Solutions to Exercises 9

9A The set of primary colors, the set of colors of the rainbow, and the set of colors in the national flag are:

primary = {red, green, blue}
rainbow = {red, orange, yellow, green, blue, indigo, violet}
flag = {red, white, blue} (in the UK flag, for example)

Their sizes are:

#primary = 3
#rainbow = 7
#flag = 3

The set of rainbow colors in the national flag is *rainbow* ∩ *flag*, which evaluates to {red, blue}.

The set of rainbow colors not in the national flag is *rainbow* – *flag*, which evaluates to {orange, yellow, green, indigo, violet}.

The assertion that all colors in the national flag occur in the rainbow is *rainbow* ⊇ *flag*, which evaluates to false.

The assertion that the national flag contains exactly the primary colors is *primary* = *flag*, which evaluates to false.

9B Removing step 2.1 of Eratosthenes' sieve algorithm would cause it to remove multiples of an integer, *i*, that is not a member of the set *sieve*. This could happen only if *i* was previously removed as a multiple of some smaller integer, say *j*. In this case, all multiples of *j* will have already been removed, since any multiple of *i* is also a multiple of *j*. So removing step 2.1 will only cause the algorithm to remove integers from *sieve* that have already been removed (which is a harmless but time-consuming operation).

9D Alternative `Set<E>` interface in which the mutative transformers are replaced by applicative transformers:

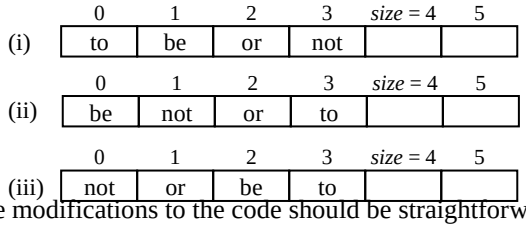
```
public interface Set<E> {
   // Each Set<E> object is a homogeneous set whose members are
   // of type E.
   //////////// Accessors ////////////
   …
   //////////// Transformers ////////////
   public Set<E> empty ();
   // Return an empty set.
   public Set<E> union1 (E it);
   // Return the new set obtained by adding it to this set.
   public Set<E> difference1 (E it);
   // Return the new set obtained by removing it from this set.
   public Set<E> union (Set<E> that);
   // Return the union of this set and that.
   public Set<E> difference (Set<E> that);
   // Return the difference between this set and that.
   public Set<E> intersection (Set<E> that);
   // Return the intersection of this set and that.
   //////////// Iterator ////////////
   …
}
```

Implement the `union1` operation in the `ArraySet<E>` class as follows:

```
public Set<E> union1 (E it) {
   ArraySet<E> that =
        new ArraySet<E>(elems.length+1);
   for (int i = 0; i < this.size; i++)
      that.elems[i] = this.elems[i];
   that.size = this.size;
   int pos = that.search(it);  // binary search
   if (! it.equals(that.elems[pos])) {
      // it is not already a member.
      for (int i = that.size; i > pos; i--)
         that.elems[i] = that.elems[i-1];
      that.elems[pos] = it;
      that.size++;
   }
   return that;
}
```

Implement the other operations similarly by first creating a copy of this set and then modifying the copy accordingly.

9E Here are three possible representations of the set of words {'be', 'not', 'or', 'to'} when the set is represented by an unsorted array:



The modifications to the code should be straightforward.

The algorithms and their complexities are summarized in this table:

Operation	Algorithm	Time complexity
`contains`	linear search	$O(n)$
`equals`	repeated linear search	$O(n_1 n_2)$
`containsAll`	variant of repeated linear search	$O(n_1 n_2)$
`add`	linear search combined with insertion	$O(n)$
`remove`	linear search combined with deletion	$O(n)$
`addAll`	repeated insertions	$O(n_1 n_2)$
`removeAll`	repeated deletions	$O(n_1 n_2)$
`retainAll`	linear search combined with removal	$O(n_1 n_2)$

9F The modifications to remove the instance variable `size` from the array implementation of bounded sets are straightforward. One consequence is that that the `size` operation now has $O(n)$ time complexity. The other consequence is that each operation must either check for null elements in the array, or first determine the set's size and then proceed as before.

9G A suitable representation for sets of ISO-LATIN-1 (8-bit) characters is a boolean array. Each set has at most 256 members, and an array of 256 booleans would not waste too much space. Most sets of characters, however, are likely to be quite small (less than about 50 elements), so `ArraySet<Integer>` might also be suitable.

