Solutions to Exercises in Chapter 11

11.2 Figure S11.1 shows how the maps *Roman*, *NAFTA*, and *EC* would be represented by arrays with *maxcard* = 10, and how the maps of Figure 12.3 would be represented by arrays with *maxcard* = 12.

11.3 Figure S11.2 shows how the maps of Exercise 11.2 would be represented by SLLs.

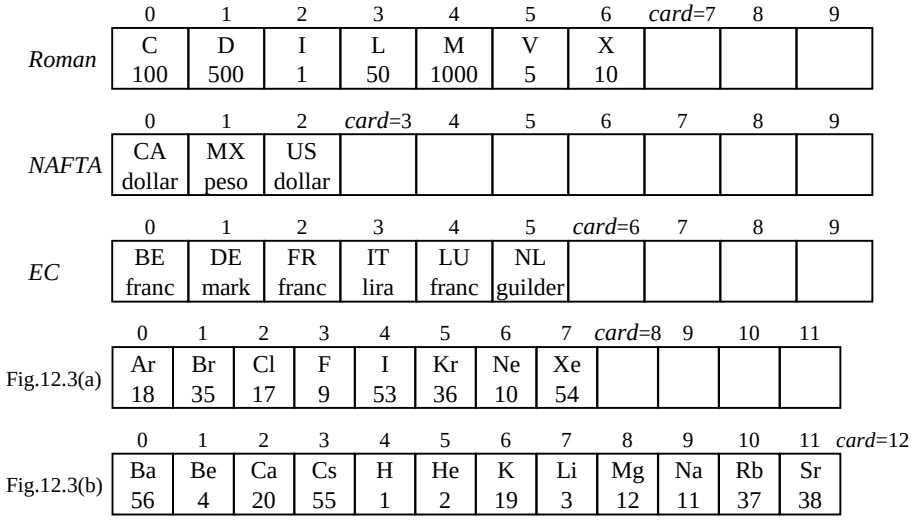


Figure S11.1 Representation of maps using arrays (with *maxcard* = 10 or 12).

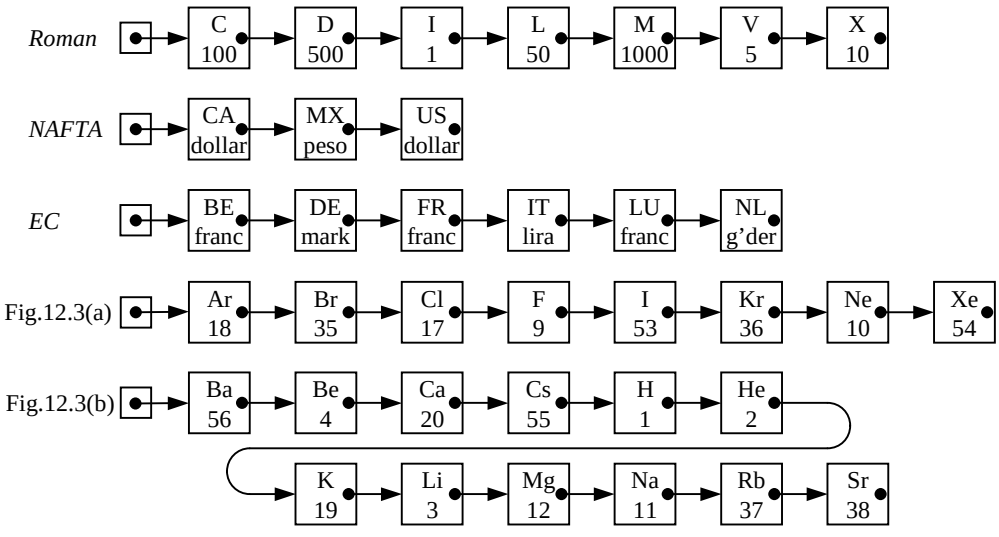


Figure S11.2 Representation of maps using SLLs.

11.3 Figure S11.3 shows how the maps of Exercise 11.2 might be represented by BSTs. (The exact shape of each BST depends on the order in which the entries were added.)

