Github:
https://github.com/ddbl/FLCD

Github lab 1 PR:
https://github.com/ddbl/FLCD/pull/1

Symbol table – hash table implementation

Our hash table will need a few fields to keep it together. It needs a `size`, which will be the number of elements that have been inserted. It needs a `capacity`, which will determine the size of our internal arrays. Last, it needs `table` - this is a collection of arrays, storing each inserted value in a list based on the provided value.

Node

Because the hash table uses **separate chaining**, each list will actually contain a LinkedList of nodes containing the objects stored at that index. This is one method of **collision resolution.**

Collisions

Whenever two keys have the same hash value, it is considered a collision. With separate chaining, we create a Linked List at each index of our `buckets` array, containing all keys for a given index. When we need to look up one of those items, we iterate the list until we find the Node matching the requested key.

Insert

To insert a key/value pair into our hash table, we will follow these steps:

1. Increment size of hash table.
2. Compute `index` of key using hash function.
3. If the bucket at `index` is empty, create a new node and add it there.
4. Otherwise, a collision occurred: there is already a linked list of at least one node at this index. Iterate to the end of the list and add a new node there.

Find

After storing data in our hash table, we will surely need to retrieve it at some point. To do this, we'll perform the following steps:

1. Compute the `index` for the provided key using the hash function.
2. Go to the bucket for that `index`.
3. Iterate the nodes in that linked list until the key is found, or the end of the list is reached.
4. Return the value of the found node, or None if not found.