



Ask a Question



# Introduction to Hash Maps

Let's go over the Hash Maps pattern, its real-world applications, and some problems we can solve with it.


We'll cover the following... ▼

## About the pattern

A **hash map**, also known as a hash table, is a data structure that stores key-value pairs. It provides a way to efficiently map keys to values, allowing for quick retrieval of a value associated with a given key. Hash maps achieve this efficiency by using a hash function behind the scenes to compute an index (or hash code) for each key. This index determines where the corresponding value will be stored in an underlying array.

Below is an explanation of the staple methods of a hash map:



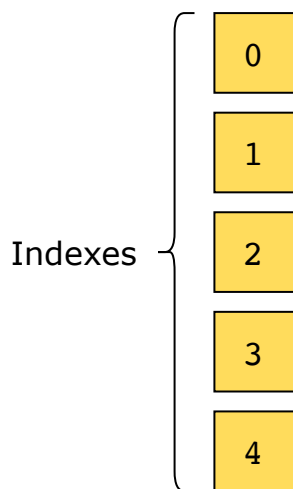
- 
- **Insert(key, value):** When a key-value pair is inserted into a hash map, the hash function computes an index based on the key. This index is used to determine the location in the hash map where the value will be stored. Because different keys may hash to the same index (collision), hash maps typically include a collision resolution strategy. Common methods include chaining or open addressing. In the average case, inserting a key-value pair takes  $O(1)$  time, assuming the hash function distributes keys uniformly across the array. However, in the worst case (when all the keys hash to the same index), insertion can take up to  $O(n)$  time, where  $n$  is the number of elements in the hash map.
  - **Search(key):** To retrieve a value from the hash map, the hash function is applied to the key to compute its index. Then, the value stored at that index is returned. In the average case, searching for a value takes  $O(1)$  time. In the worst case, it can take up to  $O(n)$  time due to resizing and handling collisions.
  - **Remove(key):** Removing a key-value pair typically involves finding the index based on the key's hash and then removing the value stored at that index. In the average case, removing a key-value pair takes  $O(1)$  time. In the worst case, it can take up to  $O(n)$  time due to resizing and handling collisions.

The following illustration shows an example of these methods being used in a hash map:





We will use a hash map to store the marks of students in key-value pairs. Here, the name of the student will be the key, and their corresponding marks will be the value. Programming languages typically initialize the hash map with a small initial capacity. For the sake of this example, let's assume that capacity is 5.

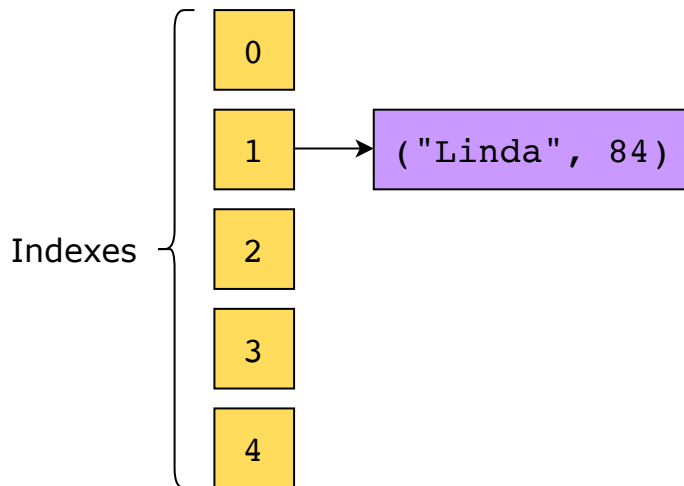


1 of 6



**Insert("Linda", 84):** We will insert the key-value pair, ("Linda", 84) into the hash map. The specified hash function will take the key, "Linda" as a parameter and return the index in which this key-value pair should be stored. For the sake of this example let's assume that the returned index is **1**.

