Cursul 9 – 10 Plan

- Tipul de dată abstract Set
  - Specificare algebrică
  - Implementare cu liste
  - Implementare cu arbori
  - Arbori echilibrați (AVL)
- Tipul abstract Bag
  - Reprezentare cu arbori heap

Tipuri de date abstracte

- O declarație `data` introduce un nou tip de dată prin descrierea modului de construire a elementelor sale

- Un tip în care sunt descrise valorile fără a descrie operațiile se numeşte ***tip concret***

- Un **tip abstract de date** este definit prin specificarea operațiilor sale fără a descrie modul de reprezentare a valorilor

- În general reprezentarea adt se poate schimba fără a afecta validitatea scripturilor ce utilizează acest adt

Coada

- Operațiile primitive – numele, tipul:
  - `empty   :: Queue a`
  - `join    :: a -> Queue a -> Queue a`
  - `front   :: Queue a -> a`
  - `back    :: Queue a -> Queue a`
  - `isEmpty :: Queue a -> Bool`
- Semnificația

- Lista operațiilor împreună cu tipul acestora reprezintă **signatura tipului de dată abstract**

Coada

- **Specificare algebrică (axiomatică):** o listă de axiome ce trebuie să fie satisfăcute de operații
- Pentru **`Queue a`**:

```
isEmpty empty = True
isEmpty (join x xq) = False
front (join x empty) = x
front (join x(join y xq)) = front (join y xq)
back(join x empty) = empty
back(join x (join y xq)) = join x(back (join y xq))
```

ADT - Implementare

- Implementarea adt înseamnă:
  - furnizarea unei reprezentări pentru valorile sale
  - definirea operațiilor în termenii acestei reprezentări
  - dovedirea faptului că operațiile implementate satisfac specificațiile algebrice
- Cel ce implementează adt este liber în a alege dintre posibilele reprezentări, în funcție de :
  - eficiență
  - simplitate
  - ...gusturi

Module

- Modulul – mecanism pentru definirea unui adt
- Sintaxa:

```
module Nume_modul(Lista_export)where
            Implementare
```

- *Nume _modul* începe cu literă mare
- *Lista_export* conține:
  - Numele tipului abstract de date – acelaşi cu numele modulului
  - Numele operațiilor
- Nici un alt nume sau valoare declarat în modul şi care nu apare în lista export nu poate fi utilizat în altă parte
- Asta înseamnă că implementarea descrisă în modul este ascunsă în orice script ce foloseşte modulul

Mulțimi

- Mulțimile pot fi reprezentate prin:
  - Liste
  - Liste fără elemente duplicate
  - Liste ordonate
  - Arbori
  - Funcții booleene
  - etc.
- Reprezentarea este aleasă în funcție de operațiile ce se folosesc

Mulțimi - signatura

- Operații pe mulțimi:

```
empty    :: Set a
isEmpty  :: Set a -> Bool
member   :: Set a -> a -> Bool
insert   :: a -> Set a -> Set a
delete   :: a -> Set a -> Set a
```

- Un tip bazat pe cele 5 operații: dicționar

```
union :: Set a -> Set a -> Set a
meet  :: Set a -> Set a -> Set a
minus :: Set a -> Set a -> Set a
```

Mulțimi – specificarea algebrică

```
insert x (insert x xs) = insert x xs
insert x (insert y xs) = insert y (insert x xs)

isEmpty empty = True
isEmpty(insert x xs) = False

member empty y = False
member(insert x xs) y = (x == y) `or` member xs y

delete x empty = empty
delete x (insert y xs) = if x == y then delete x xs
                         else insert y (delete x xs)
```

Mulțimi – specificarea algebrică

```
union xs empty = xs
union xs (insert y ys) = insert y (union xs ys)

meet xs empty = empty
meet xs (insert y ys) =
  if member xs y then insert y (meet xs ys)
                 else meet xs ys

minus xs empty = xs
minus xs (insert y ys) = minus (delete y xs) ys
```

Mulțimi – reprezentarea cu liste

- Presupunând că a este instanță a clasei Eq, mulțimile se pot reprezenta cu liste
- Funcția abstr:

```
abstr :: [a] -> Set a
abstr = foldr insert empty
```

