2.1 STACK

Stack is a linear data structure and very much useful in various applications of computer science.

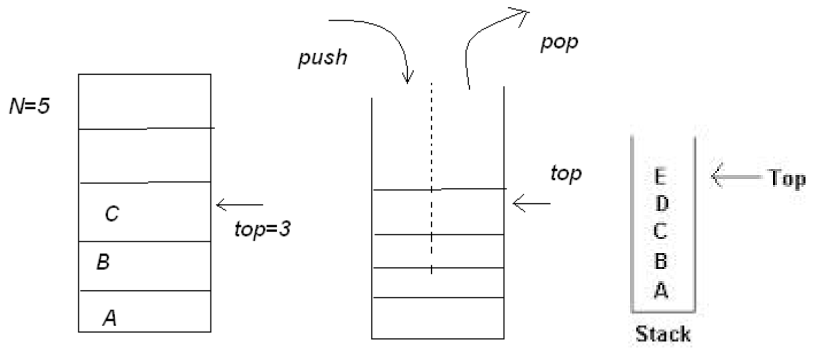
Definition

A stack is an ordered collection of homogeneous data elements, where the insertion and deletion operation takes place at one end only, called the top of the stack.

Like array and linked list, stack is also a linear data structure, but the only difference is that in case of former two, insertion and deletion operations can take place at any position.

In a stack the last inserted element is always processed first. Accordingly, stacks are called as Last-in-First-out (**LIFO**) lists. Other names used for stacks are **"piles"** and "**push-down lists**".

Stack is most commonly used to store local variables, parameters and return addresses when a function is called.



Stack Model of Stack

The above figure is a pictorial representation of a stack. N=5 is the maximum capacity of the stack. Currently there are three elements in the stack, so the variable top value is 3.The variable top always keeps track of the top most element or position.

STACK OPERATIONS

1. PUSH: "**Push**" is the term used to insert an element into a stack.
2. POP: "**Pop**" is the term used to delete an element from a stack.

Two additional terms used with stacks are "overflow" & "underflow". Overflow occurs when we try to push more information on a stack that it can hold. Underflow occurs when we try to pop an item from a stack which is empty.

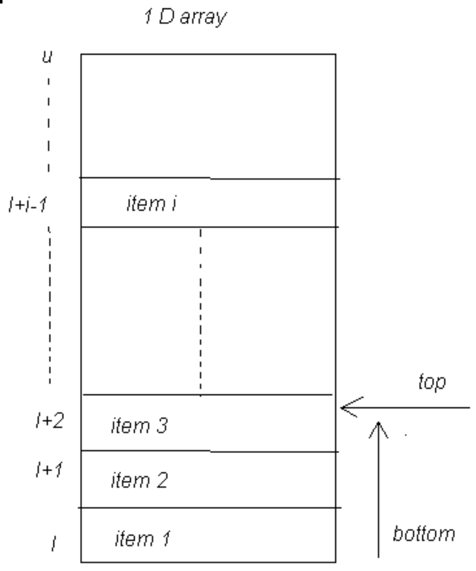
2.2 REPRESENTATION OF STACKS

A stack may be represented in the memory in various ways. Mainly there are two ways. They are:

1. Using one dimensional arrays(Static Implementation)
2. Using linked lists(Dynamic Implementation)

2.2.1 Representation of Stack using Array

First we have to allocate a memory block of sufficient size to accommodate the full capacity of the stack. Then, starting from the first location of the memory block, items of the stack can be stored in sequential fashion.



In the figure *Item*$_i$ denotes the ith item in the stack; l & u denote the index range of array in use; usually these values are 1 and size respectively. Top is a pointer to point the position of array upto which it is filled with the items of stack. With this representation following two statuses can be stated:

EMPTY: Top<l

FULL: Top>=u+l-1

Algorithms for Stack Operations

Algorithm PUSH (STACK, TOP, MAXSTK, ITEM)

This algorithm pushes an ITEM onto a stack.

```
Step 1:   If TOP = MAXSTK, then:          \\ Check Overflow?
              Print: OVERFLOW, and Exit.
Step 2:   Set TOP := TOP + 1.             \\ increases TOP by 1
Step 3:   Set STACK[TOP] := ITEM.         \\ Inserts ITEM in new TOP position.
Step 4:   Exit.
```

Here we have assumed that array index varies from 1 to SIZE and Top points the location of the current top most item in the stack.

