1 Advanced Data Structures

1.1 Rot-Schwarz-Bäume

- **Definition**
  - Binärer Suchbaum mit Zusatzeigenschaften
  - Zusatzeigenschaften:
    * Jeder Knoten hat die Farbe rot oder schwarz
    * Die Wurzel ist schwarz
    * Wenn ein Knoten rot ist, sind seine Kinder schwarz ("Nicht-Rot-Rot-Regel")
    * Für jeden Knoten hat jeder Pfad zu einem Blatt die selbe Anzahl an gleichen schwarzen Knoten
  - Halbblätter im RBT sind schwarz
  - Schwarzhöhe eines Knoten:
    Eindeutige Anzahl von schwarzen Knoten auf dem Weg zu einem Blatt im Teilbaum des Knoten
  - Für leeren Baum gibt Schwarzhöhe = 0 ($SH(nil) = 0$)
  - Höhe eines Rot-Schwarz-Baums
    * $h \leq 2 \cdot log_2(n+1)$ (n Knoten)
    * In jedem Unterteilbaum gleiche Anzahl schwarzer Knoten
    * Maximal zusätzlich gleiche Anzahl roter Knoten auf diesem Pfad
    * Einigermaßen ausbalanciert $\Rightarrow$ Höhe $O(log\ n)$
  - Alle folgenden Algorithmen arbeiten mithilfe eines Sentinels (zeigt auf sich selbst)

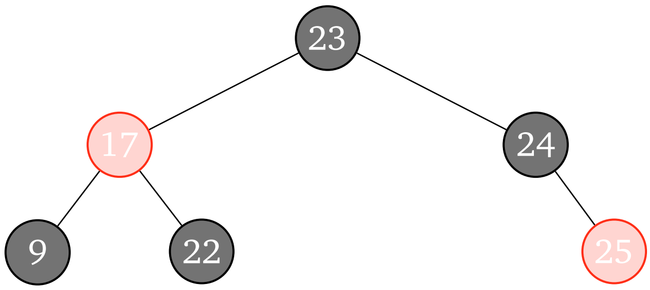


Abbildung 1: Beispielhafte Darstellung wie in 1.3

- **Einfügen**
  - Laufzeit: $\Theta(h)$ (h jedoch $log\ n$)

1. Finde Elternknoten wie im BST (BST-Einfüge Algorithmus)
2. Färbe den neuen Knoten rot
3. Wiederherstellen der Rot-Schwarz-Bedingung

fixColorsAfterInsertion(T,z)

```
WHILE z.parent.color == red DO                     // solange der Elternknoten rot ist
    IF z.parent == z.parent.parent.left THEN      // Linkes Kind (if-Fall)
        y = z.parent.parent.right;
        IF y != nil AND y.color == red THEN        // Fall 1
            z.parent.color = black;
            y.color = black;
            z.parent.parent.color = red;
            z = z.parent.parent;                   // rekursiv nach oben weiterführen
        ELSE                                       // Fall 2
            IF z == z.parent.right THEN            // Zwischenfall (2.1)
                z = z.parent;
                rotateLeft(T,z);
            z.parent.color = black;
            z.parent.parent.color = red;
            rotateRight(T, z.parent.parent);
    ELSE                                           // Rechtes Kind (else-Fall)
        // Tauschen von rechts und links
    T.root.color = black;                          // Setzen der Wurzel auf Schwarz
```

- Hilfsmethode `rotateLeft`

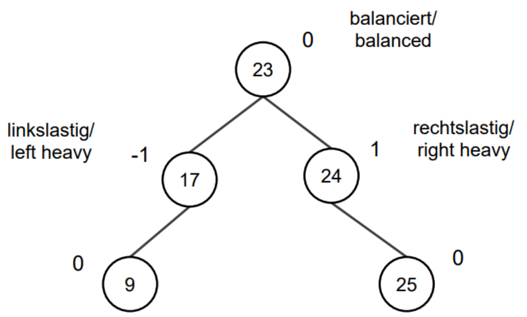
rotateLeft(T,x)

```
y = x.right;
x.right = y.left;
IF y.left != nil THEN
    y.left.parent = x;
y.parent = x.parent;
IF x.parent == T.sent THEN
    T.root = y;
ELSE
    IF x == x.parent.left THEN
        x.parent.left = y;
    ELSE
        x.parent.right = y;
y.left = x;
x.parent = y;
```

