

Solution: Remove All Adjacent Duplicates In String

Let's solve the Remove All Adjacent Duplicates In String problem using the Stacks pattern.

We'll cover the following

- Statement
- Solution
 - Naive approach
 - Optimized approach using stacks
 - Solution summary
 - Time complexity
 - Space complexity

Statement

You are given a string consisting of lowercase English letters. Repeatedly remove adjacent duplicate letters, one pair at a time. Both members of a pair of adjacent duplicate letters need to be removed.

Constraints:

- $1 \leq \text{string.length} \leq 10^5$
- `string` consists of lowercase English alphabets.

Solution

So far, you've probably brainstormed some approaches and have an idea of how to solve this problem. Let's explore some of these approaches and figure out which one to follow based on considerations such as time complexity and any implementation constraints.

Naive approach

The naive approach is to generate a set of all possible adjacent duplicates in a string of small English alphabets, that is, `{"aa", "bb", "cc", "dd", ...}`. Traverse over the string to see if such duplicates are present. We'll use nested loops to check this condition. If duplicate characters are found, we'll remove the adjacent duplicates from the input string. For example, in the `"abaac"` string, we'll remove `"aa"` from the string, which results in `"abc"`.

Since we're using nested loops, the time complexity of this approach is $O(n^2)$, which is not optimal. Let's see if we can use the stacks pattern to reduce the time complexity of our solution.

Optimized approach using stacks

We can remove adjacent duplicates in pairs, which is similar to matching opening and closing parentheses. In such problems, we typically use stacks to store intermediate results to deal with nested pairs of parentheses. We'll use the same concept here.

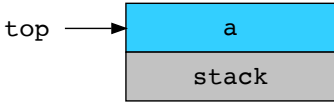
We'll initialize a stack first. Then, we'll iterate the complete string from left to right, and for every character, we'll check the following:

- If the stack is empty, we'll push the character onto the stack.

- If the stack is not empty, we'll perform the following tasks:
 - If the string's character is different from the stack's top character, we'll push the string's character onto the stack because we did not find the adjacent duplicate.
 - If the string's character is the same as the stack's top character, it means that we found the adjacent duplicate characters. We'll pop that character from the stack and move to the next character in the string.
- After the entire string has been traversed, the remaining characters in the stack will be our result. We'll form a string from those characters and return it.

The slides below illustrate how we'd like the algorithm to run:

a	z	b	b	z	y	b	a	a	b
---	---	---	---	---	---	---	---	---	---



The stack is empty. Push the first character 'a' onto the stack.

1 of 11



Let's look at the code for this solution:

Java

```

1  import java.util.*;
2
3  class RemoveAdjacent {
4
5      public static String removeDuplicates(String str) {
6          // Create an empty stack.
7          Stack <Character> stack = new Stack<>();
8
9          // Iterate over the string
10         for (char c: str.toCharArray()) {
11             // If stack has at least one character and
12             // stack's top character is same as the string's character.
13             if (!stack.isEmpty() && stack.peek() == c) {
14                 // Pop a character from the stack.
15                 stack.pop();
16             } else {
17                 // Otherwise, push that character onto the stack.
18                 stack.push(c);
19             }
20         }
21
22         // Create an empty result string.
23         StringBuilder result = new StringBuilder();
24         // Iterate till stack is empty
25         while (!stack.isEmpty()) {
26             // Get top element and concatenate to the result string.
27             result.insert(0, stack.pop());
28         }
29     }
30 }
  
```

▶

📄

↶

⌂

🔄

?

Tt

🔄



To recap, the solution to this problem can be divided as follows:

1. Create an empty stack to store characters.
2. Iterate over the input string and push the character onto the stack if the stack is either empty or the stack's top element is different from the string's character.
3. If they both are the same, pop an element from the stack.
4. After the entire string has been traversed, the remaining characters in the stack represent the input string without adjacent duplicates.

Time complexity

We iterate over all the n characters of the given string exactly once. For each character, we'll perform $O(1)$ operations. Therefore, the time complexity of this solution is $O(n)$.

Space complexity

The space complexity of this solution is $O(n)$ in the worst-case scenario when the stack stores all the characters. This will happen when each character appears exactly once in the input string.

[< Back](#)

Remove All Adjacent ...

[Next >](#)

Minimum Remove to ...



Mark as
Completed

