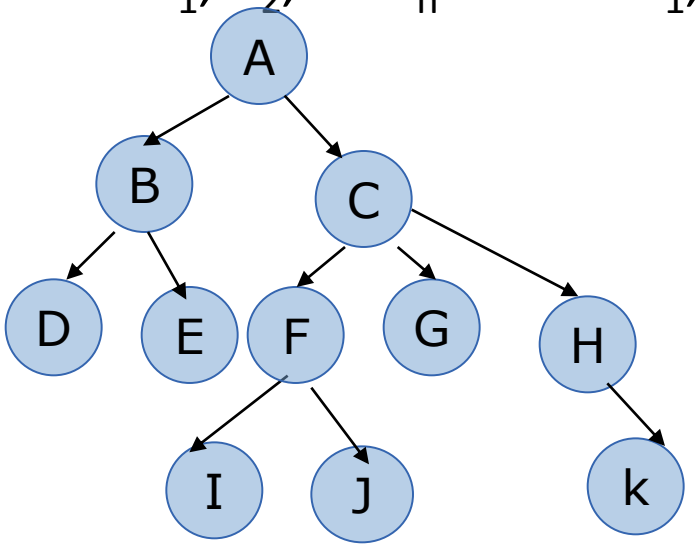
Non Linear Data Structures

By

Kalyani C.Waghmare

Non Linear data structures

- **Tree & Graph**
- **Tree**
  - Finite set of one or more nodes such that
    - There is a special node called as root.
    - Remaining nodes are partitioned into n>=0 disjoint sets T_1, T_2,.....T_n where T_1,.....T_n are sub trees of root.



Tree Basics

- **Node –** contains item information & branches /links to other nodes.
- **Degree of node :-** The number of subtrees of a node
- **Degree of tree :-** Maximum degree of nodes in the tree.
- **Leaf/terminal nodes-** The nodes having degree zero
- **Non terminal/Non leaf/Internal nodes –** other than leaf nodes
- **Siblings** :- Children of same parent
- **Ancestor** :- Node N_1 is an ancestor of N_2 if N_1 is father of N_2 or father of ancestor of N_2

Tree Basics continued...

- **Level –** Node will be at level 'l' & its children will be at l+1
- **Height/Depth :-** Maximum level of any node in the tree.
- **Binary Tree** :- Finite set of nodes that either empty or consists of a root and two disjoint binary trees called left subtrees & right subtrees.
- **Right Descendant :-** Node N_2 is right descendant of N_1 if node N_2 is either the right son of N_1 or descendant of right son of N_1.
- **Left Descendant** :- Node N_2 is left descendant of N_1 if node N_2 is either the left son of N_1 or descendant of left son of N_1.

Tree Basics continued...

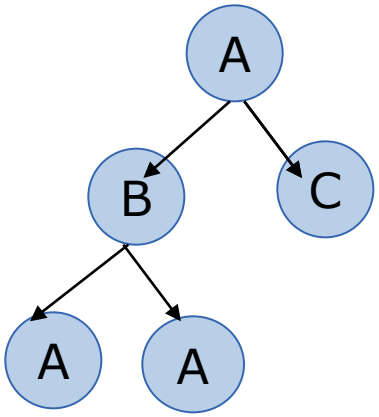
- **Left skewed Tree** :- No right subtree
- **Right skewed tree** :- No left subtree
- **Maximum No. Of nodes in a binary tree – sum of all nodes in a binary tree**
- **Max no. Of nodes of binary tree of depth k**
- **$= 2^k - 1$ if k>=1**

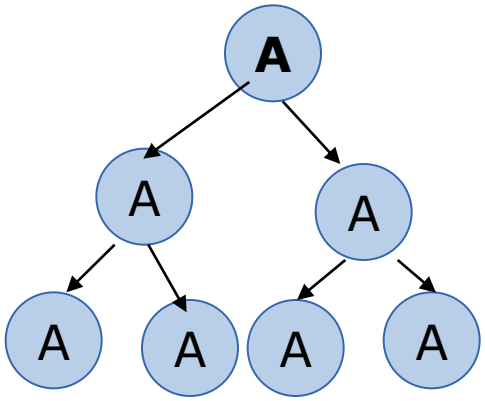
  or $= 2^{K+1} - 1$ if k>=0
- **Max no. Of nodes of at level i in a binary tree**
- **$= 2^{i-1}$ if i>=1**

  or $= 2^i$ if i>=0

Tree Basics continued...

