UCONN
UNIVERSITY OF CONNECTICUT

Department of Computer Science and Engineering

Data Structures and Object-Oriented Design

(CSE – 2050)

Hasan Baig

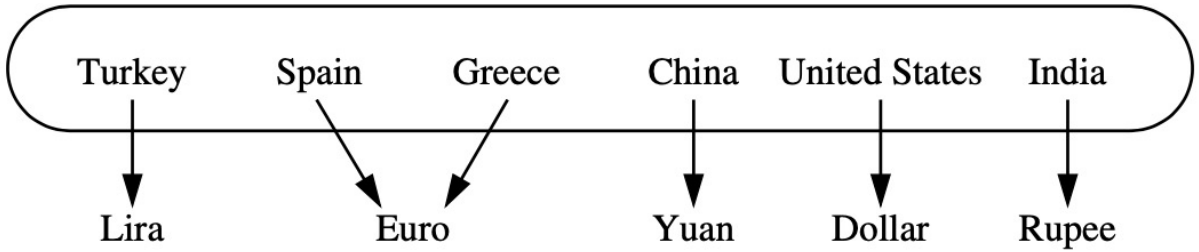
Office: UConn (Stamford), 305C
email: hasan.baig@uconn.edu

Module 8

Mapping and Hashing

Mapping

- A mapping can be defined as an association between two objects.
  - Example: map between Stamford and New York
- These associated objects are also referred as “key-value” pair
- The array of such key-value pairs are sometimes referred as “associative arrays” or ”maps”
- Python has a built-in data type available to hold key-value pairs → Dictionary
- Key must always be unique



Mapping

- Maps uses an array-like syntax for indexing
  - Currency[Greece] to access a value (currency) associated with a given key (Greece)
  - Currency [Greece] = new → to update the value associated with a key “Greece”

- Unlike a standard array, indices for a map need not be consecutive nor even numeric.

Applications

- Student ID (key) → student’s record (name, address, gpa, etc)

- DNS maps a hostname (www.Wikipedia.com) to an IP address, such as 208.215.179.142

Map ADT

- Unlike Python, many other programming languages do not have a built-in mapping data type like dictionary
- We will learn to implement “map” ADT
- At minimum, we should have:
  - get(k) → returns the value associated to the key, *k*
    raise Error if key is not present
  - put(k,v) → to put the key-value pair (k, v) to the Map

Map ADT

- Minimal implementation using list

- We can create a simple class to hold key-value variables as objects

mapping.py

```
1  class Entry:
2      def __init__(self, key, value):
3          self.key = key
4          self.value = value
5
6      def __str__(self):
7          return str(self.key) + " : " + str(self.value)
```

```
1  from mapping import Entry
2
3  class ListMappingSimple:
4      def __init__(self):
5          self._entries = []
6
7      def put(self, key, value):
8          for e in self._entries:
9              if e.key == key:
10                 e.value = value
11                 return
12         self._entries.append(Entry(key, value))
13
14     def get(self, key):
15         for e in self._entries:
16             if e.key == key:
17                 return e.value
18         raise KeyError
```

Map ADT

- To implement a complete ADT with collection, we might want to have the following methods:

`get(k)`	→ return the value associate to the key k, raise error if key is absent
`put(k,v)`	→ add the key-value pair (k, v) to the mapping
`remove(k)`	→ remove the entry with key k if it exists
`__contains__(k)`	→ returns True if the mapping contains a pair with key k
`__len__()`	→ returns the number of keys in the dictionary

First four functions require list traversing, therefore we can write a separate function to traverse through the map

_entry(k)	→ traverse the map to return the element with key k

Activity

Based on the functionality given, determine the correct names of methods of Map ADT

```
from mapping import Entry

class ListMapping:
    def __init__(self):
        self._entries = []
```

```
    def _entry(self, key):
        for e in self._entries:
            if e.key == key:
                return e
        return None
```

4

```
def
    if self._entry(key) is None:
        return False
    else:
        return True
```

3

```
def
    e = self._entry(key)
    if e is not None:
        return e.value
    else:
        raise KeyError
```

1

```
def
    e = self._entry(key)
    if e is not None:
        e.value = value
    else:
        self._entries.append(Entry(key, value))
```

2

```
def
    return len(self._entries)
```

Activity Solution

Based on the functionality given, determine the correct names of methods of Map ADT

```
from mapping import Entry

class ListMapping:
    def __init__(self):
        self._entries = []
```

```
    def _entry(self, key):
        for e in self._entries:
            if e.key == key:
                return e
        return None
```

