Lösungshinweise

Test 4 - Dynamische Datenstrukturen und spezifische Algorithmen

Modul prog, WS23/24
Trier University of Applied Sciences
Informatik Fernstudium (M.C.Sc.)

09.01.2024

Thorsten Suckow-Homberg, https://thorsten.suckow-homberg.de

Das Abrufdatum aller in diesem Dokument aufgeführten Webseiten war der 08.01.2024.

Die Lösungshinweise beinhalten nicht die Testaufgaben, bei denen eine bestimmte Programmausgabe nachvollzogen werden musste.

Um Konzepte des Lehrmaterials deutlicher von weiterführenden, für den Kurs weniger relevante Themen abzugrenzen, sind diese nun unter **Exkurs** in den jeweiligen Abschnitten zusammengefasst.

Inhaltsverzeichnis

1

Binärer Suchbaum

Lösung

Gesucht ist der Baum, der folgendes Kriterium erfüllt[1]:

> Für jeden Knoten p gilt: Die Schlüssel im linken Teilbaum von p sind sämtlich kleiner als der Schlüssel von p, und dieser ist wiederum kleiner als sämtliche Schlüssel im rechten Teilbaum von p. ([OW17a, 263, Hervorhebungen i.O.])

Da von den drei gegebenen Möglichkeiten nur eine Antwort richtig ist, können zwei Darstellungen ausgeschlossen werden. Das gilt zum einen für den Baum, dessen Knoten[2] *60* die *45* als direkten Nachfolger hat. Diese Schlüssel gehören zu dem rechten Teilbaum des Knotens *50*, folglich ist das Kriterium, sämtliche Schlüssel im *rechten Teilbaum* müssen größer als *50* sein, verletzt, denn $45 < 50$.

Genauso ist der Baum, der den Knoten *60* mit direktem Nachfolger *85* enthält, kein den Kriterien entsprechender Suchbaum: Da dieser Teilbaum der linke Teilbaum des Knotens *70* ist, muss für jeden Schlüssel innerhalb dieses Teilbaums gelten, dass er kleiner als *70* ist, aber $85 > 70$).

Der gesuchte Baum lässt sich folgendermaßen durch sukzessizes Einfügen der Schlüssel konstruieren:

Abb. 1.1. Einfügen des Schlüssels 50.

[1] im Skript (Teil 2) auf Seite 143 in Defintion 6.3.3.1
[2] im Folgenden wird "Noten mit dem Schlüssel *[ZAHL]*" zu "Knoten *[ZAHL]*" abgekürzt

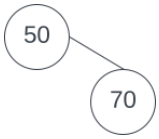

Abb. 1.2. Einfügen der Schlüssel 50, 70.

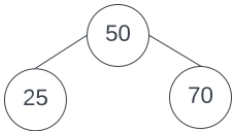

Abb. 1.3. Einfügen der Schlüssel 50, 70, 25.

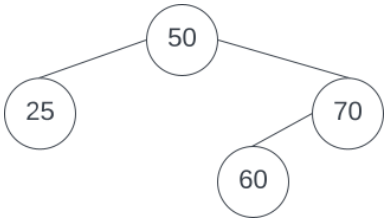

Abb. 1.4. Einfügen der Schlüssel 50, 70, 25, 60.

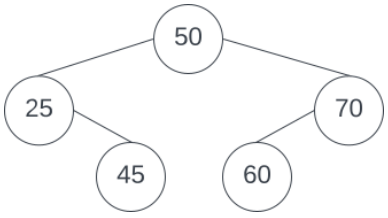

Abb. 1.5. Einfügen der Schlüssel 50, 70, 25, 60, 45.

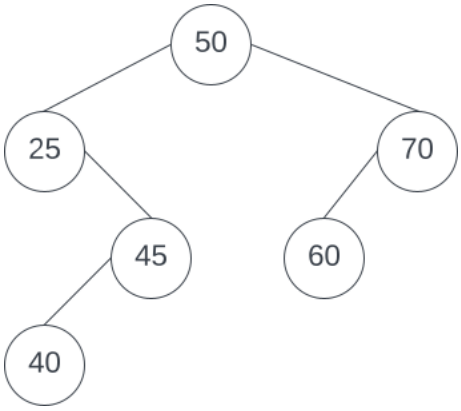

Abb. 1.6. Einfügen der Schlüssel 50, 70, 25, 60, 45, 40.

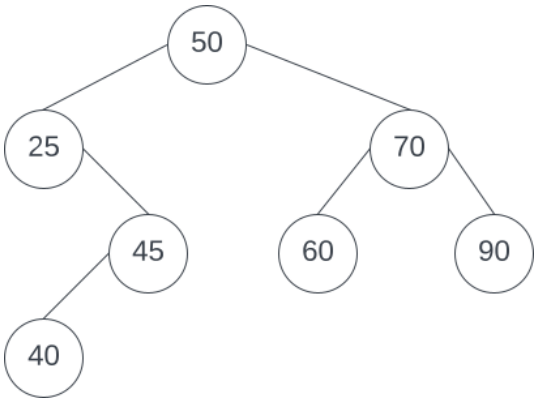

Abb. 1.7. Einfügen der Schlüssel 50, 70, 25, 60, 45, 40, 90.

