TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK
Lehrstuhl für Sprachen und Beschreibungsstrukturen SS 2016
Grundlagen: Algorithmen und Datenstrukturen **Übungsblatt 9**
Prof. Dr. Helmut Seidl, J. Kranz, A. Reuss, R. Vogler 11.06.2016

Abgabe: **19.06.2016** (bis 23:59 Uhr)

Aufgabe 9.1 (P) Multiple-Choice

Kreuzen Sie in den folgenden Teilaufgaben jeweils die richtigen Antworten an. Es können pro Teilaufgabe keine, einige oder alle Antworten richtig sein.

a) Wir erweitern den Bubblesort-Algorithmus, indem vor **jeder** Vergleichsoperation eine Funktion `isSorted` aufgerufen wird. Diese überprüft, ob das gesamte Feld bereits sortiert ist, indem jedes Paar von benachbarten Elementen überprüft wird. Dazu wird das gesamte zu sortierende Feld einmal komplett durchlaufen. Ist das Feld sortiert, wird das modifizierte Sortierverfahren sofort beendet.

Es bezeichne f die **Worst-Case**-Laufzeit des Original-Bubblesort-Algorithmus und g die des modifizierten Bubblesort-Algorithmus. Was gilt?

☐ $f \in \Theta(g)$
☐ $f \notin \Theta(g)$, aber $f \in \mathcal{O}(g)$
☐ $f \notin \Theta(g)$ und $g \in \mathcal{O}(f)$
☐ weder $g \in \mathcal{O}(f)$ noch $f \in \mathcal{O}(g)$

b) Kreuzen Sie in den Zeilen (1) bis (3) jeweils das stärkste passende Symbol an. D. h. wenn z.B. $\Delta = o$ (bzw. $\Delta = \Theta$) möglich ist, wählen Sie $\Delta = o$ (bzw. $\Delta = \Theta$) und **nicht** $\Delta = \mathcal{O}$. Falls die Funktionen unvergleichbar sind, kreuzen Sie u. („unvergleichbar“) an. Setzen Sie also in jeder Zeile genau ein Kreuz!

Bsp:	$n \in \Delta(n^2)$	☒ o	☐ $\mathcal{O}$	☐ ω	☐ Ω	☐ Θ	☐ u.
(1)	$7\log_2(n) \in \Delta(4\log_2(n^2))$	☐ o	☐ $\mathcal{O}$	☐ ω	☐ Ω	☐ Θ	☐ u.
(2)	$n^3 \in \Delta(n^8(n \bmod 2))$	☐ o	☐ $\mathcal{O}$	☐ ω	☐ Ω	☐ Θ	☐ u.
(3)	$4^n \in \Delta(2^{4n})$	☐ o	☐ $\mathcal{O}$	☐ ω	☐ Ω	☐ Θ	☐ u.

Hinweis: Der Operator mod berechnet den Divisions-Rest.

c) Welche Aussagen sind wahr?

☐ Jeder AVL-Baum ist zugleich ein binärer Suchbaum.
☐ Jeder binäre Suchbaum ist zugleich ein Binärer Heap.
☐ Jeder Binäre Heap ist zugleich ein AVL-Baum.
☐ Jeder Binäre Heap ist zugleich ein binärer Suchbaum.

d) Welche Operation muss in einem Binären Min-Heap beim Ausführen von decreaseKey (Verringern eines Schlüssel-Wertes) u.U. aufgerufen werden, um die Heap-Invariante wiederherzustellen?

☐ `siftUp` ☐ `siftDown` ☐ `deleteMin` ☐ `rotateLeft`

e) Die durchschnittliche Laufzeit von Algorithmen (*Average Case*) ist

☐ uninteressant, da man ausschließlich am *Worst-Case* interessiert ist.

☐ oft nur mit großem Aufwand zu berechnen.

☐ stets am besten geeignet, um den passenden Algorithmus zu wählen.

f) Der MergeSort-Algorithmus

☐ ist für alle Eingaben schneller als jede gute Implementierung von InsertionSort.

