Programare declarativă

Monoid - aplicații

Traian Florin Șerbănuță
Ioana Leuștean

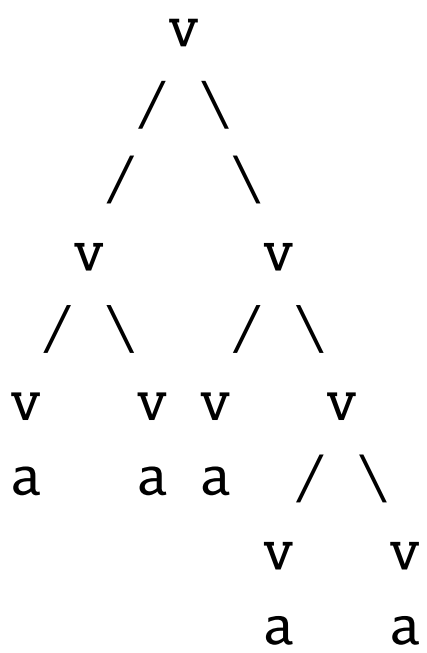
Departamentul de Informatică, FMI, UNIBUC
traian.serbanuta@fmi.unibuc.ro

clasa Monoid - aplicații

Structuri de căutare

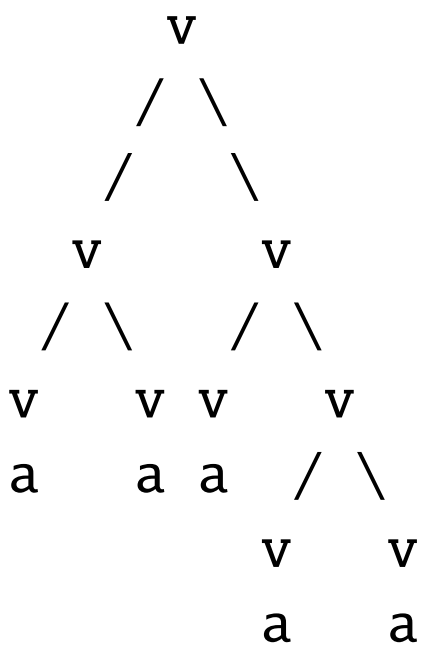
https://apfelmus.nfshost.com/articles/monoid-fingertree.html

- implementarea eficientă și unitară a structurilor de date funcționale
- informația se află în frunze
- nodurilor interne conțin valori care
  - structura lor determină funcționalitatea arborelui
  - e.g., spun dacă informația căutată se găsește în arbore



Search Tree

```
data  STree v a =   Leaf v a
                  | Node v  (STree v a) (STree v a)
                  deriving Show
```



Search Tree

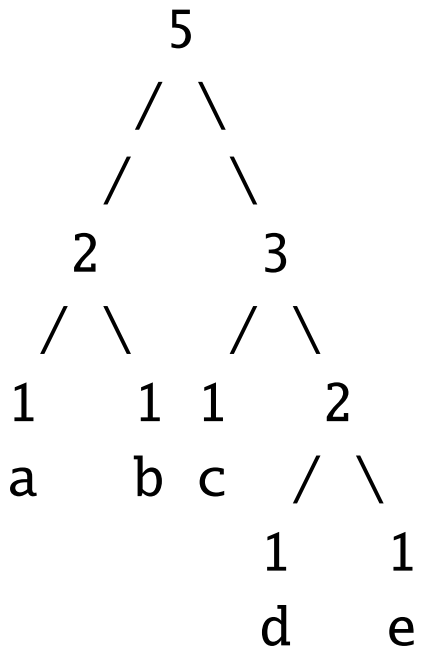
```
data  STree v a =   Leaf v a
                  | Node v  (STree v a) (STree v a)
                  deriving Show

exFT = Node  5
             (Node 2
                   (Leaf 1 'a')
                   (Leaf 1 'b'))
             (Node 3
                   (Leaf 1 'c')
                   (Node 2
                         (Leaf 1 'd')
                         (Leaf 1 'e')))


*Main> :t exFT
exFT :: STree Int String
```

