Data Structure:

Maps and Dictionaries

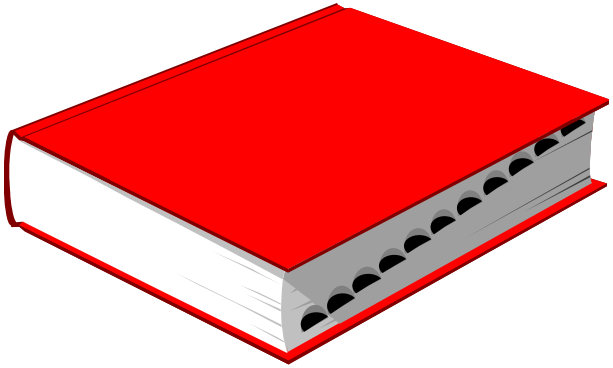
2017

Instructor: Prof. Young-guk Ha
Dept. of Computer Science & Engineering

Contents

- Data structures to be discussed
  - Maps
    - List-based map
    - Hash map
      - ***Hash table*** data structure
  - Dictionaries
    - List-based dictionary (unordered list)
    - Hash table dictionary
    - Search table dictionary (ordered list)
      - ***Binary search*** algorithm

Dictionaries



Dictionary

- Like a map, a ***dictionary*** models a searchable collection of (*key*, *value*) entries
- Unlike a map, multiple entries with the same key are allowed
  - Cf., **`put`** of map vs. **`insert`** of dictionary
- The main operations of a dictionary are also searching, inserting and deleting entries
- Applications
  - English dictionary (same words)
  - Phone book (same names)
  - Etc.

Dictionary Operations

- `entry` **`find`**`(`***k***`)`
  - If the dictionary has an entry with key ***k***, returns it, else, returns null
- `list` **`findAll`**`(`***k***`)`
  - Returns a collection (list) of all entries with key ***k***
- `entry` **`insert`**`(`***k***`, `***v***`)`
  - Inserts and returns the entry (***k***, ***v***)
  - Just inserts, does **NOT** replace the old entry
- `entry` **`remove`**`(`***e***`)`
  - Removes and returns the entry ***e*** from the dictionary, if fails, returns null
  - Cf., `entry` **`remove`**`(`***k***`)`: find an entry with key ***k*** and remove it
- `list` **`entries`**`()`
  - Returns a list of all the entries in the dictionary
- Common operations: **`size`**`()`, **`isEmpty`**`()`

Example

Operation	*Output*	*Dictionary*
insert(5,*A*)	(5,*A*)	(5,*A*)
insert(7,*B*)	(7,*B*)	(5,*A*), (7,*B*)
insert(2,*C*)	(2,*C*)	(5,*A*), (7,*B*), (2,*C*)
insert(8,*D*)	(8,*D*)	(5,*A*), (7,*B*), (2,*C*), (8,*D*)
insert(2,*E*)	(2,*E*)	(5,*A*), (7,*B*), (2,*C*), (8,*D*), (2,*E*)
find(7)	(7,*B*)	(5,*A*), (**7**,*B*), (2,*C*), (8,*D*), (2,*E*)
find(4)	**null**	(5,*A*), (7,*B*), (2,*C*), (8,*D*), (2,*E*)
find(2)	(2,*C*)	(5,*A*), (7,*B*), (**2**,*C*), (8,*D*), (2,*E*)
findAll(2)	{(2,*C*), (2,*E*)}	(5,*A*), (7,*B*), (**2**,*C*), (8,*D*), (**2**,*E*)
size()	5	(5,*A*), (7,*B*), (2,*C*), (8,*D*), (2,*E*)
remove(find(5))	(5,*A*)	(7,*B*), (2,*C*), (8,*D*), (2,*E*)
find(5)	**null**	(7,*B*), (2,*C*), (8,*D*), (2,*E*)

A List-Based Dictionary

- A dictionary implemented by means of an unordered list is called ***log file*** or ***audit trail***
  - Stores the entries of the dictionary in a list (node list or array list) in arbitrary order
- Performance
  - **`insert`** takes $O(1)$ time, since we can just insert the new entry at the beginning or at the end of the list
  - **`find`** and **`remove`** take $O(n)$ time, since we must traverse the entire list to look for an entry with the given key in the worst case
- The log file is effective only for
  - Dictionaries of small size, or
  - Dictionaries on which insertions are the most common operations, while searches and removals are rarely performed (e.g., system logs)

Find-All Operation

```
Algorithm findAll(k):
    Input: A key k
    Output: An iterable collection of entries with key equal to k
    Create an initially-empty list L
    for each entry e in D.entries() do
        if e.getKey() = k then
            L.addLast(e)
    return L          {the elements in L are the selected entries}
```

Insertion and Removal Operations

Algorithm insert(k, v):
Input: A key k and value v
Output: The entry (k, v) added to D
Create a new entry $e = (k, v)$
S.addLast(e) {S is unordered}
return e

