Projektblatt 5 (24 + 4* P)

Abgabe: Donnerstag 17.10 2024, 12:00h

Entpacken Sie zunächst die Archiv-Datei `vorgaben-p5.zip`, in der sich neben den Rahmendateien für die zu lösenden Aufgaben auch mehrere Hilfsklassen sowie eine Datei `out.txt` mit den Test-Ausgaben für Aufgabe 3 befinden. Ergänzen Sie alle .java Dateien zunächst durch einen Kommentar, der Ihren Namen beinhaltet. Ergänzen Sie die Dateien dann durch Ihre Lösungen gemäß der Aufgabenstellung unten. Der abgeänderte bzw. hinzugefügte Java-Sourcecode in den Rahmendateien sollte syntaktisch richtig und vollständig formatiert sein. Alle Dateien mit der Endung `.java` sollten am Ende fehlerfrei übersetzt werden können.
Verpacken Sie die Dateien für Ihre Abgabe in einem ZIP-Archiv mit dem Namen `IhrNachname.IhrVorname.P5.zip`, die Sie auf Ilias hochladen.
Führen Sie dazu in dem Verzeichnis, in dem Sie die Dateien bearbeitet haben, folgenden Befehl auf der Kommandozeile aus:

```
zip  IhrNachname.IhrVorname.P5.zip *.java
```

Aufgabe 1: Rekursion verstehen 4 P

Im Folgenden ist eine rekursive Klassenmethode `r` gegeben sowie eine nichtrekursive Methode `f`, die nur dazu da ist, die rekursive Methode mit einem beliebigen ganzzahligen, positiven Wert `n` und einem initialen Wert für den zweiten Parameter der Methode `r` aufzurufen:

```
1   public static void r (int n, int k){
2       if (n > 1){
3           if ((n % k) == 0) {
4               System.out.print (k + "␣");
5               n /= k;
6           } else {
7               k++;
8           }
9           r (n,k);
10      }
11  }
12
13  public static void f (int n){
14      r(n,2);
15  }
```

Sie finden diesen Code auch in der Klasse `Rekursion1`.

(a) Was gibt die Methode für den Wert $n = 21294$ aus? Geben Sie dazu die vollständige Aufrufhierarchie für die (rekursive) Methode `r` an, wenn diese von der nichtrekursiven Methode f aufgerufen wird:

(b) Was gibt die Methode `f` für einen beliebigen positiven ganzzahligen Wert `n` aus? Beschreiben Sie den Algorithmus, der durch diese Methode umgesetzt wird.

(c) Ergänzen Sie die Klasse `Rekursion1` um eine Klassenmethode, die das gleiche Ergebnis wie die Methode `f` iterativ berechnet.

Aufgabe 2: Rekursive Methoden 10 P

Implementieren Sie in der Datei `Rekursion2.java` die im Folgenden beschriebenen Klassenmethoden und testen Sie die Methoden dann mit Hilfe des Codes in der `main`-Methode der Klasse.

(a) Entwickeln Sie aus der folgenden Gleichung ein rekursives Schema zur Berechnung des Quadrats a^2 einer positiven ganzen Zahl a und implementieren Sie dieses in der Klassenmethode `quadratRek`, um das Quadrat eines Werts vom Typ `int` zu berechnen und als Ergebnis zurückzugeben.

$$a^2 = ((a-1)+1)^2 = (a-1)^2 + 2 * a - 1 \qquad \forall a > 0$$

(b) Nun sollen Sie alle Möglichkeiten bestimmen, eine Treppe mit n Stufen hochzugehen, wenn es dabei in jedem Schritt die Möglichkeit gibt, entweder eine oder zwei Treppenstufen auf einmal zu erklimmen. Implementieren Sie hierzu die Klassenmethode `treppen`, deren Parameter die Anzahl n der (noch) zu erklimmenden Stufen sowie einen String beinhalten, welcher die bisherigen Schritte beschreibt. Der String `"-1-1"` repräsentiert beispielsweise den Umstand, dass bis zu dem aktuellen Aufruf der Methode zweimal eine Treppenstufe erklommen wurden.

