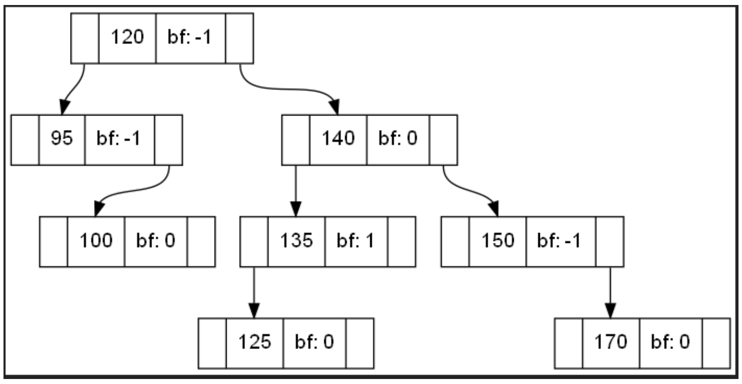
AVL_TREE *214101019*

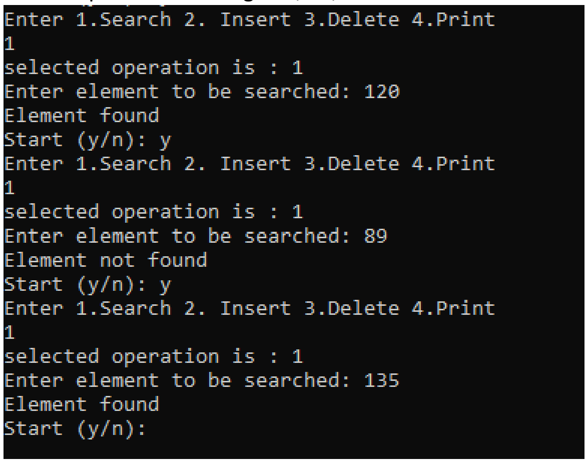
❖ Balance Factor = (height (left subtree) – height (right subtree))

AVL_Search(int k):

- This function searches whether element k is present in the tree or not.
- If element is present then it will return true
- Else false.
- Examples:
  1. Element exists
     Original Tree is:



♦ Output for searching 120, 89, 135:



```
Enter 1.Search 2. Insert 3.Delete 4.Print
1
selected operation is : 1
Enter element to be searched: 120
Element found
Start (y/n): y
Enter 1.Search 2. Insert 3.Delete 4.Print
1
selected operation is : 1
Enter element to be searched: 89
Element not found
Start (y/n): y
Enter 1.Search 2. Insert 3.Delete 4.Print
1
selected operation is : 1
Enter element to be searched: 135
Element found
Start (y/n):
```

AVL_Insert(int k):

- This function performs following steps:

1. First check if element exists in tree or not .
2. If element exists then terminate
3. Else following steps are performed

   a. Here normal BST insertion procedure can be followed however tree may become unbalanced so modifications are required.
   b. Whenever we insert a node ‘x’ as child of already existing node ‘y’ then following values of balance factor(bf) is possible for node y -1,0,+1.

c. Whenever we insert a node x then rebalancing may be required under following cases possible:
   - If bf of y = 1 (if x inserted in left subtree resulting in increase in height)
   - If bf of y = -1(if x inserted in right subtree resulting in increase in height)

d. No rebalancing required if bf of y = 0 as even if height increases then bf will become +1 or -1.

e. Here we maintain following pointers:
   - p = it will move down the tree to the location where the node is to be inserted
   - s = points to the place where rebalancing is required
   - t = parent of node

f. So we follow the simple path to find the place where the node k needs to be inserted this becomes p

g. During this process we check if node has bf = +1 or -1 then we store it in s and its parent in t.

h. Once we reach to the location where node is to be inserted we insert the node as left or right child (depending on the value)

i. We use variable 'a' which will be useful for rotations ; if value of s_key > k then a = 1 else a = -1 and start from p = left child or p = right child of s respectively till node k and perform following steps:
   - If p_key > k; set bf =1;
   - Else set bf = -1

j. If s_bf = 0 ; set s_bf = a (case node inserted as child of head node); terminate

