Fall 2015

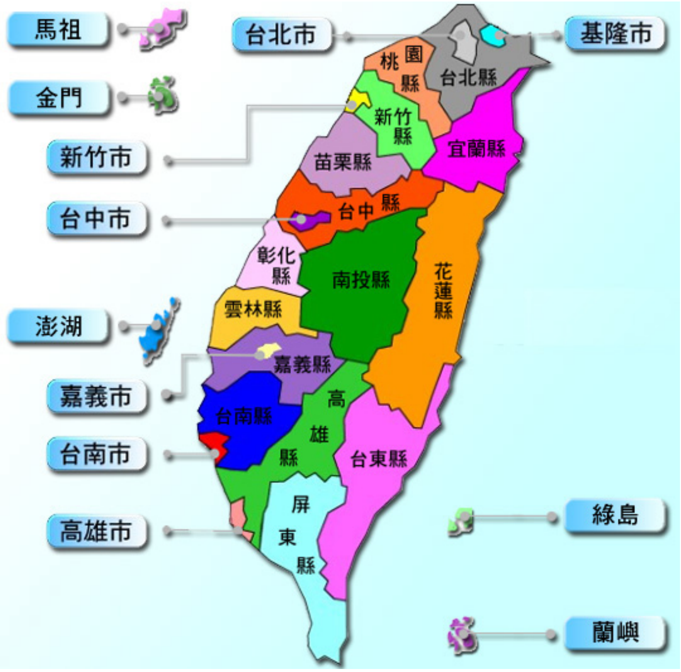
Data Structures Lecture 12

Fang Yu

Software Security Lab.
Dept. Management Information Systems,
National Chengchi University

Announcement

- We will talk about Hash Table today and Graphs on Jan. 8 as the last two lectures
- The project demo is scheduled on Jan. 14
- The makeup exam will be held on Jan. 14.
  - The questions cover the materials in the last three lectures: dynamic programming, binary search tree, maps and hash tables, and graphs



Advance ADTs

Maps and Hash Tables

Maps

- A map models a searchable collection of key-value entries
- Key is unique
- The main operations of a map are for searching, inserting, and deleting items
- Multiple entries with the same key are not allowed
- Applications:
  - address book
  - student-record database

The Map ADT

- get(k): if the map M has an entry with key k, return its associated value; else, return null
- put(k, v): insert entry (k, v) into the map M; if key k is not already in M, then return null; else, return the old value associated with k
- remove(k): if the map M has an entry with key k, remove it from M and return its associated value; else, return null

The Map ADT

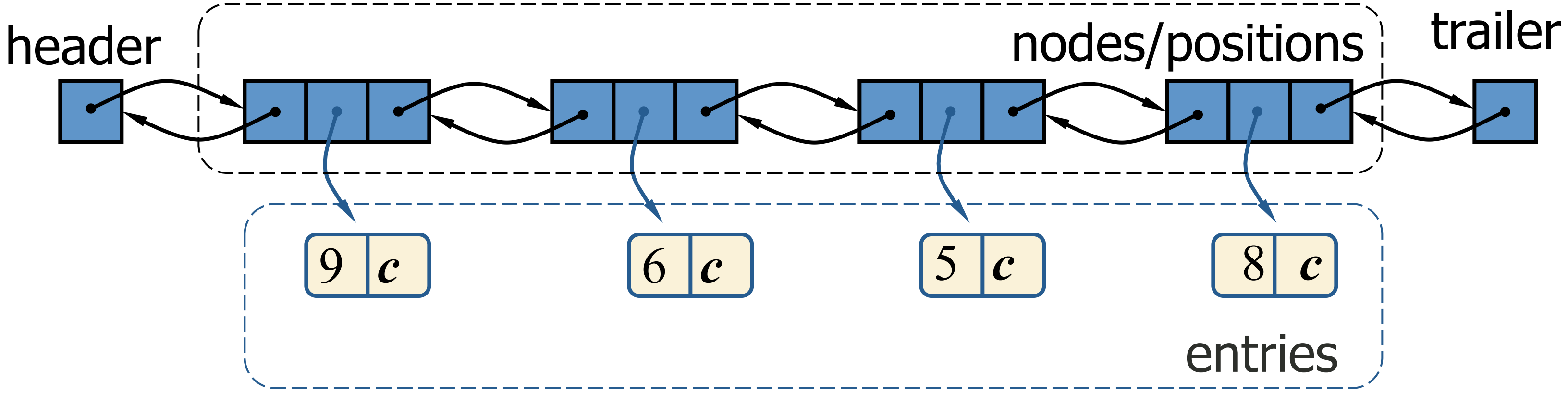
- size(), isEmpty()
- entrySet(): return an iterable collection of the entries in M
- keySet(): return an iterable collection of the keys in M
- values(): return an iterator of the values in M

