Aufgabensammlung ADS-Repetitorium WS 25/26

Elementare Datenstrukturen – Augmentierung – Amortisierte Analyse

Aufgabe 1: 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 1:

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

Algorithmus 2:

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

Algorithmus 3:

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

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

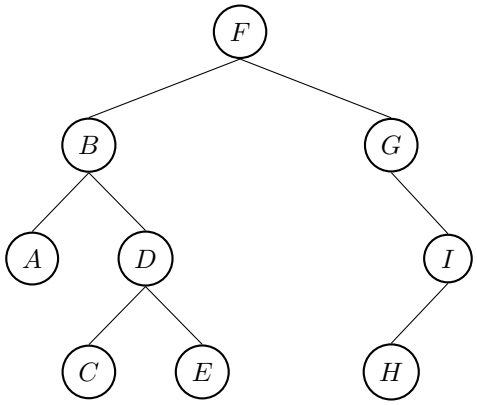


Abbildung 1: Binärbaum mit Buchstaben als *key* für Aufgabe **??**.

(a) Wenden Sie jeweils die Algorithmen **??-??** auf den Baum in Abbildung **??** an und geben Sie den Output an!

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 4: LevelOrder(Node x)

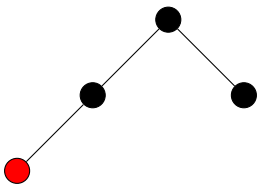
```
1  Q = new Queue()
2  Q.Enqueue(x)
3
4  while Q ≠ ∅ do
5      v = Q.Dequeue()
6      print(v.key)
7
8      if v.left ≠ null then
9          Q.Enqueue(v.left)
10     if v.right ≠ null then
11         Q.Enqueue(v.right)
```

Aufgabe 2: Rot-Schwarz-Bäume

Zeichnen Sie die folgenden Beispiele. Im Folgenden werden die *nil*-Blätter nicht mitgezählt. Ein Baum, der nur aus einer Wurzel besteht, hat demnach einen Knoten und die Höhe eins.

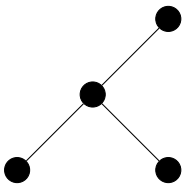
(a) Gesucht ist ein gültiger Rot-Schwarz-Baum der Höhe drei, bei dem die Anzahl der Knoten minimal ist.

Lösung:



(b) Gesucht ist ein binärer Suchbaum der Höhe drei mit maximaler Knotenzahl, für den es keine gültige Rot-Schwarz-Färbung gibt.

Lösung:



(c) Gesucht ist eine Permutation von sieben paarweise verschiedenen Zahlen, sodass diese beim Einfügen in einen Binärbaum alle Ebenen komplett ausfüllen (der Binärbaum ist also balanciert).

Lösung: Eine mögliche Permutation ist $\langle 4, 2, 1, 3, 6, 5, 7 \rangle$. Diese ist nicht eindeutig. Wichtig ist, das jeweils der Median der noch nicht eingfügten, zusammenhängenden Teillisten eingefügt wird.

Aufgabe 3: Henne-Ei-Problem

(a) Implementieren Sie eine Queue mit den Operationen dequeue() und enqueue(key k), die intern zwei Stacks verwendet.

Lösung: Idee: Unsere Queue hat intern zwei Stacke s_1 und s_2. Wir benutzen s_1, für die enqeue-Operationen und s_2 für die dequeue-Operationen. Wenn sich die Aktionen ändern, legen wir alle Elemente auf den jeweils anderen Stapel.

Algorithmus 5: dequeue()

```
1 while s1.hasElements() do
2     s2.push(s1.pop())
3 return s2.pop()
```

Algorithmus 6: enqueue(key k)

```
1 while s2.hasElements() do
2     s1.push(s2.pop())
3 return s1.push(k)
```

(b) Implementieren Sie einen Stack mit Operationen pop() und push(key k), der zwei Queues verwendet.

Lösung: Idee: Unser Stack hat intern zwei Queues q_1 und q_2. Wir benutzen q_1 und q_2 jeweils abwechselnd, um die Elemente in den Stapel einzufügen. Dabei werden die Elemente bei jeder Einfügeoperation in die jeweils andere Queue kopiert, um die Reihenfolge der Elemente umzukehren. Die Lösch-Operation liest jeweils das nächste Element aus der nicht-leeren Queue.

