DS LAB CS513: Assignment 02

Vijay Purohit | 214101058

M.Tech CSE, IITG

Contents

List of Figures

1 FILES AND FUNCTIONS DESCRIPTION

1.1 avl_nodes.h

This header file contains node structure for AVL Tree. Node Name is "ANode".

1.2 avl_functions.h

This header file contains the definition of functions for AVL Class "ATree". It contains the following functions.

1.2.1 Copy Constructor Functions

- **ATree(const ATree &avl_o)**
  This is copy constructor which is used to copy one AVL Class object to another AVL Class object.
- **void operator¯(const ATree &avl_o)**
  This is overloading of assignment operator, used to copy from one AVL Class object to another AVL Class object.
- **ANode* copyLeftRightNode(ANode* tnode)**
  This function is a helper function for copy constructor, taking input as root and recursively copying the tree structure. It creates clone by copying key, left and right pointers.

1.2.2 void t_display(ANode *ptr, int level)

This is helper function for Displaying Tree in the console for better run-time visualization and proper operations handling at run-time. It start from printing level as 1 as input and taking root as input.

1.2.3 int search(int item)

This function is used for searching a node item in AVL Tree. The time complexity is $O(\log n)$ since we are moving from root to leaf path.

1.2.4 void t_insert(int k)

This is insert function for inserting a node key 'k' into AVL Tree and for performing necessary rotations. If element already present throw a exception.

- **LL Imbalance (right rotation)**
  This part calls another function (Refer 1) which performs the right single rotation. Adjusting balance factor at the end.
- **RR Imbalance (left rotation)**
  This part calls another function (Refer 2) which performs the left single rotation. Adjusting balance factor at the end.
- **LR Imbalance (double rotation)**
  This part calls another function (Refer 3) which performs the double rotation, left rotation then right rotation. Adjusting balance factor at the end.
- **RL Imbalance (double rotation)**
  This part calls another function (Refer 4) which performs the double rotation, right rotation then left rotation. Adjusting balance factor at the end.

The time complexity for this function is roughly $O(\log n)$. Number of rotation would be roughly at most one during inserting a single key.

1.2.5 void Delete(int k)

This is delete function for deleting a node key ‘k’ into AVL Tree and for performing necessary rotations.

- **Deleting Node with one or zero child**
  This part calls another function (Refer void case_a(ANode* par, ANode* loc)) which delete the node with key ‘k’ which having zero or one child.
- **Deleting Node with both children**
  This part calls another function (Refer void case_b(ANode* par, ANode* loc)) which delete the node with key ‘k’ which having both the children.
- **LL Imbalance (right rotation)**
  This part calls another function (Refer 1) which performs the right single rotation. Adjusting balance factor at the end.
- **RR Imbalance (left rotation)**
  This part calls another function (Refer 2) which performs the left single rotation. Adjusting balance factor at the end.
- **LR Imbalance (double rotation)**
  This part calls another function (Refer 3) which performs the double rotation, left rotation then right rotation. Adjusting balance factor at the end.
- **RL Imbalance (double rotation)**
  This part calls another function (Refer 4) which performs the double rotation, right rotation then left rotation. Adjusting balance factor at the end.

The time complexity for this function is roughly $O(\log n)$. Number of rotations can be $O(\log n)$ while deleting a node and adjusting balance factor from that node to root of the tree.

1.2.6 void case_a(ANode* par, ANode* loc)

This is helper function for Delete operation for node having one or no child. par is parent of the node to be deleted and loc is the location of node to be deleted. It delete the leaf node and update the pointer of its parent to deleted node child (if any) else to nullptr.

1.2.7 void case_b(ANode* par, ANode* loc)

This is helper function for Delete operation for node having both the children. par is parent of the node to be deleted and loc is the location of node to be deleted. It finds the inorder successor of the node and replace the key of node at location with successor node key, then deleting the successor node with function (Refer void case_a(ANode* par, ANode* loc)).

1.2.8 Rotation Functions

1. **ANode* Imbalance_LL(ANode* s, ANode *r, int s_bf, int r_bf)**
   This function is used to perform right rotation whenever there is LL Imbalance at the node ‘s’. ‘s’ will point to the place where re-balancing may be necessary. ‘r’ point to the node where firsts potential rotation may happen. ‘s_bf’ is the new balance factor of s. ‘r_bf’ is the new balance factor of r. (Figure 2a)
2. **ANode* Imbalance_RR(ANode* s, ANode *r, int s_bf, int r_bf)**
   This function is used to perform left rotation whenever there is RR Imbalance at the node ‘s’. ‘s’ will point to the place where re-balancing may be necessary. ‘r’ point to the node where firsts potential rotation may happen. ‘s_bf’ is the new balance factor of s. ‘r_bf’ is the new balance factor of r. (Figure 2b)

3. **ANode* Imbalance_LR(ANode* s, ANode *r, int s_bf, int r_bf)**
   This function is used to perform double rotation (left then right rotation) whenever there is LR Imbalance at the node 's'. 's' will point to the place where re-balancing may be necessary. 'r' point to the node where firsts potential rotation may happen. 's_bf' is the new balance factor of s. 'r_bf' is the new balance factor of r. (Figure 3)

4. **ANode* Imbalance_RL(ANode* s, ANode *r, int s_bf, int r_bf)**
   This function is used to perform double rotation (right then left rotation) whenever there is RL Imbalance at the node 's'. 's' will point to the place where re-balancing may be necessary. 'r' point to the node where firsts potential rotation may happen. 's_bf' is the new balance factor of s. 'r_bf' is the new balance factor of r. (Figure 4)

1.2.9 void print_graphviz(string file_name)

This function does a level order traversal of the tree and generate two files according to file name provided. One is '.gv' file for graph-viz to read and another final 'png' output file of the tree. It will automatically execute the graph-viz command to generate the file.

2 OPERATION LOGIC

Balance Factor is calculated as the height of left subtree minus the height of the right subtree.

