Technische Universität Wien
Institut für Computergraphik und Algorithmen
Arbeitsbereich für Algorithmen und Datenstrukturen

186.172 Algorithmen und Datenstrukturen 1 VL 4.0

Nachtragstest SS 2009

30. Juni 2009

Machen Sie die folgenden Angaben bitte in deutlicher Blockschrift:

Nachname: _____ Vorname: _____

Matrikelnummer: _____ Studienkennzahl: _____

Anzahl abgegebener Zusatzblätter: _____

Legen Sie bitte Ihren Studentenausweis vor sich auf das Pult.

Sie können die Lösungen entweder direkt auf die Angabeblätter oder auf Zusatzblätter schreiben, die Sie auf Wunsch von der Aufsicht erhalten. Es ist nicht zulässig, eventuell mitgebrachtes eigenes Papier zu verwenden.

Die Verwendung von Taschenrechnern, Mobiltelefonen, Skripten, Büchern, Mitschriften, Ausarbeitungen oder vergleichbaren Hilfsmitteln ist unzulässig.

Die Arbeitszeit beträgt 55 Minuten.

	A1:	A2:	A3:	Summe:
Erreichbare Punkte:	16	14	20	50
Erreichte Punkte:				

Viel Erfolg!

Aufgabe 1.A: Optimierung **(16 Punkte)**

a) (12 Punkte) Gegeben sei ein Rucksack mit Kapazität 5 sowie die unten angegebenen sieben Gegenstände. Diese Instanz des 0/1-Rucksackproblems soll mittels *Dynamischer Programmierung* über die möglichen Gesamtgewichte gelöst werden. Dazu wird die unten stehende 8×6-Matrix $\mathbf{m}$ verwendet, wobei der Eintrag im Feld $m_{i,j}$ angibt, welcher Wert mit den ersten i Gegenständen erreicht werden kann, wenn das Gesamtgewicht der gewählten Gegenstände kleiner oder gleich j ist.

Die Felder der Matrix können wie folgt berechnet werden:

$$m_{0,j} = 0 \qquad \text{für } j = 0, \ldots, 5$$

$$m_{i,j} = \begin{cases} m_{i-1,j} & \text{falls } w_i > j, \\ \max\{m_{i-1,j-w_i} + c_i,\ m_{i-1,j}\} & \text{sonst.} \end{cases} \qquad \text{für } \begin{cases} i = 1, \ldots, 7 \\ j = 0, \ldots, 5 \end{cases}$$

Gegenstand i	Gewicht w_i	Wert c_i
1	2	5
2	3	4
3	6	10
4	2	2
5	1	3
6	3	3
7	2	3

$i \backslash j$	0	1	2	3	4	5
0	0	0	0	0	0	0
1						
2						
3						
4						
5						
6						
7						

- Lösen Sie die gegebene Instanz des 0/1-Rucksackproblems, indem Sie die obenstehende Matrix vervollständigen.
- Geben Sie eine optimale Lösung der Probleminstanz an, und markieren Sie in der Matrix jene Zellen, die beim Rekonstruieren der Lösung betrachtet werden.

b) (2 Punkte) Kreuzen Sie an, ob die folgenden Aussagen korrekt sind:

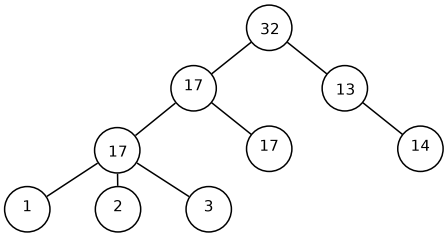
Die MST-Algorithmen von Prim und Kruskal liefern immer das selbe Ergebnis ...

... bezogen auf das Gesamtgewicht ☐ Ja ☐ Nein
... bezogen auf die ausgewählten Kanten ☐ Ja ☐ Nein

c) (2 Punkte) Beschreiben Sie *kurz* die Unterschiede und Gemeinsamkeiten von *Teile und Herrsche* und *Dynamischer Programmierung*.

Aufgabe 2.A: Sortieren **(14 Punkte)**

a) (4 Punkte) Eigentlich sollte die folgende Abbildung einen **Maximum-Heap** (wie aus der LVA bekannt als Datenstruktur für Heap-Sort) als Baum darstellen, allerdings haben sich einige Fehler eingeschlichen. Markieren Sie die Fehler und beschreiben Sie diese *kurz*.



b) (8 Punkte) Sortieren Sie die folgenden Zahlen **aufsteigend** mittels **Quick-Sort**:

$$\langle 4, 1, 2, 3, 7, 8, 6, 5 \rangle.$$

Geben Sie die Zahlenfolge nach **jedem Aufruf** von `Partition()` an. Markieren Sie in jeder dieser Folgen die Zahlen, die sich bereits an ihrer endgültigen Position befinden.