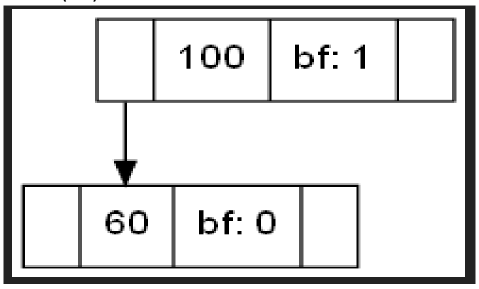
k. If s_bf = -a ; set s_bf = 0 as node inserted in sub tree whoses height is one less than the other subtree; terminate

l. If bf of s and of child in which subtree the node is inserted are same then it means that node is inserted in the subtree whose height was more (in its left subtree if a =1 and right subtree if a = -1) so left rotation if a = 1 and right rotation if a = -1; terminate.

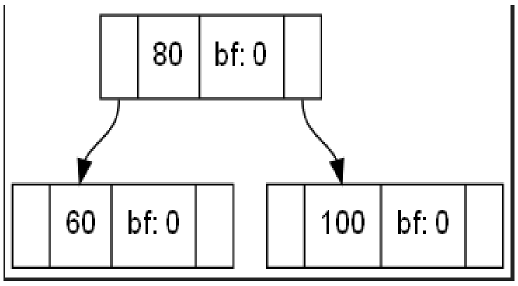
m. If bf of s and of child in which subtree the node is inserted are opposite then it means that node is inserted in the subtree whose height was more (in its right subtree if a =1 and left subtree if a = -1) so two rotations required; left right rotation if a = 1 and right left rotation if a = -1; terminate.

- Test Cases:

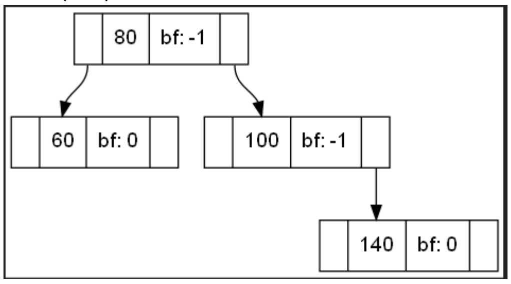
Insert(100)
Insert(60)



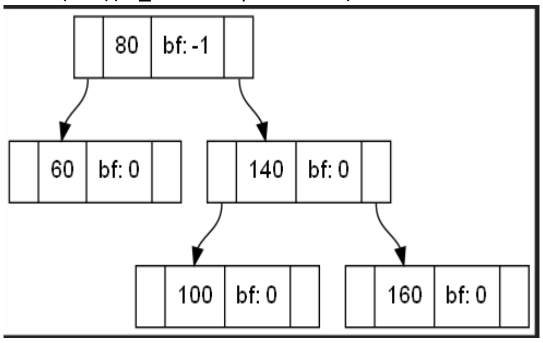
Insert(80)(RR_Rotation performed)



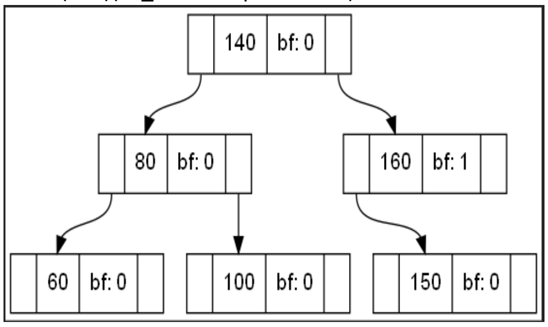
Insert(140)



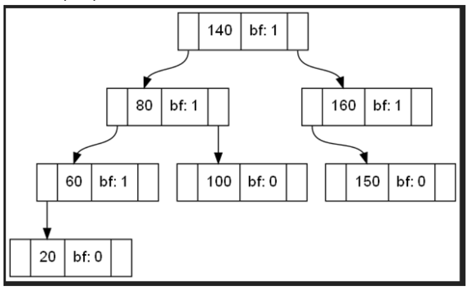
Insert(160)(LL_Rotation performed)



