Documentation for the Solution

The problem requires us to determine the length of the longest absolute path to a file in a file system represented as a string. The file system is structured hierarchically, with directories and files separated by newline characters (\n), and the nesting depth is indicated by tab characters (\t). Each file or directory has a unique absolute path, which is the sequence of directories leading to it, concatenated by / characters. The goal is to parse this string representation and compute the length of the longest absolute path to a file. If no files exist, the function should return 0. This problem simulates traversing a directory tree, and the challenge lies in efficiently managing the hierarchical structure and calculating path lengths.

The intuition behind the solution is to use a stack to keep track of the cumulative length of the directory structure at each level. By maintaining a stack, we can simulate the directory hierarchy and efficiently compute the length of absolute paths. We determine each line in the input's depth by counting the number of tabs. If the current depth is less than the stack size, it means we have moved up in the directory hierarchy, and we adjust the stack accordingly. For files, we calculate the total path length by adding the length of the file name to the length of the parent directory path (stored at the top of the stack). We push the current path length (including a / separator) onto the stack for directories. This approach ensures that we correctly track the directory structure and compute the required path lengths.

The stack is a critical component of this solution. It helps us manage the hierarchical structure of directories and files by storing the cumulative lengths of directory paths at each level. When we encounter a line with a depth less than the stack size, it indicates that we have moved up in the directory hierarchy. In such cases, we pop elements from the stack until the stack size matches the current depth. This ensures that the stack always reflects the correct directory structure. We push the current path length onto the stack for directories, and for files, we use the stack to compute the total path length. This mechanism allows us to dynamically adjust the stack based on the depth of the current line, ensuring accurate path length calculations.

The time complexity of this solution is O(n), where n is the length of the input string. This is because we process each line exactly once, and the stack operations (push/pop) are performed in constant time. The space complexity is O(d), where d is the maximum depth of the directory structure. This is because the stack stores the cumulative path lengths at each level, and its size is proportional to the maximum depth of the directory hierarchy. This makes the solution efficient and scalable, even for large inputs.

The solution handles various edge cases effectively. For example, if the input contains no files (e.g., "a"), the function correctly returns 0. It also handles deeply nested directories and files, ensuring that the stack correctly manages the hierarchy. Additionally, it works for inputs with multiple files at different depths, accurately computing the maximum path length. These edge cases are handled seamlessly by the stack-based approach, which dynamically adjusts to the structure of the input.

In conclusion, this solution provides an efficient and elegant way to solve the problem by leveraging a stack to manage the directory hierarchy and compute the required path lengths. The approach is both time and space-efficient, making it suitable for handling the problem constraints effectively. By carefully processing each line and maintaining the stack, we ensure that the solution is robust and handles all edge cases correctly. This method not only solves the problem but also demonstrates the power of using data structures like stacks to manage hierarchical data.