HashMaps

A real world example: I stay in a hotel for a few days, I ask the desk clerk if there are any messages for me. Behind his back is a dovecot-like cupboard, with 26 entries, labeled A to Z. Because he knows my last name, he goes to the slot labeled W, and takes out three letters. One is for Robby Williams, one is for Jimmy Webb, and one is for me.

The clerk only had to inspect three letters. How many letters would he have to inspect if there would have been only one letter box

Maps, dictionaries, and associative arrays all describe the same abstract data type. But hash map implementations are distinct from treemap implementations in that one uses a hash table and one uses a binary search tree.

Hashtable is a data structure that **maps** keys to values

Going back to the drawer analogy, bins have a label rather than a number.

HashMap is like a drawer that stores things on bins and labels them

clothes
toys
books

In this example, if you are looking for the book, you don't have to open bin 1, 2, and 3. You go directly to the container labeled as "books". That's a huge gain! Search time goes from *O(n)* to *O(1)*.

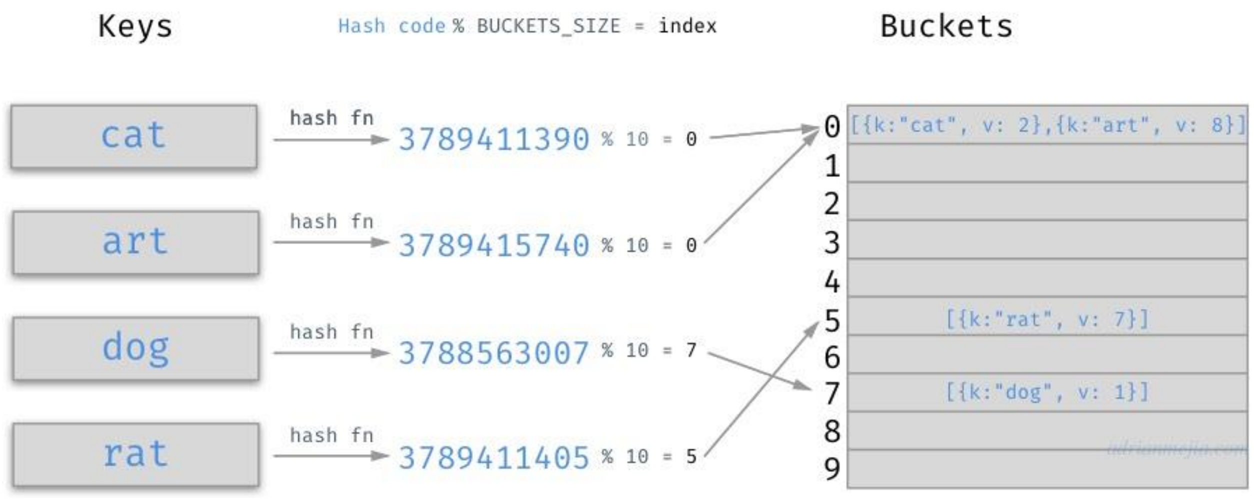
In arrays, the data is referenced using a numeric index (relatively to the position). However, HashMaps uses labels that could be a string, number, Object, or anything. Internally, the HashMap uses an Array, and it maps the labels to array indexes using a *hash function*.

There are at least two ways to implement hashmap:

1. **Array**: Using a hash function to map a key to the array index value. Worst: `O(n)`, Average: `O(1)`
2. **Binary Search Tree**: using a self-balancing binary search tree to look up for values (more on this later). Worst: *`O(log n)`*, Average: *`O(log n)`*.

We will cover Trees & Binary Search Trees, so don't worry about it for now. The most common implementation of Maps is using an **array** and `hash` function. So, that's the one we are going to focus on.

