CPTS 223 Advanced Data Structure C/C++ Fall 2024

PA2: AVL Trees

1 Learning Objectives

At the conclusion of this programming assignment, participants should be able to:

- Develop a C++ program that applies an AVL tree data structure;
- Design and implement an AVL tree data structure with common operations;
- Understand the balancing method in AVL trees and how it controls the average time complexity for searching in AVL trees.

2 Prerequisite

Before starting this programming assignment, participants should be able to:

- Apply and implement class templates;
- Design, implement, and test medium programs in C++;
- Cross-platform compilation with CMake;
- Maintain coding projects with Git on github.
- Finish reading §4.1 to §4.4 of Textbook.

3 Overview and Requirements

3.1 Overview

For this assignment, we will design and implement an AVL tree data structure using class template for two verification experiments to (i) test the correctness of the operations in our implementation and (ii) investigate the balance condition of AVL trees.

In the *first experiment*, we will generate a sequence of odd integers and insert them into three AVL trees in three different orders, i.e., the ascending order, descending order and random order. The goal of this experiment is to comprehensively test whether the BST order condition and the AVL balance condition hold at the same time. As a result, We will test these AVL trees with three test functions as follows.

- `testBST` tests whether the AVL tree satisfies the BST order requirement for each node (e.g., see §4.3 on Page 132 in Textbook for the BST order requirement).
- `testBalanced` tests whether the AVL tree satisfies the AVL balance condition for each node (e.g., see §4.4 on Page 144 and 145 in Textbook for the AVL balance condition).
- `testContains` tests whether the AVL tree contains all the given data (all odd integers within the given range).

The *second experiment* follows the setting used by the example of Figure 4.29 and Figure 4.30 in §4.3.6 of Textbook, i.e., random insert/remove pair operations in BSTs (rather than AVL trees). Here is a recap for the two stages in this example: in the first stage, we insert 500 randomly generated integers into BST (so the size of this BST is 500); in the second stage, we perform 500^2 $(250,000)$ times of random insert/remove pair operations. The random insert/remove pair operation particularly means that we insert a random integer and immediately remove a random element from the BST, so after 500^2 times of random insert/remove pair operations, the BST size should be still 500.

Although the tree size is identical before and after 500^2 random insert/remove pair operations, according to the authors, the average depth over all nodes [1] is 9.98 in the first stage, while the average depth over all nodes is 12.51 after the second stage, i.e., around 25% relative increase from 9.98. This growth of the average depth is due to the design of `delete` function in BSTs that "favors making the left subtrees deeper than the right, because we are always replacing a deleted node with a node from the right subtree", according to the authors. Because the average depth can be regarded as an estimator to the height h in the average-case analysis of BSTs, we expect self-balanced BSTs to maintain a stable and small height that is invariant to the repeated insert/remove pair operations, rather than the standard BSTs.

As a result, the goal of the second experiment is to investigate the following question:

Q 1. *Is the average depth of AVL trees invariant to random insert/remove pair operations?*

We will reproduce the setting used by Figure 4.29 and Figure 4.30, based on which we test and monitor the AVL tree's behavior to understand more about the self-balancing properties of AVL trees. Finally we will have a conclusion for this critical question.

Specifically, we will use three key test functions.

- `testBalanced` is the same test function as in the first test case. Passing this test means that our AVL tree works properly for this experiment setup (not just a standard BST).
- `testSize` tests whether the tree size matches the number of input data before and after the random insert/remove pair operations. Passing this test means that the random insert/remove pair operation does not increase or decrease the number of nodes contained by the AVL tree (so we will only use it in stage 2). Based on this condition, we can verify whether the average depth of AVL trees is invariant to the random insert/remove pair operations.
- `testHeight` tests whether the member variable "height" of each AVL Node is correctly maintained. This verification can be done by comparing the "height" member variable with the height computed by DFS (depth-first search, see Figure 4.61 in Textbook) on each AVL Node.

