CMSC 341 - Project 2: Swarm of Search & Rescue Robots- Fall 2021

Due: Tuesday, Oct 26, before 9:00 pm

Objectives

- Implementing balanced binary search tree (BST) data structures.
- Practice writing rebalancing routines.
- Practice using recursion in programs.

Introduction

In large disaster zones finding and reaching people who need help is a huge operation. Among the challenges we can name the followings:

- The victims are spread across large areas. Therefore, a large number of rescuers and equipment are required for complete coverage.
- The disaster areas are covered with different types of obstacles such as water, mud, construction materials, fire, smoke, etc. Therefore, different types of skills and equipment are required.

A research team is trying to simulate a search and rescue operation using a swarm of robotic devices. The brain of every robot needs to be aware of the status of the whole team. You are assigned the task of developing a data structure that can store a database of all robots during the operation. In this application you use a balanced binary search tree to store the information for all robots. Every node in this BST represents a robot.

The Binary Search Tree (BST)

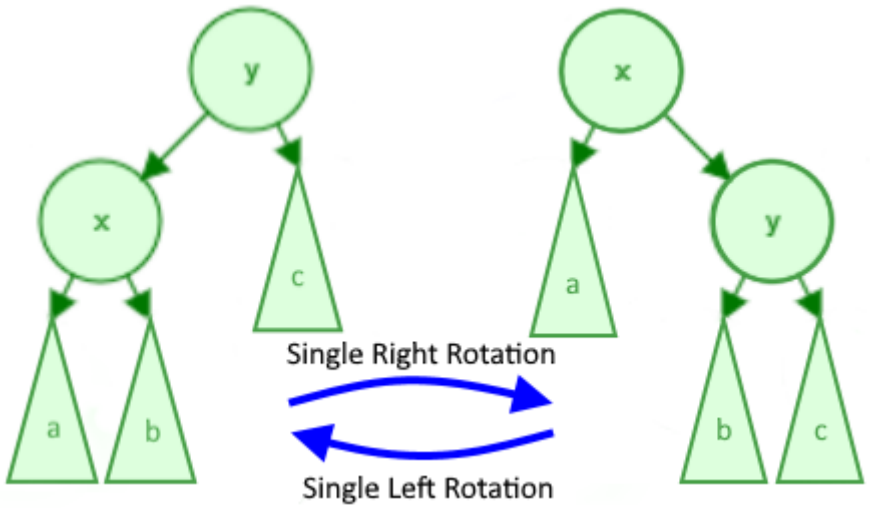
A binary tree is a tree structure in which each node has either 0, 1, or 2 children. A BST is a derivative of a binary tree where each node contains a key and value pair. The key determines the nodes' placement in the tree, and the value is the data to be stored. Given a set of rules about how to order the keys, we can create a structure where we can query data from it with a specified key. For a BST, we define these rules as follows:

1. If the target key is less than the key at the current node, traverse to the left child.
2. If the target key is greater than the key at the current node, traverse to the right child.
3. If the keys are equal, the action is determined by our application of the tree. More on this later.

A BST on its own can be efficient, but as the dataset increases in size, we can start running into problems. In the worst case, our BST can become a linked list where each of the new keys is greater than or less than the previous one inserted. On the contrary, the best case is inserting elements into the tree in a way to make it a complete tree. Either case is rare to occur with a large dataset, but imbalances are common. An imbalance can be defined when one subtree on a node becomes significnelty larger in size or height compared to the other subtree. As the tree becomes increasingly imbalanced, our average query times begin to increase. Luckily, we have methods to prevent large imbalances.

