Kapitel 10: Bäume

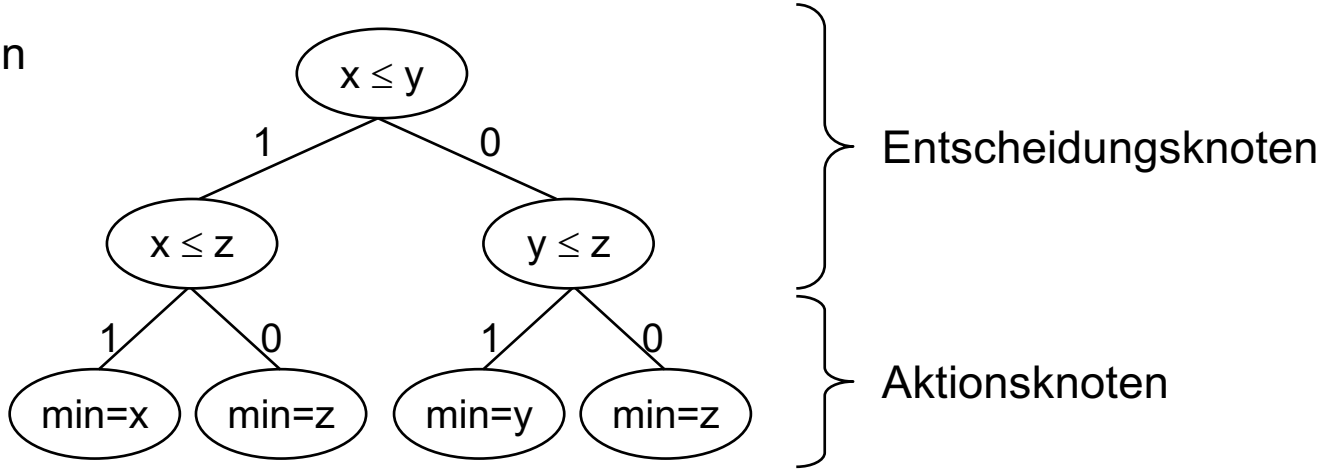
- Beispiele
- Definition und Eigenschaften
- Implementierungen
- Durchlaufen von Bäumen
- Binäre Suchbäume

Beispiele (1)

Bäume gehören zu den wichtigsten Datenstrukturen in der Informatik.

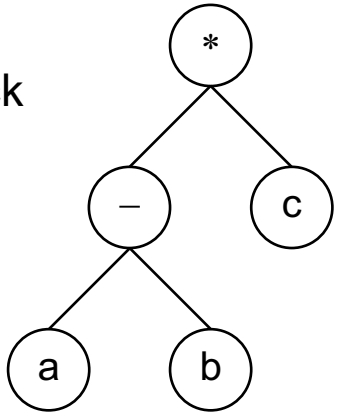
Entscheidungsbäume

Minimum von
3 Zahlen



Syntaxbäume

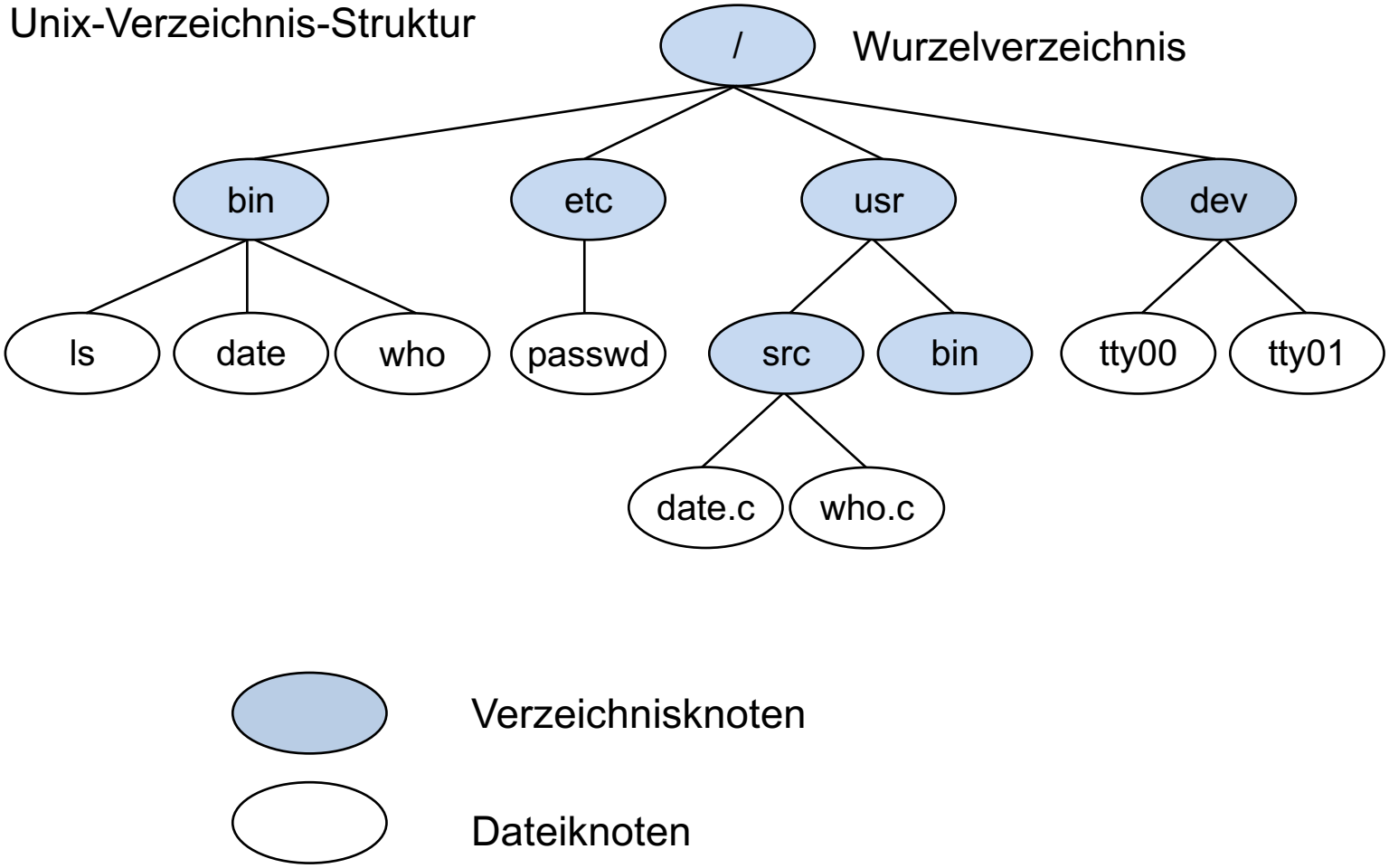
Syntaxbaum für den
arithmetischen Ausdruck
(a – b) ∗ c



Beispiele (2)

