Overview

• Hash table
• Hash functions
• Collision resolution
• Map data type
• Analysis of hashing
**Objective**

- Understand the principles of hash tables and hash functions
- Learn how to resolve collisions in hash functions
- Be able to implement hash tables and hash functions

**Hashing**

- Data structure that can be searched in $O(1)$ time
- Need to know more about where items are when searched for in collection
- Single comparison if item is where it should be
Hash Table

- Collection of items stored in a way which makes them easy to find later
- Position in hash table often called slot
  - Holds an item
  - Named by integer value
  - Initially, every slot is empty

Hash Table

- Implement hash table using list
- Each element initialized to special Python value None
- Hash table of size $m = 11$
  - $m$ slots
  - Named 0 through 10
Hash Function

• Mapping between item and slot where it belongs in is called **hash function**
• Function take any item in collection and return integer in range of slot names (0, ..., m – 1)

Hash Function: Example

• Set of integer items 54, 26, 93, 17, 77, and 31
• ”remainder method” takes item and dives it by table size => \( h(item) = item \% 11 \)

<table>
<thead>
<tr>
<th>Item</th>
<th>Hash Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>54</td>
<td>10</td>
</tr>
<tr>
<td>26</td>
<td>4</td>
</tr>
<tr>
<td>93</td>
<td>5</td>
</tr>
<tr>
<td>17</td>
<td>6</td>
</tr>
<tr>
<td>77</td>
<td>0</td>
</tr>
<tr>
<td>31</td>
<td>9</td>
</tr>
</tbody>
</table>
**Hash Function: Example**

- After hash values computed, insert each item into hash table
- 6 of 11 slots are now occupied => load factor $\lambda = \frac{\text{number of items}}{\text{table size}}$ (here $\lambda = 6/11$)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>77</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>25</td>
<td>93</td>
<td>17</td>
<td>None</td>
<td>None</td>
<td>31</td>
<td>54</td>
</tr>
</tbody>
</table>

**Hash Function: Example**

- Use hash function to compute slot name and check if item is present
- $O(1)$ since constant amount of time is required
  - to compute hash value
  - index hash table at that location
- $\Rightarrow$ Constant time search algorithm
Hash Function: Issue

- Only works if each item maps to unique location in hash table
- If item 44 is next in collection
  - Hash value $44 \% 11 = 0$
  - Same index as for value 77
  - Collision

Perfect Hash Function

- Function that maps each item into a unique slot
- Perfect hash function can be constructed if items never change
- No systematic way to construct perfect hash function given arbitrary collection
- Good news: hash function does not need to be perfect
**Perfect Hash Function: Approach I**

- Increase size of hash table
  - Each value in the item range can be accommodated
  - Unique slot for each item
- Practical for small number of items, not feasible when number is large
- Items: 9-digit SSN => ~one billion slots

**Perfect Hash Function: Goal**

- Goal:
  - Minimize collisions
  - Easy to compute
  - Evenly distributes items in hash table
Perfect Hash Function: Folding Method

• Divide item into equal size pieces (might not work for last one)
• Add pieces together to calculate hash value
• Example:
  • Phone number: 413-545-0444 (41, 35, 45, 4, 44)
  • $41 + 35 + 45 + 4 + 44 = 169$
  • $169 \mod 11 = 4$
  • 4th slot for 413-545-0444

Perfect Hash Function: Mid-Square Method

• First square item, then extract some portion of resulting digits
• Example:
  • Item 44 $\Rightarrow 44^2 = 1,936$
  • Extracting middle two digits $\Rightarrow 93$
  • $93 \mod 11 = 5$
Perfect Hash Function: Comparison

<table>
<thead>
<tr>
<th>Item</th>
<th>Remainder</th>
<th>Mid-Square</th>
</tr>
</thead>
<tbody>
<tr>
<td>54</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>26</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>93</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>17</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>77</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>31</td>
<td>9</td>
<td>6</td>
</tr>
</tbody>
</table>

Collision Resolution

- How to place two items in hash table if they hash to same slot?
- Since avoiding collisions is impossible, collision resolution is essential
Collision Resolution: Open Addressing

• Try to find another open slot to hold item causing collision

• Start at original hash position and sequentially move through slots (loop around to start to cover entire table)

• Systematically probing each slot one at a time => linear probing
Collision Resolution: Search

