Funkcionális programozás
10. előadás

MÁRTON Gyöngyvér

Sapientia Egyetem, Matematika-Informatika Tanszék
Marosvásárhely, Románia
mgyongyi@ms.sapientia.ro

2023, tavaszi félév

Miről volt szó?

- rekord típusok, tárolás szövegállományban
- algebrai adattípusok
- feladatok (rekord típusok):
  - maximum keresés
  - rendezés, a rendezési feltétel mint függvényparaméter
- feladatok (algebrai adattípusok):
  - összeszámolás, kiválogatás
  - kiíratási lehetőségek

Miről lesz szó?

- paraméterezett típusok
- rekurzív típusok
- bináris keresőfák
  - létrehozás
  - bejárások: inorder, preorder, postorder
  - maximum elem, a bináris keresőfa mélysége
- kombinatorikai feladatok
  - `n` elem `m`-ed rendű kombinációi
  - az `n` királynő feladat

Paraméterezett típusok

- valamely adat vagy adatszerkezet típusának definiálásakor is használhatunk típusváltozót, azaz valamely típuskonstruktor is lehet paraméterezett
- a legismertebb paraméterezett adattípus a `Maybe`, ami azt jelenti, hogy alkalmazható tetszőleges típusú adat vagy adatszerkezet esetében:

```
data Maybe a = Nothing | Just a
    deriving (Show)
```

- paraméterezett adattípust mi is definiálhatunk, például létrehozhatunk egy saját `Maybe` típust

```
data MyMaybe a = Semmi | Ertek a
    deriving (Show)
```

- a korábban megírt `init` függvényt is átírhatjuk, használva a `MyMaybe` típust:

```
initMyMaybe :: [a] -> MyMaybe [a]
initMyMaybe = foldr op Semmi
    where
    op k Semmi = Ertek []
    op k (Ertek ve) = Ertek (k : ve)

> initMyMaybe ""
Semmi

> initMyMaybe "Retyezat Nemzeti Park"
Ertek "Retyezat Nemzeti Par"
```

Rekurzív típusok

- lehetőség van rekurzív adatszerkezetek definiálására is, például a lista adatszerkezetet is meg lehet adni rekurzív formában:

```
data MyList a = Ures | Fuz a (MyList a)
    deriving (Show)
```

- a `MyList` típusú adat kétfajta értékkonstruktorral rendelkezik. Az első azt az esetet kezeli, amikor a lista üres (`Ures`), a második pedig azt az esetet, amikor egy `MyList` típusú adatot úgy hozunk létre, hogy kombinálunk egy tetszőleges típusú a elemet és egy tetszőleges típusú `MyList` elemet
- függvényeket is írhatunk a `MyList` típusra:

```
myLength :: Num t => MyList t1 -> t
myLength Ures = 0
myLength (Fuz k ve) = 1 + myLength ve

> ls1 = Fuz 4 Ures
> ls2 = Fuz 3 (Fuz 4 Ures)
> ls3 = Fuz 1 $ Fuz 2 ls2
> ls3
Fuz 1 (Fuz 2 (Fuz 3 (Fuz 4 Ures)))

> myLength ls3
4
```

Bináris fák

- egy bináris fa szerkezetét a következő rekurzív típusdeklarációval lehet megadni (a bináris fa legtöbbször rekurzívan van értelmezve):

```
data BinFa a = UresFa | Csomop a (BinFa a) (BinFa a)
    deriving (Show, Read, Eq)
```

- a listával ellentétben, amely egy lineáris struktúra, a bináris fa egy hierarchikus szerkezet, éppen ezért az adatokat kezelő algoritmusok időigénye jobb: a hozzáadás, keresés, törlés stb. hatékonyan megvalósítható
- bináris fák esetében egy adatból, azaz egy csomópontból maximum két további adatot (csomópontot) tudunk elérni
- a definiált `BinFa` egy olyan szerkezet, amely két értékkonstruktorral rendelkezik,
  - az egyik esetben az értékkonstruktor üres értéket definiál: `UresFa`
  - a másik értékkonstruktornak a `Csomop` csomópontbeli értéke mellett még két `BinFa` típusú mezője is van
  - a `BinFa` egy paraméterezett típus lesz, ami azt jelenti, hogy a csomópontokban tetszőleges típusú adat tárolható

