

# AVL Trees

Ilia Ilmer

November 11, 2019

## Abstract

AVL-trees are an efficient improvement of binary search trees. One advantage of this data structure is that it is a self-balancing tree. AVL-trees will have more consistent behavior due to the balance maintenance ensuring the running times of insertion, deletion, and searching are kept at  $\mathcal{O}(\log n)$  where  $n$  is the number of nodes. In this project, a C++ implementation of AVL-trees is presented together with description of its native operations. Analysis of the implementation confirms the efficiency of all operations on large trees (up to  $10^7$  nodes).

## 1 Introduction

For practical problems of computer science, it is essential to find a way to efficiently store data. Efficiency can be characterised by many factors, such as speed of finding, adding, or deleting an item from the data set. One of the first structures studied in any computer science classes is an array. In its simplest form of a one-dimensional sequence of numbers <sup>1</sup> it can be an efficient way to access data quickly once the position of the desired element is known. However, as sizes of arrays grow, so do time requirements for performing operations of insertion and finding.

Consider an example: an array with  $n = 10^6$  items is read in an unsorted order. The task is to insert a new element into the array in such a way that it is surrounded by its closest neighbors (i.e. it is inserted in a sorted order, if we want to insert 4 we want it to be between 3 and 5, etc.) The task can be solved by simply sorting the array first which takes  $\mathcal{O}(n \log n)$ . If the array is not filled to the maximum then we can relatively inexpensively in terms of time insert the value (worst-case linear time). The issue can arise from the memory requirements: in the worst case, we have to shift  $\mathcal{O}(n)$  numbers in the array to insert the new one correctly. And things could get worse if the array is full. We would have to copy all items into a new memory location first and then figure out the insertion. This requires additional  $\mathcal{O}(n)$  both in memory and in running time.

A data structure called binary search trees remedy the situation. The idea is to store the values in a hierarchical way so that one could access the next value only going through previous ones. Binary search trees are a particular instance of graphs, so each item in it is called a node and each node stores a pointer to a left (right) child and a parent (a node above). Additionally, the elements are stored according their value: a node on the left (right) stores a number lower (greater) than its parent. An example is presented in Figure 1. Binary trees solve the problem with data access that arrays have. Given a

---

<sup>1</sup>We will focus on numbers throughout this work for the sake of simplicity, however, other data types could be stored as well.

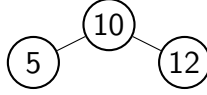


Figure 1: Binary tree, a family-friendly example.

binary tree on  $n = 10^6$  nodes, assume we want to insert a new value. We no longer need to copy any data; by simply following the tree items from the root to the next node we can find the position that a new item satisfies (by comparing its value to the values of other nodes). Since every time we either go left or right we discard (roughly)  $\frac{n}{2}$  candidates. This will make the total runtime  $\mathcal{O}(\log n)$  where  $\log n = 6$  (assuming base 10).

We conclude that binary trees can drastically improve the runtime from  $\mathcal{O}(n \log n)$  to simply  $\log n$ .

## 2 Problem Motivation

But this is not always an ideal solution. A problem may arise when a binary tree is created by the following method: the elements added to a tree are always less than the root and are being added in the sorted order. Following the example above, consider Figure 2. We add numbers  $\{13, 14, 15\}$  to the tree one at a time. What happens is that the tree becomes imbalanced: one half of the tree contains significantly more nodes than the other. In the worst case, one can have all  $n$  nodes skewed to one side. This will make search, insertion, and deletion run in  $\mathcal{O}(n)$ .

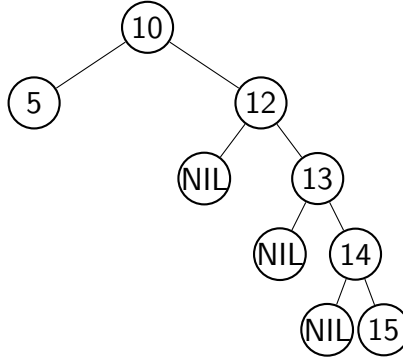


Figure 2: Binary tree, an imbalanced example.

A self-balancing data structure called AVL-tree addresses the issue. This type of binary tree was introduced in 1962 by Russian mathematicians Adeson-Velsky and Landis [3, 1]. AVL-trees are based off of binary trees with the main difference being that one keeps track of the difference between subtree heights for every node. Let us call this value balance factor (or, simply, balance). For a given node  $u$ , a *balance factor* is the difference between heights of a subtrees rooted at left and right children of  $u$ . AVL-trees demand a constraint on balance  $b$  as follows

$$|b| \leq 1. \tag{1}$$

If we were to transform the tree on Figure 2, we can obtain an AVL-tree, see Figure 3.

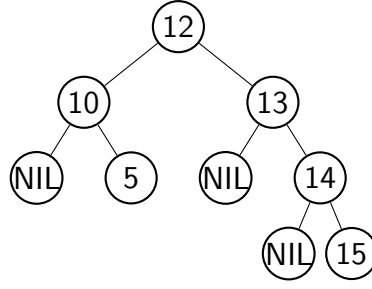


Figure 3: An AVL-tree obtained from modifying a binary search tree on Figure 2. The height of tree rooted at 10 is 1, that of node 13 is 2, the balance is -1 which satisfies the balance constraint of AVL-trees.

