∂

CSE 333 25au Homework 1

Out: Monday, September 29, 2025
Due: Thursday, October 9, 2025 by 11:59 pm

Goals

For Homework #1, you will finish our implementation of two C data structures: a doubly-linked list (Part A) and a chained hash table (Part B). You will gain experience and proficiency with C programming, particularly memory management, pointers, and linked data structures.

Please read through this entire document before beginning the assignment, and please start early! This assignment involves messy pointer manipulation and malloc/free puzzles, and these can cause arbitrarily awful bugs that take time and patience to find and fix.

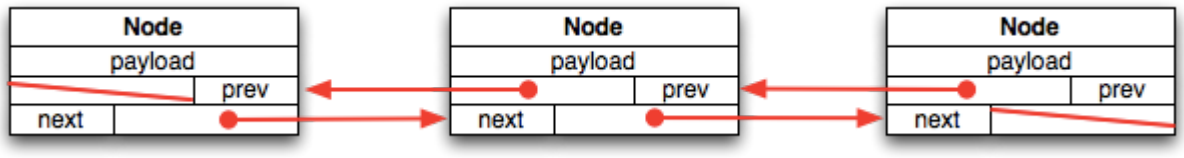
All CSE 333 assignments, including this homework, are only supported on the *current* CSE Linux environment (attu, CSE lab workstations, and the 25au CSE Home Linux VM). We *do not* support building and running this assignment in any other work environments, including other versions of Linux.

C Data Structures

Part A: Doubly-Linked List

If you've programmed in Java, you're used to having a fairly rich library of elemental data structures upon which you can build, such as vectors and hash tables. In C, you don't have that luxury: the C standard library provides you with very little. In this assignment, you will add missing pieces of code in our implementation of a generic doubly-linked list.

At a high-level, a doubly-linked list looks like this:

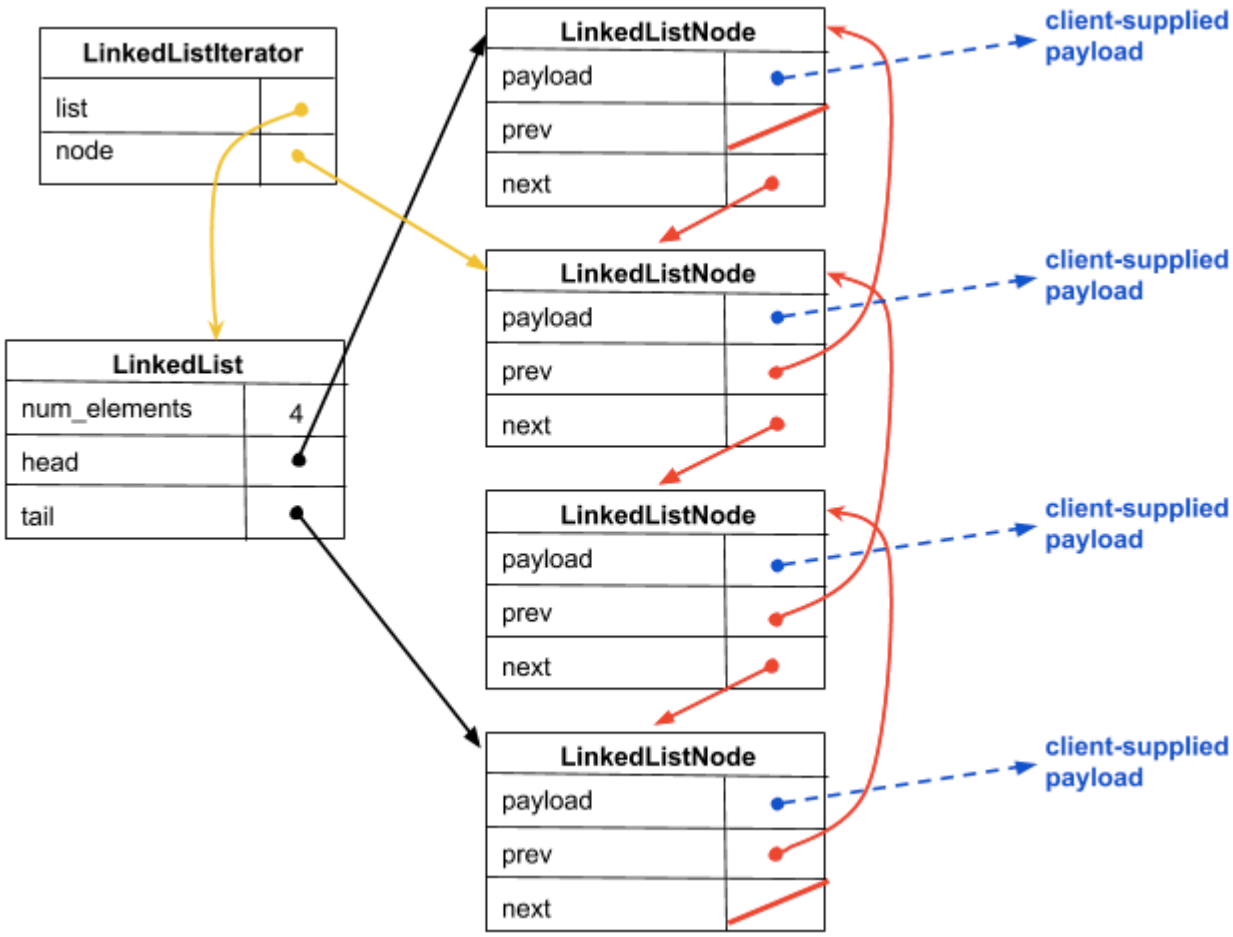


Each node in a doubly-linked list has three fields: a payload, a pointer to the previous element in the list (or `NULL` if there is no previous element), and a pointer to the next element in the list. If the list is empty, there are no nodes. If the list has a single element, both of its next and previous pointers are `NULL`.

So, what makes implementing this in C tricky? Quite a few things:

- We want to make the list useful for storing arbitrary kinds of payloads. In practice, this means the payload element in a list node needs to be a pointer supplied by the customer of the list implementation. Given that the pointer might point to something malloc'ed by the customer, this means we might need to help the customer free the payload when the list is destroyed.
- We want to hide details about the implementation of the list by exposing a high-level, nicely abstracted API. In particular, we don't want our customers to fiddle with next and previous pointers in order to navigate through the list, and we don't want our customers to have to stitch up pointers in order to add or remove elements from the list. Instead, we'll offer our customers nice functions for adding and removing elements and a Java-like iterator abstraction for navigating through the list.
- C is not a garbage-collected language: you're responsible for managing memory allocation and deallocation yourself. This means we need to be malloc'ing structures when we add nodes to a list, and we need to be free'ing structures when we remove nodes from a list. We also might need to malloc and free structures that represent the overall list itself.

Given all of these complications, our actual linked list data structure ends up looking like this:



Specifically, we define the following types and structures:

- **LinkedList**: The structure containing our linked list's metadata, such as head and tail pointers. When our customer asks us to allocate a new, empty linked list, we malloc and initialize an instance of this structure then return a pointer to that malloc'ed structure to the customer.
- **LinkedListNode**: this structure represents a node in a doubly-linked list. It contains a field for stashing away (a pointer to) the customer-supplied payload and fields pointing to the previous and next LinkedListNode in the list. When a customer requests that we add an element to the linked list, we malloc a new LinkedListNode to store the pointer to that element, do surgery to splice the `LinkedListNode` into the data structure, and update our `LinkedList` 's metadata.
- **LLIterator**: sometimes customers want to navigate through a linked list; to help them do that, we provide them with an iterator. `LLIterator` contains bookkeeping associated with an iterator. In particular, it tracks the list that the iterator is associated with and the node in the list that the iterator currently points to. Note that there is a consistency problem here: if a customer updates a linked list by removing a node, it's possible that some existing iterator becomes inconsistent because it referenced the deleted node. So, we make our customers promise that they will free any live iterators before mutating the linked list. (Since we are generous, we do allow a customer to keep an iterator if the mutation was done using that iterator.) When a customer asks for a new iterator, we malloc an instance and return a pointer to it to the customer.

Instructions

You should follow these steps to do this part of the assignment:

1. Make sure you are comfortable with C pointers, structures, malloc, and free. We will cover them in detail in lecture, but you might need to brush up and practice a bit on your own; you should have no problem Googling for practice programming exercises on the Web for each of these topics.

