2296. Design a Text Editor Design String Doubly-Linked List Simulation Stack Linked List Leetcode Link Hard

Problem Description

right than the last character of the text.

cursor left or right. Unlike standard text editors that deal with complex rendering and formatting, our task is to simulate how the internal state of the text editor changes as these operations are performed. The editor will keep track of the cursor position and allow the following actions: Add text: Insert a given string of text where the cursor is currently positioned. After the text is added, the cursor should be at

The problem requires us to design a simple text editor with basic functionalities such as adding text, deleting text, and moving the

- the end of the newly added text. Delete text: Remove a specified number of characters from the text, directly to the left of the cursor position. The function
- should return the actual number of characters deleted, which could be less than the requested if there are not enough characters. Move cursor left: Shift the cursor to the left 'k' positions. If there aren't 'k' characters to the left of the cursor, move the cursor as far as possible. After moving, the function should return a string containing the last 10 (or fewer, if less than 10 characters are
- available) characters to the left of the cursor. Move cursor right: Shift the cursor to the right 'k' positions. Similarly, this should also return the last 10 (or fewer) characters to the left of the text where the cursor is now placed after the movement.
- Intuition A straightforward approach to solve this problem is to use two stacks—a "left" stack to represent the text before the cursor and a

The cursor is limited to the bounds of the current text content, meaning it cannot move further left than the first character or further

"right" stack to represent the text after the cursor. This method leverages the properties of stacks to manage the characters efficiently in both directions relative to the cursor:

1. When we add text, we push the new characters to the "left" stack, as they are added where the cursor currently is, which is at

the end of the existing text in the "left" stack. 2. When performing the delete operation, we pop characters from the "left" stack, simulating the deletion of characters to the left of the cursor.

3. To move the cursor to the left, we pop characters from the "left" stack and push them onto the "right" stack, effectively moving

- the cursor backwards. The last 10 characters of the "left" stack are then concatenated into a string and returned. 4. To move the cursor to the right, we pop characters from the "right" stack and push them onto the "left" stack, like 'undoing' a leftward cursor move. Again, we concatenate and return the last 10 characters of the "left" stack.
- Through the push and pop operations of stacks, we can control the text before and after the cursor efficiently, and maintain an accurate picture of what the characters around the cursor would be after each operation.
- The solution approach for the text editor design involves using two stacks, referred to in the code as self.left and self.right, to hold characters on each side of the cursor. Here's a detailed explanation of how each method in the TextEditor class works: • Constructor (__init__): Initializes two lists, self.left and self.right, which act as stacks. These stacks are empty at the start,

 Add text (addText): Takes a string text as input and extends the self.left stack with the new characters. This simulates typing at the cursor's position, as characters are appended where the cursor is. After adding, the cursor ends up to the right of the new text, naturally.

representing an empty text state with the cursor at the initial position.

• Move cursor left (cursorLeft): Transfers k characters from the self.left stack to the self.right stack, if available, moving the

editor design.

1. Add text "hello".

3. Add text "X".

2. Move cursor left by 2 positions.

right = [] (Cursor is now at the end of "hello")

cursor's position, as described in the solution approach.

self.left_stack = []

self.right_stack = []

return k

def addText(self, text: str) -> None:

self.left_stack.extend(text)

def deleteText(self, k: int) -> int:

def cursorLeft(self, k: int) -> str:

for _ in range(k):

k = min(k, len(self.left_stack))

def cursorRight(self, k: int) -> str:

k = min(k, len(self.right_stack))

return ''.join(self.left_stack[-10:])

4. Delete text of 1 character.

Solution Approach

deletible characters. The method then pops these characters from the self.left stack and returns the count of the deleted characters. Each pop removes a character on the left of the cursor, mimicking a backspace operation.

• Delete text (deleteText): Takes an integer k representing the number of characters to delete. The actual number of deletions

cannot exceed the number of characters in self.left, so min(k, len(self.left)) is calculated to determine the true number of

cursor left k times. If there are fewer than k characters, it moves the cursor to the start. It then returns a string composed of the

last 10 (or fewer if less than 10 are available) characters of the self.left stack, reflecting the text immediately left of the cursor after the move. • Move cursor right (cursorRight): Moves the cursor to the right by transferring k characters from the self.right stack back to

the self. left stack, simulating a forward cursor movement. As with the left move, it only moves as far as characters exist in

These stack operations (push and pop) allow the text editor to efficiently mimic the addition, deletion, and cursor movement actions.

Stacks are chosen due to their LIFO (last in, first out) nature, which perfectly suits the operations we need to simulate for this text

self.right. It then returns a string of up to the last 10 characters from the self.left stack.

