SUNY Korea

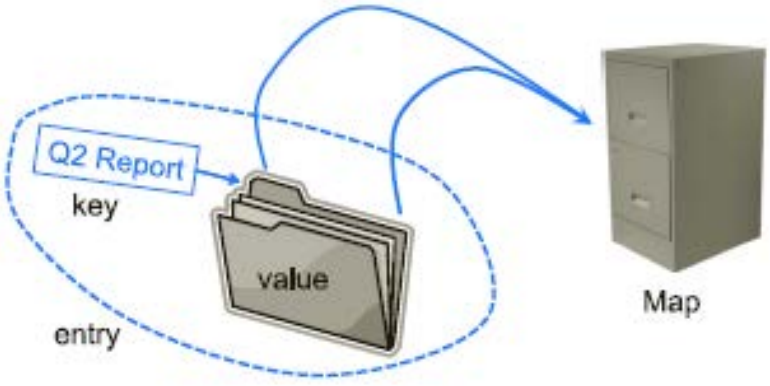
Stony Brook University

MAPS (CH 10.1)

Presentation for use with the textbook

1. Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014
2. Data Structures Abstraction and Design Using Java, 2nd Edition by Elliot B. Koffman & Paul A. T. Wolfgang, Wiley, 2010

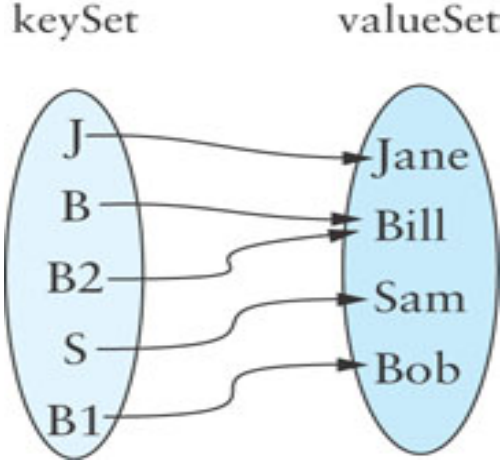
MAPS



- 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 is 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`)

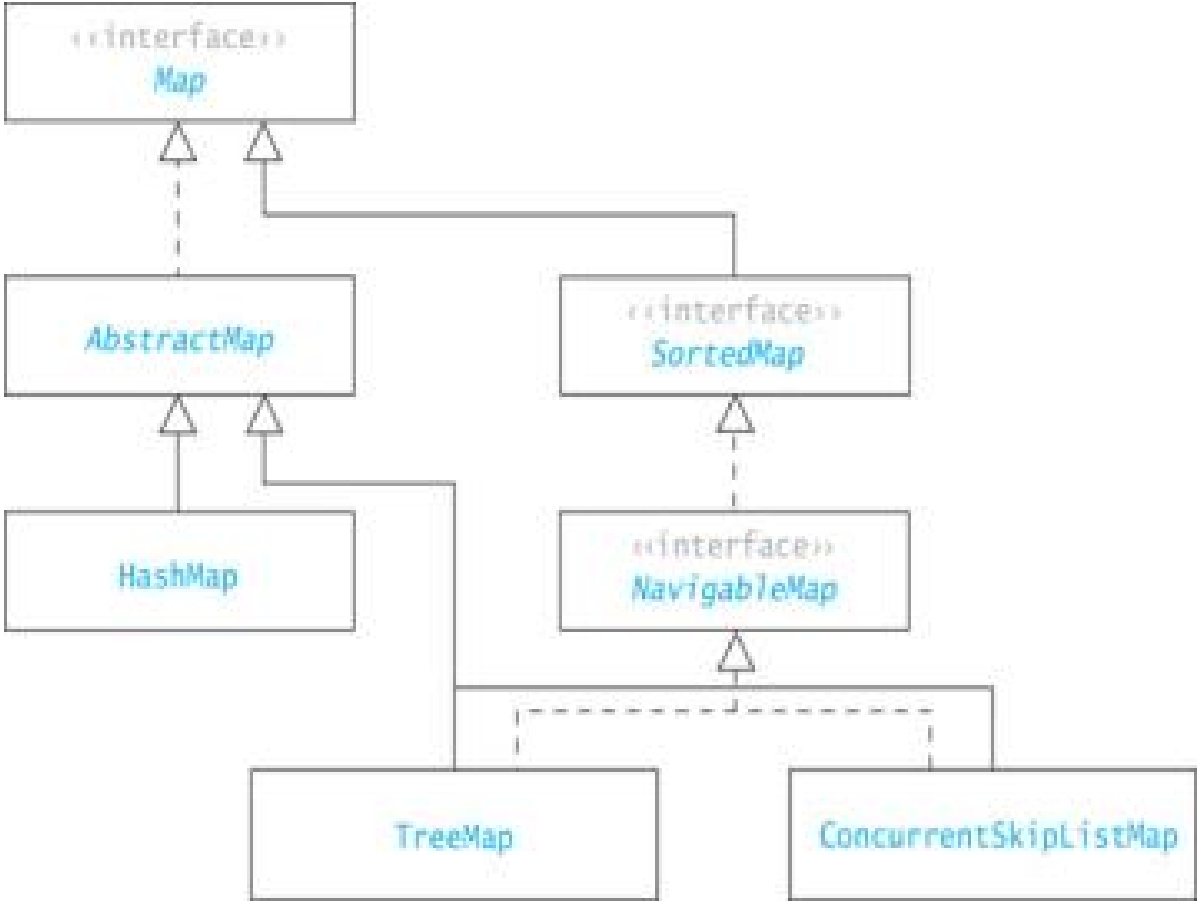



```
{(J, Jane), (B, Bill),
 (S, Sam), (B1, Bob),
     (B2,  Bill)}
```

