Lecture 9: ADT Map

Dr. Kieran T. Herley

Department of Computer Science
University College Cork

2013/14

Summary

ADT Map: definition and usage. Simple array-based implementation of the ADT.

ADT Map

- Collection of *entries*.
- entry = *key* + *value*.
  - keys may be almost anything Integer, Strings *etc.* but
    - keys *must* be distinct
    - must be “compare-able”
  - value could be anything
- Supports methods for:
  - also adding/removing an item
  - seraching for an item based on its key

Potential Applications of Map-like Concept

- Telephone book
  (key = name, value = phone number)
- Simple grades notebook
  (key = student id, value = grades)
- Compiler symbol table
  (key = identifier, value = type *etc.*)
- Other applications based on map-like structures
  - Unix utilities: finger,
  - DNS?

ADT Map

The specifications below are expressed in terms of K and V, representing the key type and value type respectively.

get(k): If map contains an entry with key equal to k, then return the value of that entry, else return null. *Input:* K; *Output:* V.

put(k, v): If the map does not have an entry with key equal to k, add entry (k, e) and return null, else, replace with v the existing value of the entry and return its old value. *Input:* K, V; *Output:* V.

remove(k): Remove from the map the entry with key equal to k and return its value; if there is no such entry, return null. *Input:* K; *Output:* V.

iterator(): Return an iterator of the entries stored in the map. *Input: None*; *Output: Iterator<Entry<K, V>>*.

- Also size() and isEmpty() plus others. See handout.

Illustration

Operation	Map	Output
put(5, A)	$\boxed{\frac{5}{A}}$	null
put(13, B)	$\boxed{\frac{5}{A}}\boxed{\frac{13}{B}}$	null
get(5)	$\boxed{\frac{5}{A}}\boxed{\frac{13}{B}}$	A
put(3, C)	$\boxed{\frac{5}{A}}\boxed{\frac{13}{B}}\boxed{\frac{3}{C}}$	null
remove(13)	$\boxed{\frac{5}{A}}\boxed{\frac{3}{C}}$	B
size()	$\boxed{\frac{5}{A}}\boxed{\frac{3}{C}}$	2
get(101)	$\boxed{\frac{5}{A}}\boxed{\frac{3}{C}}$	null

Example

Problem Scan a list of positive numbers (from a random number generator) and output the distinct values that appear in the list.

Idea

- Maintain a map of numbers seen
  - number x represented with [key $=$x, value $=$x]
  - (actually never use values)
- Use searchability of map to check novelty of each number
- For each number read, check to see if it already appears in the map:
  - if it does, simply ignore it
  - if it doesn't, add it to the map and print it

Code cont'd

```
// Class that generates random numbers in range 1 to 10
RandomNumGenerator numSource = new RandomNumGenerator(10);
Map<Integer, Integer> numsSeen =
   new ArrayBasedMap<Integer, Integer>();
int  current;

/* generate seq. of numbers and print unique values */
```

Code cont'd

```
RandomNumGenerator numSource = new RandomNumGenerator(10);
Map<Integer, Integer> numsSeen =
   new ArrayBasedMap<Integer, Integer>();
int current;

System.out.println("The numbers are:");
for (int i = 0; i < 20; i++)
{  current = numSource.nextNumber();
   /* deal with current */
}
```

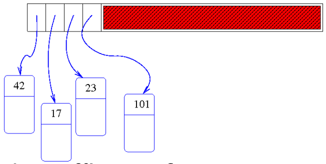
Code cont'd

```
RandomNumGenerator numSource =
   new RandomNumGenerator(10);
Map<Integer, Integer> numsSeen =
   new ArrayBasedMap<Integer, Integer>();
int current;
```

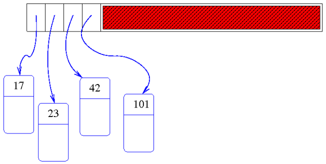
```
System.out.println("The numbers are:");
for (int i = 0; i < 20; i++)
{  curent = numSource.nextNumber();
   if (numsSeen.get(current) == null)
   {  System.out.println(current);
      numsSeen.put(current, current);
   }
}
```

Array-Based Implementations of ADT Map

Unsorted array:



Sorted array:



Both inefficient for remove:

$$\text{running time for remove} = O(\text{map size}),$$

but sorted version allows for more efficient get (binary search). More on this later. [1]

[1]For we treat $O(\text{map size})$ as meaning "proportional to the sizwe of the map".

