

Übungsserie 5

Datenstrukturen & Algorithmen

Universität Bern
Frühling 2018

Übungsserie 5

- > 5 theoretische Aufgaben
 - Linked Lists
 - Trees
 - > 3 praktische Aufgaben
 - KD-Tree
 - Eher aufwändig!
 - > Poolstunde: Montag 17:00 – 18:00
-

Linked Lists

- > **Erinnerung** **Linked List** != **Array** !
 - Zugriff auf n -tes Element: $O(n)$ vs. $O(1)$
 - Insertion an bestimmter Position: $O(1)$ vs. $O(n)$

- > **Falsch**
 - Linked list A , $A[k]$ im (Pseudo)code

- > **Richtig**
 - Elemente der Reihe nach abarbeiten

```
current = head
while (current != Nil)
    next = current.next
    <work on current>
    current = next
```

Theoretische Aufgaben

- > **Aufgabe 1** Reihenfolge einer Liste umkehren
 - > **Aufgabe 2** Queue mittels Linked List realisieren
 - > **Aufgabe 5** Zwei sortierte Linked Lists zu einer sortierten Linked List zusammenfügen

 - > **!!! Verboten** Liste in Array kopieren & auf Array arbeiten !!!

 - > **Aufgabe 1 & 5** Keine neue Liste für den Output erstellen, Elemente der Inputliste(n) umhängen
-

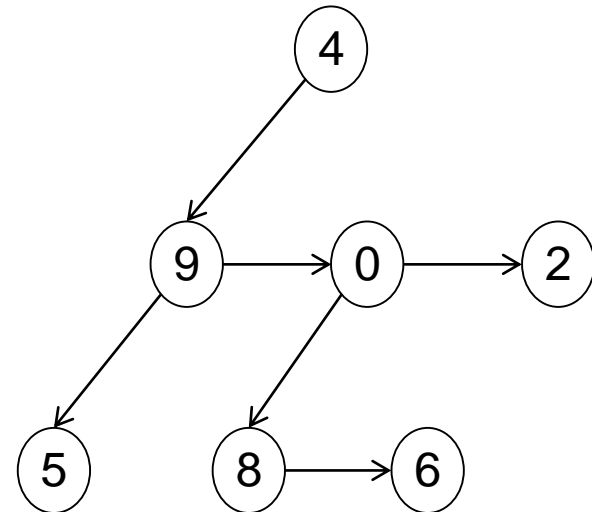
Theoretische Aufgaben 3 & 4

> Trees mit unbeschränktem Grad

— Spezielle Baumstruktur

— Node

- *key*
- *left-child*
- *right-sibling*



> **Aufgabe 3** Rekursiv traversieren

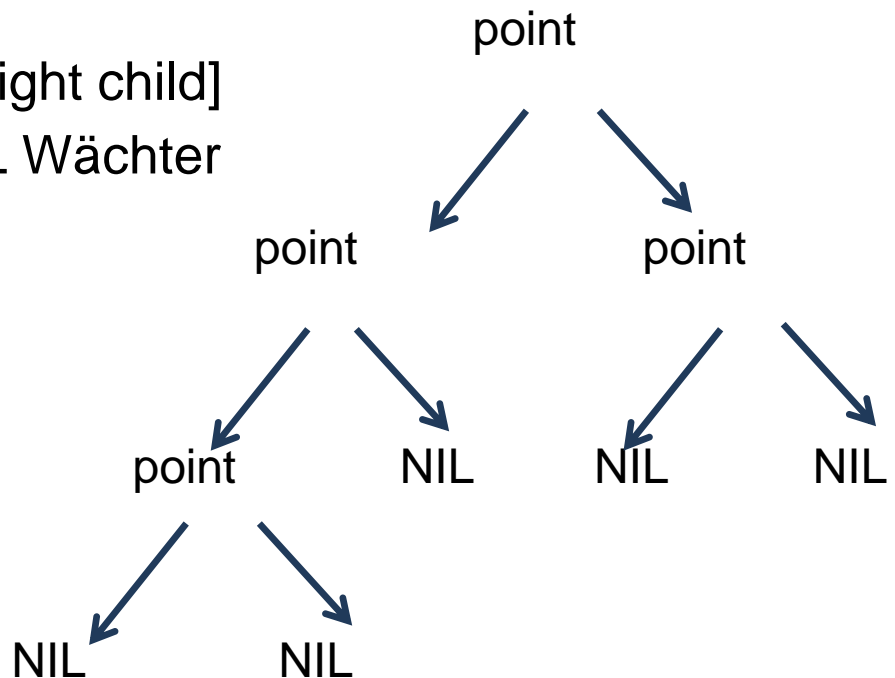
> **Aufgabe 4** NICHT-rekursiv traversieren, Stack benutzen!

Praktische Aufgaben

- > **Aufgabe 1** Nächste Nachbarn finden
 - > **Aufgabe 2** KD-Tree aufbauen
 - > **Aufgabe 3** Nächste Nachbarn im KD-Tree suchen
-

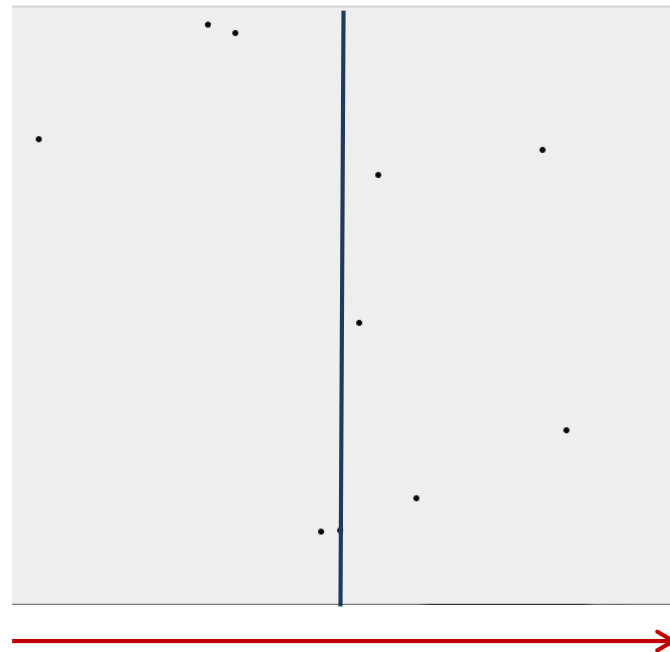
Praktische Aufgaben

- > Repetition KD-Tree
 - http://en.wikipedia.org/wiki/Kd_tree
- > **KD-Tree** Binärer Baum
 - Knoten besteht aus [Wert (2D Punkt), left child, right child]
 - Kein Folgeknoten: NULL/NIL Wächter



