TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Lehrstuhl für Sprachen und Beschreibungsstrukturen
Einführung in die Informatik 2
Prof. Dr. Helmut Seidl, T. M. Gawlitza, S. Pott, M. Schwarz

WS 2008/09
Übungsblatt 9
09.12.2008

Abgabe: 16.12.2008 (vor der Vorlesung)

Aufgabe 9.1 (H) Heapsort

In dieser Aufgabe dürfen Sie *Heap-Sort* implementieren. Es sei folgender Datentyp zur Repräsentation binärer Bäume definiert:

```
type 'a btree = Leaf | Node of int*'a*'a btree*'a btree
```

Ein binärer Baum ist also

- entweder ein Blatt
- oder ein innerer Knoten, der aus einem `'a`-Wert, dem Wert des Knotens, und zwei Kind-Bäumen besteht.

Ein binärer Baum ist genau dann ein Heap, falls für jeden Knoten gilt, dass sein Wert größer oder gleich aller Werte in den Teilbäumen ist (Heap-Eigenschaft). Insbesondere ist der größte Wert in der Wurzel gespeichert. Zusätzlich wird gefordert, dass der in dem Knoten gespeicherte Integer-Wert die Länge des kürzesten Pfades zu einem Blatt ist. Beispielsweise ist

```
Node(1,"c",Node(1,"a",Leaf,Leaf),Leaf)
```

ein Heap.

a) Schreiben Sie eine Funktion `insert : 'a -> 'a btree -> 'a btree`, so dass der Aufruf `insert x h` den Wert `x` in den Heap `h` einfügt. Dabei soll sowohl darauf geachtet werden die Heap-Eigenschaft nicht zu verletzen als auch die Tiefe des Heaps, falls möglich, nicht zu erhöhen. Ein Wert `x` wird wie folgt in den Heap eingefügt: Zunächst wird `x` mit dem Wert `w` an der Wurzel des Heaps verglichen. Der größere der beiden Werte wird der neue Wert der Wurzel, während der kleinere in einen Teil-Heap eingefügt wird. Um eine Degeneration zu verhindern ist dabei derjenige Teil-Heap auszuwählen, der den kürzesten Pfad zu einem Blatt hat. Beispielsweise soll der Aufruf

   ```
   insert "b" (Node(1,"c",Node(1,"a",Leaf,Leaf),Leaf))
   ```

   den Heap

   ```
   Node(2,"c",Node(1,"a",Leaf,Leaf),Node(1,"b",Leaf,Leaf))
   ```

   zurück liefern. Wichtig dabei ist, dass das Element `"b"` in den kleineren Teil-Heap eingefügt worden ist.

b) Schreiben Sie eine Funktion `remove : 'a btree -> 'a * 'a btree`, so dass der Aufruf `remove heap` ein Paar bestehen aus dem Wert der Wurzel und einem aus den verbleibenden Elementen bestehenden Heap liefert. Dazu kann es sinnvoll sein, eine Funktion zu schreiben, die zwei Heaps zu einem vereinigt. Beispielsweise soll sich der Aufruf

```
remove(Node(2,"c",Node(1,"a",Leaf,Leaf),Node(1,"b",Leaf,Leaf)))
```

zu dem Tupel

```
("c", Node(1,"b",Node(1,"a",Leaf,Leaf),Leaf))
```

auswerten.

c) Verwenden Sie Ihre Implementierung zur Realisierung des Heap-Sort-Algorithmus.

d) Sortieren Sie die Liste $[[3;2;4];[3];[3;5]]$. Wieso kommt diese Sortierung zustande?