2. Get the source files for hw1. Navigate to a directory that contains a checked-out copy of your cse333 Git repository and run the command `git pull`. (See the CSE 333 Gitlab Guide (../../gitlab/index.html) linked from the course Resources page for some tips if the `pull` command fails because you have unstaged changes or other problems. Github also has a useful document about how to resolve merge conflicts (link (https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/addressing-merge-conflicts/resolving-a-merge-conflict-using-the-command-line)). In our case, the default merge actions suggested by git should do the job since hw1 adds new folders to your gitlab repository but should not be introducing any changes to your existing files or folders.) After the `pull` command finishes you should see at least the following directories and files in your repository:

   ```
   bash$ ls
   exercises gtest  hw0  hw1
   ```

3. Look inside the hw1 directory. You'll see a number of files and subdirectories, including these that are relevant to Part A:

   - **Makefile**: a makefile you can use to compile the assignment using the Linux command `make` on the CSE Linux machines.

   - **LinkedList.h**: a header file that defines and documents the API of the linked list. A customer of the linked list includes this header file and uses the functions defined within in. Read through this header file very carefully to understand how the linked list is expected to behave.

   - **LinkedList_priv.h**: a *private header file* included by `LinkedList.c`; it defines the structures we diagrammed above. These implementation details would typically be withheld from the client by placing the contents of this header directly in `LinkedList.c`; however, we have opted to place them in a "private .h" instead so that our unit test code can verify the correctness of the linked list's internals.

   - **LinkedList.c**: contains the partially completed implementation of our doubly-linked list. Your task will be to finish the implementation. Find the labels that say "STEP X:" -- these labels identify the missing pieces of the implementation that you will finish. Take a minute and read through both `LinkedList_priv.h` and `LinkedList.c`

   - **example_program_ll.c**: this is a simple example of how a customer might use the linked list; in it, you can see how a customer can allocate a linked list, add elements to it, create an iterator, use the iterator to navigate a bit, and then clean up.

- **test_linkedlist.cc**: this file contains unit tests that we wrote to verify that the linked list implementation works correctly. The unit tests are written to use the Google Test (http://code.google.com/p/googletest/) unit testing framework, which has similarities to Java's JUnit testing framework. As well, this test driver will assist the course staff in grading your assignment. As you add more pieces to the implementation, the test driver will make it further through the unit tests, and it will print out a cumulative score along the way. You don't need to understand what's in the test driver for this assignment, though if you peek inside it, you might get hints for what kinds of things you should be doing in your implementation!

- **solution_binaries**: in this directory, you'll find some Linux executables, including `example_program_ll` and `test_suite`. These binaries were compiled with a complete, working version of `LinkedList.c`; you can run them to explore what **should** be displayed when your assignment is working!

4. Run `make` on a CSE Linux machine to verify that you can build your own versions of `example_program_ll` and `test_suite`. `make` should print out a few things, and you should end up with new binaries inside the hw1 directory.

5. Since you haven't yet finished the implementation of `LinkedList.c`, the binaries you just compiled won't work correctly yet. Try running them, and note that `example_program_ll` halts with an assertion error or a segfault and `test_suite` prints out some information indicating failed tests, and may crash before terminating.

6. This is the hard step: finish the implementation of `LinkedList.c`. Go through `LinkedList.c`, find each comment that says "STEP X", and place working code there (please keep the "STEP X" comment for your graders' sanity so they can locate your code!). The initial steps are meant to be relatively straightforward, and some of the later steps are trickier. You will probably find it helpful to read through the code from top to bottom to figure out what's going on. You will also probably find it helpful to recompile frequently to see what compilation errors you've introduced and need to fix. When compilation works again, try running the test driver to see if you're closer to being finished.

   - Note: You ***may not*** modify any header files or interfaces in this or later project assignments. We may test your code by extracting your implementations and compiling them with the original header files or in some other test harness where they are expected to behave as specified. You certainly are free, of course, to add additional private (eg, `static`) helper functions in your implementation, and you should do that when it improves modularity.

   - Debugging hint: `Verify333` is used in many places in the code to check for errors and terminate execution if something is wrong. You might find it helpful to discover the function that is called when this happens so you can place a debugger breakpoint there.

