The Iterator Design Pattern (illustrated through the Map ADT)

Readings: DSA Chapter 10

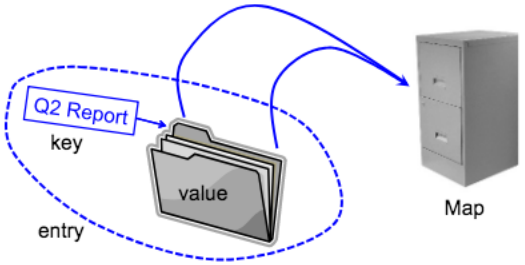
YORK UNIVERSITÉ UNIVERSITY

EECS2030: Advanced Object Oriented Programming
Spring 2017

CHEN-WEI WANG

What is a Map?

- A *map* stores a collection of *entries*.
- Each *entry* is a pair: an *object* and its *(search) key*.
- Each *search key*:
  - *Uniquely* identifies an object in the map
  - Can be used to *efficiently* retrieve the associated object
- Search keys must be *unique* (i.e., do not contain duplicates).



Learning Outcomes of this Lecture

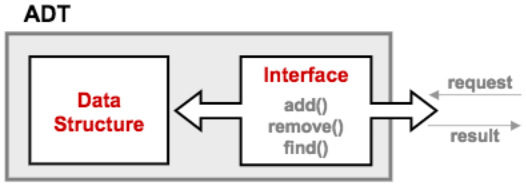
Understand:

- Concept of a Map
- Map ADT
- Map in Java [interface, classes]

Arrays are Maps

- Each array *entry* is a pair: an object and its *numerical* index.
- *Search keys* are the set of numerical index values.
- The set of index values are *unique* [e.g., $0 .. (a.length - 1)$]
- Given a *valid* index value i, we can
  - *Uniquely* determines where the object is [$(i+1)^{th}$ item]
  - *Efficiently* retrieves that object [`a[i]` takes $O(1)$]
- Maps in general may have *non-numerical* key values:
  - Student ID [student record]
  - Social Security Number [resident record]
  - Passport Number [citizen record]
  - Residential Address [household record]
  - Media Access Control (MAC) Address [PC/Laptop record]
  - Web URL [web page]
  - ...

The Map ADT



- *Accessors*
  - ***size***
  - ***isEmpty***
  - ***get(k)*** [*value* associated with *k*]
  - ***keySet*** [iterable collection of *keys*]
  - ***values*** [iterable collection of *values*]
  - ***entrySet*** [iterable collection of *key-value pairs*]
- *Mutators*
  - ***put(k, v)*** [*add* a new entry or replace value in existing pair]
  - ***remove(k)*** [*remove* an existing entry with *k*]

Map: Illustration (1)

Consider the following Map operations:

- *Initial map*

$$\begin{aligned} m.keySet &= \{\,\} \\ m.values &= \{\,\} \\ m.EntrySet &= \{\,\} \end{aligned}$$

- *put("jim", 6478271029)*
  "jim" is *not* an existing search key in the map

$$\begin{aligned} m.keySet &= \{\,"jim"\,\} \\ m.values &= \{\,6478271029\,\} \\ m.EntrySet &= \{\,("jim", 6478271029)\,\} \end{aligned}$$

Map: Illustration (2)

- *put("jonathan", 4162534876)*
  "jonathan" is *not* an existing search key in the map

$$\begin{aligned} m.keySet &= \{\,"jim",\ "jonathan"\,\} \\ m.values &= \{\,6478271029,\ 4162534876\,\} \\ m.EntrySet &= \{\,("jim", 6478271029),\ ("jonathan", 4162534876)\,\} \end{aligned}$$

Map: Illustration (3)