Map.java

```
import java.util.Iterator;
public interface Map<KeyType, ValueType>
       extends java.lang.Iterable<Entry<KeyType, ValueType>>
{ public int size();
  public boolean isEmpty();
  public ValueType get(KeyType k);
  public ValueType put(KeyType k, ValueType e);
  public ValueType remove(KeyType k);
  public Iterator<Entry<KeyType, ValueType>> iterator();
}
```

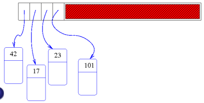
Treat iterator-related material as “invisible” for now

Map.java cont'd

```
. . .
public interface Map<KeyType, ValueType>
         . . .
{ public int  size ();
  public boolean isEmpty();
  public ValueType get(KeyType k);
  public ValueType put(KeyType k, ValueType e);
  public ValueType remove(KeyType k);
  . . .
}
```

A Simple Map Implementation

Representation

- 

- entries– *unsorted* array of Entry/MapEntry objects
- numEntries– the no. of entries

Entry(Interface)/MapEntry(Implementation)

- members: key (KeyType), value (ValType)
- operations:
    - key()– return key
    - value()– return value

Useful "Helper" Operation

-
  ```
  Algorithm findEntry(k):
      for i ←0 to numEntries−1 do
          if k = entries[i].key() then
              return i
      return −1
  ```
- Snag– expressing comparisons in Java?
  - want to allow for different key types
  - want non-type specific implementation
  - will revisit issue this later

Operation get

```
**Algorithm** get(k):
    int index = findEntry(k)
    if (index $\neq -1$)
        return entries[index].value()
    else
        return null
```

Operation put

```
**Algorithm** put(k, e):
    index = findEntry(k)
    newEntry = new Entry with key k, value e
    if (index $\neq -1$)
        /* handle replacement case */
    else
        /* handle insertion case */
```

Operation put cont'd

```
**Algorithm** put(k, e):
    index = findEntry(k)
    newEntry = new Entry with key k, value e
    if (index $\neq -1$)
        oldVal = entries[index].value()
        entries[index] = newEntry
        return oldVal
    else
        /* handle insertion case */
```

Operation put cont'd

```
**Algorithm** put(k, e):
    index = findEntry(k)
    newEntry = new Entry with key k, value e
    **if** (index $\neq -1$)
        oldVal = entries[index].value()
        entries[index] = newEntry
        **return** oldVal
    **else**
        **if** (numEntries = capacity) **then** . . .
        entries[numEntries++] = newEntry
        **return** null
```

ArrayBasedMap

```
public class ArrayBasedMap<KeyType, ValueType>
              implements Map<KeyType, ValueType>
{  public ArrayBasedMap()
   {   entries  = new MapEntry [INITCAPACITY];
       capacity  = INITCAPACITY;
       numEntries = 0;
   }
   . . .
   private static final int INITCAPACITY = 100;
   protected static final int NOSUCHKEY = −1;
   protected Entry<KeyType, ValueType> [] entries;
   protected int numEntries;
}
```

ArrayBasedMap–findEntry

```
/* temporary placeholder-- better version later */
private boolean isEqualTo(KeyType k1, KeyType k2)
{  return (k1.equals(k2));
}

/* temporary placeholder-- better version later */
private int findEntry(KeyType key)
{  for (int i = 0; i < numEntries; i++)
   {  if (isEqualTo(key, entries[i].getKey()))
      {  return i;
      }
   }
   return NOSUCHKEY;
}
```

ArrayBasedMap –get

```
public ValueType get(KeyType k)
{   int indexWithKey = findEntry(k);
    if (indexWithKey != NOSUCHKEY)
        return entries[indexWithKey].getValue();
    else
        return null;
}
```

ArrayBasedMap–put

```
public ValueType put(KeyType k, ValueType e)
{  Entry<KeyType, ValueType> newEntry =
      new MapEntry<KeyType, ValueType>(k, e);
   int index = findEntry(k);
   if (index != NOSUCHKEY)
   {  ValueType oldVal = entries[index].getValue();
      entries[index] = newEntry; return oldVal;
   } else
   {  expandIfNecessary();
      entries[numEntries++] = newEntry;
   }
   return null;
}
```

Using ArrayBasedMap

```
Map<String, String> myFriends;

myFriends = new ArrayBasedMap<String, String>();
myFriends.put("tom", "123456");
myfriends.put("dick", "234567");
myFriends.put("harry", "345678");

String  tomsPhone = myFriends.get("tom");
```