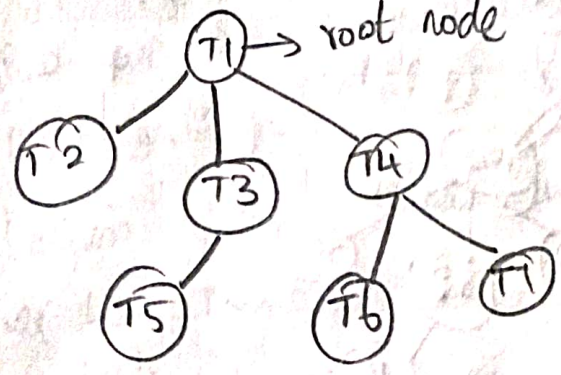
Unit 4
Tree structure

1) Tree ADT

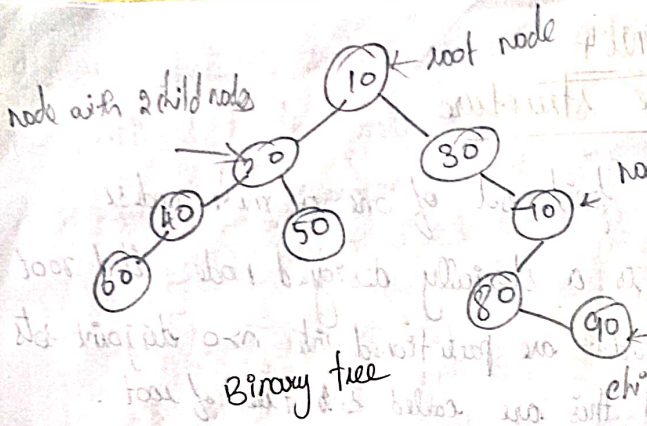
A tree is a finite set of one or more nodes such that i) There is a specially designed node called root.
* The remaining nodes are partitioned into $n>0$ disjoint sets $T1, T2 \ldots Tn$ and this are called sub trees of root.



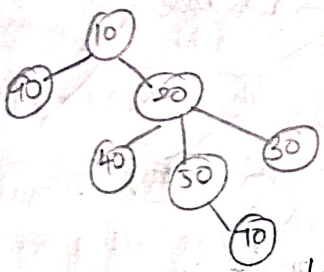
various operation performed on tree.
1. creation of a tree
2. Insertion of node in the tree as a child of desired node.
3. Deletion of any node from the tree.
4. modification of node value of the tree.
5. Searching particular node from the tree.

2) Binary tree ADT:

Binary tree is a finite set of nodes which is either empty or consists of a root & two disjoint binary tree called left subtree and right subtree.



Binary tree



Not a binary tree

Abstract Datatype BinT(root)

& Instances: Binary tree is a nonlinear data structure which contains every node except the leaf node at most two child nodes.

operations:

1) Insertion: This operation is used to insert the node in the binary tree. By inserting desired no: of nodes, the binary tree gets created.

2) Deletion: This operation is used to remove any node from the tree. Note that if root node is removed the tree becomes empty

}

(3) Tree traversals

Tree traversal means visiting each node exactly once.

* There are six ways to traverse a tree. For traversals we will use following notations.

L for moving to left child.
R for moving to right child.
D for parent node.

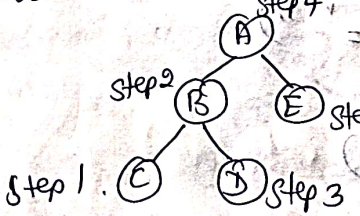
Thus with LRD we will have six combination such as LDR, LRD, DLR, DRL, RLD, RDL.

* From computation point of view we will consider only three combination as LDR, DLR & LRD. That is inorder, preorder & postorder respectively.

Inorder traversal:

The left node is visited, then parent node & then right node is visited.

eg:



Inorder traversal is CBDAE

Alg:

1. If tree is not empty then
   a) Traverse the left subtree in inorder.
   b) visit the root node
   c) Traverse the right subtree in inorder.

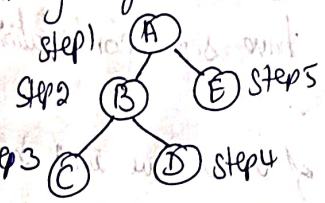
Recursive routine

if root :
    inorder (root.left)
    print (root.val, end = " ")
    inorder (root.right)

2) preorder Traversal:

The parent node or root node is visited first, then left node and finally right node will be visited.

eg :-

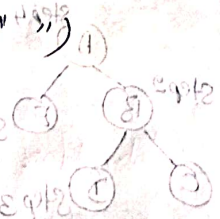


Preorder traversed
A, B, C, D, E.

Alg 1. If tree is not empty then
    a) visit the root node
    b) Traverse the left subtree in preorder
    c) Traverse the right subtree in preorder.

Recursive routine

if root : print (root.val, end = " ")
    preorder (root.left)
    preorder (root.right)



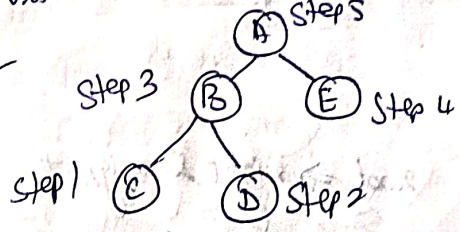
3) postorder Traversal:

The left node is visited first, then right node & finally parent node is visited.

Alg If tree is not empty then.
    a) Traverse the left subtree in postorder
    b) Traverse the right "

i) visit the root node.

eg :-



Postorder Traversal
C D B E A.

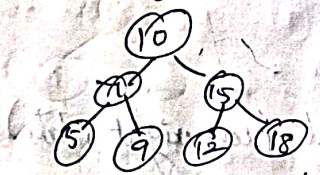
Recursive routine

def postorder (root):
    if root :
        postorder (root.left)
        postorder (root.right)
        print (root.val, end = " ").

(4) Binary search tree :

Binary search tree is a binary tree in which nodes are arranged in specific order.

* The value at left subtree are less than root node value. The values at right subtree are greater than root node.



* Its based on binary search alg.

operation on Binary search Tree.

1. Insertion of a node in a binary Tree :

alg: 1. create a new BST node & assign values to it
    2. Insert (root, data)
        1) if root == None,
            return the new node to calling fun.

ii) If root.val == data then,
return the root node

iii) If root ⇒ val < data.
call the insert fun with root ⇒ right & assign the return value in root ⇒ right

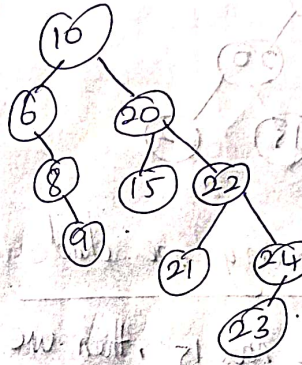
iv) if root ⇒ data > data.
call the insert fun with root ⇒ left & assign the return value in root ⇒ left.