The AVL Tree

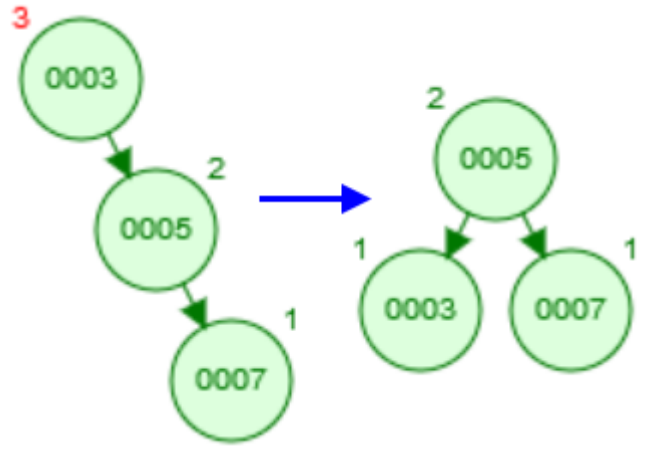
An AVL tree employs rotations during insertions or deletions to balance a BST. As the name implies, nodes are literally rotated up the tree to keep its structure complete. A complete tree, or ideally a perfect tree, is the most efficient kind of binary tree. Insertions, deletions, and queries all take O(log(n)) time in such a case. AVL trees have two types of rotations, left and right, which are shown in the diagram below:



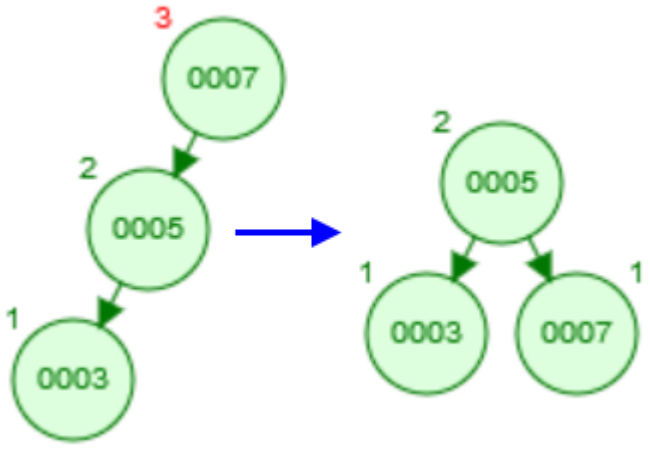
The variables "x" and "y" refer to 2 specific nodes whereas the subtrees "a", "b", and "c" refer to subtrees (which is just a pointer to a node which may or may not have more children). Note that the pointers to "a", "b", and/or "c" can be null, but "x" nor "y" will never be null.

The key to keeping an AVL tree efficient is when we perform these rotations. A rotation is performed on a node that is imbalanced, and an imbalance occurs when the node's children's heights differ by more than 1. For example, in the above diagram, consider node "y" to be imbalanced in the right rotation and node "x" to be imbalanced in the left rotation. Using a left and right rotation, we can perform four rotation combinations. The imbalance in the following examples occurs on the node with the height of 3 (in red).

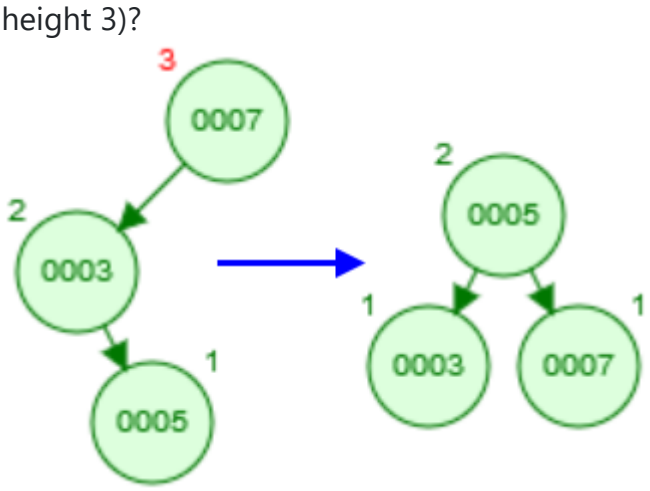
1. Single left rotation: This is a simple case where we can apply a left rotation to the top node to balance the tree.



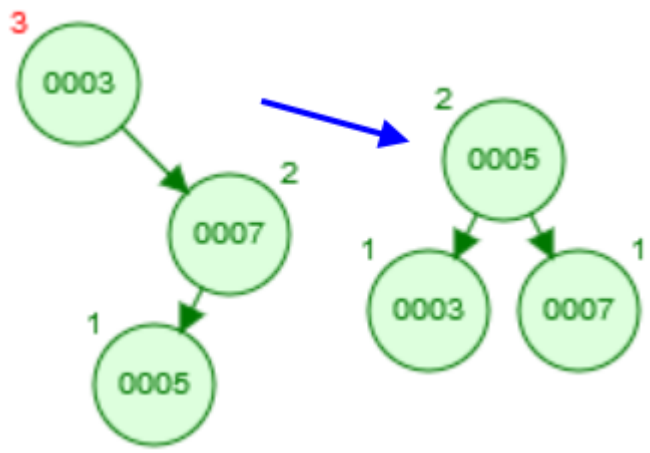
2. Single right rotation: Similar to the above case, we can apply a single right rotation to the top node to balance the tree.



