Cursul 7 - 8 Plan

- Arbori binari augmentați
- Arbori binari de căutare
  - Căutare, Sortare, Inserare, Ștergere
- Arbori heap
- Tipuri de date abstracte (ADT)
  - Concepte de bază
  - Modulul – mecanism pentru implementarea unui adt
  - Exemplul1: Tipul abstract Queue
    - Specificare algebrică
    - Implementare
- Data.List, Data.Char, Data.Set, Data.Map,

- Studiu de caz: Arbori binari

Structura de data “arbori binari”

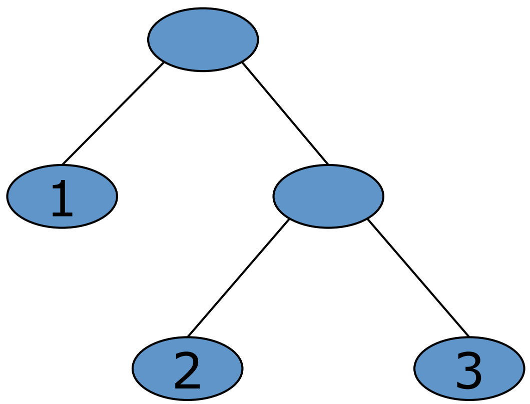
- O valoare a tipului **`Btree a`** este fie:
  - un nod frunză (Leaf) ce conține o valoare de tip **`a`**
  - un nod ramificație (Fork) şi doi noi arbori, subarborele stâng al nodului ramificație respectiv cel drept
  - o frunză se numeşte nod exterior
  - un nod ramificație se numeşte nod interior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

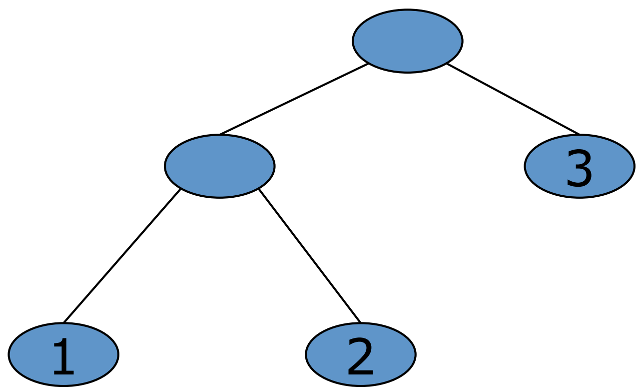
Structura de date arbori binari

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

```
        data Btree a = Leaf a | Fork(Btree a)(Btree a)
                               deriving (Show)
Leaf :: a -> Btree a
Fork :: Btree a -> Btree a -> Btree a

Fork(Leaf 1)(Fork(Leaf 2)(Leaf 3))
Fork(Fork(Leaf 1)(Leaf 2))(Leaf 3)
```





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)
```

- O idee 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 augmentați

- 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 augmentați

- 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 augmentați

- 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) 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))
```

Arbori binari heap

- minHeap – arbore binar în care cheia din fiecare nod este mai mică decât cheile din nodurile fii. Analog maxHeap
- Vom ilustra minHeap
- Structura de dată:

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

- Htree este virtual echivalent cu Stree: etichetele la constructorul Fork sunt plasate diferit
- Pentru a pune în evidență proprietatea heap se defineşte corespunzător funcția **`flatten`**

Arbori binari heap

- Funcția flatten:

```
flatten :: (Ord a) => Htree a -> [a]
flatten Null = []
flatten (Fork x xt yt) =
     x:merge (flatten xt) (flatten yt)

merge :: (Ord a) => [a] -> [a] -> [a]
merge [] ys = ys
merge (x:xs)[] = x:xs
merge (x:xs) (y:ys) = if x <= y then x:merge
  xs (y:ys) else y:merge (x:xs) ys
```

