006 - Tables

A Table is an ***unordered collection of records***. Each record consists of a key-value pair. Within the same table, keys are unique. That is only one record in the table may have a certain key. Values do not have to be unique.

Simple Implementation using arrays

```python
class Table:
    def __init__(self):
        self.table = []
        arr = []
    def insert(self, key, value):
        low = 0
        high = len(self.table) - 1
        while low <= high:
            mid = (low + high) // 2
            if self.table[mid][0] == key:
                self.table[mid] = (key, value)
                return
            elif self.table[mid][0] < key:
                low = mid + 1
            else:
                high = mid - 1
        self.table.insert(low, (key, value))
```

```python
    def remove(self, key):
        low = 0
        high = len(self.table) - 1
        while low <= high:
            mid = (low + high) // 2
            if self.table[mid][0] == key:
                del self.table[mid]
                return
            elif self.table[mid][0] < key:
                low = mid + 1
            else:
                high = mid - 1
    def search(self, key):
        low = 0
        high = len(self.table) - 1
        while low <= high:
            mid = (low + high) // 2
            if self.table[mid][0] == key:
                return self.table[mid][1]
            elif self.table[mid][0] < key:
                low = mid + 1
            else:
                high = mid - 1
        return None
```

In the above implementation, searching has a time complexity of O(log n), however, insertion and deletion have a time complexity of O(n).

Hash Tables:

A hash table is a data structure that uses a hash function to map keys to indices in an array. It provides efficient insertion, deletion, and retrieval of records based on their keys. The underlying array acts as "buckets" or "slots" where the records are stored.

Hash Functions:

A hash function takes a key as input and returns a unique numeric value, which is used to determine the index in the array where the record will be stored. The ideal hash function should produce uniformly distributed hash values for different keys, ensuring an even spread of records across the array.

In an example of storing customer information using telephone numbers, a hash function could be designed to use the last four digits of the phone number as the hash value, as they are more likely to provide variation between customers.

Note:

It is actually a defining feature of all hash functions that they greatly reduce any possible inputs (any string you can imagine) into a much smaller range of potential outputs (an integer smaller than the size of our array). For this reason, hash functions are also known as *compression functions*.

Much like an image that has been shrunk to a lower resolution, the output of a hash function contains less data than the input. Because of this, hashing is not a reversible process. With just a hash value it is impossible to know for sure the key that was plugged into the hashing function.

Load Factor:

The load factor of a hash table is a measure of how full the table is. It is calculated as the ratio of the number of records in the table to the total number of locations (or buckets) in the array. A high load factor indicates that the table is approaching its capacity and may need resizing to maintain efficiency.

> 💡 **Load Factor (λ) = number of records in table / number of locations**

Recipe for saving to a hash table:

- Take the key and plug it into the hash function, getting the hash code.
- Modulo that hash code by the length of the underlying array, getting an array index.
- Check if the array at that index is empty, if so, save the value (and the key) there.
- If the array is full at that index continue to the next possible position depending on your collision strategy.

Recipe for retrieving from a hash table:

- Take the key and plug it into the hash function, getting the hash code.
- Modulo that hash code by the length of the underlying array, getting an array index.
- Check if the array at that index has contents, if so, check the key saved there.
- If the key matches the one you're looking for, return the value.
- If the keys don't match, continue to the next position depending on your collision strategy.

Collisions:

Collisions occur when two or more keys hash to the same index in the array. Since the number of possible keys is often greater than the number of available locations in the array, collisions are inevitable. Dealing with collisions is an important aspect of implementing hash tables effectively.

Pigeon Hole Principle:

The pigeon hole principle is a mathematical concept that states that if you have more items to distribute than available slots, there will be at least one slot with multiple items. In the context of hash tables, it means that with a limited number of array locations (slots) and a larger number of possible keys, collisions will occur.

When collisions happen, it is necessary to have strategies for resolving them efficiently. There are various methods to handle collisions, including:

1. **Separate Chaining:** Each slot in the array contains a linked list or another data structure. When a collision occurs, new records are appended to the list at that location.
2. **Open Addressing:** In this approach, when a collision occurs, the algorithm probes for the next available slot in the array to store the record. This is done using techniques like linear probing, quadratic probing, or double hashing.

   Linear Probing only allows one item at each element. There is no second dimension to look. Linear probing is an example of open addressing. Open addressing collision resolution methods allow an item to be placed at a different spot other than what the hash function dictates. Aside from linear probing, other open addressing methods include quadratic probing and double hashing.

   A ***cluster*** is a group of records without any empty spots. Thus, any search begins with a hashindex within a cluster searches to the end of the cluster.
3. **Robin Hood Hashing:** This is a variation of open addressing where records are rearranged in such a way that records with longer probe sequences (i.e., those that had more collisions) are placed closer to the beginning of the probe sequence. This reduces the average search time.

