Trees

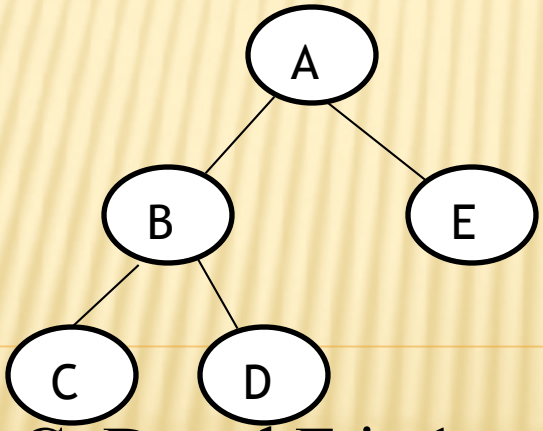
- Trees are natural structures for representing certain kinds of hierarchical data.(How our files get saved under hierarchical directories)
- Tree is a data structure which allows us to associate a parent-child relationship between various pieces of data and thus allows us to arrange our records, data 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

- 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.

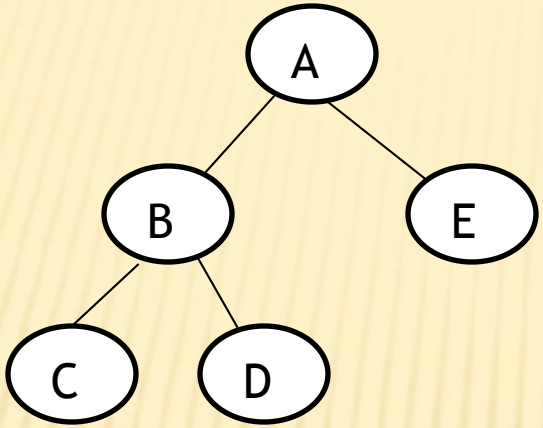
Directed tree:

- 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:

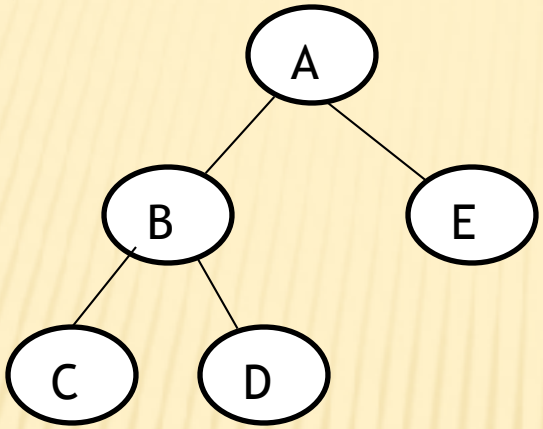
- The nodes which are reachable from a particular node using only a single edge are called children of that node and this node is called the father of those nodes.

  Children of A are B and E.

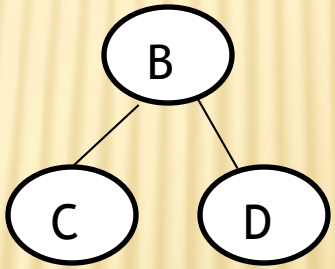
  Children of B are C and D.

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.



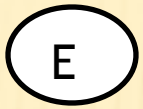
left subtree of A is



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:

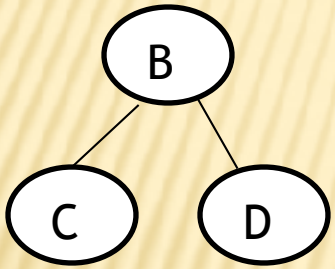
- Height of a tree is one more than the maximum level in the tree.

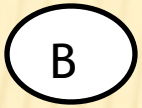
  In the previous tree , maximum level is 2 and height is 3

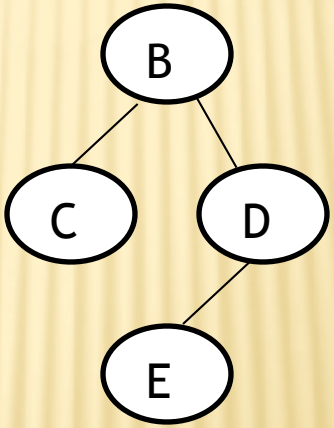
Binary trees:

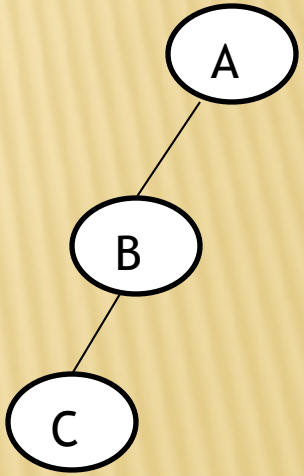
- A binary tree is a directed tree in which outdegree of each node is less than or equal to 2. i.e each node can have 0, 1 or 2 children.

Examples of binary tree





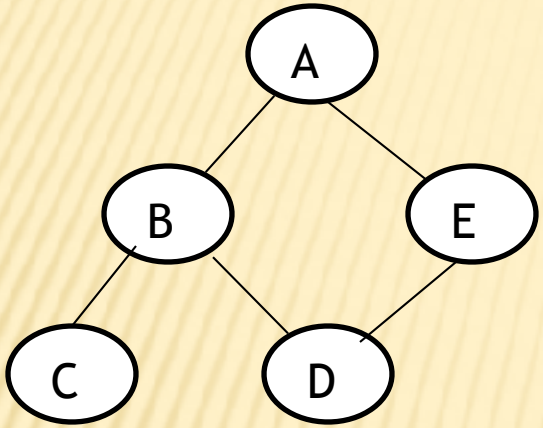




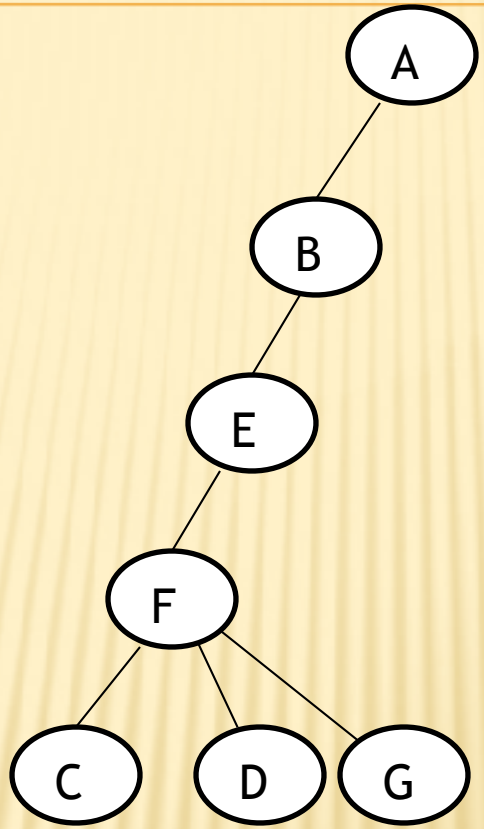
BINARY TREE

- Definition: A binary tree is a finite set of elements that is either empty or partitioned into 3 disjoint subsets. The first subset contains root of the tree and other two subsets themselves are binary trees called left and right subtree. Each element of a binary tree is called a NODE of the tree.

Examples of non binary trees:



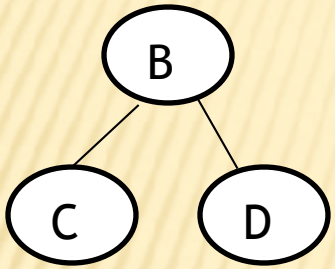
D has indegree 2, hence not directed tree

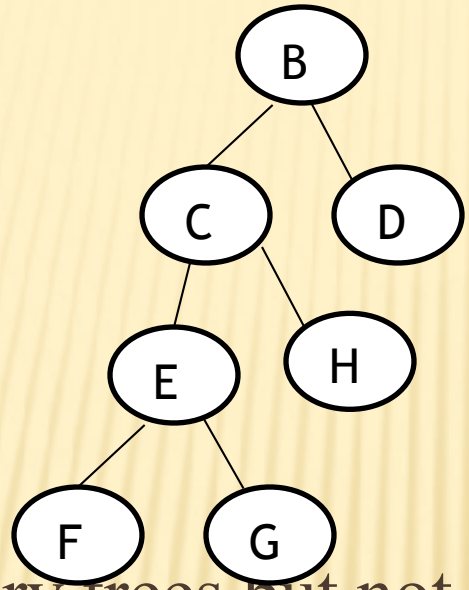


F has outdegree 3.

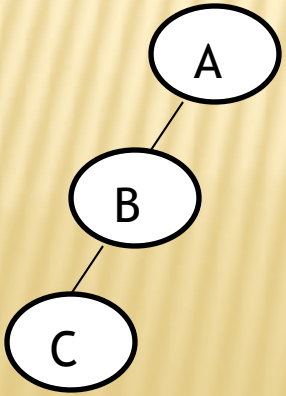
Strictly binary tree:

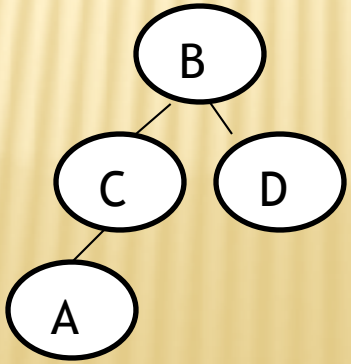
- If the out degree of every node in a tree is either 0 or 2(1 not allowed), then the tree is strictly binary tree.





Tress which are binary trees but not strictly binary:

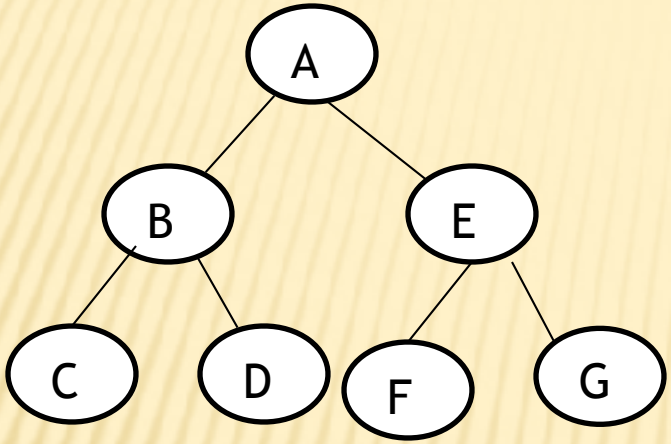




- If every non leaf node in a BT has non empty left and right subtrees ,the tree is called strictly binary tree.
- Properties
- If a SBT has n leaves then it contains 2n-1 nodes.
- Depth: it is the maximum level of any leaf in the tree.

Complete binary tree:

- Is a strictly binary tree in which the number of nodes at any level ‘i’ is pow(2,i).



Number of nodes at level 0(root level) is pow(2,0)→1

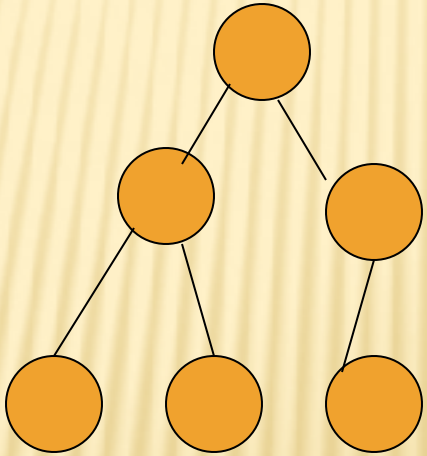
Number of nodes at level 1(B and E) is pow(2,1)→2

Number of nodes at level 2(C,D,F and G) is pow(2,2)→4

- A complete binary tree of depth d is the strictly binary tree all of whose leaves are at level d.
- The total number of nodes at each level between 0 and d equals the sum of nodes at each level which is equal to 2^{d+1} -1
- No of non leaf nodes in that tree=2^{d}-1
- No of leaf nodes =2^{d}

ALMOST COMPLETE BT

- All levels are complete except the lowest
- In the last level empty spaces are towards the right.



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.

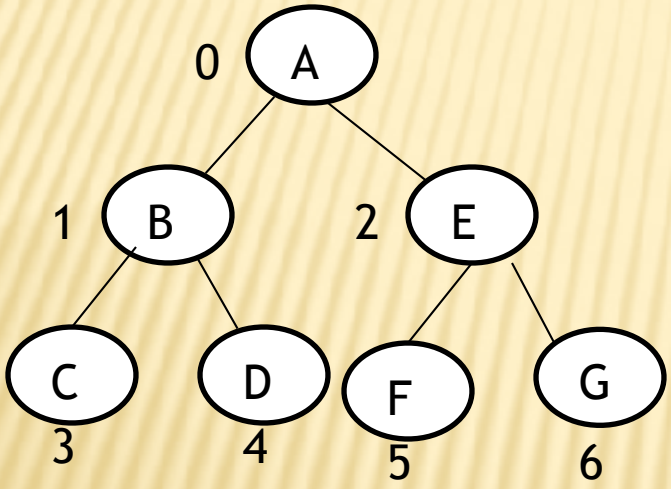
```
struct node
{
 int info;
 struct node *llink;
 struct node *rlink;
};
 typedef struct node *NODEPTR;
```

Implementing a binary tree:

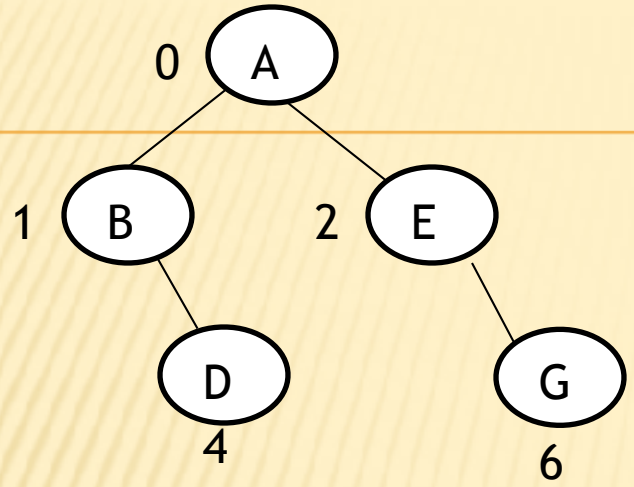
- A pointer variable root is used to point to the root node.
- Initially root is NULL, which means the tree is empty.

  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.

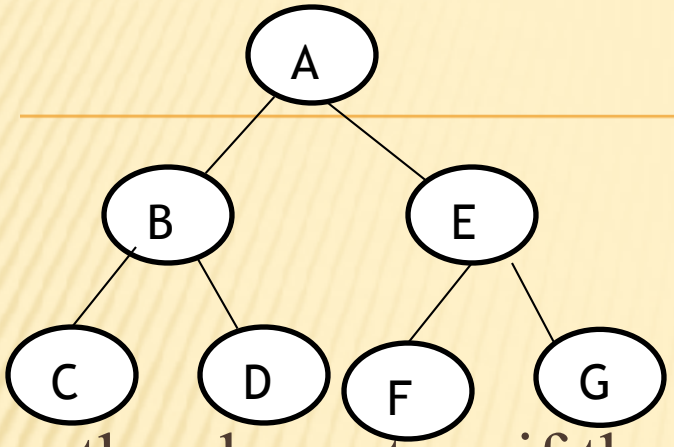
Copy: to obtain exact copy of given tree.

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.

