Searching operation scans an existing list to learn whether a number is in it. We implement this operation with the Boolean method isInList().

-The method uses a temporary variable tmp.

-List starting from the head node

-The number stored in each is compared to the number being sought.

-Otherwise, tmp is updated to tmp.next.

-After reaching the last node and executing tmp=tmp.next, tmp becomes null which indicates number is not in the list

- If tmp is not null number is found and isInList() is returns true, Otherwise isInList() returns false.

<u>Defining a node class of a singly linked list: -</u>

```
class Node
{
	int data;
	Node next;
	Node(int data)
	{
		this.data=data;
		next=null;
	}
}
```

Doubly Linked List

A doubly link list is a data structure in which every node in the list has two reference field.

1. **One to the successor.**
2. **Another to the predecessor**

A doubly linked list can be traversed in both direction

<u>Function to add a node at the beginning of DLL:-</u>

```
void add_first
{
        Node  newnode=new Node(10);
        newnode.next=head;
        head.previous=newnode;
        head=newnode;
}
```

<u>Funciton to add node at the end of DLL:-</u>

```
void add_first
{
        Node  newnode=new Node(10);
        Node  tmp=tail;
        newnode.next=null;
        newnode.previous=tmp;
        tmp.next=newnode;
        tail=newnode;

}
```

<u>Function to add node at any point in DLL</u>

```
class  addingatanyposition
{
        Node  newnode=new Node(100);
        Node  current=null;
        if(current==tail)
        {
                newnode  next=null;
                tail=newnode;

        }
        else
        {
                current.next=newnode.next;
                current.next.previous=newnode.next;
        }
        current.next=newnode;
        newnode.previous=current;
}
```

Funciton to delete 1st node of DLL

```
class deleteingfrom1st
{
        if(isEmpty())
        {
                return 0;
        }
        if(head==tail)
        {
                head=tail=null;
        }
        else
        {
                head=head.next;
                head.previous=null;
        }


}
```

Funciton to delete Last node of DLL

```
class deleteingfromLast
{
        if(isEmpty())
        {
                return 0;
        }
        if(head==tail)
        {
                head=tail=null;
        }
        else
        {
                tail=tail.previous;
                tail.next=null;
        }


}
```

Funciton to delete node after any position of DLL

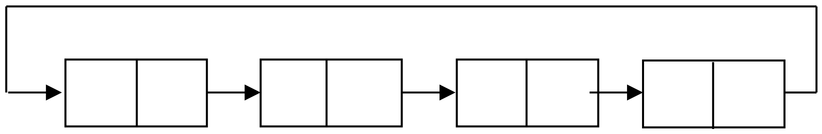
Circular Linked List

A circular linked list is a data structure in which every node has a successor.

Circular linked list is used where we need nodes to form a ring.

Types of circular linked list:-

1. **Circular singly linked list**
2. **Circular doubly linked list**



Function to add a node at beginning of CSLL

```
class insert_at_front()
{
        Node newnode=new Node(2);
        if(isEmpty())
        {
                head=tail=newnode;
        }
        else
        {
                newnode.next=head;
                tail.next=newnode;
                head=newnode;
        }
}
```

Function to add a node at last of CSLL

```
class insert_at_last()
{
        Node newnode=new Node(5);
        if(isEmpty())
        {
                head=tail=newnode;
        }
        else
        {
                tail.next=newnode;
                newnode.next=head;
                tail=newnode;
        }
}
```

Function to delete first node of CSLL

```
class delete_first()
{
        if(head==tail)
                head=tail=null;
        else
        {
                head=head.next;
                tail.next=head;
        }
}
```

Function to delete first node of CSLL

```
class delete_last()
{
        if(head==tail)
                head=tail=null;
        else
        {
                Node tmp=head;
                While(tmp.next!=tail)
                {
                        tmp=tmp.next;
                }
                tail=tmp;
                tail.next=head;
        }
}
```

Circular doubly linked list

Function to add a node at beginning of CDLL

```
class insertatfirst
{
        Node newnode=new Node(6);
        if(isEmpty())
        {
                head=tail=newnode;
        }
        else
        {
                newnode.next=head;
                head.previous=newnode;
                newnode.previous=tail;
                tail.next=newnode;
                head=newnode;
        }
}
```