Algorithmus 7: pop()

```
1 if q1.hasElements() then
2     return q1.dequeue()
3 else if q2.hasElements() then
4     q2.dequeue()
5 else
6     throw error
```

Algorithmus 8: push(key k)

```
1 if q1.hasElements() then
2     q2.enqueue(k)
3     while q1.hasElements() do
4         q2.enqueue(q1.dequeue())
5 else if q2.hasElements() then
6     q1.enqueue(k)
7     while q2.hasElements() do
8         q1.enqueue(q2.dequeue())
```

Aufgabe 4: Waggons stapeln

Wir betrachten einen Zug mit n verschiedenen Güterwagons, die mit den Zahlen 1 bis n aufsteigend beschriftet sind. Wir betrachten folgenden Algorithmus:

1. Nimm vom Anfang des Zuges eine zufällige Anzahl von Waggons.
2. Schiebe diese Waggons auf das Abstellgleis.
3. Nimm eine zufällige Anzahl von Waggons auf dem Abstellgleis und schiebe sie aufs Ausfahrtsgleis.
4. Wiederhole ab 1., bis alle Waggons auf dem Ausfahrtgleis sind.

(a) Formulieren Sie den obigen Algorithmus in Pseudocode. Die Eingabe ist eine Zahl n, die Ausgabe soll ein entsprechend permutiertes Feld der Zahlen 1 bis n sein.

Lösung: Idee: Fasse Abstellgleis als Stapel auf und verwalte Waggons in doppelt-verketteter Liste, damit wir einfacher löschen können.

Algorithmus 9: stackShuffle(n)

```
1  A = doppelt verkettete Liste der Zahlen 1 bis n
2  B = Feld der Länge n mit 0ern gefüllt
3  abstell = new Stack(n)
4  untouched = n
5  finished = 0
6  while finished < n do
7      if untouched > 0 then
8          rnd = zufällige Zahl zwischen 1 und untouched
9          for i = 1 to rnd do
10             abstell.push(A.head)
11             A.remove(A.head)
12         untouched = untouched − rnd
13     rnd = zufällige Zahl zwischen 1 und abstell.size()
14     for i = 1 to rnd do
15         finished = finished + 1
16         B[finished] = abstell.pop()
17 return B
```

(b) Geben Sie einen Algorithmus an, der für eine Waggonfolge entscheidet, ob diese durch dieses Verfahren zustande gekommen ist. Beispiel: Die Folge 3 – 2 – 4 – 1 ist entstanden, indem zuerst die Wagons 1, 2 und 3 auf das Abstellgleis wanderten. Dann wurde Waggons 3 und 2 aufs Ausfahrtsgleis gestellt, danach Waggon 4 vom Einfahrtsgleis aufs das Ausfahrtsgleis umgeparkt und zuletzt Waggon 1 hinten an den Zug angehängt.

Lösung:
Die Idee ist es, den Rangiervorgang rückgängig zu machen. Dazu gehen wir das Eingabefeld rückwärts durch. Solange die Zahlen dabei auf dem Weg nach vorne steigen, speichern wie sie in einem Stapel. Dies funktioniert, da wir die kleineren Zahlen praktisch zwischenspeichern müssen. Sobald auf dem Weg von hinten die Zahlen jedoch wieder kleiner werden, nehmen wir die großen Zahlen vom Stapel und fügen sie in die Ausgabeliste ein. Wenn diese am Ende die Zahlen in korrekter Reihenfolge enthält, dann war die Eingabefolge gültig.

Algorithmus 10: checkStackShuffle(B)

```
1  result = doppelt-verkettete Liste
2  size = 0
3  abstell = new Stack(B.length)
4  index = B.length
5  while size < B.length do
6      abstell.push(B[index])
7      index = index - 1
8      while index > 0 and B[index] > abstell.top() do
9          abstell.push(B[index])
10         index = index - 1
11     while abstell.size() > 0 and B[index] < abstell.top() do
12         result.insert(abstell.pop())
13         size = size + 1
14 if result enthält Zahlen in richtiger Reihenfolge then
15     return true
16 return false
```

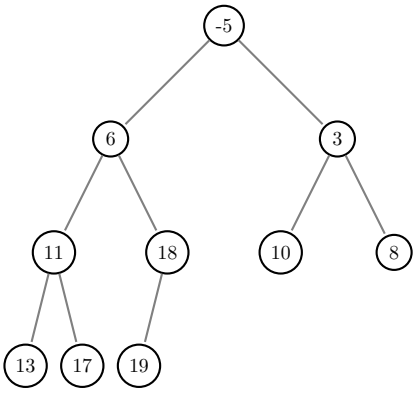
Aufgabe 5: Min-Heaps

Gegen sei folgender Min-Heap in Feld-Darstellung:

$$[-5, 6, 3, 11, 18, 10, 8, 13, 17, 19]$$

(a) Wandeln Sie den Min-Heap in Baumdarstellung um und begründen Sie, dass es sich um einen Min-Heap handelt.

Lösung:



Dies ist ein korrekter Min-Heap, da jeder Knoten kleiner als seine beiden Kindknoten ist.

(b) Fügen Sie in den Min-Heap die Zahl 15 ein. Geben Sie das Ergebnis als Feld an.

