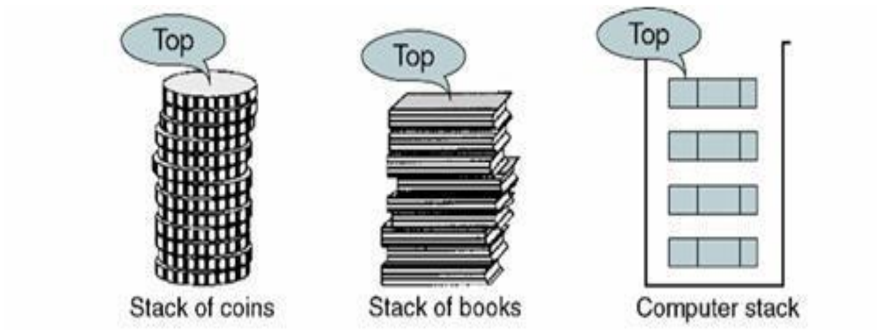
UNIT II LINEAR DATA STRUCTURES – STACKS, QUEUES

Stack ADT – Operations - Applications - Evaluating arithmetic expressions- Conversion of Infix to postfix expression - Queue ADT – Operations - Circular Queue – Priority Queue - deQueue – applications of queues.

2.1 CONCEPT OF STACK- STACK ADT

Stack is a linear data structure. Stack is an ordered collection of elements in which insertions and deletions are restricted to only one end. The end from which the elements are added and are removed is referred to as "TOP". The stack is also called as piles or push down list. The Mechanism which is followed in stack is Last In First Out (LIFO).



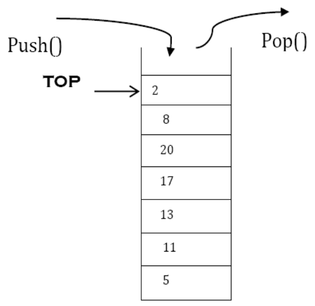


Fig: 2.1 Concept of Stack

Stack can be implemented by

- An Array
- Linked list.

Operations of Stack

- Push( ) : Insertion
- Pop( ) : Deletion

Possible ADT's of Stack using array:

1. Push
2. Pop
3. Is Empty
4. Is Full
5. Display top/peek( )
6. Display all
7. Search

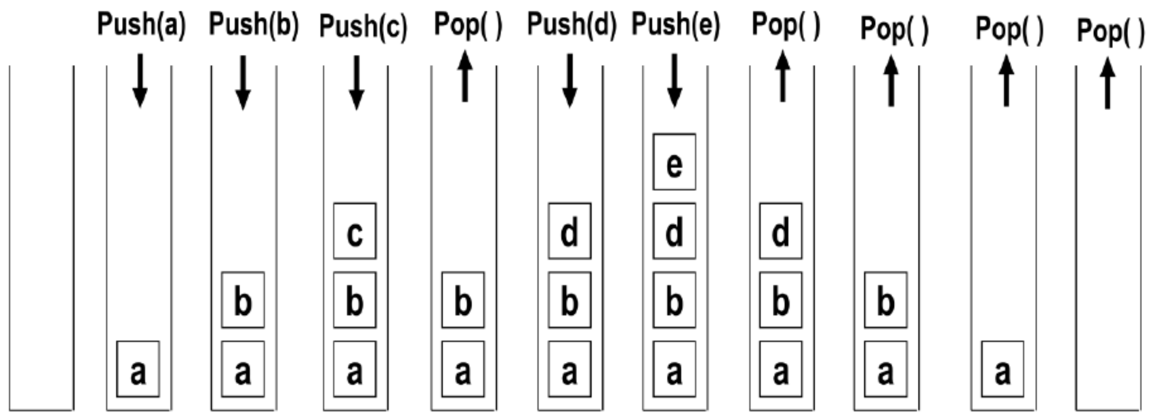
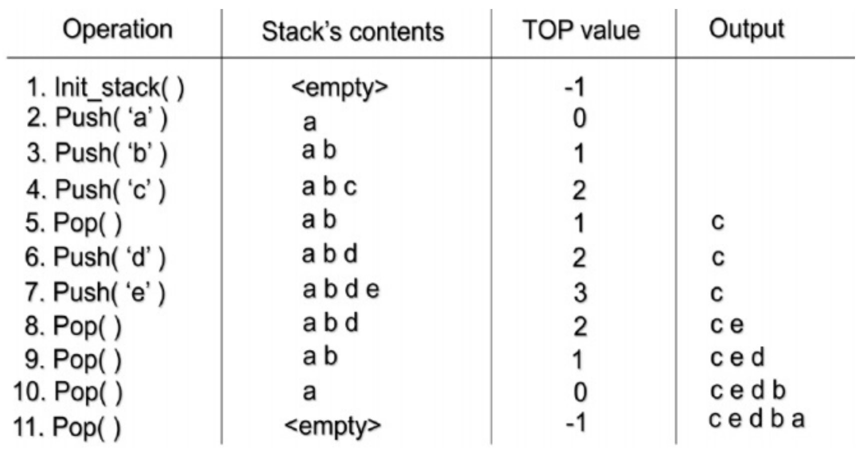


Fig: 2.2 Stack Operations

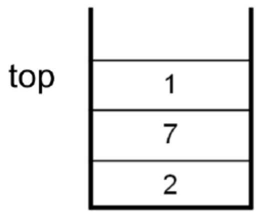
2.1.1 STACK OPERATIONS



Operation	Stack's contents	TOP value	Output
1. Init_stack( )	<empty>	-1	
2. Push('a')	a	0	
3. Push('b')	a b	1	
4. Push('c')	a b c	2	
5. Pop( )	a b	1	c
6. Push('d')	a b d	2	c
7. Push('e')	a b d e	3	c
8. Pop( )	a b d	2	c e
9. Pop( )	a b	1	c e d
10. Pop( )	a	0	c e d b
11. Pop( )	<empty>	-1	c e d b a

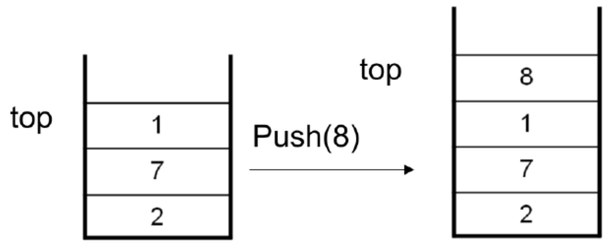
1. Creation :

```
Void create( int stack[ ], int top, int n)
{
for(top=0;top<n; top++)
scanf("%d", &stack[top]);
--top;
}
```



2. Push :

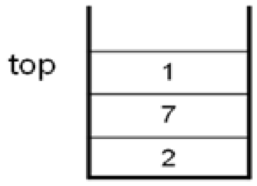
```
Void push(int stack[ ], int top, int num)
{
if(top== -1)
stack[++top]= num;
```



```
else if(top== size – 1 )
printf("Top= %d , Stack is full",top);
else
stack[++top]=num;
}
```

3. Display Top :

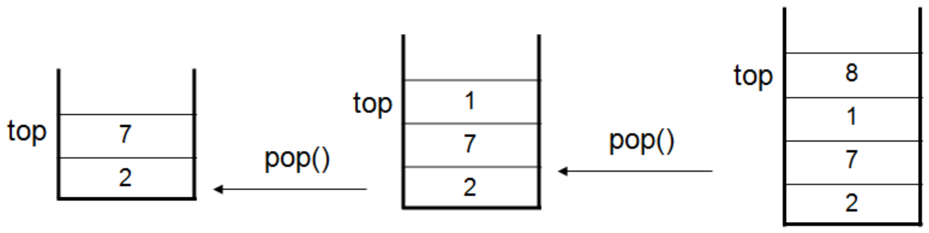
```
Void displaytop( int stack[ ], int top)
{
if(top== -1)
printf(" Stack is Empty");
else
printf("Top element is %d ", Stack[top]);
}
```



The top element is 1

4. Pop :

```
Void pop(int stack[ ], int top)
{
int x;
if(top== -1)
return 1;
else
{
x=stack[top];
--top;
} return x;
}
```



5. Display :

```
Void display(int stack[ ], int top)
{
int i;
for(i=top; i>=0; i--)
printf("%d", stack[i]);
}
```

6. Is Empty :

```
int isempty(int stack[ ],int top)
{
if(top== -1)
return 1; // Stack is empty
else
return 0; // Stack is not empty
}
```

7. Is Full :

```
int isfull(int stack[ ], int top)
{
if(top== size – 1)
return 1; //Full Stack
else
return 0; //Stack is not full.
}
```

Underflow – If Stack is empty and if we try to pop stack underflow occurs.
Overflow --- If Stack is full and if we try to push stack overflow occurs.

