Unit-2

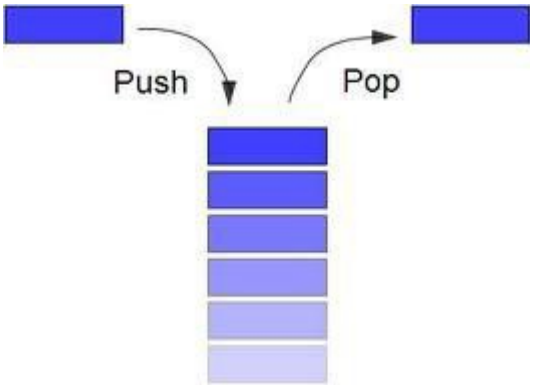
Stacks: Introduction-Definition-Representation of Stack-Operations on Stacks-Applications of Stacks.

Queues: Introduction, Definition- Representations of Queues- Various Queue Structures- Applications of Queues. **Tables**: Hash tables

Stacks:

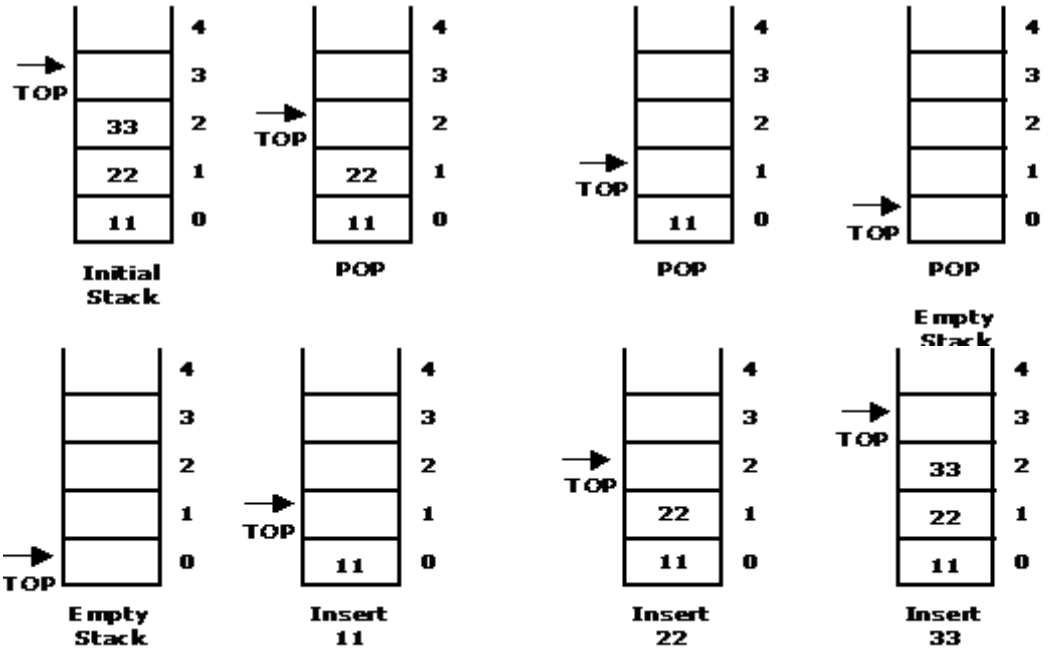
Introduction-

A stack is a container of objects that are inserted and removed according to the last-in first-out (LIFO) principle. In the pushdown stacks only two operations are allowed: **push** the item into the stack, and **pop** the item out of the stack. A stack is a limited access data structure - elements can be added and removed from the stack only at the top. **push** adds an item to the top of the stack, **pop** removes the item from the top. A helpful analogy is to think of a stack of books; you can remove only the top book, also you can add a new book on the top.
A stack may be implemented to have a bounded capacity. If the stack is full and does not contain enough space to accept an entity to be pushed, the stack is then considered to be in an overflow state. The pop operation removes an item from the top of the stack. A pop either reveals previously concealed items or results in an empty stack, but, if the stack is empty, it goes into underflow state, which means no items are present in stack to be removed.



Representation of a Stack using Arrays:
Let us consider a stack with 6 elements capacity. This is called as the size of the stack. The number of elements to be added should not exceed the maximum size of the stack. If we attempt to add new element beyond the maximum size, we will encounter a stack overflow condition. Similarly, you cannot remove elements beyond the base of the stack. If such is the case, we will reach a stack underflow condition.

When a element is added to a stack, the operation is performed by push().

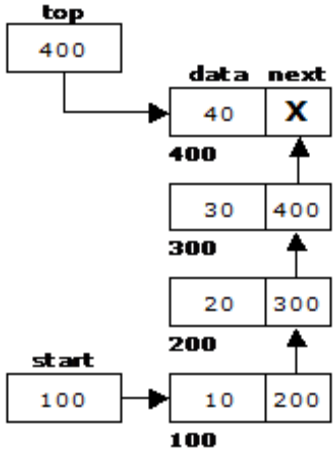


procedure:

STACK: Stack is a linear data structure which works under the principle of last in first out. Basic operations: push, pop, display.
1. PUSH: if (top==MAX), display Stack overflow else reading the data and making stack [top] =data and incrementing the top value by doing top++.
2. Pop: if (top==0), display Stack underflow else printing the element at the top of the stack and decrementing the top value by doing the top.
3. DISPLAY: IF (TOP==0), display Stack is empty else printing the elements in the stack from stack [0] to stack [top].

