Data Structures and Algorithms

Lesson 6. Hashing and Symbol Tables

Alexandra Ciobotaru

31 mar 2022

Syllabus

- **Lesson 1. Introduction: Algorithms, Data Structures and Cognitive Science.**
- **Lesson 2. Python Data Types and Structures.**
- **Lesson 3. Principles of Algorithm Design.**
- **Lesson 4. Lists and Pointer Structures.**
- **Lesson 5. Stacks and Queues.**
- **Lesson 6. Trees.**
- **Lesson 7. Hashing and Symbol Tables.**
- Lesson 8. Graphs and Other Algorithms.
- Lesson 9. Searching.
- Lesson 10. Sorting. Mid-term evaluation quiz (30%)
- Lesson 11. Selection Algorithms.
- Lesson 12. String Algorithms and Techniques.
- Lesson 13. Design Techniques and Strategies.
- Lesson 14. Implementations, Applications and Tools.
- Lab (20%) + Project (50%).

What is Hashing?

- Before we flesh out what a hash table is, let's define hashing. According to HackerEarth (https://www.hackerearth.com/practice/data-structures/hash-tables/basics-of-hash-tables/tutorial/), "**Hashing is a technique that is used to uniquely identify a specific object from a group of similar objects**."
- In other words, hashing is the process of creating a *unique identifier*. This is done through the use of a predefined function that, given the same input, will always produce the same output.
- Have you ever downloaded an install file and seen a SHA1 or MD5 checksum next to the link? Those are hashes of the executable so you can be confident the file you are installing has not been tampered with.

What are Hash Tables?

- Now that we know a hash is nothing more than a predictably generated identifier, let's pull back the curtain on what a hash table is.
- A hash table is the product of a **hash function**
- The result of the hash (also called a search key) functions as the identifier to a specific space that holds data.
  - In the case of a library book's Dewey number, that translates to a row and shelf. In the case of a programming language, that translates to an index in a list.

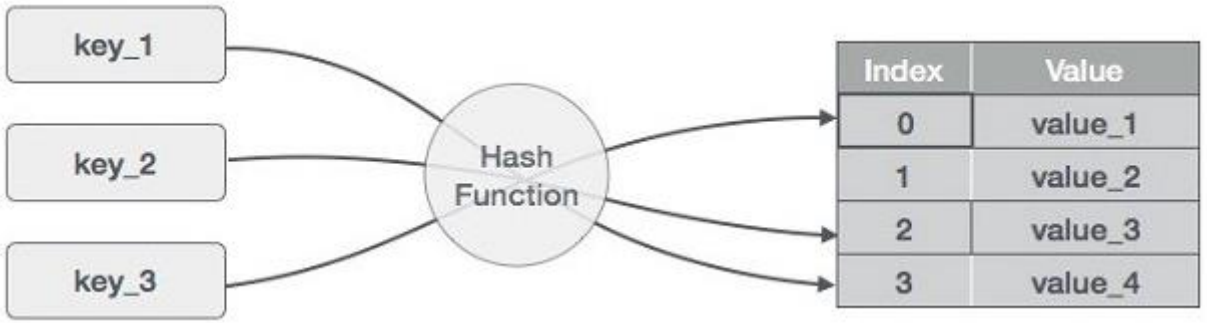


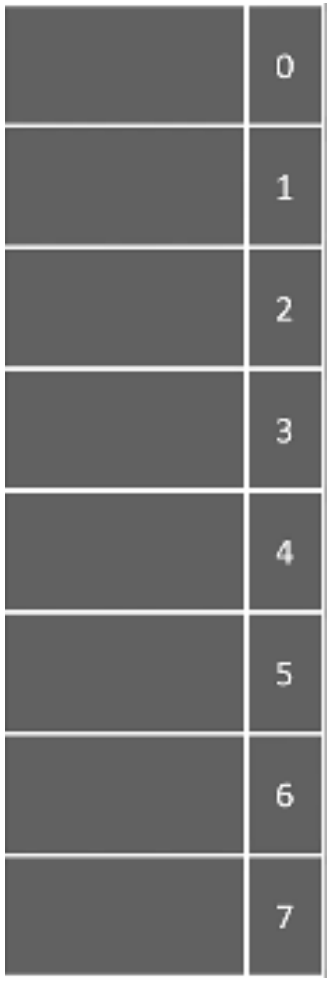
Image source: https://betterprogramming.pub/what-are-hash-tables-and-why-are-they-amazing-89cf52246f91

