



Cursul 9 - ADT - continuare

- Tipul abstract Set
 - Reprezentarea cu arbori
 - Arbori echilibrați (AVL)
- Tipul abstract Bag
 - Reprezentare cu arbori heap



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
```



Mulțimi - reprezentarea cu arbori

- ❑ Funcțiile `join` și `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, ștergere, 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)

- ❑ Introduși de G. Adelson-Velski și 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 x_t este un arbore echilibrat de dimensiune n și înălțime h atunci

$$h \leq 1.4404 \log(n+1) + O(1)$$

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

Implementarea operațiilor - aceeași schemă plus reechilibrare



Reprezentarea mulțimilor prin arbori AVL

- ❑ Folosim tipul `Stree` și construim un altul, `ASTree`, adăugând pentru fiecare arbore și 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` construiește arbore echilibrat numai dacă `xt` și `yt` sunt echilibrați și diferența de înălțime este cel mult 1.
- ❑ Pentru a asigura construcția numai a arborilor echilibrați va trebui să proiectăm o altă funcție, de același tip:
`spoon :: AStree a -> a -> AStree a -> AStree a`
- ❑ Funcțiile de inserare, ștergere, etc. se definesc la fel ca în cazul `Stree` înlocuind `Fork` prin `spoon`



Funcțiile insert și delete

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

```
splitTree :: ATree a -> (a, ATree 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 ștergere alterează înălțimea arborelui cu 1
- ❑ Este suficient să implementăm $\text{spoon } xt \ y \ zt$ în condițiile în care xt și yt sunt echilibrați și înălțimile lor diferă cu cel mult 2
- ❑ Dacă $ht \ xt = ht \ zt + 1$ atunci spoon este fork
- ❑ Să presupunem $ht \ xt = ht \ zt + 2$ și atunci xt este nevid: $xt = \text{Fork } n \ ut \ v \ wt$
 - Cazul $ht \ zt = ht \ xt + 2$ este simetric
- ❑ $\text{spoon } xt \ y \ zt$ depinde de înălțimile relative ale lui ut și wt



Implementare spoon

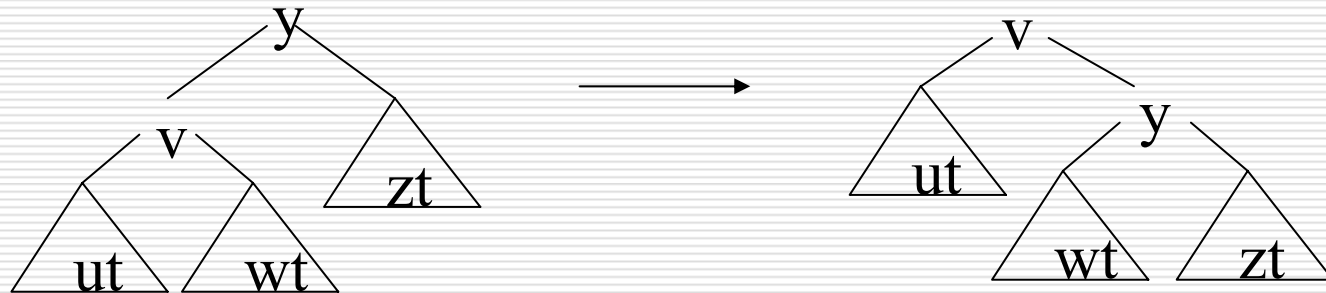
□ Cazul 1: $ht\ wt \leq ht\ ut$ Atunci:

$$ht\ zt = ht\ xt - 2 = ht\ ut - 1 \leq ht\ wt \leq ht\ ut$$

În acest caz definim:

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

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





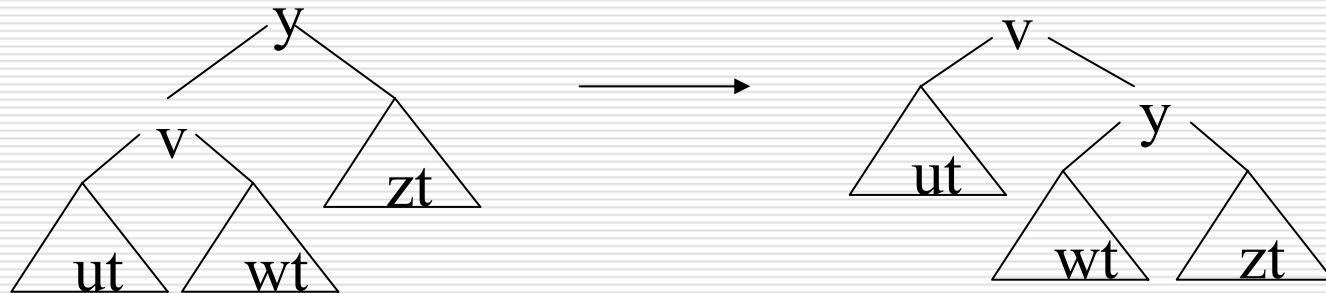
Implementare spoon

□ Cazul 1 este corect:

$\text{abs}(\text{ht } ut - \text{ht}(\text{fork } wt \ y \ zt)) = \quad (\text{def } \text{ht})$

$\text{abs}(\text{ht } ut - 1 - \max(\text{ht } wt, \text{ht } zt)) =$

$\text{abs}(\text{ht } ut - 1 - \text{ht } wt) \leq 1$





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:

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

și atunci definim:

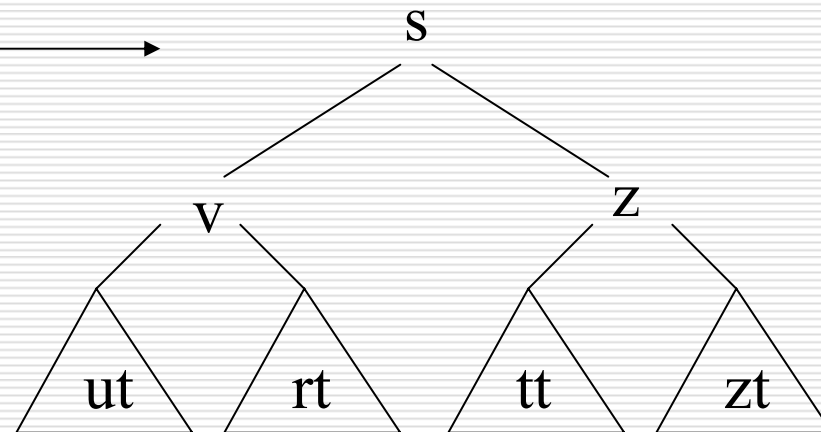
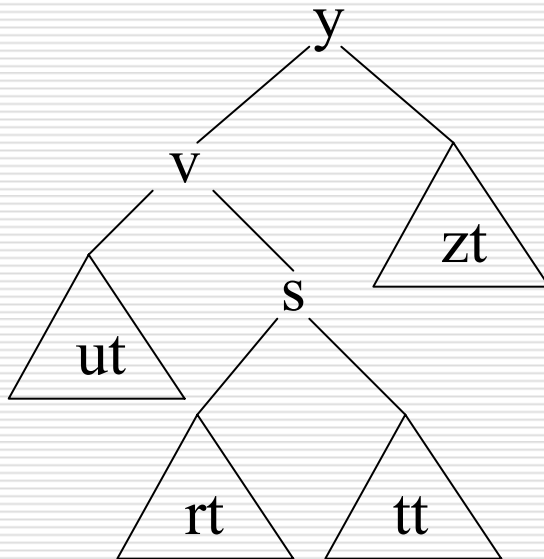
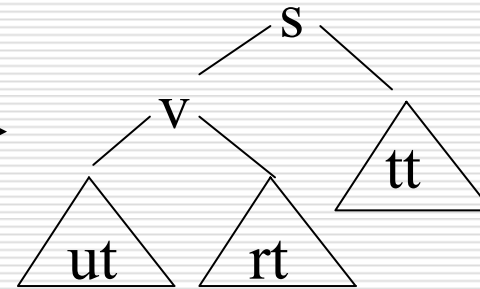
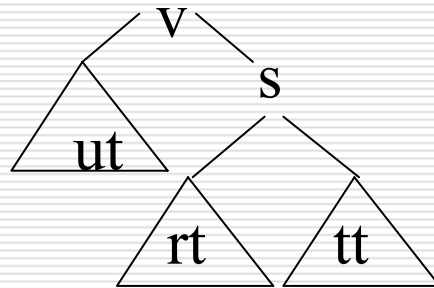
$$spoon\ xt\ y\ zt = rotr(fork(rotl\ xt)\ y\ zt)$$

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:

$\text{spoon } xt \ y \ zt = \text{fork}(\text{fork } ut \ v \ rt) \ s \ (\text{fork } tt \ y \ zt)$

și deci:

$$\begin{aligned} \text{abs}(\text{ht } (\text{fork } ut \ v \ rt) - \text{ht}(\text{fork } tt \ y \ zt)) &= \\ \text{abs}(\max(\text{ht } ut, \text{ht } rt) - \max(\text{ht } tt, \text{ht } zt)) &= \\ \text{abs}(\max(\text{ht } ut, \text{ht } rt) - \text{ht } zt) &= \\ \text{abs}(\text{ht } ut - \text{ht } zt) &= 0 \end{aligned}$$



Implementare spoon

□ Implementarea completă:

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

`spoon xt y zt`

<code> (hz + 1 < hx) && (bias xt < 0)</code>	<code>= rotr(fork(rotl xt) y zt)</code>
<code> (hz + 1 < hx)</code>	<code>= rotr(fork xt y zt)</code>
<code> (hx + 1 < hz) && (0 < bias zt)</code>	<code>= rotl(fork xt y (rotr zt))</code>
<code> (hx + 1 < hz)</code>	<code>= rotl(fork xt y zt)</code>
<code> otherwise</code>	<code>= fork xt y zt</code>

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`



Example

```
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
 $\{1,2,2,3\} \neq \{1,2,3\}$
- Tipul Bag a cu a din clasa Ord
- Operații:
 - mkBag $:: [a] \rightarrow \text{Bag } a$
 - isEmpty $:: \text{Bag } a \rightarrow \text{Bool}$
 - union $:: \text{Bag } a \rightarrow \text{Bag } a \rightarrow \text{Bag } a$
 - minBag $:: \text{Bag } a \rightarrow a$
 - delMin $:: \text{Bag } a \rightarrow \text{Bag } a$



Multiset-uri (Bags)

□ Specificare algebrică:

- $\text{isEmpty}(\text{mkBag } xs) = \text{null } xs$
- $\text{union}(\text{mkBag } xs) (\text{mkBag } ys) = \text{mkBag}(xs ++ ys)$
- $\text{minBag}(\text{mkBag } xs) = \text{minlist } xs$
- $\text{delMin}(\text{mkBag } xs) = \text{mkBag}(\text{deleteMin } xs)$

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 $O(n \log n)$
 - 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 introduși 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 și dimensiunea subarborelui drept corespunzător



Multiset-uri (Bags)

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

```
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)
```