Lösungsvorschlag Prüfung WS 2020/21 (Java & DsAlg)

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

22.03.2024
Thorsten Suckow-Homberg, `https://thorsten.suckow-homberg.de`

Inhaltsverzeichnis

1 Klassen und Objekte **1**
- 1.1 Lösungsvorschlag . . . 1
- 1.2 Anmerkung und Ergänzungen . . . 2

2 Vererbung **4**
- 2.1 Lösungsvorschlag . . . 4
- 2.2 Anmerkung und Ergänzungen . . . 4

3 Felder, Anweisungen und Ausdrücke **6**
- 3.1 Lösungsvorschlag . . . 6
- 3.2 Anmerkung und Ergänzungen . . . 6

4 Zeichenketten **8**
- 4.1 Lösungsvorschlag . . . 8
- 4.2 Anmerkung und Ergänzungen . . . 8

5 Binäre Bäume **10**
- 5.1 Lösungsvorschlag a . . . 10
  - 5.1.1 Anmerkung und Ergänzungen . . . 10
- 5.2 Lösungsvorschlag b . . . 11
  - 5.2.1 Anmerkung und Ergänzungen . . . 11
- 5.3 Lösungsvorschlag c . . . 13
  - 5.3.1 Anmerkung und Ergänzungen . . . 13

6 Verkette Liste **17**
- 6.1 Lösungsvorschlag . . . 17
- 6.2 Anmerkung und Ergänzungen . . . 17

7 Sortierverfahren **21**
- 7.1 Lösungsvorschlag . . . 21
  - 7.1.1 Aufgabenteil a) . . . 21
  - 7.1.2 Aufgabenteil b) . . . 22
  - 7.1.3 Aufgabenteil c) . . . 22

7.1.4 Aufgabenteil d) 25
7.1.5 Aufgabenteil e) 25

8 Hashing **26**
8.1 Lösungsvorschlag 26

9 Rekursion **29**
9.1 Lösungsvorschlag 29

10 O-Notation **31**
10.1 Lösungsvorschlag 31
10.1.1 Teil 1 31
10.1.2 Teil 2 31
10.1.3 Teil 2 31

1

Klassen und Objekte

1.1 Lösungsvorschlag

Der folgenden Lösungsvorschlag enthält eine komplette Ausimplementierung der Klasse:

```
1  public class Auto {
2
3      private int kmStand;
4      private double verbrauch;
5      private double tankVolumen;
6      private double kraftstoffVorrat;
7
8      public Auto(int kmStand, double verbrauch, double tankVolumen,
9          double kraftstoffVorrat) {
10         this.kmStand = kmStand;
11         this.verbrauch = verbrauch;
12         this.tankVolumen = tankVolumen;
13         this.kraftstoffVorrat = kraftstoffVorrat;
14     }
15
16     public String toString() {
17         return "verbrauch: " + verbrauch +
18             "; tankVolumen: " + tankVolumen +
19             "; kmStand: " + kmStand +
20             "; kraftstoffVorrat: " + kraftstoffVorrat;
21     }
22
23     public void fahren(int km) {
24
25         if (km <= 0) {
26             return;
27         }
28
```

```
29          double proKm = verbrauch / 100;
30
31          int maxReichweite = (int) (kraftstoffVorrat / proKm);
32
33          if (maxReichweite < km) {
34              kmStand += maxReichweite;
35              kraftstoffVorrat = 0;
36          } else {
37              kmStand += km;
38              kraftstoffVorrat -= proKm * km;
39          }
40      }
41
42      public void tanken(double liter) {
43          if (liter <= 0) {
44              return;
45          }
46
47          if (liter + kraftstoffVorrat >= tankVolumen) {
48              kraftstoffVorrat = tankVolumen;
49          } else {
50              kraftstoffVorrat += liter;
51          }
52      }
53
54      public static void main (String[] args) {
55          Auto gogoMobil = new Auto(0, 5.0, 50, 30);
56          gogoMobil.fahren(300);
57          gogoMobil.tanken(25);
58      }
59  }
```

1.2 Anmerkung und Ergänzungen

- Es ist darauf zu achten, die Aufgabenstellung hinsichtlich der Parameterbedingungen aufmerksam zu lesen.
  So ergibt sich bspw. für `tanken()`, dass die Methode nichts tut, wenn eine 0 oder ein *negativer* Wert übergeben wird - die Überprüfung der übergebenen Argumente resultiert in diesen Fällen in einer direkten Rückkehr aus der Methode.
  Gleiches gilt für `fahren()`, Werte für `int km` ≤ 0 wirken sich nicht auf eine Änderung des Objekt-Zustands aus.
- Variablenüberdeckung ist o.k. wenn man bedenkt, dass auch das ASB-System diesbzgl. keinen checkstyle-Fehler produziert hat (s. `Flugzeug`-Aufgabe).

- **reelle Zahl** - `float` oder `double` benutzen? In Java ist `double` der Standarddatentyp für Fließkommazahlen[1]. Ist der Datentyp nicht ausdrücklich beschrieben, sollte `double` ohne Einschränkung verwendet werden können.

[1] Konstanten wie 123.45 werden im Quellcode automatisch als *double* behandelt; sollen sie als *float* behandelt werden, muss ein großes oder ein kleines "f" hintenangestellt werden: 1234.56*f* (vgl [Ull23, 124 f.])

2

Vererbung

2.1 Lösungsvorschlag

```
1  public class Omnibus extends Auto {
2      private int anzahlSitzplaetze;
3      public Omnibus(
4              int anzahlSitzplaetze,
5              int kmStand,
6              double verbrauch,
7              double tankVolumen,
8              double kraftstoffVorrat) {
9          super(kmStand, verbrauch, tankVolumen, kraftstoffVorrat);
10         if (anzahlSitzplaetze > 0) {
11             this.anzahlSitzplaetze = anzahlSitzplaetze;
12         }
13     }
14
15     public String toString() {
16         return "anzahlSitzplaetze: " + anzahlSitzplaetze +
17                 "; " + super.toString();
18     }
19 }
```

2.2 Anmerkung und Ergänzungen

- `Omnibus` ist eine *Spezialisierung* von `Auto`, umgekehrt ist `Auto` eine *Generalisierung* von `Omnibus` (s. Abbildung 2.1)
- Der Konstruktor wird um einen formalen Parameter `anzahlSitzplaetze` erweitert.
- Ein `Omnibus` ist ein `Auto`[1] und verhält sich auch so: Es kann alles tun, was ein `Auto` tun kann, und deshalb können alle Methoden auch in `Omnibus` wiederverwendet werden.

[1] *Fahrzeug* würde als Generalisierung vlt. etwas besser passen

- Die Vererbung schließt nicht die in `Auto` als `private` deklarierten Attribute mit ein - diese stehen der Klasse `Omnibus` nicht zur Verfügung. Ein Zugriff auf als `private` deklarierte Felder über Methoden, die **public**, **protected** oder **package-private**[2] sind, ist aber trotzdem möglich[3].
  Gleiches gilt für den Konstruktor, der die Initialisierung der Felder `kmStand`, `verbrauch`, `tankVolumen`, `kraftstoffVorrat` übernimmt.
- Der erste Aufruf in einem Konstruktor muss stets den Konstruktor der Elternklasse aufrufen.
  Fehlt solch ein expliziter Aufruf, erfolgt ein *impliziter* Aufruf durch den Compiler.
  Soll *explizit* der Konstruktor der Oberklasse aufgerufen werden, muss dieser Aufruf als erstes erfolgen.
  Erst danach darf das Attribut `anzahlSitzplaetze` initialisiert werden[4].
- Eine sinnvolle Initialisierung von `anzahlSitzplaetze` erfolgt mit Werten > 0. (standardmäßig erfolgt die Initialisierung von Objekt-Attributen vom Typ `int` mit dem Wert 0 (vgl. [Ull23, 127])).

