

Algorithmen und Datenstrukturen

Kapitel 06: Bäume

Prof. Dr. Adrian Ulges

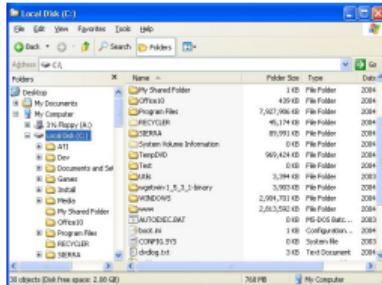
B.Sc. *Informatik*
Fachbereich DCSM
Hochschule RheinMain



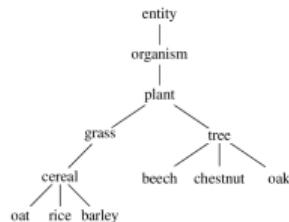
Bäume: Beispiele

Bäume sind **die Datenstruktur** in der Informatik.
 Sie begegnen uns in den **verschiedensten Teilgebieten!**

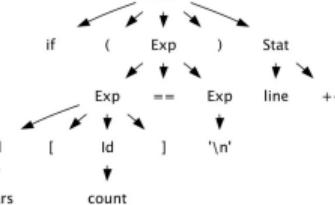
Dateisysteme



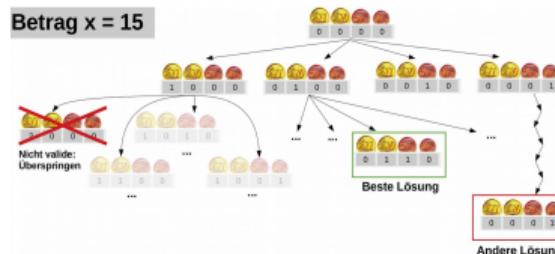
Objekthierarchien



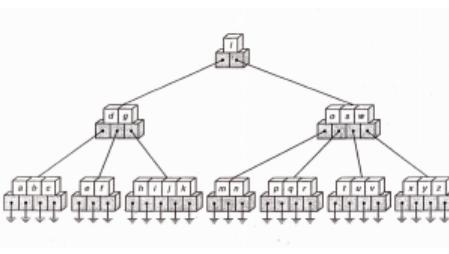
Syntaxbäume



Algorithmik: Heaps, Backtracking ...



Datenbanken: B-Bäume





Outline

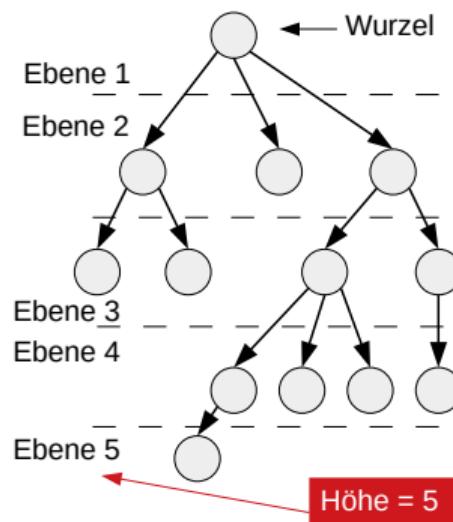
1. Grundlagen
2. Binärbäume: Implementierung
3. Traversierung von Bäumen
4. Iteratoren in Java
5. Iterator für Bäume: Implementierung

Bäume: Einführung

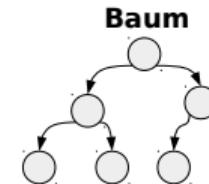


Dynamische Datenstrukturen

- ▶ Referenzierung zwischen Knoten.
- ▶ **Bisher:** Knoten verweist auf **max. 1 Nachfolger** → Liste.
- ▶ **Jetzt:** Knoten verweist auf **max. B Nachfolger** → Bäume.