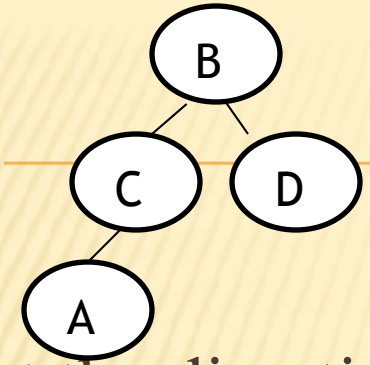
```
/*function to insert item into tree*/
NODEPTR insert(int item, NODEPTR root)
{
    NODEPTR temp, cur, prev;
    char direction[10];
    int i;
    temp=getnode();
    temp→info=item;
    temp→llink=temp→rlink=NULL;
    if(root==NULL)
        return temp;
    cout<<“enter the direction of insertion”;
    cin>>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))
{
   cout<<“insertion not possible”;
   delete 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= strlen(direction) or when cur==NULL.
- For insertion to be possible, control should come out when cur==NULL and i = strlen(direction) both at the same time.

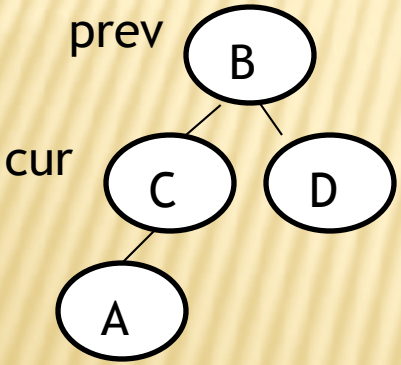
  i.e when we get the position to insert(cur==NULL), the string should be completed.



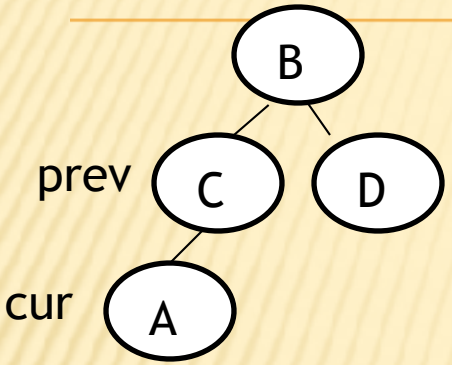
Let the direction string be “LLR”. String length of string is 3.

Initially cur ==root and prev==NULL

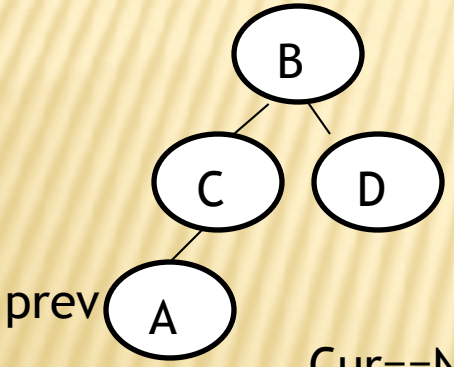
- Loop 1: i=0 ,direction[0]=‘L’



- Loop 2: i=1, direction[1]='L'



- Loop 3: i=2, directiom[2]='R'

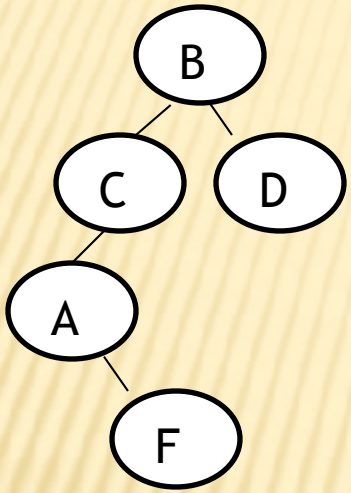


Cur==NULL

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

- Now direction[i-1] is direction[3-1] which gives ‘R’.
- Hence prev→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==strlen(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.

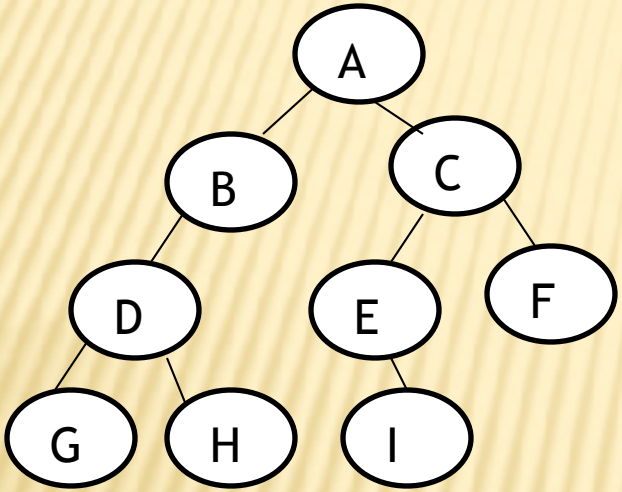
There are 3 main traversals techniques

- 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.

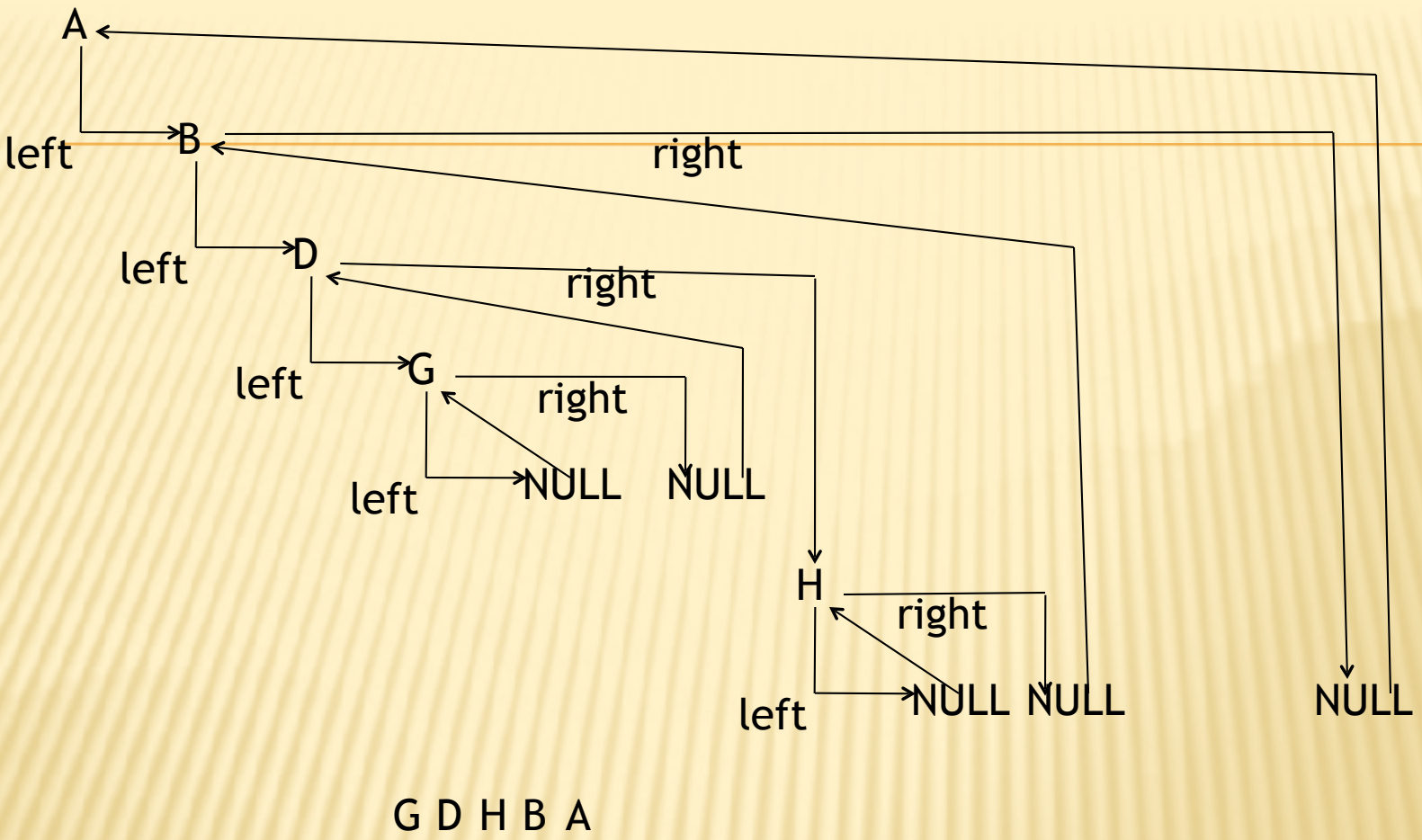
- Now move to the right of G, which is also NULL. Hence go back to root of G and print it. So D is printed next.
- Next go to the right of D, which is H. Now another root H is visited.
- Now move to the left of H, which is NULL. So go back to root H and print it and go to right of H, which is NULL.
- 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.

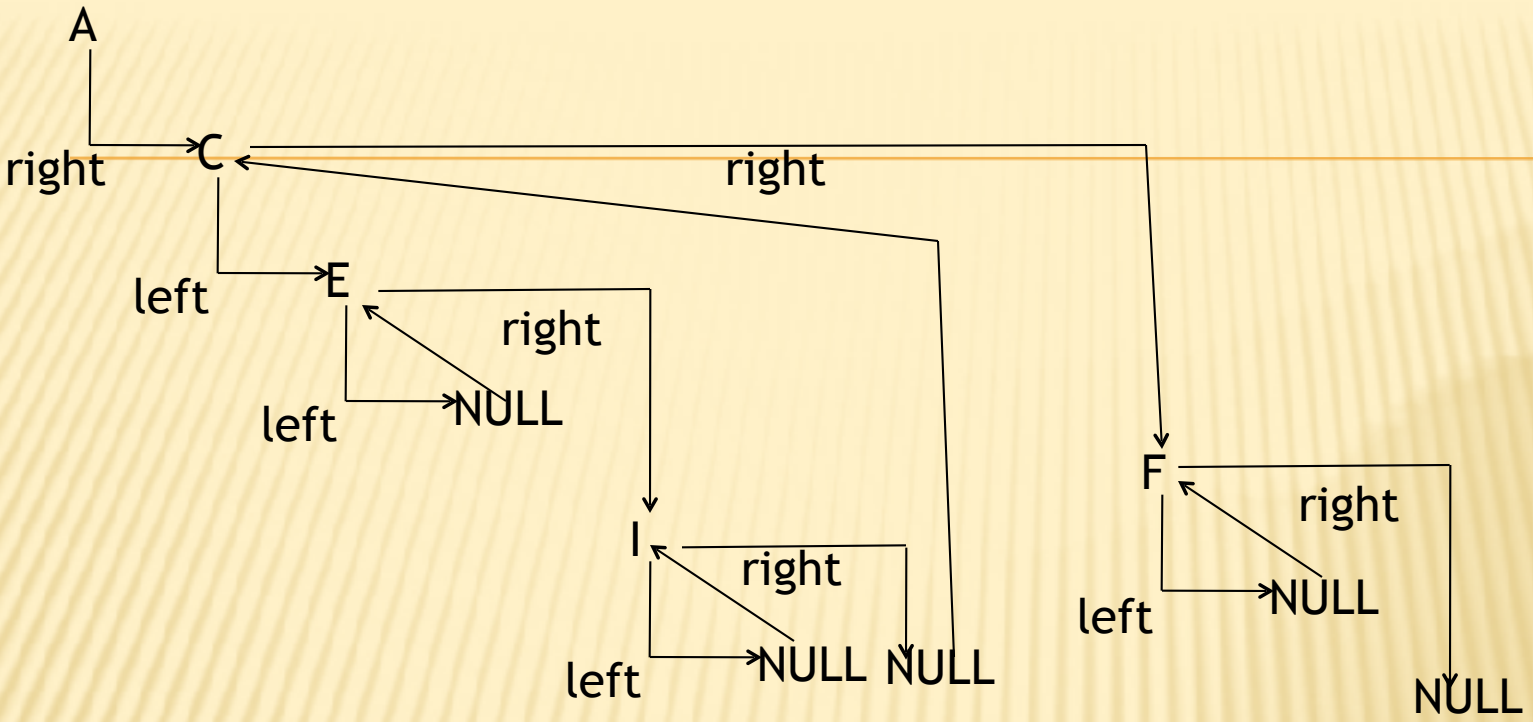
Hence inorder traversal of above tree gives GDHBAEICF

/*recursive algorithm for inorder traversal*/

```
Void inorder(NODEPTR root)
{
    if(root!=NULL)
    {
        inorder(root→llink);
        cout<<root→info;
        inorder(root→rlink);
    }
}
```



Traversing left subtree of A inorder



E I C F

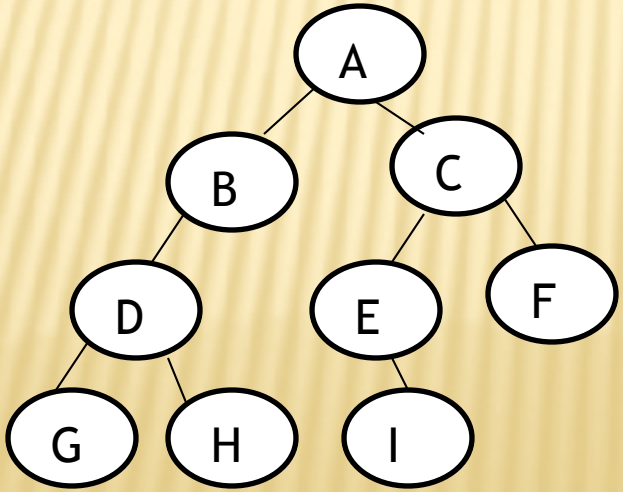
Traversing right subtree of A inorder

Preorder traversal:

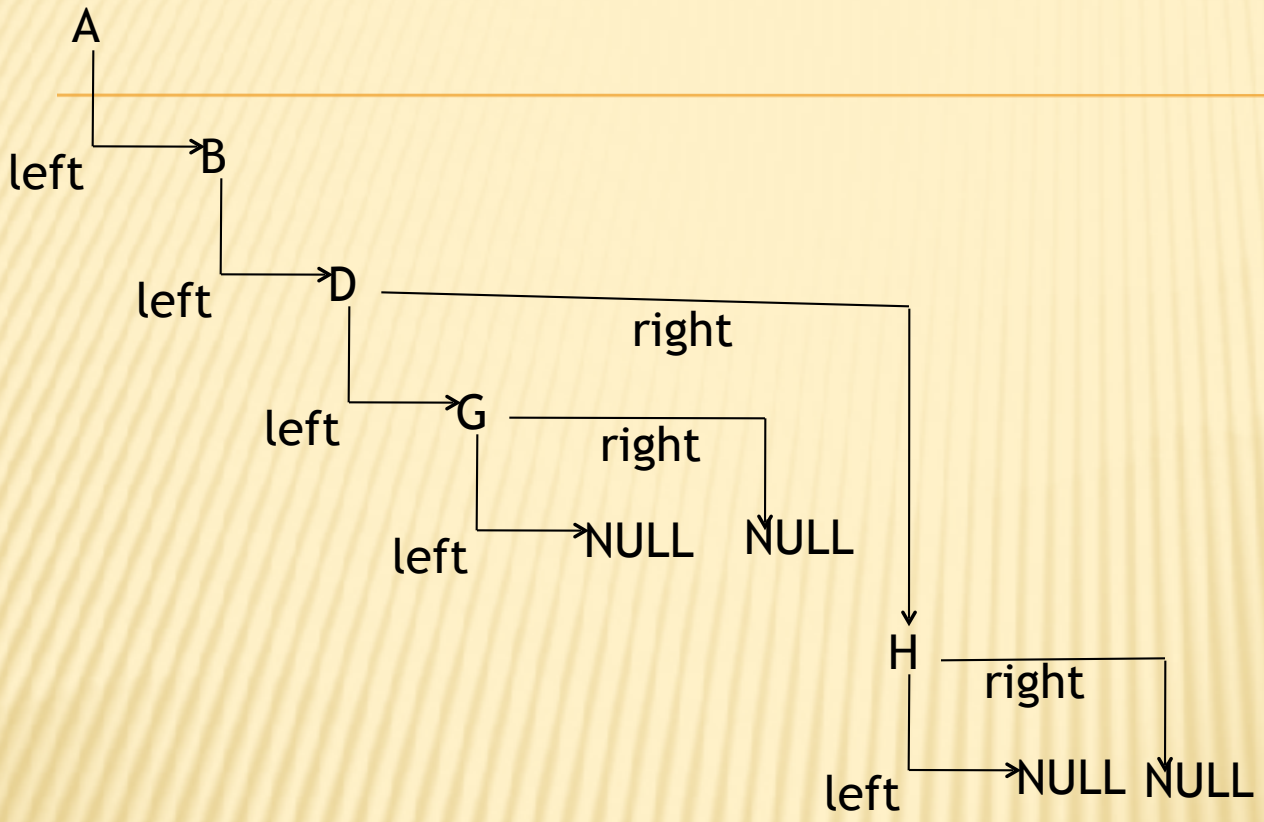
Preorder traversal is defined as

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.

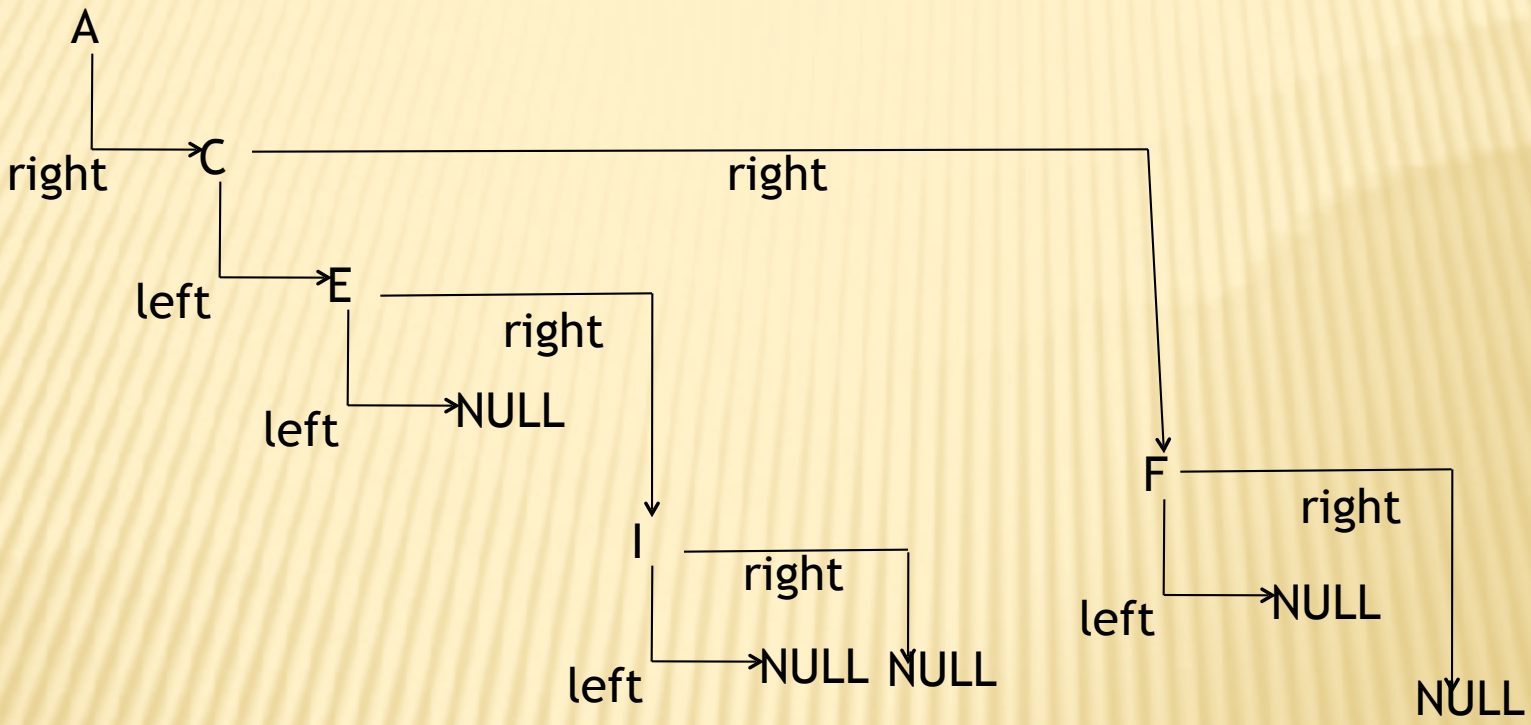


Traversing left subtree in preorder



A B D G H

Traversing right sub tree in preorder



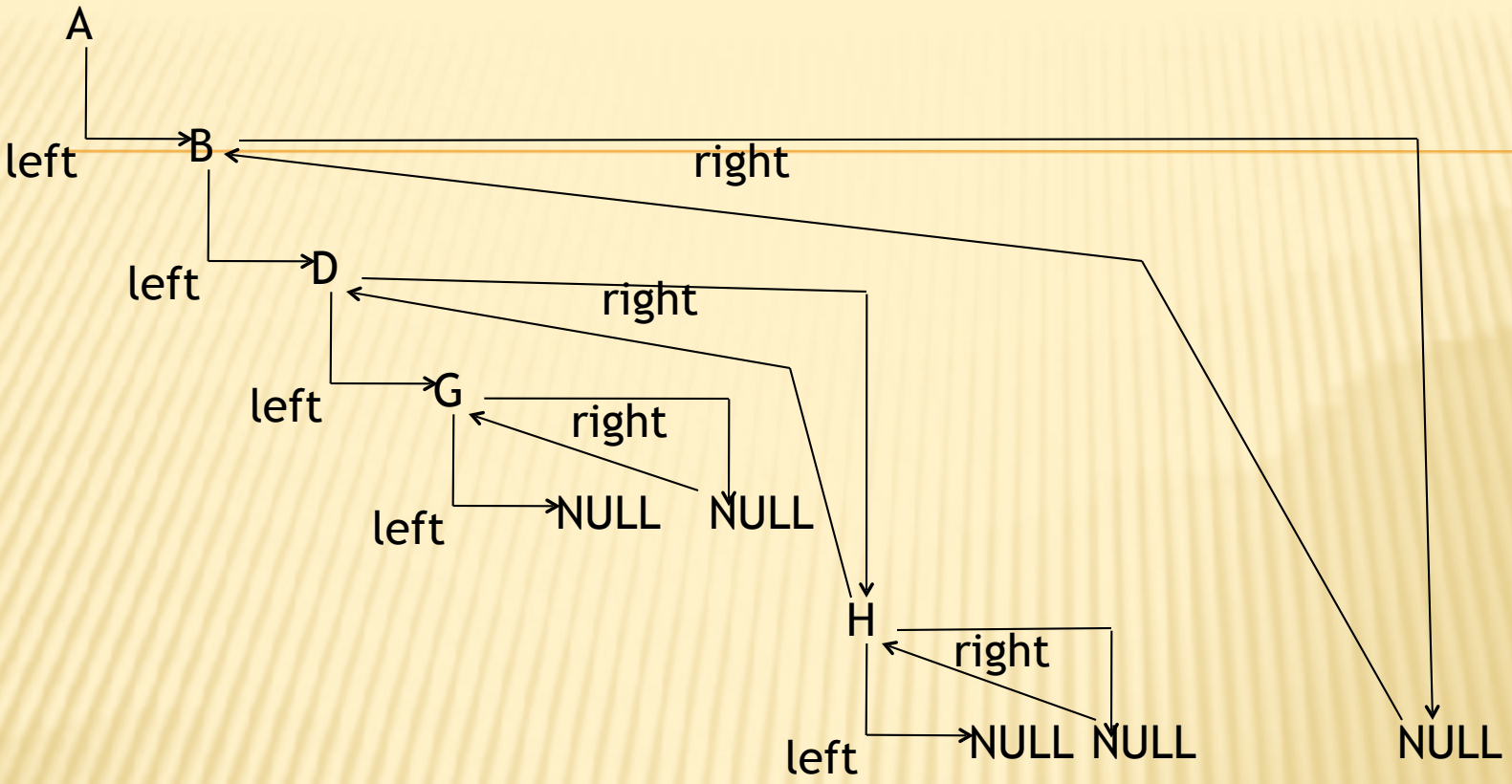
C E I F

Post order traversal

Post order traversal is defined as

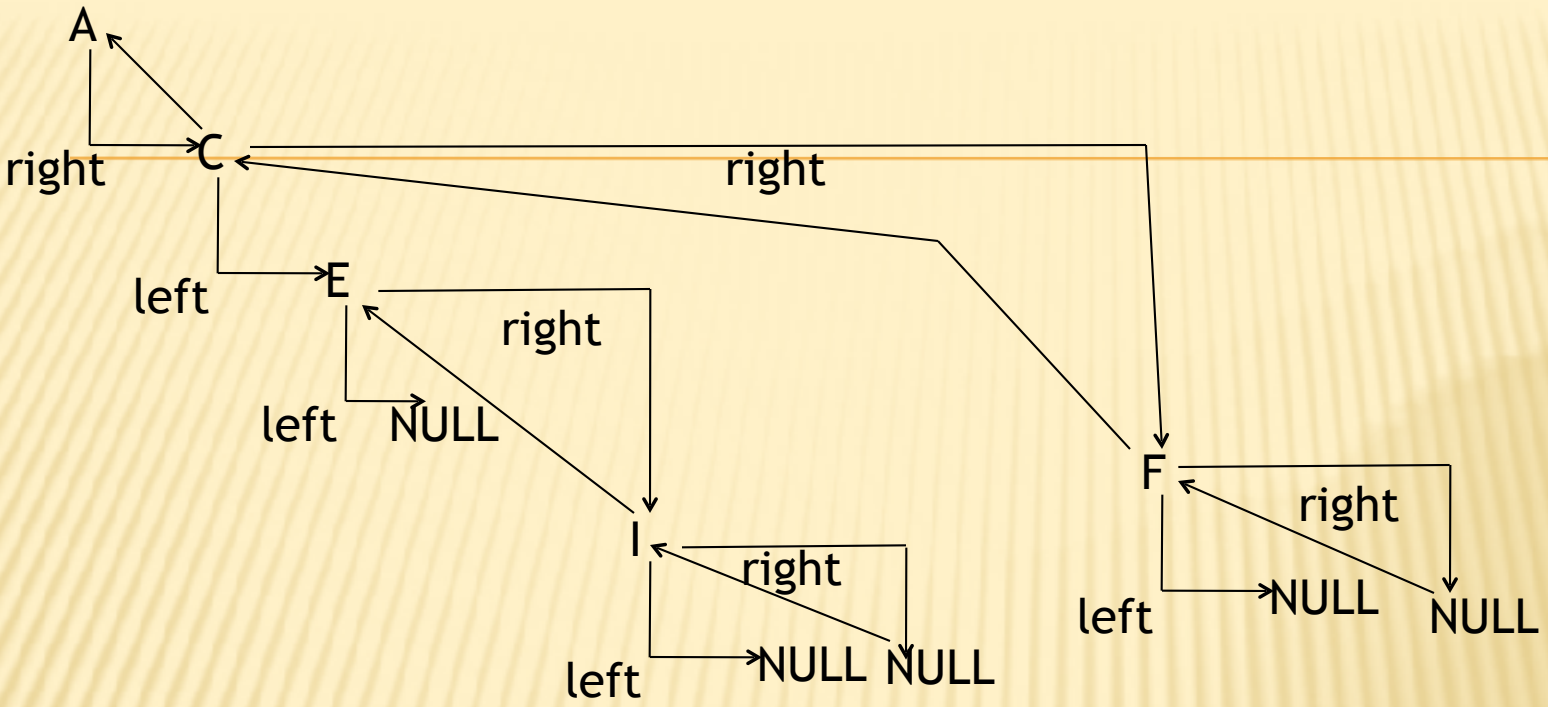
1. Traverse the left subtree in postorder.
2. Traverse the right subtree in postorder.
3. Process the root node.

- In post order traversal, we first traverse towards left, then move to right and then visit the root. This process is repeated recursively.



G H D B

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

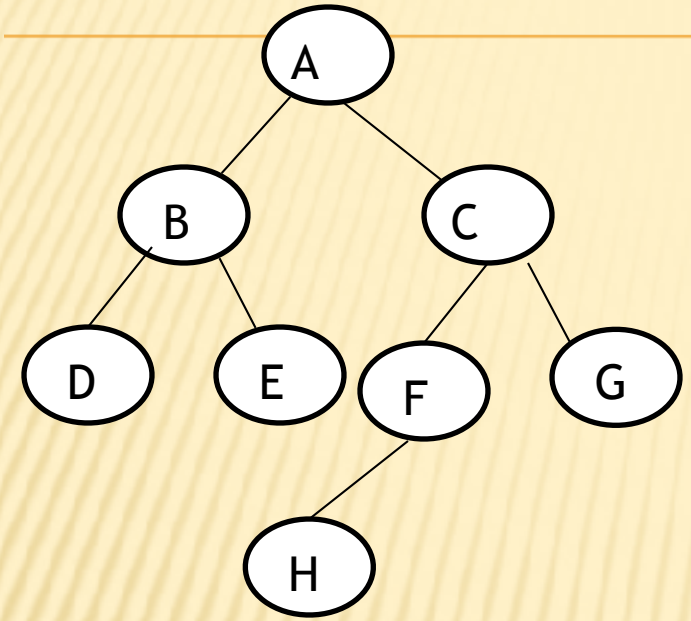
Copying a tree:

- Getting the exact copy of the given tree.

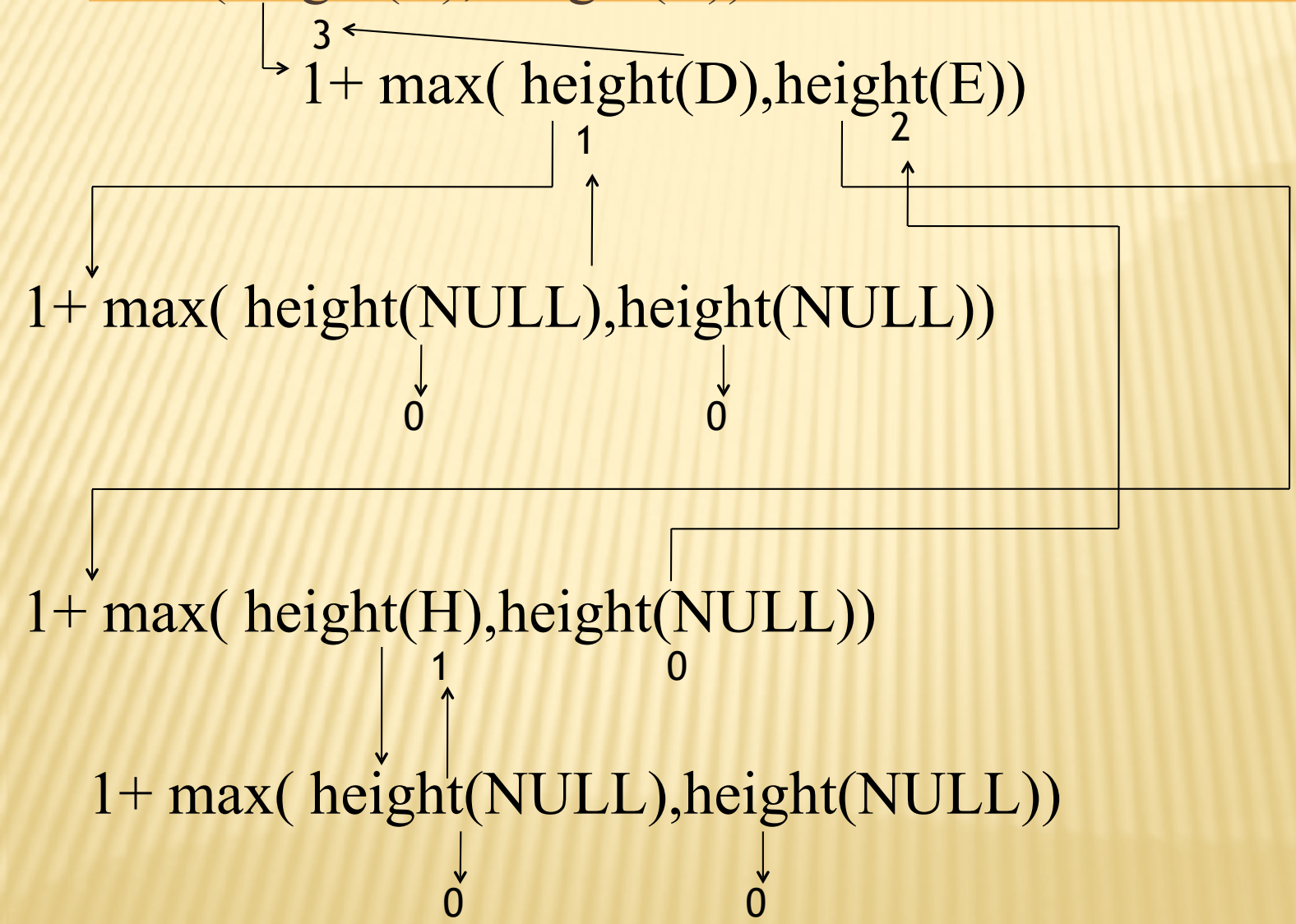
```
/*recursive function to copy a tree*/
NODEPTR copy (NODEPTR root)
{
    NODEPTR temp;
    if(root == NULL)
        return NULL;
    temp=getnode();
    temp→info=root→info;
    temp→llink=copy(root→llink);
    temp→rlink=copy(root→rlink);
    return temp;
}
```

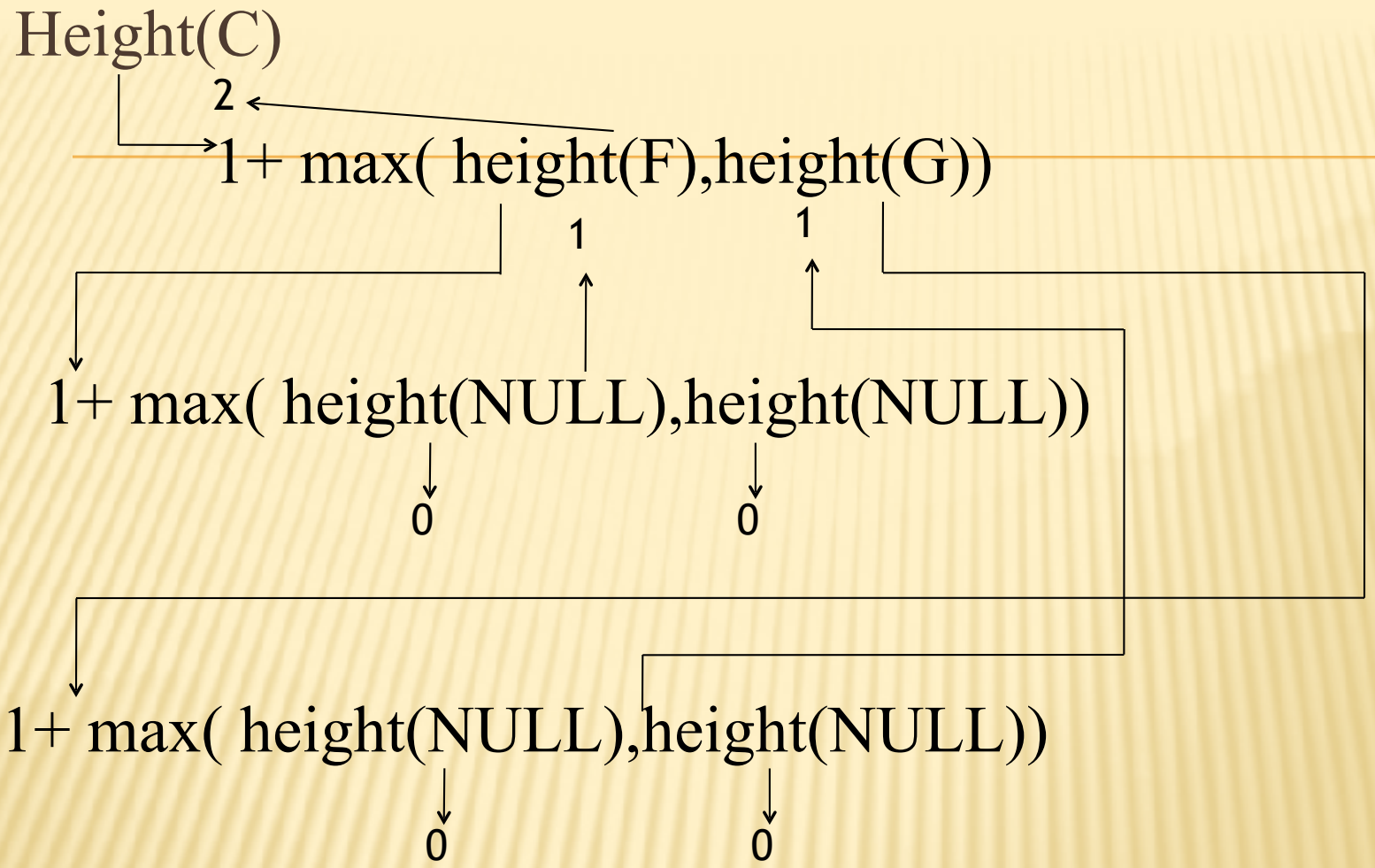
```
/*recursive function to find the height of a tree*/
Int height (NODEPTR root)
{
    if(root==NULL)
        return 0;
    return( 1+ max(height (root→llink), height(root→rlink)));
}
/*max function*/
Int max(int a, int b)
{
    if(a>b)
        return a;
    else
        return b;
}
```

Finding the height of the tree using recursion



1+max(height(B), height(C))





Finally,

1+max(height(B),height(C)→1+max(3,2)→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:

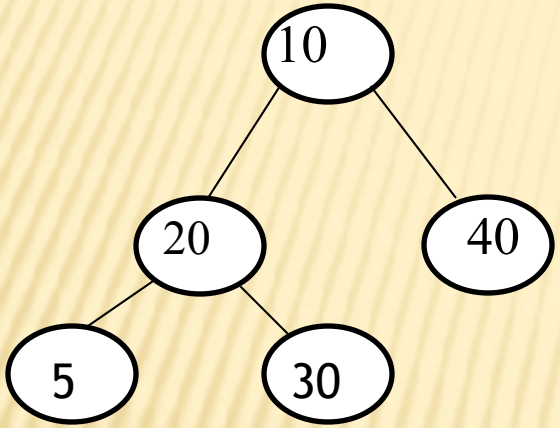
- Traverse the tree in any of the 3 techniques and every time a node is visited, check whether the right and left link of that node is NULL. If yes, count is incremented.

