BINARY EXPRESSION TREE

- expression tree construction algo:
    - Scan the postfix expression till the end, one symbol at a time
    - Create a new node, with symbol as info and left and right link as NULL
    - If symbol is an operand, push address of node to stack
    - If symbol is an operator
        - Pop the address from stack and make it right child of new node
        - Pop the address from stack and make it left child of new node
        - Now push address of new node to stack
    - Finally, stack has only element which is the address of the root of expression tree

```
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
#include<ctype.h>
typedef struct node
{
        char value;
        struct node* left_child, *right_child;
}NODE;
NODE* get_node()
{
        NODE* x;
        x=(NODE*)malloc(sizeof(struct node));
        x->left_child=NULL;
        x->right_child=NULL;
        return x;
}
NODE* create_tree(char postfix[])
{
        //creates postfix tree when postfix expression is given
        //postfix[] is the postfix expression
        NODE temp,s[70];
        char symb;
        int i,top=-1;
        for (i=0;postfix[i]!='\0';i++)
        {
                sym=postfix[i];
                temp=getnode();
                temp->value=symb;
```

```
                  if (isalnum(symb))
                  {
                          s[++top]=temp;
                  }
                  else
                  {
                          temp->right_children=s[top--];
                          temp->left_children=s[top--];
                          s[++top]=temp;
                  }
          }
          return s[top--];//last element in stack will be root,so by poppin it
  returns the tree
  }
```

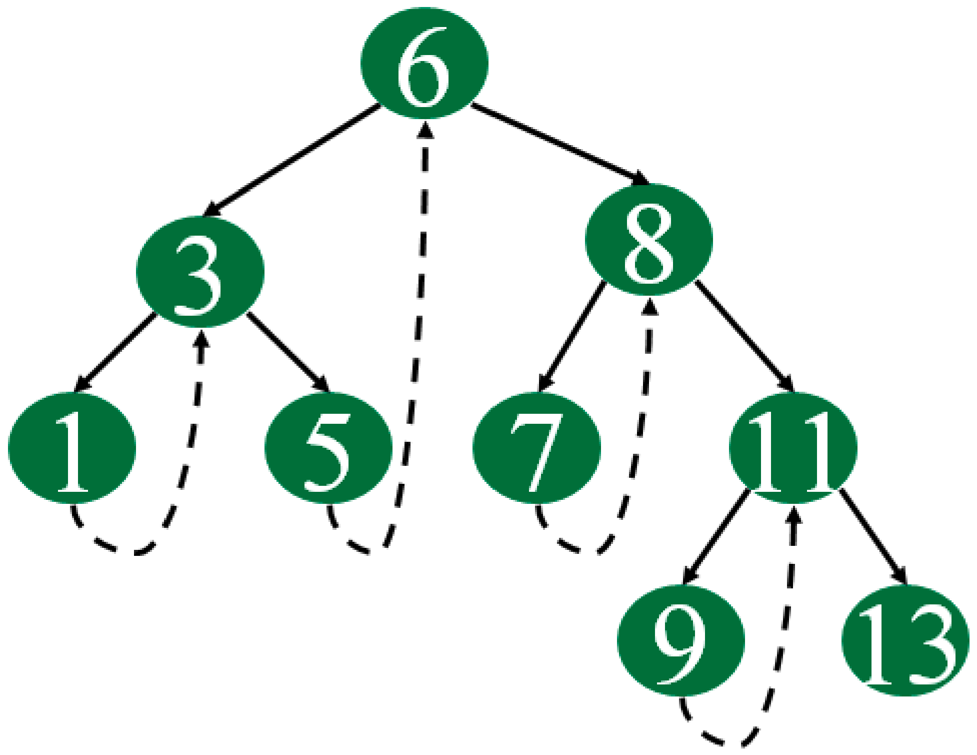
- expression tree evaluation
    - algo :

```
eval(t) // 't' has the address of the root node of expression tree
        if t->data is an operator
                return eval (t->left) t->data eval(t->right)
        return t->data
```

- code:

```
int eval(t)
{
  switch(t->data)
  {
   case '+':
        return eval(t->left)+eval(t->right)
    case '-':
        return eval(t->left)-eval(t->right)
    case '*':
        return eval(t->left)*eval(t->right)
    case '/':
        return eval(t->left)/eval(t->right)
    default:
            return t->data;
  }
}
```

THREADED BINARY SEARCH TREE



- two types of TBST
    - Single threaded BST

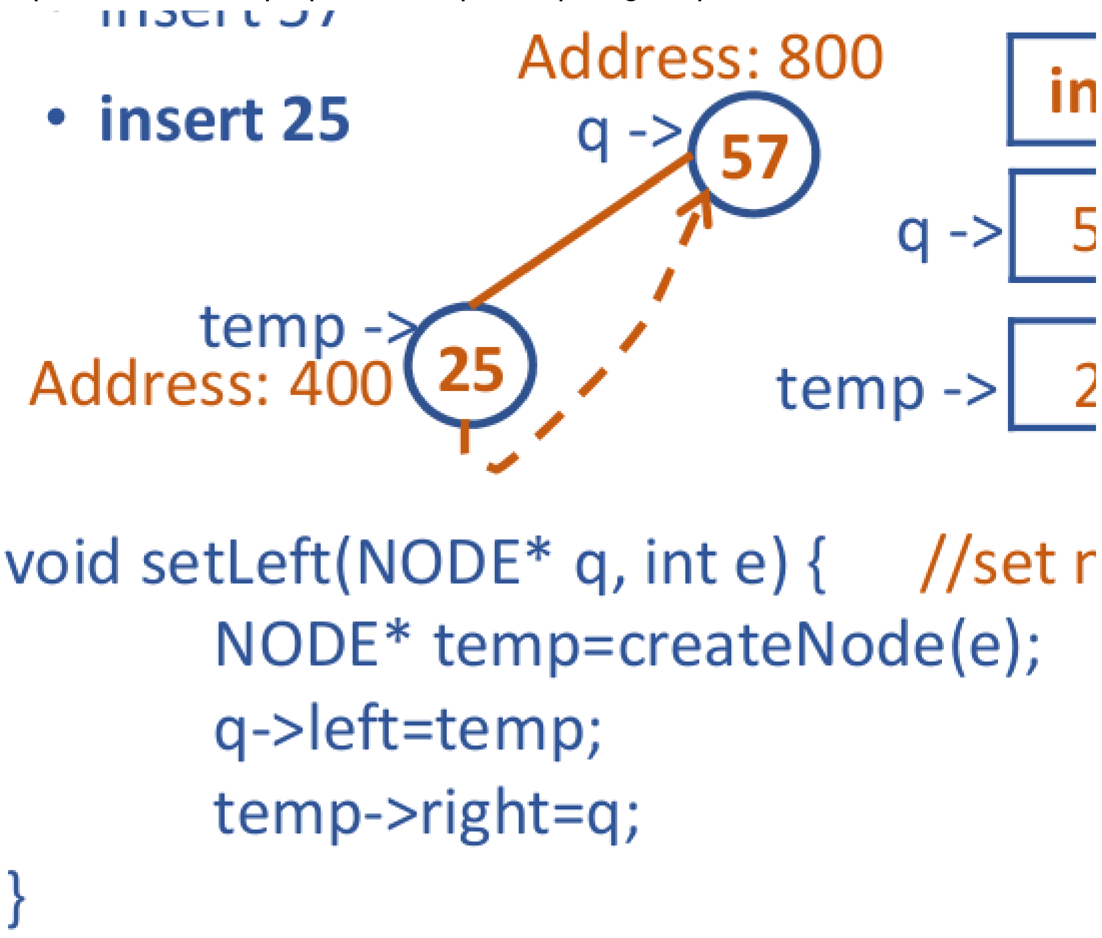
```
struct node
{
   char data;
   struct node *left, *right;
   bool left_thread;
   bool right_thread;
}
```

