Project 5 – mymap

CS 251, Fall 2022

Collaboration Policy: By submitting this assignment, you are acknowledging you have read the collaboration policy in the syllabus for projects. This project should be done individually. You may not receive any assistance from anyone outside of the CS 251 teaching staff. Also, please see copyright at the bottom of this description. Do not post your work for this class publicly, even after the course is over.

Shared Tests

You may collaborate with your peers on tests, if you use tests from a peer ***you should properly cite your sources within your tests.cpp file for the individual or group of tests***. Using a common framework such as Google Tests or Catch comes in handy when sharing tests.
As with other projects, you still CANNOT collaborate on the implementation code.

What to Submit to Gradescope: (1) mymap.h; (2) tests.cpp.

Starter Code

Project Background

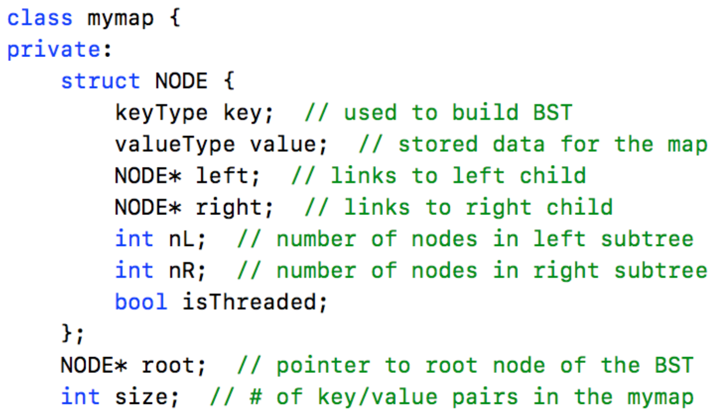
Maps are one of the most useful abstractions. You are already an expert at using them, now it is time to write your own! For this project, you will be implementing a general purpose, templated map class. It will have a similar functionality to the C++ standard library map container. However, we have specific implementation details that you must adhere to. Enjoy!

mymap – Implementation Details

"Under the hood" of the MyMap abstraction could be a tree, an array with hashing, etc. You are going to implement the map with an underlying representation of a self-balancing threaded binary search tree. This is going to be a challenging project. It will expose any bad habits you have developed with debugging and testing, as well as force you to hone skills that you might have been avoiding (memory mapping, diagramming, gdb, segmentation faults, etc.). There will be many edge cases and considerations that you will need to design into the implementation. It will also test your project management skills. You should do **more** thinking, drawing, and designing and **less** coding and hacking away at functions. Please follow our advice on how to develop the code step by step. Thoroughly test your code before moving on.

WARNING: Implementing the MyMap class using a BST is fairly straightforward. In fact, most of the code is already provided in the BST lectures. Implementing with a threaded BST is a little more complicated, but with the correct decomposition and design, it is actually not too much work. Implementing our *self-balancing*, threaded BST will be where you are most challenged in this assignment (particularly with a few of the member functions). Grading is set up so that if you can get the MyMap implemented without any balancing, you will still receive a decent amount of credit. This is why it is important to work incrementally and test often. Develop lots of tests for BSTs that are balanced (no balancing is required at any insertion into the tree). Once those are working, build lots of tests for BSTs that need balancing.

The data structure you are going to use is a threaded, self-balancing binary search tree. The node structure looks like this:

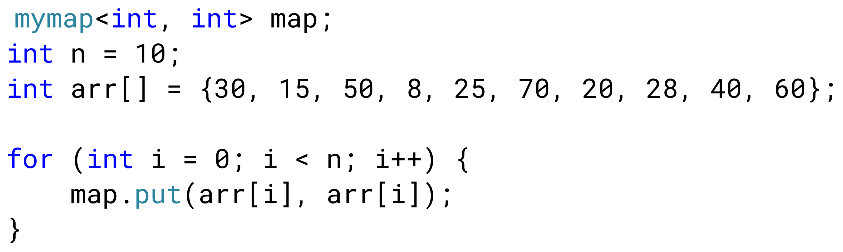


```
class mymap {
private:
    struct NODE {
        keyType key;  // used to build BST
        valueType value;  // stored data for the map
        NODE* left;  // links to left child
        NODE* right;  // links to right child
        int nL;  // number of nodes in left subtree
        int nR;  // number of nodes in right subtree
        bool isThreaded;
    };
    NODE* root;  // pointer to root node of the BST
    int size;  // # of key/value pairs in the mymap
```

Therefore, your nodes keep track of more than a traditional BST node. First, keep in mind that you will build a BST based on the key (which is templated). However, the node will also store the value associated with that key (where the value is also templated). This means the key and value can each, independently, store integers, doubles, strings, etc.

