Searching Techniques

Objectives

At the end of the class, students are expected to be able to do the following:

- Understand the **searching technique concept** and the purpose of searching operation.
- Understand the implementation of basic searching algorithm;
  1. Sequential search.
     - Sequential search on unsorted data.
     - Sequential search on sorted data.
  2. Binary Search.
- Able to analyze the **efficiency** of the searching technique.
- Able to **implement** searching technique in problem solving.

Introduction

- **Searching Definition**
  - Clifford A. Shaffer[1997] define searching as a process to determine whether an element is a **member** of a certain data set.
  - The process of **finding the location** of an element with a specific value (key) within a collection of elements
  - The process can also be seen as an attempt to **search** for a certain record in a file.
    - Each record contains **data field** and **key field**
    - **Key field** is a group of characters or numbers used as an **identifier** for each record
    - Searching can done based on the key field.

Introduction to Search Algorithms

- Search: locate an item in a list of information

- Two algorithms we will examine:
  - Linear search
  - Binary search

Linear Search

- Also called the sequential search
- Starting at the first element, this algorithm sequentially steps through an array examining each element until it locates the value it is searching for.

Basic Sequential Search

- Basic sequential search usually is implemented to search item from **unsorted** list/ array.
- The technique can be implemented on a **small size** of list. This is because the **efficiency** of sequential search is **low** compared to other searching techniques.
- In a sequential search:
  1. Every element in the array will be **examine sequentially**, starting from the first element.
  2. The process will be **repeated** until the **last element** of the array or until the searched data is **found**.

- The **simplest search algorithm**, but is also the slowest
- Searching strategy:

1. **Examines** each element in the array one by one (sequentially) and compares its value with the one being looked for – the search key
2. Search is successful if the search key **matches** with the value being compared in the array. Searching process is **terminated**.
3. else, if no matches is found, the search process is **continued to the last** element of the array. Search is **failed** array if there is no matches found from the array.

Linear Search - Example

- Array numlist contains:

17	23	5	11	2	29	3

- Searching for the value 11, linear search examines 17, 23, 5, and 11
- Searching for the value 7, linear search examines 17, 23, 5, 11, 2, 29, and 3

Linear Search

- Algorithm:

set found to false; set position to –1; set index to 0
while index < number of elts. and found is false
    if list[index] is equal to search value
        found = true
        position = index
    end if
    add 1 to index
end while
return position

A Linear Search Method:

```
int searchList(int list[], int numElems, int value)
{
   int index = 0;       // Used as a subscript to search array
   int position = -1;   // To record position of search value
   bool found = false; // Flag to indicate if value was found

   while (index < numElems && found == false)
   {
      if (list[index] == value) // If the value is found
      {
         found = true; // Set the flag
         position = index; // Record the value's subscript
      }
      index++; // Go to the next element
   }
return position; // Return the position, or -1
}
```

```
int SequenceSearch( int    search_key,
                    const int array [ ],
                    int    array_size )
{    int p;
     int index =-1; //-1 means record is not found
     for ( p = 0; p < array_size; p++ ){
        if ( search_key == array[p] ){
            indeks = p;//assign current array index
            break;
        }//end if
     } //end for
    return index;
} //end function
```

Every element in the array will be examined until the search key is found

Or until the search process has reached the last element of the array

Sequential Search Analysis

- Searching time for sequential search is O(n).
- If the searched key is located at the end of the list or the key is not found, then the loop will be repeated based on the number of element in the list, O(n).
- If the list can be found at index 0, then searching time is, O(1).

Improvement of Basic Sequential Search Tech.

- **Problem:**
  - Search key is compared with all elements in the list, **O(n)** time consuming for large datasets.

- **Solution:**
  - The efficiency of basic search technique can be improved by searching on a **sorted list**.
  - For searching on ascending list, the search key will be compared one by one until :
    1. the searched key is **found**.
    2. Or until the searched **key value is smaller than the item compared** in the list.

=> This will minimize the searching process.

Linear Search – Sorted Input Example

- Array numlist3 contains:

2	3	5	11	17	23	29

- Searching for the value 11, linear search examines 2, 3, 5, and 11
- Searching for the value 7, linear search examines 2, 3, 5, 11 and declared that it is not found.