Example

Operation	***Output***	***Map***
isEmpty()	**true**	Ø
put(5,*A*)	**null**	(5,*A*)
put(7,*B*)	**null**	(5,*A*),(7,*B*)
put(2,*C*)	**null**	(5,*A*),(7,*B*),(2,*C*)
put(8,*D*)	**null**	(5,*A*),(7,*B*),(2,*C*),(8,*D*)
put(2,*E*)	*C*	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
get(7)	*B*	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
get(4)	**null**	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
get(2)	*E*	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
size()	4	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
remove(5)	*A*	(7,*B*),(2,*E*),(8,*D*)
remove(2)	*E*	(7,*B*),(8,*D*)
get(2)	**null**	(7,*B*),(8,*D*)
isEmpty()	**false**	(7,*B*),(8,*D*)

A Simple List-Based Map

- We can efficiently implement a map using an unsorted list
  - We store the items of the map in a list S (based on a doubly-linked list), in arbitrary order



The get(k) Algorithm

```
Algorithm get(k):
  B = S.positions() //B is an iterator of the positions in S
  while B.hasNext() do
        p = B.next()          // the next position in B
        if p.element().getKey() = k   then
                return p.element().getValue()
  return null //there is no entry with key equal to k
```

The put(k,v) Algorithm

```
Algorithm put(k,v):
B = S.positions()
while B.hasNext() do
   p = B.next()
   if p.element().getKey() = k  then
            t = p.element().getValue()
            S.set(p,(k,v))
            return t   //return the old value
S.addLast((k,v))
n = n + 1              //increment variable storing number of entries
return null            //there was no entry with key equal to k
```

The remove(k) Algorithm

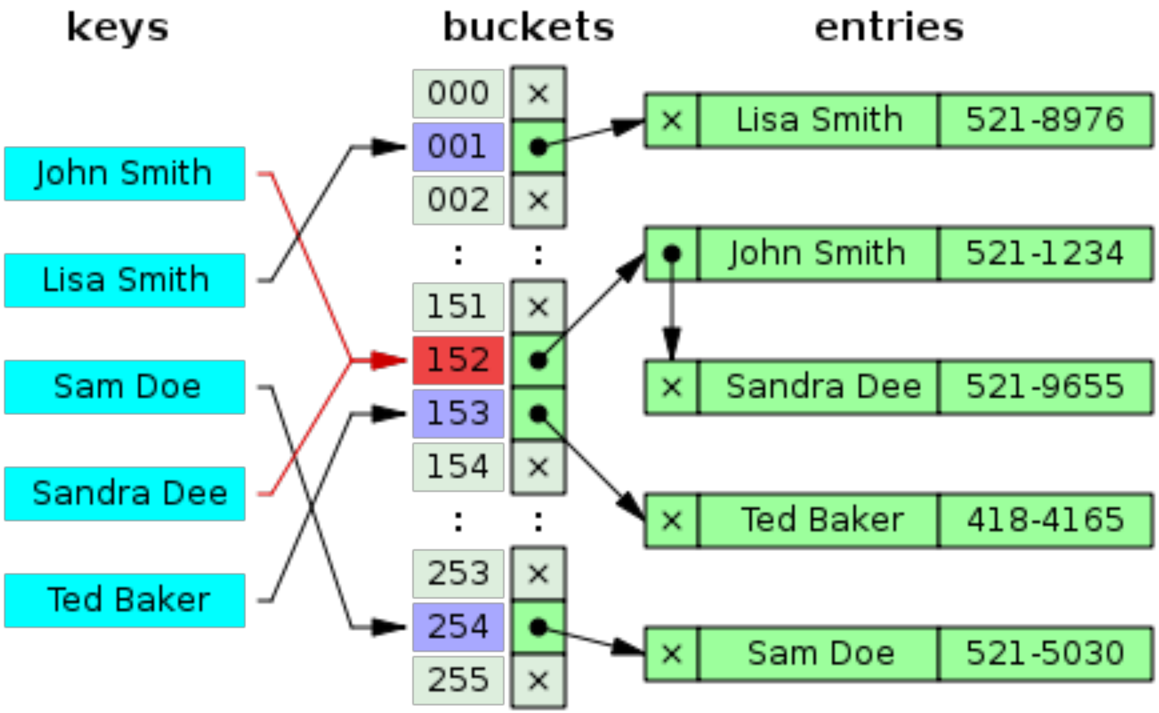
```
Algorithm remove(k):
B =S.positions()
while B.hasNext() do
   p = B.next()
   if p.element().getKey() = k  then
            t = p.element().getValue()
            S.remove(p)
            n = n – 1           //decrement number of entries
            return t            //return the removed value
return null                     //there is no entry with key equal to k
```

