My idea:

1. Traverse from start to end of string
2. If character is star => result pop back
3. Else => result push back

World idea:

**Solution**

**Overview**

We are given a string s made up of lower case english letters and stars \*.

We have to perform some operations on the given string such that for each star, we must remove both the closest non-star character to its left and the star itself.

We have to return the formed string after all stars are removed.

**Approach 1: Stack**

**Intuition**

To solve the problem, we must keep track of the most recently seen non-star character while iterating from the beginning to the end of the string. Our task should be easy if we are able to use a data structure or write an algorithm that keeps track of the most recent non-star character, which when removed gives the second most recent non-star character, which when removed gives the third most recent non-star character, and so on.

There are several approaches to solving such a problem, one of which is to use a **stack** data structure.

If you are new to stack data structure, please see our [Leetcode Explore Card](https://leetcode.com/explore/interview/card/leetcodes-interview-crash-course-data-structures-and-algorithms/706/stacks-and-queues/" \t "_blank) for more information on it!

We iterate through the given string s from the start. We check the character at every index i of s. If s[i] is a non-star character, we push it into the stack. Otherwise, if s[i] is a star character, we pop the character from the stack because the character at the top of the stack is the most recently seen non-star character from the characters that have been seen up to this point and are not removed.

When we've finished iterating over the string, the stack will contain the required string in reverse order. We pop all of the characters from the stack to form a string and then return the reverse of that string as our answer.

Here's a visual representation of how the approach works in the first example given in the problem description:

1 / 12

**Algorithm**

1. Initialize a stack of characters st.
2. Iterate over the string s from the start and for each index i of the string:
   * If s[i] == '\*', we perform the pop operation to remove the top character from the stack.
   * Otherwise, we have a non-star character, so we push it into the stack.
3. Create an empty string variable answer.
4. While the stack is not empty:
   * Append the top character of the stack to answer and remove it from the stack.
5. Return the reverse of answer.

**Implementation**

class Solution {

public:

    string removeStars(string s) {

        stack<char> st;

        for (int i = 0; i < s.size(); i++) {

            if (s[i] == '\*') {

                st.pop();

            } else {

                st.push(s[i]);

            }

        }

        string answer = "";

        while (!st.empty()) {

            answer.push\_back(st.top());

            st.pop();

        }

        reverse(answer.begin(), answer.end());

        return answer;

    }

};

**Complexity Analysis**

Here, nn*n* is the length of s.

* Time complexity: O(n)O(n)*O*(*n*)
  + We iterate over s and for every character we either push it in the stack or pop the top character from the stack which takes O(1)O(1)*O*(1) time per character. It takes O(n)O(n)*O*(*n*) time for nn*n* characters.
  + To form the answer string, we remove all the characters from the stack. Because a stack can have maximum of nn*n* characters, it would also take O(n)O(n)*O*(*n*) time in that case.
  + We also require O(n)O(n)*O*(*n*) time to reverse answer which can have nn*n* characters.
* Space complexity: O(n)O(n)*O*(*n*)
  + The stack used in the solution can grow to a maximum size of nn*n*. We would need O(n)O(n)*O*(*n*) space in that case.

**Approach 2: Strings**

**Intuition**

We can also use strings in place of a stack to handle the required operations. It can provide similar operations as stack when dealing with characters.

**Algorithm**

1. Create an empty string variable answer that will store the string while performing the required operations.
2. Iterate over the string s from start and for each index i of the string:
   * If s[i] == '\*', delete the last character from answer.
   * Otherwise, we have a non-star character, so we append it to answer.
3. Return answer.

Note: This approach does not work for Python as the strings are immutable in Python, so this would result in an O(n2)O(n^2)*O*(*n*2) time complexity.

**Implementation**

class Solution {

public:

    string removeStars(string s) {

        string answer = "";

        for (int i = 0; i < s.size(); i++) {

            if (s[i] == '\*') {

                answer.pop\_back();

            } else {

                answer.push\_back(s[i]);

            }

        }

        return answer;

    }

};

**Complexity Analysis**

* Time complexity: O(n)O(n)*O*(*n*)
  + We iterate over s and for every character we either append it to answer or delete the last character from answer which takes O(1)O(1)*O*(1) time per character. It takes O(n)O(n)*O*(*n*) time for nn*n* characters.
* Space complexity: O(n)O(n)*O*(*n*)
  + The answer string can have a maximum of nn*n* characters, requiring O(n)O(n)*O*(*n*) space. Normally, we do not count the answer towards the space complexity, but in this case we are performing logic on the answer variable, so we are counting it.

**Approach 3: Two Pointers**

**Intuition**

As previously discussed, our task is to iterate over the string while keeping track of the most recently seen non-star character who has not yet been removed.

We can also use two pointers to solve this problem, one to iterate over the given string s and the other to point to the position where the most recent non-star character is to be inserted (it will also help in removing the characters).

We create an array of characters ch having the same size as s. We also create two pointers i = 0 and j = 0.

We will iterate over s with the pointer i and add and remove non-star characters with the pointer j. The pointer j will point to the index where next non-star character is to be inserted. We will insert the characters in a data structure ch, and after iterating over the entire string s, we will have our required string in ch from index 0 till j - 1 (both inclusive).

We iterate the given string s from the start. For every index i of the string, if s[i] is a non-star character, we add s[i] to ch at index j. We increment j by 1 to insert the next non-star character at the next position.

Otherwise, if s[i] is a star character we decrement j by 1, resulting in the removal of the last character.

If we decrement j, you'll notice that whenever a non-star character is now met, it will override some character previously added in ch at the j index, resulting in the removal of the required non-star character. If there are not enough non-star characters to cover the position until where j previously went, they are still removed because we are only using indices from 0 to j - 1 to form the required string.

**Algorithm**

1. Create an array of characters ch having the same size as s.
2. Create an integer variable j that will point to the index in ch where a non-star character is to be inserted. We initialize it to 0 because the first character should be inserted at index 0.
3. Iterate over the string s from the start and for each index i of the string:
   * If s[i] == '\*', decrement j by 1 to remove the most recently seen non-star that hasn't already been removed.
   * Otherwise, we have a non-star character, we add it to ch at index j. We also increment j by 1.
4. Create an empty string variable answer.
5. Iterate over ch from index 0 till j - 1 and append all the characters to answer.
6. Return answer.

**Implementation**

class Solution {

public:

    string removeStars(string s) {

        vector<char> ch(s.size());

        int j = 0;

        for (int i = 0; i < s.size(); i++) {

            if (s[i] == '\*') {

                j--;

            } else {

                ch[j++] = s[i];

            }

        }

        string answer = "";

        for (int i = 0; i < j; i++) {

            answer.push\_back(ch[i]);

        }

        return answer;

    }

};

**Complexity Analysis**

* Time complexity: O(n)O(n)*O*(*n*)
  + It takes O(n)O(n)*O*(*n*) time to initialize a character array of nn*n* size.
  + Iterating over the string s to form the required string in ch also takes O(n)O(n)*O*(*n*) time.
  + As there can be at most nn*n* characters in answer, it takes O(n)O(n)*O*(*n*) time to form it.
* Space complexity: O(n)O(n)*O*(*n*)
  + The character array ch takes O(n)O(n)*O*(*n*) space.