

Algorithmen und Programmierung III

Probeklausur - Musterlösung

Aufgabe 1

konkrete Invariante

Alle `Cell`-Objekte der über `first` erreichbaren verketteten Liste haben paarweise verschiedene `index`-Attribute und sind in der verketteten Liste aufsteigend nach diesem Index sortiert. Alle `value`-Attribute der `Cell`-Objekte sind von 0 verschieden.

```
-- l enthalten die Tupel aus der über first erreichbaren verketteten Liste
inv l n = sort idxs == idxs &&          -- aufsteigend sortiert
         idxs == nub idxs &&          -- paarweise verschieden
         all (/=0) (map snd l) &&    -- Werte nicht Null
         where idxs = map fst l
```

Abstraktionsfunktion (im Gegensatz zur Aufgabe mit bekanntem n)

Der abstrakte Wert ist ein Vektor v mit n Komponenten mit den Indizes 0 bis $n-1$. Die Werte der Komponenten ergeben sich aus den Einträge der über `first` erreichbaren verketteten Listen von `Cell`-Elementen, wobei `index` den Index im Vektor v und `value` den Wert beschreibt. Alle Komponenten, für die es kein `Cell`-Objekt gibt, haben den Wert 0.

```
abst l n = [ lookup2 i l | i <- [0..n-1] ]
  where lookup2 i [] = 0
        lookup2 i ((k,v):l)
          | i < k = 0
          | i == k = v
          | otherwise = lookup2 i l
```

Voraussetzung firstIndexNeg

Es gibt in der über `first` erreichbaren verketteten Liste mindestens ein `Cell`-Objekt dessen `value`-Attribut kleiner als 0 ist.

```
pre [] = False
pre ((i,v):l) = v < 0 || pre l
```

Effekt firstIndexNeg

Zurückgeliefert wird der `index`-Wert des ersten `Cell`-Objektes der verketteten Liste, dessen `value`-Attribut kleiner als 0 ist.

```
post l = result == head [i | (i,v) <- l, v < 0] -- liste ist sortiert
```

Aufgabe 2

a)

```
public class HashedMapping<K,D> implements Mapping<K,D> {
```

```

// Repraesentation
private final int M;          // Groesse der Hashtabelle
private Bucket<K,D>[] map;   // die Hashtabelle
private class Bucket<K,D> {
    K key;
    D data;
    Bucket<K,D> next;
}
public HashedMapping(int size) {
    M = size;
    map = new Bucket[M]; // raw
}
public void iterate(Action<D> a) {
    for(Bucket<K,D> b : map) {
        while (b!=null) {
            if (a.cond(b.data))
                b.data = a.func(b.data);
            b = b.next;
        }
    }
}
}

```

b)

```

class Raise implements Action<Float> {
    private float factor;
    private float limit;
    public Raise(float factor, float limit) {
        this.factor = factor;
        this.limit = limit;
    }
    public boolean cond(Float x) {
        return x <= limit;
    }
    public Float func(Float x) {
        return Math.min(limit, x*factor);
    }
}

```

Aufgabe 3

```

public void insert(float x) {
    if (root==null) { root = new Node(true, x, null, null); return; }
    Node y = root;
    while (!y.leaf)
        if (x<y.value) y = y.left;
        else          y = y.right;
    if (x==y.value) return; // schon drin
    float yval = y.value;
    y.value = (x+yval)/2;
    Node left  = new Node(true, Math.min(x,yval), y.left, null);
    Node right = new Node(true, Math.max(x,yval), left, y.right);
    left.right = right;
}

```

```
    if (y.left != null) y.left.right = left; // benachbarte Blätter anpassen
    if (y.right != null) y.right.left = right;
    y.left = left; // als Kinder unten y einklinken
    y.right = right;
    y.leaf = false;
}
```