CS121 Data Structures A, C
Maps

Varduhi Yeghiazaryan
vyeghiazaryan@aua.am

Fall 2021

Maps

A **map** is an abstract data type designed to efficiently store and retrieve values based upon a uniquely identifying **search key** for each

A map stores key-value pairs (k, v), called **entries**, where k is the key and v is its corresponding value

Keys are required to be **unique**, i.e. multiple entries with the same key are **not** allowed

For example, the mapping from URLs (key) to page content (value)

The main operations of a map are for searching, inserting, and deleting items

Maps are also known as **associative arrays**, because the entry's key serves somewhat like an index into the map, in that it assists the map in efficiently locating the associated entry

The Map ADT

A map M is a collection of key-value pairs and supports:

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

Our definition of the map ADT is a simplified version of the `java.util.Map` interface

We rely on our composite `Entry` interface; the `java.util.Map` relies on the nested `java.util.Map.Entry` interface

Each of the operations `get(`k`)`, `put(`k`,`v`)`, and `remove(`k`)` returns `null` if the map doesn't have such an entry

Some implementations of the `java.util.Map` interface explicitly forbid use of a `null` value (and `null` keys)

The interface contains a boolean method, `containsKey(`k`)` to check whether k exists as a key

This helps resolve the ambiguity when `null` is an allowed value

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()	$\{B,D\}$	$\{(7,B),(8,D)\}$

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 }
```

Application: Counting Word Frequencies

Case study: consider the problem of counting the number of occurrences of words in a document

A map is an ideal data structure to use here, for we can use words as keys and word counts as values

We begin with an empty map, mapping words to their integer frequences

We first scan through the input, considering adjacent alphabetic characters to be words, which we then convert to lowercase

For each word found, we attempt to retrieve its current frequency from the map using the `get` method, with a yet unseen word having frequency zero

We then (re)set its frequency to be one more to reflect the current occurrence of the word

After processing the entire input, we loop through the `entrySet()` of the map to determine which word has the most occurrences

Application: Counting Word Frequencies (cont'd)

```
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 }
```

An AbstractMap Base Class

We will be providing many different implementations of the map ADT using a variety of data structures, each with its own trade-off of advantages and disadvantages

We design an `AbstractMap` base class that provides functionality common for all map implementations:

An AbstractMap Base Class

We will be providing many different implementations of the map ADT using a variety of data structures, each with its own trade-off of advantages and disadvantages

We design an `AbstractMap` base class that provides functionality common for all map implementations:

- 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
- concrete implementations of the `keySet` and `values` methods, based upon an adaptation to the `entrySet` method

Note that this approach is reminiscent of providing an iteration of all elements of a positional list given an iteration of all positions of the list

An AbstractMap Base Class in Java

```
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 in Java (cont'd)

```
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  }
```

A Simple Unsorted Map Implementation

We demonstrate 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`

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

Thus, a nonpublic utility `findIndex(key)` returns the index at which such an entry is found, or -1 if no such entry is found

Can we implement the update step of the `remove` method in constant time?

A Simple Unsorted Map Implementation

We demonstrate 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`

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

Thus, a nonpublic utility `findIndex(key)` returns the index at which such an entry is found, or -1 if no such entry is found

Can we implement the update step of the `remove` method in constant time? Relocate the last entry to that location.

On a map with n entries, each of the fundamental methods takes

A Simple Unsorted Map Implementation

We demonstrate 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`

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

Thus, a nonpublic utility `findIndex(key)` returns the index at which such an entry is found, or -1 if no such entry is found

Can we implement the update step of the `remove` method in constant time? Relocate the last entry to that location.

On a map with n entries, each of the fundamental methods takes $O(n)$ time in the worst case since we need to scan through the entire list

A Simple Unsorted Map Implementation in Java I

```
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   }
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   }
```

A Simple Unsorted Map Implementation in Java II

```
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    }
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    }
```

A Simple Unsorted Map Implementation in Java III

```
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  }
```

The Sorted Map ADT

A **sorted map** is an extension of the standard map, that, in addition, supports:

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

The Sorted Map Interface

It is straightforward to implement the `SortedMap` interface

Try it as an exercise.

