

Maps

Lecturer: Dr Anwarul Patwary

Outline

- ▶ What is a map?
- ▶ Map representations

What is a map?

- ▶ Often in programs, we'll want to be able to look up some data, using another piece of data.
- ▶ For instance –
 - ▶ we may have a record of students and their marks for an exam, and want to look up a particular *mark*, given the name of a particular *student*
 - ▶ or we might have a record of cities and their populations, and want to look up what the population is, given the name of a particular city.
- ▶ When talking about maps, the thing we want to *retrieve* – the mark, in the first example – is called a *value*; and the input we provide in order to retrieve it is called a *key* (the student name, in this example).

What is a map?

- ▶ Mathematically, we can consider a map to be a function from one domain (the domain of keys) to another (the domain of values).
- ▶ In fact, when talking about functions, we often describe their domain and codomain by saying the function “maps from” the domain to the codomain.

Domain and codomain

As an example, consider a function $isEven(n)$, which takes a single natural number n , and tells us whether n is even or odd.

We would say the *domain* of the function is the natural numbers, and the *codomain* is the booleans (true and false).

<hr/>	
n	isEven(n)
<hr/>	
0	true
1	false
2	true
3	false
\vdots	\vdots

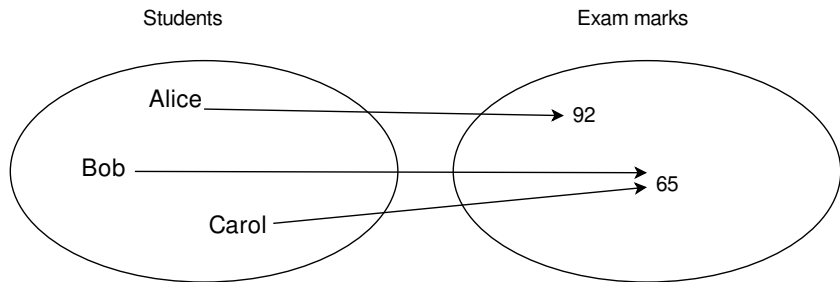
domain = {0, 1, 2, 3...}

codomain = {true,false}

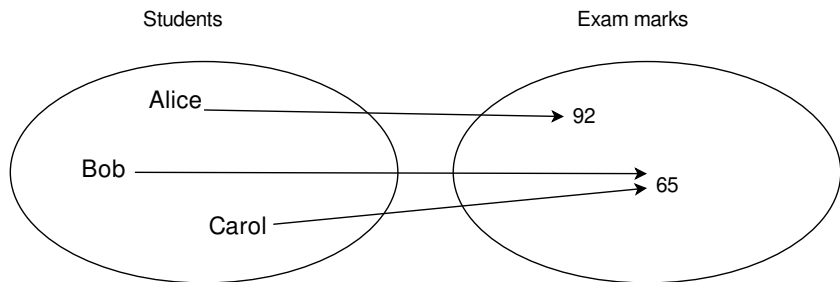
Finite, many-to-one functions

When considering them as data structures, we will normally be concerned with maps where the set of keys is *finite*.

Additionally, maps represent the sort of function called “many to one” – multiple keys may map to the same value.



Terminology



We use the word “image” to refer to “the thing mapped to”.

So we say that “92 is the image of Alice”.

Map ADT

As an abstract data type, the operations we typically want a map to support include:

- ▶ *isEmpty()*: return *true* if the map is empty, *false* otherwise.
- ▶ *isDefined(k)* or *hasKey(k)*: return *true* if the key *k* is defined in the map, *false* otherwise.
- ▶ *assign(k,v)* or *set(k,v)*: assign the value *v* as the image of the key *k*.
- ▶ *image(k)* or *get(k)*: return the image of the key *k* if it is defined.

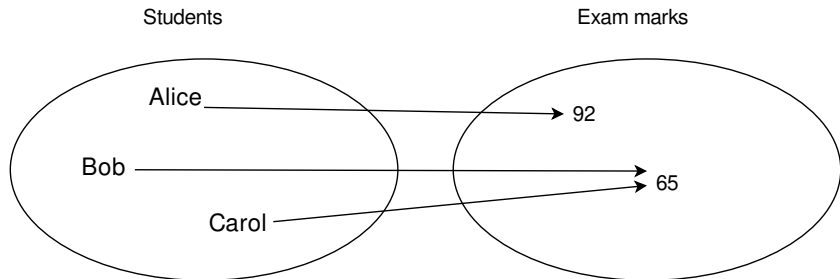
Otherwise we might throw an exception, or return a special value to (like `null`) to indicate the key is not present.

