

## 1. Aufgabe

Der abstrakte Datentyp Prioritätsschlange kann in der Heapausführung so umgemodelt werden, dass es auch  $removeMax()$  und  $changeKey(e, k)$  effizient durchführen kann. Und zwar nimmt man sich zwei Heaps her, der eine ganz normal aufgebaut (Jetzt immer mit  $A$  bezeichnet) und der andere so, dass er die selben Elemente enthält, nur dass diesmal der Schlüssel des Elternknotens größergleich der Schlüssel in den Kindknoten ist (im folgenden mit  $B$  bezeichnet). Zudem hat jeder Knoten in  $A$  und  $B$  einen Verweis auf das Äquivalent im anderen Heap. Somit steht in  $B$  in der Wurzel also das größte Element (das mit der niedrigsten Priorität).

Jetzt zu den Algorithmen und Analysen

$removeMax()$  In  $B$  kann das Maximum ganz einfach entfernt werden. Zugleich muss es in  $A$  in einem der Blätter stehen, da es in keinem inneren Knoten stehen darf, sonst würde die Heapstruktur verletzt werden.

```
maxB = Wurzel von B
maxA = Verweis von max auf Äquivalent in A
letzten Knoten von A mit maxA tauschen
maxA löschen
```

```
lösche maxB
in B das letzte Element lastB (ganz unten rechts)
  in die Wurzel schreiben
entferne Blatt von lastB
while (lastB < ein Kind von lastB)
  vertausche lastB und größtes Kind
```

Die Wurzel von  $B$  wird in Konstanter Zeit gefunden, demnach  $maxA$  auch, weil nur dem Link gefolgt werden muss. Der Tausch von dem letzten Knoten in  $A$  mit  $maxA$  geht natürlich auch konstanter Zeit, also für den ersten Teil haben wir eine Laufzeit von  $O(1)$ . Das Löschen von  $maxB$  und das in die Wurzel schreiben vom letzten Element geht in konstanter Zeit. Für das Durchsickern kann es im schlechtesten Fall passieren, dass wir von der Wurzel bis zum Blatt tauschen müssen, was der Höhe von Vertauschungen entspricht. Da wir einen Binären Baum haben, ist die Höhe durch  $\log_n$  gegeben. Demnach ist hier  $O(\log_2 n)$  die Laufzeit.

Insgesamt ergibt sich als Laufzeit  $O(1) + O(\log_2 n) = O(\log_2 n)$ .

$changeKey(e, k)$

```
eA = e
eB = Verweis von eA auf Äquivalent in B
eA.key = k
aB.key = k
```

```
if(keyAlt < k) {
    while(eA > eA Kind)
        vertausche eA mit dem kleineren der beiden Kinder
    while(eB < eB Vater)
        vertausche eB mit dem Vater
}
else if(keyAlt > k) {
    while(eA < eA Vater)
        vertausche eA mit dem Vater
    while(eB > eB Kind)
        vertausche eB mit dem größeren der beiden Kinder
}
```

Das Umändern des Schlüssels geht natürlich in konstanter Zeit. Desweiteren ist das durchsickern wie vorher. Auch hier sieht man, dass man bei dem Vertauschen nach oben oder unten maximal die Höhe tauschen kann. Das gilt natürlich für den  $A$  und  $B$  Heap, also  $O(2 * \log_2 n) = O(\log_2 n)$ .