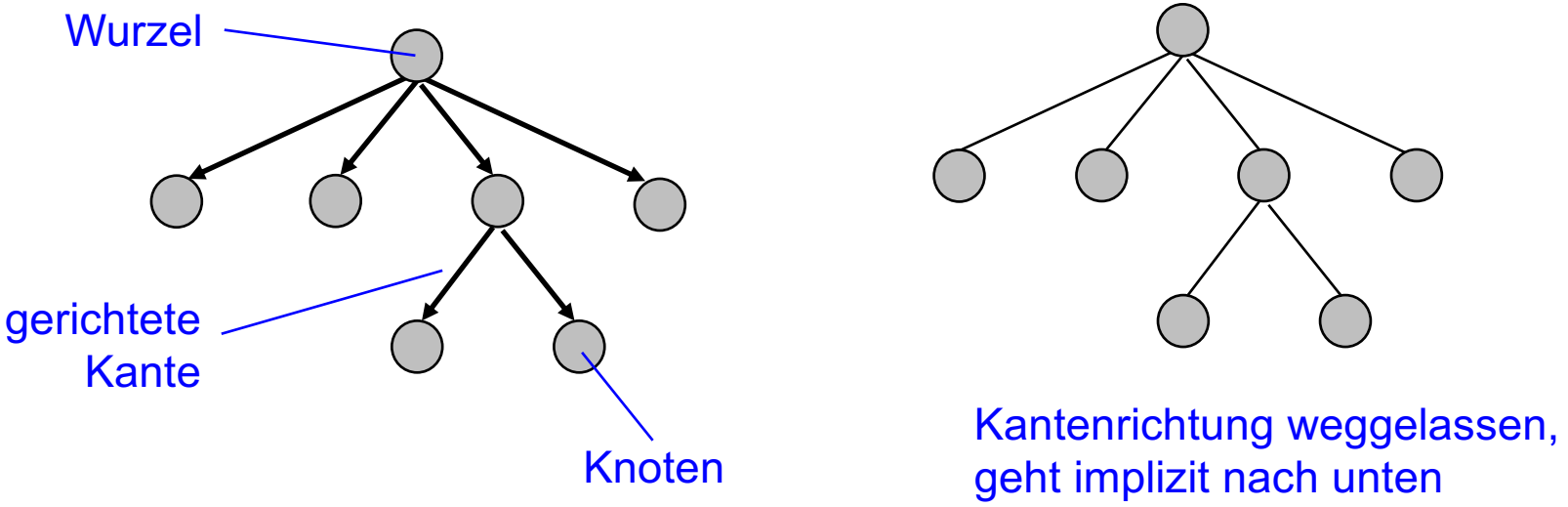
Verzeichnis-Strukturen

Ausschnitt aus einer
Unix-Verzeichnis-Struktur



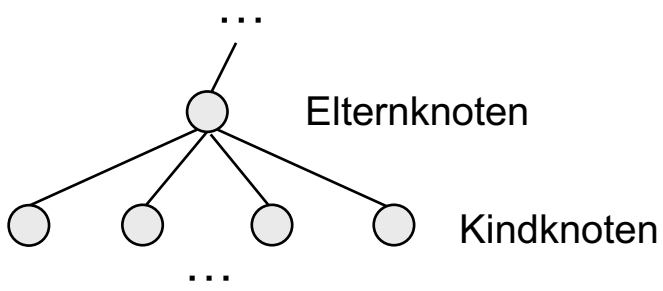
Bäume: Definition und Begriffe (1)

- Ein Baum besteht aus einer Menge von Knoten und einer Menge von gerichteten Kanten, die jeweils zwei Knoten verbinden.
- Es gibt genau einen Knoten, der keine eingehende Kante hat: die sogenannte Wurzel.
- Alle anderen Knoten haben genau eine eingehende Kante.
- Zyklen sind nicht erlaubt.
- Zur graphischen Darstellung:
  Wurzel steht meistens oben und die Kanten sind nach unten gerichtet.
  Kantenrichtung wird weggelassen und ist dann implizit nach unten.

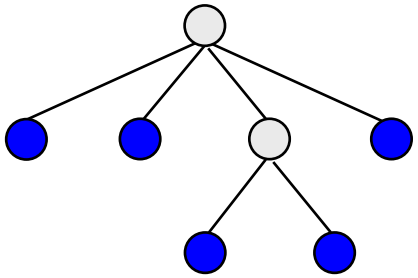


Bäume: Definition und Begriffe (2)

- Die Knoten unterhalb eines Knoten k werden auch Kindknoten und k Elternknoten genannt.



- Knoten ohne Kinder werden Blätter genannt.



- Ein Baum heißt geordnet, falls die Reihenfolge der Kinder eine Rolle spielt (s. Bsp. Syntaxbaum für arithm. Ausdruck)
- Üblicherweise enthalten die Knoten Daten. Man spricht dann von einem markierten Baum.

Bäume: Definition und Begriffe (3)

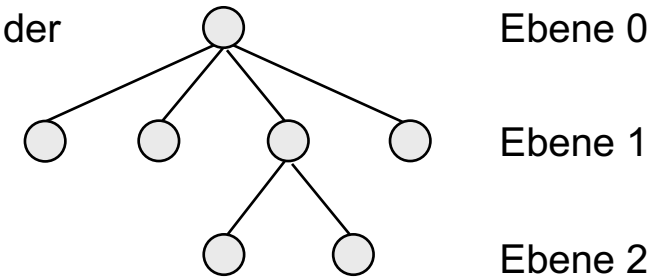
- Die Tiefe eines Knotens ist sein Abstand zur Wurzel,
  d.h. die Anzahl der Kanten von der Wurzel zu dem Knoten.

- Die Menge aller Knoten der Tiefe i wird auch i-te Ebene
  (level) genannt.

- Die Höhe eines Baums ist die maximale Tiefe,
  die von einem Knoten erreicht werden kann.

Beispiel

Ein Baum der Höhe 2.

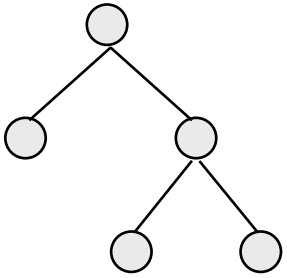


Binärbäume

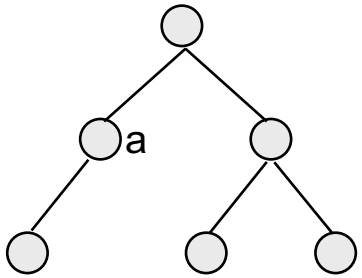
Definition

- Ein Binärbaum ist ein geordneter Baum, bei dem jeder Knoten maximal 2 Kinder hat.
- Die beiden Kinder werden linkes und rechtes Kind genannt.
- Hat ein Knoten nur ein Kind, dann muss es entweder ein linkes oder ein rechtes Kind sein.

