Unit-IV

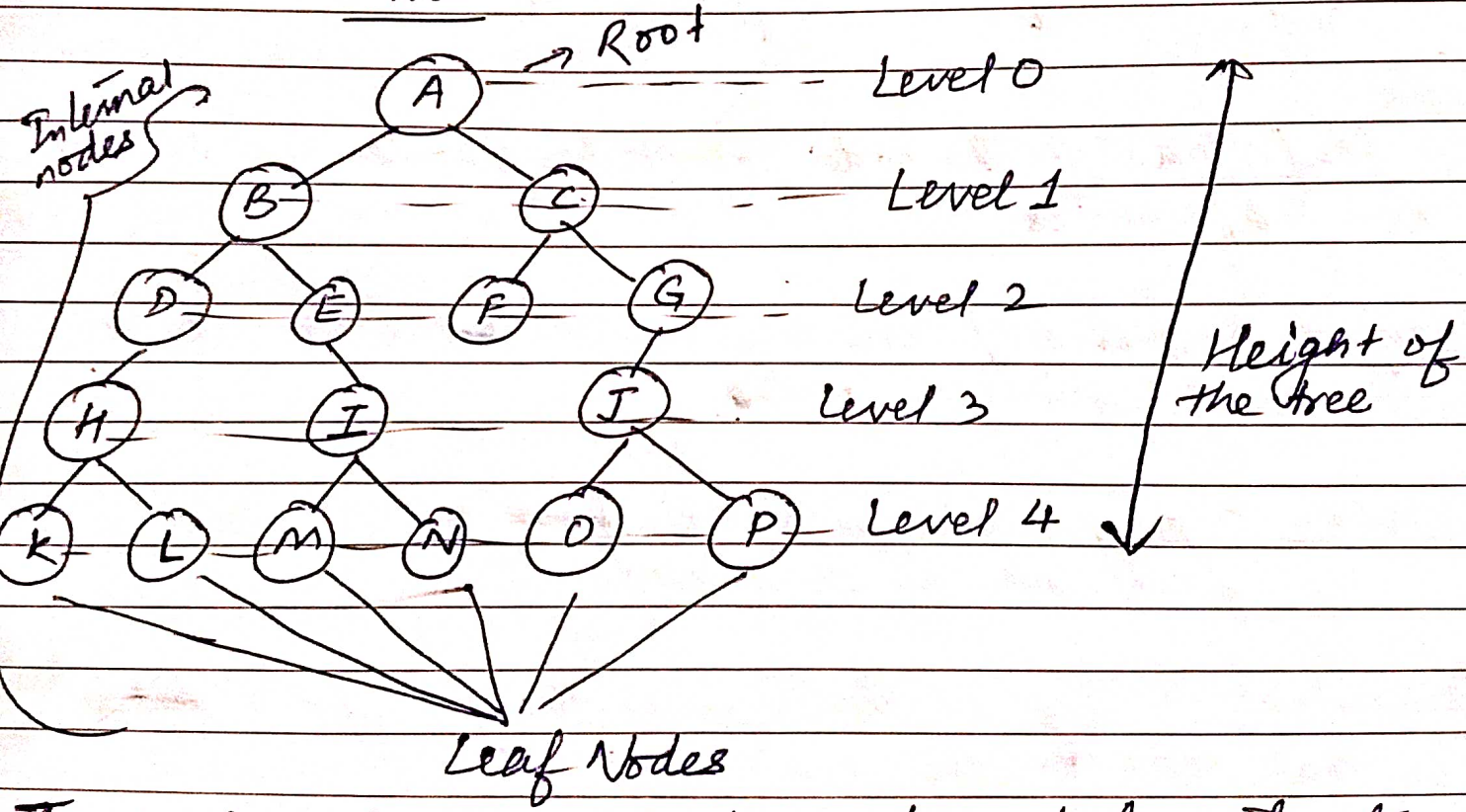
Date .....

Trees

A tree data structure is a hierarchical structure that is used to represent and organise data in a way that is easy to navigate and search. It is a collection of nodes that are connected by edges and has a hierarchical relationship between the nodes.

The topmost node of the tree is called the root and the nodes below it are called child nodes. Each node can have multiple child nodes, and these child nodes can have their own child nodes forming a recursive structure.

Tree



Traversal in tree can only be performed from top to bottom.

Date .....

Basic Terminologies Related to Tree

① Root → The top most node of the tree is called Root node. There is exactly one root in tree.

② Parent → The immediate predecessor of any node is called parent node.

③ Child → The immediate successor of any node is called child of that node.

④ Leaf node → The node which is having no child is called leaf node. They are also called as external nodes.

⑤ Non Leaf Node → The nodes except leaf nodes are called non leaf nodes.

⑥ Path → Sequence of consecutive edges from source to destination node.

⑦ Ancestor → Any predecessor node on the path from root to that node.

⑧ Descendant → Any successor node on the path from that node to the leaf node.

⑨ Subtree :- A part of tree containing all its descendants.

⑩ Sibling :- Children of same parent are called siblings.

⑪ Degree → Degree of a node is the no of child nodes

Speral

Date .....

attached to it.

Degree of leaf node is always 0.

Degree of tree is the maximum degree of any node.

(12) Depth of a node :- The length of the path from root to that node. Depth of root node is always 0.

(13) Height of node :- No. of edges in the longest path from that node to leaf node.

* Height and depth of a node may or may not be same.

(14) Height of Tree :- Height of tree is always equal to height of root node.

(15) Depth of Tree :- No. of edges in the longest path from root to leaf node.

Height of tree is equal to depth of tree

(16) Level of a node :- No. of edges in the path from root to that node.

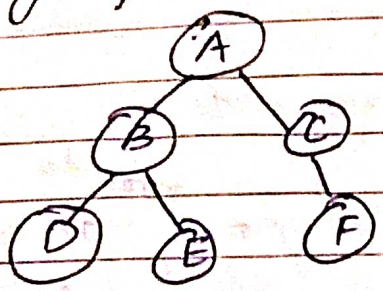
Level of a node = Depth of node

Level of tree is equal to height of tree

If in a tree there are 'n' nodes, then there will be (n-1) edges.

Date .....

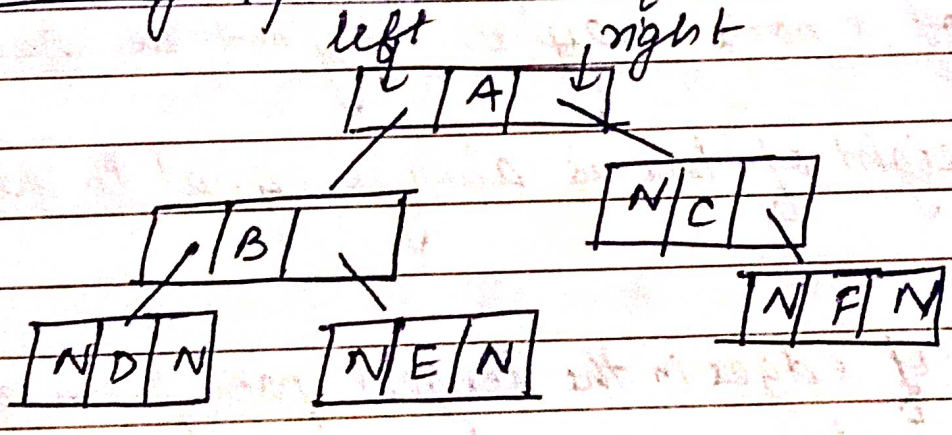
Memory Representation of a tree



→ Binary tree

A tree is said to be a binary tree if each node is having atmost 2 children.

Memory Representation of each node

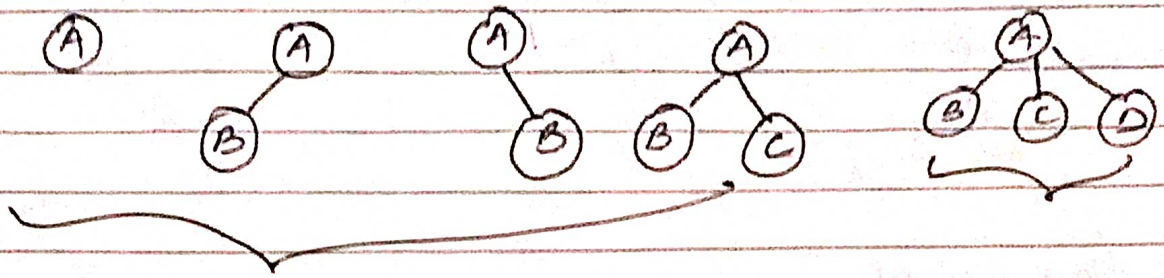


We will use Doubly linked list to represent nodes in memory.

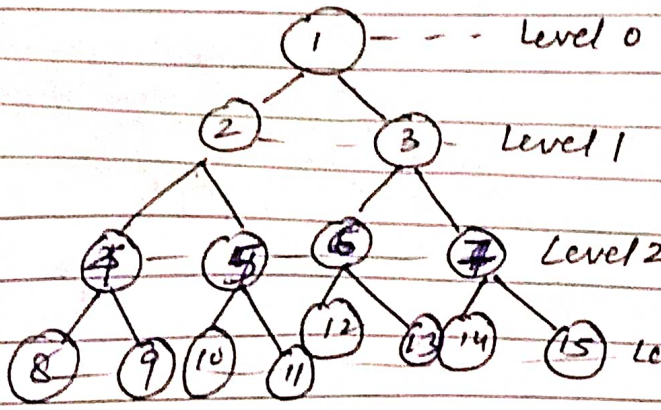
Date .....

Binary Tree & its types

A binary tree is a tree in which each node can have maximum of two children.



All are valid binary trees



Level 0 (Max no of nodes = 1)

Level 1 (Max no of nodes = 2)

Level 2 (Max no of nodes = 4)

Level 3 (Max no of nodes = 8)

Properties of binary tree

(1) Maximum no of nodes at any level is 2^i. where i are level nos.

(2) Maximum no of nodes possible at height $h = 2^0 + 2^1 + 2^2 + \cdots 2^h$

$= \underline{2^{h+1} - 1}$

Min no of nodes possible at height $= 2^{h+1-1} - 1$

Min no of nodes possible at height $h = h+1$

(3) If there are 'n' max no of possible nodes, then the height of tree

Date .....

is: $n = 2^{h+1} - 1$

$n+1 = 2^{h+1}$

Taking log on both side

$\log_2(n+1) = \log_2(2^{h+1})$

$\log_2(n+1) = h+1$

$h = \log_2(n+1) - 1$

↓

min height

Max height

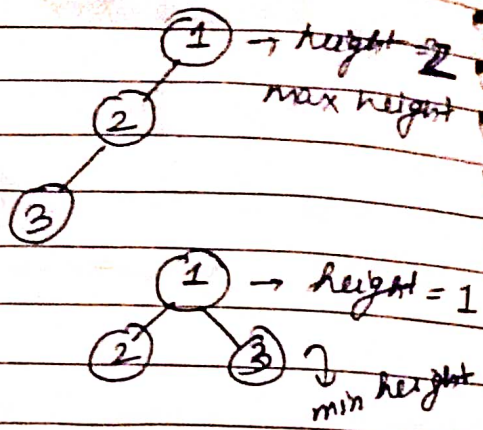
$n = h+1$

$\boxed{n-1 = h}$

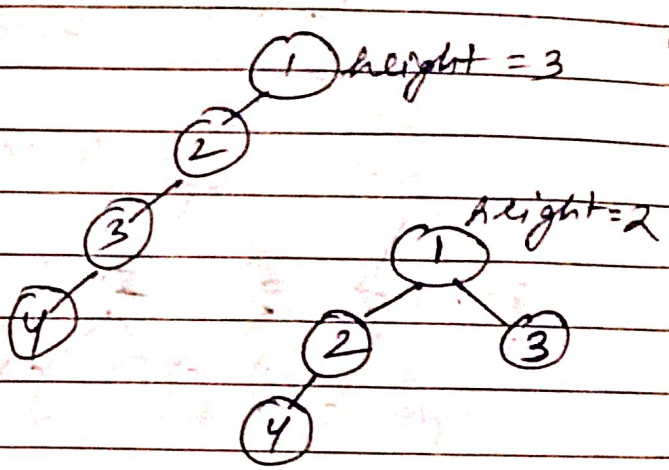
↓

max height

$n = 3$



$n = 4$



Binary Tree types

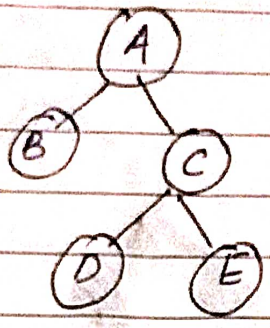
There are four types of binary tree

1. Full / Proper / Strict
2. Complete Binary Tree
3. Perfect Binary Tree
4. Degenerate Binary Tree

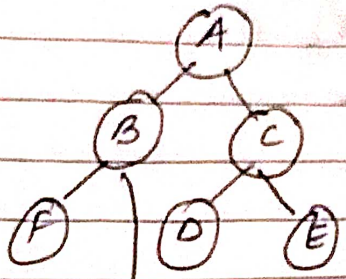
Date .....

Full/Proper/Strict Binary Tree

It is a binary tree where each node contains either 0 or 2 children. Each node contains exactly two children except leaf nodes.



Full Binary tree



Not full binary tree

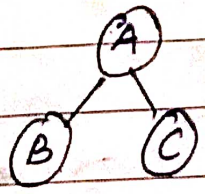
No. of leaf nodes = no. of internal nodes + 1

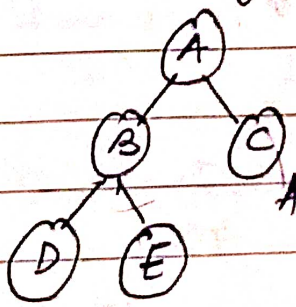
Max no. of nodes at each level = 2^i

Max no. of nodes at height h = $2^{h+1} - 1$

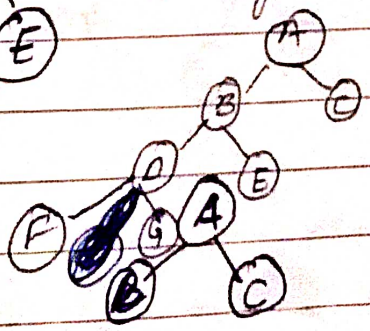
Min. no of nodes = $2h + 1$

At height 0 = 1 At height 1 = 3 At height 2 = 5





At height 3



Min height

$n = 2^{h+1} - 1$

$n+1 = 2^{h+1}$

$\log_2(n+1) = \log_2(2^{h+1})$

$\log_2(n+1) = h+1$

$h = \log_2(n+1) - 1$

Spiral

Date .....

Max Height

$n = 2^h - 1$

$n - 1 = 2^h$

Taking log on both sides,

$\log_2(n-1) = \log_2 2^h$

$\log_2(n-1) = h$

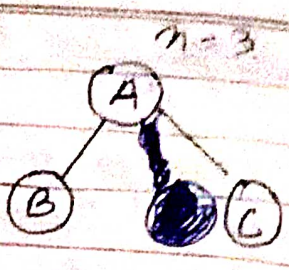
$\frac{n-1}{2} = h$

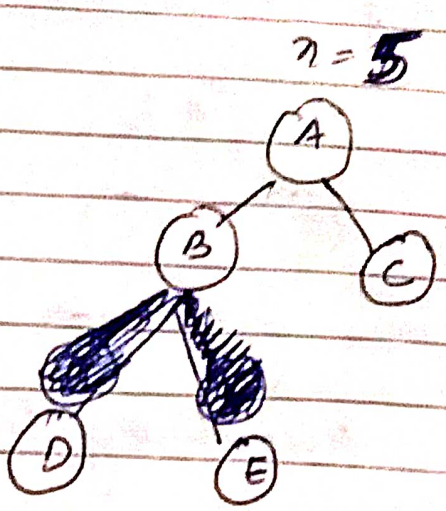
If $n = 3$

$h = \frac{2}{2} = 1$

If $n = 5$

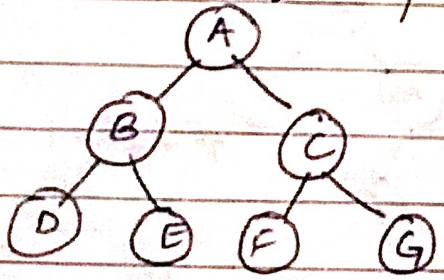
$h = \frac{4}{2} = 2$

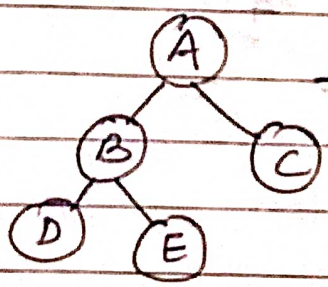


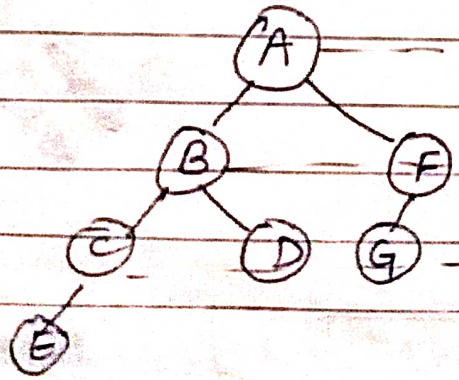


Complete Binary Tree

A binary tree is said to be complete binary tree if all the levels except last level is completely filled. Last level has nodes as left as possible.







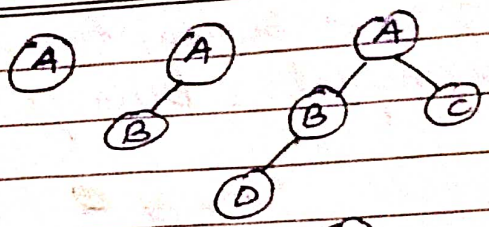
→ Not complete binary tree

Spiral

Date .....

Max no. of nodes = $2^{h+1} - 1$

Min no. of nodes = 2^h



Min height

$n = 2^{h+1} - 1$

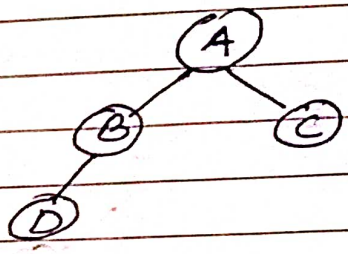
$n + 1 = 2^{h+1}$

$\log_2(n+1) = \log_2(2^{h+1})$

$\log_2(n+1) = h + 1$

$\log_2(n+1) - 1 = h$

$n = 4$



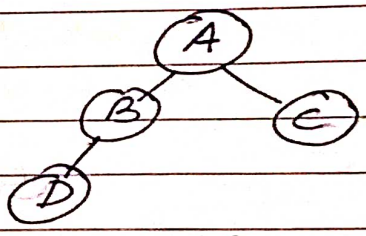
Max height

$n = 2^h$

$\log_2 n = \log_2 2^h$

$h = \log_2 n$

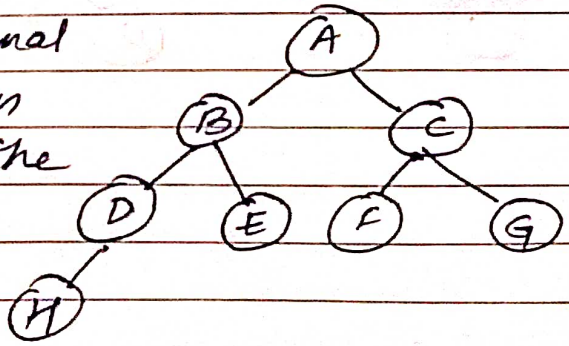
$n = 4$

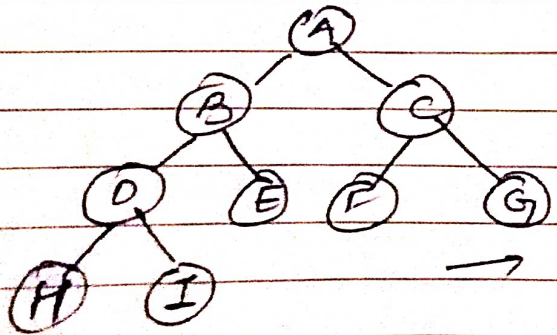


Perfect Binary Tree

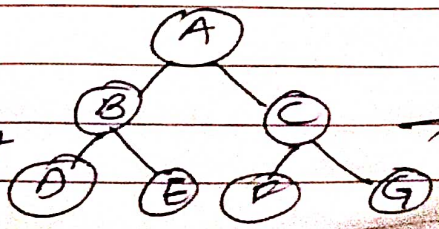
A Tree is said to be perfect binary tree if all the internal nodes are having 2 children and all leaf nodes are at the same level.

$n = 8$





→ Not perfect



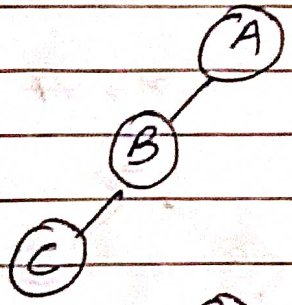
→ Perfect

Date .....

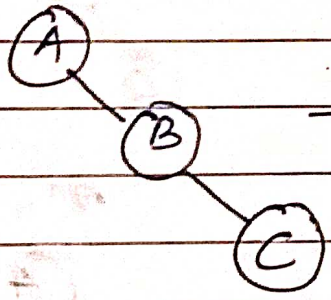
Every perfect binary tree will be full binary tree as well as complete binary tree.

Degenerate Binary Tree

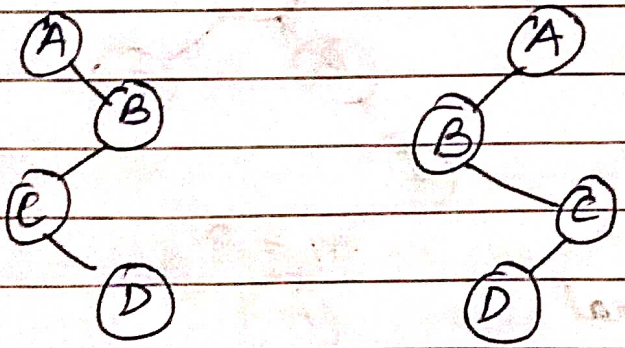
A binary tree is said to be degenerate binary tree if all the internal nodes have only one child.



→ Left skewed binary tree.



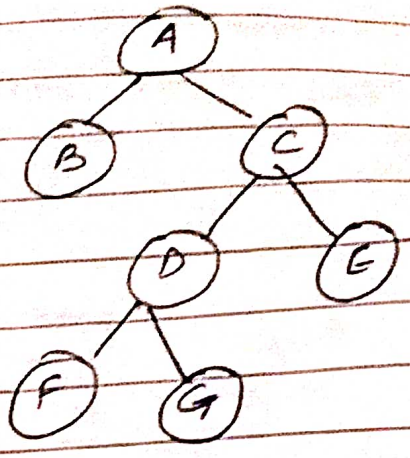
→ Right skewed binary tree

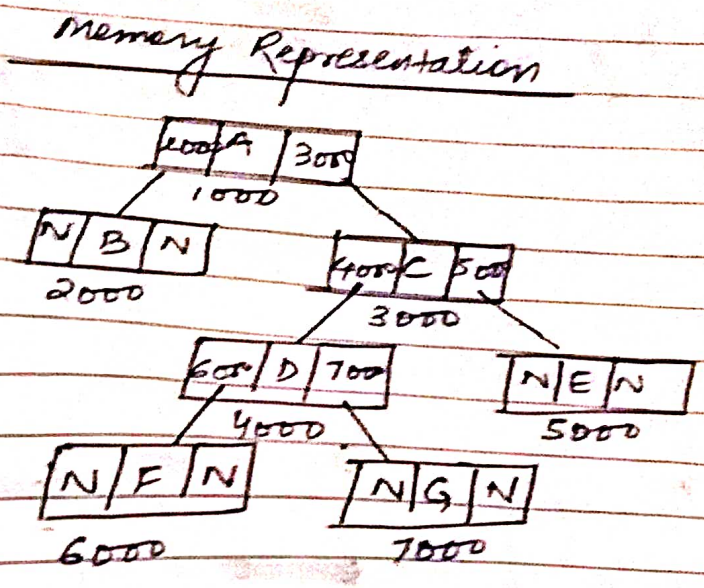


→ Degenerate tree

Date .....

