

# Datenstrukturen & Algorithmen – Gesamtprüfungssheet (ausführlich)

---

## 1) Allgemeine Theorie zu Algorithmen

### 1.1 Was sind rekursive Algorithmen?

Ein **rekursiver Algorithmus** löst ein Problem, indem er **sich selbst** aufruft – aber mit einem **kleineren Teilproblem**.

Damit Rekursion funktioniert, brauchst du immer zwei Bausteine:

1. **Abbruchfall (Base Case)** Der Fall, bei dem die Funktion **nicht** weiter rekursiv aufruft.  
Ohne Base Case → Endlosrekursion.
2. **Rekursiver Fall** Der Fall, bei dem das Problem **verkleinert** wird (z. B.  $n \rightarrow n-1$ , Teilbaum → Kind-Teilbaum).

**Merksatz:**

Rekursion ist wie „zerlege in kleinere Kopien desselben Problems“, bis du unten an kommst.

### Wann ist Rekursion praktisch?

- Bäume (Traversieren, zählen, Höhe berechnen)
- Divide-and-Conquer (Mergesort, Quicksort)
- Backtracking (z. B. Sudoku, Pfadsuche)
- Graphen (DFS oft rekursiv)

### Typische Prüfungsfälle

- Base Case vergessen
- Problem wird nicht wirklich kleiner (z. B. falsche Parameter)
- Rückgabe/Accumulator falsch (z. B. `return` fehlt)

---

### 1.2 Python-Code: Anzahl Knoten in einem Baum rekursiv zählen (vollständige Funktion)

Annahme: Ein Knoten hat Attribute `left` und `right`. Leerer Baum ist `None`.

### Idee in Worten:

- Wenn kein Knoten da ist → 0
- Sonst: 1 (dieser Knoten) + Anzahl links + Anzahl rechts

```
def count_nodes(node):
    """
    Zählt rekursiv die Anzahl Knoten in einem (binären) Baum.
    node: Wurzelknoten oder None
    return: int
    """

    if node is None:
        return 0

    left_count = count_nodes(node.left)
    right_count = count_nodes(node.right)
    return 1 + left_count + right_count
```

### Quick-Check:

- Blattknoten → `left/right = None` → 1
- Leerer Baum → 0

```
def count_nodes(node):
    """Zählt alle Knoten im (Teil-)Baum mit Wurzel 'node' rekursiv."""
    if node is None:
        return 0
    total = 1 # zähle den aktuellen Knoten
    for child in (node.children() or []):
        total += count_nodes(child)
    return total

def count_nodes_iter_dfs(root):
    """Iteratives Zählen per Tiefensuche (Stack)."""
    if root is None:
        return 0
    stack = [root]
    count = 0
    while stack:
        node = stack.pop()
        count += 1
        # Kinder auf den Stack Legen (Reihenfolge egal)
        children = node.children() or []
        stack.extend(children)
    return count
```

---

## 2) Array-Liste und verkettete Liste

## 2.1 Was ist eine Array-Liste?

Eine **Array-Liste** (z. B. Python `list`, Java `ArrayList`) speichert Elemente in einem **zusammenhängenden Speicherbereich**.

Eigenschaften:

- Zugriff per Index ist schnell: `a[i]`
- Einfügen/Entfernen in der Mitte ist teuer (weil verschoben werden muss)
- Oft wird intern mit **Capacity** gearbeitet: wenn voll, wird ein größeres Array angelegt und kopiert

**Typisches Bild:**

Index:	0	1	2	3
Wert:	[A]	[B]	[C]	[D]

---

## 2.2 Was ist eine verkettete Liste?

Eine **verkettete Liste** besteht aus Knoten (Nodes). Jeder Knoten speichert:

- den Wert
- einen Verweis auf den nächsten (und evtl. vorherigen) Knoten

Typen:

- **Singly linked list:** nur `next`
- **Doubly linked list:** `next` und `prev`

**Typisches Bild (singly):**

[A] -> [B] -> [C] -> [D] -> None

---

## 2.3 Wo unterscheiden sich die beiden?

- Array-Liste: **Index-Zugriff schnell**, aber **Verschieben teuer**
- Verkettete Liste: **Einfügen/Entfernen lokal billig**, aber **Suchen/Index langsam**

## 2.4 Komplexität: Element entnehmen (löschen)

**Array-Liste**

- Anfang löschen: **O(n)** (alles rutscht nach)
- Mitte löschen: **O(n)** (Verschieben)
- Ende löschen: **O(1)** amortisiert

## Verkettete Liste (singly)

- Anfang löschen: **O(1)**
  - Ende löschen: **O(n)** (vorletztes finden)
  - Mitte löschen: **O(n)** (erst finden) + O(1) (Pointer)  $\Rightarrow$  **O(n)**
- 

