Program Design & Data Structures (Course 1DL201)
Uppsala University – Spring 2021
Homework Assignment 3: Data Compression

Prepared by Johannes Borgström, Andreína Francisco, and Tjark Weber

Lab:	Friday 29 January 2021
Submission Deadline:	**17:00, Friday 5 February 2021**

Data Compression

A *binary digit* or *bit* is the basic unit of information in computing. A bit can only have one of two values, `0` (`False`) or `1` (`True`). On today's computers, all data (e.g., numbers, characters, strings) is ultimately encoded in bits.

Computer hardware, e.g., hard disks, can only store a limited number of bits. Also computer networks can only transmit a limited number of bits per second. When we want to store or transmit information, it is therefore often desirable to use an encoding that reduces the number of bits in the representation of the data. This is the purpose of *data compression.* Data compression is widely used to reduce the size of audio and video files, web pages, email attachments, messages sent by spacecrafts, and in many other applications.

For this assignment, we will focus on *lossless* compression. Lossless compression allows the original data to be reconstructed perfectly from the compressed data.

Huffman Coding

Huffman coding is a lossless compression technique based on the observation that certain characters typically occur much more often than others in a (natural-language) message.[1] For instance, in a typical English text, the character `e` would appear much more often than the character `z`. Instead of using a *fixed-length* encoding that uses the same number of bits to represent each character, we can thus reduce the number of bits required to represent the message by using a *variable-length* encoding that assigns shorter codes to more common characters, while less common characters are assigned longer codes.

Huffman coding determines an optimal code for each character. It employs the following algorithm, which was developed by David A. Huffman (1952) while he was a PhD student at MIT, and is now used as a building block in many other compression algorithms, including ZIP, GZIP and JPEG.

1. Count the number of occurrences of each character in the message.

   For instance, if the message is the string "`this is an example of a huffman tree`" (without the quotes), we obtain the following character counts:

space	a	e	f	h	i	m	n	s	t	l	o	p	r	u	x
7	4	4	3	2	2	2	2	2	2	1	1	1	1	1	1

[1]Here, we will focus on compressing strings as messages. Huffman coding can similarly be used for other data.

2. From these character counts, build a *Huffman tree.* A Huffman tree is a full binary tree such that

   - each leaf is labeled with a character;
   - each sub-tree (i.e., each leaf and each node) is labeled with the count of all characters in that sub-tree; and
   - sub-trees with larger character counts do not occur at a lower level of the tree than sub-trees with smaller character counts.

   Figure 1 shows a possible Huffman tree from (i.e., for the character counts of) the string "`this is an example of a huffman tree`".

   To be more specific, a Huffman tree for given character counts can be built by using a priority queue (whose data items are Huffman trees). The priority of each tree in the queue is given by the character count at its root node. Starting from an empty priority queue, the algorithm proceeds as follows:

   (a) For each character x with associated count c_x, insert a tree that consists just of a leaf labeled with x and c_x into the priority queue.

   Figure 2 shows a possible priority queue for the string "`this is an example of a huffman tree`" after all characters have been inserted as leafs.

   (b) If the priority queue contains at least two trees, remove the two trees with minimal priority, say t_1 with priority c_1 and t_2 with priority c_2. Merge them into a single tree by adding a new root node (labeled with $c_1 + c_2$) whose sub-trees are t_1 and t_2. Insert the new tree into the priority queue.

   Figure 3 shows the priority queue from Figure 2 after this merge operation has been applied once.

   Continue with step (b).

   (c) Otherwise, the only tree in the priority queue is the desired Huffman tree.

A Huffman tree defines a binary code (i.e., a list of bit values) for each character that appears in the tree as follows. Label each edge to a left sub-tree with `0` (`False`) and each edge to a right sub-tree with `1` (`True`) (see Figure 4). The *Huffman code* of a character x is the sequence of `0`/`1` labels from the root to the leaf that contains x.

