

# Hashing & Hash Tables

*MPCS 51042*

# Symbol-Table (ADT)

- Types of data-structures where the elements are associated with keys
  - For example: Dictionaries (Python), in this case the value is stored instead of the key.
- Basic Operations
  - `def insert (item):` # Adds an item, which should be a “key-value” object
  - `def lookup(key):` # Returns the key’s value.
  - `def isEmpty():` # Returns true if the table is empty, false otherwise

# Hash Table

- One implementation of a symbol table 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 ( $M$ ) (it helps with hashing)

# Basic Implementation

```
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, item):
```

```
        key, value = item
```

```
        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
  - Typical operation is to convert k into an integer and 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 the String "s"

$$\begin{aligned} p(x) &= a_0 + a_1x + a_2x^2 + a_3x^3 + \dots + a_nx^n \\ &= a_0 + x \left( a_1 + x \left( a_2 + x \left( a_3 + \dots + x(a_{n-1} + x a_n) \dots \right) \right) \right) \end{aligned}$$

- Where the "a<sub>i</sub>" are the integer values for the characters in our string "s"
- Where "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

Terms:  $N \approx$  Number of possible keys,  $M$  = table size

Two main schemes when keys resolve to the same hash code:

- Separate chaining
  - $M$  much smaller than  $N$
  - $\sim N/M$  keys per table position
  - put keys that collide in a list
  - Need to search lists during lookup
- Open addressing (linear probing, quadratic probing, double hashing)
  - $M$  much larger than  $N$
  - plenty of empty table slots
  - when a new key collides, find an empty slot
  - complex collision patterns

# Linear Probing

- Probing: Resolving a collision by moving to another index.
  - Linear Probing - Move 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).

```
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):  $(\# \text{ of items}) / (\text{hash table length})$ 
  - Many hash tables rehash when load factor  $\sim .75$
  - Can use big prime numbers as hash table sizes to reduce collisions