Example Walkthrough Let's consider we are using the TextEditor class described above to perform a series of operations on our text editor. Suppose we start with an empty text field, and we want to perform the following actions:

5. Move cursor right by 1 position. Here's how the TextEditor class handles these operations: • Initial State: (Cursor at the start) left = [], right = []

Add text "hello": The addText method appends each character of "hello" to self.left.left = ['h', 'e', 'l', 'l', 'o'],

Add text "X": The addText method adds the "X" character to self.left.left = ['h', 'e', 'l', 'X'], right = ['l', 'o']

• Delete text of 1 character: The deleteText(1) method deletes 1 character from the left of the cursor by popping from self.left.

• Move cursor left by 2: The cursorLeft(2) method moves the cursor 2 positions to the left. It pops two characters from

self.left and pushes them into self.right.left = ['h', 'e', 'l'], right = ['l', 'o'] Last 10 characters of the left stack: "hel"

"hell"

class TextEditor:

6

8

9

10

11

12

13

14

15

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

43

55

6

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

35

36

37

38

39

40

41

42

43

45

46

47

48

49

50

51

52

53

54

55

57

56 };

59 /*

66 */

67

8

9

10

14

15

16

17

18

26

27

28

29

30

31

32

34

38

39

40

41

42

43

44

45

46

48

47 }

49 // Example usage:

Time Complexity

50 // addText("hello");

33 }

19 }

private:

string cursorRight(int k) {

// Example of how to use the TextEditor class

TextEditor* editor = new TextEditor();

62 int deletedChars = editor->deleteText(3);

let textLeftOfCursor: string = "";

textLeftOfCursor += text;

let textRightOfCursor: string = "";

function addText(text: string): void {

12 function deleteText(k: number): number {

return actualDeleteCount;

function cursorRight(k: number): string {

51 // let deletedChars: number = deleteText(3);

Time and Space Complexity

1. addText(text: str) -> None:

52 // let leftSubstring: string = cursorLeft(2);

53 // let rightSubstring: string = cursorRight(2);

while (k-->0) {

k = Math.min(k, textRightOfCursor.length);

63 string leftSubstring = editor->cursorLeft(2);

64 string rightSubstring = editor->cursorRight(2);

delete editor; // Release the allocated memory

61 editor->addText("hello");

Typescript Solution

while (k--) {

(Cursor is now after "X")

left = ['h', 'e', 'l'], right = ['l', 'o'] 1 character was deleted. • Move cursor right by 1: The cursorRight(1) method moves the cursor 1 position to the right. It pops one character from

After all these operations, the state of the text editor and stacks are as follows:

Stack for the text on the left side of the cursor

Stack for the text on the right side of the cursor

Extend the left stack with characters from the text

"""Delete 'k' characters to the left of the cursor."""

"""Move the cursor 'k' positions to the left."""

"""Move the cursor 'k' positions to the right."""

Ensure 'k' does not exceed the number of characters on the left

Ensure 'k' does not exceed the number of characters on the left

Ensure 'k' does not exceed the number of characters on the right

Move 'k' characters from the right stack to the left stack

Return the last 10 characters on the left stack or all if less than 10

Move 'k' characters from the left stack to the right stack

self.right_stack.append(self.left_stack.pop())

"""Add text to the left side of the cursor."""

Text Content: "hell o" (with the cursor positioned at "|") left = ['h', 'e', 'l', 'l'] right = ['o']

self.right and pushes it into self.left.left = ['h', 'e', 'l', 'l'], right = ['o'] Last 10 characters of the left stack:

Python Solution

Each of these operations efficiently alters the state of the text editor by using the two-stack approach while keeping track of the

16 k = min(k, len(self.left_stack)) 17 # Pop 'k' characters from the left stack 18 for _ in range(k): 19 self.left_stack.pop() 20 # Return the number of characters deleted

38 for _ in range(k): 39 self.left_stack.append(self.right_stack.pop()) 40 # Return the last 10 characters on the left stack or all if less than 10 41 return ''.join(self.left_stack[-10:]) 42

47 # Add text

Java Solution

class TextEditor {

Example usage:

49 # Delete 3 characters

45 # Initialize the text editor

48 text_editor.addText("hello")

50 deleted_chars = text_editor.deleteText(3)

52 display_text = text_editor.cursorLeft(2)

54 display_text = text_editor.cursorRight(2)

// Constructor for the text editor.

