Computer Science 303: Data Structures Project #2B

attachments and source available at https://github.com/alexskc/cs303

Aleksander Charatonik

November 26, 2018

1 Encoding algorithm

Given a text, convert it to morse. This one is straightforward, and, unfortunately, inefficient.
1. Open morse.txt file
2. Convert every line to a string in a vector
3. Go through every letter in the string to encode
4. Look at the first letter of each array element until we find a matching letter.
5. Append the corrosponding morse (all the characters in that array element after the first) and a space.

This means we are potentially going through the entire array for every letter, giving us a time complexity of $O(m * n)$, where m is the number of letters in the alphabet, and n is the length of the string. A preferable solution would be a map, which would have a complexity of $O(\log(m))$ for lookup, so $O(n * \log(m))$.

2 Tree-building algorithm

This takes the text in morse.txt and builds a tree out of it, as provided in the assignment description.
1. Open morse.txt file, and convert every ine to a string in a vector.
2. Sort the vector by length of morse code. Shortest to longest. This avoids complexity later on.
3. Go down the vector, building the tree as we do so.
3a. If we see a dot, go to the left subtree. If we see a dash, go to the right subtree. 3b. If a subtree does not exist, create it with the assigned letter. This will always be a lear (at least at first) because the list is sorted, so we don't have to wrorry about not knowing a value.

This is reasonably efficient. The time complexity of appending a node is $O(\log(n))$, where n is the number of letters. The sort used is std::sort, which (likely, this depends on implementation) is $O(n \log(n))$, and reading the text file is of course $O(n)$, giving us a time complexity of $O(n \log(n))$.

3 Decoding algorithm

Given some morse code and a tree for decoding it, convert it back into letters.
1. If we see a ., go down the left subtree. If we see a dot, go down the right subtree.
2. Repeat until we get to a space.
3. Return the character at the current node.
4. Repeat for the whole text.

Straightforward. Going down the tree has a time complexity of $O(\log(n))$, where n is the number of letters in the alphabet, and we multiply that by the length of the message.

4 UML diagram

Not relevant. No new classes/models are introduced. This is a normal binary tree, with nodes holding chars, and dots/dashes symbolized by left/right.

5 Assumptions, mistakes, references

1. The tree-building algorithm and the encoding algorithm both directly interpret the text file, and make assumptions about the format. If the format were to change, we'd have to fix them as well. Ideally, reading in morse.txt should be decoupled from these.
2. The implementation of binary trees was taken from Canvas. 3. Standard library implementations of sort, vector, string, fstream and iostream were used.
4. The strings to encode/decode are hard-coded in main.cpp, rather than being input by the user.
5. Automated tests should have been written. They were not.
6. Sort is run twice (!) because of a quirk with std::sort.
7. There is no graceful failure for incorrect morse. Anything too long, or not mentioned in morse.txt will not work.