Lösung:

$$[-5, 6, 3, 11, 15, 10, 8, 13, 17, 19, 18]$$

(c) Führen Sie auf dem originalem (nicht auf dem aus Teilaufgabe **??**) enstandenem) Min-Heap die Methode ExtractMin aus. Geben Sie das Ergebnis als Feld an.

Lösung:

$$[3, 6, 8, 11, 18, 10, 19, 13, 17]$$

(d) Geben Sie an, ob folgende Aussage korrekt ist. Begründen Sie ihre Antwort:
In einem Min-Heap hat der Knoten mit dem größten Element immer maximale Tiefe.

Lösung: Die Aussage ist falsch. Betrachten Sie folgendes Gegenbeispiel:

$$[1, 2, 10, 3]$$

(e) Geben Sie an, ob folgende Aussage korrekt ist. Begründen Sie ihre Antwort:
Das drittkleinste Element in einem Min-Heap ist nicht notwendigerweise ein Kind der Wurzel.

Lösung: Die Aussage ist korrekt. Betrachten Sie folgendes Beispiel:

$$[1, 2, 10, 3]$$

Aufgabe 6: Augmentierung von Rot-Schwarz-Bäumen

Ein Rot-Schwarz-Baum zur Verwaltung einer dynamischen Menge verschiedener ganzer Zahlen soll so augmentiert werden, dass man zu jeder Zeit bestimmen kann, für welche zwei Zahlen i, j der Menge mit $i < j$ die Differenz $j - i$ am kleinsten ist. Geben Sie die Methode minGap(RedBlackTree T) in Pseudocode an, die das gesuchte Zahlenpaar in konstanter Zeit liefert. Welche Extrainformationen speichern Sie im Baum und wie lassen sich diese beim Einfügen, Löschen und Suchen aufrechterhalten, ohne die Laufzeiten der entsprechenden Methoden zu verschlechtern? Lässt sich das Problem auch mit konstantem Zusatzspeicher lösen, wenn man auf die Delete-Methode verzichtet?

Lösung: Wir augmentieren den Rot-Schwarz-Baum so, dass jeder Knoten v zusätzlich seinen Successor und den Knoten im Teilbaum mit Wurzel v, welcher unter allen Knoten im Teilbaum die minimale betragsmäßige Differenz zu seinem Successor hat. Dann löst folgender Algorithmus in $\mathcal{O}(1)$ die Aufgabe:

Algorithmus 11: minGap(RedBlackTree T)

```
1 root = T.root
2 return (root.minGapNode, root.minGapNode.successor)
```

Beim Einfügen eines neuen Knotens y wird der Successor in $\mathcal{O}(n \log n)$ Zeit und der Predecessor x von y in $\mathcal{O}(n \log n)$ berechnet. Anschließend werden in konstanter Zeit die successor-Attribute von x und y aktualisiert. Vor RBInsertFixUp ist y ein Blatt und y.minGapNode zeigt auf y. Der Wert von x.minGapNode ist nun x oder der bisherige Wert. Dies lässt sich in $\mathcal{O}(1)$ Zeit feststellen. Nun müssen die Werte nach oben bis zur Wurzel getragen werden und gegebenfalls aktualisiert werden. Das kann in $\mathcal{O}(\log n)$ geschehen. Bei den nun folgenden Rotationen können die minGapNode in konstanter Zeit neu gesetzt werden. Daher ändert sich die asymptotische Laufzeit der Einfügeoperation nicht. Das Löschen funktioniert analog.

Verzichtet man auf die Delete Methode, kann man das Problem mit konstanten Zusatzspeicher lösen. Man speichert sich einfach immer die Knoten minGapNode1 ,minGapNode2, die in dem Moment den geringsten Abstand haben. Beim Einfügen muss man lediglich in $\mathcal{O}(\log n)$ Zeit, den Predecessor und Successor des neuen Knoten bestimmen und ggf. minGapNode1 und minGapNode2 aktualisieren.

Aufgabe 7: Monotoner Stapel

Wir wollen in einem Stapel eine Menge von aufsteigenden Zahlen verwalten. Dabei wurde die `Push` Operation folgendermaßen mittels Algorithmus **??** modifiziert:

Algorithmus 12:

```
1 PushMonotone(Stack S, int x)
2     y = S.Pop()
3     while y > x and S ≠ ∅ do
4         y = S.Pop()
5     S.Push(x)
```

(a) Was ist die worst-case Laufzeit von `PushMonotone`?

> **Lösung:** Falls der Stack nur Elemente enthält, die größer sind als x, wird der gesamte Stapel geleert, bevor x eingefügt wird. Dementsprechend ist die worst-case Laufzeit linear in der Anzahl der Elemente im Stack.

(b) Zeigen Sie, dass `PushMonotone` bei einer Sequenz von n Operationen amortisiert konstant ist.