Milestone #1 – Implement the base functionality of mymap using a standard BST
To start, you should implement the member functions in the table below (in the order listed below) using a standard BST. Much of the code for this is provided in lecture. To do this, you will only need to use part of the NODE struct: key, value, left, and right. Leave the remaining members of the NODE struct as default values. You will use them later.

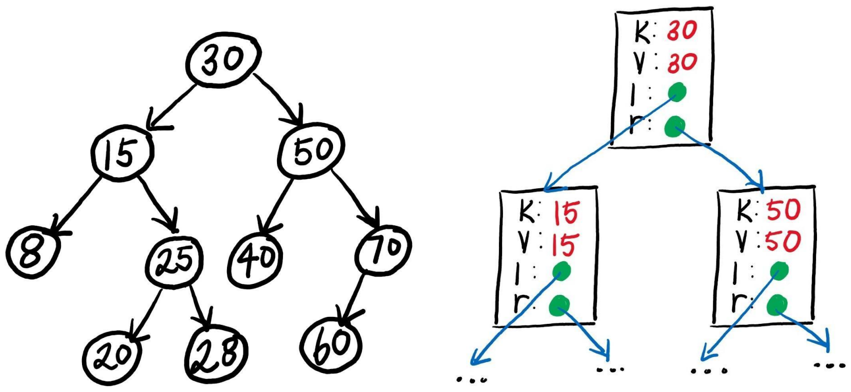
Here is a example *mymap* that you can get started with:



```
 mymap<int, int> map;
int n = 10;
int arr[] = {30, 15, 50, 8, 25, 70, 20, 28, 40, 60};

for (int i = 0; i < n; i++) {
    map.put(arr[i], arr[i]);
}
```

This is a mymap of integers keys and integer values. Here is a visual of how your BST will lay out the memory of this example mymap:



The left image above is the full BST with only the keys shown. The right image above is a more accurate example of what the NODEs look like in memory (ignoring the member variables we aren’t using yet). This is a great mymap to start with because the insertion order was set up such that the tree is always balanced.

As you write each function, ***you will need to test the function extensively***. Your first step to test these functions is to build a mymap to test the functions on. We recommend that you come up with a few mymaps where the underlying BST remains balanced (definition of “balanced” is provided in Milestone #5 using the Seesaw-Balanced property, for now, just stick with trees that are perfectly balanced and you’ll be fine). Remember that the class is templated, so you will need to test key/value pairs that are various types…int, double, string, char, etc. While it is convenient to start with mymaps that have the same key and value, don’t forget to test mymaps where the key and value are not the same.

The tables below gives you a starting point, but your tests will need to be significantly more extensive. In order to implement mymap correctly, you will need 100s…1000s….10000000s of assertions.

In summary, for Milestone #1:
Write these functions, in this order, using a standard BST (no threading, no balancing):

	Member Functions	**How to Call/Test**	**Details**
1	default constructor	`mymap<int, int> map;`	Creates an empty mymap.

2	put	`int arr[] = {2, 1, 3};`
		`for (int i = 0; i < 3; i++) {`
		`  map.put(arr[i], arr[i]);`
		`}`
		`EXPECT_EQ(map.Size(), 3);`
			Inserts the key/value pair into the mymap. If the key already exists in the mymap, simply update the value.

3	size	`See above.`	Returns the number key/value pairs in the mymap.

4	contains	`for (int i = 0; i < 3; i++) {`
		` EXPECT_TRUE(map.contains(arr[i]);`
		`}`
			Returns true if key is in mymap, returns false if not.

5	get	`for (int i = 0; i < 3; i++) {`
		` EXPECT_EQ(map.get(arr[i]), arr[i]);`
		` EXPECT_EQ(map.Size(), 3);`
		`}`
			Returns the value for the given key; if the key is not found, the default value is returned (but not added to mymap).

