

Trees

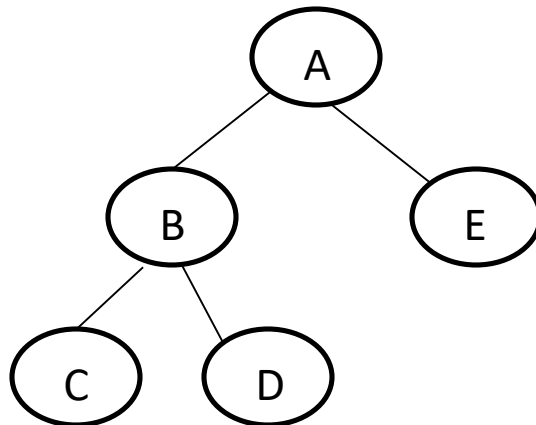
- Trees are natural structures for representing certain kinds of hierarchical data.(How our files get saved under hierarchical directories)
- Tree is a data structure which allows you to associate a parent-child relationship between various pieces of data and thus allows us to arrange our records, data and files in a hierarchical fashion.
- Trees have many uses in computing. For example a *parse-tree* can represent the structure of an expression.
- Binary Search Trees help to order the elements in such a way that the searching takes less time as compared to other data structures.(speed advantage over other D.S)
- Trees are often used in implementing games, particularly “board” games.

Trees

- Linked list is a linear D.S and for some problems it is not possible to maintain this linear ordering.
- Using non linear D.S such as trees and graphs more complex relations can be expressed.

Indegree and outdegree of a node:

- Indegree of a node is the number of edges incident on a node and outdegree is number of edges leaving a node.



- Indegree of B, C, D and E is 1 and that of A is 0.
- Outdegree of A and B is 2 and that of C, D and E is 0.

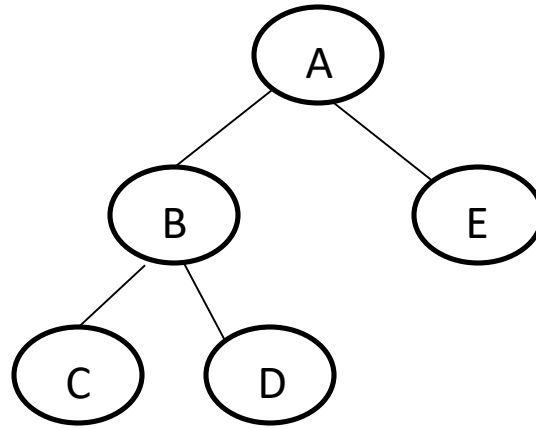
Directed tree:

- Is a tree which has only one node with indegree 0 and all other nodes have indegree 1.
- The node with indegree 0 is called the root and all other nodes are reachable from root.

Ex: The tree in the above diagram is a directed tree with A as the root and B, C, D and E are reachable from the root.

Ancestors and descendants of a node:

- In a tree, all the nodes that are reachable from a particular node are called the descendants of that node.



Descendants of A are B, C, D and E.

Descendants of B are C and D.

- Nodes from which a particular node is reachable starting from root are called ancestors of that node.

Ancestors of D and C are B and A.

Ancestors of E and B is A.

Children of a node:

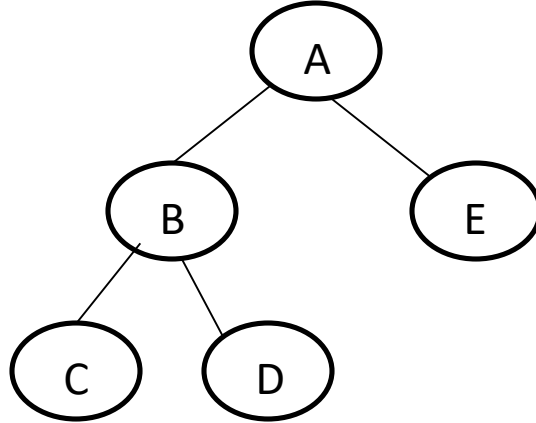
- The nodes which are reachable from a particular node using only a single edge are called children of that node and this node is called the father of those nodes.

Children of A are B and E.

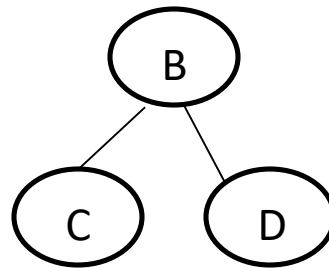
Children of B are C and D.

Left subtree and right subtree of a node:

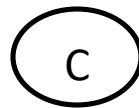
- All nodes that are all left descendants of a node form the left subtree of that node.



left subtree of A is

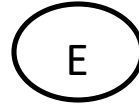


left subtree of B is

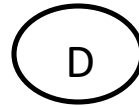


- All nodes that are right descendants of a node form the right subtree of that node.

right subtree of A is



right subtree of B is



right subtree of E is empty tree.

Terminal node or leaf node:

- All nodes in a tree with outdegree zero are called the terminal nodes, leaf nodes or external nodes of the tree.
- All other nodes other than leaf nodes are called non leaf nodes or internal nodes of the tree.

In the previous tree, C, D and E are leaf nodes and A, B are non leaf nodes.

Levels of a tree:

- Level of a node is the number of edges in the path from root node.

Level of A is 0.

Level of B and E is 1.

Level of C and D is 2.

Height of a tree:

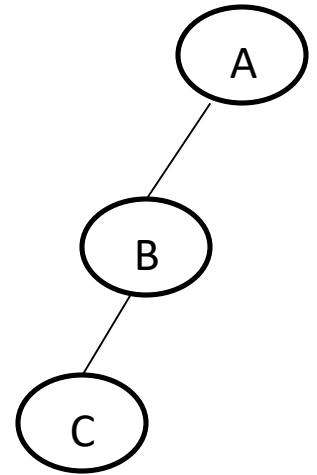
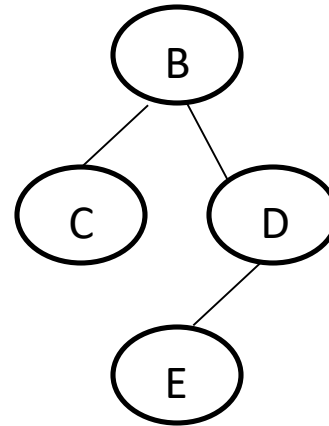
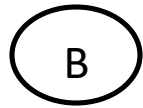
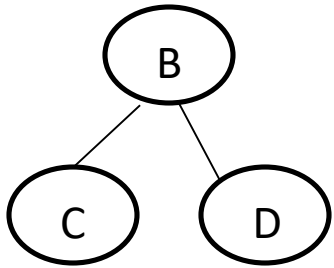
- Height of a tree is one more than the maximum level in the tree.

In the previous tree , maximum level is 2 and height is 3

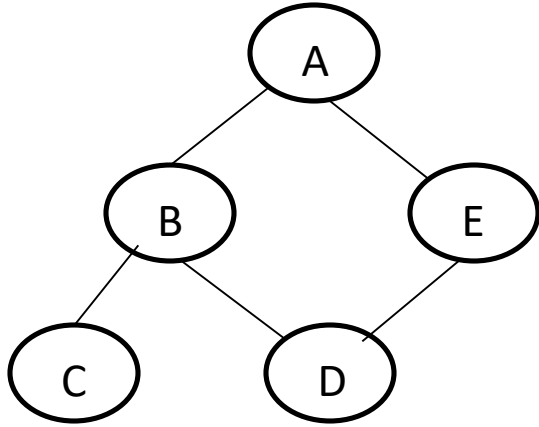
Binary trees:

- A binary tree is a directed tree in which outdegree of each node is less than or equal to 2. i.e each node can have 0, 1 or 2 children.

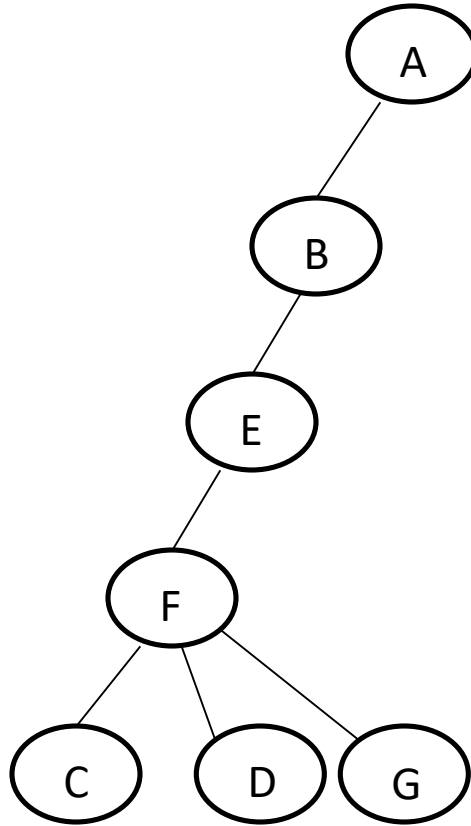
Examples of binary tree



Examples of non binary trees:



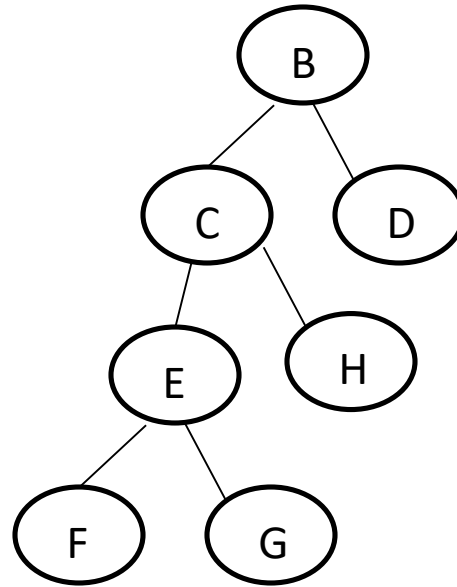
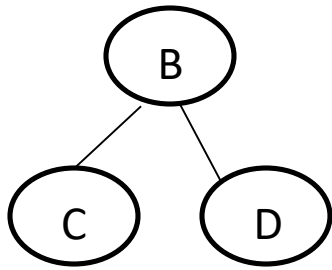
D has indegree 2, hence
not directed tree



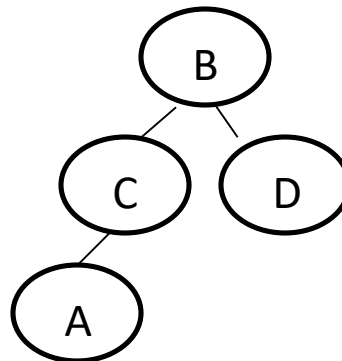
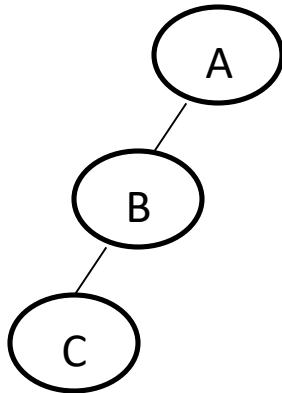
F has outdegree 3.

Strictly binary tree:

- If the out degree of every node in a tree is either 0 or 2(1 not allowed), then the tree is strictly binary tree.

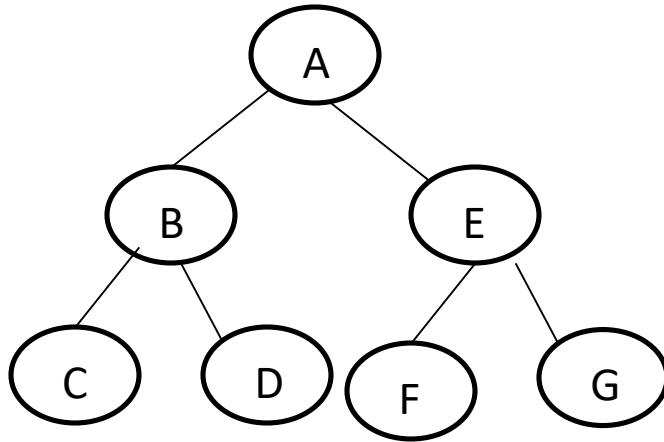


Trees which are binary trees but not strictly binary:



Complete binary tree:

- Is a strictly binary tree in which the number of nodes at any level 'i' is $\text{pow}(2,i)$.



Number of nodes at level 0(root level) is $\text{pow}(2,0) \rightarrow 1$

Number of nodes at level 1(B and E) is $\text{pow}(2,1) \rightarrow 2$

Number of nodes at level 2(C,D,F and G) is $\text{pow}(2,2) \rightarrow 4$

Storage representation of binary trees:

- Trees can be represented using sequential allocation techniques(arrays) or by dynamically allocating memory.
- In 2nd technique, node has 3 fields
 1. Info : which contains actual information.
 2. Llink :contains address of left subtree.
 3. Rlink :contains address of right subtree.

```
struct node
```

```
{
```

```
    int info;
```

```
    struct node *llink;
```

```
    struct node *rlink;
```

```
};
```

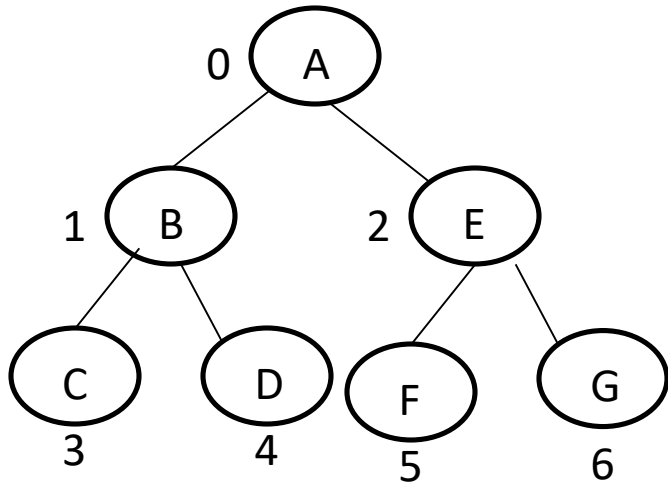
```
typedef struct node *NODEPTR;
```

Implementing a binary tree:

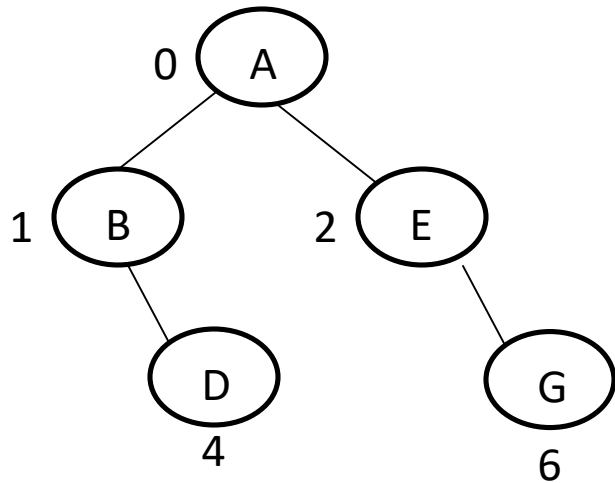
- A pointer variable root is used to point to the root node.
- Initially root is NULL, which means the tree is empty.

NODEPTR root=NULL;

Array representation of binary tree:



0	1	2	3	4	5	6
A	B	E	C	D	F	G



0	1	2	3	4	5	6
A	B	E		D		G

- Given the position 'i' any node, $2i+1$ gives the position of left child and $2i+2$ gives the position of right child.

In the above diagram, B's position is 1. $2*1+2$ gives 4, which is the position of its right child D.

- Given the position 'i' of any node, $(i-1)/2$ gives the position of its father.

Position of E is 2. $(2-1)/2$ gives 0, which is the position of its father A.

Various operations that can be performed on binary trees:

Insertion: inserting an item into the tree.

Traversal: visiting the nodes of the tree one by one.