7. We'll also be testing whether your program has any memory leaks. We'll be using Valgrind (http://valgrind.org/) to do this. To try out Valgrind for yourself, do this:

- From the hw1 directory run the following command:

  ```
  valgrind --leak-check=full ./solution_binaries/example_program_ll
  ```

  Note that Valgrind prints out that no memory leaks were found. Similarly, try running the test driver under Valgrind:

  ```
  valgrind --leak-check=full ./solution_binaries/test_suite
  ```

  and note that Valgrind again indicates that no memory leaks were found.

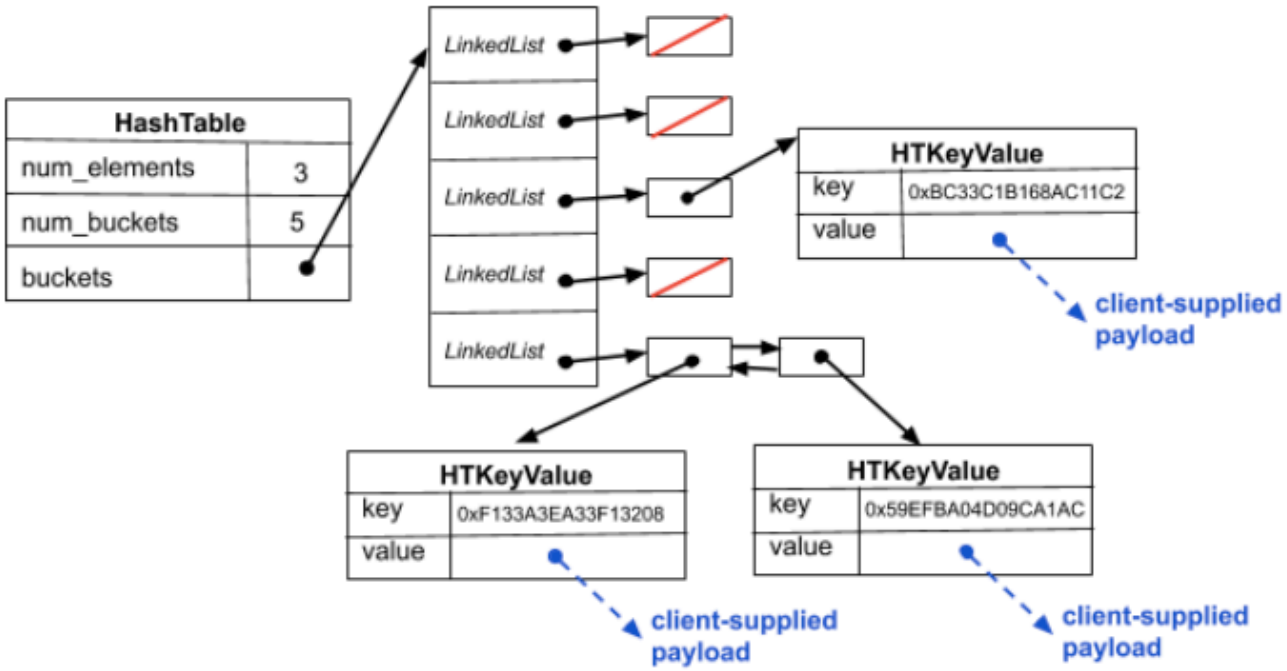
- The previous commands check the supplied sample solutions for memory leaks to demonstrate how Valgrind works. You will want to check your own code to be sure it doesn't have memory problems. While still in the hw1 directory, compile your versions of the `example_program_ll` and `test_suite` binaries, and try running them under Valgrind. If you have no memory leaks and the `test_suite` runs the linked list tests to completion, you're done with Part A!

Part B: Chained Hash Table

A chained hash table is a data structure that consists of an array of buckets, with each bucket containing a linked list of elements. When a user inserts a key/value pair into the hash table, the hash table uses a hash function to map the key into one of the buckets, and then adds the key/value pair onto the linked list. There is an important corner case: if the key of the inserted key/value pair already exists in the hash table; our implementation of a hash table *replaces* the existing key/value pair with the new one and returns the old key/value pair to the customer.

Over time, as more and more elements are added to the hash table, the linked lists hanging off of each bucket will start to grow. As long as the number of elements in the hash table is a small multiple of the number of buckets, lookup time is fast: you hash the key to find the bucket, then iterate through the (short) chain (linked list) hanging off the bucket until you find the key. As the number of elements gets larger, lookups become less efficient, so our hash table includes logic to resize itself by increasing the number of buckets to maintain short chains.