## 3 Related Data Structures

### 3.1 Red-Black Trees

AVL-trees are not the only solution to tree balance problem. Red-Black trees are binary trees with self-balancing properties. Each node has a binary property "color" that can be either "Red" or "Black". All Red-Black trees must satisfy the following rules:

1. Every node is either red or black.
2. The root is black.
3. All leaves are black.
4. If a node is red, all its children are black.
5. Every path from a given node to any of its descendant NIL nodes contains the same number of black nodes.

The height of a Red-Black tree on  $n$  nodes is at most  $4 \log(n + 1)$  [4, Chapter 13].

### 3.2 B-trees

Another example of self-balancing trees are B-trees [4, Chapter 18]. This type of trees is defined as follows.

1. Every node  $x$  carries the number  $n$  of keys (values) stored in  $x$ , the keys  $x.key_i, i = 1, \dots, n$ , a flag  $x.leaf$  that stores true if  $x$  is a leaf;
2. Each internal node  $x$  stores  $n + 1$  pointers  $c_i, i = 1, \dots, n$  to its children
3. For each key  $k_i$  in the subtree with root  $x.c_i$ ,

$$k_i \leq x.key_i \leq k_{i+1}, i = 1, \dots, n.$$

4. All leaves have the same depth,  $h$
5. Given a number  $t$  called minimum degree, the following properties hold. Every node other than the root must have at least  $t - 1$  keys. Every internal node, other than the root has at least  $t$  children. A tree that is not empty must have at least one key.

6. Every node may contain at most  $2t - 1$  keys. An internal node can have at most  $2t$  children.

B-trees allow same kinds of operations, such as search, insertion, deletion. These operations are very efficient, for instance the search operation can run in  $\mathcal{O}(th) = \mathcal{O}(t \log_t h)$  where  $h$  is the height of the tree. Same running time asymptotic is true for insertion.

### 3.2.1 2-3-4 trees

A specific case of B-trees are 2-3-4 trees when  $t = 2$  [4, Chapter 18].

## 3.3 Weight-balanced Trees

Weight-balanced trees are balanced according to the weight which is defined as the number of leaves. The number of leaves in the left and right subtrees should be balanced [2].

## 4 Applications

AVL-trees are some of the oldest self-balancing trees. Due to limitations of computing power in 1960s, they found little application at the time of invention [2]. The interest towards this method of data handling was revived as the computational powers grew.

A naive implementation of AVL trees can be expensive in the sense that operations that require adjustment of nodes to keep the balance factor  $b$  within the  $\{\pm 1, 0\}$  constraint can be implemented inefficiently. If we do not keep track of and store every node's height, we might end up making insertion and deletion polynomial in time. Concretely, if getting a height of a node at level  $i$  requires  $i \log i$  operations, insertion could contain  $\sum_i i \log i$  term which worsens the usual  $\mathcal{O}(\log n)$  efficacy. Such naive implementation of AVL-trees would be advantageous only when insertion or deletion operation are much less frequently used than lookup. The latter is identical to that of a regular binary search tree and therefore is inexpensive, since it does not require any maintenance. In this work, we keep track of heights, so efficiency is always  $\mathcal{O}(\log n)$  for a tree of size  $n$ .

## 5 Examples

In this section, we will go through some of the native operations of AVL-trees. We focus mostly on insertion and deletion while the search operation, due to being identical to that of regular binary trees, can be discussed in fewer details. For the rest of this section, let us use the following array

$$A = \langle 1, 2, 3, 4, 5, 6, 7, 8, 9 \rangle.$$

### 5.1 Insertion

Denote an empty AVL-tree by  $T$ . We will begin insertion assigning  $T.root = Node(A[1])$ , which assigns root of the tree to be a node with key  $A[1] = 1$ . This is presented on Figure 4. This is a trivial example, the tree of height 0, no balance adjustment is required. Next, we add  $A[2]$ . We should add it as a right child of the root, thus creating a tree on Figure 5. Once again, no rebalancing required. Let us add the value  $A[3] = 3$ . We will see that no subtree of any node requires rebalancing. On Figure 6 we show two insertions,



Figure 4: Single root AVL-tree.

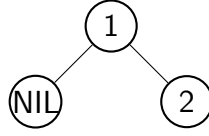


Figure 5: Two-value tree.

$A[3]$ ,  $A[4]$ . We notice a problem with the tree on the right of that figure: the balance factor if computed at the root, is

$$b = \text{Height}(\text{NIL}) - \text{Height}(A[2]) = 0 - 2 = -2,$$

which breaks the requirement  $|b| \leq 1$ .

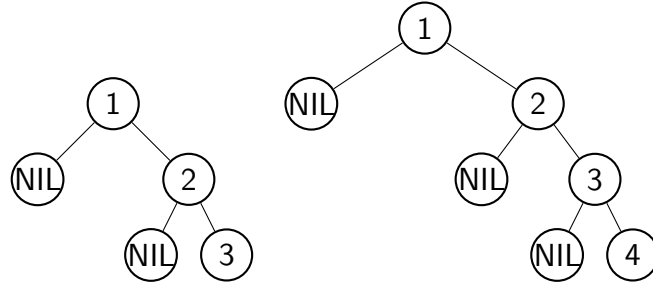


Figure 6: Left tree is the result of inserting  $A[3]$  as a binary tree fashion. Right tree is the result of inserting  $A[4]$ .

To fix that issue, we use the concept of rotations. We will use the left rotation in this case, see figure Figure 7.