Linked List Implementation of Stack:

We can represent a stack as a linked list. In a stack push and pop operations are performed at one end called top. We can perform similar operations at one end of list using top pointer.



Stack Applications:

1. Stack is used by compilers to check for balancing of parentheses, brackets and braces.
2. Stack is used to evaluate a postfix expression.
3. Stack is used to convert an infix expression into postfix/prefix form.
4. In recursion, all intermediate arguments and return values are stored on the processor's stack.
5. During a function call the return address and arguments are pushed onto a stack and on return they are popped off.

In-fix- to Postfix Transformation:

Procedure:

Procedure to convert from infix expression to postfix expression is as follows:

1. Scan the infix expression from left to right.
2. a) If the scanned symbol is left parenthesis, push it onto the stack.

b) If the scanned symbol is an operand, then place directly in the postfix expression (output).

c) If the symbol scanned is a right parenthesis, then go on popping all the items from the stack and place them in the postfix expression till we get the matching left parenthesis.

d) If the scanned symbol is an operator, then go on removing all the operators from the stack and place them in the postfix expression, if and only if the precedence of the operator which is on the top of the stack is greater than (or equal) to the precedence of the scanned operator and push the scanned operator onto the stack otherwise, push the scanned operator onto the stack.

Evaluating Arithmetic Expressions:

Procedure:

The postfix expression is evaluated easily by the use of a stack. When a number is seen, it is pushed onto the stack; when an operator is seen, the operator is applied to the two numbers that are popped from the stack and the result is pushed onto the stack.

PROGRAMS:

1: Write a program to convert infix expression to postfix expression and evaluate postfix expression

PROGRAM:

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include<ctype.h>
int st[100],sttop=-1;

void inpost(char [ ],int,int);
void pushitem(int it);
int popitem();
int stisp(char t);
int sticp(char t);
int main()
{
char in[100],post[100];
int low=0,high;
printf("enter infix exp\n");
scanf("%s",in);
printf("the length is %d\n",strlen(in));
high=strlen(in)-1;
printf("the given  expression is %s\n",in);
inpost(in,0,high);
return 0;
}
```

```
void pushitem(int it)
{
if(sttop==99)
{
printf("stack full\n");
}
st[++sttop]=it;
}

int popitem()
{
int it;
if(sttop==-1)
{
printf("stack empty");
}
return st[sttop--];
}

void inpost(char in[ ],int low,int high)
{
int x=0,y=0,z,result=0;
char a,c,post[100];
char t;
pushitem('\0');
```

```
t=in[x];



while(t!='\0')
        {
        if(isalnum(t))
                {
                post[y]=t;
                y++;
                }
                else if(t=='(')
                {
                pushitem('(');
                }
                else if(t==')')
                {
                        while(st[sttop]!='(')
                        {
                        c=popitem();
                        post[y]=c;
                        y++;
                        }
                c=popitem();
                }
                else
                {

                        while(stisp(st[sttop])>=sticp(t))
```

```
                    {
                    c=popitem();
                    post[y]=c;
                    y++;
                    }
            pushitem(t);
              }
              x++;
              t=in[x];
        }


while(sttop!=-1)
{
c=popitem();
post[y]=c;
y++;
}



printf("\n The postfix exp is\n");
for(z=0;z<y;z++)
{
printf("%c",post[z]);
}
}
```

```
int stisp(char t)
{
switch(t)
{
case '(':return 10;
case ')':return 9;
case '+':return 7;
case '-':return 7;
case '*':return 8;
case '/':return 8;
case '\0':return 0;
default:printf("invalid expression");
break;
}
return 0;
}
```

```
int sticp(char t)
{
switch(t)
{
case '(':return 10;
case ')':return 9;
case '+':return 7;
case '-':return 7;
```

```
case '*':return 8;
case '/':return 8;
case '\0':return 0;
default:printf("invalid expression");
break;
}
return 0;
}
```

INPUT/OUTPUT:

