Maps

MAPS (CH 10.1)

Presentation for use with the textbook
A map models a searchable collection of key-value pairs \((k,v)\), which we call entries.

- Keys are required to be unique.
- Maps are also known as associative arrays.

- entry’s key serves somewhat like an index into the map, in that it assists the map in efficiently locating the associated entry.

Unlike a standard array, a key of a map need not be numeric, and does not directly designate a position within the structure.

The Map is related to the Set, mathematically, a Map is a set of ordered pairs whose elements are known as the key and the value.

The main operations are for searching, inserting, and deleting items.
MAP PROPERTIES

- You can think of each key as a “mapping” to a particular value
- Keys must be unique, but values need not be unique
- Multiple entries with the same key are not allowed
- A map provides efficient storage and retrieval of information in a table
- A map can have **many-to-one** mapping: (B, Bill), (B2, Bill)

{(J, Jane), (B, Bill), (S, Sam), (B1, Bob), (B2, Bill)}
A **multimap** is similar to a traditional map, in that it associates values with keys; however, in a multimap the same key can be mapped to multiple values.

For example, the index of a book maps a given term to one or more locations at which the term occurs.

**Hash tables** (implemented by a Map or Set) store objects at arbitrary locations and offer an average constant time for insertion, removal, and searching.
Maps

**Map Java Hierarchy**

```
<<interface>>
Map

AbstractMap

HashMap

<<interface>>
SortedMap

NavigableMap

<<interface>>

TreeMap

ConcurrentSkipListMap
```
Java Interface Hierarchy

- `java.util.Map<K,V>`
  - `java.util.SortedMap<K,V>`
    - `java.util.NavigableMap<K,V>`
  - `java.util.Map.Entry<K,V>`
- `java.lang.Iterable<T>`
  - `java.util.Collection<E>`
    - `java.util.List<E>`
    - `java.util.Queue<E>`
      - `java.util.Deque<E>`
    - `java.util.Set<E>`
      - `java.util.SortedSet<E>`
        - `java.util.NavigableSet<E>`
- `java.util.Iterator<E>`
  - `java.util.ListIterator<E>`

http://docs.oracle.com/javase/7/docs/api/java/util/package-tree.html
Maps

JAVA CLASS HIERARCHY

- java.util.AbstractMap<K,V> (implements java.util.Map<K,V>)
    - java.util.LinkedHashMap<K,V> (implements java.util.Map<K,V>)
    - java.util.WeakHashMap<K,V> (implements java.util.Map<K,V>)

- java.util.Dictionary<K,V>
    - java.util.Properties

http://docs.oracle.com/javase/7/docs/api/java/util/package-tree.html
When information about an item is stored in a table, the information should have a unique ID.

A unique ID may or may not be a number.

This unique ID is equivalent to a key.

<table>
<thead>
<tr>
<th>Type of item</th>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>University student</td>
<td>Student ID number</td>
<td>Student name, address, major, grade point average</td>
</tr>
<tr>
<td>Online store customer</td>
<td>E-mail address</td>
<td>Customer name, address, credit card information, shopping cart</td>
</tr>
<tr>
<td>Inventory item</td>
<td>Part ID</td>
<td>Description, quantity, manufacturer, cost, price</td>
</tr>
<tr>
<td>The domain-name system (DNS) maps</td>
<td>host name, such as <a href="http://www.wiley.com">www.wiley.com</a></td>
<td>Internet-Protocol (IP) address such as 208.215.179.146.</td>
</tr>
</tbody>
</table>
Maps

**THE MAP ADT**

A map $M$ supports the following methods:

- **size()**: Returns the number of entries in $M$.
- **isEmpty()**: Returns a boolean indicating whether $M$ is empty.
- **get($k$)**: Returns the value $v$ associated with key $k$, if such an entry exists; otherwise returns null.
- **put($k$, $v$)**: If $M$ does not have an entry with key equal to $k$, then adds entry $(k,v)$ to $M$ and returns null; else, replaces with $v$ the existing value of the entry with key equal to $k$ and returns the old value.
- **remove($k$)**: Removes from $M$ the entry with key equal to $k$, and returns its value; if $M$ has no such entry, then returns null.
- **keySet()**: Returns an iterable collection containing all the keys stored in $M$.
- **values()**: Returns an iterable collection containing all the *values* of entries stored in $M$ (with repetition if multiple keys map to the same value).
- **entrySet()**: Returns an iterable collection containing all the key-value entries in $M$. 
Some implementations of the java.util.Map interface explicitly forbid use of a null value (and null keys, for that matter).

