Cursul 7

- Arbori heap
- Tipuri de date abstracte (ADT)
  - Concepte de bază
  - Modulul – mecanism pentru implementarea unui adt
  - Exemplul1: Tipul abstract Queue
    - Specificare algebrică
    - Implementare

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 evidenţă proprietatea heap se defineşte corespunzător funcţia flatten

Arbori binari heap

- Funcţia 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]
```

Arbori binari heap vs. arbori binari de căutare

- Arborii binari de căutare :
  - Căutare eficientă
  - Inserare
  - Ştergere
- Arborii heap:
  - Aflarea min (respectiv max) în timp constant
  - Operaţie "merge" eficient
  - Heapsort

Construirea unui heap

- Se poate defini o nouă versiune a funcţiei mkBtree cu îndeplinirea condiţiei de heap. Nu se obţine 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 condiţia 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)
```

- Funcţia `sift` are 3 argumente (x xt yt); reconstruieşte arborele prin deplasarea lui x "în jos" până se obţine proprietatea de heap

Construirea unui heap - funcţia 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 declaraţie **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 operaţiile se numeşte *tip concret*
- Un tip abstract de date este definit prin specificarea operaţiilor 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 operaţiile 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: restricţii privind operaţiile
- Programatorul doreşte o implementare eficientă a acestui adt dar nu o realizează (încă!) ci se preocupă de descrierea operaţiilor
  - Operaţiile primitive – numele şi descrierea lor

Coada

- Operaţiile 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`
- Semnificaţia
- Lista operaţiilor împreună cu tipul acestora reprezintă signatura tipului de dată abstract

Coada

- Specificare algebrică (axiomatică): o listă de axiome ce trebuie să fie satisfăcute de operaţii
- 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 ecuaţii seamănă cu definiţiile formale ale funcţiilor, bazate pe un tip de dată ce are costructorii empty şi join
- Nu se specifică nicăeri constrângerea de a implementa coada cu aceşti constructori; ecuaţiile trebuiesc privite ca o exprimare a relaţiilor între funcţiile adt
- Din ecuaţiile de mai sus se deduce că orice coadă poate fi exprimată printr-un număr finit de aplicaţii ale operaţiilor de join şi back aplicate cozii empty

ADT - Implementare

- Implementarea adt înseamnă furnizarea unei reprezentări pentru valorile sale, definirea opereţiilor în termenii acestei reprezentări şi dovedirea faptului că operaţiile implementate satisfac specificaţiile algebrice
- Cel ce implementează adt este liber în a alege dintre posibilele reprezentări, în funcţie de :
  - eficienţă
  - simplicitate
  - ...gusturi

Coada - Implementare(1)

- Liste finite
- Operaţiile (le numim cu sufixul c pentru a le diferenţia 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 operaţiile necesită timp constant, excepţie joinc care se face în $\Theta(n)$ paşi

Coada - Implementare(1)

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

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

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

$reprn.abstr = id_{[a]}$

$abstr.reprn = id_{Queue\ a}$

Coada - Implementare(1)

- Să dovedim că au loc ecuaţiile 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 ys sunt elementele listei xs ++ reverse ys, cu o condiţie 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 funcţia abstr:

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

- Funcţia 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
```

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

Coada - Implementare(2)

- Implementare operaţiilor:

```
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)
```

- Funcţia menţine invariantul tipului de dată
- Toate operaţiile 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`

- 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 obţinute prin înlocuirea operaţiilor 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 (modulo valid) = abstr.gc (modulo valid)

Coada - Implementare(2)

- Verificarea specificării, altă abordare: este suficient să aibă loc următoarele ecuaţii, 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:

```
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* conţine:
  - Numele tipului abstract de date – acelaşi cu numele modulului
  - Numele operaţiilor
  - 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 declaraţie 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],[])
```