Aufgabensammlung ADS-Repetitorium WS 24/25

rekursive Algorithmen – elementare Datenstrukturen – Bäume

Aufgabe 1: Rekursive Gleichung aufstellen
Gegeben sei folgender Algorithmus:

Algorithmus 1: RecursiveAlgo(int A[ ], int l= 1, int r=A.length)

```
1 if l < r then
2     m = ⌊(l + r)/2⌋
3     RecursiveAlgo(A, l, m)
4     RecursiveAlgo(A, m + 1, r)
5     InsertionSort(A, l, r)
```

(a) Stellen Sie eine Rekursionsgleichung T für den gegebenen Algorithmus auf.

Lösung:

$$T(n) = \begin{cases} 2 \cdot T(n/2) + n^2 & \text{falls } n > 1 \\ 1 & \text{sonst} \end{cases}$$

(b) Finden Sie eine Funktion f, für die $T \in \Theta(f)$ gilt.

Lösung: Lösung mit der Meistermethode: $a = 2$, $b = 2$ und $f(n) = n^2$. Es ist $\log_b a = \log_2 2 = 1$ und $f(n) \in \Omega(n^{\log_b a + \epsilon})$ für $\epsilon = 0,1$. Damit ist $T(n) \in \Theta(n^2)$, falls die Regularitätsbedingung erfüllt ist. Dies ist einfach überprüfbar, wähle dann c entsprechend $0,5 \leq c < 1$.

Aufgabe 2: 3-MergeSort
Gegeben sei folgende Algorithmus 2, das eine Variante vom bekannten MergeSort ist.

Algorithmus 2: `3MergeSort`(**int**[] A, **int** $l = 1$, **int** $r = A.length$)

```
1 if l < r then
2     m1 = ⌊(r+l)/3⌋
3     m2 = ⌊2(r+l)/3⌋
4     3MergeSort(A, l, m1)
5     3MergeSort(A, m1 + 1, m2)
6     3MergeSort(A, m2 + 1, r)
7     Merge(A, l, m1, m2)
8     Merge(A, l, m2, r)
```

(a) Formulieren Sie die Laufzeit $T(n)$ von `3MergeSort` als Rekursionsgleichung in Abhängigkeit von der Länge des Feldes A.

Lösung:

$$T(n) = \begin{cases} \Theta(1) & \text{falls } n = 1 \\ 3T\left(\frac{n}{3}\right) + \Theta(n) & \end{cases}$$

(b) Bestimmen Sie die asymptotische Laufzeit von `3MergeSort` mithilfe der Meistermethode. Geben Sie dabei den Fall an.

Lösung: Mit $a = 3$, $b = 3$ und $f(n) = \Theta(n)$ passt Fall 2 der Meistermethode, da $f \in \Theta(n^{\log_b(a)}) = \Theta(n)$ ist. Somit ist die asymptotische Laufzeit von `3MergeSort` $\Theta(n \log n)$.

Aufgabe 3: Rekursive Laufzeiten

Finden Sie für die nachstehenden Rekursionsgleichungen jeweils eine Funktion f, für die $T \in \Theta(f)$ gilt. Sie können davon ausgehen, dass die Laufzeit im Basisfall konstant ist.

(a) $T(n) = 4T(\lfloor n/2 \rfloor) + \frac{1}{2}n^2\sqrt{n}$

Lösung: Wir setzen $a = 4$, $b = 2$ und $f(n) = 0,5n^2\sqrt{n}$. Dann ist $\log_b a = \log_2 4 = 2$.
Damit gilt $f(n) \in \Omega(n^{2+\varepsilon})$ für $\varepsilon = 0,1$. Das ist der dritte Fall des Meistertheorems.
Damit ist $T(n) \in \Theta(0,5n^2\sqrt{n})$.
Die Regularitätsbedingung ist erfüllt:

$$\begin{aligned} 4 \cdot 0,5(n/2)^2\sqrt{n/2} = \cdot 0,5n^2\sqrt{n/2} &\leq c \cdot f(n) \\ \sqrt{n/2} &\leq c \cdot \sqrt{n} \\ \sqrt{n}/\sqrt{2} &\leq c \cdot \sqrt{n} \end{aligned}$$

Wähle $1/\sqrt{2} < c < 1$.

(b) $T(n) = 4T(\lfloor n/2 \rfloor) + n^2 \log n + n$

Lösung: Sei $a = 4$, $b = 2$ und $f(n) = n^2 \log n + n$. Dann ist $\log_b a = \log_2 4 = 2$ und $f(n) \notin \Omega(n^{2+\varepsilon})$.
Deshalb wenden wir die Rekursionsbaum-Methode an:

Ebene	Knoten	Beitrag
0	1	$n^2 \log n + n$
1	4	$4 \cdot (n/2)^2 \log(n/2) + 4 \cdot n/2 = n^2 \log(n/2) + 2n$
2	16	$16 \cdot (n/4)^2 \log(n/4) + 4^2 \cdot n/4 = n^2 \log(n/4) + 4n$
3	64	$64 \cdot (n/8)^2 \log(n/8) + 4^3 \cdot n/8 = n^2 \log(n/8) + 8n$
i	4^i	$4^i \cdot (n/2^i)^2 \log(n/2^i) + 4^i \cdot n/2^i = n^2 \log(n/2^i) + 2^i \cdot n$
$\log_2 n$	$4^{\log_2 n} = n^2$	1

So ergibt sich die folgende Summe:

$$\begin{aligned} T(n) &= n^2 + \sum_{i=0}^{\log_2 n - 1} n^2 \log(n/2^i) + 2^i n = n^2 + n^2 \sum_{i=0}^{\log_2 n - 1} \log(n/2^i) + \frac{2^i}{n} \\ &= n + n^2 \left(\sum_{i=0}^{\log_2 n - 1} \log(n) - \sum_{i=0}^{\log_2 n - 1} \log(2^i) + \sum_{i=0}^{\log_2 n - 1} \frac{2^i}{n} \right) \\ &= n^2 + n^2 \left(\log_2 n \cdot \log(n) - \frac{(\log_2 n - 1)(\log_2 n)}{2} + \frac{1}{n} \sum_{i=0}^{\log_2 n - 1} 2^i \right) \\ &< n^2 + n^2 \left(\log_2 n \cdot \log(n) - \frac{(\log_2 n)(\log_2 n)}{2} + \frac{1}{n} \sum_{i=0}^{\log_2 n - 1} 2^i \right) \\ &= \Theta(n^2) + \Theta(n^2 \log^2 n) + \underbrace{\Theta(1)}_{\sum_{i=0}^{\log_2 n - 1} 2^i = n-1} \in \Theta(n^2 \log^2 n) \end{aligned}$$

(c) $T(n) = T(n-3) + 2n$

Lösung: Meistermethode nicht anwendbar, aber einfache Lösung möglich:

$$T(n) = 2n + 2(n-3) + 2(n-6) + \cdots = 2\sum_{i=0}^{n/3} n - 3i = \frac{n}{3} \cdot 2n - 6 \cdot \frac{(n/3)(n/3+1)}{2} \in \Theta(n^2)$$

(d) $T(n) = 2T(\lfloor n/4 \rfloor) + 3\sqrt{n}$

Lösung: Wir setzen $a = 2$, $b = 4$ und $f(n) = 3\sqrt{n}$. Dann ist $\log_b a = \log_4 2 = 0.5$.
Damit gilt $f(n) \in \Theta(n^{0.5})$. Das ist der zweite Fall des Meistertheorems.
Damit ist $T(n) \in \Theta(\sqrt{n} \log n)$.

