



Name: _____

Abiturprüfung 2013

Informatik, Leistungskurs

Aufgabenstellung:

In einer *Ahnentafel* werden die Vorfahren eines Individuums, des sogenannten Probanden, dargestellt. Da jedes Individuum zwei Elternteile hat, die aber nicht beide bekannt sein müssen, lässt sich die Ahnentafel als Binärbaum darstellen, wobei der Proband in der Wurzel des Baumes steht. Die Ebenen des Baumes enthalten die Individuen einer Generation. Der Proband hat die Generationsnummer Null.

Jedes Individuum in der Ahnentafel wird durch die eindeutige Kekulé-Zahl gekennzeichnet, die nach folgenden Regeln gebildet wird:

- (i) Der Proband erhält immer die Kekulé-Zahl 1.
- (ii) Hat ein Individuum die Kekulé-Zahl n , so erhält sein Vater die Nummer $2 \cdot n$ und seine Mutter die Nummer $2 \cdot n + 1$.

Die folgende Abbildung 1 zeigt die Ahnentafel des Probanden Johann Wolfgang Goethe als Binärbaum bis zur dritten Generation.

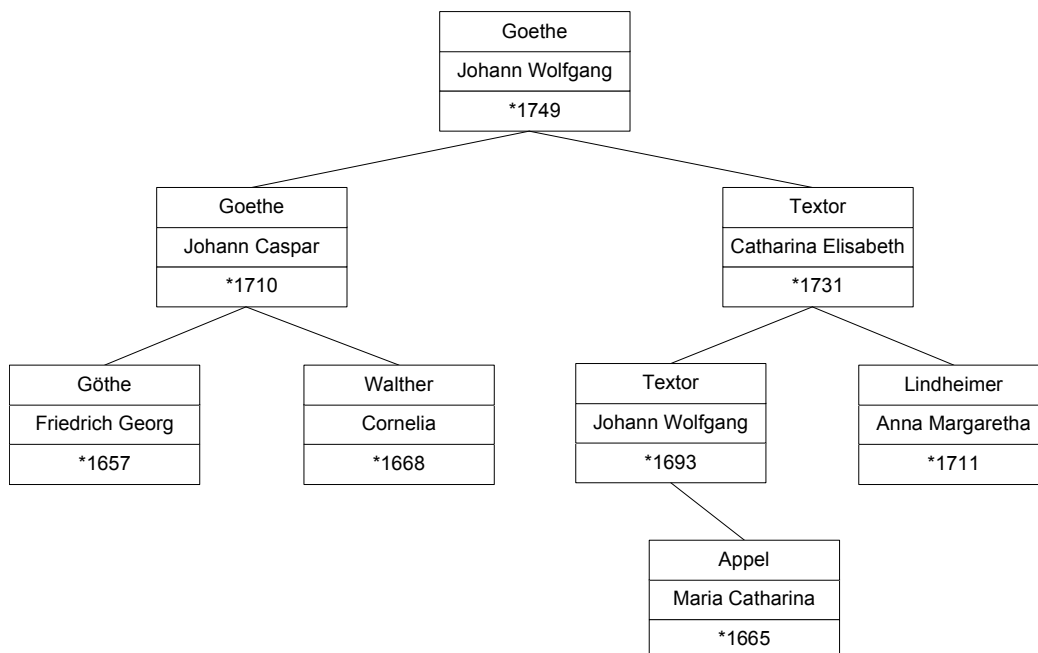


Abbildung 1: Ausschnitt aus der Ahnentafel von Johann Wolfgang Goethe



Name: _____

a) *Geben Sie die Kekulé-Zahlen der Individuen in der Anlage 1 an.*

Beschreiben Sie, wie die Kekulé-Zahl des Individuums Maria Catharina Appel, ausgehend vom Probanden, zustande kommt.

Georg Walther, geboren 1638, war der Vater von Johann Wolfgang Goethes Großmutter väterlicherseits.

Ermitteln Sie die Kekulé-Zahl von Georg Walther und ordnen Sie das Individuum in den Baum in der Anlage 1 ein.

Anna Maria Walther, geboren 1633, hat die Kekulé-Zahl 27.