A suitable representation for sets of Unicode (16-bit) characters is `ArraySet<Integer>`. Since there are 65536 possible characters, a boolean array would be too wasteful of space. (*Note:* A balanced search-tree (§10) or hash-table (§12) would however be more suitable.)

Assuming that `digits` and `letters` represent sets of ISO-LATIN-1 characters, we can write the following expressions:

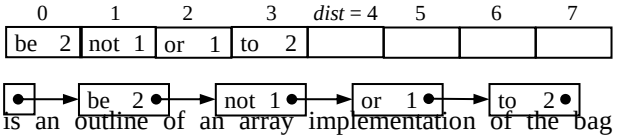
`digits.contains(ch)`	(tests whether `ch` is a digit)
`letters.contains(ch)`	(tests whether `ch` is a letter)
`letters.contains(ch) || digits.contains(ch)`	(tests whether `ch` is a letter or digit)

9H A suitable representation for sets of countries would depend on the anticipated sizes of the sets. If the sets are expected to be small, `ArraySet<Country>` would be reasonably efficient whilst not wasting too much space. If the sizes are large or unknown, neither `ArraySet<Country>` nor `LinkedSet<Country>` would really be suitable. (*Note:* A balanced search-tree (§10) or hash-table (§12) would be suitable.)

9J Here is a possible contract for a bag ADT:

```
public interface Bag<E> {
  // Each Bag object is a homogeneous bag whose members are of type E.
  /////////////  Accessors  /////////////
  public boolean isEmpty ();
  // Return true if and only if this bag is empty.
  public int size ();
  // Return the size of this bag, i.e., the total number of instances.
  public boolean contains (E it);
  // Return true if and only if it is a member of this bag.
  public int instances (E it);
  // Return the number of instances of it in this bag (0 if it is not a
  // member).
  public boolean equals (Bag<E> that);
  // Return true if and only if this bag is equal to that.
  public boolean containsAll (Bag<E> that);
  // Return true if and only if this bag subsumes that (i.e., every member of
  // that has at least as many instances in this bag).
  /////////////  Transformers  /////////////
  public void clear ();
  // Make this bag empty.
  public void add (E it);
  // Add an instance of it to this bag.
  public void remove (E it);
  // Remove an instance of it from this bag.
  public void addAll (Bag<E> that);
  // Make this bag the union of itself and that.
  /////////////  Iterator  /////////////
  public Iterator<E> iterator();
  // Return an iterator that will visit all members of this bag, in no particular
  // order.
}
```

We can represent a bag in the same way as a set, except that each member is paired with a positive integer (which is the number of instances of that member). The following illustrate sorted array and sorted SLL representations of the bag {'to', 'be', 'or', 'not', 'to', 'be'}:



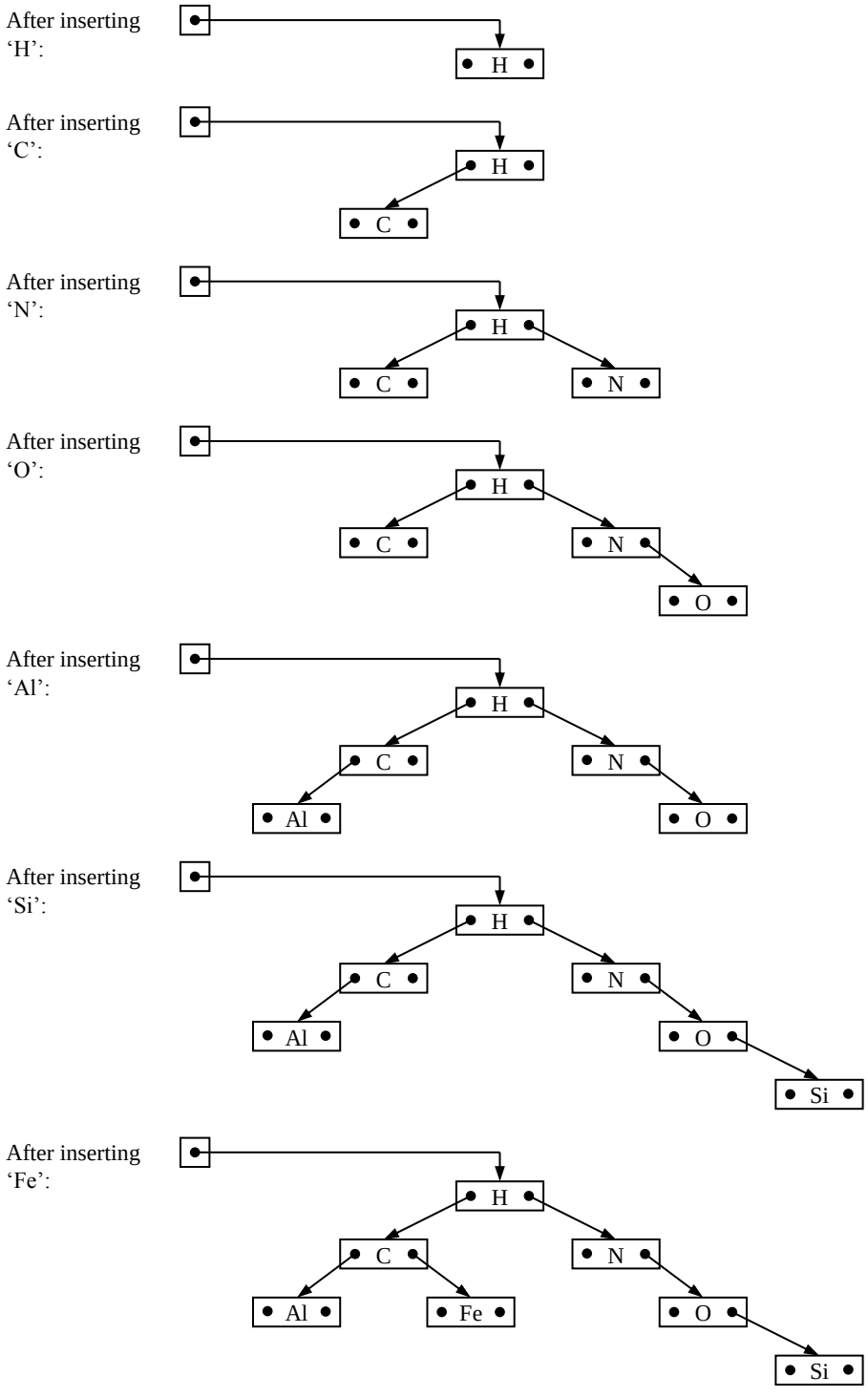
Here is an outline of an array implementation of the bag ADT. (An SLL implementation would be written along similar lines.)

```
public class ArrayBag<E> implements Bag<E> {
   // Each ArrayBag<E> object is a bag whose members are values of type
   // T.
   // This bag is represented as follows. The number of distinct members is
   // held in dist. The members are held in the sorted subarray
   // entries[0...dist–1], each member paired with its number of
   // instances.
   private Entry[] entries;
   private int dist;
   ////////////// Constructor //////////////
   public ArrayBag (int cap) {
      entries = (Entry[]) new Object[cap];
      dist = 0;
   }
   ////////////// Accessors //////////////
   public boolean isEmpty () {
      return (dist == 0);
   }
   public int size () {
      int s = 0;
      for (int i = 0; i < dist; i++)
         s += entries[i].count;
      return s;
   }
   public int instances (E it) {
      int pos = search(it);  // binary search
      if (it.equals(entries[pos].member))
         return entries[pos].count;
      else
         return 0;
   }
   ...
   ////////////// Transformers //////////////
   public void add (E it) {
      int pos = search(it);  // binary search
      if (it.equals(entries[pos].member))
         entries[pos].count++;
      else {
         if (dist == entries.length)  expand();
         for (int i = dist; i > pos; i--)
            entries[i] = entries[i-1];
         entries[pos] = new Entry(it, 1);
         dist++;
      }
   }
   ...
   ////////////// Auxiliary methods //////////////
```

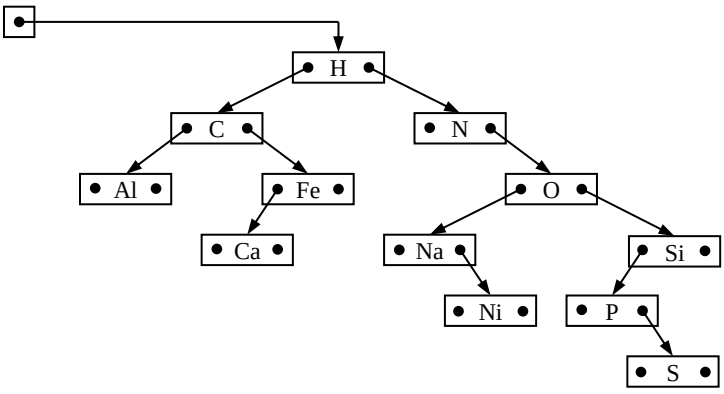
```
  private void expand () { … }
  private int search (E it) { … }
  /////////////  Inner class  /////////////
  private class Entry {
    // Each Entry object is a pair consisting of a bag member and
    // the number of instances of that member.
    private E member;
    private int count;
    private Entry (E it, int n) {
      member = it;  count = n;
    }
  }
}
```