Algorithm POP (STACK, TOP, ITEM)

This algorithm deletes the top element of STACK and assigns it to the variable ITEM.

```
Step 1:  If TOP = NULL, then:             \\ Check Underflow.
             Print: UNDERFLOW, and Exit.
Step 2:  Set ITEM := STACK[TOP].          \\ Assigns TOP element to ITEM.
Step 3:  Set TOP := TOP – 1.              \\ Decreases TOP by 1.
Step 4:  Exit.
```

/* C implementation of Stack using Arrays */

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#define size 5
struct stack
{
        int s[size];
        int top;
} st;

int isfull() {
  if (st.top >= size - 1)
    return 1;
  else
    return 0;
}
void push(int item) {
  st.top++;
  st.s[st.top] = item;
}

int isempty() {
  if (st.top == -1)
    return 1;
  else
    return 0;
}

int pop() {
  int item;
  item = st.s[st.top];
  st.top--;
  return (item);
}

void display() {
  int i;
  if (isempty())
    printf("\nStack Is Empty!");
  else {
    for (i = st.top; i >= 0; i--)
      printf("\n%d", st.s[i]);
  }
}

int main() {
  int item, choice;
  char ans;
  st.top = -1;
  clrscr();
  printf("\nImplementation Of Stack");
printf("\n----------------------------");
  do {
    printf("\nMain Menu");
    printf("\n1.Push    \n2.Pop    \n3.Display
\n4.exit");
    printf("\nEnter your choice:");
    scanf("%d", &choice);
    switch (choice) {
    case 1:
      printf("\nEnter   the   item   to   be
pushed:");
      scanf("%d", &item);
      if (isfull())
        printf("\nStack is Full!");
      else
        push(item);
      break;
    case 2:
      if (isempty())
        printf("\nEmpty   stack...!   Underflow
!!");
      else {
        item = pop();
        printf("\nThe popped element is %d",
item);
      }
      break;
    case 3:
      display();
      break;
    case 4:
      exit(0);
    }
    printf("\nDo you want to continue?");
    ans = getche();
  } while (ans == 'Y' || ans == 'y');
  return 0;
}
```

2.2.2 Linked representation of Stacks

The linked representation of a stack, commonly termed linked stack is a stack that is implemented using a singly linked list. The INFO fields of the nodes hold the elements of the stack and the LINK fields hold pointers to the neighboring element in the stack. The START pointer of the linked list behaves as the TOP pointer variable of the stack and the null pointer of the last node in the list signals the bottom of the stack.

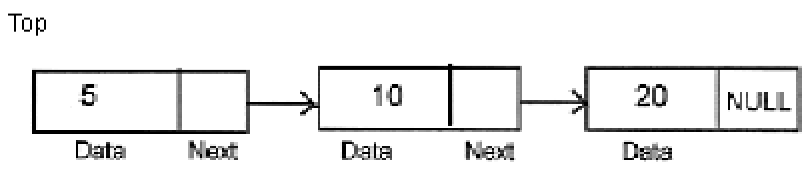


Fig. Linked list representation of a stack

Algorithms for Stack Operations

Algorithm: PUSH_LINKSTACK(INFO, LINK, TOP, AVAIL, ITEM).
This algorithm pushes an ITEM into a linked stack.

```
Step 1:  If AVAIL = NULL, then:                 \\ Check Overflow?
         Write: OVERFLOW, and Exit.
Step 2:  [Take node from AVAIL list.]
         Set NEW := AVAIL, and  AVAIL := LINK[AVAIL].
Step 3:  [Copies ITEM into new node.]
         Set INFO[NEW] := ITEM.
Step 4:  [Add this node to the STACK.]
         Set LINK[NEW] := TOP, and TOP := NEW.
Step 5:  Exit.
```

Algorithm: POP_LINKSTACK(INFO, LINK, TOP, AVAIL, ITEM).

This algorithm deletes the top element of a linked stack and assigns it to the variable ITEM.

```
Step 1:   If TOP =  NULL, then:            \\ Check Underflow?
              Write: UNDERFLOW and Exit.
Step 2:   Set ITEM := STACK[TOP].          \\ Copies the TOP element of STACK into ITEM.
Step 3:   Set TEMP := TOP, and
          TOP := LINK[TOP].                \\  Reset the position of TOP.
Step 4:   [Add node to the AVAIL list.]
              Set LINK[TEMP] := AVAIL,   AVAIL := TEMP.
Step 5:   Exit.
```

2.3 APPLICATIONS OF STACKS

1. Reversing elements in a list or string.
2. Recursion Implementation.
3. Memory management in operating systems.
4. Evaluation of postfix or prefix expressions.
5. Infix to postfix expression conversion.
6. Tree and Graph traversals.
7. Checking for parenthesis balancing in arithmetic expressions.
8. Used in parsing.

2.4 Stack as ADT

The Stack ADT implementation in C is quite simple. Instead of storing data in each node, we store pointer to the data.

ADT implementation of Stacks

The node structure consists only of a data pointer and link pointer. The Stack head structure also contains only two elements – a pointer to the top of the stack and a count of the number of entries in the stack

2.5 Arithmetic Expressions

An expression is a collection of operators and operands that represents a specific value. In above definition,

- ***Operator*** is a symbol (+,-,>,<,..) which performs a particular task like arithmetic or logical or relational operation etc.,
- ***Operands*** are the values on which the operators can perform the task.

Expression Types

Based on the operator position, expressions are divided into THREE types. They are as follows:

1. Infix Expression
2. Postfix Expression
3. Prefix Expression

Infix Expression

In infix expression, operator is used in between operands. This notation is also called as ***polish notation***. The general structure of an Infix expression is:

```
<Operand1> <Operator> <Operand2>          Ex:  A + B
```

Postfix Expression

In postfix expression, operator is used after operands. We can say that "Operator follows the Operands". This notation is also called as ***reverse - polish notation***. The general structure of Postfix expression is:

```
<Operand1> <Operand2> <Operator>          Ex:  A B +
```

Prefix Expression

In prefix expression, operator is used before operands. We can say that "Operands follows the Operator". The general structure of Prefix expression is:

```
<Operator><Operand1> <Operand2>           Ex:  + A B
```

In 2nd and 3rdnotations, parentheses are not needed to determine the order evaluation of the operations in any arithmetic expression.

Eg.:	**Infix**	**Prefix**	**Postfix**	
1.	A+B*C	A+{*BC}	A+{BC*}	
		+A*BC	ABC*+	{ } indicate partial translation
2.	(A+B)*C	{+AB}*C	{AB+}*C	
		*+ABC	AB+C*	

The computer usually evaluates an arithmetic expression written in infix notation in two steps.

1. It converts the expression to postfix notation and
2. It evaluates the postfix expression.

In each step, the stack is the main tool that is used to accomplish the given task.

2.5.1 Transforming Infix Expression into Postfix Expression

In the infix expression we assume only exponentiation(^ or **), multiplication(*), division(/), addition(+), subtraction(-) operations. In addition to the above operators and operands they may contain left or right parenthesis. The operators three levels of priorities are:

Priority	Operators
HIGHEST	^ or **
NEXT HIGHEST	* , /
LOWEST	+ , -

The following algorithm transforms the infix expression Q into the equivalent postfix expression P. The algorithm uses a stack to temporarily hold operators and left parenthesis.

ALGORITHM INFIX-TO-POSTFIX (Q)

```
Step1:  Push '(' onto stack and add ')' to the end of Q.
Step 2: Scan Q from left to right and repeat steps 3 to 6 for each element of Q until STACK is empty.
Step 3: If an operand is encountered, add it to P.
Step 4: If a left parentheses is encountered, push it onto STACK.
Step 5: If an operator ϴ is encountered, then:
            a) Repeatedly pop from STACK and add to P each operator (on the top of
               STACK) which has the same precedence as or higher precedence than ϴ.
            b) Add ϴ to the STACK
        [End of If Structure]
Step 6: If a right parentheses is encountered, then:
            a) Repeatedly pop from STACK and add to P each operator (on the top of
               STACK) until a left parentheses is encountered.
            b) Remove the left parenthesis. [Do not add this to P]
Step 7: Exit.
```

EXAMPLE 1: Consider the following infix expression Q: A + (B * C - (D / E ^ F) * G) * H

Symbol Scanned	**Stack Contents**	**Expression P**
Initial	(	
(1) A	(	A
(2) +	(+	A
(3) (	(+ (	A
(4) B	(+ (	AB
(5) *	(+ (*	AB
(6) C	(+ (*	ABC
(7) -	(+ (-	ABC *
(8) (	(+ (- (	ABC*
(9) D	(+ (- (	ABC*D
(10) /	(+ (- (/	ABC*D
(11) E	(+ (- (/	ABC*DE
(12) ^	(+ (- (/ ^	ABC*DE
(13) F	(+ (- (/ ^	ABC*DE

(14))	(+ (-	ABC*DE^/
(15) *	(+ (- *	ABC*DE^/
(16) G	(+ (- *	ABC*DE^/G
(17))	(+	ABC*DE^/G*-
(18) *	(+ *	ABC*DE^/G*-
(19) H	(+ *	ABC*DE^/G*-H
(20))		ABC*DE^/G*-H * +

OUTPUT P: ABC*DE^/G*-H * + (POSTFIX FORM)

EXAMPLE 2: Consider the infix expression Q: (A+B)*C + (D-E)/F)

Symbol Scanned	Stack Contents	Expression P
Initial	(	
(1) (	((	
(2) A	((	A
(3) +	((+	A
(4) B	((+	AB
(5))	(	AB+
(6) *	(*	AB+
(7) C	(*	AB+C
(8) +	(+	AB+C*
(9) (	(+(	AB+C*
(10) D	(+(	AB+C*D
(11) -	(+(-	AB+C*D
(12) E	(+(-	AB+C*DE
(13))	(+	AB+C*DE-
(14) /	(+/	AB+C*DE-
(15) F	(+/	AB+C*DE-F
(16))		AB+C*DE-F/+

OUTPUT P: AB+C*DE-F/+ (POSTFIX FORM)

2.5.2 EVALUATION OF A POSTFIX EXPRESSION

Suppose P is an arithmetic expression written in postfix . The following algorithm uses a STACK to hold operands during the evaluation of P.

Algorithm EVALPOSTFIX(p)