```
gpcetadmin@gpcetadmin-MS-7528:~$ gcc anil1infix.c
gpcetadmin@gpcetadmin-MS-7528:~$ ./a.out
enter infix exp
5+2
the length is 3
the given  expression is 5+2
 The postfix exp is
52+
```

2. Write a program to evaluate a postfix expression

PROGRAM:

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include<ctype.h>
void pushitem(int it);
int popitem();
int cal(char post[],int,int);
int st[100],sttop=-1;

int main()
{
char post[100];
int low=0,high,result=0;
printf("enter postfix exp\n");
scanf("%s",post);
printf("the length is %d\n",strlen(post));
high=strlen(post)-1;
result=cal(post,low,high);
printf("%d\n",result);
return 0;
}
void pushitem(int it)
{
if(sttop==99)
{
printf("stack full\n");
}
```

```
st[++sttop]=it;
}

int popitem()
{
int it;
if(sttop==-1)
{
printf("stack empty");
}
return st[sttop--];
}

int cal(char post[ ],int low,int high)
{
int m,n,x,y,j=0,len;
len=strlen(post);
while(j<len)
{
if(isdigit(post[j]))
{
x=post[j]-'0';
pushitem(x);
}
else
{
m=popitem();
n=popitem();
```

```
switch(post[j])
{
case '+':x=n+m;break;
case '-':x=n-m;break;
case '*':x=n*m;break;
case '/':x=n/m;break;
}
pushitem(x);
}
j++;
}
if(sttop>0)
{
printf("no of operands are more than operators\n");
}
else
{
y=popitem();
return y;
}
return 0;
}
```

INPUT/OUTPUT:

```
gpcetadmin@gpcetadmin-MS-7528:~$ gedit anil1eval.c
gpcetadmin@gpcetadmin-MS-7528:~$ gcc anil1eval.c
gpcetadmin@gpcetadmin-MS-7528:~$ ./a.out
enter postfix exp
```

52+

the length is 3

7

gpcetadmin@gpcetadmin-MS-7528:~$

3. Write a program to implement stack using array

PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
int t=-1,n;
int starr[20];
void pushele(int ele);
int popele();
void display();
int main()
{
int num1=0,num2=0,choice;
printf("enter size of stack\n");
scanf("%d",&n);
while(1)
{
printf("\n1.push");
printf("\n2.pop");
printf("\n3.display");
printf("\n 4.exit");
```

```
printf("\n enter u r choice");
scanf("%d",&choice);
switch(choice)
{
case 1:printf("enter element to be pushed\n");
        scanf("%d",&num1);

        pushele(num1);
        break;
case 2:num2=popele();
        printf("element deleted=%d\n",num2);
        break;
case 3:display();
        break;
case 4:exit(0);
        break;
default:printf("invalid choice");
        break;
}
}
return 0;
}

void pushele(int ele)
{
if(t==n-1)
{
printf("stack full\n");
```

```
}
else
{
starr[++t]=ele;
}
}

int popele()
{
if(t==-1)
{
printf("stack empty");
}
return starr[t--];
}

void display()
{
int i;
for(i=t;i>=0;i--)
{
printf("%d\t",starr[i]);
}
}
```

INPUT/OUTPUT:

```
gpcetadmin@gpcetadmin-Veriton-Series:~$ gedit sta.c
gpcetadmin@gpcetadmin-Veriton-Series:~$ gcc sta.c
```

```
gpcetadmin@gpcetadmin-Veriton-Series:~$ ./a.out
enter size of stack
3
1.push
2.pop
3.display
 4.exit
 enter u r choice1
enter element to be pushed
5

1.push
2.pop
3.display
4.exit
 enter u r choice1
enter element to be pushed
10

1.push
2.pop
3.display
4.exit
 enter u r choice1
enter element to be pushed
15

1.push
```

2.pop

3.display

4.exit

enter u r choice1

enter element to be pushed

20

stack full

1.push

2.pop

3.display

4.exit

enter u r choice3

15 10 5

1.push

2.pop

3.display

4.exit

enter u r choice2

element deleted=15

1.push

2.pop

3.display

4.exit

enter u r choice3

10 5

1.push

2.pop

3.display

4.exit

enter u r choice 4

4. Write a program to implement stack using linked list

PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
struct stpoint
{
int ele;
struct stpoint *l;
}*t;
int i;
void pushele(int i);
int popele();
void displayele();
int main()
{
int num1=0,num2=0,choice;
while(1)
{
printf("\n 1. push");
printf("\n 2.pop");
printf("\n 3.display");
printf("\n 4.exit");
```

```
printf("enter u r choice");
scanf("%d",&choice);
switch(choice)
{
case 1:printf("enter element to be pushed\n");
        scanf("%d",&num1);
        pushele(num1);
        break;
case 2:num2=popele();
        printf("element deleted=%d\n",num2);
        break;
case 3:displayele();
        break;
case 4:exit(1);
        break;
default:printf("invalid choice");
        break;
}
}
return 0;
}
void pushele(int j)
{
struct stpoint *m;
m=(struct stpoint*)malloc(sizeof(struct stpoint));
m->ele=j;
m->l=t;
t=m;
```

```
return;
}


int popele()
{
if(t==NULL)
{
printf("stack empty\n");
}
else
{
int i;
i=t->ele;
t=t->l;
return i;
}
return 0;
}

void displayele()
{
struct stpoint *pointer=NULL;
pointer=t;
while(pointer!=NULL)
{
printf("%d\t",pointer->ele);
pointer=pointer->l;
```

```
}
}
```

