5DV086

Lab 2 - Huffman

Submission v1.0

Teacher: Henrik Björklund

Tutor: Oscar Kamf

Gustaf Söderlund - et14gsd - (gusa0038@cs.umu.se)

March 3, 2022

Contents

List of Figures

1 Introduction

Syftet med laborationsuppgiften är att få en förståelse och applicera egendefinerade och rekursiva datatyper.

I laborationsuppgiften utförs Huffman-kodning som används primärt för att komprimera stora filer. I denna laborationsuppgift kommer vi att komprimera strängar istället för stora filer.

En sträng består av x-antal chars. Varje char är 8 bitar eller 1 byte stor. Om vi t.ex. har strängen "Hej" kommer storleken på strängen att vara $3X8 = 24$. Om vi istället komprimerar denna sträng med Huffman-kodning kan vi minimera storleken till 5 bitar genom att koda om strängen så att j = 0, e = 10 och H = 11 alltså [11100].

Laborationsuppgiften är uppdelad i fyra stycken deluppgifter. Som beskrivs i mer detalj i kapitel 2.

2 Systembeskrivning

Laborationsuppgiften är uppdelad i fyra stycken deluppgifter: *statistics*, *maketree*, *encode* och *decode*.
Programmet använder sig av följande datatyper som kommer att refereras till i systembeskrivningen.

```
1  data Htree = Leaf Char | Branch Htree Htree deriving Show
2  data WeightedTree = WL Integer Char | WB Integer
   ↪  WeightedTree WeightedTree deriving Show
3  data Ftree = FLeaf Char [Integer] | Node Ftree Ftree
   ↪  deriving Show
```

2.1 Statistics

I den fösta deluppgiften *statistics* ska en lista av tuplar som innehåller integers och chars produceras utifrån en inputsträng. Signaturen för funktionen *statistics* presenteras nedan.

```
statistics :: String -> [(Integer,Char)]
```

Inputsträngen kan vara t.ex strängen "test". Listan som ska skapas kommer att se ut som följande: [(1,'e'),(1,'s'),(2,'t')] alltså en lista med varje char från inputsträngen och en integer som säger hur många av den charen som finns i inputsträngen.

Statistics börjar med att använda Haskells inbyggda funktion *fromListWith* för att skapa ett key och value par för varje karaktär i strängen. Om en karaktär upprepas ökar value med 1.

Funktionen *fromListWith* skapar en lista på följande struktur [(Char,Integer)] och här behöver man vända på tuplarna i listan med funktionen *reverseFunc* och *reversetuple* för att få den att stämma överens med signaturen för *statistics*.
Haskells inbyggda funktion *sort* används för att sortera värdena i listan i storleksordning från minst till störst value.

2.2 Maketree

I den andra deluppgiften *maketree* ska ett Huffman-träd konstrueras, *Htree*, från en given lista med tuplar som innehåller integers och chars. signaturen för maketree ser ut som följande:

```
maketree :: [(Integer, Char)] -> Htree
```

2.2.1 Skapa löv till trädet

För att skapa ett *Htree* används funktionen *makeLeafs* som rekursivt går igenom [(Integer, Char)]- listan och vid varje iteration skapar man ett löv. Löven skapas med hjälp av funktionen *createL* som har signaturen:

```
createL :: (Integer,Char) -> WeightedTree
```

Funktionen *createL* använder sig av typkonstruktorn *WL* för att skapa löven.
När listan av löv är skapad behöver man kolla efter ett specialfall, om användaren har angett en input- sträng som innehåller t.ex. "aaaaaaaa". Detta kontrolleras med hjälp av funktionen *checkDups*:

```
checkDups :: [WeightedTree] -> [WeightedTree]
```

Funktionen *checkDups* kollar om listan av löv innehåller endast ett löv eller flera. Om listan av lös endast innehåller ett löv behöver man skapa en kopia. Om listan

innehåller fler än ett löv låter man listan vara som den är.

2.2.2 Skapande av träd

Efter att listan av löv är skapad och specialfallet är kontrollerat börjar man att konstruera trädet. Detta görs rekursivt med hjälp av funktionen *makeBranchedTree*:

```
makeBranchedTree :: [WeightedTree] -> [WeightedTree]
```

Funktionen *makeBranchedTree* tar in en lista av *WeightedTree*, i denna lista innehåller nu bara löv. Men genom att rekursivt travesera denna lista och använda typkonstruktorn *WB* kan man skapa en ny lista som endast innehåller ett element av *WeightedTree*.