2.1 Print Function Logic

We print the graph using package graphviz. Initially we traverse the tree in Level order manner (BFS Traversal) and store the node information, its children in the file with extension '.gv' according to graphviz format. Then program execute the graphviz command to generate the respective '.png' file. In a program, we can provide the file name suffix in order to distinguish our different '.png' file.

2.2 Search Function Logic

- We are finding the location of search element 'k' starting from the root down to the leaf, until either element exist or we get nullptr reference.
- If it is equal to the current node Key value, then return its reference.
- If it is less than current node Key value, then go to left sub-tree, repeat process.
- Else we go to the right sub-tree and repeat the process.

2.3 Insert Function Logic

The logic is borrowed from the book "The Art of Computer Programming Vol 3, D.E. Knuth". (Function Reference: void t_insert(int k)).

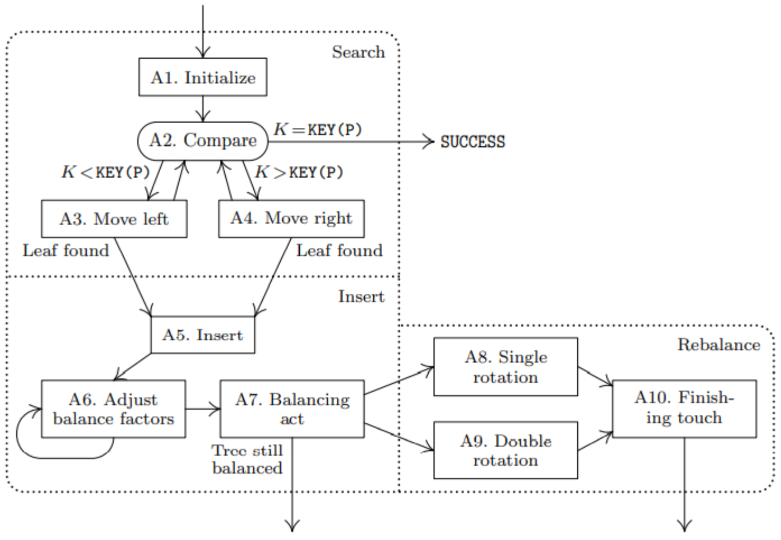


Figure 1: Logic: Balanced Tree Search and Insertion.

A1. Initialization of pointers T, S and P. The pointer variable P will move down the tree; S will point to the place where rebalancing may be necessary, and T always points to the parent of S.

A2. Comparison with existing Key in the Tree, if Key found then terminate otherwise move down left or right subtree of node comparing with the key 'k'.

A3. Moving left $Q \leftarrow LEFTC(P)$, we found the leaf, then we insert the new node there. If BF of any node is non-zero then we update the respective S and P pointers. Set $T \leftarrow P$ and $S \leftarrow Q$. Finally set $P \leftarrow Q$ and Return To A2.

A4. Moving right $Q \leftarrow RIGHTC(P)$, we found the leaf, then we insert the new node there. If BF of any node is non-zero then we update the respective S and P pointers. Set $T \leftarrow P$ and $S \leftarrow Q$. Finally set $P \leftarrow Q$ and Return To A2.

A5. Insertion of New Node. [If From Left] $LEFTC(P) \leftarrow Q$. [If From Right] $RIGHTC(P) \leftarrow Q$.

A6. Adjusting Balance Factors. Now the balance factors on nodes between S and Q need to be changed from zero to ± 1. For that we maintain a variable a which denotes the position of new node with respect to S i.e. If $k < Key(S)$ then set $a \leftarrow +1$ Else $a \leftarrow -1$.
Accordingly with respect to a we SET P and R pointer to the Left or right child of S. From P pointer to Q (i.e. new Node) pointer we update the balance factor of all intermediate nodes between them. If k is in left side of P, then BF is 1 Else -1.

A7. Balancing Act. Several Cases:

- If $BF(S) = 0$ (Previous), then tree has grown higher now (after insertion). Set $BF(S) \leftarrow a$. Terminate.
- If $BF(S) = -a$ (Previous), then tree has gotten more balanced (after insertion). Set $BF(S) \leftarrow 0$. Terminate.
- If $BF(S) = a$ (Previous), then tree has gotten out of balance (after insertion).
  If $BF(R) = a$, Go to A8. Else If $BF(R) = -a$ Go to A9.

A8. Single Rotation. If $a = 1$, Go to (LL Imbalance Logic). Else If $a = -1$ Go to (RR Imbalance Logic).

A9. Double Rotation. If $a = 1$, Go to (LR Imbalance Logic). Else If $a = -1$ Go to (RL Imbalance Logic).

A10. (Final Touch) Since after rotation P now points to New Subtree Root, and T point to the parent of the old subtree root of s. We therefore adjust T pointer to P. If $S == RIGHTC(T)$ (Previous) then set $RIGHTC(T) \leftarrow P$ Else $LEFTC(T) \leftarrow P$.

2.4 Delete Function Logic

The logic is borrowed from the book "The Art of Computer Programming Vol 3, D.E. Knuth". (Function Reference: void Delete(int k)).

A1. Searching. We search for a node value 'k' to be deleted traversing from root to leaf path. At the same time we push nodes we encounter in the path to the stack for potential rotation purpose.

A2. Deletion. If Element exist then we delete it from the tree like a normal balanced binary tree deletion procedure. If node to be deleted has one or zero children (Refer Function void case_a(ANode* par, ANode* loc).
Else if node to be deleted has both the children (Refer Function void case_b(ANode* par, ANode* loc). In this case we also push nodes into stack, until we encounter the successor. At the end, since we deleted the successor node, we changed our node to be deleted value (k=successor key), as now we may need to perform rotation from there.

A3. Adjusting Balance Factor. We Repeat this, until we reach to the head node, i.e no rotation needed.
Now, for current iteration, with the help of the stack, we pop the top element as current S, Next element popped to be as current T.
Also for we maintain a variable a which denotes the position of deleted node with respect to S i.e. If $k < Key(S)$ then set $a \leftarrow +1$ Else $a \leftarrow -1$.

