IIF
INSTITUT FÜR
INFORMATIK
FREIBURG

Vorlesung Informatik 2
Algorithmen und Datenstrukturen

(18 – Bäume: Grundlagen und natürliche Suchbäume)

Prof. Dr. Susanne Albers

Bäume (1)

Bäume sind

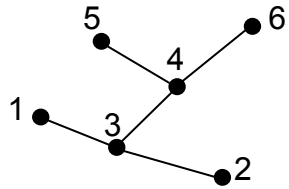
- verallgemeinerte Listen
  (jedes Knoten-Element kann mehr als einen Nachfolger haben)
- spezielle Graphen:
  – Ein allgemeiner Graph $G = (V,E)$ besteht aus Knoten V (vertices) und Kanten $E \subseteq V \times V$ (edges).
  – Die Kanten sind entweder gerichtet oder ungerichtet.
  – Knoten und Kanten können markiert sein (sie tragen weitere Informationen).

  Ein Baum ist ein zusammenhängender azyklischer Graph. Es gilt:
  # Knoten= # Kanten+1
- ein allgemeines & zentrales Konzept zur hierarchischen Strukturierung von Informationen:
  – Entscheidungsbäume
  – Codebäume
  – Syntaxbäume

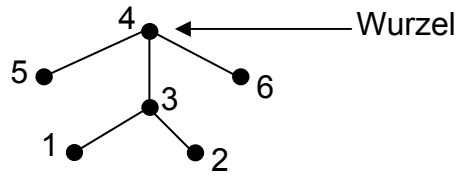
Bäume (2)

Unterscheide verschiedene Arten von Bäumen

- Ungerichteter Baum



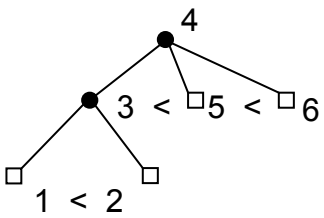
- Gewurzelter Baum (Wurzelbaum, ein Knoten ist als Wurzel ausgezeichnet)



  – Von jedem Knoten *k* führt genau ein Pfad (Folge paarweise benachbarter Kanten) zur Wurzel
  – Vater (Elter, direkter Vorgänger) eines Knotens *k* ist der erste Nachbar auf dem Pfad von *k* zur Wurzel
  – Söhne (Kinder, direkte Nachfolger) sind die anderen Nachbarn von *k*
  – Der Rang von *k* ist die # Söhne des Knotens *k*

Bäume (3)

- Gewurzelter Baum:
  - Wurzel: einziger Knoten ohne Vater
  - Blätter: Knoten ohne Söhne
  - innere Knoten: alle Knoten, die kein Blatt sind
  - Ordnung eines Baumes *T*: maximaler Rang eines Knotens von *T*
  - Der Begriff Baum wird oft als Synonym für Wurzelbaum gebraucht
- Geordneter Wurzelbaum: Unter den Söhnen jedes Knotens ist eine Ordnung festgelegt, etwa über < -Relation unter Schlüsseln in Knoten



- Binärbaum: Geordneter Wurzelbaum der Ordnung 2, Söhne werden mit linker Sohn, rechter Sohn bezeichnet
- Vielwegbaum: geordneter Wurzelbaum der Ordnung > 2

Bäume (4)

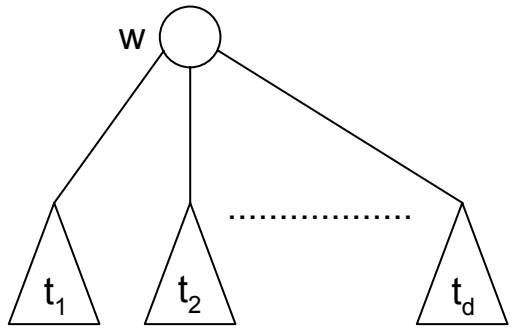
Exaktere Definition für die Menge M_d

der geordneten Wurzel-Bäume der Ordnung d (d ≥ 1):

- Ein einzelner Knoten ist in M_d
- Sind $t_1, \ldots, t_d$ in M_d und ist w ein Knoten, dann ist *w* mit den Wurzeln von $t_1, \ldots, t_d$ als Nachfolgern (v.l.n.r.) ein Baum *t*. Die t_i sind Teilbäume von *t*.
  - Nach dieser Definition muss jeder Knoten Rang *d* haben.
  - Allgemeiner darf der Rang auch kleiner sein.
  - Knoten von Binär-Bäumen haben entweder 0 oder 2 Söhne.
  - Man könnte auch Knoten mit genau 1 Sohn zulassen, indem in obiger Definition auch leere Teil-Bäume erlaubt werden.