3) finally, return the original root pointer to the calling fun.

Python code:

```
def insert(root,data):
    if root is none:
        return node(data)
    else:
        if root.val==data:
            return root
        elif root.val<data:
            root.right = insert(root.right,data)
        else:
            root.left = insert(root.left,data)
        return root
```

* Start comparing new node with each node of the tree.
* The node which is to be inserted is greater than the value of current node we move on to the right subbranch otherwise we move onto left subbranch.
* we want to insert 23

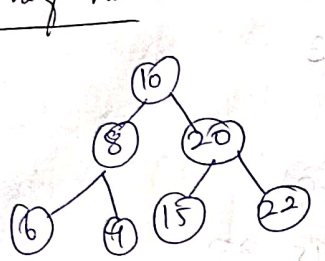


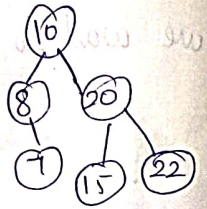
(D) Deleting of element from the binary tree:

deletion of any node from binary search tree there are 3 cases.

i) Deletion of leaf node
ii) Deletion of node having one child.
iii) Deletion of node having two child.

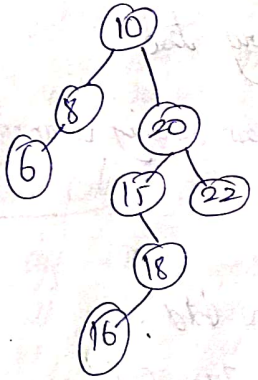
1) Deletion of leaf node: Delete 6

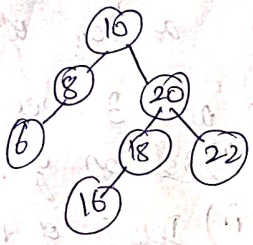




ii) Deletion of a node having one child:

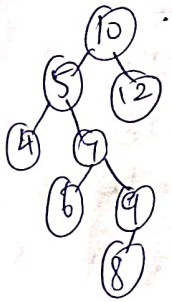
Delete node 15, then we will simply copy node 18 at place of 15 & then set node free.

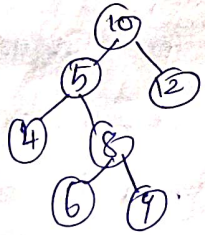




iii) The node having two child:

Delete node (9). The inorder successor will be simply copied at location of node 7. & will be copied to location 7. left ptr of 9 as NULL.

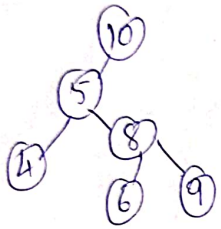




Python Code:

```
def delete (root, key) :
if root is None :
return root
if key < root.val
 root.left = delete (root.left, key)
elif (key > root.val):
root.right = delete(root.right, key)
else:
    if root.left is None:
        temp = root.right
        root = None
        return temp
    elif root.right is None:
        temp = root.left
        root = None
        return temp.
# Deleting a node with two children
temp = inorderSuccessor (root.right)
root.val = temp.val
root.right = delete (root.right, temp.val)
return root.
```

3- Searching a node from binary search tree:



* The search start from root node, if the value of key node is greater than current node, then search on right subbranch otherwise search on left subbranch.
* In the above example search 9. compare with 10. 9 is lesser so search on left subbranch. compared with 5, greater than 5, so go to right subbranch. compared with 8, but 9 is greater than 8. so search on right subbranch. finally 9 is on right of 8.

Python code:

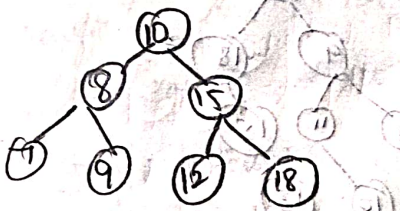
```
def search (root, key):
    if root is none or root.val == key:
        return root
    if root.val < key:
        return search (root.right, key)
    return search (root.left, key)
```

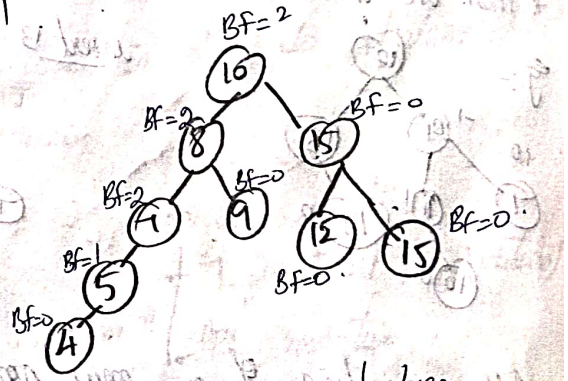
(5) AVL Tree

An empty tree is height balanced if T is a non empty binary tree with T_L & T_R as its left & right subtrees.
* The T is height balanced if and only if
  i) T_L & T_R are height balanced
  ii) $|h_L - h_R| <= 1$ where h_L & h_R are height of T_L & T_R.

Balancing of tree is obtained by balance factor of tree.

Definition of Balance factor:
The BF(T) of a node is binary tree is defined to be $h_L - h_R$ where h_L is a height of left subtree & h_R is height of right subtree.
* BF(T) is -1, 0 or +1.





Difference between AVL Tree & Binary search tree

AVL Tree	Binary Search Tree
1) AVL tree is height balanced search tree	It's not a Balanced tree
2) There is a req of computation of balanced factor for each	There is no concept of balanced factor.

node in an AVL tree.

3) Searching of any desired node is faster due to balancing of height

4) complex to impl.

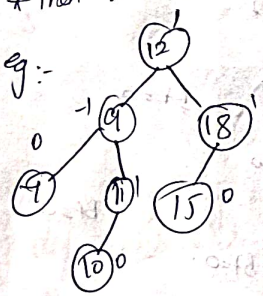
Searching is not efficient when there are large number of node in the tree

Simple to impl.