```
/*counting number of leaf nodes using inorder technique*/
Void count_leafnodes( NODEPTR root)
{
    if(root!=NULL)
    {
        count_leafnodes(root→llink);
        if(root→llink==NULL && root→rlink==NULL)
            count++;
        count_leafnodes(root→rlink);
    }
}
```

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

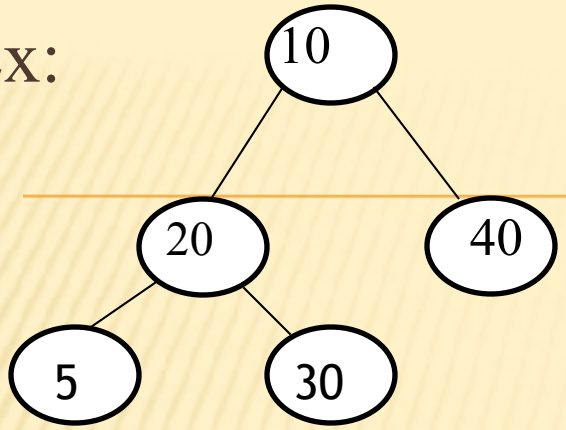
- Once traversing the left side is over, pop the most pushed node and go to its right.

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:

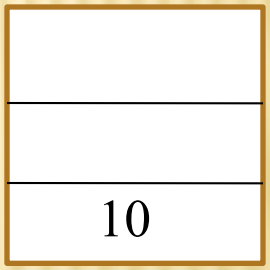


1. After 1st iteration of while loop

10 is printed

Node 10 is pushed to stack

Cur=cur→llink; i.e Cur=20

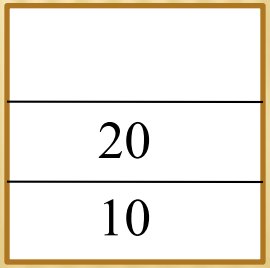


2. After 2nd iteration of while loop

20 is printed

Node 20 is pushed to stack

Cur=cur→llink; i.e Cur=5

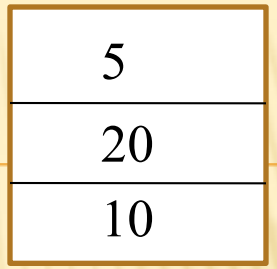


3. After 3rd iteration of while loop

5 is printed

Node 5 is pushed to stack

Cur=cur→llink; i.e Cur=NULL

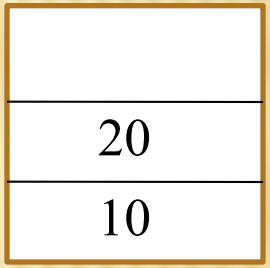


4. While loop terminates since cur==NULL

Stack is not empty

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

cur=cur→rlink; i.e cur=NULL

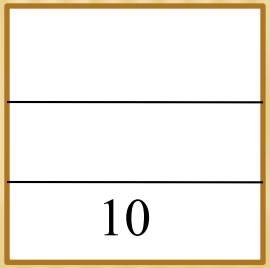


5. cur==NULL, while loop not entered

Stack not empty

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

cur=cur→rlink; i.e cur=30

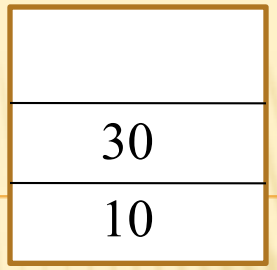


6. While loop is entered

30 is printed

Node 30 is pushed to stack

Cur=cur→llink; i.e cur=NULL

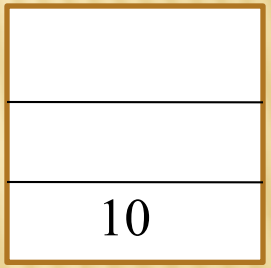


7. While loop terminates since cur==NULL

Stack not empty

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

cur=cur→rlink; i.e cur=NULL

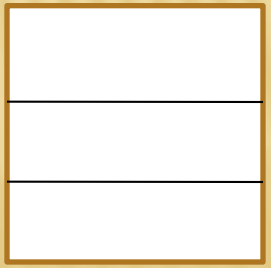


8. cur==NULL, while loop not entered

Stack not empty

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

cur=cur→rlink; i.e cur=40

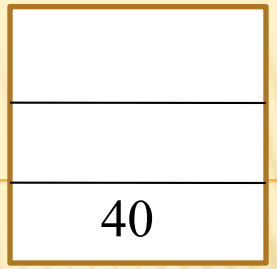


9. While loop is entered

40 is printed

Node 40 is pushed to stack

cur=cur→llink; i.e cur=NULL

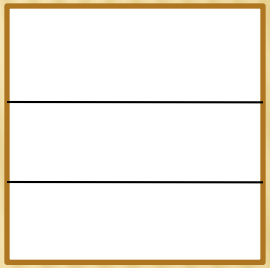


10. While loop terminates since cur==NULL

Stack not empty

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

cur=cur→rlink; i.e cur=NULL



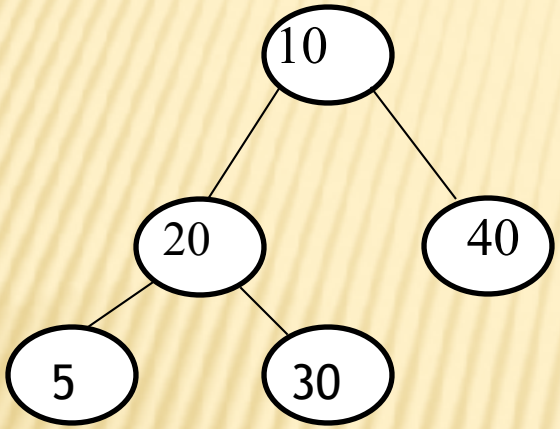
11. cur==NULL and stack empty

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.

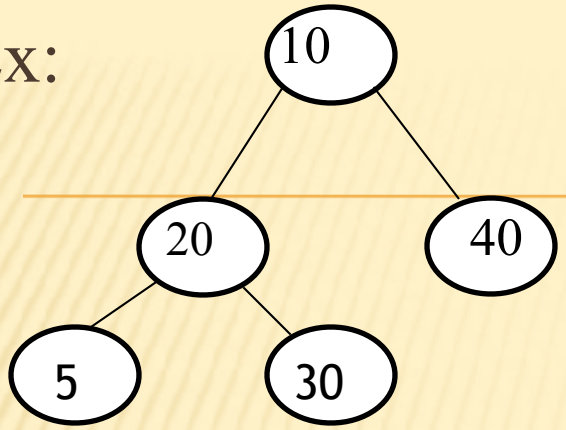


Here 10, 20 , 5 is pushed to stack. Then pop 5, print it and move right.
Now pop 20, print it and move right and push 30 and move left.
Pop 30, print it and move right.
Pop 10, print it and move right and push 40 and so on

