Tree (Hierarchical data structure)

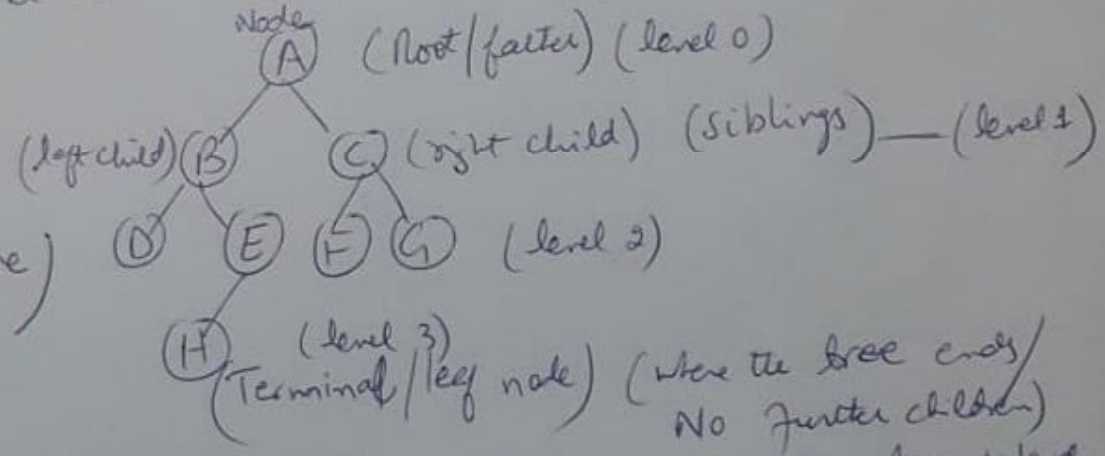
Limitation of Linked list — linear data structure
Worst $T(n) \in O(n)$

$T(n)$ of Tree $\in O(\log_2 n)$ (useful data structure)

(Tree)

Binary Tree (BT) — Finite set of nodes.

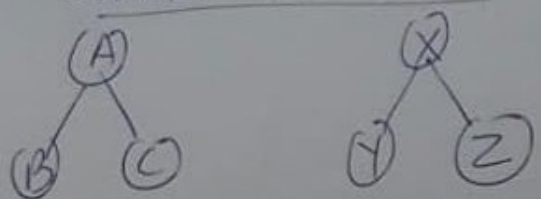
(i) Either empty
(ii) Consists of a node called root with left and right subtrees.



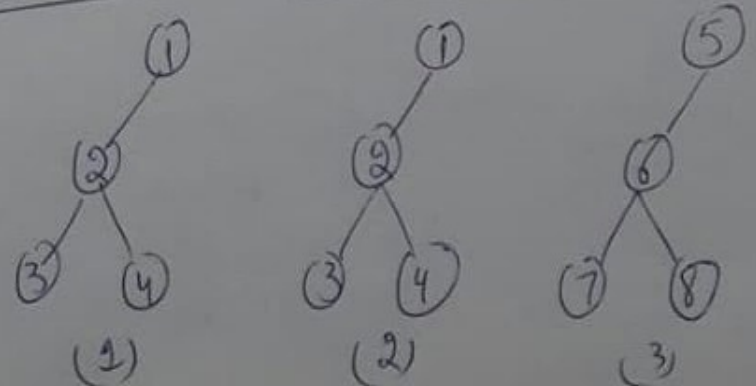
(Inner nodes are B, C, E)

Depth/Height of a tree — 1 more than the largest level.

Similar trees — Similar data structure

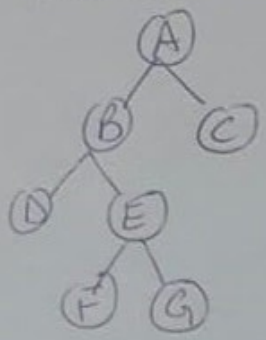


Copies trees — Similar data structure as well as same data

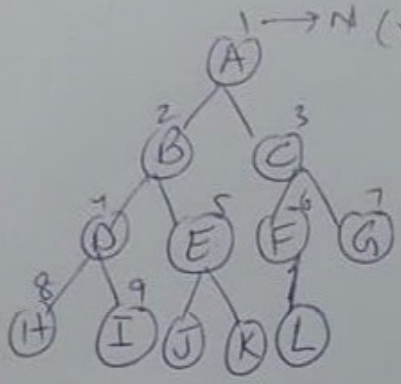


All are similar but (1) and (2) are copies trees.

Strictly BT Every internal node has its nonempty left and right children.



Complete BT Each node has both left and right children except last node.



