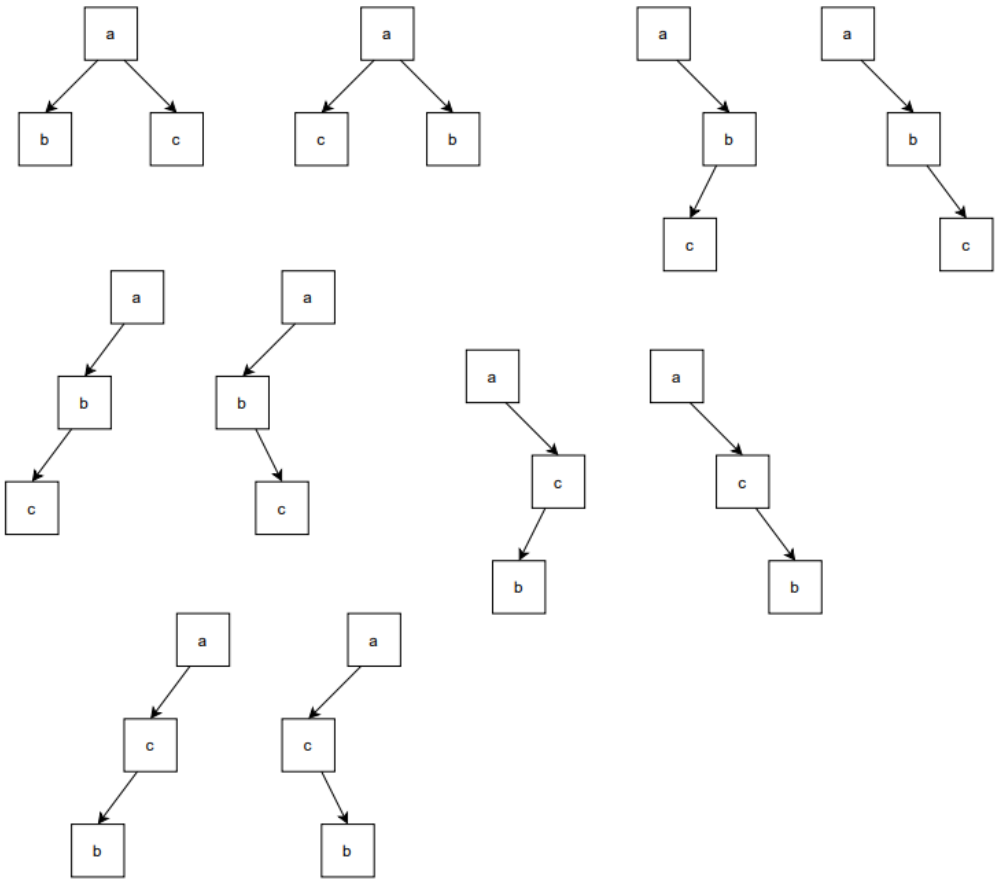
Algorithmen und Datenstrukturen - ueb07

Aufgabe 7.1

7.1.a)



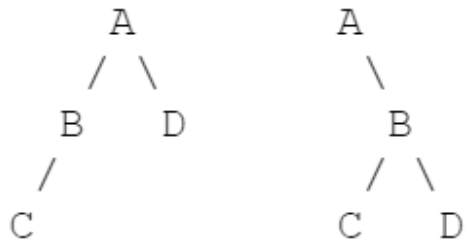
7.1.b)

```
start programm binaryTreeReconstructor(inorder, postorder)
if inorder == null then
    return null

// until gibt alle Elemente der Liste bis zum Grenzwert zurück, exklusiv
// from gibt alle Elemente der Liste vom Grenzwert zum Ende zurück, inklusiv
left = binaryTreeReconstructor(inorder.until(postorder.letztesElement),
postorder.until((inorder.until(postorder.letztesElement)).next))

right = binaryTreeReconstructor(inorder.from(postorder.letztesElement),
postorder.from((inorder.until(postorder.letztesElement)).next.next.letztesElement).letztesElem
entEntfernen)

return new BinaryTree(left , postorder.letztesElement, right)
end programm

start programm postorder(PostOrderList: Aray of Type Node, start: Integer, end: Integer): Void
    if start > end then
        return NULL
    root := NodeList.GetLastElement
```

