Put the Liquor on Ice
Programming Project 4
30 points

Just before first light, the ritual begins to slow, as it becomes clear that a winner has emerged from the ashes of the undead pirate ghosts dancing in the pale moonlight of the otherworldly congaline of decadent death. As the victor, a pirate named Victor, reaches for the double-edged sword of power, you, Captain Shaq Sorrow, approach him. With a quick flick of the wrist, you unsheath the *dark sabre*, a weaponized `void` pointer that has seen a great many segmentation faults, and liberate Victor's head from his tail. One might say that Victor is now suffering from *listlessness*, i.e., the state or condition of lacking energy, enthusiasm, and a linked list.

Within seconds, the ground begins to shake, and to your horror, you realize that Victor was a load-bearing pirate. The desserted island begins to crumble, but not in the good peanut butter crumble way. Rising from the depths of the island are creatures spoken of only in hushed tones, as if they are merely wives' tales. . . LiquorIce Ice Babies. These abominable yeti-like snowmen made of frozen razor-sharp shards of licorice come in two flavors: blackberry and strawberry. Each carries a terrifying weapon that defies description (a gun) and fires red or black licorice at any fool dumb enough to stand in the way. Each gun also comes equipped with its own pew-pew-pew noises, described here.[1] One could also describe these as Winter Soldiers, but that might give them too much individual credit.

Frankly, you thought they would be vanilla flavored, to stay in theme with bad 90s pop stars, but whatevs.

You yawn, because you've been up all night, and also because this whole business of getting a new protagonist every single project description is really getting a bit overly routine. You draw your other, lighter sabre, the *blood sabre*, and marvel at how the color palette really matches the whole visual of the impending fight scene. . . On the one hand, the LiquorIce Ice Babies in strawberry and blackberry, and you wielding the dark and blood sabre. Almost as if this was pre-ordained to happen. Anyway, you climb into your *Buick Le Sabre*, rev it up, and drive straight into the LIBbies. You're going to carve them up with your sabres, pile up their bodies, and build a strip mall with them.

That will bring *balance* to this island and let you *tree-t* them with respect. It's a weak transition, but it's all we've got.

Job Details. This project allows you to work in groups of 2 or 3, using the same Pair Programming mechanics as in earlier projects. In this project, you will be implementing a *red black tree*, a self-balancing binary search tree that we have learned about in class. You will be modifying source code for the `Tree` class, which is a standard binary search tree (BST). Your task is to augment/modify that code to implement the `RBTree` class, along with any associated `.h` or `.cpp` files, a `makefile` to build your project, a clean `valgrindoutput.txt` file, and a `git` repository that contains well-documented code and commits for the entire process. For your reference, the valgrind command your program needs to come "clean" with is

```
valgrind --tool=memcheck --log-file=valgrindoutput.txt --leak-check=full ./blah
```

[1] `https://www.youtube.com/watch?v=jlLpV2Ge_LY`

where `blah` is the obvious best name of the executable file. Detecting (and removing) memory leaks is a key component of your assessment on this project.

You should probably attempt this project in phases, making sure that your code works correctly at the conclusion of each phase. *I cannot emphasize enough that if your code does not work AND IS NOT MEMORY LEAK FREE after each phase, you should likely not move on.* These phases are designed to keep you from writing a mess of code that does not work and then being unusually sad at some later point in life, when you reconsider your entire life's choices. But the thing is, that's a lot of choices to reconsider. And this project has a deadline.

Phase 1 – Adoption. Cap'n Shaq Sorrow (that's you) has *planned parenthood* in his/her/their future. Your first task is to augment the current code base so that each `TreeNode<NODETYPE>` now also has a `parent` pointer and a `color`. I recommend that you declare the color as an `enum` (short for *enumeration*), which is a user-defined type that specifies constant variable names that substitute in for integer constants. For example, you can create the following enumeration just before your `TreeNode` class in `treenode.h`:

```
enum Color {BLACK = 0, RED};
```

This allows you to create a new variable `Color color` in `TreeNode` that can be either `BLACK` or `RED`. Fancy. Now your code is perhaps now a bit easier to read than it might otherwise be if you were using `0` and `1` to fill in for black and red. Once you have created these two new variables in a `TreeNode`, you need to consider modifying *all* of the functions from that were previously written to account for these new variables and make sure that they are maintained and set appropriately where they need to be set.

In particular, you might want to augment your traversal functions to also print out parent and color information so that when you traverse, you can confirm that all pointers are correct every time you make a change to your tree. *Do not move on until your code is 100% working and 100% memory leak free!*

Phase 2 – Liquor(ice) Up. Create a new class `LicorIce` that will have the information about the licor-ice ice babys that are bearing down upon your current location and trying to retain control of their own dessert island. How dare they! In this project, there is no sample class header that has been created for this object. . . you are entirely on your own and have to create everything from scratch. It's almost as if we have taught you things, and now that you know these things, you can showcase that skill set in this phase and beyond. Amazing.

