Theoretische Aufgaben

Aufgabe 1

Listing 1: Aufgabe 1

```
reverse(list l)
//falls Liste 0 oder 1 Element enthält
if list.size <= 1
  return l

// Schleife startet mit zweitem Element der Liste.
// 'next' ist der Zeiger auf das nächste Feld,
// die Funktion zeigeAuf() ermittelt die Adresse des Elements.
for i=1 to l.size
    l[i].next = zeigeAuf(l[i-1])

return l
```

Aufgabe 2

Listing 2: Aufgabe 2

```
void enqueue(element e)
//Wir brauchen 'tail', um in linearer Zeit Elemente hinzufügen zu können.
//Da die Listenelemente gemäss Aufgabe nebst 'key' nur das Feld 'next'
//besitzt, wird dieses verwendet, um auf das vorherige Element zu zeigen.

//Zeiger des neuen Elements auf NULL setzen
e.next = NULL

//Element vor tail soll auf e zeigen
(tail.next).next = zeigeAuf(e)

//tail soll auf e zeigen
tail.next = zeigeAuf(e)
```

```
element dequeue()
//Wenn Liste leer, NULL zurückgeben und nichts verändern
if tail.next = head
  return NULL

element k = head.next
head.next = zeigeAuf((head.next).next)

//Wenn letztes vorhandenes Element entfernt werden soll:
//Durch obere Anweisung wurde head.next bereits auf NULL
//gesetzt. Nun muss noch tail auf head zeigen:
if head.next = NULL
  tail.next = zeigeAuf(head)

return k
```

`Start`: {(head) ← (tail)}
`ENQUEUE(3)`: {(head) → 3 ← (tail)}
`ENQUEUE(5)`: {(head) → 3 → 5 ← (tail)}
`DEQUEUE()`: {(head) → 5 ← (tail)}, return 3
`ENQUEUE(2)`: {(head) → 5 → 2 ← (tail)}
`DEQUEUE()`: {(head) → 2 ← (tail)}, return 5
`ENQUEUE(8)`: {(head) → 2 → 8 ← (tail)}
`ENQUEUE(9)`: {(head) → 2 → 8 → 9 ← (tail)}
`DEQUEUE()`: {(head) → 8 → 9 ← (tail)}, return 2
`DEQUEUE()`: {(head) → 9 ← (tail)}, return 8
`DEQUEUE()`: {(head) ← (tail)}, return 9
`DEQUEUE()`: {(head) ← (tail)}, return NULL
`Ende`: {(head) ← (tail)}

Aufgabe 3

Listing 3: Aufgabe 3

```
42 output(node n)
43 print n.key
44 if n.left-child != NULL
45   output(n.left-child)
46 if n.right-sibling != NULL
47   output(n.right-sibling)
```

Aufgabe 4

Listing 4: Aufgabe 4

```
48 output(node n)
49
50 while stack != empty
51   print n
52   if n.left-child != NULL
53     stack.push(n.left-child)
54   if n.right-sibling != NULL
55     stack.push(n.right-sibling)
56   n=stack.pop
```