Solutions to Exercises 10

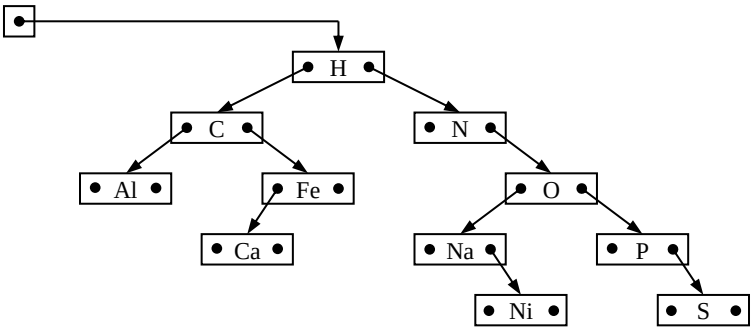
10A The effects of successive additions followed by deletions in a BST of chemical elements are shown below:



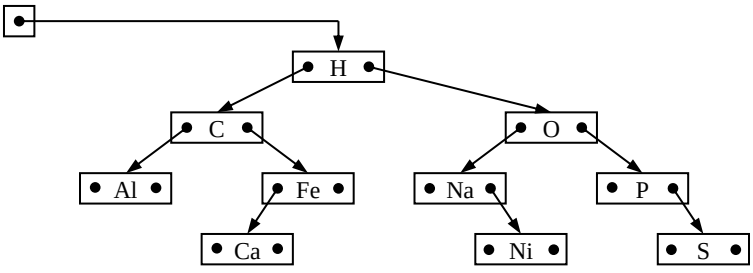
After inserting ‘Na’, ‘P’, ‘S’, ‘Ni’, and ‘Ca’:



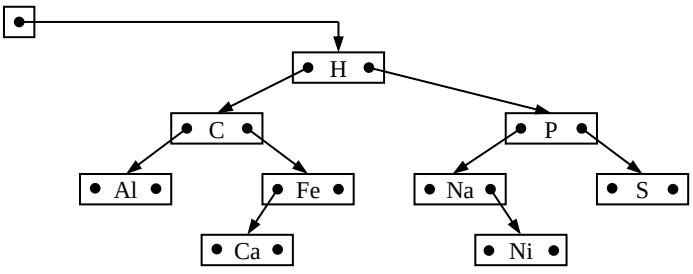
After deleting ‘Si’:



After deleting ‘N’:



After deleting ‘O’:



10B Implementation of the algorithm as a recursive Java method:

```
public static BST.Node search (BST.Node top,
              Comparable target);
// Find which if any node of the subtree whose topmost node is top
// contains an element equal to target. Return a link to that node (or
// null if there is none).
   if (top == null)
     return null;
   else {
     int comp = target.compareTo(top.element);
     if (comp == 0)
       return top;
     else if (comp < 0)
       return search(top.left, target);
     else // comp > 0
       return search(top.right, target);
   }
}
```

10C Implementation of the algorithm as a recursive Java method:

```
public static BST.Node insert (BST.Node top,
              Comparable elem);
// Insert the element elem in the subtree whose topmost node is top.
// Return a link to the modified subtree.
   if (top == null) {
     return new BST.Node(elem, null, null);
   } else {
     int comp = elem.compareTo(top.element);
     if (comp < 0)
       top.left = insert(top.left, elem);
     else if (comp > 0)
       top.right = insert(top.right, elem);
     }
     return top;
   }
}
```

10D To delete the element *elem* from the subtree whose topmost node is *top* (recursive version):

1. If *top* is null:
   1.1. Terminate with answer *top*.
2. If *top* is not null:
   2.1. If *elem* is equal to node *top*'s element:
      2.1.1. Delete the topmost element in the subtree whose topmost node is node *top*, and let *del* be a link to the modified subtree.
      2.1.2. Terminate with answer *del*.
   2.2. Else, if *elem* is less than node *top*'s element:
      2.2.1. Delete *elem* from the subtree whose topmost node is node *top*'s left child, updating *top*'s left child accordingly.
      2.2.2. Terminate with answer *top*.
   2.3. Else, if *elem* is greater than node *top*'s element:
      2.3.1. Deleting *elem* from the subtree whose topmost node is node *top*'s right child, updating *top*'s right child accordingly.
      2.3.2. Terminate with answer *top*.