4

```
def
    if self._entry(key) is None:
        return False
    else:
        return True
```

3

```
def
    e = self._entry(key)
    if e is not None:
        return e.value
    else:
        raise KeyError
```

1

```
def
    e = self._entry(key)
    if e is not None:
        e.value = value
    else:
        self._entries.append(Entry(key, value))
```

2

```
def
    return len(self._entries)
```

Map ADT

- Elements are stored as objects in the list
  → Searching a particular key requires O(n) time complexity

- How dictionary has O(1) time complexity for all operations?

- Python Dictionary is implemented with another data structure

Hash Tables

- The most practical data structure to implement “map”
- Used in Python’s `dict` class implementation.
- Consider the following map with “indices” as keys

M		Sue			Tim	Ali	Mia	Sam			
	0	1	2	3	4	5	6	7	8	9	10

- Values can be accessed in O(1) time
- Example:
  - Access value: M[5] → Ali
  - Update a value M[7] = Sameer

Hash Tables

- What if the value to be saved has a key with big number, e.g. 1002

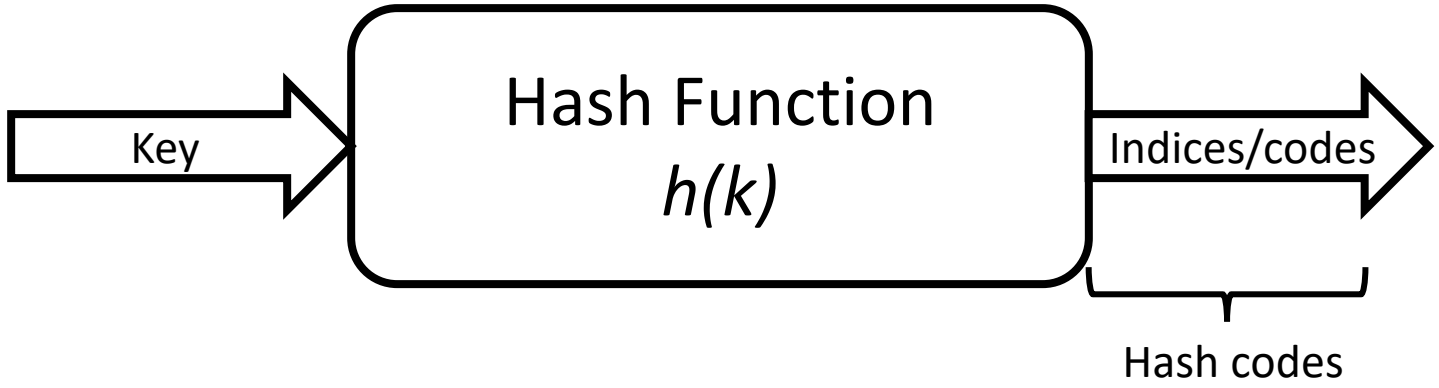
- Require a big chunk of memory reserved for M while wasting the intermediate empty memory locations

Challenges

1. Do not want to allocate a large space N to hold only small amount of elements n (N >> n) → inefficient implementation
2. Do not usually/always have keys as integer values

Hash Tables

- We can pass any "type" of key into a mechanism to convert it into indices/codes



- The goal of a hash function, *h*, is to map each key, *k*, to an integer in the range [0, *N - 1*]
  - N → capacity of a hash table

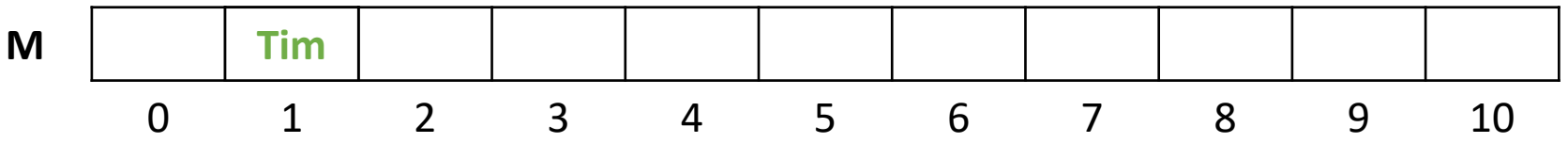
Hash Tables | Hash Function

- There are many different strategies to generate hash code
  - One way: take ASCII value of all characters in a string and calculate the sum of them
    - Example: For “Tim”, ASCII value of:
      - T = 84
      - i = 105
      - m = 109

      ➔ Sum = 298

      How can we store “Tim” with key 298 in the map, M, containing N=11 memory locations?