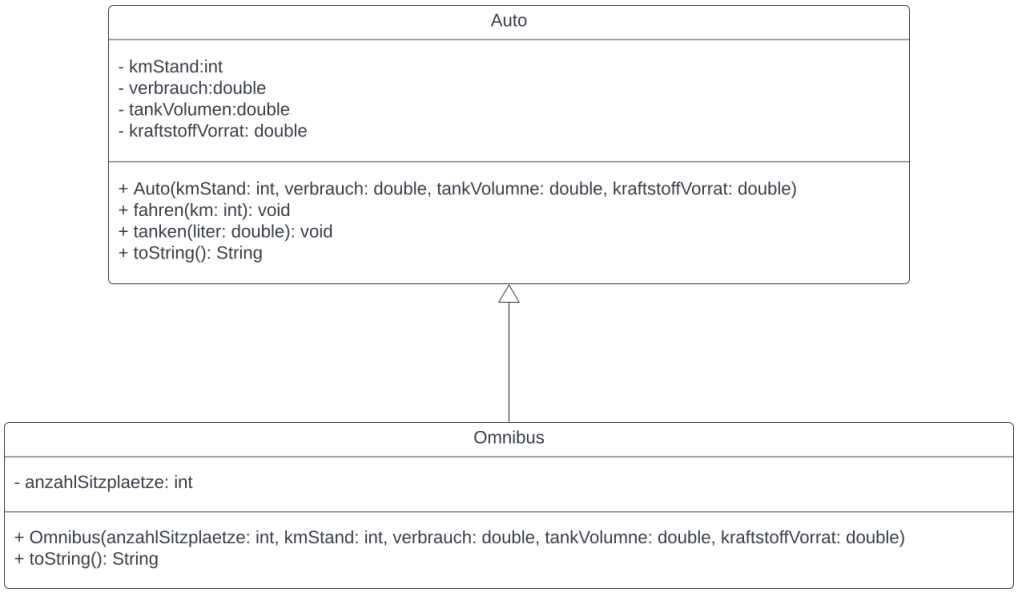

Abb. 2.1: Das UML-Klassendiagramm für die Generalisierungsbeziehung zwischen *Omnibus* und *Auto*

[2] *package access*, s. `https://docs.oracle.com/javase/specs/jls/se21/html/jls-6.html#d5e11105` - abgerufen 8.2.2024

[3] s.a. "Private Members in a Superclass": `https://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html` - abgerufen 8.2.2024

[4] Nachdem Aufrufe zu *this()* bzw. `-NoValue-` erfolgt sind und die Initialisierung der Attribute des zu erzeugenden Objektes abgeschlossen sind, wird die übrige Konstruktorimplementierung abgearbeitet. S.a. "12.5 Creation of New Class Instances": `https://docs.oracle.com/javase/specs/jls/se21/html/jls-12.html#jls-12.5` - abgerufen 8.2.2024

3

Felder, Anweisungen und Ausdrücke

3.1 Lösungsvorschlag

- `3, 6, 2, 8, 5`
- `3, -12, -6, 32, 25`
- `3, -2, 8, 5, -6`

3.2 Anmerkung und Ergänzungen

Eckige Klammern sollten bei der Variablendeklaration bzw. bei der Typdeklaration von Feldern hinter dem Datentyp stehen, *nicht* hinter dem Variablennamen[1]:

```
// statt
int numbers[]
// besser
int[] numbers
```

- Die Lesbarkeit wird verbessert: Die eckigen Klammern direkt hinter dem Typ weisen darauf hin, dass die Variable/der Parameter ein **Feld** von Werten des entsprechenden Typs ist.
- Bei folgendem Code ist nicht direkt ersichtlich, was gemeint ist:
  ```
  int[] vector, matrix[];
  ```
  Die Schreibweise ist äquivalent zu
  ```
  int vector[], matrix[][];
  // bzw.
  int[] vector;
  int[][] matrix;
  ```
  Die Sprachspezifikationen weisen daraufhin, dass diese Schreibweise nicht empfohlen wird:
  > We do not recommend "mixed notation" in array variable declarations, where bracket pairs appear on both the type and in declarators; nor in method declarations, where bracket pairs appear both before and after

[1] s.a. `https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html` - abgerufen 9.2.2024

> the formal parameter list. ("10.2. Array Variables": `https://docs.oracle.com/javase/specs/jls/se21/html/jls-10.html#jls-10.2` - abgerufen 9.2.2024)

Gegenüber folgender Schreibweise

```
int[] numbers = new int[]{1, 2, 3, 4};
```

ist folgende Schreibweise

```
int[] numbers = {1, 2, 3, 4};
```

übersichtlicher, redundante Informationen sind weggefallen.
Beides ist erlaubt und führt am Ende zur Initialisierung des `int`-Arrays `[1, 2, 3, 4]`.
Eine Übergabe von `{1, 2, 3, 4}` als Argument an eine Methode, deren Signatur bspw.

```
foo(int[])
```

entspricht, ist dagegen nicht möglich; hier muss `new int[]{1, 2, 3, 4}` verwendet werden.

Bei mehrdimensionalen Arrays muss bei der Deklaration der lokalen Variable / des formalen Parameters immer die Anzahl der eckigen Klammern `[]` mit der Anzahl der Dimensionen des Arrays übereinstimmen.
Bei der Initialisierung eines mehrdimensionalen Arrays muss immer mindestens eine führende Dimension mit der Anzahl aufzunehmender Werte bestimmt werden:

```
// funktioniert nicht:
int[][] numbers = new int[][];
int[][] numbers = new int[][4];

// funktioniert:
int[][] numbers = new int[4][];
```

In Java sind mehrdimensionale Arrays **Arrays von Arrays** - sie müssen deshalb nicht rechteckig sein, wie das letzte Beispiel zeigt (vgl. [Ull23, 273 ff.]).

4

Zeichenketten

4.1 Lösungsvorschlag

```java
public static void main(String[] args) {
    int sum = 0;
    for (int i = 0; i < args.length; i++) {
        String arg = args[i];
        char c;
        int v;
        for (int j = 0; j < arg.length(); j++) {
            c = arg.charAt(j);
            v = c - '0';
            if (v >= 0 && v <= 9) {
                sum += v;
            }
        }
    }
    System.out.println("Ergebnis: " + sum);
}
```

4.2 Anmerkung und Ergänzungen

Der Datentyp `char` ist in Java ein ganzzahliger Datentyp[1], hat eine Länge von 16 Bit, ist vorzeichenlos (im Unterschied zu den anderen ganzzahligen Datentypen `byte`, `short`, `int`, `long`) und besitzt damit einen Wertebereich von $0 - 65.535$. Damit unterstützt `char` Unicode-Zeichen im hexadezimalen Bereich von $0x0000$ bis $0xFFFF$[2].