Arbori binari heap

- Exemplu:

```
Main> flatten( Fork 1 (Fork 2 Null Null)( Fork 3 Null Null))
[1,2,3]
Main> flatten( Fork 1 (Fork 3 Null Null)( Fork 2 Null Null))
[1,2,3]
Main> t1
Fork 1 (Fork 3 (Fork 4 Null Null)(Fork 5 Null Null))(Fork 2
   Null Null)
Main> flatten t1
[1,2,3,4,5]
Main> t2
Fork 15 (Fork 18 (Fork 29 Null Null) (Fork 25 Null Null))
   (Fork 29 Null Null)
Main> flatten t2
[15,18,25,29,29]
```

heap vs. căutare

- Arborii binari de căutare utili pentru :
  - Căutare eficientă
  - Inserare
  - Ştergere
- Arborii heap:
  - Aflarea min (respectiv max) în timp constant
  - Operație “merge” eficient (combinarea a doi arbori de cautare este ineficienta)
  - Heapsort

Construirea unui heap

- Se poate defini o nouă versiune a funcției mkBtree cu îndeplinirea condiției de heap. Nu se obține însă arbore complet
- Este posibilă construirea unui arbore binar heap dintr-o listă, în timp liniar
  - Se construieşte mai întâi un arbore binar de adâncime minimă (mkHtree)
  - Se rearanjează etichetele astfel încât să fie îndeplinită condiția heap (heapify)

Construirea unui heap

```
mkHtree :: (Ord a)=> [a] -> Htree a
mkHtree [] = Null
mkHtree (x:xs)
        | (m == 0)  = Fork x Null Null
        | otherwise = Fork x (mkHtree ys)(mkHtree zs)
        where   m = (length xs)
                (ys, zs) = splitAt (m`div`2) xs

mkHeap :: (Ord a) => [a] -> Htree a
mkHeap = heapify.mkHtree

heapify :: (Ord a) => Htree a -> Htree a
heapify Null = Null
heapify (Fork x xt yt) = sift x (heapify xt) (heapify yt)
```

- Funcția **`sift`** are 3 argumente (x xt yt); reconstruieşte arborele prin deplasarea lui x “în jos” până se obține proprietatea de heap

Construirea unui heap – funcția sift

```
sift :: (Ord a) => a -> Htree a -> Htree a -> Htree a

sift x Null Null = Fork x Null Null

sift x (Fork y a b) Null =
   if x <= y   then Fork x (Fork y a b) Null
               else Fork y (sift x a b) Null

sift x Null (Fork z c d) =
   if x <= z   then Fork x Null (Fork z c d)
               else Fork z Null (sift x c d)

sift x (Fork y a b) (Fork z c d)
   | x <= (y `min` z) = Fork x (Fork y a b) (Fork z c d)
   | y <= (x `min` z) = Fork y (sift x a b) (Fork z c d)
   | z <= (x `min` y) = Fork z (Fork y a b) (sift x c d)
```

Construirea unui heap – Exemple

```
Main> mkHtree["unu","doi","trei","patru"]
Fork "unu" (Fork "doi" Null Null) (Fork "trei" Null (Fork "patru" Null Null))

Main> mkHeap["unu","doi","trei","patru"]
Fork "doi" (Fork "unu" Null Null) (Fork "patru" Null (Fork "trei" Null Null))

Main> mkHtree[5,7,2,4,1,3,6]
Fork 5 (Fork 7 (Fork 2 Null Null) (Fork 4 Null Null)) (Fork 1 (Fork 3 Null
   Null (Fork 6 Null Null))
Main> flatten( mkHtree[5,7,2,4,1,3,6])
[5,1,3,6,7,2,4]

Main> mkHeap[5,7,2,4,1,3,6]
Fork 1 (Fork 2 (Fork 7 Null Null) (Fork 4 Null Null)) (Fork 3 (Fork 5 Null
   Null) (Fork 6 Null Null))

Main> flatten(mkHeap[5,7,2,4,1,3,6])
[1,2,3,4,5,6,7]
```

Heapsort

```
heapsort :: (Ord a) => [a] -> [a]
heapsort = flatten.mkHeap

Main> heapsort[3,1,3,5,2,1]
[1,1,2,3,3,5]
Main> heapsort ["unu", "doi","trei", "patru", "cinci",
   "sase","sapte", "opt", "noua"]
["cinci", "doi", "noua", "opt", "patru", "sapte", "sase",
   "trei", "unu"]
Main> heapsort [[1,2,4],[1,2,3],[1,2,1],[1,2,2]]
[[1,2,1],[1,2,2],[1,2,3],[1,2,4]]

Main> heapsort [[1,2,5,4],[2,3],[1,1,2,1],[1,2,2],[1],[3]]
[[1],[1,1,2,1],[1,2,2],[1,2,5,4],[2,3],[3]]
```