Implementation of Binary Tree

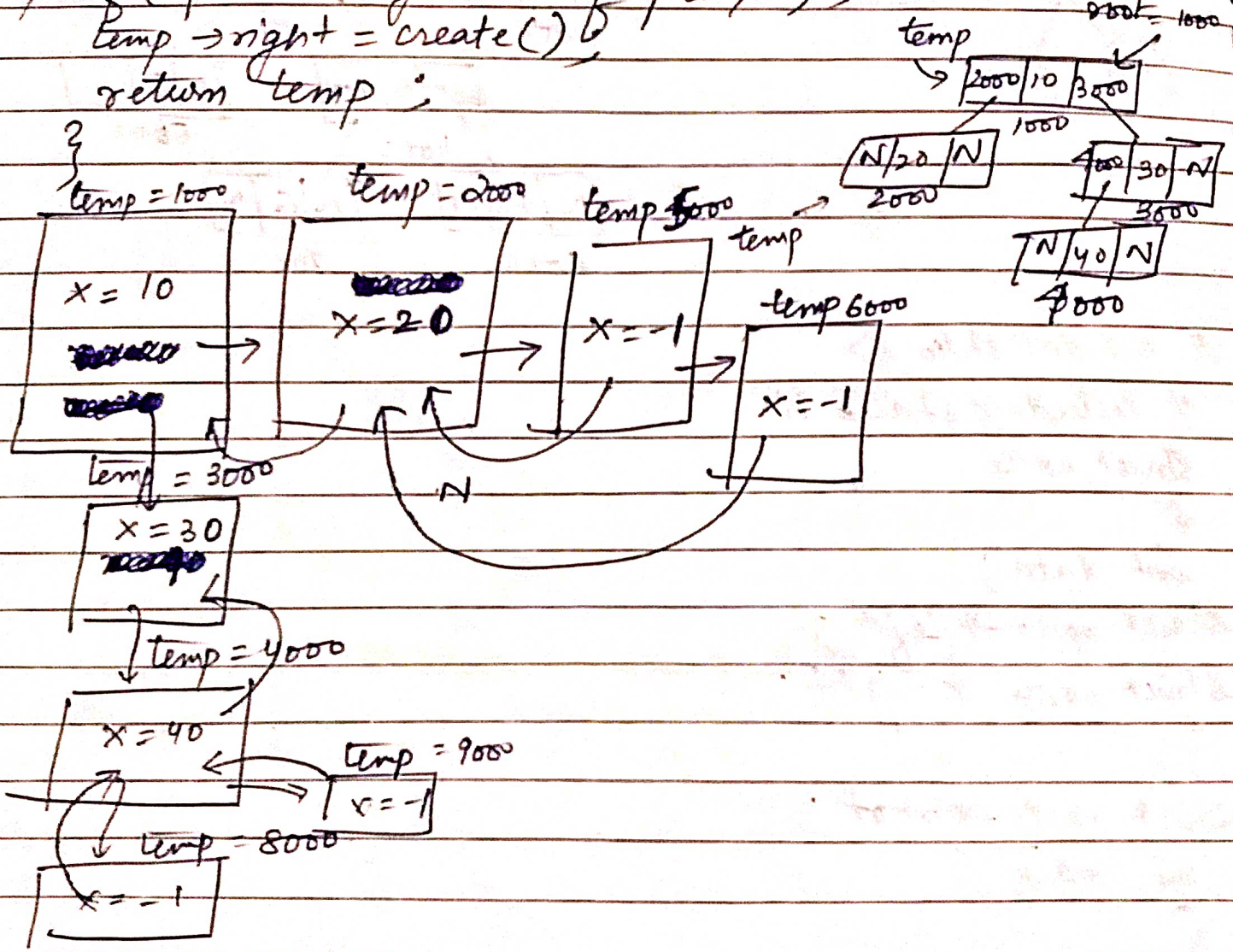





```
# include<stdio.h>
# include<stdlib.h>
Struct node
{
 int data;
Struct node *left;
Struct node *right;
};
Struct node *root;
 int main()
{
  root = NULL;
root = create();
return 0;
}
int create()
{
 Int x;
Struct node *temp;
```

Date .....

```
temp = (struct node *) malloc (sizeof (struct node));
printf ("\n Enter data");
scanf ("%d", &x); if (x == -1) { return 0; }
temp -> data = x; temp -> left = NULL; temp -> right = NULL;
printf ("\n Enter left child of %d", x);
temp -> left = create();
printf ("\n Enter right child of %d", x);
temp -> right = create();
return temp;
}
```

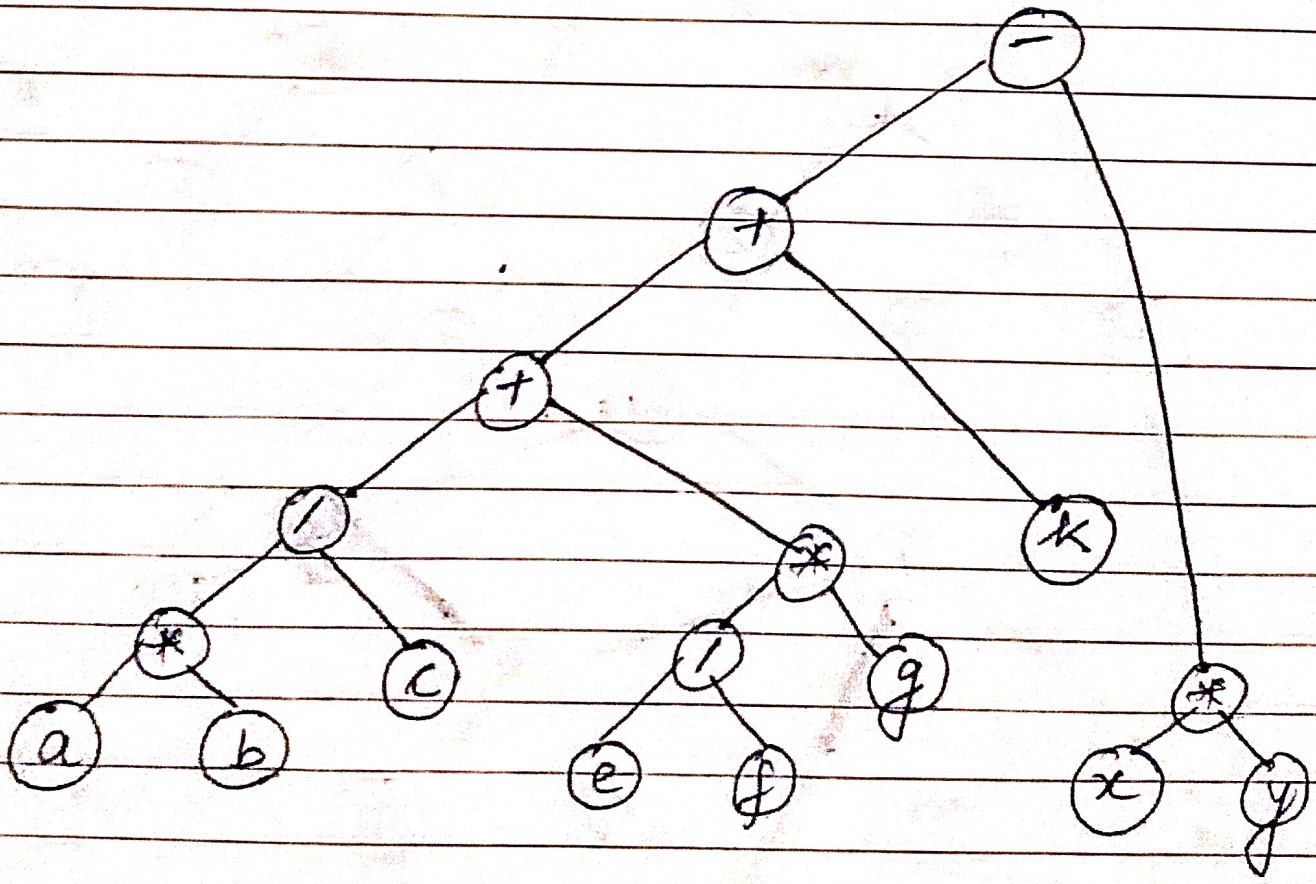


Date .....

Expression Tree

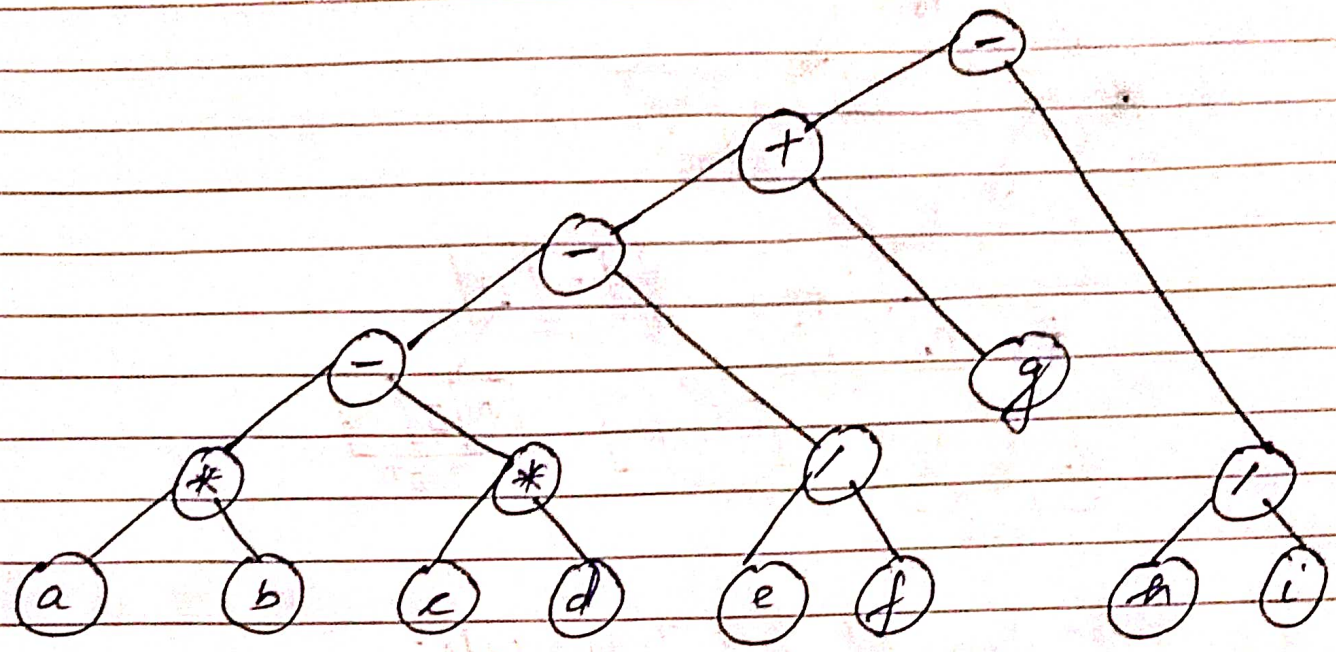
Expression Tree is a special kind of binary tree which are used to represent expressions. Leaves will contain all the operands. Internal nodes contain operators.

Expression :- $a * b / c + e / f * g + k - x * y$

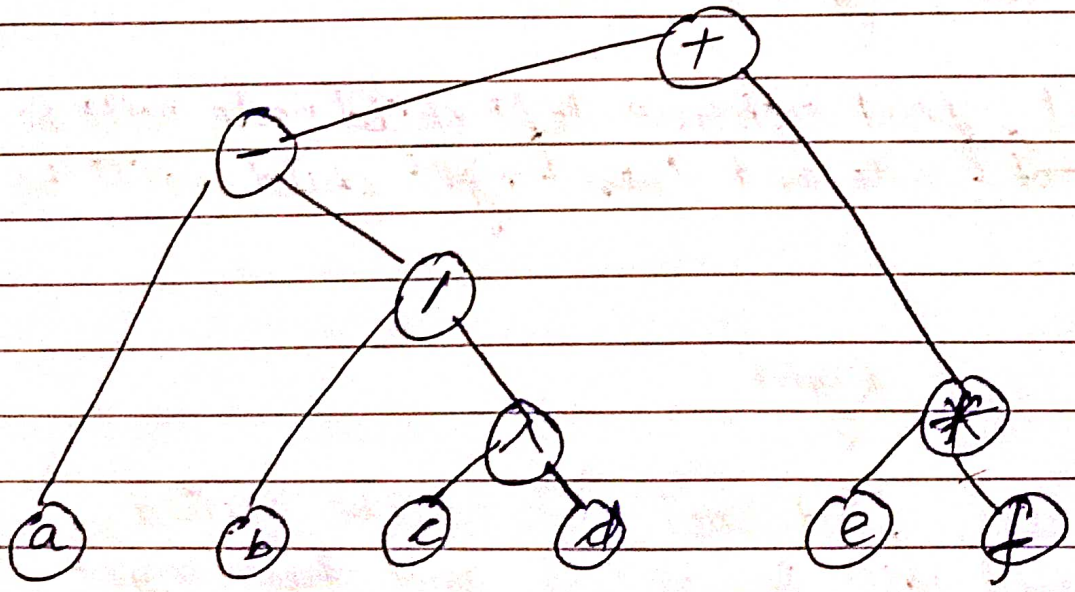


Date .....

a * b - c * d - e / f + g - h / i

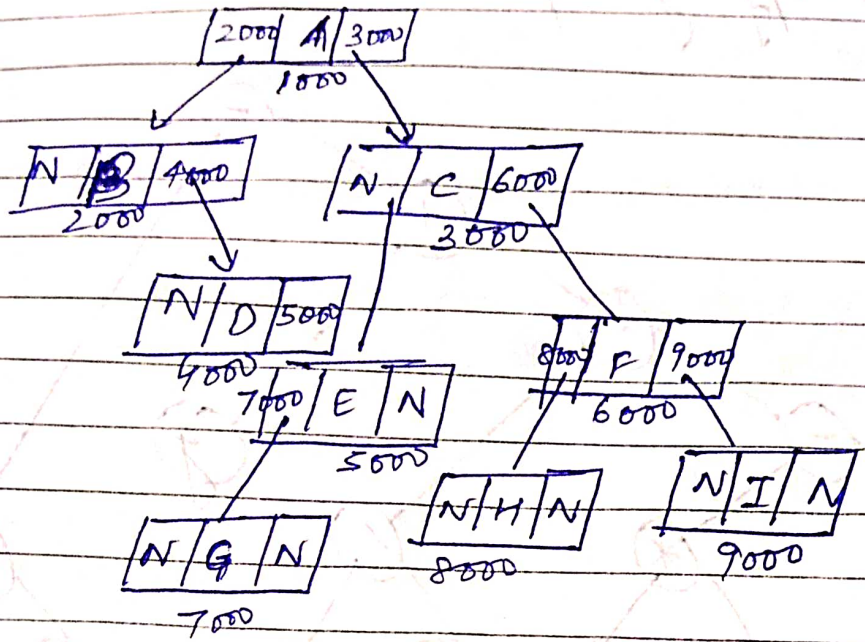


a - b / c ^ d + e * f



Date .....

Binary Tree Traversals



There are three types of Tree Traversals:

1. Inorder Traversal
2. Preorder Traversal
3. Postorder Traversal

Inorder:- Left, Root, Right

In Inorder traversal, first extreme left child node will be visited, then root node and then right child will be visited.

Preorder:- Root, Left, Right

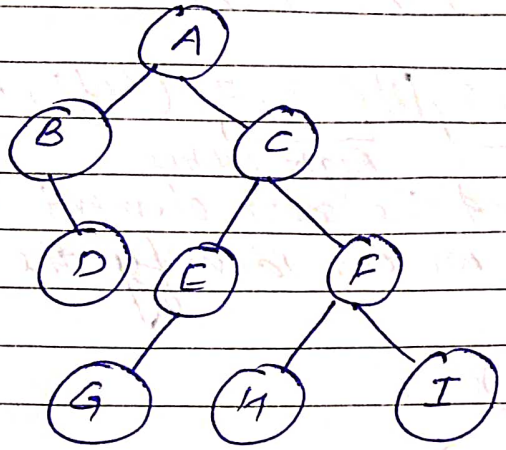
In Preorder Traversal, first root node will be visited, then extreme left child will be visited and then right child will be visited.

Postorder:- Left, Right, Root

Skiral

Date .....

In Postorder Traversal, first extreme left child will be visited, then right child will be visited and then root node will be visited



Inorder of the given binary tree :- B, D, A, G, E, C, H, F, I

Postorder of the given binary tree :- D, B, G, E, H, I, F, C, A

Preorder of the given binary tree :- A, B, D, C, E, G, F, H, I

Date .....

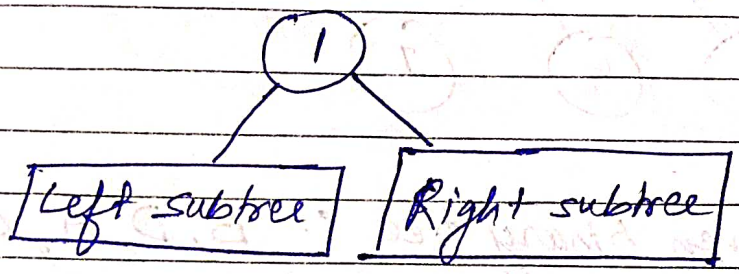
Construct Binary Tree from Preorder and Inorder

Preorder :- 1, 2, 4, 8, 9, 10, 11, 5, 3, 6, 7 (Root, Left, Right)

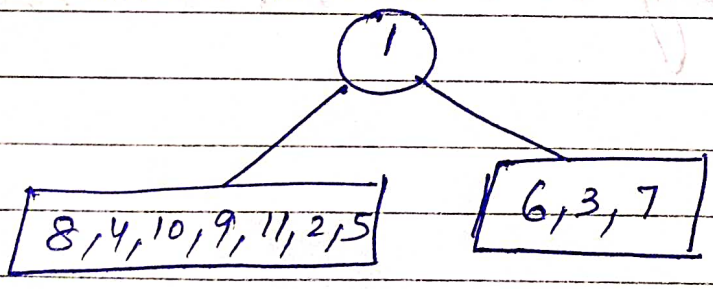
Inorder :- 8, 4, 10, 9, 11, 2, 5, 1, 6, 3, 7 (Left, Root, Right)

Left subtree | Right subtree

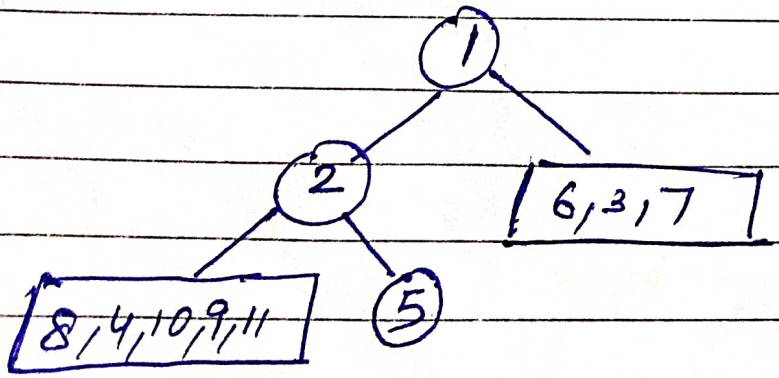
Step 1 :- First of all find the root element.
Using preorder we can identify the root element.



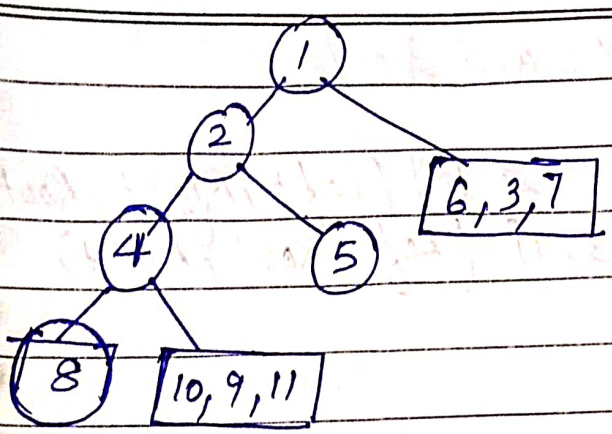
Step 2 :- Next step is to find out the left subtree elements and right subtree elements. Locate the root in inorder traversal.

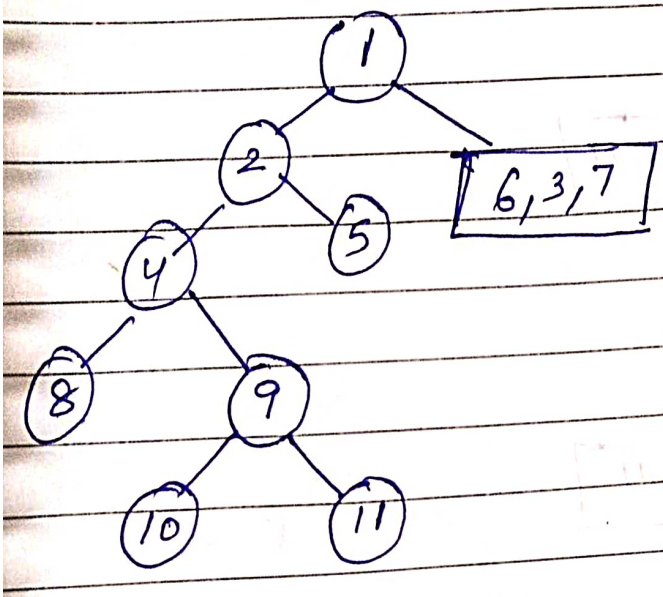


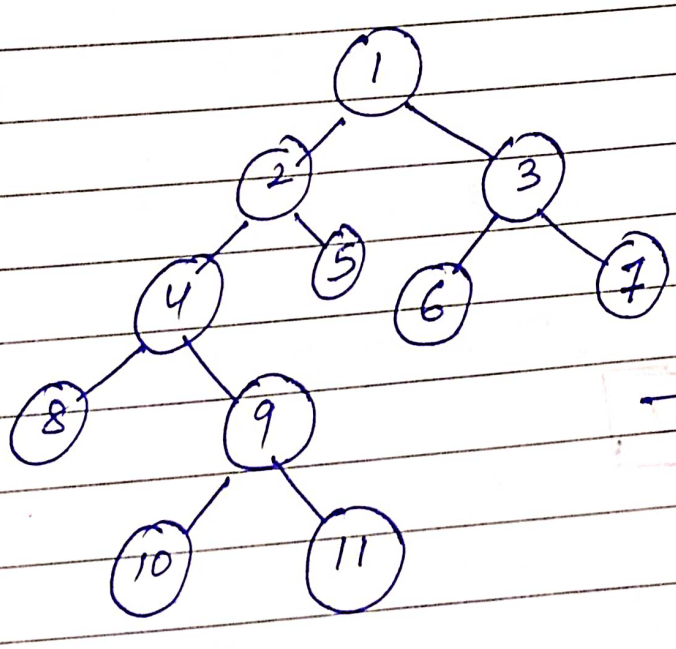
Step 3 :- Now repeat the above two steps on left subtree and right subtree.



Signal







→ Final tree

Preorder :- 1, 2, 4, 8, 9, 10, 11, 3, 6, 7

Inorder :- ~~8, 10, 11, 9, 4, 2, 6, 7, 3, 1~~

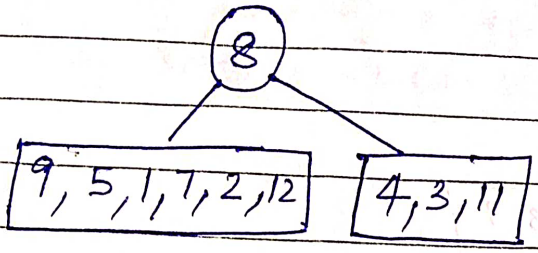
8, 4, 10, 9, 11, 2, 5, 1, 6, 3, 7

Date .....

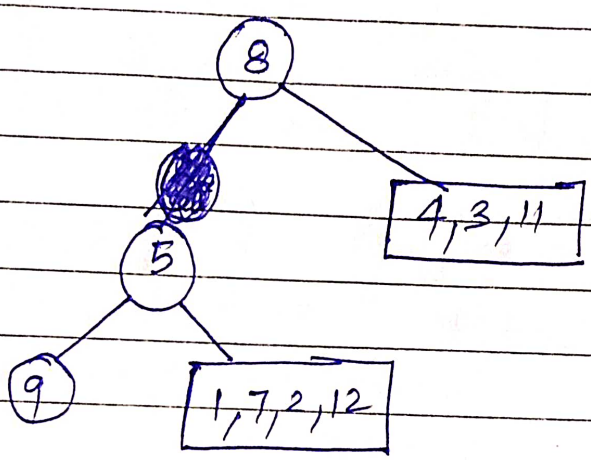
Construct Binary Tree from Postorder and Inorder

Postorder: 9, 1, 2, 12, 7, 5, 3, 11, 4, 8 (Left, Right, Root)
Inorder: 9, 5, 1, 7, 2, 12, 8, 4, 3, 11 (Left, Root, Right)

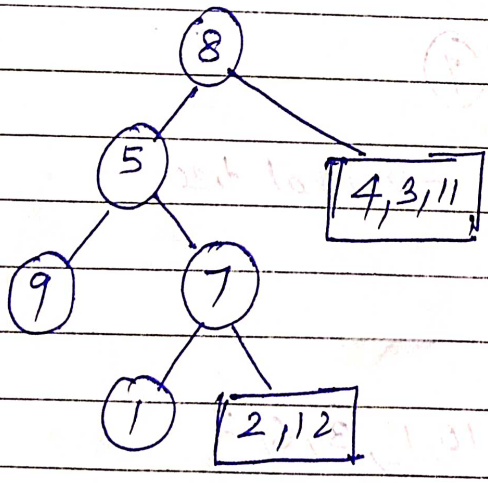
(1) Find root



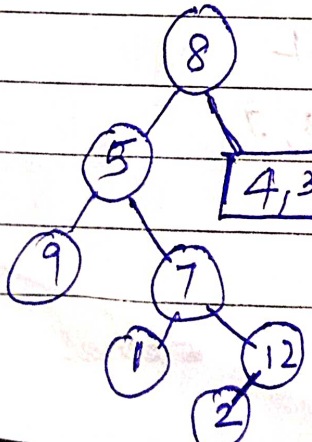
(2)



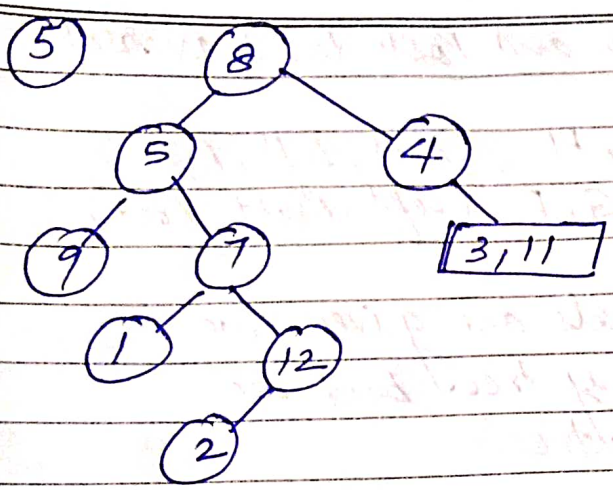
(3)

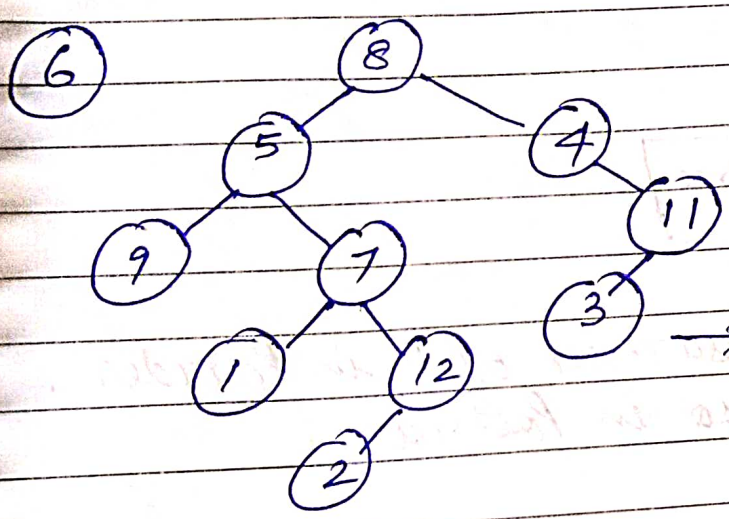


(4)



Spiral





→ Final Tree

Postorder :- 9, 1, 2, 12, 7, 5, 3, 11, 4, 8

Inorder :- 9, 5, 1, 7, 2, 12, 8, 4, 3, 11

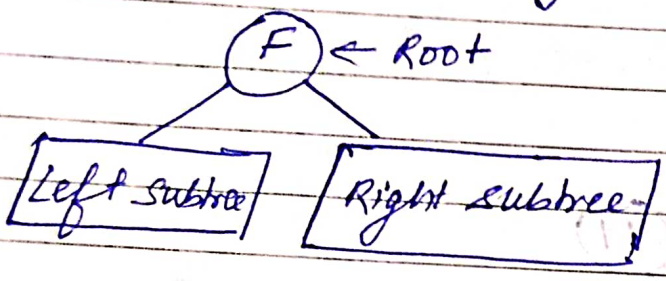
Date .....

Construct Binary Tree from Preorder and Postorder Traversal

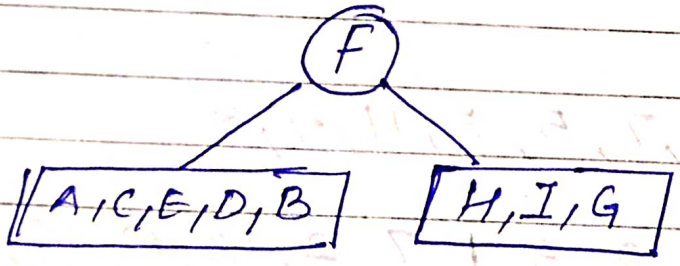
Preorder : F, B, A, D, C, E, G, I, H (Root, Left, Right)
Postorder : A, C, E, D, B, H, I, G, F (Left, Right, Root)

If Preorder and Postorder traversals are given, we cannot construct a unique binary tree but we can construct a ~~unique~~ full binary tree.

