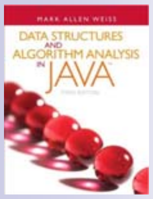
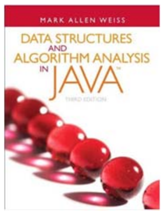


FIU | FLORIDA INTERNATIONAL UNIVERSITY

Hashing (I)

Dr. Antonio L. Bajuelos

FIU | School of Computing & Information Sciences



COP-3530 - Data Structures

Module #5: Hashing (part I)

Outline:

- **The hash table ADT**
- **Hashing - main ideas:**
  - **hash table and hash function**
  - **hash functions for string keys**
- **The collision resolution:**

  (I) Separate chaining
  - **Complexity analysis and Java code**

Remember that…

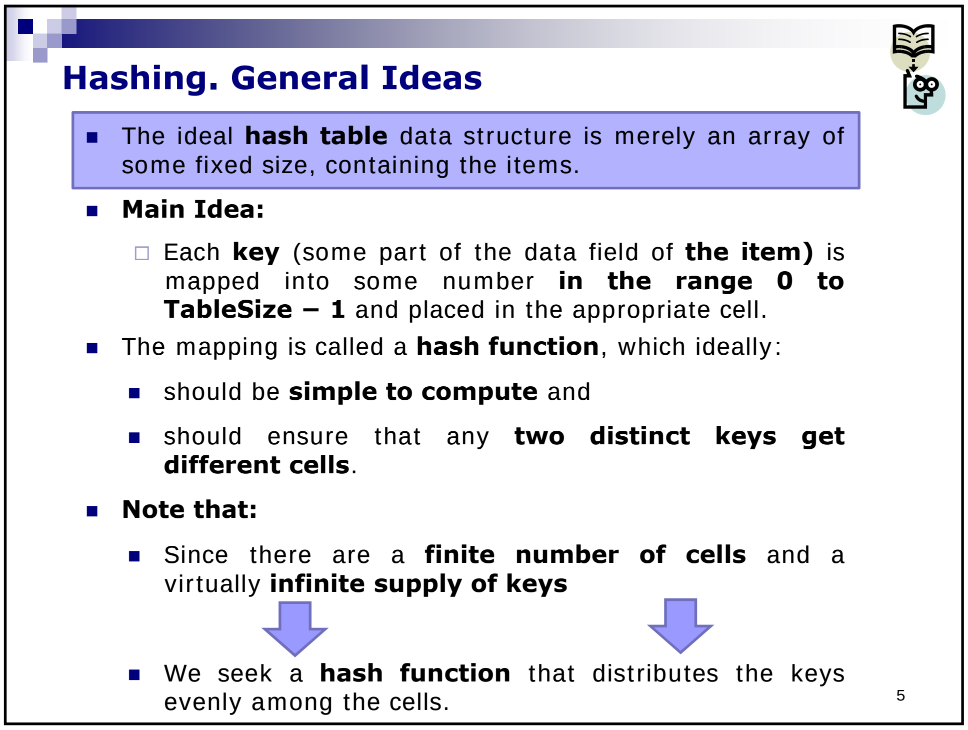
- For a collection with N keys/values we have:

	insert	**find**	**delete**
Unsorted linked-list	O(1)	O(N)	O(N)
Unsorted array	O(1)	O(N)	O(N)
Sorted linked list	O(N)	O(N)	O(N)
Sorted array	O(N)	O(logN)	O(N)
AVL-tree	O(logN)	O(logN)	O(logN)*
Ideal ADT	**O(1)**	**O(1)**	**O(1) ???**