```
/*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;
}}
```

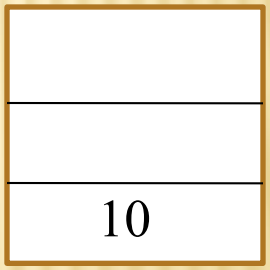
Ex:



1. After 1st iteration of while loop

Node 10 is pushed to stack

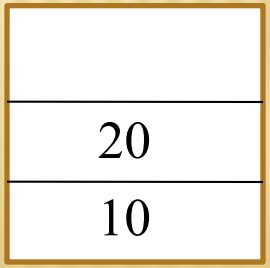
Cur=cur→llink; i.e Cur=20



2. After 2nd iteration of while loop

Node 20 is pushed to stack

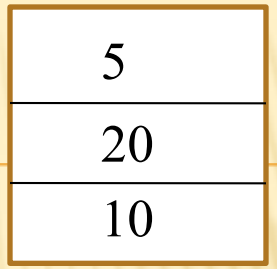
Cur=cur→llink; i.e Cur=5



3. After 3rd iteration of while loop

Node 5 is pushed to stack

Cur=cur→llink; i.e Cur=NULL



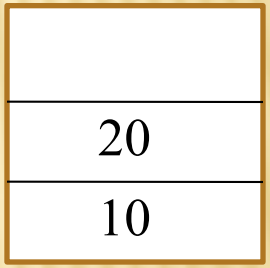
4. While loop terminates since cur==NULL

Stack is not empty

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

Print 5

cur=cur→rlink; i.e cur=NULL



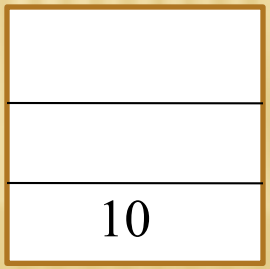
5. cur==NULL, while loop not entered

Stack not empty

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

Print 20

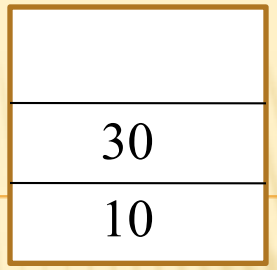
cur=cur→rlink; i.e cur=30



6. While loop is entered

Node 30 is pushed to stack

Cur=cur→llink; i.e cur=NULL



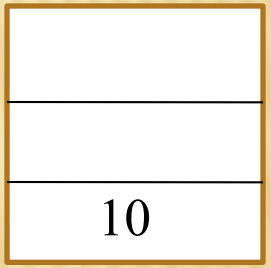
7. While loop terminates since cur==NULL

Stack not empty

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

Print 30

cur=cur→rlink; i.e cur=NULL



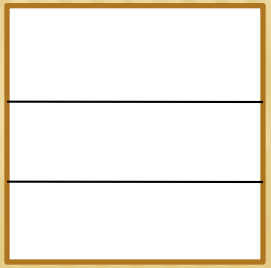
8. cur==NULL, while loop not entered

Stack not empty

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

Print 10

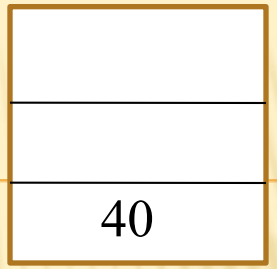
cur=cur→rlink; i.e cur=40



9. While loop is entered

Node 40 is pushed to stack

cur=cur→llink; i.e cur=NULL



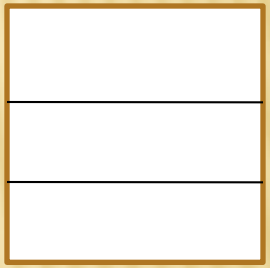
10. While loop terminates since cur==NULL

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:

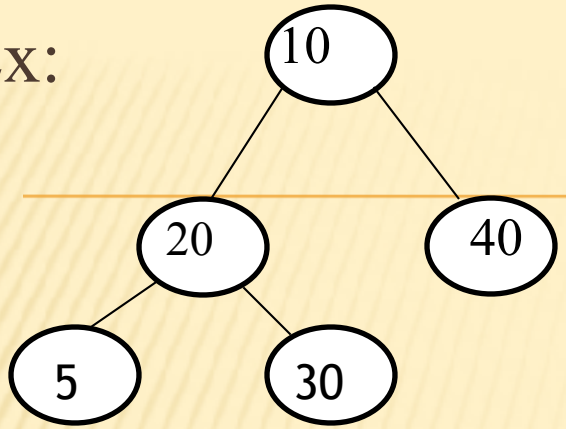
- Traverse left and push the nodes to stack with their flags set to 1, until NULL is reached.
- Then flag of current node is set to -1 and its right subtree is traversed. Flag is set to -1 to indicate that traversing right subtree of that node is over.
- Hence if flag is -1, it means traversing right 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)
    {                                   /*traverse left of tree and push
        s[++top].node=cur;                the nodes to the stack and
        s[top].flag=1;                    set flag to 1*/
        cur=cur→llink;
    }
    while(s[top].flag<0)
    {
        cur=s[top--].node;      /*if flag is –ve, right subtree
        printf("%d", cur→info);           is visited and hence node
                                          is poped and printed*/
        if(stack_empty(top))            /*if stack is empty, traversal
                return;                   is complete*/
    }
```

```
cur= s[top].node;          /*after left subtree is
cur=cur→rlink;                traversed, move to right and
s[top].flag=-1;               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→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→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;

20	-1
10	1

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;

10	1

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

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.

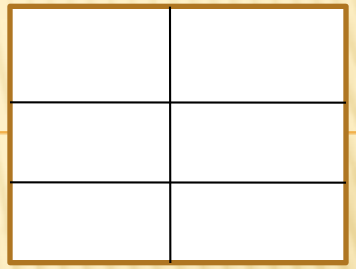
cur=s[top].node; i.e cur=node 40;

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;

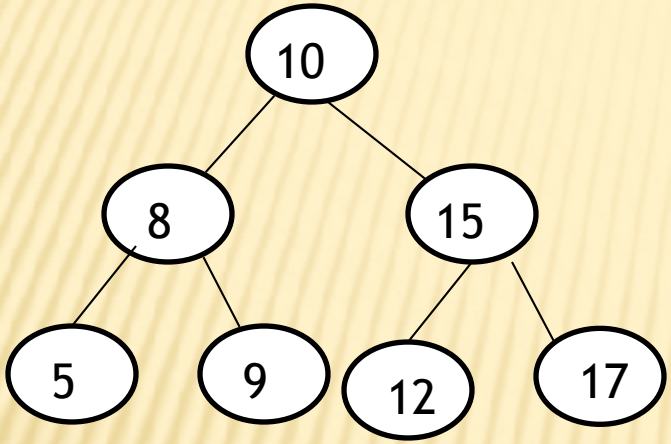
Print 10

Stack is empty, stop;

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

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.

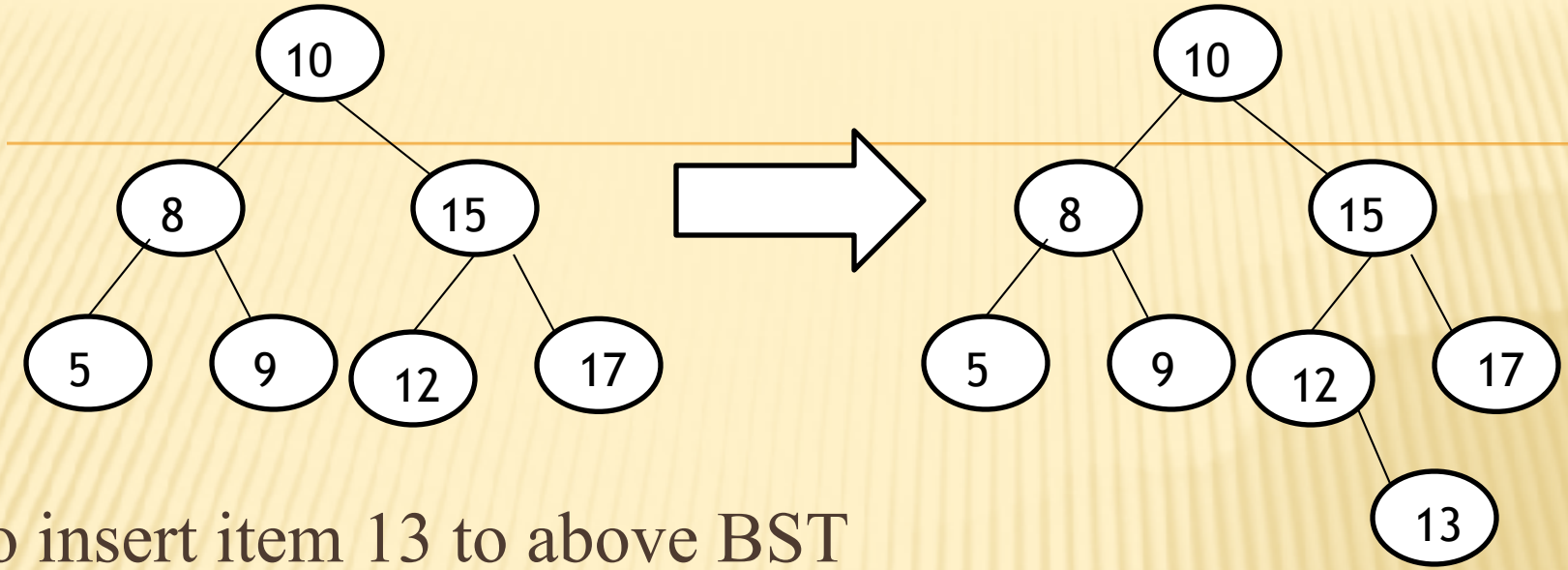
- Inorder traversal of BST results in ascending order of elements

Operations performed on BST

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

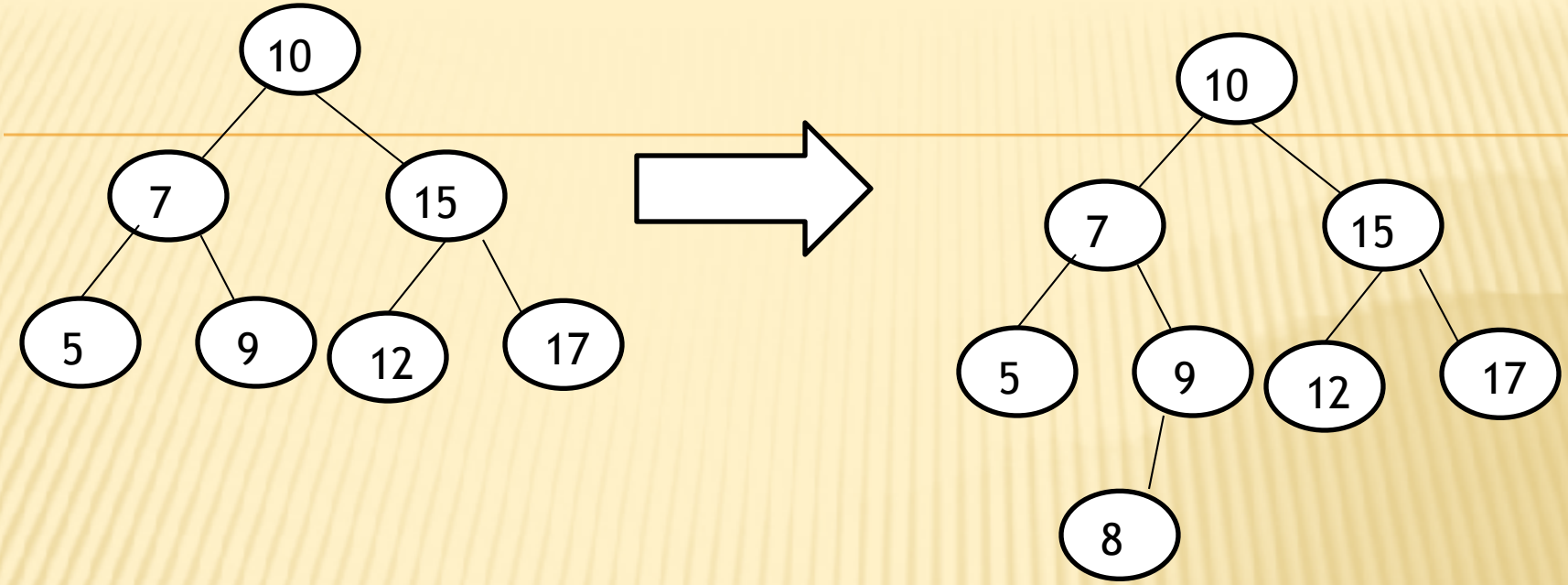
Insertion

- 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 id 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.

```
/* 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=NULL;
    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)          /*if item< info of current
        cur=cur→llink;              node, move left*/
else
        cur=cur→rlink;           /*else move right*/
}                                /*end of while*/
if(item< prev→info)              /*if item< parent, insert
   prev→llink=temp;                new node to its left*/

else                             /*else insert to its right*/
   prev→rlink=temp;

return root;
}
```

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 temp;
   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:

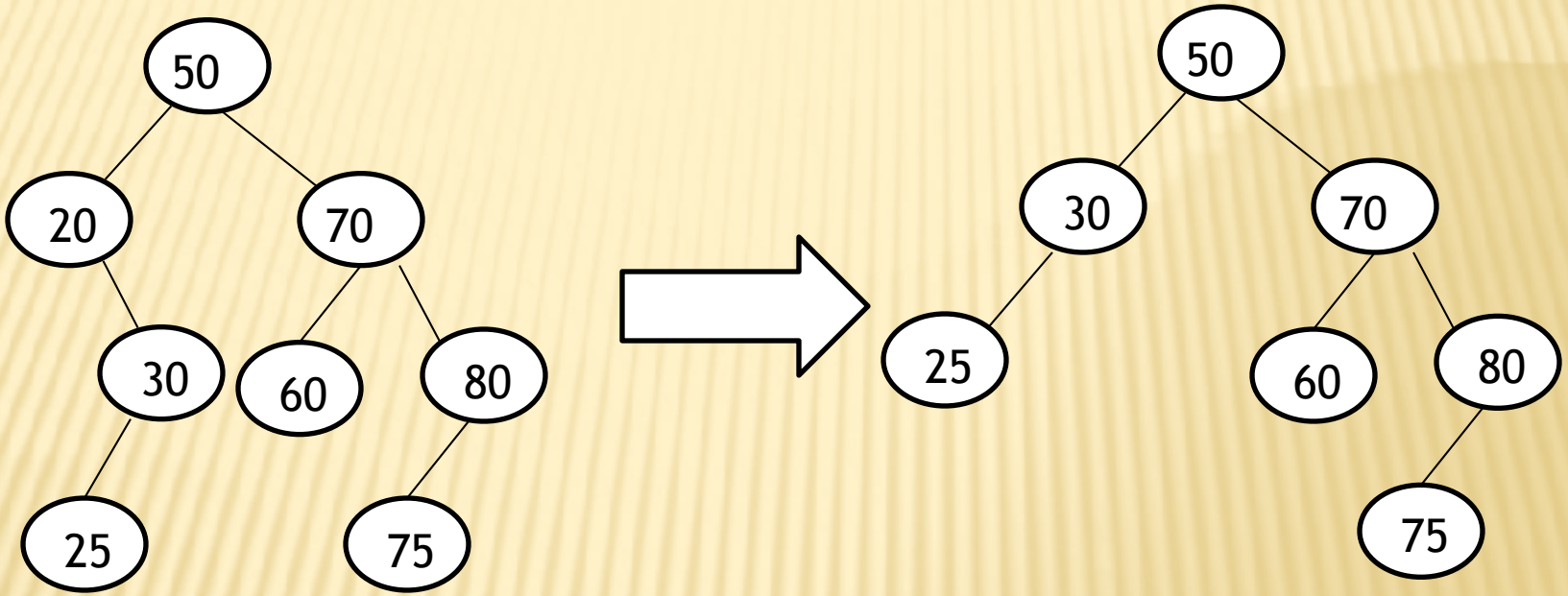
1. Finding the maximum element in BST: maximum element will always be the last right child in a BST. Move to the rightmost node in a BST and you will end up in the maximum element.
2. Finding the minimum element: Move to the leftmost child and you will reach the least element in BST.
3. Finding the height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

Deleting a node from a BST:

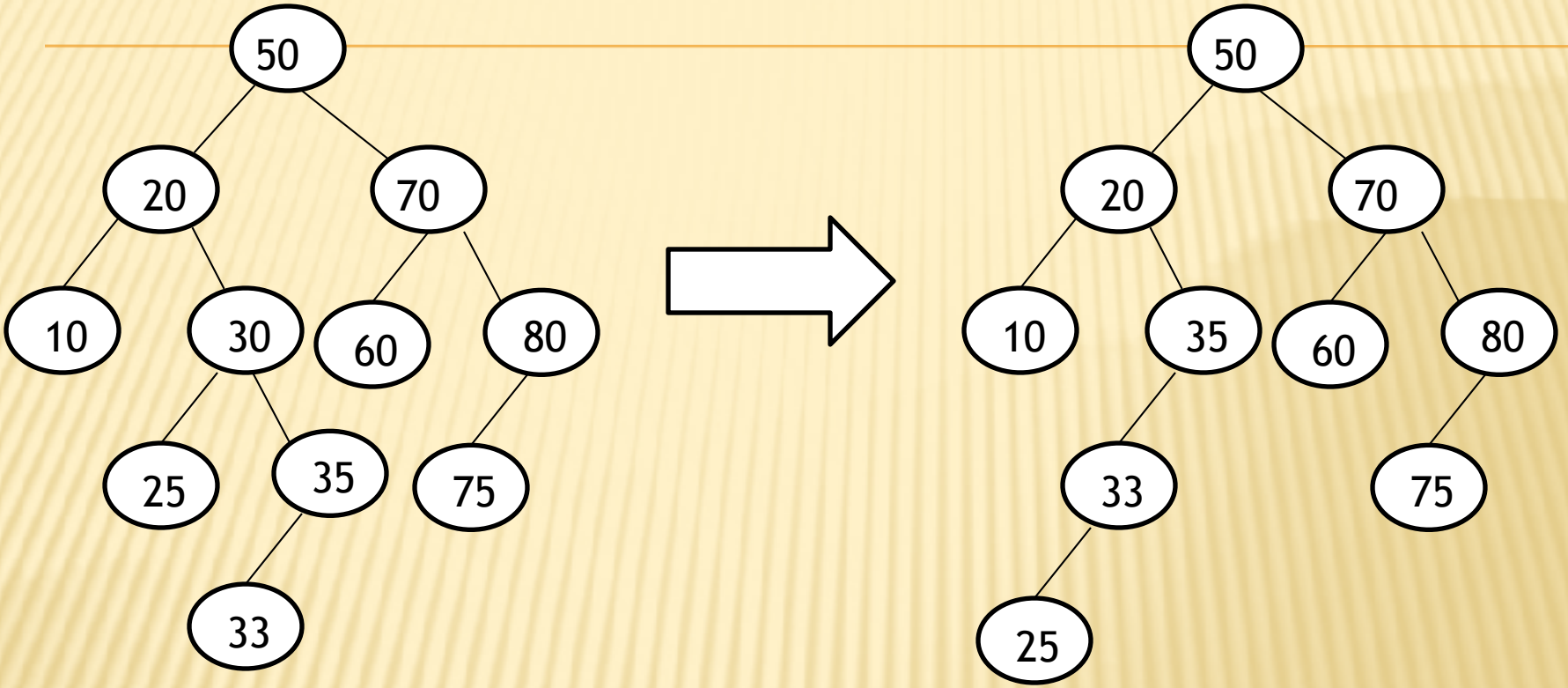
- Search for the item to be deleted by traversing the tree.
- If item not found, appropriate message is printed.
- If item is found, node is deleted and ordering of tree is maintained by adjusting the links.

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)

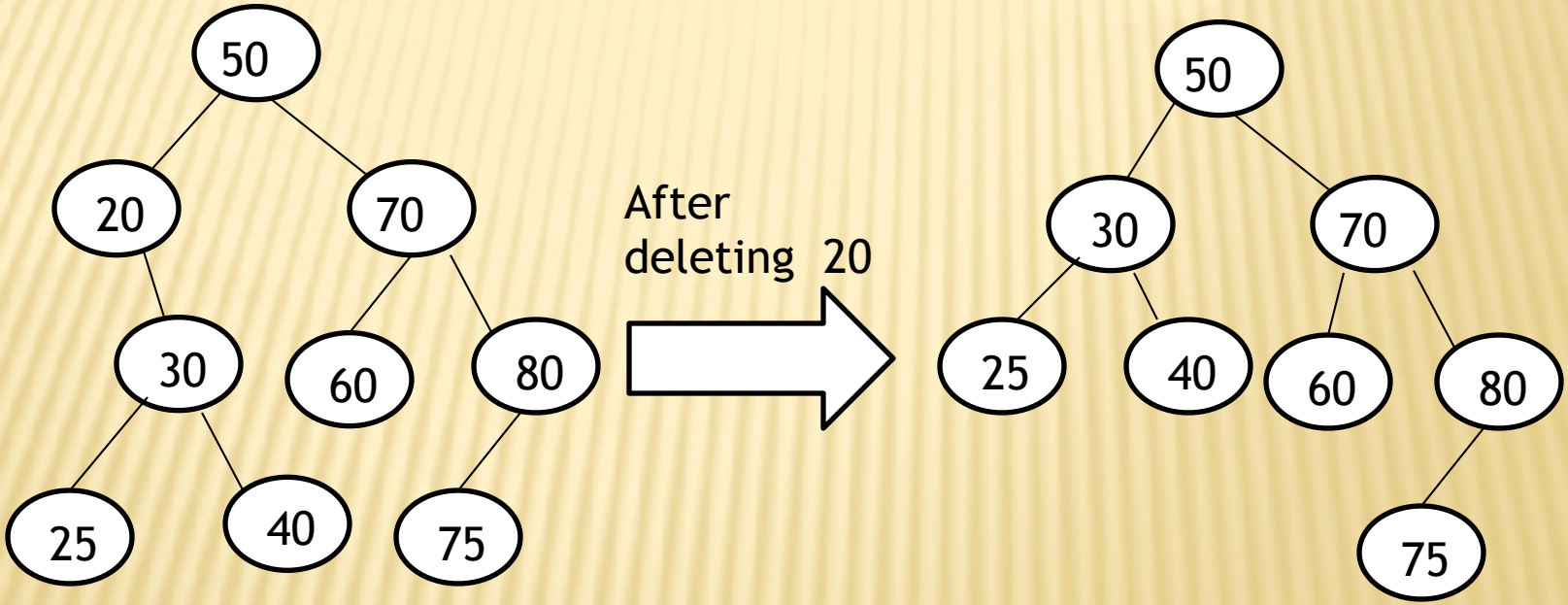


In a BST, node to be deleted will have 2 cases

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:

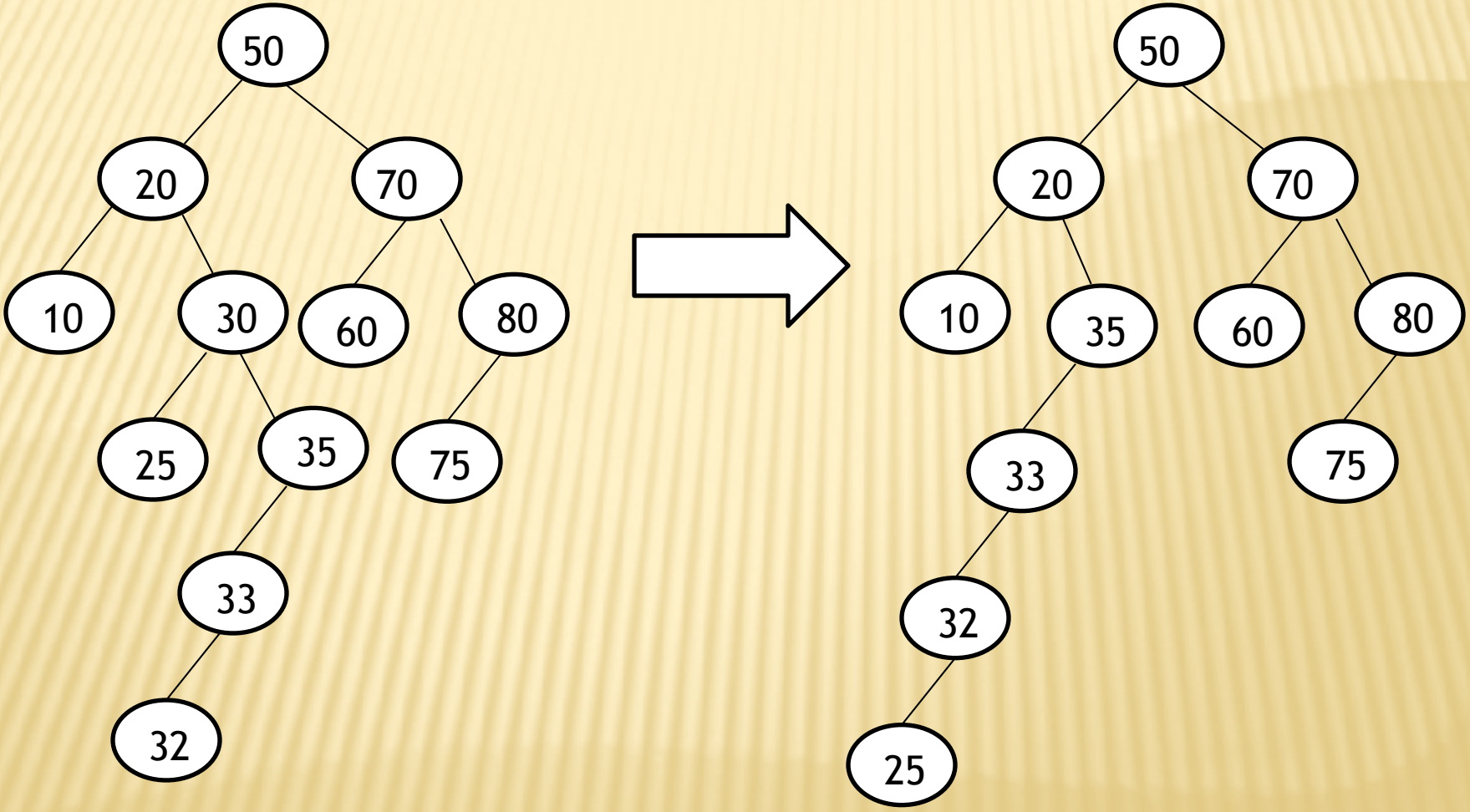
- Simple and straight forward.
- Parent of the node to be deleted is made to point to non empty subtree of the node to be deleted.
- Node to be deleted is freed.



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
    q=cur→rlink;                    subtree */


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