**hash function("Linda") = 1**

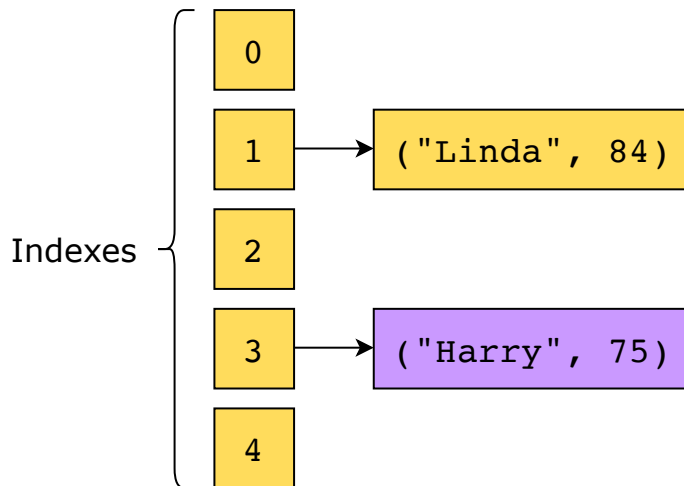


2 of 6



**Insert("Harry", 75):** We will insert the key-value pair, ("Harry", 75) into the hash map. The specified hash function will take the key, "Harry" as a parameter and return the index in which this key-value pair should be stored. For the sake of this example let's assume that the returned index is **3**.

**hash function("Harry") = 3**

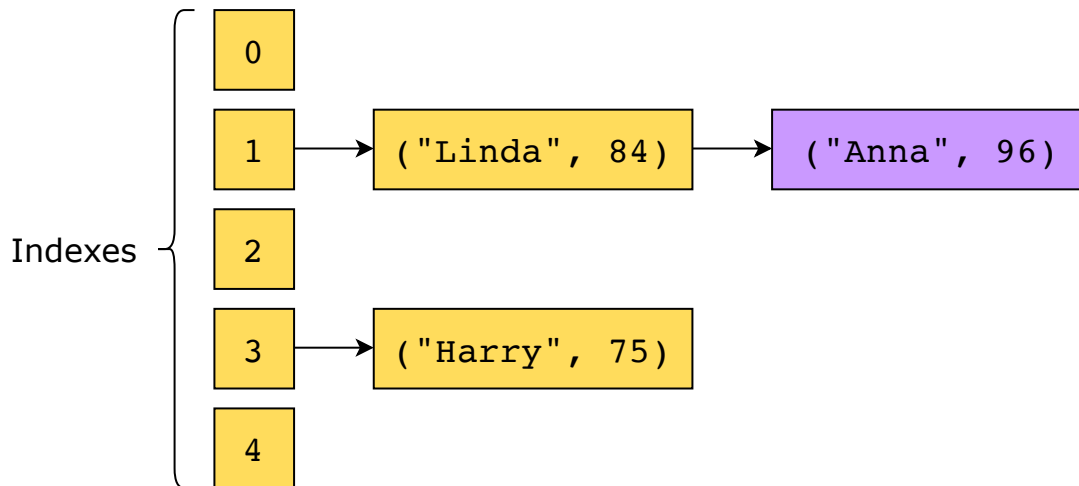


3 of 6



**Insert("Anna", 96):** We will insert the key-value pair, ("Anna", 96) into the hash map. The specified hash function will take the key, "Anna" as a parameter and return the index in which this key-value pair should be stored. For the sake of this example let's assume that the returned index is **1**. Because a key-value pair is already stored at index **1**, we will use chaining to store this new pair in a linked list.

**hash function("Anna") = 1**



4 of 6

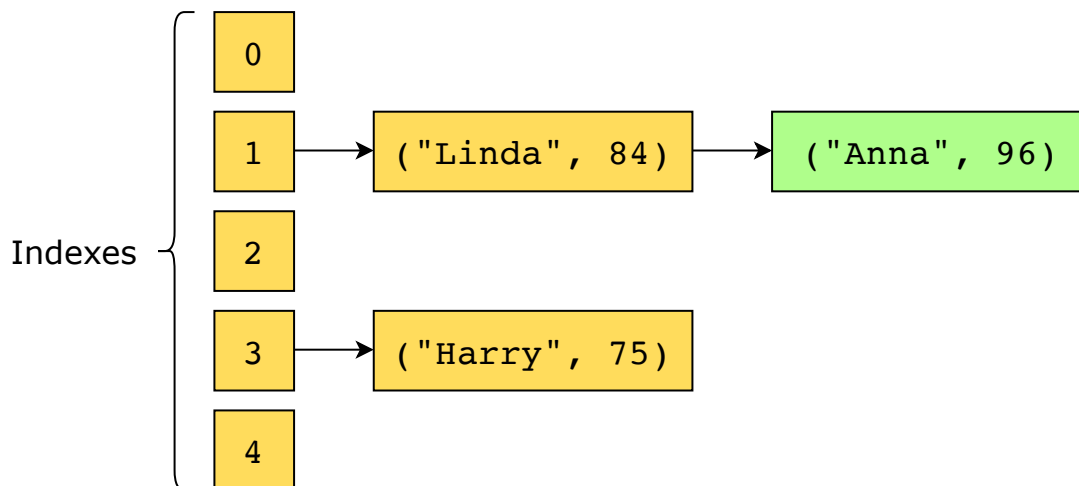
?