INPUT/OUTPUT :

```
gpcetadmin@gpcetadmin-Veriton-Series:~$ gcc stl.c
gpcetadmin@gpcetadmin-Veriton-Series:~$ ./a.out


 1. push
 2.pop
 3.display
 4.exit
enter u r choice1
enter element to be pushed
5

 1.push
 2.pop
 3.display
 4.exit
enter u r choice1
enter element to be pushed
10

 1.push
 2.pop
 3.display
 4.exit
```

enter u r choice1

enter element to be pushed

15

1. push

2.pop

3.display

4.exit

enter u r choice1

enter element to be pushed

20

1.push

2.pop

3.display

4.exit

enter u r choice3

20 15 10 5

1.push

2.pop

3.display

4.exit

enter u r choice2

element deleted=20

1.push

2.pop

3.display

4.exit

enter u r choice3

15 10 5

1.push

2.pop

3.display

4.exit

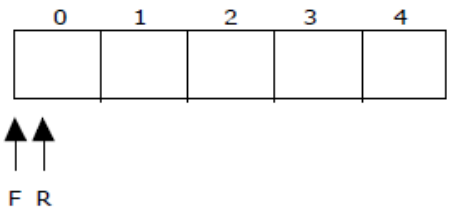
enter u r choice 4

Queues: **Introduction**

A queue is a data structure that is best described as "first in, first out". A queue is another special kind of list, where items are inserted at one end called the rear and deleted at the other end called the front. A real world example of a queue is people waiting in line at the bank. As each person enters the bank, he or she is "enqueued" at the back of the line. When a teller becomes available, they are "dequeued" at the front of the line.

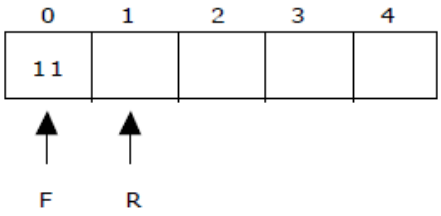
Representation of a Queue using Array:

Let us consider a queue, which can hold maximum of five elements. Initially the queue is empty.



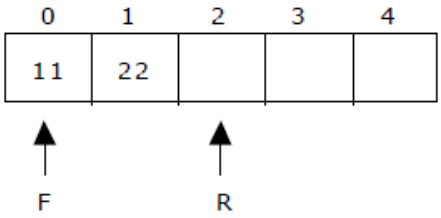
Queue Empty
FRONT = REAR = 0

Now, insert 11 to the queue. Then queue status will be:



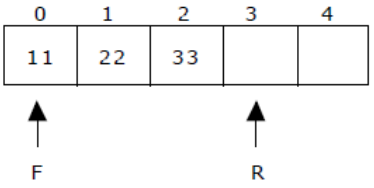
REAR = REAR + 1 = 1
FRONT = 0

Next, insert 22 to the queue. Then the queue status is:



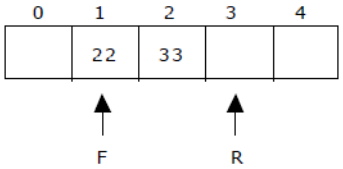
REAR = REAR + 1 = 2
FRONT = 0

Again insert another element 33 to the queue. The status of the queue is:



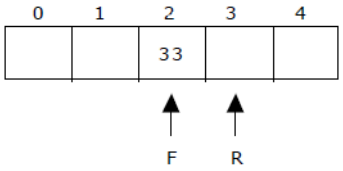
REAR = REAR + 1 = 3
FRONT = 0

Now, delete an element. The element deleted is the element at the front of the queue. So the status of the queue is:



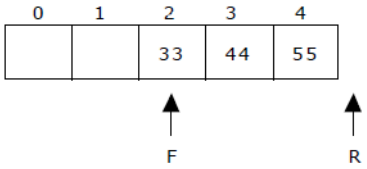
REAR = 3
FRONT = FRONT + 1 = 1

Again, delete an element. The element to be deleted is always pointed to by the FRONT pointer. So, 22 is deleted. The queue status is as follows:



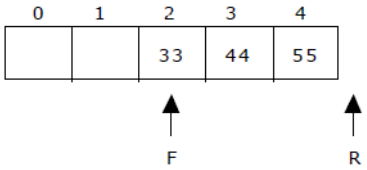
REAR = 3
FRONT = FRONT + 1 = 2

Now, insert new elements 44 and 55 into the queue. The queue status is:



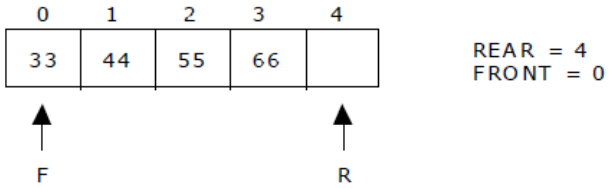
REAR = 5
FRONT = 2

Next insert another element, say 66 to the queue. We cannot insert 66 to the queue as the rear crossed the maximum size of the queue (i.e., 5). There will be queue full signal. The queue status is as follows:



REAR = 5
FRONT = 2

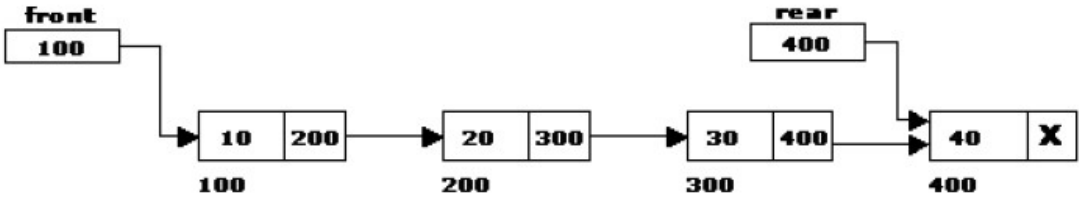
Now it is not possible to insert an element 66 even though there are two vacant positions in the linear queue. To over come this problem the elements of the queue are to be shifted towards the beginning of the queue so that it creates vacant position at the rear end. Then the FRONT and REAR are to be adjusted properly. The element 66 can be inserted at the rear end. After this operation, the queue status is as follows:



This difficulty can overcome if we treat queue position with index 0 as a position that comes after position with index 4 i.e., we treat the queue as a **circular queue.**

Procedure for Queue operations using array:
In order to create a queue we require a one dimensional array Q(1:n) and two variables *front* and *rear*. The conventions we shall adopt for these two variables are that *front* is always 1 less than the actual front of the queue and rear always points to the last element in the queue. Thus, front = rear if and only if there are no elements in the queue. The initial condition then is front = rear = 0.
The various queue operations to perform creation, deletion and display the elements in a queue are as follows:
1. insertQ(): inserts an element at the end of queue Q.
2. deleteQ(): deletes the first element of Q.
3. displayQ(): displays the elements in the queue.

Linked List Implementation of Queue: We can represent a queue as a linked list. In a queue data is deleted from the front end and inserted at the rear end. We can perform similar operations on the two ends of a list. We use two pointers front and rear for our linked queue implementation.



Applications of Queues:

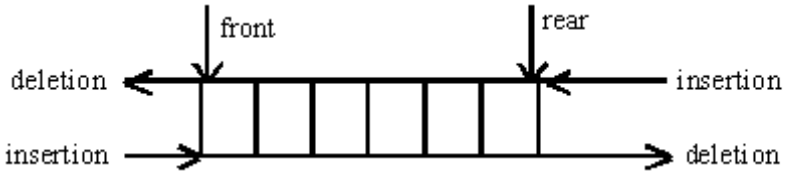
1. It is used to schedule the jobs to be processed by the CPU.

2. When multiple users send print jobs to a printer, each printing job is kept in the printing queue. Then the printer prints those jobs according to first in first out (FIFO) basis.

3. Breadth first search uses a queue data structure to find an element from a graph.

Disadvantages of Linear Queue:
There are two problems associated with linear queue. They are:
1. Time consuming: linear time to be spent in shifting the elements to the beginning of the queue.
2.Signaling queue full: even if the queue is having vacant position

DEQUE(Double Ended Queue):
A **double-ended queue** (**dequeue**, often abbreviated to **deque**, pronounced *deck*) generalizes a queue, for which elements can be added to or removed from either the front (head) or back (tail).It is also often called a **head-tail linked list.**
Like an ordinary queue, a double-ended queue is a data structure it supports the following operations: enq_front, enq_back, deq_front, deq_back, and empty.
Dequeue can be behave like a queue by using only enq_front and deq_front , and behaves like a stack by using only enq_front and deq_rear.
The DeQueue is represented as follows.



DEQUE can be represented in two ways they are
1) Input restricted DEQUE(IRD)
2) output restricted DEQUE(ORD)
The output restricted DEQUE allows deletions from only one end and input restricted DEQUE allow insertions at only one end. The DEQUE can be constructed in two ways they are
1) Using array
2)Using linked list

Applications of DEQUE:

1. The A-Steal algorithm implements task scheduling for several processors (multiprocessor scheduling).
2. The processor gets the first element from the deque.
3. When one of the processor completes execution of its own threads it can steal a thread from another processor.
4. It gets the last element from the deque of another processor and executes it.

Circular Queue:

Circular queue is a linear data structure. It follows FIFO principle. In circular queue the last node is connected back to the first node to make a circle.

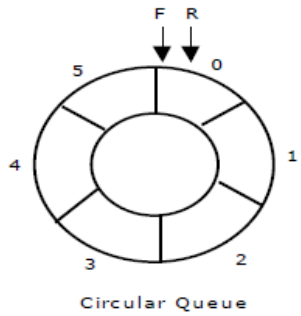
- Circular linked list fallow the First In First Out principle
- Elements are added at the rear end and the elements are deleted at front end of the queue
- Both the front and the rear pointers points to the beginning of the array.
- It is also called as “Ring buffer”.
- Items can inserted and deleted from a queue in O(1) time.

Circular Queue can be created in three ways they are

- Using single linked list
- Using double linked list
- Using arrays

Representation of Circular Queue:

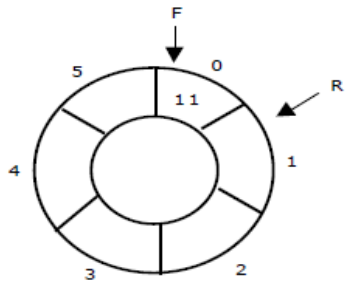
Let us consider a circular queue, which can hold maximum (MAX) of six elements. Initially the queue is empty.



