Hash Table

1. Introduction

Recap - Dictionary

Dictionary is a built-in data structure which store data in key-value pair.

- Values are accessed by key.
- Keys must be unique in a dictionary.

Construct a Dictionary

In [1]:

```
d = {'name':'Mark', 'gender':'Male', 'address':'Singapore, Earth'}
```

Access an Item

In [2]:

```
d['name']
```

Update an Item

In [3]:

```
d['name'] = 'Markov'
```

Add an Item

In [4]:

```
d['age'] = 18
```

Hashable Key

If you try to use a list as a key, it will throw a `TypeError` exception.

This shows that a Dictionary internally uses a **hash table** structure to store data.

In [5]:

```
# d[['height','weight']] = [1.69, 72]
```

Hash Table

Hash table is data structure that maps **keys** to **values (data)**. This is similar to a dictionary.

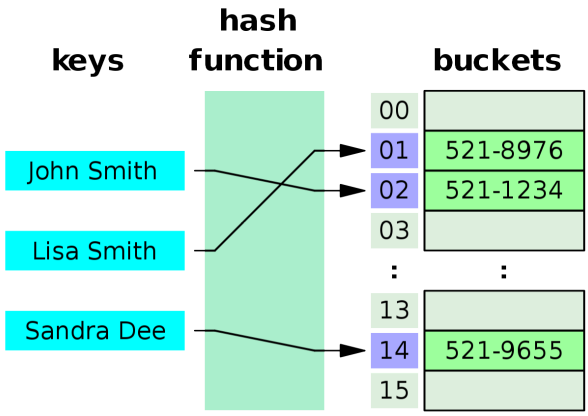
For example, to store a phone book, you uses a person's name as key, and his phone number is the data to be looked up.

Hash function is function which takes in a value and generates another value.

- Key is passed into the **hash function** to generate an index value which points to a location where data is stored.

Bucket is the place where data is stored.

- Potentially multiple data may be stored in the same bucket, i.e. multiple keys may point to same bucket.



https://en.wikipedia.org/wiki/Hash_table

2. Basic Hash Table

Let's implement a hash table for a phone book. Each entry in the phone book is a pair of `Name` and `Phone`.

- `Name` is used as the key.
- `(Name, Phone)` tuple is saved as the data.

Hash Table

We will define a class `HashTable` to store the data.

- It has a list attribute `buckets` which keeps all data.
- Initialize the list size, i.e. how many buckets, by input parameter `size`.
- It has a static function `_hash()` which returns an `index` value based on input parameter `key`.
- The `index` value specifies which bucket to put the data.

Hash Function

The logic to be implemented in `_hash()` function is straight forward. We will simply return length of the `key` as the `index` value.

In [6]:

```
class HashTable:

    def __init__(self, size):
        self.buckets = [None]*size

    @staticmethod
    def _hash(key):
        return len(key)
```

Test

Test - Add Items

Let's try to add following items into the HashTable.

- Create a hash table of 10 buckets.
- For each contact,
  - Use `_hash()` function to find out which bucket it belongs to;
  - Put the contact in the bucket.
- Print out the `buckets` to view how contacts are stored.

```
contacts = [
    ('Ben', '357-0394'),
    ('Alan', '558-9171'),
    ('Freddi', '760-2466'),
    ('Stephanie', '299-5109')]
```

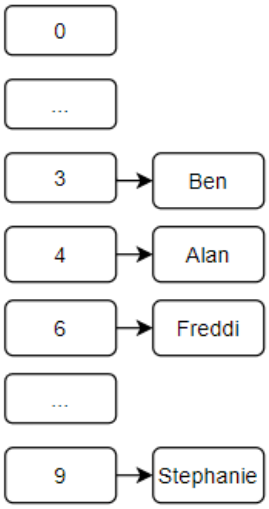
In [7]:

```
contacts = [
    ('Ben', '357-0394'),
    ('Alan', '558-9171'),
    ('Freddi', '760-2466'),
    ('Stephanie', '299-5109')]

table = HashTable(10)
for c in contacts:
    i = HashTable._hash(c[0])
    table.buckets[i] = c

print(table.buckets)
```

```
[None, None, None, ('Ben', '357-0394'), ('Alan', '558-9171'), None, ('Fredd
i', '760-2466'), None, None, ('Stephanie', '299-5109')]
```



In this case, the time spent in finding an item is **O(1)**.

