Hash Tables

CPE202

Hash Table

To search for an item in an array with a key:

a. Compute a hash using a function that transforms the key into an array index known as hash value.
b. Use the hash value to access an item in the array.
c. Might need to do collision resolution

Transforming a key to an array index (Hash Value)

- Use modulo operator: key % M, where M is the size of the table.
  - Computed hash values will be between 0 and M - 1
  - Using a prime number for M usually results in good distribution of key values in the table.

Hashing String Key

```
def hash(string, m, r=31):
    “”” hash function for string key weighted by a prime number.
    Args:
        string (str) : string key
        m (int): the size of the hash table (array)
        r (int) : a prime number
    Returns:
        Int : hash value
    “””
    h = 0
    for c in string:
        h = (r * h + ord(c)) % m
    return h
```

Hashing String Key

```
>>> hash('eat', 11)
7
>>> hash('ate', 11)
4
```

Hashing Compound Key such as date

```
def hash(date, r=31):
    “”” hash function for compound key weighted by a prime number.
    Args:
       date (tuple) : tuple of 3 int (year, month, day)
       r (int) : a prime number
    Returns:
       Int : hash value
    “””
    year, month, day = date
    h = (day * r + month) * r + year
    return h
```

Hash Function Examples

- Folding method
    - Phone number 805-222-1234
    - [80, 52, 22, 12, 34]
    - (80 + 52 + 22 + 12 + 34) % M
- Mid-square method
    - Square the key value and extract the mid 2 digits and modulo it by the table size.

Requirements for Good Hash Function

- It should be consistent
- It should be efficient to compute
- It should uniformly distribute the set of keys

Operations

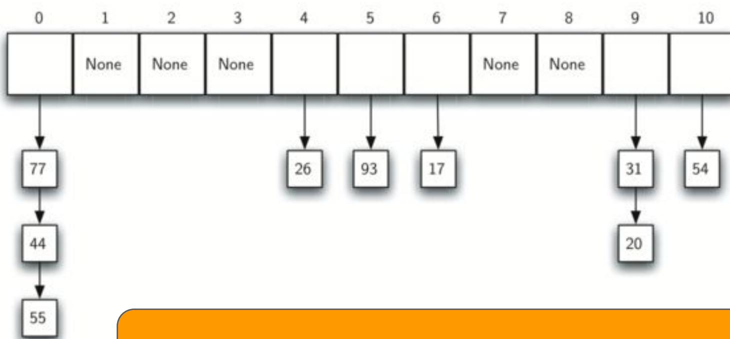
- put
  - Insert a key - value pair to the table
- get
  - Get the value associated with a key
- delete
  - Simply delete a key - value pair from the list in the table slot.

Collision Resolution

When multiple keys hashed to the same hash value, we need to resolve the collisions.

Separate Chaining

- Each slot stores a linked list of keys resulted in the same hash.

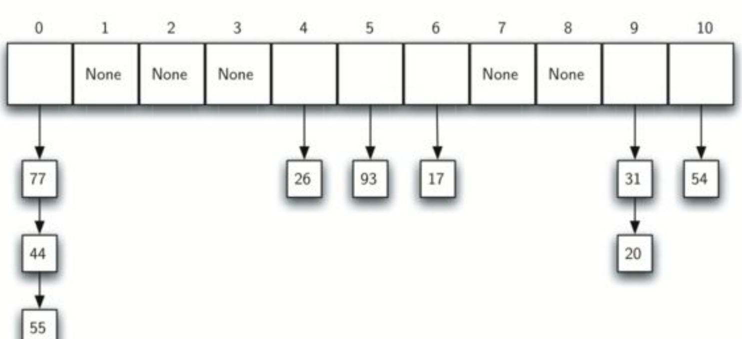


Do we need to store keys as well?

You need to store keys as well!

Separate Chaining

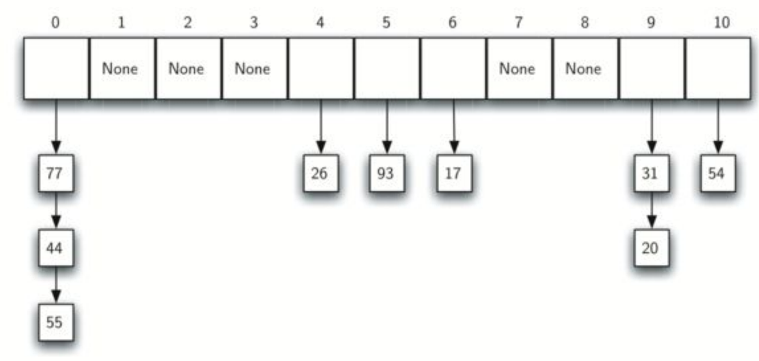
- Each slot stores a linked list of keys resulted in the same hash.



Let's put key=11, val='Eleven'

Separate Chaining

- Each slot stores a linked list of keys resulted in the same hash.