But the textbook implementation allows operations get(k), put(k, v), and remove(k) returns the existing value associated with key k, if the map has such an entry, and otherwise returns null. This introduces ambiguity in an application for which null is allowed as a natural value associated with a key k.
The following statements build a Map object:

```java
Map<String, String> aMap = new HashMap<String, String>();

aMap.put("J", "Jane");
aMap.put("B", "Bill");
aMap.put("S", "Sam");
aMap.put("B1", "Bob");
aMap.put("B2", "Bill");
```
aMap.get("B1")

returns:

"Bob"
aMap.get("Bill")

returns:

null

("Bill" is a value, not a key)
### Example

<table>
<thead>
<tr>
<th>Method</th>
<th>Return Value</th>
<th>Map</th>
</tr>
</thead>
<tbody>
<tr>
<td>isEmpty()</td>
<td>true</td>
<td>{}</td>
</tr>
<tr>
<td>put(5, A)</td>
<td>null</td>
<td>{ (5, A) }</td>
</tr>
<tr>
<td>put(7, B)</td>
<td>null</td>
<td>{ (5, A), (7, B) }</td>
</tr>
<tr>
<td>put(2, C)</td>
<td>null</td>
<td>{ (5, A), (7, B), (2, C) }</td>
</tr>
<tr>
<td>put(8, D)</td>
<td>null</td>
<td>{ (5, A), (7, B), (2, C), (8, D) }</td>
</tr>
<tr>
<td>put(2, E)</td>
<td>C</td>
<td>{ (5, A), (7, B), (2, E) }</td>
</tr>
<tr>
<td>get(7)</td>
<td>B</td>
<td>{ (5, A), (7, B), (2, E), (8, D) }</td>
</tr>
<tr>
<td>get(4)</td>
<td>null</td>
<td>{ (5, A), (7, B), (2, E), (8, D) }</td>
</tr>
<tr>
<td>get(2)</td>
<td>E</td>
<td>{ (5, A), (7, B), (2, E), (8, D) }</td>
</tr>
<tr>
<td>size()</td>
<td>4</td>
<td>{ (5, A), (7, B), (2, E), (8, D) }</td>
</tr>
<tr>
<td>remove(5)</td>
<td>A</td>
<td>{ (7, B), (2, E), (8, D) }</td>
</tr>
<tr>
<td>remove(2)</td>
<td>E</td>
<td>{ (7, B), (8, D) }</td>
</tr>
<tr>
<td>get(2)</td>
<td>null</td>
<td>{ (7, B), (8, D) }</td>
</tr>
<tr>
<td>remove(2)</td>
<td>null</td>
<td>{ (7, B), (8, D) }</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>false</td>
<td>{ (7, B), (8, D) }</td>
</tr>
<tr>
<td>entrySet()</td>
<td>{ (7, B), (8, D) }</td>
<td>{ (7, B), (8, D) }</td>
</tr>
<tr>
<td>keySet()</td>
<td>{ 7, 8 }</td>
<td>{ (7, B), (8, D) }</td>
</tr>
<tr>
<td>values()</td>
<td>{ B, D }</td>
<td>{ (7, B), (8, D) }</td>
</tr>
</tbody>
</table>
APPLICATION: COUNTING WORD FREQUENCIES

- Problem: Counting the number of occurrences of words in a document.
- Using Map as data structure: use words as keys and word counts as values.
  - ChainHashMap class
- Procedure:
  - Scan through the input, considering adjacent alphabetic characters to be words,
  - Convert words to lowercase.
  - For each word found, retrieve its current frequency from the map using the get method (unseen word having frequency zero.)
  - (re)set its frequency to be one more to reflect the current occurrence of the word.
  - After processing the entire input, loop through the entrySet( ) of the map to determine which word has the most occurrences.
IMPLEMENTATION CONSIDERATIONS FOR MAPS AND SETS
Figure 10.2: Our hierarchy of map types (with references to where they are defined).
SIMPLIFIED VERSION OF THE MAP ADT. INTERFACE

Map Interface

```java
public interface Map<K, V> {
    int size();
    boolean isEmpty();
    V get(K key);
    V put(K key, V value);
    V remove(K key);
    Iterable<K> keySet();
    Iterable<V> values();
    Iterable<Entry<K, V>> entrySet();
}
```