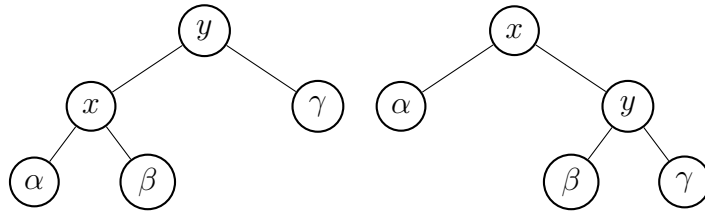


Figure 7: To go from configuration on the left to that on the right we apply right rotation, to go back we apply left rotation.

Next, let us present algorithms LEFT-ROTATE and RIGHT-ROTATE.

LEFT-ROTATE( $T, x$ )

```

1   $y = x.right$ 
2   $x.right = y.left$ 
3  if  $y.left \neq T.nil$ 
4      // check if  $y$ 's left child is not a NIL
5       $y.left.p = x$ 
6   $y.p = x.p$ 
7  if  $x.p == T.nil$ 
8      // check if  $x$ 's parent is a NIL
9       $T.root = y$ 
10 elseif  $x == x.p.left$ 
11      $x.p.left = y$ 
12 else  $x.p.right = y$ 
13  $y.left = x$ 
14  $x.p = y$ 

```

RIGHT-ROTATE( $T, y$ )

```

1   $x = y.left$ 
2   $y.left = x.right$ 
3  if  $x.right \neq T.nil$ 
4       $x.right.p = y$ 
5   $x.p = y.p$ 
6  if  $y.p == T.nil$ 
7       $T.root = x$ 
8  elseif  $y == y.p.right$ 
9       $y.p.right = x$ 
10 else  $y.p.left = x$ 
11  $x.right = y$ 
12  $y.p = x$ 

```

Applying left rotation to the right-side tree in Figure 6, we obtain

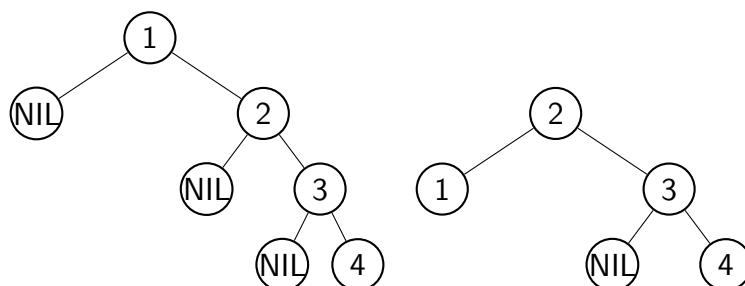


Figure 8: Left: before LEFT-ROTATE. Right: after LEFT-ROTATE.

In order to rebalance the tree, we needed to apply such kind of rotation that the excess would move from the right to the left. Negativity of the balance factor  $b$  reflects how much the right subtree "outweighs" the left one.

Let us consider the following scenarios (in all figures below the node's label signifies the height of the tree rooted at this node). The first scenario is, as we described, when the right subtree outweighs the left one, see Figure 9.

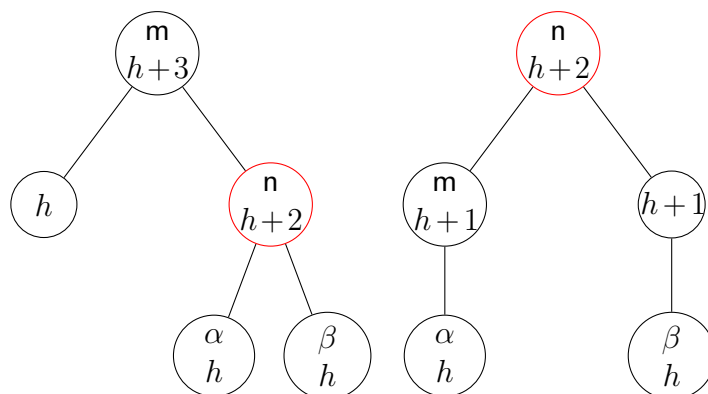


Figure 9: Applying left rotation to a tree that is biased to the right, that is, for a given root, the right subtree greater height. Left: before rotation. Right: after rotation.

By the symmetry, we can easily construct an example where a right rotation should be applied. Instead, let us discuss a more interesting scenario. Assume a leaf has been added to a tree where nodes  $x$ ,  $y$  are, see Figure 10. This will require a combination of rotations. First of all, notice that the height of  $c$  will change thus causing imbalance that can be resolved by a right rotation for  $b$  and  $c$ , Figure 11.

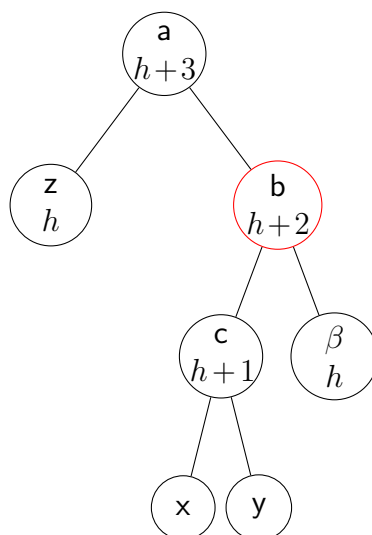


Figure 10: Example of leaf insertion. Node  $c$  will change its height and hence will require a right rotation.

After that, we apply left rotation on  $a$  and  $c$  to bring the balance back to the tree, Figure 12. This situation lowers (possibly) the heights of  $z$  and  $\beta$ , but the important fact is that the balance is restored. This sequence is called **RIGHT-LEFT-ROTATION**.

