Datastrukturer i Haskell

Bror Bjerner
Inst. för data- och informationsteknik
Göteborgs universitet & Chalmers tekniska högskola

2010

1 Fibbonacci

Om vi beräknar det n:te fibonnacitalet enligt sin dubbelrekursiva definition, så får vi en komplexitet av $\mathbf{0}(\mathit{fib}(n))$, dvs exponentiellt.

I det iterativa fallet löser vi det genom att göra en loop i stället. För det rekursiva fallet löser vi det genom att göra en s.k. värdeförloppsrekursion, *course-of-value recursion*.

```
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = covfib n 0 1
    where covfib 1 prev curr = curr
          covfib n prev curr = covfib (n-1) curr (prev+curr)
```

// bättre att ha 'i' istället för 'n' i covfib-definitionen // bör förklara att covfib är Haskellversionen av en iterativ loop

Denna funktion är uppenbarligen av $\mathbf{0}(n)$, precis som den iterativa lösningen.

2 Sortering

Då det gäller sortering i listor i stället för fält, så har vi ingen direktaccess till olika positioner i listan, vilket vi ju har för fält. Algoritmer, som bygger på

swap-operationen, är därför olämpliga, att använda i funktionell programmering. Snabbsortering och urvalssortering bygger på swap-metoden och är därför olämpliga i funktionell programmering.

Insättnings- och sammansmältningssortering passar däremot alldeles förträffligt.

2.1 Insättningssortering

Eftersom det är svårt att 'backa' i en lista så stoppar vi i stället in varje element i den sorterade svansen.

```
insertionSort :: Ord a => [a] -> [a]
insertionSort []     = []
insertionSort (a:as) = insert a (insertionSort as)
    where insert a [] = [a]
          insert a bbs@(b:bs)
              | a > b     = b : insert a bs
              | otherwise = a:bbs
```

I värsta fallet, omvänt sorterat, blir rekursionsdjupet för `insert` antalet element i den givna listan. Totalt får vi då $\sum_{i=0}^{n-1} i$ anrop, dvs komplexiteten blir av $\mathbf{0}(\, n^2\,)$.

I bästa fallet, redan sorterat, blir rekursionsdjupet för `insert` endast 1, dvs komplexiteten blir av $\mathbf{0}(\, n\,)$

I genomsnitt blir rekursionsdjupet för `insert` halva antalet element i den givna listan, varför även här komplexiteten blir av $\mathbf{0}(\, n^2\,)$

2.2 Sammansmältningssortering

Låt oss först titta på sammansmältningen av två redan sorterade dellistor.

```
merge :: Ord a => [a] -> [a] -> [a]
merge [] bs = bs
merge as [] = as
merge aas@(a:as) bbs@(b:bs)
    | a < b     = a : merge as bbs
    | otherwise = b : merge aas bs
```

För att nu sortera en godtycklig lista gör vi först varje element till en lista endast innehållande detta element. Därefter sammansmältes dessa parvis, via en generell rekursion, tills endast en lista återstår.

```
mergeSort :: Ord a => [a] -> [a]
mergeSort []     = []
mergeSort as@[a] = as
mergeSort as     = untilOneList [ [a] | a <- as ]
    where untilOneList [as]     = as
          untilOneList ass      = untilOneList (mergePairWise ass)
          mergePairWise []      = []
          mergePairWise ass@[_] = ass
          mergePairWise (a:b:s) = merge a b : mergePairWise s
```

Eftersom listan av listor reduceras till hälften i varje 'varv' och de är från början är n stycken blir det totalt ${}^2log\, n$ 'varv'. Antalet element som gås igenom i varje 'varv' är åtminstone $\frac{n}{2}$. Komplexiteten blir alltså $\mathbf{O}(\, n * {}^2log\, n\,)$ för `mergeSort`.

3 Stackar

För att göra en stack definierar vi en modul med föjande signatur:

```
module Stack (
  Stack,      -- type of stacks
  emptyStack, -- Stack a
  isEmpty,    -- Stack a -> Bool
  push,       -- a -> Stack a -> Stack a
  pop,        -- Stack a -> Stack a
  top         -- Stack a -> a
) where
```

Implementeringen blir enkel, eftersom en lista i Haskell uppför sig precis som en stack. Vi väljer därför helt enkelt en lista som implementering och översätter stackfunktionerna med motsvarande listfunktioner.

```
newtype Stack a = Stack [a]
                  deriving ( Eq, Show, Read )

emptyStack = Stack []

isEmpty (Stack []) = True
isEmpty _          = False

push x (Stack as) = Stack (x:as)

pop (Stack [])     = error "empty stack"
pop (Stack (x:as)) = Stack as

top (Stack [])     = error "empty stack"
top (Stack (x:as)) = x
```

Notera att vi här använder oss av `error` för att fånga begreppet exception i Java.

