Klausur Informatik II (20.09.2004)

Vorname	Klaus
Nachname	Wachtler
Matrikelnummer	12345678
Prüfer	Klaus Wachtler

Die Antworten meiner Musterlösung sind in blau gehalten, während Punktzahlen und Anmerkungen zur Bewertung in rot eingetragen sind.

Allgemeine Hinweise

Die meisten Fragen sind durch freien (aber bitte lesbaren) Text im entsprechenden Feld zu beantworten. Der vorgesehene Platz ist **NICHT** bindend: Einerseits kann man manche Fragen auch sinnvoll beantworten, ohne den ganzen Platz zu benötigen; andererseits kann man auch auf einem zusätzlichen Blatt fortfahren, wenn man einen Roman schreiben will.

Jedes zusätzliche Blatt bitte mit Name und Matrikelnummer versehen sowie der Nummer der zugehörigen Frage, und spätestens bei der Abgabe mit den ausgeteilten Blättern zusammenheften.

Bitte ausschließlich dokumentenechte schwarze oder blaue Stifte (**keinen Bleistift**, keinesfalls rot) verwenden!

Unlesbares wird nicht gewertet. Unverständliche Sätze oder Stichworte ohne Zusammenhang gelten als falsch.

Im Notfall fragen!

Wenn eine Frage unklar oder falsch formuliert ist, dann lieber sofort fragen, anstatt mit meinen Fehlern Zeit zu verschwenden.

Wer eine Wissenslücke hat, die man schnell beheben könnte, und wegen dieser Lücke den Rest einer Aufgabe nicht lösen kann, kann sich einen Hinweis geben lassen, der mit dem Anschreiben angemessener Strafpunkte einhergeht. Damit kann er dann hoffentlich den Rest der Aufgabe lösen, und vermeidet dadurch Folgefehler.

Und jetzt bitte beruhigen, tief Luft holen, und viel Glück!

Prüfungsfragen

1. Beschreiben Sie in Worten, wie in einem *Binären Suchbaum* nach einem gegebenen Element **rekursiv** gesucht wird:

   Indem man folgenden Algorithmus `rsuche` ausführt:

   (a) Prüfe, ob die Wurzel existiert.
   Falls nicht: Abbruch (Element nicht gefunden)

   (b) Prüfe, ob die Wurzel gleich dem gesuchten Element ist.
   Falls ja: Ende (Wurzel ist das gesuchte Element)

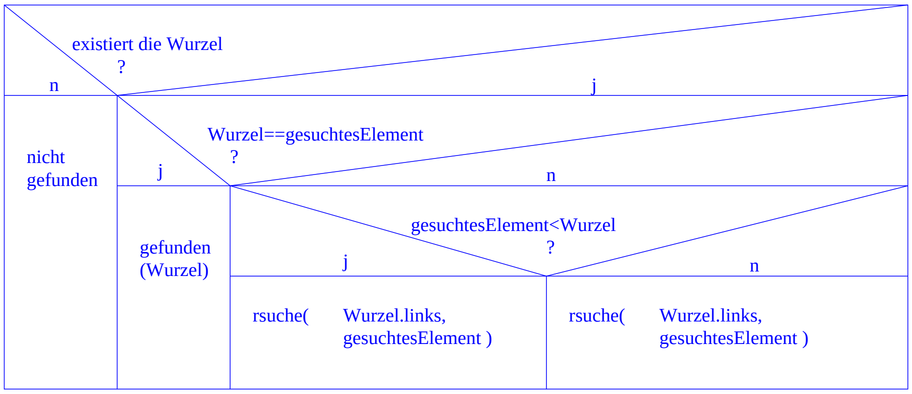
   (c) Falls das gesuchte Element kleiner ist als die Wurzel: rufe `rsuche` mit dem linken Teilbaum auf; sonst: rufe `rsuche` mit dem rechten Teilbaum auf

   Nach der Laufzeitordnung war eigentlich nicht gefragt; aber wenn es trotzdem jemand angibt, dann hoffentlich richtig: im b.c. O(1) (wenn die Wurzel das gesuchte Element ist), im a.c. O(log n) (wenn der Baum halbwegs ausgeglichen ist) und im w.c. O(n) (wenn er entartet ist).

