Übungsblatt 13

1. Gegeben sei folgende Klasse:

```
class List {
   int i;
   List next;

   // invariant   next != null  =>  i == next.i

   // i in der ganzen Liste erhöhen
   void inc() {
        …
   }
}
```

a. Schreiben Sie eine formale Spezifikation für die Methode `inc()`

b. Schreiben Sie alle Beweisverpflichtungen für `inc()` auf

c. Schreiben Sie eine rekursive Implementierung der Methode `inc()`

d. Überprüfen Sie, ob die Implementierung die Beweisverpflichtungen erfüllt

e. Falls nicht, geben Sie eine alternative Lösung an, die die Beweisverpflichtungen erfüllt

2. Gegeben seien folgende Klassen:

```
public class DLHead {

  private class DLNode {
    DLNode prev;

    DLNode next;

    Object elem;
  }

  private DLNode first;

  private DLNode last;

  public void add( Object obj ) { /* ... */ }

  public Object getFirst() { return first.elem; }

  public void removeFirst() { /* ... */ }

  ...
}
```

1. Diese Klassen sollen eine übliche doppelt-verkettete Liste beschreiben. Schreiben Sie die notwendigen Invarianten für die Klassen `DLHead` und `DLNode`.

2. Es soll zusätzlich sichergestellt werden, dass die Knoten aufsteigend nach dem Hashwert der Elemente sortiert werden.
   a. Welche zusätzlichen Invarianten sind dafür notwendig?
   b. Wie kann diese Eigenschaft durch die Verwendung der zusätzlichen Model-Fields `min` und `max` sichergestellt werde?
      Das Model-Field `min` enthält den minimalen Hashwert der Liste, das Feld `max` enthält den maximalen Hashwert der Liste.

3. **Spezifikationsfrage von der letztjährigen Prüfung.**
   Diese Aufgabe setzt Nummer 4 von Übungsblatt 7 fort.

   Gegeben ist der Quellcode für einen sortierten Baum:

```
public class SortedTree {
  // the root of the sorted tree
  private Node root;

  // insert a new element at the correct location in the tree
  public void insert(int ke, Object val) {
       if (root == null) { root = new Node(ke, val); }
       else { root.insert(ke, val); }
  }

  // merge the given tree with the current tree
  public void merge(SortedTree toinsert) {
       if (root == null) { root = toinsert.root; }
       else { root.merge(toinsert.root); }
  }
}
```

   Mit der Hilfsklasse Node:

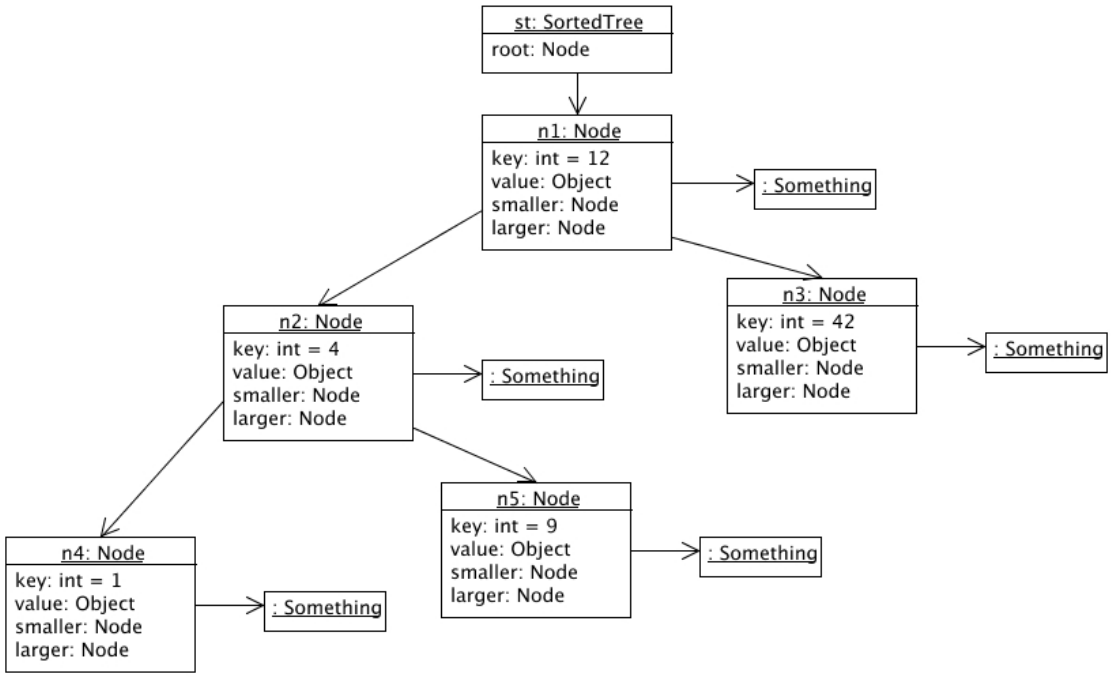
```
// this class is a helper class and uses default
// visibility everywhere
class Node {
  // subtree with smaller elements
  Node smaller;
  // subtree with larger elements
  Node larger;
  // the key of this Node
  int key;
  // the value of this Node
  Object value;

  Node(int ke, Object val) {
       this.key = ke; this.value = val;
  }

  // insert the new key-value pair at the correct position
  void insert(int ke, Object val) {
       if (ke <= this.key) {
            if (smaller == null) {
                 smaller = new Node(ke, val); }
            else { smaller.insert(ke, val); }
       } else {
            if (larger == null) {
                 larger = new Node(ke, val); }
            else { larger.insert(ke, val); }
       }
  }
```

```
        // insert the given subtree at the correct position
        void merge(Node toinsert) {
            if (toinsert.key <= this.key) {
                if (smaller == null) { smaller = toinsert; }
                else { smaller.merge(toinsert); }
            } else {
                if (larger == null) { larger = toinsert; }
                else { larger.merge(toinsert); }
            }
        }
}
```

