Fr., 5.11.2004 Informatik 13 M (Kr)

**1. Klausur 13/I**

Dauer: 4 Schulstunden Name: __www.r-krell.de____

Hilfsmittel: normaler Taschenrechner,

* *Achte auf sorgfältige Darstellung mit vollständigem, nachvollziehbarem Lösungsweg!* *
* *Kommentiere deine Programme!* *

**1** Sortierbaum (für Ganzzahlen)

Anders als im Unterricht sollen nicht beliebige Objekte (mit Sortierung) in den Baum: Jetzt werden immer nur ganze Zahlen gespeichert, d.h. der *inhalt* in jedem Knoten sei stets vom Typ *int*. Eine Methode *vergleicheMit* ist unnötig, weil <, >=, ... reichen. Keine Vererbung!

a) Baue von Hand den Sortierbaum auf (die Zahlen kommen nacheinander in einen vorher leeren Baum, wobei gleiche Zahlen nach rechts sollen): 13 - 16 - 15 - 24 - 30 - 24 - 16 - 24

b) Statt z.B. die 16 mehrfach im Baum zu speichern, wäre es auch möglich, sie nur einmal zu speichern und sich im einzigen Knoten mit dem Inhalt 16 zusätzlich *anzahl*=2 zu merken.

b1) Zeichne wieder von Hand für die Zahlen aus a) einen solchen Baum mit Anzahlen!

b2) Schreibe in Java die Klasse *AnzKnoten*, wenn jeder Knoten für einen Sortierbaum eine ganze Zahl und deren Anzahl aufnehmen kann. Der Konstruktor soll einen neuen Knoten für eine (erstmals aufgetretene) übergebene Zahl anlegen.

b3) Schreibe passend den Anfang der Klasse *AnzBaum* mit allen benötigten globalen Variablen (Datenfeldern), einem Konstruktor für einen neuen leeren Baum und den Methoden *istLeer* und *sortRein* (wobei letztere natürlich mit der *anzahl* arbeiten soll)

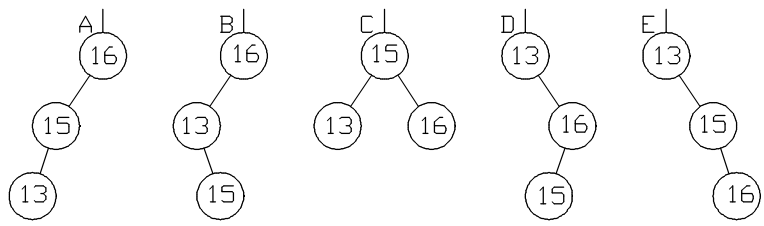
b4) Statt *zeigeGesucht* ist beim Baum mit Anzahl die Methode *public int wieOft (int gesuchterInhalt)* sinnvoller, die sagt, wie oft die gesuchte Zahl als Inhalt im Baum steht – wenn sie nicht vorkommt oder der Baum leer ist, ist das Ergebnis 0. Ergänze die Klasse aus b3) um die Methode *wieOft* (kommentierter Java-Text).

b5) Die Methode *amMeisten* soll die größte *anzahl* im Baum finden und nennen (im Beispiel aus b1) wäre das Ergebnis 3, weil die 24 drei Mal vorkommt). Schreibe in Java und beschreibe auch die Reihenfolge, in der die Knoten besucht werden.

c) Jetzt soll von Hand mit den Zahlen aus a) ein AVL-Baum aufgebaut werden (ohne Anzahlen: wieder kommen gleiche Zahlen nach rechts). Schreibe die Balance auf, die einen Ausgleichsoperation erfordert, zeichne danach neu und nenne die Operation (LL, LR, RL oder RR).

d) Erkläre, wozu AVL-Operationen durchführt werden und nenne den Aufwand der üblichen Baum-Methoden *istLeer*, *sortRein* und *zeigeGesucht* jeweils in Abhängigkeit von der Gesamtzahl *n* der vorhandenen Knoten bei (1) einem vollständig ausgewogenen Baum, (2) bei einem AVL-Baum und (3) im schlimmsten Fall (und: wie sieht ein solcher schlimmster Baum aus? Gibt es nur eine oder mehrere schlimmste Formen?)

e) Offenbar gibt es mindestens fünf verschiedene Arten, drei Ganzzahlen sortiert in einem Baum anzuordnen.




e1) Wird im Baum A fünf Mal nach der Zahl 13 gefragt (jeweils drei Schritte bis zum Auffinden), einmal nach der 15 (zwei Schritte) und drei Mal nach der 16 (ein Schritt), so sind insgesamt 5·3 + 1·2 + 3·1 = 20 Schritte nötig. Bestimme ebenso für die Bäume B bis E die Gesamtschrittzahlen, wenn 13 immer fünf Mal, 15 immer ein Mal und 16 immer drei Mal gesucht wird. Welches ist der günstigste Baum?

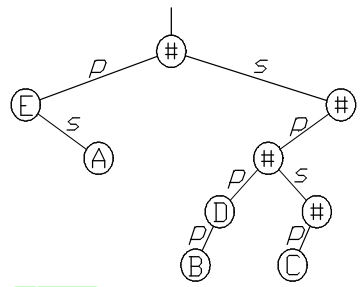
e2) Überlege: ist es nach e1) noch sinnvoll, einen Baum im Supermarkt, in dem die Waren sortiert nach der Artikelnummer gespeichert sind, auszugleichen?