- Funcția de validare poate avea două variante:

```
valid xs = True
```

  - În acest caz orice listă reprezintă o mulțime. Operațiile se implementează uşor dar sunt dezavantaje majore

```
valid xs = nonduplicated xs
```

  - Se obțin rezultate mai bune la inserție şi reuniune

Mulțimi – reprezentarea cu liste

- Dacă orice listă este o reprezentare validă atunci:

```
member xs x = some(==x) xs
insert x xs = x:xs
delete x xs = filter(\= x) xs
union xs ys = xs ++ ys
minus xs ys = filter(not.member ys) xs

some :: (a -> Bool) -> [a] -> Bool
some p = or.map p
```

- Avantaj: insert necesită timp constant
- Dezavantaj: lista poate fi mult mai mare decât mulțimea. Dacă n este lungimea reprezentării (listei), delete este de complexitate $\Theta(n)$ iar minus $\Theta(n^2)$, în timp ce mulțimea poate să aibă m elemente cu m << n.

Mulțimi – reprezentarea cu liste

- Dacă se restricționează reprezentarea la liste cu elemente distincte atunci, o primă variantă de implementare:

```
insert x xs = x:filter(\=x)xs
union xs ys = xs ++ filter(not.member xs)ys
```

- Dar complexitatea este mare: $\Theta(n)$ respectiv $\Theta(n^2)$. Dacă presupunem a instanță a lui Ord şi impunem pentru validare liste strict ordonate atunci:

```
member xs x = if null xs then False
                 else(x == head ys)
      where ys = dropWhile (<x) xs
union [] ys = ys
union (x:xs)(y:ys) | (x<y)  = x:union xs (y:ys)
                   | (x==y) = x:union xs ys
                   | (x>y)  = y:union (x:xs)ys
```

Mulțimi – reprezentarea cu arbori

```
module Set(Set, empty, isEmpty, member,insert,delete) where

data Set a = Null | Fork (Set a) a (Set a)
  deriving (Show)

empty :: Set a
empty = Null

isEmpty :: Set a -> Bool
isEmpty Null = True
isEmpty(Fork xt y zt) = False

member :: (Ord a) => Set a -> a -> Bool
member Null x = False
member (Fork xt y zt) x
  | (x < y)  = member xt x
  | (x == y) = True
  | (x > y)  = member zt x
```

Mulțimi – reprezentarea cu arbori

```
insert :: (Ord a) => a -> Set a -> Set a

insert x Null = Fork Null x Null
insert x (Fork xt y zt)
    | (x < y)  = Fork(insert x xt) y zt
    | (x == y) = Fork xt y zt
    | (x > y)  = Fork xt y (insert x zt)

delete :: (Ord a) => a -> Set a -> Set a

delete x Null =  Null
delete x (Fork xt y zt)
    | (x < y)  = Fork(delete x xt) y zt
    | (x == y) = join xt  zt
    | (x > y)  = Fork xt y (delete x zt)

join :: Set a -> Set a -> Set a

join xt yt = if isEmpty yt then xt else Fork xt y zt
        where (y, zt) = splitTree yt

splitTree :: Set a -> (a, Set a)

splitTree(Fork xt y zt) = if isEmpty xt then (y, zt) else (u, Fork vt y zt)
                 where(u, vt) = splitTree xt
```

λ Mulțimi – reprezentarea cu arbori

- Funcțiile join şi splitTree nu pot fi utilizate în scripturi ce importă modulul Set
- Funcția join nu poate fi folosită pentru reuniune: valoarea **`join xt yt`** este definită cu presupunerea că fiecare element din xt este mai mic decât fiecare element din yt
- Reprezentarea mulțimilor prin arbori este eficientă doar pentru operațiile dicționar: inserare, ştergere, căutare

Exemplu de utilizare

```
import Set
toSet :: Ord a => [a] -> Set a
toSet [] = empty
toSet (x:xs) = insert x (toSet xs)
t1, t2, t3 :: Set Int
t1 = toSet [4,1,6]
t2 = toSet [11,9,8]
t3 = empty

Main> toSet [8,3,5,2,1]
Fork Null 1 (Fork Null 2 (Fork (Fork Null 3 Null) 5 (Fork Null 8
   Null)))
Main> insert 7 t1
Fork (Fork Null 1 (Fork Null 4 Null)) 6 (Fork Null 7 Null)
Main> insert 5 t2
Fork (Fork Null 5 Null) 8 (Fork Null 9 (Fork Null 11 Null))
Main> t2
Fork Null 8 (Fork Null 9 (Fork Null 11 Null))
Main> insert 10 t2
Fork Null 8 (Fork Null 9 (Fork (Fork Null 10 Null) 11 Null))
```

