**Project 2B: Morse Code**

Matt Miller & Laureen Nsoh-Awasom

CS 303: Data Structures

Prof. Mayanka Chandra Shekar

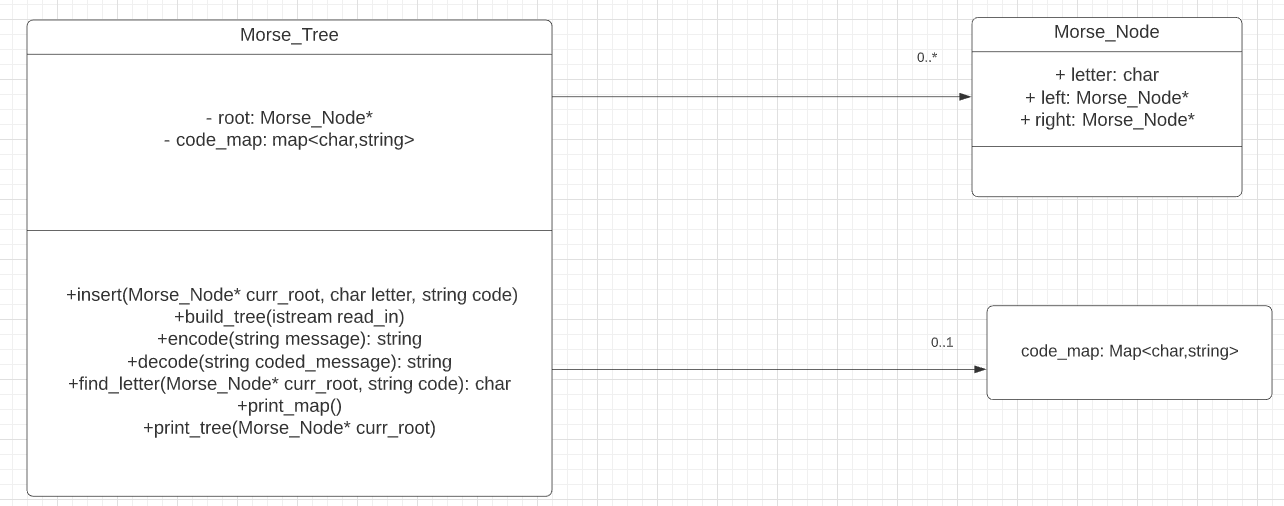
July 22, 2020

GitHub: <https://github.com/MattMiller1989/Morse-Decoder-Encoder>

Assumptions

* The file provided, “Morse.txt”, was used as the source file for providing the alphanumeric character and its Morse Code equivalence.
* The Morse Code uses the character: ‘\_’ as a dash and the character: ‘.’ As a dot.
* The code uses a space as a delimeter between the characters in the Morse Code.
* When the binary tree is built, the contents from the “Morse.txt” file are read from the file provided the value pair of char letter and string Morse Code equivalence are stored in a map with the letter being the key and the Morse Code as the value.
* The encode function will take an alphanumeric string and iterate through each character and look up the value of the character in the code\_map and return its Morse Code equivalence.
* The encode function will work with capital letters. However, when the code is decoded, only the lower case value is returned.

UML Diagram



Algorithms and Methods

**Morse\_Tree::insert(Morse\_Node \*&curr\_root, char letter, string code)-**

The time complexity for insertion into a binary-tree of Morse-Code equivalencies is , where n is the number of nodes to be inserted. This is because the maximum number of times the insert function needs to be called is directly proportional to the height of the tree, which will be . The reason for the “+1” at the end of the formula is because the root of the tree is left blank. Since the number of nodes is fixed at 26 for this program, the height of the tree used will be fixed at 5. It is also worth noting that the given height of a node is also directly related to the length of the Morse Code equivalent for each letter. The number of times the function is called for each node is equal to the length of its Morse Code. If we wanted to use a Morse Code file with more characters, the height of the tree would increase logarithmically with the formula . The algorithm that was used is a pretty standard insertion method for a binary tree, so there is little room for improvement for algorithm efficiency (Adamchik).

**Morse\_Tree::build\_tree(istream& read\_in)-**

This function takes in a string and stores each character of the string into the map of characters and their Morse Code equivalencies. It also takes that same value pair and calls the insert function to store each character as a node in the binary tree. The time complexity of this program is where m is the length of the message and n is the number of nodes to be inserted into the tree. This function loops through each character of the input string with a time complexity of . Within this loop, the insert function, with time complexity of . The time complexity of these two functions would be multiplies together, resulting in a total time complexity of . We made the choice to use this method to build both the map and the binary tree because this allows us to only have to iterate through the external Morse Code resource file one time, rather than have to do it once for the building of the map, and once for the building of the tree (Ozer).

**String Morse\_Tree::encode(string message)-**

The time-complexity of this function is O(m), where m is the number of characters in the message. The function iterates through each character of the message and uses that character as a key and returns the value associated with that key in the code map, which is its Morse Code equivalencies. The time-complexity of accessing the value of a key in a map is O(1), so the complexity of the entire function is simply linearly related to the size of the message. The requirement of the encode method was the primary factor in choosing to store the values of the Morse Code and letter inside of a map. If we chose to use the same Morse tree as we did in the decode method, we would have to use a breadth first search function to find the letter in the tree. This would have a time-complexity of O(n) where n is the number of nodes in the tree. This would result in the whole function having time-complexity of O(m\*n) where m is the length of the message to encode.

**String Morse\_Tree::decode(string coded\_message) -**

This method is very similar to the build tree function. The total time complexity of this function is , where m is the size of the message and n is the number of nodes in the tree. This function will iterate through each of the Morse Code characters in the string coded\_message. Each of these Morse Code characters are stored as a string and that string is passed on to the find\_letter function. This function has a time complexity of , where n is the number of nodes in the binary tree.

**String Morse\_Tree::find\_letter(Morse\_Node \*& curr\_root, string code) –**

This is a function that is recursively called to find the alphanumeric equivalent of the string of Morse Code provided. The time-complexity is where n is the number of nodes in the tree.

**Morse\_Tree::print\_map()**

This function has a time complexity of O(n) where n is the number of keys in the map.

**Morse\_Tree::print\_tree()**

This function uses in-order traversal to print all of the nodes of the tree. The time complexity for this function is O(n), where n is the number of nodes in the binary-tree.

References

Adamchik, Victor S. *Binary Trees*. 2009.

Ozer, Berk. *How Tree Data Structures Help Us Understand Morse Code*. 6 April 2020. 24 July 2020.