# Goals

- Additional practice with key-value pair collections and the Map API;
- Practice implementing hash tables via separate chaining;
- Additional practice creating realizable (dynamic) arrays;
- More practice with asymptotic analysis and performance tests.

Note that you may use whatever environment you like for this class, but your programs must be able to compile and run on ada (which is running Ubuntu) using `g++`, `cmake`, `make`, `valgrind`, and `gdb`. *It is highly recommended, however, that you use the Department-supplied virtual machine (VM) on your computer for this class (which is also based on Ubuntu).*

# Instructions

1. Accept the GitHub classroom repo for HW-6, and clone it to your local environment. See piazza for the classroom link.

2. Copy your `map.h`, `sequence.h`, `arrayseq.h`, `arraymap.h`, and `binsearchmap.h` files into your repository from HW-5. Be sure these files are included in your final submission.

3. Implement `HashMap` (in `hashmap.h`). See below for additional details. Be sure to read the details below carefully.

4. Ensure your function implementations pass all provided unit tests. Note that the unit tests provided are not comprehensive, i.e., even if all unit tests for your implementation pass, there still could be issues (which may reveal themselves in the performance tests).

5. Ensure your implementation does not have memory issues as reported by valgrind. To run valgrind, use the command: `valgrind ./hw6.test`. This will run the tool over the unit tests, and will report any memory leaks or other memory issues in your implementation.

6. Run the performance tests via the `hw6.perf` executable. As in HW-6, you will need to redirect the output of the tests to a file `output.dat`. Once generated, use the provided gnuplot script to create performance graphs from your results.

7. Create an assignment write up that includes the performance graphs and an explanation of the results reported in the graphs. Focus on explaining the differences between the implementations and give reasonable rationale explaining why the differences exist. In addition, fill out the following table regarding the worst-case asymptotic complexity, using $\textit{Big-O}$ notation, for each of the functions listed below for each Map implementation.
Map Implementation

<table>
<thead>
<tr>
<th>Operation</th>
<th>ArrayMap</th>
<th>LinkedMap</th>
<th>BinSearchMap</th>
<th>HashMap</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>erase</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>contains</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>find_keys</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sorted_keys</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Finally, your write up should also contain a brief paragraph on any implementation issues and/or challenges you ran into and how you addressed them (if applicable).

3 Additional Details and Requirements

Be sure to review the lecture notes regarding the Hash Table implementation for HW-6. Additional information is provided below regarding suggestions and requirements.

**Private member variables.** The underlying table is represented as a resizable array of linked lists. You must implement the resizable array portion and linked list portion from scratch (i.e., you are not permitted to directly use `ArraySeq` and `LinkedSeq` to implement the table). The linked list nodes have the structure:

```cpp
// the chain (linked list) nodes
struct Node {
    K key;
    V value;
    Node* next;
};
```

The table has a default capacity of 16 elements and is defined as:

```cpp
// array of linked lists
Node** table = new Node*[capacity];
```

This means an “empty” hash map must still have a capacity of 16 elements. Thus, your move operations must leave the “right hand” side object in its default state (with space for 16 elements). The `make_empty()` function should still completely delete (and set to `nullptr`) the table (also setting the capacity to 0). To work correctly, the table must be initialized such that each linked-list chain is set to `nullptr` (i.e., each `table[i] == nullptr`). This initialization should be performed using the private `init_table` function described below. The additional member variables (`count`, `capacity`, and `load_factor_threshold`) were further explained in class.

**Private helper functions.** You must implement and use the following private functions in your implementation:

```cpp
// the hash function
int hash(const K& key) const;
```
void resize_and_rehash();
void init_table();
void make_empty();

The hash function takes a key and returns the corresponding table index as described in class. The resize and rehash function, also described in class, is similar to ArraySeq's resize function but checks the load factor against the threshold (set to 75%). The load factor is defined as the count divided by the capacity. Note that you must convert count or capacity to a double to avoid integer division (e.g., \(\frac{\text{count} \times 1.0}{\text{capacity}}\)). The init_table function should set each table entry to nullptr. This function will be used in a number of places. Finally, the make_empty function should delete each chain along with the entire table. It should also set the count and capacity to zero.

**Statistics functions.** You must implement the following three public “statistics” functions:

```c
int min_chain_length();
int max_chain_length();
double avg_chain_length();
```

These should compute the minimum chain length (of table entries with at least one chain), the maximum chain length, and the average chain length. Note that the minimum chain length must be 1 or larger, unless the hash table is empty. To compute the average chain length, you will need to check for an empty hash table so you do not divide by zero when computing the average. The values of these functions are reported in the performance test data and plotted with the gnuplot script.

**Changes for performance tests.** In HashMap, your sorted_keys function should use merge_sort over the temporary key array sequence as opposed to quick_sort. Similarly, you should modify your ArrayMap implementation to also use merge_sort. This should improve the overall performance of sorted_keys, which will decrease the time it takes to run the performance tests and provide better test results. With these changes (and with correct implementations of map), the performance tests should take approximately 40 seconds (give or take) to run.

**Extra credit.** For (up to 10 points) of extra credit, experiment with different load factor threshold values (which can easily be modified by changing the load factor threshold and recompiling your code). For each threshold value, rerun the unit and performance tests, and produce new graphs. Analyze the performance changes along with the chain statistics. Your goal is to see if you can find a threshold value that improves overall performance of the hash map implementation (comparing how the threshold values affect performance of the five main map operations). Include your analysis as a separate PDF file (clearly named) with your submission. Your analysis should state what threshold values you tried, the threshold value that performed the best (outside of the default value of 0.75), and a general description of what happens to performance as the threshold values change. You
only need to provide graphs in your write up for the threshold value that did the best (outside of the default of 0.75).