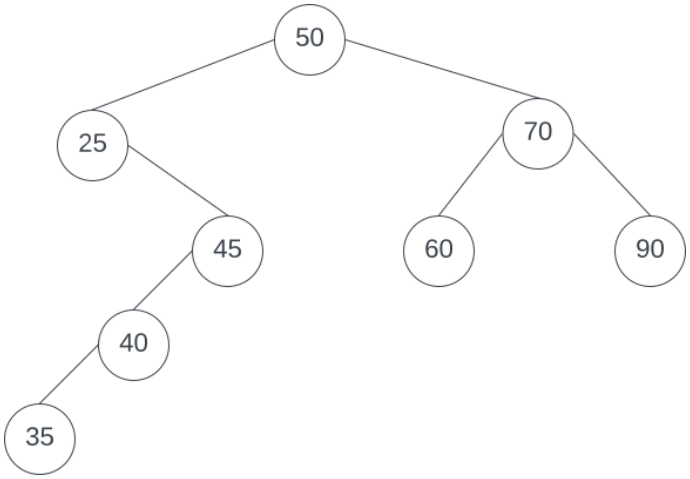

Abb. 1.8. Einfügen der Schlüssel 50, 70, 25, 60, 45, 40, 90, 35.

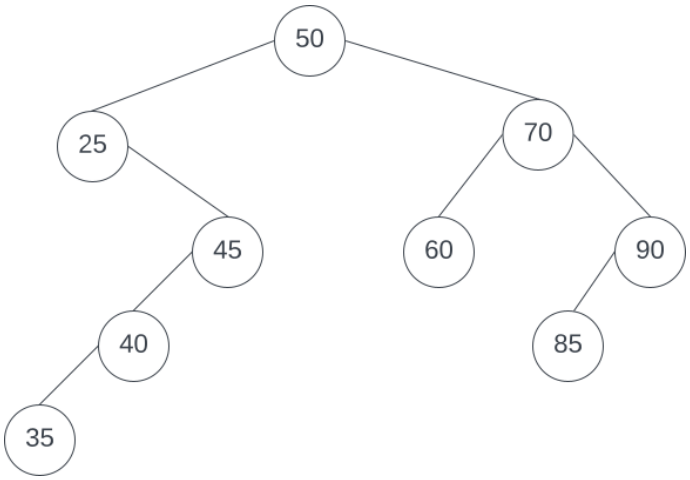

Abb. 1.9. Einfügen der Schlüssel 50, 70, 25, 60, 45, 40, 90, 35, 85.

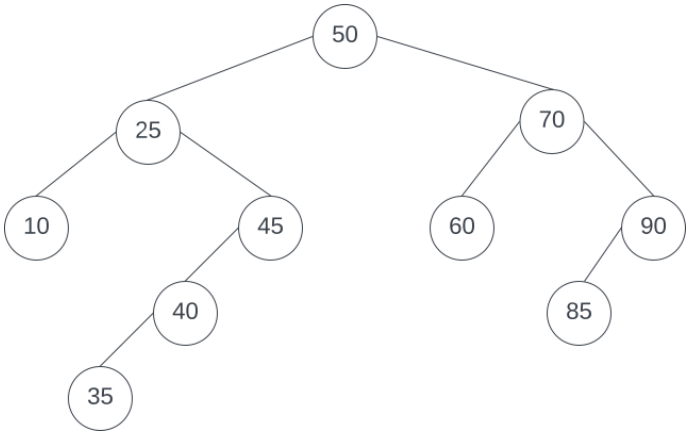

Abb. 1.10. Einfügen der Schlüssel 50, 70, 25, 60, 45, 40, 90, 35, 85, 10.

2

Datenstrukturen

Lösung

- Bei einer Liste können Elemente an einer beliebigen Position eingefügt oder entfernt werden.
- Eine Sequenz darf Duplikate enthalten.

Anmerkungen und Ergänzungen

Einfügen an beliebiger Position

Sowohl die **Queue** als auch der **Stack** sind Datenstrukturen[1], die im Gegensatz zu (linearen) Listen weder Einfügen noch Entfernen an beliebiger Position erlauben (vgl. [OW17b, 41]): Beide geben die Hinzufüge- und Löschenreihenfolge vor(vgl. [OW17b, 42])[2].

- Die **Queue** arbeitet nach dem **FIFO**-Prinzip (*first in, first out*): Das Element, das als Erstes in die Liste eingefügt wurde, wird auch als Erstes wieder entfernt. In der ASB-Aufgabe 3 wurden hierzu die Operationen `enqueue` (für Hinzufügen) bzw. `dequeue` (für Entfernen) implementiert. `java.util.Queue` stellt diese Operationen unter `add` bzw `remove` zur Verfügung [3]
- Der **Stack** arbeitet nach dem **LIFO**-Prinzip (*last in, first out*): Das Element, das zuletzt in die Liste eingefügt wurde, wird auch als Erstes wieder entfernt. In der ASB-Aufgabe 3 wurden hierzu die Operationen `push` (für Hinzufügen) bzw. `pop` (für Entfernen) implementiert. `java.util.Stack` stellt diese Operationen unter den gleichen Namen zur Verfügung [4]

[1] wiederum basierend auf der Datenstruktur *Liste*
[2] im Skript (Teil 2) in dem Abschnitt 5.2 ausführlich erklärt
[3] `https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/Queue.html`
[4] `https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/Stack.html`

Duplikate in einer Sequenz

Im Skript (Teil 2) auf Seite 70 wird dies als ein Kriterium für Sequenzen (repräsentiert durch eine **Liste** als Datenstruktur[5]) festgehalten [6].