Mulțimi – reprezentarea cu arbori

- Probleme la reprezentarea cu arbori:
  - timpul de execuție pentru member, insert, delete depind de înălțimea arborelui
  - o mulțime cu n elemente poate fi reprezentată cu un arbore de înălțime log(n+1)
  - pentru păstrarea relației h = O(log n) - arbori echilibrați (arbori AVL)

Arbori echilibrați (AVL)

- Introduşi de G. Adelson-Velski şi Y. Landis
- Un arbore este AVL dacă înălțimile subarborilor stâng respectiv drept pentru fiecare nod, diferă cu cel mult o unitate
- Formalizare: funcția **`balanced`**

```
balanced :: Stree a -> Bool
balanced Null = True
balanced(Fork xt x yt)= abs(height xt - height yt) <= 1
                     && balanced xt && balanced yt
```

- Arborele xt este echilibrat dacă

  `balanced xt` este **`True`**

Arbori echilibrați (AVL)

Teoremă: Dacă xt este un arbore echilibrat de dimensiune n şi înălțime h atunci

$$h <= 1.4404\log(n+1) + O(1)$$

Consecință: Operațiile de bază pe arbori echilibrați (inserare, ştergere,..) sunt de complexitate O(log n)

Implementarea operațiilor – aceeaşi schemă plus reechilibrare

Reprezentarea mulțimilor prin arbori AVL

- Folosim tipul Stree şi construim un altul, AStree, adăugând pentru fiecare arbore şi valoarea înălțimii sale:

```
data AStree a = Null
              | Fork Int (AStree a) a (AStree a)
```

- Să considerăm o funcție ht care extrage dintr-un AStree eticheta de tip Int. Atunci putem descrie o funcție fork ce păstrează invariantul: "Eticheta de tip Int a unui Astree xt reprezintă înălțimea lui xt"

```
ht :: AStree a -> Int
ht Null = 0
ht(Fork h xt y zt) = h
```

Reprezentarea mulțimilor prin arbori AVL

- Funcția fork:

```
fork :: AStree a -> a -> AStree a -> AStree a
fork xt y zt = Fork h xt y zt
      where h = 1 + (max (ht xt  ht zt))

Main> fork Null 7 Null
Fork 1 Null 7 Null
Main> fork Null 7 (fork Null 2 (fork Null 3 Null))
Fork 3 Null 7 (Fork 2 Null 2 (Fork 1 Null 3 Null))
Main> fork Null 6 (Fork 4 Null 4 Null)
Fork 5 Null 6 (Fork 4 Null 4 Null)
```

Reprezentarea mulțimilor prin arbori AVL

- Funcția fork construieşte arbore echilibrat numai dacă xt şi zt sunt echilibrați şi diferența de înălțime este cel mult 1.

- Pentru a asigura construcția numai a arborilor echilibrați va trebui să proiectăm o altă funcție, de acelaşi tip:

```
spoon :: AStree a -> a -> AStree a -> AStree a
```

- Funcțiile de inserare, ştergere, etc. se definesc la fel ca în cazul Stree înlocuind Fork prin spoon

Funcțiile insert şi delete

```
insert :: (Ord a) => a -> AStree a -> AStree a
insert x Null = fork Null x Null
insert x (Fork h xt y zt)
   | (x < y)  = spoon (insert x xt) y zt
   | (x == y) = Fork h xt y zt
   | (x > y)  = spoon xt y (insert x zt)

delete :: (Ord a) => a -> AStree a -> AStree a
delete x Null =  Null
delete x (Fork h xt y zt)
   | (x < y)  = spoon(delete x xt) y zt
   | (x == y) = join xt  zt
   | (x > y)  = spoon xt y (delete x zt)

join :: AStree a -> AStree a -> AStree a
join xt yt = if isEmpty yt then xt else spoon xt y zt
       where (y, zt) = splitTree yt

splitTree :: AStree a -> (a, AStree a)
splitTree(Fork h xt y zt) = if isEmpty xt then (y,zt)
                                  else (u,spoon vt y zt)
              where(u,vt) = splitTree xt
```

