316. Remove Duplicate Letters

Medium Stack Greedy String Monotonic Stack Leetcode Link

Problem Description

The problem at hand is to take an input string `s` and remove duplicate letters from it, such that the resulting string contains each letter only once. However, there's an additional constraint: the resulting string must be the smallest possible string in lexicographical order. Lexicographical order is essentially the dictionary order or alphabetical order. So, if given two strings, the one that comes first in a dictionary is said to be lexicographically smaller. Our task is to ensure that, among all the possible unique permutations of the input string's letters, we pick the one that's smallest lexicographically.

Intuition

The intuition behind the solution is to construct the result string character by character, making sure that:

- Each letter appears once, and
- The string is the smallest in lexicographical order.

To accomplish this, we use a **stack** to keep track of the characters in the result string while iterating over the input string. The algorithm uses two data structures—a stack and a set:

- The stack `stk` is used to create the result string.
- The set `vis` keeps track of the characters currently in the stack.

As we go through each character in the input string, we perform checks:

- If the current character is already in `vis`, we skip adding it to the stack to avoid duplicates.
- If it's not in `vis`, we check whether it can replace any characters previously added to the stack to ensure the lexicographically smallest order. We do this by popping characters from the stack that are greater than the current character and appear later in the string (their last occurrence in the string is at an index greater than the current index).

By using the stack in this way, we're able to maintain the smallest lexicographical order while also ensuring we don't skip any characters that must be in the final result because we know where their last appearance is in the string, thanks to the `last` dictionary.

The process of building the result string incorporates the following steps:

1. If current character is not in `vis`, continue to step 2. Otherwise, go to the next character.
2. While there is a character at the top of the stack that is lexicographically larger than the current character and it appears again at a later index, pop it from the stack and remove it from `vis`.
3. Push the current character onto the stack and add it to `vis`.

Once we finish iterating over the string, the result is constructed by joining all characters in the stack. This result is guaranteed to be void of duplicates and the smallest lexicographical permutation of the input string's characters.

Solution Approach

The solution is implemented in Python, and it employs a stack, a set, and a dictionary comprehensively to ensure that duplicates are removed and the lexicographically smallest order is achieved.

Here's how the implementation breaks down:

1. We first create a dictionary `last` which holds the last occurrence index of each character in the string. This helps us to know if a character on the stack can be popped out or not (if a character can be found later in the string, perhaps we want to remove it now to maintain lexicographical order).

```
last = {c: i for i, c in enumerate(s)}
```

2. We then initialize an empty list `stk` which acts as our stack, and an empty set `vis` which holds the characters that have been added to the stack.

```
stk = []
vis = set()
```

3. We iterate over the characters and their indices in the string. For each character `c`:
   - If `c` is already in `vis`, we continue to the next character as we want to avoid duplicates in our stack.
   - If `c` is not in `vis`, we compare it with the characters currently in the stack and see if we can make our string smaller by removing any characters that are greater than `c` and are also present later in the string (`last[stk[-1]] > i`).
   - This is done in a `while` loop that continues to pop characters from the top of the stack until the top character is smaller than `c`, or there is no more occurrence of that character later in the string.

```
for i, c in enumerate(s):
    if c in vis:
        continue
    while stk and stk[-1] > c and last[stk[-1]] > i:
        vis.remove(stk.pop())
    stk.append(c)
    vis.add(c)
```

4. Finally, we join the characters in the stack and return the resulting string. The stack, at this point, contains the characters of our resulting string in the correct order, satisfying both of our conditions: no duplicates, and smallest lexicographical order.

```
return ''.join(stk)
```

The key algorithmic patterns used in this solution are a stack to manage the order of characters and a set to track which characters are in the stack to prevent duplicates. The use of a dictionary to keep track of the last occurrences of characters helps to decide whether to pop a character from the stack to maintain the lexicographically smallest order.

In conclusion, the code systematically manages to satisfy the requirements of the problem statement by ensuring unique characters using a set and optimizing for the lexicographically smallest output by intelligently popping from a stack with the aid of a dictionary that keeps track of characters' last positions.

Example Walkthrough

Let's walk through the solution with a simple example. Consider the input string `s = "bcabc"`. We want to remove duplicates and return the smallest lexicographical string using the approach described.