4 Köer

Köer är lite besvärligare att implementera om man vill beakta komplexiteten. Om kön skulle implementeras som en lista kostar det ju $\mathbf{O}(n)$ att stoppa in ett element sist i kön. Däremot kan vi implementera en kö med hjälp av två listor, där vi stoppar in elementen i den ena listan och plockar ut ur den andra. Detta ger en amorterad komplexitet av $\mathbf{O}(1)$. Varje element läggs först i bakre kön och ingår sedan i en och bara en `reverse`-operation.

```
module Queue (
  Queue,      -- type of queues
  emptyQueue, -- Queue a
  isEmpty,    -- Queue a -> Bool
  enqueue,    -- a -> Queue a -> Queue a
  dequeue,    -- Queue a -> Queue a
  front,      -- Queue a -> a
) where

data Queue a = Queue [a] [a]
               deriving ( Eq, Show, Read )

emptyQueue = Queue [] []

isEmpty (Queue [] []) = True
isEmpty _             = False

enqueue x (Queue as bs) = Queue as (x:bs)

dequeue (Queue [] [])     = error "empty queue"
dequeue (Queue [] bs)     = dequeue (Queue (reverse bs) [])
dequeue (Queue (x:as) bs) = Queue as bs

front (Queue [] [])     = error "empty queue"
front (Queue (x:as) bs) = x
```

5 Binära träd

Vi skall här endast betrakta binära sökträd. Vi kommer att definiera två former: det enkla binära sökträdet och det höjdbalanserade AVL-trädet.

5.1 Enkla binära sökträd

Vi definierar följande modul med sin signatur.

```
module BinarySearchTree (
  BST,       -- type of binary search trees
  emptyTree, -- BST a
  isEmpty,   -- BST a -> Bool
  leftSub,   -- BST a -> BST a
  rightSub,  -- BST a -> BST a
  rootVal,   -- BST a -> a
  insert,    -- Ord a => a -> BST a -> BST a
  remove,    -- Ord a => a -> BST a -> BST a
  inorder,   -- BST a -> [a]
  get        -- Ord a => a -> BST a -> Maybe a
) where
```

Om vi gör en jämförelse med Java kan vi notera:

- att typen ej behöver ha samma namn som modulen och dess konstruktorer, vilket krävs i Java för en klass.
- Att de typer som förkommer i signaturen fungerar som `public` i Java och att övriga definitioner fungerar som `private` i Java, dvs endast internt i modulen.

Då det gäller träd finns ingen konkret datastruktur att bygga på, varför vi definierar en ny rekursiv datatyp. Ett träd är antingen det tomma trädet eller så är det en nod bestående av ett rotvärde, ett vänster delträd och ett höger delträd.

```
data BST a = Empty |
             Node a (BST a) (BST a)
             deriving ( Eq, Show, Read )
```

Notera att vi här inte har angett `Empty` och `Node` i signaturen, varför vi behöver funktioner för att skapa ett nytt tomt träd, samt att plocka sönder ett träd i dess beståndsdelar.

```
emptyTree = Empty

isEmpty Empty = True
isEmpty _     = False


leftSub Empty        = error "no left subtree"
leftSub (Node _ l _) = l

rightSub Empty        = error "no right subtree"
rightSub (Node _ _ r) = r

rootVal Empty        = error "no root value"
rootVal (Node a _ _) = a
```

Orsaken till att vi gömmer konstruktorerna för typen `BST`, är att vi vill, att all modifiering av trädet skall gå via funktionerna `insert` och `remove`, så att egenskapen att vara ett binärt sökträd alltid bibehålls.

För `insert` gör vi som i Java. Vi letar upp första lediga 'hål', placerar det nya elementet där och låter övriga delar av trädet vara som före insättningen.

```
insert a Empty =  Node a Empty Empty
insert a (Node b l r)
     | a < b     = Node b (insert a l) r
     | otherwise = Node b l  (insert a r)
```

Då det gäller `remove`, så gör vi samma knep som i det imperativa fallet. Om den nod som skall förstöras har två icke-tomma subträd, hämtar vi upp det största elementet i vänster subträd, samt plockar ut det ur det vänstra subträdet.

```
remove a Empty  = Empty
remove a (Node b l r)
    | a < b     = Node b (remove a l) r
    | a > b     = Node b l (remove a r)
    | isEmpty l = r
    | isEmpty r = l
    | otherwise = Node maxL (remove maxL l) r
    where maxL  = findMax l

findMax (Node a _ Empty) = a
findMax (Node a _ r)     = findMax r
```

Notera att `findMax` ej är känd utanför modulen, varför vi inte behöver definiera den för det tomma trädet.