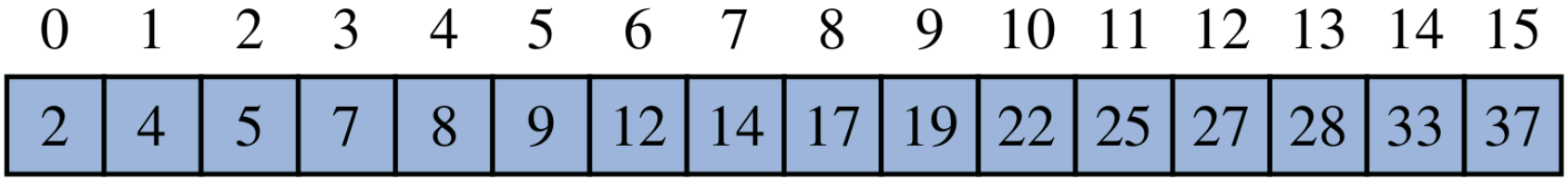
We also implement an abstract class `AbstractSortedMap` combining the functionality that is common for all sorted maps

An AbstractSortedMap Base Class in Java

```
1   public abstract class AbstractSortedMap<K,V>
2     extends AbstractMap<K,V> implements SortedMap<K,V> {
3     /** The comparator defining the ordering of keys in the map. */
4     private Comparator<K> comp;
5     /** Initializes the comparator for the map. */
6     protected AbstractSortedMap(Comparator<K> c) { comp = c; }
7     /** Initializes the map with a default comparator. */
8     protected AbstractSortedMap() {
9       this(new DefaultComparator<K>());                    // default comparator uses natural ordering
10    }
11    /** Method for comparing two entries according to key */
12    protected int compare(Entry<K,V> a, Entry<K,V> b) {
13      return comp.compare(a.getKey(), b.getKey());
14    }
15    /** Method for comparing a key and an entry's key */
16    protected int compare(K a, Entry<K,V> b) {
17      return comp.compare(a, b.getKey());
18    }
19    /** Method for comparing a key and an entry's key */
20    protected int compare(Entry<K,V> a, K b) {
21      return comp.compare(a.getKey(), b);
22    }
23    /** Method for comparing two keys */
24    protected int compare(K a, K b) { return comp.compare(a, b); }
25    /** Determines whether a key is valid. */
26    protected boolean checkKey(K key) throws IllegalArgumentException {
27      try {
28        return (comp.compare(key,key)==0);                 // see if key can be compared to itself
29      } catch (ClassCastException e) {
30        throw new IllegalArgumentException("Incompatible key");
31      }
32    }
33  }
```

Sorted Search Tables

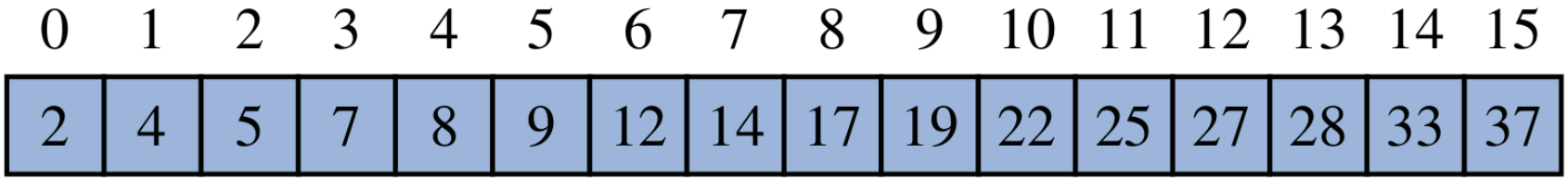
In a **sorted search table** implementation of a sorted map, we store the map's entries in an array list A so that they are in increasing order of their keys



Space requirements:

Sorted Search Tables

In a **sorted search table** implementation of a sorted map, we store the map's entries in an array list A so that they are in increasing order of their keys



Space requirements: $O(n)$

This array-based representation allows us to use **binary search**

Note that we have already seen a binary search implementation that returns an index or -1 instead of `true/false`

As a utility method, we define a recursive binary search that:

- returns the index of the entry with the given key or,
- if the key is absent, it returns the index at which a new entry with that key would be inserted

```
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); }
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   }
```

```
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    }
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    }
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  }
```

Comparing Map Implementations

A sorted search table can be used to implement the map ADT even if we don't want to use the additional functions of the ordered map ADT

Method	Unsorted Search Table	Sorted Search Table
size, isEmpty	$O(1)$	$O(1)$
get	$O(n)$	$O(\log n)$
put	$O(n)$	$O(n)$; at times $O(\log n)$
remove	$O(n)$	$O(n)$
firstEntry, lastEntry		$O(1)$
ceilingEntry, floorEntry, lowerEntry, higherEntry		$O(\log n)$
subMap		$O(s + \log n)$
entrySet, keySet, values	$O(1)$ **lazy**; $O(n)$ snap.	$O(1)$ lazy; $O(n)$ **snap.**

When should we use sorted search tables?

Summary

Reading

Section 10.1 Maps

Section 10.3 Sorted Maps

Questions?