

CS206 Data Structures

Maps & Hashes

Sung-eui Yoon (윤성의)

Department of Computer Science
KAIST

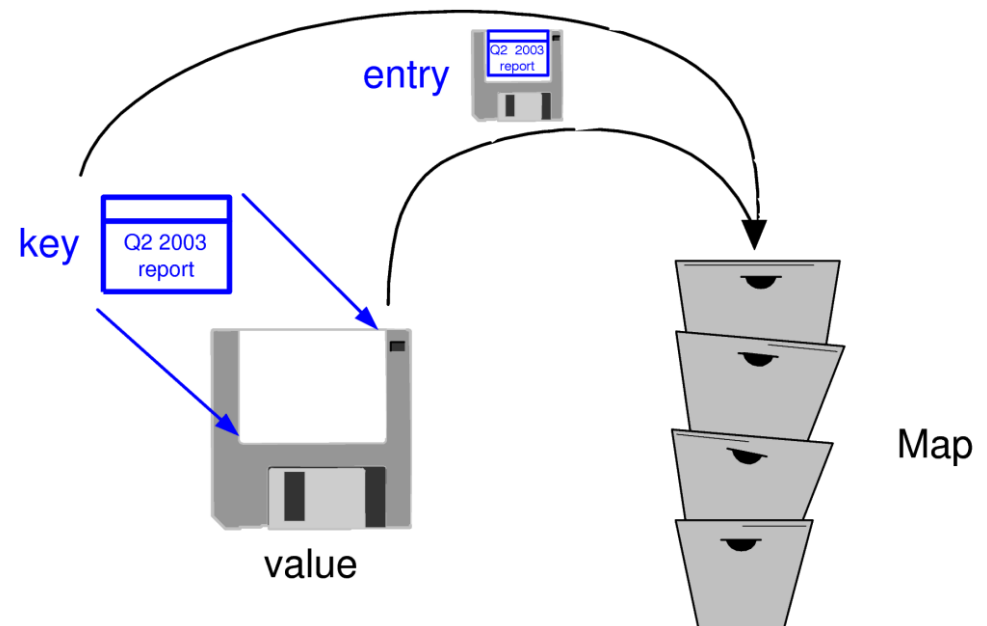
<http://sglab.kaist.ac.kr/~sungeui>

Class Objectives (Ch. 10)

- Understand the concept of a map and implement a list-based map
- Understand a hash and its different operations

Maps

- ❑ A map models a searchable collection of key-value entries
- ❑ 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

□ Map ADT methods:

- `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 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
- `size()`, `isEmpty()`
- `keys()`: return an iterator of the keys in `M`
- `values()`: return an iterator of the values in `M`

Example

<i>Operation</i>	<i>Output</i>	<i>Map</i>
isEmpty()	true	\emptyset
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)

Comparison to java.util.Map

Map ADT Methods

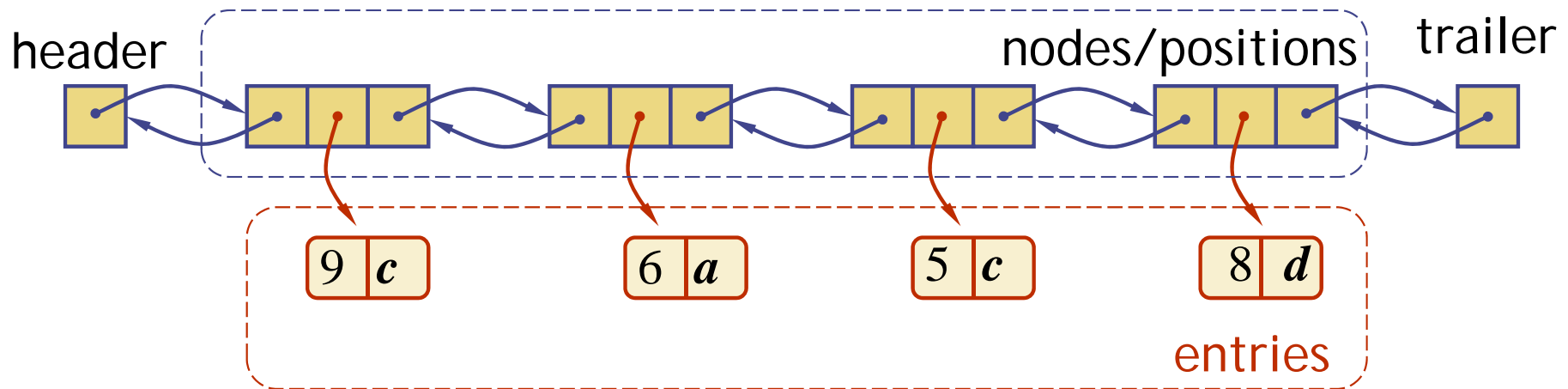
size()
isEmpty()
get(k)
put(k, v)
remove(k)
keys()
values()

java.util.Map Methods

size()
isEmpty()
get(k)
put(k, v)
remove(k)
keySet().iterator()
values().iterator()

