1209. Remove All Adjacent Duplicates in String II

Medium Stack String

Problem Description

The problem provides us with a string `s` and an integer `k`. The task is to repetitively remove `k` adjacent, identical characters from the string until no further such group of `k` identical characters exists. After each removal, the left and right parts of the string are concatenated together. This process of removal is known as a `k` duplicate removal. The final string obtained after performing as many `k` duplicate removals as possible is the output. It is assured that the outcome obtained is unique for the given input string and integer `k`.

Intuition

The solution employs a stack data structure to track groups of identical characters. The stack helps us efficiently manage the groups and their counts without repeatedly scanning the whole string, which would be less efficient.

Here's the step-by-step intuition behind the solution:

1. Iterate over each character in the input string.
2. For each character `c`, check if the top element of the stack is the same character. If it is, we increment the count associated with that character at the top of the stack.
3. If the count of the top element becomes equal to `k` after incrementing, it means we have found `k` adjacent identical characters, and we can remove them from the stack (mimicking the removal from the string).
4. If the stack's top element is not `c`, or if the stack is empty, we push the new character `c` onto the stack with an initial count of one, as it starts a new potential group.
5. After the iteration is complete, we are left with a stack that contains characters and their respective counts that are less than `k`. The final answer is constructed by repeating the characters in the stack according to their remaining counts.
6. We return the reconstructed string as the result.

This method leverages the stack's Last-In-First-Out (LIFO) property, allowing us to process characters in a way that mimics the described `k` duplicate removal process. In essence, we maintain a running "active" portion of the string, with counts reflecting how many consecutive occurrences of each character there are. When any count reaches `k`, we've identified a 'completed' group which we can remove, emulating the removal of `k` duplicates in the string itself.

Solution Approach

The implementation uses a stack which is one of the most suited data structures for problems involving successive pairs or groups of elements with some common characteristics, like consecutive duplicate characters in this case.

Here is how the code works:

1. **Initializing the Stack:** We begin by initializing an empty list `stk` which will serve as our stack. The stack will store sublists where each sublist contains a character and a count of how many times it has occurred consecutively.
2. **Iterating Over the String:** We iterate over each character `c` in the string `s`:
   - **Top of the Stack Comparison:** We look at the top element of the stack (if it isn't empty) to see if `c` is the same as the character at the top of the stack (`stk[-1][0]` represents the character at the top of the stack).
   - **Incrementing Count:** If the top element is the same as `c`, we increment the count (`stk[-1][1]`) and check if this count has reached `k`. If the count is `k`, this signifies a `k` duplicate removal, and therefore we pop the top of the stack.
   - **Pushing New Characters:** If the stack is empty or `c` is different from the top character of the stack, we push a new sublist onto the stack with `c` and the count `1`.
3. **Reconstructing the String:** After the iteration, the stack contains characters that did not qualify for `k` duplicate removal, with their counts being less than `k`. Now, to reconstruct the final string, we multiply (`*`) each character `c` by its count `v` and join (`"".join()`) the multiplied strings to form the answer.

The mathematical formula for the reconstruction process is given by:

```
ans = "".join([c * v for c, v in stk])
```

Here is the breakdown using the code:

- `if stk and stk[-1][0] == c:` This checks if the stack is not empty and whether the top element on the stack is the same as the current character.
- `stk[-1][1] = (stk[-1][1] + 1) % k:` Here, we increment the count and use the modulus operator to reset it to `0` if it reaches `k` (since `x % k is 0`).
- `if stk[-1][1] == 0: stk.pop()` If after incrementing, the count becomes `0`, meaning we have `k` duplicates, we remove (pop) the top element from the stack.
- `else: stk.append([c, 1])` In case the character `c` is different or the stack is empty, we append `c` along with the initial count `1` as a new group in the stack.
- `ans = [c * v for c, v in stk]` We build the answer list by multiplying the character `c` by its count `v`.
- `return "".join(ans)` Join all strings in the `ans` list to get the final answer.

The code takes advantage of Python's built-in list operations to simulate stack behavior efficiently, making the solution succinct and highly readable.

