CSCE 221 Spring 2021

Programming Assignment 4

Red-Black Trees

Approved Includes

```
<cstddef>
<iostream>
<sstream>
<stdexcept>
<utility>
<tuple>
"red_black_tree.h"
```

Code Coverage

You must submit a test suite for each task that, when run, covers at least 90% of your code. You should, at a minimum, invoke every function at least once. Best practice is to also check the actual behavior against the expected behavior, e.g. verify that the result is correct. You should be able to do this automatically, i.e. write a program that checks the actual behavior against the expected behavior.

Your test suite should include ALL tests that you wrote and used, including tests you used for debugging. You should have MANY tests.

Starter Code

```
compile_test.cpp
Makefile
red_black_tree.h
red_black_tree_tests.cpp
```

Files to Submit

```
red_black_tree.h
red_black_tree_tests.cpp
```

Task

Implement a Red-Black tree (auto-balancing binary search tree).

You can implement a top-down or bottom-up Red-Black tree. The top-down tree is more efficient but, since it must be done iteratively as opposed to recursively, the code will be more complex and therefore more prone to errors.

Requirements

Files

`red_black_tree.h` - contains the template definitions
`red_black_tree_tests.cpp` - contains the test cases and test driver (main)

Class

```
template <typename Comparable>
class RedBlackTree;
```

Member types (public)

`enum Color {RED,BLACK}` - member constants for node color, e.g. `RedBlackTree<T>::RED`
`struct Node` - must have attributes Comparable value, Node* left, Node* right, and int color (in any order). The name of the class does not need to be Node.

Functions (public)

Constructors & Rule of Three

`RedBlackTree()` - makes an empty tree
`RedBlackTree(const RedBlackTree&)` - constructs a copy of the given tree
`~RedBlackTree()` - destructs this tree
`RedBlackTree& operator=(const RedBlackTree&)` - assigns a copy of the given tree

Element Access

N/A

Iterators

Optional

Capacity

Optional

Modifiers

`void insert(const Comparable&)` - insert the given lvalue reference into the tree
`void remove(const Comparable&)` - remove the specified value from the tree (use minimum of right child tree when value has two children)

Lookup

`bool contains(const Comparable&) const` - returns Boolean true if the specified value is in the tree
`const Comparable& find_min() const` - return the minimum value in the tree or throw `std::invalid_argument` if the tree is empty
`const Comparable& find_max() const` - return the maximum value in the tree or throw `std::invalid_argument` if the tree is empty

Visualization

Optional, but strongly recommended.

For Testing

`int color(const Node* node) const` - return the color of the specified node (the null pointer is black). This function is required to be public by the instructor's tests.
`const Node* get_root() const` - return the root of the tree or `nullptr` if the tree is empty. This function is required by the instructor's tests.

Optional

`RedBlackTree(RedBlackTree&&)` - move constructs a copy of the given (rvalue) tree
`RedBlackTree& operator=(RedBlackTree&&)` - move assigns a copy of the given (rvalue) tree
`bool is_empty() const` - returns Boolean true if the tree is empty
`void insert(Comparable&&)` - insert the given rvalue reference into the tree using move semantics
`void make_empty()` - remove all values from the tree
`void print_tree(std::ostream&=std::cout) const` - pretty print the tree

Example

```
// make an empty tree
RedBlackTree<int> tree;

// insert 5 values into the tree
std::cout << "insert 6, 4, 2, 8, 10 " << std::endl;
tree.insert(6);
tree.insert(4);
tree.insert(2);
tree.insert(8);
tree.insert(10);

// print the tree (this is not executed on Gradescope)
std::cout << "tree: " << std::endl;
{
  std::ostringstream ss;
  tree.print_tree(ss);
  std::cout << ss.str() << std::endl;
}

std::cout << "contains 4? " << std::boolalpha <<tree.contains(4) << std::endl;
std::cout << "contains 7? " << std::boolalpha << tree.contains(7) <<
std::endl;

// remove the root
std::cout << "remove 4" << std::endl;
tree.remove(4);

// find the minimum element
std::cout << "min: " << tree.find_min() << std::endl;

// find the maximum element
std::cout << "max: " << tree.find_max() << std::endl;

// print the tree (this is not executed on Gradescope)
std::cout << "tree: " << std::endl;
{
  std::ostringstream ss;
  tree.print_tree(ss);
  std::cout << ss.str() << std::endl;
}
```

Example Output

```
insert 6, 4, 2, 8, 10
tree:
      R:10
  8
      R:6
4
  2
contains 4? true
contains 7? false
remove 4
min: 2
max: 10
tree:
      R:10
  8
6
  2
```

Red-Black Tree Insert and Remove Help

1. The textbook has the cases. The rules are different for bottom-up and top-down trees, so it depends on which way you go. Assuming you go top-down (which is recommended for efficiency), we have the following rules for insert:
    1. on the way down, a black parent with 2 red kids → flip the colors
        1. if the grandparent is also red, do the appropriate AVL rotation and re-color (new "root" is black, kids are red)
    2. after insert, if the parent is red, do the appropriate AVL rotation around grandparent and re-color (new "root" is black, kids are red)
2. For remove, the idea is to make sure the node we eventually remove (which will be a leaf because removing an internal node is just a value replacement) will be red
    1. turn the parent of the root red (pretend the root has a parent and it is red)
    2. on the way down
        1. the parent is always red (this is the property we maintain)
        2. therefore the current node and its sibling are black
        3. if the current node has 2 black kids, there are 4 cases:
            1. sibling also has 2 black kids → flip the colors
            2. sibling has a red kid on the inside (near the current node) → AVL double rotation to bring that red kid up to the root of the subtree (where parent is now) and re-color (red kid stays red, parent and sibling are black, current node is red)
            3. sibling has a red kid on the outside (away from the current node) → AVL single rotation to bring the sibling up to the root of the subtree (where parent is now) and re-color (sibling turns red, parent and red kid turn black, current node turns red)
            4. if the sibling has two red kids, then you can do either rotation (i.e. choose case 2 or 3).
        4. else, the current node has a red kid so we fall through to the next level and hope we land on the red kid so we can continue (fall through again because we want the parent to be red)
            1. if we land on the black kid, we know the parent is black and sibling is red, so we rotate the parent with the sibling to make the current node's new parent red and its grandparent black. then we continue from the first main case (parent is red, current and sibling are black)