- *put("alan", 9058729384)*
  "alan" is *not* an existing search key in the map

$$\begin{aligned} m.keySet &= \{\,"jim",\ "jonathan",\ "alan"\,\} \\ m.values &= \{\,6478271029,\ 4162534876,\ 9058729384\,\} \\ m.EntrySet &= \{\,("jim", 6478271029),\ ("jonathan", 4162534876),\ ("alan", 9058729384)\,\} \end{aligned}$$

Map: Illustration (4)

- *put("jim", 5039283049)*

  "jim" *is* an existing search key in the map

  Update the value of the entry identified by "jim"!

$$
\begin{aligned}
m.keySet &= \left\{ \begin{array}{l} "jim", \\ "jonathan", \\ "alan" \end{array} \right\} \\
m.values &= \left\{ \begin{array}{l} 5039283049, \\ 4162534876, \\ 9058729384 \end{array} \right\} \\
m.EntrySet &= \left\{ \begin{array}{l} ("jim", 5039283049), \\ ("jonathan", 4162534876), \\ ("alan", 9058729384) \end{array} \right\}
\end{aligned}
$$

Map: Illustration (5)

- *remove("jonathan")*

  "jonathan" *is* an existing search key in the map

  Remove the entry identified by "jonathan"!

$$
\begin{aligned}
m.keySet &= \left\{ \begin{array}{l} "jim", \\ "alan" \end{array} \right\} \\
m.values &= \left\{ \begin{array}{l} 5039283049, \\ 9058729384 \end{array} \right\} \\
m.EntrySet &= \left\{ \begin{array}{l} ("jim", 5039283049), \\ ("alan", 9058729384) \end{array} \right\}
\end{aligned}
$$

Map: Illustration (6)

- *remove("simon")*

  "simon" is *not* an existing search key in the map

  No change to the map!

$$
\begin{aligned}
m.keySet &= \left\{ \begin{array}{l} "jim", \\ "alan" \end{array} \right\} \\
m.values &= \left\{ \begin{array}{l} 5039283049, \\ 9058729384 \end{array} \right\} \\
m.EntrySet &= \left\{ \begin{array}{l} ("jim", 5039283049), \\ ("alan", 9058729384) \end{array} \right\}
\end{aligned}
$$

Generic Map in Java (Version 1)

```
public class Entry<K, V> {
  private K key;
  private V value;
  Entry(K key, V value) { ... }
  K getKey() { ... }
  V getValue() { ... }
  void setValue(V value) { ... }
}
public interface Map<K, V> {
   int size();
   boolean isEmpty();
   V get(K key);
   void put(K key, V value);
   V remove(K key);
   K[] keySet();
   V[] values();
   Entry<K, V>[] entrySet();
}
```

Implementing Map ADT: Two Arrays

- Maintain two parallel arrays:
  - One stores *keys*
  - The other stores *values*
- The *keys* array and the *values* array:
  - Are of the same size
  - Are filled withe the same number of elements
- Given a valid index *i*, (*keys*[*i*], *values*[*i*]) denotes a valid entry in the map.
- Maintain a **private** counter on the # of *currently-stored entries*.
- How to implement the removal of an entry?
  - First locate the key, say at index *i* in the *keys* array. [*linear* RT]
  - Empty the slots at index *i* in both *keys* and *values*.
  - Two possible solutions:
    - Move items at indices > *i* to the left by one position [*linear* RT]
    - Store *i* in a stack/queue for later reuse [*constant* RT]

Implementing Map ADT: Two Arrays (1.1)

```
public class ArrayMap1<K, V> implements Map<K, V> {
  private K[] keys;
  private V[] values;
  private final int MAX_SIZE = 1000;
  /* number of entries
   * index of next available slot
   */
  private count;
  ArrayMap1() {
    keys = new K[MAX_SIZE];
    values = new V[MAX_SIZE];
    count = 0;
  }
  int size() { return count; }
  boolean isEmpty() { return count == 0; }
  ...
}
```

Running time? *O*(1)