A4. Balancing Act. Several Cases:

- If $BF(S) = a$ (Previous), Node deleted is on same side where tree height is one more than other side. Set $BF(S) \leftarrow 0$. Continue Next iteration of A3.
- If $BF(S) = 0$ (Previous), Tree was perfectly balanced previously, but now one side has extra height after deletion. Set $BF(S) \leftarrow -a$. Terminate.

- If $BF(S) = -a$ (Previous), Deleting a node on one side, make it imbalance due to other side, therefore tree has gotten out of balance (after deletion).
  R pointer points to the opposite side of S in comparison of node to be deleted. If $BF(S) == 1$ then set $R \leftarrow LEFTC(S)$ Else $R \leftarrow RIGHTC(S)$.
  If $BF(R) = -a$, Go to A5. Else If $BF(R) = 0$ Go to A6. Else If $BF(R) = a$ Go to A7.

A5. Single Rotation. If $a = -1$, Go to (LL Imbalance Logic). Else If $a = 1$ Go to (RR Imbalance Logic). Repeat A3.

A6. Single Rotation. If $a = -1$, Go to (LL Imbalance Logic). Else If $a = 1$ Go to (RR Imbalance Logic). Terminate as height of subtree is not changed, so no more rotation required.

A7. Double Rotation. If $a = -1$, Go to (LR Imbalance Logic). Else If $a = 1$ Go to (RL Imbalance Logic). Repeat A3.

A8. (Final Touch) Since after rotation P now points to New Subtree Root, and T point to the parent of the old subtree root of s. We therefore adjust T pointer to P. If $S == RIGHTC(T)$ (Previous) then set $RIGHTC(T) \leftarrow P$ Else $LEFTC(T) \leftarrow P$.

2.5 LL Imbalance Logic

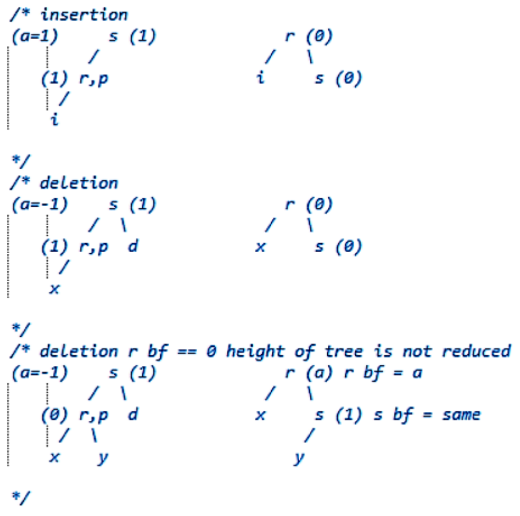
(Rotation Function Reference: 1).
Here we are performing right rotation and change the necessary pointers. (Refer Figure 2a).

1. (Insertion) $BF(S)$ and $BF(R)$ are changed to 0.
2. (Deletion $BF(R) == BF(S)$) $BF(S)$ and $BF(R)$ are changed to 0.
3. (Deletion $BF(R) == 0$) $BF(S)$ is same as previous and $BF(R) = a$ is changed.

2.6 RR Imbalance Logic

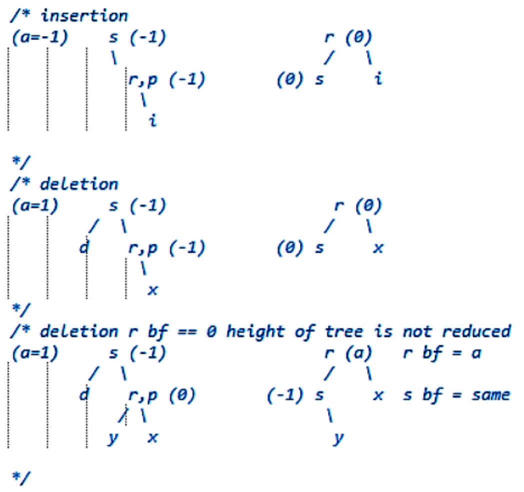
(Rotation Function Reference: 2).
Here we are performing left rotation and change the necessary pointers. (Refer Figure 2b).

1. (Insertion) $BF(S)$ and $BF(R)$ are changed to 0.
2. (Deletion $BF(R) == BF(S)$) $BF(S)$ and $BF(R)$ are changed to 0.
3. (Deletion $BF(R) == 0$) $BF(S)$ is same as previous and $BF(R) = a$ is changed.



```
/* insertion
(a=1)     s (1)              r (0)
            /               /   \
   (1) r,p                 i     s (0)
        /
       i

*/
/* deletion
(a=-1)    s (1)              r (0)
            /  \            /   \
   (1) r,p   d             x     s (0)
        /
       x

*/
/* deletion r bf == 0 height of tree is not reduced
(a=-1)    s (1)              r (a) r bf = a
            /  \            /   \
   (0) r,p   d             x     s (1) s bf = same
        /   \                   /
       x     y                 y

*/
```

(a) LL BF Logic



```
/* insertion
(a=-1)    s (-1)             r (0)
           \                /   \
            r,p (-1)   (0) s     i
             \
              i

*/
/* deletion
(a=1)     s (-1)              r (0)
         /  \                /   \
        d    r,p (-1)   (0) s     x
              \
               x
*/
/* deletion r bf == 0 height of tree is not reduced
(a=1)     s (-1)             r (a)   r bf = a
         /  \               /   \
        d    r,p (0)  (-1) s     x  s bf = same
            /  \            \
           y    x            y

*/
```

(b) RR BF Logic

Figure 2: Rotation: Single Rotation BF Logic

2.7 LR Imbalance Logic

(Rotation Function Reference: 3) Here we are performing double rotation (left then right rotation) and change the necessary pointers. $BF(S)$ and $BF(R)$ are changed according to $BF(P)$. (Refer Figure 3).