As with the linked list in Part A, we've given you a partial implementation of a hash table. Our hash table implementation looks approximately like this:



Specifically, we defined the following types and structures:

- **HashTable**: The structure containing our hash table's metadata, such as the number of elements and the bucket array. When our customer asks us to allocate a new, empty hash table, we malloc and initialize an instance of this (including malloc'ing space for the bucket array that it uses and allocating `LinkedLists` for each bucket), and return a pointer to that malloc'ed structure to the customer.
- **HTIterator** (not shown in the diagram): sometimes customers want to iterate through all elements in a hash table; to help them do that, we provide them with an iterator. `HTIterator` points to a structure that contains bookkeeping associated with an iterator. Similar to a linked list iterator, the hash table iterator keeps track of the hash table the iterator is associated with and in addition has a linked list iterator for iterating through the bucket linked lists. When a customer asks for a new iterator we malloc an `HTIterator` and return a pointer to it.

Instructions

You should follow these steps to do this part of the assignment:

1. The code you fetched in Part A also contains the files you'll need to complete your hash table implementation and test it. Similar to the linked list, the hash table implementation is split across a few files: `HashTable.c` contains the implementation you need to finish, `HashTable.h` contains the public interface to the hash table and documents all of the functions & structures that customers see, and `HashTable_priv.h` contains some private, internal structures that `HashTable.c` uses.
2. Read through `HashTable.h` first to get a sense of what the hash table interface semantics are. Then, take a look at `example_program_ht.c`; this is a program that uses the hash table

interface to insert/lookup/remove elements from a hash table, and uses the iterator interface to iterate through the elements of the hash table.

3. As before, `test_hashtable.cc` contains our Google Test unit tests for the hash table. Run this -- on its own, and using valgrind -- to see how close you are to finishing your hash table implementation.

4. Look through `HashTable.c`, find all of the missing pieces (identified by STEP X comments, as before), and implement them.

5. As before, in `solution_binaries`, we've provided linux executables (i.e. `example_program_ht` and the same `test_suite`) that were compiled with our complete, working version of `HashTable.c` You can run them to explore what should be displayed when your part B implementation is working and look at the source code for examples of how to use the data structures.

Bonus: Code Coverage Statistics

You'll notice that we provided a second Makefile called `Makefile.coverage`. You can use it to run the `gcov` code coverage generation tool. Figure out how to (a) use it to generate code coverage statistics for `LinkedList.c` and `HashTable.c`, (b) note that the code coverage for HashTable is worse than that for the LinkedList, and (c) write additional HashTable unit tests to improve HashTable's code coverage.

The bonus task is simple, but we're deliberately providing next to no detailed instructions on how to do it – figuring out how is part of the bonus task!

Please make sure your additional unit tests don't change the scoring mechanism that we use, obviously. (We'll be checking that.) Place your additional unit tests in a separate file from the original test suite. That will make it easier for us to find and evaluate your tests.

If you do this bonus part of the project, you **must** include a `hw1-bonus` tag in your repository to identify the commit that contains the bonus work. This tag is in addition to the `hw1-final` tag for the basic project, which must still be present (see the Submission section below for more about tags that need to be included). When grading the project we will still test the `hw1-final` basic part of the project and it must work correctly, even if you do the bonus part. If your repository contains a `hw1-bonus` tag, we will evaluate your work for this bonus part. If that extra tag is not present, we will assume you did not do the bonus part, which is fine and will not affect your grade for the basic project in any way.

Also, if you do the bonus part of the project, you **must** add a `readme-hw1-bonus.md` text file to your `hw1/` directory and include this in the files you push to your gitlab repo. This file should contain a brief description of the bonus work you have done, including your observations about code coverage in the original project, the changes you made to increase this, and the results you observed. This should be brief - a few sentences or a couple of paragraphs at the most should be sufficient, and you do not need to provide extensive charts, diagrams, or data.