[5] S. 71 im Skript (Teil2)

[6] als Beispiel: Ein Pfad in einem zyklischen Graphen, dargestellt als Liste von Knoten, kann Duplikate enthalten

3

delete

Lösung

Nach dem Löschen eines Knotens in einem Suchbaum soll der durch die Löschoperation entstehende Baum wieder ein Suchbaum sein, und die in Abschnitt 1 aufgeführten Kriterien erfüllen.

Die Ausführung der Löschoperation findet sich im Skript (Teil 2) auf Seite 147 ff., aus der sich die korrekte Antwort herleiten lässt.

Anmerkungen und Ergänzungen

Etwas ausführlicher beschreiben *Ottmann und Widmayer* die Knoten, die an die Stelle des gelöschten Knotens treten können, als den *symmetrischen Vorgänger* bzw. den *symmetrischen Nachfolger* (vgl. [OW17a, 288 f.])[1].

Der **symmetrische Vorgänger** u des Knotens p ist der Knoten im linken Teilbaum von p, der am weitesten rechts steht.

Der **symmetrische Nachfolger** v des Knotens p ist der Knoten im rechten Teilbaum von p, der am weitesten links steht[2].

Die folgende Darstellung (Abbildung 3.1) verdeutlicht den Zusammenhang von einem Knoten mit seinem symmetrischen Vorgänger bzw. Nachfolger.

In dem geg. Suchbaum gilt, dass der Betrag der Differenz von $x - y$ bzw. $x - z$ jeweils der kleinste Betrag aller möglichen Beträge ist, die aus der Menge der möglichen Schlüssel des linken bzw. rechten Teilbaums mit x gebildet werden können. Die beiden Knoten nähern sich mit ihrem Schlüsselwert also dem Schlüsselwert des Knotens p am ehesten an.

[1] in Abschnitt 5.1.2, ebenda, wird die *Inorder*-Durchlaufordnung auch als *Symmetrische Reihenfolge* bezeichnet.

[2] Da wir einen Suchbaum vorliegen haben, gilt als Voraussetzung für den *symmetrischen Nachfolger*, dass der Schlüssel z des Knotens v größer als der Schlüssel x des Knotens p ist, und für den *symmetrischen Vorgänger*, dass der Schlüssel y des Knotens u kleiner als der Schlüssel x des Knotens p ist

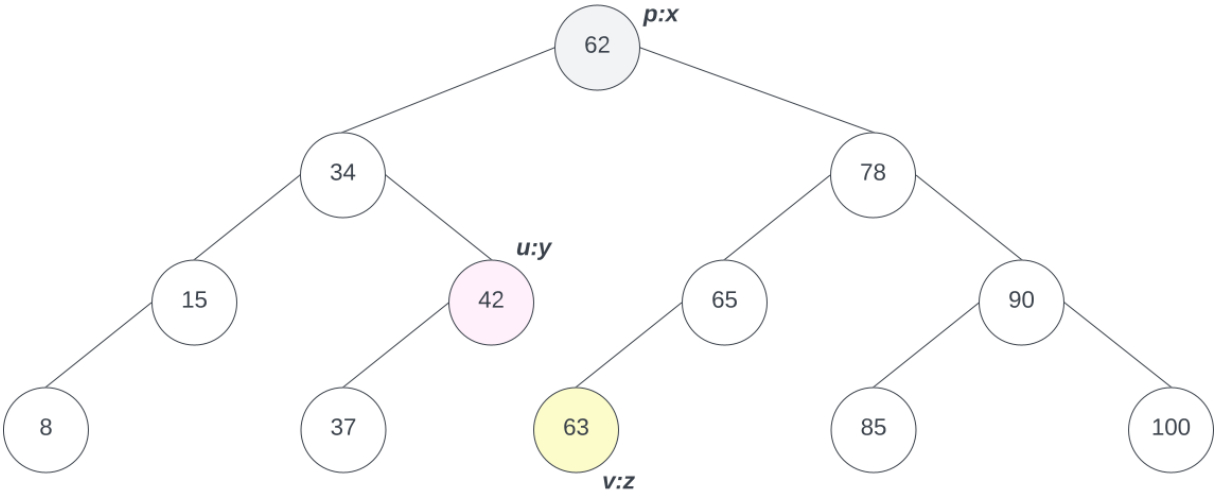

Abb. 3.1. Der symmetrische Vorgänger u und der symmetrische Nachfolger v des Knotens p.

Mit dem Wissen kann man den Knoten p durch den Knoten u oder den Knoten v ersetzen. Aufgrund der Tatsache, dass die gelöschten Knoten die Eigenschaft *symmetrischer Vorgänger* bzw *symmetrischer Nachfolger* besitzen, können die Knoten keinen bzw. einen linken oder rechten Nachfolger in Form eines Blattes bzw. inneren Knotens haben, weshalb sich das Ersetzen dieser weggefallenen Knoten als trivial erweist[3] [4]