Ermitteln Sie die Kekulé-Zahl ihres Kindes und ordnen Sie Anna Maria in den Baum in der Anlage 1 ein.

(8 Punkte)

Die Verwaltung einer Ahnentafel soll mit dem Rechner erfolgen. Dazu wird die Ahnentafel durch eine Klasse `Ahnentafel` repräsentiert, die einen Binärbaum `ahnenbaum` der Klasse `BinaryTree` beinhaltet. Im Baum werden die Individuen gespeichert. Die Merkmale eines Individuums werden in einem Objekt der Klasse `Individuum` festgehalten. Zu den Klassen ist im Anhang eine entsprechende Dokumentation beigelegt.

b) *Entwickeln Sie ein Implementationsdiagramm für die beiden Klassen `Ahnentafel` und `Individuum`, bei dem auch die Beziehungen dargestellt werden. Berücksichtigen Sie dabei die Dokumentation. Die Methoden müssen nicht angegeben werden.*

(6 Punkte)



Name: _____

c) Gegeben ist die Methode `gibIndividuum` in der Klasse `Ahnentafel`:

```
1 public Individuum gibIndividuum(int pKekule) {
2     if (pKekule > 0) {
3         Stack pFadZumIndividuum = new Stack();
4         int tempKekule = pKekule;
5         while (tempKekule > 1) {
6             pFadZumIndividuum.push(new Boolean(tempKekule % 2 == 0));
7             tempKekule = tempKekule / 2;
8         }
9         BinaryTree aktTeilbaum = gibAhnentafel();
10        while (!pFadZumIndividuum.isEmpty() &&
11                !aktTeilbaum.isEmpty()) {
12            if (((Boolean) pFadZumIndividuum.top()).booleanValue()) {
13                aktTeilbaum = aktTeilbaum.getLeftTree();
14            } else {
15                aktTeilbaum = aktTeilbaum.getRightTree();
16            }
17            pFadZumIndividuum.pop();
18        }
19        if (!aktTeilbaum.isEmpty()) {
20            return (Individuum) aktTeilbaum.getObject();
21        } else {
22            return null;
23        }
24    } else {
25        return null;
26    }
27 }
```

Analysieren Sie die Methode, indem Sie den Ablauf der Methode für den Aufruf `gibIndividuum(13)`, angewandt auf den Ahnentafel (siehe Abbildung 1), simulieren. Dokumentieren Sie hierzu mit Hilfe des Protokolls in Anlage 2 zu diesem Aufgabenteil die Werte der dort angegebenen Variablen.

Erläutern Sie die Idee des Algorithmus.

(14 Punkte)



Name: _____

- d) Die Klasse `Ahnentafel` soll um eine Methode `gibAnzahlGenerationen` mit dem Methodenkopf

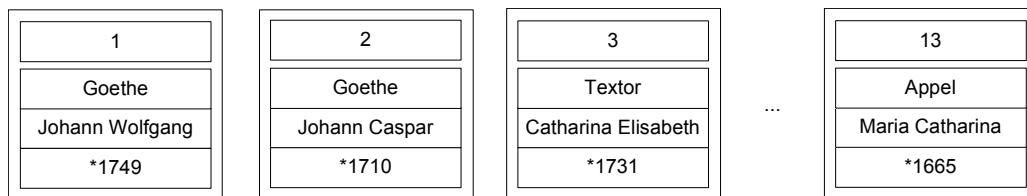
```
public int gibAnzahlGenerationen()
```

erweitert werden, die die Anzahl der in der Ahnentafel gespeicherten Generationen zurückgibt. Die Generationen müssen dabei nicht unbedingt vollständig gespeichert sein.

Implementieren Sie die Methode `gibAnzahlGenerationen` und gegebenenfalls weitere Hilfsmethoden.

(10 Punkte)

- e) In Büchern ist die Darstellung einer umfangreichen Ahnentafel wie etwa die von Goethe in Form eines Baumes unpraktisch. In diesem Fall stellt man die Ahnentafel als Liste dar, die nach aufsteigender Kekulé-Zahl sortiert ist.



