Contents

CS513: AVL Tree Assignment

Prateekshya Priyadarshini

M.Tech CSE

1 Approaches

1.1 Keeping a dummy node to hold the tree

For making rotation easier, one dummy node is taken. The actual tree is stored at the right child of this dummy node. It makes rotation easier when the root itself is imbalanced.

1.2 Using balance factor for each node

A usual approach is to store the height of the subtree for each node. But the demerit of this approach is, when rotation happens, we need to update the height for the entire subtree. So instead of using height we can use balance factor for each node to check for imbalance.

1.3 Using level order traversal for printing the tree

Since the order of nodes mentioned in the graphviz file is important, we need to do a traversal of the tree starting from the root such that the root should be mentioned at the top. This can be done using either preorder traversal or level order traversal. Here level order traversal is being considered and a separate queue is implemented for that purpose.

1.4 Two global variables for printing the tree

1. **fileCount**
   This variable is initialized to 0. It gets incremented everytime a new graphviz file is created. So that we can create different files each time. For example-graph0.gv,graph1.gv,graph2.gv etc.

2. **fileType**
   This variable stores 0 for windows OS, 1 for linux OS. More options can be added as per requirement. This variable helps us to decide which type of commands file we have to create. If we are in windows, we need to add all the commands needed to convert graphviz files to png files into a batch file. Similary in linux we have to add all of them to a shell script.

1.5 Extra variables in Tree Node structure

To implement AVL Tree operations one integer variable is used. It stores the balance factor for each node. Balance factor is the difference between the height of left subtree and right subtree i.e. $bf = h(lst) + h(rst)$

1.6 Deleting the nodes with two children using successor replacement strategy

When a node with two children is being deleted, it can be replaced with either with its predecessor or with its successor. Here successor replacement strategy is chosen.

1.7 Creating batch files or shell scripts

Since the commands to generate a .png file from a .gv file are not straight forward for a beginner, this program generates batch file in windows and shell script in linux which contains all those commands. Every time a new tree is printed, these files get executed and the respective images get generated. For reference, the images won't be deleted or replaced till the program is being executed.

1.8 Storing the entire execution process

The entire execution process is stored in $< user\_given\_name >$**output.txt**. This file can be referred to check what went wrong, which images refer to which trees etc. Also the graphviz files and png files are stored along with a sequence number i.e. $< user\_given\_name >$**1.png** or $< user\_given\_name >$**9.gv**.

1.9 Commands used for various operations

1.9.1 For makefile

1. g++ -c AVLTreeImpl.cpp
2. g++ -o AVLTreeImpl AVLTreeImpl.o

1.9.2 For valgrind

1. g++ -o AVLTreeImpl -g AVLTreeImpl.cpp
2. valgrind –tool=memcheck –leak-check=yes –show-reachable=yes –num-callers=20 –track-fds=yes ./AVLTreeImpl

2 Files, Functions and Their Logic

2.1 nodes.h

This header file contains node structures for linked list, queue and binary search tree.

2.2 datastructures.h

This header file contains linked list class, queue class and binary search tree class along with their respective function prototypes.

2.3 stack.h

This header file contains the below mentioned functions for linked list.

2.3.1 void push(TreeNode *)

This function takes a node of type TreeNode and adds the entire node to the top of the stack. Time complexity is constant.

2.3.2 TreeNode * pop()

This function removes and returns the top element of the stack. Time complexity is constant.

2.3.3 bool isEmpty()

This function checks whether the stack is empty or not. If the size is zero then it return true otherwise returns false. Time complexity is constant.

2.3.4 int getSize()

This function returns the current size of the stack. Time complexity is constant.

2.3.5 TreeNode * viewTop()

This function returns the top element of the stack without removing it. Time complexity is constant.

2.4 queue.h

This header file contains the below mentioned functions for queue.

2.4.1 void enqueue(TreeNode * node)

This function takes a node of type TreeNode and adds the entire node to the queue at the rear end. Time complexity is constant.

2.4.2 TreeNode * dequeue()

This function removes and returns the front element of the queue. Time complexity is constant.

2.4.3 bool isEmpty()

This function checks whether the queue is empty or not. If the size is zero then it returns true otherwise returns false. Time complexity is constant.

2.4.4 int getSize()

This functions returns the current size of the queue. Time complexity is constant.

2.5 avlt.h

The whole logic of AVL Tree using balance factors revolves around a fact that a fully balanced node cannot have imbalance after insertion. Imbalance will only happen when the node is already having one extra height either on left or on right. Mathematically, if a node has balance factor 0, it won't be affected with one insertion, but if it is 1 or -1, we need to perform rotation. According to the formula, 1 indicates that the node has more height on left subtree and -1 indicates that the node has more height on the right subtree. We are also using a variable $newBF$ to indicate which subtree the new node got inserted to or deleted from. If it is 1, changes occured on left subtree and if it is -1, changes occured in the right subtree. This header file contains the below mentioned functions for threaded binary search tree.

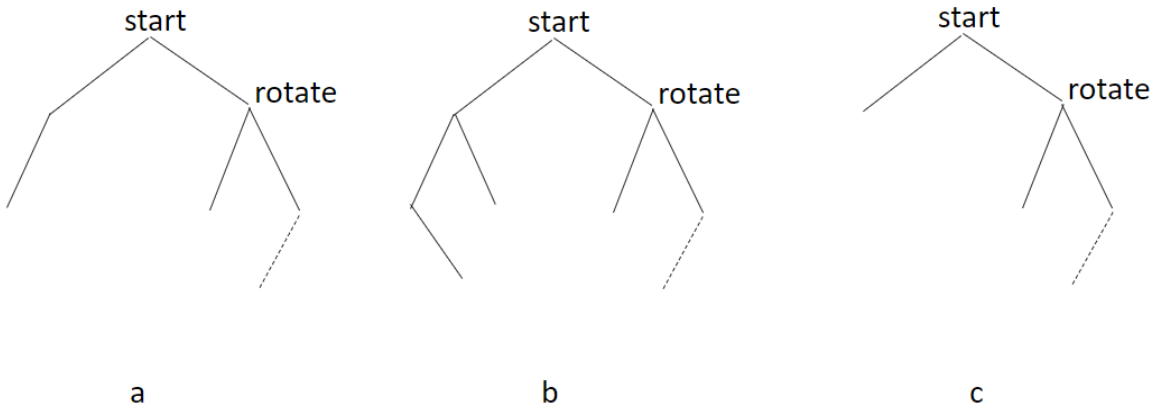


Figure 1: Various cases

2.5.1 void insert(int k)

This function is called from the main function. It searches for the location where k can be inserted. If k is already present somewhere in the tree, it throws **Duplicate Value Exception**.
Some variables used for convenience are-

1. *start* pointer points to the node where the potential imbalance may happen after insertion.
2. *tree* pointer stores the parent of *start* so that we can attach the rotated subtree to it.
3. *newBF* will be updated according to the subtree of *start* to which the new node got inserted i.e. (store 1 if inserted to left of *start*, -1 otherwise).
4. *end* pointer points to the new node which is inserted.
5. *rotate* pointer points to the child node of *start* on $start - end$ path and that is the node where the first rotation will happen.

After updating all the pointers, we start updating balance factors for each node on the $start-end$ path except the two nodes themselves. The logic goes same i.e. make it 1 if inserted on left, -1 otherwise. Now three cases arise.

1. **start->bF == 0**
   Store *newBF* in it and return. Because there is no need of rotation. Since *newBF* already indicates on which subtree the height got increased, we can straight away store it. Refer **Figure 1 (a)**.
2. **start->bF == -1*newBF**
   This indicates that, the node which *start* points to, was already having more height on one side, but the insertion actually happened on the other side. So we need to update the balance factor to 0 and return. Refer **Figure 1 (b)**.
3. **start->bF == newBF**
   This indicates that, the node which *start* points to, was already having more height on one side, and the insertion also happened on the same side. So here we need to perform rotation. Again two cases arise here. Refer **Figure 1 (c)**.

