COSC 2306
Data Programming

Search

HashTable coding

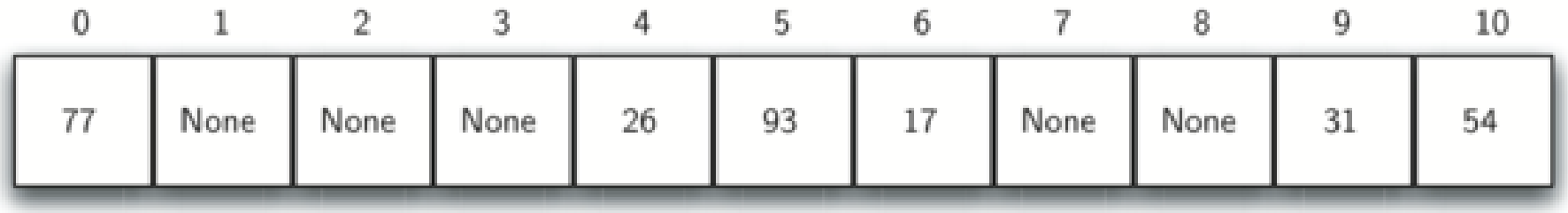
- Coding logics
  - Search item in HashTable (get)
    - Use hash function to compute slot
    - Loop over HashTable
    - if slot == key, return

```
def get(self, key):
    position = self.hash_function(key, len(self.slots))
    if self.slots[position] == key:
        return self.data[position]
    else:
        raise KeyError("Search key failed!")
```

Search

Search:
1) use the hash function to compute the slot name
2) check the hash table to see if it is present

Complexity: O(1)

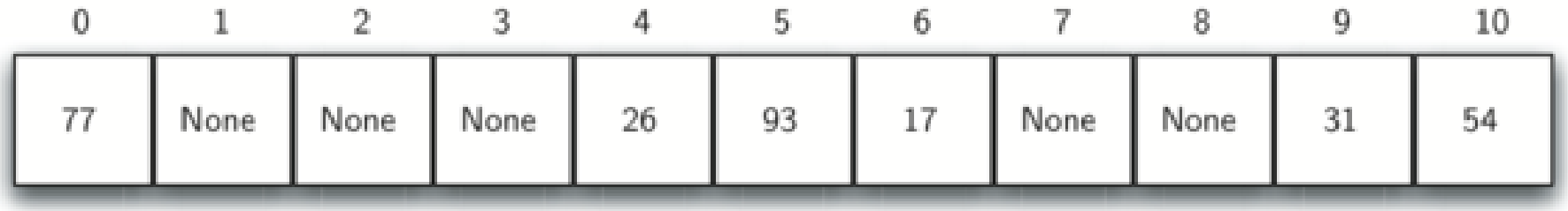


Problems?

Search

Example: add another item 44 into this hash table
44%11 = 0

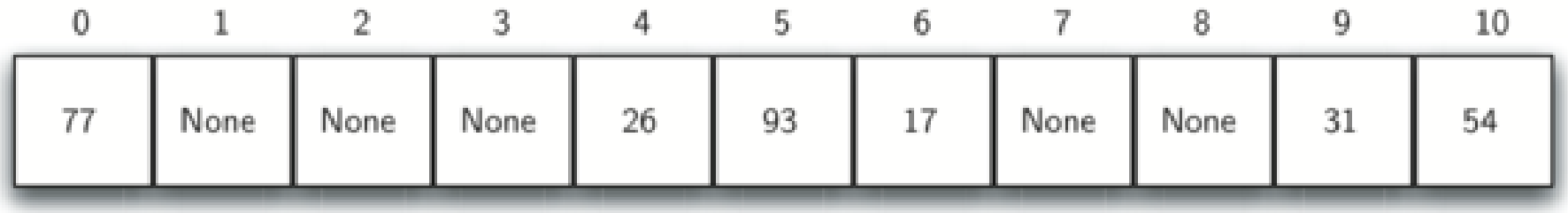
Now we have both 77 and 44 in the first slot



Hash function

Perfect hash function: maps each item into a unique slot

No systematic way to construct a perfect hash function



What to do?