## 2.5 Komplexität: Element aus der Mitte entnehmen

- Array-Liste: Zugriff O(1), Verschieben **O(n)**  $\Rightarrow$  **O(n)**
- Verkettete Liste: Suchen **O(n)**, Entfernen O(1)  $\Rightarrow$  **O(n)**

**Merksatz:** Beide sind O(n), aber aus unterschiedlichen Gründen.

---

## 3) Komplexitäten von Sortieralgorithmen (ausführliche Übersicht)

### 3.1 Elementare Sortieralgorithmen

Algorithmus	Best Case	Average Case	Worst Case	Stabil?	In-place?
Bubble Sort	O(n) (mit early-exit)	O(n <sup>2</sup> )	O(n <sup>2</sup> )	ja	ja
Selection Sort		O(n <sup>2</sup> )	O(n <sup>2</sup> )	nein (typisch)	ja
Insertion Sort	O(n)	O(n <sup>2</sup> )	O(n <sup>2</sup> )	ja	ja

---

### 3.2 Sortieren durch Einfügen (Insertion Sort)

**Idee:** Links ist sortiert, rechts unsortiert. Nimm das nächste Element und schiebe es nach links bis es passt.

Komplexität:

- Best: **O(n)** (schon sortiert)
  - Average: **O(n<sup>2</sup>)**
  - Worst: **O(n<sup>2</sup>)** (umgekehrt sortiert)
-

## 3.3 Quicksort

- Average/Expected: **O(n log n)** (unter üblichen Annahmen bewiesen, z. B. zufälliges Pivot)
- Worst: **O(n<sup>2</sup>)** (schlechte Pivotwahl, ungünstige Eingabe)

Formulierung, die in Prüfungen oft passt:

Für eine konkrete Pivotstrategie gibt es keine Garantie, dass immer  $n \cdot \log(n)$  erreicht wird; Worst-Case kann  $n^2$  sein.

---

## 3.4 Mergesort

- Best / Average / Worst: **O(n log n)**
  - stabil: ja
  - extra Speicher:  $O(n)$
- 

## 3.5 Sonstige komplexe Sortieralgorithmen (Überblick)

Algorithmus	Average	Worst	Notizen
Heapsort	$O(n \log n)$	$O(n \log n)$	nicht stabil
Timsort	$\sim O(n \log n)$	$O(n \log n)$	sehr gut bei teilweise sortiert
Counting Sort	$O(n+k)$	$O(n+k)$	Wertebereich klein
Radix Sort	$O(d*(n+k))$	$O(d*(n+k))$	ziffernweise

---

## 4) Heap-Sort (mit Anleitung)

### 4.1 Wie ist ein Heap aufgebaut?

Heap = vollständiger Binärbaum.

- Max-Heap: Eltern  $\geq$  Kinder (Root ist Maximum)
- Min-Heap: Eltern  $\leq$  Kinder (Root ist Minimum)

Array-Indizes:

- linkes Kind:  $2*i + 1$
- rechtes Kind:  $2*i + 2$

- parent:  $(i-1)/2$
- 

## 4.2 Wie wird ein Element eingefügt?

1. Element ans Ende
  2. heapify-up: solange Element > parent, tauschen  $\rightarrow O(\log n)$
- 

## 4.3 Wie wird ein Element entnommen?

1. Root entfernen (Maximum)
  2. letztes Element nach oben
  3. heapify-down: mit größerem Kind tauschen, solange Kind größer  $\rightarrow O(\log n)$
- 

## 4.4 Wie kann mit einem Heap ein Sortierverfahren erstellt werden?

Heapsort (aufsteigend):

1. Max-Heap bauen  $\rightarrow O(n)$
  2. n-mal Root entfernen + heapify-down  $\rightarrow O(n \log n)$  Nicht stabil.
- 

# 5) Binäre und balancierte Suchbäume

## 5.1 Was ist ein AVL Tree?

- BST + Balance-Faktor BF = Höhe(l) – Höhe(r)
- erlaubt  $BF \in \{-1, 0, +1\}$
- Rotationen: LL, RR, LR, RL

## 5.2 Was ist ein Rot-Schwarz-Baum?

- BST + Farbenregeln
  - Root schwarz, NIL schwarz
  - kein Rot-Rot
  - gleiche Black-Height
-

## 5.3 RB Beispiel: 7 Inserts (nach jedem Insert skizzieren)

Sequenz: [10, 20, 30, 15, 25, 5, 1]

1. 10

10B

2. 20

10B

\

20R

3. 30 (Rotation + Recolor)