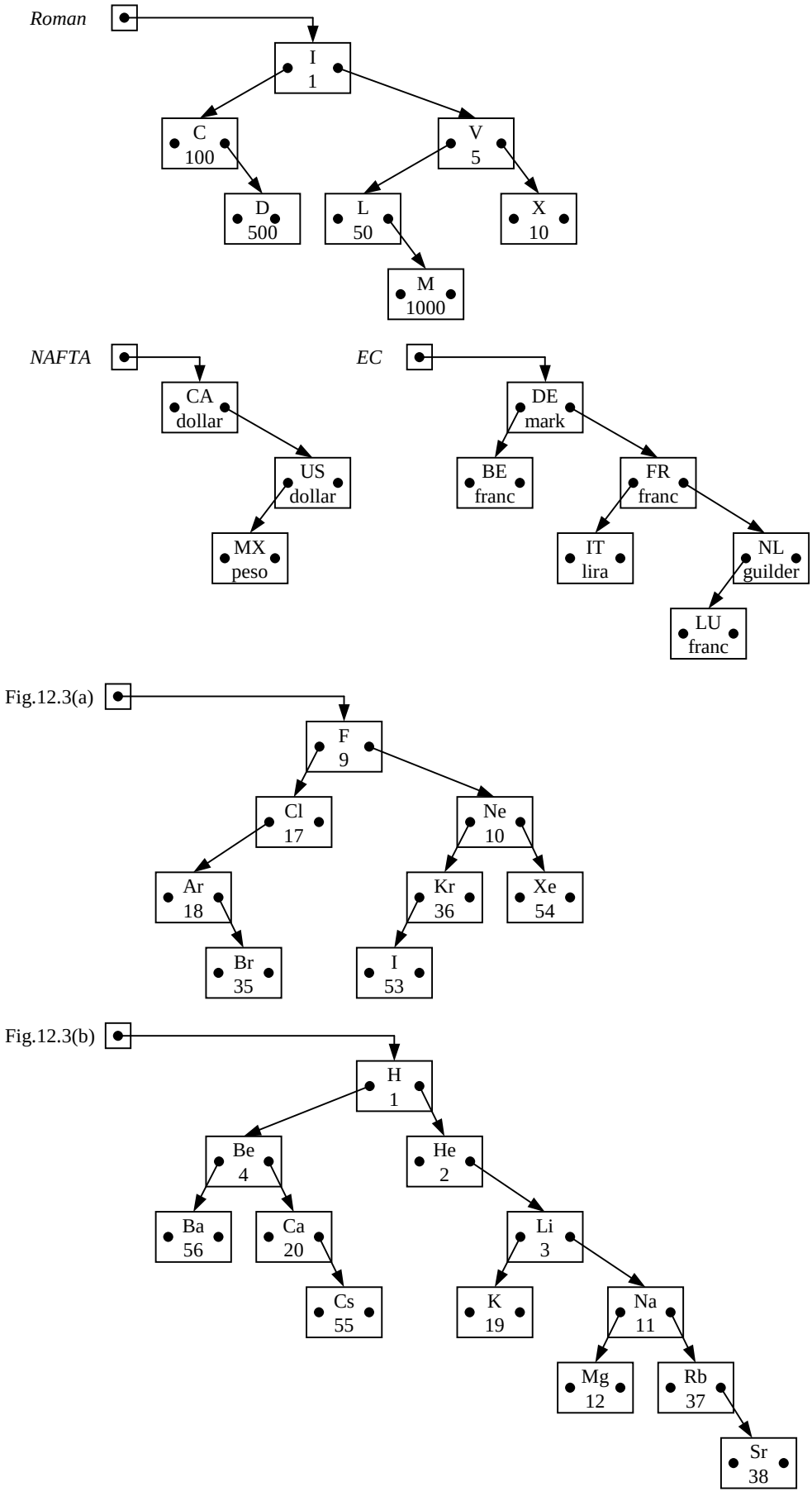


Figure S11.3 Representation of maps using BSTs.

