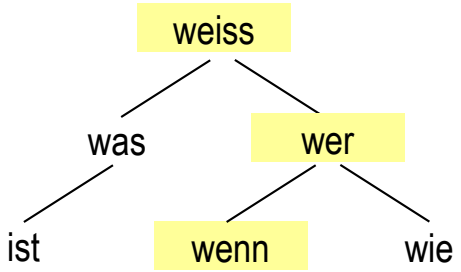
5. Tries und kd-Bäume

- Digitale Suchbäume (Tries)
- kd-Bäume

Tries (1)

Problem mit den bisherigen Suchbäumen

- Beim Suchen wird bei jedem durchlaufenen Knoten immer der komplette Schlüssel mit dem im Knoten abgespeicherten Schlüssel verglichen.
- Bei langen String-Schlüsseln kann das sehr aufwendig werden.

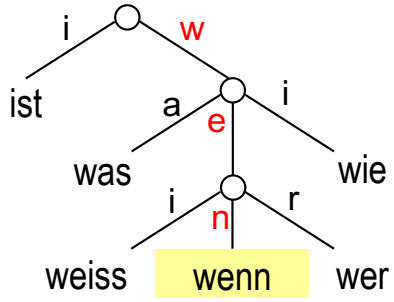


Suche nach „wenn“:

3 Vergleiche mit komplettem Schlüssel

Tries (aus retrieval; wird wie „try“ ausgesprochen)

- Die Daten sind bei den Blättern abgespeichert.
- Der Schlüssel wird ziffern- bzw. zeichenweise betrachtet (daher auch der Name digitale Suchbäume) und entschieden, in welchem Teilbaum rekursiv weitergesucht wird.
- Erst beim Blatt wird der komplette Schlüssel verglichen.



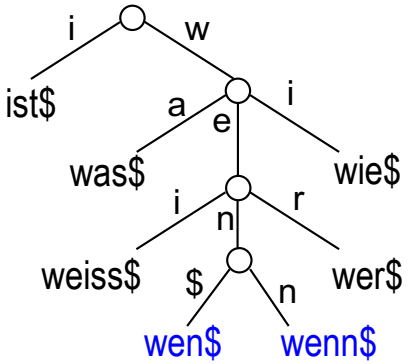
Suche nach „wenn“:

3 Zeichen-Vergleiche und
1 Vergleich mit komplettem Schlüssel

Tries (2)

Präfixproblematik:

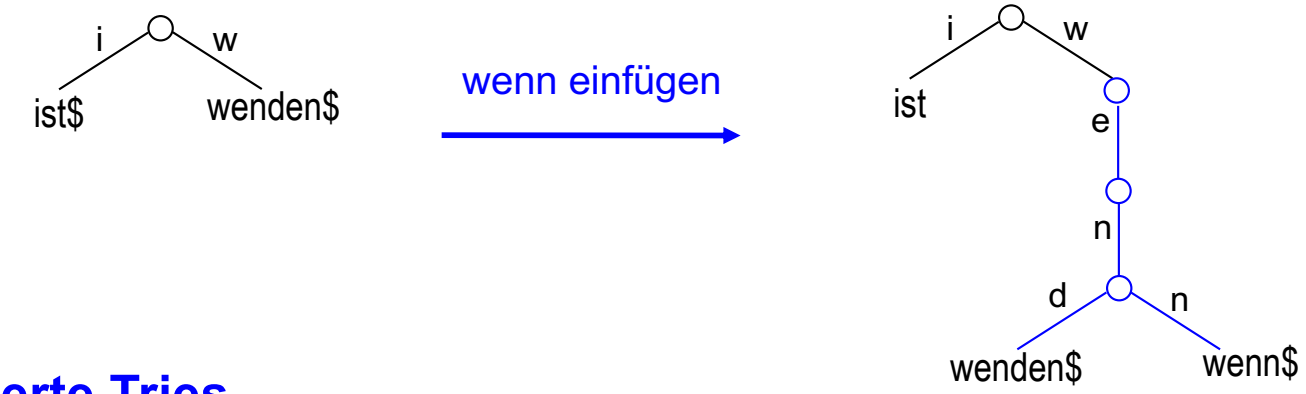
- Kein Schlüssel darf Präfix (Anfangsstück) eines anderen Schlüssels sein.
- Dies kann künstlich erreicht werden, indem jeder Schlüssel mit einem Spezialzeichen – z.B. '$' – abgeschlossen wird.



