Add/Remove & Search in Hash Table (Implementation)

This lesson will cover the Pythonic implementation for search, insertion, and deletion in hash tables.

We'll cover the following

In the previous lesson, we built a `HashTable` class and designed a hash function. Using that code, we will implement the main functionalities of a hash table.

Resizing in a Hash Table#

To start things off, we will make sure that the hash table doesn't get loaded up beyond a certain threshold. Whenever it crosses the threshold, we shift the elements from the current table to a new table with double the capacity. This helps us avoid collisions.

To implement this, we will make the `resize()` function.

> Have a look at `resize()` function in `HashTable.py`.

main.py

HashTable.py

HashEntry.py

```
from HashEntry import HashEntry


class HashTable:
    # Constructor
    def __init__(self):
        # Size of the HashTable
        self.slots = 10
        # Current entries in the table
        # Used while resizing the table when half of the table gets f
        self.size = 0
        # List of HashEntry objects (by default all None)
        self.bucket = [None] * self.slots
        self.threshold = 0.6

    # Helper Functions
    def get_size(self):
        return self.size

    def is_empty(self):
        return self.get_size() is 0

    # Hash Function
    def get_index(self, key):
        # hash is a built in function in Python
        hash_code = hash(key)
        index = hash_code % self.slots
        return index
```

Insertion in Table#

Insertion in hash tables is a simple task and it usually takes a constant amount of time. When the hash function returns the index for our input key, we check if there is a hash entry already present at that index (if it does, a collision has occurred). If not, we simply create a new hash entry for the key/value. However, if the index is not `None`, we will traverse through the bucket to check if an object with our key exists.

It is possible that the key we are inserting already exists. In this case, we will simply update the value. Otherwise, we add the new entry at the end of the bucket. The average cost of insertion is O(1). However, the worst case is O(n) as for some cases, the entire bucket needs to be traversed where all n elements are in a single bucket.

After each insertion, we will also check if the hash table needs resizing. The `threshold` will be a data member of the `HashTable` class with a fixed value of 0.6.

> Have a look at `insert()` function in `HashTable.py`.

main.py

HashTable.py

HashEntry.py

```
from HashEntry import HashEntry


class HashTable:
    # Constructor
    def __init__(self):
        # Size of the HashTable
        self.slots = 10
        # Current entries in the table
        # Used while resizing the table when half of the table gets filled
        self.size = 0
```

```
        # List of HashEntry objects (by default all None)
        self.bucket = [None] * self.slots
        self.threshold = 0.6

    # Helper Functions
    def get_size(self):
        return self.size

    def is_empty(self):
        return self.get_size() is 0

    # Hash Function
    def get_index(self, key):
        # hash is a built in function in Python
        hash_code = hash(key)
        index = hash_code % self.slots
        return index

    def resize(self):
        new_slots = self.slots * 2
        new_bucket = [None] * new_slots
        # rehash all items into new slots
        for item in self.bucket:
            head = item
            while head is not None:
                new_index = hash(head.key) % new_slots
                if new_bucket[new_index] is None:
                    new_bucket[new_index] = HashEntry(head.key, head.value)
                else:
                    node = new_bucket[new_index]
                    while node is not None:
                        if node.key is head.key:
                            node.value = head.value
                            node = None
                        elif node.nxt is None:
                            node.nxt = HashEntry(head.key, head.value)
                            node = None
                        else:
                            node = node.nxt
                head = head.nxt
        self.bucket = new_bucket
        self.slots = new_slots

    def insert(self, key, value):
        # Find the node with the given key
        b_index = self.get_index(key)
        if self.bucket[b_index] is None:
            self.bucket[b_index] = HashEntry(key, value)
            print(key, "-", value, "inserted at index:", b_index)
            self.size += 1
        else:
            head = self.bucket[b_index]
```

```
        while head is not None:
            if head.key == key:
                head.value = value
                break
            elif head.nxt is None:
                head.nxt = HashEntry(key, value)
                print(key, "-", value, "inserted at index:", b_index)
                self.size += 1
                break
            head = head.nxt

    load_factor = float(self.size) / float(self.slots)
    # Checks if 60% of the entries in table are filled, threshold = 0.6
    if load_factor >= self.threshold:
        self.resize()
```

Search in a Hash Table

One of the features that make hash tables efficient is that search takes *O(1)* amount of time. The `search` function takes in a `key` and sends it through the hash function to get the corresponding index in the table. If a hash entry with the desired key/value pair is found at that index, its value is returned. Search can take up to *O(n)* time, where **n** is the number of hash entries in the table. This is possible if all values get stored in the same bucket, then we would have to traverse the whole bucket to reach the entry.