Implementation of this algorithm as a Java method:

```
public static BST.Node delete (BST.Node top,
              Comparable elem);
// Delete the element elem from the subtree whose topmost node is
// top. Return a link to the modified subtree.
   if (top == null) {
      return top;
   } else {
      int comp = elem.compareTo(top.element);
      if (comp == 0)
         return top.deleteTopmost();
      else if (comp < 0)
         top.left = delete(top.left, elem);
      else // comp > 0
         top.right = delete(top.right, elem);
      }
      return top;
   }
}
```

10E Java methods to return the depth of a given BST, and to return an element given its index:

```
public static int depth (BST.Node top) {
// Return the depth of the BST with root node top.
   if (top == null)
        return -1;
   else
      return 1 + Math.max(depth(top.left),
           depth(top.right));
}
public static Object get (BST.Node top, int k) {
// Return the element in the k'th node from the left of the BST with
// root node top. Throw an exception if there is no such element.
   if (top == null)
      throw …;
   else {
      int leftSize = subtreeSize(top.left);
      if (k == leftSize)
         return top.element;
      else if (k < leftSize)
         return get(top.left, k);
      else // k > leftSize
         return get(top.right, k - leftSize - 1);
   }
}
```

10F For example, a pre-order traversal of BST (b) on slide 10-11 produces the following output:

cat, pig, fox, dog, lion, tiger, rat

Inserting these words in this sequence, one by one into an initially empty BST, does reproduce the original BST.

10G An in-order traversal of a BST visits the elements in ascending order. For example, an in-order traversal of BST (b) on slide 10-11 produces the following output:

cat, dog, fox, lion, pig, rat, tiger

Inserting these words in this sequence, one by one into an initially empty BST, would result in an extremely unbalanced BST.

10H To test whether a given BST is well-balanced, add the following method (which uses the `depth` method from Exercise 10E) to the `BST` class:

```
public boolean isBalanced () {
// Return true if and only if this BST is balanced.
   if (root == null)
      return true;
   else {
      int depth = BST.Node.depth(root);
      return BST.Node.isBalanced(root, depth, 0);
   }
}
```

Also add the following method to the `BST.Node` class:

```
public static boolean isBalanced(BST.Node top,
                int treeDepth, int nodeDepth) {
// Return true if and only if the subtree whose topmost node is top is
// balanced, where treeDepth is the depth of the entire BST and
// nodeDepth is the depth of the node top.
   if (top == null)
      return (nodeDepth <= treeDepth);
   else
      return isBalanced(top.left, treeDepth,
           nodeDepth+1) && isBalanced(top.right,
           treeDepth, nodeDepth+1);
}
```

10J To balance a given BST, add the following method to the `BST` class. This method first uses the `isBalanced` method from Exercise 10H to test whether the BST is ill-balanced. If so, it copies all its elements to an auxiliary array, makes the BST empty, and then reinserts the elements in a suitable order to produce a balanced BST.

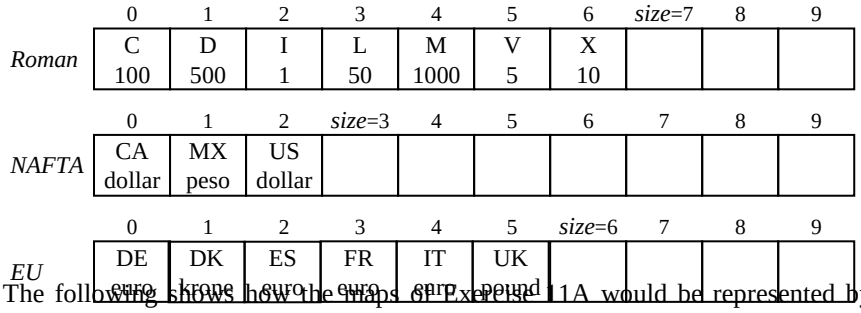
```
public void balance () {
// Balance this BST if it is ill-balanced.
   if (! isBalanced()) {
      int size = BST.Node.size(root);
      Comparable elems = new Comparable[size];
      BST.Node.fillArray(root, elems, 0);
      root = null;
      insertAll(elems, 0, size-1);
   }
}
private void insertAll (Comparable[] elems,
                        int left, int right) {
// Insert the elements of the sorted subarray elems[left...right]
// into this BST in such a way that this BST remains balanced.
   if (left <= right) {
      int mid = (left + right)/2;
      insert(elems[mid]);
      insertAll(elems, left, mid-1);
      insertAll(elems, mid+1, right);
   }
}
```