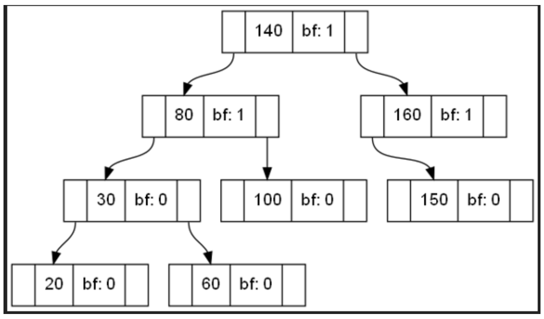
Insert(150)(RL_Rotation performed)



Insert(20)



Insert(30)(LR_Rotation performed)



AVL_Delete(int k)

- This function performs following steps:

1. First check if element exists in tree or not .
2. If element exists then terminate
3. Else following steps are performed
   a. Here normal BST insertion procedure can be followed however tree may become unbalanced so modifications are required.
   b. Here we locate where the node k is present and as we traverse we keep on pushing the node in a stack(node k is not pushed).
   c. Stack stores the nodes traversed and these are the potential nodes where rebalancing may be required.
   d. Once the node k is located then the node can be of following types:
      - ♦ Leaf node
      - ♦ Node with single child
      - ♦ Node with two children
   e. If node k is a leaf node then Following bf's possible for x.
      - ♦ If x has only one child i.e. node k then bf = 0 and remove parent from stack as rebalancing is done delete node k.
      - ♦ If x has two child both leaf node then bf = -1 if k is left child and +1 if k is right child and remove parent from stack as rebalancing is done; delete node k.
      - ♦ If x has two children and the other child has bf =+1 or -1 then Simply delete node k.
   f. If node k has one children
      - ♦ Replace the value of node k with its child's key value
      - ♦ Set bf = 0
      - ♦ Set both child pointers to 0.
   g. If node k has two children's then move to its inorder successor 'i' and push the nodes traversed in this process into the stack.
      - ♦ Rebalancing node k and later when rebalancing of the elements present in the stack will be done don't rebalance it if bf =+1,0,-1.
      - ♦ Rebalancing is done in following ways:
      - ♦ If the parent of inorder successor has two children then don't rebalance it now as height will not be changed.
      - ♦ If node i is the only element in the right subtree of its parent then increment bf by 1 and don't rebalance it when the stack elements will be rebalanced.
      - ♦ If node i is the only child in the left subtree of its parent and its parent has only one child i.e. k then increment its bf by 1.
      - ♦ If node i is the root of the right subtree of its parent (obvious that node i will have no left child) then if it stack top = k then increment bf 2(here incrementing by 2 due to execution purpose) else increment by 1.
      - ♦ Delete node i maintaining the child pointers.

h. For each element present in the stack do the following :
   - Pop stack top node ‘x’
   - If key value of node x is less than k then increment bf by 1(as node k deleted from right subtree) else decrement it by 1(as node k deleted from left subtree).
   - Now following cases of rebalancing possible :
   - If bf of node x >1 and that of its left child >=0 then perform right right rotation

```
   X(bf:2)
  /      \
  Y (bf:0 /1)   P

 / \

Z    W
```

Height of z >= height of w and height of subtree Y>P

   - If bf of node x >1 and that of its left child < 0 then perform left right rotation

```
   X(bf:2)
  /      \
  Y (bf:-1)   P
  / \

Z    W
```

Height of w height of z and height of subtree Y>P

   - If bf of node x <-1 and that of its right child <=0 then perform right right rotation

```
   X(bf:-2)

  /      \
  P        Y (bf:0 /-1)
  / \

 Z    W
```

Height of W>= height of Z and height of subtree Y>P

   - If bf of node x <-1 and that of its right child >0 then perform right left rotation

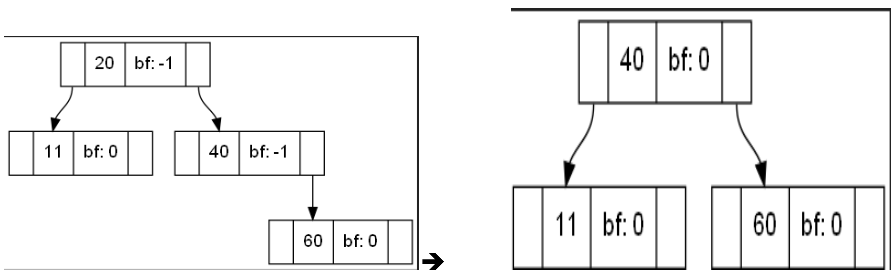
```
   X(bf:-2)

  /      \
  P        Y (bf:1)

 / \

Z    W
```