1. (Insertion at Left side of S ($a = 1$))
   If P got balanced with $BF(P) = 0$ after insertion, then after rotation S and R is also balanced with

$$BF(S) = 0,\ BF(R) = 0 \tag{1}$$

   If P has left child $BF(P) = a\ (1)$ after insertion, then after rotation it will move to the right child of R.

$$BF(S) = -a,\ BF(R) = 0 \tag{2}$$

   If P has right child $BF(P) = -a\ (-1)$ after insertion, then after rotation it will move to the left child of S.

$$BF(S) = 0,\ BF(R) = a \tag{3}$$

2. (Deletion at Right side of S ($a = -1$))
   If P is already balanced with $BF(P) = 0$ before deletion 'd', after deletion S and R is also balanced with

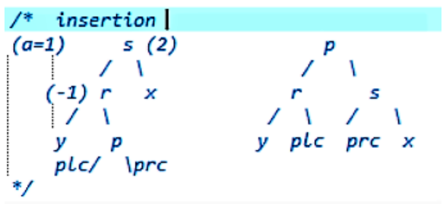
$$BF(S) = 0,\ BF(R) = 0 \tag{4}$$

   If P has only right child $BF(P) = a\ (-1)$ as well as deletion from right side of S, then after rotation it will move to the left child of S

$$BF(S) = 0,\ BF(R) = -a \tag{5}$$

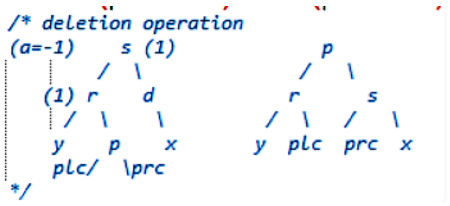
   If P has only left child $BF(P) = -a\ (1)$ as well as deletion from right side of S, then after rotation it will move to the right child of R

$$BF(S) = a,\ BF(R) = 0 \tag{6}$$



```
/*  insertion
(a=1)      s (2)               p
           /  \              /   \
   (-1) r    x             r       s
        /  \             /  \    /   \
       y     p          y  plc  prc   x
        plc/   \prc
*/
```

(a) Insertion



```
/* deletion operation
(a=-1)      s (1)            p
           /  \            /   \
   (1) r     d            r       s
       /  \    \        /  \    /   \
      y    p    x      y  plc  prc   x
      plc/   \prc
*/
```

(b) Deletion

Figure 3: Rotation: LR BF Logic

2.8 RL Imbalance Logic

(Rotation Function Reference: 4) Here we are performing double rotation (right then left rotation) and change the necessary pointers.$BF(S)$ and $BF(R)$ are changed according to $BF(P)$. (Refer Figure 4).

1. (Insertion at Right side of S $(a = -1)$)
   If P got balanced with $BF(P) = 0$ after insertion, then after rotation S and R is also balanced with

$$BF(S) = 0,\ BF(R) = 0 \tag{7}$$

   If P has right child $BF(P) = a\,(-1)$ after insertion, then after rotation it will move to the left child of R.

$$BF(S) = -a,\ BF(R) = 0 \tag{8}$$

   If P has left child $BF(P) = -a\,(-1)$ after insertion, then after rotation it will move to the right child of S.

$$BF(S) = 0,\ BF(R) = a \tag{9}$$

2. (Deletion at Left side of S $(a = 1)$)
   If P is already balanced with $BF(P) = 0$ before deletion 'd', after deletion S and R is also balanced with

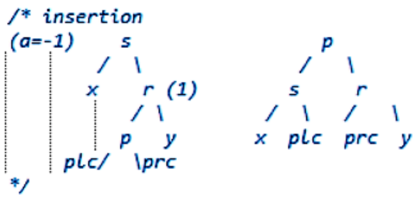
$$BF(S) = 0,\ BF(R) = 0 \tag{10}$$

   If P has only left child $BF(P) = a\,(1)$ as well as deletion from left side of S, then after rotation it will move to the right child of S

$$BF(S) = 0,\ BF(R) = -a \tag{11}$$

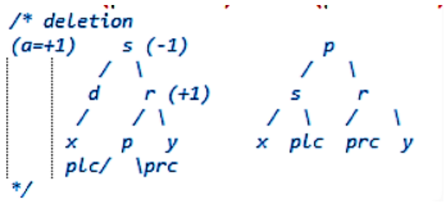
   If P has only right child $BF(P) = -a\,(-1)$ as well as deletion from left side of S, then after rotation it will move to the left child of R

$$BF(S) = a,\ BF(R) = 0 \tag{12}$$



```
/* insertion
(a=-1)      s                p
           /  \            /   \
          x    r (1)      s       r
              /  \       /  \    /   \
             p    y     x  plc  prc   y
      plc/   \prc
*/
```

(a) Insertion



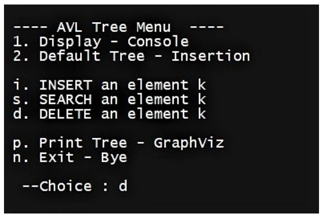
```
/* deletion
(a=+1)      s (-1)           p
           /  \            /   \
          d    r (+1)     s       r
         /    /  \       /  \    /   \
        x    p    y     x  plc  prc   y
        plc/   \prc
*/
```

(b) Deletion

Figure 4: Rotation: RL BF Logic

3 TEST CASES

The Section contains the test cases for main operations performed in the program.



```
---- AVL Tree Menu  ----
1. Display - Console
2. Default Tree - Insertion

i. INSERT an element k
s. SEARCH an element k
d. DELETE an element k

p. Print Tree - GraphViz
n. Exit - Bye

 --Choice : d
```