HashMap implemented with an array

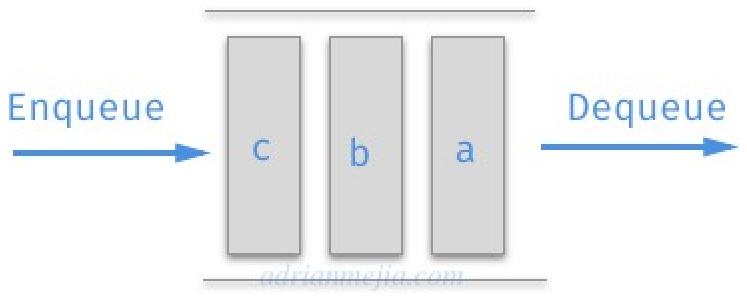


As you can see in the image, each key gets translated into a **hash code**. Since the array size is limited (e.g., 10), we have to loop through the available buckets using the modulus function. In the buckets, we store the key/value pair, and if there's more than one, we use a collection to hold them.

Now, What do you think about covering each of the HashMap components in detail? Let's start with the **hash function**.

Queues

Queues are a data structure where the first data to get in is also the first to go out. A.k.a First-in, First-out (FIFO). It’s like a line of people at the movies, the first to come in is the first to come out.



We could implement a Queue using an array, very similar to how we implemented the Stack.

Queue implemented with Array(s)

A naive implementation would be this one using `Array.push` and `Array.shift`:

```
class Queue {
  constructor() {
    this.input = [];
  }

  add(element) {
    this.input.push(element);
  }

  remove() {
    return this.input.shift();
  }
}
```

What’s the time complexity of `Queue.add` and `Queue.remove`?

- `Queue.add` uses `array.push` which has a constant runtime. Win!

- `Queue.remove` uses `array.shift` which has a linear runtime. Can we do better than `O(n)`?

Think of how you can implement a Queue only using `Array.push` and `Array.pop`.

```
class Queue {
  constructor() {
    this.input = [];
    this.output = [];
  }

  add(element) {
    this.input.push(element);
  }

  remove() {
    if(!this.output.length) {
      while(this.input.length) {
        this.output.push(this.input.pop());
      }
    }
    return this.output.pop();
  }
}
```

Now we are using two arrays rather than one.

```
const queue = new Queue();

queue.add('a');
queue.add('b');

queue.remove() // a
queue.add('c');
queue.remove() // b
queue.remove() // c
```

When we remove something for the first time, the `output` array is empty. So, we insert the content of `input` backward like `['b', 'a']`. Then we pop elements from the `output` array. As you can see, using this trick, we get the output in the same order of insertion (FIFO).

What's the runtime?

If the output already has some elements, then the remove operation is constant *`O(1)`*. When the output arrays need to get refilled, it takes *`O(n)`* to do so. After the refilled, every operation would be constant again. The amortized time is *`O(1)`*.

We can achieve a `Queue` with a pure constant if we use LinkedList. Let's see what it is in the next section!

Queue implemented with a Doubly Linked List

We can achieve the best performance for a `queue` using a linked list rather than an array.

```
const LinkedList = require('../linked-lists/linked-list');

class Queue {
  constructor() {
    this.input = new LinkedList();
  }

  add(element) {
    this.input.addFirst(element);
  }

  remove() {
    return this.input.removeLast();
  }

  get size() {
    return this.input.size;
  }
}
```

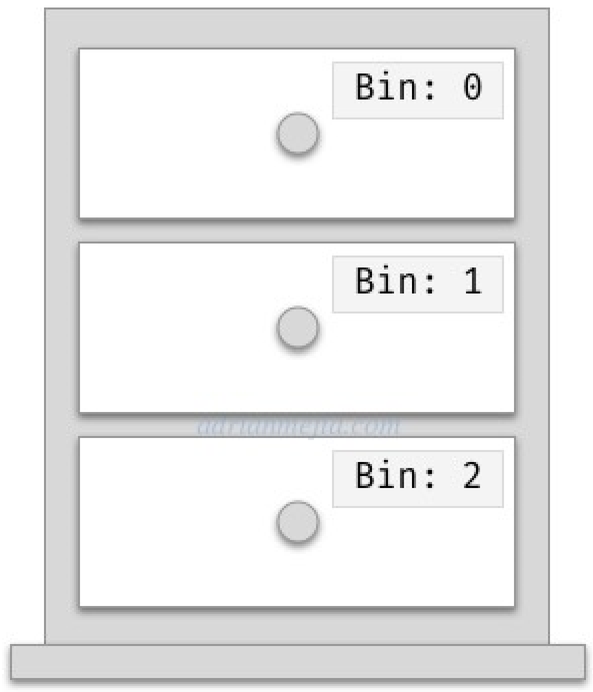
Using a doubly-linked list with the last element reference, we achieve an `add` of *O(1)*. That's the importance of using the right tool for the right job