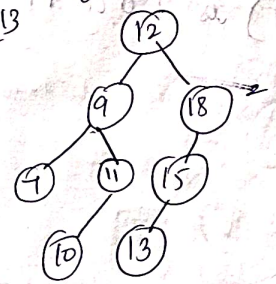
Representation of AVL Tree :

* AVL tree follows the property of binary search tree.
* After insertion of any node in an AVL tree if the balance factor of any node becomes other than -1, 0 or 1 then it said that AVL property is violated.
* Then we have to restore the destroyed balance condition.

eg :-

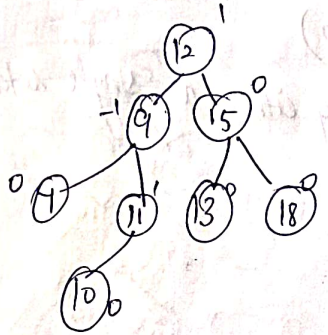


insert 13



After insertion of a new node if balance condition gets destroyed, then the nodes on that path needs to be readjusted. That is only affected subtree is to be rebalanced.

* The rebalancing should be in the way it should satisfy - AVL property.



Insertion :

There are four different cases when rebalancing is required after insertion of new node.

1) An insertion of new node into left subtree of left child (LL)
2) An insertion of new node into right subtree of left child (LR)
3) An insertion of new node into left subtree of right child (RL)
4) An insertion of new node into right subtree of right child (RR)

The modification done on AVL tree in order to rebalance it is called rotation of AVL tree.

- Single rotation
  - LL rotation
  - RR Rotation
- Double rotation
  - Left - Right (LR rotation)
  - RL rotation

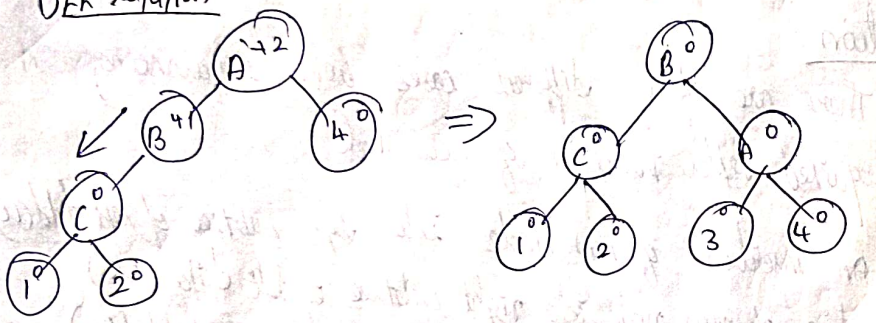
Insertion alg :

1) Insert a new node as new leaf just as in ordinary binary search tree.
2) Now trace the path from insertion pt towards root. For each node 'n' encountered, check if height of left(n) & right(n) diff by at most 1.

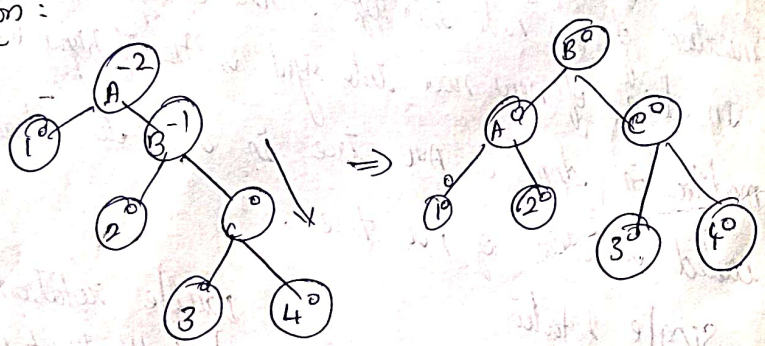
a) if yes, move towards parent(n)
b) otherwise restructure by doing either a single rotation or double rotation.

Different rotation in AVL Tree:

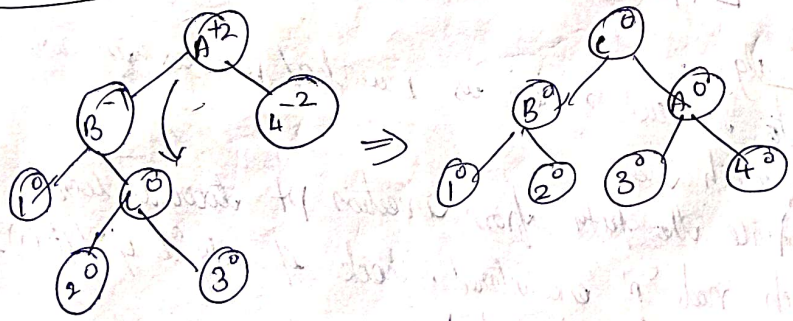
LL rotation



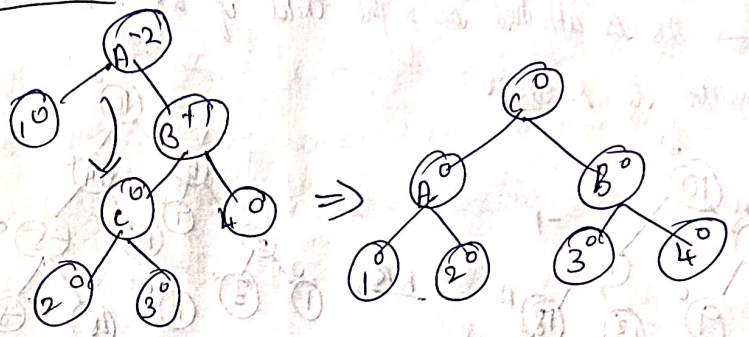
RR rotation:



LR rotation:

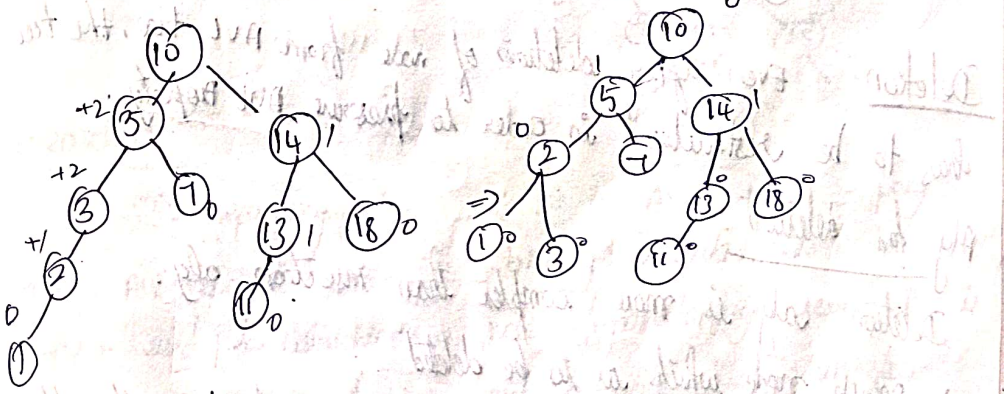


RL Rotation:

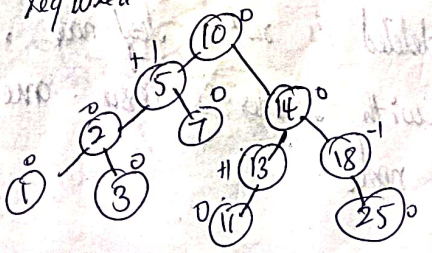


eg:- consider AVL Tree as given below:
we have to insert 1, 25, 28, 12 in this tree

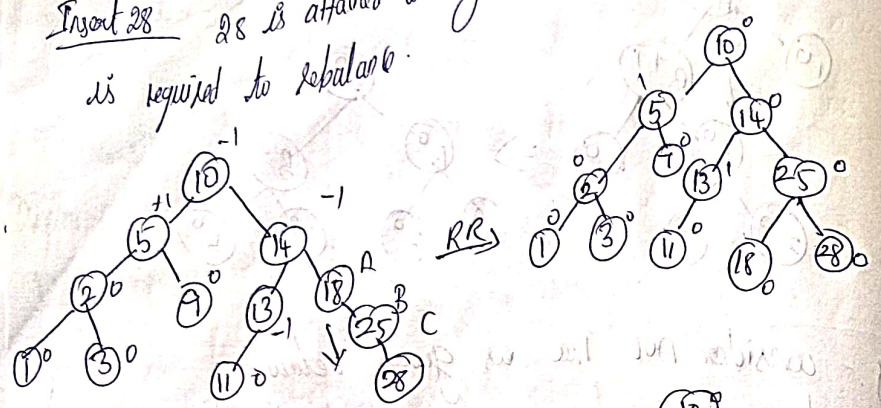
Insert 1 To insert node 1 we have to attach to left of 2. This is unbalanced tree. we have to apply LL rotation to preserve AVL property.



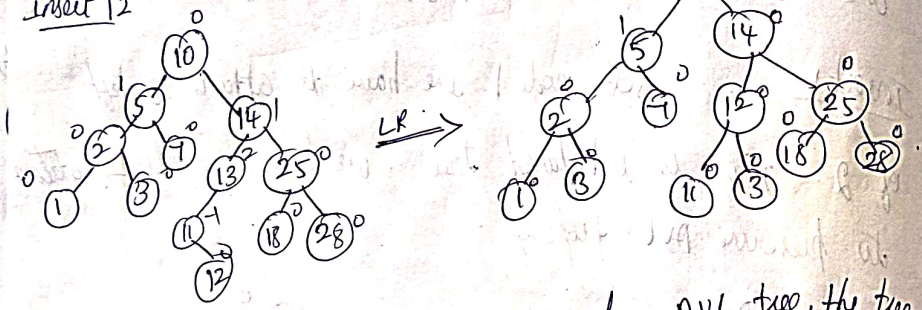
Now insert 25:
25 is inserted at the right of 18. No balancing is required as entire tree is balanced.



Insert 28 — 28 is attached as right child of 25. RR rotation is required to rebalance.



Insert 12



Deletion :- Even after deletion of node from AVL tree, the tree has to be restructured in order to preserve AVL Property.

Alg for deletion :-

Deletion alg is more complex than insertion alg.

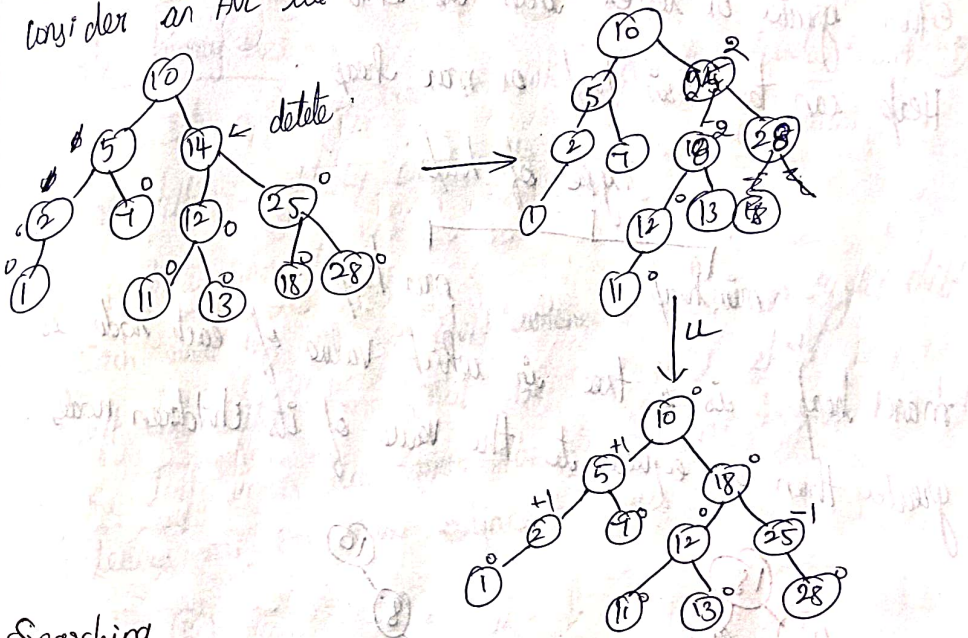
1- search node which as to be deleted.

2. a) If leaf node to be deleted simply make it null after delete.

b) If node to be deleted is not a leaf node, then the node must be swapped with inorder successor. once swapped, we can remove this node.

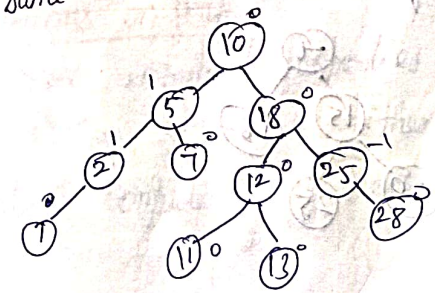
3) now we have to traverse back up the path towards root, checking the balance factor of every node along the path. If we encounter unbalancing in some subtree then balance the subtree using appropriate single or double rotation.

Consider an AVL tree



Searching

The searching of a node in an AVL tree is very simple. AVL tree is basically binary search tree, the algorithm used for searching a node from binary search tree is the same one is used to search a node from AVL tree.