- Look up 93
  - Hash value => 5
  - Slot value => 93

- Look up 20
  - Hash value => 9
  - Slot value => 31
  - Sequential search starting at index 10

- Find 20 or empty slot

Collision Resolution: Clustering

- If many collisions occur for same hash value, number of surrounding slots will be filled

- Negative impact when inserting other items

- Example of inserting 20 (hashing to 0)
Collision Resolution: Slot Skipping

• Skip slots
  • More evenly distribute items that have caused collision
  • Reduce clustering
• Example: plus 3 probing

Collision Resolution: Rehashing

• Linear probing: \( \text{rehash}(pos) = (pos + 1) \mod \text{sizeof table} \)
• Rehash “plus 3”: \( \text{rehash}(pos) = (pos + 3) \mod \text{sizeof table} \)
• General: \( \text{rehash}(pos) = (pos + \text{skip}) \mod \text{sizeof table} \)
• Note: \( \text{skip} \) such that all slots in table will be used
• Often prime number is used (11 in case of example)
Collision Resolution: Quadratic Probing

• Rehash function that increments have value by 1, 3, 5, 7, 9
• $H, h + 1, h + 4, h + 9, h + 16$
• Quadratic probing uses skip of successive squares

Chaining

• Many items at same location
• Search: use hash function then search to decide whether item is present
Implementing Hash Table

• Dictionary => data type to store key:value pairs
• Key is used to look up associated data value
• Often referred to as map

Map: Abstract Data Type

• Map() creates a new, empty map; returns an empty map collection.
• put(key, val) adds new key-value pair; if key already in map, replace old with new value
• get(key) returns value stored in map or none otherwise
• del delete key-value pair using statement del map[key]
• len() returns number of key-value pairs stored in map
• in returns True for statement key in map, False otherwise
Map

• Benefit: given key look up associated data quickly
• Implementation that supports efficient search
• Hash table potentially O(1) performance

Hash Table Implementation

• Class HashTable uses two lists
  • slots holds keys
  • data holds value
  • Initial size 11 in example

```python
class HashTable:
    def __init__(self):
        self.size = 11
        self.slots = [None] * self.size
        self.data = [None] * self.size
```
**Hash Table Implementation**

```python
def put(self, key, data):
    hashvalue = self.hashfunction(key, len(self.slots))

    if self.slots[hashvalue] == None:
        self.slots[hashvalue] = key
        self.data[hashvalue] = data
    else:
        if self.slots[hashvalue] == key:
            self.data[hashvalue] = data  # replace
        else:
            nextslot = self.rehash(hashvalue, len(self.slots))
            while self.slots[nextslot] != None and self.slots[nextslot] != key:
                nextslot = self.rehash(nextslot, len(self.slots))

            if self.slots[nextslot] == None:
                self.slots[nextslot] = key
                self.data[nextslot] = data  # replace
            else:
                self.data[nextslot] = data  # replace
```

```python
def hashfunction(self, key, size):
    return key % size

def rehash(self, oldhash, size):
    return (oldhash + 1) % size
```
Hash Table Implementation

```python
def get(self, key):
    startslot = self.hashfunction(key, len(self.slots))
    data = None
    stop = False
    found = False
    position = startslot
    while self.slots[position] != None and \
       not found and not stop:
        if self.slots[position] == key:
            found = True
            data = self.data[position]
        else:
            position=self.rehash(position, len(self.slots))
            if position == startslot:
                stop = True
    return data
```

• Overload `__getitem__` and `__setitem__` to allow using “[]”
• This will make index operator available
Hash Table Analysis

• Best case: $O(1)$

• Analyze load factor $\lambda$
  • Small $\lambda$ -> lower chance of collisions
  • Large $\lambda$ -> table is filling up, more collisions

Hash Table Analysis

• Open addressing with linear probing
  • Successful search $\frac{1}{2} \left(1 + \frac{1}{1-\lambda}\right)$

  • Unsuccessful search $\frac{1}{2} \left(1 + \left(\frac{1}{1-\lambda}\right)^2\right)$

  • Chaining:
    • Successful search $1 + \frac{1}{\lambda}$
    • Unsuccessful search $\lambda$
Next Steps

• Next lecture on Tuesday
• Discussion on Thursday
• Homework due Thursday