Bináris keresőfák

1. feladat

Írjunk egy-egy Haskell függvényt, amely segítségével egy egyelemű bináris fát hozunk létre, illetve beszúrunk egy új csomópontot egy bináris fába.

```
egyelemuFa :: Ord a => a -> BinFa a
egyelemuFa x = Csomop x UresFa UresFa

beszurFa :: Ord a => a -> BinFa a -> BinFa a
beszurFa x UresFa = egyelemuFa x
beszurFa x (Csomop a bal jobb)
    | x <= a = Csomop a (beszurFa x bal) jobb
    | x > a = Csomop a bal (beszurFa x jobb)
```

- a típusdeklarációkból látható, hogy az adatok, amelyeket a csomópontokba teszünk, azok az `Ord` típusosztályhoz kell tartozzanak
- a beszúrás során a bal oldali ághoz csatoljuk azt a csomópontot, ami a szülő csomópontban levő értékhez képest kisebb értéket tárol, és a jobb oldalihoz a nagyobbat
- a beszúrási algoritmus tulajdonképpen egy bináris keresőfát épít
- az adatok rendezett sorrendjét valamely bejárási algoritmussal hatékonyan meg lehet határozni

Bináris keresőfák

- a `letrehoz` függvény egy 4 csomópontból álló bináris keresőfát épít, ahol a csomópontokba `String` típusú adatok kerülnek:

```
letrehoz = do
    let f1 = beszurFa "mari" UresFa
    let f2 = beszurFa "zsuzsa" f1
    let f3 = beszurFa "mari" f2
    let f4 = beszurFa "feri" f3
    print f4

> letrehoz
Csomop "mari" (Csomop "mari" (Csomop "feri" UresFa UresFa)
    UresFa) (Csomop "zsuzsa" UresFa UresFa)
```

- a csomópontok száma azért lesz 4, mert a `"mari"` adat kétszer is szerepelni fog a keresőfában
- ha azt szeretnénk, hogy a csomópontok egyedi értékeket tároljanak, akkor a `beszurFa` a következőképpen módosul:

```
beszurFa_ :: Ord a => a -> BinFa a -> BinFa a
beszurFa_ x UresFa = egyelemuFa x
beszurFa_ x (Csomop a bal jobb)
   | x == a = Csomop x bal jobb
   | x < a = Csomop a (beszurFa_ x bal) jobb
   | x > a = Csomop a bal (beszurFa_ x jobb)
```

Bináris keresőfák

- a `letrehozFa` függvény a `beszurFa` függvény alkalmazásával listában megadott elemek alapján hozza létre a bináris keresőfát
- a függvény bemeneti paramétere egy olyan lista kell legyen, amelynek elemei az `Ord` típusosztályhoz tartoznak, kimenete pedig egy `BinFa a` típusú adat lesz

```
letrehozFa :: (Ord a) => [a] -> BinFa a
letrehozFa [] = UresFa
letrehozFa (k : ve) = beszurFa k $ letrehozFa ve

> letrehozFa ["mari", "zsuzsa", "feri", "mari"]
Csomop "mari" (Csomop "feri" UresFa (Csomop "mari"
    UresFa UresFa)) (Csomop "zsuzsa" UresFa UresFa)
```

Bináris keresőfák

- ha lépésenkét szeretnénk látni, ahogy a listaelemek alapján épül a fa, akkor módosíthatjuk a fenti függvényt:

```
letrehozFaIr :: (Ord a, Show a) => [a] -> IO (BinFa a)
letrehozFaIr [] = return UresFa
letrehozFaIr (k : ve) = do
        rFa <- letrehozFaIr ve
        putStr $ show rFa ++ "\n"
        return $ beszurFa k rFa

> letrehozFaIr ["mari", "zsuzsa", "feri", "mari"]
UresFa
Csomop "mari" UresFa UresFa
...
```