20B  
/ \  
10R 30R

4. 15 (Uncle rot → Recolor)

20B  
/ \\  
10B 30B  
\ /  
15R

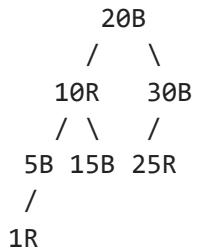
5. 25

20B  
/ \\  
10B 30B  
\ /  
15R 25R

6. 5

20B  
/ \\  
10B 30B  
/ \ /  
5R 15R 25R

7. 1 (Uncle rot → Recolor)



## 6) Hashtabelle

### 6.1 Was ist eine Hashtabelle?

- Schlüssel → Hashfunktion → Index
- durchschnittlich  $O(1)$

### 6.2 Kollisionen lösen (mit Beispielen)

#### Separate Chaining

$m=5$ ,  $h(k)=k \bmod 5$ , Schlüssel: 7,12,17 → Index 2

- Index 2: 7 → 12 → 17

#### Linear Probing

$m=7$ ,  $h(k)=k \bmod 7$ , Schlüssel: 10,17,24

- 10→3
  - 17→3 (Kollision) → 4
  - 24→3 (Kollision) → 5
- 

## 7) Graphentheorie

### 7.1 Traversieren

- DFS (Stack/Rekursion)
- BFS (Queue)

### 7.2 Breitensuche (BFS) + Python-Code

```

from collections import deque

def bfs(graph, start):
    visited = set([start])
    order = []
    q = deque([start])

    while q:
        v = q.popleft()
        order.append(v)
        for n in graph[v]:
            if n not in visited:
                visited.add(n)
                q.append(n)

    return order

```

## 7.3 Tiefensuche (DFS) + Python-Code

```

def dfs(graph, start, visited=None, order=None):
    if visited is None:
        visited = set()
    if order is None:
        order = []

    visited.add(start)
    order.append(start)

    for n in graph[start]:
        if n not in visited:
            dfs(graph, n, visited, order)

    return order

```

---

# 8) Spannbäume, Kruskal, Prim, Dijkstra

## 8.1 Was ist ein Spannbaum?

- verbindet alle Knoten
- keine Zyklen
- $V-1$  Kanten
- MST minimiert Gesamtkosten

## 8.2 Kruskal (wie funktioniert er?)

1. Kanten sortieren

2. Kanten nehmen, wenn kein Zyklus
3. stoppen bei  $V-1$

## 8.3 Prim (wie funktioniert er?)

1. Startknoten wählen
2. billigste Kante nach außen wählen
3. wiederholen bis alle Knoten drin

## 8.4 Dijkstra (Abstandsbestimmung)

- $\text{dist}[\text{start}] = 0$ , sonst  $\infty$
  - wiederholt: kleinste offene Distanz fixieren
  - Nachbarn relaxen
  - keine negativen Gewichte
- 

# 9) String-Matching

## 9.1 KMP

- Muster vorverarbeiten (LPS/Prefix)
- Text nicht zurück, Muster springt
- typ. linear

## 9.2 Boyer-Moore

- von rechts nach links
- Bad-Character / Good-Suffix
- oft sehr schnell

## 9.3 KMP vs Boyer-Moore

- KMP: sehr systematisch, garantiert gute Worst-Case-Idee
- BM: sehr schnell in der Praxis, große Sprünge, mehr Heuristik

# AVL-Baum – 12 Inserts Schritt für Schritt

Einfüge-Reihenfolge: 10, 20, 30, 25, 28, 27, 5, 4, 3, 8, 9, 7

Notation:

- $\text{BF} = \text{Höhe(left)} - \text{Höhe(right)}$
  - AVL ist ok, wenn  $\text{BF} \in \{-1, 0, +1\}$
  - Bei  $|\text{BF}| > 1$  musst du rotieren (LL/RR/LR/RL)
- 

## Schritt 1: Insert 10

**Balance-Check:** Kein Knoten mit  $|\text{BF}| > 1 \rightarrow \text{keine Rotation.}$

**Baum nach dem Schritt:**

10

---

## Schritt 2: Insert 20

**Balance-Check:** Kein Knoten mit  $|\text{BF}| > 1 \rightarrow \text{keine Rotation.}$

**Baum nach dem Schritt:**

10  
  \  
  20

---

## Schritt 3: Insert 30

**Erster unausgeglichener Knoten (z):**

- $z = 10$
- Höhe links = 0, Höhe rechts = 2
- $\text{BF}(z) = -2 \Rightarrow |\text{BF}| > 1 \rightarrow \text{Rotation nötig}$
- Fall: **RR**
- Rotation(en): **Linksrotation(at 10)**