- ▶ Note that if we return `null`, then it is impossible to distinguish between the case where the key doesn't appear in the map, and the case where it *does* appear, but maps to the `null` value.
- ▶ Nevertheless, this is what the Java implementation of Map does.
- ▶ *deassign(k)* or *remove(k)*: if the image of the key *k* is defined, make it undefined.

Representing maps with linked lists

We can consider a map to be a list of *pairs* of things.

For instance, this map:



could be represented as the three-item list:

`[(Alice, 92), (Bob, 65), (Carol, 65)]`

Linked list implementation in Java

In order to implement a map in this way, we need a class to represent a key-value pair.

We can do this as follows:

```
class Pair<K,V> {  
    K key;  
    V value;  
  
    public Pair (K key, V value) {  
        this.key = key;  
        this.value = value;  
    }  
}
```

Linked list implementation in Java

Given a Pair class, we can now start to define an implementation of the Map ADT based on linked lists:

```
public class MapLinked<K,V> {  
    private ListNode< Pair<K,V> > list;  
  
    // ... methods will go here ...
```

Linked list implementation in Java

We can then implement the methods of our map using operations on a linked list:

- ▶ `isEmpty()` by checking whether the `list` is null
- ▶ `hasKey(k)` by doing sequential search for the key
- ▶ `get(k)` by doing sequential search for the key
- ▶ `set` by doing sequential search for the key, and changing the value (or, if the key is not found, creating a new `ListNode` containing that key–value pair)
- ▶ `remove` by doing sequential search for the key, and removing that node from the list.

Linked list performance

However, the performance is not especially good – most operations will have $O(n)$ complexity (where n is the number of keys in the map).

Operation	Complexity
<i>isEmpty</i>	$O(1)$
<i>hasKey</i>	$O(n)$
<i>get</i>	$O(n)$
<i>set</i>	$O(n)$
<i>remove</i>	$O(n)$

Binary search tree implementation

A more efficient way of implementing the map ADT is to use *binary search trees*.

If we want to do this, our keys and pairs must implement the interface `Comparable` – which lets us compare two things to see if one is less than or greater than the other.

We change the start of our `Pair` class slightly:

```
class Pair<K extends Comparable<? super K>,V> implements Comparable {
    K key;
    V value;

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

    // methods go here
```

The line at the start of the class means “Whatever type *K* you use for a key must be something you can *compare with* another key; and you can compare key–value pairs”.

Making pairs comparable

And we add a `compareTo` method to our `Pair`:

```
@Override
public int compareTo(Pair<K,V> other) {
    return this.key.compareTo ( other.key );
}

} // end of class
```

BST implementation

Now we can implement the methods of our map ADT using operations on a binary search tree:

- ▶ `isEmpty()` just calls `BinarySearchTree.isEmpty()`
- ▶ `hasKey(k)` calls `BinarySearchTree.find()`, and checks whether the result is `null`
- ▶ `get(k)` calls `BinarySearchTree.find()`, and returns the value from the resulting `BinaryTreeNode`
- ▶ `set(k,v)` calls `BinarySearchTree.find()` to see if the key is already defined.

If it is, we set the value of the resulting pair; if it is not, we call `BinarySearchTree.insert()` to insert a new pair.

- ▶ `remove(k)` by calling `BinarySearchTree.remove()`.

BST implementation

The Java code looks like this:

```
public class MapBST<K,V> {
    private BinarySearchTree< Pair<K,V> > bst;

    public boolean isEmpty()      { return bst.isEmpty(); }
    public boolean hasKey(K key)  {
        return
            bst.find( new Pair(key, null) ) == null;
    }
    public V get(K key)           { return
        bst.find( new Pair(key, null) ).value;
    }
    public V set(K key, V value)  { Pair<K,V> result =
        bst.find( new Pair(key, null) );
        if (result == null) {
            bst.insert( new Pair(key, value) );
        } else {
            result.value = new Pair(key, value);
        }
    }
    public V remove(K key)        { bst.remove( new Pair(key, null) );
    }
    // ... constructor and other methods go here ...
}
```

BST implementation performance

The performance now is better – many operations will now have $O(\log n)$ complexity, since that is the complexity of the underlying BST operations.

Operation	Complexity
<i>isEmpty</i>	$O(1)$
<i>hasKey</i>	$O(\log n)$
<i>get</i>	$O(\log n)$
<i>set</i>	$O(\log n)$
<i>remove</i>	$O(\log n)$

In fact, this is more or less how the `TreeMap` class in the Java Collections framework is defined.