Function to add a node at last of CDLL

```
class insertatlast
{
        Node newnode=new Node(6);
        if(isEmpty())
        {
                head=tail=newnode;
        }
        else
        {
                newnode.next=head;
                head.previous=newnode;
                newnode.previous=tail;
                tail.next=newnode;
                tail=newnode;
        }
}
```

Function to delete a node at begining of CDLL

```
class deletefromlst
{
        if(isEmpty())
        {
                return 0;
        }
        if(head=tail)
        {
                head=tail=null;
        }
        else
        {
                head=head.next;
                head.previous=tail;
                tail.next=head;
        }
}
```

Skip List

A skip list is a variant of ordered linked list that makes a non-sequential search possible.

Self organizing list

A list can be organized by organized following four ways

1. Move to Front method.

   After the desired element is located, put it at the beginning of the list.

2. Transpose Method

   After the desired element is located, swap it with its predecessor unless it is at the head of the list.

3. Count Method

4. Ordering Method

   Order the list using certain criteria.

Unit 3: Stack & Queue

Stack is a linear data structure that can be accessed only at one of its ends for storing and retrieving data.

Stack is a LIFO structure,

E.g.: - piles of trays in cafeteria.

Operations of stack

1. clear(): clears the stack.
2. isEmpty(): check to see if stack is empty.
3. isFull(): check to see if stack is Full.
4. push(el): put an element el at the top of stack.
5. pop(): take the topmost element of a stack.
6. topEl(): return the topmost element without removing it.

Push operation

=> if stack is full, print overflow

Algorithm:

1. If stack is full, print overflow
2. set top= top+1,
3. stack[top]=item,
4. return

Function/method:

```
Void push(int stack[],int top,int item,int sizes)
{
        if(top==size-1)
        {
                System.out.println("Stack overflow");
        }
        else
        {
                top=top+1;
                stack[top]=item;
        }
}
```

Pop operation

=> if stack is empty, print underflow

Algorithm:

1. **If stack is empty, print underflow**
2. **Item=stack[top];**
3. **top=top-1;**
4. **return**

Function/method:

```
Void pop(int stack[],int top,int item,int sizes)
{
        if(top==-1)
        {
                System.out.println("Stack underflow");
        }
        else
        {
                item=stack[top];
                top=top-1;
        }
}
```

Application of stack

1. **Delimiter matching**

Algorithm:

1. **Delimeter_matching(file)**
2. **while not end of file**
3. **if ch is '(','[', or '{'**
4. **push(ch);**
5. **else if ch is '/'**
6. **read next character**
7. **if this character is '*'**
8. **skip all characters until '*/' is found and report an error if the end of file is reached before '*/' is encountered;**
9. **else ch=the character read in ;**
10. **continue;**
11. **else if ch is ')','[', or '}'**
12. **if ch and poped of delimeter do not match;**
13. **failure**
14. **else**
15. **read next character ch from file;**
16. **if stack is empty**

17. success;
18. else
19. failure;

2. Adding two large numbers

Algorithm:

```
1. Adding large numbers ()
2. Read the numerical of the first number and store the numbers corresponding to them on
   one stack;
3. Read the numerical of the second number and store the numbers corresponding to them
   on another stack
4. Result=0;
5. While at least one stack is not empty
6. Pop a number from each non-empty stack and add them to result;
7. Push the unit part on the result stack;
8. Store carry in result;
9. Push carry on the result stack if it is not zero
10. Pop numbers from the result stack and display them;
```

Queue

- Queue is a linear data structure where operations of insertion and removal are performed at separate ends known as rear and front.
- It is a first in first out structure
- Whenever is new element is added to the queue, rear pointer is used.
- During insertion operation rear is increment by 1. And data is stored in the queue at that location indicated by rear.
- The front pointer is used to remove an element from the queue
- The process of inserting and element at the last of the queue is called enqueue.
- The process of removing the first element of a queue is called dequeuer.