[1] "4.2.1. Integral Types and Values": https://docs.oracle.com/javase/specs/jls/se21/html/jls-4.htmljls-4.2.1 - abgerufen 10.2.2024

[2] "The Java Tutorials - Unicode": `https://docs.oracle.com/javase/tutorial/i18n/text/unicode.html`

`char`-Variablen können als Ganzzahlwert ausgegeben werden, was dem Dezimalwert des Unicode-Zeichens entspricht[3]:

```
int toInt = 'c'; // entspricht dem Integer-Wert 99
```

Mittels `charAt(pos: int):char` der Klasse `String`[4] kann ein einzelner *character* der `String`-Argumente dahingehend überprüft werden, ob dieser eine Ziffer repräsentiert.
Eine Ziffer bedeutet in diesem Fall ein *character* aus der Liste 0..9.
Dadurch, dass arithmetische Operationen auf dem Ganzzahl-Typ `char` möglich sind, ermittelt man für ein einzelnes Zeichen nun seine *relative Position* (oder auch **Distanz**) zu dem Zeichen `'0'` - indem man diesen Wert einfach von dem zu vergleichenden Zeichen abzieht:

```
int relC = 'c' - '0'; // 99 - 48 = 51
int relA = 'a' - '0'; // 97 - 48 = 49
int rel0 = '0' - '0'; // 48 - 48 = 0
int rel7 = '7' - '0'; // 55 - 48 = 7
```

Die Distanz liegt für die Ziffern 0..9 dann in genau diesem Bereich, wodurch man den numerischen Wert des `chars` enthält. Darüber läßt sich dann einfach die Summe berechnen.

[3] s. "List of Unicode characters": `https://en.wikipedia.org/wiki/List_of_Unicode_characters` - abgerufen 10.2.2024
[4] "Class String": `https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/String.html#charAt(int)` - abgerufen 10.2.2024

5

Binäre Bäume

5.1 Lösungsvorschlag a

- Im **worst-case** liegen Einfügeoperationen in einem binären Suchbaum in der Komplexitätsklasse $O(n)$
- Im **average-case** liegen Einfügeoperationen in einem binären Suchbaum in der Komplexitätsklasse $O(logn)$

Im

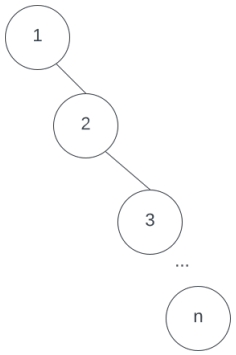

Abb. 5.1: Für einen entarteten binären Suchbaum liegen Einfügeoperationen im **worst-case** in der Komplexitätsklasse $O(n)$ (Quelle: eigene)

5.1.1 Anmerkung und Ergänzungen

In der Literatur wird manchmal die Definition **vollständiger** und **fast vollständiger** binärer Baum unterschiedlich gehandhabt. Bei *Ottmann und Widmayer* ist ein **vollständiger Baum** ein Baum, der auf jedem *Niveau*[1] die maximal mögliche Knotenzahl hat und sämtliche Blätter dieselbe Tiefe haben (vgl. [OW17a, 261]). Bei *Güting und Dieker* entspricht die Definition der eines **fast vollständigen Baumes**, also ein Baum, der bis auf die letzte Ebene vollständig besetzt ist (vgl. [GD18c, 96], außerdem [CL22, 161] und [Knu97, 401]).

[1] Knoten eines Baumes gleicher Tiefe

Wir folgen der Definition von *Ottmann und Widmayer* (s. Skript (Teil 2), S. 101).

Der **worst-case** ergibt sich, wenn ein binärer Suchbaum **entartet**[2] ist - die Reihenfolge der Knoten entspricht dann der Anordnung in einer verketteten List, in der das kleinste Element am Anfang der Liste steht, das größte Element am Ende der Liste.
Soll jetzt ein Element eingefügt werden, dessen Schlüssel größer als alle in dem Suchbaum enthaltenen Werte ist, muss der *rechte Teilbaum* bis zum Ende durchwandert werden, um das Element einzufügen.
Bei n vorhandenen Knoten ergibt sich somit ein Zeitaufwand von $O(n)$ (Nachweis u.a. bei [GD18a, 135 f.]).

Für den **average-case** stellen *Sedgewick und Wayne* fest:

> The running time of algorithms on binary search trees depend on the shapes of the trees, which, in turn, depend on the order in which keys are inserted. In the best case, a tree with N nodes could be perfectly balanced [...] the balance in typical trees turns out to be much closer to the best case than the worst case. ([SW11, 403])

Ein **balancierter Baum** ist ein Baum, bei dem die Differenz der Höhe des linken Teilbaums und die Höhe des rechten Teilbaums eines Knotens max. 1 ist (vgl. [OW17a, 284]).
Die maximale Pfadlänge in einem balancierten Baum ist $O(log\ n)$[3] (vgl.[GD18a, 135]).
Hiermit ergibt sich im *ungünstigsten Fall*, dass mindestens $O(log\ n)$ Operationen nötig sind, um einen Schlüssel für eine Einfügeoperation zu finden.[4]
Eine vollständige Analyse, die für den **average-case** zu $O(log\ n)$ führt, findet sich bei *Güting und Dieker* ([GD18a, 136 ff.]).

5.2 Lösungsvorschlag b

- **Preorder**: 70, 50, 10, 25, 20, 45, 40, 35
- **Inorder**: 10, 50, 20, 25, 70, 40, 35, 45
- **Postorder**: 10, 20, 25, 50, 35, 40, 45, 70

5.2.1 Anmerkung und Ergänzungen

Die eindeutige Rekonstruktion des Bäumes kann mithilfe der **Preorder und Inorder**-Reihenfolge oder **Postorder und Inorder**-Reihenfolge realisiert werden.

[2] vgl. [GD18d, 136]
[3] n = Anzahl der Knoten
[4] In einem vollständigen Baum (*nicht* Suchbaum!) mit $2^{h-1} - 1$ Knoten (mit h = Höhe des Baumes) müssen im ungünstigsten Fall genausoviele (also der Anzahl der Knoten entsprechende) Vergleiche durchgeführt werden, um ein Blatt zu finden.

Rekonstruktion mit Preorder & Inorder

1. In der Preorder-Reihenfolge werden zuerst die Wurzelknoten besuche, dann der linke Teilbaum der Wurzel, dann der rechte (**KLR**).
   Knoten **(70)** kommt in der Preorder-Reihenfolge als erster Knoten vor, also muss das der Wurzelknoten des zu rekonstruierenden Baumes sein.
   (70) wird in der Inorder-Reihenfolge markiert; dadurch erkennt man den linken und den rechten Teilbaum: Der linke Teilbaum muss die Knoten **(10, 50, 20, 25)** enthalten, der rechte Teilbaum die Knoten **(40, 35, 45)** (s. Abbildung 5.2).
2. Im zweiten Schritt wird wieder zunächst die Reihenfolge der Knoten der Preorder-Reihenfolge untersucht: Der linke Teilbaum hat als einen Wurzelknoten die **(50)**, die wieder als Knoten in der Inorder-Reihenfolge markiert wird; gleiches passiert mit **(45)** des rechten Teilbaums der Preorder-Reihenfolge.
   Aus der Inorder-Reihenfolge lassen sich dann nach Markieren der Knoten weitere Teilbäume auslesen, die für den Knoten **(50)** den linken Teilbaum **(10)** und den rechten Teilbaum **(20, 25)** ergeben. Der Knoten **(45)** hat nur einen rechten Teilbaum mit dem Knoten **(40, 35)**.
3. Im letzten Schritt werden die verbliebenen Teilbäume untersucht.
   Der Teilbaum in Prerorder-Reihenfolge **(25, 20)** und in Inorder-Reihenfolge **(20, 25)** kann zu dem eindeutigen Teilbaum **(20)** (linker Nachfolgerknoten) und **(25)** (Vorgängerknoten) rekonstruiert werden.
   Der Teilbaum **(40, 35)** in Preorder- und Inorder-Reihenfolge lässt sich zu dem eindeutigen Teilbaum **(40)** (Vorgängerknoten) und **(35)** (rechter Nachfolgerknoten) rekonstruieren.

Die Rekonstruktion anhand Postorder und Inorder erfolgt analog.