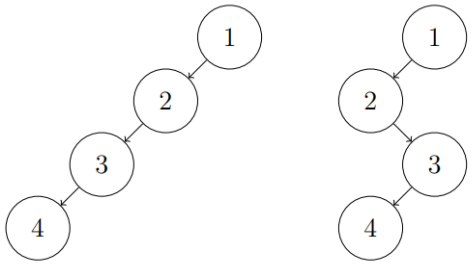
```
    var I := end

    while I >= start do
        if PostOrderListe[I] < root.Key
            break
        I = I - 1

    root.Right = postorder(PostOrderList, I+1, end-1)
    root.Left = postorder(PostOrderListe, start, I)
end programm
```

7.1.c)

Zu einer preorder- oder postorder-Sequenz kann es mehrere Bäume geben. Beispielsweise ist A - B - C - D die preorder-Sequenz zu folgenden zwei, nicht gleichen Bäumen.



Beispielsweise ist 4 - 3 - 2 - 1 die postorder-Sequenz zu folgenden zwei, nicht gleichen Bäumen.



7.1.d)

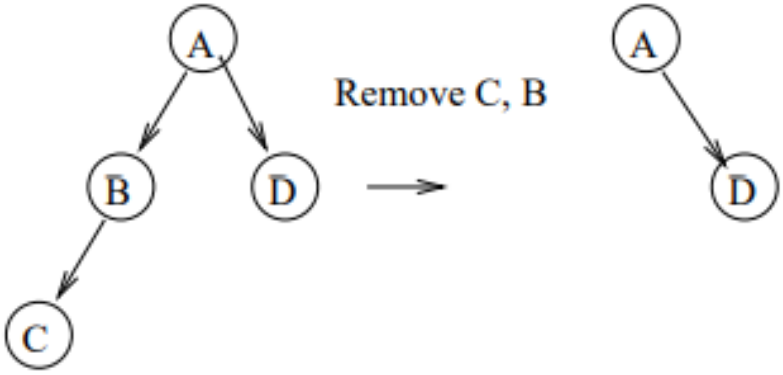
Behauptung: Sei b die Anzahl von Knoten mit Grad 2 und z die Anzahl von Blättern. Wir zeigen durch Induktion, dass in einem nicht-leeren Binärbaum folgende Eigenschaft gilt: b + 1 = z
Beweis durch vollständige Induktion:

1. Induktionsanfang: b = 0
A(0) = 0 + 1 = 1
Ein Binärbaum mit 0 Knoten von Grad 2 hat ein Blatt. Der Knoten selbst ist das Blatt.

2. Induktionsvoraussetzung:
A(b) ist wahr. Wir zeigen, dass A(b+1) wahr ist.

3. Induktionsschritt: b -> b+1
Angenommen wir haben (b+1) Knoten vom Grad 2, dann gibt es (b+2) Blätter.
A(b+1) = (b+1) + 1 = b + 2 = z

Man nehme ein Blatt und entferne den Elternknoten rekursiv (d.h. man entferne auch deren Elternknoten usw.) bis die Wurzel erreicht ist. Die Wurzel ist ein Knoten mit Grad 1, da eins
seiner Kinder entfernt wurde. Der Baum insgesamt hat damit einen Knoten mit Grad 2 weniger und deshalb auch ein Blatt weniger.



Wir haben folglich nur noch b Knoten vom Grad 2. Der Induktionsvoraussetzung zufolge existieren (b+1) Blätter.
Wenn wir nun sämtliche Knoten, die zuvor entfernt wurden, dem Baum wieder hinzufügen, dann fügen wir ein Blatt ein und haben wieder einen Knoten vom Grad 2. Deshalb haben wir (b+1) Knoten vom Grad zwei und (b+2) Blätter.
Die Aussage ist damit bewiesen.

7.1.e)

Behauptung: Sei L die Menge von Blättern in einem Binärbaum und l ein Element von L mit Höhe h in einem Binärbaum w(l) = 1/2^(h-1). Dann gilt:

$$\sum_{l \in L} w(l) <= 1$$

Beweis:

1. Induktionsanfang: h = 1
A(1) = 1/2^(1-1) = 1/2^0 = 1 <= 1

In diesem Fall hat der Baum nur einen Knoten (Höhe 1) und floglich nur ein Blatt. Der Knoten selbst ist das Blatt.

