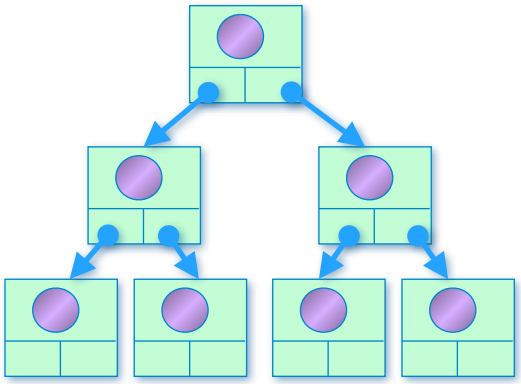
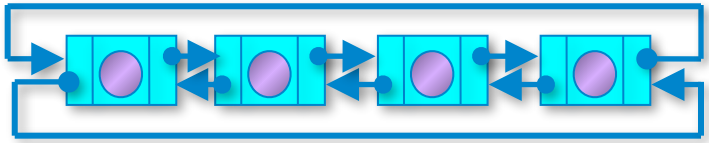
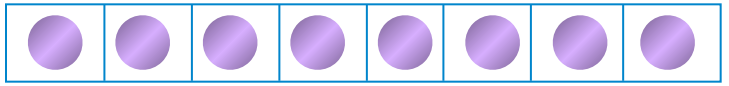
Freie Universität Berlin

OOP
Dynamische Datenmengen
Datenabstraktion
Teil III



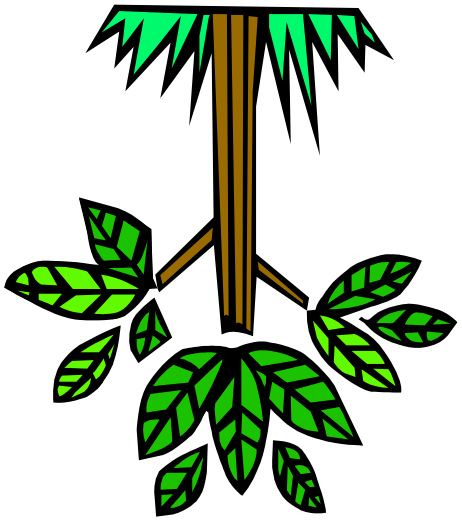




SoSe 2018

Oliver Wiese

Inhalt

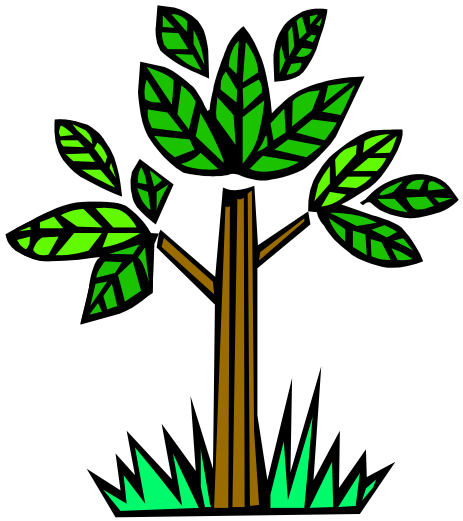


1. Einführung
2. Warum Bäume?
3. Listen und Arrays vs. Bäume
4. Einfach verkettete binäre Suchbäume
   - Baumtraversierung
   - Suchen
   - Einfügen
5. Doppelt verkettete binäre Bäume
   - Löschen

Warum Bäume?

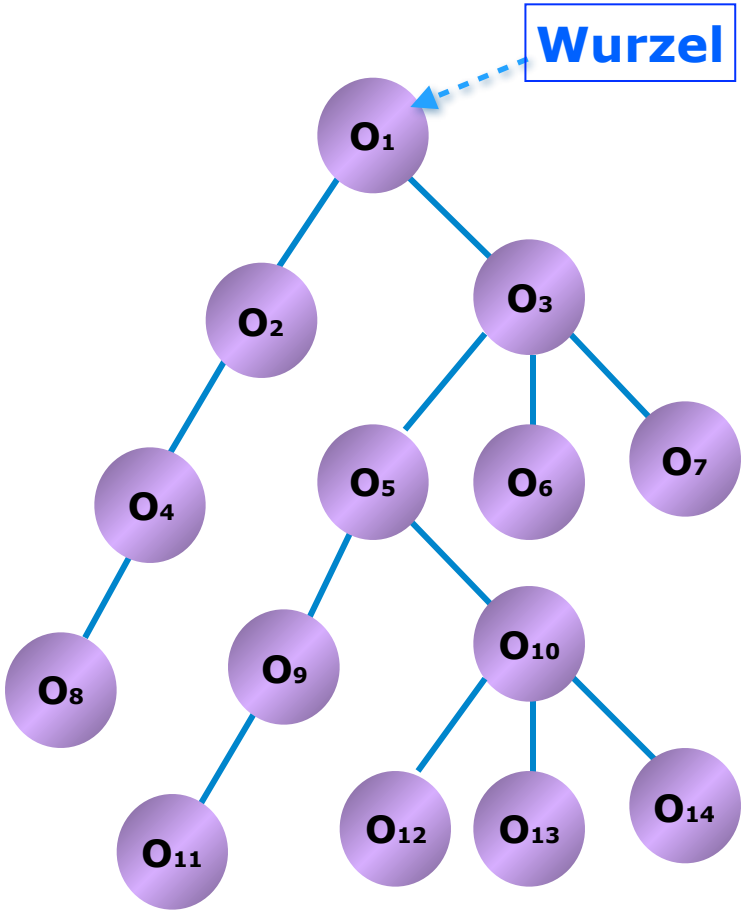
Bäume sind fundamentale Datenstrukturen für:

Betriebssysteme	CFS von Linux (RB-Baum)
Datenbanken	B-Bäume, R-Bäume
Übersetzerbau	Abstrakte Syntaxbäume
Textverarbeitung	
3D Graphik-Systeme	
Datenkompression	Huffman-Kodierung
KI, Spiele	Entscheidungsbäume
… usw.	



Was ist ein Baum?

Eine spezielle Graph-Struktur ohne Zyklen



1. Er hat eine Wurzel.
2. Alle Knoten außer der Wurzel haben genau eine Verbindung mit einem Vorfahren.
3. Es existiert genau ein Weg zwischen der Wurzel und jedem beliebigen Knoten.

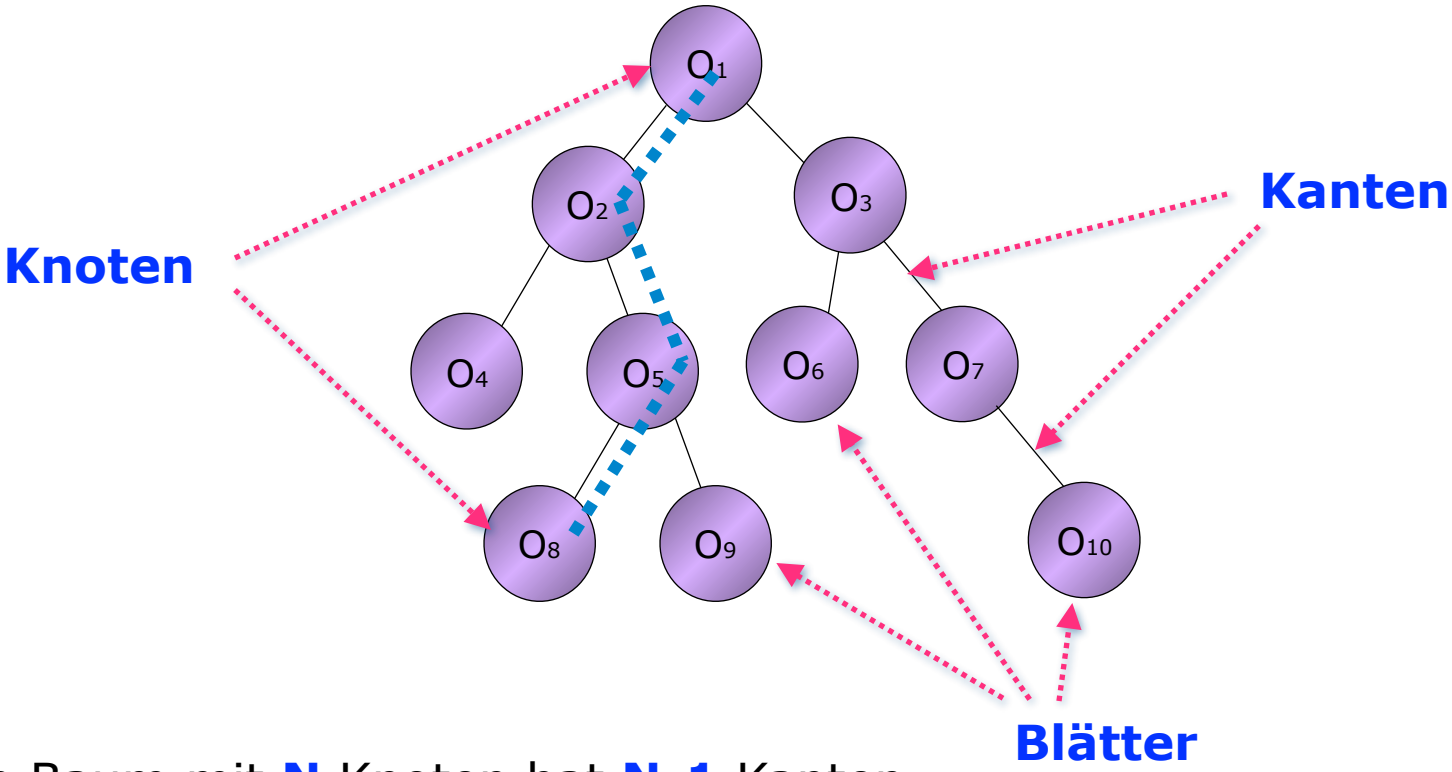
Eigenschaften von Bäumen

Nehmen wir an, wir haben einen Baum **t**, dann gilt:

- **|*t*|** bezeichnet die Größe des Baumes ***t*** oder die gesamte Anzahl seiner Knoten.
- Die Tiefe (*Level*) eines Knotens ist sein Abstand zur Wurzel. Die Tiefe der Wurzel ist gleich 0.
- Die Höhe ***h*(*t*)** ist der maximale Abstand zwischen der Wurzel und den Knoten.
- Blätter sind Knoten ohne Kinder.
- Die Pfadlänge des Baumes sei definiert als die Summe der Tiefen aller Knoten des Baumes.

Eigenschaften von Bäumen

- Zwischen zwei beliebigen Knoten in einem Baum existiert genau ein Pfad, der sie verbindet.

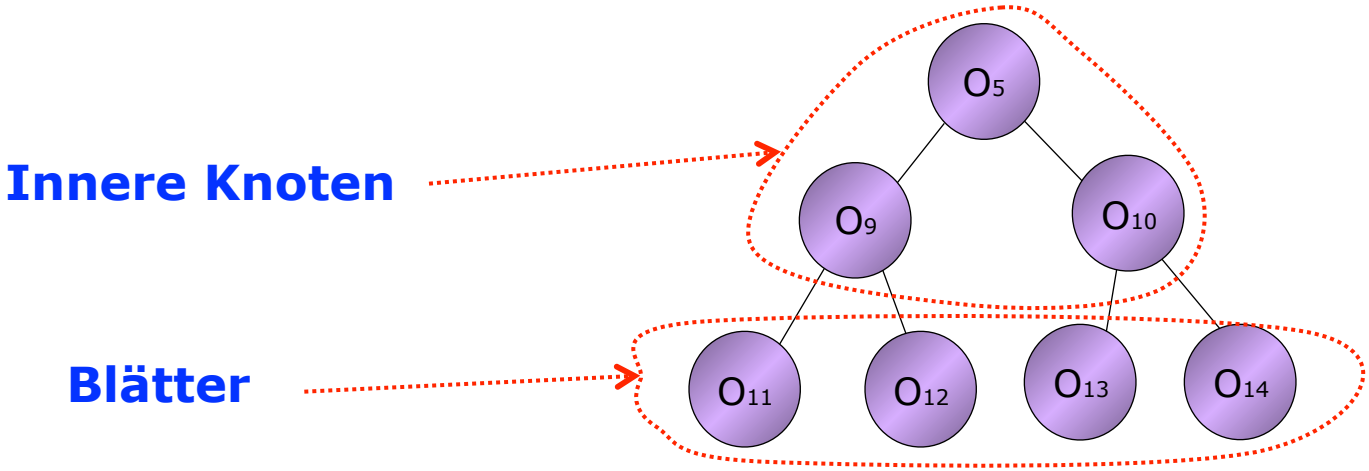


- Ein Baum mit **N** Knoten hat **N-1** Kanten.

Binärbäume

Bäume, in denen jeder Knoten höchstens zwei Kinder hat.

Ein Binärbaum mit **N** inneren Knoten hat **N+1** äußere Knoten oder Blätter.



Warum Bäume?

Weil die Grundoperationen für dynamische Datenmengen damit viel effizienter realisiert werden können.

Elementare Operationen für dynamische Mengen
- Suchen
- Einfügen
- Löschen

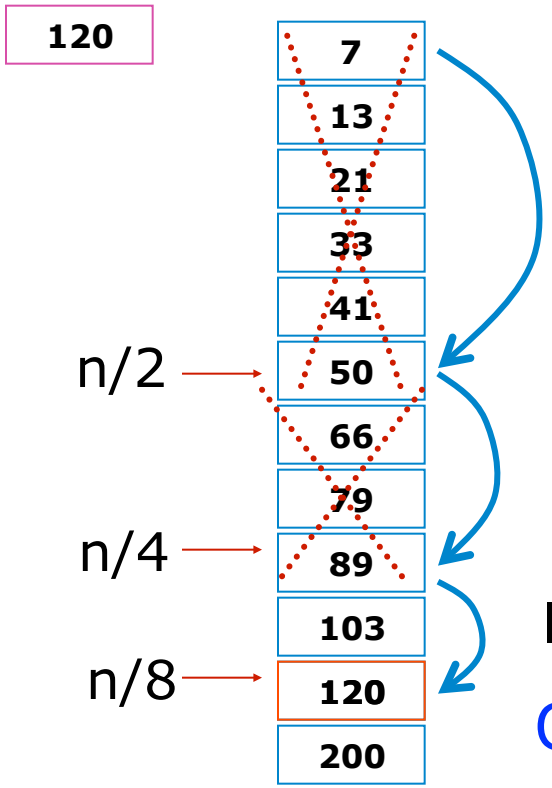
Bäume kombinieren die Vorteile der zwei Datenstrukturen, die wir bereits diskutiert haben: Felder (Arrays) und Listen.

Suchen

Sortiertes Array

mit n Elementen

Binärsuche



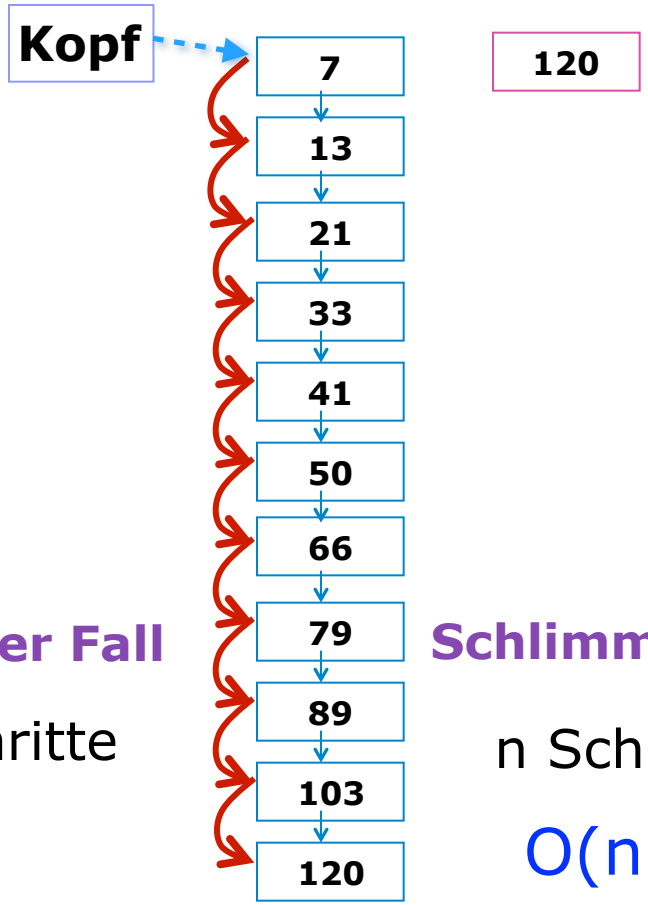
Schlimmster Fall

$\log_2(n)$ Schritte

$O(\log_2 n)$

Sortierte Liste

mit n Elementen



Schlimmster Fall

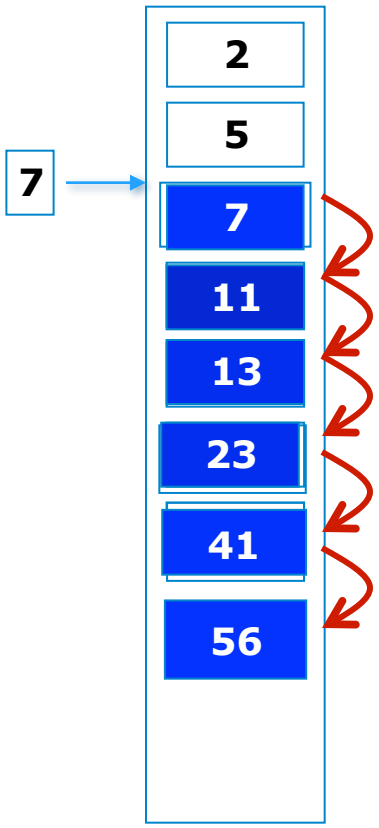
n Schritte

$O(n)$

Einfüge- und Lösch-Operationen

(wenn die Position bereits bekannt ist)

Array



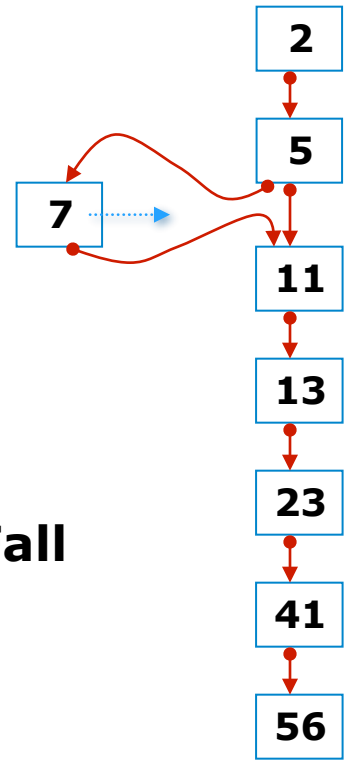
Die Laufzeit hängt linear von der Länge der bereits gespeicherten Daten ab.

Schlimmster Fall

n Schritte

O(n)

Liste

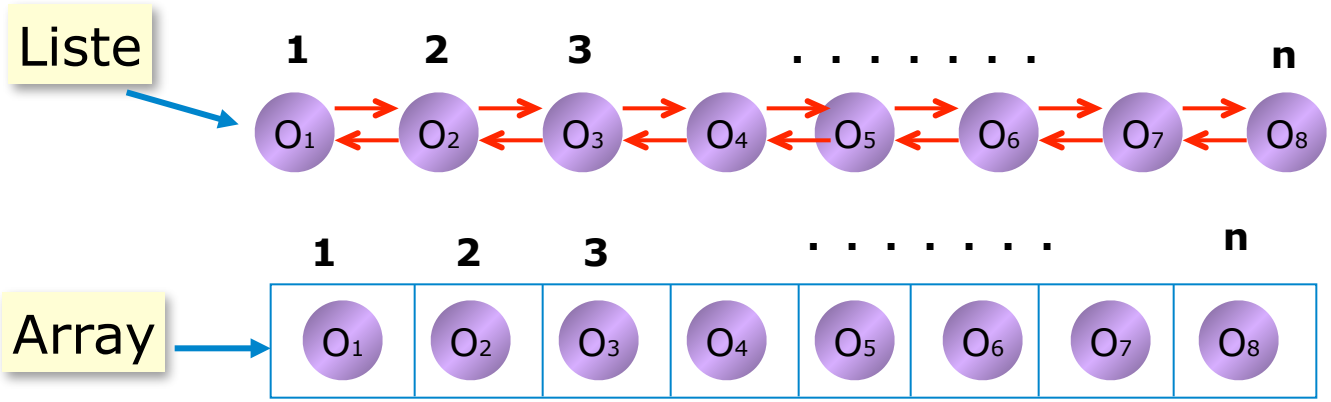


Die Laufzeit ist immer konstant.