Tries (3)

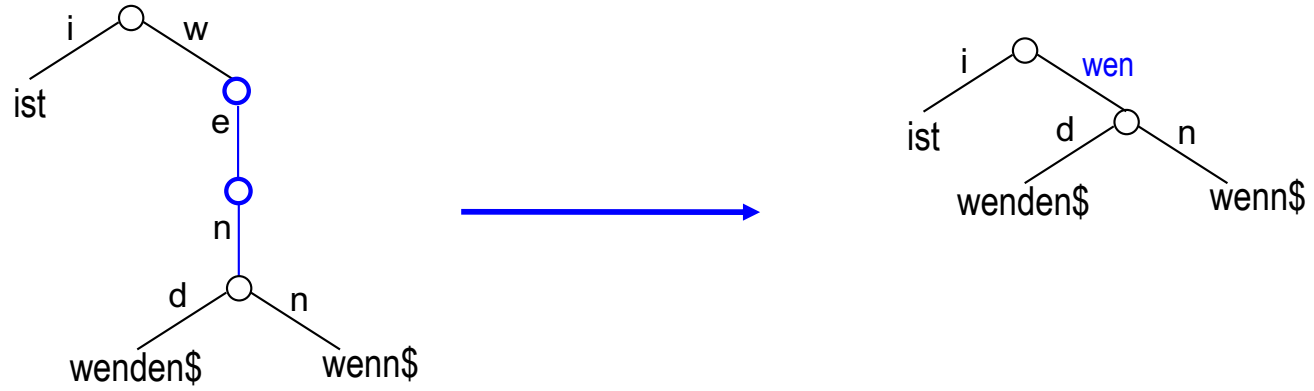
Einfügen:

- Beim Einfügen eines neuen Schlüssels müssen evtl. mehrere neue Knoten eingefügt werden.



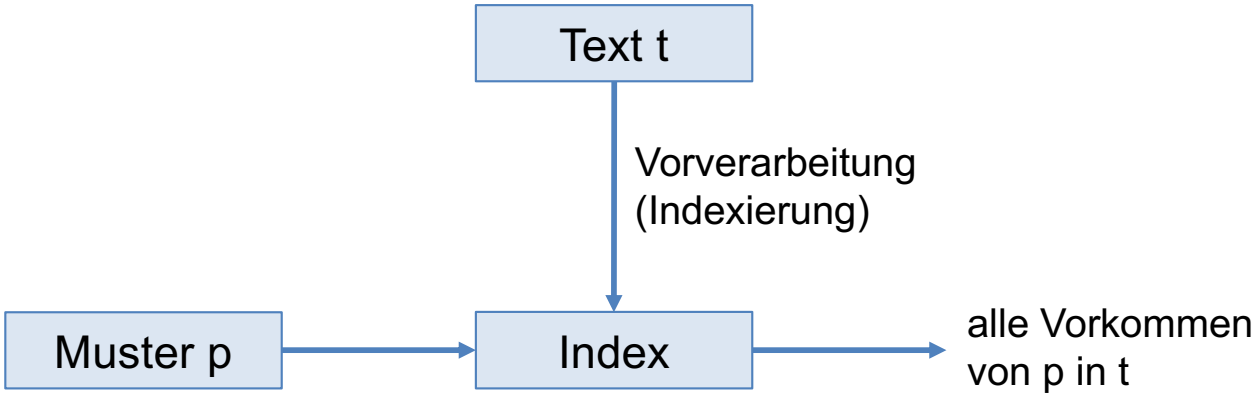
Kontrahierte Tries

- Eliminiere aufeinander folgende Knoten, die nur ein Kind haben.
  Für eine Verzweigung ist dann im allgemeinen nicht nur 1 Zeichen sondern ein ganzer String relevant.



Anwendung: Indexierungsverfahren

- Ziel: Finde alle Vorkommen eines Musters (pattern) p in einem (typischerweiser längerem) Text t.
- Dabei ist der Text t statisch (ändert sich nicht), so dass eine einmalige Vorverarbeitung (Indexierung) des Textes in Betracht kommt.
- Die eigentliche Mustersuche geschieht im Index und ist nur noch von der Länge des Musters abhängig.