For instance, given the Huffman tree in Figure 4, `e` is encoded as `[0,0,0]`, `n` is encoded as `[0,0,1,0]`, `o` is encoded as `[0,0,1,1,0]`, etc.

Compression To compress a message (i.e., a list of characters), we first need a Huffman tree that contains all the characters that appear in the message. This can be an existing Huffman tree, or one that is generated from the message. We then traverse the Huffman tree to build a *code table* that maps each character to its Huffman code.

Using this code table, we look up the Huffman code for each character in the message, and simply concatenate all codes to obtain the encoding of the message.

For instance, given the Huffman tree in Figure 1, the encoding of "`this is an example of a huffman tree`" would be a bit list starting with `[0,1,1,0,1,0,1,0,1,0,0,0,...]`.

Decompression To decompress a sequence of bits into the original message, we also need to know the Huffman tree that was used for compression. The algorithm for decompression then proceeds as follows:

1. Start at the root of the Huffman tree.

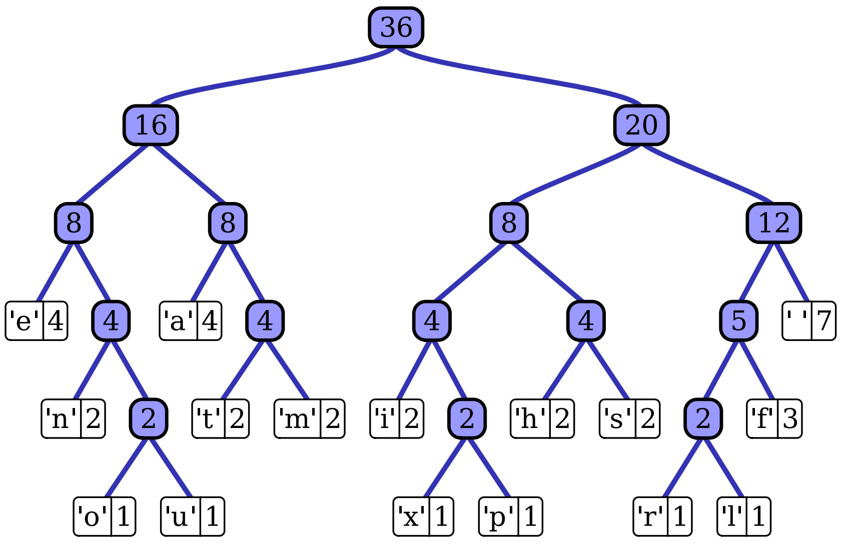

Figure 1: A possible Huffman tree from the string "`this is an example of a huffman tree`".

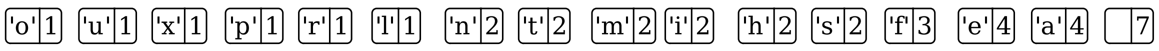

Figure 2: A possible priority queue from "`this is an example of a huffman tree`" after insertion of each character (step (a) of the algorithm).

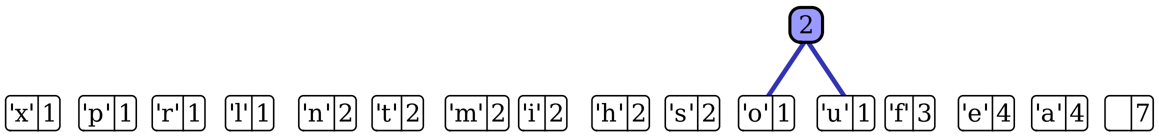

Figure 3: The priority queue from Figure 2 after the merge operation (step (b) of the algorithm) has been applied once.

