

Efficient Algorithms

Ekkapot Charoenwanit

Software Systems Engineering

TGGS

KMUTNB

Lecture 4: Data Structures (Part II)

Hashing

Direct-Addressing

Direct addressing is used to implement a dynamic set T denoted by $T[0 \dots m - 1]$ consisting of m slots and $T[i]$ denotes a **key value** stored at the i^{th} slot of the set T .

- Each slot stores **at most one element**
- Direct addressing works well when the universe $U = \{0, 1, 2, \dots, m - 1\}$ of keys is **small**, that is, when m is **small**
- For an empty slot i , $T[i] = NULL$

Direct-Address Table

Implementation:

- Each slot stores a pointer to the actual object.
- Objects consist of two parts: *key* and *data*.

Implementation Alternative:

- Instead of storing pointers, we can store objects in T to save space.

Example :

- It is given that the universe $U = \{0, 1, 2, \dots, 9\}$.
- Currently, the direct-address table T has 4 elements.
- The keys of the current elements in T are from U , namely, 2, 3, 5 and 8.

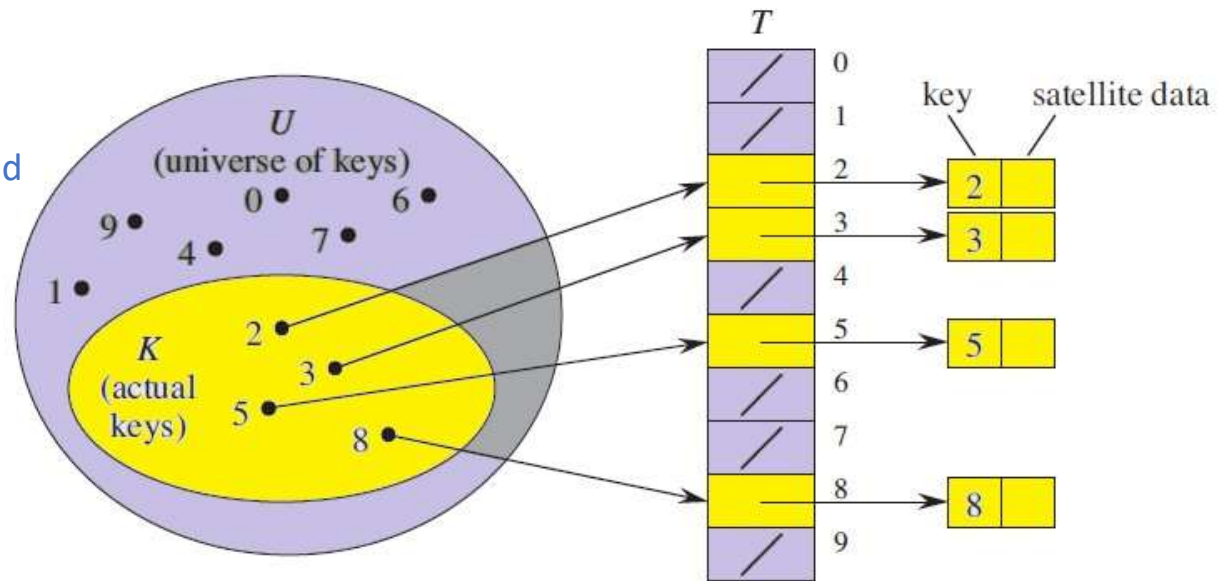


Illustration from the CLRS book
(Figure 11.1)

Basic Operations

```
1: procedure DIRECT-ADDRESS-SEARCH( $T, k$ )  
2:   return  $T[k]$ 
```

```
1: procedure DIRECT-ADDRESS-INSERT( $T, x$ )  
2:   return  $T[x.key] = x$ 
```

```
1: procedure DIRECT-ADDRESS-DELETE( $T, x$ )  
2:   return  $T[x.key] = \text{NULL}$ 
```

All the three basic operations of the direct-address table are $O(1)$ operations.

Downsides of Direct Addressing

Observations:

- When the universe U is large, storing a table T of size $|U|$ using direct addressing is impractical.
- The set K of keys **actually stored** may be so small relative to the size of the universe U that most of the space would be wasted.

To fix these issues,
we use **hash tables**.

Hash Table

When the set K of keys stored is much less than the universe U of all possible keys, a **hash table** requires **much less storage** than a **direct-address table**.

