Assignment 3: Binary trees and Huffman coding

Date: 22-02-2017
Deadline: 03-03-2017 23:59

Objectives

You must implement a data (de)compressor and exercise your understanding of binary trees.

Requirements

Your deliverable must contain two programs `encode` and `decode`, behaving as detailed in the next section.

You must submit your work as a tarball [1]. Next to the source code, your archive must contain a text file file named "AUTHORS" containing your name and Student ID(s).

Behavior of the encoder and decoder programs

The `encode` program must:

- accept an optional command-line argument that specifies which tree it should use -- if this argument is not specified, the `encode` program should decide a Huffman tree itself;
- read data to encode from its standard input; and
- produce on its standard output:
  - on the first line, a representation of a Huffman tree it used, using the format documented for the `print_tree` function;
  - starting from the 2nd line, the encoding of the input data in ASCII-coded binary, that is using one full character "0" for binary 0 and one full character "1" for binary 1;
  - at the end of the encoded data, the final marker character "`.`" followed by a newline character;
  - on the last line of output: the number of characters from the input that were encoded, the number of nodes in the Huffman tree, the number of binary digits in the encoded output, and the compression ratio as percentage (rounded down).

The `decode` program must read data from its standard input:

- on the first line, a representation of a Huffman tree using the same format as `encode`;
- starting from the 2nd line, the encoded input data in ASCII-coded binary, terminated by "`.`";
- the remainder of the input, if any, is silently discarded.

Then prints on its standard output the result of decoding the provided input using the provided tree.

Order of work (strongly suggested)

1. Implement the missing `print_tree()` function which represents its tree argument using RPN notation:
   - a single node tree with node value `X` is printed as X.
   - the binary tree with two children X and Y is printed by printing X, then printing Y, then printing ".".

   For example this tree:

```
(root)
 /  \
A   / \
   B   C
```

Will be printed as: `ABC..`

And this tree:

```
  (root)
  /    \
 / \    C
A   B
```

Will be printed as: `AB.C.`

You can test your `print_tree()` at this point by uncommenting the `print_tree()` call in `decode.c` and running `./decode` with any kind of input. This will use the fixed tree generated by `fixed_tree()`. Verify that your output matches that tree.

2. Using the provided example Huffman tree in the code as constant tree input (ignoring the command-line argument), complete:
   - the definition of the `code` struct (you need to decide this yourself);
   - the function `compute_code_table()` which translates a tree to a code table,
   - the function `print_code()` which prints the encoded sequence of `0` and `1` character for each input character.

   This way, the provided `encode` program can use both your `print_tree` function from step 1 and your algorithm in this step to produce a coded tree and a coded input valid for the provided `decode.ref` program. You can then use `decode.ref` to check whether your work up to this point is correct. The command `echo "abca" | ./encode | ./decode.ref` should print `abca`.
3. Again using the provided example Huffman tree as constant (ignoring the first line of input), complete the `main()` function of `decode` to decompress input data using that tree.

   You can then use your `encode` program from step 2 to check your newly minted `decode` program.
4. Complete your `decode` program by implementing the missing `load_tree` function which reads a tree definition created by `print_tree` and re-creates the corresponding tree. Hint: you may want to use the generic stack implemented in `stack.c`.

   Then you can use the provided `encode.ref` to check that your decode program can now handle inputs with different trees. Check if the trees printed by `encode.ref` and `decode` match. The command `echo "xxyzzz" | ./encode.ref | ./decode` should print `xxyzzz`.
5. Complete your `encode` program by writing the `compute_tree()` algorithm that creates an optimal Huffman tree from the input, instead of using the provided example.

Grading

Your grade starts from 0, and the following tests determine your grade:

- +0,5pt if you have submitted an archive in the right format with an `AUTHORS` file.
- +0,5pt if your source code builds without errors and you have modified `tree.c`, `encode.c` or `decode.c` in any way.
- +1pt if your `print_tree` function works.

- +2pt if your `encode` program works using only the provided example tree.
- +2pt if your `decode` program works using only the provided example tree.
- -1pt if `valgrind` reports errors while running your converter.
- -1pt if `clang -W -Wall` reports warnings when compiling your code

The following extra features will be tested to obtain higher grades, but only if you have obtained a minimum of 5 points on the list above already:

- +1pt if your `load_tree` function works properly.
- +1pt if your `decode` program works using arbitrary Huffman trees provided as input.
- +2pt if your `encode` program constructs minimal Huffman trees for arbitrary inputs.

Overview of Huffman coding

Algorithm to encode the data:

1. Compute frequency table of input
2. Translate the frequency table to a tree - This is where the `encode` program in this assignment also prints out the coding tree.
3. Translate the tree to an encoding table
4. Use the encoding table to encode the data - This is where the `encode` programs emits the encoded output.

---

1 http://lmgtfy.com/?q=how+to+make+a+tarball