(1)



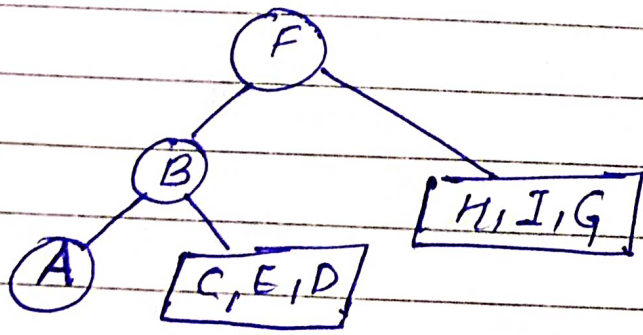
(2) Find out the successor of F in Preorder. and locate the successor in Postorder.



Preorder :- B, A, D, C, E
Postorder :- A, C, E, D, B

Preorder: G, I, H
Postorder: H, I, G

(3)

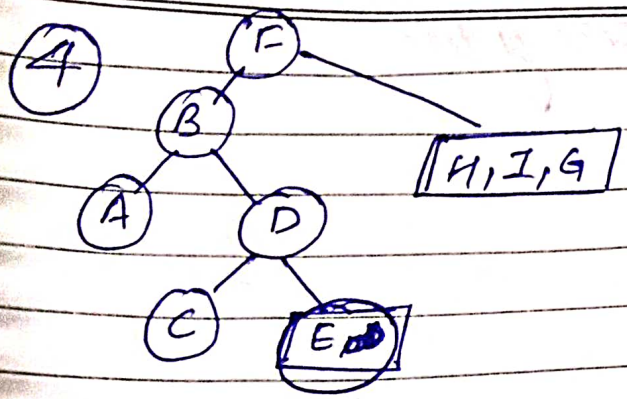


Preorder :- D, C, E
Postorder :- C, E, D

Preorder: G, I, H
Postorder: H, I, G

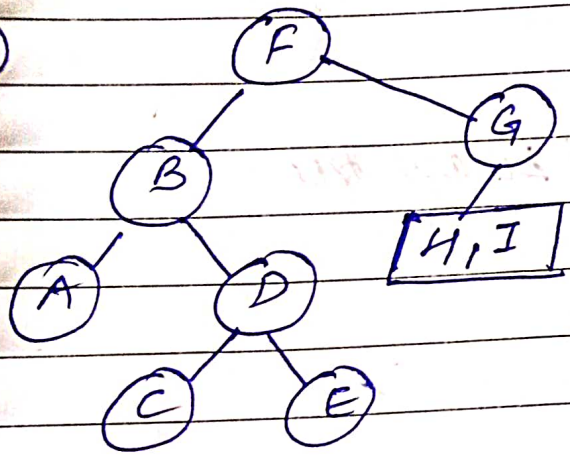
Spiral

4



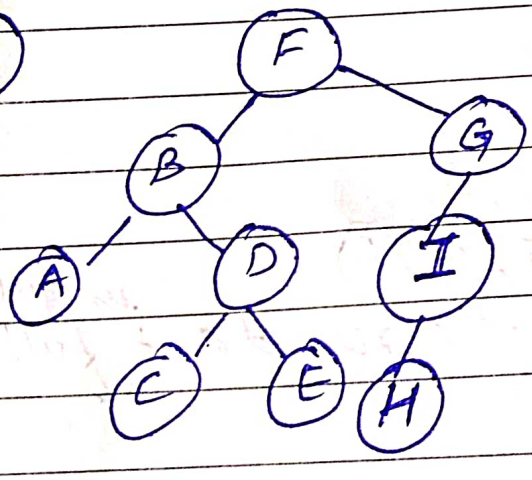
Pre: G, I, H
Post: H, I, G

5



Pre :- I, H
Post: H, I

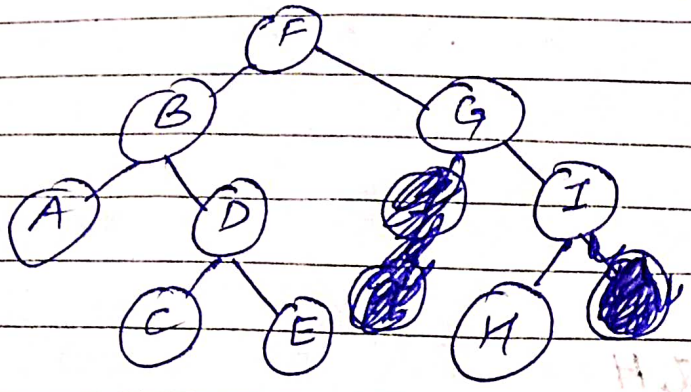
6



Preorder :- F, B, A, D, C, E, G, I, H
Postorder :- A, C, E, D, B, H, I, G, F

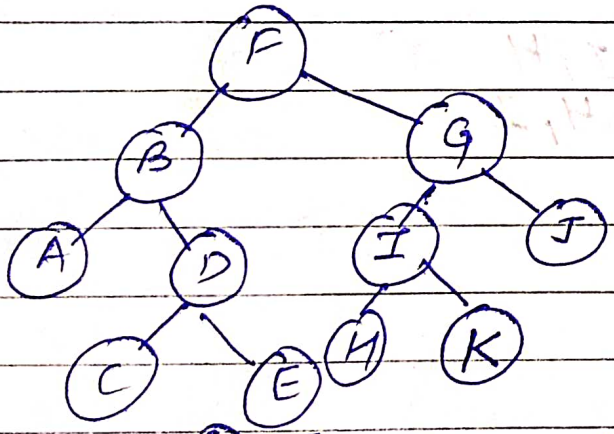
Date .....

Here we cannot construct a unique binary tree.



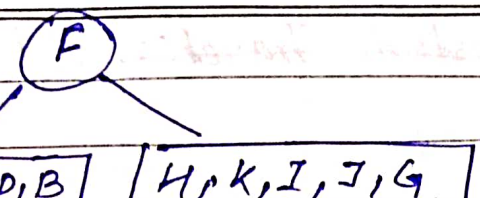
Pre :- F, B, A, D, C, E, G, I, H
Post: A, C, E, D, B, H, I, G, F

But we can construct a unique full binary tree.



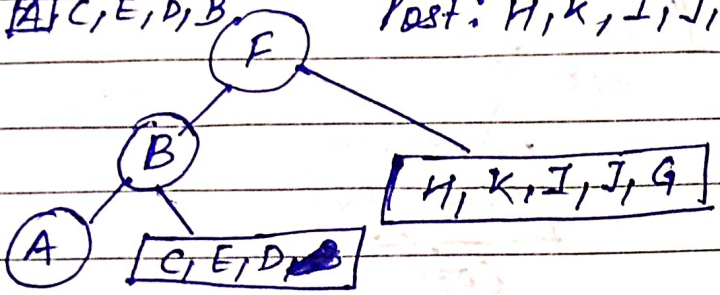
If Preorder is : F, B, A, D, C, E, G, I, H, K, J
Postorder is : A, C, E, D, B, H, K, I, J, G, F
For the given traversal, above will be the resultant unique full binary tree.

Date .....



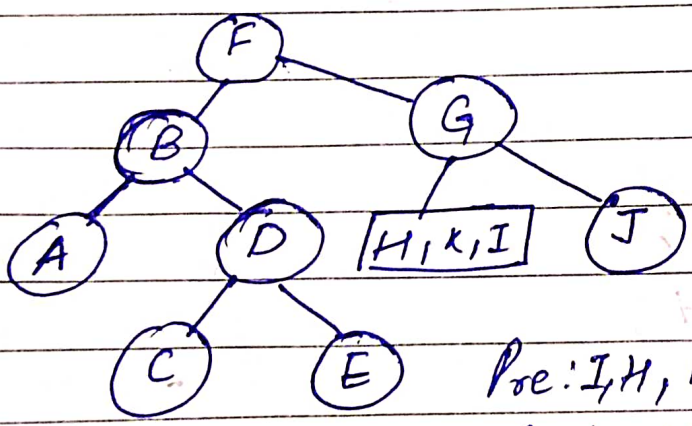
Pre: ~~B~~, A, D, C, E
Post: A, C, E, D, B

Pre: G, I, H, K, J
Post: H, K, I, J, G

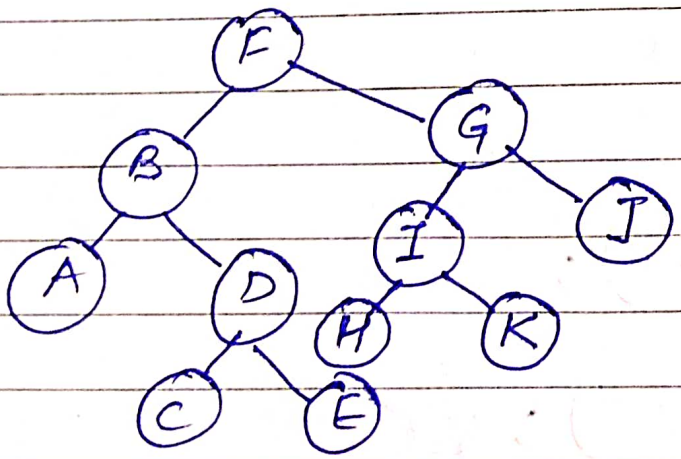


Pre: D, C, E
Post: C, E, D

Pre: ~~H, K, I, J~~ G, I, H, K, J
Post: H, K, I, J, G

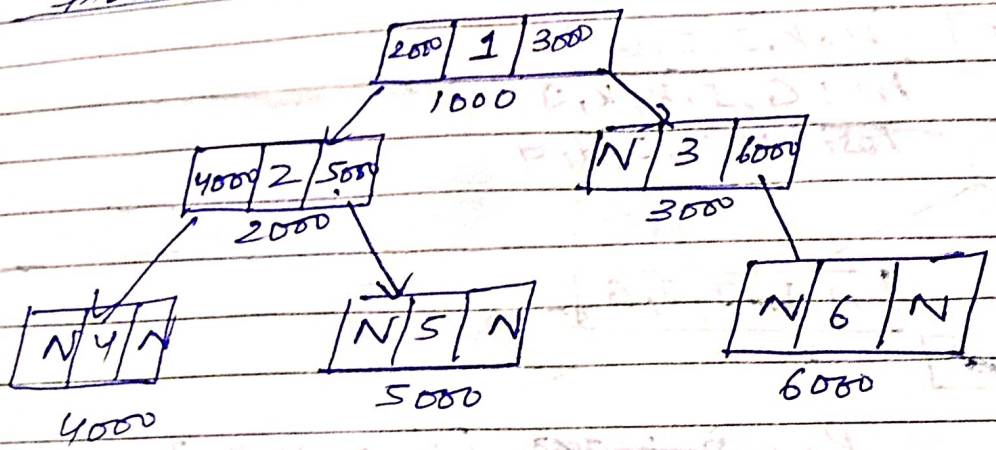


Pre: I, H, K
Post: H, K, I



Date .....

Inorder, Preorder and Postorder Traversal code.




```
# include <stdio.h>
# include <stdlib.h>
 Struct node
 {
  int data;
 Struct node * left;
 struct node * right;
 };
Struct node *root;
 int main()
 {
  root = NULL;
  root = create();
  printf("\n Preorder is : ");
   Preorder(root);
  printf("\n Postorder is : ");
   Postorder(root);
  printf("\n Inorder is : ");
   Inorder(root);
  return 0;
  }
```

Spiral

Date .....

```
int create()
{
 int x;
struct node *temp;
temp = (struct node *) malloc (sizeof (struct node));
printf ("\n Enter node data ");
scanf ("%d", &x);   temp->data = x; temp->left = NULL;
                                    temp->right = NULL;
  if (x == -1)
  {
    return 0;
  }
printf ("\n Enter left child of %d", x);
 temp->left = create ();
printf ("\n Enter right child of %d", x);
temp->right = create ();
 return temp;
}
void preorder (struct node *root)
{
  if (root == 0)
  {
  return;
  }
else
{
printf ("%d", root->data);
 preorder (root->left);
 preorder (root->right);
 }
}
```

Date .....

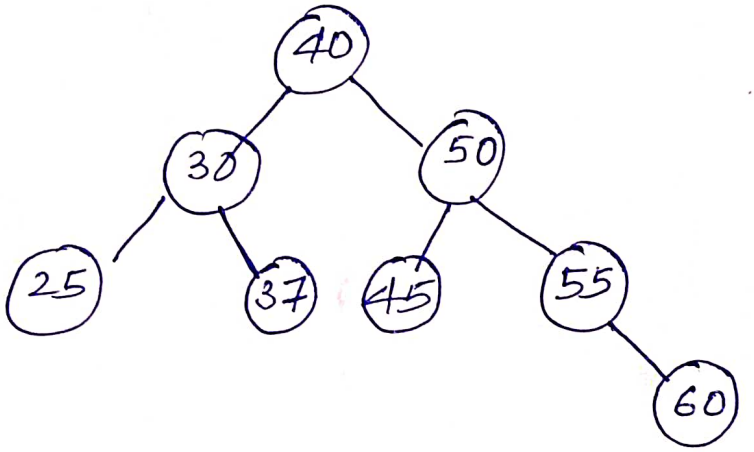
```
void postorder(struct node *root)
{
  if (root == 0)
  {
   return;
  }
  else
  {
   postorder(root->left);
   postorder(root->right);
   printf("%d", root->data);
  }
}

void inorder(struct node *root)
{
  if (root == 0)
  {
   return;
  }
  else
  {
   inorder(root->left);
   printf("%d", root->data);
   inorder(root->right);
  }
}
```

Binary Search Tree

A binary tree is said to be Binary Search Tree if the value of left child is less than or equal to value of root and value of right child is greater than or equal to value of root.

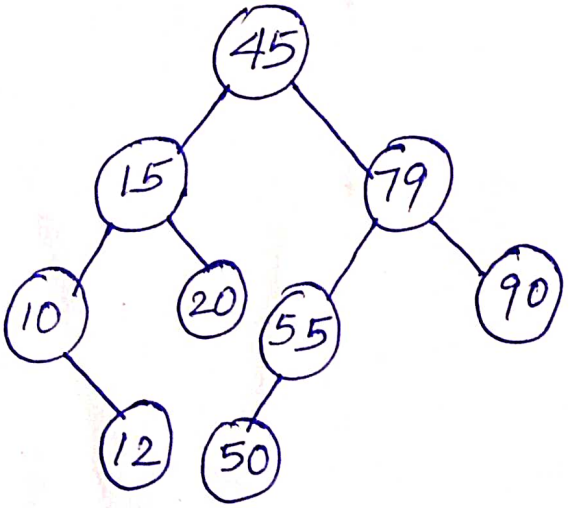
> Left child (value) <= Root (value) <= Right child (value)

Example of Binary search Tree



Construct Binary Search Tree using following elements:

45, 15, 79, 90, 10, 55, 12, 20, 50

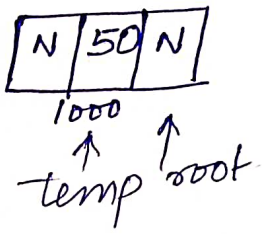


Operations on Binary Search Tree

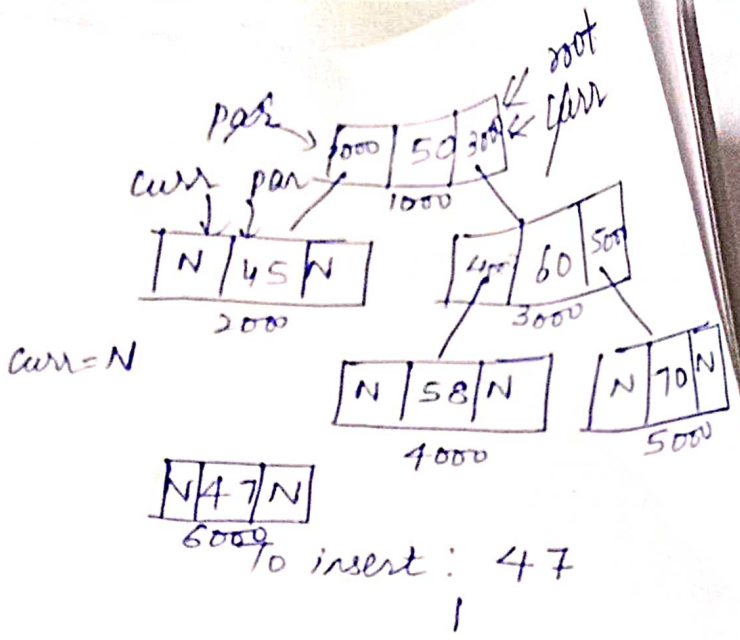
1) Insertion
2) Deletion
3) Display

(1) INSERTION OF ELEMENTS IN BST

```
# include <stdio.h>
# include <stdlib.h>

  struct node
  {
    int data;
  struct node * left;
  struct node * right;
    };
  struct node * root;

  void insert()
  {
   struct node * temp;
  temp = (struct node *) malloc(sizeof(struct node));
   printf("\n Enter node data");
scanf("%d", &temp -> data);
   temp -> left = NULL;
  temp -> right = NULL;
    if (root == NULL)
    {
      root = temp;
    }
```



```
else
{
struct node *curr, *par;
curr = root;
while (curr != NULL)
{  par = curr;
if (temp -> data > curr -> data)
 {
 curr = curr -> right;
 }
 else
{
 curr = curr -> left;
 } }
if (temp -> data > par -> data)
  {
   par -> right = temp;
   }
  else
  {
  par -> left = temp;
   }
}
```



ELETION IN BINARY SEARCH TREE

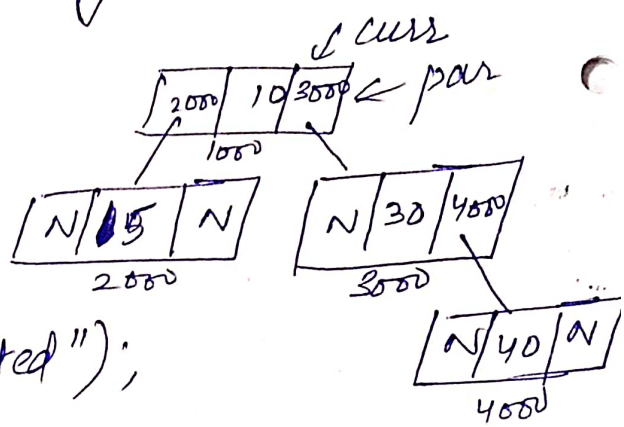
There are 3 cases of Deleting a Node in BST :-

1. Deleting a leaf node having no child
2. Deleting a node having 1 child
3. Deleting a node having 2 children

↓

Replace the deleted value with either maximum value from left subtree or minimum value from right subtree.

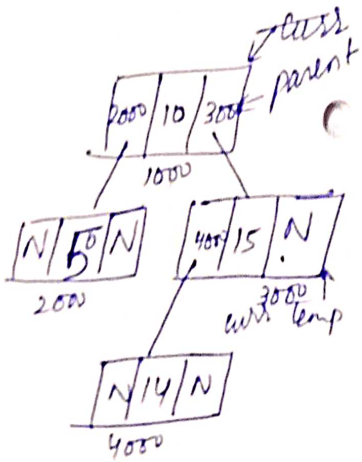
Deleting a leaf node




```
void delete()
{   int item;
struct node *temp;
struct node *curr, *par;
printf("\n Enter data to be deleted");
scanf("%d", &item);
curr = root;
while(curr != NULL)
 {  par = curr;
 if(item < curr->data)
   {
    curr = curr->left;
   }
 else
 {
  curr = curr->right;
 }
} else if(item == curr->data)
  { temp = curr; }
if(curr = par->right)
{  par->right = NULL;
}
```

```
else
{
par -> left = NULL;
}
free (temp);
}
```

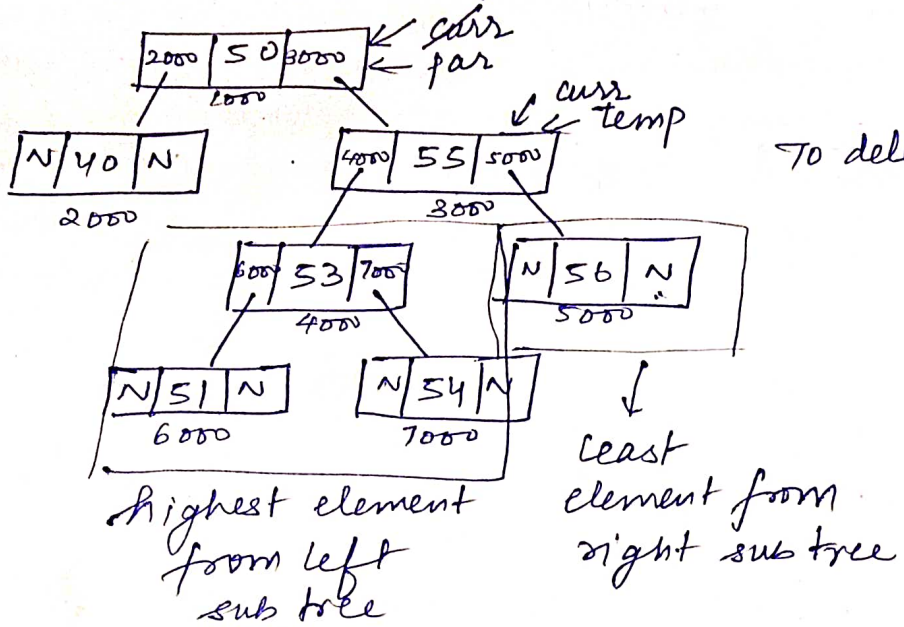
Deleting a node having 1 child (either right or left)




```
void delete()
{
int item;
struct node *temp, *par, *curr;
printf ("\n Enter item to be deleted");
scanf ("%d", &item);
while (curr != NULL)
{
if (item == curr -> data)
{
temp = curr; break;
} par = curr;
else if (item < curr -> data)
{
curr = curr -> left;
}
else
{
curr = curr -> right;
}
}
```

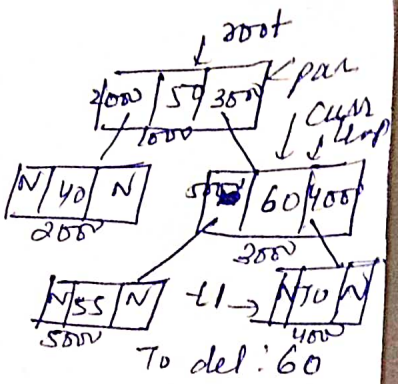
```
if (temp -> left != NULL)
{
if (temp == par -> right) {
par -> right = temp -> left;
}
temp -> left = NULL;
free (temp);
}
else if (temp -> right != NULL)
{
if (temp == par -> right)
{
par -> right = temp -> right;
}
temp -> right = NULL;
free (temp);
}
else if (temp -> left != NULL)
{
if (temp == par -> left)
{
par -> left = temp -> left;
}
temp -> left = NULL;
free (temp);
}
else
{
if (temp == par -> ~~right~~ left)
{
par -> left = temp -> right;
}
Temp -> left = NULL;
free (temp); } }
```

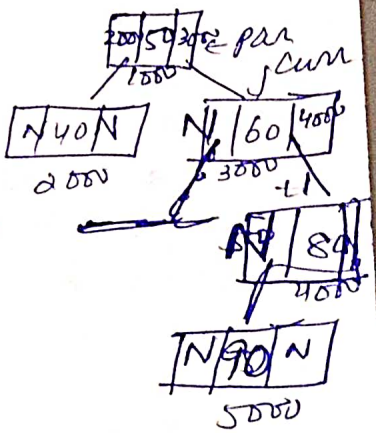
Deleting a node having two children




```
void delete ( )
{
struct node *temp, *par, *curr;
 curr = root;      printf("Enter item to be deleted");
while(curr != NULL)   scanf("%d", &item);
 {
  if(item == curr -> data)
   {
      temp = curr;
      break;
   }
   par = curr;
else if(item <= curr -> data)
   {
    curr = curr -> left;
   }
  else
   {
    curr = curr -> right;
   }
}
```

```
if (temp->left != NULL && temp->right != NULL)
{
struct node *t1, *t2;
t1 = temp -> right;
if (t1->left == NULL && t1->right == NULL)
{
   temp->data = t1->data;
   temp->right = NULL;
   free(t1);
}
else if (t1->left == NULL && t2->right != NULL)
{
   temp->data = t1->data;
   temp->right = t1->right;
   t1->right = NULL;
   free(t1);
}
else
{
  t2 = t1->left;
  while (t2 != NULL)
  {
   t2 = t2->left;
  }
temp->data = t2->data;
t1->left = t2->right;
t2->left = NULL;
free(temp);
}
}
if (t2->left == NULL && t2->right == NULL)
{
  t1->left = NULL;
  free(t2);
}
```





```
else
{
t1->left = t2->right;
t2->right = NULL;
free(t2);
}
}
}
```

AVL Tree

AVL tree is invented by three scientists GM Adelson & Velsky and EM Landis in 1962.

AVL Tree can be defined as height balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right subtree from height of its left subtree.

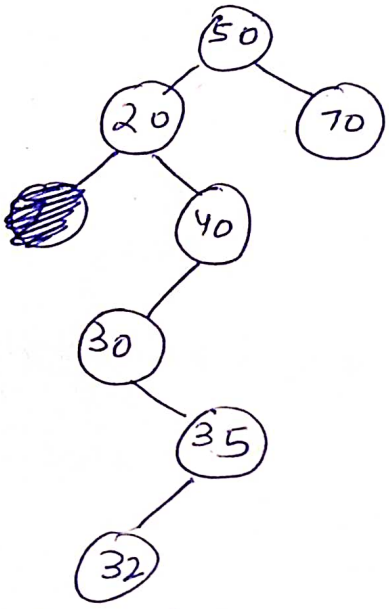
Tree is said to be balanced if balance factor of each node is between -1 to 1 i.e bf = -1, 0, 1, otherwise tree will be unbalanced and need to be balanced.

Balance factor (node) = height of left subtree - height of right subtree.

1. If b.f of any node is 1, it means that left subtree is one level higher than right subtree.
2. If bf of any node is 0, it means that left subtree and right subtree has equal height.
3. If bf of any node is -1, it means that left subtree is one level higher than right subtree.

Problem with BST is that if height is increasing and ~~last~~ element to be searched is at end, searching will take a lot of time.

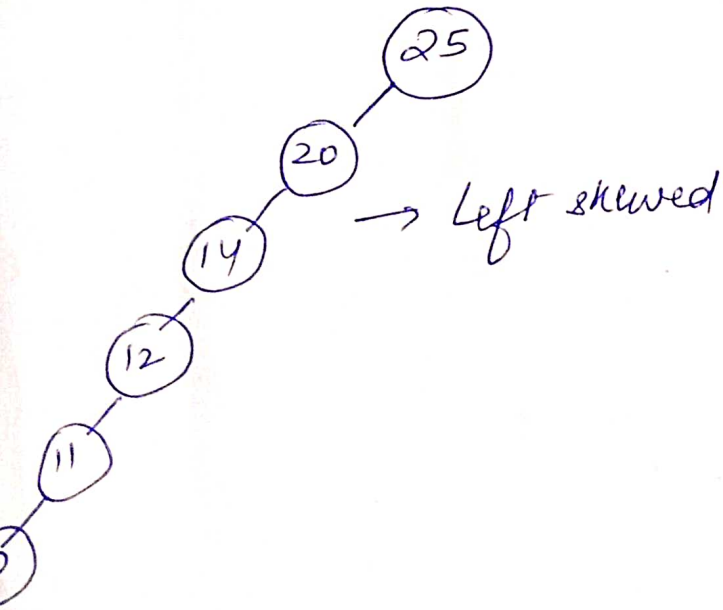
For ex: 50, 20, 70, 40, 30, 35, 32



If we want to search 32, we have to traverse the entire tree of height h.