A Simple List-Based Map

- We can 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

$p = B.next()$

 if $p.element().key() = k$ then

 return $p.element().value()$

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 $p.element().key() = k$ then

$t = p.element().value()$

$B.replace(p, (k, v))$

 return t {return the old value}

$S.insertLast((k, v))$

$n = n + 1$ {increment variable storing number of entries}

return null {there was no previous entry with key equal to k }

The remove(k) Algorithm

Algorithm remove(k):

$B = S.positions()$

while $B.hasNext()$ do

$p = B.next()$

 if $p.element().key() = k$ then

$t = p.element().value()$

$S.remove(p)$

$n = n - 1$ {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 can take $O(1)$ 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

Recall the Map ADT

□ Map ADT methods:

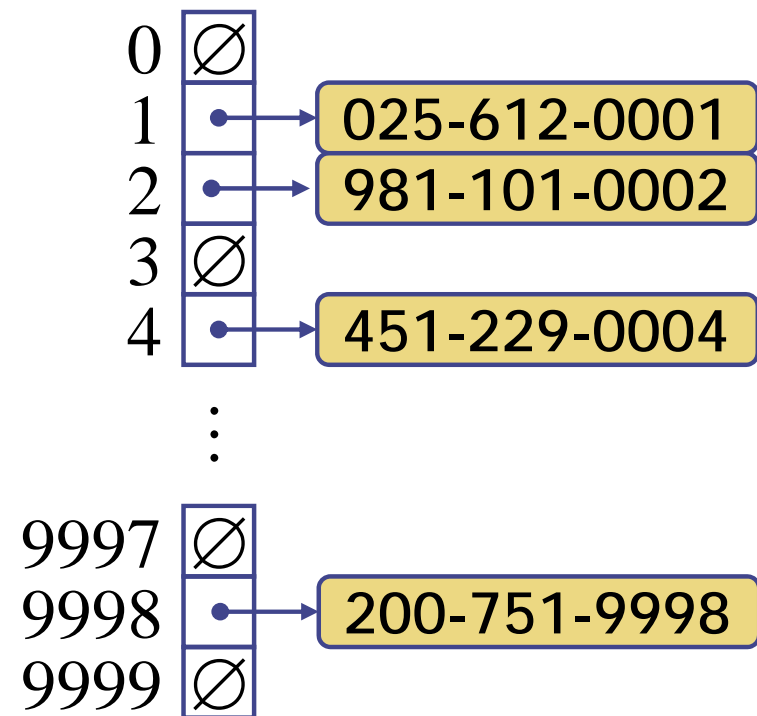
- `get(k)`: if the map `M` has an entry with key `k`, return its associated value; else, return null
- `put(k, v)`: insert entry `(k, v)` into the map `M`; if key `k` is not already in `M`, then return null; else, return old value associated with `k`
- `remove(k)`: if the map `M` has an entry with key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

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 $h(x) = \text{last four digits of } x$



Hash Functions

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

Hash code:

$h_1: \text{keys} \rightarrow \text{integers}$

Compression function:

$h_2: \text{integers} \rightarrow [0, N - 1]$

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

$$h(x) = h_2(h_1(x))$$

□ 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)

□ Integer cast:

- We reinterpret the bits of the key as an integer

□ 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)

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)

$$a_0 \ a_1 \ \dots \ a_{n-1}$$

- We evaluate the polynomial

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

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

$$p_0(z) = a_{n-1}$$

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

□ We have $p(z) = p_{n-1}(z)$

Compression Functions

□ Division:

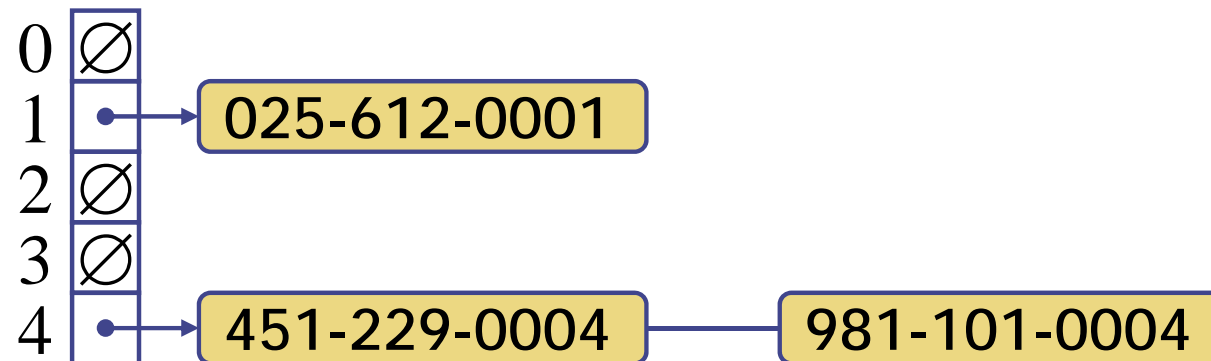
- $h_2(y) = y \bmod N$
- The size N of the hash table is usually chosen to be a prime
- The reason has to do with number theory and is beyond the scope of this course