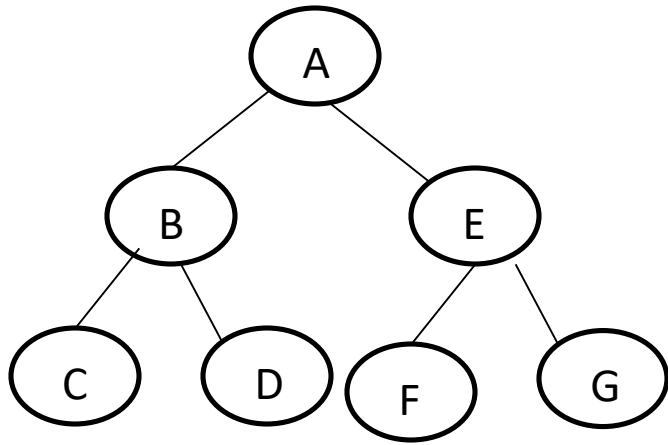
Search: search for the specified item in the tree.

Copy: to obtain exact copy of given tree.

Deletion: delete a node from the tree.

Insertion:

- A node can be inserted in any position in a tree(unless it is a binary search tree)
- A node cannot be inserted in the already occupied position.
- User has to specify where to insert the item. This can be done by specifying the direction in the form of a string.
For ex: if the direction string is “LLR”, it means start from root and go left(L) of it, again go left(L) of present node and finally go right(R) of current node. Insert the new node at this position.



For the above tree if the direction of insertion is “LLR”

- Start from root. i.e A and go left. B is reached.
- Again go left and you will reach C.
- From C, go right and insert the node.

Hence the node is inserted to the right of C.

- To implement this, we make use of 2 pointer variables prev and cur. At any point prev points to the parent node and cur points to the child.

/*function to insert item into tree*/

NODEPTR insert(int item, NODEPTR root)

{

 NODEPTR temp, cur, prev;

 char direction[10];

 int i;

 temp=getnode();

 temp→info=item;

 temp→llink=temp→rlink=NULL;

 if(root==NULL)

 return temp;

 printf(“enter the direction of insertion”);

 scanf(“%s”,direction);

```

prev=NULL;
cur=root;
for(i=0;i<strlen(direction)&&cur!=NULL; i++)
{
    prev=cur;                /*keep track of parent*/
    if(direction[i]=='L')    /*if direction is L, move left of
        cur=cur→llink;        current node*/
    else                      /*if direction is R, move right of
        cur=cur→rlink;        current node*/
}
If(cur!=NULL || i!=strlen(direction))
{
    printf("insertion not possible");
    freenode(temp);
    return root;
}

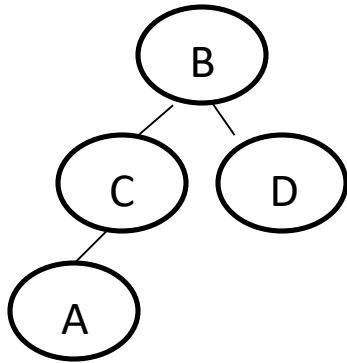
```

```

if (direction[i-1]=='L')    /* if last direction is 'L', point the
    prev→llink=temp;        llink of parent to new node*/
else                       /* else point rlink of parent to new
    prev→rlink=temp;        node*/
return root;
}

```

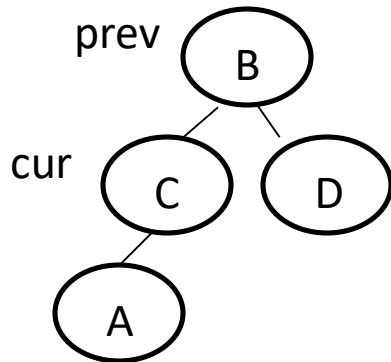
- Control comes out of for loop, if $i = \text{strlen}(\text{direction})$ or when $\text{cur} == \text{NULL}$.
- For insertion to be possible, control should come out when $\text{cur} == \text{NULL}$ and $i = \text{strlen}(\text{direction})$ both at the same time. i.e when we get the position to insert ($\text{cur} == \text{NULL}$), the string should be completed.



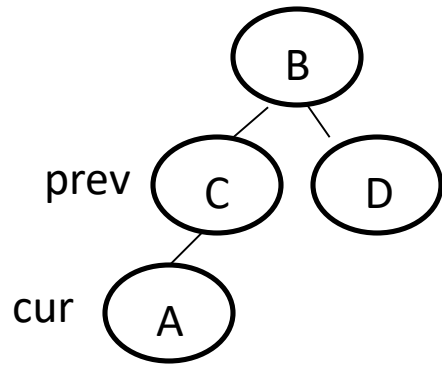
Let the direction string be “LLR”. String length of string is 3.

Initially $cur == root$ and $prev == NULL$

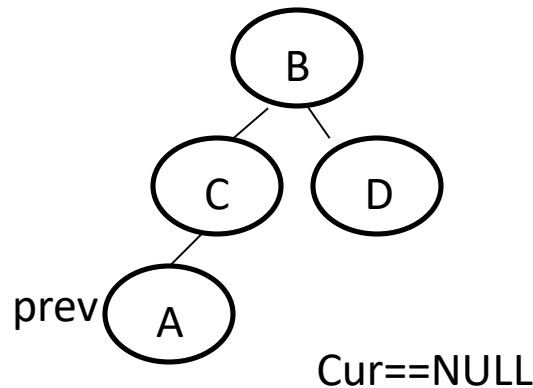
- Loop 1: $i=0$, $direction[0]='L'$



- Loop 2: $i=1$, $\text{direction}[1]='L'$



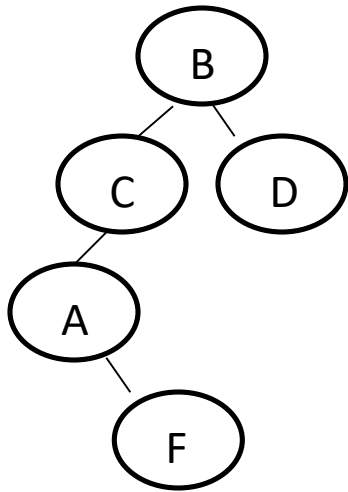
- Loop 3: $i=2$, $\text{direction}[2]='R'$



- $i=3$, Now loop terminates since $\text{cur}==\text{NULL}$. Here $i==3$ (i.e. $\text{strlen}(\text{direction})$) and $\text{cur}==\text{NULL}$. Hence insertion is possible

- Now $\text{direction}[i-1]$ is $\text{direction}[3-1]$ which gives 'R'.
- Hence $\text{prev} \rightarrow \text{rlink}$ is made to point to new node.

After insertion tree looks like



- If direction of insertion is “LLR” for the above tree. Here for loop terminates when $i == \text{strlen}(\text{direction})$ i.e 3. But at this point cur is not equal to NULL. Hence insertion is not possible.

Traversals:

- Is visiting each node exactly once systematically one after the another.

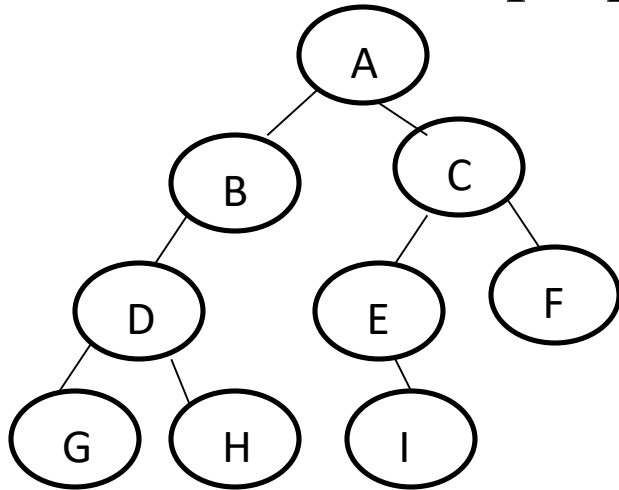
There are 3 main traversals techniques

- Inorder traversal
- Preorder traversal
- Postorder traversal

Inorder traversal

- It can be recursively defined as follows.
 1. Traverse the left subtree in inorder.
 2. Process the root node.
 3. Traverse the right subtree in inorder.

- In inorder traversal, move towards the left of the tree(till the leaf node), display that node and then move towards right and repeat the process.
- Since same process is repeated at every stage, recursion will serve the purpose.



Here when we move towards left, we end up in G. G does not have a left child. Now display the root node(in this case it is G). Hence G is displayed first.

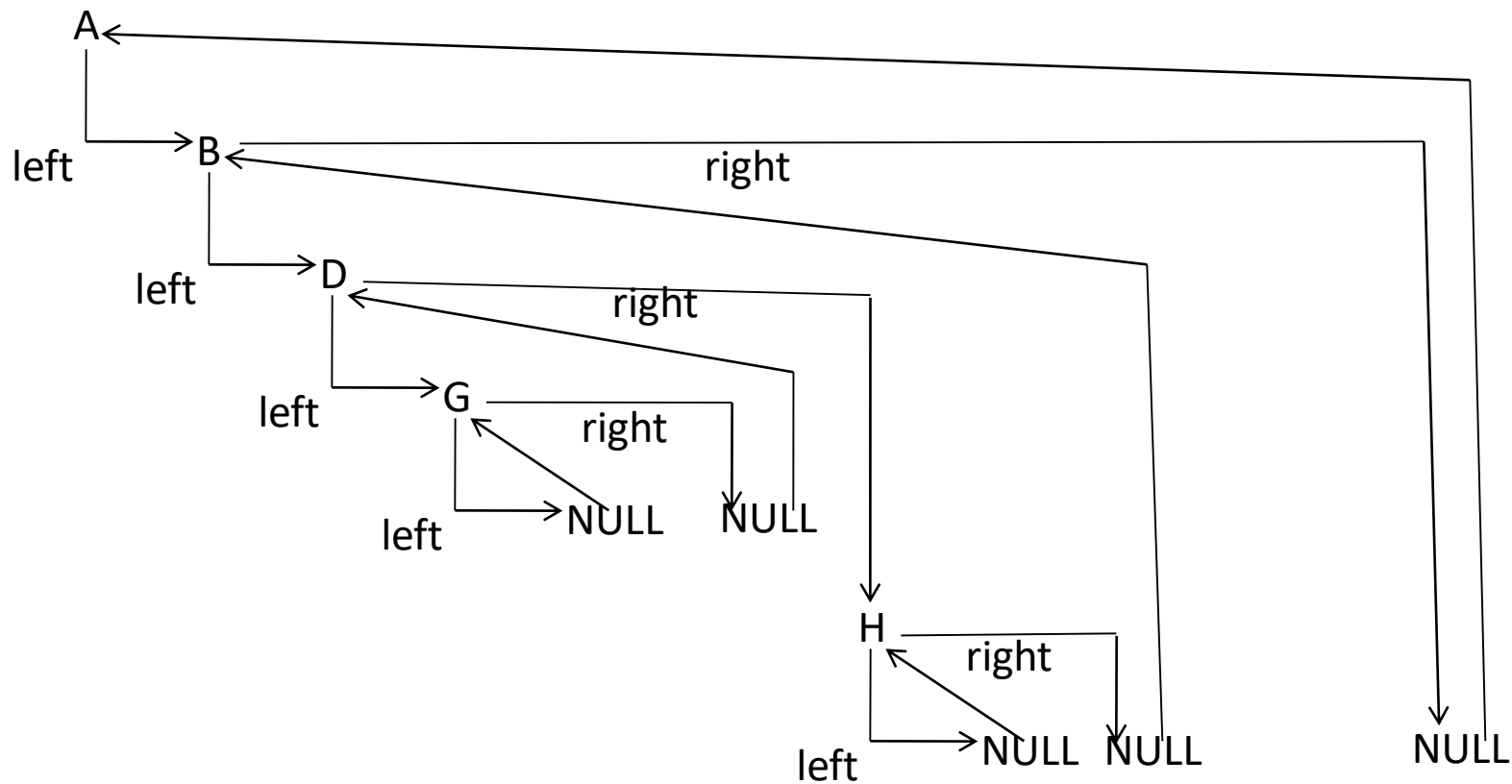
- Now move to the right of G, which is also NULL. Hence go back to root of G and print it. So D is printed next.
- Next go to the right of D, which is H. Now another root H is visited.
- Now move to the left of H, which is NULL. So go back to root H and print it and go to right of H, which is NULL.
- Next go back to the root B and print it and go right of B, which is NULL. So go back to root of B, which is A and print it.
- Now traversing of left is finished and so move towards right of it and reach C.
- Move to the left of C and reach E. Again move to left, which is NULL. Print root E and go to right of E to reach I.

- Move to left of I, which is NULL. Hence go back to root I, print it and move to its right, which is NULL.
- Go back to root C, print it and go to its right and reach F.
- Move to left of F, which is NULL. Hence go back to F, print it and go to its right, which is also NULL.
- Traversal ends at this point.

Hence inorder traversal of above tree gives GDHBAEICF

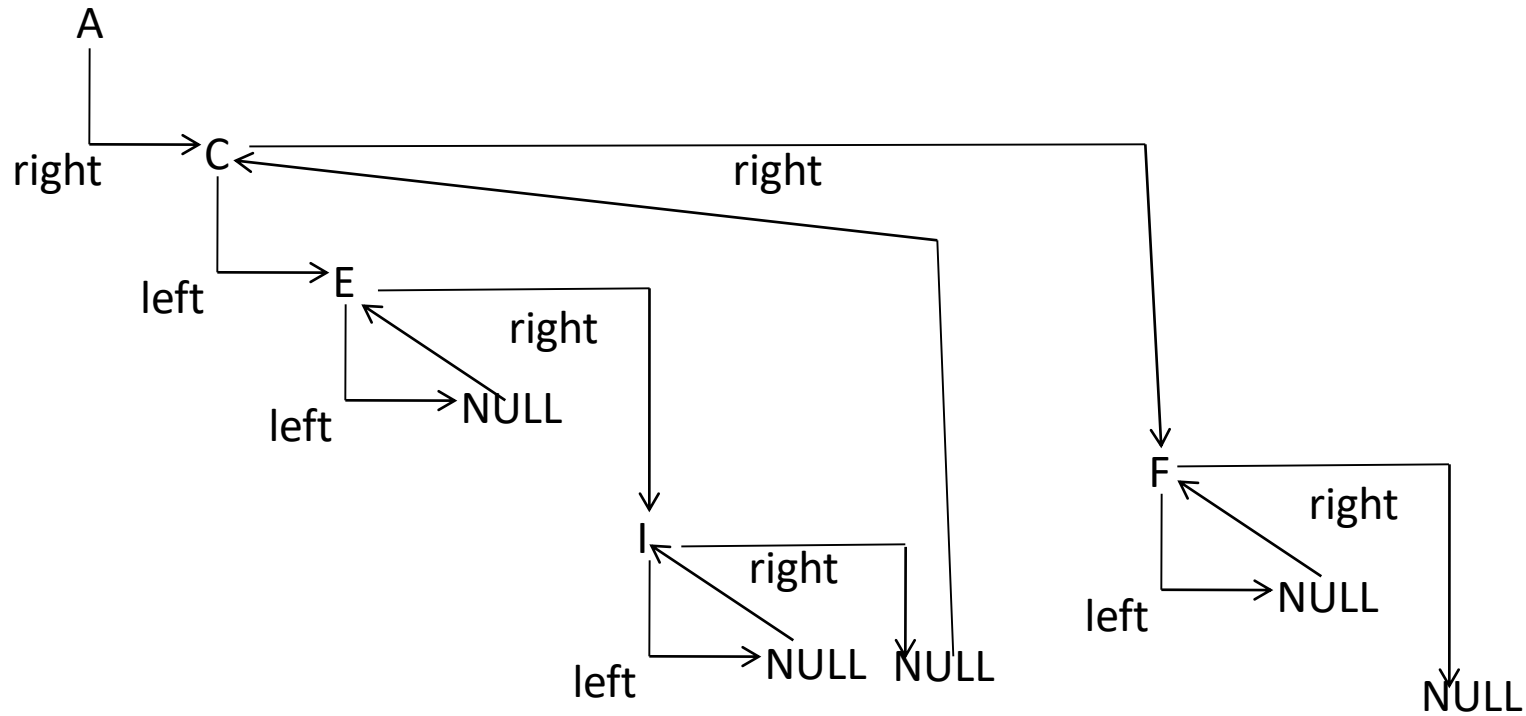
*/*recursive algorithm for inorder traversal*/*

```
Void inorder(NODEPTR root)
{
    if(root!=NULL)
    {
        inorder(root→llink);
        printf("%d", root→info);
        inorder(root→rlink);
    }
}
```