**Baum nach dem Schritt:**

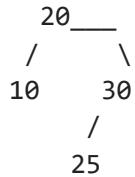
20  
  /  
  \  
10  30

---

## Schritt 4: Insert 25

**Balance-Check:** Kein Knoten mit  $|BF| > 1 \rightarrow$  keine Rotation.

Baum nach dem Schritt:

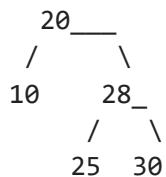


## Schritt 5: Insert 28

**Erster unausgeglichener Knoten (z):**

- $z = 30$
- Höhe links = 2, Höhe rechts = 0
- $BF(z) = 2 \Rightarrow |BF| > 1 \rightarrow$  Rotation nötig
- Fall: **LR**
- Rotation(en): **Linksrotation(at 25), dann Rechtsrotation(at 30)**

Baum nach dem Schritt:

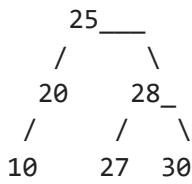


## Schritt 6: Insert 27

**Erster unausgeglichener Knoten (z):**

- $z = 20$
- Höhe links = 1, Höhe rechts = 3
- $BF(z) = -2 \Rightarrow |BF| > 1 \rightarrow$  Rotation nötig
- Fall: **RL**
- Rotation(en): **Rechtsrotation(at 28), dann Linksrotation(at 20)**

**Baum nach dem Schritt:**

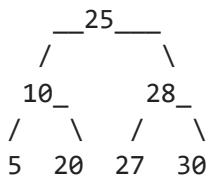


## Schritt 7: Insert 5

**Erster unausgeglichener Knoten (z):**

- $z = 20$
- Höhe links = 2, Höhe rechts = 0
- $BF(z) = 2 \Rightarrow |BF| > 1 \rightarrow \text{Rotation nötig}$
- Fall: **LL**
- Rotation(en): **Rechtsrotation(at 20)**

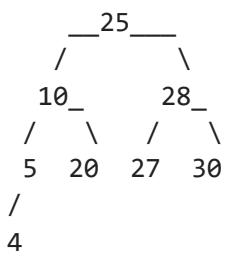
**Baum nach dem Schritt:**



## Schritt 8: Insert 4

**Balance-Check:** Kein Knoten mit  $|BF| > 1 \rightarrow \text{keine Rotation.}$

**Baum nach dem Schritt:**

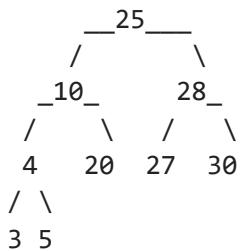


## Schritt 9: Insert 3

### **Erster unausgeglichener Knoten (z):**

- $z = 5$
- Höhe links = **2**, Höhe rechts = **0**
- $BF(z) = 2 \Rightarrow |BF| > 1 \rightarrow \text{Rotation nötig}$
- Fall: **LL**
- Rotation(en): **Rechtsrotation(at 5)**

### **Baum nach dem Schritt:**



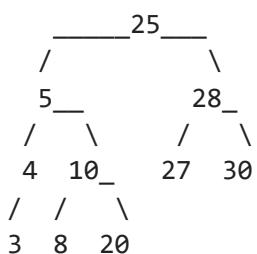
---

## **Schritt 10: Insert 8**

### **Erster unausgeglichener Knoten (z):**

- $z = 10$
- Höhe links = **3**, Höhe rechts = **1**
- $BF(z) = 2 \Rightarrow |BF| > 1 \rightarrow \text{Rotation nötig}$
- Fall: **LR**
- Rotation(en): **Linksrotation(at 4), dann Rechtsrotation(at 10)**

### **Baum nach dem Schritt:**



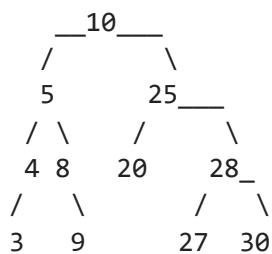
---

## **Schritt 11: Insert 9**

### **Erster unausgeglichener Knoten (z):**

- $z = 25$
- Höhe links = 4, Höhe rechts = 2
- $BF(z) = 2 \Rightarrow |BF| > 1 \rightarrow \text{Rotation nötig}$
- Fall: **LR**
- Rotation(en): **Linksrotation(at 5), dann Rechtsrotation(at 25)**

### **Baum nach dem Schritt:**



## **Schritt 12: Insert 7**

**Balance-Check:** Kein Knoten mit  $|BF| > 1 \rightarrow \text{keine Rotation.}$

### **Baum nach dem Schritt:**