Performance of a List-Based Map

- Performance:
  - put takes $O(1)$ time since we can insert the new item at the beginning or at the end of the sequence
  - get and remove take $O(n)$ time since in the worst case (the item is not found) we traverse the entire sequence to look for an item with the given key
- The unsorted list implementation is effective only for maps of small size or for maps in which puts are the most common operations, while searches and removals are rarely performed (e.g., historical record of logins to a workstation)

Hash Tables

- Use keys to store and access entries (in constant time)



Hash Functions and Hash Tables

- A hash function h maps keys of a given type to integers in a fixed interval $[0, N-1]$

- Example:

  $$h(x) = x \bmod N$$

  is a hash function for integer keys

- The integer $h(x)$ is called the hash value of key x

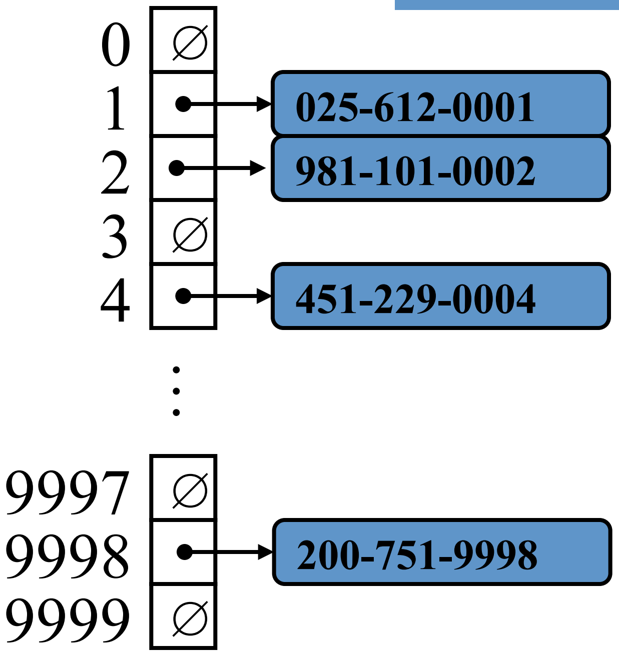
- A hash table for a given key type consists of
  - Hash function h
  - Array (called table) of size N

- When implementing a map with a hash table, the goal is to store item (k, o) at index $i = h(k)$

Example

- We design a hash table for a map storing entries as (SSN, Name), where SSN (social security number) is a nine-digit positive integer

- Our hash table uses an array of size $N = 10{,}000$ and the hash function
  $\boldsymbol{h}(\boldsymbol{x})$ = last four digits of $\boldsymbol{x}$



Hash Functions

- A hash function is usually specified as the composition of two functions:

  Hash code:
  $\boldsymbol{h}_1$: keys → integers

  Compression function:
  $\boldsymbol{h}_2$: integers → $[0, N - 1]$

- The hash code is applied first, and the compression function is applied next on the result, i.e.,

  $$\boldsymbol{h}(\boldsymbol{x}) = \boldsymbol{h}_2(\boldsymbol{h}_1(\boldsymbol{x}))$$

- The goal of the hash function is to "disperse" the keys in an apparently random way

Hash Codes

- Memory address:
  - We reinterpret the memory address of the key object as an integer (default hash code of all Java objects)
  - Good in general, except for numeric and string keys
- Integer cast:
  - We reinterpret the bits of the key as an integer
  - Suitable for keys of length less than or equal to the number of bits of the integer type (e.g., byte, short, int and float in Java)
- Component sum:
  - We partition the bits of the key into components of fixed length (e.g., 16 or 32 bits) and we sum the components (ignoring overflows)
  - Suitable for numeric keys of fixed length greater than or equal to the number of bits of the integer type (e.g., long and double in Java)

Hash Codes (cont.)

- Polynomial accumulation:
  - We partition the bits of the key into a sequence of components of fixed length (e.g., 8, 16 or 32 bits)

    $$a_0\, a_1 \,\ldots\, a_{n-1}$$

  - We evaluate the polynomial

    $$p(z) = a_0 + a_1 z + a_2 z^2 + \ldots$$
    $$\ldots + a_{n-1} z^{n-1}$$

    at a fixed value z, ignoring overflows
  - Especially suitable for strings (e.g., the choice $z = 33$ gives at most 6 collisions on a set of 50,000 English words)

- Polynomial $p(z)$ can be evaluated in $O(n)$ time using Horner's rule:
  - The following polynomials are successively computed, each from the previous one in $O(1)$ time

    $$p_0(z) = a_{n-1}$$
    $$p_i(z) = a_{n-i-1} + z p_{i-1}(z)$$
    $$(i = 1, 2, \ldots, n-1)$$

- We have $p(z) = p_{n-1}(z)$

Compression Functions

- Division:
  - $h_2(y) = y \bmod N$
  - The size N of the hash table is usually chosen to be a prime
  - The reason has to do with number theory and is beyond the scope of this course

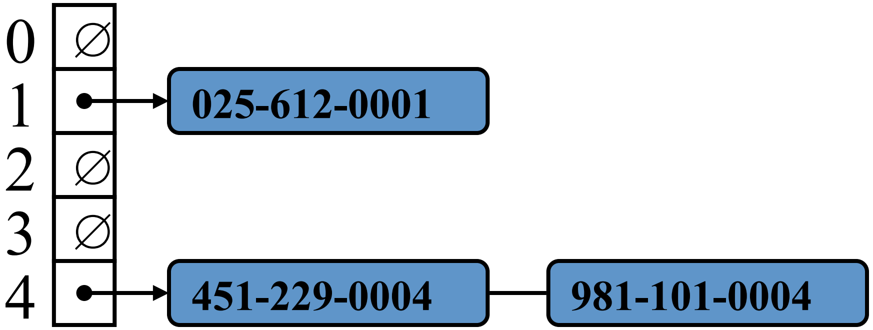
- Multiply, Add and Divide (MAD):
  - $h_2(y) = (ay + b) \bmod N$
  - a and b are nonnegative integers such that

    $$a \bmod N \neq 0$$
  - Otherwise, every integer would map to the same value b

Collision Handling

- Collisions occur when different elements are mapped to the same cell

- Separate Chaining: let each cell in the table point to a linked list of entries that map there