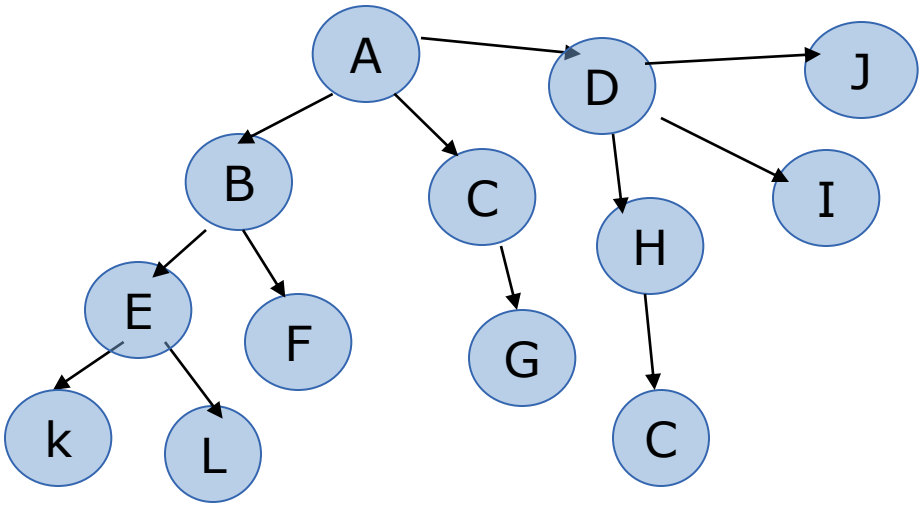
- **Full Binary Tree :-** A full binary tree of depth k is binary tree of depth k having $2^k - 1$ nodes k>=1
- **Complete Binary tree :-** A binary tree with n nodes and of depth k is complete iff its nodes corresponds to the nodes which are numbered one to n in the full binary tree