⑥ Heaps

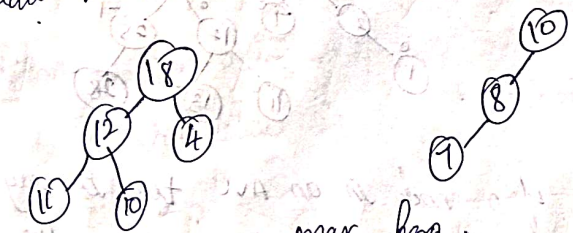
Heap is a complete binary tree or a almost complete binary tree in which every parent node be either greater or lesser than its child nodes.

Heap can be min heap or max heap.

Type of heap.

min heap max heap

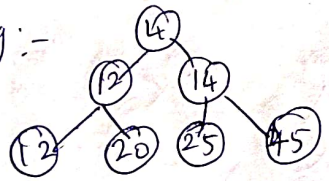
max heap is a tree in which value of each node is greater than or equal to the value of its childeren nodes.

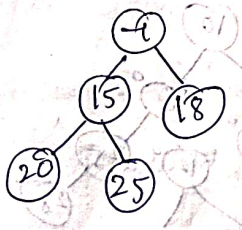


max heap.

Min heap is a tree in which value of each node is less than or equal to value of the childeren node.

eg:-





8 4 4 2 18

* parent being greater or lesser in help is called parental property. Thus heap has two important property.

i) It should be complete binary tree

ii) it should satisfy parental property.

Complete binary tree:

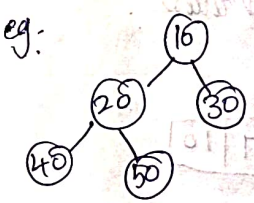
Its a binary tree in which all the levels of binary tree are filled completely except the lowest level nodes

Almost complete binary tree:

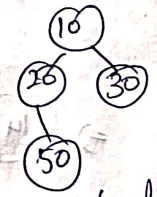
i) Each node has a left child whenever it has a right child. for left child there may not be a right child.

ii) The leaf in a tree must be present at height h or h-1. the leaves are on two adjacent levels.

eg:



almost complete binary Tree.

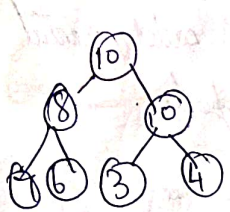


Not almost binary Tree.

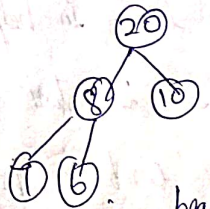
Operations on Heap:

Some important properties.

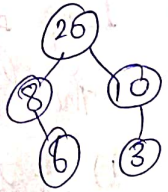
1) There should be either complete binary tree or almost complete binary tree.



is a heap



is a heap



not a heap

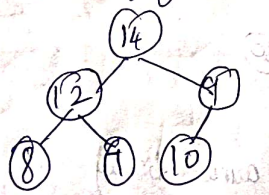
Insertion of element:

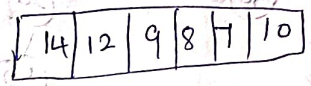
algorithm for inserting an element:

1. Insert element at last position in heap.
2. Compare with its parent, swap the two nodes if parental dominance condition gets violated.

   parental dominance = parent is greater than its children.
3. Continue comparing the new element with node up, each time by satisfying parental dominance condition.

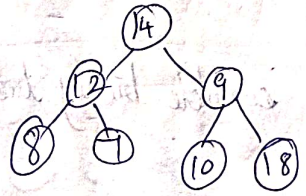
eg:-





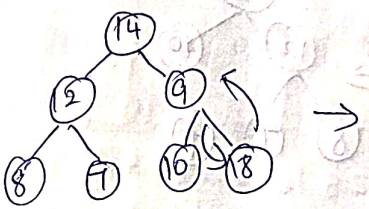
array rep of heap

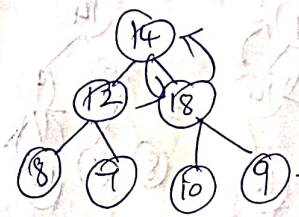
insert 18.

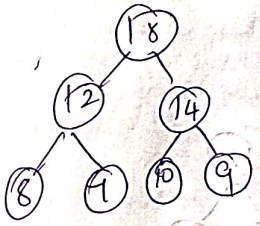


← attach this node at a last node.

parental dominance property is violated (9<18) we swap two values.







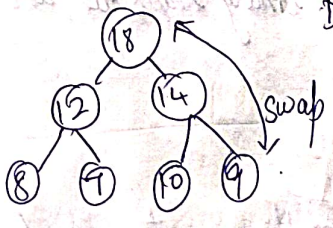
In top down approach the heap is constructed from top to bottom scanning of a tree, each time checking parent dominance property.

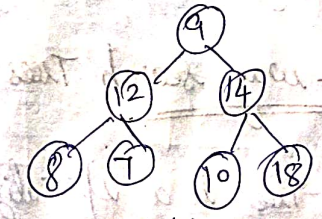
Deletion of Element:

Algorithm for deletion of elements from heap.

1. Exchange the root with last leaf.
2. Decrease the heap size by 1
3. Heapify the smaller tree using bottom up construction algorithm.

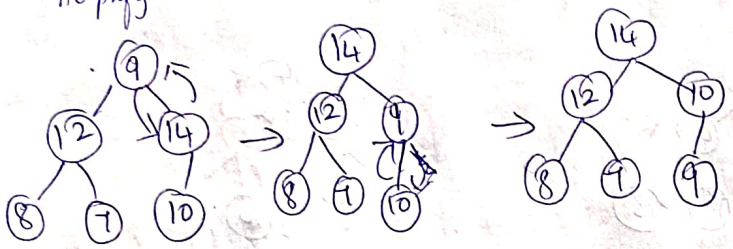
Delete 18 first swap with last node.





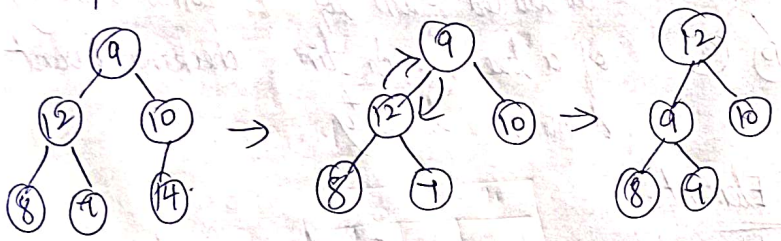
Now delete 18.