Implementare spoon

- Fiecare inserare sau ştergere alterează înălțimea arborelui cu 1

- Este suficient să implementăm spoon xt y zt în condițiile în care xt şi zt sunt echilibrați şi înălțimile lor diferă cu cel mult 2

- Dacă ht xt = ht zt + 1 atunci spoon este fork
- Să presupunem ht xt = ht zt + 2 şi atunci xt este nevid:

  xt = Fork n ut v wt

  – Cazul ht zt = ht xt + 2 este simetric

- spoon xt y zt depinde de înălțimile relative ale lui ut şi wt

Implementare spoon

- Cazul 1: ht wt <= ht ut Atunci:

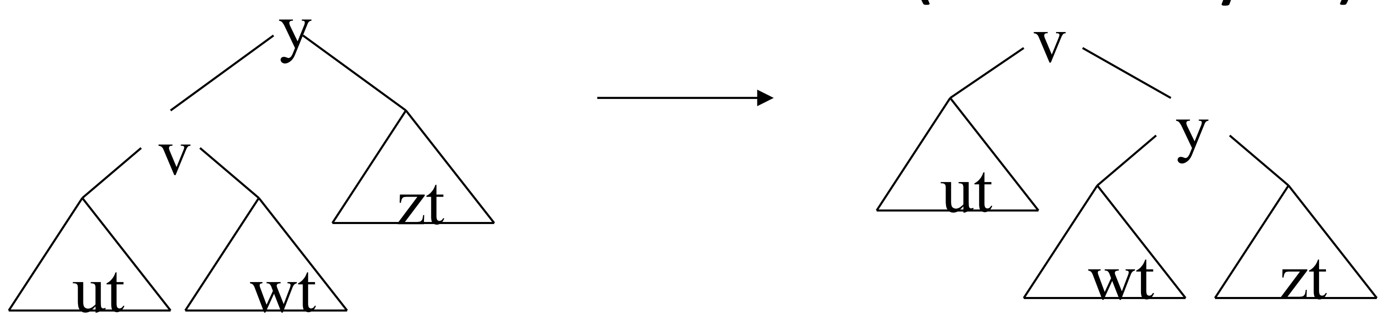
  ht zt = ht xt – 2 = ht ut – 1 <= ht wt <= ht ut

În acest caz definim:

  spoon xt y zt = rotr (fork xt y zt)

  rotr(Fork m (Fork n ut v wt) y zt) =

  fork ut v (fork wt y zt)



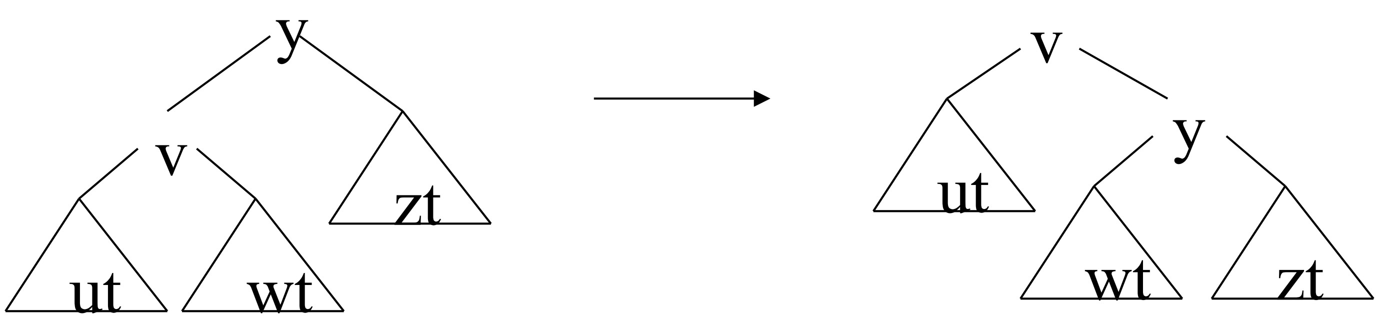
Implementare spoon

- Cazul 1 este corect:

abs(ht ut – ht(fork wt y zt)) = (def ht)

abs(ht ut -1 –max (ht wt, ht zt)) =

abs(ht ut – 1 – ht wt) <= 1



Implementare spoon

- Cazul 2: ht wt = 1 + ht ut

Atunci wt nu poate fi vid; fie wt = Fork p rt s tt

iar xt = Fork n ut v wt

Au loc:

ht zt = ht xt -2 = ht ut = ht wt -1= max(ht rt, ht tt)