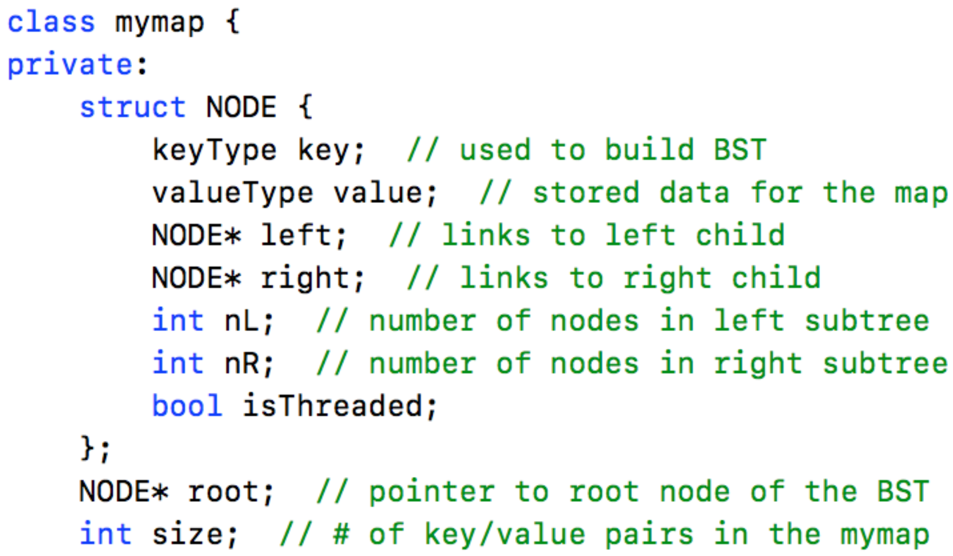
6	toString	`string sol = "key: 1 value: 1\nkey: 2 value: 2\nkey: 3 value: 3\n”;`
		
		`EXPECT_EQ(sol, map.toString());`
			Returns a string of the entire map, in order.

7	operator[]	`for (int i = 0; i < 3; i++) {`
		` EXPECT_EQ(map[arr[i]], arr[i]);`
		` EXPECT_EQ(map.Size(), 3);`
		`}`
			Returns the value for the given key; if the key is not found, the default value is returned (and the resulting new key/value pair is inserted into the map).

Milestone #2 – Add threading to your BST
Starting this milestone, you should have a functional mymap class with a default constructor and put, size, contains, get, toString, and operator[]. You should also have developed a large test suite with at least a couple of mymaps that require no balancing. Now, you are going to add threading to your BST.

A threaded BST takes advantage of the observation that half the pointers in a tree are nullptr; that's a lot of wasted space. Instead, we re-use the **right** pointers as follows: if that pointer is nullptr, we re-use it as a **thread** to the next inorder key; see the dashed links below. Threads make it possible to traverse a tree in key order without recursion or a stack. There is some record keeping needed.

A boolean "**isThreaded**" field is added to every node:



```
class mymap {
private:
    struct NODE {
        keyType key;  // used to build BST
        valueType value;  // stored data for the map
        NODE* left;  // links to left child
        NODE* right;  // links to right child
        int nL;  // number of nodes in left subtree
        int nR;  // number of nodes in right subtree
        bool isThreaded;
    };
    NODE* root;  // pointer to root node of the BST
    int size;  // # of key/value pairs in the mymap
```

The "isThreaded" field will be set to true when the right pointer is being re-used as a thread, and false otherwise.

One of the side-effects of a threaded tree is that traversing the tree is now different. For example, during search, in a non-threaded tree we traverse left or right in the typical manner:

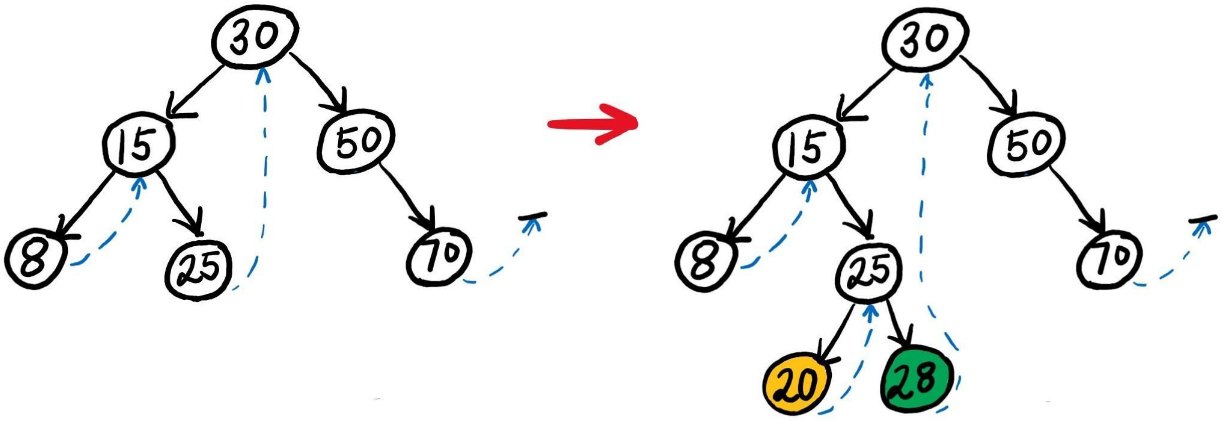
```
if (key < cur->key) {
  cur = cur->left;
} else {
  cur = cur->right;
}
```