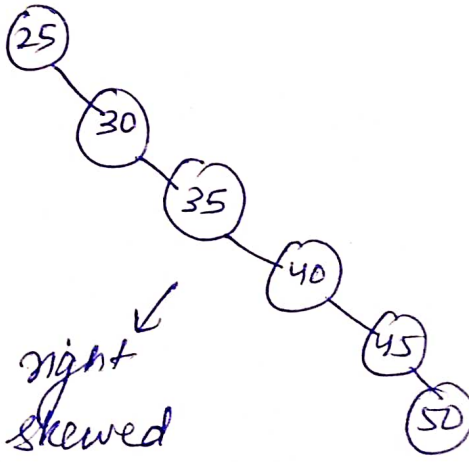
Also using BST, we can get left skewed or right skewed binary search tree.

For ex: 25, 20, 14, 12, 11, 10



→ Left skewed

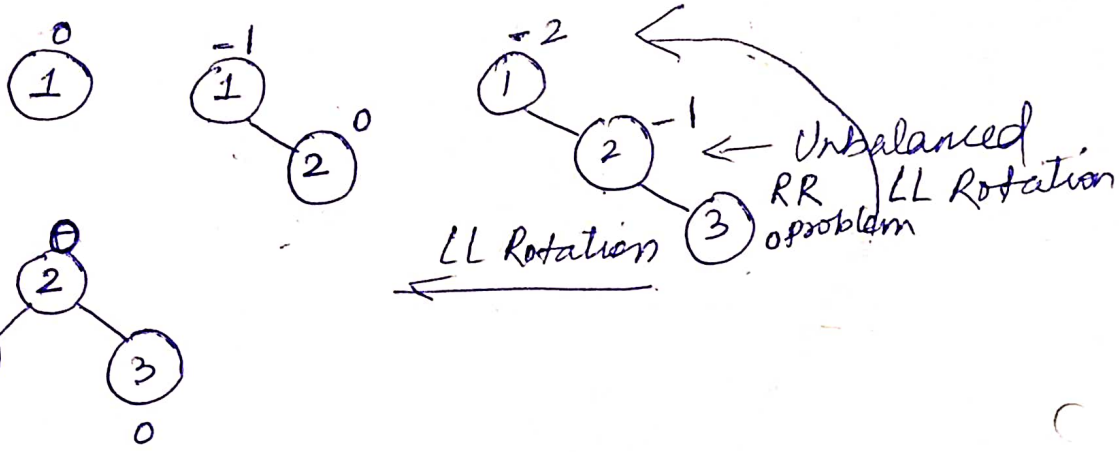
For ex: 25, 30, 35, 40, 45, 50



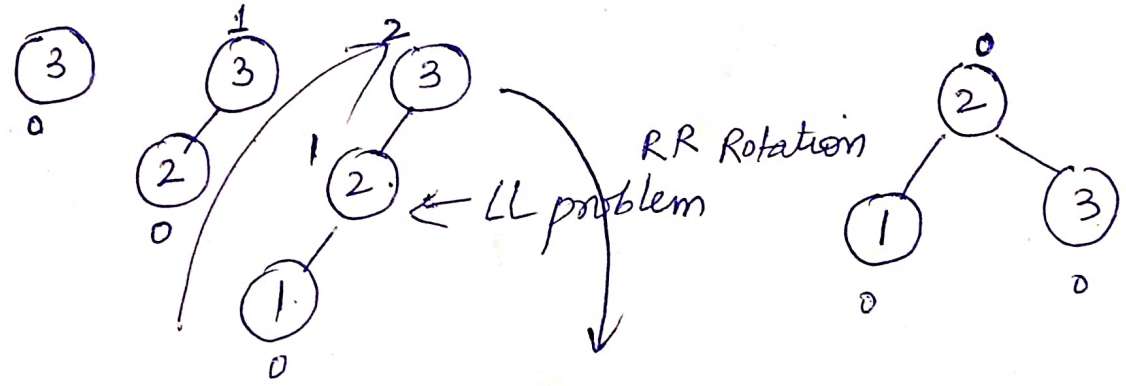
right skewed

To avoid the problem of unbalanced binary search tree we ~~select~~ use AVL tree, which makes the binary search tree balanced.

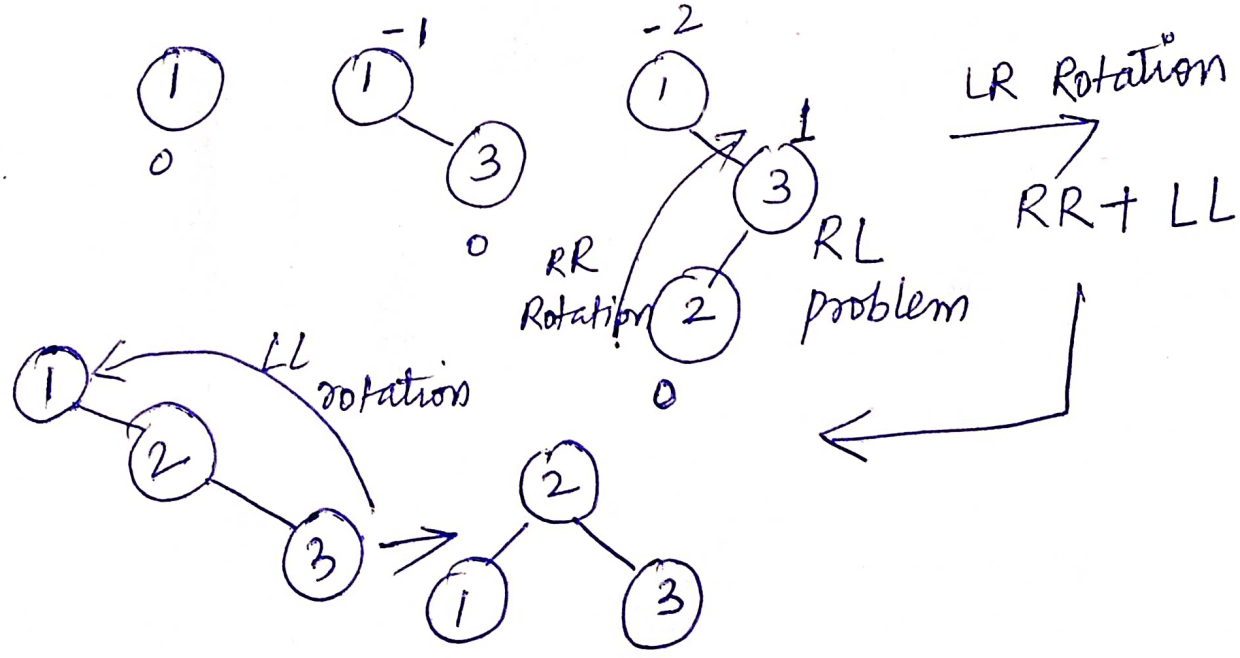
For ex:- To insert: 1, 2, 3



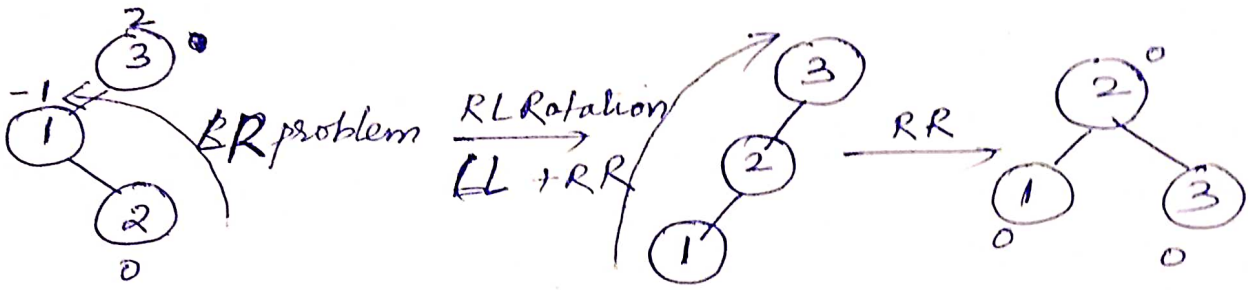
To insert 3, 2, 1



To insert: 1, 3, 2

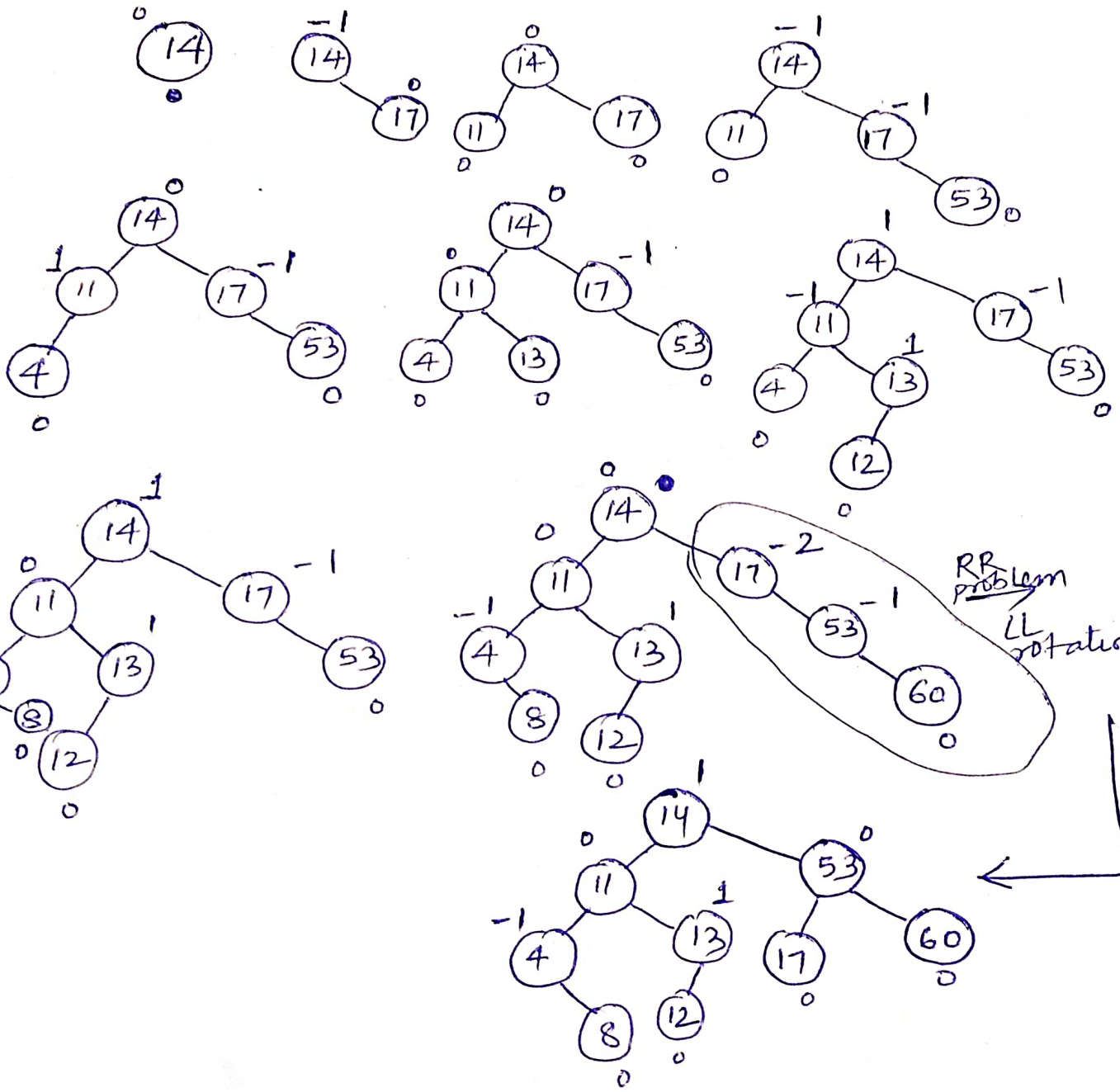


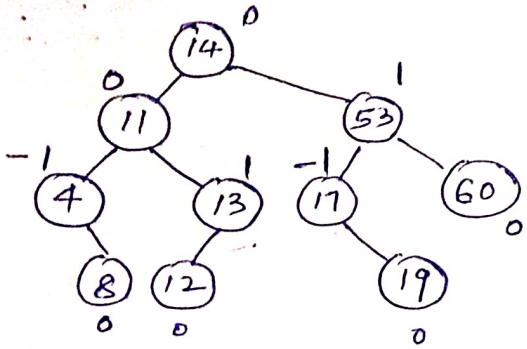
To insert: 3, 1, 2

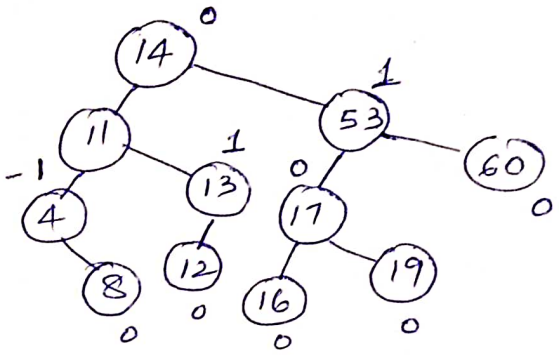


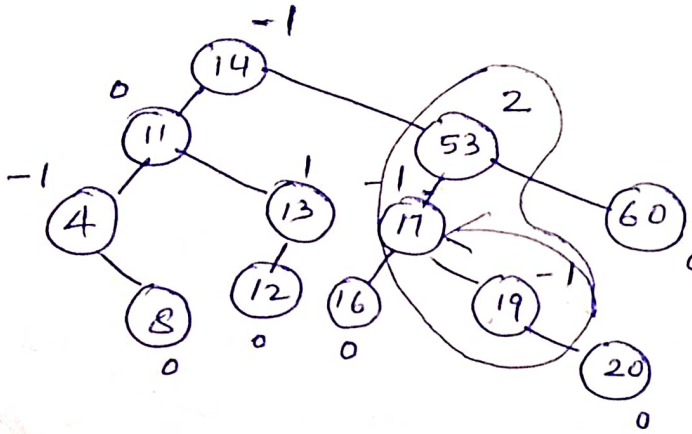
Ques → Construct AVL Tree by inserting the following data :

14, 17, 11, 53, 4, 13, 12, 8, 60, 19, 16, 20





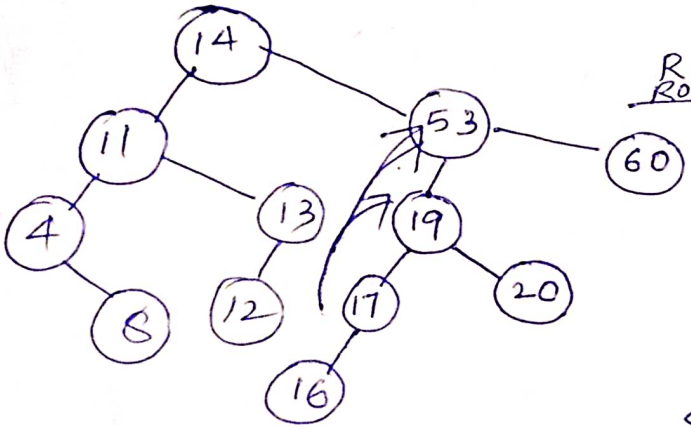




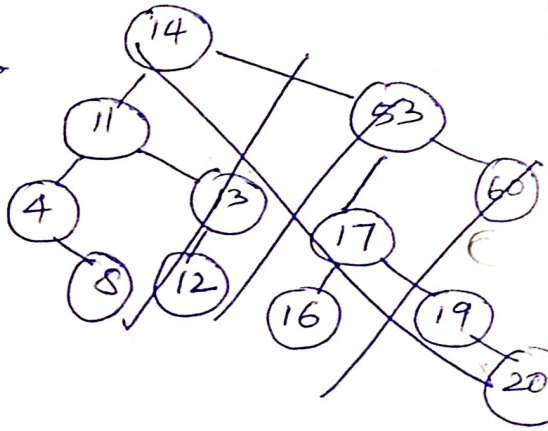
LR Problem

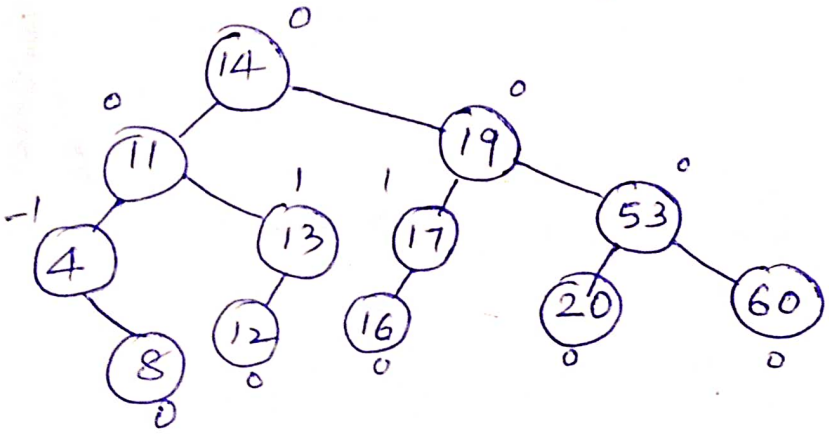
RL Rotation

LL Rotation



RR Rotation



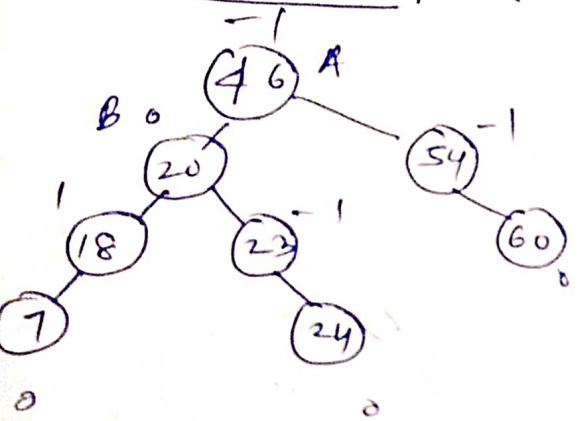


AVL Tree Deletion

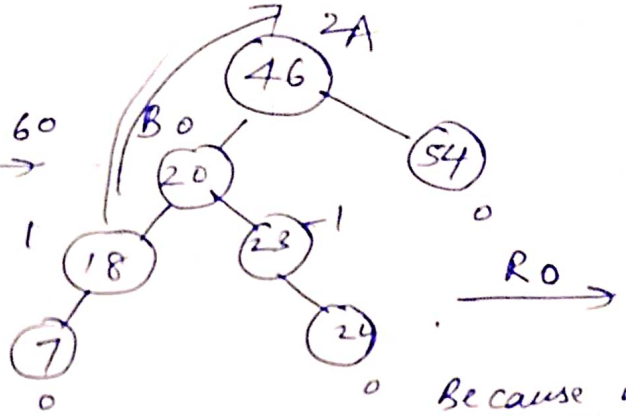
There are different types of Rotation in Deletion :

L0, R0, L(-1), R(-1), L(1), R(1)

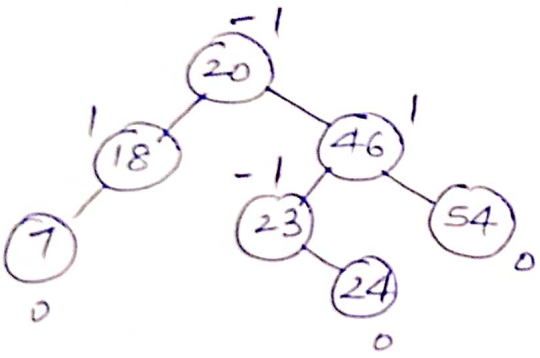
R0 Rotation / RR Rotation



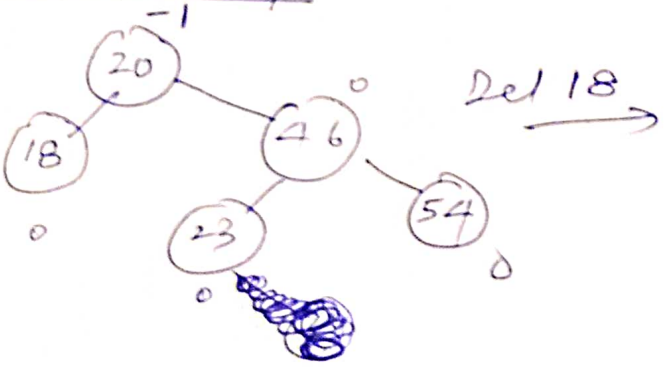
To del 60 →



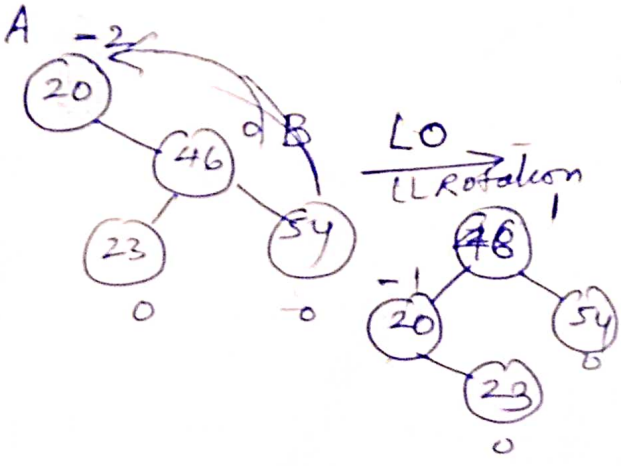
Because we have deleted from right hand side and degree of B is 0. We will do right shift from node B



