UMEÅ UNIVERSITET
Institutionen för Datavetenskap
Rapport

3 mars 2022

OU2

Huffman

Programspråk

version 1.0

Albin Dalbert dv19adt

Kursansvarig
Henrik Björklund
Handledare
Oscar Kamf

Innehåll

1 Introduction

2 User guide

This program consists of 4 main functions. statistics, maketree, encode and decode. all of them play a key role in the compression and decompression of a text sequence.

If compressing and decompressing a string is all one want to do. You only need to use encode and decode.

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

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

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

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

to use the program. Use the terminal, navigate to the directory of the *Huffman.hs* file. Now start the *ghci* interpreter and compile the *Huffman.hs* file as seen below.

```
$ ghci
Prelude> :l Huffman
*Huffman>
```

Now the given functions above can be called.

note: the function uncurry might be useful. For example.

```
uncurry decode (encode "fancy string to encode")
```

3 Code

3.1 Functions

```
data Htree = Leaf {c :: Char} | Branch {h0 :: Htree, h1
    :: Htree} deriving (Show, Eq, Ord)
data Wtree = L {weight :: Integer, cha :: Char} | B {
    weight :: Integer, t1 :: Wtree, t2 :: Wtree} deriving
    (Show, Eq, Ord)

type BitTableElem = (Char, [Integer])
```

3.1.1 Encode

takes the string to be compressed and returns the compressed bit sequence and the Huffman tree used to decode the bitsequence.

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

Generates a list of type *BitTableElem*. This is a list of all chars found in the leafs of the given huffman tree. Then it stores the bit sequences for each char in this list. During the compression it uses this bit table to find what sequence to write for each char.

```
makeBitTable :: Htree -> [Integer] -> [BitTableElem]
```

Creates the optimal sequence of bits based on the given bit sequence table and the string to compress.

```
makeBitSequence :: [BitTableElem]-> String -> [Integer]
   -> [Integer]
```

3.1.2 Statistics

Takes in a String and returns the frequency of each character in the string.

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

Recursive function that goes through each char and saves the frequency of each.

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

Replaces the value on the given index. (used by statRec)

```
replaceAtIndex :: Int -> Integer -> [Integer] -> [Integer
    ]
```

3.1.3 maketree

Generates a huffman tree from a freqency table (What statistics above returns)

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

Takes a list of weighted trees and returns a single weighted tree.

```
makeWtree :: [Wtree] -> Wtree
```

Takes in two trees and makes them the children of a new tree it returns

```
createWtreeOfTrees :: Wtree -> Wtree -> Wtree
```

Takes a frequency table and generates a Wtree leaf for each char with it's frequency as the weight.

```
generateWtreeList :: [(Integer, Char)] -> [Wtree] -> [
    Wtree]
```

Creates a huffman tree based on the given weighted tree.

```
wtreeToHtree :: Wtree -> Htree
```

3.1.4 Decode

Decodes the given bit sequence using the given huffman tree.

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

Traverses the huffman tree and the bit sequence to decode the chars

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

3.2 Overview

Encode and decode are the two main function to use the program. Encode uses both statistics and maketree to make the Huffman tree and bit sequence of the given string. It also uses a bit table to save the bit sequence for each char to avoid having to search for the given char each time it is encountered. Then we just need to look in the table during the actual compression process.

maketree, used by Encode, has a special case for when the input string only contains one unique leaf. (A lift of only one leaf). Then in appends a NUL with a weight of 0 to the list of leafs. Then the huffman tree will contain two leafs.

Decode simply goes through the bit sequence, bit by bit and traverses the given huffman tree in pararell. If it encounters a leaf in the tree, it appends that char to the string and start over at the root and continues until the bit sequence it completed.

4 Testning

Testerna testar inte vad som händer om indatat resulterar i att det finns bilar kvar i parkering huset. Men detta är även nämnt som ett odefinierat fall i Användarhandldningen.