Schlimmster Fall

O(1)

Liste vs. Array

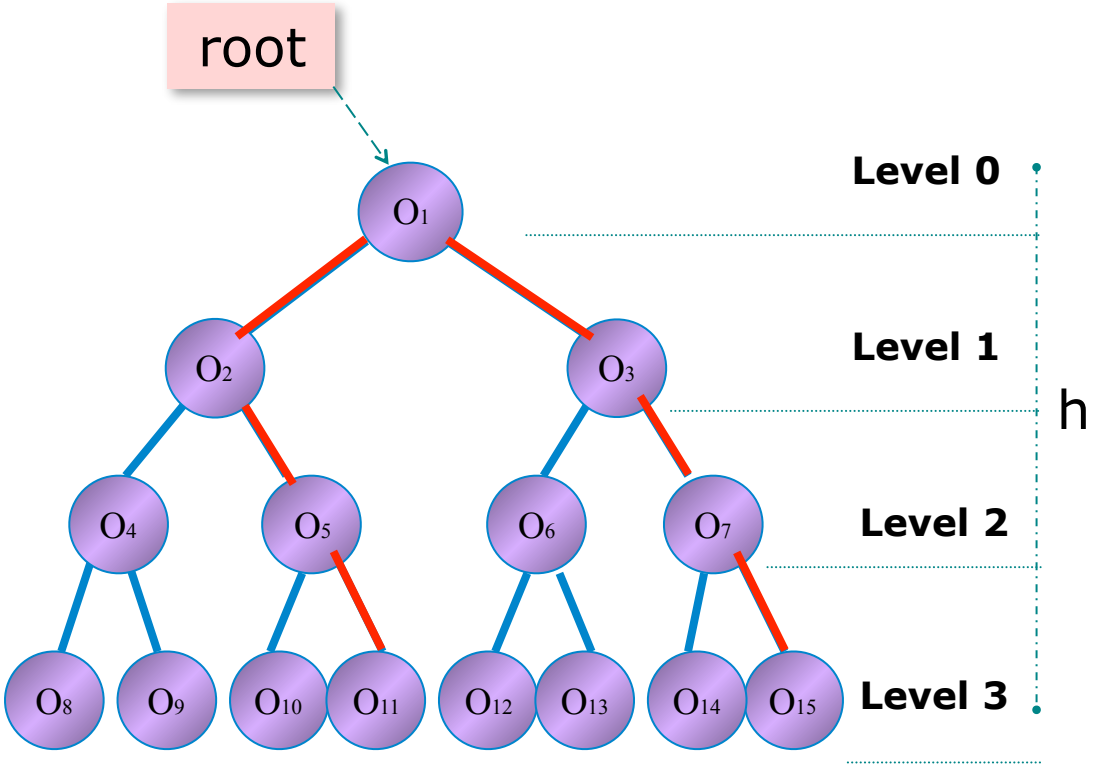


Elementare Operationen für dynamische Mengen

	Liste sortiert	Liste nicht sortiert	Array sortiert	Array nicht sortiert
Suchen	O(n)	O(n)	$O(\log_2(n))$	O(n)
Einfügen	O(n)	O(1)	O(n)	O(1)
Löschen	O(n)	O(n)	O(n)	O(n)

Vollständige Binärbäume

Ein vollständiger binärer Baum hat $2^h - 1$ innere Knoten und 2^h Blätter



$$n = 2^{h+1} - 1$$

$$\Downarrow$$

$$n + 1 = 2^{h+1}$$

$$\Downarrow$$

$$\log_2(n+1) = \log_2(2^{h+1})$$

$$\Downarrow$$

$$\log_2(n+1) = h+1$$

$$\Downarrow$$

$$\boxed{h = \lceil \log_2(n+1) \rceil - 1}$$

Eigenschaften von Binär-Bäumen

Rekursive Definitionen:

Anzahl der inneren Knoten

$$|t| = |t_l| + |t_r| + 1$$

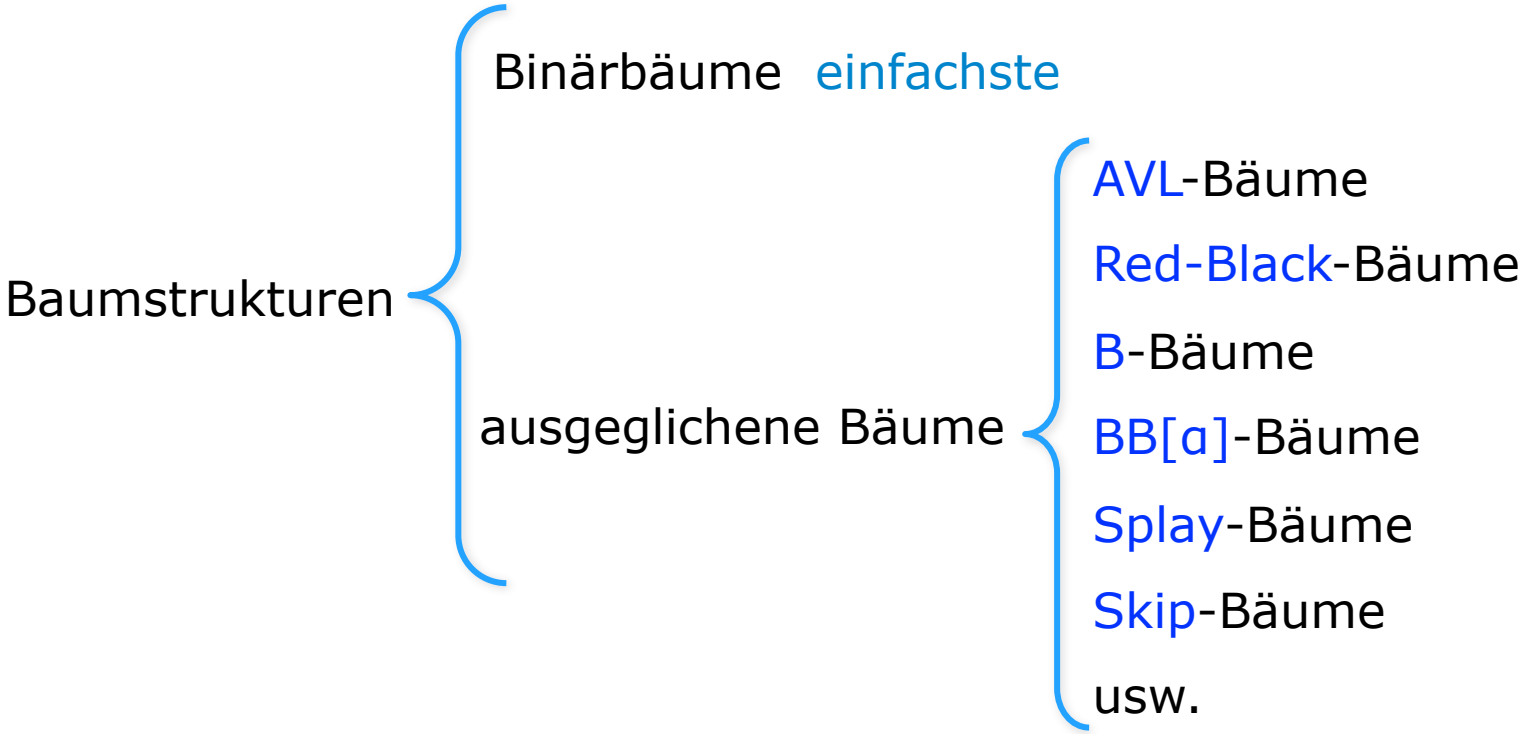
Höhe des Baumes

$$h(t) = 1 + \max(h(t_l), h(t_r))$$

Innere Pfadlänge des Baumes
(Summe der Tiefen aller inneren Knoten)

$$\pi(t) = \pi(t_l) + \pi(t_r) + |t| - 1$$

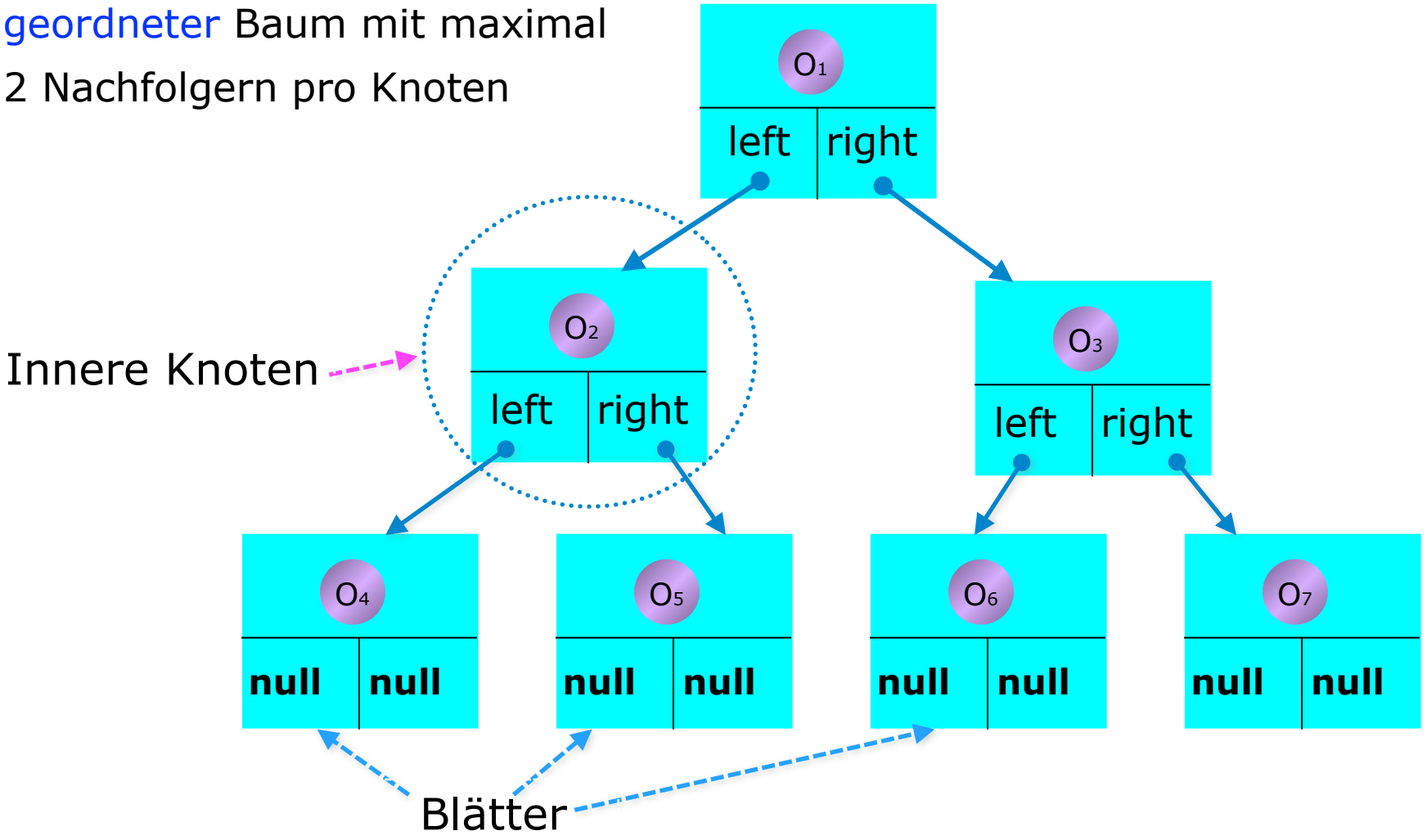
Bäume



Die wichtigste Voraussetzung für die effiziente Verwaltung von Datenmengen mit Hilfe von Bäumen ist, dass die Bäume balanciert sind.

Binäre Suchbäume

geordneter Baum mit maximal 2 Nachfolgern pro Knoten

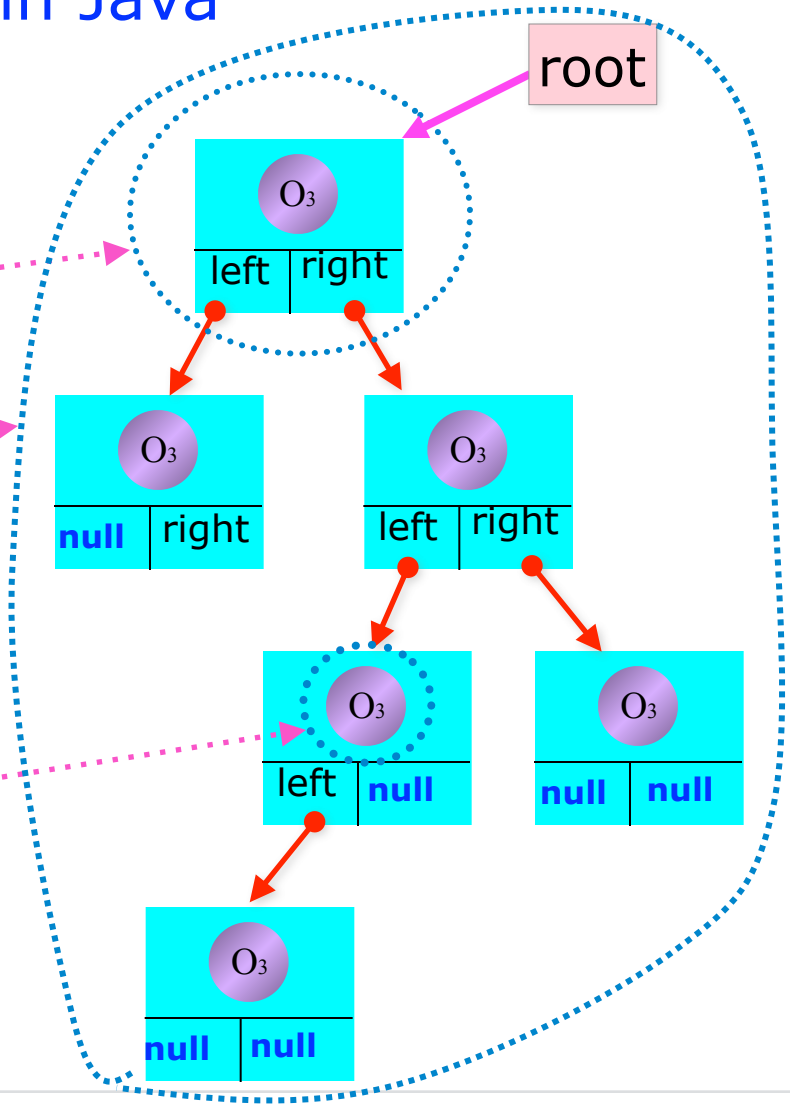


Wie können wir Binärbäume in Java implementieren?

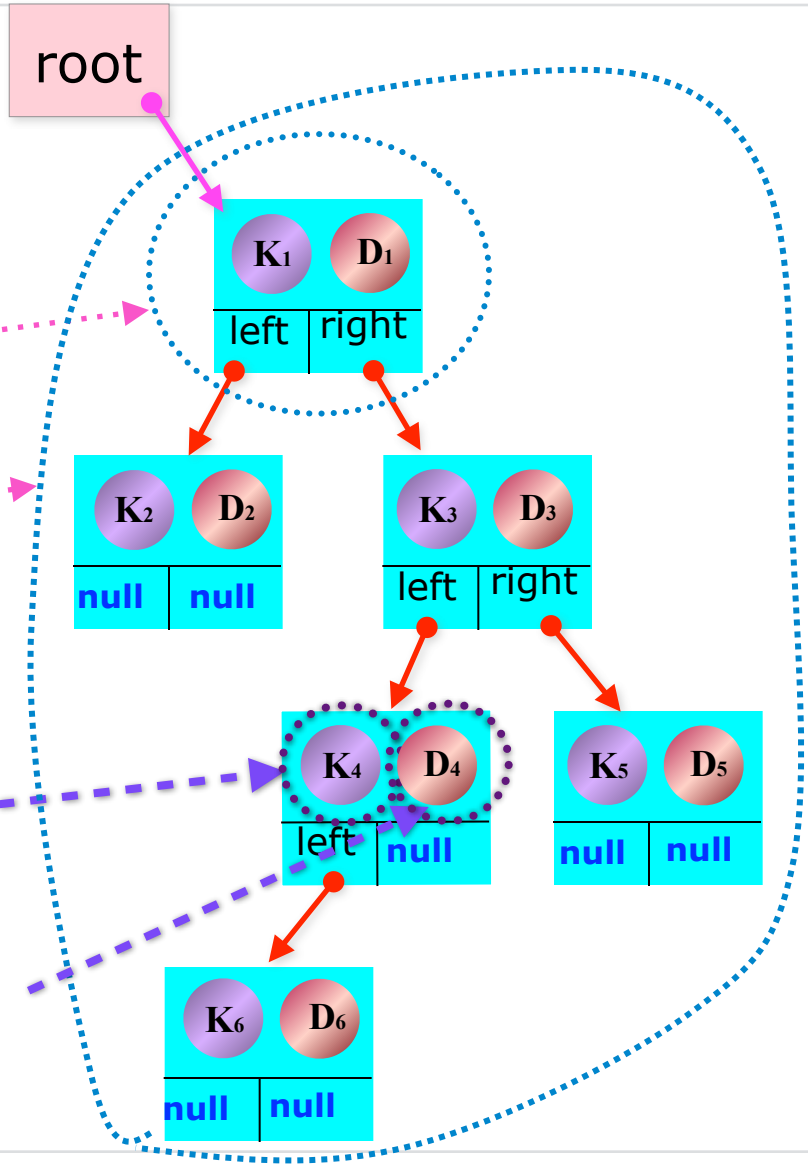
TreeNode-Klasse

BinLinkedTree-Klasse

Die Objekte werden nach einem Schlüssel einsortiert.



Beispiel:



TreeNode-Klasse

BinLinkedTree-Klasse

Die Objekte werden nach einem Schlüssel **K** einsortiert

Datenobjekt **D**, das zum Schlüssel verbunden ist.

Binäre Suchbäume

Sortierbare Schlüssel

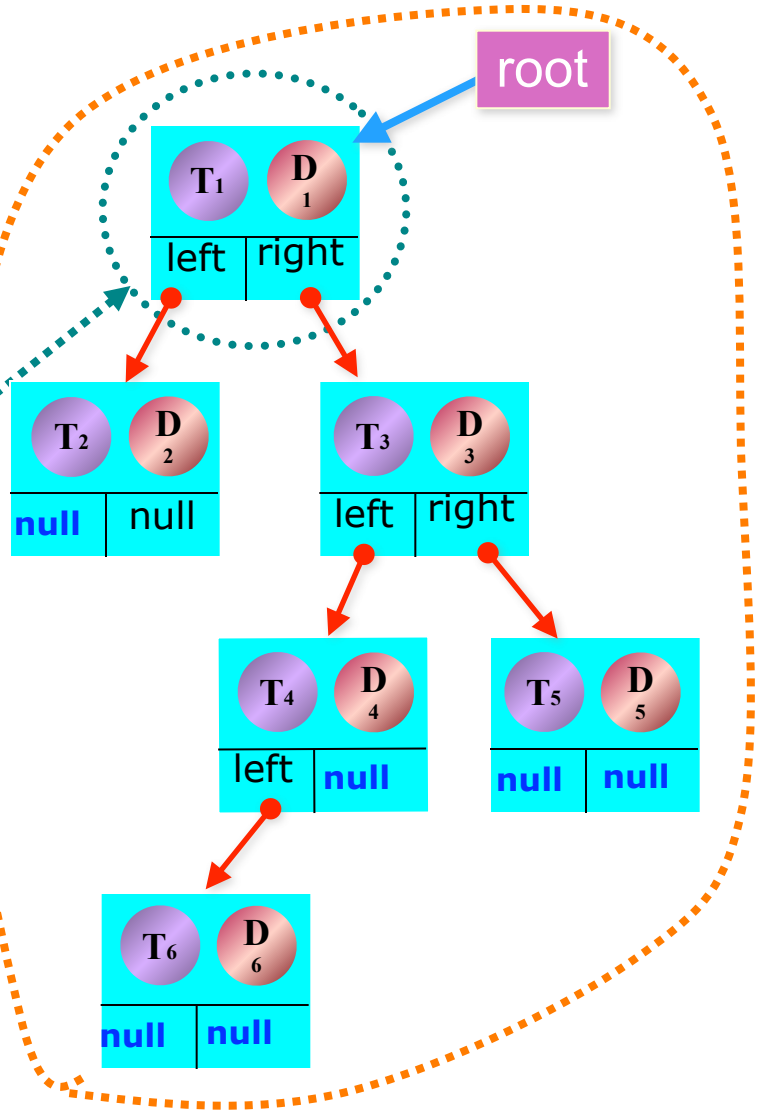
Daten

```
public class BinLinkedTree <T extends Comparable<T>, D>
                                        implements Iterable<T> {

    private TreeNode root;
    private int size;   // Anzahl der TreeNode-Objekte

    public BinLinkedTree() { // constructor
        root = null;
        size = 0;
    }
    public int size() {
        return size;
    }
    …
```

