Exercise 13 Solutions

1. a) The insertion might fail, if there are already 100 elements in the array. Since there is an explicit requirement that `x` not be an element of the array, the only way this can happen is if the array has elements that are equal to each other. The following invariant fixes the problem:

```
∀i,j:int. 0≤i<size ∧ 0≤j<size ∧ i≠j  ⇒  c[i]≠c[j]
```

b) After the first statement is executed, we have:

```
s.size = 0
```

After the second statement is executed, we have:

```
s.size = 1 ∧ s.c[0]=5
```

The assertion to prove, using the postcondition of `contains`, becomes:

```
s.c[0]=5 ∧ s.c[0]≠7
```

which is true.

c) This is impossible to do modularly. The proof in (b) depends on the internal representation of `SetN`, which we must now change.

d) The specification must not refer to the implementation of the class. A way to do that is to use pure methods, such as `contains`. Here is how we can do it here:

```
class SetN
{
        ...

        public SetN()
        // ensures ∀x:int. 0≤x<100 ⇒ ¬contains(x)
        { ... }

        public void insert(int x)
        // requires 0≤x<100
        // requires ¬contains(x)
        // ensures ∀y:int. 0≤y<100 ∧ x≠y ⇒
        //           contains(y)=old(contains(y))
        // ensures contains(x)
        { ... }

        public bool contains(int x)  // pure
        // requires 0≤x<100
        { ... }
}
```

e) After the first statement is executed, we have:

```
∀x:int. 0≤x<100 ⇒ ¬contains(x)
```

After the second statement is executed, we have:

```
∀y:int. 0≤y<100 ∧ 5≠y ⇒ ¬contains(y)
```

and

```
contains(5)
```

which proves the assertion immediately.

f) The new implementation keeps the same specifications, which means that the proof of (e) is preserved:

```
class SetN
{
      public boolean[] b;
      // invariant b.length=100

      public SetN()
      // ensures ∀x:int. 0≤x<100 ⇒ ¬contains(x)
      {
            b=new boolean[100];
            for(int i=0; i++; i<100) b[i]=false;
      }

      public void insert(int x)
      // requires 0≤x<100
      // requires ¬contains(x)
      // ensures ∀y:int. 0≤y<100 ∧ x≠y ⇒
      //           contains(y)=old(contains(y))
      // ensures contains(x)
      { b[x]=true; }

      public bool contains(int x)  // pure
      // requires 0≤x<100
      { return b[x]; }
}
```

2. a) The invariant can be written as follows:

```
∀i.0≤i<theTree.length/2⇒
      theTree[i] = theTree[2*i+1]+theTree[2*i+2]
```

Note that the condition `0≤i<theTree.length/2` says that node `i` is not a leaf (proof by induction on the height). Note also that "height" means the maximum distance of the root to the leaves (so a single node is a 0-height tree)

Of course, there should also be an invariant saying that the tree is complete:

$\exists h{:}int.\ h \geq 0 \wedge theTree.length = 2^{h+1}-1$

b)

- The method clearly does not preserve the invariant. For example, imagine a three-node tree [10,5,5] and a call to `addToNode(0,100)`

- When `addToLeaf` is called on a leaf, a sequence of recursive calls to `addToNode` begins. The first call adds a number `s` to the leaf, which

temporarily breaks the invariant, because the parent of that leaf no longer holds the correct sum. Each subsequent call of `addToNode` corrects the sum of its current node, similarly making the sum of its parent (if there is one) outdated. The calls to `addToNode` happen recursively all the way up from the leaf to the root, at which point the invariant is fixed.

So, either the method `addToNode` is called on a leaf or the invariant must be broken exactly at the node we call `addToNode`. Furthermore, the sum of the children of that node must be exactly `s` less than what it is supposed to be.

- Precondition for `addToNode` that expresses this requirement:

```
    theTree.length/2≤i<theTree.length
∨   (( ∀j.0≤j<theTree.length/2 ∧ j≠i ⇒
         theTree[j]=theTree[2*j+1]+theTree[2*j+2])
    ∧ theTree[i]=theTree[2*i+1]+theTree[2*i+2]-s)
```

- The method `addToNode` is private and therefore can be called only from two places: The first place is `addToLeaf`, which, by its precondition, satisfies the first disjunct of the precondition of `addToNode`.

  The second place is recursively from `addToNode` itself, if `i>0`. Assuming that the precondition a call of `addToNode` holds, we need to show that the precondition also holds when we make a recursive call to `addToNode`.