Figure 5: AVL TREE Main Menu Console

3.1 Search(k)

- **Original Tree**

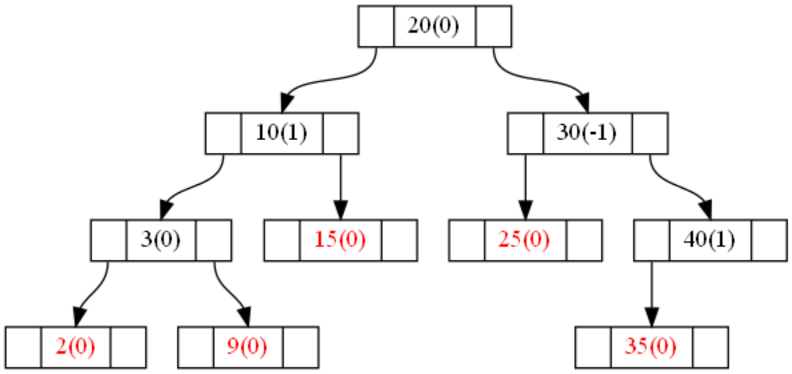
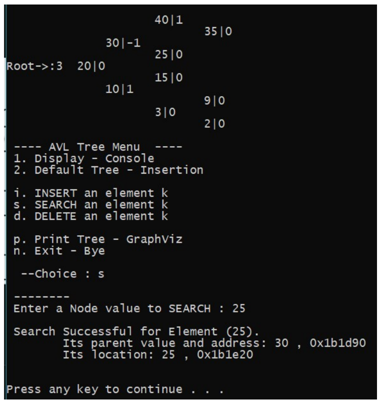


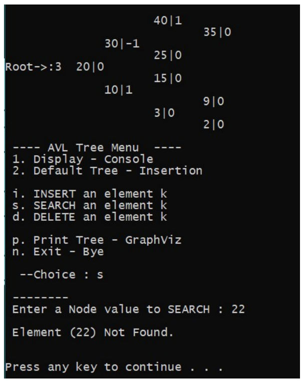
Figure 6: Searching: Original Tree

- **Search: 25, 22**



```
                         40|1
                                35|0
                30|-1
                         25|0
Root->:3  20|0
                         15|0
                10|1
                                9|0
                         3|0
                                2|0

 ---- AVL Tree Menu  ----
 1. Display - Console
 2. Default Tree - Insertion

 i. INSERT an element k
 s. SEARCH an element k
 d. DELETE an element k

 p. Print Tree - GraphViz
 n. Exit - Bye

  --Choice : s

 --------
 Enter a Node value to SEARCH : 25

 Search Successful for Element (25).
        Its parent value and address: 30 , 0x1b1d90
        Its location: 25 , 0x1b1e20


Press any key to continue . . .
```

(a) Search 25: Exist



```
                         40|1
                                35|0
                30|-1
                         25|0
Root->:3  20|0
                         15|0
                10|1
                                9|0
                         3|0
                                2|0

 ---- AVL Tree Menu  ----
 1. Display - Console
 2. Default Tree - Insertion

 i. INSERT an element k
 s. SEARCH an element k
 d. DELETE an element k

 p. Print Tree - GraphViz
 n. Exit - Bye

  --Choice : s

 --------
 Enter a Node value to SEARCH : 22

 Element (22) Not Found.


Press any key to continue . . .
```

(b) Search 22: Not Exist

Figure 7: Searching: Search 25, 22

3.2 Insert(k)

Test Case 1: Final Output of Insertion and Duplicate Value Detection.

- **Inserting: 20, 10, 30, 3, 15, 25, 40, 2, 9, 35**

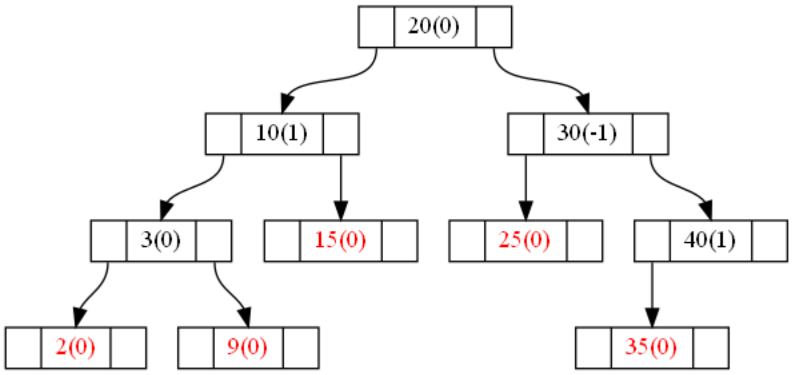
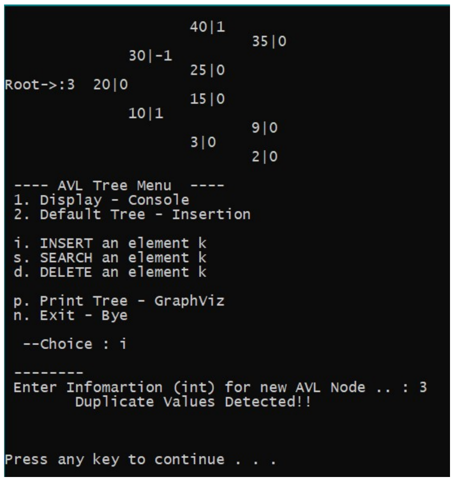


Figure 8: Insertion: Initial Insertion

- **Inserting: Again 3 (Duplicate)**



```
                     40|1
                            35|0
              30|-1
                     25|0
Root->:3  20|0
                     15|0
              10|1
                            9|0
                     3|0
                            2|0

 ---- AVL Tree Menu  ----
 1. Display - Console
 2. Default Tree - Insertion

 i. INSERT an element k
 s. SEARCH an element k
 d. DELETE an element k

 p. Print Tree - GraphViz
 n. Exit - Bye

  --Choice : i

 --------
 Enter Infomartion (int) for new AVL Node .. : 3
        Duplicate Values Detected!!


Press any key to continue . . .
```

Figure 9: Insertion: Duplicate Insertion

Test Case 2: Checking Rotations (Inserting in Sequence: 21, 26, 30, 9, 4, 14, 28)