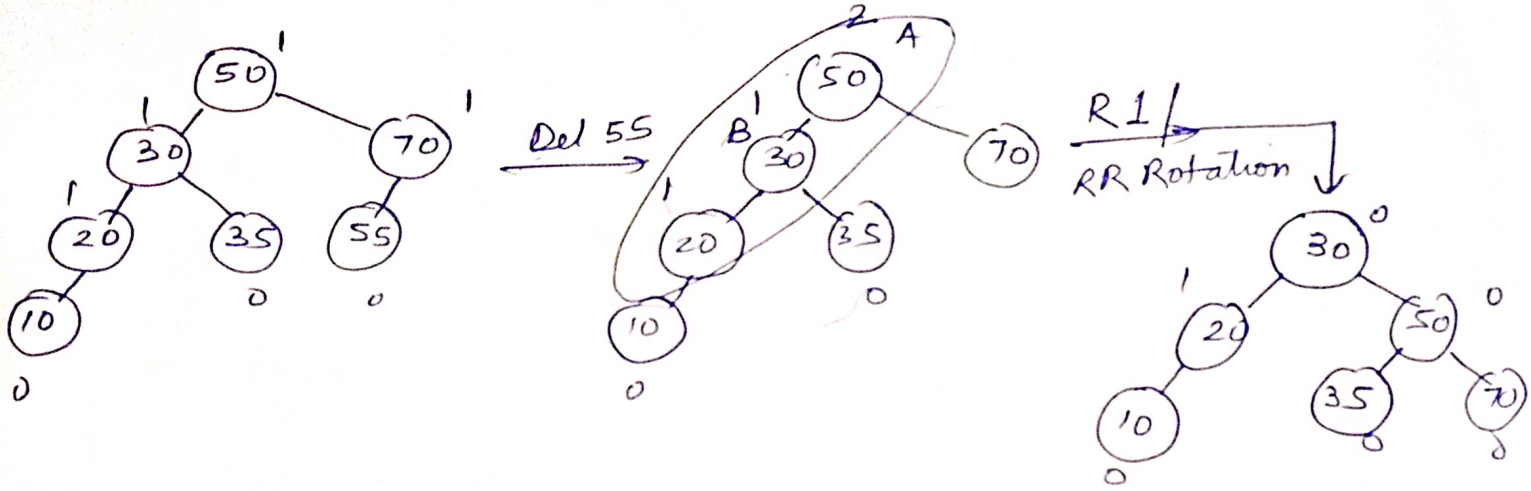
L0 Rotation / LL Rotation



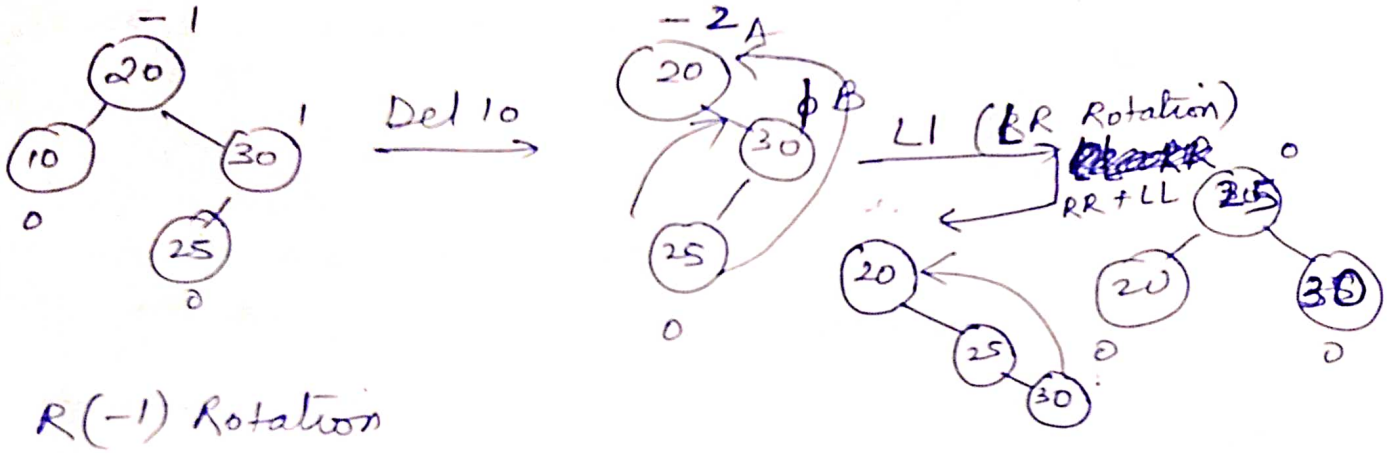
Del 18 →



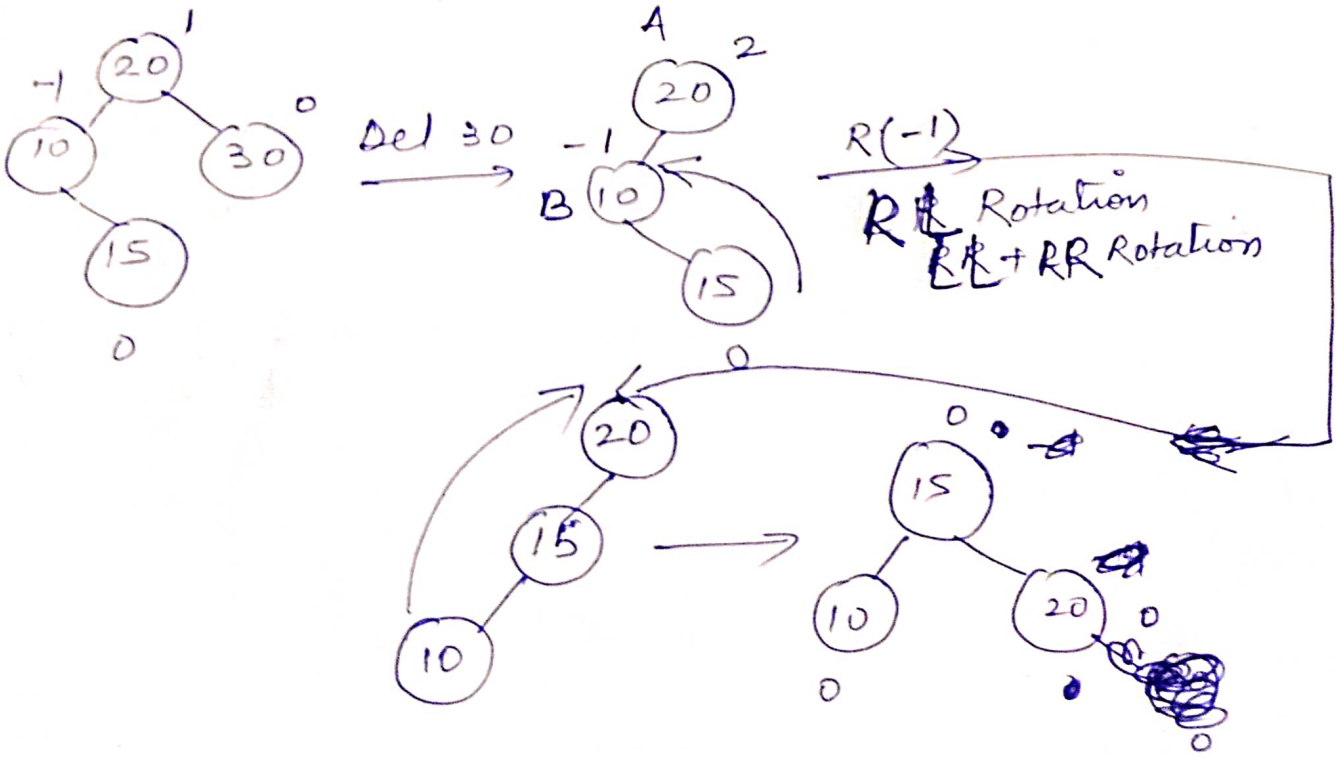
R1 Rotation



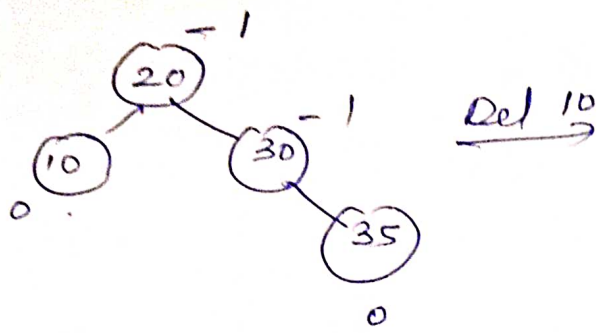
L1 Rotation

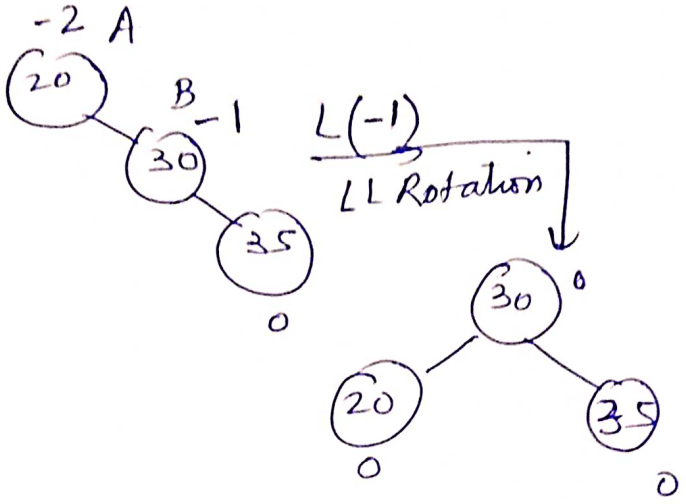


R(-1) Rotation

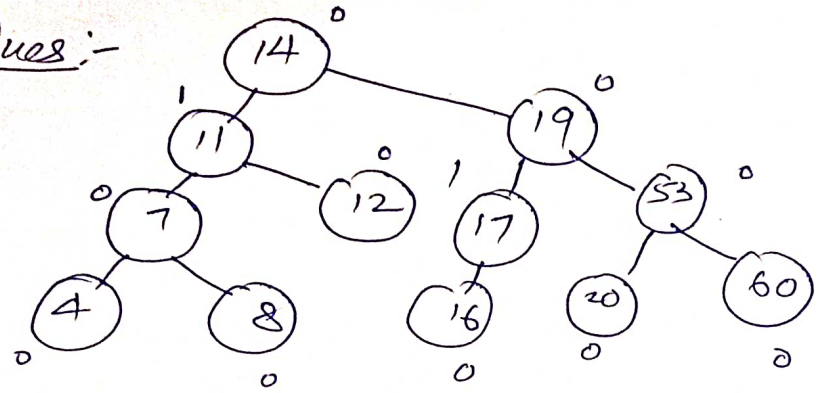


L(-1) Rotation

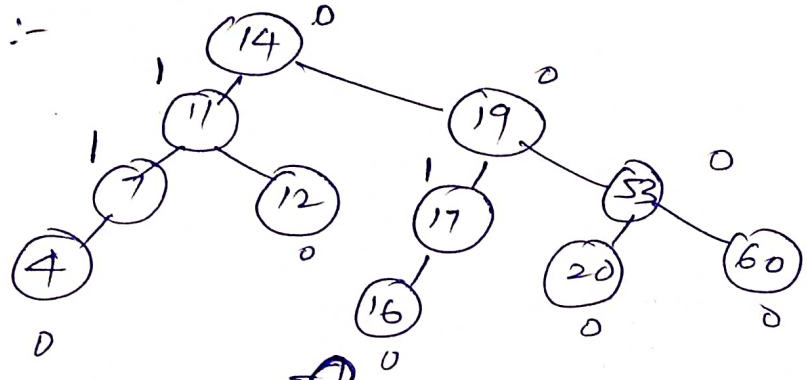




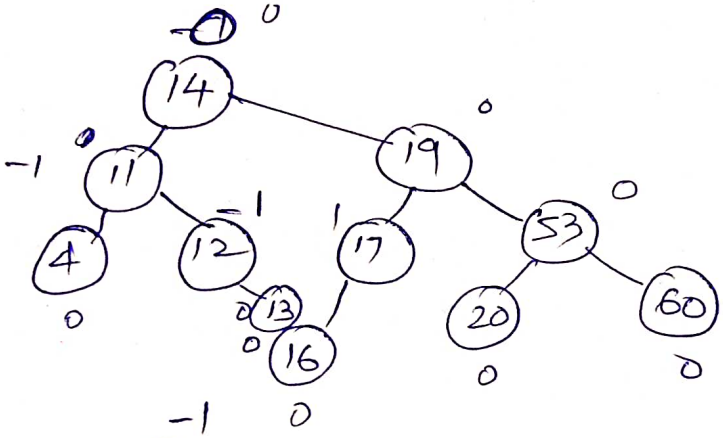
Ques :-



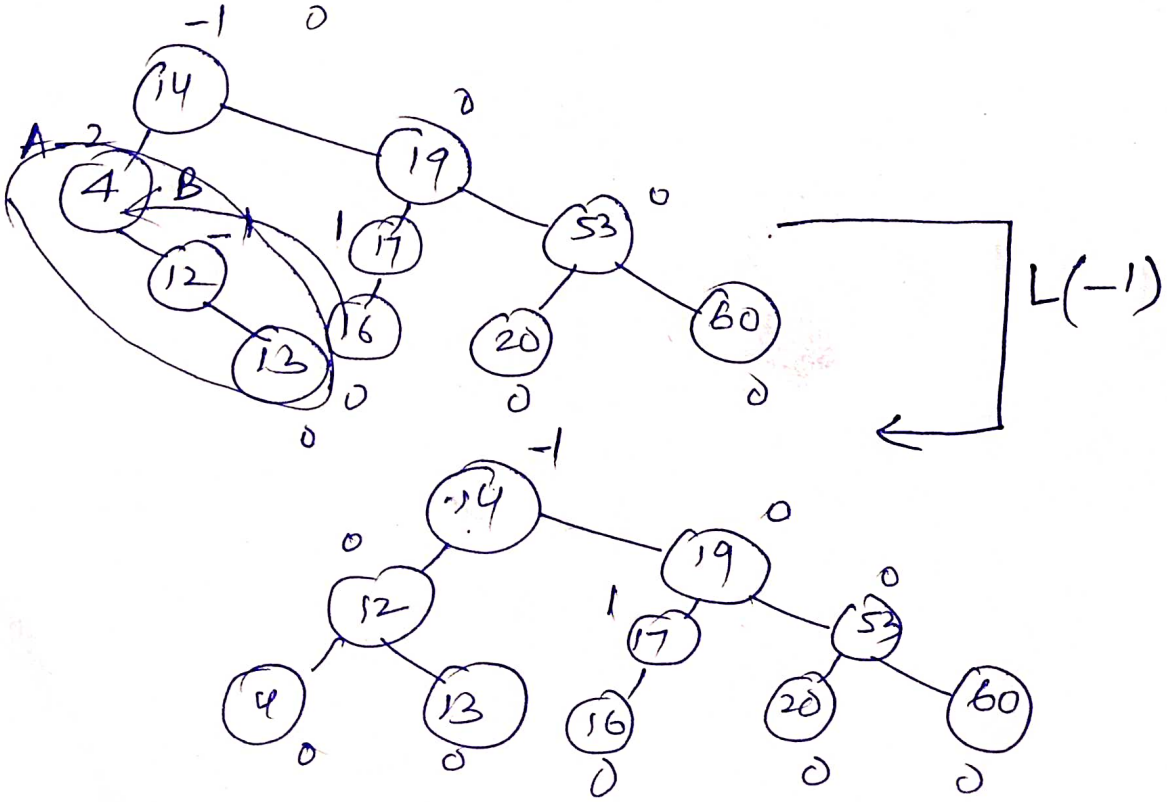
Del 8 :-



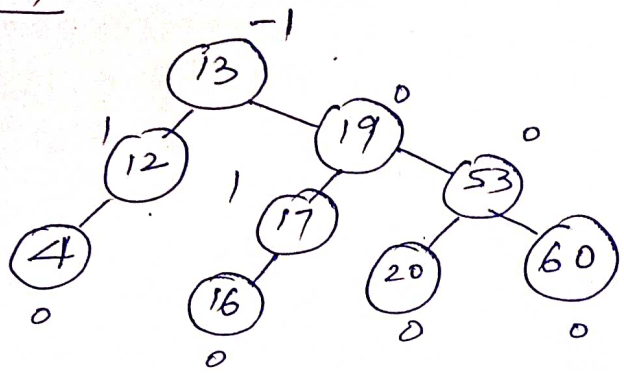
Del 7 :-



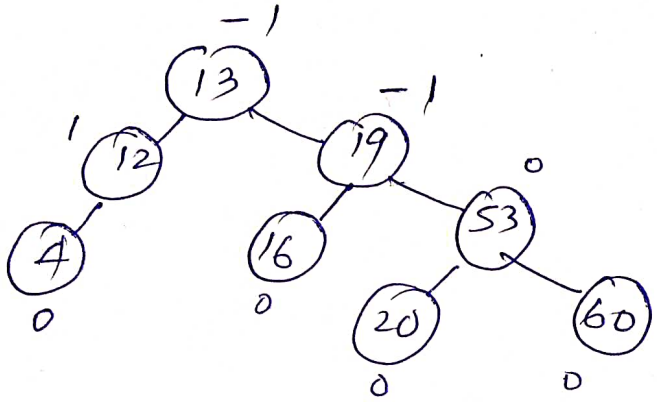
Del 11 :-



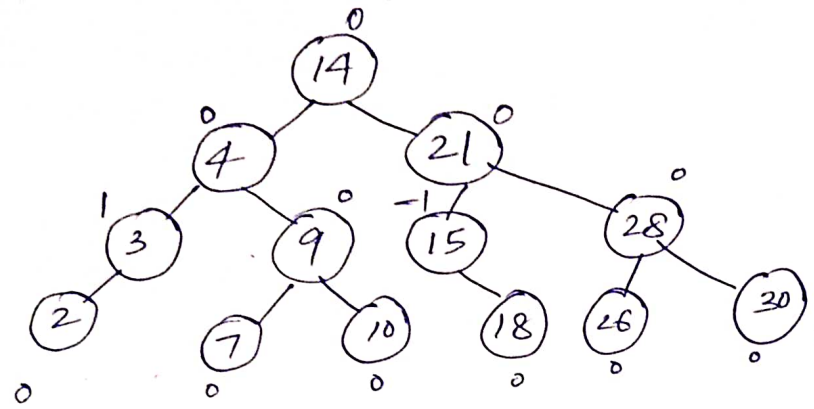
Del 14



Del 17

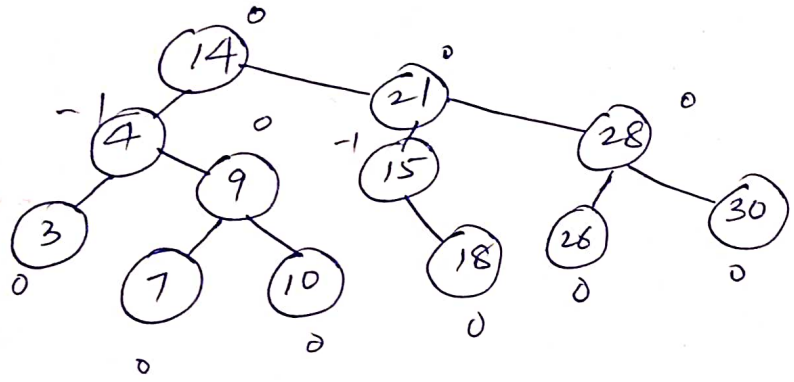


Ques:- Consider the following AVL Tree.