A hash table requires storage of $\Theta(|K|)$ we maintain the benefit that searching for an element still requires $O(1)$ in the **average case**.

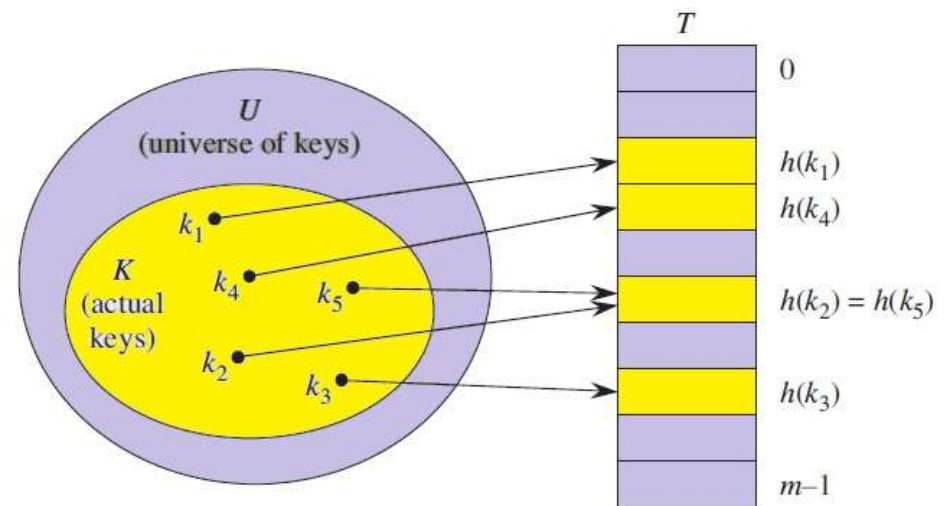


Illustration from the CLRS book
(Figure 11.2)

Hash Function

- With **direct-addressing**,
an element with key k is stored in slot k .
- With **hashing**,
an element with key k is stored in slot $h(k)$, where $h: U \mapsto \{0, 1, 2, \dots, m - 1\}$ is a
called a **hash function**.

Terminology:

Key k hashes to slot $h(k)$.

$h(k)$ is the hash value of key k .

Hashing Collision

When two different keys hash to the same slot, we call this situation a **collision**.

- k_2, k_5, k_7 hash to the same slot because $h(k_2) = h(k_5) = h(k_7)$.

The solution is to avoid collisions or at least minimize their number.

- Choosing a good hash function h is key to minimizing collisions
- h should **appear random** but must be **deterministic** in that given key k the hash function h must always produce the same output $h(k)$.

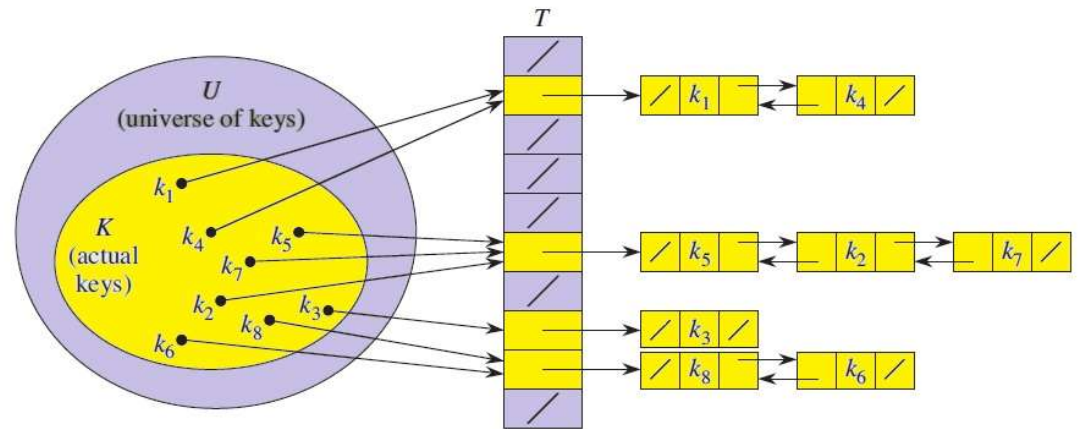


Illustration from the CLRS book (Figure 11.3)

Collision Resolution

Because the universe is larger than the hash table size ($|U| > m$),
there must be at least two different keys k_i and k_j that hash to the same value $h(k_i) = h(k_j)$ by the **Pigeonhole Principle**.