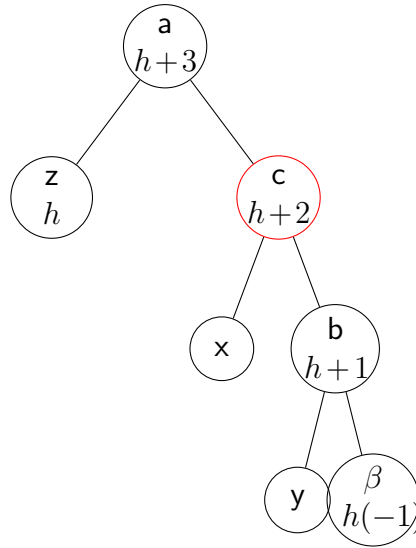


Figure 11: Example of leaf insertion. Node  $c$  changed its place after right rotation. The imbalance is still present.

Let us give the full pseudocode of insertion operation.

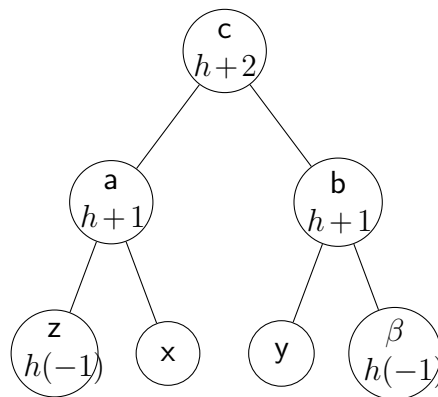


Figure 12: Final result of leaf insertion.



AVL-TREE-INSERT(*root*, *x*)

```
1  if root == NIL
2      return node x
3  // Below follows the search for the new node's rightful place
4  // We do not allow equal keys
5  if x.key < root.key
6      root.left = AVL-TREE-INSERT(root.left, x)
7      root.left.p = x
8  elseif x.key > root.key
9      root.right = AVL-TREE-INSERT(root.right, x)
10     root.right.p = x
11  else
12      return root // Node is already present
13  root.UPDATE-HEIGHT() // Keeping track of height,  $\mathcal{O}(1)$ 
14  // Begin rebalancing.
15  balance = root.GET-BALANCE()
16  if balance > 1 and root.left  $\neq$  NIL and x.key < root.left.key
17      // Imbalance in the left subtree
18      RIGHT-ROTATE(ROOT)
19  if balance > 1 and root.left  $\neq$  NIL and x.key > root.left.key
20      // Imbalance in a right subtree of a left descendant
21      LEFT-ROTATE(ROOT)
22      RIGHT-ROTATE(ROOT)
23  if balance < -1 and root.left  $\neq$  NIL and x.key < root.left.key
24      // Imbalance in a left subtree of a right descendant
25      RIGHT-ROTATE(ROOT)
26      LEFT-ROTATE(ROOT)
27  if balance < -1 and root.left  $\neq$  NIL and x.key > root.left.key
28      // Imbalance in the right subtree
29      LEFT-ROTATE(ROOT)
```

## 5.2 Deletion

Let us now discuss the deletion algorithm. The simplest way to describe deletion in AVL-trees is to recall the same procedure in regular binary trees. We first perform regular deletion ignoring balance. After that, we begin rebalancing where required, going from bottom to the top in the recursion tree. For simplicity, we provide only the rough outline of the algorithm.

AVL-TREE-DELETE(*root*, *key*)

- 1 Find the node with the required key (recursively)
- 2 Perform the deletion with transplantation ([4, Chapter 12])
- 3 Rebalance as in the AVL-TREE-INSERTION algorithm

Transplant operation is described in [4, Chapter 12] together with deletion for regular binary trees.

## 6 Analysis

In this section we provide two types of analysis: theoretical and practical. Theoretical analysis is rather straightforward to perform. The practical part will be analyzed using code.

### 6.1 Theoretical

#### 6.1.1 Insertion analysis

Consider the complexity of insertion procedure. First, notice that for a given AVL-tree, its height satisfies the following theorem.

**Theorem 1** ([1]). For a tree on  $N$  nodes, its height does not exceed  $1.5 \log_2(N + 1)$ .

The proof reduces to considering the fact that, given an AVL-tree on  $N_n$  nodes, we can derive:

$$N_h = N_{h-1} + N_{h-2} + 1, \quad (2)$$

which generates a Fibonacci sequence. Solution to this equation is will contain the famous golden ratio in the form  $\phi^h$ . This will yield [1]

$$h < \log_\phi(N + 1) < 1.5 \log_2(N + 1).$$

Thus, any traversal during insertion will be  $\mathcal{O}(\log(n))$ , where  $n$  is the number of nodes. Each rotation is a constant-time memory manipulation. This means that each time we go up from recursion, rebalancing will happen in constant time. Therefore, we rebalance  $\mathcal{O}(\log n)$  in time and the AVL-TREE-INSERT procedure takes  $\mathcal{O}(\log n)$  in time.

#### 6.1.2 Deletion analysis

Analysis of AVL-TREE-DELETE can be simply broken into parts as follows. Line 1 of the algorithm performs regular binary search, which is  $\mathcal{O}(\log n)$ . Further, the actual deletion is performed in  $\mathcal{O}(1)$ . Rebalancing on the way back up from recursion is going to take at least as much time as the search and in the worst case it is  $\mathcal{O}(\log n)$ .