(e) $T(n) = 3T(\lfloor n/2 \rfloor) + \frac{n}{6}$

Lösung: Wir setzen $a = 3$, $b = 2$ und $f(n) = n/6$. Dann ist $\log_b a = \log_2 3$.
Damit gilt $f(n) \in \mathcal{O}(n^{\log_2 3 - \varepsilon})$ für $\varepsilon = 0,1$. Das ist der erste Fall des Meistertheorems.
Damit ist $T(n) \in \Theta(n^{\log_2 3})$

(f) $T(n) = 3T(\lfloor n/5 \rfloor) + \frac{1}{2}\sqrt{n}$

Lösung: Wir setzen $a = 3$, $b = 5$ und $f(n) = 0,5\sqrt{n}$. Dann ist $\log_b a = \log_5 3$.
Damit gilt $f(n) \in \mathcal{O}(n^{\log_5 3 - \varepsilon})$ für $\varepsilon = 0,1$. Das ist der erste Fall des Meistertheorems.
Damit ist $T(n) \in \Theta(n^{\log_5 3})$

(g) $T(n) = 12T(\lfloor n/2 \rfloor) + n^4$

Lösung: Wir setzen $a = 12$, $b = 2$ und $f(n) = n^4$. Dann ist $\log_b a = \log_2 12 \approx 3,32$.
Damit gilt $f(n) \in \Omega(n^{3.32+\varepsilon})$ für $\varepsilon = 0,1$. Das ist der dritte Fall des Meistertheorems.
Damit ist $T(n) \in \Theta(n^4)$.
Die Regularitätsbedingung ist erfüllt:

$$\begin{aligned} 12 \cdot \frac{n}{2}^4 &\leq c \cdot n^4 \\ \frac{3n^4}{4} &\leq c \cdot n^4 \\ \frac{3}{4} &\leq c \end{aligned}$$

Wähle $3/4 < c < 1$.

(h) $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n \log n)$

Lösung: Meistermethode wegen des Θ-Terms nicht anwendbar. Deshalb Rekursionsbaum-Methode:

Ebene	Knoten	Beitrag
0	1	$n \log n$
1	2	$n \log n/2$
2	4	$n \log n/4$
i	2^i	$n \log n/2^i$
$\log_2 n$	n	n

Dadurch ergibt sich:

$$\begin{aligned} T(n) &= n + \sum_{i=0}^{\log_2 n - 1} n(\log_2 n - \log_2 2^i) \\ &= n + \left(n \sum_{i=0}^{\log_2 n - 1} \log_2 n \right) - \left(n \sum_{i=0}^{\log_2 n - 1} i \right) \\ &= n + n \log_2 n (\log_2 n - 1) - n \frac{\log_2 n (\log_2 n - 1)}{2} \\ &= n + n \frac{\log_2 n (\log_2 n - 1)}{2} \in \Theta(n \log^2 n) \end{aligned}$$

Aufgabe 4: Induktionsbeweis eines Algorithmus
Betrachten sie folgenden Algorithmus zur Berechnung der Fakultät:

Algorithmus 3: int fakultät(int k)

```
1 f = j = k
2 while j > 1 do
3 |   j = j − 1
4 |   f = f · j
5 return f
```

(a) Geben Sie einen Algorithmus in Pseudocode an, der die Fakultät rekursiv berechnet.

Lösung:

Algorithmus 4: int fakultätRek(**int** k)

```
1 if k = 0 then
2 |   return 1
3 return k · fakultätRek(k − 1)
```

(b) Beweisen Sie mittels vollständiger Induktion die Korrektheit Ihrer rekursiven Variante.

Lösung: Wir beweisen die Korrektheit mittels vollständiger Induktion über k:

Induktionsanfang Für $k = 0$ ist $k! = 1$. Das gibt genau die **if**-Abfrage in Zeile 1 zurück.

Induktionsschritt Sei der Algorithmus korrekt für ein beliebiges $k - 1$. Wir zeigen, dass der Algorithmus dann auch für k korrekt ist. Wir wissen, dass $k! = k \cdot (k-1)!$. Genau dies führt die Algorithmus in Zeile 3 durch. Da der Algorithmus für $k - 1$ korrekt ist, ist auch die Berechnung für $k! = k \cdot (k-1)!$ korrekt.

Aufgabe 5: Korrektheitsbeweise und Rekursion
Betrachten Sie den folgenden Algorithmus.

Algorithmus 5: int doSomethingSimple(**int** A[ ], **int** $i = 1$)

Data: Feld mit natürlichen Zahlen A, natürliche Zahl i
Result: Ein Wert, der mit A zusammenhängt

```
1 if i == A.length then
2     return A[i]
3 k = doSomethingSimple(A, i + 1);
4 if k > A[i] then
5     return k
6 else
7     return A[i]
```

(a) Beschreiben Sie in einem Satz, was der Algorithmus macht.

Lösung: Der Algorithmus findet das Maximum des Feldes A.

(b) Beweisen Sie die Korrektheit des Algorithmus.

Lösung:

Beweis. Wir beweisen die Korrektheit per Induktion über $n = \mathsf{A.length} - i + 1$.

Induktionsanfang Sei $n = 1$. Dann ist $\mathsf{A.length} = i$ und die **If**-Abfrage in Zeile 1 wird mit *wahr* ausgewertet. Es wird $\mathsf{A}[i]$ zurückgegeben, was das Maximum des ein-elementigen Feldes A[A.length..A.length] ist.

Induktionsschritt Angenommen, der Algorithmus ist korrekt für n. Wir zeigen, dass er auch für $n + 1$ korrekt ist. In Zeile 3 wird doSomethingSimple für ein $i + 1$ aufgerufen, das heißt, für diesen Aufruf ist $n' = \mathsf{A.length} - (i + 1) + 1 = \mathsf{A.length} - i < n$. Nach Induktionsannahme liefert Zeile 3 folglich das Maximum aus dem Teilfeld $\mathsf{A}[i + 1..\text{A-length}]$. In der nun folgenden **If**-Abfrage wird das Maximum aus $\mathsf{A}[i + 1..\text{A-length}]$ und $\mathsf{A}[i]$ ausgewählt. Damit liefert der Algorithmus insgesamt das Maximum aus $\mathsf{A}[i..\text{A-length}]$.

□

(c) Geben Sie einen Algorithmus an, der äquivalent zu doSomethingSimple ist, ohne Rekursion zu verwenden.

Lösung:

Algorithmus 6: int findMax(int[ ] A)

```
1 if A.length == 0 then
2     return ∅
3 max = A[1]
4 for i = 2 to A.length do
5     if A[i] > max then max = A[i]
```

(d) Geben Sie eine Schleifeninvariante für Ihren inkrementellen Algorithmus an.

Lösung: Vor der j. Iteration enthält max immer das Maximum des Teilfelds $\mathsf{A}[1..j]$.