K designating the key type;
V designating the value type

Entry Interface

```java
/** Interface for a key-value pair. */
public interface Entry<K, V> {
    K getKey(); // returns the key stored in this entry
    V getValue(); // returns the value stored in this entry
}
```
JAVA IMPLEMENTATION OF ABSTRACTMAP

```java
public abstract class AbstractMap<K,V> implements Map<K,V> {
    public boolean isEmpty() { return size() == 0; }

    // nested MapEntry class
    protected static class MapEntry<K,V> implements Entry<K,V> {
        private K k; // key
        private V v; // value
        public MapEntry(K key, V value) {
            k = key;
            v = value;
        }
        // public methods of the Entry interface
        public K getKey() { return k; }
        public V getValue() { return v; }
        // utilities not exposed as part of the Entry interface
        protected void setKey(K key) { k = key; }
        protected V setValue(V value) {
            V old = v;
            v = value;
            return old;
        }
    }
}
```

- An implementation of the `isEmpty` method, based upon the presumed implementation of the `size` method.

- A nested `MapEntry class` that implements the public `Entry` interface, while providing a composite for storing key-value entries in a map data structure.
Concrete implementations of the **keySet** and **values** methods, based upon an adaption to the **entrySet** method.

In this way, concrete map classes need only implement the **entrySet** method to provide all three forms of iteration.

```java
// Support for public keySet method...
private class KeyIterator implements Iterator<K> {
    private Iterator<Entry<K,V>> entries = entrySet().iterator(); // reuse entrySet
    public boolean hasNext() { return entries.hasNext(); }
    public K next() { return entries.next().getKey(); } // return key!
    public void remove() { throw new UnsupportedOperation(); }
}
private class KeyIterable implements Iterable<K> {
    public Iterator<K> iterator() { return new KeyIterator(); }
    public Iterable<K> keySet() { return new KeyIterable(); }
}
```
// Support for public values method...
private class Valuelterator implements Iterator<V> {
    private Iterator<Entry<K, V>> entries = entrySet().iterator(); // reuse entrySet
    public boolean hasNext() { return entries.hasNext(); }
    public V next() { return entries.next().getValue(); } // return value!
    public void remove() { throw new UnsupportedOperationException(); }
}

private class Valuelterable implements Iterable<V> {
    public Iterator<V> iterator() { return new Valuelterator(); }
    public Iterable<V> values() { return new Valuelterable(); }
}
A SIMPLE UNSORTED MAP IMPLEMENTATION: UNSORTEDTABLEMAP

The use of the AbstractMap class with a very simple concrete implementation of the map ADT that relies on storing key-value pairs in arbitrary order within a Java ArrayList.

```java
public class UnsortedTableMap<K, V> extends AbstractMap<K, V> {
    // Underlying storage for the map of entries. */
    private ArrayList<MapEntry<K, V>> table = new ArrayList<>();

    /** Constructs an initially empty map. */
    public UnsortedTableMap() {
    }

    // private utility
    /** Returns the index of an entry with equal key, or -1 if non
     * private int findIndex(K key) {
     *     int n = table.size();
     *     for (int j=0; j < n; j++)
     *         if (table.get(j).getKey().equals(key))
     *             return j;
     *     return -1;
     *     // special value denotes tl
    }
```

Private `findIndex(key)` method that returns the index at which such an entry is
Private `findIndex(key)` method that returns the index at which such an entry is found, or -1 if no such entry is found by scanning the array to determine whether an entry with key equal to `k` exists.
Unfortunately, the UnsortedTableMap class on the whole is not very efficient. Fundamental methods, get(\(k\)), put(\(k, \nu\)), and remove(\(k\)), takes \(O(n)\) time in the worst case because of the need to scan through the entire list when searching for an existing entry.
Why not use `remove` method of the `ArrayList` class?

- that would result in an unnecessary loop to shift all subsequent entries to the left.

Because the map is unordered, we prefer to fill the vacated cell of the array by relocating the last entry to that location.
Remember that concrete implementations of the `keySet` and `values` methods hasNext(), next() and remove(), are based upon `entrySet` method.
Sorted Maps allows **inexact or range searches** of keys.

Application: Computer system that maintains information about events that have occurred (such as financial transactions), with a **time stamp** (key) marking the **occurrence of each event** (value)
SORTED MAPS INTERFACE

Includes all behaviors of the standard map, plus the following methods:

- **firstEntry( ):** Returns the entry with smallest key value (or null, if the map is empty).
- **lastEntry( ):** Returns the entry with largest key value (or null, if the map is empty).
- **ceilingEntry(k):** Returns the entry with the least key value greater than or equal to \( k \) (or null, if no such entry exists).
- **floorEntry(k):** Returns the entry with the greatest key value less than or equal to \( k \) (or null, if no such entry exists).
- **lowerEntry(k):** Returns the entry with the greatest key value strictly less than \( k \) (or null, if no such entry exists).
- **higherEntry(k):** Returns the entry with the least key value strictly greater than \( k \) (or null if no such entry exists).
- **subMap(k1, k2):** Returns an iteration of all entries with key greater than or equal to \( k1 \), but strictly less than \( k2 \).
**Sorted search table:** Store the map’s entries in an array list $A$ so that they are in increasing order of their keys.