*******This means avoiding collisions altogether is impossible !!!

We will talk about **two approaches** to resolving collisions:

- Separate Chaining
- Open Addressing

Separate Chaining

In separate chaining,

- elements that hash to the same slot are placed into the same **linked list**
- slot j **stores a pointer** to the head of the linked list of all stored elements that hash to j
- For an empty slot j , $T(j) = \text{NULL}$

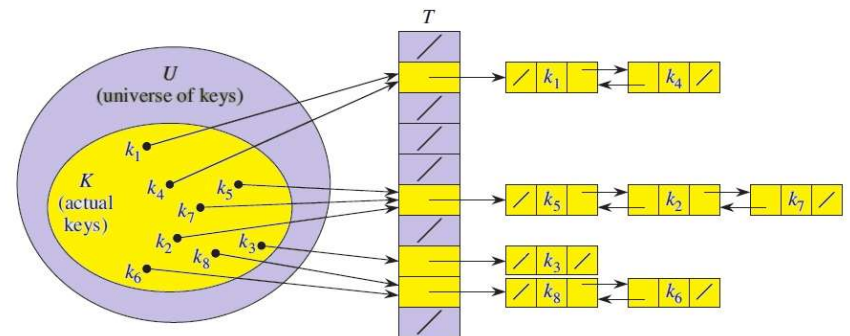


Illustration from the CLRS book (Figure 11.3)

Basic Operation: Insert

- A new element x with key $x.key$ is always inserted **at the head of the linked list** of slot $h(x.key)$.

Assuming x is **not present**,

- The worst-case running time for insertion is $O(1)$.
 - Insertion costs $O(1)$ time because it involves updating
 - **two pointers** for an implementation using **singly-linked lists** $\Rightarrow O(1)$
 - **three pointers** for an implementation using **doubly-linked lists** $\Rightarrow O(1)$

Assuming x is **present**, we search for x first before we insert.

```
1: procedure CHAINED-HASH-INSERT( $T, x$ )
2:   insert  $x$  at the head of  $T[h(x.key)]$ 
```

Basic Operation: Search

- The worst-case running time for insert is *proportional to the length of the list*.
- We will do analysis on the average cost of this operation in detail.

```
1: procedure CHAINED-HASH-SEARCH( $T, k$ )  
2:   search for an element with key  $k$  in  $T[h(k)]$ 
```

Basic Operation: Delete

- The worst-case running time of deleting an element is $O(1)$
 - for an implementation using *doubly-linked lists*

```
1: procedure CHAINED-HASH-DELETE( $T, x$ )  
2:   delete  $x$  from  $T[h(x.key)]$ 
```

Basic Operation: Delete by Key

To delete an element x by key k , we can make use of the search and the delete operation previously discussed as follows:

```
1: procedure CHAINED-HASH-DELETE-BY-KEY( $T, k$ )
2:    $x = \text{CHAINED-HASH-SEARCH}(T, k)$ 
3:   if  $x \neq \text{NULL}$  then
4:     CHAINED-HASH-DELETE( $T, T[h(x.key)]$ )
```

Load Factor

Given a hash table T with m slots that stores n elements, we define the **load factor** denoted by α for T as $\frac{n}{m}$.

In other words, the load factor α is the **average number of elements** stored in a chain.

Our probabilistic analysis will be in terms of the load factor α , which can be less than, equal to or greater than **1**.

Load Factor

The load factor α measures how full a hash table is.

- the load factor $\alpha=0 \Rightarrow$ the hash table is *empty*.
- the load factor $\alpha=1 \Rightarrow$ the hash table is *full*.

In chaining, the table size is the number of linked lists.

- α is the average length of the linked lists

Average-Case Analysis

The worst-case behavior of hashing is still terrible

- The key of all n elements hash to **the same slot**, resulting in a long chain of length n .
- Therefore, the worst-case time is $\Theta(n)$ plus the time to compute the hash value.

Clearly, we **do not** use **hash tables** for their **worst-case performance** !!!

We depend on how well the hash function h distributes the set of keys U to be stored among the m slots **in the average case**.

Simple Uniform Hashing

Our probabilistic analysis is based on the assumption of *Simple Uniform Hashing*.