Abb. 5.2: Rekonstruktion des Baumes anhand seiner Preorder- und Inorder-Reihenfolge (Quelle: eigene)

5.3 Lösungsvorschlag c

Es gibt mehrere Bäume (s. Abbildung 5.3).

5.3.1 Anmerkung und Ergänzungen

Die angegebenen Reihenfolgen t_{Pre}: A B C D E und t_{Post}: C B E D A lassen sich wie folgt untersuchen:

Der Baum t_{Pre} hat als Wurzelknoten den Knoten **A**.
Der Baum t_{Post} hat diesen Knoten ebenfalls als Knoten (s. Abbildung 5.4).

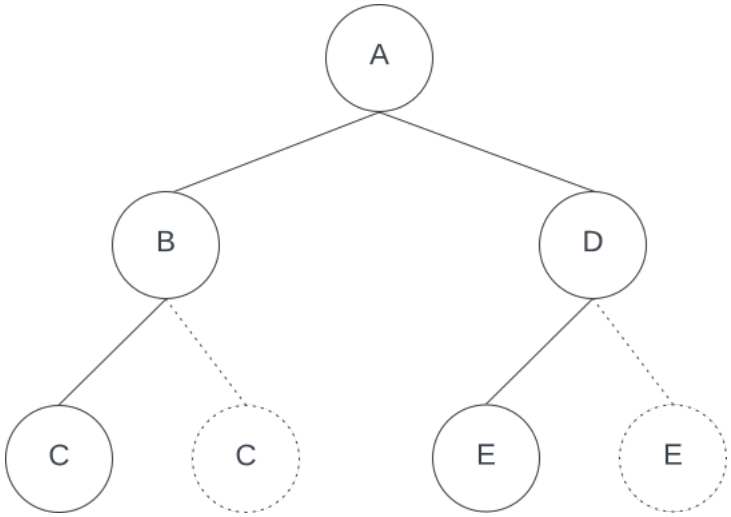

Abb. 5.3: Die angegebenen Lienarisierungen werden von mehreren Bäumen gleichzeitig erfüllt (Quelle: eigene)

Abb. 5.4

t_{Pre}: **A** B C D E
t_{Post}: C B E D **A**

Es verbleiben die Teilbäume mit den Knoten **(B C D E)** (Preorder) sowie **(C B E D)** (Postorder).

Für t_{Pre} kann der Knoten **(B)** als linker Nachfolger von **(A)** verwendet werden. Diese Bedingung kann auch von t_{Post} erfüllt werden (s. Abbildung 5.5).

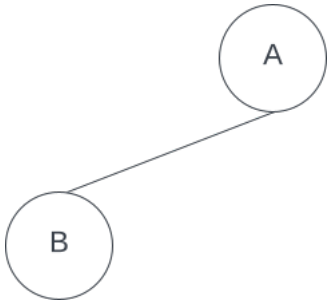

Abb. 5.5

Da **(C B E D)** in Postorder-Reihenfolge angegeben ist, muss folglich **(C)** ein Nachfolger von **(B)** sein, und zwar ein linker oder ein rechter Nachfolger.
Die Bedingung kann ebenfalls von t_{Pre} erfüllt werden (s. Abbildung **??**).

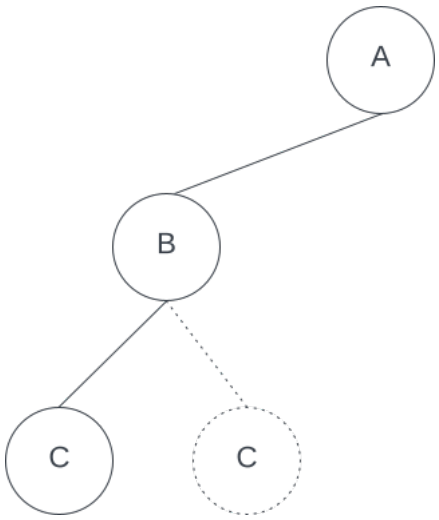

Abb. 5.6

t_{Pre}: **A B C** D E
t_{Post}: **C B** E D **A**

Mit **(B)** als direkter Nachfolger von dem Wurzelnoten **(A)** können jetzt aber **D** und **E** keine Vorgänger von **(B)** mehr sein für t_{Post}, sie sind somit Nachfolger von **A**.
Diese Bedingung läßt sich auch von t_{Pre} erfüllen, **E** kann entweder linker oder rechter Nachfolger von D sein (s. Abbildung 5.7).

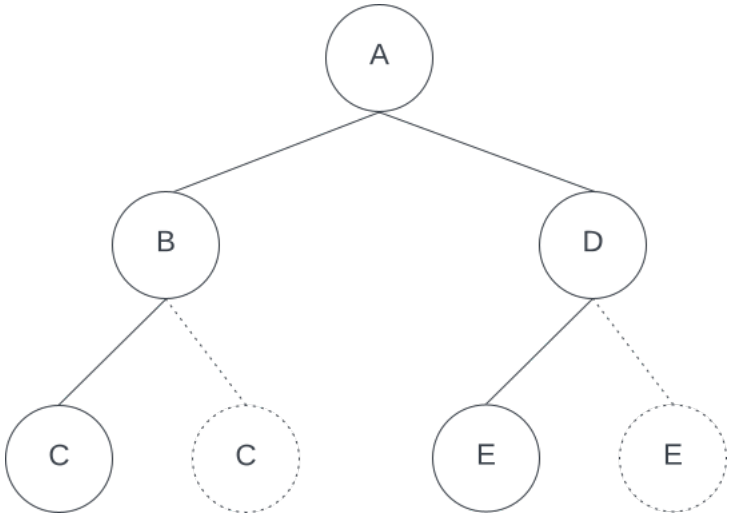

Abb. 5.7

6

Verkette Liste

6.1 Lösungsvorschlag

```
1  boolean istEnthalten(Object daten) {
2      ListElement next = kopf;
3
4      while (next != null) {
5          if (next.getDaten().equals(daten)) {
6              return true;
7          }
8          next = next.getNaechstes();
9      }
10
11     return false;
12 }
```

6.2 Anmerkung und Ergänzungen

Eine Liste ist eine **Sequenz** von Daten. Auf **Sequenzen** ist eine Ordnung definiert - es gibt ein erstes, zweites, ..., letztes Element und zu jedem Element einen Vorgänger und Nachfolger (vgl.[GD18c, 63]).
Sequenzen dürfen *Duplikate* enthalten.

In der Aufgabe ist die Implementierung einer einfach verketteten Liste (auch *lineare Liste*, vgl. [Knu97, 238], außerdem [GD18c, 73]) gegeben:

> A *linked list* is a recursive data structure that is eitehr empty (*null*) or a reference to a *node* having a generic item and a reference to a linked list. ([SW11, 142])

Eine *lineare Liste* hat ein Element, auf das kein anderes Element zeigt (*head*, Angfang der Liste), und mindestens ein Element, welches auf kein weiteres Element

zeigt (*tail*, Ende der Liste)- ansonsten zeigt ein Element auf seinen Nachfolger (vgl. [CL22, 259])[1].
Es gibt unterschiedliche Implementierungsmöglichkeiten für lineare Listen, die entsprechende Zugriffsreihenfolgen auf die in der Liste gespeicherten Daten ermöglichen, u.a. **Stack** und **Queue**.

Stack

Ein **Stack**[2] arbeitet nach dem **LIFO**-Prinzip: Das Element, was als letztes (*auf* dem Stack) hinzugefügt wurde, wird als erstes wieder entnommen.
Eingefügt wird am Ende des Stacks.