Sequential Searching on Sorted Data

```
int SortedSeqSearch ( int search_key, const int
array[ ],
                    int array_size)
{   int p;
    int index = -1; //-1 means record not found
    for ( p = 0; p < array_size; p++ )
    {  if (search_key < array [p] )
            break;
           // loop repetition terminated
           // when the value of search key is
           // smaller than the current array element
        else if (search_key == array[p])
        {
            index = p;  // assign current array index
            break;
        } // end else-if
    }//end for
    return index; // return the value of index
} //end function
```

Linear Search - Tradeoffs

- Benefits:
  - Easy algorithm to understand
  - Array can be in any order

- Disadvantages:
  - Inefficient (slow): for array of N elements, examines N/2 elements on average for value in array, N elements for value not in array

Binary Search :

Requires array elements to be in order

1. Divides the array into three sections:
   - middle element
   - elements on one side of the middle element
   - elements on the other side of the middle element
2. If the middle element is the correct value, done. Otherwise, go to step 1. using only the half of the array that may contain the correct value.
3. Continue steps 1. and 2. until either the value is found or there are no more elements to examine.

Binary Search - Example

- Array numlist2 contains:

2	3	5	11	17	23	29

- Searching for the value 11, binary search examines 11 and stops
- Searching for the value 7, linear search examines 11, 3, 5, and stops.

Binary Search

Set first index to 0.
Set last index to the last subscript in the array.
Set found to false.
Set position to -1.
While found is not true and first is less than or equal to last
    Set middle to the subscript half-way between array[first] and array[last].
    If array[middle] equals the desired value
        Set found to true.
        Set position to middle.
    Else If array[middle] is greater than the desired value
        Set last to middle - 1.
    Else
        Set first to middle + 1.
    End If.
End While.
Return position.

A Binary Search Function

```
int binarySearch(int array[], int size, int value)
{
   int first = 0,                // First array element
       last = size - 1,          // Last array element
       middle,                   // Mid point of search
       position = -1;            // Position of search value
   bool found = false;           // Flag

   while (found == false && first <= last)
   {
      middle = (first + last) / 2;     // Calculate mid point
      if (array[middle] == value)      // If value is found at mid
      {
         found = true;
         position = middle;
      }
      else if (array[middle] > value)  // If value is in lower half
         last = middle - 1;
      else
         first = middle + 1;           // If value is in upper half
   }
   return position;
}
```

Binary Search - Tradeoffs

- Benefits:
  - Much more efficient than linear search. For array of N elements, performs at most $\boldsymbol{log_2 N}$ comparisons

- Disadvantages:
  - Requires that array elements be sorted

Hashing:

- **Dictionaries :**
- Dictionaries stores elements so that they can be located quickly using **keys.**
- **Dictionary** = data structure that supports mainly two basic operations: insert a new item and return an item with a given key
- **For eg :** A Dictionary may hold bank accounts. In which key will be account number. And each account may stores many additional information.

How to Implement a Dictionary?

- Different data structure to realize a key
  - Array , Linked list
  - Binary tree
  - **Hash table**
  - Red/Black tree
  - AVL Tree

Why Hashing?

- The sequential search algorithm takes time proportional to the data size, i.e, **O(n)**.
- Binary search improves on liner search reducing the search time to **O(log n)**.
- With a BST, an **O(log n)** search efficiency can be obtained; but the worst-case complexity is **O(n)**.
- To guarantee the **O(log n)** search time, BST height balancing is required (i.e., AVL trees).

Why Hashing?

- Suppose that we want to store 10,000 students records (each with a 5-digit ID) in a given container.
- A linked list implementation would take **O(n)** time.
- A height balanced tree would give **O(log n)** access time.
- Using an array of size 10,000 would give **O(1)** access time but will lead to a lot of space wastage.
- Is there some way that we could get **O(1)** access without wasting a lot of space?
- The answer is **hashing.**

Hashing :

- Another important and widely useful technique for implementing dictionaries.
- Constant time per operation (on the average) Like an array, come up with a function to map the large range into one which we can manage.