Collisions

- If two items lead to the same slot, we have a **collision**
- Hash functions should be **easy to compute** and **minimize** the number of **collisions**

Folding and mid-square

Extension of reminder method

- **Folding** method: dividing the item into equal size pieces (the last piece may not be of equal size)

  Example: phone number 436-555-4601

  - divide them into groups of 2 (43, 65, 55, 46, 01)
  - 43 + 65 + 55 + 46 + 01, we get 210.
  - assume our hash table has 11 slots, then 210%11 is 1,
  - the phone number 436-555-4601 hashes to slot 1
- **Mid-Square** method: first square the item, and then extract some portion of the resulting digits

  Example: if the item were 44

  - first compute 44^2 = 1,936
  - extract the middle two digits, 93
  - performing the remainder step to get 5 (93%11)

Rehashing

- **Rehashing**: use different hash functions in case of collision
- Method 1: **Open Addressing**: one way to resolve the collision is to find another open slot to hold the item that caused the collision

Question: which open slot to use?

Find the next available open slot, called **Linear Probing**

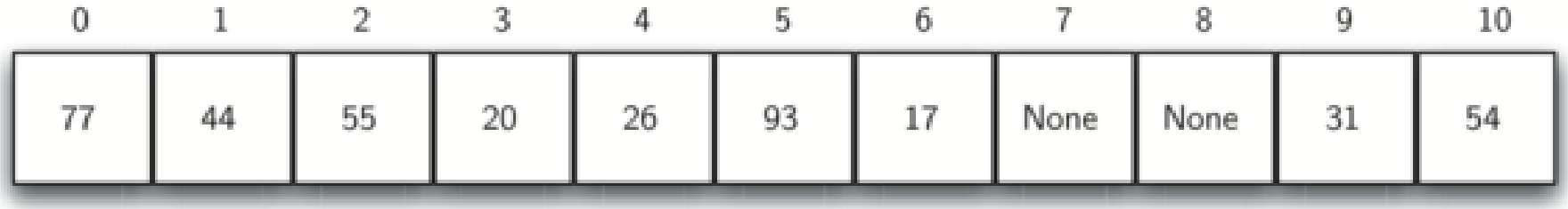
Question: during search, which slot is the right one?

Rehashing

- Use the same method to search for item
- Starting at the position of hash value
- Conduct a sequential search
- Ends until the item is found or the next empty slot

Linear probing

If location t is occupied, move to the next



Linear probing

If location t is occupied, move to the next

ID	*h*(ID)	(*h*(ID) + 1) % 13	(*h*(ID) + 2) % 13
197354864	5		
933185952	10		
132489973	5	6	
134152056	12		
216500306	9		
106500306	3		
216510306	12	0	
197354865	6	7	

Linear probing - coding

- Coding logics
  - HashTable: table size, slot, data
  - Hash function: reminder (key % table_length)
  - Rehash: using linear probing (plus 1)
  - Store item in HashTable (put)
    - Compute hash value using hash function
    - if a slot is None, put key and data directly
    - else a slot is occupied, if slot == key: replace, elif slot != key: rehash
    - Resolve collision: rehash using linear probing (plus 1)
  - Search item in HashTable (get)
    - Use hash function to compute slot
    - Loop over HashTable
    - if slot == key, return
    - else rehash using linear probing (plus 1)

Linear probing - coding

- Coding logics
  - Rehash: using linear probing (plus 1)

```
class HashTable:

    def __init__(self, hash_size):
        # Define a hash table with size of hash_size
        self.size = hash_size
        self.slots = [None] * self.size
        self.data = [None] * self.size

    def hash_function(self, key, size):
        # Define a hash function using remainder
        return key % size
```

Linear probing - coding

- Coding logics
  - Rehash: using linear probing (plus 1)

```
class HashTable:

    def __init__(self, hash_size):
        # Define a hash table with size of hash_size
        self.size = hash_size
        self.slots = [None] * self.size
        self.data = [None] * self.size

    def hash_function(self, key, size):
        # Define a hash function using remainder
        return key % size


    def rehash(self, old_hash, size):
            return (old_hash + 1) % size
```