11.5 Alternative `Map` interface in which mutative transformers are replaced by applicative transformers:

```
public interface Map {

   // Each Map object is a map in which the keys and values are
   // arbitrary objects.

   ////////////// Accessors //////////////

   …

   ////////////// Transformers //////////////

   …

   public Map remove (Object key);
   // Return the new map obtained by removing the entry with key
   // (if any) from this map.

   public Map overlay1 (Object key, Object val);
   // Return the new map obtained by overlaying this map with the
   // single entry ‹key, val›. (Throw a ClassCastException if
   // this map cannot contain a key with the class of key.)

   public Map overlay (Map that);
   // Return the new map obtained by overlaying this map with that.

}
```

Implement the `remove` operation in the `ArrayMap` class as follows:

```
public Map remove (Object key) {
   ArrayMap that = this.clone();
   if (key instanceof Comparable) {
      int pos = that.search((Comparable) key);
      if (key.equals(that.entries[pos].key)) {
         for (int i = pos+1; i < that.card; i++)
            that.entries[i-1] = that.entries[i];
         that.entries[--that.card] = null;
      }
   }
   return that;
}
```

This assumes that `ArrayMap` is made to implement the `Cloneable` interface. Implement the `overlay1` and `overlay` operations similarly by first creating a clone of this map and then updating the clone accordingly.

Implement the applicative transformers in the `LinkedMap` and `BSTMap` classes in similar fashion.

11.6 The complete implementation of the `IntKeyMap` class can be found at the companion Web site.

11.7 If bounded maps were represented by *unsorted* arrays, the modifications to the Java implementation would be straightforward.

The algorithms and their complexities are summarized in Table S11.4.

Table S11.4 Implementation of bounded maps using unsorted arrays: summary of algorithms (where n, n_1, and n_2 are the cardinalities of the maps involved).

Operation	Algorithm	Time complexity
`get`	linear search	$O(n)$
`remove`	linear search plus deletion	$O(n)$
`put`	linear search plus insertion	$O(n)$
`equals`	repeated linear search	$O(n_1 n_2)$
`putAll`	repeated insertions	$O(n_1 n_2)$

11.10 An array or SLL representation would be suitable for a map of font names to fonts, since the number of entries is likely to be small.

If a font is identified by a font name, a size, and a style, we could introduce a simple class `Typeface` containing this information:

```
public class Typeface {
   public String fontName;
   public int size;
   public String style;
}
```

and then create a map whose keys are `Typeface` objects and whose values are fonts.

Alternatively, we could create a map whose keys are font names and whose values are themselves maps; in each of these maps the keys are sizes and the values are again maps; in each of the latter maps the keys are styles and the values are fonts.

11.11 The `java.util.Set` and `java.util.TreeSet` classes share the same tree representation, in which each node has a key field and a value field. In the representation of a set, the key fields contain the members and the value fields are ignored. The set operations are implemented in terms of map operations as shown in Table S11.5. (Some trivial operations are omitted.)

The main advantage is to avoid duplication of (rather complex) code.

The main disadvantage is waste of space in the representation of a set: the value fields are redundant.

Table S11.5 Set operations implemented in terms of map operations (where set `s` is represented by map `m`).

Set operation	Map operation
`s.contains(x)`	`m.containsKey(x)`
`s.equals(s2)`	`m.equals(m2)`
`s.containsAll(s2)`	Traverse `m2`, and for each key `x` test `m.containsKey(x)`.
`s.add(x)`	`m.put(x, null)`
`s.remove(x)`	`m.remove(x)`
`s.addAll(s2)`	`m.putAll(m2)`
`s.removeAll(s2)`	Traverse `m2`, and for each key `x` call `m.remove(x)`.
`s.retainAll(s2)`	Traverse `m`, and for each key `x` such that `m2.containsKey(x)` returns false call `m.remove(x)`.
s`.iterator()`	`m.keySet().iterator()`

11.12 We can represent a list of length *n* by a map whose keys are `Integer` objects, ranging from `Integer(0)` through `Integer(`*n*–1`)`. The list operations are then implemented by map operations as shown in Table S11.6.

The time complexities of the list operations depend on the underlying map representation. Table S11.6 assumes a well-balanced BST representation. Even with that optimistic assumption, this is clearly a very inefficient representation of lists.

Table S11.6 List operations implemented in terms of map operations (where list `l` is represented by map `m`).