För att traversera elementen i ett träd, så genererar vi en lista där vi ger elementen i inorder, dvs en sorterad lista.

```
inorder t = inord t []
    where inord Empty as        = as
          inord (Node a l r) bs = inord l ( a : inord r bs )
```

Här använder vi oss av en högre ordnigens funktion `inord` som 'pressar' fram en delberäkning av 'svansen'. Vi kunde använt oss av konkatenering enligt: `inorder (Node a l r) = inorder l ++ a : inorder r`

Motiveringen till att välja funktionen `inord`, är att med den senare definitionen, får vi konkateneringar på varje rekursionsnivå, dvs en komplexitet av minst $\mathbf{0}(\, n * {}^2log\, n\,)$. Med den högre ordningens funktion får vi däremot komplexiteten $\mathbf{0}(\, n\,)$ vilket givetvis är att fördra.

Slutligen har vi en funktion, som ur ett givet träd, hämtar ett element, som är lika stort som det givna argumentet. Eftersom det inte är säkert att det finns något sådant element använder vi oss av typen `Maybe`, jämför med att ge `null` som resultat i Java. Funktionen kan vara bra att ha, om man till exempel vill göra något som liknar en `Map` i Java.

```
get a Empty = Nothing
get a (Node a' l r)
    | a < a'    = get a l
    | a > a'    = get a r
    | otherwise = Just a'
```

5.2 AVL träd

Givetvis kan vi använda oss av balanseringar i Haskell. Vi skall här använda höjdbalanserade träd enligt AVL-algoritmerna. Modulens signatur, förutom själva typen `AVL` blir den samma som för `BinSearchTree`.

```
module AVLTree (
  AVL,       -- type of AVL balanced search trees
  emptyTree, -- AVL a
  isEmpty,   -- AVL a -> Bool
  leftSub,   -- AVL a -> AVL a
  rightSub,  -- AVL a -> AVL a
  rootVal,   -- AVL a -> a
  insert,    -- Ord a => a -> AVL a -> AVL a
  remove,    -- Ord a => a -> AVL a -> AVL a
  inorder,   -- AVL a -> [a]
  get        -- Ord a => a -> AVL a -> Maybe a
) where
```

För att inte behöva beräkna höjderna ideligen så introduceras, precis som i det imperativa fallet, en del av noden där höjden för trädet finns beräknat. Vi får den nya datatypen:

```
data AVL a = Empty |
             Node a Int (AVL a) (AVL a)
             deriving ( Eq, Show, Read )
```

Funktionerna `emptyTree`, `isEmpty`, `leftSub`, `rightSub`, `rootVal`, `get`, `inorder` och `findMax` blir desamma förutom vid mönstermatchningen av en `Node` där det extra integerfältet måste läggas till. Det görs med hjälp av ett wild card, eftersom höjdinformationen ej behövs för dessa funktioner. Däremot behöver vi en ny intern funktion som ger höjden för ett träd eller subträd.

```
height Empty          = 0
height (Node _ h _ _) = h
```

Vid insättning eller urtagning av element i trädet kan det däremot uppstå obalans, varför denna information dels behövs för att kontrollera balansen och dels behövs för uppdatering.

Vid obalans finns det fyra möjliga balanseringar att tillämpa. Vi definierar dem som egna funktioner eftersom de behövs både vid insättning och vid urtagning. Enkel rotation åt vänster eller åt höger, samt dubbel rotation åt vänster respektive höger. Vi redovisar här endast vänsterrotationerna:

```
rotateLeft (Node a ah al (Node b bh bl br))
    = Node b (1 + max newh (height br)) (Node a newh al bl) br
    where newh = 1 + max (height al) (height bl)

doubleRotateLeft(Node a ah al (Node b bh (Node c ch cl cr) br))
    = Node c (1 + max newh1 newh2)
           (Node a newh1 al cl)
           (Node b newh2 cr br)
    where newh1 = (1+max(height al)(height cl))
          newh2 = (1+max(height cr)(height br))
```

På varje nivå behövs en kontroll av höjderna tillsammans med en kontroll av obalans. Om obalans kan ha uppstått genom att vänster delträd kan ha blivit högre anropas `checkLeftHeavy` annars anropas `checkRightHeavy`

```
checkLeftHeavy at@(Node a ah bt@(Node b bh bl br) ar)
    | bh - arh < 2          = Node a ( 1 + max bh arh ) bt ar
    | height bl < height br = doubleRotateRight at
    | otherwise             = rotateRight at
    where arh = height ar
```

Vi är nu färdiga att definiera insättning och borttagning av element i trädet:

```
insert x Empty  = Node x 1 Empty Empty
insert x (Node a ah al ar)
    | x < a     = checkLeftHeavy (Node a ah (insert x al) ar)
    | otherwise = checkRightHeavy (Node a ah al (insert x ar))

remove x Empty   = Empty
remove x (Node a ah al ar)
    | x < a      = checkRightHeavy (Node a ah (remove x al) ar)
    | x > a      = checkLeftHeavy (Node a ah al (remove x ar))
    | isEmpty al = ar
    | isEmpty ar = al
    | otherwise  = checkRightHeavy (Node maxL ah (remove maxL al) ar)
    where maxL   = findMax al
```