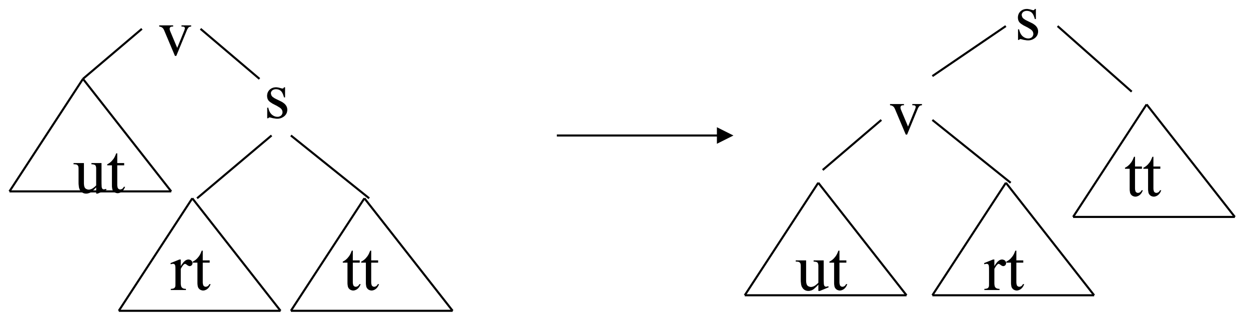
şi atunci definim:

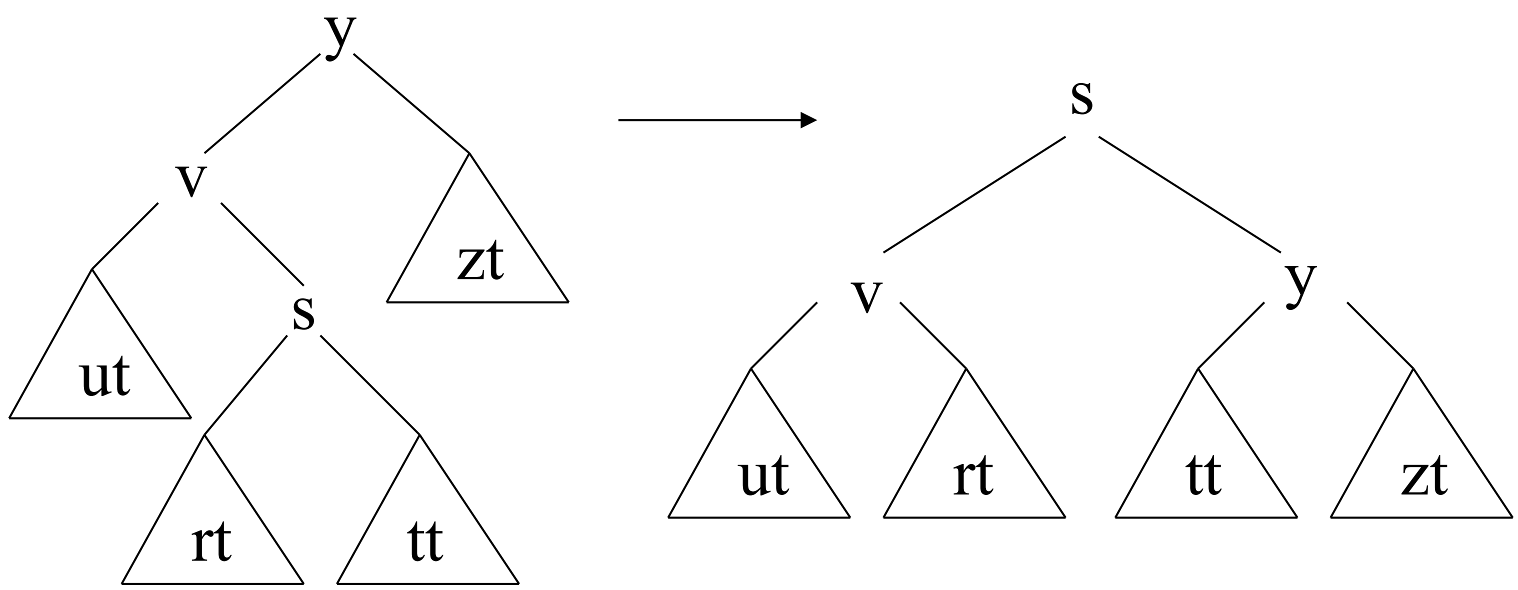
spoon xt y zt = rotr(fork(rotl xt) y zt)

unde

rotl(Fork n ut v (Fork p rt s tt)) = fork (fork ut v rt) s tt

Rotație dublă





Implementare spoon

- Cazul 2 este corect:

  spoon xt y zt = fork(fork ut v rt) s (fork tt y zt)

şi deci:

abs(ht (fork ut v rt) – ht(fork tt y zt)) =

abs(max(ht ut, ht rt) – max(ht tt, ht zt))=

abs(max(ht ut, ht rt) – ht zt) =

abs(ht ut-ht zt) = 0

Implementare spoon

- Implementarea completă:

```
spoon :: AStree a -> a -> AStree a -> AStree a
spoon xt y zt
        | (hz + 1 < hx) && (bias xt < 0)        = rotr(fork(rotl xt) y zt)
        | (hz + 1 < hx)                         = rotr(fork xt y zt)
        | (hx + 1 < hz) && (0 < bias zt)        = rotl(fork xt y (rotr zt))
        | (hx + 1 < hz)                         = rotl(fork xt y zt)
        | otherwise                             = fork xt y zt
                where        hx = ht xt
                             hz = ht zt

bias :: AStree a -> Int
bias(Fork h xt y zt) = ht xt - ht zt

rotr :: AStree a -> AStree a
rotr(Fork m (Fork n ut v wt) y zt) = fork ut v (fork wt y zt)

rotl :: AStree a -> AStree a
rotl(Fork m ut v (Fork n rt s tt)) = fork(fork ut v rt) s tt
```

Exemple

```
Main> insert 3 (insert 5 (insert 1 (insert 4
   (insert 2 Null))))
Fork 3 (Fork 1 Null 1 Null) 2 (Fork 2 (Fork 1 Null
   3 Null) 4 (Fork 1 Null 5 Null
))
Main> insert 3 (insert 5 (insert 1 (insert 4
   (insert 2 (insert 0 Null)))))
Fork 3 (Fork 2 Null 0 (Fork 1 Null 1 Null)) 2 (Fork
   2 (Fork 1 Null 3 Null) 4 (Fork 1 Null 5 Null))
Main> insert 3 (insert 5 (insert 1 (insert 4
   (insert 2 (insert 10 Null)))))
Fork 3 (Fork 2 (Fork 1 Null 1 Null) 2 (Fork 1 Null
   3 Null)) 4 (Fork 2 (Fork 1 Null 5 Null) 10 Null)
```

Multiset-uri (Bags)

- Nu contează ordinea elementelor

  {1,2,2,3} = {3,2 1,2}
- Contează elementele duplicate

  {1,2,2,3} ≠ {1,2,3}
- Tipul Bag a cu a din clasa Ord
- Operații:
  - mkBag :: [a] -> Bag a
  - isEmpty :: Bag a -> Bool
  - union :: Bag a -> Bag a -> Bag a
  - minBag :: Bag a -> a
  - delMin :: Bag a -> Bag a

Multiset-uri (Bags)

- Specificare algebrică:
  - isEmpty(mkBag xs) = null xs
  - union(mkBag xs) (mkBag ys) = mkBag(xs ++ ys)
  - minBag(mkBag xs) = minlist xs
  - delMin(mkBag xs) = mkBag(deleteMin xs)

unde deleteMin este funcția ce elimină o apariție a celui mai mic element dintr-o listă

Multiset-uri (Bags)