um for-each-Schleifen verwenden zu können

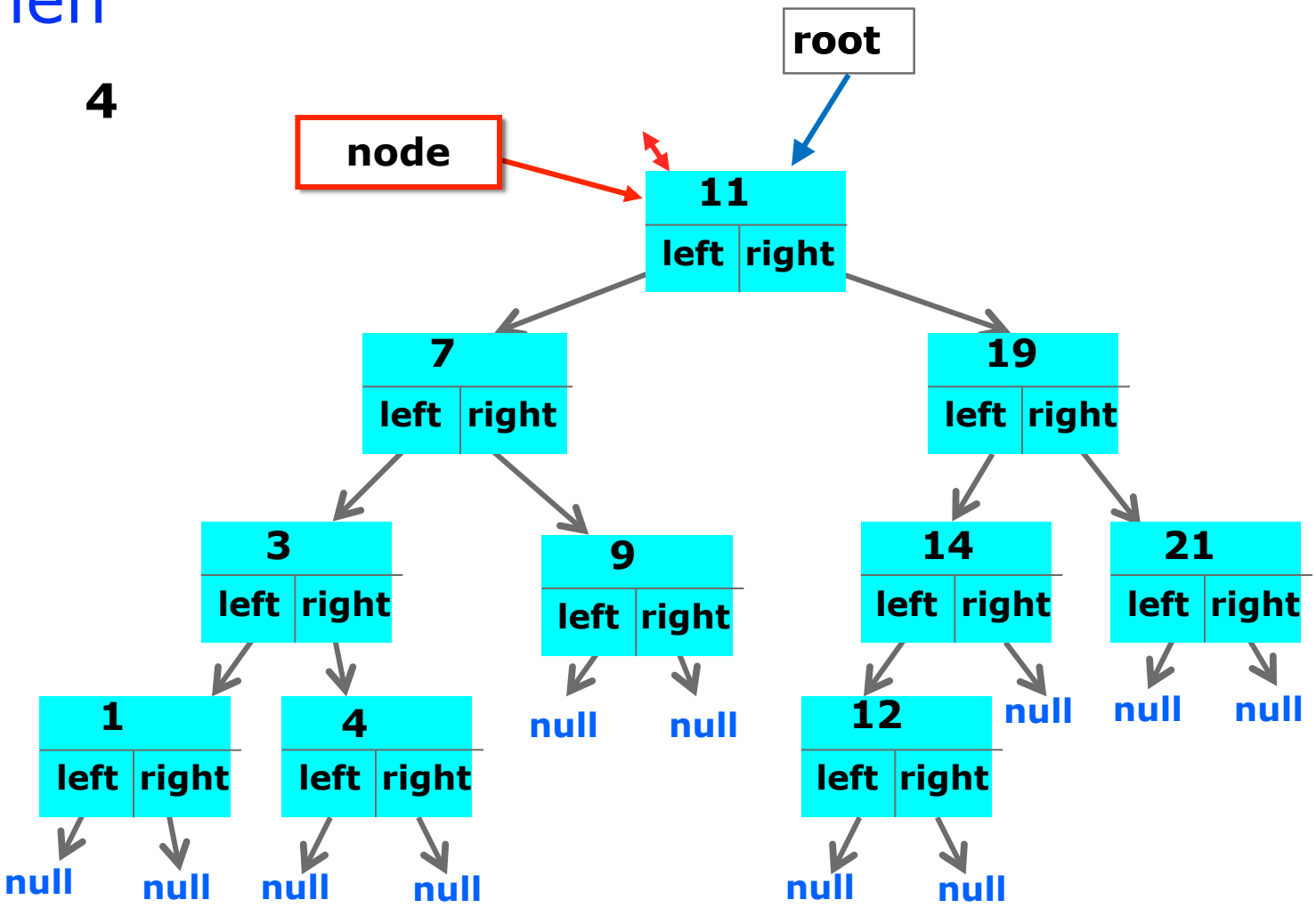
Binäre Suchbäume

```
public class BinLinkedTree ….
…
    class TreeNode {
        private T key;
        private D data;
        private TreeNode left;
        private TreeNode right;

        TreeNode (T key, D data) {
            this.key = key;
            this.data = data;
        }
    } // end of class TreeNode
…
```

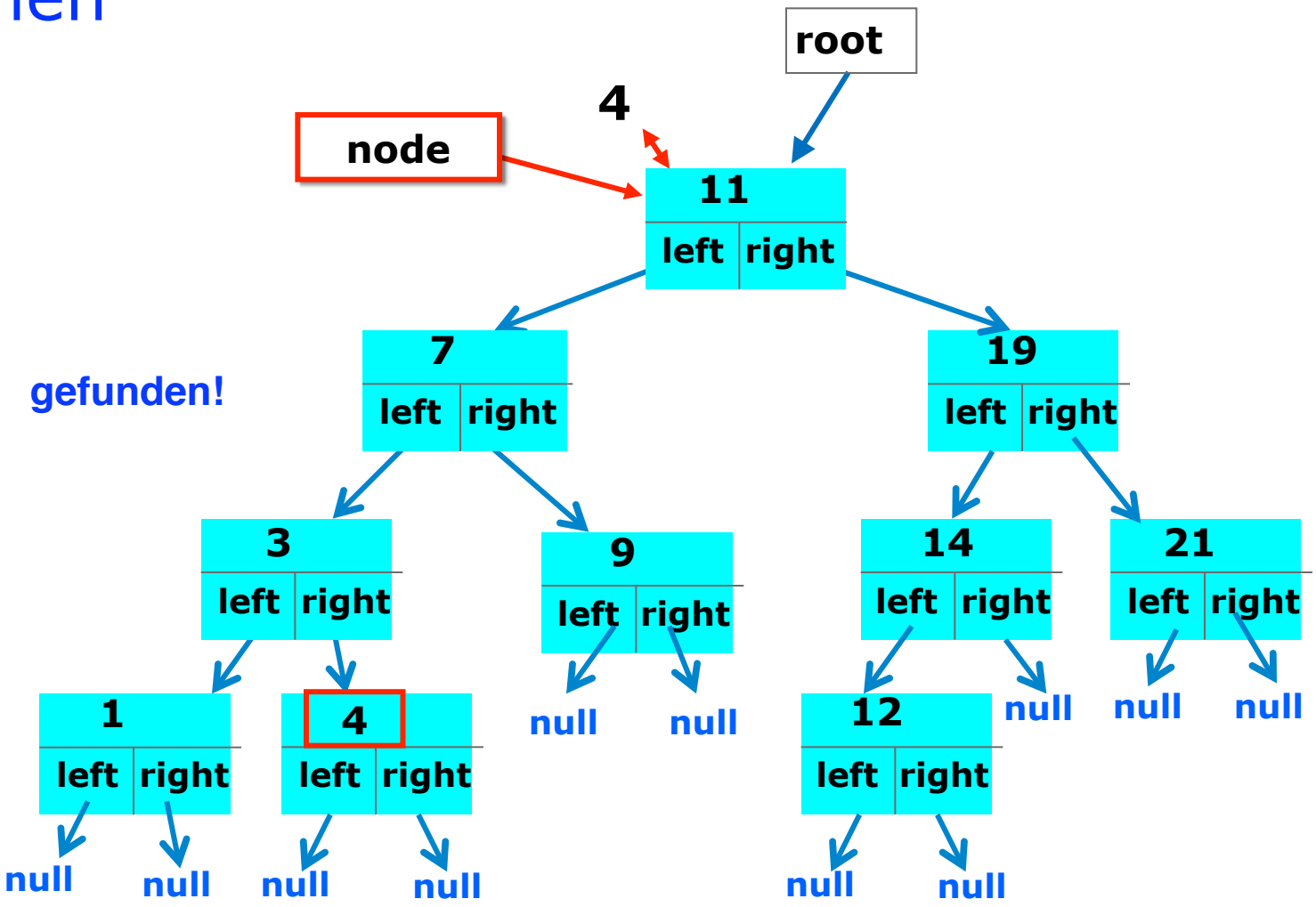


Suchen

4

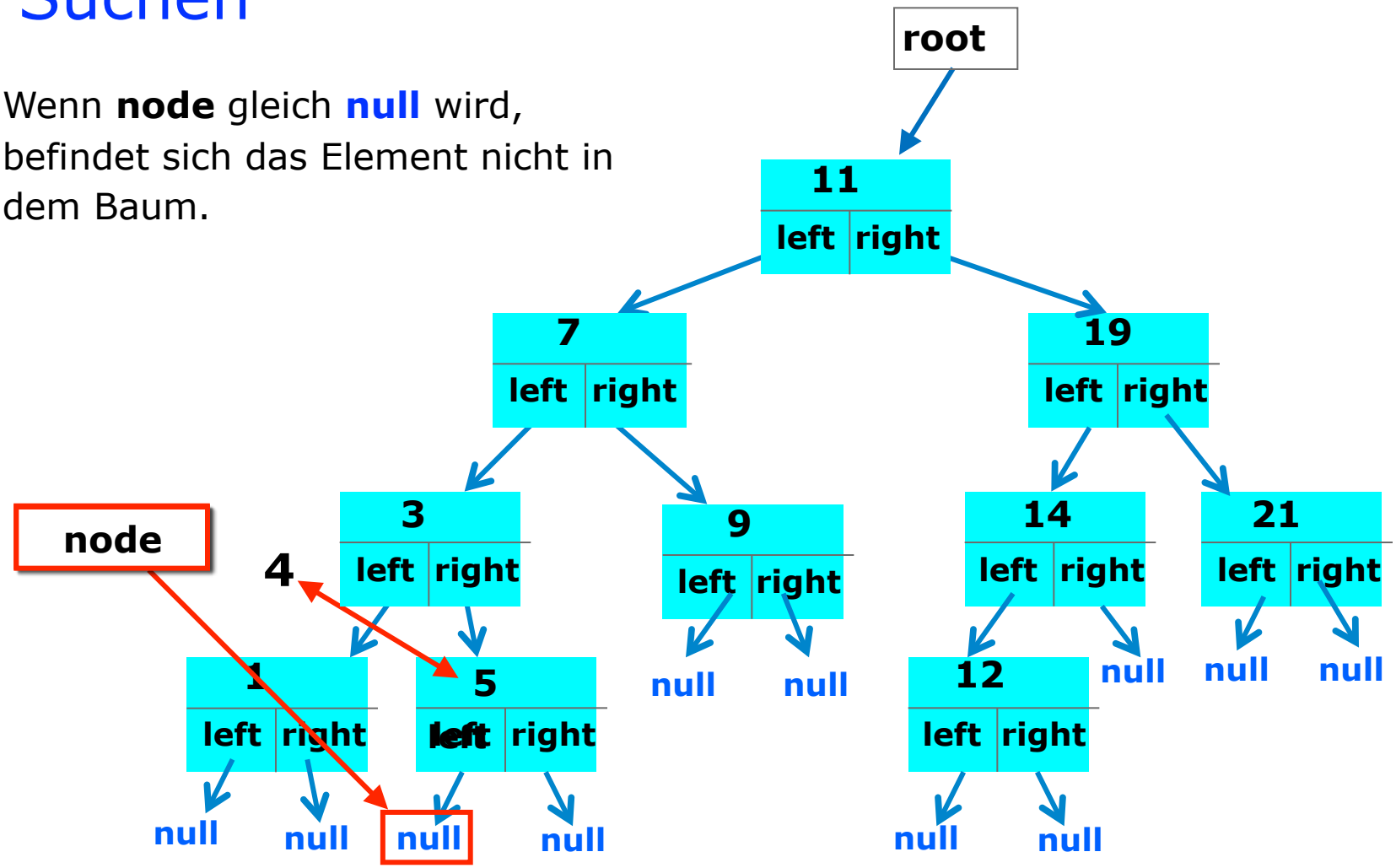


Suchen



Suchen

Wenn **node** gleich **null** wird, befindet sich das Element nicht in dem Baum.



Binäre Suchbäume

```
 ...
public boolean contains(T key) {
       return getData( key ) != null;
}

public D getData(T key) {
       TreeNode node = root;
       while (node != null) {
             int compare = key.compareTo(node.key);
             if (compare < 0)
                   node = node.left;
             else if (compare > 0)
                   node = node.right;
             else
                   return node.data;
       }
       return null;
}
...
```

Ein Schlüssel wird gesucht.

Gibt die Daten, die mit einem Schlüssel verbunden sind, zurück oder null, wenn der Schlüssel nicht vorhanden ist.

Binäre Suchbäume

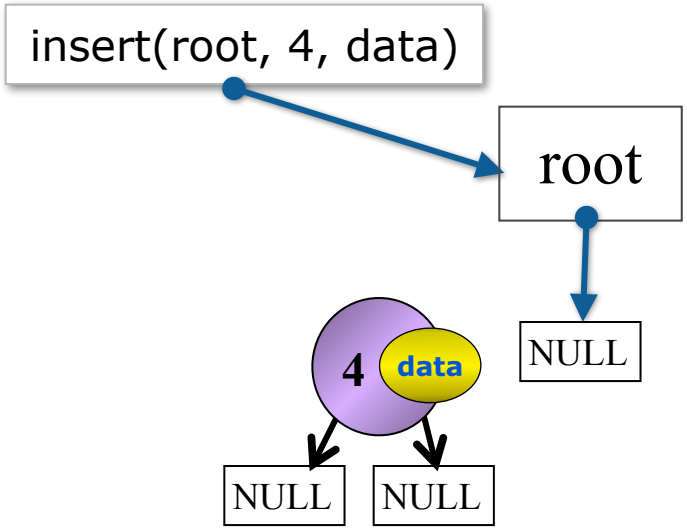
```
public class BinLinkedTree <T extends Comparable<T>, D>
                                        implements Iterable<T>{
    …
    public void store(T key, D data) {
            root = insert( root, key, data );
    }
    private TreeNode insert(TreeNode node, T key, D data) {
            if (node == null) {
                    size++;
                    return new TreeNode(key, data);
            }
            int compare = key.compareTo(node.key);
            if (compare < 0)
                    node.left  = insert(node.left,  key, data);
            else if (compare > 0)
                    node.right = insert(node.right, key, data);
            else
                    node.data = data;
            return node;
    }
    …
```

Ein Schlüssel und das damit verbundene Daten-Objekt werden eingegeben.

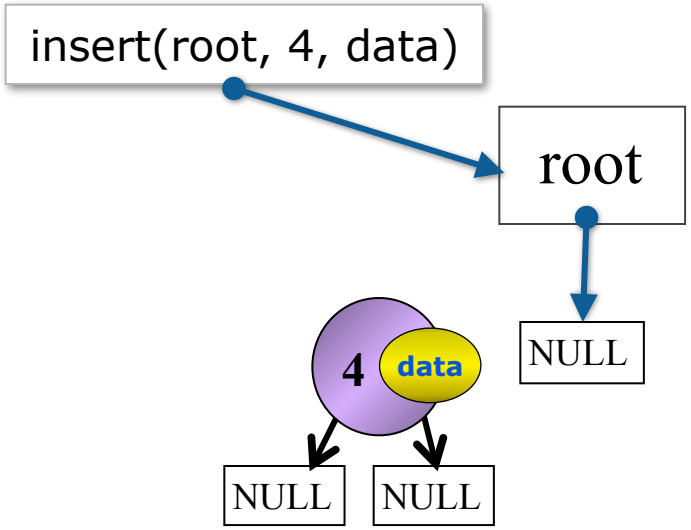
Ein neues Objekt wird nach seinem Schlüssel in einem Blatt einsortiert.

Wenn der Schlüssel bereits existiert, werden die Daten überschrieben.

