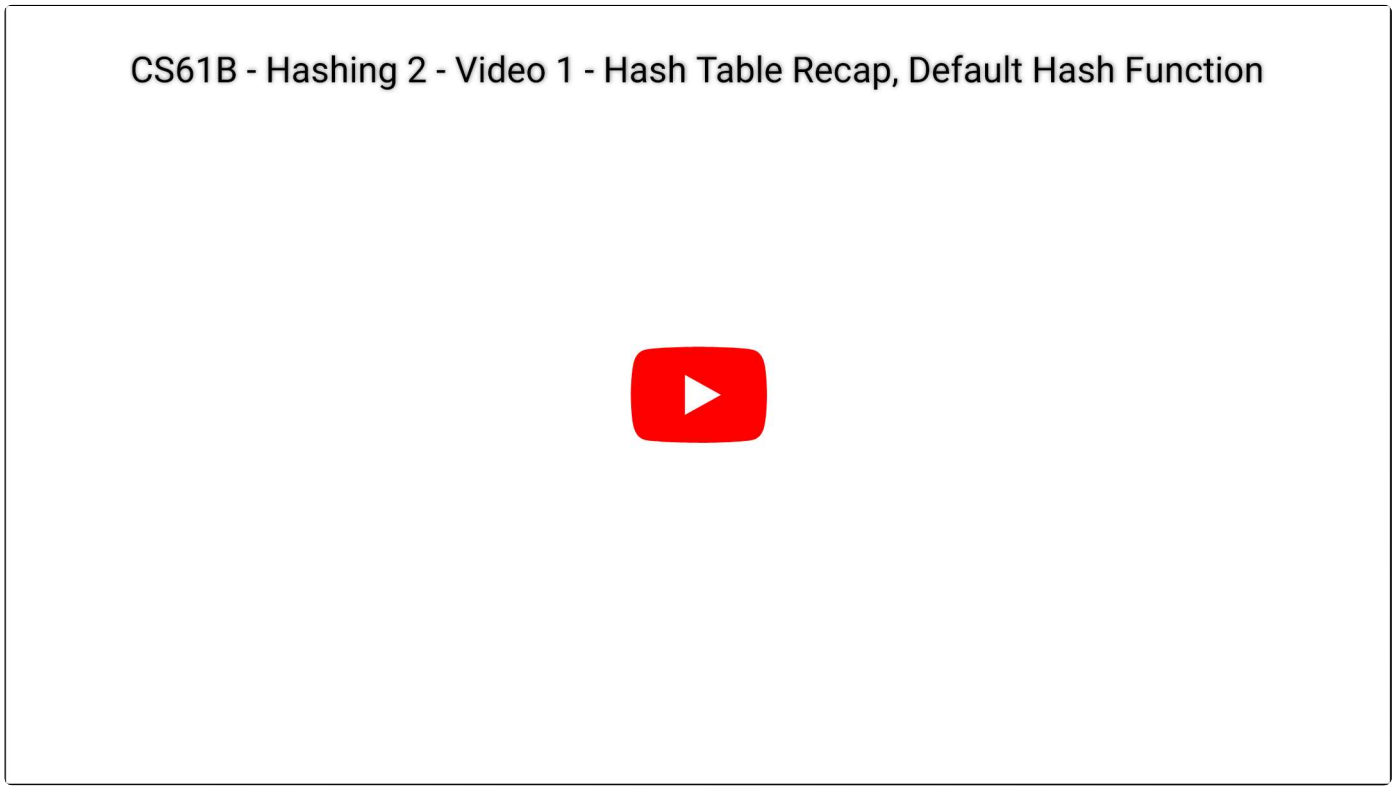
20.1 Hash Table Recap, Default Hash Function

"The whole point is that we have a bunch of lists that are all short." - Professor Hug.



Hash Table Recap, Default Hash Function

Let's continue understanding Hashing. We've now seen implementations for sets and maps.

1. Red-Black Based Tree Approach: TreeSet/TreeMap
   - requires the items to be comparable (the notion of less or greater than)
   - logarithmic time complexity
2. HashTable based approach: HashSet/HashMap
   - constant time operations if the hashCode spreads the item nicely (few collisions)

Recall that with a hash table, the idea is that for any piece of data, like a String (or any other type of object) we want to store in our hash table, we need to turn this into a number

called a hash code. So a string like "Mihir" can be converted to -2101281024.

> Fun Fact: And that integer is anything between about negative two billion and about two billion, the space of all Java Integers. This range yields about 4 billion integers or 2^32 integers. If you are interested in why this is the case, please take CS 61C!

Once we have our number, we want to convert this number to our bucket number (i.e. which of the many linked lists I want to add this new entry to). In the first example, we will use the `Math.floorMod(x, 4)`, since the length of the underlying buckets array has length 4.

If we take the converted hash code for "Mihir", -2101281024, and then mod this by 4, we get 0. This reduces our hash code, -2101281024 to a valid index, 0. This means that we would use the 0th bucket to place our data, "Mihir", in our LinkedList at the 0th bucket.

> You can essentially think of Java HashTables as just an array of LinkedLists categorized under bucket labels and hopefully have better performance by utilizing these LinkedLists.

But how many LinkedLists/Buckets should we use? This is an important question, because as we insert more and more items into our buckets, the length of the LinkedLists will inevitably grow, which in turn compromises the runtime for the hash table's operations.

For example, let's say we inserted strings like "Mihir", "Mirchandani", "loves", and "61B", all of which yielded hash codes that were divisible by 4. In this case, all strings would be put into the 0th index bucket and all strings would be inserted into the same LinkedList. What happens to the runtime for a search operation? Well, we would have to search through an entire LinkedList to look up our data! This runs in linear time and is too slow for HashMap's famous title of holding fast lookup times.

Last time, we saw that we can have a variable number of LinkedLists. The idea is that we resize that array of linked lists whenever the load factor (N/M, where N is the number of elements in our table and M is the number of buckets) exceeds some constant. Java picks 0.75 as we'll see later. This prevents collisions from happening too frequently. So long as our items are spread out nicely, between the buckets, the LinkedLists at each bucket for the most part have a very small size, which means we'll on average get constant run time!

Example: If the HashTable has load factor 3, and our `hashCode()` function spreads out the entries evenly, we are going to end up with just 3 items per bucket, and our search operation takes $\Theta(1)$time.

Comparing Data Structure Run Times!

	contains(x)	add(x)
Bushy BSTs	Θ(log N)	Θ(log N)
Separate Chaining Hash Table with NO resizing	Θ(N)	Θ(N)
Separate Chaining Hash Table with resizing	Θ(1)	Θ(1)

Previous
20. Hashing II

Next
20.2 Distribution By Other Hash Functions

Last updated 1 year ago