PIC 10C Section 1 - Homework # 2 (due Friday, April 8, 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 runs on Visual Studio 2022.

TEMPLATED BINARY SEARCH TREE

Alas, it is time to get a little bit into the weeds and review that pesky Binary Search Tree data structure from 10B. Once again the notes on Data Structures may be handy. In this case, we'll do it with templates. In the end, you will submit **bst.h** and "Honesty.txt" as described in the syllabus. Failure to provide the Honesty file will automatically result in a zero.

Recall that a **binary search tree** is an associative data structure that stores its elements in "sorted order". Assuming the data come in randomly enough then if it stores n elements, inserting/looking up/erasing are $O(\log n)$ costs. It also requires special **node** and **iterator** classes to function.

For this homework, the **only header files you are allowed to use** are:

- **utility**
- **functional**
- **stdexcept**

Once again this homework is about review. It is necessary to have a solid foundation here because we are going to study self-balancing trees later on and they are much, much more involved. 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. The Timer header file can be found here:
https://github.com/3k1m/cpp/blob/master/timing/Timer.h.

Beware of memory leaks!!! To help check for leaks, you can use this: https://github.com/3k1m/cpp

You must write all three classes. Here are the requirements:

The **bst** 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*** (the **node** class manages the tree values) 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 with $O(n)$ runtime cost;
- have copy and move constructors (copying must result in an identically structured tree);
- have copy and move assignment operators;
- have a member and free-function **swap** functions to swap two **bst**s (and the free function must be visible at the **pic10c** namespace level);
- have an **insert** member function with $O(\log n)$ time cost, suitably implemented 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 with $O(\log n)$ time cost;
- have an **erase** member function with $O(\log n)$ time cost 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 **bst**; and
- have a **find** member function running in $O(\log n)$ time cost returning the **iterator** to the node with a given value if found and otherwise returning the past-the-end iterator.

The **node** class needs to be **nested inside of bst** and have:

- **value**, of type **T**, being the value it stores and

- **left**, **right**, and **parent** of type **node*** to track where its left/right/parent nodes are.

You can determine the rest of the functions/implementations of **node**.

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 (without allowing modifications to the tree elements); and
- overload **operator arrow**.

A test case, code and output, are provided below. Note the runtimes: The time for N insertions should be $O(N \log N)$, hence the roughly 100-ish factor in time between the small and big insertions. The time for destrucing N values should be $O(N)$. The traversal times for the two trees are more or less the same because the copied object should have the identical tree structure.

```
#include "bst.h"
#include "Timer.h"

#include<iostream>
#include<string>
#include<algorithm>
#include<vector>
#include<utility>

// bah class can be made many different ways -- test for emplace
struct bah {
  int i, j, k;
  constexpr bah(int _i = 0, int _j = 0, int _k = 0) : i(_i), j(_j), k(_k) {}
};

constexpr bool operator<(const bah& b1, const bah& b2) {
  return std::tie(b1.i, b1.j, b1.k) < std::tie(b2.i, b2.j, b2.k);
}

std::ostream& operator<<(std::ostream& out, const bah& b) {
  return out << '[' << b.i << ' ' << b.j << ' ' << b.k << ']';
}
```

```
// compares reversed strings
bool rev_str(std::string s1, std::string s2) {
  std::reverse(s1.begin(), s1.end());
  std::reverse(s2.begin(), s2.end());
  return s1 < s2;
}

int main() {

  pic10c::bst<int> a;
  {
    pic10c::bst<int> b;
    b.insert(1);
    b.insert(3);
    a = std::move(b);
  }
  a.insert(2);
  for (auto i : a) { std::cout << i << '\n'; }


  // standard case
  pic10c::bst<int> b1;
  b1.insert(3);
  b1.insert(4);
  b1.insert(0);

  // print the values: requires proper iterators for range-for
  std::cout << "b1:\n";
  for (const auto& i : b1) {
    std::cout << i << '\n';
  }

  pic10c::bst<std::string, bool (*)(std::string, std::string)> b2(rev_str);

  b2.emplace(3, 'a');
  b2.insert("aaa"); // duplicate!
  b2.insert("zzzzzzzza");
  b2.insert("aaaaaaaaz");
  b2.emplace(6, 'm');

  auto b3 = b2;

  std::cout << "b3:\n";
  for (const auto& s : b3) { // print b3, copy of b2
```