Linear probing - coding

- Coding logics
  - Store item in HashTable (put)
    - Compute hash value using hash function
    - if a slot is None, put key and data directly
    - else a slot is occupied, if slot == key: replace, elif slot != key: rehash
    - Resolve collision: rehash using linear probing (plus 1)

```
def put(self, key, data):
    hash_value = self.hash_function(key, len(self.slots))
    if self.slots[hash_value] is None:
        self.slots[hash_value] = key
        self.data[hash_value] = data
    elif self.slots[hash_value] == key:
        self.data[hash_value] = data  # replace data
    else:
        raise KeyError("Insert value failed!")
```

Linear probing - coding

```
def put(self, key, data):
        hash_value = self.hash_function(key, len(self.slots))
        if self.slots[hash_value] is None:
            self.slots[hash_value] = key
            self.data[hash_value] = data
        else:
            if self.slots[hash_value] == key:
                self.data[hash_value] = data  # replace data
            else:
                next_slot = self.rehash(hash_value, len(self.slots))
                while (
                    self.slots[next_slot] is not None
                    and self.slots[next_slot] != key
                ):
                    next_slot = self.rehash(next_slot, len(self.slots))
                if self.slots[next_slot] is None:
                    self.slots[next_slot] = key
                    self.data[next_slot] = data
                else:
                    self.data[next_slot] = data # replace data
```

Linear probing - coding

- Coding logics
  - Search item in HashTable (get)
    - Use hash function to compute slot
    - Loop over HashTable
    - if slot == key, return
    - else rehash using linear probing (plus 1)

```
def get(self, key):
    position = self.hash_function(key, len(self.slots))
    if self.slots[position] == key:
        return self.data[position]
    else:
        raise KeyError("Search key failed!")
```

Linear probing - coding

- Coding logics
  - Search item in HashTable (get)
    - Use hash function to compute slot
    - Loop over HashTable
    - if slot == key, return
    - else rehash using linear probing (plus 1)

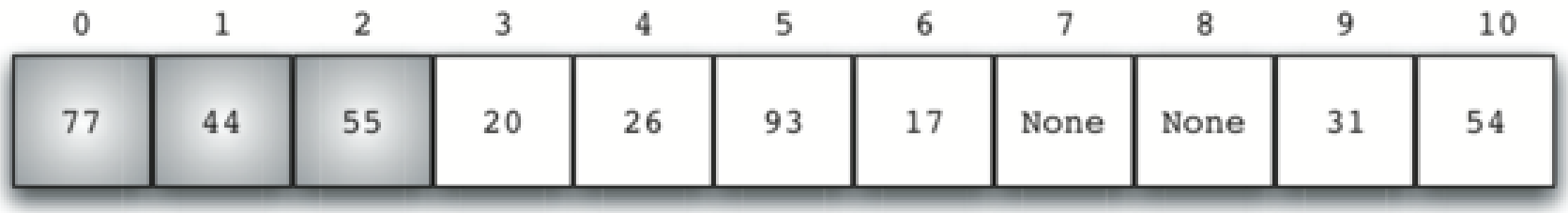
```
def get(self, key):
        start_slot = self.hash_function(key, len(self.slots))
        position = start_slot
        while self.slots[position] is not None:
            if self.slots[position] == key:
                return self.data[position]
            else:
                position = self.rehash(position, len(self.slots))
                if position == start_slot:
                    return None
```

Linear probing

Problems?

Clustering

- many collisions occur at the same hash value, a number of surrounding slots will be filled
- have an impact on other items that are being inserted
- more values together/less empty: slow search



Solutions?