G D H B A

Traversing left subtree of A inorder



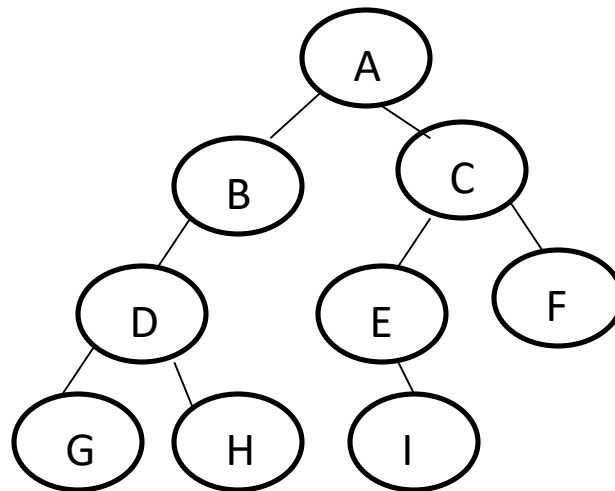
E I C F

Traversing right subtree of A inorder

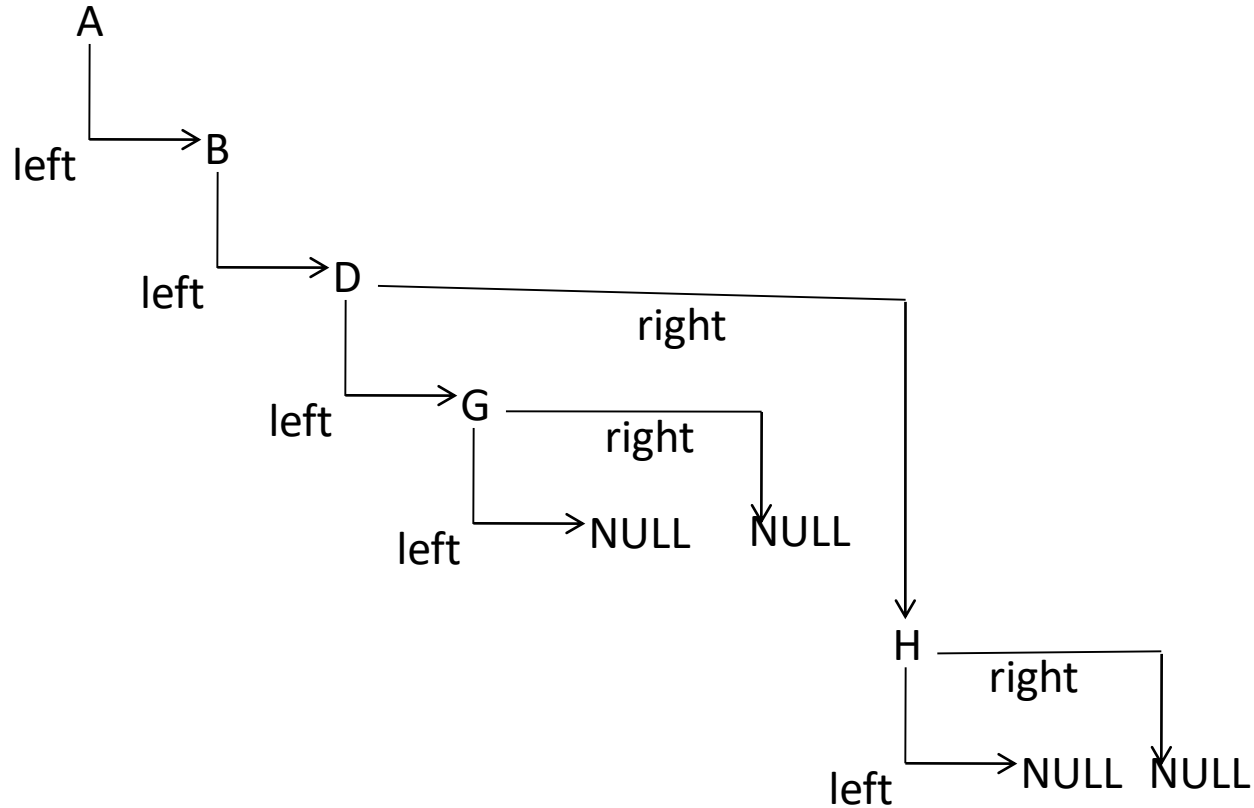
Preorder traversal:

Preorder traversal is defined as

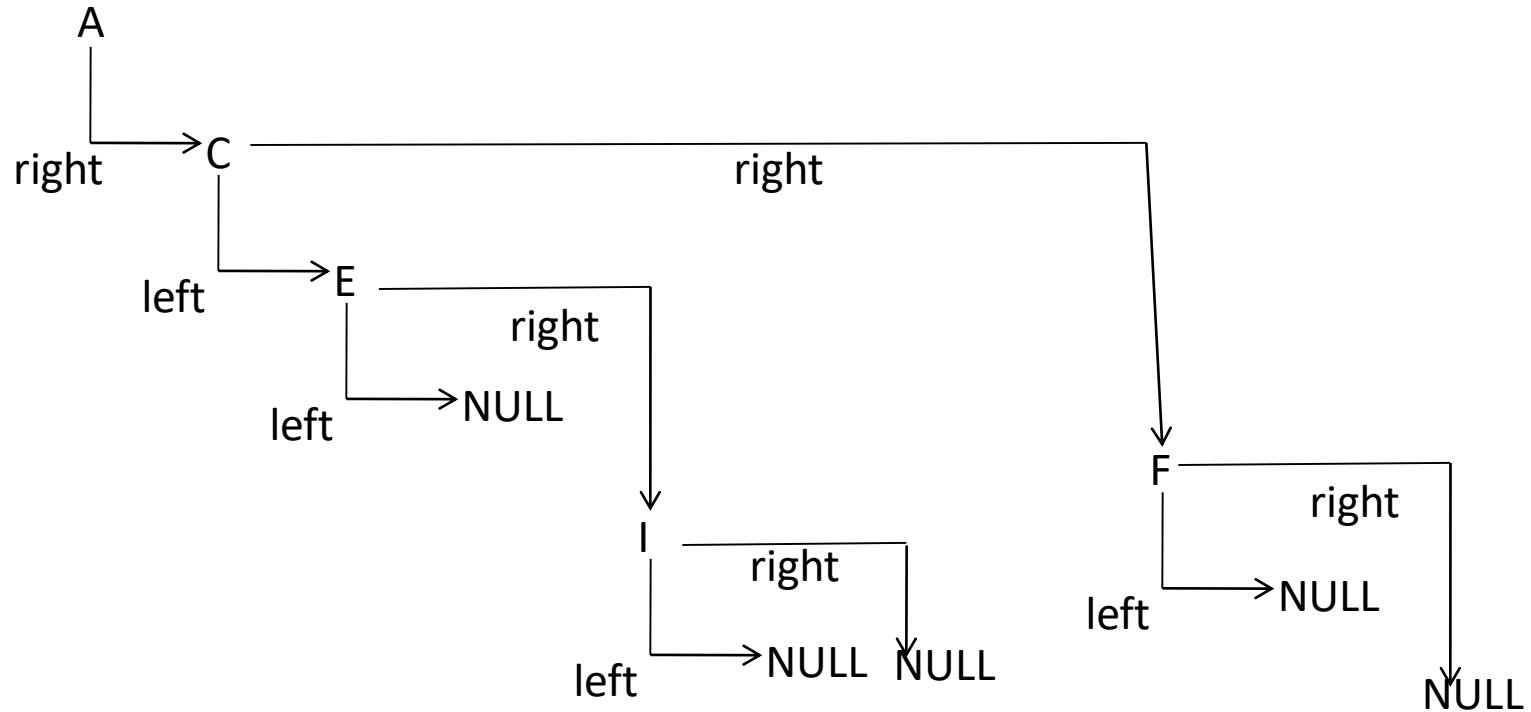
1. Process the node.
 2. Traverse the left subtree in preorder.
 3. Traverse the right subtree in preorder.
- In preorder, we first visit the node, then move towards left and then to the right recursively.



Traversing left subtree in preorder



Traversing right sub tree in preorder



C E I F

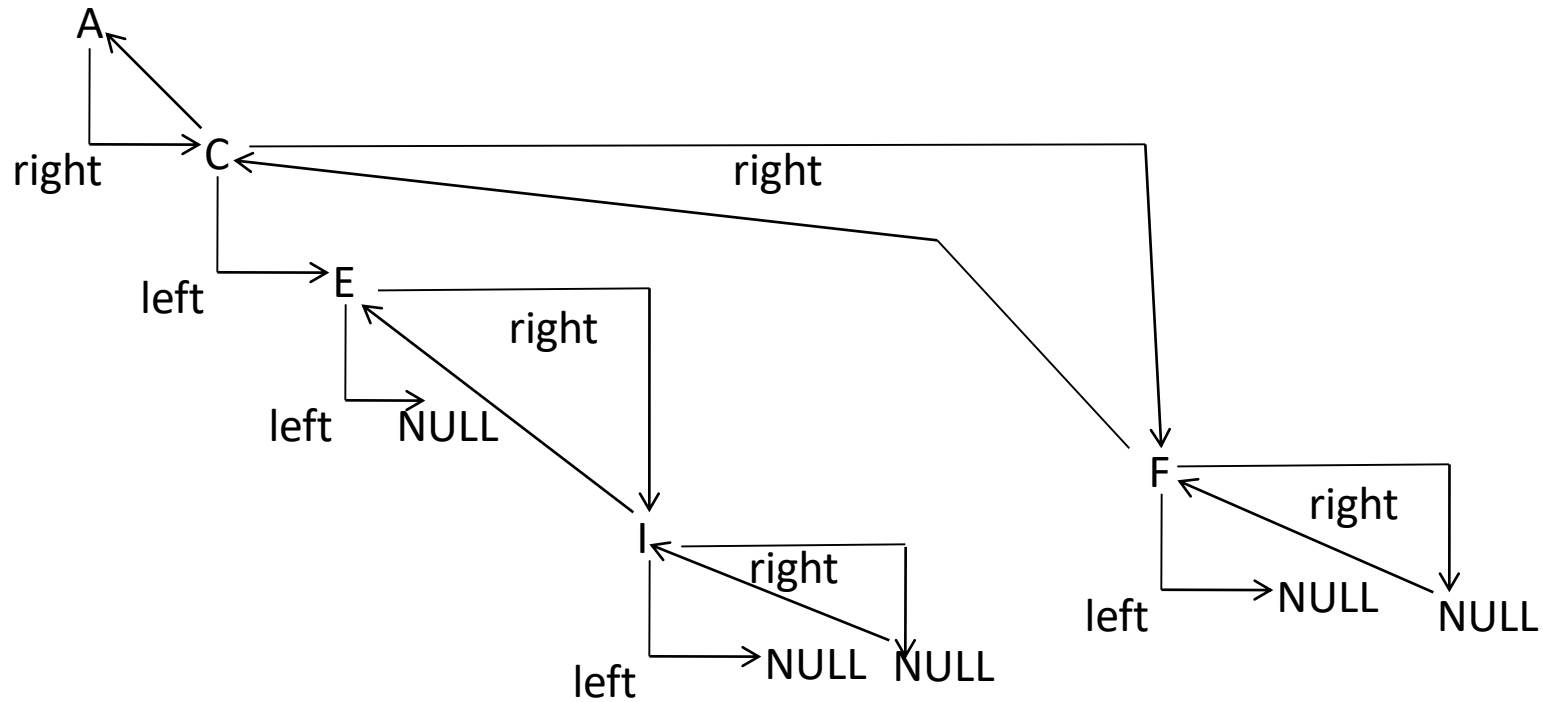
Post order traversal

Post order traversal is defined as

1. Traverse the left subtree in postorder.
 2. Traverse the right subtree in postorder.
 3. Process the root node.
- In post order traversal, we first traverse towards left, then move to right and then visit the root. This process is repeated recursively.



Traversing left subtree of A postorder



I E F C A

Traversing right subtree of A in postorder

Searching:

- Searching an item in the tree can be done while traversing the tree in inorder, preorder or postorder traversals.
- While visiting each node during traversal, instead of printing the node info, it is checked with the item to be searched.
- If item is found, search is successful.

```
Void search(int item, NODEPTR root, int *flag)
```

```
{  
    if(root!=NULL)  
    {  
        search(item, root→llink, flag);  
        if(item==root→info)  
        {  
            *flag=1  
            return;  
        }  
        search(item, root→rlink, flag);  
    }  
}
```

Copying a tree:

- Getting the exact copy of the given tree.

/*recursive function to copy a tree*/

NODEPTR copy (NODEPTR root)

{

 NODEPTR temp;

 if(root == NULL)

 return NULL;

 temp=getnode();

 temp→info=root→info;

 temp→llink=copy(root→llink);

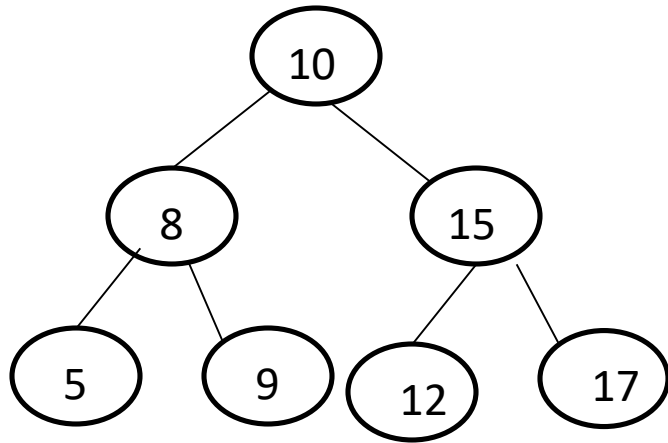
 temp→rlink=copy(root→rlink);

 return temp;

}

Binary search tree(BST)

- Is a special case of binary tree in which for any node, say A, elements in the left subtree are less than info of A and elements in right subtree are greater than or equal to info of A.



Here for any node, elements to its left are less than its info and elements to its right are either greater than or equal to its info.

