

# Exercise 13

## Invariants

1. 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-1]=vct;
    }

    public Vector sum()
    {
        int x=0, y=0;
        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 with invariants, such that it is a behavioural 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); }
}
```

2. 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*. Let us call this invariant **U** (for “undented”; cf. “dented invariants” on Lecture 9, Slide 11).

The following class uses the above-mentioned representation.

```
final 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);
    }
}
```

- Write formally the invariant **U**.
- The method `addToNode` does not preserve **U**. Instead, its purpose is to *fix* **U**, when it is temporarily broken. Describe how this is done.
- Describe informally the precondition under which the method `addToNode` has to be called, such that **U** holds when the method terminates.
- Dent **U** accordingly so that the precondition above is formally expressible. **Hint:** Dented usually uses a single boolean field (see Lecture 9, Slide 11). Here, you need more than one boolean field.

## Concepts of Object-Oriented Programming

- e) Add assignments to the new boolean fields in the bodies of all the methods and write specifications for all the methods. All methods must preserve the dented invariant.
- f) Explain why the public interface of the class preserves **U**.