Project 3: Playleafs

Instructor: Sadab Hafiz

Due: July 14, 2024

Introduction

In the last class, we looked at how a Binary Search Tree (BST) can be implemented using smart pointers to efficiently handle memory leaks. In addition to that, we discussed the efficiency of using a tree data structure compared to a linear data structure in terms of runtime. It is your turn to implement a BST. In this project, you will implement a `Playlist` class which will keep track of songs and artists in a BST structure. Furthermore, you will test your class to verify its functionality.

Submitting through GitHub

Assuming you did project 1, you should have a GitHub account. Furthermore, you should now be comfortable submitting assignments on Gradescope from GitHub. If not, watch this video which shows the entire process, from accepting an assignment to submitting it on Gradescope. **The link to accept the GitHub Classroom assignment can be found on Blackboard. You must submit your assignment from the repository created by GitHub classroom**.

Documentation and Design

Documentation and design is worth 20% of each project like spring semester. The requirements are:

Documentation Requirements (15%)

- All files should start with block comments with at least your name, date, and a brief description of the code implemented in that file (a rough idea of what this document is).
- All functions in the `.hpp` files should have a comment describing the function:

```
/**
 * @brief: describe the function in general and what it does
 *
 * @pre: describes any precondition
 * @param: one for each parameter the function takes
 * @return: describes the return type
 * @post: describes any postconditions
 * @throws: describe any exceptions that the function may throw
 */
```

  For the required functions in each project, these will be provided to you in the instructions. However, you have to write them for any additional helper functions that you add and implement.

- While function comments are descriptive, they are intended to let users of the class know how to use the functions. Thus, they don't go into the details of the implementation. In order to explain the implementation, you should add inline comments where necessary (loops, conditionals, etc.):

```
// comment explaining what is happening in the for-loop
for (DataType x: some_arr_){
    // comment explaining the if-else statements
    if (...x meets some condition...){
        ...some code in here...
    }
    else{
        ...some code in here...
    }
}
```

  If something feels necessary to be explained to someone looking at your code, you should leave an inline comment explaining it.

- Avoid commenting every single line. Things that are self-explanatory should not be commented.

Design Requirements (5%)

- You must follow the naming convention discussed in class.

```
string my_variable;
class MyClass{};
MyClass class_instance;
string my_data_member_;
void myFunction();
const int MY_CONSTANT;
```

- Any decisions you make, make sure to be consistent. This applies to the choice of include guards (`#pragma` or `#ifndef`), the order of things in classes, comments, code formatting, etc.
- Avoid making decision choices that violate OOP principles. For example, `public` data members and global variables are to be avoided unless explicitly mentioned. Things should be `protected` only if the class will have other classes inheriting from it.
- For any helper functions, the keyword `const` and `&` should be used where appropriate. For example, if your function doesn't modify the private data members, the functions should be defined as `const`.
- Do not include `using namespace std;` in the header files. Avoid using it altogether.
- Include `.cpp` files at the end of class template headers. Libraries should be included in the `.hpp` files.
- As long as you follow the standard practices that are demonstrated in class, you will not lose points.

Helpful Tips

If you are using vscode, there are extensions that can help you. The documentation generated in the header files I share use an extension that generate comments compatible with Doxygen. Doxygen is a tool that provides robust support for documenting `C++` code. Here are the extensions that I highly recommend you to use to automate some documentation generation:

- Doxygen Documentation Generator : Generate Doxygen compatible comment templates
- `C/C++` for VSCode : VSCode language support for `C/C++` editing and debugging

If you don't have a working `C/C++` environment, contact me as soon as possible. The starter files show you how you can document your files. Make sure your final submission is documented and formatted properly.

Starter Code Tips

In the provided files, make sure you address each `TODO` comment and delete the comments after addressing them. If you plan to modify a function that is explicitly stated not to be modified, make a copy of the original. You can also look at the original starter files from GitHub commit history.

Testing

Unlike the previous projects, **testing your code is required to get full credit**. You must submit a `main.cpp` file with tests written for each function you implement. Having a function that you don't test will lead to penalties for that function. More on this later in the project instructions.

Plagiarism and Citing

Your code in all the `.cpp` files should not 100% match with another student. If that happens, I will assume it is a violation of academic integrity. It can happen with the use of generative AI, it can also happen if you both copy code from the same source. If you use tools or functions that are not mentioned in the slides, or I didn't teach in class, you have to cite them with an inline comment. You should also be able to explain such lines of code if I ask you to explain what you wrote.

Implementing Playlist Class

Implement the `Playlist` class. The header file is in the starter code, **which you can access through the GitHub classroom link in Blackboard**. Carefully go through each of the function comments to figure out how to implement them. Test one function at a time before implementing the next.