Die Klasse `Ahnentafel` soll um die Methode `gibAhnentafelAlsListe` mit dem Methodenkopf

```
public List gibAhnentafelAlsListe()
```

erweitert werden, die die Individuen des Ahnenbaums zusammen mit ihrer Kekulé-Zahl in eine nach aufsteigender Kekulé-Zahl sortierten Liste schreibt.

Entwickeln Sie eine Lösungsidee und implementieren Sie die Methode `gibAhnentafelAlsListe` und gegebenenfalls weitere Klassen und Hilfsmethoden.

(12 Punkte)

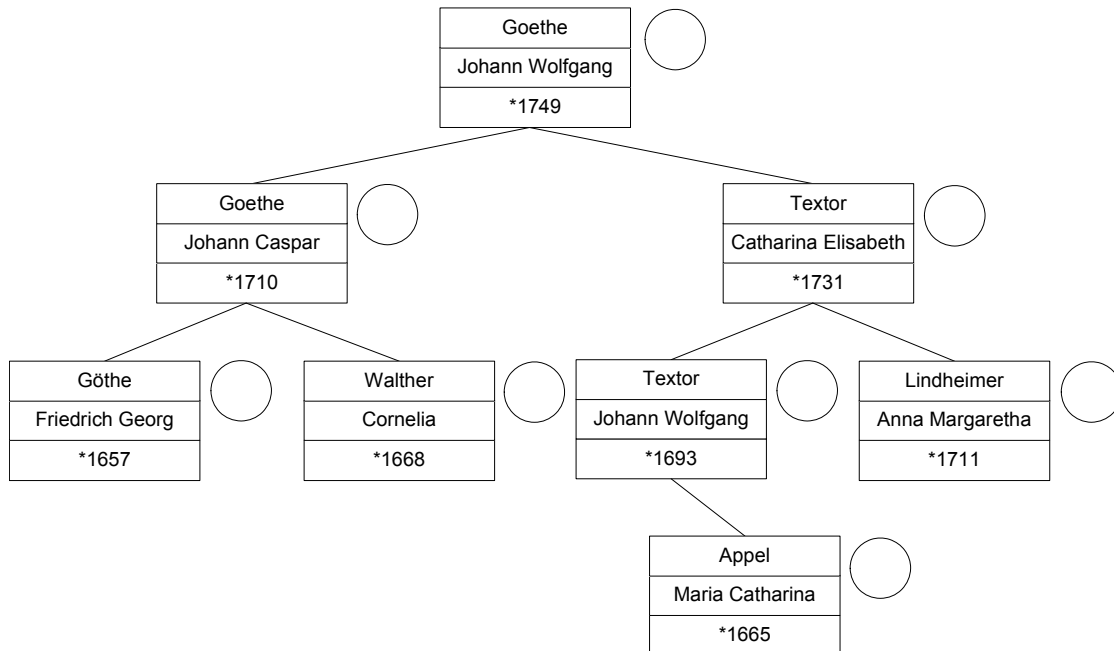
Zugelassene Hilfsmittel:

- Wörterbuch zur deutschen Rechtschreibung
- Taschenrechner



Name: _____

Anlage 1: Ahnentafel von Johann Wolfgang Goethe für Aufgabenteil a)





Name: _____

Anlage 2: Protokollvorlage für Aufgabenteil c)

Protokoll der ersten while-Schleife (Z. 5 – Z. 8)

Durchlauf	Wert von tempKekule (nach dem Durchlauf)	Stack (nach dem Durchlauf)			
1.		<table><tr><td></td></tr><tr><td></td></tr><tr><td></td></tr></table>			
2.		<table><tr><td></td></tr><tr><td></td></tr><tr><td></td></tr></table>			
3.		<table><tr><td></td></tr><tr><td></td></tr><tr><td></td></tr></table>			

Protokoll der zweiten while-Schleife (Z. 10 – Z. 17)

Durchlauf	Inhaltsobjekt von aktTeilbaum (nach dem Durchlauf) <i>Notieren Sie Vorname und Name</i>	Stack (nach dem Durchlauf)			
1.		<table><tr><td></td></tr><tr><td></td></tr><tr><td></td></tr></table>			
2.		<table><tr><td></td></tr><tr><td></td></tr><tr><td></td></tr></table>			
3.		<table><tr><td></td></tr><tr><td></td></tr><tr><td></td></tr></table>			



Name: _____

Anhang

Die Klasse Individuum

Objekte der Klasse **Individuum** verwalten die Daten (Nachname, Vorname, Geburtsjahr, Geschlecht) einzelner Individuen.