Implementing Map ADT: Two Arrays (1.2)

```
Entry<K, V>[] entrySet() {
  Entry<K, V>}[] eSet = new Entry<K, V>[count];
  for(int i = 0; i < count; i ++) {
    eSet[i] = new Entry<K, V>(keys[i], values[i]);
  }
  return eSet;
}
K[] keySet() {
  /* similar to how entrySet() is implemented */
}
V[] values() {
  /* similar to how entrySet() is implemented */
}
```

Running time? *O*(*n*)

Implementing Map ADT: Two Arrays (1.3)

```
private int indexOfKey(K key) {
  int index = -1;
  boolean keyFound = false;
  for(int i = 0; !keyFound && i < count; i ++) {
    if (keys[i].equals(key)) {
      keyFound = true;
      index = i;
    }
  }
  return index;
}
```

- *Stay* in the `for` loop as long as the parameter `key` is **not** yet found *and* the loop counter `i` is still a valid index of `keys`.
- *Exit* the `for` loop either when the parameter `key` is already found *or* when the loop counter `i` has been incremented to exceed the maximum index of `keys`.
- Running time? *O*(*n*)

Implementing Map ADT: Two Arrays (1.4)

```
V get(K key) {
  int index = indexOfKey(key);
  if (index < 0) {
    return null;
  }
  else {
    return values[index];
  }
}
```

Running time? $O(n)$

Implementing Map ADT: Two Arrays (1.5)

```
void put(K key, V value) {
  int index = indexOfKey(key);
  if(index < 0) { /* case: inserting a new entry */
    keys[count] = key;
    values[count] = value;
    count ++;
  }
  else { /* case: updating the value of an existing entry */
    values[index] = value;
  }
}
```

Running time? $O(n)$

Implementing Map ADT: Two Arrays (1.6)

```
V remove(K key) {
  int index = indexOfKey(key);
  if (index < 0) { /* nothing to remove */
    return null;
  }
  else { /* remove by shifting */
    V removedVal = values[index];
    for(int i = index; i < count - 1; i ++) {
      keys[i] = keys[i + 1];
      values[i] = values[i + 1];
    }
    count --;
    keys[count] = null;
    values[count] = null;
    return removedVal;
  }
}
```

Running time? $O(n)$

Implementing Map ADT: Two Arrays (2.1)

```
public class ArrayMap2<K, V> implements Map<K, V> {
  private K[] keys;
  private V[] values;
  private Stack<Integer> freeIDs;
  private final int MAX_SIZE = 1000;
  private count; /* number of currently-stored entries */
  ArrayMap2() {
    keys = new K[MAX_SIZE];
    values = new V[MAX_SIZE];
    count = 0;
    freeIDs = new ArrayStack<>();
  }
  int size() { return count; }
  boolean isEmpty() { return count == 0; }
  ...
}
```

Running time? $O(1)$

Implementing Map ADT: Two Arrays (2.2)

```
Entry<K, V>[] entrySet() {
  Entry<K, V>}[] eSet = new Entry<K, V>[count];
  int next = 0;
  for(int i = 0; i < keys.length; i ++) {
    if (keys[i] != null) {
      eSet[next] = new Entry<K, V>(keys[i], values[i]);
      next ++;
    }
  }
  return eSet;
}
K[] keySet() { /* similar to entrySet() */ }
V[] values() { /* similar to entrySet() */ }
```

Running time? $O(n)$

Implementing Map ADT: Two Arrays (2.3)

```
private int indexOfKey(K key) {
  int index = -1;
  boolean keyFound = false;
  for(int i = 0; !keyFound && i < keys.length; i ++) {
    if (keys[i] != null && keys[i].equals(key)) {
      keyFound = true;
      index = i;
    }
  }
  return index;
}
```

- *Stay* in the `for` loop as long as the parameter `key` is **not** yet found *and* the loop counter `i` is still a valid index of `keys`.
- *Exit* the `for` loop either when the parameter `key` is already found *or* when the loop counter `i` has been incremented to exceed the maximum index of `keys`.
- Running time? $O(n)$