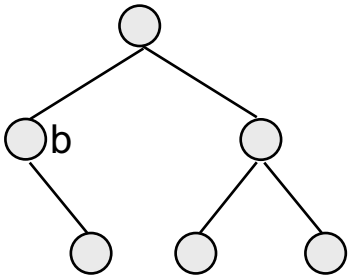
Beispiele



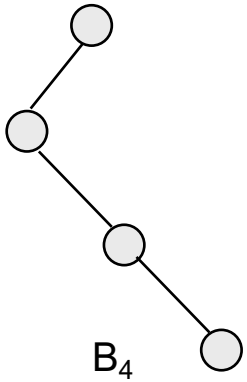
B_1



B_2



B_3



B_4

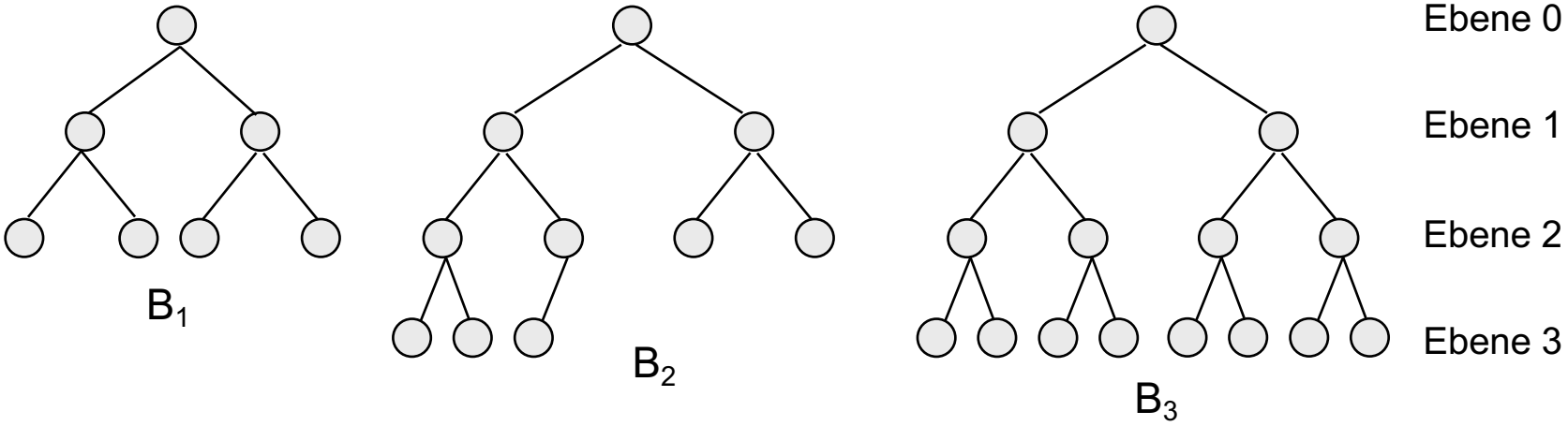
Bäume B_2 und B_3 sind strukturell unterschiedlich:
in B_2 hat Knoten a ein linkes Kind während
in B_3 Knoten b ein rechtes Kind hat.

Vollständige Binärbäume

Definition

Ein vollständiger Binärbaum ist ein Binärbaum, bei der jeder Ebene (bis auf die letzte) vollständig gefüllt und die letzte Ebene von links nach rechts lückenlos gefüllt ist.

Beispiele



Höhe von vollständigen Binärbäumen:

- Ein vollständiger Binärbaum der Höhe h mit vollständig gefüllter letzter Ebene hat

  $n = 2^0 + 2^1 + \ldots + 2^h = 2^{h+1} - 1$

  viele Knoten (vergleiche Baum B_3). Damit gilt auch

  $h = \log_2(n+1)-1 = \lfloor \log_2 n \rfloor$ (abgerundet!)

- Ein beliebiger vollständiger Binärbaum mit n Knoten (vergleiche Baum B_2) hat die Höhe

  $h = \lfloor \log_2 n \rfloor$

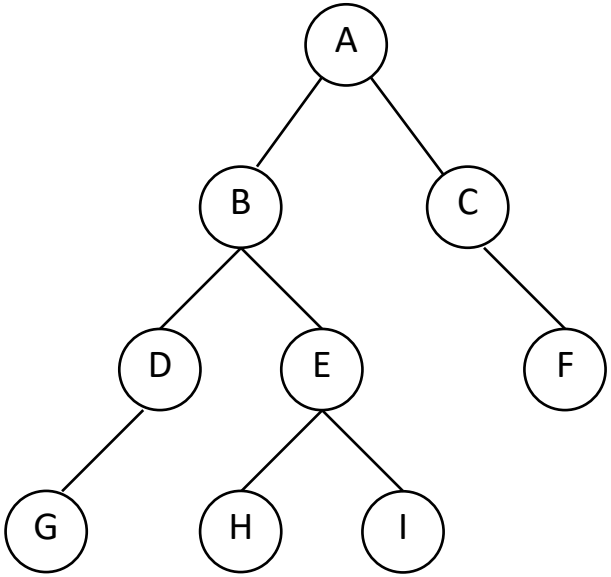
Kapitel 10: Bäume

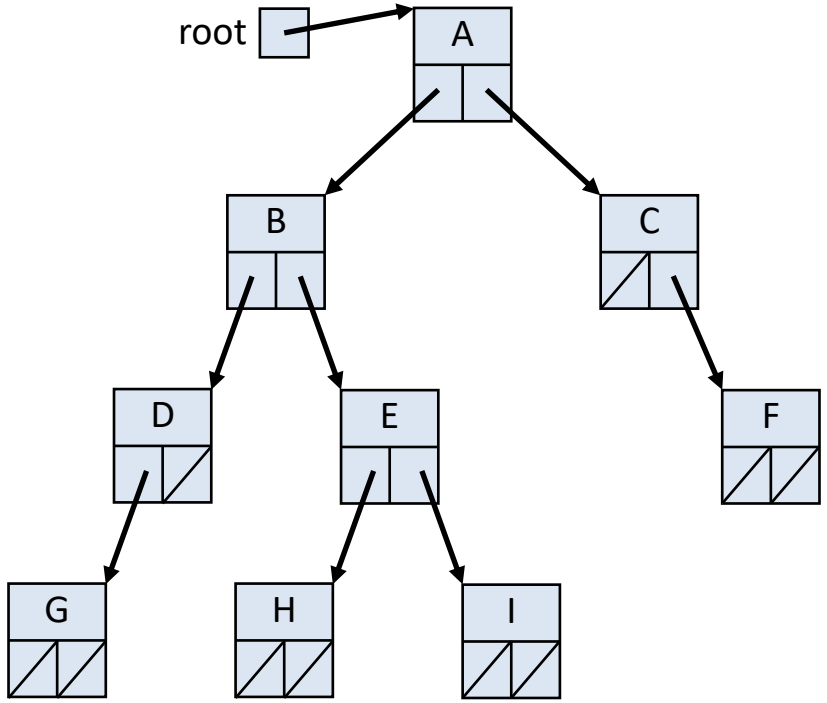
- Beispiele
- Definition und Eigenschaften
- Implementierungen
- Durchlaufen von Bäumen
- Binäre Suchbäume

Implementierung von Binärbäumen

Implementierung mit verketteten Knoten

- Jeder Knoten hat jeweils eine Referenz für das linke und das rechte Kind.