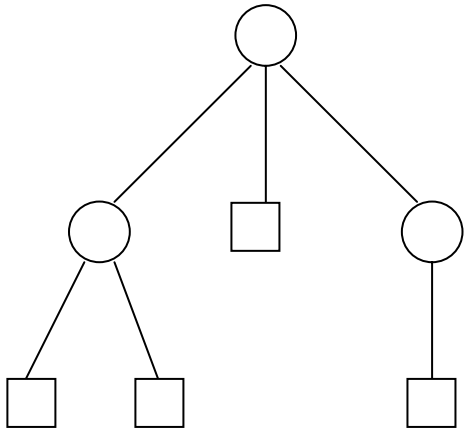
Rekursive Definition

1. $\square$ ist ein Baum der Ordnung d, mit Höhe 0.
2. Sind $t_1,\ldots,t_d$ beliebige, disjunkte Bäume der Ordnung d, so erhält man einen (weiteren) Baum der Ordnung d, indem man die Wurzeln von $t_1,\ldots,t_d$ zu Nachfolgern einer neu geschaffenen Wurzel w macht. Die Höhe h des neuen Baums ist dann max $\{h(t_1),\ldots,h(t_d)\}+1$.

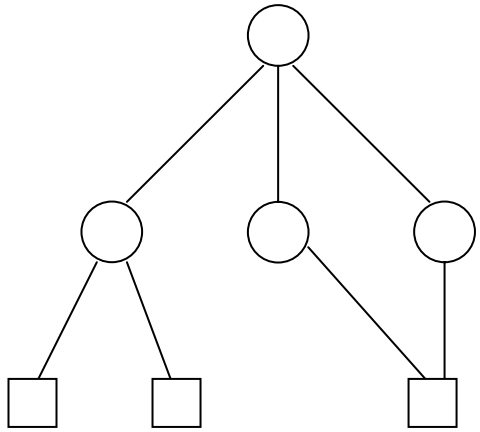


Festlegung: $d = 2$ Binärbäume, $d > 2$ Vielwegbäume.

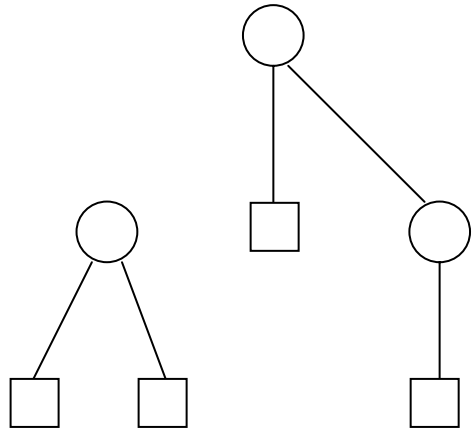
Beispiele



Baum



kein Baum



kein Baum
(aber zwei Bäume)

Strukturelle Eigenschaften von Bäumen

- Tiefe eines Knotens *k*: # Kanten von der Wurzel des Baums bis *k*
  (Abstand von k zur Wurzel)
- Höhe *h*(*t*) eines Baumes *t*: Maximale Tiefe eines Blattes von *t*.
  Alternative (rekursive) Definition:
  – *h*(Blatt) = 0
  – $h(t) = 1 + \max\{ h(t_i) \mid \text{Wurzel von } t_i \text{ ist Sohn von Wurzel von } t\}$
- Niveau *i*: alle Knoten in Tiefe *i*
- Vollständiger Baum: Baum, bei dem jedes nichtleere Niveau volle Knotenanzahl hat.
  → alle Blätter haben die gleiche Tiefe.

Anwendung für Bäume

Verwendung von Bäumen für die Lösung des WBP:

- Knoten: Speicher für Datensätze
- Baum: Speicher für Datenmengen
- Besonderer Vorteil (gegenüber Hash-Tabellen): Durchlauf der gesamten Menge von Sätzen (etwa in sortierter Reihenfolge) leicht möglich.

