Huffman Encoding in Haskell

Algorithmic Problem Solving
Semester 1, 2008

Plan for today

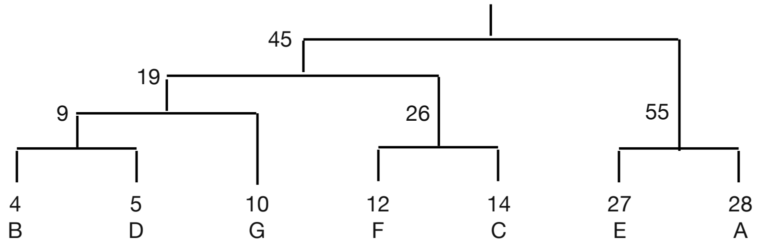
- Solving a non-trivial problem with Haskell: The data compression problem we discussed in the last lecture and its solution using Huffman encoding.
- QoT survey

Announcements

- Project 2 will be returned before the end of semester, hopefully this Wednesday.
- Wednesday I will cover binary search trees, accumulator recursion, and anything that you feel is missing.
- Thursday's lecture is review and, primarily, some talk about the exam.
- A practice exam will be made available on the LMS.

A Haskell implementation of Huffman encoding

- We need the tree structure (from the last lecture) which labels trees with their weight.



```
data Trie = Leaf Int Char
          | Node Int Trie Trie
```

Huffman codes and code tables

- Let us make our own type for bits:

```
data Bit = L | R
           deriving (Eq, Show)
```

- Huffman codes are lists of bits, and a table will map characters to their Huffman codes.

```
type HCode = [Bit]

type Table = [(Char,HCode)]
```

Coding a message

- To code a message according to a code table:

```
encode :: Table -> String -> HCode
encode tbl msg
  = concat (map (lookupTable tbl) msg)
```

- To look up the code of an individual character in a table:

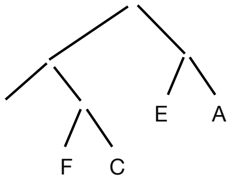
```
lookupTable :: Table -> Char -> HCode
lookupTable [] c
  = error ("lookupTable: No " ++ [c])
lookupTable ((ch,n):tb) c
  | (ch==c)   = n
  | otherwise = lookupTable tb c
```

Decoding a message

- To decode a message using a Huffman tree. The first argument is constant, being the tree of codes; the second gives the current position in the tree relative to the (partial) Huffman code read so far.

```
decode :: Trie -> HCode -> String
decode wholeTrie path
  = decodeByt wholeTrie path
    where
      decodeByt (Node _ t1 _) (L:rest)
        = decodeByt t1 rest
      decodeByt (Node _ _ t2) (R:rest)
        = decodeByt t2 rest
      decodeByt (Leaf _ c) rest
        = c : decodeByt wholeTrie rest
      decodeByt _ []
        = []
```

Try this out using the trie from before, on string RRLRRRL



Building the Huffman tree

- There are two stages: First calculate the frequency of each character, then build the trie:

```
codes :: String -> Trie
codes
  = makeTrie . charFrequencies
```

Calculating frequencies of characters

- The idea is to sort the string, then group like characters together, then find the length of each group:

```
charFrequencies :: String -> [(Char,Int)]
charFrequencies s
  = map freq groups
    where
      groups = group (sort s)
      freq g = (head g, length g)
```

Building the trie

- First convert each (character, frequency) pair to a single-node trie, then join the list of tries so they form a single trie:

```
makeTrie :: [(Char,Int)] -> Trie
makeTrie
  = makeCodes . toTrieList
```

- To generate the list of initial single-node tries:

```
toTrieList :: [(Char,Int)] -> [Trie]
toTrieList freqs
  = [Leaf n c | (c,n) <- freqs]
```

Building the trie

- From here on, Huffman codes are created bottom up: look for the two tries with smallest weights, graft these to make a new trie, and repeat until one trie remains.

- To find the weight of a trie:

```
weight :: Trie -> Int
weight (Leaf n _)
  = n
weight (Node n _ _)
  = n
```

Building the trie

- To graft two tries:

```
graft :: Trie -> Trie -> Trie
graft t1 t2
  = Node (weight t1 + weight t2) t1 t2
```

- To insert a trie into a list of tries sorted by increasing weight:

```
insertTrie :: Trie -> [Trie] -> [Trie]
insertTrie t []
  = [t]
insertTrie t (s:ss)
  | (weight t <= weight s) = t:s:ss
  | otherwise = s : insertTrie t ss
```

Building the trie

- To join the first two tries from a list of tries, and insert the result according to its weight:

```
join:: [Trie] -> [Trie]
join (t1:t2:ts)
  = insertTrie (graft t1 t2) ts
```

- To make the codes, we just need to join the whole list of tries, until we have a single trie:

```
makeCodes :: [Trie] -> Trie
makeCodes [t]
  = t
makeCodes ts
  = makeCodes (join ts)
```

Making the code table

- To make a code table from a Huffman tree:

```
codeTable :: Trie -> Table
codeTable t
  = convert [] t

convert :: HCode -> Trie -> Table
convert path (Leaf _ c)
  = [(c, reverse path)]
convert path (Node _ t1 t2)
  = convert (L:path) t1 ++ convert (R:path) t2
```

The first parameter is an *accumulator*: it accumulates the path, or code for each leaf character (but in reverse).

We call this style of recursion where one parameter is used to build up the result: *accumulator recursion*

Trying it out

- First we build the code table for string abracadabra:

```
trie = codes “abracadabra”
table = codeTable trie

? trie
 Node 11 (Node 4 (Node 2 (Leaf 1 'c') (Leaf 1 'd'))(Leaf 2 'r'))
(Node 7 (Leaf 5 'a') (Leaf 2 'b'))
? table
 [('c',[L,L,L]),('d',[L,L,R]),('r',[L,R]),('a',[R,L]),('b',[R,R])]
```

- Then we can encode and decode:

```
? encode table “abracadabra”
 [R,L,R,R,L,R,R,L,L,L,L,R,L,L,L,R,R,L,R,R,L,R,R,L]
? decode trie $$
 “abracadabra”
```

24 bits