

Presentation for use with the textbook **Data Structures and Algorithms in Java, 6th edition**, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Maps



Maps

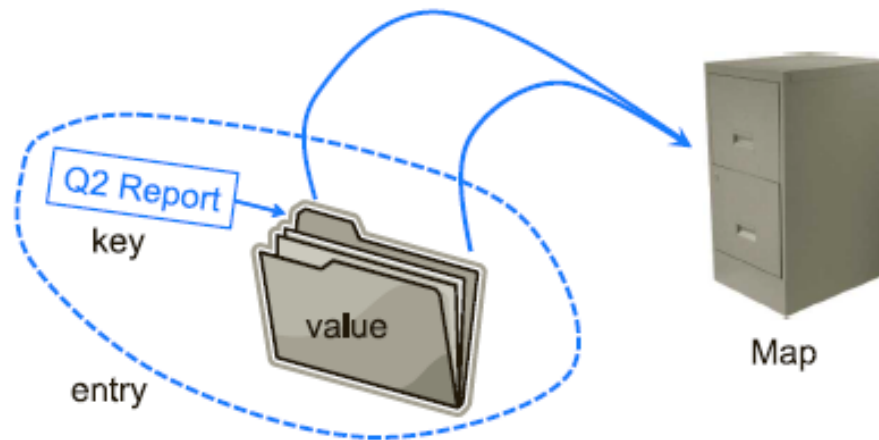


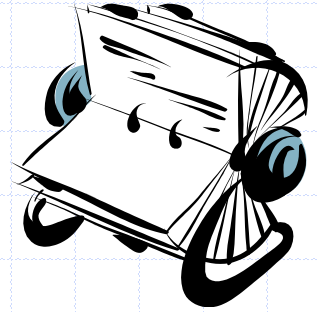
Figure 10.1: A conceptual illustration of the map ADT. Keys (labels) are assigned to values (folders) by a user. The resulting entries (labeled folders) are inserted into the map (file cabinet). The keys can be used later to retrieve or remove values.



Maps

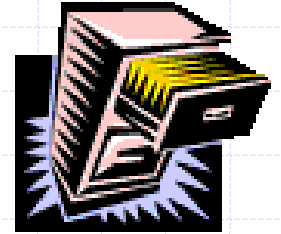
- ❑ A map models a searchable collection of key-value entries
- ❑ The main operations of a map are for searching, inserting, and deleting items
- ❑ Multiple entries with the same key are **not** allowed
- ❑ Applications:
 - address book
 - student-record database

Maps



- ❑ Maps are also known as associative arrays, because the entry's key serves as an index into the map
- ❑ Key assists the map in efficiently locating the associated entry.
- ❑ However, unlike a standard array, a key of a map need not be numeric, and it does not directly designate a position within the structure.

The Map ADT



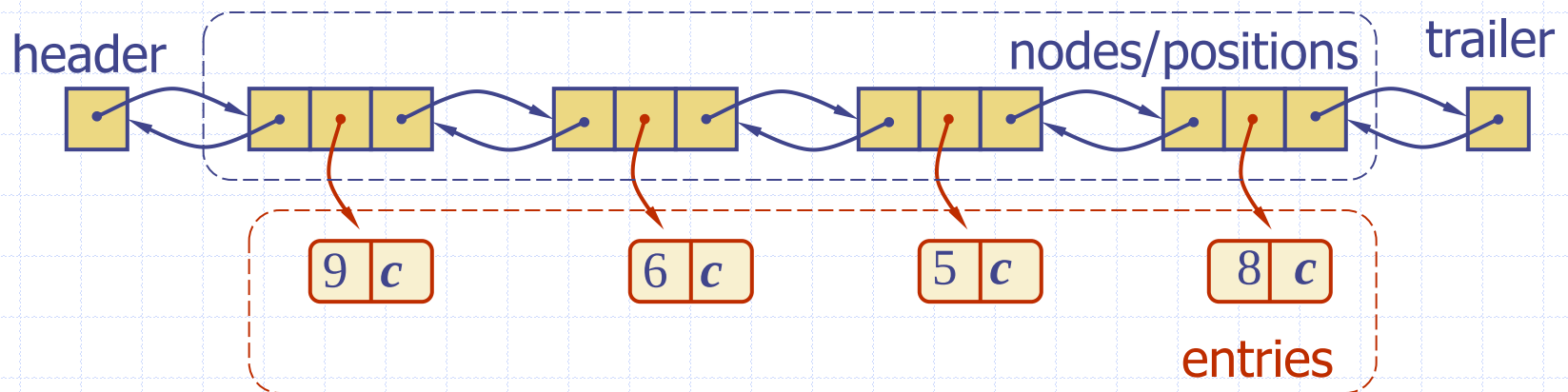
- ❑ **get(k)**: if the map M has an entry with key k , return its associated value; else, return null
- ❑ **put(k, v)**: insert entry (k, v) into the map M ; if key k is not already in M , then return null; else, return old value associated with k
- ❑ **remove(k)**: if the map M has an entry with key k , remove it from M and return its associated value; else, return null
- ❑ **size()**, **isEmpty()**
- ❑ **entrySet()**: return an iterable collection of the entries in M
- ❑ **keySet()**: return an iterable collection 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)