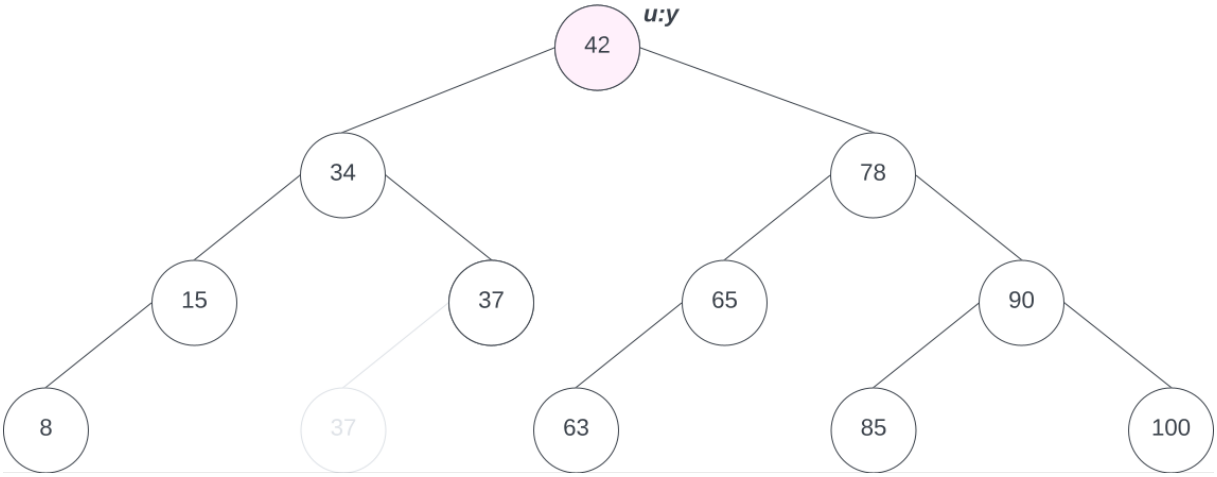

Abb. 3.2. Löschen des Knotens p und Ersetzen durch seinen symmetrische Vorgänger u.

[3] Hinweise hierzu im Skript (Teil 2) auf Seite 149 und in [OW17a, 269]: "Der Knoten q [v] ist der am weitesten links stehende innere Knoten im rechten Teilbaum von p und kann daher höchstens einen inneren Knoten als rechten Sohn haben."

[4] *Ottmann und Widmayer* weisen darauf hin, dass die für den Löschvorgang ausgewählte Strategie maßgeblich Einfluss auf die Struktur eines Baumes hat (vgl. [OW17a, 272]).

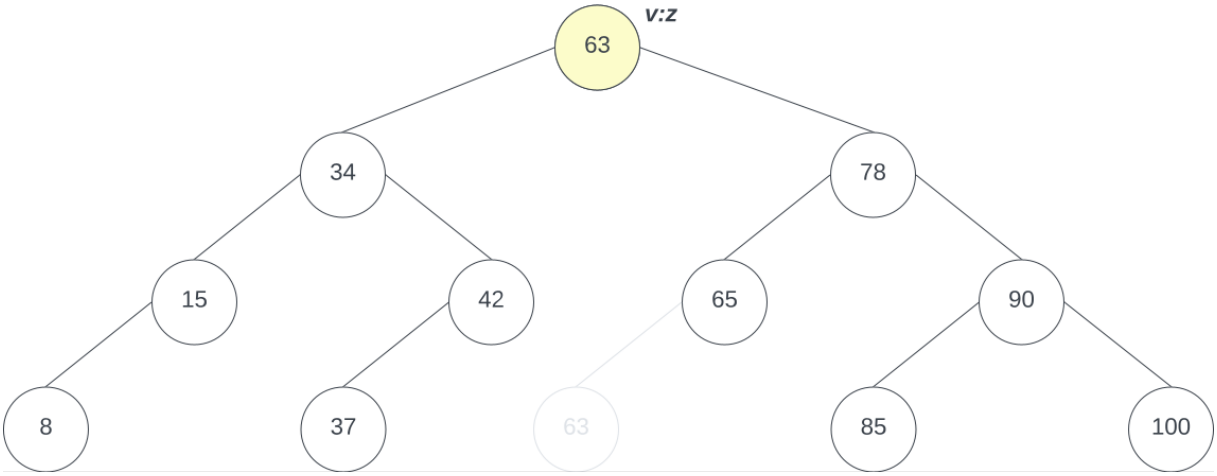

Abb. 3.3. Löschen des Knotens p und Ersetzen durch seinen symmetrische Nachfolger v.

4

Pre-, In- und Postorder

Lösung

- **Preorder**: 35, 20, 10, 12, 24, 42, 40, 38, 80
- **Inorder**: 10, 12, 20, 24, 35, 38, 40, 42, 80
- **Postorder**: 12, 10, 24, 20, 38, 40, 80, 42, 35

Anmerkungen und Ergänzungen

Die Durchlaufordnungen werden im Skript (Teil 2) auf Seite 106 f. erklärt.

Ottmann und Widmayer führen für die Durchlaufordnung in Binärbäumen die Bezeichnungen **Hauptreihenfolge**, **Nebenreihenfolge** und **Symmetrische Reihenfolge** an (vgl. [OW17a, 272]):

- **Hauptreihenfolge**: *Preorder*
- **Symmetrische Reihenfolge**: *Inorder*
- **Nebenreihenfolge** *Postorder*

Die Präfixe "Pre", "Post" und "In" dürfen als Eselsbrücke dienen, denn sie beziehen sich auf die Reihenfolge, in der die Knoten betrachtet werden:

- **Pre**: **Knoten zuerst**, dann linker Teilbaum, dann rechter Teilbaum (KLR[1])
- **In**: erst linker Teilbaum, **danach Knoten**, dann rechter Teilbaum (LKR)
- **Post**: erst linker Teilbaum, dann rechter Teilbaum, **zuletzt Knoten** (LRK)

[1] L = Linker Teilbaum, R = Rechter Teilbaum, K = Knoten

5

Reihenfolge-Kombinationen

Lösung

- Preorder- und Inorder-Reihenfolge
- Postorder- und Inorder-Reihenfolge

Anmerkungen und Ergänzungen

Zwei unterschiedliche Bäume seien wie in Abbildung 5.1 gegeben, wobei der linke Baum den Knoten B als linken Nachfolger hat, der rechte Baum den gleichen Knoten als rechten Nachfolger. Die Preorder-Darstellung beider Bäume lautet AB, die Postorder-Darstellung lautet BA. Folglich lässt sich nicht jeder binäre Baum allein durch die Kenntnis seiner Pre- und Postorder-Darstellung *eindeutig* rekonstruieren.