DEFINITIONS

- 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

MAP JAVA HIERARCHY



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

JAVA CLASS HIERARCHY

- java.util.**AbstractMap**<K,V> (implements java.util.Map<K,V>)
  - java.util.**EnumMap**<K,V> (implements java.lang.Cloneable, java.io.Serializable)
  - java.util.**HashMap**<K,V> (implements java.lang.Cloneable, java.util.Map<K,V>, java.io.Serializable)
    - java.util.**LinkedHashMap**<K,V> (implements java.util.Map<K,V>)
  - java.util.**IdentityHashMap**<K,V> (implements java.lang.Cloneable, java.util.Map<K,V>, java.io.Serializable)
  - java.util.**TreeMap**<K,V> (implements java.lang.Cloneable, java.util.NavigableMap<K,V>, java.io.Serializabl
  - java.util.**WeakHashMap**<K,V> (implements java.util.Map<K,V>)
- java.util.**AbstractMap.SimpleEntry**<K,V> (implements java.util.Map.Entry<K,V>, java.io.Serializable)
- java.util.**AbstractMap.SimpleImmutableEntry**<K,V> (implements java.util.Map.Entry<K,V>, java.io.Serializable)

- java.util.**Dictionary**<K,V>
  - java.util.**Hashtable**<K,V> (implements java.lang.Cloneable, java.util.Map<K,V>, java.io.Serializable)
    - java.util.**Properties**

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

MAPS AND THE MAP INTERFACE(CONT.)

- When information about an item is stored in a table, the informati on should have a unique ID
- A unique ID may or may not be a number
- This unique ID is equivalent to a key

Type of item	Key	Value
University student	Student ID number	Student name, address, major, grade point average
Online store customer	E-mail address	Customer name, address, cred it card information, shopping c art
Inventory item	Part ID	Description, quantity, manufac turer, cost, price
The domain-name system (DNS) maps	host name, such as www.wiley.com,	Internet-Protocol (IP) address such as 208.215.179.146.

THE MAP ADT

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*.

MAPS IN THE JAVA.UTIL PACKAGE

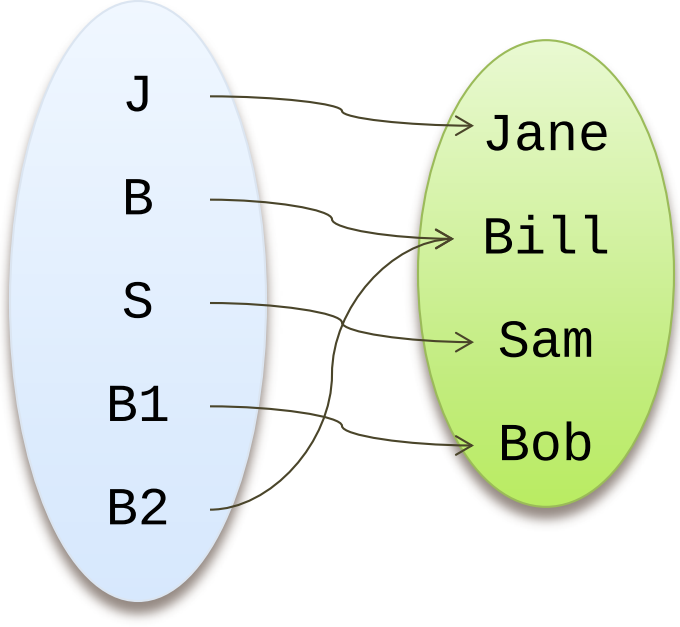
- Some implementations of the java.util.Map interface explicitly forbid use of a null value (and null keys, for that matter).
- But the text book implementation allows operations get(*k*), put(*k*, *v*), and remove(*k*) returns the existing value associated with key *k*, if the map has such an entry, and otherwise returns null.
  - Introduces ambiguity in an application for which null is allowed as a natural value associated with a key *k*.