- Multi threaded BST

```
struct node
{
   char data;
   struct node *left, *right;
   bool left_thread;
   bool right_thread;
}
```

- Iterative Inorder Traversal requires Explicit stack.to eliminate use of stack , structural modification applied to bst to produce tbst

- `right in threaded bst` : use the right pointer of a node to point to the inorder successor if in case it is not pointing to the child.
- `left in threaded bst` : use the left pointer of a node to point to the inorder predecessor if in case it is not pointing to the child.
- `in threaded bst` : when we use both the above pointers
- right threaded bst :
  - insert operation :
    - say temp is new node to be inserted , q is parent of temp
    - temp = createnode(e)
    - if temp is left child of q : q->left=temp ; temp->right=q



```
void setLeft(NODE* q, int e) {      //set r
            NODE* temp=createNode(e);
            q->left=temp;
            temp->right=q;
}
```

    - if temp is right child of q , temp->right=q->right ; q->right = temp ; q->rthread=0
    - insertnode for right bst

```
void setLeft(NODE* q,int e) //set node to left of queue
{
        NODE* temp=createNode(e);
        q->left=temp;
        temp->right=q;
}
void setRight(NODE* q,int e)
{
        NODE* temp=createNode(e);
        temp->right=q->right; //thread
        q->right=temp;
        q->rthread=0;
}
void insert(TREE *pt,int e)
{
        while(p!=NULL)
        {
                q=p;
                if(e<p->info)
                        p=p->left;
                else
                {
                        if(p->rthread)
                                p=NULL;
                        else
                                p=p->right;
                }
        }
        if(e < q->info)
                setLeft(q,e);
        else
                setRight(q,e);
}
```

- inorder traversal :

```
void inOrder(TREE *t)
{
        NODE *p=t->root;
        NODE *q;
        do{
                q=NULL;
                while(p!=NULL)
                {
                        q=p;
                        p=p->left;
                }
                if(q!=NULL)
                {
                        printf("%d ",q->info);
                        p=q->right;
                        while(q->rthread && p!=NULL)
                        {
                                printf("%d ",p->info);
                                q=p;
                                p=p->right;
                        }
                }
        }while(q!=NULL);
}
```

- code for in order bst :

```
NODE* insertNode(NODE *root,int data){
    NODE *t=root,*parent,*nn;
    nn=createNode(data);
    if(t==NULL){ //root node
        nn->lthread=1;
        nn->rthread=1;
        nn->left=NULL;
        nn->right=NULL;
        root=nn;
        return root;
    }
    /*if lthread=0, it means it has a left child
    rthread=0, it means it has a right child*/
    //To determine the parent of the node to be inserted
    while(t!=NULL){
        parent=t;
        if(data<t->data){
            if(t->lthread==0)
                t=t->left;
            else
                break;
        }
        else if(data>t->data){
            if(t->rthread==0)
                t=t->right;
            else
                break;
        }
        else{
            printf("\nDuplicate Key");
            return root;
        }
    }
    if(data<parent->data){//left child
           nn->lthread=1;
           nn->rthread=1;
           nn->data=data;
           nn->left=parent->left;
           nn->right=parent;
           parent->left=nn;
           parent->lthread=0;
        }
    else{//right child
          nn->lthread=1;
          nn->rthread=1;
          nn->data=data;
          nn->left=parent;
          nn->right=parent->right;
          parent->right=nn;
          parent->rthread=0;
        }
    return root;
}
NODE* inorderSuccessor(NODE *t){
    if(t->rthread==1)//if it doesn't has a right child
        return t->right;//inorder successor link is present in the right
link
    //if(t->rthread==0)//if it has a right child
    t=t->right;
    while(t->lthread==0)
        t=t->left;
    return t;
}
void inorder(NODE *root){
    NODE *t=root;
    if(root!=NULL){
        while(t->lthread==0)//t->left!=NULL
            t=t->left;
        while(t!=NULL){
            printf("%d ",t->data);
            t=inorderSuccessor(t);
        }
    }
}
```

HEAP

- binary tree with keys to nodes
- conditions for heap
  - shape : The binary tree is essentially complete, that is, all its levels are full except possibly the last level, where only some rightmost leaves may be missing
  - parental dominance : The key at each node is greater than or equal to the keys at its children.
- properties :
  - i=0 is unoccupied
  - parental nodes present in 1st n/2 positions , leaf nodes in last n/2 positions
  - children of key in position i present in 2i and 2i+1
  - parent of key in position i present in i/2
- code :
- bottom up construction :
  - start from postion n/2 to 0 (these are parental nodes) , for every parental node check with child node nd swap if parental node < child node

```
// in this example position 0 is unused
for(i=n/2;i>=1;i--) //start index from last parent node to first
{
        k=i;
        v=h[k];
        heap=0;
        while(!heap && 2*k<=n) // done so that changing level 0 , changes level 1
        {
                j=2*k;
                if(j<n)
                        if(h[j+1]>h[j])
                                j=j+1; //largest child node
                if(v>h[j]) // if parent > child then its heap
                        heap=1;
                else
                {                // parent < child so swap
                        h[k]=h[j];
                        k=j;
                }
        }
        h[k]=v;
}
```

- top down construction(shiftup)

    - start from position after root node nd iterate till end of array , for every node check if node in current position > parent node , if this is true then swap node in current nd parent position . repeat til condition is true

```
// in this example position 0 has root element
for(i=1;i<n;i++)
 {
         elt=a[i];
         c=i;
         p=(c-1)/2;
         while(c>0 && a[p]<elt)
         {
                 a[c]=a[p];
                 c=p;
                 p=(c-1)/2;
         }
         a[c]=elt;
 }
```

- Ascending Heap: Root will have the lowest element. Each node's data is greater than or equal to its parent's data. It is also called min heap.
- Descending Heap: Root will have the highest element. Each node's data is lesser than or equal to its parent's data. It is also called max heap.
- Ascending Priority Queue: is a collection of items into which items can be inserted arbitrarily and from which only the smallest item can be removed
- Descending Priority Queue(dpq): is a collection of items into which items can be inserted arbitrarily and from which only the largest item can be removed
- dpq can be done with heap as every node greater than children so its like dpq.elements can be inserted randomly nd shift up using top down construction/ bottom up.so root node will be highest priority and can be deleted and rearrangin other nodes will take o(logn) time as logn levels
- insertion also takes logn times

```
Algorithm for siftup (element is iniitally inserted at end and has to be
shifted up depending on its value nd parents value)
c = k;
p = (c-1)/2;
while(c>0 && dpq[p]<elt) {
        dpq[c]=dpq[p];
        c=p;
        p=(c-1)/2;
}
dpq[c]=elt;
```