1. We start by constructing the `last` dictionary to record the last occurrence of each character:

```
last = {'b': 3, 'c': 4, 'a': 2}
```

2. We initialize the `stk` list and `vis` set:

```
stk = []
vis = set()
```

3. We iterate over the characters in the string "bcabc":
   - For the first character `'b'`, it's not in `vis`, so we add `'b'` to `stk` and `vis`.

```
stk = ['b']
vis = {'b'}
```

   - Next is `'c'`, also not in `vis`, so we add `'c'` to `stk` and `vis`.

```
stk = ['b', 'c']
vis = {'b', 'c'}
```

   - Then, we encounter `'a'`. It's not in `vis`, but before we add it to `stk`, we check if we can pop any characters that are lexicographically larger and occur later. `'c'` and `'b'` are both larger, but `'c'` occurs later (`last['c'] > i`), so we pop `'c'` from `stk` and remove it from `vis`. `'b'` also occurs later, so we pop `'b'` as well.

```
stk = []
vis = {}
```

   Now we add `'a'`.

```
stk = ['a']
vis = {'a'}
```

   - For the next `'b'`, it is not in `vis`, and 'a' in stack is smaller than 'b', so `'b'` is simply added to `stk` and `vis`.

```
stk = ['a', 'b']
vis = {'a', 'b'}
```

   - Finally, we have another `'c'`. It isn't in `vis`, and `'b'` in the stack is smaller than `'c'`, so again we add `'c'` to both `stk` and `vis`.

```
stk = ['a', 'b', 'c']
vis = {'a', 'b', 'c'}
```

4. The iteration is over and we join the stack to get the result:

```
return ''.join(stk)  # 'abc'
```

The final output is `'abc'`, which contains no duplicate letters and is the smallest possible string in lexicographical order made from the letters of the original string.

Python Solution

```python
class Solution:
    def removeDuplicateLetters(self, s: str) -> str:
        # Create a dictionary to store the last occurrence of each character
        last_occurrence = {char: index for index, char in enumerate(s)}

        # Initialize an empty stack to keep track of the characters in result
        stack = []

        # Set to keep track of characters already in the stack to avoid duplicates
        visited = set()

        # Iterate over each character and its index in the string
        for index, char in enumerate(s):
            # Skip if the character is already in the visited set
            if char in visited:
                continue

            # Ensure the top element of the stack is greater than the current character
            # and the top element occurs later in the string as well
            while stack and stack[-1] > char and last_occurrence[stack[-1]] > index:
                # The top element can be removed and thus it is no longer visited.
                visited.remove(stack.pop())

            # Add the current character to the stack and mark it as visited
            stack.append(char)
            visited.add(char)

        # Convert the stack to a string by joining the characters
        return ''.join(stack)

```

Java Solution

```java
class Solution {
    public String removeDuplicateLetters(String s) {
        int stringLength = s.length();
        int[] lastIndex = new int[26]; // to store the last index of each character

        // Fill array with the last position of each character in the string
        for (int i = 0; i < stringLength; ++i) {
            lastIndex[s.charAt(i) - 'a'] = i;
        }

        Deque<Character> stack = new ArrayDeque<>(); // stack to hold the characters for result
        int bitmap = 0; // to keep track of which characters are already in stack

        // Iterate through the string characters
        for (int i = 0; i < stringLength; ++i) {
            char currentChar = s.charAt(i);
            // Check if the current character is already in stack (bit is set)
            if (((bitmap >> (currentChar - 'a')) & 1) == 1) {
                continue; // Skip if character is already present
            }

            // Ensure characters in stack are in the correct order and remove any that aren't
            while (!stack.isEmpty() && stack.peek() > currentChar && lastIndex[stack.peek() - 'a'] > i) {
                bitmap ^= 1 << (stack.pop() - 'a'); // Set the bit to 0 for popped character
            }

            stack.push(currentChar); // Add current character to the stack
            bitmap |= 1 << (currentChar - 'a'); // Set the bit to 1 for current character
        }

        StringBuilder resultBuilder = new StringBuilder();
        // Build the result string from the characters in stack
        for (char c : stack) {
            resultBuilder.append(c);
        }

        // The order of characters in stack is reversed so we need to reverse the string
        return resultBuilder.reverse().toString();
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to remove duplicate letters and return the smallest in lexicographical order string that contains every letter of s e
    string removeDuplicateLetters(string s) {
        int n = s.size();

        // Initialize an array to store the index of the last occurrence of each character.
        int lastIndex[26] = {0};
        for (int i = 0; i < n; ++i) {
            lastIndex[s[i] - 'a'] = i;
        }

        // String to store the answer.
        string answer;

        // This will serve as a bitmask to keep track of which characters have been added to the answer.
        int usedMask = 0;

        // Iterate through the characters of the string.
        for (int i = 0; i < n; ++i) {
            char currentChar = s[i];

            // If the current character has already been used, skip it.
            if ((usedMask >> (currentChar - 'a')) & 1) {
                continue;
            }

            // While the answer is not empty, the last character in the answer is greater than the current character,
            // and the last occurrence of the last character in the answer is after the current position,
            // we remove the last character from the answer and update the mask.
            while (!answer.empty() && answer.back() > currentChar && lastIndex[answer.back() - 'a'] > i) {
                // Remove the last character from the used mask.
                usedMask ^= 1 << (answer.back() - 'a');
                // Remove the last character from the answer.
                answer.pop_back();
            }

            // Append the current character to the answer.
            answer.push_back(currentChar);
            // Mark the current character as used in the mask.
            usedMask |= 1 << (currentChar - 'a');
        }

        return answer;
    }
};