(a) **start->bF == rotate->bF**
This indicates that the new node got inserted to either left of *start* as well as left of *rotate* or right of *start* as well as right of *rotate*. In such case single rotation will be needed. According to *newBF* we need to call the respective functions i.e. if it is 1 that means it is *LL*-imbalance and we need to call TreeNode * rotateLL(TreeNode * oldRoot, TreeNode * rotate, int oldRoofBF, int rotateBF) otherwise we need to call TreeNode * rotateRR(TreeNode * oldRoot, TreeNode * rotate, int oldRoofBF, int rotateBF). After rotation, the balance factrors of *start* and *rotate* will be 0 since these kind of cases arise towards leaf nodes and after rotation *start* becomes leaf and *rotate* becomes fully balanced.

(b) **start->bF == -1*rotate->bF**
This indicates that the new node got inserted to either left of *start* and right of *rotate* or vice versa. In such case double rotation will be needed. According to *newBF* we need to call the respective functions i.e. if it 1 that means it is *LR*-imbalance and we need to call TreeNode * rotateLR(TreeNode * oldRoot, TreeNode * rotate, int newBF, int operation) otherwise we need to call TreeNode * rotateRL(TreeNode * oldRoot, TreeNode * rotate, int newBF, int operation). Balance factor updation is discussed within the rotation functions.

After rotation void transplant(TreeNode * mainTree, TreeNode * replace, TreeNode * subTree) function is called to cut the link of the second parameter from the first parameter and attach the third parameter as the left or right child of first parameter.

The time complexity of this function is $O(h)$ where h is the height of the tree.

2.5.2 void deleteK(int k)

This function searches for the node with value k and deletes it. It also adds the ancestors of this node to a stack. If k is not present then it throws **Missing Node Exception**.
Similar variables as insertion are used here for convenience.
First of all we check if the *node* to be deleted is having two children or not. If it doesn't have two children we call TreeNode * deleteNodeWithNullChildren(TreeNode * node, TreeNode * parent) function to delete this node. Otherwise we find out the successor of the *node* and delete it using the same function TreeNode * deleteNodeWithNullChildren(TreeNode * node, TreeNode * parent). Since that function returns the deleted node, we can catch it here and replace it with the *node* to be deleted.
The time complexity of this function is $O(h)$ where h is the height of the tree.

2.5.3 bool search(int k)

This function runs a standard search process. It returns *true* if a node with value k is present and $false$ otherwise. The time complexity of this function is $O(h)$ where h is the height of the tree.

2.5.4 void print(const char * fileName)

This function does a level order traversal of the tree and generates two files. One is a graphviz(.gv) file and the other file depends on the operating system we are using. For windows it generates a batch file and for linux it generates a shell script. The file name given by the user is used to name all the files. For avoiding conflicts, numbers are added at the end of the file name.

2.5.5 bool isLeaf(TreeNode * node)

This function returns true if the given node is a leaf otherwise returns false.

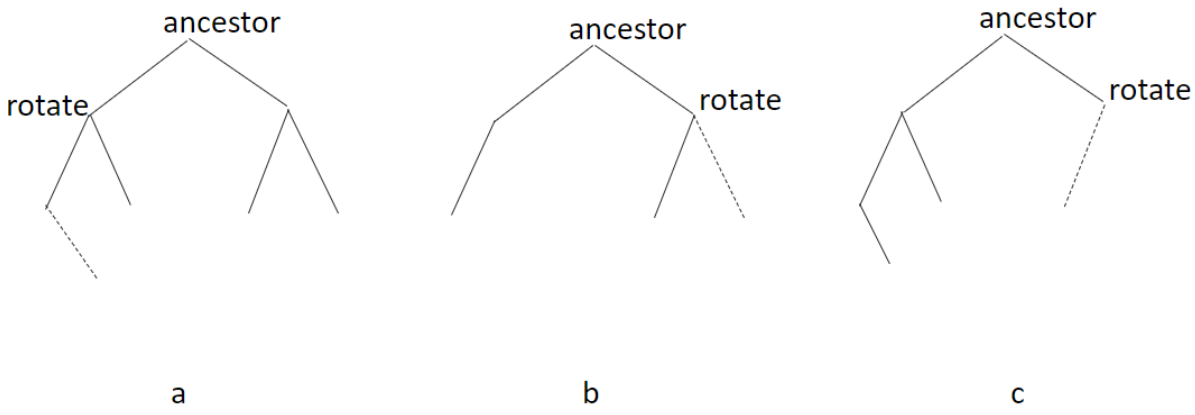


Figure 2: Various cases

2.5.6 TreeNode * searchNode(int k)

This function is similar to bool search(int k). The only difference is it returns a pointer to the node having value k (*nullptr* if k is absent). Added to that, it also adds all the ancestors of that node to a stack which helps in deletion.

2.5.7 TreeNode * deleteNodeWithNullChildren(TreeNode * node, TreeNode * parent)

This function is called from void deleteK(int k). It takes two parameters i.e. the node to be deleted and its parent which are denoted as *node* and *parent* respectively. First it replaces *parent*'s respective child with *node*'s child or *nullptr* depending on whether *node* is having one child or it is a leaf node. Then a loop is executed for all the stack elements. So in each iteration we move from *parent* towards *root*.
Some variables used for convenience are-

1. *ancestor* points to the current ancestor of *node* on $parent - root$ path.
2. *tree* points to the parent of *ancestor*.

Other variables like *rotate* and $newBF$ are similar to insertion function.
We pop *ancestor* from stack and update $newBF$ according to which subtree of *ancestor*, *node* belongs to i.e. 1 for left subtree and -1 for right subtree.
Now three cases arise again.

1. **ancestor->bF == newBF**
   This indicates that *ancestor* had one extra height on the same side from where the node got deleted. That means now *ancestor* is fully balanced. So store 0 in $ancestor- > bF$ and continue the loop. Refer **Figure 2 (a)**.
2. **ancestor->bF == 0**
   This indicates that, the *ancestor* was fully balanced. After deletion, there is no need of rotation. But since we deleted from one subtree, the height becomes smaller on that side but the other subtree is still same. So store $-1 * newBF$ in $ancestor- > bF$ and return from the function. No need to check for further ancestors since the actual height of that subtree is intact. Refer **Figure 2 (b)**.
3. **ancestor->bF == -1*newBF**
   This indicates that, *ancestor* was having one extra height on one of its subtrees and the deletion

occured from the other side. Hence it became more imbalanced and we need to perform rotation. Again three cases arise here. Refer **Figure 2 (c)**.

(a) **ancestor->bF == rotate->bF**
This indicates that similar to *ancestor*, *rotate* was also having one extra height on the same side subtree as *ancestor* and the deletion occured from the other side. In such case single rotation will be needed. According to $newBF$ we need to call the respective functions i.e. if it is 1 that means it is RR-imbalance and we need to call TreeNode * rotateRR(TreeNode * oldRoot, TreeNode * rotate, int oldRoofBF, int rotateBF) otherwise we need to call TreeNode * rotateLL(TreeNode * oldRoot, TreeNode * rotate, int oldRoofBF, int rotateBF). After rotation, the balance factrors of *ancestor* and *rotate* will be 0 for similar reasons mentioned above.

(b) **rotate->bF == 0**
This case doesn't arise during insertion. Because while inserting, we rotate the subtree as soon as we get the first imbalance. So we can never get balance factor 0 at the node pointed by *rotate*. But here the node pointed by *rotate* can have 0 as the balance factor. In this case also single rotation is needed. According to $newBF$ we need to call the respective functions i.e. if it is 1 that means it is RR-imbalance and we need to call TreeNode * rotateRR(TreeNode * oldRoot, TreeNode * rotate, int oldRoofBF, int rotateBF) otherwise we need to call TreeNode * rotateLL(TreeNode * oldRoot, TreeNode * rotate, int oldRoofBF, int rotateBF). After rotation, the balance factror of *rotate* will be $newBF$.

(c) **ancestor->bF == -1*rotate->bF**
This indicates that *rotate* has excatly opposite case as *ancestor* i.e. if for *ancestor*, extra height was on left subtree and node from right subtree got deleted then for *rotate*, extra height was on right subtree and node from left subtree got deleted and vice versa. In such case double rotation will be needed. According to $newBF$ we need to call the respective functions i.e. if it 1 that means it is RL-imbalance and we need to call TreeNode * rotateRL(TreeNode * oldRoot, TreeNode * rotate, int newBF, int operation) otherwise we need to call TreeNode * rotateLR(TreeNode * oldRoot, TreeNode * rotate, int newBF, int operation). Balance factor updation is discussed within the rotation functions.