This class needs to have the following functionality:

- Each licor-ice baby has a *shatter score* from 0 to s that indicates its relative strength. A high score indicates that it is difficult to shatter, whereas a low score indicates that it is a wuss. This score will be read in from a text file, and s can be up to 10000. Each baby will be inserted into the tree based on its shatter score.
- Each baby also has a *weapon*, with a power from 0 to p. Low is bad, high is good. p can range up to 100.
- Each baby will have a function that fires its weapon and will return a random integer from 0 to $p - 1$.
- Each baby has a name, stored as a `string`.
- The s and p values for the babies will be read in from a text file.

Test your tree by using the `LicorIce` class in place of `int` and confirm that it still works. *Do not move on until your code is 100% working and 100% memory leak free!*

Phase 3 – Rotate Yer Partner. Your next task, should you choose to accept it, is to write the functions `leftRotate(x)` and `rightRotate(y)`. You will need to use the pseudocode from class to get an idea of how to implement each of these functions. Make sure to test your code where x and y are all over the tree, with pictures, and make sure it works. If you implement them efficiently, each function should take around 15 lines of code (or so). However, if your code is longer or shorter, don't freak out. It's important that your code is correct and memory-leak free, not that it's necessarily the most efficient implementation of the code. (To be clear, you should aim for efficient code, but if you don't hit that goal, it's fine.)

Do not move on until your code is 100% working and 100% memory leak free! (Have you perhaps sensed a theme?)

Phase 4 – Recolor and Insert. Implement the `recolor(k)` function, and include it at the end of the recursion `insertNodeHelper` as the pseudocode dictates. You need to think carefully about where to call the function to begin with, and then you will have to go one step at a time through each case. If you ever need help simulating what your tree should be doing, you can use this simulator `https://www.cs.usfca.edu/~galles/visualization/RedBlack.html`. Here is how I recommend doing this portion of the project:

1. Choose a particular case that you are going to work on.
2. Build a sample tree that has that specific case and does *not* have a "bubble up" (or recursive) situation.
3. Work on your code until you can get that specific sample case working correctly.
4. Try other sample trees with your specific case until you're confident it works.
5. If this isn't your first case, see if your other cases still work after you coded this case.
6. After *all* cases are coded, try to do a one-level "bubble-up" example to see if it still works.
7. If that works, go nuts and try to do it all out with all the cases. Test your code well here.

Do not move on until your code is 100% working and 100% memory leak free! Surprising.

Phase 5 – Delete (BONUS). Implement the `doubleBlack(x)` function, and include it in the appropriate place in the `deleteNodeHelper` function as the pseudocode dictates. This function is complicated and challenging. It is also worth many bonus points. In particular, 30 bonus points. If you successfully write `delete` correctly, with no memory leaks, you can earn 30 bonus points on this project, which I can clearly say is many bonus points. The strategy to tackle this problem is the same as in the insert case.

Here is how you calculate whether you are allowed to delete a node. Normally, you can just delete a node from a red-black tree. In this case, you can only delete a node if you can defeat its power with your sabres. You have a red power and a black power. The ice baby you are trying to delete gets a power equal to s plus the random value that comes from its weapon. If your power exceeds that total, you are allowed to delete the ice baby. Otherwise, you aren't allowed to delete it and you take 1 damage. If you take 10 damage before you empty the tree, Cap'n Shaq Sorrow must retreat to his Shaq.

One other slight issue with deleting. . . . Your sabres share power and have a total of 11000 to share between them. If you spend black power, you gain an equivalent amount of red power, and vice versa. For example, if you shatter a black ice baby that has a total strength of 4000, you gain 4000 red power and lose 4000 black power. So, the ideal way for you to schedule your

deletes is to iterate between deleting big and small elements as long as those are different colors. However, you will notice that your tree does NOT have a search function. That is because you can't search to find out the color of a value for a node before you delete it.

1 Turn In Materials

You will need to share your GitHub repository for this code. You can submit a single repository for this project. Your repository should contain:

- All `.h` and `.cpp` code for your project, and NO `.o` or executable files.
- A makefile that correctly builds your project.
- A `valgrind` output file that shows the memory leaks that you have squashed.
- Two output files that shows insert working for each of the cases (along with bubble-ups) for at least two different trees.
- If you completed the bonus, two output files that does also shows delete operations for each of its cases on the same trees.
- If you submit output files in a different organization, just include a README so that we can understand how and where we can get the runs we need to see the necessary output.