HashMap

May 28, 2020

(In addition to having fun) We write programs to solve real world problems. Data structures help us in representing and efficiently manipulating the data associated with these problems.

Let us see if we can use any of the data structures that we already know to solve the following problem

0.0.1 Problem Statement

In a class of students, store heights for each student.

The problem in itself is very simple. We have the data of heights of each student. We want to store it so that next time someone asks for height of a student, we can easily return the value. But how can we store these heights?

Obviously we can use a database and store these values. But, let's say we don't want to do that for now. We want to use a data structure to store these values as part of our program. For the sake of simplicity, our problem is limited to storing heights of students. But you can certainly imagine scenarios where you have to store such `key-value` pairs and later on when someone gives you a `key`, you can efficiently return the corrresponding `value`.

The class diagram for HashMaps would look something like this.

```
In [10]: class HashMap:

             def __init__(self):
                 self.num_entries = 0

             def put(self, key, value):
                 pass

             def get(self, key):
                 pass

             def size(self):
                 return self.num_entries
```

1 Arrays

Can we use arrays to store `key-value` pairs?

We can certainly use one array to store the names of the students and use another array to store their corresponding heights at the corresponding indices.

What will be the time complexity in this scenario?

To obtain height of a student, say `Potter, Harry`, we will have to traverse the entire array and check if the value at a particular index matches `Potter, Harry`. Once we find the index in which this value is stored, we can use this index to obtain the height from the second array.

Thus, because of this traveral, complexity for `get()` operation becomes $O(n)$. Even if we maintain a sorted array, the operation will not take less than $O(log(n))$ complexity.

What happens if a student leaves a class? We will have to delete the entry corresponding to the student from both the arrays.

This would require another traversal to find the index. And then we will have to shift our entire array to fill this gap. Again, the time complexity of operation becomes $O(n)$

1.0.1 Linked List

Is it possible to use linked lists for this problem?

We can certainly modify our `LinkedListNode` to have two different value attributes - one for name of the student and the other for height.

But we again face the same problem. In the worst case, we will have to traverse the entire linked list to find the height of a particular student. Once again, the cost of operation becomes $O(n)$.

1.0.2 Stacks and Queues

Stacks and Queues are LIFO and FIFO data structures respectively. Can you think why they too do not make a good choice for storing `key-value` pairs?

Can we do better? Can you think of any data structure that allows for fast `get()` operation?

Let us circle back to arrays.

When we obtain the element present at a particular index using something like `arr[3]`, the operation takes constant i.e. `O(1)` time.

For review - Does this constant time operation require further explanation?

If we think about `array indices as keys` and the `element present at those indices as values`, we can fairly conclude that at least for non-zero integer `keys`, we can use arrays.

However, like our current problem statement, we may not always have integer keys.

`If only we had a function that could give us arrays indices for any key value that we gave it!`

1.1 Hash Functions

Simply put, hash functions are these really incredible `magic` functions which can map data of any size to a fixed size data. This fixed sized data is often called hash code or hash digest.

Let's create our own hash function to store strings

```
In [1]: def hash_function(string):
            pass
```

For a given string, say abcd, a very simple hash function can be sum of corresponding ASCII values.

Note: you can use ord(character) to determine ASCII value of a particular character e.g. ord('a') will return 97

```
In [3]: def hash_function(string):
            hash_code = 0
            for character in string:
                hash_code += ord(character)
            return hash_code
```

```
In [5]: hash_code_1 = hash_function("abcd")
        print(hash_code_1)
```

```
394
```

Looks like our hash function is working fine. But is this really a good hash function?

For starters, it will return the same value for abcd and bcda. Do we want that? We can create 24 different permutations for the string abcd and each will have the same value. We cannot put 24 values to one index.

Obviously, this makes it clear that our hash function must return unique values for unique objects.

When two different inputs produce the same output, then we have something called a collision. An ideal hash function must be immune from producing collisions.

Let's think something else.