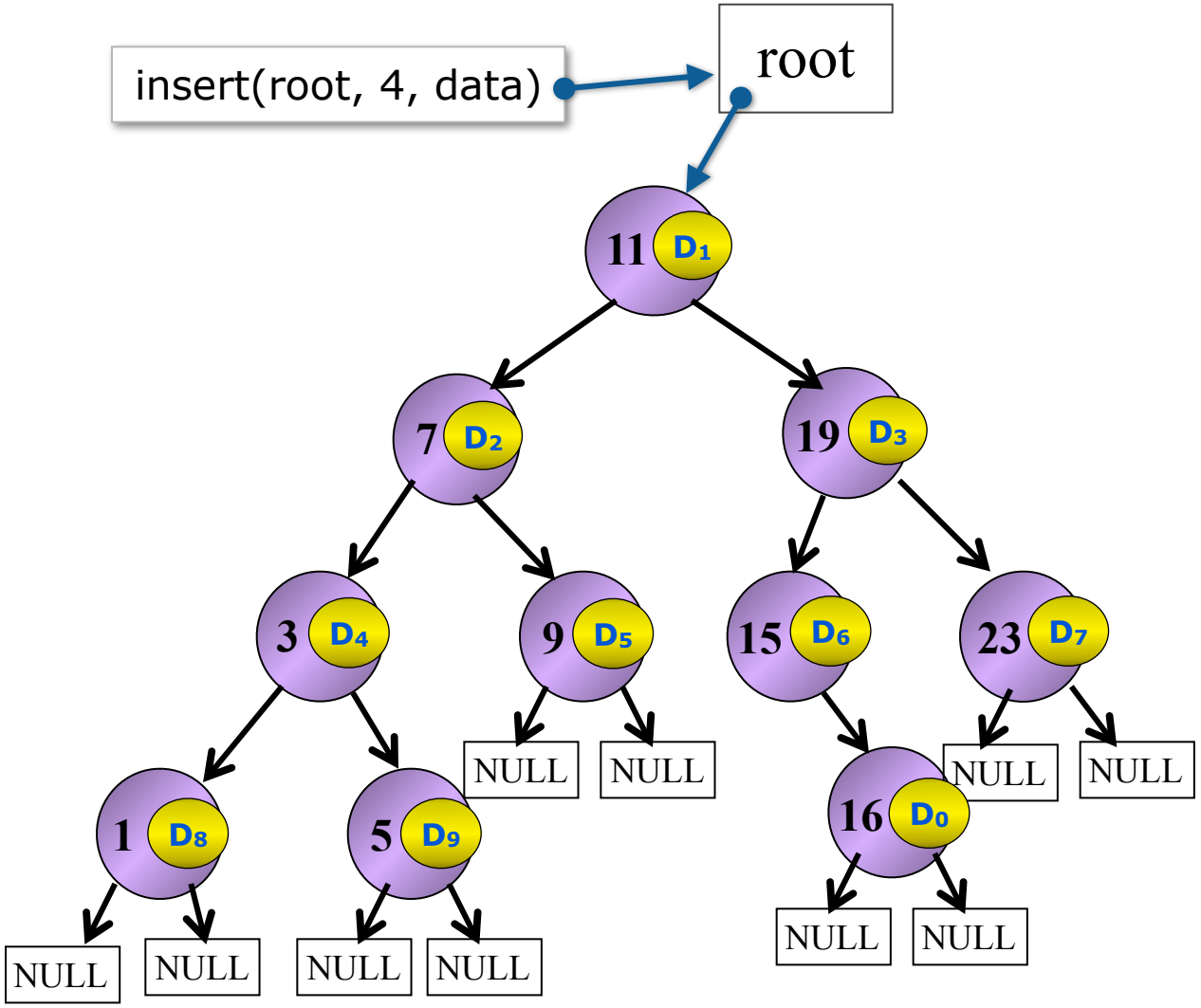
Einfügen



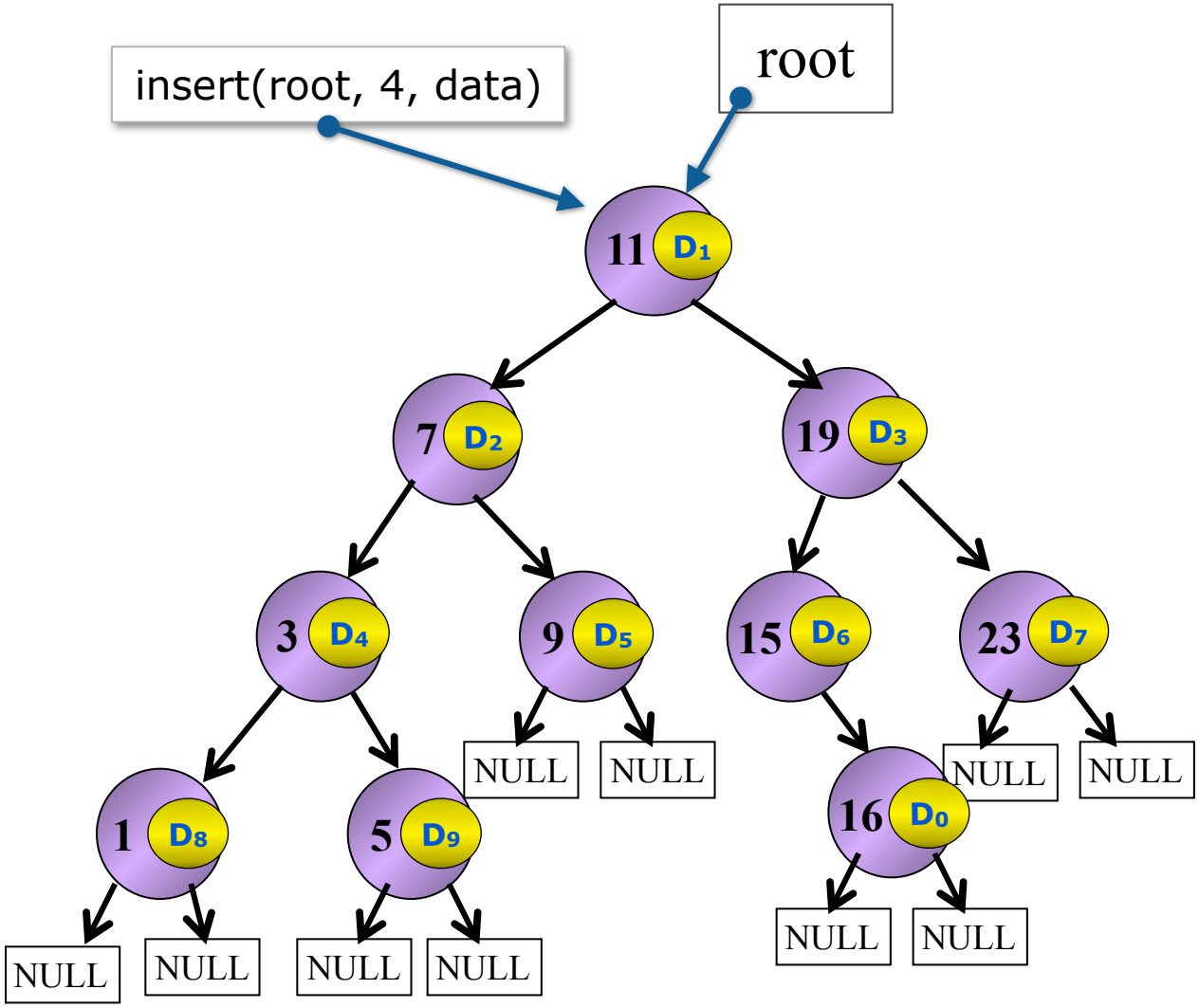
Einfügen



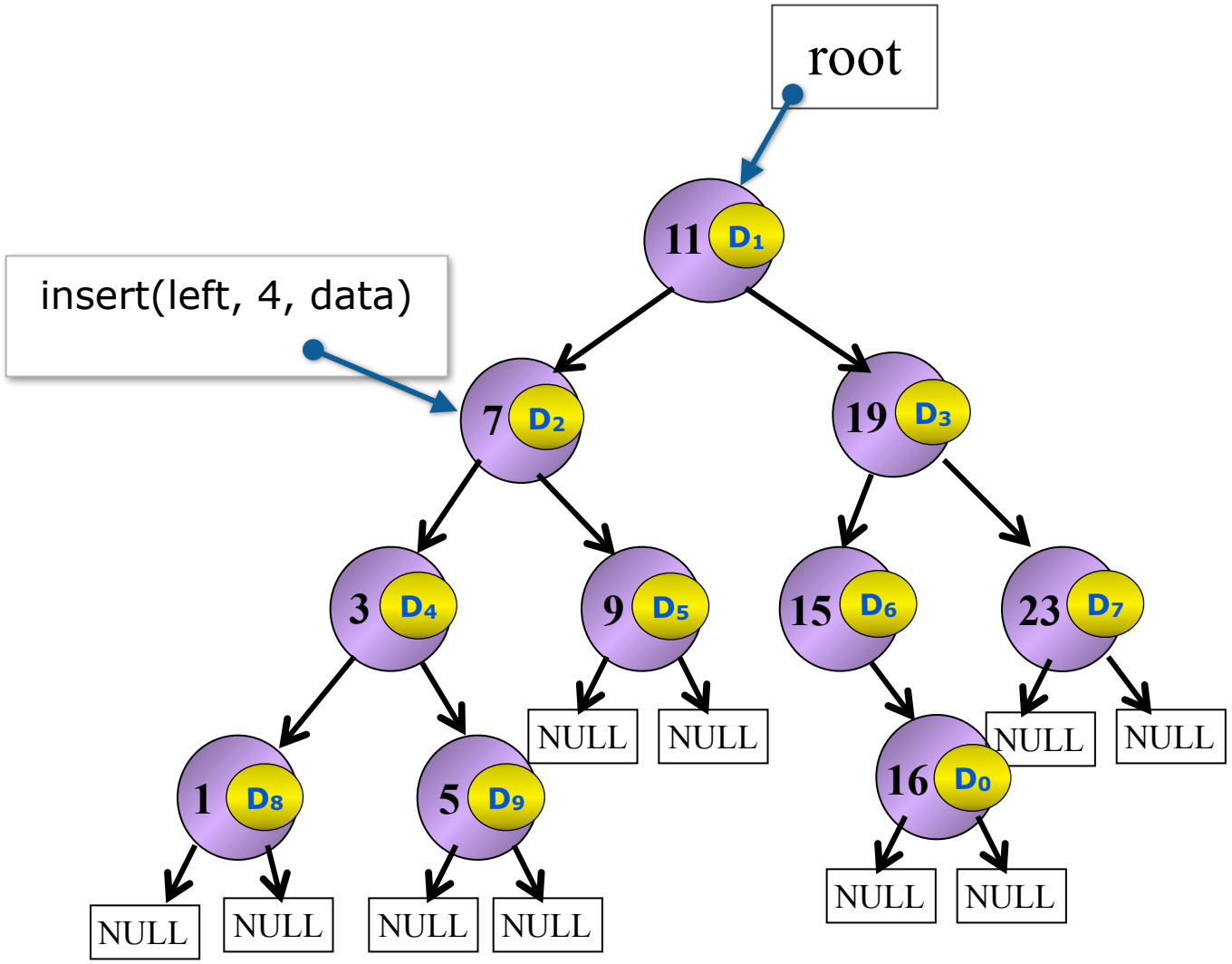
Einfügen



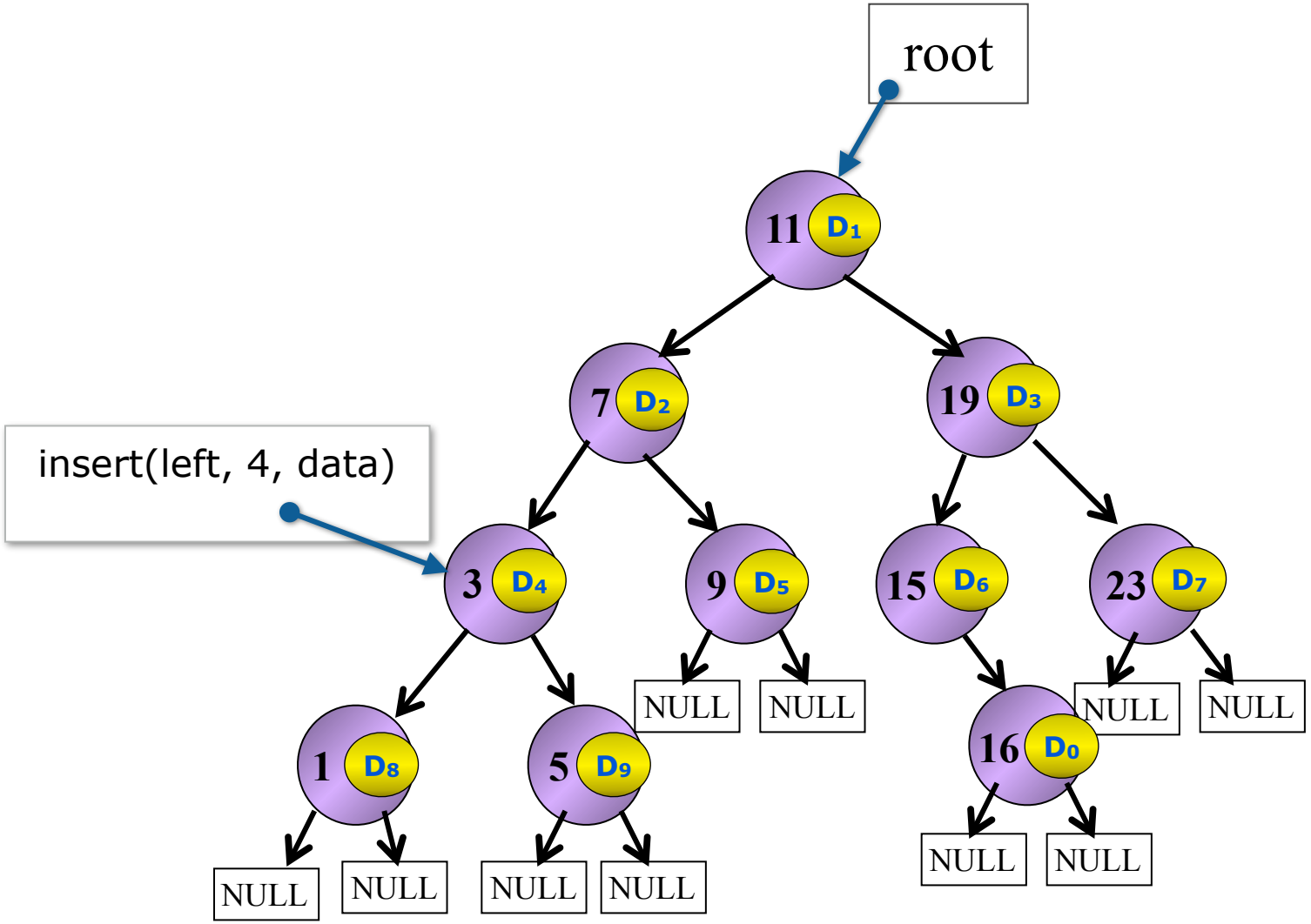
Einfügen



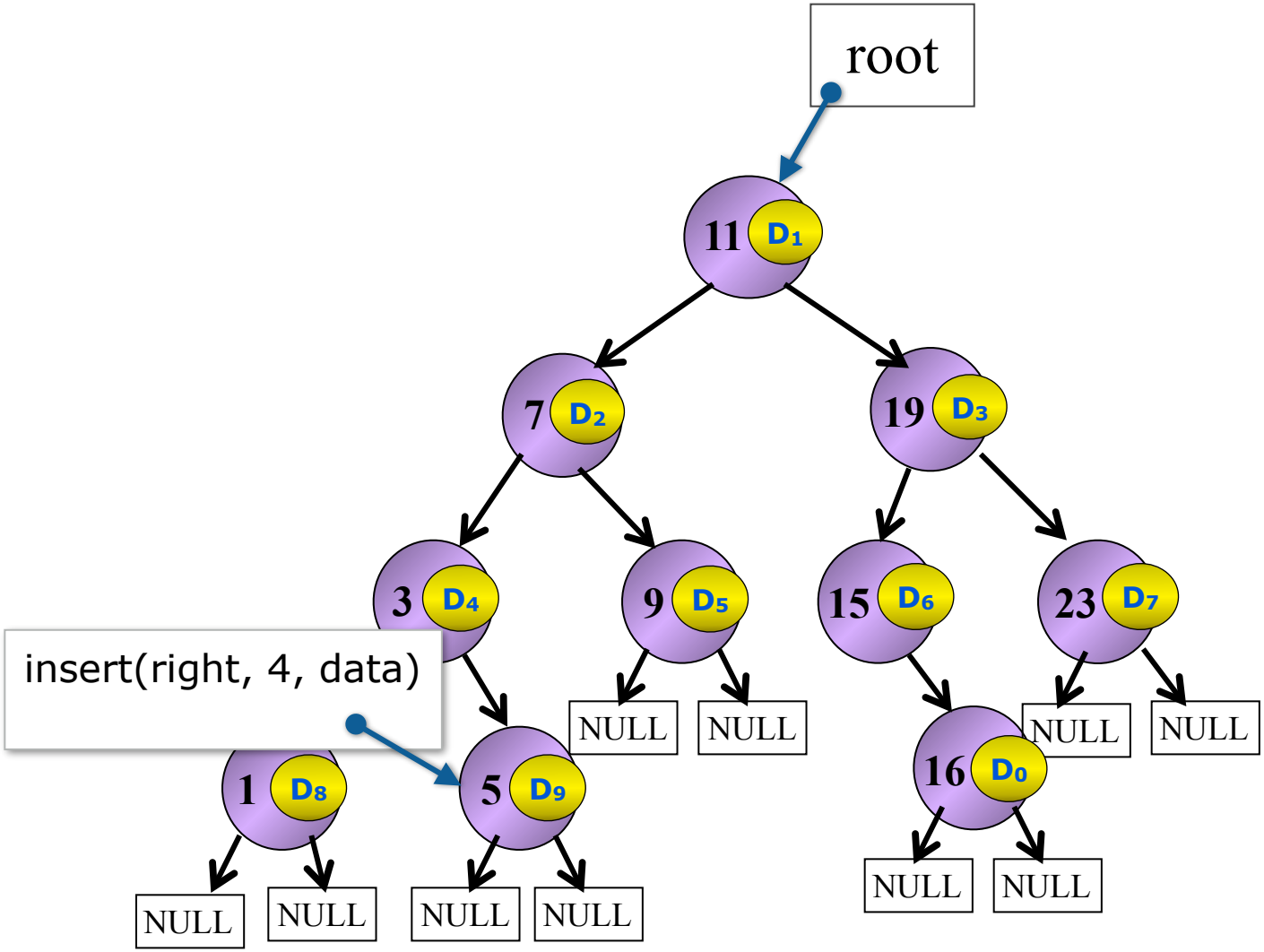
Einfügen



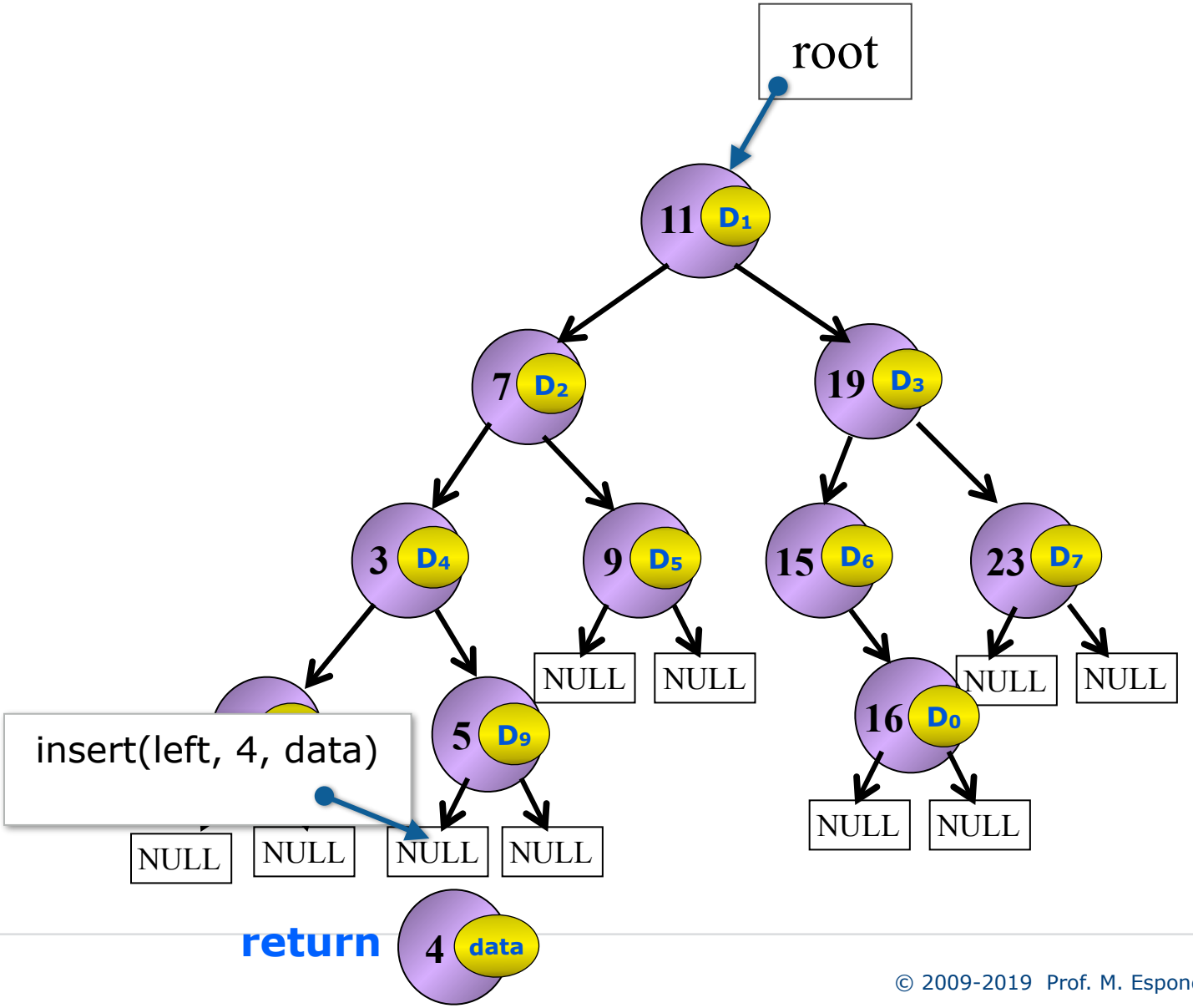
Einfügen



Einfügen

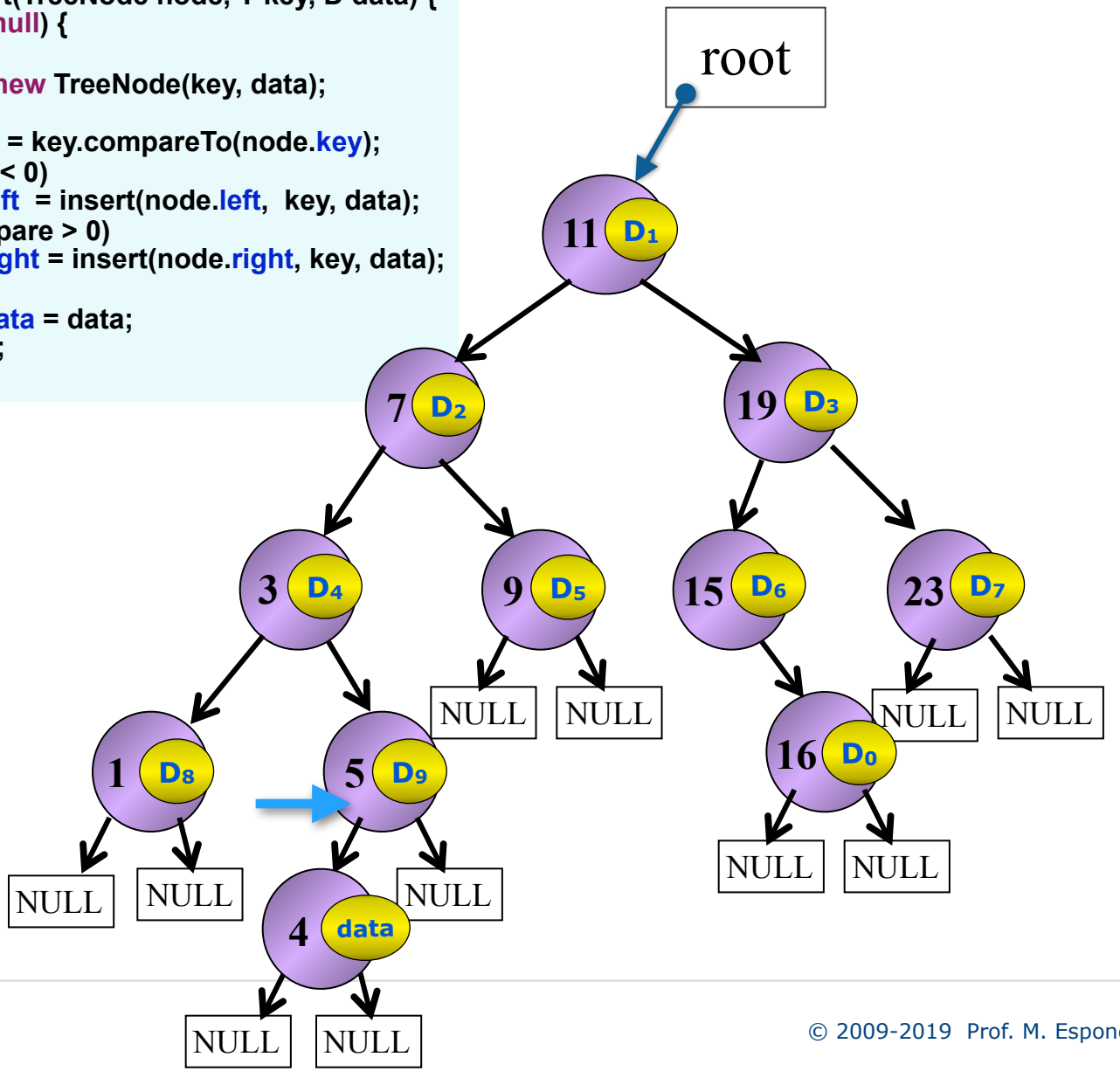


Einfügen



Einfügen (rekursiv)


```
private TreeNode insert(TreeNode node, T key, D data) {
        if (node == null) {
            size++;
            return new TreeNode(key, data);
        }
        int compare = key.compareTo(node.key);
        if (compare < 0)
            node.left  = insert(node.left,  key, data);
        else if (compare > 0)
            node.right = insert(node.right, key, data);
        else
            node.data = data;
        return node;
}
```



Traversierung binärer Bäume

Baumtraversierung bedeutet, alle Knoten des Baumes in einer bestimmten Reihenfolge zu besuchen.