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
- Exemplu: Float este adt în Haskell pentru că nu este specificat modul de reprezentare al elementelor ci doar operațiile ce se aplică
- În general reprezentarea adt se poate schimba fără a afecta validitatea scripturilor ce utilizează acest adt

Exemplu - Coada

- Tipul abstract **`Queue a`** al cozilor peste un tip a

- O coadă este o listă specială: restricții privind operațiile

- Programatorul doreşte o implementare eficientă a acestui adt dar nu o realizează (încă!) ci se preocupă de descrierea operațiilor
  - Operațiile primitive – numele şi descrierea lor

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))
```

Coada - Specificare algebrică

- Aceste ecuații seamănă cu definițiile formale ale funcțiilor, bazate pe un tip de dată ce are constructorii empty şi join
- Nu se specifică nicăieri constrângerea de a implementa coada cu aceşti constructori; ecuațiile trebuiesc privite ca o exprimare a relațiilor între funcțiile adt
- Din ecuațiile de mai sus se deduce că orice coadă poate fi exprimată printr-un număr finit de aplicații ale operațiilor de join, front şi back aplicate cozii empty

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

Coada – Implementare(1)

- Reprezentarea cu liste finite
- Operațiile (le numim cu sufixul c pentru a le diferenția de cele abstracte):

```
emptyc :: [a]
emptyc = []
joinc :: a -> [a] -> [a]
joinc x xs = xs ++ [x]
frontc :: [a] -> a
frontc(x:xs) = x
backc :: [a] -> [a]
backc(x:xs) = xs
isEmptyc :: [a] -> Bool
isEmptyc xs = null xs
```

- Toate operațiile necesită timp constant, excepție joinc care se face în $\Theta(n)$ paşi

Coada – Implementare(1)

- Pentru a dovedi că sunt îndeplinite axiomele se observă că există o corespondență biunivocă între liste finite şi cozi:

```
abstr :: [a] -> Queue a
abstr = (foldr join empty).reverse

abstr []        = (foldr join empty).reverse []
                = foldr join empty [] = empty
abstr [x]       = (foldr join empty).reverse [x]
                = (foldr join empty) [x]
                = join x empty
abstr [x,y]     = foldr join empty.reverse [x, y]
                = foldr join empty [y,x]
                = join y (join x empty)
```

```
reprn :: Queue a -> [a]
reprn empty = []
reprn (join x xq) = reprn xq ++ [x]
```

```
reprn (join x empty)         = reprn empty ++ [x]
                             = [] ++ [x] = [x]
reprn (join x (join y empty)) = reprn (join y empty)
                    ++ [x] = [y] ++ [x] = [y, x]
```

$\texttt{reprn.abstr} = \texttt{id}_{\texttt{[a]}}$

$\texttt{abstr.reprn} = \texttt{id}_{\texttt{Queue a}}$

```
reprn.abstr [x,y]   = reprn(abstr [x,y])
                    = reprn(join y (join x empty))
                    = [x, y]
abstr.reprn (join x (join y empty)) =
abstr [y,x] = join x (join y empty)
```

Coada – Implementare(1)

- Să dovedim că au loc ecuațiile pentru `front`:

```
      front(join x empty) = x

front(join x empty) = front(join x []) = front [x] = x

      front (join x(join y xq)) = front (join y xq)

front (join x(join y xq)) = front (join x( xq ++ [y])) =
= front ( xq ++ [y] ++ [x]) = front (xq + [y])

front (join y xq) = front ( xq ++ [y])
```

Coada – Implementare(2)

- O coadă xq se reprezintă ca o pereche de liste (xs, ys) astfel ca elementele lui xq sunt elementele listei xs ++ reverse ys, cu o condiție în plus: dacă în perechea (xs, ys) ce reprezintă o coadă, xs este lista vidă atunci şi ys este lista vidă

- Nu orice pereche de liste reprezintă o coadă; de exemplu perechea ([], [1])

- Două perechi distincte de liste pot reprezenta aceeaşi coadă: ([1,2], []) şi ([1], [2]) reprezintă coada join 2(join 1 empty)

Coada – Implementare(2)

- Pentru implementare definim funcția abstr:

```
abstr :: ([a],[a]) -> Queue a
abstr (xs, ys) = (foldr join empty.reverse)(xs ++ reverse ys)
```

- Funcția abstr nu este injectivă: de exemplu se arată uşor că abstr([1,2], []) = abstr([1], [2])
- Formalizarea faptului că nu toate reprezentările sunt valide:

```
valid :: ([a],[a]) -> Bool
valid(xs, ys) = not(null xs) `or` null ys
```

- Funcția valid este un invariant al tipului de dată
- Perechea (abstr, valid) formalizează reprezentarea cozilor prin perechi de liste

Coada – Implementare(2)

- Implementare operațiilor:

```
emptyc = ([],[])
isEmptyc(xs, ys) = null xs
joinc x (ys,zs) = mkValid(ys, x:zs)
frontc(x:xs, ys) = x
backc(x:xs, ys) = mkValid(xs, ys)

mkValid :: ([a], [a]) -> ([a], [a])
mkValid (xs, ys) = if null xs then (reverse ys, [])
                                  else (xs, ys)
```

- Funcția `mkValid` menține invariantul tipului de dată
- Toate operațiile necesită timp constant exceptând **`back`** în cazul în care xs este **`[x]`**

Coada – Implementare(2)

- Verificarea specificării:
  - Faptul că o coadă poate avea mai multe reprezentări, verificarea axiomelor “mot-a-mot” duce la eşec. Axioma:

```
back(join x (join y xq)) = join x back (join y xq))
```

implică:

```
backc(joinc x (joinc y (joinc z emptyc))) =
backc(joinc x (backcc (joinc y (joinc z (joinc u
   emptyc)))))
```

adică:

```
      ([y,x], []) = ([y], [x])
```

ceea ce nu este adevărat! Dar cele 2 perechi reprezintă aceeaşi coadă!

Coada – Implementare(2)

- Verificarea specificării: axioma să conducă la faptul că cele 2 perechi rezultate, chiar dacă sunt diferite, reprezintă aceeaşi coadă:

```
abstr.backc.joinc x.joinc y =
= abstr.joinc x.backc.joinc y
```

Coada – Implementare(2)

- În general, pentru orice axiomă de forma f = g unde f şi g returnează cozi, trebuie să aibă loc

  abstr.fc = abstr.gc

  unde fc şi gc sunt rezultatele obținute prin înlocuirea operațiilor abstracte cu implementările lor. Dacă f şi g returnează altceva decât cozi nu se foloseşte abstr

- Axiomele trebuie verificate doar pentru reprezentări valide:

  abstr.fc = abstr.gc (modulo valid)

Coada – Implementare(2)

- Verificarea specificării, altă abordare: este suficient să aibă loc următoarele ecuații, modulo valid:

```
abstr emptyc = empty
abstr.joinc x = join x.abstr
abstr.frontc = front.abstr
abstr.backc = back.abstr
isEmptyc = isEmpty.abstr
```

- Odată verificate acestea se dovedeşte că au loc axiomele. De exemplu, pentru ultima axiomă:

```
abstr.backc.joinc x.joinc y = back.join x.join y.abstr
abstr.joinc x.backc.joinc y = join x.back.join y.abstr
```

iar

```
back.join x = join x.back
```

este adevărată (axioma 2 pentru back)

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

Exemplu: modulul Queue

```
module Queue(Queue, empty, isEmpty, join, front, back) where
newtype Queue a = MkQ([a], [a])
   --deriving (Show)

empty :: Queue a
empty = MkQ([], [])

isEmpty :: Queue a -> Bool
isEmpty(MkQ(xs, ys)) = null xs

join :: a -> Queue a -> Queue a
join x (MkQ(ys,zs)) = mkValid(ys, x:zs)

front :: Queue a -> a
front(MkQ(x:xs, ys)) = x

back :: Queue a -> Queue a
back(MkQ(x:xs, ys)) = mkValid(xs, ys)

mkValid :: ([a], [a]) -> Queue a
mkValid (xs, ys) = if null xs then MkQ(reverse ys, [])
           else MkQ(xs, ys)
```

Module - utilizare

- Utilizarea unui modul într-un script se face folosind o declarație import în acel script:

  import *Nume_modul*

- Exemplu: scriptul coada.hs

