Hashing | Data Structures

Hashing is the process of transforming any given key or a string of characters into another value. It is a searching technique that requires constant time to find a value.

1. Implementing Hash Table

```
class HashTable:
    def __init__(self): #Creates a hashtable
        self.MAX = 10
        self.arr = [None for i in range(self.MAX)]

    def get_hash(self, key): #Maps the hash values
        hash = 0
        for char in key:
            hash += ord(char)
        return hash % self.MAX

    def __getitem__(self, index): #Getting the values after hash mapping
        h = self.get_hash(index)
        return self.arr[h]

    def __setitem__(self, key, val): #Creates the hash value for the indices
        h = self.get_hash(key)
        self.arr[h] = val

    def __delitem__(self, key): #Deletes a value after hash mapping
        h = self.get_hash(key)
        self.arr[h] = None
```

2. Setting values into the Hash Table

```
t = HashTable()
t["march 6"] = 310 #Setting a value, calls the __setitem__ method
t["march 7"] = 420 #Setting a value, calls the __setitem__ method
```

```
t.arr #Let us see what changes are made into the hash table
```

```
[420, None, None, None, None, None, None, None, None, 310]
```

3. Getting a value

```
print(t["march 6"]) #Getting a value, calls the __getitem__ method
```

```
310
```

4. Deleting a hash value

```
del t["march 6"] #Deleting a value, calls the __delitem__ method
```

```
t.arr #Let us see what changes are made into the hash table
```

```
[420, None, None, None, None, None, None, None, None, None]
```

5. Collisions

```
t["march 6"] = 305 #Setting a value, calls the __setitem__ method
t["march 17"] = 369 #Setting a value, calls the __setitem__ method
```

```
t.arr
```

```
[420, None, None, None, None, None, None, None, None, 369]
```

Here, “march 6” and “march 17” both are hashed to index 9. Therefore, “march 6” gets replaced.

Exercise

Write a python program that creates a hash table of length 15 and a hashing technique that can map the hash values properly. Your program should include Get, Set, and Delete operations. Assume that no collision will occur. While hashing, you have to implement the following conditions:

a. For lower-case letters, take the ascii code of the upper case of the previous letter. For example, if you encounter a, you have to take 90 (ascii of Z). Again, if you get b, you have to take 65 (ascii of A).
b. for upper-case letters, take their ascii codes. For example, if you encounter A, you have to take 65 (ascii of A).
c. For numbers (0-9), take the integer value of that number. For example, if you encounter “2”, take 2.
d. For all other characters, take 5. For example, if you encounter $, take 5.

Hashing | Data Structures

6. Collision Handling

a. Chaining

Regular Hashing	Chaining

No Node class needed.	class Node:
	  def __init__(self, elem, key=None, next=None):
	  self.elem=elem
	  self.next=next
	  self.key=key

def __init__(self):
  self.MAX = 10
  self.arr = [None for i in range(self.MAX)]
	def __init__(self):
	  self.MAX = 10
	  self.arr = [None for i in range(self.MAX)]

def get_hash(self, key):
  hash = 0
  for char in key:
  hash += ord(char)
  return hash % self.MAX
	def get_hash(self, key):
	  hash = 0
	  for char in key:
	  hash += ord(char)
	  return hash % self.MAX

def __getitem__(self, key):
  h = self.get_hash(key)
  return self.arr[h]
	def __getitem__(self, key):
	  hash = self.get_hash (key)
	  head= self.arr [hash]
	  while head != None:
	  if (head.key == key):
	  return head.elem
	  head = head.next
	  return "Not Found"

def __setitem__(self, key, val):
  h = self.get_hash(key)
  self.arr[h] = val
	def __setitem__(self, key, val):
	  h = self.get_hash (key)
	  print(h, key)
	  if (self.arr[h] == None):
	  self.arr[h] = Node (val, key)
	  else:
	  x= Node (val, key)
	  x.next = self.arr [h]
	  self.arr[h] =x

def __delitem__(self, key):
  h = self.get_hash(key)
  self.arr[h] = None
	def __delitem__(self, key):
	  h = self.get_hash(key)
	  parent= None
	  child= None
	  n= self.arr[h]
	  if (n.next == None):
	  n= None
	  else:
	  while (n.next != None):
	  if n.next.key == key:
	  parent = n
	  child = n.next.next
	  parent.next = child
	  break
	  n= n.next
	  print(f"{key} deleted!")

b. Linear Probing

Regular Hashing	Linear Probing

No Node class needed.	class Node:
	  def __init__(self, elem, key=None, next=None):
	  self.elem=elem
	  self.next=next
	  self.key=key

def __init__(self):
  self.MAX = 10
  self.arr = [None for i in range(self.MAX)]
	def __init__(self):
	  self.MAX = 10
	  self.arr = [None for i in range(self.MAX)]

def get_hash(self, key):
  hash = 0
  for char in key:
  hash += ord(char)
  return hash % self.MAX
	def get_hash(self, key):
	  hash = 0
	  for char in key:
	  hash += ord(char)
	  return hash % self.MAX

def __getitem__ (self, key):
  h = self.get_hash(key)
  return self.arr[h]
	def __getitem__(self, key):
	  h = self.get_hash (key)
	  for i in range (0, self.MAX):
	  if self.arr[(h+i) % self.MAX] != None:
	  if self.arr [(h+i) % self.MAX].key == key:
	  return self.arr [(h+i) % self.MAX].elem
	  return f"Not found"

def __setitem__(self, key, val):
  h = self.get_hash(key)
  self.arr[h] = val
	def __setitem__(self, key, val):
	  h = self.get_hash (key)
	  for i in range (0, self.MAX):
	  if (self.arr[(h+i) % self.MAX]) == None:
	  self.arr[(h+i) % self.MAX] = Node (val,key)
	  print(h, self.arr [(h+i) % self.MAX].key)
	  break

def __delitem__(self, key):
  h = self.get_hash(key)
  self.arr[h] = None
	def __delitem__(self, key):
	  h = self.get_hash (key)
	  for i in range (0, self.MAX):
	  if self.arr [(h+i) % self.MAX] != None:
	  if self.arr [(h+i) % self.MAX].key == key:
	  self.arr [(h+i) % self.MAX] = None
	  print(f"{key} deleted!")

Exercise

Write two python programs that create a hash table of length 12 and a hashing technique that can map the hash values properly. Your program should include Get, Set, and Delete operations. **Resolve collision using Chaining and Linear Probing.** While hashing, you have to implement the following conditions:

a. For lower-case letters, take the ascii code of the upper case of the previous letter. For example, if you encounter a, you have to take 90 (ascii of Z). Again, if you get b, you have to take 65 (ascii of A).
b. for upper-case letters, take their ascii codes. For example, if you encounter A, you have to take 65 (ascii of A).
c. For numbers (0-9), take the integer value of that number. For example, if you encounter “2”, take 2.
d. For all other characters, take 5. For example, if you encounter $, take 5.