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*.

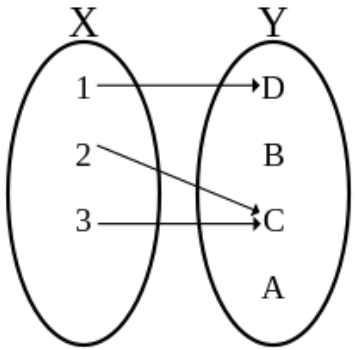
We say that the key is *mapped* to the value.

- 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).

```java
// A class implementing a key-value pair.  This class associates an
// immutable key with a mutable value.  Used in many other structures.
// Example Usage:
//      Association [] classesTaken = new Association[5];
//      classesTaken[0] = new Association("Barbara", new Integer(1));
public class Association<K,V> implements Map.Entry<K,V> {
    // The immutable key.  An arbitrary object.
    protected K theKey;// the key of the key-value pair

    // The mutable value.  An arbitrary object.
    protected V theValue;// the value of the key-value pair

    // Constructs a pair from a key and value.
    public Association(K key, V value) {
        Assert.pre(key != null, "Key must not be null.");
        theKey = key;
        theValue = value;
    }
```

`Association.java` in the `structure` package.



A diagram of a function or map.
The element 2 is mapped to C.
In other words, 2 is a key, and C is its 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.

A	D	C	C	B	B
0	1	2	3	4	5

An array is a specific type of map.

keys	values
Williams College	Ephs 
MCLA Massachusetts College of Liberal Arts	Trailblazers 
Simon's Rock Bard College at Simon's Rock	Llamas 
BCC Berkshire Community College	Falcons 

A map of Berkshire colleges to their mascots.

`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.

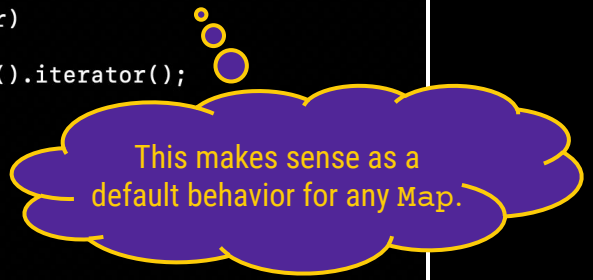
```java
// Associations establish a link between a key and a value.
// An associative array or map is a structure that allows a disjoint
// set of keys to become associated with an arbitrary set of values.
public interface Map<K,V>
{
    // @post returns the number of entries in the map
    public int size();

    // @post returns true iff this map does not contain any entries
    public boolean isEmpty();

    // @pre k is non-null
    // @post returns true iff k is in the domain of the map
    public boolean containsKey(K k);

    // @pre v is non-null
    // @post returns true iff v is the target of at least one map entry;
    // that is, v is in the range of the map
    public boolean containsValue(V v);

    // @pre k is a key, possibly in the map
    // @post returns the value mapped to from k, or null
    public V get(K k);

    // @pre k and v are non-null
    // @post inserts a mapping from k to v in the map
    public V put(K k, V v);

    // @pre k is non-null
    // @post removes any mapping from k to a value, from the mapping
    public V remove(K k);
```

```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);

    // @post removes all map entries associated with this map
    public void clear();

    // @post returns a set of all keys associated with this map
    public Set<K> keySet();

    // @post returns a structure that contains the range of the map
    public Structure<V> values();

    // @post returns a set of (key-value) pairs, generated from this
    public Set<Association<K,V>> entrySet();

    // @pre other is non-null
    // @post returns true iff maps this and other are entry-wise equ
    public boolean equals(Object other);

    // @post returns a hash code associated with this structure
    public int hashCode();
}
```

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.

```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();
    }
}
```




```java
// This could also be added
// to the AbstractMap class.
public boolean isEmpty()
{
    return size() == 0;
}
```