```
import Queue
toQ :: [a] -> Queue a
toQ = foldr join empty.reverse
fromQ :: Queue a -> [a]
fromQ q = if isEmpty q then [] else front q:fromQ(back q)

c1 :: Queue Int
c2 :: Queue [Char]
c1 = join 1(join 2( join 3( join 4(join 5 empty))))
c2 = join "ion" (join "vasile"(join "ana" empty))
```

Exemple

```
Main> join 1(join 2(join 3 empty))
MkQ ([3],[1,2])
Main> c1
MkQ ([5],[1,2,3,4])
Main> c2
MkQ (["ana"],["ion","vasile"])
Main> toQ [1,2,3,4,5]
MkQ ([1],[5,4,3,2])
Main> toQ ['a','b','c']
MkQ ("a","cb")
Main> join 8 c1
MkQ ([5],[8,1,2,3,4])
Main> join "horia" c2
MkQ (["ana"],["horia","ion","vasile"])
Main> front c2
"ana"
Main> front (back c2)
"vasile"
```

Exemple

```
Main> mkValid ([1,2,3], [4,5])
ERROR - Undefined variable "mkValid“
-- daca adaug mkValid în lista_export
Main> :r
Main> mkValid ([1,2,3], [4,5])
MkQ ([1,2,3],[4,5])
Main> mkValid ([], [2,4,5])
MkQ ([5,4,2],[])
Main> c1
MkQ ([5],[1,2,3,4])
Main> front c1
5
Main> let cc = back c1
Main> cc
MkQ ([4,3,2,1],[])
Main> isEmpty cc
False
```

Modules, **Loading modules**

- The syntax for importing modules in a Haskell script is **import <module name>.**
- One script can, of course, import several modules. Just put each import statement into a separate line.
- When you do import Data.List, all the functions that Data.List exports become available in the global namespace

Modules, **Loading modules**

```
Prelude> nub [2,3,2,3,4,3,2,4,4,2,4]
<interactive>:2:1: Not in scope: ‘nub’
Prelude> :m + Data.List
Prelude Data.List>nub [2,3,2,4,3,2,4,2,4]
[2,3,4]
```

- Loading modules in GHCi:

Prelude> :m + Data.Char

Prelude Data.Char> :m + Data.List Data.Map Data.Set

Prelude Data.Char Data.List Data.Map Data.Set>

Modules, **Loading modules**

- If you just need a couple of functions from a module, you can selectively import just those functions:

```
import Data.List (nub, sort)
```

- That's often useful when several modules export functions with the same name and you want to get rid of the offending ones:

```
import Data.List hiding (nub)
```

Modules, **Loading modules**

- Another way of dealing with name clashes is to do qualified imports:

  import qualified Data.Map

- This makes it so that if we want to reference Data.Map's filter function, we have to do Data.Map.filter

  import qualified Data.Map as M

- Now, to reference Data.Map's filter function, we just use M.filter

Data.List

- the Prelude module exports some functions from Data.List (map, filter, etc)

- **intersperse** takes an element and a list and then puts that element in between each pair of elements in the list:

Prelude Data.List> intersperse '.' "MONKEY"

"M.O.N.K.E.Y"

Prelude Data.List> intersperse 0 [1,2,3,4,5,6]

[1,0,2,0,3,0,4,0,5,0,6]

- **intercalate** takes a list of lists and a list. It then inserts that list in between all those lists and then flattens the result:

```
Prelude Data.List> intercalate " " ["hey","there","guys"]
"hey there guys"
Prelude Data.List> intercalate [0,0,0] [[1,2,3],[4,5,6],[7,8,9]]
[1,2,3,0,0,0,4,5,6,0,0,0,7,8,9]
```

- **transpose** transposes a list of lists. If you look at a list of lists as a 2D matrix, the columns become the rows and vice versa:

```
Prelude Data.List> transpose [[1,2,3],[4,5,6],[7,8,9]]
[[1,4,7],[2,5,8],[3,6,9]]
Prelude Data.List> transpose ["hey","there","guys"]
["htg","ehu","yey","rs","e"]
```

- **iterate** takes a function and a starting value. It applies the function to the starting value, then it applies that function to the result, then it applies the function to that result again, etc. It returns all the results in the form of an infinite list.

