Cursul 6

- Arbori binari
  - Măsuri
- Arbori binari augmentați
- Arbori binari de căutare

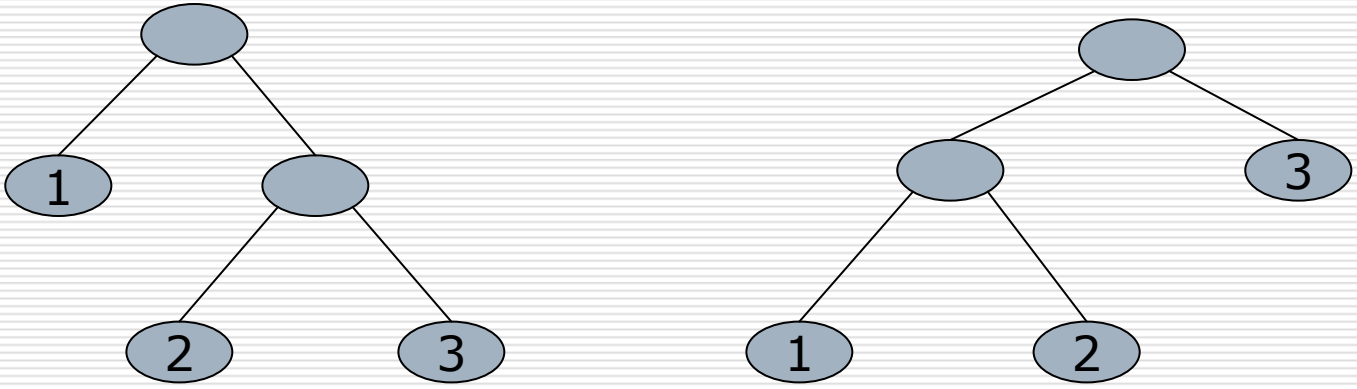
Structura de data "arbori binari"

- O valoare a tipului **`Btree a`** este fie:
  - un nod frunză (Leaf) ce conține o valoare de tip **`a`**
  - un nod ramificație (Fork) şi doi noi arbori, subarborele stâng al nodului ramificație respectiv cel drept
  - o frunză se numeşte nod exterior
  - un nod ramificație se numeşte nod interior

Structura de date arbori binari

- Sintaxa pentru introducerea tipul **`Btree a`** este:

```
data Btree a = Leaf a | Fork(Btree a)(Btree a)
                      deriving (Show)
Fork(Leaf 1)(Fork(Leaf 2)(Leaf 3))
Fork(Fork(Leaf 1)(Leaf 2))(Leaf 3)
```



Structura de date arbori binari

- Variabilele ce desemnează arbori binari le vom nota xt, yt, ...
- Demonstrarea unei propoziții P(xt) se face prin inducție structurală:
  - P(Leaf x)
  - P(xt), P(yt) ⟹ P(Fork xt yt)
- Un arbore finit este un arbore ce are un număr finit de frunze

Măsuri în Btree

- size : numărul nodurilor frunză

```
size :: Btree a -> Int
size(Leaf x) = 1
size(Fork xt yt) = size xt + size yt
```

- Legătura cu length de la liste:

```
size = lenght.flatten

flatten :: Btree a -> [a]
flatten(Leaf x) = [x]
flatten(Fork xt yt) = flatten xt ++ flatten yt
```

Exemple:

```
    t1, t2, t3 :: Btree Int
    t1 = Fork(Leaf 1)(Fork(Leaf 2)(Leaf 3))
    t2 = Fork(Fork(Leaf 1)(Leaf 2))(Leaf 3)
    t3 = Fork(Fork(Leaf 1)(Fork(Leaf 2)(Leaf 3)))
       (Fork(Fork(Leaf 4)(Leaf 5))(Leaf 6))
Main> size t1
3
Main> size t2
3
Main> flatten t1
[1,2,3]
Main> size t3
6
Main> flatten t3
[1,2,3,4,5,6]
Main> (length.flatten) t3
6
```

Măsuri în Btree

- nodes: numărul nodurilor interne

```
nodes :: Btree a -> Int
nodes(Leaf x) = 0
nodes(Fork xt yt) = 1 + nodes xt + nodes yt
```

- height: lungimea drumului maxim de la radacina la frunze

```
height :: Btree a -> Int
height(Leaf x) = 0
height(Fork xt yt) =
                1+(max (height xt) (height yt))
```