```
Step 1:  Add a right parentheses ')' at the end of P.        //this acts as a sentinel
Step 2:  Scan P from left to right and repeat steps 3 and 4 for each element of P until the
         sentinel ')' is encountered.
Step 3:  If an operand is encountered, push on STACK.
Step 4:  If an operator ϴ is encountered, then:
```

```
            a) Remove the two top elements of STACK, where A is the top element and B is
               the next- to-top element.
            b) Evaluate B Θ A.
            c) Push the result of (b) back on STACK
        [End of If structure]
        [End of step2 loop]
Step 5: Set VALUE equal to top element on STACK.
Step6:  Return(VALUE)
Step 7: Exit.
```

EXAMPLE: Consider the following postfix expression:

P: 5, 6, 2, +, *, 12, 4, /, -

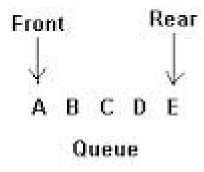
Symbols Scanned	STACK
(1) 5	5
(2) 6	5, 6
(3) 2	5, 6, 2
(4) +	5, 8
(5) *	40
(6) 12	40, 12
(7) 4	40, 12, 4
(8) /	40, 3
(9) -	37
(10))	

The result of this postfix expression is 37.

2.6 QUEUES

Queue is a linear data structure used in various applications of computer science. Like people stand in a queue to get a particular service, various processes will wait in a queue for their turn to avail a service.

Queue is a linear list in which insertions takes place at one end called the *rear* or *tail* of the queue and deletions at the other end called as *front* or *head* of the queue.

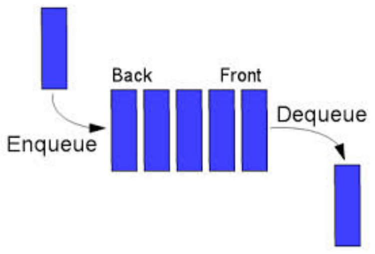


When an element is to join the queue, it is inserted at the rear end of the queue and when an element is to be deleted, the one at the front end of the queue is deleted automatically.

In queues always the first inserted element will be the first to be deleted. That's why it is also called as *FIFO – First-in First-Out* data structure (or FCFS – First Come First Serve data structure).

APPLICATIONS of QUEUE

- CPU Scheduling (Round Robin Algorithm)
- Printer Spooling
- Tree & Graph Traversals
- Palindrome Checker
- Undo & Redo Operations in some Software's



OPERATIONS ON QUEUE

The queue data structure supports two operations:

Enqueue: Inserting an element into the queue is called *enqueuing* the queue. After the data have been inserted into the queue, the new element becomes the rear.

Dequeue: Deleting an element from the queue is called *dequeuing* the queue. The data at the front of the queue are returned to the user and removed from the queue.

TYPES OF QUEUES

1. **Simple or Single Ended Queue**: In this queue insertions take place at one end and deletions take place at other end.
2. **Circular Queue**: It is similar to single ended queue, but the front is connected back to rear. Here the memory can be utilized effectively.
3. **Double Ended Queue**: In double ended queue both the insertions and deletions can take place at both the ends.
4. **Priority Queue**: In priority queue the elements are not deleted according to the order they entered into the queue, but according to the priorities associated with the elements.

2.6.1 IMPLEMENTATION OF QUEUES

Queues can be implemented or represented in memory in two ways:

1. Using Arrays (Static Implementation).
2. Using Linked Lists (Dynamic Implementation).

2.6.1.1 Implementation of Queue Using Arrays

A common method of implementing a queue data structure is to use another sequential data structure, viz, arrays. With this representation, two pointers namely, Front and Rear are used to indicate two ends of the queue. For insertion of next element, pointer Rear will be adjusted and for deletion pointer Front will be adjusted.

However, the array implementation puts a limitation on the capacity of the queue. The number of elements in the queue cannot exceed the maximum dimension of the one dimensional array. Thus a queue that is accommodated in an array Q[1:n], cannot hold more than *n* elements. Hence every insertion of an element into the queue has to necessarily test for a *QUEUE FULL* condition before executing the insertion operation. Again, each deletion has to ensure that it is not attempted on a queue which is already empty calling for the need to test for a *QUEUE EMPTY* condition before executing the deletion operation.

Algorithm of insert operation on a queue