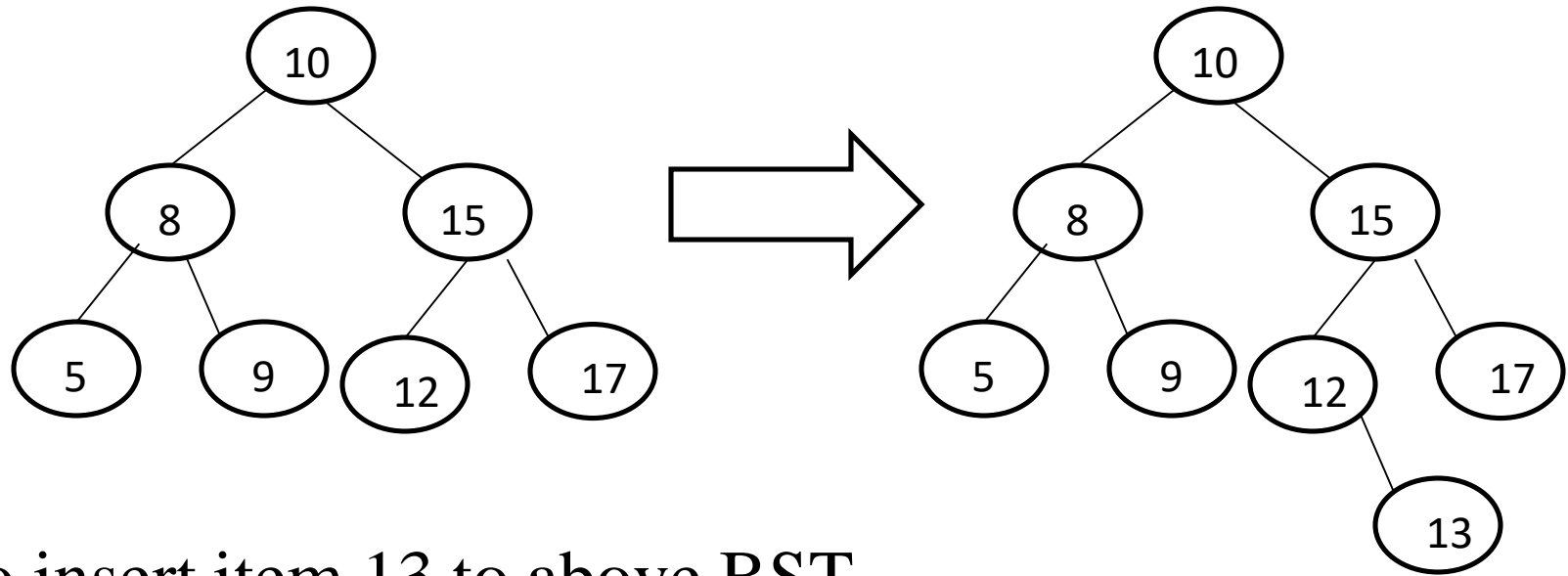
- Inorder traversal of BST results in ascending order of elements

Operations performed on BST

1. Insertion.
2. Search.
3. Deletion.

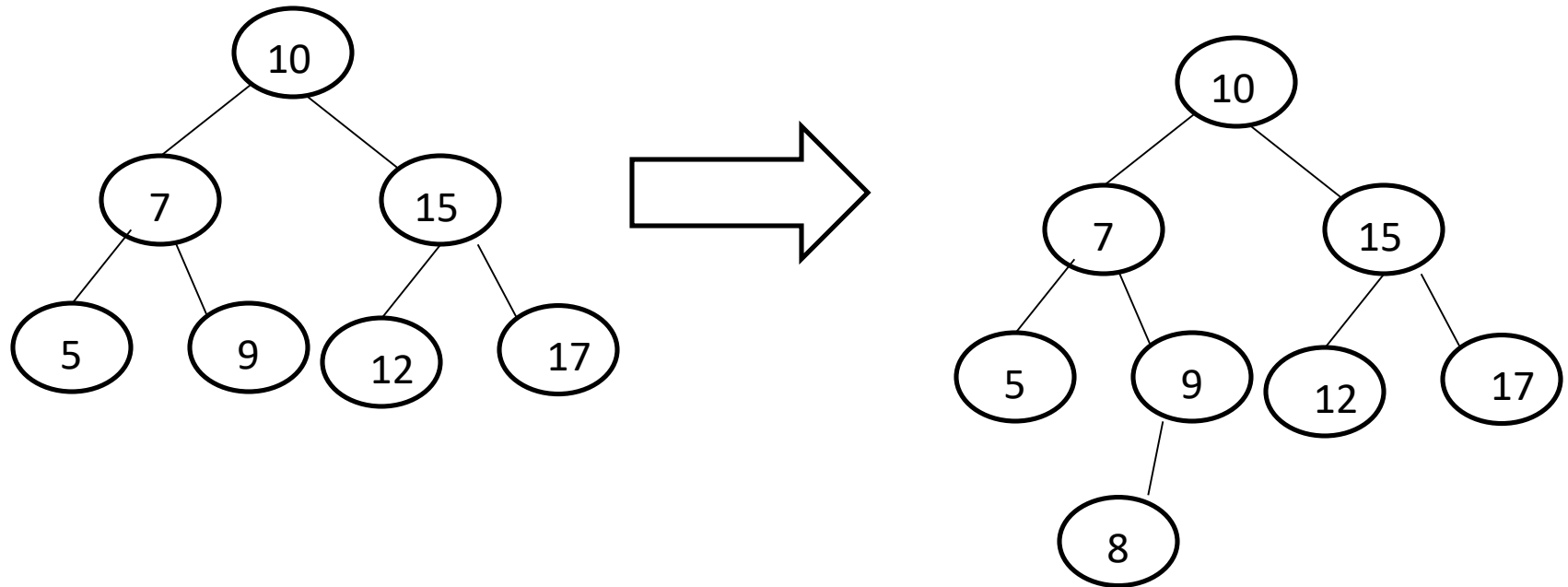
Insertion

- Initially item to be inserted is compared with the root item.
- If item is lesser than the root item, move to left or else move to the right of root node.
- This process is repeated until the correct position is found.



To insert item 13 to above BST

- Compare with the root item. $13 > 10$, hence move to right and reach 15.
- Now $13 < 15$, So go to left and reach 12.
- $13 > 12$, hence move right.
- Now the correct position is found and hence insert the new node to the right of 12.



To insert 8 into the above tree

- Compare with root item. $8 < 10$, hence move left and reach 7.
- Now $8 > 7$. So move right and reach 9.
- $8 < 9$. Move left and the correct position is obtained.
- Insert 8 to the left of 9.

```

/*c program to insert an item into BST(duplicate not allowed)*/
NODEPTR insert (int item, NODEPTR root)
{
    NODEPTR temp, cur, prev;
    temp=getnode();
    temp→info=item;
    temp→llink=temp→rlink=NULL;
    if(root==NULL)
        return temp;
    prev=cur;
    cur=root;
    while(cur!=NULL)                /* traverse until correct position is
                                    found*/
    {
        prev=cur;
        if (item==cur→info)
        {
            printf(“duplicate not allowed”);
            freenode(temp);
            return;
        }
    }
}

```

```
else if(item< cur→info)
    cur=cur→llink;
```

```
else
```

```
    cur=cur→rlink;
```

```
}
```

```
if(item< prev→info)
    prev→llink=temp;
```

```
else
```

```
    prev→rlink=temp;
```

```
return root;
```

```
}
```

```
/*if item< info of current
node, move left*/
```

```
/*else move right*/
```

```
/*end of while*/
```

```
/*if item< parent, insert
new node to its left*/
```

```
/*else insert to its right*/
```

Searching for an item in BST:

- Searching is similar to insertion when it comes to traversal.
- Start from root and compare the item to be searched with the nodes. Hence there are 3 cases
 1. Item is equal to info of node: search is successful.
 2. Item less than the info of node: move to the left of node and continue the process.
 3. Item greater than the info of node: move to the right of node and continue the process.
- The above process is repeated until the search is successful or item not found.

```
/*program for search in BST*/
```

```
Main()
```

```
{
```

```
    int item=50, flag=0;
```

```
    search(item, root, &flag);
```

```
    if(flag==1)
```

```
        printf(“search successful”);
```

```
    else
```

```
        printf(“search is unsuccessful”);
```

```
}
```

```
Void search (int item, NODEPTR root, int *flag)
```

```
{
```

```
    NODEPTR root;
```

```
    temp=root;
```

```
    while(temp!=NULL)
```

```
        /*loop until there are no  
        nodes to be searched*/
```

```
    {
```

```
        if(item==temp→info)
```

```
        {
```

```
            *flag=1;
```

```
            return;
```

```
        }
```

```
    else if(item<temp→info)
```

```
        /*if item is less than info of  
        node, move to left*/
```

```
        temp=temp→llink;
```

```
    else
```

```
        temp=temp→rlink;
```

```
        /*else move right*/
```

```
    }
```

```
}
```

Other operations:

1. Finding the maximum element in BST: maximum element will always be the last right child in a BST. Move to the rightmost node in a BST and you will end up in the maximum element.
2. Finding the minimum element: Move to the leftmost child and you will reach the least element in BST.
3. Finding the height of a tree: height is nothing but maximum level in the tree plus one. It can be easily found using recursion.

/*recursive function to find the height of a tree*/

Int height (NODEPTR root)

{

 if(root==NULL)

 return 0;

 return(1+ max(height (root→llink), height(root→rlink)));

}

/*max function*/

Int max(int a, int b)

{

 if(a>b)

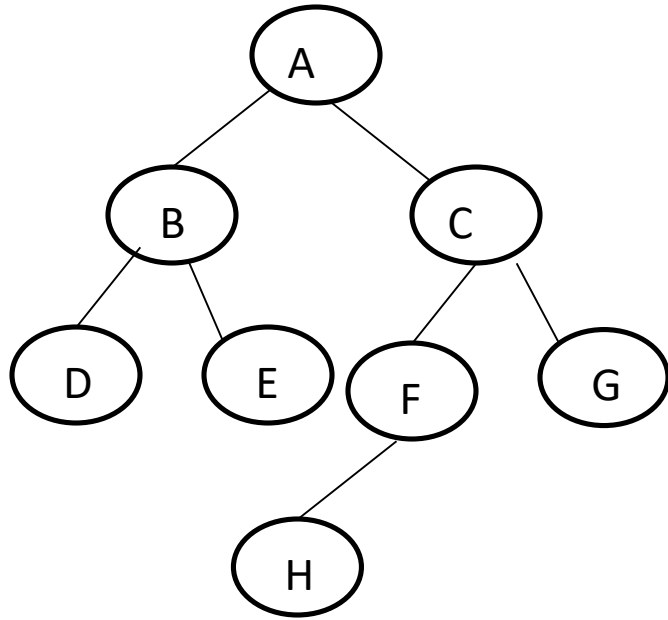
 return a;

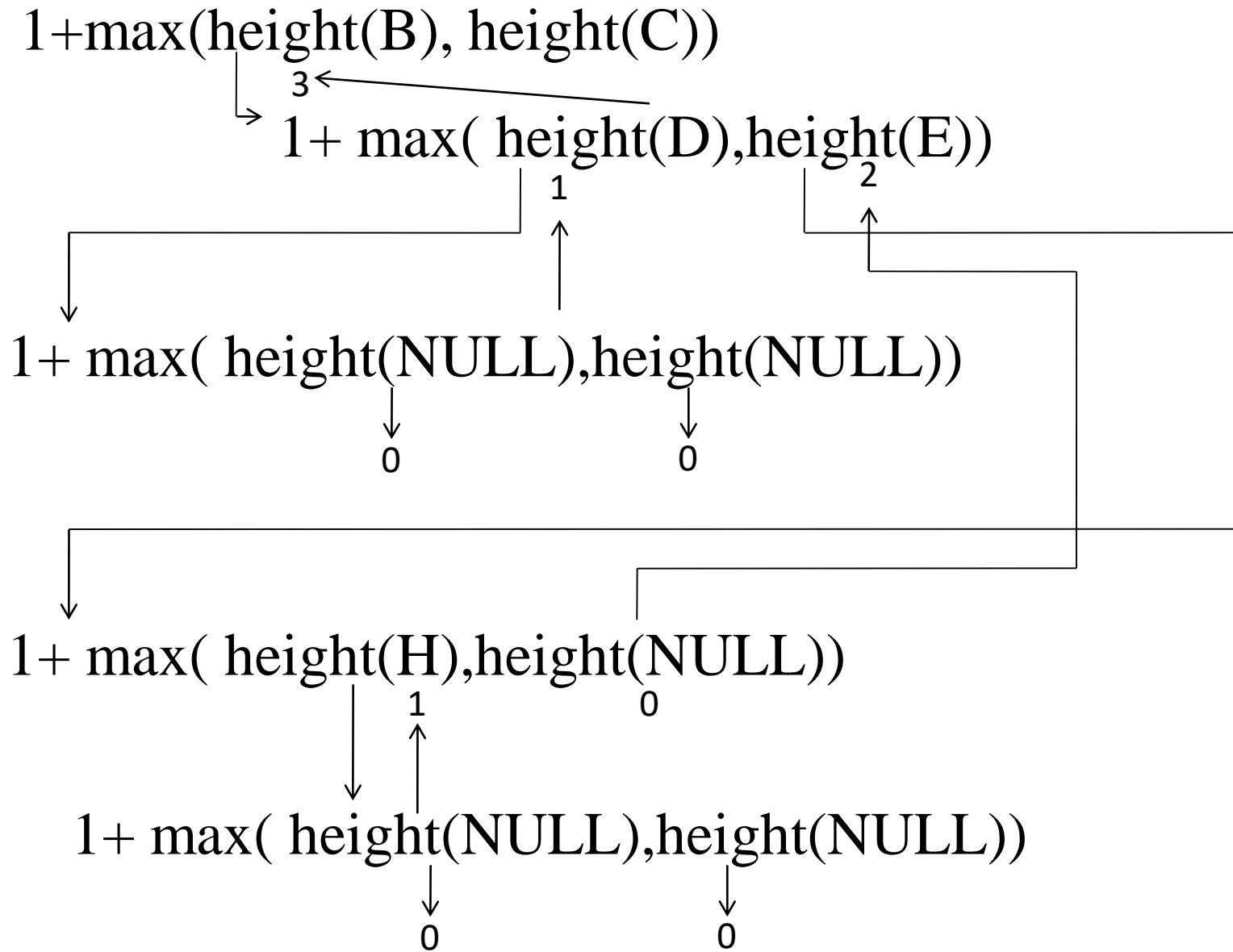
 else

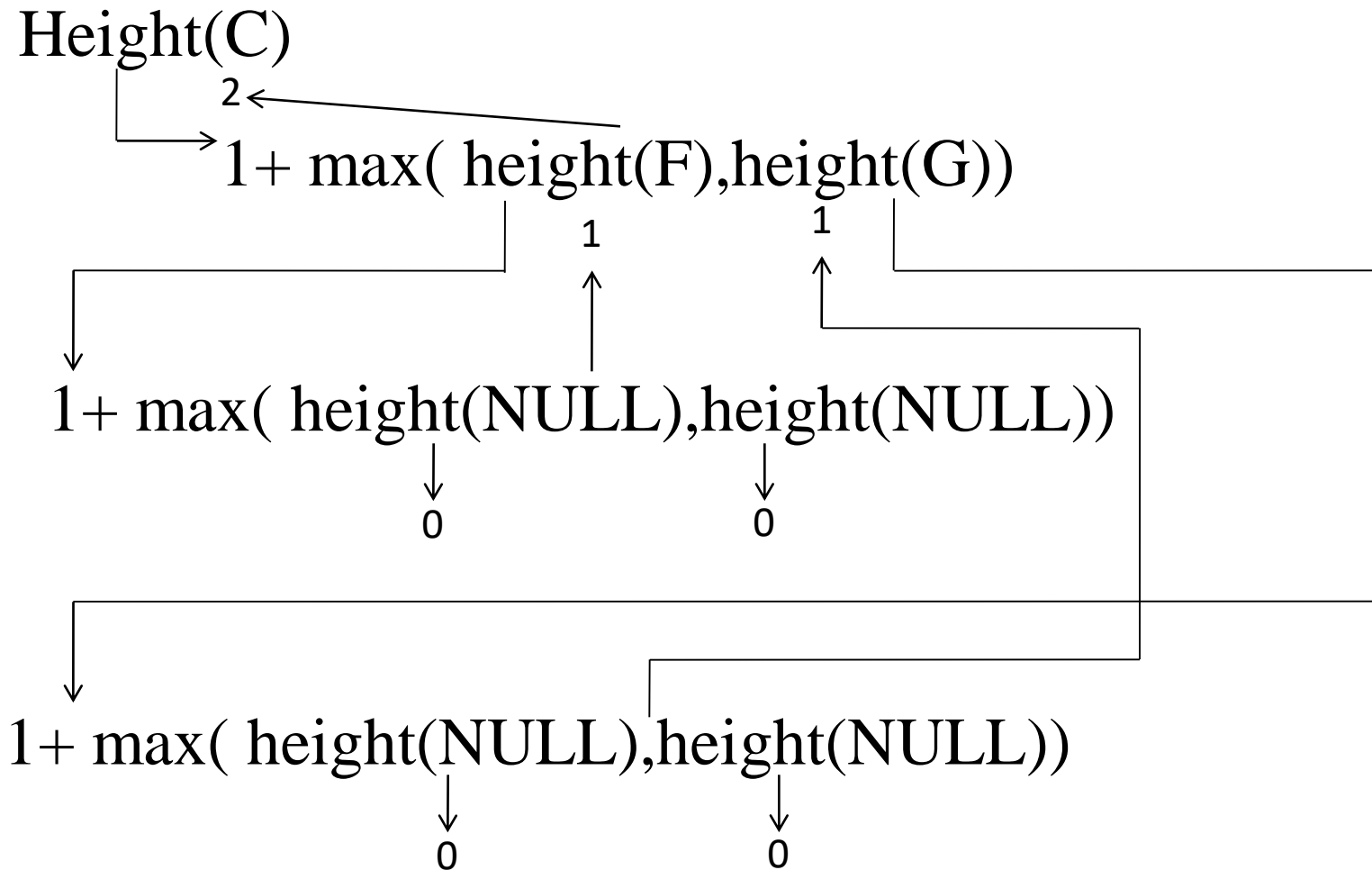
 return b;

}

Finding the height of the tree using recursion







Finally,

$1 + \max(\text{height}(B), \text{height}(C)) \rightarrow 1 + \max(3, 2) \rightarrow 4$, which is the height of the tree.