KD-Tree Aufbau

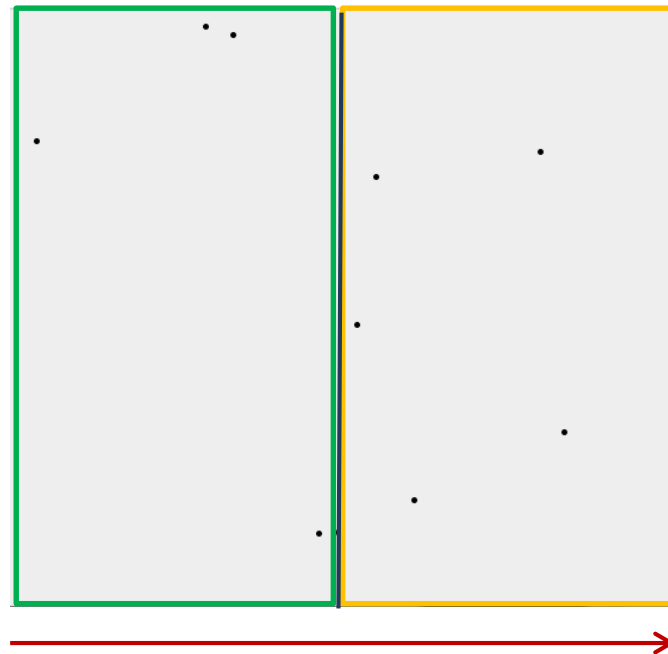
- > Vorlesung 5 Folien 63ff
- > `build(points, axis)`
 `median(points, axis)`



KD-Tree Aufbau

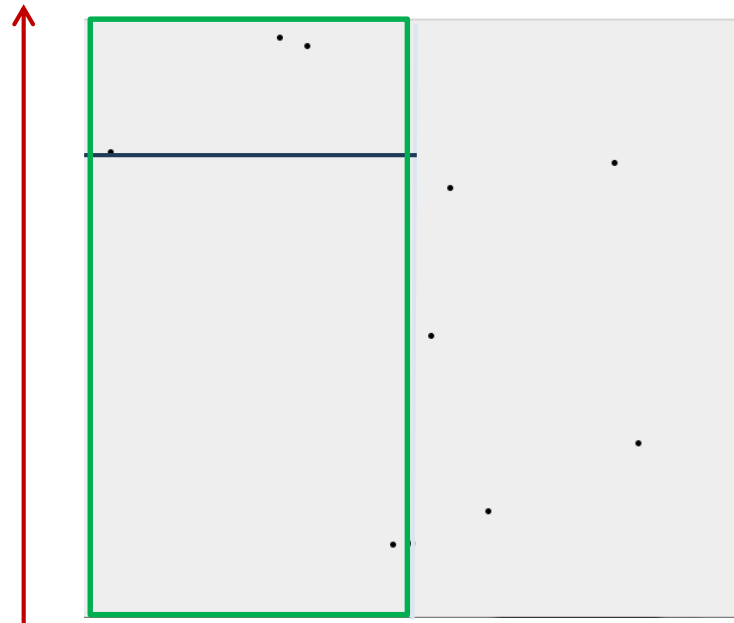
- > Vorlesung 5 Folien 63ff
- >

```
build(points, axis)
    median(points, axis)
    build(points<=median,
          switch(axis))
    build(points>median,
          switch(axis))
```

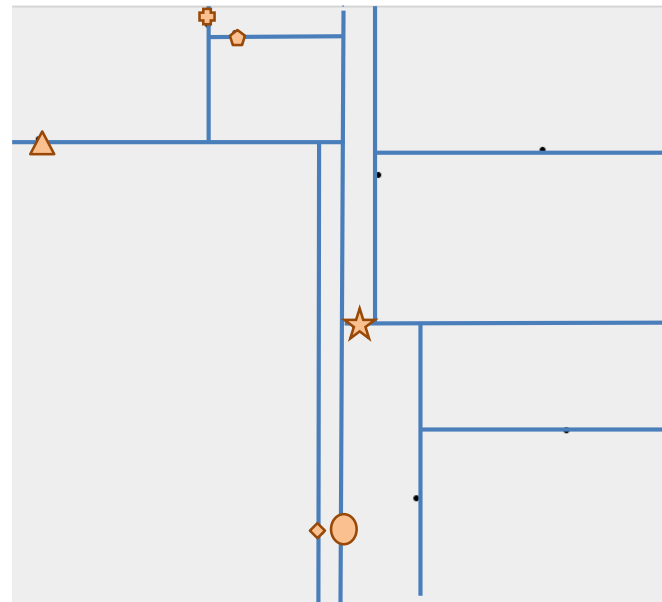
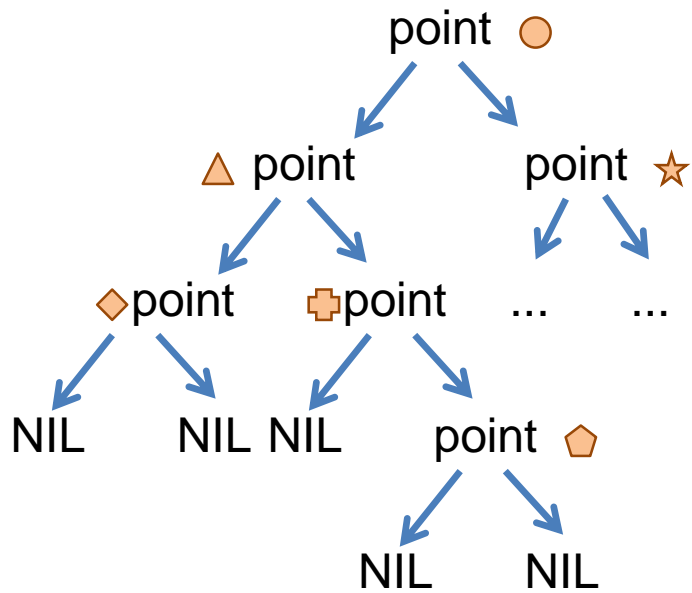