```
Procedure INSERTQ (Q, n, ITEM, REAR)
// this procedure inserts an element ITEM into Queue with capacity n

Step 1:  if(REAR=n ) then
          Write:"QUEUE_FULL" and Exit
Step 2: REAR=REAR + 1                  //Increment REAR
Step 3: Q[REAR]= ITEM                  //Insert ITEM as the rear element
Step 4: Exit
```

It can be observed that addition of every new element into the queue increments the REAR variable. However, before insertion, the condition whether the queue is full is checked.

Algorithm of delete operation on a queue

```
Procedure DELETEQ (Q, FRONT, REAR, ITEM)
Step 1:  If (FRONT>REAR) then:
                Write: "QUEUE EMPTY" and Exit.
Step 2:  ITEM = Q[FRONT]
```

Step 3: FRONT=FRONT + 1
Step 4: Exit.

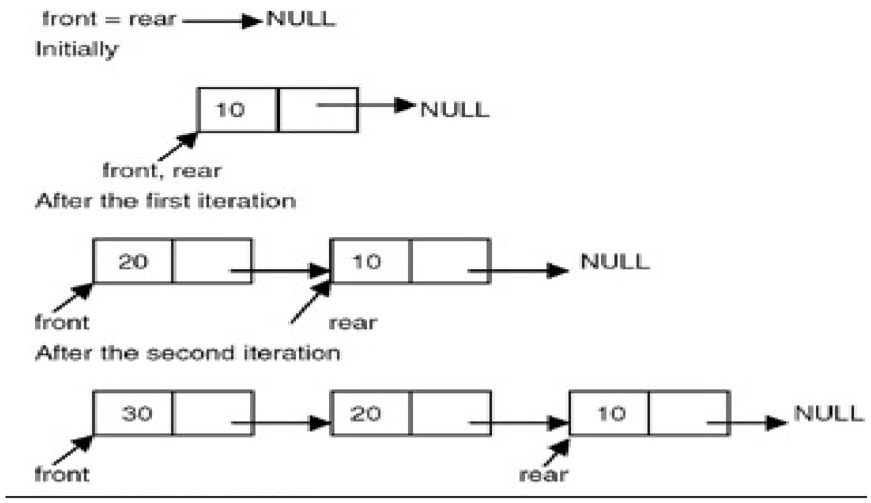
To perform delete operation, the participation of both the variables FRONT and REAR is essential. Before deletion, the condition FRONT=REAR checks for the emptiness of the queue. If the queue is not empty, the element is removed through ITEM and subsequently FRONT is incremented by 1.

/*C implementation of Queue using Arrays*/

```
#include <stdio.h>
#include <conio.h>
#define MAX 5

int insert();   /* function prototypes  */
int delete();
void display();

int queue[MAX], rear = -1, front = - 1;

main()
{
   int choice;
   clrscr();
   printf("Implmentation of Queue\n");
   printf("--------------------\n");
   while (1)
   {
        printf("1.Insert\n2.Delete\n3.Displa
y\n4.Quit\n");
        printf("Enter your choice : ");
        scanf("%d", &choice);
        switch (choice)
        {
           case 1:
           insert();
           break;
           case 2:
           delete();
           break;
           case 3:
           display();
           break;
           case 4:
           exit(1);
        } /*End of switch*/
   } /*End of while*/
} /*End of main()*/

int insert()
{
   int add_item;
   if (rear == MAX - 1)
   printf("Queue Overflow \n");
   else
   {
        if (front == - 1)
        /*If queue is initially empty */
        front = 0;
        printf("Inset the element in queue :
");
        scanf("%d", &add_item);
        rear = rear + 1;
        queue[rear] = add_item;
   }
   return;
} /*End of insert()*/

int delete()
{
   if (front == - 1 || front > rear)
   {
        printf("Queue Underflow \n");
        return ;
   }
   else
   {
        printf("Element deleted from queue
is : %d\n", queue[front]);
        front = front + 1;
   }
   return;
} /*End of delete() */

void display()
{
   int i;
   if (front == - 1)
        printf("Queue is empty \n");
   else
   {
        printf("Queue is : \n");
        for (i = front; i <= rear; i++)
           printf("%d ", queue[i]);
        printf("\n");
   }
} /*End of display() */
```

2.6.1.2 Linked Representation

A linked queue is a queue implemented as a linked list with two pointer variables FRONT and REAR pointing to the nodes which is in the FRONT and REAR of the queue.



Algorithm: LINKQ_INSERT(INFO, LINK, FRONT, REAR, ITEM, AVAIL).
This algorithm inserts an element ITEM into a linked queue.