0 ∅

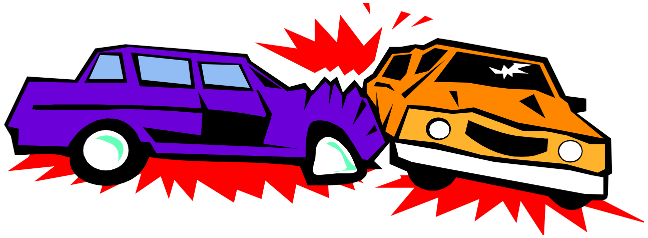
1 → 025-612-0001

2 ∅

3 ∅

4 → 451-229-0004 — 981-101-0004

- Separate chaining is simple, but requires additional memory outside the table



Map with Separate Chaining

```
**Algorithm** get(k):
return A[h(k)].get(k)
```

```
**Algorithm** put(k,v):
t = A[h(k)].put(k,v)
if t = null then //k is a new key
    n = n + 1
return t
```

```
**Algorithm** remove(k):
t = A[h(k)].remove(k)
if t ≠ null then    //k was found
    n = n - 1
return t
```

Linear Probing

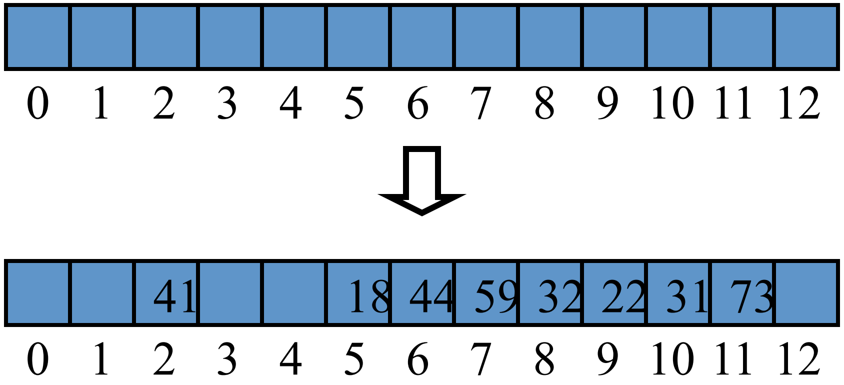
- Open addressing: the colliding item is placed in a different cell of the table

- Linear probing: handles collisions by placing the colliding item in the next (circularly) available table cell

- Each table cell inspected is referred to as a “probe”

- Colliding items lump together, causing future collisions to cause a longer sequence of probes

- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



0	1	2	3	4	5	6	7	8	9	10	11	12

⇩

		41			18	44	59	32	22	31	73	
0	1	2	3	4	5	6	7	8	9	10	11	12

Search with Linear Probing

- Consider a hash table A that uses linear probing

- get(k)
  - We start at cell $h(k)$
  - We probe consecutive locations until one of the following occurs
    - An item with key k is found, or
    - An empty cell is found, or
    - N cells have been unsuccessfully probed

```
Algorithm get($k$)
    $i \leftarrow h(k)$
    $p \leftarrow 0$
    repeat
        $c \leftarrow A[i]$
        if $c = \varnothing$
            return $null$
        else if $c.getKey() = k$
            return $c.getValue()$
        else
            $i \leftarrow (i + 1) \bmod N$
            $p \leftarrow p + 1$
    until $p = N$
    return $null$
```

Updates with Linear Probing

- To handle insertions and deletions, we introduce a special object, called ***AVAILABLE***, which replaces deleted elements
- remove(***k***)
  - We search for an entry with key ***k***
  - If such an entry (***k, o***) is found, we replace it with the special item ***AVAILABLE*** and we return element ***o***
  - Else, we return ***null***

Updates with Linear Probing

- put(***k, o***)
  - We throw an exception if the table is full
  - We start at cell ***h***(***k***)
  - We probe consecutive cells until one of the following occurs
    - A cell ***i*** is found that is either empty or stores ***AVAILABLE***, or
    - ***N*** cells have been unsuccessfully probed
  - We store (***k, o***) in cell ***i***

Double Hashing

- Double hashing uses a secondary hash function $d(k)$ and handles collisions by placing an item in the first available cell of the series
  $$(i + jd(k)) \bmod N$$
  for $j = 0,\ 1, \dots, N - 1$
- The secondary hash function $d(k)$ cannot have zero values
- The table size N must be a prime to allow probing of all the cells