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

auto b4 = std::move(b3); // move construct b4 from b3...
std::cout << "size b3 " << b3.size() << '\n';
std::cout << "size b4 " << b4.size() << '\n';

std::cout << "iterator stuff:\n";
pic10c::bst<std::string, decltype(&rev_str)>::iterator it =
  b4.begin();
std::cout << *it << '\n';
std::cout << it->size() << '\n';
b4.erase(it);

std::cout << "b4:\n";
for (const auto& s : b4) { // no more "aaa"
  std::cout << s << '\n';
}

// test emplace here
pic10c::bst<bah> bahs;
bahs.emplace();
bahs.emplace(1);
bahs.emplace(-1, 7);
bahs.emplace(0, 9, 146);

for (const bah& b : bahs) {
  std::cout << b << '\n';
}

constexpr size_t small_n = 50;
constexpr size_t big_n = 5000;

std::vector<double> vals_to_insert;
vals_to_insert.reserve(big_n);
for (size_t i = 0; i < big_n; ++i) {
  vals_to_insert.push_back(std::rand() / (RAND_MAX + 1.));
}

simple_timer::timer<'u'> clock;

{
  pic10c::bst<double> timed_tree;
```

```
  clock.tick();

  for (size_t i = 0; i < small_n; ++i) {
    timed_tree.insert(vals_to_insert[i]);
  }
  auto small_insert = clock.tock();
  std::cout << "small insert: " << small_insert << '\n';
  clock.tick();
}
auto small_destruct = clock.tock();
std::cout << "small destruct: " << small_destruct << '\n';

{
  pic10c::bst<double> timed_tree;

  clock.tick();

  for (size_t i = 0; i < big_n; ++i) {
    timed_tree.insert(vals_to_insert[i]);
  }
  auto big_insert = clock.tock();
  std::cout << "big insert: " << big_insert << '\n';
  clock.tick();
}
auto big_destruct = clock.tock();
std::cout << "big destruct: " << big_destruct << '\n';

// now test copying preserves structure
pic10c::bst<double> rands;
for (size_t i = 0; i < big_n; ++i) {
  rands.insert(vals_to_insert[i]);
}
auto rands2 = rands;

double tot = 0, tot2 = 0;
clock.tick();
for (double d : rands) {
  tot += d;
}
auto rands_time = clock.tock();
clock.tick();
for (double d : rands2) {
  tot2 += d;
}
```

```
  auto rands2_time = clock.tock();

  std::cout << "times are " << rands_time << ' ' << rands2_time << '\n';

  return 0;
}
```

A few hints...

1. If you are still not crazy about templates, write a binary search tree class for ints or some other concrete type first. Then you can do a few replacements of int to T, $<$ to a call to pred, etc.
2. Use the Visual Studio breakpoints as part of the debugger as you test things out if they go wrong.
3. Get the insert function working first.
4. Write the begin/end functions next.
5. Then work on traversal (which won't work unless insert is wired up correctly and begin/end are correct).
6. Then you can worry about the other functions.

```
1
2
3
b1:
0
3
4
b3:
aaa
zzzzzzzza
mmmmmm
aaaaaaaaz
size b3 0
size b4 4
iterator stuff:
aaa
3
b4:
zzzzzzzza
mmmmmm
aaaaaaaaz
[-1 7 0]
[0 0 0]
[0 9 146]
[1 0 0]
small insert: 28.2us
small destruct: 8.1us
big insert: 3898.4us
big destruct: 684.3us
times are 390.2us 330.5us
```