MAP INTERFACE (CONT.)

- The following statements build a Map object:

```
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");
```

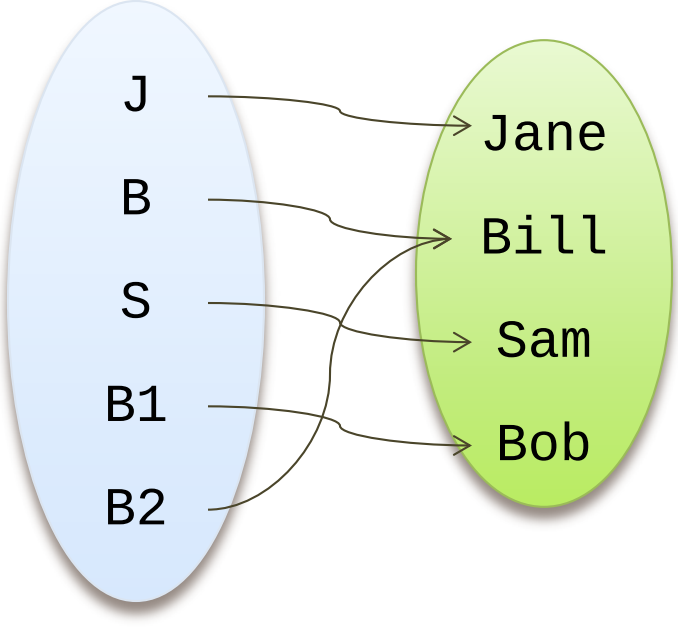


MAP INTERFACE (CONT.)

```
aMap.get("B1")
```

returns:

"Bob"



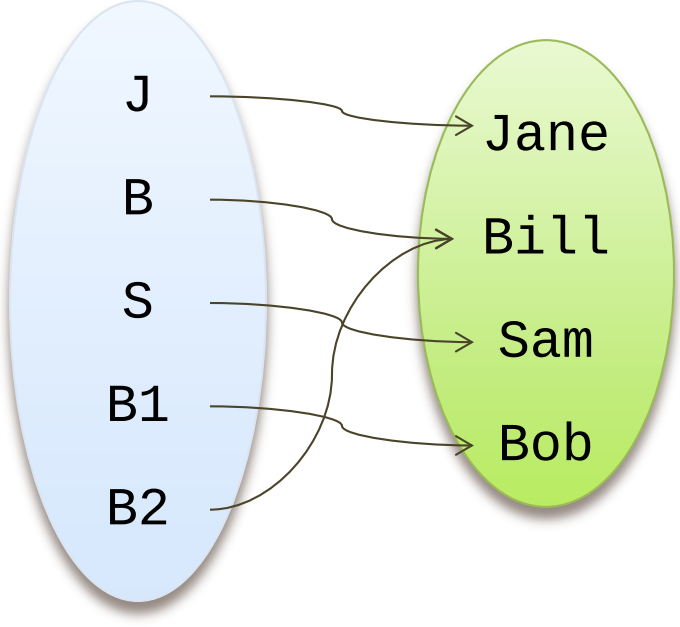
MAP INTERFACE (CONT.)

```
aMap.get("Bill")
```

returns:

```
null
```

("`Bill`" is a value, not a key)



EXAMPLE

Method	*Return Value*	*Map*
isEmpty()	true	$\{\}$
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	$\{(7,B),(8,D)\}$
remove(2)	null	$\{(7,B),(8,D)\}$
isEmpty()	false	$\{(7,B),(8,D)\}$
entrySet()	$\{(7,B),(8,D)\}$	$\{(7,B),(8,D)\}$
keySet()	$\{7,8\}$	$\{(7,B),(8,D)\}$
values()	$\{B,D\}$	$\{(7,B),(8,D)\}$