- **Inserting: 21, 26, 30 ... (RR Imbalance)**

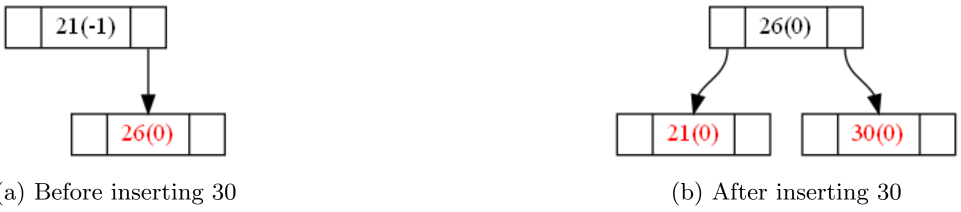
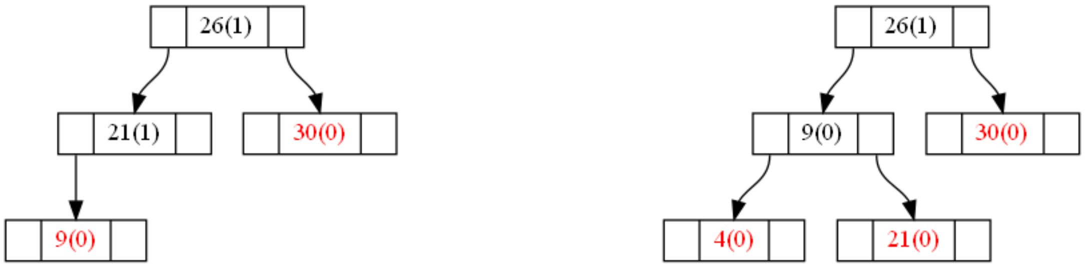


Figure 10: Insertion: RR Imbalance Case

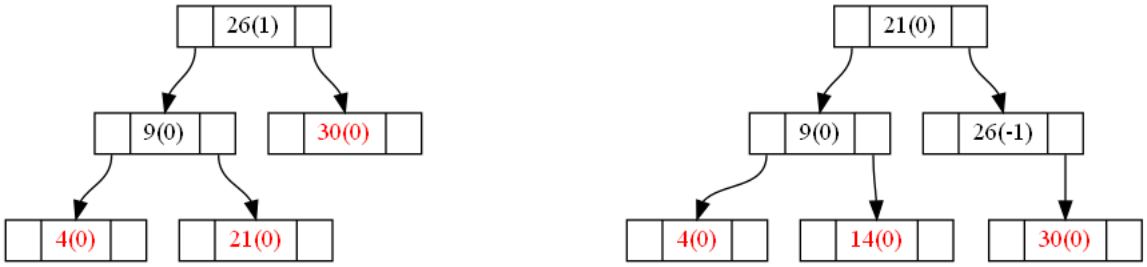
- **Inserting: ... 9, 4 (LL Imbalance)**



(a) Before inserting 4 (b) After inserting 4

Figure 11: Insertion: LL Imbalance Case

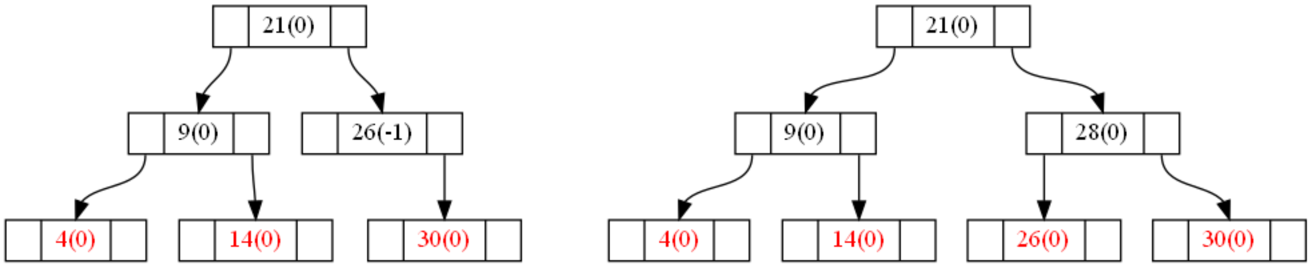
- **Inserting: ... 14 (LR Imbalance)**



(a) Before inserting 14 (b) After inserting 14

Figure 12: Insertion: LR Imbalance Case

- **Inserting: ... 28 (RL Imbalance)**



(a) Before inserting 28 (b) After inserting 28

Figure 13: Insertion: RL Imbalance Case

3.3 Delete(k)

- **Original Tree**

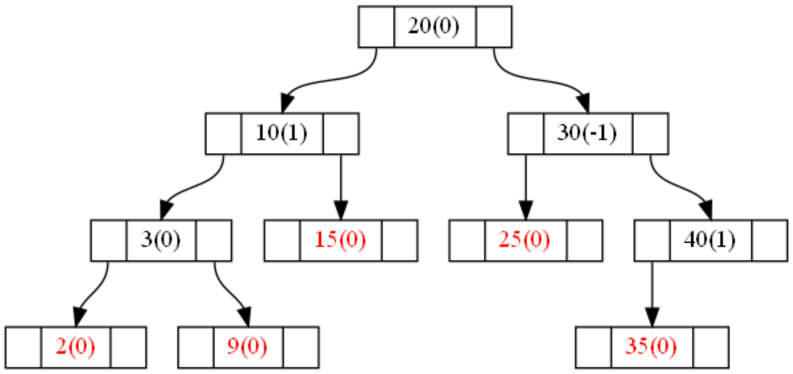
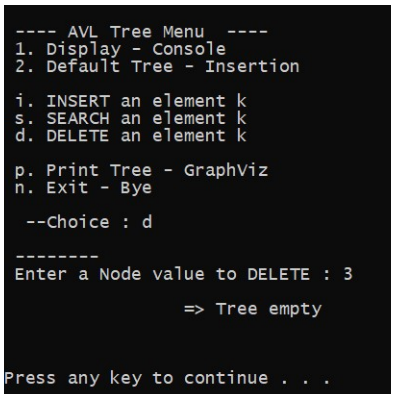