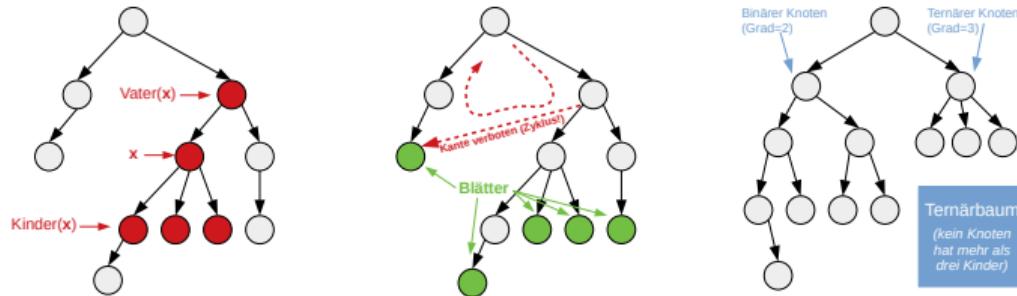
Verkettete Liste



Bäume

- ▶ Bäume bestehen aus Knoten und Kanten.
- ▶ Die Knoten sind in **Ebenen** angeordnet.
Auf Ebene 1 befindet sich die **Wurzel**.
- ▶ Die **Höhe** (bzw. **Tiefe**) des Baums entspricht der maximalen Ebene.

Bäume: Grundlagen



- ▶ Alle Knoten (*außer der Wurzel*) besitzen einen **Elternknoten** (oder Vaterknoten) und sind ihrerseits **Kinder** dieses Knotens.
- ▶ Wir unterscheiden zwischen **Blättern** und **inneren Knoten**. Innere Knoten besitzen Kinder, Blätter nicht.
- ▶ Bäume enthalten **keine Zyklen**.
- ▶ **Grad eines Knotens** = Anzahl seiner Kinder (*binärer Knoten = 2 Kinder, ternärer Knoten = 3 Kinder, ...*).
- ▶ **Ordnung (engl. “branch factor”)** eines Baums = **maximaler Grad** seiner Knoten.

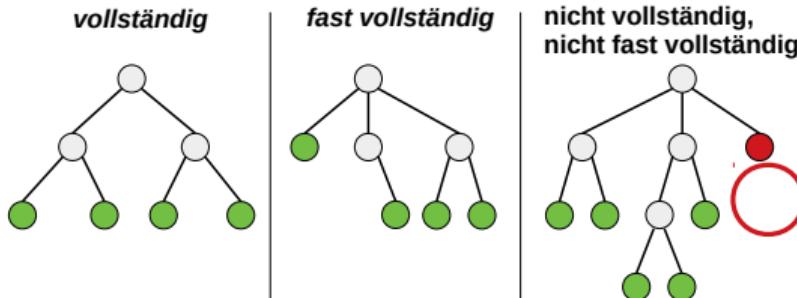
Exkurs: Bäume

Vollständigkeit

Vollständige Bäume

besitzen nur auf der letzten Ebene Blätter,

Fast vollständige Bäume auf den letzten zwei Ebenen.



Die Höhe eines vollständigen Baums ist logarithmisch!

- ▶ Was ist die Höhe eines **vollständigen binären Baums** mit **n Elementen**?
- ▶ Ein Baum der Höhe h kann $n = 2^h - 1$ Elemente speichern (*Beweis: Induktion*).
- ▶ Gleichung umstellen $\rightarrow h = \log_2(n + 1)$.
- ▶ **Zentrale Eigenschaft von Bäumen: Vollständige Bäume mit sehr vielen Elementen sind ziemlich flach! (Bsp. 1,000,000 Elemente \rightarrow Höhe 20).**