> **Lösung:** In einem Stack kann ein Element nur eingefügt und gelöscht werden. Wenn wir jedem Element beim Einfügen Kosten von 2 geben, so benutzen wir Kosten von 1, um das Element einzufügen und Kosten von 1, um dieses Element später wieder zu entfernen. Nachdem ein Element erst eingefügt werden muss, bevor es entfernt werden kann, kommen wir nicht in die Miesen. Allen anderen Operationen geben wir konstante Kosten und erhalten folglich ingesamt über eine Sequenz von n Operationen maximale Kosten von $\mathcal{O}(n)$. Demnach ist `PushMonotone` amortisiert konstant.

Aufgabe 8: 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 9: 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 14: Telephonbuchsuche(**(Name, Nummer)**[] $Telephonbuch$, $GesuchterName$)

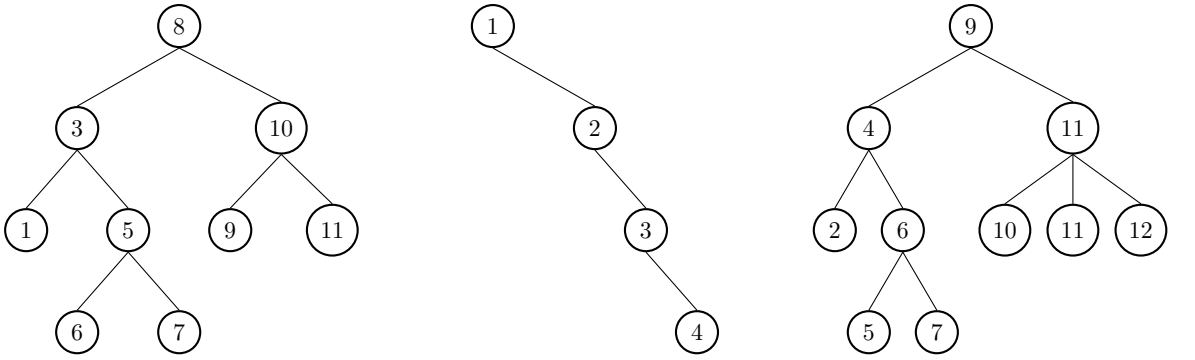
```
1  n = Telephonbuch.length
2  l = 1
3  r = n
4  while l < r do
5      m = ⌊(l+r)/2⌋
6      if Telephonbuch[m].Name = GesuchterName then
7          return Telephonbuch[m].Nummer
8      else
9          if Telephonbuch[m].Name > GesuchterName then
10             r = m − 1
11         else
12        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

Aufgabe 10: 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:

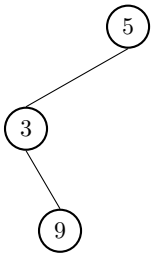
- Der erste Baum ist kein binärer Suchbaum, da $6 > 5$ aber 6 ein linkes Kind von 5 ist.
- Der zweite Baum ist ein korrekter binärer Suchbaum.
- Der dritte Baum ist kein korrekter binärer Suchbaum, da der Knoten 11 drei Kinder besitzt, aber ein binärer Suchbaum nur maximal 2 Kinder haben darf.

(c) Ein Kommilitone von Ihnen hat einen Algorithmus (Algo. **??**) 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 15: `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 11: 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 12: 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 16: push(key k)

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

Algorithmus 17: pop()

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

Algorithmus 18: 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 19: 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 13: 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 20: 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 ≠ j).

Lösung:

Algorithmus 21: 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 22: 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 14: DivContainer-Datenstruktur

In dieser Aufgabe sollen Sie eine Datenstruktur implementieren, die über zwei Operationen verfügt: insert(x) fügt eine beliebige, positive Zahl in die Datenstruktur ein und get(d), die eine beliebige Zahl aus der Datenstruktur ausgibt, die durch d teilbar ist.

(a) Geben Sie eine Implementation beider Methoden an, wenn get(d) keine Laufzeitbeschränkungen hat und

die zurückgegebene Zahl nicht aus der Datenstruktur entfernt werden soll. Welche Laufzeiten haben ihre Methoden?

Lösung: Unsere Datenstruktur besitzt intern eine doppelt-verkettete Liste list, die die üblichen Methoden hat. Dann können beide Operationen wie folgt implementiert werden:

Algorithmus 23: insert(x)

```
1 list.insert(x)
```

Algorithmus 24: get(d)