Let's get value for key= 44

Separate Chaining Performance

- The search time for a key on a hash table with the separating chaining depends on the length of the lists.
- On average, the length of the lists is n / m, where n is the number of items and m is the size of the table.
- n / m is also known as a load factor α

Load Factor

Load Factor: number of items / table size

Operations

- resize
  - Optionally you can resize the table to improve the performance when the load factor gets too large.
  - Double the size (+1) and rehash all key-value pairs in the table.

resize

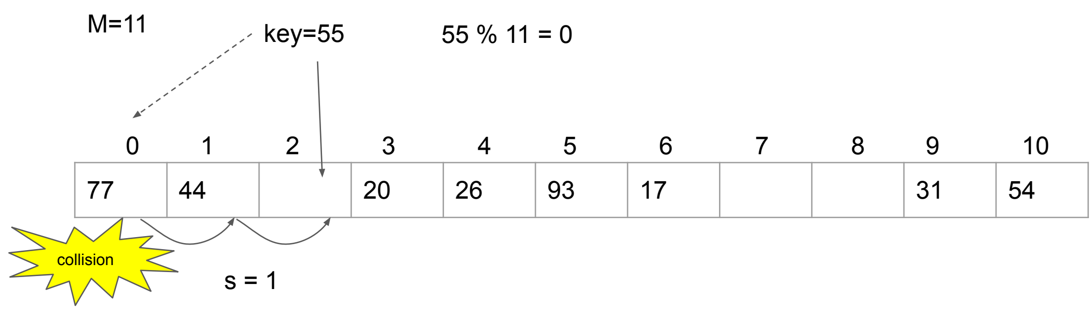
- create a new array whose size is a double of the old size + 1
- for each key, val pair in the old array
  - put(key, val) into the new array
- Replace the old array with the new array

Open Addressing

- If the slot is already occupied go to other slot … until an empty slot is found or come back to the original slot.
- The search is conducted in the same manner.
- May introduce clustering of keys.

Linear Probing

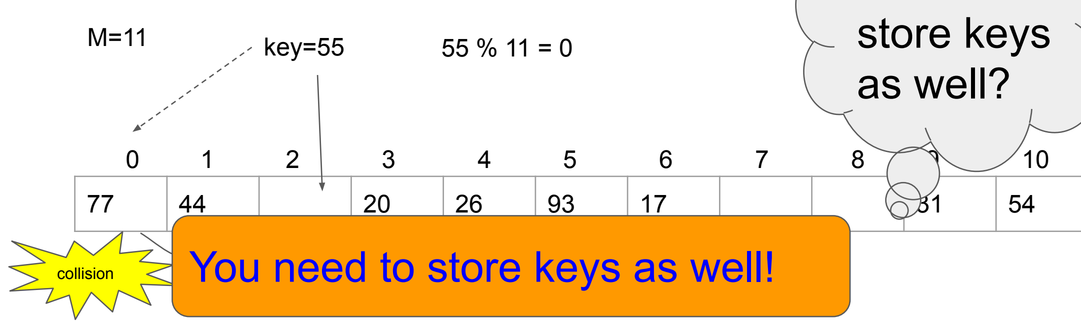
- If the slot is occupied, probe the next slot or +s slot, where s is a number of slots to skip.
- rehash(h) = (h + s) % M



Linear Probing

- If the slot is occupied, probe the next slot or +s slot, where s is a number of slots to skip.
- rehash(h) = (h + s) % M

Do we need to store keys as well?



You need to store keys as well!

Linear Probing Performance

- The search time for a key on a hash table with the linear probing depends on the way in which the entries clump together into contiguous groups of occupied table entries, called ***clusters***.
- Short clusters is an requirement for efficient performance.

Can the load factor exceed 1?

Linear Probing Limitation

- The load factor can not exceed 1
  - The load factor must be smaller than 1: $\boldsymbol{\alpha} < 1$
  - It requires resizing when $\boldsymbol{\alpha}$ reaches 1

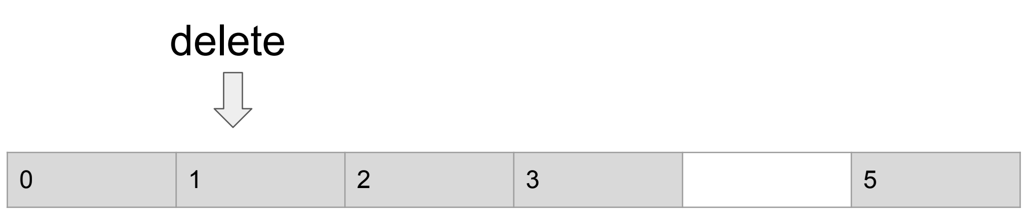
Operations

- put
  - Insert a key - value pair to the table
- get
  - Get the value associated with a key
- delete
  - Can you simply delete an entry?

```
def get(table, key):
    """get the value associated with a given key in hash table
    Args:
        table (list) : hash table
        key (int) : the key
    Returns:
        any : the value
    Raises:
        KeyError : if the key is not found
    """
    m, h = len(table), hash(key)
    i = h % m
    #assuming that the load factor < 1 is always True
    while table[i] is not None and key != table[i].key:
        i = (i + 1) % m
    if table[i] is not None and key == table[i].key:
        return table[i].val
    raise KeyError
```

this is for s = 1

```python
def contains(table, key):
    """check if a given key exists in hash table
    Args:
        table (list) : hash table
        key (int) : the key
    Returns:
        bool : True if the key is found, otherwise False
    """
    try:
        val = get(table, key)
        return True
    except:
        return False
```