This process goes on for all the ancestors on the path till root of the tree. The rotation function calls in case of deletion are opposite to the insertion function. After each rotation void transplant(TreeNode * mainTree, TreeNode * replace, TreeNode * subTree) function is called to cut the link of the second parameter from the first parameter and attach the third parameter as the left or right child of first parameter. Then we check if the balance factor of the root of rotated subtree is same as the previous balance factor of the old root of the same subtree i.e. before rotation, that simply indicates that the height of the subtree has not changed after rotation. So we can stop the loop there and return. No need to rotate any further ancestors.

2.5.8 TreeNode * getParent(TreeNode * node)

This function finds the parent node of a given node in $O(h)$ where h is the height of the tree.

2.5.9 TreeNode * leftMost(TreeNode * node)

This function finds the leftmost node of a given node in $O(h)$ where h is the height of the tree.

2.5.10 TreeNode * rightMost(TreeNode * node)

This function finds the rightmost node of a given node in $O(h)$ where h is the height of the tree.

2.5.11 TreeNode * rotateLL(TreeNode * oldRoot, TreeNode * rotate, int oldRoofBF, int rotateBF)

Here *oldRoot* points to the node where imbalance occured and *rotate* points to its child on the path of imbalance. Since it is *LL*-imbalance, we do one right rotation. We point *newRoot* to *rotate*. Then we attach the right subtree of *rotate* as the left child of *oldRoot* and *oldRoot* as the right child of *rotate*. Then we update the balance factor given in the parameters.

2.5.12 TreeNode * rotateRR(TreeNode * oldRoot, TreeNode * rotate, int oldRoofBF, int rotateBF)

oldRoot and *rotate* follow similar conventions. Since it is *RR*-imbalance, we do one left rotation. We point *newRoot* to *rotate*. Then we attach the left subtree of *rotate* as the right child of *oldRoot* and *oldRoot* as the left child of *rotate*. Then we update the balance factor given in the parameters.

2.5.13 TreeNode * rotateLR(TreeNode * oldRoot, TreeNode * rotate, int newBF, int operation)

oldRoot and *rotate* follow similar conventions. Since it is *LR*-imbalance, we do one left rotation followed by one right rotation. We point *newRoot* to *rotate*'s right child since that is going to be the new root. Then we attach the left subtree of *newRoot* as the right child of *rotate* and *rotate* as the left child of *newRoot*. This finishes the first left rotation. Then we attach the right subtree of *newRoot* as the left child of *oldRoot* and *oldRoot* as the right child of *newRoot* which suffices the second right rotation.
We need to update the balance factors of *oldRoot* and *rotate* according to that of *newRoot* and set the balance factor of *newRoot* to 0. If the balance factor of *newRoot* is 0, then store 0 for both *oldRoot* and *rotate*. For other cases the balance factors are getting updated according to the balance factor of the subtree that gets attached to the respective nodes during rotation. In case of deletion, the updation of *oldRoot* and *rotate* will get exchanged.

2.5.14 TreeNode * rotateRL(TreeNode * oldRoot, TreeNode * rotate, int newBF, int operation)

oldRoot and *rotate* follow similar conventions. Since it is *RL*-imbalance, we do one right rotation followed by one left rotation. We point *newRoot* to *rotate*'s left child since that is going to be the new root. Then we attach the right subtree of *newRoot* as the left child of *rotate* and *rotate* as the right child of *newRoot*. This finishes the first right rotation. Then we attach the left subtree of *newRoot* as the right child of *oldRoot* and *oldRoot* as the left child of *newRoot* which suffices the second left rotation.
We need to update the balance factors of *oldRoot* and *rotate* according to that of *newRoot* and set the balance factor of *newRoot* to 0. If the balance factor of *newRoot* is 0, then store 0 for both *oldRoot* and *rotate*. For other cases the balance factors are getting updated according to the balance factor of the subtree that gets attached to the respective nodes during rotation. In case of deletion, the updation of *oldRoot* and *rotate* will get exchanged.

2.5.15 void transplant(TreeNode * mainTree, TreeNode * replace, TreeNode * subTree)

This function cuts the link of *replace* from *mainTree* and attaches *subTree* at the same location of *mainTree*.

2.5.16 TreeNode * copyNodes(TreeNode * root, unorderedmap map)

This is a recursive function called by the copy constructor which creates a replica of the given tree.

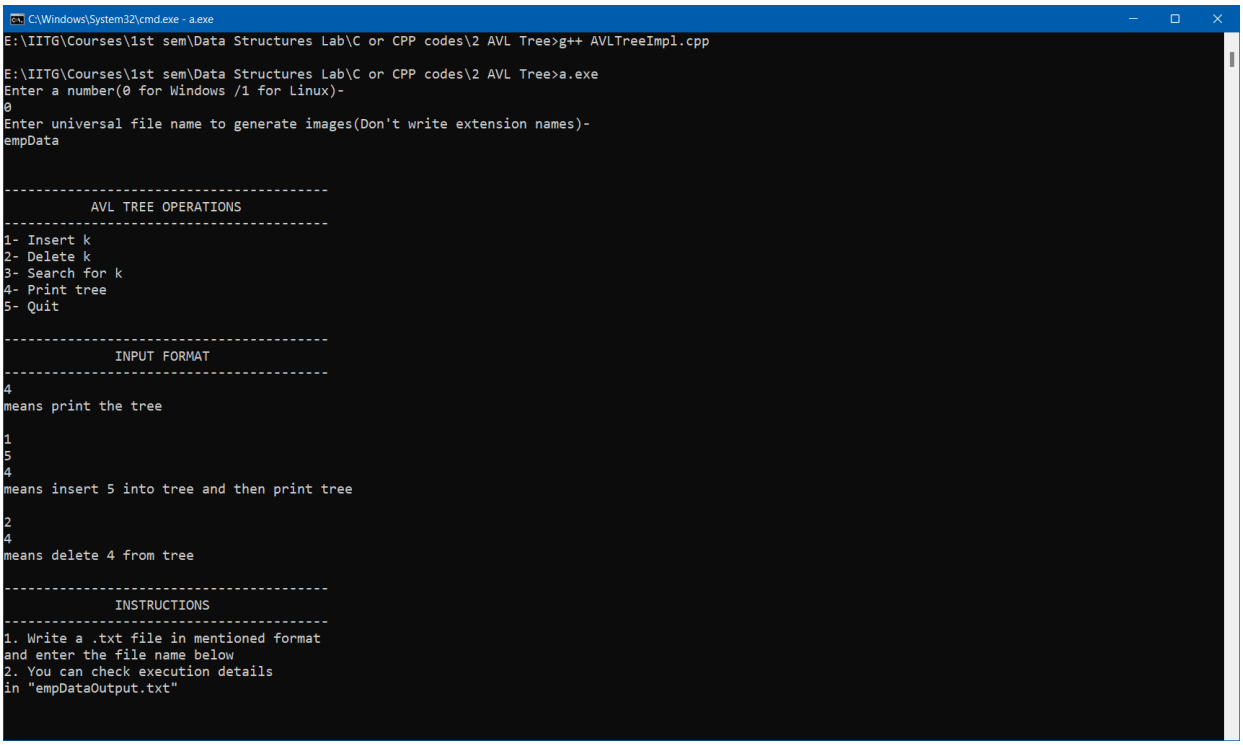


Figure 3: Sample prompt

2.6 functions.h

This file contains some utility functions which are being used multiple times like generating .png files or printing a long message.

3 How to create test cases

3.1 User prompt

After executing the program, a prompt will ask the user to enter a number indicating the current operating system. Then a prompt will be printed mentioning the sequence of operations and some small example test cases. It will ask the user to write a file name. So here the user needs to create a .txt file with a sequence of integers and save it in the same directory. Then the same file name should be given in the prompt. A sample prompt is shown in **Figure 3**. If the last instruction is not to quit the execution, after executing the current .txt file the same prompt will reappear for further execution. A new .txt file can be created and the name can be given. For quitting the execution the last instruction in the .txt file should be 5. Also, "quit" or "exit" or similar keywords can be given instead of file name to quit the execution. Currently added keywords are - quit, exit, stop, return. More can be added at line number - 55 in AVLTreeImpl.cpp . Similarly, for generating images midway, "view" or "print" or similar keywords can be given instead of file name. Currently added keywords are - image, view, print, visualize, run, picture. More can be added at line number - 56 in AVLTreeImpl.cpp .

