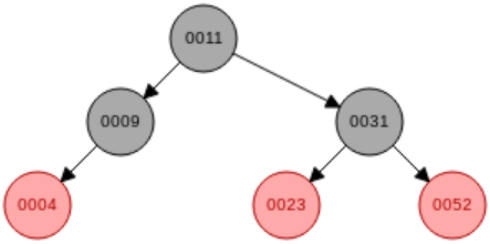
Assignment 6

For the final assignment in CS3 you will implement a red black tree. This is a big under-taking. The red black tree is like, a really really big and really complicated thing. Like a nuclear submarine, or another big thing. But, it's something that *you are able to do* and then, after you're done, you'll feel like "wow!" "I did that!" You'll try to tell other people in your life, but they maybe won't be able to appreciate the gravity of the situation. You'll say, "I implemented a fully functional red black tree with perfect memory management!" and they might say, "what the hell is a red black tree?" Or, "who are you?"



For the first part of the assignment you will need to implement most of the red black tree functions, but they comprise only about ½ of the total work to be done. `Remove()` is the most complex function to implement and you will **not** do that in part 1. Here is a list of what you will implement in part 1.

- A regular constructor
- A copy constructor (maybe do this last)
- An `Insert()` function that takes an integer and returns `void`
- A `Contains()` function that takes an integer and returns a boolean (true if the integer is somewhere in the tree)
- `GetMin()` which returns the minimum item in the tree (just the integer value, not location)
- `GetMax()` which returns the maximum item in the tree (just the integer value, not location)
- `Size()` which returns the size of the tree (the number of nodes).

There are three public `ToString()` variants. For each of these public methods there is a respective private `To<In/Pre/Post>String(root)` method. You need to put these in the public section and implement the private versions.

- `string ToInfixString() const {return ToInfixString(root);};`
- `string ToPrefixString() const { return ToPrefixString(root);};`
- `string ToPostfixString() const { return ToPostfixString(root);};`

In the private section you should also have two instance variables

- `unsigned long long int numItems = 0;`
- `RBTNode *root = nullptr;`

After following these instructions up until this point you probably have an incomplete header file. A lot of the design details in this assignment are left up to you. You can add arbitrary things to your header file and make design choices as you see fit. But, please don't modify any of the things I've specified here. Also, your code will need to pass the tests provided. These tests are minimal and, as always, you should add your own!

The Next Steps

1. Create the header file and `Makefile` as described above.
2. Write the `RBTNode struct` (*see video lecture for details*)
3. Create the `RedBlackTree.cpp` file and write the constructor.
4. In your cpp file write the `ToInfixString()` method.
5. Comment out all the tests in the test file except for the first one: `TestSimpleConstructor()`
6. Comment out the necessary things in the header file.
7. Compile and run the program and pass the first test.
8. Consider running the program with `valgrind` to guarantee no memory issues (leaks, invalid reads/writes, etc.) so far.

There is a lot more to do now. As you will see in the video lecture, `Insert()` is complicated! Plan the work out however you want.

You're done when your program passes all of the provided tests. Again, consider other tests to add. I will grade with many more tests than I provided to you! The rest of this document is a reference of details, edge-cases, and interesting tid-bits that you may or may not find useful. Please consider checking the reference section below if you get stuck / confused. Don't forget to include a `Makefile` in your submission!

Recommendations for what to do when you're stuck, tired, frustrated, confused, etc.:

- Slam your firsts on the table and yell "ugggh!!? What the fuck!?!"
- Try it again without changing anything and hope that it works for no reason.
- Try valgrind on it. I recommend compiling with `-g` and `-Wall` flags and then running valgrind with the `--leak-check=full` flag.
- Consider life as a "normal" person that doesn't know anything about red black trees and is happy about it.
- Take a walk.
- Draw pictures on a piece of paper to diagram what is happening vs what should be happening.
- Eat a snack.
- Do something mundane and physical, but easy (e.g., wash the dishes, do the laundry, etc).
- Put in a million print statements to have the program explain what the heck it is doing.
- Use `gdb` on it (again, compile with `-g`)
- Try the simulator: https://www.cs.usfca.edu/~galles/visualization/RedBlack.html
- Complain quietly to yourself or loudly to others about the situation.
- Consider the long term gains of spending time and effort improving your computer science and programming skills. Knowledge is *compounding*. What you are learning now may seem pedantic or pointless, but it is not.