Applications:

- **Keeping track of customer account information at a bank**
  - Search through records to check balances and perform transactions
- **Keep track of reservations on flights**
  - Search to find empty seats, cancel/modify reservations
- **Search engine**
  - Looks for all documents containing a given word

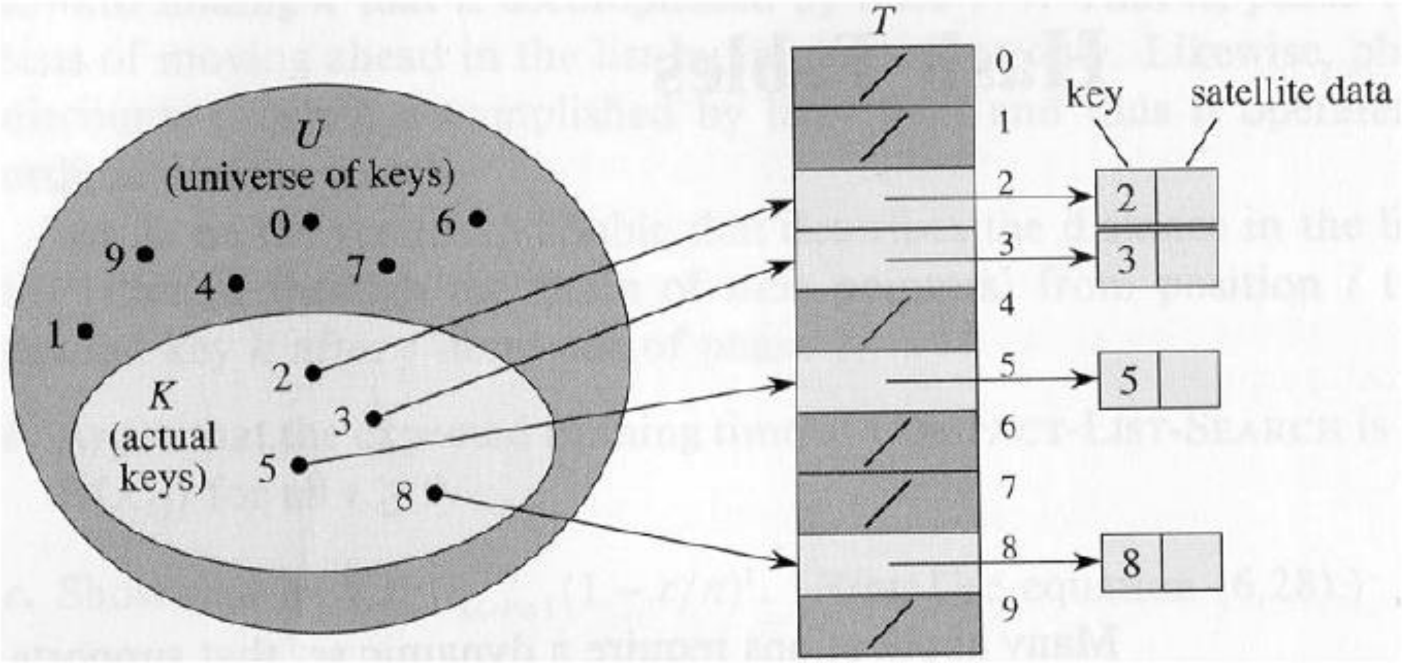
Direct Addressing

- Assumptions:
  - Key values are distinct
  - Each key is drawn from a universe U = {0, 1, . . . , m - 1}
- Idea:
  - Store the items in an array, indexed by keys

- **Direct-address table** representation:

– An array T[0 . . . m - 1]
– Each **slot**, or position, in T corresponds to a key in U
– For an element x with key k, a pointer to x (or x itself) will be placed in location T[k]
– If there are no elements with key k in the set, T[k] is empty, represented by NIL

Direct Addressing (cont'd)



(insert/delete in O(1) time)

Examples Using Direct Addressing

Example 1:

(i) Suppose that the keys are integers from 1 to 100 and that there are about 100 records

(ii) Create an array A of 100 items and store the record whose key is equal to i in $A[i]$

Example 2:

(i) Suppose that the keys are nine-digit social security numbers

(ii) We can use the same strategy as before but it very inefficient now: an array of 1 billion items is needed to store 100 records !!