The above three test functions provide a solid foundation to compute correct average depth over all nodes, based on which we can directly compare our average depth with that reported in Figure 4.29 and Figure 4.30. To this end, in the first stage (insert random integers), we compute and keep the average depth of our AVL tree (denoted by $\bar{D}_1$) is same as or close to the average depth in Figure 4.29, i.e., $\bar{D}_1 \approx 9.98$. Then, after the second stage (500^2 random insert/remove pair operations), we compute the average depth of our updated AVL tree (denoted by $\bar{D}_2$) and compare $\bar{D}_2$ with (i) the average depth in our first stage $\bar{D}_1$ as well as (ii) the average depth in Figure 4.30 (i.e., 12.51). Our expected result should include: (i) $\bar{D}_1$ is very close to $\bar{D}_2$ and (ii) $\bar{D}_2 < 12.51$ with a large margin.

As a result, the following member function will be included in AVL tree class for the verification.

- `averageDepth` returns the average depth (double data type) over all nodes in a BST tree.

3.2 Requirements

The starter code provides five files:

- "AVLTree.h" for the declaration and implementation of the class of AVL tree (partially finished and to be finished);

[1] It is worth recalling that the sum of depth over all nodes is equal to the internal path length, as introduced in the same Chapter of Textbook.

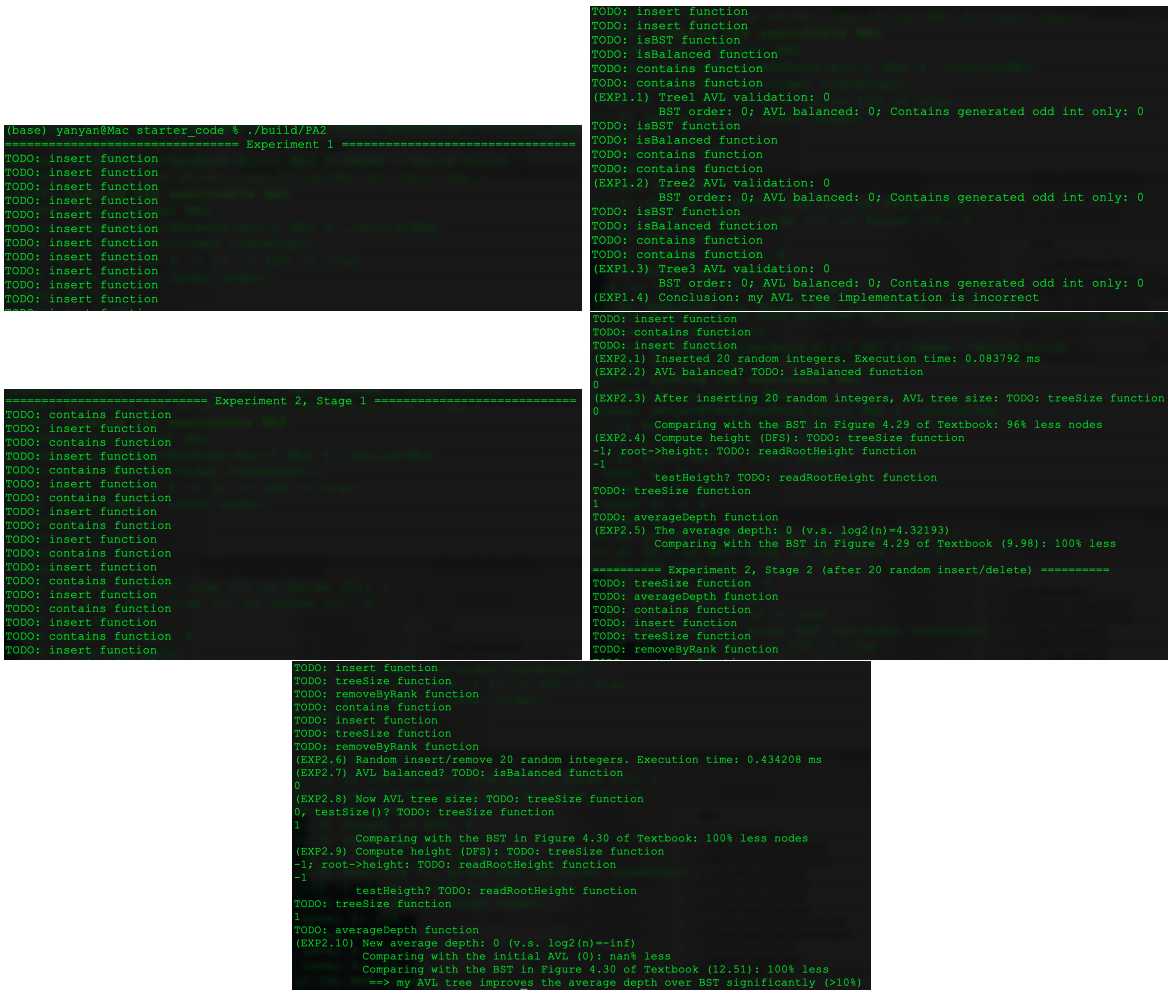

Figure 1: Example of running the starter code.

- "experimentFunctions.h" for the declaration of the functions used by two experiments;
- "experimentFunctions.cpp" for the implementation of the functions used by two experiments;
- "main.cpp" for the main function;
- "CMakeLists.txt" for CMake configurations.

Our main task is to finish the implementation for the class of AVL tree in "AVLTree.h". The AVLNode struct and four member functions have been implemented already:

- Constructor `AVLTree`;
- Destructor ~`AVLTree` (following BST, as Figure 4.27 in Textbook);
- `findMin` (following BST, as Figure 4.20 in Textbook);
- `findMax` (following BST, as Figure 4.21 in Textbook).

After downloading the starter code and cmaking these code, we will have some outputs shown as in Figure 1.