Main Advantage Left/right child can be easily found. Parent can also be found.

Parent of I = $9/2 = 4$ (only integer part is taken). $(N/2)$

Right child $(2N+1)$ of D = $2 \times 4 + 8 + 1 = 9$ (i.e., I)

left child of D = $2N = 2 \times 4 = 8$ (ie., H).

Depth = $\log_2 N + 1$.

If N = 1000, Depth is only $\log_2 1000 + 1 = 10 + 1 = 11$

Algebraic expressions Representation in BT

already done when converted the infix to postfix and prefix.

Three traversals are done ie., Inorder traversal (left, root, right)
preorder traversal (root, left, right)
Postorder traversal left, right, root)

Level order traversal is left which will be done in the next pages.

BST (Binary search Tree)

Each node is greater than its left children and less than the right children.

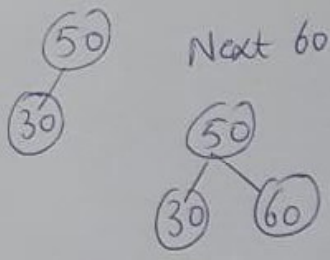
Creation/Insertion, Deletion, Traversal (Three data structure operations)

50, 30, 60, 22, 38, 55, 54, 34, 21, 100, 52 (n = 11)

Creation/Insertion First node is always root.

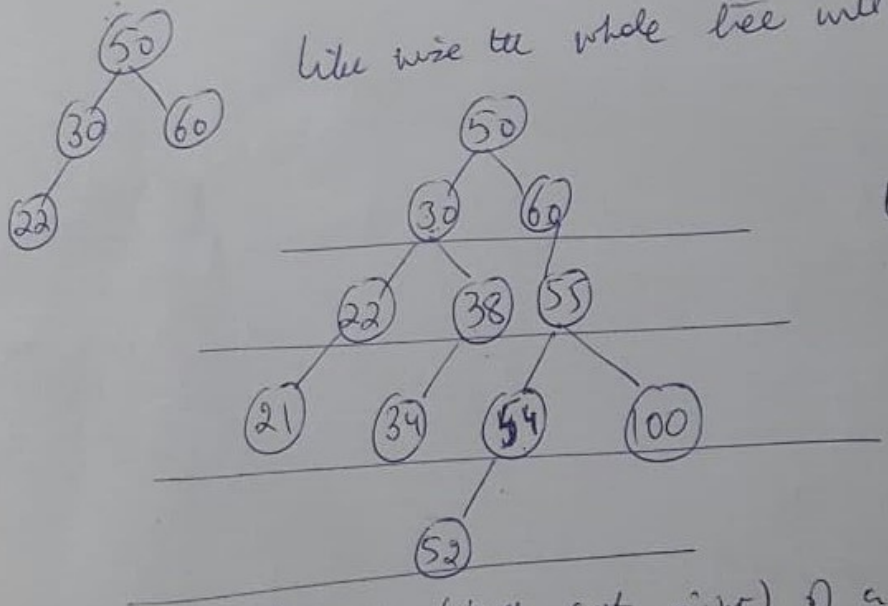
(50) Now as 30 is less, so it will be inserted on the left side

Next 60 is greater than 50, so on the right side.



Similarly 22 is less than the 50, we move to the left and then again less than 30, so it will be inserted on the left of 30.

Like wise the whole tree will be:



levels should be aligned.

Inorder traversal (left, root, right) of a BST always gives sorted data. If this does not happen, there must be some mistake in the creation of tree.

Inorder traversal 21, 22, 30, 34, 38, 50, 52, 54, 55, 60, 100

Code of Insertion in BST

```
void insert(int item)
{
    find(item);

    temp = new node();
    temp->info = item;
    temp->lchild = NULL;
    temp->rchild = NULL;
    if (parent == NULL) root = temp;
    else if (item < parent->info)
        parent->lchild = temp;
    else parent->rchild = temp;
}
```

find() function finds the location where a new node is to be inserted.

```
if item < root->info ptr = root->lchild;
    else ptr = root->rchild;
    ptrsave = root;
if 30 is to be inserted,
        ptr = root->lchild = NULL
```

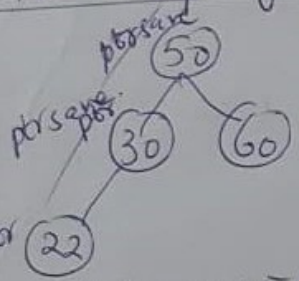
(50) ptrsave root

So while loop as given under will not execute:

```
while (ptr != NULL)
{
    if (item == ptr->info)
    {
        location = ptr;
        parent = ptrsave;
        return;
    }
    ptrsave = ptr;
    if (item < ptr->info)
        ptr = ptr->lchild;
    else ptr = ptr->rchild;
}
location = NULL;
parent = ptrsave;
```

These line values will be returned by the find() function.