Tt



**Search("Anna"):** We will first compute the index of "Anna", to shortlist the position of the key-value pair in the hash map. The specified hash function will take the key, "Anna" as a parameter and return the index, 1, in which this key-value pair is stored. Because there are two key-value pairs at index 1, the linked list will be traversed to find the value corresponding to the key "Anna". Finally, the value, **96**, will be returned.

**hash function("Anna") = 1**

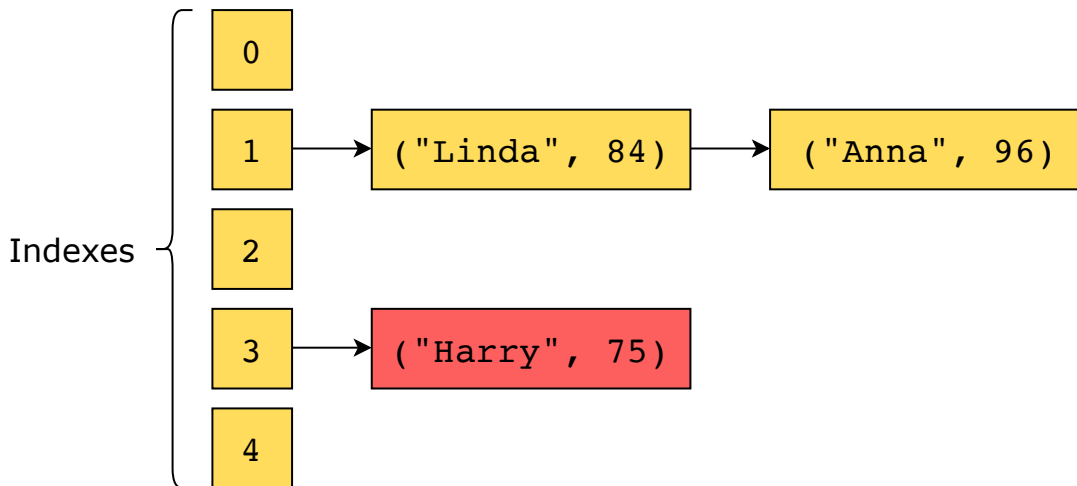


5 of 6



**Delete("Harry"):** We will first compute the index of "Harry", to shortlist the position of the key-value pair in the hash map. The specified hash function will take the key, "Harry" as a parameter and return the index, **3**, in which this key-value pair is stored. The key-value pair stored at this index will be removed.

**hash function("Harry") = 3**



6 of 6

—

⌂

For the example above, had we used arrays, we would have to create the following two arrays:

- **names:** This stores the names of the students.
- **marks:** This stores the marks of the students.

We would have to make sure that the **names** and **marks** arrays store their values in the same order. This means if **John** is at index **4** in the **names** array, his marks should also be present at the same index in the **marks** array. This process is very tedious because we first have to make a lookup in the **names** array in  $O(n)$  time and then, using the corresponding



index, perform another lookup in the **marks** array in  $O(1)$  time. Using a hash map makes our lives much easier. We can use the names of the students as keys, and their marks are the corresponding values. Now, we only have to perform one lookup for the marks of a student in  $O(1)$  time on average.

When using hash maps, it's important to understand the underlying operations being performed, i.e., hash functions and collision resolution strategies, as they aid us in determining the time complexities of its methods. However, in the upcoming lessons, we'll be using a simpler approach to illustrate hash maps. This will allow us to focus on the algorithm at hand. The following illustration shows the same example of storing the marks of students in this simpler representation:

**Insert("Linda", 84)**

hash map
"Linda": 84

1 of 5



**Insert("Harry", 75)**

hash map
"Linda": 84
"Harry": 75

2 of 5

**Insert("Anna", 96)**

hash map
"Linda": 84
"Harry": 75
"Anna": 96

3 of 5

**Search("Anna")**

hash map
"Linda": 84
"Harry": 75
"Anna": 96

?

Tt



**Delete("Harry")**

hash map
"Linda": 84
"Harry": 75
"Anna": 96



## Examples

The following examples illustrate some problems that can be solved with this approach:

1. **Two sum:** Check for a pair in an array with a target sum.



Traverse the array and at each element, calculate the difference between the target value and the current value, and assign it to a variable called temp. For example, if we're at value 11 in the array,