Can product help? We will again run in the same problem.

The honest answer is that we have differernt hash functions for different types of keys. The hash function for integers will be different from the hash function for strings, which again, will be different for some object of a class that you created.

However, let's try to continue with our problem and try to come up with a hash function for strings.

1.2 Hash Function for Strings

For a string, say abcde, a very effective function is treating this as number of prime number base p. Let's elaborate this statement.

For a number, say 578, we can represent this number in base 10 number system as

$$5 * 10^2 + 7 * 10^1 + 8 * 10^0$$

Similarly, we can treat abcde as

$$a * p^4 + b * p^3 + c * p^2 + d * p^1 + e * p^0$$

Here, we replace each character with its corresponding ASCII value.

A lot of research goes into figuring out good hash functions and this hash function is one of the most popular functions used for strings. We use prime numbers because the provide a good distribution. The most common prime numbers used for this function are 31 and 37.

Thus, using this algorithm, we can get a corresponding integer value for each string key and store it in the array.

Note that the array used for this purpose is called a `bucket array`. It is not a special array. We simply choose to give a special name to arrays for this purpose. Each entry in this `bucket array` is called a `bucket` and the index in which we store a bucket is called `bucket index`.

Let's add these details to our class.

```
In [9]: class HashMap:

            def __init__(self, initial_size=10):
                self.bucket_array = [None for _ in range(initial_size)]
                self.p = 37
                self.num_entries = 0

            def put(self, key, value):
                pass

            def get(self, key):
                pass

            def get_bucket_index(self, key):
                return self.get_hash_code(key)

            def get_hash_code(self, key):
                key = str(key)
                num_buckets = len(self.bucket_array)
                current_coefficient = 1
                hash_code = 0
                for character in key:
                    hash_code += ord(character) * current_coefficient
                    current_coefficient *= self.p
                    current_coefficient = current_coefficient

                return hash_code
```

```
In [4]: hash_map = HashMap()

        bucket_index = hash_map.get_bucket_index("abcd")
        print(bucket_index)
```

```
5204554
```

```
In [5]: hash_map = HashMap()

        bucket_index = hash_map.get_bucket_index("bcda")
        print(bucket_index)
```

5054002

1.3 Compression Function

We now have a good hash function which will return unique values for unique objects. But let's look at the values. These are huge. We cannot create such large arrays. So we use another function - `compression function` to compress these values so as to create arrays of reasonable sizes.

A very simple, good, and effective compression function can be `mod len(array)`. The `modulo operator %` returns the remainder of one number when divided by other.

So, if we have an array of size 10, we can be sure that modulo of any number with 10 will be less than 10, allowing it to fit into our bucket array.

Because of how modulo operator works, instead of creating a new function, we can write the logic for compression function in our `get_hash_code()` function itself.

https://www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/modular-multiplication

```
In [11]: class HashMap:

             def __init__(self, initial_size = 10):
                 self.bucket_array = [None for _ in range(initial_size)]
                 self.p = 31
                 self.num_entries = 0

             def put(self, key, value):
                 pass

             def get(self, key):
                 pass

             def get_bucket_index(self, key):
                 bucket_index = self.get_hash_code(keu)
                 return bucket_index

             def get_hash_code(self, key):
                 key = str(key)
                 num_buckets = len(self.bucket_array)
                 current_coefficient = 1
                 hash_code = 0
                 for character in key:
                     hash_code += ord(character) * current_coefficient
                     hash_code = hash_code % num_buckets                       # compress hash_c
                     current_coefficient *= self.p
                     current_coefficient = current_coefficient % num_buckets   # compress coeffi

                 return hash_code % num_buckets                                # one last compre
```

```
            def size(self):
                return self.num_entries
```

1.4 Collision Handling

As discussed earlier, when two different inputs produce the same output, then we have a collision. Our implementation of `get_hash_code()` function is satisfactory. However, because we are using compression function, we are prone to collisions.

Consider the following scenario.