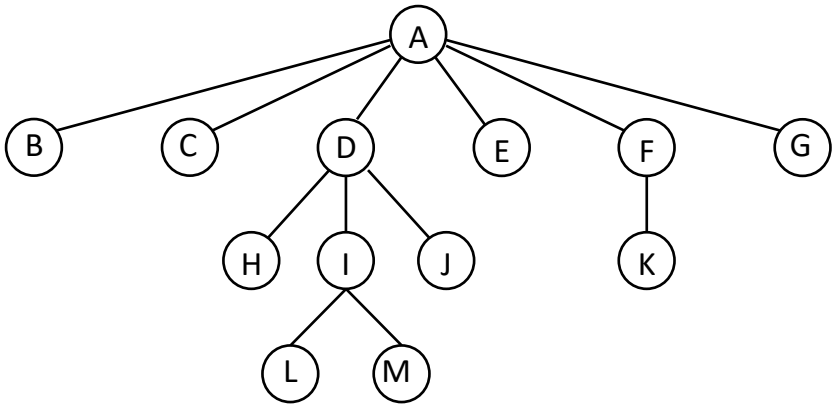


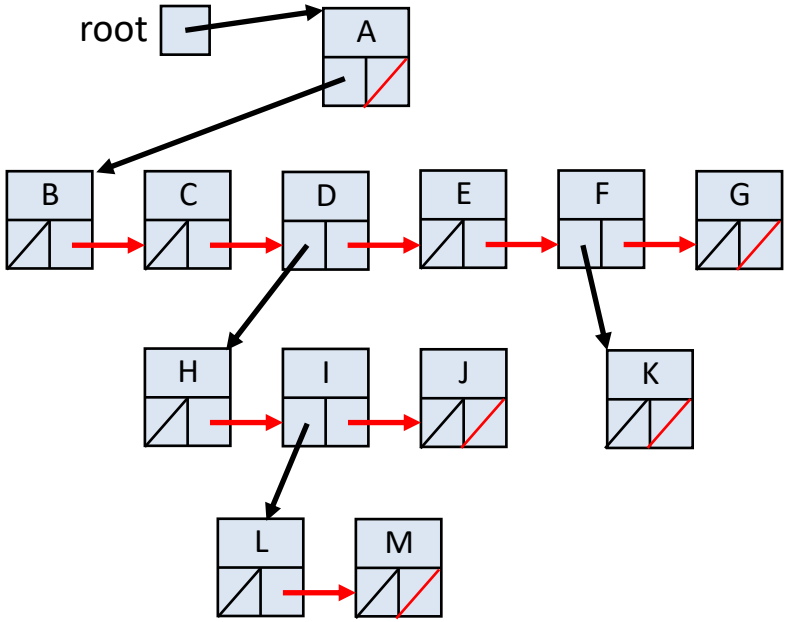

```
class Node {
    int data;
    Node left;   // linkes Kind
    Node right;  // rechtes Kind
}
```

Implementierung von beliebigen Bäumen (1)

Geschwister-Knoten linear verkettet:

- Jeder Knoten enthält 2 Referenzen.
- Eine Referenz für das erste Kind
- Eine Referenz für den rechts liegenden Geschwisterknoten (engl. sibling)





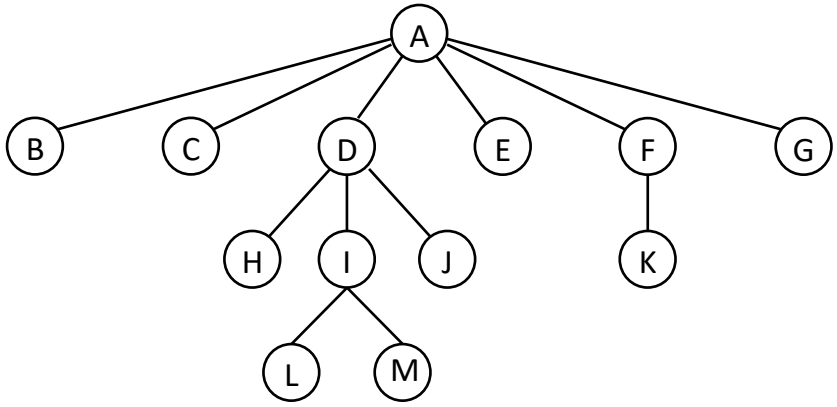

```
class Node {
    int data;
    Node firstChild;
    Node nextSibling;
}
```

Implementierung von beliebigen Bäumen (2)

Geschwisterknoten als LinkedList

- alternativ lassen sich auch auch die Geschwisterknoten in einem Listen-Container (z.B. LinkelList) abspeichern

```
class Node {
    int data;
    LinkedList<Node> children;
}
```



Kapitel 10: Bäume

- Beispiele
- Definition und Eigenschaften
- Implementierungen
- Durchlaufen von Bäumen
- Binäre Suchbäume

Überblick

Ziel

- Traversierung: Besuchen aller Knoten in einer bestimmten Reihenfolge.

Durchlaufreihenfolge

- PreOrder: besuche Wurzel und besuche dann alle Kinder;
- PostOrder: besuche alle Kinder und besuche dann Wurzel;
- InOrder: nur bei Binärbäumen:
  besuche linken Teilbaum; besuche Wurzel; besuche rechten Teilbaum;
- LevelOrder: besuche Knoten ebenenweise

Bemerkungen

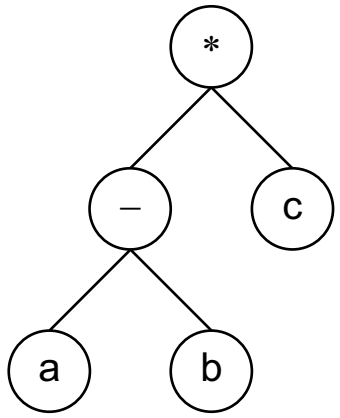
- Die Präfixe *Pre*, *Post* bzw. *In* bedeuten vorher, nachher und dazwischen.
  Gemeint ist damit der Zeitpunkt, an dem die Wurzel besucht wird.
- Einfachheitshalber werden in den folgenden Beispielen die
  Traversierungsalgorithmen nur für Binärbäume behandelt.

PreOrder-Durchlauf

```
static void preOrder(Node p) {
    if (p != null){
        bearbeite(p.data);
        preOrder(p.left);
        preOrder(p.right);
    }
}
```

```
static class Node {
    int data;
    Node left;
    Node right;
}

private Node root;

//...

static void main(...) {
    preOrder(root);
}
```

Beispiel

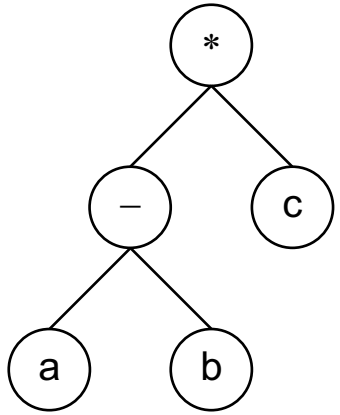


Durchlaufreihenfolge: * - a b c

PostOrder-Durchlauf

```
static void postOrder(Node p) {
    if (p != null){
        postOrder(p.left);
        postOrder(p.right);
        bearbeite(p.data);
    }
}
```

Beispiel



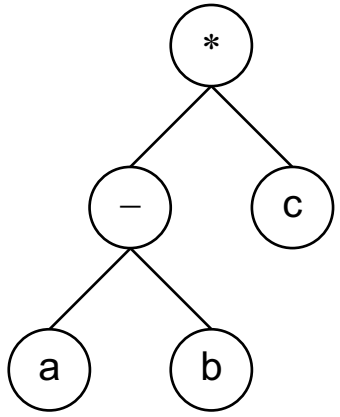
Durchlaufreihenfolge: a b – c *

(Entspricht der sog. Postfix-Notation für arithmetische Ausdrücke)