2.1.2 IMPLEMENTATION OF STACK USING LINKED LIST

Stack Using Linked List
Struct stack
{
Int data;
Struct node *next;
} *top=NULL;

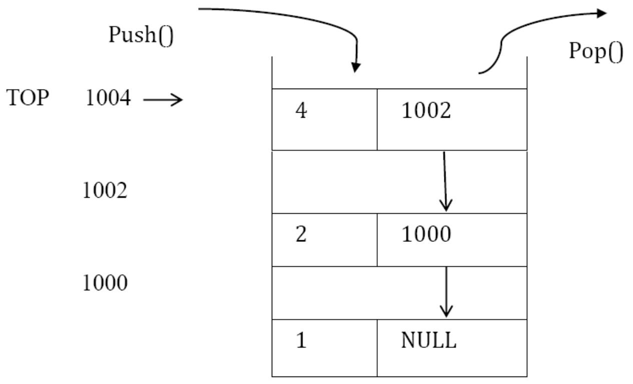


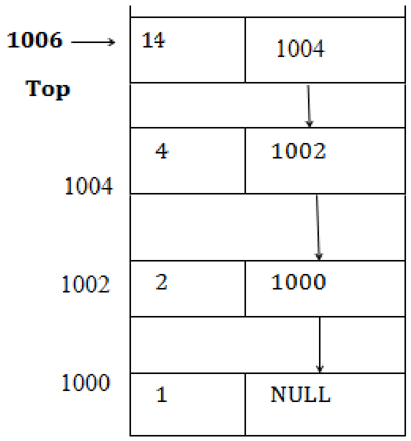
Fig: 2.3 Stack Using Linked List

1. Creation :

```
Void create(struct stack *top, int x)
{
Struct stack *new;
top=malloc(sizeof(struct stack));
top-->data=x;
top-->next=NULL;
top=new;
scanf("%d",&x);
do
{
new=malloc(sizeof(struct stack));
new-->data=x;
new-->next=top;
top=new;
scanf("%d", &x);
} while(x!=0);
}
```

2. Push( ) :

```
Void push(struct stack *top, int num)
{
Struct stack *new;
If(top==NULL) {
Printf("Stack is initially empty");
top=malloc(sizeof(struct   stack));
top-->data=num;
top-->next=NULL;
}
Else
{
new=malloc(sizeof(struct stack));
new-->data=num;
new-->next=top;
top=new;
}
}
```



3. Displaytop( ) / peek( ) :

```
Void peek( struct stack *top)
{
printf("%d",top-->data);
}
```

4. Displayall( ) :

```
Void displayall(struct stack *top)
{
Struct stack *temp;
temp=top;
while(temp != NULL)
{
Printf("%d",temp-->data);
temp=temp-->next;
}
}
```

5. Pop( ) :

```
Void pop(struct stack *top)
{
Struct stack *temp;
temp=top;
top=top-->next;
free(temp);
}
```

Advantages of stack:

- very simple data type.
- very fast.
- space efficient.
- direct access to last (first)element added.

Disadvantages of stack:

- set of operations is very restricted(no access to elements other than last(first), no searching, no iterating)

2.1.3 APPLICATION OF STACK

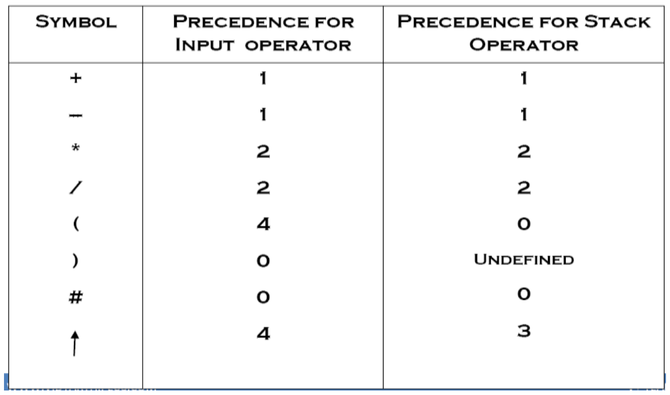
1. Expression evaluation
2. Backtracking (game playing, finding paths, exhaustive searching)
3. Memory management, run-time environment for nested language feature

2.2 THE EVALUATION OF ARITHMETIC EXPRESSION

There are three forms to evaluate arithmetic expressions:

1. Infix Operand Operator Operand.
2. Postfix Operand Operand Operator.
3. Prefix Operator Operand Operand.

Priority Table



Symbol	Precedence for Input operator	Precedence for Stack Operator
+	1	1
–	1	1
*	2	2
/	2	2
(	4	0
)	0	Undefined
#	0	0
↑	4	3

Table: 2.1- Priority Table

EVALUATION OF POSTFIX:

1. The first element you pop off of the stack in an operation should be evaluated on the right hand side of the operator.
2. For multiplication and addition order does not matter. But for subtraction and division, your answer will be incorrect if you change your operands alone.
3. If the character is an operand, push it on to the stack. If the character is an operator, pop first two values from the stack and apply the operator to them and push the result on to the stack again.

For instance, the postfix expression
6 5 2 3 + 8 * + 3 + *
is evaluated as follows: The first four symbols are placed on the stack. The resulting stack is

- Next a '+' is read, so 3 and 2 are popped from the stack and their sum, 5, is pushed.
- Next 8 is pushed.
- Now a '*' is seen, so 8 and 5 are popped as 8 * 5 = 40 is pushed.
- Next a '+' is seen, so 40 and 5 are popped and 40 + 5 = 45 is pushed.
- Now, 3 is pushed.
- Next '+' pops 3 and 45 and pushes 45 + 3 = 48.

Finally, a '*' is seen and 48 and 6 are popped, the result 6 * 48 = 288 is pushed.

BALANCING SYMBOLS

Compilers check your programs for syntax errors, but frequently a lack of one symbol (such as a missing brace or comment starter) will cause the compiler to spill out a hundred lines of diagnostics without identifying the real error.

A useful tool in this situation is a program that checks whether everything is balanced. Thus, every right brace, bracket, and parenthesis must correspond to their left counterparts. The sequence [()] is legal, but [(]) is wrong.

The simple algorithm uses a stack and is as follows:

1. Make an empty Stack.
2. Read the character until end of file.
3. If the character is an opening symbol, push it on to the Stack.
4. If the character is a closing symbol, then if the stack is empty , report an error, otherwise pop the stack.
5. If the Symbol popped is not the corresponding opening symbol, then report an error.
6. At the end of file, if the stack is not empty, report an error.

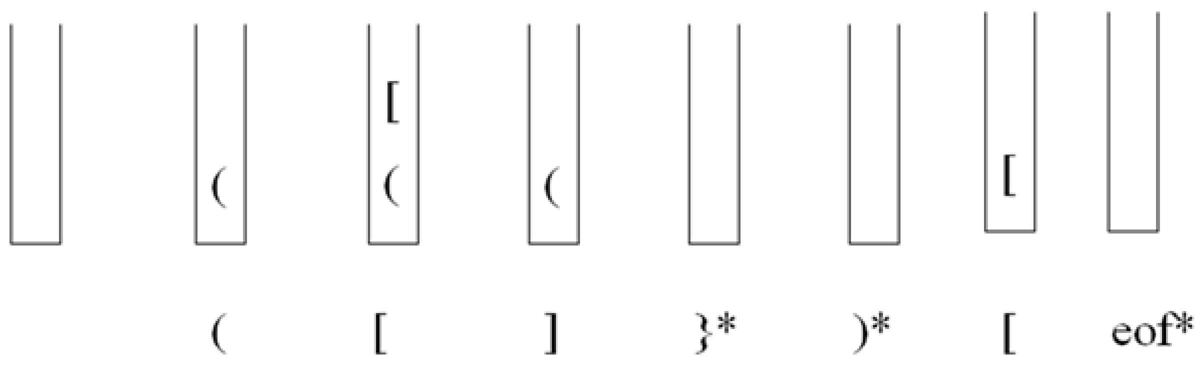


Fig: 2.4 Balancing Symbols

It is clearly linear and actually makes only one pass through the input. It is thus on-line and quite fast. Extra work can be done to attempt to decide what to do when an error is reported--such as identifying the likely cause.

2.3 CONVERSION INFIX TO POSTFIX

Algorithm for Infix to Postfix :

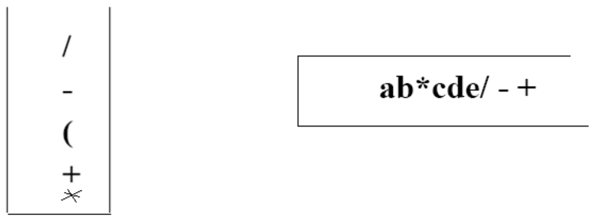
Rule 1: Read the infix expression, one character at a time,until it encounters the demitor.
Rule 2: If the character is an operand, place it onto the output. If the character is an operator, then follow the conditions given below.

- If the stack operator has the highest or equal priority than the input operator, then pop that operator from the stack, and place it onto the operator into the stack.
- If the stack operator has the lowest Priority than the input operator, then simply push the input operator into the stack.

Rule 3: If the character is a left parenthesis, push it on to the stack. If the character is a right parenthesis, pop all the operators from the stack, till it encounters left parenthesis and discard both the parenthesis for the output.