Implementing Map ADT: Two Arrays (2.4)

```
V get(K key) {
  int index = indexOfKey(key);
  if (index < 0) {
    return null;
  }
  else {
    return values[index];
  }
}
```

Running time? $O(n)$

Implementing Map ADT: Two Arrays (2.5)

```
void put(K key, V value) {
  int index = indexOfKey(key);
  if(index < 0) { /* case: inserting a new entry */
    if ( freeIDs.isEmpty() ) { /* no middle null slots */
      keys[count] = key;
      values[count] = value; }
    else { /* there are some middle null slots */
      freeID = freeIDs.pop();
      keys[freeID] = key;
      values[freeID] = value; }
    count ++; }
  else { /* case: updating the value of an existing entry */
    values[index] = value;
  }
}
```

Running time? $O(n)$

Implementing Map ADT: Two Arrays (2.6)

```
V remove(K key) {
  int index = indexOfKey(key);
  if (index < 0) { /* nothing to remove */
    return null;
  }
  else { /* remove by emptying slots */
    V removedVal = values[index];
    keys[index] = null;
    values[index] = null;
    freeIDs.push(index);
    count --;
    return removedVal;
  }
}
```

Running time? still $O(n)$ even without the shift!

Implementing Map ADT: Homework

- Implement the `Map` interface using parallel doubly-linked lists:

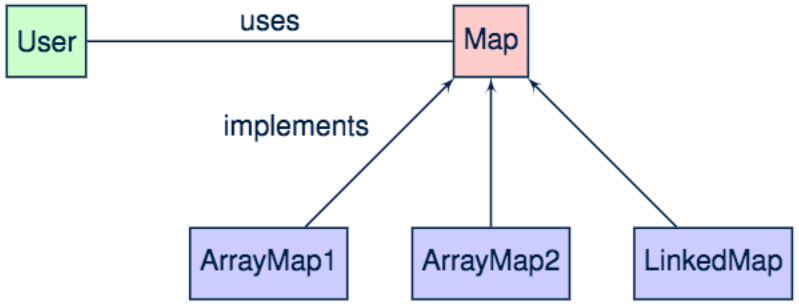
```
public class LinkedMap<K, V> implements Map<K, V> {
  ...
}
```

- A *stack/queue* to store the freed (but not reused) indices?
  No! ∵ **no** `null` slots in-between linked nodes
- Analogous to the `int indexOf(K key)` method in array-based maps, define a helper method:

```
private Node<V> nodeOf(K key) {
  /* Return the linked node that contains key */
}
```

- Determine the *running time* of all `Map` operations implemented using two linked lists.

Implementing Map ADT: Architecture



Implementing Map ADT: Design Issue of `keySet`, `values`, `entrySet` (1)

Return Types of keySet, values, and entrySet are all *arrays*.

Design Issue

- Users are forced to use *LinkedMap* like an array

```
1 Entry<String, Integer>[] entries = m.entrySet();
2 boolean hasNegItem = false;
3 for(int i = 0; !hasNegItem && i < entries.length; i ++) {
4   hasNegItem = entries[i].getValue() < 0;
5 }
```

  - **Lines 3 and 4** use array-specific features: `length` and indexing
- But the *order of map entries does not matter*!

We may only allow users to access contents of a collection *without revealing their internal order*.

Implementing Map ADT: Design Issue of `keySet`, `values`, `entrySet` (2)

Solution: A way for accessing map contents, *despite* the underlying implementations (e.g., arrays, linked lists, *etc.*), that is:

- *uniformed*: access is via operations that are *not array-specific*.
- *abstract*: access does *not reveal the order* of stored entries.

