Universität der Bundeswehr
Fakultät für Informatik – Institut 2
Prof. Dr. Wolfgang Hommel
Michael Grabatin, Michael Steinke

WT 2020
Übungsblatt 5

Einführung in die Informatik II

14. und 17.02.2020

1 Listen

In der Vorlesung sind bereits einige Funktionen für Listen, die aus der funktionalen Programmierung bekannt sind, nachgebildet worden. In dieser Aufgabe sollen zusätzliche Funktionen für einfach verkettete Listen implementiert werden. Folgende Typdefinitionen sind gegeben:

```
1 class Elem(var value : Int = 0, var rest : Elem = null)
```

Des Weiteren können die Funktionen `cons` und `append` aus der Vorlesung (vgl. K3-28) als gegeben betrachtet werden. Zudem ist die Funktion `copy` gegeben, die eine Kopie einer Liste erstellt.

a) Implementieren Sie zunächst eine Funktion `mkString`, die eine Liste in einen String umwandelt. In Anlehnung an die gleichnamige Methode für Arrays und Scala-Listen soll sie folgender Definition entsprechen:

```
1 def mkString(list : Elem, l : String,
2   m : String, r: String) : String = ...
```

Zurückgegeben werden soll ein String, der mit `l` beginnt, mit `r` endet und zwischen je zwei Elementen ein `m` vorsieht.

Zur Erinnerung: `List(1, 2, 3).mkString("(", ", ", ")")` liefert `"(1, 2, 3)"`.

b) Implementieren Sie eine Funktion `intersect`, welche zu zwei gegebenen aufsteigend sortierten Listen, die „Schnittliste“ der gemeinsamen Elemente (ebenfalls aufsteigend sortiert) zurückliefert. Die übergebenen Listen sollen unverändert bleiben.

c) Implementieren Sie eine Funktion, die die Schnittliste dreier Listen bestimmt, ohne die Funktion aus der vorherigen Teilaufgabe zu verwenden.

d) Implementieren Sie auf geeignete Weise folgende Funktionen, die den gleichnamigen Methoden für Scala-Listen entsprechen:

```
1 head, tail, init, last, length, take, drop
```

Sehen Sie bei ungültig übergebenen Argumenten das Werfen entsprechender `Exceptions` vor.

2 Binärbäume – Isokline

Für die Knoten `N` eines nichtleeren Binärbaums mit Wurzel `R` sind natürliche Zahlen `height(N)` und `depth(N)` wie folgt festgelegt:

- `depth(N)` ist die Länge des direkten Kantenzugs von `R` nach `N` (angegeben in Anzahl der Kanten).
- `height(N)` ist die Höhe des Teilbaums B mit der Wurzel `N`. Die Höhe eines Binärbaums ist die Länge des längsten direkten Kantenzugs von seiner Wurzel zu einem seiner Blätter (ebenfalls in Anzahl der Kanten gerechnet).

Gegeben ist folgende Definition:

```
1 class Tree(var height : Int = 0, var depth : Int = 0,
2            var left : Tree = null, var right : Tree = null,
3            var iso : Tree = null)
```

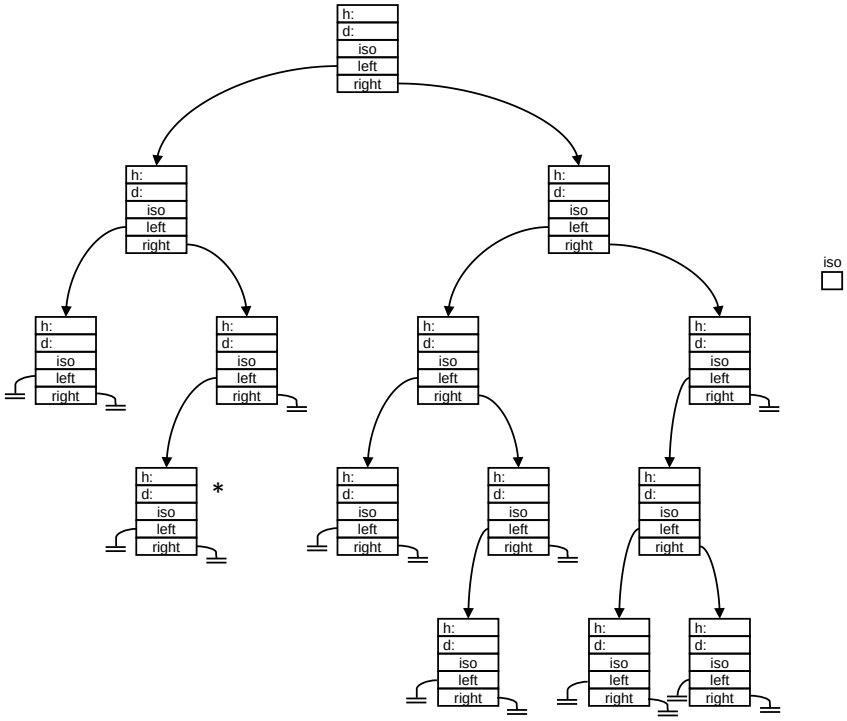
a) In der folgenden Abbildung finden Sie einen Binärbaum, dessen Knoten dieser Definition entsprechen (die einzelnen Felder sind durch ihre Anfangsbuchstaben identifiziert).

