DATA STRUCTURES

UNIT-1:

Data Structure:

It is a particular way of organizing and storing data in a computer. So that it can be accessed and modified efficiently. A data structure will have a collection of data and functions or operations that can be applied on the data.

TYPES OF DATA STRUCTURES

1. Primitive and Non-Primitive Data Structure

Primitive Data Structure defines a set of primitive elements that do not involve any other elements as its subparts. These are generally built-in data type in programming languages.

E.g.:- Integers, Characters etc.

Non-Primitive Data Structures are that defines a set of derived elements such as Arrays, Structures and Classes.

2. Linear and Non-Linear Data Structure

A Data Structure is said to be linear, if its elements from a sequence and each element have a unique successor and predecessor.

E.g.:- Stack, Queue etc.

Non-Linear Data Structures are used to represent data that have a hierarchical relationship among the elements. In non-linear Data Structure every element has more than one predecessor and one successor.

E.g.:- Trees, Graphs

3. Static and Dynamic Data Structure

A Data Structure is referred as Static Data Structure, if it is created before program execution i.e., during compilation time. The variables of Static Data Structure have user specified name.

E.g.:- Array

Data Structures that are created at run time are called as Dynamic Data Structure. The variables of this type are known always referred by user defined name instead using their addresses through pointers.

E.g.:- Linked List

4. Sequential and Direct Data Structure

This classification is with respect to access operation associated with the data type. Sequential access means the locations are accessed in a sequential form.

E.g.:- To access n^{th} element, it must access preceding $n-1$ data elements

E.g.:- Linked List

Direct Access means any element can access directly without accessing its predecessor or successor i.e., n^{th} element can be accessed directly.

E.g.:- Array

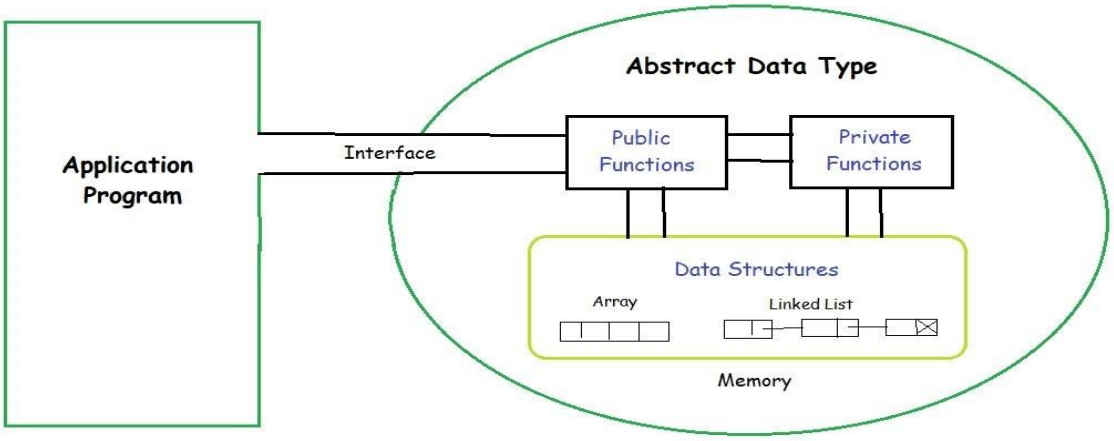
OPERATIONS ON DATA STRUCTURE

The basic operations that can be performed on a Data Structure are:

i) Insertion: - Operation of storing a new data element in a Data Structure.
ii) Deletion: - The process of removal of data element from a Data Structure.
iii) Traversal: - It involves processing of all data elements present in a Data Structure.
iv) Merging: - It is a process of compiling the elements of two similar Data Structures to form a new Data Structure of the same type.
v) Sorting: - It involves arranging data element in a Data Structure in a specified order.
vi) Searching: - It involves searching for the specified data element in a Data Structure.

ABSTRACT DATA TYPE

An Abstract Data Type (ADT) is a conceptual model that defines a set of operations and behaviours for a data type, without specifying how these operations are implemented or how data is organized in memory. The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called “abstract” because it provides an implementation-independent view. The process of providing only the essentials and hiding the details is known as abstraction.



For example, we use primitive values like int, float, and char with the understanding that these data types can operate and be performed on without any knowledge of their implementation details. ADTs operate similarly by defining what operations are possible without detailing their implementation.

IMPORTANCE OF AN ADT

1. Used to create a fast algorithm in less time.
2. Help to manage and organize the data.
3. It makes code clean and easy to understand.
4. It makes the execution program fast.
5. Consume less time.

COMPLEXITY OF ALGORITHMS

Generally algorithms are measured in terms of time complexity and space complexity.

1. Time Complexity

Time Complexity of an algorithm is a measure of how much time is required to execute an algorithm for a given number of inputs. And it is measured by its rate of growth relative to a standard function. Time Complexity can be calculated by adding compilation time and execution time. Or it can do by counting the number of steps in an algorithm.