Reference

colors

- You should represent the colors with integers.
- You should use `#define COLOR_RED 0` directives at the top of the header file. Then in your source code you can do `RBTNode->color = COLOR_RED;` which is more readable.
- The decision about what number corresponds to what color is arbitrary and up to you.
- `COLOR_DOUBLE_BLACK` will be used in the remove algorithm(s) only.
- You will never see anything that actually appears **red** or **black**. The colors are only represented as integers and sometimes as "R" or "B"

testing and errors

- A useful resource is this red black tree simulator:
  https://www.cs.usfca.edu/~galles/visualization/RedBlack.html
  Note that it may sometimes animate / visualize a different process than what we are used to. But, in general the final state of the tree after any operation (insert, remove) is correct. Your program should match the final output, but not necessarily the process to get there.
- Pro-tip: write a unit test for every method you write.
- Tests may be done for private methods. In `RedBlackTree.cpp` make a public method called `PrivateTests()` which implements a bunch of tests of your various private methods. In `RedBlackTreeTests.cpp` add a test that calls `PrivateTests()`
- Public methods should throw exceptions on invalid input (e.g., inserting a duplicate item). Private methods do not need to throw exceptions or do much of any error checking. Private methods are private, so it is safe to assume that all odd / erroneous conditions are handled internally and fully at the public level. It is generally safe to assume that private methods are not passed invalid inputs since you're the one calling them!
- Another pro-tip: the `ToString()` methods are recursive.
- Tricky stuff: If you have a red black tree and you randomly do a `RotateLeft()` on it at some random point (for example when testing `RotateLeft()`) you will likely have violated the properties of the red black tree and other operations may not work correctly anymore on that same tree.
- Pro-tip: Aim for 100% test coverage. That is, write enough tests so that *every* line of code in your `RedBlackTree.cpp` file is executed by your tests at least once.
- Many questions about how to implement the methods, or exactly what they should return can be answered by checking the provided minimal tests.

memory stuff

- `nullptr` is the best way to implement pointers that point to "nothing"
  `x->left = nullptr;`
- If you try to do `->` on a variable that contains `nullptr` (nothing) it will cause a segmentation fault without any more information. Many places you should use code such as
  `if(x != nullptr) { x->blah }` to avoid the issue.

- Use `valgrind` to get a line number / more information when you get a segmentation fault. Don't forget to include `-g` in your compile line!
- Keeping empty "null" nodes in the red black tree is useful for leaves that are yet to be filled. An empty / null / leaf node is considered black.
- When you first create the tree and it has no nodes at all, root should be equal to `nullptr`.
- Keep in mind memory allocations. If you have a "`new`" you will need a corresponding "`delete`" at some point in the program. Many memory issues will be fully addressed in the next part of this assignment.
- Memory issues include leaks as well as "invalid reads/writes." If you read or write data that has already been deleted the program may run correctly / as intended or it may crash. The behavior is undefined. You should use valgrind to check:
  ```
  valgrind --leak-check=full ./rbt-tests
  ```

Submission: Your program will be submitted using canvas and git + github.

1) You should create a private `github.com` repository (repo) with your code in it. The name you choose for the repo is arbitrary but I suggest "HW6" as the repo name.

2) Your repository on github.com should be **private** and you should add the account(s) listed below as "collaborators" by going to "settings" → "manage access" → "invite collaborator" and entering the username(s):

```
fmresearchnovak
```

3) On canvas you should upload a link to your github repository

```
https://github.com/<replace with your github username>/HW6.git
```

Note: You can submit multiple times on canvas and/or make multiple new commits any time before the deadline. You can also submit early on canvas, but continue to make changes (new commits) to the code through git/github. I will grade the final commit made before the due-date.