```
Prelude Data.List> take 10 (iterate (*2) 5)
[5,10,20,40,80,160,320,640,1280,2560]
Prelude Data.List> take 10 $ iterate (*2) 5
[5,10,20,40,80,160,320,640,1280,2560]
```

takeWhile, dropWhile

```
Prelude Data.List> takeWhile (>3)
  [6,5,4,3,2,1,2,3,4,5,4,3,2,1]
[6,5,4]
Prelude Data.List> dropWhile (/=' ') "This is a
  sentence"
" is a sentence”
Prelude Data.List> sum $ takeWhile (<10000) $ map
  (^3) [1..]
53361

Prelude Data.List> let stock = [(994.4,2008,9,1),
  (995.2,2008,9,2),(999.2,2008,9,3),
  (1001.4,2008,9,4),(998.3,2008,9,5)]
Prelude Data.List> head (dropWhile (\(val,y,m,d) ->
  val < 1000) stock)
(1001.4,2008,9,4)
```

break, span

```
Prelude Data.List> break (==4)
  [1,2,3,4,5,6,7]
([1,2,3],[4,5,6,7])
Prelude Data.List> span (==4)
  [1,2,3,4,5,6,7]
([],[1,2,3,4,5,6,7])
Prelude Data.List> span (/=4)
  [1,2,3,4,5,6,7]
([1,2,3],[4,5,6,7])
```

- **break p** equivalent of **span (not . p).**

sort, group, find

```
Prelude Data.List> sort [8,5,3,2,1,6,4,2]
[1,2,2,3,4,5,6,8]
Prelude Data.List> sort "This will be sorted soon"
"    Tbdeehiillnooorssstw”
Prelude Data.List> group [1,1,1,1,2,2,2,2,3,3,2,2,2,5,6,7]
[[1,1,1,1],[2,2,2,2],[3,3],[2,2,2],[5],[6],[7]]
Prelude Data.List> group.sort $
   [1,1,2,7,5,7,2,2,1,1,2,2,7,6,2,2,3,3,2,2,2,5,6,7]
[[1,1,1,1],[2,2,2,2,2,2,2,2,2,2],[3,3],[5,5],[6,6],
   [7,7,7,7]]
Prelude Data.List> :i find
find :: (a -> Bool) -> [a] -> Maybe a -- Defined in
   ‘Data.List’
Prelude Data.List> find (>4) [1,2,3,4,5,6]
Just 5
Prelude Data.List> find (>44) [1,2,3,4,5,6]
Nothing
```

elem, notElem, elemIndex, elemIndices

```
Prelude Data.List> elem 2 [1,2,3,4,5]
True
Prelude Data.List> notElem 2 [1,2,3,4,5]
False
Prelude Data.List> 4 `elemIndex` [1,2,3,4,5,6]
Just 3
Prelude Data.List> 10 `elemIndex` [1,2,3,4,5,6]
Nothing

Prelude Data.List> ' ' `elemIndices`
  "Facultatea de Informatica"
[10,13]
Prelude Data.List> ' ' `elemIndices`
  "FacultateadeInformatica"
[]
```

delete, \\, union, intersect, insert