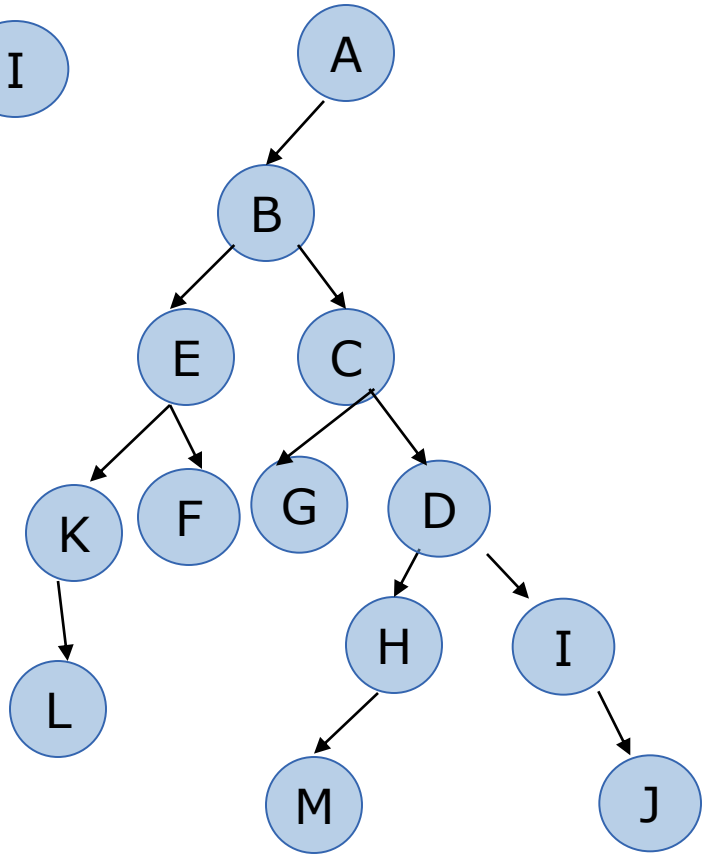
Tree to Binary tree conversion

- Left child – right sibling Representation



General Tree

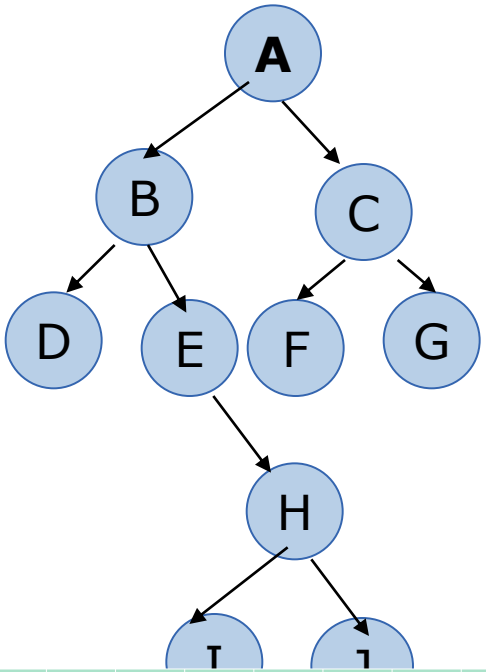
Binary Tree



Representation Methods

- **Static Representation – using arrays**
- **Dynamic Representation – using linked organization**
- **Static Representation**
  - Nodes are numbered from 1 to n
  - One dimensional array could be used
  - Parent of i^{th} element is at i/2 if i!=1
  - If i is 1 i is root & no parent it has
  - leftchild(i) is always at 2i
  - rightchild(i) is always at 2i+1

Tree Representation



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
	A	B	C	D	E	F	G	-	-	-	H	-	-	-	-	-	-	-	-	-	-	I	J

Representation Methods

- **Disadvantages**
  - Wastage of memory if tree is not complete binary tree
  - Not suitable for frequent insertion and deletion
- **Linked organization**
  - Using dynamic memory allocation.
  - Collection of nodes having one data field & two link fields.
  - Llink to point leftsubtree
  - Rlink to point rightsubtree

Representation Methods

- **Class node {**
    - int data;
    - tree_node *lchild;
    - tree_node *rchild;
  - Public:
    - Friend class tree;
    - node(int x)
    - {data = x; lchild=rchild = NULL; }
- **};**
- **Class tree {**
    - node *root;
  - Public:
    - Tree() { root = NULL ; }
    - Void create();
    - Void insert(int,*tree_node);
- **};**

Tree Traversals Method

- **Inorder**
- **Preorder**
- **Postorder**
- **BFS – Breadth First Search**
- **DFS – Depth First search**
- Inorder Traversal - LVR
  - The order is left-child, root node, Right-child
- Preorder traversal – VLR
  - The order is root node, left child , right child.
- Postorder traversal – LRV
  - The order is left child, right child, root node.

Inorder Traversal

1. It is same as infix expression
2. Moving down the tree towards the left until NULL is reached.
3. Visit the node.
4. Move one node to right & continue with step 2
5. If you can not move to right, go back one more node & continue with step 2

Pre-order Traversal

1. It is same as prefix expression
2. Visit the node.
3. Moving down the tree towards the left & visit each node until NULL is reached.
4. Move one node to right & continue with step 2.
5. If you can not move to right, go back one more node & continue with step 2

Tree Traversals – Recursive Method

Procedure Inorder(node *T)

If (T!= NULL)

- inorder(t->lchild);
- Print t->data;
- inorder(t->rchild);

End if

End inorder

Procedure preorder(node *T)