```
Step 1:  [OVERFLOW?]
         If AVAIL := NULL, then:
              Write: OVERFLOW and Exit.
Step 2:  [Remove first node from AVAIL list.]
         Set NEW := AVAIL, and
         AVAIL := LINK[AVAIL].
Step 3:  Set INFO[NEW] := ITEM, and
         LINK[NEW] := NULL.      [Copies ITEM  into new node.]
Step 4:  If FRONT = NULL, then:
              Set FRONT := NEW and REAR := NEW.
              [If Q is empty then ITEM is the first element in the queue Q.]
         Else:
              Set LINK[REAR] := NEW, and
              REAR := NEW.
              [REAR points to the new node appended to the end of the list.]
Step 5: Exit.
```

Algorithm: LINKQ_DELETE(INFO, LINK, FRONT, REAR, ITEM, AVAIL).
This algorithm deletes the front element of the linked queue and stores it in ITEM.

```
Step 1:   [UNDERFLOW?]
          If FRONT = NULL, then:
              Write: UNDERFLOW and Exit.
Step 2:   Set ITEM := INFO[FRONT].          [Save the data value of FRONT.]
Step 3:   Set NEW := FRONT, and             [Reset FRONT to the next position.]
          Set FRONT := LINK[FRONT].
Step 4:   Set LINK[NEW] := AVAIL, and       [Add node to the AVAIL list.]
          AVAIL := NEW
Step 5:   Exit.
```

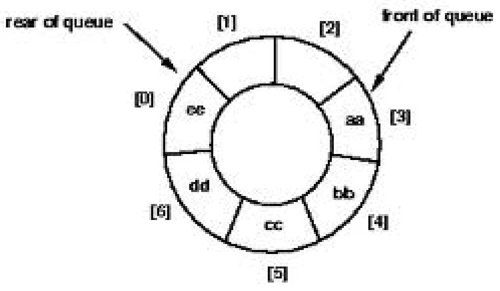
2.7 CIRCULAR QUEUE

One of the major problems with the linear queue is the lack of proper utilization of space. Suppose that the queue can store 10 elements and the entire queue is full. So, it means that the queue is holding 10 elements. In case, some of the elements at the front are deleted, the element at the last position in the queue continues to be at the same position and there is no efficient way to find out that the queue is not full.

In this way, space utilization in the case of linear queues is not efficient. This problem is arising due to the representation of the queue.

The alternative representation is to depict the queue as circular. In case, we are representing the queue using arrays, then, a queue with *n* elements starts from index 0 and ends at n-1. So, clearly, the first element in the queue will be at index 0 and the last element will be at n-1 when all the positions between index 0 and n-1 (both inclusive) are filled. Under such circumstances, front will point to 0 and rear will point to n-1. However, when a new element is to be added and if the rear is pointing to n-1, then, it needs to be checked if the position at index 0 is free. If yes, then the element can be added to that position and rear can be adjusted accordingly. In this way, the utilization of space is increased in the case of a circular queue.

In a circular queue, front will point to one position less to the first element. So, if the first element is at position 4 in the array, then the front will point to position 3. When the circular queue is created, then both front and rear point to index 1. Also, we can conclude that the circular queue is empty in case both front and rear point to the same index. Figure depicts a circular queue.



Enqueue(value) - Inserting value into the Circular Queue

```
Step 1: Check whether queue is FULL.
        If ((REAR == SIZE-1 && FRONT == 0) || (FRONT == REAR+1))
               Write "Queue is full " and Exit
Step 2: If ( rear == SIZE - 1 && front != 0 ) then:
        SET REAR := -1.
Step 4: SET REAR = REAR +1
Step 5: SET QUEUE[REAR] = VALUE
Step 6: Exit and check 'front == -1' if it is TRUE, then set front = 0.
```

deQueue() - Deleting a value from the Circular Queue

```
Step 1: Check whether queue is EMPTY?
         If (FRONT == -1 && REAR == -1)
               Write "Queue is empty" and Exit.
Step 2: DISPLAY QUEUE[FRONT]
Step 3: SET FRONT := FRONT +1 .
Step 4: If( FRONT = SIZE, then:
        SET FRONT := 0.
Step 5: Exit
```

/* Program to implement Circular Queue using Array */

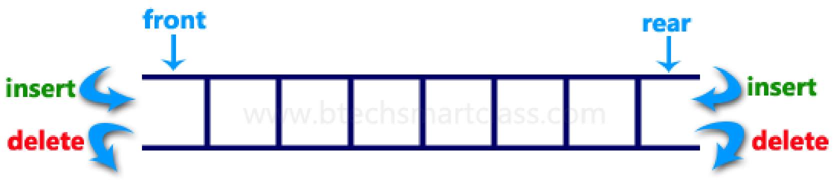
```
#include<stdio.h>
#include<conio.h>
#define SIZE 5

