Lab 3. Binary Trees

Release: **7th February 2024**
Supervision: **20th February 2024**
Due: **26th February 2024**

Vladimir Tarasov

1 Introduction

The purpose of this task is to create a function to perform deletion of an element in the *right sub-tree* of an AVL tree. Implementation of this function will utilize the similarity of rotations after deletion and insertion. This task aims at achieving conceptual understanding of how insertion and deletion operations work with AVL trees.

Note: this task is not heavy on the coding part but it will require effort to achieve conceptual understanding of how insertion and deletion operations work with AVL trees.The best way to do this is to study both code and illustrations (in the slides or textbook) together and, at the same way, make simple sketches on paper.

2 Deletion in AVL Trees

A quick recap of deletion in AVL trees (see the lecture slides or ch. 10 in the textbook):

- Deletion of a node in an AVL tree is similar to that of binary search trees.
- Deletion may disturb the AVLness of the tree, so to re-balance the AVL tree we need to perform rotations.
- The balance factor of a node
  `Balance factor = Height (left sub-tree) - Height (right sub-tree)`
- If node is in the *right sub-tree*, R rotation is performed
- There are R0, R−1 and R1 rotations

To implement the rotations, we will utilize the similarity of rotations after deletion and insertion:

- R0 rotation after deletion is similar to LL rotation after insertion (see Fig. 1).
- R1 rotation after deletion is similar to LL rotation after insertion (see Fig. 1).
- $R-1$ rotation after deletion is similar to LR rotation after insertion (see Fig. 2).

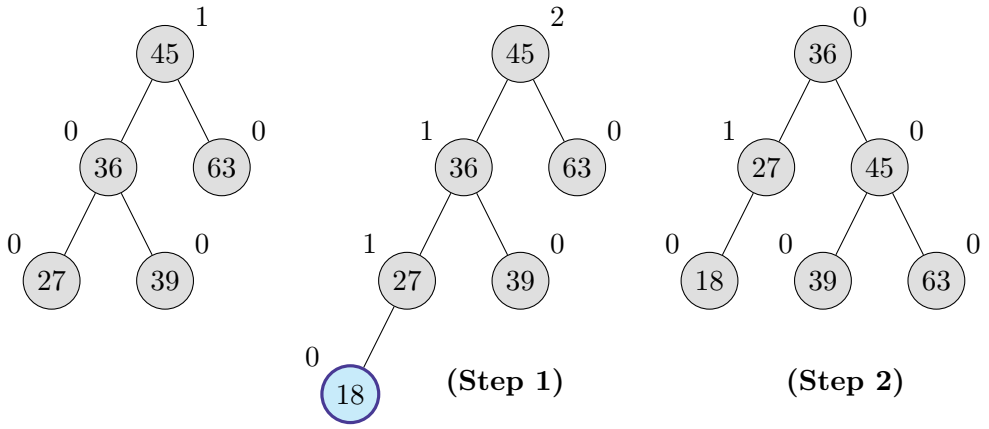

Figure 1: LL rotation after insertion of **18** (fig. 10.41)

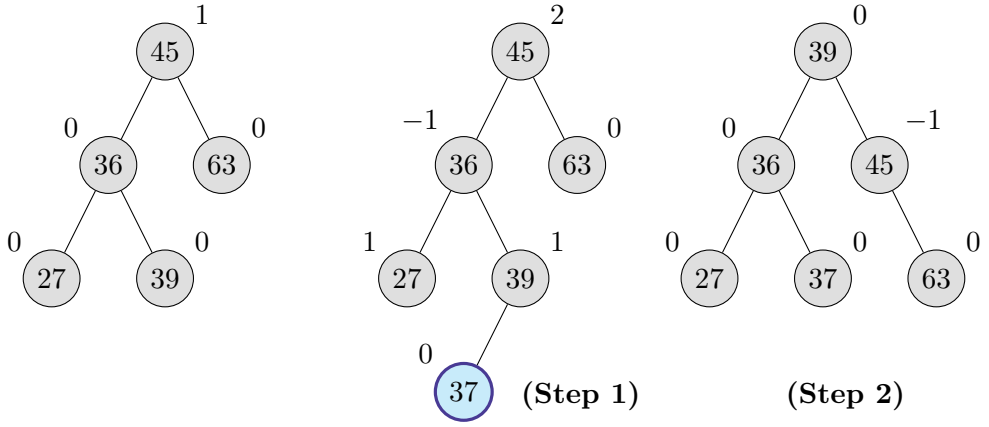

Figure 2: LR rotation after insertion of **37** (fig. 10.45)