Vid varje iteration behöver man sortera listan så att det löv med högst vikt befinner sig längst bak i listan. Det görs med hjälp av funktionerna *sortList* och *compareWeight*. Här kontrolleras om ett element har större, mindre eller samma vikt som nästa element i *WeightedTree*- listan.

När listan den nya listan av *WeightedTree* är konstruerad kommer den att se ut som följande [WB 4 (WB 2 (WL 1 'e') (WL 1 's')) (WL 2 't')].

2.2.3 Skapande av Htree

Nu ska ett *Htree* konstrueras utifrån denna lista. Eftersom det enbart är ett element i *WeightedTree*-listan kan man använda funktionen *rmList* som tar ut det första elementet i listan. Därefter används funktionen *getTree* som har signaturen :

```
getTree ::  WeightedTree -> Htree
```

Funktionen *getTree* går rekursivt igenom *WeightedTree* och bygger ett Löv med hjälp av typkonstruktorn *Leaf* när indata är av typkonstruktorn *WL*. När indata är av typen *WB* skapar man en gren med hjälp av typkonstruktorn *Branch*. Slut resultatet om man använt strängen "test" blir följande: Branch (Branch (Leaf 'e') (Leaf 's')) (Leaf 't') som visuellt presenteras som ett träd i Figur 2.2.3.

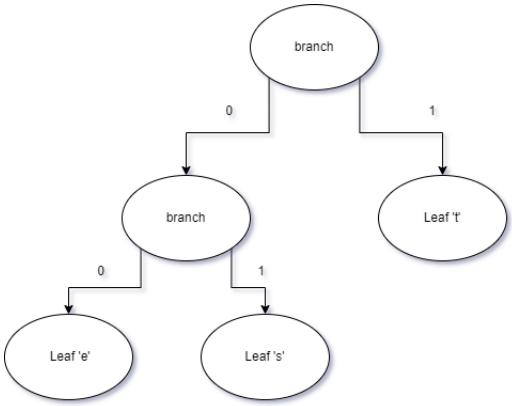

Figure 1: En visuell presentation av strängen "test" i ett Huffman-träd.

Som man ser i Figur 2.2.3 har 't' kortare väg därför att det uppstår två 't' i strängen "test", därför vill man ha en kort väg till det karaktären.

2.3 Encode

I den tredje deluppgiften kommer man använda funktionerna *statistics* och *maketree* från tidigare deluppgifter för att skapa funktionen *encode* med signaturen:

```
encode :: String -> (Htree, [Integer])
```

Funktionen encode tar in en sträng och ska returnera en tuple som innehåller Huffman-trädet, *Htree*, samt en lista av heltal som ska representera strängen i huffman-kod. Listan kommer endast att bestå utav ettor och nollor. *Htree* som ska returneras görs tillsammans med funktionen *maketree* från deluppgift 2 och *statistics* från deluppgift 1.

2.3.1 Skapa Paths

För att skapa Listan av heltal används först *maketree* och *statistics* för att skapa huffman-trädet. När trädet är skapat kallar man på funktionen *createPaths*. Denna funktion har signatur:

```
createPaths :: Htree  -> [Integer] -> Ftree
```

Funktionen tar in Huffman-trädet samt en tom lista och kallas rekursivt i sig själv för att skapa ett *Ftree*. Funktionen kallar sig själv rekursivt genom att kontrollera om den nuvarande indatat är en *Branch*. Om indatat är en Branch kommer man att skapa typkonstruktorn *Node* och kallar *createPaths* rekursivt med vänster branch och adderar en nolla till listan som man började med. Samma sak sker för höger Branch, men i det fallet adderas en 0:a till listan istället. Detta utförs fram tills man når ett *Leaf* då skapas typkonstruktorn *Fleaf* som får en char samt en lista som innehåller pathen genom trädet för att nå charen.

När funktionen *createPaths* är färdig kommer det att existera ett *Ftree* som kan representeras såhär med strängen "test"

```
Node (Node (FLeaf 'e' [0,0]) (FLeaf 's' [0,1])) (FLeaf 't' [1])
```

I resultatet ovan ser man att varje *Fleaf* håller en char och en lista med vägen till charen.