Counting the number of nodes in a tree:

- Traverse the tree in any of the 3 techniques and every time a node is visited, count is incremented.

*/*counting number of nodes using inorder technique*/*

```
Void count_nodes( NODEPTR root)
```

```
{
```

```
    if(root!=NULL)
```

```
    {
```

```
        count_nodes(root→llink);
```

```
        count++;
```

```
        count_nodes(root→rlink);
```

```
    }
```

```
}
```

Counting the number of leaf nodes in a tree:

- Traverse the tree in any of the 3 techniques and every time a node is visited, check whether the right and left link of that node is NULL. If yes, count is incremented.

/*counting number of leaf nodes using inorder technique*/

```
Void count_leafnodes( NODEPTR root)
```

```
{
```

```
    if(root!=NULL)
```

```
    {
```

```
        count_nodes(root→llink);
```

```
        if(root→llink==NULL && root→rlink==NULL)
```

```
            count++;
```

```
        count_nodes(root→rlink);
```

```
    }
```

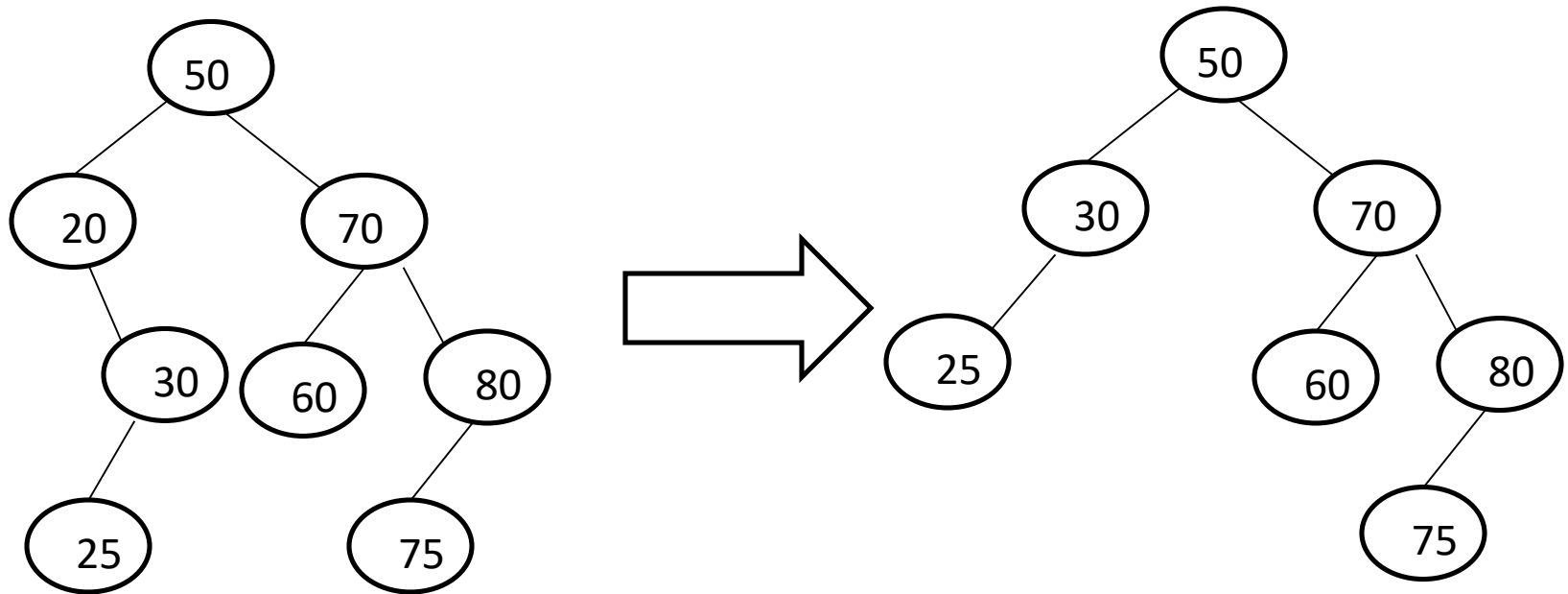
```
}
```

Deleting a node from a BST:

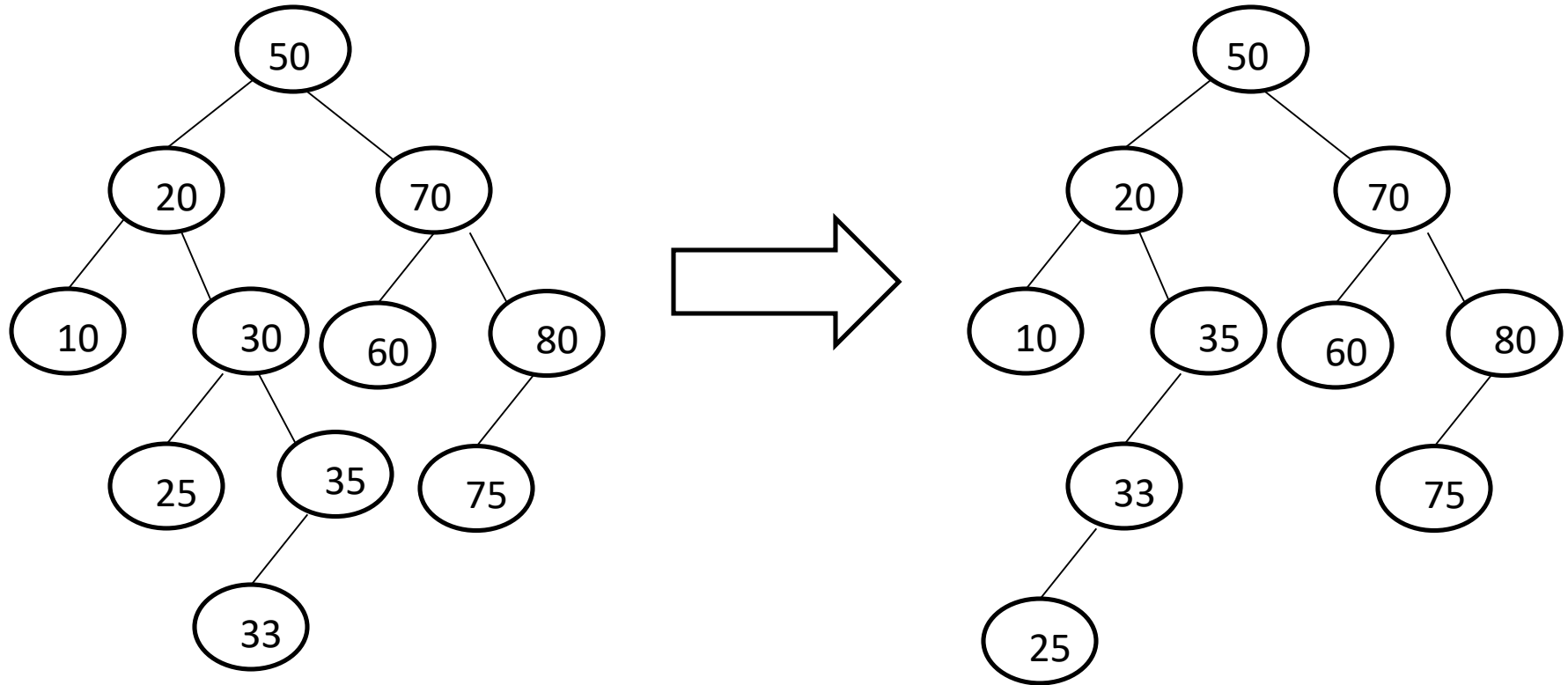
- Search for the item to be deleted by traversing the tree.
- If item not found, appropriate message is printed.
- If item is found, node is deleted and ordering of tree is maintained by adjusting the links.

In the third step, ordering should be maintained means even after deleting the node, elements in the left subtree should be lesser and elements in the right subtree should be higher.

For ex: In the tree given below, after deleting 20(1st case)



After deleting 30 from the below tree(2nd case)

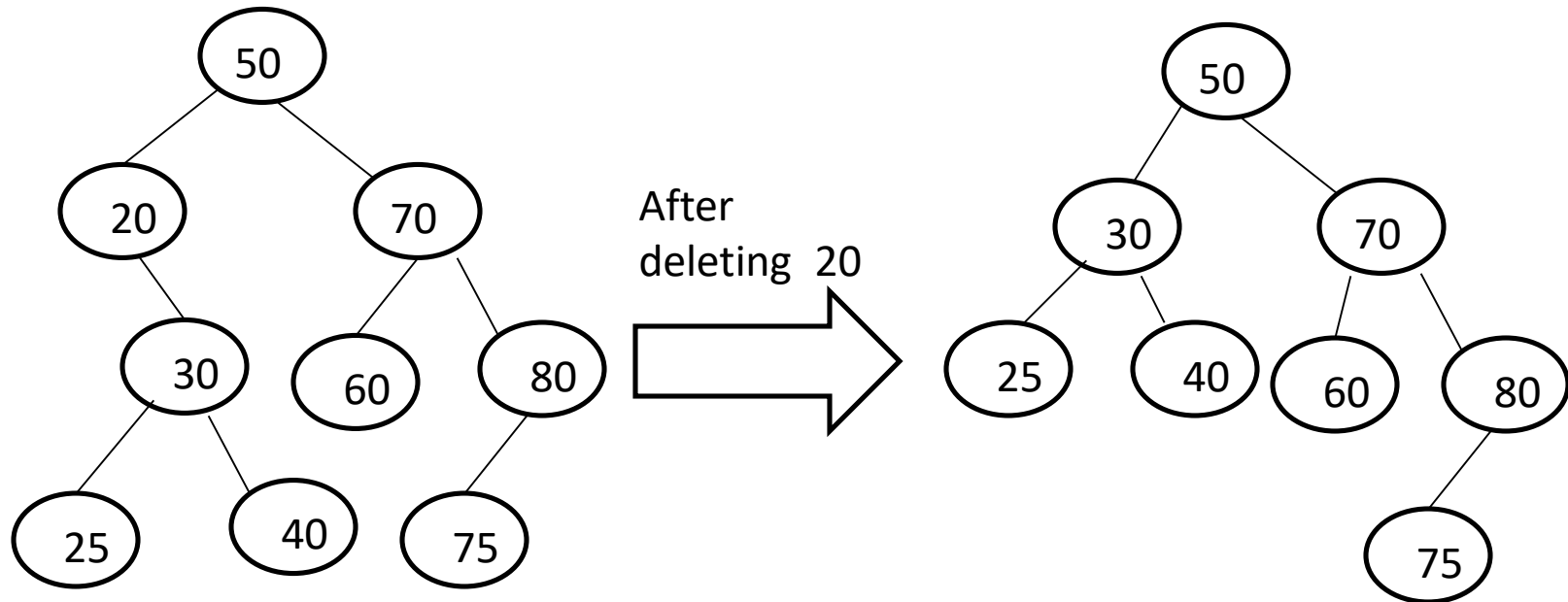


In a BST, node to be deleted will have 2 cases

1. Node to be deleted will have an empty left subtree and non empty right subtree or an empty right subtree and non empty left subtree.(node having empty left child and empty right child is also considered in this case).
2. Node to be deleted is will have non empty left subtree and non empty right subtree.

Case 1:

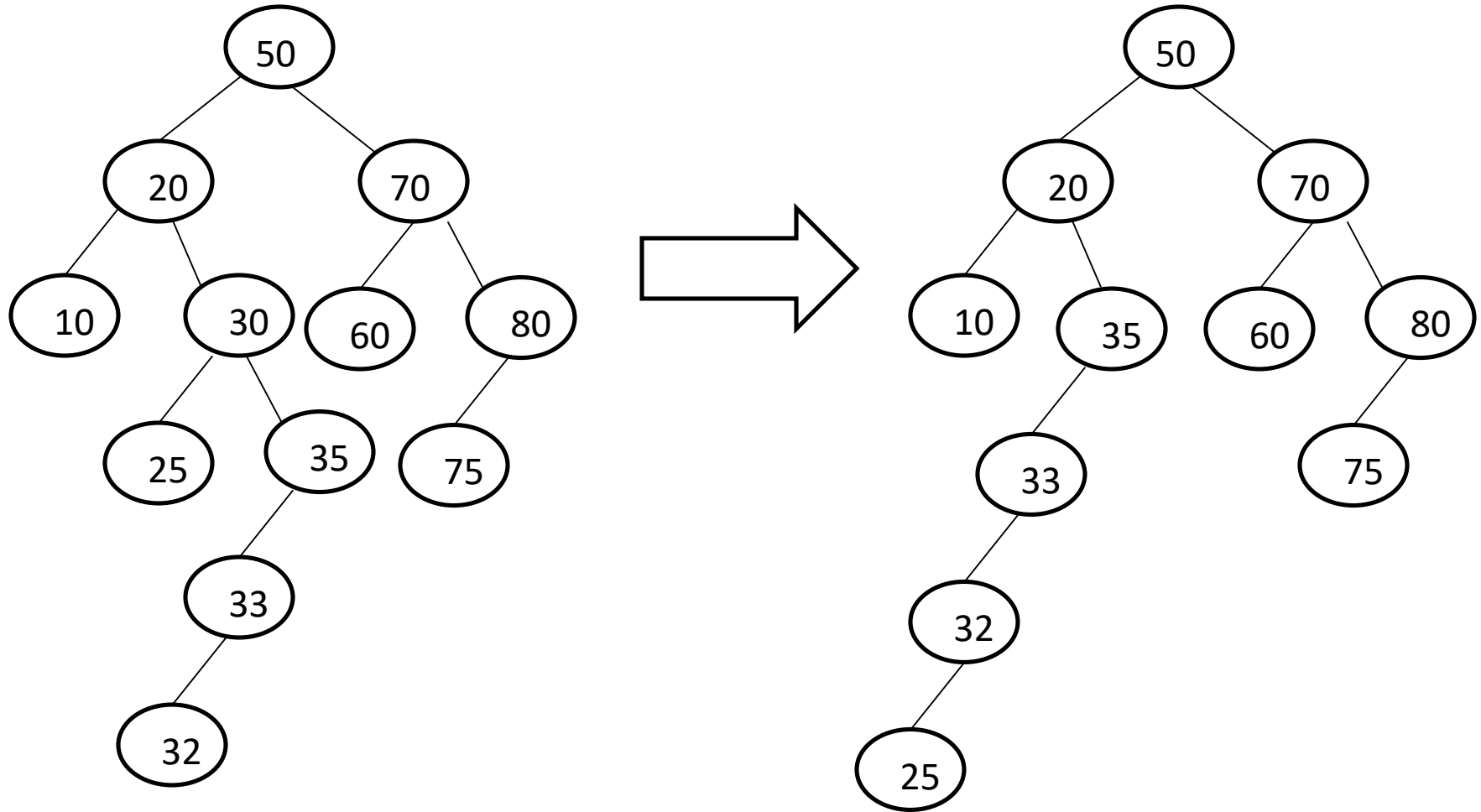
- Simple and straight forward.
- Parent of the node to be deleted is made to point to non empty subtree of the node to be deleted.
- Node to be deleted is freed.



Case 2:

- First move one step right of the node to be deleted.
- Then move to the extreme left of the node just visited in the first step.
- Then left link of the extreme left node visited is made point to the left sub tree of the node to be deleted.
- Finally parent of the node to be deleted is made to point to the node visited in first step.
- The node to be deleted is freed.