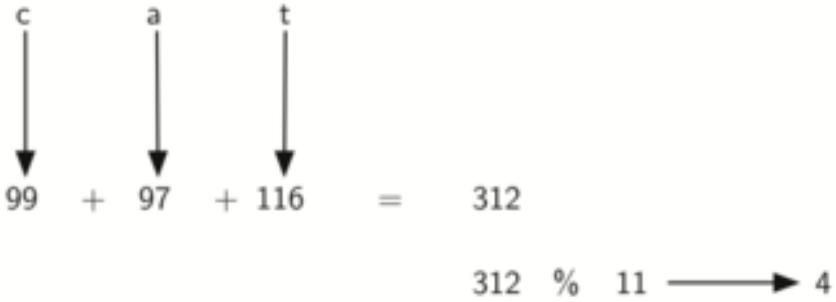
Other solutions

- Quadratic probing
  - First position: t
  - Second position: (t+1) % HTSize
  - Third position: $(t+2^2)$ % HTSize
  - Fourth position: $(t+3^2)$ % HTSize
  - …
- Random probing
  - Use random number to find next available slot
  - $(h(X) + r_i)$ % HTSize

In-class practice: string hashing

- Hashing a string using ordinal values
  - The ord() function returns an integer representing the Unicode character
  - E.g., print(ord('c')) print(ord('5'))

    >>>99 >>>53
  - Any word can be thought of a sequence of ordinal values
  - take these ordinal values, add them up, and use the remainder method to get a hash value



In-class practice: string hashing

- Hashing a string using ordinal values
  - The ord() function returns an integer representing the Unicode character
  - E.g., print(ord('c')) print(ord('5'))
    >>>99 >>>53
  - Any word can be thought of a sequence of ordinal values
  - take these ordinal values, add them up, and use the remainder method to get a hash value

```
def stringhash(a_string, table_size):
    sum = 0
    for pos in range(len(a_string)):
        sum = sum + ord(a_string[pos])
    return sum % table_size
```

In-class practice: string hashing

- Hashing a string using ordinal values
  - The ord() function returns an integer representing the Unicode character
  - E.g., print(ord('c')) print(ord('5'))
    >>>99 >>>53
  - Any word can be thought of a sequence of ordinal values
  - take these ordinal values, add them up, and use the remainder method to get a hash value

```
def stringhash(self, a_string, table_size):
    sum = 0
    for position in range(len(a_string)):
        sum = sum + ord(a_string[position])
    return sum % table_size
```

In-class practice: string hashing

For a given list: animals = ['cat', 'dog', 'cow', 'goat', 'lion', 'duck'], write code to store both the elements in animals (in data) and the corresponding ord values (in slots) into a hash table and print the slots and data lists respectively.

Example output: [None, None, None, None, 312, 434, 314, 423, None, 427, 329]
[None, None, None, None, 'cat', 'lion', 'dog', 'duck', None, 'goat', 'cow']

The HashTable class is given below:

```
class HashTable:

    def __init__(self, hash_size):
        self.size = hash_size
        self.slots = [None] * self.size
        self.data = [None] * self.size

    def stringhash(self, a_string, table_size):
        sum = 0
        for position in range(len(a_string)):
            sum = sum + ord(a_string[position])
        return sum, sum % table_size
```

In-class practice: string hashing

For a given list: animals = ['cat', 'dog', 'cow', 'goat', 'lion', 'duck'], write code to store both the elements in animals (in data) and the corresponding ord values (in slots) into a hash table and print the slots and data lists respectively.

Example output: [None, None, None, None, 312, 434, 314, 423, None, 427, 329]
[None, None, None, None, 'cat', 'lion', 'dog', 'duck', None, 'goat', 'cow']

```
animals = ['cat', 'dog', 'cow', 'goat', 'lion', 'duck']
h = HashTable(11)
for animal in animals:
    slot, pos = h.stringhash(animal, 11)
    h.slots[pos] = slot
    h.data[pos] = animal
print(h.slots)
print(h.data)
```

In-class practice: string hashing

How to query whether an animal is in the hashtable? E.g., check whether 'pig' is in the hashtable and the result would be False.

In-class practice: string hashing

How to query whether an animal is in the hashtable? E.g., check whether 'pig' is in the hashtable and the result would be False.

