# Hashing

## Overview

Hashing is the concept of sending objects through a function (hash function) that will spit out a hashcode for that object which will be used to make data storage and accessibility faster. Here is a picture of how you can imagine the hashing works.

$$\begin{tikzcd}\text{Object} \arrow[r] & \boxed{\text{Hash Function}} \arrow[r] & \text{Hash Code}\end{tikzcd}$$

## Arrays and Linked Lists Review

Recall Arrays and Linked Lists and the runtimes of their functions. Knowing their runtimes will help you understand the ways we can use hashing to create a faster data structure. Here is a table for reference.

| Function | Array | Linked List |
|---|---|---|
| Find | $\Theta(1)$ | $\Theta(n)$ |
| Insert (front) | $\Theta(n)$ | $\Theta(1)$ |
| Insert (back) | $\Theta(1)$ | $\Theta(n)$ |
| Remove (front) | $\Theta(n)$ | $\Theta(1)$ |
| Remove (back) | $\Theta(1)$ | $\Theta(n)$ |
| Contains | $\Theta(n)$ | $\Theta(n)$ |

**Arrays: Find**
If you want to look for an item at index $i$ in an array, you can simply find with by indexing into the array at that specific index in constant time $\implies \Theta(1)$.

**Linked List: Find**
If you want to look for an item at index $i$ in a linked list, you would need to, worst case, traverse the entire linked list if the index of the item you want is the last item in the linked list $\implies \Theta(n)$.

**Array: Insert Front**
If you want to insert at the front of an array, you would have to shift all the other elements right one position $\implies \Theta(n)$.

**Linked List: Insert Front**
If you want to insert at the front of a linked list, you could create a new node and set it's tail to the rest of the linked list $\implies \Theta(1)$.

**Array: Insert Back**
If you want to insert at the back of an array, you could simply insert at the end without shifting any of the other elements $\implies \Theta(1)$.

**Linked List: Insert Back**
If you want to insert at the back of an linked list, you would have to traverse the entire linked list and then add the new value $\implies \Theta(n)$ (The actual insertion takes $\Theta(1)$, but the traversing takes $\Theta(n)$).

**Array: Remove Front**
If you want to remove from the beginning of an array, you would have to shift all the other elements left one space $\implies \Theta(n)$.

**Linked List: Remove Front**
If you want to remove from the beginning of a linked list, you could just move the pointer of that linked list

to it's tail $\implies \Theta(1)$.

**Array: Remove Back**
If you want to remove from the back of an array, you can simply remove the last element in the array $\implies \Theta(1)$.

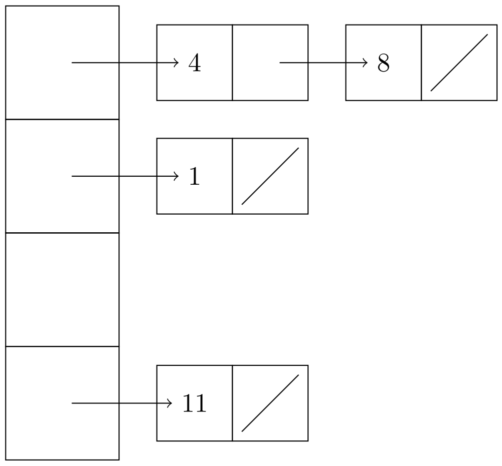
**Linked List: Remove Back**
If you want to remove from the back of a linked list, you would have to traverse the entire linked list and then set the tail of the element before to null $\implies \Theta(n)$.

**Array: Contains**
If you want to know if an element exists in an array, you would have to look through the entire array to check for the value $\implies \Theta(n)$.

**Linked List: Contains**
If you want to know if an element exists in an array, you would have to look through the entire linked list to check for the value $\implies \Theta(n)$.

## HashTables

If you look at the table of runtimes, you will see that there are specific functions that one data structure is better than the other. To take the best of both worlds, we combine arrays and linked lists to create a HashTable. A HashTable is essentially an array of linked lists. Here's a picture for reference.



You can see that the left column is an array of length 4 and each entry in the array (bucket) is actually a linked list that is what holds our data. This is what we call a HashTable. It takes the advantages of arrays, indexing into a bucket in constant time, and the advantages of linked lists, inserting and removing in constant time (disregarding finding the spot to insert/remove).

If the hash function just returned the inputted number, we could get a HashTable that looks like the one above. Our HashTable takes the hash code from the hash function and will *mod*(%) it by the length of our array (4) and whatever is left is which bucket that specific item belongs to.

**Insertion**
When we insert into our HashTable, we run the object through the hash function and get a hash code for that object. We then *mod* this number by the length of our array and insert that object to the end of the linked list at that bucket.

**Find**
To find an object in our HashTable, we run it through the hash function and get a hash code. We then *mod* this number by the length of our array and look in that bucket for it.

**Remove**
To remove an object from our HashTable, we run it through the hash function, get the hash code, *mod* the hash code by the length of our array, and remove it from that bucket's linked list.

## Hash Functions/Hash Codes

The hash functions we use greatly determine the performance of our hash data structures. We want a hash function that will do it's best to evenly distribute objects into the different buckets. This is because the more evenly our buckets are distributed, the less traversing down a list we would need to do in the worst case.

**Valid Hash Codes**
The hash function needs to return a valid hash code. What this essentially means is that if an object is *.equals*() another object, then they are the same and should have the same hash code when they pass the has function. Valid hash codes are not necessarily *good* hash codes, but they are valid ones that allow us to use these hash data structures. Invalid hash codes make it impossible to guarantee the success of specific actions in our hash data structures.

## Other Hash Data Structures

Other hash data structures build off of the HashTable concept for their implementation.

**HashSet**
A HashSet is implemented exactly the same as a HashTable, except it cannot contain duplicates, so if it sees a duplicate, it will not add that value. All elements in a Set are **unique**.

**HashMap**
A HashMap is also implemented in the same way, but instead of storing just values, we are storing (key, value) pairs. We use the **key** to determine the hash code and our keys **MUST** be unique. The values can be anything and **CAN** be duplicates. Other than these, it works the same as our original HashTable.

***NOTE:** sometimes the terms HashMap and HashTable are use interchangeably when the HashTable deals with (key, value) pairs.

## TL;DR

1. Objects pass through a hash function that returns that object's hash code
(Object $\implies$ Hash Function $\implies$ Hash Code)
2. HashTables are arrays of linked lists (best of both worlds)
3. The hash code is used to determine which bucket said object belongs to (*mod* length of array)
4. Valid hash codes are ones where if an object is *.equals*() another object they have the same hash code
5. HashSets cannot have duplicates
6. HashMaps store (key, value) pairs and keys have to be unique
7. Check out the Oracle Docs for functionalities supported by these hashing data structures (google "Java [name of data structure]" and click the link that says "Oracle Docs")