2390. Removing Stars From a String

Medium Stack String Simulation

Problem Description

In this problem, you are given a string `s` which consists of regular characters mixed with star characters (`*`). Your task is to modify the string by repeatedly performing a specific operation until all star characters are removed.

The operation allowed is:

- Select a star character in the string `s`.
- Remove the character that is immediately to the left of the selected star (this will always be a non-star character due to the provided input guarantees).
- Remove the star itself.

You must continue performing this operation until there are no more stars in the string. The problem guarantees that it is always possible to perform the operation, meaning you will never run out of non-star characters on the left side of a star to remove. The problem also assures you that the final string will be unique, so there is only one correct solution.

The goal is to return the resulting string after all the stars and their adjacent left characters have been removed.

Intuition

The intuition behind the solution is to process the given string `s` from left to right, keeping track of the characters that would remain after each star operation. To implement this approach efficiently, we can use a stack data structure, which provides an easy way to remove the last character added when we encounter a star.

Here's how the solution approach is derived:

- Initialize an empty list `ans` which will function as a stack to hold the characters that would remain in the string.
- Iterate over each character `c` in the string `s`.
  - If the current character `c` is a star (`*`), it indicates that we need to remove the character nearest to its left. In stack terms, this means we need to pop the last element from the stack.
  - If the current character `c` is not a star, we should add this character to the stack, as it is (for now) a part of the resulting string.
- After we finish iterating through the string `s`, the stack contains all the characters that survived the star operations.
- We then join the characters in the stack to form the final string, which is the string after all stars and their adjacent left characters have been removed.
- Return the final string as a result.

This approach simulates the given operations by keeping a dynamic list of surviving characters, which allows us to efficiently apply the given rules and arrive at the correct result.

Solution Approach

The solution to the problem utilizes a stack-based approach to process the input string. A stack is a data structure that allows adding (pushing) elements to the end and removing (popping) the last element added.

Here's how we implement the solution step-by-step:

1. Initialize an empty list `ans`, which we will use as a stack to keep track of the surviving characters in the string `s`.

   ```
   ans = []
   ```

2. Iterate over each character `c` in the string `s`.

   ```
   for c in s:
   ```

3. For each character `c` encountered, check whether it is a star (`*`) or not.

   ```
   if c == '*':
   ```

   - If `c` is a star, we need to remove the most recent character added to the `ans` list, which represents the character to the left of the star in the original string. Since `ans` acts as our stack, we can simply pop the last element from it:

     ```
     ans.pop()
     ```

   - If `c` is not a star, this character is potentially part of the final string and should be added (pushed) to the `ans` list.

     ```
     else:
         ans.append(c)
     ```

   The key algorithmic insight here is that the stack automatically takes care of the 'closest non-star character to its left' detail mentioned in the problem description. The last element on the stack will always be the closest character to any star that we encounter.

4. After the loop has finished, all stars have been processed, and `ans` contains only the characters that are part of the final string. We need to reconstruct the string from the list.

   ```
   return ''.join(ans)
   ```

   Using the `join` method on an empty string and passing `ans` as an argument concatenates all the elements in `ans` into a single string without any delimiters between them (since we're joining with an empty string).

5. Return the final string as the result of the `removeStars` method.

By following these steps and applying a simple stack-based algorithm, we are able to simulate the character removal process specified in the problem statement and arrive at the correct solution in a clear and efficient manner.

Example Walkthrough

Let's use a simple example to illustrate the solution approach. Consider the string `s = "ab*c*"`; our task is to remove the star characters (`*`) along with the immediate left non-star character until no stars are left in the string.

1. Initialize the stack `ans` as an empty list.

   ```
   ans = []
   ```

2. Iterate over each character in `s`, which is `"ab*c*"`. We will process the characters one by one:

   - When we see `'a'`, since it is not a star, we add it to `ans`.

     ```
     ans = ['a']
     ```

   - When we see `'b'`, again it is not a star, so we add it to `ans`.

     ```
     ans = ['a', 'b']
     ```

   - Now we encounter `'*'`. It's a star, so we need to remove the most recent non-star character, which is `'b'`. We pop `'b'` from `ans`.

     ```
     ans = ['a']
     ```

   - Next, we see `'c'`, which is not a star. So we add `'c'` to `ans`.

     ```
     ans = ['a', 'c']
     ```

   - Finally, we encounter another star `'*'`. We remove the latest non-star character `'c'` by popping it from `ans`.

     ```
     ans = ['a']
     ```

3. With all the characters processed, we now have the final `ans` stack:

   ```
   ans = ['a']
   ```

   This represents the characters that survived after all the star operations.