After deleting node with item 30



```

/*c function to delete an item from the tree*/
NODEPTR delete_item(int item, NODEPTR root)
{
    NODEPTR cur, parent , suc, q;
    if(root == NULL)
    {
        printf("tree is empty");
        return root;
    }
    /* traverse the tree until item found or not found*/
    parent=NULL;
    cur=root;
    while(cur!=NULL && item!=cur->info)
    {
        parent =cur;          /*keep track of parent node*/
        if(item<cur->info)
            cur=cur->llink;
        else
            cur=cur->rlink;
    }
}

```

```
if(cur==NULL)
{
    printf("item not found");
    return root;
}
/* item found and check for case 1*/
```

```
If(cur→llink==NULL)    /*node to be deleted has empty left
                        subtree. So get the address of right
                        subtree */
    q=cur→rlink;
```

```
else if(cur→rlink==NULL) /*node to be deleted has empty right
                        subtree. So get the address of left
                        subtree */
    q=cur→llink;
```

```
/*case 2*/
```

```
else
```

```
{
```

```
    suc=cur→rlink;          /*get the address of the right node  
                             of the node to be deleted*/
```

```
    while(suc→llink!=NULL)
```

```
        suc=suc→llink;      /*then move to the leftmost node  
                             of the node visited in previous  
                             step*/
```

```
    suc→llink=cur→llink; /* attach left subtree of node to be  
                           deleted to the left of the leftmost  
                           node reached in previous step*/
```

```
    q=cur→rlink;           /*obtain address of the right  
                           subtree of the node to be  
                           deleted*/
```

```
}
```



```
if(parent == NULL)          /* if parent does not exist, return
    return q;                q as root*/
if(cur==parent→llink)        /* if root to be deleted is left of
    parent→llink=q;          parent, connect left of parent to
                              q*/
else
    parent→rlink=q;          /*else connect right of parent to q

free(cur);
return root;
}
```

BST as a priority queue:(ascending priority queue)

- Insert the elements into the BST as already discussed.
- While deleting, always delete the leftmost element in a tree.
- Display the tree using inorder traversal.

BST as a priority queue:(descending priority queue)

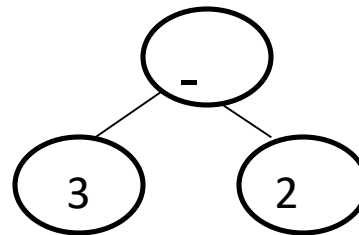
- Insert the elements into the BST as already discussed.
- While deleting, always delete the rightmost element in a tree.
- Display the tree using inorder traversal.

Applications of binary trees

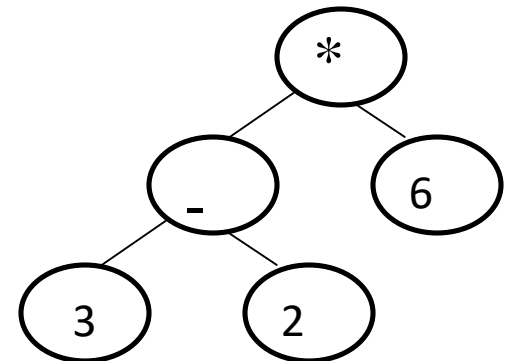
Conversion of expressions:

- An infix expression consisting of operators and operands can be represented using a binary tree with root as operator.
- Non leaf node contains the operator and leaf nodes contain operands.

$(3-2)$ can be represented as



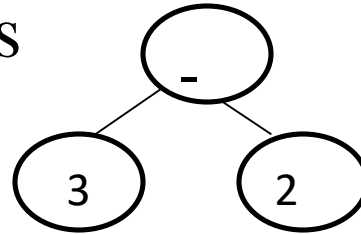
$((3-2)*6)$ can be represented as



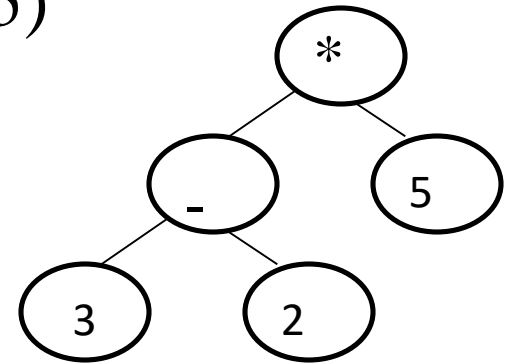
Represent the following expression using binary tree

$$((6+(3-2)*5)^2)$$

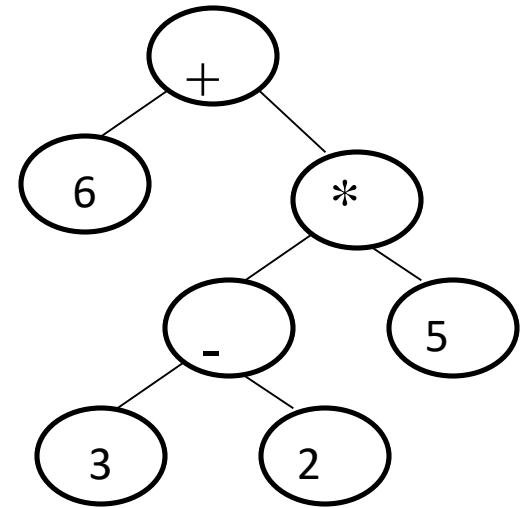
First innermost parenthesis is considered, which is $(3-2)$ and partial tree is drawn for this



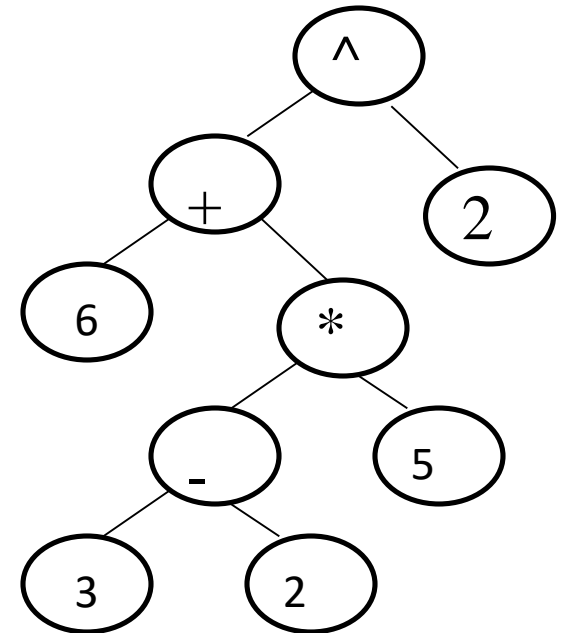
Next tree is extended to include $((3-2)*5)$



After considering $(6+(3-2)*5)$



After $(6+(3-2)*5)^2$



- When the binary tree for infix expression is traversed in inorder technique, infix expression is obtained.
- Preorder traversal gives the prefix expression and postorder traversal gives the postorder expression.

Inorder traversal of above tree: $(6+(3-2)*5)^2$

Preorder traversal: $^+6*-3252$

Postorder traversal: $632-5*+2^$

Creating a binary tree for postfix expression:

Steps:

1. Scan the expression from left to right.
2. Create a node for each symbol encountered.
3. If the symbol is an operand, push the corresponding node on to the stack.
4. If the symbol is an operator, pop top node from stack and attach it to the right of the node with the operator. Next pop present top node and attach it to the left of node with the operator. Push the operator node to the stack.
5. Repeat the process for each symbol in postfix expression. Finally address of root node of expression tree is on top of stack.

```

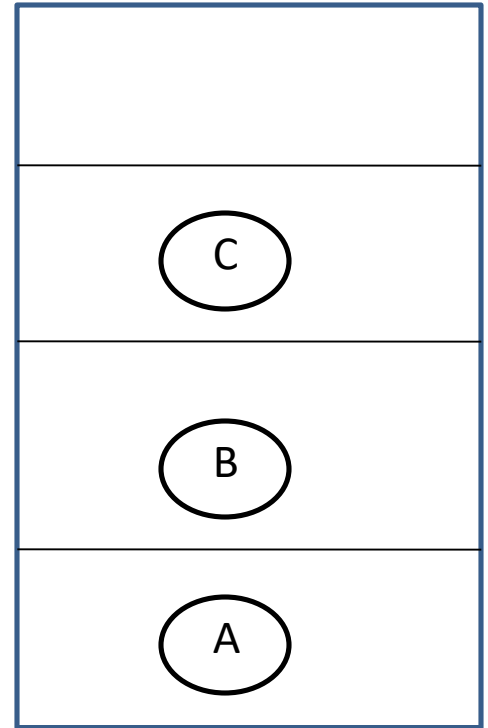
/* C function to create a binary tree for postfix expression*/
NODEPTR create_tree( char postfix[])
{
    NODEPTR temp;
    NODEPTR stack[20];
    int i=0, k=0;
    char symbol;
    while(symbol=postfix[i]!='\0')
    {
        temp=getnode();
        temp->info=symbol;
        temp->llink=temp->rlink=NULL;
        if(isalnum(symbol))                /* if operand push it*/
            stack[k++]=temp;
        else                                /* else pop element add it to the
                                           right of operator node. Pop
                                           temp->llink=stack[--k]; next element and add to
                                           temp->rlink=stack[--k]; left. Push operator node*/
                                           stack[++k]=temp
    }
    return(stack[--k])                    /* return root*/
}

```

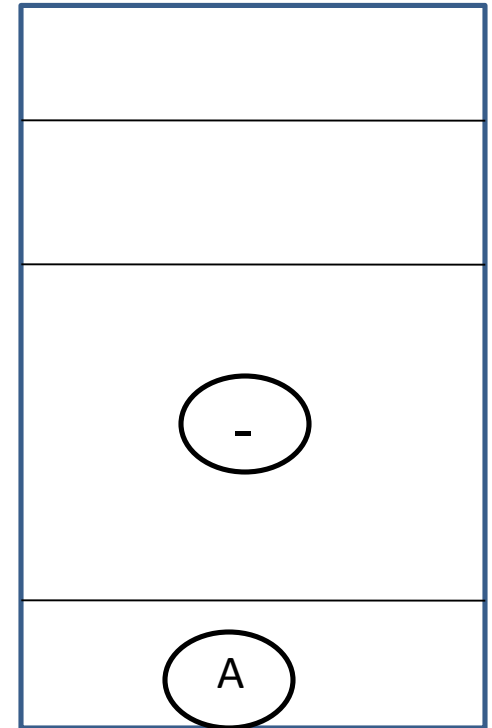
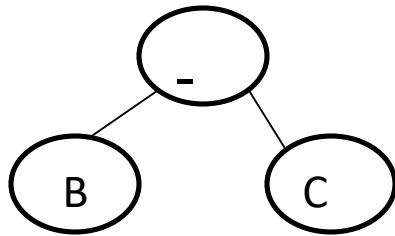

Create a binary tree for the given postfix expression:

abc-d*+

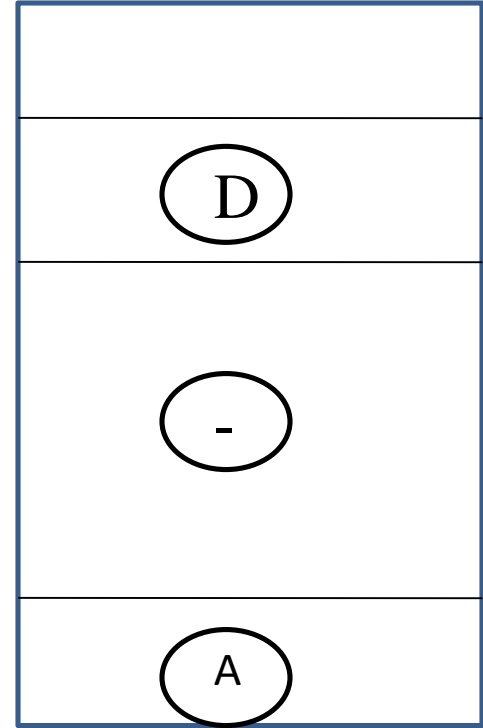
1. First 3 symbols are operands, hence after pushing these 3 symbols, stack of nodes looks like



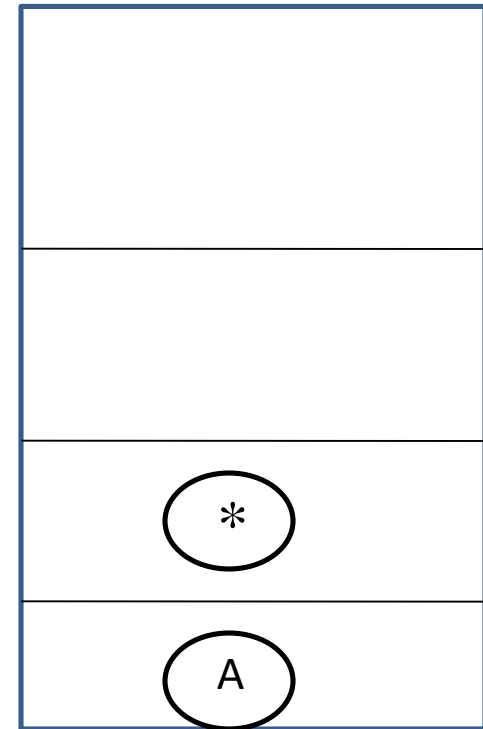
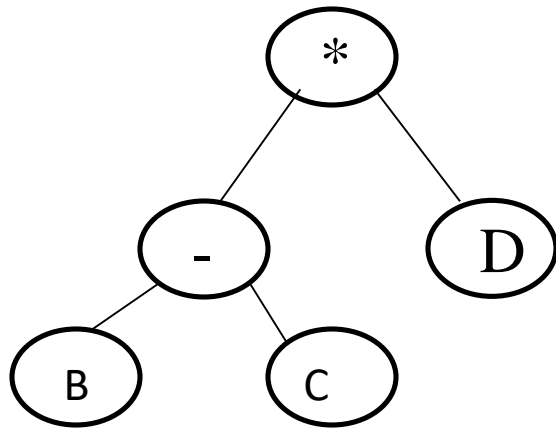
2. Now we get operator '-'. Pop top 2 elements and add them to right and left of node with '-' respectively and push node with operator '-' to stack.



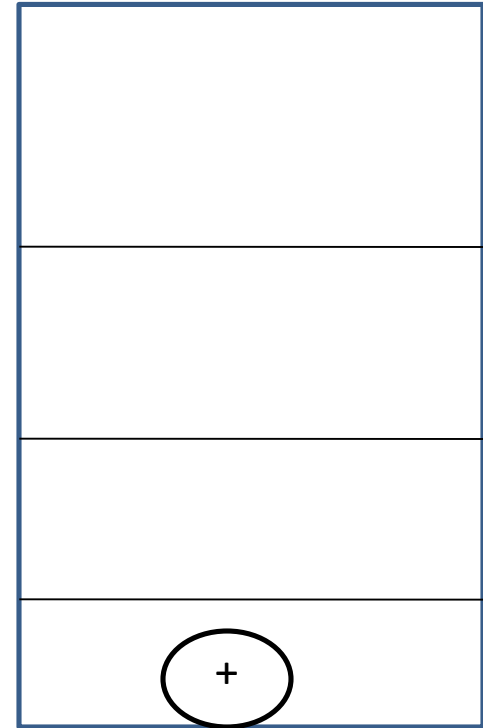
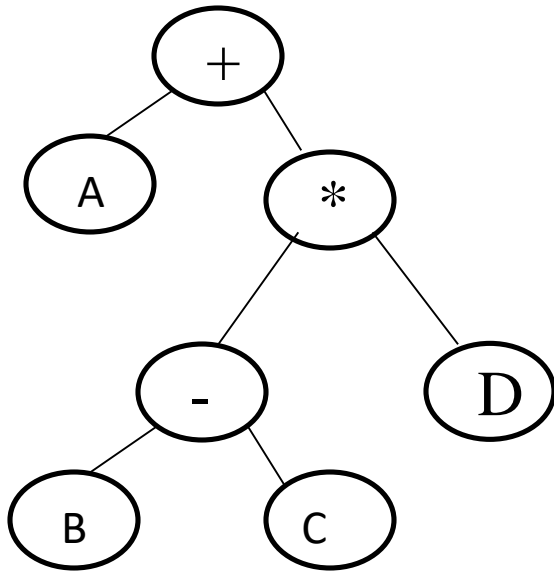
3. Push symbol 'D' to stack



4. Now the operator is '*'. Pop top 2 elements and add it to right and left of node with '*' respectively and push the operator node to stack.



5. Next is the operator '+'. Hence after popping and pushing, stack will be



Now stack top will have the root of the final tree.