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

HIERARCHY OF MAP TYPES IN THE TEXTBOOK

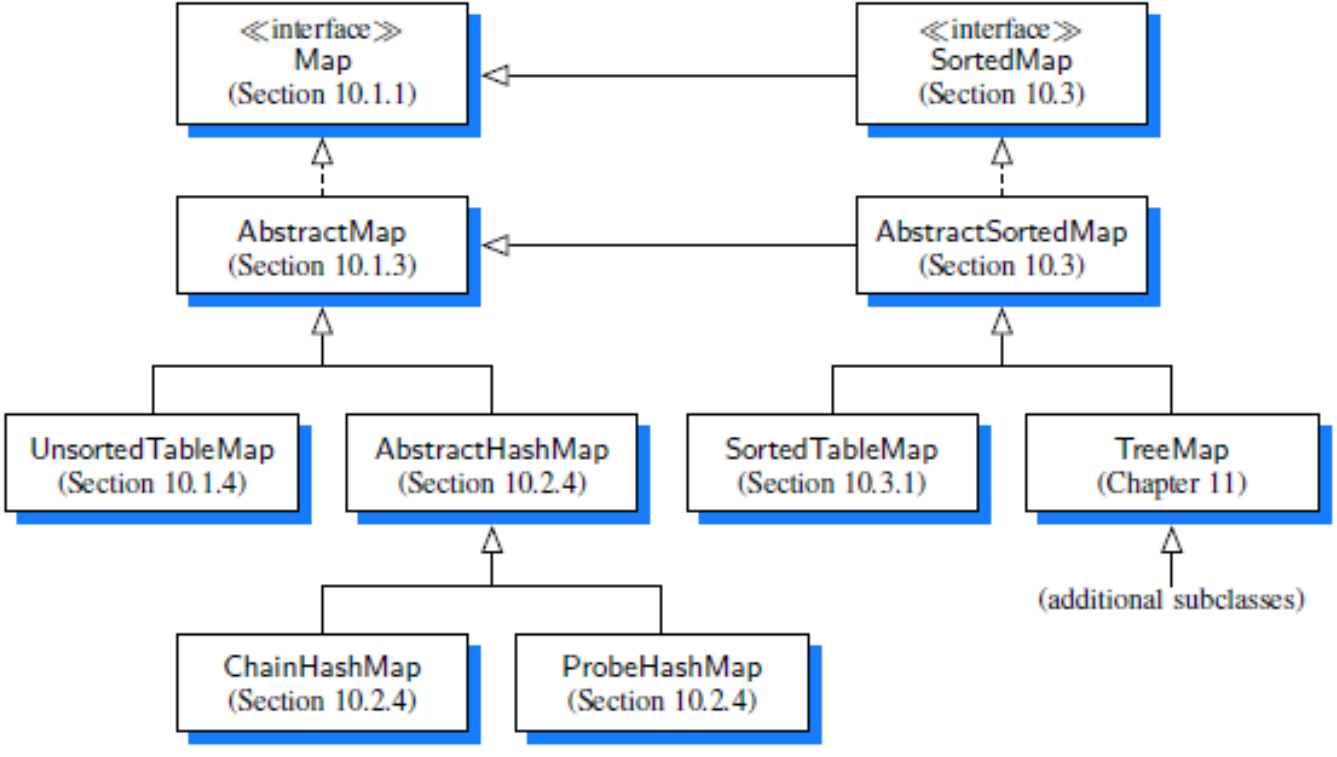


Figure 10.2: Our hierarchy of map types (with references to where they are defined).

SIMPLIFIED VERSION OF THE MAP ADT. INTERFACE

Map Interface

```
1  public interface Map<K,V> {
2    int size( );
3    boolean isEmpty( );
4    V get(K key);
5    V put(K key, V value);
6    V remove(K key);
7    Iterable<K> keySet( );
8    Iterable<V> values( );
9    Iterable<Entry<K,V>> entrySet( );
10 }
```

K designating the key type;
V designating the value type

Entry Interface

```
1  /** Interface for a key-value pair. */
2  public interface Entry<K,V> {
3    K getKey( );                    // returns the key stored in this entry
4    V getValue( );                  // returns the value stored in this entry
5  }
```