Circular Queue

Queue Empty
MAX = 6
FRONT = REAR = 0
COUNT = 0

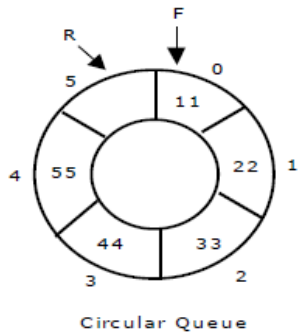
Now, insert 11 to the circular queue. Then circular queue status will be:



Circular Queue

FRONT = 0
REAR = (REAR + 1) % 6 = 1
COUNT = 1

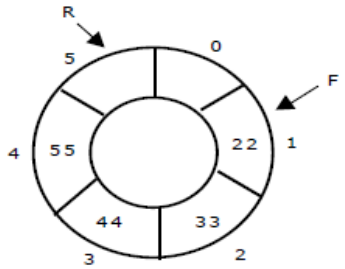
Insert new elements 22, 33, 44 and 55 into the circular queue. The circular queue status is:



Circular Queue

FRONT = 0, REAR = 5
REAR = REAR % 6 = 5
COUNT = 5

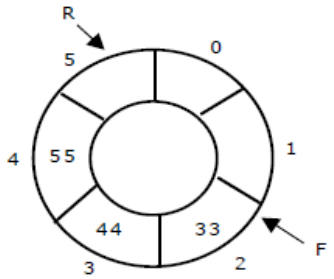
Now, delete an element. The element deleted is the element at the front of the circular queue. So, 11 is deleted. The circular queue status is as follows:



Circular Queue

```
FRONT = (FRONT + 1) % 6 = 1
REAR = 5
COUNT = COUNT - 1 = 4
```

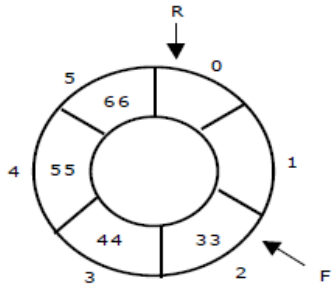
Again, delete an element. The element to be deleted is always pointed to by the FRONT pointer. So, 22 is deleted. The circular queue status is as follows:



Circular Queue

```
FRONT = (FRONT + 1) % 6 = 2
REAR = 5
COUNT = COUNT - 1 = 3
```

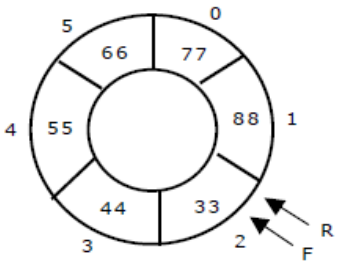
Again, insert another element 66 to the circular queue. The status of the circular queue is:



Circular Queue

```
FRONT = 2
REAR = (REAR + 1) % 6 = 0
COUNT = COUNT + 1 = 4
```

Now, insert new elements 77 and 88 into the circular queue. The circular queue status is:



Circular Queue

FRONT = 2, REAR = 2
REAR = REAR % 6 = 2
COUNT = 6

Now, if we insert an element to the circular queue, as COUNT = MAX we cannot add the element to circular queue. So, the circular queue is *full*.

PROGRAMS:

1. Write a program to implement linear queue using arrays .

PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
int q[100];
int f=-1,r=-1,n;
int main()
{
int i,choice,item,ri;
printf("enter size\n");
scanf("%d",&n);
while(1)
{
printf("\n 1.insertion");
```

```
printf("\n 2.deletion");
printf("\n 3. display");
printf("\n 4.exit");
printf("\n enter u r choice");
scanf("%d",&choice);
switch(choice)
{
case 1:insert();break;
case 2:delete();break;
case 3:display();break;
case 4:exit(0);break;
default:printf("invalid option\n");
        break;
}
}
return 0;
}

insert()
{
int item;
if(r==n-1)
{
printf("queue full\n");
}
else if(f==-1 && r==-1)
{
```

```
f=0;
r=0;
printf("enter item\n");
scanf("%d",&item);
q[r]=item;
}
else
{
r++;
printf("enter item\n");
scanf("%d",&item);
q[r]=item;
}
}

delete()
{
int ri;
if(f==-1)
{
printf("queue empty");
}
else if(f==r)
{
ri=q[f];
f=-1;
r=-1;
}
```

```
else
{
ri=q[f];
f++;
}
printf("deleted data = %d",ri);
}



display()
{
int i;
for(i=f;i<=r;i++)
{
printf("%d\t",q[i]);
}
}
```

INPUT/OUTPUT:

```
gpcetadmin@gpcetadmin-Veriton-Series:~$ gcc lqa.c
gpcetadmin@gpcetadmin-Veriton-Series:~$ ./a.out
enter size
3