4. Convert the `ans` stack back into a string:

   ```
   return ''.join(ans)
   ```

   This results in the string `"a"`.

5. Return the final string `"a"`, which is the string after all stars and their adjacent left characters have been removed.

Using this example, we can see how the stack-based approach efficiently simulates the operation of removing a star along with its immediately left non-star character. Since we keep adding non-star characters to the stack and only pop when we see a star, the stack maintains a correct representation of the characters to the left that have not been removed by any star operation.

Solution Implementation

Python

```python
class Solution:
    def removeStars(self, s: str) -> str:
        # Initialize an empty list to act as a stack
        result_stack = []

        # Iterate over each character in the string
        for char in s:
            # If the character is a star, we pop the last element from the stack
            # This simulates removing the character before a star
            if char == '*':
                result_stack.pop()
            # If the character is not a star, we add the character to the stack
            else:
                result_stack.append(char)

        # Join the characters in the stack to get the final string
        # This represents the string after all the removals
        return ''.join(result_stack)
```

Java

```java
class Solution {

    // Method to remove characters from a string that are followed by a star.
    // Each star (*) character means a backspace operation on the current string.
    public String removeStars(String s) {
        // Initialize a StringBuilder to hold the resulting characters after backspaces are applied.
        StringBuilder resultBuilder = new StringBuilder();

        // Iterate over each character in the string.
        for (int i = 0; i < s.length(); ++i) {
            // Check if the current character is a star
            if (s.charAt(i) == '*') {
                // If star, remove the preceding character from the StringBuilder if it exists
                if (resultBuilder.length() > 0) {
                    resultBuilder.deleteCharAt(resultBuilder.length() - 1);
                }
            } else {
                // If not a star, append the character to the StringBuilder
                resultBuilder.append(s.charAt(i));
            }
        }
        // Convert the StringBuilder back to a String and return it.
        return resultBuilder.toString();
    }
}
```

C++

```cpp
class Solution {
public:
    // Function to remove characters followed by a star (*) character in the string
    string removeStars(string s) {
        // The result string that will store the final output after removal
        string result;

        // Iterate through each character in the input string
        for (char c : s) {
            if (c == '*') {
                // If current character is a star, remove the last character from result
                if (!result.empty()) { // Ensure there is a character to remove
                    result.pop_back();
                }
            } else {
                // Otherwise, append the current character to the result string
                result.push_back(c);
            }
        }

        // Return the modified string after all stars and their preceding characters are removed
        return result;
    }
};
```

TypeScript

```typescript
// This function removes all characters that are followed by a '*' in a given string.
// @param {string} s - The input string containing characters and '*'
// @return {string} - The modified string with characters followed by '*' removed
function removeStars(s: string): string {
    // Initialize an array to act as a stack to manage the characters
    const result: string[] = [];

    // Loop through each character in the input string
    for (const char of s) {
        if (char === '*') {
            // If the current character is '*', remove the last character from the stack
            result.pop();
        } else {
            // Otherwise, add the current character to the stack
            result.push(char);
        }
    }

    // Join the characters in the stack to form the final string and return it
    return result.join('');
}
```

```python
class Solution:
    def removeStars(self, s: str) -> str:
        # Initialize an empty list to act as a stack
        result_stack = []

        # Iterate over each character in the string
        for char in s:
            # If the character is a star, we pop the last element from the stack
            # This simulates removing the character before a star
            if char == '*':
                result_stack.pop()
            # If the character is not a star, we add the character to the stack
            else:
                result_stack.append(char)

        # Join the characters in the stack to get the final string
        # This represents the string after all the removals
        return ''.join(result_stack)
```

Time and Space Complexity

The given Python code implements a function to process a string by removing characters that should be deleted by a subsequent asterisk (`*`). The algorithm uses a stack represented by a list (`ans`) to manage the characters of the string.

Time Complexity:

The time complexity of the code is $O(n)$, where n is the length of the input string `s`. This is because the code scans the string once, and for each character, it either appends it to the stack or pops the last character from the stack. Both appending to and popping from the end of a list in Python are operations that run in constant time, $O(1)$.

Therefore, the loop iterates n times, with $O(1)$ work for each iteration, resulting in a total time complexity of $O(n)$.

Space Complexity:

The space complexity of the code is also $O(n)$, which is the space needed to store the stack (`ans`) in the worst case where there are no asterisks in the string. In this case, all characters will be appended to the stack.

However, if there are asterisks, each asterisk reduces the size of the stack by one, potentially lowering the space used. The actual space usage depends on the distribution of non-asterisk characters and asterisks in the string, but the worst-case space complexity remains $O(n)$, as we are always limited by the length of the initial string.