  Let `o` be the value of the old tree and `t` be the value of the new tree. Let `L` be the length of both trees. Then assumption becomes:

```
    L/2 ≤ i <L
∨   (     (∀j.0≤j<L/2 ∧ j≠i ⇒o[j]=o[2*j+1]+o[2*j+2])
    ∧     o[i]=o[2*i+1]+o[2*i+2]-s )
```

  and the two trees are connected by the relation

```
(∀j. 0≤j<L ∧ j≠i  ⇒ o[j]=t[j]) ∧ t[i]=o[i]+s
```

  We need to show (for `i>0`) that:

```
    L/2 ≤ i/2 < L
∨   ( (∀j.0≤j<L/2∧j≠(i-1)/2⇒t[j]=t[2*j+1]+t[2*j+2])
    ∧  t[(i-1)/2]=t[2*((i-1)/2)+1]+t[2*((i-1)/2)+2]-s)
```

  We can get the first disjunct out of the way, since it is false anyway. It suffices to prove that

```
(( ∀j.0≤j<L/2∧j≠(i-1)/2⇒t[j] = t[2*j+1]+t[2*j+2])
∧   t[(i-1)/2] = t[2*((i-1)/2)+1]+t[2*((i-1)/2)+2]-s)
```

Consider the last conjunct. Exactly one of the two indices `2*((i-1)/2)+1` and `2*((i-1)/2)+2` is equal to `i`. By the relationship between `o,t` the last conjunct becomes:

```
o[(i-1)/2] = o[2*((i-1)/2)+1]+o[2*((i-1)/2)+2]+s-s
```

which becomes

```
o[(i-1)/2] = o[2*((i-1)/2)+1]+o[2*((i-1)/2)+2]
```

From the universal quantification, we break the case `j=i`. The whole formula becomes:

```
      (∀j.0≤j<L/2 ∧ j≠(i-1)/2 ∧ j≠i ⇒
                  t[j]=t[2*j+1]+t[2*j+2] )
∧     t[i]=t[2*i+1]+t[2*i+2]
∧     o[(i-1)/2] = o[2*((i-1)/2)+1]+o[2*((i-1)/2)+2]
```

By the relationship between the trees (note that none of `2*j+1` and `2*j+2` can be equal to `i`, if `j≠i/2`):

```
      ( ∀j.0≤j<L/2 ∧ j≠(i-1)/2 ∧ j≠i ⇒
                  o[j]=o[2*j+1]+o[2*j+2])
∧ o[i]+s=o[2*i+1]+o[2*i+2]
∧ o[(i-1)/2] = o[2*((i-1)/2)+1]+o[2*((i-1)/2)+2]
```

Finally, we combine the first and the third conjunct, and we get exactly the precondition (of the original call) that we assumed holds.

- To show now that, given the precondition of `addToNode` holds in the beginning, then the invariant holds in the end, notice that the method does not make further calls to itself if and only if `i=0`. In that case, given that the precondition holds in the beginning of the call:

```
      theTree.length/2≤0<theTree.length
∨     (( ∀j.0<j<theTree.length/2 ⇒
            theTree[j] = theTree[2*j+1]+theTree[2*j+2])
      ∧ theTree[0] = theTree[1]+theTree[2]-s)
```

After the call, `theTree[0]` is incremented by `s`, and no other change happens. So we have:

```
      (∀j.0<j<theTree.length/2 ⇒
            theTree[j]=theTree[2*j+1]+theTree[2*j+2])
∧     theTree[0] = theTree[1]+theTree[2]
```

which is equivalent to the invariant.

3. a) We need the following specifications:

```
class SumVectors
{
     public Vector[] a=new Vector[0];

     // invariant forall v:a. v≠null

     public void insert(Vector vct)
     // requires vct≠null
     // ensures a.length=old(a).length+1
     // ensures ∀i:int. 0≤i<old(a).length ⇒
     //             a[i]=old(a)[i]
     // ensures a[old(a).length]=vct
     { ... }

     public Vector sum()
     // ensures result = Σ_{i=0}^{a.length-1} a[i]
     // (uses vector addition)
     { ... }
}
```

b) We only need one invariant:

$$sx=\Sigma_{i=0}^{a.length-1}\ a[i].x \ \wedge\ sy=\Sigma_{i=0}^{a.length-1}\ a[i].y$$