- Ein wichtiger Ansatz für Indexierungsverfahren sind Suffixbäume

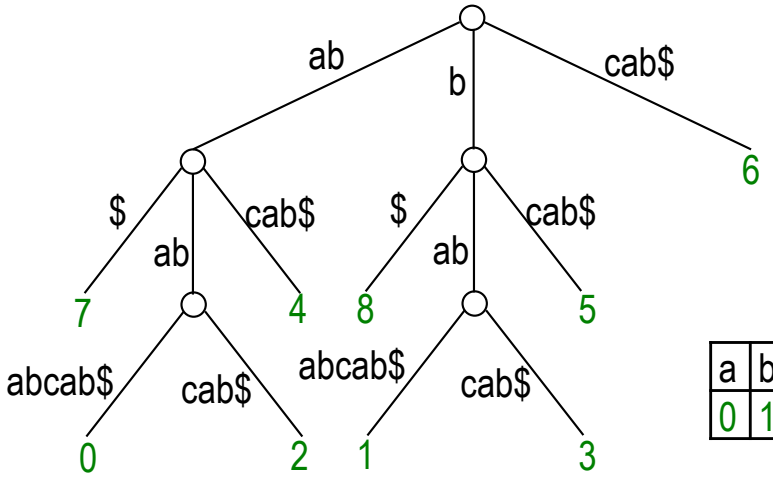
Suffixbaum

- Ein Suffixbaum für ein Text t ist ein kontrahierter Trie, der alle Suffixe des Textes t enthält.
- Um die Präfixproblematik zu vermeiden (kein Suffix darf Präfix eines anderen Suffixes sein), wird der Text mit einem Sonderzeichen '$' abgeschlossen, das sonst nicht im Text vorkommt.
- Jedes Blatt im Suffixbaum stellt genau ein Suffix des Textes dar. Im Blatt wird die Position des Suffix im Text abgespeichert.
- Beispiel:

Text t:
abababcab$

Suffixe von t:
abababcab$
bababcab$
ababcab$
babcab$
abcab$
bcab$
cab$
ab$
b$

Suffixbaum des Textes t:

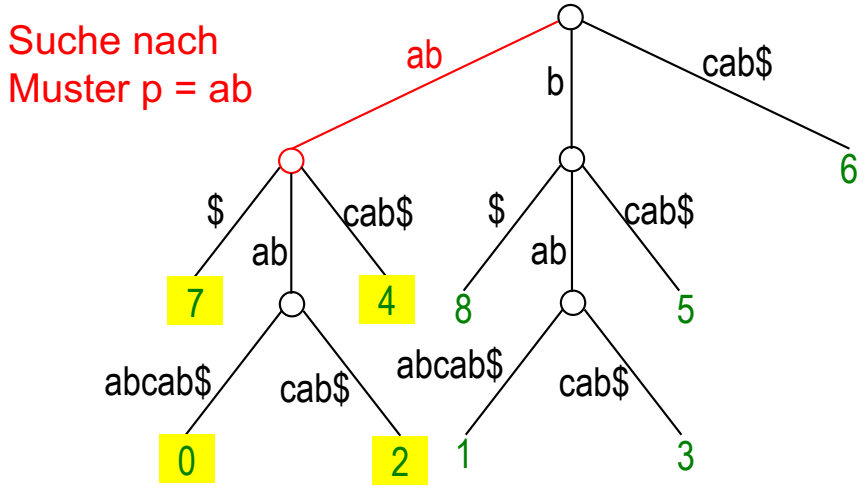


a	b	a	b	a	b	c	a	b	$	Text t
0	1	2	3	4	5	6	7	8		Position

Suche im Suffixbaum

- Um die Positionen eines Musters p in einem Text t zu ermitteln, wird das Muster p im Suffixbaum von t gesucht.
- Im allgemeinen endet die Suche bei einem inneren Knoten k.
- Alle Blätter des Teilbaums mit Wurzel k geben die gesuchten Positionen an.
- Beispiel:

Suffixbaum des Textes t = abababcab$

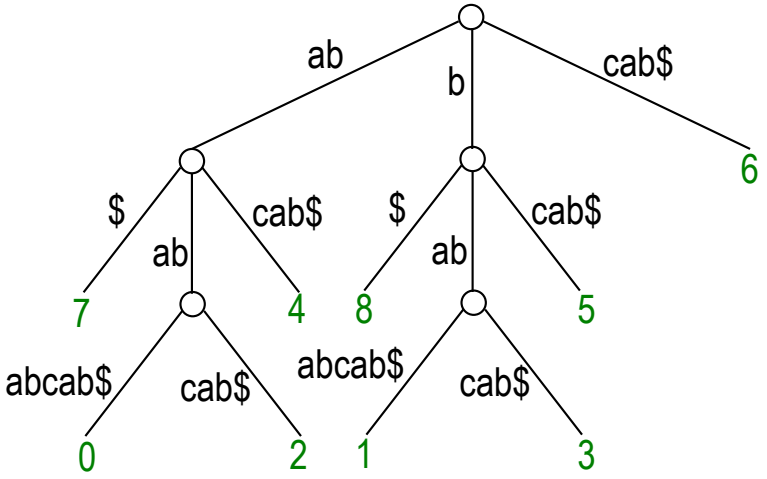


a	b	a	b	a	b	c	a	b	$	Text t
0	1	2	3	4	5	6	7	8		Position

Die gesuchten Positionen sind gelb markiert.