1. **Evaluate the expression**

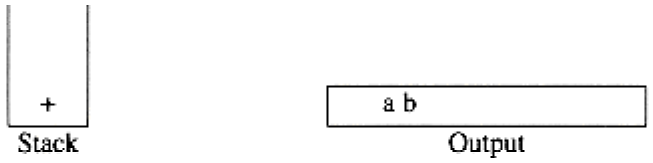
a*b+(c – d / e)



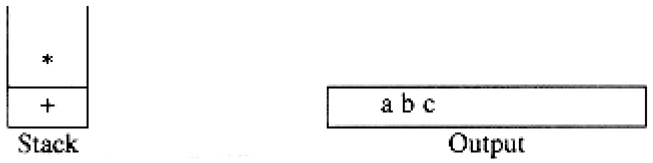
2. convert the infix expression

a + b * c + (d * e + f) * g

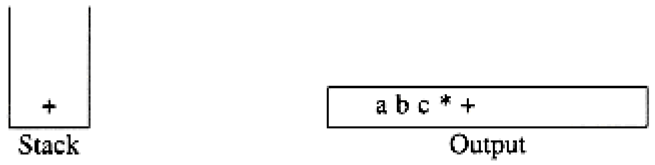
First, the symbol a is read, so it is passed through to the output. Then '+' is read and pushed onto the stack. Next b is read and passed through to the output. The state of affairs at this juncture is as follows:



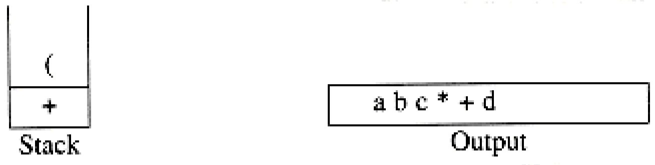
Next a '*' is read. The top entry on the operator stack has lower precedence than '*', so nothing is output and '*' is put on the stack. Next, c is read and output. Thus far, we have



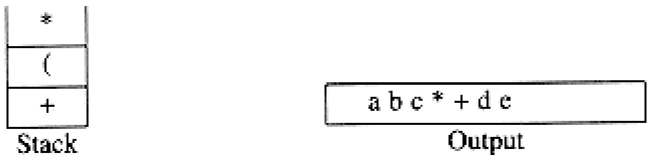
The next symbol is a '+'. Checking the stack, we find that we will pop a '*' and place it on the output, pop the other '+', which is not of lower but equal priority, on the stack, and then push the '+'



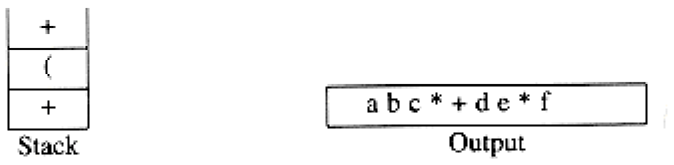
The next symbol read is an '(', which, being of highest precedence, is placed on the stack. Then d is read and output.



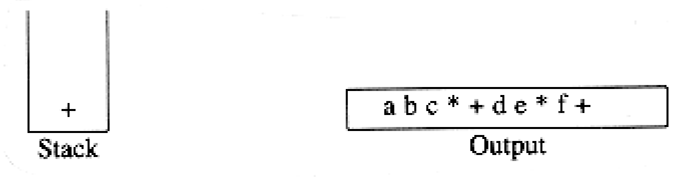
We continue by reading a '*'. Since open parentheses do not get removed except when a closed parenthesis is being processed, there is no output. Next, e is read and output.



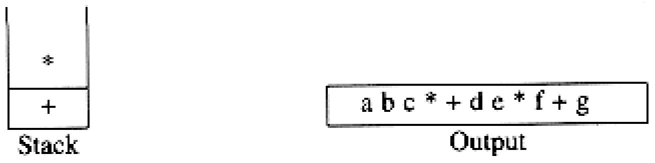
The next symbol read is a '+'. We pop and output '*' and then push '+'. Then we read and output



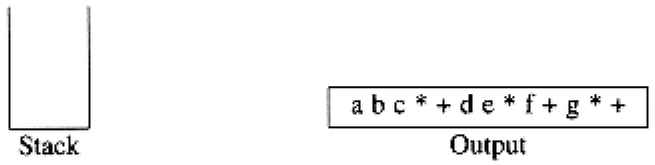
Now we read a ')', so the stack is emptied back to the '('. We output a '+'.



We read a '*' next; it is pushed onto the stack. Then g is read and output.



The input is now empty, so we pop and output symbols from the stack until it is empty.



```
#include<stdio.h>
#include<alloc.h>
char inf[40],post[40];
int top=0,st[20];
void postfix();
void push(int);
char pop();
void main()
{
printf("\n\nEnter the infix expression:");
 scanf("%s",inf);
 postfix();
}
void postfix()
{
int i,j=0;
for(i=0;inf[i]!='\0';i++)
{
switch(inf[i])
{
case  '+':  while(st[top]>=1)
                post[j++]=pop();
                push(1);
                break;
case '-': while(st[top]>=1)
```

```
              post[j++]=pop();
              push(2);
              break;
  case  '*':while(st[top]>=3)
              post[j++]=pop();
              push(3);
              break;
  case  '/': while(st[top]>=3)
              post[j++]=pop();
              push(4);
              break;
  case  '^': while(st[top]>=4)
              post[j++]=pop();
              push(5);
              break;
  case '(': push(0);
              break;
  case ')': while(st[top]!=0)
              post[j++]=pop();
              top--;
              break;
  default:post[j++]=inf[i];
 }
} while(top>0)
post[j++]=pop();
printf("\nThe postfix expression is
%s",post);
}
```

```
void push(int ele)
{ top++;
st[top]=ele;
}
char pop()
{
 int e1; char
 e;
 e1=st[top];
 top--;
 switch(e1)
 {
  case 1:e='+';
          break;
   case 2:e='-';
          break;
  case 3:e='*';
          break;
  case 4:e='/';
          break;
   case 5:e='^';
          break;
  }
  return (e);
  }
```

2.4 CONCEPT OF QUEUE – QUEUE ADT

The queue is one of the more simple abstract data types, closely related to the stack. In many ways the queues is a “backward stack”; where a stack is a first in last out data storage medium, a queue is a ***first in first out*** structure.

Queue is an ordered collection of elements in which insertion is done at the rear end, and deletion is done at the front end. Technical name for insertion is Enqueue( ) and technical name for deletion is Dequeue( ).

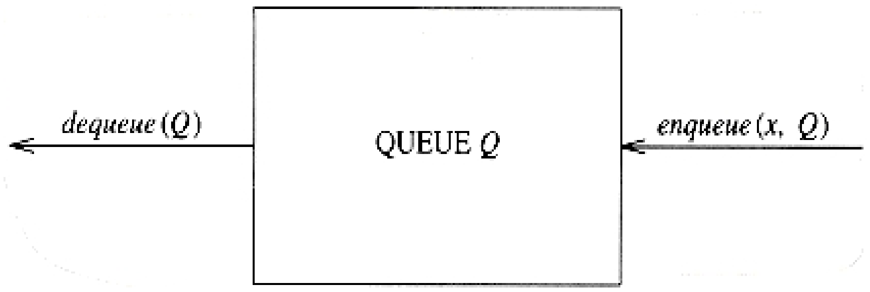


Fig: 2.5 Concept of Queue

Types Of Queue:

- **Linear queues**
- **Circular queues**
- **Priority queue**
- **Deque**

LINEAR QUEUE

Linear Queue can be implemented using

- array
- linked list.

Queue using array

For each queue data structure, we keep an array, QUEUE[], and the positions q_front and q_rear, which represent the ends of the queue. We also keep track of the number of elements that are actually in the queue, q_size.

To enqueue an element x, we increment q_size and q_rear, then set QUEUE[q_rear] = x. To dequeue an element, we set the return value to QUEUE[q_front], decrement q_size, and then increment q_front.

2.4.1 QUEUE OPERATIONS

The possible ADT's are,

1) Is Empty
2) Is Full
3) Display rear
4) Display Front
5) Enqueue
6) Dequeue
7) Count

Representation of Queue using array:

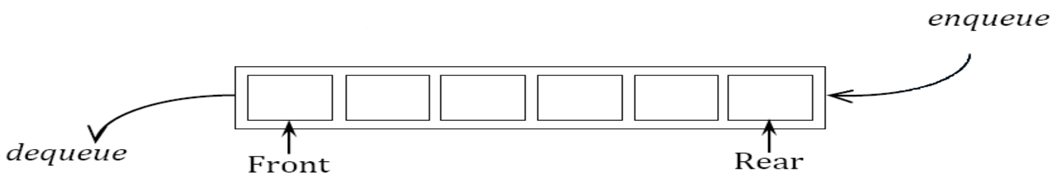
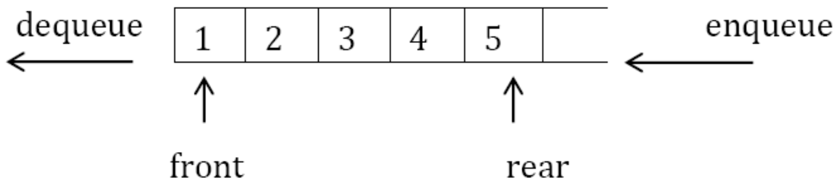


Fig: 2.6 Representation of Queue

Example:



1.Is Empty :

```
Void isempty(int queue[ ] ,int front, int rear)
{
if((front== -1) && (rear== -1))
printf("Queue is empty");
else
printf("Queue is not empty");
}
```

2. Is Full :

```
Void isfull(int queue[ ], int rear)
{
If(rear == size-1)
Printf("Queue is full");
Else
Printf("Queue is not full");
}
```