- pq delete operation : after deleting root node , shifts largest child of root node to root node positions nd adjusts its sub tree of that node accordinly

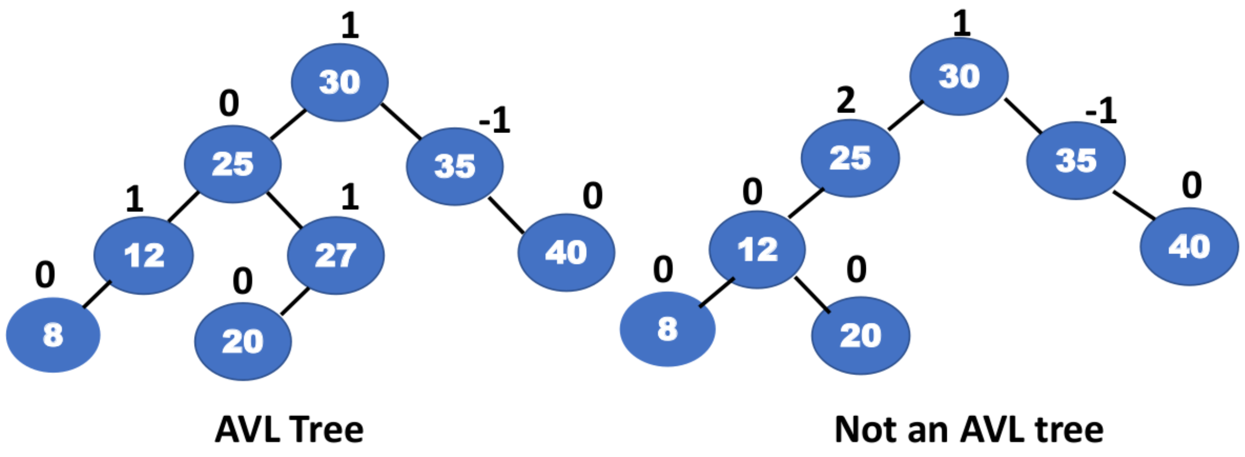
```
Algorithm  largechild(p,m)
        c = 2*p+1;
        if(c+1 <= m && x[c] < x[c+1])
                c=c+1;
        if(c > m)
                return -1;
        else
                return (c);
Algorithm adjustheap(root,k) // recursive version
        p = root;
        c = largechild(p,k-1);
        if(c >= 0 && dpq[k] < dpq[c]){
                dpq[p] = dpq[c];
                adjustheap(c,k);
        }
        else
                dpq[p] = dpq[k];

Algorithm adjustheap(root,k) //Iterative version
        p = root;
        kvalue = dpq[k];
        c = largechild(p,k-1);
        while(c >= 0 && kvalue < dpq[c]){
                dpq[p] = dpq[c];
                p = c;
                c = largechild(p,k-1);
        }
        dpq[p] = kvalue;
```

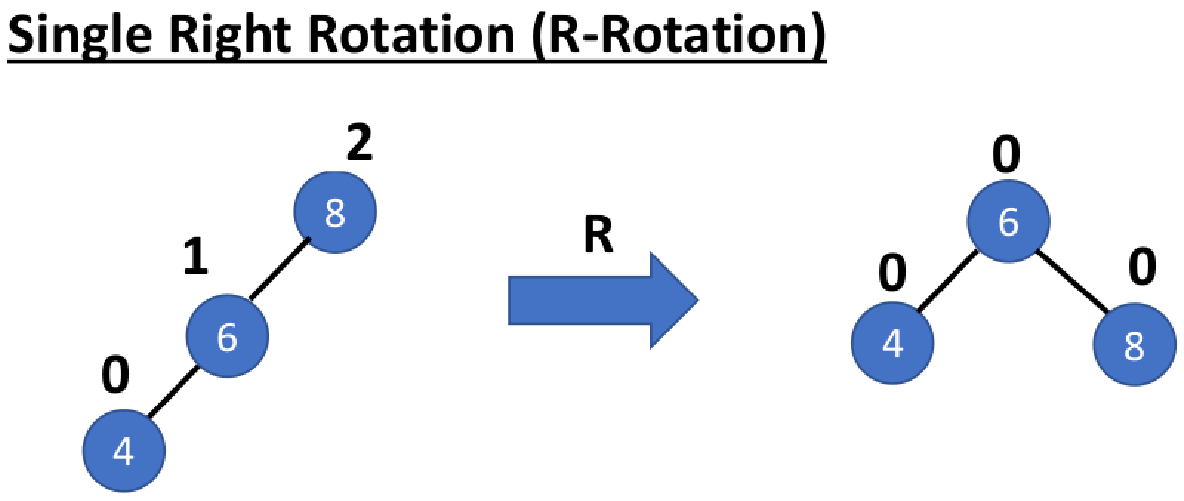
BALANCED TREES - AVL

- search nd insertion nd algo doesnt ensure tree is balanced.so if values greater we can keep inserting at one end and tree height becomes n so time for most operations worst case becomes o(n)
- we can construct balanced trees so that height becomes logn so worst case for operations become o(logn) which is less than o(n)

- AVL tree is a binary search tree in which, for every node, the difference between the heights of the left and right subtrees, called the balance factor is either 0 or +1 or -1
- Balance Factor = Height(left subtree) – Height(right subtree)

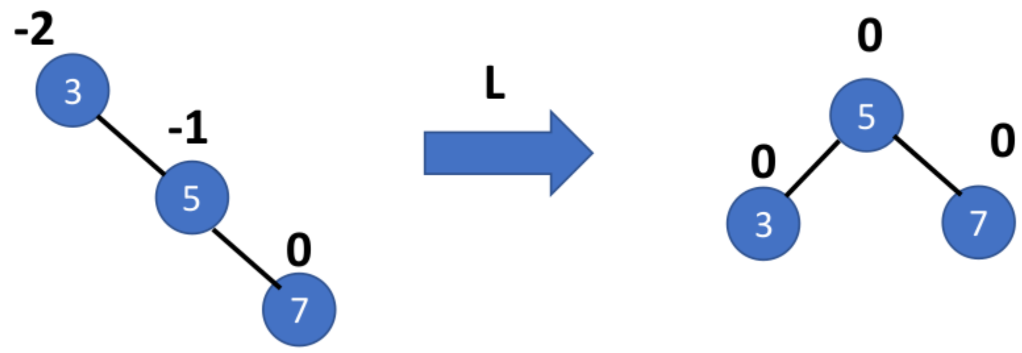


-
- Rotation in a AVL tree is a local transformation of its subtree rooted at a node whose balance has become either +2 or -2
- In case there are several such nodes, The rotation is always performed for a subtree rooted at “unbalanced" node closest to the newly inserted leaf node.
- note that if new node is added to a current nodes right, balance factor of newnode -=1
- if new node is added to current nodes left balance factor of newnode +=1