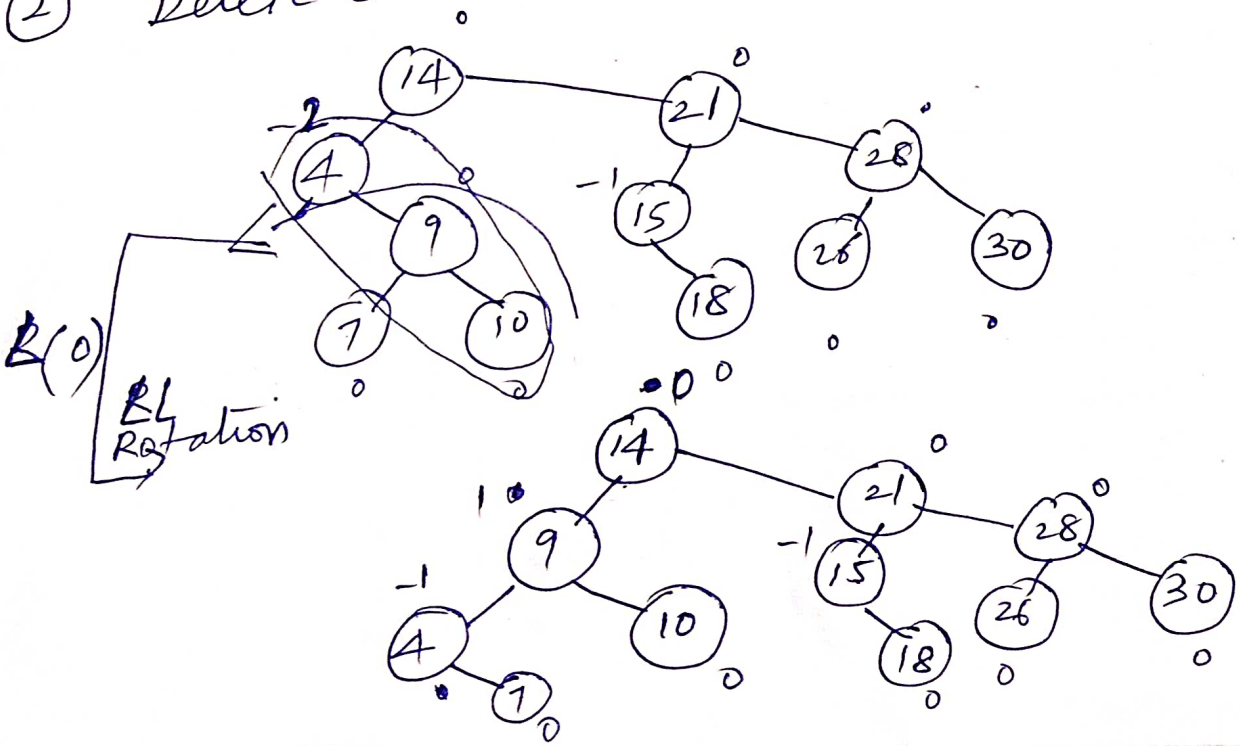


Delete the following : 2, 3, 10, 18, 4, 9, 14, 7, 15

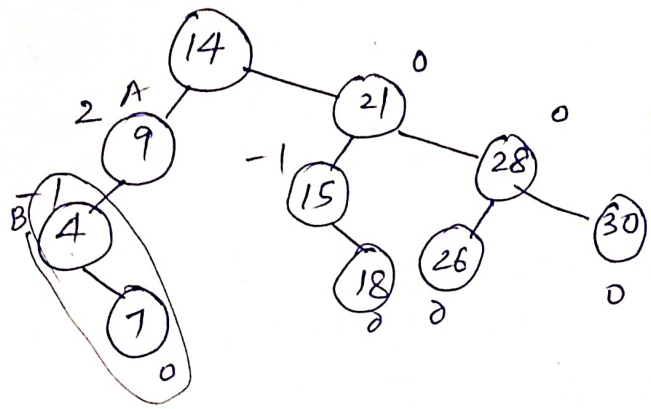
(1) Delete 2



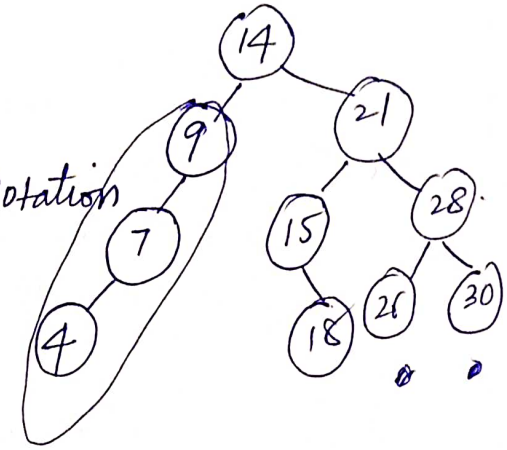
(2) Delete 3

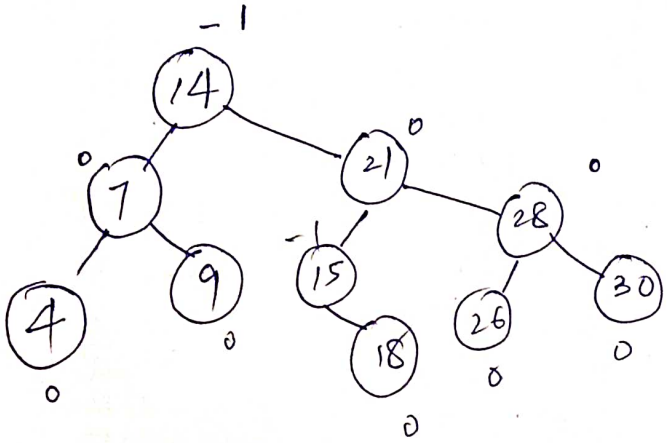


③ Delete 10

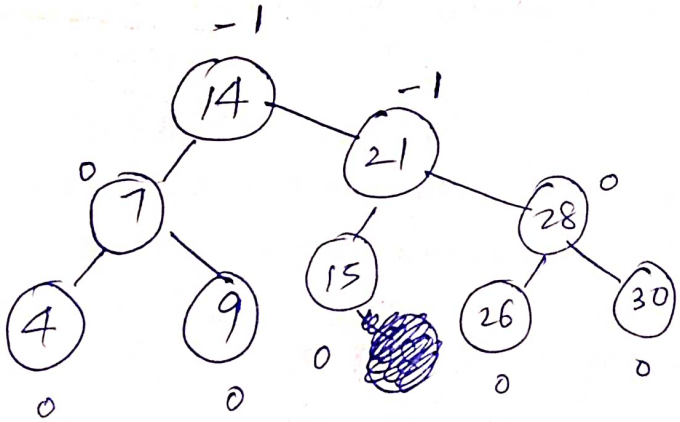


R(-1)
RL Rotation

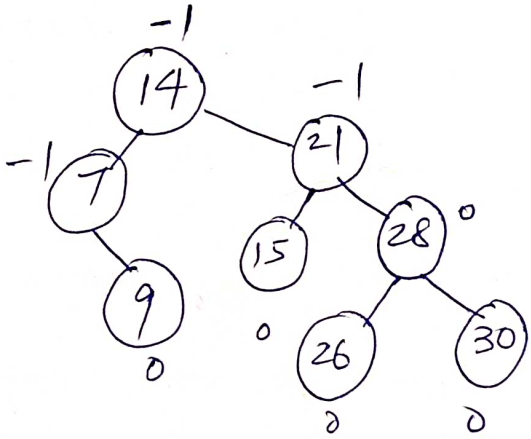




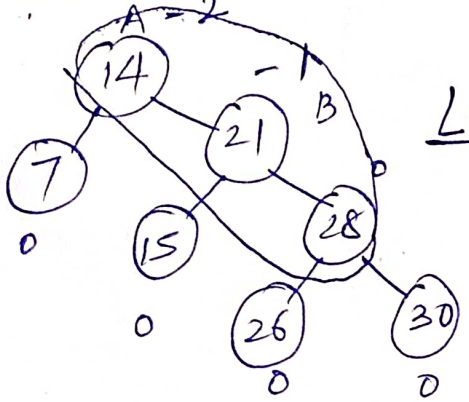
④ Delete 18



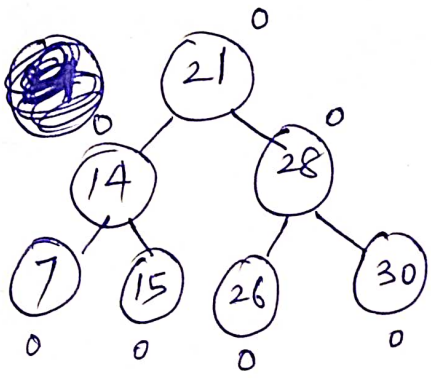
⑤ Delete 4



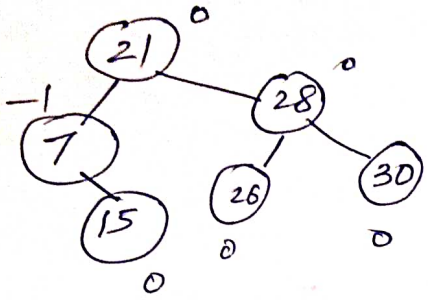
⑥ Delete 9



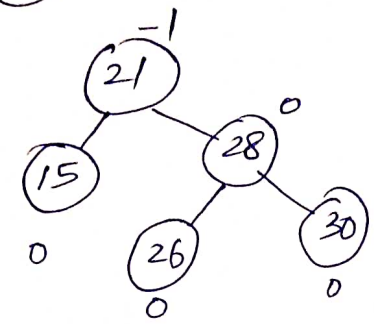
L(-1)
LL Rotation



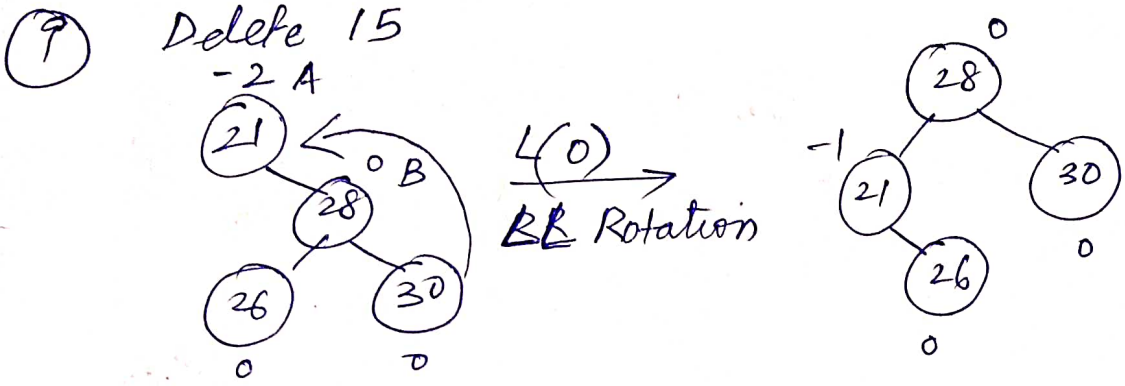
(7) Delete 14



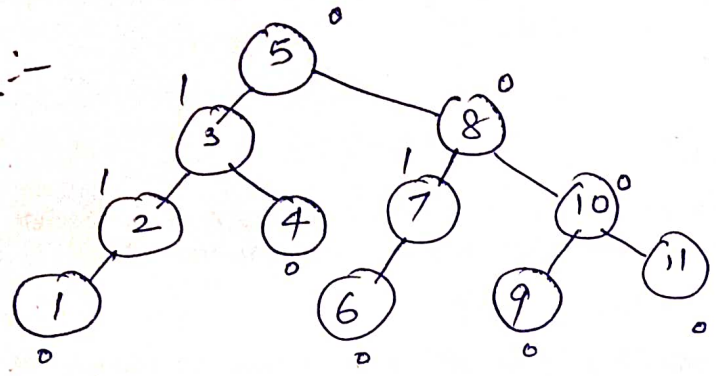
(8) Delete 7



(9) Delete 15

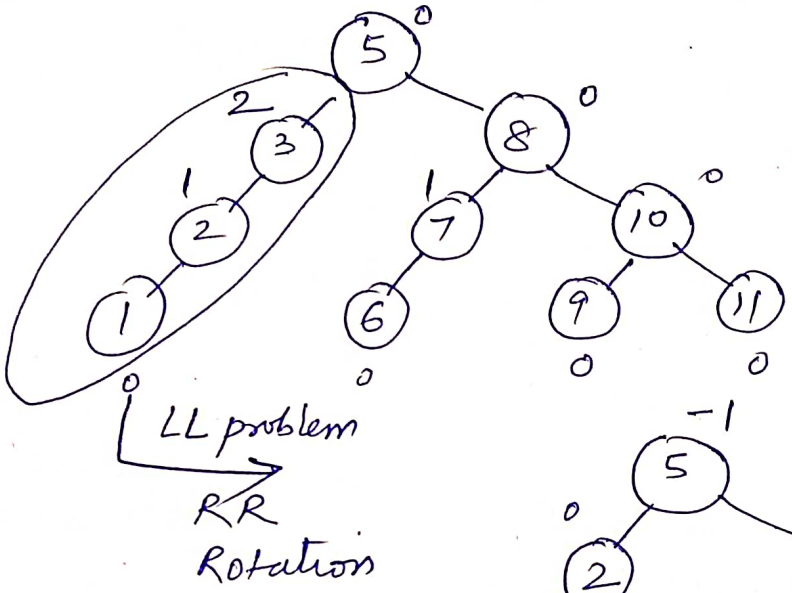


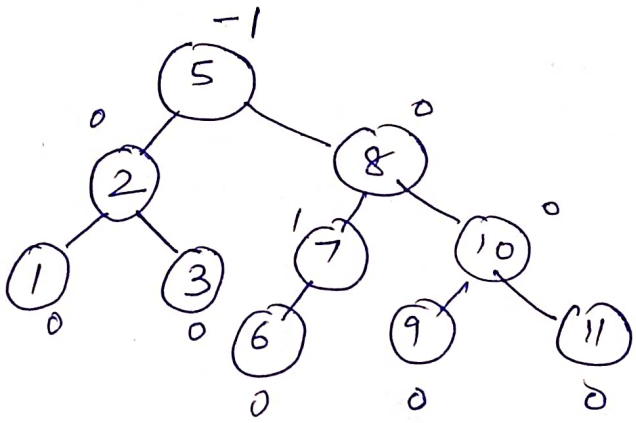
Ques :-



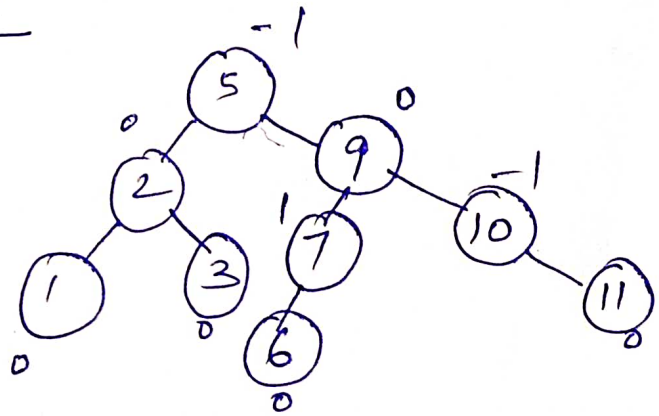
To delete :- 4, 8, 6, 5, 2, 7

Delete 4

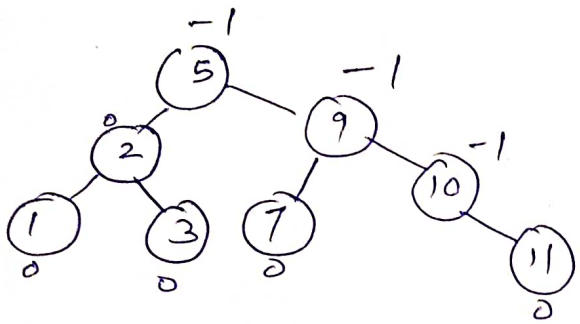




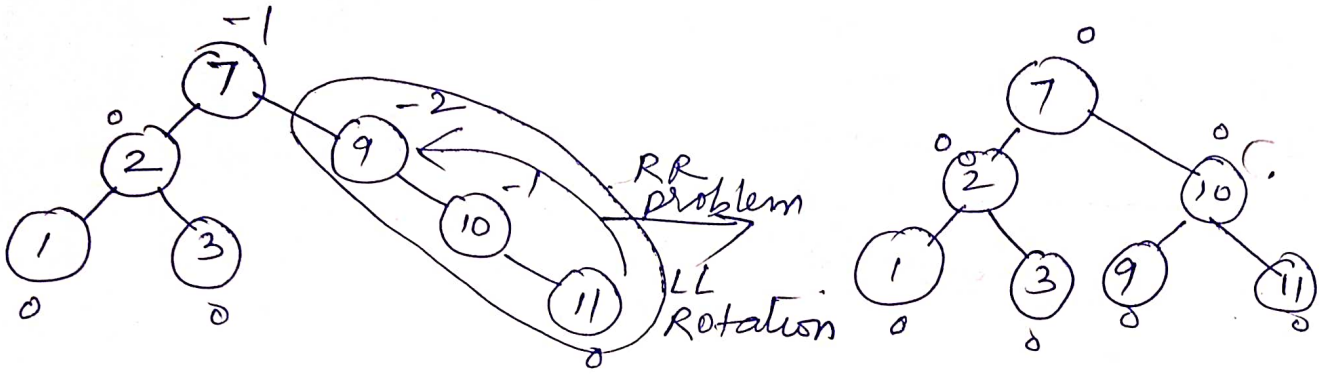
Delete 8



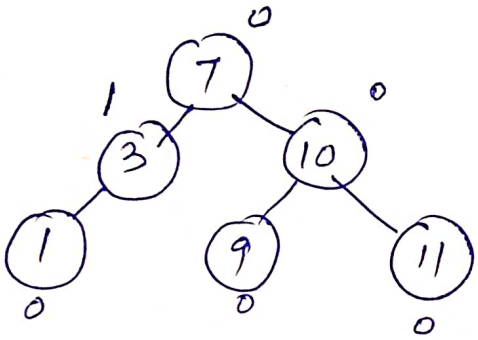
Delete 6



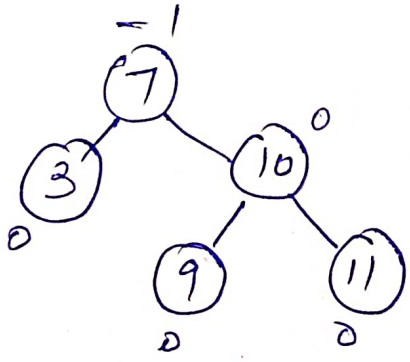
Delete 5



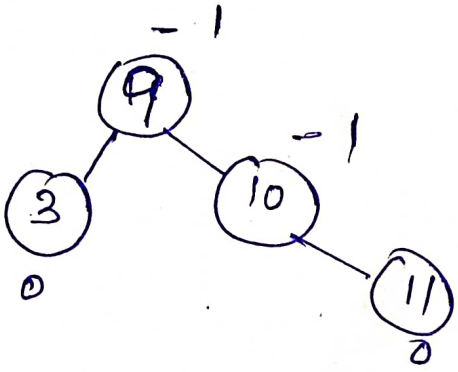
Delete 2



Delete 1



Delete 7



Heap Tree

A heap is a tree based data structure and the heap tree is a complete binary tree

There are two types of heap tree :

1. Min Heap Tree
2. Max Heap Tree

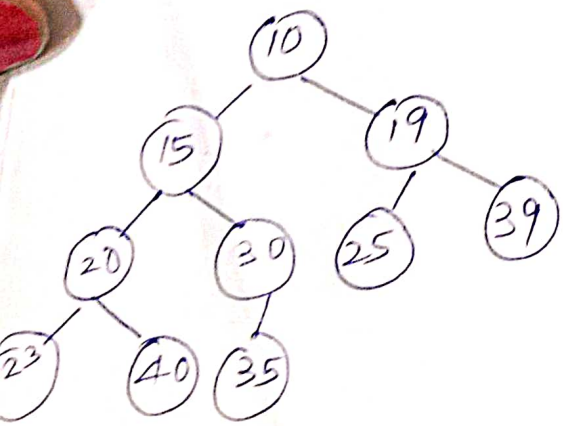
Min Heap Tree

In Min Heap Tree, each parent node is less than or equal to its both child nodes. Except root node

Max Heap Tree

In Max Heap Tree, each parent node is greater than or equal to its both child nodes. Except root node.

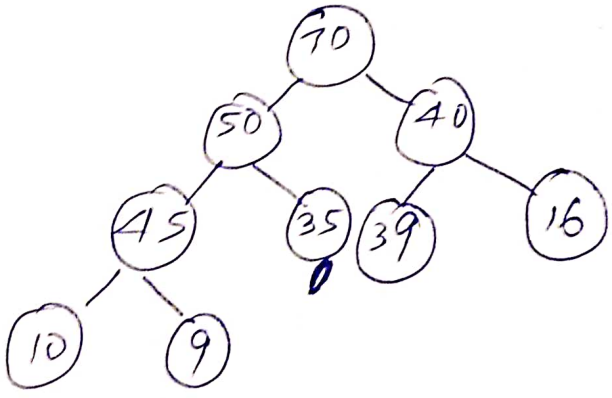
Min Heap



$A[Parent[i]] \leq A[i]$

	10	15	19	20	30	25	39	23	40	35
	0	1	2	3	4	5	6	7	8	9

Max Heap



$A[Parent[i]] \geq A[i]$

A =	70	50	40	45	35	39	16	10	9
	0	1	2	3	4	5	6	7	8

In Heap Tree, if the parent is at index i, then left child will be at index $2i+1$ and right child will be at index $2i+2$.

If child is at index i, then its parent will be at index

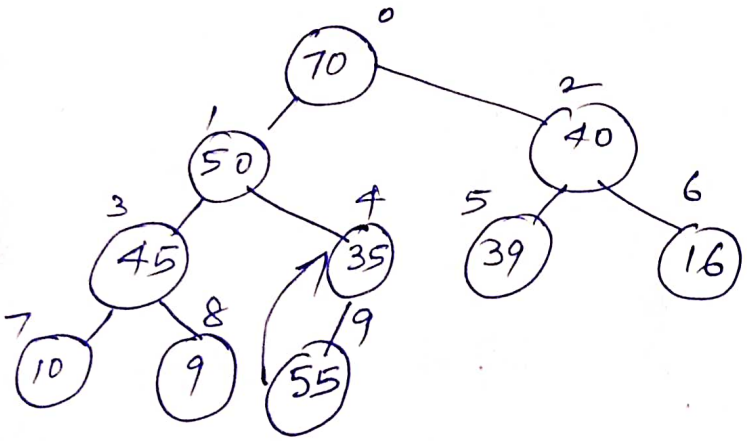
$$\left\lceil \frac{i}{2} \right\rceil - 1$$

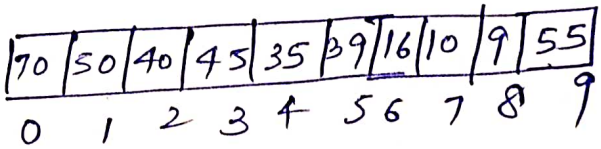
↳ ceil

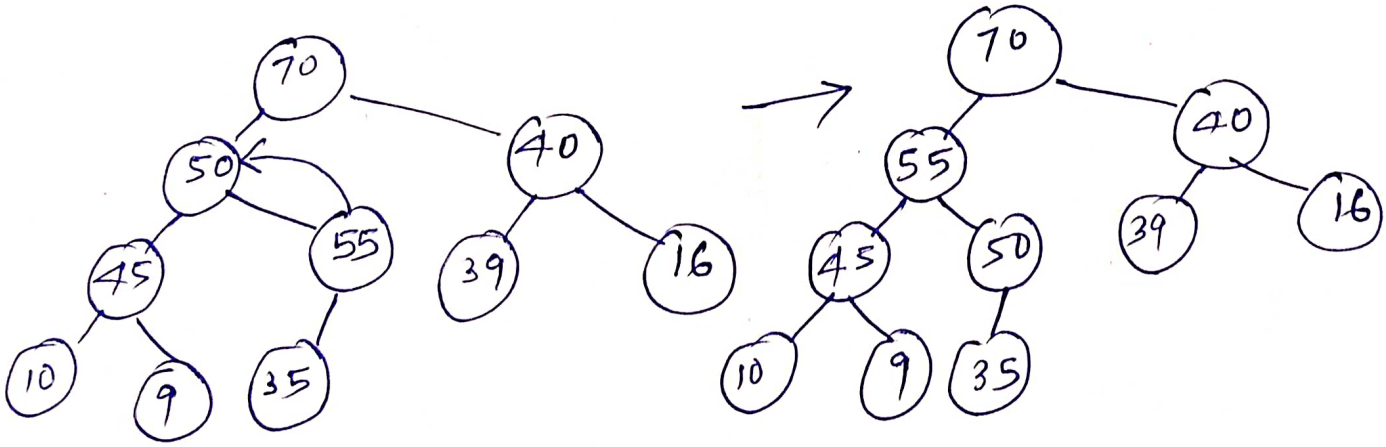
In heap Tree, data will always be inserted in leaf node.

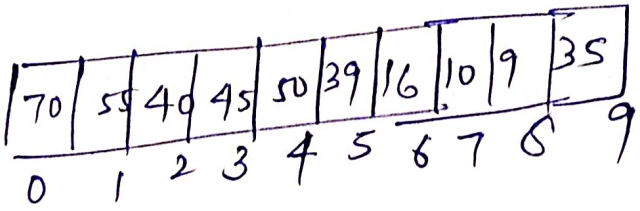
Max-heap

To insert : 55









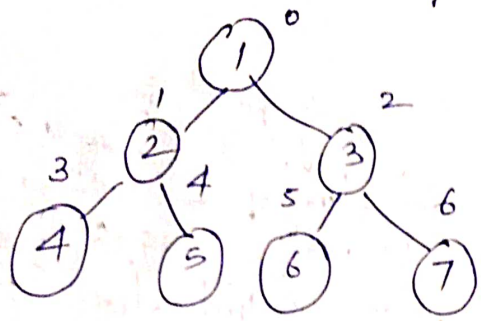
```
int insertHeap(A, n, value)
{
    m = n - 1 + 1
    A[m] = value
    i = m;
    While (i > 0)
    {
      Parent = ⌈i/2⌉ - 1;
      if (A[Parent] < A[i])
      {
        swap(A[Parent], A[i]);
        i = Parent;
      }
      else
      {
      return;
      }
    }
}
```

DELETION IN HEAP TREE

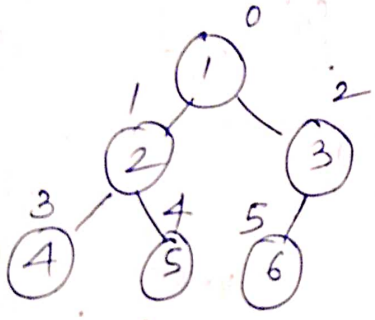
There are 2 ways to delete an element in Heap tree:

1. Either delete the last element of the last level
2. Delete the root element and replace it with the last element of the last level.

Consider a min heap tree

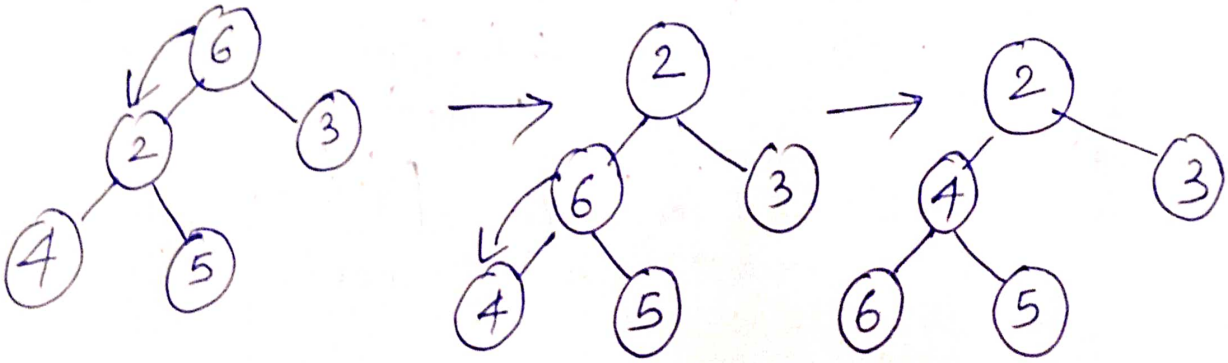


Delete 7



Delete 1

Replace 1 with 6

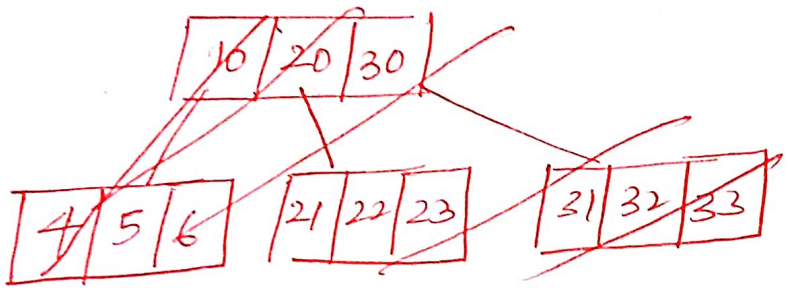


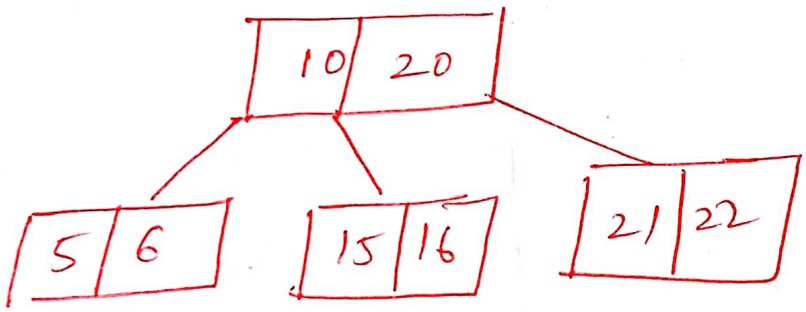
M-way Search Tree

M-way search tree are multiway search trees which are generalised version of binary trees where each node contains multiple elements.

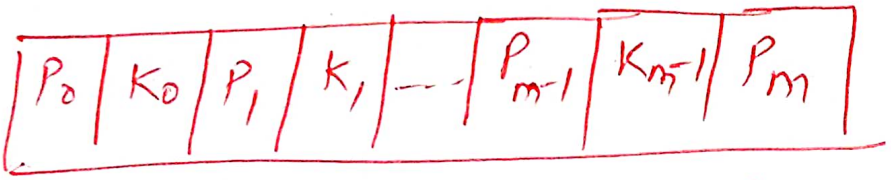
In a m-way tree of order m, each node contains a maximum of m-1 elements and m children.
(i.e 1 to (m-1). 0 to m)

If m = 3





Representation



m-1 values
m pointers

$K_0 < K_1 < K_2 - - - - K_{m-1}$

Construct M way search tree for the following elements

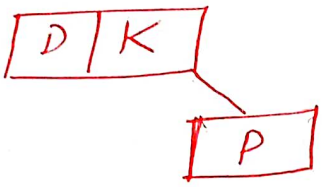
D, K, P, V, A, G , $m = 3$

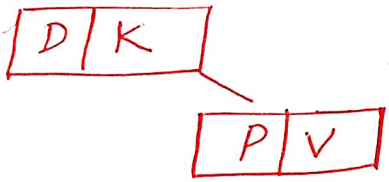
Maximum children in each node 2

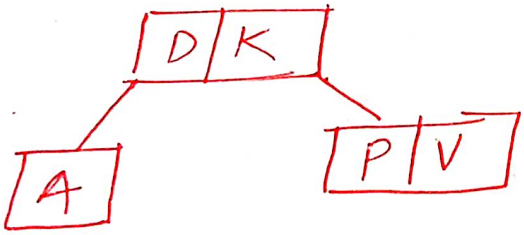
① 

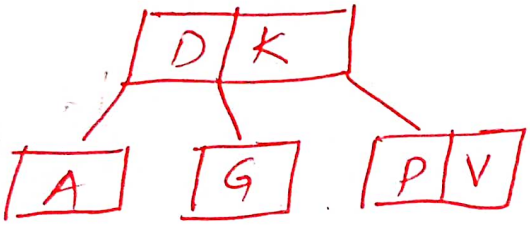
② 

③ 

④ 

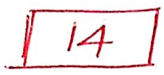
⑤ 

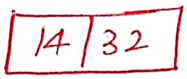
⑥ 

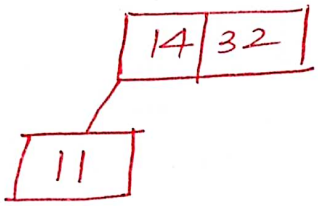
⑦ 

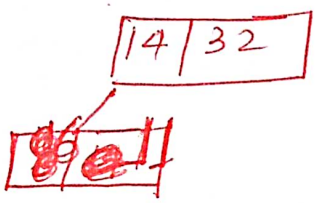
Q. Create a m-way search tree by inserting the following data ; order = 3

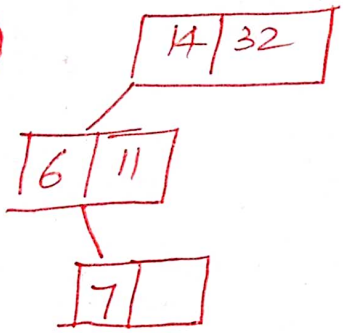
14, 32, 11, 6, 7, 4, 3, 88, 93, 54, 37, 21, 17, 89, 62, 64, 75, 35, 36