Queue

Im Gegensatz zum Stack arbeitet eine **Queue**[3] nach dem **FIFO**-Prinzip: Das Element, was als erstes hinzugefügt wurde, wird als erstes entnommen.
Wie beim Stack wird auch bei der Queue am Ende der Liste eingefügt.

Die Kosten für Einfügen und Entfernen ("Entnahme" sind für beide Listenarten konstant mit $O(1)$.
Suchoperationen benötigen im `best-case` $O(1)$, im `worst-case` $O(n)$. Im Durchschnitt muss man nur die Hälfte der Liste nach einem Element durchsuchen, bis es gefunden wird, was im **average-case** wieder zu einer linearen Laufzeit von $O(n)$ führt ($\frac{n}{2} = \frac{1}{2} * n \implies O(n)$).

Warum equals()?

Die Klasse `java.lang.Object` als Elternklasse jeder anderen Klasse in Java[4] implementiert die Methode `equals(obj: Object):boolean` wie folgt:

> The equals method for class Object implements the most discriminating possible equivalence relation on objects; that is, for any non-null reference values x and y, this method returns true if and only if x and y refer to the same object (x == y has the value true). In other words, under the reference equality equivalence relation, each equivalence class only has a single element. ("equals": `https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/Object.html#equals(java.lang.Object)` - abgerufen 13.2.2024).

[1] Ebenda: ist eine Liste *doppelt verkettet* (Element besitzt Zeiger auf Vorgänger*prev*) und zeigt *head* auf *tail* und *tail* auf *head*, handelt es sich um eine *zyklische Liste* (*Ring*) (s.a. [GD18c, 105])
[2] auch *Keller* oder *Stapel*
[3] auch *(Warte)schlange*
[4] "4.3.2. The Class Object": `https://docs.oracle.com/javase/specs/jls/se8/html/jls-4.html#jls-4.3.2` - abgerufen 13.2.2024

Die Methode `equals()` sollte in Unterklassen überschrieben, *nicht* überladen werden (vgl. [Ull23, 654], weitere Hinweise auf die korrekte Implementierung der Methode ebenda, ausserdem bei *Bloch* in [Blo18, 37]).

Da keine Angaben über den speziellen Typ von dem in der Liste gespeicherten Objekt gegeben ist, würde der Vergleich auf Identität spätestens bei Typen wie `String` dazu führen, dass Elemente nicht gefunden werden:

```
1  boolean istEnthalten(Object daten) {
2      ListElement next = kopf;
3
4      while (next != null) {
5          // ==, Vergleich auf Identität
6          if (next.getDaten() == daten) {
7              return true;
8          }
9          next = next.getNaechstes();
10     }
11
12     return false;
13 }
14
15 // liefert immer false zurück, auch wenn vorher über
16 // liste.add(new String("foo")) ein entsprechendes
17 // Eement mit gleichem Inhalt hinzugefügt wurde
18 boolean enthalten = liste.istEnthalten("foo");
```

In einer korrekten Implementierung von `istEnthalten` wird nun die `equals()`-Implementierung der Klasse `String` aufgerufen, die einen *inhaltlichen* Vergleich vornimmt, und mit

```
new String("foo").equals("foo")
```

auch `true` zurückliefert, im Gegensatz zu `new String("foo") == stringVariable`, wobei `stringVariable` vom Typ `String` ist und inhaltlich mit `"foo"` übereinstimmt.

String-Literale

Eine Verwendung von `==` *kann* bei dem Vergleich von **String-Literalen** `true` zurückliefern, was daran liegt, das String-Literale in einer Art Cache vorgehalten werden, damit die JVM nicht laufend neue String-Objekte erzeugen muss:

„ All literal strings and string-valued constant expressions are interned. “ (“intern“: `https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/String.html#intern()` - abgerufen 13.2.2024)

Die Sprachspezifikation beinhaltet diesbzgl. weitere Informationen (s. “3.10.5. String Literals“: `https://docs.oracle.com/javase/specs/jls/se21/html/jls-3.html#jls-3.10.5` - abgerufen 13.2.2024).

7

Sortierverfahren

7.1 Lösungsvorschlag

7.1.1 Aufgabenteil a)

- Es gilt, dass jedes allgemeine Sortierverfahren mindestens $\Omega(n\ log\ n)$ Schlüsselvergleiche benötigt (vgl. [OW17c, 154]).
  Diese untere Schranke gilt sowohl für die *elementaren Sortierverfahren*[1] (**Insertion-**, **Selection-** und **Bubble**-Sort), die im **worst-case** $O(n^2)$ Zeit benötigen, als auch für die Verfahren, die eine **divide and conquer**-Strategie implementieren, wie **Quicksort** ($O(n^2)$) und **Merge-Sort** ($O(n\ log\ n)$).
- Bei sehr wenigen Datensätzen (in der Aufgabe mit $n \leq 100$ angegeben) ist die Wahl des Sortierverfahren hinsichtlich Effizienz im Sinne von Speicherplatzverbrauch als auch der benötigten Laufzeit eher nebensächlich, wenn man davon ausgeht, dass das Sortieren auf einem der heutigen Technik entsprechenden Rechner mit einem der o.a. Verfahren durchgeführt wird.
  Qualitätskriterien wie Einfachheit der Implementierung und Verständlichkeit können hier eher ins Gewicht gefallen (vgl. [GD18b, 5 f.]).
- Andere Kriterien, die die Eingabedaten betreffen, können jedoch die Auswahl des Sortierverfahrens beeinflussen: Sind die Daten überwiegend vorsortiert, kann bspw. **Insertion-Sort** verwendet werden, das bei vorsortierten Daten lineare Laufzeit erreichen kann (vgl. [CL22, 188])[2].
- Bei kleinen Problemgrößen fällt genau die damit verbundene Anzahl der Eingabedaten n stärker ins Gewicht. Benötigt bspw. eine Implementierung von Insertion-Sort $8 * n^2$ Schritte und eine Implementierung von Merge-Sort $64 * n\ log\ n$ Schritte, so ist Insertion Sort für $n \leq 43$ effizienter als Merge-Sort (s. Abbildung 7.1)[3] [4].

[1] s. [OW17c, 81]

[2] *Ottmann und Widmayer* führen außerdem **Smoothsort** von Dijkstra an, dass $O(n)$ für eine vorsortierte Folge und $O(n\ log\ n)$ benötigt (vgl. [OW17c, 112])

[3] *Cormen et. al.* empfehlen einen *Hybrid* aus Merge-Sort und Insertion-Sort in [CL22, 45, Problem 2-1], worauf auch *Sedgewick und Wane* in [SW11, 275] hinweisen.

[4] bei einem direkten Vergleich von Merge-Sort und Bubble-Sort dürfte schnell klar werden, dass ein unsortiertes Feld mit $n = 2$ von Bubble-Sort mit weniger Operationen sortiert wird, als das bei Merge-Sort der Fall ist, wo neben dem Sortieren und Verschmelzen der Eingabefolgen außerdem zusätzlich linear viel Speicher für die Teilfolgen benötigt wird.

- In der Literatur finden sich Empfehlungen für Quick-Sort, das für typische Eingabedaten im Durchschnitt $O(n\ log\ n)$ Zeit benötigt und in-place arbeitet (vgl. [CL22, 182 ff.]).
- Für eine hohe Zahl von Datensätzen, die auch im **worst-case** optimale Laufzeit aufweist, kann Merge-Sort verwendet werden, das mit $O(n\ log\ n)$ *worst-case-optimal* ist (vgl.[OW17c, 116]).

