**Note: L1081.**[Smallest Subsequence of Distinct Characters](https://leetcode.com/problems/smallest-subsequence-of-distinct-characters/) **same as this problem**

<https://leetcode.com/problems/remove-duplicate-letters/description/>

Given a string s, remove duplicate letters so that every letter appears once and only once. You must make sure your result is **the smallest in lexicographical order** among all possible results.

**Example 1:**

Input: s = "bcabc"

Output: "abc"

**Example 2:**

Input: s = "cbacdcbc"

Output: "acdb"

**Constraints:**

1. 1 <= s.length <= 10^4
2. s consists of lowercase English letters.

**Note:** This question is the same as 1081: <https://leetcode.com/problems/smallest-subsequence-of-distinct-characters/>

**Attempt 1: 2023-03-24**

**Solution 1: Monotonic Increasing Stack (30 min)**

**Style 1: Last index (similar usage as Style 2 Frequency array) + visited check set (same as Style 2 inStack array)**

class Solution {

    public String removeDuplicateLetters(String s) {

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

        Set<Character> visited = new HashSet<Character>();

        int[] lastIndex = new int[26];

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

            lastIndex[s.charAt(i) - 'a'] = i;

        }

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

            char c = s.charAt(i);

            if(!visited.contains(c)) {

                while(!stack.isEmpty() && stack.peek() > c && i < lastIndex[stack.peek() - 'a']) {

                    char ch = stack.pop();

                    visited.remove(ch);

                }

                stack.push(c);

                visited.add(c);

            }

        }

        StringBuilder sb = new StringBuilder();

        while(!stack.isEmpty()) {

            sb.append(stack.pop());

        }

        return sb.reverse().toString();

    }

}

**Refer to**

<https://leetcode.com/problems/remove-duplicate-letters/solutions/1859410/java-c-detailed-visually-explained/>

First Of all, we have to pick the character's if it is not already visited. If, that's the case we'll try to pick these character's. We'll also make sure, the previously picked character is smaller then the current character in order to maintain **lexicographically order**. But, how we can check the previously picked character is best for!! **And the answer is Stack!!**

What we'll do, use the **stack** to keep track of selected character's. We try to put the character's only once & maintain the

lexicographicall smallest one. **So, how we do that :-**

1. If the stack is empty, we'll put the **current character into our stack**
2. We'll also keep here **boolean array** which will mark, whether we have seen this character or not. So, that if we are getting again the same character and we have already seen that. We'll ignore that character.
3. So, the length of **boolean array** will be **26**