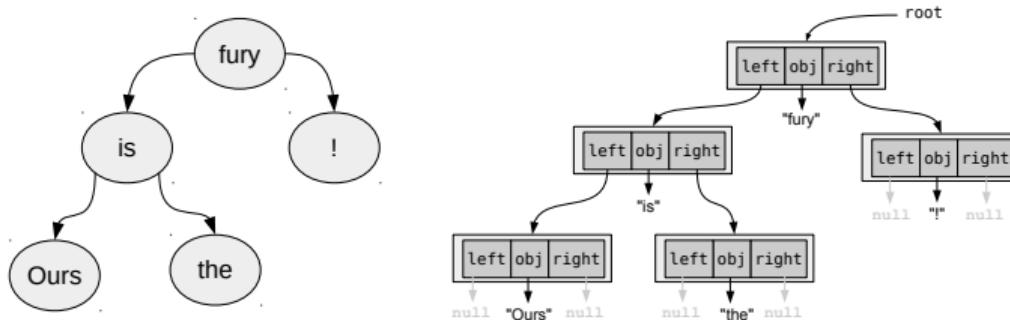
Höhe h	#Elemente n
1	1
2	3
3	7
4	15
5	31
...	...
10	1023
...	...
h	$2^h - 1$



Outline

1. Grundlagen
2. Binärbäume: Implementierung
3. Traversierung von Bäumen
4. Iteratoren in Java
5. Iterator für Bäume: Implementierung

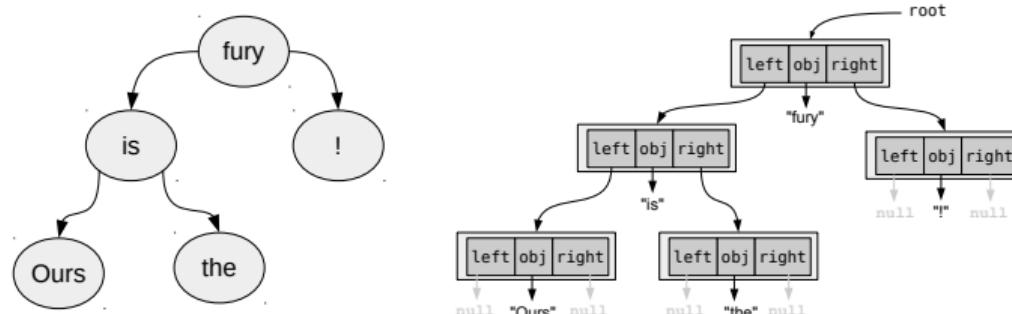
Implementierung von Binärbäumen



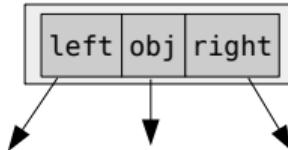
Implementierung (vgl. verkettete Listen)

- ▶ **Knoten** enthalten Nutzdaten (`obj`) und Referenzen auf die Kinder (`left`, `right`).
- ▶ Ein **Binärbaum** besitzt eine Referenz auf die Wurzel (`root`).

Implementierung von Binärbäumen



```
class Node {
    T obj;
    Node left;
    Node right;
};
```



```
public class BinTree<T> {

    private Node root;

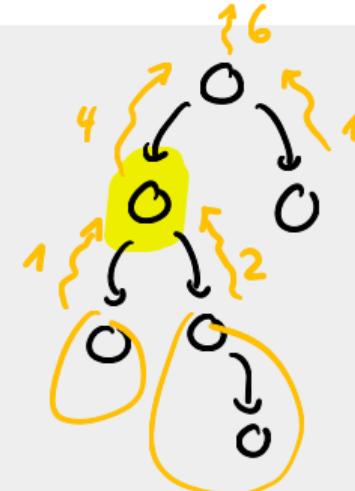
    // Node
    ...

    // Methoden
    ...
}
```

Beispiel: size()