5.3 Bredden först för binära sökträd

Vi skall betrakta ett litet exempel på att använda de moduler som vi definierat. Exemplet är att traversera ett binärt sökträd enligt bredden först. För att enkelt hålla reda på ordningen för elementen, använder vi som vanligt vid bredden först en kö.

```
import Queue
import BinarySearchTree

breadthFirst :: BST a -> [a]
breadthFirst t = bF t emptyQueue
    where bF t que
              | BinarySearchTree.isEmpty t = checkQue que
              | otherwise = (rootVal t :
                             checkQue (enqueue (rightSub t)
                                       (enqueue (leftSub t) que)))
          checkQue que
              | Queue.isEmpty que = []
              | otherwise         = bF (front que) (dequeue que)
```

Om vi vill använda oss av AVL-träd så byter vi ut alla **BinSearchTree** mot **AVLTree**. Detta är inga problem då vi använder Hugs, eftersom Hugs är textbaserat.

Om vi däremot vill använda oss av färdigkompilerade delar i Haskell får vi i stället göra en `class` där vi definierar vad för funktioner och dess typer som utgör ett binärt sökträd. Ungefär som att göra ett typat interface i Java. Detta ingår dock ej i denna kurs.

6 Prioritetsköer

Att implementera en heap, ser först ut att vara svårt i Haskell. En heap är ju nivåbalanserad, vilket det är svårt att ge en rekursiv definition för. Vi kan emellertid ge en något modifierad definition, där trädet är nodbalanserat:

Ett *heapträd* är antingen det tomma trädet, eller så är rotelementet det minsta elementet i hela trädet, antalet noder i högra delträdet minus antalet noder i det vänstra delträdet är 0 eller 1 och höger och vänster delträd i sin tur är heapträd.

Vi börjar med att ge en signatur för prioritetskön:

```
module PriorityQueue (
  PriQue,   -- type of priority queues
  emptyQue, -- PriQue a
  isEmpty,  -- PriQue a -> Bool
  add,      -- Ord a => a -> PriQue a -> PriQue a
  getMin,   -- Ord a => PriQue a -> a
  removeMin -- Ord a => PriQue a -> PriQue a
) where
```

Typen är alltså ett träd för vilket vi definierar konstruerarna:

```
data PriQue a = Empty |
                Node a (PriQue a) (PriQue a)
                deriving ( Eq, Show, Read )
```

`emptyQue` ger den tomma kön och `isEmpty` kollar om det är en tom kö:

```
emptyQue = Empty

isEmpty Empty = True
isEmpty _     = False
```

Att bibehålla heapträdegenskapen vid addering av element till en kö, löser vi genom att stoppa in ett element i det vänstra delträdet och sedan byta vänster och höger delträd med varandra. Det element vi stoppar in i det vänstra delträdet, är det största av rotvärdet och det nya elementet. Det minsta blir nytt rotvärde.

```
add a Empty     = Node a Empty Empty
add a (Node b l r)
    | a < b     = Node a r (add b l)
    | otherwise = Node b r (add a l)
```

Att hämta det minsta elementet, innebär nu bara, att returnera rotvärdet för hela heapträdet.

```
getMin Empty          = error "empty priority queue"
getMin (Node a _ _ ) = a
```

`removeMin` är lite besvärligare. Vi börjar med att ta bort och hämta det högraste elementet i trädet samtidigt som vi vänder tillbaka alla högra delträd på vägen ner i trädet, för att bibehålla kravet på antal noder i delträden.

```
getMostRight (Node a Empty Empty) = (a,Empty)
getMostRight (Node a l r) = (b, Node a bt l)
    where (b,bt) = getMostRight r
```

Det bortagna elementet måste sjunka ned i trädet till en plats där det uppfyller kravet att vara det minsta värdet i subträdet.

```
restore a Empty Empty = Node a Empty Empty
restore a Empty bt@(Node b Empty Empty)
    | a < b     = Node a Empty bt
    | otherwise = Node b Empty (Node a Empty Empty)
restore a bt@(Node b bl br) ct@(Node c cl cr)
    | a < b && a < c = Node a bt ct
    | b < c          = Node b (restore a bl br) ct
    | otherwise      = Node c bt (restore a cl cr)
```

Nu definieras `removeMin` enkelt med de två hjälpfunktionerna.

```
removeMin Empty = error "empty priority queue"
removeMin (Node _ l r ) = restore b bt l
    where (b,bt) = getMostRight r
```