*Lazy deletion

The Hash Table ADT

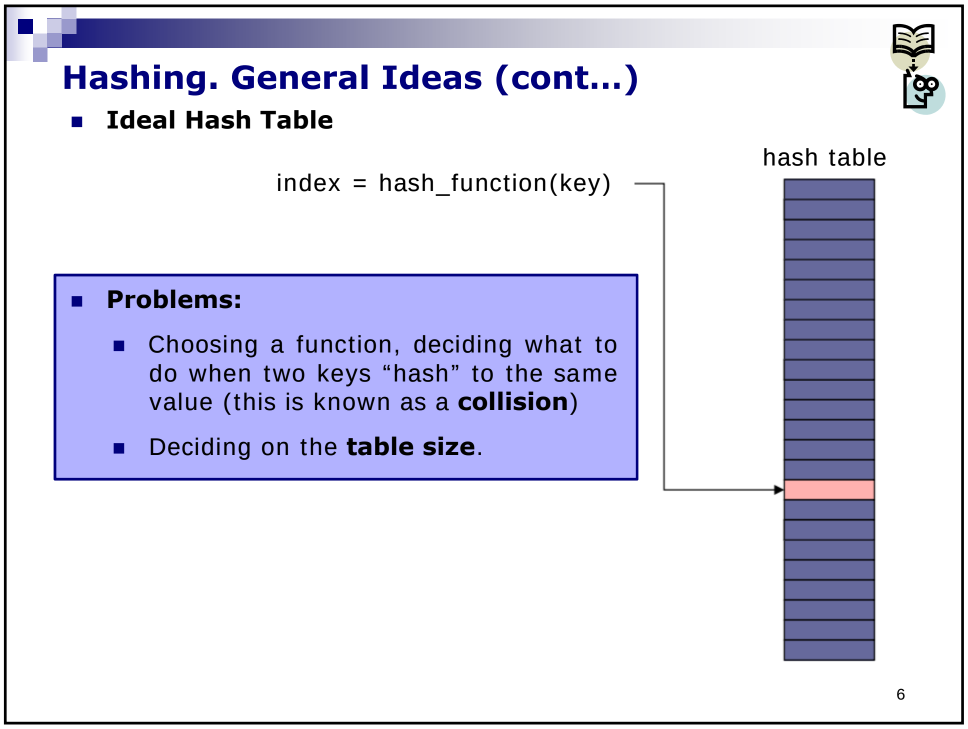
- In this module we discuss the **hash table ADT**, which supports only a subset of the operations allowed by binary search trees.
- The implementation of **hash tables** is frequently called **hashing**.
- **Hashing** is a technique used for performing insertions, deletions, and searches in constant average time.
- But!, tree operations that require any ordering information among the elements are not supported efficiently. Thus, operations such as **findMin** and **findMax** are not supported.
- The ideal **hash table** data structure is merely an array of some fixed size, containing the items.



Hashing. General Ideas

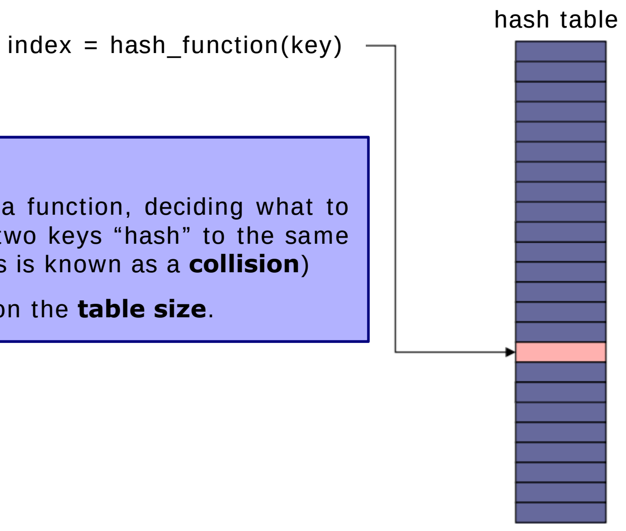
- The ideal **hash table** data structure is merely an array of some fixed size, containing the items.
- **Main Idea:**
  - Each **key** (some part of the data field of **the item)** is mapped into some number **in the range 0 to TableSize − 1** and placed in the appropriate cell.
- The mapping is called a **hash function**, which ideally:
  - should be **simple to compute** and
  - should ensure that any **two distinct keys get different cells**.
- **Note that:**
  - Since there are a **finite number of cells** and a virtually **infinite supply of keys**
  - We seek a **hash function** that distributes the keys evenly among the cells.

5



Hashing. General Ideas (cont...)

- **Ideal Hash Table**



- **Problems:**
  - Choosing a function, deciding what to do when two keys “hash” to the same value (this is known as a **collision**)
  - Deciding on the **table size**.

