

Lernzettel

Bäume: Traversierungen und Implementierung

Universität: Technische Universität Berlin
Kurs/Modul: Einführung in die Informatik - Vertiefung
Erstellungsdatum: September 20, 2025



Zielorientierte Lerninhalte, kostenlos!
Entdecke zugeschnittene Materialien für deine Kurse:

<https://study.AllWeCanLearn.com>

Einführung in die Informatik - Vertiefung

Lernzettel: Bäume: Traversierungen und Implementierung

(1) Grundbegriffe. Ein Binärbaum ist eine hierarchische Datenstruktur, bestehend aus Knoten. Jeder Knoten hat höchstens zwei Kinder: links (**left**) und rechts (**right**). Der Wurzelknoten besitzt keinen Elternknoten. In der Praxis dienen Bäume zur strukturieren Repräsentation von hierarchischen oder sortierten Informationen.

(2) Traversierungen (DFS) – Tiefensuche. Die drei klassischen Traversierungen definieren die Reihenfolge, in der Knoten besucht werden:

- Preorder (VORORDNUNG): Root, links, rechts.
- Inorder (INORDNERUNG): links, Root, rechts.
- Postorder (NACHORDINIERUNG): links, rechts, Root.

(2.1) Rekursive Implementierungen.

```
void preorder(Node<T> n){
    if(n==null) return;
    visit(n);
    preorder(n.left);
    preorder(n.right);
}

void inorder(Node<T> n){
    if(n==null) return;
    inorder(n.left);
    visit(n);
    inorder(n.right);
}

void postorder(Node<T> n){
    if(n==null) return;
    postorder(n.left);
    postorder(n.right);
    visit(n);
}
```

(2.2) Iterative Traversal mit Stack.

```
List<T> preorderIter(Node<T> root){
    List<T> res = new ArrayList<>();
    if(root==null) return res;
    Stack<Node<T>> st = new Stack<>();
    st.push(root);
    while(!st.isEmpty()){
        Node<T> n = st.pop();
        visit(n);
        if(n.right != null) st.push(n.right);
        if(n.left != null) st.push(n.left);
    }
}
```

```
    }
    return res;
}

List<T> inorderIter(Node<T> root){
    List<T> res = new ArrayList<>();
    Stack<Node<T>> st = new Stack<>();
    Node<T> curr = root;
    while(curr != null || !st.isEmpty()){
        while(curr != null){
            st.push(curr);
            curr = curr.left;
        }
        curr = st.pop();
        visit(curr);
        curr = curr.right;
    }
    return res;
}
```

(2.3) Level-Order Traversal (Breitensuche) mit Queue.

```
void levelOrder(Node<T> root){
    Queue<Node<T>> q = new LinkedList<>();
    if(root != null) q.add(root);
    while(!q.isEmpty()){
        Node<T> n = q.poll();
        visit(n);
        if(n.left != null) q.add(n.left);
        if(n.right != null) q.add(n.right);
    }
}
```

(3) Implementierung: Repräsentationen und Grundoperationen. Drei verbreitete Repräsentationen:

- Verknüpfte Knoten (Node<T> mit Feldern left, right).
- Array-basierter Binärbaum (Heap-Indexierung, z. B. für vollständige Bäume).

(3.1) Verknüpfte Knoten (Linked Structure).

```
public class Node<T> {
    T value;
    Node<T> left;
    Node<T> right;
    Node(T v){ value = v; left = right = null; }
}
```

```
public class BinaryTree<T> {  
    protected Node<T> root;  
    public boolean isEmpty(){ return root == null; }  
}
```

(3.2) BST-Implementierung (Grundoperationen).

```
public class BST<T extends Comparable<T>> {  
    Node<T> root;  
    public void insert(T v){  
        root = insert(root, v);  
    }  
    private Node<T> insert(Node<T> n, T v){  
        if(n == null) return new Node<>(v);  
        int cmp = v.compareTo(n.value);  
        if(cmp < 0) n.left = insert(n.left, v);  
        else if(cmp > 0) n.right = insert(n.right, v);  
        return n;  
    }  
    public boolean contains(T v){  
        return contains(root, v);  
    }  
    private boolean contains(Node<T> n, T v){  
        if(n == null) return false;  
        int cmp = v.compareTo(n.value);  
        if(cmp == 0) return true;  
        return cmp < 0 ? contains(n.left, v) : contains(n.right, v);  
    }  
}
```

(3.3) Array-basierter Binärbaum.

```
public class ArrayBinaryTree<T> {  
    private T[] data;  
    private int size;  
    public ArrayBinaryTree(int capacity){  
        data = (T[]) new Object[capacity];  
        size = 0;  
    }  
    // Einfügung an der ersten freien Position (vollständiger Baum)  
    public void add(T v){  
        data[size++] = v;  
    }  
    public T leftChild(int i){ return data[2*i+1]; }  
    public T rightChild(int i){ return data[2*i+2]; }  
}
```

(4) Beispiel: Traversierungen eines Beispielbaums. Gegeben sei der Baum mit Wurzel 8, links 3, rechts 10, weiter links 1 und 6, rechts 14.

- Preorder: 8, 3, 1, 6, 10, 14 - Inorder: 1, 3, 6, 8, 10, 14 - Postorder: 1, 6, 3, 14, 10, 8 -
Level-Order: 8, 3, 10, 1, 6, 14

(5) Komplexität. - Traversierungen: Jede Kante wird genau einmal besucht $\rightarrow$ Zeit $O(n)$. -
Speicherbedarf: DFS im Worst-Case $O(h)$ Zusatzspeicher (Stack), Level-Order $O(n)$ Queue. -
BST-Operationen (Durchschnitt): $O(\log n)$; im Worst-Case (degenerierter Baum): $O(n)$.

(6) Übungen. - Implementieren Sie eine iterative Inorder-Traversierung für einen generischen
Binärbaum. - Implementieren Sie eine Methode, die die Höhe eines Binärbaums berechnet.
- Analysieren Sie die Auswirkungen der Baumform (ausbalanciert vs. degeneriert) auf die
Komplexität von Insert und Search in einem BST.