Preorder: **Wurzel – linker Unterbaum – rechter Unterbaum**

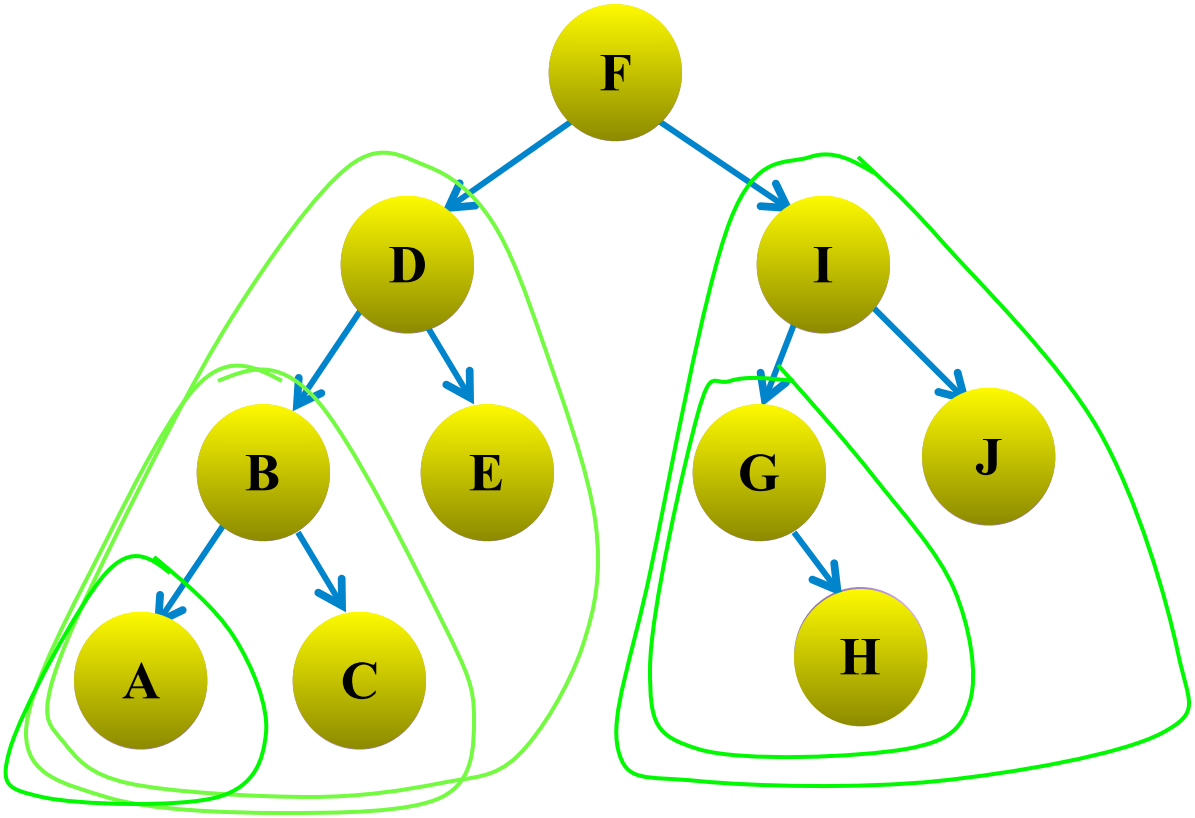
Inorder: **linker Unterbaum - Wurzel – rechter Unterbaum**

Postorder: **linker Unterbaum – rechter Unterbaum - Wurzel**

Levelorder: **von oben nach unten in jeder Ebene von links nach rechts**

Traversierung binärer Bäume

Inorder Linker Unterbaum - Wurzel - Rechter Unterbaum



A B C D E F G H I J

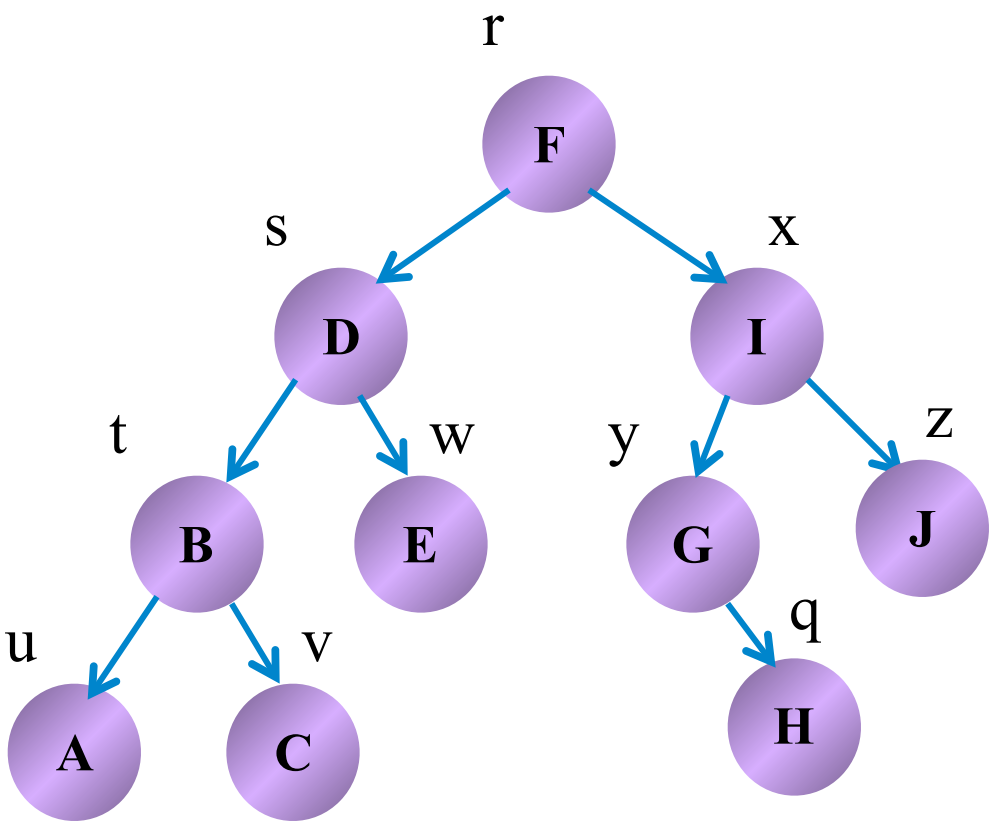
Implementierung einer **Iterator**-Klasse als Innere Klasse

```
public class BinLinkedTree <T extends Comparable<T>, D>
                                        implements Iterable<T> {
    ...
    public InorderIterator  iterator() {
            return new InorderIterator();
    }
    private class InorderIterator implements Iterator<T> {
      ...
    }
}
```

InorderIterator-Klasse, die den Baum in sortierter Reihenfolge durchläuft.