Another example of find function:



For 21 to be inserted, find function will work as follows:

item < root->info (21 < 50)

ptr = root->lchild i.e, ptr = 30
and ptrsave = root i.e 50.

Now, as ptr != null, so while loop as given above will execute and will terminate when ptr = ptr->lchild = null
i.e, ptr = left child of 22.

So loop will give at the end location = null i.e, 21 is not present.
and parent = ptrsave i.e., 22

Which means that when it will go back to the insert function it will insert 21 as the left child of 22.

Deletion

(5) cases.

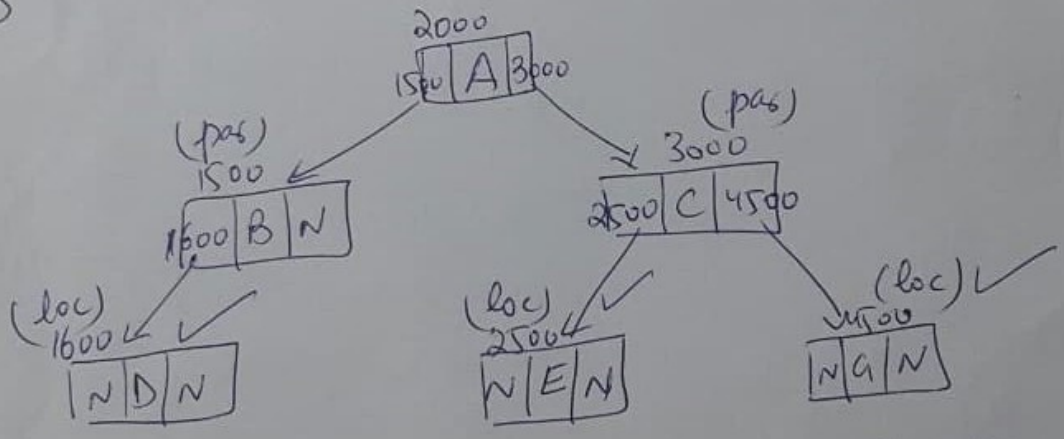
1. No nodes in the tree (Tree empty)
2. Node to be deleted is not present in the tree.
3. Node to be deleted is leaf node/ terminal node.
4. Node to be deleted has only one child.
5. Node to be deleted has two children.

As 1 and 2 are simple and clear, so we start from 3.

3. Node to be deleted is leaf node/ terminal node.

```
void case a(node par, node loc)
{
   if( par == null) root = null;
   else if( loc == par.lchild)
                par.lchild = null;
   else par.rchild = null;
}
```

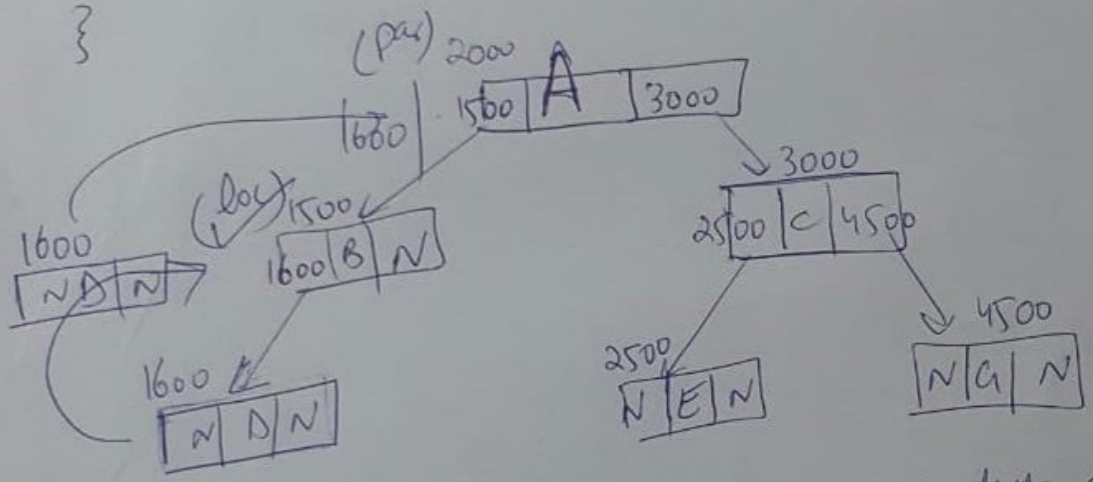
Remember that find() function will be called to get parent and location values.