We have a bucket array of length 10 and we get two different hash codes for two different inputs, say 355, and 1095. Even though the hash codes are different in this case, the bucket index will be same because of the way we have implemented our compression function. Such scenarios where multiple entries want to go to the same bucket are very common. So, we introduce some logic to handle collisions.

There are two popular ways in which we handle collisions.

1. Closed Addressing or Separate chaining
2. Open Addressing
3. Closed addressing is a clever technique where we use the same bucket to store multiple objects. The bucket in this case will store a linked list of key-value pairs. Every bucket has it's own separate chain of linked list nodes.
4. In open addressing, we do the following:
   - If, after getting the bucket index, the bucket is empty, we store the object in that particular bucket
   - If the bucket is not empty, we find an alternate bucket index by using another function which modifies the current hash code to give a new code

Separate chaining is a simple and effective technique to handle collisions and that is what we discuss here.

Implement the `put` and `get` function using the idea of separate chaining.

```
In [64]: class LinkedListNode:

             def __init__(self, key, value):
                 self.key = key
                 self.value = value
                 self.next = None

         class HashMap:

             def __init__(self, initial_size = 10):
                 self.bucket_array = [None for _ in range(initial_size)]
                 self.p = 31
```

```
        self.num_entries = 0

    def put(self, key, value):
        bucket_index = self.get_bucket_index(key)

        new_node = LinkedListNode(key, value)
        head = self.bucket_array[bucket_index]

        # check if key is already present in the map, and update it's value
        while head is not None:
            if head.key == key:
                head.value = value
                return
            head = head.next

        # key not found in the chain --> create a new entry and place it at the head of
        head = self.bucket_array[bucket_index]
        new_node.next = head
        self.bucket_array[bucket_index] = new_node
        self.num_entries += 1

    def get(self, key):
        bucket_index = self.get_hash_code(key)
        head = self.bucket_array[bucket_index]
        while head is not None:
            if head.key == key:
                return head.value
            head = head.next
        return None

    def get_bucket_index(self, key):
        bucket_index = self.get_hash_code(key)
        return bucket_index

    def get_hash_code(self, key):
        key = str(key)
        num_buckets = len(self.bucket_array)
        current_coefficient = 1
        hash_code = 0
        for character in key:
            hash_code += ord(character) * current_coefficient
            hash_code = hash_code % num_buckets                       # compress hash_c
            current_coefficient *= self.p
            current_coefficient = current_coefficient % num_buckets   # compress coeffi

        return hash_code % num_buckets                                # one last compre

    def size(self):
```

```
                return self.num_entries
```

```
In [65]: hash_map = HashMap()

         hash_map.put("one", 1)
         hash_map.put("two", 2)
         hash_map.put("three", 3)
         hash_map.put("neo", 11)

         print("size: {}".format(hash_map.size()))


         print("one: {}".format(hash_map.get("one")))
         print("neo: {}".format(hash_map.get("neo")))
         print("three: {}".format(hash_map.get("three")))
         print("size: {}".format(hash_map.size()))
```

```
size: 4
one: 1
neo: 11
three: 3
size: 4
```

1.5 Time Complexity and Rehashing

We used arrays to implement our hashmaps because arrays offer $O(1)$ time complexity for both put and get operations.

Note: in case of arrays put is simply `arr[i] = 5` and get is `height = arr[5]`

1. Put Operation

- In the put operation, we first figure out the bucket index. Calculating the hash code to figure out the bucket index takes some time.
- After that, we go to the bucket index and in the worst case we traverse the linked list to find out if the key is already present or not. This also takes some time.

To analyze the time complexity for any algorithm as a function of the input size `n`, we first have to determine what our input is. In this case, we are putting and gettin key value pairs. So, these entries i.e. key-value pairs are our input. Therefore, our `n` is number of such key-value pair entries.

Note: time complexity is always determined in terms of input size and not the actual amount of work that is being done independent of input size. That "independent amount of work" will be constant for every input size so we disregard that.