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`.

```
~/GitLohani/js/src/structure5$ grep Map *.java
AbstractMap.java: * of different pieces of information simultaneously.  Maps are sometimes
AbstractMap.java:public abstract class AbstractMap<K,V> implements Map<K,V>
AbstractMap.java:    public void putAll(Map<K,V> other)
Association.java:import java.util.Map;
Association.java:public class Association<K,V> implements Map.Entry<K,V>
ChainedHashtable.java:public class ChainedHashtable<K,V> extends AbstractMap<K,V> implements Map<K,V>, Iterable<V>
ComparableAssociation.java:import java.util.Map;
ComparableAssociation.java:    , Map.Entry<K,V>
Entry.java:import java.util.Map;
Entry.java: * An implementation of the the java.util.Map.Entry interface, Entry
Entry.java: * also implement the Map interface and have expanded functionality.
Entry.java:public class Entry<K,V> implements java.util.Map.Entry<K,V>
Entry.java:        Map.Entry<?,?> otherEntry = (Map.Entry<?,?>)other;
GraphList.java:     * Map associating vertex labels with vertex structures.
GraphList.java:    protected Map<V,GraphListVertex<V,E>> dict; // label -> vertex
GraphListEIterator.java:    public GraphListEIterator(Map<V,GraphListVertex<V,E>> dict)
GraphMatrix.java:    protected Map<V,GraphMatrixVertex<V>> dict;   // labels -> vertices
Hashtable.java:public class Hashtable<K,V> implements Map<K,V>, Iterable<V>
Hashtable.java:    public void putAll(Map<K,V> other)
Hashtable.java:     * @post returns a set of Associations associated with this Map
Hashtable.java:     * @post returns a Set of keys used in this Map
Map.java: * of different pieces of information simultaneously.  Maps are sometimes
Map.java: *      Map dict = new {@link structure.MapBST#MapBST()};
Map.java: *          dict.{@link structure.MapBST#put(Object,Object) put(word,def)};
Map.java:public interface Map<K,V>
Map.java:    public void putAll(Map<K,V> other);
MapList.java:import java.util.Map.Entry;
MapList.java: * of different pieces of information simultaneously.  Maps are sometimes
MapList.java: *      Map dict = new {@link #MapList()};
MapList.java:public class MapList<K,V> implements Map<K,V>
MapList.java:    public MapList()
MapList.java:    public MapList(Map<K,V> source)
MapList.java:    public void putAll(Map<K,V> other)
MapList.java:        MapList<?,?> that = (MapList<?,?>)other;
OrderedMap.java: * An interface the supports a Map whose values are kept
OrderedMap.java: * in increasing order.  Values stored within an OrderedMap
OrderedMap.java: * @version $Id: OrderedMap.java 35 2007-08-09 20:38:38Z bailey $
OrderedMap.java:public interface OrderedMap<K extends Comparable<K>,V> extends Map<K,V>
Table.java:import java.util.Map.Entry;
Table.java: *      {@link OrderedMap} dict = new {@link #Table()};
Table.java:public class Table<K extends Comparable<K>,V> extends AbstractMap<K,V> implements OrderedMap<K,V>
Table.java:        OrderedMap<String,String> dict = new Table<String,String>();
```

`Map` is used extensively in the `structure` package.

- We'll discuss OrderedMap

```
/**
 * @pre k is a key, possibly in the map
 * @post returns the value mapped to from k, or null
 */
public V get(K k);

/**
 * @pre k and v are non-null
 * @post inserts a mapping from k to v in the map
 */
public V put(K k, V v);

/**
 * @pre k is non-null
 * @post removes any mapping from k to a value, from the mapping
 */
public V remove(K k);
```

```
/**
 * Enter a key-value pair into the table.  if the key is already
 * in the table, the old value is returned, and the old key-value
 * pair is replaced.  Otherwise null is returned.  The user is cautioned
 * that a null value returned may indicate there was no prior key-value
 * pair, or --- if null values are inserted --- that the key was
 * previously associated with a null value.
 *
 * @pre key is non-null object
 * @post key-value pair is added to table
 *
 * @param key The unique key in the table.
 * @param value The (possibly null) value associated with key.
 * @return The prior value, or null if no prior value found.
 */
public V put(K key, V value)
```

If the key `K` is currently in the `Map`, then return its current value; otherwise, return `null`.

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?