- $|U|$ can be very large
- $|K|$ can be much smaller than $|U|$

Hash Tables

- When K is much smaller than U, a **hash table** requires much less space than a **direct-address table**
  - Can reduce storage requirements to $|K|$
  - Can still get $O(1)$ search time, but on the average case, not the worst case

Hash Tables

Idea:

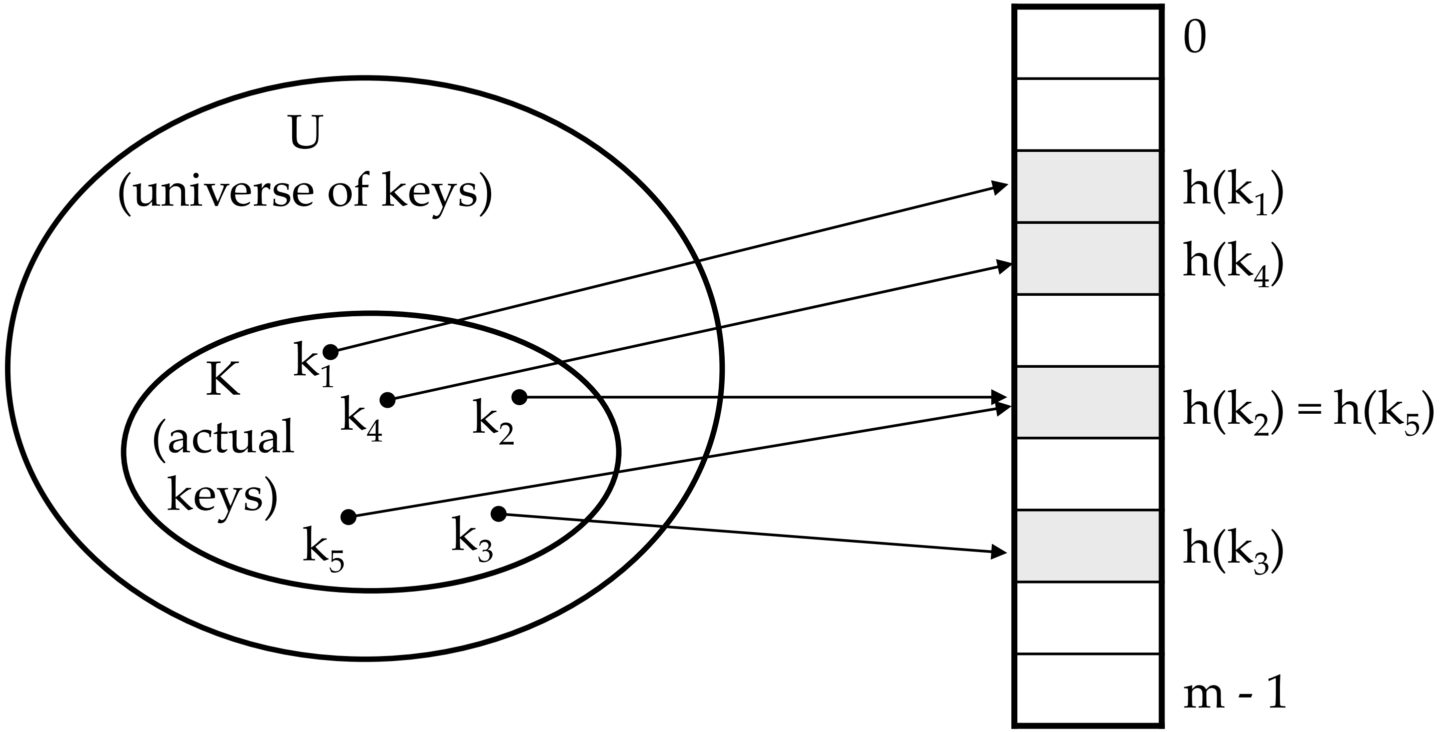
- Use a function h to compute the slot for each key
- Store the element in slot $h(k)$

- A **hash function** h transforms a key into an index in a hash table $T[0...m-1]$:

$$h : U \rightarrow \{0, 1, \ldots, m - 1\}$$

- We say that k **hashes** to slot $h(k)$
- Advantages:
  - Reduce the range of array indices handled: m instead of $|U|$
  - Storage is also reduced

Example: HASH TABLES



Revisit Example 2

Suppose that the keys are nine-digit social security numbers

Possible hash function

$$h(ssn) = sss \bmod 100 \text{ (last 2 digits of ssn)}$$

e.g., if $ssn = 10123411$ then $h(10123411) = 11)$

Hash Functions:

- **Division Method:**
  - H(k) = k (mod m) or H(k) = k (mod m) + 1
- **Midsquare Method:**
  - H(k) = l
- **Folding Method:**
  - H(k) = k1 + k2 + … +Kr

The Division Method :

- **Idea:**
  - Map a key k into one of the m slots by taking the remainder of k divided by m

$$h(k) = k \bmod m$$

- **Advantage**:
  - fast, requires only one operation
- **Disadvantage**:
  - Certain values of m are bad, e.g.,
    - power of 2
    - non-prime numbers

Example - The Division Method

- If $m = 2^p$, then $h(k)$ is just the least significant p bits of k
  - $p = 1 \Rightarrow m = 2$

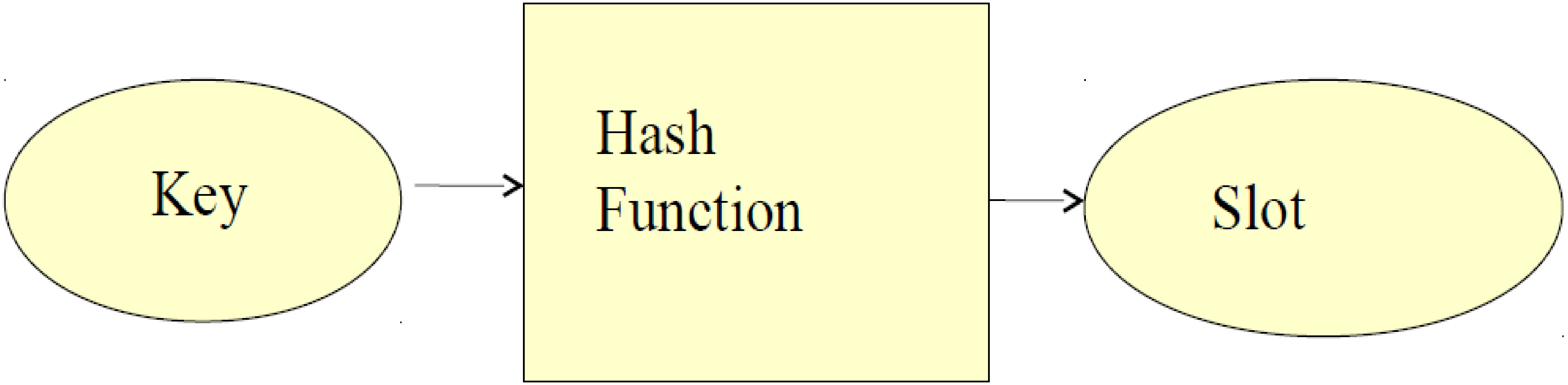
    $\Rightarrow h(k) = \{0, 1\}$, least significant 1 bit of k
  - $p = 2 \Rightarrow m = 4$

    $\Rightarrow h(k) = \{0, 1, 2, 3\}$ least significant 2 bits of k
- Choose m to be a prime, not close to a power of 2
  - Column 2: k mod 97
  - Column 3: k mod 100

	m 97	m 100
16838	57	38
5758	35	58
10113	25	13
17515	55	15
31051	11	51
5627	1	27
23010	21	10
7419	47	19
16212	13	12
4086	12	86
2749	33	49
12767	60	67
9084	63	84
12060	32	60
32225	21	25
17543	83	43
25089	63	89
21183	37	83
25137	14	37
25566	55	66
26966	0	66
4978	31	78
20495	28	95
10311	29	11
11367	18	67

Hash Functions:

- A Good Hash function is one which distribute keys evenly among the slots.
- And It is said that Hash Function is more art than a science. Because it need to analyze the data.



Hash Functions:

- **Need of choose a good Hash function**
  - Quick Compute.
  - Distributes keys in uniform manner throughout the table.
- **How to deal with Hashing non integer Key???**
  - Find some way of turning keys into integer.
  - For Example: if key is in character then convert it into integer using ASCII
  - Then use standard Hash Function on the integer.