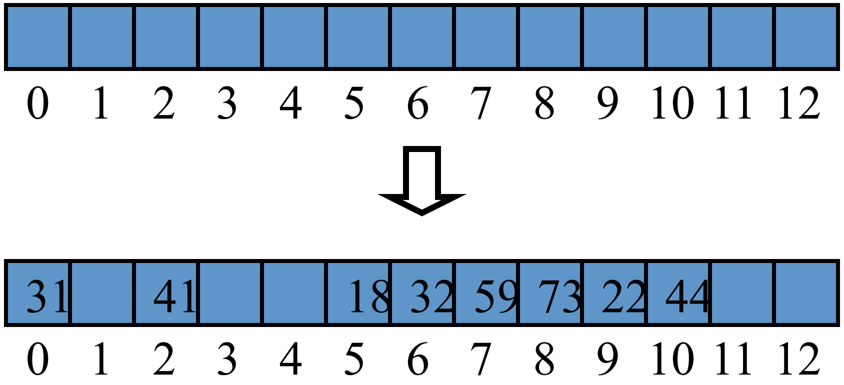
- Common choice of compression function for the secondary hash function:
  $d_2(k) = q - k \bmod q$

  where
  - $q < N$
  - q is a prime
- The possible values for $d_2(k)$ are
  $$1, 2, \dots, q$$

Example of Double Hashing

- Consider a hash table storing integer keys that handles collision with double hashing
  - $N = 13$
  - $h(k) = k \bmod 13$
  - $d(k) = 7 - k \bmod 7$
- Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

k	$h(k)$	$d(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		



0	1	2	3	4	5	6	7	8	9	10	11	12

⇩

31		41			18	32	59	73	22	44		
0	1	2	3	4	5	6	7	8	9	10	11	12

Performance of Hashing

- In the worst case, searches, insertions and removals on a hash table take $O(n)$ time
- The worst case occurs when all the keys inserted into the map collide
- The load factor $\alpha = n/N$ affects the performance of a hash table
- Assuming that the hash values are like random numbers, it can be shown that the expected number of probes for an insertion with open addressing is

  $1 / (1 - \alpha)$

Performance of Hashing

- The expected running time of all the dictionary ADT operations in a hash table is $O(1)$
- In practice, hashing is very fast provided the load factor is not close to 100%
- Applications of hash tables:
  - small databases
  - compilers
  - browser caches

HW11 (Due on Jan. 10)

Webrize BMI!

- Create a web page that takes users' height and weight and return his/her BMI
- This is the final HW. Use the same skills to webrize your project

BONUS HW12 (Due on Jan. 10)

Build a hash table for websites!

- A website w consists of three fields: (url, name, score)
- Use w’s url to compute w’s hash key
  - Hash code: url → integer
  - Compression function: integer → [0..N-1]
- Use separate chains/linear probing/double hashing to resolve collisions
- You may use java.util.Hashtable

Operations

Given a sequence of operations in a txt file, parse the txt file and execute each operation accordingly

operations	description
put(key k, website w)	Put a new website by its key k to the hash table
find(key k)	Return the website associated with k
remove(key k)	Remove the website associated with k

An input file

Similar to HW10,

1. You need to read the sequence of operations from a txt file
2. The format is firm
3. Raise an exception if the input does not match the format

```
put www.nccu.edu.tw NCCU 8
put www.cs.ucsb.edu UCSB 3
put www.google.com Google 5
find www.google.com
remove www.cs.ucsb.edu
find www.nccu.edu.tw
```

```
[Google, http://www.google.com, 5]
[NCCU, http://www.nccu.edu.tw, 8]
```

Project Demo on Jan. 14

Beat Google:

- Stage 1 : Rank web pages by keywords
- Stage 2 : Rank web sites by keywords
- Stage 3 : Re-rank google web sites by keywords
- Stage 4 : Derive relative keywords by top-ranked web sites
- Stage 5: *Webrize* your search engine
- Stage 6: Mobilize your search engine

Project Demo

- Location: The MIS 5F PC classroom
- Each team gives 8 minutes PPT presentation focusing on the project interests, key ideas, and achievements + 7 minutes system demo
- In the demo, each team needs to **run your system** to show how it works and how it achieves the requirement for each stage. I will also check your source code.
- BONUS: Students who successfully challenge other team’s system may get extra points.