Algorithm remove(e):
Input: An entry e
Output: The removed entry e
{We don't assume here that e stores its location in S}
for each node ***p*** in ***S***.nodes() **do**
  if p.element() $= e$ **then**
    S.remove(p)
    return e
return null {there is no entry e in D}

* ***S***: node list implementing the dictionary ***D***

Hash Table Dictionary

- We can also use the hash table to implement dictionaries
  - With separate chaining, we have a list-based dictionary at each cell in the bucket array
  - If we use separate chaining to handle collisions, actual operations can be delegated to the list-based dictionary stored at each cell

- Performance
  - **`find`**, **`remove`** and **`insert`** take $O(1)$ running time (expected)
  - **`findAll`** takes $O(1 + s)$ expected running time, where s is the number of entries with the same key

Hash Table Implementation

Algorithm insert(k, v):
Input: A key k and value v
Output: The entry (k, v) added to D
if $(n+1)/N > \lambda$ **then** { load factor $\lambda = 0.9$ }
Double the size of A and rehash all the existing entries
$e \leftarrow A[h(k)]$.insert(k, v) { delegate the insert to the list-based dictionary at $A[h(k)]$ }
$n \leftarrow n+1$ { increase the number of entries stored in ***D*** by 1 }
return e

Algorithm findAll(k):
Input: A key k
Output: An iterable collection of entries with key equal to k
return $A[h(k)]$.findAll(k) { delegate the findAll to the list-based dictionary at $A[h(k)]$ }

Algorithm remove(e):
Input: An entry e
Output: The removed entry e
$t \leftarrow A[h(k)]$.remove(e) { delegate the remove to the list-based dictionary at $A[h(k)]$ }
if $t \neq$ **null** $n \leftarrow n-1$ { decrease the number of entries stored in ***D*** by 1 }
return t

Ordered Search Table

- An ***ordered search table*** (***search table*** for short) is a dictionary implemented with a sorted array list
  - Stores the entries of the dictionary in an array list sorted by the keys
  - We can use an external comparator for arbitrary object keys
- Performance of ordered search tables
  - `insert` and `remove` take $O(n)$ time
    - With linear search, we need to probe n entries to find the location of the entry to be inserted or removed
    - In addition, we have to shift n entries at maximum to make room for the new entry or compact the array list after the removal
  - Whereas, we can perform `find` in $O(\log n)$ time using ***binary search***
- A search table is effective for
  - Dictionaries on which searches are the most common operations, while insertions and removals are rarely performed
  - E.g., Credit card authorizations
    - Once a person registers the credit card information, it is retrieved every time he uses the credit card

Binary Search

- ***Binary search*** performs **`find`**(***k***) on an ordered search table
  - Similar to the "high-low" game
  - Starting with n entries, the number of candidate entries is halved at each step
  - Thus, it terminates after $O(\log n)$ number of steps

- Recursive procedure of binary search
  - To find an entry with key k from n entries, let **low** = 0, **high** = n-1 and e be the entry at index **mid** = $\lfloor$(**low** + **high**) / 2$\rfloor$
    - If $k = e$.getKey(), then we have found the entry we were looking for, and the search terminates successfully returning e.
    - If $k < e$.getKey(), then we recur on the first half of the array list, that is, on the range of indices from low to mid $-$ 1. (high = mid−1)
    - If $k > e$.getKey(), we recur on the range of indices from mid + 1 to high. (low = mid+1)
  - If low > high, it is fail to find k in the array list

Binary Search Algorithm

Algorithm BinarySearch$(S, k, \text{low}, \text{high})$: { initial call is BinarySearch(***S***, ***k***, 0, ***n***-1) }