Heapify



Delete 14

Swap with last node



Application of Binary Heaps:

1. Binary heaps are useful data structures for sorting the elements using heapsort method.
2. Binary heaps are used to find the shortest path using Dijkstra's shortest path alg.
3. For finding k^{th} smallest element binary heaps are used.

(1) Multi-way search Trees:

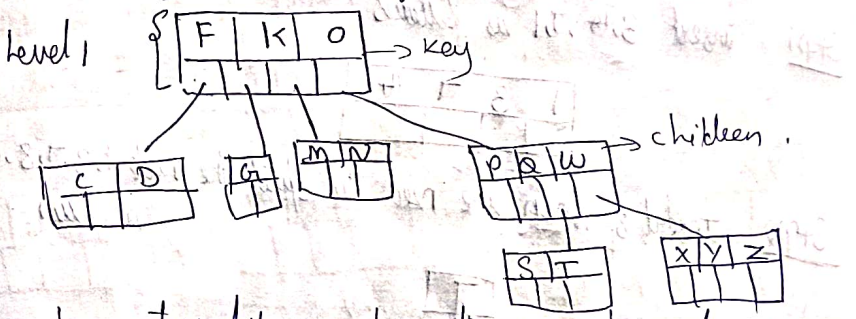
B-Tree:

B-Tree is a specialized multiway tree used to store the records in a disk. There are number of subtrees to each node. So that the height of the tree is relatively small. So that only small number of nodes must to read from disk to retrive an item. The goal is to get fast access of the data.

multi way search

B-Tree is a multiway search tree of order m is an ordered tree where each node has at most m children. If there are n number of children in a node then (n-1) is the number of keys in the node.

eg:- Following is a tree of order 4.



From above tree following observation can be made.

1. The node which has n children poses (n-1) keys
2. The keys in each node are in ascending order.
3. For every node Node.child[i] has only keys which are less than Node.key[i]. Similarly, node.child[i] has only keys which are greater than node.key[i].

Properties:

Rule 1: All the leaf nodes are on the bottom level.

Rule 2: The root node should have at least 2 children.

Rule 3: All the internal node except root node have at

least ceil (m/2) nonempty children.

Rule4 : Each leaf node must contain at least ceil (m/2)-1 keys.

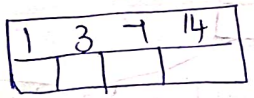
Insertion :

Construct B tree with order 5 for following numbers :

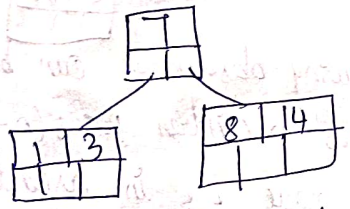
3, 14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26, 4, 16, 18, 24, 25, 19.

The order 5 means at the most 4 keys are allowed.
The internal node should have at least 3 non empty children and each leaf node must contain atleast 2 keys.

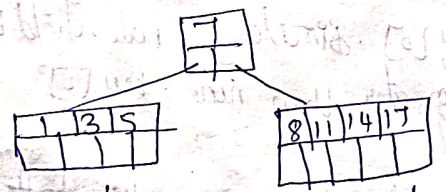
Step1 insert 3, 14, 7, 1 as follows :



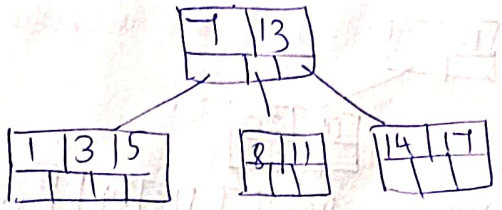
Step2 Insert 8, then we need to split the node 1, 3, 7, 8, 14 at medium.



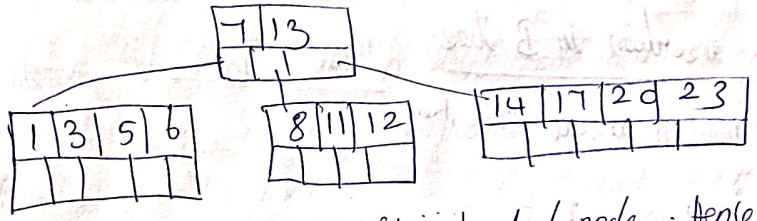
Step3 : Insert 5, 11, 17 which can be easily inserted in B tree.



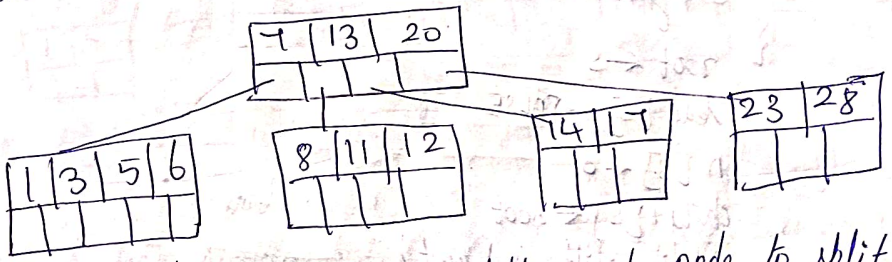
Step4 Now insert 13 then leaf node will have 5 keys. which is not allowed. Hence 8, 11, 13, 14, 17 is split & medium node 13 is moved up.



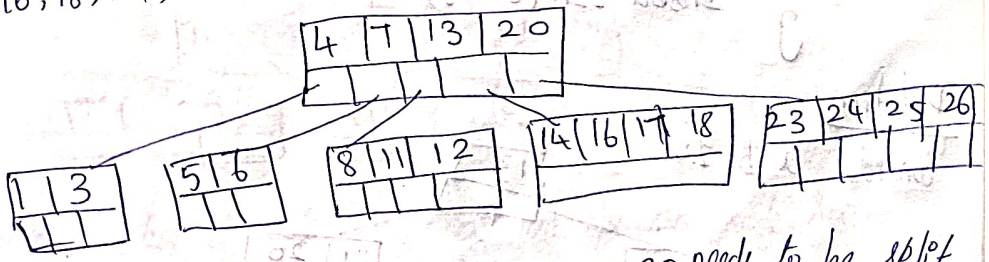
Step5 Now insert 6, 23, 12, 20 without any split.



Step6 : 26 is inserted at rightmost leaf node. Hence 14, 17, 20, 23, 26 the node is split & 20 will be moved up.

