ECS 36C: Programming Assignment #4

Instructor: Aaron Kaloti

Winter Quarter 2022

Contents

1 Changelog

You should always refer to the latest version of this document.

- v.1: Initial version.
- v.2: Autograder details.

2 General Submission Details

Partnering on this assignment is prohibited. If you have not already, you should read the section on academic misconduct in the syllabus.

This assignment is due the night of Wednesday, 03/09. Gradescope will say 12:30 AM on Thursday, 03/10, due to the "grace period" (as described in the syllabus). *Be careful about relying on the grace period for extra time; this could be risky.*

3 Purpose of This Assignment

- To increase your familiarity with hash tables, priority queues (binary heaps), and the change key version of a priority queue.
- To give you more experience with using templates to implement data structures (like you did in P1).
- To give you more experience with a more sizable coding assignment.

4 Reference Environment

The autograder, which is talked about at the end of this document, will compile and run your code in a Linux environment. That means that you should make sure your code compiles and behaves properly in a sufficiently similar environment. The CSIF, which I talk about in the syllabus, is one such environment. Do not assume that because your code compiles on an *insufficiently* similar environment (e.g. directly on your Mac laptop), it will compile on the Linux environment used by the CSIF.

You should avoid causes of undefined behavior in your code, such as uninitialized variables. If you have things like this in your code, then your code could end up generating the wrong answer when autograded, even if it generated the right answer when you tested it in a sufficiently similar environment.

5 Files

You will submit the following files:

- `hash_table.hpp`
- `hash_table.inl`
- `priority_queue.hpp`
- `priority_queue.inl`

You should start with the versions of `hash_table.hpp` and `priority_queue.hpp` provided on Canvas. You must create the other two files. You are allowed to modify the provided header files, but you cannot modify the names / arguments / return value types / etc. of the methods.

6 Restrictions

These restrictions will probably all be checked manually after the deadline. *If you have any questions about what is OK and what is not OK, feel free to ask me.*

You are NOT allowed to use any STL containers. `std::array`, `std::vector`, `std::unordered_map`, `std::map`, and all other STL containers are all banned. (Note that in this assignment, you are essentially making your own versions of `std::unordered_map` and `std::priority_queue`.) Smart pointers are allowed. **Using an STL container may result in you losing a huge amount of points, if not all points (depending on the severity), in this assignment, since it may defeat the point of a lot of the assignment.**

You should avoid looking up hash table or priority queue code as you do this assignment. You should use your understanding of hash table and priority queue concepts to do this assignment. Looking up resources on these concepts is fine, but looking up specific implementations is not. You can and (for your own protection) probably should cite online sources you consult in comments at the top of your file. Ultimately, the goal here is not to trap anyone; the goal is for the assignment to make sense and be worth doing, and I think we can all agree that it would be pretty stupid and a total waste of time if you were allowed to copy hash table code for an assignment that required you to implement a hash table.

Each method's declaration (in the provided header files) has a comment before it that may give a runtime requirement. I try to explain the runtime requirement as best I can, but when it comes to hash tables, it is a bit difficult to explain (you'll see what I mean when you read about the phrase "constant time" in double quotes), so feel free to seek clarification. **Violating a method's runtime requirement may result in you losing all of the points for that method.**

There will be autograder test cases that check for memory leaks. (As should come as no surprise, they will probably be hidden cases.)

7 Part #1: Hash Table

I give some tips for both parts after the section on part #2.

Implement each method that is declared in `hash_table.hpp`. Each method has a comment before it explaining its behavior. With the exception of `operator<<`, which seemingly must be defined within the class definition, it is up to you whether you define each method in `hash_table.hpp` or `hash_table.inl`. Note that the version of `hash_table.hpp` provided on Canvas includes `hash_table.inl` towards the very end.

On Canvas, you can find a file called `demo_hash_table.cpp`. The output of this program is given in `demo_hash_table_output.txt`. You should compile with the following command:

```
g++ -Wall -Werror -g -std=c++14 demo_hash_table.cpp -o demo_hash_table
```

8 Part #2: Priority Queue with Extended API

Implement each method that is declared in `priority_queue.hpp`. With the exception of `operator<<`, which seemingly must be defined within the class definition, it is up to you whether you define each method in `priority_queue.hpp` or `priority_queue.inl`.

On Canvas, you can find a file called `demo_priority_queue.cpp`. The output of this program is given in `demo_priority_queue_output.txt`. You should compile with the following command:

```
g++ -Wall -Werror -g -std=c++14 demo_priority_queue.cpp -o demo_priority_queue
```

9 Tips

- You can create structs, including templated structs. For instance, some of you may wish to create a struct that pairs a key and a value. Since the value is of a templated type, the struct would need the template argument too. You can create a struct like so:

```
template <typename ValueType>
struct Pair
{
    ...
};
```

- You can create helper methods. Note that if it would not make sense for a user of the class to call these methods, then such methods should be `private`, not `public`.
- *Unit testing*: I personally made significant use of `assert()`-based unit tests (which we talked about in slide deck #1). In an assignment like this, it is quite easy for you to do something in a later part that breaks an earlier part. For instance, you could break your hash table insertion when you implement the rehashing. Unit tests are great for catching such issues, since you can easily run all of the tests you have written and get immediate feedback thanks to `assert()` (rather than having to manually/visually compare the output with the expected output).
- *Regarding smart pointers*: If you use smart pointers, the destructors are trivial. Of course, the drawback of this is that getting used to smart pointers is a bit challenging, because the way to use them (e.g. creating a smart pointer instance) isn't intuitive sometimes, and since smart pointers are template types, the compiler error messages are a bunch of garbage sometimes. However, if you are going to put C++ on your résumé and potentially apply to C++ coding jobs, knowing more C++ features is better than knowing less.

10 Other Notes

The information in this section is not essential for the assignment. It is miscellaneous information/thoughts for the curious student that I felt was too important to not share with you.

- Some of the accessor methods return a null pointer to indicate that an element cannot be found, forcing the return type to be a pointer. A reference cannot be used here because a null reference is prohibited. I could have made the return type a reference, but this would have required that these accessor methods instead throw exceptions to indicate that an element cannot be found; I think the STL containers do this.
- *Regarding handling of duplicates*: Both the hash table and the priority queue that you implement in this assignment reject duplicates. During lecture, we have stayed away from how duplicates may be handled, since the handling of duplicates (e.g. whether they should be rejected or not) is too application-dependent. In my opinion, this assignment gives some idea of why rejecting duplicates is not a good default. For instance, it is rather awkward that the `PriorityQueue` methods `decreaseKey()` and `increaseKey()` have to reject key changes that would lead to duplicate keys. This would make the priority queue worthless if it were to be used in implementing an algorithm like Dijkstra's algorithm or Prim's algorithm, since ties could easily come up.

11 Grading Breakdown

As stated in the updated syllabus, this assignment is worth 10% of your final grade.

12 Autograder Details

If you haven't already, you should read what I say in the syllabus about the Gradescope autograder.

I don't think there will be as many timeout issues as there were in P3, but of course, if you get a timeout issue, then – if it isn't caused by an infinite loop or your program's not finishing on that particular test case (which you can verify since you are given all of the visible test cases' inputs / `main()` implementations) – you should email me about it.

You will submit the following files:

- `hash_table.hpp`
- `hash_table.inl`
- `priority_queue.hpp`
- `priority_queue.inl`

Your output must match mine *exactly*, except that the autograder ignores trailing whitespace.

The autograder will grade your submission out of 99 points.

You can find the visible test cases' inputs (the `main()` implementations) on Canvas.