Der rekursive Algorithmus, den Sie hier verwenden sollen, kann wie folgt beschrieben werden:

(1) Wenn keine Treppenstufen mehr übrig bleiben, muss der als Parameter übergebene String ausgegeben werden.

(2) Wenn nur noch eine Treppenstufe übrig ist, muss noch `"-1"` an den String angehängt und dieser dann ausgegeben werden.

(3) Wenn noch mehr Treppenstufe übrig sind, sollen rekursive Aufrufe für alle Möglichkeiten (mit den entsprechend angepassten Parametern) durchgeführt werden.

Die Methode `treppen` soll jeweils die Anzahl der Möglichkeiten zurückgeben, d.h. hier die Anzahl der ausgegebenen Strings. In den Schritten (1) und (2) muss daher der Wert 1 zurückgegeben werden und in Schritt (3) die Summe der Rückgabewerte aller rekursiven Aufrufe.

Hinweis Für $n = 20$ gibt es bereits 10946 Möglichkeiten. Sie sollten daher beim Testen kleinere Werte verwenden.

(c) Eine Collatzfolge ist eine Zahlenfolge $(a_i)_{i=0,1,2,\ldots}$, die, beginnend mit einer beliebigen natürlichen Zahl, nach folgendem Schema generiert wird:

$$a_{n+1} = \begin{cases} a_n/2 & \text{falls } a_n \text{ gerade} \\ 3 * a_n + 1 & \text{falls } a_n \text{ ungerade} \end{cases}$$

Für $a_0 = 19$ wird beispielsweise die Folge $19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1, 4, 2, 1, \ldots$ generiert.
Obwohl bisher unbewiesen, läuft offenbar jede Folge mit $a_0 > 0$ in den Zyklus $4, 2, 1$, der sich dann unendlich wiederholt. Als Länge einer Collatz-Folge wird die Anzahl der Elemente bis zur ersten 1 (einschließlich dieses Elements) bezeichnet.

Mit Hilfe der Methoden `naechstesElement` und `collatz` sollen Sie nun diese Folge generieren und ausgeben. Zusätzlich soll am Ende die Länge der Sequenz ausgegeben werden.

Die Methode `naechstesElement` soll dabei für ein als Parameter übergebenes Folgenelement a_i das nächste Folgenelement a_{i+1} berechnen und als Ergebnis zurückgeben.

Die Methode `collatz` soll nun rekursiv eine Collatzfolge bis zu dem ersten Vorkommen der 1 in der Folge berechnen und ausgeben. Hierzu wird der Methode ein Element und eine Zählvariable für die Länge der Folge übergeben. In jedem Rekursionsschritt muss das aktuelle Element ausgegeben und überprüft werden, ob noch ein weiterer Rekursionsschritt gemacht werden soll oder die Berechnung beendet und nur noch die Länge der Folge ausgegeben werden muss.

Für die Eingabewerte

(a) z = 5

(b) n = 5

(c) a_0 = 19

sollte die Ausgabe des Programms dann wie folgt aussehen:

```
(a)
z = 5
z^2 = 25

(b)
n = 5
-1-1-1-1-1
-1-1-1-2
-1-1-2-1
-1-2-1-1
-1-2-2
-2-1-1-1
-2-1-2
-2-2-1
Anzahl der Möglichkeiten: 8

(c)
a_0 = 19
19 58 29 88 44 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
Laenge der Folge: 21
```

Für Teil (b) kann die Reihenfolge der ausgegebenen Strings, je nach Implementierung, variieren.

Aufgabe 3: Listen — 10 + 4* Punkte

In dieser Aufgabe sollen Sie Methoden zur Verwaltung einer Liste von Ausgaben entwickeln. Dazu sind die vier Klassen `Ausgabe`, `Listenelement`, `Liste` und `Aufgabe_5_3` vorgegeben:

- In der Klasse `Ausgabe` werden vier private Attribute für den Tag und dem Monat der Ausgabe sowie für den ausgegebenen Betrag und eine Beschreibung der Ausgabe deklariert. Für jedes der Attribute gibt es eine `getXXX`-Methode, die den Wert des zugehörigen Attributs liefert. Die Methode `print` können Sie verwenden, um ein Objekt der Klasse `Ausgabe` auf der Kommandozeile auszugeben. Mit Hilfe der Methode `istImMonat` können Sie überprüfen, ob die Ausgabe in einem bestimmten Monat stattgefunden hat, welcher der Methode als Parameter übergeben wird.

