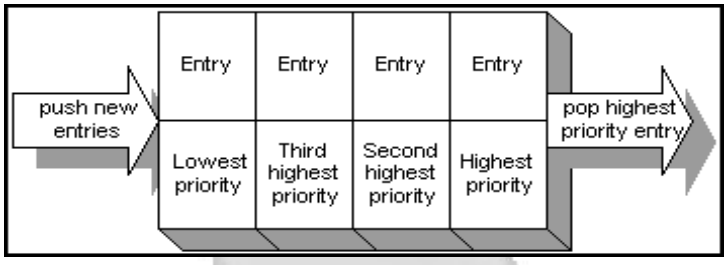


Fig. 9.13

Example:

Q6) Program to implement priority queue [File Name : u9q6.c]

```
/**** PRIORITY QUEUE  USING ARRAY ****/
#define ROW 20
#define COL 20
int priority_q[ROW][COL];
int priority_front[COL];
int priority_rear[COL];
void main()
{
char choice, menu();
int priority, value,i,j;
void push(), list(), pop();
do{
switch(menu())
{
case '1':
  push();  break;
case '2':
  list(); break;
case '3':
  pop(); break;
```

```
case '4':
   exit();
}
}while(1);
}

/*****************************************************/
void list()
{
int i, j;
clrscr();
printf("\t\t\t   The priority Queue is given below\n");
printf("\t\t\t~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n");
printf("\t\t\t   Priority ..... Value(s)\n");
for(i=0; i<ROW; i++)
{
 printf("\t\t\t%5d ->\t  ",i+1);
 for(j=priority_front[i]; j<priority_rear[i];j++)
  printf("%4d", priority_q[i][j]);
 printf("\n");
}
printf("\t\t\t~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n");
getch();
}
/*********************************************************/
void push()
{
int priority, value;
char choice;
```

```
clrscr();
printf("\nEnter Values in Priority Queue...\n\n");
do{
  printf("\nEnter Priority Value : ");   scanf("%d", &priority);
  if(priority>ROW)
  {
   printf("The Priority must not exit %d", ROW); getch();
   choice='Y'; continue;
  }
  printf("Enter Value         : ");   scanf("%d", &value);
  priority_q[priority-1][priority_rear[priority-1]]=value;
  priority_rear[priority-1]++;
  printf("Continue...[y/n] : ");
  choice=toupper(getche());
 }while(choice=='Y');
}
/*********************************************************************/
void pop()
{
int priority;
printf("\n\nEnter the priority : "); scanf("%d", &priority);
priority_front[priority-1]++;
}
/*********************************************************************/
char menu()
{
  clrscr();
  printf("\n\n\t\t\t M A I N    M E N U \n\n");
  printf("\t\t1....Push to  Priority Queue\n");
```

```
  printf("\t\t2....List to  Priority Queue\n");
  printf("\t\t3....Pop  the Priority Queue\n");
  printf("\t\t4....Exit\n");
  printf("\t\tYour Choice please [1-4] : ");
  return(toupper(getche()));
 }
/************************************************************/
```

9.8 Applications of Queues

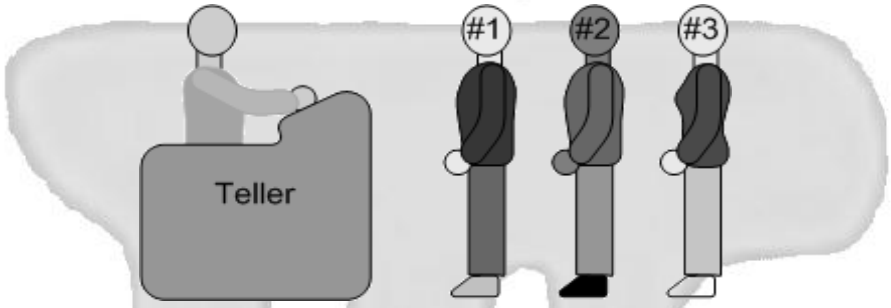


Fig. 9.15: A Queue

In general, queues are often used as "waiting lines". Here are a few examples of where queues would be used:

- In operating systems, for controlling access to shared system resources such as printers, files, communication lines, disks and tapes.

  A specific example of print queues is as follows:

  - In the situation where there are multiple users or a networked computer system, you probably share a printer with other users. When you request to print a file, your request is added to the print queue. When your request reaches the front of the print queue, your file is printed. This ensures that only one person at a time has access to the printer and that this access is given on a first-come, first-served basis.

- For simulation of real-world situations. For instance, a new bank may want to know how many tellers to install. The goal is to service each customer within a "reasonable" wait time, but not to have too many tellers for the number of customers. To find out a good number of tellers, they can run a computer simulation of typical customer transactions using queues to represent the waiting customers.
- When placed on hold for telephone operators. For example, when you phone the toll-free number for your bank, you may get a recording that says, "Thank you for calling A-1 Bank. Your call will be answered by the next available operator. Please wait." This is a queuing system.

9.9 Summary

In computer science, a Stack and Queues are abstract data type and data structure based on the principle of Last in First out (LIFO) and First in First out (FIFO) respectively. Stacks are used extensively at every level of a modern computer system. For example, a modern PC uses stacks at the architecture level, which are used in the basic design of an operating system for interrupt handling and operating system function calls. A queue is a pile in which items are added a one end and removed from the other. This unit covers algorithms and applications of stack and all kinds of queues.

Self Assessment Questions

1. Stack operates on FIFO. (True / False)
2. We can get any data randomly from Stack. (True / False)
3. The Polish Notation comes in the honour of Jan Luksiewiez. (True / False)
4. ab+ is the prefix notation. (True / False)
5. Queue maintains the algorithm LIFO. (True / False)

9.10 Terminal Questions

1. Define Stack. Discuss the Push and Pop Operations.
2. Write a C Program to implement the stack operations using arrays.
3. Discuss the various STACK applications with a suitable examples.
4. Explain how Stacks are useful in evaluation of arithmetic expressions with an example.
5. Explain the procedure for evaluating a Postfix Expression using a suitable example.
6. Illustrate the C program to represent the Stack Implementation on POP and PUSH operation.
7. Explain the ordinary Queue Insert at the rear end with a suitable example.
8. Discuss the deletion of an item from queue with suitable examples.
9. Write a C Program to display the contents from a queue.

9.11 Answers to Self-Assessment and Terminal Questions

Self Assessment Questions

1. False
2. False
3. True
4. False
5. False

Terminal Questions

1. Section 9.2
2. Section 9.3
3. Section 9.5
4. Section 9.5
5. Section 9.6.2
6. Section 9.4.2
7. Section 9.8.1
8. Section 9.7