Example Walkthrough

Let's take a small example and walk through the solution approach to illustrate how it works. Assume `s = "aabbbacc"` and `k = 3`. Here's how the algorithm would process this:

1. **Initializing the Stack:** We start with an empty stack `stk`.
2. **Iterating Over the String:**
   - We begin to iterate over `s`. First character is `'a'`. The stack is empty, so we push `['a', 1]`.
   - Next character is `'a'`. The top of the stack is `'a'`, we increment the count: `['a', 2]`.
   - We move to the next character `'b'`. Stack's top is `'a'`, which is different, so we push `['b', 1]`.
   - Next is another `'b'`. The top is `'b'` with count 1. Increment count: `['b', 2]`.
   - Another `'b'` comes. Increment count: `['b', 3]`. But now we've hit `k` duplicates, so we remove this from the stack.
   - The next character is `'a'`. Top of the stack is `'a'` with count 2. Increment the count and now we have `'a'` with count 3, which meets the removal condition, and thus it is removed from the stack as well.
   - We then see `'c'`. The stack is empty now, so we push `['c', 1]`.
   - Another `'c'` comes up. Top is `'c'` with count 1. Increment count: `['c', 2]`.

   At the end of this process, our stack `stk` is `[ ['c', 2] ]` since we only have two `'c'` characters, which is less than the `k` duplicate threshold.
3. **Reconstructing the String:**
   - We can't remove any more elements, so it's time to reconstruct the string from the stack items. We multiply each character by its count: `'c' * 2` which yields `'cc'`.

Hence, the final string after performing `k` duplicate removals is `"cc"`.

Here is the resultant stack operations visualized at each step:

Input	Stack	Operation
`'a'`	`['a', 1]`	Push `'a'`
`'a'`	`['a', 2]`	Increment count of `'a'`
`'b'`	`['a', 2], ['b', 1]`	Push `'b'`
`'b'`	`['a', 2], ['b', 2]`	Increment count of `'b'`
`'b'`	`['a', 2]`	Increment count of `'b'` and remove `'b'` as count=k
`'a'`	`[]`	Increment count of `'a'` and remove `'a'` as count=k
`'c'`	`['c', 1]`	Push `'c'`
`'c'`	`['c', 2]`	Increment count of `'c'`

And the final string obtained by concatenating characters based on their count in the stack is `'cc'`.

Python Solution

```python
class Solution:
    def removeDuplicates(self, s: str, k: int) -> str:
        # Initialize an empty list to use as a stack.
        character_stack = []

        # Iterate over each character in the input string.
        for character in s:
            # If the stack is not empty and the top element of the stack has the same character,
            # increase the count of that character in the stack.
            if character_stack and character_stack[-1][0] == character:
                character_stack[-1][1] += 1
                # If the count reaches k, remove (pop) it from the stack.
                if character_stack[-1][1] == k:
                    character_stack.pop()
            else:
                # If the character is not at the top of the stack (or if the stack is empty), add it with a count of 1.
                character_stack.append([character, 1])

        # Reconstruct the string without duplicates by multiplying the character by its count.
        # This joins all the tuples in the stack, which holds the count of each character (not removed).
        result = ''.join(character * count for character, count in character_stack)
        return result
```