      Take modulo with size of M → 298 % 11 ➔ 1



Hash Tables | Hash Function

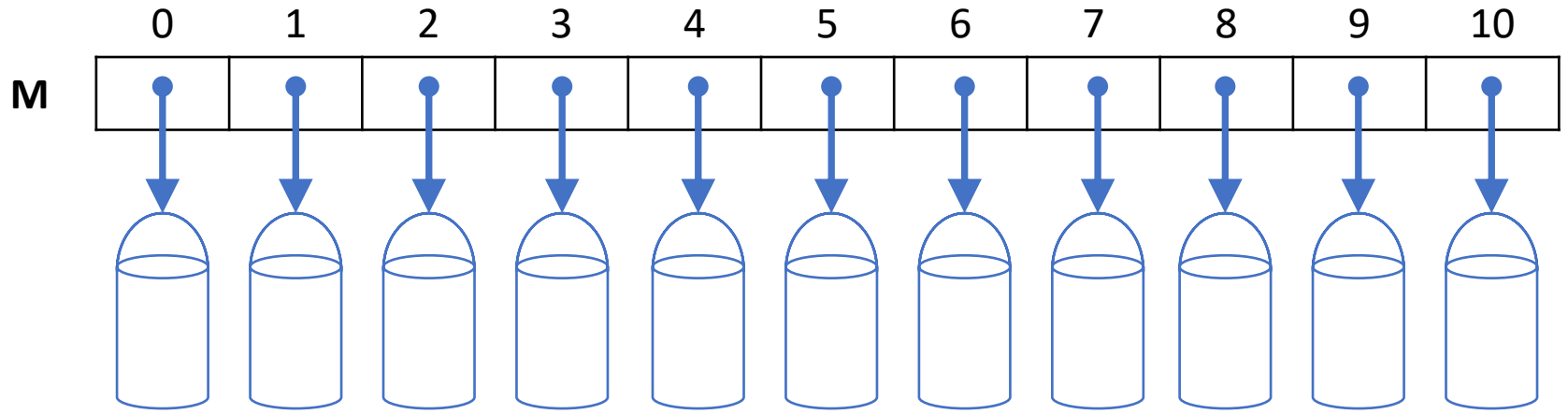
- Similarly,
  - Ali → 65 + 108 + 105 = 278 → 278 % 11 ➔ 3
  - Sue → 83 + 117 + 101 = 301 → 301 % 11 ➔ 4
  - Mia → 77 + 105 + 97 = 279 → 279 % 11 ➔ 4

  - The same key is generated again which has been used before
    ➔ **Hash Collision**

Hash Tables

Collision-Handling Schemes

- Conceptualize hash-table as Bucket Array



- We can store multiple items in any specific bucket

UCONN
UNIVERSITY OF CONNECTICUT

Department of Computer Science and Engineering

Data Structures and Object-Oriented Design

(CSE – 2050)

Hasan Baig

Office: UConn (Stamford), 305C
email: hasan.baig@uconn.edu

Quick Recap

- Completed sorting algorithms
  - Discussed time complexity of quick sort: O(nlogn) and $O(n^2)$
- Mapping → association between two objects → key-value pairs
  - Python has dictionary data structure to hold key-value pairs
- Developed a simple Map ADT with lists having get(k), put(k, v) methods

Quick Recap

- Entry Class to hold key-value pairs

mapping.py

```
1  class Entry:
2      def __init__(self, key, value):
3          self.key = key
4          self.value = value
5
6      def __str__(self):
7          return str(self.key) + " : " + str(self.value)
```

- ListMapping ADT

```
from mapping import Entry

class ListMapping:
    def __init__(self):
        self._entries = []
```

```
def get(self, key):
    e = self._entry(key)
    if e is not None:
        return e.value
    else:
        raise KeyError
```

```
def put(self, key, value):
    e = self._entry(key)
    if e is not None:
        e.value = value
    else:
        self._entries.append(Entry(key, value))
```

```
def __contains__(self, key):
    if self._entry(key) is None:
        return False
    else:
        return True
```

```
def __len__(self):
    return len(self._entries)
```