```
/*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*/
}
```

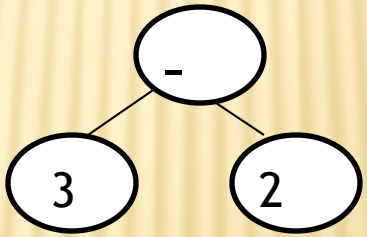
```
if(parent ==NULL)          /* if parent does not exist, return
   return q;                 q as root*/
if(cur==parent→llink)      /* if root to be deleted is left of
   parent→llink=q;           parent, connect left of parent to
                             q*/
else
   parent→rlink=q;         /*else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

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

Applications of binary trees

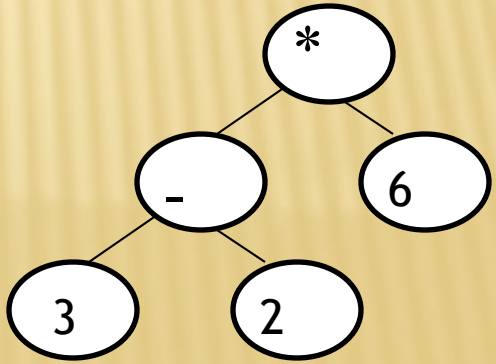
Conversion of expressions:

- 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

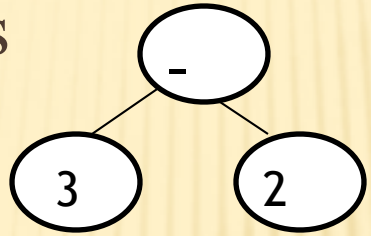


```
Void levelorder()
{
    InsertQ(root);
    while(!IsEmpty())
    {
        Node *temp;
        temp= DeleteQ();
        cout<<temp->data;
        if (temp->lchild) InsertQ(temp->lchild);
        if (temp->rchild) InsertQ(temp->rchild);
    }
}
```

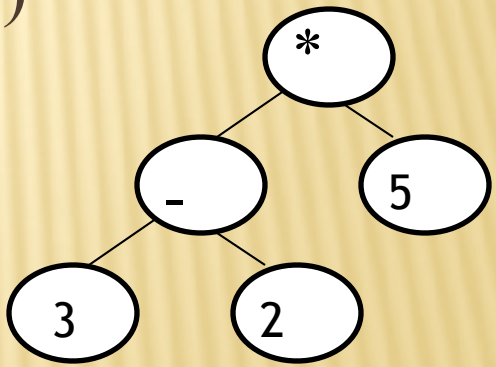
Represent the following expression using binary tree

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

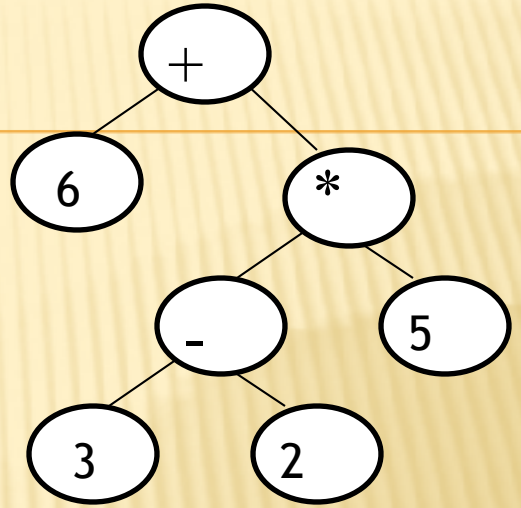
First innermost parenthesis is considered, which is (3-2) and partial tree is drawn for this



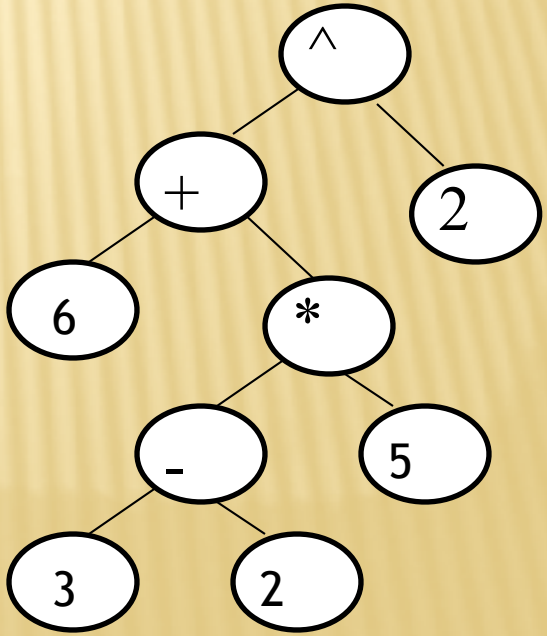
Next tree is extended to include ((3-2)*5)



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:

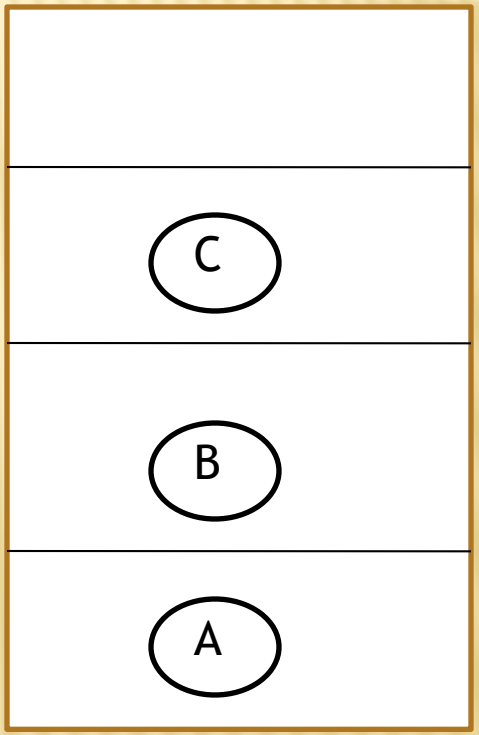
1. Scan the expression from left to right.
2. Create a node for each symbol encountered.
3. If the symbol is an operand, push the corresponding node on to the stack.
4. If the symbol is an operator, pop top node from stack and attach it to the right of the node with the operator. Next pop present top 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*/
                Push(temp);
         else                                  /* else pop element add it to the
        {                                          right of operator node. Pop

                temp→rlink=Pop();  next element and add  to

                temp→llink=Pop();  left. Push operator node*/
                Push(temp)
        }
    }
return(Pop());                          /* return root*/
}
```

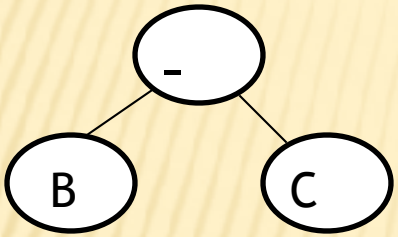
Create a binary tree for the given postfix expression:

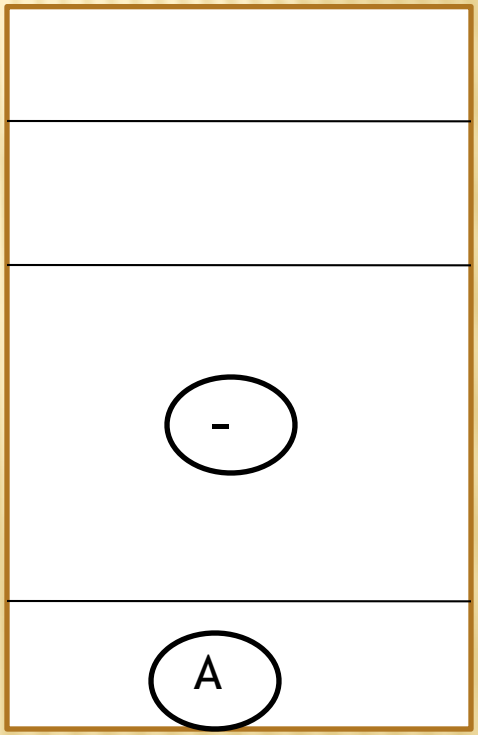
abc-d*+

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

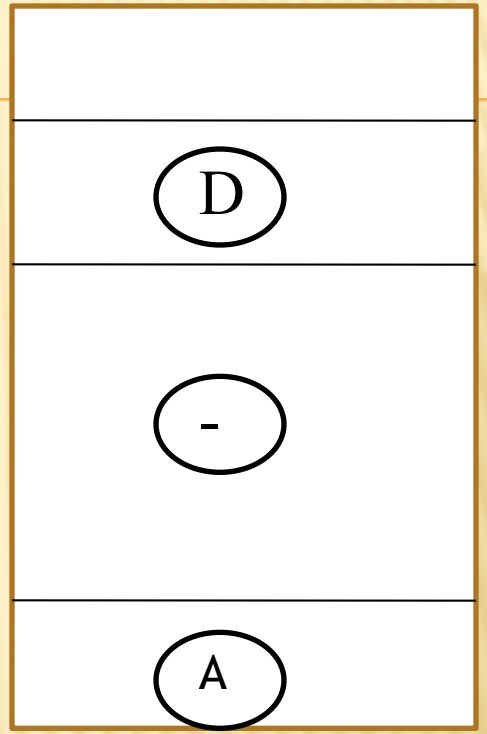


2. Now we get operator ‘–’. Pop top 2 elements and add them to right and left of node with ‘-’respectively and push node with operator ‘-’ to stack.

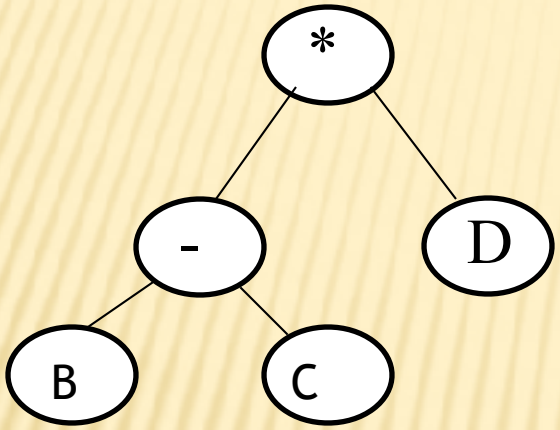


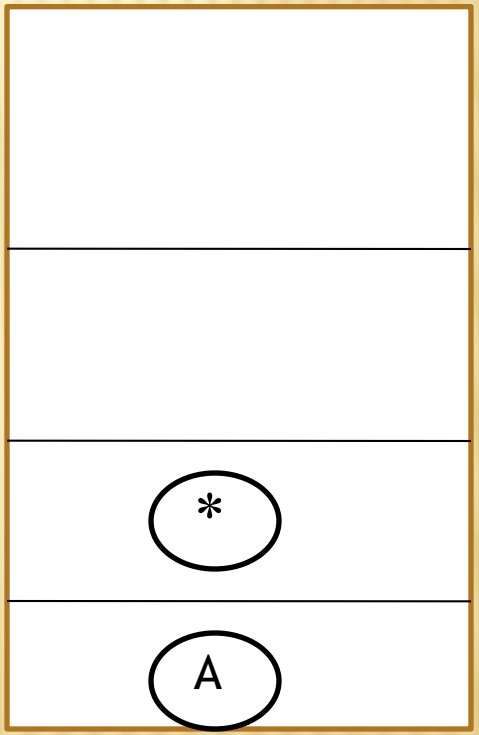


3. Push sumbol ‘D’ to stack

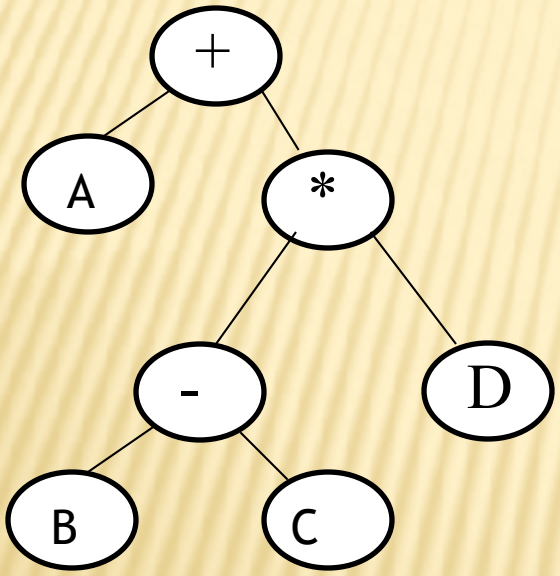


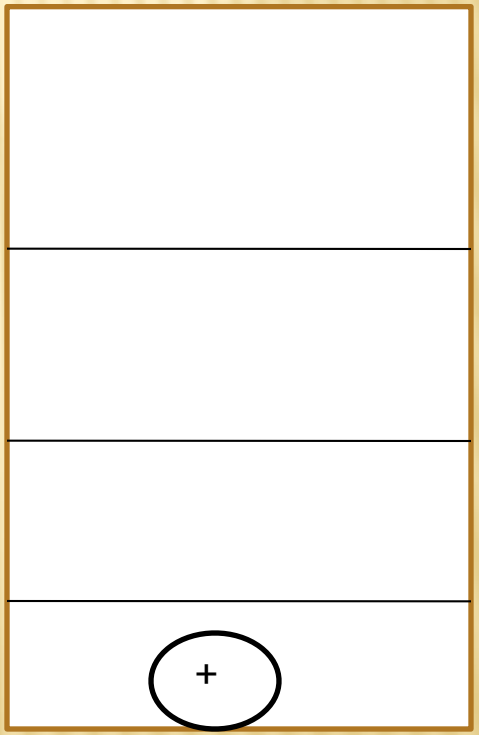
4. Now the operator is ‘*’. Pop top 2 elements and add it to right and left of node with ‘*’ respectively and push the operator node to stack.





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

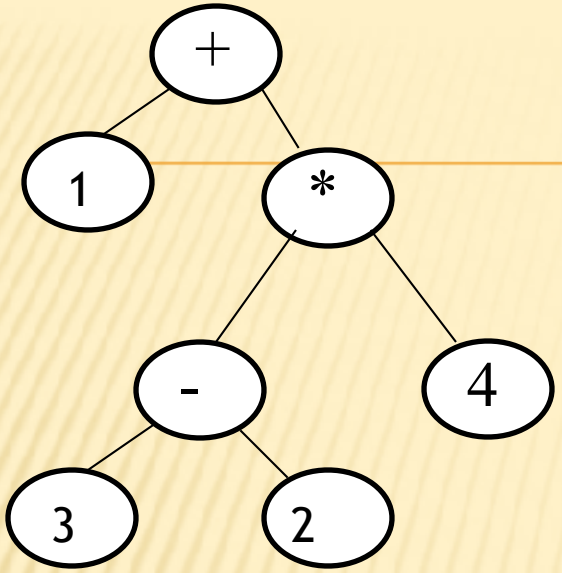




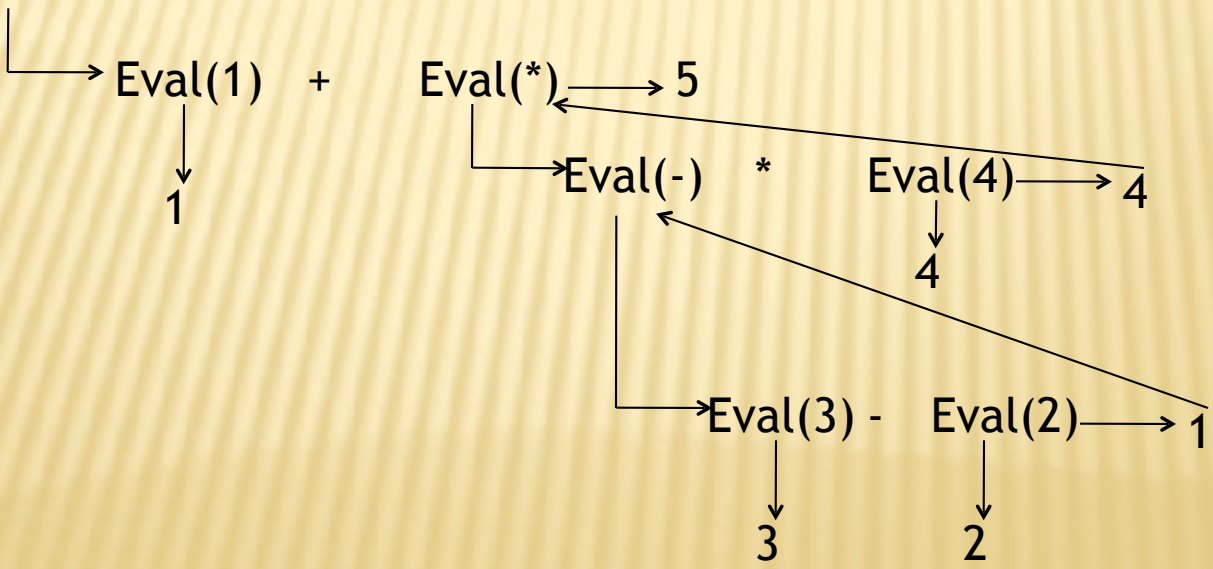
Now stack top will have the root of the final tree.

Evaluating the expression tree using recursion:

```
int 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 – '0');
    }
}
```

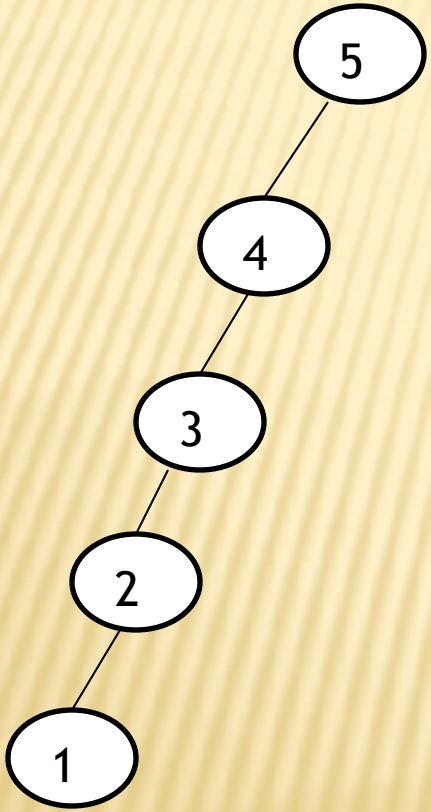


Eval(+)

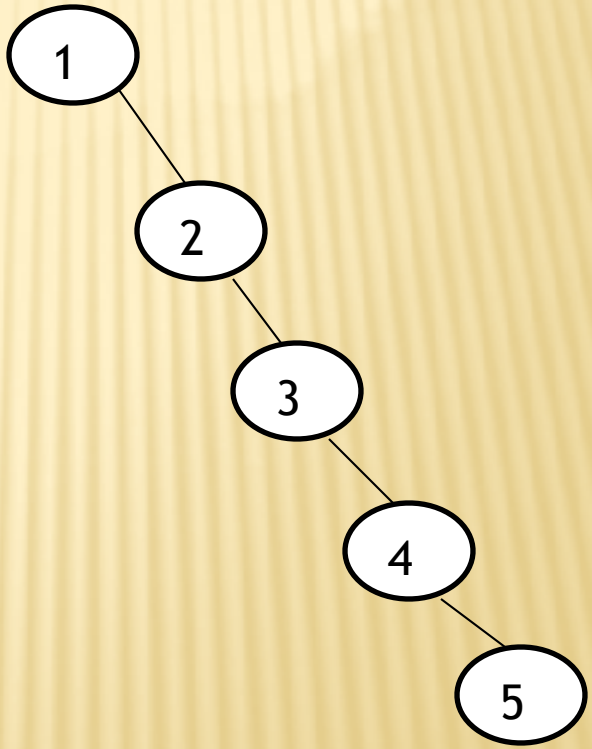


Balanced Trees

- Searching BST becomes inefficient if it gets right-skewed or left skewed.



left skewed



right skewed

- Right skewed and left skewed problem can be overcome by using the concept of AVL trees(a type of balanced trees).

AVL Trees

- In an AVL tree, the heights of the two child subtrees of any node differ by at most one. Therefore, it is also said to be height balanced.
- The AVL tree is named after its two inventors, G.M. Adelson-Velskii and E.M. Landis.
- The **balance factor** of a node is the height of its left subtree minus the height of its right subtree, and a node with balance factor 1, 0, or -1 is considered balanced.

Basic strategy of the AVL method

- After each insertion or deletion check whether the tree is still balanced.
- If the tree is unbalanced, restore the balance.

What is restoring the balance?

- Is nothing but bringing back the tree to balanced state(AVL state) by performing rotations on the trees.

What are different types of rotations?

- Single rotations(L and R)
- Double rotations(RL and LR)

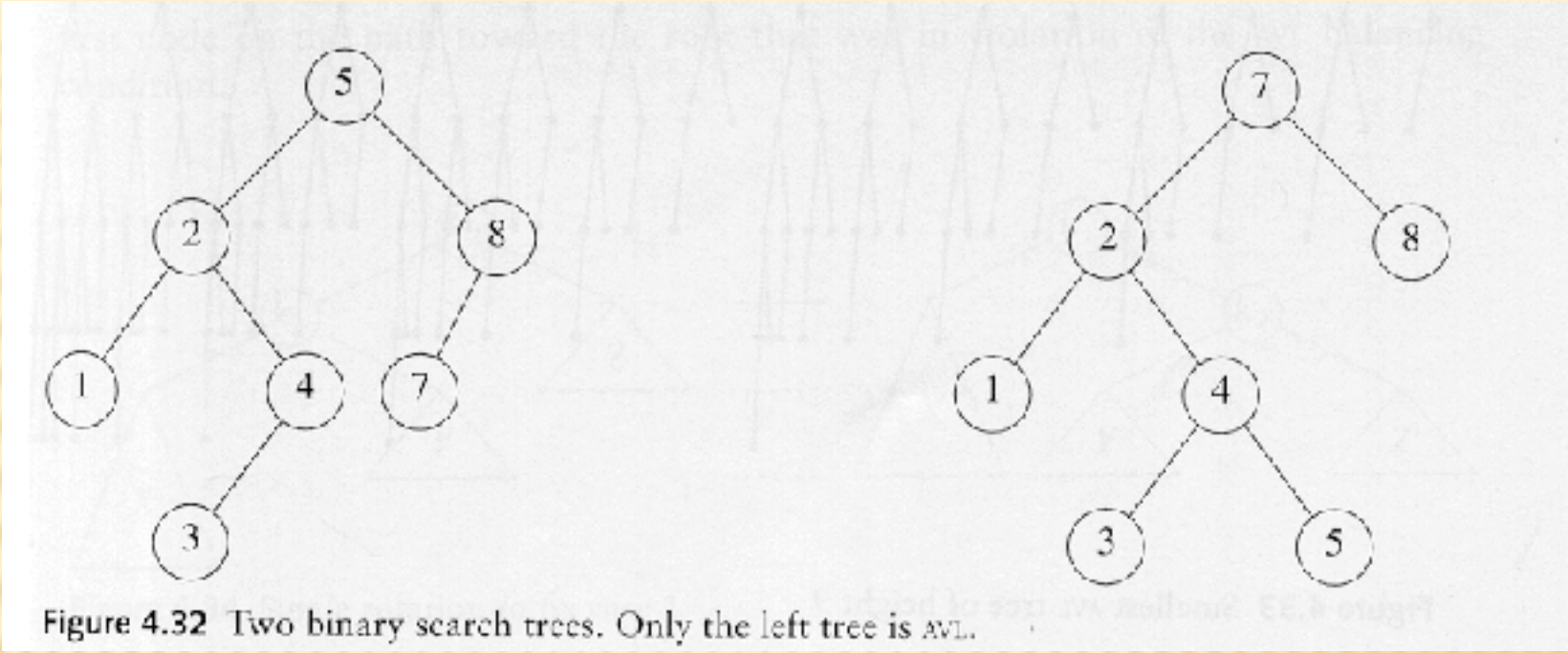
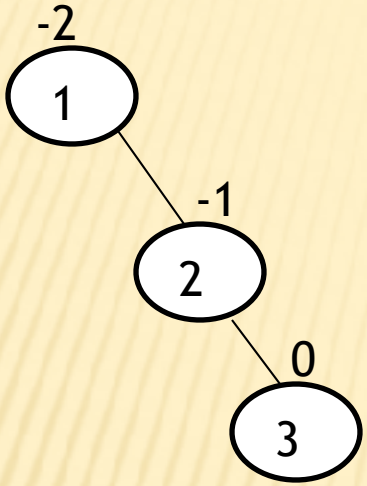


Figure 4.32 Two binary search trees. Only the left tree is AVL.

In left tree, balance factor of 5 is 1, 2 is -1, 4 is 1, 1 is 0, 3 is 0, 8 is 1 and 7 is 0. hence it is AVL tree.

In right tree, balance factor of 7 is 2, which violates AVL property. Hence it is not AVL tree. Rotation has to be performed on it to make it AVL.

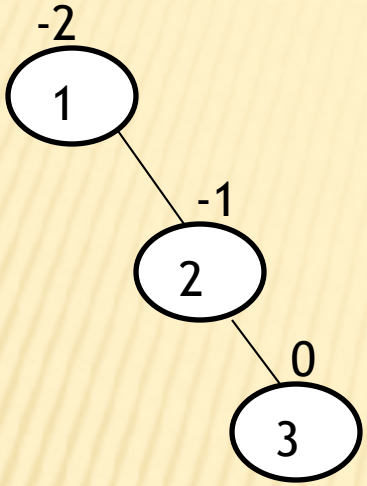
Inserting 1,2,3 ,4,5 into AVL tree



When 3 is inserted, AVL property is violated and hence rotation has to be performed to make it balanced before proceeding further.

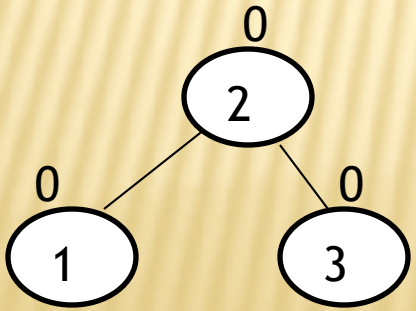
How to decide whether to perform a single or double ratation?

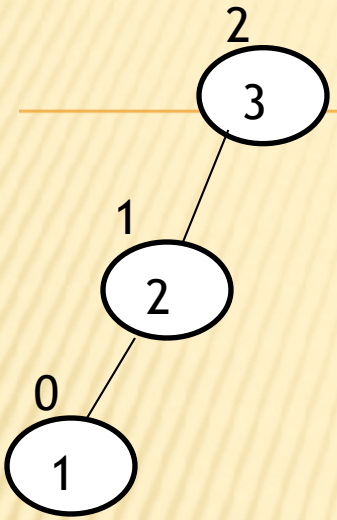
- Find the youngest ancestor which is out of balance(i.e not having balance factor of -1, 0 or 1) and restrict the attention to that node and two nodes immediately below the path.(path that we have followed from leaf node).
- If 3 nodes lie in a straight line, single rotation results in AVL tree.
- If 3 nodes do not lie in a straight line(a bend in the path), a double rotation is required.



All 3 nodes in a straigt line. Hence Single rotation will do.

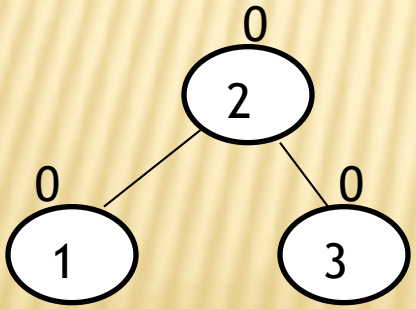
After performing single left(L) rotation at node 1



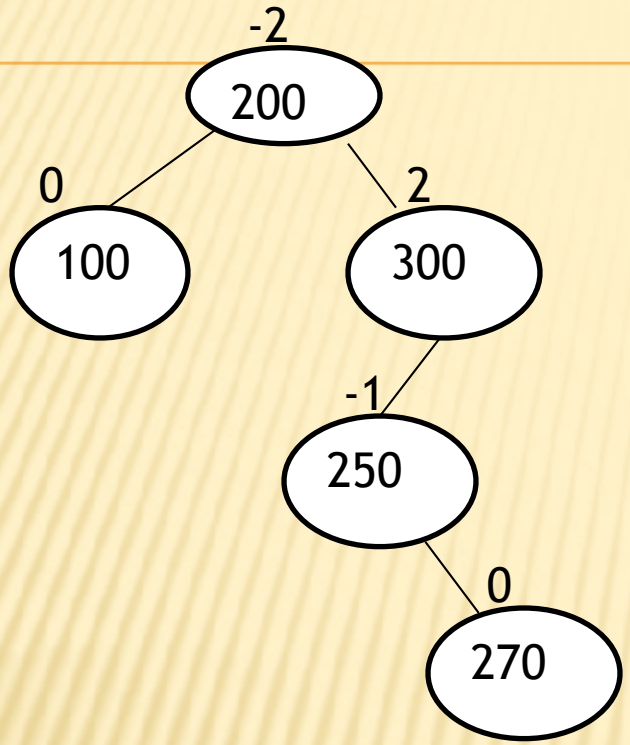


All 3 nodes in a straigt line. Hence Single rotation will do.

After performing single right(R) rotation at node 3



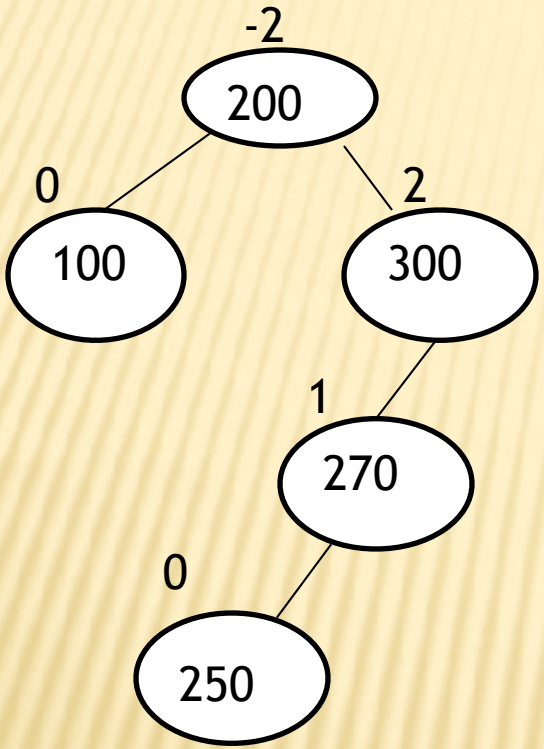
Now consider the following tree



Here youngest ancestor out of balance is 300. Two nodes (250 and 270) immediately below the path are not in a straight line . Here we need double rotations(two single rotations).

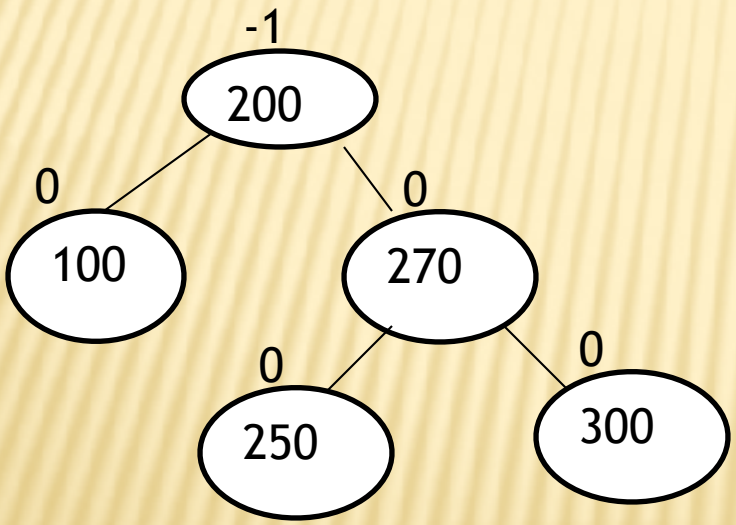
First single rotation is done at the first layer below the node where imbalance is detected. i.e at node 250.

After Rotating left at 250;(first single rotation(L))



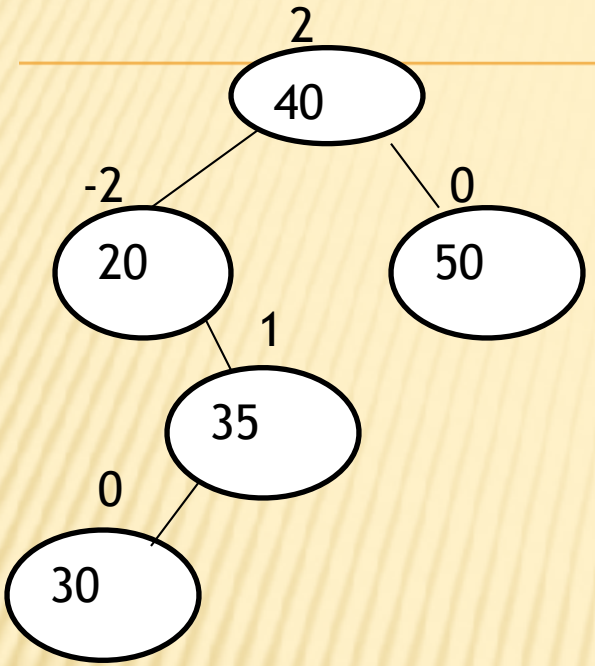
Second single rotation is done at the node where the imbalance was found i.e at node 300.

After Rotating right at 300: (second single rotation(R))



Hence after 2 rotations (LR) rotations tree is balanced.

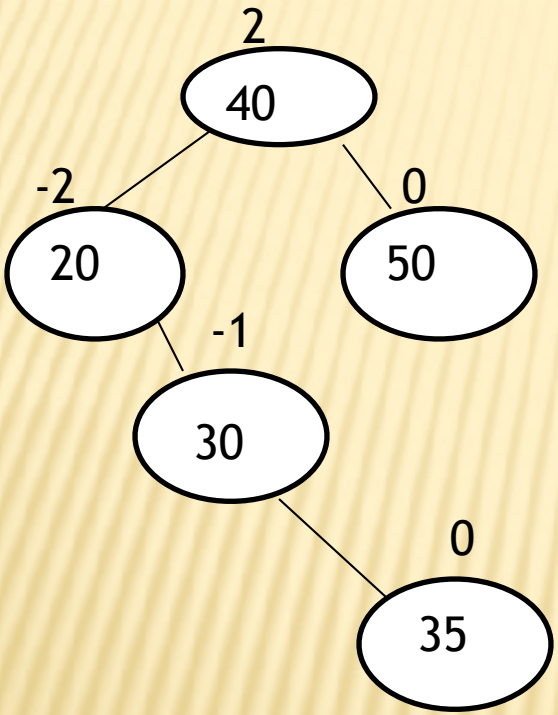
Consider the following tree



Here youngest ancestor out of balance is 20. two nodes (35 and 30) immediately below the path are not in a straight line . Here we need double rotations(two single rotations).

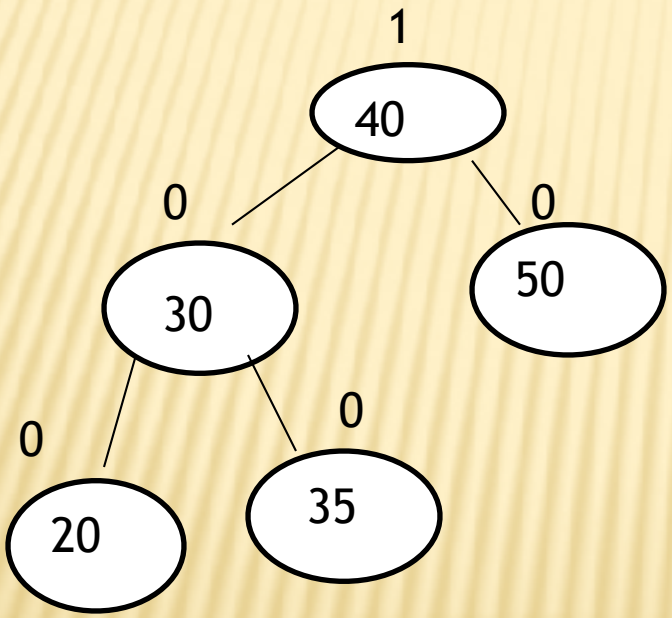
First single rotation is done at the first layer below the node where imbalance is detected. i.e at node 35.

After Rotating right at 35:(first single rotation(R))



Second single rotation is done at the node where the imbalance was found i.e at node 20.

After Rotating left at 20: (second single rotation(L))



Hence after 2 rotations (RL) rotations tree is balanced.

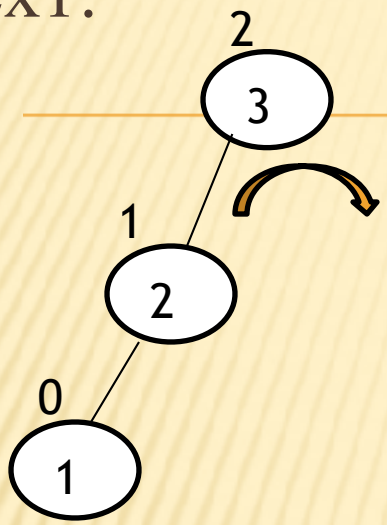
Single R-Rotation:

- If there is a subtree whose balance factor is 1(left heavy) and if an item is inserted to the left of its left subtree resulting in a straight line, a single right rotation is sufficient to rebalance the tree.

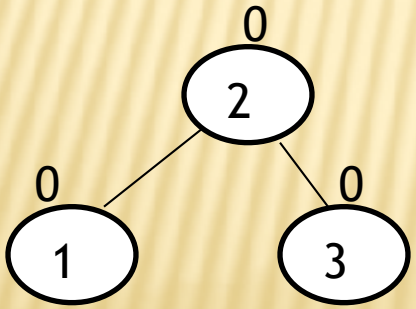
Algo:
I/P: The node X where right-rotation takes place.
O/P: returns root of right-rotated subtree.

- Step 1: obtain the left child of node to be rotated.
  Y=left(X)