Eine gültige Objekt-Struktur sieht wie folgt aus:



Der Programmierer möchte sicherstellen, dass seine Implementierung korrekt ist und möchte daher Spezifikationen für sein Programm schreiben. Es sollen die folgenden Eigenschaften sichergestellt werden:

Eigenschaft 1: Der Graph der Node-Objekte mit ihren `smaller` und `larger`-Referenzen ist ein Baum. D.h., es gibt keine Zyklen zwischen den Knoten des Graphen und kein Aliasing von Knoten.

Eigenschaft 2: Der Baum ist korrekt nach dem Schlüssel `key` sortiert.
D.h., alle Schlüssel im `smaller` Unterbaum eines Knotens sind kleiner oder gleich dem Schlüssel des Knotens selbst. Dieser wiederum ist kleiner oder gleich allen Schlüssel im `larger` Unterbaum.

Es sind folgende Teilaufgaben zu lösen:

a) Es gibt ein irreflexives, transitives Prädikat `reachable(Node n, Node m)`, welches ausdrückt, dass man vom Knoten `n` aus den Knoten `m` erreichen kann, indem man den `smaller` und `larger` Referenzen folgt.

   Wie kann man unter Verwendung dieses Prädikats die obigen beiden Eigenschaften als Contract spezifizieren?

b) Welche generelle Regel gilt für Methodenaufrufe in Spezifikationen?

c) Das Prädikat `reachable` wird nicht vom System angeboten und muss als Methode implementiert werden.
   Kann man eine entsprechende Methode implementieren, die man in Spezifikationen benutzen kann?
   Berücksichtigen Sie bei der Begründung Ihrer Antwort auch die Antwort auf Teilfrage b.
   Sie brauchen die Methode nicht zu implementieren.

d) In die Klasse Node werden zwei Model-Felder eingefügt:

```
class Node {
  Node smaller;
  Node larger;

  model int min;
  represents min <- (smaller!=null) ? smaller.min : this.key;

  model int max;
  represents max <- (larger!=null) ? larger.max : this.key;

  int key;
  Object value;
  ...
```

   Schreiben Sie mit Hilfe dieser Model-Felder einen Contract für Eigenschaft 2 ohne das Prädikat `reachable` zu verwenden.

e) Die gegebene Implementierung der Klassen `Node` und `SortedTree` erfüllt die zweite Eigenschaft nicht. Geben Sie ein Beispiel, an dem man das Fehlverhalten sieht.

f) Korrigieren Sie die fehlerhaften Methoden.

g) Wird die Spezifikation der Datenstruktur durch die Verwendung des Universe Typsystems vereinfacht? Diskutieren Sie die beiden Eigenschaften für beide Ownership-Strukturen aus Nummer 4 von Übungsblatt 7. Welche Contracts sind jeweils noch notwendig?