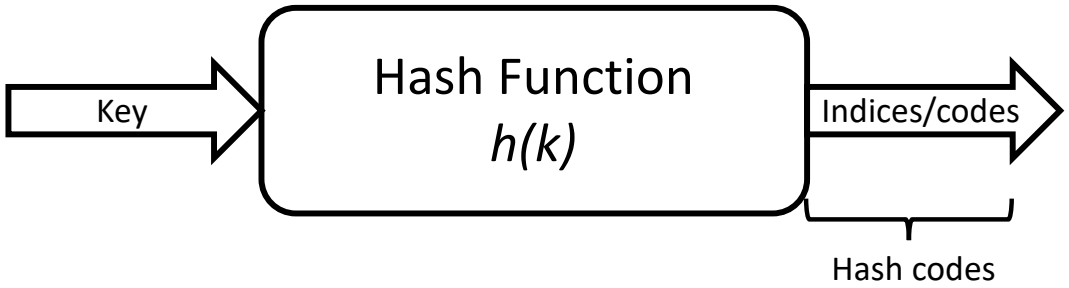
```
def _entry(self, key):
    for e in self._entries:
        if e.key == key:
            return e
    return None
```

Quick Recap

- Next we considered list "indices" as the keys
  - Problem: storing elements with having big numbers as keys

M		Sue			Tim	Ali	Mia	Sam			…	Amy
	0	1	2	3	4	5	6	7	8	9	…	1002

  - Requires large amount of memory to store small number of elements
  - Keys are not "integers" always

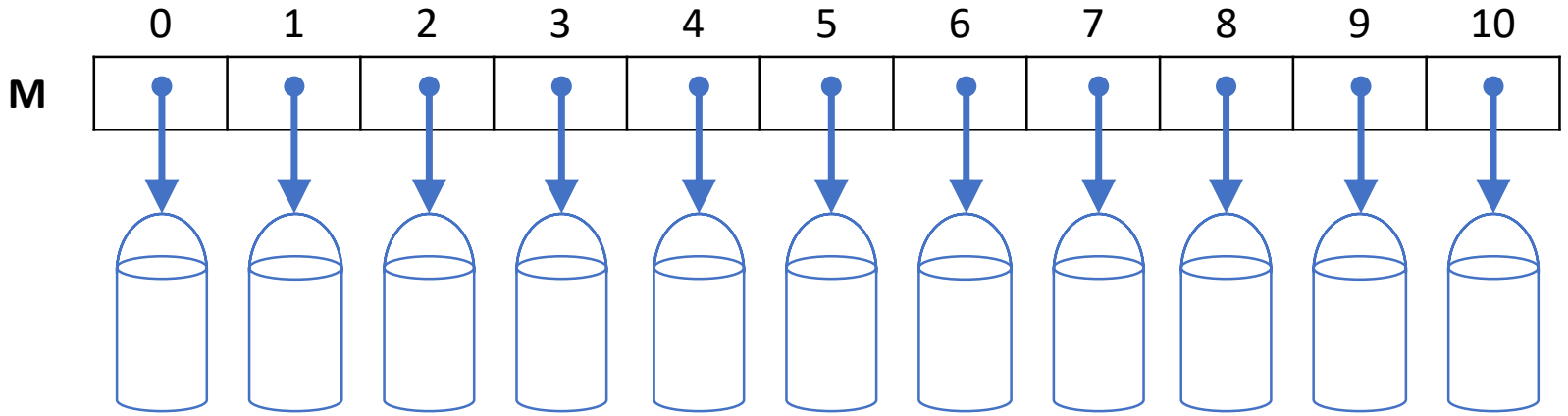


  - Different approaches of generating hash codes
    - Summing up the ASCII codes of each character in a key
    - Take a mod to map the key in the hash table of size N

Quick Recap

- Next problem we came across?
  - Hash codes generated for certain keys are same
  - → map different keys at same location ➔ Hash Collision

- Instead of having a single object at each location in hash table, we conceptualized to have buckets



Hash Tables | Collision-Handling Schemes | Separate Chaining

- Have each bucket M[j] store its own container to carry multiple (k, v) items
  - M is the array of buckets
  - j is the location of j^{th} bucket