Test - Find an Item

With the populated hash table, how do you retrieve the data of for a name, e.g. `'Freddi'` ?

- Use `_hash()` function to find `index` value.
- Locate the bucket by index.
- Return the bucket.

In [8]:

```
idx = HashTable._hash('Freddi')
data = table.buckets[idx]
print(data)
```

```
('Freddi', '760-2466')
```

Test - Remove an Item

We may need to remove an item, e.g. `'Freddi'` , from the hash table.

- Use _hash() function to find index value.
- Locate the bucket by index and set it to `None` .

In [9]:

```
idx = HashTable._hash('Freddi')
table.buckets[idx] = None
print(table.buckets)
```

```
[None, None, None, ('Ben', '357-0394'), ('Alan', '558-9171'), None, None, No
ne, None, ('Stephanie', '299-5109')]
```

Support Basic Operations

A Hash Table class commonly implement methods to support **add**, **find** and **remove** operations.

With knowledge of previous session, Enhance `HashTable` class by implementing `add(key, data)`, `find(key)` and `remove(key)` methods.

In [10]:

```
class HashTable:

    def __init__(self, size):
        self.buckets = [None]*size

    @staticmethod
    def _hash(key):
        return len(key)

    def add(self, key, data):
        i = HashTable._hash(key)
        self.buckets[i] = data

    def find(self, key):
        i = HashTable._hash(key)
        return self.buckets[i]

    def remove(self, key):
        i = HashTable._hash(key)
        self.buckets[i] = None

```

Test:

In [11]:

```
contacts = [
    ('Ben', '357-0394'),
    ('Alan', '558-9171'),
    ('Freddi', '760-2466'),
    ('Stephanie', '299-5109')]

table = HashTable(10)
for c in contacts:
    table.add(c[0], c)

print(table.buckets)
```

```
[None, None, None, ('Ben', '357-0394'), ('Alan', '558-9171'), None, ('Fredd
i', '760-2466'), None, None, ('Stephanie', '299-5109')]
```

In [12]:

```
table.find('Freddi')
```

Out[12]:

```
('Freddi', '760-2466')
```

In [13]:

```
table.remove('Freddi')
print(table.buckets)
```

```
[None, None, None, ('Ben', '357-0394'), ('Alan', '558-9171'), None, None, No
ne, None, ('Stephanie', '299-5109')]
```

3. Better Hash Table

Support Multiple-Items Bucket

What if we need store following data in the hash table?

```
contacts = [
    ('Amanda', '357-0394'),
    ('Christ', '558-9171'),
    ('Freddi', '760-2466'),
    ('Steven', '299-5109')]
```

Since all contacts' name has length of 6 characters, their hashed indexes point to the same bucket. Thus 6th bucket needs to be able to hold multiple contacts.

Since we still need to scan all items in a bucket, a **linked-list** implementation is more common because it is more memory efficient.

For simplicity, We will implement a bucket as a list.

Hash Node

The data can be any data type. To make Hash Table methods usable for any data type. It is better to use one common data type for item in the hash table.

We will create a class `HashNode` where each data element is stored in a `HashNode` object.

Define a `HashNode` class with instance attributes `key` and `data`.

- Implement its `__init__()` function to initialize `key & data`.
- Implement its `__str__()` function to return `data` in string format.
- Implement its `__repr__()` function to return same value as `__str__()`.
- Implement its `__eq__()` function to compare 2 nodes by their `key`.

In [14]:

```
class HashNode:

    def __init__(self, key, data):
        self.key = key
        self.data = data

    def __str__(self):
        return str(self.data)

    def __repr__(self):
        return self.__str__()

    def __eq__(self, other):
        return self.key == other.key
```

Hash Table with Hash Node

Modify the `HashTable` class with following enhancements:

- Implement each bucket as a list.
- Use `HashNode` to hold data

In [15]:

```
class HashTable:

    def __init__(self, size):
        self.buckets = [None]*size

    @staticmethod
    def _hash(key):
        return len(key)

    def add(self, key, data):
        node = HashNode(key, data)
        i = HashTable._hash(key)

        if self.buckets[i] is None:
            self.buckets[i] = [node]
        elif node in self.buckets[i]:
            print('Data exists')
            return False
        else:
            self.buckets[i].append(node)
            return True

    def find(self, key):
        i = HashTable._hash(key)
        node = HashNode(key,None)

        if node not in self.buckets[i]:
            return None
        idx = self.buckets[i].index(node)
        return self.buckets[i][idx]

    def remove(self, key):
        i = HashTable._hash(key)
        node = HashNode(key,None)

        if node not in self.buckets[i]:
            return
        self.buckets[i].remove(node)

```

Test:

Test the basic `add()`, `find()` and `remove()` functions.

