

CS 228: Introduction to Data Structures

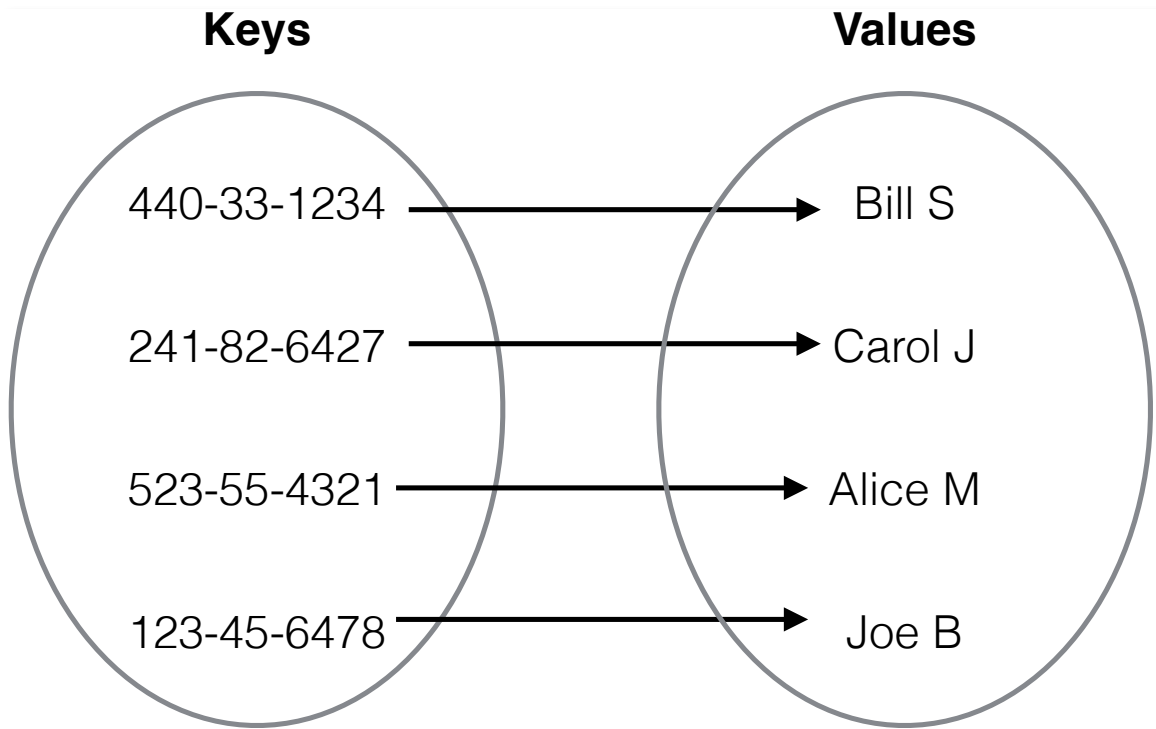
Lecture 32

Monday, November 16, 2016

Maps

A **map** is an object that maps a finite set of **keys** to a collection of **values**. Each key can map to at most one value, and a map cannot contain duplicate keys. Maps correspond to the mathematical concept of a **function**. Maps have many applications. Here are three.

- Dictionaries: The keys are words, the values are their definitions (e.g., as in Project 4). In fact, some authors call maps dictionaries, and the (now obsolete) predecessor of Java's Map class was called Dictionary.
- Associating social security numbers to people's names. The keys are 9-digit integers; the values are Strings.
- Associating IP addresses (keys, Strings) to host names (values, Strings).



The Java Map<K, V> Interface

Java's Map interface specifies the methods of a mapping from a set of keys, of type K, to a set of values, of type V.

```
public interface Map<K, V>
```

Here is a partial list of the methods in the Map interface.

```
boolean containsKey(Object key)
```

Returns true if this map contains a mapping for key.

```
boolean containsValue(Object value)
```

Returns true if this map maps one or more keys to value.

`V get(Object key)`

Returns the value to which key is mapped, or null if this map contains no mapping for key.

`boolean isEmpty()`

Returns true if this map contains no key-value mappings.

`V put(K key, V value)`

Associates value with key in this map.

`V remove(Object key)`

Removes the mapping for key from this map if it is present.

`int size()`

Returns the number of key-value mappings in this map.

The Map interface provides three collection views, which allow a map's contents to be viewed as a set of keys, collection of values, or set of key-value mappings.

```
Set<K> keySet()
```

Returns a Set view of the keys contained in this map.

```
Collection<V> values()
```

Returns a Collection view of the values contained in this map.

```
Set<Map.Entry<K,V>> entrySet()
```

Returns a Set view of the mappings contained in this map.

The order of a map is defined as the order in which the iterators on the map's collection views return their elements. Some map implementations make specific guarantees as to their order; others do not.

