Associative Arrays

The Map ADT

Introduction

- In Java, there are three basic collection interfaces:
  1. The Set ADT: Stores an *unordered* collection of elements.
  2. The List ADT: Stores a position *ordered* collection of elements.
  3. The Map ADT: Stores a collection of *key-value* pair entries.
- In an ordinary array, a value stored in the array is directly accessed by an integer index corresponding to the *location* where that value is stored.
- A Map (some times called associative array) is an abstract generalization of the ordinary array concept so that a value stored in the array is directly accessed by its associated key.

The Map ADT Definition

- A map models a searchable collection of key-value entries, where each key, k, is associated with a corresponding value, v.
- The main operations of a map are for searching, inserting, and deleting entries.
- Multiple entries with the same key are **not** allowed.
- There are two types of maps: **Sorted** or **Unsorted**.
- The keys and values can be of **any** object type.
- Example Applications:
  - Address book.
  - Student record database.
  - Compiler's symbol table.

The Unsorted Map ADT Operations

- Unsorted Map ADT Operations:
  - get(k): if the map M has an entry with key k, return its associated value, v; else, return null.
  - put(k, v): insert entry (k, v) into the map M, keeping k unique; if key k is not already in M, then return null; else, replace and return the old value, v associated with k.
  - remove(k): if the map M has an entry with key k, remove it from M and return its associated value, v; else, return null.
  - size(), isEmpty(): As before.
  - keys(): return an iterable collection of all the keys in M.
  - values(): return an iterable collection of the values of all entries in M.
  - entries(): return an iterable collection of all the key-value entries in M.

Unsorted Map Implementations

- An Unsorted Map can be implemented using four main data structures:
  - An unordered list, (e.g. ArrayList or LinkedList).
  - A search table, (e.g. Sorted ArrayList).
  - A hash table, using bucket array & external chaining.
  - A skip list.

Comparing Unsorted Map Implementations

- All implementations require $O(n)$ space. Assume the following:
  n: The number of entries in the Map,
- The time requirements of all operations are shown in the following table:

Operation Using	get	put	remove	entries	size	isEmpty	clear
Unordered list	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(1)$
Search table	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(1)$
Hash table	$O(1)$ exp. $O(n)$ worst	$O(1)$ exp. $O(n)$ worst	$O(1)$ exp. $O(n)$ worst	$O(n)$	$O(1)$	$O(1)$	$O(1)$
Skip List	$O(\log n)$ exp.	$O(\log n)$ exp.	$O(\log n)$ exp.	$O(n)$	$O(1)$	$O(1)$	$O(1)$

The Sorted Map ADT Operations

- Sorted Map ADT Operations:
  - All the unsorted Map operations as before, and the following operations:
  - first(): Returns the entry with smallest key, or null, if the map is empty.
  - last(): Returns the entry with largest key, or null, if the map is empty.
  - ceiling(k): Returns the entry with the least key greater than or equal to k, or null, if no such entry exists.
  - floor(k): Returns the entry with the greatest key less than or equal to k, or null, if no such entry exists.
  - lower(k): Returns the entry with the greatest key strictly less than k, or null, if no such entry exists.
  - higher(k): Returns the entry with the least key strictly greater than k, or null, if no such entry exists.
  - subMap(k_1, k_2): Returns an iterable collection of all the entries in M with keys greater than or equal to k_1 and strictly less than k_2.

Sorted Map Implementations

- A Sorted Map can be implemented efficiently using three main data structures:
  - A balanced search tree, (e.g. red-black or AVL binary trees); and a B-tree (or its variants) can also be used when the map is too large to reside entirely in main memory.
  - A search table, (e.g. Sorted ArrayList).
  - A skip list.

