Data Structures & Algorithms

Hashing-I

B.Bhuvaneswaran, AP (SG) / CSE

9791519152

bhuvaneswaran@rajalakshmi.edu.in

RAJALAKSHMI ENGINEERING COLLEGE

An AUTONOMOUS Institution

Affiliated to ANNA UNIVERSITY, Chennai

Introduction

- In the most basic terms, a data structure is a format for organizing data in an efficient way.
- In practical terms, we can split data structures into two things:
  - the interface and
  - the implementation.
- The interface is like a contract that specifies how we can interact with the data structure - what operations we can perform on it, what inputs it expects, and what outputs we can expect.

Introduction

- The interface is like a contract that specifies how we can interact with the data structure - what operations we can perform on it, what inputs it expects, and what outputs we can expect.

Example

- For example, consider a dynamic array. The interface would include operations like appending, insertion, removal, updating, and more.
- These operations are well-defined and have specific rules that we must follow when we use them.
- If we want to append an element, we use the built-in method like .append() or .push() while passing in the element we want to add as an argument.
- Typically this operation doesn't return anything.

Example

- Now, the implementation is the code that actually makes the data structure work.
- This is where the details of how the data is stored and how the operations are performed come into play.
- For example, the implementation of a dynamic array might involve allocating memory for the list, tracking the size, and rearranging the elements when an operation like remove is called.

Example

- For many data structures, the implementation can be quite complex, involving intricate algorithms and data manipulation.
- However we don't need to worry about those details - we only need to understand the interface and how to use it properly.

Example

- The more important thing is to understand the interface.
- All major data structures have built-in implementations in all major programming languages.

Hash Function

- A hash function is a function that takes an input and deterministically converts it to an integer that is less than a fixed size set by the programmer.
- Inputs are called keys and the same input will always be converted to the same integer.

Example

- Here's an example hash algorithm for a string containing letters of the English alphabet:
  1. Declare an integer total.
  2. Iterate over the string. For each character, convert it to its position in the alphabet. For example, a -> 1, c -> 3, z -> 26.
  3. Take that value, and multiply it by the current position in the string (index + 1). Add this to total. For example, given the string "abc", the b is at position 2 in the alphabet and position 2 in the string, so it would contribute 2 * 2 = 4 towards total.
  4. After going through every character, total is the converted value.

Example

- This algorithm isn't actually a good hash function but is just an example of how one could convert strings into integers.
- You may be wondering: don't we need to limit total to a fixed size?
  - Correct! Right now, this algorithm is wrong. Let's say the limit we set was x. Then change step 4 to:
    - After going through every character, total % x is the converted value.
- % is the modulo operation, and makes sure the final converted value will be in the range [0, x - 1].

What is the point of a hash function?

- We know that arrays have O(1) random access.
- Given an arbitrary index, we can access and update its value in the array in constant time.
- The main constraint with arrays is that they are a fixed size, and the indices have to be integers.
- Because hash functions can convert any input into an integer, we can effectively remove the constraint of indices needing to be integers.
- When a hash function is combined with an array, it creates a hash map, also known as a hash table or dictionary.

What is the point of a hash function?

- With arrays, we map indices to values.
- With hash maps, we map keys to values, and a key can be almost anything.
- Typically, the only constraint on a hash map's key is that it has to be immutable (this is language dependent but generally a good rule of thumb). Values can be anything.

What is the point of a hash function?

- A hash map is probably the most important concept in all of algorithm interviewing.
- It is extremely powerful and allows you to reduce the time complexity of an algorithm by a factor of O(n) for a huge amount of problems.
- Every major language has a built-in implementation of a hash map.
- For example, in Python they're called dictionaries and declaring one is as simple as dic = {}. If you could only take one thing from this course, it should be to master the hash map interface for the programming language you use.

What is the point of a hash function?

- To summarize, a hash map is an unordered data structure that stores key-value pairs.
- A hash map can add and remove elements in O(1), as well as update values associated with a key and check if a key exists, also in O(1).
- You can iterate over both the keys and values of a hash map, but the iteration won't necessarily follow any order (there are many implementations and this is language dependent for the built-in types).

Note

- An ordered data structure is one where the insertion order is "remembered".
- An unordered data structure is one where the insertion order is not relevant.