Bináris keresőfák

- listaelemek alapján egy bináris fa felépítésének legegyszerűbb kódsora a `foldr` függvény használatával adható meg:

```
letrehozFaFold :: Ord a => [a] -> BinFa a
letrehozFaFold = foldr beszurFa UresFa

> letrehozFaFold ["mari", "zsuzsa", "feri", "mari"]
Csomop "mari" (Csomop "feri" UresFa (Csomop "mari"
    UresFa UresFa)) (Csomop "zsuzsa" UresFa UresFa)
```

Bináris keresőfák

Egy bináris fa elemeit három különböző bejárási sorrend szerint lehet kiíratni: preorder, inorder, posztorder:

- a preoder esetében először a szülő, utána a bal oldali, majd a jobb oldali csomópontot kell érinteni
- az inoder esetében először a bal oldali, utána a szülő, majd a jobb oldali csomópontot kell érinteni
- a posztorder esetében először a bal oldali, utána a jobb oldali, majd a szülő csomópontot kell érinteni

Az inorder bejárási sorrend rendezett sorrendben adja meg az elemeket:

```
inorderFa :: (Ord a) => BinFa a -> [a]
inorderFa UresFa = []
inorderFa (Csomop a bal jobb) =
        inorderFa bal ++ [a] ++ inorderFa jobb

> f = letrehozFaFold ["mari", "zsuzsa", "feri", "mari", "kati", "laci"]
> inorderFa f
["feri","kati","laci","mari","mari","zsuzsa"]
```

Bináris keresőfák

A preorder, illetve posztorder bejárásokat a `preorderFa`, illetve `postorderFa` függvények végzik. Habár ismert a függvények hatékonyabb változata, mégis a következő implementációkat adjuk meg, mert ezek felépítése a korábban megadott értelmezéseket követik.

```
preorderFa :: (Ord a) => BinFa a -> [a]
preorderFa UresFa = []
preorderFa (Csomop a bal jobb) = [a] ++ preorderFa bal
        ++ preorderFa jobb

postorderFa :: (Ord a) => BinFa a -> [a]
postorderFa UresFa = []
postorderFa (Csomop a bal jobb) = postorderFa bal
        ++ postorderFa jobb ++ [a]
```

Bináris keresőfák

- egy bináris keresőfa legjobboldalibb eleme a legnagyobb eleme lesz a fának, meghatározását `maximumFa` végzi:

```
maximumFa :: BinFa a -> Maybe a
maximumFa UresFa = Nothing
maximumFa (Csomop a bal UresFa) = Just a
maximumFa (Csomop a bal jobb) = maximumFa jobb

> maximumFa f
Just "zsuzsa"
```

- egy bináris fa mélysége elsősorban a keresési, illetve bejárási algoritmusok hatékonyságát befolyásolja
- a `melysegFa` meghatározza a bináris fa mélységét, ahol két elem maximumának a meghatározásához a `max` könyvtárfüggvényt használjuk:

```
melysegFa :: BinFa a -> Int
melysegFa UresFa = 0
melysegFa (Csomop a bal jobb) = 1 + max (melysegFa bal) (melysegFa jobb)

> melysegFa f
3
```

Kombinatorikai feladatok

2. feladat

Írjunk egy Haskell-függvényt, amely meghatározza az összes olyan `m` hosszúságú listát, amely egy bemeneti lista elemeiből képezhető.

- az `lGen_` halmazműveleteket alkalmaz, létrehozza azokat a listákat, amelyek első elemei rendre a bemeneti lista elemei lesznek, a többi elemet pedig rekurzívan generálja
- a rekurzív hívásban a függvény első paramétere az eredeti lista lesz, a második paramétert pedig minden egyes alkalommal csökkentjük eggyel

```
lGen_ :: (Eq a) => [a] -> Int -> [[a]]
lGen_ ls 0 = [[]]
lGen_ ls m = [k : ve | k <- ls, ve <- lGen_ ls (m-1)]

> lGen_ [0, 1] 4
[[0,0,0,0],[0,0,0,1],[0,0,1,0],[0,0,1,1],[0,1,0,0],[0,1,0,1],[0,1,1,0],
[0,1,1,1],[1,0,0,0],[1,0,0,1],[1,0,1,0],[1,0,1,1],[1,1,0,0],[1,1,0,1],
[1,1,1,0],[1,1,1,1]]

> lGen_ "ab" 3
["aaa","aab","aba","abb","baa","bab","bba","bbb"]
```

