Lecture 26

Maps

- Maps
  - structure Package
  - MapList Implementation
  - Better Implementations
Maps
Associations and Maps

Earlier in the course, we looked at a simple object called an association, which has a key and a value.

In the `structure` package we have the following:

- The key is not null.
- The value can be null.

There are many situations in which we want to store a set of associations. These are often called maps.

We say that the key is mapped to the value.

- The keys must be unique in the map.
- The values do not need to be unique.

The terminology is from mathematical functions, in which each element in the domain (i.e. the key) is mapped to one element in the range (i.e. the value).
Maps vs Arrays

A map is a generalization of an array. This is because an array can be viewed as follows:

- The keys are the non-negative integer indices.
- Each key is mapped to the value at that index.

For example, if array[2] = C, then we can view the array as mapping the key 2 to the value C.

As a result, maps are also known as associative arrays.

Maps generalize arrays in several ways:

- The types of keys.
- The keys don’t necessarily need to be comparable.
- The keys may not be known in advance.
- The number of keys may not be known in advance.
structure Package
Exercise: What is Map in structure?

What do you think a Map should be in the structure package?

- Will it be an interface, a class, or an abstract class?
- Will it implement any other interfaces?
- Will it extend from any other classes?
- Will it have any type variables?

Think to yourself for 30 seconds.
Debate with a neighbor for 1 minute.

There is no “correct” answer.
However, you should be able to justify your answer.
Map is an abstract concept. Also, there isn't a single most obvious way to implement it.

Map is an interface in the structure package. There are many methods with get, put, and remove having particular importance.
The `AbstractMap` class provides default implementations for a couple of methods.

- Its `putAll` method runs `put` on each (key, value) pair from the other `Map`. This saves time for other implementations of `Map` that extend `AbstractMap`. 

```java
public abstract class AbstractMap<K, V> implements Map<K, V> {
    /**
     * @pre other is a valid map
     * @post adds the map entries of other map into this, possibly replacing value
     */
    public void putAll(Map<K, V> other) {
        Iterator<K> i = other.keySet().iterator();
        while (i.hasNext()) {
            K k = i.next();
            put(k, other.get(k));
        }
    }

    /**
     * Compute the hashCode for elements of this map
     */
    public int hashCode() {
        return values().hashCode();
    }
}
```

// This could also be added to the `AbstractMap` class.
public boolean isEmpty() {
    return size() == 0;
}
Map is used extensively in the structure package.

- We’ll discuss OrderedMap
Three of the most important methods in `Map`.

The documentation is a little bit sparse here.

- What does the `put` method return?

This is clarified in another class that implements the Map interface.
Exercise: How to implement Map?

What is the most basic implementation of a Map that you can design?

There are many methods in Map so just focus on the following:

- `get(K k)`  // returns the value currently associated with the key K (or null)
- `put(K k, V v)`  // sets key K's mapping to value V and returns its current value (or null)

What are their run-times in your implementation?

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>get</th>
<th>put</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>O(n)-time</td>
<td>O(n)-time</td>
</tr>
<tr>
<td>vector</td>
<td>O(n)-time</td>
<td>O(n)-time</td>
</tr>
<tr>
<td>linked list</td>
<td>O(n)-time</td>
<td>O(n)-time</td>
</tr>
</tbody>
</table>

Run-times for a Map with n entries when implemented with unsorted linear data structures that store Associations. (The put run-times assume doubling the array when full.)

New nodes can be added to a linked list in worst-case O(1)-time, and to an array or Vector in amortized O(1)-time (using the double-when-full approach), but this doesn’t give O(1)-time for put.

A map stores one value per key, so put must first determine if the key is already present (and it it returns the current value if it is). In other words, put is more like update than an add method.
MapList Implementation
One of the simplest implementations of a map uses an unsorted (singly) linked list.

- Each node in the list contains a single (key, value) pair.

This approach is used by the MapList class in the structure package.

- Each node in the list contains a single Association object.

The worst-case run-times of various operations when implementing a map using a singly linked list as in MapList.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>get</td>
<td>$O(n)$-time</td>
<td>$O(n)$-time</td>
</tr>
<tr>
<td>put</td>
<td>$O(n)$-time</td>
<td>$O(n)$-time</td>
</tr>
<tr>
<td>remove</td>
<td>$O(n)$-time</td>
<td>$O(n)$-time</td>
</tr>
<tr>
<td>contains Key</td>
<td>$O(n)$-time</td>
<td>$O(n)$-time</td>
</tr>
<tr>
<td>contains Value</td>
<td>$O(n)$-time</td>
<td>$O(n)$-time</td>
</tr>
</tbody>
</table>

The number of (key, value) pairs currently in the map is $n$.

The run-times of these operations are $\Omega(n)$-time (i.e., at least $O(n)$-time) because we may need to search every (key, value) pair in the structure. This is also true for array / vector implementations.

**Question:** How can we improve these to $O(\log n)$-time? Think about structures that we have studied.
MapList is a simple implementation of a `Map` in the `structure` package.

- data is declared as a `List` (an interface) and instantiated as a `SinglyLinkedList`.
- There is no attempt to order the data. In fact, the data might not be `Comparable`. 
MapList is a simple implementation of a Map in the structure package.

- data is declared as a List (an interface) and instantiated as a SinglyLinkedList.
- There is no attempt to order the data. In fact, the data might not be Comparable.
Class Discussion: `putAll` and Map Iteration

It is interesting to note that `MapList` does not extend from `AbstractMap`. As a result, it does not inherit the implementation of `putAll`.

```java
public abstract class AbstractMap<K,V> implements Map<K,V>
{
    /**
     * @pre other is a valid map
     * @post adds the map entries of other map into this, possibly
     * replacing value
     */
    public void putAll(Map<K,V> other)
    {
        Iterator<K> i = other.keySet().iterator();
        while (i.hasNext())
        {
            K k = i.next();
            put(k,other.get(k));
        }
    }
}
```

The `putAll` method in `AbstractMap`.

```java
// @pre other is non-null
// @post all the mappings of other are installed in this map,
// overriding any conflicting maps
public void putAll(Map<K,V> other) {
    Iterator<Association<K,V>> i = other.entrySet().iterator();
    while (i.hasNext())
    {
        Association<K,V> e = i.next();
        put(e.getKey(), e.getValue());
    }
}
```

The `putAll` method in `MapList`.

Questions:

- The two implementations iterate over different sets: `keySet` vs `entrySet`.
- Which implementation is faster? Why?
- Do you have any questions or suggestions or theories regarding the structure package?