Hash Tables

- First let’s look at an address space, which we’ll consider to be a list for now
- We are going to create a hardware store in which we’ll put items and then assign a number to those items
- **Dictionaries are built-in hash tables**
- {“nails”: 1000}

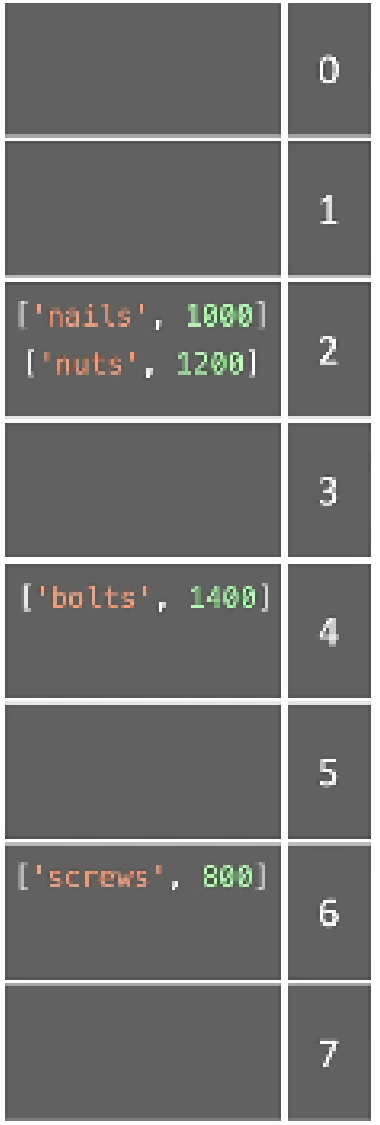
  key value

- We perform a hash on the key, meaning that we pass the key through the hash function, and the output is the key-value pair, and also an *address*, at which we’ll store the key-value pair



The Hash

- Characteristics:
  - The hash is one way (“nails” receives memory 2, but 2 can’t receive “nails”)
  - The hash is deterministic (for a particular hash function, every time we put the “nails” in we expect to get the value of 2)
  - It has a prime number of addresses (it increases the randomness)
- So even if Python has a form of hash table (the dictionary), we are going to build our own
  - We will create out own address space (by creating a list)
  - We’ll create methods – ex. `set_item(key, value)`
  - We will create our own hash to hash the key and produce an address
  - `get_hash(key)` will return the address where that key-value pair is stored – we can go directly to that address and get the value of that key



Collisions

- A collision happens when you put a key-value pair at an address where you already have a key-value pair
- A way to do that is to put both key-value pairs to coexist within a list, at that address – the method is called **Separate Chaining** (this is how we'll handle collisions today)
- Another method is to go down the address list until you find an empty space and put the key-value pair there – the method is called **Linear Probing**, and that is a form of open addressing
- Another way of doing separate chaining is by using linked lists

Hashing

- We will create a hash function that converts strings into integers
- We can obtain the unique ordinal value of any character by using the `ord()` function
- ord('f') -> 102
- **sum(map(ord, 'hello world'))** -> 1116

h	e	l	l	o		w	o	r	l	d	
104	101	108	108	111	32	119	111	114	108	100	= 1116

- However, if we change the order of letters we get the same hash
- A better hash function is one that can get me a unique hash value for a given string

Hashing

- We can add a multiplier that continuously increases as we progress in the string

h	e	l	l	o		w	o	r	l	d	
104	101	108	108	111	32	119	111	114	108	100	= 1116
1	2	3	4	5	6	7	8	9	10	11	
104	202	324	432	555	192	833	888	1026	1080	1100	= 6736