Natürliche, binäre Suchbäume (1)

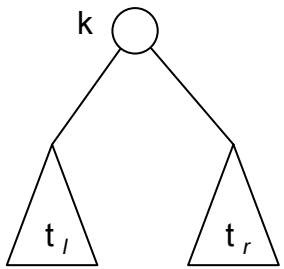
Ziel: Speichern, Wiederfinden von Datensätzen (allgemeiner: WBP)

Zwei alternative Speicherungsformen:

- Suchbäume: Schlüssel werden in inneren Knoten gespeichert
  Blätter sind leer (oft=null) sie repräsentieren Intervalle zwischen den Schlüsseln
- Blattsuchbäume: Schlüssel werden in Blättern gespeichert
  innere Knoten enthalten Hinweise zum Auffinden

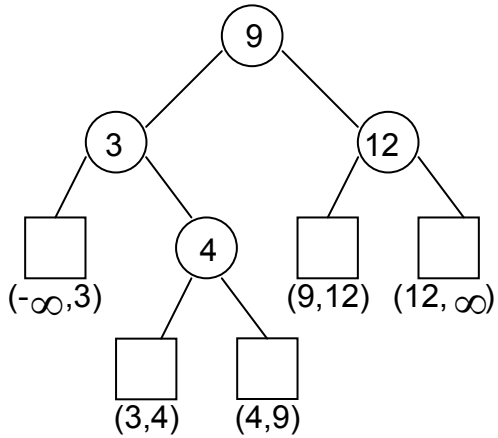
Suchbaum-Bedingung:

für jeden inneren Knoten *k* gilt: alle Schlüssel links unterhalb von *k* (im linken Teil-Baum t_l) sind < als der Schlüssel in *k* und alle Schlüssel rechts unterhalb von *k* (im rechtenTeil-Baum t_r) sind > als der Schlüssel in *k*



Natürliche Bäume (2)

Blätter in Suchbaum repräsentieren Intervalle zwischen den Schlüsseln der i. Knoten