Operations of queue

Clear(); clear the queue

isEmpty(): check to see if queue is empty

isFull();: check to see if queue is Full

enqueue(el):Insert the element at the last of the element

dequeuer()remiove the element from the beginning of the queue

firstEL():

Methods of function to enqueue an element on a queue.

```
Void enqueue(int queue[],int rear,int data,int sizes)
{
        if(isFull())
        {
                System out.println("Overflow");
        }
        Else
        {
                rear++;
                queue[rear]=data;
                size++;
        }

}
```

Methods of function to dequeue an element on a queue.

```
Void dequeue(int queue[],int front,int data,int sizes)
{
        if(isEmpty())
        {
                System out.println("Underflow");
        }
        Else
        {
                front++;
                data=queue[front];
                size--;
        }

}
```

Chapter-1 Complexity Analysis

Data structure and algorithm: Data structure is the way of string data in a computer so it can be used efficiently.

Types of Data Structure:

1) **Linear Data structure: In a linear data structure, data elements are stored in contiguous memory locations. e.g.: Array, stack, queue.**
2) **Linear Data structure: In a non-linear data structure, data elements are stored in non-contiguous memory locations. e.g.: Graph, Tree, linked list.**
3) **Static data structure: A static data structure is one whose size is fixed at creation. e.g.: array**
4) **Dynamic data structure: A dynamic data structure is one whose size is variable at creation. e.g.: Linked list.**

Operation of structure

a) **Traversalling**

Unit 4. Recursion

Recursion is the ability of a function defining an object in terms of itself.

It is a process by which a function calls itself repeatedly until some condition has been satisfied.

For problems to be solved recursively two conditions must be satisfied

1) **The problem must be expressed in recursive form.**
2) **The problem statement must include a terminating condition.**

Recursive definition consists of two parts.

1) **Anchor or ground case: The basic elements that are building blocks of the other elements of the set are listed in anchor case.**
2) **Recursive or inductive case: In this case rules are given that allow for the construction of new object.**

   e.g. calculating factorial

```
        n!=1, if(n<=1)       -- Ground case
          =n*(n-1), if(n>1)    --Inductive/Recursive case
```

Function-- Factorial

```
int factorial (int n)
{
        if(n<=1)
                return 1;
        else
                return n*factorial(n-1);
}
```

Function-- Fibonacci

```
Fib(n)=n, if n<2
        =fib(n-1) +fib(n-2), if n>=2
```

```
int fib (int n)
{
        if(n<2)
        return n;
        else
                return fib(n-1) +fib(n-2)
}
```

The product of two positive integers

The product of two positive integers a*b, where a and b are two positive integers, may be defined recursively as a added b times.

Function--

```
a*b=a          if,  b=1
   =a+ a*(b-1)  , b>1
```

```
int product (int a, int b)
{
        if(b==1)
                return a;
        else
                return(a+product(a,b-1));
}
```

The function that defines any number X to a non-negative integer power n is recursively defined as

Function--

$X^n = 1$, if, $n=0$
$\quad = X * X^{n-1}$, $n>0$

```
int power (int x, int n)
{
        if(n==0)
                return 1;
        else
                return(x*power(x,n-1));
}
```

Method calls and recursion implementation

What information should be preserved when a method is called.

Activation record

An activation record is a data area containing all local variables, the values of the methods parameters and the return address indicating where to restart its caller.

The activation record is allocated on runtime stack.

The activation record usually contains following information.

- **Values for all the parameters to the method.**
- **The return address to resume, control by the caller.**
- **The dynamic link which is a pointer to the callers' activation record.**
- **The returned value for a method.**

Contents of the runtime stack when main () calls method f1(),

f1() calls method f2() and f2() calls method f3()

Activation record of f3()

Chapter 5. Binary Tree

- The value of left child of a node is less than the value of that node.
- The value of right child of a node is greater than the value of that node.

Implementing a Binary Tree

1) Array Implementation

   Root: i=0
   Left child=2i+1
   Right child=2i+2

   Since, there exists the wastage of memory space while implementing a binary tree with an array, So we implement the binary tree from Linked List structure.

The array implementation of binary tree consumes approximately 2^H memory space where H is height of that binary tree

2) Linked list Implementation

   Defining a node

```
class node
{
        int data;
        node left, right;
        public node()
        {
                data=0;
                left=null;
                right=null;
        }

        public node(int el)
        {
                data=el;
                left=null;
                right=null;
        }

}
```

3) Method for searching a binary search tree

```
Node BST(int el)
{
        Node p=root;
        while(p.el!=el)
        {
                if(el<p.el)
                        p=p.left;
                else
                        p=p.right;
                if(p==null)
                        return null;
        }
        return p;
}
```

Tree traversal

Tree traversal is the process of visiting each node of a tree exactly once.