Height of Z> height of W and height of subtree Y>P

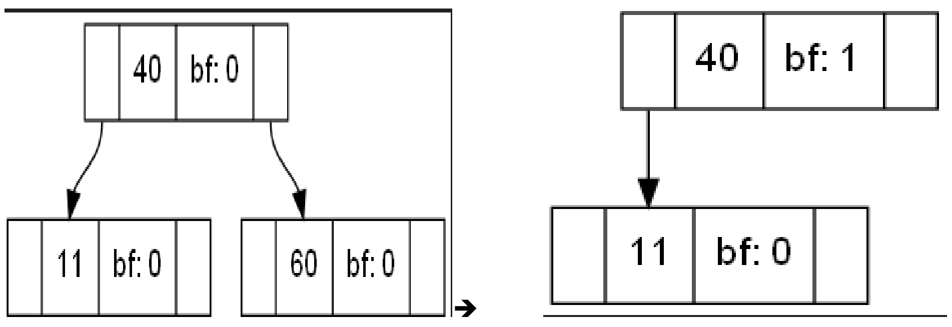
i. Terminate

TestCases:

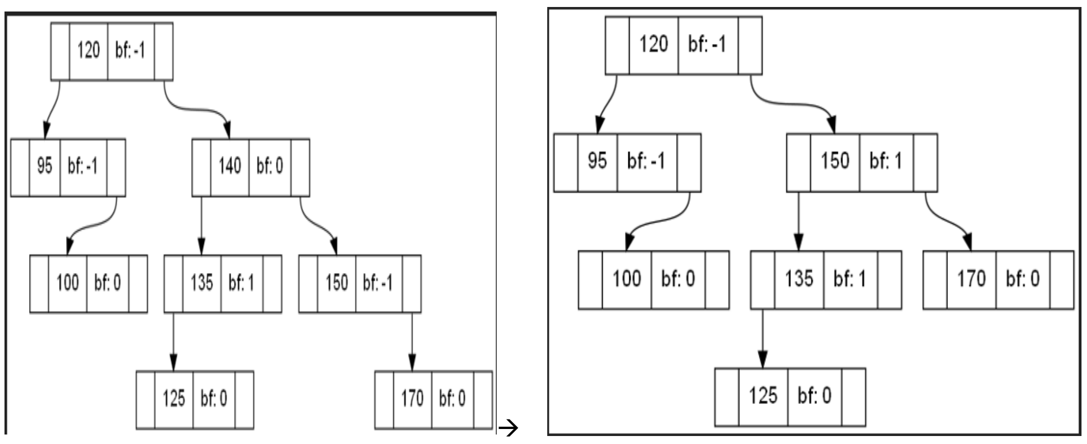
Delete(20)



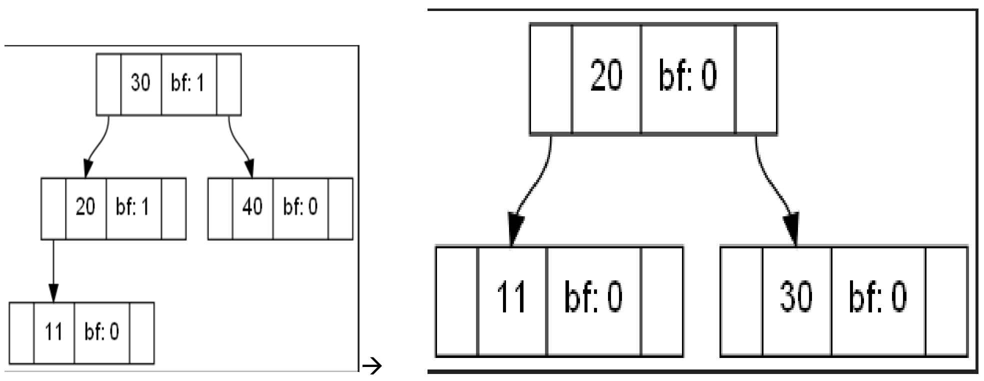
Delete(60)