InOrder-Durchlauf für Binärbäume

```
static void inOrder(Node p) {
    if (p != null){
        inOrder(p.left);
        bearbeite(p.data);
        inOrder(p.right);
    }
}
```

Beispiel

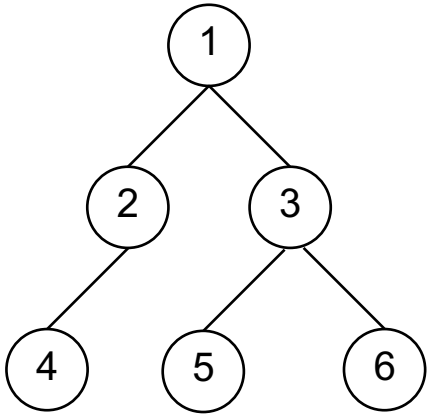


Durchlaufreihenfolge: a - b * c

LevelOrder-Durchlauf

```
static void levelOrder(Node p) {
    Queue<Node> queue = new ArrayDeque<Node>();
    if (p != null)
        queue.add(p);

    while (!queue.isEmpty()) {
        Node q = queue.remove();
        bearbeite(q.data);
        if (q.left != null)
            queue.add(q.left);
        if (q.right != null)
            queue.add(q.right);
    }
}
```

Die Knoten werden ebenenweise in einer Schlange gespeichert und in einer while-Schleife abgearbeitet.

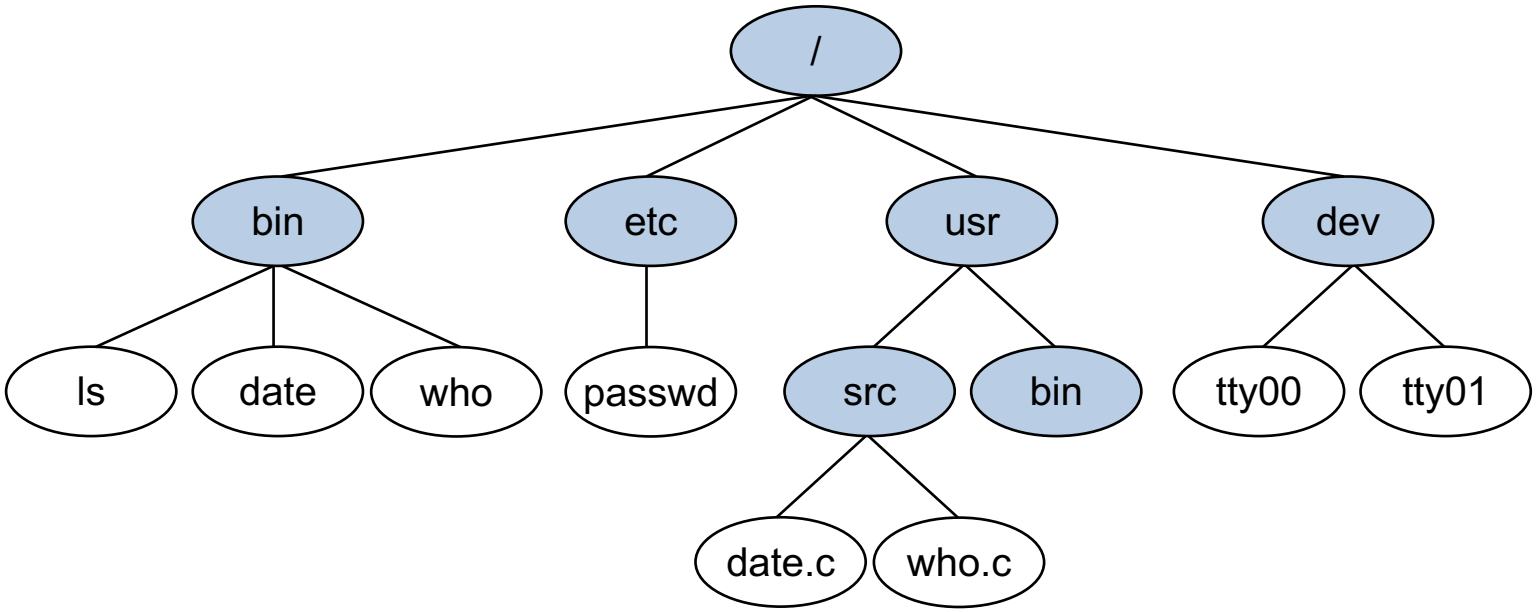


Durchlaufreihenfolge: 1, 2, 3, 4, 5, 6

Aufgabe 10.1

- Die Klasse `Dir` gestattet den Aufbau baumartiger Verzeichnis-strukturen.
- Ein Verzeichnis besteht aus einem Namen (`String`) und einer Liste (`LinkedList`) von Unterverzeichnissen.

```
class Dir {
    private String name;
    private LinkedList<Dir> dirs;

    public Dir(String n) {
        name = n;
        dirs = new LinkedList<>();
    }
    // ...
}
```



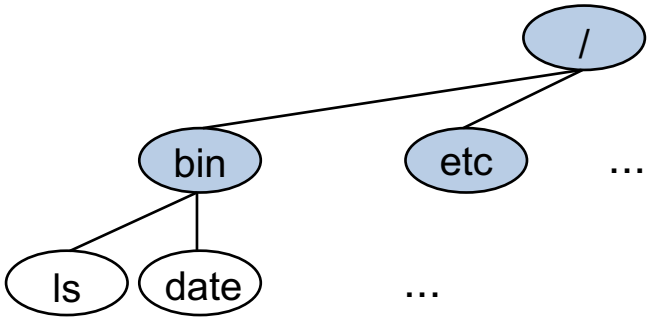
Aufgabe 10.1 (Forts.)

```
class Dir {
    // ...

    public void mkDir(String n) {
        dirs.add(new Dir(n));
    }

    public Dir cd(String n) {
        for (Dir f : dirs)
            if (f.name.equals(n))
                return f;
        return null;

    public static void main() {
        Dir root = new Dir("/");
        root.mkDir("bin");
        root.mkdir("etc");
        // ...
        root.cd("bin").mkdir("ls");
        root.cd("bin").mkdir("date");
        // ...
    }
}
```

mkDir fügt zum aktuellen Verzeichnis ein Unterverzeichnis mit dem Namen n hinzu.

Mit cd kann in ein Unterverzeichnis mit dem Namen n gewechselt werden, indem das entsprechende Unterverzeichnis zurückliefert wird.

main baut eine kleine Verzeichnisstruktur auf.



Aufgabe 10.1 (Forts.)

- Beschreiben Sie anhand eines Beispiels, wie Sie Verzeichnisstrukturen textuell ausgeben würden. Welche Durchlaufreihenfolge wählen Sie dabei? Schreiben Sie eine Methode

  ```
  void print()
  ```

  um Verzeichnisstrukturen auszugeben.

- Schreiben Sie eine Methode

  ```
  int getSize()
  ```

  die die Anzahl aller Unterverzeichnisse (Anzahl aller Knoten) bestimmt und zurückliefert.