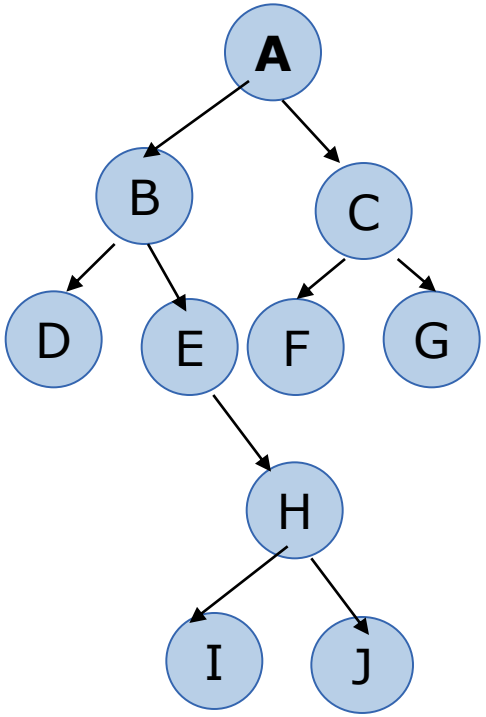
If (T!= NULL)

- Print t->data;
- preorder(t->lchild);
- preorder(t->rchild);

End if

End preorder

Tree Representation



Inorder	DBEIHJAFCG
Preorder	ABDEHIJCFG
postorder	DIJHEBFGCA
BFS	ABCDEFGHIJ

Postorder Traversal

1. It is same as postfix expression
2. Moving down the tree towards the left until NULL is reached.
3. Move one node to right & continue with step 2
4. Visit the node
5. If you can not move to right, go back one more node

Tree Traversals Recursive Method

Procedure postorder(node *T)

If (T!= NULL)

- postorder(t->lchild);
- postorder(t->rchild);
- Print t->data;

End if

End postorder

Tree Traversals – Non Recursive Method

Procedure Non-Inorder(stack s)

while(1)

- while(T!=NULL)
  - s.push(T);
  - T = T->lchild;
- End while
- if(s.empty()) then return;
- T = s.pop();
- Print t->data;
- T = T->rchild;

End while

End non-inorder

Tree Traversals – Non Recursive Method

Procedure Non-preorder(stack s)

while(1)

- while(T!=NULL)
  - Print t->data;
  - if(t->rchild !=NULL)
    - s.push(T->rchild);
  - T = T->lchild;
- End while
- if(s.empty()) then return;
- T = s.pop();

End while

End non-preorder

Tree Traversals – Non Recursive Method

Procedure Non-postorder(stack s)

Node * temp, *p1; Temp = new node('-1');

while(1)

- while(T!=NULL)
  - s.push(T);
  - if(t->rchild !=NULL)
    - s.push(T->rchild); s.push(temp);

  End if
  - T = T->lchild;
- End while
- if(s.empty()) then return;
- p1 = s.pop();
- while(p1!=temp &&!s.empty())
  - print p1->data; P1 = s.pop();

End while

If(!e.empty()) then T = s.pop();

End while

Class for Tree

- **Class node {**
    - int data;
    - tree_node *lchild;
    - tree_node *rchild;
  - Public:
    - Friend class tree;
    - Friend class stack;
    - node(int x)
    - {data = x; lchild=rchild = NULL; }
- **};**
- **Class tree {**
    - node *root;
  - Public:
    - Tree() { root = NULL ; }
    - Void create();
    - Void insert(int,*tree_node);
- **};**

Class for Tree continue...

```
Class stack
{ int top = -1;     node *s[20];
Public:
stack()
{           Top = -1;    }
void push(node *p)
{    If(top ==19)
            Print " stack is full";
      Else
            Top++;  S[top] = p;
}
node * pop()
{     node *x;
      If(top ==-1)
            Print " stack is empty";
      Else
            •     X = s[top];   Top—;  Return(x);
            }}
```