Exemple:

```
    t1, t2, t3 :: Btree Int
    t1 = Fork(Leaf 1)(Fork(Leaf 2)(Leaf 3))
    t2 = Fork(Fork(Leaf 1)(Leaf 2))(Leaf 3)
    t3 = Fork(Fork(Leaf 1)(Fork(Leaf 2)(Leaf 3)))
       (Fork(Fork(Leaf 4)(Leaf 5))(Leaf 6))
Main> nodes t1
2
Main> nodes t3
5
Main> height t1
2
Main> height t3
3
```

Măsuri în Btree

- depths: funcție ce înlocuieşte într-un arbore valoarea din fiecare frunză cu adâncimea (depth) frunzei în arbore
- În acest fel, înălțimea unui arbore este maximum din adâncimile frunzelor

```
depths :: Btree a -> Btree Int
depths = down 0

down :: Int -> Btree a -> Btree Int
down n (Leaf x) = Leaf n
down n (Fork xt yt) = Fork(down(n+1)xt)(down(n+1)yt)
```

Măsuri în Btree

- Înălțimea unui arbore este maximum din adâncimile frunzelor:

```
height = maxBtree.depths

maxBtree :: (Ord a)=>Btree a -> a
maxBtree(Leaf x) = x
maxBtree(Fork xt yt) =
          max (maxBtree xt)(maxBtree yt)
```

Exemple:

```
t4 = Fork(Fork(Leaf '1')(Fork(Leaf '2')(Leaf '3')))
   (Fork(Fork(Leaf '4')(Leaf '5'))(Leaf '6'))

Main> flatten t4
"123456"
Main> (maxBtree.depths) t4
3
Main> t4
Fork (Fork (Leaf '1') (Fork (Leaf '2') (Leaf '3')))
   (Fork (Fork (Leaf '4') (Leaf '5')) (Leaf '6'))
Main> depths t4
Fork (Fork (Leaf 2) (Fork (Leaf 3) (Leaf 3)))
   (Fork (Fork (Leaf 3) (Leaf 3)) (Leaf 2))
```

Arbori perfecți

- Un arbore binar se zice *perfect* dacă toate frunzele sale au aceeaşi adâncime

```
Main> t5
Fork (Fork (Leaf 1) (Leaf 2)) (Fork (Leaf 3) (Leaf 4))
Main> depths t5
Fork (Fork (Leaf 2) (Leaf 2)) (Fork (Leaf 2) (Leaf 2))
Main> (maxBtree.depths) t5
2
Main> flatten t5
[1,2,3,4]
```

Proprietăți

- Dacă **xt** este un arbore perfect atunci **size xt** este o putere a lui 2; există exact un arbore perfect pentru fiecare putere a lui 2 (excepție făcând de valorile frunzelor)

- **height xt < size xt <= $2^{\text{height xt}}$**

- **$\lceil$log(size xt)$\rceil$ <= height xt < size xt**

Construcția unui arbore

- Dată o lista xs de lungime n să se construiască un arbore xt pentru care:

`flatten xt = xs, height xt = log n`

- Există mai mulți arbori cu această proprietate
- O soluție: se împarte xs în două şi se construieşte recursiv, pentru fiecare jumătate câte un arbore

Construcția unui arbore

```
mkBtree :: [a] -> Btree a
mkBtree xs
    | (m == 0)   = Leaf(unwrap xs)
    | otherwise = Fork(mkBtree ys)(mkBtree zs)
    where m = (length xs) `div` 2
          (ys, zs) = splitAt m xs
          unwrap[x] = x
```

- De la liste:

```
splitAt n xs = (take n xs, drop n xs)
```

Construcția unui arbore: Exemple

```
Main> mkBtree [1]
Leaf 1
Main> mkBtree [1,2,3]
Fork (Leaf 1) (Fork (Leaf 2) (Leaf 3))
Main> mkBtree [1,2,3,4]
Fork (Fork (Leaf 1) (Leaf 2)) (Fork (Leaf 3) (Leaf 4))
Main> mkBtree ['a','b','c','d','e']
Fork (Fork (Leaf 'a') (Leaf 'b')) (Fork (Leaf 'c') (Fork
    (Leaf 'd') (Leaf 'e')))
Main> (flatten.mkBtree) ['a','b','c','d','e']
"abcde"
```