D, E and G are leaf nodes. So location can be 1600, 2500 or 4500. And parents are 1500(B), 3000(C). So, what we will do simply is to make the left or right address of parent node = null, so that the terminal node could be cut off and afterwards can be deleted.

4. Node to be deleted has only one child.

```
void caseb(node par, node loc)
{
    node child;
    if( loc.lchild != null) child= loc.lchild;
      else   child= loc.rchild;
    (This will find out the child of node to be
    deleted so that we could attach its child to its
    parent in order to delete that node)
      if (par == null) root = child;
    else if (loc == par.lchild) par.lchild = child;
       else par.rchild = child;
}
```



B is the only node which has one left child.
So what we will do is to make left child of B (i.e, D) as the left child of node B parent (i.e, A)
This is how node B will be assumed to be deleted because now node A will point to left child of B (i.e, D).

5. Node to be deleted has two children

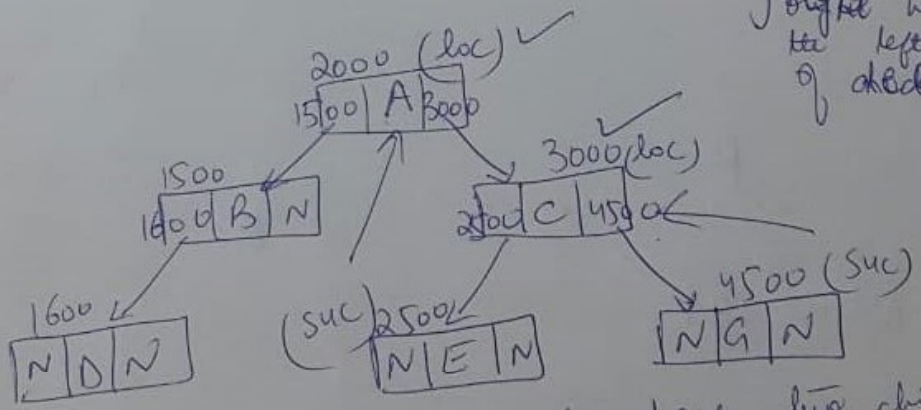
```
void casec(node par, node loc)
{
    node ptr, ptrsave, suc, parsuc;
    ptrsave = loc;
    ptr = loc.rchild;
    while(ptr.lchild != null)
    {
        ptrsave = ptr;
        ptr = ptr.lchild;
    }
    suc = ptr;  (Inorder successor).
    parsuc = ptrsave;
    if(suc.lchild==null && suc.rchild==null) casea(parsuc, suc);
        else caseb(parsuc, suc);
    if(par == null) root = suc;  (only one node).
    else if(loc == par.lchild) par.lchild = suc;
        else par.rchild = suc;
    suc.lchild = loc.lchild;
    suc.rchild = loc.rchild;
}
```

Such a node will be replaced with its inorder successor.

This code will find the inorder successor of such node.
inorder(left, root, child);

Successor node will take the place of node to be deleted.

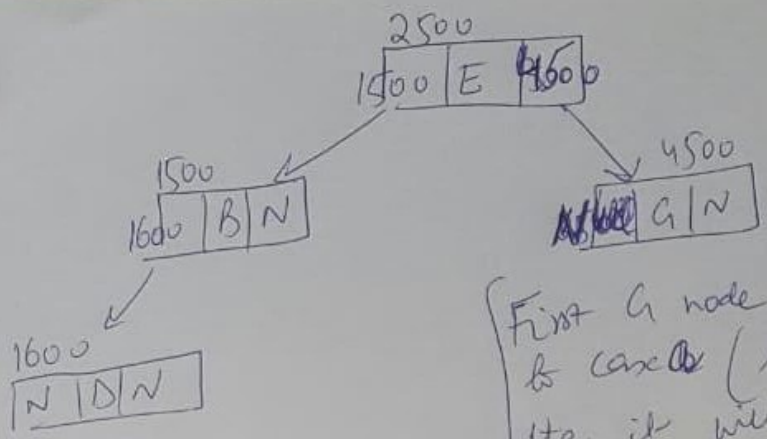
Successor node left and right will be updated with the left and right address of deleted node.



A and C are such node who have two children.

