1047. Remove All Adjacent Duplicates In String

Easy Stack String

Problem Description

In this problem, you're presented with a string `s` that contains only lowercase English letters. Your task is to repetitively remove pairs of adjacent, identical characters until no such pairs remain. The process is to look for two adjacent letters that are the same, and then remove them from the string. This is done over and over until it's no longer possible to do so. The result is the final, reduced string with no adjacent pairs of the same character. The uniqueness of the answer is guaranteed, which means no matter how you remove the pairs, the final string will be the same.

Intuition

The idea behind solving this problem involves simulating the described process using a data structure that allows for efficient addition and removal of elements, mainly when these operations are performed at one end. A stack is perfect for this case because it allows us to add elements to the top and also remove from the top, which can be thought of as the end of the string we are building.

As you iterate through the string, you examine each character. If the current character is the same as the one at the top of the stack (meaning they are adjacent duplicates in the context of the string), you remove the character from the stack (simulating the removal of the adjacent duplicate). If the current character is not a duplicate, or if the stack is empty, you push the current character onto the stack.

The process is akin to folding paper; whenever you spot a foldable (matchable) pair, you fold (remove) it. By the end of the string, the stack will contain all the characters that couldn't be matched and removed. Finally, you convert the stack to a string and return it, giving you the reduced string without any adjacent duplicates.

Solution Approach

The solution to this problem utilizes a commonly used data structure known as a stack. Here's a step-by-step approach to how the algorithm works with the stack to remove duplicates:

1. Initialize an empty stack, `stk`. This will be used to keep track of characters in the string as they are processed.
2. Iterate through each character `c` in the input string `s`.
3. For each character, check if the stack is not empty and if the character at the top of the stack (the last character added to the stack) is equal to the current character `c`.
   - If yes, that means we have found two adjacent, equal characters. The character at the top of the stack is removed using the `pop()` operation, which effectively removes the pair of adjacent duplicates from our consideration.
   - If no, there is no pair to remove. In this case, the current character `c` is added to the top of the stack with the `append()` method.
4. Repeat this process until all characters from the input string have been examined.

After the iteration is complete:

5. The stack now contains the final string with all possible pairs removed. Since a stack is a LIFO (last-in, first-out) structure, to form the final string from the leftover characters, we convert the stack to a string by joining the elements in the stack.

We use `join(stk)` to create a string from the elements in the stack.

The implementation in Python using the described approach is efficient because it only processes each character once and uses `O(n)` space, where `n` is the length of the string (in the worst case when there are no duplicates).

This algorithm is an example of a greedy approach, where we make the local optimal choice at each step (removing adjacent duplicates whenever possible) that leads to the global optimal solution — the fully reduced string.

Here is the Python code for the reference:

```
class Solution:
    def removeDuplicates(self, s: str) -> str:
        # Initialize an empty {stack}{/problems/stack_intro}
        stk = []
        # Iterate through each character in the string
        for c in s:
            # Check if the stack is not empty and the top element is equal to the current character
            if stk and stk[-1] == c:
                # Remove the last element from the stack i.e., remove duplicate pair
                stk.pop()
            else:
                # Add the current character to the stack
                stk.append(c)
        # Convert the stack to a string and return
        return ''.join(stk)
```

This solution has a time complexity of `O(n)` because each character in the string is processed once. The space complexity is also `O(n)` to account for the stack that holds the characters in the process of removing duplicates.

Example Walkthrough

Let's use a small example to illustrate the solution approach. Consider the string `s = "abbaca"`. We will go through this string and apply the algorithm step by step:

1. Initialize an empty stack `stk`. Currently, `stk = []`.
2. Iterate through each character in the string `s`.
3. The first character is `a`. The stack is empty, so we add `a` to the stack. Now, `stk = ['a']`.
4. The second character is `b`. It isn't equal to the top of the stack, which is `a`, so add `b` to the stack. Now, `stk = ['a', 'b']`.
5. The third character is `b`. It is equal to the top of the stack. Remove the top of the stack, which is the last `b`. Now, `stk = ['a']`.
6. The fourth character is `a`. It equals the top of the stack, so remove `a` from the stack. Now, `stk = []`.
7. The fifth character is `c`. The stack is empty, so add `c` to the stack. Now, `stk = ['c']`.
8. The last character is `a`. It isn't equal to the top of the stack, so add `a` to the stack. The stack now looks like this: `stk = ['c', 'a']`.

After the iteration is complete, the stack contains `['c', 'a']`. Join these to form the final string, which is `"ca"`. This string is the reduced form of the original string `s`, with all adjacent duplicates removed. The process and the resulting string are confirmed to be correct as per our algorithm.

The above steps embody the described algorithm to remove pairs of adjacent identical characters from a string and achieve the desired outcome.