Tree Traversals Method

```
Procedure BFS()
     Node *q[20]; int f =r =-1;                    // Queue of tree node
     F++;  r++;    q[r] = t;                       // adding root in Queue
    while (f !=-1)                                 //while queue not empty
      Temp = q[f];   f++;                          //delete from queue
      if(f=r+1) then f= r= -1;                     // if only item in Queue
      Print temp->data;                            //print data deleted from queue
      if(temp->lchild!=NULL)                       // if left child is present
            if(f==-1) then f++;
            R++;     q[r] = temp->lchild);    // add it in Queue
      endif
      if(temp->rchild!=NULL)          // if right child is present
            if(f==-1) then f++;
            R++;     q[r] = temp->rchild);       // add it in Queue
      End if
End while
```

Tree Creation & Insertion

```
Procedure create( int  x)
      Node *P;
      If (Root== NULL)
      P = new node(x);
      Root= P;
      Else
      Print' Tree root is already created calling insert function';
      Root = insert(Root,x)
End if
End create
```

Insertion

```
Procedure insert( node *t, int x)
      Node *P;
       If (T== NULL)
       P = new node(x)
       return(P);
      Else
            Print' where you want to insert on left or right of (l/r) ' + t->data;
            cin>>ans;
            if(ans=='l'  or ans=='L')
             t->lchild = insert(t->lchild,x)
           Else
             t->rchild = insert(t->rchild,x)
       End if
Return(t);
End if
```

Insertion Breadth wise

```
Procedure insert(Queue q1)
     node *temp,*p;                                  //  node pointer
     q1.add(Root);                                      // adding root in Queue
    while (Queue is not empty)                    //while queue not empty
      Temp = q.delete()                          //delete from queue
      Print' Do you want to insert on left of' + temp->data;
            cin>>ans;
            if(ans=='y'  or ans=='Y')                 // if only item in Queue
             p = new node(x)                  // scan x value & create one node
             temp->lchild = p;
             q1.add(p);
            End if
      Print' Do you want to insert on right of' + temp->data;
            cin>>ans;
            if(ans=='y'  or ans=='Y')                 // if only item in Queue
             p = new node(x)                  // scan x value & create one node
             temp->rchild = p;
             q1.add(p);
End while
End insert
```

Queue operations

```
Queue :: Queue()
{          front = rear = -1;    }
Queue :: add(node * val)
{
      If(rear ==19)
            Print “Queue is full”;
      Else
            rear++ ;  if (rear ==0) then    front = rear;
            q[rear] = val;
}
node * Queue :: delete()
{
      If(front ==-1)
            Print “Queue is empty”;
      Else
            X = q[front];
            Front++;   if(front ==19)   then    front =rear =-1
            Return(x);
```

Height of tree

```
Procedure height ( node *t)
    int h1 h2;
    If (t== NULL)
    return 0;
    If (t->lc == NULL && t->rc = =NULL)
    return 0;
    h1 = height(t->lc);
    h2 = height(t->rc);
    if(h1>h2)
        return(h1+1);
    else
        return(h2+1);
    End if
End height
```

Mirror image

```
Procedure mirror ( node *t)
Node *temp
If (t!== NULL)
    mirror(t->lc);
    mirror(t->rc);
     temp = t->lc;
     t->lc = t->rc;
     t->rc = temp;
endif
End mirror
```

```
Procedure node *mirror ( node *t)
Node *temp
If (t!== NULL)
    temp = newnode(t->data);
    temp->lc = mirror(t->rc);
    temp->rc = mirror(t->lc);
    return(temp);
Else
     return(NULL);
endif
End mirror
```

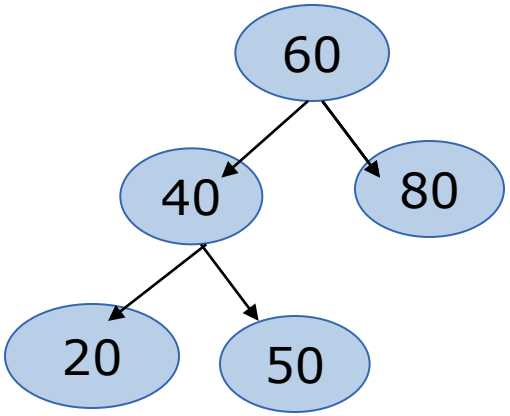
Create tree from pre-order & post-order traversal