```
1 item = list.head
2 while item.next ≠ null and d ∤ item.key
    do
3   item = item.next
4 if item.next = null and d ∤ item.key then
5   throw error: value not present
6 return item.key
```

Die insert-Funktion hat konstante Laufzeit, die get(d)-Funktion hat lineare Laufzeit.

(b) Nun soll die zurückgegebene Zahl aus der Datenstruktur gelöscht werden. Wie müssen Sie Ihre Implementationen aus a) verändern, damit dies möglich wird? Ändern sich die Laufzeiten?

Lösung: Wir rufen vor Zeile 6 die Methode list.remove(item) auf, die das Löschen für uns übernimmt. Die Laufzeit ändert sich hierdurch nicht.

Aufgabe 15: Bank-Schließfächer und Heaps

Bei einer Bank können Sie Schließfächer mieten. In jedes Schließfach passt genau ein Gegenstand. Der Service ist aber nicht kostenlos. Für das Einlagern des n. Gegenstands stellt die Bank $\log n$ Euro in Rechnung. Ebenfalls werden bei der Rückgabe des n. Gegenstands $\log n$ Euro fällig.

(a) Geben Sie die Gesamtkosten für das Ein- und Auslagern den n. Gegenstands in Θ-Notation an.

Lösung: Die Kosten für das Ein- und Auslagern des n. Gegenstands sind in $\Theta(\log n)$.

(b) Wie muss die Bank ihre Bezahlpolitik ändern, sodass das Abholen der Gegenstände kostenlos ist, die Bank aber trotzdem denselben Gewinn macht?

Lösung: Die Bank verlangt die Kosten für das Abholen des Gegenstands bereits bei der Einlagerung. Die Kosten für das Einlagern des n. Gegenstands sind dann $2 \log n$.

(c) Geben Sie die Kosten für das Ein- und Auslagern mit der neuen Bezahlpolitik in Θ-Notation an.

Lösung: Die Kosten für das Hinbringen bleiben in $\Theta(\log n)$, während die Kosten für das Abholen in $\Theta(1)$ liegen.

(d) Übertragen Sie Ihre Überlegungen aus den vorherigen Teilaufgaben auf einen MaxHeap. Zeigen Sie mit der Buchhaltermethode, dass die Insert-Methode amortisiert eine Laufzeit von $\mathcal{O}(\log n)$ und ExtractMax eine konstante Laufzeit hat.

Lösung: Wir zahlen für eine Einfügeoperation der i. Zahl $\hat{e}_i = 2 \log i$ und für das Löschen der i. Zahl $\hat{l}_i = 1$. Für das Einfügen von n Zahlen und das anschließende Extrahieren zahlen wir also $\hat{c} = \sum_{i=1}^{n} 2 \log i + \sum_{i=1}^{n} 1$. Vergleichen wir das mit den tatsächlichen Worst-Case-Kosten $c = \sum_{i=1}^{n} \log i + \sum_{i=1}^{n} \log i$, so erkennen wir, dass $\hat{c} > c$. Da $\hat{c} \in \mathcal{O}(n \log n)$ und auch die tatsächlichen Kosten $c \in \mathcal{O}(n \log n)$, schließen wir daraus, dass Insert eine amortisierte Laufzeit von $\mathcal{O}(\log n)$ hat und ExtractMax amortisiert in $\mathcal{O}(1)$ liegt.

(e) Lösen Sie die Aufgabe nun mit der Potentialmethode.

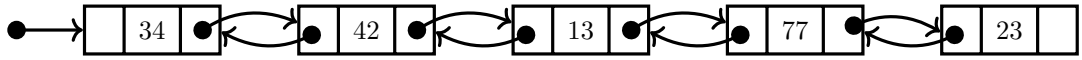
Lösung: Die Potentialmethode bei n Elementen im Heap definieren wir mit $\Phi(n) = \sum_{i=1}^{n} \log i$, also der Summe aller Höhen im Heap. Damit ist $\Phi(0) = 0$ und es gibt keinen Wert n, sodass $\Phi(k) < 0$. Nun berechnen wir die amortisierten Kosten der Operationen:
Insert: $\hat{c}_i = c_i + \Phi(n) - \Phi(n-1) = \log n + \sum_{i=1}^{n-1} \log i + \log n - \sum_{i=1}^{n-1} \log i = 2 \log n \in \mathcal{O}(\log n)$
ExtractMax: $\hat{c}_i = c_i + \Phi(n-1) - \Phi(n) = \log n + \sum_{i=1}^{n-1} \log i - (\sum_{i=1}^{n-1} \log i + \log n) = 0 \in \mathcal{O}(1)$

Aufgabe 16: Doppelt-Verkettete Listen

Gegeben sei folgender Algorithmus

Algorithmus 25: 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 17: 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 18: Graham's Scan

Gegeben sei eine Menge $S \subseteq \mathbb{R}^2$ von Punkten auf der Ebene. Wir wollen ein Polygon finden, das alle Punkte von S enthält und nur Punkte aus S als Eckpunkte enthält. Abbildung **??** zeigt ein Beispiel für das gesuchte Polygon.

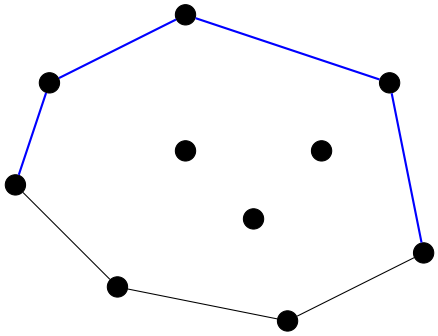


Abbildung **??**: Beispiel für S und das dazugehörige Polygon in Aufgabe **??**.

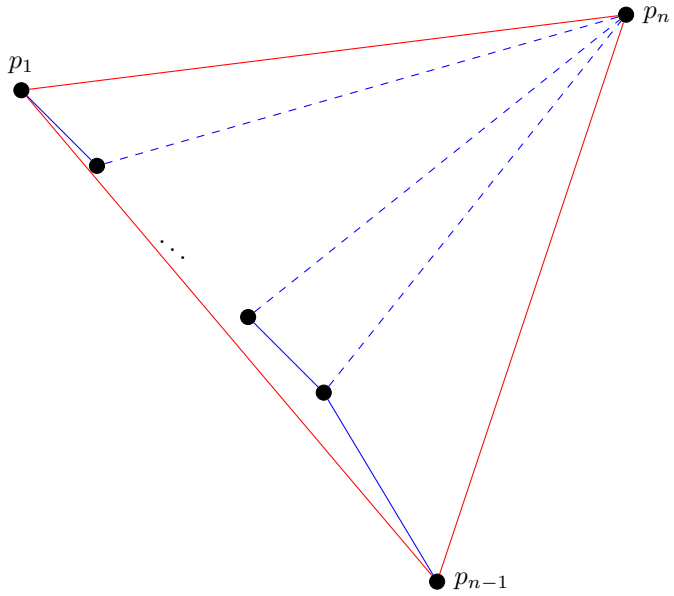
Algorithmus 26: Graham's Scan