List operation	Map operation	Time complexity (well-balanced BST)
`l.get(i)`	`m.get(new Integer(i))`	$O(\log n)$
`l.equals(l2)`	`m.equals(m2)`	$O(n_2 \log n)$
`l.set(i, x)`	`m.put(new Integer(i), x)`	$O(\log n)$

`l.add(i, x)`	`for (int k = m.size()-1; k>=i; k--)`
	`  m.put(new Integer(k+1),`
	`  m.remove(new Integer(k)));`
	`m.put(new Integer(i), x)`
		$O(n)$

`l.add(x)`	`m.put(new Integer(m.size()), x)`	$O(\log n)$

`l.addAll(l2)`	`int n = m.size();`
	`for (int k = 0; k < m2.size(); k++)`
	`  m.put(new Integer(n+k),`
	`  m2.get(new Integer(k)));`
		$O(n_2 \log n)$

`l.remove(i)`	`m.remove(new Integer(i))`	$O(\log n)$

11.13 Enhanced `Map` interface:

```
public interface Map {

  // Each Map object is a map in which the keys and values are
  // arbitrary objects.

  //////////// Accessors ////////////

  …

  public boolean containsKey (Object key);
  // Return true if and only if this map contains an entry whose key
  // is key.

  public boolean containsValue (Object val);
  // Return true if and only if this map contains an entry whose value
  // is val.

  //////////// Transformers ////////////

  …

}
```

Implementation of these operations in the `ArrayMap` class:

```
public boolean containsKey (Object key) {
  return (get(key) != null);
}
```

```
public boolean containsValue (Object val) {
  for (int p = 0; p < card; p++) {
    if (val.equals(entries[p].value))
      return true;
  }
  return false;
}
```

(*Note:* The `containsKey` operation can use binary search, but the `containsValue` operation must use linear search.)

Implementation of these operations in the `LinkedMap` class:

```
public boolean containsKey (Object key) {
  return (get(key) != null);
}

public boolean containsValue (Object val) {
  for (LinkedMap.Entry curr = first;
       curr != null; curr = curr.succ) {
    if (val.equals(curr.value))
      return true;
  }
  return false;
}
```

Implementation of these operations in the `BSTMap` class:

```
public boolean containsKey (Object key) {
  return (get(key) != null);
}

public boolean containsValue (Object val) {
  return subtreeContainsValue(root, val);
}

private static boolean subtreeContainsValue (
              BSTMapEntry top, Object val) {
  // Return true if and only if the subtree whose topmost node is top
  // contains an entry whose value is val.
  if (top == null)
    return false;
  else if (val.equals(top.value))
    return true;
  else
    return subtreeContainsValue(top.left, val)
      || subtreeContainsValue(top.right, val);
}
```

11.14 Enhanced `Map` interface:

```
public interface Map {

  // Each Map object is a map in which the keys and values are
  // arbitrary objects.

  //////////// Accessors ////////////

  …

  //////////// Transformers ////////////

  …

  public Map subMap (Object key1, Object key2);
  // Return a map containing just those entries of this map whose keys
  // are not less than key1 and not greater than key2.
```

```
}
```

Implementation of this operation in the `ArrayMap` class:

```
public Map subMap (Object key1, Object key2) {
  Comparable k1 = (Comparable) key1,
       k2 = (Comparable) key2;
  int p1 = search(k1), p2 = search(k2);
  if (! k2.equals(entries[p2].key))  p2--;
  Map that = new ArrayMap(p2-p1+1);
  for (int p = p1; p <= p2; p++)
    that.put(entries[p].key, entries[p].value);
  return that;
}
```

Implementation of this operation in the `LinkedMap` class:

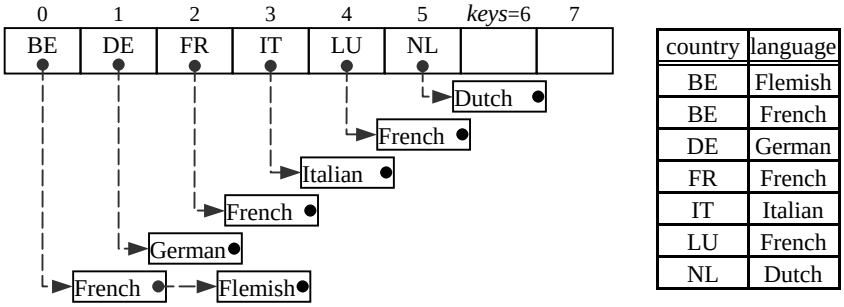
```
public Map subMap (Object key1, Object key2) {
  Map that = new LinkedMap();
  Comparable k1 = (Comparable) key1,
       k2 = (Comparable) key2;
  if (k1.compareTo(k2) >= 0) {
    int comp1 = -1, comp2 = -1;
    for (LinkedMap.Entry curr = first;
         curr != null; curr = curr.succ) {
      if (comp1 < 0)
        comp1 = curr.key.compareTo(k1);
      if (comp1 >= 0) {
        comp2 = curr.key.compareTo(k2);
        if (comp2 > 0)  break;
        that.put(curr.key, curr.value);
      }
    }
  }
  return that;
}
```