- **Löschen**
  - Laufzeit: $O(h) = O(log\ n)$
  - analog zum binären Suchbaum, aber neue Node erbt Farbe der alten Node
  - Wenn "neue"Node schwarz war $\Rightarrow$ Fixup
  - Verschiedene Fälle, die auch gegenseitig Voraussetzungen füreinander sind
  - Da das Ganze jedoch etwas umfangreicher ist, findet es sich nicht hier in der Zusammenfassung
- **Worst-Case-Laufzeiten**
  - Einfügen: $\Theta(log\ n)$
  - Löschen: $\Theta(log\ n)$
  - Suchen: $\Theta(log\ n)$

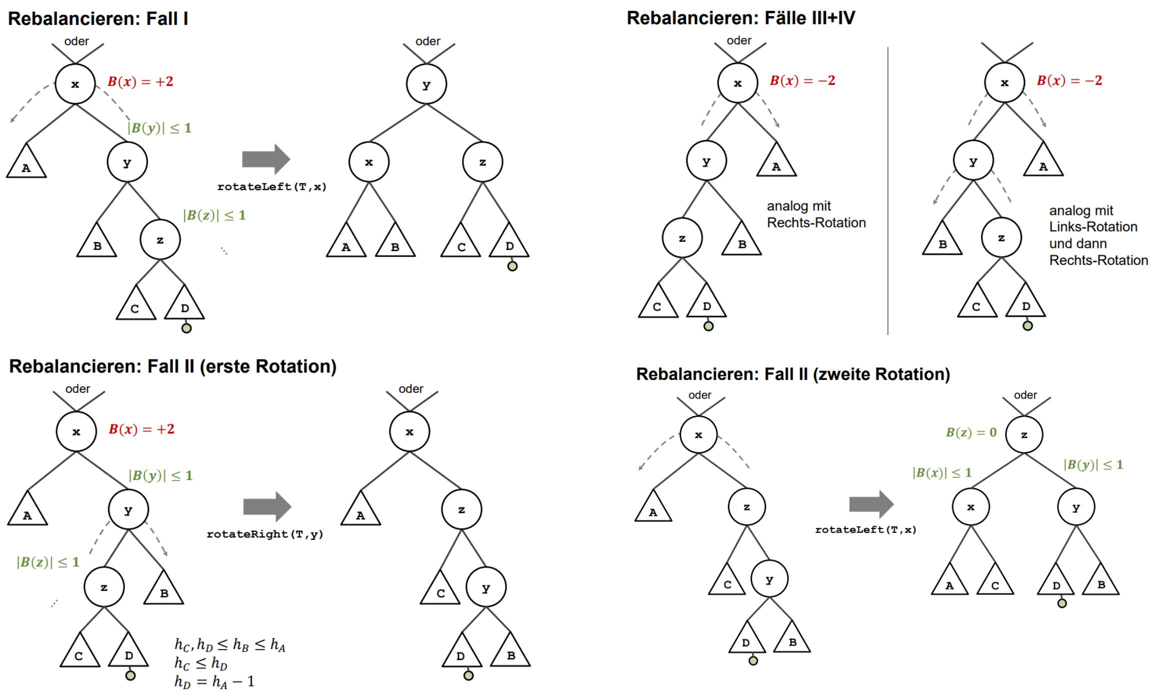
1.2 AVL-Bäume

- **Definition:**
  - $h \leq 1.441 \cdot log\ n$ (optimierte Konstanten - 1,441 vs 2 (RBT))
  - Binärer Suchbaum
  - Allerdings Balance in jedem Knoten nur $-1, 0, 1$
  - Balance für x: $B(x)$ = Höhe(rechter Teilbaum) - Höhe(linker Teilbaum)



- **AVL vs. Rot-Schwarz**
  - *AVL:*
    * Einfügen und Löschen verletzen in der Regel öfter die Baum-Bedingung
    * Aufwendiger zum Rebalancieren
  - *Rot-Schwarz:*
    * Suchen dauert evtl. länger
  - *Konklusion:*
    * AVL geeigneter, wenn mehr Such-Operationen und weniger Einfügen und Löschen
  - Gemeinsamkeiten:
  - AVL $\subset$ Rot-Schwarz
  - AVL Baum $\Rightarrow$ Rot-Schwarz-Baum mit Höhe $\lceil \frac{h+1}{2} \rceil$
  - Für jede Höhe $h \geq 3$ gibt es einen RBT, der kein AVL-Baum ist (AVL $\neq$ RBT)