### 6.2 Practical

We construct the tree by simply iterating through numbers between 1 and  $n$ . This process takes the longest out of the rest of the program,  $\mathcal{O}(n)$ . After that we check insertion by adding a pseudorandom number from the specified range. From Table 1 we can see that even at the sizes of  $10^6$ , the runtime is relatively small for the operations. This confirms our theoretical analysis of these procedures' runtimes.

n	AVL-TREE-INSERT ( $\mu s$ )	AVL-TREE-DELETE ( $\mu s$ )	SEARCH
10	0	0	0
$10^2$	1	1	1
$10^3$	3	3	3
$10^4$	1	1	1
$10^5$	3	3	3
$10^6$	15	15	15
$10^7$	7	7	7

Table 1: Runtimes in microseconds for each operation of the AVL-Tree. The efficiency of the operations is evident, even at sizes of  $10^7$ , we still cannot reach the runtime higher than order of several microseconds.

## References

- [1] George M Adel'son-Vel'skii and Evgenii Mikhailovich Landis. "An algorithm for organization of information". In: *Doklady Akademii Nauk*. Volume 146. 2. Russian Academy of Sciences. 1962, pages 263–266.
- [2] Peter Brass. *Advanced data structures*. Volume 193. 2008.
- [3] Donald E Knuth. *The art of computer programming, vol. 3: Searching and sorting*. 1973.
- [4] Charles Eric Leiserson, Ronald L Rivest, Thomas H Cormen, and Clifford Stein. *Introduction to algorithms*. Volume 6. MIT press Cambridge, MA, 2001.

## A Code

Below we present code for each part of the program. We begin by showing that the program works. We will show the output first and then the code for the data structure.

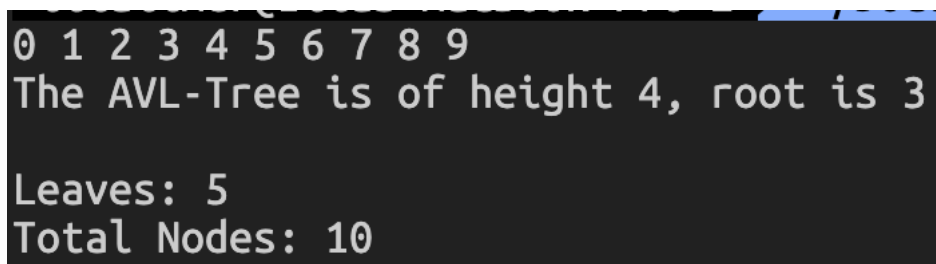
The main program consists of multiple parts. In the first part, the program takes a number of nodes  $n$  as a command line input and creates an instance of an AVL-tree with keys 1 through  $n$ . It then outputs the tree using the in-order traversal.

Listing 1: The beginning part of the main program.

---

```
1 #include "avltree.h"
2 //... other #include's
3 int main(int argc, char const *argv[])
4 {
5     Node *root = NIL;
6
7     AVLTree tree = AVLTree(root);
8
9     int operation_count = 0;
10    for (size_t i = 0; i < atoi(argv[1]); i++)
11    {
12        tree.root = tree.insert(tree.root, i, &operation_count);
13    }
14
15    operation_count = operation_count / atoi(argv[1]);
16    tree.inorder_tree_walk(tree.root);
17    printf("\nThe AVL-Tree is of height %d, root is %d\n",
18        tree.root->height, tree.root->get_val());
19    //... program continues
20 }
```

---



```
0 1 2 3 4 5 6 7 8 9
The AVL-Tree is of height 4, root is 3

Leaves: 5
Total Nodes: 10
```

Figure 13: The output from Listing 1.

Afterwards, we prompt the user to enter how many new elements they would like to insert into the tree. For simplicity, we choose 3 values which are 185, -35, -16. We expect the output to be

-35, -16, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 185.

The command line output is presented in Figure 14 and the code is in Listing 2

```
How many values to insert?
3
Enter a value to insert:
185
The new AVL-Tree: 0 1 2 3 4 5 6 7 8 9 185
Leaves: 5
Nodes: 11
Height: 4
Insertion time 143 microseconds, size 11
Operation count 5 operations
Enter a value to insert:
-35
The new AVL-Tree: -35 0 1 2 3 4 5 6 7 8 9 185
Leaves: 5
Nodes: 12
Height: 4
Insertion time 47 microseconds, size 12
Operation count 3 operations
Enter a value to insert:
-16
The new AVL-Tree: -35 -16 0 1 2 3 4 5 6 7 8 9 185
Leaves: 5
Nodes: 13
Height: 4
Insertion time 58 microseconds, size 13
Operation count 4 operations
```

Figure 14: Insertion of 185, -16, and -35.

Listing 2: The insertion part of the main program.

---

```

1  #include "avltree.h"
2  //... other #include's
3  int main(int argc, char const *argv[])
4  {
5      //... initialization
6      cout << "How many values to insert?" << endl;
7      int i, n, to_insert;
8      cin >> n;
9      i = 0;
10     while (i < n)
11     {
12         i += 1;
13         cout << "Enter a value to insert:" << endl;
14         cin >> to_insert;
15
16         operation_count = 0;
17         auto start = high_resolution_clock::now();
18         tree.root = tree.insert(tree.root, to_insert,
19                                &operation_count);
20         auto stop = high_resolution_clock::now();
21         auto duration = duration_cast<microseconds>(stop - start);
22         // count leaves
23         counter = 0;
24         nodes = countNodes(tree.root);
25         countLeaves(tree.root, &counter);
26         printf("The new AVL-Tree: ");
27         tree.inorder_tree_walk(tree.root);
28         printf("\nLeaves: %d\nNodes: %d\nHeight: %d\n",
29               counter, nodes, tree.root->height);
30
31         cout << "Insertion time " << duration.count()
32              << " microseconds, size " << nodes << endl;
33
34         cout << "Operation count " << operation_count
35              << " operations" << endl;
36     }
37     //... program continues
38 }

```

---

After insertion we prompt the user to search for an item. In case of success, we output the value, in case of failure (item not found) we output an error message, see Figure 15. The code for the search proces is presented in Listing 3

```
Enter a value to find:
-35
Success, found -35,      operation count 3
Search time 0 microseconds, size 13
```

```
Enter a value to find:
223
Search failed, 223 not found :( Operation count 5
Search time 9 microseconds, size 13
```

Figure 15: Search, two cases: success and failure.