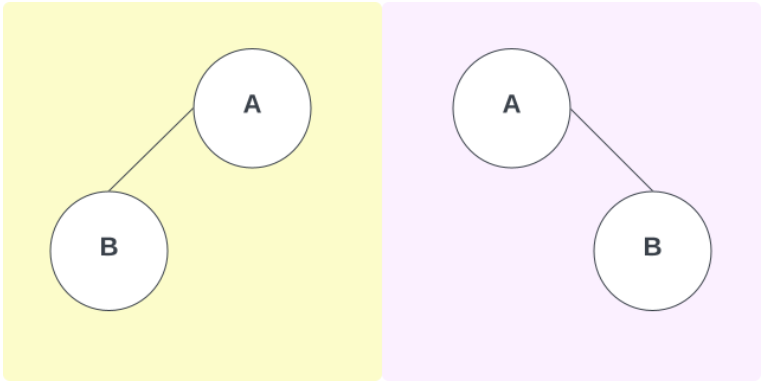

Abb. 5.1. Beide Bäume besitzen die Preorder-Darstellung AB und die Postorder-Darstellung BA, eine eindeutige Rekonstruktion ist nicht möglich.

Knuth weist in [Knu97, 564] darauf hin, das sich ein *vollständiger Baum*[1] tatsächlich durch seine Pre- und Postorder-Darstellung rekonstruieren läßt (s. a. Ab-

[1] Zur Definition "vollständiger Baum" siehe Abschnitt 6.

bildungen 5.2, 5.3 und 5.4)[2]. Ebenda zeigt er, dass Pre- oder Postorder zusammen mit Inorder ausreichen, um einen binären Baum eindeutig zu rekonstruieren.

> From the preorder, the root is known; then from the inorder, we know the left subtree and the right subtree; and in fact we know the preorder and inorder of the nodes in the latter subtrees. [...] Similarly, postorder and inorder together characterize the structure. ([Knu97, 564, 7.])

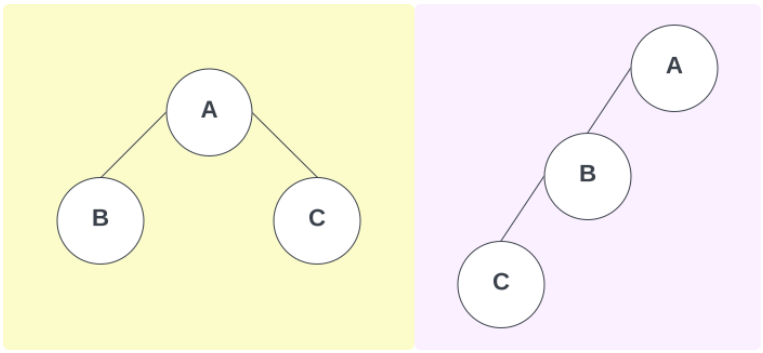

Abb. 5.2. Auch für einen vollständigen Baum (links) reicht seine Preorder-Darstellung zur eindeutigen Rekonstruktion nicht aus. Der Baum auf der rechten Seite besitzt dieselbe Preorder-Reihenfolge (*ABC*).

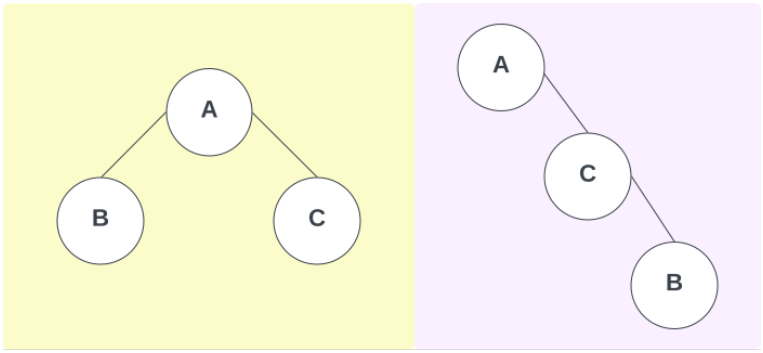

Abb. 5.3. Der vollständige Baum (links) mit der Postorder-Darstellung *BCA* und sein Pendant mit derselben Postorder-Reihenfolge.

Weitere Ausführungen zur Rekonstruktion von Binärbäumen finden sich bspw. bei *Cameron, Bhattacharya und Merks* [CBM89].

[2] "If all nonterminal nodes of a binary tree have *both* branches are nonempty, its structure *is* characterized by preorder and postorder." [Knu97, 564, 7., Hervorhebungen i.O.]

