Albert-Ludwigs-Universität
Institut für Informatik
Prof. Dr. F. Kuhn
M. Fuchs, G. Schmid

Algorithmen und Datenstrukturen
Sommersemester 2026
Musterlösung Übungsblatt 6

Abgabe: Dienstag, 9. Juni, 2026, 10:00 Uhr

Aufgabe 1: Rekonstruktion aus Traversierungen *(13 Punkte)*

(a) Rekonstruieren Sie den binären Suchbaum, dessen **Pre-Order** Traversierung gerade

$$P = (8, 4, 2, 6, 5, 7, 12, 10, 14, 13)$$

ist. *(2 Punkte)*

(b) Geben Sie die **Post-Order** Traversierung des rekonstruierten Baums an. *(2 Punkte)*

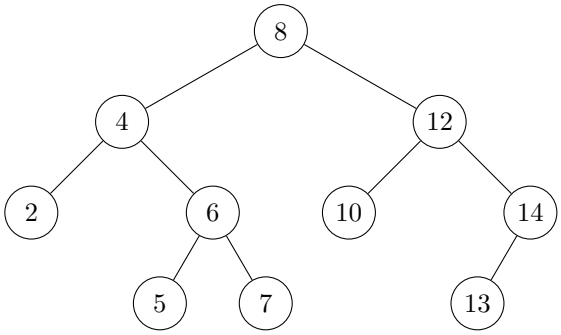
(c) Beschreiben Sie einen rekursiven Algorithmus, der aus einer **Pre-Order** Traversierung eines binären Suchbaums mit paarweise verschiedenen Schlüsseln die **Post-Order** Traversierung rekonstruiert. Der Algorithmus soll eine Laufzeit von $O(n)$ haben.

Hinweis: Eine $O(n \log n)$ Lösung gibt bis zu 4 Punkte. *(7 Punkte)*

(d) Seien T und T' zwei binäre Suchbäume, die aus den exakt gleichen Schlüsseln bestehen. Beweisen oder widerlegen Sie folgende Aussage: *Wenn die **In-Order** Reihenfolge von T und T' identisch ist, dann sind auch T und T' identisch.* *(2 Punkte)*

Musterlösung

(a) Der Baum ist eindeutig bestimmt, da es sich um einen binären Suchbaum mit paarweise verschiedenen Schlüsseln handelt. Die Wurzel ist das erste Element der Pre-Order Traversierung, also 8. Danach gehören alle folgenden Schlüssel, die kleiner als 8 sind, zum linken Teilbaum, und alle folgenden Schlüssel, die größer als 8 sind, zum rechten Teilbaum. Rekursiv erhält man den folgenden Baum:



Äquivalent beschrieben: 8 hat die Kinder 4 und 12, 4 hat die Kinder 2 und 6, 6 hat die Kinder 5 und 7, 12 hat die Kinder 10 und 14, und 14 hat das linke Kind 13.

(b) Die Post-Order Traversierung besucht zuerst den linken Teilbaum, dann den rechten Teilbaum und zuletzt die Wurzel. Für den Baum aus Teilaufgabe (a) ergibt sich daher

$$(2, 5, 7, 6, 4, 10, 13, 14, 12, 8).$$

(c) Wir verwenden die Suchbaum-Eigenschaft direkt. Statt die Grenze zwischen linkem und rechtem Teilbaum jedes Mal durch lineares Suchen zu bestimmen, lesen wir die Pre-Order Liste nur einmal von links nach rechts und übergeben in der Rekursion ein erlaubtes Schlüsselintervall.

Sei P die gegebene Pre-Order Traversierung und sei i ein globaler Index auf das nächste noch nicht verwendete Element von P. Der Aufruf

$$\textsc{Post}(a, b)$$

verarbeitet genau den Teilbaum, dessen Schlüssel im offenen Intervall (a, b) liegen müssen.

```
i = 0

Post(a,b):
    if i == n:
        return

    if P[i] <= a or P[i] >= b:
        return

    r = P[i]
    i = i + 1

    Post(a,r)    // linker Teilbaum
    Post(r,b)    // rechter Teilbaum
    output r
```

Der initiale Aufruf ist

$$\textsc{Post}(-\infty, +\infty).$$

Zur Korrektheit: In einer Pre-Order Traversierung steht die Wurzel eines Teilbaums immer vor den Knoten seiner beiden Teilbäume. Der aktuelle Wert $P[i]$ ist also genau dann die Wurzel des momentan betrachteten Teilbaums, wenn er im erlaubten Intervall liegt. Für den linken Teilbaum werden die zulässigen Schlüssel auf (a, r) eingeschränkt, für den rechten Teilbaum auf (r, b). Da der Algorithmus zuerst den linken Teilbaum, dann den rechten Teilbaum und erst danach r ausgibt, ist die Ausgabe eine Post-Order Traversierung.

Jeder Schlüssel wird genau einmal als Wurzel eines rekursiven Aufrufs akzeptiert und verarbeitet. Alle übrigen Prüfungen sind konstant teuer. Damit beträgt die Laufzeit $O(n)$. Der zusätzliche Speicherbedarf ist $O(h)$ für den Rekursionsstack, wobei h die Höhe des Baums ist.