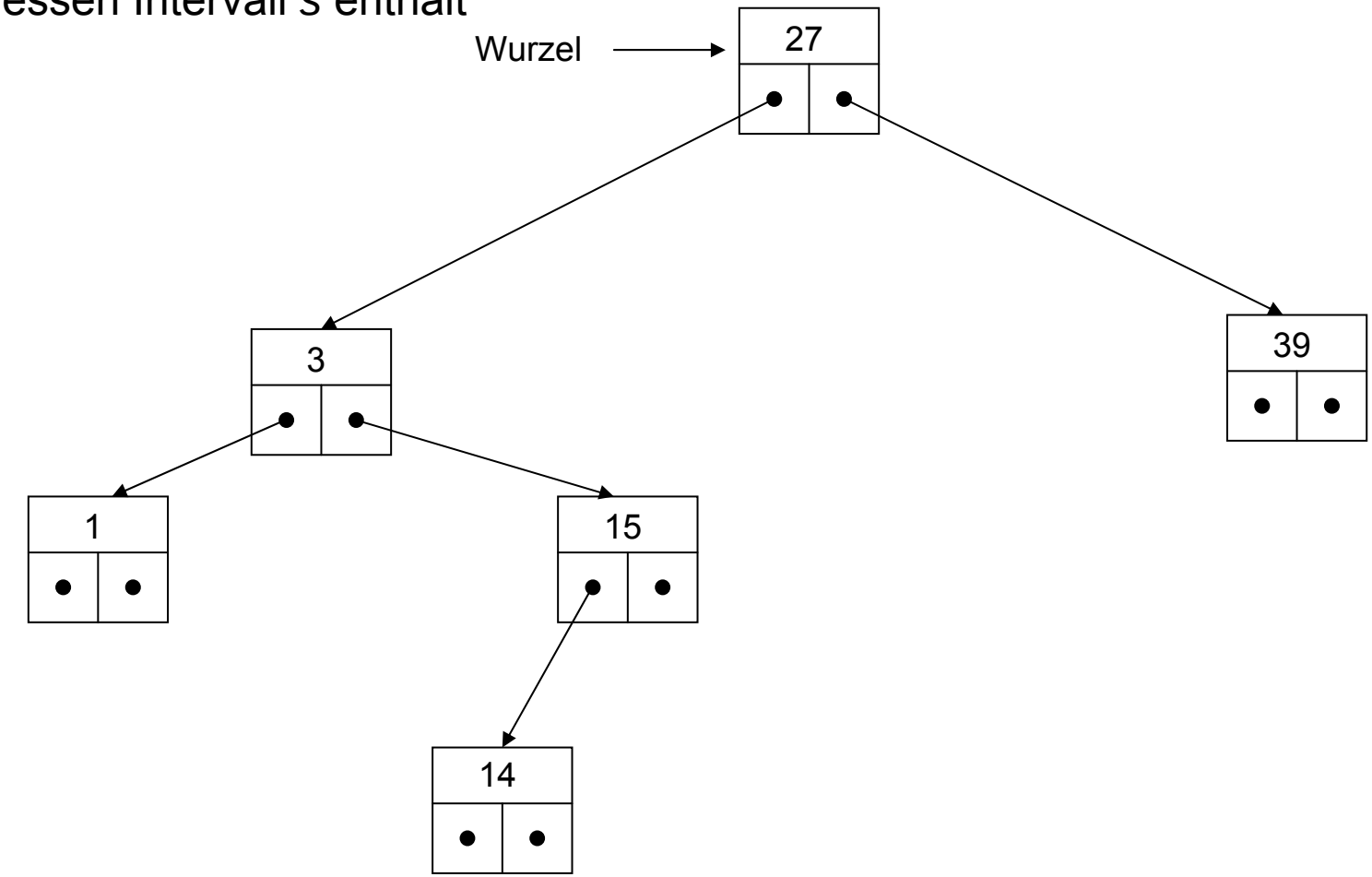


Wie sucht man in einem Suchbaum nach Schlüssel *s*? (Blatt ≅null)

```
k = wurzel;
while (k != null) {
    if (s == k.key) return true;
    if (s < k.key)  k = k.left;
    else            k = k.right
}
return false;
```

Beispiel (ohne Stopper)

Suche nach Schlüssel *s* endet beim Suchbaum in Knoten *k* mit *k.key* == *s* oder in Blatt, dessen Intervall *s* enthält



Natürliche Bäume (3)

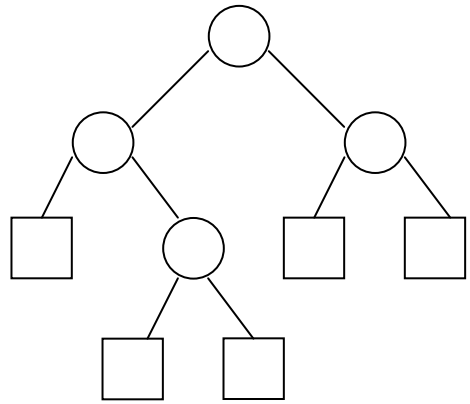
Blattsuchbaum:

- Schlüssel werden in Blättern gespeichert
- Hinweise (Wegweiser) werden in inneren Knoten gespeichert, so dass $s_l \leq s_k \leq s_r$ (s_l : Schlüssel in linkem TB, s_k : Wegweiser in k, s_r Schlüssel in rechtem TB)

- Wahl für s: entweder maximaler Schlüssel von t_l (üblich) oder minimaler Schlüssel von t_r.

Beispiel: Blattsuchbaum

Blätter enthalten die Schlüssel, innere Knoten Wegweiser.



Natürliche Bäume (4)

Wie sucht man in einem Blattsuchbaum nach Schlüssel *s*? (Blatt $\cong$ Knoten mit 2 null-Ref.)

```
k = wurzel;
if (k == null) return false;
while (k.left != null) { // damit auch k.right != null
    if (s <= k.key) k = k.left;
    else k = k.right;
}                                  // jetzt in Blatt
return s==k.key;
```

Im Folgenden werden Suchbäume (nicht Blatt-) betrachtet.

Natürliche Bäume (5)

```
class SearchNode {
    int content;
    SearchNode left;
    SearchNode right;
    SearchNode (int c){    // Konstruktor fuer einen Knoten
        content = c;       // ohne Nachfolger
        left = right = null;
    }
} //class SearchNode

class SearchTree {
    SearchNode root;
    SearchTree () { // Konstruktor fuer leeren Baum
        root = null;
    }
    // ...
```

Natürliche Bäume (6)

```
/* Suche nach c im Baum */
boolean search (int c) {
    return search (root, c);
}
boolean search (SearchNode n, int c){
    while (n != null) {
        if (c == n.content) return true;
        if (c < n.content) n = n.left;
        else n = n.right;
    }
    return false;
}
```