3 Implementation of Deletion Operation on AVL Trees

To carry out this task, the following steps are recommended:

1. Read and study the supplied code in `avl_tree_insert.c`. Detailed comments are added there with references to the lecture and textbook.
2. Observe how the shape of a resulting AVL tree depends on the order of entering node values. Run the code several times with different inputs and the same inputs but in different order.
3. Examine how the AVL tree is re-balanced after insertion. Use examples in Fig. 1 and Fig. 2.
4. Implement deletion of a node in the *right sub-tree* of an AVL tree. In this task we will work only with the right sub-tree.
5. The detailed specification of the functions can be found in the header file `lab_binary_trees.h`, which must be included in your `.c` file. The details of the implementation are given below.

Details of the Implementation of Deletion Function

- This lab will reuse the source code that is needed for insertion from `avl_tree_insert.c` from the textbook (with additional comments explaining how insertion works):
  - An enumeration emulating a boolean type.
  - A structure implementing a node of an AVL tree.
  - A function to insert a new element into an AVL tree.
  - A function to display an AVL tree.
- You will need to add to the code in the `avl_tree_insert.c` file these functions:
  - A function to *find the largest element* in a AVL tree.
  - A function to *delete an element* from an AVL tree
- The general structure of the deletion function should be similar to the one from `ex_bst_9.c` and the function to find the largest element can be reused from there.
- To re-balance the AVL tree after the deletion, perform a rotation: R0, R−1 or R1 rotation.
- Employ the similarities between insertion rotations and deletion rotations. Reuse the code from `avl_tree_insert.c` for LL and LR rotations to im-

plement R1, R0, and R−1 rotations. Refer to the lecture slides or ch. 10 in the textbook if necessary.

- Take into account that **R rotations** are performed after deletion in the **right sub-tree**, however, you need to reuse *LL and LR rotations* that are performed after insertion in the *left sub-tree*. So be careful when you copy and modify the code.
- One more hint: The value of the `ht_inc` variable that controls re-balancing (see the comments for explanation) needs NOT to be changed in the branch `case -1: /* Right heavy */` after deletion because −1 will be on path to critical node and re-balancing will be needed at a higher level.
- Modify `main()` function from `avl_tree_insert.c`, to test the deletion operation with two test cases from the textbook. Fig. 3 for $R-1$ rotation (10.53 from the textbook) and Fig. 4 for R0 rotation (modified 10.54 from the textbook).
  - Create two `int` arrays of fixed size with the required nodes. Make sure that the node values are in *the right order* to create the trees from the test cases.
  - Do insertion of the nodes in a loop.
  - Then display a menu to delete a node and display the tree after the deletion.
- Test and debug the code using the two test cases until your solution provides an output similar to the one given below

4 Test Cases for Deletion in AVL Trees

There are two test cases that you need to use to check your code. The input AVL trees are shown in Fig. 3 and Fig. 4. The output is detailed in Sec. 5.1.

5 Deliverables

The deliverable is the source code that includes the tree node structure and functions specified in Sect. 3 as well as the `main()` function that is supposed to demonstrate the functionality of the deletion function.

The header file `lab_binary_trees.h` must be included in your implementation. Your implementation will be compiled with *the given header file* during marking. It is up to