int cinsert(int);
int cdelete();
void display();

int cq[SIZE], front = -1, rear = -1;
void main()
{
  int choice, ele;
  clrscr();
  while(1){
     printf("CIRCULAR QUEUE IMPLEMENTATION\n");
     printf("----------------------------\n");
     printf("****** MENU ******\n");
     printf("1. Insert\n2. Delete\n3. Display\n4. Exit\n");
     printf("Enter your choice: ");
     scanf("%d",&choice);
     switch(choice){
         case 1: printf("\nEnter the value to be insert:  ");
                 scanf("%d",&ele);
                 cinsert(ele);
                 break;
         case 2: cdelete();
                 break;
         case 3: display();
                 break;
         case 4: exit(1);
     }  /*End of switch */
  }  /*End of while */
}  /*End of main */

int cinsert(int value)
{
  if((front == 0 && rear == SIZE - 1) || (front == rear+1))
     printf("\nCircular Queue is Full! Insertion not possible!!!\n");
  else{
     if(rear == SIZE-1 && front != 0)
         rear = -1;
     cq[++rear] = value;
     printf("\nInsertion Success!!!\n");
     if(front == -1)
         front = 0;
  }
  return;
}  /*End of cinsert() */

int cdelete()
{
  if(front == -1 && rear == -1)
     printf("\nCircular Queue is Empty! Deletion is not possible!!!\n");
  else{
     printf("\nDeleted element : %d\n",cq[front++]);
     if(front == SIZE)
         front = 0;
     if(front-1 == rear)
         front = rear = -1;
  }
  return;
} /*End of cdelete() */