- Die Klasse `Listenelement` repräsentiert ein einzelnes Element einer Liste und beinhaltet als Attribute zwei Referenzvariablen, die auf ein Objekt der Klasse `Ausgabe` bzw. auf das nächste Element in der Liste verweisen.

- Die Klasse `Liste` repräsentiert eine Listenstruktur. Um die Verwaltung der Liste zu vereinfachen, sind zwei Pseudo-Listenelemente für den Kopf und das Ende der Liste definiert. Diese werden im einzigen Konstruktor der Liste initialisiert. In der Klasse werden bereits folgende Methoden zur Verfügung gestellt:

  - Die Methode `print` gibt alle (echten) Elemente der Liste nacheinander aus.
  - Die Methode `length` bestimmt die Anzahl aller (echten) Listenelemente und gibt diese als Ergebnis zurück.
  - Die Methode `insert` fügt ein neues Listenelement am Kopf der Liste ein, welches die als Parameter übergebene Instanz der Klasse Ausgabe (also das eigentliche Datenelement) referenziert.

- Die Klasse `Aufgabe_5_3` besitzt lediglich eine `main`-Methode, mit der Sie Ihre Implementierung testen können. Hierzu müssen Sie später die Kommentarzeichen vor einigen der Codezeilen entfernen.

Ergänzen Sie nun die Klasse `Liste` wie im Folgenden beschrieben:

(a) Implementieren Sie die Instanz-Methode `getSummeAusgaben` in der Klasse `Liste`: In der Methode soll die Summe aller Ausgabe-Beträge in der Liste berechnet und als Ergebnis zurückgegeben werden. Durchlaufen Sie dazu die Liste vom ersten bis zum letzten Element und summieren Sie die Betragswerte aus den einzelnen Daten-Objekten der Listenelemente auf. Geben Sie die Betragssumme am Ende zurück.

(b) Implementieren Sie die Methode `getAusgabenListeMonat` die eine neue Liste generieren soll, welche alle Ausgaben enthält, die in dem als Parameter übergebenen Monat getätigt wurden. Diese neue Liste soll am Ende zurückgegeben werden.

Hierzu müssen Sie zunächst eine neue Listen-Instanz anlegen. Dann durchlaufen Sie die aktuelle Listen-Instanz vollständig und fügen alle

Datenelemente, die sich auf den im Parameter spezifizierten Monat beziehen, mit Hilfe der Methode `insert` in die neue Liste ein. Die Datenobjekte sollen hierbei nicht kopiert werden, sondern nur die Referenzen auf diese. Der Vergleich der Monatsnamen soll mit Hilfe der Methode `istImMonat` der Klasse `Ausgabe` durchgeführt werden.

(c) Schreiben Sie eine Methode mit dem Namen `sort`, welche die aktuelle (in der Instanz gespeicherten) Liste mit Hilfe eines vereinfachten Bubblesort-Algorithmus sortiert. Durchlaufen Sie dazu die Liste $(n-1)$ Mal, wobei n der Länge der Liste, also der Anzahl der echten Listenelemente entspricht. Definieren Sie sich weiter zwei Referenz-Variablen $e1$ und $e2$ vom Typ `Listenelement`. Diese beiden Referenzen sollen bei einem einzelnen Durchlauf durch die Liste immer auf je zwei aufeinanderfolgende Listenelemente verweisen. Gehen Sie dabei bei einem einzelnen Durchlauf wie folgt vor:

Algorithm 1 Sortierschritt Bubblesort auf Listen

```
1: zwei Referenzen mit erstem und zweitem Listenelement initialisieren
2: while zweites Listenelement zeigt nicht auf Ende do
3:     vergleiche Datenelement von erstem und zweitem Listenelement
4:     if erstes Element ist größer als zweites Datenelement then
5:         tausche Elemente aus
6:     beide Listenelemente um eine Position weitersetzen
```

Schreiben Sie sich für die Vergleichs- und die Tausch-Operationen zwei Methoden `groesser` und `tausche`. Die Methode `groesser` benötigt als Parameter zwei Referenzen auf die zu vergleichenden Listenelemente und soll den Wert `true` zurückgeben, falls der in dem Datenelement des ersten Listenelements gespeicherte Tag nach dem Tag liegt, der in dem Datenelement des zweiten Listenelements gespeichert ist (der Monat wird dabei nicht berücksichtigt) und den Wert `false`, falls dies nicht der Fall ist.