2. Space Complexity

Space Complexity of an algorithm is a measure of how much storage is required by the algorithm. Thus space complexity is the amount of computer memory required during program execution as a function of input elements. The space requirement of algorithm can be

performed at compilation time and run time.

ASYMPTOTIC ANALYSIS

For analysis of algorithm it needs to calculate the complexity of algorithms in terms of resources, such as time and space. But when complexity is calculated, it does not provide the exact amount of resources required. Therefore instead of taking exact amount of resources, the complexity of algorithm is represented in a general mathematical form which will give the basic nature of algorithm.

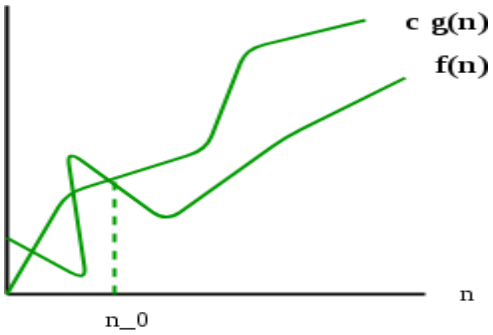
Thus on analyzing an algorithm, it derives a mathematical formula to represent amount of time and space required for the execution. The Asymptotic analysis of algorithm evaluates the performance of an algorithm in terms of input size It calculates how does the time taken by an algorithm increases with input size. It focuses on:

i) Analyzing the problem with large input size.
ii) Considers only leading terms of formula. Since the lower order term contracts lesser to the overall complexity as input grows larger.
iii) Ignores the coefficient of leading term.

ASYMPTOTIC NOTATION

The commonly used Asymptotic Notations to calculate the complexity of algorithms are:

1. **Big Oh – O: -** The notation **O(n)** is the formal way to express the upper bound of an algorithm's complexity.

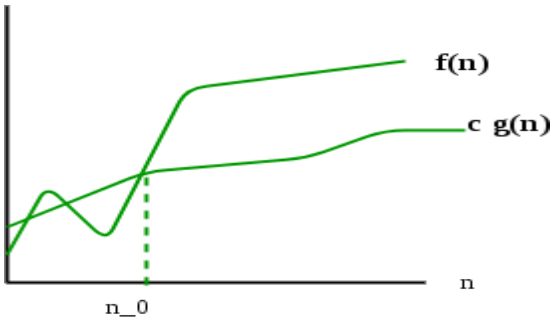


f(n) = O(g(n))

O(g(n)) = { f(n): there exist positive constants c and

n0 such that 0 <= f(n) <= c*g(n) for all n >= n0}

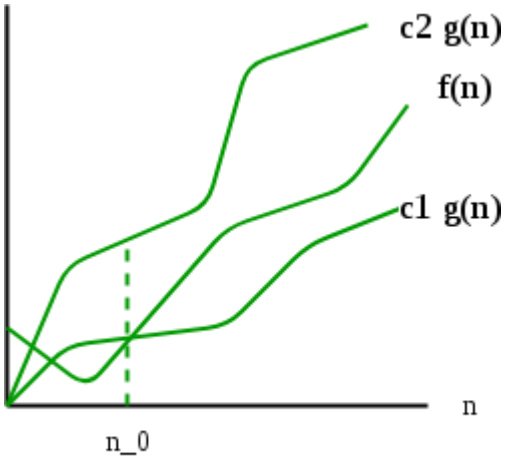
2. **Big Omega – Ω: -** The notation **Ω(n)** is the formal way of representing lower bound of an algorithm's complexity.



f(n) = Omega(g(n))

Ω (g(n)) = {f(n): there exist positive constants c and
n0 such that 0 <= c*g(n) <= f(n) for all n >= n0}.

3. **Big Theta** – **Θ**: **-** The notation **Θ(*n*)** is the formal way to representing lower bound and upper bound of an algorithm's complexity.



f(n) = theta(g(n))

Θ(g(n)) = {f(n): there exist positive constants c1, c2 and n0 such that 0 <= c1*g(n) <= f(n) <=
c2*g(n) for all n >= n0}

Unit 2:

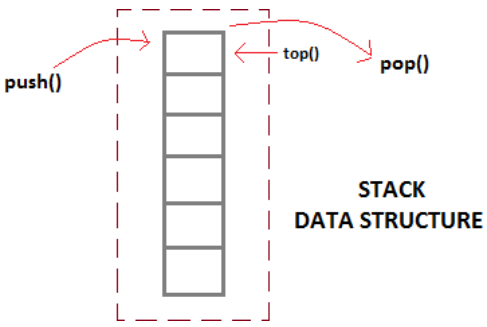
STACK

Stack is a linear Data Structure in which all the insertion and deletion operations are done at one end, called top of the stack and operations are performed in Last In First Out [**LIFO**] order. A stack can be implemented by using an array or by using linked list.