```
1 Sortiere Punkte in S nach x-Koordinate,
    falls gefordert
2 L ← ⟨p_1, p_2⟩
3 for i = 3 to n do
4 |   L.Insert(p_i)
5 |   while |L| > 2 and die letzten 3 Punkte
  |     bilden einen links Knick do
6 |   |   entferne den vorletzten Punkt aus L
7 return L
```

Algorithmus **??** kann als Unterroutine benutzt werden, um ein solches Polygon zu finden. Dabei berechnet dieser Algorithmus die obere Hülle des Polygons. Das Beispiel in Abbildung **??** enthält die obere Hülle in blau eingezeichnet.

(a) Geben Sie eine worst-case Laufzeit für die Schleife in den Zeilen 5-6 an. Geben Sie weiterhin ein Beispiel an, für die dieser worst-case eintritt.

Lösung: Ein worst-case wäre folgendes Set S:



Hier sind $n-2$ Punkte in einer Geraden nach unten angereiht und ein weitere Punkt (p_{n-1}) ist unter beziehungweise über (p_n) dieser Geraden, sodass der Punkt p_1 ganz links auf der Geraden und die beiden Punkte über bzw. unter der Geraden das Polygon bilden. Der Algorithmus findet nun für die ersten $n-1$ Punkte keinen Linksknick, doch erst beim Einfügen des letzten Punkts (über der Gerade) entsteht ein Linksknick. Nun müssen wir die gesamte Liste L bis auf die Punkte p_1 und p_n entfernen. Folglich kann die Laufzeit der While-Schleife in den Zeilen 5-6 $\mathcal{O}(n)$ Zeit in Anspruch nehmen.

(b) Gehen Sie nun davon aus, dass S bereits nach x-Koordinate sortiert ist. Zeigen sie mittels amortisierter Analyse, dass die Laufzeit von Algorithmus **??** $\mathcal{O}(n)$ ist.

Lösung: Die Gesamtanzahl der Aufrufe des Schleifenkopfes hängt von der Länge der Elemente in L ab, da wir immer stoppen, wenn $|L| \leq 2$ ist, und wir bei einer Iteration genau ein Element löschen. Wir fügen einen Punkt p_i nur einmal in L ein und löschen diesen Punkt auch nur maximal einmal. Somit ist die Gesamtanzahl der Aufrufe des Schleifenkopfes $\mathcal{O}(n)$.

Aufgabe 19: Dynamic Table

In der Vorlesung sind wir häufig davon ausgegangen, dass unsere elementaren Datenstrukturen, wie eine Schlange oder ein Stapel, statischen Speicher haben. Das heißt, beim Anlegen der Datenstruktur geben wir bereits vor, wie viele Elemente die Struktur insgesamt speichern kann. Diese Größe ist nicht immer zu Beginn bekannt. Im Folgenden untersuchen wir, wie wir den Speicher einer Datenstruktur dynamisch vergrößern bzw. verkleinern können, sodass wir immer genug Platz haben aber gleichzeitig nicht unnötig viel Speicher verschwenden. Eine amortisierte Analyse wird zeigen, dass Insertion und Deletion dennoch $\mathcal{O}(1)$ Zeit benötigt.

Sei zunächst T eine Tabelle, die lediglich die Operation Insert unterstützt. Das Attribut T.table soll einen Pointer auf einen Speicherblock enthalten (ein Array). T.num ist die Anzahl der Elemente, die derzeitig in T enthalten sind und T.size ist die Größe des Speicherblocks. Zu Beginn gilt $T.\text{num} = T.\text{size} = 0$.

(a) Eine herkömmliche Heuristik beim Expandieren einer Tabelle ist, den vorherigen Speicher zu verdoppeln wenn die Tabelle voll ist. Mit anderen Worten: Wenn nur Insert Operationen unterstützt werden, sind immer mindestens T.size/2 Elemente in T enthalten. Schreiben Sie einen Pseudocode der die Operation Insert implementiert.

Lösung: Pseudocode:

Algorithmus 27: Insert Operation von T, die ein neues Element x einfügt

