Goals

Implement a singly linked list that stores key value pairs. Nodes in the list can't share the same key. If a key/value pair is inserted into the list and another node in the list already has that key, that node's value is replaced with the new value. You need to implement all functions as described in the header file. The comments above each function tell you what you need to do, but if you have any questions post them on Piazza.

Submission instructions

Upload your hash_list.h and hash_list.cpp to Gradescope. Any tests that you pass will be in green, any that you fail will be in red, and for any failed test the output from that run (ie. everything that was printed to the terminal) will be displayed in a box under the tests name

You may print out useful information for debugging purpose. However, when making your final submission on Gradescope, please make sure to comment out these printing statement. You may submit multiple time on Gradescope.

Restrictions

- Your .cpp file is only allowed to include hash_list.h, no other includes are allowed.
- You may not use any standard containers.
- You aren't allowed to delete any provided public/private functions/member variables from hash_list.h. You may add private or public functions/member variables if you want, but you can't remove anything from hash_list.h that was there when we distributed it to you.

Example

Let's suppose we have a list with one node

```
list_head -> old_node(key = 1, value = 0.3)
```

We decide to insert the following node into the list

```
new_node(key = 0, value = 10)
```

We first traverse the list and find that there are no nodes that share a key with our new_node, so we're free to insert the new_node into our list. In this example we're inserting at the front of the list. Your implementation is free to insert nodes into the list wherever you want, you just need to make sure that no two nodes share a key.

```
list_head-> new_node(key = 0, value = 10) -> old_node(key = 1, value =
   0.3)
```

Let's then insert another node that shares a key with the original node

```
duplicate_key_node(key = 1, value = 0.9)
```

Since the key is already shared with another node we replace the value of old_node with the value of duplicate_key_node

```
list_head-> new_node(key = 0, value = 10) -> old_node(key = 1, value =
   0.9)
```

std::optional<T> discussion

Imagine you implement a function for your list that takes an index as an input and returns the value of the node at that index.

```
int get_value_at_index(int index)
{
    //return value at node index
}
```

What should you do if the user passes you an index that's larger than the size of your list? Or they pass in a negative value? You could return a special value, say -1, but there's the possibility that -1 is actually stored in your list, so a special placeholder value doesn't work. Really what you want is to return 2 things.

```
1. Some indication that your function succeeded
2. A value that is only relevant if the function succeeded
```

In C you could do something like this (assuming you've included stdbool.h)

```
int get_value_at_index(int index, bool* success)
{
    if (index >= size || index < 0)
    {
        *success = false;
        return 0;
    }

    *success = true;
    // search for value at index and return it
}
```

This solves the problem, but it leads to an additional problem. What happens if you call get_value_at_index but then forget to check if it succeeded?

```
// assume we have a list called our_list
bool success;
int index = 10;

int ret_val = our_list.get_value_at_index(index, &success);

// forgot to check if the function succeeded, now your program might
   print out garbage

std::cout << "Value at index " << index << " is " << ret_val << std::endl;
```

The same issue exists with null pointers. Functions that return a pointer usually indicate failure by returning a null pointer, but the user is free to use the returned pointer without checking if it's null. The best case scenario here is that your program crashes immediately and you spend a few hours debugging. The worst case is that your program doesn't crash, you ship the code, and then your bug leads to a security vulnerability that costs your company a lot of money.

The fundamental issue is that we want to force the caller of our function to check if our function succeeded before they use the value we returned. And this is where optionals come into play.

Optionals are basically a struct that holds a boolean flag and a value (note: this isn't real c++ code, but it's close enough to be useful)

```
struct Optional<T>
{
    bool has_value;
    T stored_value;
}
```

Note: That weird `<T>` syntax is a template, something that we'll get to later in the course. For now all that you need to know is that

```
1. T needs to be the type you want to store in the optional (so int,
   float, std::string, etc)
2. When you declare an optional you need to specify that type it will
   contain. So std::optional<int> is an optional that holds an integer.
```

The reason this is so powerful is that now you can return the success flag and the value (if the function succeeded) as one object. The code below gives an example of what this looks like

```
std::optional<int> get_value_at_index(int index)
{
    if (index >= size)
    {
        return std::nullopt; // std::nullopt is an empty optional, could
           also just return {}
    }

    // search for value at index and return it
}
```

Now the user can't just use the optional without thinking about if it succeeded

```
int index = 10;
std::optional<int> ret_val = our_list.get_index(10);

std::cout << "Value at index " << index << " is " << ret_val.value() <<
   std::endl; // Will crash if ret_val is empty
```

This might look the same but the key difference is that if `ret_val` doesn't contain a value, the call to `value()` will fail and the program will crash. This means that even if you forget to check if the optional has a value you won't get silent errors like in C. Once you realize why your program crashed you'll add a check like the following

```
int index = 10;
std::optional<int> ret_val = our_list.get_index(10);

if (ret_val.has_value()) // has_value() returns true if optional has a
   value, otherwise false
{
    std::cout << "Value at index " << index << " is " <<
       ret_value.value() << std::endl;
}
```

Now we've forced the user to check if the function succeeded (unless you want your program to crash)*. This makes it much less likely that you'll end up hunting for bugs later, as the crash will happen when you try and access the value as opposed to at some random point in your program.

*There is a small caveat that you could still ignore checking if your optional has a value and just get lucky and only pass in valid indexes. This is why good testing is required; if you have a good test suite the chance

of using the value of an unverified optional AND never passing in invalid arguments to the functions you're testing is small.

If you have questions about optionals I recommend this piece of documentation. This also shows you how to return an optional that contains a value. The above example only shows you how to return an empty optional.

Destructor discussion

You should pay particular attention to the implementation of the destructor for this lab. Here are a few things to think about

- Why should you define an explicit destructor for this lab? [Hint: Memory leaks]
- Your destructor can't just delete the head of the linked list. Why? What pitfalls could arise?

For questions about how destructor work I recommend looking at the following FAQ page

Testing Locally

We have provided some example traces and a `main.cpp` file to help you test your code locally before submitting. The `main.cpp` contains some basic tests for both parts 1 and 2 of the lab, and the trace files are examples of the traces we will run your program on when testing.

The provided `Makefile` has targets `part1` and `part2` to run the default tests for parts 1 and 2 in `main.cpp`.

To build and run `part1`, run:

```
make part1
```

Similarly to build and run `part2`, run:

```
make part2
```

There are two forms of testing: unit tests and integration tests. Unit tests verify the functionality of individual components and will be discussed in class. We have provided five integration tests in the form of "trace" functions that perform a series of Inserts, Gets, and Removes on your list. We are providing only a subset of traces used on Gradescope. You must do systematic local testing on your own to increase confidence your code would work on a more general set of traces beyond the ones we make available to you.