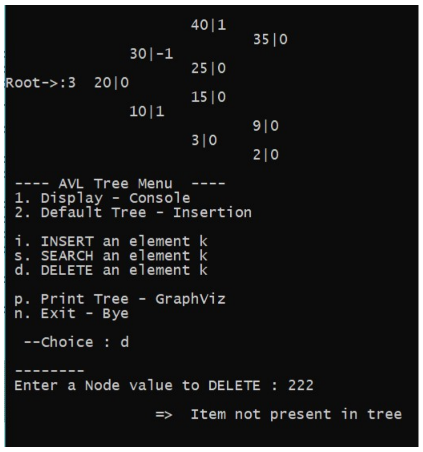
Figure 14: Deletion: Orgirnal Tree

Test Case 1: Empty Tree and Element Does not Exist.



```
---- AVL Tree Menu  ----
1. Display - Console
2. Default Tree - Insertion

i. INSERT an element k
s. SEARCH an element k
d. DELETE an element k

p. Print Tree - GraphViz
n. Exit - Bye

 --Choice : d

--------
Enter a Node value to DELETE : 3

                => Tree empty


Press any key to continue . . .
```

(a) Deletion on Empty Tree



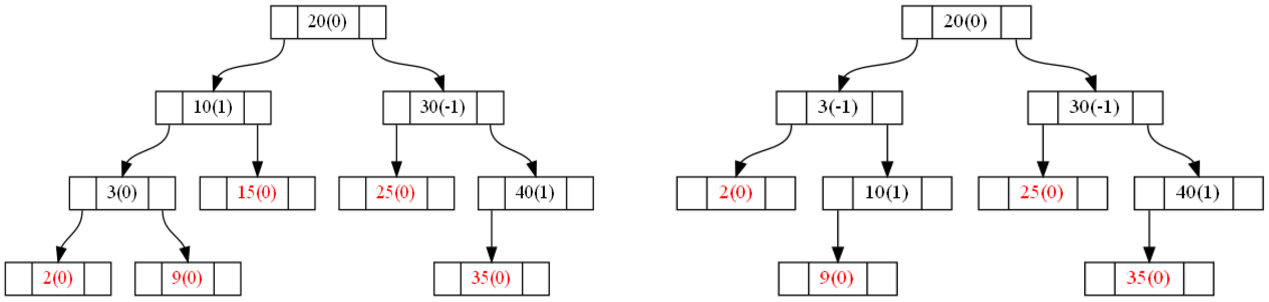
```
                    40|1
                            35|0
             30|-1
                    25|0
Root->:3  20|0
                    15|0
             10|1
                            9|0
                    3|0
                            2|0

 ---- AVL Tree Menu  ----
 1. Display - Console
 2. Default Tree - Insertion

 i. INSERT an element k
 s. SEARCH an element k
 d. DELETE an element k

 p. Print Tree - GraphViz
 n. Exit - Bye

  --Choice : d

 --------
 Enter a Node value to DELETE : 222

                 =>  Item not present in tree
```

(b) Deleting 222, Element does not exist

Figure 15: Deletion: Empty Tree/Element not exist

Test Case 2: Checking Rotations.