JAVA IMPLEMENTATION OF ABSTRACTMAP

```
1  public abstract class AbstractMap<K,V> implements Map<K,V> {
2    public boolean isEmpty( ) { return size( ) == 0; }
3    //--------------- nested MapEntry class ---------------
4    protected static class MapEntry<K,V> implements Entry<K,V> {
5      private K k;    // key
6      private V v;    // value
7      public MapEntry(K key, V value) {
8        k = key;
9        v = value;
10     }
11     // public methods of the Entry interface
12     public K getKey( ) { return k; }
13     public V getValue( ) { return v; }
14     // utilities not exposed as part of the Entry interface
15     protected void setKey(K key) { k = key; }
16     protected V setValue(V value) {
17       V old = v;
18       v = value;
19       return old;
20     }
21   } //---------- end of nested MapEntry class ----------
22
```

- 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.

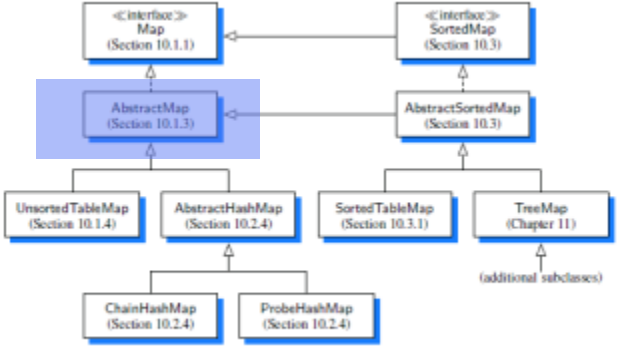


Figure 10.2: Our hierarchy of map types (with references to where they are defined).

JAVA IMPLEMENTATION OF ABSTRACTMAP (CONT.)

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.

```
23    // Support for public keySet method...
24    private class KeyIterator implements Iterator<K> {
25      private Iterator<Entry<K,V>> entries = entrySet().iterator(); // reuse entrySet
26      public boolean hasNext() { return entries.hasNext(); }
27      public K next() { return entries.next().getKey(); }              // return key!
28      public void remove() { throw new UnsupportedOperationException(); }
29    }
30    private class KeyIterable implements Iterable<K> {
31      public Iterator<K> iterator() { return new KeyIterator(); }
32    }
33    public Iterable<K> keySet() { return new KeyIterable(); }
34
```

JAVA IMPLEMENTATION OF ABSTRACTMAP (CONT.)

```
34
35    // Support for public values method...
36    private class ValueIterator implements Iterator<V> {
37      private Iterator<Entry<K,V>> entries = entrySet().iterator(); // reuse entrySet
38      public boolean hasNext() { return entries.hasNext(); }
39      public V next() { return entries.next().getValue(); }          // return value!
40      public void remove() { throw new UnsupportedOperationException(); }
41    }
42    private class ValueIterable implements Iterable<V> {
43      public Iterator<V> iterator() { return new ValueIterator(); }
44    }
45    public Iterable<V> values() { return new ValueIterable(); }
46  }
```

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.**

```
1  public class UnsortedTableMap<K,V> extends AbstractMap<K,V> {
2    /** Underlying storage for the map of entries. */
3    private ArrayList<MapEntry<K,V>> table = new ArrayList<>();
4
5    /** Constructs an initially empty map. */
6    public UnsortedTableMap() { }
7
8    // private utility
9    /** Returns the index of an entry with equal key, or -1 if non
10   private int findIndex(K key) {
11     int n = table.size();
12     for (int j=0; j < n; j++)
13       if (table.get(j).getKey().equals(key))
14         return j;
15     return -1;                          // special value denotes tl
16   }
```

Private **findIndex(key)** method that returns the index at which such an entry is

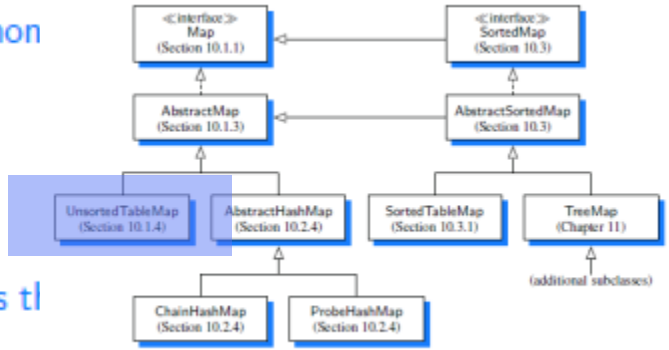


Figure 10.2: Our hierarchy of map types (with references to where they are defined).