3. Double left-right rotation: The following two cases become more complicated and require two rotations. In this example, the imbalance still occurs at the node with height 3. If we perform a single right rotation, we still end up with an unbalanced tree, just mirrored (draw a diagram). So, we must perform two rotations. The first left rotation should transform the tree into a form we can balance with a second right rotation. Which node should the first rotation be performed on (hint: it's not necessarily the node with height 3)?



4. Double right-left rotation: Likewise, this case uses a right rotation followed by a left rotation.



At most one rotation will occur during a rotation, and at least zero rotations will occur during a deletion. Also, it is not necessary to scan the entire tree after a change to the structure is made.

Assignment

Your assignment is to implement a binary search tree with balancing methods.

For this project, you are provided with the skeleton .h and .cpp files and a sample driver:

- swarm.h – Interface for the Swarm class.
- swarm.cpp – A skeleton for the implementation of the class Swarm.
- driver.cpp – a sample driver program. (**Note:** this file is provided to show a typical usage. Since the project is not implemented, trying to compile and run this driver program will not generate the sample output in driver.txt. Once you develop your project, you should be able to generate the same output as driver.txt by running this driver program.)
- driver.txt – a sample output produced by driver.cpp

Please note, you may not change any of the private variables or public function declarations or file names. Also, any provided function implementations may not be modified. You may, however, add your own private variables and functions. The current private function declarations are provided as a backbone to help you.

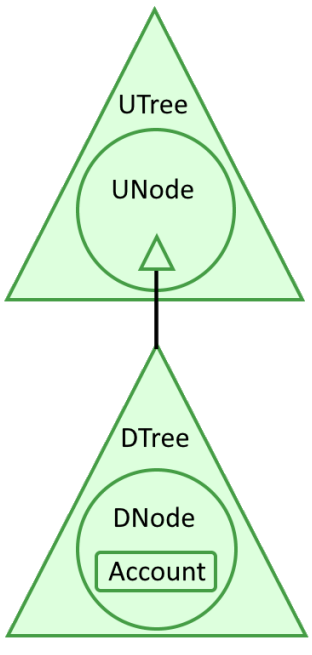
Additionally, you are responsible for thoroughly testing your program. Your test program, mytest.cpp, must be submitted along with other files. For grading purposes, your implementation will be tested on input data of varying sizes, including very large data. Your submission will also be checked for memory leaks and memory errors.

Specifications

This project has two classes: DTree and UTree. The class Utree is an AVL tree in which every node is of the type UNode. Every UNode has a pointer to a DTree object. The DTree class is a BST which will be rebalanced based on some specific criteria. The DTree object uses an array to rebalance. The nodes in the DTree object are of the type DNode. Every DNode holds an object of the type Account. The Account object holds the information for a user.

Architecture

This is a short explanation about how the tree structures come together to form a Discord account database. Discord allows multiple accounts to have the same username as long as they have different discriminators. The username tree (UTree) is the outer tree and each user node (UNode) must store multiple accounts, so it holds a discriminator tree (DTree). Since the discriminators attached to a username must be unique, the discriminator tree is the nested tree and each discriminator node (DNode) holds one account. However, accounts are allowed to share discriminators as long as they have different usernames, hence why each user node holds a unique discriminator tree. Below is a structure diagram:



As an example, if 2 accounts have the usernames and discriminators “john#1000” and “john#1004”, they would share the same UNode but have separate DNodes and Accounts.

Class UTree

The UTree class is short for a UserTree and implements an AVL tree. Each node in the UTree is called a UNode (short for UserNode). Each UNode holds a pointer to a DTree and some node attributes. For the BST traversal rules, use the username (stored in the Account) as the key. Similar to the DTree, the _root pointer points to the root UNode in the tree, and all UNodes should be dynamically allocated during insertion. Along with the DTree pointer, each UNode also contains a _left and _right UNode pointer and a _height member variable.