- the sorted search table has a space requirement that is $O(n)$.
- array-based implementation allows us to use the *binary search* algorithm for a variety of efficient operations.
public class SortedTableMap<K,V> extends AbstractSortedMap<K,V> {
    private ArrayList<MapEntry<K,V>> table = new ArrayList<>();
    public SortedTableMap() { super(); }
    public SortedTableMap(Comparator<K> comp) { super(comp); }
    /** Returns the smallest index for range table[low..high] inclusive storing an entry
     * with a key greater than or equal to k (or else index high+1, by convention). */
    private int findIndex(K key, int low, int high) {
        if (high < low) return high + 1; // no entry qualifies
        int mid = (low + high) / 2;
        int comp = compare(key, table.get(mid));
        if (comp == 0)
            return mid; // found exact match
        else if (comp < 0)
            return findIndex(key, low, mid - 1); // answer is left of mid (or possibly mid)
        else
            return findIndex(key, mid + 1, high); // answer is right of mid
    }
    /** Version of findIndex that searches the entire table */
    private int findIndex(K key) { return findIndex(key, 0, table.size() - 1); }
**FINDINDEX METHOD**

**findIndex** method uses the **recursive binary search algorithm**,

- returns the *index* of the leftmost entry in the search range having key greater than or equal to *k*;
- if no entry in the search range has such a key, we return the index just beyond the end of the search range.

⇒ If an entry has the target key, the search returns the index of that entry. (Recall that keys are unique in a map.)
⇒ If the key is absent, the method returns the index at which a new entry with that key would be inserted
SORTEDTABLEMAP IMPLEMENTATION (CONT.)

```java
/** Returns the number of entries in the map. */
public int size() {
    return table.size();
}

/** Returns the value associated with the specified key (or else null). */
public V get(K key) {
    int j = findIndex(key);
    if (j == size() || compare(key, table.get(j)) != 0) return null; // no match
    return table.get(j).getValue();
}

/** Associates the given value with the given key, returning any overridden value. */
public V put(K key, V value) {
    int j = findIndex(key);
    if (j < size() && compare(key, table.get(j)) == 0) return null; // match exists
    table.get(j).setValue(value);
    table.add(j, new MapEntry<K, V>(key, value)); // otherwise new
    return null;
}

/** Removes the entry having key k (if any) and returns its associated value. */
public V remove(K key) {
    int j = findIndex(key);
    if (j == size() || compare(key, table.get(j)) != 0) return null; // no match
    return table.remove(j).getValue();
}
```

Size: $O(1)$
Get: $O(\log n)$
Put: $O(n)$; $O(\log n)$ if map has entry with given key
Remove: $O(n)$
/** Utility returns the entry at index j, or else null if j is out of bounds. */
private Entry<K,V> safeEntry(int j) {
    if (j < 0 || j >= table.size()) return null;
    return table.get(j);
}

/** Returns the entry having the least key (or null if map is empty). */
public Entry<K,V> firstEntry() { return safeEntry(0); }

/** Returns the entry having the greatest key (or null if map is empty). */
public Entry<K,V> lastEntry() { return safeEntry(table.size() - 1); }

/** Returns the entry with least key greater than or equal to given key (if any). */
public Entry<K,V> ceilingEntry(K key) {
    return safeEntry(findIndex(key));
}

/** Returns the entry with greatest key less than or equal to given key (if any). */
public Entry<K,V> floorEntry(K key) {
    int j = findIndex(key);
    if (j == size() || !key.equals(table.get(j).getKey()))
        j--; // look one earlier (unless we had found a perfect match)
    return safeEntry(j);

/** Returns the entry with greatest key strictly less than given key (if any). */
public Entry<K,V> lowerEntry(K key) {
    return safeEntry(findIndex(key) - 1); // go strictly before the ceiling entry
}

/** Returns the entry with least key strictly greater than given key (if any). */
public Entry<K,V> higherEntry(K key) {
    int j = findIndex(key);
    if (j < size() && key.equals(table.get(j).getKey()))
        j++; // go past exact match
    return safeEntry(j);
}
subMap: \(O(s+\log n)\) where \(s\) items are reported:

- It begins with a binary search to find the first item within the range (if any).
- After that, it executes a loop that takes \(O(1)\) time per iteration to gather subsequent values until reaching the end of the range.
# Analysis of Our SortedTableMap

<table>
<thead>
<tr>
<th>Method</th>
<th>Running Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>get</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>put</td>
<td>$O(n)$; $O(\log n)$ if map has entry with given key</td>
</tr>
<tr>
<td>remove</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>firstEntry, lastEntry</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>ceilingEntry, floorEntry, lowerEntry, higherEntry</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>subMap</td>
<td>$O(s + \log n)$ where $s$ items are reported</td>
</tr>
<tr>
<td>entrySet, keySet, values</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>
**Keys** are Flight objects that contain fields corresponding to four parameters. 
\( k = (\text{origin}, \text{destination}, \text{date}, \text{time}) \).

**Value:** Additional information about a flight, such as the flight number, the number of seats still available in first (F) and coach (Y) class, the flight duration, and the fare.

Searching for a flight requires inexact searching.

**EX:** Given a user query key \( k \), we could call ceilingEntry(\( k \)) to return the first flight between the desired cities, having a departure date and time matching the desired query or later.

**EX:** With well-constructed keys, we could use subMap(\( k_1, k_2 \)) to find all flights within a given range of times. For example, if \( k_1 = (\text{ORD}, \text{PVD}, 05\text{May}, 09:30) \), and \( k_2 = (\text{ORD}, \text{PVD}, 05\text{May}, 20:00) \).

\[
\begin{align*}
(\text{ORD, PVD, 05May, 09:53}) & : (\text{AA 1840, F5, Y15, 02:05, $251}), \\
(\text{ORD, PVD, 05May, 13:29}) & : (\text{AA 600, F2, Y0, 02:16, $713}), \\
(\text{ORD, PVD, 05May, 17:39}) & : (\text{AA 416, F3, Y9, 02:09, $365}), \\
(\text{ORD, PVD, 05May, 19:50}) & : (\text{AA 1828, F9, Y25, 02:13, $186})
\end{align*}
\]
APPLICATIONS OF SORTED MAPS: MAXIMA SETS

Example problem: Allow one to query a car database to find the fastest car a person can possibly afford by maintaining the set of maxima of a collection of cost-performance pairs

key-value pair: (cost, speed)

we would like to
* add new pairs to this collection (new cars),
* query this collection for a given dollar amount, \( d \), to find the fastest car that costs no more than \( d \) dollars.

Note: point \( p \) is strictly better than points \( c, d, \) and \( e \), but may be better or worse than points \( a, b, f, g, \) and \( h \),
A cost-performance pair \((a, b)\) *dominates* pair \((c, d)\) \(!= (a, b)\) if \(a \leq c\) and \(b \geq d\), (first pair has no greater cost and at least as good performance.) A pair \((a,b)\) is called a *maximum* pair if it is not dominated by any other pair.

store the set of maxima pairs in a sorted map

Implement operations
* **add**\((c, \rho)\), which adds a new cost-performance entry \((c, \rho)\),
* **best**\((c)\), which returns the entry having best performance of those with cost at most \(c\).
MAINTAINING A MAXIMA SET WITH A SORTED MAP

```java
1. /** Maintains a database of maximal (cost, performance) pairs. */
2. public class CostPerformanceDatabase {
3.     
4.     SortedMap<Integer, Integer> map = new SortedTableMap<>();
5.     
6.     /** Constructs an initially empty database. */
7.     public CostPerformanceDatabase() {
8.         
9.     /** Returns the (cost, performance) entry with largest cost not exceeding c.
10.     * (or null if no entry exist with cost c or less).
11.     */
12.     public Entry<Integer, Integer> best(int cost) {
13.         return map.floorEntry(cost);
14.     }
15. }
16. ```
**Add a new entry with given cost \( c \) and performance \( p \).**

```java
/** Add a new entry with given cost \( c \) and performance \( p \). */
public void add(int c, int p) {
    Entry<Integer, Integer> other = map.floorEntry(c); // other is at least as cheap
    if (other != null && other.getValue() >= p) // if its performance is as good,
        return; // (c, p) is dominated, so ignore
    map.put(c, p); // else, add (c, p) to database
    // and now remove any entries that are dominated by the new one
    other = map.higherEntry(c); // other is more expensive than \( c \)
    while (other != null && other.getValue() <= p) { // if not better performance
        map.remove(other.getKey()); // remove the other entry
        other = map.higherEntry(c);
    }
}
```

\( O(n) \) worst-case running time.