För att testa koden använde jag mig utav data vi blivit tilldelade av Oscar Kamf, filen *Register.hs*. Sedan gjorde jag en haskell fil *PhusTests.hs* som importar både *Registr* och *Phus*. Denna modul innehåller både enskilda test för varje testdata i *Register* som kan kallas med respektive kommando

---

Tests if statistics returns a expected list.

```
testStatistics :: Bool
```

Check if the total weight of a weighted tree is the sum of all leafs

```
testWeightMakeTree :: Bool
```

Tests if maketree returns a expected tree.

```
testMakeTree :: Bool
```

---

test Encode if the bit sequence is expected.

```
testEncode :: Bool
```

---

tests if encode and decode to see if the output string is equal to the one given in encode. There is no uuniqie test for only decode as we would need to do all the work encode already thus. And if testEncode passes. we can conclude that decode is flawed if this test is not passed.

```
testEncodeAndDecode :: Bool
```

---

5 Diskussion

This was a much more interesting. From my previous experience with implementing Huffman. Haskell is much better then C. It also really made me realise the strengths of functional programming compared to imperetive and object-oriented. It also felt someone much easier then phus as I know understood the language.

6 Appendix

6.1 Huffman.hs

```haskell
module Huffman (statistics, maketree, encode, decode, Htree (Leaf, Branch)) where
import Data.List (find, delete, sortBy, sortOn)
import Data.Maybe (isJust, fromJust)
import GHC.Char ( chr )
import Data.Tree ()

data Htree = Leaf {c :: Char} | Branch {h0 :: Htree, h1 :: Htree} deriving (Show, Eq, Ord)
data Wtree = L {weight :: Integer, cha :: Char} | B {weight :: Integer, t1 :: Wtree, t2 :: Wtree} deriving (Show, Eq, Ord)

type BitTableElem = (Char, [Integer])


-- Encode ------------------------------------------------------------------------------------------------

-- Input:   The string to be compressed
-- Output:  The resulting bit sequence and huffman tree which is used in decompression

encode :: String -> (Htree , [Integer])
encode str =    let tree = maketree (statistics str)
                in (tree, makeBitSequence (makeBitTable tree []) str [])


-- Makes a bit table with each chars bit sequence.
makeBitTable :: Htree -> [Integer] -> [BitTableElem]
makeBitTable (Leaf c) bits = [(c, bits)]
makeBitTable (Branch t1 t2) bits = makeBitTable t1 (bits ++ [0]) ++ makeBitTable t2 (bits ++ [1])


-- generates the bit sequence from the huffman tree and bitTable
makeBitSequence :: [BitTableElem] -> String -> [Integer] -> [Integer]
makeBitSequence _ [] bitSeq = bitSeq
makeBitSequence bitTable (c:rest) bitSeq = makeBitSequence bitTable rest (bitSeq ++ getBitsForChar c bitTable)


-- returns the bit sequence for the given char
getBitsForChar :: Char -> [BitTableElem] -> [Integer]
getBitsForChar c ((ch, bits):rest) =    if c == ch
                                        then bits
                                        else getBitsForChar c rest


-- Statistics ------------------------------------------------------------------
```

```haskell
-- Input:   A String to generate frequency table of
-- Output:  A list of tuples with a Char from the string and
--          it's corresponding frequency in the string

statistics :: String -> [(Integer, Char)]
statistics text = filtering (statRec text (replicate 256 0))


-- The recursive function of statistics above
statRec :: String -> [Integer] -> [(Integer, Char)]
statRec [] list = zip list (map chr [0..255])
statRec (c:rest) list = statRec rest (replaceAtIndex (fromEnum c) ((list!!fromEnum c)+1) list)


-- filter away all chars with 0 in frequency
filtering :: [(Integer, Char)] -> [(Integer, Char)]