Summe: 10

2. Geben Sie für die vorige Aufgabe ein Nassi-Shneiderman-Diagramm an:

rsuche(Wurzel, gesuchtesElement):



Anmerkung: Gelegentlich sieht man hier auch eine Schleife; das ist natürlich falsch. Das Weitergehen von einem Knoten zu einem seiner Nachfolger steckt in der rekursiven Version implizit im rekursiven Aufruf des Algorithmus.

Summe: 10

3. Beschreiben Sie in Worten, wie in einem *Binären Suchbaum* nach einem gegebenen Element **iterativ** gesucht wird:

   Indem man folgenden Algorithmus `rsuche` ausführt:

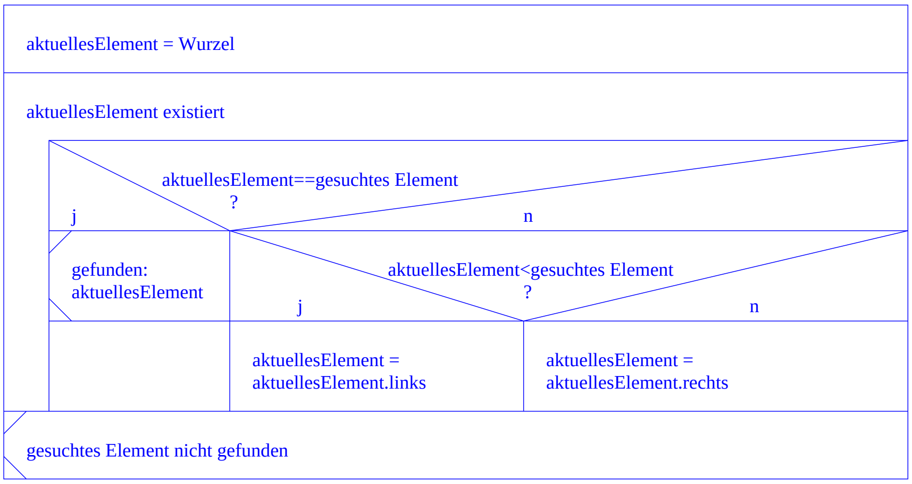
   (a) Setze aktuellesElement=Wurzel
   (b) Prüfe, ob aktuellesElement existiert.
   Falls nicht: Abbruch (Element nicht gefunden)
   (c) Prüfe, ob aktuellesElement gleich dem gesuchten Element ist.
   Falls ja: Ende (aktuellesElement ist das gesuchte Element)
   (d) Falls das gesuchte Element kleiner ist als aktuellesElement:
   setze aktuellesElement=aktuellesElement.links
   Sonst:
   setze aktuellesElement=aktuellesElement.rechts
   (e) gehe zu Schritt (b)

   Bei dieser Aufgabe braucht man eine Schleife (oder wie hier beschrieben einen Sprung zu (b)). Letztlich werden aber dieselben Elemente in derselben Reihenfolge besucht wie bei Aufgabe 1.
   Ggf. die gleiche Laufzeitordnung wie bei Aufgabe 1 (eigentlich nicht nötig)

Summe: 10

4. Geben Sie für die vorige Aufgabe ein Nassi-Shneiderman-Diagramm an:

isuche(Wurzel, gesuchtesElement) :



Summe: 10

5. Welche Möglichkeiten gibt es, über alle Elemente eines Containers zu iterieren? Vergleichen Sie die Möglichkeiten und nennen Sie die Vor- und Nachteile.

   (a) **Cursor**: Der Container merkt sich intern die Position eines *aktuellen Elements* oder *Cursors* und bietet dafür zumindest die folgenden Methoden an:

      i. Setzen des Cursors auf erstes Element

      ii. Um ein Element weitersetzen

      iii. Liefern des aktuellen Elements (am Cursor)

      iv. Prüfen, ob der Cursor noch auf ein gültig zeigt (ggf. in Kombination mit der vorigen Methode)

      Die Schleife über alle Objekte liegt im Code des Anwenders des Containers; mit jedem Schleifendurchlauf wird mit der Methode (ii) um ein Element weitergesetzt, mit der Methode (iii) wird auf das Element zugegruiffen und (iv) wird für den Schleifenabbruch verwendet.