 1.insertion
 2.deletion
 3. display
 4.exit
```

enter u r choice1

enter item

10

1.insertion

2.deletion

3. display

4.exit

enter u r choice1

enter item

20

1.insertion

2.deletion

3. display

4.exit

enter u r choice1

enter item

30

1.insertion

2.deletion

3. display

4.exit

enter u r choice1

queue full

1.insertion

2.deletion

3. display

4.exit

enter u r choice2

deleted data = 10

1.insertion

2.deletion

3. display

4.exit

enter u r choice3

20 30

1.insertion

2.deletion

3. display

4.exit

enter u r choice

4

2.Write a program to implement linear queue using linked list

PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
struct queue
```

```
{
int element;
struct queue *next;
};
struct queue *front=NULL;
struct queue *rear=NULL;
void add(int);
int rem();
void display();
int main()
{
int num1,num2,choice;

while(1)
{
printf("\n 1.insert");
printf("\n 2.delete");
printf("\n 3.display");
printf("\n 4.exit");
printf("\n enter u r choice");
scanf("%d",&choice);
switch(choice)
{
case 1:printf("enter element to be inserted into queue");
        scanf("%d",&num1);
        add(num1);
        break;
case 2:num2=rem();
```

```
        if(num2==-9999)
        printf("queue empty");
        else
        printf("element removed from queue=%d",num2);
        break;
case 3:display();break;
case 4:exit(1);break;
default:printf("invalid choice");break;
}
}
return 0;
}

void add(int value)
{
struct queue *ptr=(struct queue*)malloc(sizeof(struct queue));
ptr->element=value;
if(front==NULL)
{
front=rear=ptr;
ptr->next=NULL;
}
else
{
rear->next=ptr;
ptr->next=NULL;
rear=ptr;
}
```

```
}



int rem()
{
int i;
if(front==NULL)
{
return -9999;
}
else
{
i=front->element;
front=front->next;
return i;
}
}

void display()
{
struct queue *ptr=front;
if(front==NULL)
{
printf("queue is empty");
return;
}
else
{
```

```
while(ptr!=rear)
{
printf("%d\t",ptr->element);
ptr=ptr->next;
}
printf("%d\t",rear->element);
}
}
```

INPUT/OUTPUT:

```
gpcetadmin@gpcetadmin-Veriton-Series:~$ gcc lql.c
gpcetadmin@gpcetadmin-Veriton-Series:~$ ./a.out

 1.insert
 2.delete
 3.display
 4.exit
 enter u r choice1
enter element to be inserted into queue10


 1.insert
 2.delete
 3.display
 4.exit
 enter u r choice1
enter element to be inserted into queue20
```

1.insert

2.delete

3.display

4.exit

enter u r choice1

enter element to be inserted into queue30

1.insert

2.delete

3.display

4.exit

enter u r choice3

10 20 30

1.insert

2.delete

3.display

4.exit

enter u r choice2

element removed from queue=10

1.insert

2.delete

3.display

4.exit

enter u r choice3

20 30

```
 1.insert
 2.delete
 3.display
 4.exit
 enter u r choice
4
```

3. Write a program to implement circular queue using arrays

PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
int q[50],f=-1,r=-1,n;
int main()
{
int i,choice,item,ri;
printf("enter size\n");
scanf("%d",&n);
while(1)
{
printf("\n menu");
printf("\n1.insertion");
printf("\n2.deletion");
printf("\n3.display");
printf("\n4.exit");
printf("\n enter u r choice");
scanf("%d",&choice);
```

```
switch(choice)
{
case 1:insertion();break;
case 2:deletion();break;
case 3:display();break;
case 4:exit(0);break;
default:printf("invalid choice");break;
}
}
return 0;
}

insertion()
{
int item;
if((f==0 && r==n-1) || (f==r+1))
{
printf("queue is full");
}
else if(f==-1 && r==-1)
{
f=0;

r=0;
printf("enter item\n");
scanf("%d",&item);
}
else if(r==n-1)
```

```
{
r=0;
printf("enter item\n");
scanf("%d",&item);
}
else
{
r++;
printf("enter item\n");
scanf("%d",&item);
}
q[r]=item;
}

deletion()
{
int ri;
ri=q[f];
if(f==-1)
{
printf("queue is empty");
}
else if(f==r)
{
f=-1;
r=-1;
}
else if(f==n-1)
```

```
{
f=0;
}
else
{
f++;
}
printf("deleted data=%d\n",ri);
}
```

```
display()
{
int i;
if(f<=r)
{
for(i=f;i<=r;i++)
printf("%d\t",q[i]);
}
else
{
for(i=f;i<=n-1;i++)
printf("%d\t",q[i]);
for(i=0;i<=r;i++)
```

```
printf("%d\t",q[i]);
}
}
```

INPUT/OUTPUT:

```
gpcetadmin@gpcetadmin-Veriton-Series:~$ gedit cqa.c
gpcetadmin@gpcetadmin-Veriton-Series:~$ gcc cqa.c
gpcetadmin@gpcetadmin-Veriton-Series:~$ ./a.out
enter size
3

 menu
1.insertion
2.deletion
3.display
4.exit
 enter u r choice1
enter item
10
 menu
1.insertion
2.deletion
3.display
4.exit
 enter u r choice1
enter item
20
```

menu

1.insertion

2.deletion

3.display

4.exit

enter u r choice3

10 20

menu

1.insertion

2.deletion

3.display

4.exit

enter u r choice2

deleted data=10

menu

1.insertion

2.deletion

3.display

4.exit

enter u r choice3

20

menu

1.insertion

2.deletion

3.display

4.exit

enter u r choice1

enter item

30

menu

1.insertion

2.deletion

3.display

4.exit

enter u r choice3

20 30

menu

1.insertion

2.deletion

3.display

4.exit

enter u r choice1

enter item

40

menu

1.insertion

2.deletion

3.display

4.exit

enter u r choice3

20 30 40

menu

1.insertion

2.deletion

3.display

4.exit

enter u r choice1

queue is full

menu

1.insertion

2.deletion

3.display

4.exit

enter u r choice 4

Tables:

Hashtables:

Introduction
Any large information source (data base) can be thought of as a table (with multiple fields), containing information.

For example:

A telephone book has fields *name*, *address* and *phone number*. When you want to find somebody's phone number, you search the book based on the *name* field.

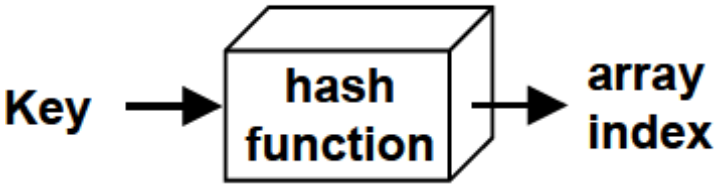
A user account on AA-Design, has the fields *user_id*, *password* and *home folder*. You logon using your *user_id* and *password* and it takes you to your *home folder*.

To find an entry (field of information) in the table, you only have to use the contents of **one** of the fields (say name in the case of the telephone book). You **don't** have to know the contents of all the fields. The field you use to find the contents of the other fields is called the **key**.Ideally, the key should uniquely identify the entry, i.e.

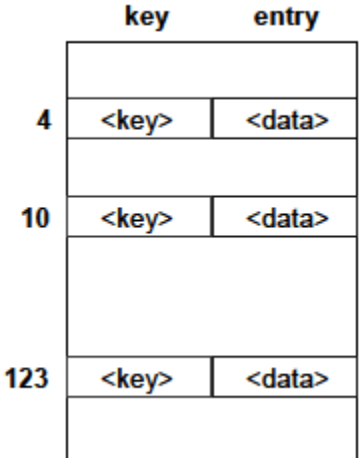
if the key is the *name* then no two entries in the telephone book have the same name. We can treat the Table formulation as an abstract data type

Hashing
Having an insertion, find and removal of O(log(N)) is good but as the size of the table
becomes larger, even this value becomes significant. We would like to be able to use an
algorithm for finding of O(1). This is when hashing comes into play!

Hashing using Arrays
When implementing a hash table using arrays, the nodes are not stored consecutively,
instead the location of storage is computed using the key and a ***hash*** function. The computation of the array index can be visualized as shown below:



The value computed by applying the hash function to the key is often referred to as the
hashed key. The entries into the array, are scattered (not necessarily sequential) as can be
seen in figure below.



The cost of the insert, find and delete operations is now only O(1). Can you think of why?
Hash tables are very good if you need to perform a lot of search operations on a relatively
stable table (i.e. there are a lot fewer insertion and deletion operations than search operations).
One the other hand, if traversals (covering the entire table), insertions, deletions are a lot
more frequent than simple search operations, then ordered binary trees (also called AVL trees) are the preferred implementation choice.

Hashing Performance
There are three factors the influence the performance of hashing:
Hash function

- should distribute the keys and entries evenly throughout the entire table
- should minimize collisions

Collision resolution strategy

- Open Addressing: store the key/entry in a different position
- Separate Chaining: chain several keys/entries in the same position

Table size

- Too large a table, will cause a wastage of memory
- Too small a table will cause increased collisions and eventually force *rehashing* (creating a new hash table of larger size and copying the contents of the current hash table into it)
- The size should be appropriate to the hash function used and should typically be a prime number. Why? (We discussed this in class).
-

Selecting Hash Functions
The hash function converts the key into the table position. It can be carried out using:

Modular Arithmetic: Compute the index by dividing the key with some value and use the remainder as the index. This forms the basis of the next two techniques.
For Example: index := keyMOD table_size
Truncation: Ignoring part of the key and using the rest as the array index. The problem with this approach is that there may not always be an even distribution throughout the table.
For Example: If student id's are the key 928324312 then select just the last three digits as the index i.e. 312 as the index. => the table size has to be atleast 999.Why?
Folding: Partition the key into several pieces and then combine it in some convenient way.
For Example:

- For an 8 bit integer, compute the index as follows:
  Index := (Key/10000 + KeyMOD 10000) MOD Table_Size.
- For character strings, compute the index as follows:

```
Index :=0
For I in 1.. length(string)
   Index := Index + ascii_value(String(I))
```