Question 2:
The code will not run because of 2 reasons:

- The data field is a string slice, which is a reference to a string. Rust's borrowing rules require that there must be a defined lifetime for the reference.
- The left_child and right_child fields are trying to directly store an instance of the TreeNode struct, which leads to a recursive type definition. Rust needs to know the size of the type at compile time, and this is not possible with recursive types.

Question 4:

- The insertion code for an enum-based tree will require adjustments to accommodate the Tree::Empty variant. When inserting a new value, if we encounter an Empty node, we need to replace it with a new Node variant containing the value. This change eliminates the use of Option<Box<TreeNode>>, as the Empty variant inherently represents the absence of a node. Consequently, the pattern matching in insert_node method will match against Tree::Node and Tree::Empty instead of Some and None. The rest of the insertion logic remains similar, recursively traversing the tree and choosing the left or right child based on the value.
- The Empty variant in the enum serves a specific purpose: it represents a null or absent child node in the tree. This is a more Rust-idiomatic way of indicating that a particular branch of the tree does not exist. It essentially takes the place of None in an Option<Box<TreeNode>>. By using Empty, we provide a clear and explicit representation of a leaf node's absence, which can help prevent errors that might occur from mistakenly assuming that a node is present.
- The enum-based solution is considered more ideal in Rust and is typically clearer. The Empty variant is superior to an Option<Box<TreeNode>> in expressing the absence of a node because it is more explicit and integrated into the type system. With Empty, the type system enforces the handling of the case where a child node does not exist, whereas Option<Box<TreeNode>> requires additional unwrapping and can be more error-prone due to the implicit nature of None. Furthermore, enums in Rust allow for pattern matching, which leads to safer and more maintainable code by exhaustively handling all cases. Therefore, the enum-based solution not only aligns better with Rust's design principles but also offers improved safety and clarity, which can be especially beneficial as the complexity of the data structure grows.