Inorder traversal (D, B, A, E, C, G) → Inorder successor of C

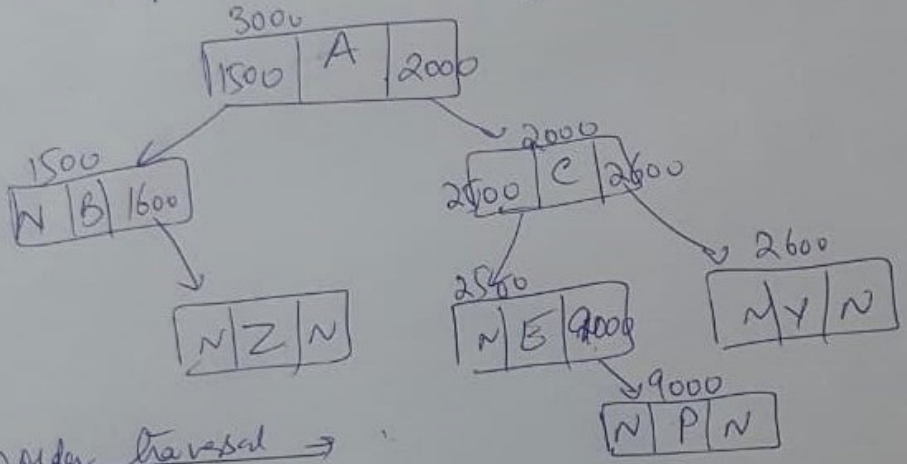
↓ Inorder successor of A.



[First G node will be deleted according to case a (terminal node) and then it will come to the place of node C.

[Node E will be deleted according to case a (terminal node) and then it will come to the place of node A].

Both successor nodes (G and E) were terminal nodes.
Let's see if the successor node belongs to case b (i.e, either has left child or rchild).

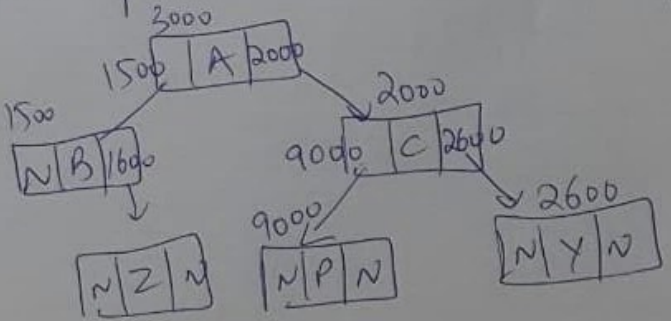


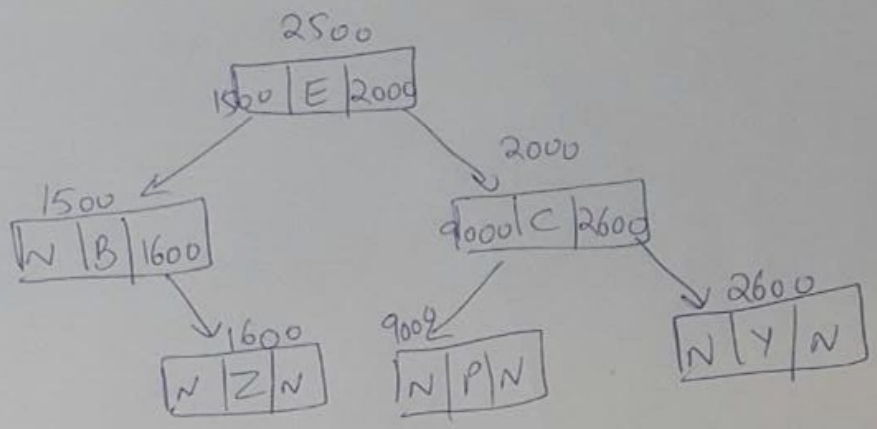
Inorder traversal →

B, Z, A, E, P, C, N

↓ Inorder successor of A but it has one child P.

So, first case b will be applied to this node (E) which means its right child which is P will come to its position like as follows:





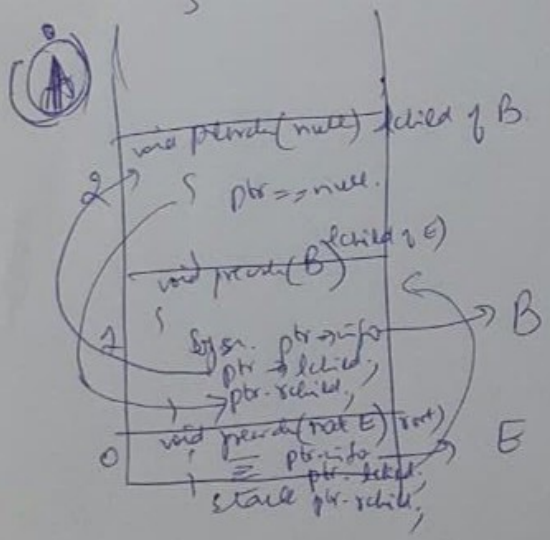
BST traversals

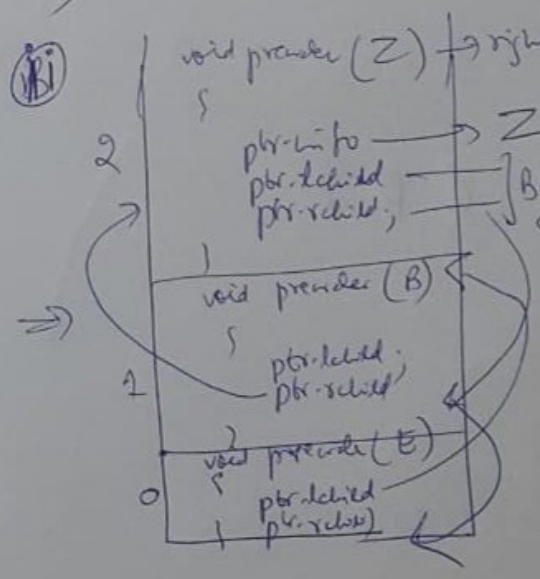
Recursive codes (stack will be used)

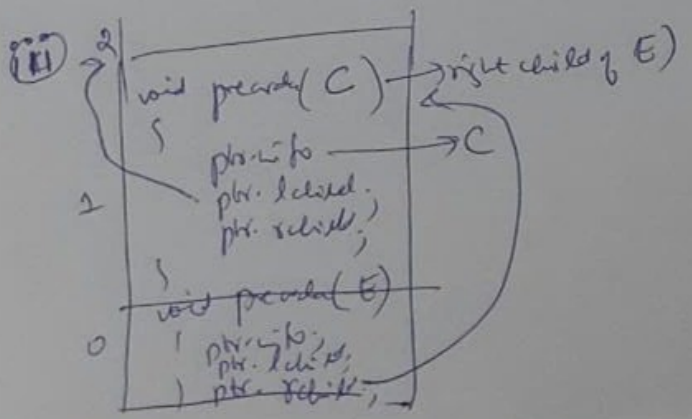
```
void preorder (node ptr)
{
  if (ptr != null)
  {
    System.out.print (ptr.info);
    preorder( ptr.lchild);
    preorde( ptr.rchild);
  }
}
```

Preorder traversal of above tree.

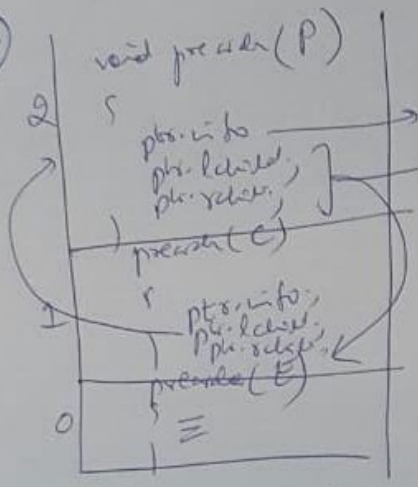
E, B, Z, C, P, Y





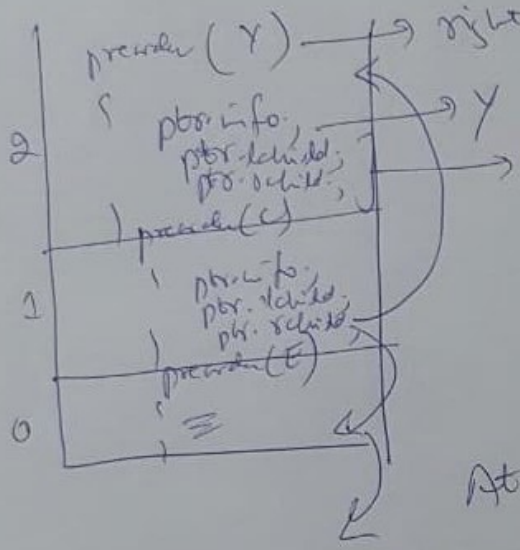


(iv)



→ Both are null, so these calls will come back from position 3 of stack, one after the other.

(v)



→ Both are null, so the calls will come back from position 3 of stack, one after the other.

At the end stack will be empty.

Same stacks can be drawn for inorder and postorder traversals.