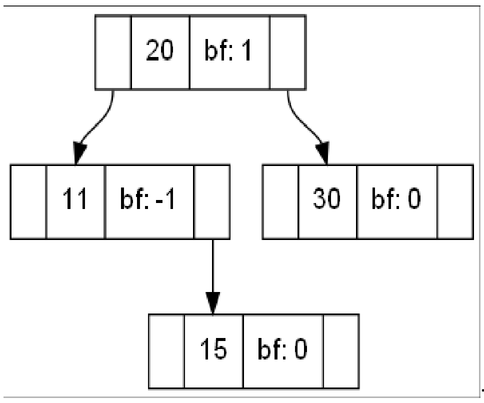
Delete(140)



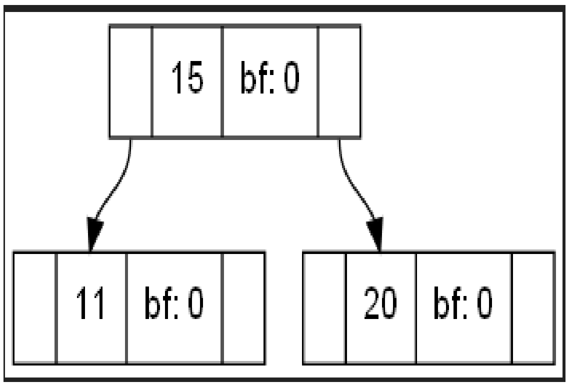
Delete(40)



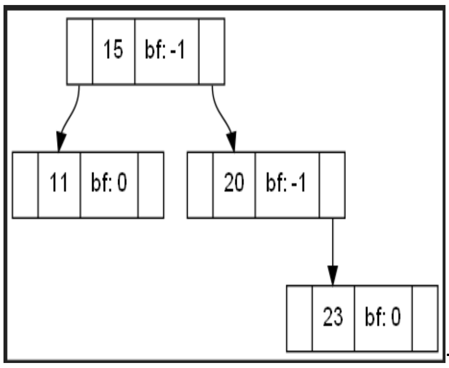
Delete(30)



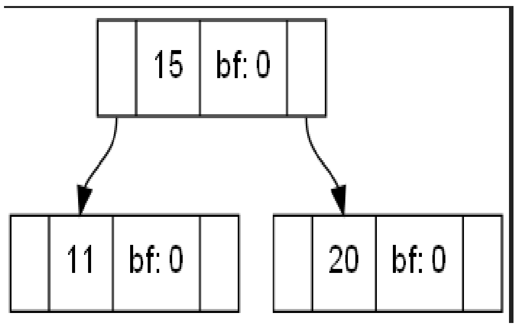
→



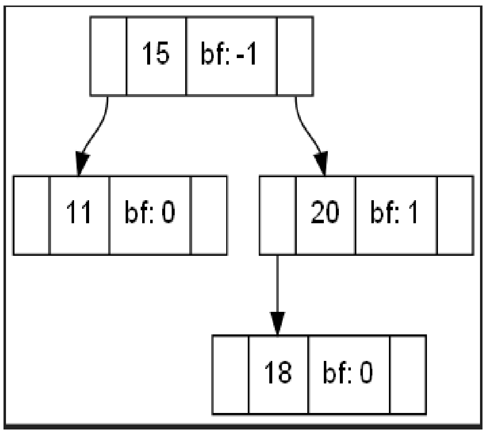
Delete(23)



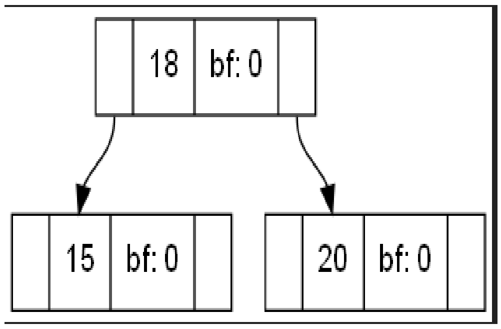
→



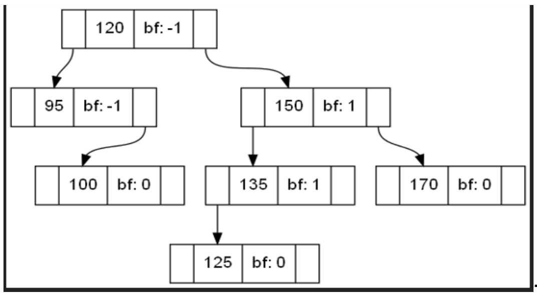
Delete(11)