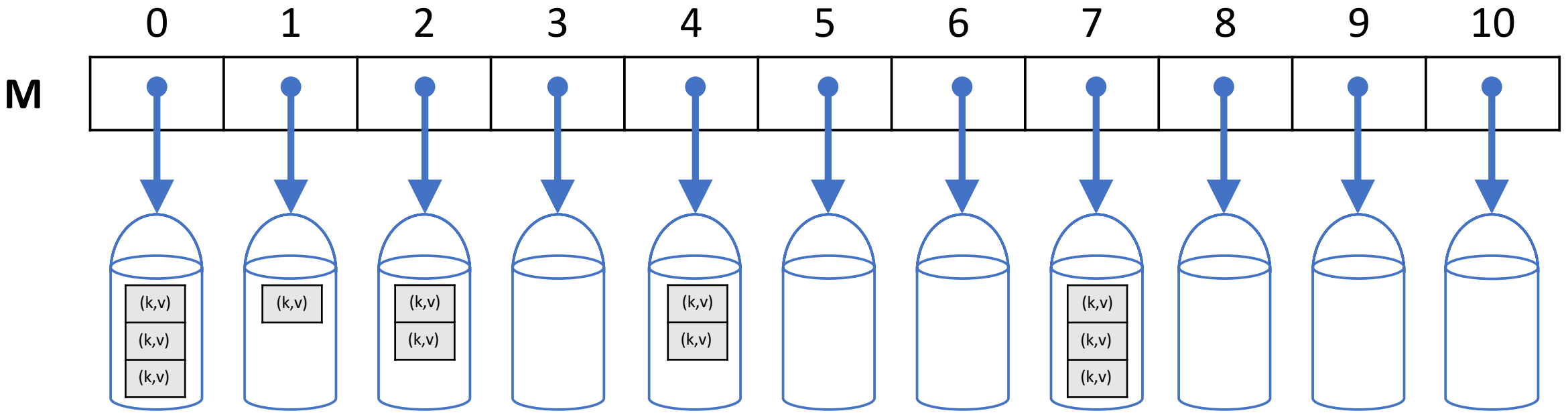
- h(k) = j → hash code

Hash Tables

Collision-Handling Schemes

Separate Chaining

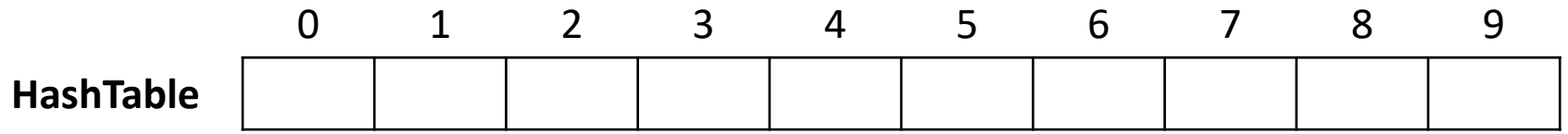
- Have each bucket M[j] store its own container to carry multiple (k, v) items
  - Natural choice of a second container at each location would be a list/linked-list
  - We can use the same `ListMapping` container which we have already created