The array representation method needs to set the amount memory needed initially. So they are limited in size, i.e. it can't shrink or expand. Linked List representation uses Dynamic memory management method. So they are not limited in size i.e. they can shrink or expand.

The class declaration for stack using array is represented as:

```
class stack
 {
        int a[10], st;
        public:
        stack()
        {
        st =-1;
        }
        void push(int);
        void pop();
};
```



The operations performed on Stack are:

1. **Push Operation:** This function is used to store data into the stack. The Push operation performs following steps:
   - **i.** Check whether stack is full or not.
   - **ii.** If full, message that stack is full.

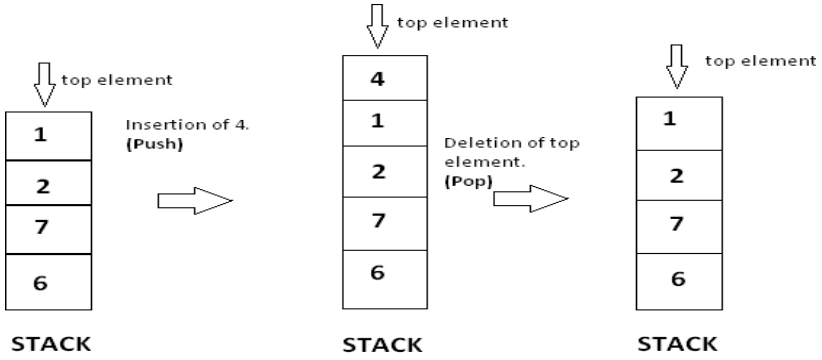
iii. If not full, it will increase the stack top by 1 and to that location new data is stored.

```
void stack :: push(int d)
{
        if (st>=size-1)
                cout<<"stack is full";
        else
                a[++st]=d;
}
```

2. **Pop Operation:** This function is used to remove data from stack. The Pop function performs following functions:
   i. Check if stack is empty or not.
   ii. If stack is empty, print a message.
   iii. If stack is not empty, print the element at the stack top and decrement stack top by one.

```
void stack :: pop()
{
        if(st==-1)
                cout<<"stack is empty";
        else
                cout<<a[st--];
}
```

The Time complexity of push and pop operations are O(1).



Implement Stack ADT using array:

```
//Template class for Stack
#include<iostream>
using namespace std;
template<class T>
class stack
{
int size,top;
T *a;
public:
stack(int);
int
isEmpty();
int isFull();
void push();
void pop();
};
template<class T>
stack<T>::stack(int s)
{
```

```
size=s;
a= new
T[size]; top=-
1;
}
template<class T>
int stack<T>::isEmpty()
{
if(top==-
1) return
1; else
return 0;
}
template<class T>
int
stack<T>::isFull()
{
if(top==size-1)
return 1;
else
return 0;
}
template<class T>
void stack<T>::push()
{
if(isFull())
{
cout<<"Stack is Full"<<endl;
}
```

```
else
{
int item;
cout<<"Enter the Data"<<endl;
cin>>item;
top++;
a[top]=item;
}}
template<class T>
void stack<T>::pop()
{
if(isEmpty())
cout<<"Stack is Empty"<<endl;
else
{
T x;
x=a[top];
top--;
cout<<"Data= "<<x<<endl;
} }
main()
{
stack<int>S(3)
; int y=1,op,t;
while(y==1)
{
cout<<"Enter 1-PUSH 2-POP 3-EXIT"<<endl;
cin>>op;
switch(op)
```

```
{
case 1: S.push();
        break;
case 2: S.pop();
        break;
case 3: y=0;
        break;
default: cout<<"Invalid Option"<<endl;
} }
return 0; }
```

APPLICATIONS OF STACK

1) Converting infix expression to postfix and prefix expression.
2) Evaluating postfix expression.
3) Checking nested parenthesis.
4) Reversing a string.
5) Processing function calls.
6) Stimulating recursion.
7) Parsing.
8) Book tracking algorithm.

POLISH NOTATIONS

This gives two alternatives to represent an arithmetic expression called postfix and prefix notations. The fundamental property of polish notation is that the order in which the operations are to be performed is determined by positions of operators and operands in the expression. Hence the advantage is that parenthesis is not required while writing expressions in polish notations. It also overrides associatively of operators. The conventional way of writing expression is called infix because in binary operations the operators occur between the operands. In postfix notation the operator is written after its operands.

In postfix notation, the operator is written after its operands.

E.g.: AB+

In prefix notation, the operator precedes its operands.

E.g.: +AB

Infix	Postfix	Prefix
(A+B)*C	AB+C*	*+ABC

Need for Prefix and Postfix expressions

1) Need for parenthesis as in infix expression is overcome in postfix and prefix notations.
2) Priority of the operators is no longer relevant.
3) The order of evaluation depends on the position of the operator and not on priority and associativity.
4) Expression evaluation is simpler.