1. Until you reach an empty slot, go to the next (+s) slot on the right.
2. Hash the key value pair at the slot again (do put).

```
def delete(table, key):
    """deletes an entry from hash table
    Args:
        table (list) : hash table
        key (int) : the key of an entry to be deleted
    Raises:
        KeyError: if the key is not found
    """
    if key not in table:
        raise KeyError
    m, h = len(table), hash(key)
    i = h % m
    while key != table[i].key:
        i = (i + 1) % m
    table[i] = None
    i = (i + 1) % m
    while table[i]:
        key_to_redo, val_to_redo = table[i].key, table[i].val
        table[i] = None
        put(table, key_to_redo, val_to_redo)
        i = (i + 1) % m
    #optionally shrink table size if the number of items < threshold
```

this is for s = 1

Operations

- resize
  - You need to resize the table before the load factor reaches 1 otherwise you may not be able to put more items in the table.
  - Double the size (+1) and rehash all key-value pairs in the table.

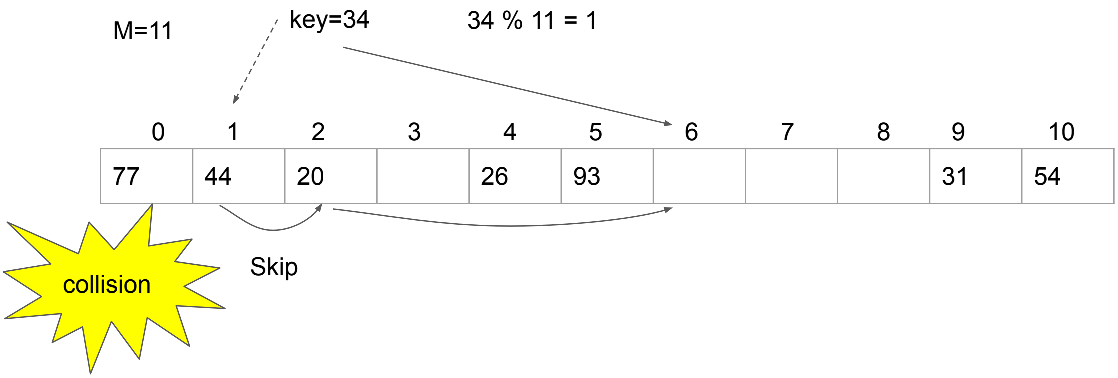
Resize Operation

- Create a new hash table with doubled size + 1
- For each key - value pair in the original table
  - Do put(key, val) to the new table
- Swap the new table with the old table

Quadratic Probing

- The interval between probes is increased by adding the successive outputs of a quadratic polynomial to the starting value given by the original hash computation
  - Example
    - rehash(h) = (h + f(i)) % M, where $f(i) = i^2$ and i = 1, 2, …
    - Example, $h_1 = h_0 + 1$, $h_2 = h_1 + 4$, $h_3 = h_2 + 9$, ...

Quadratic Probing



Analysis of Hash Table

Lookup can be O(1) if there are no collisions

The Theoretical Optimum

Analysis of Hash Table

- In reality, the performance depends on the load factor (***α***)
  - Chaining
    - Average in successful search
      - $1 + \alpha/2$
    - Average in unsuccessful search
      - α

Analysis of Hash Table

- In reality, the performance depends on the load factor (***α***)
  - Linear probing
    - Average # of comparisons in successful search
      - Approximately $\frac{1}{2}*(1 + 1/(1-\alpha))$
    - In unsuccessful search
      - Approximately $\frac{1}{2}*(1 + 1/(1-\alpha)^2)$

Map Abstract Data Type

- Map - Associates a key with a value. A.K.A. Associative Array, Hash Map
  - A good example of map is dictionary in python
- Python dictionary
  - d = {‘course’: ‘CPE202’, ‘quarter’: ‘Fall 2019’}
  - course = d[‘course’] if ‘course’ in d else None

Map Abstract Data Type

- Map - Associates a key with a value. A.K.A. Associative Array, Hash Map
  - A good example of map is dictionary in python
- Python dictionary
  - d = {‘course’: ‘CPE20
  - course = d[‘cours

What other data structure have we learned that can be used for Map ADT?

Map Abstract Data Type

- `Map()` Create a new, empty map. It returns an empty map collection.
- `put(key,val)` Add a new key-value pair to the map. If the key is already in the map then replace the old value with the new value.
- `get(key)` Given a key, return the value stored in the map or `None` otherwise.
- `remove(key)` Delete the key-value pair from the map.
- `contains(key)` Return `True` if the given key is in the map, `False` otherwise.