Institut für Informatik
Ludwig-Maximilians-Universität München

SS 2008
21. April 2008
Übungsblatt 2

Prof. Dr. Hans-Peter Kriegel
Thomas Bernecker, Tobias Emrich

Übungen zur Vorlesung

Effiziente Algorithmen

Aufgabe 2.1: *Doppelt verkettete Listen*

In der Vorlesung wurden lineare Listen als einfachste dynamische Datenstruktur eingeführt. Sie unterstützen ein effizientes Einfügen und Löschen von Elementen am Anfang der Liste. Will man aber das letzte Element der Liste manipulieren, muss zunächst die gesamte Liste durchlaufen werden (Laufzeitkomplexität *O(N)*). **Doppelt verkettete Listen** sind eine Erweiterung linearer Listen, die den Aufwand der Manipulation an beiden Enden konstant hält.

a) Modellieren Sie eine doppelt verkettete Liste in der Zeiger-und-Kästchen-Darstellung (siehe lineare Listen im Skript)! Die Liste soll dabei mindestens 3 Elemente enthalten.

b) Vervollständigen Sie das untenstehende Gerüst eines Java-Programms, so dass die ersten 6 angegebenen Methoden in konstanter Zeitkomplexität ausführbar sind!

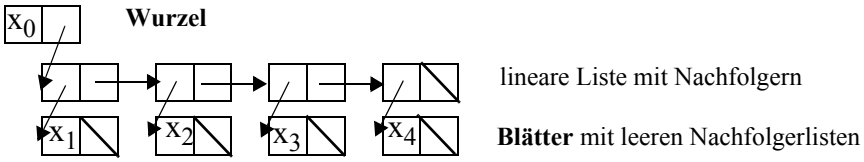
```
public class DoppeltVerketteteListe
{
  ...
  // zu implementierende Methoden
  void insertLast(int value) {...} // Einfügen am Ende
  void insertFirst(int value) {...} // Einfügen am Anfang
  void deleteLast() {...} // Lösche letztes Element
  void deleteFirst() {...} // Lösche erstes Element
  int getFirst() {...} // Ausgabe erstes Element
  int getLast() {...} // Ausgabe leztes Element


 // Beispielaufruf der Klasse DoppeltVerketteteListe
  public static void main(String args[])
  {
    DoppeltVerketteteListe dvk = new DoppeltVerketteteListe();
    dvk.insertFirst(1);
    dvk.insertLast(2);
    dvk.insertFirst(3);
    dvk.deleteLast();
    System.out.println( dvk.getFirst() + " | " + dvk.getLast());
  }
}
```

Aufgabe 2.2: *Verschiedene Darstellung von Bäumen m-ten Grades*

Die gebräuchlichste Darstellung von Bäumen *m*-ten Grades ist in jedem Knoten einen Pointer für jeden der *m* möglichen Teilbäume vorzusehen. Dabei werden jeweils nur *deg(k)* (Grad des Knotens) viele Pointer tatsächlich verwendet. Der Rest besteht aus Null-Pointern.

Eine andere Methode ist in jedem Knoten eine lineare Liste von Pointern auf Teilbäume zu führen. Ist der Knoten ein Blatt, wird dabei nur ein Null-Pointer als Zeichen dafür benötigt, daß der Knoten keine Nachfolger hat. Allerdings ist die Darstellung eines Knotens *k*, dessen Grad *deg(k)* nahe an *m* liegt, wiederum schlechter, da die lineare Liste zusätzliche Pointer auf die folgenden Elemente benötigt.



Graphik 1: Pointer- und Kästchen-Darstellung einer Baumes mit linearer Liste zur Verwaltung der Nachfolger.

Definition: Ein Pointer *p* sei ein *Overhead-Pointer,* falls *p* nicht auf einen Teilbaum zeigt. Ein Null-Pointer ist also auch ein *Overhead-Pointer.*

a) Wieviele *Overhead-Pointer* besitzt ein Knoten bei der Darstellung mit linearer Liste? Geben Sie Ihr Ergebnis in Abhängigkeit von *deg(k)* an und begründen Sie es kurz!

b) Sei K die Menge aller Knoten in einem Baum mit *n* Knoten. Wie hoch ist folgende Summe und warum: $\sum_{k \in K} deg(k)$?

c) Berechnen Sie die Speicherplatzverschwendung der Darstellung mit linearen Listen (*Overhead-Pointer/ alle Pointer im Baum*)! Verwenden Sie dazu die Ergebnisse aus den ersten beiden Teilaufgaben.

Aufgabe 2.3: *Eigenschaften allgemeiner Bäume*

Wir betrachten einen Baum vom Grad $d >= 2$.

a) Wie groß ist die maximale Anzahl von Knoten auf Level *i* eines Baumes vom Grad *d*?

b) Wie groß ist die maximale Anzahl von Knoten in einem Baum der Höhe *h* vom Grad *d*?

c) Wie groß ist die maximale Höhe eines Baumes vom Grad *d* mit *n* Knoten?

d) Wie groß ist die minimale Höhe eines Baumes vom Grad *d* mit *n* Knoten?

Beweisen Sie Ihre Aussagen!

Aufgabe 2.4: *binäre Suchbäume*

Gegeben sei ein binärer Suchbaum, dessen Knotenwerte natürliche Zahlen seien. Bei Eingabe eines Intervalls $I = [lower, upper]$ $(lower, upper \in N)$ sollen alle Knotenwerte *w* in aufsteigender Reihenfolge ausgegeben werden, für die gilt: *lower* <= *w* <= *upper*.

a) Geben Sie einen möglichst effizienten Algorithmus (in Java-Notation) für dieses Problem an! Sie können dabei die Klassen und Methodendefinitionen aus dem Skript benutzen. Weiterhin soll der Kopf Ihrer Methode folgenden Aufbau haben:

public void FindValuesInInterval(int lower, int upper, BinaryNode t)

b) Bestimmen Sie die Komplexität Ihres Algorithmus, falls der Suchbaum balanciert ist ($h_{max}=O(\log n)$)!