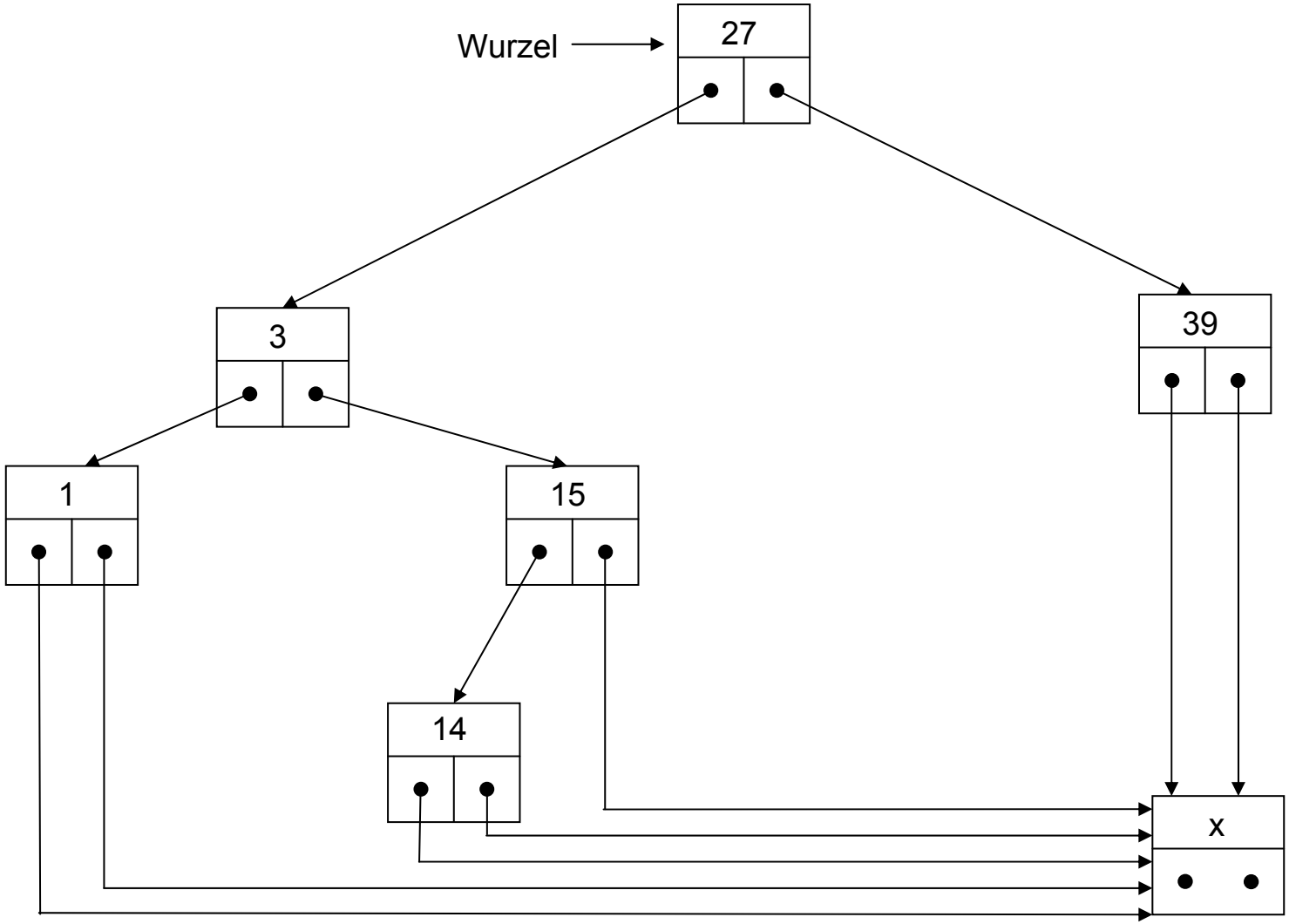
Natürliche Bäume (7)

Alternative Baumstruktur:

- Statt Blatt ≅ null verwende: Blatt ≅ Referenz auf besonderen Stopper-Knoten *b*
- Bei Suche speichere Suchschlüssel *s* in *b* und spare Vergleiche in inneren Knoten.

Verwendung eines Stoppers für die Suche!

Beispiel (mit Stopper)