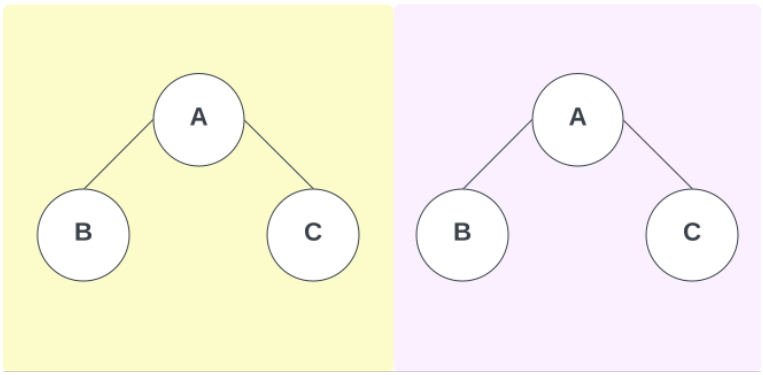

Abb. 5.4. Mit der Kenntnis von Pre- **und** Postorder-Reihenfolge (ABC und BCA) lässt sich ein vollständiger Baum eindeutig rekonstruieren.

6

Eigenschaften binärer Bäume

Lösung

- Der Grad eines Blattes ist immer 0.
- Ein Baum ist vollständig, falls alle inneren Knoten zwei direkte Nachfolger haben und alle Blätter die gleiche Tiefe aufweisen.

Anmerkungen und Ergänzungen

Der Grad jeden Knotens ist 2.

In einem binären Baum kann ein einzelner Knoten maximal zwei Nachfolger besitzen (vgl. [OW17a, 259])[1]. Demnach gilt: Der Grad eines Knotens in einem binären Baum ist höchstens 2.

Der Grad eines Knotens ist die Anzahl all seiner transitiv folgenden Knoten.

Eine *transitive Relation*[2] kann man sich auch als Graph vorstellen (vgl. [Hof22, 46]): Existiert eine Verbindung von einem Knoten A zu einem Knoten C, und eine Verbindung vom Knoten C zu Knoten D, so ist D ein transitiver Nachfolger von A (s. Abbildung 6.1).

Mit *Grad eines Knotens* (auch *Rang*(vgl. [OW17a, 260])) wird die Anzahl seiner direkten Nachfolger bezeichnet[3].

Der Grad eines Blattes ist immer 0.

Ein Knoten in einem Baum ist entweder ein *innerer Knoten* oder ein *Blatt*. Ein *innerer Knoten* ist ein Knoten, der mindestens einen Nachfolger besitzt. Ein *Blatt* ist ein Knoten, der keine Nachfolger hat, und deshalb den Grad 0 besitzt[4]:

[1] das Skript (Teil 2) weist ausdrücklich auf Seite 101 darauf hin.
[2] $\forall x \in X : \forall y \in X : \forall z \in X : xRy \land yRz \implies xRz$, mit X als Menge und R als Relation
[3] siehe hierzu auch im Skript (Teil 2) Seite 101
[4] auch in diesem Zusammenhang ist das Skript (Teil 2) auf Seite 101 sehr aussagekräftig.

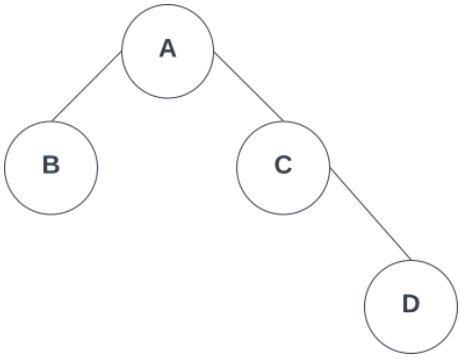

Abb. 6.1. Ein *transitiver Nachfolger* von A ist D. Nur B und C sind *direkte Nachfolger* von A.

> Da die Menge der Knoten eines Baumes stets als endlich vorausgesetzt wird, muss es Knoten geben, die keine Söhne haben. Diese Knoten werden üblicherweise als *Blätter* bezeichnet; alle anderen Knoten nennt man *innere Knoten*. ([OW17a, 259, Hervorhebungen i.O.])

Ein innerer Knoten ist ein Knoten mit einem Grad > 1.

Mit den bis hierhin aufgeführten Kriterien bzgl. Grad eines Knotens darf man schließen, dass es richtigerweise "mit einem Grad >= 1" heißen muss[5].

Ein Baum ist vollständig, falls alle inneren Knoten zwei direkte Nachfolger haben und alle Blätter die gleiche Tiefe aufweisen.

Bestätigt u.a. durch *Ottmann und Widmayer*:

> Ein Baum heißt vollständig, wenn er auf jedem *Niveau* die maximal mögliche Knotenzahl hat und sämtliche Blätter dieselbe Tiefe haben. ([OW17a, 261, Hervorhebungen i.O.])

Mit *Niveau* werden bei *Ottmann und Widmayer* (ebenda) Knoten gleicher Tiefe zusammengefasst.

Jedes Blatt eines Baumes hat dieselbe Höhe.

Als *Höhe* eines Baumes versteht man den maximalen Abstand eines Blattes von der Wurzel.

Der Abstand eines Knotens (und damit auch Blattes) von der Wurzel wird durch seine *Tiefe* ausgedrückt[6].

Damit jedes Blatt in einem Baum dieselbe Tiefe hat wie alle in dem Baum vorhandenen Blätter, muss der Baum *vollständig sein*. Richtigerweise müßte es also heißen: Jedes Blatt eines vollständigen Baumes hat dieselbe Höhe.