(e) Beweisen Sie die Korrektheit Ihres Algorithmus mit der von Ihnen aufgestellten Schleifeninvariante.

Lösung:

Intialisierung Unmittelbar vor der ersten ($j = 1$) Iteration der **for**-Schleife steht in max der Wert von A[i]. Damit ist max das Maximum aus A[1..1], und die Invariante ist erfüllt.

Aufrechterhaltung Es gelte die Schleifeninvariante vor der j Ausführung, das heißt, max enthält das Maximum aus A[1..j]. In der j. Iteration ist i=$j + 1$ und der Wert von max wird in der **if**-Abfrage mit A[i] ausgetauscht, falls A[i] größer als max ist. Nach Auswertung von Zeile 5 enthält max das Maximum aus A[1..i]. Da $j = i+1$, enthält max das Maximum aus A[1..$j+1$], womit die Schleifeninvariante vor der $j + 1$. Iteration erfüllt ist.

Terminierung Die Schleife wird A.length-1 mal durchlaufen. Damit gilt nach der letzten Iteration wegen der Schleifeninvariante, dass max das Maximum aus A[1..A.length$-1 + 1$] enthält.

Aufgabe 6: Implementieren einer eigenen Datenstruktur

Gesucht ist eine Datenstruktur MinStack zum Verwalten einer dynamischen Menge S von Zahlen. Es sollen wie bei einem Stapel die Methoden push(key k) und pop() zur Verfügung stehen, zusätzlich eine Methode Minimum(), welche die kleinste Zahl der Menge S zurück gibt. Alle Operationen sollen in konstanter Zeit ablaufen. *Tipp*: Verwenden Sie intern mehr als eine Datenstruktur.

(a) Geben Sie eine Implementierung der Datenstruktur in Pseudocode an.

Lösung: Die Datenstruktur besitzt zwei Stacks s_1 und s_2. In s_1 werden ganz herkömmlich die Zahlen gespeichert. In s_2 wird eine Zahl nur dann gespeichert, falls sie das aktuelle Minimum ist. Ein Attribut minimum speichert das aktuelle Minimum:

Algorithmus 7: push(key k)

```
1 if k ≤ minimum then
2 |   s2.push(k)
3 |   minimum = k
4 s1.push(k)
```

Algorithmus 8: pop()

```
1 k = s1.pop()
2 if k == minimum then
3 |   s2.pop()
4 return k
```

Algorithmus 9: getMinimum()

```
1 return s2.top()
```

(b) Zeigen Sie, dass es keine Datenstruktur geben kann, die zusätzlich zu den obigen Operationen eine weitere Operation popMinimum() mit konstanter Laufzeit anbietet. Diese Operation löscht das aktuelle Minimum aus dem MinStack.

Lösung: Betrachte folgenden Sortieralgorithmus:

Algorithmus 10: sort(int[ ] A)

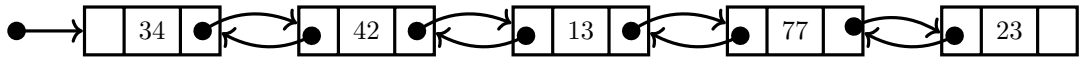
```
1 minStack = new MinStack()
2 for i = 1 to A.length do
3 |   minStack.push(A[i])
4 for i = 1 to A.length do
5 |   A[i] = minStack.popMinimum()
```

Die Laufzeit dieses Sortieralgorithmus ist offenbar linear, was im Widerspruch zum Resultat aus der Vorlesung steht, dass man zum Sortieren von n beliebigen Zahlen $\Omega(n \log n)$ Zeit braucht.

Aufgabe 7: Doppelt-Verkettete Listen
Gegeben sei folgender Algorithmus

Algorithmus 11: modifyList(List L)

```
1  item = L.head
2  size = 1
3  while item.next ≠ null do
4  |   item = item.next
5  |   size = size + 1
6  item = L.head
7  for i = 1 to ⌊size/2⌋ do
8  |   item = item.next
9  |   item.prev.next = item.next
10 |   item.next.prev = item.prev
11 |   item = item.next
```

(a) Sei $L =$ 

Zeichnen Sie die Liste für jede Iteration der Schleife in Zeile 7.

(b) Beschreiben Sie, was der Algorithmus allgemein macht.

> **Lösung:** Der Algorithmus löscht jedes zweite Element, angefangen beim zweiten, aus der Liste.

(c) Der Algorithmus enthält zwei Fehler. Geben Sie eine Liste an, sodass die Ausführung von Zeile 3 fehlschlägt. Geben Sie außerdem eine Liste an, die zu einem Fehler in Zeile 10 führt. Verbessern Sie den Pseudocode.

> **Lösung:** Falls die Liste leer ist, also L.head ist leer, dann wird ein Fehler in Zeile drei erzeugt. Zur Verbesserung muss man zu Beginn des Algorithmus eine Abfrage durchführen, ob die Liste leer ist. Ist dies der Fall, kann man die Ausführung des Algorithmus sofort abbrechen. Falls die Liste eine gerade Länge hat, wird versucht, das letzte Element zu löschen. item.next ist aber nicht definiert, sodass auf item.next.prev nicht zugriffen werden kann. Man kann dies ebenfalls durch eine entsprechende Abfrage lösen.

(d) Was würde passieren, wenn Zeile 11 gelöscht würde?

> **Lösung:** Es wird die Subliste angefangen bei Element 2 bis zum ersten Element nach der Hälfte (inklusive) gelöscht.

(e) Welche Augmentierung der Datenstruktur List schlagen Sie vor, um den Code zu verkürzen?

> **Lösung:** Durch die Verwendung eines Attributs size, welches die Länge der Liste angibt, können die ersten 5 Zeilen gelöscht werden.

Aufgabe 8: Löschen in einer Hash-Tabelle
Gegeben sei eine Hashtabelle H mit einer Hash-Funktion $h(x, i)$. Es wird offene Adressierung verwendet.

(a) Beschreiben Sie in Worten, wie der Algorithmus Search(**int** k) aus der Vorlesung funktioniert.

Lösung: Der Algorithmus bildet den Hash h von k und prüft H[h]. Wenn der Inhalt dem gesuchten Wert entspricht, wird dieser zurückgegeben, ansonsten wird i inkrementiert und mit der verwendeten Methode zur Auflösung von Kollisionen ein neuer Hash h gebildet. Das wird wiederholt, bis entweder ein leerer Eintrag oder der gesuchte Wert gefunden wird.

(b) Ein Element k soll aus der Tabelle gelöscht werden. Warum sollte man den Wert nicht mit der folgenden Befehlsfolge löschen?
$j = \mathsf{Search}(k)$
$\mathsf{H}[j] = -1$

Lösung: Nach dem in a) beschrieben Vorgehen der Methode Search(k) bricht die Suche ab, wenn ein leerer Eintrag gefunden wird. Wenn nun mitten in der Sondierfolge ein Eintrag gelöscht wird, bricht die Suche nach einem anderen, später eingefügten Element eventuell zu früh ab und findet das Element nicht mehr.

(c) Implementieren Sie die Operation Delete(**int** k), die einen Schlüssel aus der Tabelle löscht, ohne dass das Problem aus b) auftritt. *Tipp*: Verwenden Sie einen besonderen Wert, um gelöschte Zellen zu markieren.