☐ ist für bestimmte Eingabeklassen signifikant schneller als eine deterministische Implementierung von Quicksort.

☐ ist im Schnitt um einen in der Eingabe linearen Faktor schneller als Quicksort.

☐ sortiert eine Eingabe im *Best Case* in linearer Zeit.

g) Beim Hashing mit *linear probing*

☐ werden Elemente, deren Schlüssel auf den gleichen Wert gehasht werden, in einer Liste abgelegt.

☐ ist die Hashfunktion nicht besonders wichtig, da auf ineffiziente Listen verzichtet wird.

☐ ist das Löschen von Elementen kompliziert, da Löcher in der Hashtabelle das Auffinden von anderen Elementen verhindern können.

h) Der PermutationSort-Algorithmus sortiert ein Feld, indem er die Elemente wiederholt umordnet und jede so entstandene Anordnung auf Sortiertheit prüft. Es werden systematisch alle möglichen Anordnungen durchprobiert. Wir gehen im Folgenden davon aus, dass PermutationSort ein Feld ohne Duplikate sortiert.

☐ Für die *Worst-Case*-Laufzeit f von PermutationSort gilt $f \in \mathcal{O}(n^4)$.

☐ Für die *Average-Case*-Laufzeit f von PermutationSort gilt $f \in \Theta(n * n!)$.

☐ Für die *Worst-Case*-Laufzeit f von PermutationSort gilt $f \in \mathcal{O}(n^n)$.

☐ Mit einer sehr geringen, positiven Wahrscheinlichkeit terminiert PermutationSort für bestimmte Eingaben niemals.

☐ PermutationSort hat eine lineare *Best-Case*-Laufzeit.

Aufgabe 9.2 (P) Abbaumbaumaßnahmen

Führen Sie auf einem anfangs leeren $(2, 3)$-Baum die folgenden Operationen aus:

a) insert(73)

b) insert(45)

c) insert(85)

d) insert(0)

e) insert(39)

f) insert(90)

g) insert(42)

h) insert(91)

i) insert(86)

j) insert(68)

k) remove(86)

l) remove(73)

m) remove(0)

n) remove(91)

o) remove(68)

p) remove(39)

q) remove(90)

r) remove(42)

s) remove(85)

Hinweis: Zeichnen Sie den Baum nach jedem Schritt. Sie dürfen in Ihrer Zeichnung auf Blattknoten verzichten.

Beachten Sie außerdem das Folgende:

- Beim Aufspalten von Knoten während dem Einfügen wandert das Element am Index $\lfloor b/2 \rfloor$ nach oben.
- Beim Löschen von Elementen aus inneren Knoten wird üblicherweise versucht, entweder den symmetrischen Vorgänger oder symmetrischen Nachfolger intelligent zu wählen. Für diese Aufgabe soll darauf verzichtet werden. Stattdessen wird stets der symmetrische Vorgänger verwendet. Beim *Stehlen* von Elementen wird zunächst der linke Nachbar betrachtet. Beim *Verschmelzen* werden Knoten mit ihrem linken Nachbarn vereinigt.

Aufgabe 9.3 (P) Graph-Traversierung

Gegeben sei ein ungerichteter Graph $G = (V, E)$ mit $V = \{0, 1, 2, \ldots, 9\}$. Die Kantenmenge E sei gegeben durch folgende Adjazenzlisten:

0: 5	1: 2,5,8	2: 1,3,7
3: 2,4,9	4: 3,5,7	5: 0,1,4,6,9
6: 5	7: 2,4	8: 1
9: 3,5		

a) Geben Sie eine entsprechende Adjazenzmatrix für den Graphen an.

b) Führen Sie eine Tiefensuche (DFS) auf dem Graphen durch. Beginnen Sie bei Knoten 0, die Adjazenzlisten werden von links nach rechts abgearbeitet. Geben Sie die Besuchsreihenfolge, dfsNum und finishNum an.

c) Führen Sie eine Breitensuche auf dem angegebenen Graphen beginnend bei Knoten 0 aus. Geben Sie in jedem Schritt an, welche Knoten sich in der Queue befinden und illustrieren Sie Ihr Vorgehen graphisch.