For the UTree class, you must implement the following methods in utree.cpp:

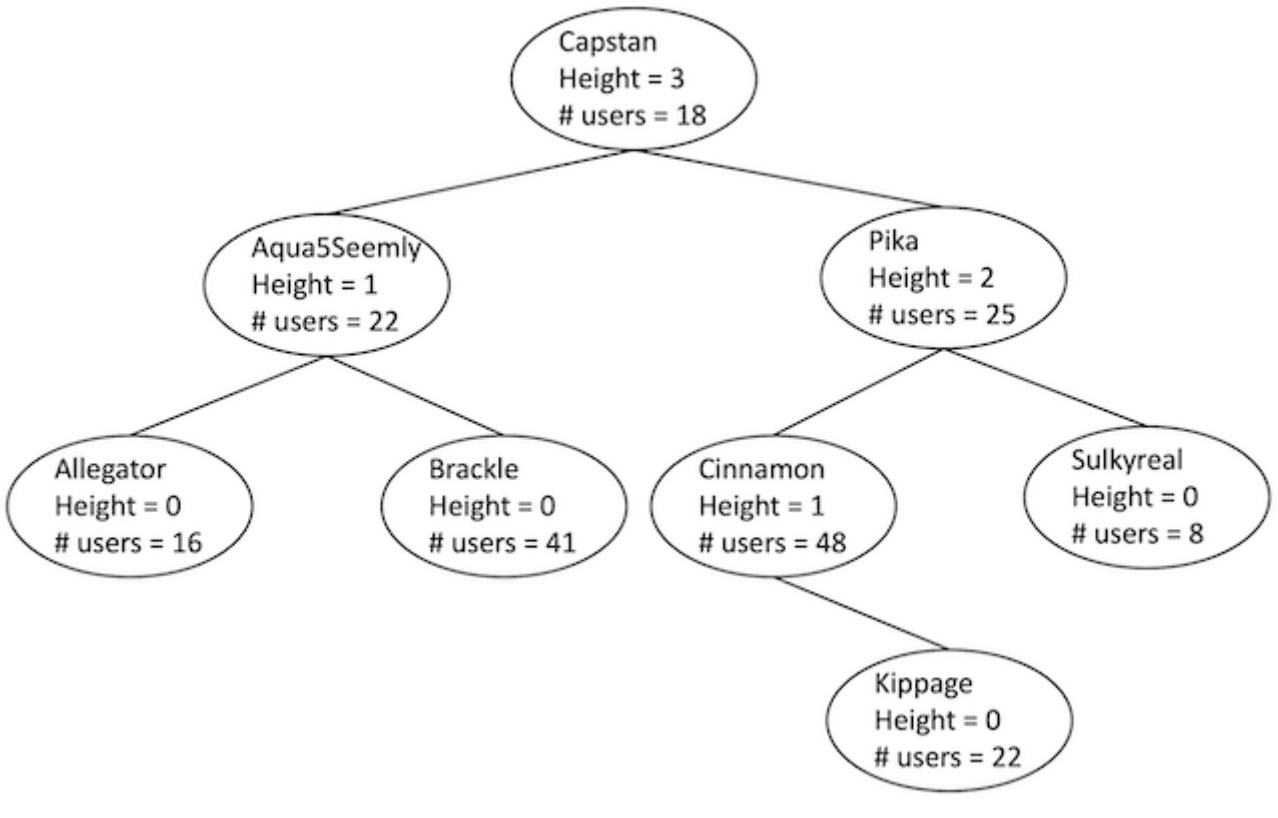
UTree::~UTree()	Destructor, must clean up all dynamic memory and nullify all pointers.
void UTree::loadData(string infile, bool append = true)	This function takes a path to a csv (comma-separated values) file that contains a database of Accounts in the infile parameter. Each line of the .csv file should contain Account information in the following format: username, discriminator, nitro, nickname, status. The username should be a string, discriminator an integer, nitro a 0 (false) or 1 (true), nickname an optional string, and status an optional string. If an account does not have a nickname or status, that field should be left blank. An example entry without a nickname is: “username, 3912, 0,,this is my status”. If append is false, then the tree should be cleared before inserting any data. Otherwise, the data in the .csv file will be appended to the existing data. **Note:** The implementation of this function is provided to you. You do not need to modify this function.
bool UTree::insert(Account newAcct)	This function inserts an Account object into the UNode with a matching username. If no UNode is found, one must be created. Then, the UNode should pass the Account object into its DTree’s insertion function. The username should be used as the key to traverse the UTree and abide by BST traversal rules. The comparison operators (<, >, ==, !=) work with the string type in C++. After an insertion, this function should also update the height of each node on the path traversed down the tree as well as check for an imbalance at each node in this path using checkImbalance(UNode*). If an imbalance is found, rebalance() should be called. Finally, this function should return true if the account was successfully inserted into the DTree. If the Account already existed, the insertion should fail and return false. (Hint: if you want to implement the functionality recursively, to facilitate a recursive implementation you can introduce a helper function.)
bool UTree::removeUser(string username, int disc, DNode*& removed)	This function removes the user specified by the username and discriminator. This function should locate the UNode with the matching username and delete the account with the matching discriminator from the DTree. The UNode should only be deleted if the DTree no longer contains any non-vacant nodes. Finally, the removeUser(...) function should return true if an Account was removed and false otherwise. For deleting a UNode you can write a helper function which its job is to remove a node in an AVL tree. The following presents a sample algorithm to delete a node: 1. If the node to delete has a left subtree, locate the largest node in the left subtree. This node will be referenced as node X moving forward. Node X can be found by traversing down to the left once and then as far right as possible. Copy node X’s value (DTree) into the node with the empty DTree. (Hint: you overloaded an operator to perform to help here). 2. If node X has a left child, the child will take node X’s place.

