Prim-Algorithmus – Prüfungssheet (Abgrenzung zu BST)

Datenstrukturen & Algorithmen – Spickzettel

1️⃣ Grundidee des Prim-Algorithmus

Der **Prim-Algorithmus** berechnet einen **minimalen Spannbaum (MST)** eines **ungerichteten, gewichteten Graphen**.

Ziel:

- alle Knoten verbinden
- keine Zyklen
- **minimales Gesamtgewicht**

❗ Prim hat **nichts mit einem BST zu tun**. Prim arbeitet auf **Graphen**, ein BST ist eine **Datenstruktur** für geordnete Werte.

2️⃣ Voraussetzungen

- ungerichteter Graph
- gewichtete Kanten
- Graph ist zusammenhängend (sonst: minimaler Spannwald)

3️⃣ Wichtige Abgrenzung: Prim vs. BST

Prim-Algorithmus	Binary Search Tree (BST)
Graph-Algorithmus	Baum-Datenstruktur
arbeitet mit Kantengewichten	arbeitet mit Schlüsselwerten
Ziel: minimale Gesamtkosten	Ziel: effiziente Suche
kein Suchbaum	Suchbaum
kein Vergleich < >	basiert auf Ordnung < >

👉 **Ein MST ist kein BST.** 👉 **Prim benutzt keinen BST**, sondern typischerweise eine **Priority Queue**.

4️⃣ Algorithmus in Worten (prüfungsreif)

1. Wähle einen beliebigen Startknoten
2. Lege ihn in die Menge der besuchten Knoten
3. Betrachte alle Kanten, die von besuchten zu unbesuchten Knoten führen
4. Wähle die Kante mit dem **kleinsten Gewicht**
5. Füge diese Kante und den neuen Knoten zum Spannbaum hinzu
6. Wiederhole, bis alle Knoten enthalten sind

5️⃣ Wichtige Eigenschaften

- Greedy-Algorithmus
- lokale optimale Wahl → global optimales Ergebnis
- MST enthält genau **V − 1 Kanten**
- Ergebnis ist eindeutig, wenn alle Kantengewichte verschieden sind

---

6️⃣ Typische Prüfungsfalle

❌ Falsch: „Prim berechnet kürzeste Wege."

✅ Richtig: Prim minimiert das **Gesamtgewicht des Netzes**, nicht einzelne Pfade.

Kürzeste Wege:

- ungewichtet → BFS
- gewichtet → Dijkstra

7️⃣ Datenstrukturen bei Prim

Typisch verwendet:

- Menge der besuchten Knoten
- **Priority Queue (Min-Heap)** für Kanten

❗ Kein BST nötig oder sinnvoll

8️⃣ Python-Beispiel (Prim mit Priority Queue)

```python
import heapq

def prim(graph, start):
    visited = set()
    mst = []
    pq = [(0, start, None)]  # (gewicht, knoten, vorgaenger)

    while pq:
        weight, node, parent = heapq.heappop(pq)
        if node in visited:
            continue

        visited.add(node)
        if parent is not None:
            mst.append((parent, node, weight))

        for neighbor, w in graph[node]:
            if neighbor not in visited:
                heapq.heappush(pq, (w, neighbor, node))

    return mst
```

9️⃣ Mini-Beispiel (gedanklich)

Graph: A--1--B | / | 4 2 3 | / | C--5---D

Prim ab A:

- A–B (1)
- B–C (2)
- B–D (3)

Gesamtgewicht = 6

🔟 Merksätze für die Prüfung

> Prim = minimaler Spannbaum Prim ≠ kürzester Weg Prim ≠ BST

> BST speichert Werte geordnet. Prim verbindet Knoten günstig.

1️⃣1️⃣ Ultra-Kurz-Zusammenfassung

- Prim arbeitet auf Graphen
- Ziel: minimale Gesamtkosten
- Greedy-Verfahren

- nutzt Priority Queue
- kein Bezug zu BST als Suchbaum