The member functions in "AVLTree.h" to be completed are listed as follows.

1. (4 pts) `contains`: follow BST implementation. See Figure 4.17 and Figure 4.18 in Textbook;
2. (4 pts) `insert`: follow BST implementation + balance(). See Figure 4.42 in Textbook;

3. (4 pts) `remove`: follow BST implementation + balance(). See Figure 4.47 in Textbook;
4. (8 pts) `balance`: handling the update of the "height" field for each node involved in insert and remove. See Figure 4.42 in Textbook;
5. (8 pts) `rotateWithLeftChild`: AVL rotation for case 1. See Figure 4.43 (visualization) and Figure 4.44 (code) in Textbook;
6. (2 pts) `rotateWithRightChild`: AVL rotation for case 4, the mirrored situation of case 1;
7. (8 pts) `doubleWithLeftChild`: AVL rotation for case 2. See Figure 4.45 (visualization) and Figure 4.46 (code) in Textbook;
8. (2 pts) `doubleWithRightChild`: AVL rotation for case 3, the mirrored situation of case 2;
9. (10 pts) `isBST`: returns true if each node satisfies BST order requirement. It can be implemented in a recursive fashion or directly based on findMin/findMax functions (any other implementation strategies are also encouraged);
10. (10 pts) `isBalanced`: returns true if each node satisfies AVL balance condition. It can be implemented in a recursive way;
11. (4 pts) `treeSize`: returns an integer, the number of nodes in AVL. It can be implemented in a recursive manner;
12. (4 pts) `computeHeight`: computes the height of the AVL using DFS. See Figure 4.61 in Textbook. Note that this function should not be built based on the "height" field of AVL node, because it is used for testing whether the maintenance of the "height" value of AVL node is correct (adjustment after insert and remove);
13. (4 pts) `readRootHeight`: directly returns the "height" field in AVL node. We will compare readRootHeight with computeHeight in testHeight;
14. (18 pts) `averageDepth`: computes the average depth over all nodes;
15. (10 pts) `removeByRank`: removes a node from AVL according to its rank, which is used for random insert/remove pair operations, i.e., randomly remove a node with an input rank.