Iterable and Iterator in Java

Two *interfaces* to learn to both use and implement:

1. The *`Iterator<E>`* Interface
   - https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html
   - Supports *operations* for accessing a collection *without* knowing its internal data structure (e.g., arrays, linked lists, trees, or graphs)
   - `boolean hasNext()`
     determines if there are more elements to iterate over
   - `E next()`
     returns an element from the collection
2. The *`Iterable<E>`* Interface
   - https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html
   - Supports a single operation
   - `Iterator<E> iterator()`
     returns an iterator object

Generic Map in Java (Version 2)

```
public interface Map<K, V> {
  ...
  Iterable<K> keySet();
  Iterable<V> values();
  Iterable<Entry<K, V>> entrySet();
}
public class ArrayMap<K, V> implements Map<K, V> {
  ...
  Iterable<K> keySet() { ... };
  Iterable<V> values() { ... };
  Iterable<Entry<K, V>> entrySet() { ... };
}
public class LinkedMap<K, V> implements Map<K, V> {
  ...
  Iterable<K> keySet() { ... };
  Iterable<V> values() { ... };
  Iterable<Entry<K, V>> entrySet() { ... };
}
```


Using Iterable and Iterator in Java (1)

- Two library classes implementing the `Iterable<E>` interface:
  - `ArrayList<E>`: an array implementation of a list
  - `LinkedList<E>`: a doubly-linked list
- Given a list

```
ArrayList<String> list = new ArrayList<String>();
list.add("Alan"); list.add("Mark"); list.add("Tom");
```

- **Usage One**: `for-each` loop at the *`Iterable`* level:

```
for (String s : list) {
  System.out.println(s);
}
```

- **Usage Two**: `while` loop at the *`Iterator`* level:

```
Iterator<String> it = list.iterator();
while(it.hasNext()) {
  System.out.println(it.next());
}
```

Implementing Iterable and Iterator in Java

Strategy:

- Make the internal data structure *iteratble*.
- For example, transform
  - keys and values arrays; or
  - keys and values linked-lists

  using one of the library classes that are *iteratble*:
  - `ArrayList`
  - `LinkedList`
- In the `ArrayList` and `LinkedList` library classes, the `iterator()` method is already implemented for you to use!

Implementing `keySet`, `values`, `entrySet` in `ArrayMap`

```
1 Iterable<Entry<K, V>> entrySet() {
2   ArrayList<Entry<K, V>> list = new ArrayList<>();
3   for(int i = 0; i < size; i ++) {
4     list.add(new Entry<K, V>(keys[i], values[i]));
5   }
6   return list;
7 }
```

- **L1**: *Return type* of `entrySet` declared as interface `Iterable`.
- **L2**: *Static type* of `list` declared as `ArrayList`.
- **L6**: *Valid* ∵ `ArrayList` (static type of `list`) is a *descendent class* of `Iterable` (return type of method `entrySet`).

  Exercise: Implement `Iterable<K> keys()` and `Iterable<V> values()` in `ArrayMap` and `LinkedMap`.

Implementing `keySet`, `values`, `entrySet` in `LinkedMap`

- Similar to how you implement these operations in `ArrayMap`, except that you have to construct a Java `ArrayList` out of *two linked lists* rather than two arrays.
- **Exercise** for you!

Using Iterable and Iterator in Java (2)

- Given a map

```
Map<String, Integer> m = new LinkedMap<>();
m.add("Alan", 1); m.add("Mark", 2); m.add("Tom", 3);
```

- **Usage One**: `for-each` loop at the `Iterable` level:

```
Iterable<Entry<String, Integer>> entries = m.entrySet();
for (Entry<String, Integer> e : entries) {
  System.out.println(e.getValue());
}
```

- **Usage Two**: `while` loop at the `Iterator` level:

```
Iterator<Entry<String, Integer>> it
    = m.entrySet().iterator();
while(it.hasNext()) {
  System.out.println(it.next());
}
```

Index (1)

Index (2)

Index (3)