Algorithm for evaluating Postfix expression

1) Start
2) Let E denote postfix expression.
3) Let stacktop=-1.
4) While(1) do
   Begin x=getnext(E)
   If(x==#)
   Then return
   If x is an operand, then push(x) otherwise
   Begin
    op2=pop,
    op1=pop,
    op3=evaluate(op1,x,op2)
    Push(op3)
   End
   End while
5) Stop.

Algorithm for Infix to Postfix conversion

1) Start
2) Let E be the Infix expression.
3) Scan expression E from left to right character by character till character is #.
   ch= getnext(E)
4) While(ch!=#
) If (ch==')')
   Then ch = pop();
   While(ch!='(')

Display ch
ch=pop();
End while
If(ch==operand) display
ch If(ch==operator)
If(icp>isp), then push(ch)
otherwise
While(icp
<=isp) ch =
pop() display ch
end while
ch= getnext()
end While

5) If(ch==#)
Then while(!isempty())
ch=pop()
display ch
end while
6) Stop.

Q. Implement stack ADT for infix to postfix conversion

```
//PROGRAM FOR INFIX TO POSTFIX CONVERSION
#include<iostream>
#include<string.h>
using namespace std;
template<class T>
class stack
{
int size,top;
T *a;
public:
stack(int);
int
isEmpty();
int isFull();
```

```
void push(T);
T pop();
T
getNext(T*,int);
int isOperator(T);
int ICP(T);
int ISP(T);
};
template<class T>
T stack<T>::getNext(T w[], int i)
{
return w[i];
}
template<class T>
int stack<T>::isOperator(T x)
{
int i;
if(x=='+'||x=='-'||x=='*'||x=='/'||x=='^'||x=='(') return
1;
else
return 0;
}
template<class T>
int stack<T>::ISP(T
y)
{
if(y=='^')
        return 3;
else if(y=='*'||y=='/')
```

```
        return 2;
else if(y=='+'||y=='-')
        return 1;
else if(y=='(')
        return 0;
}
template<class T>
int stack<T>::ICP(T
z)
{
if(z=='^'||z=='(')
        return 4; else
if(z=='*'||z=='/')
        return 2;
else if(z=='+'||z=='-')
        return 1;
}
template<class T>
stack<T>::stack(int s)
{
size=s;
a= new
T[size]; top=-
1;
}
template<class T>
int stack<T>::isEmpty()
{
if(top==-
```

```
1) return
1; else
return 0;
}
template<class T>
int
stack<T>::isFull()
{
if(top==size-1)
return 1;
else
return 0;
}
template<class T>
void stack<T>::push(T item)
{
if(!isFull())
a[++top]=item;
}
template<class T>
T stack<T>::pop()
{
if(!isEmpty())
return a[top--];
}
main()
{
stack<char>S(10)
; char c[20],ch,t;
```

```
int i=0,l;
cout<<"Enter the INFIX expression ended with #"<<endl;
cin>>c;
ch=S.getNext(c,i++);
while(ch !='#')
{
        if(ch==')')
        {
                ch=S.pop();
                while(ch != '(')
                {
                cout<<ch;
                ch=S.pop();
                }
        }
        else if(S.isOperator(ch))
                {
                t=S.pop();
                if(S.ICP(ch)>S.ISP(t))
                {
                S.push(t);
                S.push(ch);
                }
                else
                {
                while(S.ICP(ch)<=S.ISP(t))
                {
                cout<<t;
                t=S.pop();
```

```
                }
                S.push(t);
                S.push(ch);
                }
                }
        else
                cout<<ch;

ch=S.getNext(c,i++);
}
                while(!S.isEmpty())
                cout<<S.pop();
cout<<endl;
return 0;
}
```

Q. Implement Post fix Evaluation

```
#include<iostream>
#include<string.h>
using namespace std;
template<class T>
class stack
{
int size,top;
T *a;
public:
stack(int);
int
isEmpty();
int isFull();
```

```
void push(T);
T pop();
T
getNext(T*,int);
int isOperator(T);
T eval(T,T,T);
};
template<class T>
T stack<T>::eval(T a,T b,T c)
{
T t;
switch(b)
{
case'+': t=a+c;
        return t;
        break;
case'-': t=a-c;
        return t;
        break;
case'*': t=a*c;
        return t;
        break;
case'/': t=a/c;
        return t;
} }
template<class T>
T stack<T>::getNext(T w[], int i)
{
return w[i];
```

```
}
template<class T>
int stack<T>::isOperator(T x)
{
int i;
if(x=='+'||x=='-'||x=='*'||x=='/'||x=='^'||x=='(') return
1;
else
return 0;
}
template<class T>
stack<T>::stack(int s)
{
size=s;
a= new
T[size]; top=-
1;
}
template<class T>
int stack<T>::isEmpty()
{
if(top==-
1) return
1; else
return 0;
}
template<class T>
int
stack<T>::isFull()
```

```
{
if(top==size-1)
return 1;
else
return 0;
}
template<class T>
void stack<T>::push(T item)
{
if(!isFull())
a[++top]=item;
}
template<class T>
T stack<T>::pop()
{
if(!isEmpty())
return a[top--];
}
main()
{
stack<char>S(10);
char c[20],ch,op1,op2,op3;
int i=0;
cout<<"Enter the POSTFIX expression ended with #"<<endl;
cin>>c;
ch=S.getNext(c,i++);
while(ch !='#')
{
        if(S.isOperator(ch))
```

```
        {
        op2=S.pop();
        op1=S.pop();
        op3=S.eval(op1,ch,op2);
        S.push(op3);
        }
        else
        {
        ch=ch-'0';
        S.push(ch);
        }
ch=S.getNext(c,i++);
}
        if(ch=='#')
        {
        op3=S.pop();
        op3=op3+'0';
        cout<<op3<<endl;
        }
return 0;
}
```

