Prof. Dr. Leif Kobbelt
Stefan Dollase, Ira Fesefeldt, Alexandra Heuschling, Gregor Kobsik

Lösung - Übung 7

Tutoraufgabe 1 (Optimaler Suchbaum):

Gegeben sind folgende Knoten mit dazugehörigen Zugriffswahrscheinlichkeiten:

Knoten	l_0	N_1	l_1	N_2	l_2	N_3	l_3	N_4	l_4	N_5	l_5
Werte	$(-\infty,1)$	1	(1,2)	2	(2,3)	3	(3,4)	4	(4,5)	5	$(5,\infty)$
Wahrscheinlichkeit	0	0.4	0	0.2	0	0.1	0	0.1	0	0.2	0

Konstruieren Sie einen optimalen Suchbaum wie folgt.

a) Füllen Sie untenstehende Tabellen für $W_{i,j}$ und $C_{i,j}$ nach dem Verfahren aus der Vorlesung aus. Geben Sie in $C_{i,j}$ ebenfalls **alle möglichen Wurzeln** des optimalen Suchbaums für $\{i,\dots,j\}$ an.

$W_{i,j}$	0	1	2	3	4	5
1						
2	–					
3	–	–				
4	–	–	–			
5	–	–	–	–		
6	–	–	–	–	–	

$C_{i,j}$ ($R_{i,j}$)	0	1	2	3	4	5
1		( )	( )	( )	( )	( )
2	–		( )	( )	( )	( )
3	–	–		( )	( )	( )
4	–	–	–		( )	( )
5	–	–	–	–		( )
6	–	–	–	–	–	

b) Geben Sie einen optimalen Suchbaum für die Knoten mit den gegebenen Zugriffswahrscheinlichkeiten und der gegebenen Reihenfolge der Knoten graphisch an.

c) Ist der optimale Suchbaum für die Knoten mit den gegebenen Zugriffswahrscheinlichkeiten und der gegebenen Reihenfolge der Knoten eindeutig? Geben Sie dazu eine kurze Begründung an.

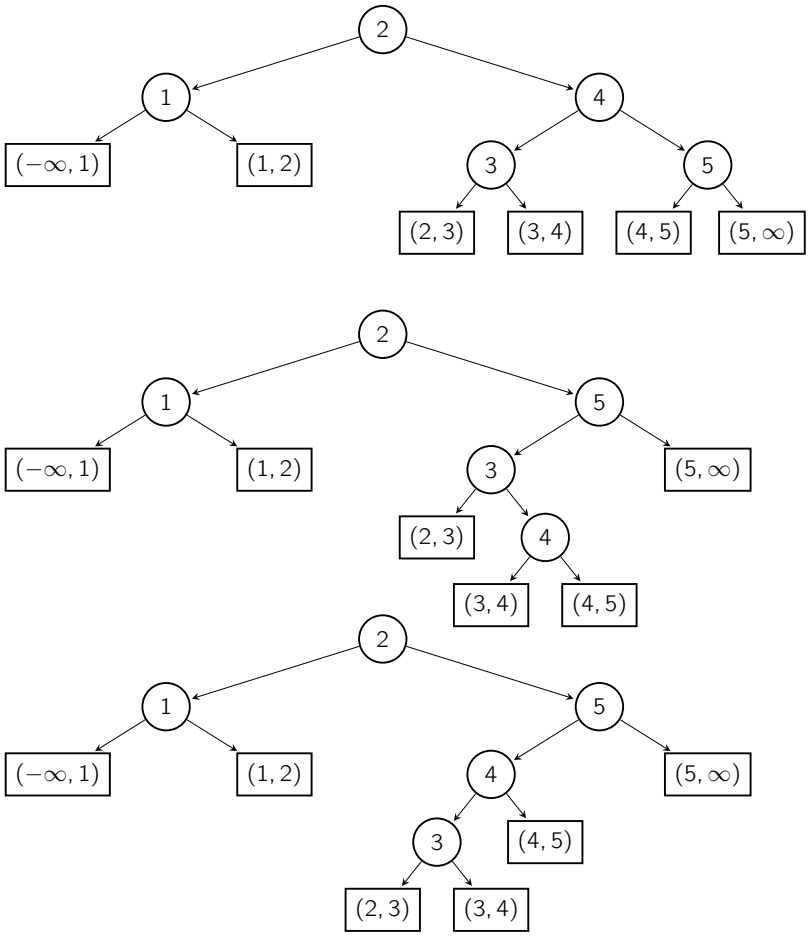
Lösung

a)