	3. Delete node X and, if it exists, shift the left child into its spot. This can also be done by copying node X's child's value into node X and deleting the child instead. 4. On the way back up the path taken to find node X, check for imbalances. 5. If the node with an empty DTree does not have a left subtree, shift its right child into its spot. After removing the UNode we should continue updating the heights and checking for imbalances as traversing back up the tree. For an interactive visualization of removal operation in an AVL tree you can visit https://www.cs.usfca.edu/~galles/visualization/AVLtree.html
UNode* UTree::retrieve(string username)	This function returns a UNode with a matching username. If no node with a matching username is found, this function returns nullptr. (Hint: if you want to implement the functionality recursively, to facilitate a recursive implementation you can introduce a helper function.)
DNode* UTree::retrieveUser(string username, int disc)	This function locates a UNode with a matching username and then queries the DTree for a DNode with a matching discriminator. If the username or the discriminator is not found, this function returns nullptr.
int UTree::numUsers(string username)	This function returns the number of valid Accounts (non-vacant) that share a specific username.
void UTree::clear()	Helper for the destructor to clear dynamic memory. (Hint: if you want to implement the functionality recursively, to facilitate a recursive implementation you can introduce a helper function.)
void UTree::printUsers() const	This function prints the Account details of all accounts in all DTrees. It calls the DTree::printAccounts() function. (Hint: if you want to implement the functionality recursively, to facilitate a recursive implementation you can introduce a helper function.)
void UTree::updateHeight(UNode* node)	This function updates the height of the node passed in. The height of a leaf node is 0. The height of all internal nodes can be calculated based on the heights of their immediate children.
int UTree::checkImbalance(UNode* node)	This function checks if there is an imbalance at the node passed in. For an imbalance to occur, the heights of the children node must differ by more than 1.
void UTree::rebalance(UNode*& node) **Or** **UTree* UTree::rebalance(UNode* node)**	This function begins and manages the rebalancing process. It is recommended to write additional helper functions to implement left and right rotations. You can use rebalance() function to determine which combination of rotations is necessary. For this function, you can either choose the function that passes node by reference or the one that returns a node. Both functions accomplish the same task and are a matter of personal style preference.

