CSCI 2110 Data Structures and Algorithms

Module 8: Hashing and Hash Tables

MOTIVATION

- Name the three most important and common operations that you would perform on any data structure.

- **SEARCH, SEARCH AND SEARCH!**
- How did we do in terms of the time complexity of search on different data structures?
  - Unordered List: **O(n)**
  - Ordered List: **O($\log_2 n$)** (because of binary search)
  - Binary Tree: **O(n)** (because it can be just a linear set of nodes)
  - Binary Search Tree: **O($\log_2 n$)** if the tree is complete (balanced) – Good tree!
    O(n) if the tree is not balanced – Bad tree!

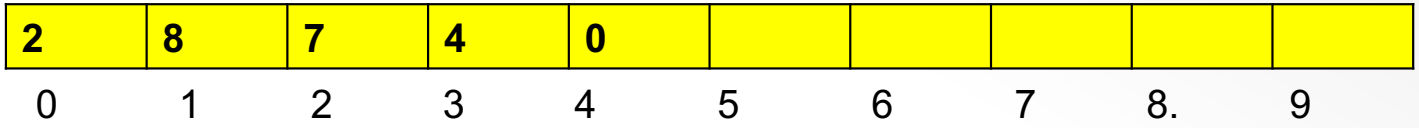
- **CAN WE DO BETTER???**

Can we aim for O(1) complexity for search?

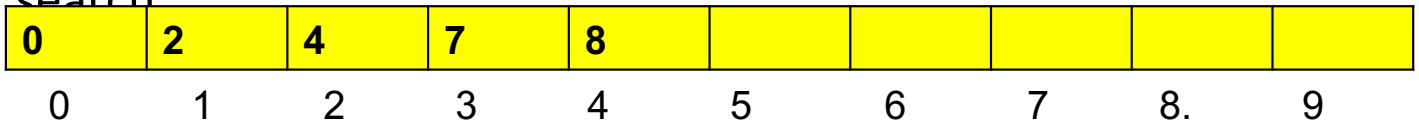
- Let's set ourselves a tall order: O(1) for search (at least on the average).
- We know that in an array, given its index, we can retrieve an element in O(1) time.
- Can we look back to the array for help?
- How can we put a set of keys onto an array so that the search complexity is O(1)?

Can we aim for O(1) complexity for search?

- Suppose we have an arbitrary set of keys 2, 8, 7, 4, 0.
- Remember that each key is associated with an object: if you get the key, you get the object.
- If we store the keys in the array arbitrarily, we need O(n) time → Sequential search

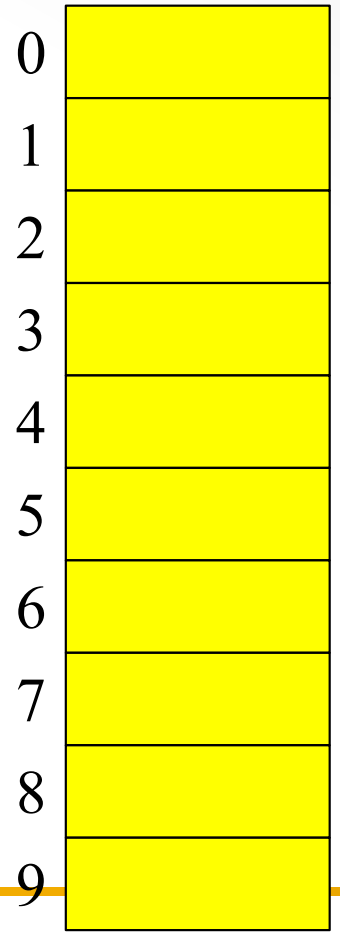


- If we store the keys in the array sorted, we still need O($\log_2$ n) time → Binary search

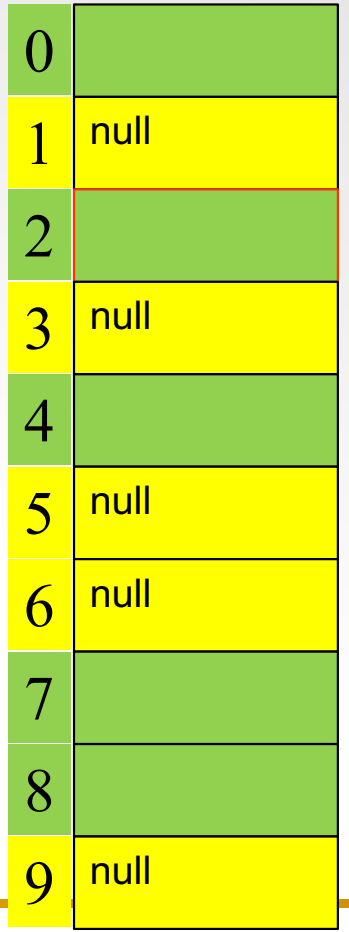


- **BUT WHAT IF WE MAKE THE KEY THE INDEX?**
- **Time for search: O(1) (insert exclamations here).**

Here's the idea:



Keys:
2, 8, 7, 4,0



What is the catch?

- In our example, we chose the keys to be 2, 8, 7, 4, 0 and hence we declared an array of size 10.
- **What if the keys are 1, 43, 102, 131,15143 and 19992?**
- Would you declare an array of size 20000?
- It appears that whatever you gained in time complexity was lost in space complexity.
- Can we have the best of both – good time complexity and good space complexity?
- ***The trick: MAP the key to an index!***
- **The process: HASHING**
- **The data structure: HASH TABLE**

Hashing and Hash Table - Definition

- A hash table is a storage array.
- A hash function maps a key to a specific index on the hash table.
- Hashing is the process of storing objects in the hash table using a hash function.

A simple hash function or mapping function can be written as:

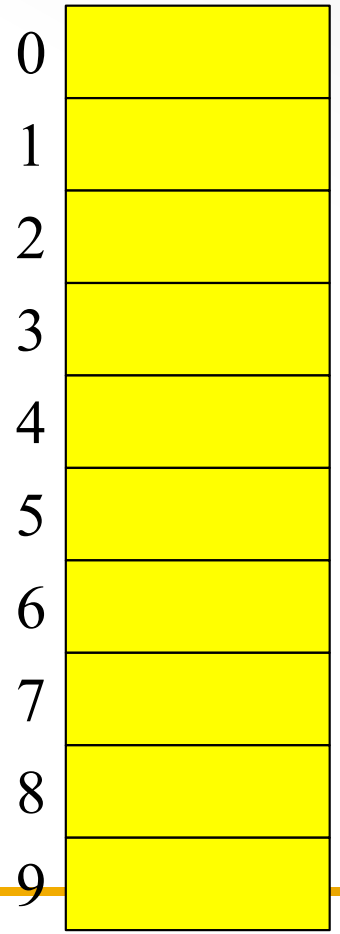
key % N

where N is the table size.

For example, if the table size is 10, then we get our index as follows:

index = key % 10

Example



Suppose we have these 5 keys:
9876, 234, 5672, 89990, 6777

We choose a table of size 10 (enough to fit the number of keys).

Then we apply the hash function

index = key % 10

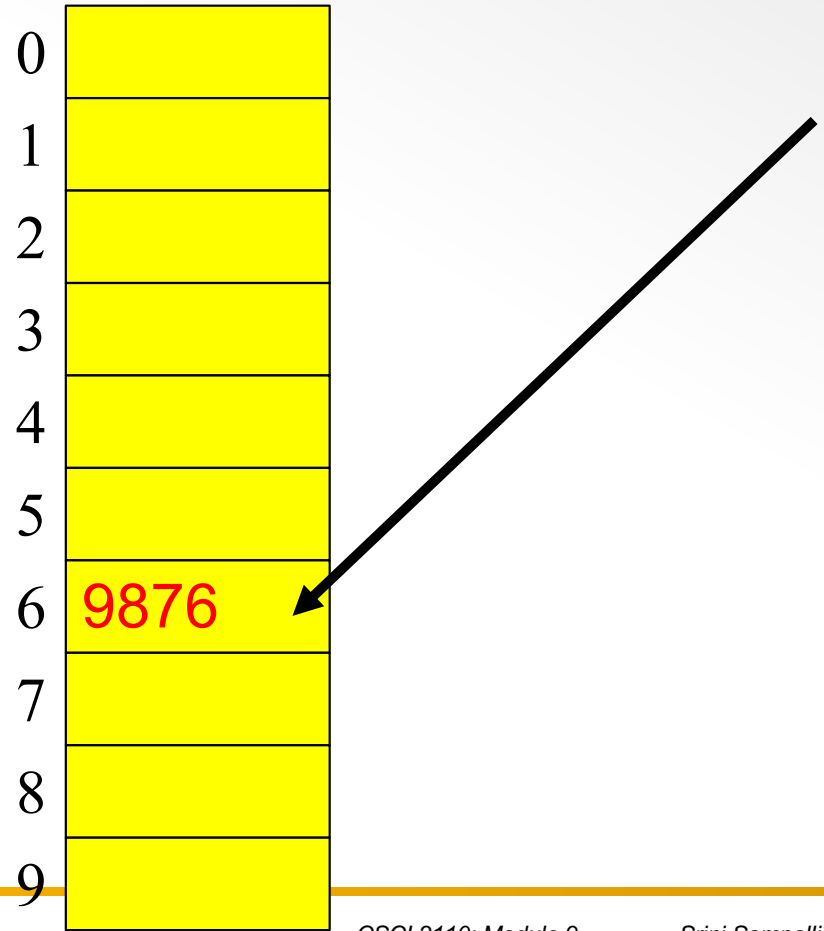
to map the keys to the table.

