Programmierstarthilfe SS 2008

Fakultät für Ingenieurwissenschaften und Informatik

12. Blatt

Für den 14.7. und 15.7.2008

Organisatorisches

Um auf die Mailingliste aufgenommen zu werden schicke einfach eine Mail an `guido.de-melo@uni-ulm.de`.

Die Webseiten zur Veranstaltung sind unter `http://www.uni-ulm.de/in/mi/lehre/ss2008/programmierstarthilfe.html` zu finden.

Wir wünschen euch viel Erfolg bei euren Prüfungen!

Fragestunde

In der PI-Vorlesung am Donnerstag wird es eine Fragestunde geben. Wir werden uns zur Verfügung stellen, um Fragen zur Vorlesung, zu den Tutorien und zum Programmieren zu beantworten. Überlegt euch also bis dahin, was euch noch unklar ist, damit ihr diese Chance entsprechend nutzen könnt.

1. Bäume

Lineare Listen als Datenstruktur kennen wir bereits. In einer Liste hat jedes Element einen Inhalt und eine Referenz auf genau einen Nachfolger. Bäume funktionieren im Wesentlichen nicht anders, nur dass hier mehrere Nachfolger existieren. In den untersten Elementen, den Blättern ist der Nachfolger die `null`-Referenz Im Beispiel gehen wir von binären Bäumen aus, also solchen Bäumen, in denen jedes Element zwei Nachfolger haben kann. Es gibt aber durchaus auch Bäume mit einer größeren Anzahl an Nachfolgern pro Element. Hier ein Beispiel für einen einfachen binären Baum mit Integer-Werten als Inhalt:

Einen Baum wie diesen nennt man auch Suchbaum, das heißt, dass von jedem Knoten aus, der linke Nachfolger einen kleineren oder gleichen Wert hat, und der rechte Nachfolger einen größeren.

In Binärbäumen lässt sich besonders gut rekursiv arbeiten. Dazu baut man Methoden, die die Wurzel eines (Teil-)Baums übergeben bekommen und sich dann wieder selbst aufrufen. Zum Beispiel so:

```
1 public static int tuWas(Node wurzel) {
2         if (wurzel==null)
3                 return 0;
4         else
5                 return wurzel.wert
6                         +tuWas(wurzel.linkerNachfolger)
7                         +tuWas(wurzel.rechterNachfolger);
8         }
```

Kommst du darauf, was die Methode `tuWas` berechnet?

2. Aufgaben

2.1. Binäre Suchbäume

Schreibe eine Klasse `BinBaumKnoten`, mit der sich Knoten eines binären Suchbaums erstellen lassen. Jeder Knoten enthält einen int-Wert und Referenzen auf einen linken und einen rechten Nachfolgerknoten vom gleichen Typ. Du kannst dich dabei an der Liste vom letzten Mal orientieren.

Schreibe außerdem eine Klasse `BinBaum`, die eine Referenz auf die Wurzel des Binärbaums enthält.

- Schreibe einen Konstruktor für die Klasse BinBaumKnoten, der einen int-Wert übergeben bekommt und einen Knoten mit entsprechendem Wert anlegt.
- Schreibe einen Konstruktor für die Klasse BinBaum, der einen int-Wert übergeben bekommt und einen neuen Baum erzeugt, dessen Wurzel ein Knoten mit dem übergebenen Wert erstellt.
- Schreibe eine einfache Einfügemethode für die Klasse BinBaum. Sie bekommt einen int-Wert und die Wurzel des Baums übergeben. Ist der Wert kleiner oder gleich als der der Wurzel ruft sie sich rekursiv auf, allerdings mit dem linken Nachfolgerknoten, ansonsten mit dem rechten. Ist der entsprechende Nachfolger `null` wird an dieser Stelle ein neuer Knoten mit dem entsprechenden Wert eingefügt.
- Schreibe in der Klasse BinBaum eine Methode `boolean contains(BinBaumKnoten wurzel,int wert)`, die überprüft ob ein Integer-Wert im Baum vorhanden ist. Bei der Suche kannst du dich an der vorherigen Teilaufgabe orientieren.

- Schreibe eine zweite Methode (z.B. `boolean contains2(BinBaumKnoten wurzel,int wert)`), die auch prüft, ob ein Wert vorhanden ist. Allerdings soll diese Methode auch auf nicht-Suchbäumen funktionieren, auf denen Knoten beliebig angeordnet sind.

