Datenstrukturen und Algorithmen
Übung 5, Frühling 2011

24. März 2011

Diese Übung muss zu Beginn der Übungsstunde bis spätestens um 16 Uhr 15 am 31. März abgegeben werden. Die Abgabe der DA Übungen erfolgt immer in schriftlicher Form auf Papier. Programme müssen zusammen mit der von ihnen erzeugten Ausgabe abgegeben werden. Drucken Sie wenn möglich platzsparend 2 Seiten auf eine A4-Seite aus. Falls Sie mehrere Blätter abgeben heften Sie diese bitte zusammen (Büroklammer, Bostitch, Mäppchen). Alle Teilnehmenden dieser Vorlesung müssen Ihre eigene Übung abgeben. Vergessen Sie nicht, Ihren Namen und Ihre Matrikelnummer auf Ihrer Abgabe zu vermerken.

Theoretische Aufgaben

1. Geben Sie eine in Zeit $\theta(n)$ laufende, nichtrekursive Prozedur an, welche die Reihenfolge einer einfach verketteten Liste aus n Elementen umkehrt. Die Prozedur sollte nur konstant viel Speicherplatz benutzen, abgesehen von dem, der für die Liste selbst gebraucht wird.

2. Schreiben Sie Pseudocode, um eine Wartschlaufe mit einer einfach verketteten Liste zu Implementieren. Nehmen Sie an, die Listenelemente hätten ein Feld *next* mit dem Zeiger auf das nächste Element, und ein Feld *key* mit dem Schlüssel. Die Operationen *ENQUEUE* und *DEQUEUE* sollten noch immer in Zeit $O(1)$ arbeiten.

   Illustrieren Sie den Ablauf der folgenden Operationen, indem Sie für jeden Schritt die Liste darstellen und gegebenenfalls den Rückgabewert angeben: *ENQUEUE(3); ENQUEUE(5); DEQUEUE(); ENQUEUE(2); DEQUEUE(); ENQUEUE(8); ENQUEUE(9); DEQUEUE(); DEQUEUE(); DEQUEUE(); DEQUEUE();* **2 Punkte**

3. Schreiben Sie Pseudocode für eine rekursive Prozedur, die alle Knoten eines gerichteten Baumes mit unbeschränktem Grad besucht und jeweils den Schlüssel des Knotens ausgibt. Nehmen Sie an, die Knoten des Baumes hätten die Felder *left-child* und *right-sibling* für die Zeiger auf das sich am weitesten links befindende Kind und den rechten Bruder und das Feld *key* für den Schlüssel. (**1 Punkt**)

4. Schreiben Sie Pseudocode für eine *nicht-rekursive Prozedur*, die alle Knoten eines gerichteten Baumes mit unbeschränktem Grad besucht und jeweils den Schlüssel des Knotens ausgibt. Verwenden Sie dazu einen Stack. Nehmen Sie an, der Stack unterstützt die Operationen *push(node)* und *pop*, wobei *node* ein Knoten des Baumes ist. Der Rückgabewert von *pop* ist ein Knoten *node* oder NIL wenn der Stack leer ist. **1 Punkt**)

Praktische Aufgaben

In dieser Aufgabe werden sie einen KD-Tree implementieren. Wir stellen auf der Vorlesungs Webpage Code zur Verfügung, auf dem Sie aufbauen können. Der Code enthält eine Klasse *KDTreeTester* die das Programm startet und ein Fenster mit zufällig generierten Punkten anzeigt. Die Variabeln w, h und n steuern die Grösse des Fensters und die Anzahl Punkte.

Die Klasse *KDTreeVisualization* enthält verschiedene Funktionen zum Generieren und Anzeigen der zufälligen Punkte. Die Punkte werden in einer verketteten Liste gespeichert. Die Klasse enthält auch eine Funktion um die Reihenfolge der Punkte in der verketteten Liste zu visualisieren.

Eine detaillierte Beschreibung von kd-Bäumen mit Pseudocode für die nächste Nachbar Suche finden Sie auf Wikipedia: `http://en.wikipedia.org/wiki/Kd_tree`.

1. Schreiben Sie eine Funktion, die für einen gegebenen Punkt seinen nächsten Nachbarn in der Liste sucht. Implementieren Sie dazu die Funktion *listSearchNN* in *KDTreeVisualization*. Sie können Ihre Funktion testen, indem Sie im Menu *Search "Search List for NN"* wählen. Die aufgerufene Funktion sucht den nächsten Nachbarn für x Punkte und misst dabei die benötigte Zeit. (**1 Punkt**)

2. Implementieren Sie die Funktion *createKDTree*, die die Punkte aus der Liste in einem kd-Baum speichert. Brauchen Sie dazu die innere Klasse *TreeNode*. Ein Objekt dieser Klasse repräsentiert, wie der Name bereits sagt, einen Knoten im kd-Baum. Die Variable *kdRoot* soll die Wurzel Ihres Baumes enthalten.

   Um die Punktelisten zu sortieren, können Sie die Klasse *PointComperator* und die Java Funktion *Collections.sort(List list, Comparator c)* verwenden.

   Sie können den kd-Baum mit *Visualize-KD Tree* anzeigen lassen. (**2 Punkte**)

3. Implementieren Sie nun eine Funktion, die die Suche nach dem nächsten Nachbarn auf dem kd-Baum durchführt. Erweitern Sie dazu die Funktion *treeSearchNN*. Vergleichen Sie dann die Laufzeit der Suche auf dem kd-Baum mit der Suche auf der Liste. Führen Sie dazu eine Suche nach dem nächsten Nachbarn für verschiedene Mengen zu suchender Punkte und mehrere unterschiedliche Punktemengen von unterschiedlicher Grösse durch. Stellen Sie die Resultate in einer Liste und grafisch dar. Stimmen Ihre Messungen mit der theoretisch erwarteten Zeitkomplexität überein? (Suche in der Liste: $O(n)$, Suche im Baum: $O(\lg n)$.) (**2 Punkte**)