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 doublylinked 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() { the next position in B }

if p.element().getKey() = k **then**

return p.element().getValue()

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().getKey() = k **then**

 t = p.element().getValue()

 S.set(p,(k,v))

return t {return the old value}

S.addLast((k,v))

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

return null { there was no 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().getKey() = k **then**

 t = p.element().getValue()

 S.remove(p)

 n = n - 1

return t

return null

{decrement number of entries}

{return the removed value}

{there is no entry with key equal to k}

Performance of a List-Based Map

□ Performance:

- **put** takes $O(n)$, however it could be $O(1)$ 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)

A Java Interface for the Map ADT

```
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 }
```

Code Fragment 10.1: Java interface for our simplified version of the map ADT.

Application: Counting Word Frequencies

- ❑ This is a standard task when performing a statistical analysis of a document, for example, when categorizing an email or news article.
- ❑ A map is an ideal data structure to use here, for we can use words as keys and word counts as values.
- ❑ We show such an application in Code Fragment 10.2.

```

1  /** A program that counts words in a document, printing the most frequent. */
2  public class WordCount {
3      public static void main(String[] args) {
4          Map<String,Integer> freq = new ChainHashMap<>(); // or any concrete map
5          // scan input for words, using all nonletters as delimiters
6          Scanner doc = new Scanner(System.in).useDelimiter("[^a-zA-Z]+");
7          while (doc.hasNext()) {
8              String word = doc.next().toLowerCase(); // convert next word to lowercase
9              Integer count = freq.get(word); // get the previous count for this word
10             if (count == null)
11                 count = 0; // if not in map, previous count is zero
12             freq.put(word, 1 + count); // (re)assign new count for this word
13         }
14         int maxCount = 0;
15         String maxWord = "no word";
16         for (Entry<String,Integer> ent : freq.entrySet()) // find max-count word
17             if (ent.getValue() > maxCount) {
18                 maxWord = ent.getKey();
19                 maxCount = ent.getValue();
20             }
21         System.out.print("The most frequent word is '" + maxWord);
22         System.out.println("' with " + maxCount + " occurrences.");
23     }
24 }

```

Code Fragment 10.2: A program for counting word frequencies in a document, printing the most frequent word. The document is parsed using the Scanner class, for which we change the delimiter for separating tokens from whitespace to any nonletter. We also convert words to lowercase.

An AbstractMap Base Class

- ❑ AbstractMap base class provides functionality that is shared by all of map implementations.
- ❑ The base class provides the following support:
 - ❑ An implementation of the isEmpty method, based upon implementation of the size method.
 - ❑ A nested MapEntry class that implements the public Entry interface.
 - ❑ 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.

An AbstractMap Base Class

```
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 AbstractMap Base Class

```
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
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 }
```

Code Fragment 10.3: Implementation of the AbstractMap base class.

Simple Unsorted Map Implementation

- We use 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.
- The presentation of such an UnsortedTableMap class is given in Code Fragments 10.4 and 10.5.

Simple Unsorted Map Implementation

- ❑ Each of the fundamental methods `get(k)`, `put(k, v)`, and `remove(k)` requires an initial scan of the array to determine whether an entry with key equal to `k` exists.
- ❑ For this reason, we provide a nonpublic utility, `findIndex(key)`, that returns the index at which such an entry is found.
- ❑ One subtlety is implementation of remove method.
- ❑ Although we could use the remove method, 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. Such an update step runs in constant time.

Simple Unsorted Map Implementation- Performance

- ❑ UnsortedTableMap class is not very efficient.
- ❑ On a map with n entries, each of the fundamental methods takes $O(n)$ time in the worst case because of the need to scan through the entire list when searching for an existing entry.

Simple Unsorted Map Implementation

```
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    }
```

Code Fragment 10.4: An implementation of a map using a Java ArrayList as an unsorted table. (Continues in Code Fragment 10.5.) The parent class AbstractMap is given in Code Fragment 10.3.

Simple Unsorted Map Implementation

```
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  }
```

Code Fragment 10.5: An implementation of a map using a Java ArrayList as an unsorted table (continued from Code Fragment 10.4).

Simple Unsorted Map Implementation

```
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  }
```

Code Fragment 10.5: An implementation of a map using a Java ArrayList as an unsorted table (continued from Code Fragment 10.4).

Simple Unsorted Map Implementation

```
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 }
```

Code Fragment 10.5: An implementation of a map using a Java ArrayList as an unsorted table (continued from Code Fragment 10.4).