Project 1: Gator AVL Project

Submit Assignment

Due Wednesday by 11:59pm **Points** 100 **Submitting** a file upload **File Types** cpp, pdf, doc, and docx
Available Feb 3 at 12am - Feb 28 at 11:59pm 26 days

Gator AVL Project

Problem Statement

Binary Search Trees (BST) can often be an efficient and useful way to store and retrieve sorted data. However, the efficiency of these data trees relies heavily on how balanced a BST is. For example, searching through the BST on the left is much more efficient than searching through the BST on the right, despite both figures showing valid BST with the exact same elements.

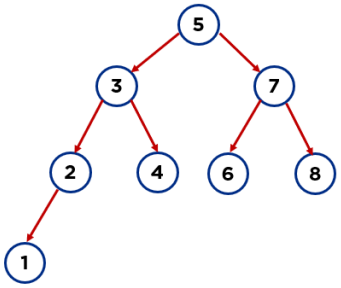

Fig.1 : A nearly balanced BST

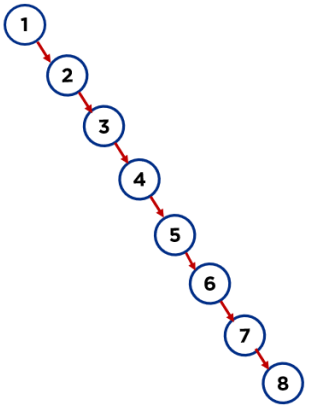

Fig.2 : A skewed BST

To avoid inefficient binary search trees, we use balanced Binary Search Trees. A balanced BST has a balance factor of less than ±*threshold*, where the balance factor is the difference in heights of the left and right subtrees at any given tree node. One such balanced tree is an AVL tree that maintains a *threshold* of **1**. As soon as a node in an AVL tree hits a balance factor of +2/-2, “tree rotations” will be performed to maintain balance in the tree.

In this project, you will be designing a custom AVL tree to organize UF student accounts based on GatorIDs. You will write methods for the insertion, deletion, search, and traversal for an AVL tree data structure. These methods would be called based on certain commands that are invoked in the main method. You are responsible for

- Designing the interface/modules/functions of the standard AVL Tree and the operations required to execute the respective commands.
- Parsing the input and ensuring the constraints are met.
- Building the main function to parse the inputs and calling the respective functions to match the output.
- Testing your code within the constraints.

Functionality

Your program must support the following commands:

Command	Function
insert *NAME ID*	• Add a Student object into the tree with the specified name, *NAME* and GatorID number, *ID*. • Balance the tree automatically if necessary. • The *ID* must be unique. • The *ID* and *NAME* must satisfy the constraints stated below. • Also, prints “successful” if insertion is successful and prints “unsuccessful” otherwise. • *NAME* identifier will be separated by double inverted commas for parsing, e.g. "Josh Smith".
remove *ID*	• Find and remove the account with the specified *ID* from the tree. • [**Optional**: Balance the tree automatically if necessary. We will test your code only on cases where the tree will be balanced before and after the deletion. So you can implement a BST deletion and still get full credit] • If deletion is successful, print “successful”. • If the *ID* does not exist within the tree, print “unsuccessful”. • You must prioritize replacing a removed node with its inorder *successor* for the case where the deleted node has two children.
search *ID*	• Search for the student with the specified *ID* from the tree. • If the *ID* was found, print out their *NAME*. • If the *ID* does not exist within the tree, print “unsuccessful”.
search *NAME*	• Search for the student with the specified name, *NAME* in the tree. • If the student name was found, print the associated *ID*. • If the tree has more than one object with the same *NAME*, print each *ID* on a new line with no other spaces and in the same relative order as a pre-order traversal. • If the name does not exist within the tree, print “unsuccessful”. • *NAME* identifier will be separated by double inverted commas for parsing, e.g. "Josh Smith".
printInorder	• Print out a comma separated inorder traversal of the names in the tree.
printPreorder	• Print out a comma separated preorder traversal of the names in the tree.
printPostorder	• Print out a comma separated postorder traversal of the names in the tree.
printLevelCount	• Prints the number of levels that exist in the tree. • Prints 0 if the head of the tree is null. For example, the tree in ***Fig. 1*** has 4 levels.
removeInorder *N*	• Remove the N^{th} GatorID from the inorder traversal of the tree (*N* = 0 for the first item, etc). • If removal is successful, print “successful”. • [**Optional**: Balance the tree automatically if necessary. We will test your code only on cases where the tree will be balanced before and after the deletion. So you can implement a BST deletion and still get full credit] • If the N^{th} GatorID does not exist within the tree, print “unsuccessful”.

