Goals

In the previous part of this lab, you implemented a statically typed hashmap (ie. the key had to be an int and the value had to be a float). In this part you're going to be implementing a templated hash map (this will also require modifying your hash list). You need to make the map capacity dynamic, which involves keeping the load factor within a certain range. (See load factor section below for details).

Instructions

You will need to

```
* Change your hash map and hash list code to support templates
* Replace your hashing function (see section on std::hash below for
   details)
* Implement dynamic resizing as well as whatever logic is necessary for it
  (you may consider implementing two private functions a need_to_rehash()
     and rehash() to make this easier)
* Implement get_all_sorted_keys()
```

Note: Your implementations needs to be put into `.hpp` files, NOT `.cpp` files. Similar to the previous part of this lab, we have not provided a `main.cpp` file; you should write your own to test with.

To submit this assignment you must upload the following files to Gradescope.

```
hash_map.h
hash_map.hpp    -> Must implement everything in hash_map.h
hash_list.h
hash_list.hpp   -> Must implement everything in hash_list.h
```

std::hash<K>

In the previous part of this lab your hashing function was the following

```
size_t index = abs(key) % _capacity;
```

This works for integers, but now that we're doing a generalized hash map we need a generalized hashing function that can hash any type. And you can't take the absolute value of a string, or a node, or an arbitrary type user defined type. As a result we need a more general hash function, which is what std::hash<K> does. std::hash<K> creates an object that you can use to hash any object of type K. In our case K is the type of our key, so we can now hash any input type correctly

To hash a key of type K all you need to do it

```
size_t index = _hash(key) % _capacity;
```

Load factor and resizing

Hash maps only work well if the ratio of elements in the map to the number of buckets is below a certain threshold. That threshold is called the load factor. For the second part of this lab you're going to be dynamically updating the size of the hash_map whenever the load factor goes above an upper threshold or below a lower threshold. We've provided the capacities to use in hash_map.h.

If the following condition is true following an insert/remove then you go to next highest capacity and re-hash.

```
    // We check this condition AFTER an insert/remove
    _size > _upper_load_factor * capacity
```

If the following condition is true following an insert/remove then you go to next lowest capacity and re-hash.

```
    // We check this condition AFTER an insert/remove
    _size < _lower_load_factor * capacity
```

note: only one of these conditions will ever be true at a time (as long as _upper_load_factor > _lower_load_factor, which it always will be when grading).

This is a good explanation of how dynamic resizing works. This only explains expanding the capacity of the hash map, but the same approach applies for making the hash list smaller.

Note: If your capacity is at/above the maximum capacity you don't increase the capacity any more, regardless of how large the load factor becomes. Similarly, if your capacity is at/below the minium capacity you don't decrease it any more, regardless of how small the load factor becomes. We will also only be testing your code with initial capacities that are between the highest and lowest capacities given in hash_map.h.

Poorly chosen upper and lower load factors can cause undesirable situations. For example:

- Assume we have two capacity options 209 and 1021, and we choose the lower load factor to be 0.95 and upper load factor to be 1.
- When you insert the 210th element, it will be above the upper load factor. After increasing the capacity, the load factor becomes 210 / 1021 = 0.206, which is already lower than the lower load factor.

Your code should handle such cases by only rehashing once for any given insert/remove to avoid an infinite loop. In the above example, the insert would finish with the load factor being 0.206.

Restrictions

- You must submit all listed files
- Your hash_map.hpp file is only allowed to include hash_map.h and must implement all functions in hash_map.h
- You may not use any standard containers.
- You aren't allowed to delete any provided public/private functions/member variables from hash_map.h. You may add private or public functions/member variables if you want, but you can't remove anything from hash_map.h that was there when we distributed it to you.