Example

Keys:
9876, 234, 5672, 89990, 6777

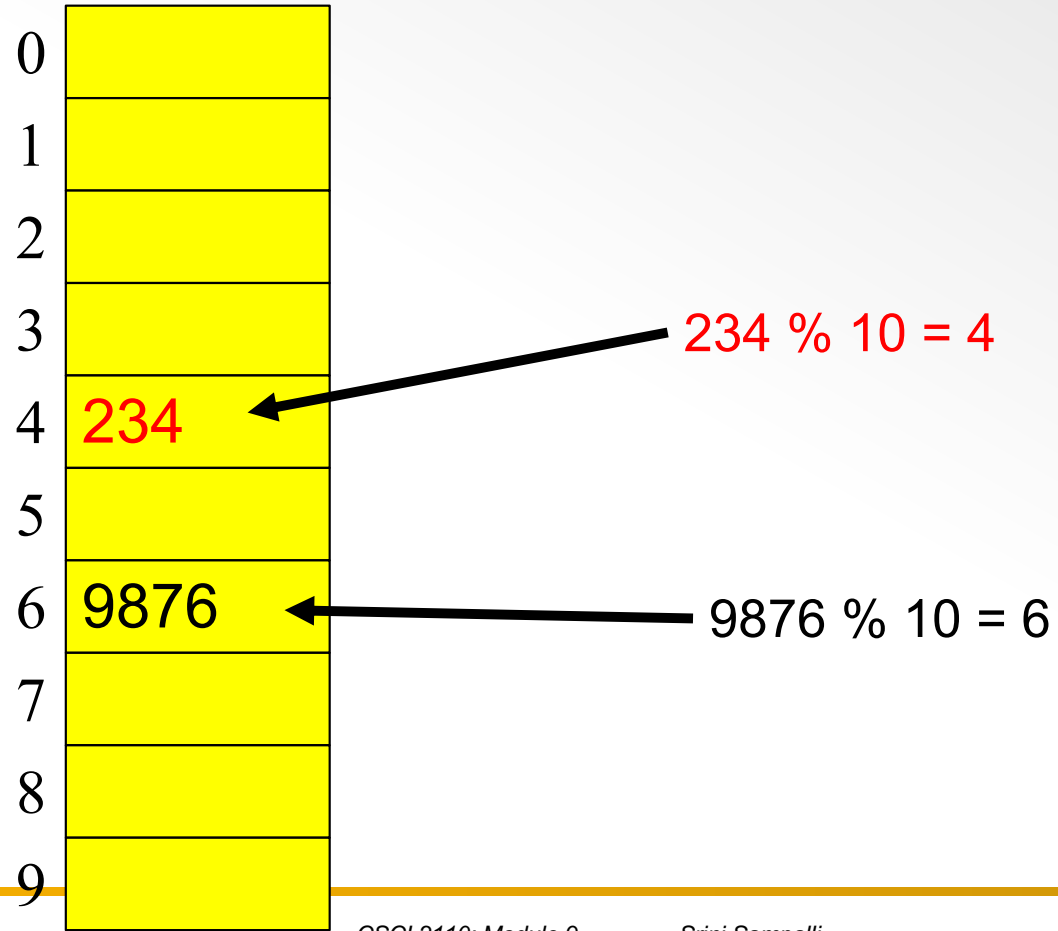
9876 % 10 = 6

Key 9876 goes to location 6



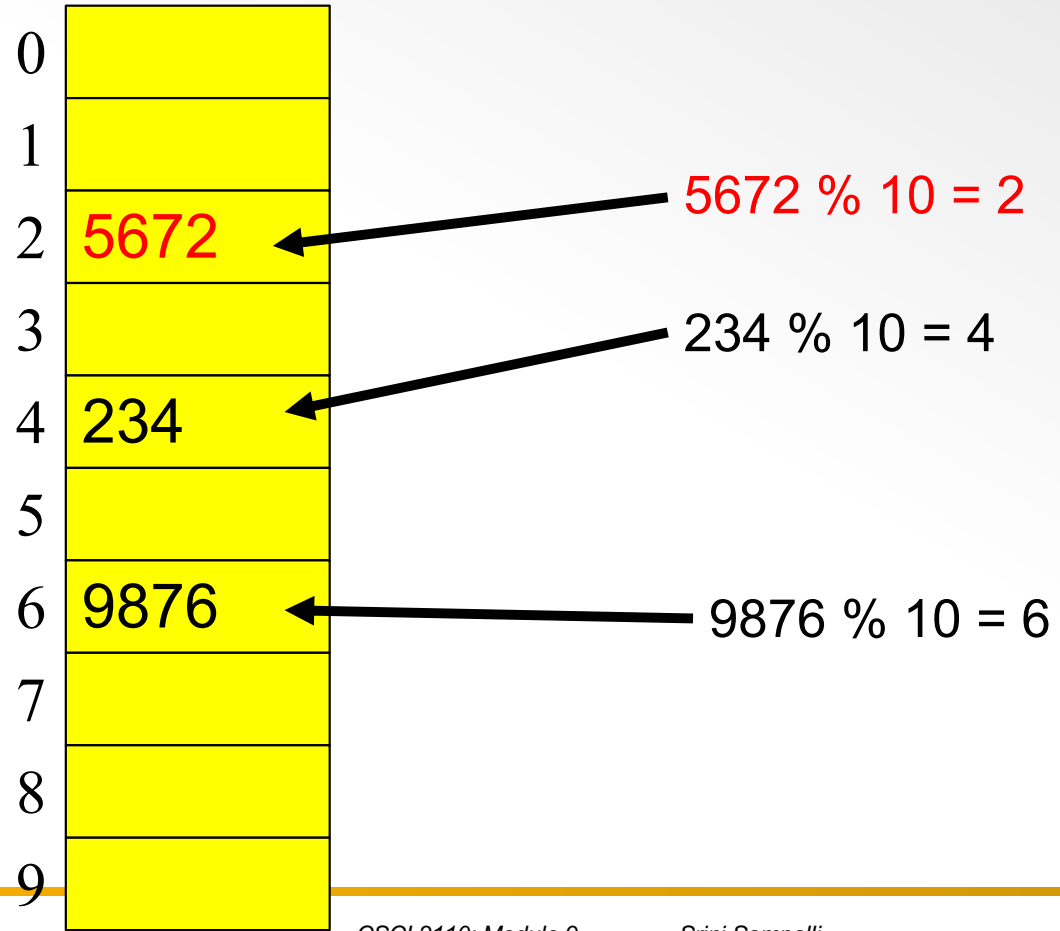
Example

Keys:
9876, 234, 5672, 89990, 6777



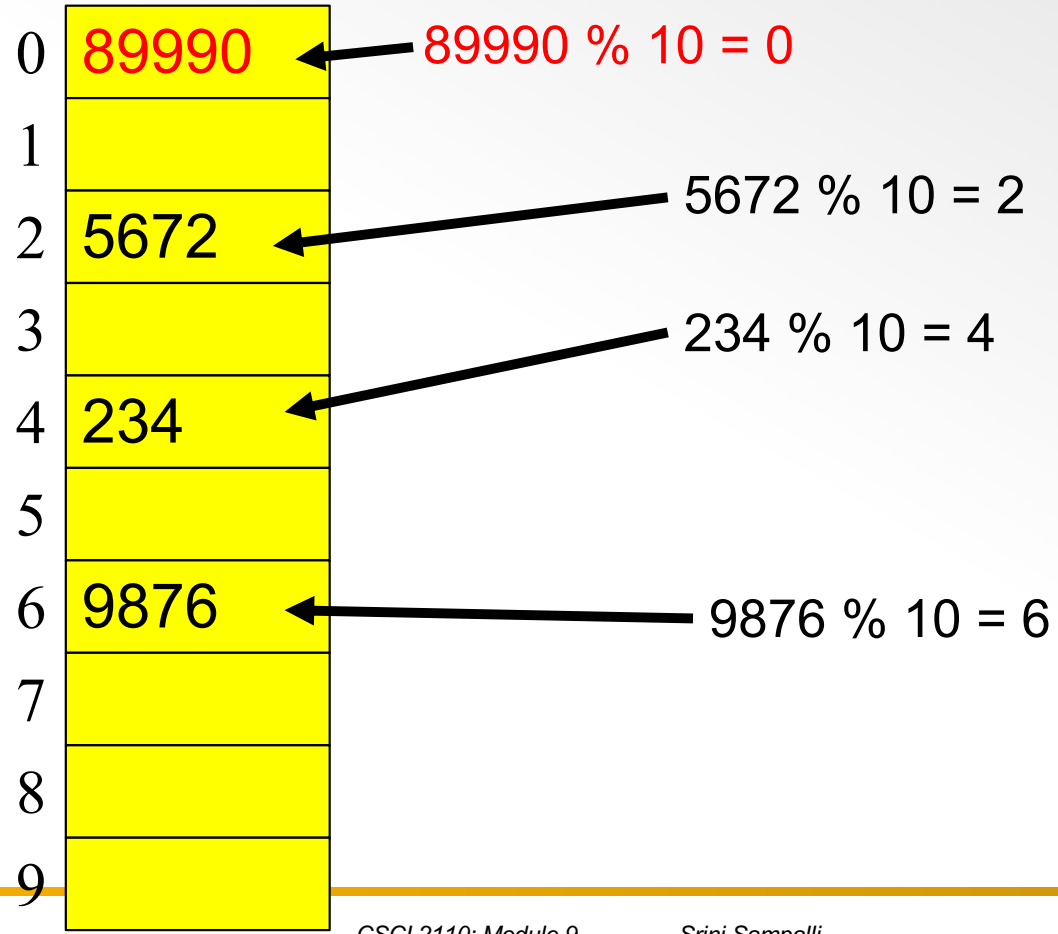
Example

Keys:
9876, 234, 5672, 89990, 6777



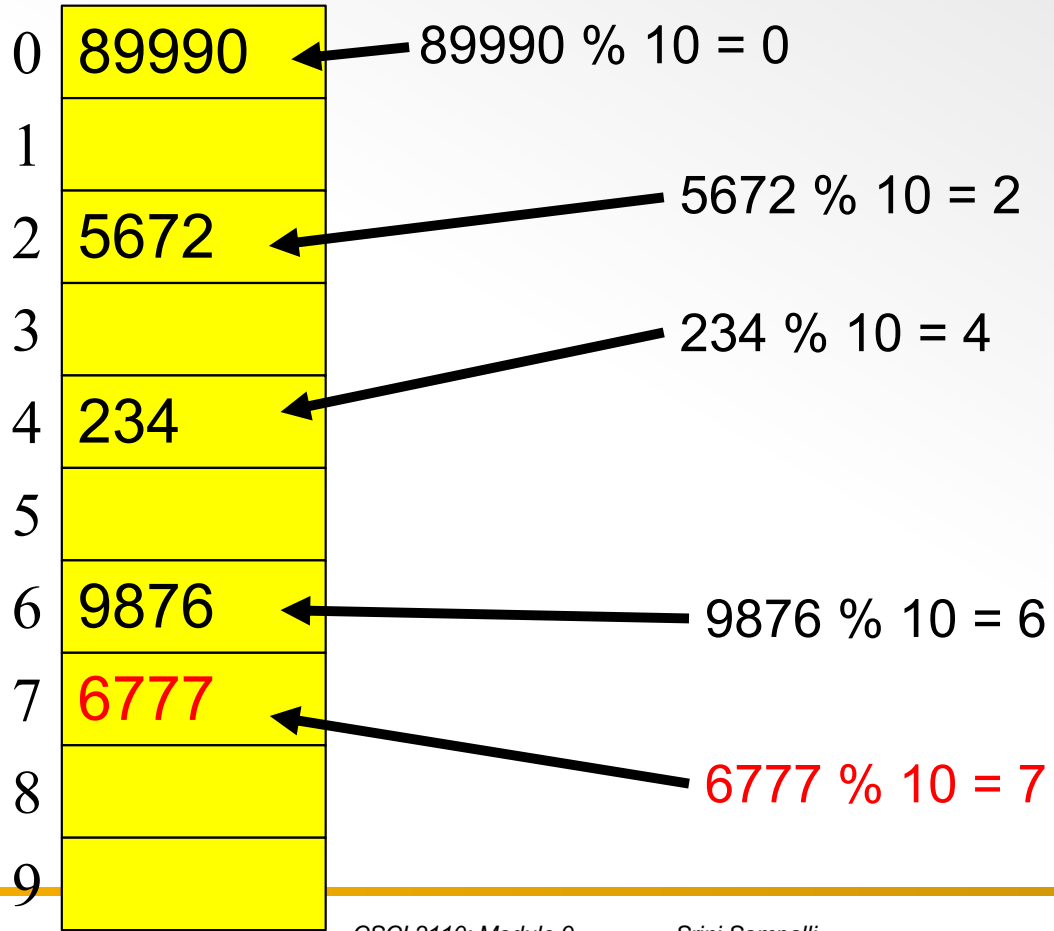
Example

Keys:
9876, 234, 5672, 89990, 6777

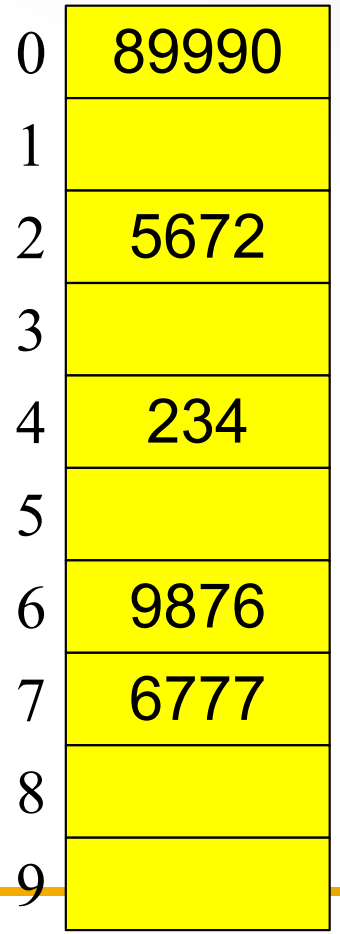


Example

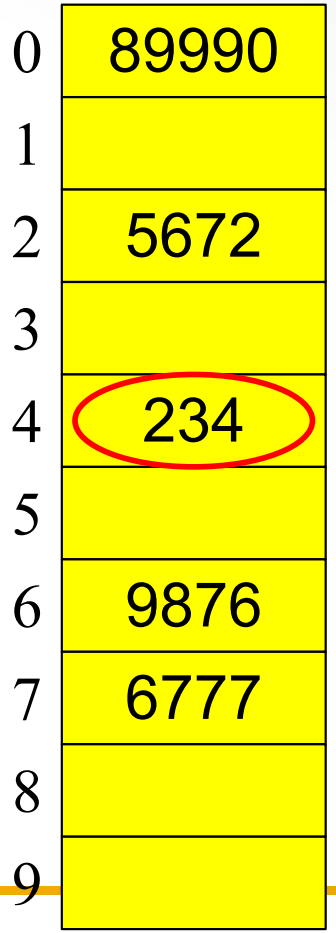
Keys:
9876, 234, 5672, 89990, 6777



Now look at the search operations!