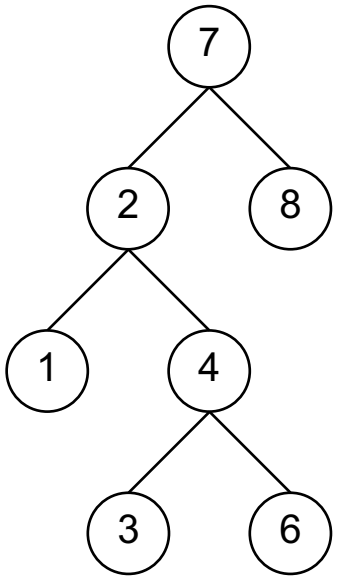
Kapitel 10: Bäume

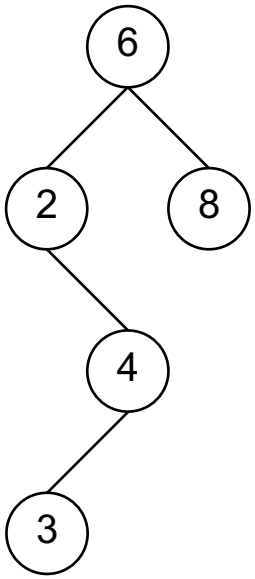
- Beispiele
- Definition und Eigenschaften
- Implementierungen
- Durchlaufen von Bäumen
- Binäre Suchbäume

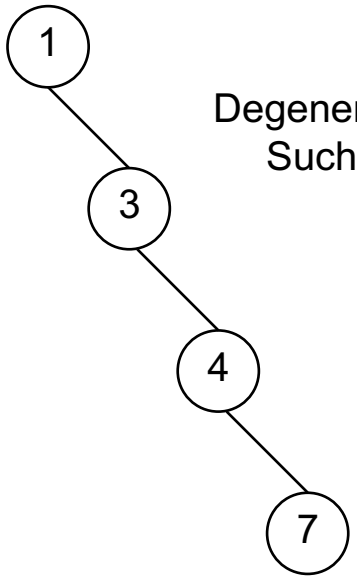
Definition binärer Suchbäumen

- Ein binärer Suchbaum ist ein Binärbaum, bei dem für alle Knoten k folgende Eigenschaften gelten:
  - Alle Zahlen im linken Teilbaum sind kleiner als k
  - Alle Zahlen im rechten Teilbaum sind größer als k
- Beachte, dass die Zahlen hier eindeutig sein müssen. Es ist aber auch möglich, dass gleiche Zahlen mehrfach vorkommen dürfen, was kleinere Änderungen in den Algorithmen erfordert.

Beispiele:







Degenerierter Suchbaum

Klasse BinarySearchTree

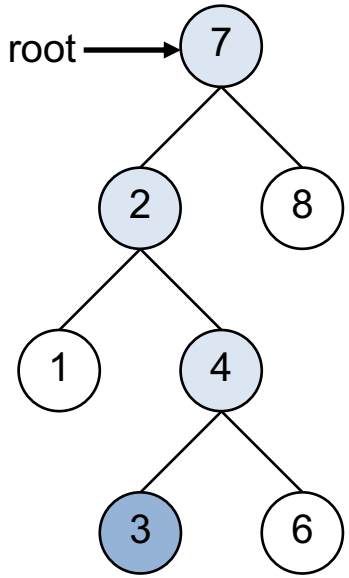
```
public class BinarySearchTree {

    private static class Node {
       int data;
       Node left;
       Node right;
       Node(int x) {
            data = x;
            left = null;
            right = null;
       }
    }

    private Node root = null;

    // ...
}
```

Suchen in binären Suchbäumen

```
public boolean contains(int x) {
    return containsR(x,root);
}

private boolean containsR(int x, Node p) {
    if (p == null)
        return false;
    else if (x < p.data)
        return containsR(x, p.left);
    else if (x > p.data)
        return containsR(x, p.right);
    else
        return true;
}
```

Private rekursive Methode.

Beispiel



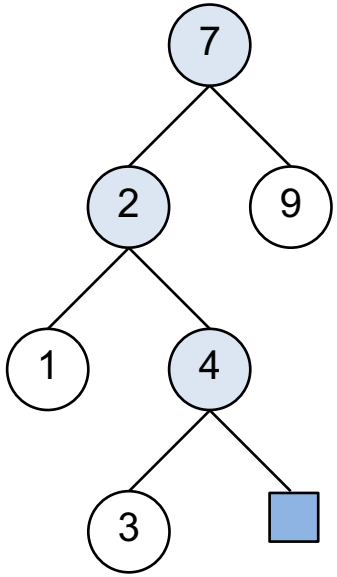
containsR(3, root)
liefert true

Einfügen in binären Suchbäumen (1)

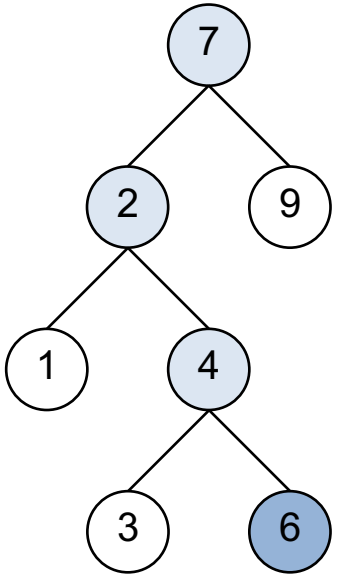
Idee

- Um eine Zahl x einzufügen, wird zunächst nach x gesucht.
- Falls x nicht bereits im Baum vorkommt,
  endet die Suche erfolglos bei einer null-Referenz.
- An dieser Stelle wird dann ein neuen Knoten mit Eintrag x eingefügt.