3. Bonusaufgaben

3.1. Fibonacci-Zahlen

Eine typische Standartaufgabe für Rekursion sind die Fibonacci-Zahlen, eine Reihe von Zahlen f_i. Diese sind rekursiv definiert: $f_0 = 0$, $f_1 = 1$, $f_{n+1} = f_n + f_{n-1}$

a) Schreibe ein rekursive Methode für die Berechnung des n-ten Reihenelements der Fibonacci-Reihe, das die obige Definition nutzt.

b) Da jedes Element die Summe seiner beiden Vorgänger ist, kann man die Berechnung auch recht gut itterativ vornehmen. Dafür beginnt man bei f_0 und f_1, addiert diese zu f_2 und merkt sich f_1 und f_2 für den nächsten Schritt, dann addiert man diese usw. Schreibe diese itterative Methode.

c) Mit `System.currentTimeMillis()` erhälts du die aktuelle Systemzeit in Millisekunden. Miss damit die jeweilige Laufzeit beiden Methode für etwas größere n, z.B. $n = 50$.

d) Wie kommt es zu den Laufzeitunterschieden? Mach dir z.B. klar, wie die rekursive Variante arbeitet, indem du einen Aufrufbaum zeichnest.

3.2. Sieb des Eratosthenes

Ein berühmtes und sehr altes Verfahren zur Primzahlbestimmung ist das “Sieb des Eratosthenes” (siehe `http://de.wikipedia.org/wiki/Sieb_des_Eratosthenes`).

Schreibe eine Methode, das die Überprüfung vornimmt, ob eine Zahl p eine Primzahl ist. Die Methode soll nach dem Prinzip des “Sieb des Eratosthenes” vorgehen. Zu Beginn deines Programmes, solltest du also ein Array von Zahlen anlegen ($p \leq 10000$) und darauf das Sieb-Verfahren anwenden.

3.3. Worte sortieren

Schreibe ein Programm, das eine Liste von Strings vom Benutzer einliest und diese alphabetisch sortiert ausgibt. Dazu könnten dir folgende Überlegungen helfen:

a) Der Benutzer gibt einige Worte ein. Diese können zum Beispiel zeilenweise angenommen werden, oder anhand von Leerzeichen geteilt (schau dir dafür in der API die String-Methoden an).

b) Die Eingaben sollten gespeichert werden. Da könntest du ein Array nehmen, wenn eine feste Menge von Eingaben erfolgt. Oder du nimmst eine Liste, was aber erfordert, dass die Liste Strings speichern kann (für Experten: Die LinkedList-Klasse in der API anschauen). Denk bei deiner Wahl auch daran, dass du mit dieser Wahl später sortieren können sollst.

c) Strings lassen sich nicht mit < oder >, und oft nicht einmal mit == vergleichen. Schreibe eine Methode, die zwei Strings alphabetisch vergleicht, also zurückgibt, ob der erste oder der zweite String im Lexikon zuerst kommt.

d) Nimm dir nun eine Sortiermethode, die auf dem von dir gewählten Datentyp (Liste oder Array) funktioniert, und schreibe sie so um, dass sie mit Strings umgehen kann sortieren kann (rufe dazu deine Vergleichmethode auf).

3.4. Doppelt-verkettete Liste

Implementiere, ausgehend von deiner Listen-Klasse, die gleichen Methoden für eine doppelt verkettete Liste, d.i. eine Liste, in der jedes Element nicht nur eine Referenz auf seinen Nachfolger, sondern auch auf seinen Vorgänger besitzt. Die Liste führt dann nicht nur den Kopf, sondern auch den Fuss mit.

Viele Methoden ändern sich nicht groß dabei, manchmal werden sie sogar einfacher, weil du den Vorgänger jetzt direkt benutzen kannst und nicht extra mitspeichern musst. Achte aber darauf, dass die Vorgänger immer richtig gesetzt werden, gerade beim Löschen und Einfügen.

Schreibe auch eine Suchmethode, die von hinten sucht.

3.5. Rekursion nachvollziehen

Betrachte folgende rekursive Methode:

```
1 int ackermann(int n, int m){
2         if(n == 0) return m + 1;
3         else if(m == 0) return ackermann(n - 1, 1);
4         else return ackermann(n -1, ackermann(n, m-1));
5 }
```

Was gibt sie für den Aufruf ackermann(2, 2) zurück? Vollziehe zur Beantwortung dieser Frage die Aufrufstruktur auf dem Papier nach. Keinen Computer verwenden!