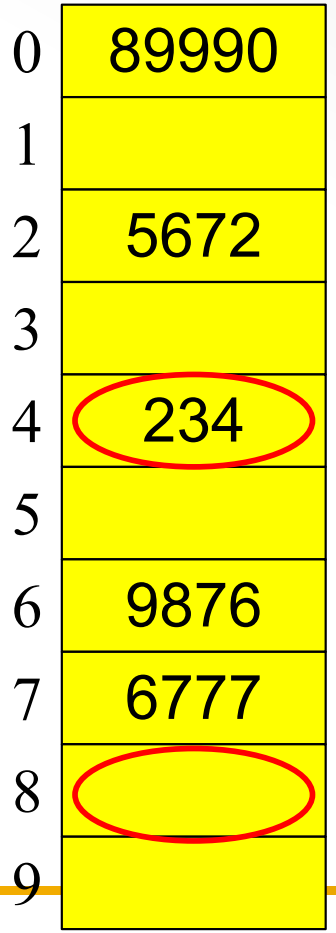


Now look at the search operations!



Search for key 234
Apply the hash function
234 %10 = 4
Go to location 4
Found!
O(1) time complexity!

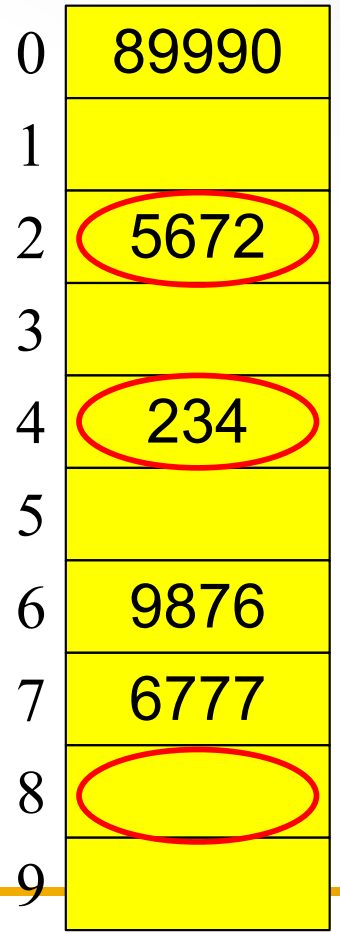
Now look at the search operations!



Search for key 234
Apply the hash function
234 %10 = 4
Go to location 4
Found!
O(1) time complexity!

Search for key 878
Apply the hash function
878 %10 = 8
Go to location 8
Not found!
O(1) time complexity!

Now look at the search operations!



Search for key 234
Apply the hash function
234 %10 = 4
Go to location 4
Found!
O(1) time complexity!

Search for key 878
Apply the hash function
878 %10 = 8
Go to location 8
Not found!
O(1) time complexity!

Search for key 312
Apply the hash function
312 %10 = 2
Go to location 2
Not found!
O(1) time complexity!

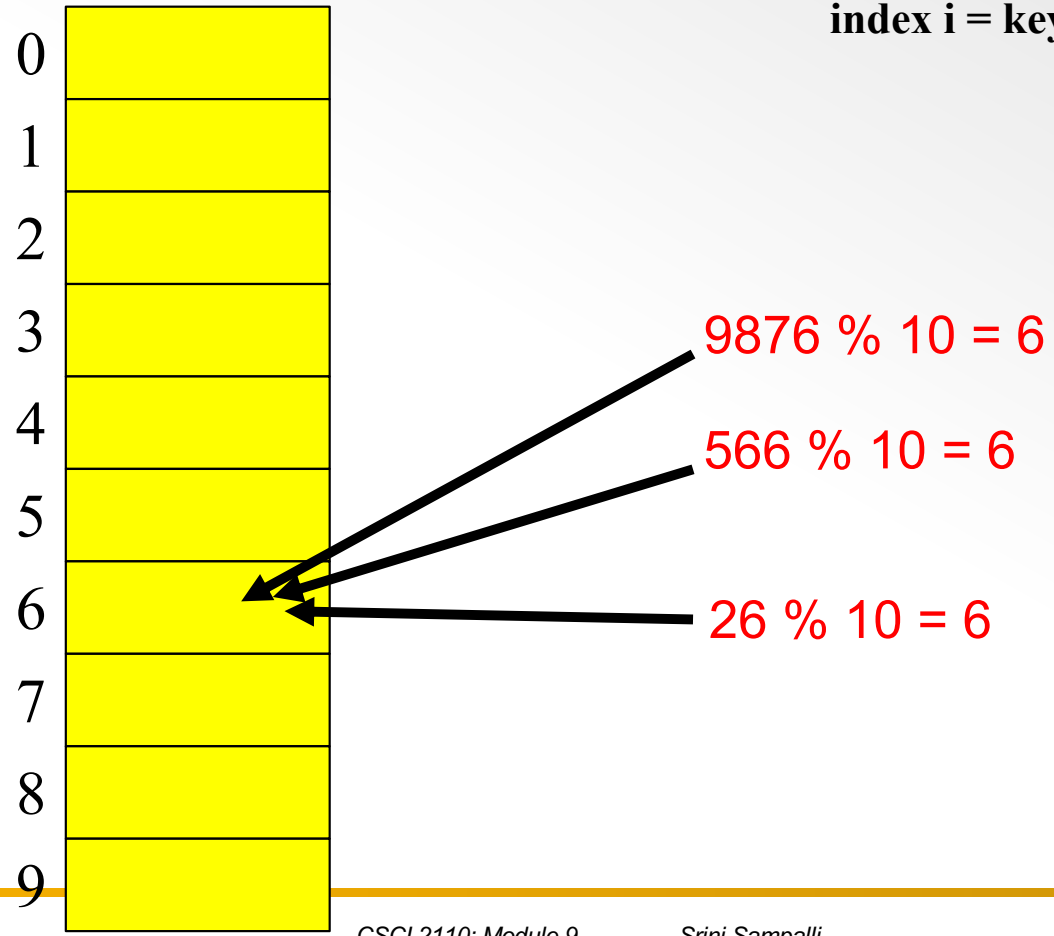
Problem:
What if two or more keys map to the same index?

This is bound to happen since we are mapping a large range of keys into a small table.

Example

Keys:
9876, 566 and 26

index i = key % 10



HASH CLASH!

The Hash Clash Problem

If two or more keys map to the same index, it is called a Hash Clash or a Hash Collision.

The Hash Clash Problem

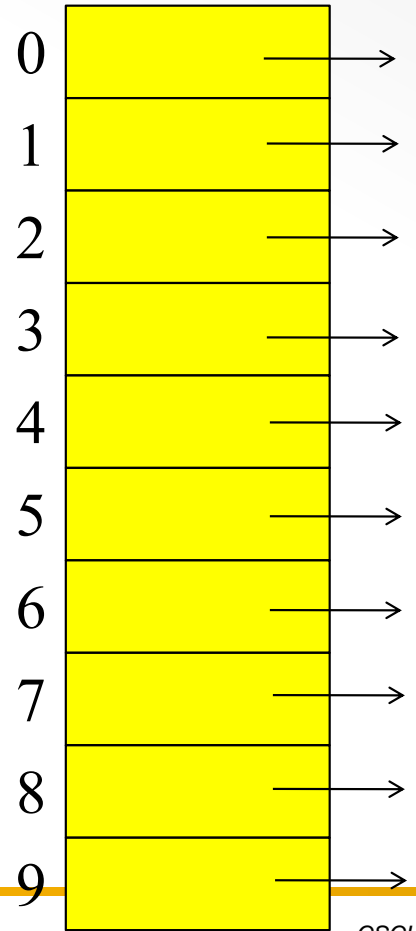
If two or more keys map to the same index, it is called a Hash Clash or a Hash Collision.

Two Solutions:

- ***OPEN HASHING (SEPARATE CHAINING):***
- ***CLOSED HASHING***

Open Hashing or Separate chaining

Concept: Create an ArrayList of Linked Lists

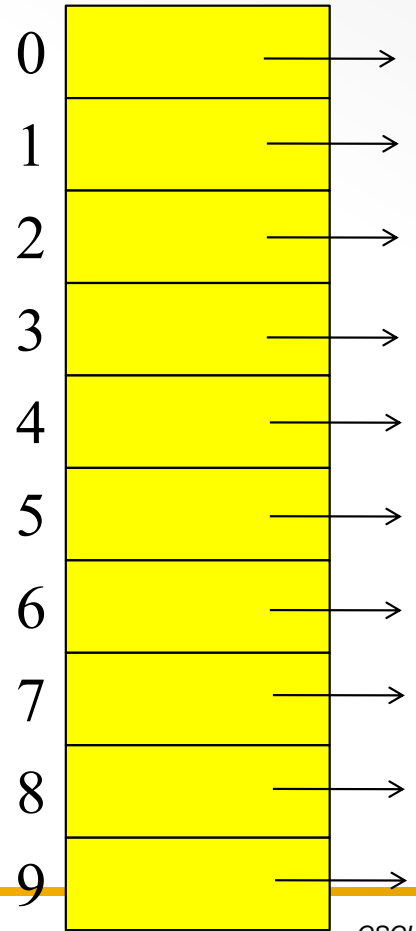


Keys:

9876, 234, 5672, 89990, 6777, 366, 7446, 212, 4676

Open Hashing or Separate chaining

Concept: Create an ArrayList of Linked Lists

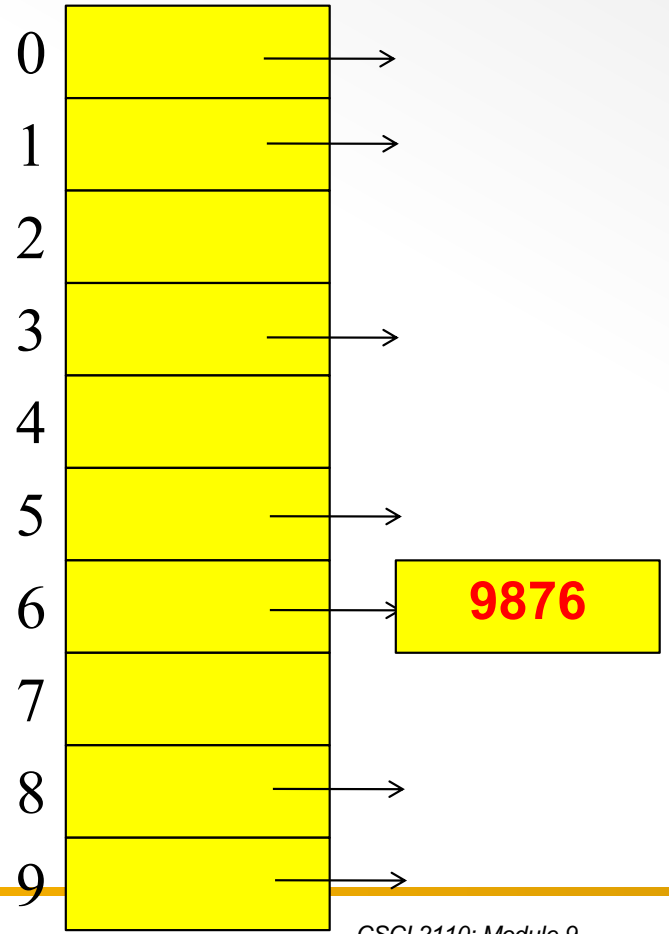


Keys:

9876, 234, 5672, 89990, 6777, 366, 7446, 212, 4676

Open Hashing or Separate chaining

Concept: Create an ArrayList of Linked Lists

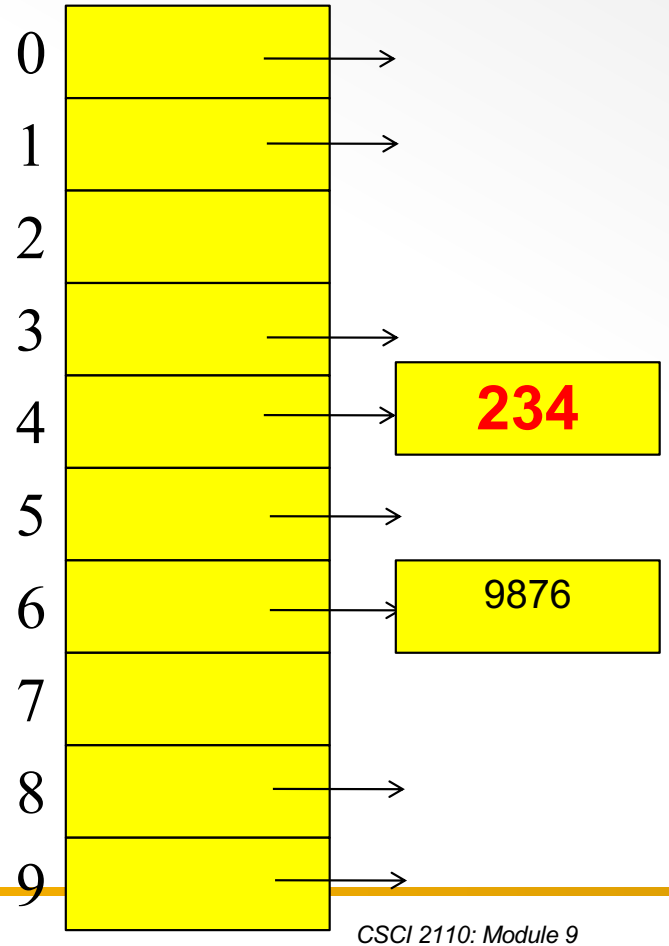


Keys:

9876, 234, 5672, 89990, 6777, 366, 7446, 212, 4676

Open Hashing or Separate chaining

Concept: Create an ArrayList of Linked Lists

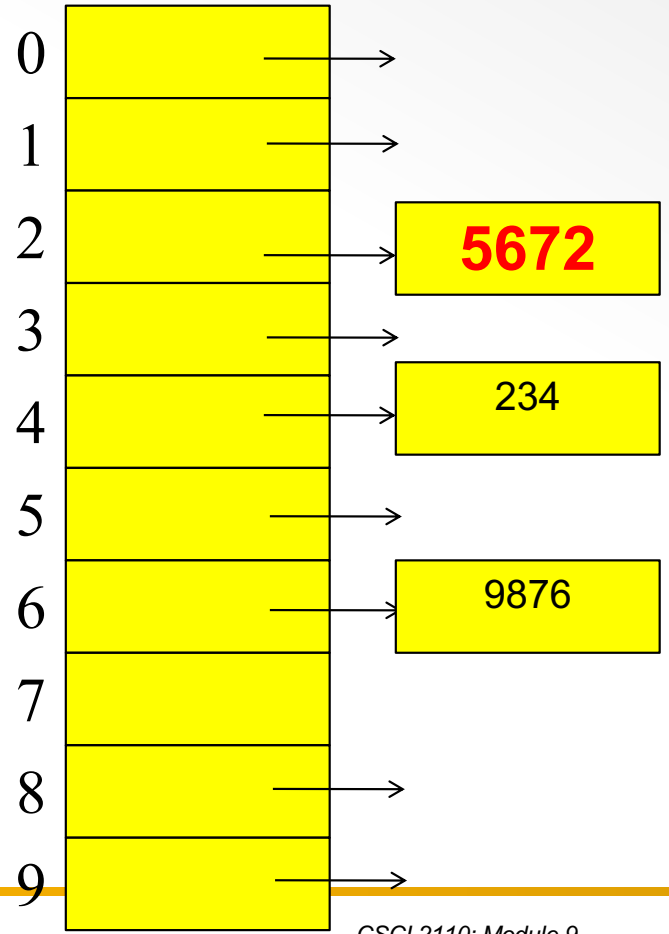


Keys:

9876, 234, 5672, 89990, 6777, 366, 7446, 212, 4676

Open Hashing or Separate chaining

Concept: Create an ArrayList of Linked Lists

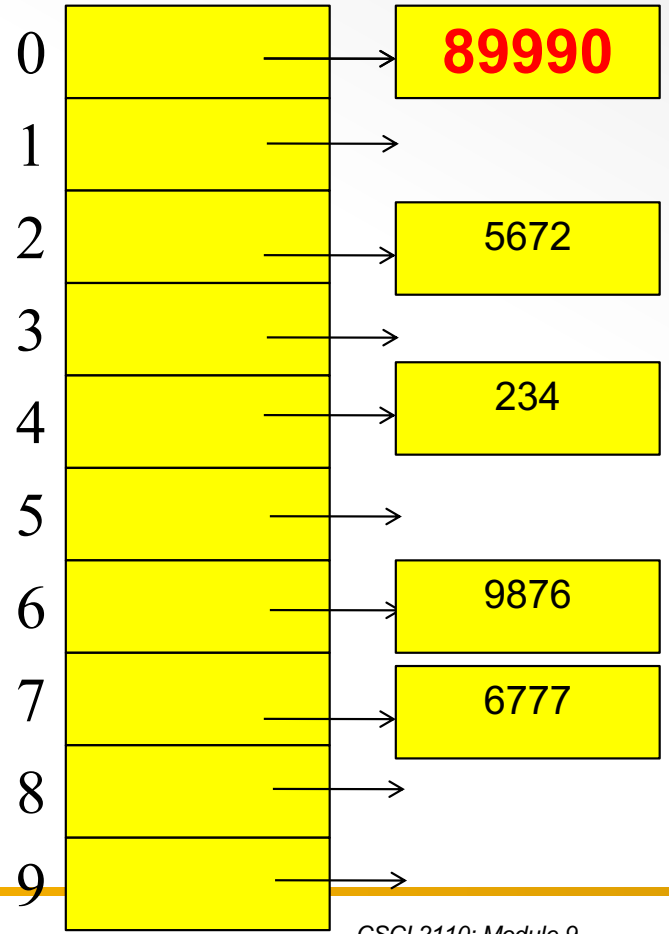


Keys:

9876, 234, 5672, 89990, 6777, 366, 7446, 212, 4676

Open Hashing or Separate chaining

Concept: Create an ArrayList of Linked Lists

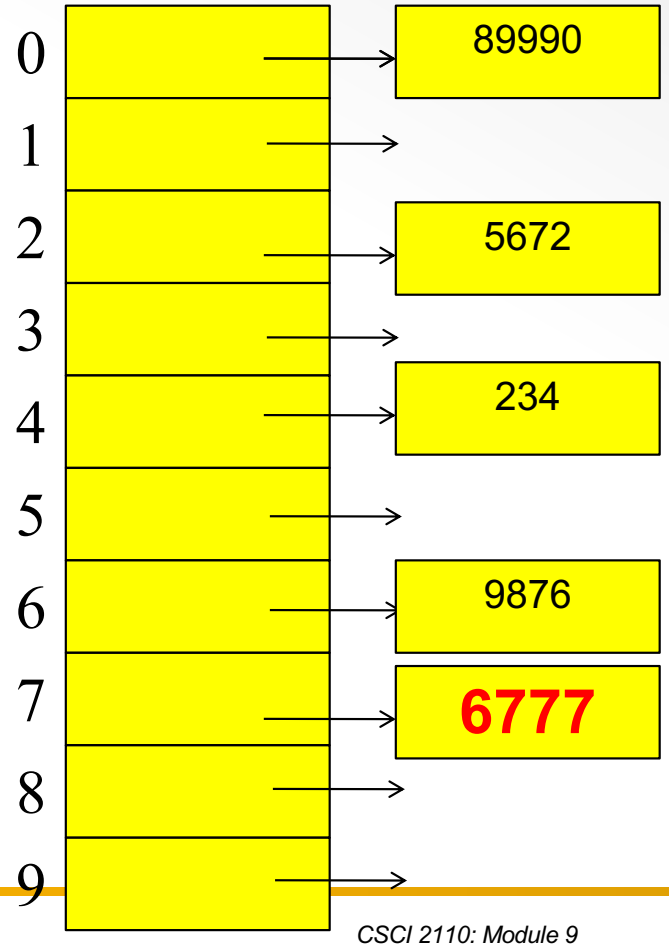


Keys:

9876, 234, 5672, 89990, 6777, 366, 7446, 212, 4676

Open Hashing or Separate chaining

Concept: Create an ArrayList of Linked Lists

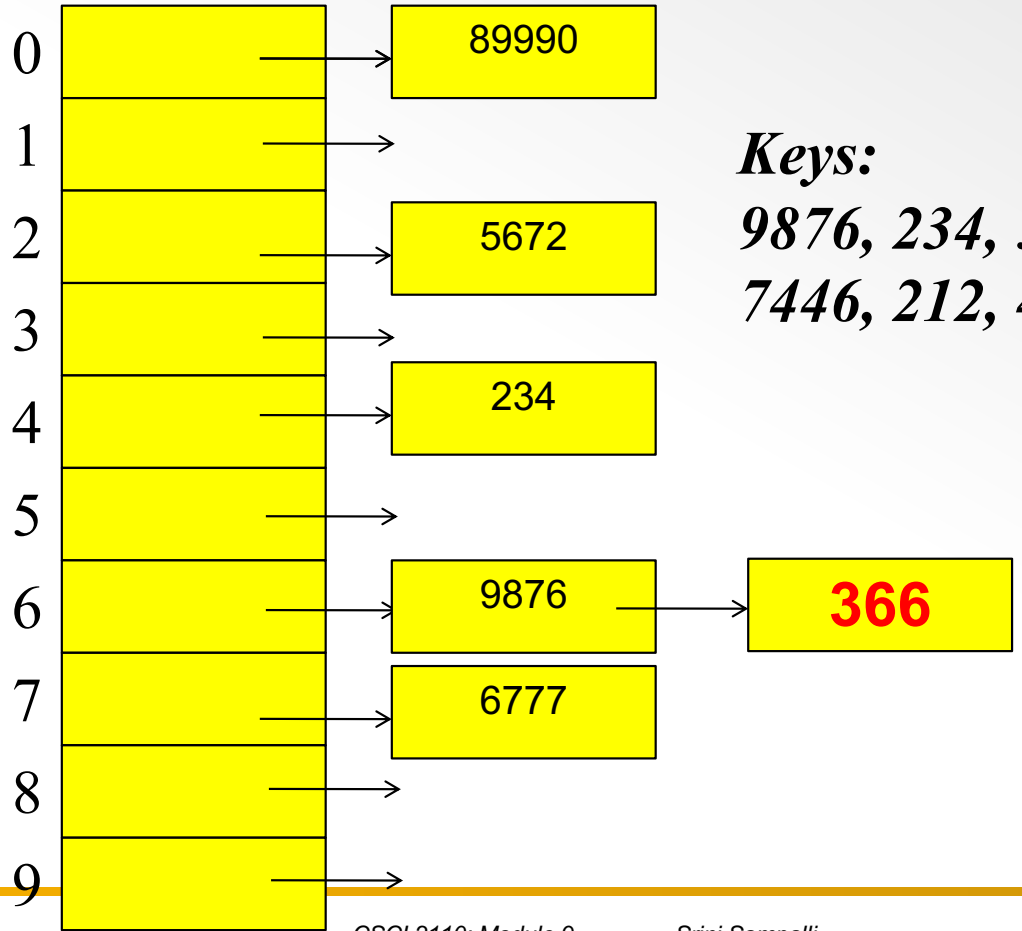


Keys:

9876, 234, 5672, 89990, 6777, 366, 7446, 212, 4676

Open Hashing or Separate chaining

Concept: Create an ArrayList of Linked Lists

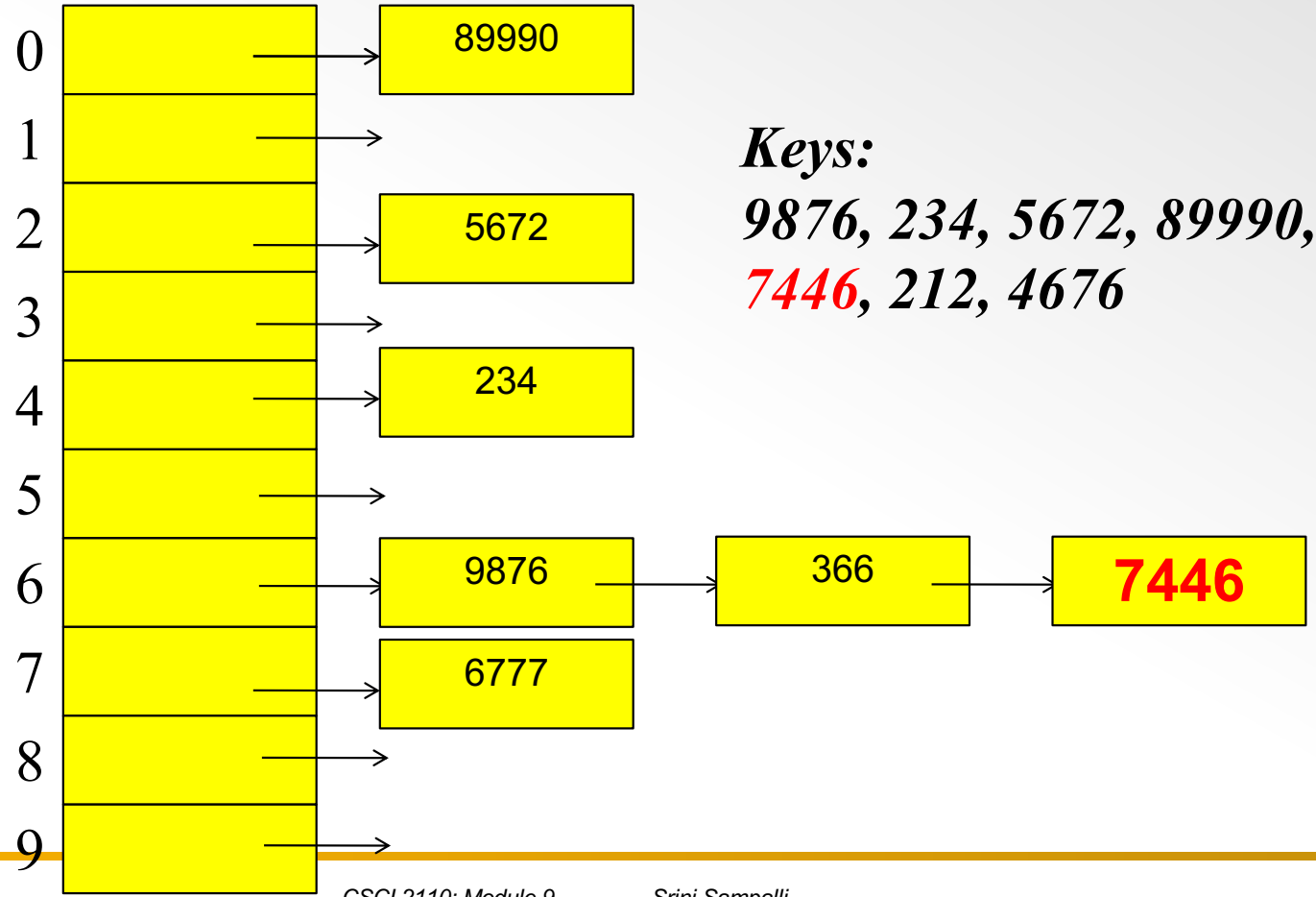


Keys:

9876, 234, 5672, 89990, 6777, 366, 7446, 212, 4676

Open Hashing or Separate chaining

Concept: Create an ArrayList of Linked Lists

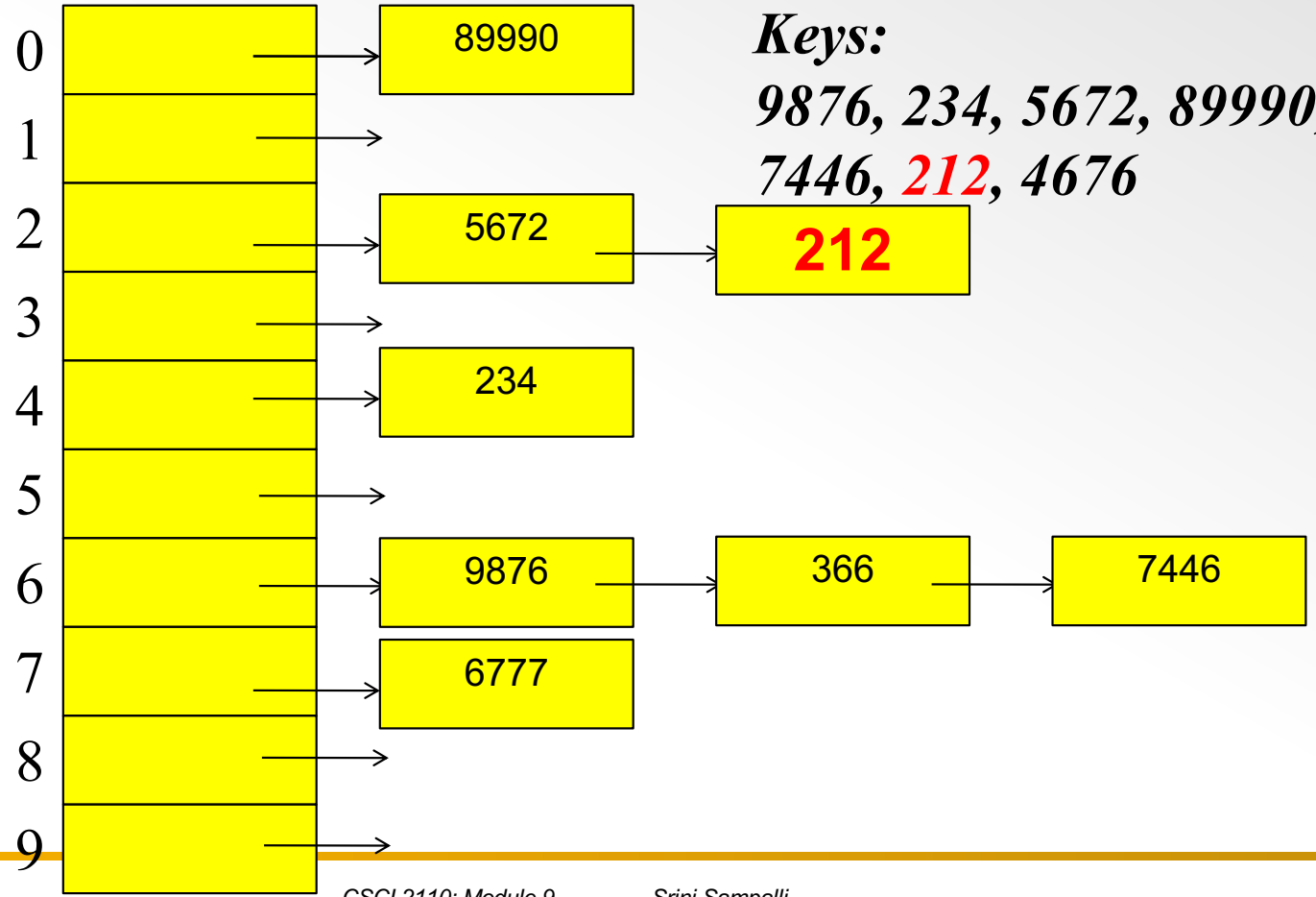


Keys:

9876, 234, 5672, 89990, 6777, 366, 7446, 212, 4676

Open Hashing or Separate chaining

Concept: Create an ArrayList of Linked Lists

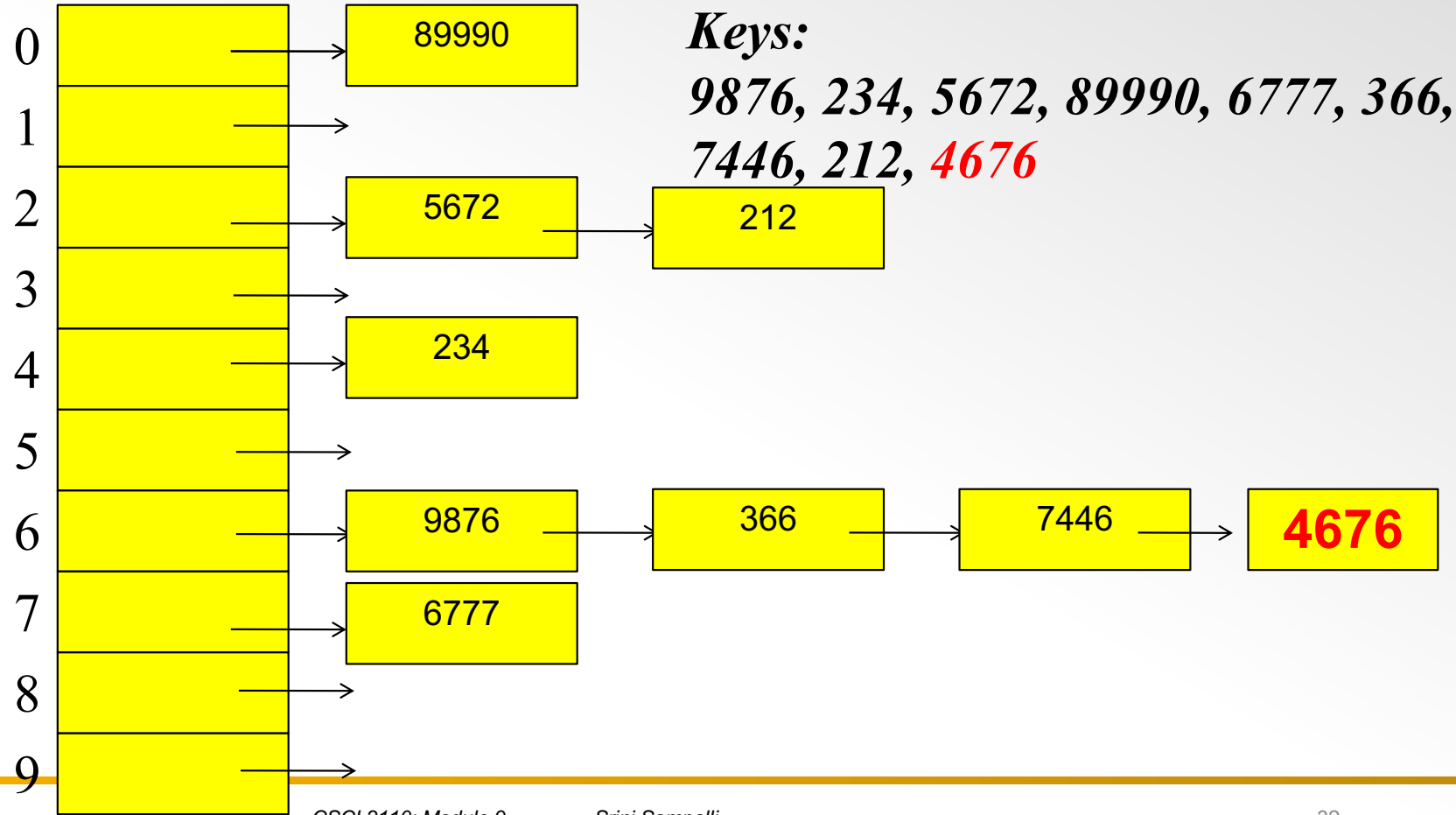


Keys:

9876, 234, 5672, 89990, 6777, 366, 7446, 212, 4676

Open Hashing or Separate chaining

Concept: Create an ArrayList of Linked Lists

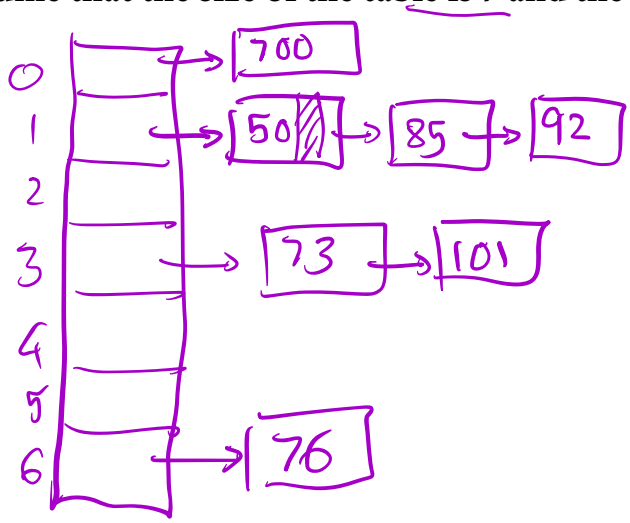


Keys:

9876, 234, 5672, 89990, 6777, 366, 7446, 212, 4676

MORE EXAMPLES WITH OPEN HASHING

1. **Draw the hash table with open hashing resulting from the insertion of the following sequence of keys**
   50, 700, 76, 85, 92, 73, 101
   Assume that the size of the table is 7 and the hash function is key mod 7

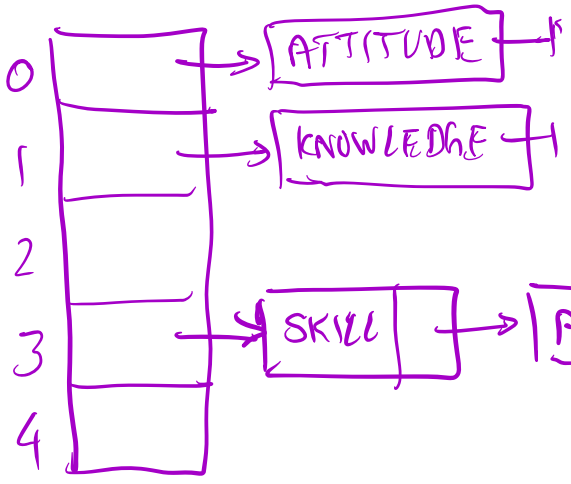


50 mod 7 = 1
700 mod 7 = 0
76 mod 7 = 6
85 mod 7 = 1
92 mod 7 = 1
73 mod 7 = 3
101 mod 7 = 3

2. Draw the hash table with open hashing resulting from the insertion of the following sequence of keys (words):
SKILL, ABILITY, KNOWLEDGE, HARDWORK, ATTITUDE

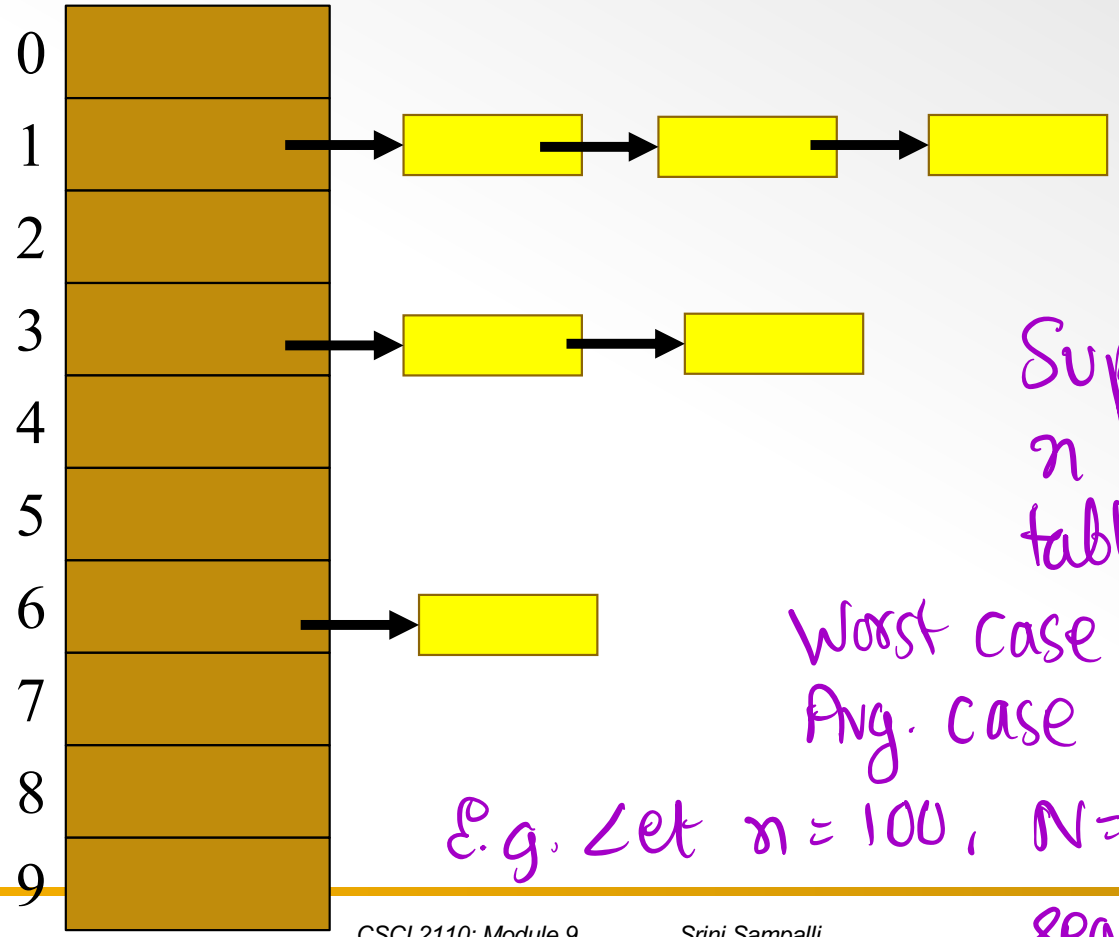
Assume that the size of the table is 5 and the hash function is defined as follows: add the positions of the word's letters in the alphabet (A=1, B=2, etc.) and compute the sum's remainder after division by 5.

SKILL	**= 19 + 11 + 9 + 12 + 12**	**= 63**
ABILITY	**= 1 + 2 + 9 + 12 + 9 + 20 + 25**	**= 78**
KNOWLEDGE	**= 11 + 14 + 15 + 23 + 12 + 5 + 4 + 7 + 5**	**= 96**
HARDWORK	**= 8 + 1 + 18 + 4 = 23 + 15 + 18 + 11**	**= 98**
ATTITUDE	**= 1 + 20 = 20 + 9 + 20 + 21 + 4 + 5**	**= 100**



63 mod 5 = 3
78 mod 5 = 3
96 mod 5 = 1
98 mod 5 = 3
100 mod 5 = 0

Separate chaining: An arraylist of linked lists - Search Complexity



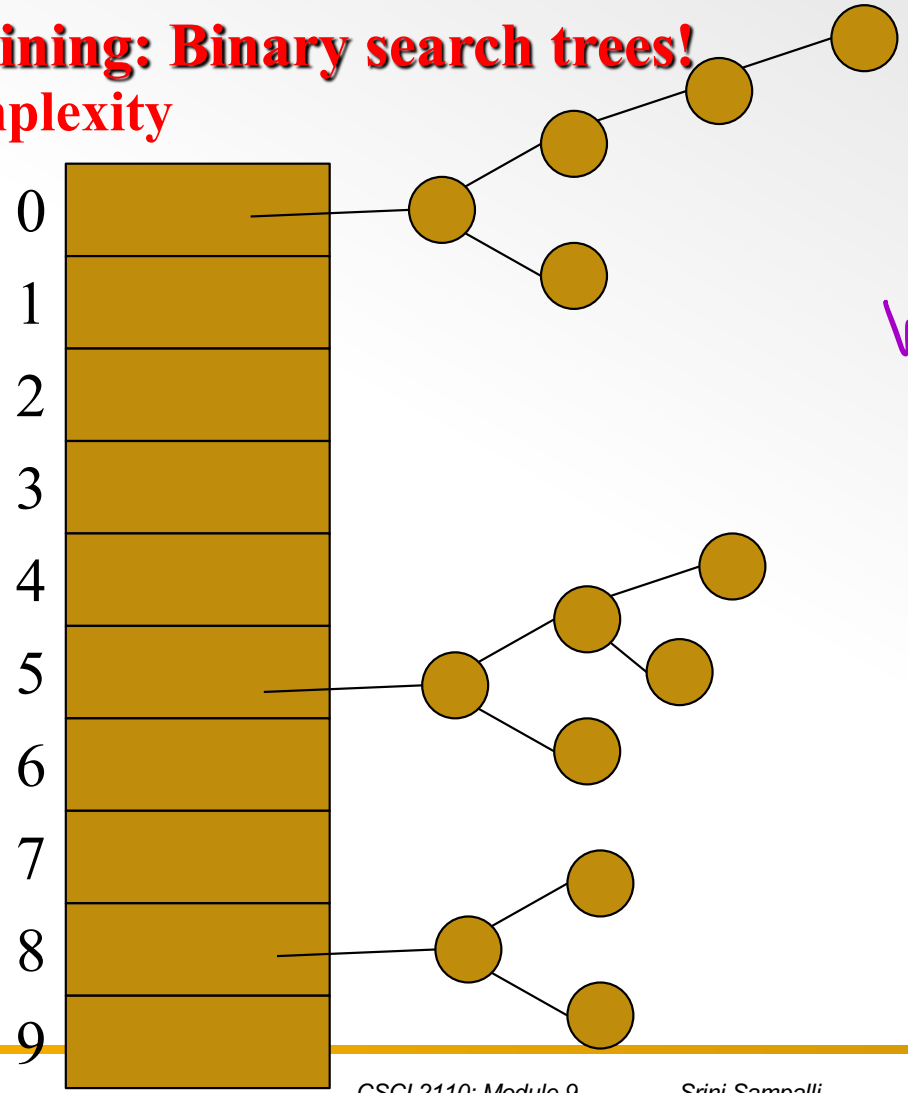
Suppose there are n keys & the table size is N.

