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)"`.

> Eine mögliche Implementierung sieht wie folgt aus:
>
> ```
> 1  def mkString(list : Elem, l : String, m : String,
> 2    r: String) : String = {
> 3    var p = list
> 4    var rtn = ""
> 5    if (l != null) rtn = l
> 6    while (p != null) {
> 7      rtn += p.value
> 8      p = p.rest
> 9
> 10     if (p != null) rtn += m
> 11   }
> 12   return if (r != null) rtn + r else rtn
> 13 }
> ```

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.

> Unter Verwendung eines Dummy-Elements lässt sich folgende Implementierung verwenden:
>
> ```
> 1 def intersect(l1 : Elem, l2 : Elem) : Elem = {
> 2   var dummy = new Elem
> ```

```
  var p1 = l1
  var p2 = l2
  var p = dummy

  while (p1 != null && p2 != null) {
    if (p1.value < p2.value) p1 = p1.rest
    else if (p1.value > p2.value) p2 = p2.rest
    else {
      p.rest = new Elem(p1.value)
      p = p.rest
      p1 = p1.rest
      p2 = p2.rest
    }
  }

  return dummy.rest
}
```

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

Soll die Verwendung von `intersect` aus der vorherigen Teilaufgabe vermieden werden, sind die drei Listen parallel zu durchlaufen. In folgender Implementierung ist je ein Zeiger pro Liste vorgesehen. In jedem Schleifendurchlauf wird sichergestellt, dass (sofern nicht alle Zeiger auf ein gleich großes Element zeigen) genau ein Zeiger verändert wird. Dabei wird der erste gefundene Zeiger „erhöht“, der auf das „nichtgrößte“ Element zeigt. Existiert kein solcher Zeiger müssen alle zu betrachtenden Elemente gleich groß sein und es werden alle Zeiger angepasst:

```
def intersect3(l1 : Elem, l2 : Elem, l3 : Elem) : Elem = {
  var dummy = new Elem
  var p1 = l1
  var p2 = l2
  var p3 = l3
  var p = dummy

  while (p1 != null && p2 != null && p3 != null) {
    if (p1.value < p2.value || p1.value < p3.value) p1 = p1.rest
    else if (p2.value < p1.value || p2.value < p3.value) p2 = p2.
        rest
    else if (p3.value < p1.value || p3.value < p2.value) p3 = p3.
        rest
    else {
      p.rest = new Elem(p1.value)
      p = p.rest
      p1 = p1.rest
      p2 = p2.rest
      p3 = p3.rest
    }
  }
  return dummy.rest
}
```

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.

Zunächst definieren wir eine Prozedur, die im Falle einer leeren Liste eine `IllegalArgumentException` wirft:

```
1  def throwIfEmpty(l : Elem) = {
2    if (l == null) throw new
3      IllegalArgumentException("List must not be empty!")
4  }
```

Die Funktionen `head` und `tail`:

```
1  def head(l : Elem) : Int = {
2    throwIfEmpty(l)
3    return l.value
4  }
5
6  def tail(l : Elem) : Elem = {
7    throwIfEmpty(l)
8    return copy(l.rest)
9  }
```

Die Funktion `last` liefert das letzte, die Funktion `init` alle Elemente außer dem letzten Element. Auch hier wird bei leeren Listen eine `IllegalArgumentException` geworfen.

```
1  def init(l : Elem) : Elem = {
2    throwIfEmpty(l)
3    var dummy = new Elem
4    var pl = l
5    var pr = dummy
6
7    while(pl.rest != null) {
8      pr.rest = new Elem(pl.value)
9      pr = pr.rest; pl = pl.rest
10   }
11   return dummy.rest
12 }
13
14 def last(l : Elem) : Int = {
15   throwIfEmpty(l)
16   var p = l
17   while (p.rest != null) p = p.rest
18
19   return p.value
20 }
```

Die Funktion `length` zur Bestimmung der Länge einer Liste:

```
1  def length(l : Elem) : Int = {
2    var n = 0
3    var p = l
4    while (p != null) {
5      p = p.rest
6      n += 1
```

```
  }
  return n
}
```

