

# Binärer Suchbaum (Binary Search Tree, BST)

## 1 Grundidee

Ein binärer Suchbaum ist ein **binärer Baum mit Ordnungsregel**:

- Alle Werte im **linken Teilbaum** sind **kleiner** als der Knoten
- Alle Werte im **rechten Teilbaum** sind **größer** als der Knoten
- Beide Teilbäume sind selbst wieder binäre Suchbäume

Diese Eigenschaft erlaubt **effizientes Suchen, Einfügen und Löschen**.

Siehe BFS für: Suche levelweise von oben nach unten und von links nach rechts. Also Ebene für Ebene über alle Zweige hinweg.

Siehe DFS für: Traversierungen (Inorder, Preorder, Postorder). Also einen Zweig komplett durch und dann zum nächsten

---

## 2 Voraussetzungen

- Elemente müssen **vergleichbar** sein (Ordnung  $<$ ,  $>$ )
  - **keine Duplikate** (oder klare Regel, wohin Duplikate eingefügt werden)
- 

## 3 Laufzeiten & Eigenschaften

Eigenschaft	Wert
Suche (Best / Avg)	$O(\log n)$
Suche (Worst)	$O(n)$
Einfügen	$O(\log n) / O(n)$
Löschen	$O(\log n) / O(n)$
Speicherbedarf	$O(n)$
In-place	nein
Stabil	nein

**Hinweis:** Der Worst Case tritt auf, wenn der Baum **degeneriert** (z. B. sortierte Eingabe).

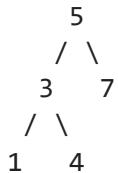
---

## 4 Schritt-für-Schritt-Beispiel

Einfügen der Werte:

[5, 3, 7, 1, 4]

### Aufbau des Baums



### Suche nach 4

- $4 < 5 \rightarrow$  gehe links
  - $4 > 3 \rightarrow$  gehe rechts
  - gefunden
- 

## 5 Besonderheiten / Prüfungsrelevante Hinweise

- Inorder-Traversierung liefert **sortierte Reihenfolge**
  - Performance hängt stark von der **Baumhöhe** ab
  - Grundlage für AVL-Bäume und Rot-Schwarz-Bäume
  - Nicht selbstbalancierend
- 

## 6 Vor- und Nachteile

### Vorteile

- effiziente Suche bei balanciertem Baum
- dynamische Datenstruktur
- natürliche Sortierung möglich

### Nachteile

- Worst Case  $O(n)$
  - kein automatisches Balancing
  - höherer Speicherbedarf als Arrays
-

# Merksatz für die Prüfung

Ein binärer Suchbaum speichert geordnete Daten, erlaubt effiziente Suche bei balancierter Struktur, kann aber im Worst Case linear werden.

---

## 7 Python-Implementierung

In [1]:

```
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

class BinarySearchTree:
    def __init__(self):
        self.root = None

    def insert(self, value):
        if self.root is None:
            self.root = Node(value)
        else:
            self._insert_recursive(self.root, value)

    def _insert_recursive(self, node, value):
        if value < node.value:
            if node.left is None:
                node.left = Node(value)
            else:
                self._insert_recursive(node.left, value)
        else:
            if node.right is None:
                node.right = Node(value)
            else:
                self._insert_recursive(node.right, value)

    def search(self, value):
        return self._search_recursive(self.root, value)

    def _search_recursive(self, node, value):
        if node is None:
            return False
        if node.value == value:
            return True
        if value < node.value:
            return self._search_recursive(node.left, value)
        else:
            return self._search_recursive(node.right, value)

    def inorder(self):
        result = []
        self._inorder_recursive(self.root, result)
```

```

    return result

def _inorder_recursive(self, node, result):
    if node:
        self._inorder_recursive(node.left, result)
        result.append(node.value)
        self._inorder_recursive(node.right, result)

```

## Baum-Traversierungen (Inorder, Preorder, Postorder)

### 1 Grundidee

Baum-Traversierungen legen fest, **in welcher Reihenfolge** die Knoten eines Baumes besucht werden. Bei binären Bäumen unterscheidet man drei klassische **Depth-First-Traversierungen (DFS)**.

- **Inorder:** links → Knoten → rechts
  - **Preorder:** Knoten → links → rechts
  - **Postorder:** links → rechts → Knoten
- 

### 2 Voraussetzungen

- Es muss ein **Baum** vorhanden sein (z. B. binärer Suchbaum)
  - Jeder Knoten kann maximal **zwei Kinder** haben
- 

### 3 Laufzeiten & Eigenschaften

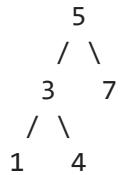
Eigenschaft	Wert
Laufzeit	$O(n)$
Speicher (rekursiv)	$O(h)$
In-place	ja
Stabil	–

**Hinweis:**  $n$  = Anzahl Knoten,  $h$  = Höhe des Baumes

---

### 4 Schritt-für-Schritt-Beispiel

Gegebener Baum:



**Inorder (links → Knoten → rechts)**

[1, 3, 4, 5, 7]

**Preorder (Knoten → links → rechts)**

[5, 3, 1, 4, 7]

**Postorder (links → rechts → Knoten)**

[1, 4, 3, 7, 5]

---

## 5 Besonderheiten / Prüfungsrelevante Hinweise

**Inorder**

- Liefert bei **binären Suchbäumen** eine **sortierte Reihenfolge**
- Sehr häufige Prüfungsfrage

**Preorder**

- Geeignet zum **Kopieren / Serialisieren** von Bäumen
- Wurzel wird zuerst verarbeitet

**Postorder**

- Geeignet zum **Löschen von Bäumen**
  - Kinder werden vor dem Elternknoten verarbeitet
- 