Simple Uniform Hashing

- Any given element is equally likely hash to any of the m slots, independently of where any other element has hashed to.

Simple Uniform Hashing

Let n_i denote the length of the list $T[i]$,

so that $n = n_0 + n_1 + \cdots + n_{m-1}$ and the expected value of n_i is $E(n_i) = \alpha = \frac{n}{m}$.

Assume that the hash function runs in constant time $\Theta(1)$.

The time required to **search for an element** with key k is **linearly proportional to the length** $n_{h(k)}$ of $T[h(k)]$.

Determining the average complexity of the search operation boils down to finding the expected number of elements examined in $T[h(k)]$ by the search operation to see whether any element has a key whose value equal to the given key k .

The average-case complexity of search

We shall consider *two cases* as follows:

Case I: an unsuccessful case

Theorem: In a hash table where collisions are resolved by chaining, a successful search takes average-case time of $\Theta(1 + \alpha)$ under the assumption of simple uniform hashing.

Case II: a successful case

Theorem: In a hash table where collisions are resolved by chaining, a successful search takes average-case time of $\Theta(1 + \alpha)$ under the assumption of simple uniform hashing.

Unsuccessful Search

Theorem: In a hash table where collisions are resolved by chaining, an unsuccessful search takes average-case time of $\Theta(1 + \alpha)$ under the assumption of simple uniform hashing.

Proof: Under the assumption of simple uniform hashing, any key k not already stored in the hash table is equally likely to hash to any of the m slots.

The expected time to unsuccessfully search for an element with key k is the expected time to search to the end of the list $T[h(k)]$, which is proportional to the expected length $E(n_{h(k)}) = \alpha$.

Therefore, the average-case time is $\Theta(1) + \Theta(\alpha) = \Theta(1 + \alpha)$, where $\Theta(1)$ is the time for the hash function. ■

Successful Search

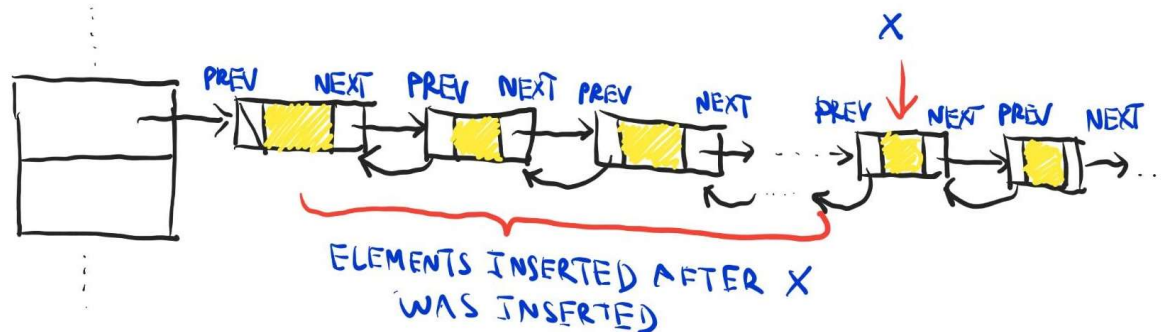
Theorem: In a hash table where collisions are resolved by chaining, a successful search takes average-case time of $\Theta(1 + \alpha)$ under the assumption of simple uniform hashing.

Proof: Under the assumption of simple uniform hashing, we assume that the element x being searched for is equally likely to be any of the n elements stored in the table.

Successful Search

Key Observation I: The number of elements examined during a successful search for x is one more than the number of elements that appear before x in the chain.

Key Observation II: New elements are placed at the front of the chain \Rightarrow elements before x in the chain were all inserted after x was inserted



Successful Search

Let x_i denote the i^{th} element inserted into the table, for $i = 1, 2, \dots, n$ and let $k_i = x_i$.key.

For keys k_i and k_j , we define the **indicator random variable** $X_{i,j} = I\{h(k_i) = h(k_j)\}$, which means

$$X_{i,j} = \begin{cases} 0 & \text{if } h(k_i) \neq h(k_j) \\ 1 & \text{if } h(k_i) = h(k_j) \end{cases}$$

Under the assumption of simple uniform hashing, $\Pr\{h(k_i) = h(k_j)\} = \frac{1}{m}$.