Hinweis: Im Folgenden ist eine Implementierung des Pseudocodes in Python gegeben:

```python
import math

def get_postorder(pre):
    """Outer function to encapsulate the state and initialize variables.

    Note: The input 'pre' must be a valid preorder traversal of a Binary Search
    Tree (BST).
    """
    post = []
    i = 0  # Our encapsulated pointer

    def pre_to_post(a, b):
        """Inner recursive function."""
        nonlocal i # Tell Python to use the 'i' from the outer scope

        if i == len(pre):
            return

        # If-statement checks if the current element is outside the
        # valid BST range (a, b) for the current subtree.
        if pre[i] <= a or pre[i] >= b:
            return

        r = pre[i]
        i += 1

        pre_to_post(a, r) # Left subtree
        pre_to_post(r, b) # Right subtree
        post.append(r)

    # Start the recursion
    pre_to_post(-math.inf, math.inf)

    return post

if(__name__ == "__main__"):
    pre = [15, 6, 3, 2, 4, 7, 13, 9, 18, 17, 20]
    post = get_postorder(pre)
    assert(post == [2, 4, 3, 9, 13, 7, 6, 17, 20, 18, 15]) # example from lecture
```

(d) Die Aussage ist falsch. Für jeden binären Suchbaum mit paarweise verschiedenen Schlüsseln ist die In-Order Traversierung gerade die sortierte Reihenfolge der Schlüssel. Die In-Order Traversierung hängt also nur von der Schlüsselmenge ab, nicht von der Form des Baums.

Ein Gegenbeispiel erhält man mit den Schlüsseln $\{1, 2, 3\}$. Sei T der Baum mit Wurzel 2, linkem Kind 1 und rechtem Kind 3. Sei T' der Baum mit Wurzel 1, rechtem Kind 2 und rechtem Kind 3 von 2. Beide Bäume haben die In-Order Traversierung

$$(1, 2, 3),$$

sind aber offensichtlich nicht identisch.

Aufgabe 2: Gemeinsame Schlüssel zweier Suchbäume *(7 Punkte)*

Gegeben seien zwei binäre Suchbäume T_1 und T_2 mit insgesamt n_1 beziehungsweise n_2 Knoten. Alle Schlüssel innerhalb eines Baums seien paarweise verschieden. Gesucht sind alle Schlüssel, die in beiden Bäumen vorkommen.

Eine naheliegende Lösung wäre, für jeden Schlüssel aus T_1 eine Suche in T_2 durchzuführen. Diese Lösung ist jedoch zu langsam. Welche Laufzeit hätte diese Herangehensweise? *(2 Punkte)*

Beschreiben Sie einen Algorithmus, der alle gemeinsamen Schlüssel von T_1 und T_2 ausgibt. Ihr Algorithmus soll nur $O(n_1 + n_2)$ Zeit benötigen. Begründen Sie! *(5 Punkte)*

Musterlösung

Die naheliegende Lösung durchsucht für jeden Schlüssel aus T_1 den Baum T_2. Eine einzelne Suche in T_2 kostet im Worst Case $O(h_2)$, wobei h_2 die Höhe von T_2 ist. Da ein binärer Suchbaum nicht notwendigerweise balanciert ist, kann $h_2 = O(n_2)$ gelten. Für alle n_1 Schlüssel aus T_1 ergibt sich im Worst Case also

$$O(n_1 \cdot n_2).$$

Falls man zusätzlich annimmt, dass T_2 Höhe h_2 hat, kann man die Laufzeit genauer als $O(n_1 h_2)$ angeben.
Eine schnellere Lösung verwendet die Tatsache, dass die In-Order Traversierung eines binären Suchbaums die Schlüssel in sortierter Reihenfolge ausgibt. Wir erzeugen daher die In-Order Folgen beider Bäume und vergleichen sie anschließend wie beim Merge-Schritt von Mergesort.

```
CommonKeys(T1,T2):
    A = Inorder(T1)
    B = Inorder(T2)

    i = 0
    j = 0

    while i < len(A) and j < len(B):
        if A[i] == B[j]:
            output A[i]
            i = i + 1
            j = j + 1
        else if A[i] < B[j]:
            i = i + 1
        else:
            j = j + 1
```

Die Korrektheit folgt aus der Sortiertheit der beiden In-Order Folgen. Wenn $A[i] < B[j]$, dann kann $A[i]$ in B nicht mehr vorkommen, da alle noch nicht betrachteten Elemente von B mindestens $B[j]$ sind. Also kann $A[i]$ verworfen werden. Analog kann bei $A[i] > B[j]$ der Schlüssel $B[j]$ verworfen werden. Wenn $A[i] = B[j]$, wurde ein gemeinsamer Schlüssel gefunden und beide Zeiger werden weitergeschoben. Da alle Schlüssel innerhalb eines Baums paarweise verschieden sind, wird jeder gemeinsame Schlüssel genau einmal ausgegeben.
Die In-Order Traversierung von T_1 kostet $O(n_1)$ Zeit, die von T_2 kostet $O(n_2)$ Zeit. Der anschließende Merge-Schritt erhöht in jeder Iteration mindestens einen der beiden Zeiger und kostet daher ebenfalls $O(n_1 + n_2)$ Zeit. Insgesamt erhält man also

$$O(n_1 + n_2).$$

Alternativ kann man die In-Order Traversierungen mit zwei Iteratoren direkt über die Bäume durchführen. Dann bleibt die Laufzeit ebenfalls $O(n_1 + n_2)$, und man benötigt zusätzlich nur Speicher proportional zu den Höhen der beiden Bäume.