Beispiel 1: füge 6 ein

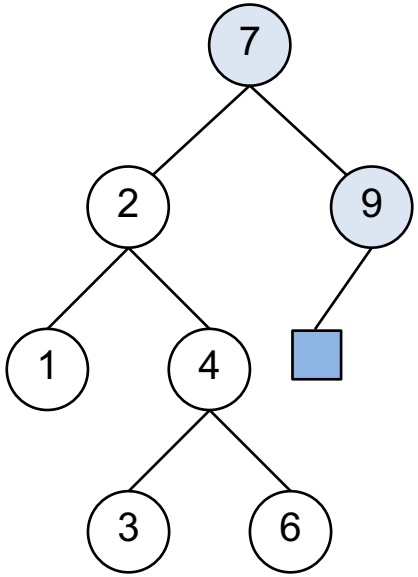


Suche von 6 endet bei null

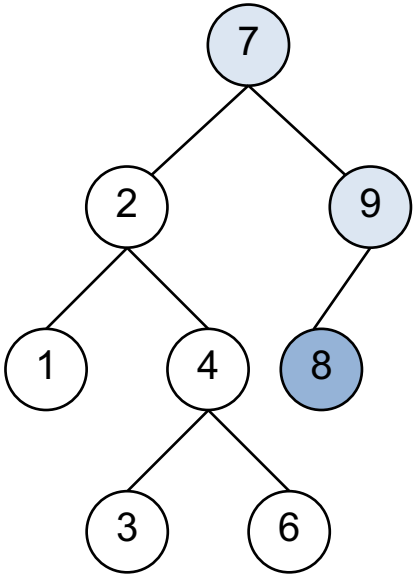


Ersetzte null durch neuen Knoten mit Eintrag 6

Beispiel 2: füge 8 ein



Suche von 8 endet bei null

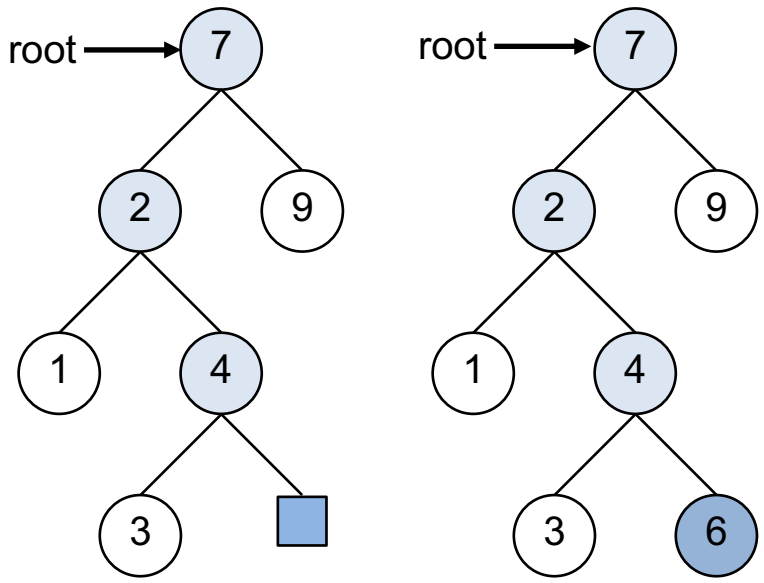


Ersetzte null durch Knoten mit Eintrag 8

Einfügen in binären Suchbäumen (2)

```
public void insert(int x) {
    root = insertR(x,root);
}

private Node insertR(int x, Node p) {
    if (p == null)
        p = new Node(x);
    else if (x < p.data)
        p.left = insertR(x,p.left);
    else if (x > p.data)
        p.right = insertR(x,p.right);

    return p;
}
```

Da keine doppelten Werte erlaubt sind, ist im else-Fall nichts zu tun.



insertR(6, root);

Löschen in binären Suchbäumen (1)

Idee

- Um eine Zahl x zu löschen, wird zunächst nach x gesucht.
  Es sind dann 4 Fälle zu unterscheiden:

- Fall „Nicht vorhanden“:

  x kommt nicht vor. Dann ist nichts zu tun.

- Fall „Keine Kinder“:

  x kommt in einem Blatt vor (keine Kinder):
  dann kann der Knoten einfach entfernt werden.

- Fall „Ein Kind“:

  Der Knoten, der x enthält, hat genau ein Kind:
  s. nächste Folie

- Fall „Zwei Kinder“:

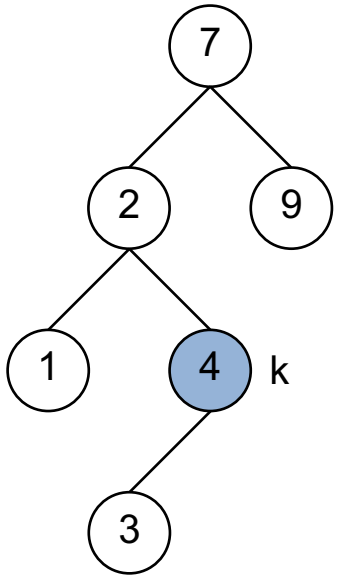
  Der Knoten, der x enthält, hat zwei Kinder:
  s. übernächste Folie

Löschen in binären Suchbäumen (2)

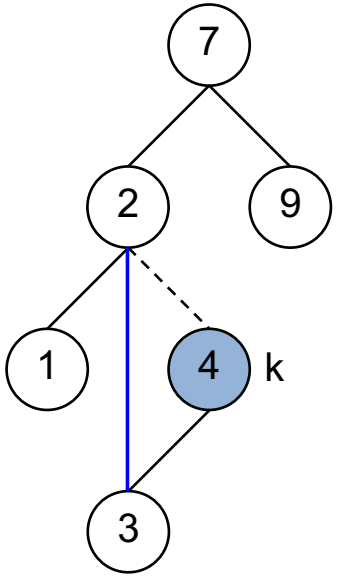
Fall: der zu löschende Knoten k hat ein Kind

- Lösche (überbrücke) den Knoten k, indem der Elternknoten von k auf das Kind von k verzeigert wird.

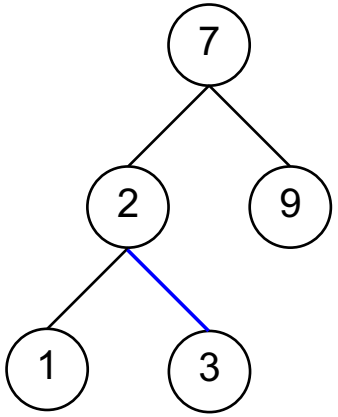
Beispiel: lösche Knoten k mit Inhalt 4



Suche 4



Knoten 4 wird "gelöscht"
(überbrückt)

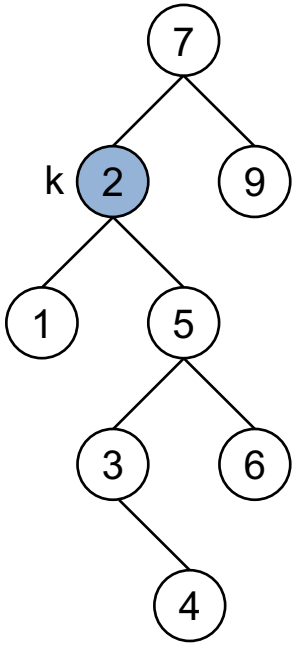


Löschen in binären Suchbäumen (3)

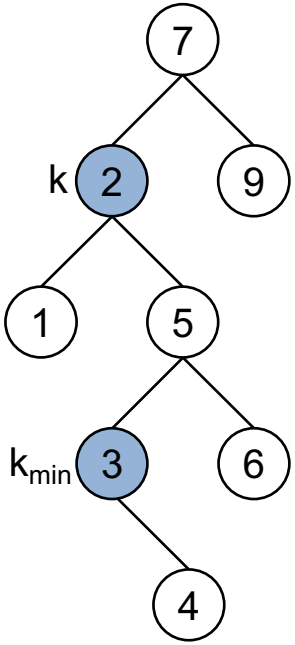
Fall: der zu löschende Knoten k hat zwei Kinder

- Ersetze den Knoten k durch den kleinsten Knoten k_{min} im rechten Teilbaum von k.
- Lösche dann k_{min}.
- Da der Knoten k_{min} kein linkes Kind haben kann, kann das Löschen von k_{min} wie im Fall „Ein Kind“ bzw. „Keine Kinder“ behandelt werden.