UNSORTEDTABLEMAP

```
1  public class UnsortedTableMap<K,V> extends AbstractMap<K,V> {
2    /** Underlying storage for the map of entries. */
3    private ArrayList<MapEntry<K,V>> table = new ArrayList<>();
4
5    /** Constructs an initially empty map. */
6    public UnsortedTableMap() { }
7
8    // private utility
9    /** Returns the index of an entry with equal key, or -1 if none found. */
10   private int findIndex(K key) {
11     int n = table.size();
12     for (int j=0; j < n; j++)
13       if (table.get(j).getKey().equals(key))
14         return j;
15     return -1;                          // special value denotes that key was not found
16   }
```

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.

UNSORTEDTABLEMAP (CONT.)

```
17    /** Returns the number of entries in the map. */
18    public int size( ) { return table.size( ); }
19    /** Returns the value associated with the specified key (or else null). */
20    public V get(K key) {
21      int j = findIndex(key);
22      if (j == -1) return null;                          // not found
23      return table.get(j).getValue( );
24    }
25    /** Associates given value with given key, replacing a previous value (if any). */
26    public V put(K key, V value) {
27      int j = findIndex(key);
28      if (j == -1) {
29        table.add(new MapEntry<>(key, value));           // add new entry
30        return null;
31      } else                                             // key already exists
32        return table.get(j).setValue(value);             // replaced value is returned
33    }
```

Unfortunately, the UnsortedTableMap class on the whole is not very efficient. Fundamental methods, get(*k*), put(*k*, *v*), 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.

UNSORTEDTABLEMAP (CONT.)

```
34    /** Removes the entry with the specified key (if any) and returns its value. */
35    public V remove(K key) {
36      int j = findIndex(key);
37      int n = size( );
38      if (j == −1) return null;                          // not found
39      V answer = table.get(j).getValue( );
40      if (j != n − 1)
41        table.set(j, table.get(n−1));    // relocate last entry to 'hole' created by removal
42      table.remove(n−1);                                 // remove last entry of table
43      return answer;
44    }
```

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.

UNSORTEDTABLEMAP (CONT.)

```
45    // Support for public entrySet method...
46    private class EntryIterator implements Iterator<Entry<K,V>> {
47      private int j=0;
48      public boolean hasNext( ) { return j < table.size( ); }
49      public Entry<K,V> next( ) {
50        if (j == table.size( )) throw new NoSuchElementException( );
51        return table.get(j++);
52      }
53      public void remove( ) { throw new UnsupportedOperationException( ); }
54    }
55    private class EntryIterable implements Iterable<Entry<K,V>> {
56      public Iterator<Entry<K,V>> iterator( ) { return new EntryIterator( ); }
57    }
58    /** Returns an iterable collection of all key-value entries of the map. */
59    public Iterable<Entry<K,V>> entrySet( ) { return new EntryIterable( ); }
60  }
```

Remember that concrete implementations of the **keySet** and **values** methods hasNext(), next() and remove(), are based upon **entrySet** method.

SORTED MAPS ADT

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)

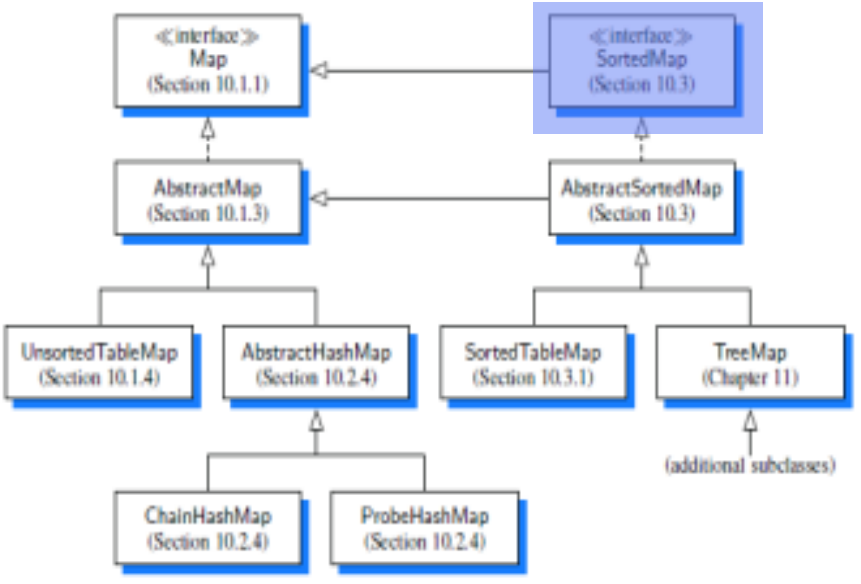


Figure 10.2: Our hierarchy of map types (with references to where they are defined).