      5

      Vor-/Nachteile:

      + meist leicht zu implementieren (je nach Art des Containers, z.B. bei Listen)
      - bei manchen Containern kann es schwierig sein, von einem Element zum nächsten zu kommen. Bei Bäumen ist es leicht, von einem Knoten zu einem Kindknoten zu kommen, aber meist umso schwieriger (oder uneffektiv), zu einem Vaterknoten aufzusteigen.
      - pro Container existiert nur ein solcher Cursor. Damit ist es weder möglich, zwei ineinander geschachtelte Schleifen über alle Elemente zu bauen (um beispielsweise jedes Element mit jedem anderen zu kombinieren), noch ist es möglich, in zwei Threads eines Programms gleichzeitig Schleifen über die Elemente zu bauen.
      - Für jeden Iterationsschritt sind viele Methodenaufrufe nötig; deshalb nicht so (laufzeit-) effizient wie die Möglichkeiten wie (c) und (d).

      5

   (b) **Enumerator, Iterator**: Der Container definiert einen zusätzlichen Datentyp, der ein aktuelles Element enthält und die oben genannten Methoden anbietet. Die Schleife über alle Objekte liegt ebenfalls im Code des Anwenders des Containers;

die Schleifensteuerung erfolgt mit den Methoden des Iterators.

Der Vorteil gegenüber (a) liegt darin, daß beliebig viele Iteratoren voneinander unabhängig genutzt werden können (und in jedem eine aktuelle Position enthalten ist). Dadurch sind geschachtelte Schleifen oder Multithreading möglich.

Die je nach interner Darstellung der Daten leichte oder schwierige Implementation bleibt wie bei (a).

(c) ***call back*-Funktion**: Um alle Elemente zu durchlaufen, ruft der Anwender einmalig eine Methode des Containers auf, und übergibt als Parameter eine Funktion, die vom Container einmal je Datenelement aufgerufen wird. Diese Funktion erhält eines der Datenelemente als Parameter und verarbeitet es nach den Wünschen des Anwenders (was bei (a) und (b) also im Schleifenrumpf steht, landet hier im Rumpf der Funktion).

Vor-/Nachteile:

- \- Nicht mit allen Sprachen möglich (Pascal, Fortran lassen Unterprogramme als Parameter zu; C und C++ verwenden dafür Zeiger auf Funktionen, die als Parameter verwendet werden können; Java hat dafür keine Möglichkeit)
- \+ sehr laufzeiteffektiv, da wenige Funktionsuafrufe nötig sind, und von Compilern gut optimierbar
- \+ bei korrekter Implementierung ebenso wie Multithreading-tauglich, ebenso sind ineinander geschachtelte Schleifen möglich (wenn auch nur mit etwas Aufwand: es sind zwei *call back*-Funktionen nötig)
- \+ Für alle Arten von Containern möglich, auch bei Bäumen durch einen rekursiven Druchlauf (beispielsweise inorder-Durchlauf).
- \+ für den Anwender leicht zu schreiben, da er keine Schleife bauen muß; die Schleife liegt im Container.

(d) **Funktionsobjekte**: Sehr ähnlich wie (c), aber anstatt eine Funktion als Parameter zu übergeben, wird ein Objekt erzeugt (Funktionsobjekt), das eine passende Methode trägt.

Ansonsten wie bei (c): der Container durchläuft alle Elemente und ruft für jedes Datenelement genau einmal die Methode des übergebenen Funktionsobjekts auf. 5

Vorteile wie bei (c), aber auch in Java möglich. Zudem kann ein Funktionsobjekt noch mehr bieten, indem es neben der aufzurufenden Methode noch Daten enthalten kann, um von Aufruf zu Aufruf beispielsweise etwas zu addieren oder ähnliches. 5

Für alle vier Arten jeweils 5 Punkte für verständliche Kurzbeschreibung, und 5 für die Vor- und Nachteile.

Einiges, was hier möglicherweise fehlt, kann ausgeglichen werden, wenn es in der nächsten Aufgabe steht (und umgekehrt).