Komplexität

- Der Suffixbaum für ein Text t der Länge n hat genau n Blätter und maximal n-1 innere Knoten. Der String an den Kanten des Suffixbaums sind Teile des Textes und können durch Speicherung der Anfangs- und Endposition im Text dargestellt werden. Die Speicherung des Suffixbaums benötigt daher O(n).
- Die Konstruktion eines Suffixbaums für ein Text t der Länge n kann in T = O(n) durchgeführt werden. (siehe [Ottmann und Widmayer])
- Die Suche eines Musters p der Länge m in einem Suffixbaum benötigt T = O(m). Sollen alle Vorkommen des Musters p im Text t ermittelt werden, dann ist T = O(m+k) notwendig, wobei k die Anzahl der Vorkommen von p in t ist.



Suffixbaum des Textes
t = abababcab$ mit
8 Blättern und
5 inneren Knoten.

5. Tries und kd-Bäume

- Digitale Suchbäume (Tries)
- kd-Bäume

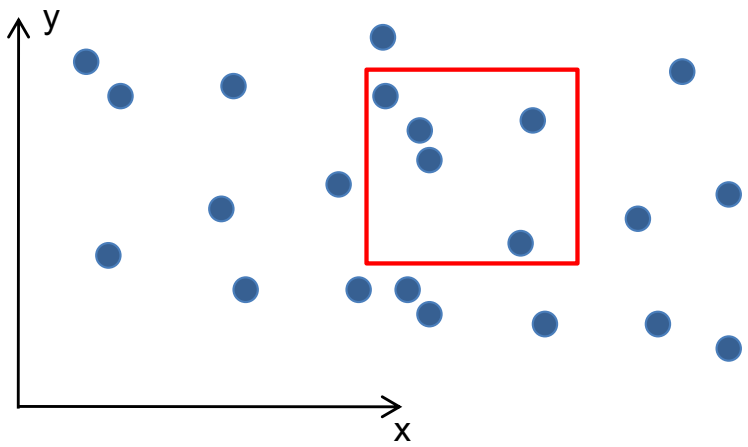
Motivation (1)

Ziel

- Verwaltung von Punkten bzw. Daten aus einem k-dimensionalen Raum in einem Suchbaum
- Hier: k = 2 (2-dimensional)
  Verallgemeinerung auf beliebiges k ist dann offensichtlich.

Typische Operationen

- (Orthogonale) Bereichssuche:
  finde alle Punkte (x,y) mit $x_{min} \leq x \leq x_{max}$ und $y_{min} \leq y \leq y_{max}$
- finde denjenigen Punkt (x,y), der zu einem gegebenen Punkt (x_0, y_0) am nächsten liegt (nearest neighbour search).

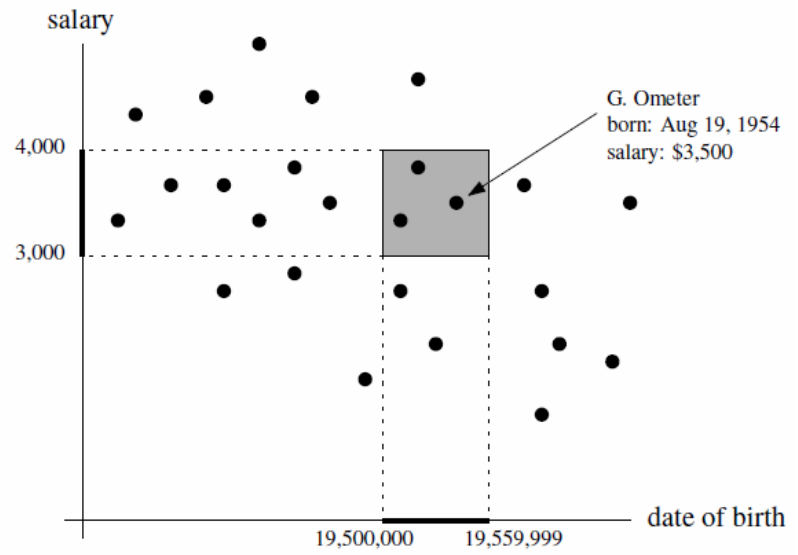


Bereichssuche:
suche alle Punkte im
rot umrandeten Bereich

Motivation (2)