```
Prelude Data.List> delete 6 [1,2,1,2,6,3,5,6]
[1,2,1,2,3,5,6]
Prelude Data.List> [1..10] \\ [2,5,9]
[1,3,4,6,7,8,10]
Prelude Data.List> [1,2,1,2,6,3,5,6]\\[6]
[1,2,1,2,3,5,6]
Prelude Data.List> [1,2,1,2,6,3,5,6]\\[6, 6]
[1,2,1,2,3,5]
Prelude Data.List> [1..7] `union` [5..10]
[1,2,3,4,5,6,7,8,9,10]
Prelude Data.List> [1..7] `intersect` [5..10]
[5,6,7]
Prelude Data.List> insert 4 [3,5,1,2,8,2]
[3,4,5,1,2,8,2]
Prelude Data.List> insert 4 [1,3,4,4,1]
[1,3,4,4,4,1]
Prelude Data.List> insert 4 [1,2,3,5,6,7]
[1,2,3,4,5,6,7]
```

Data.Char

isControl , isSpace , isLower , isUpper , isAlpha , isAlphaNum , isPrint , isDigit , isOctDigit , isHexDigit , isLetter , isMark , isNumber , isPunctuation , isSymbol , isSeparator , isAscii , isLatin1 , isAsciiUpper , isAsciiLower, toUpper, toLower, toTitle, digitToInt, intToDigit, ord, chr

```
Prelude> all isAlphaNum "bobby283”
<interactive>:2:5: Not in scope: ‘isAlphaNum’
Prelude> :m + Data.Char
Prelude Data.Char> all isAlphaNum "bobby283”
True
Prelude Data.Char> all isAlphaNum "eddy the fish!”
False
Prelude Data.Char> map ord "abcdefgh”
[97,98,99,100,101,102,103,104]
Prelude Data.Char> chr 97
'a'
```

Data.Map

- **fromList** function takes an association list (in the form of a list) and returns a map with the same associations.

Prelude Map> Map.fromList [("betty","555-2938"), ("bonnie","452-2928"),("lucille","205-2928")]

fromList [("betty","555-2938"), ("bonnie","452-2928"),("lucille","205-2928")]

- **empty**

Prelude Map> Map.empty

fromList []

insert, null, size

```
Prelude Map> let xm = Map.insert 3 100 Map.empty
Prelude Map> xm
fromList [(3,100)]
Prelude Map> Map.insert 5 600 (Map.insert 4 200 ( Map.insert 3 100
   Map.empty))
fromList [(3,100),(4,200),(5,600)]
Prelude Map> Map.insert 5 600 (Map.insert 4 200 ( Map.insert 3 100
   xm))
fromList [(3,100),(4,200),(5,600)]
Prelude Map> Map.null xm
False
Prelude Map> Map.size xm
1
Prelude Map> xm
fromList [(3,100)]
Prelude Map> let ym = Map.insert 5 600 (Map.insert 4 200 ( Map.insert
   3 100  xm))
Prelude Map> ym
fromList [(3,100),(4,200),(5,600)]
Prelude Map> Map.size ym
3
```

singleton, lookup, member, map, filter

```
Prelude Map> Map.singleton 3 9
fromList [(3,9)]
Prelude Map> Map.lookup 9 (Map.insert 5 9 $
  Map.singleton 3 9)
Nothing
Prelude Map> Map.lookup 3 (Map.insert 5 9 $
  Map.singleton 3 9)
Just 9
Prelude Map> Map.member 3 $ Map.fromList [(3,6),
  (4,3),(6,9)]
True
Prelude Map> Map.map (*100) $ Map.fromList [(1,1),
  (2,4),(3,9)]
fromList [(1,100),(2,400),(3,900)]
Prelude Map Data.Char> Map.filter isUpper $
  Map.fromList [(1,'a'),(2,'A'),(3,'b'),(4,'B')]
fromList [(2,'A'),(4,'B')]
```

toList, keys, elems, fromListWith, insertWith

```
Prelude Map Data.Char> Map.toList . Map.insert 9 2 $
  Map.singleton 4 3
[(4,3),(9,2)]
Prelude Map Data.Char> Map.keys ym
[3,4,5]
Prelude Map Data.Char> Map.elems ym
[100,200,600]
Prelude Map Data.Char> Map.fromListWith max [(2,3),
  (2,5),(2,100),(3,29),(3,22),(3,11),(4,22),(4,15)]
fromList [(2,100),(3,29),(4,22)]
Prelude Map Data.Char> Map.fromListWith (+) [(2,3),
  (2,5),(2,100),(3,29),(3,22),(3,11),(4,22),(4,15)]
fromList [(2,108),(3,62),(4,37)]
Prelude Map Data.Char> Map.insertWith (+) 3 100 $
  Map.fromList [(3,4),(5,103),(6,339)]
fromList [(3,104),(5,103),(6,339)]
```

Data.Set