Listing 3: The search part of the main program.

---

```
1 #include "avltree.h"
2 //... other #include's
3 int main(int argc, char const *argv[])
4 {
5     //... insertion
6     cout << "Enter a value to find:" << endl;
7     cin >> val;
8     Node *found;
9     operation_count = 0;
10    auto start = high_resolution_clock::now();
11    found = tree.search(tree.root, val, &operation_count);
12    auto stop = high_resolution_clock::now();
13    auto duration = duration_cast<microseconds>(stop - start);
14    if (found != NIL)
15    {
16        printf("Success, found %d,\toperation count %d\n",
17            found->get_val(), operation_count);
18    }
19    else
20    {
21        printf("Search failed, %d not found :(
22            \tOperation count %d\n", val, operation_count);
23    }
24    cout << "Search time " << duration.count()
25    << " microseconds" << endl;
26
27    //... program continues
28 }
```

---

Next, we propose to test deletion. The user enters a key from the keyboard and if that key is found then deletion happens, otherwise an error message appears and no deletion occurs, see Figure 16.

```

Enter a value to delete
225
Before: -35 -16 0 1 2 3 4 5 6 7 8 9 185
Leaves: 5
Nodes: 13
Height: 4
After: -35 -16 0 1 2 3 4 5 6 7 8 9 185
Leaves: 6
Nodes: 13
Height: 4
Deletion time 1 microseconds, tree size 13 nodes.
Deletion status is 0(1 - success, 0 - fail, no such element found)

Enter a value to delete
0
Before: -35 -16 0 1 2 3 4 5 6 7 8 9 185
Leaves: 5
Nodes: 13
Height: 4
After: -35 -16 1 2 3 4 5 6 7 8 9 185
Leaves: 5
Nodes: 12
Height: 4
Deletion time 24 microseconds, tree size 12 nodes.
Deletion status is 1(1 - success, 0 - fail, no such element found)

```

Figure 16: Deletion, two cases: success (*right*) and failure.

The code for the deletion portion is presented in Listing 4.

Listing 4: The deletion part of the main program.

---

```

1 #include "avltree.h"
2 //... other #include's
3 int main(int argc, char const *argv[])
4 {
5     //... search
6
7     cout << "Enter a value to delete" << endl;
8     cin >> val;
9     operation_count = 0;
10    printf("\n");
11    printf("Before: ");
12    if (atoi(argv[1]) <= 200)
13    {
14        tree.inorder_tree_walk(tree.root);
15    }
16    else
17    {
18        printf("Tree too big, no inorder output.");
19    }
20    printf("\n\tLeaves: %d\n\tNodes: %d\n\tHeight: %d\n",
21    counter, nodes, tree.root->height);
22    start = high_resolution_clock::now();
23    Node *deleted = tree._delete(tree.root, val, &operation_count);
24    stop = high_resolution_clock::now();
25    duration = duration_cast<microseconds>(stop - start);
26
27    counter = 0;
28    nodes = countNodes(tree.root);
29    countLeaves(tree.root, &counter);
30    printf("\n");
31    printf("After: ");
32    if (atoi(argv[1]) <= 200)
33    {
34        tree.inorder_tree_walk(tree.root);
35    }

```



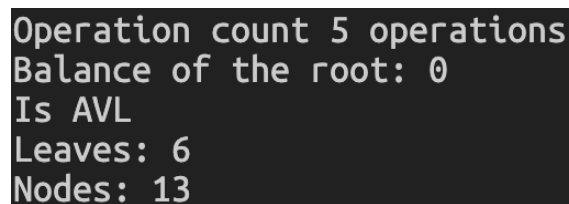
```

36     else
37     {
38         printf("Tree too big, no in-order output.");
39     }
40     printf("\n\tLeaves: %d\n\tNodes: %d\n\tHeight: %d\n",
41         counter, nodes, tree.root->height);
42     printf("\n");
43     cout << "Deletion time " << duration.count()
44     << " microseconds, tree size " << nodes << " nodes." << endl;
45     cout << "Deletion status is " << tree.deletion_success
46     << " (1 - success, 0 - fail, no such element found)" << endl;
47     printf("\n");
48     cout << "Operation count " << operation_count
49     << " operations" << endl;
50
51     //... program continues
52 }

```

---

At the end of the program we output some information about the tree, see Figure 17 and Listing 5.



```

Operation count 5 operations
Balance of the root: 0
Is AVL
Leaves: 6
Nodes: 13

```

Figure 17: Final output of the code: 3 elements were inserted, none were deleted.