This no longer works, because the right pointer could be threaded. In a threaded tree, traversal is now performed as follows:

```
if (key < cur->key) {
  cur = cur->left;
} else {
   // there is no right pointer in traditional traversal:
  if (cur->isThreaded) {
    cur = nullptr;
  } else {
    cur = cur->right;
  }
}
```

When traversing a tree in a normal top-down fashion, a threaded pointer is equivalent to nullptr.

How are threads added to the tree? Threads are added during insertion. The algorithm is simple: suppose a new node N is being inserted. If N is added to the left of its parent P, then N's thread points to P. If N is added to the right of its parent P, then N inherits the thread of its parent. For example, consider the example (now threaded) BST presented earlier in Milestone #1. Let's look at when 20, and 28 are inserted. See below, on the left, you can see the threaded BST after 70 was inserted. Focus on node 25, who's thread denotes 30:



When 20 is inserted to the left of 25, then 20's thread is set to 25 --- the next inorder key. When 28 is inserted to the right of 25, then 25's right pointer is no longer threaded --- it now denotes 28. Instead, 28 inherits 25's thread, so that 28's thread now denotes 30 --- the next inorder key. The updated tree is shown above on the right.

What will use the threads for? We will do that on the next milestone!

In summary, for Milestone #2:
Add threading to your BST implemented during Milestone #1. Go through all the functions you wrote and consider what changes will need to be made to thread your BST. As described above, anytime you traverse your tree OR anytime you insert a NODE in your tree, you will need to consider threading. Think carefully about function decomposition and code re-use, it could help you dramatically during development.

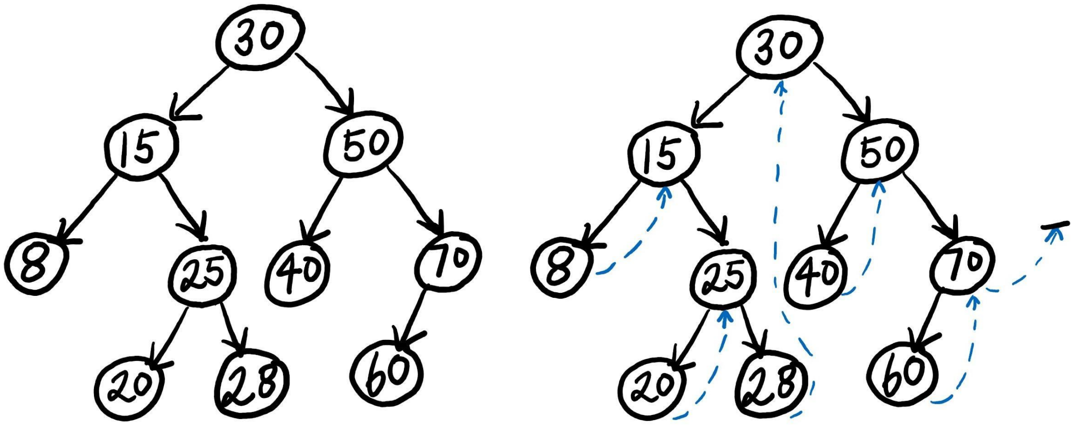
Milestone #3 – add for-each loop functionality

The **map** abstraction in C++ offers the ability to iterate through the (key, value) pairs in key order. For example:

```
for (auto& pair : map) {
   output << pair.first << ": " << pair.second << endl;
}
```

Our map is implemented using a BST - where recursion is typically used - how is this done?