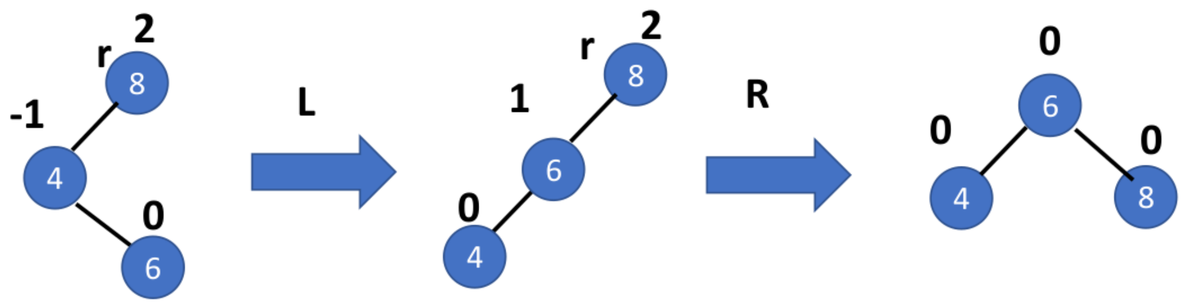


-

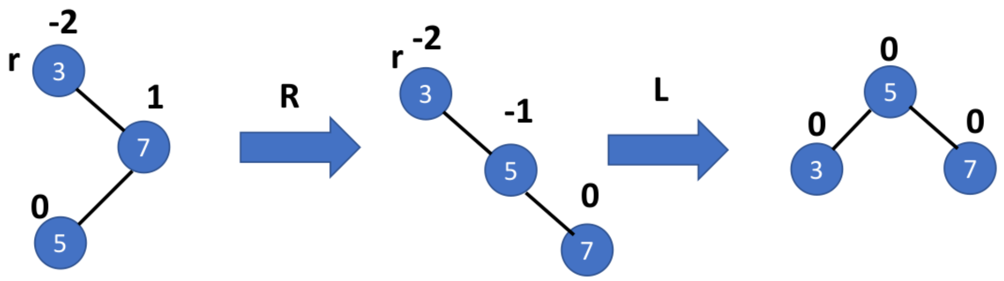
Single Left Rotation:



Double Left Right Rotation:

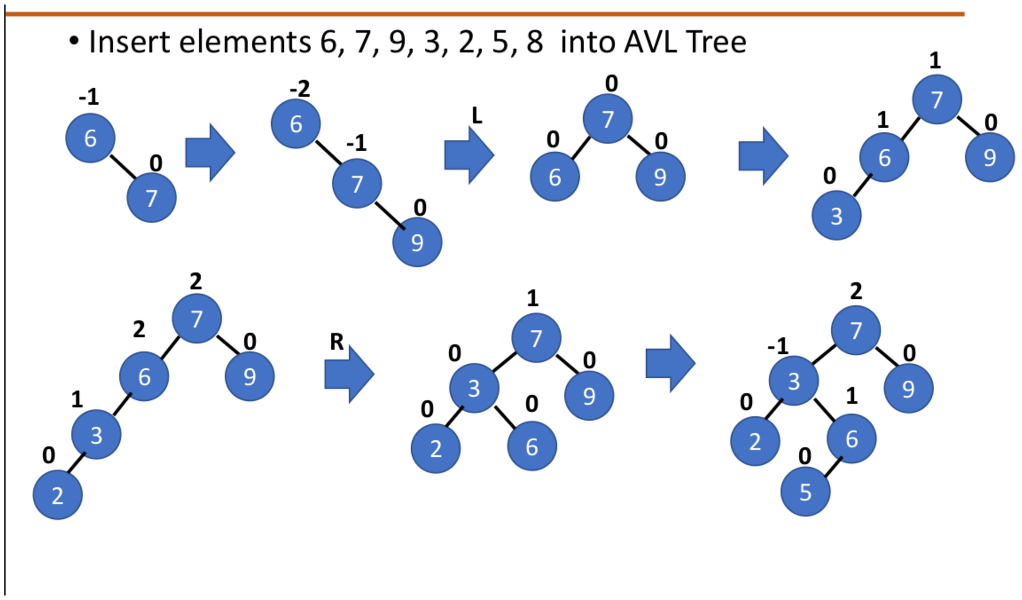


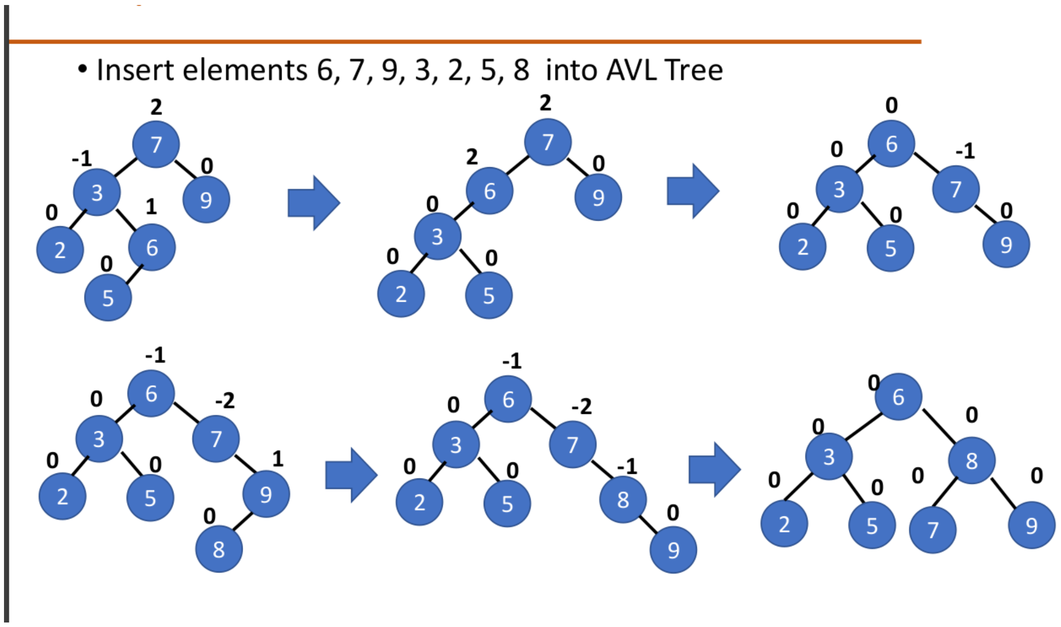
Double Right Left Rotation:



- insertion in avl tree :
    - after insertion first we find youngest ancestor which becomes unbalanced after element is inserted. then we look for above 4 patterns of rotation(where pattern starts with youngest unbalanced ancestor is at top) and rebalance using those rotations