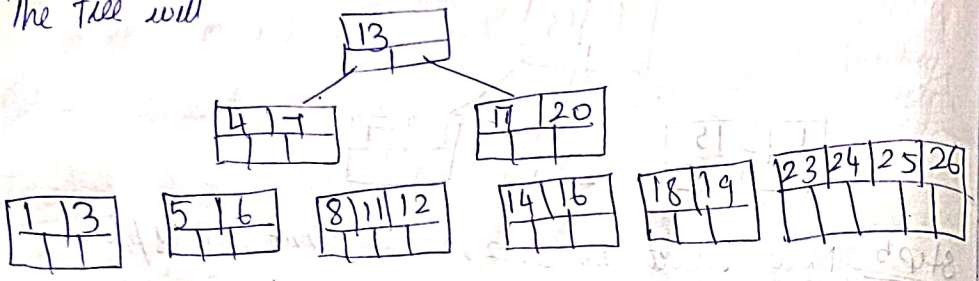


Step7 Insertion of 4 causes left most node to split. The 1, 3, 4, 5, 6 causes key 4 to move up. Then insert 16, 18, 24, 25.



Step8 Finally insert 19, Then 4, 7, 13, 19, 20 needs to be split The median 13 will be moved up to form a root node.

The tree will

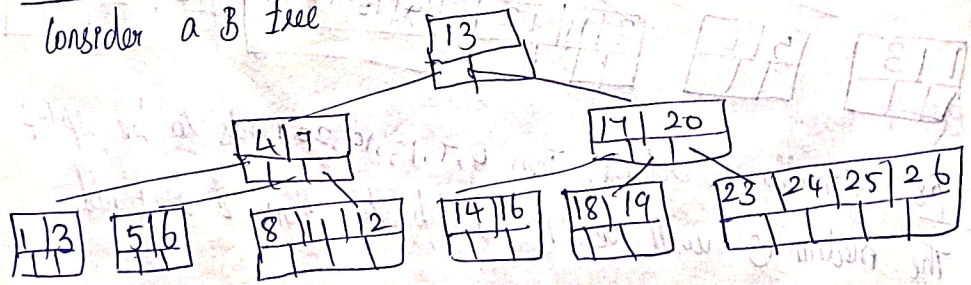


Alg for insertion in B-tree:

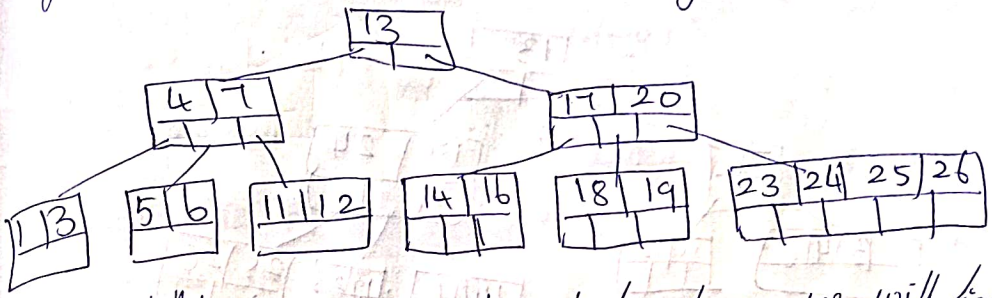
```
Algorithm insert (root, Key)
{
    temp ← root
    if (n[temp] = 2t-1) then
      S ← get_node()
    {
      root ← S
      leaf [S] ← FALSE
      n [S] ← 0
      child i [S] ← root
      Split_child (S, 1, root)
      Insert_In (S, Key)
    }
    else
      Insert_In (S, Key)
}
```

Deletion:

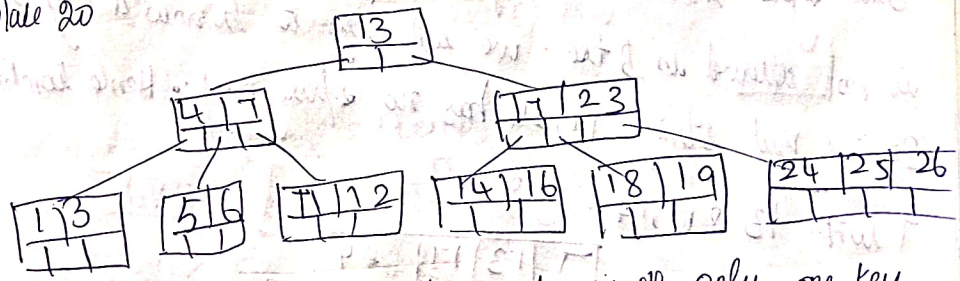
Consider a B tree



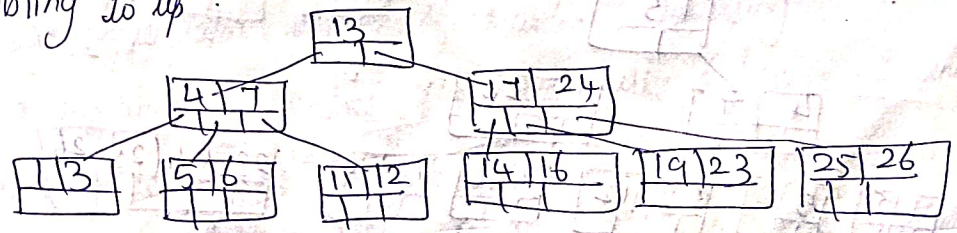
If we want to delete 8 then its very simple.



Now delete 20, 20 is not a leaf node so we will find successor which is 23. Hence 23 will be moved up to replace 20

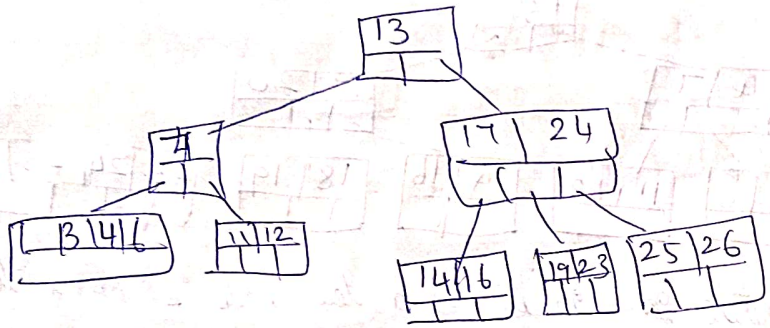


Next delete 18. will cause the node with only one key, which is not desired in B-tree of order 5. The sibling node is immediately right has a extra key. so we can borrow a key from parent and move spare key of sibling to up.