- Step 2: obtain the right child of Y.
  Z=right(Y)
- Step 3: rotate towards right.
  right(Y)=X
  left(X)=Z
- Step 4: return the root of right rotated subtree.
  return Y

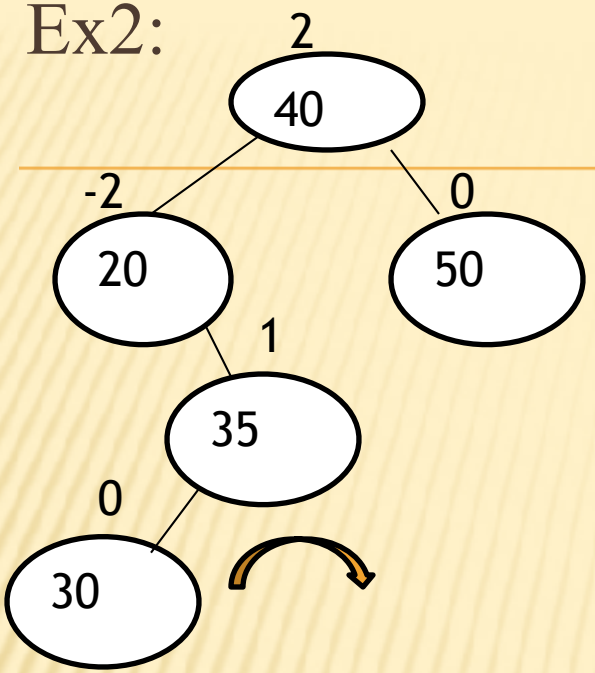
Ex1:

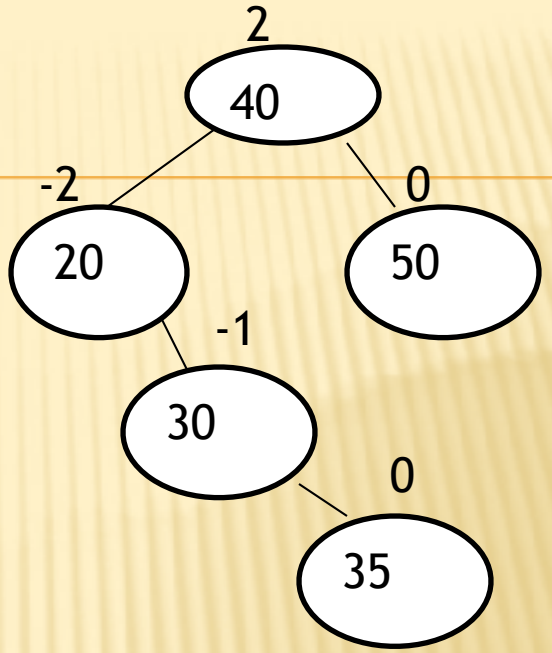


Imbalance found in node 3 and two nodes below it form straight line . Since left subtree is heavy (B.F 2), a single R rotation is performed to balance it.



- Ex2:





Youngest ancestor where imbalance found is node 20 and two nodes below it do not form a straight line. We need double rotation : first rotation occurs at first layer below 20 i.e 35. since B.F of 35 is 1 i.e left heavy, a right rotation is performed on 35 to make it…….

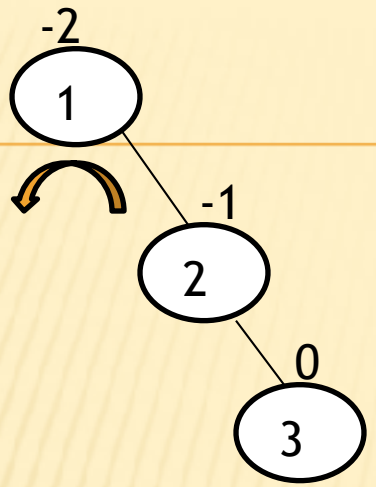
Single L-Rotation:

- If there is a subtree whose balance factor is -1(right heavy) and if an item is inserted to the right of its right subtree resulting in a straight line, a single left rotation is sufficient to rebalance the tree.

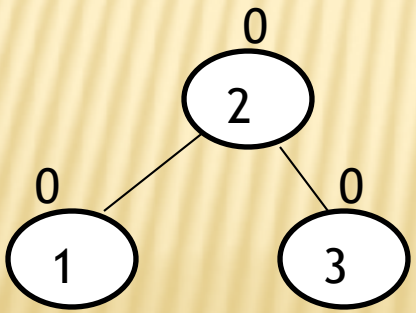
Algo:
I/P: The node X where left-rotation takes place.
O/P: returns root of left-rotated subtree.

- Step 1: obtain the right child of node to be rotated.
  Y=right(X)
- Step 2: obtain the left child of Y.
  Z=left(Y)
- Step 3: rotate towards left.
  left(Y)=X
  right(X)=Z
- Step 4: return the root of left rotated subtree.
  return Y

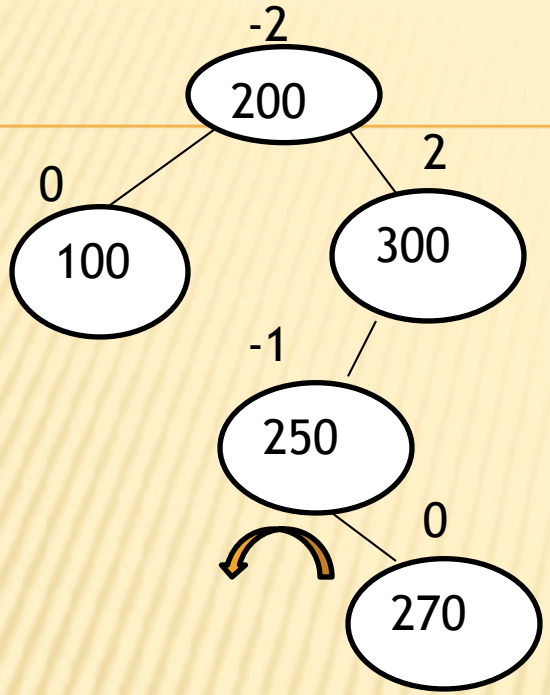
Ex1:

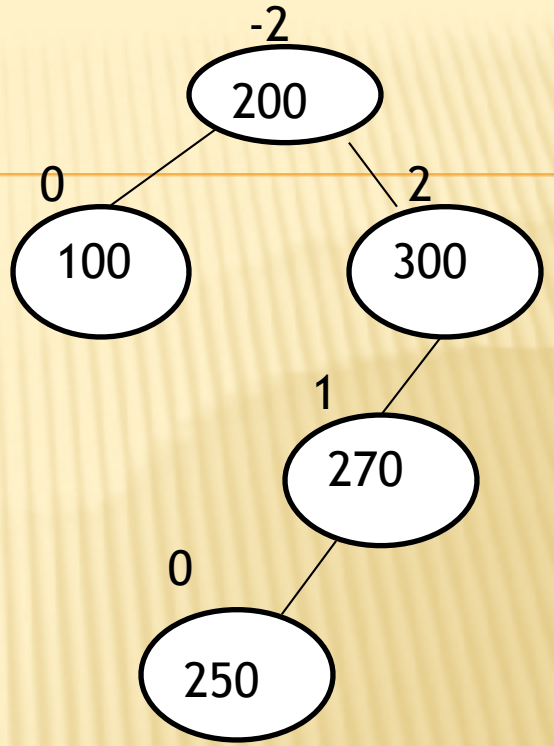


Imbalance found in node 1 and two nodes below it form straight line . Since right subtree is heavy (B.F -2), a single L rotation is performed to balance it.



Ex2:





Youngest ancestor where imbalance found is node 300 and two nodes below it do not form a straight line. We need double rotation : first rotation occurs at first layer below node 300 i.e 250. since B.F of 250 is -1 i.e right heavy, a left rotation is performed on 250 to make it…….

Base line:

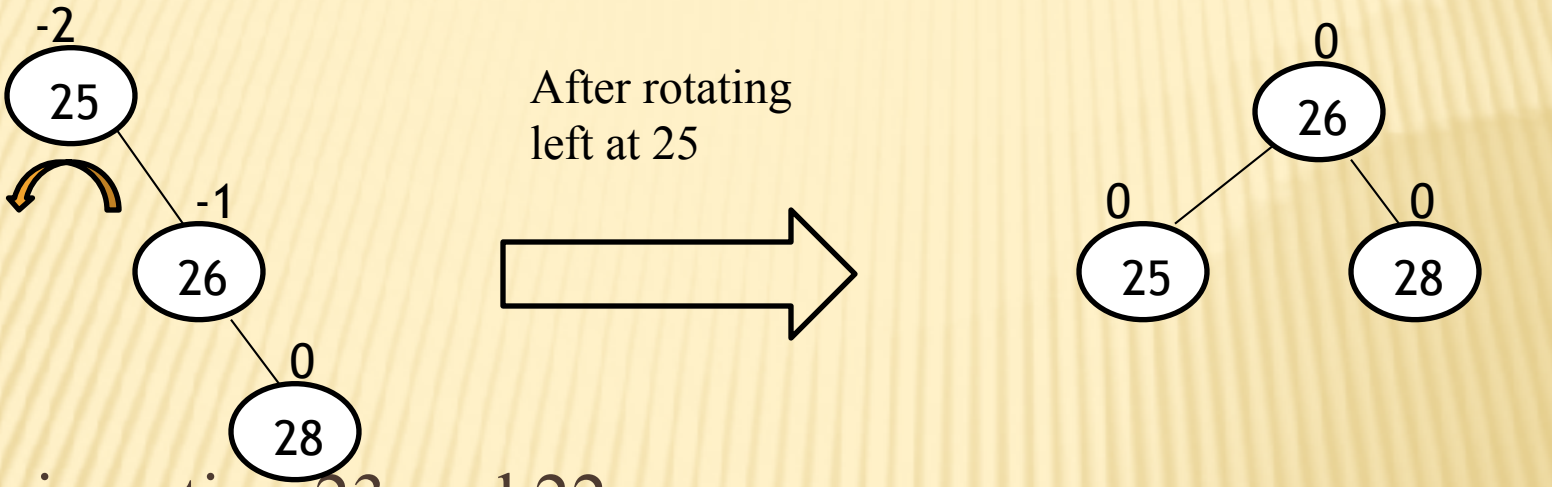
- If the three nodes lie in a straight line and
  - If unbalanced node is right heavy, perform left rotation.
  - If unbalanced node is left heavy, perform right rotation.
- If three nodes do not lie in a straight line and
  - If First layer node below imbalanced node is right heavy, perform a left rotation on that node. Then perform a right rotation on the imbalanced node.
  - If First layer node below imbalanced node is left heavy, perform a right rotation on that node. Then perform a left rotation on the imbalanced node.

Exercise:

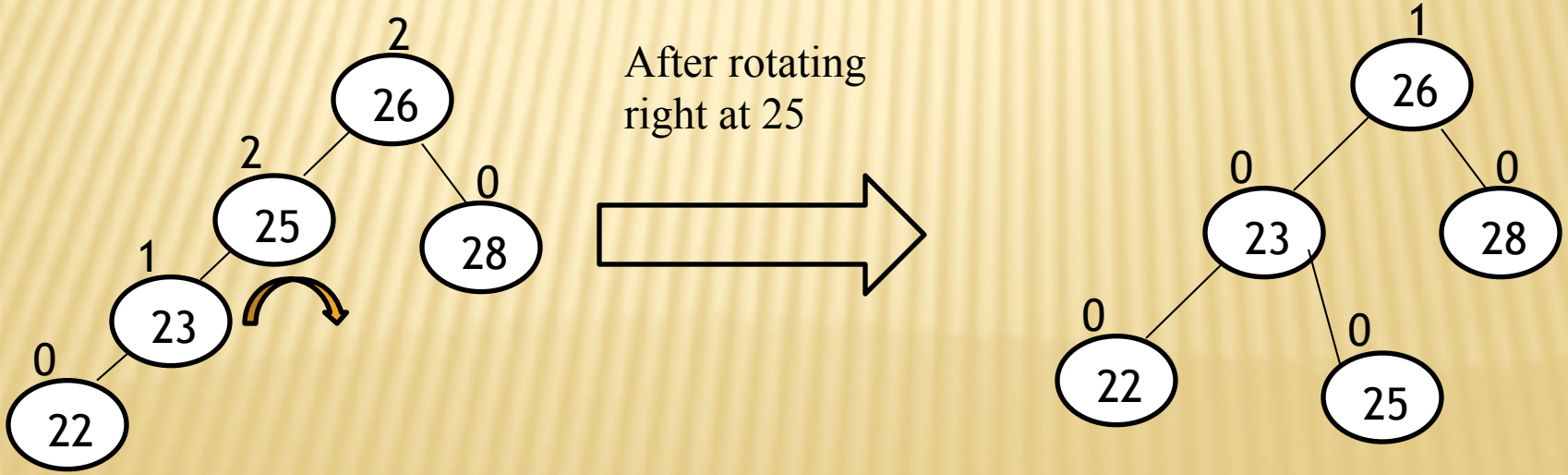
1. Construct an AVL tree for items 25, 26, 28, 23, 22, 24, and 27.(check if tree is balanced after every insertion. If not balanced, balance it by performing rotations)
2. Construct an AVL tree for the items 1,2,3, 4, 5 and 6.
3. Construct an AVL tree for the items 4, 2, 1, 3, 6, 5, 7.

Exercise1:

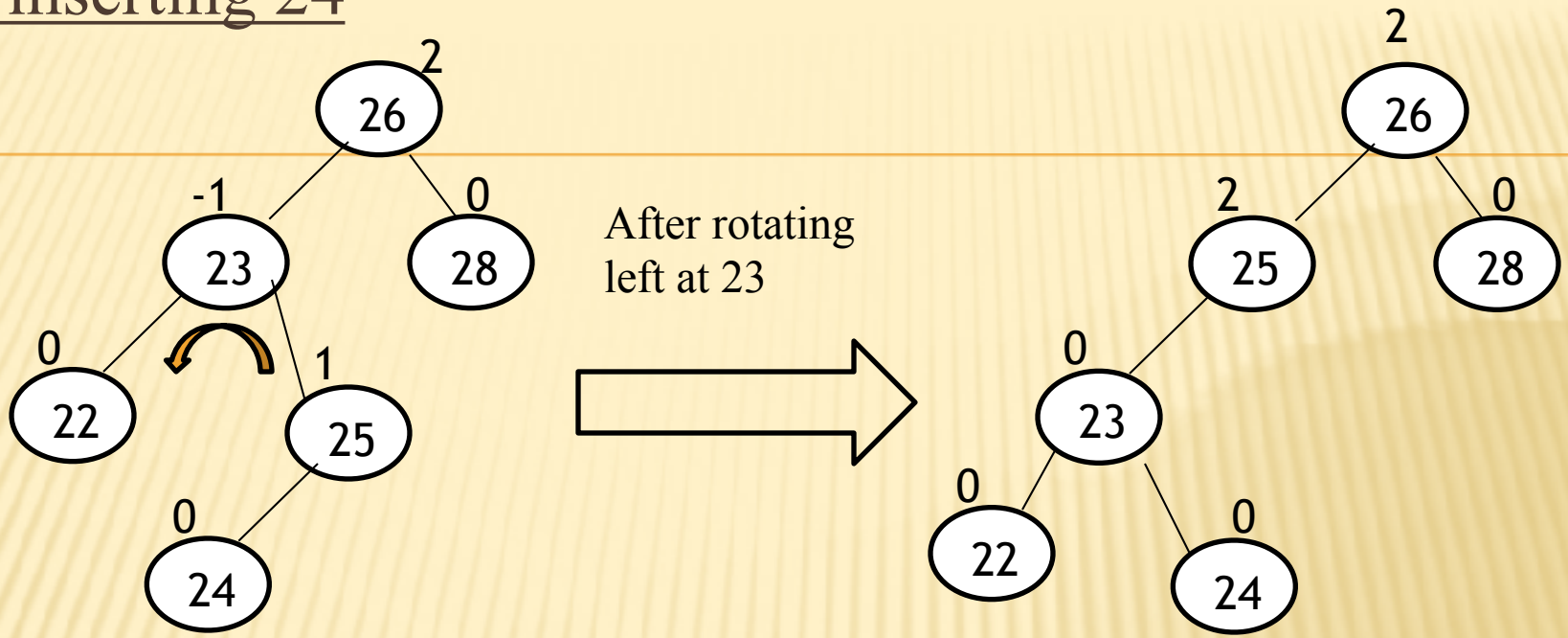
After inserting 25,26 and 28

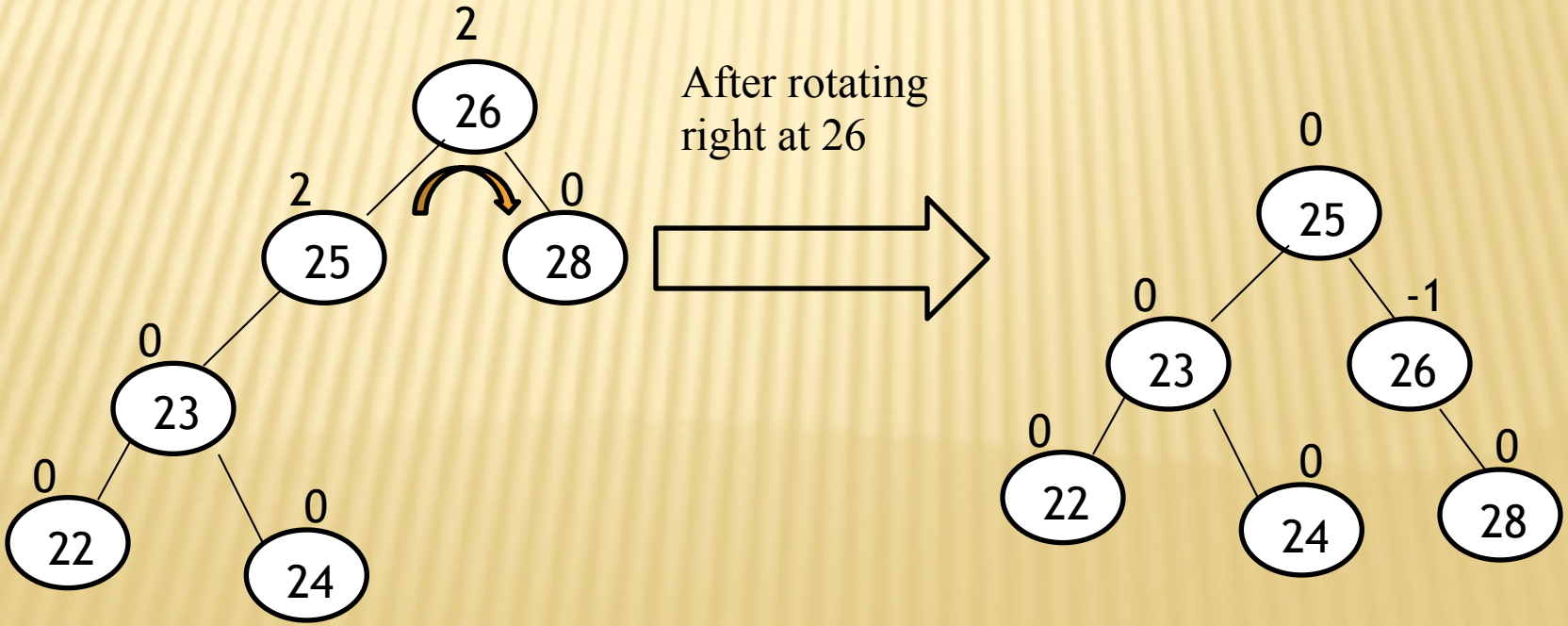


After inserting 23 and 22

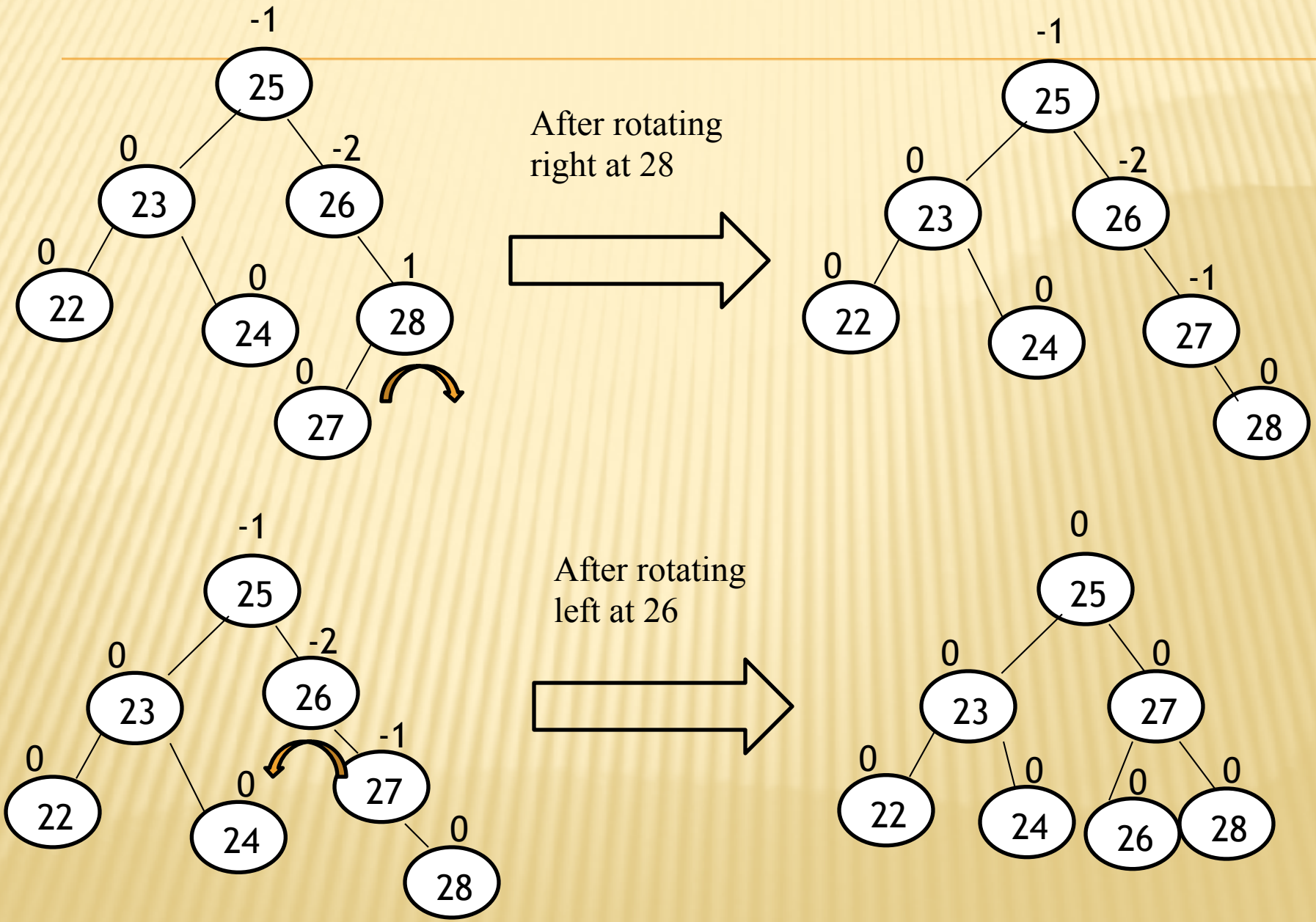


After inserting 24





After inserting 27



PRIORITY QUEUES

- In a priority queue, the element to be deleted is the one with highest (or lowest) priority.
- An element with arbitrary priority can be inserted into the queue according to its priority.
- A data structure supports the above two operations is called max (min) priority queue.

EXAMPLES OF PRIORITY QUEUES

- Suppose a server that serve multiple users. Each user may request different amount of server time. A priority queue is used to always select the request with the smallest time. Hence, any new user's request is put into the priority queue. This is the min priority queue.
- If each user needs the same amount of time but willing to pay more money to obtain the service quicker, then this is max priority queue.

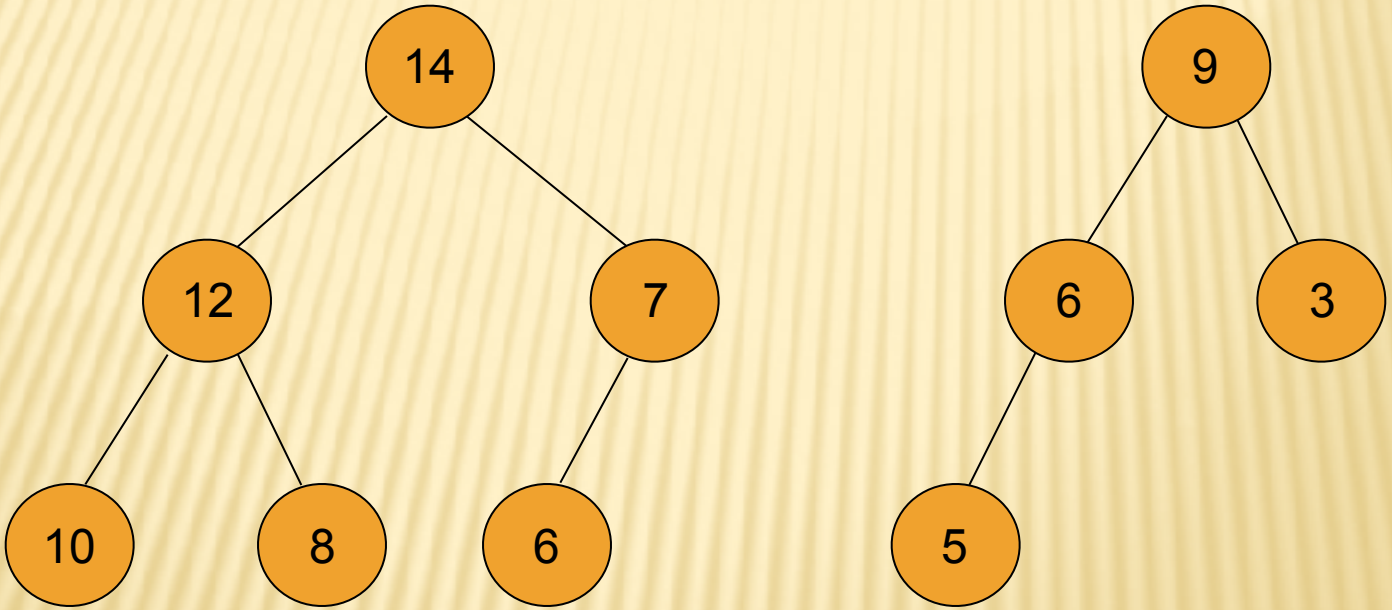
PRIORITY QUEUE REPRESENTATION

- Unordered Linear List
  - Addition complexity: O(1)
  - Deletion complexity: O(n)
- Chain
  - Addition complexity: O(1)
  - Deletion complexity: O(n)
- Ordered List
  - Addition complexity: O(n)
  - Deletion complexity: O(1)

MAX (MIN) HEAP

- Heaps are frequently used to implement priority queues.
- Definition: A max (min) tree is a tree in which the key value in each node is no smaller (larger) than the key values in its children (if any). A max heap is a complete binary tree that is also a max tree. A min heap is a complete binary tree that is also a min tree.

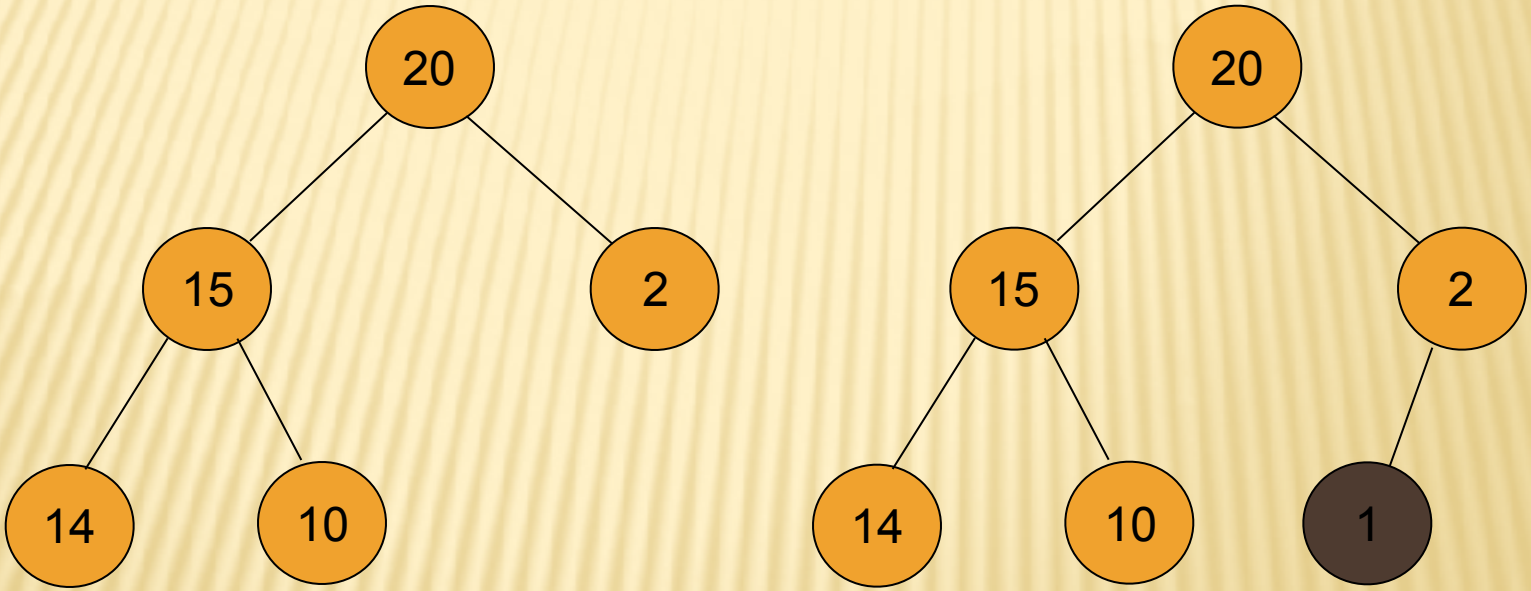
MAX HEAP EXAMPLES



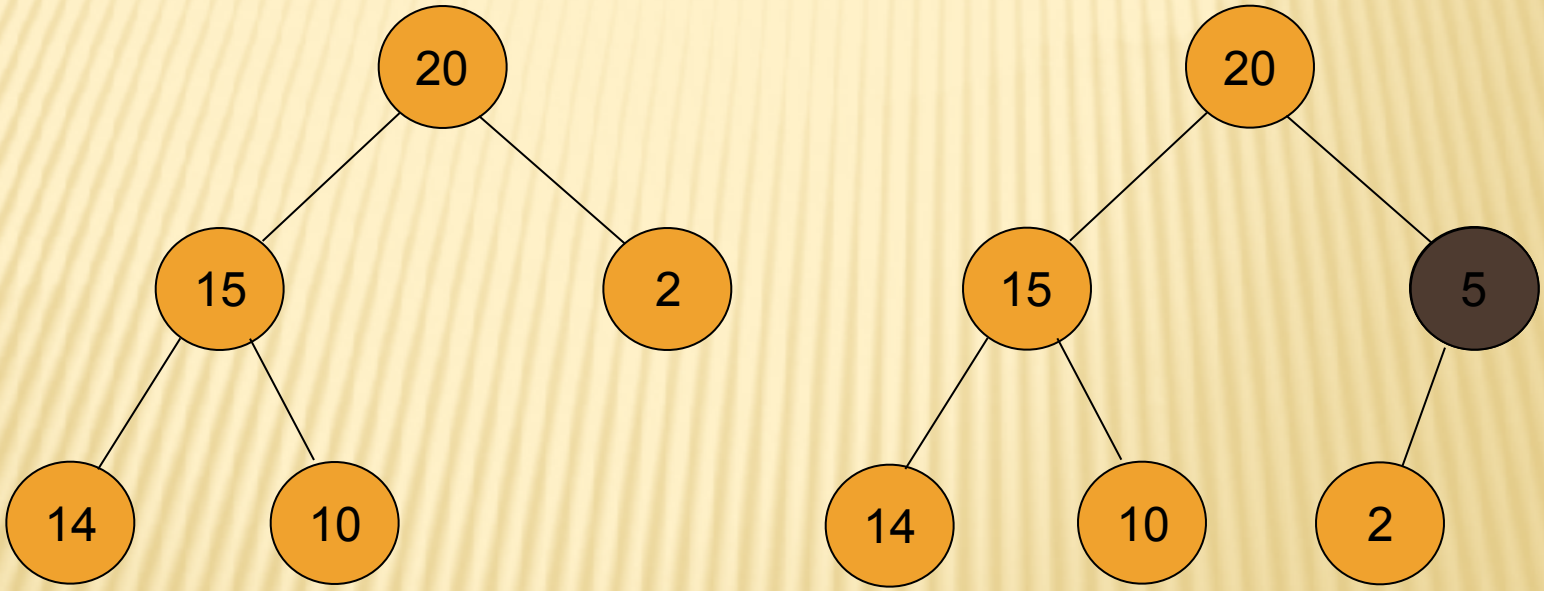
INSERTION INTO A MAX HEAP