Hash Tables | Collision-Handling Schemes | Separate Chaining

Implementation

1. Create a HashTable of size 10




```
class HashTable:
    def __init__(self):
        self._htsize = 10
```

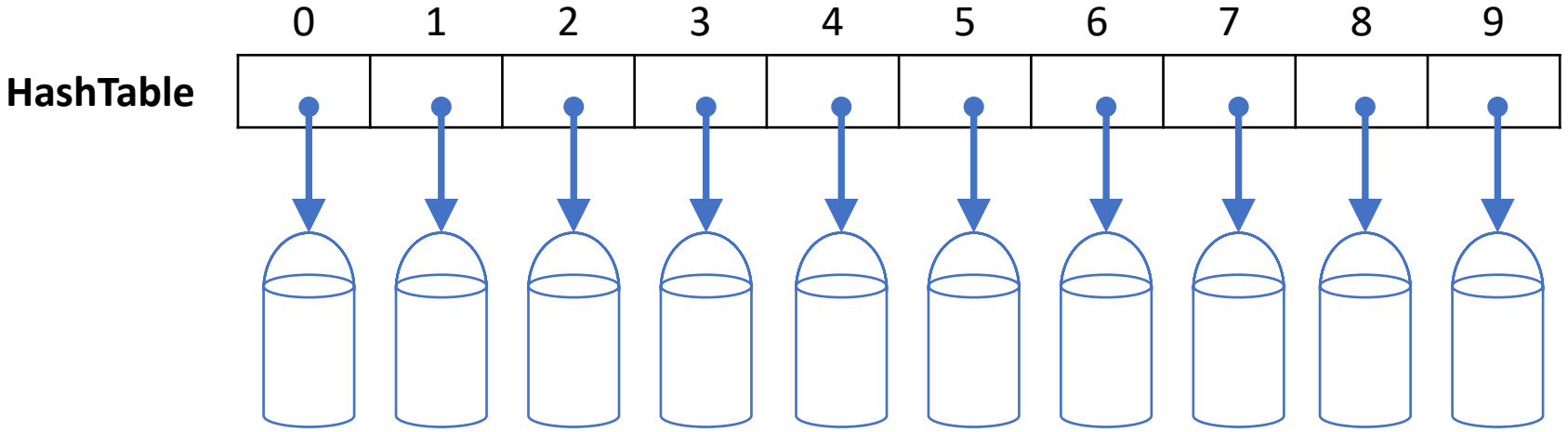
Hash Tables

Collision-Handling Schemes

Separate Chaining

Implementation

2. Create 10 buckets (bucket array) each containing secondary structure (ListMapping)




```
from ListMapping import ListMapping

class HashTable:
    def __init__(self):
        self._htsize = 10
        self._buckets_array = [ListMapping() for i in range(self._htsize)]
```

Hash Tables | Collision-Handling Schemes | Separate Chaining

Implementation

3. Putting a key-value pair in the bucket
   - Get the bucket first
     - Generate a hash function for the key → using Python's built-in `hash` function
     - Take a modulo with the size of a hash table to calculate j^{th} index (bucket location)
     - Return j^{th} bucket
   - Put the key-value pair in the j^{th} bucket

4. Retrieving a value
   - Get the bucket (using the same above procedure) and return the value associated to the key

Activity

Implement the following methods in `HashTable` class

`put(self, key, value):` to put a key-value pair in the bucket

- Get the bucket first
  - Generate a hash function for the key → using Python's built-in `hash` function
  - Take a modulo with the size of a hash table to calculate j^{th} index (bucket location)
  - Return j^{th} bucket
- Put the key-value pair in the j^{th} bucket

`get(key):` to retrieve the value associated with the key

- Get the bucket (using the same above procedure) and return the value associated to the key

Activity Solution

Implement the following methods in HashTable class

`put(self, key, value):` to put a key-value pair in the bucket

```
def put(self, key, value):
    bucket = self._get_bucket(key)
    bucket.put(key,value)
    # bucket[key] = value
```

```
def _get_bucket(self, key):
    j = hash(key) % self._htsize
    return self._buckets_array[j]
```

`get(key):` to retrieve the value associated with the key

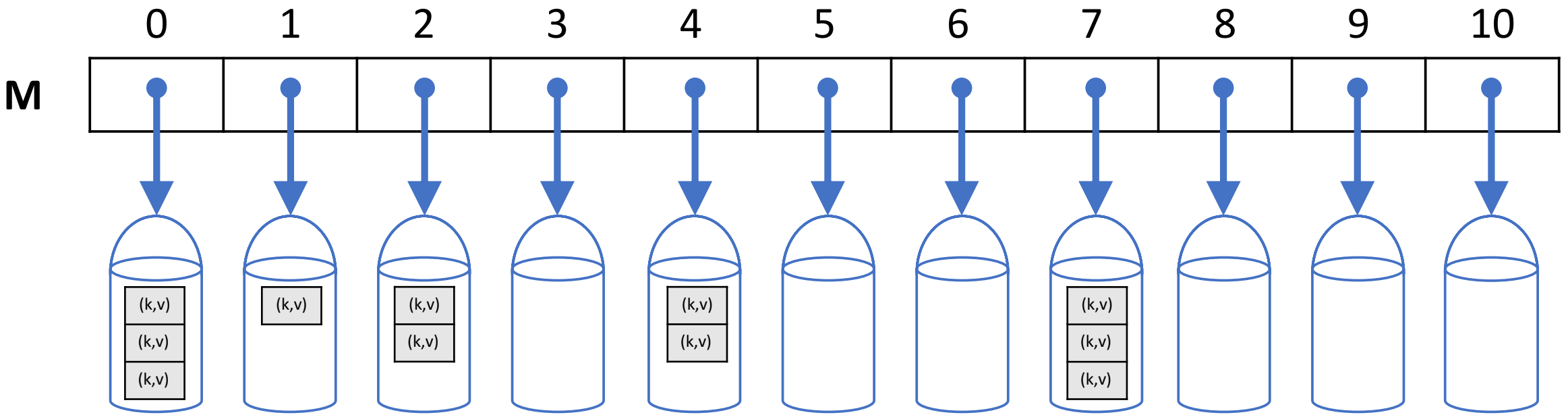
```
def get(self, key):
    bucket = self._get_bucket(key)
    return bucket.get(key)
```

