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.
- Practice writing unit tests.

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)

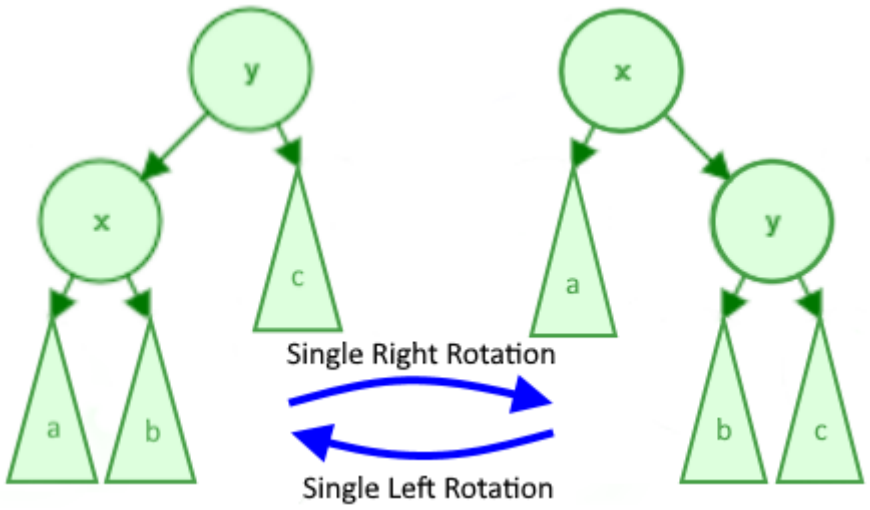
A binary tree is a tree structure in which each node has either 0, 1, or 2 children. A BST is a derivative of a binary tree where each node contains a key and value pair. The key determines the nodes' placement in the tree, and the value is the data to be stored. Given a set of rules about how to order 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

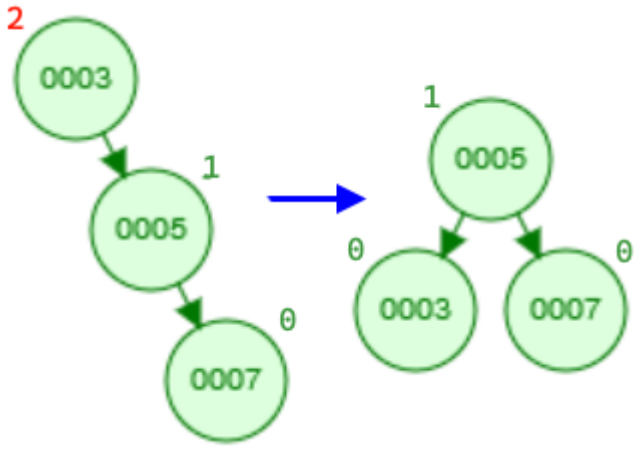
An AVL tree employs rotations during insertions or deletions to balance a BST. As the name implies, nodes are literally rotated up the tree to keep its structure complete. A complete tree, or ideally a perfect tree, is the most efficient kind of binary tree. Insertions, deletions, and queries all take 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:



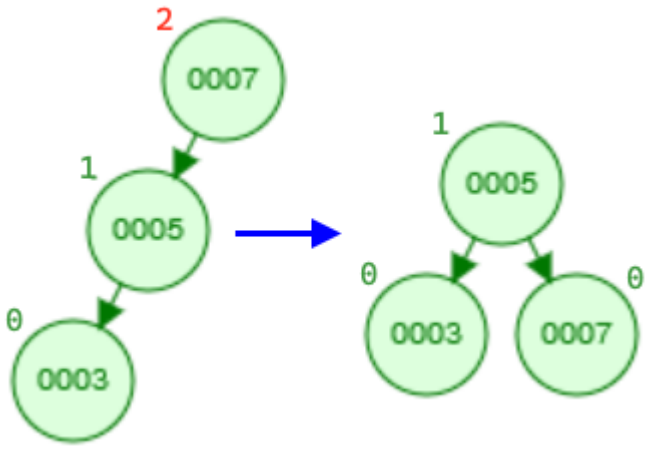
The variables "x" and "y" refer to 2 specific nodes whereas the subtrees "a", "b", and "c" refer to subtrees (which is just a pointer to a node which may or may not have more children). Note that the pointers to "a", "b", and/or "c" can be null, but "x" nor "y" will never be null.