**2** Morsealphabet/Morsebaum

Beim Morsealphabet werden die Buchstaben durch Zeichenfolgen aus Punkten und Strichen kodiert, z.B. 'A' ="."-", 'B'="-…", 'C'="-.-.", D="-..", 'E'="." und 'F'="..-." – siehe Klasse *MorseZchn*!

```
public class MorseZchn
{
  char buchstabe;      // z.B. 'A'
  String morsecode;  //z.B. ".-"
}
```

Um Morsetext schnell in Klartext zu verwandeln, wird ein Baum benutzt, bei dem die 26 Morsezeichen nach dem Morsecode sortiert sind (Objekt *mBaum* der Klasse *MorseBaum;* das Bild zeigt *mBaum* nach Eintrag der ersten fünf Buchstaben): Jeder Punkt (*p*) im Code führt zum linken Nachfolger, jeder Strich (*s*) zum rechten Nachfolger. Manche Knoten sind (noch) nicht mit einem gültigen Morsezeichen besetzt (hier mit # symbolisiert). So bleibt z.B. die Wurzel ohne Eintrag: ganz ohne Punkt oder Strich gibt's kein Morsezeichen. Und als anfangs das 'A' eingetragen wurde, mussten die Wurzel und der nachher mit 'E' gefüllte Knoten mit '#' angelegt werden. (Hinweis: *"abcd".charAt(2)* liefert den Buchstaben 'c' [die Nummerierung beginnt mit 0], während *"abcd".substring(1)* den (Teil-)String "bcd" ab dem ersten Buchstaben zurück gibt.)




a) Der nach Code sortierte Baum soll mit nebenstehender Methode aus einer alphabetischen Liste (wie oben angefangen) vollständig aufgebaut werden. Schreibe die zugehörige Methode *sortRein*, die ein *mz* in den Baum einfügt (und dazu auch evtl. benötigte Zwischenknoten mit # einfügt oder bisherige Zwischenknoten mit einem gültigen Zeichen füllt oder einen neuen Knoten für das Morsezeichen *mz* anlegt)! Schreibe außerdem die verwendete Klasse *MorseKnoten*.

```
public void baueAufAus (MorseListe liste)
{ MorseZchn  mz;
  liste.anDenAnfang();
  while (! liste.istAmEnde())
  { mz = liste.zeige(); liste.weiter();
    sortRein (mz); }
}
```

b) Schreibe (für die Klasse *MorseBaum*) eine Methode *entschlüssele*, die z.B. beim Aufruf von *entschlüssele("-.-.")* den Buchstaben 'C' zurück gibt. Der Morsebaum sei vorher aufgebaut.

c) *mBaum* ist nicht vollständig ausgewogen. Während ein ausgewogener Baum beim Entschlüsseln zufällig erzeugter sinnloser Zeichenkombinationen vorteilhafter wäre, bewährt sich der *mBaum* (wie oben im Anfangsstadium dargestellt) beim Entschlüsseln einer aus normalem deutschen oder englischen Text erzeugten Morsenachricht. Erkläre!

d) Ist der gezeigte Morsebaum auch zum Verschlüsseln geeignet, d.h. z.B. um mit einem Methoden-Aufruf wie *verschlüssele('A')* den Code ".-" zu erhalten? Erkläre das grundsätzliche Vorgehen in deutsch (kein Java-Text) und überlege eventuell bessere Alternativen.

**3** Scanner, Compiler und Rechenbaum

Differenzen mit positiven Ganzzahlen und Klammern sollen erkannt und berechnet werden. Erlaubte Eingaben sind beispielsweise "1457 – 78 – 406" oder auch "870 – ((769 – 34) – 6)". Mindestens zwei bis theoretisch beliebig viele Zahlen sollen durch Minuszeichen verbunden werden. Zwischen den Zahlen und dem Rechenzeichen (nur '–' ist erlaubt!) oder einer Klammer braucht kein, dürfen aber beliebig viele Leerzeichen ' ' stehen.

a) Schreibe eine Klasse *DiffScanner* mit dem Konstruktor *DiffScanner (String eingabezeile)* und der Methode *nächstesToken()*. Die Methode *nächstesToken()* soll bei jedem Aufruf eine (die nächste) Sinneinheit aus der Eingabezeile als String zurück geben (also entweder alle Ziffern einer Zahl, eine Klammer oder das Minuszeichen). Ein Automatengraph kann helfen. Kommentiere!

b) Zeichne ein Syntaxdiagramm für erlaubte Differenzen. Tipp: Klammern stellen letztlich Zahlen dar (nämlich das Ergebnis der Differenz in Klammern) und sollten daher auch wie Zahlen behandelt werden! Sinnlose Klammern sollen nicht erlaubt sein.

c) Im Unterricht hatten wir aus dem Syntaxdiagramm noch keinen Compiler entwickelt. Deshalb

soll die jeweils eingegebene Differenz noch in einem Rechenbaum dargestellt werden, in dem nur Minuszeichen und Zahlen stehen, keine Klammern (die Klammerung wirkt sich allerdings auf die Form des Baumes aus).

c1) Zeichne für die beiden Beispiel-Eingaben jeweils den Rechenbaum von Hand und berechne das Ergebnis

c2) Notiere in Deutsch bzw. in Tabellenform, wann das nächste vom Scanner erhaltene Token als neue Wurzel in den Rechnebaum eingebaut werden muss bzw. wo es sonst hinkommt (ggf. unter Veränderung des bisherigen Baums). Skizziere ggf. Beispiele möglicher Fälle!

c3) Angenommen, ein Baum ist fertig (z.B. wie in c1) dargestellt). Wie muss er durchlaufen werden, um das richtige Ergebnis zu liefern? Wird die ursprüngliche Klammerung richtig berücksichtigt? (In Deutsch; nur falls Zeit, auch in Java. Hinweis für Java: *int x = Integer.parseInt ("1457");* macht aus dem String "1457" die rechenbare Zahl 1457!)