Hash Tables

Collision-Handling Schemes

Separate Chaining

- In Worst case:
  - Time to search for the bucket is O(1)
  - Time to search a key in the bucket depends on the size of bucket: For size m → O(m)



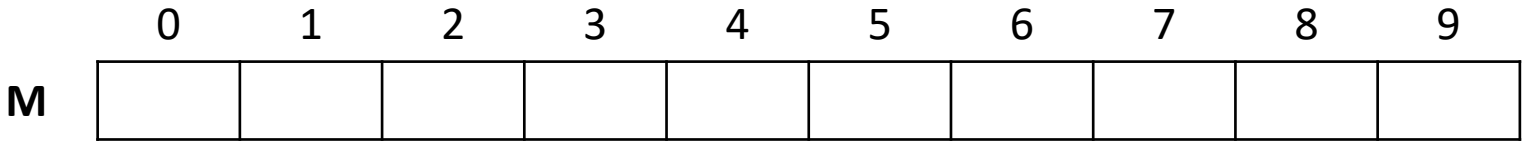
Hash Tables | Collision-Handling Schemes

- A good hash function index “n” items of a map in a bucket array of capacity “N”
  → The expected size of a bucket is n/N

- The ration λ = n/N is called “**Load Factor**” of the hash table
  - Bounded by a small constant (preferably below 1)

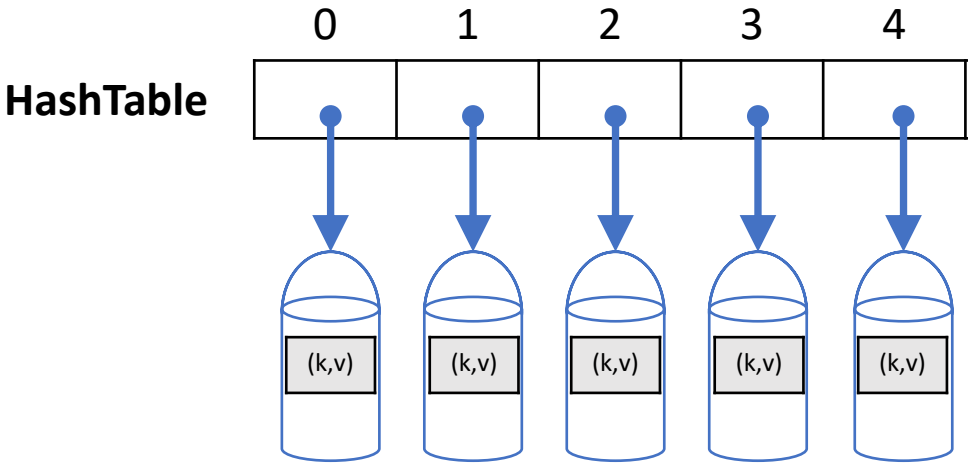
  - Example:
    - n = 15
    - N = 10

    ➔ λ = 1.5 → collision



Hash Tables | Collision-Handling Schemes

- Dynamic resizing - Double the size of hash table when load factor increases beyond 1



- Implement a method to double the size of hash table
  - Python resizes hash table when Load factor > 0.66

- After resizing, all the items in the old hash table has to be added in the new hash table → O(n)

Hash Tables

Collision-Handling Schemes

Open Addressing

- The nice property of chaining method is its easy implementation
  - Drawback → requires auxiliary data structure – list to hold items with colliding keys

- For applications where space is an issue (like hand-held devices)
  → We can consider storing each item in a separate table slot
  → Load factor is always at most required to be 1
  → Dealing with “collisions” becomes more complicated

- This approach of storing each element in a separate bucket in the bucket array (hash table) having a load factor of at most 1 is called “Open Addressing”.