AVL Tree Data Structure

In order to receive full credit for this project, you must attempt to create an AVL Tree data structure/object class that is used in your main program. Additionally, this AVL tree should:

- Also meet the requirements for a Binary Search Tree (BST)
- Be sorted by the numerical GatorID, not the lexical Name

- Be sorted from least to greatest (nodes of lesser value are in the left subtree, nodes of greater value are in the right subtree)
- Make appropriate use of public and private methods

Constraints

- Tests
  - 1 <= *No. of Commands* <= 200,000
  - 1 <= *Unique UFIDs* <= *100,000*
- Data
  - UFIDs are strictly 8 digits long.
  - Name must include only alphabets from [a-z, A-Z, and spaces]
  - Commands: We will stick to the input format and will not test misspelled commands.
- Design/Implementation
  - You must design your own AVL tree from scratch.
  - You must use the four standard AVL rotations to keep all results standardized for our test cases.
  - You must use C++11 and ensure that your code runs on Stepik. **You will get a 0 on the coding points if your code does not execute on Stepik.**

Sample Input/Output

Input

```
8
insert “Brandon” 45679999
insert “Brian” 35459999
insert “Briana” 87879999
insert “Bella” 95469999
printInorder
remove 45679999
removeInorder 2
printInorder
```

* **Note**: Line 1 denotes the number of lines or the total number of commands that follow.

Output

```
successful
successful
successful
successful
Brian, Brandon, Briana, Bella
successful
successful
Brian, Briana
```

Testing

Test your code on Stepik before submitting your implementation. You have five available test cases and you can submit any number of times. The **sixth** test case is fake to prevent you from accessing your peers' code. **Link to Stepik**:

https://stepik.org/lesson/486088/step/1?unit=477268 (https://stepik.org/lesson/486088/step/1?unit=477268)

In addition to the 5 public test cases, after the due date your submission will be tested with 10 additional test cases. In order to maximize your grade, you should **create your own test cases** and run them against your code **on Stepik**. In particular, you should test your code with test cases that include over **100 insertions** into your tree.

FAQs

The course staff will maintain an active FAQ Google document to answer your questions. **Post your questions in Slack, but we will answer in this document and send you the link.** (https://docs.google.com/document/d/1X1giXYQStDsqXYawDLNuEJaucUcHsmr9kgSlVnwC_Z4/edit#)

Grading

- **Implementation [75 points]**
  - Your code will be tested on 15 test cases each worth 5 points:
    - 5 publicly available test cases
    - 10 test cases that will be added by the course staff during grading
    - On Stepik, test case 6 is fake. So if your code passes test cases 1-5. That means you will get 25/75 points automatically.
- **Documentation [15 Points]**
  - Submit a document addressing these prompts:
    - What is the computational complexity of each method in your implementation? Reflect for each scenario: Best, Worst and Average. [10 points]
    - What did you learn from this assignment and what would you do differently if you had to start over? [5 points]
- **Code Style and Design [10 Points]**
  - 2.5 points for design decisions and code modularity
  - 2.5 points for appropriate comments
  - 2.5 points for whitespaces
  - 2.5 points for consistent naming conventions
- **Bonus [5 points] - Capped to 100 points**
  - You can receive 5 bonus points if you submit a separate file containing 5 test cases (1 point/test) using the Catch Framework. These tests should be different than the public test cases. **Your score is however capped to 100 points for this project.** This means if you pass 14 tests and submit the bonus test files, you can get a 100 provided you score full points on the documentation. Also, if you pass 15 tests and score 23 on documentation and design + 5 points on bonus, you will still score 100 points.
  - TA's Hamish and Ori have created a GitHub repository with starter files to help enable students to get started with Catch2. The repository link: **avl-project.** (https://github.com/orileibovici/avl-project)

Submission

- One or more .cpp or .h files that have your implementation. It is ***recommended*** to upload just ***one*** .cpp file that has all your implementation and is tested on Stepik. Remember this code should be the same that you tested on Stepik.
- One pdf file that has your documentation.

Project Authors: Andres Holguin, Joshua Zimmerman, and Amanpreet Kapoor
Version: 1.0
Last Modified: 02/03/2021

Last Edited by: Amanpreet Kapoor
Change Log: *removeInorder* function must print successful after successful removal and no need to check for balance in case of successful removal.

AVL Tree

Criteria	Ratings		Pts
Implementation	**75 to >0.0 pts** **Test Cases** 5 point per correct test case	**0 pts** **No test cases passed**	75 pts
Reflection What would you do differently?	**5 pts** **Full Marks** Reflection on what you would do differently	**0 pts** **No Marks**	5 pts
Computational Complexity	**10 to >0.0 pts** **Describing Complexity of each method** 1. Big O complexity of each method 2. Best case, Worst case and Average case described 3. The variables used in Big O are correct and same as in implementation	**0 pts** **No Marks**	10 pts
Comments	**2.5 to >0.0 pts** **Full Marks** Code has appropriate comments	**0 pts** **No Marks**	2.5 pts
Whitespace	**2.5 to >0.0 pts** **Full Marks** Code has appropriate whitespace	**0 pts** **No Marks**	2.5 pts
Naming convention	**2.5 to >0.0 pts** **Full Marks** Code follows a naming convention that is coherent and consistent	**0 pts** **No Marks**	2.5 pts
Modularity Code is modular with thought out design	**2.5 to >0.0 pts** **Full Marks** Code has appropriate functions	**0 pts** **No Marks**	2.5 pts
			Total Points: 100