2.3.2 Plocka ut löv från *Ftree*

När man skapat ett *Ftree* kommer funktionen *extractLeafs* att annropas vars syfte är att plocka ut alla löv från *Ftree* så att man får en [(Char,[Integer])]- lista. Signaturen för *extractLeafs* går som följande:

```
extractLeafs :: String -> Ftree-> Ftree -> [(Char,[Integer])]
```

I denna funktion tar man in inputsträngen och *Ftree* som precis skapades tillsammans med en kopia av *Ftree*. Anlednignen till varför man tar in en kopia är för att kunna återställa när det behövs. Funktionen anropas rekursivt. Det slutliga resultatet från denna funktion kommer att se ut som följande:

```
[('e',[0,0]),('s',[0,1]),('t',[1])]
```

2.3.3 Skapande av Huffman-kod

När listan av chars och dess path är färdig annropas funktionen *setBits* som har följande signatur:

```
setBits :: String -> [(Char,[Integer])] -> [(Char,[Integer])] -> [Integer]
```

Funktionen tar in inputsträngen, samt två kopior av listan som precis skapades. Denna funktion kommer att skapa en lista med huffmankod utifrån det angivna

inputdatat. Genom att rekursivt anropas *setbits* och jämföra första charen i inputsträngen med den char som finns i [(Char,[Integer])]-listan. Om de matchar adderar man pathen till en lista med hjälp av funktionen *getPath* och annropar *setBits* fast man tar bort huvudet på input strängen och resetar [(Char,[Integer])]-listan med kopian. Skulle de inte matcha kommer man travesera vidare i [(Char,[Integer])]-listan fram tills dess att de matchar.

Slutresultatet för *Encode* funktionen kommer att ge följande resultat med inputsträngen "test":

```
(Branch (Branch (Leaf 'e') (Leaf 's')) (Leaf 't'),[1,0,0,0,1,1])
```

2.4 Decode

I den fjärde och sista deluppgiften ska skapades funktionen *decode* som har följande signatur:

```
decode :: Htree -> [Integer] -> String
```

Syftet med funktionen är att ta in ett Huffman-träd och en lista med huffman-kod för en specifik sträng. Funktionen ska returnera denna specifika sträng.

Funktionen börjar med att anropa funktion *traverseTree* som har följande signatur:

```
traverseTree :: Htree -> Htree -> [Integer] -> String
```

Här tar man in två kopior på Huffman-trädet, *Htree* för att kunna travesera och återställa till original-trädet. Samt listan med Huffman-kod. Funktionen *traverseTree* annropas rekursivt fram tills dess att listan av Huffman-kod är tom.

Vid varje iteration kontrollerar man i listan av huffman-kod om det är en etta eller en nolla. Om det är en etta klättrar man till höger i trädet, annars vänster om det är en nolla. Detta gör man fram tills dess att man når ett *Leaf*. När man når ett *Leaf* tar plockar man ur dess char och sparar i en lista, sen återställer man trädet med hjälp av kopian som man skickade in.

När listan av Huffman-kod är slut kommer man ha hela strängen tillhands.

3 Användarhandledning

För att använda programmet behöver man först installera GHCI på sin maskin. Det går att installera via www.haskell.org/ghc/. När GHCI är installerat startar man en terminal och navigerar fram till den mapp som innehåller filen Huffman.hs. Därefter skriver man in kommandot *GHCI* i terminalen följt av kommandot *:l Huffman.hs*. Programmet kommer att kompileras och är redo att användas. I Figur 3 presenteras hur terminalen kommer att se ut efter kompilering.

Figure 2: Kompilering av programmet Huffman.hs

3.1 Statistics

För att använda funktionen *statistics* behöver användaren ange kommandot *statistics* följt av en inputsträng. Inputsträngen måste ha citattecken runt sig, annars kommer Haskell att ge varningar. Funktionen kommer att spotta ut följande data om användaren anger t.ex. inputsträngen "test".

$$[(1,'e'),(1,'s'),(2,'t')] \tag{1}$$

3.2 Maketree

Om användaren vill använda funktionen *maketree* behöver man ange kommandot *maketree* följt av en lista på följande struktur [(Integer, Char)]. Char representerar en karaktär från en sträng och Integer ska representera hur många gånger denna karaktär upprepas i strängen. Alternativt kan användaren använda funktionen *statistics* för att skapa listan. Nedan ser man ett huffman-träd som funktionen kommer att spotta ut med [(1,'e'),(1,'s'),(2,'t')] som indata.

$$Branch(Branch(Leaf'e')(Leaf's'))(Leaf't') \tag{2}$$

3.3 Encode

Om användaren vill använda funktionen *encode* börjar man med att ange kommandot *encode* följt av en inputsträng. Inputsträngen måste ha citat-tecken runtom sig för att undvika varningar från haskell. I Figur 3.3 åskådliggörs en körning med funktionen *encode*.

```
*Huffman> encode "test"
(Branch (Branch (Leaf 'e') (Leaf 's')) (Leaf 't'),[1,0,0,0,1,1])
```

Figure 3: Exempel körning med funktionen *encode* med strängen "test" som inputsträng.

