Problem 3563: Lexicographically Smallest String After Adjacent Removals

Problem Information

Difficulty: Hard
Acceptance Rate: 0.00%
Paid Only: No

Problem Description

You are given a string

s

consisting of lowercase English letters.

You can perform the following operation any number of times (including zero):

Remove

any

pair of

adjacent

characters in the string that are

consecutive

in the alphabet, in either order (e.g.,

'a'

and

'b'

, or

'b'

and

'a'

).

Shift the remaining characters to the left to fill the gap.

Return the

lexicographically smallest

string that can be obtained after performing the operations optimally.

Note:

Consider the alphabet as circular, thus

'a'

and

'z'

are consecutive.

Example 1:

Input:

s = "abc"

Output:

"a"

Explanation:

Remove

"bc"

from the string, leaving

"a"

as the remaining string.

No further operations are possible. Thus, the lexicographically smallest string after all possible removals is

"a"

.

Example 2:

Input:

s = "bcda"

Output:

""

Explanation:

Remove

"cd"

from the string, leaving

"ba"

as the remaining string.

Remove

"ba"

from the string, leaving

""

as the remaining string.

No further operations are possible. Thus, the lexicographically smallest string after all possible removals is

""

.

Example 3:

Input:

s = "zdce"

Output:

"zdce"

Explanation:

Remove

"dc"

from the string, leaving

"ze"

as the remaining string.

No further operations are possible on

"ze"

.

However, since

"zdce"

is lexicographically smaller than

"ze"

, the smallest string after all possible removals is

"zdce"

.

Constraints:

1 <= s.length <= 250

s

consists only of lowercase English letters.

Code Snippets

C++:

```
class Solution {
public:
string lexicographicallySmallestString(string s) {

}
};
```

Java:

```
class Solution {
public String lexicographicallySmallestString(String s) {

}
}
```

Python3:

```
class Solution:
def lexicographicallySmallestString(self, s: str) -> str:
```

Python:

```
class Solution(object):
def lexicographicallySmallestString(self, s):
"""
:type s: str
:rtype: str
"""
```

JavaScript:

```
/**
* @param {string} s
* @return {string}
*/
var lexicographicallySmallestString = function(s) {

};
```

TypeScript:

```
function lexicographicallySmallestString(s: string): string {
```

```
};
```

C#:

```
public class Solution {
public string LexicographicallySmallestString(string s) {

}
}
```

C:

```
char* lexicographicallySmallestString(char* s) {

}
```

Go:

```
func lexicographicallySmallestString(s string) string {

}
```

Kotlin:

```
class Solution {
fun lexicographicallySmallestString(s: String): String {

}
}
```

Swift:

```
class Solution {
func lexicographicallySmallestString(_ s: String) -> String {

}
}
```

Rust:

```
impl Solution {
pub fn lexicographically_smallest_string(s: String) -> String {
```

```
}
}
```

Ruby:

```
# @param {String} s
# @return {String}
def lexicographically_smallest_string(s)

end
```

PHP:

```
class Solution {

/**
* @param String $s
* @return String
*/
function lexicographicallySmallestString($s) {

}
}
```

Dart:

```
class Solution {
String lexicographicallySmallestString(String s) {

}
}
```

Scala:

```
object Solution {
def lexicographicallySmallestString(s: String): String = {

}
}
```

Elixir:

```
defmodule Solution do
@spec lexicographically_smallest_string(s :: String.t) :: String.t
def lexicographically_smallest_string(s) do

end
end
```

Erlang:

```
-spec lexicographically_smallest_string(S :: unicode:unicode_binary()) ->
unicode:unicode_binary().
lexicographically_smallest_string(S) ->
.
```

Racket:

```
(define/contract (lexicographically-smallest-string s)
(-> string? string?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Lexicographically Smallest String After Adjacent Removals
* Difficulty: Hard
* Tags: string, graph, dp
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

class Solution {
public:
string lexicographicallySmallestString(string s) {

}
};
```

Java Solution:

```
/**
* Problem: Lexicographically Smallest String After Adjacent Removals
* Difficulty: Hard
* Tags: string, graph, dp
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

class Solution {
public String lexicographicallySmallestString(String s) {

}
}
```

Python3 Solution:

```
"""
Problem: Lexicographically Smallest String After Adjacent Removals
Difficulty: Hard
Tags: string, graph, dp

Approach: String manipulation with hash map or two pointers
Time Complexity: O(n) or O(n log n)
Space Complexity: O(n) or O(n * m) for DP table
"""

class Solution:
def lexicographicallySmallestString(self, s: str) -> str:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def lexicographicallySmallestString(self, s):
"""
:type s: str
:rtype: str
"""
```

JavaScript Solution:

```
/**
* Problem: Lexicographically Smallest String After Adjacent Removals
* Difficulty: Hard
* Tags: string, graph, dp
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

/**
* @param {string} s
* @return {string}
*/
var lexicographicallySmallestString = function(s) {

};
```

TypeScript Solution:

```
/**
* Problem: Lexicographically Smallest String After Adjacent Removals
* Difficulty: Hard
* Tags: string, graph, dp
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

function lexicographicallySmallestString(s: string): string {

};
```

C# Solution:

```
/*
* Problem: Lexicographically Smallest String After Adjacent Removals
* Difficulty: Hard
* Tags: string, graph, dp
```

```
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

public class Solution {
public string LexicographicallySmallestString(string s) {

}
}
```

C Solution:

```
/*
* Problem: Lexicographically Smallest String After Adjacent Removals
* Difficulty: Hard
* Tags: string, graph, dp
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

char* lexicographicallySmallestString(char* s) {

}
```

Go Solution:

```
// Problem: Lexicographically Smallest String After Adjacent Removals
// Difficulty: Hard
// Tags: string, graph, dp
//
// Approach: String manipulation with hash map or two pointers
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) or O(n * m) for DP table

func lexicographicallySmallestString(s string) string {

}
```

Kotlin Solution:

```
class Solution {
fun lexicographicallySmallestString(s: String): String {

}
}
```

Swift Solution:

```
class Solution {
func lexicographicallySmallestString(_ s: String) -> String {

}
}
```

Rust Solution:

```
// Problem: Lexicographically Smallest String After Adjacent Removals
// Difficulty: Hard
// Tags: string, graph, dp
//
// Approach: String manipulation with hash map or two pointers
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) or O(n * m) for DP table

impl Solution {
pub fn lexicographically_smallest_string(s: String) -> String {

}
}
```

Ruby Solution:

```
# @param {String} s
# @return {String}
def lexicographically_smallest_string(s)

end
```

PHP Solution:

```
class Solution {

/**
* @param String $s
* @return String
*/
function lexicographicallySmallestString($s) {

}
}
```

Dart Solution:

```
class Solution {
String lexicographicallySmallestString(String s) {

}
}
```

Scala Solution:

```
object Solution {
def lexicographicallySmallestString(s: String): String = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec lexicographically_smallest_string(s :: String.t) :: String.t
def lexicographically_smallest_string(s) do

end
end
```

Erlang Solution:

```
-spec lexicographically_smallest_string(S :: unicode:unicode_binary()) ->
unicode:unicode_binary().
lexicographically_smallest_string(S) ->
.
```

Racket Solution:

```
(define/contract (lexicographically-smallest-string s)
(-> string? string?)
)
```