Comparison with arrays

- In terms of time complexity, hash maps blow arrays out of the water.
- The following operations are all O(1) for a hash map:
  - Add an element and associate it with a value
  - Delete an element if it exists
  - Check if an element exists

Comparison with arrays

- A hash map also has many of the same useful properties as an array with the same time complexity:
  - Find length/number of elements
  - Updating values
  - Iterate over elements

Note

- Hash maps are also just easier/cleaner to work with.
- Even if your keys are integers and you could get away with using an array, if you don't know what the max size of your key is, then you don't know how large you should size your array.
- With hash maps, you don't need to worry about that, since the key will be converted to a new integer within the size limit anyways.

Disadvantages

- However, from a practical perspective, there are some disadvantages to using hash maps, and it's important to know them as it is common in interviews to talk about tradeoffs.
- The biggest disadvantage of hash maps is that for smaller input sizes, they can be slower due to overhead.
- Because big O ignores constants, the O(1) time complexity can sometimes be deceiving - it's usually something more like O(10) because every key needs to go through the hash function, and there can also be collisions, which we will talk about in the next section.

Disadvantages

- Hash tables can also take up more space. Dynamic arrays are actually fixed-size arrays that resize themselves when they go beyond their capacity. Hash tables are also implemented using a fixed size array - remember that the size is a limit set by the programmer. The problem is, resizing a hash table is much more expensive because every existing key needs to be re-hashed, and also a hash table may use an array that is significantly larger than the number of elements stored, resulting in a huge waste of space. Let's say you chose your limit as 10,000 items, but you only end up storing 10. Okay, you could argue that 10,000 is too large, but then what if your next test case ends up needing to store 100,000 elements? The point is, when you don't know how many elements you need to store, arrays are more flexible with resizing and not wasting space.

Note

- Remember that time complexity functions only involve the variables you define.
- When we say that hash map operations are O(1), the variable we are concerned with is usually n, which is the size of the hash map.
- However, this may be misleading. For example, hashing a string requires O(m) time, where m is the length of the string.
- The constant time operations are only constant relative to the size of the map.

Collisions

- When different keys convert to the same integer, it is called a collision.
- Without handling collisions, older keys will get overridden and data will be lost.
- There are multiple ways to handle collisions, but here we'll talk about a common one called chaining.

Collisions

- When using chaining, we store linked lists inside the hash map's array instead of the elements themselves.
- The linked list nodes store both the key and the value. If there are collisions, the collided key-value pairs are linked together in a linked list.
- Then, when trying to access one of these key-value pairs, we traverse through the linked list until the key matches.

Note

- If this part is confusing to you, don't worry. Every major programming language's hash map implementation will handle collisions automatically.
- The only reason to understand the inner workings of a hash map is that an interviewer may ask you trivia or want to discuss tradeoffs of using a hash map, but this is rare.

Collisions

- Collisions are problematic because handling them is necessary, and handling them takes time, slowing down the overall speed and efficiency of the hash map.
- How can we design our hash map to minimize collisions?
  - The most important thing is that the size of your hash table's array and modulus is a prime number.
- Prime numbers near significant magnitudes that are common to use are:
  - 10,007
  - 1,000,003
  - 1,000,000,007

Sets

- A set is another data structure that is very similar to a hash table.
- It uses the same mechanism for hashing keys into integers.
- The difference between a set and hash table is that sets do not map their keys to anything.
- Sets are more convenient to use when you only care about checking if elements exist.
- You can add, remove, and check if an element exists in a set all in O(1).
- An important thing to note about sets is that they don't track frequency. If you have a set and add the same element 100 times, the first operation adds it and the next 99 do nothing.

Arrays as keys?

- We said that being immutable is usually a requirement for being a hash map key.
- Arrays are mutable, so how do we store an ordered collection of elements as a key?
  - Depending on the language you're using, there are several ways to convert an array into a unique immutable key. In Python, tuples are immutable, so it's as easy as doing tuple(arr). Another trick is to convert the array into a string, delimited by some character that is guaranteed to not show up in any element. For example, use a comma to separate integers. [1, 51, 163] --> "1,51,163". Some languages like C++ support mutable keys in their built-in implementations.