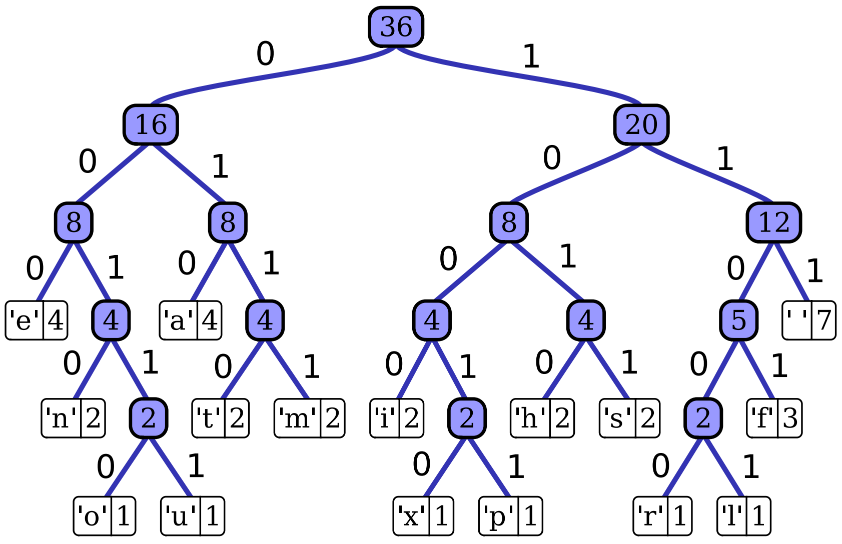

Figure 4: A possible Huffman tree from the string "`this is an example of a huffman tree`" with edge labels shown: `0` for left sub-trees, `1` for right sub-trees. Note that the labels are only shown for clarity—it is *not* necessary to actually store them in the tree data structure.

2. For each bit in the sequence:
   (a) If it is a `0`, go to the left sub-tree. If it is a `1`, go to the right sub-tree.
   (b) When we have reached a leaf, return the corresponding character. Go back to the root of the tree and decompress the remaining bits in the sequence.

Work to be Done

For this assignment, you will implement Huffman coding in Haskell.

Download the files `Table.hs`, `PriorityQueue.hs` and `BinomialHeap.hs` from Studium. The polymorphic type `PriorityQueue a` of priority queues holding data of type `a` (with priorities of type `Int`) has the following interface:

```
-- the empty priority queue
empty :: PriorityQueue a

{- isEmpty q
   RETURNS: True if and only if q is empty
 -}
isEmpty :: PriorityQueue a -> Bool

{- insert q (x,p)
   RETURNS: the queue q with element x inserted at priority p
 -}
insert :: PriorityQueue a -> (a, Int) -> PriorityQueue a

{- least q
   PRE:  q is not empty
   RETURNS: ((x,p), q'), where x is an element of minimum priority in q,
            p is the priority of x, and q' is q without x
 -}
least :: PriorityQueue a -> ((a, Int), PriorityQueue a)
```

Do not modify these files.

Download the file `Huffman.hs` from Studium. Define a data type `HuffmanTree` to represent Huffman trees, add missing comments, and implement the following functions (empty implementations are already provided):

```
{- characterCounts s
   RETURNS: a table that maps each character that occurs in s to the
         number of times the character occurs in s
 -}
characterCounts :: String -> Table Char Int

{- huffmanTree t
   PRE:  t maps each key to a positive value
   RETURNS: a Huffman tree based on the character counts in t
 -}
huffmanTree :: Table Char Int -> HuffmanTree
```

```
{- codeTable h
   RETURNS: a table that maps each character in h to its Huffman code
 -}
codeTable :: HuffmanTree -> Table Char BitCode

{- encode h s
   PRE: All characters in s appear in h
   RETURNS: the concatenation of the characters of s encoded using the Huffman code table
 -}
encode :: HuffmanTree -> String -> BitCode

{- compress s
   RETURNS: (a Huffman tree based on s, the Huffman coding of s under
         this tree)
 -}
compress :: String -> (HuffmanTree, BitCode)

{- decompress h bits
   PRE:  bits is a concatenation of valid Huffman code words for h
   RETURNS: the decoding of bits under h
 -}
decompress :: HuffmanTree -> BitCode -> String
```

Make sure that compression and decompression are inverse operations, i.e., that

```
let (h, bits) = compress s in decompress h bits
```

evaluates to `s` for all possible input strings. If `h` is a Huffman tree that contains each character that appears in `s`, we should also have `s == decompress h $ encode h s`. Pay special attention to edge cases, such as `""` or `"xxx"`, and if necessary adjust your implementation so that it handles these cases correctly.

