Documentation: Solution for Removing Duplicate Letters to Form the Lexicographically Smallest Result

The problem of removing duplicate letters to form the lexicographically smallest result involves balancing two requirements: ensuring each character appears only once and maintaining the smallest possible order. Given a string s, the goal is to construct a result string that satisfies both conditions. For instance, from s = "cbacdcbc", the expected output is "acdb". The solution must achieve this efficiently, as the string can have up to 10,000 characters.

The primary challenge is deciding when to retain or remove a character while processing the string. A straightforward approach like sorting the characters would violate the order in the original string. At the same time, a brute-force method of checking all permutations would be computationally infeasible for large inputs. This necessitates a more structured approach that leverages efficient data structures and algorithms.

A greedy algorithm combined with a stack-based structure is used to address these challenges. The stack dynamically builds the result string while ensuring lexicographical order. The algorithm iterates through each character in the string, evaluating its inclusion in the result based on two conditions: whether it is already present in the result and whether removing the top character of the stack (if larger) would allow it to reappear later.

The process involves three main data structures. A "last occurrence" map records the last index of each character in the string, enabling the algorithm to determine if a character can safely be removed. A "visited set" tracks the characters already in the result to prevent duplicates. Lastly, a stack is used to maintain the final sequence of characters, ensuring that additions and removals preserve the required order.

As the algorithm iterates through the string, it decides whether to skip a character if it is already in the stack. If the current character is smaller than the top character of the stack and the top character appears again later, the top character is removed. The current character is then added to the stack and marked as visited. This process continues until all characters are processed.

The final result is obtained by concatenating the characters in the stack. This ensures the output string is lexicographically smallest and contains no duplicate characters. The use of the stack guarantees that the solution efficiently builds the result in a single pass through the string, with no unnecessary operations.

The algorithm runs in O(n) time complexity, where n is the length of the string. Each character is pushed and popped from the stack at most once, making it highly efficient. The space complexity is O(1) additional space, as the stack and visited set contains at most 26 characters, limited by the alphabet. This design ensures the solution is scalable and optimal for large inputs.