PIC 10C Section 1 - Homework # 5 (due Friday, April 29, by 11:59 pm)

You should upload each .cpp and/or .h file separately and submit them to CCLE before the due date/time! Your work will otherwise not be considered for grading. Do not submit a zipped up folder or any other type of file besides .h and .cpp.

Be sure you upload files with the precise name that you used in your editor environment otherwise there may be linker and other errors when your homeworks are compiled on a different machine.

Also be sure your code compiles and renders the correct output on g++ with -O3 optimization on the C++20 Standard!!!

TEMPLATED RED BLACK TREE

In this homework, you will get to write your very own red black tree! The course notes provide sketches of every possible case you'll need to deal with. For this homework, the **only header files you are allowed to use** are:

- **utility**
- **stdexcept**
- **iostream**

You will submit **rbt.h**. Please refer to the syllabus for how the work will be graded: note that more than half of the marks come from good coding practices and code documentation. **Beware of memory leaks!!!** You may find these files helpful in detecting basic memory leaks.

You can assume the code will be compiled with a **makefile**:

```
hw5: main.cpp rbt.h
    g++-8 -std=c++2a -O3 main.cpp -o hw5
```

for a **main.cpp** that could differ from the example given! The file Timer.h referenced in the main routine is found here.

The description of the **rbt** class follows. Note that you still need to write **node**, **iterator**, and **const_iterator** classes although those details are more standard. You should be able to use your homework #2 as a starting point for this homework, simply adding a few extra member functions.

The **rbt** class must:

- be defined within the **pic10c** namespace;
- be templated by the type of data it stores, **T**, and a comparison operator, **compare_type**, which should be **std::less<T>** by default;
- store a **node*** for its **root**, a **compare_type** called **pred** as its predicate, and a variable to track its size;
- have a single constructor with a single argument such that if no arguments are provided, **pred** will be set to compare_type{} and otherwise uses the value provided as the value for **pred**;
- have a destructor;
- have copy and move constructors;
- have copy and move assignment operators;
- have a **swap** function (visible at the **pic10c** namespace level) to swap two **bst**s;
- have an **insert** member function, suitably implement to take in a **T** and attempt to add the value to the tree, dealing with rvalues and lvalues appropriately;
- have an **emplace** function that accepts a variadic list of arguments to construct a **T** and attempt to place it within the tree;
- have an **erase** member function accepting an **iterator** (but not a **const_iterator**) and removing the **node** managed by the **iterator** from the tree;
- have **begin** and **end** member functions returning a **(const-)iterator** to the first **node** and a **(const-)iterator** *one past the final* **node**, respectively — these must be overloaded on const;
- have a **size** member function returning the number of elements in the **bst**;
- have a **find** member functions returning the **(const-)iterator** to the node with a given value if found and otherwise returning the past-the-end **(const-)iterator** – overloaded on const; and
- have a **print** function (as you can see in the sample output) to display the red black tree structure with '(r)' or '(b)' indicating a node is red or black. The - indicates the root, a / indicates being a right child and a \ indicates being a left child.

iterator and **const_iterator** must also be a nested class within **bst** and it must:

- overload the prefix and postfix version of ++;
- overload the prefix and postfix version of --;
- overload == and != as comparison operators;
- overload the dereferencing operator; and
- overload **operator arrow**.