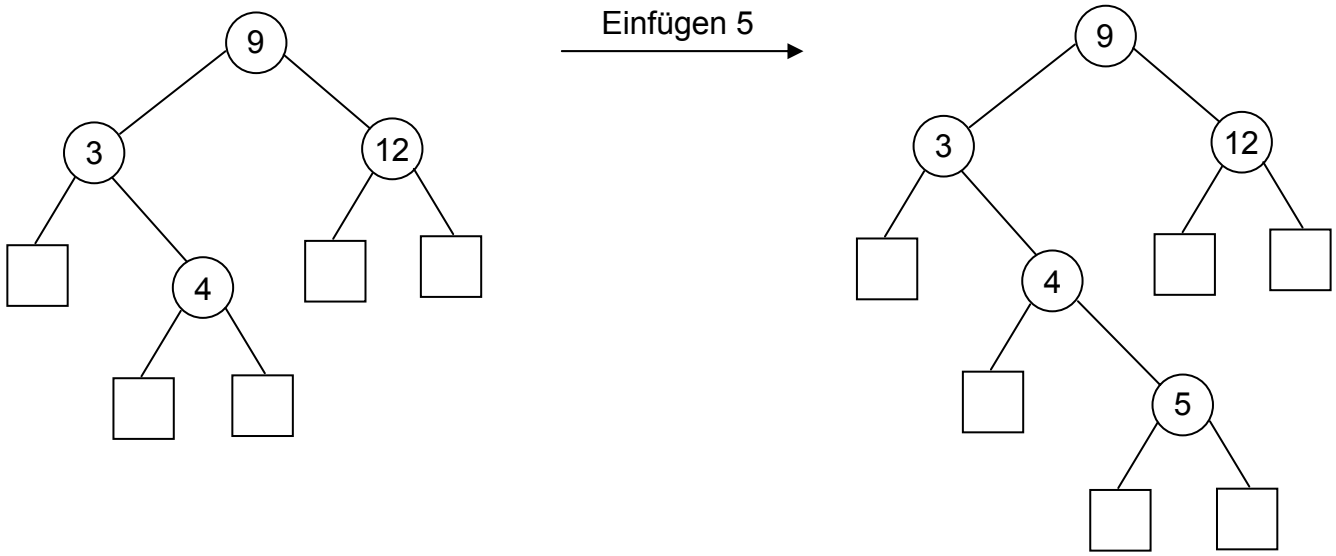
Natürliche Bäume (7)

Einfügen eines Knotens mit Schlüssel *s* in Suchbaum *t*:

1. Suche nach *s* endet in Knoten mit *s*: nicht einfügen, da sonst doppelte Schlüssel
2. Suche endet in Blatt *b*: Mache *b* zu innerem Knoten mit *s* als Schlüssel und zwei neuen Blättern.

→ Baum bleibt Suchbaum!

Natürliche Bäume (8)



- Baum-Struktur hängt von Einfügereihenfolge in anfangs leeren Baum ab
- Höhe kann linear zunehmen, sie kann aber auch in O(log *n*) sein, genau $\lceil \log_2 (n+1) \rceil$.

Natürliche Bäume (9)

```
int height (){
    return height (root);
}
int height (SearchNode n){
    if (n == null) return 0;
    else return 1 + Math.max (height (n.left), height (n.right));
}
/* Fuege c im Baum ein; gib true zurueck, falls erfolgreich
   und false, falls schon vorhanden */
boolean insert (int c) { // Fuege c ein;
    if (root == null){
        root = new SearchNode (c);
        return true;
    } else return insert (root, c);
}
```

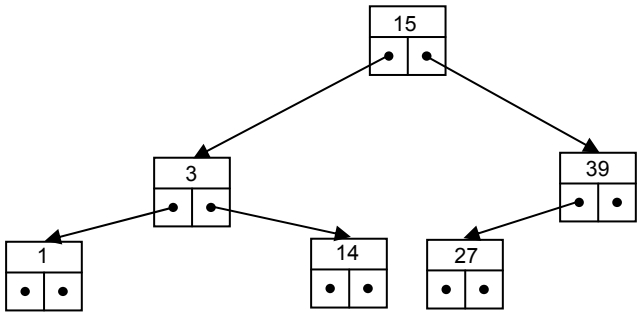
Natürliche Bäume (10)

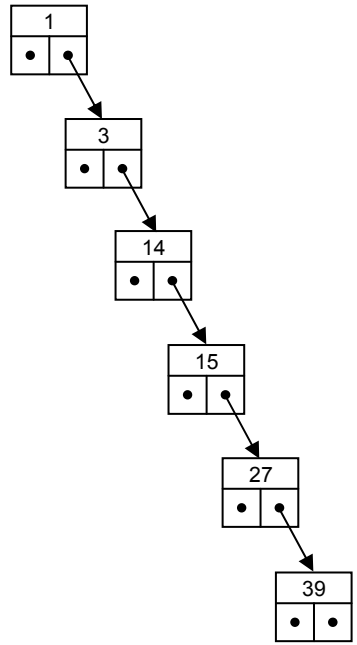
```
boolean insert (SearchNode n, int c){
    while (true){
        if (c == n.content) return false;
        if (c < n.content){
            if (n.left == null) {
                n.left = new SearchNode (c);
                return true;
            } else n = n.left;
        } else { // c > n.content
            if (n.right == null) {
                n.right = new SearchNode (c);
                return true;
            } else n = n.right;
        }
    }
}
```

Sonderfälle

Die Struktur des resultierenden Baums hängt stark von der Reihenfolge ab, in der die Schlüssel eingefügt werden. Die minimale Höhe ist $\lceil \log_2 n + 1 \rceil$ und die maximale Höhe ist n.