Non-Recursive codes of BST traversals

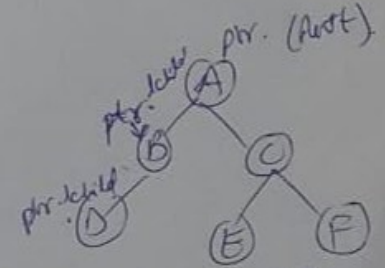
Preorder (Root, left, Right).

```
nrecpre( node ptr)
{
    stack[++top] = ptr;
    while(top != -1)
    {
        ptr = stack[top--];
        if (ptr != null)
        {
            output ptr.info;
            stack[++top] = ptr->rchild;
            stack[++top] = ptr->lchild;
        }
    }
}
```

because left child be take out first because it is stack

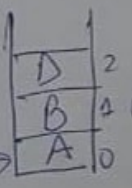
Inorder (Left, Root, Right)

```
nrecin(node ptr)
{
    while (top != -1 || ptr != null)
    {
        if (ptr != null)
        {
            stack [++top] = ptr;
            ptr = ptr->lchild;
        }
        else
        {
            ptr = stack[top--];
            output ptr.info;
            ptr = ptr->rchild;
        }
    }
}
```



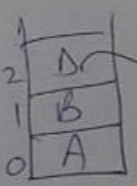
Inorder traversal (Left, Root, Right)
D, B, A, E, C, F.

(i)



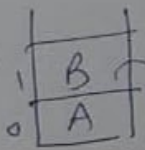
Now ptr = null b/c left child of D is null, so else statement will be executed

(ii)

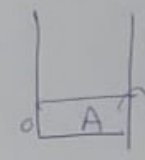


ptr = stack[top--];
D (ptr.info),
ptr = null
(right child of D is null).

(iii)



D, B ptr = null b/c right child of B is null.

(iv) 

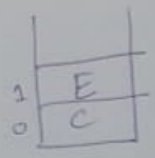
D, B, A

ptr = C b/c right child of ptr = C

As it is not null, so if statement will be executed.

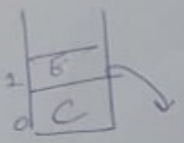
(v) 

ptr = E b/c left child of C is E.

(vi) 

As ptr = E is not null, so if statement will be executed

ptr = null b/c left child of E is null.

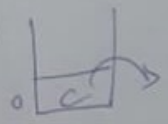
(vii) 

Else statement will be executed.

ptr = E and output it.

D, B, A, E

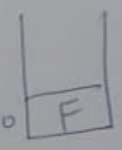
ptr = null b/c E right child is null.

(viii) 

Else statement will be executed.

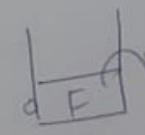
D, B, A, E, C

ptr = ~~Null~~ b/c right child of C is F.

(ix) 

If statement will be executed b/c ptr = F is not null.

ptr = null b/c ~~left~~ child of F is null.

(x) 

D, B, A, E, C, F

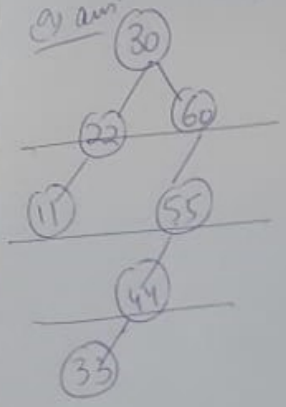
~~As stack gets empty~~ top = -1, so while loop will terminate b/c

Similarly write the non recursive code of Postorder traversal

Question a) Create BST from the following: (13)

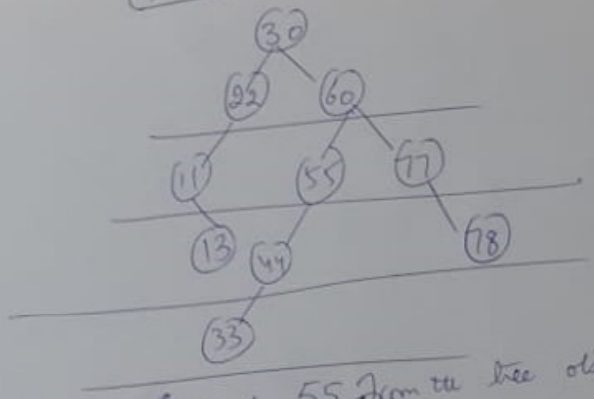
30, 60, 22, 11, 55, 44, 33 (n=7)

a) ans



b) Insert nodes 77, 78 and 13 in the above tree.

b) ans.



c) Delete nodes 30, 60 and 55 from the tree obtained in (b).

c) ans.

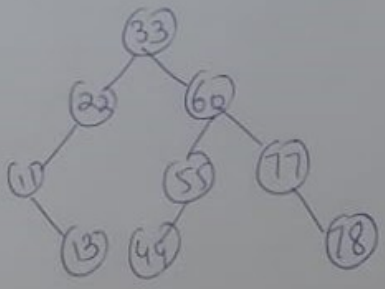
(i) Node 30 has two children, so it will be replaced/deleted with its inorder successor.