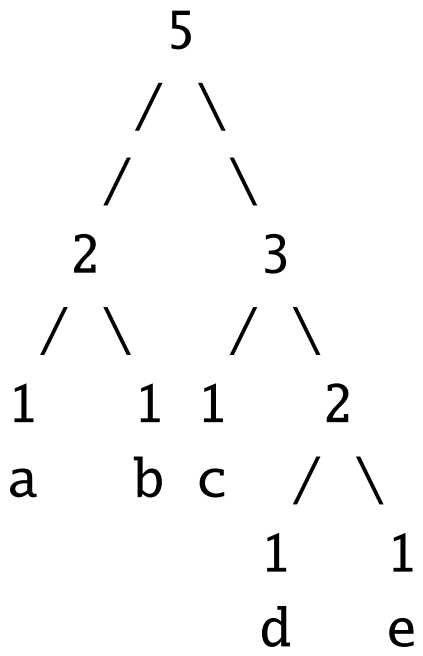
Search Tree

Ce reprezinta informatia din arbore? Depinde de ce vrem să căutăm în el.



Search Tree

Ce reprezinta informatia din arbore? Depinde de ce vrem să căutăm în el.



Căutarea elementului de pe o poziție dată

(P1) Valoarea unui nod intern reprezintă numărul de frunze

```
type Size = Int
exFT :: STree Size Char
```

Search Tree

Căutarea elementului de pe o poziție dată

- (P1) Valoarea unui nod intern reprezintă numărul de frunze

```
tag :: STree v a -> v
tag (Leaf n _) = n
tag (Node n _ _) = n
```

- Lista datelor este alcătuită din frunze, de la stânga la dreapta

```
toList :: STree v a -> [a]
toList (Leaf _ a)     = [a]
toList (Node _ x y) = toList x ++ toList y
```

Search Tree

Căutarea elementului de pe o poziție dată

```
data  STree v a =   Leaf v a
                  | Node v  (STree v a) (STree v a)
                  deriving Show
```

Funcții specifice pentru construirea unui arbore cu (P1)

```
type Size = Integer

leaf :: a -> STree Size a
leaf x = Leaf 1 x

node :: STree Size a -> STree Size a -> STree Size a
node  t1 t2 = Node ((tag t1) + (tag t2)) t1 t2
```

```
exFT == node (node (leaf 'a')(leaf 'b'))
             (node (leaf 'c') (node (leaf 'd')(leaf 'e')))
```

Search Tree

Căutarea elementului de pe o poziție dată

```
(!!!) :: STree Size a -> Int -> a
(Leaf _ a)       !!! 0 = a
(Node _ x y)  !!! n
      | n < tag x      = x !!! n
      | otherwise      = y !!! (n - tag x)
```

```
*Main> exFT !!! 3
'd'
```

Timpul de acces este dat de înălțimea arborelui.

Search Tree

Coadă cu priorități

(P2) Valoarea fiecărui nod intern reprezintă cea mai mica prioritate din arborele respectiv.

```
pqFT = Node  2
             (Node 4
                   (Leaf 16 'a')
                   (Leaf 4 'b'))
             (Node 2
                   (Leaf 2 'c')
                   (Node 8
                         (Leaf 32 'd')
                         (Leaf 8 'e')))
```

```
type Priority = Int
pqFT :: STree Priority Char
```

Search Tree

Coadă cu priorități

Funcții specifice pentru construirea unui arbore cu (P2)

```
type Priority = Int

pleaf ::Priority -> a -> STree Priority a
pleaf n x = Leaf n x

pnode :: STree Priority a -> STree Priority a -> STree
    Priority a
pnode  t1 t2 = Node ((tag t1) `min` (tag t2)) t1 t2
```

```
exFT == pnode (pnode (pleaf 16 'a')(pleaf 4 'b'))
              (pnode (pleaf 2 'c') (pnode (pleaf 32 'd')(
                  pleaf 8 'e')))
```

Search Tree

Coadă cu priorități

```
winner :: STree Priority a -> a
winner t = go t
    where
    go (Leaf _ a)          = a
    go (Node _ x y)
        | tag x == tag t = go x
        | tag y == tag t = go y
```

```
*Main> winner pqFT
'c'
```

Dacă arborele se menține echilibrat, timpul de acces poate fi îmbunătățit.