Worst case complexity = $O(n)$

Avg. case complexity = $O\left(\frac{n}{N}\right)$

E.g. Let $n = 100$, $N = 10$. Avg # searches = $\frac{100}{10} = 10$

Separate chaining: Binary search trees!
- Search Complexity

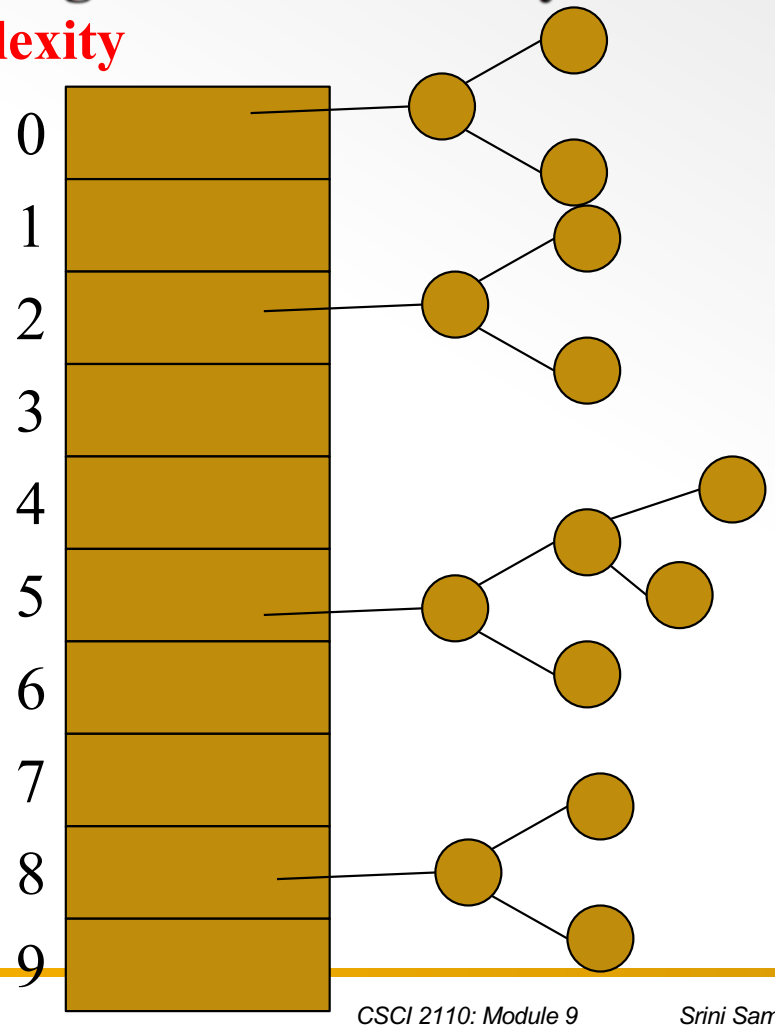


Worst Case Complexity $= O(n)$

Average case complexity $= O\left(\frac{n}{N}\right)$

Separate chaining: Balanced Binary Search Trees!

- Search Complexity



Worst case complexity = $O(\log_2 n)$

Average case complexity = $O\left(\frac{\log_2 n}{N}\right)$

inching closer & closer to $O(1)$

2. Closed Hashing

Concept: Hash tables store keys directly within the array. A collision is resolved by trying alternative locations within the array until a free spot is found.

There are two important techniques for trying alternative locations:

a) Linear Probing

b) Quadratic Probing

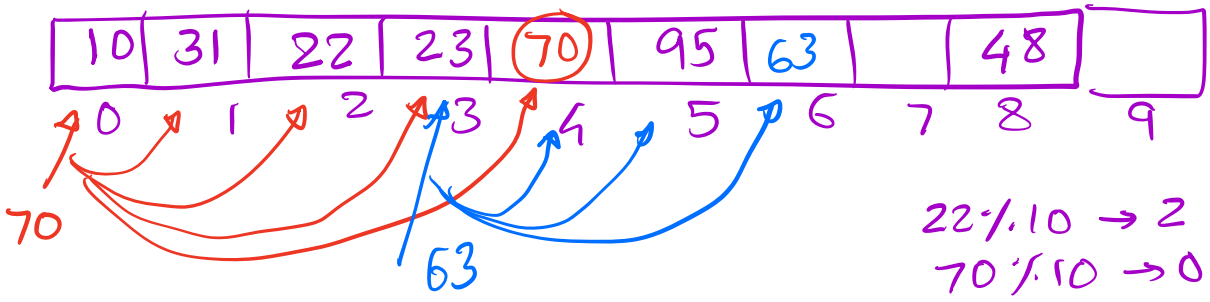
a) Linear Probing:

Find the index i where the key should be mapped using the hash function.

If a collision occurs at index i, try index i+1, i+2, i+3, etc. until an empty spot is found.

If you reach the end of the array, wraparound.

Example 1: keys are 22, 31, 10, 23, 95, 70, 63, 48



Hash function

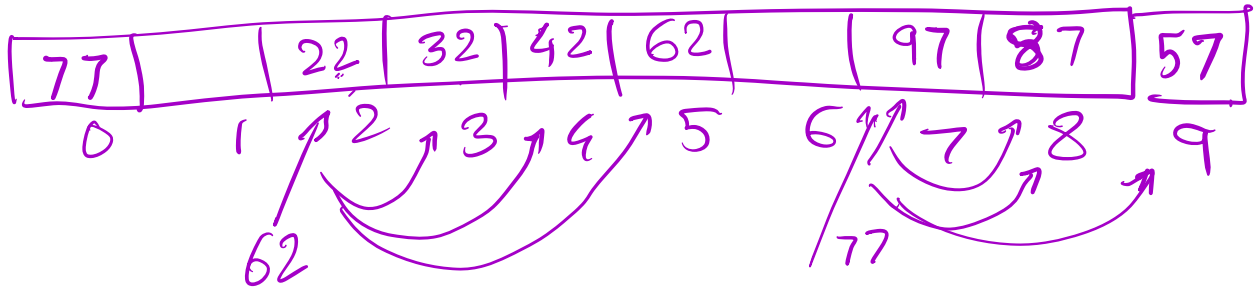
index = (key % 10 + x) % 10 ← for wraparound

where x = 0, 1, 2, 3, etc.

In general index = (key % table_size + x) % table_size

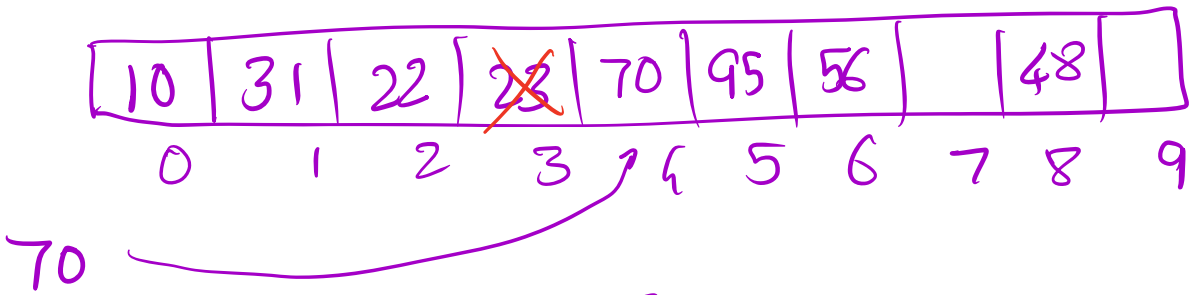
where x = 0, 1, 2, 3, etc.

Example 2: Insert keys 22, 32, 42, 97, 87, 57, 62, 77



Problems with linear probing:

1. Tends to form clusters (see example 2)
2. Problem with deletion



Delete key 23
Search for key 70
Empty spot in index 3. So search says 70 not found!

Solution: Lazy deletion.
Do not remove the deleted entry, just flag it as deleted.
Periodically, the array is flushed & the keys are remapped.

b) Quadratic Probing:

Find the index i where the key should be mapped using the hash function.

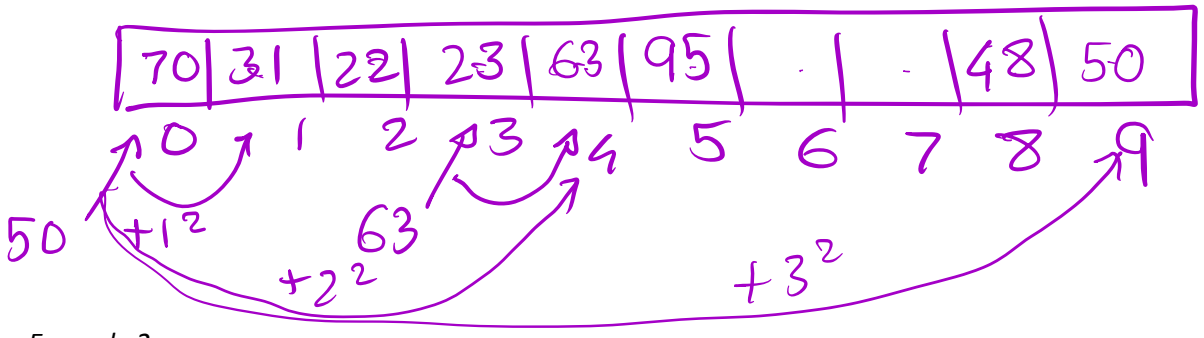
If a collision occurs at index i, try index $i+1^2$, $i+2^2$, $i+3^2$, etc. until an empty spot is found.

If you reach the end of the array, wraparound.

$$index = (key \% tablesize + x^2) \% tablesize$$

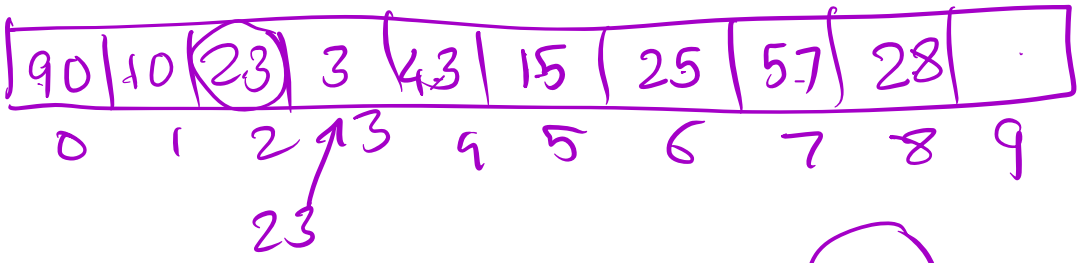
where $x = 0, 1, 2, 3,$ etc.

Example 1: Keys 22, 31, 23, 95, 70, 63, 48, 50

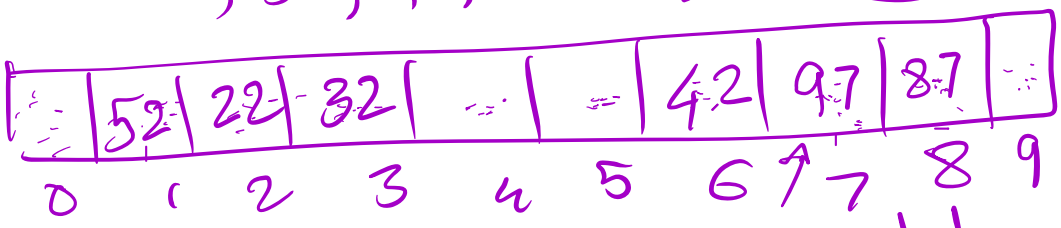


Example 2:

15, 25, 3, 43, 90, 28, 10, 57, 23



Example 3: 22, 32, 42, 52, 97, 87, 57



57 has no spot!

