Sarah Ertel
Patrick Greher
Eugen Ljavin

1	2	3	4	Σ
8	4	5	2	19

Übungsblatt Nr. 7

(Abgabetermin 14.06.2018)

Aufgabe 1

Algorithm 1: Plane Sweep algorithm for circle intersection detection*

```
1  function getIntersections(circles [ c_1, ..., c_n ])
2  xStructure;
3  yStructure;
4  intersections;
5  for circle in circles do
6  |  xStructure.insert(circle.x - circle.r) as startpoint;
7  |  xStructure.insert(circle.x + circle.r) as endpoint;
8  end
9  for circle in circles do
10 |  if isStartpoint(x) then
11 |  |  yStructure.insert(y);
12 |  |  c_0 ← getCircleByX(x);
13 |  |  c_1 ← getCircleByY(yStructure.getSuccessor(y));
14 |  |  c_2 ← getCircleByY(yStructure.getPredecessor(y));
15 |  |  if exists(c_1) then
16 |  |  |  s[ ] ← Schnitt(c_0, c_1);
17 |  |  |  if notEmpty(s) then
18 |  |  |  |  intersections.add(s);
19 |  |  |  end
20 |  |  end
21 |  |  if exists(c_1) then
22 |  |  |  s[ ] ← Schnitt(c_0, c_2);
23 |  |  |  if notEmpty(s) then
24 |  |  |  |  intersections.add(s);
25 |  |  |  end
26 |  |  end
27 |  else
28 |  |  yStructure.delete(y);
29 |  |  y_1, y_2 ← yStructure.getNewNeighbors();
30 |  |  if exists(y_1) ∧ exists(y_2) then
31 |  |  |  c_1 ← findByY(y_1);
32 |  |  |  c_2 ← findByY(y_2);
33 |  |  |  s[ ] ← Schnitt(c_1, c_2);
34 |  |  |  if notEmpty(s) then
35 |  |  |  |  intersections.add(s);
36 |  |  |  end
37 |  |  end
38 |  end
39 end
40 return intersections;
```

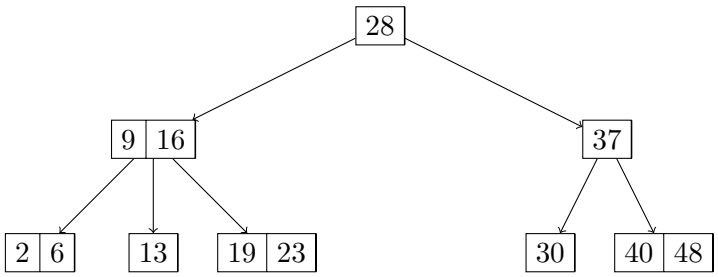
*Es wird angenommen, dass zu einem Start- und Endpunkt mittels $findByX(...)$ oder zu einem y-Wert mittels $findByY(...)$ der zugehörige Kreis z.B. durch eine Hashtable in konstanter Laufzeit ermittelt werden kann.

Zunächst wird für jeden der n Kreise sein Start- sowie Endpunkt ($x-r := \mathit{Startpunkt}$), ($x+r := \mathit{Endpunkt}$) berechnet und in einem AVL Baum gespeichert (X-Struktur). Dies beansprucht eine Laufzeit von $\mathcal{O}(n) * \mathcal{O}(\log n) = \mathcal{O}(n \log n)$.
Die X-Struktur wird in der Laufzeit $\mathcal{O}(n)$ in-order traversiert. Für jeden Startpunkt wird in der Laufzeit $\mathcal{O}(\log n)$ die y-Koordinate des Startpunktes in einen AVL-Baum (Y-Struktur) eingefügt. Bei n Startpunkten ergibt das eine Laufzeit von $\mathcal{O}(n) * \mathcal{O}(\log n) = \mathcal{O}(n \log n)$.
Beim Einfügen der y-Koordinate eines Startpunktes in die Y-Struktur wird geprüft, ob sich der zugehörige Kreis mit seinen unmittelbaren Nachbarkreisen (das nächstkleinere Element (successor) sowie das nächstgrößere Element (predecessor) in der Y-Struktur) schneiden. So wird der Schnitt (falls vorhanden) zwischen dem betrachteten Kreis und den direkten Nachbarn erkannt. Das Überprüfen, ob sich die Kreise schneiden ist in konstanter Zeit $\mathcal{O}(1)$ möglich (siehe Aufgabenbeschreibung). Das finden des nächstkleineren / nächstgrößeren Elements ist im AVL Baum in der Zeit $\mathcal{O}(\log n)$ möglich.
Beim traversieren der X-Struktur wird für jeden Endpunkt der zughörige y-Wert in der Y-Struktur gelöscht, was ebenfalls eine Laufzeit von $\mathcal{O}(n) * \mathcal{O}(\log n) = \mathcal{O}(n \log n)$ ergibt.
Nach dem Löschen des y-Werts wird des weiteren geprüft, ob sich die neu ergebenden unmittelbaren Nachbarkreise schneiden. Dies ist notwendig, da sich zwei Kreise auch schneiden können, wenn ein anderer Kreis zwischen ihnen liegt.
Insgesamt ergibt sich eine Laufzeit von $\mathcal{O}(n \log n)$.