KD-Tree Aufbau

```
> build(points<=median, switch(axis))  
  // Axis switched!  
  median ...  
  build(points<=median,  
        switch(axis))  
  build(points>median,  
        switch(axis))  
  
etc...
```

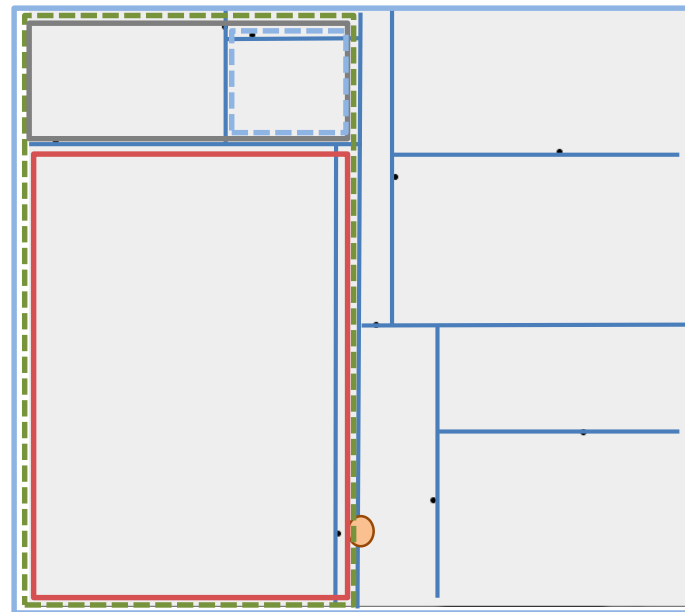
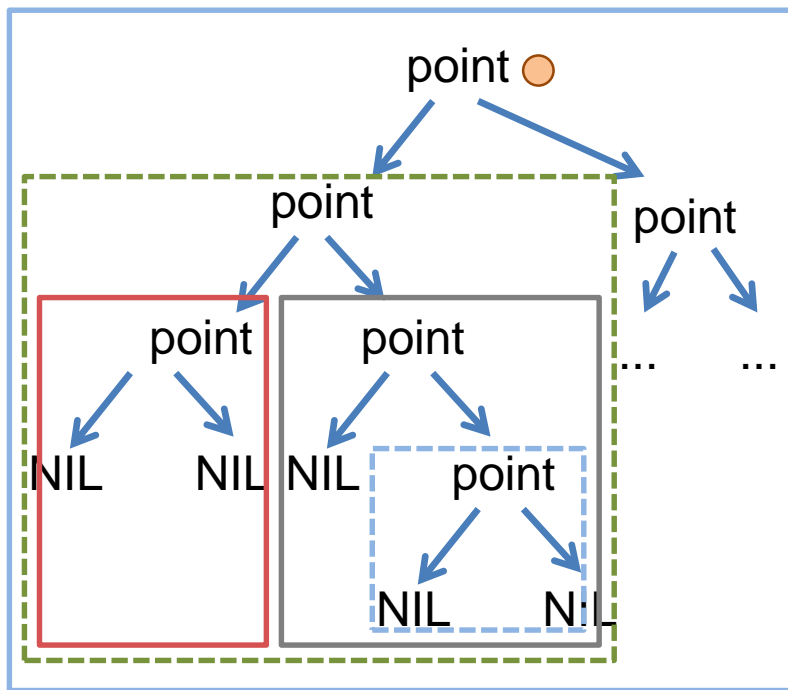


KD-Tree Aufbau



KD-Tree Eigenschaften

- > Punkte in Unterbäumen liegen in verschachtelten Volumen



KD-Tree Eigenschaften

- > Je nach Tiefe im Baum stellt die x oder die y Koordinate des Knotens die Split-Linie dar

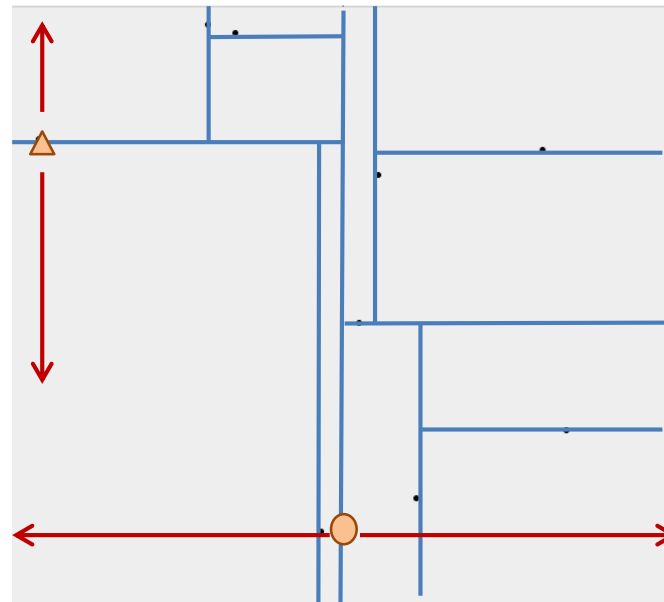
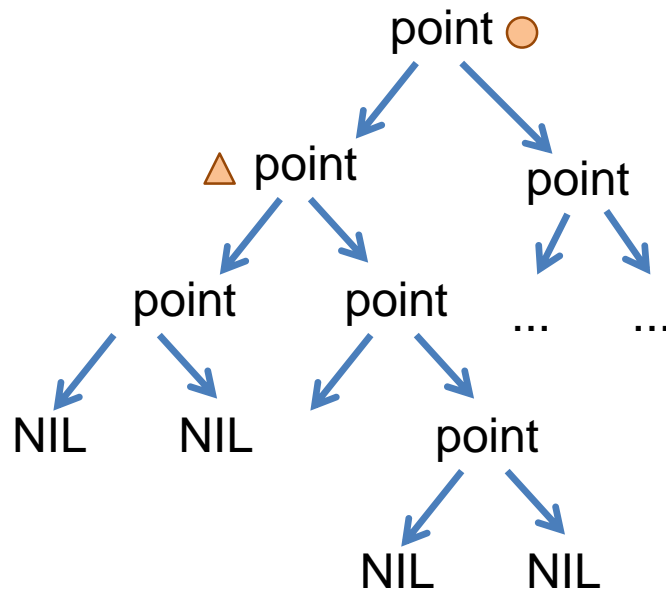
Achse

X

Y

X

Y



Find Nearest Neighbor (NN)

> Input

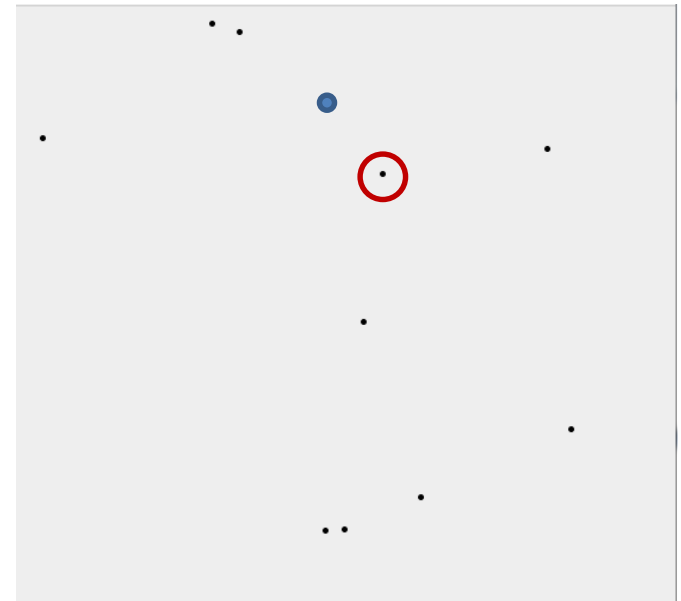
- Punktemenge A
- Abfragepunkt \bullet

> Output

- Der nächstgelegene Punkt \bigcirc
in A

> Bruteforce

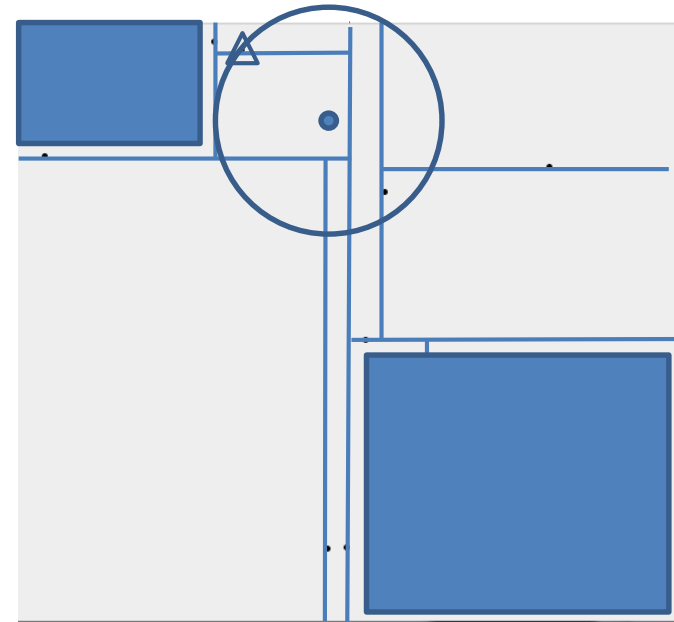
- Vergleiche alle Punkte
- $O(n)$



KD-Tree: findNN

- > **Idee** Gegeben einen NN-Kandidat Δ
 - Bessere Kandidaten liegen nicht in Zellen, die auf der entgegengesetzten Seite einer Split-Axis liegen, wenn die Achse weiter als der Kandidat entfernt ist

- > **Vorgeschlagener Algorithmus**
 - Traversiere Tree rekursiv
 - Überspringe Äste die weiter weg sind als der aktuelle Kandidat
 - etwas anders als in Buch/Vorlesung
 - leichter zu implementieren







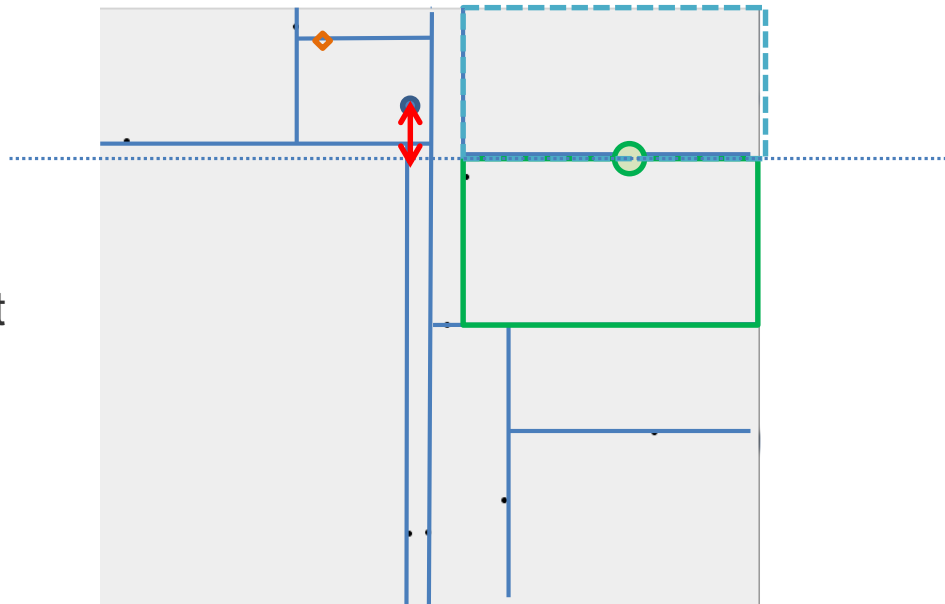
KD-Tree: findNN (rekursiv)

- >

```
Node findNN(node, candidate, point, depth):  
    If(node = NIL)  
        return candidate  
    If(node closer to point than candidate)  
        candidate = node  
  
    node_near = nearBranch(node, point, axis(depth))  
    node_far = farBranch(node, point, axis(depth))  
    // rekursiver Aufruf  
    candidate = findNN(node_near, candidate, point, depth+1)  
    if(farBranch closer than candidate)  
        candidate = findNN(node_far, candidate, point,  
                             depth+1)
```
- > **node_near**: Der Knoten der auf derselben Seite der Split-axis liegt wie point
- > Ohne `if(farBranch closer than candidate)` funktioniert der Code auch, ist aber $O(n)$!

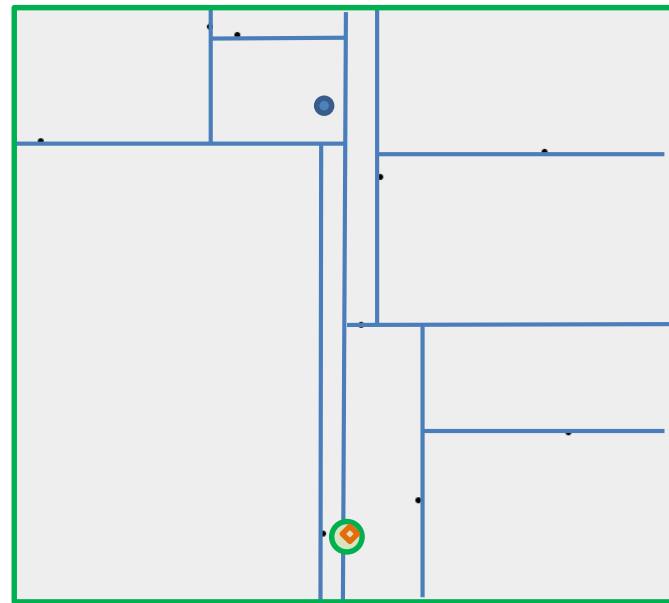
KD-Tree: findNN

- > `if(farBranch closer than candidate)`
 - Sei \diamond der aktuelle Kandidat, \circ der aktuelle Knoten und \bullet der Abfragepunkt
 - farBranch 
 - nearBranch 
 - Distanz zu far:
 -  = $|\text{point.y} - \circ.y|$
 -  ist zwar weiter Weg, aber das macht nichts



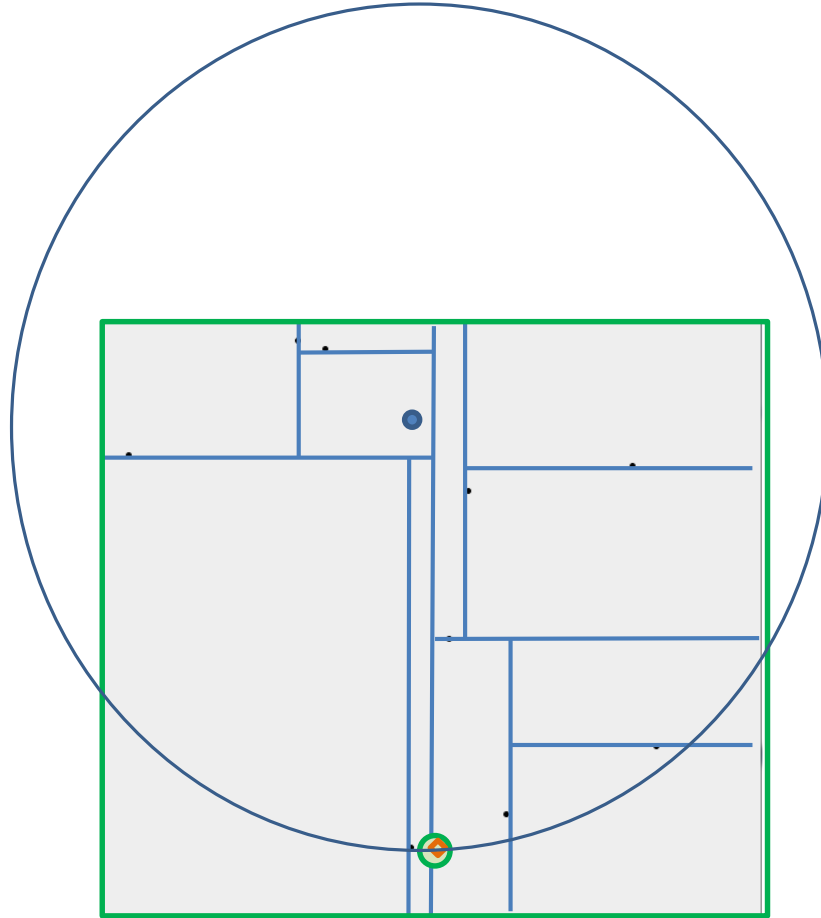
KD-Tree: findNN: Beispiel

- > Abfragepunkt •
- > Candidate ♦
- > Current node ○



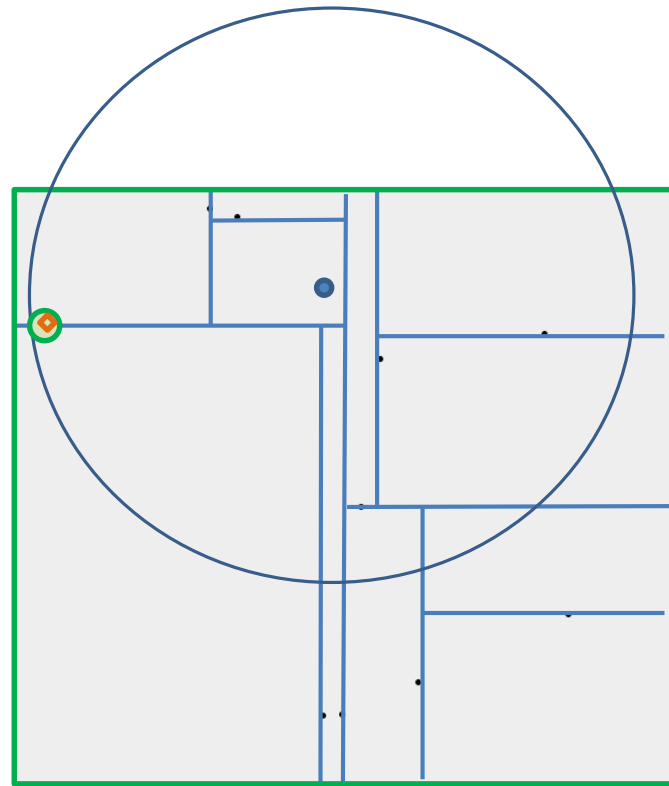
KD-Tree: findNN: Beispiel

- > Abfragepunkt •
- > Candidate ♦
- > Current node ○



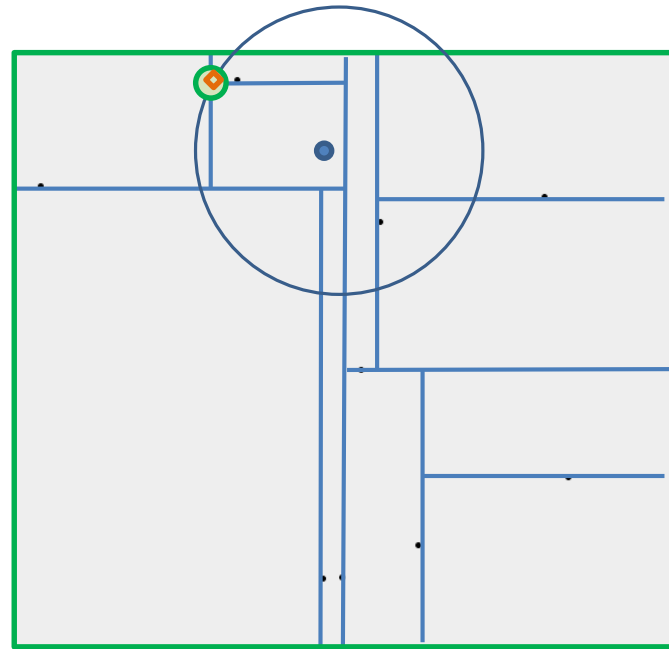
KD-Tree: findNN: Beispiel

- > Abfragepunkt •
- > Candidate ♦
- > Current node ○



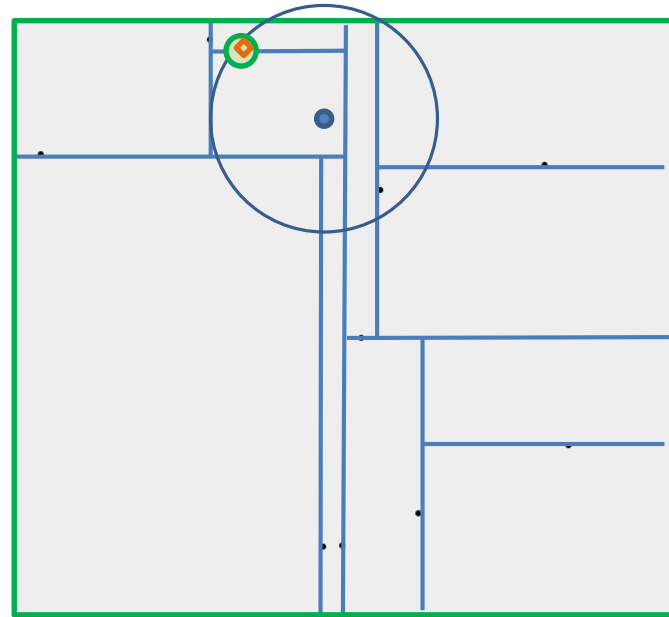
KD-Tree: findNN: Beispiel

- > Abfragepunkt •
- > Candidate ♦
- > Current node ○



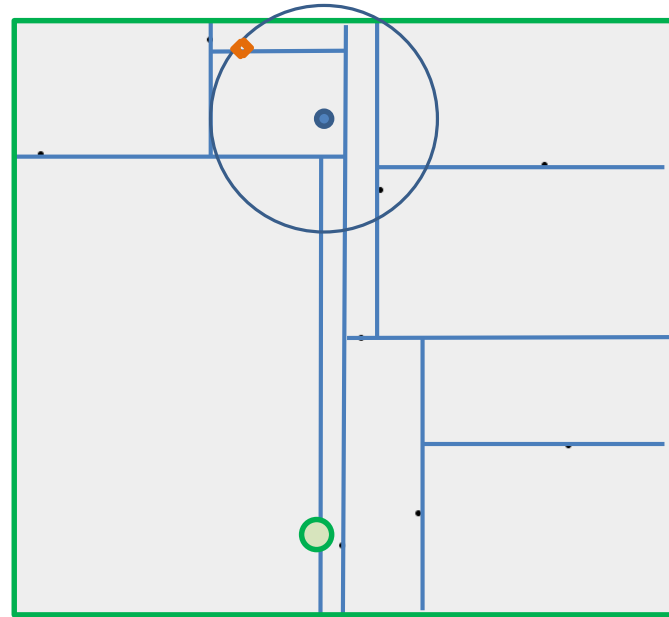
KD-Tree: findNN: Beispiel

- > Abfragepunkt •
- > Candidate ♦
- > Current node ○



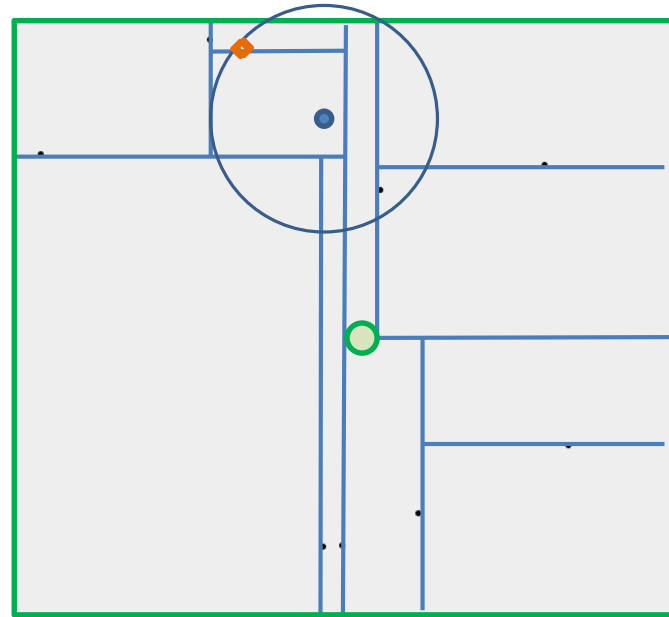
KD-Tree: findNN: Beispiel

- > Abfragepunkt •
- > Candidate ♦
- > Current node ○



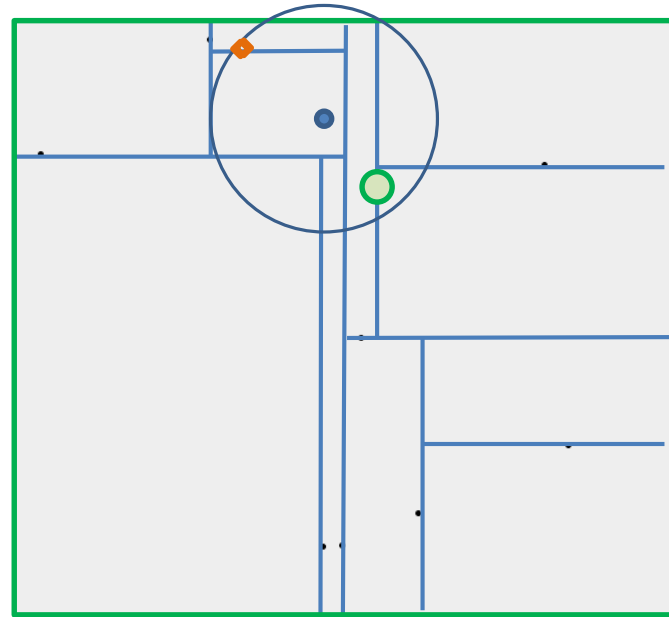
KD-Tree: findNN: Beispiel

- > Abfragepunkt •
- > Candidate ♦
- > Current node ○



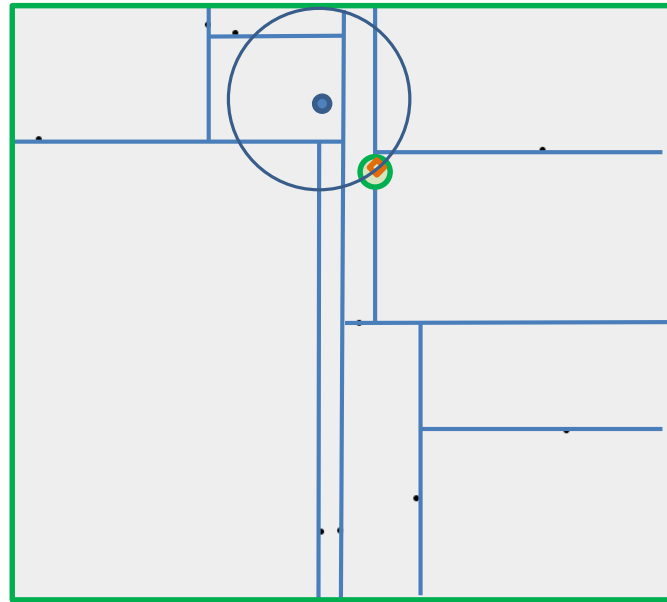
KD-Tree: findNN: Beispiel

- > Abfragepunkt •
- > Candidate ♦
- > Current node ○



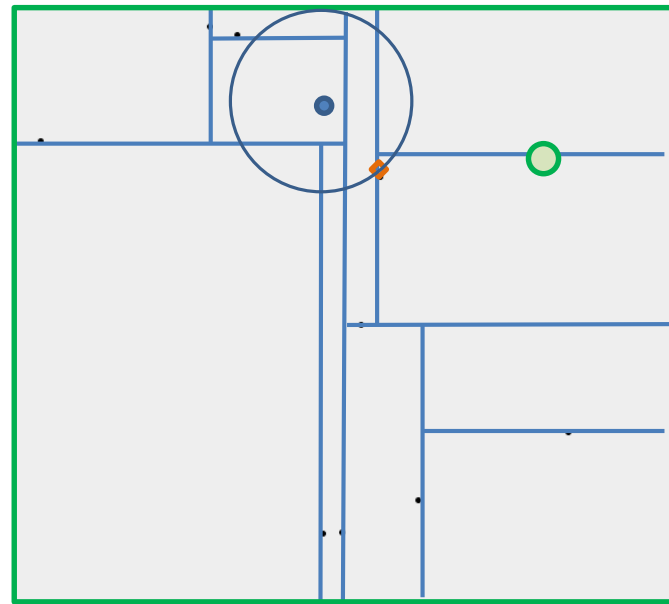
KD-Tree: findNN: Beispiel

- > Abfragepunkt •
- > Candidate ♦
- > Current node ○



KD-Tree: findNN: Beispiel

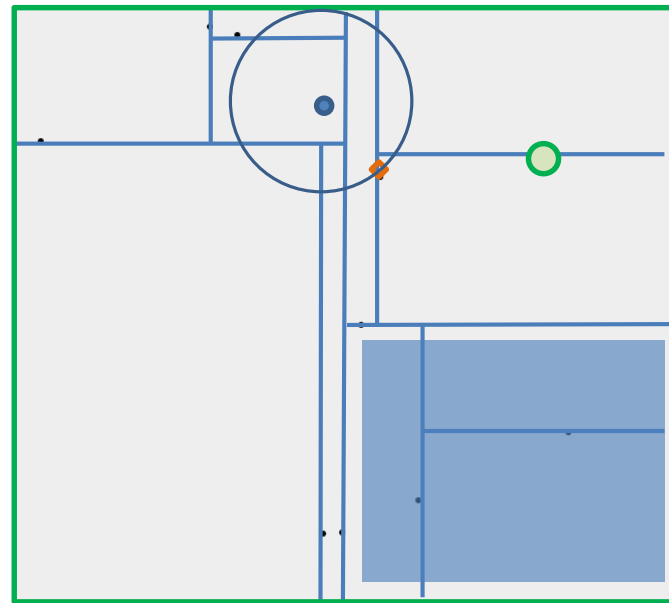
- > Abfragepunkt •
- > Candidate ♦
- > Current node ○



KD-Tree: findNN: Beispiel

- > Abfragepunkt •
- > Candidate ♦
- > Current node ○

- >  Ignoriert, zu weit weg



Praktische Aufgaben

- > Liste nach nächstem Nachbar (NN) durchsuchen, $O(n)$
 - > KD-Tree erstellen
 - > NN im KD-Tree suchen, $O(\log(n))$
 - > Zeitkomplexität der beiden Algorithmen experimentell bestätigen
-

Vorgegebener Code

- > `KDTreeTester.java`
 - Main
 - Braucht nicht verändert zu werden, ausser
 - n Anzahl Punkte
 - x Anzahl Abfragepunkte
 - w, h Breite, Höhe des Interfaces