- Implementare prin liste nedescrescătoare
  - union este implementată cu *merge*
  - mkBag este implementat cu *sort*
- Complexitatea operațiilor:
  - mkBag xb   O(nlog n)
  - isEmpty xb   O(1)
  - union xb yb   O(m + n)
  - minBag xb   O(1)
  - delMin xb   O(1)

Multiset-uri (Bags)

- Implementare prin arbori heap augmentați

- Complexitatea operațiilor:
  - mkBag xb — O(n)
  - isEmpty xb — O(1)
  - union xb yb — O(log m + log n)
  - minBag xb — O(1)
  - delMin xb — O(log n)

Multiset-uri (Bags)

- Arbori heap augmentați
  - Arbori heap introduşi anterior:

```
data (Ord a) => Htree a = Null | Fork a (Htree a) (Htree a)
```

  - Se adaugă un întreg – dimensiunea subarborelui stâng

```
data Htree a = Null | Fork Int a (Htree a) (Htree a)
```

- Arbore “leftist”: Dimensiunea fiecărui subarbore stâng este cel puțin cât şi dimensiunea subarborelui drept corespunzător

Multiset-uri (Bags)

- Construirea cu funcția fork:
  - Dintr-o etichetă şi 2 arbori heap face un heap la care se adaugă informația dimensiune şi, eventual, schimbă ordinea membrilor

```
fork :: (Ord a) => a -> Htree a -> Htree a -> Htree a
fork x yt zt = if m < n then Fork p x zt yt else Fork p x yt zt
               where   m = size yt
                       n = size zt
                       p = m + n + 1

size :: Ord a=>Htree a -> Int
size Null = 0
size(Fork n x yt zt) = n
```

Multiset-uri (Bags)

- Implementarea operațiilor:

```
isEmpty :: Htree a -> Bool
isEmpty Null = True
isEmpty(Fork n x yt zt) = False

minBag :: Htree a -> a
minBag(Fork n x yt zt) = x

delMin :: Htree a -> Htree a
delMin(Fork n x yt zt) = union yt zt
```

Multiset-uri (Bags)

- Implementarea operațiilor:

```
union :: Htree a -> Htree a -> Htree a
union Null yt = yt
union(Fork m u vt wt) Null = Fork m u vt wt
union(Fork m u vt wt) (Fork n x yt zt)
      |u <= x = fork u vt (union wt (Fork n x yt zt))
      |x < u  = fork x yt (union (Fork m u vt wt) zt)
```

Multiset-uri (Bags)

- Implementarea operațiilor:

```
mkBag ::[a] -> Htree a
mkBags xs = fst(mkTwo(length xs) xs)

mkTwo :: Int -> [a] -> (Htree a , [a])
mkTwo n xs
   | (n == 0) = (Null, xs)
   | (n == 1) = (fork(head xs) Null Null , tail xs)
   | otherwise= (union xt yt, zs)
  where (xt, ys) = mkTwo m xs
        (yt, zs) = mkTwo(n-m) ys
         m       = n div 2
```

Multiset-uri (Bags) - Exemple

```
Main> mkBag [8,4,2,2,3,2,1]
Fork 7 1 (Fork 4 2 (Fork 2 2 (Fork 1 8 Null Null) Null) (Fork 1 4
   Null Null)) (Fork 2 2 (Fork 1 3 Null Null) Null)
Main> mkBag [2,2,2,3]
Fork 4 2 (Fork 2 2 (Fork 1 3 Null Null) Null) (Fork 1 2 Null Null)

Main> delMin(mkBag [2,2,2,3])
Fork 3 2 (Fork 1 3 Null Null) (Fork 1 2 Null Null)

Main> minBag(mkBag [2,2,2,3])
2
Main> union (mkBag [2,2,2,3]) (mkBag [2,3,1,1])
Fork 8 1 (Fork 5 1 (Fork 4 2 (Fork 2 2 (Fork 1 3 Null Null) Null)
 (Fork 1 2 Null Null)) Null) (Fork 2 2 (Fork 1 3 Null Null) Null)

Main> delMin (union (mkBag [2,2,2,3]) (mkBag [2,3,1,1]))
Fork 7 1 (Fork 4 2 (Fork 2 2 (Fork 1 3 Null Null) Null) (Fork 1 2
   Null Null)) (Fork 2 2 (Fork 1 3 Null Null) Null)
```