- **Einfügen**
  - Einfügen funktioniert wie beim Binary Search Tree mit Sentinel
  - Erfordert danach jedoch Rebalancieren weiter oben im Baum
  - Rebalancieren: (verschiedene Fälle)



- **Löschen**
  - Analog zum binären Suchbaum
  - Rebalancieren bis eventuell in die Wurzel notwendig
- **Worst-Case-Laufzeiten**
  - Einfügen: $\Theta(log\ n)$
  - Löschen: $\Theta(log\ n)$
  - Suchen: $\Theta(log\ n)$
  - theoretisch bessere Konstanten als RBT
  - in Praxis aber nur unwesentlich schneller

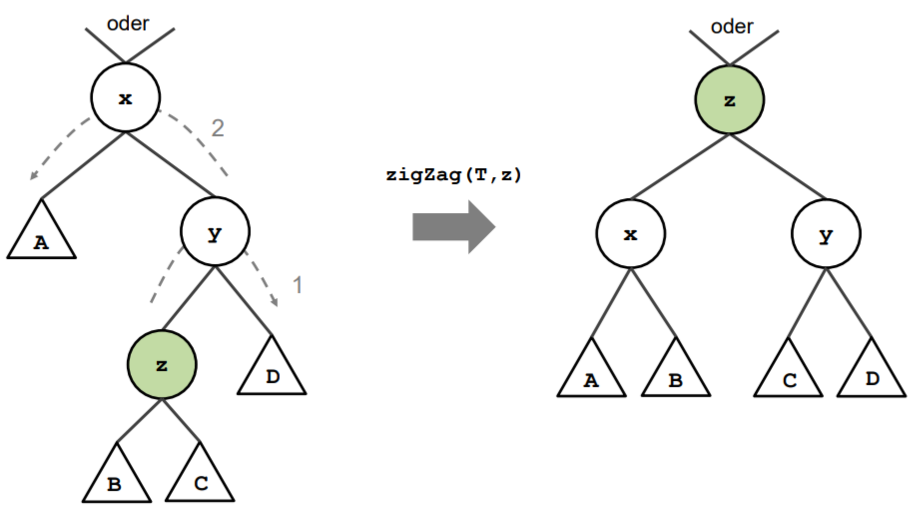
1.3 Splay-Bäume

- **Definition**
  - selbst-organisierende Listen
  - Ansatz: Einmal angefragte Werte werdeb wahrs. noch öfter angefragt
  - Angefragte Werte nach oben schieben
  - Splay-Bäume sind Untermenge von BST
- **Splay-Operationen**
  - Suchen oder Einfügen: Spüle gesuchten oder neu eingefügten Knoten an die Wurzel
  - Splay: (Folge von Zig-,Zig-Zig-, Zig-Zag-Operationen)

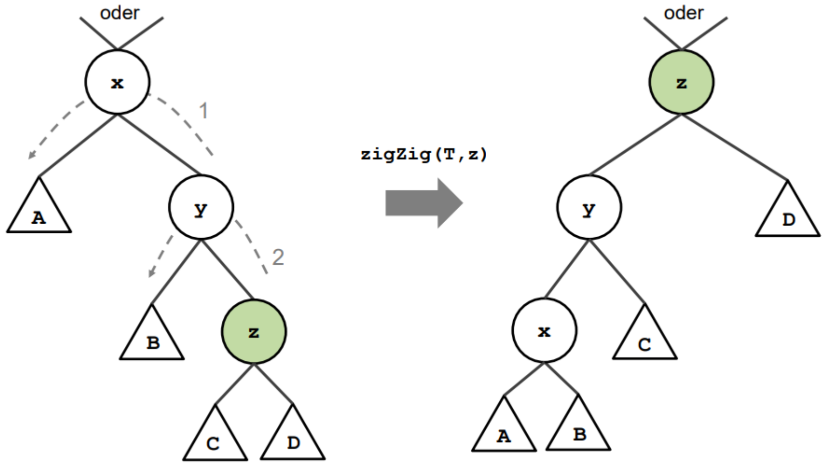
splay(T,z)

```
WHILE z != T.root DO
    IF z.parent.parent == nil THEN
        zig(T,z);
    ELSE
        IF z == z.parent.parent.left.left OR
           z == z.parent.parent.right.right THEN
           zigZig(T,z);
        ELSE
            zigZag(T,z);
```