Aufgabe 9.4 [5 Punkte] **(H) (a,b)-Baumbau**
Gegeben seien ein aufsteigend sortiertes Feld x der Länge n sowie a und b.

a) Geben Sie eine möglichst genaue obere Grenze für die Tiefe t (in Kanten) eines (a,b)-Baumes mit $n \geq 1$ Elementen an.

b) Beschreiben Sie einen Algorithmus, der daraus mit einer Laufzeit in $\mathcal{O}(n)$ einen (a,b)-Baum erzeugt.

c) Zeigen Sie, warum Ihr Algorithmus in $\mathcal{O}(n)$ arbeitet.

d) Führen Sie den Algorithmus für das folgende Beispiel aus und zeichnen Sie das Ergebnis: $x = [0, 1, 2, 3, 4, 5, 6, 7, 8], a = 2, b = 3$.

Aufgabe 9.5 [5 Punkte] **(H) Bäumchen über Bäumchen!**
Führen Sie auf einem anfangs leeren $(2, 4)$-Baum die folgenden Operationen aus:

a) insert(9)

b) insert(120)

c) insert(108)

d) insert(126)

e) insert(87)

f) insert(89)

g) insert(123)

h) insert(82)

i) insert(76)

j) insert(48)

k) insert(59)

l) insert(105)

m) insert(34)

n) remove(82)

o) remove(108)

p) remove(76)

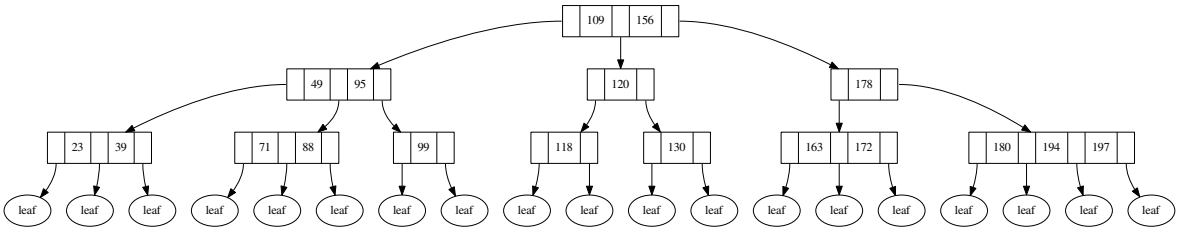

Abbildung 1: Beispiel eines $(2, 4)$-Baumes

q) remove(120)

r) remove(59)

s) remove(34)

t) remove(87)

u) remove(105)

v) remove(89)

w) remove(123)

x) remove(48)

Hinweis: Zeichnen Sie den Baum nach jedem Schritt. Sie dürfen in Ihrer Zeichnung auf Blattknoten verzichten.

Beachten Sie außerdem das Folgende:

- Beim Aufspalten von Knoten während dem Einfügen wandert das Element am Index $\lfloor b/2 \rfloor$ nach oben.
- Beim Löschen von Elementen aus inneren Knoten wird üblicherweise versucht, entweder den symmetrischen Vorgänger oder symmetrischen Nachfolger intelligent zu wählen. Für diese Aufgabe soll darauf verzichtet werden. Stattdessen wird stets der symmetrische Vorgänger verwendet. Beim *Stehlen* von Elementen wird zunächst der linke Nachbar betrachtet. Beim *Verschmelzen* werden Knoten mit ihrem linken Nachbarn vereinigt.

Aufgabe 9.6 [10 Punkte] (H) Programmieraufgabe: Mauba

In dieser Aufgabe geht es darum, einen (a,b)-Baum zu implementieren. Es soll dabei nur die Baumstruktur, nicht jedoch die darunterliegende Liste implementiert werden. Stattdessen befinden sich auf der Blattebene Instanzen der Klasse `ABTreeLeaf`, die selbst keine Daten halten (vgl. Abbildung 1). Ein Vorlage für Ihre Implementierung finden Sie im Ordner `mauba-angabe/`. Kommentare im gegebenen Quelltext enthalten wichtige Hinweise und Beispiele.