In [16]:

```
contacts = [
    ('Amanda', '357-0394'),
    ('Christ', '558-9171'),
    ('Freddi', '760-2466'),
    ('Steven', '299-5109')]

table = HashTable(10)
for c in contacts:
    table.add(c[0], c)

print(table.buckets)
```

```
[None, None, None, None, None, None, [('Amanda', '357-0394'), ('Christ', '55
8-9171'), ('Freddi', '760-2466'), ('Steven', '299-5109')], None, None, None]
```

In [17]:

```
table.find('Freddi')
```

Out[17]:

```
('Freddi', '760-2466')
```

In [18]:

```
table.remove('Freddi')
print(table.buckets)
```

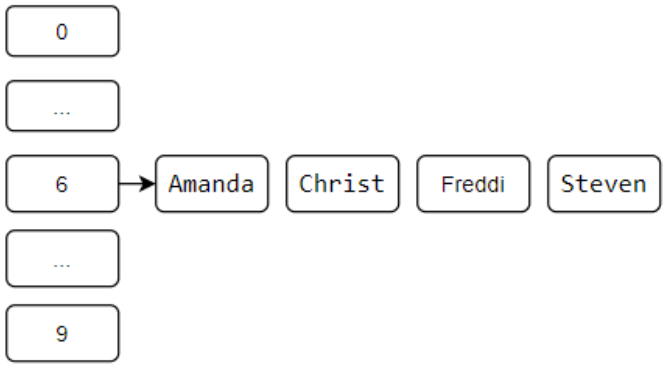
```
[None, None, None, None, None, None, [('Amanda', '357-0394'), ('Christ', '55
8-9171'), ('Steven', '299-5109')], None, None, None]
```

4. Importance of Hash Function

Hash Table Collision

Ideally, the hash function will assign each key to a unique bucket. But since a hash function returns a small number for a big key, there is possibility that two keys result in same value. That is **hash table collision**.

In previous example, the hash function generates same index value for all entries, and all data are stored in same bucket.



This is the worst case where a hash table acts a list and time spent in searching is **O(n)**. To improve efficiency, we need a better hash function.

Good Hash Function

To achieve a good hashing mechanism, It is important to have a good hash function with the following basic requirements:

Easy to Compute

- A hash function, should be easy to compute the unique keys.

Less Collision

- When elements equate to the same key values, there occurs a collision. There should be minimum collisions as far as possible in the hash function that is used. As collisions are bound to occur, we have to use appropriate collision resolution techniques to take care of the collisions.

Uniform Distribution

- Hash function should result in a uniform distribution of data across the hash table and thereby prevent clustering.

Hash Function v2

Python provides a `hashlib` module implementing different cryptographic hashing algorithms. These hashing functions take variable length of bytes and converts it into a fixed length sequence.

- md5
- sha1
- sha224
- sha256
- sha384
- sha512

Following code converts a string `hello world` to an integer value.

In [19]:

```
import hashlib

int(hashlib.md5('hello world'.encode('utf-8')).hexdigest(), 16)
```

Out[19]:

```
125893641179230474042701625388361764291
```

We can enhance our `_hash()` function in `HashTable` class.

In [20]:

```
import sys

def _hash(key):
    bins = 10
    h = int(hashlib.md5(key.encode('utf-8')).hexdigest(), 16)
    return h % bins
```

Above `_hash()` function gives a better result than using length of the string.

In [21]:

```
print(_hash('Amanda'), _hash('Christ'), _hash('Freddi'), _hash('Steven'))
```

```
8 6 5 4
```

Don't Use `hash()`

Python has a hashing function `hash()` which can be apply to any object, and returns an integer in the range `-2**31` to `2**31 - 1` on 32-bit system, and `-2**63` to `2**63 - 1` on 64-bit system.

But starting from Python version 3.3, "for security reason", `hash()` generates different values in different Python session.

Summary

The performance of a hash table depends on following factors:

1. How good the hash function could distribute the keys evenly over the hash table
2. Size of the hash table

In this example, we store all the data inside the hash table. In practise, we store pointers to the actual records which could be in the memory or permanent storage (such as disc).