> Have a look at `search()` function in `HashTable.py`.

main.py

HashTable.py

HashEntry.py

```python
from HashEntry import HashEntry


class HashTable:
    # Constructor
    def __init__(self):
        # Size of the HashTable
        self.slots = 10
        # Current entries in the table
        # Used while resizing the table when half of the table gets filled
        self.size = 0
        # List of HashEntry objects (by default all None)
        self.bucket = [None] * self.slots
        self.threshold = 0.6

    # Helper Functions
    def get_size(self):
        return self.size

    def is_empty(self):
        return self.get_size() is 0

    # Hash Function
    def get_index(self, key):
        # hash is a built in function in Python
        hash_code = hash(key)
        index = hash_code % self.slots
        return index

    def resize(self):
        new_slots = self.slots * 2
        new_bucket = [None] * new_slots
        # rehash all items into new slots
        for item in self.bucket:
            head = item
            while head is not None:
                new_index = hash(head.key) % new_slots
                if new_bucket[new_index] is None:
                    new_bucket[new_index] = HashEntry(head.key, head.value)
                else:
                    node = new_bucket[new_index]
                    while node is not None:
                        if node.key is head.key:
                            node.value = head.value
                            node = None
                        elif node.nxt is None:
                            node.nxt = HashEntry(head.key, head.value)
                            node = None
                        else:
                            node = node.nxt
                head = head.nxt
        self.bucket = new_bucket
```

```
        self.slots = new_slots

    def insert(self, key, value):
        # Find the node with the given key
        b_index = self.get_index(key)
        if self.bucket[b_index] is None:
            self.bucket[b_index] = HashEntry(key, value)
            print(key, "-", value, "inserted at index:", b_index)
            self.size += 1
        else:
            head = self.bucket[b_index]
            while head is not None:
                if head.key == key:
                    head.value = value
                    break
                elif head.nxt is None:
                    head.nxt = HashEntry(key, value)
                    print(key, "-", value, "inserted at index:", b_index)
                    self.size += 1
                    break
                head = head.nxt

        load_factor = float(self.size) / float(self.slots)
        # Checks if 60% of the entries in table are filled, threshold = 0.6
        if load_factor >= self.threshold:
            self.resize()
        # Return a value for a given key

    def search(self, key):
        # Find the node with the given key
        b_index = self.get_index(key)
        head = self.bucket[b_index]
        # Search key in the slots
        while head is not None:
            if head.key == key:
                return head.value
            head = head.nxt
        # If key not found
        return None
```

Deletion in Table

Deletion can take up to *O(n)* time where **n** is the number of hash entries in the table. If they all get stored in the same bucket, we would have to traverse the whole bucket to reach the entry we want to delete. The a e however, is still O(1).

> Have a look at `delete()` function in `HashTable.py`.

main.py

HashTable.py

HashEntry.py

```
from HashEntry import HashEntry


class HashTable:
    # Constructor
    def __init__(self):
        # Size of the HashTable
        self.slots = 10
        # Current entries in the table
        # Used while resizing the table when half of the table gets filled
        self.size = 0
        # List of HashEntry objects (by default all None)
        self.bucket = [None] * self.slots
        self.threshold = 0.6

    # Helper Functions
    def get_size(self):
        return self.size

    def is_empty(self):
        return self.get_size() is 0

    # Hash Function
    def get_index(self, key):
        # hash is a built in function in Python
        hash_code = hash(key)
        index = hash_code % self.slots
        return index

    def resize(self):
        new_slots = self.slots * 2
        new_bucket = [None] * new_slots
        # rehash all items into new slots
```

```
        for item in self.bucket:
            head = item
            while head is not None:
                new_index = hash(head.key) % new_slots
                if new_bucket[new_index] is None:
                    new_bucket[new_index] = HashEntry(head.key, head.value)
                else:
                    node = new_bucket[new_index]
                    while node is not None:
                        if node.key is head.key:
                            node.value = head.value
                            node = None
                        elif node.nxt is None:
                            node.nxt = HashEntry(head.key, head.value)
                            node = None
                        else:
                            node = node.nxt
                head = head.nxt
        self.bucket = new_bucket
        self.slots = new_slots

    def insert(self, key, value):
        # Find the node with the given key
        b_index = self.get_index(key)
        if self.bucket[b_index] is None:
            self.bucket[b_index] = HashEntry(key, value)
            print(key, "-", value, "inserted at index:", b_index)
            self.size += 1
        else:
            head = self.bucket[b_index]
            while head is not None:
                if head.key == key:
                    head.value = value
                    break
                elif head.nxt is None:
                    head.nxt = HashEntry(key, value)
                    print(key, "-", value, "inserted at index:", b_index)
                    self.size += 1
                    break
                head = head.nxt

        load_factor = float(self.size) / float(self.slots)
        # Checks if 60% of the entries in table are filled, threshold = 0.6
        if load_factor >= self.threshold:
            self.resize()

    # Return a value for a given key
    def search(self, key):
        # Find the node with the given key
        b_index = self.get_index(key)
        head = self.bucket[b_index]
        # Search key in the slots
```

```
        while head is not None:
            if head.key == key:
                return head.value
            head = head.nxt
        # If key not found
        return None

    # Remove a value based on a key
    def delete(self, key):
        # Find index
        b_index = self.get_index(key)
        head = self.bucket[b_index]
        # If key exists at first slot
        if head.key == key:
            self.bucket[b_index] = head.nxt
            print(key, "-", head.value, "deleted")
            # Decrease the size by one
            self.size -= 1
            return self
        # Find the key in slots
        prev = None
        while head is not None:
            # If key exists
            if head.key == key:
                prev.nxt = head.nxt
                print(key, "-", head.value, "deleted")
                # Decrease the size by one
                self.size -= 1
                return
            # Else keep moving in chain
            prev = head
            head = head.nxt

        # If key does not exist
        print("Key not found")
        return
```

In the next lesson, we will bring all these functions together to create our complete `HashTable` class.

companies apply to you instead of you applying to them. See how

Back

Building a Hash Table from Scratch

Next

A Quick Overview of Hash Tables

Mark as Completed

Report an Issue