- **Deleting: 15 ... (LL Imbalance)**

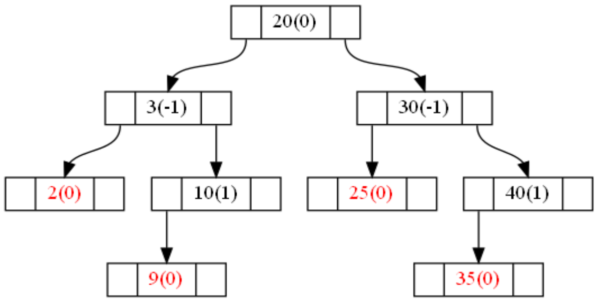


(a) Before Deleting 15

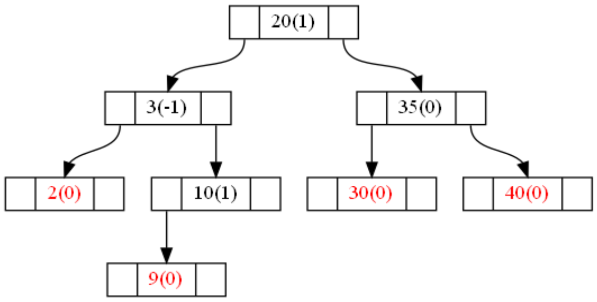
(b) After Deleting 15

Figure 16: Deletion: LL Imbalance Case

- **Deleting: ... 25 (RL Imbalance)**



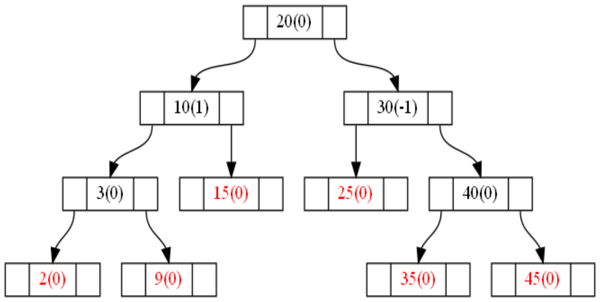
(a) Before Deleting 25



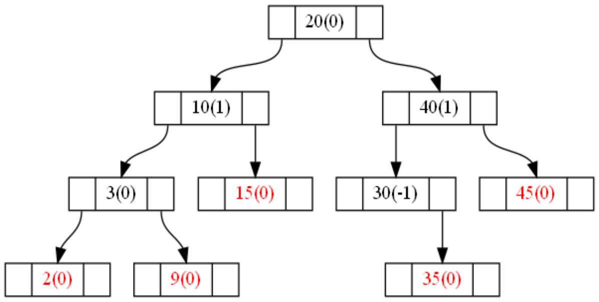
(b) After Deleting 25

Figure 17: Deletion: RL Imbalance Case

- **Deleting: 25 (RR Imbalance)**



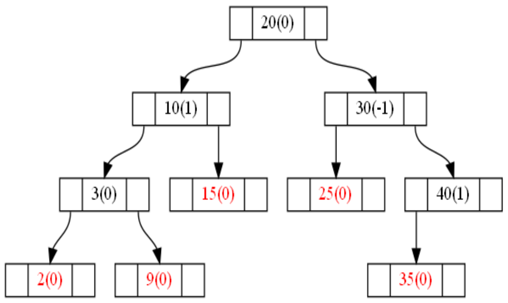
(a) Before Deleting 25



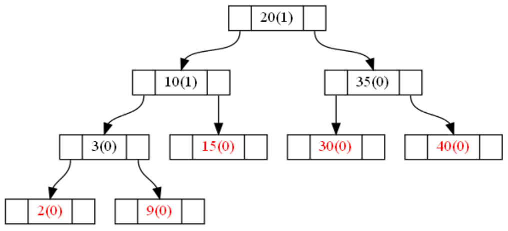
(b) After Deleting 25

Figure 18: Deletion: RR Imbalance Case

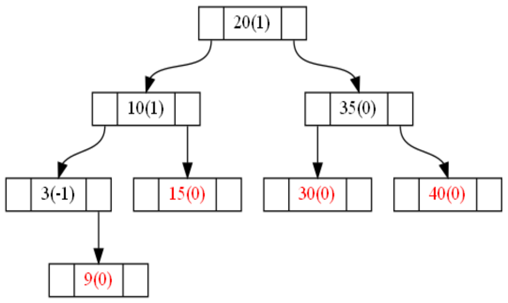
- **Deleting: 25, 2, 15 (LR Imbalance)**



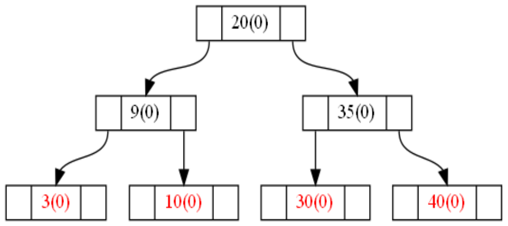
(a) Before Deleting 25



(b) After Deleting 25



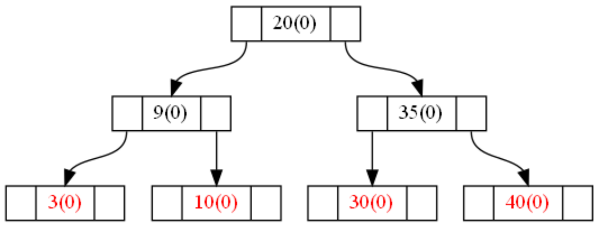
(c) After Deleting 2



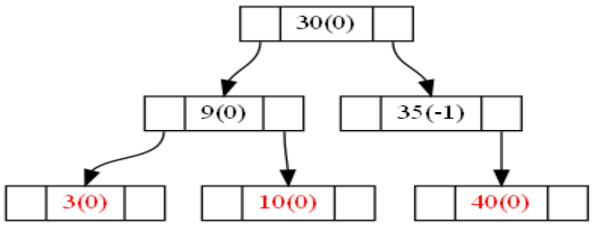
(d) After Deleting 15

Figure 19: Deletion: LR Imbalance Case

- **Deleting: 25, 2, 15, 20 (Deleting with two child)**



(a) Before Deleting 20

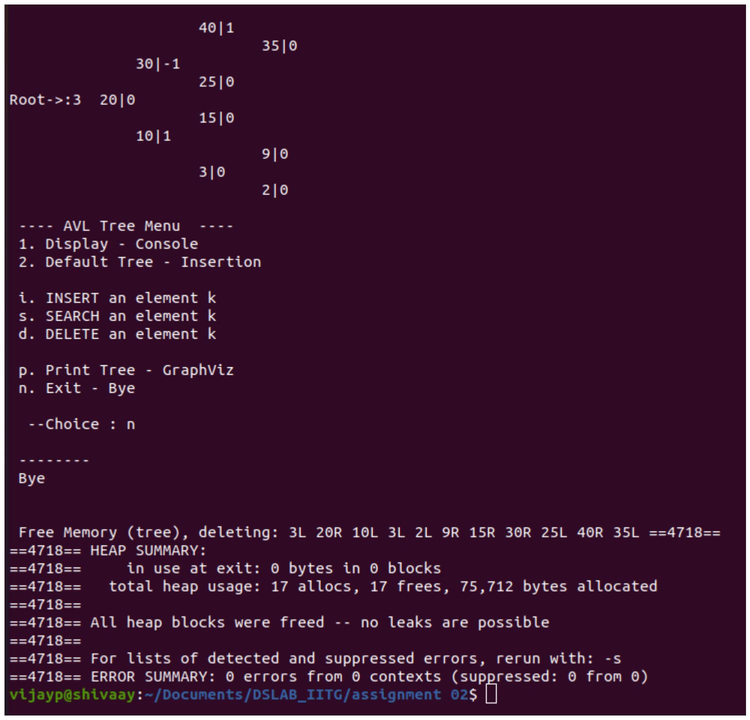


(b) After Deleting 20

Figure 20: Deletion: Two Child Case

3.4 Exit

Test Case 1: Exiting from program and checking memory leak.



```
                     40|1
                            35|0
              30|-1
                     25|0
Root->:3  20|0
                     15|0
              10|1
                            9|0
                     3|0
                            2|0

 ---- AVL Tree Menu  ----
 1. Display - Console
 2. Default Tree - Insertion

 i. INSERT an element k
 s. SEARCH an element k
 d. DELETE an element k

 p. Print Tree - GraphViz
 n. Exit - Bye

  --Choice : n

 --------
 Bye


 Free Memory (tree), deleting: 3L 20R 10L 3L 2L 9R 15R 30R 25L 40R 35L ==4718==
==4718== HEAP SUMMARY:
==4718==     in use at exit: 0 bytes in 0 blocks
==4718==   total heap usage: 17 allocs, 17 frees, 75,712 bytes allocated
==4718==
==4718== All heap blocks were freed -- no leaks are possible
==4718==
==4718== For lists of detected and suppressed errors, rerun with: -s
==4718== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
vijayp@shivaay:~/Documents/DSLAB_IITG/assignment 02$
```

Figure 21: Final Exit