CSCE 221 Spring 2021

Programming Assignment 3

Binary Search Tree and AVL Tree

Approved Includes

```
<cstddef>
<iostream>
<sstream>
<stdexcept>
<utility>
"avl_tree.h"
"binary_search_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.

Your test suite should include ALL tests that you wrote and used, including tests you used for debugging. You should have MANY tests (about 3-4 times as many lines of test code as you have lines of functional code).

Starter Code

```
avl_tree.h
avl_tree_tests.cpp
binary_search_tree.h
binary_search_tree_tests.cpp
build_a_tree.cpp
compile_test.cpp
Makefile
```

You should **not** modify `build_a_tree.cpp`.

Files to Submit

```
avl_tree.h
avl_tree_tests.cpp
binary_search_tree.h
binary_search_tree_tests.c
```

Task 1

Implement a binary search tree.

Requirements

Files

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

Class

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

Functions (public)

`BinarySearchTree()` - makes an empty tree

```
+--Rule of Three-------------------------------------------------+
| BinarySearchTree(const BinarySearchTree&) - copy constructor   |
| ~BinarySearchTree() - destructor |                             |
| BinarySearchTree& operator=(const BinarySearchTree&) - copy assignment operator |
+----------------------------------------------------------------+
```

`bool contains(const Comparable&) const` - returns Boolean true if the specified value is in the tree
`void insert(const Comparable&)` - insert the given value into the tree
`void remove(const Comparable&)` - remove the specified value from the tree (replace with minimum of right child tree when value's node has two children)
`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
`void print_tree(std::ostream&=std::cout) const` - pretty print the tree (rotated 90 degrees anti-clockwise, two spaces per level; see example below) to the specified output stream (default `std::cout`). Print "`<empty>\n`" if the tree is empty.

Optional

`BinarySearchTree(BinarySearchTree&&)` - move constructs a copy of the given (rvalue) tree
`BinarySearchTree& operator=(BinarySearchTree&&)` - 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 into the tree using move semantics
`void make_empty()` - remove all values from the tree

Example

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

// insert 5 values into the tree
tree.insert(6);
tree.insert(4);
tree.insert(2);
tree.insert(8);
tree.insert(10);

// search the tree
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
tree.remove(6);

// 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
std::cout << "tree: " << std::endl;
tree.print_tree();
```

Example Output

```
contains 4? true
contains 7? false
min: 2
max: 10
tree:
  10
8
  4
    2
```

Task 2

Implement an AVL tree (auto-balancing binary search tree).

Requirements

Files

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

Class

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

Functions (public)

`AVLTree()` - makes an empty tree

```
+--RUle of Three-----------------------------------------+
| AVLTree(const AVLTree&) - copy constructor             |
| ~AVLTree() - destructor                                |
| AVLTree& operator=(const AVLTree&) - copy assignment operator |
+--------------------------------------------------------+
```

`bool contains(const Comparable&) const` - returns Boolean true if the specified value is in the tree
`void insert(const Comparable&)` - insert the given value into the tree
`void remove(const Comparable&)` - remove the specified value from the tree (replace with minimum of right child tree when value's node has two children)
`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
`void print_tree(std::ostream&=std::cout) const` - pretty print the tree (rotated 90 degrees anti-clockwise, two spaces per level; see example below) to the specified output stream (default `std::cout`). Print "`<empty>\n`" if the tree is empty.

Optional

`AVLTree(AVLTree&&)` - move constructs a copy of the given (rvalue) tree
`AVLTree& operator=(AVLTree&&)` - 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 into the tree using move semantics
`void make_empty()` - remove all values from the tree

Example

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

// insert 5 values into the tree
tree.insert(6);
tree.insert(4);
tree.insert(2);
tree.insert(8);
tree.insert(10);

// search the tree
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
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
std::cout << "tree: " << std::endl;
tree.print_tree();
```

Example Output

```
contains 4? true
contains 7? false
min: 2
max: 10
tree:
    10
  8
6
  2
```

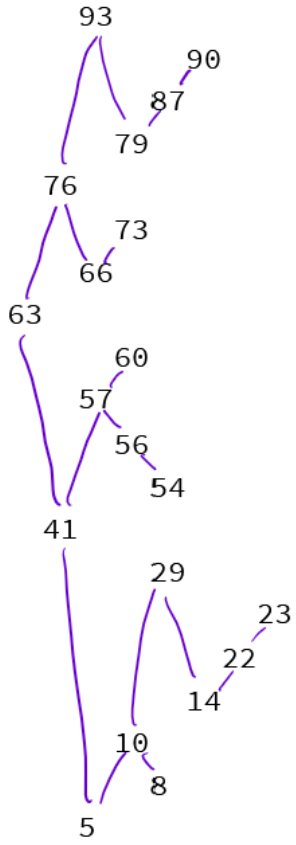
Bigger Example of Print Tree

```
int A[] = {63, 41, 76, 93, 66, 5, 10, 57, 8, 79, 29, 14, 73, 56, 54, 87, 60,
22, 23, 90};
BinarySearchTree<int> tree;
for (size_t index = 0; index < 20;; index++) {
  tree.insert(A[index]);
}
tree.print_tree();
```

Bigger Example Output

```
    93
          90
        87
      79
  76
      73
    66
63
      60
    57
      56
        54
  41
        29
              23
            22
          14
      10
        8
    5
```

(a) What you see on the console



(b) What you see in your mind

How To Measure Coverage with Gcov

Compile with coverage

```
g++ -std=c++17 -g --coverage <source files>
```

Run

```
./a.out
```

Generate coverage report

```
gcov -mr <source file>
```

View coverage report

```
cat <source file>.gcov
```

```
‘-’ means the line is not executable (does not count for coverage)
‘#####’ means the line is executable but was executed 0 times
‘126’ means the line was executed 126 times
```

Identify lines which are not covered

```
grep “#####” <source file>.gcov
```

Clean up before next measurement

```
rm -f *.gcov *.gcno *.gcda
```