$W_{i,j}$	0	1	2	3	4	5
1	0	0.4	0.6	0.7	0.8	1.0
2	–	0	0.2	0.3	0.4	0.6
3	–	–	0	0.1	0.2	0.4
4	–	–	–	0	0.1	0.3
5	–	–	–	–	0	0.2
6	–	–	–	–	–	0

$C_{i,j}$ ($R_{i,j}$)	0	1	2	3	4	5
1	0	0.4 (1)	0.8 (1)	1.1 (1)	1.5 (1,2)	2.1 (2)
2	–	0	0.2 (2)	0.4 (2)	0.7 (2,3)	1.2 (3,4)
3	–	–	0	0.1 (3)	0.3 (3,4)	0.7 (4,5)
4	–	–	–	0	0.1 (4)	0.4 (5)
5	–	–	–	–	0	0.2 (5)
6	–	–	–	–	–	0

b) Die folgenden Lösungen sind korrekte optimale Suchbäume.



c) Der optimale Suchbaum ist nicht eindeutig, da wir aus der Tabelle drei verschiedene optimale Suchbäume konstruieren konnten.

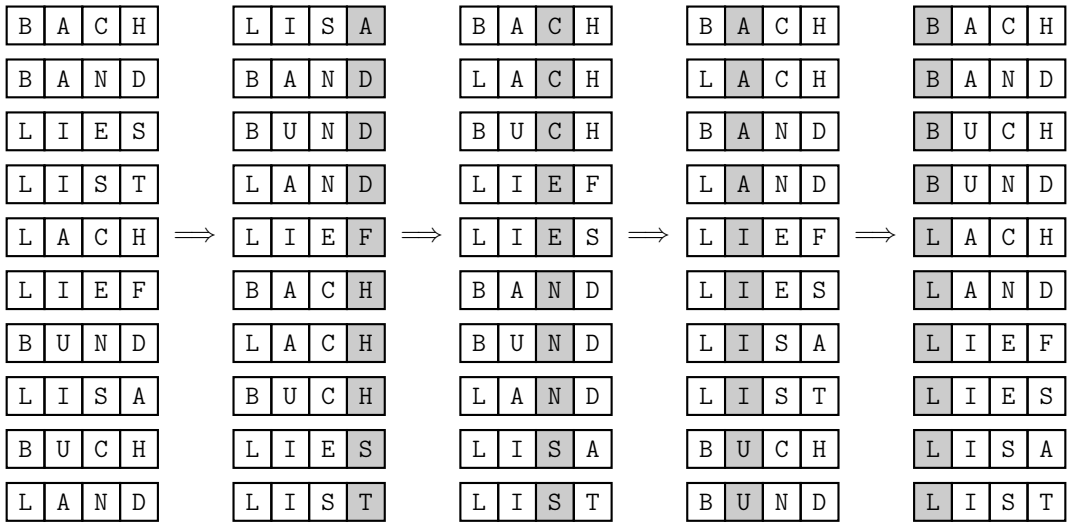
Tutoraufgabe 2 (Radixsort):

Sortieren Sie das folgende Array mithilfe von Radixsort. Geben Sie dazu das vollständige Array nach jedem Sortiertschritt an. Hierbei stellt ein Sortierschritt das einmalige Anwenden eines stabilen Sortierverfahrens auf eine der Schlüsselpositionen dar.

Sie dürfen ein beliebiges stabiles Sortierverfahren für die einzelnen Schritte verwenden. Es ist nicht notwendig dazu Zwischenschritte anzugeben.

B A C H
B A N D
L I E S
L I S T
L A C H
L I E F
B U N D
L I S A
B U C H
L A N D

Lösung



Tutoraufgabe 3 (k-Sort):

Im Folgenden beschäftigen wir uns mit fast sortierten Arrays, bei denen die Position aller Einträge nur um einen konstanten Wert von der Zielposition abweicht.
Sei a ein duplikatfreies Array der Länge n und a' das Array, welches durch das Sortieren von a entsteht. Sei außerdem V die Menge aller Elemente in a. Wir nennen i_e den Index des Elements e im Array a und i'_e den Index des Elements e im Array a' für alle $e \in V$. Es gilt also $a[i_e] = e = a'[i'_e]$. Wir nennen a nun *k-sorted* gdw. $|i'_e - i_e| < k$ für alle $e \in V$.
Das folgende Beispielarray b ist 3-sorted, aber nicht 2-sorted:

b	2	4	1	3	7	5	8	6
b'	1	2	3	4	5	6	7	8

Betrachten Sie nun den `ksort`-Algorithmus, welcher ein Array a, welches k-sorted ist vollständig sortiert.

```
def ksort(a, k):
  previous = -1
  for next in range(2 * k, len(a) + 1, 2 * k):
    swap(a, next - 2 * k, next - 1)
    current = partition(a, previous + 1, next - 1)
    heapsort(a, previous + 1, current - 1)
    previous = current
  heapsort(a, previous + 1, len(a) - 1)
```

Dazu nutzt `ksort` die Funktion `swap(a, p1, p2)`, welche schlicht im Array `a` die Werte an den Positionen `p1` und `p2` vertauscht.
Weiterhin nutzt `ksort` die Funktion `partition(a, left, right)`, welche aus der Vorlesung für den Quicksort-Algorithmus bekannt ist und welche immer `right` als initiale Pivot-Position wählt, dann die Partitionierung durchführt und anschließend die neue Pivot-Position zurückgibt.
Außerdem nutzt `ksort` die Funktion `heapsort(a, left, right)`, welche eine Variante des aus der Vorlesung bekannten Heapsort-Algorithmus darstellt mit dem einzigen Unterschied, dass nur das Teilarray zwischen `left` (inklusive) und `right` (inklusive) sortiert wird. Ansonsten hat die Funktion die selben Eigenschaften welche aus der Vorlesung für Heapsort bekannt sind.

a) Ist `ksort` ein in-place Verfahren? Begründen Sie Ihre Antwort.

b) Zeigen Sie, dass die Worst-Case Laufzeit von `ksort` nach oben durch $O(n \cdot \log(k))$ beschränkt ist. Sie dürfen dazu die aus der Vorlesung bekannten Eigenschaften für die Funktionen `swap`, `partition` und `heapsort` annehmen.

c) Begründen Sie, warum der Aufruf `ksort(a, k)` das Array `a` vollständig sortiert, falls dieses vorher bereits k-sorted war.

Lösung

a) Um die Frage zu beantworten schauen wir uns zunächst die Teile an aus denen `ksort` besteht. Die Funktionen `swap` und `partition` sind nicht rekursiv und alloziiert nur konstant viel zusätzlichen Speicher. Für Heapsort ist aus der Vorlesung bekannt, dass er in-place arbeitet. Die Funktion `ksort` ist selbst ebenfalls nicht rekursiv und alloziiert nur konstant viel zusätzlichen Speicher.

Insgesamt ist also auch `ksort` ein in-place Verfahren.

b) Der Algorithmus wählt $\lfloor \frac{n}{2k} \rfloor$ Pivot-Elemente mit Abstand $2k$, wobei das i-te Pivot-Element zu Beginn an der Position $p_i = 2k(i-1)$ steht. Zum Beispiel steht das erste Pivot-Element an der Position 0, das zweite an der Position $2k$, usw. Wir nennen die Zielposition des i-ten Pivot-Elementes im sortierten Array p'_i. Da a k-sorted ist gilt $|p'_i - p_i| < k$ und somit $|p'_{i-1} - p_i| < 3k$. Schauen wir uns nun die aufgerufenen Methoden an:

- Es wird $\lfloor \frac{n}{2k} \rfloor$ mal die Funktion `swap` aufgerufen, deren Laufzeit in $O(1)$ liegt.
- Es wird $\lfloor \frac{n}{2k} \rfloor$ mal die Funktion `partition(a, left, right)` aufgerufen, deren Worst-Case Laufzeit nach oben durch $O(\texttt{right} - \texttt{left})$, also $O(k)$ beschränkt ist.
- Es wird $\lfloor \frac{n}{2k} \rfloor + 1$ mal die Funktion `heapsort(a, left, right)` aufgerufen, deren Worst-Case Laufzeit nach oben durch $O((\texttt{right} - \texttt{left}) \cdot \log(\texttt{right} - \texttt{left}))$, also $O(k \cdot \log(k))$ beschränkt ist.

Da $1 \leq k \leq n$ gilt, erhalten wir insgesamt für die Worst-Case Laufzeit folgende obere Schranke:

$$O(\lfloor \frac{n}{2k} \rfloor + \lfloor \frac{n}{2k} \rfloor \cdot k + (\lfloor \frac{n}{2k} \rfloor + 1) \cdot k \cdot \log(k)) = O(n \cdot \log(k))$$

c) Ein formaler Korrektheitsbeweis ist recht kompliziert nachzuvollziehen, daher nennen wir hier nur die wichtigsten Bausteine eines solchen Korrektheitsbeweises um zu erkennen warum der Algorithmus korrekt ist.

- Es werden $m = \lfloor \frac{n}{2k} \rfloor$ Pivot-Element $p_1, \ldots, p_m$ an den Stellen $p_i = 2k(i-1)$ gewählt.
- Die Reihenfolge der Pivot-Elemente ist zu Beginn richtig, die Position aber noch nicht.
- Falls ein Element im vollständig sortierten Array auf der linken Seite des i-ten Pivot-Elements steht, so kann es im Eingabearray nicht auf der rechten Seite des $(i+1)$-ten Pivot-Elements gestanden haben.
- Die Partitionierung mit dem i-tem Pivot-Element sorgt dafür, dass alle Einträge mit Wert zwischen dem $(i-1)$-ten und dem i-ten Pivot-Element im Array zwischen diesen beiden Pivot-Elementen stehen. Gleichzeitig wird links vom $(i-1)$-ten Pivot-Element keine Änderung mehr vorgenommen.
- Nach den Partitionierungen haben die Pivot-Elemente bereits die richtige Position. Die Elemente zwischen den Pivot-Elementen werden noch mit Heapsort korrekt sortiert.

Tutoraufgabe 4 (Abstrakte Datentypen):

Ein abstrakter Datentyp wird definiert durch die Spezifikationen der Methoden mit denen man auf diesen Datentyp zugreifen kann. Daher kann ein abstrakter Datentyp mehr als eine Implementierung haben, die unterschiedliche Vor- oder Nachteile, insbesondere unterschiedliche Laufzeitkomplexitäten haben können. Wir haben in dieser Aufgabe abstrakte Datentypen (mit möglicherweise etwas anderen Methodennamen) mit einer unüblichen und ineffizienten Implementierung angegeben. Welche abstrakten Datentypen haben wir hier implementiert?

a) `PRIORITY_QUEUE` ist eine Prioritätswarteschlange, mit den aus der Vorlesung bekannten Methode.

```
class ADT_one:
    def __init__(self):
        self.prio_queue = PRIORITY_QUEUE()
        self.index = 0

def put(x, adt_one):
    prio_item = (adt_one.index, x)
    enqq(prio_item, adt_one.prio_queue)
    adt_one.index = adt_one.index+1

def take(adt_one):
    (prio, item) = get(adt_one.prio_queue)
    deqq(adt_one.prio_queue)
    return item
```

i. Geben Sie an, welcher abstrakte Datentyp hier implementiert wurde. Begründen Sie auch ihre Antwort.

ii. Wie könnte der abstrakte Datentyp besser implementiert werden?

b) `DoubleLinkedList` ist eine doppelt verkette Liste mit den aus der Vorlesung und Übung bekannten Methoden.

```
class ADT_two:
    def __init__(self):
        self.double_list = DoublyLinkedList()

def compute(x, adt_two):
    reset(adt_two.double_list)
    (c1,c2) = (None,None)
    while x != c1 and not isLast(adt_two.double_list):
        next(adt_two.double_list)
        element = get(adt_two.double_list)
        (c1, c2) = element if element != "Sentinel" else (None,None)
    return c2

def put(x, y, adt_two):
    reset(adt_two.double_list)
    (c1,c2) = (None,None)
    while x != c1 and not isLast(adt_two.double_list):
        next(adt_two.double_list)
```

```
        element = get(adt_two.double_list)
        (c1,c2) = element if element != "Sentinel" else (None,None)
    if x == c1:
        delete(adt_two.double_list)
    insert((x,y),adt_two.double_list)
```

i. Geben Sie an, welcher abstrakte Datentyp hier implementiert wurde. Begründen Sie auch ihre Antwort.

ii. Wie könnte der abstrakte Datentyp besser implementiert werden?

Lösung

a) Hier wird ein Stack implementiert. Fügen wir ein Element mittels `put` auf den Stack, bekommt es die höchste Priorität. Nehmen wir mittels `take` ein Element von dem Stack herunter, so nehmen wir das Element mit höchster Priorität von der Warteschlange. Aber da die höchste Priorität an das zuletzt eingefügte, noch nicht entfernte Element ging, haben wir hier das Verhalten der Stack Operationen.

Da wir hier für jede Operation logarithmische Laufzeitkomplexität haben, wäre es sinnvoller den Stack entweder als Array oder mit einer verketten Liste zu implementieren.

b) Dieser abstrakte Datentyp stellt eine Abbildung dar. Die Methode `compute` gibt für ein Element `x` das Bild wieder, welches wir als Paar in einer Liste speichern. Die Methode `put` fügt eine neue Abbildung vom Element `x` auf sein Bild `y` hinzu, indem es erst überprüft, ob die Abbildung bereits `x` auf etwas abbildet und überschreibt es unter Umständen. Hier sind statt Abbildung auch die Begriffe Map oder Funktion passend.

Eine Liste zu benutzen ist hier sicherlich ungeeignet, da so beide Operationen, `compute` und `put`, lineare Laufzeitkomplexität haben. Stattdessen sollte eine Hashtabelle genutzt werden, die mindestens im Average case bessere Laufzeiten garantieren kann.

Aufgabe 5 (Optimaler Suchbaum): 7 + 2 + 1 = 10 Punkte

Gegeben sind folgende Knoten mit dazugehörigen Zugriffswahrscheinlichkeiten:

Knoten	I_0	N_1	I_1	N_2	I_2	N_3	I_3	N_4	I_4	N_5	I_5
Werte	$(-\infty,1)$	1	(1,2)	2	(2,3)	3	(3,4)	4	(4,5)	5	$(5,\infty)$
Wahrscheinlichkeit	0	0.1	0	0.2	0	0.1	0	0.3	0	0.3	0

Konstruieren Sie einen optimalen Suchbaum wie folgt.

a) Füllen Sie untenstehende Tabellen für $W_{i,j}$ und $C_{i,j}$ nach dem Verfahren aus der Vorlesung aus. Geben Sie in $C_{i,j}$ ebenfalls **alle möglichen Wurzeln** des optimalen Suchbaums für $\{i,\dots,j\}$ an.

$W_{i,j}$	0	1	2	3	4	5
1						
2	–					
3	–	–				
4	–	–	–			
5	–	–	–	–		
6	–	–	–	–	–	

$C_{i,j}$ ($R_{i,j}$)	0	1	2	3	4	5
1		( )	( )	( )	( )	( )
2	–		( )	( )	( )	( )
3	–	–		( )	( )	( )
4	–	–	–		( )	( )
5	–	–	–	–		( )
6	–	–	–	–	–	

b) Geben Sie einen optimalen Suchbaum für die Knoten mit den gegebenen Zugriffswahrscheinlichkeiten und der gegebenen Reihenfolge der Knoten graphisch an.

c) Ist der optimale Suchbaum für die Knoten mit den gegebenen Zugriffswahrscheinlichkeiten und der gegebenen Reihenfolge der Knoten eindeutig? Geben Sie dazu eine kurze Begründung an.

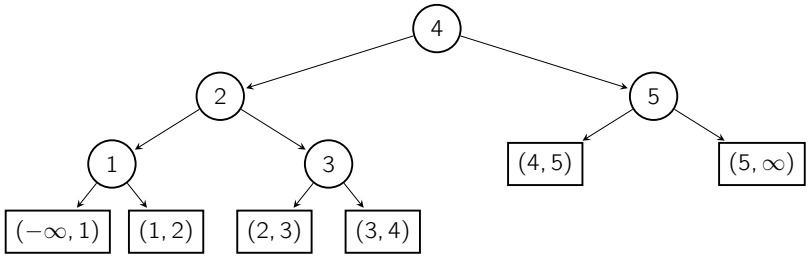
Lösung

a)

$W_{i,j}$	0	1	2	3	4	5
1	0	0.1	0.3	0.4	0.7	1.0
2	–	0	0.2	0.3	0.6	0.9
3	–	–	0	0.1	0.4	0.7
4	–	–	–	0	0.3	0.6
5	–	–	–	–	0	0.3
6	–	–	–	–	–	0

$C_{i,j}$ ($R_{i,j}$)	0	1	2	3	4	5
1	0	0.1 (1)	0.4 (2)	0.6 (2)	1.3 (2,4)	1.9 (4)
2	–	0	0.2 (2)	0.4 (2)	1.0 (4)	1.6 (4)
3	–	–	0	0.1 (3)	0.5 (4)	1.1 (4)
4	–	–	–	0	0.3 (4)	0.9 (4,5)
5	–	–	–	–	0	0.3 (5)
6	–	–	–	–	–	0

b) Nur die folgende Lösung ist ein korrekter optimaler Suchbaum.



c) Der Optimale Suchbaum ist eindeutig, denn bei der Konstruktion von $C_{i,j}$ war jedes relevante Minimum eindeutig.

Aufgabe 6 (Bucketsort): 10 Punkte

Sortieren Sie das folgende Array mithilfe von Bucketsort. Geben Sie dazu an, welche Buckets Sie verwenden, für welche Intervalle diese stehen und welche Elemente in welcher Reihenfolge in diese Buckets eingefügt werden. Geben Sie zusätzlich den Inhalt der Bucket an, nachdem diese sortiert worden sind. Zuletzt geben Sie das vollständig sortierte Array an.

Sie dürfen ein beliebiges Sortierverfahren nutzen um die einzelnen Buckets zu sortieren. Es ist nicht notwendig dazu Zwischenschritte anzugeben.

0.12	0.26	0.69	0.86	0.93	0.52	0.34	0.25	0.43	0.31

Lösung

Unsortierte Eingabe (n=10)

0.12	0.26	0.69	0.86	0.93	0.52	0.34	0.25	0.43	0.31

Eingeordnet in Buckets

0	$[0.0, 0.1)$			
1	$[0.1, 0.2)$	$\rightarrow$	0.12	
2	$[0.2, 0.3)$	$\rightarrow$	0.26	$\rightarrow$ 0.25
3	$[0.3, 0.4)$	$\rightarrow$	0.34	$\rightarrow$ 0.31
4	$[0.4, 0.5)$	$\rightarrow$	0.43	
5	$[0.5, 0.6)$	$\rightarrow$	0.52	
6	$[0.6, 0.7)$	$\rightarrow$	0.69	
7	$[0.7, 0.8)$			
8	$[0.8, 0.9)$	$\rightarrow$	0.86	
9	$[0.9, 1.0)$	$\rightarrow$	0.93	

Buckets wurden sortiert

0	$[0.0, 0.1)$			
1	$[0.1, 0.2)$	$\rightarrow$	0.12	
2	$[0.2, 0.3)$	$\rightarrow$	0.25	$\rightarrow$ 0.26
3	$[0.3, 0.4)$	$\rightarrow$	0.31	$\rightarrow$ 0.34
4	$[0.4, 0.5)$	$\rightarrow$	0.43	
5	$[0.5, 0.6)$	$\rightarrow$	0.52	
6	$[0.6, 0.7)$	$\rightarrow$	0.69	
7	$[0.7, 0.8)$			
8	$[0.8, 0.9)$	$\rightarrow$	0.86	
9	$[0.9, 1.0)$	$\rightarrow$	0.93	

Buckets wurden hintereinander gehängt

0.12	0.25	0.26	0.31	0.34	0.43	0.52	0.69	0.86	0.93

Aufgabe 7 (Mastertheorem - Wiederholung): 2 + 2 + 2 + 2 + 2 = 10 Punkte

Bestimmen Sie mithilfe des Mastertheorems die beste mögliche O-Klasse für folgende Rekursionsgleichungen. Geben Sie zusätzlich an, welches p Sie gewählt haben. Begründen Sie, warum weder durch eine andere Wahl von p, noch durch die Anwendung eines anderen Falls des Mastertheorems eine bessere O-Klasse erreicht werden kann.

a)

$$T(1) = 1$$
$$T(n) = 2 \cdot T(\frac{n}{4}) + n + 4 \qquad \text{für } n > 1$$

b)

$$T(1) = 1$$
$$T(n) = 9 \cdot T(\frac{n}{3}) + n \cdot \sqrt{n} \qquad \text{für } n > 1$$

c)

$$T(1) = 1$$
$$T(n) = 16 \cdot T(\frac{n}{2}) + 12 \cdot n^4 + 16 \cdot n^3 + 100 \qquad \text{für } n > 1$$

d)

$T(1) = 1$

$T(n) = 6 \cdot T(\frac{n}{6}) + \frac{n}{2}$ für $n > 1$

e)

$T(1) = 1$

$T(n) = 8 \cdot T(\frac{n}{4}) + \frac{n^2}{6} + n \cdot \sqrt{n}$ für $n > 1$

Lösung

a) Wir wählen $p = 1$, dann ist $n + 4 \in O(n)$ und $2 < 4$, damit ist nach dem Mastertheorem auch $T(n) \in O(n)$. Ein kleineres p kann nicht gewählt werden ohne das $n + 4 \in O(n) = O(n^p)$ gilt, damit ist dies der einzige zutreffende Fall.

b) Wir wählen $p = 1.5$, dann ist $n \cdot \sqrt{n} = n^{1.5} = O(n^{1.5})$ und $9 > 6 > 3^{1.5}$ und $\log_3(9) = 2$, somit ist nach dem Mastertheorem $T(n) \in O(n^2)$. Für den zweiten oder ersten Fall hätten wir $p \geq 2$ wählen müssen, womit die O-Klasse jedoch schlechter wäre.

c) Wir wählen $p = 4$, dann ist $12 \cdot n^4 + 16 \cdot n^3 + 100 = O(n^4)$ und $16 = 2^4$, somit ist nach dem Mastertheorem $T(n) \in O(n^4 \cdot \log(n))$. Die Wahl eines größeren p würde die O-Klasse verschlechtern. Mit einem kleineren p würde die Bedingung $12 \cdot n^4 + 16 \cdot n^3 + 100 = O(n^p)$ nicht mehr gelten.

d) Wir wählen $p = 1$, dann ist $\frac{n}{2} = O(n)$ und $6 = 6^1$, somit ist nach dem Mastertheorem $T(n) \in O(n \cdot \log(n))$. Die Wahl eines größeren p würde die O-Klasse verschlechtern. Mit einem kleineren p würde die Bedingung $\frac{n}{2} = O(n^p)$ nicht mehr gelten.

e) Wir wählen $p = 2$, dann ist $\frac{n^2}{6} + n \cdot \sqrt{n} < 2 \cdot n^2 = O(n^2)$ und $8 < 4^2$, somit ist nach dem Mastertheorem $T(n) \in O(n^2)$. Mit einem kleineren p würde die Bedingung $\frac{n^2}{6} + n \cdot \sqrt{n} < 2 \cdot n^2 = O(n^2)$ nicht mehr gelten.

Aufgabe 8 (Abstrakte Datentypen): 5 + 5 = 10 Punkte

Ein abstrakter Datentyp wird definiert durch die Spezifikationen der Methoden mit denen man auf diesen Datentyp zugreifen kann. Daher kann ein abstrakter Datentyp mehr als eine Implementierung haben, die unterschiedliche Vor- oder Nachteile, insbesondere unterschiedliche Laufzeitkomplexitäten haben können. Wir haben in dieser Aufgabe abstrakte Datentypen (mit möglicherweise etwas anderen Methodennamen) mit einer unüblichen und ineffizienten Implementierung angegeben. Welche abstrakten Datentypen haben wir hier implementiert?

a) Die Methoden `insert`, `delete` und `max` sind die Methoden für binäre Suchbäume wie in der Vorlesung und den Übungen vorgestellt.

```
class ADT_three:
    def __init__(self):
        self.root = None

def push(x,p,adt_three):
    adt_three.root = insert(adt_three.root,p,x)

def pop(adt_three):
    if adt_three.root is None:
        return None
    m = max(adt_three.root)
    adt_three.root = delete(m.key,adt_three.root)
    return m.value
```

Geben Sie an, welcher abstrakte Datentyp hier implementiert wurde. Begründen Sie auch ihre Antwort.

b)

```
class ADT_four:
    def __init__(self):
        self.dict_1 = {}
        self.dict_2 = {}

def all(adt_four):
    all_list = []
    for key in adt_four.dict_2.keys():
        if adt_four.dict_2[key]:
            all_list.append(key)
    return all_list

def all_n(v,adt_four):
    i = 0
    all_n_list = []
    while str(v)+str(i) in adt_four.dict_1:
        all_n_list.append(adt_four.dict_1[str(v)+str(i)])
        i += 1
    return all_n_list

def add(v,adt_four):
    adt_four.dict_2[v] = True

def connect(v1,v2,adt_four):
    i=0
    while str(v1)+str(i) in adt_four.dict_1 and adt_four.dict_1[str(v1)+str(i)] != v2:
        i += 1
    adt_four.dict_1[str(v1)+str(i)] = v2
```

Geben Sie an, welcher abstrakte Datentyp hier implementiert wurde. Begründen Sie auch ihre Antwort.

Lösung

a) Der abstrakte Datentyp dieser Implementierung ist eine Prioritätswarteschlange. Mittels `push` wird ein Objekt mit Priorität, bzw Schlüssel `x` in den binären Suchbaum hinzugefügt. Die Methode `pop` entfernt nun aber das Objekt mit der größten Priorität, bzw Schlüssel und gibt diesen zurück.

b) Der abstrakte Datentyp dieser Implementierung ist ein Graph in Form einer Adjazenzliste. Die Methode `all` gibt alle Knoten des Graphen zurück – diese werden in den Dictionary `dict_2` in Form eines Bitvektors gespeichert. Durch die Methode `add` können wir nun Knoten in diesem Dictionary hinzufügen. Die Methode `all_n` gibt die Adjanzentliste des Knoten `v` zurück. Die Adjanzentlisten speichern wir durchnummeriert in dem Dictionary `dict_1`. Durchnummeriert in dem Sinne, dass die benachbarten Knoten unter dem Schlüssel `v` konkateniert mit einer Zahl speichern. `connect` fügt nun eine Kante zwischen zwei Knoten hinzu, falls diese noch nicht existiert.

Dabei wird durch `connect` nicht überprüft, ob der Knoten bereits hinzugefügt worden ist. Das ist in der aktuellen Implementation nicht so drastisch, da viele Spezifikationen undefiniertes Verhalten in Fällen wie dieses erlaubt. Jedoch viel schlimmer ist, dass `connect` Kanten überschreibt, sollte die Knoten Zahlen als Namen haben: Gäbe es Knoten 1 und Knoten 11, wobei Knoten 11 eine Kante mit Index 1 und Knoten 1 eine Kante mit Index 11 haben, so wären beide Kante mit dem String 111 kodiert.