Kombinatorikai feladatok

- a következő `lGen`-ben módosítjuk a listaelemek generálási módjának a sorrendjét
- először generáljuk a `ve` listaelemeket, és csak ezután adjuk meg a `k` értékének a generálási módját:

```
lGen :: (Eq a) => [a] -> Int -> [[a]]
lGen ls 0 = [[]]
lGen ls m = [k : ve | ve <- lGen ls (m-1), k <- ls]
```

- a módosítás a listaelemek más sorrendjét fogja eredményezni:

```
> lGen "ab" 3
["aaa","baa","aba","bba","aab","bab","abb","bbb"]
```

- az algoritmus hatékonyságát is javítottuk, amelyet a következő oldalon megadott `foGen` és a `foGen_` időigényei közötti lényeges különbség jelez

Kombinatorikai feladatok

```
foGen :: (Show a, Eq a) => [a] -> Int -> IO ()
foGen ls m = do
    let rLs = lGen ls m
    print $ last rLs

> :set +s
> foGen [0,1] 20
[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]
(0.67 secs, 352,405,392 bytes)

foGen_ :: (Show a, Eq a) => [a] -> Int -> IO ()
foGen_ ls m = do
    let rLs = lGen_ ls m
    print $ last rLs

> foGen_ [0,1] 20
[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]
(3.93 secs, 3,120,641,320 bytes)
```

- a `foGen`, illetve `foGen_` csak a kigenerált lista utolsó elemét írja ki, így a függvények időigényének értékét nem befolyásolta a listaelemek kiíratásának időigénye

n elem m-ed rendű kombinációi

3. feladat

Írjunk egy Haskell-függvényt, amely meghatározza n elem m-ed rendű kombinációit.

```
komb :: (Ord a) => [a] -> Int -> [[a]]
komb ls 0 = [[]]
komb ls m = [k : ve |
        ve <- komb ls (m-1),
        k <- ls, feltKomb k ve]

feltKomb :: (Ord a) => a -> [a] -> Bool
feltKomb x [] = True
feltKomb x (k : ve)
    | k <= x = False
    | otherwise = feltKomb x ve

> komb [4, 1, 7, 9] 3
[[1,4,7],[1,4,9],[4,7,9],[1,7,9]]

> komb "wxyz" 2
["wx","wy","xy","wz","xz","yz"]
```

n elem m-ed rendű kombinációi

- a `komb` az `lGen`-en alapszik
- a `komb` paraméterként egy listát és az `m` értéket kapja meg, ahol a lista elemszáma adja a feladat megfogalmazásában szereplő `n` értéket
- mivel a kigenerált listák nem mindegyike felel meg a feladat kritériumának, ezért egy tesztelő `feltKomb` függvényt is alkalmazunk
- a `feltKomb` kimenete akkor lesz `True` ha a bemeneti lista elemei szigorúan növekvő sorrendben vannak
- például a fenti bemenet esetében az eredmény lista nem tartalmazhatja az `[1, 4, 7]`, `[1, 7, 4]`, `[7, 4, 1]` stb. mindegyikét, csak az `[1, 4, 7]`-t fogja tartalmazni
- a `feltKomb`-ban, ha módosítjuk a feltételt `k >= x = False`-ra, akkor a `[7, 4, 1]` fog szerepelni az eredménylistában
- a következő kódsor a `feltKomb` egy kompaktabb változata:

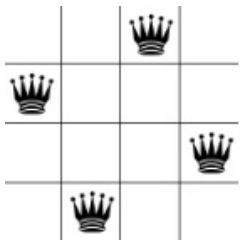
```
feltKomb_ :: (Ord a) => a -> [a] -> Bool
feltKomb_ x = all (aux x)
    where
    aux x k = k > x
```

Az n királynő feladat

4. feladat

Írjunk egy Haskell-függvényt, amely elhelyez egy sakktáblán 8 királynőt úgy, hogy azok ne üssék egymást. Általánosan is oldjuk meg a feladatot, azaz helyezzünk el egy nxn-es sakktáblán n királynőt úgy, hogy azok ne üssék egymást.