Lösung:

Algorithmus 12: Delete(**int** k)

```
1 j = Search(k)
2 H[j] = deleted value
```

Der Wert *deleted value* ist ein Wert, der garantiert nie in die Datenstruktur eingetragen wird, aber auch nicht äquivalent zur leeren Zelle ist. Alternativ kann man auch eine zweite Hashtabelle verwalten, in der man boolesche Werte einträgt, die angeben, ob die Zelle gelöscht wurde oder nicht.

(d) Welche Änderung muss nun in den Methoden Insert(**int** k) und Search(**int** k) vorgenommen werden?

Lösung: Die Insert(**int** k)-Methode muss nun jeden Wert, der ihr während der Sondierreihenfolge begegnet, auf den *special value* überprüfen. Wenn sie einen solchen findet, darf sie ihn ersetzen. In der Search(**int** k) muss keine Änderung vorgenommen werden, da der *special value* ungleich dem leeren Wert ist. Die Suche geht also einfach über die gelöschte Zelle hinweg.

(e) Beschreiben Sie kurz die Auswirkungen Ihrer Änderungen auf die Laufzeit der Operationen.

Lösung: Die Laufzeit der Insert(**int** k) ändert sich nicht, aber die Laufzeit von Search(**int** k) verschlechtert sich, wenn das gesuchte Element nicht in der Datenstruktur vorhanden ist. Nehmen wir an, alle Werte aus einer ehemals vollen Datenstruktur wurden gelöscht. Die Search(**int** k)-Funktion muss nun trotzdem alle Felder betrachten, da in allen Feldern der *special value* steht, bevor sie **false** zurückgibt.

Aufgabe 9: Doppeltes Hashing

Welche der folgenden Funktionen eignen sich für eine Hashtabelle der Länge 25, wenn doppeltes Hashing verwendet wird und die Hashfunktion $h(k,i) = (h_0(k) + ih_1(k)) \bmod 25$ mit $h_0(k) = (4k+2) \bmod 25$ ist? Begründen Sie Ihre Entscheidungen.

(a) $h_1(k) = 1$

Lösung: Diese Hashfunktion eignet sich. Die Sondierfolge durchläuft alle Tabelleneinträge, da 1 und 25 teilerfremd sind. Diese Methode entspricht dem linearen Sondieren.

(b) $h_1(k) = 9 - (k \bmod 4)$

Lösung: Diese Hashfunktion eignet sich. Die Sondierfolge durchläuft alle Tabelleneinträge, da die Ergebnisse der Hashfunktion $\{6, 7, 8, 9\}$ alle teilerfremd zu 25 sind.

(c) $h_1(k) = k \bmod 17$

Lösung: Diese Hashfunktion eignet sich *nicht*. Die Sondierfolge durchläuft alle nicht Tabelleneinträge, da $h_1(5) = 5$ nicht teilerfremd zu 25 ist.

(d) $h_1(k) = (3 + 5k) \bmod 25$

Lösung: Diese Hashfunktion eignet sich. Die Sondierfolge durchläuft alle Tabelleneinträge, da die Ergebnisse der Hashfunktion $\{3, 8, 13, 18, 23\}$ alle teilerfremd zu 25 sind.

(e) $h_1(k) = (4k - 1) \bmod 13$

Lösung: Diese Hashfunktion eignet sich *nicht*. Die Sondierfolge durchläuft alle nicht Tabelleneinträge, da $h_1(6) = 10$ nicht teilerfremd zu 25 ist.

Aufgabe 10: Gute Sondierfolgen erkennen

Gegeben sind folgende Hashfunktionen $h(k, i)$. Geben Sie an, um welche Methode der Kollisionsauflösung es sich handelt und gegebenfalls Probleme der Sondierfolgen, bei den gegebenen Tabellengrößen, an.

(a) $h(k, i) = (h_0(k) + (i + 1) \cdot i) \mod m$, $m = 512$

Lösung:

$$h(k, i) = (h_0(k) + (i + 1) \cdot i) \mod m = (h_0(k) + i^2 \cdot i) \mod m$$

Damit benutzt die Hashfunktion quadratisches Sondieren zur Kollisionsauflösung. Dabei tritt das Problem des sekundären Clusterings auf, also falls $h_0(k) = h_0(l)$ gilt, haben k und l die gleiche Sondierfolge. Ein weiteres Problem ist, dass $(i + 1) \cdot i$ immer gerade ist und m ebenfalls gerade ist. Wenn $h_0(k)$ gerade ist, werden nur Felder mit geradem Index besucht. Wenn $h_0(k)$ ungerade ist, werden nur Felder mit ungeraden Index besucht.

(b) $h(k, i) = (h_0(k) + 2i) \mod m$, $m = 511$

Lösung: Die Hashfunktion benutzt lineares Sondieren. Da m ungerade ist, werden alle Felder besucht. Es tritt aber trotzdem das Problem des primären Clusterings auf.

(c) $h(k, i) = (h_0(k) + i \cdot h_1(k)) \mod m$, $h_1(k) = m - (2k \mod m) - 1$, $m = 16$.

Lösung: Die Hashfunktion benutzt doppeltes Hashing zur Kollisionsauflösung. Es treten keine Probleme bei dieser Hashfunktion auf, da $h_1(k)$ für alle k ungerade ist. Da m eine Zweierpotenz ist, werden alle Felder in der Sondierfolge besucht.

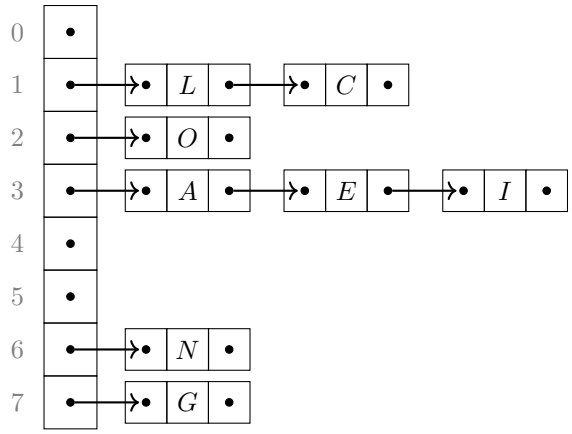
Aufgabe 11: Hashing ausführen

Gegeben seien die folgenden Schlüssel k mit entsprechenden Werten der Hashfunktion $h(k)$:

k	N	I	C	E	A	L	G	O
$h(k)$	6	3	1	3	3	1	7	2

(a) Fügen Sie die Schlüssel in obiger Reihenfolge in eine Hash-Tabelle der Größe 8 ein. Kollisionen sollen durch Verkettung aufgelöst werden.

Lösung: Da die Reihenfolge der Tabelle uns vorgibt, wann welcher Schlüssel eingefügt wurde und ein Eintrag einer Hashttabelle mit Verkettung eine einfach verkettete Liste ist, in der wir neue Elemente immer vorne einfügen, sieht die Hashtabelle folgendermaßen aus:



(b) Fügen Sie nun die Werte in der gleichen Reihenfolge in eine neue Hashtabelle mit Größe 8 ein. Lösen Sie Kollisionen mit linearem Sondieren auf. Geben Sie für jeden Schlüssel an, wie viele Felder der Tabelle betrachtet wurden.