Resultierende Suchbäume für die Reihenfolgen 15, 39, 3, 27, 1, 14 und 1, 3, 14, 15, 27, 39:





Natürliche Bäume (11)

Ein Natürlicher Baum entsteht durch iteriertes Einfügen in den anfangs leeren Baum.

- Welche Bäume sind die häufigeren/typischeren, die ausgeglichenen, die degenerierten?
- Wie teuer ist das Einfügen?

Natürliche Bäume (11)

Entfernen eines Knotens mit Schlüssel *s* aus einem Baum (Beibehaltung der Suchbaum Eigenschaft)

1. Suche nach *s*, falls nicht da: fertig; sonst endet die Suche mit *k.key* == *s* und
2. *k* hat keinen, einen oder zwei Söhne:
   (a) kein Sohn: fertig, Vater bekommt Zeiger auf null
   (b) nur ein Sohn ist da : lasse Vater *v* von *k* darauf statt auf *k* zeigen
   (c) zwei Söhne: suche kleinsten Wert in rechtem Teilbaum, d.h. mache einen Schritt nach rechts und beliebig viele nach links bis zu *p* (symmetrischer Nachfolger von *k*); kopiere *p.key* nach *k*, lösche *p* (max. einen Sohn, also nach *a*, *b* behandeln)

Symmetrische Nachfolger

Definition: Ein Knoten *q* heißt der symmetrische Nachfolger eines Knotens *p*, wenn *q* den kleinsten Schlüssel enthält, der größer als der Schlüssel von *p* ist.

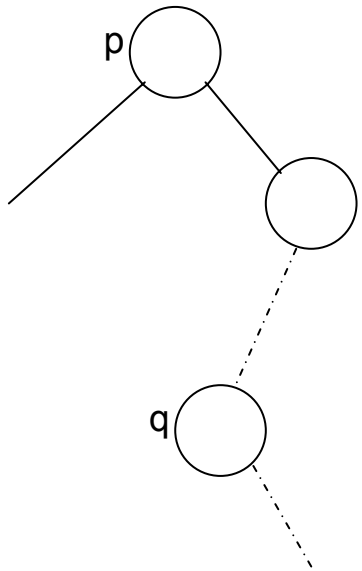
Beobachtungen:

1. Der symmetrische Nachfolger *q* von *p* ist der am weitesten links stehende Knoten im rechten Teilbaum von *p*.
2. Der symmetrische Nachfolger hat höchstens einen Nachfolger, welcher der rechte ist.

Auffinden des symmetrischen Nachfolgers

Beobachtung: Wenn *p* einen rechten Nachfolger hat, gibt es immer einen symmetrischen Nachfolger.

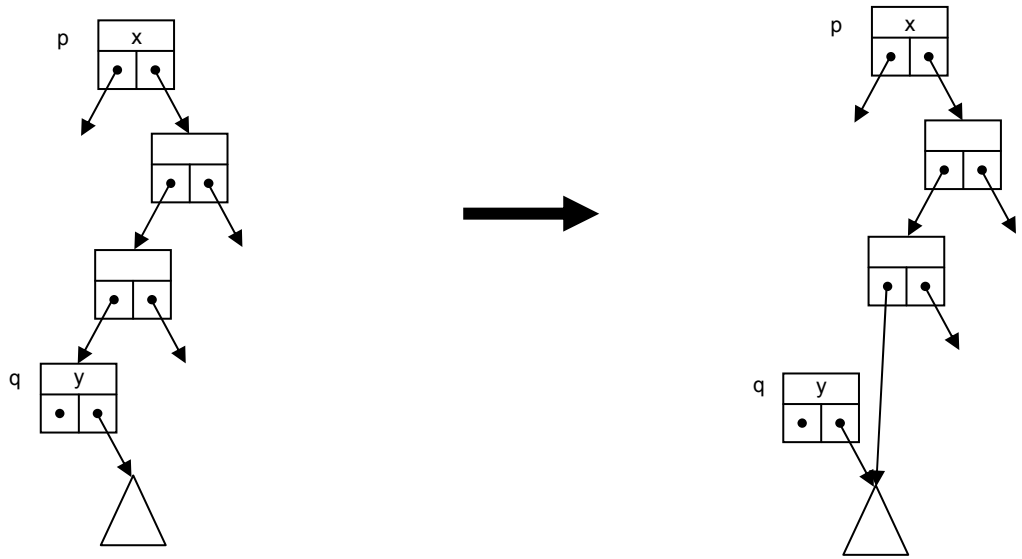
- Zunächst gehen wir zum rechten Nachfolger von *p*.
- Von dort aus gehen wir solange jeweils zum linken Nachfolger, bis wir einen Knoten ohne linken Nachfolger finden.