Array

Arrays are collections of zero or more elements. Arrays are one of the most used data structures because of their simplicity and fast way of retrieving information.

You can think of an array as a drawer where you can store things in the bins.

Array is like a drawer that stores things on bins



When you want to search for something, you can go directly to the bin number. That's a constant time operation (`O(1)`). However, if you forgot what cabinet had, you will have to open one by one (`O(n)`) to verify its content until you find what you are looking for. That same happens with an array.

Depending on the programming language, arrays have some differences. For some dynamic languages like JavaScript and Ruby, an array can contain different data types: numbers, strings, words, objects, and even functions. In typed languages like Java/C/C++, you have to predefine the Array size and the data type. In JavaScript, it would automatically increase the size of the Array when needed.

Arrays built-in operations

Depending on the programming language, the implementation would be slightly different.

For instance, in JavaScript, we can accomplish append to end with `push` and append to the beginning with `unshift`. But also, we have `pop` and `shift` to remove from an array. Let's describe some everyday operations that we are going to use through this post.

Common JS Array built-in functions

Function	Runtime	Description
array.push	O(1)	Insert element to the end of the array
array.pop	O(1)	Remove element to the end of the array
array.shift	O(n)	Remove element to the beginning of the array
array.unshift	O(n)	Insert element(s) to the beginning of the array
array.slice	O(n)	Returns a copy of the array from `beginning` to `end`.

Function	Runtime	Description
array.splice	O(n)	Changes (add/remove) the array

Insert element on an array

There are multiple ways to insert elements into an array. You can append new data to the end or add it to the beginning of the collection.

Let’s start with append to tail:

```
function insertToTail(array, element) {
  array.push(element);
  return array;
}

const array = [1, 2, 3];
console.log(insertToTail(array, 4)); // => [ 1, 2, 3, 4 ]
```

Based on the language specification, push just set the new value at the end of the Array. Thus,

The `Array.push` runtime is a *O(1)*

Let’s now try appending to head:

```
function insertToHead(array, element) {
  array.unshift(element);
  return array;
}

const array = [1, 2, 3];
console.log(insertToHead(array, 0)); // => [ 0, 1, 2, 3 ]
```

What do you think is the runtime of the `insertToHead` function? It looks the same as the previous one, except that we are using `unshift` instead of `push`. But there’s a catch! unshift algorithm makes room for the new element by moving all existing ones to the next position in the Array. So, it will iterate through all the elements.

The `Array.unshift` runtime is an *O(n)*

Access an element in an array

If you know the index for the element that you are looking for, then you can access the element directly like this:

```
function access(array, index) {
  return array[index];
}

const array = [1, 'word', 3.14, {a: 1}];
access(array, 0); // => 1
access(array, 3); // => {a: 1}
```

As you can see in the code above, accessing an element on an array has a constant time:

Array access runtime is *O(1)*

Note: You can also change any value at a given index in constant time.

Search an element in an array

Suppose you don’t know the index of the data that you want from an array. You have to iterate through each element on the Array until we find what we are looking for.

```
function search(array, element) {
  for (let index = 0; index < array.length; index++) {
    if(element === array[index]) {
      return index;
    }
  }
}

const array = [1, 'word', 3.14, {a: 1}];
```

```
console.log(search(array, 'word')); // => 1
console.log(search(array, 3.14)); // => 2
```

Given the for-loop, we have:

Array search runtime is *O(n)*

Deleting elements from an array

What do you think is the running time of deleting an element from an array?