---

Listing 5: The final part of the main program.

```

1  #include "avltree.h"
2  //... other #include's
3  int main(int argc, char const *argv[])
4  {
5      printf("Balance of the root: %d\n", tree.get_balance(tree.root));
6      // printf("Height: %d\n", tree.get_height(tree.root));
7
8      if (tree.is_avl(tree.root))
9      {
10         printf("Is AVL\n");
11     }
12     else
13     {
14         printf("Is Not AVL\n");
15     }
16     counter = 0;
17     nodes = countNodes(tree.root);
18     countLeaves(tree.root, &counter);

```

```
19     printf("Leaves: %d\nNodes: %d\n", counter, nodes);
20     return 0;
21 }
```

---

```

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 4
7 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 9
1 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 12
6 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148
The AVL-Tree is of height 8, root is 63
Leaves: 74
Total Nodes: 149
How many values to insert?
2
Enter a value to insert:
-10
The new AVL-Tree: -10 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83
84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120
121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148
Leaves: 74
Nodes: 150
Height: 8
Insertion time 4 microseconds, size 150
Operation count 7 operations
Enter a value to insert:
256
The new AVL-Tree: -10 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83
84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120
121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 256
Leaves: 75
Nodes: 151
Height: 8
Insertion time 7 microseconds, size 151
Operation count 9 operations
Insertion finished, begin search

```

Figure 18: Part 2 of the output on 149 initial elements.

```

Insertion finished, begin search
Enter a value to find:
256
Success, found 256, operation count 8
Search time 1 microseconds, size 151
Search finished, begin deletion
Enter a value to delete
-10
Before: -10 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 4
3 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 8
7 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 12
3 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 256
Leaves: 75
Nodes: 151
Height: 8
After: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44
45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88
89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124
125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 256
Leaves: 75
Nodes: 150
Height: 8
Deletion time 23 microseconds, tree size 150 nodes.
Deletion status is 1(1 - success, 0 - fail, no such element found)
Operation count 7 operations
Balance of the root: -1
is AVL
Leaves: 75
Nodes: 150

```

Figure 19: Part 2 of the output on the 149 initial elements.

We finalize this section with one more screenshot on a larger dataset of 149 elements., see Figure 18 and Figure 19

## A.1 Node of the tree

The following listing creates a class Node with some inherent, node-specific methods and attributes.

Listing 6: C++ code for the Node class

---

```
1  #define NIL NULL
2
3  class Node
4  {
5  private:
6  public:
7      int val;
8      int height;
9      Node *left;
10     Node *right;
11     Node *p;
12
13     int get_val();
14     int get_left();
15     int get_right();
16     int get_parent();
17     void set_left(Node *left_child);
18     void set_right(Node *right_child);
19
20     Node(int value);
21     Node();
22     ~Node();
23 };
24
25 Node::Node()
26 {
27 }
28
29 Node::Node(int value)
30 {
31     Node::val = value;
32     Node::height = 1;
33     Node::left = NIL;
34     Node::right = NIL;
35     Node::p = NIL;
36 }
37
38 Node::~~Node()
39 {
40 }
41
42 int Node::get_val()
43 {
```

```
44     return this->val;
45 }
46
47 Node *newNode(int value)
48 {
49     Node *node = new Node();
50     node->val = value;
51     node->left = NIL;
52     node->right = NIL;
53     node->p = NIL;
54     node->height = 0;
55     return node;
56 }
```

---

## A.2 AVL-Tree code

In this section, we will present the code for AVL-Tree class.

Listing 7: C++ code for the AVL-tree class.

---

```
1 #include "node.h"
2 // some other includes are omitted
3 class AVLTree
4 {
5 public:
6     Node *root;
7
8     AVLTree(Node *root); // initialize a tree with a root
9     AVLTree(vector<int> A, int start, int finish);
10    ~AVLTree();
11    int deletion_success;
12    void inorder_tree_walk(Node *node); // O(n)
13    void _transplant(Node *u, Node *v); // O(1)
14    int height(Node *root); // O(1)
15    int get_height(Node *root); // O(1)
16    int get_balance(Node *node); // O(1)
17    bool is_avl(Node *root); // O(n log n)
18    Node *insert(Node *root, int key); // O(log n) + O(log n)
19    Node *_delete(Node *root, int key); // O(log n) + O(log n)
20    Node *left_rotate(Node *node); // O(1)
21    Node *right_rotate(Node *node); // O(1)
22    Node *search(Node *x, int key); // O(log n)
23    Node *tree_minimum(Node *x); // O(log n)
24    Node *tree_maximum(Node *x); // O(log n)
25    Node *tree_successor(Node *x); // O(log n)
26 };
```

---

For the sake of brevity, we will present details of implementation of the most relevant functions. The construction of the tree is realized through repeated insertion, let us present the code for that method.

Listing 8: Code for AVL-TREE-INSERT method.

---

```

1 Node *AVLTree::insert(Node *node, int key)
2 {
3     /*
4     insert a node into an avl tree.
5     */
6     if (node == NIL)
7     {
8         return (newNode(key)); // O(1)
9     }
10    if (key < node->val)
11    {
12        node->left = this->insert(node->left, key); // T(n/2)
13        node->left->p = node;
14    }
15    else if (key > node->val)
16    {
17        node->right = this->insert(node->right, key); // T(n/2)
18        node->right->p = node;
19    }
20    else
21    {
22        return node;
23    }
24    node->height = 1 + max(height(node->left),
25                          height(node->right)); // O(1)
26    /*Begin rebalancing*/
27    int balance = this->get_balance(node); // O(1)
28    if (balance > 1 && node->left != NIL && key < node->left->val)
29    {
30        return this->right_rotate(node); // O(1)
31    }
32    if (balance > 1 && node->left != NIL && key > node->left->val)
33    {
34        node->left = this->left_rotate(node->left);
35        return this->right_rotate(node); // O(1)
36    }
37    if (balance < -1 && node->right != NIL && key > node->right->val)
38    {
39        return this->left_rotate(node); // O(1)
40    }
41    if (balance < -1 && node->right != NIL && key < node->right->val)
42    {
43        node->right = this->right_rotate(node->right);

```

```
44         return this->left_rotate(node); // O(1)
45     }
46     return node;
47 }
```

---



Next, we present the listing for deletion.