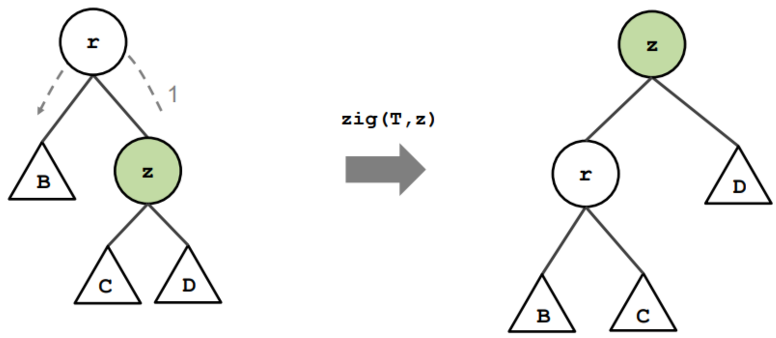
Zig-Zag-Operation =Rechts-Links- oder Links-Rechts-Rotation



Zig-Zig-Operation =Links-Links- oder Rechts-Rechts-Rotation



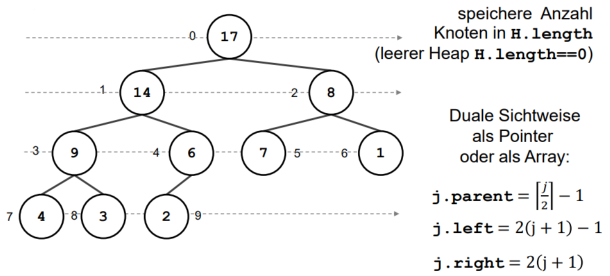
Zig-Operation =einfache Links- oder Rechts-Rotation



- **Suchen**
  - Laufzeit: $O(h)$
  - Suche des Knotens wie im BST
  - Hochspülen des gefundenen Knotens (alternativ zuletzt besuchter Knoten, falls nicht gefunden)
- **Einfügen**
  - Laufzeit: $O(h)$
  - Suche der Position wie im BST
  - Einfügen und danach hochspülen des eingefügten Knotens
- **Löschen**
  - Laufzeit: $O(h)$
  1. Spüle gesuchten Knoten per Splay-Operation nach oben
  2. Lösche den gesuchten Knoten (Wenn einer der beiden entstehenden Teilbäume leer, dann fertig)
  3. Spüle den größten Knoten im linken Teilbaum nach oben (kann kein rechtes Kind haben)
  4. Hänge rechten Teilbaum an größten Knoten aus 3. an
- **Laufzeit Splay-Bäume**
  - Amortisierte Laufzeit: Laufzeit pro Operation über mehrere Operationen hinweg
  - Worst-Case-Laufzeit pro Operation: $O(log_n\ n)$

1.4 Binäre Max-Heaps

- **Definition**
  - Heaps sind keine BSTs
  - Eigenschaften binäre Max-Heaps:
    * bis auf das unterste Level vollständig und dort von links gefüllt ist
    * Für alle Knoten gilt: `x.parent.key` $\geq$ `x.key`
    * Maximum des Heaps steht damit in der Wurzel
  - $h \leq log\ n$, da Baum fast vollständig
- **Heaps durch Arrays**



0	1	2	3	4	5	6	7	8	9
17	14	8	9	6	7	1	4	3	2

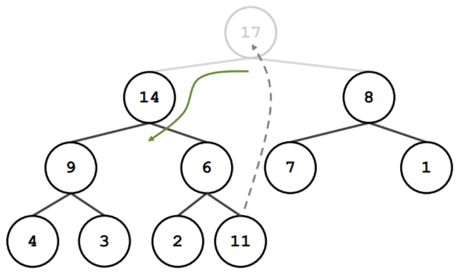
- **Einfügen**
  - Idee: Einfügen und danach Vertauschen nach oben, bis Max-Eigenschaft wieder erfüllt ist
  - Laufzeit: $O(h) = O(log\ n)$

insert(H,k) // als unbeschränktes Array

```
H.length = H.length + 1;
H.A[H.length-1] = k;

i = H.length - 1;
WHILE i > 0 AND H.A[i] > H.A[i.parent]
    SWAP(H.A, i, i.parent);
    i = i.parent;
```

