Project 1: 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.

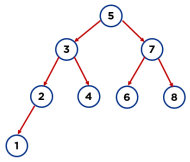

Fig.1 : A nearly balanced BST

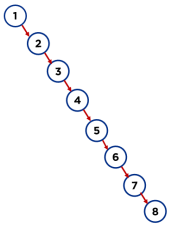

Fig.2 : A skewed BST

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 of **1**. As soon as a node in an AVL tree has a balance factor of +2/-2, “tree rotations” are 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 build methods for insertion, deletion, search, and traversals 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 data and command validation,
- 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 surrounded by double quotes 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 numerical GatorID, not 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

Testing Constraints

- 1 <= *No. of Commands <= 1000*
- 1 <= *Unique UFIDs <= 100,000*
- 1 <= *Length of a command <= 1000*

- A command will always run for a single line and will never contain new line characters ('\n'), except at the end.

Data Validation and Design Requirements

- Data (You need to validate the following and print "unsuccessful" if data or commands are invalid and move to the next command)
  - UFIDs are strictly 8 digits long.
  - Names must include only alphabets from [a-z, A-Z, and spaces]
  - Any invalid or misspelled commands must be ignored with an "unsuccessful" message followed by the execution of the next command.
- 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++14 and ensure that your code runs public test cases on Gradescope.
  - Additionally a starter template is provided which you can use for locally testing your code which can help you in saving some delay time if you were to repeatedly use a cloud based grader: Project1.zipDownload Project1.zip

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 Starter template and/or Gradescope. You have five available test cases and you can submit any number of times.

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 need to **create your own test cases** and run them against your code **in the starter template**. In particular, you should test your code with test cases that include over **100 insertions** into your tree and small tests that cover every case of the four rotations.

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.Links to an external site.

Grading

- **Implementation [75 points]**
  - Your code will be tested on 15 test cases each worth 5 points:
    - 5 publicly available test cases that are a part of Starter template and/or Gradescope
    - 10 test cases that will be added by the course staff during grading
- **Documentation [15 Points]**
  - Submit a document addressing these prompts:
    - What is the time complexity of each method in your implementation? Reflect on the worst case time complexity represented in Big O notation. [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]**
  - 6 points for design decisions and code modularity
    - We inspect the following in your code:
      - High cohesionLinks to an external site.
      - Appropriate modularity
      - Relevant access modifiers
      - Proper memory management

        - Efficient mechanisms for passing parameters in your function signatures
        - The client (your int main() method) should not have objects that interact with memory directly - or a well defined API for your AVL Tree
    - 1 point for appropriate comments
    - 1 point for consistent whitespace mechanism
    - 2 points for consistent naming conventions
- **Bonus [5 points] - Capped to 100 points**
    - You can score 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 that if you pass 14 tests and submit bonus test files, you will 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 [your score is 103 but is capped to 100 in this case].

Submission

- You are required to upload on Gradescope the following:
    - at least ***one*** .cpp file that has your implementation of the entire project including the main. If you use header files or more than one .cpp file, upload all of them together on Gradescope. Feel free to choose the names for your header and .cpp files. **Make sure at least one .cpp file contains the main() method.**
    - one pdf file that has your documentation and your name on the front page of the pdf. Name this pdf as ***Report.pdf***.
- Optionally, you can submit the catch tests in a pdf file called: ***Test.pdf*** if you decide to submit the unit tests.
- **Failure to follow these instructions will lead to 10% non-negotiable penalty**

Rubric

Criteria	Ratings	Pts
This criterion is linked to a Learning OutcomeImplementation	**75 to >0.0 ptsTest Cases5 point per correct test case0 ptsNo test cases passed**	75 pts
This criterion is linked to a Learning OutcomeReflectionWhat would you do differently?	**5 ptsFull MarksReflection on what you would do**	5 pts

	differently0 ptsNo Marks	
This criterion is linked to a Learning OutcomeTime Complexityinsert NAME IDremove IDsearch IDsearch NAMEprintInorderprintPreorderprintPostorderprintLevelCountremoveInorder N	**10 to >0.0 ptsDescribing worst case Big O complexity of each method1. State the variable under consideration (33.3%) 2. Define the variable (33.3%) 3. Justify complexities (33.3%) Note: The complexities must match what you did in the program0 ptsNo Marks**	10 pts
This criterion is linked to a Learning OutcomeComments	**1 to >0.0 ptsFull MarksCode has appropriate comments0 ptsNo Marks**	1 pts
This criterion is linked to a Learning OutcomeWhitespace	**1 to >0.0 ptsFull MarksCode has appropriate whitespace0 ptsNo Marks**	1 pts
This criterion is linked to a Learning OutcomeNaming convention	**2 to >0.0 ptsFull MarksCode follows a naming convention that is coherent and consistent0 ptsNo Marks**	2 pts
This criterion is linked to a Learning OutcomeDesign Decisions1. High cohesion2. Appropriate modularity3. Relevant access modifiers4. Proper memory management5. Efficient mechanisms for passing parameters in your function signatures6. The client (your int main() method, should not have objects that interact with memory directly)	**6 to >0.0 ptsFull MarksAll six aspects meet expectation High cohesion Appropriate modularity Relevant access modifiers Proper memory management**	6 pts

	Efficient mechanisms for passing parameters in your function signatures The client (your int main() method, should not have objects that interact with memory directly)0 ptsNo Marks	
Total Points: 100		