If implementation is completed correctly, the results from the two experiments will be something as shown in Figure 2. Given the experiment outcomes, will you make any empirical conclusion for our Question 1 in Section 3.1? Is the average depth of AVL trees is invariant to the random insert/remove pair operations?

4 How to Submit: Github (and Share with TA)

1. In your Git repository for this class's coding assignments, **create a new branch called "PA2"** (Refer to this YouTube video about how to create a branch from Github web interface or the terminal). In the current working directory, also create a new directory called "PA2" and place all PA2 files in the "PA2" directory. All files for PA2 should be added, committed and pushed to the remote origin which is your private GitHub repository created when during PA1 (NO NEED TO CREATE A NEW REPO).
2. You should submit at least the following files:
   - the header file ("AVLTree.h"), where you will implement the BST class.
   - the main C++ source file ("main.cpp"), where you will have all tests passed;
   - all experiment functions in "experimentFunctions.h" and "experimentFunctions.cpp";
   - a "CMakeLists.txt" file containing your CMake building commands on Linux/WSL/MacOS which can compile your code.

```
(base) yanyan@Mac PA2 % ./build/PA2
================================ Experiment 1 ================================
(EXP1.1) Tree1 AVL validation: 1
         BST order: 1; AVL balanced: 1; Contains generated odd int only: 1
(EXP1.2) Tree2 AVL validation: 1
         BST order: 1; AVL balanced: 1; Contains generated odd int only: 1
(EXP1.3) Tree3 AVL validation: 1
         BST order: 1; AVL balanced: 1; Contains generated odd int only: 1
(EXP1.4) Conclusion: my AVL tree implementation is correct

============================ Experiment 2, Stage 1 ============================
(EXP2.1) Inserted 3000 random integers. Execution time: 2.04529 ms
(EXP2.2) AVL balanced? 1
(EXP2.3) After inserting 3000 random integers, AVL tree size: 3000
         Comparing with the BST in Figure 4.29 of Textbook: 500% more nodes
(EXP2.4) Compute height (DFS): 13; root->height: 13
         testHeigth? 1
(EXP2.5) The average depth: 9.84667 (v.s. log2(n)=11.5507)
         Comparing with the BST in Figure 4.29 of Textbook (9.98): 1.33601% less

========== Experiment 2, Stage 2 (after 250000 random insert/delete) ==========
(EXP2.6) Random insert/remove 250000 random integers. Execution time: 35104.1 ms
(EXP2.7) AVL balanced? 1
(EXP2.8) Now AVL tree size: 3000, testSize()? 1
         Comparing with the BST in Figure 4.30 of Textbook: 500% more nodes
(EXP2.9) Compute height (DFS): 13; root->height: 13
         testHeigth? 1
(EXP2.10) New average depth: 9.81233 (v.s. log2(n)=11.5507)
          Comparing with the initial AVL (9.84667): 0.3499% less
           ==> average depth of my AVL tree does not change significantly (<=1%)
          Comparing with the BST in Figure 4.30 of Textbook (12.51): 21.5641% less
           ==> my AVL tree improves the average depth over BST significantly (>10%)
```

Figure 2: Example of running the final code for two experiments.

3. You can refer the example GitHub template project for how to use CMake at https://github.com/DataOceanLab/CPTS-223-Examples.

4. Please invite the GitHub accounts of TAs (see Syllabus page and check TA's names as well as their Github usernames before submitting) as the collaborators of your repository. You should submit a URL link to the branch of your private GitHub repository on Canvas. Otherwise, we will not be able to see your repository and grade your submission.

5. Please push all commits of this branch before the due date for submitting your Github link to Canvas portal. Otherwise it might be considered as late submission.

Here is a checklist for submitting your code:

- Invite all TAs and instructor as collaborators of your created repository "CPTS223_assignments". Their github username can be found in Syllabus on Canvas.
- Commit and push your local code to your repository.
- Make sure that your repository reflects and contains all your latest files.
- Copy the link to your repository in the Canvas submission portal.

5 Grading Guidelines

This assignment is worth 100 points. We will grade according to the following criteria: See Section 3.2 for individual points.