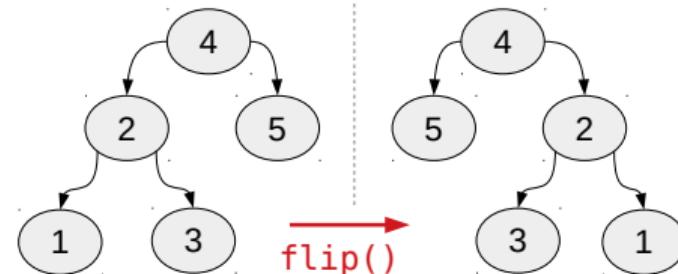
Die Methode **size()** soll die **Anzahl der Knoten** eines Baums zurückliefern:

```
public class BinTree<T> {  
    int size() {  
        return size(root);  
    }  
  
    int size(Node u) {  
        if (u == null)  
            return 0;  
        int l = size(u.left);  
        int r = size(u.right);  
        return l + r + 1;  
    }  
}
```



Do-Binäräume-Yourself

Implementieren Sie eine Methode `flip()`,
die einen gegebenen Binärbaum
horizontal **spiegelt**!



```
public class BinTree {
```

```
    ...
```

```
}
```



Outline

1. Grundlagen

2. Binärbäume: Implementierung

3. Traversierung von Bäumen

4. Iteratoren in Java

5. Iterator für Bäume: Implementierung

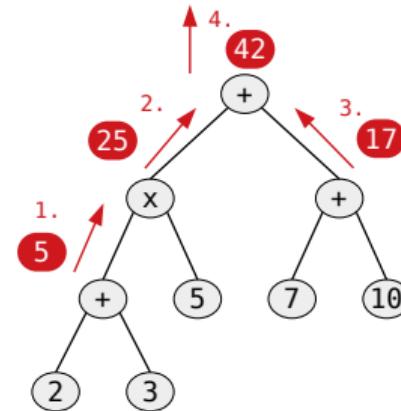
Traversierung: Definition

Was bedeutet "Traversierung"?

- ▶ Traversierung = Rekursives **Besuchen aller Knoten** im Baum
- ▶ Die besuchten Knoten werden "**verarbeitet**" / **ausgegeben**.

Beispiel: Evaluation von Ausdrücken

- ▶ Java überführt Ausdrücke (z.B. $(2+3) \times 5 + (7+10)$) in Bäume.
- ▶ Ziel: Bestimme den **Wert** des Ausdrucks.
- ▶ Wir traversieren den Baum. Innerer Knoten: Berechne **Wert des Teilausdrucks**.
- ▶ In welcher **Reihenfolge** müssen wir den Baum durchlaufen?
Erst die Kinder, dann die Eltern (engl. "post-order").

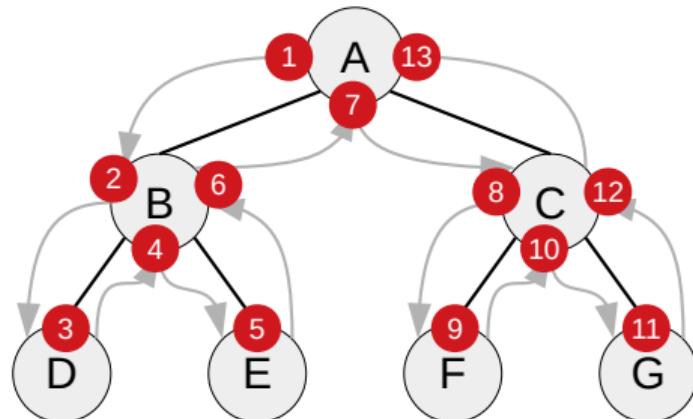


Traversierung: Optionen bei der Reihenfolge

- ▶ **pre-order:** Besuche erst den Elternknoten, dann die Kinder.
- ▶ **in-order:** linkes Kind, dann Elternknoten, dann rechtes Kind.
- ▶ **post-order:** erst Kinder, dann Elternknoten.
- ▶ **level-order:** Erst 1. Ebene, dann 2. Ebene, dann 3. ...

Beispiel: Knoten A wird besucht...

... in Schritt 1 (**pre-order**) bzw. Schritt 7 (**in-order**) bzw. Schritt 13 (**post-order**).