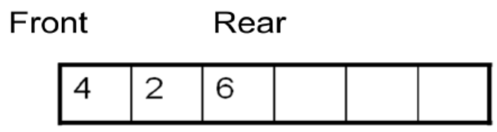
3. Creation :

```
Void create(int queue[ ],int front, int rear,int n)
{
```

```
int i; for(i=0;i<n;i++)
scanf("%d", &queue[i]);
front=0;
rear=n-1;
}
```

4. Display Rear :

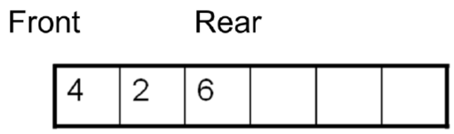
```
Void displayrear(int queue[ ],int rear)
{
Printf("%d",queue[rear]);
}
```



Rear is 6

5. Display Front :

```
Void dispfront(int queue[ ],int front)
{
Printf("%d", &queue[front]);
}
```



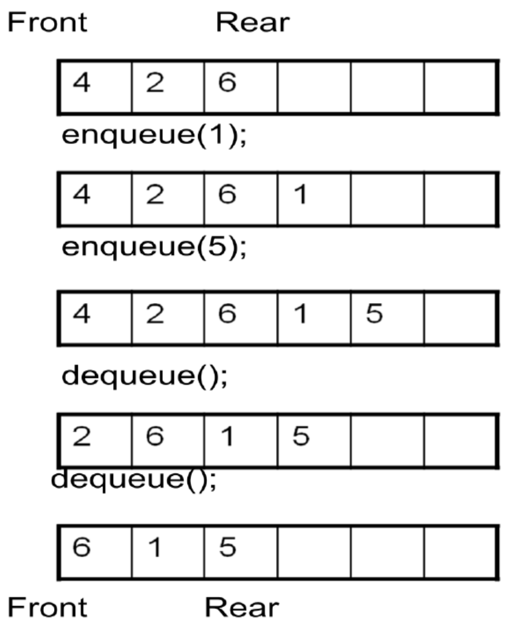
Front is 4

6. Insertion :

```
Void enqueue(int queue[ ],int rear,int num)
{
if(rear==size-1)
printf("Overflow");
else
queue[++rear]=num;
}
```

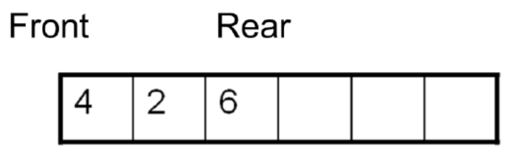
7. Deletion | Dequeue :

```
Void dequeue(int queue[ ],int front)
{
if(front== -1)
printf("Underflow");
else
{
printf("%d",queue[front]);
++front;
}
}
```



8. Display :

```
Void display(int queue[ ],int front, int rear)
{
int i;
for(i=front;i<=rear;i++)
printf("%d",queue[i]);
}
```



OUTPUT : 4 2 6

2.4.2 LINKED IMPLEMENTATION OF QUEUE

The first decision in planning the linked-list implementation of the Queue class is which end of the list will correspond to the front of the queue. Recall that items need to be added to the rear of the queue, and removed from the front of the queue. Therefore, we should make our choice based on whether it is easier to add/remove a node from the front/end of a linked list.

If we keep pointers to both the first and last nodes of the list, we can add a node at either end in constant time. However, while we can remove the first node in the list in constant time, removing the last node requires first locating the **previous** node, which takes time proportional to the length of the list. Therefore, we should choose to make the end of the list be the rear of the queue, and the front of the list be the front of the queue.

Representation of queue

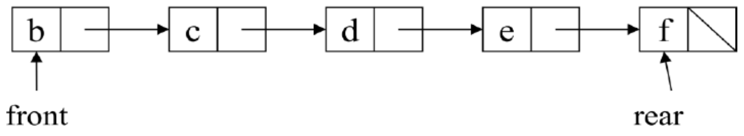
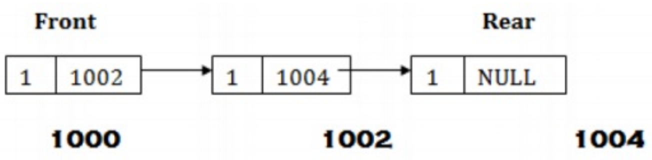


Fig: 2.7 Representation of Linked List in Queue

Struct queue

```
{
int data;
struct *next;
} *front, *rear;
front=NULL;
rear=NULL;
```



1. Is Empty :

```
Void isempty(struct queue *front, struct queue *rear)
{
If((front==NULL) && (rear==NULL))
Printf("Queue is empty");
}
```

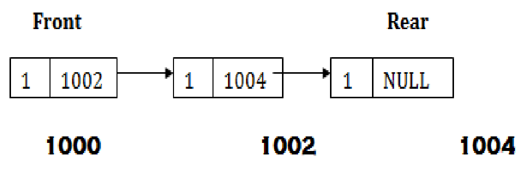
2. Creation :

```
Void create(struct queue *front, struct queue *rear, int n)
{
front=malloc(sizeof(struct queue));
front-->data=n;
front-->next=NULL;
rear=front;
scanf("%d",&n);
while(n!=0)
{
rear-->next=malloc(sizeof(struct queue));
rear=rear-->next;
scanf("%d", &n);
}
rear-->next=NULL;
}
```

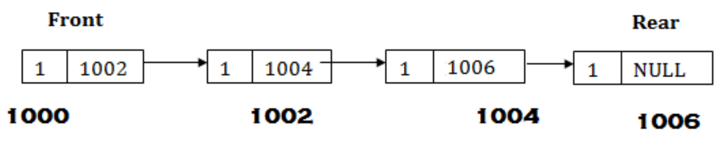
3. Enqueue :

```
Void enqueue(struct queue *rear, int num)
{
rear-->next=malloc(sizeof(struct queue));
rear=rear-->next;
rear-->data=num;
rear-->next=NULL;
}
```

Before Insertion



After Insertion

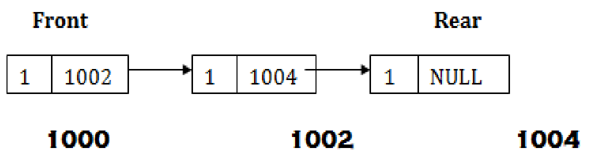


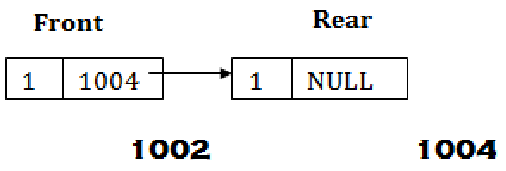
4.Dequeue :

```
Void dequeue(struct queue *front)
{
```

```
Struct queue *t;
t=front;
front=front-->next;
free(t);
}
```

Before Deletion: **After Deletion:**





5. Display Rear :

```
Void displayrear(struct queue *rear)
{
printf("%d",rear-->data);
}
```

2.5 CIRCULAR QUEUE

In Circular Queue, the insertion of a new element is performed at the very first location of the queue if the last location of the queue is full, in which the first element comes just after the last element.

To view the array that holds the queue as a circle rather than as a straight line. That is, we imagine the first element of the array as immediately following its last element.

This implies that even if the last element is occupied, a new value can be inserted behind it in the first element of the array as long as that first element is empty. This concept of queue is known as ***Circular Queue***.

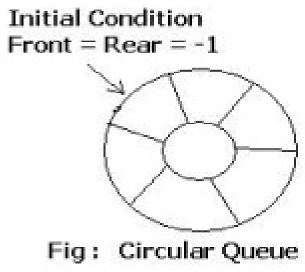


Fig: 2.8 Representation of Circular Queue

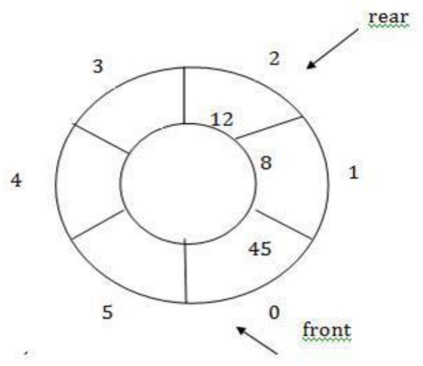
CIRCULAR QUEUE USING ARRAYS

ADT's of circular queue using array :

1) Is empty
2) Is full
3) Enqueue
4) Dequeue
5) Display Rear
6) Display Front
7) Display Queue
8) Creation

1. IsEmpty :

```
Void isempty(int queue[ ], int front, int rear)
{
If(front==rear== -1)
Printf("Queue is empty");
Else
Printf("Queue is not empty");
}
```



2. Is Full :

```
Void isfull(int queue[ ],int front, int rear)
{
If((rear+1) % qsize==front)
Printf("Queue is Full");
Else
Printf("queue is not full");
}
```

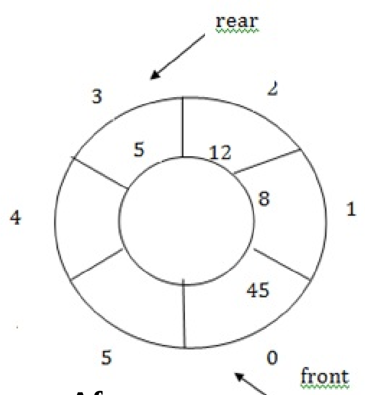
3. Creation :

```
Void create(int queue[ ],int front,int rear,int n)
{
int i; for(i=0;i<n;i++)
scanf("%d",&queue[i]);
front=0;
rear=n-1;
}
```