Gehen Sie vor wie folgt:

a) Die eingebetteten Klassen der Klasse `ABTree` sind nicht statisch. Wieso ist dies hier sinnvoll?

b) Implementieren Sie die Methode `int height()` der Klasse `ABTree`. Die Methode ermittelt die Höhe des Baumes.

c) Implementieren Sie die Methode `boolean validAB()` der Klasse `ABTree`, die Ihnen beim Debuggen hilft. Sie testet, ob der (a,b)-Baum alle Forderungen an (a,b)-Bäume erfüllt. Konkret wird folgendes geprüft:

- Die Höhe aller Teilbäume eines Knotens ist gleich.
- Jeder innere Knoten (außer der Wurzel, für Details siehe Vorlesung) hat mindestens a und höchstens b viele Kinder.
- Die Schlüssel jedes Knotens sind aufsteigend sortiert. Ein Kindbaum eines Knotens zwischen den Schlüsseln e_1 und e_2 enthält lediglich Schlüssel e mit $e_1 < e < e_2$. Alle Schlüssel des linkesten Teilbaums (an Index 0) sind kleiner als alle Schlüssel des aktuellen Knotens, alle Schlüssel des rechtesten Teilbaums sind größer als alle Schlüssel des aktuellen Knotens.

Sie dürfen dieser Methode, wenn nötig, weitere Parameter hinzufügen.

d) Implementieren Sie die Methode `boolean find(int key)` der Klasse `ABTree`, die einen Schlüssel im (a,b)-Baum sucht. Sie gibt zurück, ob der Schlüssel gefunden wurde.

e) Implementieren Sie die Methode `void insert(int key)` der Klasse `ABTree`, die einen Schlüssel in den (a,b)-Baum einfügt. Achten Sie darauf, dass der Baum nach dem Einfügen alle Forderungen an (a,b)-Bäume erfüllt.

Hinweis: Beim Aufspalten von Knoten wandert das Element am Index $\lfloor b/2 \rfloor$ nach oben.

f) Implementieren Sie die Methode `boolean remove(int key)` der Klasse `ABTree`, die einen Schlüssel aus dem (a,b)-Baum löscht. Achten Sie darauf, dass der Baum nach dem Löschen alle Forderungen an (a,b)-Bäume erfüllt. Die Methode gibt zurück, ob der Schlüssel im Baum gefunden und gelöscht wurde oder nicht.

Hinweis: Beim Löschen von Elementen aus inneren Knoten wird üblicherweise versucht, entweder den symmetrischen Vorgänger oder symmetrischen Nachfolger intelligent zu wählen. Für diese Aufgabe soll darauf verzichtet werden. Stattdessen wird stets der symmetrische Vorgänger verwendet. Beim *Stehlen* von Elementen wird zunächst der linke Nachbar betrachtet. Beim *Verschmelzen* werden Knoten mit ihrem linken Nachbarn vereinigt.

Bitte beachten Sie, dass Sie für eine funktionierende Implementierung die Klasse `ABTree` (und auch die eingebetteten Klassen) gegebenenfalls noch an anderen Stellen als innerhalb der jeweiligen Methoden verändern müssen. Es empfiehlt sich, die Aufgaben in Teilprobleme zu zerlegen und diese in Hilfsmethoden zu implementieren. Sie dürfen davon ausgehen, dass keine Duplikate in den Baum eingefügt werden.

Die Klasse `ABTree` enthält die Methode `String dot()`, die den Baum in das Graphviz-Format[1] umwandelt. Es ist sehr empfehlenswert, sich die eigenen Graphen z.B. mit Xdot[2] anzusehen, um Fehler besser zu verstehen.

[1] `http://www.graphviz.org/`
[2] `https://github.com/jrfonseca/xdot.py`