Lösung:

Index	0	1	2	3	4	5	6	7
Schlüssel	O	C	L	I	E	A	N	G
Felder betrachtet	7	1	1	1	2	3	1	1

(c) Fügen Sie wieder die Schlüssel in eine Tabelle der Größe 8 ein. Verwenden Sie dafür folgende Hashfunktion $l(k, i)$, welche Kollisionen mittels doppelten Hashing auflöst. Wobei m die Tabellengröße ist.

$$l(k, i) = (h(k) + i \cdot h_1(k)) \mod m$$

Im Folgenden sind die Werte von $h_1(k)$ für die Schlüssel angegeben. Geben Sie für jeden Schlüssel an, wie viele Felder der Tabelle betrachtet wurden.

k	N	I	C	E	A	L	G	O
$h_1(k)$	3	5	3	7	5	1	7	3

Lösung:

Index	0	1	2	3	4	5	6	7
Schlüssel	A	C	E	I	L	O	N	G
Felder betrachtet	2	1	2	1	4	2	1	1

Aufgabe 12: Zweimal sortiertes Feld

Gegeben ist ein Feld von Zahlen $[a_1, a_2, ..., a_n]$. Dabei gilt, dass die Elemente mit ungeraden Indexen aufsteigend sortiert, es gilt also $a_1 \leq a_3 \leq a_5 \leq ...$ und die mit geraden Indexen absteigend sortiert sind. Also gilt $a_2 \geq a_4 \geq a_6 \geq ....$ Sie können davon ausgehen, dass n gerade ist. Ihre Aufgabe ist den Index einer gegeben

Zahl im Feld zu finden. Dies soll in $\mathcal{O}(\log n)$ Zeit geschehen. Geben Sie einen Algorithmus in Pseudocode an.

Lösung: Um diese Aufgabe zu lösen können zwei binäre Suchen ausgeführt werden. Einmal über die ungeraden Indexe und einmal über die geraden Indexe. Es ist zu beachten, dass die einzelnen Elemente nicht in ein anderes Feld dafür kopiert werden können, da dies schon $\mathcal{O}(n)$ Zeit brauchen würde.

Algorithmus 13: Suche(**int**[] A, **int** k)

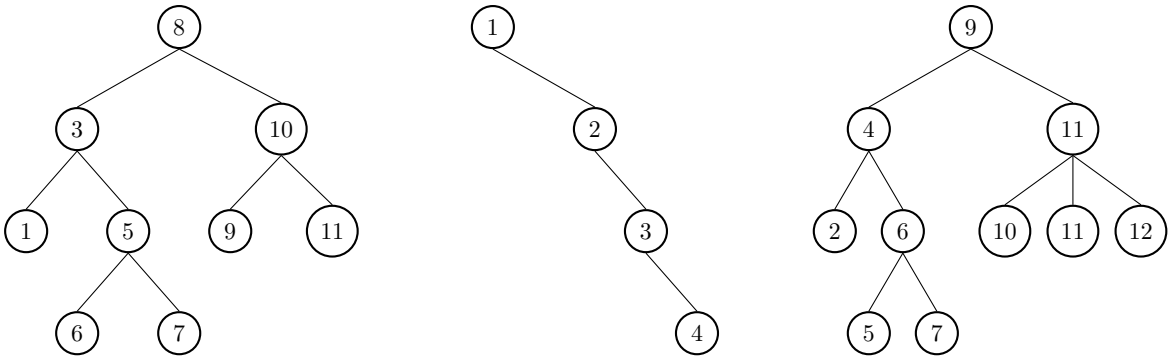
```
1  n = A.length
2   // erste binäre Suche
3  l = 1
4  r = n/2
5  while l ≤ r do
6  |  m = ⌊(l+r)/2⌋
7  |  i = m · 2 − 1
8  |  if A[i] = k then
9  |  |  return i
10 |  else
11 |  |  if a[i] < k then
12 |  |  |  l = m + 1
13 |  |  else
14 |  |  |  r = m − 1
15  // zweite binäre Suche
16 l = 1
17 r = n/2
18 while l ≤ r do
19 |  m = ⌊(l+r)/2⌋
20 |  i = m · 2
21 |  if A[i] == k then
22 |  |  return i
23 |  else
24 |  |  if a[i] > k then
25 |  |  |  l = m + 1
26 |  |  else
27 |  |  |  r = m − 1
28 return −1
```

Aufgabe 13: Binäre (Such-)Bäume – Eigenschaften

(a) Sei ein gefüllter Binärbaum ein binärer Suchbaum, dessen Knoten entweder 0 oder 2 Kinder besitzen. Angenommen ein gefüllter Binärbaum besitzt n Blätter. Wie viele Knoten hat der Baum insgesamt?

Lösung: Sei $I(n)$ die Anzahl der inneren Knoten im gesamten Baum und $X(n)$ die Anzahl aller Knoten im Baum und die Anzahl aller Kanten ist dann $X(n) - 1$. Es gilt: $X(n) = I(n) + n$. Da der Baum vollständig ist, hat jeder innere Knoten 2 Kinder, wodurch die inneren Knoten insgesamt $2 \cdot I(n)$ Kanten beitragen $\Rightarrow 2 \cdot I(n) = X(n) - 1 = I(n) + n - 1 \Rightarrow I(n) = n - 1 \Rightarrow X(n) = 2n - 1$.

(b) Finden Sie die Fehler in den verschiedenen binären Suchbäumen.



Lösung:

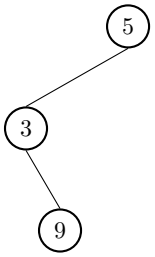
- Der erste Baum ist kein binärer Suchbaum, da $6 > 5$ aber 6 ein linkes Kind von 5 ist.
- Der zweite Baum ist ein korrekter binärer Suchbaum.
- Der dritte Baum ist kein korrekter binärer Suchbaum, da der Knoten 11 drei Kinder besitzt, aber ein binärer Suchbaum nur maximal 2 Kinder haben darf.

(c) Ein Kommilitone von Ihnen hat einen Algorithmus (Algo. 14) geschrieben, der testen soll, ob ein gegebener binärer Suchbaum T mit Wurzel r ein korrekter binärer Suchbaum ist. Geben Sie ein Gegenbeispiel an, sodass der Algorithmus Ihres Kommilitonen ein falsches Ergebnis liefert und begründen Sie kurz, wieso der Algorithmus nicht funktioniert.

Algorithmus 14: `isBinarySearchTree`(**Node** $v = r$)

```
1 if v == null then
2     return true
3 if v.left ≠ null and v.left.key ≥ v.key then
4     return false
5 if v.right ≠ null and v.right.key ≤ v.key then
6     return false
7 return isBinarySearchTree(v.left) and isBinarySearchTree(v.right)
```

Lösung: Der Algorithmus liefert ein falsches Ergebnis für folgenden binären Suchbaum:



Der Algorithmus gibt für diesen Baum *true* zurück, allerdings ist der Baum kein gültiger binärer Suchbaum, da die Suchbaum-Eigenschaft nicht erfüllt ist, denn $9 > 5$ aber 9 ist ein linkes Kind von 5. Das liegt daran, dass der Algorithmus nicht berücksichtigt, in welchem Teilbaum er sich gerade befindet und nur lokal bzw. unmittelbar die Suchbaum-Eigenschaft überprüft. Denn jeder *key* in einem linken Teilbaum von v ist nach oben durch $v.key$ beschränkt und jeder *key* in einem rechten Teilbaum von v ist nach unten durch $v.key$ beschränkt.