4. Enqueue :

```
Void enqueue(int queue[ ],int front,int rear, int n)
{
```

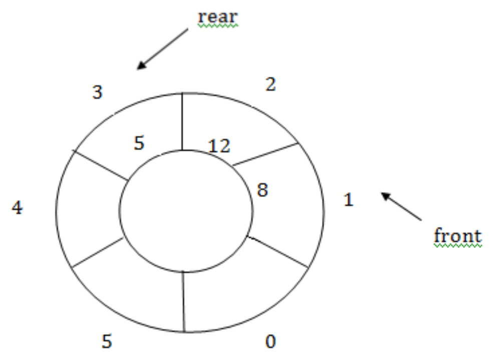
```
if((rear+1) % qsize==front)
printf("Overflow");
else
{
rear=(rear+1)%qsize;
queue[rear]=n;
}
}
```



After enqueue

5. Dequeue :

```
Void dequeue(int queue[ ],int front, int rear)
{
int x; if(front==rear==
-1) printf("Queue
empty"); else
if(front==rear)
{ x=queue[front];
front=(front+1) % qsize;
}
Printf("%d",x);
}
```



6. Display rear :

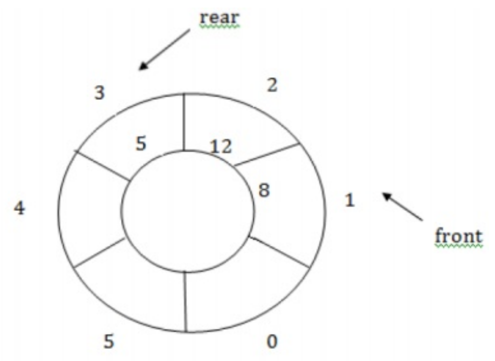
```
Void disprear(int queue[ ],int rear)
{
Printf("%d",queue[rear]);
}
```

7.Display Front :

```
Void displayfront(int queue[ ], int front)
{
Printf("%d", queue[front]);
}
```

8. Display Queue :

```
Void display(int queue[ ], int front, int rear)
{
If(rear<front)
{
```



Front =8 Rear = 3
Display all Output: 8 12 5

```
For(i=front;i<qsize;i++)
Printf("%d",queue[i]);
For(i=0;i<=rear;i++)
Printf("%d",queue[i]);
}
Else
{
For(i=front;i<=rear;i++)
Printf("%d",queue[i]);
}
}
```

CIRCULAR QUEUE USING LINKED LIST

```
struct queue
{
Int data;
Struct queue *next;
} *rear=NULL;
```

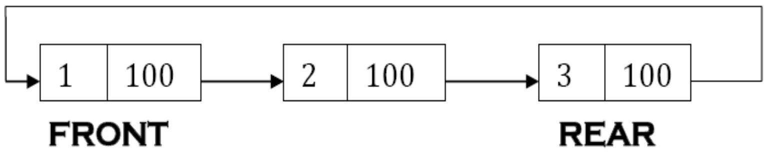


Fig: 2.9 Representation of Circular Queue using Linked List

1.IsEmpty:

```
Void isempty(struct queue *front, struct queue *rear)
{
If(front==rear==NULL)
Printf("queue is empty");
Else
Printf("queue is not empty");
}
```

2.Enqueue

```
void enqueue(struct queue *front, struct *rear, int num)
{
If(front!=NULL && rear!=NULL)
{ Struct queue *new;
new=malloc(sizeof(struct queue));
new-->data=num;
rear-->next=new;
rear=new;
new-->next=front;
}
Else if(front==rear==NULL)
{
front==malloc(sizeof(struct queue));
front-->data=num;
front-->next=front;
rear=front;
} }
```

3.Dequeue

```
Void dequeue(struct queue *front, struct queue *rear)
{
Struct queue *temp;
temp=front;
front=front-->next;
rear-->next=front;
free(temp);
}
```

4.Display Rear

```
Void displayrear(struct queue *rear)
{
Printf("%d",rear-->data);
}
```

5.Display Front

```
Void displayfront(struct queue *front)
{
Printf("%d",front-->data);
}
```

6.Display

```
Void display(struct queue *front, struct queue *rear)
{
Struct queue *temp;
temp=front;
while(temp-->next!=front)
{
Printf("%d",temp-->data);
temp=temp-->next;
}
Printf("%d",temp-->data);
}
```

2.6 PRIORITY QUEUE

Priority Queue is more specialized data structure than Queue. Like ordinary queue, priority queue has same method but with a major difference. In Priority queue items are ordered by key value so that item with the lowest value of key is at front and item with the highest value of key is at rear or vice versa. So we're assigned priority to item based on its key value. Lower the value, higher the priority. Following are the principal methods of a Priority Queue.

Basic Operations

- Insert / Enqueue − add an item to the rear of the queue.
- Remove / Dequeue − remove an item from the front of the queue.

Priority Queue Representation

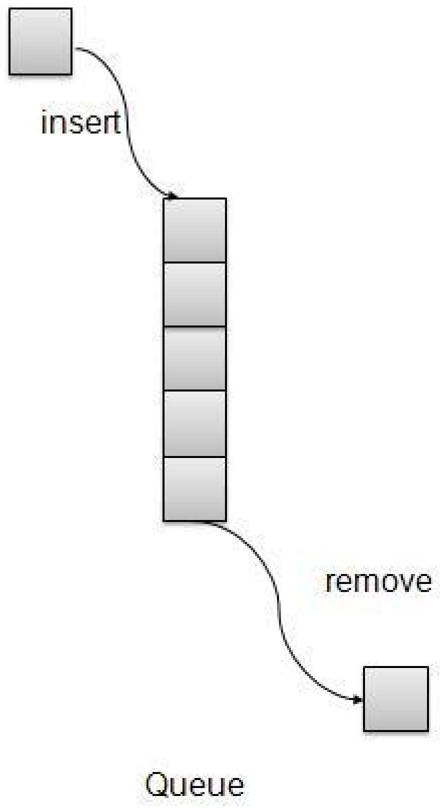


Fig: 2.10 Representation of Priority Queue

We're going to implement Queue using array in this article. There is few more operations supported by queue which are following.

- Peek − get the element at front of the queue.
- isFull − check if queue is full.
- isEmpty − check if queue is empty.

Insert / Enqueue Operation

Whenever an element is inserted into queue, priority queue inserts the item according to its order. Here we're assuming that data with high value has low priority.

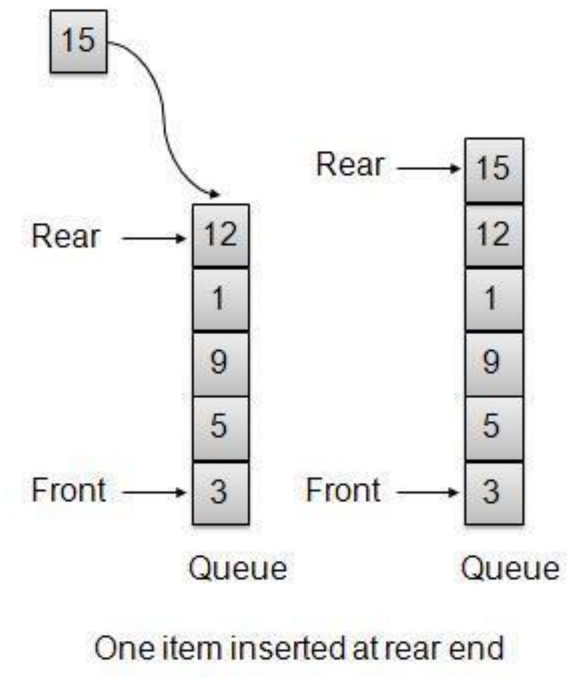


Fig: 2.11 Representation of Priority Queue Operation-Insertion

```
void insert(int data)
{
  int i = 0;
  if(!isFull()){
    // if queue is empty, insert the data
         if(itemCount == 0){
      intArray[itemCount++] = data;
    }else{
      // start from the right end of the queue
      for(i = itemCount - 1; i >= 0; i-- ){
        // if data is larger, shift existing item to right end
        if(data > intArray[i]){
          intArray[i+1] = intArray[i];
        }else{
          break;
        }
      }
      // insert the data
      intArray[i+1] = data;
      itemCount++;
    }
  }
}
```

Remove / Dequeue Operation

Whenever an element is to be removed from queue, queue get the element using item count. Once element is removed. Item count is reduced by one.