- a gyakorlatban egy `nxn`-es sakktáblán, ha el akarunk helyezni `n` királynőt úgy, hogy azok ne üssék egymást, ez azt fogja jelenteni, hogy nem tehetjük őket ugyanabba a sorba, oszlopba, illetve átlóra
- a megoldás listák kigenerálását fogja jelenti:
  - a listaelemek egész számok lesznek
  - az első listaelem az első sorban levő királynő oszlop-pozícióját fogja jelenteni, a második listaelem a második sorban levő királynő oszlop-pozícióját jelenti, és így tovább
- egy 4x4-es sakktáblán `[3, 1, 4, 2]` helyes megoldás azt fogja jelenteni, hogy a királynőt az első sorban a harmadik, a második sorban az első, a harmadik sorban a negyedik és a negyedik sorban a második oszlopba tettük:



Az n királynő feladat

```
kiralyno :: Int -> [[Int]]
kiralyno n = auxK n n
    where
    auxK :: Int -> Int -> [[Int]]
    auxK n 0 = [[]]
    auxK n m = [k : ve |
            ve <- auxK n (m-1),
            k <- [1..n], feltKir k ve ]
feltKir :: Int -> [Int] -> Bool
feltKir x ls = auxFeltK 1 x ls
    where
    auxFeltK :: Int -> Int -> [Int] -> Bool
    auxFeltK i x [] = True
    auxFeltK i x (k : ve)
        | k == x = False
        | abs (x - k) == i = False
        | otherwise = auxFeltK (i + 1) x ve

> kiralyno 5
[[4,2,5,3,1],[3,5,2,4,1],[5,3,1,4,2],[4,1,3,5,2],[5,2,4,1,3],
[1,4,2,5,3],[2,5,3,1,4],[1,3,5,2,4],[3,1,4,2,5],[2,4,1,3,5]]
```

Az `n` királynő feladat

- az algoritmus a kombinációkat meghatározó függvény egy változata lesz
- a feltételek teljesülését vizsgáló függvény a `feltKir` lesz
- ennél a feladatnál a listaelemek nem kell növekvő sorrendben legyenek, elég ha különböznek
- az eredménylisták elemeit akkor határozzuk meg, amikor *jövünk vissza* a rekurzióbó, például az `5x5`-ös sakktábla esetében a `[4,2,5,3,1]` lista elsőként meghatározott eleme az `1` lesz, majd eléje kerül a `3`, majd az elé az `5`-ös, és így tovább
- az átlók menti ütközés kizárását a `abs (x - k) == i = False` feltétel biztosítja
- ha igaz a fenti feltétel az azt jelenti, hogy az `x` elem a már kigenerált listabeli elemek közül a `k`-val, amely `i` pozícióyira helyezkedik el az `x`-től, az átlók mentén, ütközni fog, például a `6x6`-os sakktábla esetében a `[2,6,3]` lista elejére nem lehet `1`-est tenni, mert az átlós ütközést eredményezne a `2`-vel (`abs (2 - 1) == 1`), de `4`-est sem lehet tenni, mert az a `6`-tal okozna átlós ütközést (`abs(6 - 4) == 2`)

Az n királynő feladat

a `feltKir` tömörebb változata a következő:

```
feltKir_ :: Int -> [Int] -> Bool
feltKir_ x ls = auxFeltK 1 x ls
    where
    auxFeltK :: Int -> Int -> [Int] -> Bool
    auxFeltK i x ls = all (aux x) $ zip [1..] ls
        where
        aux x (i, k) = (k /= x) && (abs(x - k) /= i)
```

Az n királynő feladat

A feladat megoldásait elegánsabban is meg tudjuk jeleníteni a képernyőn, a `foKiralyno` meghívásával:

```
kiirSor :: Int -> Int -> IO()
kiirSor n k = do
    mapM_ (auxF k) [1..n]
    putStrLn ""
        where
        auxF k x =
            if k == x then putStr "Q " else putStr ". "

kiirTabla :: Int -> [Int] -> IO()
kiirTabla n ls = do
    mapM_ (kiirSor n) ls
    putStrLn ""

foKiralyno :: Int -> IO()
foKiralyno n = mapM_ (kiirTabla n) $ kiralyno n

> foKiralyno 5
```