(d) Zeigen Sie, dass sich zwei unterschiedliche binäre Suchbäume mit gleichen Schlüsseln durch Rotationen

ineinander verwandeln lassen, egal wie sie davor aussahen. Wie viele Rotationen benötigen sie maximal? *Hinweis: Zeigen Sie zunächst, dass sich jeder Binärbaum in eine Kette verwandeln lässt.*

Lösung: Wir zeigen, dass wir mittels Rotationen einen Baum B_1 in einen anderen Baum B_2 transformieren können. Solange der Baum B_1 keine nach rechts verlaufende Kette ist (also noch linke Kinder hat), selektiere einen Knoten dieser Kette, der noch ein linkes Kind ($\neq$ nil) besitzt. Führe eine Rechtsrotation um diesen Knoten aus $\Rightarrow$ Kette wird um ein Element länger und der linke Teilbaum wird um ein Element kleiner.

Das Gleiche können wir auch für B_2 tun, wobei wir uns merken, welchen Knoten wir wann rotieren. Diese Rotationen können wir dann für B_1 invers ausführen, indem wir die letzte Rotation in B_2 als Linksrotation in B_1 ausführen (Man könnte z.B. die Rotationen in einem Stapel speichern).

Da die jeder Baum mindestens bereits ein Element auf der Kette liegen hat (die Wurzel), reichen $n-1$ Rotationen in jedem Falle aus, um einen beliebigen Binärbaum in eine Kette zu verwandeln. Somit benötigen wir insgesamt höchstens $2n-2$ Rotationen.

(e) Beweisen oder widerlegen Sie die folgende Aussage über Rot-Schwarz-Bäume: "Jeder Geschwisterknoten eines Blattknotens ist entweder selbst ein Blatt oder rot"

Lösung: Die Aussage ist wahr: Sei v der Blatt Knoten und w sein Geschwisterknoten. Wäre w ein schwarzer Geschwisterknoten, der kein Blatt ist, hätten die Kinder von w eine andere Schwarzhöhe als v. Dies ist ein Widerspruch zu (E5).

Aufgabe 14: Rot-Schwarz-Bäume verstehen

(a) Zeichnen Sie den perfekt balancierten binären Suchbaum mit den Schlüsseln $\{1,\ldots,15\}$. Färben Sie diesen Baum auf drei verschiedene Arten, sodass jeweils ein gültiger Rot-Schwarz-Baum entsteht.

Lösung: Eine korrekte Färbung erhält man, indem man alle Knoten schwarz färbt. Eine weitere Färbung erhält man, indem man beide Kinder der Wurzel rot und alle anderen Knoten schwarz färbt. Außerdem könnte man auch statt der Kinder der Wurzel nur die Kinder der Kinder der Wurzel rot färben.

(b) Angenommen wir haben einen Baum, der bis auf (E2) („Die Wurzel ist schwarz.") alle Rot-Schwarz-Eigenschaften erfüllt. Dieser Baum darf eine rote Wurzel haben. Ist der Baum ein gültiger Rot-Schwarz-Baum, wenn wir seine Wurzel ggf. schwarz färben?

Lösung: Ja. Dadurch verändern wir die Schwarz-Höhe auf allen Pfaden gleichermaßen. (E3) und (E4) können durch ein Umfärben einer roten Wurzel nicht verletzt werden.

(c) Es sei ein Binärbaum mit n Knoten gegeben. Wie viele verschiedene Rotationen können auf diesem Baum ausgeführt werden?

Lösung: Der Baum hat $n-1$ Kanten. Für jede Kante können wir auf dem Elternknoten je nach Richtung der Kante entweder eine Links- oder Rechtsrotation ausführen. Insgesamt sind also $n-1$ Rotationen möglich.

(d) Sei T ein gültiger Rot-Schwarz-Baum. Beweisen Sie, dass der längste Wurzel-Blatt-Pfad in T höchstens doppelt so lang ist, wie der kürzeste Wurzel-Blatt-Pfad in T.

Lösung: Sei ℓ die Länge eines längsten Wurzel-Blatt-Pfades in T. Wegen (E4) können höchstens $\ell/2$ Knoten auf einem solchen Pfad rot sein. Somit liegen mindestens $\ell/2$ schwarze Knoten auf dem

längsten Wurzel-Blatt-Pfad unterhalb der Wurzel. Wegen (E5) liegen auch auf dem kürzesten solchen Pfad mindestens $\ell/2$ schwarze Knoten unterhalb der Wurzel, wodurch dieser Pfad mindestens $\ell/2$ lang ist.

Aufgabe 15: InsertionSort ∪ MergeSort

Obwohl die asymptotische worst-case Laufzeit von MergeSort $\Theta(n \log n)$ ist und die asymptotische worst-case Laufzeit von InsertionSort $\Theta(n^2)$ ist, läuft InsertionSort wegen der versteckten Konstanten in der O-Notation für kleine Eingaben oft schneller. Betrachte folgende Modifikation von MergeSort: Wir benutzen MergeSort bis wir n/k Teilfelder der Größe k haben. Diese sortieren wir mit InsertionSort und mergen sie anschließend mit dem bekannten Merge-Mechanismus von MergeSort.

(a) Zeigen Sie, dass InsertionSort die n/k Teilfelder insgesamt in $\Theta(nk)$ worst-case sortieren kann!

Lösung: Für jede der n/k Teilfelder der Länge k wird InsertionSort aufgerufen, die Gesamtlaufzeit ergibt sich demnach durch

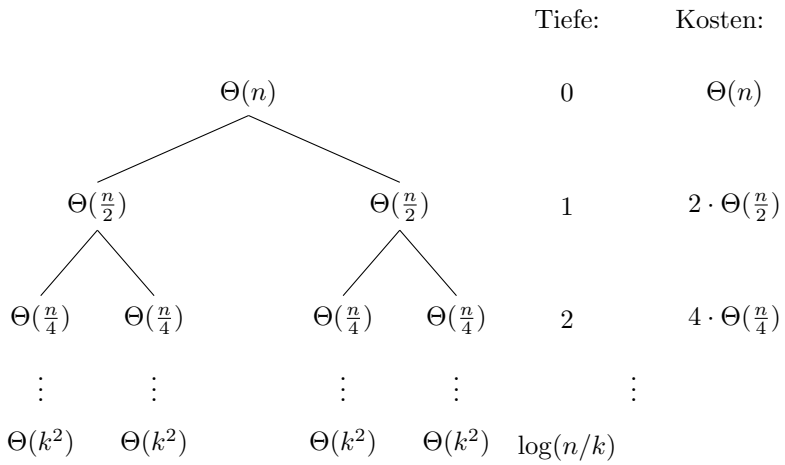
$$T(n) = \frac{n}{k} \cdot \Theta(k^2) = \Theta\left(\frac{nk^2}{k}\right) = \Theta(nk)$$

(b) Zeigen Sie, dass die oben beschriebene Variante von MergeSort in $\Theta(nk + n \log(n/k))$ worst-case läuft!