Strings

Ordinal values

Multiplier

Result of multiplication

- The ordinal value of each character is progressively multiplied by a number
- Can you implement this hash function? (see book page 183)

The Constructor

```
class HashTable: # for building the hashing list of default size 7
    def __init__(self, size=7):
        self.data_map = [None] * size # creates a list of 7 None items

    def __hash(self, key):  # the hash method
        my_hash = 0
        for letter in key:  #we loop through the letters of the key that we passed
            my_hash = (my_hash + ord(letter) * 23) % len(self.data_map)
        return my_hash
        # 'ord' returns the ascii number of each letter as we are looping through the key
    def print_table(self):
        for i, val in enumerate(self.data_map):
            print(i, ": ", val)

my_hash_table = HashTable()
my_hash_table.print_table()
```

The remainder will be anything from 0 to 6

Set a key-value pair

- set_item(“nuts”, 1000) ->
  - applies the hash on the key to create the address,
  - puts the key-value pair in a list and
  - positions this list at the value of 4 in the address list, inside a separate list

```
 def set_item(self, key, value):
    index = self.__hash(key)
    if self.data_map[index] == None:
        self.data_map[index] = []
    self.data_map[index].append([key, value])
my_hash_table = HashTable()
my_hash_table.set_item('bolts', 1400)
my_hash_table.set_item('lumber', 70)
```

If we don’t have anything in the index we initialize with an empty list

Get a value

- If ‘washers’ is stored at 4 with the value 1400, `get_item('washers')` will:
  - Give us an address of 4
  - We go to the 4 address
  - We loop through the key-value pairs till we get to ‘washers’
  - We return its value

```
 def get_item(self, key):
    index = self.__hash(key)
    # we search only in those places where we have values
    if self.data_map[index] is not None:
        for i in range(len(self.data_map[index])):
            if self.data_map[index][i][0] == key:
                return self.data_map[index][i][1]
    return None
my_hash_table = HashTable()
my_hash_table.set_item('bolts', 1400)
my_hash_table.set_item('washers', 50)
print(my_hash_table.get_item('bolts'))
```

We compare the key

And return the value

Keys Method

- We want to get all of the keys from the hash table, put them into a list, and then return that list
- We will need a for loop that goes through the datamap list to see if there's anything in any of the addresses
- And when it finds something at an address, we'll use another for loop to loop through the key-value pairs in that list (we run it if the address map is not None)

```
 def keys(self):
    all_keys = []
    for i in range(len(self.data_map)):
        if self.data_map[i] is not None:
            for j in range(len(self.data_map[i])):
                all_keys.append(self.data_map[i][j][0])
    return all_keys
my_hash_table = HashTable()
my_hash_table.set_item('bolts', 1400)
my_hash_table.set_item('lumber', 70)
print(my_hash_table.keys())
```

Big O

- The BigO of the hash method itself is O(1)
- set_item(key, value) – append at an address - is O(1)
- get_item(nails)
  - O(1) to find the address
  - And then we need to iterate through the list to find nails – O(N)

Question

- Suppose you have two lists:
  - [1, 3, 5]
  - [2, 4, 5]
- We want to determine if these lists have an item in common
- A. Naïve approach
  - Create nested for loops (but this is $O(N^2)$)

```
def item_in_common(list1, list2):
    for i in list1:
        for j in list2:
            if i==j:
                return True
    return False
print(item_in_common([1,2,3],[3,4,5]))
```

- B. A more elevated approach
  - Loop through list1 and add list elements as keys (O(N))
  - Search by key (every time you look for an item in a dictionary is O(1))
  - O(N) + O(1) = O(N)

```
def item_in_common(list1, list2):
    my_dict = {}
    for i in list1:
        my_dict[i] = True
    for j in list2:
        if j in my_dict:
            return True
    return False
print(item_in_common([1,2,3],[3,4,5]))
```

Thank you!