***Let's Understand it's working***

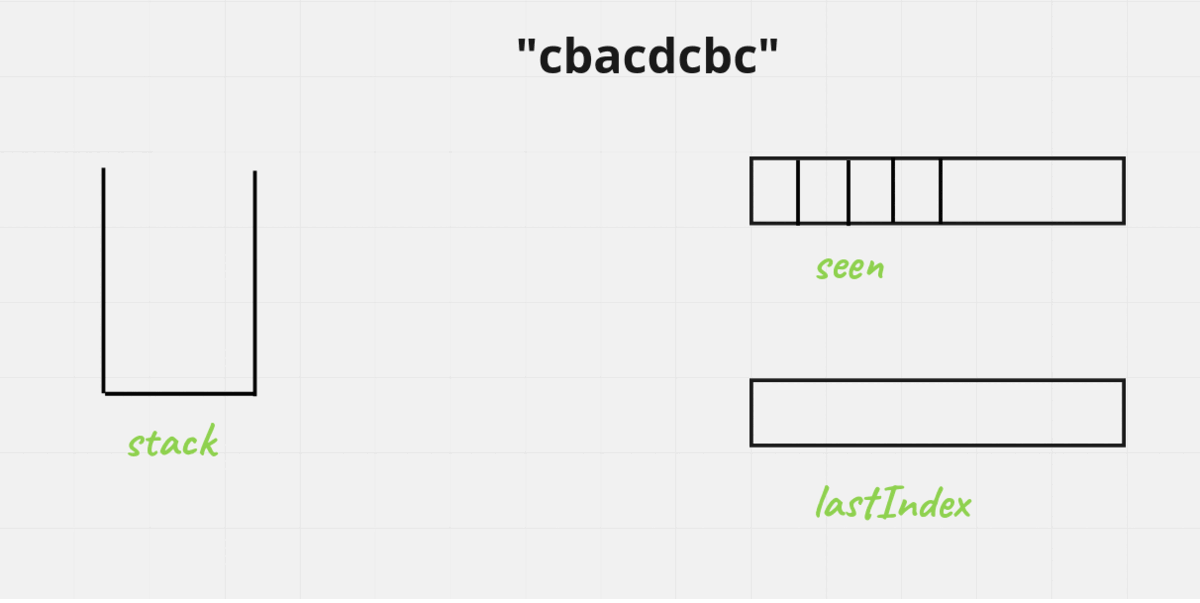
1. First we put character c into our stack and mark it as true
2. Then we come to next character i.e. b we check is **b < c** to maintain lexicographically order. Yes b is smaller then c we'll remove it from the stack.
3. But before removing we have to check that, is c more present in our string. So, how will we quickly check that for that we'll keep one more Array which will keep track of last index of all the character's present in our string
4. So, we see that c exists on 7th index.
5. We'll remove c from the stack & don't forgot to mark c in boolean array from **true to false**
6. Now add b into our stack. And mark b in boolean array as true
7. Now next character is a which is smaller then b & do the same process of checking if it exists somewhere in array & if so, remove it from stack update boolean to false. And put a into the stack. And in boolean array mark it as true.
8. Let's add c in the stack mark it as true & c > a so carry on....
9. Let's add d in the stack mark it as true & d > c so carry on....
10. Now we encounter c which is already visited so, carry on....
11. Let's add b in the stack mark it as true & b < d so carry on....

This, line explanation : BY @nirala\_414 "now try to add "b" in the stack and mark it as true, for this since the current\_element("b") is less than the peek(top) element of stack("d")so we should pop the "d" for now and push "b" in stack but before  poping "d" check is there "d" is again persent in given string after this "b" , since "d" is not more  persent after this "b" so we can't do that pop and push operation for "d" and "b" respectively , simply  add "b" into the stack and mark is as true that's set, but in some case if "d" is available again after  that "b" then we will definitely be poping "d" from the stack and adding "b" into the stack( we can take  example for this case as "cbadcbcd" and try to dry run it )

1. Now we encounter c which is already visited so, carry on....
2. End of the string.

Now whatever character we have present in our stack, take them out. i.e. **bdca** now reverse it **acdb** and this is our smallest lexicographically string

Let's understand it visually :- You can download all the images from here **G-Drive**



class Solution {

    public String removeDuplicateLetters(String s) {

        int[] lastIndex = new int[26];

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

            lastIndex[s.charAt(i) - 'a'] = i; // track the lastIndex of character presence

        }

        boolean[] seen = new boolean[26]; // keep track seen

        Stack<Integer> st = new Stack();

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

            int curr = s.charAt(i) - 'a';

            if (seen[curr]) continue; // if seen continue as we need to pick one char only

            while (!st.isEmpty() && st.peek() > curr && i < lastIndex[st.peek()]){

                seen[st.pop()] = false; // pop out and mark unseen

            }

            st.push(curr); // add into stack

            seen[curr] = true; // mark seen

        }

        StringBuilder sb = new StringBuilder();

        while (!st.isEmpty())

            sb.append((char) (st.pop() + 'a'));

        return sb.reverse().toString();

    }

}

ANALYSIS :-

Time Complexity :- BigO(N)

Space Complexity :- BigO(N)

**Style 2: Frequency + inStack array**

class Solution {

    public String removeDuplicateLetters(String s) {

        // Boolean array to track characters that are already in the stack

        boolean[] inStack = new boolean[26];

        // Frequency array to count occurrences of each character

        int[] freq = new int[26];

