

Informatik II - SS 2014

(Algorithmen & Datenstrukturen)

Vorlesung 11 (4.6.2014)

Binäre Suchbäume II



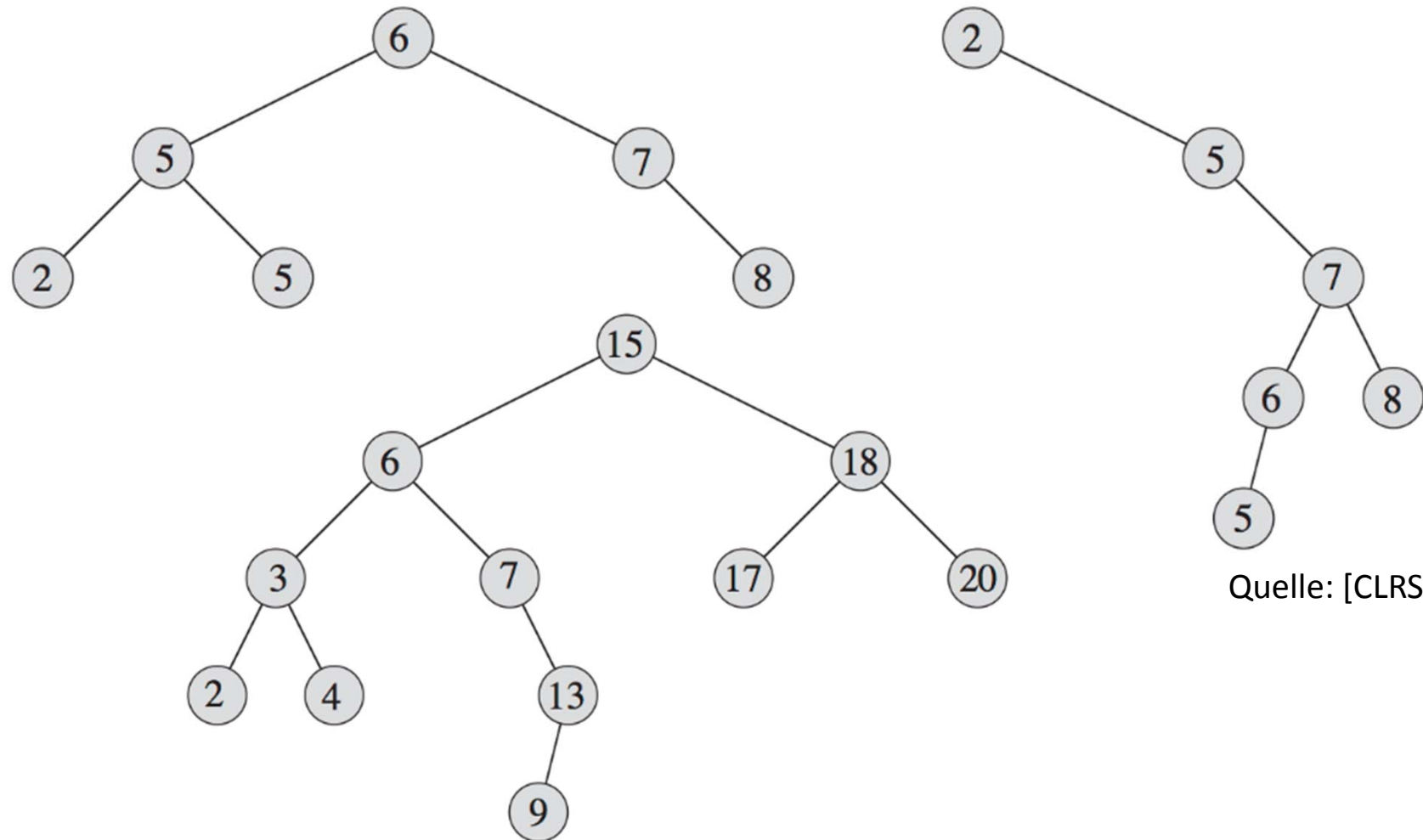
**UNI
FREIBURG**

Fabian Kuhn

Algorithmen und Komplexität

Binäre Suchbäume

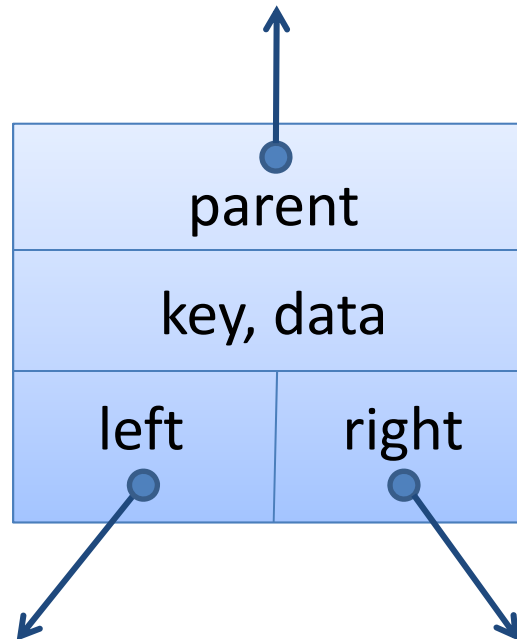
- Binäre Suchbäume müssen nicht immer so schön symmetrisch sein...



Quelle: [CLRS]

Binärer Suchbaum : Elemente

TreeElement:



Implementierung: gleich wie bei den Listen-Elementen

Die Operationen

find, min, max, predecessor, successor, insert, delete

haben alle **Laufzeit $O(\text{Tiefe des Baums})$** .

Was ist die Tiefe eines binären Suchbaums?

- Best Case: $\Theta(\log n)$
- Worst Case: $\Theta(n)$

Programmieren einer Binary Search Tree Klasse

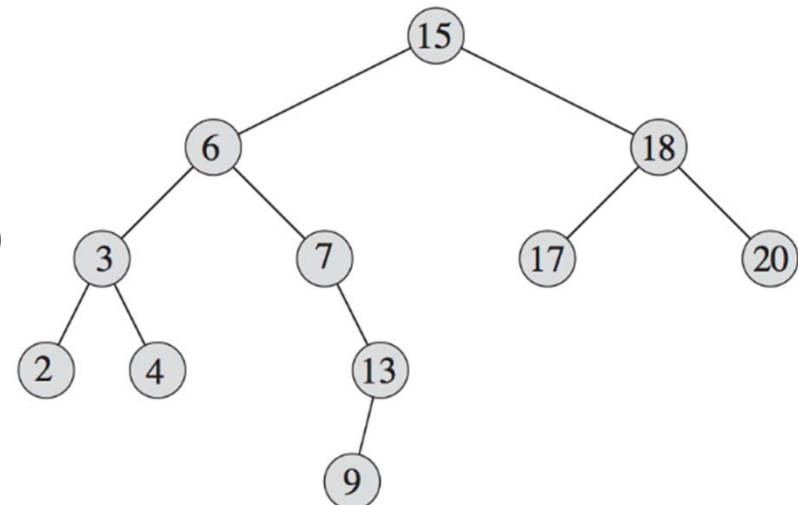
- Sprache: C++, Java, Python
- Operationen: find, insert, delete, size
- Zusätzlich:
 - toArray (gibt sortiertes Array zurück)
 - avgDepth (gibt durchschnittliche Knotentiefe zurück)
- Vorgabe: Struktur der Klasse
 - Signatur aller public Methoden
- Wir werden jetzt gleich mal die C++-Vorgabe anschauen und eine erste Methode programmieren...

Sortieren mit binärem Suchbaum

1. Füge alle Elemente in einen binären Suchbaum ein
2. Lese die Elemente in sortierter Reihenfolge aus
 - Einfachste Lösung: suche und entferne das Minimum
 - Oder: suche Minimum und dann $n - 1$ Mal *getSuccessor*

Bessere Lösung: Auslesen aller Elemente:

- Rekursiv:
 1. Lese linken Teilbaum aus (rekursiv)
 2. Lese Wurzel aus
 3. Lese rechten Teilbaum aus (rekursiv)