For a binary tree of n Nodes there exists n! factorial traversal.

Types of Tree traversal

1) Breadth first traversal (BFT)

   It is visiting each node from the highest (or lowest) level and moving down (or up) level by level, visiting nodes on each level from left to right (or from right to left).

   In BFT it is not possible to visit a node at level n+1 before visiting all the nodes of level n

   <u>Method</u>

```
void BFT()
{
        Node p=root;
        Queue queue=new Queue<node>();
        if(p!=null)
        {
                queue.enqueue(p);
                while(!queue.isEmpty())
                {
                        p=queue.dequeue();
```

```
                    visit(p);
                    if(p.left!=null)
                         queue.enqueue(p.left);
                    if(p.right!=null)
                         queue.enqueue(p.right);
               }
          }
     }
```

2) Depth first traversal (DFT)
   Three types of depth first traversal:
   a. Preorder Traversal (VLR)
      To traverse a non-empty binary tree in pre order, we perform three activities
      i. Visit the root node
      ii. Traverse the left subtree in preorder
      iii. Traverse the right subtree in preorder

      Method

```
void preorder(Node p)
{
     visit(p);
     preoder(p.leftchild);
     preoder(p.rightchild);
}
```

   b. In order Traversal (LVR)
      To traverse a non-empty binary tree in in order, we perform three activities
      i. Traverse the left subtree in in order
      ii. Visit the root node
      iii. Traverse the right subtree in in order

      Method

```
void inorder(Node p)
{
     inorder(p.leftchild);
     visit(p);
     inoder(p.rightchild);
}
```

   c. Post Traversal (LRV)
      To traverse a non-empty binary tree in postorder, we perform three activities
      i. Traverse the left subtree in postorder

ii. Traverse the right subtree in post order
iii. Visit the root node

Method

```
void postorder(Node p)
{
      postorder(p.leftchild);
      postoder(p.rightchild);
      visit(p);
}
```

Algorithm for inserting a node in a binary tree

```
if root is null
      create root node;
      return;
if root exists
      compare the data with node.data
      while until the insertion position is located
            if data is greater than node.data
                  goto right subtree;
            else
                  goto left subtree;
      end while
      insert data;
end if
```

Function to insert a node in a binary tree

```
void insert(int el)
{
      Node p=root, prev=null;
      while(p!=null)
      {
            prev=p;
            if(el<p.el)
                  p=p.left;
            else
                  p=p.right;
      }

      if(root==null)
            root=new Node(el);
      else if(el<prev.el)
            prev.left=new Node(el);
      else
            prev.right=new Node(el);
}
```

There are three different ways to delete a node from a binary tree

1. Delete a leaf / terminal node
2. Delete a node with one child
3. Delete a node with two children

   Two approaches of deleting a node with two children

   a. Deletion by merging
   b. Deletion by copying

Balancing a tree

Two arguments that support the trees are follows:

1. Tree is used to represent hierarchical structure of a certain domain and sub process is search is much faster using tree than using linked list.
2. If Tree is not balanced, it is more or less similar to the linked list.

Height Balanced Tree or Balanced Tree

A binary tree is a height balanced tree or balanced tree if the difference in height of both subtrees of any node in the tree is either 0 or 1;

Balancing a binary tree using binary search technique

```
void balance(int arr[],int first,int last)
{
    if(first<=last)
    {
        int mid=(first+last)/2;
        insert(arr[mid]);
        balance(arr,first,mid-1);
        balance(arr,mid+1,last);
    }
}
```

DSW Algorithm

- To avoid sorting procedure while balancing a binary tree, we use DSW algorithm.
- The building block, for tree transformation of this algorithm is rotation.
- There are two rotation: right rotation and left rotation, which are symmetrical to each other.
- After rotating a tree, the tree is transformed into a backbone.
- Then the backbone is finally transformed into a balanced tree.
-

Backbone

The DSW algorithm transforms an arbitrary binary search tree into a linked list like structure, called backbone.

