Data Structures: Lab 3

Following the American Psychological Association's Guidelines

Markham M. Shofner

Johns Hopkins University

Data Structures: Lab 3

Analysis Document

The primary data structures I used in this lab were: a binary tree class, a tree node class, a frequency data class, and an array of trees. The binary tree class tracks the root of the tree and where we are in the tree (for an actions). The binary tree class offers constructors based on a frequency data item, two existing trees, or nothing. The binary tree class also supports classic methods such as MakeTree, SetRight, SetLeft as well as additional ones such as DisplayTraverse, PrintInfo and others. The tree node class houses the Data, Left, and Right pointers that comprise the tree itself, as well as a String that tracks the binaryTrace needed to get to a certain node. The frequency data class houses String values and frequencies, and can combine existing frequency data to create a new aggregated data structure. Lastly, I used an array (with a simple insertion sort) to combine nodes and order the nodes (based firstly on frequency, then size, then alphabetically).

Using a binary tree seemed the classic way to solve the Huffman encoding problem, and a natural extension of the logic required in that space. In terms of implementation, I decided to use a linked implementation for the tree (as opposed to an array-based implementation) because we aren't guaranteed that any of the trees will be complete or almost complete (which is the ideal scenario for arrays in this sense because it would end up wasting the least amount of space).

In terms of process, this one took me the longest, but ended up being the most fulfilling (thus far!). I rewatched the lectures on binary trees and pulled a lot of the base implementation for a tree (using a linked list) directly from the lecture and built out the rest from there. Compared to the previous two labs, I spent more time on the frontend sharpening my axe and planning out the road ahead, but – as always – probably could have spent more time here. It is tough to fully plan out a complicated app ahead of time though. There is so much information that one picks up on throughout the build process that informs future decisions. So I guess in terms of what I would do differently next time knowing that, I think it would make sense to stop periodically and do a full reassessment of where I am given all the new information gained in the meantime, and deciding if the current route is actually valuable, or whether I am pursuing it simply because it is the one I am one. Making aggressive and surgical design decisions periodically can end up saving a lot of development time, even if it means throwing away something that currently may be interesting to work on.

Regarding efficiency, using a linear implementation saved me space, and also some traversal time. Admittedly, parts of this app are not the most efficient. As with many Huffman encoding tables, this was a nice way to use fewer bits to represent more information with fewer bits. The height of the tree (and thus the longest path of a binaryTrace) was lower than the standard amount of bits needed to represent a character in ASCII or a Binary Character Table (`01000101` [8 bits] would normally be needed to represent the letter E) but here we can do it in 3. In terms of tie-breaks, flipping the orders around would change things, but not substantially or always. Given the way to tree was constructed (and mainly how the frequency table is already read alphabetically), the alphabetical tie was not used. Though if we changed the way the frequency table was written, then this would have potentially changed how the alphabetical tie break sort could be leveraged.

Finally, and kind of embarrassingly, it took me quite some time to realize that EIEIOH was a correct result. May be slightly more user friendly to future students if the first decoding is something that is more clearly a word or phrase. Thankfully MERLIN snuck in there at the end to show me the light.

References

zyBooks. 605.202: Data Structures. Module 7 – Trees.

zyBooks. 605.202: Data Structures. Module 10 – Sorting and Searching Sorted Data.