Auslesen eines Teils der Elemente

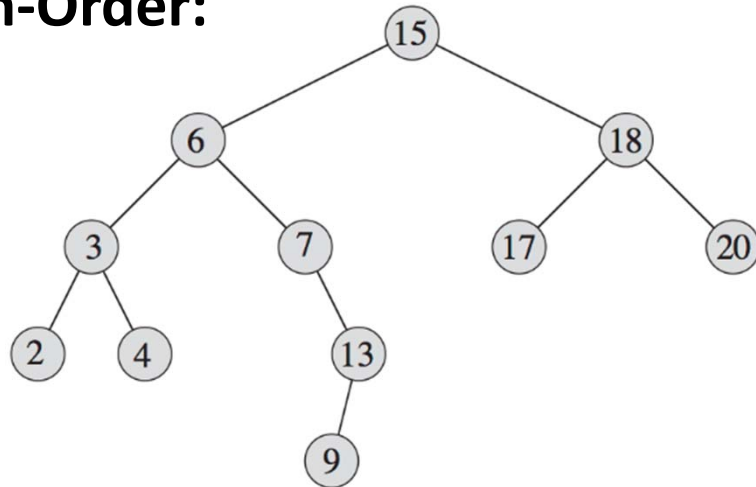
Gegeben: Schlüssel x_{\min} und x_{\max} ($x_{\min} \leq x_{\max}$)

Ziel: Gebe **alle Schlüssel x , $x_{\min} \leq x \leq x_{\max}$** aus.

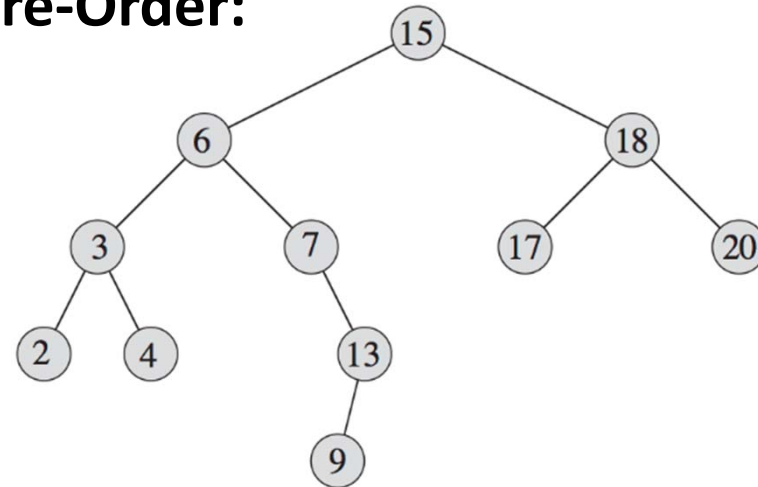
Traversieren eines binären Suchbaums

Ziel: Besuche alle Knoten eines binären Suchbaums einmal

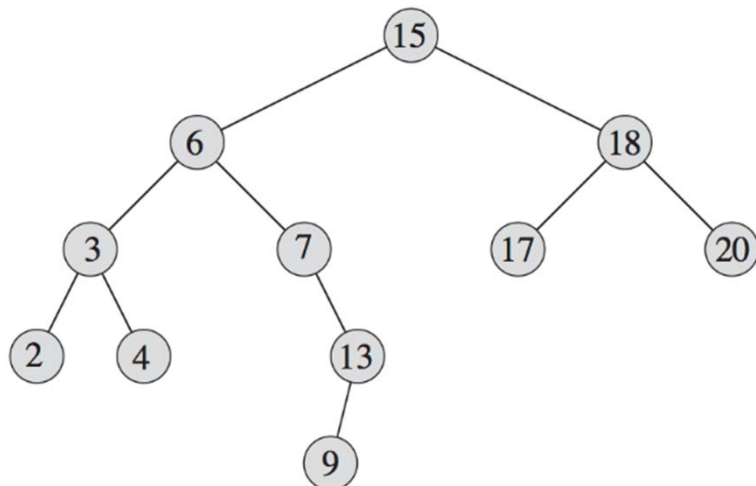
In-Order:



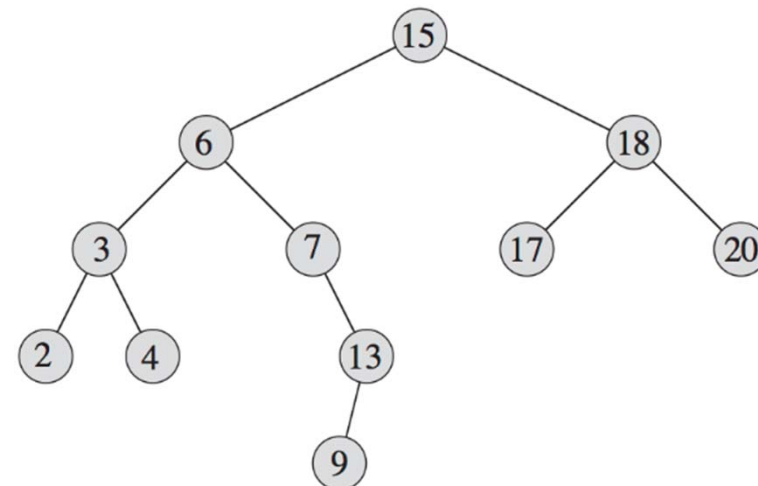
Pre-Order:



Post-Order:



Level-Order:



Traversieren eines binären Suchbaums

Tiefensuche (Depth First Search / DFS Traversal)

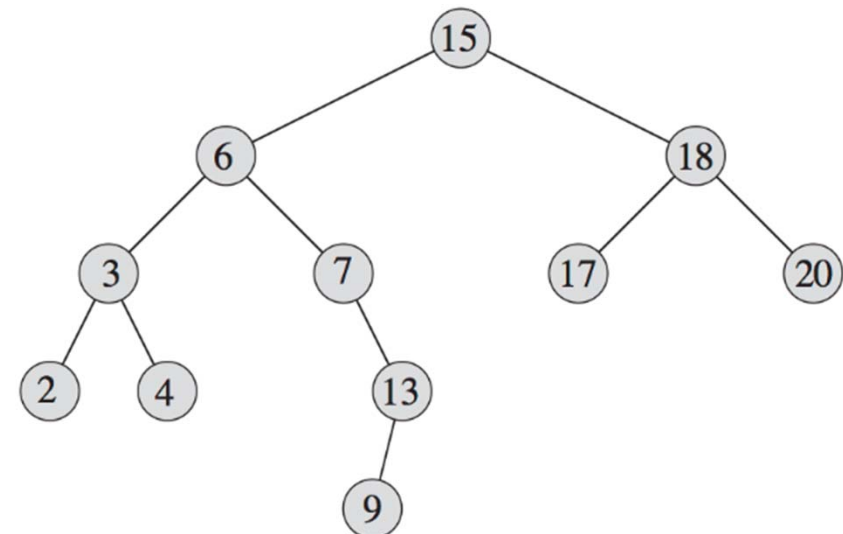
Pre-Order:

In-Order:

Post-Order:

Breitensuche (Breadth First Search / BFS Traversal)

Level-Order:



preorder(node):

```
if node != null
    visit(node)
    preorder(node.left)
    preorder(node.right)
```

inorder(node):

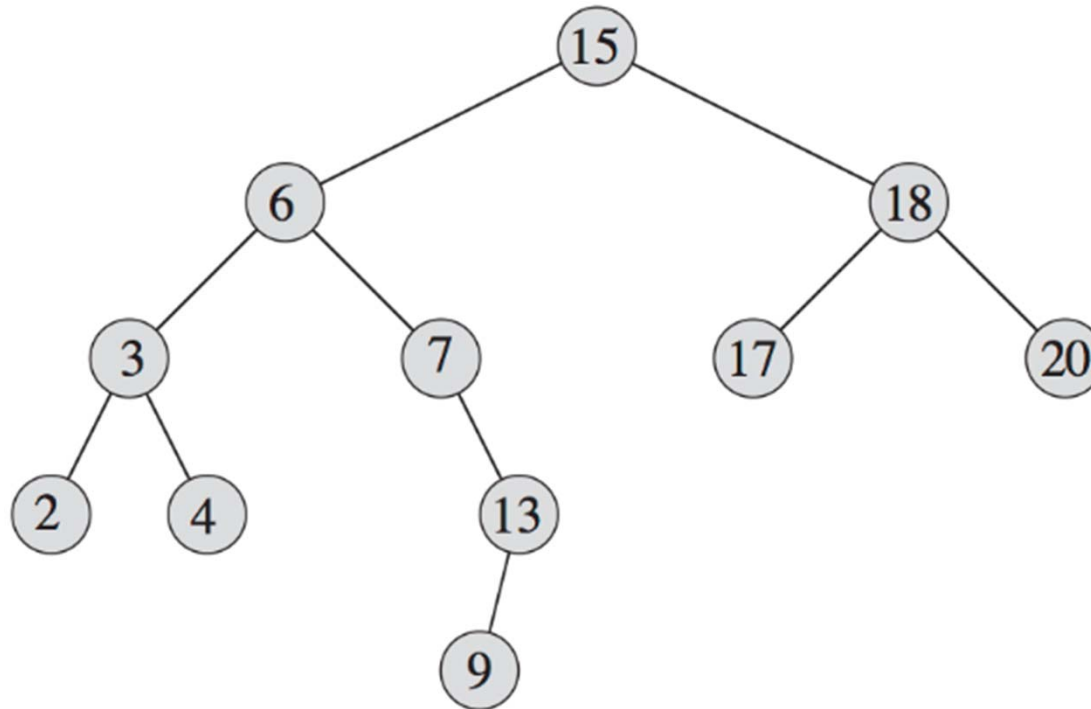
```
if node != null
    inorder(node.left)
    visit(node)
    inorder(node.right)
```