```

Typescript Solution

```typescript
// Function to remove duplicate letters and return the smallest
// lexicographical order string that contains every letter of s exactly once.
function removeDuplicateLetters(s: string): string {
    const n: number = s.length;

    // Initialize an array to store the index of the last occurrence of each character.
    const lastIndex: number[] = new Array(26).fill(0);
    for (let i = 0; i < n; ++i) {
        lastIndex[s.charCodeAt(i) - 'a'.charCodeAt(0)] = i;
    }

    // String to store the answer.
    let answer: string = '';

    // This will serve as a bitmask to keep track of which characters have been added to the answer.
    let usedMask: number = 0;

    // Iterate through the characters of the string.
    for (let i = 0; i < n; ++i) {
        const currentChar: string = s[i];

        // If the current character has already been used, skip it.
        if ((usedMask >> (currentChar.charCodeAt(0) - 'a'.charCodeAt(0))) & 1) {
            continue;
        }

        // While the answer is not empty, the last character in the answer is greater than the current character,
        // and the last occurrence of the last character in the answer is after the current position,
        // we remove the last character from the answer and update the mask.
        while (answer && answer[answer.length - 1] > currentChar &&
               lastIndex[answer.charCodeAt(answer.length - 1) - 'a'.charCodeAt(0)] > i) {
            // Remove the last character from the used mask.
            usedMask ^= 1 << (answer.charCodeAt(answer.length - 1) - 'a'.charCodeAt(0));
            // Remove the last character from the answer.
            answer = answer.slice(0, -1); // String slice operation to remove last character
        }

        // Append the current character to the answer.
        answer += currentChar;
        // Mark the current character as used in the mask.
        usedMask |= 1 << (currentChar.charCodeAt(0) - 'a'.charCodeAt(0));
    }

    return answer;
}

```

Time and Space Complexity

Time Complexity

The main loop in the given code iterates through each character of the input string `s`. Operations inside this loop include checking if a character is in the visited set, which is an `O(1)` operation, and comparing characters which is also `O(1)`. Additionally, the while loop may result in popping elements from the stack. In the worst case, this popping can happen `n` times throughout the entire for loop, but each element is popped only once. So, the amortized time complexity for popping is O(1). These operations inside the loop do not change the overall linear time traversal of the string. Hence, the time complexity of the code is `O(n)`, where `n` is the length of the string.

Space Complexity

We use a stack `stk`, a set `vis`, and a dictionary `last` to store the characters. The size of the stack and the set will at most grow to the size of the unique characters in `s`, which is limited by the number of distinct characters in the alphabet (at most 26 for English letters). Hence, we can consider this to be `O(1)` space. The dictionary `last` contains a key for every unique character in the string, so this is also `O(1)` under the assumption of a fixed character set. Therefore, the overall space complexity is `O(1)` because the space required does not grow with the size of the input string, but is rather constant due to the fixed character set.