This is where the idea of a threaded **binary search tree** makes things easier. A threaded BST takes advantage of the observation that half the pointers in a tree are nullptr; that's a lot of wasted space. Instead, we re-use the **right** pointers as follows: if that pointer is nullptr, we re-use it as a **thread** to the next inorder key; see the dashed links below. *Threads make it possible to traverse a tree in key order without recursion or a stack.*



The threads are used when the tree is traversed in key order. In our case, 3 functions will be added to the mymap class to enable this traversal: **begin** and **end** and **iterator's operator++**.

Now that your threaded BST is fully threaded, in this milestone, we now get to use the threads to implement these functions:

	Member Functions	How to Call/Test	Details

8	iterator – operator++	`mymap<int, int> map;`
		`int arr[] = {2, 1, 3};`
		`int order[] = {1, 2, 3};`
		`for (int i = 0; i < 3; i++) {`
		`   map.put(arr[i], arr[i]);`
		` }`
		`int i = 0;`
		`for (auto key : map) {`
		`  EXPECT_EQ(order[i++], key);`
		`}`
			This function should advance the pointer curr to point to the "next" in-order NODE.

9	begin		This function should return an iterator to the first in-order node (so the NODE with the smallest key).
10	end		This function is provided, there is nothing you need to do here.

NOTE: A note about our design of the foreach loop. You will notice, it does not work the same as the standard C++ libarary's map's foreach loop. In our case, our foreach loop loops through the keys. In C++'s map, the foreach loop loops through a <key, value> pair. We could mimic that syntax, but chose this design as a slightly cleaner syntax (similar to other higher level languages). You will notice in the operator*() provided in the starter code does not return a keyType& (reference to the key). This is on purpose, because we do not want the user to be given access to change keys while using a foreach loop (that could potentially destroy BST ordering). **Therefore, you cannot write your foreach loops like this:**

```
for (auto& key : map) {
  EXPECT_EQ(order[i++], key);
}
```

Lastly, this design of a foreach loop is not the most efficient. Think on your own….why? By choosing to not allow the reference in the operator*() and by choosing to only give access to the key (instead of giving access to the key, value pair), what does this cost us? How does this affect the space and time complexity of our foreach loop? While we sometimes feel the pain of C++'s verbose syntax, writing your implementation of a map can provide evidence of why those choices are being made. C++ values performance and language consistency. Other languages like Java, Python put more value on ease of use and clean syntax. Can you come up with ways to redesign this foreach loop that would improve both the time and space complexity? No need to do that for the submission, but just an interesting thing to think about.

In summary, for Milestone #3:
Implement begin() and the operator++ for the iterator using your BST's threads. Then, test thoroughly using for-each loops.

Milestone #4 – finish remaining member functions using threaded BST

There are only four remaining member functions to implement in our class. These are all the functions that do a lot of memory gymnastics. At this point, we are going to implement using our threaded BST. Please follow the order in the table. Once you have fully tested these functions' functionality, you will need to run valgrind and make sure your implementation is free of memory errors and memory leaks before moving on.

	Member Functions	How to Call/Test	Details

11	clear	`map.clear();`
		`EXPECT_EQ(map.Size(), 0);`
			Frees all memory associate with mymap. What kind of tree traversal should be used for this function?

12	operator=	`mymap<int, int> mapCopy;`
		`mapCopy = map;`
			Clears “this” mymap and then makes a copy fo the “other” mymap. What kind of tree traversal should be used for the copy?

13	copy constructor	`mymap<int, int> mapCopy = map;`	Constructs a new mymap which is a copy of the “other” mymap. What kind of tree traversal should be used for the copy?
14	destructor	This is called automatically anytime a mymap object goes out of scope. Test clear() and use valgrind to test.	You should test this in this milestone and get it fully working. However, when you move on to the next milestone, we recommend commenting out the destructor while developing and adding it back in at the end. Sometimes having a destructor can make errors hard to interpret while in development.

In summary, for Milestone #4:
Implement clear, operator=, copy constructor, and destructor. Test and make sure the valgrind report is clean (no memory errors or leaks).

Milestone #5 – threaded BST   self-balancing, threaded BST

As you have learned, a simple BST does not guarantee logarithmic height. Operations like lookup, insertion, and removal are linear time in the worst case (for example, if you build the BST using nodes in sorted order). In order to have logarithmic heigh lookup, insertion, and removal, we must keep the BST reasonably balanced. In lecture we learned about AVL trees, Red-Black trees, etc., which are all great balancing approaches. We will be balancing our BST, but not using any of those strategies.