postorder(node):

```
if node != null
    postorder(node.left)
    postorder(node.right)
    visit(node)
```

Breitensuche (BFS Traversierung)

- Funktioniert nicht so einfach rekursiv wie die Tiefensuche



Breitensuche (BFS Traversierung)

- Funktioniert nicht so einfach rekursiv wie die Tiefensuche
- Lösung mit einer Warteschlange:
 - Wenn ein Knoten besucht wird, werden seine Kinder in die Queue eingereiht

BFS-Traversal:

```
Q = new Queue()
Q.enqueue(root)
while not Q.empty() do
    node = Q.dequeue()
    visit(node)
    if node.left != null
        Q.enqueue(node.left)
    if node.right != null
        Q.enqueue(node.right)
```

Tiefensuche:

- Jeder Knoten wird genau einmal besucht
- Kosten pro Knoten: $O(1)$
- **Gesamtzeit** für DFS Traversierung: $O(n)$

Breitensuche:

- Jeder Knoten wird genau einmal besucht
 - Kosten pro Knoten ist linear in der Anzahl Kinder
 - Aber: Jeder Knoten wird genau einmal in die FIFO-Queue eingefügt
- Kosten pro Knoten (amortisiert): $O(1)$
- **Gesamtzeit** für BFS Traversierung: $O(n)$

In-Order Traversierung:

- Besucht die Elemente eines binären Suchbaums in sortierter Reihenfolge
- Sortieren:
 1. Einfügen aller Elemente
 2. In-Order Traversierung
- Beobachtung: Reihenfolge hängt nur von der Menge der Elemente (Schlüssel) ab, nicht aber von der Struktur des Baums

Pre-Order Traversierung:

- Aus der Pre-Order-Reihenfolge lässt sich der Baum in eindeutiger (und effizienter) Weise rekonstruieren
- Geeignet, um den Baum z.B. in einer Datei zu speichern

Beispiel: Pre-Order 8, 5, 4, 2, 1, 3, 7, 6, 10, 9, 13, 11, 12, 14

Post-Order Traversierung:

- Löschen eines ganzen binären Suchbaums
- Zuerst muss der Speicher der Teilbäume freigegeben werden, dann kommt die Wurzel

```
delete-tree(node)
    if (node != null)
        delete-tree(node.left)
        delete-tree(node.right)
        delete node
```

Tiefe eines binären Suchbaums

Worst-Case Laufzeit der Operationen in binären Suchbäumen:

$O(\text{Tiefe des Baums})$

- Im **besten Fall** ist die Tiefe $\log_2 n$
 - Definition Tiefe: Länge des längsten Pfades von der Wurzel zu einem Blatt
- Im **schlechtesten Fall** ist die Tiefe $n - 1$

Was ist die **Tiefe in einem typischen Fall**?

- Was ist ein typischer Fall?

Ist es möglich, in einem **binären Suchbaum** immer Tiefe $O(\log n)$ zu garantieren?

“Typischer” Fall

Zufälliger binärer Suchbaum:

- n Schlüssel werden in zufälliger Reihenfolge eingefügt

Beobachtung:

- Mit Wahrscheinlichkeit $1/4$ haben beide Teilbäume der Wurzel mindestens $n/4$ Knoten.