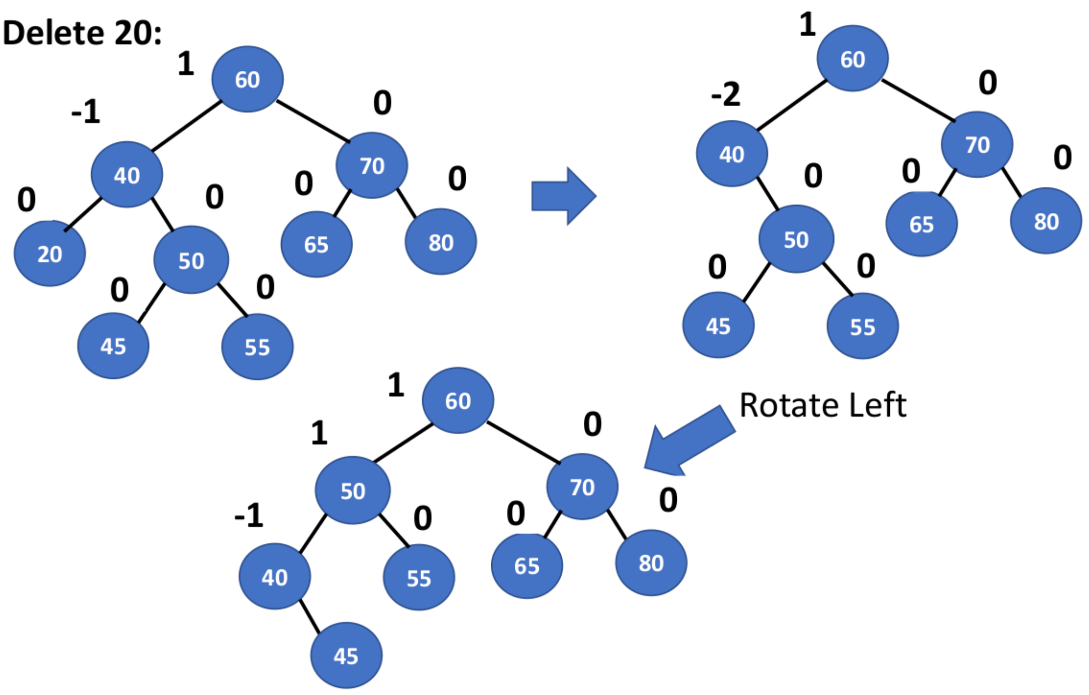
- ex :


• Insert elements 6, 7, 9, 3, 2, 5, 8 into AVL Tree

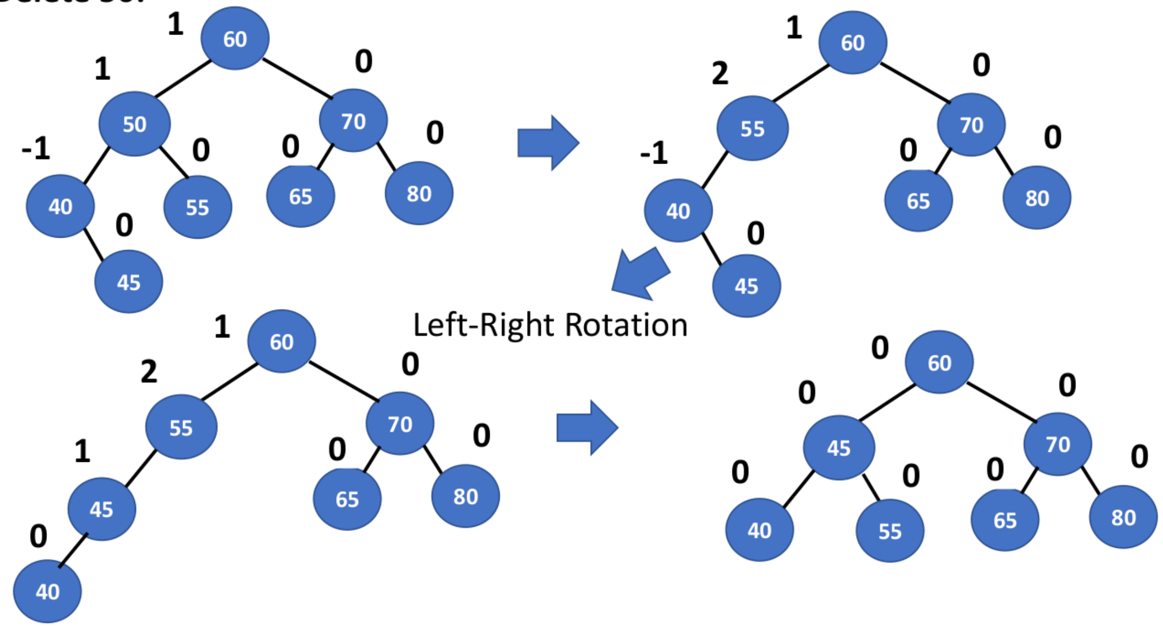

• Insert elements 6, 7, 9, 3, 2, 5, 8 into AVL Tree

- deletion in avl :
  - after deletion first we find youngest ancestor which becomes unbalanced after element is deleted. then we look for above 4 patterns of rotation(where youngest unbalanced ancestor is at top of pattern to search for) and rebalance using those rotations

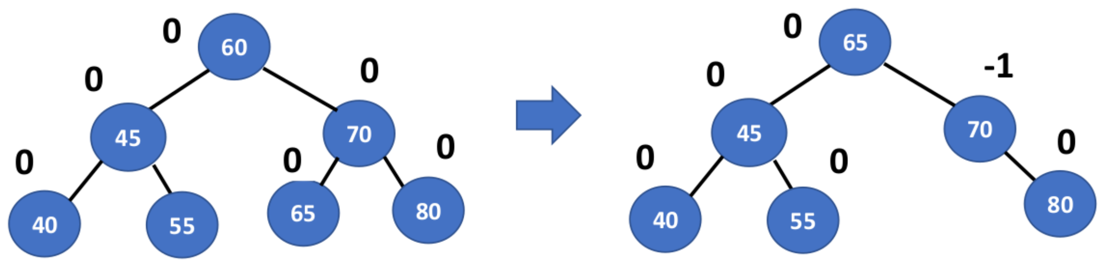
Delete 20:



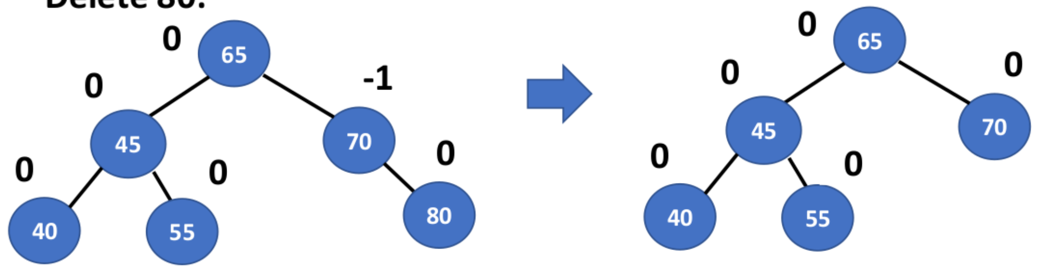
Delete 50:



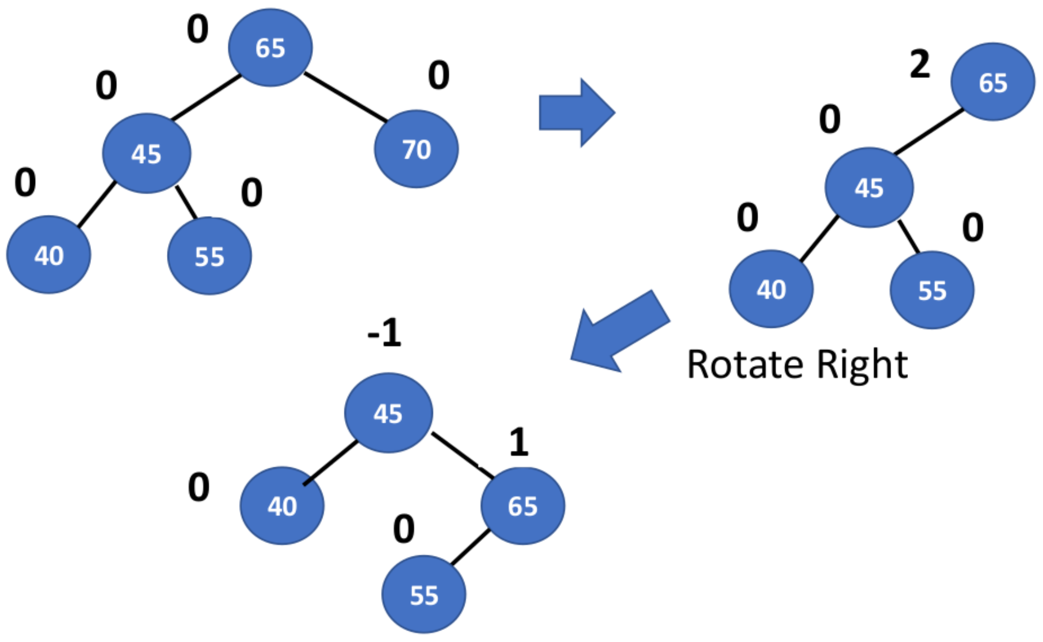
Delete 60:

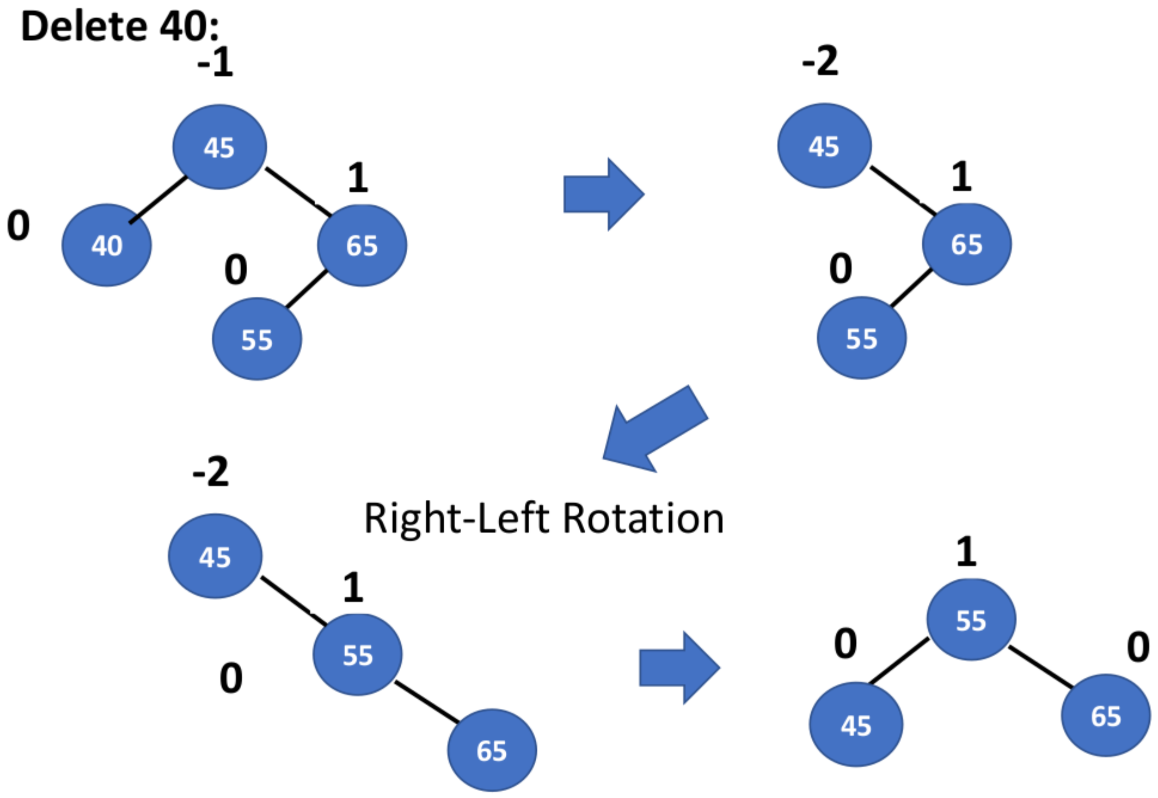


Delete 80:



Delete 70:

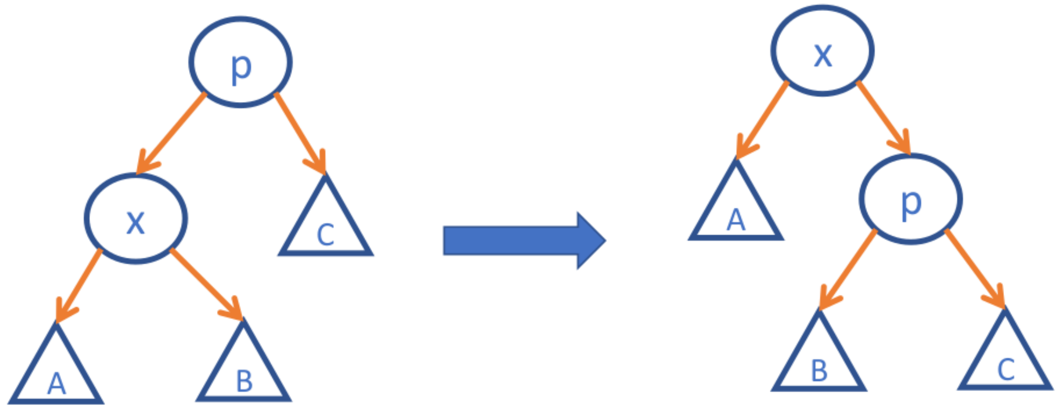




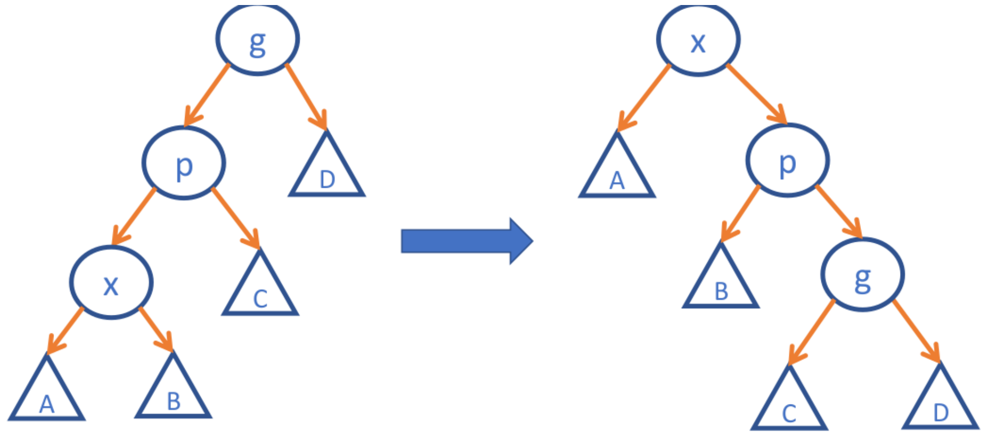
SPLAY TREE

- A splay tree is a binary search tree with the additional property that recently accessed elements are quick to access again.
- it performs op in o(logn) time . all normal ops in bst combined with splaying operation
- splayin the tree rearranges the tree so that element is placed at top.like if we inserted on element , new element iis root. if we search for element, that element becomes root
- splaying : makin x as top/root
  - p is parent of x , x is element to splay , g is grandparent of x
  - zig step : p is root
    - if x is lchild of p, make p rchild of x

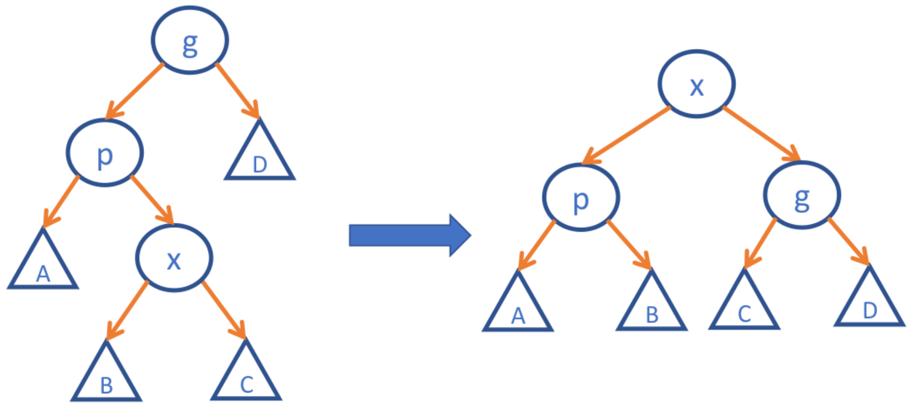
- if x is rchild of p make p lchild of x



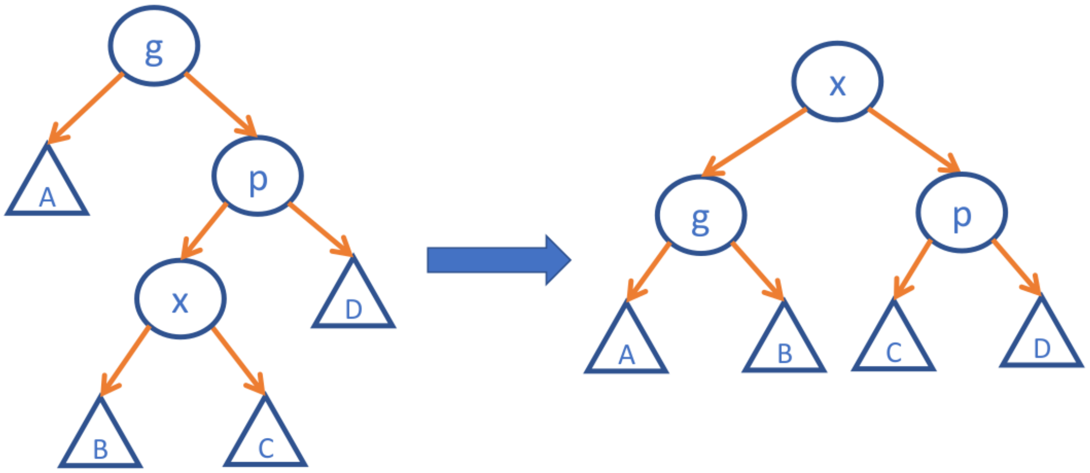
- zig zig step : x,p are either both lchildren or both rchildren
  - if x,p both lchildren of g, make p,g children of x
  - if x,p both rchildren of g make p,g lchildren of x



- zig zag step : x is rchild,p is lchild or x is lchild, p is rchild
  - x>p , so make p lchild of x, x<g so make g rchild of x



        - x<p,so make p rchild of x,x>g so make g lchild of x



- delete :
    - To delete a node x, use the same method as with a binary search tree:
    - If x has two children:
        - Swap its value with that of either the rightmost node of its left sub tree (its in-order predecessor) or the leftmost node of its right subtree (its in-order successor).
        - Remove that node instead.
        - In this way, deletion is reduced to the problem of removing a node with 0 or 1 children.
    - Unlike a binary search tree, in a splay tree after deletion, we splay the parent of the removed node to the top of the tree.

N-ARY TREES

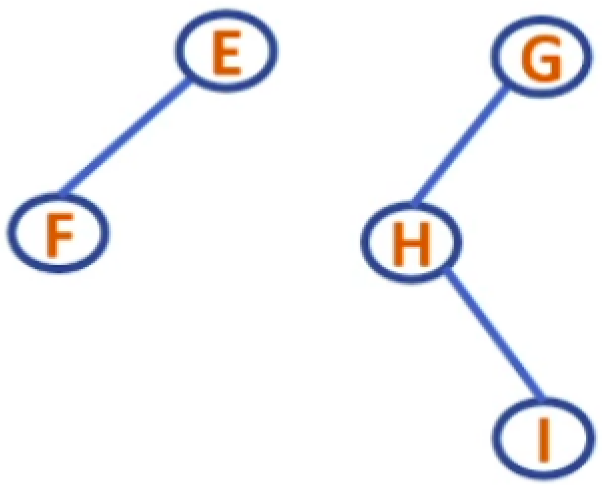
- Ordered tree: relative ordering of nodes matte.First child is oldest , last youngest
- N-Ary Tree: Each node has no more than `N` Nodes.
- Forest : ordered set of ordered trees