6

Hashing. General Ideas (cont…)

- **Some considerations:**
  - If the input keys are integers, then the most simply **hash function**:

    Key **mod** TableSize

  - But if for instance, if the table size is 10 and the keys all end in zero choice of hash function needs to be carefully considered. The hash function (Key mod TableSize) is a bad choice.
  - It is a good idea to ensure that the **table size is prime!** **Why?**
    - Real-life data tends to have a pattern.
    - “Multiples of 61” are probably less likely than “multiples of 60”.
  - If the input keys are random integers, then the function *Key* **mod** *TableSize* is a very simple to compute and distributes the keys evenly.

Hash Function

Some considerations:

- If the input keys are integers, then simply returning Key mod TableSize is generally a reasonable strategy.
- If the keys are strings the **hash function** can be chosen **adding up the ASCII** (or **Unicode**) values of the characters in the string. Example:

```
public static int hash(String key, int tableSize)
{
    int hashVal = 0;

    for( int i = 0; i < key.length(); i++ )
        hashVal += key.charAt(i);

    return hashVal % tableSize;
 }
```

Hash Function

- **The <u>first</u> hash function:**

```
public static int hash(String key, int tableSize)
{
   int hashVal = 0;

   for( int i = 0; i < key.length(); i++ )
      hashVal += key.charAt(i);

   return hashVal % tableSize;
 }
```

- **But:** if the table size is large, the function does not distribute the keys well.
- **Example:**
  - **TableSize = 10,007** (10,007 is a prime number) and suppose that $\forall$keys, length(keys) ≤ 8
  - Facts: **0 $\leq$key.charAt(i) $\leq$127** then

    0 $\leq$hashVal $\leq$ 1,016, which is 127$*$8 then

The keys will not have a uniform distribution in the table!

9

Hash Function

- **The <u>second</u> hash function:**

```
public static int hash(String key, int tableSize)
{
   return (key.charAt(0) + 27 * key.charAt(1) +
           729 * key.charAt(2)) % tableSize;
}
```

- This **hash function** assumes that *Key* has at least three characters.
- The value 27 represents the number of letters in the English alphabet (26 plus the blank) and 729 is 27^2. The Max value is 96139 (127 + 27*127 + 729 * 127)
- This function examines only the first three characters but if these are random and the table size is 10,007, as before, then we would expect a reasonably uniform distribution.
- **Unfortunately, English is not random** and on reality only 28 percent on the table can be used by hashed to.