Additional Requirements

- **Requirement:** The class declarations (DTree) and (UTree) and provided function implementations in dtree.cpp and utree.cpp may not be modified in any way. No additional libraries may be used. However, additional "using" statements and private helper functions are permitted.
- **Requirement:** No STL containers or additional libraries may be used.
- **Requirement:** Your code should not have any memory leaks or memory errors.
- **Requirement:** Follow all coding standards as decribed on the C++ Coding Standards. In particular, indentations and meaningful comments are important.
- **Requirement:** The function UTree::dump(...) prints out the nodes information in an in-order traversal. For every node, it prints the username followed by the height of the node followed by the number of accounts (users) in the DTree. The following example, presents a sample output of the dump() function. **Note:** The implementation for this requirement is provided to you.

```
(((Allegator:0:16)Aqua5Seemly:1:22(Brackle:0:41))Capstan:3:18((Cinnamon:1:48(Kippage:0:22))Pika:2:25(Sulkyreal:0:8)))
```



- **Requirement:** The function DTree::dump(...) prints out the nodes information in an in-order traversal. For every node, it prints the discrimintaor number followed by the size of the node (size of the subtree) followed by the number of vacant nodes in the subtree.
  Note: The implementation for this requirement is provided to you.

Testing

Following is a non-exhaustive list of tests to perform on your implementation.

Note: Testing incrementally makes finding bugs easier. Once you finish a function and it is testable, make sure it is working correctly.

Testing DTree Class

- Test the overloaded assignment operator properly. You use it in another part of your project. If it is not working correctly, the caller of this function will not work correctly.
- Test whether the tree is balanced after a decent number of insertions. (Note: this requires visiting all nodes and checking the size values)
- Test whether the BST property is preserved after all insertions. (Note: this requires visiting all nodes and comparing key values)
- Test the remove functionality, here is a sample algorithm:
  1. Insert some nodes,
  2. Remove some of them (being tagged as vacant),
  3. Insert more nodes to force a rebalance,
  4. Check whether the vacant nodes are removed.
- Test the insertion and removal operations for edge cases.

Testing UTree Class

- Test whether the tree is balanced after a decent number of insertions. (Note: this requires visiting all nodes and checking the height values)
- Test whether the BST property is preserved after all insertions. (Note: this requires visiting all nodes and comparing key values)
- Test the remove functionality, here is a sample algorithm:
  1. Insert multiple accounts (usernames, discriminators)
  2. Remove all discriminators for a specific username (this makes all nodes of DTree vacant)
  3. Check whether the node for that username is removed from Utree
- Test the insertion and removal operations for edge cases.

Memory leaks and errors

- Run your test program in valgrind; check that there are no memory leaks or errors.
  Note: If valgrind finds memory errors, compile your code with the -g option to enable debugging support and then re-run valgrind with the -s and --track-origins=yes options. valgrind will show you the lines numbers where the errors are detected and can usually tell you which line is causing the error.
- Never ignore warnings. They are a major source of errors in a program.

Implementation Notes

- Implement incrementally based on the dependencies between classes and between the functions in a class.
- It'll be much more convenient if you first come up with a development plan.
- To compare keys in the UTree class (usernames) you can simply use overloaded comparison operators provided by C++ STL <string> library. The comparison operators (<, >, ==, !=) work with the string type in C++. These operators compare the strings lexicographically just like the way words are ordered in a dictionary.
- The lowest level of nodes which store the keys have zero height.
- In the Account class, we need to set _username and _disc variables. The trees are using these values as keys. Other variables in the Account class can be set to same defualt values for all accounts.

What to Submit

You must submit the following files to the proj2 directory.

- dtree.h
- dtree.cpp
- utree.h
- utree.cpp
- mytest.cpp - (**Note:** This file contains the declaration and implementation of your Tester class as well as all your test cases and a main function.)

  > This test file should compile, run to completion, and output the results of all test cases. The file should not be interactive, and it should not wait for the user input.

If you followed the instructions in the Project Submission page to set up your directories, you can submit your code using the following command:

```
cp dtree.h dtree.cpp utree.h utree.cpp mytest.cpp ~/cs341proj/proj2/
```

Grading Rubric

The following presents a course rubric. It shows how a submitted project might lose points.

- Conforming to coding standards make about 10% of the grade.
- Correctness and completeness of your test cases (mytest.cpp) make about 15% of the grade.
- Passing tests make about 30% of the grade.

If the submitted project is in a state that receives the deduction for all above items, it will be graded for efforts. The grade will depend on the required efforts to complete such a work.