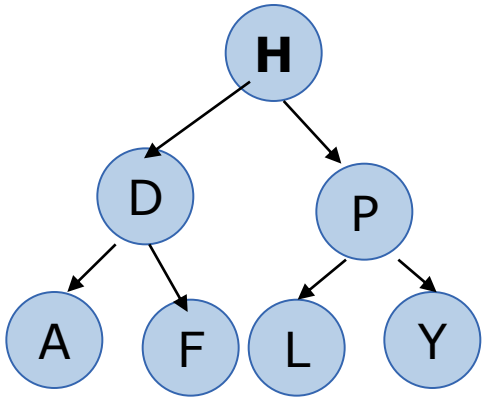
- **Algorithm**

1. **Read last node of postorder traversal that will be root node – mark as y**
2. Read previous node of root in the post order traversal mark as 'x'
3. If x is immediate after root node (y) in pre-order then x is left child of root otherwise if x is coming sequentially after y in pre-order then x is right child of y
4. Repeat step 2,3 for all nodes in post-order traversal in backward direction with respect to every root node having sub-trees are null.

Binary Search Tree

- It is a binary tree. It may be empty .If it is not empty then it satisfies the following propertities
- Every element has key & number two elements have same key
- The keys in left subtree are smaller than the key in root
- The keys in rightt subtree are greater than the key in root
- The left & right subtrees are also BST.





Binary Search Tree Creation & Insertion

```
Procedure create( int  x)
      If (Root== NULL)
        Root = new node(x);
      Else
             Node *p,q ; p = q = root;
            While(p!=Null)
        q=p;
        if(x < p->data)
               p = p->lchild;
        Else
                p = p->rchild;
           End while
  if(x < q->data)
        T = new node(x);    q->lchild = T;
           Elseif(x > p->data)
        T = new node(x);    q->rchild = T;
          Else
        Print 'duplicate data'
         Endif
```

Binary Search Tree Creation & Insertion

```
Procedure node * create(node * t,  int  x)
      If (t== NULL)
       t = new node(x);
                return(t);
Else
            if(x < t->data)
             t->lchild = create(t->lchild,x);
       Else
            t->rchild = create(t->rchild,x);
              Endif
Return t
   Endif
End create.
```

Deletion of node from binary/Binary search tree

- **Deletion of leaf node element**
  - If node is left of parent then make left child of parent as NULL
  - If node is right of parent then make right child of parent as NULL
- **Deletion of non leaf node with only one subtree**
  - Non leaf node has left subtree & not right subtree then parent of non leaf node will point to left child of that node.
  - Non leaf node has right subtree & not left subtree then parent of non leaf node will point to right child of that node.
- **Deletion of non leaf node with both child**
  - Non leaf node element is replaced by either largest element in left subtree(in-order predecessor) or smallest element in right subtree (in-order successor).
  - Actual(physical) deletion is of leaf node only.

Deletion of node from binary/Binary search tree

```
Procedure delbst(node * t,  int  x)
      If (t== NULL)
       return(0);
Elseif(x < t->data)
            delbst(t->lchild,x);
Elseif(x > t->data)
            delbst(t->rchild,x);
      Else if(t->lchild==NULL AND t->rchild==NULL)
            P = t;
            free(p);
            Return(1);
    else If (t->lchild != NULL)
            p = t->lchild;
```

Deletion of node from binary/Binary search tree

```
                while(p->rchild!=NULL)
                » P = p->rchild;
                » t->data = p->data;
                » return(delbst(p,p->data);
        Else
                » p = t->rchild;
                » while(p->lchild!=NULL)
                » P = p->lchild;
                » t->data = p->data;
                » return(delbst(p,p->data);
        Endif
Endif
```