Therefore, $E(X_{i,j}) = 1(\frac{1}{m}) + 0(\frac{m-1}{m}) = \frac{1}{m}$.

To find the expected number of elements, we take the average, over the n elements in the table, of one plus the expected number of elements inserted to x 's list after x was inserted to the list.

Successful Search

The expected number of examined elements in a *successful search* is

$$\begin{aligned} E\left(\frac{1}{n}\sum_{i=1}^n(1 + \sum_{j=i+1}^n X_{i,j})\right) &= \frac{1}{n}\sum_{i=1}^n\left(1 + \sum_{j=i+1}^n E(X_{i,j})\right) \\ &= \frac{1}{n}\sum_{i=1}^n\left(1 + \sum_{j=i+1}^n \frac{1}{m}\right) \\ &= 1 + \frac{1}{mn}\sum_{i=1}^n(n-i) \\ &= 1 + \frac{1}{mn}\sum_{i=1}^n n - \sum_{i=1}^n i \\ &= 1 + \frac{n-1}{2m} \\ &= 1 + \frac{\alpha}{2} - \frac{\alpha}{2n} = \Theta(1 + \alpha) \end{aligned}$$

Successful Search

The expected number of examined elements in a *successful search* is $\Theta(1 + \alpha)$.

Therefore, the total average running time (after taking into account the time $\Theta(1)$ required to compute the hash function) is

$$\begin{aligned} & \Theta(1) + \Theta(1 + \alpha) \\ &= \Theta(2 + \alpha) \\ &= \Theta(1 + \alpha). \blacksquare \end{aligned}$$

Constant-Time Operations

If the load factor α is bound by some constant,

all the three basic operations run in $O(1)$ time

- *Search*
- *Insert*
- *Delete*

Hash Function

A *good hash function* should (*approximately*) satisfy the assumption of simple uniform hashing: each key is equally likely to hash to any of the m slots, independently of where any other element has hashed to.

- Unfortunately, we typically have no way to check this condition, since we rarely know the probability distribution from which the keys are drawn.
- Moreover, the keys may not be drawn independently.

Heuristically Good Hash Functions

In practice, we can often employ *heuristics* to create a hash function that performs well.

We will discuss *two variants* of *heuristics* for *creating good hash functions*:

- Division Method
- Multiplication Method

Interpreting keys as natural numbers

Most hash functions assume that the universe of keys is the set of **natural numbers** $\mathbb{N} = \{0, 1, 2, \dots\}$.

From now, we will assume keys are natural numbers.

If they are not, we can often find a way to treat them as natural numbers somehow.

- We can interpret a **string of characters** as an integer expressed in suitable **radix notation**.
- For example, we can convert the string “**Hash**” to $72 \cdot 128^3 + 97 \cdot 128^2 + 115 \cdot 128^1 + 104 = 152599016$ using a **radix-128 integer**.
- ASCII Code: $H = 72, a = 97, s = 115$ and $s = 104$

Division Method

In the **division method**, we map a key k into one of the m slots by taking the remainder of k divided by m .

The hash function is $h_m(k) = k \bmod m$.

m should *****not***** be a power of two since if $m = 2^p$, the hash value $k \bmod 2^p$ will be the lowest p order bits.

Rule of Thumb:

m should be a prime that is *not too close* to an *exact power of two*.

Multiplication Method

1. Multiply the key k by some number $0 < A < 1$
2. Extract the fractional part of kA from **STEP (1)**
3. Multiply it the fractional part from **STEP (2)** by m
4. Take the floor of the result of **STEP (3)**

In other words, $h_{m,A}(k) = \lfloor m \cdot (kA - \lfloor kA \rfloor) \rfloor$

- Although this method works with any value of A , it works better with some values than with others.
- The optimal choice depends on the characteristics of the data being hashed.
- Knuth suggests that $A = \frac{\sqrt{5}-1}{2} \approx 0.6180339887$ works pretty well.

Open Addressing

In *open addressing*,

each slot stores at most one element, that is, each table slot either *contains an element* or *is empty*.

With the notion of *probing*,

- each key does not need to always get mapped to a single slot.
- in a collision, we perform collision resolution by *successively examining in a systematic way* the hash table until we eventually find an *empty* slot, into which the new element is inserted.
- such a systematic way of examining the hash table is called *probing*.