Beispiel: 2-dimensionale Bereichssuche bei Personendaten

- Suche alle Personen mit Einkommen zwischen 3000 und 4000 USD und einem Geburtsdatum zwichen 01.01.1950 (= 19500101) und 31.12.1955 (= 19551231)



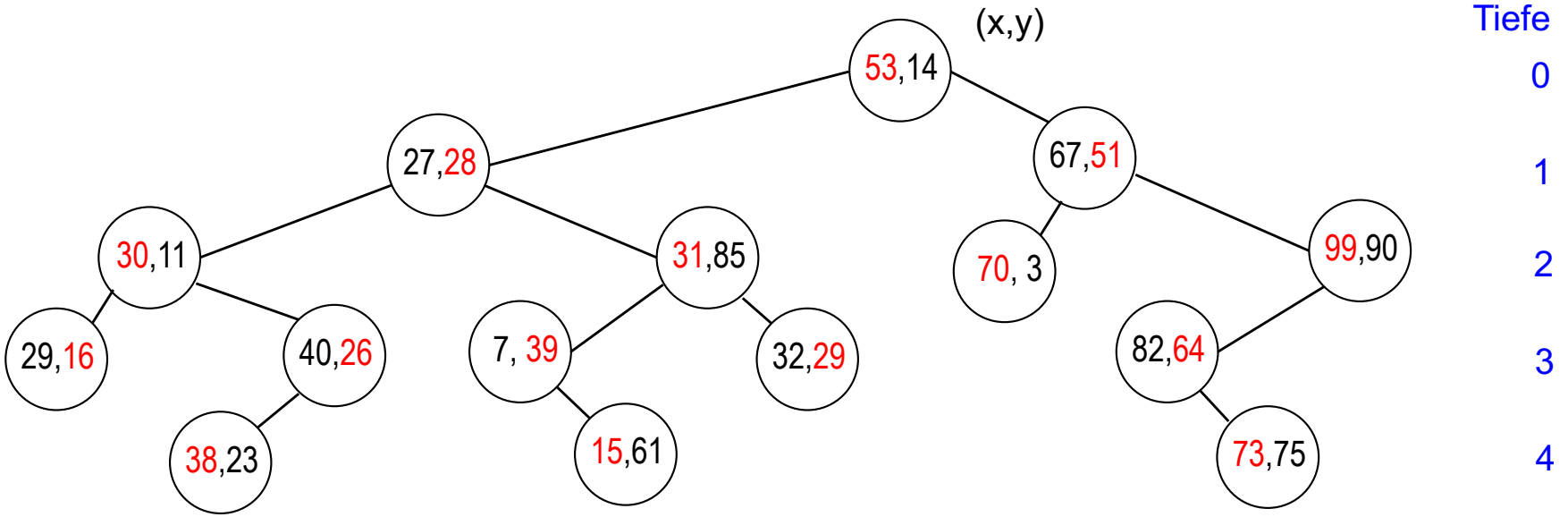
Beispiel aus Berg et al., Computational Geometry, Springer Verlag.

Bemerkung

- Bereichssuche lässt sich geometrisch interpretieren.
- Daher sind kd-Bäume in der Literatur auch oft im Bereich der geometrischen Algorithmen (computational geometry) zu finden.

