Introduction

The goals of this lab are:

- To get you familiar with how hash tables work
- To give you practice working with chaining and probing for hash tables

Don't hesitate to make use of other resources for this lab, including the course notes, your TAs, instructor, or other students.

This lab will be focused more on the concepts of a hash table as opposed to the code of it. If you'd like, you may try to implement insert() and get_value() methods for hash tables that follow the rules in this lab.

Hash tables with Chaining

Recall from lecture that chaining is when we have buckets at each index instead of storing a single element at each index of our hash table.

Suppose we have a hash table with a size of 2 (i.e. indexes 0 ~ 1). Suppose we have the following hash values:

- hash("a") == 0
- hash("b") == 1
- hash("c") == 2
- hash("d") == 4
- hash("e") == 5
- hash("f") == 9

Suppose we tried to add to following (key, value) pairs to our hash table -- assuming our hash table never changes in size:

- ("a", "apple")
- ("b", "banana")
- ("c", "cat")
- ("d", "dog")
- ("e", "egg")
- ("f", "flamingo")

What would our hash table look like?

How many items would we have to look through if we wanted to get the value corresponding to the key "f"?

Suppose we have a hash table that doubles in size whenever a bucket contains > size of the hash table // 2 items. For example, our hash table has a size of 2 initially. Once a bucket contains more than 1 (2 // 2) item in it, we make a new hash table with a size of 4 and re-insert
everything. Afterwards, once a bucket contains > 2 items, we would repeat this process. Make the same additions from above to this hash table. How does it change?

How many items would we have to look through if we wanted to get the value corresponding to the key "f"?

In the worst case scenario, all keys we try to insert would have the same index. If all of the keys mentioned before hashed to the index 0, what would the hash table look like? What's the worst-case runtime of getting a value from this hash table?

(In practice, we use hash functions that are well-distributed, so the worst case scenario would be extremely rare. Don't let the worst-case scenario scare you away from thinking that hash tables are bad because their worst-case runtime is inefficient!)

**Hash tables with Probing**

Recall from lecture that probing is when we have only a single (key, value) pair at each index, and if the index we're checking has a value, we continue to another index based on some probing rule.

Suppose we have a hash table with a size of 2 (i.e. indexes 0 ~ 1). Suppose we have the following hash values:

- hash("a") == 0
- hash("b") == 1
- hash("c") == 2
- hash("d") == 4
- hash("e") == 5
- hash("f") == 9

Suppose our probing rule is as follows: We check the index, and if it's taken, we use a second hash function, the results of which are as follows:

- hash("a") == 4
- hash("b") == 3
- hash("c") == 7
- hash("d") == 5
- hash("e") == 1
- hash("f") == 0

If the position at that hash value is also taken, then we add 1 to that index. We continue adding 1 until we run out of indexes to check (i.e. the index == the number of spots in our hash table), at which point, we double the size of our hash table, re-insert everything, and try this process again.

Suppose we tried to add to following (key, value) pairs to our hash table:

- ("a", "apple")
- ("b", "banana")
- ("c", "cat")
What would our hash table look like?

How many items would we have to look through if we wanted to get the value corresponding to the key "f"?

In the worst case scenario, all of our keys would hash to the same index. Suppose they all mapped to index 0 (for both our probing rule, and our original hash values). What's the worst-case runtime of getting a value from this hash table? If our probing rule's hash values didn't change, is the run-time better?