## 6 Vor- und Nachteile

**Vorteile**

- einfache Implementierung
- klare Struktur
- alle Knoten werden genau einmal besucht

## Nachteile

- rekursiv → Stackverbrauch
  - iterative Varianten komplexer
- 



## Merksätze für die Prüfung

- *Inorder liefert beim BST die sortierte Reihenfolge.*
  - *Preorder besucht zuerst die Wurzel.*
  - *Postorder besucht die Wurzel zuletzt.*
- 



## Python-Implementierung

In [2]:

```
class Node:  
    def __init__(self, value):  
        self.value = value  
        self.left = None  
        self.right = None  
  
def inorder(node, result):  
    if node:  
        inorder(node.left, result)  
        result.append(node.value)  
        inorder(node.right, result)  
  
def preorder(node, result):  
    if node:  
        result.append(node.value)  
        preorder(node.left, result)  
        preorder(node.right, result)  
  
def postorder(node, result):  
    if node:  
        postorder(node.left, result)  
        postorder(node.right, result)  
        result.append(node.value)  
  
# Beispiel  
root = Node(5)  
root.left = Node(3)  
root.right = Node(7)  
root.left.left = Node(1)  
root.left.right = Node(4)  
  
res = []
```

```

inorder(root, res)
print("Inorder:", res)

res = []
preorder(root, res)
print("Preorder:", res)

res = []
postorder(root, res)
print("Postorder:", res)

```

Inorder: [1, 3, 4, 5, 7]  
 Preorder: [5, 3, 1, 4, 7]  
 Postorder: [1, 4, 3, 7, 5]

## Breitensuche (BFS / Level-Order-Traversierung)

### 1 Grundidee

Die **Breitensuche (Breadth-First Search, BFS)** besucht die Knoten eines Baumes **levelweise von oben nach unten und von links nach rechts** innerhalb eines Levels.

Bei Bäumen wird BFS oft auch **Level-Order-Traversierung** genannt.

- Start bei der Wurzel
  - Nutzung einer **Queue (FIFO)**
  - Zuerst alle Knoten einer Ebene, dann die nächste Ebene
- 

### 2 Voraussetzungen

- Eine **Baumstruktur** (z. B. binärer Baum oder BST)
  - Zugriff auf Kinderknoten
  - Eine **Queue** zur Speicherung der nächsten Knoten
- 

### 3 Laufzeiten & Eigenschaften

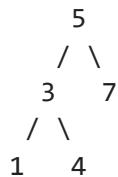
Eigenschaft	Wert
Laufzeit	$O(n)$
Speicherbedarf	$O(n)$
In-place	nein
Stabil	–

**Hinweis:** Im Worst Case (breiter Baum) befinden sich viele Knoten gleichzeitig in der Queue.

---

## 4 Schritt-für-Schritt-Beispiel

Gegebener Baum:



## Ablauf der BFS

1. Starte bei 5 → Queue: [5]
2. Besuche 5, füge Kinder ein → Queue: [3, 7]
3. Besuche 3, füge Kinder ein → Queue: [7, 1, 4]
4. Besuche 7 → Queue: [1, 4]
5. Besuche 1 → Queue: [4]
6. Besuche 4 → Queue: []

## Besuchsreihenfolge

[5, 3, 7, 1, 4]

---

## 5 Besonderheiten / Prüfungsrelevante Hinweise

- BFS nutzt immer eine **Queue**
  - Bei Bäumen: auch **Level-Order-Traversierung**
  - Bei Graphen: BFS liefert **kürzeste Wege in ungewichteten Graphen**
  - Gegensatz zu DFS (Stack / Rekursion)
- 

## 6 Vor- und Nachteile

### Vorteile

- intuitive Traversierung
- kürzeste Wege (bei Graphen)
- keine Rekursion nötig

### Nachteile

- höherer Speicherbedarf als DFS
  - Queue kann groß werden
- 

## Merksatz für die Prüfung

*Die Breitensuche besucht Knoten levelweise mithilfe einer Queue und wird bei Bäumen als Level-Order-Traversierung bezeichnet.*

---

## Python-Implementierung

In [3]:

```
from collections import deque

class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

def bfs(root):
    if root is None:
        return []

    result = []
    queue = deque([root])

    while queue:
        node = queue.popleft()
        result.append(node.value)

        if node.left:
            queue.append(node.left)
        if node.right:
            queue.append(node.right)

    return result

# Beispiel
root = Node(5)
root.left = Node(3)
root.right = Node(7)
root.left.left = Node(1)
root.left.right = Node(4)

print(bfs(root)) # [5, 3, 7, 1, 4]
```

[5, 3, 7, 1, 4]

# Beispiellösung für Übungsaufgabe

Aufgabenstellung:

Implementieren Sie eine Methode, die alle Knoten eines Binary Search Trees auf einem bestimmten Level (Tiefe) von links nach rechts zurückgibt. Die Wurzel befindet sich auf Level 0.

Vorgegebener Code:

```
class Node:
    def __init__(self, key, val):
        self.key = key
        self.val = val
        self.left = None
        self.right = None

class BST:
    def __init__(self):
        self.root = None

    # Implementieren Sie diese Methode
    def get_keys_at_level(self, level):
        # Ihre Implementierung hier
        pass
```

Lösung als Python-Code:

```
In [4]: def get_keys_at_level(self, level):
    if self.root is None:
        return []

    result = []
    queue = [(self.root, 0)]

    while queue:
        node, current_level = queue.pop(0)

        if current_level == level:
            result.append(node.key)
        elif current_level < level:
            if node.left:
                queue.append((node.left, current_level + 1))
            if node.right:
                queue.append((node.right, current_level + 1))

    return result
```