In order traversal of above tree

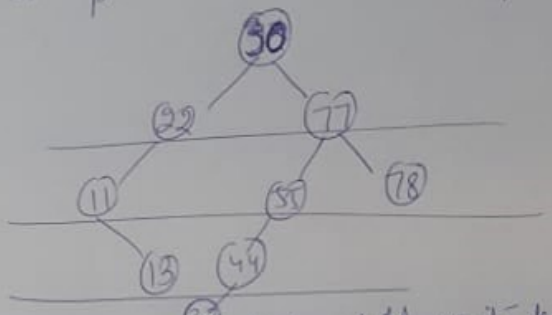
11, 13, 22, 30, 33, 44, 55, 60, 77, 78

↓ Inorder successor and it is a terminal node.

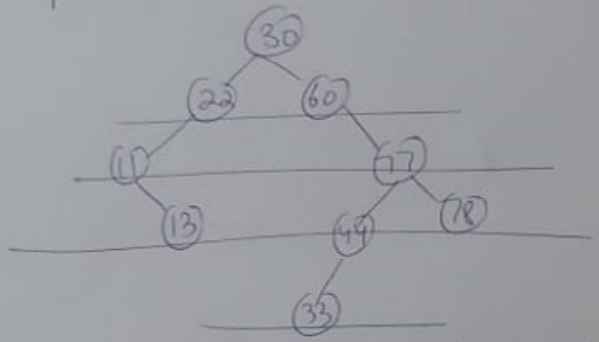
So, simply we will take it at the place of node 30.



(ii) Node 60 has also two children and its successor is 77 which is one child node, so first its child will come at its place and then it will go to the node 60.



(iii) Node 55 has only left child, so its left child will come at its place to delete it.



Creation of BT from Preorder and Inorder traversals

Preorder A B D H E C F G
Inorder D H B E A F C G

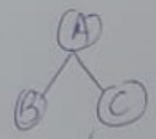
First node of preorder traversal is the root of the tree and in inorder, we can find the left and right subtrees of A.

Pre A B D H E C F G
In D H B E A F C G

Cut the node wherever you insert in the tree to avoid confusion.

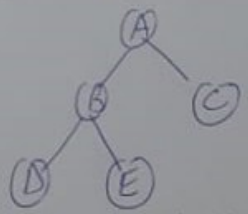
Pre → A B D H E C F G
In → D H B E A F C G

Now on the left and right subtrees of A, we will see that which node come first from the left side of preorder. It shows that out of the nodes of left subtree of A (D, H, B, E in inorder) node B comes first in preorder so, we will take it first and similarly on the right side (F, C, G) node C comes first from the left side of preorders. So, we will take C first as follows:



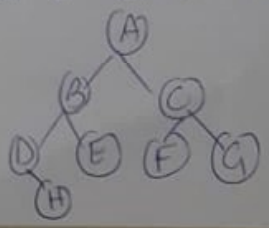
Pre A B D H E C F G
In D H B E A F C G

Now on the left side of E in inorder nodes D, H lie but out of them node D comes first in preorder, so we will write D first and on the right side of B, there is only one node E, so this will be written as it is:



Now there is only one node on the right side of D ie, H. So it will be written like it is. For node C, only one node ie, F is on the left side and only one node on the right side ie, G. So they will be written like they are:

Do the preorder and inorder traversals again for verification.

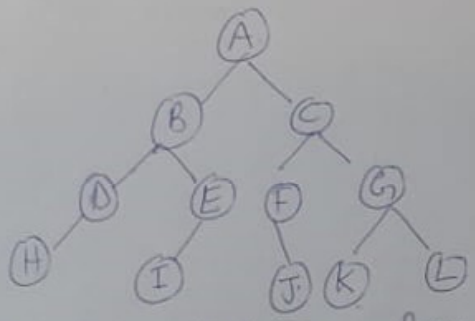


Pre A B D H E C F G
In D H A B E A F C G

Creation of BT from Postorder and Inorder traversals

Postorder H D I E B J F K L G C A ←
Inorder H D B I E A F J C K G L

Whole procedure is same, the only difference is that we start from the right side of postorder as opposite to preorder in which we take and see from the left side of preorder.



Do the postorder and inorder again for verification.

Points to remember

① left and right subtrees indication from inorder traversal.
② Which will be write first out of a number of nodes, the one which comes first in preorder (from the left side) and in postorder (from the right side) traversal.
③ Root is first node from the left side in pre order and postorder traversal. First node from the right side in
④ Cut the nodes whichever you include in the tree to avoid confusion.
⑤ Perform the traversals again to avoid confusion.