3.4 Decode

Om användaren vill använda funktionen *decode* börjar man med att ange kommandot *decode* följt av ett huffman-träd och en lista med huffman-kod som indata. Detta indata går att skapa med hjälp av funktionen *encode* ett exempel går att se i Figur 3.3. Alternativt att man skriver in det själv, vilket kan vara lite pilligt.

I Figur 3.4 kan man se en körning med funktionen *decode* där strängen "test" skapas utifrån indatat (Branch (Branch (Leaf ’e’) (Leaf ’s’)) (Leaf ’t’)) och listan [1,0,0,0,1,1].

```
*Huffman> decode (Branch (Branch (Leaf 'e') (Leaf 's')) (Leaf 't')) [1,0,0,0,1,1]
"test"
```

Figure 4: Exempel körning med funktionen *decode*.

4 Testning

Testningen av programmet gjordes under implementation av respektive funktion som programmet är uppbyggt utav. Varje deluppgift delades upp i mindre uppgifter.

Vid skapandet av varje funktion har minimalt med kod försökt användas för att enklare kunna felsöka. Vid testning av funktionerna har terminalen använts där man anger indata manuellt och kontrollerar utdata.

5 Diskussion

Denna uppgift var mycket lättare att komma igång med jämfört med den tidigare Parkeringshus uppgiften som utfördes. Jag kände att det blev lättare och lättare att skriva kod och leta information på nätet när man börjat vänja sig med diverse dokumentations hemsidor.

Jag har inte testat att göra Huffman-kodning i andra programspråk som är imperativa eller objektorienterade. Men det är något som jag kommer att försöka mig på. Att utföra laborationen i ett funktionellt språk som Haskell kändes väldigt bra. Rekursion för skapande av träd och travesera sig fram genom trädet känns som en naturlig lösning.

I ett objektorienterat språk kan jag tänka mig att varje löv kan vara ett objekt som man skapar med en karaktär och en path.
I ett imperativt språk som C skulle jag använda mig utav struct för löven och någon form av rekursiv lösning för byggande och travesering av träd.

Det som har varit den svåraste utmaningen i uppgiften har varit att hänga med i rekursionen och hur träden skulle byggas upp. Jag fick skriva kod flera gånger som jag sedan fått tagit bort då jag insett att det makes ingen sense det jag skrivit efter man testat den.

Det jag känner mig mest nöjd över är deluppgift 4 som jag lyckades skriva ihop väldigt snabbt efter att suttit och bråkat med deluppgift 3.

6 källkod

```haskell
module Huffman where

--Imports
import Data.List
import Data.Map
import Data.Ord hiding(compare)
import Prelude hiding(compare)
import qualified Data.Map as Map

----------------- datatyper ------------------------------
data Htree = Leaf Char | Branch Htree Htree deriving Show
data WeightedTree = WL Integer Char | WB Integer WeightedTree
 ↪  WeightedTree deriving Show
data Ftree = FLeaf Char [Integer] | Node Ftree Ftree deriving
 ↪  Show
----------------- Deluppgift 1 ------------------------------

statistics :: String -> [(Integer, Char)]
statistics [] = [] --basecase
statistics input = sort(reverseFunc(toList $ fromListWith (+)
 ↪  [(k,1)| (k) <- input]))

--use to reverse the tuples in the list
reverseFunc :: [(Char,Integer)] -> [(Integer,Char)]
reverseFunc [] = []
```

```haskell
reverseFunc  (x:xs) = reversetuple x: reverseFunc xs

--reverse the tuple to match the output.
reversetuple :: (Char, Integer) -> (Integer, Char)
reversetuple (c,i) = (i,c)

----------------- Deluppgift 2 ------------------------------
maketree :: [(Integer, Char)] -> Htree
maketree input =  getTree( head( makeBranchedTree(checkDups(
 ↪  makeLeafs(input)))))

--------------make leafs------------
-- create a list of leafs---
makeLeafs :: [(Integer, Char)] -> [WeightedTree]
makeLeafs [] = []
makeLeafs (x:xs) =  createL x : makeLeafs xs

-- set WL ----
createL :: (Integer,Char) -> WeightedTree
createL (fst,snd) = WL fst snd

--This function is used if the input string has only 1 letter
-- for example "aaaaaaaaaa", If so i want to duplicate the
 ↪  leaf
-- in order to be able to create the branch.
checkDups :: [WeightedTree] -> [WeightedTree]
```

```haskell
checkDups [] = []
checkDups (x:xs) =   if (length (x:xs) < 2)
                        then x:(x:xs)
                    else (x:xs)

------------Begin branching---------

--B Integer WeightedTree WeightedTree deriving Show
makeBranchedTree :: [WeightedTree] -> [WeightedTree]
makeBranchedTree [] = []
makeBranchedTree (a:[]) = [a]
makeBranchedTree (a:b:[]) = [WB (getWeight(a) + getWeight( b))
 ↪  a b]
makeBranchedTree (a:b:c) = makeBranchedTree( sortList(WB
 ↪  (getWeight(a) + getWeight(b)) a b : sortList c))

-- get the weight
getWeight :: WeightedTree -> Integer
getWeight (WL weight _ ) = weight
getWeight (WB weight _ _ ) = weight

-- sort the tree, the leaf with least weight is first in list
-- the leaf with highest weight is last in the list.
sortList :: [WeightedTree] -> [WeightedTree]
sortList tree = sortBy compareWeight tree

