Hashtables

1 Hashtables

Say we wanted to store items in an array like structure, where our key has some correlation with it's index; however, what if our key is not an integer, say a String, or Animal? Or what if we have integers with huge differences like 1 and 100000000. The first case would be impossible to achieve in an organized and efficient fashion in an array because we would have no idea where keys would be- what index would an Armadillo be at? The second case is possible; however, the memory that would be taken up would be enormous, and it would be unfeasible to use so much space for only 2 items. These above 2 problems are the motivation behind hashing.

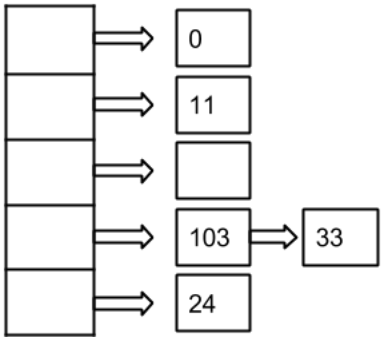
Hashing is the process of creating a number that represents some key. Usually the hashed representation of a key has to do something with an attribute or set of attributes of the object that is being hashed. It is not always trivial to make a good hashing function; however, given enough thought, we can hash names, numbers, food- the possibilities are endless!

Usually, ***hashmaps*** are represented by arrays, which means that when we hash some key, we should get some index in the array- we will refer to the amount of indices in the array to be buckets. Let's start with a basic example where our key is some integer and M is the amount of buckets we currently have in our Hashtable. [1]

1 `key%M`

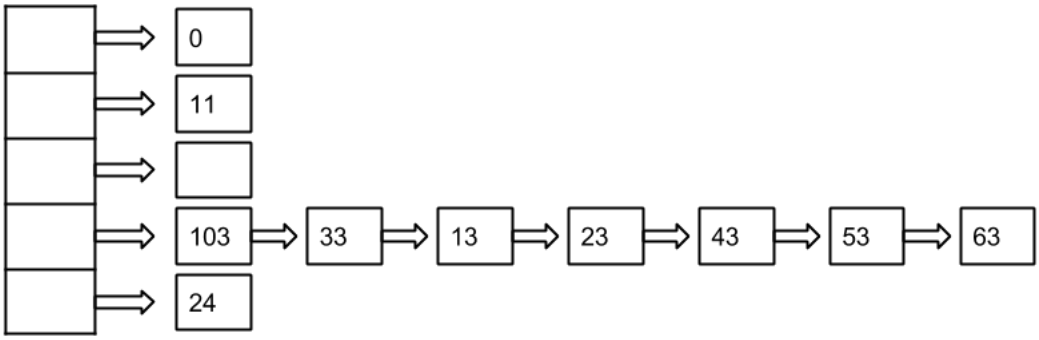
This hash function, like all hash functions, will return some integer between 0 and M - 1. We can use the number returned by this function to map a particular key to some index in the array. Let's try it out on an array of size 5, so M =5, with the calls in the following order: 0,11, 24, 103, 33.

At this point, we have inserted 0, 11, 23, and 104. But wait there's a problem now, 33 and 103 hash to the same index in the array. How can we fix this? Well instead of storing 1 element within the array, we can instead store some data structure, such as linkedlist at each index, this will allow us to store more than 1 element per index.



[1] Note usually in hashing functions, we perform some arithmetic on the key before moding it, so that our encryption is more secure.

Now that we can take care of more than 1 element, we should be good to go right? Let's try inserting the following keys: 13, 23, 43,53,63.



Do you see the problem? If we keep inserting items that have a remainder of 3 when divided by 5, our runtime to get or put an item would be no better than a linkedlist, we would just be using more memory because we have an array too. In order to fix this, while inserting items, we should have some clause that if $\frac{N}{M} \geq$ some number, where N is the total amount of items we have and M is the amount of buckets, we resize our array. This basically means if the average amount of items per bucket is greater than or equal to some number we resize the array. After resizing our array, we rehash all of our items since the amount of buckets has changed. Then , we should get a more even distribution. The question now is how should we increase our buckets? Let's consider the following options:

```
1 M = M * 2
2 M = M + 1000
```

When considering which resizing factor we should choose, we want to make sure that we do not need to resize too frequently because resizing is a relatively expensive operation. At first glance $M = M + 1000$ may seem tempting; however, it is important to realize that the number of buckets being added is not increasing with N as it grows larger. This means that, in the long run, the 1000 will not be substantial enough to make the resizing factor negligible. The best resizing option we have is $M = M * 2$ since the amount of buckets that we add at each resize operation grows with the amount of buckets we currently have.

Let's analyze the runtime of Hashmaps now. First let's consider the average case for all these functions. On average, we have a hashmap with $O(N/M) = O(L)$ items per bucket. This means that, on average, we a Put or Get operation will take O(L) time where L is the load factor. This is because the amount of items we would need to look through would be upper bounded by O(L) as we would only look through 1 bucket.

Let's now consider the the amortized case. To do this, we will revist the average case. If we can ensure that the average amount of items per bucket, or L, is small, essentially constant, then we would only have to do a constant amount of work to find any given item. This means that Get and Put each take $\Theta(1)$ amortized time.

Now let's discuss the worst case runtime for Hashmaps. In a worst case for Put, we would need to resize the array and rehash all the items, which would take $\Theta(N)$ time. There is also another case for which worst case runtime can occur. If we have a bad hashing function, it could be exploited so that, with a series of inserts, everything hashes to the same bucket, essentially making one bucket act like a huge LinkedList while the other buckets have nearly nothing in then. This would make both Get and Put take $O(N)$ time as we would need to check the entire bucket to see if an element is already in the hashmap. This is why it is imperative that a good hashing function is used in a hashmap.

To use hashmaps, we need to use the following functions.

```
1 public int hashcode(){
2     ......
3     return some hashcode;
4     }
```

```
5 public boolean equals(Object obj){
6     ....
7     return if obj equals your current object based off a factor of your choosing
8 }
```

One key idea to remember when analyzing hashing functions is that any 2 items that are equal should hash to same bucket. This means that if *a.equals(b)* then *a.hashcode()* must equal *b.hashcode()*. The implication of this is that if you override the equals method, you must also overwrite the hashcode method. Another important characteristic of hash functions is that they should provide a relatively even distribution meaning that one bucket should not be hashed to a disproportionate amount compared to other buckets.