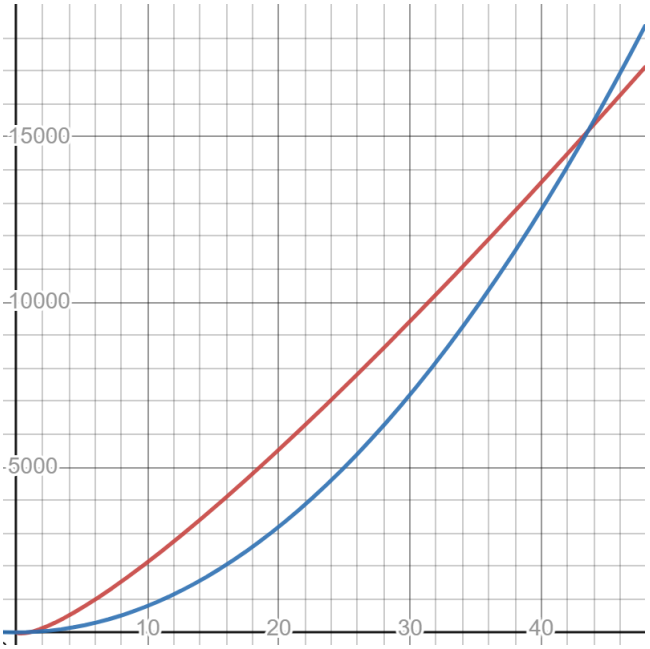

Abb. 7.1: Für eine Eingabelänge von $n \leq 43$ arbeitet eine Implementierung von Insertion-Sort, die im Beispiel $8 * n^2$ Schritte (blau) benötigt, effizienter als eine Implementierung von Merge-Sort, die im Beispiel $64 * n * log\ n$ Schritte (rot) benötigt. (Quelle: eigene)

7.1.2 Aufgabenteil b)

Bei dem vorgestellten Sortierverfahren handelt es sich um **Selection-Sort**. Bei dem Sortierverfahren wird ein Feld A mit einer Problemgröße n und einem geg. Index i mit $0 \leq i \leq n-2$ Teilfolgen $[A_{i+1}, ..., A_n]$ nach einem kleinsten Schlüssel $A_{min} < A_i$ durchsucht, wobei A_{min} in der jeweiligen Teilfolge auch der kleinste Schlüssel ist.
Der gefundene Schlüssel wird dann mit A_i getauscht.
Die Teilfolgen enthalten so jeweils einen kleinsten, dann den zweitkleinsten, dann den drittkleinsten, ... , und schließlich den größten Schlüssel, die Vertauschung ergibt dann zum Schluss ein aufsteigend sortiertes Feld (s. [OW17c, 82]).

7.1.3 Aufgabenteil c)

Die Laufzeit eines Algorithmus ist die Summe der Laufzeiten jeder einzelnen ausgeführten Anweisung.

Eine Anweisung, die aus insg. c_k (Elementar-)Operationen besteht, und die m-mal aufgerufen wird, trägt zu der Laufzeit mit $c_k * m$ bei (vgl. [CL22, 29 f.]).

Im Folgenden werden die Kosten für die Aufrufe von Zeile 5 berechnet, aus denen der Wert von `vergleich` abgeleitet werden kann:

```
1  while (i < arr.length - 1) {                          // c1   n
2      min = arr[i];                                     // c2   n − 1
3      minIndex = i;                                     // c3   n − 1
4      for (int j = i + 1; j < arr.length; j++) {        // c4   ∑_{i=0}^{n−2} t_i
5          vergleiche++;                                 // c5   ∑_{i=0}^{n−2} (t_i − 1)
6          if (arr[j] < min) {                           // c5   ∑_{i=0}^{n−2} (t_i − 1)
7              minIndex = j;                             // c7   ∑_{i=0}^{n−2} (t_i − 1)
8              min = arr[j];                             // c8   ∑_{i=0}^{n−2} (t_i − 1)
9          }
10     }
11     arr[minIndex] = arr[i];                           // c9   n − 1
12     arr[i] = min;                                     // c10  n − 1
13     i++;                                              // c11  n − 1
14 }
```

Für $i = 0, 1, 2, ..n-2$ ist t_i die Anzahl der Aufrufe der Schleifenbedingung in Zeile 5.
Wie schon an der Gesamtlaufzeit für Zeile 1 und Zeile 2 ersichtlich, wird eine Schleifenbedingung immer ein mal mehr aufgerufen, als das davon abhängige Statement (der *Block* in unserem Fall[5]).

Für Zeile 1 des o.a. Listings ergeben sich mit $i = 0$ und $n = arr.length$ somit $n-1$ Aufrufe[6].
Das nachfolgende Statement der while-Schleife[7] (in Form eines *Blocks* in Zeile 2-

[5] s. "14.12. The while Statement": `https://docs.oracle.com/javase/specs/jls/se21/html/jls-14.html#jls-14.12` - abgerufen 19.2.2024
[6] Ergibt der Ausdruck in der Schleifenbedingung `-NoValue-`, ist das Feld entweder leer ($n = 0$) oder es gilt $i < n - 1$. Wir gehen im folgenden von $n > 0$ aus.
[7] s. "14.12. The while Statement": `https://docs.oracle.com/javase/specs/jls/se21/html/jls-14.html#jls-14.12` - abgerufen 19.2.2024

13) wird $n-2$-mal aufgerufen.
Der Block der `for`-Schleife (Zeile 5-9) wird in Abhängigkeit von i aufgerufen: Für $i=0$ wird er $n-1$ mal aufgerufen, für $i=1$ $n-2$-mal usw.

Für die Zählvariable j gilt

$$j := n \in \mathbb{N}, j \geq i+1, j \leq n-1 \tag{7.1}$$

Für die Anzahl der Aufrufe der Schleifenbedingung t_i gilt[8]:

$$t_i = \sum_{j=i+1}^{n} 1 \tag{7.2}$$

Die Kosten $T(n_4)$ für Zeile 4 lassen sich somit wie folgt berechnen:

$$T(n_4) = c_4 * \sum_{i=0}^{n-2} t_i = c_4 * \sum_{i=0}^{n-2} \sum_{j=i+1}^{n} 1 \tag{7.3}$$

Die Summe lässt sich weiter auflösen zu

$$\begin{aligned}
\sum_{i=0}^{n-2} \sum_{j=i+1}^{n} 1 &= \sum_{i=0}^{n-2} \sum_{j=1}^{n-i} 1 \\
&= \sum_{i=0}^{n-2} (n-i) \\
&= \sum_{i=1}^{n-1} (n-(i-1)) \\
&= \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} (i+1) \\
&= n*(n-1) - \frac{n*(n-1)}{2} + n - 1 \\
&= \frac{n*(n+1)}{2} - 1
\end{aligned} \tag{7.4}$$

wodurch sich letztendlich die Kosten berechnen lassen mit

$$T(n_4) = c_4 * (\frac{n*(n+1)}{2} - 1) \tag{7.5}$$

Die Anzahl der Aufrufe von c_4 ist dann $m_{c_4} = \frac{n*(n+1)}{2} - 1$

[8] Endwert der Summe ist n, da die Schleifenbedingung ein mal mehr als der nachfolgende Block aufgerufen wird

Jeder Aufruf von c_5 findet aufgrund des Abbruchs der Schleife[9] $t_i - 1$-mal statt. Für die Gesamtzahl der Aufrufe m_{c_5} von $c5$ muss deshalb $n-1$ von m_{c_4} subtrahiert werden:

$$m_{c5} = (\frac{n*(n+1)}{2} - 1) - (n-1)) = \frac{n*(n-1)}{2} \tag{7.6}$$

was der Anzahl der Aufrufe von Zeile 5 entspricht, und somit zu dem Wert von `vergleiche` führt[10].