Tragen sie dort in jedem Knoten seine Höhe `h` und Tiefe `d` ein.

Verbinden Sie zusätzlich die Knoten `N`, die der *Isoklinenbedingung*

```
1 height(N) = depth(N)
```

genügen, über die Felder `iso` von *rechts nach links* zu einer linearen Liste. Der Zeiger `iso` soll auf den ersten Knoten dieser Liste zeigen.



Wie ändert sich das Ergebnis, wenn der mit $\star$ markierte Knoten entfernt wird?

b) Ergänzen Sie die fehlenden Teile der unten skizzierten Scala-Prozedur `updateTree` so, dass der Aufruf `updateTree(R,0)` in allen Knoten die Höhen und Tiefen einträgt, sowie die Knoten die der Isoklinenbedingung genügen zu einer Liste verbindet (wie in Teilaufgabe (a)). Der Zeiger `iso : Tree` befindet sich im selben Block wie die Prozedur.

```
1  //Isoklinenliste
2  var iso : Tree = null
3
4  def updateTree (node : Tree, depth : Int) : Unit = {
5    //node auf null pruefen
6    //Tiefe von node eintragen
7    //rekursive Aufrufe
8    //Hoehe von node berechnen und eintragen
9    //node ggfs. in Isoklinenliste eintragen
10 }
```

3 Tries

Tries sind Bäume, in denen Wortmengen zeichenweise gespeichert sind. Zeichen stehen an den Kanten. Die Wörter teilen sich die gemeinsamen Anfangsstücke. So beginnen beispielsweise die Wörter `zauber` und `zahl` beide mit dem Pfad `z-a`.

Verwenden Sie die folgenden Funktionen und Definitionen:

```
1  //Anzahl der Zeichen: 26
2  val cAlph = 'z' - 'a' + 1
3
4  //Alphabet als Array von Char 'a', ..., 'z', 'A', ..., 'Z'
5  val alphabet = Array.tabulate(2 * cAlph)(i =>
6      if (i < cAlph) i + 'a'
7      else i - cAlph + 'A').map(_.toChar)
8
9  //Zeichen als idx von Alphabet
10 def char2Idx(c : Char) : Int = {
11   if (c >= 'a' && c <= 'z') c - 'a'
12   else if (c >= 'A' && c <= 'Z') c - 'A' + cAlph
13   else throw new IllegalArgumentException("'" + c + "' is invalid!")
14 }
15
16 //Idx in Zeichen von Alphabet
17 def idx2Char(idx : Int) : Char = alphabet(idx)
18
19 class TrieNode(
20   var sons : Array[TrieNode] = Array.fill(2 * cAlph)(null),
21   var cWords : Int = 0)
22
23 def createTrie = new TrieNode
```

Beachten Sie, dass `cWords` nicht (wie in der Vorlesung) vom Typ `Boolean`, sondern vom Typ `Int` ist. Dieses speichert die Anzahl der Worte, die auf diesem Knoten enden.

In `sons` werden alle Söhne eines Knotens gespeichert.

a) Schreiben Sie folgende Prozeduren:

- Zum Hinzufügen eines neuen Wortes in einen Trie: `def addTrie(tr: TrieNode, str: String): Unit = ???`
- Zum Einfügen eines oder mehrerer, durch Leerzeichen getrennten, Worte in einen Trie unter Verwendung der `addTrie` Funktion: `def fillTrie(tr: TrieNode, str: String): Unit = ???`
  Mit der Funktion `str.split("")` kann ein String in ein Array von, durch den Parameter angegebenen Zeichen getrennten, Teilstrings aufgeteilt werden.

b) Ergänzen Sie Ihr Programm um die Funktion `isElement`, die **`true`** zurückliefert, falls ein Wort im Trie enthalten ist.

c) Schreiben Sie eine Prozedur `printTrie`, mit der ein Trie in folgendem Format ausgegeben werden kann:

```
a
 u
  t
   o ( 1 )
z
 a
```

```
h
 l ( 2 )
 n ( 1 )
u
 b
  e
   r ( 2 )
```

Der Trie wurde mit `fillTrie(testTrie, "zauber zahl zahl zahn zauber auto")` befüllt.