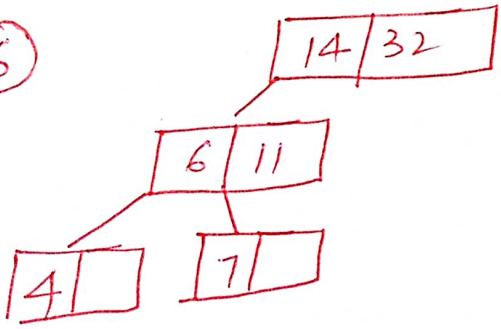
① 

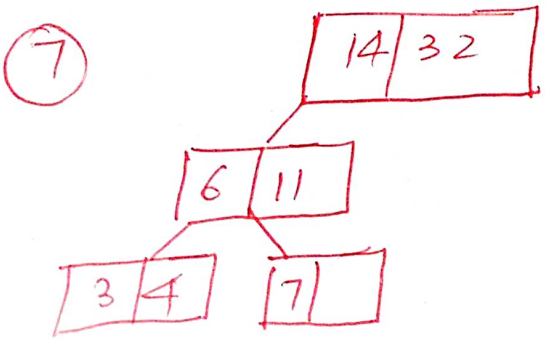
② 

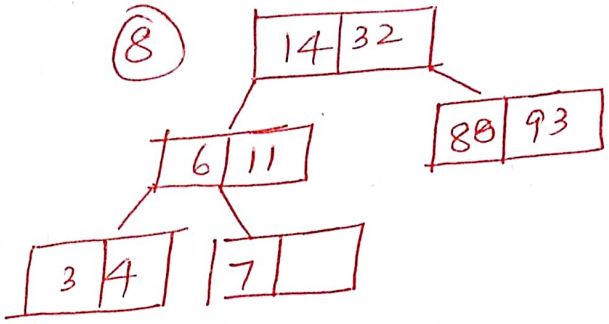
③ 

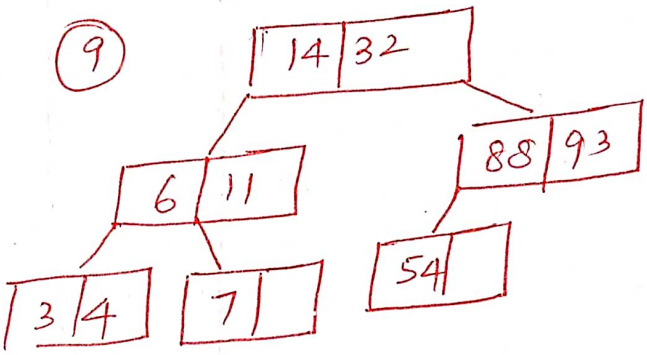
④ 

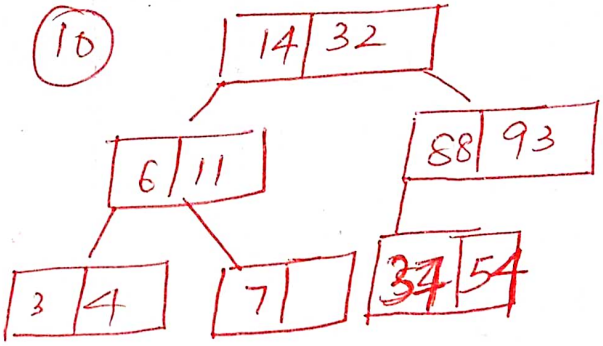
⑤ 

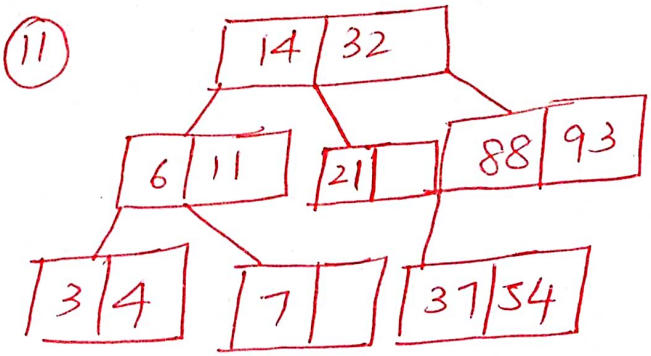
⑥ 

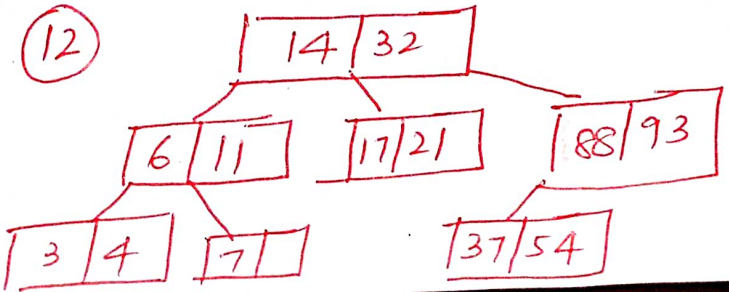
⑦ 

⑧ 

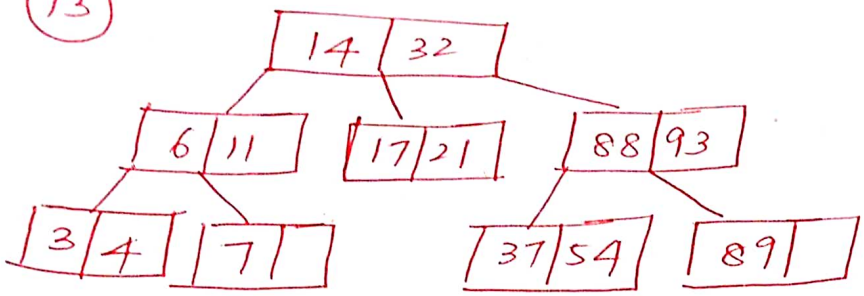
⑨ 

⑩ 

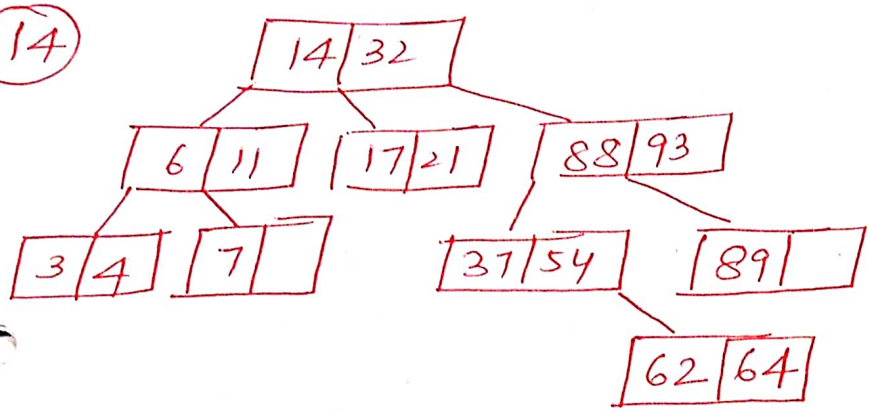
⑪ 

⑫ 

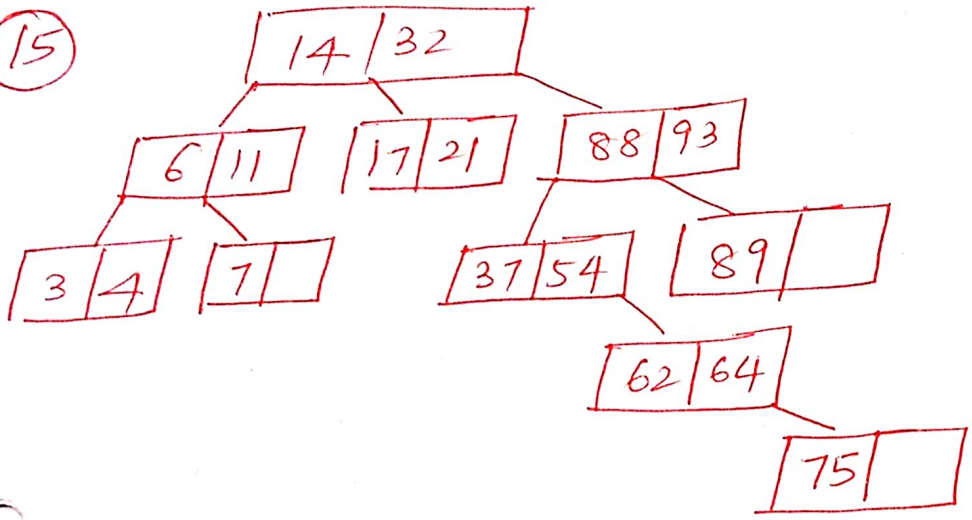
13


14 32
6 11
17 21
88 93
3 4
7
37 54
89

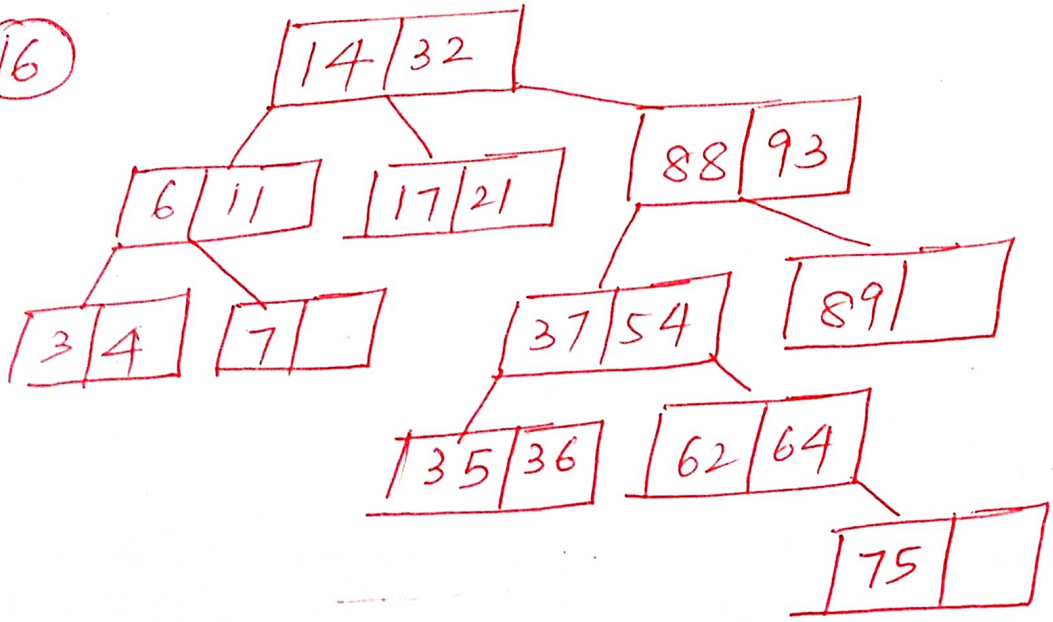
14


14 32
6 11
17 21
88 93
3 4
7
37 54
89
62 64

15


14 32
6 11
17 21
88 93
3 4
7
37 54
89
62 64
75

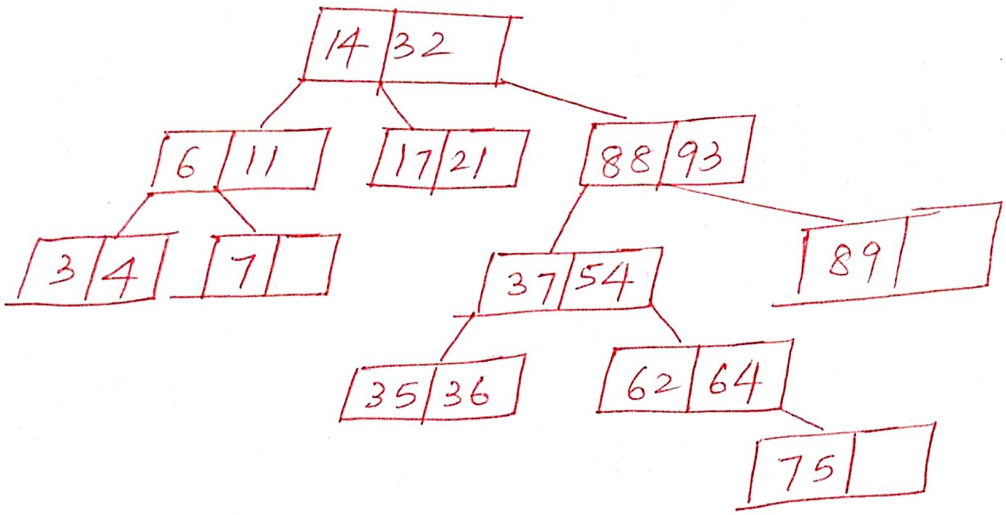
16


14 32
6 11
17 21
88 93
3 4
7
37 54
89
35 36
62 64
75

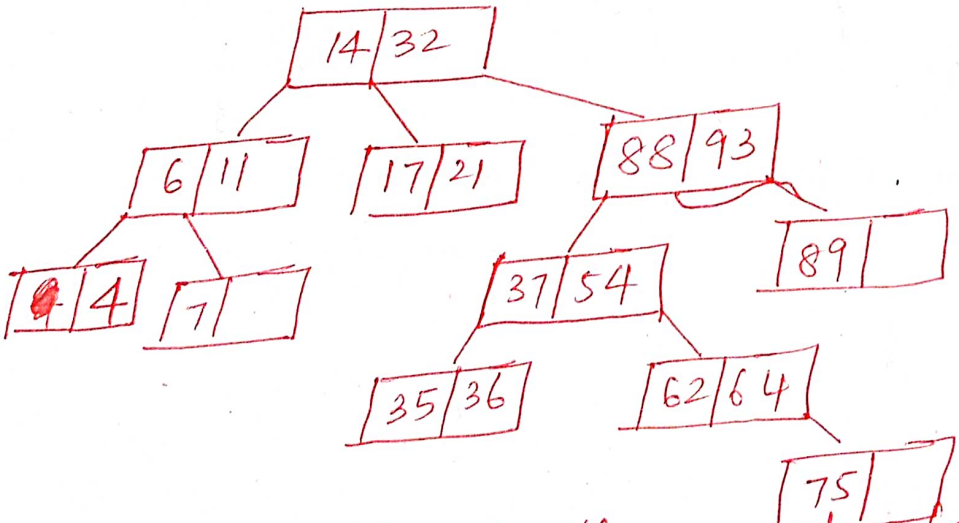
Deletion in m-way search tree

There are 4 cases

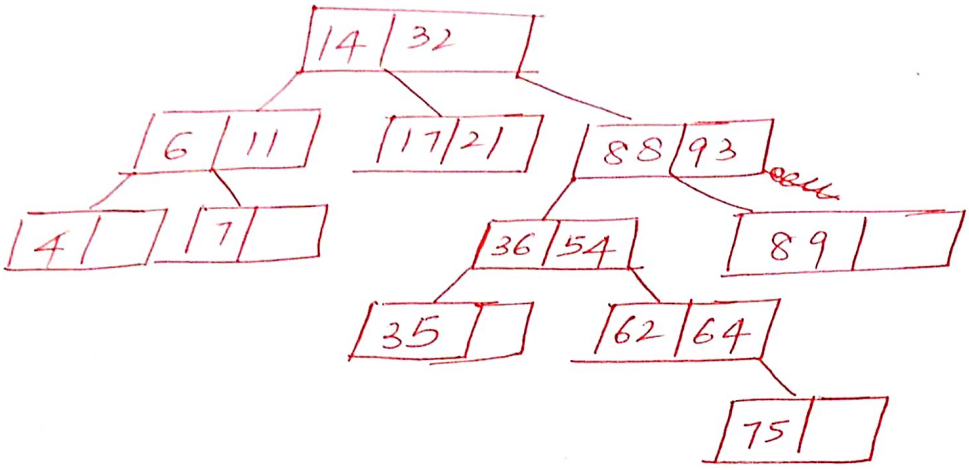
1. Deletion of leaf node
2. Deletion of key with left subtree
3. Deletion of key with right subtree
4. Deletion of key with both left and right subtree



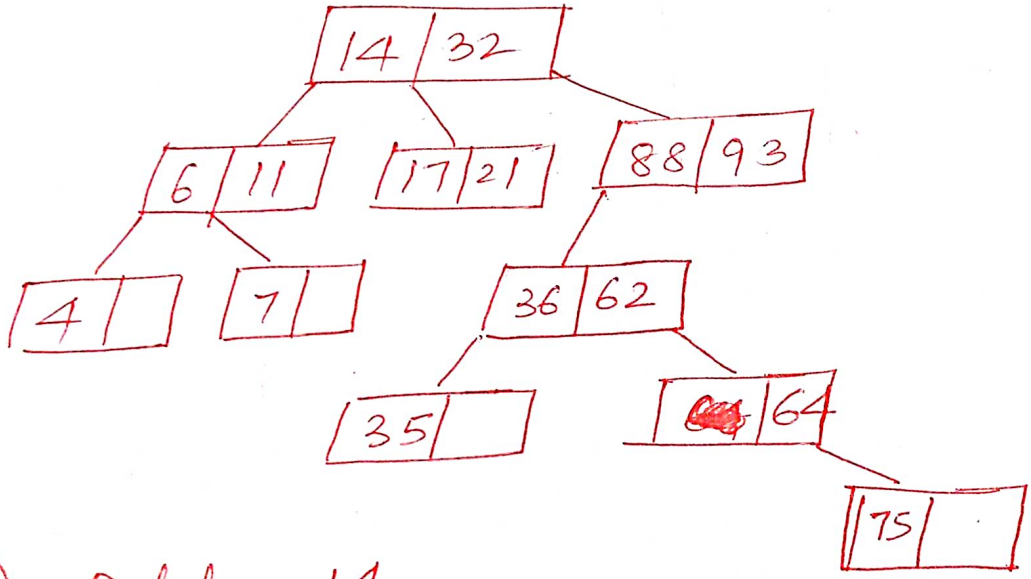
(1) Delete 3



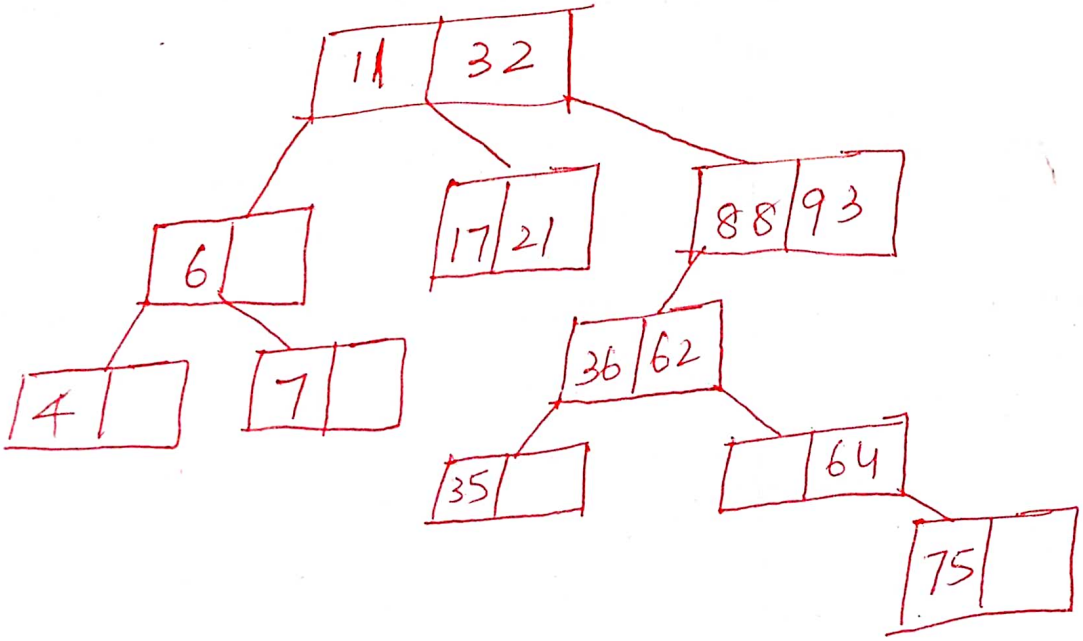
(2) Delete 37 (Replace it with max value from left subtree)



(3) Delete 54 (Replace it with min from right subtree)



(4) Delete 14



B-Tree

B-Tree is a specialised m-way tree that is used for disk access. A B-Tree of order m can have atmost m-1 keys and m children. B-Tree has the capability to stored large number of keys in a single node and keeping the height of the tree relatively small.

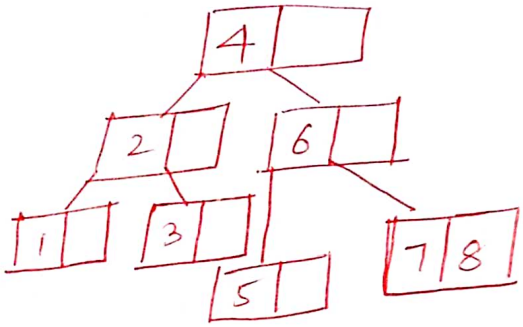
A B-Tree contains all the properties of m way tree. In addition it contains the following properties:

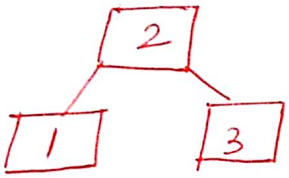
1. Every node in B-Tree can contain at most m children.
2. Every node in B-Tree except the root node and the leaf node contains atleast $m/2$ children. $\lceil m/2 \rceil$
3. The root node must have atleast two children.
4. All leaf nodes must be at the same level.
5. Every node will have maximum k-1 keys
6. Root node will have min key as 1
7. All other nodes except root node will have $\lceil m/2 \rceil - 1$ min keys.

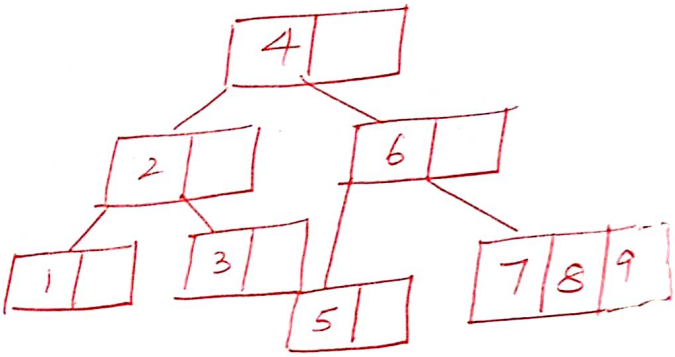
Here values will always be inserted in leaf node.
B-Tree will always grows upwards.

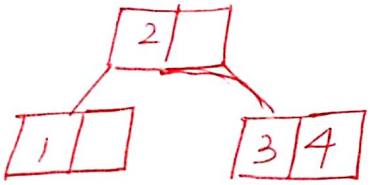
Q Data : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

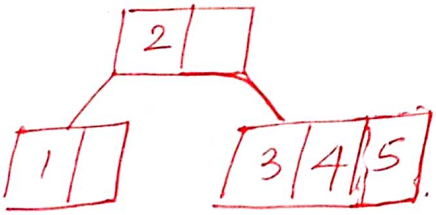
B-Tree of order 3

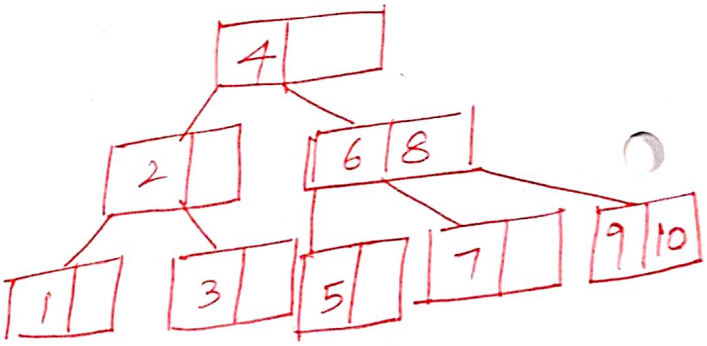


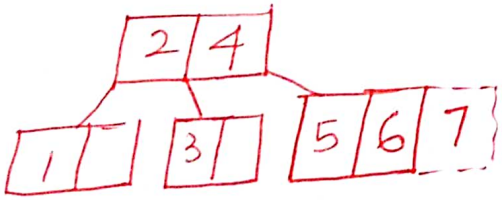


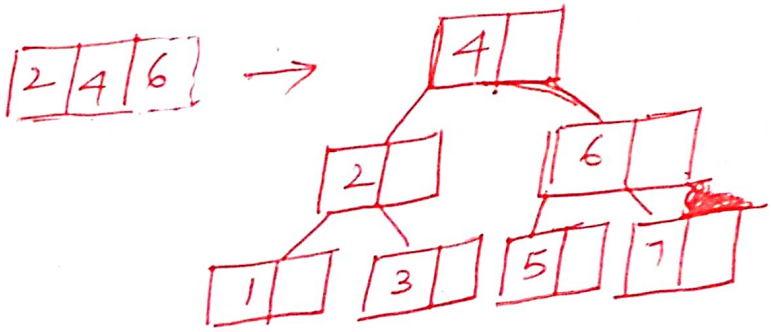










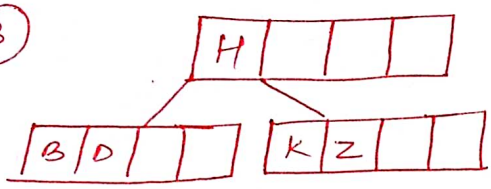


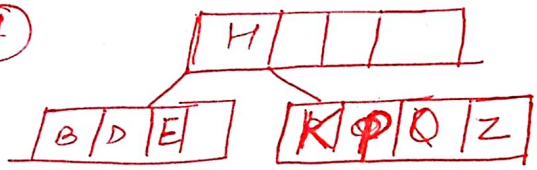
B-Tree of Order 5

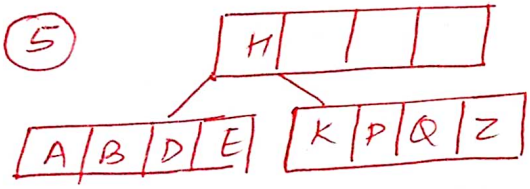
Data :- D, H, Z, K, B, P, Q, E, A, S, W, T, C, L, N, Y, M

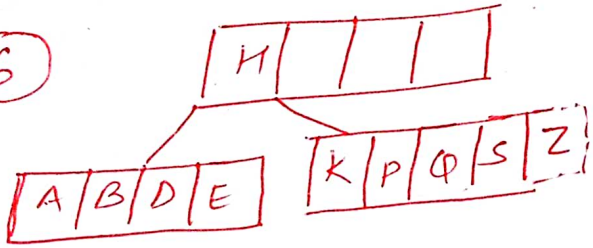
1. 

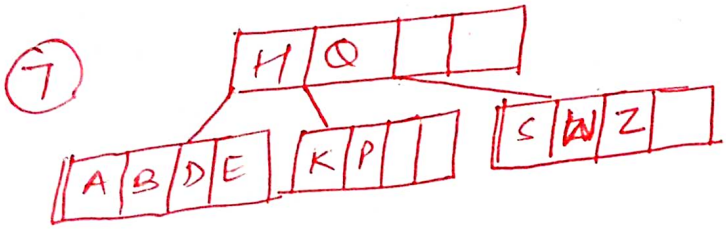
2. 

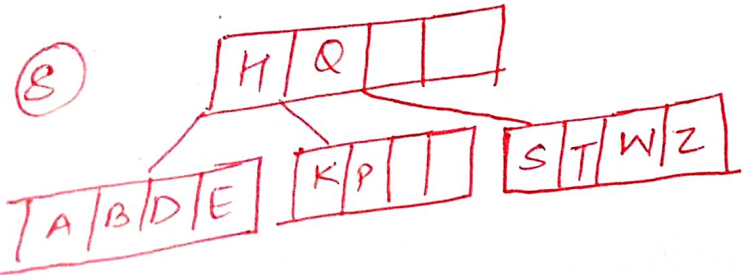
3. 

4. 

5. 

6. 

7. 

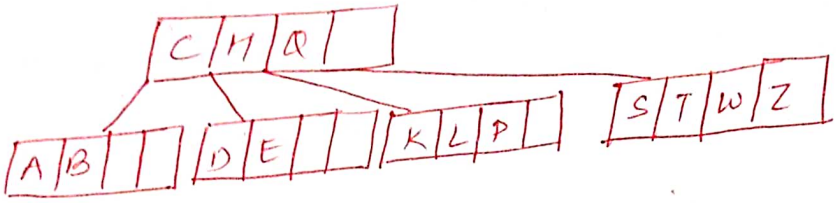
8. 

(9)

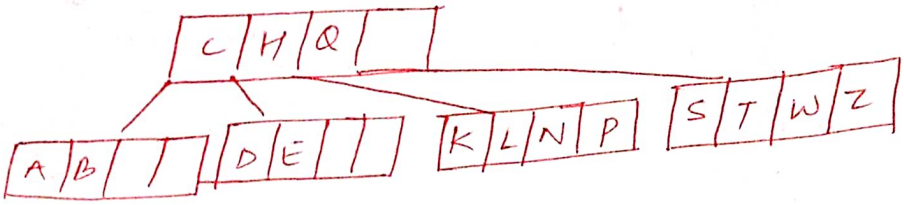

H Q
A B C D E
K P
S T W Z


C H Q
A B
D E
K P
S T W Z

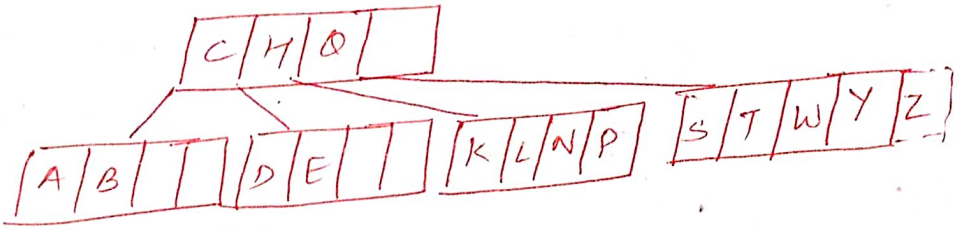
(10)

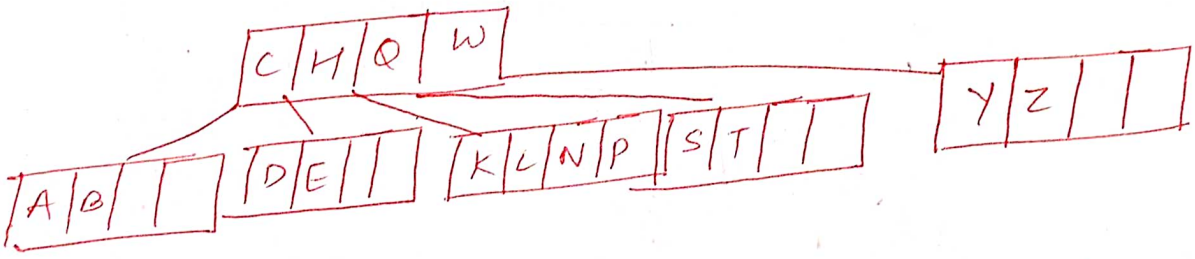

C H Q
A B
D E
K L P
S T W Z

(11)

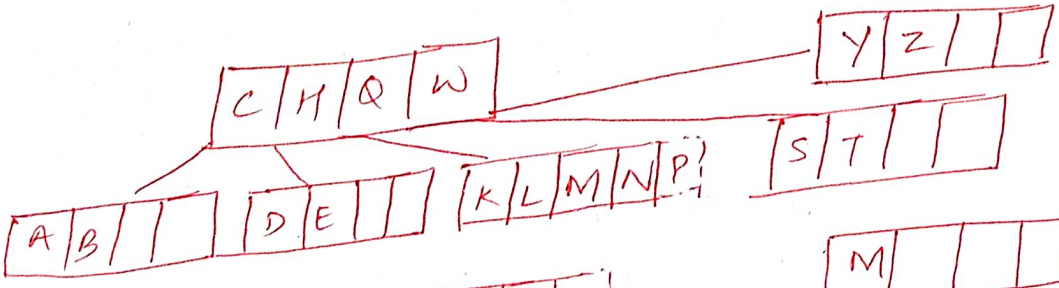

C H Q
A B
D E
K L N P
S T W Z

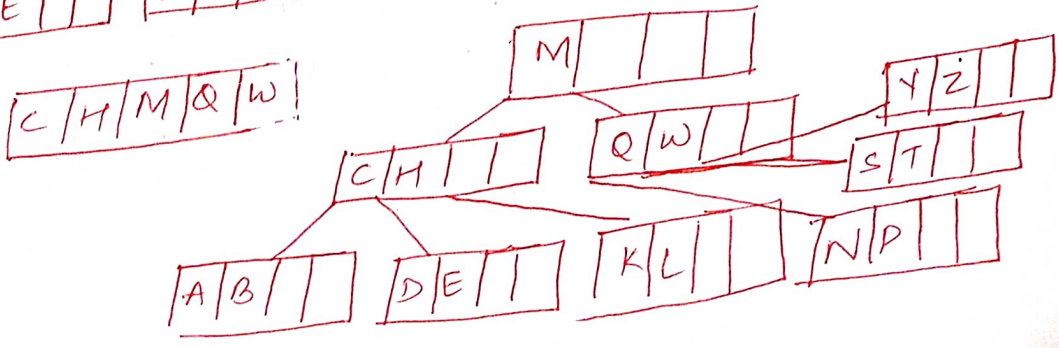
(12)