Testing

As with hw0, you will compile your implementation using the `make` command. This creates several output files, including an executable called `test_suite`, which contains a variety of tests for all parts of the project. You can run all of the tests with the command

```
bash$ ./test_suite
```

It is also possible to run only selected portions of the tests by supplying arguments to the `test_suite` program. For example, to only run the LinkedList tests, you can use the command

```
bash$ ./test_suite --gtest_filter=Test_LinkedList.*
```

If you only want to test Push and Pop from LinkedList, you can use

```
bash$ ./test_suite --gtest_filter=Test_LinkedList.PushPop
```

You can specify which tests are run for any parts of the assignment. You just need to know the names of the tests, and you can do this by running

```
bash$ ./test_suite --gtest_list_tests
```

These settings can be helpful when debugging specific parts of the assignment, especially since `test_suite` can be run with these options when it is executed by valgrind or gdb! One caution though: some parts of `test_suite` are fairly complex. If one of the larger tests fails it can often be very frustrating to try to debug your code by digging through the complex test code to figure out what happened. An often effective strategy is to use the `test_suite` program to identify parts of your code that seem to be misbehaving, then write some small test programs of your own to isolate the problems in a much simpler setting and debug/fix them there.

Code Quality (Style)

In addition to passing tests, your code should be high quality and readable. This includes several aspects:

- **Modularity:** Your code should be divided into reasonable modules (functions) and should not have excessive redundancies that could be removed by replacing redundant code with calls to suitable, possibly new, functions. If you create any additional private (e.g., `static`) helper functions, be sure to provide good comments that explain the function inputs, outputs, and

behavior. These comments can often be relatively brief as long as they convey to the reader the information needed to understand how to use the function and what it does when executed.
- **Readability:** Your code should blend smoothly with the code surrounding it. Follow the existing conventions in the code for capitalization; naming of functions, variables, and other items; using comments to document aspects of the code; and layout conventions such as indenting and spacing.
- **Style checker (linter):** Use the `cpplint.py --clint` tool to check for style issues. Be sure to fix issues reported before submitting your code. Exception: if `cpplint` reports style problems in the supplied starter code, you should leave that code as-is.
- **Style guide:** Refer to the Google C++ Style Guide for advice. Much of the guide applies equally well to C code as well as C++.
- **Good development practices:** We will look through your git activity (eg, tags and commits) to verify that you are following the development practices discussed in sections and lecture. This includes correct tagnames, succinct commit messages, and incremental checkins (eg, commiting after major milestones like passing a test or implementing a feature or finishing a section of the assignment, and specifically *not* having one giant commit at the end of the project with all of your code and changes but no incremental commits of earlier parts of the project).

Project Submission & Evaluation

When you are ready to turn in your assignment, you should follow exactly the same procedures you used in hw0, except this time tag the repository with `hw1-final` (spelled *exactly* that way) instead of `hw0-final`. Remember to clean up and commit *and push* all necessary files to your repository before you add and push the tag.

If you do the bonus part of the assignment, you should also create and push a `hw1-bonus` tag corresponding to the commit with the extra work. If this tag is present, it will be used to check out and grade the bonus part of the project. If it is not present, we will assume you didn't do the bonus part, which will have no effect on your grade for the basic project. The basic project `hw1-final` tag must still be present in the repository and the basic part of the project will be graded separately even if the bonus tag is present.

After you have created and pushed the tag(s), ***be absolutely sure*** to test everything ***ON ATTU OR A LAB LINUX WORKSTATION OR THE CURRENT CSE LINUX VM*** by creating a new clone of the repository in a separate, empty directory, checkout the `hw1-final` tag, and verify that everything works as expected. If you did the bonus task, also checkout the `hw1-bonus` tag and test that code. If you discover any problems, you ***must*** delete this new repository copy (clone) you've used for verification and fix the problems in your original working repository. Then make a new clone and

check again to be sure the problems are really fixed. Refer to the hw0 submission instructions for details and follow those steps carefully, including steps for deleting a tag and then tagging a later commit if you need to make some changes to the version you initially tagged.

If you fail to check your work and your project doesn't build properly when the same steps are done by the course staff to grade it, you may lose a huge amount of the possible credit for the assignment even if almost absolutely everything is actually correct.

Grading

We will be basing your grade on several elements:

- The degree to which your code passes the unit tests. If your code fails a test, we won't attempt to understand why: we're planning on just using the number of points that the test drivers print out.
- We have some additional unit tests that test a few additional cases that aren't in the supplied test drivers. We'll be checking to see if your code passes these as well.
- The quality and readability of your code. We'll be judging this on several qualitative aspects described above.

Both code correctness and code quality matter. Both are weighed significantly in the evaluation of your project.