Abschließend die Funktionen, welche die ersten n Elemente einer Liste (`take`) zurückliefert bzw. eben diese entfernt und die restlichen zurückgibt (`drop`):

```
def take(l : Elem, n : Int) : Elem = {
  var count = 0
  var dummy = new Elem
  var pl = l
  var pr = dummy

  while(count < n && pl != null) {
    pr.rest = new Elem(pl.value)
    pr = pr.rest
    pl = pl.rest
    count += 1
  }
  return dummy.rest
}

def drop(l : Elem, n : Int) : Elem = {
  var count = 0
  var pl = l

  while(count < n && pl != null) {
    pl = pl.rest
    count += 1
  }
  return copy(pl)
}
```

Hinweis: Mithilfe von `length` ließe sich die Funktion `init` auch wie folgt implementieren:

```
def init2(l : Elem) : Elem = take(l : Elem, length(l) - 1)
```

Allerdings würde dann die Liste jeweils zweimal durchlaufen, was annähernd doppelt so viel Aufwand als bei der direkten Implementierung bedeutet.

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)
```

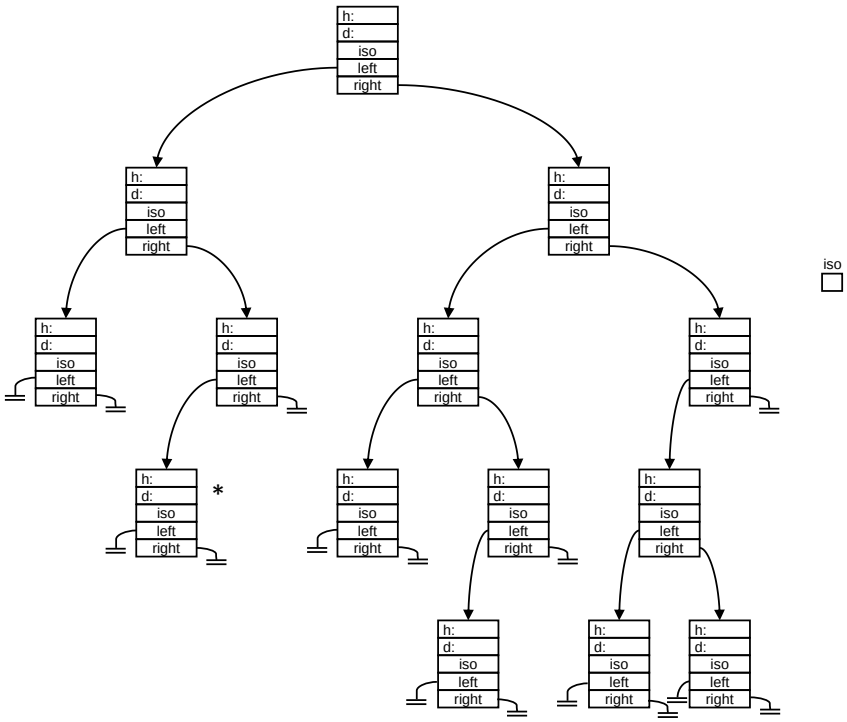
a) In der folgenden Abbildung finden Sie einen Binärbaum, dessen Knoten dieser Definition entsprechen (die einzelnen Felder sind durch ihre Anfangsbuchstaben identifiziert).

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

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

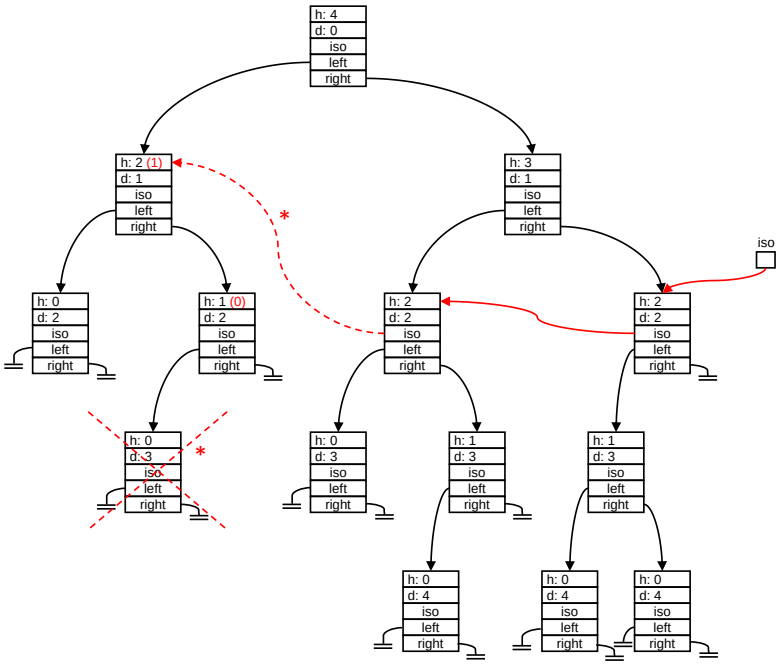
```
1 height(N) = depth(N)
```

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?

Die Lösung ist wie folgt (in Klammern dargestellte Angaben sowie die gestrichelten Linien stellen die Änderungen dar, die sich durch das Entfernen des mit $\star$ markierten Knotens ergeben):



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 }
```