Arbori de căutare

Unificarea celor două exemple

Observăm că

- Pentru arbori cu (P1) funcția **tag** verifică:
  $tag :: STree\ Size\ a \to Size$
  $tag(Leaf\ \_) = 1$
  $tag(Node\ \_\ x\ y) = tag\ x\ +\ tag\ y$

- Pentru arbori cu (P2) funcția **tag** verifică:
  $tag :: STree\ Priority\ a \to Priority$
  $tag(Leaf\ \_\ a) = priority\ a$
  $tag(Node\ \_\ x\ y) = tag\ x\ `min`\ tag\ y$

Arbori de căutare

Unificarea celor două exemple

Observăm că

- Pentru arbori cu (P1) funcția **tag** verifică:
  $tag :: STree\ Size\ a \to Size$
  $tag(Leaf\ \_) = 1$
  $tag(Node\ \_\ x\ y) = tag\ x\ +\ tag\ y$

  $(Size, +, 0)$ monoid

- Pentru arbori cu (P2) funcția **tag** verifică:
  $tag :: STree\ Priority\ a \to Priority$
  $tag(Leaf\ \_\ a) = priority\ a$
  $tag(Node\ \_\ x\ y) = tag\ x\ `min`\ tag\ y$

  $(Priority, min, maxBound)$ monoid

Arbori de căutare

Unificarea celor două exemple folosind monoizi

- Pentru arbori cu (P1) definim o instanța **Monoid** a lui **Size**:

```
instance Monoid Size where
    mempty  = 0
    mappend = (+)
```

- Pentru arbori cu (P2) definim o instanța **Monoid** a lui **Priority**:

```
instance Monoid Priority where
    mempty  = maxBound
    mappend = min
```

Atenție!

În acest exemplu **Size** și **Priority** sunt redenumiri ale lui **Int**. Pentru a putea fi făcute instanțe ale clasei **Monoid** simultan trebuie folosit **newtype**.

Arbori de căutare

Unificarea celor două exemple folosind monoizi

Constructorul pentru **Node**

```
node :: Monoid v => STree v a -> STree v a -> STree v a
node x y = Node (tag x <> tag y) x y
```

Constructorul pentru **Leaf**

Cum transmitem tag-urile asociate frunzelor?

```
leaf :: Monoid v => (a->v) -> a -> STree v a
leaf measure x = Leaf (measure x) x
```

Transmitem ca parametru o funcție care asociază fiecărei date tag-ul corespunzător.

Arbori de căutare

Unificarea celor două exemple folosind monoizi

Constructorii pentru **Node** și **Leaf**

```
node :: Monoid v => STree v a -> STree v a -> STree v a
node x y = Node (tag x <> tag y) x y

leaf :: Monoid v => (a->v) -> a -> STree v a
leaf measure x = Leaf (measure x) x
```

```
priority :: Char -> Int
*Main> leaf priority 'a'
Leaf 16 'a'
*Main> node (leaf priority 'a') (leaf priority 'b')
Node 3 (Leaf 16 'a') (Leaf 4 'b')
*Main> node (Leaf 16 'a') (Leaf 4 'b') :: STree Int Char
Node 3 (Leaf 16 'a') (Leaf 4 'b')
```

Arbori de căutare

Unificarea căutării

```
search :: Monoid v => (v -> Bool) -> STree v a -> Maybe a
search  p t
    | p (tag t) = Just (go mempty p t)
    | otherwise      = Nothing
    where
    go i p (Leaf _ a) = a
    go i p (Node _ l r)
        | p (i <> tag l) = go i p l
        | otherwise           = go (i <> tag l) p r
```

Arbori de căutare

Unificarea căutării

Int ca Size

```
instance Monoid Int where
    mempty  = 0
    mappend = (+)

win k t = search (>= k) t
```

Int ca Priority

```
instance Monoid Int where
    mempty  = maxBound
    mappend = min        -- Int ca Priority

win t = search (== tag t) t
```