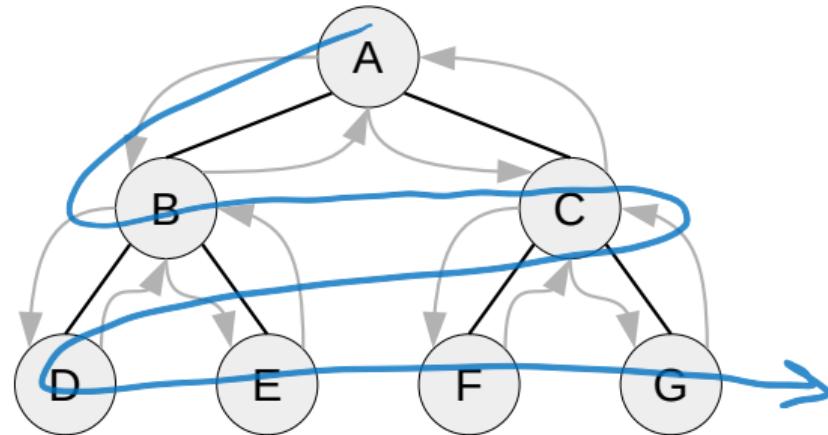


```
void traverse(Node n) {  
    process(n); // pre-order  
    if (n.left != null)  
        traverse(n.left);  
    process(n); // in-order  
    if (n.right != null)  
        traverse(n.right);  
    process(n); // post-order  
}
```

Traversierung: Beispiel

Wir traversieren die Knoten und geben ihre Bezeichner aus.

- ▶ **pre-order:** A B D E C F G
- ▶ **in-order:** D B E A F C G
- ▶ **post-order:** D E B F G C A
- ▶ **level-order:** A B C D E F G





Outline

1. Grundlagen

2. Binärbäume: Implementierung

3. Traversierung von Bäumen

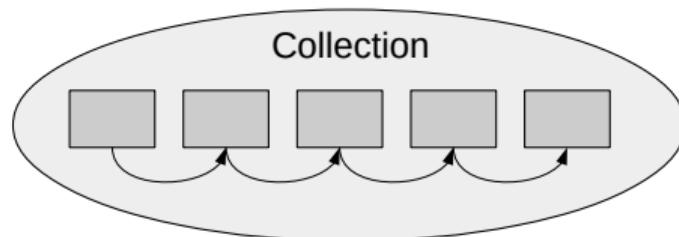
4. Iteratoren in Java

5. Iterator für Bäume: Implementierung

Iteratoren

Bei Collections **“durchwandert”** man häufig alle Elemente.

- ▶ Hierfür kennt Java das Konstrukt der **Foreach-Schleife**.
- ▶ Das Durchwandern sollte **unabhängig** vom konkreten Container sein (*Austauschbarkeit gewährleisten!*).



```
Collection collection
    = new LinkedList<String>();

for (String s : collection) {
    System.out.println(s);
}
```

Durchwandere...

- ▶ Arrays
- ▶ Listen
- ▶ Bäume
- ▶ ...

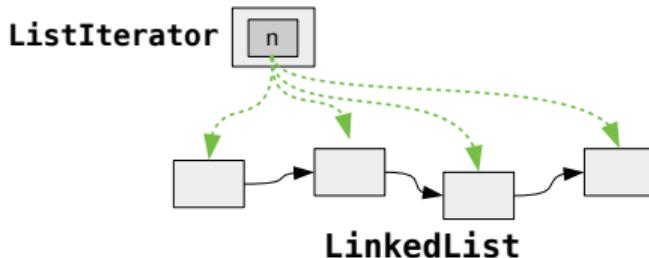
Lösung: Iteratoren

Wie machen wir unsere Collection “foreach-fähig”?