Funcțiile mapBtree şi foldBtree

- Sunt funcțiile analoge funcțiilor map şi fold de la liste
- Funcția mapBtree:

```
mapBtree :: (a -> b) -> Btree a -> Btree b
mapBtree f (Leaf x) = Leaf f x
mapBtree f (Fork xt yt) = Fork(mapBtree f xt)(mapBtree f yt)
```

- Proprietăți:

```
mapBtree id  = id
mapBtree (f.g) = mapBtree f . mapBtree g
map f . flatten = flatten . mapBtree f
```

Funcțiile mapBtree şi foldBtree

- Btree are 2 constructori:

```
Leaf :: a -> Btree a
Fork :: Btree a -> Btree a -> Btree a
```

- Funcția foldBtree trebuie să furnizeze aplicarea a 2 funcții pentru un arbore dat:
  - `f :: a -> b` (se aplică valorilor din frunze)
  - `g :: b -> b -> b` (se aplică rezultatelor aplicării lui f)
- Funcția foldBtree:

```
foldBtree :: (a -> b) -> (b -> b -> b) -> Btree a -> b
foldBtree f g (Leaf x)          = f x
foldBtree f g (Fork xt yt)      =
                    g(foldBtree f g xt)(foldBtree f g yt)
```

Exemple

- Funcția mapBtree:

```
Main> mapBtree (+5) t2
Fork (Fork (Leaf 6) (Leaf 7)) (Leaf 8)
Main> mapBtree (+1) t5
Fork (Fork (Leaf 2) (Leaf 3)) (Fork (Leaf 4) (Leaf 5))
```

- Funcția foldBtree:

```
Main> foldBtree(const 1)(+) t5
4
Main> foldBtree(const 1)(+) t4
6
Main> foldBtree(id)(max) t3
6
Main> foldBtree(id)(max) t4
'6'
```

Exemple

- Funcțiile definite la început se pot exprima cu foldBtree:

```
size   = foldBtree(const 1)(+)
height = foldBtree(const 0)(⊕)
                 where m ⊕ n = 1 + (m `max` n)
flatten = foldBtree wrap (++)
                 where wrap x = [x]
maxBtree = foldBtree id (max)
mapBtree f = foldBtree(Leaf.f) Fork
```

Arbori binari augmentați

- Arborii binari introduşi au informații doar în frunze
- Este util în aplicații să adăugăm informații în nodurile interioare:

```
data Atree a = Leaf a | Fork Int (Atree a)(Atree a)
```

- Ideea este ca în arborele **`Fork n xt yt`** n să fie size xt + size yt

- Acest lucru este asigurat prin construirea de noduri fork utilizând o funcție adecvată

Arbori binari etichetați

- Funcția de construire a nodurilor ramificație

```
fork :: Atree a -> Atree a -> Atree a
fork xt yt = Fork n xt yt
     where n = lsize xt + lsize yt

lsize :: Atree a -> Int
lsize(Leaf x) = 1
lsize(Fork n xt yt) = n
```

Arbori binari etichetați

- Funcția mkAtree:

```
mkAtree :: [a] -> Atree a
mkAtree xs
        | (m == 0)   = Leaf(unwrap xs)
        | otherwise = fork(mkAtree ys)(mkAtree zs)
        where m = (length xs) `div` 2
              (ys, zs) = splitAt m xs
              unwrap [x] = x
```

Arbori binari etichetați

- Funcția mkAtree:

```
Main> mkAtree [1,2,3,4]
Fork 4 (Fork 2 (Leaf 1) (Leaf 2)) (Fork 2 (Leaf 3)
    (Leaf 4))
Main> mkAtree ['a', 'b', 'c', 'd', 'e']
Fork 5 (Fork 2 (Leaf 'a') (Leaf 'b'))
    (Fork 3 (Leaf 'c') (Fork 2 (Leaf 'd') (Leaf 'e')))
```

Arbori binari de căutare

- Structura de dată arbore binar de căutare trebuie să fie legată de un tip din Ord:

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

- Un arbore Stree nu mai are noduri frunză
- Constructorul Null denotă arborele vid
- Un arbore nevid are un subarbore stâng, o etichetă de tip a şi un subarbore drept
- Arborii Stree se mai numesc arbori etichetați

Arbori binari de căutare

- Crearea unui arbore de căutare de la o secvență dată:

```
mkStree :: (Ord a) => [a] -> Stree a
mkStree [] = Null
mkStree (x:xs) = Fork(mkStree ys) x (mkStree zs)
       where (ys, zs) = partition (<= x) xs

partition :: (a -> Bool) -> [a] -> ([a],[a])
partition p xs = (filter p xs, filter (not.p) xs)
```

- Nu se construieşte arborele de adâncime minimă

Arbori binari de căutare

- Crearea unui arbore de căutare de la o secvență dată:

```
Main> mkStree [1,2,3,4,5,6,7]
Fork Null 1 (Fork Null 2 (Fork Null 3 (Fork Null 4 (Fork Null 5 (Fork
    Null 6 (Fork Null 7 Null))))))
Main> mkStree [3,5,2,6,7]
Fork (Fork Null 2 Null) 3 (Fork Null 5 (Fork Null 6 (Fork Null 7
    Null)))
Main> mkStree ['a','d','e','m','a']
Fork (Fork Null 'a' Null) 'a' (Fork Null 'd' (Fork Null 'e'
        (Fork Null 'm' Null)))
```

Arbori binari de căutare

- Funcția de căutare **member** :

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

Main> member 2 (mkStree [5,3,6,1,2,4])
True
Main> member 5 (mkStree [5,3,6,1,2,4])
True
Main> member 7 (mkStree [5,3,6,1,2,4])
False
```

Arbori binari de căutare

- Funcția de sortare :

```
sort :: (Ord a) => [a] -> [a]
sort = flatten.mkStree

Main> flatten (mkStree [5,3,6,1,2,4])
[1,2,3,4,5,6]
Main> sort [5,3,6,1,2,4]
[1,2,3,4,5,6]
Main> sort [5,3,6,1,2,4,2,6]
[1,2,2,3,4,5,6,6]
```

Arbori binari de căutare-inserare

- Adăugarea unui nod de etichetă dată:
  - Dacă eticheta există atunci nu se adaugă nici un nod
  - Dacă eticheta nu există se adaugă ca şi nod fără descendenți, la locul lui

```
insert :: (Ord a) => a -> Stree a-> Stree a
insert x Null = Fork Null x Null
insert x (Fork xt y yt)
   | (x < y) = Fork (insert x xt) y yt
   | (x == y)= Fork xt y yt
   | (x > y) = Fork xt y (insert x yt)
```

Arbori binari de căutare-inserare

- Exemple:

```
Main> mkStree [2,5,3,1]
Fork (Fork Null 1 Null) 2 (Fork (Fork Null 3 Null) 5 Null)

Main> insert 6 (mkStree [2,5,3,1])
Fork (Fork Null 1 Null) 2 (Fork (Fork Null 3 Null) 5 (Fork Null 6 Null))

Main> mkStree "info"
Fork (Fork Null 'f' Null) 'i' (Fork Null 'n' (Fork Null 'o' Null))

Main> insert 'a' (insert 'b' (mkStree "info"))
Fork (Fork (Fork (Fork Null 'a' Null) 'b' Null) 'f' Null) 'i' (Fork Null 'n'
     (Fork Null 'o' Null))
```

Arbori binari de căutare-ştergere

- Ştergerea unui nod de etichetă dată presupune ca cei 2 subarbori rămaşi prin eliminarea rădăcinii să fie combinați astfel ca noul arbore să aibă proprietățile cerute
- O funcție `join` care combină în acest mod 2 arbori ar trebui să îndeplinească condiția

```
flatten(join xt yt) = flatten xt ++ flatten yt
```

- O soluție (nu cea mai bună) este să se înlocuiască cel mai din dreapta subarbore Null din xt cu yt:

```
join :: (Ord a) => Stree a -> Stree a -> Stree a
join Null yt = yt
join (Fork ut x vt) yt = Fork ut x (join vt yt)
```

Arbori binari de căutare-ştergere

- Exemplu:

```
Main> t1
Fork (Fork Null 1 Null) 4 (Fork Null 6 Null)
Main> t2
Fork (Fork Null 8 Null) 9 (Fork Null 11 Null)
Main> flatten(join t1 t2)
[1,4,6,8,9,11]
Main> flatten t1 ++ flatten t2
[1,4,6,8,9,11]
Main> flatten (Fork t1 7 t2)
[1,4,6,7,8,9,11]
```

Arbori binari de căutare-ştergere

- Funcția delete se poate defini astfel:

```
delete :: (Ord a) => a -> Stree a -> Stree a
delete x Null = Null
delete x (Fork xt y yt)
   | (x < y) = Fork (delete x xt) y yt
   | (x == y)= join xt yt
   | (x > y) = Fork xt y (delete x yt)
```

Arbori binari de căutare-ştergere

- Exemple:

```
Main> flatten(delete 7 (Fork t1 7 t2))
[1,4,6,8,9,11]
Main> flatten(delete 4 (Fork t1 7 t2))
[1,6,7,8,9,11]
Main> flatten(delete 3 (Fork t1 7 t2))
[1,4,6,7,8,9,11]
Main> flatten(delete 1 (Fork t1 7 t2))
[4,6,7,8,9,11]
Main> flatten(delete 11 (Fork t1 7 t2))
[1,4,6,7,8,9]
Main> delete 1 (Fork t1 7 t2)
Fork (Fork Null 4 (Fork Null 6 Null)) 7 (Fork (Fork Null 8 Null) 9
    (Fork Null 11 Null))
Main> delete 3 (Fork t1 7 t2)
Fork (Fork (Fork Null 1 Null) 4 (Fork Null 6 Null)) 7 (Fork (Fork
    Null 8 Null) 9 (Fork Null 11 Null))
```

Arbori binari de căutare-ştergere

- Altă soluție pentru join: cea mai mică etichetă a arborelui yt să devină rădacină pentru noul arbore
  - Asta implementează faptul că:

    xs ++ ys = xs ++ [head ys] ++ tail ys
  - Folosim funcțiile **headTree** şi **tailTree** cu proprietățile:

```
headTree :: (Ord a) => Stree a -> a
headTree = head.flatten

tailTree :: (Ord a) => Stree a -> Stree a
flatten.tailTree = tail.flatten

join :: (Ord a) => Stree a -> Stree a -> Stree a
join xt yt = if yt == Null then xt else
                  Fork xt (headTree yt)(tailTree yt)
```

Arbori binari de căutare-ştergere

- Implementarea funcțiilor headTree şi tailTree:

```
splitTree :: Ord a => Stree a -> (a, Stree a)
splitTree (Fork xt y yt) =
   if xt== Null then (y, yt) else (x, Fork wt y yt)
   where (x, wt) = splitTree xt

headTree :: (Ord a) => Stree a -> a
headTree(Fork xt y yt) = fst (splitTree (Fork xt y yt))

tailTree :: (Ord a) => Stree a -> Stree a
tailTree (Fork xt y yt) = snd (splitTree (Fork xt y yt))
```

Arbori binari de căutare-ştergere

- Exemple:

```
Main> delete 1 (Fork t1 7 t2)
Fork (Fork Null 4 (Fork Null 6 Null)) 7 (Fork (Fork
   Null 8 Null) 9 (Fork Null 11
 Null))
Main> flatten(delete 11 (Fork t1 7 t2))
[1,4,6,7,8,9]
Main> flatten(delete 1 (Fork t1 7 t2))
[4,6,7,8,9,11]
Main> flatten(delete 3 (Fork t1 7 t2))
[1,4,6,7,8,9,11]
Main> flatten(delete 4 (Fork t1 7 t2))
[1,6,7,8,9,11]
```

Arbori binari de căutare-ştergere

- Exemple:

```
Main> flatten(delete 7 (Fork t1 7 t2))
[1,4,6,8,9,11]
Main> delete 7 (Fork t1 7 t2)
Fork (Fork (Fork Null 1 Null) 4 (Fork Null 6 Null)) 8
   (Fork Null 9 (Fork Null 11 Null))
Main> delete 4 (Fork t1 7 t2)
Fork (Fork (Fork Null 1 Null) 6 Null) 7 (Fork (Fork
   Null 8 Null) 9 (Fork Null 11 Null))
Main> delete 9 (Fork t1 7 t2)
Fork (Fork (Fork Null 1 Null) 4 (Fork Null 6 Null)) 7
   (Fork (Fork Null 8 Null) 11 Null)
Main> delete 6 (Fork t1 7 t2)
Fork (Fork (Fork Null 1 Null) 4 Null) 7 (Fork (Fork
   Null 8 Null) 9 (Fork Null 11 Null))
```