We will be using the following Seesaw-Balanced property to determine if our threaded BST is balanced (see below). The time complexity for these trees is not quite as strong as AVL or Red-Black trees, however, they do an amortized logarithmic runtime for insertion and removal (which just basically means it is logarithmic on average over some number of insert or remove operations rather than guaranteed for each insert or remove operations).

Seesaw-Balance property of a node:

DEF: Given a node **n** in a threaded BST with **nL** nodes in its left subtree and **nR** nodes in its right subtree. Node **n** is SeeSaw-Balanced if the following condition is true:

$$max(nL, nR) \leq 2 \times min(nL, nR) + 1$$

Seesaw-Balance property for a threaded, BST:

DEF: The threaded, BST is Seesaw-Balanced if all nodes **n** in the threaded, BST are Seesaw-Balanced.

Your implementation must ensure that the threaded BST is always Seesaw-Balanced. The only way a violation of the property can happen is if you are changing the tree in some way. For your mymap, that only happens when we insert something into the tree (put, operator[], etc.). (NOTE: this would also affect removal from mymap, but we are not including this function in this project).

Algorithm for enforcing the Seesaw-Balanced Property during insertion:

1. When inserting a NODE into the threaded, BST, you should check that the Seesaw-Balanced property is satisfied along the insertion path. If a violation occurs, keep track of the NODE closest to the root that violates the Seesaw-Balanced property. If no violation occurs, then no re-balancing is needed (ignore the rest of this algorithm) and you should insert as before.
2. Populate a temporary array or vector with all the NODE*s in the subtree that needs rebalancing (this should be in key order). You should not make copies of the NODES, this is just a rewiring process. Therefore, this part of the algorithm should have O(1) space complexity. Also, this part of the algorithm should have O(m) time complexity, where m is the number of NODEs in the subtree. You should *not* be sorting this array as the data is already sorted in the subtree. .

3. From this array or vector, re-construct a perfectly balanced tree that can replace the original subtree. To this, let's call **left** the first index of the array, **right** the last index of the array, and **mid** the middle index of the array (**mid** = (**left** + **right**) / 2). You will insert the NODE located at **mid** first, then recurse on the array from **left** to (**mid** – 1) AND recurse on the array (**mid**+1) to **right**. Recursion continues in this manner until all NODES have been inserted. This part of the algorithm should have O(mlogm) time complexity (or you can even do O(m), if you'd like). See section on time complexity below.
4. Re-connect your new perfectly balanced subtree back to the original threaded-BST. Re-balancing should not have changed the total number of NODES in the subtree.

We have two public member functions we will test against that will help you develop your threaded, self-balancing BST. You should implement these functions first, they will help you develop and test the code needed to implement the algorithm above. These public member functions are not needed for the mymap functionality, but are required and will be tested.

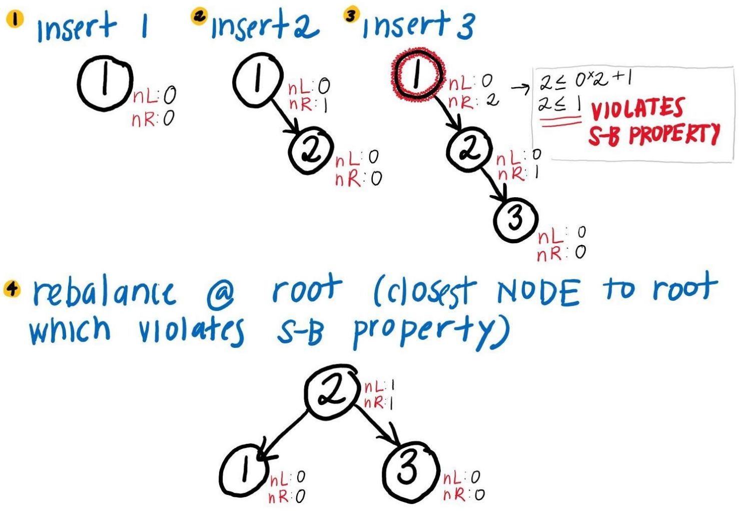
	Member Functions	**How to Call/Test**	**Details**

15	toVector	`vector<pair<int, int> > solution;`
		`…fill the solution vector`
		`EXPECT_EQ(solution, map.toVector());`
			This function will return a vector of key/value pairs of the entire map, in-order. You will notice in the algorithm about you need a similar function that will return a vector of NODE*s. Getting this function working first and testing it, can give you more confidence in implementing a function that gets a vector of NODE*s.