Collisions

- Two or more keys hash to the same slot!!
- For a given set K of keys
  - If $|K| \leq m$, collisions may or may not happen, depending on the hash function
  - If $|K| > m$, collisions will definitely happen (i.e., there must be at least two keys that have the same hash value)
- Avoiding collisions completely is hard, even with a good hash function

Handling Collisions

- We will review the following methods:
  - Chaining
  - Open addressing
    - Linear probing
    - Quadratic probing
    - Double hashing
- We will discuss chaining first, and ways to build “good” functions.

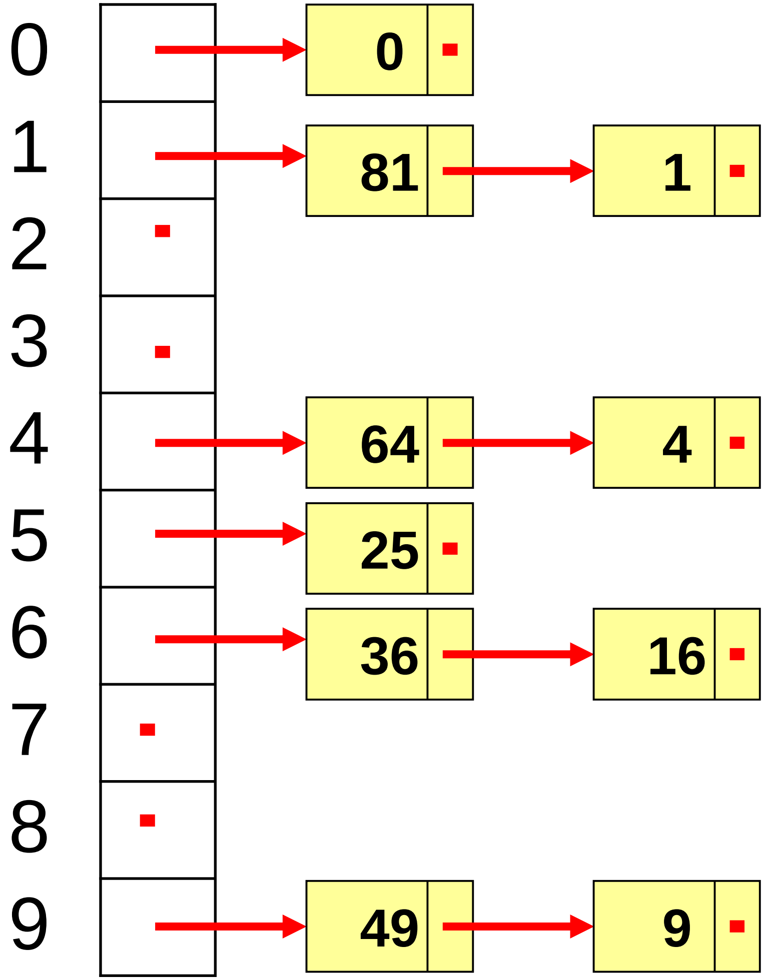
Collision Resolution Schemes: Chaining

The hash table is an array of linked lists

Insert Keys: 0, 1, 4, 9, 16, 25, 36, 49, 64, 81

Notes:

- As before, elements would be associated with the keys
- We're using the hash function h(k) = k mod m



Linear Probing

Function f is linear. Typically, $f(i) = i$

So, $h(k, i) = (h'(k) + i) \bmod m$

Offsets: 0, 1, 2, …, m-1

With $H = h'(k)$, we try the following cells with wraparound:

H, $H + 1$, $H + 2$, $H + 3$, …

What does the table look like after the following insertions?

Insert Keys: 0, 1, 4, 9, 16, 25, 36, 49, 64, 81

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Linear Probing :

0	0
1	1
2	49
3	
4	4
5	25
6	16
7	36
8	64
9	9

Practice Problem:

- Given input {4371, 1323, 6173, 4199, 4344, 9679, 1989} and a hash function $h(x) = x \mod 10$, show the resulting
  - separate chaining hash table
  - hash table using linear probing

Practice Problem:

- Given input {4371, 1323, 6173, 4199, 4344, 9679, 1989} and a hash function $h(x) = x \mod 10$, show the resulting
  - separate chaining hash table
  - hash table using linear probing

0	9679
1	4371
2	1989
3	1323
4	6173
5	4344
6	
7	
8	
9	4199