I&C SCI 46 Lecture Fall 2022 Project 5: Lewis Carroll revisited

Due **November 16 at 7:30 AM**. This is eligible for late submissions.

Introduction

We're going to revisit the word ladder puzzle from project three, using a different data structure to try a different type of search. This one is neither breadth-first nor depth-first. We will see how a priority queue helps make this a more *informed* search than those two.

Choosing a project partner

You *have the **option*** to work with a second person for this assignment. If you do so, I expect you to work via pair programming. That is, you may **not** split the assignment, such as by having one person implement the priority queue and the other person write code for the search code.

Similarly, any academic dishonesty arising from a group will be treated as an offense by both partners.

To declare a partnership, **both partners** need to fill out the following form by **November 10** at 7:30 AM. *There will be no exceptions granted. Be sure you fill it out correctly and that you know what your UCINetID is and how it is not your UCI ID number or your UCI email address. Be sure your partner has submitted it.*

https://docs.google.com/forms/d/e/1FAIpQLSf4we2njYHzsS-FexGUfouHhalaQYkp-7bjSTB8wYwcMnPK3w/viewform

Requirements

Part 1: Priority Queue

In this project you will be implementing the priority queue data structure as a class named `MyPriorityQueue`. The implementation must be an **array-based binary min-heap** (as shown in class). The `MyPriorityQueue` class consists of the following functions which you are responsible for implementing and have been started for you in `MyPriorityQueue.hpp`:

`MyPriorityQueue()`

> This is the constructor for the class. You should initialize any variables you add to the class here. Namely, you'll want to consider what structure you'll use to store the elements of your priority queue (hint: some kind of indexable array-like structure) and how it will be initialized.

`~MyPriorityQueue()`

This is the class destructor. You are responsible for freeing any allocated memory here. You will most likely be allocating memory to store the elements in the priority queue themselves. You **are** allowed to use `std::vector` in this project, which does allocate and free memory for you. If you decide to use a dynamically allocated array you will be responsible for freeing that yourself.

`size_t` **`size()`** `const noexcept`

This function returns the number of elements stored in the priority queue. It returns the count as a `size_t`. It is marked const (also known as a constant member function) because it should not *modify* any member variables that you've added to the class or call any function functions that are not marked const as well. The advantage of marking this function as `const` is that it can be called on constant `MyPriorityQueue` instances. It also allows the compiler to make additional optimizations since it can assume the object this function is called on is not changed. This is a fairly good StackOverflow answer that goes into additional detail.

`bool` **`isEmpty()`** `const noexcept`

This function simply returns whether or not the priority queue is empty, or in other words, if it does not contain any elements. Marked const because it should not change any member data. Marked noexcept because it should not throw any exceptions.

`void` **`insert`**`(const Object & elem)`

Insert an element of type `Object` into the priority queue. Recall from lecture that min-heaps have two properties:

1) They are a *complete* binary tree (**not** necessarily a binary *search* tree).
2) No parent has a value greater than either of its children.

We went over the insert procedure in the lecture, but section 8.1 of the Zybook does a decent job explaining it as well. The key idea is to insert the element in the bottom-most, left-most empty position (hence violating min-heap property #2 above, but preserving tree completeness), and then "percolating" the element up until the min-heap property is restored. Note that this should happen in *O(log n)* time where *n* is the number of nodes.

IMPORTANT: Just like the AVL tree, for comparing keys use the "natural" comparison offered by <.

`const Object & **min**() const`

Simply returns a constant reference to the minimum element in the priority queue. This should be pretty easy if you implemented the min-heap correctly. Note that this function is marked const because it should not modify any internal data of the MyPriorityQueue instance. Namely, it should *not* alter the structure of your heap. This function should run in *O(1)* time. Note: there is **not** a non-const version of `min()` for this project.