Probe Sequence

We extend the hash function to include the *probe number* as a second input:

$$h: U \times \{0, 1, 2, \dots, m-1\} \mapsto \{0, 1, 2, \dots, m-1\}$$

With open addressing,

- we require that for every key k , the probe sequence $\langle h(k, 0), h(k, 1), \dots, h(k, m-1) \rangle$ be a permutation of $\langle 0, 1, 2, \dots, m-1 \rangle$.
- This behavior of the hash function ensures that all m slots will be *eventually* probed in the worst case (i.e. *when the hash table is full*).

Basic Operation: Insert

For the purpose of simplicity,

we assume that keys k_i and elements x_i have the same value, i.e., $k_i = x_i.key$.

Each slot contains either a value or a **NULL** (if the slot is empty.)

It either returns a **slot number** or it returns **-1** to signify that the table is already full.

```
1: procedure OPEN-ADDRESSING-HASH-INSERT( $T, k$ )
2:    $i = 0$ 
3:   while  $i \leq m$  do
4:      $j = h(k, i)$ 
5:     if  $T[j] = \text{NULL}$  then
6:        $T[j] = k$ 
7:       return  $j$ 
8:     else
9:        $i = i + 1$ 
10:  return -1
```

Basic Operation: Search

Successful Search:

The search will return the element being searched for if the element is stored in the hash table.

Unsuccessful Search:

There are two possibilities for an unsuccessful search:

- an empty slot is encountered
- the end of the hash table is reached

```
1: procedure OPEN-ADDRESSING-HASH-SEARCH( $T, k$ )
2:    $i = 0$ 
3:   while  $i \leq m \vee T[j] = \text{NULL}$  do
4:      $j = h(k, i)$ 
5:     if  $T[j] = k$  then
6:       return  $j$ 
7:      $i = i + 1$ 
8:   return NULL
```

Observation:

The algorithm for searching for key k probes the same sequence of slots as the insertion algorithm examined when key k was inserted.

If the algorithm finds an empty slot mid-way, it means key k is not present in the table. Otherwise, key k would have been inserted in this empty slot and not later in its probe sequence.

Basic Operation: Deletion

In open addressing, deletion is not as straightforward.

When we delete key k from its slot, we cannot simply mark it as empty by storing **NULL**.

We solve this problem by storing a special flag **DELETED** instead of **NULL**. (PS 4.3.1 , 4.3.2)

Probing Techniques

We will show three **probing techniques** that can be used to produce ***probe sequences***:

- Linear Probing
- Quadratic Probing
- Double Hashing

Linear Probing

In **linear probing**, when a collision occurs,

- we move forward by **one position** (wrapping around when reaching the last slot) to see if it is an empty slot.
- we continue moving forward by **one position** until an empty slot is found.
- otherwise, it means the hash table is full.

Hash functions for linear probing are of the form:

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

where $h': U \mapsto \{0, 1, 2, \dots, m - 1\}$ is an **auxiliary hash function** and $i = 0, 1, 2, \dots, m - 1$.

The initial position probed is $T[h'(k)]$; later positions probed will be offset by i (wrapping around for the last slot).

Quadratic Probing

In **quadratic probing**, we use a hash function of the form:

$$h(k, i) = (h'(k) + c_1i + c_2i^2) \bmod m$$

where $h'(k)$ is an **auxiliary hash function** and c_1, c_2 are positive constants and $i = 0, 1, \dots, m - 1$.

The initial position probed is $T[h'(k)]$; later positions probed are offset by some amount that depends on the probe number i and the two constants c_1, c_2 .

Double Hashing

In **double hashing**, we use a hash function of the form:

$$h(k, i) = (h_1(k) + ih_2(k)) \bmod m$$

where both h_1 and h_2 are auxiliary hash functions.

With **double hashing**,

- a larger number of probe sequences are made possible
- the probe sequence depends on the key k in two ways (the initial probe position or the offset or both may vary)
- to allow the entire hash table to be searched, h_2 and m must be chosen in such a way that they are **relatively prime**.

Double hashing makes probe sequences **look more random** than linear and quadratic probing so it performs better.

Summary

We have learned the following topics:

- *Direct-Address Table*
- *Hash Table*
- *Collision Resolution*
 - *Separate Chain Method*
 - *Open Address Method*

Next time, we will cover *sorting algorithms*.