Think to yourself for 1 minute.
Discuss with a neighbor for 2 minutes.

	`get`	`put`
array	O(n)-time	O(n)-time
vector	O(n)-time	O(n)-time
linked list	O(n)-time	O(n)-time

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

`MapList`

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.

A map stored in a singly linked list as in `MapList`.
Each node stores both a key and value.
Hence, `BCC` is mapped to `Falcons`.

`get`	`put`	`remove`	`containsKey`	`containsValue`
O(n)-time	O(n)-time	O(n)-time	O(n)-time	O(n)-time

The worst-case run-times of various operations when implementing a map using a singly linked list as in `MapList`.
The number of (key, value) pairs currently in the map is `n`.

The run-times of these operations are **Ω**(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.

```
public class MapList<K,V> implements Map<K,V> {

    // List for storing the entries in this map
    protected List<Association<K,V>> data;

    // Construct an empty map, based on a list
    public MapList() {
        data = new SinglyLinkedList<Association<K,V>>();
    }

    // Construct a map with values found in source
    public MapList(Map<K,V> source) {
        this();
        putAll(source);
    }

    // Returns the number of entries in the map
    public int size() {
        return data.size();
    }

    // @post returns true iff this map does not contains any entries
    public boolean isEmpty() {
        return data.isEmpty();
    }

    // @pre k is non-null
    // @post returns true iff k is a key that is mapped to a value;
    //  that is, k is in the domain of the map
    public boolean containsKey(K k) {
        return data.contains(new Association<K,V>(k,null));
    }
```

```
// @pre v is non-null
// @post returns true iff v is the target of at least one map entry;
// that is, v is in the range of the map
public boolean containsValue(V v) {
    Iterator<V> i = new ValueIterator<K,V>(data.iterator());
    while (i.hasNext())
    {
        V value = i.next();
        if (value != null &&
            v.equals(value)) return true;
    }
    return false;
}

// @pre k is a key, possibly in the map
// @post returns the value mapped to from k, or null
public V get(K k) {
    int i = data.indexOf(new Association<K,V>(k,null));
    if (i >= 0) return data.get(i).getValue();
    return null;
}

// @pre k and v are non-null
// @post inserts a mapping from k to v in the map
public V put(K k, V v) {
    Association<K,V> temp = new Association<K,V>(k,v);
    Association<K,V> result = data.remove(temp);
    data.add(temp);
    if (result == null) return null;
    else return result.getValue();
}
```

`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`.

```java
// @pre k is non-null
// @post removes any mapping from k to a value, from the mapping
public V remove(K k) {
    Association<K,V> v = data.remove(new Association<K,V>(k,null));
    if (v == null) return null;
    else return v.getValue();
}

// @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());
    }
}

// @post removes all map entries associated with this map
public void clear() {
    data.clear();
}

// @post returns a set of all keys associated with this map
public Set<K> keySet() {
    Set<K> result = new SetList<K>();
    Iterator<Association<K,V>> i = data.iterator();
    while (i.hasNext())
    {
        Association<K,V> a = i.next();
        result.add(a.getKey());
    }
    return result;
}
```

```java
// @post returns a structure that contains the range of the map
public Structure<V> values() {
    Structure<V> result = new SinglyLinkedList<V>();
    Iterator<V> i = new ValueIterator<K,V>(data.iterator());
    while (i.hasNext())
    {
        result.add(i.next());
    }
    return result;
}

// @post returns a set of (key-value) pairs, generated from this map
public Set<Association<K,V>> entrySet() {
    Set<Association<K,V>> result = new SetList<Association<K,V>>();
    Iterator<Association<K,V>> i = data.iterator();
    while (i.hasNext())
    {
        Association<K,V> a = i.next();
        result.add(a);
    }
    return result;
}

// @pre other is non-null
// @post returns true iff maps this and other are entry-wise equal
public boolean equals(Object other) {
    MapList<?,?> that = (MapList<?,?>)other;
    return data.equals(that.data);
}

// @post returns a hash code associated with this structure
public int hashCode() {
    return data.hashCode();
}
```

`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`.

```
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`.

```
// @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?