3.2 Example testcase

1
9
1
5
1
15
1
3
1
7
1
11
1
17
2
15
3
5
4
5

The above sequence can be written as a test case in an input file, which indicates the following.
1 (Insert the next value)
9
1 (Insert the next value)
5
1 (Insert the next value)
15
1 (Insert the next value)
3
1 (Insert the next value)
7
1 (Insert the next value)
11
1 (Insert the next value)
17
2 (Delete the next value)
15
3 (Search the next value)
5
4 (Print the tree)
5 (Quit)

4 Instructions to execute the code

1. If you're using Dev C++, Follow the steps to support *unordered_map*. Go to tools–compiler option–general tab–tick mark option (add the following commands when calling compiler)–add -std=c++11 there. Then build and execute the project.

2. In linux or GNU windows compiler, open the terminal or command prompt and type "g++

AVLTreeImpl.cpp" and hit enter.

3. Then for linux type "./a.out". For windows type "a.exe" or "AVLTreeImpl.exe" (One of them should work). Hit enter.
4. You can also write "./AVLTreeImpl" and hit enter to execute the makefile which is provided.
5. Now the prompt will be displayed. Enter 0 or 1 for windows or linux respectively.
6. Give a name for all the files which are going to be generated during execution.
7. Write a .txt file according to the given format and enter the file name there. If you give wrong sequence, it can destroy the execution.
8. You can write "quit" or "view" instead of file name to exit from the program or to generate the images till now respectively.
9. The entire execution process can be visualized in output.txt.
10. After printing a tree, you can view the images in the same directory. The respective file names are written in output.txt.
11. You can give multiple .txt files one after another. For example- in the first run, you can insert a few nodes and delete one of them to check how the tree rotates, and after seeing the result you can write another .txt file to delete the root node.
12. 5 is used for quit. Do not write 5 unnecessarily inside the .txt file. It is better to keep it at the end.
13. You can run the batch file or shell script to manually generate the images.

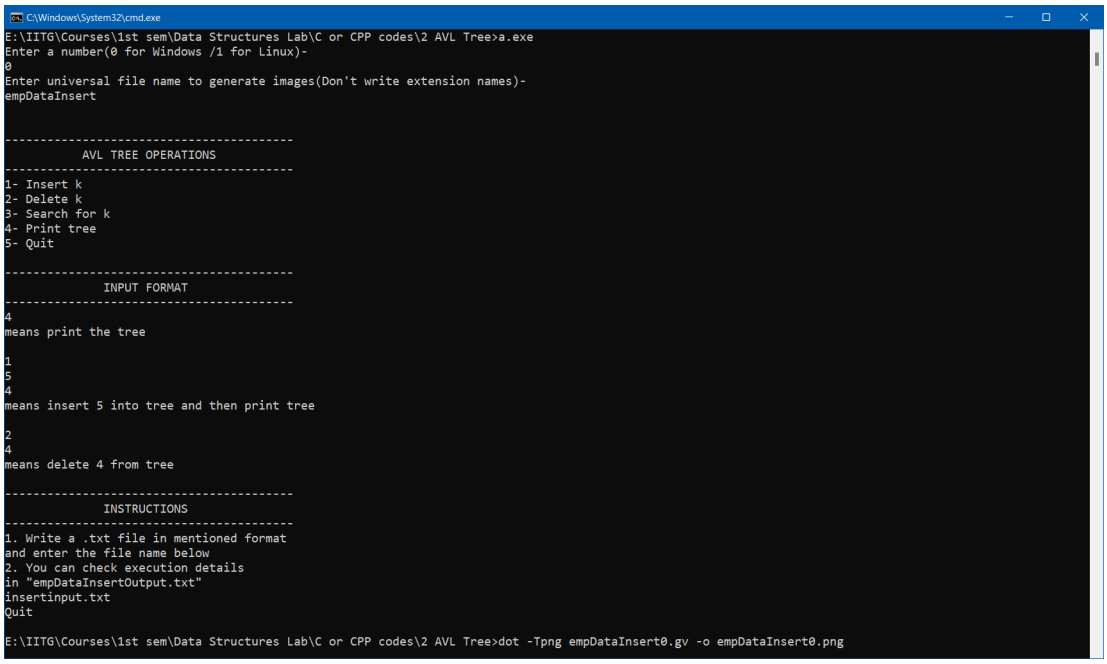
5 Testing

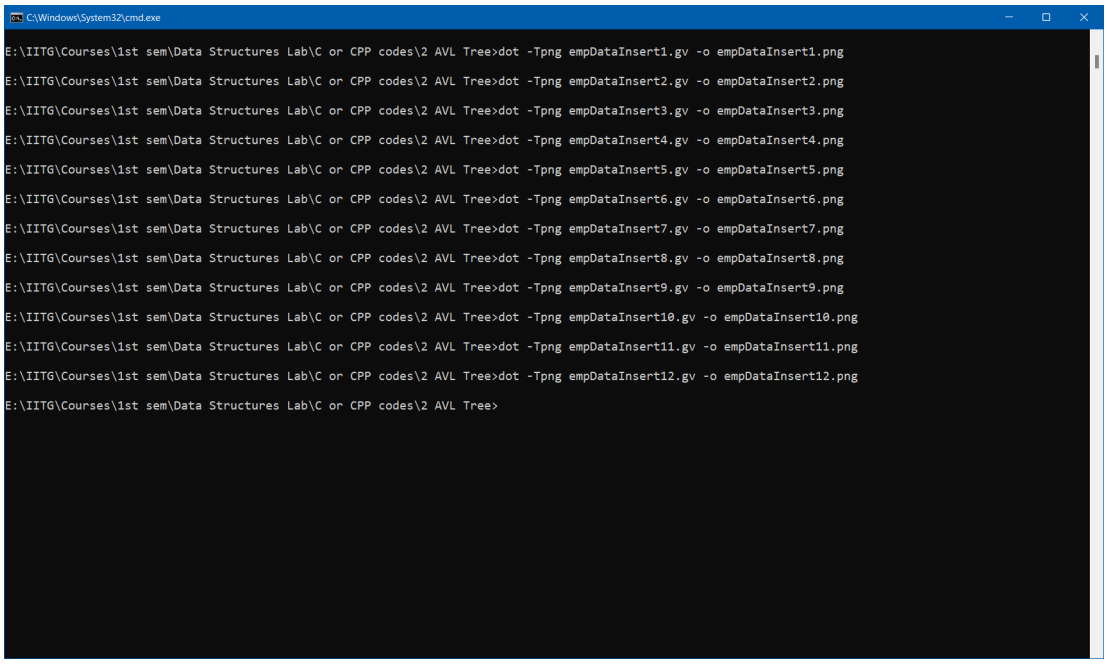
The input files, output files, graphviz files, png files, console sequence for insertions and deletions are attached in "insert cases" and "delete cases" folders respectively. Those files can be referred for better understanding. Here only a few cases are mentioned by covering all possible cases.

