Exercise 13

Invariants

1. The following code implements sets whose elements are natural numbers between 0 and 99:

```
class SetN
{
        public int[] c;
        public int size;
        // invariant c≠null ∧ c.length=100
        // invariant 0≤size<100
        // invariant ∀i:int. 0≤i<size ⇒ 0≤c[i]<100

        public SetN()
        // ensures size=0
        { c=new int[100]; size=0; }

        public void insert(int x)
        // requires 0≤x<100
        // requires ∀i:int. 0≤i<size ⇒ c[i]≠x
        // ensures ∀i:int. 0≤i<old(size) ⇒ c[i]=old(c)[i]
        // ensures size=old(size)+1 ∧ c[old(size)]=x
        { c[size]=x; size++; }

        public bool contains(int x)  // pure
        // requires 0≤x<100
        // ensures result = ∃i:int. 0≤i<size ∧ c[i]=x
        {
                for(int i=0; i++; i<size)
                        if(c[i]==x) return true;
                return false;
        }
}
```

a) The method `insert` does not satisfy its specification. Why? Can we fix the problem by adding an invariant?

b) Here is a client:

```
SetN s = new SetN();
s.insert(5);
// assert s.contains(5) ∧ ¬ s.contains(7)
```

Prove (informally) the assertion at the end of the code.

c) Suppose that we want to change our implementation of `SetN` as follows: instead of a list of all the contents in the set, we maintain an array `b` of 100

booleans, such that `b[i]=true` if and only if `i` is in the represented set. Can we do that modularly (i.e., without affecting client reasoning)? In particular, can we do it in a way that will not disturb the proof of question (b)?

d) Change the specifications of the code in question (a), so that there is no modularity problem.

e) Redo the proof of question (b) using the new specifications.

f) Re-implement the class, according to question (c) and the specifications that you wrote to answer question (d).

If your solution to questions (d-f) is correct, then your proof for question (e) should be adaptable for the new implementation.

2. In this question, we relax the requirement that all methods of a class should preserve its invariant: a *private* method of a class may break its invariant.

A technique to represent a complete binary tree T using an array A, is:

- store the root in A[0]
- for any node N stored in A[i], store the children of N to A[2i+1] and A[2i+2].

The size of the array should be equal to 2^{h+1}-1, where *h* is the height of the tree.

Consider the following invariant on a complete binary tree of integers: *any non-leaf node stores the sum of the integers stored in its two children*.

The following class uses the above-mentioned representation.

```
class CompleteBinaryTree
{
	private int[] theTree;

	public CompleteBinaryTree(int h)
	{
		theTree = new int[Math.pow(2,h+1)-1];
		for(int i=0; i<theTree.length; i++)
			theTree[i]=0;
	}

	// requires 0 ≤ i < theTree.length
	public int getNode(int i) { return theTree[i]; }

	// requires theTree.length/2 ≤ i < theTree.length
	// this means i must be a leaf
	public void addToLeaf (int i, int s)
	{ addToNode (i, s); }

	private void addToNode (int i, int s)
	{
		theTree[i]+=s;
		if (i>0) addToNode((i-1)/2, s);
	}
}
```

a) Write formally the invariant described above

b) Show that the invariant is always preserved by the public methods of the class. Hints:

- does the method `addToNode` preserve the invariant?
- what expectations do you have of the state of the object when `addToNode` is called?
- write a precondition for `addToNode` expressing these expectations
- prove that the precondition is always satisfied when the method is called
- is it true that the invariant holds after every call to `addToNode` (assuming your precondition was satisfied when the call was made)?

3. Consider the following Java classes:

```
class Vector
{
     public int x, y;
     Vector(int x, int y)
     { this.x=x; this.y=y; }
}

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

     public void insert(Vector vct)
     {
          Vector[] o=a;
          a=new Vector[a.length+1];
          for(int i=0; i<o.length; i++) a[i]=o[i];
          a[a.length]=vct;
     }

     public Vector sum()
     {
          int x=0, y=0;
          if(a!=null)
               for(Vector v : a)
               { x+=v.x; y+=v.y; }
          return new Vector(x, y);
     }
}
```

- Annotate the classes with specifications that ensure that there is no null-pointer dereferencing, that method `insert` inserts a new Vector object in the end of the array `a`, and that method `sum` computes the sum of all vectors in the array `a`.
- Annotate the following class, such that it is a behavioral subtype of `SumVectors`:

```
class FastSumVectors extends SumVectors
{
     int sx=0, sy=0;

     public void insert(Vector vct)
     {
          super.insert(vct);
          sx+=vct.x; sy+=vct.y;
     }

     public Vector sum()
     { return new Vector(sx, sy); }
}
```