- **Lösche Maximum**
  1. Ersetze Maximum durch "letztes"Blatt
  2. Vertausche Knoten durch Maximum der beiden Kinder (`heapify`)



extract-max(H)

```
IF isEmpty(H) THEN return error "underflow";
ELSE
    max = H.A[0];
    H.A[0] = H.A[H.length - 1];
    H.length = H.length - 1;
    heapify(H, 0);
    return max;
```

heapify(H, i)

```
maxind = i;
IF i.left < H.length AND H.A[i]<H.A[i.left] THEN
    maxind = i.left;
IF i.right < H.length AND H.A[maxind]<H.A[i.right] THEN
    maxind = i.right;

IF maxind != i THEN
    SWAP(H.A, i, maxind);
    heapify(H, maxind);
```

- **Heap-Konstruktion aus Array**
  - Blätter sind für sich triviale Max-Heaps
  - Bauen von Max-Heaps für Teilbäume mithilfe Rekursion per `heapify`
  - (Array nicht unbedingt in richtiger Reihenfolge)

buildHeap(H.A) // Array in H.A

```
H.length = A.length;
FOR i = ceil((H.length-1)/2) - 1 DOWNTO 0 DO
    heapify(H.A,i);
```

- **Heap-Sort**
  - Idee: Bauen des Heaps aus Array und dann Extraktion des Maximums

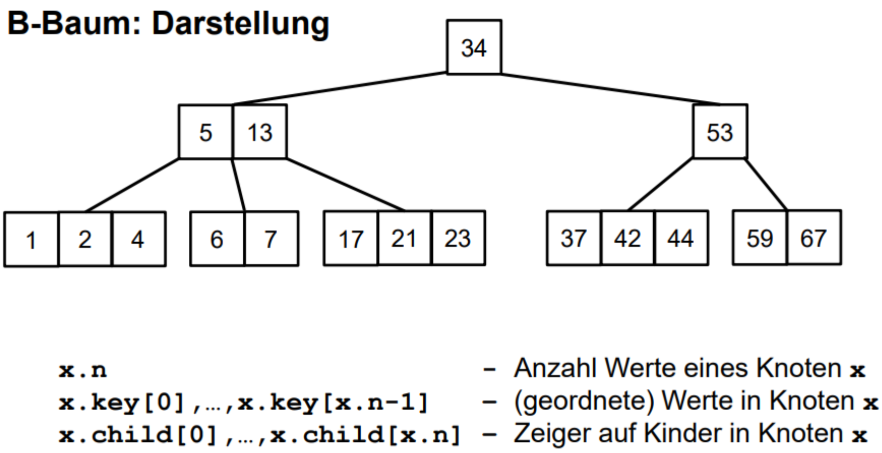
heapSort(H.A)

```
buildHeap(H.A)                // Bauen des Heaps
WHILE !isEmpty(H) DO
    PRINT extract-max(H);     // Ausgabe des Maximums bis Heap leer ist
```

1.5 B-Bäume

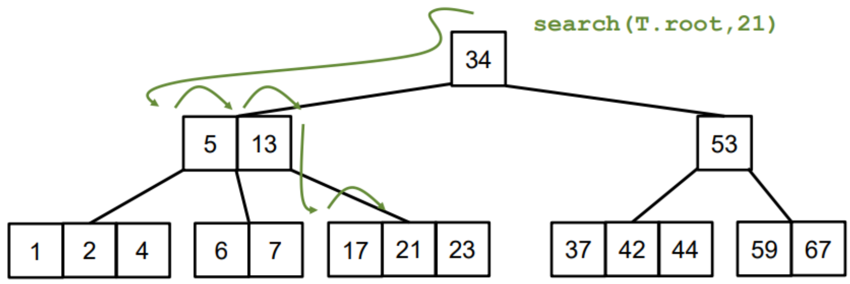
- **Definition**
  - Jeder B-Baum hat einen angebenen Grad also z.B. $t = 2$
  - Eigenschaften:
    * Wurzel zwischen $[1, ..., 2t - 1]$ Werte
    * Knoten zwischen $[t - 1, ...2t - 1]$ Werte
    * Werte innerhalb eines Knotens aufsteigend geordnet
    * Blätter haben alle die gleiche Höhe
    * Jeder innere Knoten mit n Werten hat $n + 1$ Kinder, sodass gilt:

      $k_0 \leq key[0] \leq k_1 \leq key[1] \leq ... \leq k_{n-1} \leq key[n - 1] \leq k_n$



  - Höhe B-Baum: $h \leq log_t \frac{n+1}{2}$ (Grad t und n Werte)
  - B-Baum wird für größere t flacher

- **Suche**



search(x, k)

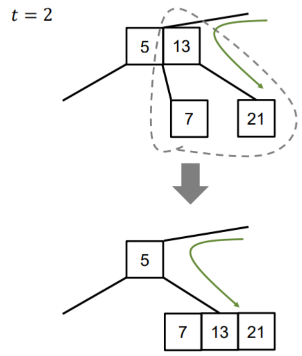
```
WHILE x != nil DO
    i = 0;
    WHILE i < x.n AND x.key[i] < k DO
        i++;
    IF i < x.n AND x.key[i] == k THEN
        return(x, i);
    ELSE
        x = x.child[i];
return nil;
```

- **Einfügen**
  - Einfügen erfolgt immer in einem Blatt
  - Falls das Blatt voll ist, muss jedoch gesplittet werden
  - ⇒ Beim Durchlaufen des Baumes an jeder notwendigen (voll) Position splitten
  - Splitten:
    * Bricht volle Node auf und fügt mittleren Wert zur Elternnode hinzu
    * Aus den anderen Werten entstehen nun jeweils eigene Kinder
    * An der Wurzel splitten erzeugt neue Wurzel und erhöht Baumhöhe um eins
  - Ablauf zusammengefasst:
    (a) Start bei Wurzel, falls kein Platz mehr splitten
    (b) Durchlaufen des Baumes bis zur richtigen Position und immer, falls voll, splitten
    (c) Einfügen der Node (fertig)

insert(T, z)

```
Wenn Wurzel schon 2t-1 Werte, dann splitte Wurzel
Suche rekursiv Einfügeposition:
    Wenn zu besuchendes Kind 2t-1 Werte, splitte es erst
Füge z in Blatt ein
```

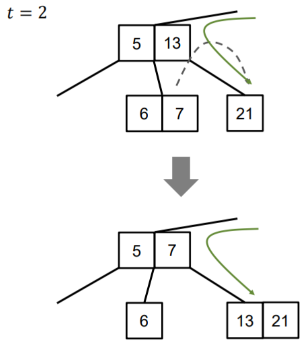
- **Löschen**
  - Wenn Blatt noch mehr als t-1 Werte, kann der Wert einfach entfernt werden
  - Allerdings durchlaufen wir hier den Baum auch wieder von oben und stellen gewisse Voraussetzungen her
  - Durchlaufen des Baumes von oben und Anwendung der folgenden Algorithmen



Allgemeines Verschmelzen:

* Kind und alle rechten/linken Geschwisterknoten nur $t-1$ Werte
* Wenn Elternknoten vorher min. t Werte

  $\Rightarrow$ keine Änderung oberhalb notwendig



Allgemeines Rotieren/Verschieben:

* Kind nur $t-1$ Werte
* Geschwister jedoch mehr als $t-1$ Werte
* keine Änderung oberhalb notwendig

  - Code:

delete(T, k)

```
Wenn Wurzel nur 1 Wert und beide Kinder t-1 Werte,
verschmelze Wurzel und Kinder (reduziert Höhe um 1)
Suche rekursiv Löschposition:
    Wenn zu besuchendes Kind nur t-1 Werte,
    verschmelze es oder rotiere/verschiebe
Entferne Wert k im inneren Knoten/Blatt
// Ohne Probleme, aufgrund vorheriger Anpassung
```

- **Laufzeiten**
  - Einfügen: $\Theta(log_t\ n)$
  - Löschen: $\Theta(log_t\ n)$
  - Suchen: $\Theta(log_t\ n)$
  - Nur vorteilhaft wenn Daten blockweise eingelesen werden
  - $\mathcal{O}$-Notation versteckt hier konstanten Faktor t für Suche innerhalb eines Knotens