Übungen zur Vorlesung
Praktische Informatik I
Wintersemester 2000/01

Serie X

Aufgaben zur freiwilligen Wiederholung und Vertiefung

Aufgabe 1: Größter gemeinsamer Teiler – rekursiv und iterativ (-)

Um den größten gemeinsamen Teiler (ggT) zweier natürlicher Zahlen $a, b > 0$ mit $a \geq b$ zu berechnen hat Euklid etwa im Jahre 300 vor Christus den folgenden rekursiven Algorithmus entwickelt:

> Wenn b den Wert 0 besitzt, so ist a der gesuchte ggT der beiden ursprünglichen Zahlen. Ansonsten ergibt sich der ggT durch Anwendung dieses Algorithmus auf die Zahlen b und a `mod` b

Bei dem rekursiven Aufruf ist auf die richtige Reihenfolge der Parameter zu achten, es gilt $b \geq (a \texttt{ mod } b)$.
Schreiben Sie zwei Java–Methoden, welche den ggT zweier Zahlen berechnen, wobei eine direkt den rekursiven Originalalgorithmus umsetzen soll und eine das Problem mit Hilfe einer Schleife ohne rekurisve Aufrufe lösen soll. Schreiben Sie zusätzlich ein kleines Rahmenprogramm, welches in einer Schleife jeweils zwei Zahlen vom Benutzer einlesen soll und zu diesen den ggT mit beiden Methoden berechnet und die beiden Ergebnisse auf dem Bildschirm ausgibt.

Aufgabe 2: Primzahlen mit dem Sieb des Eratosthenes (-)

Das Sieb des Eratosthenes beschreibt ein einfaches Verfahren, um alle Primzahlen bis zu einem bestimmten Wert n zu ermitteln. Es läßt sich folgendermaßen beschreiben:

1. Schreibe die Folge der Zahlen von 2 bis n auf und setze einen Marker auf die Zahl 2.

2. Lösche alle Vielfachen der momentan markierten Zahl aus der Folge.

3. Falls in der Folge noch Zahlen vorkommen, die größer sind als die momentan markierte, so setze den Marker um eine Zahl weiter (also auf die nächsthöhere vorhandene Zahl), ansonsten ist der Algorithmus beendet.

4. Fahre fort mit Schritt 2.

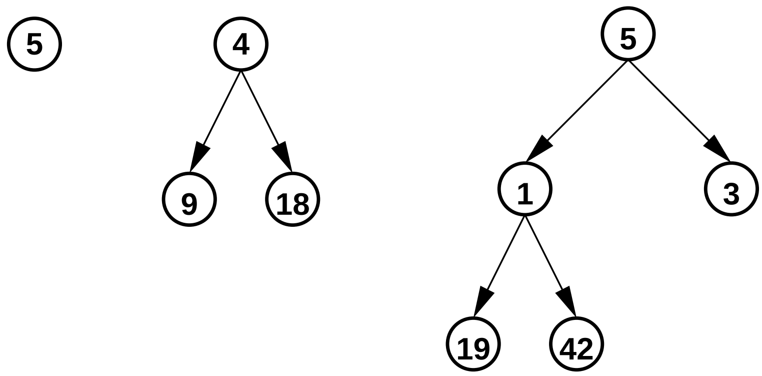
Die in der Folge verbleibenden Zahlen sind alle prim.
Setzen Sie dieses Verfahren in einem Java-Programm um, welches die Primzahlen von 2 bis 1000 auf dem Bildschirm ausgeben soll. Dabei eignet sich beispielsweise ein Boolean–Array `prim` um für jede Zahl zu markieren, ob sie noch in der Liste vorhanden ist oder nicht. Für eine Variable `p` ist dann nach Beenden des Algorithmus `prim[p]` genau dann `true`, wenn der Wert von `p` eine Primzahl ist.

Aufgabe 3: Binäre Bäume (-)

Bäume sind ein in der Informatik vielfach verwendeter Datentyp zur strukturierten (oder sortierten) Speicherung von Daten. Sie stellen eine Verallgemeinerung des Listenkonzepts dar. Die hier betrachtete Variante von binären Bäumen läßt sich folgendermaßen charakterisieren:
Ein binärer Baum ist entweder

- ein leerer Baum oder
- ein Knoten, welcher einen linken und einen rechten Nachfolger besitzt, die wiederum jeweils binäre Bäume sind, sowie ein Datum.