Die Methode `tausche` soll als Parameter ebenfalls zwei Referenzen auf die zu tauschenden Listenelemente besitzen und deren Datenelemente (über die Referenzen) vertauschen.

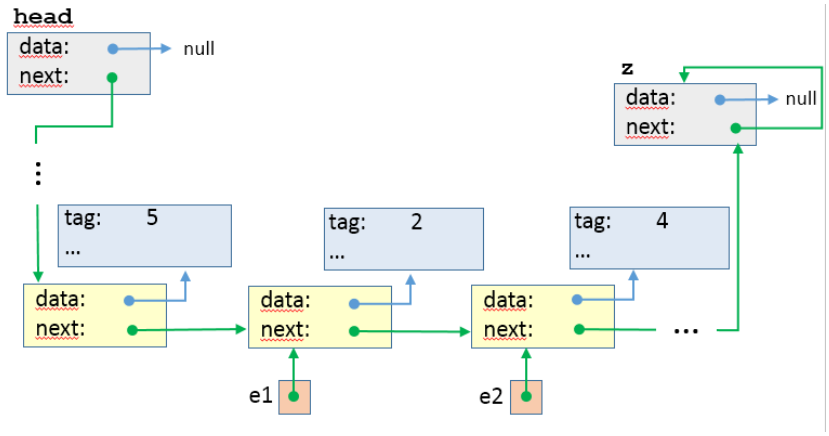
(d***) Schreiben Sie eine Methode `sort2`, die den gleichen Sortieralgoritmus wie die Methode `sort` implementiert, bei der aber beim Tausch zweier Elemente die kompletten Listenelemente in der Liste vertauscht werden. Definieren Sie dazu einen weitere Referenz-Variable p, die immer auf das Element unmittelbar vor den beiden zu vergleichenden Listenelementen zeigt. Überladen Sie dann auch die Methode `tausche` mit einer Methode, die drei Referenzen auf (aufeinanderfolgende) Listenelemente als Parameter besitzt und die die `next`-Referenzen so "verbiegt", das die Reihenfolge des zweiten und dritten Listenelements in der Liste vertauscht wird (siehe Bild ganz unten).

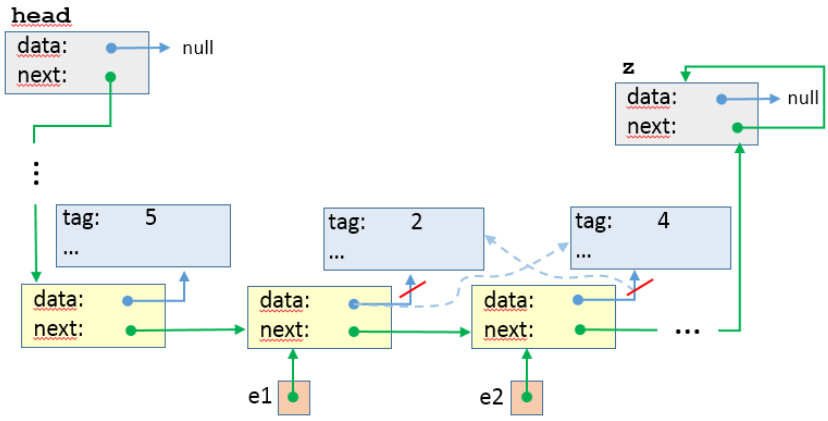
In der `main`-Methode finden Sie (z.T. auskommentierten) Code, durch den zunächst eine neue Liste angelegt und mit einer Reihe von Ausgabe-Objekten (unsortiert) gefüllt wird. Diese Liste wird dann ausgegeben. Danach werden zwei neue Listen generiert, die nur die Ausgaben für August bzw. September beinhalten. Für diese Listen wird dann die Summe der Ausgaben bestimmt und ausgegeben. Danach werden die Monatslisten nach dem Tag im Monat sortiert und am Ende ebenfalls ausgegeben. Nutzen Sie diesen Code, um Ihre Methoden zu testen.
Bei korrekter Implementierung sollte folgende Ausgabe erzeugt werden:

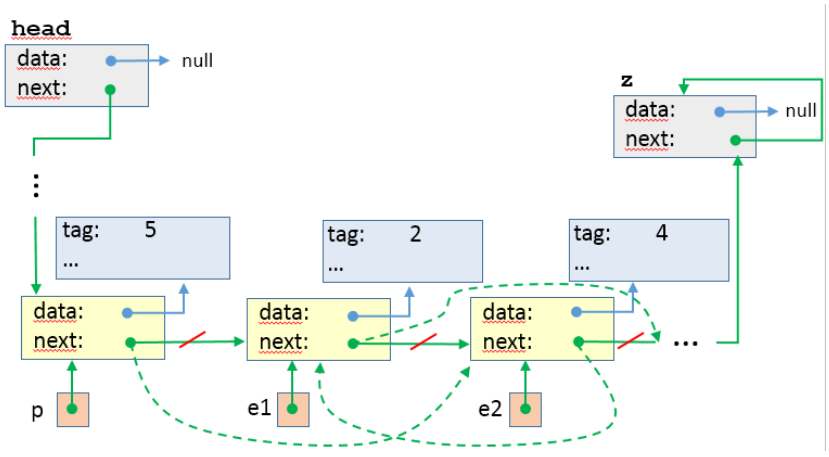
```
1. August: 14.95 (Bücher)
16. September: 52.1 (Lebensmittel)
8. August: 12.35 (Schreibwaren)
1. August: 3.5 (Zeitung)
6. August: 0.95 (Backwaren)
2. September: 37.5 (Lebensmittel)
18. August: 10.2 (Backwaren)
19. August: 29.5 (Lebensmittel)
5. August: 35.56 (Lebensmittel)
11. August: 8.8 (Backwaren)
28. September: 39.0 (Fahrkarte)
3. September: 178.0 (Bekleidung)
10. August: 3.8 (Backwaren)
4. August: 7.5 (Backwaren)
9. September: 42.1 (Lebensmittel)
1. August: 3.55 (Backwaren)
12. August: 31.55 (Lebensmittel)
```

```
Summe der Ausgaben im August: 162.20999999999998
Summe der Ausgaben im September: 348.70000000000005

1. August: 3.55 (Backwaren)
1. August: 3.5 (Zeitung)
1. August: 14.95 (Bücher)
4. August: 7.5 (Backwaren)
5. August: 35.56 (Lebensmittel)
6. August: 0.95 (Backwaren)
8. August: 12.35 (Schreibwaren)
10. August: 3.8 (Backwaren)
11. August: 8.8 (Backwaren)
12. August: 31.55 (Lebensmittel)
18. August: 10.2 (Backwaren)
19. August: 29.5 (Lebensmittel)

2. September: 37.5 (Lebensmittel)
3. September: 178.0 (Bekleidung)
9. September: 42.1 (Lebensmittel)
16. September: 52.1 (Lebensmittel)
28. September: 39.0 (Fahrkarte)
```

Hinweis: Die folgenden Bilder zeigen eine verkettete Liste mit zusätzlichen Referenzen auf einzelne Elemente. Das erste Bild zeigt eine Situation, in der die Referenzen e1 und e2 auf zwei zu vertauschende Listenelemente zeigen. Das zweite und das dritte Bild zeigen, welche Referenzen (gestrichelte Linien) in der Methode `tausche` jeweils in Aufgabenteil (c) bzw. (d) neu gesetzt werden müssen.


head
data:
next:
null
z
data:
next:
null
tag: 5
...
tag: 2
...
tag: 4
...
data:
next:
data:
next:
data:
next:
...
e1
e2


head
data:
next:
null
z
data:
next:
null
tag: 5
...
tag: 2
...
tag: 4
...
data:
next:
data:
next:
data:
next:
...
e1
e2


head
data:
next:
null
z
data:
next:
null
tag: 5
...
tag: 2
...
tag: 4
...
data:
next:
data:
next:
data:
next:
...
p
e1
e2