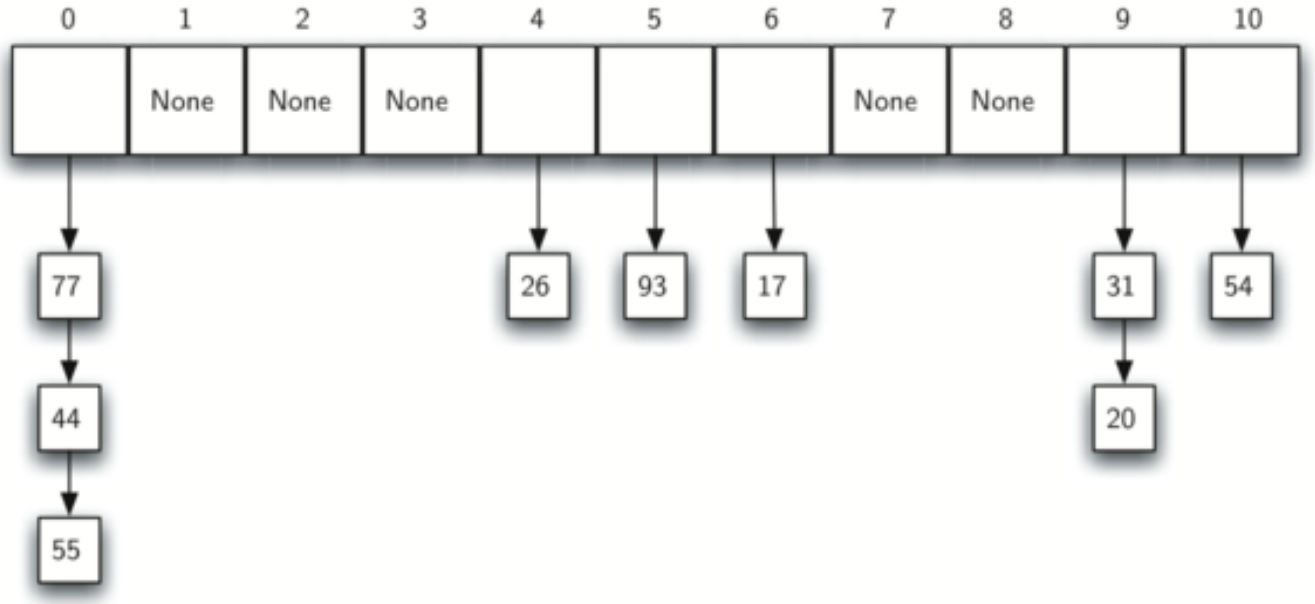
```
x = 'pig'
slot, pos = h.stringhash(x, 11)
if h.data[pos] == x:
    print(True)
else:
    print(False)

#False
```

Chaining

Method 2: **Chaining**: allow each slot to hold a reference to a collection (or chain) of items

Issue: increase collision results in increased number of items on each chain



Chaining - coding

Implement a map/dictionary style HashTable to Store key-value (or key-data) pairs using chaining

```
class HashTable:
    def __init__(self, hash_size):
        self.size = hash_size
        # Each slot starts as an empty list to store chains
        self.table = [[] for _ in range(self.size)]

    def hash_function(self, key):
        return key % self.size

    def put(self, key, data):
        # Check if key already exists, if found update value
        # Otherwise append a new key-value pair

    def get(self, key):


h = HashTable(5)
h.put(10, "apple")
h.put(15, "banana")
h.put(20, "orange")

print(h.get(15))  # banana
print(h.get(100)) # None
```

Chaining - coding

```
class HashTable:
    def __init__(self, hash_size):
        self.size = hash_size
        # Each slot starts as an empty list to store chains
        self.table = [[] for _ in range(self.size)]

    def hash_function(self, key):
        return key % self.size

    def put(self, key, data):
        hash_value = self.hash_function(key)
        # Check if key already exists, if found update value
        for i in range(len(self.table[hash_value])):
            k, v = self.table[hash_value][i]
            if k == key:
                self.table[hash_value][i] = (key, data)
                return
        # Otherwise append a new key-value pair
        self.table[hash_value].append((key, data))

    def get(self, key):
        hash_value = self.hash_function(key)
        for k, v in self.table[hash_value]:
            if k == key:
                return v
        return None
```

```
h = HashTable(5)
h.put(10, "apple")
h.put(15, "banana")
h.put(20, "orange")

print(h.get(15))  # banana
print(h.get(100)) # None
```