Deletion of node from binary/Binary search tree

```
Procedure delbst(node * p,  node *c)
    if(c->lchild!=NULL AND c->rchild!=NULL)
      P = c;
      – Node * c_s = c->rchild;
      – while(c_s->lchild!=NULL)
            • {P = c_s;        c_s = c_s->lchild;}
   c->data = c_s->data;   C = c_s;
    endif
    if(c->lchild==NULL AND c->rchild!=NULL)
      if(p->rchild ==c)
            • p->rchild = c->rchild;
      else
       p->lchild = c->rchild;
      endif
      Delete (c )
    endif
```

Deletion Non recursive of node from binary/Binary search tree

```
if(c->lchild!=NULL AND c->rchild==NULL)
        if(p->rchild ==c)
            • p->rchild = c->lchild;
         else
         p->lchild = c->lchild;
         endif
         Delete (c )
endif
if(c->lchild==NULL AND c->rchild==NULL)
        if(p->rchild ==c)
            • p->rchild = NULL;
         else
                p->lchild = NULL;
          endif
 Delete(c )
endif
```

Threaded Binary Tree (TBT)

- Binary tree with n nodes have n+1 null links.
- Wastage of memory b'coz of null links.
- Recursion is required to traverse the tree completely.
- TBT utilizes null links.
- The use of null links removes requirement of recursion.
- Can create inoder, preorder, postorder TBT by using respective traversal.
- Inorder TBT -
  - Non-null left child will point to its left child act as left link
  - Non-null right child will point to its right child act as right link
  - null left child will point to inorder predecessor act as thread
  - null right child will point to inorder successor act as thread
- Need additional field to distinguish link & thred
- Boolean lbit, rbit
- Lbit =0 Llink – thread Lbit =1 Llink –Link (pointing to left child)
- Rbit =0 Rlink – thread Rbit =1 Rlink –Link (pointing to right child)

Representation Methods

- **Class tbt_node {**
    - int data,lbit,rbit;
    - tbt_node *lc;
    - tbt_node *rc;
  - Public:
    - Friend class tbt;
    - tbt_node(int x)
    - {data = x; lc=rc = NULL; lbit=rbit=1; }
- **};**
- **Class tbt {**
    - tbt_node *root;
  - Public:
    - tbt() { root = NULL ; }
    - Void create();
    - Void insert(int,*tbt_node);
- **};**

Tree Creation & Insertion

```
Procedure create()
    tbt_node *P;
    If (head == NULL)
    P = new tbt_node();
    head = P;
    P->lc = P->rc = P;
    P->lbit =0; P->rbit = 1;
    Else
    Print' tree head is created';
    – Queue q1;
    – Input Enter data want to insert in TBT in x
    insert(q1,x); //breadthwise creation of TBT
    End if
End create
```

Insertion Breadth wise

```
Procedure insert(Queue q1, int x)
     node *temp,*p;                                  //  node pointer
     q1.add(head);                                        // adding root in Queue
    while (Queue is not empty)                      //while queue not empty
        Temp = q.delete();                          //delete from queue
        Print' Do you want to insert on left of' + temp->data;
        cin>>ans;
       if(ans=='y'  or ans=='Y')                 // if only item in Queue
                p = new tbt_node(x);                 // create one node for x
                linsert(temp,p);  q1.add(p);
        End if
        if( temp!=head)
             Print' Do you want to insert on right of' + temp->data;
            cin>>ans;
            if(ans=='y'  or ans=='Y')                // if only item in Queue
                p = new tbt_node(x)                  // scan x value & create one node
                rinsert(temp,p); q1.add(p);
            endif
      End while
End insert
```

```
Procedure create()
      tbt_node *P,*q;
      If (head == NULL)
            P = new tbt_node();
            head = P;
            P->lc = P->rc = P; P->lbit =0; P->rbit = 1;
      Input Enter data for root to insert in TBT in x
            q = new tbt_node(x);
           linsert(head,q);
      Else
      Print' tree head is created';
      Input Enter data want to insert in TBT in x
      head->lc = inser_depth(head->lc,x);
     //Depthwise creation of TBT
      End if
End create
```