The `.cpp` file is also provided with the implementation for the `remove` function, and it's helper functions. This implementation serves as an example of how you can **use the code we went over in class to implement the other functions**. Take a look at the comments of this function and make sure you understand each line. In addition to that, the recursive implementation for `inorderTraversal` is also provided as an example of how you can implement the other traversal functions.

Instead of using the `BinaryNode` we went over in class, use the `SongNode` struct. Structs are like classes where everything is public by default. Try using the given struct and get familiar with its functionality. You will use this struct pretty much everywhere in the project. **Do not change anything in this struct.**

A `SongNode` contains two string data members `song_` and `artist_`. For the BST, how do we compare two `SongNode` objects? Songs can have the same title but be made by different artists. Therefore, the comparison has to be made considering with the song name and the artist. In order to make the comparison, you will use the provided `getKey()` function. This function adds the song name string with the artist name string to form a combined string. When comparing two `SongNode`, you can compare their combined version returned by this function. This function is provided to you as part of the `Playlist` class.

I highly suggest looking at the implementation I showed in class, which you can find in the course website, and using that as a starting point while implementing these functions. You are free to add as many member functions as required as long as they follow standard `C++` practices, and they are documented. However, **you are not allowed to add any additional data members. Do not modify the function prototypes or data members of the functions in the starter code. Do not modify the function implementations provided in the starter code.**

Compiling and Testing

There are no template classes in this project. Compile using `g++ --std=c++17 Playlist.cpp main.cpp`. **I cannot stress enough that Gradescope is not a testing platform. Just because your code compiles doesn't mean it works properly. You need to write test cases and make sure your functions are doing what the instructions want**.

Unlike the previous projects, **there will be penalties for not testing your code**. Pick at least five of your favorite songs, add them to a `Playlist` in a `main.cpp` file. Write test cases for all the functions you implement using these songs. Leave inline comments in the `main` function clarifying what you are testing in that part of the code. If your code doesn't compile on Gradescope due to one or two functions not working, I can give you credit for the functions that you tested thoroughly. **You will lose points for each function that you don't test. A lack of `main.cpp` file will result in a 0.**

Testing is a very important skill that you need to learn by trial and error. If you get an error, look at the error message and find out which line it is coming from. I recommend implementing each function and testing it before implementing the next one. This ensures that one function's error is not happening due to errors in another function. If you pass this course without knowing how to test your code, I have failed as an instructor. Without this skill, it is highly unlikely that you will pass coding interviews.

Submissions

You will submit your code to Gradescope through GitHub Classroom. Gradescope is expecting the files: `Playlist.hpp`, `Playlist.cpp`, and `main.cpp`. To prevent the abuse of autograder and to encourage testing, **the autograder output will be vague**. Furthermore, **I will release the autograder after the final exam**. This will hopefully make you test your own functions instead of relying on the autograder. Your code needs to compile on Gradescope in order to get any feedback from the autograder. If it doesn't compile, make sure you fulfill all the requirements mentioned in the project specifications. The autograder will not run unless all the above files are provided and all of them contain the required functions exactly how they are defined in the starter code. Once you pass the test cases on Gradescope, make sure to verify the documentation and design requirements before making the final submission.

Tips and Hints

Here are some tips and hints for completing this project. I will add more here if necessary, so keep an eye on this document. Each modification will be logged at the end of this file:

- You can find a BST implementation on the course webpage. I highly encourage you to take a look at that. You can implement most of the functions by modifying that code.
- Do not wait for the last second. I cannot guarantee that I will answer emails after the final exam, as I will be busy grading them as soon as possible to allow regrade requests. There will be no extensions for this project, as it is due two days before my deadline to submit grades.
- Don't worry about handling duplicate items. If you follow the implementation that was shown in class, duplicates should be added to the left side of the tree. Gradescope will not test for duplicate entries. However, you are free to test what happens in case of duplicates.
- Make sure to take a look at the `remove` function, and it's helper function's implementation. If you understand how that is implemented, you should be able to implement the rest of the functions.

Debugging and Gradescope

Debugging is a skill you need to pick up at this point of your Computer Science journey. Before asking me to debug your code, make sure you do the following:

- If Gradescope is not compiling, check if your file names are exactly as stated in the submission instructions. Furthermore, check if you are using the same compiling command as stated in the instructions. If this fails, there is something wrong with your includes.
- Make sure all your functions are implemented. Make sure you are not changing the function prototypes from what was given in the starter code. Make sure you are not modifying anything that was specified not to be modified in the starter code.
- If you ask me to debug, give me the specific function, as opposed to "my code isn't working". Use your `main.cpp` and try to find which function the error is coming from.
- Check the instructions file **in the course webpage** to make sure nothing was added or modified. You can quickly find modifications by scrolling down to the Changelog. **Note that changes will not show up in the instruction PDF in your GitHub repository. It will show up in the project link from the course webpage.**

Changelog

While any major changes are unlikely, changes made to this document will be listed here