[5] so auch im Skript (Teil 2) auf Seite 101 vermerkt.
[6] dargestellt im Skript (Teil 2) auf Seite 102

7

Einfügen in einen Suchbaum II

Lösung

Der gesuchte Baum lässt sich mit den in Abschnitt 1 aufgeführten Kriterien durch sukzessives Einfügen der Schlüssel konstruieren (s. Abbildung 7.1):

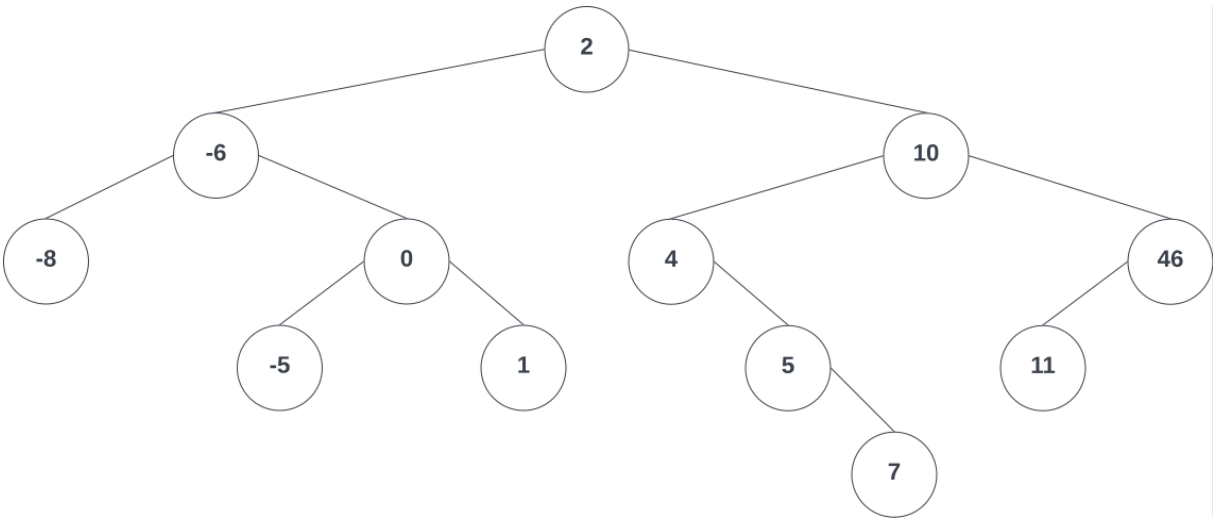

Abb. 7.1. Der Suchbaum nach Einfügen der Folge $2, 10, -6, 4, 46, 0, 5, -5, 7, -8, 11, 1$.

8

Verkettete Listen und Arrayimplementierung

Lösung

- Einfügeoperationen können bei der Arrayimplementierung einen höheren Aufwand erfordern als bei einer verketteten Implementierung.

Anmerkungen und Ergänzungen

Löschoperationen der einfach verketteten Listenimplementierug arbeiten immer mit gleichem Aufwand wie Löschoperationen der Arrayimplementierung.

Löschoperationen der Arrayimplementierung[1] erfordern unter Umständen ein Verschieben nachfolgender Elemente um eine Position nach links in dem Array. Wird bspw. das Element an Position 0 gelöscht, hat diese Operation den Aufwand $O(1)$, wenn die Liste nur einen Eintrag enthält. Besitzt hingegen die Liste eine Größe von n Einträgen, müssen die Positionen der verbliebenen $n-1$ Einträge aktualisiert werden[2]. Wird hingegen das Element an der letzten Position gelöscht, ist kein Verschieben nachfolgender Elemente nötig, der Aufwand ist hier mit $O(1)$ konstant.

Löschoperationen in einer einfach verketteten Liste besitzen i.d.R. eine mittlere Laufzeit von $O(n)$, da in der Liste zu dem Vorgängerelement des zu löschenden Elements gelaufen werden muß, um dessen Zeiger zu aktualisieren, und zwar auf den Nachfolger des zu löschenden Elements[3]. Bei n Einträgen in der Liste ist dann Löschen des letzten Elements mit einem Aufwand von $O(n)$ verbunden.

Einfügeoperationen können bei der Arrayimplementierung einen höheren Aufwand erfordern als bei einer verketteten Implementierung.

Einfügeoperationen bei der Arrayimplementierung erfordern nicht nur ein Verschieben von Folgeelementen, sondern ggfl. auch eine Anpassung der Größe des

[1] bei *Ottmann und Widmayer* als "Sequenziell gespeicherte lineare Liste" bezeichnet (vgl. [OW17b, 30 ff.])

[2] s. Skript (Teil 2) S. 89 "Nachteile der Darstellung als Array"

[3] Skript (Teil 2) S. 82. Auf Seite 83 ist eine alternative Realisierung angegeben, in der Löschoperationen in $O(1)$ möglich sind.

Arrays[4]. Einfügeoperationen in verketteten Listen benötigen hingegen eine konstante Laufzeit[5].

Die doppelt verkettete Liste und die Arrayimplementierung müssen die Positions- und Zeigerwerte von jedem Element nach jeder Einfüge- und Löschoperation aktualisieren.

Bei der Einfügeoperation in einer doppelt verketteten Liste werden die Zeiger eines Vorgängerelements und des eingefügten Elements aktualisiert. Der Rest der Liste bleibt unberührt[6]. Auch bei der Arrayimplementierung muss nicht in jedem Fall jedes Element aktualisiert werden[7]

Das Einfügen am Beginn des Arrays benötigt immer Zeit $O(1)$.

Wie bis hierher gesehen, bedingt das Einfügen eines Elements am Beginn eines Arrays eine Laufzeit von $O(n)$, falls $n-1$ Folgeelemente verschoben werden müssen. Muss das Array vergrößert werden, erhöht das auch die Laufzeit[8].

Falls bereits $(n-1)$-Elemente im Array liegen, dann benötigt die Einfügeoperation am Schluss Zeit $O(n)$.