```
Prelude Set> let text = "Facultatea de
  Informatica”
Prelude Set> let set1 = Set.fromList text
Prelude Set> set1
fromList " FIacdefilmnortu”
Prelude Set> let set2 = Set.fromList
  [1,4,2,3,2,1,5,3,2,4,3]
Prelude Set> set2
fromList [1,2,3,4,5]
Prelude Set> Set.fromList [True, True, False]
fromList [False,True]
Prelude Set> Set.fromList ["True", "True",
  "False"]
fromList ["False","True"]
```

intersection, difference, union

```
Prelude Set> let set3 = Set.fromList
   [1,2,3,2,1,6,3,2,3,7]
Prelude Set> set3
fromList [1,2,3,6,7]
Prelude Set> set2
fromList [1,2,3,4,5]
Prelude Set> Set.intersection set2 set3
fromList [1,2,3]
Prelude Set> Set.difference set2 set3
fromList [4,5]
Prelude Set> Set.difference set3 set2
fromList [6,7]
Prelude Set> Set.union set2 set3
fromList [1,2,3,4,5,6,7]
```

null, size, member, empty, singleton, insert and delete

```
Prelude Set> Set.null Set.empty
True
Prelude Set> Set.null $ Set.fromList [3,4,5,5,4,3]
False
Prelude Set> Set.size $ Set.fromList [3,4,5,3,4,5]
3
Prelude Set> Set.singleton 9
fromList [9]
Prelude Set> Set.insert 4 $ Set.fromList [9,3,8,1]
fromList [1,3,4,8,9]
Prelude Set> Set.insert 8 $ Set.fromList [5..10]
fromList [5,6,7,8,9,10]
Prelude Set> Set.delete 4 $ Set.fromList [3,4,5,4,3,4,5]
fromList [3,5]
```

isSubsetOf, isProperSubsetOf, map, filter

```
Prelude Set> Set.fromList [2,3,4] `Set.isSubsetOf`
  Set.fromList [1,2,3,4,5]
True
Prelude Set> Set.fromList [1,2,3,4,5]
  `Set.isProperSubsetOf` Set.fromList [1,2,3,4,5]
False
Prelude Set> Set.fromList [2,3,4,8] `Set.isSubsetOf`
  Set.fromList [1,2,3,4,5]
False
Prelude Set> Set.filter odd $ Set.fromList
  [3,4,5,6,7,2,3,4]
fromList [3,5,7]
Prelude Set> Set.map (+1) $ Set.fromList
  [3,4,5,6,7,2,3,4]
fromList [3,4,5,6,7,8]
```

Making our own modules

```
--Geometry.hs

module Geometry ( sphereVolume , sphereArea , cubeVolume , cubeArea , cuboidArea , cuboidVolume ) where
sphereVolume :: Float -> Float
sphereVolume radius = (4.0 / 3.0) * pi * (radius ^ 3)
sphereArea :: Float -> Float
sphereArea radius = 4 * pi * (radius ^ 2)
cubeVolume :: Float -> Float
cubeVolume side = cuboidVolume side side side
cubeArea :: Float -> Float
cubeArea side = cuboidArea side side side
cuboidVolume :: Float -> Float -> Float -> Float
cuboidVolume a b c = rectangleArea a b * c
cuboidArea :: Float -> Float -> Float -> Float
cuboidArea a b c = rectangleArea a b * 2 + rectangleArea a c * 2 + rectangleArea c b * 2
rectangleArea :: Float -> Float -> Float  rectangleArea a b = a * b
```

Submodules: Sphere.hs, Cuboid.hs, Cube.hs

```
--Sphere.hs

module Geometry.Sphere  ( volume  , area  )
where
volume :: Float -> Float
volume radius = (4.0 / 3.0) * pi * (radius ^ 3)
area :: Float -> Float
area radius = 4 * pi * (radius ^ 2)


import qualified Geometry.Sphere as Sphere
import qualified Geometry.Cuboid as Cuboid
import qualified Geometry.Cube as Cube
```