Dokumentation der Klasse Individuum

Konstruktor `Individuum(String pName, String pVorname,
int pGeburtsjahr, char pGeschlecht)`

Ein Objekt der Klasse `Individuum` wird erzeugt und mit den übergebenen Daten gesetzt. Dabei steht 'm' für männlich und 'w' für weiblich.

Anfrage `String gibName()`
Die Anfrage liefert den Nachnamen des Individuums.

Anfrage `String gibVorname()`
Die Anfrage liefert den Vornamen des Individuums.

Anfrage `int gibGeburtsjahr()`
Die Anfrage liefert das Geburtsjahr des Individuums.

Anfrage `char gibGeschlecht()`
Die Anfrage liefert das Geschlecht des Individuums. Dabei steht 'w' für weiblich und 'm' für männlich.



Name: _____

Die Klasse Ahnentafel

Objekte der Klasse **Ahnentafel** verwalten die Ahnentafel eines Probanden.

Ausschnitt aus der Dokumentation der Klasse Ahnentafel

Konstruktor Ahnentafel()

Eine leere Ahnentafel wird erzeugt.

Konstruktor Ahnentafel(Individuum pProband)

Eine Ahnentafel mit den Daten eines Probanden wird erzeugt.

Anfrage Individuum gibProband()

Die Anfrage liefert den Probanden als Objekt der Klasse Individuum zurück.

Anfrage Individuum gibIndividuum(int pKekule)

Die Anfrage liefert das Individuum mit der übergebenen Kekulé-Zahl als Objekt der Klasse Individuum zurück. Existiert zur übergebenen Kekulé-Zahl kein Individuum, so wird null zurückgegeben.

Anfrage BinaryTree gibAhnenbaum()

Die Anfrage liefert den Ahnenbaum zurück.

Anfrage int gibAnzahlGenerationen()

Die Anfrage liefert die Anzahl der Generationen in der Ahnentafel.
Die Methode wird vom Prüfling im Aufgabenteil d) implementiert.

Anfrage List gibAhnentafelAlsListe()

Die Anfrage liefert eine nach der Kekulé-Zahl aufsteigend sortierte Liste der Individuen in der Ahnentafel zusammen mit ihrer Kekulé-Zahl zurück.
Die Methode wird vom Prüfling im Aufgabenteil e) implementiert.



Name: _____

Die Klasse Stack

Objekte der Klasse **Stack** (Keller, Stapel) verwalten beliebige Objekte nach dem Last-In-First-Out-Prinzip, d. h., das zuletzt abgelegte Objekt wird als erstes wieder entnommen.

Dokumentation der Klasse Stack

Konstruktor **Stack()**

Ein leerer Stapel wird erzeugt.

Anfrage **boolean isEmpty()**

Die Anfrage liefert den Wert `true`, wenn der Stapel keine Objekte enthält, sonst liefert sie den Wert `false`.

Auftrag **void push(Object pObject)**

Das Objekt `pObject` wird oben auf den Stapel gelegt. Falls `pObject` gleich `null` ist, bleibt der Stapel unverändert.

Auftrag **void pop()**

Das zuletzt eingefügte Objekt wird von dem Stapel entfernt. Falls der Stapel leer ist, bleibt er unverändert.

Anfrage **Object top()**

Die Anfrage liefert das oberste Stapelobjekt. Der Stapel bleibt unverändert. Falls der Stapel leer ist, wird `null` zurückgegeben.



Name: _____

Die Klasse Queue

Objekte der Klasse **Queue** (Warteschlange) verwalten beliebige Objekte nach dem First-In-First-Out-Prinzip, d. h., das zuerst abgelegte Objekt wird als erstes wieder entnommen.

Dokumentation der Klasse Queue

Konstruktor **Queue()**

Eine leere Schlange wird erzeugt.

Anfrage **boolean isEmpty()**

Die Anfrage liefert den Wert `true`, wenn die Schlange keine Objekte enthält, sonst liefert sie den Wert `false`.