Hash Tables | Collision-Handling Schemes | Open Addressing

- There are couple of variants of Open Addressing

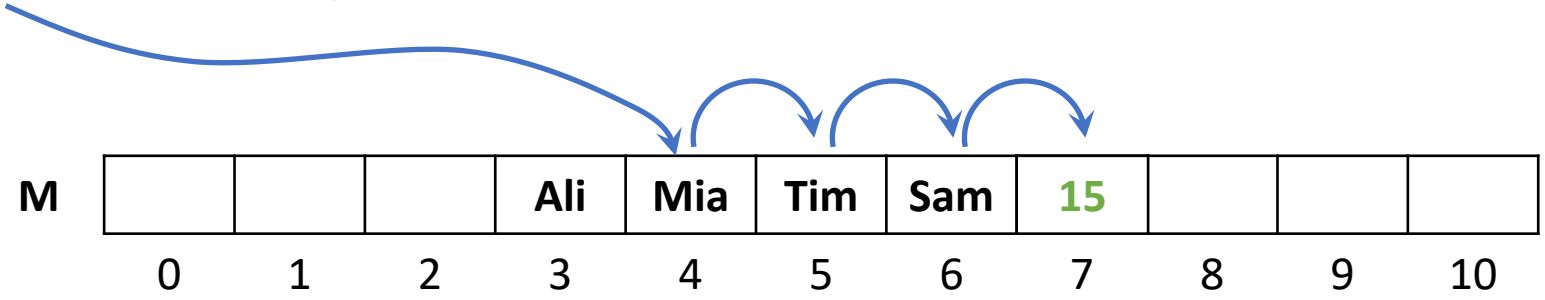
Linear Probing

- Inserting an element (k,v) at M[j]
  - j is the index generated by hash function
- If j^{th} place is occupied, we try M[(j+1) % N]
- If this place is also occupied, we next try M[(j+2) % N], and so on

Hash Tables | Collision-Handling Schemes | Open Addressing

Linear Probing – Example

- Inserting a new element with key k = 15 → k mod N → 15 mod 11 = 4
- This new item should be placed at location 4

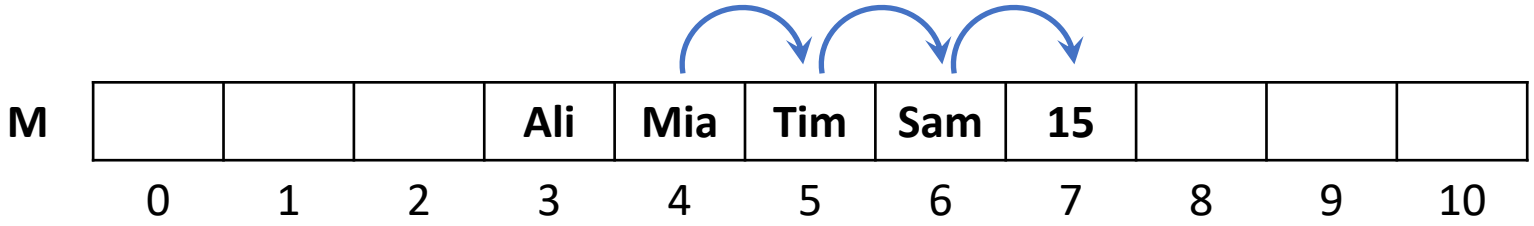


- This requires additional implementation to search for an existing key
- Accessing cell array is analogous to probing the bucket to find its content

Hash Tables | Collision-Handling Schemes | Open Addressing

Linear Probing – Searching

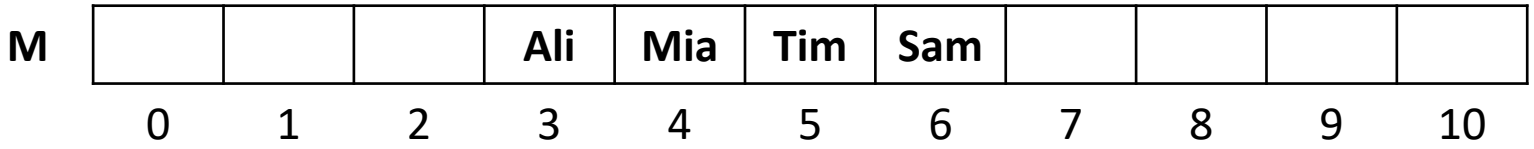
- To locate items in the hash-table, we start off by reading a key from M[h(k)]
- Then keep moving forward until either the element is found or an empty space is encountered



Hash Tables | Collision-Handling Schemes | Open Addressing

Linear Probing – Deleting

- Item cannot be simply deleted as soon as it is located



- Deleted location has to be masked with ”sentinel”