2. Induktionsvoraussetzung:
A(h) gilt. Wir zeigen, dass A(h+1) wahr ist.

3. Induktionsschritt: h -> h+1
A(h+1) = 1/2^((h+1) - 1) <= 1

F(T) bezeichnet die Anzahl von Blättern in einem Baum. Wenn T aus einem einzigen Knoten besteht, dann ist F(T) = 1/2^(1-1) = 1/2^0 = 1 (siehe oben).
Besteht T aus mehr als nur einem Knoten, dann kann man hieraus schließen dass ein rechter F(Tr) und linker F(Tl) Unterbaum existiert. Aus der Induktionsvoraussetzung folgt, da es sich auch um Binärbäume handelt, dass sowohl für F(Tr) als auch F(Tl) gilt:

$$\sum_{l \in L} w(l) <= 1$$

Lediglich die Tiefe jedes Blattknotens im Baum ist eins mehr (h+1) im Gegensatz zum ursprünglichen Baum T.

Hieraus folgt:

$$F(T) = \sum_{l \in L} 2^{-(h-1)} = \sum_{linkerUnterbaum} 2^{-(h-1)} + \sum_{rechterUnterbaum} 2^{-(h-1)} = 1/2 * (S(Tl) + S(Tr)) <= 1/2(1+1) <= 1$$

Diese Ungleichung stimmt allerdings nur, wenn es keine Knoten mit nur einem Kind gibt. Wenn es einen Knoten mit nur einem Kind gibt, stimmt die Ungleichung nicht, denn durch das Hinzufügen des zweiten Kindes wäre die Anzahl der Blätter höher als 1. Wenn es kein Knoten mit nur einem Kind gibt, dann können wir einen Knoten mit zwei Kindern auffinden und diese Kinder entfernen, womit wir einen neuen Baum erzeugen.

Aufgabe 7.3

Aufgabe 7.3.a)

Ein Heap der Höhe h kann insgesamt 2^(h) -1 Elemente aufnehmen, vorausgesetzt Level 1 ist der Wurzelknoten. Die Anzahl der Kinder wächst exponentiell durch 2^h. Da man die Knoten zuvor berücksichtigen muss für die Addition wir die -1 ergänzt.

Aufgabe 7.3.d)

Die Auswertung hat ergeben, dass der HeapStack für das Einfügen etwa so 3mal so lang(ca 150 : 50 ms), für das peeken immerhin nur 5mal so kurz(5 : 26) und für das Entfernen 7mal so lang(175:25) wie der normale Stack.

Die HeapQueue braucht für das das Einfügen etwas 2mal so lang(50 : 25) für das peeken gleich lang(8 : 8) und das
Entfernen 19mal so lang(219 : 12) wie die ArrayDeque

Dies liegt vorallem daran, dass die Heapstruktur bei dem Einfügen von zufälligen Zahlen sehr lange brauchen kann bis diese am richtigen Platz angekommen sind , und auch bei dem entfernen muss die Struktur neu organisiert werden, da der Wurzelknoten entfernt wird, nur das peeken geht bei der Heapstruktur sehr schnell, das das zu peekende Element immer an der Wurzel liegt und somit hier eine konstante Zeit benötigt wird.
Insbesondere braucht der Heapstack beim Einfügen im Durchschnitt länger als die HeapQueue, da der Heapstack immer an der Wurzel einfügt und so die Struktur neu angepasst werden muss während die HeapQueue am nächsten freien Blatt einfügt. Aber im Allgemeinen läuft sowohl das Einfügen als auch das entfernen in Logaritmischer Zeit ab.

Im Gegensatz dazu wird bei dem Stack einfach in das nächste freie Feld ,über eine Countervariable, eines Arrays eingefügt und wieder von dort entfernt womit diese Operationen in konstanter Laufzeit und somit schneller als logarithmisch von statten gehen. Die Peekfunktion des Stacks läuft zwar eigentlich auch konstant ab braucht aber durch interne Berechnungen etwas länger

Die ArrayDeque verhält sich ähnlich wie eine Liste mit Head und Tail nur dass dazwischen ein Array liegt. Trotzdem ermöglicht auch diese Struktur ein konstantes Einfügen im Gegensatz zu der logarithmischen Einfügezeit der HeapDeque.
Beim Peeken benötigen sowohl die ArrayDeque als auch die HeapDeque durch den Head bzw. die Wurzel nur eine konstante Zeit und bestätigen auch so das Endergebnis (beim Entfernen wird einfach der Head der ArrayDeque um 1 nach hinten im Array versetzt und somit wird das erste Element entfernt(vereinfacht))