3.6. Mergesort

Im folgenden soll Mergesort stückweise nachgebaut werden. Lege hierfür eine Klasse `mergesort-tester` mit `main`-Methode an.

a) Erstellen zwei Listen, und befülle sie folgendermaßen:

   Liste 1: 2 3 4 5 6
   Liste 2: 1 5 7 8 9

   Wichtig ist, dass die Listen an sich sortiert sind.

b) Schreibe eine Methode `Liste merge(Liste a, Liste b)`, die zwei sortierte Listen `a` und `b` sortiert in eine Liste `c` mischt, und diese zurückgibt. Teste die Methode mit dem beiden Listen aus a).

c) Schreibe eine Methode `Liste mergesort(Liste x)`, die eine unsortierte Liste `x` sortiert, indem sie die Liste in der Mitte teilt und dann beide Teile rekursiv mit `mergesort(teilliste)` sortieren lässt. Die sortierten Teillisten sollen dann mit `merge(teilA, teilB)` zusammen gemischt werden. Teste deine Methode mit folgender Liste: 8 3 0 1 -3 7 9 14 5 3.

d) Skizziere einen Aufrufbaum für die Liste aus c). Als Knoten schreibst du dabei die Parameter auf, z.B. merge([8, 3, 0, 1, -3], [7, 9, 14, 5, 3]). Wenn der Baum komplett gezeichnet ist, dann schreibe an die Kanten die zurückgegebenen Listen.

e)* Ändere die Methoden so ab, dass sie mit Arrays, anstatt mit Listen funktionieren.

f)** Verbesserungen: Durch die Rekursion werden immer wieder kleine Fälle aufgerufen, die man leichter mit einen einfachen Insertion-Sort sortiert. Wenn man dies tut kann man die Laufzeit um etwa 10-15% verbessern. Außerdem kann man versuchen die Zeit zum Kopieren in das Hilfsarray zu eliminieren, indem man in jeder Ebene es Baums das Hilfs- und das Eingabearray vertauscht.

g)*** Verbesserungen 2: Diese Prozedur könnte man nun wiederum optimieren, indem man die Tests auf das Ende der Eingabearrays in mergeAB() eliminiert. Hierfür müsste man je eine Methode für das Mischen und das Sortieren schreiben, die die Arrays in aufsteigender und absteigender Reihenfolge füllen.

Mit der letzten Optimierung wird der Mergesort um ca. 40% schneller, aber immer noch 25% langsamer als Quicksort.

3.7. Quicksort

Einer der schnellsten Algorithmen zum sortieren ist der Quicksort, der auch sehr gut unter `http://de.wikipedia.org/wiki/Quicksort` beschrieben wird.

a) Lege eine Klasse `quicksort-tester` mit `main`-Methode an. Erstelle ein `int`-Array und befülle es folgendermaßen: 8 3 0 1 -3 7 9 14 5 3.

b) Schreibe eine Methode `int partition(int[] a, int left, int right)`, die ein Array `a` und zwei Positionsbegrenzer `left` und `right` übergeben bekommt. Zuerst suchen wir uns eine Zahl aus `a` (am besten `a[right]`) aus und nennen diese Zahl„Pivot-Element. Dann teilen wir die Zahlen zwischen `left` und `right-1` jeweils in „größer Pivot und „kleiner Pivot indem wir uns zwei Variablen `fromLeft` und `fromRight` vom Typ `int` zu Hilfe nehmen.

   Verwende den Algorithmus wie er bei `http://de.wikipedia.org/wiki/Quicksort` beschrieben ist und bennene die Methoden wie oben beschrieben.

c) Schreibe eine Methode `int[] quicksort(int[] a, int l, int r)`, die eine unsortiertes Array `a` sortiert, indem sie mit `partition(a, l, r)` die Position des ersten Pivot-Elements bekommt und dann die Teile links und rechts davon jeweils rekursiv mit `quicksort(a, ...)` sortieren lässt. Teste die Methode mit dem Array aus a).

d) Skizziere die Aufrufe für die Liste aus c) auf einem Blatt Papier. Als Knoten schreibst du dabei die Parameter auf, der Übersicht halber aber nur den Teil des Arrays, der dann auch verwendet wird z.B. quicksort([-3, 1, 0, 3]). Wenn der Baum komplett gezeichnet ist, dann schreibe an die Kanten die zurückgegebenen Arrayteile.