<https://leetcode.com/problems/remove-all-adjacent-duplicates-in-string/description/>

You are given a string s consisting of lowercase English letters. A duplicate removal consists of choosing two adjacent and equal letters and removing them.

We repeatedly make duplicate removals on s until we no longer can.

Return the final string after all such duplicate removals have been made. It can be proven that the answer is unique.

**Example 1:**

**Input:** s = "abbaca"

**Output:** "ca"

**Explanation:** For example, in "abbaca" we could remove "bb" since the letters are adjacent and equal, and this is the only possible move. The result of this move is that the string is "aaca", of which only "aa" is possible, so the final string is "ca".

**Example 2:**

**Input:** s = "azxxzy"

**Output:** "ay"

**Constraints:**

1 <= s.length <= 10^5

s consists of lowercase English letters.

**Attempt 1: 2024-11-27**

**Solution 1: Stack (10 min)**

**Style 1: Use Stack + StringBuilder**

class Solution {

    public String removeDuplicates(String s) {

        Stack<Character> stack = new Stack<>();

        for(char c : s.toCharArray()) {

            if(!stack.isEmpty() && stack.peek() == c) {

                stack.pop();

            } else {

                stack.push(c);

            }

        }

        StringBuilder sb = new StringBuilder();

        while(!stack.isEmpty()) {

            sb.append(stack.pop());

        }

        return sb.reverse().toString();

    }

}

Time Complexity: O(n)

Space Complexity: O(n)

**Style 2: Use StringBuilder only**

class Solution {

    public String removeDuplicates(String s) {

        StringBuilder sb = new StringBuilder();

        for(char c : s.toCharArray()) {

            int curLen = sb.length();

            if(curLen > 0 && sb.charAt(curLen - 1) == c) {

                sb.deleteCharAt(curLen - 1);

            } else {

                sb.append(c);

            }

        }

        return sb.toString();

    }

}

Time Complexity: O(n)

Space Complexity: O(n)

**Refer to**

<https://algo.monster/liteproblems/1047>

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();

}

}

**Solution 2: Two Pointers (60 min)**

**Style 1: Move back two steps in one time**

class Solution {

    public String removeDuplicates(String s) {

        int len = s.length();

        char[] result = s.toCharArray();

        int i = 0;

        for(int j = 0; j < len; j++, i++) {

            result[i] = result[j];

            if(i > 0 && result[i] == result[i - 1]) {

                i -= 2;

            }

        }

        return new String(result, 0, i);

    }

}

Time Complexity: O(n)

Space Complexity: O(1)

**Refer to**

<https://leetcode.com/problems/remove-all-adjacent-duplicates-in-string/solutions/294893/java-c-python-two-pointers-and-stack-solution/>

i refers to the index to set next character in the output string.

j refers to the index of current iteration in the input string.

Iterate characters of S one by one by increasing j.

If S[j] is same as the current last character S[i - 1],

we remove duplicates by doing i -= 2.

If S[j] is different as the current last character S[i - 1],

we set S[i] = S[j] and increment i++.

public String removeDuplicates(String s) {

int i = 0, n = s.length();

char[] res = s.toCharArray();

for (int j = 0; j < n; ++j, ++i) {

res[i] = res[j];

if (i > 0 && res[i - 1] == res[i]) // count = 2

i -= 2;

}

return new String(res, 0, i);

}

**Style 2: Move back one step once a time with 0 start index**

class Solution {

    public String removeDuplicates(String s) {

        int len = s.length();

        char[] result = s.toCharArray();

        // Write pointer, keep track of the position where

        // the next valid character should be written.

        int i = 0;

        for(int j = 0; j < len; j++) {

            // 'i' for next valid character written index

            // so 'i - 1' for last written index

            if(i > 0 && result[i - 1] == result[j]) {

                // Remove the duplicate by removing the last written character

                i--;

            } else {

                // Write the character at the current read pointer

                result[i] = result[j];

                i++;

            }

        }

        return new String(result, 0, i);

    }

}

Time Complexity: O(n)

Space Complexity: O(1)

**Refer to chatGPT**

**Concept:**

Use a "write" pointer (i) to keep track of the position where the next valid character should be written.

Use a "read" pointer (j) to iterate through the string.

**Steps:**

As you iterate through the string with the j pointer:

If the current character (s[j]) is the same as the character at the "write" position (s[i - 1]), it means there's a duplicate. In this case, decrement the write pointer to remove the duplicate.

Otherwise, write the character at s[j] to position i and increment i.

At the end, the first i characters in the string represent the result.

class Solution {

public String removeDuplicates(String s) {

char[] stack = s.toCharArray(); // Use the array as a stack

int i = 0; // Write pointer (stack top)

for (int j = 0; j < stack.length; j++) {

if (i > 0 && stack[i - 1] == stack[j]) {

// Remove the duplicate by moving the write pointer back

i--;

} else {

// Write the character at the current read pointer

stack[i] = stack[j];

i++;

}

}

// Build the result string from the valid portion of the array

return new String(stack, 0, i);

}

}

**Style 3: Move back one step once a time with negative start index**

class Solution {

    public String removeDuplicates(String s) {

        int len = s.length();

        char[] result = s.toCharArray();

        // Write pointer

        int i = -1;

        for(int j = 0; j < len; j++) {

            if(i < 0 || result[i] != result[j]) {

                // Move forward one step similar as push one more char on stack

                i++;

                // Record non-duplicate character on position i

                result[i] = result[j];

            } else {

                // Move back one step similar as pop out current char on stack

                i--;

            }

        }

        return new String(result, 0, i + 1);

    }

}

Time Complexity: O(n)

Space Complexity: O(1)

**Refer to**

<https://leetcode.com/problems/remove-all-adjacent-duplicates-in-string/discuss/294964/Java-three-easy-iterative-codes-w-brief-explanation-and-analysis.>

**Method 3: two pointers**

If current char is same as the end of non-adjacent-duplicate chars, decrease the counter end by 1;

otherwise, copy the current char to its end.

public String removeDuplicates(String S) {

char[] a = S.toCharArray();

int end = -1;

for (char c : a) {

if (end >= 0 && a[end] == c) {

--end;

}else {

a[++end] = c;

}

}

return String.valueOf(a, 0, end + 1);

}

**Refer to**

[L1209.Remove All Adjacent Duplicates in String II](note://WEBa1d469d052160f5ef3da969712ef4d1f)