Listing 9: Code for AVL-TREE-DELETE method.

---

```
1 Node *AVLTree::_delete(Node *node, int key)
2 {
3
4     if (node == NIL)
5     {
6         this->deletion_success = 0;
7         return node;
8     }
9     //search for the required node
10    if (key < node->val)
11    {
12        node->left = this->_delete(node->left, key);
13    }
14    else if (key > node->val)
15    {
16        node->right = this->_delete(node->right, key);
17    }
18    else
19    {
20        // found it
21        // if it has one or no children
22        if ((node->left == NIL) || (node->right == NIL))
23        {
24            Node *temp = node;
25
26            if (node->left != NIL)
27            {
28                node = node->left;
29            }
30            else
31            {
32                node = node->right;
33            }
34            this->_transplant(temp, node);
35            this->deletion_success = 1;
36            free(temp);
37        }
38        else
39        {
40            // node with two children
41            Node *temp = this->tree_minimum(node->right);
42            // find successor
43            node->val = temp->val;
44            node->right = this->_delete(node->right, temp->val);
45        }
46    }
47 }
```

```
46     if (node == NIL)
47     {
48         this->deletion_success = 1;
49         return node;
50     }
51     node->height = this->get_height(node);
52     int balance = this->get_balance(node);
53     /* For the sake of brevity, we omit full rebalancing in this listing*/
54     return node;
55 }
```

---

It is necessary to list the transplantation and rotation methods used in deletion. The listings for each are presented below.

Listing 10: Code for TRANSPLANT method.

---

```
1 void AVLTree::_transplant(Node *u, Node *v)
2 {
3
4     if (u->p == NIL)
5     {
6         this->root = v;
7     }
8     else if (u == u->p->left)
9     {
10        u->p->left = v;
11    }
12    else
13    {
14        u->p->right = v;
15    }
16    if (v != NIL)
17    {
18        v->p = u->p;
19    }
20 }
```

---

Listing 11: Code for LEFT-ROTATE method.

---

```
1 Node *AVLTree::left_rotate(Node *x)
2 {
3     Node *y = x->right;
4     x->right = y->left;
5     if (y->left != NIL)
6     {
7         y->left->p = x;
8     }
9     y->p = x->p;
10    if (x->p == NIL)
11    {
12        this->root = y;
13    }
14    else if (x == x->p->left)
15    {
16        x->p->left = y;
17    }
18    else
19    {
20        x->p->right = y;
21    }
22    y->left = x;
```

```

23     x->p = y;
24     x->height = 1 + max(height(x->left), height(x->right));
25     y->height = 1 + max(height(y->left), height(y->right));
26     return y;
27 }

```

---

Listing 12: Code for RIGHT-ROTATE method.

```

1 Node *AVLTree::right_rotate(Node *y)
2 {
3     Node *x = y->left;
4     y->left = x->right;
5     if (x->right != NIL)
6     {
7         x->right->p = y;
8     }
9     x->p = y->p;
10    if (y->p == NIL)
11    {
12        this->root = x;
13    }
14    else if (y == y->p->left)
15    {
16        y->p->left = x;
17    }
18    else
19    {
20        y->p->right = x;
21    }
22    x->right = y;
23    y->p = x;
24    x->height = 1 + max(height(x->left), height(x->right));
25    y->height = 1 + max(height(y->left), height(y->right));
26    return x;
27 }

```

---

Let us show the listing for the SEARCH method implemented iteratively.

Listing 13: Code for search method that is identical to that of a regular BST.

---

```
1 Node *AVLTree::search(Node *x, int key)
2 {
3     /* Search for a key */
4     Node *temp = x;
5     while (temp != NIL && key != temp->get_val())
6     {
7         if (key < temp->get_val())
8         {
9             temp = temp->left;
10        }
11        else
12        {
13            temp = temp->right;
14        }
15    }
16    return temp;
17 }
```

---

Finally, in listings to follow, we present some additional functions that were used to check if the tree is indeed an AVL-tree (GET-BALANCE, IS-AVL)

Listing 14: Code for additional methods.

---

```
1  int AVLTree::height(Node *node)
2  {
3      if (node == NIL)
4      {
5          return 0;
6      }
7      return node->height;
8  }
9
10 int AVLTree::get_height(Node *root)
11 {
12     if (root == NIL)
13     {
14         return 0;
15     }
16     int leftHeight = height(root->left);
17
18     int rightHeight = height(root->right);
19
20     int max_height = max(leftHeight, rightHeight) + 1;
21     root->height = max_height;
22     return max_height;
23 }
24
25 int AVLTree::get_balance(Node *node)
26 {
27     if (node == NIL)
28     {
29         return 0;
30     }
31     int left_height, right_height;
32     left_height = height(node->left);
33     right_height = height(node->right);
34     return left_height - right_height;
35 }
```

---

The full version of this code can be found on GitHub<sup>2</sup>.

---

<sup>2</sup><https://github.com/iliaailmer/avl-tree>