Well, let's think about the different cases:

1. You can delete from the end of the Array, which might be constant time. *O(1)*
2. However, you can also remove it from the beginning or middle of the collection. In that case, you would have to move all the following elements to close the gap. *O(n)*

Talk is cheap. Let's do the code!

```
function remove(array, element) {
  const index = search(array, element);
  array.splice(index, 1);
  return array;
}

const array1 = [0, 1, 2, 3];
console.log(remove(array1, 1)); // => [ 0, 2, 3 ]
```

So we are using our `search` function to find the elements' index *O(n)*. Then we use the JS built-in `splice` function, which has a running time of *O(n)*. What's the total *O(2n)*? Remember, we constants don't matter as much.

We take the worst-case scenario:

Deleting an item from an array is *O(n)*.

Array operations time complexity

We can sum up the arrays time complexity as follows:

Array Time Complexities

Operation	Worst
Access (`Array.[]`)	*O(1)*
Insert head (`Array.unshift`)	*O(n)*
Insert tail (`Array.push`)	*O(1)*
Search (for value)	*O(n)*
Delete (`Array.splice`)	*O(n)*

Quicksort

The quicksort technique is done by separating the list into two parts. Initially, a pivot element is chosen by partitioning algorithm. The left part of the pivot holds the smaller values than the pivot, and right part holds the larger value. After partitioning, each separate lists are partitioned using the same procedure.

The complexity of Quicksort Technique

- Time Complexity: O(n log n) for best case and average case, O(n^2) for the worst case.
- Space Complexity: O(log n)

Input and Output

```
Input:
The unsorted list: 90 45 22 11 22 50
Output:
Array before Sorting: 90 45 22 11 22 50
Array after Sorting: 11 22 22 45 50 90
```

Algorithm

partition(array, lower, upper)

Input: The data set array, lower boundary and upper boundary

Output: Pivot in the correct position

```
Begin
  pivot := array[lower]
  start := lower and end := upper
  while start < end do
    while array[start] <= pivot AND start < end do
      start := start +1
    done

    while array[end] > pivot do
      end := end – 1
    done
    if start < end then
      swap array[start] with array[end]
  done

  array[lower] := array[end]
  array[end] := pivot
  return end
End
```

quickSort(array, left, right

Input − An array of data, and lower and upper bound of the array

Output − The sorted Array

```
Begin
  if lower < right then
     q = partition(arraym left, right)
     quickSort(array, left, q-1)
     quickSort(array, q+1, right)
End
```

Example

```
#include<iostream>
using namespace std;

void swapping(int &a, int &b) { //swap the content of a and b
  int temp;
  temp = a;
  a = b;
  b = temp;
}

void display(int *array, int size) {
  for(int i = 0; i<size; i++)
     cout << array[i] << " ";
  cout << endl;
}

int partition(int *array, int lower, int upper) {
  //Hoare partitioning technique to find correct location for pivot
  int pivot, start, end;
  pivot = array[lower];      //first element as pivot
  start = lower; end = upper;

  while(start < end) {
    while(array[start] <= pivot && start<end) {
      start++;     //start pointer moves to right
    }

    while(array[end] > pivot) {
      end--;     //end pointer moves to left
    }

    if(start < end) {
      swap(array[start], array[end]); //swap smaller and bigger element
    }
  }

  array[lower] = array[end];
  array[end] = pivot;
  return end;
}

void quickSort(int *array, int left, int right) {
  int q;

  if(left < right) {
    q = partition(array, left, right);
    quickSort(array, left, q-1);    //sort left sub-array
    quickSort(array, q+1, right);   //sort right sub-array
  }
}

int main() {
  int n;
  cout << "Enter the number of elements: ";
  cin >> n;
  int arr[n]; //create an array with given number of elements
  cout << "Enter elements:" << endl;

  for(int i = 0; i<n; i++) {
    cin >> arr[i];
  }

  cout << "Array before Sorting: ";
  display(arr, n);
  quickSort(arr, 0, n-1); //(n-1) for last index
  cout << "Array after Sorting: ";
  display(arr, n);
}
```

Output

```
Enter the number of elements: 6
Enter elements:
90 45 22 11 22 50
Array before Sorting: 90 45 22 11 22 50
Array after Sorting: 11 22 22 45 50 90
```

Merge Sort

The merge sort technique is based on divide and conquers technique. We divide the whole dataset into smaller parts and merge them into a larger piece in sorted order. It is also very effective for worst cases because this algorithm has lower time complexity for the worst case also.