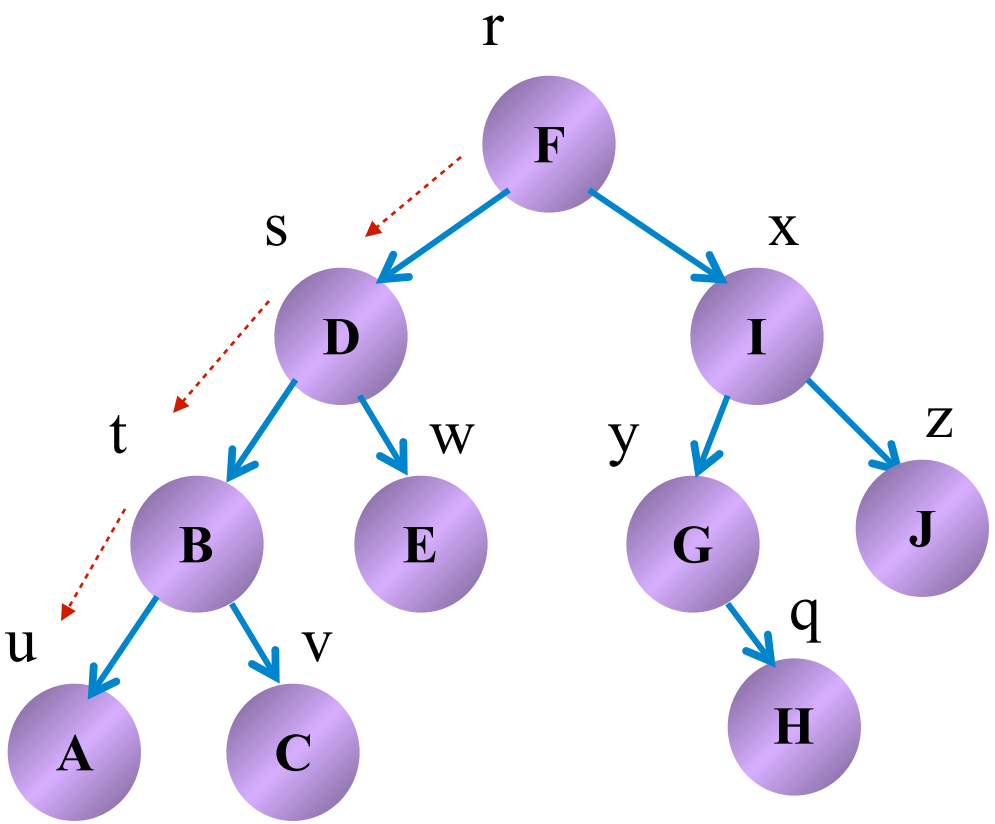
Iterative Implementierung

Stapel



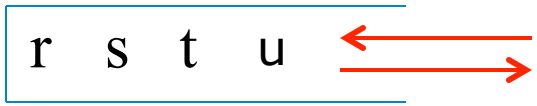
Iterative Implementierung

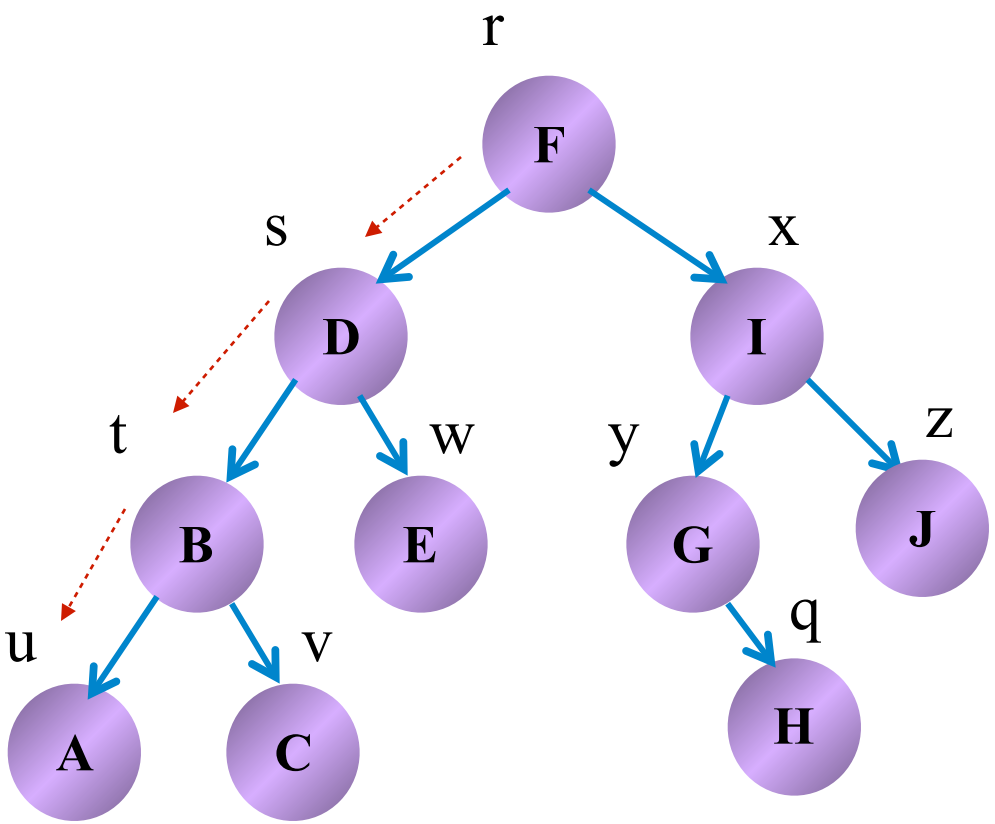
Stapel



Iterative Implementierung

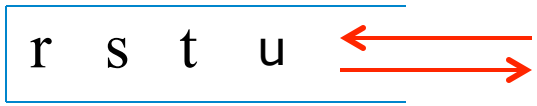
Stapel

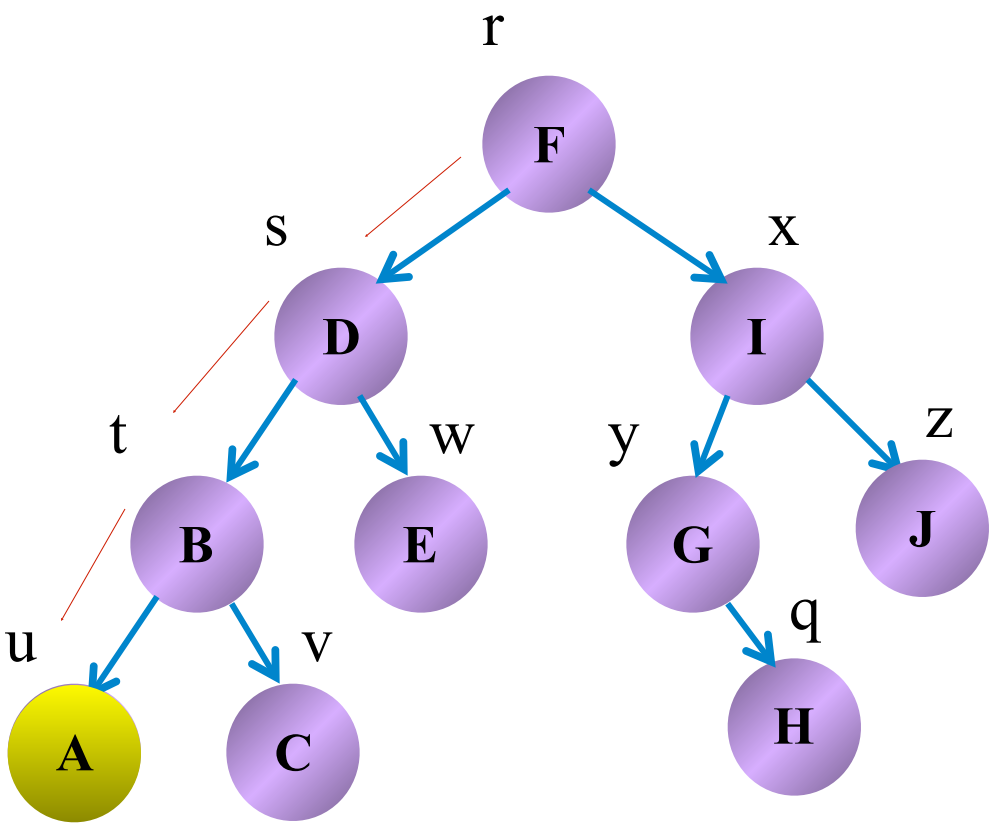




Iterative Implementierung

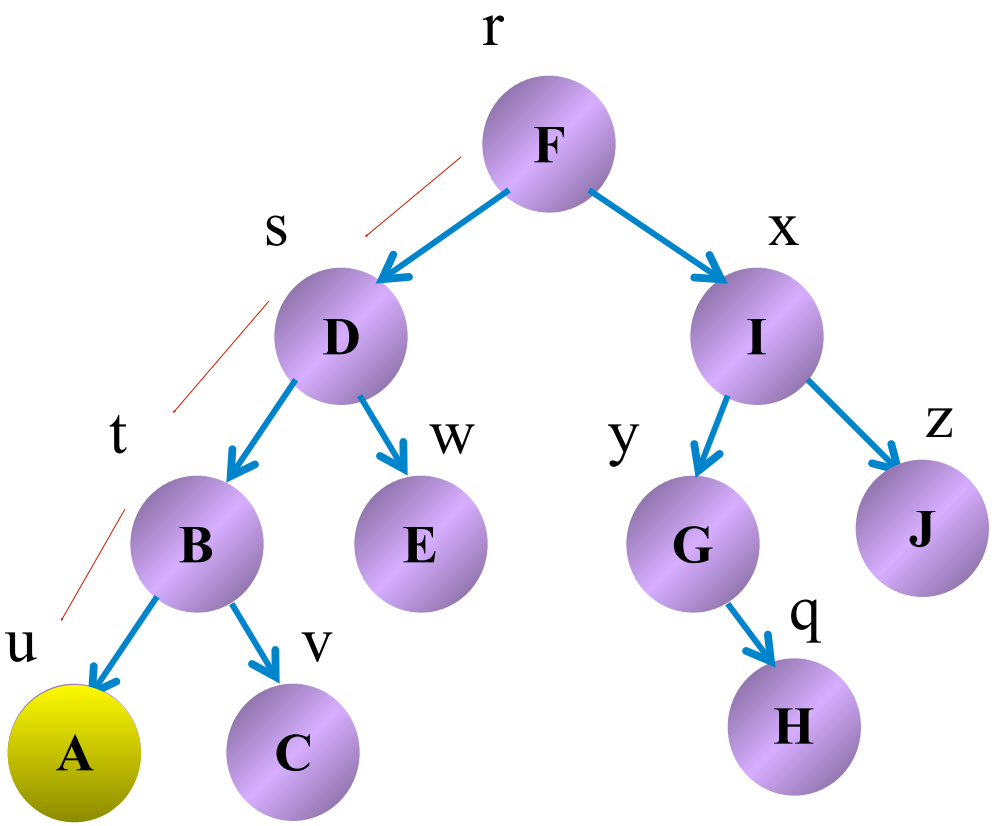
Stapel





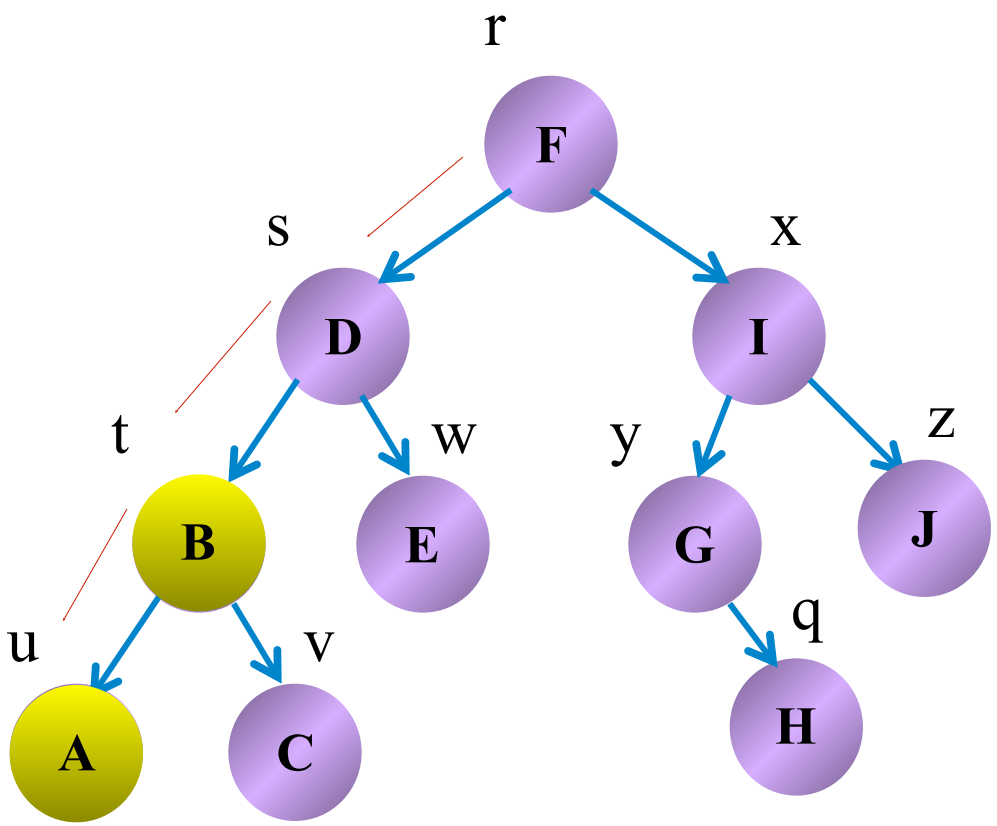
Iterative Implementierung

Stapel



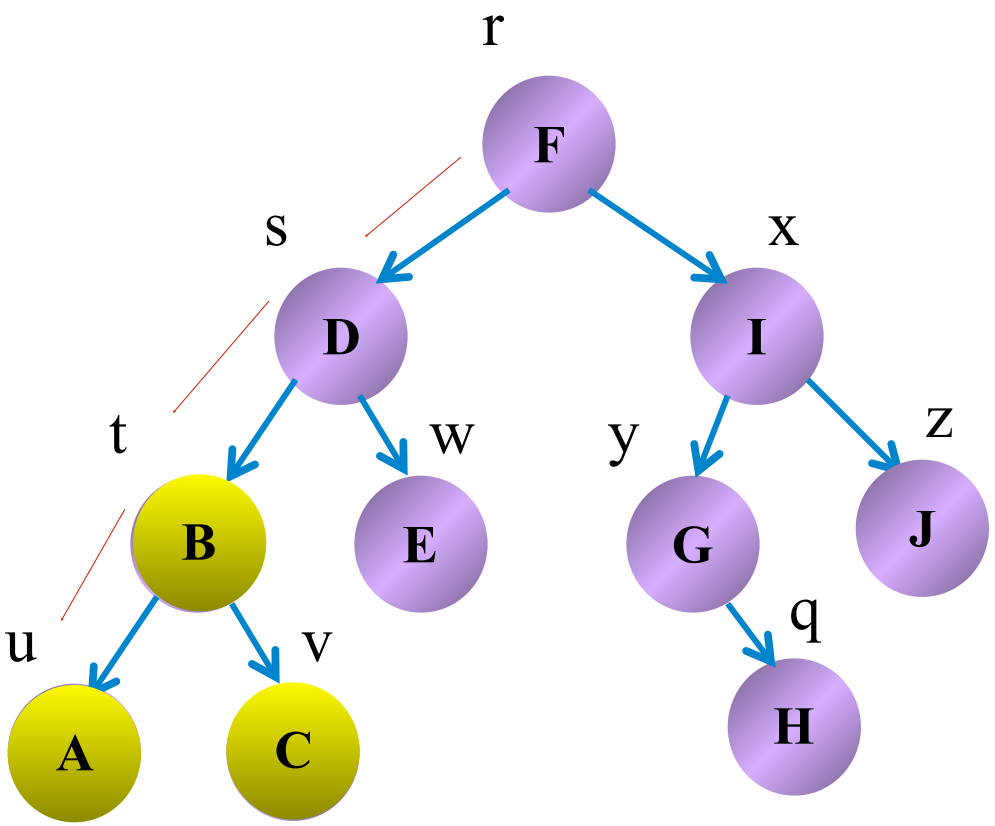
Iterative Implementierung

Stapel



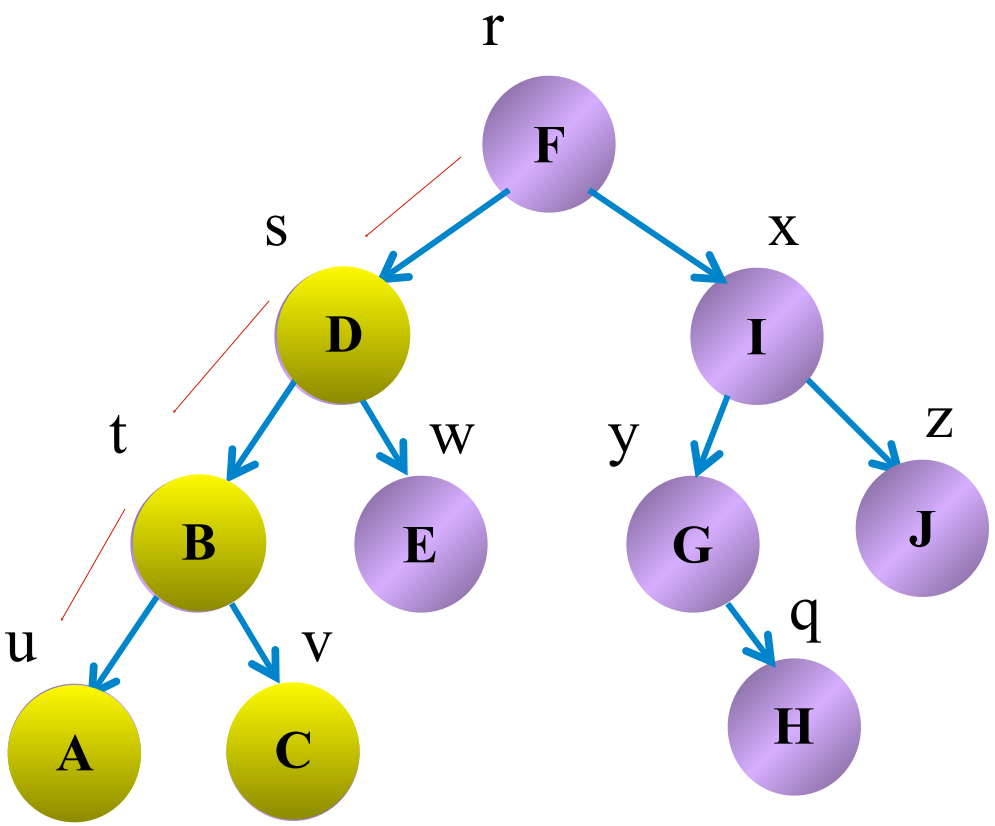
Iterative Implementierung

Stapel



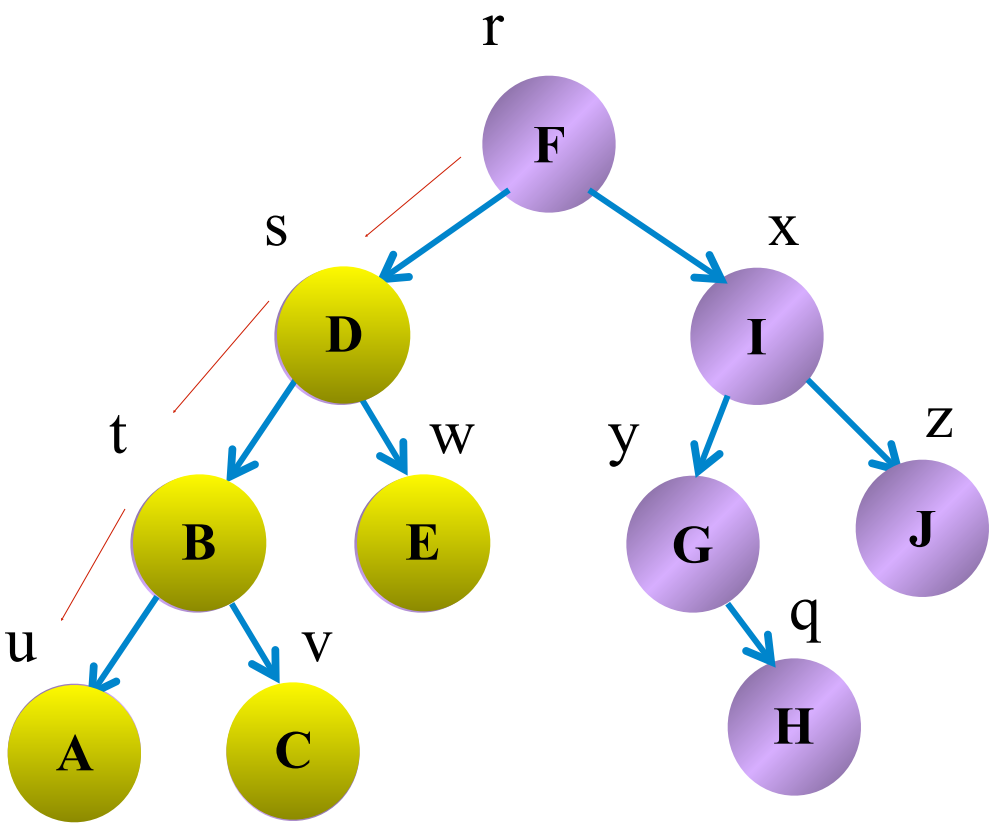
Iterative Implementierung

Stapel



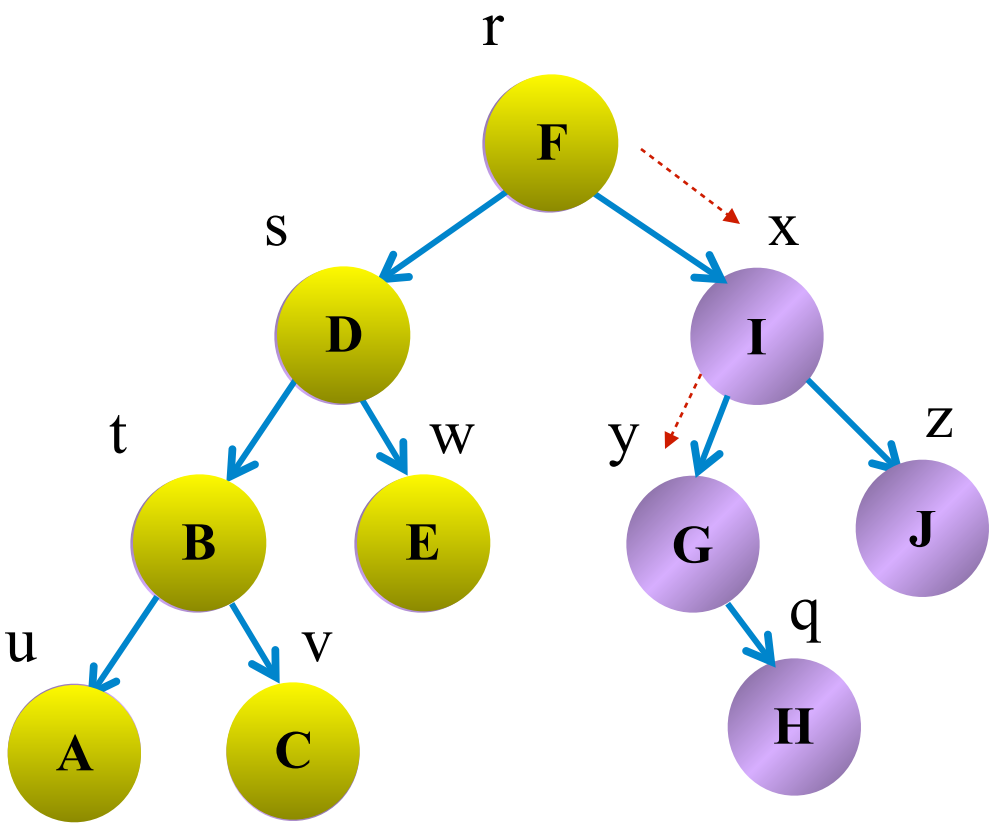
Iterative Implementierung

Stapel



Iterative Implementierung

Stapel



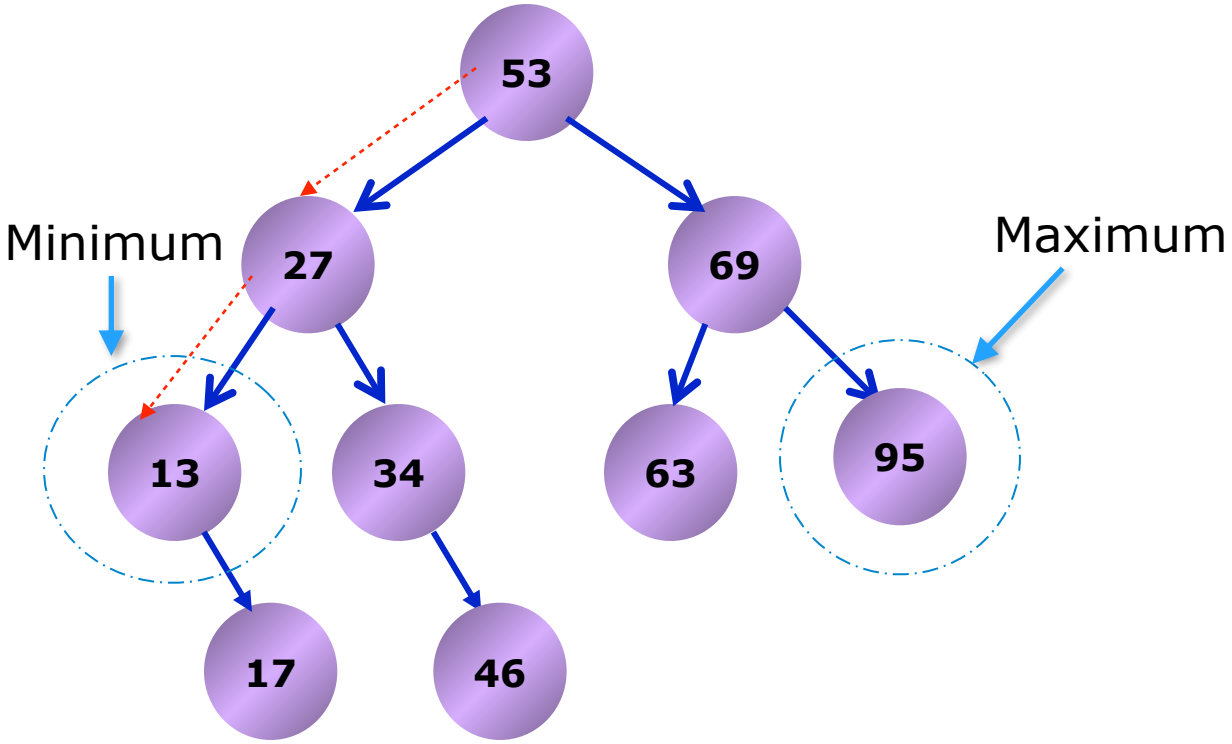
Implementierung einer **Iterator**-Klasse als Innere Klasse

```
. . .
private class InorderIterator implements Iterator<T> {
      private Stack<TreeNode> stack = new Stack<TreeNode>();
      InorderIterator() { pushLeftTree(root); }
      public boolean hasNext()  { return !stack.isEmpty(); }
      public T next() {
            if (!hasNext()) throw new NoSuchElementException();
            TreeNode node = stack.pop();
            pushLeftTree(node.right);
            return node.key;
      }
      public void pushLeftTree(TreeNode node) {
             while (node != null) {
                    stack.push(node);
                    node = node.left;
             }
      }
      public void remove() { throw new UnsupportedOperationException();}
}
. . .
```

```
public class NoSuchElementException
                    extends RuntimeException {
...}
```

```
. . .
public static void main(String[] args) {
    BinLinkedTree<Integer, String> st = new BinLinkedTree<Integer, String>();

    st.store(43901, "Peter Meyer"  );
    st.store(43021, "Nils Meyer"  );
    st.store(43002, "Andre Meyer"  );
    st.store(43101, "Hans Meyer"  );
    st.store(43000, "Joachim Meyer"  );
    st.store(43501, "Carl Meyer"  );

    for (Iterator<Integer> iter = st.iterator(); iter.hasNext(); )
            System.out.println(iter.next());

    System.out.println("size = " + st.size());

    for (Integer s : st)  {
            System.out.println(s);
    }
}
. . .
```

Minimum und Maximum



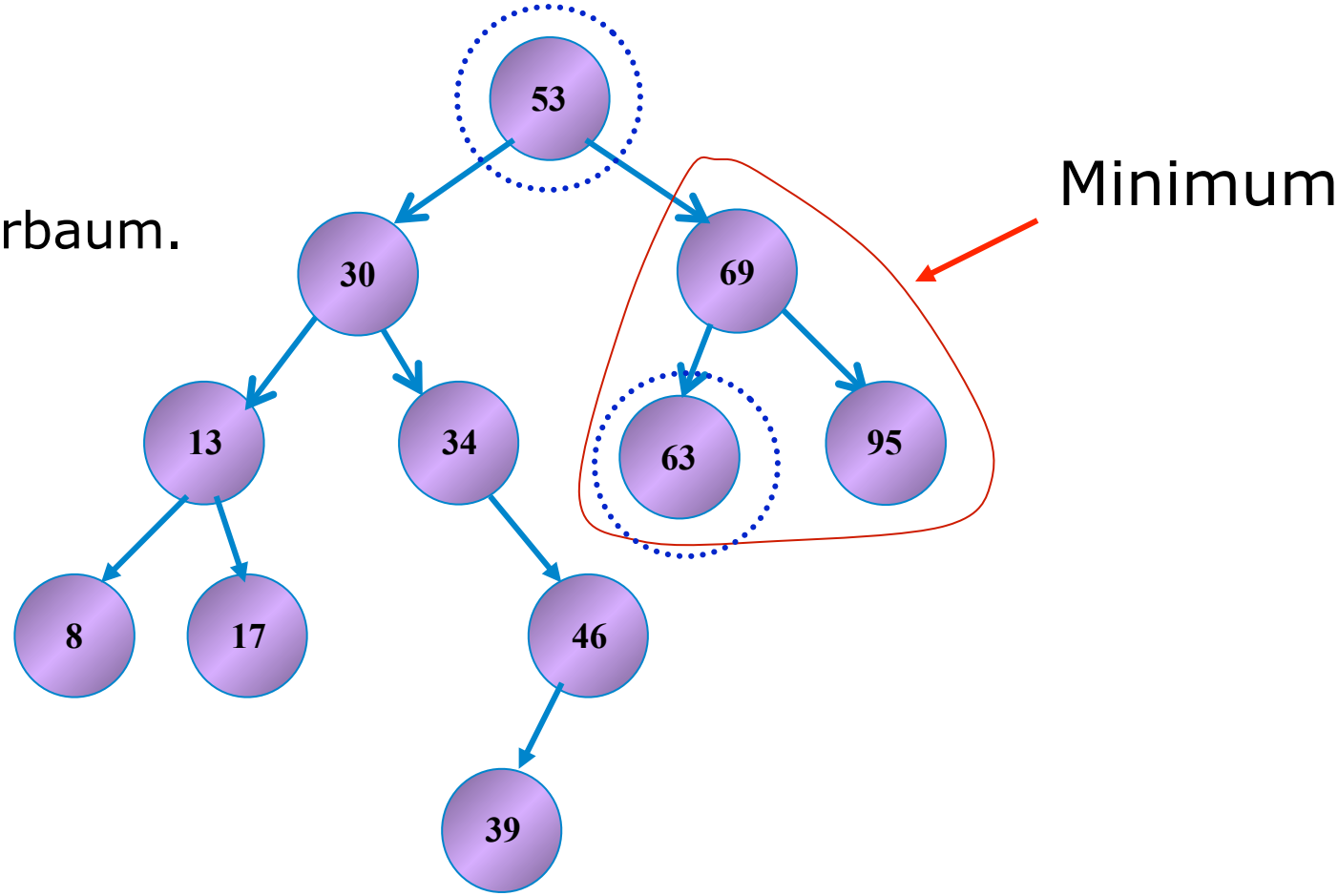
Der erste Knoten, der keine linken Kinder mehr hat, beinhaltet das kleinste Element.

Nachfolger

Freie Universität Berlin

1. Fall

Es gibt einen
rechten Unterbaum.

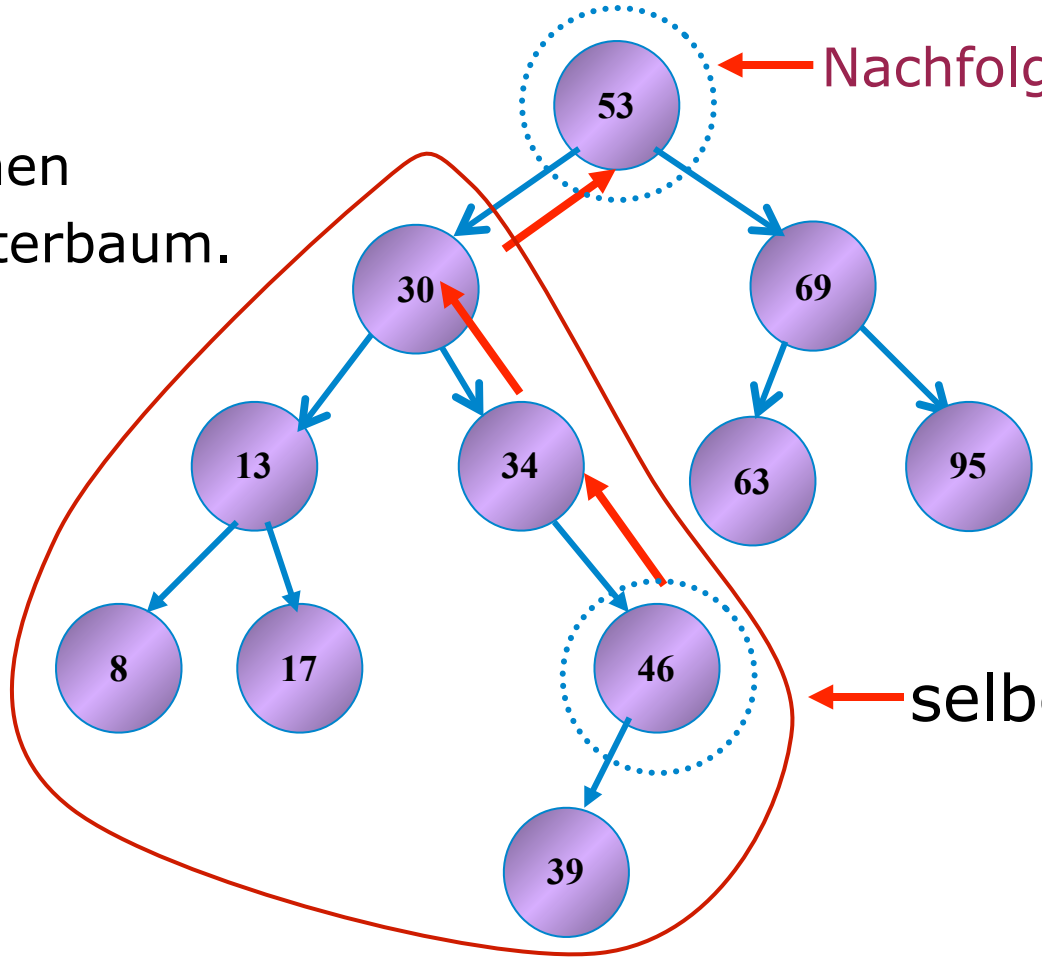


Nachfolger

Freie Universität Berlin

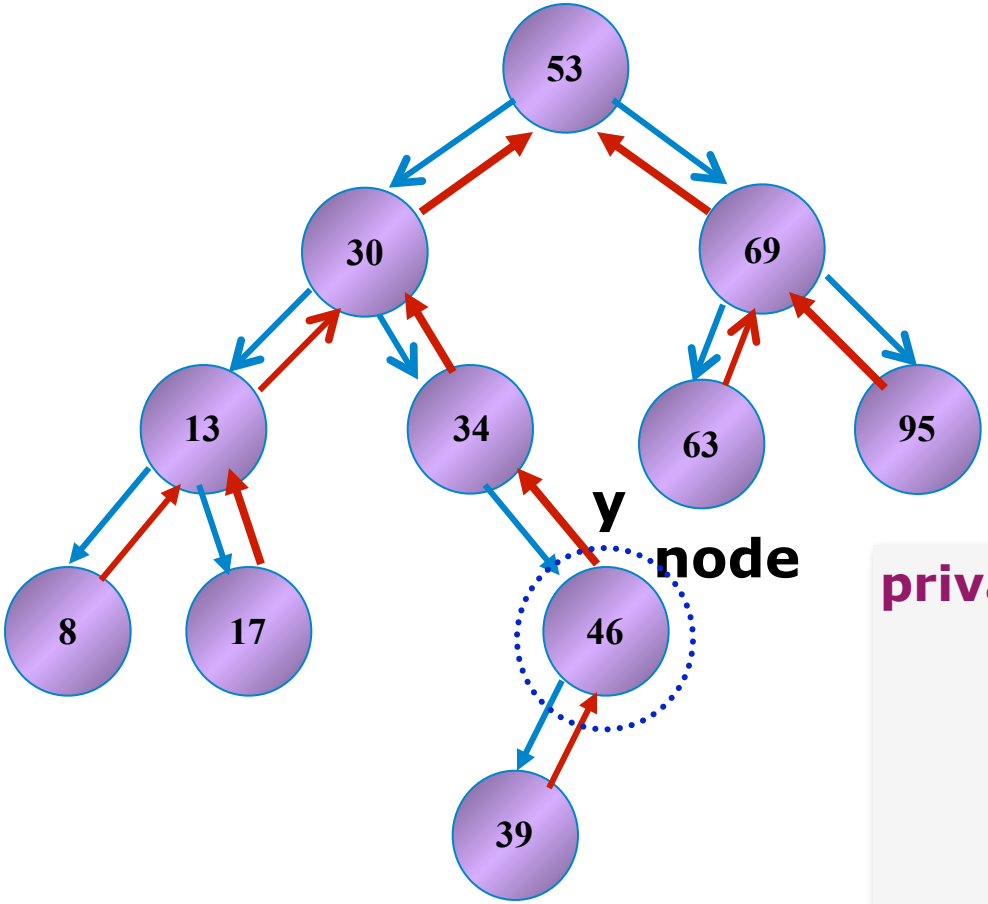
2. Fall

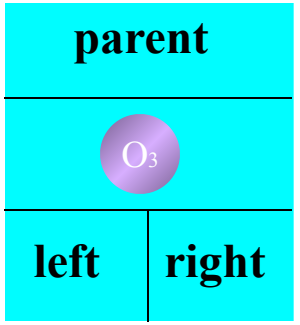
Es gibt keinen
rechten Unterbaum.