- ▶ Ein gemeinsames **Interface Iterator** für **alle Container**.
- ▶ Dieses definiert die Methoden **hasNext()** und **next()**.
- ▶ **hasNext()** ist genau dann true wenn noch nicht alle Elemente besucht wurden.
- ▶ **next()** liefert Objektreferenz des aktuellen Knotens und rückt “einen Schritt weiter”.

```
public interface Iterator<T> {  
    public boolean hasNext();  
    public T next();  
}
```

Beispiel-Iterator



Beispiel: Verkettete Liste

- ▶ Klasse **ListIterator**
- ▶ Initialisierung: Knoten *n*
(= *erster Knoten der Liste*) initialisiert.
- ▶ Anmerkung: Klasse ListIterator braucht Zugriff auf private Node-Klasse.
- ▶ **hasNext()**: Ende wenn tail erreicht.
- ▶ **next()**: Vorrücken zum nächsten Knoten,
Rückgabe der Objektreferenz.

```
public interface Iterator<T> {  
    public boolean hasNext();  
    public T next();  
}  
  
class ListIterator<T>  
    implements Iterator<T> {  
  
    private Node n; // current node  
  
    public ListIterator(Node n) {  
        this.n = n;  
    }  
  
    public boolean hasNext() {  
        // n is not tail  
        return n.next != null;  
    }  
  
    public T next() {  
        if (!hasNext())  
            throw new Exception("end!");  
        n = n.next; // move to next node  
        return n.prev.obj; // return obj  
    }  
}
```

Beispiel-Iterator

- ▶ Die Collection/LinkedList muss das Interface Iterable implementieren ...
- ▶ ... und hierzu selbst einen Iterator erstellen (iterator()) ...
- ▶ Wir können nun per **foreach-Schleife** durch die Liste wandern. ☺

```
interface Iterable<T> {  
    Iterator<T> iterator();  
    ...  
}
```

```
class LinkedList<T>  
    implements Iterable<T> {  
  
    // Liste muss einen Iterator erstellen  
    public Iterator<T> iterator() {  
        return new  
            // initialisiere mit erstem Knoten  
            ListIterator(head.next);  
    }  
    ...  
}
```

```
// Liste verwenden: foreach klappt.  
List<String> l = new  
    LinkedList<String>();  
  
for (String s: l) {  
    System.out.print(s);  
}
```



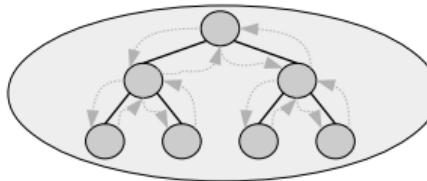
Outline

1. Grundlagen
2. Binärbäume: Implementierung
3. Traversierung von Bäumen
4. Iteratoren in Java
5. Iterator für Bäume: Implementierung

Beispiel-Iterator

Ziel: Iterator für Bäume

- ▶ Wir nutzen **Traversierung**.
- ▶ Wir entscheiden uns für die **in-order**-Variante.



```

Collection tree
    = new BinTree<String>();

for (String s : tree) {
    System.out.println(s);
}
    
```

Problem

- ▶ Wir müssen Traversierung in **Einzelschritten** realisieren (`next()`), nicht mit einem einzigen rekursiven Aufruf!

```

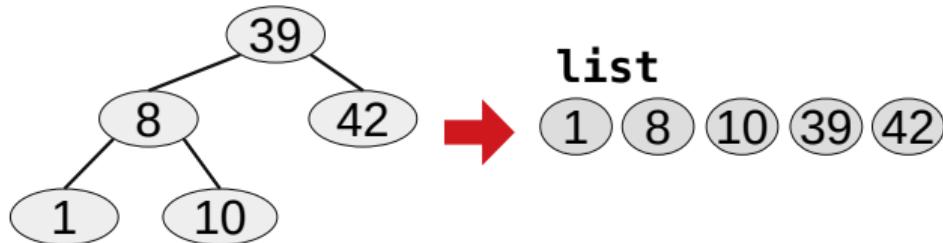
void traverse(Node n) {
    if (n.left != null)
        traverse(n.left);
    process(n); // in-order
    if (n.right != null)
        traverse(n.right);
}
    
```

```

public interface Iterator<T> {
    public boolean hasNext();
    public T next();
}
    
```



Implementierung 1: Naiv



1. Rufe `traverse()` auf und speichere **alle Knoten** in einer verketteten Liste.
2. Durchlaufe die **Liste** per Iterator.