The complexity of Merge Sort Technique

- **Time Complexity:** O(n log n) for all cases
- **Space Complexity:** O(n)

Input and Output

```
Input:
The unsorted list: 14 20 78 98 20 45
Output:
Array before Sorting: 14 20 78 98 20 45
Array after Sorting: 14 20 20 45 78 98
```

Algorithm

merge(array, left, middle, right)

Input – The data set array, left, middle and right index

Output – The merged list

```
Begin
  nLeft := m - left+1
  nRight := right – m
  define arrays leftArr and rightArr of size nLeft and nRight respectively

  for i := 0 to nLeft do
    leftArr[i] := array[left +1]
  done

  for j := 0 to nRight do
    rightArr[j] := array[middle + j +1]
  done

  i := 0, j := 0, k := left
  while i < nLeft AND j < nRight do
    if leftArr[i] <= rightArr[j] then
      array[k] = leftArr[i]
      i := i+1
    else
      array[k] = rightArr[j]
      j := j+1
    k := k+1
  done

  while i < nLeft do
    array[k] := leftArr[i]
    i := i+1
    k := k+1
  done

  while j < nRight do
    array[k] := rightArr[j]
    j := j+1
    k := k+1
  done
End
```

mergeSort(array, left, right)

Input − An array of data, and lower and upper bound of the array

Output − The sorted Array

```
Begin
  if lower < right then
    mid := left + (right - left) /2
    mergeSort(array, left, mid)
    mergeSort (array, mid+1, right)
    merge(array, left, mid, right)
End
```

Example

```
#include<iostream>
using namespace std;

void swapping(int &a, int &b) { //swap the content of a and b
  int temp;
  temp = a;
  a = b;
  b = temp;
}

void display(int *array, int size) {
  for(int i = 0; i<size; i++)
    cout << array[i] << " ";
  cout << endl;
}

void merge(int *array, int l, int m, int r) {
  int i, j, k, nl, nr;
  //size of left and right sub-arrays
  nl = m-l+1; nr = r-m;
  int larr[nl], rarr[nr];

  //fill left and right sub-arrays
  for(i = 0; i<nl; i++)
    larr[i] = array[l+i];
  for(j = 0; j<nr; j++)
    rarr[j] = array[m+1+j];

  i = 0; j = 0; k = l;
  //marge temp arrays to real array

  while(i < nl && j<nr) {
    if(larr[i] <= rarr[j]) {
      array[k] = larr[i];
      i++;
    }else{
      array[k] = rarr[j];
      j++;
    }
    k++;
  }

  while(i<nl) {      //extra element in left array
    array[k] = larr[i];
    i++; k++;
  }

  while(j<nr) {     //extra element in right array
    array[k] = rarr[j];
    j++; k++;
  }
}

void mergeSort(int *array, int l, int r) {
  int m;
  if(l < r) {
    int m = l+(r-l)/2;
    // Sort first and second arrays
    mergeSort(array, l, m);
    mergeSort(array, m+1, r);
    merge(array, l, m, r);
  }
}

int main() {
  int n;
  cout << "Enter the number of elements: ";
  cin >> n;
  int arr[n]; //create an array with given number of elements
  cout << "Enter elements:" << endl;

  for(int i = 0; i<n; i++) {
    cin >> arr[i];
  }

  cout << "Array before Sorting: ";
  display(arr, n);
  mergeSort(arr, 0, n-1); //(n-1) for last index
  cout << "Array after Sorting: ";
  display(arr, n);
}
```

Output

```
Enter the number of elements: 6
Enter elements:
14 20 78 98 20 45
Array before Sorting: 14 20 78 98 20 45
Array after Sorting: 14 20 20 45 78 98
```