```

```haskell
-- Compare weight
compareWeight :: WeightedTree -> WeightedTree -> Ordering
compareWeight x y
    | getWeight x < getWeight y  = LT
    | getWeight x > getWeight y  = GT
    | getWeight x == getWeight y = EQ

-- When there's only one WeightedTree in the list, get the head
 ↪  and remove the list.
rmList :: [WeightedTree] -> WeightedTree
rmList x = head x

-- traverse the WeightedTree and create Htree recursive.
-- w1 is left branch and w2 is right branch
getTree ::  WeightedTree -> Htree
getTree (WL _ c) = Leaf c
getTree (WB _ w1 w2) = Branch (getTree w1) (getTree w2)


--------Deluppgift 3---------------
encode :: String -> (Htree, [Integer])
encode [] = (Leaf ' ',[])
encode input =
    let y = (createPaths (maketree (statistics input)) [])
        x = (extractLeafs input y y)
    in (maketree (statistics input), setBits input x x )
```

```haskell

-- Create the paths for each leaf with a list of integers where
 ↪  the integers are 0 and 1.
-- 0 = left 1 = right in the tree.
createPaths :: Htree  -> [Integer] -> Ftree
createPaths (Leaf c) x  = FLeaf c x
createPaths (Branch h1 h2) x = Node (createPaths h1 (x++[0]))
 ↪  (createPaths h2 (x++[1]))

-- Take out all the leafs from the tree and get a list of
 ↪  tuples with
-- the char and path from the leaf
extractLeafs :: String -> Ftree-> Ftree -> [(Char,[Integer])]
extractLeafs [] _ _ = []
extractLeafs string tree (Node f1 f2) = (extractLeafs string
 ↪  tree f1) ++ (extractLeafs string tree f2)
extractLeafs (x:xs) tree (FLeaf c path) = [(c, path)]

-- Create a list of all the paths. compare the input string
 ↪  with the list of char and integers
-- when the head of the string is equal to the char in the
 ↪  list, set the path in the
-- output list.
setBits :: String -> [(Char,[Integer])] -> [(Char,[Integer])]
 ↪  -> [Integer]
setBits [] _ _  = []
```

```haskell
setBits (x:xs) org (y:ys)
                              | x == (getchar y) = (getPath y) ++ setBits xs org org
                              | x /= (getchar y) = (setBits (x:xs) org ys)

--Get the path from the tuple.
getPath :: (Char,[Integer]) -> [Integer]
getPath (_,path) = path

--Take the char from the tuple.
getchar :: (Char,[Integer]) -> Char
getchar (c,_) = c

--------Deluppgift 4---------------

decode :: Htree -> [Integer] -> String
decode _ [] = []
decode htree bitSekvens =  traverseTree htree htree bitSekvens

--Traverse the tree using the list of integers t.ex. [0,1,1,0,1,1,1,1]...
-- 0 means left 1 right. Do this until a Leaf is reached, get the char and do it again until the list of integers is empty.
```

```haskell
-- takes 2 Htrees in to the function to "reset" when a char is found in the leafs.
traverseTree :: Htree -> Htree -> [Integer] -> String
traverseTree (Branch h1 h2) _ [] = []
traverseTree (Leaf c) tree [] = [c] ++ traverseTree tree tree []

traverseTree (Leaf c) tree list = [c] ++ traverseTree tree tree list

traverseTree (Branch h1 h2) tree (x:[])
                                        | x == 0 = traverseTree h1 tree (tail [x])
                                        | x == 1 = traverseTree h2 tree (tail [x])

traverseTree (Branch h1 h2) tree (x:xs)
                                        | x == 0 = traverseTree h1 tree xs
                                        | x == 1 = traverseTree h2 tree xs
```

7 Referenser

http://learnyouahaskell.com - Gratis pdf med guider och information om haskell som använts under projektets gång.

http://zvon.org - En hemsida med dokumentation på funktioner.