7.1.4 Aufgabenteil d)

Die Laufvariable `i` wird zur Laufzeit - bis auf das in jedem Iterationsschritt erfolgende Inkrement - nicht geändert, die innere Schleife ist stets abhängig von `i`.
Des Weiteren bleiben die Schleifenbedingungen konstant.

Wie in Aufgabenteil c) gilt auch hier für die Anzahl der Vergleiche

$$vergleiche = \frac{n*(n-1)}{2} \tag{7.7}$$

woraus wiederum die Komplexitätstklasse $O(n^2)$ folgt.

7.1.5 Aufgabenteil e)

Unter einem **stabilen** Sortierverfahren versteht man Verfahren, bei denen die **relative Reihenfolge** gleicher Schlüsselelemente durch das Sortieren nicht verändert wird (vgl. [CL22, 210]).
Zu den stabilen Sortierverfahren gehören

- Bubble-Sort
- Insertion-Sort
- Merge-Sort

[9] Überprüfung der Schleifenbedingung $j < arr.length$
[10] entsprechend führt das Skript (Teil 2) auf Seite 160 die Anzahl an Vergleichen auf.

8

Hashing

8.1 Lösungsvorschlag

Die Schlüssel 4, 10, 3, 19 sollen nacheinander in eine Hashtabelle über die Hashfunktion

$$h_0(k) = k \ mod \ 7 \tag{8.1}$$

eingefügt werden.

Zur Kollisionsbehandlung[1] wird **quadratisches Sondieren mit wechselndem Vorzeichen** verwendet, wodurch die Sondierungsfolge[2] für den Schlüssel k nach folgendem Muster berechnet wird:

$\mathrm{h}_1(k) = (h_0(k) + 1^2) \ mod \ 7$

$\mathrm{h}_2(k) = (h_0(k) - 2^2) \ mod \ 7$

$\mathrm{h}_3(k) = (h_0(k) + 3^2) \ mod \ 7$

$\mathrm{h}_4(k) = (h_0(k) - 4^2) \ mod \ 7$

$\mathrm{h}_5(k) = (h_0(k) + 5^2) \ mod \ 7$

...

Die resultierende Speicherzellenbelegung ist in Tabelle 8.1 angegeben.

[1] auch: *Kollisionsauflösung*, vgl.[OW17b, Abschnitt 4.2 und 4.3]

[2] die Folge der zu betrachtenden Speicherplätze für einen Schlüssel k (vgl. [OW17b, 203])

Schlüssel	$h_i(k)$	**Ergebnis**	**Kollision**	**Speicherzelle**
4	$h_0(4) = 4 \; mod \; 7$	4	–	4
10	$h_0(10) = 10 \; mod \; 7$	3	–	3
3	$h_0(3) = 3 \; mod \; 7$	3	3	
	$h_1(3) = (3 \; + 1^2) \; mod \; 7$	4	4	
	$h_2(3) = (3 \; - 2^2) \; mod \; 7$	6	–	6
19	$h_0(19) = 19 \; mod \; 7$	5	–	5

Tabelle 8.1: Speicherzellenbelegung für die Schlüsselfolge 4, 10, 3, 19 für die Hashfunktion 8.1 und der angegebenen Kollisionsbehandlung. Es treten insgesamt 2 Kollisionen auf.

Entfernen

Bei dem Löschen eines Schlüssels k_d muss sichergestellt werden, dass die Hashadresse $h(k_d)$ als *entfernt* markiert wird - ansonsten würden Schlüssel, die aufgrund einer vorherigen Kollision über eine Sondierungsfolge eine neue Hashadresse zugewiesen bekommen haben, nicht wiedergefunden werden[3].

Bei einer *delete*-Operation müssen für den zu löschenden Schlüssel die Speicherzellen anhand der Sondierungsfunktion berechnet werden. Sobald eine nicht-belegte Speicherzelle gefunden wird, war der zu löschende Schlüssel nicht in der Hashtabelle enthalten.

Für `delete(4)` werden unter Anwendung der Sondierungsfunktion folgende Speicherzellen besucht, in dieser Reihenfolge:

1. *Speicherzelle*: 4.
   - Schlüssel 4 in Speicherzelle 4 enthalten → als gelöscht markieren → $D4$

Für `delete(3)` werden unter Anwendung der Sondierungsfunktion folgende Speicherzellen besucht, in dieser Reihenfolge:

1. *Speicherzelle*: 3.
   - Schlüssel 3 in Speicherzelle 3 enthalten → nein → weitersuchen
2. *Speicherzelle*: 4.
   - Schlüssel 3 in Speicherzelle 4 enthalten → nein → weitersuchen
3. *Speicherzelle*: 6.
   - Schlüssel 3 in Speicherzelle 6 enthalten → ja, als gelöscht markieren → $D6$

[3] vgl. Skript (Teil 2) S. 134, außerdem [OW17b, 203]

Suchen

`member` überprüft, ob ein Schlüssel in einer Hashtabelle enthalten ist.
Die Sondierungsfunktion berechnet hierbei nach der o.g. Vorgehensweise die zu durchsuchenden Speicherzellen $3, 4, 6, 5, 1$.
Die Speicherzellen 4 und 6 waren als gelöscht markiert, die Speicherzellen 3 und 5 enthalten bereits Schlüssel, aber nicht den Schlüssel 17.
Die Speicherzelle 1 enthält keinen Schlüssel, aus diesem Grund ist die Suche nach dem Schlüssel 17 erfolglos.

9

Rekursion

9.1 Lösungsvorschlag

Der Methode `recursion` wird ein ganzzahliger Wert übergeben.
Die Methode ruft sich dann 2-mal mit dem Übergabeparameter $n-1$ selber auf.
Der Rückgabewert der Methodenaufrufe wird addiert, die Summe zurückgegeben:

```
1  return recursion(n - 1) + recursion(n - 1);
```

Die *Abbruchbedingung* ist $n \leq 0$ - in dem Fall liefert die Methode ohne weiteren rekursiven Aufruf den Wert 1 zurück.

Für die Rekursionsgleichung R ergibt sich demnach in Abhängigkeit des Eingabewertes n mit $n \in \mathbb{N}$

$$R(n) = \begin{cases} 1 \; falls \; n \; = 0 \\ R(n-1) + R(n-1) \end{cases} \tag{9.1}$$

Wie in Abbildung 9.1 ersichtlich ist, verdoppeln sich die Anrufe für `recursion` mit jedem Rekursionsschritt.

Der initiale Aufruf `recursion(n)` ruft 2-mal `recursion(n-1)` auf.
Danach wird für jeden dieser Aufrufe `recursion(n-2)` aufgerufen, also insg. 4-mal. . . bis die Abbruchbedingung $n \leq 0$ erüllt ist.
Für die Anzahl m der Aufrufe von `recursion` ergibt sich somit

$$m = 1 + 2 + 4 + \cdots + 2^n = \sum_{i=0}^{n} 2^i \tag{9.2}$$

Für den Rekursionsbaum ist der Rückgabewert bei Erfüllung der Abbruchbedingung von Interesse, der den Hinweis zu dem Ergebnis in Abhängigkeit von n liefert.
Die Anzahl der Blätter des Baumes entsprechen dem letzten Summanden der Gleichung 9.2, mit $i = n$ also 2^n.

