CSE250 Fall 2016
Assignment A6 – Huffman Coding

Due: 11/27/2016, 11:59PM

Last updated: *2016-11-16 23:19*

Objectives

- Practice design and implementation of tree-based algorithms.
- Get some exposure to a priority queue.
- Implement a fun tool.

Introduction

Huffman coding is one of the most important and elegant techniques in information theory. For a given sequence of input symbols, and their counts, it builds a binary tree that can be used to generate the optimal binary encoding of each symbol. The basic algorithm to build a Huffman tree can be summarized as follows:

1. Create leaf node for every input symbol and its count.
2. Take two nodes with the lowest count.
3. Attach them to a new parent node that stores a placeholder symbol with the count equal to the sum of counts in children.
4. Repeat steps 2 and 3 until all nodes are connected.

Your task is to implement a function that will build a Huffman tree for a given input sequence of symbols, a function that will print code-words encoded in the tree, and function to release the memory occupied by the tree.

Instructions

1. Create directory `A6` where you will place your code.
2. Download `A6-handout.tar` from:
   http://www.jzola.org/courses/2016/Fall/CSE250/A/A6-handout.tar.
3. Untar handout, and move `a6.cpp`, `a6.hpp` and `symbol.hpp` to your `A6` directory. These files provide all functionality you will need.
4. Analyze `symbol.hpp`, which provides helper structures you will be using in your code.
5. In `a6.hpp` implement missing functions considering the following:
   (a) The `huffman_tree` function should return a pointer to a root node of the Huffman tree for a sequence of symbols in $[first, last)$.
   (b) Input symbols are represented via type `symbol` (inspect `symbol.hpp`).
   (c) You can safely assume that the input range is valid: it contains at least two symbols, and count of every symbol is greater than zero.

(d) Tree consists of nodes of type `bnode<symbol>` (inspect `symbol.hpp`).

(e) For a given node in the resulting Huffman tree, the left child must store symbol that is "less than" the symbol in the right child as prescribed by the `operator<` for `symbol` (inspect `symbol.hpp`). The value of a symbol (i.e. specific character) stored in the node should be minimum between the values of its children.

(f) The `release_tree` function should release the entire memory used by tree `root`.

(g) The `print_dictionary` function should print to `os` all code-words encoded in the Huffman tree represented by `root`. Each code-word should be printed as shown below. Note that `std::ostream& os` defines reference that essentially behaves the same way as `std::cout`. So for example, this code `os << "A";` will have the same effect as `std::cout << "A";`.

(h) To test your code you can use two files provided in the handout: `dummy.txt` and `loremipsum.txt`. You can see correct answers for both files in `dummy.ans` and `loremipsum.ans`. Note that the order in which you print your tree is irrelevant, however, code-words must match exactly. So for example, if you run `./a6 dummy.txt` you should see (with lines possibly in a different order):

```
B 000
U 001
H 01
A 1
```

Note that symbol and code-word should be separated by a single space and code-word should be written as a string of `0` and `1`.

Extra Challenge

This submission has extra challenge. Once you have working `a6.hpp` you can consider `a6e.cpp` and `encode.hpp` files provided in the handout. Essentially, `a6e.cpp` is a program that builds Huffman tree (using your code) and then invokes function `encode` that for a given input string will create string with its Huffman encoding. Your task is to implement `encode` in `encode.hpp` considering the following:

1. `M` is the text to encode and `root` is the corresponding, ready to use Huffman tree.
2. For a given symbol in `M` the output string should include string of `0`s and `1`s representing code-word for that symbol. For example, if `M="aaabb"` then the output should be `"11100"`, and for the text in `dummy.txt` your function should return `"00000110110110110111111 1"`.
3. You can use `a6e` to test your implementation of `encode`. Example correct output for `loremipsum.txt` is provided in `loremipsum.encode`.

Submission

1. Remove your binary code and other unrelated files (e.g. your test files).
2. Create a tarball with your `A6` folder.
3. Follow to https://autograder.cse.buffalo.edu and submit `A6.tar` for grading.
4. You have unlimited number of submissions, however, any submission after the deadline will have 50% points deducted.

Grading

- 10pt: `a6.cpp` compiles with your `a6.hpp`, runs and has no memory leaks.
- 90pt: If you pass the initial test, there will be nine benchmark tests. You will get 10pt for each correctly completed test.
- If your code is **extremely** inefficient, for instance due to infinite loop, autograder will terminate your code and you will receive 0pt.
- If you get 100pt, autograder will try to compile `a6e.cpp` with your `encode.hpp` and you will get 30pt extra if it compiles and runs correctly for three small tests.

Remarks

- Make sure that all files and directory names are exactly as instructed. Otherwise the grading system will miss your submission and you will get 0pt.
- I am guessing nobody reads this section anyway...