Programmierstarthilfe WS 2008/09
Fakultät für Ingenieurwissenschaften und Informatik

11. Blatt
Für die Woche vom 12.1. bis zum 16.1.2009 (KW 3)

Organisatorisches

Die Webseiten zur Veranstaltung sind unter `http://www.uni-ulm.de/in/mi/lehre/2008ws/programmierstarthilfe.html` zu finden.

Diese Woche arbeiten wir mit Listen. Die erste Aufgabe baut die Listenklasse aus dem Skript aus. Du kannst dir die Klasse aus dem Rubikon oder von der Webseite holen.

1 Aufgaben

1.1 Vorgefertige Klassen für Listenelemente und Listen

Nachdem du die vorgefertigte Klasse für Listen und Listen-Elemente aus dem Rubikon heruntergeladen oder aus dem Skript kopiert hast, sollst du dich nun mit dem Arbeiten mit Listen vertraut machen und zwei fehlende Methoden aus der Listenklasse implementieren.

a) Erstelle testweise zwei Objekte vom Typ `ListenElement` und verkettete sie, sodass das zwei Listenelement Nachfolger vom ersten ist. Hierfür soll noch nicht die Klasse `Liste` verwenden werden.

b) Betrachte nun die Klasse `Liste`. Die Methode `laenge()` soll die Anzahl der vorhandenen Elemente in der Liste zurückgeben. Implementiere sie!

c) Desweiteren fehlt der Listenklasse die Implementierung der Methode `fuegeHinzu(ListenElement element)`, die an das letzte Element in der Liste das übergebene neue Element anhängt.

d) Schreibe eine überladene Variante der Methode `fuegeHinzu()`, die kein Listenelement sondern den Inhalt, hier also den Zahlenwert, des hinzuzufügenden Elements übergeben bekommt. Die Methode soll dann das neue Element selbstständig erzeugen und in die Liste einhängen. Implementiere diese Methode `fuegeHinzu(int wert)`.

e) Beziehe nun die Klasse `Liste` mit in den Test aus dem ersten Aufgabenteil ein und überprüfe die Methoden auf ihre Funktionalität.

f) Implementiere eine Methode `gebeAus()` für die Klasse `Liste`. Sie soll den Inhalt der kompletten Liste auf die Konsole ausgeben.

g)* Entwickle außerdem eine Methode `gebeVerkehrtAus()`, die den Listeninhalt in umgekehrter Reihenfolge ausgibt.

1.2 Zahlen sortieren

Schreibe ein Programm, dass eine Liste von Zahlen einliest und diese aufsteigend sortiert ausgibt. Dein Programm soll die neuen Zahlen in eine schon sortierte Liste einfügen (wie beim Insertion Sort). Wenn der Benutzer keine Zahl mehr eingibt soll die fertige Liste ausgegeben werden.

1.3 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).

2 Bonusaufgaben

Durch die Aufgaben oben hast du ein Verständnis für das neue Konzept dieses Blatts bekommen. Durch Bonusaufgaben kannst du nun deine Kenntnisse vertiefen.

2.1 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 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.

2.2 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 den beiden Listen aus a).

c) Schreibe eine Methode `Liste mergesort(Liste x)`, die eine unsortierte Liste `x` sortiert, in dem 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 Für das nächste Blatt

Nächste Woche werden wir Bäume behandeln. Lies dazu im Kapitel *Einfache abstrakte Datentypen* den Teil über Bäume.