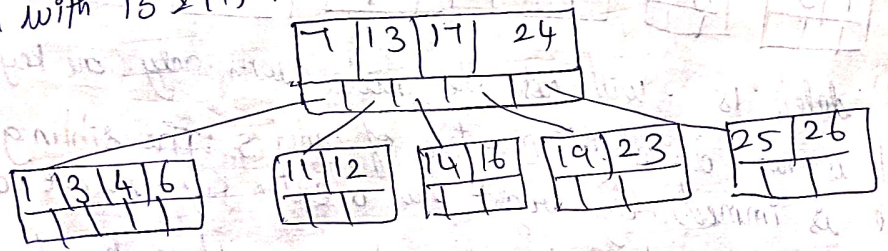


Delete 5, but its not easy, because the leaf node has no extra keys to immediate left or right. In such case combine the node with siblings. Remove 5 & combine 6 with node 1, 3.

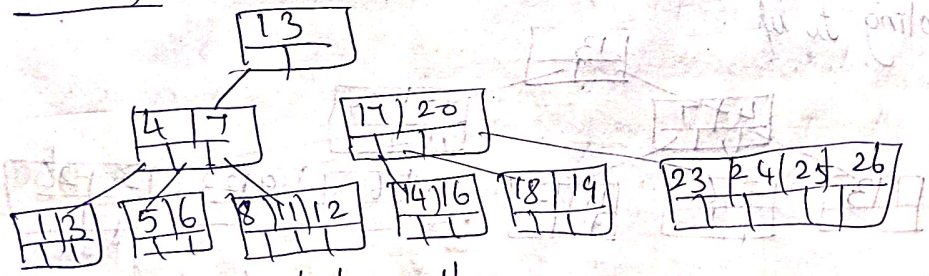
To make tree balanced we have to move parent key down.



But again internal node of 4 contain only one key which is not allowed in B tree. we will try to borrow a key from Sibling But sibling 17, 24 has no spare key. Hence combine 4 with 13 & 17, 14.



Searching



if we want to find 11 then:

i) $11 < 13$ Hence search left node
ii) $11 > 7$ move rightmost node
iii) $11 > 8$ move in second block
iv) node 11 is found.

The running time of search operation depends upon the height of the tree. It is $O(\log n)$.

Alg

```
Algorithm Search (temp, target)
{
  i ← 1
  while i ← i+1
    if ((i ≤ n[temp]) AND (target == key[temp]))
    return (temp, i)
    if (leaf[temp] == TRUE) then
    return NULL
    else READ (C[temp])
    return Search ((C, [temp], target)
}
```

B+tree

In B+tree the traversing of nodes is done in inorder manner which is time consuming. we want such a data structure of B-tree which will allow us to access data sequentially, instead of inorder traversing.

Definition: In B+tree from leaf nodes reference to any other node can be possible. The leaves in B+tree form a linked list which is useful in scanning the nodes sequentially.

* The insertion & deletion operations are similar to B-trees.
* from leaf node only any key can be accessed of entire tree. There is no need to traverse the tree in inorder fashion.

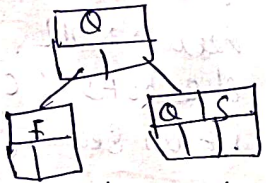
eg :- Construct B+tree for F, S, Q, K, C, L, H, T, V, W, M, P.

* Constructing B+ tree is similar to B tree but the only difference here is that, the parent node also appear in the leaf nodes. we will build B+ tree for order 5.

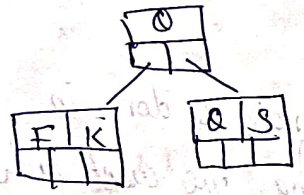
* The order 3 means at the most 2 keys are allowed.

Step1 Insert F & S

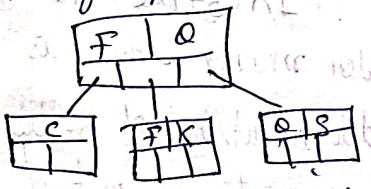
Step2 Insert Q, then seq will be F, Q, S. Then Q will go up.



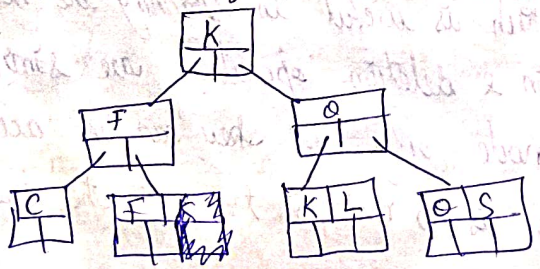
Step 3 insert K



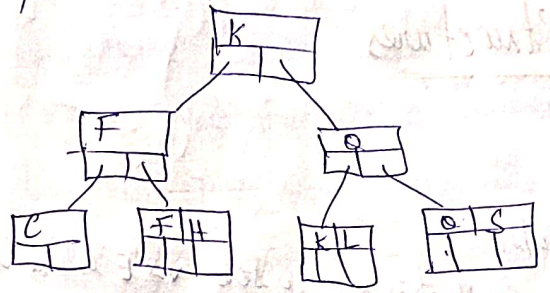
Step 4 Insert C, But this will create a seq C, F, K. This will split & F go up.



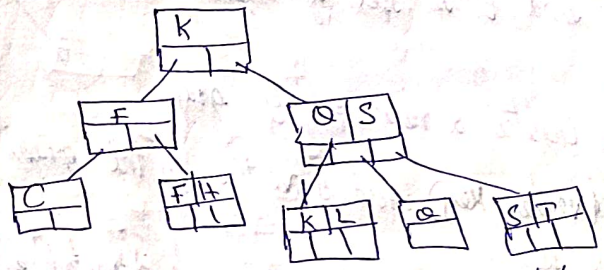
Step 5 Insert L. This will make the seq F, K, L, Again this sequence will split up & K will go up. Then the seq F, K, Q will split up & K go up.



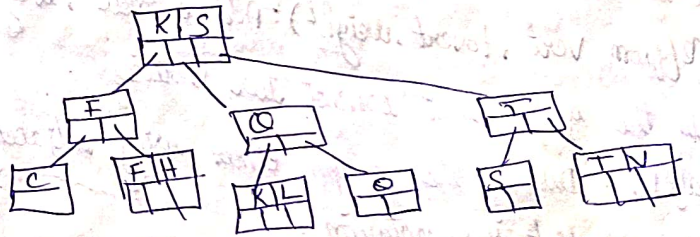
Step6 Insert H. This will make the seq F, H, K. The seq will split up & H will go up.



Step7 Insert T : Sequence Q, S, T will split up. The S will go up.



Step8 Insert V. The sequence S, T, V will split up. T will go up again the seq Q, S, T will split up & S will go up.



Step9 Insert W. The seq become T, V, W, Then V goes up.