Die Einfügeoperation im Array am Schluss des Arrays ist in konstanter Zeit $O(1)$ möglich, da keine Elemente verschoben werden müssen.

[4] s. Skript (Teil 2) S. 89 "Nachteile der Darstellung als Array". Der Fall wird bei *Ottmann und Widmayer* in [OW17b, 32] nicht berücksichtigt: Ist die Liste bei der Einfügeoperation bereits voll, wird ein Fehler ausgegeben.

[5] wenn das Element an der Zielposition erst ermittelt werden muss, ändert sich die Laufzeit zu $O(n)$.

[6] das ist allerdings von der Implementierung abhängig. Sollte bspw. noch ein Element existieren, welches das Ende der Liste repräsentiert, muss dieses ggfl. noch mit aktualisiert werden (vgl. [OW17b, 35 ff.]).

[7] im wort case müssen $n-1$ Folgeelemente aktualisiert werden, bei einer Einfügeoperation muss evtl. noch das Array vergrößert werden.

[8] mit einer neuen Größe von m freien Plätzen entspricht das einer Laufzeit von $O(m)$. Siehe hierzu Skript (Teil 2) S. 89 "Nachteile der Darstellung als Array".

9

Hashfunktion

Lösung

Die Schlüsselfolge $5, 28, 14, 24, 19, 27, 23, 17, 13$ soll mit der Hashfunktion

$$h(k) = h_0(k) = k \mod 9 \tag{9.1}$$

in eine Hashtabelle eingefügt werden.

Kollisionen sollen durch lineares Sondieren[1] behandelt werden, die Funktion hierzu lautet

$$h_i(k) = (h_0(k) + i) \mod 9 \tag{9.2}$$

Im Folgenden werden die Schlüssel der Reihenfolge an die Hashfunktionen 9.1 übergeben. Bei einer Kollision übernimmt die Sondierungsfunktion 9.2 die Berechnung der Speicherzelle (s. Tabelle 9.1):

[1] siehe [OW17c, 205], außerdem im Skript (Teil 2) S. 135 ff.

Schlüssel	$h_i(k)$	**Ergebnis**	**Kollision**	**Speicherzelle**
5	$h_0(5) = 5 \mod 9$	5	–	5
28	$h_0(28) = 28 \mod 9$	1	–	1
14	$h_0(14) = 14 \mod 9$	5	5	
	$h_1(14) = (14 \mod 9 + 1) \mod 9$	6	–	6
24	$h_0(24) = 24 \mod 9$	6	6	
	$h_1(24) = (24 \mod 9 + 1) \mod 9$	7	–	7
19	$h_0(19) = 19 \mod 9$	1	1	
	$h_1(19) = (19 \mod 9 + 1) \mod 9$	2	–	2
27	$h_0(27) = 27 \mod 9$	0	–	0
23	$h_0(23) = 23 \mod 9$	5	5	
	$h_1(23) = (23 \mod 9 + 1) \mod 9$	6	6	
	$h_2(23) = (23 \mod 9 + 2) \mod 9$	7	7	
	$h_3(23) = (23 \mod 9 + 3) \mod 9$	8	–	8
17	$h_0(17) = 17 \mod 9$	8	8	
	$h_1(17) = (17 \mod 9 + 1) \mod 9$	0	0	
	$h_2(17) = (17 \mod 9 + 2) \mod 9$	1	1	
	$h_3(17) = (17 \mod 9 + 3) \mod 9$	2	2	
	$h_4(17) = (17 \mod 9 + 4) \mod 9$	3	–	3
13	$h_0(13) = 13 \mod 9$	4	–	4

Tabelle 9.1. Speicherzellenbelegung für die Schlüsselfolge $5, 28, 14, 24, 19, 27, 23, 17, 13$ unter Verwendung der Hashfunktion 9.1 und Sondierungsfunktion 9.2. Es treten insgesamt 10 Kollisionen auf.

Literaturverzeichnis

[CBM89] R.D. Cameron, B.K. Bhattacharya und E.A.T. Merks. „Efficient reconstruction of binary trees from their traversals“. In: *Applied Mathematics Letters* 2.1 (1989), S. 79–82. ISSN: 0893-9659. DOI: https://doi.org/10.1016/0893-9659(89)90122-5. URL: https://www.sciencedirect.com/science/article/pii/0893965989901225.

[Hof22] Dirk W. Hoffmann. *Theoretische Informatik*. Carl Hanser Verlag, 2022. ISBN: 978-3446470293.

[Knu97] Donald E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. USA: Addison Wesley Longman Publishing Co., Inc., 1997. ISBN: 0201896834.

[OW17a] Thomas Ottmann und Peter Widmayer. „Bäume“. In: *Algorithmen und Datenstrukturen*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, S. 259–402. ISBN: 978-3-662-55650-4. DOI: 10.1007/978-3-662-55650-4_5. URL: https://doi.org/10.1007/978-3-662-55650-4_5.

[OW17b] Thomas Ottmann und Peter Widmayer. „Grundlagen“. In: *Algorithmen und Datenstrukturen*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, S. 1–78. ISBN: 978-3-662-55650-4. DOI: 10.1007/978-3-662-55650-4_1. URL: https://doi.org/10.1007/978-3-662-55650-4_1.

[OW17c] Thomas Ottmann und Peter Widmayer. „Hashverfahren“. In: *Algorithmen und Datenstrukturen*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, S. 191–258. ISBN: 978-3-662-55650-4. DOI: 10.1007/978-3-662-55650-4_4. URL: https://doi.org/10.1007/978-3-662-55650-4_4.