**Hash Tables**

Hash tables are Objects that contain a key/value pair.

key = nails

value = 1000

{

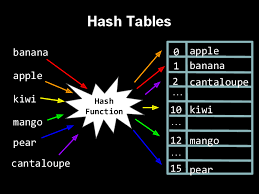
    nails: 1000

}

When we hash this Object, only the key will be hashed (nails) and placed into a table.

Example: when we hash nails, it would give us an address of 2 which is where we will place that object in our memory.

If we wanted to look up the value of nails, we could go directly to the address of 2 which is very efficient.



Hash tables are one way only meaning that we can hash nails and get the number 2 but we cannot hash 2 to get nails.

Hash tables are also deterministic meaning that when we hash nails and get 2, the next time we hash nails, it will produce 2 as well.

set('nails', 1000)

get('nails')

Since we are hashing the same string ‘nails’, it does not matter what method we are using, either set, get, etc. we will always hash them into the same address which is in this example ‘2’.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Collisions**

Collisions are when there is more than one item mapped to the same address in memory.

**Separate Chaining**

This is when we have more than one item in an address space.

**Linked List in Separate Chaining** – we can implement a linked list in the address space so that more than one item can be mapped to the address.

**Linear Probing (type of Open Addressing)**

When the item is mapped to an address space and the space is already occupied, then it will move to the next address space until it finds a space that is empty then occupies it.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Prime numbers are used for optimal randomized distribution of the items you place in hash tables.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Big O (Hash Tables)**

**O(1)**

Hash tables with linked list:

When we set an item in the table, we are going to push() it to the end of the list which results in an O(1).

When we get() an item from the table, we need to iterate through the linked list which could be an O(1) for the best case and O(n) for the worst case because we would have to iterate through the entire list to get an item that’s at the end of the the list.

However, hash tables are able to utilize any address spaces and it is very rare that there would be a collision (meaning that it is rare to have a linked list that is very long) therefore the big O of hash tables is O(1).

Collisions are rare because the \_hash() function is very efficient at randomizing address spaces so most of the time, there would only be 1 item in an address space which allows us to go to that space very quickly resulting in O(1).

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

class HashTable {

    constructor(size = 7) {

        this.dataMap = new Array(size);

    }

    \_hash(key) {

        let hash = 0;

        for (let i = 0; i < key.length; i++) {

            hash = (hash + key.charCodeAt(i) \* 23) % this.dataMap.length;

        }

        return hash;

    }

    printTable() {

        for (let i = 0; i < this.dataMap.length; i++) {

            console.log(i, ": ", this.dataMap[i]);

        }

    }

    set(key, value) {

        let index = this.\_hash(key);

        if (!this.dataMap[index]) this.dataMap[index] = [];

        this.dataMap[index].push([key, value]);

        return this;

    }

    get(key) {

        let index = this.\_hash(key);

        if (this.dataMap[index]) {

            for (let i = 0; i < this.dataMap[index].length; i++) {

                if (this.dataMap[index][i][0] === key) {

                    return this.dataMap[index][i][1];

                }

            }

        }

        return undefined;

    }

    keys() {

        let allKeys = [];

        for (let i = 0; i < this.dataMap.length; i++) {

            if (this.dataMap[i]) {

                for (let j = 0; j < this.dataMap[i].length; j++) {

                    allKeys.push(this.dataMap[i][j][0]);

                }

            }

        }

        return allKeys;

    }

}

Interview 1

function itemInCommon(arr1, arr2) {

    for (let i = 0; i < arr1.length; i++) {

        for (let j = 0; j < arr2.length; j++) {

            if (arr1[i] === arr2[j]) return true

        }

    }

    return false

}

**Interview 2**

function itemInCommon(arr1, arr2) {

  let obj = {}

  for (let i = 0; i < arr1.length; i++) {

    obj[arr1[i]] = true

  }

  for (let j = 0; j < arr2.length; j++) {

    if (obj[arr2[j]]) return true

  }

  return false

}

**Set Method**

///  WRITE SET METHOD HERE  ///

set(key, value) {

    // \_hash() the key to get an index value of where to store the data in memory

    let index = this.\_hash(key);

    // If there is no data in the address, create a new empty array[] to place the data in.

    if (!this.dataMap[index]) {

        this.dataMap[index] = [];

    }

    // Othervise, push() the new [key, value] pair.

    this.dataMap[index].push([key, value]);

    // Return the hash table.

    return this;

}

**Get Method**

///  WRITE GET METHOD HERE  ///

get(key) {

    // \_hash() the key to get an index

    let index = this.\_hash(key);

    // If there is an item at the given index, then map over each [key, value] pair in that index

    if (this.dataMap[index]) {

        for (let i = 0; i < this.dataMap[index].length; i++) {

            // If the key in the index matches the key we are looking for,

            // return that key

            if (this.dataMap[index][i][0] === key) {

                return this.dataMap[index][i][1];

            }

        }

    }

    // If the key is not found, return undefined

    return undefined;

}

**Keys**

///  WRITE KEYS METHOD HERE  ///

keys() {

    // Create an empty array[] that will return all of our KEYS

    let allKeys = [];

    // Map through the entire hash HashTable

    for (let i = 0; i < this.dataMap.length; i++) {

        // If an address contains data:

        if (this.data[i]) {

            // Map over each item in that memory location

            for (let j = 0; j < this.dataMap[i].length; j++) {

                // push() the keys from each [key, value] pairs into the allKeys[] array

                allKeys.push(this.dataMap[i][j][0]);

            }

        }

    }

    // Return the allkeys[] array

    return allkeys;

}