Java Solution

```java
import java.util.Deque;
import java.util.ArrayDeque;

class Solution {
    public String removeDuplicates(String s, int k) {
        // Initialize a stack to keep track of characters and their counts.
        Deque<int[]> stack = new ArrayDeque<>();

        // Loop through each character of the string.
        for (int i = 0; i < s.length(); ++i) {
            // Convert the character to an index (0 for 'a', 1 for 'b', etc.).
            int index = s.charAt(i) - 'a';

            // If stack is not empty and the character on the top of the stack is the same as the current one,
            // increment its count. Otherwise, push a new count of 1 onto the stack.
            if (!stack.isEmpty() && stack.peek()[0] == index) {
                // Increment the count and use modulo operation to reset to 0 if it hits the 'k'.
                stack.peek()[1] = (stack.peek()[1] + 1) % k;

                // If the count becomes 0 after reaching k, pop the element from the stack.
                if (stack.peek()[1] == 0) {
                    stack.pop();
                }
            } else {
                // If stack is empty or the top element is different, push the new character and count (1).
                stack.push(new int[] {index, 1});
            }
        }

        // Initialize a StringBuilder to collect the result.
        StringBuilder result = new StringBuilder();

        // Build the result string by iterating over the stack in LIFO order.
        for (int[] element : stack) {
            // Retrieve the character from the integer index.
            char c = (char) (element[0] + 'a');
            // Append the character element[1] (count) times.
            for (int i = 0; i < element[1]; ++i) {
                result.append(c);
            }
        }

        // The characters were added in reverse order, so reverse the whole string to get the correct order.
        result.reverse();

        // Return the resultant string.
        return result.toString();
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    // Function to remove duplicates from a string where a sequence of
    // 'k' consecutive duplicate characters should be removed.
    string removeDuplicates(string s, int k) {
        // A stack to keep track of characters and their counts
        vector<pair<char, int>> charStack;

        // Traverse the given string
        for (char currentChar : s) {
            // Check if the stack is not empty and the top element character
            // is the same as the current character
            if (charStack.empty() && charStack.back().first == currentChar) {
                // Increase the count of the current character at the top of the stack and take modulo 'k'
                charStack.back().second = (charStack.back().second + 1) % k;

                // If the count becomes 0, it means 'k' consecutive characters are accumulated; pop them
                if (charStack.back().second == 0) {
                    charStack.pop_back();
                }
            } else {
                // Otherwise, push the current character with count 1 onto the stack
                charStack.push_back({currentChar, 1});
            }
        }

        // Prepare the result string
        string result;

        // Build the string by repeating each character in the stack by its count
        for (auto& [character, count] : charStack) {
            result += string(count, character);
        }

        // Return the final result string
        return result;
    }
};
```

Typescript Solution

```typescript
// Function to remove duplicates from a string where a sequence of
// 'k' consecutive duplicate characters should be removed.
function removeDuplicates(s: string, k: number): string {
    // A stack to keep track of characters and their counts
    let charStack: Array<{ character: string; count: number }> = [];

    // Traverse the given string
    for (let currentChar of s) {
        // Check if the stack is not empty and the top element character
        // is the same as the current character
        if (charStack.length > 0 && charStack[charStack.length - 1].character === currentChar) {
            // Increase the count of the current character at the top of the stack by 1
            charStack[charStack.length - 1].count = (charStack[charStack.length - 1].count + 1) % k;

            // If the count reaches 'k', it means 'k' consecutive characters are accumulated; pop them
            if (charStack[charStack.length - 1].count === 0) {
                charStack.pop();
            }
        } else {
            // Otherwise, push the current character with count 1 onto the stack
            charStack.push({ character: currentChar, count: 1 });
        }
    }

    // Prepare the result string
    let result: string = '';

    // Build the string by repeating each character in the stack by its count
    for (let { character, count } of charStack) {
        result += character.repeat(count);
    }

    // Return the final result string
    return result;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the code can be analyzed by looking at the operations within the main loop that iterates over each character in the input string `s`. Here's the breakdown:

- For each character `c` in `s`, the code performs a constant-time check to see if the stack `stk` is not empty and if the top element's character matches `c`.
- If there's a match, it updates the count of that character on the stack, which is also a constant-time operation.
- If the count equals `k`, it pops the element from the stack. Popping from the stack takes amortized constant time.
- If there is no match or the stack is empty, it pushes the current character with count 1 onto the stack. This is a constant-time operation.

Since each character in the string is processed once and each operation is constant time or amortized constant time, the overall time complexity is $O(n)$, where n is the length of string `s`.

Space Complexity

The space complexity of the code is determined by the additional space used by the stack `stk`:

- In the worst case, if there are no `k` consecutive characters that are the same, the stack will contain all distinct characters of `s`, which would take $O(n)$ space.
- In the best case, if all characters are removed as duplicates, the stack will be empty, so no extra space is used beyond the input string.

Thus, the space complexity of the algorithm is $O(n)$, where n is the length of string `s`, representing the maximum stack size that may be needed in the worst case.