Hashing & Hash Tables

CAPP 30122

What is a Hash Table?

- A data structure that maps an identifying value (key) with some associated data (which can be the key itself and/or another value).

- Save items in a key-indexed table by using a special function called a hash function on the key.

- Always try to choose a prime number as table size (it helps with hashing).

- One implementation of a hash table is the python dictionary (dict) type.

```
class HashTable:
    def __init__(self, initSize = 7):
        self._size = initSize
        # _cells is a list that will hold the values
        self._cells = ... # Think about how you would create this
    def _hash(self, key):
         pass  # Next slide
    def insert(self, key, value):
        cell_index = self._hash(key)
        self._cells[cell_index] = (key,value) # Is this correct...?
```

Hash Function

- A function that computes a table index (integer) from a key
  - Each table position equally likely for each key
  - Converts k into an integer and then mod (modulus) it by the table size:

$$h(k) = k\_int \bmod M$$

  - If our keys were strings here's one example of a hash function:

```
def _hash(self, key):
    hashCode = 0
    for c in key:
        hashCode = hashCode + ord(c)
    return hashCode % self._size
```

Example

```
def _hash(self, key):
    hashCode = 0
    for c in key:
        hashCode = hashCode + ord(c)
    return hashCode % self._size
```

```
table = HashTable()          # _size = 7
table.insert("a", 45)        # _hash = 97 % 7  = 6
table.insert("ac", 23)       # _hash = (97+99)%7 = 0
table.insert("cat", 12)      # _hash = (99+97+116)%7 = 4
```

Index	0	1	2	3	4	5	6
Cells	("ac", 23)	None	None	None	("cat",12)	None	("a",45)

Problem: Collisions!

- Collision - when a hash function maps two or more elements to the same index
- Our hash function is not a great hash function because many collisions could happen! (In what instance will this happen?)
- Even with a well-written hash function, collisions will happen. Every hash table implementation needs a collision resolution scheme:
  - An algorithm for handling collisions

Standard String Hash Function

- A more effective approach is to use Horner's method to compute a polynomial whose coefficients are the integer values of the characters in a string.

$$
\begin{aligned}
p(x) &= a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots + a_n x^n \\
&= a_0 + x\Big(a_1 + x\Big(a_2 + x\big(a_3 + \cdots + x(a_{n-1} + x\, a_n)\cdots\big)\Big)\Big)
\end{aligned}
$$

  - "a_i" are the integer values for the characters in a string
  - "x" is a prime multiplier integer value

Standard String Hash Function

- Given the string "abc", the hash value would be computed as:

```
multiplier =   37   # should  be relatively prime
hash_value = 0
hash_value = (hash_value * multiplier + ord("a")) % self._size
hash_value = (hash_value * multiplier + ord("b")) % self._size
hash_value = (hash_value * multiplier + ord("c")) % self._size
```

- You should be able to convert the above code (using a for loop) into a simple function.

Collision Resolution

Separate chaining and **Open addressing** are the most common schemes used when keys resolve to the same hash code:

- Separate chaining:
  - Put keys that collide in a list
  - Need to search lists during lookup
- Open addressing (e.g., linear probing, quadratic probing, double hashing):
  - When a new key collides, find an empty slot in the table
  - Complex collision patterns
- Question for you: When would it be better to use over the other one?

Linear Probing

- Probing: Resolving a collision by moving to another index.
  - **Linear Probing** - move to the next available cell
  - If you cannot find an available cell then you'll need to rehash the table (make the table larger and rehash all items).
  - You linear probe for both lookup and insertion.

```
table = HashTable()
table.insert(“Sally”, 45)  # h(“Sally”) = 5
table.insert(“Bob”, 23)   # h(“Bob”) = 0
table.insert(“Joe”, 12)   # h(“Joe”) = 0, collides with cell[0], next available cell = cell[1]
table.insert(“Pam”, 9)    # h(“Pam”) = 0, collides with cell[0] & cell[1], next available cell = cell[ 2]

table.insert(“Tim”, 12)   #h(“Tim”) = 6
```

Index	0	1	2	3	4	5	6
Cells	(“Bob”, 23)	(“Joe” ,12)	(“Pam”, 9)	None	None	(“Sally” ,45)	(“Tim”, 12)

Wrap Around

- When probing, be sure to warp around to the beginning of the table if you reach the end.

```
table = HashTable()
table.insert("Ally", "a")    # h("Ally") = 5
table.insert("Carol", "b")  # h("Carol") = 6
table.insert("Roy", "c")     # h("Roy") = 5, collides with cell[5] & cell[6], so wrap around to the beginning, where cell[0] is available

table.insert("Carl", "d")     # h("Carl") = 6, collides with cell[6], so wrap around to beginning, collides with cell[0], next available cell = cell[1]
```

Index	0	1	2	3	4	5	6
Value	("Roy", "c")	("Carl", "d"	None	None	None	("Ally", "a"	("Carol", b")

Rehashing

- Rehash - Growing to a larger table when the table is too full.
  - Cannot copy the old table values to a new one. (Why not?)
- Load factor (ratio): (# of items) / (hash table length)
  - Many hash tables rehash when load factor ~.75