# Exercise 13 Solutions

1. a) We need the following specifications:

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

     // invariant a≠null ∧ ∀ 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.x = Σ_{i=0}^{a.length-1} a[i].x
     // ensures result.y = Σ_{i=0}^{a.length-1} a[i].y
     { ... }
}
```

b) We only need one invariant:

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

2. a) The invariant of the class, apart from **U**, should also contain the following conjuncts:

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

This part of the invariant is assumed throughout the solution, and we will not refer to it again.

The invariant **U** can be written as follows:

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

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

b) 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call of `addToNode` corrects the sum of its current node, similarly making the sum of its parent (if there is one) outdated.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.

c) The precondition is as follows: either (i) the method `addToNode` is called on a leaf or (ii) the invariant must be broken exactly at the node on which we call `addToNode`. In the latter case, the sum of the children of that node must be exactly `s` less than what it is supposed to be.

d) We can dent the invariant in the following way: Introduce a boolean array `b`. For every non-leaf `i`, the flag `b[i]` is true if and only if the **U** has to hold at node `i`. More formally, the dented version of the invariant is:

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

where the field `b` is declared as follows:

```
bool[] b;
```

This denting allows us to break **U** at any node in the tree, which makes the precondition described in (c) easily expressible.

Remember that the invariant must also specify that `b` is not null, and that the size of `b` is equal to the number of non-leaf nodes in the tree.

e) Here is the code together with the new field:

```
final class CompleteBinaryTree
{
     bool[] b;
     private int[] theTree;

     // invariant theTree≠null ∧ b≠null
     // invariant
           ∃h:int. h≥0 ∧ theTree.length=2^(h+1)-1
                 ∧ b.length=2^h-1
     // invariant: as mentioned in (d)


     public CompleteBinaryTree(int h)
           // ensures ∀i.b[i]
     {
           theTree = new int[Math.pow(2,h+1)-1];
           for(int i=0; i<theTree.length; i++)
                 theTree[i]=0;
           b = new bool[Math.pow(2,h)-1];
           for(int i=0; i<b.length; i++) b[i]=true;
     }
```

```
        public void addToLeaf(int i, int s)
               // requires
                     theTree.length/2 ≤ i < theTree.length
               // requires ∀j.b[j]
               // ensures theTree[i]=old(theTree[i+1])+s
               // ensures ∀j.b[j]
        { addToNode (i, s); }

        private void addToNode(int i, int s)
               // requires 0 ≤ i < theTree.length
               // requires i<theTree.length/2⇒
                ¬b[i] ∧
                 theTree[i]=theTree[2*i+1]+theTree[2*i+2]-s
               // requires ∀j. i≠j⇒b[j]
               // ensures theTree[i]=old(theTree[i+1])+s
               // ensures ∀j.b[j]
        {
               theTree[i]+=s;
               if(i<b.length) b[i]=true;
               if (i>0)
               {
                     b[(i-1)/2]=false;
                     addToNode((i-1)/2, s);
               }
        }
}
```

f) The claim is as follows: all methods preserve the dented invariant, and the public methods preserve the condition ∀j.b[j], which guarantees the undented invariant **U**.