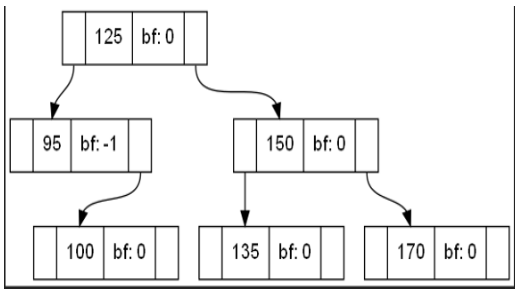
→



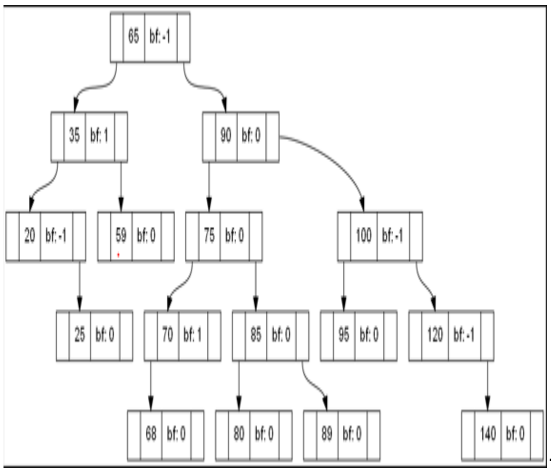
Delete(120)



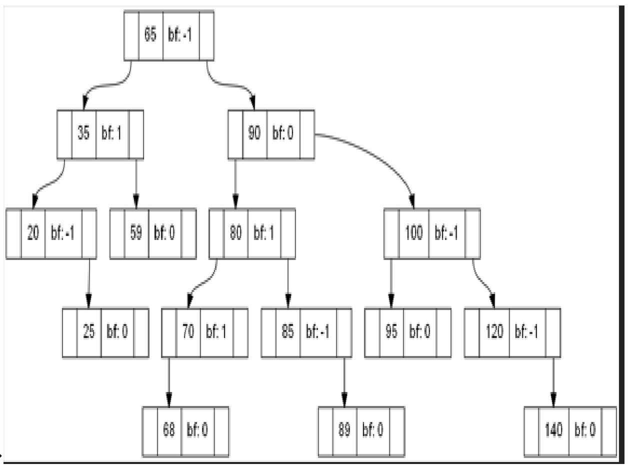
→



Delete(75)



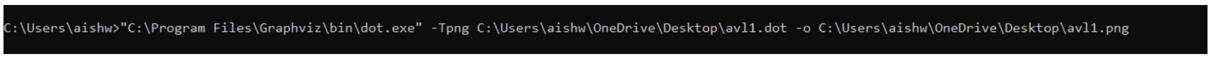
→



- **Print(filename)**

It prints the tree using graphviz

```
digraph AVL {
    node [fontname=Arial , shape = record];
18 [label = "<left> | <mid> 18 | bf: 0 | <right> "];
15 [label = "<left> | <mid> 15 | bf: 0 | <right> "];
"18":left -> "15":mid ;
20 [label = "<left> |<mid> 20 | bf: 0 | <right> "];
 "18":right -> "20": mid;
}
```



```
C:\Users\aishw>"C:\Program Files\Graphviz\bin\dot.exe" -Tpng C:\Users\aishw\OneDrive\Desktop\avl1.dot -o C:\Users\aishw\OneDrive\Desktop\avl1.png
```

1. LL_Rotate(par,par_par):

   Rotate function performs single rotation of rotating the node to left and maintain its balance property.

2. RR_Rotate

   Performs single rotation of rotating the node to right and maintaining its balance property

3. LR_Rotate

   Performs double rotation first LL followed by RR

4. RL_Rotate

   Performs double rotation first LL followed by RR