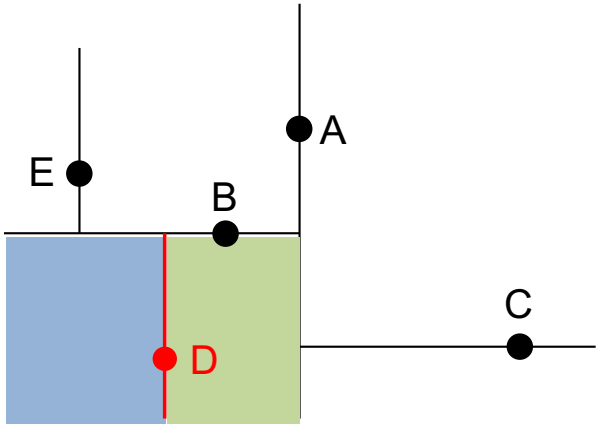
2d-Baum

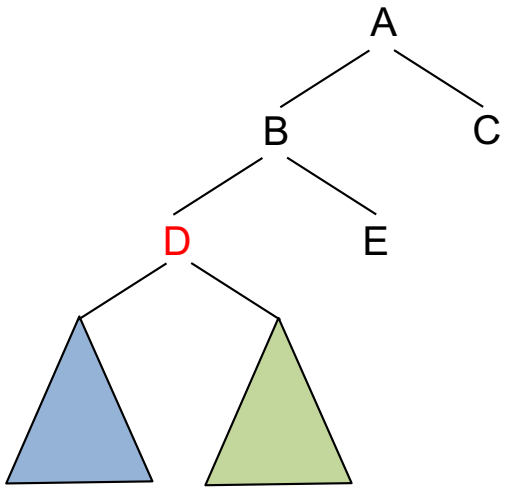
- ein 2d–Baum ist ein binärer Baum mit folgenden Eigenschaften:
- jeder Knoten speichert einen 2-dimensionalen Punkt (Daten)
- für jeden Knoten p in einer gradzahligen Tiefe gilt, dass alle Knoten im linken
- Teilbaum von p eine kleinere (oder gleiche) x-Koordinate und alle Knoten im rechten Teilbaum von p eine größere (oder gleiche) x-Koordinate haben.
- für jeden Knoten p in einer ungradzahligen Tiefe gilt, dass alle Knoten im linken Teilbaum von p eine kleinere (oder gleiche) y-Koordinate und alle Knoten im rechten Teilbaum von p eine größere (oder gleiche) y-Koordinate haben.



Geometrische Interpretation eines 2d-Baums

- Die Wurzel (Tiefe 0 ist gradzahlig) teilt die Ebene vertikal in zwei Teilebenen.
- Die beiden Kinder der Wurzel (Tiefe 1 ist ungradzahlig) teilen die beiden Teilebenen jeweils horizontal
- Die Enkel der Wurzel (Tiefe 2) teilen die Teilebenen wieder vertikal.
- In der nächsten Tiefe wird wieder horizontal geteilt; usw.





kd-Baum in Java als Klasse

```
public class KdTree {

   private class KdNode {
      double [ ] data;
      KdNode left;
      KdNode right;
      KdNode(double[ ] d, KdNode l, KdNode r) {
         data = d;
         left = l;
         right = r;
      }
   }

   private KdNode root = null;

   // ...
}
```

Feld der Größe 2 für 2-dimensionalen Punkt.

2-dimensionale Bereichsuche

```
public void printRange(double[ ] min, double[ ] max) {
    printRangeR(root, 0, min, max);
}

private void printRangeR(KdNode p, int komp, double[ ] min, double[ ] max) {
    if (p == null)
        return;
    if (min[0] <= p.data[0] && p.data[0] <= max[0] && min[1] <= p.data[1] && p.data[1] <= max[1])
        System.out.println("(" + p.data[0] + "," + p.data[1] + ")");
    if (min[komp] <= p.data[komp])
        printRangeR(p.left, 1-komp, min, max);
    if (max[komp] >= p.data[komp])
        printRangeR(p.right, 1-komp, min, max);
}
```

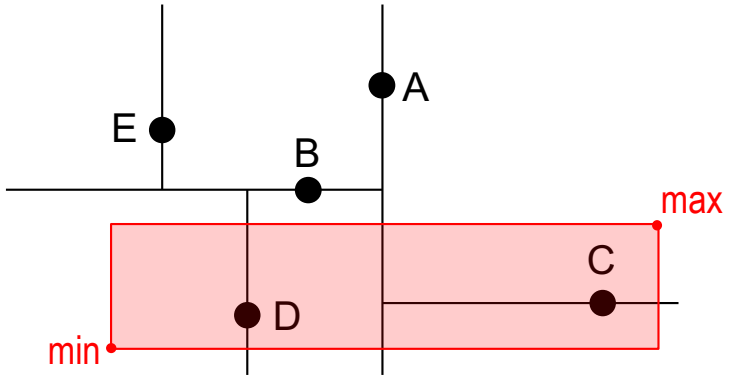
Gibt alle Punkte (x,y) aus, für die gilt:

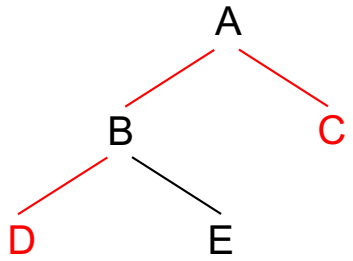
min[0] <= x <= max[0] und
min[1] <= y <= max[1]

komp = 0: gradzahlige Tiefe; es wird nach der x-Achse geordnet.

komp = 1: ungradzahlige Tiefe; es wird nach der y-Achse geordnet.