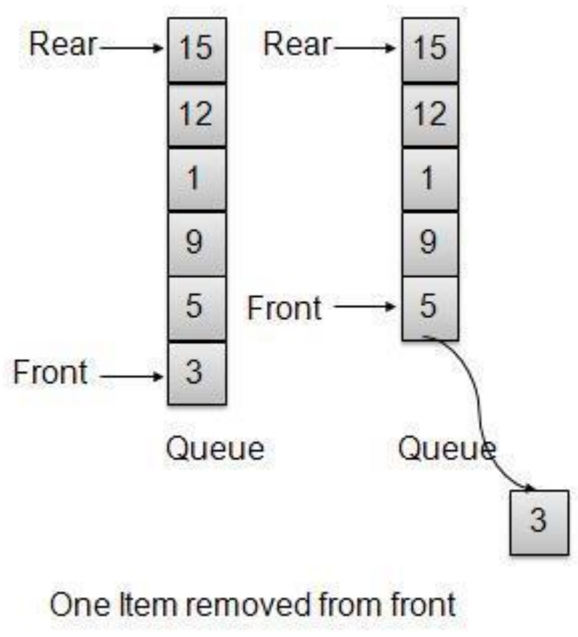


Fig: 2.12 Representation of Priority Queue Operation-Deletion

```
int removeData()

{

   return intArray[--itemCount];

}
```

2.7 DOUBLE ENDED QUEUE - DeQueue

A **dequeue** (short for *double-ended queue*) is an abstract data structure for which elements can be added to or removed from the front or back (both end). This differs from a normal queue, where elements can only be added to one end and removed from the other. Both queues and stacks can be considered specializations of deques, and can be implemented using deques.

It can be implemented using

- Array
- Linked lists

Two types of Dequeue are

1. Input Restricted Dequeue
2. Ouput Restricted Dequeue

Dequeue using Array

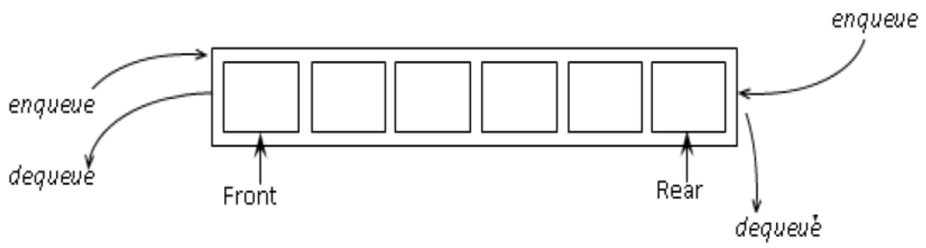


Fig: 2.13 Representation of DeQueue

1. Input Restricted Deque

Where the input (insertion) is restricted to the rear end and the deletions has the options either end

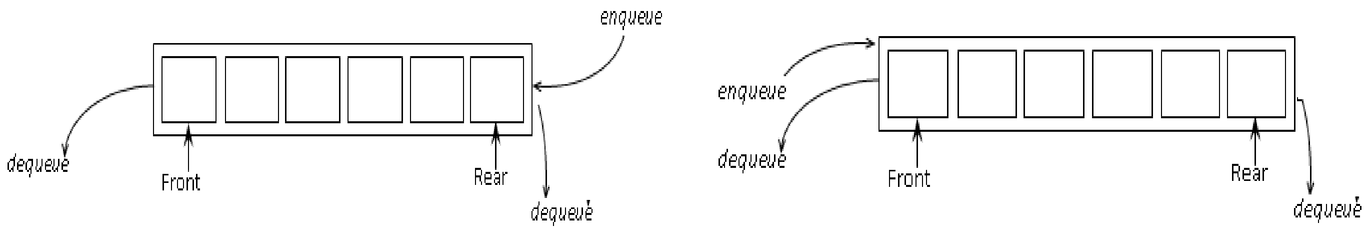


Fig: 2.14 Representation of DeQueue – Input Restricted Dequeue

```
Void enqueue-at-last(int queue[ ],int rear,int num)
{
if(rear==size-1)
printf("Overflow");
else
rear=rear+1;
queue[++rear]=num;
}
Void dequeue-at-front(int queue[ ], int front)
{
if(front== -1)
printf("Underflow");
else
```

```
{
printf("%d",queue[front]);
++front;
}
Void dequeue-at-last(int queue[ ], int front)
{
if(front== -1)
printf("Underflow");
else
{
Int x;
printf("%d",queue[front]);
rear=rear-1;
}
```

2. Ouput Restricted Deque

Where the output (deletion) is restricted to the front end and the insertions has the option either end.

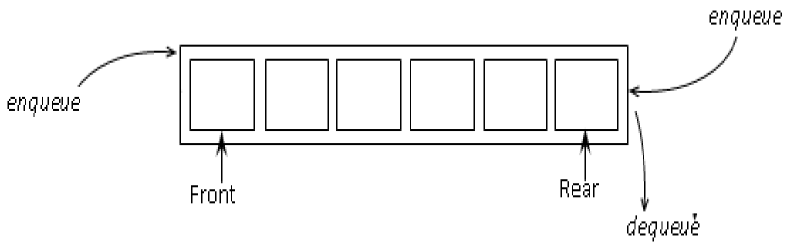


Fig: 2.15 Representation of DeQueue – Output Restricted Dequeue

```
Void enqueue-at-last(int queue[ ],int rear,int num)
{
if(rear==size-1)
printf("Overflow");
else
rear=rear+1;
queue[++rear]=num;
}
Void enqueue-at-front(int queue[ ],int front,int num)
{
front=front-1;
queue[front]=num;
}
```

```
Void dequeue-at-last(int queue[ ], int front)
{
if(front== -1)
printf("Underflow");
else
{
Int x;
printf("%d",queue[front]);
rear=rear-1;
}
```

The Operations in Double Ended Queue using Linked List

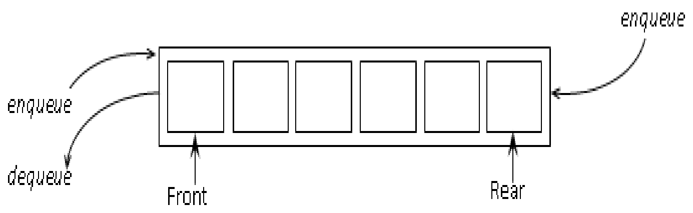


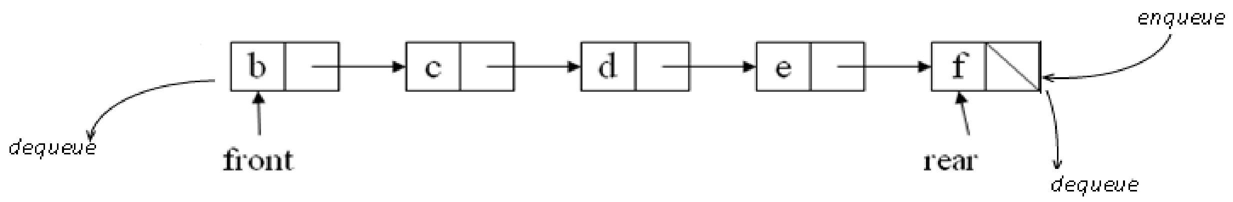
Fig: 2.16 Representation of De-Queue using Linked List

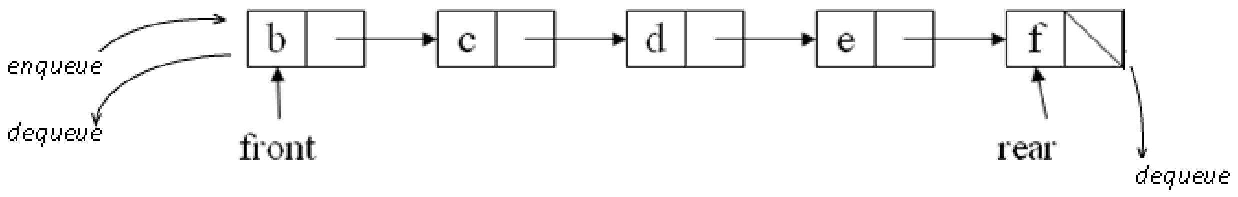
Two types of De-queue are

1. Input Restricted De-queue
2. Ouput Restricted De-queue.

1. Input Restricted De-queue

Where the input (insertion) is restricted to the rear end and the deletions has the options either end





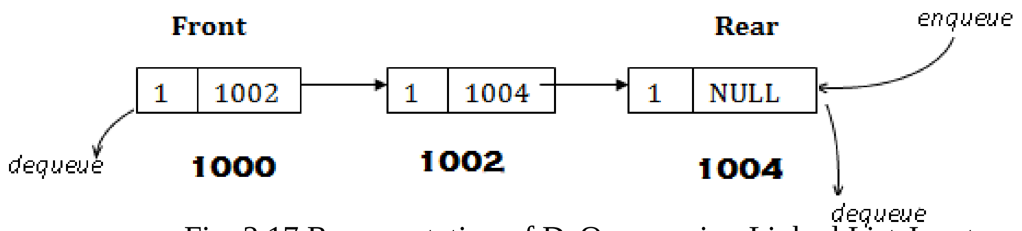


Fig: 2.17 Representation of DeQueue using Linked List-Input

```
Void enqueue-at-last(struct queue *rear, int num)
{
Struct node *temp;
temp=malloc(sizeof(struct queue));
rear=rear-->next;
rear-->data=num;
rear-->next=NULL;
}

Void dequeue-at-front(struct queue *front)
{
Struct queue *t;
t=front;
front=front-->next;
free(t);
}

Void dequeue-at-last(struct queue *front)
{
Struct node *temp,*temp1;
temp=front;
While(temp□next!=null)
{
temp1=temp;
temp=temp□next;
rear=temp1;
free(t);
}
```

2. Output Restricted De-queue

Where the output (deletion) is restricted to the front end and the insertions has the option either end.