The choice of collision resolution method depends on factors such as the expected number of records, the distribution of keys, and the desired performance characteristics of the hash table.

By using an appropriate hash function and implementing a collision resolution strategy, hash tables can provide efficient operations with a time complexity close to O(1) for insertion, deletion, and retrieval, making them powerful data structures for many applications.

HashMap Using Seperate Chaining

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

class HashTable:
    def __init__(self, capacity):
        self.capacity = capacity
        self.table = [None] * self.capacity

    def hash_function(self, key):
        return hash(key) % self.capacity

    def insert(self, key, value):
        index = self.hash_function(key)
        if self.table[index] is None or self.table[index].key == key:
            self.table[index] = Node(key, value)
            return
        else:
            current = self.table[index]
            while current.next:
                if current.key == key:
                    current = Node(key, value)
                current = current.next
            current.next = Node(key, value)
```

```python
    def delete(self, key):
        index = self.hash_function(key)
        if self.table[index] is None:
            return
        if self.table[index].key == key:
            self.table[index] = self.table[index].next
            return
        curr = self.table[index]
        while curr.next:
            if curr.next.key == key:
                curr.next = curr.next.next
                return
            curr = curr.next

    def search(self, key):
        index = self.hash_function(key)
        curr = self.table[index]
        while curr:
            if curr.key == key:
                return curr.value
            curr = curr.next
        return None

hash_map = HashTable(20)
hash_map.insert("A", "Apple")
hash_map.insert("B", "Ball")
hash_map.insert("C", "Cat")
hash_map.insert("D", "Dog")
hash_map.insert("D", "Daaru")
print(hash_map.search("C"))
print(hash_map.search("D"))
hash_map.delete("D")
print(hash_map.search("D"))
```

HashMap Using Linear Probing without tombstones

```python
class LinearProbingNoTS:
  class Record:
    def __init__(self, key = None, value = None):
      self.key = key
      self.value = value

  def my_hash(self, key):
    return sum(key.encode())

  # The initializer for the table defaults the initial table capacity to 32
  def __init__(self, cap = 32):
    self.cap = cap
    self.hash_table = [None for _ in range(cap)]
    self.elements = 0

  # This function adds a new key-value pair into the table
  def insert(self, key, value):
    if self.elements < self.cap:
      hashed_index = self.my_hash(key) % self.cap
      if self.hash_table[hashed_index] is None:
        self.hash_table[hashed_index] = self.Record(key, value)
        self.elements += 1
        self.resize()
        return True
      else:
        while self.hash_table[hashed_index] is not None:
          if self.hash_table[hashed_index].key == key:
            return False
          hashed_index = (hashed_index + 1) % self.cap
        self.hash_table[hashed_index] = self.Record(key, value)
        self.elements += 1
        self.resize()
        return True
    return False
```

```python
  # This function modifies an existing key-value pair into the table
  def modify(self, key, value):
    hashed_index = self.my_hash(key) % self.cap
    if self.hash_table[hashed_index] is not None and self.hash_table[hashed_index].key == key:
      self.hash_table[hashed_index].value = value
      return True
    else:
      while self.hash_table[hashed_index] is not None:
        hashed_index = (hashed_index + 1) % self.cap
        if self.hash_table[hashed_index] is not None and self.hash_table[hashed_index].key == key:
          self.hash_table[hashed_index].value = value
          return True
    return False

  # This function removes the key-value pair with the matching key
  def remove(self, key):
    hashed_index = self.my_hash(key) % self.cap
    curr_index = hashed_index
    while self.hash_table[curr_index] is not None:
      if self.hash_table[curr_index].key == key:
        self.hash_table[curr_index] = None
        empty_index = curr_index
        next_index = (empty_index + 1) % self.cap
        while self.hash_table[next_index] is not None:
          if (empty_index >= self.my_hash(self.hash_table[next_index].key) % self.cap) and (empty_index < next_index):
            self.hash_table[empty_index] = self.hash_table[next_index]
            self.hash_table[next_index] = None
            empty_index = next_index
          next_index = (next_index + 1) % self.cap
        self.elements -= 1
        return True
      curr_index = (curr_index + 1) % self.cap
    return False
```

```python
  # This function returns the value of the record with the matching key
  def search(self, key):
    hashed_index = self.my_hash(key) % self.cap
    if self.hash_table[hashed_index] is not None and self.hash_table[hashed_index].key == key:
      return self.hash_table[hashed_index].value
    else:
      while self.hash_table[hashed_index] is not None:
        hashed_index = (hashed_index + 1) % self.cap
        if self.hash_table[hashed_index] is not None and self.hash_table[hashed_index].key == key:
          return self.hash_table[hashed_index].value
    return None