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 2 (in red).

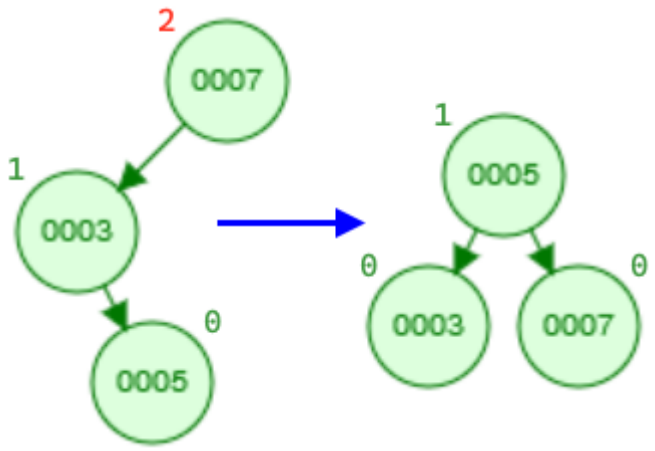
1. Single left rotation: This is a simple case where we can apply a left rotation to the top node to balance the tree.



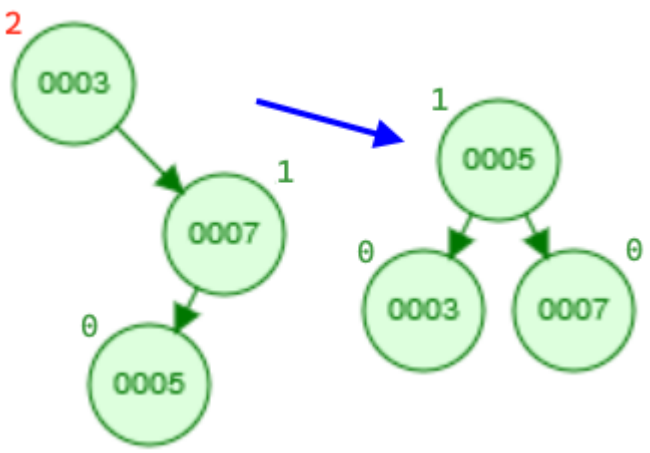
2. Single right rotation: Similar to the above case, we can apply a single right rotation to the top node to balance the tree.



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 2. 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 2)?



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 three classes: Random, Robot, and Swarm. The Random class is provided as a supplementary class to facilitate the testing. The Robot class constitutes the nodes in the binary tree. The Swarm class is the one that implements the balanced binary search tree.

Class Swarm

The Swarm class implements an AVL BST. The Swarm::m_root member variable points to a node of type Robot. Every node in Swarm is a Robot object. The nodes are organized as a binary search tree. The Swarm class supports the insertion and deletion operations. After insertion or deletion operations the class checks for any imbalance and if required it performs the rebalancing operations.

For the Swarm class, you must implement the following methods in swarm.cpp:

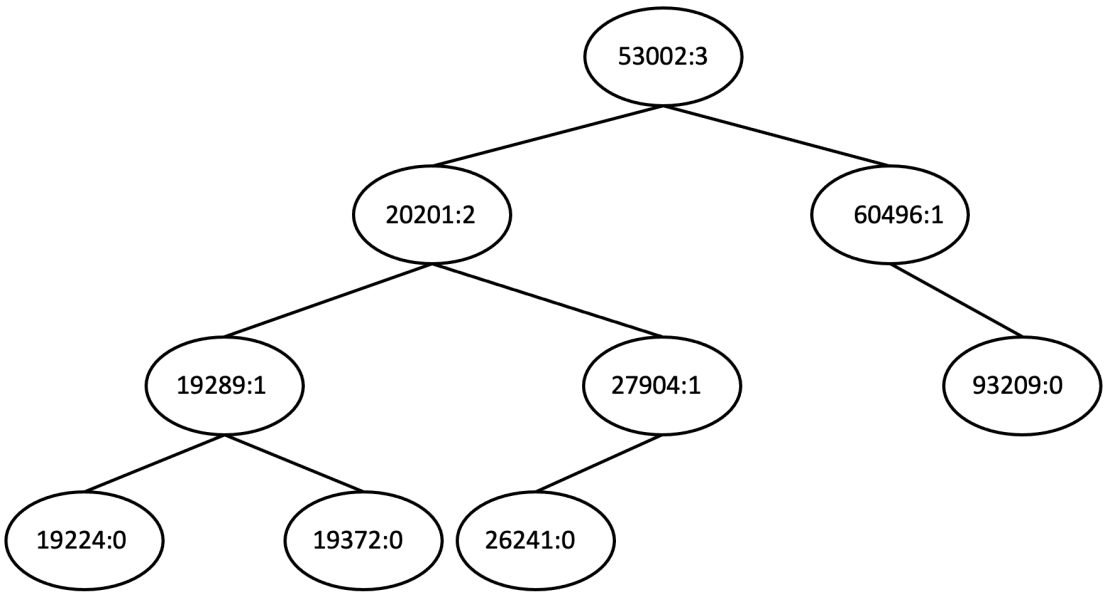
Swarm::Swarm()	Constructor, performs the required initializtions.
Swarm::~Swarm()	Destructor, performs the required cleanup including memory deallocations.
void Swarm::insert(const Robot& robot)	This function inserts a Robot object into the tree in the proper position. The Robot::m_id should be used as the key to traverse the Swarm tree and abide by BST traversal rules. The comparison operators (>, <, ==, !=) work with the int type in C++. A Robot id is a unique number, i.e. we do not allow duplicate id in the tree. Note: • In the Swarm tree data structure every node is a Robot object which is represented as a pointer to the Robot object. Therefore, for every insertion we need to allocate memory and use the information of robot to initialize the new node. Memory allocation takes place in the Swarm class.

	• After an insertion, we 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. (Hint: if you want to implement the functionality recursively, to facilitate a recursive implementation you can introduce a helper function.)
void Swarm::clear()	The clear function deallocates all memory in the tree and makes it an empty tree. (Hint: if you want to implement the functionality recursively, to facilitate a recursive implementation you can introduce a helper function.)
void Swarm::remove(int id)	The remove function traverses the tree to find a node with the id and removes it from the tree. (Note: After a removal, we 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.) (Hint: if you want to implement the functionality recursively, to facilitate a recursive implementation you can introduce a helper function.)
void Swarm::listRobots() const	This function prints a list of all robots in the tree to the standard output in the ascending order of IDs. The information for every Robot object will be printed in a new line. For the format of output please refer to the sample output file, i.e. driver.txt.
bool Swarm::setState(int id, STATE state)	This function finds the node with id in the tree and sets its Robot::m_state member variable to state. If the operation is successful, the function returns true otherwise it returns false. For example, when the robot with id does not exist in the tree the function returns false.
void Swarm::removeDead()	This function traverses the tree, finds all robots with DEAD state and removes them from the tree. The final tree must be a balnaced AVL tree.
void Swarm::updateHeight(Robot* aBot)	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 Swarm::checkImbalance(Robot* aBot)	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. This function returns the balance factor or the height difference.
Robot* Swarm::rebalance(Robot* aBot)	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.
bool Swarm::findBot(int id) const	This function returns true if it finds the node with id in the tree, otherwise it returns false.

Additional Requirements

- **Requirement:** The class declarations Swarm, Robot, Random and provided function implementations in swarm.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 Swarm::dumpTree(...) prints out the nodes information in an in-order traversal. For every node, it prints the id followed by the height of the node in the Swarm tree. The following example, presents a sample output of the dumpTree() function. **Note:** The implementation for this requirement is provided to you.

```
((((19224:0)19289:1(19372:0))20201:2((26241:0)27904:1))53002:3(60496:1(93209:0)))
```



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 Swarm Class

- Test the insertion function.
- 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 function.
- Test whether the tree is balanced after multiple removals.
- Test whether the BST property is preserved after multiple removals.
- Test the removeDead() functionality.
- Test whether the tree allows for storage of duplicates.
- Prove that the insertion performs in O(log n).
- Prove that the removal operation performs in O(log n).
- 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 the functions in a class.
- It'll be much more convenient if you first come up with a development plan.
- The lowest level of nodes which store the keys have zero height.
- Using helper functions is a convenient way to implement recursive functionality.
- The Random class is provided in the sample driver program. This class allows for generating a large amount of random numbers for testing purposes. You can directly copy this class to mytest.cpp.
- The required functionality is provided in the Robot class. There is no need for any modifications to this class.

What to Submit

You must submit the following files to the proj2 directory.

- swarm.h
- swarm.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 swarm.h swarm.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.