Zufälliger binärer Suchbaum:

- n Schlüssel werden in zufälliger Reihenfolge eingefügt

Beobachtung:

- Mit Wahrscheinlichkeit $1/4$ haben beide Teilbäume der Wurzel mindestens $n/4$ Knoten.
- Analoges gilt auch für alle Teilbäume
- Im Durchschnitt wird deshalb auf jedem 2. Schritt von der Wurzel Richtung eines Blattes, der Teilbaum um einen Faktor $3/4$ kleiner!
- Verkleinern um einen Faktor $3/4$ geht nur $O(\log n)$ oft.
- Tiefe eines zufälligen binären Suchbaums ist deshalb $O(\log n)$
- Genaue Rechnung ergibt:

Tiefe eines zufälligen bin. Suchbaums: $4.311 \cdot \ln n$

“Typischen” Fall erzwingen?

“Typischer” Fall:

- Falls die Schlüssel in zufälliger Reihenfolge eingefügt werden, hat der Baum Tiefe $O(\log n)$
- Operationen haben Laufzeit $O(\log n)$

Problem:

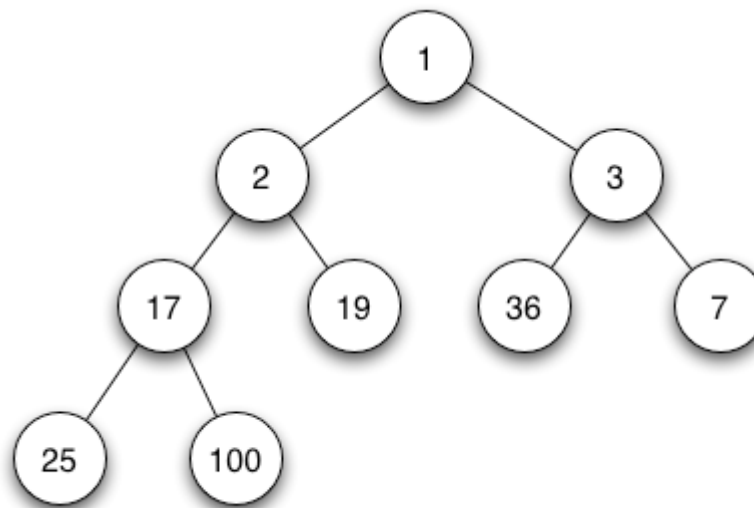
- Zufällige Reihenfolge ist nicht unbedingt der typische Fall!
- Vorsortierte Werte kann genau so typisch sein
 - Das ergibt einen sehr schlechten binären Suchbaum

Idee:

- Können wir zufällige Reihenfolge erzwingen?
- Schlüssel werden in beliebiger Reihenfolge eingefügt, aber Struktur soll immer wie bei zufälliger Reihenfolge sein!

Heap (Min-Heap) Eigenschaft:

- Gegeben ein Baum, jeder Knoten einen Schlüssel
- Ein Baum hat die Min-Heap Eigenschaft, falls **in jedem Teilbaum**, die **Wurzel** den **kleinsten Schlüssel** hat



- Heaps sind auch die “richtige” Datenstruktur, um **Prioritätswarteschlangen** zu implementieren
 - werden wir noch behandeln

Annahme:

- Jedes Element hat zwei eindeutige Schlüssel *key1* und *key2*

Ziel:

- Binärer Suchbaum bezüglich *key1*
- Einfügen in Reihenfolge, welche durch *key2* gegeben ist

Beispiel: (1,4), (2,7), (3,1), (4,10), (5,3), (6,8), (7,9), (8,2), (9,6), (10,5)

Annahme:

- Jedes Element hat zwei eindeutige Schlüssel *key1* und *key2*

Treap:

- **Binärer Suchbaum** bezüglich *key1*
- **Min-Heap** bezüglich *key2*
- Entspricht bin. Suchbaum der Schlüssel *key1*, in welchen die Schlüssel in der durch *key2* geg. Reihenfolge eingefügt wurden

Ziel:

- Zu jedem Primärschlüssel (*key1*) wird zusätzlich ein zufälliger Schlüssel $key2 \in [0,1]$ bestimmt
- **Stelle bei jedem insert / delete sicher, dass der Baum ein Treap bezüglich der Schlüssel *key1* und *key2* ist!**
- Entspricht bin. Suchbaum mit zufälliger Einfügereihenfolge