```
1  if T.size == 0 then
2      allokiere Speicher für T.table der Länge 1
3      T.size = 1
4  if T.num == T.size then
5      allokiere neuen Speicher A der Länge 2 · T.size
6      Füge alle Elemente von T.table zu A hinzu
7      T.table = A
8      T.size = 2 · T.size
9  füge x zu T.table hinzu
10 T.num = T.num + 1
```

Dieser Pseudocode ist mehr pseudo als code. Man kann zum Beispiel Zeile 6 als For Loop formulieren. In der Lösung wurde T.table absichtlich abstrakt gehalten. Falls das als herkömmliches Array implementiert wurde, ist das in Ordnung.

(b) Was ist die *worst-case* Laufzeit einer einzigen Insert Operation von Teilaufgabe (**??**)?

Lösung: Sei n die Anzahl der Elemente in T. Im worst-case ist T.table voll und somit $T.\text{num} == T.\text{size}$, wodurch T.table verdoppelt wird. Dazu müssen alle n Elemente kopiert werden und das neue Element x hinzugefügt werden. Der Gesamtaufwand beträgt also $\mathcal{O}(n)$.

(c) Betrachten Sie nun eine Folge von n Insert Operationen. Zeigen Sie mit der Aggregationsmethode, dass die Insert Operation *amortisiert* $\mathcal{O}(1)$ Zeit benötigt.
Hinweis: Eine Tabelle verdoppelt sich bei jeder Expansion. Wie häufig kann eine Expansion bei n Insert Operationen auftreten?

Lösung: Die Tabelle wird nur selten expandiert: die i-te Operation hat nur eine Expansion zur Folge, falls $i-1$ eine Zweierpotenz ist! Definiere also die Kosten der i-ten Operation als

$$c_i = \begin{cases} i & \text{falls } i-1 \text{ eine Zweierpotenz} \\ 1 & \text{sonst.} \end{cases}$$

Die Gesamtkosten sind also

$$\sum_{i=1}^{n} c_i \leq n + \sum_{j=0}^{\lfloor \log n \rfloor} 2^j < n + 2n = 3n$$

Damit haben wir amortisierte Kosten von $3 \in \mathcal{O}(1)$ pro Insert Operation.

(d) Zeigen Sie mit Hilfe der Buchhaltermethode, dass die Insert Operation *amortisiert* $\mathcal{O}(1)$ Zeit beansprucht.

Lösung: Die Buchhaltermethode kann gleichzeitig eine Intuition dafür geben, wieso die Insert Operation in der Aggregationsmethode amortisierte Kosten von 3 erhalten hat: Jedes Element das hinzugefügt wird, bezahlt für drei Operationen: die eigenen Kosten durch das Hinzufügen, die eigenen Kosten für das Rüberkopieren von sich selbst in eine erweiterte Tabelle und außerdem übernimmt das Element die Kosten für die Kopie eines weiteren Elements. Da nach einer Expansion $T.\text{size}/2$ Elemente in T enthalten sind und eine Expansion erst stattfindet, wenn $T.\text{size} == T.\text{num}$, übernimmt jedes Element, welches nach der Expansion hinzugefügt wurde genau für ein vorheriges Element die Kosten für die Kopie. Dadurch machen wir nie Schulden.

(e) Zeigen Sie mit Hilfe der Potentialmethode, dass die Insert Operation *amortisiert* $\mathcal{O}(1)$ benötigt.
Hinweis: Entwicklen Sie eine Potentialmethode, die nach einer Expansion 0 ist und sich bis zur nächsten Expansion (pro Insert Operation) auflädt. Um wie viel muss sich die Potentialfunktion pro Operation aufladen?

Lösung: Wenn wir die Potentialmethode verwenden, müssen wir uns eine Potentialfunktion Φ definieren, die uns die freiverfügbare Arbeit angibt. Die verfügbare Arbeit, müssen wir für das Kopieren der Elemente verwenden. Ähnlich wie bei der Buchhaltermethode wollen wir, dass nach einer Expansion das Potential 0 ist, also wenn $T.\text{num} = T.\text{size}/2$. Während Elemente eingefügt werden, muss sich das Potential erhöhen, sodass das Potential nach weiteren $T.\text{size}/2$ Insert Operationen alle Kopien zahlen kann. Die Anzahl der Kopien ist dann $T.\text{num} = T.\text{size}$. Das heißt für $T.\text{size}/2$ Insert Operationen muss das Potential von 0 auf $T.\text{size}$ steigen. Pro Insert Operation sollte sich das Potential also um