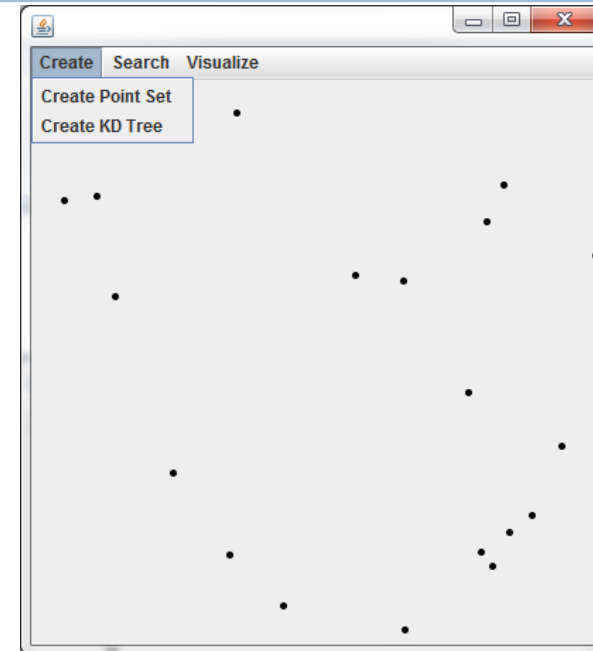
 - > `KDTreeVisualization.java`
 - **In dieser Klasse müsst ihr arbeiten**

 - > `PointComparator.java`
 - Vergleicht 2D Punkte bezüglich einer Achse

 - > `Timer.java`
-

KDTreeVisualisation.java

- > Einfaches Interface, ruft entsprechende Methoden auf
 - `initPoints()`
 - Zeigt Punkte an
 - `createPoints()`
 - Erzeugt Zufallspunkte
 - `createKDTree()`
 - `listSearchNN(Point p)`
 - `treeSearchNN(Point p)`
 - `TreeNode`
 - Private class, zur KD-Tree Implementation
 - `serachNN()`
 - Generiert x Abfragepunkte
 - Ruft entweder `treeSearchNN()` oder `listSearchNN()` auf
 - `visualize*()`
 - Visualisierungsmethoden
 - Implementiert, zu implementieren



KDTreeVisualisation.java

> Klassenvariablen

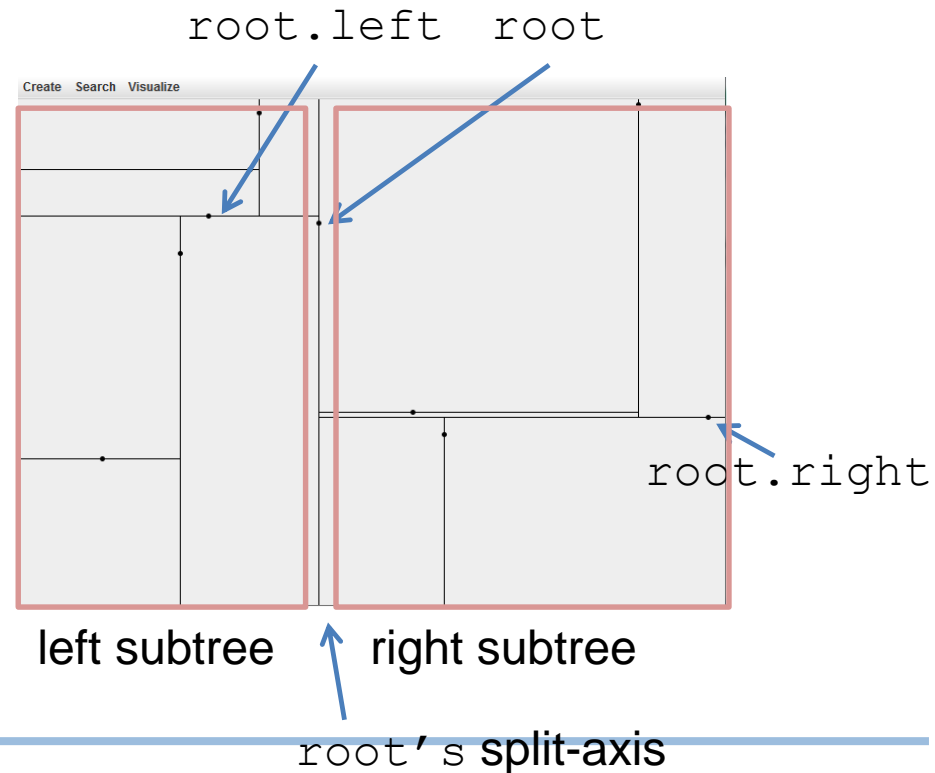
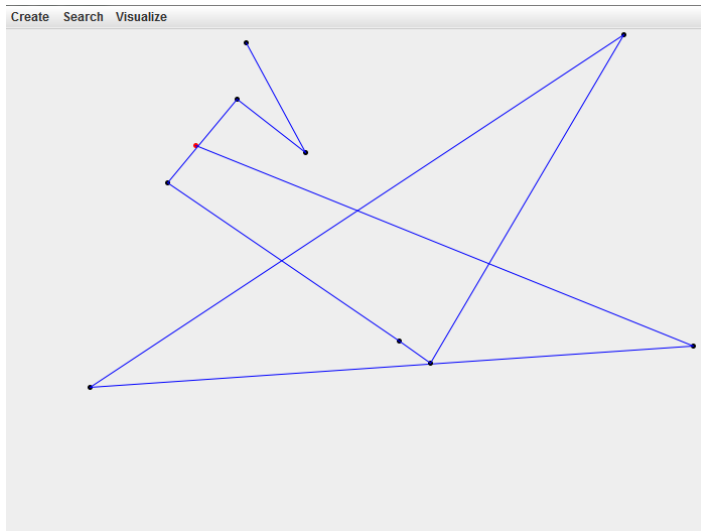
- `Treenode kdRoot`
 - Muss nach Aufruf von `createKDTree()` die KD-Tree Wurzel speichern, **sonst funktioniert die Visualisierung nicht!**
- `LinkedList<Point> points`
 - Liste der Punkte
- `int w, h`
 - Punkte liegen im Feld $[0, w] \times [0, h]$

KDTreeTester.java

> Main (implementiert)

— Visualisiert

- Liste
- KD-Tree (sobald ein KD-Tree erstellt wurde)



PointComparator.java

- > Konstruktor: `PointComparator(int i)`
 - Comparator zum Sortieren von Punkten entlang der i -ten Achse
 - `PointComparator(0)` Vergleicht nach x -Werten
 - `PointComparator(1)` Vergleicht nach y -Werten
-

Tipps

- > Median finden
 - `Java.util.Collections.sort` und `PointComparator`
 - Liste nach bestimmter Dimension sortieren ($O(n \log n)$)
 - Mittleres Element ist Median
 - Wäre auch in $O(n)$ möglich...

- > **Debugging** Teste, ob `treeSearchNN()` den selben Nachbar findet wie `listSearchNN()`