Auftrag **void enqueue(Object pObject)**

Das Objekt `pObject` wird an die Schlange angehängt. Falls `pObject` gleich `null` ist, bleibt die Schlange unverändert.

Auftrag **void dequeue()**

Das erste Objekt wird aus der Schlange entfernt. Falls die Schlange leer ist, wird sie nicht verändert.

Anfrage **Object front()**

Die Anfrage liefert das erste Objekt der Schlange. Die Schlange bleibt unverändert. Falls die Schlange leer ist, wird `null` zurückgegeben.



Name: _____

Die Klasse **List**

Objekte der Klasse **List** verwalten beliebig viele, linear angeordnete Objekte. Auf höchstens ein Listenobjekt, aktuelles Objekt genannt, kann jeweils zugegriffen werden. Wenn eine Liste leer ist, vollständig durchlaufen wurde oder das aktuelle Objekt am Ende der Liste gelöscht wurde, gibt es kein aktuelles Objekt. Das erste oder das letzte Objekt einer Liste können durch einen Auftrag zum aktuellen Objekt gemacht werden. Außerdem kann das dem aktuellen Objekt folgende Listenobjekt zum neuen aktuellen Objekt werden.

Das aktuelle Objekt kann gelesen, verändert oder gelöscht werden. Außerdem kann vor dem aktuellen Objekt ein Listenobjekt eingefügt oder ein Listenobjekt an das Ende der Liste angefügt werden.

Dokumentation der Klasse **List**

Konstruktor **List()**

Eine leere Liste wird erzeugt.

Anfrage **boolean isEmpty()**

Die Anfrage liefert den Wert `true`, wenn die Liste keine Objekte enthält, sonst liefert sie den Wert `false`.

Anfrage **boolean hasAccess()**

Die Anfrage liefert den Wert `true`, wenn es ein aktuelles Objekt gibt, sonst liefert sie den Wert `false`.

Auftrag **void next()**

Falls die Liste nicht leer ist, es ein aktuelles Objekt gibt und dieses nicht das letzte Objekt der Liste ist, wird das dem aktuellen Objekt in der Liste folgende Objekt zum aktuellen Objekt, andernfalls gibt es nach Ausführung des Auftrags kein aktuelles Objekt, d. h., `hasAccess()` liefert den Wert `false`.

Auftrag **void toFirst()**

Falls die Liste nicht leer ist, wird das erste Objekt der Liste aktuelles Objekt. Ist die Liste leer, geschieht nichts.

Auftrag **void toLast()**

Falls die Liste nicht leer ist, wird das letzte Objekt der Liste aktuelles Objekt. Ist die Liste leer, geschieht nichts.



Name: _____

Anfrage Object getObject()

Falls es ein aktuelles Objekt gibt (`hasAccess() == true`), wird das aktuelle Objekt zurückgegeben, andernfalls (`hasAccess() == false`) gibt die Anfrage den Wert `null` zurück.

Auftrag void setObject(Object pObject)

Falls es ein aktuelles Objekt gibt (`hasAccess() == true`) und `pObject` ungleich `null` ist, wird das aktuelle Objekt durch `pObject` ersetzt. Sonst bleibt die Liste unverändert.

Auftrag void append(Object pObject)

Ein neues Objekt `pObject` wird am Ende der Liste eingefügt. Das aktuelle Objekt bleibt unverändert. Wenn die Liste leer ist, wird das Objekt `pObject` in die Liste eingefügt und es gibt weiterhin kein aktuelles Objekt (`hasAccess() == false`). Falls `pObject` gleich `null` ist, bleibt die Liste unverändert.

Auftrag void insert(Object pObject)

Falls es ein aktuelles Objekt gibt (`hasAccess() == true`), wird ein neues Objekt vor dem aktuellen Objekt in die Liste eingefügt. Das aktuelle Objekt bleibt unverändert. Falls die Liste leer ist und es somit kein aktuelles Objekt gibt (`hasAccess() == false`), wird `pObject` in die Liste eingefügt und es gibt weiterhin kein aktuelles Objekt. Falls es kein aktuelles Objekt gibt (`hasAccess() == false`) und die Liste nicht leer ist oder `pObject` gleich `null` ist, bleibt die Liste unverändert.