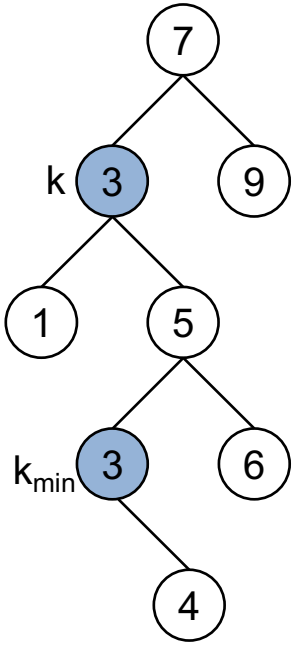
Beispiel: lösche Knoten k mit Inhalt 2



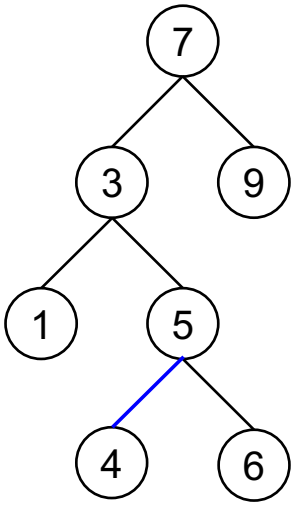
Suche k mit Inhalt 2



Suche kleinsten Knoten k_{min} in rechten Teilbaum von k



Knoten k wird durch k_{min} ersetzt



k_{min} wird gelöscht

Löschen in binären Suchbäumen (4)

```
public void remove(int x) {
    root = removeR(x,root);
}

private Node removeR(int x, Node p) {
    if (p == null)
        p = null;
    else if (x < p.data)
        p.left = removeR(x,p.left);
    else if (x > p.data)
        p.right = removeR(x,p.right);
    else { // Knoten loeschen:
        if (p.left == null || p.right == null)
            p = (p.left != null) ? p.left : p.right;
        else {
            p.data = getMin(p.right);
            p.right = removeR(p.data,p.right);
        }
    }
    return p;
}

private int getMin(Node p) {
    assert (p != null);
    while(p.left != null)
        p = p.left;
    return p.data;
}
```

Fall ein oder kein Kind

Fall zwei Kinder

Analyse

Worst-Case

- Im schlechtesten Fall kann ein binärer Suchbaum mit n Knoten zu einem Baum der Höhe n-1 entarten.
- Damit: $T_{max}(n) = O(n)$

Average-Case

- In [Ottmann und Widmayer 2002] werden zwei Ergebnisse hergeleitet, die sich darin unterscheiden, welche Verteilung der Bäume angenommen wird.
- Bäume mit n Knoten entstehen durch eine Folge von Einfüge-Operationen von n unterschiedlichen Elementen. Es wird angenommen, dass jede der n! möglichen Anordnungen der Elemente gleich wahrscheinlich ist.

  Damit: $T_{mit}(n) = O(\log_2 n)$
- Es wird angenommen, dass alle strukturell verschiedenen binären Suchbäume mit n Knoten gleichwahrscheinlich sind.

  Damit: $T_{mit}(n) = O(\sqrt{n})$

Aufgaben (1)

Aufgabe 10.2

Geben Sie den binären Suchbaum an, nachdem mit einem leeren Baum begonnen wurde und folgende Operationen durchgeführt worden sind:

- Einfügen von: 10, 5, 3, 8, 2, 4, 1, 15, 12, 17, 6, 7, 9
- Löschen von: 2, 3, 5, 12, 10

Aufgabe 10.3

Beseitigen Sie die beiden endrekursiven Aufrufe in der Funktion `containsR`.

Aufgabe 10.4

Schreiben Sie einen Kopierkonstruktor `BinarySearchTree(BinarySearchTree t)`.

Aufgabe 10.5

Überschreiben Sie in der Klasse BinarySearchTree die Methode `Object.toString()`.

Aufgaben (2)

Aufgabe 10.6

Schreiben Sie eine Methode `isSearchTree()`, die überprüft, ob ein Binärbaum die Suchbaumeigenschaft erfüllt.

Aufgabe 10.7

Schreiben Sie eine Methode, die die ersten n Zahlen einer sortierten Liste l löscht und daraus einen optimal ausbalanzierten Suchbaum aufbaut und zurückliefert.

```
private Node createTreeR(List<Integer> l, int n);
```

Hinweis: Teile-und-Herrsche-Verfahren.

Aufgabe 10.8

Schreiben Sie eine Methode `balance()`, die einen Suchbaum optimal ausbalanziert.

Hinweis: Benutzen Sie die Methode aus Aufgabe 10.7.

Aufgaben (3)

Aufgabe 10.9

Ergänzen Sie die Klasse BinarySearchTree um einen Iterator, der alle Elemente des Suchbaums in sortierter Reihenfolge durchläuft.

Damit würde sich ein Suchbaum mit einer foreach-Schleife traversieren lassen:

```
static void main(String[] args) {
    BinarySearchTree t = new BinarySearchTree();

    // Elemente einfuegen:
    int[] a = {7,2,8,1,4,3,6};
    for (int x : a)
        t.insert(x);

    // Elemente sortiert ausgeben:
    for (int x : t)
        System.out.println(x);
}
```

Entwickeln Sie zwei Lösungen:

- Alle Elemente des Suchbaums werden zuerst in einer InOrder-Reihenfolge in einer Liste abgespeichert. Dann kann über die Liste iteriert werden.
- Verwenden Sie einen Stack, um die Elemente des Suchbaums in InOrder-Reihenfolge zu traversieren.

Bemerkungen (1)

- Wie üblich kann der Elementtyp in BinarySearchTree auch generisch definiert werden: BinarySearchTree<E>.
- Elemente müssen vergleichbar sein. 2 Ansätze (wie bei generischer Sortiermethode):
  - Elementtyp E wird auf Comparable beschränkt.
  - Übergabe eines Comparator-Parameters

```
class BinarySearchTree<E implements Comparable<? super E>> {
        // ...
}
```

```
class BinarySearchTree<E> {
        private Comparator<E> cmp;
        public BinarySearchTree(Comparator<? super E> cmp) {
                this.cmp = cmp;
        }
        // ...
}
```

- Binäre Suchbäume werden benutzt, um Mengen (Sets) zu realisieren.
- In der Javi-API gibt es für Sets bereits geeignete Typen:
  Interface Set und Implementierung TreeSet (später).

Bemerkungen (2)

- Binäre Suchbäume werden auch benutzt, um Mengen von Schlüssel-Wert-Paaren (Maps) zu verwalten. Die Schlüssel (key) entsprechen den Zahlen in den Suchbäumen und die Werte (value) sind zusätzliche Nutzdaten.

  Naheliegend ist auch hier ein generischer Typ.

  Auch hier muss der Schlüsseltyp Comparable sein oder ein Comparator-Objekt als Parameter übergeben werden.

```
class BinarySearchTree<K implements Comparable<? super K>, V> {
        // ...
}
```

K = key
V = value

- Typische Anwendungen von Maps: Telefonbücher, Wörterbücher (deutsch-englisch), etc.

```
BinarySearchTree<String, Integer> telBuch = new BinarySearchTree<>();

BinarySearchTree<String, List<String>> WoerterBuch = new BinarySearchTree<>();
```

- In der Javi-API gibt gibt es für Maps bereits geeignete Typen: Interface Map und Implementierung TreeMap (später).