605.202: Introduction to Data Structures (Spring 2022)

Mukul Sherekar

Lab-3 Analysis

Due Date: April 19th, 2022

Date Turned In: March 19th, 2022

Justification of data structure/design:

Three data structures were used for this lab - node , heap and dictionary. A node (at one memory location) can have multiple properties tied to it at different memory locations. This gives immense flexibility to access properties and tremendous amount of ease to sort a node based on any property. In this use case, a node had an alphabet, frequency and two child nodes.

The second data structure used was a heap. A has a basic property being a minimum or a maximum heap. A minimum heap was implemented in this case because goal was to find the nodes with smallest frequencies and create a new complex node. This design allowed for easy extraction of minimum nodes and insertion of complex node.

The third data structure used was a dictionary to store alphabets and their codes. Although, it was not recommended by Dr.Cost but it was too late. I am don't know what are the disadvantages of using a dictionary are. I used to encode the given sentences i.e iterate over the sentence codify it using the dictionary. Maybe, finding the value based on key is not the most time efficient way. This is an area of improvement. I also don't know another method to do it.

Description of heap implementation and justification

Heap was the main crux of this lab. It was implemented using lists (I use python). My heap class initiated an empty list with first element as 0. This makes downstream task of defining parent and child nodes easier.

The main purpose of heap implementation was to ensure parent is always smaller than children nodes. For this purpose, finding smaller child nodes was crucial. This aspect also decided the "tie-breaker" in the case where alphabets (length, order, complexity) were the deciding factor (when frequency was equal). Hence a method called min root was made which compared child nodes and decided which child node should be used to swap with parent node.

Inserting in the tree and swaping are also inherent to a heap class. A new node was always inserted at top using append and also removed from from top using pop. Swapdown method used the minchild method to identify the smaller child and swap parent with child. Swapup also operated the same way.

Heaps were used to sort and sorting is successful when it is quick. For sorting to be quick, nodes have to move quickly and min heap provides excellent features. Also, heap was used to find codes for each alphabet. That is application of tree-traversal.

Description and justification of design decision

My module has __init__, __main__, lab3.py (main file that gets executed), and helper modules - convert.py, sort.py, codify.py, huffman_compress.py, huffman_decmpress.py and compression.py . Then finally two class modules - node.py, heap.py.

I designed the module to be executed in lab3.py where the user will input four files (frequency table , file to code, file to decode and output file) at the terminal. Then, process_files method gets executed

Then, frequency data is read and converted (using convert.py) to nodes. At the same time, these nodes are being inserted into a heap. Once all nodes are read and inserted into a heap, it is ready for sorting.

Then, a Hoffmann tree is created by heap sort method from sort.py.

Then, root of this huffman tree is extracted and this becomes input of codify method to get a dictionary of codes.

Then, this dictionary of codes and lines of the given file fed into compress function.

Finally, huffman tree root and line to decode are fed into decompress function.

Also, the compression function calculates % compression.

Time and space efficiency

Huffman encoding for this assignment was implemented using a heap sort. A heap sort is a comparison sort with a worst case and average case run time complexity of O(NlogN) and best case run time complexity of O(N). (N is number of files). Huffman encoding itself runs at worse case scenario of O(NlogN) where is N is number of characters to code.

What did I learn?

- Use, implementation and meaning of a node
- Use, implementation and meaning of a binary tree, heap, sorting, tree traversal, huffman tree
- I was introduced to concepts of compression and encoding

What would I do differently?

- Not waste time on futile attempts to implement in-place heap sort
- Not use dictionary to store codes
- Use better outputting formats

Analysis scenarios from the handout:

Yes, I did achieved useful data compression - 47.66%, 44.93%, 45.83%, 44.05% for the given four sentences.

When I switch the preferences i.e giving precedence to alphabets first and then to frequency, compression, codes and compression% remains the same but decoding changes for two lines. The change is just in ordering of alphabets and not in type of alphabets.

When I use a different frequency table, randomly generated by me, new compression % are lesser - 41%, 25.6%, 33.85%, 31.55%. This shows importance of frequencies using a specific text source.

When I tried to encode letters A-Z (2 x 26), compression 36.54%. Compression % remained the same if I had 4 x26 or 8 x26 (A-Z) letter. This shows limitation of huffman encoding with respect to repetitions in the text.

When I used different texts to encode, I realized I didn't have code to tackle integers. Also, when I tried to encode text from internet, I got error saying some codec error. This tells me some formatting needs to be done to encode text copied from internet. But one the texts did encode properly(included in output folder).

Comparison with conventional encoding

I don't know what's conventional encoding in the world of Computer Science and couldn't figure out examples of conventional encoding. Scott Almes gave an example of simple encoding where each letter is replaced by another letter.

Compared to this simple coding, huffman coding is much more sophisticated. The code itself is dependent on a certain text which is known to sender and recipient only. In a way huffman coding is a two layered encoding - creation of a frequency table and code itself. As an example, same combination of 0s and 1s can be decoded differently based on two different set of codes.