5.1 insertinput.txt

1. Insert $21, 26, 30, 9, 4, 14, 28, 18, 15, 10, 2, 3, 7$ in that order.
2. Print the tree after every insertion.
3. Quit

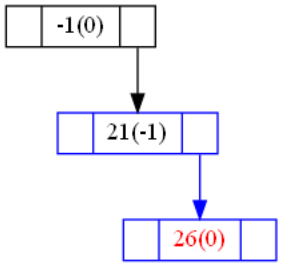
5.2 Console output



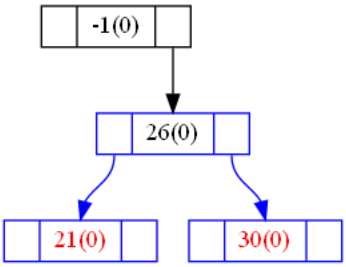


5.3 Various cases during insertion of this sequence

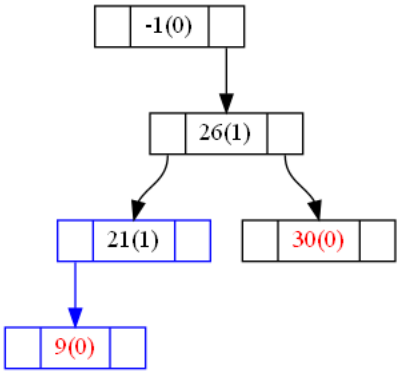
5.3.1 *RR*-imbalance



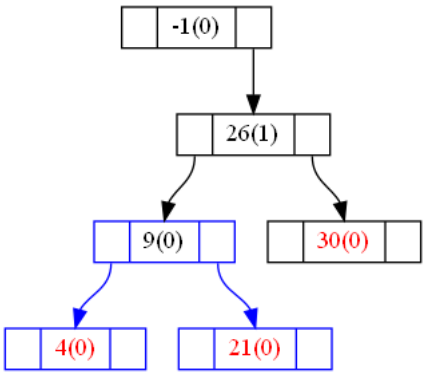
After Inserting 30 into it-



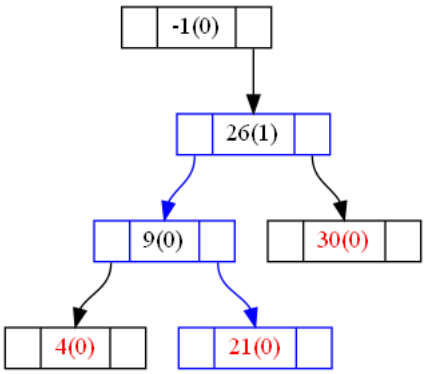
5.3.2 *LL*-imbalance



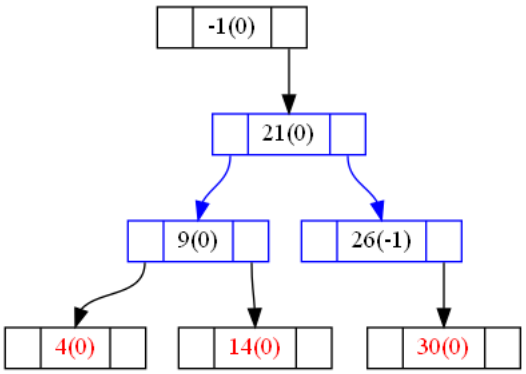
After inserting 4 into it-



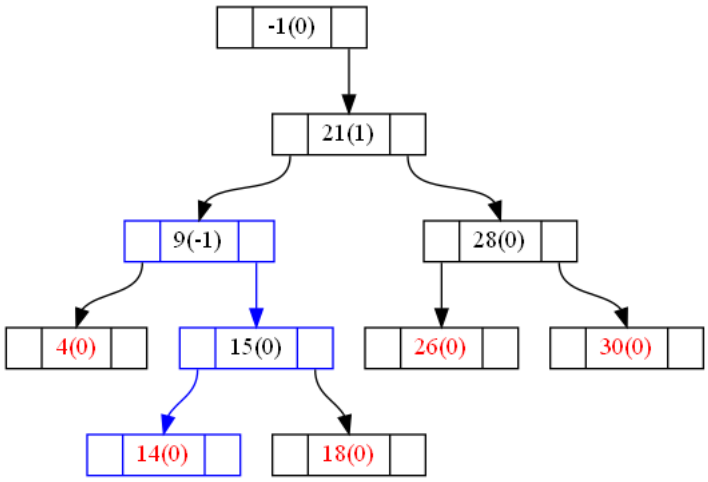
5.3.3 *LR*-imbalance



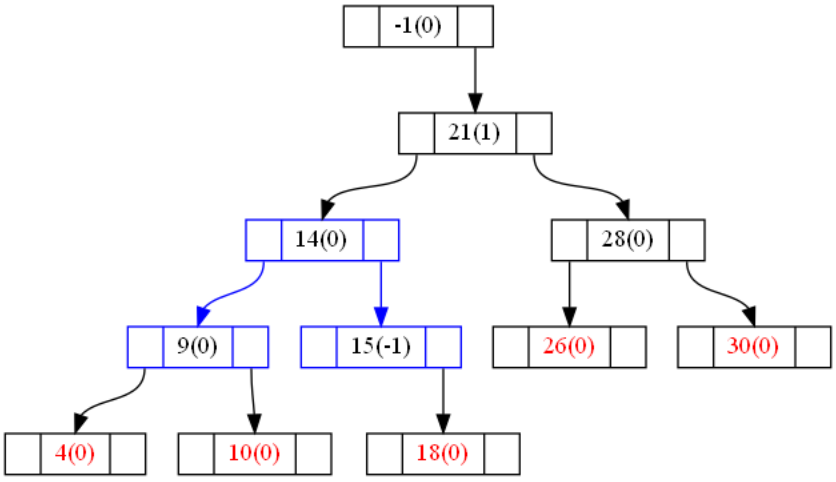
After inserting 14 into it-



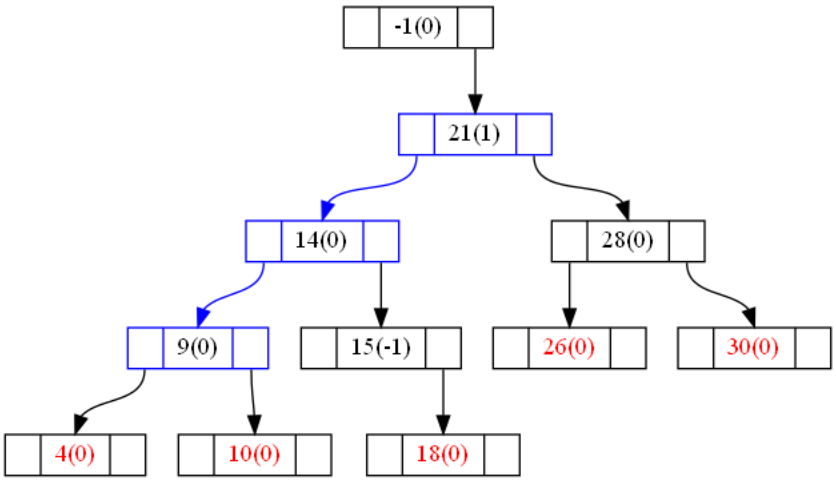
5.3.4 *RL*-imbalance



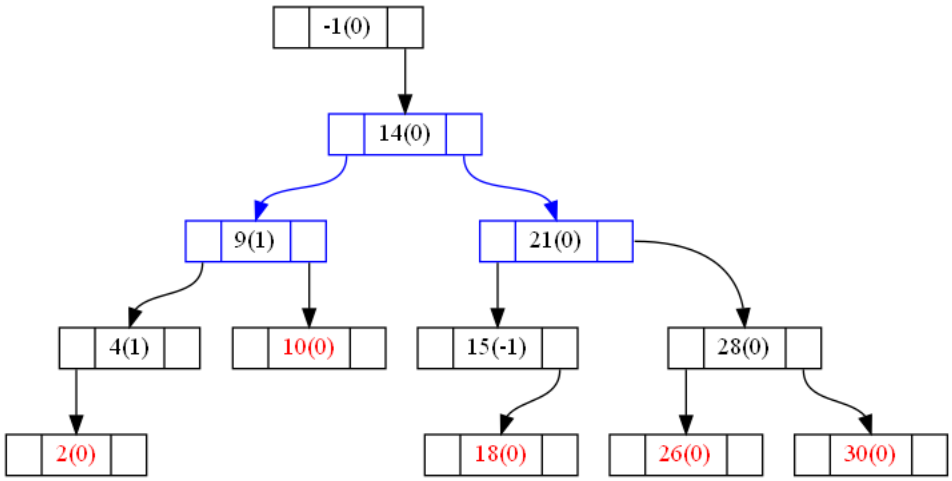
After inserting 10 into it-



5.3.5 A case which involves rotation at root node



After inserting 2 into it-



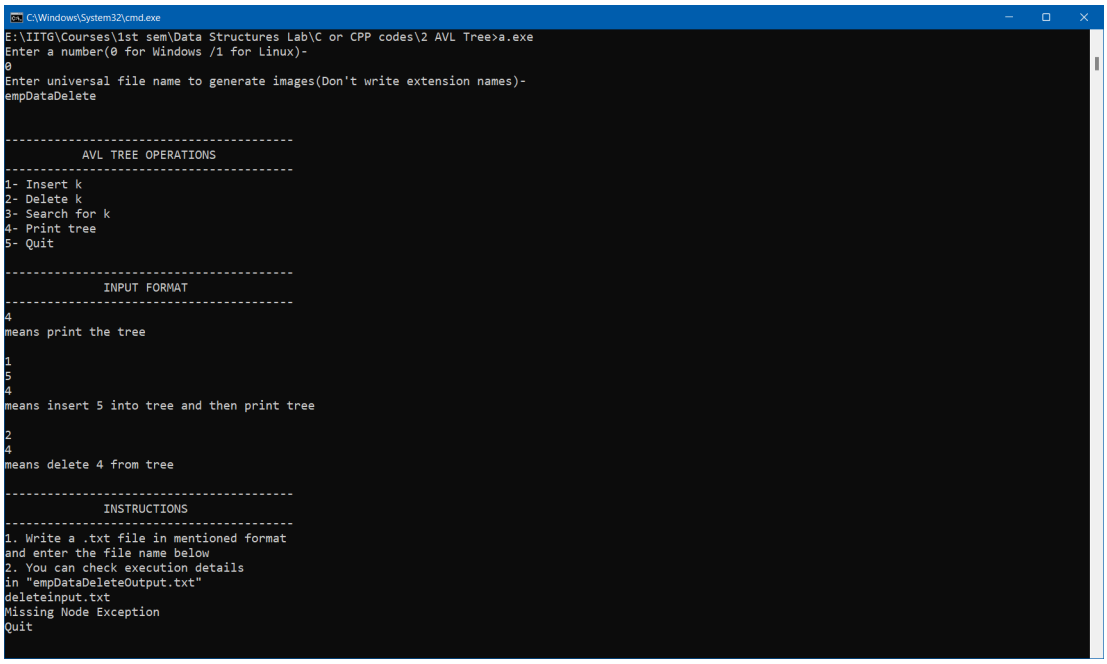
5.4 deleteinput.txt

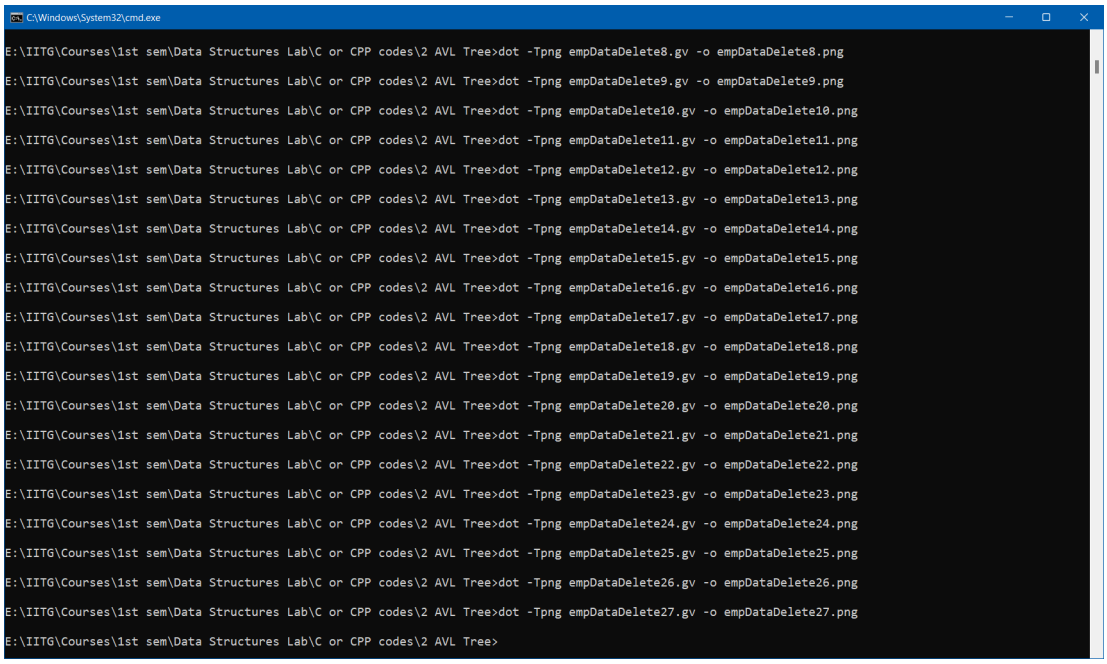
Here the insertions are not tested and the nodes are inserted in level order. For covering all kinds of cases, an AVL Tree with minimum possible nodes has been considered.