Submit (your modified version of) `Huffman.hs` via Studium. Do not change the file name.

Running and Testing

There are (at least) two ways of loading your program into GHCi:

- Type `ghci Huffman.hs` in a shell.
- Type `ghci` in a shell. Then type `:l Huffman.hs`.

After modifying and saving your code in an editor, enter `:r` within GHCi to reload the file.

Several test cases have been provided for you in the file `Huffman.hs`. These can be run by entering `runtests` in the GHCi shell. You are strongly encouraged to add further test cases to test your code more thoroughly. (When we mark your code, we will in any case run a fresh copy of the test cases provided.)

Grading

Your solution is graded on a U/6–10 scale based on two components: (1) functional correctness and (2) style and comments.

1. Functional correctness:

Your program will be run on an unspecified number of grading test cases that satisfy all preconditions but also check boundary conditions. Each test case is based on *different* inputs (messages etc.) than the ones provided.

We will run these tests automatically, so be careful to match exactly the imposed file names, function names, and argument orders—that is, don't change what we've provided for you.

Advice: Run your code in a freshly started Haskell session before you submit, so that declarations that you may have made manually do not interfere when you test the code.

The grade for this component is determined as follows:

- If your solution was submitted by the deadline, your file `Huffman.hs` loads in GHCi and it fails at most one of the *provided test cases* in `Huffman.hs`, an it also passes at least 70% of the *grading test cases*, you get (at least) a 3 for functional correctness.

  Otherwise (including when no solution was submitted by the deadline), you get a U grade for the homework assignment.

- If your program passes all of the *provided test cases*, and it passes at least 85% of the *grading test cases*, you get (at least) a 4 for functional correctness.

- If your program passes all *provided test cases* and *grading test cases*, you get a 5 for functional correctness.

2. Style and comments:

Your program is graded for style and comments according to our *Coding Convention.* The following criteria will be used:

- suitable breakdown of your solution into auxiliary/helper functions,
- function specifications and variants,
- datatype representation conventions and invariants,
- code readability and indentation,
- sensible naming conventions followed.

The grade for this component is determined as follows:

- If your program's style and comments are deemed a serious attempt at following these criteria, and there are at most two significant errors or omissions in each function comment, you get (at least) a 3 for style and comments. Otherwise, you do not pass the homework assignment (U).

- If you have largely followed these criteria, with very few significant omissions or errors, you get a 4 for style and comments.

- If you have followed these criteria with at most minor omissions or oversights, you get a 5 for style and comments.

A significant error or omission is one that directly contradicts the coding conventions, or would cause a user of your library to pick the wrong function, or cause code using the function in question to compute the wrong result, or not terminate, for some inputs.

Final Grade

Component scores are converted to a pass/fail outcome and a final grade as follows:

1. You need to pass both components (functional correctness and style and comments) in order to pass this assignment.
2. Your final grade is the sum of the two component grades.

Modalities

- The assignment will be conducted in groups of two students. Groups have been assigned via Studium. *If you cannot find your partner by Tuesday 26 January, please contact Johannes Borgström* `<johannes.borgstrom@it.uu.se>`*. You will then be assigned a new partner (if possible).*
- Assignments must be submitted via Studium. Only one solution per group shall be submitted. Ensure that **both** group members' names appear on all submitted artefacts.

By submitting a solution you are certifying that it is solely the work of your group, except where explicitly attributed otherwise. We reserve the right to use plagiarism detection tools.

Good luck!