## 2. Aufgabe

---

```
1 import java.util.List;
2
3 public class Entry<K, V> {
4     K key;
5     V value;
6
7     public Entry(){
8         key=null;
9         value=null;
10    }
11    public Entry(K key, V value){
12        this.key=key;
13        this.value=value;
14    }
15
16    public void setKey(K key){
17        this.key=key;
18    }
19    public K getKey(){
20        return key;
21    }
22
23    public void setValue(V value){
24        this.value=value;
```

```
25     }
26     public V getValue(){
27         return value;
28     }
29
30     public String toString() {
31         return "(" + key.toString() + "," + value.
32             toString() + ")";
33     }
34     =====
35     import java.util.Collection;
36
37     public interface Woerterbuch < K , V>{
38
39         public void insert(K key, V value);
40         public Entry<K,V> find (K key);
41         public Collection<Entry<K,V>> findAll(K key);
42         public void delete(Entry<K,V> key);
43         public Collection<Entry<K,V>> entries();
44         public boolean isEmpty();
45         public int size();
46     }
47     =====
48     import java.util.LinkedList;
49     import java.util.List;
50     import java.util.Vector;
51
52     class HashTabelle_S<K, V> implements Woerterbuch<K, V> {
53
54         int current = 14;
55
56         LinkedList<Entry<K, V>>[] hashtabelle;
57
58         public HashTabelle_S() {
59             hashtabelle = new LinkedList[current];
60             for (int i = 0; i < current; i++)
61                 hashtabelle[i] = new LinkedList<Entry<K
62                     , V>>();
63
64         }
65
66         public HashTabelle_S(int tabSize) {
67             hashtabelle = new LinkedList[tabSize];
68             current = tabSize;
69             for (int i = 0; i < tabSize; i++)
```

```
68         hashtablete[i] = new LinkedList<Entry<K
           , V>>();
69     }
70
71     public int hash(K key) {
72         return Math.abs(key.hashCode() % hashtablete.
           length);
73     }
74
75     public Entry<K,V> find(K key) {
76         int hash = hash(key);
77
78         for(int i = 0;i< hashtablete[hash].size();i++)
           {
79             if(hashtablete[hash].get(i).key == key)
80                 return hashtablete[hash].get(i)
           ;
81         }
82         return null;
83     }
84
85     public List<Entry<K,V>> findAll(K key) {
86
87         List<Entry<K,V>> liste = new Vector<Entry<K,V
           >>();
88         int hash = hash(key);
89         for (int i = 0; i < hashtablete[hash].size(); i
           ++){
90             if(hashtablete[hash].get(i).key == key)
91                 liste.add(hashtablete[hash].get
           (i));
92         }
93
94         return liste;
95     }
96
97     public int size() {
98
99         int d = 0;
100
101         for (int i = 0; i < current; i++)
102             d = d + hashtablete[i].size();
103
104         return d;
105     }
106
```

```
107     public boolean isEmpty() {
108
109         if (this.size() == 0)
110             return true;
111         else
112             return false;
113     }
114
115     public List<Entry<K, V>> entries() {
116
117         List<Entry<K, V>> liste = new Vector<Entry<K, V
118             >>();
119
120         for (int i = 0; i < current; i++)
121             liste.addAll(hashtabelle[i]);
122
123         return liste;
124     }
125
126     public void insert(K key, V value) {
127         Entry<K, V> eintrag = new Entry<K, V>(key,
128             value);
129         int h = hash(key);
130         hashtabelle[h].add(eintrag);
131     }
132
133     public void delete(Entry<K,V> e) {
134         int hash = hash(e.key);
135         for(int i=0;i<hashtabelle[hash].size();i++) {
136             if(hashtabelle[hash].get(i).key.equals(
137                 e.key) && hashtabelle[hash].get(i).
138                 value.equals(e.value))
139                 hashtabelle[hash].remove(i);
140         }
141     }
142
143     public static void main(String[] arguments) {
144
145         HashTabelle_S<String, Integer> a = new
146             HashTabelle_S<String, Integer>();
147
148         a.insert("hans", 2098364);
149         a.insert("martin", 4398745);
150         a.insert("martin", 4093847);
151         a.insert("halim", 9873452);
152         a.insert("zoran", 2753412);
```

```
148         a.insert("zoran", 3245963);
149         System.out.println(a.findAll("zoran"));
150         Entry<String, Integer> mar = new Entry("martin"
151             ,4093847);
152         a.delete(mar);
153         System.out.println(a.entries());
154         System.out.println("Hashtabelle leert?" + a.
155             isEmpty());
156         System.out.println("Anzahl aller Entries:" + a.
157             size());
158     }
159 }
```

---