Lösung: Zur Berechnung der Laufzeit stellen wir zunächst die Rekursionsgleichung der modifizierten Variante auf:

$$T(n) = \begin{cases} \Theta(k^2) & \text{falls } n \leq k \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & \text{sonst} \end{cases}$$

Diese Rekursionsgleichung lösen wir mit der Rekursionsbaummethode und mit Hilfe von der Teilaufgabe (a).



Sobald die Tiefe $\log(n/k)$ erreicht wurde, greift der Basisfall. Aus der Teilaufgabe (a) wissen wir bereits, dass die Gesamtkosten der Tiefe $\log(n/k)$ $\Theta(nk)$ ist. Das heißt die Gesamtkosten sind

$$T(n) = \Theta(nk) + \sum_{i=0}^{\log(n/k)-1} 2^i \Theta(n/2^i) = \Theta(nk) + \log(n/k)\Theta(n) = \Theta(nk + n\log(n/k))$$

(c) Was ist der größte Wert von k als Funktion von n für die der modifizierte MergeSort Algorithmus die gleiche asymptotische worst-case Laufzeit hat wie die ursprüngliche Variante? (mit O-Notation)

Lösung: Der ausschlaggebende Teil der Laufzeit ist der $\Theta(nk)$ Teil, der nicht $\Theta(n \log n)$ übersteigen darf. $\Rightarrow k \in \mathcal{O}(\log n)$.

(d) Wie sollte k in der Praxis gewählt werden?

Lösung: Wir wollen die Laufzeit so gut es geht reduzieren, das heißt wir wollen $nk + n\log(n/k)$ minimieren.

$$\frac{\mathrm{d}}{\mathrm{d}k}(c_1 \cdot nk + c_2 \cdot n\log(n/k)) = c_1 \cdot n - \frac{c_2' n}{k} = 0 \Leftrightarrow k = \frac{c_2'}{c_1}$$

Das ist auch in der Tat das Minimum. Der Parameter k ist also am besten eine Konstante. Diese muss empirisch ermittelt werden (sogar für jede Maschine), da dieser von den Konstanten c_1 und c_2 abhängt.

Aufgabe 16: Telephonbuchsuche

Sie haben ein Telephonbuch mit Einträgen gegeben. Jeder Eintrag enthält den Namen und die dazugehörige Telephonnummer. Die Einträge sind nach Namen in alphabetischer Reihenfolge sortiert. Die Namen sind dabei paarweise verschieden und haben eine maximale Länge von m. Schreiben Sie einen Algorithmus in Pseudocode der für einen bestimmten Namen die zugehörige Telephonnummer in $\mathcal{O}(m \cdot \log n)$ Zeit findet.

Lösung: Es wird eine binäre Suche über die Telephonbuch Einträge ausgeführt. Es kann in $\mathcal{O}(m)$ Zeit bestimmt werde, ob ein Name mit einem anderen übereinstimmt oder ob dieser davor oder danach im Alphabet ist. Dabei muss solange über die Buchstaben beider Namen iteriert werden, bis diese sich unterscheiden. Damit ist dann die Gesamtlaufzeit in $\mathcal{O}(m \cdot \log n)$

Algorithmus 15: Telephonbuchsuche(**(Name, Nummer)**[] $Telephonbuch$, $GesuchterName$)

```
1  n = Telephonbuch.length
2  l = 1
3  r = n
4  while l < r do
5  |  m = ⌊(l+r)/2⌋
6  |  if Telephonbuch[m].Name = GesuchterName then
7  |  |  return Telephonbuch[m].Nummer
8  |  else
9  |  |  if Telephonbuch[m].Name > GesuchterName then
10 |  |  |  r = m − 1
11 |  |  else
12 |  |  |  l = m + 1
13 return null
```

Aufgabe 17: Ringe

Ein Ring ist eine Datenstruktur, die auf einer doppelt-verketteten Liste aufbaut. Der Unterschied zwischen beiden Datenstrukturen ist, dass beim Ring die Attribute next und prev niemals nil sind und über next eines beliebigen Elements jedes andere Element erreicht werden kann (analog auch über prev in die andere Richtung). Jeder Ring hat einen Pointer entry auf einen beliebiges Element im Ring. Ansonsten gibt es die gleichen Operationen wie bei der Liste, wobei insert(k) vor dem aktuellen entry einfügt und dann entry aufs neue Item setzt.

(a) Zeichnen Sie den Ring, der die ersten vier Fibonacci-Zahlen enthält. Der Pointer entry soll auf eine gerade Primzahl zeigen.

Lösung: $1 \leftrightharpoons 1 \leftrightharpoons 2 \leftrightharpoons 3 \leftrightharpoons 5$, sowie zwischen 5 und 1 besteht eine wechselseitige Verbindung. Der Pointer entry zeigt auf die 2.

(b) Implementieren Sie die Methode makeRing(List l), die aus einer doppelt-verketteten Liste einen Ring macht. Der Pointer entry des entstanden Rings soll dabei auf den Kopf der ursprünglichen Liste zeigen. Die Liste darf verändert werden.

Lösung:

Algorithmus 16: makeRing(List list)

```
1 tail = list.head
2 if tail = null then return
3 while tail.next ≠ null do
4 |  tail = tail.next
5 list.head.prev = tail
6 tail.next = list.head
7 ring = new Ring()
8 ring.entry = list.head
9 return ring
```

(c) Implementieren Sie die Methode split(Ring r, Item i, Item j), die den Ring r in zwei Ringe aufspaltet. Dabei sollen alle Items zwischen i und j (inklusive, in Richtung des next-Attributs) aus r gelöscht werden und als eigener Ring zurückgegeben werden. Weisen Sie die entry-Werte beliebig, aber gültig, zu. Gehen Sie davon aus, dass mindestens ein Element in r verbleibt (mit anderen Worten i.prev $\neq$ j).

Lösung:

Algorithmus 17: split(Ring r, Item i, Item j)

```
1 ring.entry = j.next
2 i.prev.next = j.next
3 j.next.prev = i.prev
4 i.prev = j
5 j.next = i
6 extracted = new Ring()
7 extracted.entry = j
8 return extracted
```

(d) Implementieren Sie die Methode merge(Ring r, Ring u), die die Items des Rings u vor r.entry einfügt. Achten Sie darauf, dass die Reihenfolge der Elemente innerhalb der Ringe gleich bleibt.

Lösung:

Algorithmus 18: merge(Ring r, Ring u)

```
1 r.entry.prev.next = u.entry.prev
2 u.entry.next.prev = r.entry.prev
3 r.entry.prev = u.entry
4 u.entry.next = r.entry
```

Aufgabe 18: Liste mit Varianz

Gegeben sei eine doppelt verkettete Liste L, die ganze Zahlen speichert.

(a) Augmentieren Sie die Liste L so, dass sie eine Methode `Mean` bereitstellt, die in $\mathcal{O}(1)$ den Durchschnitt aller Elemente in L zurückgibt.