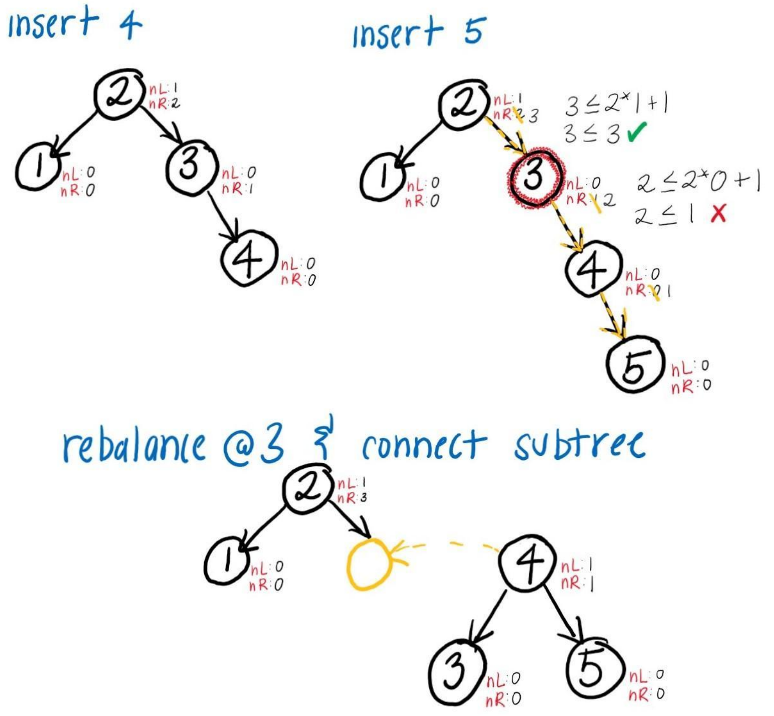
16	checkBalance	`mymap<int, int> map;`
		`int arr[] = {2, 1, 3};`
		`for (int i = 0; i < 3; i++) {`
		`   map.put(arr[i], arr[i]);`
		` }`
		`string sol = "key: 2, nL: 1, nR: 1\nkey: 1, nL: 0, nR: 0\nkey: 3, nL: 0, nR: 0\n";`
		`EXPECT_EQ(map.checkBalance(), sol);`
			Returns a string of mymap so that we can verify the tree is properly balanced. The returned string prints the key, nL, nR in pre-order.

Let’s go through a couple of examples visually. Set aside a whole notebook for diagramming. You will do a lot in order to think through this. It is a beautiful art so enjoy it. ☺

Here is an example where I am inserting the keys 1, 2, 3 in that order. From the diagram, you can see that there are no violations for inserting 1 nor insert 2. However, when we insert 3, we get a violation at the root node (which is NODE closest to the root that violates the balancing property). Finally, you can see that we rebalance the tree at the root and the algorithm will produce a perfectly balanced tree.



If we continue on and add 4 and 5, let’s see what happens:



Inserting 4 requires no re-balancing. However, when we insert 5, we can see that we check the Seesaw-Balance property on the insertion path (indicated in yellow dashes) to where 5 gets inserted. While we are at it, we will also update nL and nR along the way. We see that the NODE at the root (2) satisfies the Seasaw-Balanced property. However, once we arrive at 3, we update nL and nR and determine that the Seesaw-Balanced property is broken here. We “mark” this NODE somehow and continue on the insertion path until we get 5 in the right place. Should we keep updating nL and nR, as well as checking if the property is satisfied? You should think about that and determine for yourself.

Once we are done inserting, we now need to rebalance the substree starting at 3. Following the algorithm, we rearrange (rewire, no copies) the subtree such that it is perfectly balanced. And then, we will connect it back to the main BST (see yellow to indicate where it goes). Lots of details to manage here. What about the threading? What parent NODEs do I need to keep track of? What parent NODEs need to be updated? Can I reuse code I already wrote? Do I need to update nL and nR? Ahh!! It is a lot. Go through by hand with quite a few examples so you get the hang of all the details.