Wie können wir nach oben laufen?

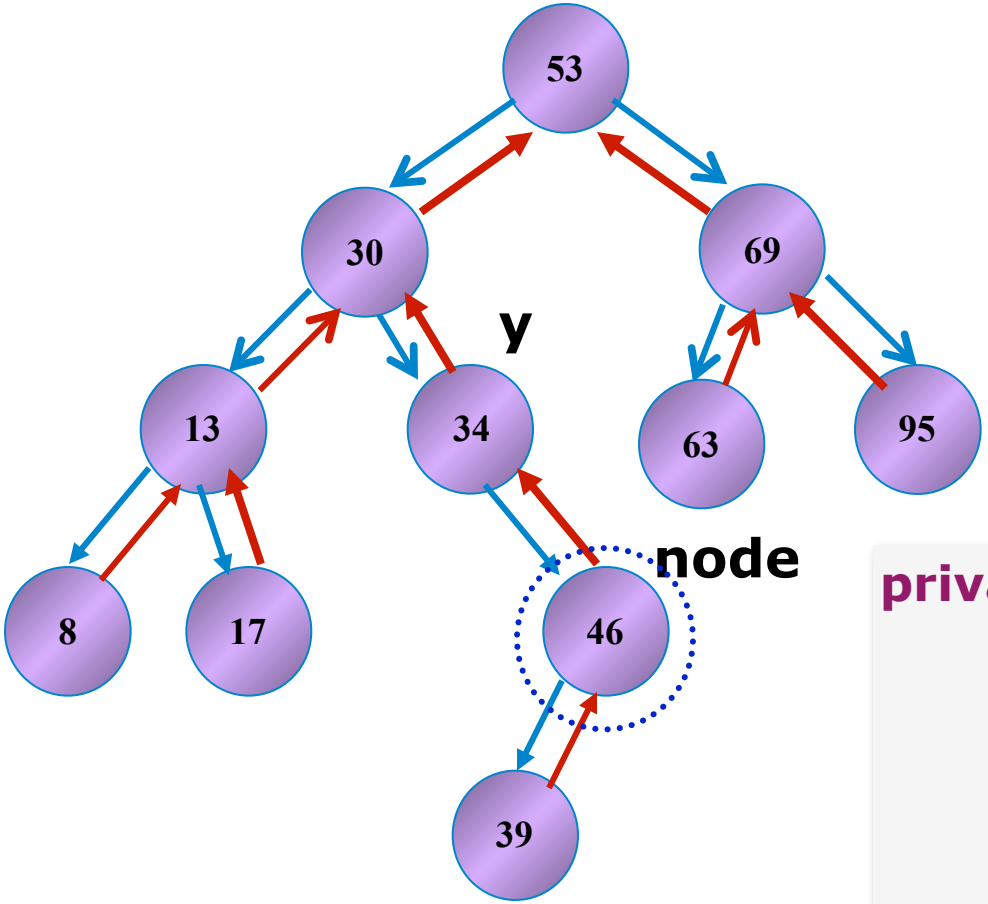
Doppelt verkettete Bäume

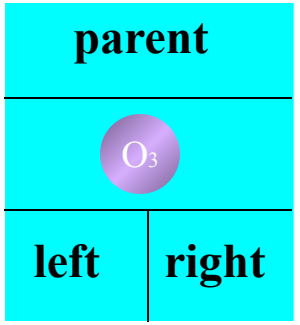





```
private class DTreeNode {
        private T key;
        private D data;
        private DTreeNode left
        private DTreeNode right;
        private DTreeNode parent;
        ...
}
```

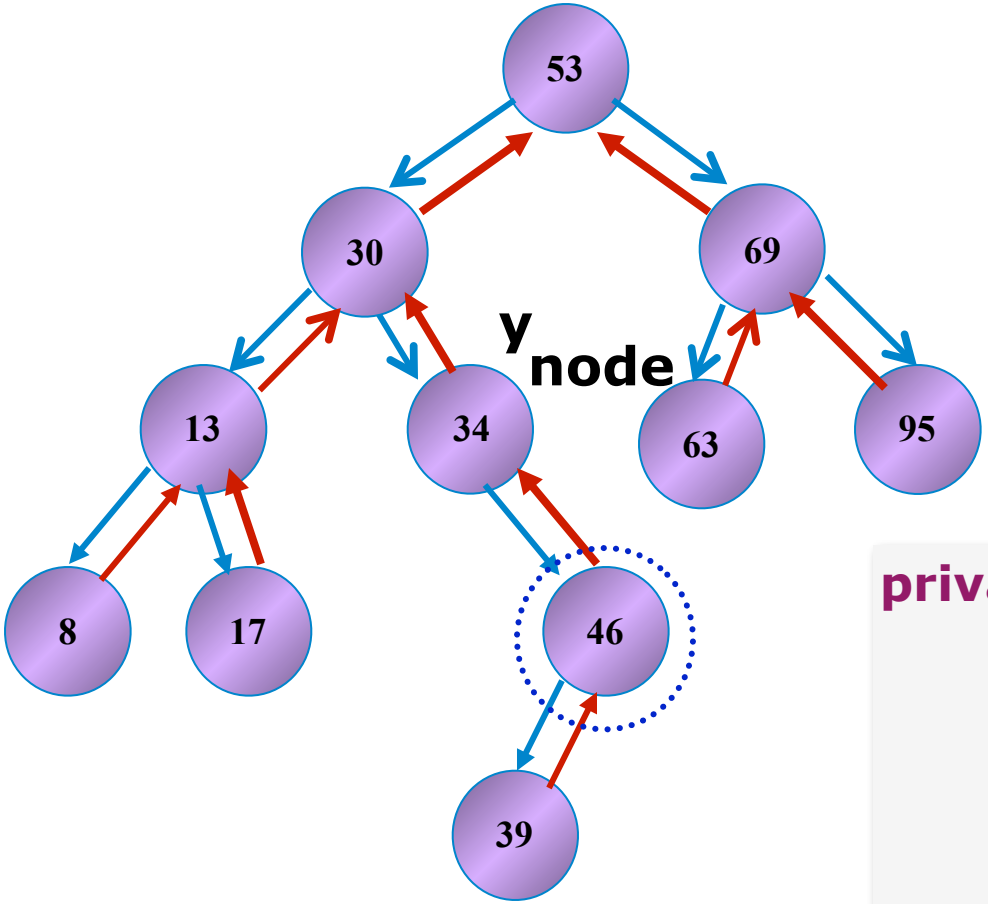
Doppelt verkettete Bäume

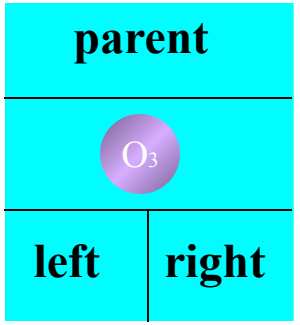





```
private class DTreeNode {
        private T key;
        private D data;
        private DTreeNode left
        private DTreeNode right;
        private DTreeNode parent;
        ...
}
```

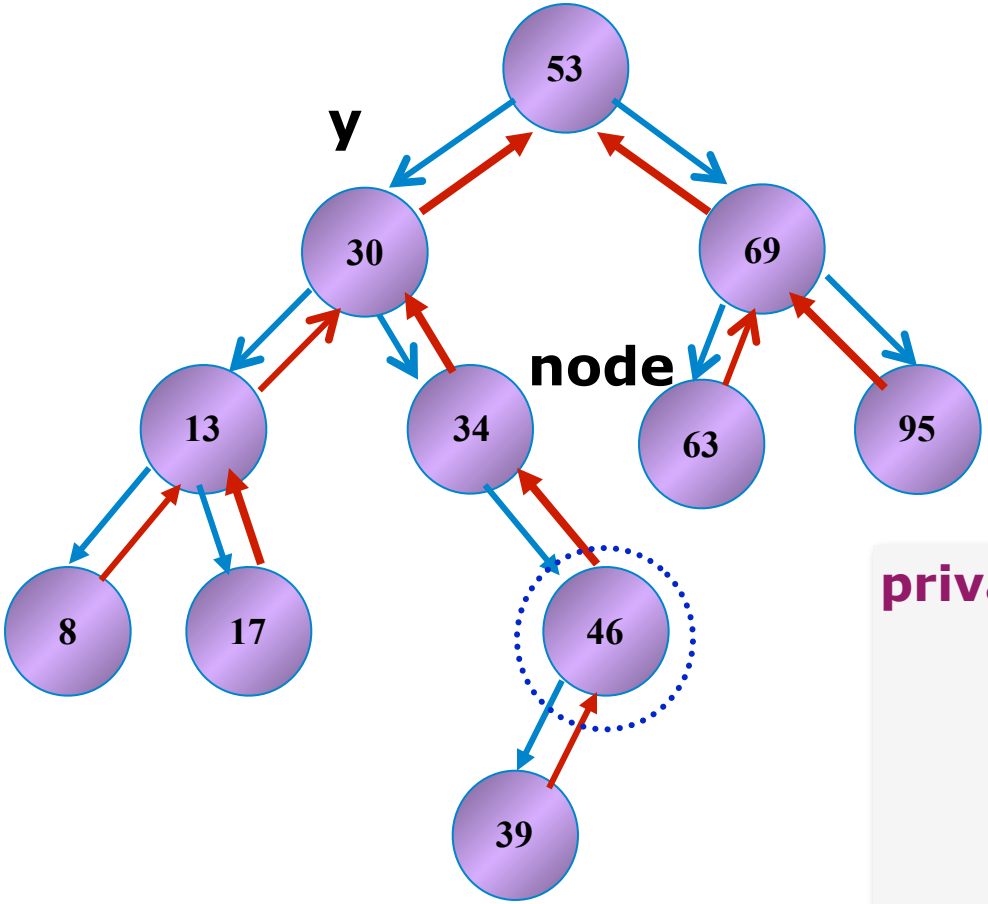
Doppelt verkettete Bäume

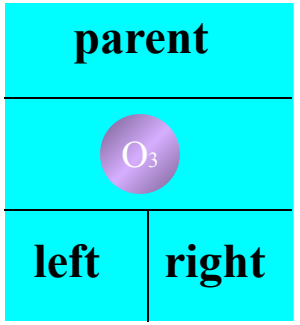





```
private class DTreeNode {
        private T key;
        private D data;
        private DTreeNode left
        private DTreeNode right;
        private DTreeNode parent;
        ...
}
```

Doppelt verkettete Bäume

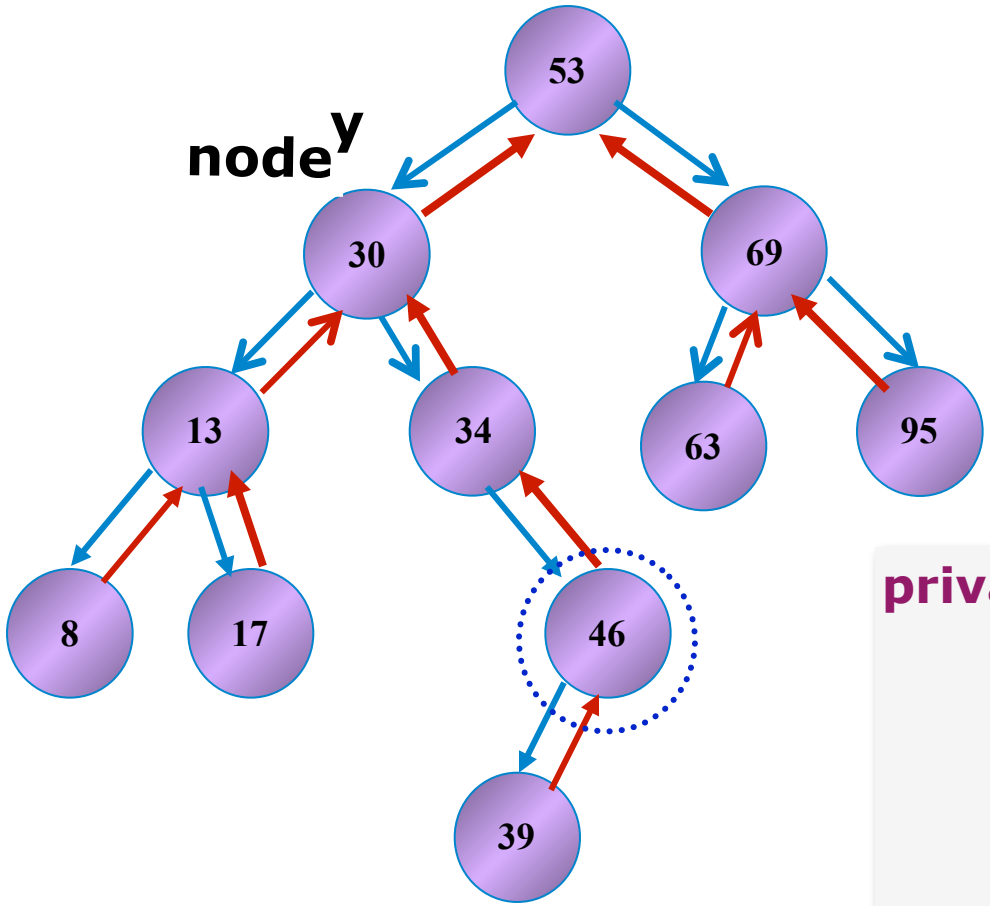


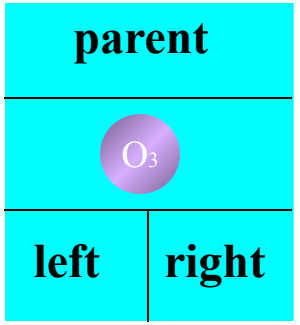



```
private class DTreeNode {
        private T key;
        private D data;
        private DTreeNode left
        private DTreeNode right;
        private DTreeNode parent;
        ...
}
```

Doppelt verkettete Bäume

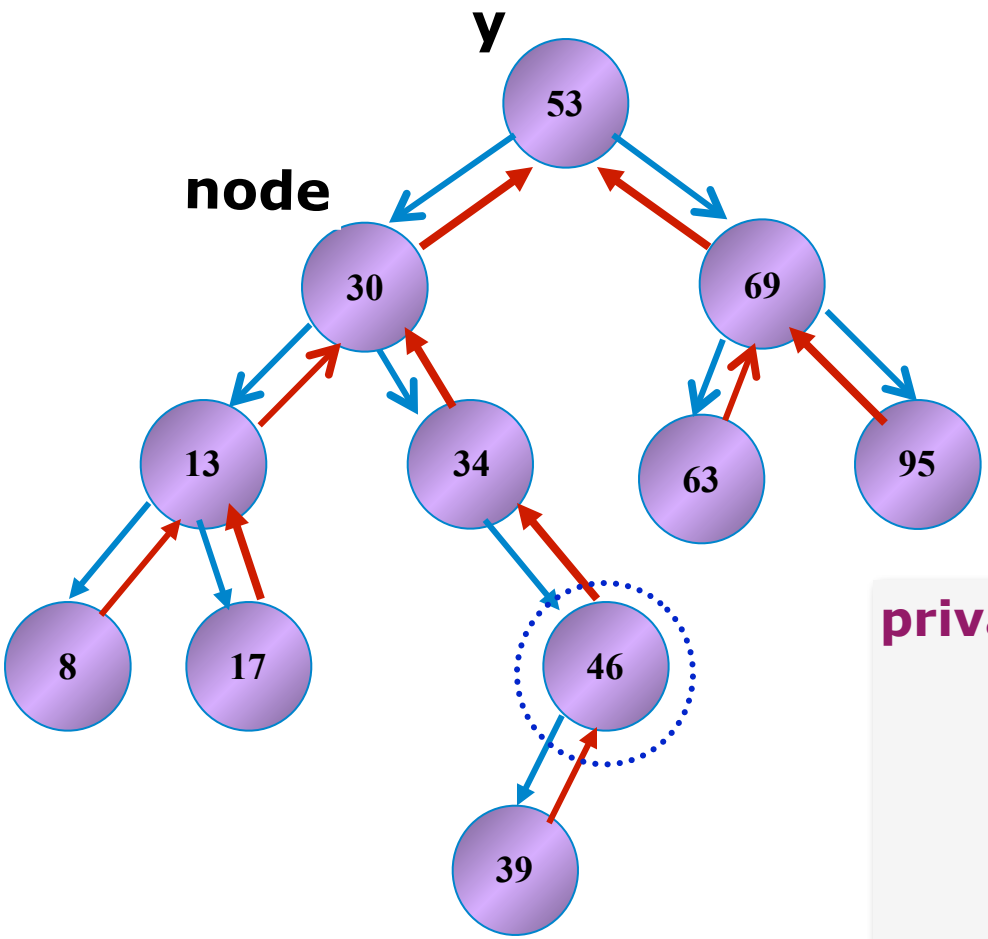
Freie Universität Berlin

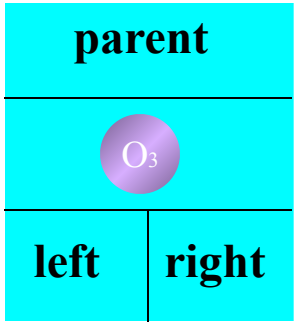





```
private class DTreeNode {
        private T key;
        private D data;
        private DTreeNode left
        private DTreeNode right;
        private DTreeNode parent;
        ...
}
```