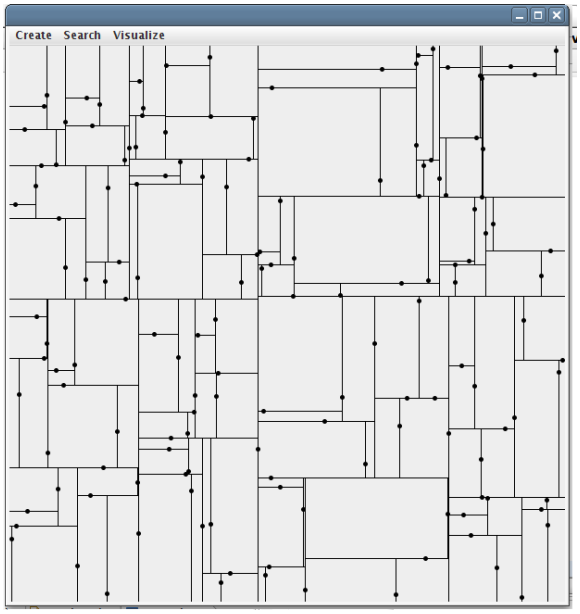
Praktische Aufgaben

Aufgabe 1

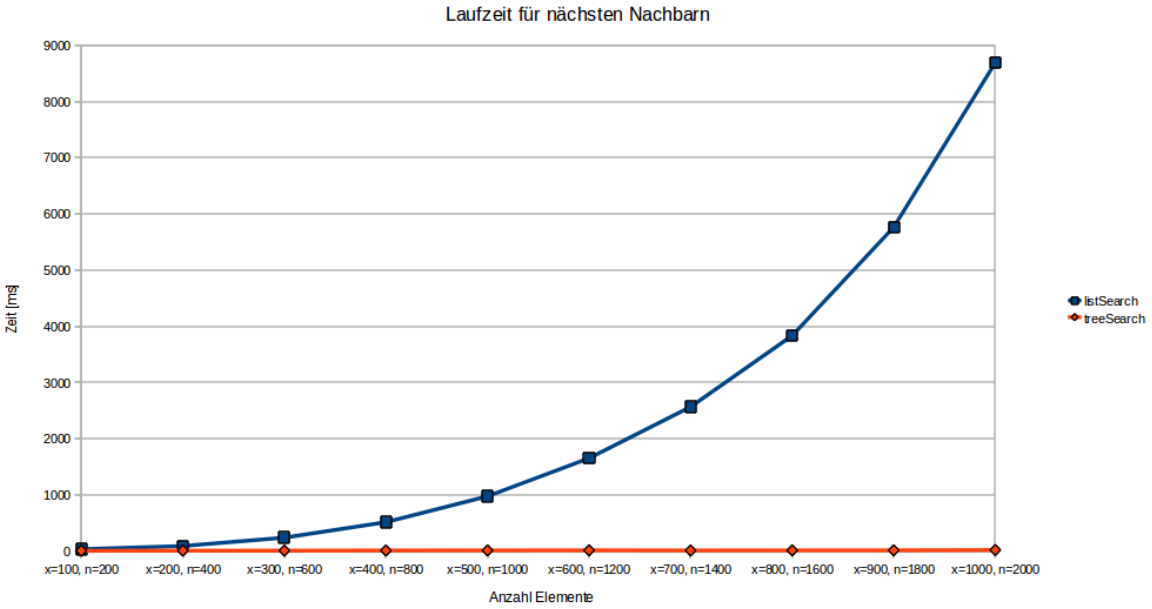
Siehe Anhang `KDTreeVisualization.java`.

Aufgabe 2

Siehe Anhang `KDTreeVisualization.java`.
Ausgabe *Visualize KD Tree*:


Create
Search
Visualize

Aufgabe 3


Laufzeit für nächsten Nachbarn
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
Zeit [ms]
x=100, n=200
x=200, n=400
x=300, n=600
x=400, n=800
x=500, n=1000
x=600, n=1200
x=700, n=1400
x=800, n=1600
x=900, n=1800
x=1000, n=2000
Anzahl Elemente
listSearch
treeSearch

Parameter	**listSearch**	**treeSearch**
x=100, n=200	36ms	8ms
x=200, n=400	92ms	11ms
x=300, n=600	241ms	10ms
x=400, n=800	517ms	13ms
x=500, n=1000	977ms	14ms
x=600, n=1200	1655ms	16ms
x=700, n=1400	2569ms	13ms
x=800, n=1600	3836ms	15ms
x=900, n=1800	5766ms	16ms
x=1000, n=2000	8693ms	21ms

Da die beiden Faktoren x und n, respektive Gesamtanzahl Punkt und zu suchende Punkte einander beinflussen, kann keine genaue Vorhersage getroffen werden. Die Grafik lässt aber vermuten, dass sich die beiden Laufzeitfunktionen $\Theta(n)$ resp. $\Theta(log(n))$ annähern.