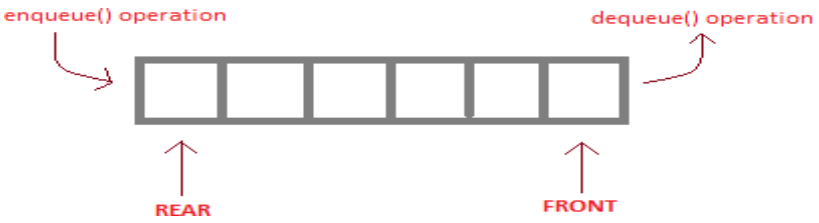
QUEUE

It's a linear Data Structure in which insertion is done at rear end and deletion at front end. The operations are performed in First In First Out order [**FIFO**]. This can be implemented by using array or by linked list. The array representation uses static memory allocation method, so they are limited in size i.e. they cannot shrink or expand. The linkedlist representation uses dynamic memory management method, so they can shrink or expand.

The class declaration for queue using array is represented as:

```
class queue
{
```

```
        int a[10], f, r;
        public:
        queue()
        {
                f=-1;
                r=-1;
        }
        void enque(int);
        void deque();
};
```



The operations performed on queue are:

1. **Insertion or Enqueue:** This function is used to store data into the Queue. The operation perform the following steps:
    - **i.** Check whether Queue is full or not.
    - **ii.** If full, print a message that queue is full.
    - **iii.** If not full, then increment rear by 1 and to that location new data is inserted.

```
void queue :: enqueue(int d)
{
        if(r>=size-1)
                cout<<"Queue is full";
        else
                a[++r]=d;
}
```

2. **Deletion or Dequeue:** This function is used to remove data from the Queue. The Dqueue function performs the following steps:
   - **i.** Check whether Queue is empty or not.
   - **ii.** If empty, message that Queue is empty.
   - **iii.** If not empty, print the element represented by the front location and increment the front by 1.

```
void queue :: dequeue
{
        if((f==-1) || (front>rear))
                cout<<"Queue is empty";
        else
                cout<<a[f++];
}
```

Q. Implement Queue ADT using array

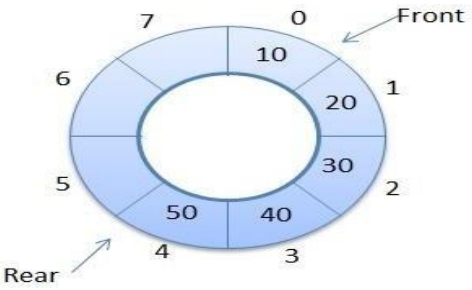
```
// Queue implementation using array
#include<iostream>
using namespace std;
template<class T>
class queue
{
int size,f,r;
T *a;
public:
queue(int);
int
isEmpty();
int isFull();
void enqueue(T);
void dequeue();
};
template<class T>
queue<T>::queue(int s)
{
size=s;
a=new
T[size]; f=-1;
r=-1;
};
template<class T>
int queue<T>::isFull()
{
```

```
if(r==size-1)
return 1;
else
return 0;
}
template<class T>
int queue<T>::isEmpty()
{
if(f==r)
return 1;
else
return 0;
}
template<class T>
void queue<T>::enqueue(T x)
{
if(isFull())
cout<<"Queue is Full"<<endl;
else
a[++r]=x;
}
template<class T>
void queue<T>::dequeue()
{
if(isEmpty())
cout<<"Queue is Empty"<<endl;
else
cout<<a[++f]<<endl;
}
```

```
int main()
{
queue<int> Q(5);
int y=1,op,t;
while(y==1)
{
cout<<"Enter 1.Enqueue 2.Dequeue 3.Exit"<<endl;
cin>>op;
switch(op)
{
case 1:  cout<<"Enter the Data"<<endl;
         cin>>t;
         Q.enqueue(t);
         break;
case 2:  Q.dequeue();
         break;
case 3:  y=0;
} }
return 0;
}
```

CIRCULAR QUEUE

It is a linear Data Structure in which operations are performed in **FIFO** order and last position is connected back to the first position to make a circle. A Circular Queue is also called as ring buffer. In a normal Queue it is possible to insert elements until queue becomes full, but once queue become full, it is impossible to add next data even if there is a space in front of queue. This situation can be overridden in Circular Queue.


0
Front
1
2
3
4
Rear
5
6
7
10
20
30
40
50

Operations on Circular Queue

1. **Enqueue:** This function is used to insert an element to the circular queue. A new data is inserted at the rear position. Steps followed for an Enqueue:
   - **i.** Check whether Queue is full or not.
   - **ii.** If queue is full, display message that queue is full.
   - **iii.** If not full, front is set to zero, rear end is moded with the circular queue size after rear is incremented by 1 and then add the element to the location.

Enqueue function can be defined as:

```
void queue :: enqueue
{
        if((isfull())
                cout<<"Queue is full";
        else
        {
                if(f==-1)
                        f=0;
                        r=(r+1)%s;
                        a[r]=d;
         }
}
```

2. **Dequeue:** This function is used to delete an element from the circular queue. In circular queue elements are always deleted from different positions. Steps followed for a Dequeue are:
   - **i.** Check whether Queue is empty.
   - **ii.** If empty, give a message queue is empty.
   - **iii.** If not empty, the element at that location is displayed. Then front end and rear end is set to -1 if front is equal to rear otherwise front is moded by the queue size after front is incremented by 1.

Dequeue function can be defined as:

```
void queue :: dequeue
{
        if((isempty())
                cout<<"Queue is empty";
        else
        {
                cout<<a[f];
                if(f==r)
                {
```

```
                f=-1;
                r=-1;
        }
        else
                f=(f+1)%s;
} }
```

Implement Circular Queue ADT:

```
//Circular Queue implementation
#include<iostream>
using namespace std;
template<class T>
class queue
{
int size,f,r;
T *a;
public:
queue(int);
int
isEmpty();
int isFull();
void enqueue(T);
void dequeue();
};
template<class T>
queue<T>::queue(int s)
{
size=s;
a=new
T[size]; f=-1;
r=-1;
}
template<class T>
int queue<T>::isEmpty()
```

```
{
if(f>r||f==-1)
return 1;
else
return 0;
}
template<class T>
int
queue<T>::isFull()
{
if((r+1)%size==f)
return 1;
else
return 0;
}
template<class T>
void queue<T>::enqueue(T x)
{
if(isFull())
cout<<"Queue is Full"<<endl;
else
{
if(f==-1)
{
f=0;
r=0;
a[r]=x;
```

```
}
else
{
r=(r+1)%size;
a[r]=x;
}}}
template<class T>
void queue<T>::dequeue()
{
if(isEmpty())
cout<<"Queue is Empty"<<endl;
else
{
cout<<a[f]<<endl;
if(f==r)
{
f=-1;
r=-1;
}
else
f=(f+1)%size
;
} }
int main()
{
queue<int> Q(5);
int y=1,op,t;
```

```
while(y==1)
{
cout<<"Enter 1.Enqueue 2.Dequeue 3.Exit"<<endl;
cin>>op;
switch(op)
{
case 1: cout<<"Enter the data"<<endl;
        cin>>t;
        Q.enqueue(t);
        break;
case 2:  Q.dequeue();
        break;
case 3:  y=0;
} }
return 0;
}
```

DOUBLE ENDED QUEUE

Dequeue defines a data structure where elements can beaded or deleted at either at front end or rear end but no changes can be made elsewhere. It support both Stack like and Queue like capability. A Dequeue can be implemented as either by using an array or by using a linked list.

The operations associated with a double ended queue are:

1. **Enqueue Front:** This function adds element at the end of queue.

2. **Enqueue Rear:** This function adds element at the rear end of the queue.
3. **Dequeue Front:** This function delete element from the front end of the queue.
4. **Dequeue Rear:** This function delete element from the rear end of the queue.

For Stack implementation using this queue, the functions enqueue front and dequeue front are used as push and pop functions respectively. A Dequeue is useful where data to be stored has to be ordered, compact storage is needed and retrieval of data has to be faster.

Variations of Dequeue

1. **Input restricted Double Ended Queue: -** These types of Double ended queues allow insertions only at one end.
2. **Output restricted Double Ended Queue: -** These types of Double ended queues allow deletion from only one end.

PRIORITY QUEUES

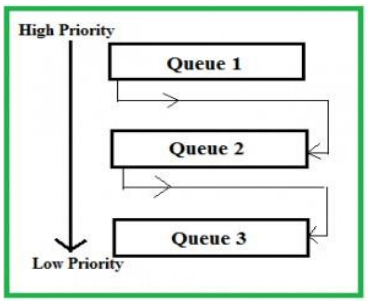
A priority Queue is a collection of a finite number of prioritised elements. Elements can be inserted in any order in the priority queue but when the element is removed, it is always the element with highest priority.

The following rules are applied to maintain the priority queue:

1. Element with highest priority is processed before any element of lower priority.
2. If there where elements with same priority, elements added first in the queue would get processed first.