□ Multiply, Add and Divide (MAD):

- $h_2(y) = (a y + b) \bmod N$
- 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



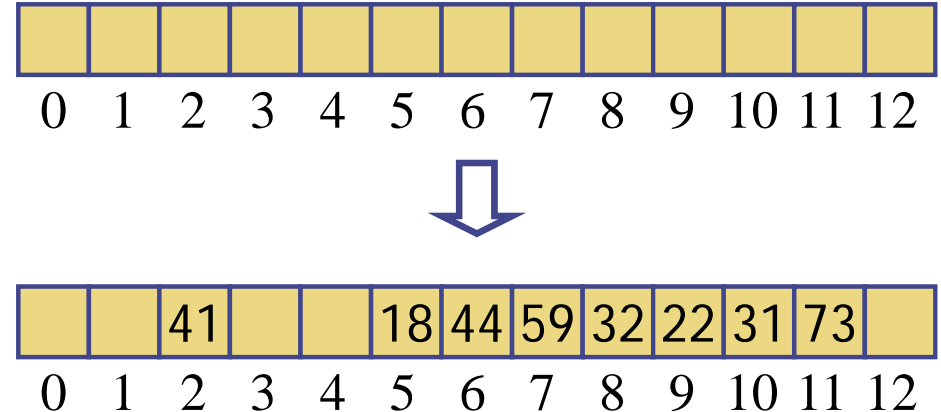
- Separate chaining is simple, but requires additional memory outside the table

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



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 ← $h(k)$

p ← 0

repeat

c ← $A[i]$

if $c = \emptyset$

return *null*

else if $c.key() = k$

return *c.element()*

else

i ← $(i + 1) \bmod N$

p ← $p + 1$

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`

□ `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 entry `(k, o)` in cell `i`

Double Hashing

- Double hashing uses a secondary hash function $h'(k)$ and handles collisions by placing an item in the first available cell of the series

$$(i + j h'(k)) \bmod N$$

for $j = 0, 1, \dots, N - 1$

- The secondary hash function $h'(k)$ 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:

$$h'(k) = q - (k \bmod q) \text{ where}$$

- $q < N$
- q is a prime

- The possible values for $h'(k)$ are

$$1, 2, \dots, q$$

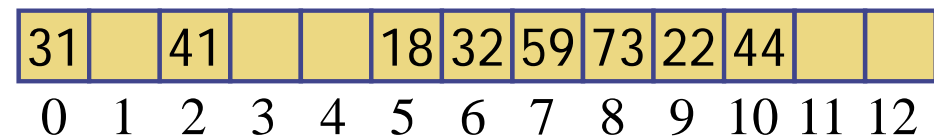
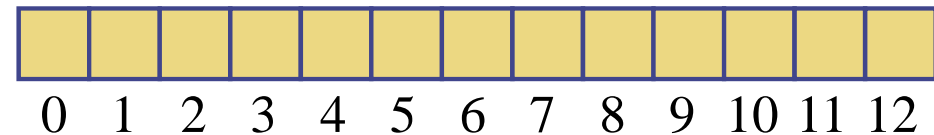
Example of Double Hashing

□ Consider a hash table storing integer keys that handles collision with double hashing

- $N = 13$
- $h(k) = k \bmod 13$
- $h'(k) = 7 - k \bmod 7$

□ Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

k	$h(k)$	$h'(k)$	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



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%
 - Apply rehashing when α is close to 1 (e.g. reshaping when it is 0.75)
- Applications of hash tables:
 - small databases
 - browser caches

Class Objectives (Ch. 10) were:

- Understand the concept of a map and implement a list-based map
- Understand a hash and its different operations

Next Time

Search trees

Questions:

- Come up with one question on what we have discussed in the class and submit at the end of the class
- 1 for typical questions and 2 for questions with thoughts or that surprised me
- Write questions at least 4 times; you can type at KLMS

HW:

- Go over the next lecture slides before the class
- Just 10 min ~ 20 min should be okay