Evaluating the expression tree using recursion:

```
float eval( NODEPTR root)
```

```
{
```

```
    float num;
```

```
    switch(root→info)
```

```
    {
```

```
        case '+':return eval(root→llink)+ eval(root→rlink);
```

```
        case '-':return eval(root→llink)- eval(root→rlink);
```

```
        case '/':return eval(root→llink)/ eval(root→rlink);
```

```
        case '*':return eval(root→llink)* eval(root→rlink);
```

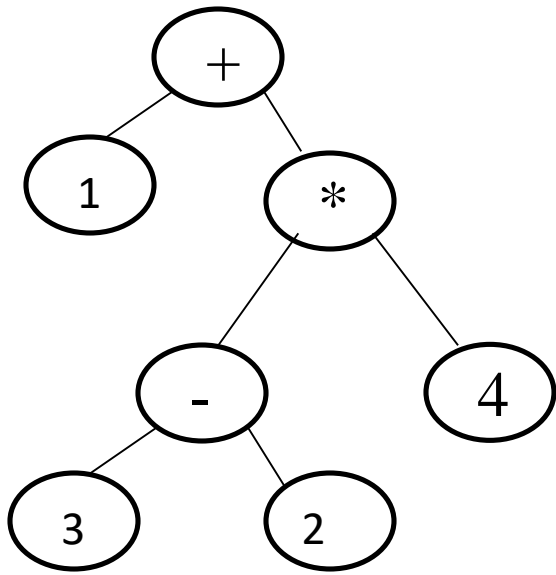
```
        case '$':
```

```
        case '^':return pow(eval(root→llink), eval(root→rlink));
```

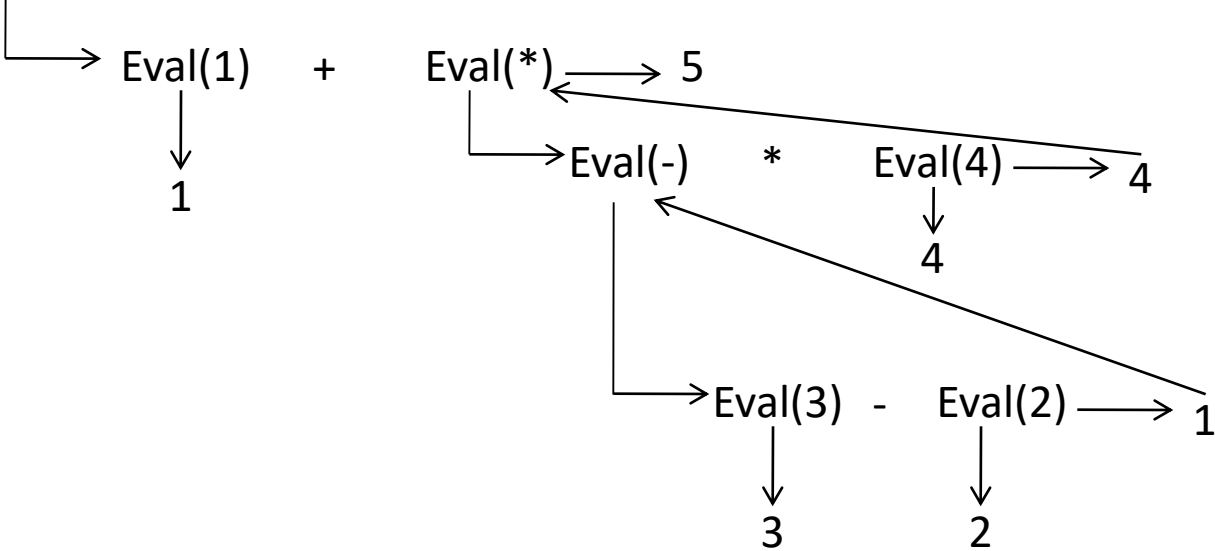
```
        default :return(root→info);
```

```
    }
```

```
}
```



Eval(+)



Printing Arithmetic Expressions(inorder)

- print operand or operator when visiting node
- print “(“ before traversing left subtree
- print “)” after traversing right subtree

Algorithm *printExpression(v)*

if isInternal (v)

print("(")

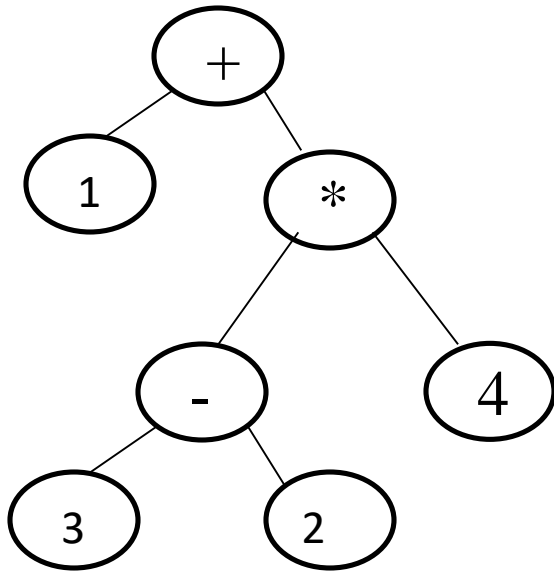
inOrder (leftChild (v))

print(v.element ())

if isInternal (v)

inOrder (rightChild (v))

print (")")

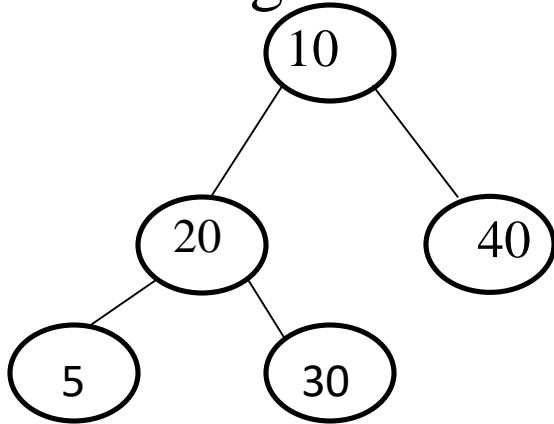


$(1+((3-2)*4))$

Iterative traversals of binary tree

Iterative preorder traversal:

- Every time a node is visited, its info is printed and node is pushed to stack, since the address of node is needed to traverse to its right later.



Here 10 is printed and this node is pushed onto stack and then move left to 20. This is done because we need to go right of 10 later.

Similarly 20, 5, 30 and 40 are moved to stack

- Once traversing the left side is over, pop the most pushed node and go to its right.

In the previous tree, after 5 is printed, recently pushed node 20 is popped and move right to 30.

```
/*function for iterative preorder traversal*/
```

```
Void preorder(NODEPTR root)
```

```
{
```

```
    NODEPTR cur;
```

```
    NODEPTR stack[20];
```

```
    int top=-1;
```

```
    if(root==NULL)
```

```
    {
```

```
        printf("tree is empty");
```

```
        return;
```

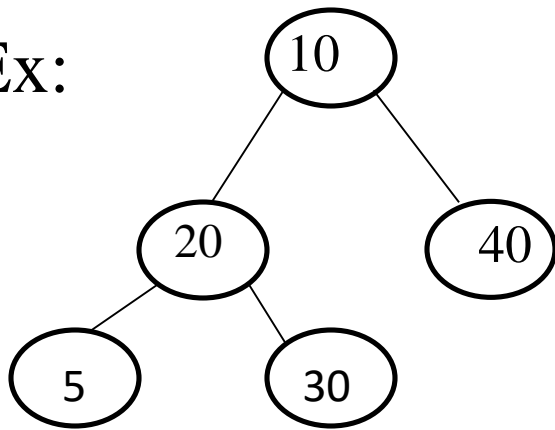
```
    }
```

```

cur=root;
for(; ;)
{
    while(cur!=NULL)
    {
        printf("%d", cur→info);
        push(cur,&top,s)           /*push the node to stack*/
        cur=cur→llink;
    }
    if(!stack_empty(top))         /*more nodes existing*/
    {
        cur=pop(&top,s);           /* pop most recent node*/
        cur=cur→rlink;             /*traverse right*/
    }
    else return;
}}

```

Ex:

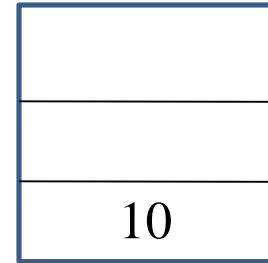


1. After 1st iteration of while loop

10 is printed

Node 10 is pushed to stack

Cur=cur→llink; i.e Cur=20

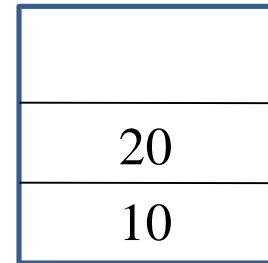


2. After 2nd iteration of while loop

20 is printed

Node 20 is pushed to stack

Cur=cur→llink; i.e Cur=5



3. After 3rd iteration of while loop

5 is printed

Node 5 is pushed to stack

Cur=cur→llink; i.e Cur=NULL

5
20
10

4. While loop terminates since cur==NULL

Stack is not empty

cur=pop(); i.e cur=node 5;

cur=cur→rlink; i.e cur=NULL

20
10

5. cur==NULL, while loop not entered

Stack not empty

cur=pop(); i.e cur=node 20

cur=cur→rlink; i.e cur=30

10

6. While loop is entered

30 is printed

Node 30 is pushed to stack

Cur=cur→llink; i.e cur=NULL

30
10

7. While loop terminates since cur==NULL

Stack not empty

cur=pop(); i.e cur=node 30;

cur=cur→rlink; i.e cur=NULL

10

8. cur==NULL, while loop not entered

Stack not empty

cur=pop(); i.e cur=node 10

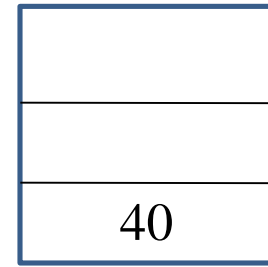
cur=cur→rlink; i.e cur=40

9. While loop is entered

40 is printed

Node 40 is pushed to stack

cur=cur→llink; i.e cur=NULL

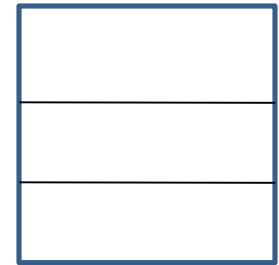


10. While loop terminates since cur==NULL

Stack not empty

cur=pop(); i.e cur=node 40;

cur=cur→rlink; i.e cur=NULL



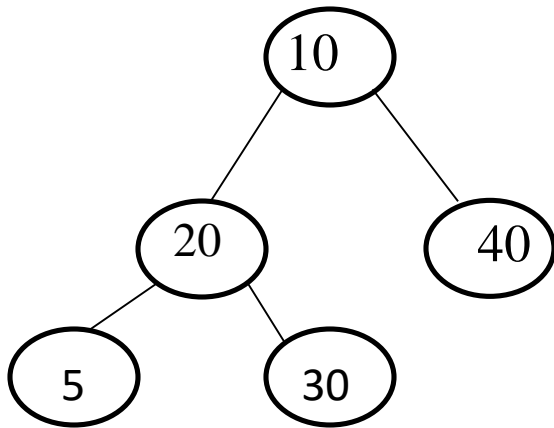
11. cur==NULL and stack empty

return

Hence elements printed are 10, 20, 5, 30, 40

Iterative inorder traversal:

- Every time a node is visited, it is pushed to stack without printing its info and move left.
- After finishing left, pop element from stack, print it and move right.



Here 10, 20 , 5 is pushed to stack. Then pop 5, print it and move right.
Now pop 20, print it and move right and push 30 and move left.
Pop 30, print it and move right.
Pop 10, print it and move right and push 40 and so on

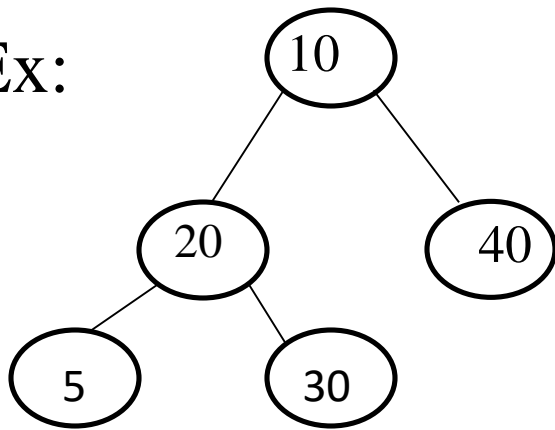
```
/*function for iterative inorder traversal*/  
Void inorder(NODEPTR root)  
{  
    NODEPTR cur;  
    NODEPTR stack[20];  
    int top=-1;  
    if(root==NULL)  
    {  
        printf(“tree is empty”);  
        return;  
    }  
}
```

```

cur=root;
for(; ;)
{
    while(cur!=NULL)
    {
        push(cur,&top,s)           /*push the node to stack*/
        cur=cur→llink;
    }
    if(!stack_empty(top))          /*more nodes existing*/
    {
        cur=pop(&top,s);           /* pop most recent node*/
        printf(“%d”, cur→info);
        cur=cur→rlink;            /*traverse right*/
    }
    else return;
}}

```

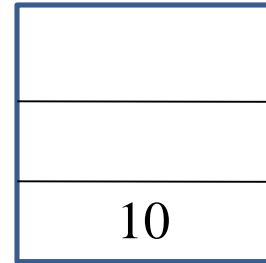
Ex:



1. After 1st iteration of while loop

Node 10 is pushed to stack

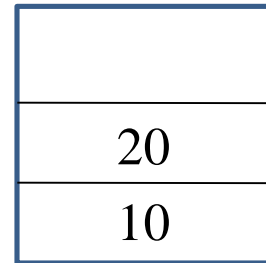
Cur=cur→llink; i.e Cur=20



2. After 2nd iteration of while loop

Node 20 is pushed to stack

Cur=cur→llink; i.e Cur=5



3. After 3rd iteration of while loop

Node 5 is pushed to stack

Cur=cur→llink; i.e Cur=NULL

5
20
10

4. While loop terminates since cur==NULL

Stack is not empty

cur=pop(); i.e cur=node 5;

Print 5

cur=cur→rlink; i.e cur=NULL

20
10

5. cur==NULL, while loop not entered

Stack not empty

cur=pop(); i.e cur=node 20

Print 20

cur=cur→rlink; i.e cur=30

10

6. While loop is entered

Node 30 is pushed to stack

Cur=cur→llink; i.e cur=NULL

30
10

7. While loop terminates since cur==NULL

Stack not empty

cur=pop(); i.e cur=node 30;

Print 30

cur=cur→rlink; i.e cur=NULL

10

8. cur==NULL, while loop not entered

Stack not empty

cur=pop(); i.e cur=node 10

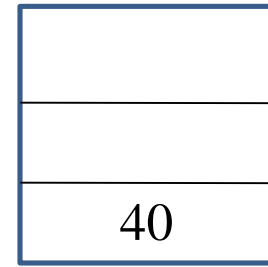
Print 10

cur=cur→rlink; i.e cur=40

9. While loop is entered

Node 40 is pushed to stack

cur=cur→llink; i.e cur=NULL



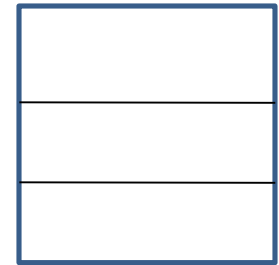
10. While loop terminates since cur==NULL

Stack not empty

cur=pop(); i.e cur=node 40;

Print 40

cur=cur→rlink; i.e cur=NULL



11. cur==NULL and stack empty

return

Hence elements printed are: 5 20 30 10 40

Iterative postorder traversal

- Here a flag variable to keep track of traversing. Flag is associated with each node. Flag== -1 indicates that traversing right subtree of that node is over.