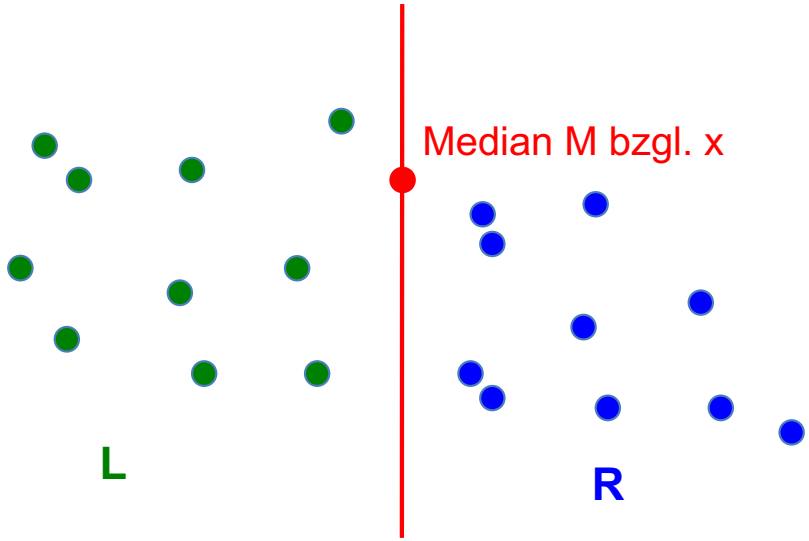
Suche alle Punkte im roten Bereich

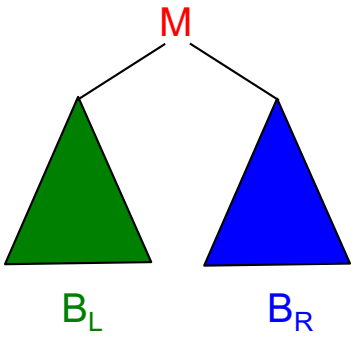




Konstruktion eines balanzierten 2d-Baums (1)

- Ziel: konstruiere für ein Feld von Punkten einen balanzierten 2d-Baum.
- Teile-und-Herrsche-Verfahren.
- Bestimme bzgl. x (bzw. y) den Median M und ordne Punkte im Feld so um, dass in der linken Hälfte L die Punkte <= M und der rechten Hälfte R die Punkte >= M sind.
- Konstruiere durch rekursiven Aufruf aus L und R jeweils einen kd-Baum B_L und B_R
- Baue Baum mit M als Wurzel und B_L und B_R als linken bzw. rechten Teilbaum.





Konstruktion eines balanzierten 2d-Baums (2)

```
public KdTree(double[ ][ ] pts) {
    root = createKdTree(pts, 0, pts.length-1, 0);
}

private KdNode createKdTree(double[ ][ ] pts, int li, int re, int komp) {
    if (li > re)
        return null;
    else if (li == re)
        return new KdNode(pts[li], null, null);
    else {
        int m = (li+re)/2;
        quickSelect(pts, li, re, m, komp);

        return
            new KdNode(pts[m],
                createKdTree(pts, li, m-1, 1-komp),
                createKdTree(pts, m+1, re, 1-komp));
    }
}
```