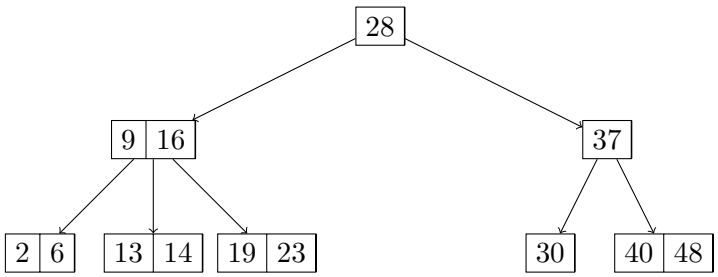
Aufgabe 2

a)

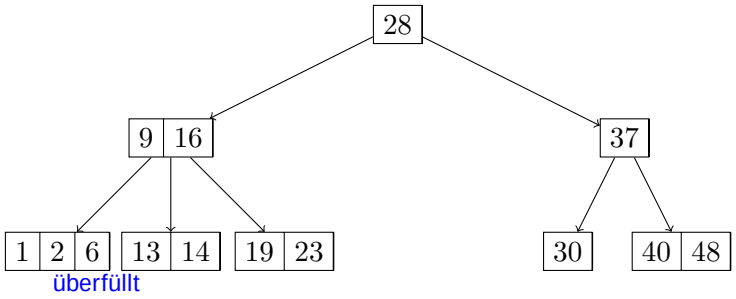
Initialer B-Baum der Ordnung 2:



Einfügen des Knotens mit dem Key 14:

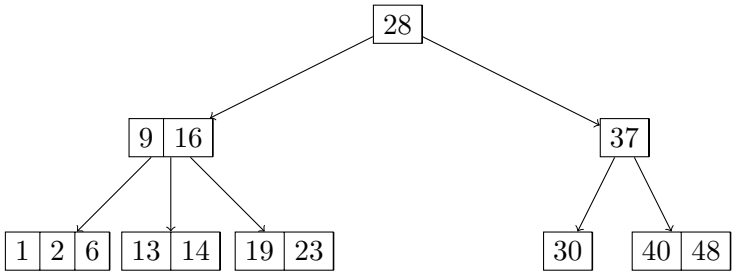


Einfügen des Knotens mit dem Key 1:

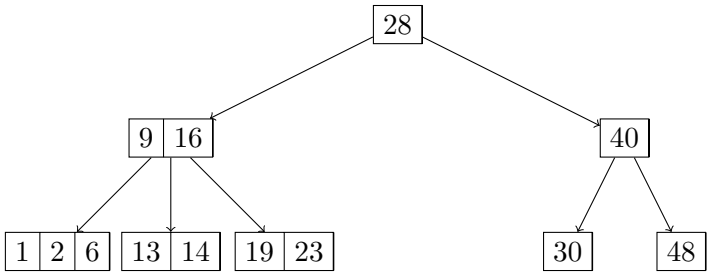


b)

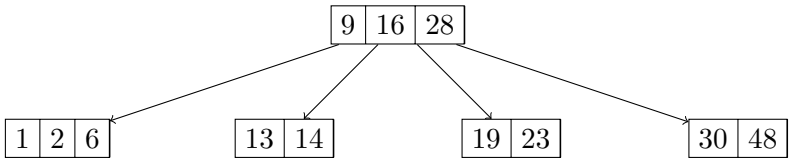
Initialer B-Baum der Ordnung 2:



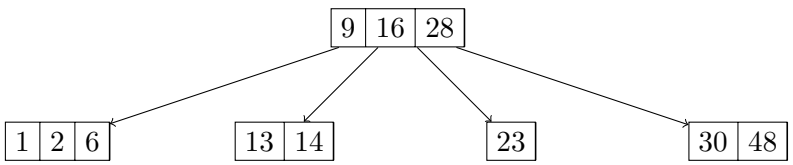
Löschen des Knotens mit dem Key 37. Rebalancierung erforderlich:



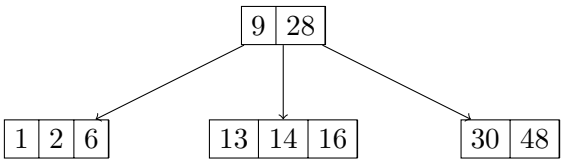
Löschen des Knotens mit dem Key 40. Rebalancierung erforderlich:



Löschen des Knotens mit dem Key 19:



Löschen des Knotens mit dem Key 23:



Folgefehler

Aufgabe 3

a)

Annahme:
hS = Höhe Baum S
hT = Höhe Baum T

Algorithm 2: merge(S,T)

```
1  if hS == hT then
2  |  root = min {T};
3  |  add S as left subtree and T as right subtree
4  end
5  if hS>hT then
6  |  root = max {S};
7  |  add T as right subtree of root;
8  |  repair new tree with insert(T);
9  end
10 if hS<hT then
11 |  root = min {T};
12 |  add S as left subtree of root;
13 |  repair new tree with insert(S);
14 end
```

Da alle Werte aus S kleiner als der kleinste Wert aus T sind, kann der gesamte Baum S an jede beliebige Wurzel aus T angehängt werden. Wählt man den kleinsten Wert aus T, kann man auch alle Werte aus T als rechten Subtree nehmen, da die restlichen Werte größer sind. Umgekehrt gilt das Selbe wenn der größte Wert aus S die Wurzel bildet.
Die Laufzeit beträgt $\mathcal{O}(\log(n)($ für die Wahl der Wurzel. Falls die Höhe beider Bäume gleich ist erfolgt das anfügen der Subtrees mit der Laufzeit $\mathcal{O}(1)$. FOlglich beträgt die Laufzeit $\mathcal{O}\log(n)$.
Ist die Höhe allerdings unterschiedlich ergibt sich die Laufzeit durch das insert von $\mathcal{O}\log(n)$. Dadurch bleibt die Laufzeit auch im worst-case bei $\mathcal{O}\log(n)$.

korrekt

b)

Algorithm 3: split(T,x)

```
1  S1 = new Tree(2,4);
2  S2 = new Tree(2,4);
3  for node in T do
4      for i = 0; i < node.keys; i++ do
5          if node.value>x then
6              insert(S1, node.value);
7              insert(S1, node.rightChilds);
8              delete(T,node.rightChilds);
9          end
10         if node.value<=x then
11             insert(S2, node.value);
12             insert(S2, node.leftChilds);
13             delete(T, node.leftChilds);
14         end
15         delete(T, node);
16     end
17 end
```

Da ein Node mit x verglichen wird gilt folgendes Prinzip:
Ist der Wert des Nodes kleiner oder gleich x, dann sind alle Elemente links des Nodes ebenfalls kleiner als x. Selbiges Prinzip gilt für die Werte größer x. Daher können diese Subtrees direkt in den neuen Subtree eingefügt werden. Es muss somit immer nur der mittlere Node weiter verfolgt werden. Durch das Löschen wird verhindert, dass diese Werte doppelt betrachtet werden
Da wie erwähnt der großteil der Nodes mit 2 Vergleichen gesplittet werden kann entsteht hierbei ein Aufwand von $\mathcal{O}(1)$. Für das Einfügen und Löschen der Nodes entsteht jeweils ein Aufwand von $\mathcal{O}(\log n)$. Zudem benötigt das Einfügen in den neuen Baum einen Aufwand von $\mathcal{O}(\log n)$. Die Gesamtlaufzeit beträgt somit $\mathcal{O}(3 \log n)$, also $\mathcal{O}(\log n)$.

so effizient, dass man fast meinen könnte, dass da ein Fehler ist...

Aufgabe 4

a)

- `enqueue`: Führe die Operation `push` auf Stack#1 aus
- `dequeue`: `push` alles von Stack#1 auf Stack#2 wenn Stack#2 leer ist. Führe die Operation `pop` auf Stack#2 aus

korrekt

b)

- `enqueue`:
- `dequeue`: