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.

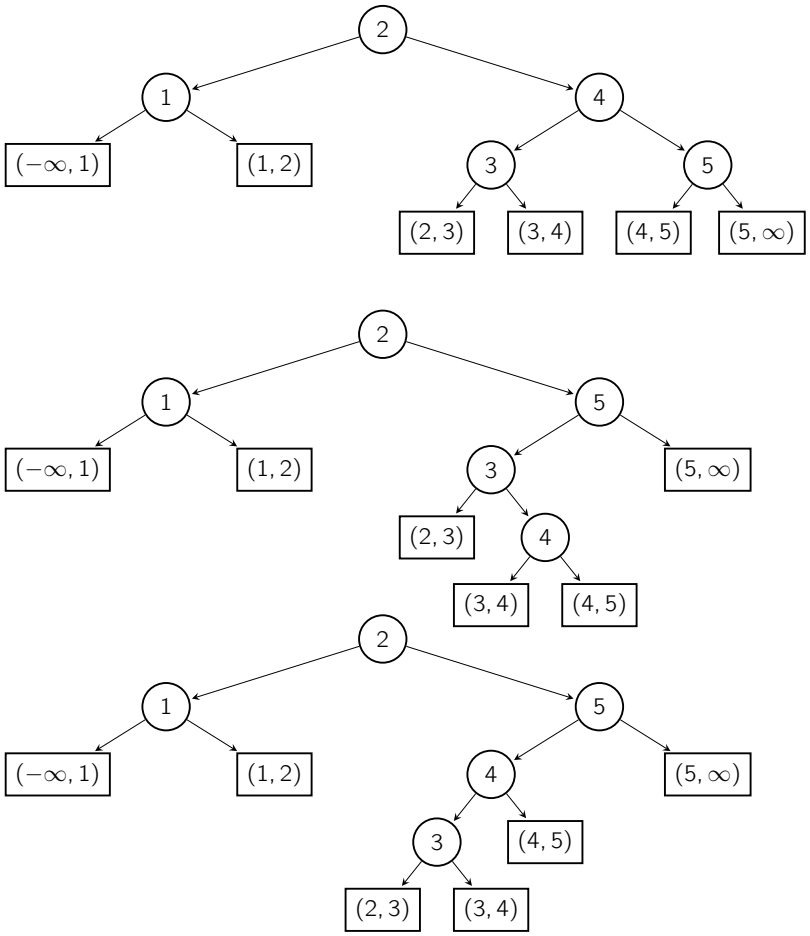
Lösung

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.

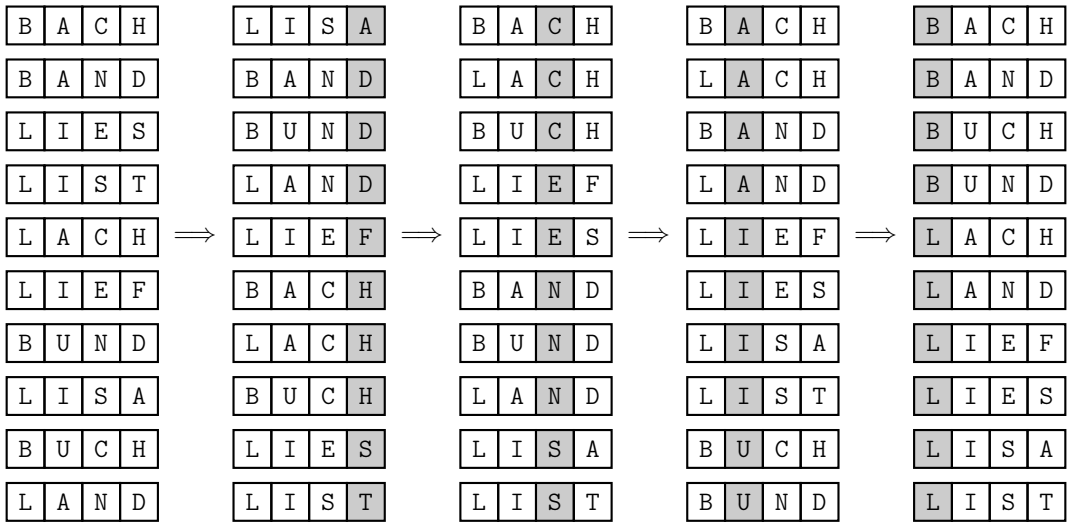
Tutoraufgabe 2 (Radixsort):

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.