Konstruiere aus einem Feld pts von 2-dimensionalen Punkten einen kd-Baum

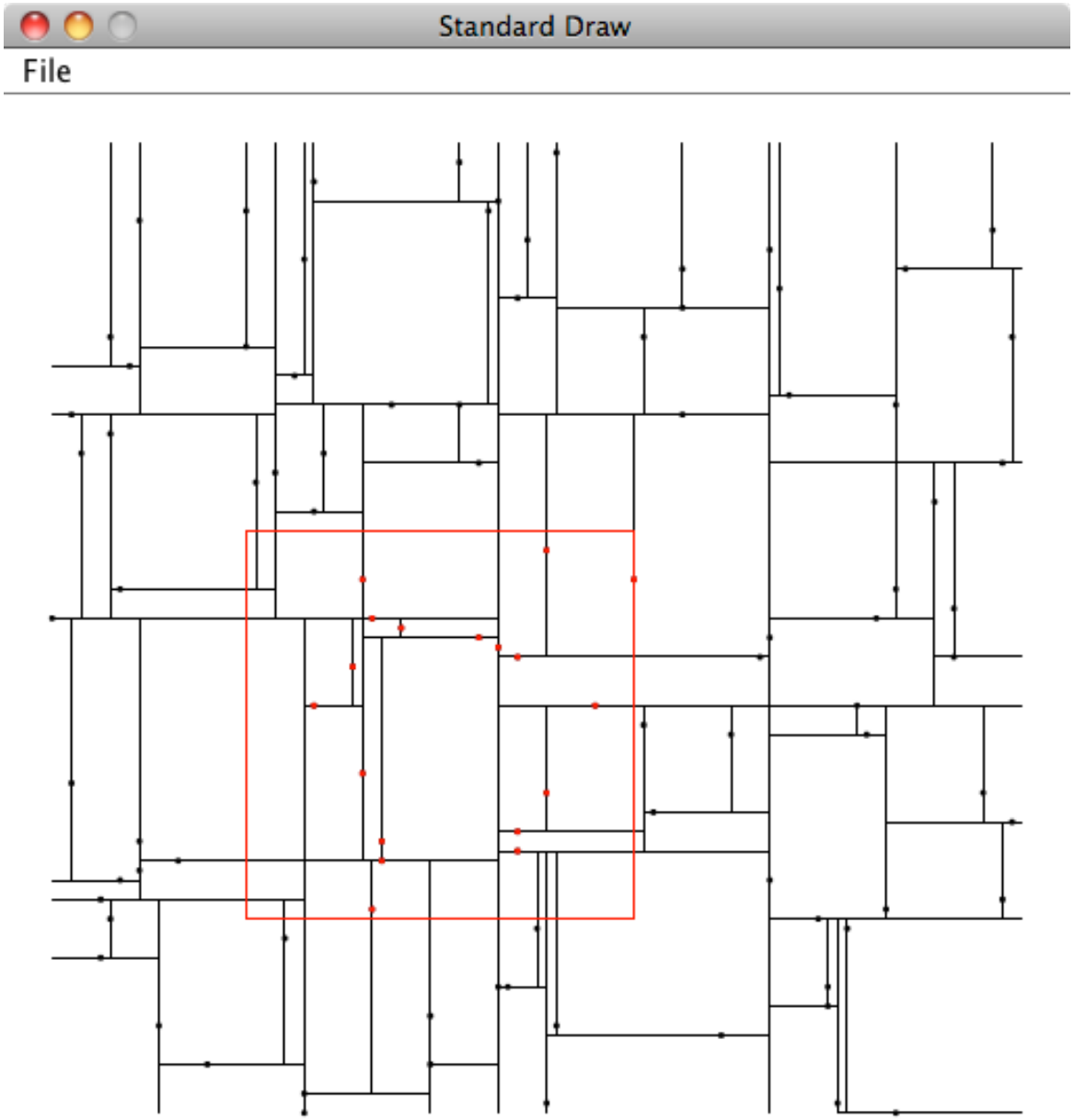
createKdTree konstruiert aus dem Punkteteilfeld pts[li] ... pts[re] einen balanzierten kd-Baum.

Funktion arbeitet nach dem Teil-und-Herrsche-Prinzip.

quickSelect ordnet das Punkteteilfeld pts[li] ... pts[re] um, so dass in pts[m] der Median steht und links von m alle Elemente <= pts[m] und rechts von m alle Elemente >= pts[m] sind.

Bei komp = 0 wird bzgl. des x-Werts verglichen und bei komp = 1 bzgl. des y-Werts.

2d-Baum mit n = 100 zufällig gewählten Punkten und Bereichsabfrage



Analyse

Bereichssuche für balanzierten kd-Baum mit n Punkten:

- $T_{max}(n) = O(\sqrt{n})$

Konstruktion eines balanzierten kd-Baums mit n Punkten:

- $T_{max}(n) = O(n \log n)$
- Begründung: Teile-und-Herrsche-Vefahren,
  wobei sich quickSelect in O(n) realisieren lässt (siehe z.B. [Cormen et al.]).
- Alternativ lässt sich quickSelect auch als quickSort-Variante formulieren
  (einfach zu realisieren; siehe Programmiertechnik II).
  Jedoch ist dann für quickSelect nur die durchschnittliche Laufzeit O(n).

Einfügen und Löschen:

- Einfügen und Löschen lässt sich ähnlich wie bei binären Suchbäumen realisieren.
- Jedoch sind keine effizienten Operationen für Einfügen und Löschen bekannt,
  die den Baum balanziert lassen.
- Rotationstechnik wie bei AVL-Bäumen lässt sich hier nicht anwenden.
- Daher: nach einer Folge von Einfüge- und Lösch-Operationen, Baum rebalanzieren
  mit einer Technik wie bei der Baum-Konstruktion.