10

Hash Function

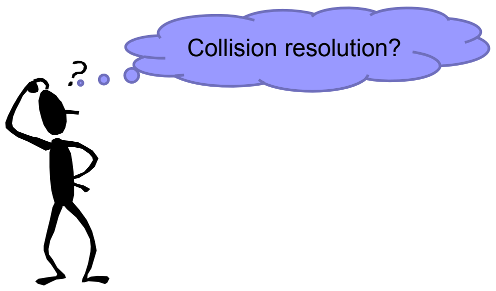
- **The <u>third</u> hash function:**

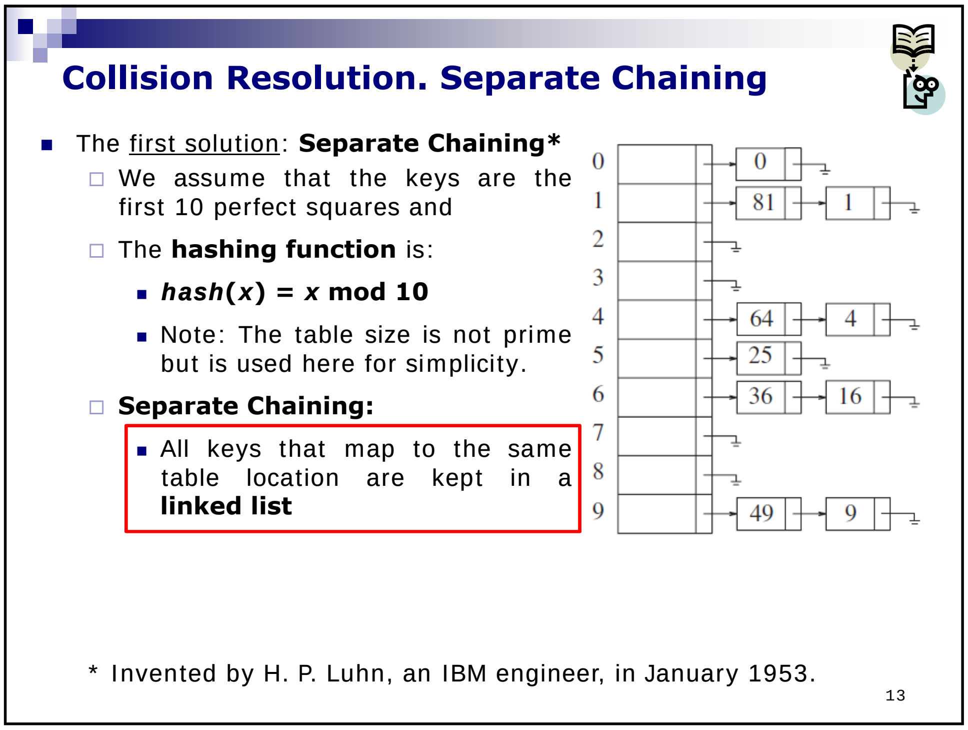
```
public static int hash(String key, int tableSize)
{
    int hashVal = 0;
    for( int i = 0; i < key.length(); i++ )
        hashVal = 37 * hashVal + key.charAt(i);

    hashVal %= tableSize;
    if( hashVal < 0 )    //overflow case
        hashVal += tableSize;
    return hashVal;
}
```

- This **hash function** involves all characters in the key and can generally be expected to distribute well.
- The code computes a polynomial function (of 37) by use of **Horner's rule**.
- The **hash function** takes advantage of the fact that overflow is allowed. This may introduce a negative number; thus the extra test at the end.

Collision

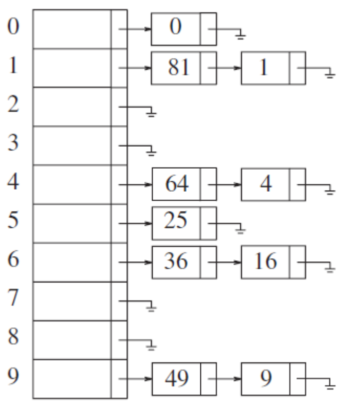
- **The Collision Problem:**
  - When <u>two keys map to the same location in the hash table</u>
  - **Naïve solution:**
    - We try to avoid it, but <u>number-of-keys exceeds table size</u>
  - So **hash tables** should support **collision resolution (handling)**