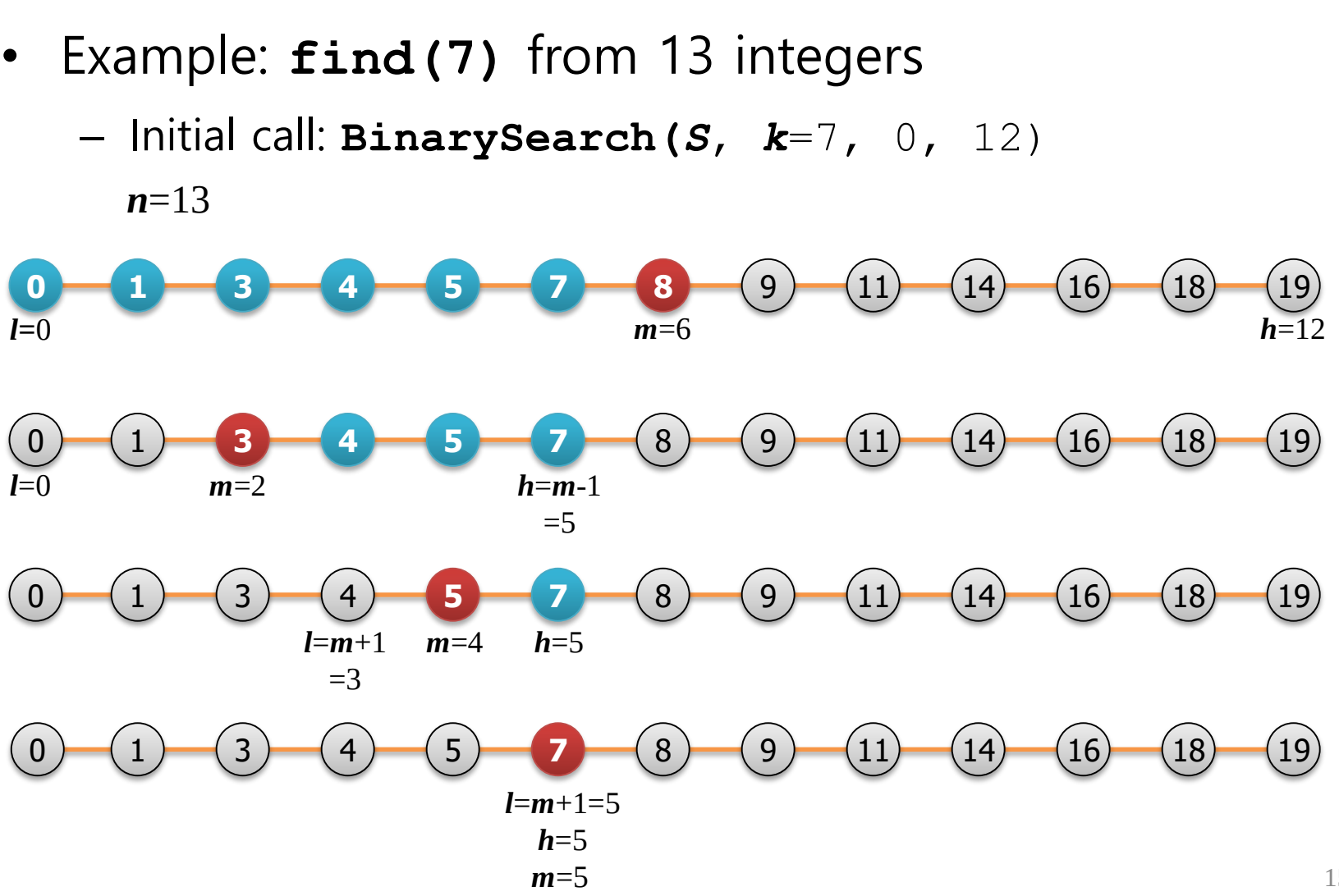
Input: An ordered array list S storing n entries and integers low and high

Output: An entry of S with key equal to k and index between low and high, if such an entry exists, and otherwise **null**

```
if low > high then    { can not find any entry with k }
    return null
else
    mid ← ⌊(low + high)/2⌋
    e ← S.get(mid)
    if k = e.getKey() then    { have found an entry with k }
        return e
    else if k < e.getKey() then
        return BinarySearch(S, k, low, mid − 1)    { search the former half of S }
    else
        return BinarySearch(S, k, mid + 1, high)    { search the latter half of S }
```

Binary Search Example

- Example: **`find(7)`** from 13 integers
  - Initial call: **`BinarySearch(`*`S`*`, `*`k`*`=7, 0, 12)`**



Comparing Dictionary Implementations

Method	List	Hash Table	Search Table
size, isEmpty	$O(1)$	$O(1)$	$O(1)$
entries	$O(n)$	$O(n)$	$O(n)$
find	$O(n)$	$O(1)$ exp., $O(n)$ worst-case	$O(\log n)$
findAll	$O(n)$	$O(1+s)$ exp., $O(n)$ worst-case	$O(\log n + s)$
insert	$O(1)$	$O(1)$ *	$O(n)$
remove	$O(n)$	$O(1)$ exp., $O(n)$ worst-case	$O(n)$

n : the number of entries

s : the number of entries to be returned (i.e., entries with the same key)

* For separate chaining, for open addressing ***O***(1) exp. (***O***(***n***) worst)

Extensions of Dictionaries

- Location-aware dictionary entries
  - As we did for priority queues, we can also use location-aware entries to speed up the **`remove`** operation in a node list-based dictionary
  - E.g., in a node list-based dictionary, **`remove`**(**`e`**) runs in $O(1)$ time with location-aware entries

- ***Ordered dictionary***
  - An ordered dictionary is a dictionary that maintains an order relation among the keys
    - Efficiently implemented with an ordered search table
  - Ordered dictionary operations
    - entry **fisrt**(): returns an entry with the smallest key
    - entry **last**(): returns an entry with the largest key
    - iterator **successors**(k): returns an iterator of the entries with keys $\geq k$
    - iterator **predecessors**(k): returns an iterator of the entries with keys $\leq k$