Idee der Löschoperation

Wir löschen *p*, indem wir den Inhalt von *p* durch den seines symmetrischen Nachfolgers *q* ersetzen. Danach löschen wir *q*.

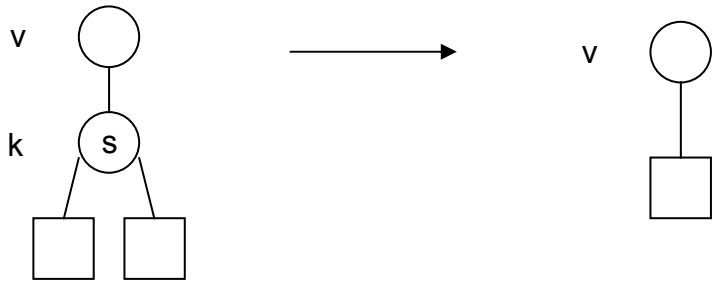
Löschen von *q* ist einfach, weil *q* höchstens einen Nachfolger hat.



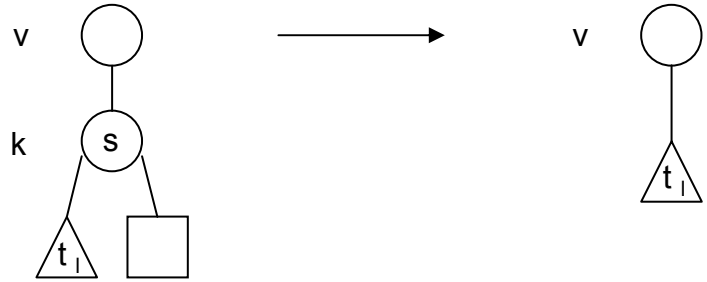
Beispiele

k hat keinen, einen oder zwei innere Söhne:

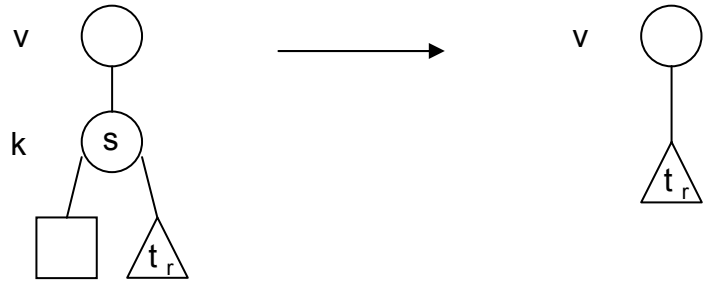
a)



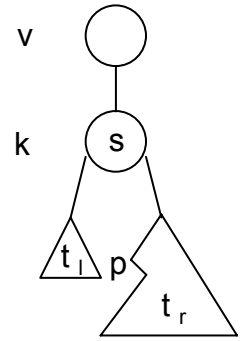
b)



c)



d)



Natürliche Bäume (12)

```
boolean delete (int c){
    return delete (null, root, c);
}
// loesche c im Baum mit Wurzel n, deren Vorgaenger vn ist
boolean delete (SearchNode vn, SearchNode n, int c){
    if (n == null) return false;
    if (c < n.content) return delete (n, n.left, c);
    if (c > n.content) return delete (n, n.right, c);
    // jetzt gilt: c == n.content
    if (n.left == null) {
        point (vn, n, n.right);
        return true;
    }
    if (n.right == null) {
        point (vn, n, n.left);
        return true;
    }
    // ...
```

Natürliche Bäume (13)

```
    // jetzt ist n.left != null und n.right != null
    SearchNode q = vSymNach (n);
    if (n == q) { // rechter Sohn von q ist vSymNach (n)
        n.content = q.right.content;
        q.right = q.right.right;
        return true;
    } else { // linker Sohn von q ist vSymNach (n)
        n.content = q.left.content;
        q.left = q.left.right;
        return true;
    }
} // boolean delete (SearchNode vn, SearchNode n, int c)
```

Natürliche Bäume (14)

```
// vn soll auf m statt auf n zeigen; ist vn == null, zeigt root auf
void point (SearchNode vn, SearchNode n, SearchNode m){
    if (vn == null) root = m;
    else if (vn.left == n) vn.left = m;
        else               vn.right = m;
}
// liefert Vater des symmetrischen Nachfolgers:
SearchNode vSymNach (SearchNode n)
    if (n.right.left != null) {
        n = n.right;
        while (n.left.left != null) n = n.left;
    }
    return n;
}
```