Bewertung

- Die Implementierung dupliziert die komplette Datenstruktur ☹.
- Extra-Speicher der Komplexität $O(n)$.

```
class TreeIterator
    implements Iterator<T> {

    LinkedList<T> list;

    TreeIterator(Node root) {
        list = new LinkedList<T>();
        traverse(root, list);
    }

    // add nodes to list (in-order)
    void traverse(Node t,
                  LinkedList<T> list) {

        if (t.left != null)
            traverse(t.left, list);

        list.addLast(t.obj);

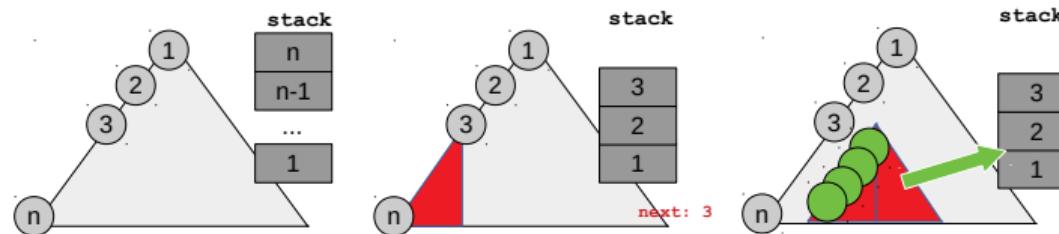
        if (t.right != null)
            traverse(t.right, list);
    }

    boolean hasNext() {
        return !list.isEmpty();
    }

    public T next() {
        return list.popFirst();
    }
}
```

Implementierung 2: Stack

- ▶ **Merke:** **Bevor** wir einen Knoten mit `next()` ausgeben, müssen wir (weil in-order) seinen **linken Teilbaum** ausgegeben haben.
- ▶ Wir merken speichern auf einem **Stack** noch zu besuchende Knoten.