$$\frac{T.\text{size}}{T.\text{size}/2} = 2$$

erhöhen. Wir können die Potentialfunktion also folgendermaßen definieren:

$$\Phi(T) = 2(T.\text{num} - T.\text{size}/2)$$

Die Funktion ist 0 gleich nach einer Expansion, also wenn $T.\text{num} == T.\text{size}/2$ und es steigt um zwei mit jeder Insertion Operation, bis sie $T.\text{size}$ groß ist. Nachdem unsere Tabelle immer mindestens $T.\text{num} \geq T.\text{size}/2$ groß ist, ist unsere Funktion immer ≥ 0 und wir kommen nie in die Mießen.
Die amortisierten Kosten einer Insert Operation falls keine Expansion stattfindet sind dann:

$$\hat{c}_i = c_i + \Delta\Phi = 1 + 2 = 3$$

Sei num_i die Anzahl der Elemente nach der i-ten Operation und sei size_i die Größe der Tabelle nach der i-ten Operation, d.h. es gilt $\Phi(D_{i-1}) = 2(\text{size}_{i-1} - \text{size}_{i-1}/2) = \text{size}_{i-1} = i - 1$. Gleich nach der Expansion ist das Potential 0 aber gleich danach wird das Element x hinzugefügt, weshalb $\Phi(D_i) = 2$ und wir für $\Delta\Phi = 2 - (i-1) = 3 - i$ erhalten. Wenn wir die Tabelle expandieren müssen, haben wir echte Kosten von i (Kopie von $i-1$ Elementen + das Einfügen von Element i). Die amortisierten Kosten sind also

$$\hat{c}_i = c_i + \Delta\Phi = i + (3 - i) = 3$$

Bisher haben wir uns lediglich die Insert Operation angeschaut. Nun betrachten wir auch die Delete Operation, die ein Element von unserer Tabelle löscht. Um nicht unnötig Speicher zu besetzen, wollen wir außerdem unsere Tabelle verkleinern, wenn T.num einen bestimmten Schwellenwert unterschreitet. Diese Verkleinerung funktioniert analog zur Expansion. Falls der Schwellenwert unterschritten wird, wird ein kleinerer Speicher allokiert und die Elemente in diesen Speicher rüberkopiert, damit anschließend der vorherige Speicher freigegeben werden kann.

(f) Ein Kommilitone von Ihnen schlägt vor, dass der Schwellenwert für eine Verkleinerung bei T.num/2 sein sollte, dessen Unterschreitung in einer Halbierung der Tablle resultiert. Zeigen Sie anhand einer Beispielfolge, dass diese Strategie nicht zu amortisiert konstanten Laufzeiten für die Insert und Delete Operation führt.
Hinweis: Sie dürfen davon ausgehen, dass eine Verdopplung der Tabelle stattfindet, wenn die Tabelle voll ist, wie in den vorherigen Teilaufgaben der Fall war.

Lösung: Sei n eine Zweierpotenz und T eine Tabelle der Größe $n/2$. Die ersten $n/2$ Operationen sind Insert Operationen, sodass am Ende dieser Operationen T.num $= T$.size $= n/2$. Anschließend führen wir Operationen in der Folge Insert, Delete, Delete, Insert, Insert, ... aus.

Die erste Insertion Operationen hat eine Expansion zur Folge, sodass T.size $= n$. Die zwei aufeinanderfolgenden Delete Operationen führen zu einer Verkleinerung der Tabelle. Dies wiederholt sich bis zum Ende der Operationen. Insgesamt gibt es dann $\Theta(n)$ Expansionen bzw. Verkleinerungen, die jeweils $\Theta(n)$ Zeit benötigen, wodurch diese Folge von Operationen $\Theta(n^2)$ beansprucht, wodurch amortisierte Kosten von $\Theta(n)$ resultieren.

(g) Das Problem der Strategie des Kommilitonen war, dass nach einer Verkleinerung der Tabelle nicht genug Insertion Operationen stattfinden konnten, um für eine Expansion zu zahlen und auch umgekehrt nicht genug Delete Operationen stattfinden konnten, um für eine Verkleinerung zu zahlen. Zeigen Sie mit der Potentialmethode, dass folgende Strategie aufgeht: Wenn T.num $< T$.size/4, dann halbiere die Größe der Tabelle.

Lösung: Siehe CLRS, Amortisierte Analyse