4	1	2	3	7	8	6	5

Hinweis: Nicht jede Zeile der Tabelle muss zwangsläufig
für eine korrekte Lösung verwendet werden.

c) (2 Punkte) Sie haben in der Vorlesung gerlernt, dass die Worst-Case Laufzeit eines allgemeinen Sortierverfahrens durch $\Omega(n \log n)$ beschränkt ist, wobei n die Anzahl der zu sortierenden Datensätze ist. Beschreiben Sie kurz, warum die Worst-Case Laufzeit des Sortierverfahrens *Fachverteilung* trotzdem durch $O(n)$ beschränkt ist.

Aufgabe 3.A: Suchen **(20 Punkte)**

a) (6 Punkte) Fügen Sie folgende Werte in genau dieser Reihenfolge in einen anfangs leeren B-Baum der Ordnung 3 ein. Zeichnen Sie lediglich den resultierenden B-Baum, Zwischenschritte müssen nicht unbedingt angegeben werden.

$$\langle 30, 1, 20, 5, 10, 3, 25, 2, 15 \rangle$$

b) (14 Punkte) Schreiben Sie detaillierten Pseudocode für die Funktion $insert(k)$, die einen Schlüssel k in einen **nicht leeren** *Threaded Binary Tree*[1] einfügt. Falls der Schlüssel bereits im Baum enthalten ist, wird er nicht eingefügt. Auf die Wurzel des Baumes kann mit *root* zugegriffen werden. Zu jedem Element x, das in einem Baum vorhanden ist wird folgendes gespeichert:

- $x.key$ ... Schlüssel des Elementes
- $x.left$ ... Verweis auf linkes Kind des Elementes
- $x.right$ ... Verweis auf rechtes Kind des Elementes
- $x.leftIsThread$ ... ist `true`, falls $x.left$ ein Thread ist, und sonst `false`
- $x.rightIsThread$ ... ist `true`, falls $x.right$ ein Thread ist, und sonst `false`

[1] Definition wie bei der ersten Programmieraufgabe: Ein Threaded Binary Tree ist ein natürlicher binärer Suchbaum mit einer kleinen Erweiterung. In einem gewöhnlichen binären Baum hat ein Knoten, der keinen linken bzw. rechten Unterknoten besitzt, `NULL` als entsprechenden Verweis gespeichert. Diese `NULL`-Verweise werden in einem Threaded Binary Tree durch sogenannte Threads ersetzt. Das sind Verweise, die im Fall eines fehlenden linken Kindes auf den Vorgänger und im Fall eines fehlenden rechten Kindes auf den Nachfolger in der Inorder-Durchmusterungsreihenfolge verweisen.

Natürlich muss der Threaded Binary Tree trotzdem genau zwei `NULL`-Verweise besitzen, denn der Knoten mit dem kleinsten Schlüssel hat keinen Vorgänger, genauso hat der Knoten mit dem größten Schlüssel keinen Nachfolger. Ein Verweis auf `NULL` wird dabei ebenfalls als Thread gekennzeichnet.

Technische Universität Wien
Institut für Computergraphik und Algorithmen
Arbeitsbereich für Algorithmen und Datenstrukturen

186.172 Algorithmen und Datenstrukturen 1 VL 4.0

Nachtragstest SS 2009

30. Juni 2009

Machen Sie die folgenden Angaben bitte in deutlicher Blockschrift:

Nachname: | Vorname:

Matrikelnummer: | Studienkennzahl:

Anzahl abgegebener Zusatzblätter:

Legen Sie bitte Ihren Studentenausweis vor sich auf das Pult.

Sie können die Lösungen entweder direkt auf die Angabeblätter oder auf Zusatzblätter schreiben, die Sie auf Wunsch von der Aufsicht erhalten. Es ist nicht zulässig, eventuell mitgebrachtes eigenes Papier zu verwenden.

Die Verwendung von Taschenrechnern, Mobiltelefonen, Skripten, Büchern, Mitschriften, Ausarbeitungen oder vergleichbaren Hilfsmitteln ist unzulässig.

Die Arbeitszeit beträgt 55 Minuten.

	B1:	B2:	B3:	Summe:
Erreichbare Punkte:	20	16	14	50
Erreichte Punkte:				

Viel Glück!

Aufgabe 1.B: Suchen **(20 Punkte)**

a) (6 Punkte) Fügen Sie folgende Werte in genau dieser Reihenfolge in einen anfangs leeren B-Baum der Ordnung 3 ein. Zeichnen Sie lediglich den resultierenden B-Baum, Zwischenschritte müssen nicht unbedingt angegeben werden.

$$\langle 28, 1, 22, 5, 13, 4, 26, 3, 14 \rangle$$

b) (14 Punkte) Schreiben Sie detaillierten Pseudocode für die Funktion $insert(k)$, die einen Schlüssel k in einen **nicht leeren** *Threaded Binary Tree*[1] einfügt. Falls der Schlüssel bereits im Baum enthalten ist, wird er nicht eingefügt. Auf die Wurzel des Baumes kann mit *root* zugegriffen werden. Zu jedem Element x, das in einem Baum vorhanden ist wird folgendes gespeichert:

- $x.key$ ... Schlüssel des Elementes
- $x.left$ ... Verweis auf linkes Kind des Elementes
- $x.right$ ... Verweis auf rechtes Kind des Elementes
- $x.leftIsThread$ ... ist `true`, falls $x.left$ ein Thread ist, und sonst `false`
- $x.rightIsThread$ ... ist `true`, falls $x.right$ ein Thread ist, und sonst `false`

[1] Definition wie bei der ersten Programmieraufgabe: Ein Threaded Binary Tree ist ein natürlicher binärer Suchbaum mit einer kleinen Erweiterung. In einem gewöhnlichen binären Baum hat ein Knoten, der keinen linken bzw. rechten Unterknoten besitzt, `NULL` als entsprechenden Verweis gespeichert. Diese `NULL`-Verweise werden in einem Threaded Binary Tree durch sogenannte Threads ersetzt. Das sind Verweise, die im Fall eines fehlenden linken Kindes auf den Vorgänger und im Fall eines fehlenden rechten Kindes auf den Nachfolger in der Inorder-Durchmusterungsreihenfolge verweisen.

Natürlich muss der Threaded Binary Tree trotzdem genau zwei `NULL`-Verweise besitzen, denn der Knoten mit dem kleinsten Schlüssel hat keinen Vorgänger, genauso hat der Knoten mit dem größten Schlüssel keinen Nachfolger. Ein Verweis auf `NULL` wird dabei ebenfalls als Thread gekennzeichnet.

Aufgabe 2.B: Optimierung **(16 Punkte)**

a) (2 Punkte) Beschreiben Sie *kurz* die Unterschiede und Gemeinsamkeiten von *Teile und Herrsche* und *Dynamischer Programmierung*.

b) (12 Punkte) Gegeben sei ein Rucksack mit Kapazität 5 sowie die unten angegebenen sieben Gegenstände. Diese Instanz des 0/1-Rucksackproblems soll mittels *Dynamischer Programmierung* über die möglichen Gesamtgewichte gelöst werden. Dazu wird die unten stehende 8×6-Matrix $\mathbf{m}$ verwendet, wobei der Eintrag im Feld $m_{i,j}$ angibt, welcher Wert mit den ersten i Gegenständen erreicht werden kann, wenn das Gesamtgewicht der gewählten Gegenstände kleiner oder gleich j ist.

Die Felder der Matrix können wie folgt berechnet werden:

$$m_{0,j} = 0 \qquad \text{für } j = 0, \ldots, 5$$

$$m_{i,j} = \begin{cases} m_{i-1,j} & \text{falls } w_i > j, \\ \max\{m_{i-1,j-w_i} + c_i,\ m_{i-1,j}\} & \text{sonst.} \end{cases} \qquad \text{für } \begin{cases} i = 1, \ldots, 7 \\ j = 0, \ldots, 5 \end{cases}$$

$i \backslash j$	0	1	2	3	4	5
0	0	0	0	0	0	0
1						
2						
3						
4						
5						
6						
7						

Gegenstand i	Gewicht w_i	Wert c_i
1	2	5
2	3	4
3	6	10
4	2	2
5	1	3
6	3	3
7	2	3

- Lösen Sie die gegebene Instanz des 0/1-Rucksackproblems, indem Sie die obenstehende Matrix vervollständigen.
- Geben Sie eine optimale Lösung der Probleminstanz an, und markieren Sie in der Matrix jene Zellen, die beim Rekonstruieren der Lösung betrachtet werden.

c) (2 Punkte) Kreuzen Sie an, ob die folgenden Aussagen korrekt sind:

Die MST-Algorithmen von Prim und Kruskal liefern immer das selbe Ergebnis ...

... bezogen auf die ausgewählten Kanten ☐ Ja ☐ Nein
... bezogen auf das Gesamtgewicht ☐ Ja ☐ Nein

Aufgabe 3.B: Sortieren **(14 Punkte)**

a) (8 Punkte) Sortieren Sie die folgenden Zahlen **aufsteigend** mittels **Quick-Sort**:

$$\langle 5, 2, 3, 4, 8, 9, 7, 6 \rangle.$$

Geben Sie die Zahlenfolge nach **jedem Aufruf** von `Partition()` an. Markieren Sie in jeder dieser Folgen die Zahlen, die sich bereits an ihrer endgültigen Position befinden.

5	2	3	4	8	9	7	6

Hinweis: Nicht jede Zeile der Tabelle muss zwangsläufig
für eine korrekte Lösung verwendet werden.

b) (2 Punkte) Sie haben in der Vorlesung gerlernt, dass die Worst-Case Laufzeit eines allgemeinen Sortierverfahrens durch $\Omega(n \log n)$ beschränkt ist, wobei n die Anzahl der zu sortierenden Datensätze ist. Beschreiben Sie kurz, warum die Worst-Case Laufzeit des Sortierverfahrens *Fachverteilung* trotzdem durch $O(n)$ beschränkt ist.

c) (4 Punkte) Eigentlich sollte die folgende Abbildung einen **Maximum-Heap** (wie aus der LVA bekannt als Datenstruktur für Heap-Sort) als Baum darstellen, allerdings haben sich einige Fehler eingeschlichen. Markieren Sie die Fehler und beschreiben Sie diese *kurz*.