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(*k*1, *k*2): Returns an iteration of all entries with key greater than or equal to *k*1, but strictly less than *k*2.

SIMPLE IMPLEMENTATION OF A SORTED MAP

Sorted search table: Store the map's entries in an array list A so that they are in increasing order of their keys.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37

Figure 10.8: Realization of a map by means of a sorted search table. We show only the keys for this map, so as to highlight their ordering.

- 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.

SORTEDTABLEMAP IMPLEMENTATION (CONT.)

```
1  public class SortedTableMap<K,V> extends AbstractSortedMap<K,V> {
2    private ArrayList<MapEntry<K,V>> table = new ArrayList<>();
3    public SortedTableMap( ) { super( ); }
4    public SortedTableMap(Comparator<K> comp) { super(comp); }
5    /** Returns the smallest index for range table[low..high] inclusive storing an entry
6        with a key greater than or equal to k (or else index high+1, by convention). */
7    private int findIndex(K key, int low, int high) {
8      if (high < low) return high + 1;          // no entry qualifies
9      int mid = (low + high) / 2;
10     int comp = compare(key, table.get(mid));
11     if (comp == 0)
12       return mid;                             // found exact match
13     else if (comp < 0)
14       return findIndex(key, low, mid − 1);  // answer is left of mid (or possibly mid)
15     else
16       return findIndex(key, mid + 1, high); // answer is right of mid
17   }
18   /** Version of findIndex that searches the entire table */
19   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.)

```
20    /** Returns the number of entries in the map. */
21    public int size( ) { return table.size( ); }
22    /** Returns the value associated with the specified key (or else null). */
23    public V get(K key) {
24      int j = findIndex(key);
25      if (j == size( ) || compare(key, table.get(j)) != 0) return null;   // no match
26      return table.get(j).getValue( );
27    }
28    /** Associates the given value with the given key, returning any overridden value.*/
29    public V put(K key, V value) {
30      int j = findIndex(key);
31      if (j < size( ) && compare(key, table.get(j)) == 0)                 // match exists
32        return table.get(j).setValue(value);
33      table.add(j, new MapEntry<K,V>(key,value));                         // otherwise new
34      return null;
35    }
36    /** Removes the entry having key k (if any) and returns its associated value. */
37    public V remove(K key) {
38      int j = findIndex(key);
39      if (j == size( ) || compare(key, table.get(j)) != 0) return null;   // no match
40      return table.remove(j).getValue( );
41    }
```

Size: $O(1)$

Get: $O(\log n)$

Put: $O(n)$; $O(\log n)$ if map has entry with given key

Remove: $O(n)$

SORTEDTABLEMAP IMPLEMENTATION (CONT.)

```
42    /** Utility returns the entry at index j, or else null if j is out of bounds. */
43    private Entry<K,V> safeEntry(int j) {
44      if (j < 0 || j >= table.size( )) return null;
45      return table.get(j);
46    }
47    /** Returns the entry having the least key (or null if map is empty). */
48    public Entry<K,V> firstEntry( ) { return safeEntry(0); }
49    /** Returns the entry having the greatest key (or null if map is empty). */
50    public Entry<K,V> lastEntry( ) { return safeEntry(table.size( )−1); }
51    /** Returns the entry with least key greater than or equal to given key (if any). */
52    public Entry<K,V> ceilingEntry(K key) {
53      return safeEntry(findIndex(key));
54    }
55    /** Returns the entry with greatest key less than or equal to given key (if any). */
56    public Entry<K,V> floorEntry(K key) {
57      int j = findIndex(key);
58      if (j == size( ) || ! key.equals(table.get(j).getKey( )))
59        j−−;     // look one earlier (unless we had found a perfect match)
60      return safeEntry(j);
61    }
```

firstEntry, lastEntry $O(1)$

ceilingEntry, floorEntry, $O(\log n)$

SORTEDTABLEMAP IMPLEMENTATION (CONT.)