Comparing Sorted Map Implementations

- All implementations require $O(n)$ space. Assume the following:
  - n: The number of entries in the sorted Map,
  - s: The size of the collection returned by the subMap() operation.
- The time requirements of all operations are shown in the following table:

Operation Using	subMap	get	put	remove	first	last	ceiling, higher	floor, lower	keys, values, entries
Balanced Search Tree	$O(\log n + s)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$		$O(\log n)$		$O(n)$
Search table	$O(\log n + s)$	$O(\log n)$	$O(n)$	$O(n)$	$O(1)$		$O(\log n)$		$O(n)$
Skip List	$O(\log n + s)$ exp.	$O(\log n)$ exp.	$O(\log n)$ exp.	$O(\log n)$ exp.	$O(1)$		$O(\log n)$ exp.		$O(n)$

Empty Slide

Dictionaries

The Dictionary ADT

The Dictionary ADT Definition

- A dictionary, like a map, models a searchable collection of key-value entries, where each key, k, is associated with a corresponding value, v.
  - Unlike a map, Multiple entries with the same key are **allowed**.
  - The keys and values can be of **any object** type.
- There are two dictionary types:
  - **Ordered**,
  - **Unordered**.
- The main operations of a dictionary are for searching, inserting, and deleting entries.
- Example Applications:
  - Log files or audit trails.
  - Language dictionaries.
  - Sorted sets.

Unordered Dictionary Operations

- Operations on an Unordered Dictionary D are:
  - find(k): if dictionary D has any entry with key k, it returns that entry; else, it returns null.
  - insert(k, v): inserts entry (k, v) into dictionary D, and returns the entry created.
  - remove(e): if dictionary D has entry e, it removes that entry from D and returns it; else, it returns null.
  - size(), isEmpty(), clear(): As defined before.
  - findAll(k): returns an iterable collection of all entries in D with keys = k.
  - entries(): returns an iterable collection of all the key-value entries in D.

Unordered Dictionary Implementations

- An Unordered Dictionary can be implemented efficiently using four main data structures:
  - An unordered list, (e.g. ArrayList or LinkedList).
  - A search table, (e.g. Sorted ArrayList).
  - A hash table, using bucket array & external chaining.
  - A skip list.

Comparing Unordered Dictionary Implementations

- All implementations require O(n) space. Assume the following:
  - n: The number of entries in the dictionary,
  - s: The size of the collection returned by operation findAll(k).
- The time requirements of all operations are shown in the following table:

Operation Using	find	findAll	remove	insert	entries	size	isEmpty	clear
Unordered list	O(n)	O(n)	O(n)	O(1)	O(n)	O(1)	O(1)	O(1)
Search table	O(log n)	O(log n + s)	O(n)	O(n)	O(n)	O(1)	O(1)	O(1)
Hash table	O(1) exp. O(n) worst	O(1+s) exp. O(n) worst	O(1) exp. O(n) worst	O(1)	O(n)	O(1)	O(1)	O(1)
Skip List	O(log n) exp.	O(log n + s) exp.	O(log n) exp.		O(n)	O(1)	O(1)	O(1)

Ordered Dictionary Operations

- Operations on an Ordered Dictionary D are:
  - All the operations of the unordered dictionary are the same for the ordered dictionary:
  - find(k), findAll(k), entries(): As defined before.
  - insert(k, v), remove(e): As defined before.
  - size(), isEmpty(), clear(): As defined before.
  - An ordered dictionary can efficiently provide the additional operations:
  - first(): returns an entry in D with the smallest key.
  - last(): returns an entry in D with the largest key.
  - successors(k): returns an iterable collection of all entries in D with keys greater than or equal to k, in non-decreasing order.
  - predecessors(k): returns an iterable collection of all entries in D with keys less than or equal to k, in non-increasing order.

Ordered Dictionary Implementations

- An Ordered Dictionary can be implemented efficiently using three main data structures:
  - A balanced search tree, (e.g. red-black or AVL binary trees); and a B-tree (or its variants) can also be used when the map is too large to reside entirely in main memory.
  - A search table, (e.g. Sorted ArrayList).
  - A skip list.

Comparing Ordered Dictionary Implementations

- All implementations require $O(n)$ space. Assume the following:
  - n: The number of entries in the dictionary,
  - s: The size of the collection returned by operation findAll(k).
- The time requirements of all operations are shown in the following table:

Operation Using	find	findAll	remove	insert	first	last	successor	predecessor
Balanced Search Tree	$O(\log n)$	$O(\log n + s)$	$O(\log n)$		$O(\log n)$		$O(\log n)$	$O(\log n)$
Search table	$O(\log n)$	$O(\log n + s)$	$O(n)$		$O(1)$		$O(1)$	$O(1)$
Skip List	$O(\log n)$ exp.	$O(\log n + s)$ exp.	$O(\log n)$ exp.		$O(1)$		$O(\log n)$ exp.	$O(\log n)$ exp.