Auftrag void concat(List pList)

Die Liste `pList` wird an die Liste angehängt. Anschließend wird `pList` eine leere Liste. Das aktuelle Objekt bleibt unverändert. Falls `pList` `null` oder eine leere Liste ist, bleibt die Liste unverändert.

Auftrag void remove()

Falls es ein aktuelles Objekt gibt (`hasAccess() == true`), wird das aktuelle Objekt gelöscht und das Objekt hinter dem gelöschten Objekt wird zum aktuellen Objekt. Wird das Objekt, das am Ende der Liste steht, gelöscht, gibt es kein aktuelles Objekt mehr (`hasAccess() == false`). Wenn die Liste leer ist oder es kein aktuelles Objekt gibt (`hasAccess() == false`), bleibt die Liste unverändert.



Name: _____

Die Klasse BinaryTree

Mithilfe der Klasse **BinaryTree** können beliebig viele Inhaltsobjekte in einem Binärbaum verwaltet werden. Ein Objekt der Klasse stellt entweder einen leeren Baum dar oder verwaltet ein Inhaltsobjekt sowie einen linken und einen rechten Teilbaum, die ebenfalls Objekte der Klasse **BinaryTree** sind.

Dokumentation der Klasse BinaryTree

Konstruktor **BinaryTree()**

Nach dem Aufruf des Konstruktors existiert ein leerer Binärbaum.

Konstruktor **BinaryTree(Object pObject)**

Wenn der Parameter `pObject` ungleich `null` ist, existiert nach dem Aufruf des Konstruktors der Binärbaum und hat `pObject` als Inhaltsobjekt und zwei leere Teilbäume. Falls der Parameter `null` ist, wird ein leerer Binärbaum erzeugt.

Konstruktor **BinaryTree(Object pObject, BinaryTree pLeftTree, BinaryTree pRightTree)**

Wenn der Parameter `pObject` ungleich `null` ist, wird ein Binärbaum mit `pObject` als Inhaltsobjekt und den beiden Teilbäumen `pLeftTree` und `pRightTree` erzeugt. Sind `pLeftTree` oder `pRightTree` gleich `null`, wird der entsprechende Teilbaum als leerer Binärbaum eingefügt. Wenn der Parameter `pObject` gleich `null` ist, wird ein leerer Binärbaum erzeugt.

Anfrage **boolean isEmpty()**

Diese Anfrage liefert den Wahrheitswert `true`, wenn der Binärbaum leer ist, sonst liefert sie den Wert `false`.

Auftrag **void setObject(Object pObject)**

Wenn der Binärbaum leer ist, wird der Parameter `pObject` als Inhaltsobjekt sowie ein leerer linker und rechter Teilbaum eingefügt. Ist der Binärbaum nicht leer, wird das Inhaltsobjekt durch `pObject` ersetzt. Die Teilbäume werden nicht geändert. Wenn `pObject` `null` ist, bleibt der Binärbaum unverändert.

Anfrage **Object getObject()**

Diese Anfrage liefert das Inhaltsobjekt des Binärbaums. Wenn der Binärbaum leer ist, wird `null` zurückgegeben.



Name: _____

Auftrag **void setLeftTree(BinaryTree pTree)**

Wenn der Binärbaum leer ist, wird pTree nicht angehängt. Andernfalls erhält der Binärbaum den übergebenen Baum als linken Teilbaum. Falls der Parameter null ist, ändert sich nichts.

Auftrag **void setRightTree(BinaryTree pTree)**

Wenn der Binärbaum leer ist, wird pTree nicht angehängt. Andernfalls erhält der Binärbaum den übergebenen Baum als rechten Teilbaum. Falls der Parameter null ist, ändert sich nichts.

Anfrage **BinaryTree getLeftTree()**

Diese Anfrage liefert den linken Teilbaum des Binärbaumes. Der Binärbaum ändert sich nicht. Wenn der Binärbaum leer ist, wird null zurückgegeben.

Anfrage **BinaryTree getRightTree()**

Diese Anfrage liefert den rechten Teilbaum des Binärbaumes. Der Binärbaum ändert sich nicht. Wenn der Binärbaum leer ist, wird null zurückgegeben.