Lösungsvorschlag 9.1

```
type 'a btree = Leaf | Node of int * 'a * 'a btree * 'a btree

let min_weg = function
  Leaf -> 0
| Node(c,_,_,_) -> c

let rec insert x = function
  Leaf              -> Node(1,x,Leaf,Leaf)
| Node(_,x',t1,t2) ->
    let (s,ns) = if x > x' then (x',x) else (x,x') in
    let (t1,t2) = if min_weg t1 <= min_weg t2 then
              (insert s t1, t2)
            else (t1, insert s t2) in
    Node(1 + min (min_weg t1) (min_weg t2), ns, t1, t2)

exception Empty

let rec repair t1 t2 =
  match (t1,t2) with
    (Leaf,t) | (t,Leaf) -> t
  | (Node(_,x1,t1',t1''), Node(_,x2,t2',t2'')) ->
      let (x,t1,t2) = if x1 > x2 then
                (x1, repair t1' t1'', t2)
              else (x2, t1, repair t2' t2'') in
      Node(1 + min (min_weg t1) (min_weg t2), x, t1, t2)

let remove = function
  Leaf             -> raise Empty
| Node(_,x,t1,t2) -> (x,repair t1 t2)

open List

let rec build_list acc = function
    Leaf -> acc
  | heap ->
      let (x,heap) = remove heap in
      build_list (x::acc) heap
```

```
let build_list l = build_list [] l

let heapsort l =
  build_list (fold_left (fun heap x -> insert x heap) Leaf l)
```

Aufgabe 9.2 (P) Module und Funktoren

Gegeben sei folgende Signatur:

```
module type Map = sig
  type ('a,'b) t
  val empty : ('a,'b) t
  val update : ('a,'b) t -> 'a -> 'b -> ('a,'b) t
  val get_keys : ('a,'b) t -> 'a list
  val get : ('a,'b) t -> 'a -> 'b option
  val string_of_map :
      ('a -> string) -> ('b -> string) -> ('a,'b) t -> string
end
```

a) Schreiben Sie ein sinnvolles Modul `ListMap` der Signatur `Map` mithilfe von Listen.

b) Implementieren Sie einen Funktor zur Erstellung von *Histogrammen*, der von folgender Signatur ist:

```
functor (M : Map) -> sig val histo : 'a list -> ('a, int) M.t end
```

Ein Histogramm einer Liste `l` ist eine Abbildung, die jedem in `l` vorkommenden Element die Anzahl seiner Vorkommen zuordnet. Beispielsweise ist die Abbildung $\{\text{"a"} \mapsto 2, \text{"b"} \mapsto 3\}$ ein Histogramm der Liste $[\text{"a"};\text{"b"};\text{"a"};\text{"b"};\text{"b"}]$.

Lösungsvorschlag 9.2

```
open List

module type Map = sig
  type ('a,'b) t
  val empty : ('a,'b) t
  val update : ('a,'b) t -> 'a -> 'b -> ('a,'b) t
  val get_keys : ('a,'b) t -> 'a list
  val get : ('a,'b) t -> 'a -> 'b option
  val string_of_map :
      ('a -> string) -> ('b -> string) -> ('a,'b) t -> string
end

module ListMap = struct
  type ('a,'b) t = ('a * 'b) list
  let empty = []
  let update m k v =
    if mem_assoc k m then (k,v)::remove_assoc k m else (k,v)::m
  let get_keys m =
    let (result,_) = split m in
    result
  let get m k =
    if mem_assoc k m then Some(assoc k m) else None
  let string_of_map string_of_key string_of_value m =
    let strings = map (fun (k,v) -> string_of_key k ^
          "␣->␣" ^ string_of_value v) m in
    "{" ^ String.concat ";␣" strings ^ "}"
```

```
end

module Histogramm(M : Map) = struct
  let rec histo his = function
    [] -> his
  | x::xs ->
      match M.get his x with
        None   -> histo (M.update his x 1) xs
      | Some n -> histo (M.update his x (n+1)) xs
  let histo liste = histo M.empty liste
end

module ListHistogramm = Histogramm(ListMap)

open ListHistogramm

let t = histo [2;3;2;5;3;4;2;1;5]
```