Insertion Depthwise

```
Procedure node * create(node * t1,  int  x)
        if(x < t1->data)
            If (t1->lbit==0)
                t = new tbtnode(x);
                linsert(t1,t);
            else
                t = create(t1->lchild,x);
            endif
         Else if(x > t1->data)
            If (t1->rbit==0)
                t = new tbtnode(x);
                rinsert(t1,t);
            else
                t = create(t1->rchild,x);
            endif
        Endif
Return t1
```

Insertion

Procedure linsert(tbt_node s, tbt_node *t)

- t->lc = s->lc;
- t->rc = s;
- t->lbit = s->lbit;
- t->rbit = 0;
- s->lc = t;
- s->lbit = 1;

End linsert

Insertion

Procedure rinsert(tbt_node s, tbt_node *t)

- t->rc = s->rc;
- t->lc = s;
- t->rbit = s->rbit;
- t->lbit = 0;
- s->rc = t;
- s->rbit = 1;

End rinsert

```
procedure TBTInorder()
T = head;
Loop
{
        T = insuccessor(T);
         If(t==head) return;
        Print t->data
}
end TBTInorder

Node *procedure insuccessor(x)
S = x->rightChild;
If(X->rbit==1) then
{
   while(s->lbit==1)
         s=s->lc;
}
end insuccessor
```

```
procedure TBTpre()
T = head->lc;
while(1)
If (t==head )  then return(0);
print t->data;
if(t->lbit ==1)
     t = t->lc;
else
     if(t->rbit==1)
          t= t->rc;
     else
          while(t->rbit!=1)
               t=t->rc;
          t = t->rc;
     endif
endif
end while
end TBTpre
```

```
procedure TBTpost ()
p = head->lc; ch = 'l';
while(1)
     switch(ch)
     case 'l' : if(p-->lbit ==1)
                  p = p->lc;
                else
                  ch = 'r' ; break;
     case 'r' : if(p->rbit ==1)
                  p = p->rc;    ch ='l';
                else
                  ch = 'v';
     case 'v' : print p->data;
                  if(p==head->lc)
                       break;
                  p = Next(p);
     end case
end while
end TBTpost
```

Deletion of node from Thread binary/Binary search tree

```
Procedure tbt_delbst(node * p,  node *t)
    if(t->lbit==1 AND t->rbit==1)
      P = c;
      – Node * c_s = t->rchild;
      – while(c_s->lbit!=0)
            • P = c_s;        c_s = c_s->lchild;
        t->data = c_s->data;
        t = c_s;
    endif
    if(t->lbit==0 AND t->rbit==0)
      if(p->lbit ==1)
            • p->lchild = t->lchild; p->lbit = 0;
       Else
       p->rchild = t->rchild;    p->rbit = 0;
      Endif
     Delete( t )
    Endif
```

Deletion Non recursive of node from binary/Binary search tree

```
if(t->lbit ==1 AND t->rbit==0)
     temp = t->lchild;
      if(p->lchild ==t)
            • p->lchild = temp;
      Else
             p->rchild = temp;
      Endif
      while(temp->rbit==1)
            temp = temp->rchild;
       temp->rchild = t->rchild;
       delete(t )
Endif
```

Deletion Non recursive of node from binary/Binary search tree

```
if(t->lbit ==0 AND t->rbit==1)
     temp = t->rchild;
      if(p->lchild ==t)
            • p->lchild = temp;
      Else
             p->rchild = temp;
      Endif
      while(temp->lbit==1)
            temp = temp->lchild;
       temp->lchild = t->lchild;
       delete(t )
Endif
```

Deletion of node from binary/Binary search tree

```
Procedure delbst(node * p,  node *c)
    if(t->lchild!=NULL AND t->rchild!=NULL)
       P = c;
       – Node * c_s = c->rchild;
       – while(c_s->lchild!=NULL)
            • P = c_s;        c_s = c_s->lchild;
   c->data = c_s->data;
          C = c_s;
    endif
    if(t->lchild==NULL AND t->rchild!=NULL)
       if(p->rchild ==c)
            • p->rchild = NULL;
   Else
        p->lchild = NULL;
    endif
```