```
62    /** Returns the entry with greatest key strictly less than given key (if any). */
63    public Entry<K,V> lowerEntry(K key) {
64      return safeEntry(findIndex(key) − 1);       // go strictly before the ceiling entry
65    }
66    public Entry<K,V> higherEntry(K key) {
67    /** Returns the entry with least key strictly greater than given key (if any). */
68      int j = findIndex(key);
69      if (j < size( ) && key.equals(table.get(j).getKey( )))
70        j++;     // go past exact match
71      return safeEntry(j);
72    }
```

lowerEntry, higherEntry
$O(\log n)$

SORTEDTABLEMAP IMPLEMENTATION (CONT)

entrySet, keySet, values: $O(n)$

```
73    // support for snapshot iterators for entrySet() and subMap() follow
74    private Iterable<Entry<K,V>> snapshot(int startIndex, K stop) {
75      ArrayList<Entry<K,V>> buffer = new ArrayList<>();
76      int j = startIndex;
77      while (j < table.size() && (stop == null || compare(stop, table.get(j)) > 0))
78        buffer.add(table.get(j++));
79      return buffer;
80    }
81    public Iterable<Entry<K,V>> entrySet() { return snapshot(0, null); }
82    public Iterable<Entry<K,V>> subMap(K fromKey, K toKey) {
83      return snapshot(findIndex(fromKey), toKey);
84    }
85  }
```

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

Method	Running Time
size	$O(1)$
get	$O(\log n)$
put	$O(n)$; $O(\log n)$ if map has entry with given key
remove	$O(n)$
firstEntry, lastEntry	$O(1)$
ceilingEntry, floorEntry, lowerEntry, higherEntry	$O(\log n)$
subMap	$O(s + \log n)$ where s items are reported
entrySet, keySet, values	$O(n)$

APPLICATIONS OF SORTED MAPS: FLIGHT DATABASES

Keys are Flight objects that contain fields corresponding to four parameters. *k* = (origin, destination, date, 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 = (ORD, PVD, 05May, 09:30), and *k*2 = (ORD. PVD. 05May. 20:00).

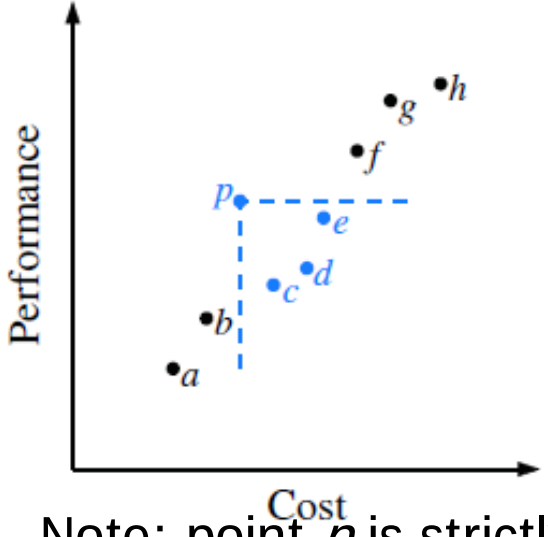
(ORD, PVD, 05May, 09:53) : (AA 1840, F5, Y15, 02:05, $251),
(ORD, PVD, 05May, 13:29) : (AA 600, F2, Y0, 02:16, $713),
(ORD, PVD, 05May, 17:39) : (AA 416, F3, Y9, 02:09, $365),
(ORD, PVD, 05May, 19:50) : (AA 1828, F9, Y25, 02:13, $186)

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*,

MAINTAINING A MAXIMA SET WITH A SORTED MAP

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*, *p*), which adds a new cost-performance entry (*c*, *p*),
***best**(*c*), which returns the entry having best performance of those with cost at most *c*.

MAINTAINING A MAXIMA SET WITH A SORTED MAP

```
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
```

MAINTAINING A MAXIMA SET WITH A SORTED MAP (CONT)

```
16    /** Add a new entry with given cost c and performance p. */
17    public void add(int c, int p) {
18      Entry<Integer,Integer> other = map.floorEntry(c); // other is at least as cheap
19      if (other != null && other.getValue( ) >= p)    // if its performance is as good,
20        return;                                       // (c,p) is dominated, so ignore
21      map.put(c, p);                                  // else, add (c,p) to database
22      // and now remove any entries that are dominated by the new one
23      other = map.higherEntry(c);                     // other is more expensive than c
24      while (other != null && other.getValue( ) <= p) {  // if not better performance
25        map.remove(other.getKey( ));                  // remove the other entry
26        other = map.higherEntry(c);
27      }
28    }
29  }
```

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