- In case of our hash function, the computation time for hash code depends on the size of each string. Compared to number of entries (which we always consider to be very high e.g. in the order of 10^5) the length of each string can be considered to be very small. Also, most of the strings will be around the same size when compared to this high number of entries. Hence, we can ignore the hash computation time in our analysis of time complexity.
- Now, the entire time complexity essentialy depends on the linked list traversal. In the worst case, all entries would go to the same bucket index and our linked list at that index would be huge. Therefore, the time complexity in that scenario would be $O(n)$. However, hash functions are wisely chosen so that this does not happen.

`On average, the distribution of entries is such that if we have n entries and b buckets, then each bucket does not have more than n/b key-value pair entries.`

Therefore, because of our choice of hash functions, we can assume that the time complexity is $O(\frac{n}{b})$. This number which determines the `load` on our bucket array `n/b` is known as load factor.

Generally, we try to keep our load factor around or less than 0.7. This essentially means that if we have a bucket array of size 10, then the number of key-value pair entries will not be more than 7.

What happens when we get more entries and the value of our load factor crosses 0.7?

In that scenario, we must increase the size of our bucket array. Also, we must recalculate the bucket index for each entry in the hashn map.

Note: the hash code for each key present in the bucket array would still be the same. However, because of the compression function, the bucket index will change.

Therefore, we need to `rehash` all the entries in our hash map. This is known as `Rehashing`.

2. Get and Delete operation Can you figure out the time complexity for get and delete operation?

*Answer: the solution follows the same logic and the time complexity is O(1). Note that we do not reduce the size of bucket array in delete operation.

1.5.1 Rehashing Code

```
In [66]: class LinkedListNode:

             def __init__(self, key, value):
                 self.key = key
                 self.value = value
                 self.next = None

         class HashMap:

             def __init__(self, initial_size = 15):
                 self.bucket_array = [None for _ in range(initial_size)]
                 self.p = 31
                 self.num_entries = 0
                 self.load_factor = 0.7

             def put(self, key, value):
```

```
        bucket_index = self.get_bucket_index(key)

        new_node = LinkedListNode(key, value)
        head = self.bucket_array[bucket_index]

        # check if key is already present in the map, and update it's value
        while head is not None:
            if head.key == key:
                head.value = value
                return
            head = head.next

        # key not found in the chain --> create a new entry and place it at the head of
        head = self.bucket_array[bucket_index]
        new_node.next = head
        self.bucket_array[bucket_index] = new_node
        self.num_entries += 1

        # check for load factor
        current_load_factor = self.num_entries / len(self.bucket_array)
        if current_load_factor > self.load_factor:
            self.num_entries = 0
            self._rehash()

    def get(self, key):
        bucket_index = self.get_hash_code(key)
        head = self.bucket_array[bucket_index]
        while head is not None:
            if head.key == key:
                return head.value
            head = head.next
        return None

    def get_bucket_index(self, key):
        bucket_index = self.get_hash_code(key)
        return bucket_index

    def get_hash_code(self, key):
        key = str(key)
        num_buckets = len(self.bucket_array)
        current_coefficient = 1
        hash_code = 0
        for character in key:
            hash_code += ord(character) * current_coefficient
            hash_code = hash_code % num_buckets                       # compress hash_c
            current_coefficient *= self.p
            current_coefficient = current_coefficient % num_buckets   # compress coeffi
        return hash_code % num_buckets                                # one last compre
```

```
            def size(self):
                return self.num_entries

            def _rehash(self):
                old_num_buckets = len(self.bucket_array)
                old_bucket_array = self.bucket_array
                num_buckets = 2 * old_num_buckets
                self.bucket_array = [None for _ in range(num_buckets)]

                for head in old_bucket_array:
                    while head is not None:
                        key = head.key
                        value = head.value
                        self.put(key, value)         # we can use our put() method to rehash
                        head = head.next
```

```
In [67]: hash_map = HashMap(7)

         hash_map.put("one", 1)
         hash_map.put("two", 2)
         hash_map.put("three", 3)
         hash_map.put("neo", 11)

         print("size: {}".format(hash_map.size()))


         print("one: {}".format(hash_map.get("one")))
         print("neo: {}".format(hash_map.get("neo")))
         print("three: {}".format(hash_map.get("three")))
         print("size: {}".format(hash_map.size()))
```

```
size: 4
one: 1
neo: 11
three: 3
size: 4
```