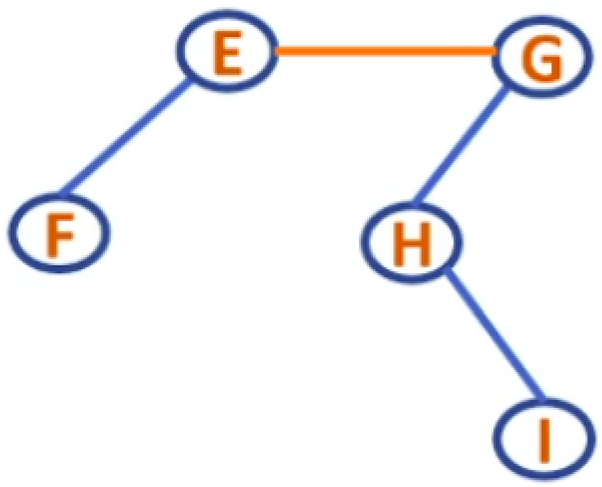
```
struct treenode{
        int info;
        struct treenode *child[MAX];
};
```

- This structure for an n-ary tree is not very efficient as some nodes may not have n children. Hence pointers are wasted.

```
typedef struct treenode{
        int info;
        struct treenode *child;
        struct treenode *sibling;
}vortex; //nocde has link to first child and immediate sibling
```

- This converts an n-ary tree into a binary tree.
- Left-Child Right-Sibling Representation
- We connect all siblings and disconnect all links from parent to everything except it's left most child.
- Root node will have no right children in this representation
    - A forest can be combined into a single binary tree by converting each one into a binary tree and then linking the *root* of the second tree as the right sibling of the first.





]

Traversals

Preorder

- Visit root of first tree

- Traverse the forest of subtrees formed by this tree
    - visit first tree
    - check sibling and repeat
    - backtrack if null
- then move to next tree

```
void preorder(tree* root){
        if(root!=NULL){
                printf("%d",root->data);
                preorder(root->child);
                preorder(root->sibling);
        }
}
```

Inorder

- Visit the first tree.
- Then visit the root of the first tree
- then move to the next trees

```
void inorder(tree* root){
        if(root!=NULL){
                inorder(root->child);
                printf("%d",root->data);
                inorder(root->sibling);
        }
}
```

Postorder

- Traverse all the desendents
- Traverse all the siblings
- Then visit root

```
void postorder(tree* root){
        if(root!=NULL{
                postorder(root->child);
                postorder(root->sibling);
                printf("%d",root->data);
        }
}
```

Quick fix

- Convert n-ary tree into binary tree using Left-Child Right-Sibling method
- Then use the binary tree traversals to figure out inorder,postorder and preorder
  - Preorder : Root,Left,Right
  - Inorder : Left,Root,Right
  - Postorder : Left,Right,Root

GRAPHS

- A Graph is a data structure that consists of set of vertices and a set of edges that relate the node to each other.
- Undirected Graph: A graph is undirected, when the pair of vertices representing any edge is unordered.here edge i,j and j,i are same .no of possible edges in an m vertex undirected graph= m*(m-1)/2
- Directed Graph: A graph with all directed edges is called diagraph or directed graph.here i,j and j,i are different edges.no of possible edges in m vertex undirected graph = m*(m-1)
- Weighted Graph: A weighted graph is a graph where each edge has a numerical value called weight.
- Adjacent Nodes : A node n is adjacent to node m if there is an edge from m to n.n is called the successor of m and m is called the predecessor of n.
- Cycle :A path from node to itself is called a cycle or cycle is path in which first and last vertices are same. A graph with at-least one cycle is called cyclic graph.
- acyclic : graph with no cycles.directed acyclic graph called dag
- Incident: A node n is incident to an edge x, if node is one of the two nodes the edge connects.
- Degree: The degree of vertex i is the number of edges incident on vertex i
- In-degree: In-degree of vertex i is the number of edges incident to i.
- Out-degree: Out-degree of vertex i is the number of edges incident from i.
- adjacency matrix :
  - m[\i][j]=1 if i,j is an edge and 0 if there is no edge between i and j
  - diagonal elements = 0 as no edge from node to itself
  - directed adjacency matrix are assymetric while undirected are symmetric
  - space complexity : o(v^2) with v nodes
  - code :

```
struct node
{
```

```c
        //information associated with each node
};
struct arc
{
        int adj;// information associated with each edge
        int weight;
};
struct graph
{
        struct node nodes[MAXNODES];
        struct arc arcs[MAXNODES][MAXNODES];
}
struct graph g;
```

~~~
   - to add edge  from node1 to node2: g\[node1\]\[node2\].adj=1 ; g\[node1\]\[node2\].weight=wt
- adjacency list : each node maintains linked list of members
   - ![[Pasted image 20231210162018.png]]
   - ![[Pasted image 20231210162038.png]]
   - in multilinked structor we hv header node which consists of all vertexs, an edge pointer which is a list of all edges vertex points to ,![[Pasted image 20231210162052.png]]
- header node is list of vertexs , sample list node is list of edges
- graph traversals :
   - Depth First Search - DFS
      - Visits all the nodes related to one neighbour before visiting the other neighbours and its related nodes. Once it reaches dead end, it backtracks along previously visited node and continue traversing from there.
      - uses stack
      - implementation (stack) : for each vertex initially push(vertex) at front, pop(vertex) at front mark vertex visitted , push(neighbours) at front
      - this way they re traversed path by path
```c
void dfs_adjMat(GRAPH *g,int start){
    static int visited[MAX];
    printf("%d",start);
    visited[start]=1;
    for(int i=1;i<=g->vertex;i++){
        if(g->adjMatrix[start][i]==1 && visited[i]==0)
            dfs_adjMat(g,i); //recursive call
    }
~~~

```
}
void dfs_adjList(NODE *l,int source){
    NODE *t;
    static int visited[MAX];
    t=&l[source];
    printf("%d->",t->data);
    visited[source]=1;
    while(t->next!=NULL){
        t=t->next;
        if(visited[t->data]==0)//if unvisited
            dfs_adjList(l,t->data);
        }
    }
```

- Breadth First Search - BFS
    - implementation (queue) : for each vertex initially enqueue(vertex) at emd, dequeu(vertex) at front mark vertex visitted , enqueue (neighbours) at end
    - this way only traversed level by level
    -

```
bfs adjacency matrix algo ()
        visited[source]=1
        enqueue(queue,source);
        while(!isempty(queue)){
                 i=dequeue(queue);  //remove from front parent node
                 for(int j=1;j<=v;j++){
                    if(g->adjMatrix[i][j]==1 && visited[j]==0){
                        enqueue(queue,j); // insert at end neighbours of
parent node
                        visited[j]=1;
                        printf("%d ",j);
                    }
                 }
             }
bfs adjacency list algo()
        visited[source]=1;
        enqueue(queue,source);
     while(!isempty(queue))
     {
         i=dequeue(queue);
         t=&l[i]; // l is adjacenct list
         while(t->next!=NULL)
         {
            t=t->next;
            j=t->data;
            if(visited[j]==0){
                enqueue(queue,j);
                visited[j]=1;
                printf("%d ",j);
            }
         }
     }
```

- if we replace above code with stack instead of queue it becomes dfs