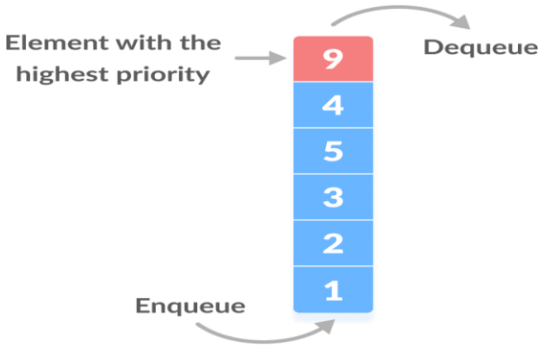
There are two ways to implement priority queue:

1. **Implement separate queues for each priority**: - In this case elements are removed from front of the queue. Elements of the second queue are removed only when the first queue is empty. And elements from third queue are removed only when the second queue is empty.



2. **Implement by using a Structure or by a Class for the queue**: -Here each element in the queue has a data part and priority part of the element. The highest priority element is stored

at the front of the queue and lowest priority element at the rear.



APPLICATION S OF QUEUE

Task Scheduling: Queues can be used to schedule tasks based on priority or the order in which they were received.

Resource Allocation: Queues can be used to manage and allocate resources, such as printers or CPU processing time.

Batch Processing: Queues can be used to handle batch processing jobs, such as data analysis or image rendering.

Message Buffering: Queues can be used to buffer messages in communication systems, such as message queues in messaging systems or buffers in computer networks.

Event Handling: Queues can be used to handle events in event-driven systems, such as GUI applications or simulation systems.

Traffic Management: Queues can be used to manage traffic flow in transportation systems, such as airport control systems or road networks.

Operating systems: Operating systems often use queues to manage processes and resources. For example, a process scheduler might use a queue to manage the order in which processes are executed.

Network protocols: Network protocols like TCP and UDP use queues to manage packets that are transmitted over the network. Queues can help to ensure that packets are delivered in the correct order and at the appropriate rate.

Printer queues : In printing systems, queues are used to manage the order in which print jobs are processed. Jobs are added to the queue as they are submitted, and the printer processes them in the order they were received.

Web servers: Web servers use queues to manage incoming requests from clients. Requests are added to the queue as they are received, and they are processed by the server in the order they were received.

Breadth-first search algorithm: The breadth-first search algorithm uses a queue to explore nodes in a graph level-by-level. The algorithm starts at a given node, adds its neighbors to the queue, and then processes each neighbor in turn.

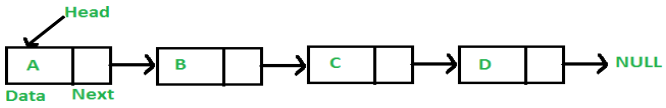
UNIT 3

LINKED LIST

It is an ordered collection of data in which each element (node) contains data and links i.e. address to next element. The first node of the linked list is called head node which indicates the beginning of the linked list. The end node is called a tail node whose link will be assigned to null to indicate the end of linked list.

The operations that can be performed on a linked list are:

1. **Insertion of a new node**
2. **Deletion of a node**
3. **Traversal**



The class structure for a node can be defined as:

```
class node
{
        int data;
        node * next;
};
```

There are two types of linked list:

1. Singly linked list
2. Doubly linked list

Implement Singly Linked list ADT:

```
#include<iostream> using
namespace std; class node
{
public:
int data; node
*next;};


class ll:public node
        {
        node*head,*tail; public:
        ll()
        {
```

```
        head=NULL;
        tail=NULL;
        }
                void create(); void
                insert(); void disp();
                void del();
        };

void ll::create()
                {
                node *temp; temp=new
                node; int n;
                cout<<"Enter the data"; cin>>n;
                temp->data=n; temp-
                >next=NULL;
                        if(head==NULL)
                        {
                        head=temp; tail=head;
                        }
                        else
                        {
                        tail->next=temp; tail=temp;
                        }
                        }
void ll::insert()
{
node *prev,*cur;
prev=NULL; cur=head;
int count=1,pos,ch,n; node
*temp;
temp=new node;
cout<<"Enter the data"<<endl; cin>>n;
temp->data=n; temp-
>next=NULL;
cout<<"Enter 1.Insert as First node 2. Insert as Tail node 3. in between node"<<endl;
cin>>ch; switch(ch)
{
case 1:
        temp->next=head;
        head=temp; break;
case 2:
        tail->next=temp;
        tail=temp; break;
case 3:
        cout<<"Enter the position"<<endl; cin>>pos;
        while(count!=pos)
        {
        prev=cur;
```

```
        cur=cur->next; count++;
        }
        if(count==pos)
        {
        temp->next=cur; prev-
        >next=temp;
        }
        else
        cout<<"Unable to Insert";
}
}

void ll::del()
{
node *prev,*cur;
prev=NULL; cur=head;
int count=1,pos,ch,n;
cout<<"Enter 1.Delete First node 2. Delete Tail node 3. Delete in between node"<<endl;
cin>>ch; switch(ch)
{
case 1:
        if(head!=NULL)
        {
        cout<<"Data deleted is "<<head->data<<endl; head=head->next;
        }
        else
        cout<<"Unable to Detete"<<endl; break;
case 2:
        while(cur!=tail)
        {
        prev=cur; cur=cur-
        >next;
        }
        if(cur==tail)
        {
        cout<<"Data deleted is "<<cur->data<<endl; prev-
        >next=NULL;
        tail=prev;
        }
        else
        cout<<"Unable to Detete"<<endl; break;
case 3:
        cout<<"Enter the position"<<endl; cin>>pos;
        while(count!=pos)
        {
        prev=cur; cur=cur-
        >next; count++;
        }
```