```
template <class Type>
void MaxHeap<Type>::Add(const Type &e)
{ // insert e into the max heap
      if (n==Maxsize) { HeapFull(); return;}
      int currentNode = ++n;
      while (currentNode != 1 && heap[currentNode/2] < e)
      { // bubble up
            heap[currentNode] = heap[currentNode/2]; // move parent down
            currentNode /= 2;
      }
      heap[currentNode] = e;
}
```

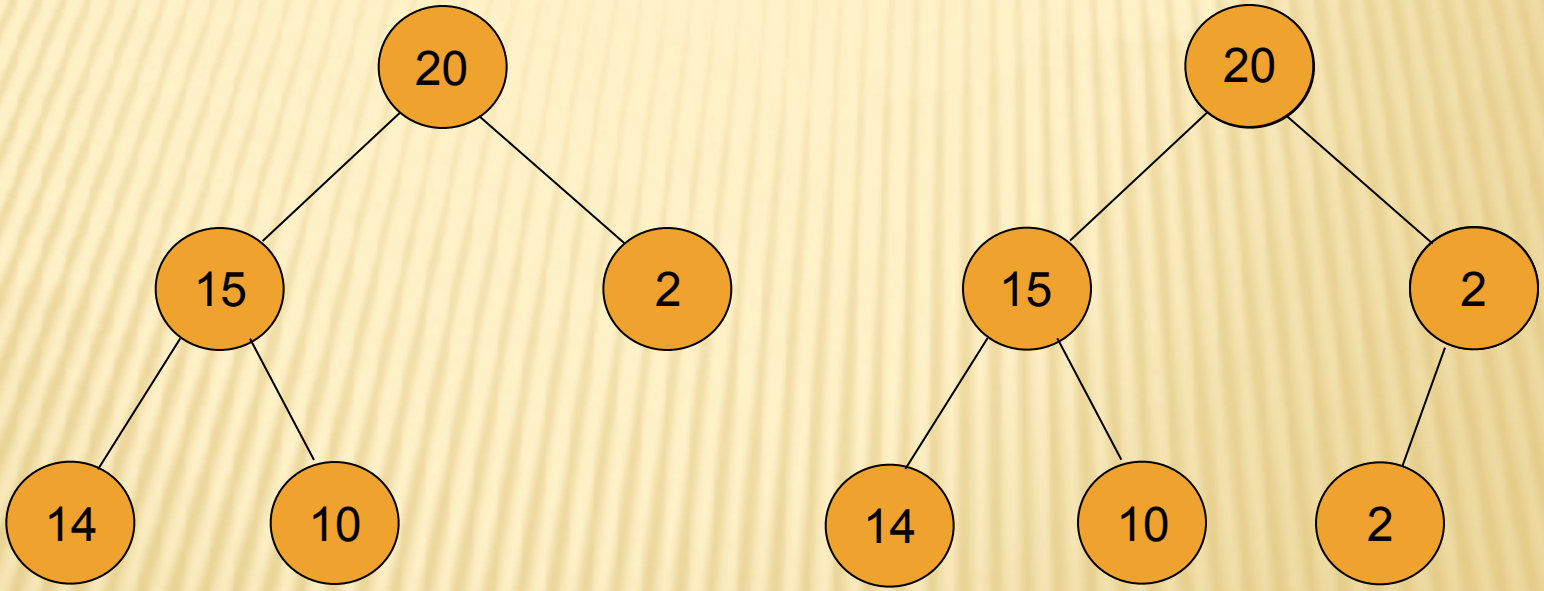
INSERTION INTO A MAX HEAP (INSERTING “1”)



INSERTION INTO A MAX HEAP (INSERTING “5”)



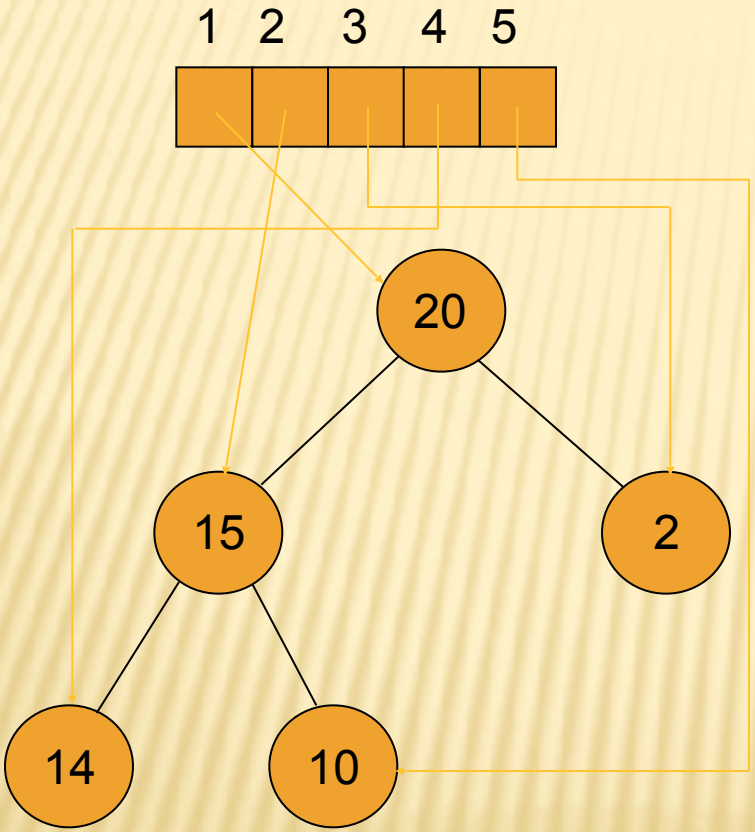
INSERTION INTO A MAX HEAP (INSERTING “21”)

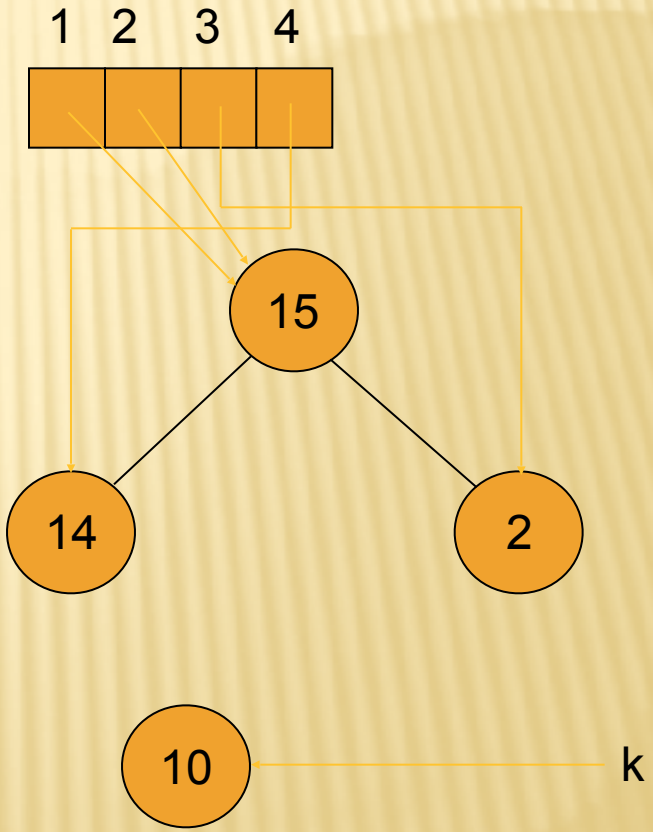


DELETION FROM A MAX HEAP

```
template <class Type>
void MaxHeap <Type>::DeleteMax()
{  // Delete from the max heap
      if (!n)   { HeapEmpty() ; return 0; }

       Type  x = heap[1];

      Type lastE = heap[n--]; // remove last element


      for(int i=1,j=2;j<=n;)
      {
            if (j<n)
                 if (heap[j]<heap[j+1]  j++; // j points to larger child
            if lastE>=heap[j]) break;
            heap[i]=heap[j]; /move larger child up
            i=j; j*=2; //move I and j down
      }
      heap[i] = lastE;
      return x;
}
```

DELETION FROM A MAX HEAP (CONT.)





i = 1

j = 2

n = 4

DELETION FROM A MAX HEAP (CONT.)