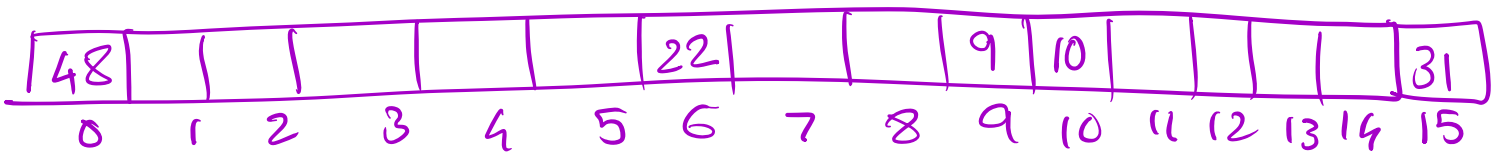
Solution: Tablesize is a prime number, & table is less than half full

HASH FUNCTIONS

We mainly considered a table of size 10 and a hash function h(key) = key % 10.
In general, we can write the hash function as

h(key) = key%N where N is the table size.

Example: Insert the keys 9, 22, 10, 31, and 48 using the hash function h(key) = key%N where N = 16 (table size).



9 % 16 = 9 10 % 16 = 10 48 % 16 = 0
22 % 16 = 6 31 % 16 = 15

The above is just one example of a hash function.
A host of other hash functions can be used.

In general, a good hash function:

- Must have O(1) time complexity
- Must distribute keys uniformly (should not favour one location over another).

Encryption

3 goals in cybersecurity: Cryptographic hashes ←
- Confidentiality: No eavesdropping
- Integrity: Message sent = Message received
- Authentication: You are who you say you are.

APPLICATIONS OF HASHING IN CRYPTOGRAPHY

1. **PASSWORD STORAGE**

M → hash → Message Digest

Let password be P.
Hash of password is generated: h(P)
A secret key is used to generate a random no. → Salt.
h(P) + salt is stored in the server.

2. **FILE MODIFICATION DETECTOR**

For each file F, compute hash (F) & store it securely.
Check if F is modified by verifying hash (F).

3. **DIGITAL SIGNATURES**

Sender: M → MD → encrypt with private key → Digital sign.

4. **BLOCKCHAIN**

Transaction T → h(T) (Miner 1) → h(h(T)) (Miner 2) → h(h(h(T))) (Miner 3)

LOAD FACTOR $\frac{\text{Number of keys}}{\text{Table Size}} = \frac{n}{N}$

Open Hashing: n can be $> N$

Closed Hashing: n must be $<= N$

TIME COMPLEXITY OF HASHING

Best case complexity for search: → Open: $O(1)$, → Closed: $O(1)$

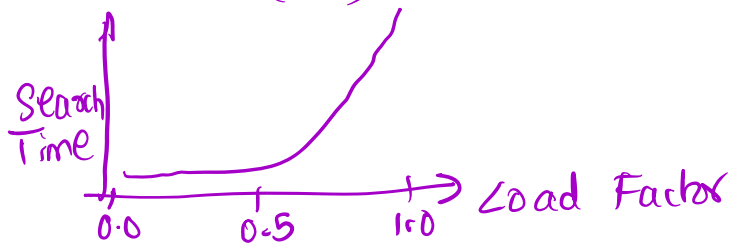
Worst case complexity for search: → Open: $O(n)$ → linked list, unbalanced tree; → Closed: $O(n)$

Average case complexity for search (depends upon the load factor):

Open: $O\left(\frac{n}{N}\right)$ in Linked list

$O\left(\log_2\left(\frac{n}{N}\right)\right)$ in Balanced binary search trees

Closed:



REHASHING

When more and more keys are inserted, the current table size may not suffice. At that time, a *rehashing* is done – a new table double its size is created, and all the keys are remapped to the new table.

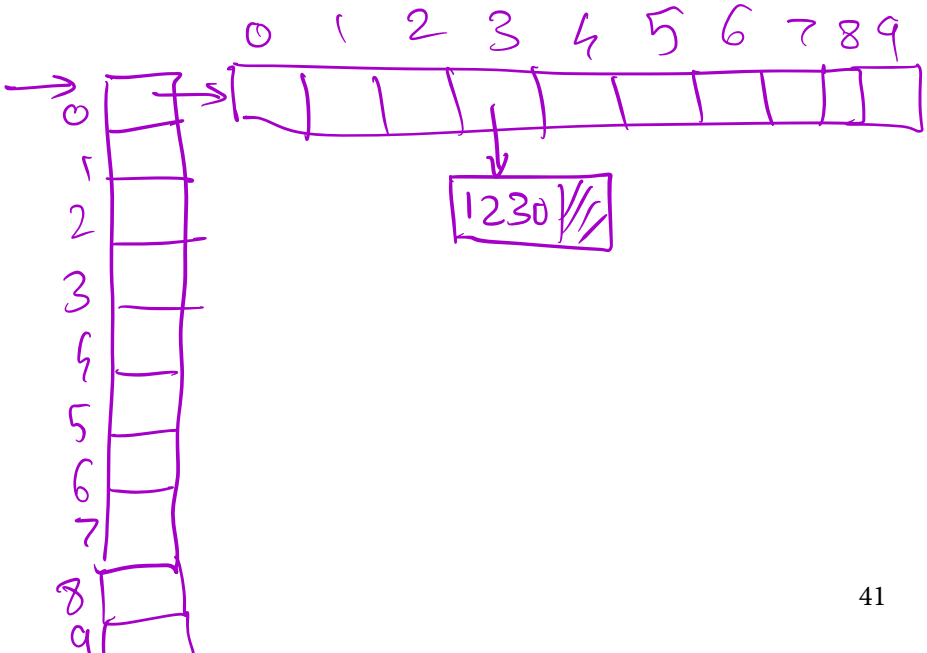
DOUBLE HASHING

The key is hashed twice to map it.

Key is 1230

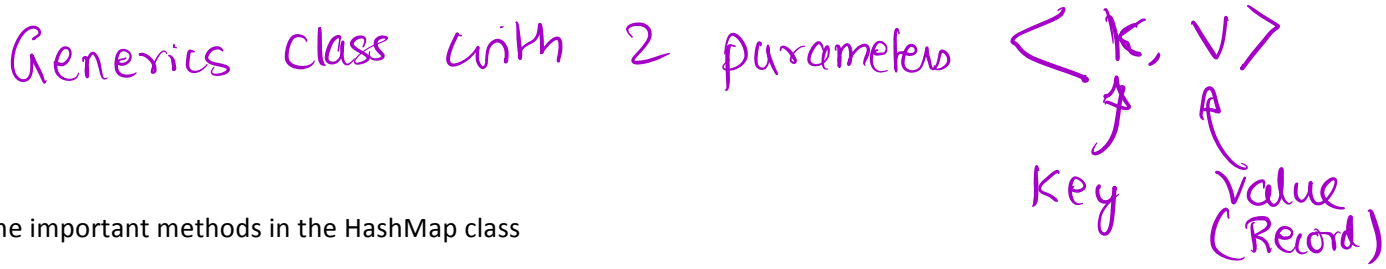
1230 % 10 → 0

123 % 10 → 3



HASH MAP IN JAVA

Java has a class called HashMap (java.util.HashMap) that lets you store and process keys and values associated with keys in a hash table.



Some important methods in the HashMap class

put(K key, V value)	Adds a key and value to the hash table
get (K key)	Returns the value associated with the key; returns null if the key has no value associated with it.
containsKey(K key)	Returns true if the key is found
values()	Returns a collection of values in the table
keySet()	Returns the set of keys in the table
remove(K key)	Deletes the entry with key K
isEmpty()	Returns true if table is empty

```
//Simple illustration of hashmap in java
//Creates a hashmap of student id numbers as keys and student names as records.
import java.util.HashMap;
import java.util.Scanner;
import java.io.*;

public class HashMapDemo
{
	public static void main(String[] args) throws IOException
	{
		HashMap<Integer, String> studentRecord = new HashMap<Integer,String>();
		Integer id;
		String name;
		Scanner keyboard = new Scanner(System.in);
		System.out.print("Enter the filename to read from: ");
		String filename = keyboard.nextLine();

		File file = new File(filename);
		Scanner inputFile = new Scanner(file);

		while (inputFile.hasNext())
		{
			id = Integer.parseInt(inputFile.next());
			name = inputFile.nextLine();
			studentRecord.put(id, name);
		}

		inputFile.close();
		System.out.println(studentRecord.values());
		System.out.println(studentRecord.keySet());

		System.out.print("Enter key: ");
		id = keyboard.nextInt();
		if (studentRecord.containsKey(id)){
			name = studentRecord.get(id);
			System.out.println(id + "\t" + name  + "  found");
		}
		else
			System.out.println(id + " not found");
	}
}
```

hash

Input file:

```
10245 James
23450 Jack
10398 Amar
10009 Boris
51430 Amy
69087 Brenda
88700 Zirui
67568 Xu
22229 Nick
17171 Chandra
```

Sample Run:

```
Enter the filename to read from: students.txt
[ Chandra,  Nick,  Xu,  Jack,  Boris,  Amar,  Amy,  James,  Brenda,  Zirui]
[17171, 22229, 67568, 23450, 10009, 10398, 51430, 10245, 69087, 88700]

Enter the search key: 10398
10398    Amar found

Process completed.
```

Two alternative classes: TreeMap and LinkedHashMap

TreeMap stores the records in a tree data structure and retrieves them in a sorted order. Default sorting is done on the keys.

```
import java.util.TreeMap;
…
TreeMap<Integer,String> studentRecord = new TreeMap<Integer, String>();
```

Sample run:
Enter the filename to read from: students.txt
[Boris, James, Amar, Chandra, Nick, Jack, Amy, Xu, Brenda, Zirui]
[10009, 10245, 10398, 17171, 22229, 23450, 51430, 67568, 69087, 88700]

Enter the search key: 23450
23450 Jack found

LinkedHashMap stores the records using a doubly linked list. Retrieval is in the order of insertion.

```
import java.util.LinkedHashMap;
…
LinkedHashMap<Integer,String> studentRecord = new LinkedHashMap<Integer, String>();
```

Sample run:
Enter the filename to read from: students.txt
[James, Jack, Amar, Boris, Amy, Brenda, Zirui, Xu, Nick, Chandra]
[10245, 23450, 10398, 10009, 51430, 69087, 88700, 67568, 22229, 17171]

Enter the search key: 10009
10009 Boris found

```
import java.util.*;

public class HashMapDemo4 {

    public static void main(String [] args) {


      Scanner keyboard = new Scanner(System.in);

      // create empty map
      HashMap<String,ArrayList<Integer>> data =
          new HashMap<String,ArrayList<Integer>>();

      // read words, one on each line, until "end"
      String word = keyboard.next();
      int lineNumber = 1;
      while (!word.equals("end")) {

       // if the word is not yet in map
            if (data.get(word) == null) {

            // create a new list and add it to the map
            ArrayList<Integer> lines = new ArrayList<Integer>();
            lines.add(lineNumber);
            data.put(word,lines);
          }
          else {

            // get the existing list for this word
            ArrayList<Integer> lines = data.get(word);

            // add the line number to that list
            lines.add(lineNumber);
          }

          // get the next word
          word = keyboard.next();
          lineNumber += 1;
      }

      // print the map
      for (String w : data.keySet()) {
          System.out.println(w + " - " + data.get(w));
      }

    }

}
```

key (word)

value

Java
C
C++
Java
Python
C++
Cobol
Java
end

Java [1, 4, 8]
C++ [3, 6]
C [2]
Cobol [7]
Python [5]