filtering = filter (\(x,_) -> x /= 0)


-- A custom replacement function
replaceAtIndex :: Int -> Integer -> [Integer] -> [Integer]
replaceAtIndex index numb list =    let (x,_:y) = splitAt index list
                                    in x++numb:y


-- MakeTree ------------------------------------------------------------
-- Input:   A frequency table generated by the function 'statistics'
-- Output:  A huffman tree with the most frequent chars high up and less frequent further down

maketree :: [(Integer, Char)] -> Htree
maketree [leaf] = wtreeToHtree (makeWtree (generateWtreeList ([leaf]++[(0,'\0')]) []))
maketree list = wtreeToHtree ( makeWtree (generateWtreeList list []))


-- makes a weighted tree, used to make a effective huffman tree
makeWtree :: [Wtree] -> Wtree
makeWtree [root] = root
makeWtree list =    if length list > 1
                    then makeWtree (sortList (drop 2 list ++ [createWtreeOfTrees (head list) (last (take 2 list))]))
                    else head list


-- Sort list of weighted leafs and subtrees
sortList :: [Wtree] -> [Wtree]
sortList = sortOn weight


-- make a new tree with the two given trees as children
createWtreeOfTrees :: Wtree -> Wtree -> Wtree
createWtreeOfTrees t1 t2 = B (getTreeW t1 + getTreeW t2) t1 t2
```

```haskell


-- returns the weight of a given tree
getTreeW :: Wtree -> Integer
getTreeW (B weight _ _) =  weight
getTreeW (L weight _) =  weight


-- returns the Char of a given leaf
getTreeC :: Wtree -> Char
getTreeC (L _ cha) =  cha


-- generates a list of leafs from a frequency table
generateWtreeList :: [(Integer, Char)] -> [Wtree] -> [Wtree]
generateWtreeList [] wTreeList = wTreeList
generateWtreeList ((i,c):rest) wTreeList = generateWtreeList rest (L i c : wTreeList)


-- transform a weighted tree into a huffman tree
wtreeToHtree :: Wtree -> Htree
wtreeToHtree (L _ c) = Leaf c
wtreeToHtree (B _ t1 t2) = Branch (wtreeToHtree t1) (wtreeToHtree t2)


-- Decode ------------------------------------------------------------
-- Input:  A huffman tree used to traverse the bit sequence, and the sequence to decompress
-- Output:  The resulting string from the compression

decode :: Htree -> [Integer] -> String
decode tree bits = decodeTraverse tree tree bits []


-- traverse the string, and the tree for each char in the string
decodeTraverse :: Htree -> Htree -> [Integer] -> String -> String
decodeTraverse _ (Leaf c) [] str = str++[c]
decodeTraverse root (Leaf c) bits newStr = decodeTraverse root root bits (newStr ++ [c])
decodeTraverse root (Branch t0 t1) (b:bits) newStr =  i
    f b == 0
    then decodeTraverse root t0 bits newStr
    else decodeTraverse root t1 bits newStr


-- Tests ------------------------------------------------------------

testStatistics :: Bool
testStatistics = statistics "Huffman" == [(1,'H'),(1,'a'),(2,'f'),(1,'m'),(1,'n'),(1,'u')]


testWeightMakeTree :: Bool
testWeightMakeTree = getTreeW (makeWtree (generateWtreeList (statistics "Huffman") [])) == 7
```

```haskell


testMakeTree :: Bool
testMakeTree = show (maketree (statistics "Huffman")) == "Branch {h0 = Branch {h0 = Leaf {c = 'H'}, h1 = Leaf {c = 'f'}}, h1 = Branch {h0 = Branch {h0 = Leaf {c = 'u'}, h1 = Leaf {c = 'n'}}, h1 = Branch {h0 = Leaf {c = 'm'}, h1 = Leaf {c = 'a'}}}}"


testEncode :: Bool
testEncode =    let (_, bitList) = encode "Huffman"
                in bitList == [0,0,1,0,0,0,1,0,1,1,1,0,1,1,1,1,0,1]

-- their is no test for only decode as making one would still need to do all the work
-- encode does. And because encode has a unique test, one could
-- deduct that decode is wrong if testEncode returns true and this test returns false.
testEncodeAndDecode :: Bool
testEncodeAndDecode = uncurry decode (encode "Huffman") == "Huffman"
```