Also add the following method to the `BST.Node` class:

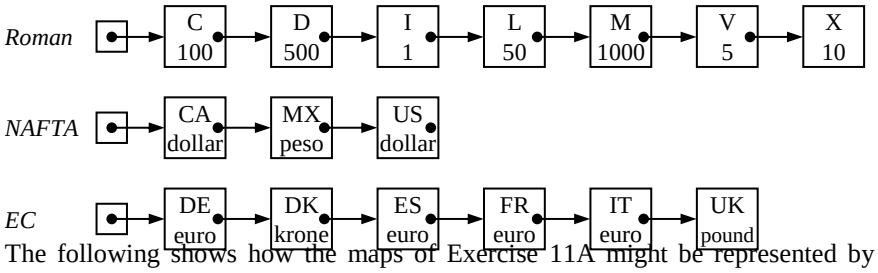
```
public static int fillArray (BST.Node top,
                             Comparable[] elems, int i) {
// Fill the subarray elems[i...] with the sorted elements of the subtree
// whose topmost node is top. Return the index of the first unused
// component of the subarray.
   if (top == null)
      return i;
   else {
      int j = fillArray(top.left, elems, i);
      elems[j] = top.element;
      return fillArray(top.right, elems, j+1);
   }
}
```

Solutions to Exercises 11

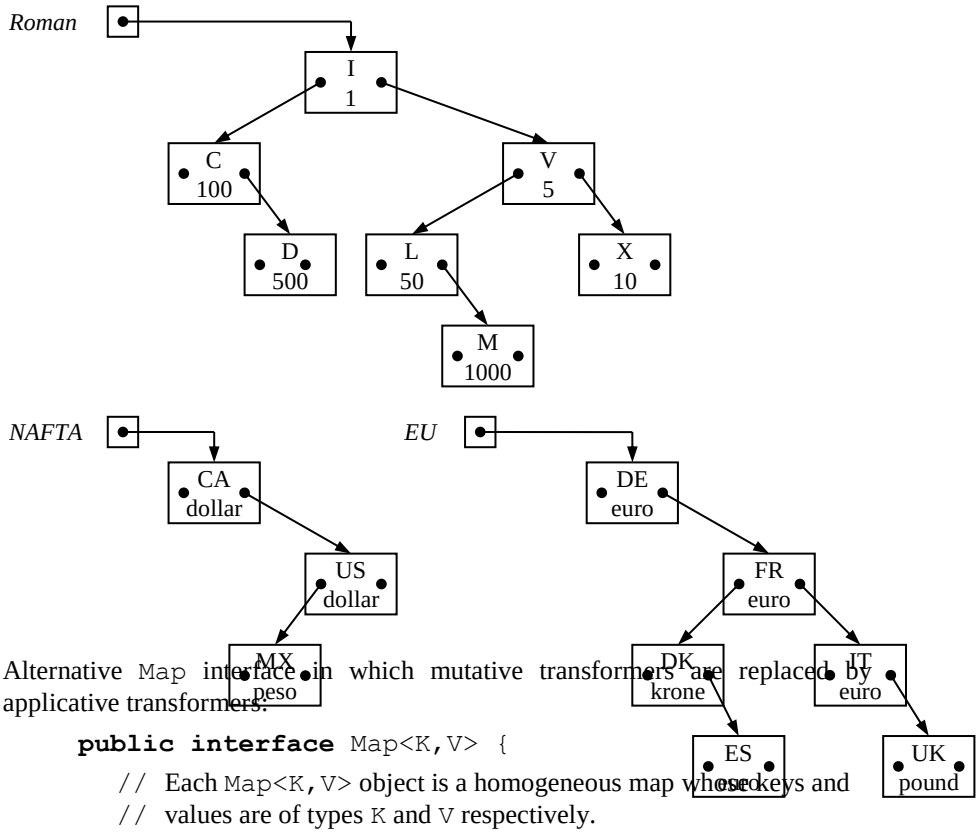
11A The following shows how the maps would be represented by entry arrays with *cap* = 10:



11B The following shows how the maps of Exercise 11A would be represented by SLLs:



11C The following shows how the maps of Exercise 11A might be represented by BSTs (but note that the exact shape of each BST depends on the order in which the entries were inserted):



11D Alternative `Map` interface in which mutative transformers are replaced by applicative transformers:

```
public interface Map<K,V> {
    // Each Map<K,V> object is a homogeneous map whose keys and
    // values are of types K and V respectively.
    /////////////  Accessors  /////////////
    …
    /////////////  Transformers  /////////////
```

```
    …
    public Map<K,V> remove (K key);
    // Return the new map obtained by removing the entry with key
    // (if any) from this map.
    public Map<K,V> overlay1 (K key, V val);
    // Return the new map obtained by overlaying this map with the
    // single entry ‹key, val›.
    public Map<K,V> overlay (Map<K,V> that);
    // Return the new map obtained by overlaying this map with that.
}
```

Implement the `remove` operation in the `ArrayMap<K,V>` class as follows:

```
public Map<K,V> remove (K key) {
  ArrayMap<K,V> that = this.clone();
  int pos = that.search(key);
  if (key.equals(that.entries[pos].key)) {
    for (int i = pos+1; i < that.dist; i++)
      that.entries[i-1] = that.entries[i];
    that.entries[--that.dist] = 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.

11E 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 the table below. (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.

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()`

11F Enhanced `Map` interface:

```
public interface Map<K,V> {
   //////////// Accessors ////////////
   …
   public boolean containsKey (K key);
   // Return true if and only if this map contains an entry whose key
   // is key.
   public boolean containsValue (V val);
   // Return true if and only if this map contains an entry whose value
   // is val.
   //////////// Transformers ////////////
   …
}
```

In the entry-array implementation, `containsKey` can be implemented by binary search, while `containsValue` must be implemented by linear search.

In the SLL implementation, both `containsKey` and `containsValue` must be implemented by linear search (but the `containsKey` linear search can exploit the fact that the entries are sorted by key).

In the BST implementation, `containsKey` can be implemented by BST search, while `containsValue` must be implemented by BST traversal terminating on a match. (Note that any traversal order will do.)

11G Enhanced `Map` interface:

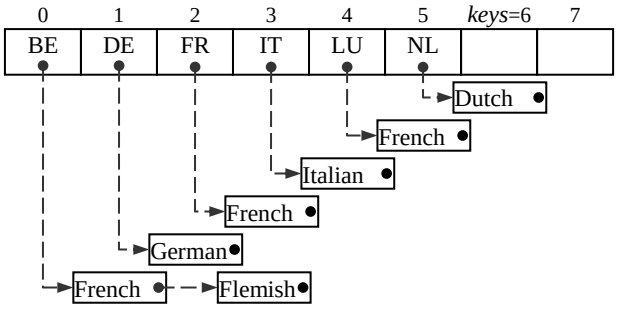
```
public interface Map<K,V> {
   //////////// Accessors ////////////
   …
   //////////// Transformers ////////////
   …
   public Map<K,V> subMap (K key1, K key2);
   // Return a map containing just those entries of this map whose keys
   // are not less than key1 and not greater than key2.
}
```

In the entry-array implementation, `subMap` can be implemented by binary searches for `key1` and `key2`, followed by copying all the entries between.

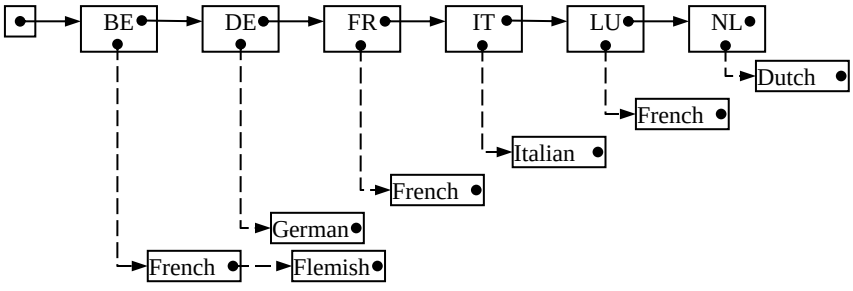
In the SLL implementation, `subMap` can be implemented by linear search for `key1`, followed by linear search for `key2`, meanwhile copying all the entries between.

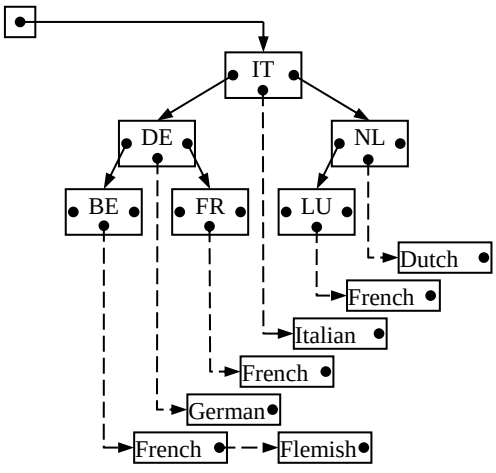
In the BST implementation, `subMap` can be implemented by a modified BST pre-order traversal that visits only subtrees containing entries between `key1` and `key2`, inserting all such entries into a new BST.

11H We can represent a multimap in the same manner as an ordinary map, except that each key is associated with a *set* of values. The following diagrams show sorted array, sorted SLL, and BST representations of an illustrative 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





Solutions to Exercises 12

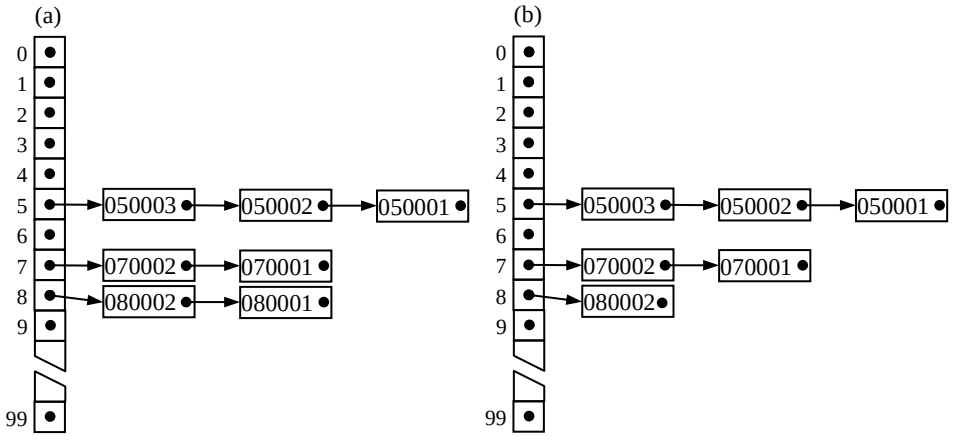
12A Diagram (a) below shows the effect of successively adding student-numbers to a CBHT.

The number of comparisons when the CBHT is searched for each key is:

080001	070001	070002	050001	080002	050002	050003
2	2	1	3	1	2	1

The average number of comparisons is (2+2+1+3+1+2+1)/7 ≈ 1.7.

Diagram (b) shows the effect of deleting 080001 from the CBHT.



12B If the keys are flight codes, and up to 200 entries are expected, we could choose a CBHT with:

m = 269
hash(k) = (weighted sum of characters of k) modulo 269

The chosen number of buckets is prime. The load factor is up to 200/269 ≈ 0.74. The chosen hash function will distribute the keys quite uniformly among the buckets.

(*Note:* A hash function that used only the serial number would be inferior, since there are likely to be patterns in the serial numbers used. A hash function that used only the airline code would be even worse, since the hash table might contain entries for few airlines, or even only one airline!)

12C If the keys are web server names, a hash function that used only the first six characters would be bad because there is a strong pattern in server names: nearly all have the prefix "www.".

A more suitable hash function would be:

hash(k) = (weighted sum of characters of k, ignoring the www prefix) modulo m

12D It makes sense to seek a perfect hash function only when the set of keys is fixed and known by the programmer. (The idea of a perfect function was invented by a compiler writer, who wanted a hash table to contain the set of keywords of the C programming language.)

A perfect hash function guarantees no collisions, so we can simplify the hash table algorithms (and code) by eliminating collision resolution.

A perfect hash function for {CA, MX, US} is:

m = 3
hash(k) = (first letter of k – 'A') / 9

A perfect hash function for {AT, BE, DE, DK, ES, FI, FR, GR, IE, IT, LU, NL,

PT, SE, UK} is:

m = 19

$hash(k)$ = (3 × (first letter of k – ‘A’)
– (second letter of k – ‘A’) / 10) modulo 19