Implementation

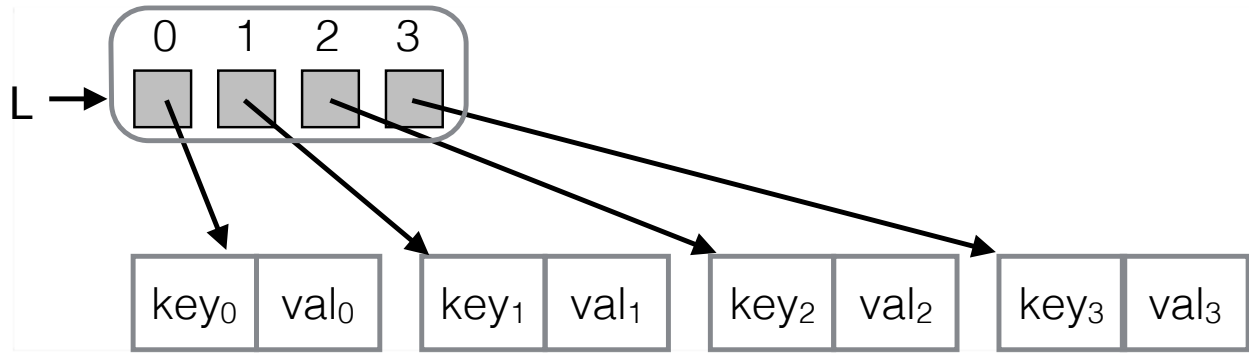
Implementing the full Java Map interface can be overwhelming, so we will not attempt to do it here. To give you an idea of how an implementation might look, we will study how to implement the following stripped-down version of the Map interface called SimpleMap.

```
public interface SimpleMap<K, V>
{
    public V put(K key, V value);
    public V get(K key);
    public V remove(K key);
    public boolean containsKey(K key);
    public int size();
    public Iterator<K> keyIterator();
}
```

Most methods in `SimpleMap` match the `Map` API. One exception is `keyIterator()`, which is equivalent to calling `keySet().iterator()` in the `Map` API. The `keySet()` method returns a “set view” of the keys; i.e., a `Set` consisting of all the keys. This `Set` comes with its own iterator.

List Implementation

An easy way to implement `SimpleMap` is to use a `List` of `MapEntry` objects, which are key/value pairs.



For encapsulation, the list and the MapEntry type are private.

```
public class ListBasedMap<K, V>
implements SimpleMap<K, V>
{
    private List<MapEntry> list =
        new LinkedList<MapEntry>();

    private class MapEntry
    {
        public K key;
        public V value;

        public MapEntry(K key, V value)
        {
            this.key = key;
            this.value = value;
        }
    }
}
```

To put () a key/value pair into the map, we first do a linear scan to check if key is already being used. If it is, we replace key's old value with the new value, returning the old value. Otherwise, we just add () the key/value pair to the list.

```
public V put(K key, V value)
{
    if (key == null)
        throw new IllegalArgumentException();
    for (MapEntry entry : list)
    {
        if (entry.key.equals(key))
        {
            V ret = entry.value;
            entry.value = value;
            return ret;
        }
    }
    list.add(new MapEntry(key, value));
    return null;
}
```

The get () method also requires a linear scan through the list in search of the key. The contains () and remove () work similarly. You can find the details on Blackboard.

Because of the need for linear scans, `put()`, `get()`, `contains()`, and `remove()` take $O(n)$ time in the worst case, where n is the size of the map.

Iterators. Note that we have to iterate over the set of keys, not over the key/value pairs. The details are hidden within a `KeyIterator` inner class.

Suppose `m` is the map represented using `list` and `kIter = m.keyIterator()`. Then `kIter` is implemented using an iterator `iter` over `list` (generated using `list.iterator()`). Since `iter` iterates over `MapEntry` objects, a call to `iter.next()` would return a pair. Thus, `kIter.next()` returns `iter.next().key`.


```
public Iterator<K> keyIterator()
{
    return new KeyIterator();
}

private class KeyIterator
implements Iterator<K>
{
    private Iterator<MapEntry> iter =
        list.iterator();

    @Override
    public boolean hasNext()
    {
        return iter.hasNext();
    }

    @Override
    public K next()
    {
        return iter.next().key;
    }

    @Override
    public void remove()
    {
        iter.remove();
    }
}
```

Binary Search Tree Implementation

If the keys have a natural ordering, we can represent maps using a BST consisting of `MapEntry` objects. To make `MapEntry` objects comparable by key, we need to override `compareTo()` appropriately. You will find the details in the class `BSTMap`, posted on Blackboard.

On a BST implementation, the running times of `get()`, `put()`, `containsKey()`, and `remove()` are all $O(\text{height}(T))$, where T is the underlying tree. Therefore, if we use **balanced** BSTs, the running time of all those methods is $O(\log n)$. In fact, Java's `TreeMap` class¹ uses red-black trees (a kind of balanced BST) to represent `Map` objects. In Project 5, you will implement a simplified version of `Map` that uses " α -balanced" trees as the backing store; in this implementation `get()`, `put()`, `containsKey()`, and `remove()` take $O(\log n)$ **amortized** time. You can also achieve $O(\log n)$ amortized time using splay trees.

¹ <https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html>