PIC 10B Section 1 - Homework # 8 (due Wednesday, May 29, by 6 pm)

You should upload each .cpp and .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 requested of you and that it matches the names 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 runs on Visual Studio 2017.

At the end of this work you should submit a header file **BinarySearchTree.h** with class interfaces and function declarations and **BinarySearchTree.cpp** that includes all the implementations.

Do not submit a main routine!

You are to write three separate classes that belong to a namespace **pic10b**. You should be writing a **Tree** class, as the binary search tree data structure that will store **double** values; a **node** class, to store the data in nodes where each node tracks its left/right child nodes *as well as its parent node*; and an **iterator** class to traverse the tree.

The **node** and **iterator** classes should be nested inside of the **Tree** class. For simplicity, we will use "iterator" as the only iterator-type class, but it will serve as a **const_iterator**.

The code **must be free of memory leaks**; when a **Tree** is destroyed or assigned-to, the managed memory must be freed. The precise manner you implement these classes could vary but at the very least you must adhere to the requirements below and ensure that your code has the same behaviour as the example shown.

Tree must:

- have a default constructor creating a tree storing nothing;
- have a destructor;
- have copy and move constructors;
- have copy and move assignment operators;
- have a **swap** function (visible at the **pic10b** namespace level) to swap two **Tree**s;
- have an **insert** member function accepting a **double** value and adding it to the tree;
- have an **erase** member function accepting an **iterator** and removing the **node** managed by the **iterator** from the tree;
- have **begin** and **end** member functions returning an iterator to the first **node** and an iterator *one past the final* **node**, respectively;
- have a **size** member function returning the number of elements in the **Tree**; and
- have a **find** member function returning the **iterator** to the node with a given value if found and otherwise returning the past-the-end iterator.

iterator must:

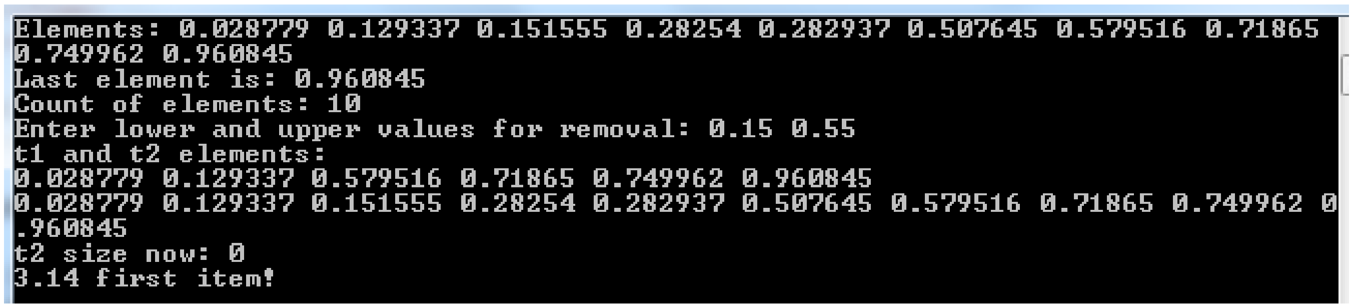
- overload the prefix and postfix version of ++;
- overload the prefix and postfix version of --;
- overload == and != as comparison operators; and
- overload the dereferencing operator.

An example of the desired output for the code below is provided. Note that due to the random numbers involved, your runs of the program will not generate the identical output.

```
#include "BinarySearchTree.h"
#include<ctime>
#include<iostream>

int main(){
  std::srand(static_cast<unsigned>(std::time(nullptr))); // seed

  pic10b::Tree t1; // empty tree
```

```
for (size_t i = 0; i < 10; ++i) { // add 10 random double's from 0 to 1
  t1.insert(static_cast<double>(std::rand()) / RAND_MAX);
}

std::cout << "Elements: "; // and print the elements
for (auto itr = t1.begin(), past_end = t1.end(); itr != past_end; ++itr) {
  std::cout << *itr << " ";
}
std::cout << '\n';

auto past_end = t1.end(); // go past the end
--past_end; // but decrement to the last element
std::cout << "Last element is: " << *past_end << '\n'; // show the last element

std::cout << "Count of elements: " << t1.size() << '\n'; // count elements in t1

pic10b::Tree t2 = t1; // t2 is a copy of t1

double low, up; // lower and upper bounds for value removal

std::cout << "Enter lower and upper values for removal: ";
std::cin >> low >> up; // read in values

auto itr = t1.begin();
while (itr != t1.end()){ // while not at the end
  if ((low <= *itr) && (*itr <= up)){ // check if node value in range
    t1.erase(itr); // if so, erase it
    itr = t1.begin(); // and go back to the beginning
  continue; // repeat the loop, ignoring the increment
  }
  ++itr; // if not in range then increment the iterator
}

// List all the elements of the two trees
std::cout << "t1 and t2 elements: " << '\n';
for (double d : t1) {
  std::cout << d << " ";
}
std::cout << '\n';
for (double d: t2) {
  std::cout << d << " ";
}

std::cout << '\n';
```

```
  t2 = std::move(pic10b::Tree()); // move a default Tree
  std::cout << "t2 size now: " << t2.size() << '\n';

  t2.insert(3.14); // add two numbers
  t2.insert(100);
  t2.insert(100); // this one is redundant

  pic10b::Tree::iterator iter_to_first = t2.begin();

  if (t2.find(3.14) == iter_to_first) { // check if 3.14 in collection and if first
    std::cout << "3.14 first item!" << '\n';
  }

  pic10b::swap(t2, t2); // prove it is available at namespace scope

  std::cin.get();
  std::cin.get();

  return 0;
}
```



```
Elements: 0.028779 0.129337 0.151555 0.28254 0.282937 0.507645 0.579516 0.71865
0.749962 0.960845
Last element is: 0.960845
Count of elements: 10
Enter lower and upper values for removal: 0.15 0.55
t1 and t2 elements:
0.028779 0.129337 0.579516 0.71865 0.749962 0.960845
0.028779 0.129337 0.151555 0.28254 0.282937 0.507645 0.579516 0.71865 0.749962 0
.960845
t2 size now: 0
3.14 first item!
```

Some initial guidance to get you on your way...

1. Be sure to draw diagrams!
2. Get the **insert** and **size** functions working first; then worry about traversing the tree with iterators. After that, you can worry about other features like **erase** and copy constructors, destructors, etc.
3. The **insertNode** function for **node** can return a **bool**, depending on whether the node gets inserted (recall duplicates are not allowed). Depending on that value, the

size counter can go up or stay the same. Be sure that the **insertNode** function returns something regardless of whether a node was actually inserted...

4. The destruction process, like the copy process, can be done recursively: from the perspective of a node, destroy all nodes to the left then all nodes to the right and then self-destruct!

5. The **find** function is a member of **Tree** and the tree is structured perfectly for binary search. Just start at the root and move left/right until the value is found or there are no more nodes to search through.