1. Insert $60, 50, 80, 20, 55, 75, 82, 10, 32, 53, 56, 73, 76, 81, 83, 7, 12, 30, 40, 52, 54, 57, 72, 74, 78, 84, 6, 8, 14, 45, 51, 71, 5$ in that order.
2. Print the tree.
3. Delete 5.
4. Print the tree.
5. Delete 71.
6. Print the tree.
7. Delete 76.
8. Print the tree.
9. Delete 78.
10. Print the tree.
11. Delete 80.
12. Print the tree.
13. Delete 73.
14. Print the tree.
15. Delete $6, 8, 14$ in that sequence.
16. Print the tree.
17. Delete 20.
18. Print the tree.
19. Insert 5.
20. Print the tree.
21. Delete 53.
22. Print the tree.
23. Delete $32, 45, 54, 75, 82, 51, 84, 72$ in that order.
24. Print the tree.
25. Delete 50.
26. Print the tree.
27. Delete 30.
28. Print the tree.
29. Delete 83.

30. Print the tree.
31. Delete 81.
32. Print the tree.
33. Delete 74.
34. Print the tree.
35. Delete 60.
36. Print the tree.
37. Delete 57.
38. Print the tree.
39. Delete 56.
40. Print the tree.
41. Delete 56.
42. Print the tree.
43. Delete 55.
44. Print the tree.
45. Delete 52.
46. Print the tree.
47. Delete 40.
48. Print the tree.
49. Delete 12.
50. Print the tree.
51. Delete 10.
52. Print the tree.
53. Delete 7.
54. Print the tree.
55. Delete 5.
56. Print the tree.
57. Quit