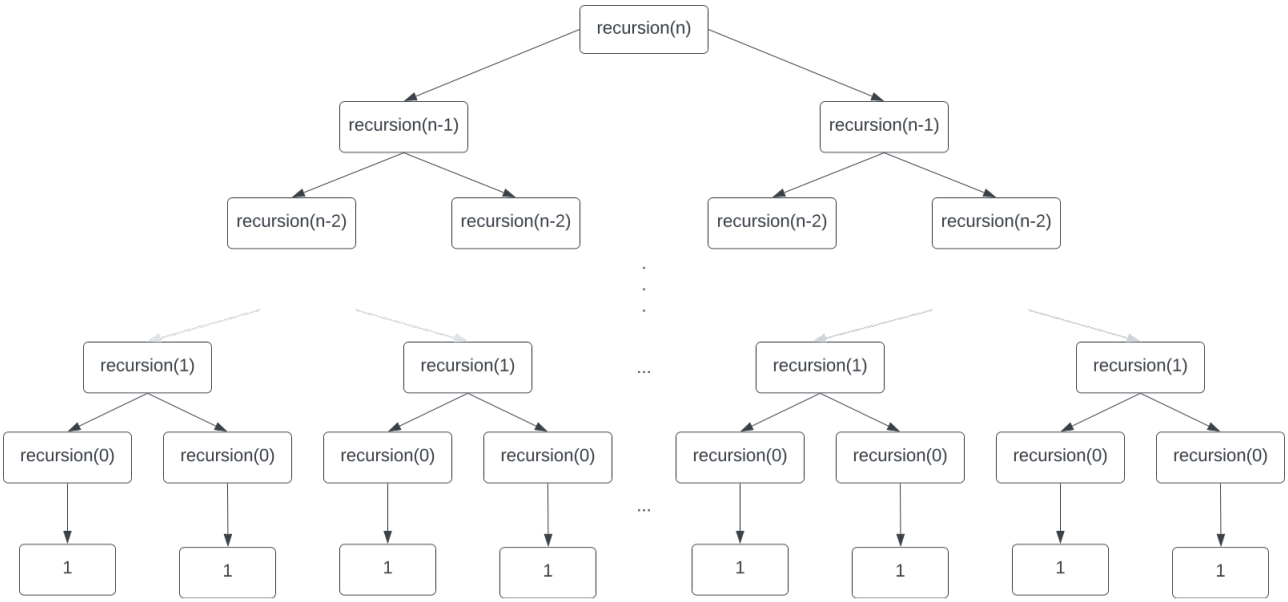

Abb. 9.1: Der Rekursionsbaum der Methode *recursion* (Quelle: eigene)

Da letztendlich deren Summe das Ergebnis der Methode `recursion` liefert, ist das gesuchte Ergebnis 2^n.

Die Rekursionstiefe des Baumes entspricht n, wobei sich auf jeder Ebene die Anzahl der rekursiven Aufrufe verdoppelt[1].
Mit der Anzahl der Rekursionsaufrufe lässt sich auch auf die Komplexitätsklasse schliessen, die mit $O(2^n)$ **exponentiell** ist.

[1] Anzahl gleichzeitig aktiver *rekursiver* Aufrufe; der erste Aufruf wird i.d.R. nicht dazugezählt und entspricht der "Wurzel" des Rekursionsbaums. Die Rekursionstiefe entspricht damit der *Höhe* des Baumes (vgl. Skript (Teil 2) S.34); bzgl. der Begriffsbestimmung siehe hierzu auch **[CK75]**.

10

O-Notation

10.1 Lösungsvorschlag

10.1.1 Teil 1

Die Schleifen können wie folgt als verschachtelte Summe formuliert werden:

$$\begin{aligned}\sum_{i=1}^{\lfloor \frac{n^2}{5} \rfloor} \sum_{j=1}^{n} 1 &= \sum_{i=1}^{\lfloor \frac{n^2}{5} \rfloor} n \\ &= n * \lfloor \frac{n^2}{5} \rfloor \\ &\approx \frac{1}{5} * n * n^2\end{aligned} \tag{10.1}$$

was zu $O(n^3)$ führt (*kubische Komplexität*).

10.1.2 Teil 2

Die Schleifen können wie folgt als verschachtelte Summe formuliert werden:

$$\begin{aligned}\sum_{i=1}^{\lceil \frac{n}{2} \rceil} \sum_{j=1}^{\lfloor log_2(n)+1 \rfloor} 1 &= \sum_{i=1}^{\lceil \frac{n}{2} \rceil} (\lfloor log_2(n) + 1 \rfloor) \\ &= \lceil \frac{n}{2} \rceil * \lfloor log_2(n) + 1 \rfloor \\ &\approx \frac{1}{2} * (n * log_2(n) + 1)\end{aligned} \tag{10.2}$$

was zu $O(n * log(n))$ führt (*linearithmische* Komplexität).

10.1.3 Teil 2

Die Schleifen können wie folgt als verschachtelte Summe formuliert werden:

$$\begin{aligned}\sum_{i=1}^{n}\sum_{j=1}^{10}\sum_{k=1}^{j} 1 &= \sum_{i=1}^{n}\sum_{j=1}^{10} j \\ &= \sum_{i=1}^{n} \frac{10 * (10+1)}{2} \\ &= n * 55\end{aligned} \tag{10.3}$$

was zu $O(n)$ führt (*lineare* Komplexität).

Literaturverzeichnis

[Blo18] Joshua Bloch. Effective Java. 3. Aufl. Addison-Wesley, 2018. ISBN: 978-0-13-468599-1.

[CL22] Thomas H Cormen und Charles E Leiserson. Introduction to Algorithms, fourth edition. en. London, England: MIT Press, Apr. 2022. ISBN: 9780262046305.

[GD18a] Ralf Hartmut Güting und Stefan Dieker. „Datentypen zur Darstellung von Mengen“. In: Datenstrukturen und Algorithmen. Wiesbaden: Springer Fachmedien Wiesbaden, 2018, S. 109–167. ISBN: 978-3-658-04676-7. DOI: 10.1007/978-3-658-04676-7_4. URL: https://doi.org/10.1007/978-3-658-04676-7_4.

[GD18b] Ralf Hartmut Güting und Stefan Dieker. „Einführung“. In: Datenstrukturen und Algorithmen. Wiesbaden: Springer Fachmedien Wiesbaden, 2018, S. 1–38. ISBN: 978-3-658-04676-7. DOI: 10.1007/978-3-658-04676-7_1. URL: https://doi.org/10.1007/978-3-658-04676-7_1.

[GD18c] Ralf Hartmut Güting und Stefan Dieker. „Grundlegende Datentypen“. In: Datenstrukturen und Algorithmen. Wiesbaden: Springer Fachmedien Wiesbaden, 2018, S. 63–107. ISBN: 978-3-658-04676-7. DOI: 10.1007/978-3-658-04676-7_3. URL: https://doi.org/10.1007/978-3-658-04676-7_3.

[GD18d] Ralf Hartmut Güting und Stefan Dieker. „Sortieralgorithmen“. In: Datenstrukturen und Algorithmen. Wiesbaden: Springer Fachmedien Wiesbaden, 2018, S. 169–200. ISBN: 978-3-658-04676-7. DOI: 10.1007/978-3-658-04676-7_5. URL: https://doi.org/10.1007/978-3-658-04676-7_5.

[Knu97] Donald E. Knuth. The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamenta 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. „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.

[OW17c] Thomas Ottmann und Peter Widmayer. „Sortieren“. In: Algorithmen und Datenstrukturen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, S. 79–165. ISBN: 978-3-662-55650-4. DOI: 10.1007/978-3-662-55650-4_2. URL: https://doi.org/10.1007/978-3-662-55650-4_2.

[SW11] Robert Sedgewick und Kevin Wayne. Algorithms, 4th Edition. Addison-Wesley, 2011, S. I–XII, 1–955. ISBN: 978-0-321-57351-3.

[Ull23] Christian Ullenboom. Java ist auch eine Insel, 17. Auflage. Galileo Computing, 2023. ISBN: 978-3-8362-9544-4.