C H Q
A B
D E
K L N P
S T W Y Z


C H Q W
A B
D E
K L N P
S T
Y Z

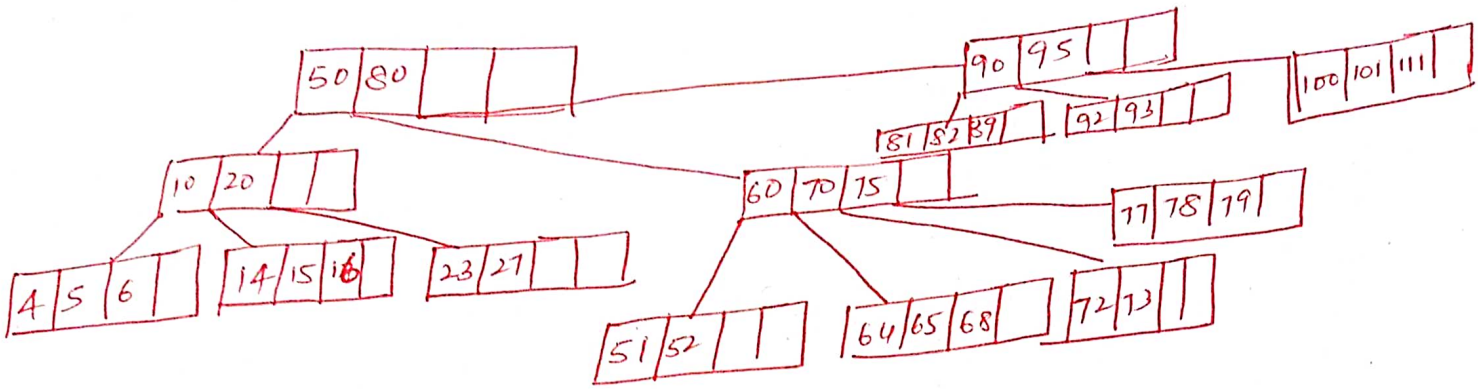
(13)


C H Q W
A B
D E
K L M N P
S T
Y Z


C H M Q W
M
C H
Q W
Y Z
S T
A B
D E
K L
N P

Deletion in B Tree

B-Tree of order 5



Min children = $\lceil \frac{m}{2} \rceil$ $\lceil \frac{5}{2} \rceil = 3$

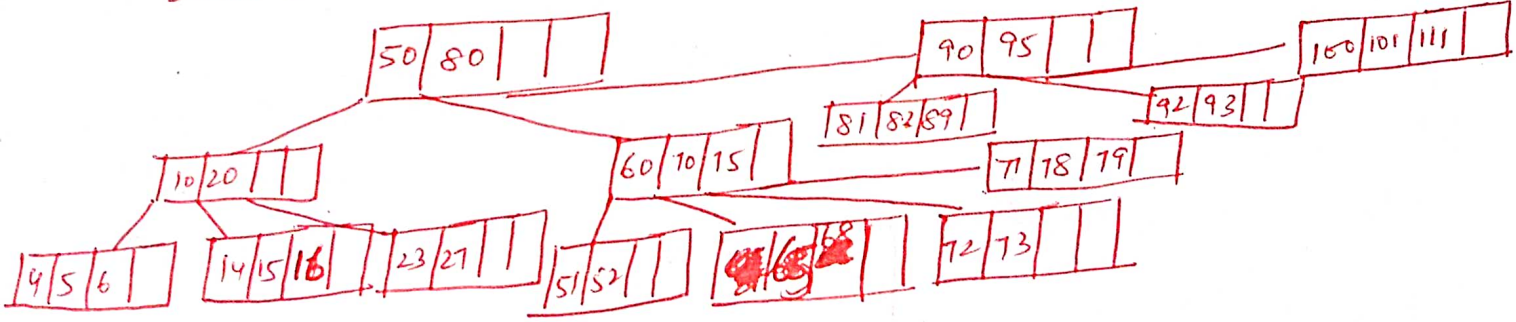
Max children = $m = 5$

Min keys = $\lceil \frac{m}{2} \rceil - 1 = 2$

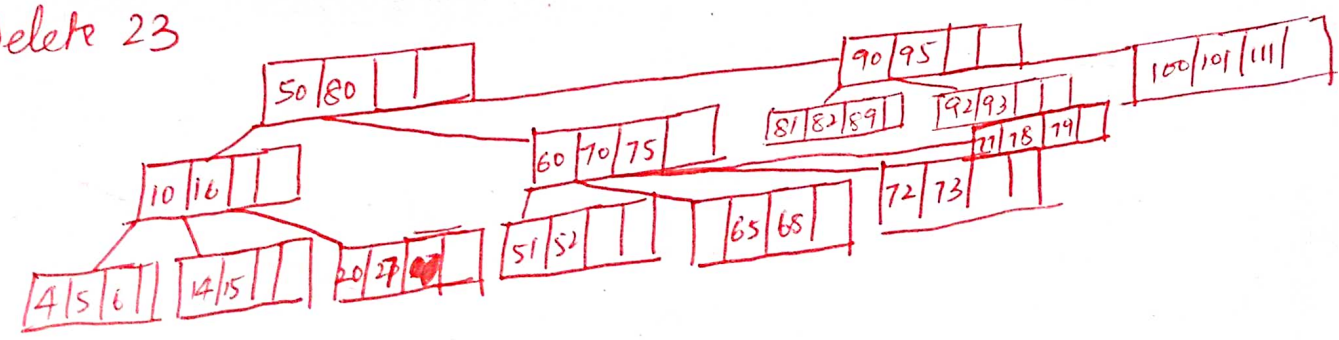
Max keys = $m - 1 = 4$

① Deletion of a leaf node

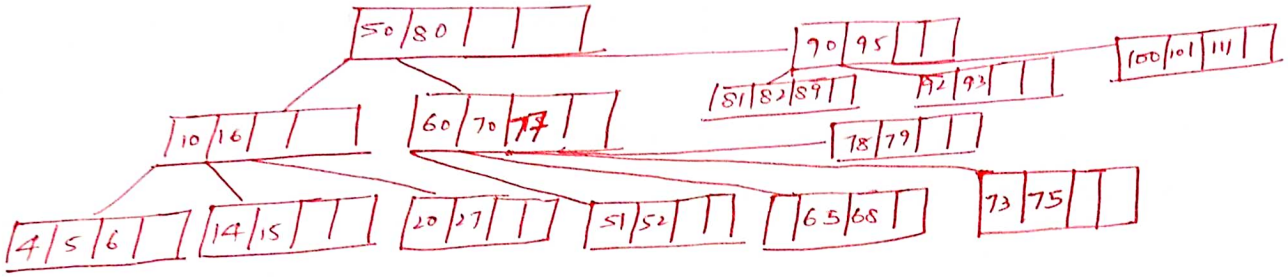
Delete 64



② Delete 23

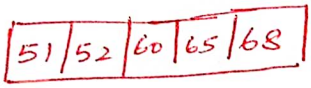


③ Delete 72

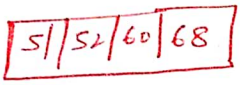


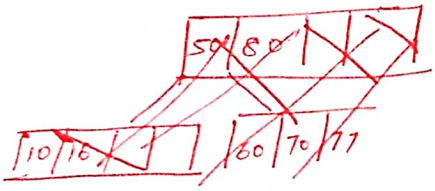
④ Delete 65

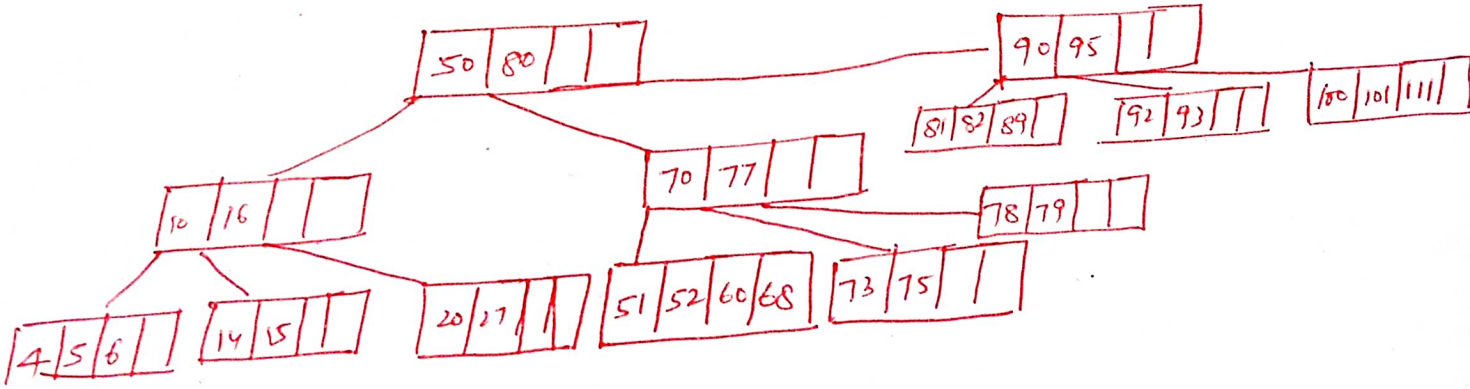
Here merging will take place because both its left and right sibling have equal no. of keys



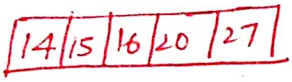
Now Delete 65

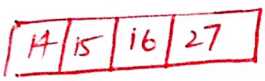


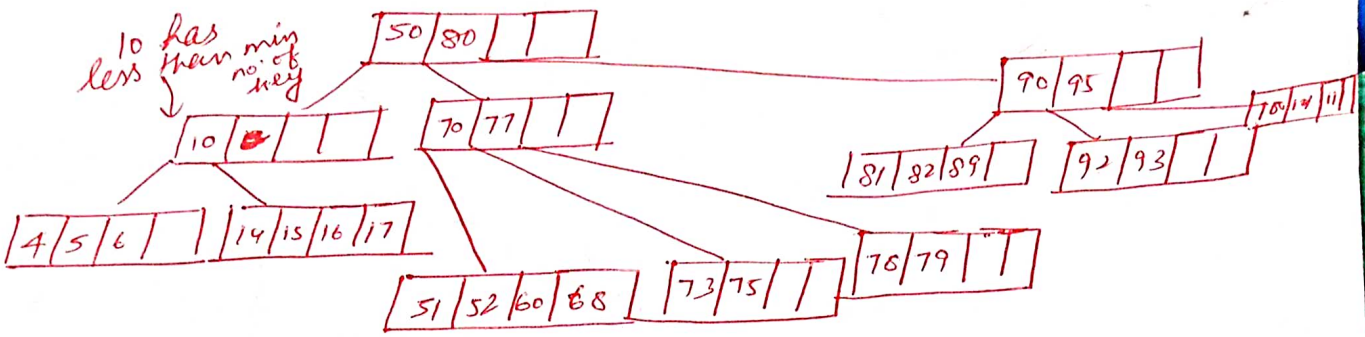


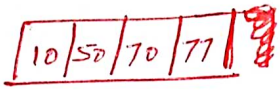


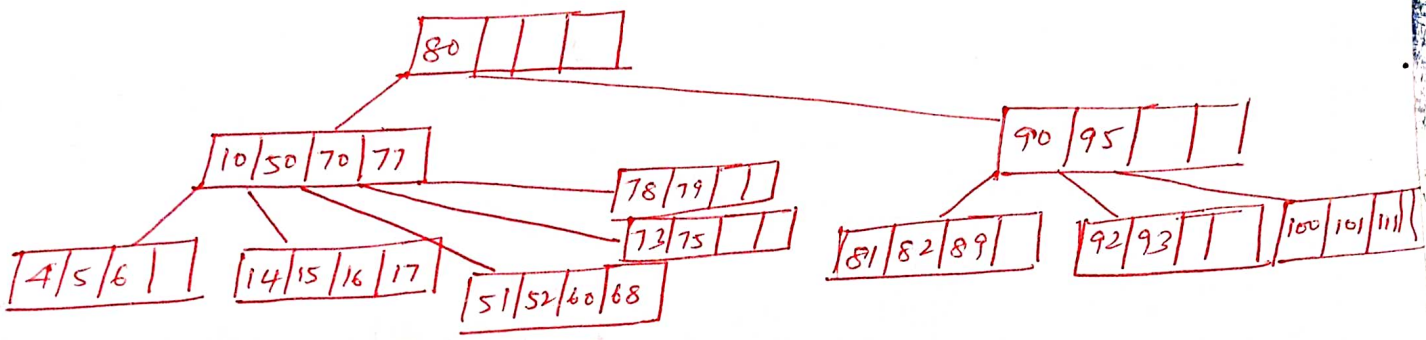
⑤ Delete 20



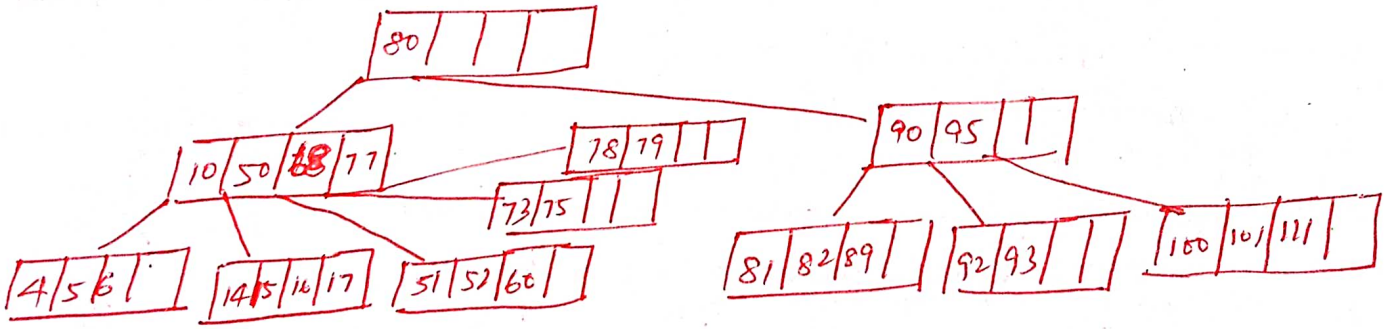




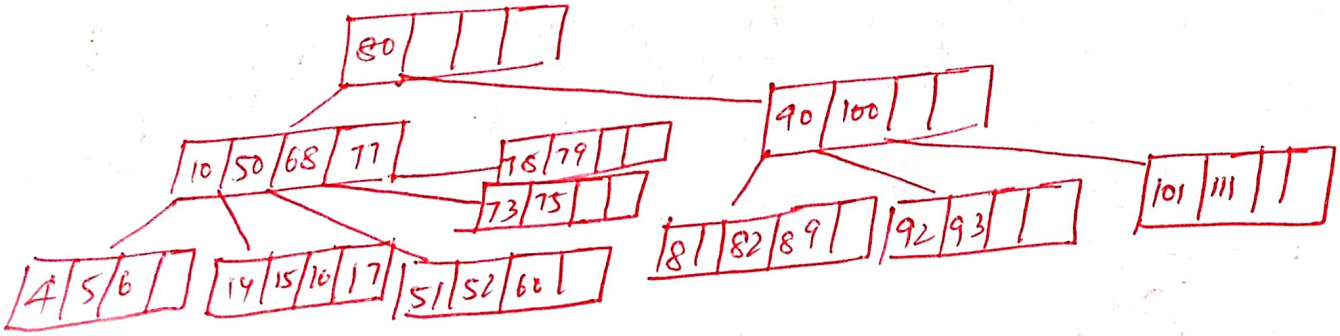




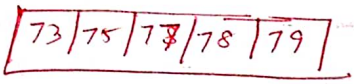
(6) Delete 70

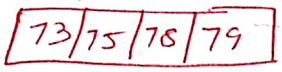


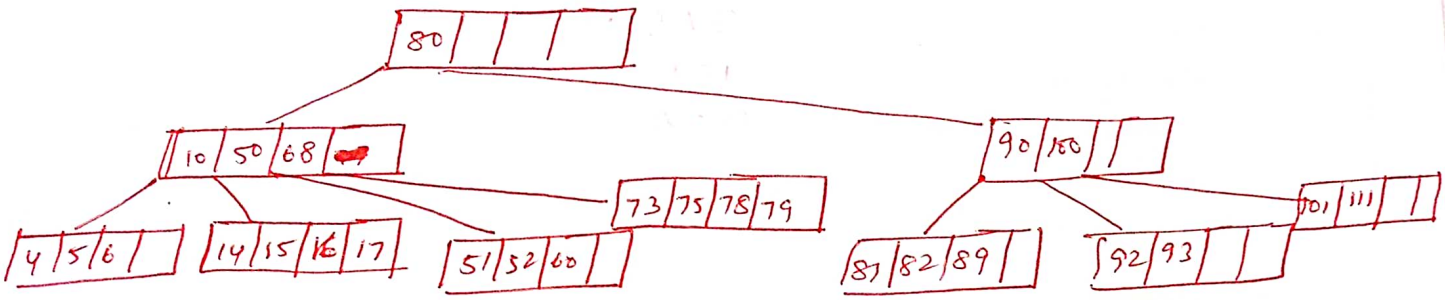
(7) Delete 95



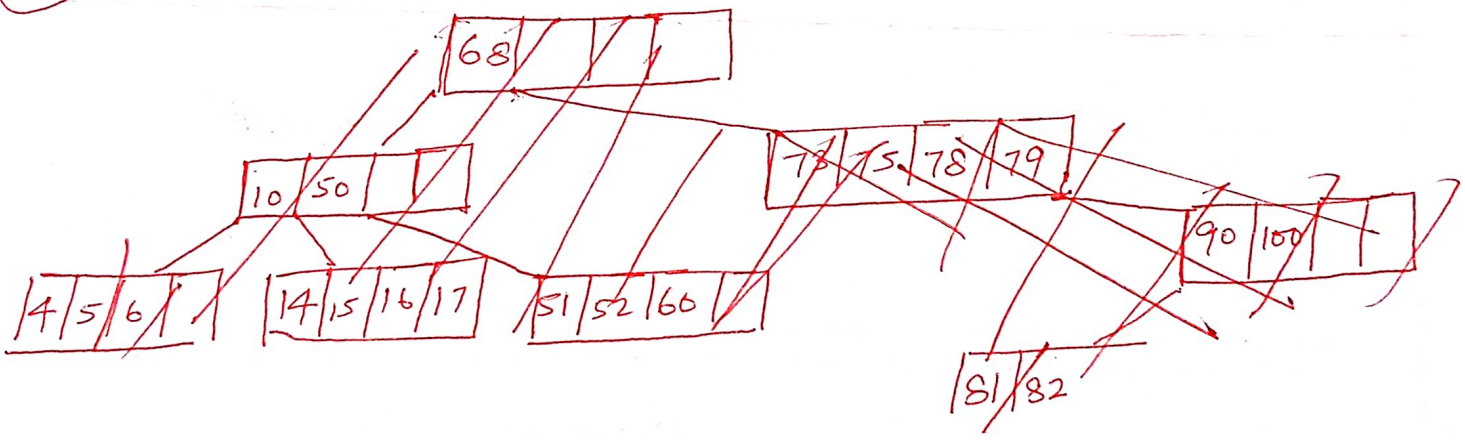
(8) Delete 77

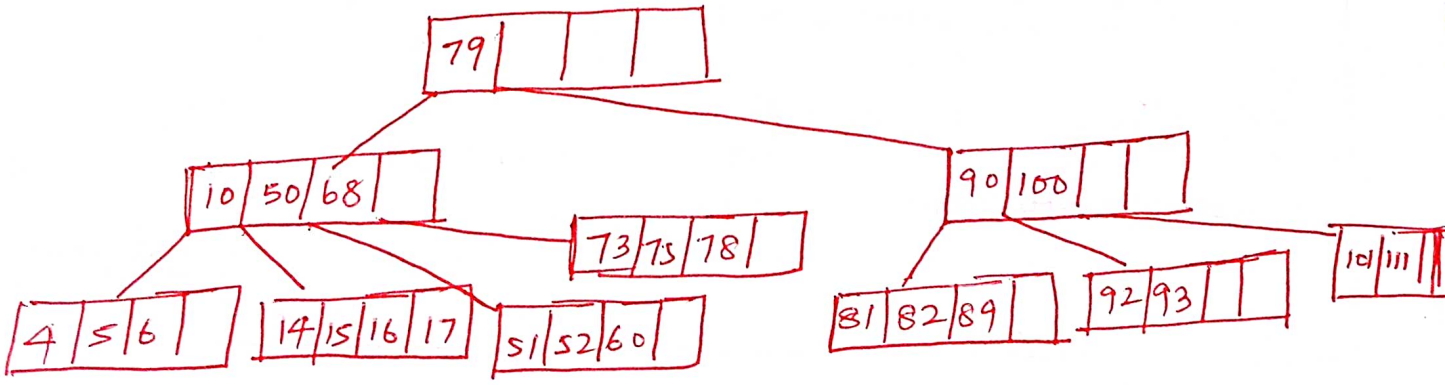




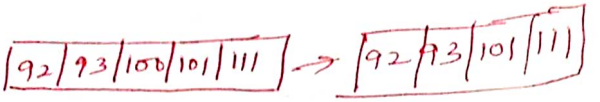


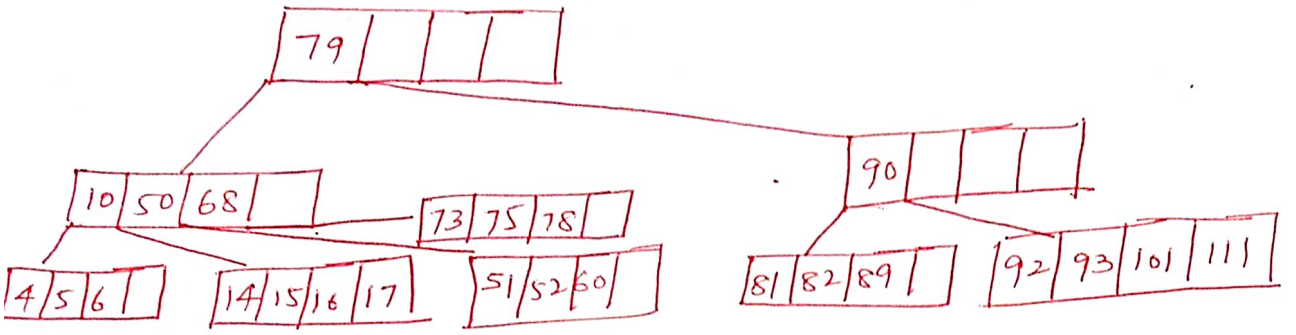
(9) Del 80

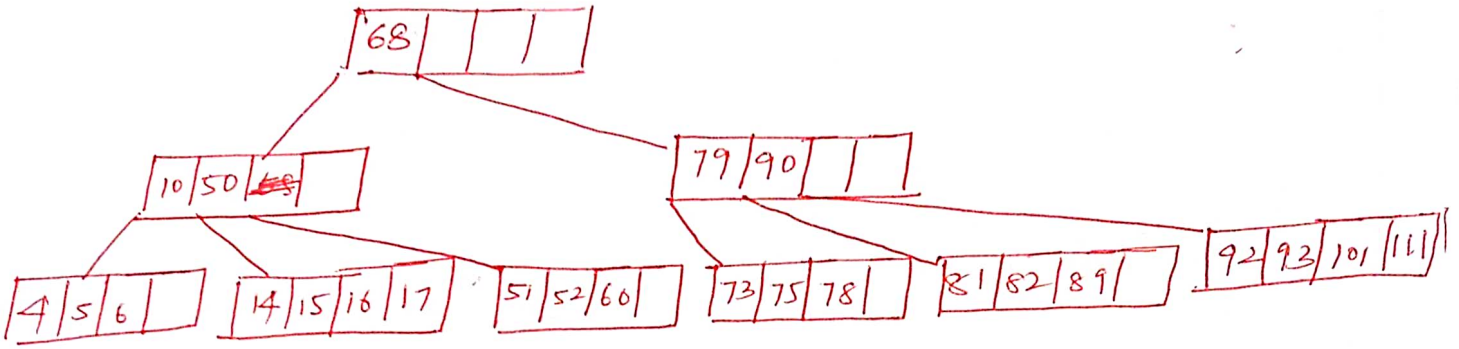




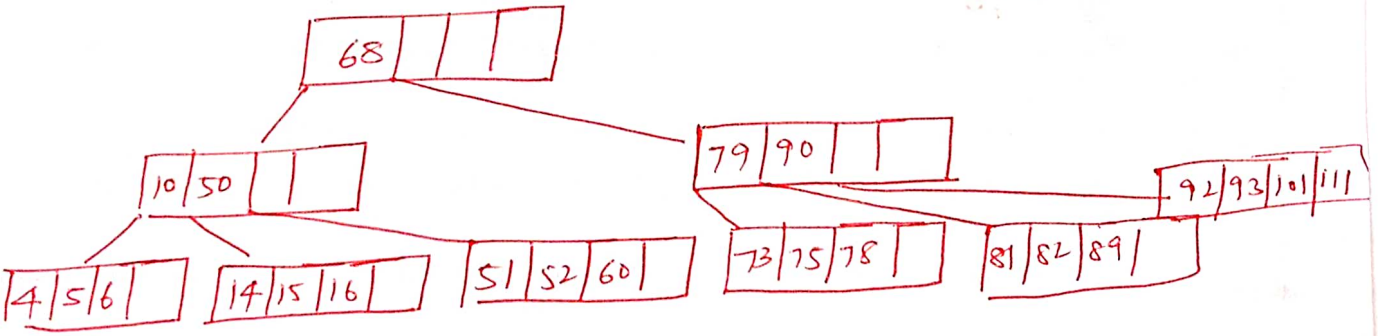
(10) Del 100



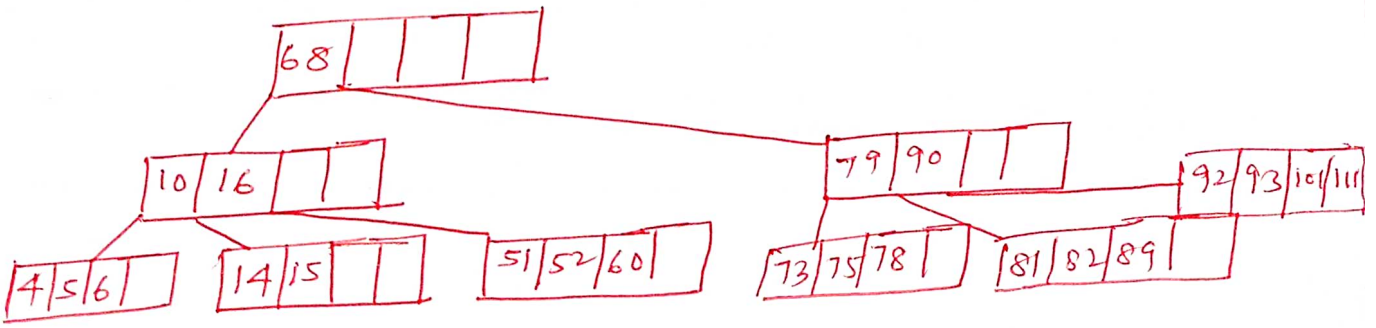




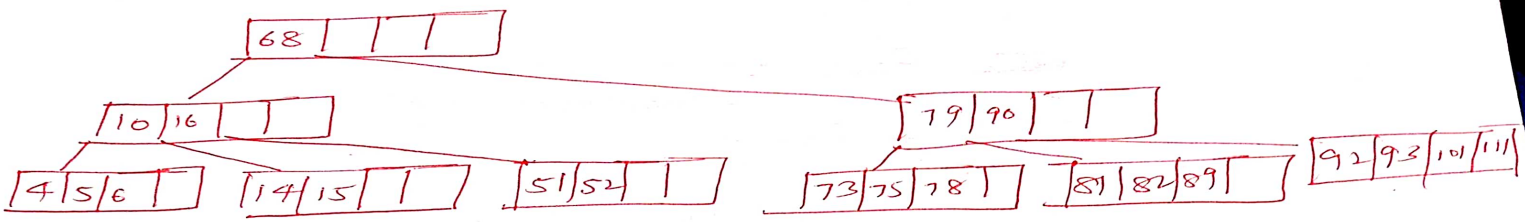
(11) Del ~~57~~



(12) Del 50



(13) Del 60



(14) Del 16 14 15 16 51 52 → 14 15 51 52