Algorithm for right rotation

The right rotation of the node Ch about its parent Par is performed as follows:

```
Right_rotation(Gr, Par, Ch)
If Par is not the root of the tree
    Grandparent Gr of child Ch becomes Ch's parent;
    Right subtree of Ch becomes left subtree of Ch's parent Par;
    Node Ch acquires Par as its right child;
```

Creating a backbone

```
create_backbone (root)
Tmp=root;
While(tmp!=null)
    If(tmp has a left child)
        Rotate this about the tmp;
        Set tmp to the child that just become parent;
    else
        Set tmp to its right child
```

Now the above backbone is transformed into a tree but this time a balanced tree as follows:

- In each pass down the backbone, every second node down to a certain point is rotated about its parent.

```
createPerfectTree(n)
m=2 Log(n+1) -1;
make n-m rotation starting from top of the backbone;
while(m>1)
m=m/2;
make m rotation starting from top of the backbone;
```

AVL Tree (Admissible Tree)

(Adelson, Velskii, Landies)

- It is a tree in which the highest of left and right subtrees of every node differ by atmost one.
- The technique for balancing AVL trees do not guarantee that the resulting tree is perfectly balanced.
- The numbers in the nodes indicates the balanced factor of the node.
- A balanced factor is a difference between a height of right subtree and left tree of a node.
  Balanced factor=Height of right subtree-Height of left subtree
- For an AVL tree, all balanced factors should be -1, 0 or +1
- If the balanced factor of any node in the AVL tree becomes less than -1 or greater than +1, the tree has to be balanced.
- The minimum number of nodes in an AVL tree of height H is determine by the following recurrence relation.

Balancing a tree after inserting a node in the right subtree of Node Q

Self-Adjusting Tree

Moving a node up in the hierarchy which is accessed frequently

Self re-structuring tree

Two possibilities of self re-structing trees are

1. **Single rotation: Rotate a child about its parent when the element in a child is accessed, unless it is in the root.**
2. **Moving to the root: Repeat child parent rotation until the element being accessed is in the root.**
3.

Splaying

Splaying is the modification of moving to root which applies single rotation in pairs in an order depending on the links between the child, parent & grandparents.

Three cases (where p=grandparent, Q=parent and R=child) are as follows

Case 1: Node R's parent is the root

Case 2: Homogeneous configuration

Node R is the left child of its parent Q and Q us also the left child of its parent P or R and Q are both right children

Case 3: Heterogeneous configuration

Node R is the right child of its parent Q and Q is the left child of its parent P or R is the left child of Q and Q is the right child of P.

Algorithm

```
splaying(p, q, r)
while R is not the Root
    if Rs parent is the root
        perform singular splay, rotate R about its parent ;
    if R is in homogenous configuration with its predeccesor
        perform a homogeneous splay, first rotate Q about P and then R about Q
    else
        perform a heterogeneous splay, first rotate R about Q and then R about P;
```

Heap:

Heap is a particular type of binary tree which has following two properties:

1. **The value of each node is greater than or equal to the values stored on each of its children**
2. **The tree is perfectly balanced and the leaves in the last level are all in the left most position**

Min Heap

If the greater than > sign of 1st property of the max heap is changed to < sign then it is called min heap

1. **The value of each node is greater than or equal to the values stored on each of its children**
2. **The tree is perfectly balanced and the leaves in the last level are all in the left most position**

Heap as Priority Queue

Algorithm for enqueuing

```
Heapenqueue(el)x
        put el at the end of heap;
        while el is not in the root and el>parent(el)
                swap el with its parents;
```

Algorithm for dequeuing

```
Heapdequeue(el)
        Extract the element fromthe root/
        Put the element fromthe last leaf in its place
        Remove the last leaf;
        P=root;
        while p is not a leaf and p< any of its children swap p with its larger child
        swap el with its parents;
```

Organizing Array as Heap Tree:

All the heaps are array but all the array is not heaps. So, we have to convert an array into a heap tree.

Two approaches of converting an array into a heap tree are :

1. **Top down method**
2. **Bottom up method (Floyd Algorithm)**

Top Down method

It Starts with an empty heap and sequentially include elements in a growing heap.

It extends the heap by enqueuing new element in the heap

Bottom up method (Floyd Algorithm)

It is a multiple small heap are formed and repeatedly merged into larger heaps.

Algorithm:

```
FLyodAlgorithm(int data[])
for i= index of the last non-leaf down to 0
    restore heap property for the tree whose root is data[i] by calling
    moveDown (data,i,n-1);
```

Example:

Data[]={2, 8, 6, 1, 10, 15, 3, 12, 11}

N=size of array=9