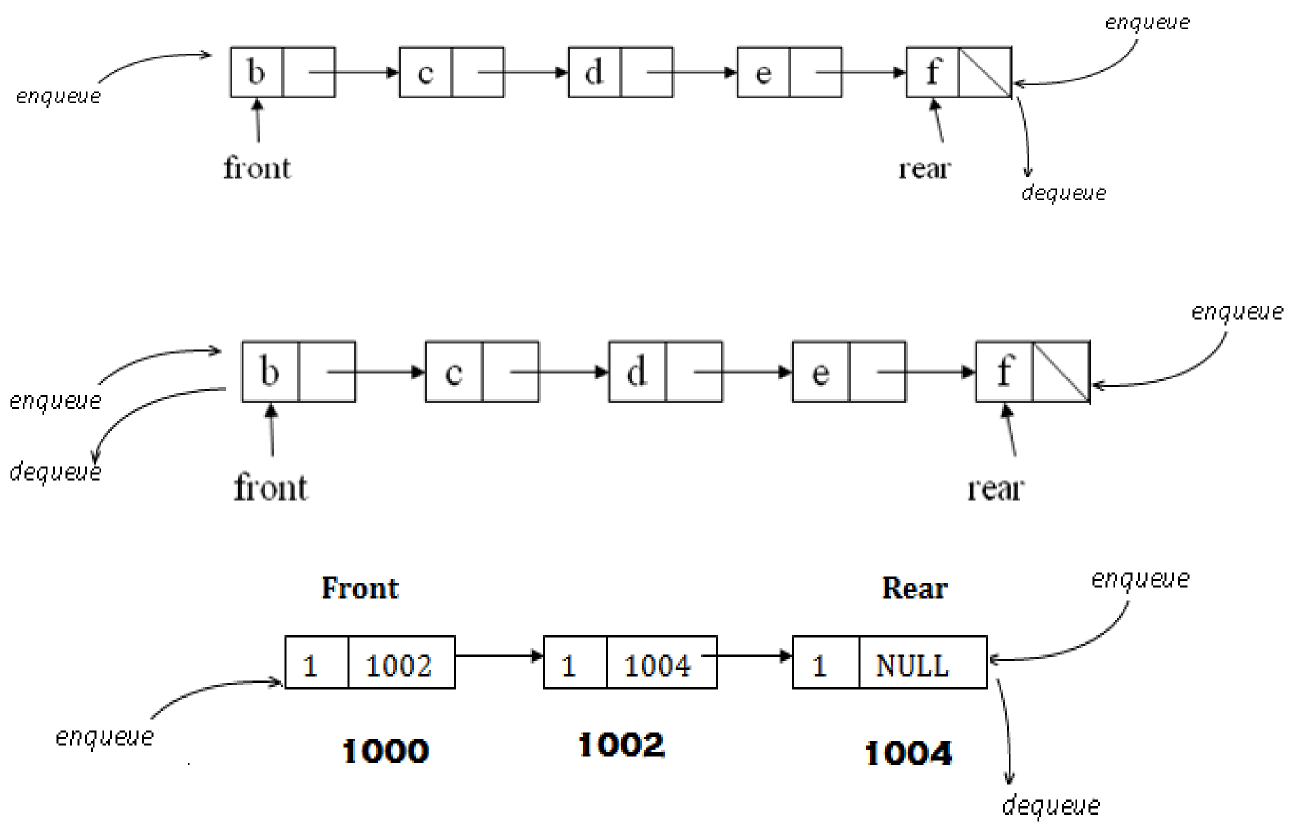


Fig: 2.18 Representation of De-Queue using Linked List-Output

```
Void enqueue-at-last(struct queue *rear, int num)
{
Struct node *temp;
temp=malloc(sizeof(struct queue));
rear=rear-->next;
rear-->data=num;
rear-->next=NULL;
}
```

```
Void enqueue-at-first(struct queue *rear,struct queue *front, int num)
{
Struct node *temp;
temp=malloc(sizeof(struct queue));
temp-->data=num;
temp-->next=front;
front=temp;
}
```

```
Void dequeue-at-last(struct queue *front)
{
Struct node *temp,*temp1;
temp=front;
While(temp next!=null)
{
temp1=temp;
temp=temp next;
rear=temp1;
free(t);
}
```

2.8 APPLICATION OF QUEUE

JOB SCHEDULING

In the operating system various programs are getting executed.

We will call these programs as jobs. In this process, some programs are in executing state. The state of this program is called as running state.

Some programs which are not executing but they are in position to get executed at any time such programs are in the ready state. And there are certain programs which are neither in running state nor in ready state.

Such programs are in a state called as blocking state.The operating system maintains a queue of all such running state, ready state, blocked state programs. Thus use of queue help the operating system to schedule the jobs.

The jobs which are in running state are removed after complete execution of each job, then the jobs which are in ready state change their state from ready to running and get entered in the queue for running state. Similarly the jobs which are in blocked state can change their state from blocked to ready state.

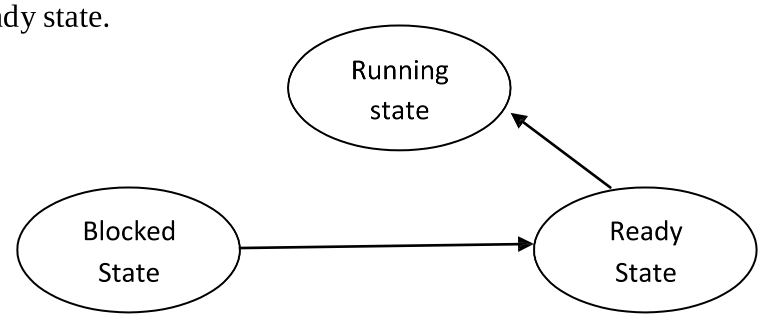


Fig: 2.19 Job Scheduling

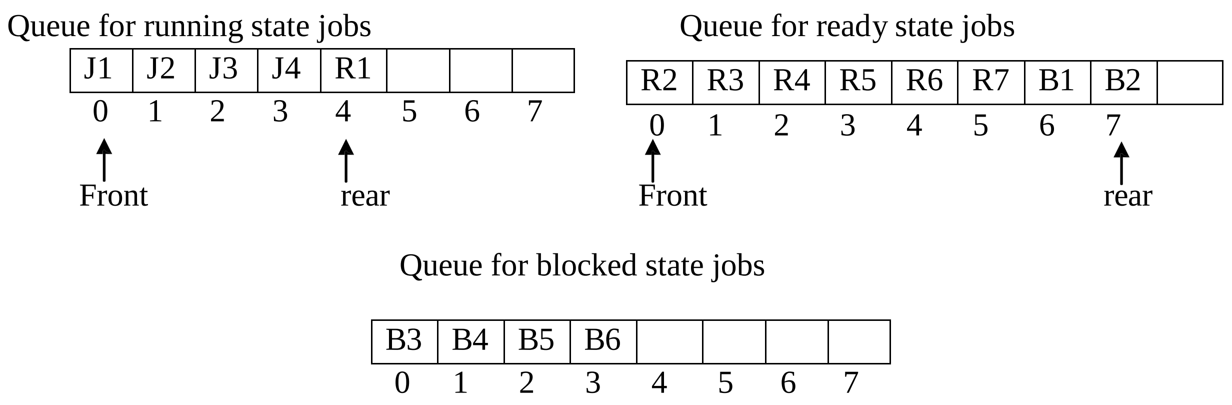


Fig: 2.20 Jobs are changing their state of execution

These jobs then can be entered in the queue for ready state jobs. Thus every jobs change its state and finally get executed. The queues are effectively used in the operating system for scheduling the jobs.

PART-A

STACK ADT

1. Define stack [L1]
2. What are the operations in stack ADT? [L1]
3. What are the applications of stack?**[N/D 2018][M/J 2014][M/J 2013]** [L1]
4. What is stack overflow? [L1]
5. What is stack pop? [L1]
6. What is stack push? [L1]
7. Define stack full and stack empty. [L1]
8. Differentiate between linked list and stack. [L1]
9. What is the data structure used to perform recursion? How? [L2]

EVALUATING ARITHMETIC EXPRESSIONS- OTHER APPLICATIONS

10. Define expression. Give its types. [L1]
11. List the types of expression representation. [L1]
12. What are the postfix and prefix forms of the expression? A+B*(C-D)/(P-R) **[N/D 14]** L3]

QUEUE ADT - APPLICATIONS OF QUEUES

13. Define queue.**[A/M 2015]** [L1]
14. What are the operations on a queue? [L1]

15. What are the types of queues? [L1]
16. List the applications of queue. [L1]
17. Differentiate between Stack and queue. [L1]

CIRCULAR QUEUE IMPLEMENTATION

18. What are circular queues? [L1]
19. How circular queue is considered superior to linear queue? [L3]
20. Differentiate between linear queue and circular queue. [L1]
21. List the applications of circular queue. [L1]
22. Give the advantage and disadvantages of circular queue. [L1]

