CSCI-1200 Data Structures — Fall 2019
Lab 12 — Priority Queues and Hash Tables

Checkpoint 1

estimate: 20-40 minutes

Let's finish the implementation of *heap sort*, a classic $O(n \log n)$ sorting algorithm. Heap sort has better worst case performance that *quicksort* (which can suffer from poor pivot selection) and better memory usage than *merge sort* (which requires a scratch vector of size n).

http://www.cs.rpi.edu/academics/courses/fall19/csci1200/labs/12_priority_queues_hash_tables/heapsort.cpp

The provided code does not implement a full Priority Queue / Binary Heap class, but instead works directly on the provided vector. You need to implement the *heapify* function and finish the *heapsort* function.

First, work out on paper the intermediate result (vector contents) should be after we call *heapify* for the two test input vectors. Then finish the code and verify your answer. For your first draft you may use a scratch vector. Once that's debugged (if time allows) you can modify the code as needed to only use the input vector. *Hint: You should not call* `push_back` *or* `pop_back` *on your vector!*

To complete this checkpoint and the entire lab: Show a TA your completed and debugged heap sort code, including your diagram of the heapified input vectors.

Checkpoint 2

estimate: 15-30 minutes

Now let's switch gears completely to hash tables. Download the following code:

http://www.cs.rpi.edu/academics/courses/fall19/csci1200/labs/12_priority_queues_hash_tables/ds_hashset.h
http://www.cs.rpi.edu/academics/courses/fall19/csci1200/labs/12_priority_queues_hash_tables/test_ds_hashset.cpp

Implement and test the `insert` function for the hashset. The insert function must first determine in which bin the new element belongs (using the hash function), and then insert the element into that bin *but only if it isn't there already.* The insert function returns a pair containing an iterator pointing at the element, and a bool indicating whether it was successfully inserted (true) or already there (false).

For the second part of this checkpoint, experiment with the hash function. In the provided code we include the implementation of a good hash function for strings. Are there any collisions for the small example? Now write some alternative hash functions. First, create a trivial hash function that is guaranteed to have many, many collisions. Then, create a hash function that is not terrible, but will unfortunately always place anagrams (words with the same letters, but rearranged) in the same bin. Test your alternate functions and be prepared to show the results to your TA.

To complete this checkpoint: Show a TA your debugged implementation of `insert` and your experimentation with alternative hash functions.

Checkpoint 3

estimate: 20-40 minutes

Next, implement and test the `begin` function, which initializes the iteration through a `hashset`. Confirm that the elements in the set are visited in the same order they appear with the `print` function (which we have implemented for debugging purposes only).

Finally, implement and test the `resize` function. This function is automatically called from the `insert` function when the set gets "too full". This function should make a new top level vector structure of the requested size and copy all the data from the old structure to the new structure. Note that the elements will likely be shuffled around from the old structure to the new structure.

To complete this checkpoint: Show a TA these additions and the test output.