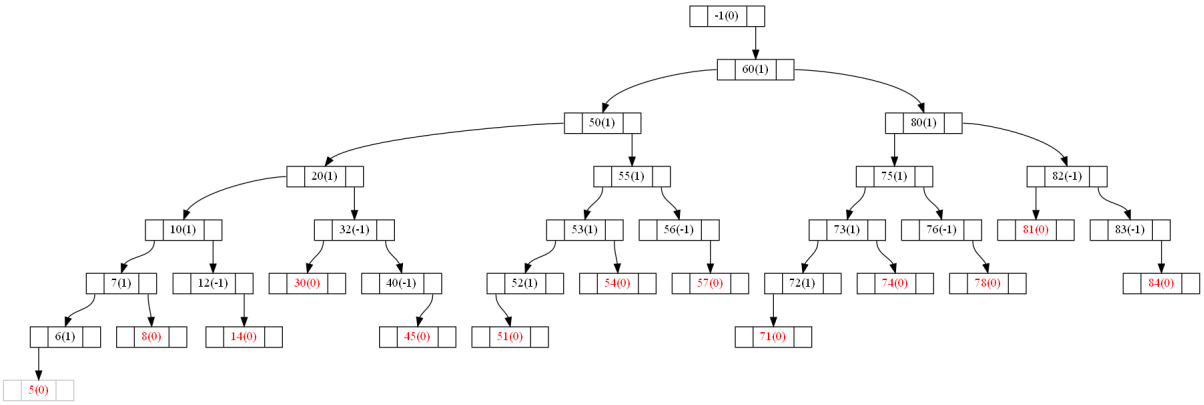
5.5 Console output



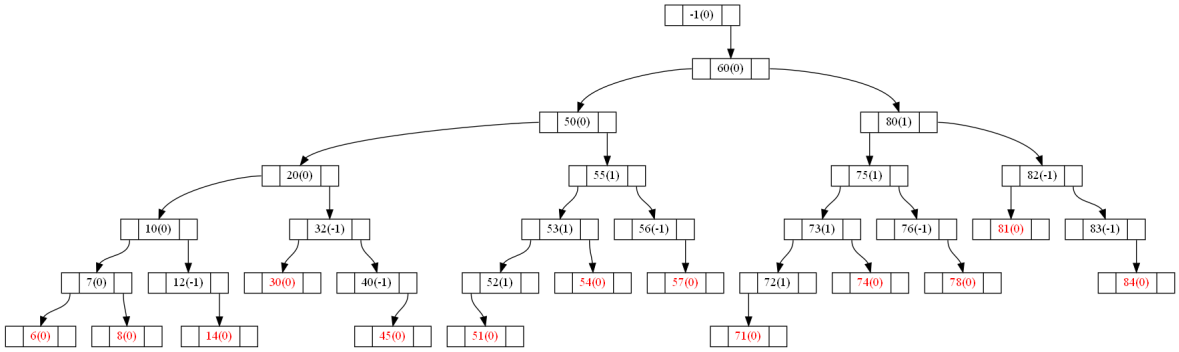


5.6 Various cases during deletion of this sequence

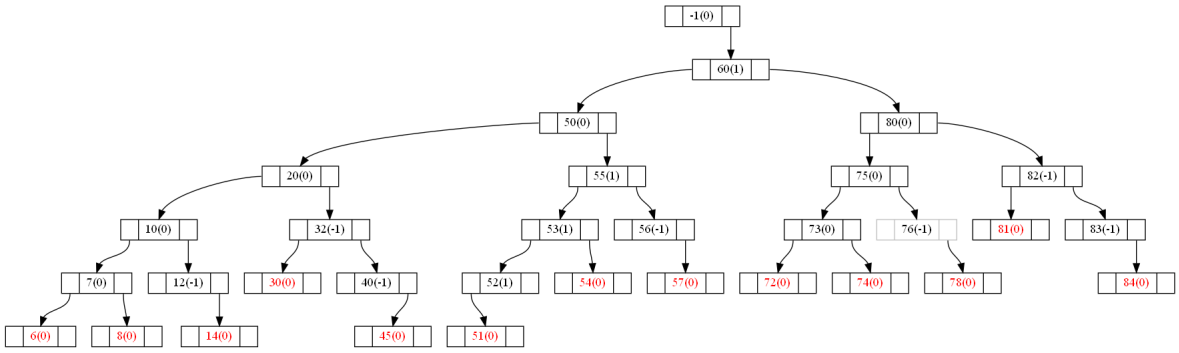
5.6.1 Deleting the smallest node which is leaf



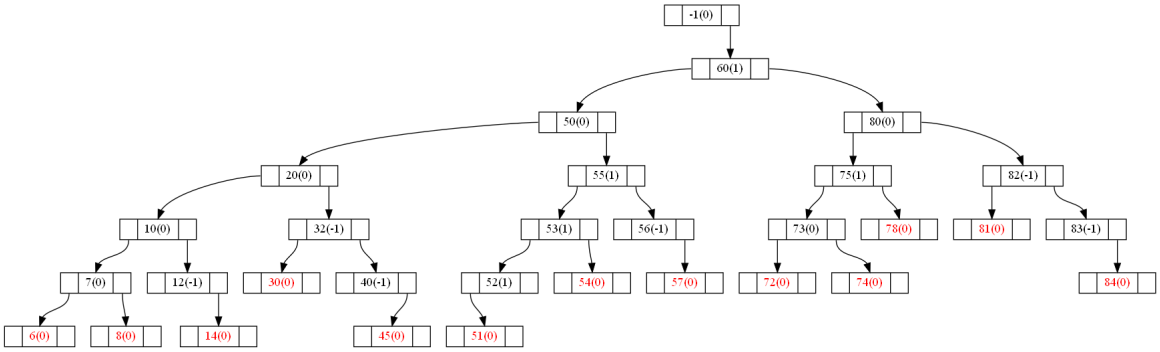
After Deleting 5 from it-



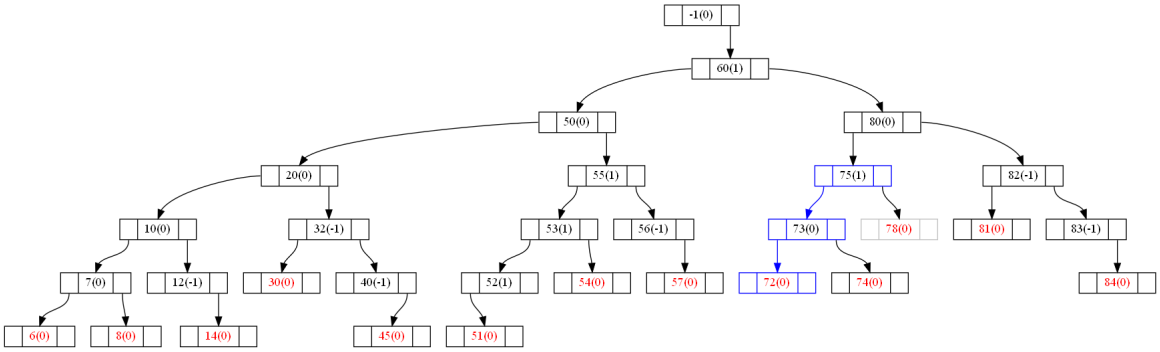
5.6.2 Deleting a node with single child



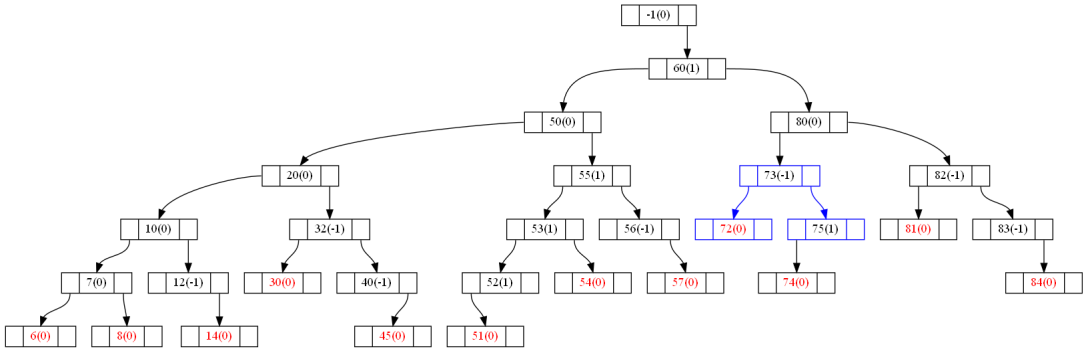
After Deleting 76 from it-



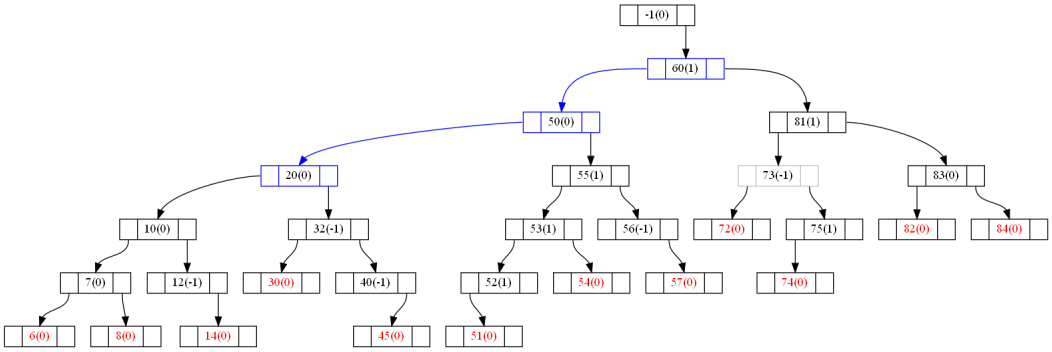
5.6.3 *LL*-imbalance which performs single rotation where balance factor of *rotate* is 0