Summe: 40

6. Angenommen, Sie haben eine Containerklasse für *offenes Hashing* beliebiger Datentypen erstellt (in Java vermutlich für `Object` oder davon abgeleitete, in C für `void*`-Daten, oder in C++ als `template` eines beliebigen Typs T), mit den üblichen Möglichkeiten des Einfügens, Suchens und Löschens.

   Beschreiben Sie bitte nachvollziehbar (in Worten, oder als Programmcode), welche **Erweiterungen** Sie vornehmen müssen, um die von Ihnen genannten Möglichkeiten zur Iteration aus der vorigen Aufgabe umzusetzen (natürlich nicht alle gleichzeitig, sondern nacheinander).

   Die Klasse für die Hashtabelle ist nicht zu schreiben, sondern nur soweit zu erklären, wie man es für das Verständnis der Erweiterungen in dieser Aufgabe benötigt.

Wichtig ist hier, daß sich die Cursorinformation aus zwei Teilen zusammensetzt; aktuelle Tabellenposition, und Position in der dortigen Liste!

   (a) **Cursor**: Der Container muß sich die aktuelle Position des Cursors merken. Diese besteht bei einer offenen Hashtabelle aus zwei Angaben:
      i. die aktuelle Tabellenposition (in welcher Liste sind wir gerade?)
      ii. Die Position innerhalb der Liste (entweder als Zahl, also die Nummer des Listenelements, oder ein Cursor bzw. Iterator, falls die verwendete Liste das anbietet)

      Also muß die Hashtabelle für den Cursor zwei Werte speichern: Tabellenposition und Position in der Liste.
      (Anmerkung: für eine Hastabelle, über deren Elemente iteriert werden soll, ist es effizient, keine Liste für jeden Tabellenplatz zu nehmen, sondern stattdessen einen Vector bzw. ein Array. Die Tabellenposition wird damit zu einer ganzen nichtnegativen Zahl, und der Zugriff auf das Listenelement ist leicht möglich über den Index.)
      Folgende Methoden bietet die Hashtabelle nach außen an:
      i. *Cursor an den Anfang setzen*: Tabellenposition des Cursor auf 0 setzen, die Position in der Liste ebenfalls zu 0 setzen (bzw. den Iterator der Liste zurücksetzen, falls vorhanden)
      ii. *Cursor um ein Element weitersetzen*: Falls möglich die Listenposition um eins weitersetzen.
      Falls man schon am letzten Listenelement war: Tabellenposition bis zur nächsten nichtleeren Liste erhöhen und Listenposition wieder auf den Anfang setzen.

iii. *aktuelles Element liefern*: Wenn mit der vorigen Methode schon hinter das letzte Element iteriert wurde (oder die ganze Hashtabelle leer ist), dann Fehler melden oder Ausnahme werfen. Ansonsten wird das aktuelle Element der Liste an der Tabellenposition geliefert.

iv. Analog zur letzten Methode melden, ob schon hinter letztem Element

(b) **Enumerator, Iterator**: Wie bei (a), aber alle Informationen (Tabellenposition, Listenposition) stecken in einem eigenen Datentyp (Unterklasse der Hashtabelle), und diese Unterklasse hat genau dieselben Methoden

(c) ***call back*-Funktion**: Die Hastabelle hat eine Methode, die eine Funktion als Parameter bekommt (oder in C/C++: einen Zeiger auf eine Funktion); diese Funktion wiederum kann ein Datenelement entgegennehmen.

Die Methode der Hastabelle enthält zwei ineinander geschachtelte Schleifen: die äußere iteriert über alle Tabellenpositionen, die innere über alle Elemente der zugehörigen Liste.

In der inneren Schleife wird die übergebene Funktion jeweils mit dem aktuellen Datenelement aufgerufen.

(d) **Funktionsobjekte**: Wie bei (c), aber es wird keine Funktion übergeben, sondern ein Objekt mit einer vereinbarten Methode (z.B. `tuwas()`).

In der inneren Schleife (siehe (c)) wird diese Methode mit dem aktuellen Datenelement aufgerufen.

Jede der vier Möglichkeiten: 8 Punkte

Summe: 32