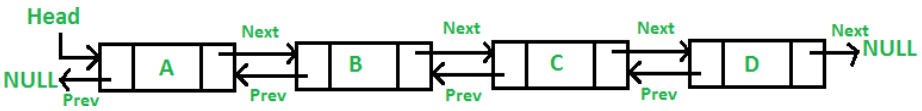
```
        if(count==pos)
        {
        cout<<"Data deleted is "<<cur->data<<endl; prev->next=cur-
        >next;
        }
        else
        cout<<"Unable to Detete"<<endl;
}
}


void ll::disp()
{
node *temp;
temp=head;
if(temp==NULL)
cout<<"EmptyList"<<endl; while(temp!=NULL)
{
cout<<temp->data<<"";
temp=temp->next;
}

}
int main()
{
ll L;
intop,y=1;
while(y==1)
{
cout<<"Enter 1.Create 2.Insert 3.Delete 4.Display 5.Exit"<<endl; switch(op)
{
case 1: L.create();
                break; case
2: L.insert();
                break; case
3: L.delete();
                break;
case 4: L.disp();
                break;
case 5: y=0;
                break;
default: cout<<"Invalid Option"<<endl;
}
return 0;
}
```

DOUBLY LINKED LIST (DLL)

In a single linked list each node provides information about where the next node is located. It has no knowledge about where the previous node is located. This causes difficulty to access $(i-1)^{th}$ or $(i-2)^{th}$ nodes from i^{th} node. In order to access such node it has to be traverse from head node. For handling such difficulties, doubly linked list (**DLL**) is introduced where each node contains two links, one to its predecessor and the other to its successor.



The node class for the DLL can be declared as:

```
class node
{
        int data;
        node * next;
        node * prev;
};
```

For inserting a new node in DLL the following steps has to be followed: Step 1: - node1 □ next = temp

Step 2: - node2 □ prev = temp

Step 3: - temp □ next = node2

Step 4: - temp □ prev = node1

For deleting a node from doubly linked list:

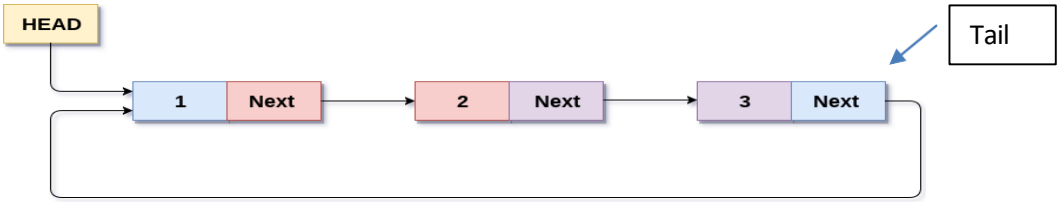
Step 1: - (cur □ prev) □ next = cur □ next

Step 2: - (cur □ next) □ prev = cur □ prev

CIRCULAR LINKED LIST

In circular linked list the last mode contains the address of first mode instead of null. This change will make last node point to first node of the list.

In this case of linked list, from any node of list it is possible to research any other node in the case and it keeps traversal procedure an unending one. Circular linked list is usually used for memory management.



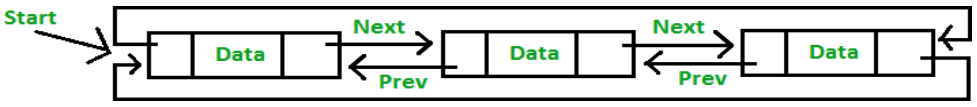
Circular Singly Linked List

DOUBLY CIRCULAR LINKED LIST

In doubly linked list the last node link is set to the first node of the list and first node link, previous link is set to the last node of the list.

Tail □next = head

Head □prev = tail



APPLICATION OF LINKED LIST

- Implementation of stacks and queues
- Implementation of graphs: Adjacency list representation of graphs is the most popular which uses a linked list to store adjacent vertices.
- Dynamic memory allocation: We use a linked list of free blocks.
- Maintaining a directory of names
- Performing arithmetic operations on long integers
- Manipulation of polynomials by storing constants in the node of the linked list
- Representing sparse matrices.