Programming Assignment 2

In programming assignment 2 you will continue to build on the templated Binary Search Tree class from Week 9 lab. Compared to the Week 9 lab exercise, this time the BST class will use a Node class with more features. In addition to pointers to the left and right children, now the Node class will also have a pointer to the parent of a node. The parent of the root node is `nullptr` , and otherwise the parent pointer of a node will point to the parent of that node. Parent pointers will allow us to easily trace our way back up a tree, say to go from a leaf back to the root.

We add parent pointers in order to maintain the second new feature of the Node class: now the Node class has integer variable for the *height* of a node, which should always be equal to the height of the node in the tree. You will have to maintain this property as keys are added and deleted from the tree.

Like in the Week 9 lab, the BST class has two private member variables, `root_`,a pointer to the root of the tree, and `size_` , which maintains the number of keys in the tree.

There are 5 member functions to implement in this assignment. Recall that our class is templated with keys of type T.

1. `void insert(T k)`
   Insert the key `k` into the tree. Like in the week 9 lab, the BST should act like a `std::set` . If `k` is already in the tree then no action is taken. After insertion, the `size_` variable and the heights of nodes should be updated accordingly. Also now when adding the new node containing `k` to the tree its parent pointer needs to be set.
2. `Node* successor(T k)`
   Return a pointer to the node containing the minimum key in the tree larger than `k`. Return nullptr if `k` is the largest key in the tree or if `k` is not in the tree.
3. `void delete_min()`
   Remove the minimum key from the tree. Remember to update `size_` and heights of nodes accordingly. `delete_min` largely serves as a warmup for the next function, `erase`.
4. `void erase(T k)`
   Locate the node containing key `k` and remove it. If `k` is not in the tree, you do not have to do anything. Don't forget to update `size_` and heights of nodes accordingly.
5. `void rotate_right(Node* node)`
   Implement a *right rotation* about the node pointed to by `node` , as described in Lecture 8.6. This will only be called when `node` has a left child. If `left_child` points to the left child of `*node`, then `*left_child` becomes the parent of `*node` , and the right subtree of `*left_child` becomes the left subtree of `*node`. Node heights should again be properly updated after this operation.

The Code

There are three files provided in the scaffold.

1. The header file bst.hpp contains the definitions of all member functions of the BST class and instructions about which functions need to be implemented. **All of your implementations should be written in this file.** This is the only file that is tested when "mark" is pressed.
2. The header file unit_tests.hpp contains tests that you can use to check the implementation of your functions. These are the same tests that are checked when you press the "mark" button. We largely recommend that you use the tests here as you are implementing the functions. This way you can test one function at a time, and can add couts as needed to the tests to help in your debugging. The error messages resulting from "mark" are also sometimes not very helpful. Once you regularly pass the tests in unit_tests.hpp you should also pass them in the marker as they are the same tests, although note that many tests are randomised.
3. The final file is bst.cpp. This is the file that is compiled when "Run" is pressed. You can modify this file as you like. We have provided a skeleton with the names of all the tests in unit_tests.hpp.

Testing

There are 15 tests that are run on your code. The first 4 tests are on the insert function. Then are are 2 tests of delete_min and 2 tests of successor. Next are 4 tests of the erase function, and finally 3 tests of the rotate_right function.

1. test_insert_string: Creates a BST with template type std::string. Checks that the size is correctly updated and BST property holds after inserting some strings. All other tests instantiate the template type with ints.
2. test_insert_size: Checks that size behaves properly when duplicate keys are tried to be inserted into the tree. Remember the BST should not insert a key again that is already in the tree.
3. test_insert_values: Tests the BST property after inserting some ints.
4. test_insert_heights: Tests that the node heights are correct after inserting some ints.
5. test_delete_min: Inserts some ints into the tree and then repeatedly calls delete_min until tree should be empty.
6. test_delete_min_heights: Tests that heights are updated properly after delete_min
7. test_successor: Tests that the successor function returns a pointer to the correct node.
8. test_successor_max: Tests the successor function on the largest key in the tree (nullptr should be returned).
9. test_erase: Tests that BST property is maintained after erasing a random key.
10. test_erase_heights: Tests that heights are updated properly after erasing a random key
11. test_erase_root: Tests erasing the key at the root.
12. test_erase_successor_child: Tests the case where we erase the key at a node whose successor is their right child.
13. test_rotate_right: Tests parent/child relationships after calling rotate_right on a random node that has a left child. Also tests that BST property is preserved.

14. test_rotate_root: Calls rotate_right on the root.
15. test_rotate_heights: Checks that heights are updated properly after calling rotate_right on a random node that has a left child.

Marking

The assignment will be marked against three components:

Functionality (26%) will be marked automatically when you press the "*mark*" button and will be verified by the teaching team. Each of the 15 tests has equal weight so the total percentage you receive for functionality will be (n/15)*26% if you pass n tests.

Design (6%) will be marked by your tutor, and all tutors will follow the same rubric as follows.

The 5 member functions you are to implement, insert, successor, delete_min, erase, rotate_right, will each be given weight 1.2% in the design score. Each member function will be scored for

1) Time complexity: All member functions should work in time O(h), where h is the height of the tree. The time complexity score contributes 0.4%.

2) Bugs: the tests do not catch all errors. The tutors will additionally look for bugs when reading your code: an edge case that would not pass, a potential segmentation fault, memory not being freed properly in delete_min or erase, etc. This contributes 0.4% to the member function score.

3) Succinctness: The path to the solution should be clear and direct. No more code is used than necessary. This contributes 0.4%.

Style (3%) will also be marked by your tutor, and all tutors will follow the same rubric as follows.

There are 4 categories for style, each will be given 0.75%.

1) comments --- comments should explain blocks of code where the intention is not immediately clear from the code itself.

2) variable names --- good naming that gives an idea of the role of a variable.

3) formatting --- consistent formatting as far as indentation, use of braces, snake case vs. camel case, etc.

4) best practices --- For example: variables are defined close to where they are used. The scope of variables is not larger than needed.

The style score is prorated by the number of member functions implemented.

Those who have in-person tutorials can demonstrate your code and get immediate feedback when you are being marked for *design* and *style* in weeks 11 and 12. For

those who have an online tutorial, your tutor will contact you through Ed with any queries about your code.

Submission

You must submit your code by pressing the "*mark*" button on Ed. No other forms of submission will be accepted. You are welcome to develop your code elsewhere, but remember it must compile and run on Ed. The underlying g++ compiler in Ed is set to C++17 for this assessment. Usually, code that compiles with clang++ should also work with g++.

You may submit as many times as you like before the due date.

Due Date

Code submission is due by 16 May 2022 at 12:00 AM. A 10% penalty applies to all late submissions made within a week after the due date. No code submissions will be accepted after 23 May 2021.

Plagiarism Checking

All code will be checked for student misconduct and plagiarism using specialised code similarity detection software.