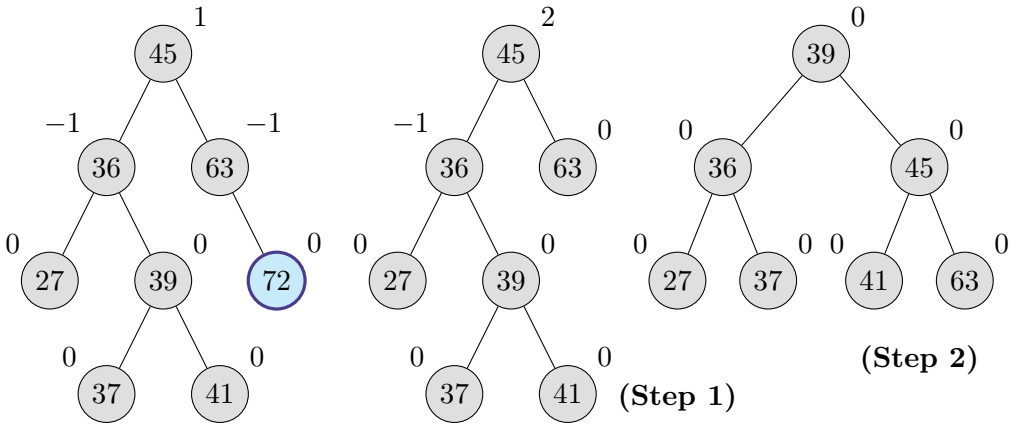

Figure 3: Example of R−1 rotation after deletion of **72** (fig. 10.53)

you to decide how you code the `main()` function but the output should be similar to the one given in Sec. 5.1.

The source code has to be tested using the test cases given in Sec. 4 as well as to adhere to the recommendations on *Standard C and portability*, which you can find in Canvas. You can compile your code using more strict flags `-Wall` together with `-pedantic` before submitting the code, e.g.

```
$ gcc -Wall -pedantic program.c -o program
```

Moreover, the source code has to include reasonable amount of comments, that is the code is supposed to be well-readable. Do not forget to add **your name in a comment** at the beginning of the `.c` file.

5.1 Example Output of the Test Cases

An example of execution of the first test case:

Output for the first test case

```
Running first test
The fisrt AVL tree is:

   72
  63
 45
    41
   39
    37
```

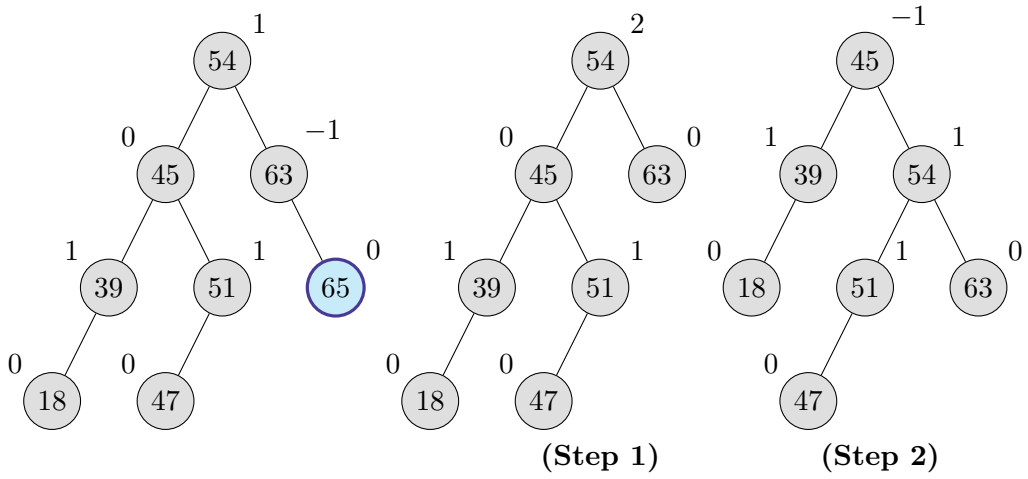

Figure 4: Example of R0 rotation after deletion of **65** (modified fig. 10.54)

```
  36
   27

1.Delete
2.Display
3.Second test
Enter your option : 1
Enter the value to be deleted : 72
R-1 rotation
1.Delete
2.Display
3.Second test
Enter your option : 2
Tree is :

   63
  45
   41
 39
   37
  36
   27
```

An example of execution of the second test case:

Output for the second test case

```
Running second test
The second AVL tree:

   65
  63
 54
   51
    47
  45
   39
    18

1.Delete
2.Display
3.Quit
Enter your option : 1
Enter the value to be deleted : 65
RO Rotation
1.Delete
2.Display
3.Quit
Enter your option : 2
Tree is :

   63
  54
   51
    47
 45
  39
   18
```