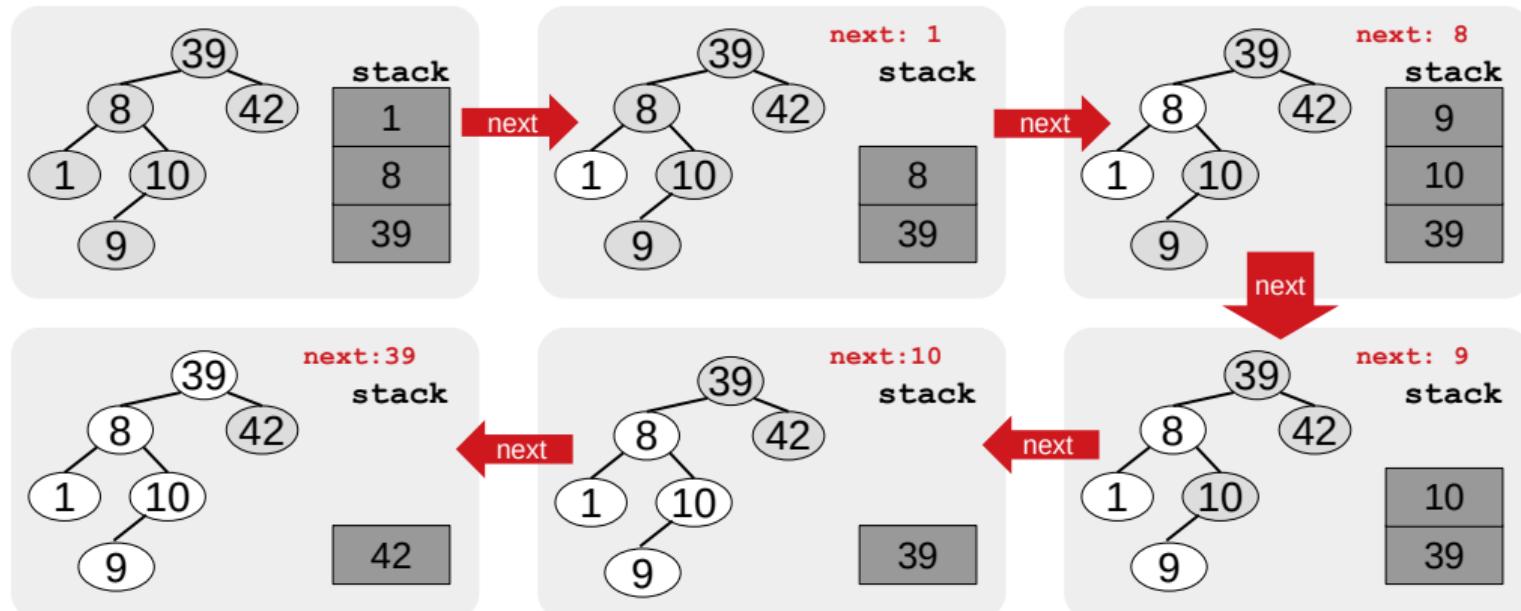
Ansatz

Jedes Mal wenn wir einen Teilbaum betreten, speichern wir die Knoten seines **linkesten Pfads** auf dem Stack.

1. Besuchen wir einen Knoten, legen pushen wir ihn auf den Stack und besuchen zunächst den linken Subbaum.
2. Holen wir den Knoten später vom Stack, geben wir ihn mit `next()` zurück und besuchen den rechten Teilbaum.



Implementierung 2: Beispiel



Implementierung 2: Code

```
class TreeIterator implements Iterator<T> {  
    Stack<Node> stack;  
  
    TreeIterator(Node root) {  
        stack = new Stack<Node>();  
        Node n = root;  
        while (n != null) {  
            stack.push(n);  
            n = n.left;  
        }  
    }  
  
    boolean hasNext() {  
        return !stack.isEmpty();  
    }  
}
```

T next () {

```
    Node result = stack.pop();  
    Node n = result.right;  
    while (n != null) {  
        stack.push(n);  
        n = n.left;  
    }  
    return result.obj;
```

}



Implementierung 2: Bewertung

Welchen zusätzlichen Speicherplatz benötigt Implementierung 2 ...

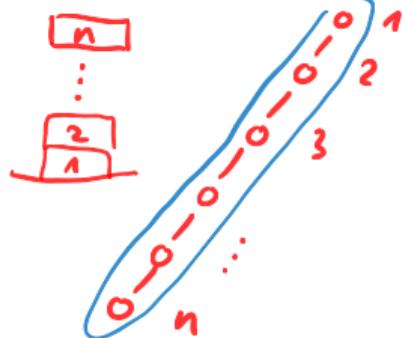
- ▶ ... im Best Case?
- ▶ ... im Worst Case?
- ▶ ... für vollständige Bäume?

Best Case



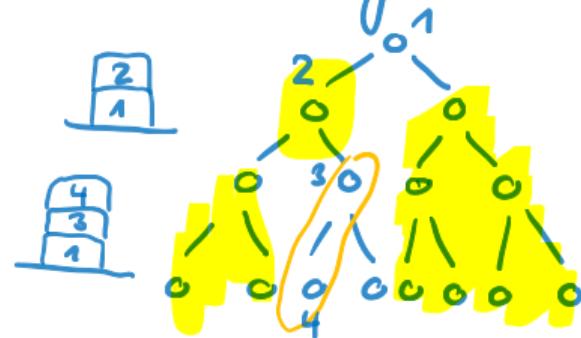
$$\Theta(1)$$

Worst Case



$$\Theta(n)$$

Vollständige Bäume



$$\Theta(\text{Höhe des Baums}) = \Theta(\log n)$$

Implementierung 2: Bewertung



References I