Doppelt verkettete Bäume

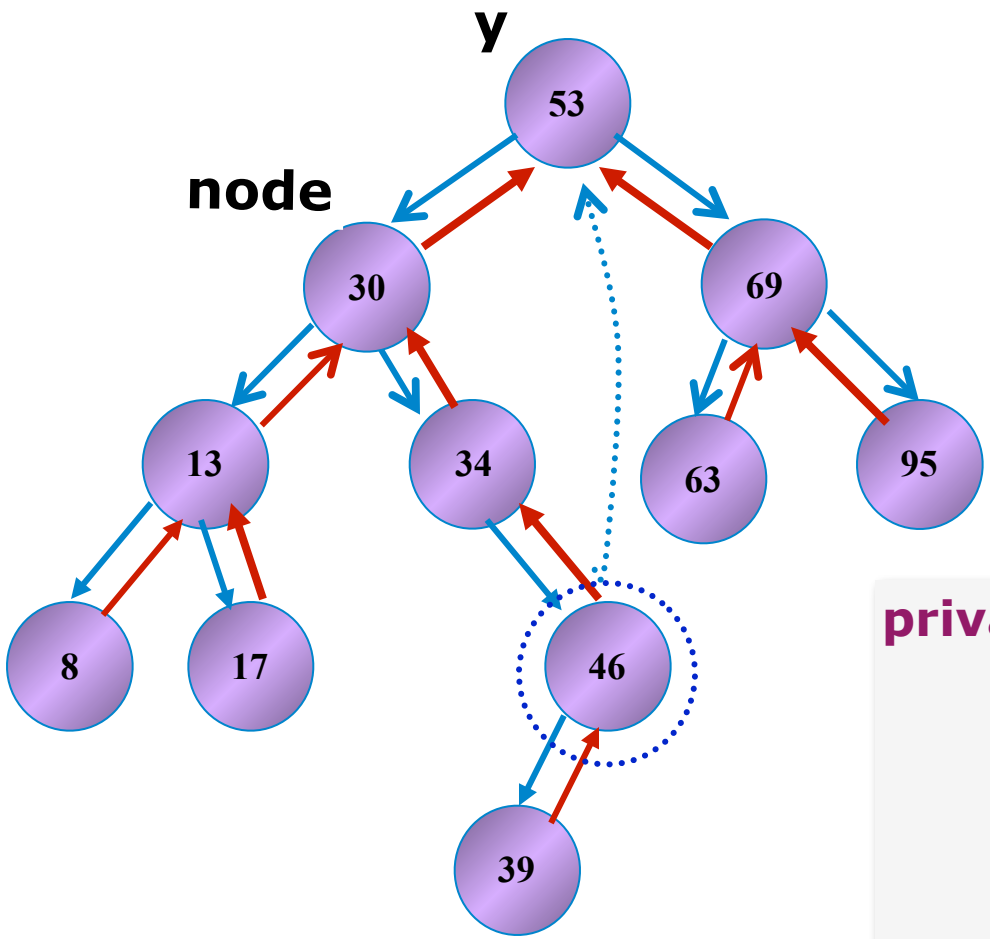


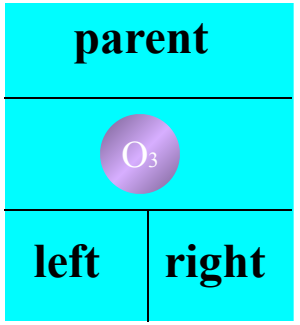



```
private class DTreeNode {
        private T key;
        private D data;
        private DTreeNode left
        private DTreeNode right;
        private DTreeNode parent;
        ...
}
```

Doppelt verkettete Bäume

Freie Universität Berlin

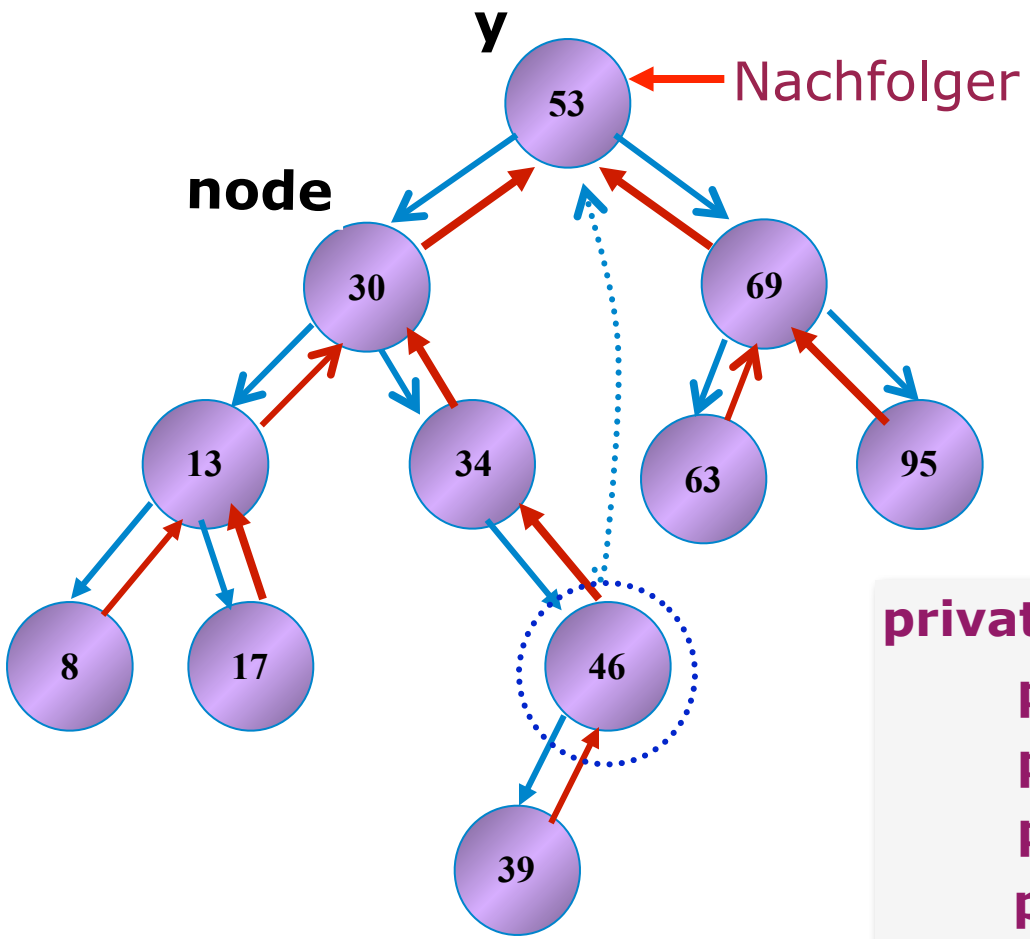


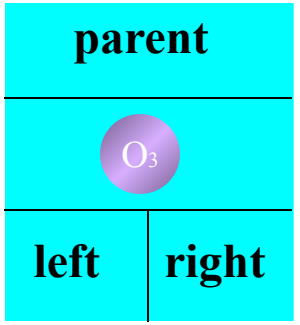



```
private class DTreeNode {
        private T key;
        private D data;
        private DTreeNode left
        private DTreeNode right;
        private DTreeNode parent;
        ...
}
```

Doppelt verkettete Bäume

Freie Universität Berlin



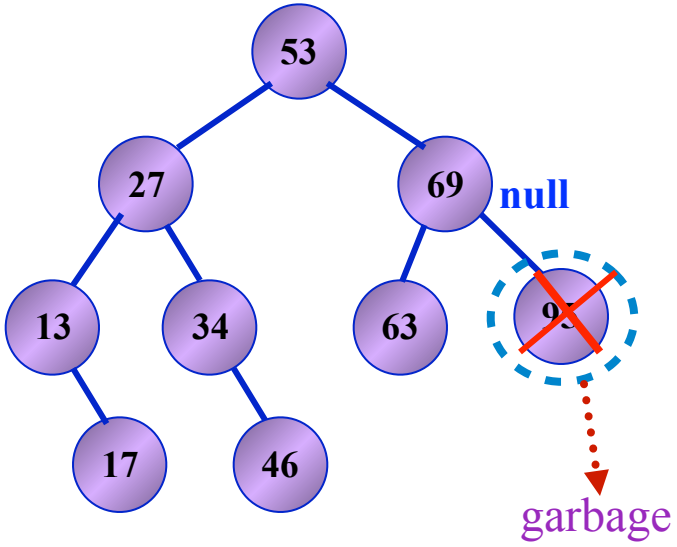



```
private class DTreeNode {
        private T key;
        private D data;
        private DTreeNode left
        private DTreeNode right;
        private DTreeNode parent;
        ...
}
```

Delete-Operation (Löschen)

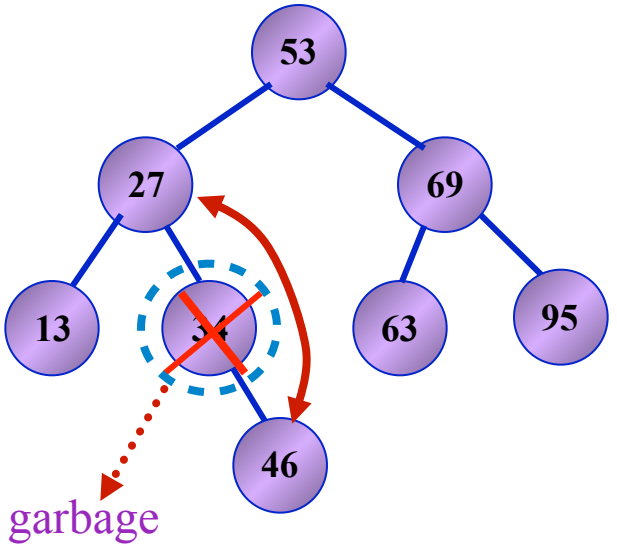
1. Fall

Löschen eines Knotens ohne Kinder



2. Fall

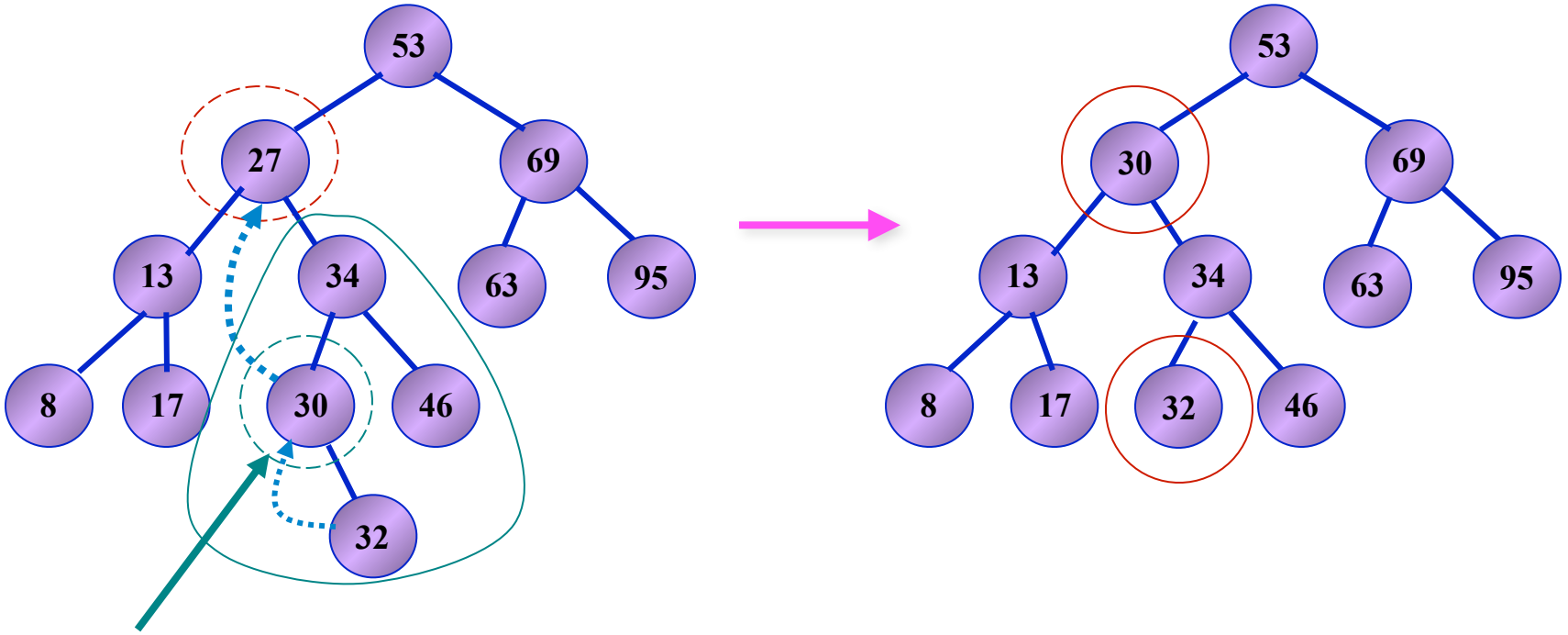
Löschen eines Knotens mit nur einem Kind



Löschen

Löschen eines Knotens mit zwei Kindern

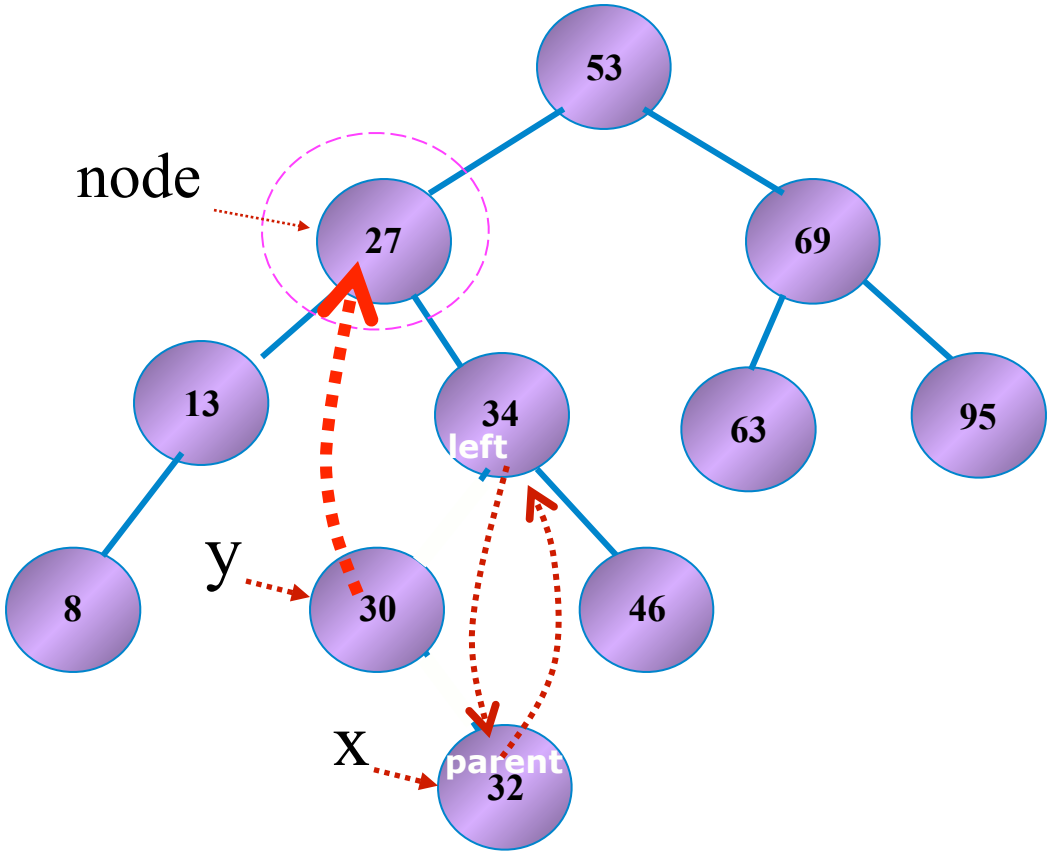
Der Knoten, den man löschen möchte, wird durch seinen Nachfolger ersetzt.



Der Nachfolger von 27 ist das Minimum des rechten Unterbaumes.

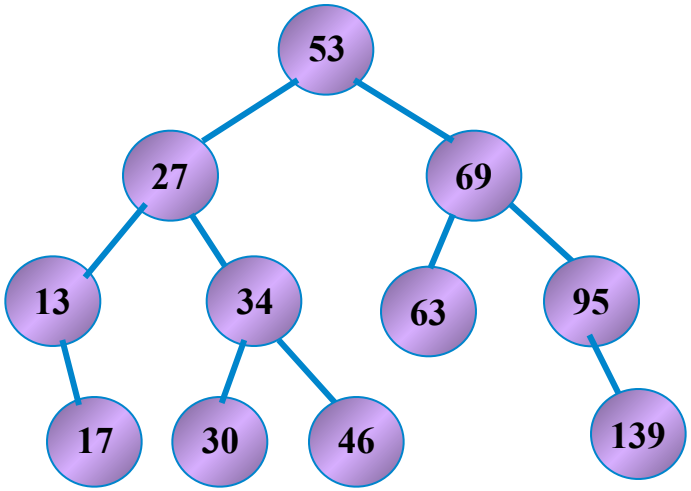
Das Minimum ist entweder ein Blatt oder hat maximal ein rechtes Kind.

Löschen

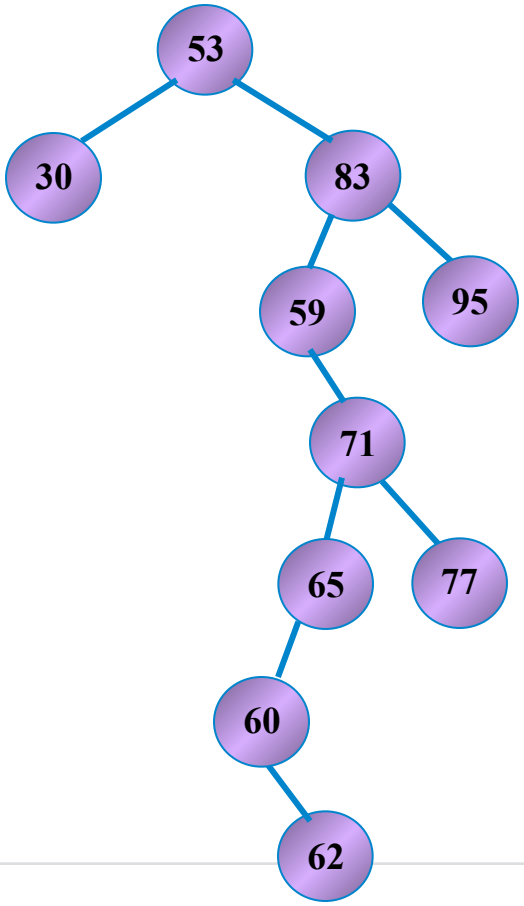


Probleme mit einfachen binären Suchbäumen

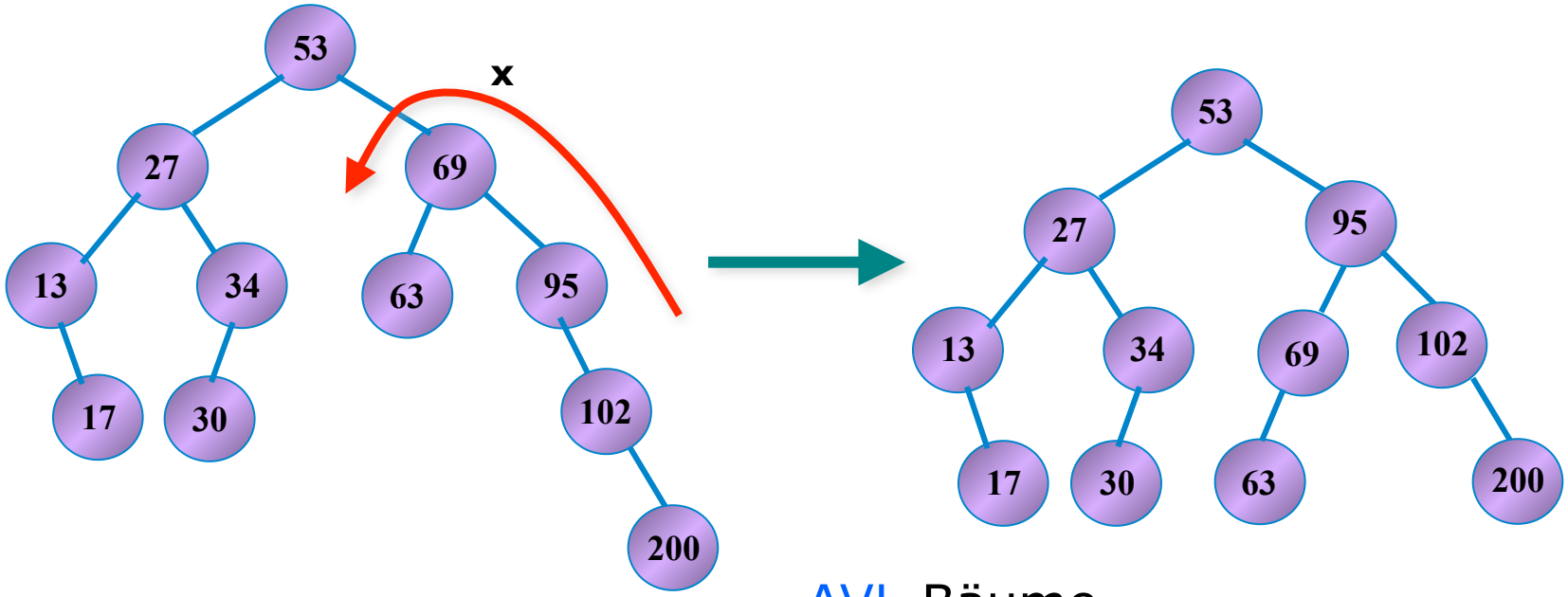
balancierter Binärbaum



nicht balancierter Binärbaum



Innerhalb der insert- und delete-Operationen wird mit Hilfe von Rotationen die Balance des Baumes ständig wiederhergestellt.



Lösungen

AVL-Bäume
Red-Black-Bäume
AA-Bäume
B-Bäume
usw.

Elementare Operationen für dynamische Mengen

	Liste	Array	Balancierter Binärbaum
	Schlimmster Fall	Schlimmster Fall	Schlimmster Fall
Suchen	O(n)	$O(\log_2(n))$	$O(\log_2(n))$
Einfügen	O(n)	O(n)	$O(\log_2(n))$
Löschen	O(n)	O(n)	$O(\log_2(n))$