DOUBLE ENDED QUEUES

23. What is Double ended queue? [L1]
24. What are the operations that can be performed on Double ended queue? [L3]
25. Differentiate between queue and double ended queue. [L1]
26. List the applications of double ended queue. [L1]
27. Give the advantages and disadvantages de-queue. [L1]
28. What is priority queue? **[N/D 2018]** [L1]

PART B

STACK ADT

1. Explain the operations and the implementation of Stack ADT using Array.**[A/M 2015]** [L1]
2. Explain in detail about linked list implementations of stack ADT. **[Pg.No:41**] [L1]

EVALUATING ARITHMETIC EXPRESSIONS- OTHER APPLICATIONS

3. Explain in detail about the evaluation of arithmetic expression. Write an algorithm to convert infix to postfix. **[N/D 2018] [M/J 2013]** [L2]
4. Explain the various application of stack? **[Pg.No:45]** [L1]

QUEUE ADT

5. Explain the operations and the implementation of Queue ADT using Array**[N/D 2012**] [L1]
6. What is queue ADT? Give linked implementation of queue.**[Pg.No:48 & 52**] [L1]

CIRCULAR QUEUE IMPLEMENTATION

7. Explain about circular queue with example. **[N/D 2018]** [L1]

DOUBLE ENDED QUEUES - APPLICATIONS OF QUEUES

8. Write an algorithm to perform the four operations in double ended queue that is implemented as array. **[Pg.No:61]** [L2]
9. Write an algorithm to perform the operations in double ended queue that is implemented as linked list. **[Pg.No:64]** [L2]
10. Write about any one application of queue. **[Pg.No:67]** [L1]

PART -A

STACK ADT

1. **Define stack**

   A stack is a linear data structure (ADT) that stores in which items can be added and removed only at one ends. We can access only the item that is currently at the top. The stack is also called as LIFO i.e. Last In First Out data structure.

2. **What are the operations in stack ADT?**
   - push() – add an item to the top of the stack
   - pop() – remove the items at the top of the stack
   - pop() – returns the element at the top of the stack without removing it
   - isEmpty() – checks if the stack is empty.

3. **What are the applications of stack?**
   - Page-visited history in a Web browser
   - Undo sequence in a text editor
   - Function calls
   - Balancing parenthesis
   - Evaluating algebraic expression
4. **What is stack overflow?**

   The attempt to push an element in an already filled stack will result in stack overflow. This is an exception condition.

5. **What is stack pop?**

   The process of removing a element on the top of the stack is pop operation.

```
void pop ( )
{
```

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

6. **What is stack push?**

The process of inserting a new element on the top of the stack is push operation. he value pushed will be inserted at the position indicated by the top pointer. After the push operation, the top will point to the next slot in the array that is ready for the next push.

```
void push(int item)
{ st.top++;
st.s[st.top]=item;
}
```

7. **Define stack full and stack empty.**

Stack full

Stackfull ( ) – This condition indicates whether the stack is full or not. if the stack is full then we cannot insert the elements in the stack. Before performing push we must check stackfull ( ) condition.

```
int stackfull ( )
{
if (st.top?=size-1)
return 1;
else
return 0;
}
```

Thus stfull is a Boolean function. If stack is full it returns 1 otherwise it returns 0.

Stack empty

This condition indicates whether the stack is empty or not. if the stack is empty then we cannot pop or remove an y element from the stack.
Before popping the elements from the stack we should check stackempty() condition.

```
int stackempty ( )
{
if (st.top==-1)
return 1;
else
return 0;
```

}

8. Differentiate between linked list and stack.

S.No	Linked List	Stack
1	A linked list is basically a series of Nodes. Each node contains two things: the data and the pointer to the next Node in the Linked List	A stack is an abstract data type where there are only two operations, push and pop.
2	Linked-List describes how data is stored	A stack deals with what data comes first or last.

9. What is the data structure used to perform recursion? How?

Stack: Because of its LIFO property it remembers its „caller" so knows whom to return when the function has to return. Recursion makes use of system stack for storing the return addresses of the function calls.

Every recursive function has its equivalent iterative (non-recursive) function. Even when such equivalent iterative procedures are written, explicit stack is to be used.

EVALUATING ARITHMETIC EXPRESSIONS- OTHER APPLICATIONS

10. Define expression. Give its types.

Expression is a string of operands and operators. Operands are some numeric values and operators are of two types: Unary operator and Binary operator.

11. List the types of expression representation.

- Infix expression (a+b) – operator in between the operands
- Postfix expression (+ab) – operator before the operands
- Prefix expression (ab+) operator after the operands

12. What are the postfix and prefix forms of the expression? A+B*(C-D)/(P-R)

- POSTFIF FORM : *ABCD-*PR-/+*
- PREFIX FORM : *+A/*B-CD-PR*

QUEUE ADT - APPLICATIONS OF QUEUES

13. Define queue.

A queue is an ADT in which items are added at one end (rear or back end) and removed from the other end (front end) in a first in, first out fashion (FIFO). The elements in a queue are accessed only at the front end. The insertion operation in a queue is called en-queue and deletion operation is called de-queue.

Representation of queue:

```
struct queue
{
int que[size], front, rear;
} Q;
```

14. What are the operations on a queue?

- enqueue() – Insert an element at the read end of the queue.
- dequeue**() –** Remove an element at the front end of the queue
- isFull() – Checks whether a queue is full.
- isEmpty() – Checks whether a queue is empty.

15. What are the types of queues?

- Simple queues.
- Circular queue.
- Double ended queue.
- Priority queue.

16. List the applications of queue.

- Job scheduling
- Categorizing data
- Simulation and modeling
- Time sharing systems
- Mathematics user queuing theory
- Computer networks
- In implanting depth first search and breadth first search.

17. Differentiate between Stack and queue.

S.No	**Stack**	**Queue**
1	Stack is a collection of object that works in LIFO (Last In First Out) mechanism	Queue is a collection of objects that works in FIFO (First In First Out)
2	In the stack the new items is inserted with push method and deleted with pop method	In the queue the new item is inserted with en-queue method and deleted with de-queue method.
3	Only one pointer (top) is needed to access the data.	Two pointer front and rear are needed to access the data.

4	The data is inserted and deleted at the top pointer	The data is inserted at the rear and deleted at the front pointer.

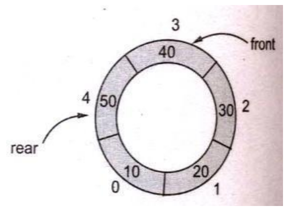
CIRCULAR QUEUE IMPLEMENTATION

18. What are circular queues?

Circular queue are special queue that are to store continuous data by starting from the beginning of the queue when the end is reached (i.e) the arrangement of element in a circular queue is in such a way that the first element comes after the last element. Circular queue have a fixed size.

19. How circular queue is considered superior to linear queue?

The circular queue overcomes the disadvantage of linear queue. In linear queue, the empty spaces caused due to dequeue operation at the front end are filled by shifting all the remaining element of the queue. This is an unnecessary overhead.



The formula which has to be applied for setting the front and rear pointers:

- rear = (rear + 1) % size
- front = (front + 1) % size

20. Differentiate between linear queue and circular queue.

S.No	Linear queue	Circular queue
1	A linear queue is like a straight line in which all elements or instructions stand one behind the other.	A circular queue is like a chain where head and tail are connected.
2	They have a definite starting and ending point.	They don’t have a definite starting and ending point.
3	They don’t use space efficiently.	They use space efficiently.
4	Their size is not fixed	Their size is fixed.

21. List the applications of circular queue

- CD/DVD burning
- Hardware print buffer
- Computer controlled traffic systems (light glow in circular fashion).

22. Give the advantage and disadvantages of circular queue.

Advantage:

- The memory of the deleted element can be reused.
- More number of insertions and deletions can be done.

Disadvantage:

- The queue is static.

DOUBLE ENDED QUEUES

23. What is Double ended queue?

A double-ended queue (deque pronounced as deck) is an abstract data structure that implements a queue for which elements can only be added to or removed from the front or rear end. It is also often called a head-tail linked list.

Types:

- *Input restricted queue:* In this type insertion is allowed at one end and deletion is allowed at both ends.
- *Output restricted queue:* In this type deletion is allowed at one end and insertion is allowed at both ends.

24. What are the operations that can be performed on Double ended queue?

- Insert an item from front end (En-queue Head)
- Insert an item from rear end (En-queue Tail
- Delete an item from front end (De-queue Head)
- Delete an item from rear end (De-queue Tail)

25. Differentiate between queue and double ended queue.

S.No	Queue	Double Ended Queue (Deque)
1	Queues support insertion at rear end and deletion at front end	In deques insertions and deletions can be made at both front and rear end.
2	Queues are classifies at circular queues and deques	Deques are classified as input restricted and output restricted deques.

26. List the applications of double ended queue.

- A-steal job scheduling algorithm (Multiprocessor scheduling)
  - The processor gets the first element from the double ended queue.
  - When one of the processor completes execution of its own thread it can steal a thread from another processor.

    - It gets the last element from the de-queue of another processor and executes it.
- Undo-Redo operations in software applications

27. Give the advantages and disadvantages dequeue.

Advantages of De-queue:

- Efficient in searching an element.
- Insertion and deletion are easy

Disadvantages of De-queue:

- More complicated programming

28. What is priority queue?

It is a type of queue in which each element is assigned certain priority such that the order of deletion of elements is decided by their associated priorities.