`void **extractMin**()`

Removes the minimum element from the priority queue. This interface is similar to the C++ container interface where `top()` is used to get the top-most element, and `pop()` is used to remove it, but `pop()` does not actually return the element itself. Just replace `top()` with `min()` and `pop()` with `extractMin()` conceptually. Note that because the heap invariants must be maintained, this function should also "fix" the heap after the minimum element has been removed. We went over this "fix" procedure as well in lecture. The key ideas are to replace the removed node with the bottom-most, rightmost filled node (preserves tree completeness property), and the "push down" this node until it's in a spot that no longer violates the min-heap property. A "push-down" involves a swap with the smallest child (value-wise, not height-wise). This function should run in *O(log n)* time.

Part 2: Word Ladder (revisited)

`std::vector<std::string> **convert**(const std::string & s1, const std::string & s2, const std::unordered_set<std::string> & words)`

`Read this part carefully.`

- Just like project three, this function will return the conversion between s1 and s2, according to the lowest *Lewis Carroll* distance. The first element of the vector should be s1, the last s2, and each consecutive should be one letter apart. Each element should be a valid word. If there are two or more equally least Lewis Carroll distance ways to convert between the two words, you may return any of them.

- If there is no path between s1 and s2, or if they are the same string, return an empty vector.

- Unlike project three, however, we are going to require a different algorithm. ***Read this carefully; reusing the algorithm from project three will be treated as academic dishonesty and sanctioned accordingly. Reusing your code***

and algorithm from project three is particularly egregious.

- In project three, you wrote this via a Breadth-First Search; a key component of that BFS was your queue. Your code started at s1, enqueued every "adjacent" (differing by one letter) word to s1 that appeared in the wordset, then dequeued whichever word had been in that queue the longest. That word was one of the words in the queue whose Lewis Carroll *distance from s1* was the smallest.

- In this project, instead of using a standard (FIFO) queue, we are going to instead use a *Priority Queue*. When we want to look for the next word to process, instead of taking the unexplored word with *a minimal Lewis Carroll distance from the start*, we are instead going to select the node with the smallest *sum* of two values: the first is the Lewis Carroll distance from the start and the second is the number of letters in the word that still differ from s2.
  - You can think of the latter value as an *optimistic estimate* of how many changes remain to convert the word in its current form to the end; if every change were a valid word, this would be the exact distance. If not, then it will take more than that many. Either way, there is no way to make fewer changes to reach s2.
- Use your priority queue to prioritize nodes with that sum of values. There are many ways to do this; we will leave it to you to choose how to do so.

- If you are going to have a struct as the object in your priority queue in this part -- which is definitely allowed -- you may want to "overload" the less-than operator instead of changing your priority queue code (which should rely on the < operator existing). For more on how to do that, please read https://en.cppreference.com/w/cpp/language/operators and/or talk to the professor about this.

Restrictions

You are explicitly **allowed** to use `std::vector, std::unordered_set, std::stack` and similar in your solution. In addition, you may use parts of the C++ standard library that solve small/trivial parts of the assignment (ex. `std::max, std::swap`) that you could implement yourself if you needed to. You may NOT use parts of the standard library that would do significant parts of the assignment for you (ex. `Std::priority_queue` or `std::make_heap()`). If you are unsure if something is considered a trivial part of the assignment, please ask.

There is one additional restriction: you may not declare any global functions or variables in MyPriorityQueue.hpp

Can I submit after the deadline?

Yes, it is possible, subject to the late work policy for this course, which is described in the section titled Late work in the course reference and syllabus.

Grading

Your grade for this project will be graded on correctness: I will run some number of test cases using Google test. Each is worth some number of points and is graded based on whether or not your code causes the test case to pass; this may test whether the priority queue works correctly or whether the word ladder was solved correctly. We also reserve the right to assign or deduct points for memory leaks as appropriate.   If it is determined that your program does not make an attempt to solve the problem at hand, you will not get these points, regardless of the result from testing. The tests will look a lot like the tests in your Google Test starting directory for this assignment; if you pass those, you’re off to a good start, but it’s not a guarantee.

The same caveats that applied to previous assignments apply here as well. Make sure you test your project extensively, including at least one test for every function in your priority queue and for multiple types. For best results, write Google Tests instead of relying on main.