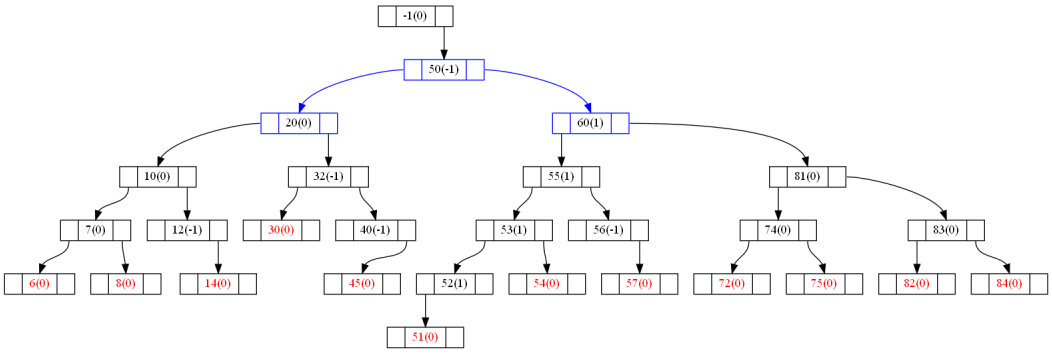
After Deleting 78 from it-



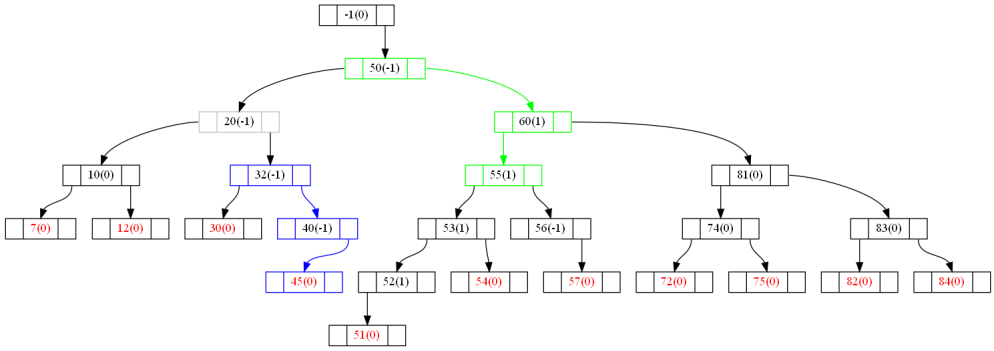
5.6.4 *LL*-imbalance which results in single rotation at root



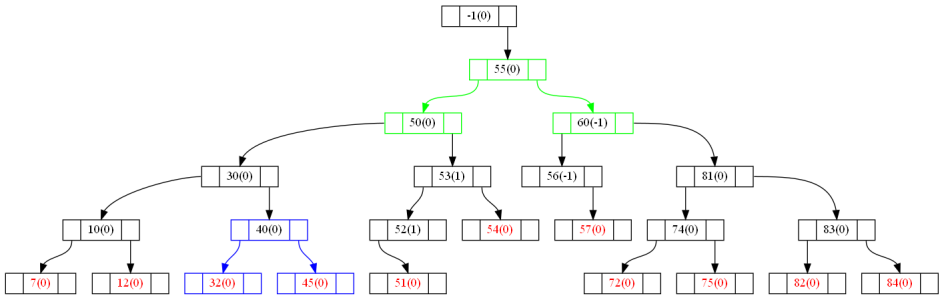
After Deleting 73 from it-



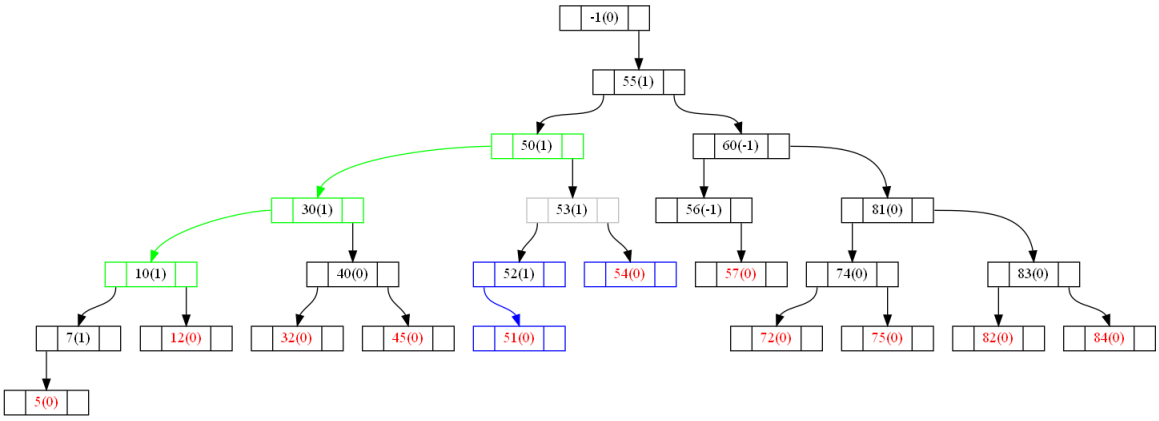
5.6.5 *RR*-imbalance followed by *RL*-imbalance which results in double rotation at root



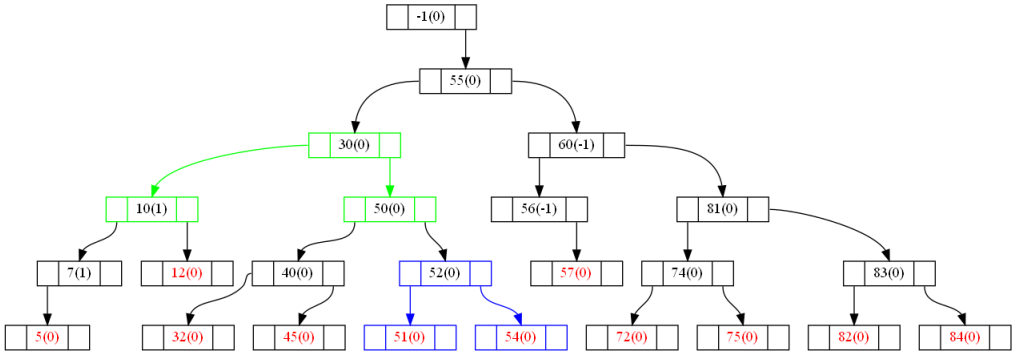
After Deleting 20 from it-



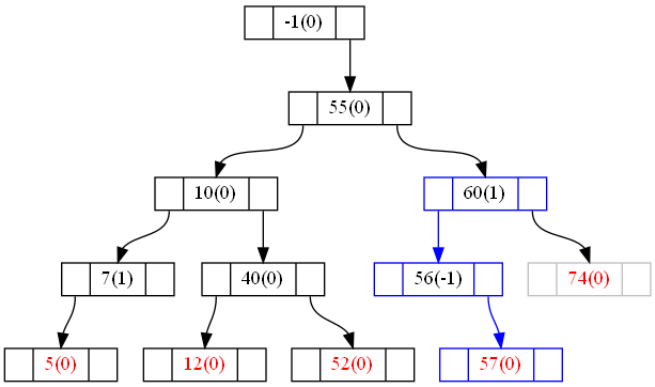
5.6.6 two LL-imbalances



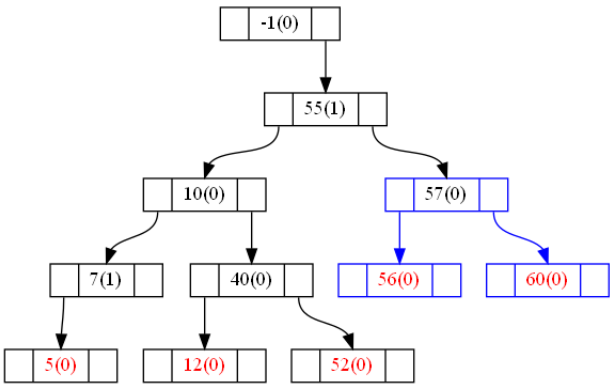
After Deleting 53 from it-



5.6.7 LR-imbalance



After Deleting 74 from it-


-1(0)
55(1)
10(0)
57(0)
7(1)
40(0)
56(0)
60(0)
5(0)
12(0)
52(0)