1.6 Delete Operation

Can you implement delete operation using all we have learnt so far?

```
In [70]: class LinkedListNode:

             def __init__(self, key, value):
                 self.key = key
                 self.value = value
                 self.next = None
```

```python
class HashMap:

    def __init__(self, initial_size = 15):
        self.bucket_array = [None for _ in range(initial_size)]
        self.p = 31
        self.num_entries = 0
        self.load_factor = 0.7

    def put(self, key, value):
        bucket_index = self.get_bucket_index(key)

        new_node = LinkedListNode(key, value)
        head = self.bucket_array[bucket_index]

        # check if key is already present in the map, and update it's value
        while head is not None:
            if head.key == key:
                head.value = value
                return
            head = head.next

        # key not found in the chain --> create a new entry and place it at the head of
        head = self.bucket_array[bucket_index]
        new_node.next = head
        self.bucket_array[bucket_index] = new_node
        self.num_entries += 1

        # check for load factor
        current_load_factor = self.num_entries / len(self.bucket_array)
        if current_load_factor > self.load_factor:
            self.num_entries = 0
            self._rehash()

    def get(self, key):
        bucket_index = self.get_hash_code(key)
        head = self.bucket_array[bucket_index]
        while head is not None:
            if head.key == key:
                return head.value
            head = head.next
        return None

    def get_bucket_index(self, key):
        bucket_index = self.get_hash_code(key)
        return bucket_index

    def get_hash_code(self, key):
```

```
                 key = str(key)
                 num_buckets = len(self.bucket_array)
                 current_coefficient = 1
                 hash_code = 0
                 for character in key:
                     hash_code += ord(character) * current_coefficient
                     hash_code = hash_code % num_buckets                      # compress hash_c
                     current_coefficient *= self.p
                     current_coefficient = current_coefficient % num_buckets   # compress coeffi
                 return hash_code % num_buckets                               # one last compre

             def size(self):
                 return self.num_entries

             def _rehash(self):
                 old_num_buckets = len(self.bucket_array)
                 old_bucket_array = self.bucket_array
                 num_buckets = 2 * old_num_buckets
                 self.bucket_array = [None for _ in range(num_buckets)]

                 for head in old_bucket_array:
                     while head is not None:
                         key = head.key
                         value = head.value
                         self.put(key, value)         # we can use our put() method to rehash
                         head = head.next

             def delete(self, key):
                 bucket_index = self.get_bucket_index(key)
                 head = self.bucket_array[bucket_index]

                 previous = None
                 while head is not None:
                     if head.key == key:
                         if previous is None:
                             self.bucket_array[bucket_index] = head.next
                         else:
                             previous.next = head.next
                         self.num_entries -= 1
                         return
                     else:
                         previous = head
                         head = head.next
```

```
In [71]: hash_map = HashMap(7)

         hash_map.put("one", 1)
         hash_map.put("two", 2)
         hash_map.put("three", 3)
         hash_map.put("neo", 11)

         print("size: {}".format(hash_map.size()))


         print("one: {}".format(hash_map.get("one")))
         print("neo: {}".format(hash_map.get("neo")))
         print("three: {}".format(hash_map.get("three")))
         print("size: {}".format(hash_map.size()))

         hash_map.delete("one")

         print(hash_map.get("one"))
         print(hash_map.size())
```

```
size: 4
one: 1
neo: 11
three: 3
size: 4
None
3
```

```
In [ ]:
```