A test case, code and output, are provided below.

```
#include "rbt.h"
#include "Timer.h"
#include<iostream>
#include<vector>
#include<string>

auto get_rbt() {
  pic10c::rbt<double, std::greater<double>> vals;
  vals.insert(3.3);
  vals.insert(1.1);
  vals.insert(4.4);
  vals.insert(5.3);
  vals.emplace(1.1); // duplicate
  vals.emplace(); // adds 0
  return vals;
}

int main() {

  // basic inserting, handling duplicates,  etc.
  pic10c::rbt<std::string> colours;
  colours.insert("red");
  colours.insert("orange");
  colours.insert("yellow");
  colours.insert("green");
  colours.insert("blue");
  colours.insert("indigo");
  colours.insert("green"); // dupicate
  colours.insert("violet");
  std::cout << "colours size: " << colours.size() << '\n';

  // print the structure...
```

```
std::cout << "colours current structure:\n";
colours.print();

// check find
std::vector<std::string> cols{ "red", "cherry", "green" };

// green will be there... and it has
std::cout << "green has " << colours.find(cols.back())->size() << " characters.\n";

// try to erase all these
for (const auto& s : cols) {
  if (auto p = colours.find(s); p != colours.end()) {
    colours.erase(p);
  }
}

std::cout << "colours new structure:\n";
colours.print();

// call a function to get rbt
const auto doubles = get_rbt();

// print the doubles
std::cout << "printing the doubles:\n";
for (const auto& d : doubles) {
  std::cout << d << '\n';
}

pic10c::rbt<int> ints;
pic10c::rbt<int> ints2;

pic10c::swap(ints, ints2);

std::cout << "now we do some time trails...\n";

simple_timer::timer<'u'> t;

std::cout << "time each of 1000 inserations:\n";
for (int i = 0; i < 1000; ++i) {
  t.tick();
  ints.insert(i);
  std::cout << t.tock() << ' ';
}
std::cout << '\n';
```

```
  std::cout << "time each of 1000 removals:\n";
  for (int i = 0; i < 1000; ++i) {
    t.tick();
    auto p = ints.find(i);
    ints.erase(p);
    std::cout << t.tock() << ' ';
  }
  std::cout << '\n';

  return 0;
}
```

An example of the output is:

```
colours size: 7
colours current structure:

          /yellow(b)

                    \violet(r)

-red(b)

                    /orange(b)

                              \indigo(r)

          \green(r)

                    \blue(b)
green has 5 characters.
colours new structure:

          /yellow(b)

-violet(b)

                    /orange(b)

          \indigo(r)

                    \blue(b)
printing the doubles:
```

```
5.3
4.4
3.3
1.1
0
now we do some time trails...
time each of 1000 inserations:
10.7us 6.9us 6.5us 20.9us 10.2us 11.5us 8.8us 8.5us 8.3us 7.2us 5.9us 7.5us 7.1us 6.8us
time each of 1000 removals:
18us 11us 8.5us 5.5us 8.5us 12.4us 10us 6.2us 9.7us 5.8us 6.8us 9.5us 13.6us 13us 11.6u
```

Some gets cut off in this display so there's also a text file **output.txt** so you can see all the times. The really amazing thing about the times is that there is virtually no dependence on the size of the red black tree even with values coming in sorted: as it grows, the time barely changes. Try doing the same thing with your plain old binary search tree and it won't fare so well...

Some hints:

- You could start by taking your binary search tree homework and defining an **enum class Colour** and adding **Colour** member variable to the **node**s.
- It is very, very useful to compute the colour, parent, grandparent, uncle, and sibling of a node. So write getters like **get_uncle**, etc. Feel free to throw a **std::logic_error** if the node does not exist, i.e., if a node has **nullptr** for a parent then there is no grandparent. Your program logic should ensure you never access a node that doesn't exist and it is way better to have an exception get thrown than to have undefined behaviour.
- Now you can write the **rotateLeft** and **rotateRight** functions. Recall that if the root of the tree is rotated about then there is a new root. If you make them member functions of **node** then it would be advisable to accept a parameter **node*&r** where **r** is a reference to the **node*** for the root of the tree.
- For the **print** function, you can think of it as follows: each node prints everything to its right(will appear above the node), then prints itself, then prints everything to its left (will appear below the node). This is a recursive process and every time a recursive call is made to go deeper, the indentation should increase.
- Assuming every node is red upon construction (should be) then you shoul be able to compile your code and run it as an ordinary binary search tree and print it with all nodes red. You can also test the effects of the left and right rotations on the root and printing the tree again. Until this works, there is no reason to do anything else.
- Now that the previous steps are done, the **insert** function should simply make an additional call to **correct_double_red** to fix possible double reds after insertion (this correction function can be recursive or iterative).

- Check the inserting works.
- Now for the removal... to **erase**, you should simply colour a black node that is being destroyed double-black, call a **correct_double_black** upon it, and then **delete** it.