  # This function returns the number of spots in the table
  def capacity(self):
    return self.cap

  # This function returns the number of Records stored in the table
  def __len__(self):
    return self.elements

  # This function grows the underlying array used to implement the hash table
  def resize(self):
    if ((self.elements / self.cap) >= 0.7):
      self.cap *= 2
      old_table = self.hash_table
      self.hash_table = [None for _ in range(self.cap)]
      self.elements = 0
      for record in old_table:
        if record is not None:
          self.insert(record.key, record.value)
```

HashMap Using Linear Probing with tombstones

```python
class LinearProbingTS:
  class Record:
    def __init__(self, key = None, value = None, status = 'E'): # Status: 'E' => Empty, 'X' => Deleted, 'O' => Occupied
      self.key = key
      self.value = value
      self.status = status

  # The initializer for the table defaults the initial table capacity to 32
  def __init__(self, cap = 32):
    self.cap = cap
    self.hash_table = [self.Record(None, None, 'E') for _ in range(cap)]
    self.elements = 0

  def my_hash(self, key):
    return sum(key.encode())

  # This function adds a new key-value pair into the table
  def insert(self, key, value):
    if self.elements < self.cap:
      hashed_index = self.my_hash(key) % self.cap
      if self.hash_table[hashed_index].status != 'O' or self.hash_table[hashed_index].status == 'X':
        self.hash_table[hashed_index].key = key
        self.hash_table[hashed_index].value = value
        self.hash_table[hashed_index].status = 'O'
        self.elements += 1
        if ((self.elements / self.cap) >= 0.7):
          self.resize()
        return True
      else:
        while self.hash_table[hashed_index].status == 'O':
          if self.hash_table[hashed_index].key == key:
            return False
          hashed_index = (hashed_index + 1) % self.cap
        self.hash_table[hashed_index].key = key
        self.hash_table[hashed_index].value = value
        self.hash_table[hashed_index].status = 'O'
        self.elements += 1
```

```
    if ((self.elements / self.cap) >= 0.7):
     self.resize()
    return True
  return False

 # This function modifies an existing key-value pair into the table
 def modify(self, key, value):
  hashed_index = self.my_hash(key) % self.cap
  orig_index = hashed_index
  if self.hash_table[hashed_index].status == 'O' and self.hash_table[hashed_in
dex].key == key:
   self.hash_table[hashed_index].value = value
   return True
  else:
   while self.hash_table[hashed_index].status != 'E' and self.hash_table[hash
ed_index].status != 'X':
    hashed_index = (hashed_index + 1) % self.cap
    if hashed_index == orig_index:
     break
    if self.hash_table[hashed_index].key == key:
     self.hash_table[hashed_index].value = value
     return True
  return False

 # This function removes the key-value pair with the matching key
 def remove(self, key):
  hashed_index = self.my_hash(key) % self.cap
  curr_index = hashed_index
  if self.hash_table[curr_index].key == key and self.hash_table[curr_index].st
atus != 'X':
    self.hash_table[curr_index].status = 'X'
    self.elements -= 1
    return True
  else:
   while self.hash_table[curr_index].status != 'E':
    curr_index = (curr_index + 1) % self.cap
    if hashed_index == curr_index:
     break
    if self.hash_table[curr_index].key == key and self.hash_table[curr_inde
x].status != 'X':
     self.hash_table[curr_index].status = 'X'
     self.elements -= 1
```

```python
        return True
    else:
      while self.hash_table[curr_index].status != 'E':
        curr_index = (curr_index + 1) % self.cap
        if hashed_index == curr_index:
          break
        if self.hash_table[curr_index].key == key and self.hash_table[curr_inde
x].status != 'X':
          self.hash_table[curr_index].status = 'X'
          self.elements -= 1
          return True
    return False


  # This function returns the value of the record with the matching key
  def search(self, key):
    hashed_index = self.my_hash(key) % self.cap
    orig_index = hashed_index
    if self.hash_table[hashed_index].status != 'E' and self.hash_table[hashed_in
dex].status != 'X' and self.hash_table[hashed_index].key == key:
      return self.hash_table[hashed_index].value
    else:
      while self.hash_table[hashed_index].status != 'E':
        hashed_index = (hashed_index + 1) % self.cap
        if hashed_index == orig_index:
          break
        if self.hash_table[hashed_index].status == 'O' and self.hash_table[hashe
d_index].key == key:
          return self.hash_table[hashed_index].value
    return None

  # This function returns the number of spots in the table
  def capacity(self):
    return self.cap

  # This function returns the number of Records stored in the table
  def __len__(self):
    return self.elements
```

```python
  # This function grows the underlying array used to implement the hash table
  def resize(self):
    self.cap *= 2
    old_table = self.hash_table
    self.hash_table = [self.Record(None, None, 'E') for _ in range(self.cap)]
    self.elements = 0
    for record in old_table:
      if record.status == 'O':
        self.insert(record.key, record.value)
```