Implementation of this operation in the `BSTMap` class:

```
public Map subMap (Object key1, Object key2) {
  Map that = new BSTMap();
  Comparable k1 = (Comparable) key1,
       k2 = (Comparable) key2;
  if (k1.compareTo(k2) >= 0)
    addFromSubtree(root, k1, k2, that);
  return that;
}
```

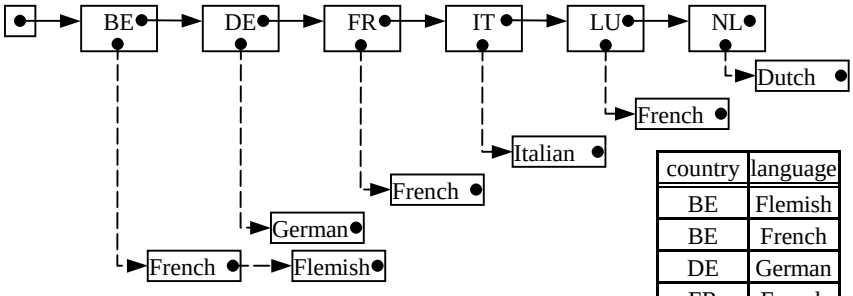
```
private static void addFromSubtree (
                BSTMapEntry top,
                Comparable k1, Comparable k2
                Map that) {
  // In the subtree whose topmost node is top, add to that just
  // those entries whose keys are not less than k1 and not greater than
  // k2.
  if (top == null)
    return;
  else {
    int comp1 = top.key.compareTo(k1),
        comp2 = top.key.compareTo(k2);
    if (comp1 >= 0 && comp2 <= 0)
      that.add(top.key, top.value);
    if (comp1 > 0)
      addFromSubtree(top.left, k1, k2, that);
    if (comp2 < 0)
      addFromSubtree(top.left, k1, k2, that);
  }
}
```

11.15 We can represent a multimap in the same manner as an ordinary map, except that each key is associated with a *set* of values. Figures S11.7, S11.8, and S11.9 illustrate sorted array, sorted SLL, and BST representations of a multimap. In each case the set of values associated with a particular key is represented by an unsorted SLL, which is adequate if it can be assumed that multiple entries with the same key will be relatively uncommon.



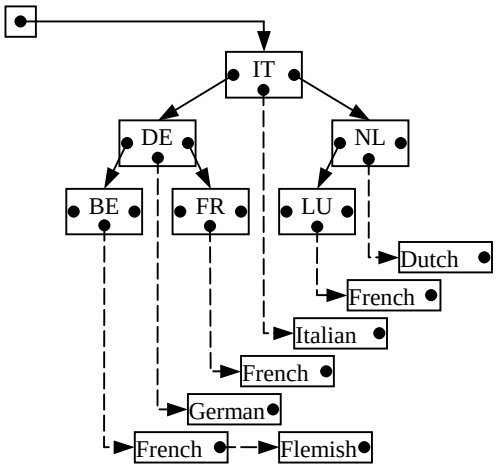
country	language
BE	Flemish
BE	French
DE	German
FR	French
IT	Italian
LU	French
NL	Dutch

Figure S11.7 Representation of a multimap using a sorted array.



country	language
BE	Flemish
BE	French
DE	German
FR	French
IT	Italian
LU	French
NL	Dutch

Figure S11.8 Representation of a multimap using a sorted SLL.



country	language
BE	Flemish
BE	French
DE	German
FR	French
IT	Italian
LU	French
NL	Dutch

Figure S11.9 Representation of a multimap using a BST.