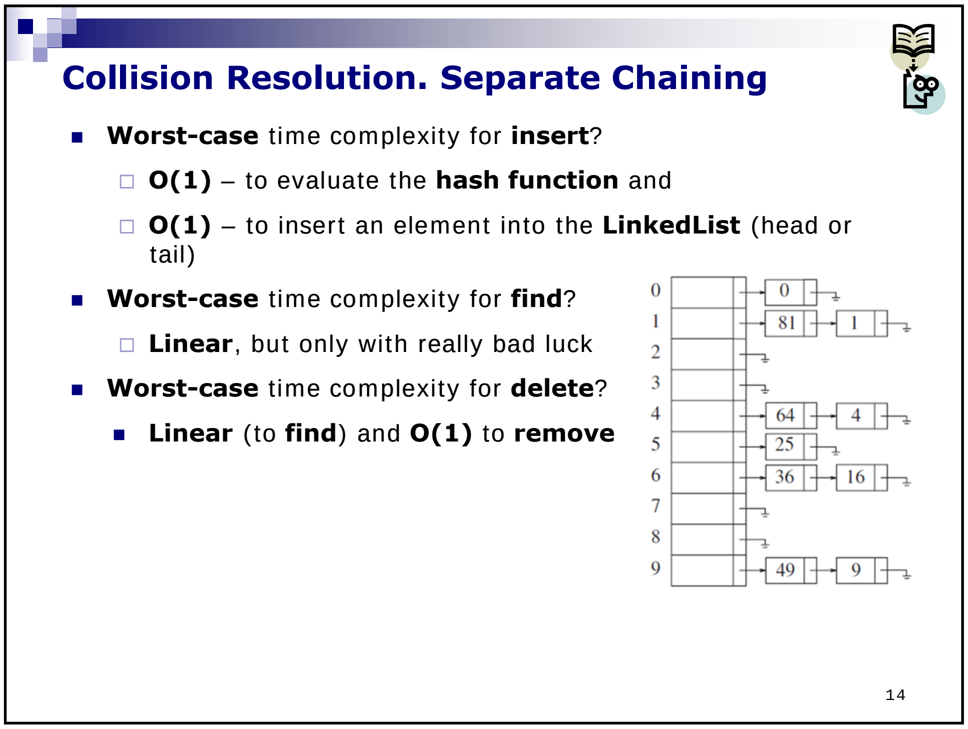
Collision Resolution. Separate Chaining

- The first solution: **Separate Chaining***
  - We assume that the keys are the first 10 perfect squares and
  - The **hashing function** is:
    - ***hash*(*x*) = *x* mod 10**
    - Note: The table size is not prime but is used here for simplicity.
  - **Separate Chaining:**
    - All keys that map to the same table location are kept in a **linked list**



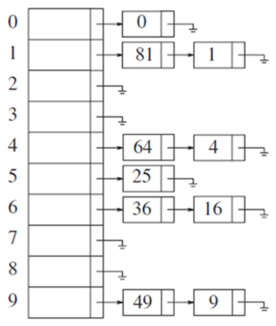
* Invented by H. P. Luhn, an IBM engineer, in January 1953.

13



Collision Resolution. Separate Chaining

- **Worst-case** time complexity for **insert**?
  - **O(1)** – to evaluate the **hash function** and
  - **O(1)** – to insert an element into the **LinkedList** (head or tail)
- **Worst-case** time complexity for **find**?
  - **Linear**, but only with really bad luck
- **Worst-case** time complexity for **delete**?
  - **Linear** (to **find**) and **O(1)** to **remove**



14

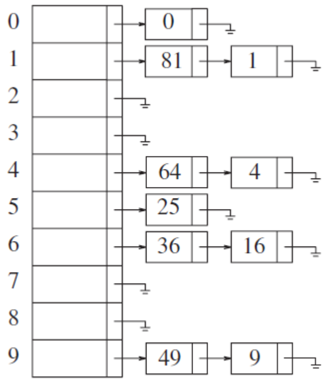
Separate Chaining. The load factor

- **Definition:** The **load factor**, λ, of a **hash table** is:

$$\boxed{\lambda = \frac{N}{TableSize}}$$

where N is the number of items in the table

- Example of table with λ = 1



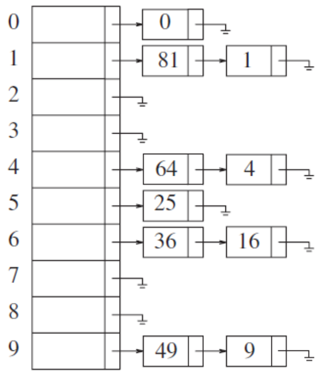
15

Separate Chaining. The load factor

Important points:

- The **average length** of a list is λ
- The **computational effort** for **search** is:
  - O(1) – to evaluate de hash function

    +
  - the time to traverse the list.
- In an **unsuccessful search**, the number of nodes to examine is λ on average.
- In a **successful search** the number of nodes to examine is about $\lambda/2$ on average.
- So, the general rule for **separate chaining** is to make the table size about as large as the number of elements expected $\lambda \cong$ **1**



16

Separate Chaining. Java Code

- See Java Code for Separate Chaining in:

http://users.cis.fiu.edu/~weiss/dsaajava2/code/SeparateChainingHashTable.java

Author: Prof. Mark A. Weiss

17

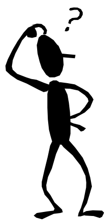
Separate Chaining.

Advantages

- Used when memory is of concern, easily implemented.

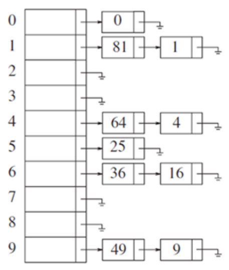
Disadvantages

- **Parts of the table/array might never be used.**
- As chains get longer, search time increases to **O(n)** in the worst case.



Next Question:

- Is there a way to use the "**unused**" space in the table/array instead of using chains to make more space?



18