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

            freq[c - 'a']++;

        }

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

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

            // Decrease the frequency of the current character

            freq[c - 'a']--;

            // If the character is already in the stack, skip it

            if(!inStack[c - 'a']) {

                // While the stack is not empty, the top of the stack is lexicographically greater

                // than the current character, and the top character can appear later, pop the stack

                while(!stack.isEmpty() && stack.peek() > c && freq[stack.peek() - 'a'] > 0) {

                    inStack[stack.pop() - 'a'] = false;

                }

                // Push the current character into the stack and mark it as in the stack

                stack.push(c);

                inStack[c - 'a'] = true;

            }

        }

        StringBuilder sb = new StringBuilder();

        while(!stack.isEmpty()) {

            sb.append(stack.pop());

        }

        return sb.reverse().toString();

    }

}

Time Complexity: O(n)

Space Complexity: O(1)

**Refer to chatGPT**

LeetCode 316 is titled "**Remove Duplicate Letters**". The task is to remove duplicate letters from a string such that every letter appears only once, and the result is the lexicographically smallest possible string.

This problem is very similar to LeetCode 1081 ("Smallest Subsequence of Distinct Characters"), and we can apply the same **greedy approach** with a **monotonic stack** to solve it.

**Problem:**

Given a string s, remove duplicate letters so that every letter appears once and only once. You must make sure that the resulting string is the smallest in lexicographical order among all possible results.

**Approach:**

To solve this, the algorithm is:

1. Count occurrences: First, count how many times each character appears in the string.
2. Stack maintenance:
   1. Use a stack (or a StringBuilder) to maintain the characters of the result.
   2. For each character, pop the stack if the current character is smaller than the top of the stack and if the top character can appear later (i.e., it still has remaining occurrences).
   3. Ensure each character appears only once by tracking which characters are already in the stack.

public class Solution {

public String removeDuplicateLetters(String s) {

// Frequency array to count occurrences of each character

int[] freq = new int[26];

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

freq[c - 'a']++;

}

// Boolean array to track characters that are already in the stack

boolean[] inStack = new boolean[26];

// Stack to maintain the smallest lexicographical order

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

// Iterate through the string

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

// Decrease the frequency of the current character

freq[c - 'a']--;

// If the character is already in the stack, skip it

if (inStack[c - 'a']) {

continue;

}

// While the stack is not empty, the top of the stack is lexicographically greater

// than the current character, and the top character can appear later, pop the stack

while (!stack.isEmpty() && stack.peek() > c && freq[stack.peek() - 'a'] > 0) {

inStack[stack.pop() - 'a'] = false;

}

// Push the current character into the stack and mark it as in the stack

stack.push(c);

inStack[c - 'a'] = true;

}

// Convert the stack to a string

StringBuilder result = new StringBuilder();

for (char c : stack) {

result.append(c);

}

return result.reverse().toString();

}

}

Explanation:

1. Frequency array: freq[] counts how many times each character appears in the string.
2. Boolean array: inStack[] keeps track of whether a character is already in the stack (result).
3. Stack logic:
   1. For each character c, we first reduce its frequency in the freq[] array.
   2. If c is already in the stack (tracked using inStack[]), we skip it.
   3. While the stack is not empty, and the top character in the stack is lexicographically larger than c, and the top character still has occurrences left (freq[stack.peek() - 'a'] > 0), we pop the top character from the stack and mark it as not in the stack.
   4. Push the current character c into the stack and mark it as being in the stack.
4. Final result: After iterating through all the characters, the stack will contain the lexicographically smallest sequence of distinct characters.

**Refer to**

[L402.Remove K Digits (Ref.L1673,L84)](note://09B83C91043B44D5BC818F72BEE20307)

[L1673.Find the Most Competitive Subsequence (Ref.L84,L402)](note://WEBd8e85b624521547b3108241deb26a6cb)

[L84.Largest Rectangle in Histogram](note://E2FC22D9715540C5A8F06F60FBBE886F)