Lösung: Speichere zusätzlich in der Liste die derzeitige Summe alle Elemente der Liste in der Variable S. Beim Einfügen eines Elements x aktualisieren wir S mit $S = S + x$ und beim Löschen eines Elements x aktualisieren wir S mit $S = S - x$. Beim Suchen und weiteren Operationen müssen wir S nicht verändern. Analog dazu halten wir auch die derzeitige Anzahl der Elemente in L in der Variable n aufrecht. Die Methode `Mean` gibt dann einfach S/n zurück, was in konstanter Zeit möglich ist.

(b) Augmentieren Sie die Liste L so, dass sie die Varianz σ^2 der Elemente in L in konstanter Zeit abgefragt werden kann. Dabei ist die Varianz von $x_1, \ldots, x_n$ folgendermaßen definiert:

$$\sigma^2 = \sum_{i=1}^{n} \frac{(x_i - \overline{x})^2}{n}, \text{ wobei } \overline{x} = \sum_{i=1}^{n} \frac{x_i}{n}$$

Lösung: Zerlegt man die Formel für die Varianz erhält man

$$\sum_{i=1}^{n} \frac{(x - \overline{x})^2}{n} = \frac{1}{n} \sum_{i=1}^{n} x_i^2 - 2x_i\overline{x} + \overline{x}^2 = \frac{1}{n} \left[\sum_{i=1}^{n} x_i^2 - 2\overline{x} \sum_{i=1}^{n} x_i + n\overline{x}^2 \right]$$

Sowohl $\overline{x}$ als auch $S = \sum_{i=1}^{n} x_i$ können wir dank der vorherigen Augmentierung in konstanter Zeit abfragen. Wir benötigen also als einzige Zusatzinformation die Summe die Quadrate. Wir speicheren diese in der Variable Q. Die Aufrechterhaltung läuft analog zur vorherigen Teilaufgabe. Die Funktion `Varianz` gibt dann

$$\frac{Q - 2 \cdot \texttt{Mean}() \cdot S}{n} + \texttt{Mean}()^2$$

zurück.

Die Laufzeit von `Insert`, `Delete` und `Search` sollen sich dabei nicht verändern.

Aufgabe 19: Traversierung von Binärbäumen

Es gibt drei gängige Arten binäre Baumstrukturen zudurchlaufen (traversieren). Diese heißen PreOrder, InOrder und PostOrder. Folgender Pseudocode zeigt eine Anwendung dieser drei Traversierungen:

Algorithmus 19:

```
1 PreOrder(Node x)
2     if x ≠ null then
3         print(x.key)
4         PreOrder(x.left)
5         PreOrder(x.right)
```

Algorithmus 20:

```
1 InOrder(Node x)
2     if x ≠ null then
3         InOrder(x.left)
4         print(x.key)
5         InOrder(x.right)
```

Algorithmus 21:

```
1 PostOrder(Node x)
2     if x ≠ null then
3         PostOrder(x.left)
4         PostOrder(x.right)
5         print(x.key)
```

Algorithmen 19-21 durchlaufen einen Binärbaum in unterschiedlichen Reihenfolgen und geben den *key* des Knoten x aus. Benutzen Sie den Binärbaum in Abbildung 1 für die folgenden Aufgaben.

(a) Wenden Sie jeweils die Algorithmen 19-21 auf den Baum in Abbildung 1 an und geben Sie den Output an!

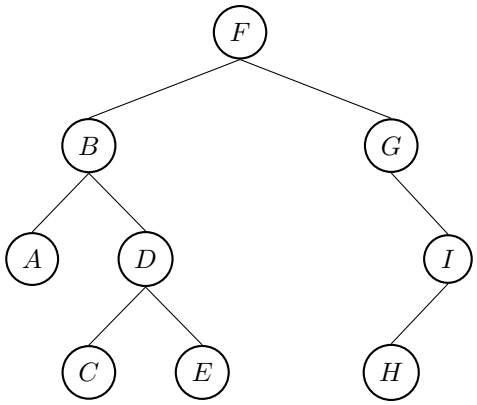


Abbildung 1: Binärbaum mit Buchstaben als *key* für Aufgabe 19.

Lösung:

- PreOrder: F, B, A, D, C, E, G, I, H
- InOrder: A, B, C, D, E, F, G, H, I
- PostOrder: A, C, E, D, B, H, I, G, F

(b) Welche rekursiven Algorithmen aus der Vorlesung kennen Sie, die die selbe Struktur haben wie die PreOrder und PostOrder Traversierungen?

Lösung:

- PreOrder: QuickSort
- PostOrder: MergeSort, MaxTeilfeld (Divide & Conquer)

(c) Neben diesen Traversierungen gibt es noch die LevelOrder Traversierung. Folgender Output wird von dieser Traversierung generiert: F, B, G, A, D, I, C, E, H. Geben Sie den Pseudocode eines Algorithmus an, der LevelOrder implementiert. Welcher Graph-Algorithmus aus der Vorlesung könnte den gleichen Output geben, wenn der Binärbaum als Graph repräsentiert wird?

Lösung: Die LevelOrder Traversierung entspricht der Breitensuche in einem Graphen mit der Wurzel als Startknoten. Immer wenn ein Knoten aus der Schlange Q geholt wird, wird dieser ausgegeben. Pseudocode:

Algorithmus 22: LevelOrder(Node x)

```
1  Q = new Queue()
2  Q.Enqueue(x)
3
4  while Q ≠ ∅ do
5      v = Q.Dequeue()
6      print(v.key)
7
8      if v.left ≠ null then
9          Q.Enqueue(v.left)
10     if v.right ≠ null then
11         Q.Enqueue(v.right)
```

Aufgabe 20: Finden von relevanten Intervallen

Angenommen Sie haben einen Sensor, der n Werte über eine Zeit in unregelmäßigen Abständen gemessen hat und diese Werte chronologisch in ein Feld A geschrieben hat. Ein Eintrag eines Feldes entspricht einem Tupel (t_i, x_i), wobei t_i die Zeit ist, an dem der Messwert x_i vom Sensor gemessen wurde. Sie wollen nun einen bestimmten Zeitraum $[t_s, t_e]$ an Messwerten abfragen. Geben Sie einen Algorithmus an, der in $o(n)$ ein Tupel von Indize (i, j) zurückgibt, sodass alle Messungen, die im Zeitraum $[t_s, t_e]$ getätigt wurden im Teilfeld $A[i..j]$ stehen.
Hinweis: Weder t_s noch t_e müssen als Werte in A existieren.

Lösung: Man kann die Indize (i, j) mittels binärer Suche finden, indem man den kleinsten Wert findet, für den gilt $t_i \geq t_s$ gilt und analog den größten Wert findet, für den gilt, dass $t_j \leq t_e$.

Algorithmus 23: `FindLowerBound`(**time** t_s, (**time**, **double**)[] A)

```
1  l, r = 1, A.length +1 // Wir setzen r so hoch, damit wir einen illegalen Wert
       bekommen, falls die untere Schranke t_s nicht exisitert.
2  /* Analog dazu kann man auch FindUpperBound formulieren. Sie Kommentare an der
       rechten Seite.                                                          */
3  while l < r do
4      m = ⌊(l+r)/2⌋
5      if A[m].time ≥ t_s then                          // A[m].time ≤ t_e
6          r = m                                                  // l = m
7      else
8          l = m + 1                                          // r = m − 1
9  return l                                                          // r
```