Interface guide

```
// Declaration: Java supports multiple implementations, but we will using
// the Map interface with the HashMap implementation. Specify the data type
// of the keys and values
Map<Integer, Integer> hashMap = new HashMap<>();

// If you want to initialize it with some key value pairs, use the following syntax:
Map<Integer, Integer> hashMap = new HashMap<>() {{
    put(1, 2);
    put(5, 3);
    put(7, 2);
}};
```

Interface guide

```
// Checking if a key exists: use .containsKey()
hashMap.containsKey(1); // true
hashMap.containsKey(9); // false

// Accessing a value given a key: use .get()
hashMap.get(5); // 3

// Adding or updating a key: use .put()
// If the key already exists, the value will be updated
hashMap.put(5, 6);

// If the key doesn't exist yet, the key value pair will be inserted
hashMap.put(9, 15);
```

Interface guide

```
// Deleting a key: use .remove()
hashMap.remove(9);


// Get size
hashMap.size(); // 3
```

Interface guide

```
// Iterate over keys: use .keySet()
for (int key: hashMap.keySet()) {
    System.out.println(key);
}

// Iterate over values: use .values()
for (int val: hashMap.values()) {
    System.out.println(val);
}
```

Example

```
public class Example {
   public static void main(String[] args) {
      Map<Integer, Integer> myHashMap = new HashMap<>();
      // myHashMap has integers for both keys and values

      myHashMap.put(4, 83);
      System.out.println(myHashMap.get(4)); // Prints 83
      System.out.println(myHashMap.containsKey(4)); // Prints true
      System.out.println(myHashMap.containsKey(854)); // Prints false

      myHashMap.put(8, 327);
      myHashMap.put(45, 82523);

      for (int key: myHashMap.keySet()) {
         System.out.println(String.format("%d: %d", key, myHashMap.get(key)));
      }
   }
}
```

Two Sum

- Given an array of integers nums and an integer target, return indices of two numbers such that they add up to target.
- You cannot use the same index twice.

Example

- Input:
  - [5, 2, 7, 10, 3, 9]
  - 8
- Output
  - [0, 4]

First Letter to Appear Twice

- Given a string s, return the first character to appear twice.
- It is guaranteed that the input will have a duplicate character.

Example

- Input:
  - "abcdeda"
- Output
  - 'd'

All Unique Numbers

- Given an integer array nums, find all the unique numbers x in nums that satisfy the following:
  - x + 1 is not in nums, and x - 1 is not in nums.

Check if the Sentence Is Pangram

- A pangram is a sentence where every letter of the English alphabet appears at least once.
- Given a string sentence containing only lowercase English letters, return true if sentence is a pangram, or false otherwise.

Example 1

- Input:
  - sentence = "thequickbrownfoxjumpsoverthelazydog"
- Output:
  - true
- Explanation:
  - sentence contains at least one of every letter of the English alphabet.

Example 2

- Input:
  - sentence = "leetcode"
- Output:
  - false

Constraints

- 1 <= sentence.length <= 1000
- sentence consists of lowercase English letters.

Missing Number

- Given an array nums containing n distinct numbers in the range [0, n], return the only number in the range that is missing from the array.

Example 1

- Input:
  - nums = [3,0,1]
- Output:
  - 2
- Explanation:
  - n = 3 since there are 3 numbers, so all numbers are in the range [0,3]. 2 is the missing number in the range since it does not appear in nums.

Example 2

- Input:
  - nums = [0,1]
- Output:
  - 2
- Explanation:
  - n = 2 since there are 2 numbers, so all numbers are in the range [0,2]. 2 is the missing number in the range since it does not appear in nums.

Example 3

- Input:
  - nums = [9,6,4,2,3,5,7,0,1]
- Output:
  - 8
- Explanation:
  - n = 9 since there are 9 numbers, so all numbers are in the range [0,9]. 8 is the missing number in the range since it does not appear in nums.

Constraints

- n == nums.length
- $1 <= n <= 10^4$
- 0 <= nums[i] <= n
- All the numbers of nums are unique.

Counting Elements

- Given an integer array arr, count how many elements x there are, such that x + 1 is also in arr.
- If there are duplicates in arr, count them separately.

Example 1

- Input:
  - arr = [1,2,3]
- Output:
  - 2
- Explanation:
  - 1 and 2 are counted cause 2 and 3 are in arr.

Example 2

- Input:
  - arr = [1,1,3,3,5,5,7,7]
- Output:
  - 0
- Explanation:
  - No numbers are counted, cause there is no 2, 4, 6, or 8 in arr.

Constraints

- 1 <= arr.length <= 1000
- 0 <= arr[i] <= 1000

Queries?

Thank You…!