Drei Beispiele für Bäume (dabei stellt das erste Beispiel ein einzelnes *Blatt* dar, wie ein Knoten ohne Nachfolger auch genannt wird):



Der Unterschied zu Listen besteht also darin, daß nicht eine lineare Kette von Elementen angelegt wird, sondern daß für jeden Knoten *zwei* Nachfolger angegeben werden müssen. Für eine Umsetzung des Datentyps Baum in Form einer Java–Klasse bedeutet dies, daß sie ein ähnliches Aussehen wie die `Link`–Klasse erhält, jedoch neben dem Nutzdatum einen *linken* und einen *rechten* Nachfolger besitzt.
Schreiben Sie, neben einer Klassendefinition für Baumelemente mit einem Integer–Datum, Java–Methoden zur Umsetzung der grundlegenden Operationen auf diesen binären Bäumen:

emptyTree: $\rightarrow$ Tree [Konstruktor]
liefert einen leeren Baum zurück

mkTree: Tree $\times$ int $\times$ Tree $\rightarrow$ Tree [Konstruktor]
erzeugt einen neuen Baum, welcher als linken Nachfolger den ersten Parameter, als rechten Nachfolger den dritten Parameter und als Wert des aktuellen Knotens den zweiten Parameter erhält und liefert den so erzeugten Baum zurück.

selectLeft: Tree $\rightarrow$ Tree [Selektor]
liefert den linken Teilbaum des übergebenen Baumes zurück.

selectRight: Tree → Tree [Selektor]
: liefert den rechten Teilbaum des übergebenen Baumes zurück.

selectData: Tree → int [Selektor]
: liefert den Wert des obersten Knotens des übergebenen Baumes zurück.

isEmptyTree: Tree → boolean [Recognizer]
: liefert einen Wahrheitswert zurück, der angibt, ob es sich bei dem übergebenen Baum um einen leeren Baum handelt.

Implementieren Sie darüberhinaus, aufsetzend auf diesen Basisoperationen, die beiden folgenden komplexeren Funktionen:

sum: Tree → int
: liefert die Summe der Werte *aller* Knoten des übergebenen Baumes zurück.

existsIn: Tree × int → boolean
: liefert genau dann `true`, wenn der übergebene Wert an beliebiger Stelle als Wert in dem übergebenen Baum vorkommt.

Aufgabe 4: Sortieren durch Einsortieren (-)

In vielen Anwendungen werden Daten intern immer sortiert gespeichert, um schnell darauf zugreifen zu können. Werden zu einem solchen Satz nun neue Daten hinzugefügt, so ist es sinnvoll, die bestehende Sortierung auszunutzen, indem das neue Datum an der richtigen Stelle eingefügt wird, so daß sofort wieder ein richtig sortierter Gesamtdatensatz entsteht.
Schreiben Sie eine Methode in Java, welche als Parameter eine sortierte Liste von Integer–Zahlen sowie einen neu einzusortierenden Wert bekommt und als Ergebnis eine um das neue Element erweiterte und sortierte Liste zurückliefert. Diese Methode soll die übergebene Liste durchlaufen und das neue Element an der richtigen Stelle einfügen.
Eine unsortierte Liste läßt sich dann sortieren, indem schrittweise eine neue Liste sortiert aufgebaut wird. Dabei können die Werte der Originalliste einfach von links nach rechts abgearbeitet und in eine anfangs leere Liste sortiert eingefügt werden. Schreiben Sie auch hierfür eine enstprechende Java–Methode, die dieses Verfahren unter Zuhilfenahme der oben entwickelten Funktion umsetzt.
Schreiben Sie die beiden Methoden zusätzlich auch in einer Variante, die nicht auf Listen, sondern auf Arrays basiert.

Aufgabe 5: Die Grammatik außerirdischer Floskeln (-)

Auf der WWW-Seite

`www.tzi.de/~alien/lehre_ws2000-2001/notizen/lesestunde8.html`

befindet sich die folgende, hier nur in EBNF wiedergegebene Grammatik zur Bildung außerirdischer Floskeln:

```
<Floskel> ::= <Prolog> <Hauptteil> <Hauptteil> { <Hauptteil> } <Epilog>
<Prolog> ::= YEPRO { <Wort> }
<Wort> ::= YEMOH | LLLX | HAJU | SIMBLFEX
<Hauptteil> ::= YEHAU <HauptteilRest>
<HauptteilRest> ::= { <OhneSIMBLFEX> } | { <OhneHAJU> }
<OhneSIMBLFEX> ::= YEMOH | LLLX | HAJU
<OhneHAJU> ::= YEMOH | LLLX | SIMBLFEX
<Epilog> ::= YEEPI { <Wort> | <Floskel> }
```

Schreiben Sie eine entsprechende Grammatikdefinition mit Hilfe der Technik der Syntaxdiagramme auf. Prüfen Sie anschliessend, ob die folgenden Sätze erlaubte Floskeln dieser Sprache darstellen oder ob und wo sie von den gegebenen Regeln abweichen. Verwenden Sie hierfür sowohl die Original-EBNF Beschreibung als auch Ihre Syntaxdiagrammdarstellung, um die Äquivalenz der beiden Grammatiken zu überprüfen:

- YEPRO YEMOH LLLX YEHAU YEMOH LLLX YEHAU YEMOH SIMBLFEX YEEPI LLLX SIMBLFEX HAJU
- YEPRO YEHAU YEHAU YEEPI
- YEPRO HAJU YEHAU YEHAU SIMBLFEX YEEPI YEPRO YEHAU HAJU YEEPI
- YEPRO YEHAU YEHAU YEEPI YEMOH YEEPI
- YEPRO YEHAU YEHAU LLLX HAJU LLLX SIMBLFEX YEEPI