Time complexity of inserting into our threaded, self-balancing BST: The starter code indicates the time complexity requirement for each public member function. For put (the function affected by the balancing algorithm), you must write the algorithm in at least O(logn + mlogm), where n is the total number of nodes in the tree and m is the number of nodes in the sub-tree that needs to be re-balanced. There are two ways of building a balanced subtree at the violating node. (1) Reconnect the NODEs using a normal BST insert operation, by searching for the spot to connect each NODE as you walk the height of the tree. This will be O(mlogm). (2) Since you are adding the nodes in perfect order, on each recursive call, you actually know exactly where to connect the NODE. Therefore, the search down the height of the tree is not necessary. If you pass in a parent pointer of where to connect each NODE to, you can write this part of the algorithm O(m). This would make it O(nlogn + m) time complexity.

Important Notes (please read carefully):

- This balancing portion is the most difficult part of the assignment. Grading will give you a good amount of credit for having a working threaded BST implementation. It is better to submit this project with a working threaded BST implementation than submitting a broken or incorrect threaded, self-balancing BST.
- ***Submissions that use AVL trees (or any other balancing strategy) will get an automatic zero on the project.***
- You must relabance at the violating NODE that is closest to the root.
- Large efficiency penalties will be given for doing any of the following:
  - rebalancing at the root anytime there is a violation;
  - rebalancing multiple NODEs during insertion instead of only the NODE closest to the root
  - inserting NODE normally and then walking the entire tree to look for a violation
  - inserting NODE normally and then rebalance at the root each time, regardless if it is needed anywhere or at the root

In summary, for Milestone #5:
You should add the self-balancing algorithm described to your threaded BST. All your previous tests should pass, but you should write additional tests for mymaps that need rebalancing. You should use checkBalance to ensure your balancing algorithm is implemented per the specifications.

Testing

The majority of this project will be spent developing scalable tests, using the Google Tests Framework or the Catch framework. Check out the Google Tests documentation and feel free to experiment with more than just EXPECT_EQ/ASSERT_EQ. It is really useful to use other containers to test the mymap (vector, set, map…). Here is an example of how you can stress test your mymap:

```
mymap<int, int> mapMine;
map<int, int> mapSol;
int n = 1000000;
for (int i = 1; i <= n; i++) {
  int key = randomInteger(0, 10000);
  int val = randomInteger(0, 10000);
  mapMine.put(key, val);
  mapSol[key] = val;
}

EXPECT_EQ(mapMine.Size(), mapSol.size());

stringstream solution("");
for (auto e : mapSol) {
  solution << "key: " << e.first << " value: " << e.second << endl;
  EXPECT_TRUE(mapMine.contains(e.first));
  EXPECT_EQ(mapMine.get(e.first), e.second);
}

EXPECT_EQ(mapMine.toString(), solution.str());
```

This is only a test you can do once you are completely done with the implementation. You will need significant testing along the way to test each member function carefully. Your implementation must be functional, but also must be implemented exactly as we have spec'd out in this handout. We will manually review your code to make sure the implementation satisfies the requirements.

This is just to get you started about writing smart tests. There is so much more you should do. All the details of the mymap implementation are provided in the handout and in the comments of the starter code.

Requirements

1. You may not change the API provided. You must keep all private and public member functions and variables as they are given to you. If you make changes to these, your code will not pass the test cases. You may not add any private member variables, public member variables, or public member functions. But you may add private member helper functions (and you should).

2. You must adhere to the time and space complexity requirements indicated in the header comment of each member function. Space complexity requirements are omitted if they are obviously O(1) or just obvious (e.g. copy constructor). Your functions should be as efficient as possible, generally O(logn) or better for time complexity and O(1) for space complexity.
3. You must implement mymap exactly as described with a threaded BST. If you get the balancing to work, it must be using the property and algorithm described here (exactly). Details of examples what is not allowed in terms of balancing are provided in Milestones #5.
4. You must have a clean valgrind report when your code is run. Check out the makefile to run valgrind on your code. All memory allocated (call new) must be freed (call delete). The test cases run valgrind on your code.
5. Your tests.cpp should have 100s of assertions for each public member function. You need to put in significant effort into testing.
6. No global or static variables.
7. You can and should use lots of containers in your tests.cpp. You may also use other useful libraries like those that allow for random generations or file reading. For example, it is really useful to test your mymap with the standard C++ library map.

Citations/Resources
Project originally crafted by Shanon Reckinger, University of Illinois Chicago.