void display()
{
  if(front == -1)
     printf("\nCircular Queue is Empty!!!\n");
  else{
     int i = front;
     printf("\nCircular Queue Elements are : \n");
     if(front <= rear){
         while(i <= rear)
            printf("%d\t",cq[i++]);
     }
     else{
         while(i <= SIZE - 1)
            printf("%d\t", cq[i++]);
         i = 0;
         while(i <= rear)
            printf("%d\t",cq[i++]);
     }
  }
}  /*End of display() */
```

2.8 DOUBLE ENDED QUEUE (DEQUEUE)

A Dequeue is a homogeneous list of elements in which insertions and deletion operations are performed on both the ends.Because of this property it is known as double ended queue i.e. Dequeue or deck. Deque has two types:

1. Input restricted queue: It allows insertion at only one end
2. Output restricted queue: It allows deletion at only one end

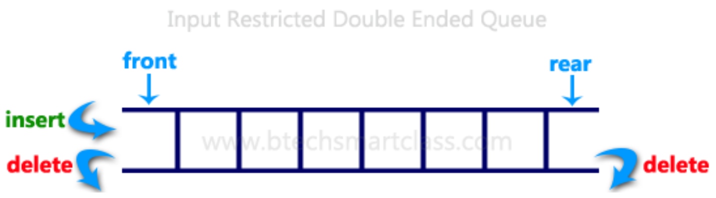
In dequeue four pointers are used. They are left front(lf), left rear(lr), right front(rf) and right rear(rr).



- *If (lf==lr) and (rf==rr) then deque is empty.*
- *If lr>rr then dequeue is full*
- For inserting we have to modify rear pointer. For deleting we have to modify front pointer.
- Always rear pointer is 1 position ahead of last element.
- After insertion on left side, left rear should be incremented.
- After insertion on right side, right rear should be decremented.

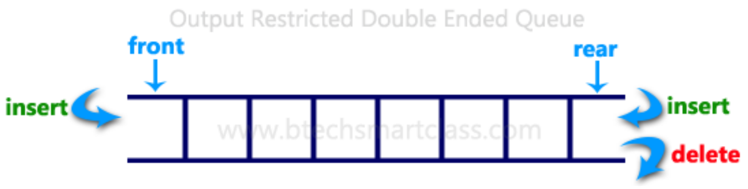
Input Restricted Double Ended Queue

In input restricted double ended queue, the insertion operation is performed at only one end and deletion operation is performed at both the ends.



Output Restricted Double Ended Queue

In output restricted double ended queue, the deletion operation is performed at only one end and insertion operation is performed at both the ends.



2.9 PRIORITY QUEUE

Def: *Priority queue is a variant of queue data structure in which insertion is performed in the order of arrival and deletion is performed based on the priority.*

In priority queue every element is associated with some priority. Normally the priorities are specified using numerical values. In some cases lower values indicate high priority and in some cases higher values indicate high priority

In priority queues elements are processed according to their priority but not according to the order they are entered into the queue.

For example, let P be a priority queue with three elements *a, b, c* whose priority factors are 2,1,1 respectively. Here, larger the number, higher is the priority accorded to that element. When a new element *d* with higher priority 4 is inserted, *d* joins at the head of the queue superseding the remaining elements. When elements in the queue have the same priority, then the priority queue behaves as an ordinary queue following the principle of FIFO amongst such elements.

The working of a priority queue may be likened to a situation when a file of patients waits for their turn in a queue to have an appointment with a doctor. All patients are accorded equal priority and follow an FCFS scheme by appointments. However, when a patient with bleeding injuries is brought in, he/she is accorded higher priority and is immediately moved to the head of the queue for immediate attention by the doctor. This is priority queue at work.

There are two types of priority queues they are as follows...

1. Max Priority Queue
2. Min Priority Queue

Max Priority Queue

In max priority queue, elements are inserted in the order in which they arrive the queue and always maximum value is removed first from the queue. For example assume that we insert in order 8, 3, 2, 5 and they are removed in the order 8, 5, 3, 2.

There are two representations of max priority queue.

1. Using One-Way List Representation
2. Using an Array

One-Way List Representation

One way to maintain a priority queue in memory is by means of a one-way list, as follows:

1. Each node in the list will contain three items of information:
    1. an information Feld INFO
    2. a priority number PRN, and
    3. a link number LINK
2. A node X precedes a node Y in the list
    a. when X has higher priority than Y or
    b. when both have same priority but X was added to the list before Y

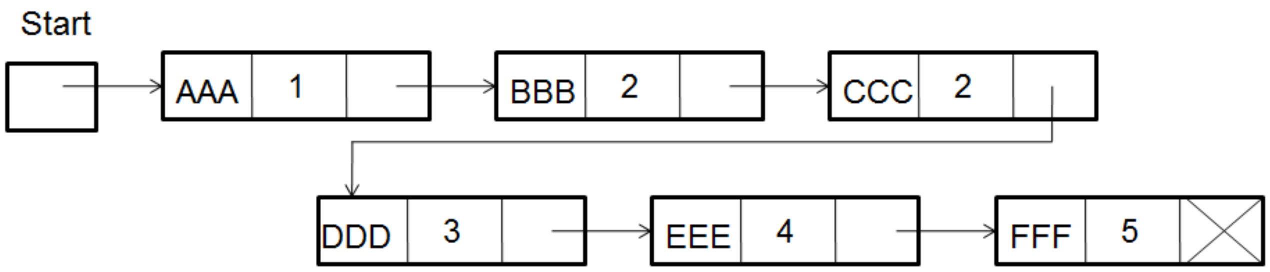


Fig.: Representation of Linked list

Algorithms for insertion and deletion

Insertion: Find the location of Insertion
Step 1: Add an ITEM with priority number N
Step 2: Traverse the list until finding a node X whose priority exceeds N. Insert ITEM in front of node X.
Step 3: If no such node is found, insert ITEM as the last element of the list.

Deletion: Delete the first node in the list.

Array representation

- ✓ Separate queue for each level of priority.
- ✓ Each queue will appear in its own circular array and must have its own pair of pointers, FRONT and REAR.
- ✓ If each queue is given the same amount space then a 2D queue can be used

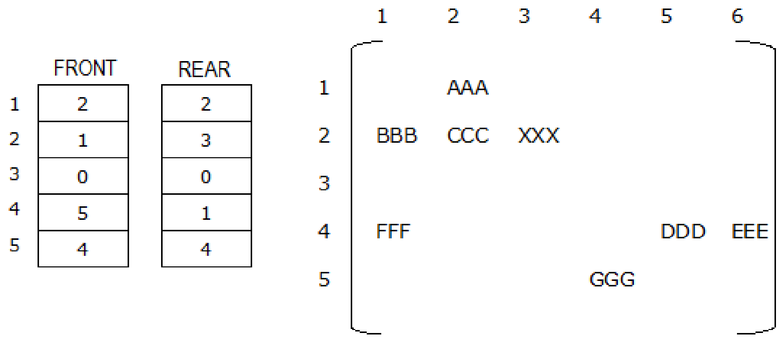


Fig. Array Representation with multiple queues

Deletion Algorithm

Step 1: Find the smallest K such that FRONT[K] ≠ NULL
Step 2: Delete and process the front element in row K of QUEUE
Step 3: Exit

Insertion Algorithm

Step 1: Insert ITEM as the rear element in row M of QUEUE
Step 2: Exit

Min Priority Queue

In min priority queue, elements are inserted in the order in which they arrive the queue and always minimum value is removed first from the queue. For example assume that we insert in order 8, 3, 2, 5 and they are removed in the order 2, 3, 5, 8.