Solution Implementation

Python

```
class Solution:
    def removeDuplicates(self, s: str) -> str:
        # Initialize an empty stack to hold characters.
        stack = []

        # Loop through each character in the input string.
        for char in s:
            # If the stack is not empty and the top element of the stack
            # is the same as the current character, then we have found
            # a duplicate and need to remove it from the stack.
            if stack and stack[-1] == char:
                stack.pop()
            else:
                # If no duplicate found, push the current character onto the stack.
                stack.append(char)

        # Join all characters left in the stack to form the processed string
        # without consecutive duplicate characters.
        return ''.join(stack)
```

Java

```
class Solution {

    // Method to remove all adjacent duplicates from the string s.
    public String removeDuplicates(String s) {
        // Initialize a StringBuilder to construct the result without duplicates.
        StringBuilder resultBuilder = new StringBuilder();

        // Iterate over each character of the string.
        for (char character : s.toCharArray()) {
            int currentLength = resultBuilder.length();
            // If the result has characters and the last character is the same as the current one,
            // we should remove the last character to eliminate the pair of duplicates.
            if (currentLength > 0 && resultBuilder.charAt(currentLength - 1) == character) {
                // Delete the last character from resultBuilder.
                resultBuilder.deleteCharAt(currentLength - 1);
            } else {
                // Otherwise, append the current character to resultBuilder.
                resultBuilder.append(character);
            }
        }

        // Convert the StringBuilder back to a String and return the result.
        return resultBuilder.toString();
    }
}
```

C++

```
#include <string> // Include string library for string usage

class Solution {
public:
    // Method to remove all adjacent duplicates in the string s
    std::string removeDuplicates(std::string s) {
        std::string result; // Create an empty string to use as a stack

        // Iterate through each character in the input string
        for (char character : s) {
            // If result is not empty and the last character is the same as the current one,
            // remove the last character from result (simulating a stack pop operation)
            if (!result.empty() && result.back() == character) {
                result.pop_back();
            } else {
                // Otherwise, add the current character to result (simulating a stack push operation)
                result.push_back(character);
            }
        }

        return result; // Return the resulting string after all duplicates are removed
    }
};
```

TypeScript

```
/**
 * Removes all instances of adjacent duplicates in the given string.
 * @param {string} inputString - The string from which to remove duplicates.
 * @return {string} The modified string with all adjacent duplicates removed.
 */
const removeDuplicates = (inputString: string): string => {
    // Initialize an empty array to use as a stack.
    const stack: string[] = [];

    // Iterate through each character of the input string.
    for (const character of inputString) {
        // If the stack is not empty and the top element matches the current character...
        if (stack.length > 0 && stack[stack.length - 1] === character) {
            // Pop the top element from the stack to remove the duplicate.
            stack.pop();
        } else {
            // If no duplicate is found, push the current character onto the stack.
            stack.push(character);
        }
    }

    // Combine the elements in the stack to form the modified string and return it.
    return stack.join('');
};


class Solution:
    def removeDuplicates(self, s: str) -> str:
        # Initialize an empty stack to hold characters.
        stack = []

        # Loop through each character in the input string.
        for char in s:
            # If the stack is not empty and the top element of the stack
            # is the same as the current character, then we have found
            # a duplicate and need to remove it from the stack.
            if stack and stack[-1] == char:
                stack.pop()
            else:
                # If no duplicate found, push the current character onto the stack.
                stack.append(char)

        # Join all characters left in the stack to form the processed string
        # without consecutive duplicate characters.
        return ''.join(stack)
```

Time and Space Complexity

The given Python code implements a function that removes all adjacent duplicates in a string by using a stack. It iterates through each character in the input string, and removes characters from the stack when a duplicate is detected or appends characters otherwise. Let's analyze the time and space complexities of the code.

Time Complexity

The time complexity is determined by the number of iterations the algorithm performs and the operations executed per iteration. For this code:

- The algorithm iterates through each character of the input string once.
- For each character, the algorithm performs a constant time operation to check if the stack is non-empty and if the top of the stack is equal to the current character.
- In the worst case (when all characters are unique), each character gets added to the stack. In the best case (when all characters are duplicates), each character gets compared and none is added to the stack.
- The `pop` and `append` operations on the stack take constant time, i.e., `O(1)`.

Therefore, the time complexity of the algorithm is `O(n)`, where `n` is the length of the input string.

Space Complexity

The space complexity considers the additional space used by the algorithm as a function of the input size:

- The stack `stk` is the primary data structure used to store characters from the input string temporarily.
- In the worst case scenario, where there are no consecutive duplicate characters, the stack will contain all `n` characters of the input string.
- In the best case, where all characters are consecutive duplicates, the stack will be empty at the end of the iteration.

Hence, the space complexity of the algorithm is also `O(n)`.

In summary, the function `removeDuplicates` has a time complexity of `O(n)` and a space complexity of `O(n)`.