Algorithm:

- Traverse left and push the nodes to stack with their flags set to 1, until NULL is reached.
- Then flag of current node is set to -1 and its right subtree is traversed. Flag is set to -1 to indicate that traversing right subtree of that node is over.
- Hence if flag is -1, it means traversing right subtree of that node is over and you can print the item. if flag is not -ve, traversing right is not done, hence traverse right.

```
/*function for iterative postorder traversal*/
```

```
Void postorder(NODEPTR root)
```

```
{  
    struct stack  
    {  
        NODEPTR node;  
        int flag;  
    };  
    NODEPTR cur;  
    struct stack s[20];  
    int top=-1;  
    if(root==NULL)  
    {  
        printf(“tree is empty”);  
        return;  
    }  
}
```

```

cur=root;
for(; ;)
{
    while(cur!=NULL)
    {
        s[++top].node=cur;
        s[top].flag=1;
        cur=cur→llink;
    }
    while(s[top].flag<0)
    {
        cur=s[top--].address;
        printf(“%d”, cur→info);

        if(stack_empty(top))
            return;
    }
}

```

/*traverse left of tree and push the nodes to the stack and set flag to 1*/

/*if flag is -ve, right subtree is visited and hence node is popped and printed*/

/*if stack is empty, traversal is complete*/

```
cur= s[top].node;
```

```
cur=cur→rlink;
```

```
s[top].flag=-1;
```

```
/*after left subtree is
```

```
traversed, move to right
```

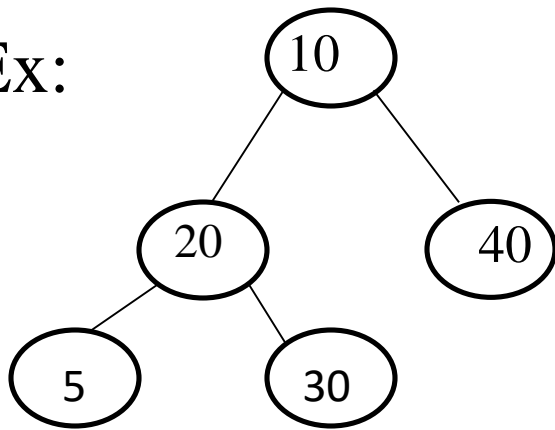
```
set its flag to -1 to indicate
```

```
right subtree is traversed*/
```

```
}
```

```
}
```

Ex:



Initially $cur = 10$;

1. After 1st iteration of 1st while loop

Node 10 is pushed to stack & its flag set to 1.
 $Cur = cur \rightarrow llink$; i.e $Cur = 20$

10	1

2. After 2nd iteration of 1st while loop

Node 20 is pushed to stack & its flag set to 1
 $Cur = cur \rightarrow llink$; i.e $Cur = 5$

20	1
10	1

3. After 3rd iteration of 1st while loop

Node 5 is pushed to stack & its flag set to 1.

Cur=cur→llink; i.e Cur=NULL

5	1
20	1
10	1

4. While loop terminates since cur==NULL

Since s[top].flag!= -1, 2nd while not entered.

cur=s[top].node; i.e cur=node 5;

Cur=cur→rlink; i.e cur=NULL;

s[top].flag = -1

5	-1
20	1
10	1

5. cur==NULL, 1st while loop not entered

Since s[top].flag < 0, 2nd while is entered.

cur=s[top].node; i.e cur=node 5;

Print 5

Stack is not empty, continue;

20	1
10	1

6. s[top].flag!= -1, 2nd While loop is exited

Cur=s[top].node; i.e cur=20;

Cur=cur→rlink; i.e cur=30;

s[top].flag = -1

20	-1
10	1

7. 1st while is entered

Node 30 is pushed to stack & its flag set to 1.

Cur=cur→llink; i.e Cur=NULL

30	1
20	-1
10	1

8. cur==NULL, 1st while loop exits

Since s[top]!= -1, 2nd while not entered.

cur=s[top].node; i.e cur=30;

cur=cur→rlink; i.e cur=NULL;

s[top].flag = -1

30	-1
20	-1
10	1

9. cur==NULL, 1st while loop not entered

Since s[top].flag<0, 2nd while is entered.

cur=s[top].node; i.e cur=node 30;

Print 30

Stack is not empty, continue;

10. s[top].flag<0, 2nd While loop continues

cur=s[top].node; i.e cur=node 20;

Print 20

Stack is not empty, continue;

11. s[top].flag!=-1, 2nd While loop is exited

Cur=s[top].node; i.e cur=10;

Cur=cur→rlink; i.e cur=40;

s[top].flag =-1

20	-1
10	1

10	1

10	-1

12. 1st while is entered

Node 40 is pushed to stack & its flag set to 1.

Cur=cur→llink; i.e Cur=NULL

40	1
10	-1

13. cur==NULL, 1st while loop exits

Since s[top]!=-1, 2nd while not entered.

cur=s[top].node; i.e cur=40;

cur=cur→rlink; i.e cur=NULL;

s[top].flag =-1

40	-1
10	-1

14. cur==NULL, 1st while loop not entered

Since s.[top].flag<0, 2nd while is entered.

cur=s[top].node; i.e cur=node 40;

Print 40

Stack is not empty, continue;

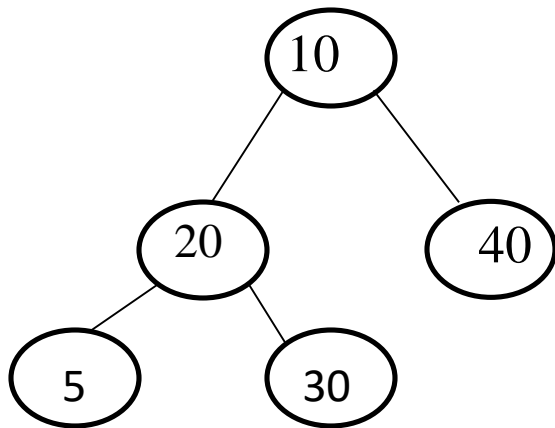
10	-1

15. s[top].flag < 0, 2nd While loop continues
cur=s[top].node; i.e cur=node 10;
Print 10
Stack is empty, stop;

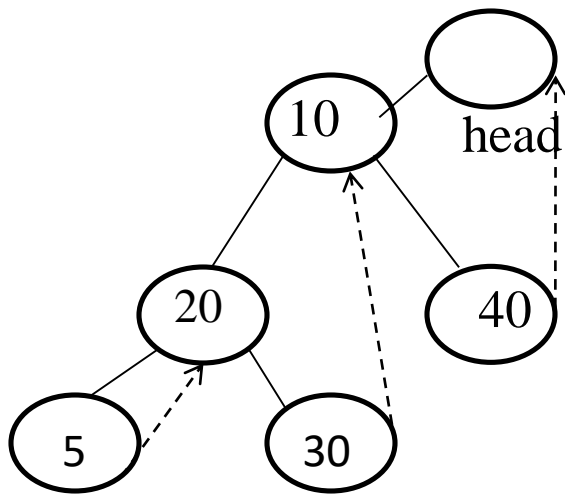
Hence elements printed in postorder are: 5, 30, 20, 40, 10

Threaded binary trees:

- In iterative traversals, node was pushed to stack because address of that node is needed after left or right subtree is traversed.
- To avoid using stacks, we can have the right link of any node which points to NULL, to point it to its inorder successor.
- Inorder successor of any node is the node which will be visited immediately after the present node in inorder traversal.



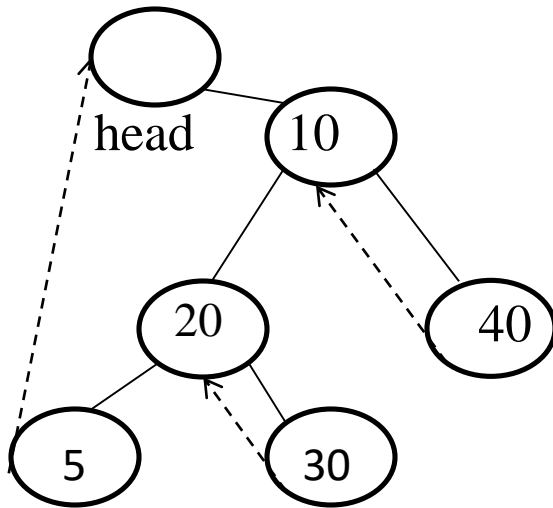
Here 20 is inorder successor(I.S) of 5 and 10 is I.S of 30.



- Right link of 5 is made to point to 20, which is its inorder successor and 30 is made to point to its I.S 10. 40 has no I.S, hence it points to header node.
- This is called right-in binary threaded tree and the link is called the thread(dotted lines in the diagram)

Left-in threaded binary tree:

- Left link of a node which points to NULL is made to point to the inorder predecessor.
- Inorder predecessor of any node is the node which was visited immediately prior to this node.



If the binary tree has both left and right threads, then it is called left-in, right-in binary threaded tree.

Node representation of threaded binary trees

Struct node

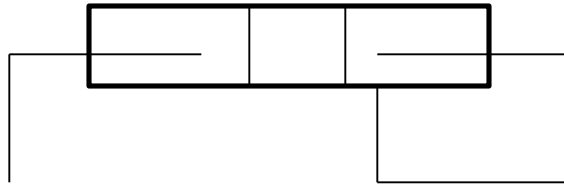
```
{  
    int info;  
    struct node *llink;  
    struct node *rlink;  
    int lthread;  
    int rthread;  
};
```

Typedef struct node *NODEPTR ;

If lthread and rthread are 1, it indicates the presence of threads.

Implementing right –in threaded tree

- To simplify the implementation, header node is used.
- Header node serves the predecessor of first node and successor of last node.
- When the tree is empty, the header node is represented as



i.e initially in main(),

```
head=getnode();
```

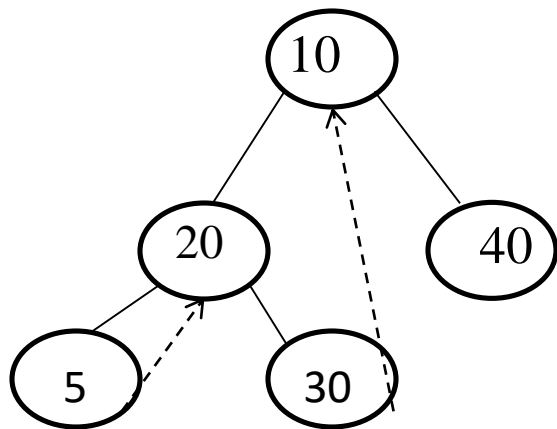
```
head→llink=NULL;
```

```
head→rlink=head;
```

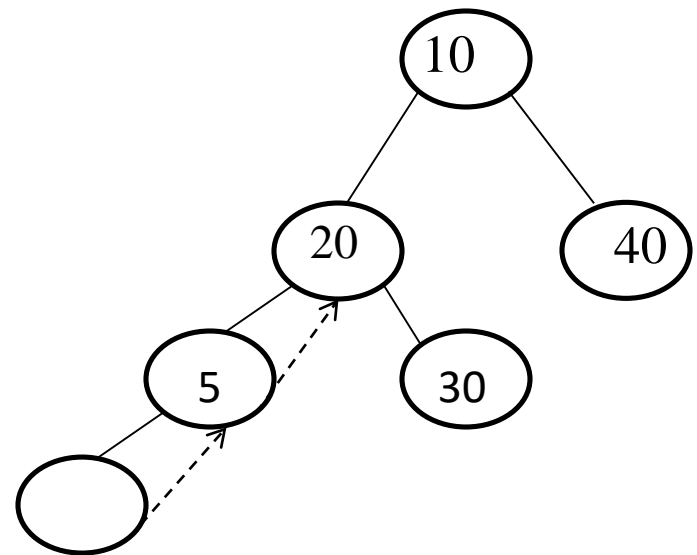
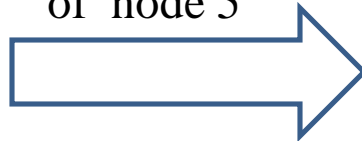
```
head→rthread=1;
```

Inserting an item into right-in threaded binary tree

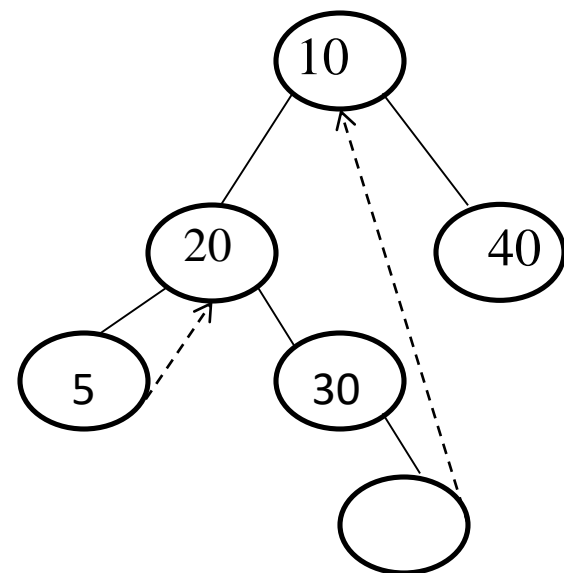
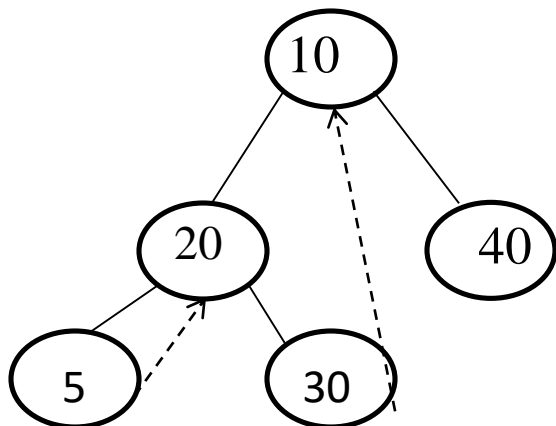
- Start from root and everytime a node is reached, user is asked whether to insert the new node to the right or left of present node.
- In insert_left function, after a new node is inserted, its right link is made to point to the present node.(thread to inorder successor)
- In insert_right function, first we store the address of right link of present node.(inorder successor present node). New node is attached to the right of present node and right link of present node is made to point to the already stored node in the first step.(inorder successor of present node becomes the inorder successor of new node after inserting to right)



After
inserting left
of node 5



After
inserting
right of node
30



```

/* c function to insert into right-in threaded binary tree*/
NODEPTR insert(int item, NODEPTR head)
{
    char direction;
    NODEPTR cur;
    if(head→llink==NULL)           /* if tree is empty, insert
    {                               new node to left of head*/

        insert_left(item, head);
        return head;
    }
    cur=head→llink;                /* else, start from root*/
    for(;;)
    {
        printf(“ insert to left or right of %d”, cur→info);
        direction=getche();
    }
}

```

```
if(direction=='L')
{
    if(cur→llink==NULL)           /* if left of present
    {                               node is NULL, insert
        insert_left(item,cur);     new node to left*/
        return head;
    }
    else                           /* else move left*/
        cur=cur→llink;
}
```

else	/* right direction*/
{	
if (cur→rthread==1)	/* right link of present
{	node is NULL, but now
insert_right(item,cur);	it has a right thread. Hence
	insert to right of present*/
return head;	
}	
else	
cur=cur→rlink;	/* no right thread. Hence
	move right*/
}// end else	
}// end for	
}// end function	

```
/* insert left function*/
```

```
Void insert_left(int item, NODEPTR present)
```

```
{
```

```
    NODEPTR temp;
```

```
    temp=getnode();
```

```
    temp→info=item;
```

```
    present→llink=temp;    /*attach new node to left*/
```

```
    temp→llink=NULL;    /*left of new is set to NULL*/
```

```
    temp→rlink=present;    /*right of new is made to point  
                             present(inorder successor)*/
```

```
    temp→rthread=1;    /*indicate rthread is present*/
```

```
}
```

```
/* insert right function*/
```

```
Void insert_right(int item, NODEPTR present)
```

```
{
```

```
    NODEPTR temp, successor;
```

```
    temp=getnode();
```

```
    temp→info=item;
```

```
    successor=present→rlink; /* store the address of inorder  
                                successor of present node*/
```

```
    present→rlink=temp;      /*attach new node to right*/
```

```
    present→rthread=0;        /* thread is absent*/
```

```
    temp→llink=NULL;         /*left of new is set to NULL*/
```

```
    temp→rlink=successor; /*right of new is made to point  
                                inorder successor*/
```

```
    temp→rthread=1;           /*indicate rthread is present*/
```

```
}
```

```

/*function to find the inorder successor of a given node*/
NODEPTR inorder_succcessor(NODEPTR present)
{
    NODEPTR temp;
    temp=present→rlink; /*get the address of right link of
                           present node*/
    if(present→rthread==1) /* if present node has
                             rthread, return its rlink*/
        return temp;
    while(temp→llink!=NULL) /* else go to leftmost node
                              of present node's right
                              node and return it*/
        temp=temp→llink;
    return temp;
}

```

Conclusion

- Trees are useful in modeling the parent child relationship(hierarchical).
- Used by operating system to save and retrieve files.
- BST is a very important concept which makes searching easier and efficient.
- Expression tree or parse tree is used by compilers to evaluate expressions.
- Trees are used to efficiently store data and for faster access.