Da `iso` am Ende eine einfach verkettete Liste repräsentieren soll, in der alle Knoten, welche die Isoklinenbedingung erfüllen, von rechts nach links aufgezählt werden, ist es entscheidend, in welcher Reihenfolge die Knoten durchlaufen werden. Damit der Kopf der Liste am Ende auf den richtigen Knoten zeigt, ist die Liste von links nach rechts aufzubauen und der jeweils zuletzt gefundene Knoten vorne anzuhängen. Daher ist im Zuge der Rekursion sicherzustellen, dass auch immer die linken Teilbäume zuerst durchlaufen werden: Eine mögliche Implementierung ist die folgende:

```
1  def updateTree (node : Tree, depth : Int) : Unit = {
2    //node auf null pruefen
3    if (node == null) return
```

```
4
5     //Tiefe von node eintragen:
6     node.depth = depth
7
8     //rekursive Aufrufe:
9     updateTree(node.left, depth + 1)  //Iso von links nach rechts
          aufbauen
10    updateTree(node.right, depth + 1)
11
12    //Hoehe von node berechnen und eintragen:
13    if (node.left != null) node.height = node.left.height + 1
14    if (node.right != null && node.right.height + 1 > node.height)
          {
15      node.height = node.right.height + 1
16    }
17
18    //node ggfs. in Isoklinenliste eintragen:
19    if (node.height == depth) {
20      node.iso = iso
21      iso = node
22    }
23  }
```

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.

> Eine mögliche Implementierung wäre die folgende:
>
> ```
> 1  def addTrie(tr: TrieNode, str: String): Unit = {
> 2    var cur = tr
> 3    for (c <- str) {
> 4      val idx = char2Idx(c)
> 5
> 6      //Sohn bestimmen bzw. wenn noetig erstellen
> 7      if(cur.sons(idx) == null) {
> 8        cur.sons(idx) = createTrie
> ```

```
    }
    cur = cur.sons(idx)
  }
  //Wort zu Ende
  cur.cWords += 1
}
```

Mit Verwendung der Prozedur `addTrie` ergibt sich folgende Implementierung:

```
def fillTrie(tr : TrieNode, str : String) : Unit = {
  for (s <- str.split(" ")) addTrie(tr, s)
}
```

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

Die Funktion `isElement`:

```
def isElement(tr : TrieNode, str : String) : Boolean = {
  var cur = tr
  for (c <- str; if cur != null) {
    cur = cur.sons(char2Idx(c))
  }
  return cur != null && cur.cWords > 0
}
```

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.

Die Prozedur `printTrie` kann folgendermaßen realisiert werden:

```
def printTrie(tr : TrieNode) : Unit = {
  def pt(tr : TrieNode, n : Int) : Unit = {
    for (idx <- 0 to tr.sons.length - 1) {
      val son = tr.sons(idx)
      if (son != null) {
        val wc = if (son.cWords > 0) " (" + son.cWords + ")"
                 else ""
        println(String.format("%" + n + "s%s",
          idx2Char(idx).toString, wc))
        pt(son, n + 1)
```

```
        }
      }
    }
    pt(tr, 1)
}
```