// Intentionally left empty.

public void addText(String text) {

leftText.append(text);

public int deleteText(int k) {

return actualDeleteCount;

public String cursorLeft(int k)

public TextEditor() {

private StringBuilder leftText = new StringBuilder();

private StringBuilder rightText = new StringBuilder();

// Adds the given text to the end of the current cursor position.

int actualDeleteCount = Math.min(k, leftText.length());

int actualMoveCount = Math.min(k, leftText.length());

leftText.deleteCharAt(leftText.length() - 1);

for (int i = 0; i < actualMoveCount; ++i) {</pre>

leftText.setLength(leftText.length() - actualDeleteCount);

rightText.append(leftText.charAt(leftText.length() - 1));

return leftText.substring(Math.max(leftText.length() - 10, 0));

// Get the substring of the last 10 characters or entire text if it's shorter.

// Deletes the 'k' characters before the cursor and returns the count of characters deleted.

// Moves the cursor to the left by 'k' positions and returns the text within 10 positions to the left of the cursor.

51 # Move cursor 2 positions to the left

53 # Move cursor 2 positions to the right

46 text_editor = TextEditor()

```
33
       // Moves the cursor to the right by 'k' positions and returns the text within 10 positions to the left of the cursor.
       public String cursorRight(int k) {
34
35
           int actualMoveCount = Math.min(k, rightText.length());
36
           for (int i = 0; i < actualMoveCount; ++i) {</pre>
37
               char ch = rightText.charAt(rightText.length() - 1);
38
               leftText.append(ch);
               rightText.deleteCharAt(rightText.length() - 1);
39
40
           // Get the substring of the last 10 characters or entire text if it's shorter.
41
           return leftText.substring(Math.max(leftText.length() - 10, 0));
42
43
44
45
46
    * Usage of TextEditor:
    * TextEditor textEditor = new TextEditor();
    * textEditor.addText(text); // Adds text at the current cursor position.
    * int deletedCount = textEditor.deleteText(k); // Deletes 'k' characters from the left of the cursor.
    * String leftText = textEditor.cursorLeft(k); // Moves cursor 'k' positions to the left.
    * String rightText = textEditor.cursorRight(k); // Moves cursor 'k' positions to the right.
53
    */
54
C++ Solution
   #include <string>
    #include <algorithm>
  4 using std::string;
  5 using std::min;
  6 using std::max;
  8 class TextEditor {
    public:
         // Constructor initializes an empty text editor
 10
 11
         TextEditor() {
             // Both strings are initially empty
 12
             textLeftOfCursor = "";
 13
             textRightOfCursor = "";
 14
 15
 16
 17
         // Function to add text directly before the cursor's current position
         void addText(string text) {
 18
 19
             textLeftOfCursor += text; // Append the text to the left part (before cursor)
 20
 21
 22
         // Function to delete 'k' characters before the cursor's position
 23
         int deleteText(int k) {
 24
             int actualDeleteCount = min(k, static_cast<int>(textLeftOfCursor.size()));
             textLeftOfCursor.resize(textLeftOfCursor.size() - actualDeleteCount); // Delete characters from the end
 25
 26
             return actualDeleteCount; // Return the number of characters that were actually deleted
 27
 28
 29
         // Function to move the cursor 'k' positions to the left, returning at most 10 characters before the cursor
         string cursorLeft(int k) {
 30
 31
             k = min(k, static_cast < int > (textLeftOfCursor.size())); // Bound k to the number of characters available to move
 32
             while (k--) {
 33
                 textRightOfCursor += textLeftOfCursor.back(); // Move last character of left text to the start of right text
 34
                 textLeftOfCursor.pop back(); // Remove the moved character from the left text
```

// Return at most 10 characters of the left text, truncating from the front if needed

return textLeftOfCursor.substr(max(0, static_cast<int>(textLeftOfCursor.size()) - 10));

textRightOfCursor.pop_back(); // Remove the moved character from the right text

// Return at most 10 characters of the left text, truncating from the front if needed

string textLeftOfCursor; // Stores the text left of the cursor

string textRightOfCursor; // Stores the text right of the cursor

1 // Initialize two strings to simulate the text to the left and right of the cursor.

// Function to add text directly before the cursor's current position.

// Determine the actual number of characters that can be deleted.

textLeftOfCursor = textLeftOfCursor.slice(0, -actualDeleteCount);

// Return the number of characters that were actually deleted.

const actualDeleteCount: number = Math.min(k, textLeftOfCursor.length);

// Move the last character of the left text to the start of the right text.

// Return at most 10 characters of the left text, truncating from the front if needed.

// Move the last character of the right text to the end of the left text.

textLeftOfCursor += textRightOfCursor.charAt(textRightOfCursor.length - 1);

// Return at most 10 characters of the left text, truncating from the front if needed.

return textLeftOfCursor.slice(Math.max(0, textLeftOfCursor.length - 10));

return textLeftOfCursor.slice(Math.max(0, textLeftOfCursor.length - 10));

// Bound 'k' to the number of characters available to move right.

// Remove the moved character from the right text.

textRightOfCursor = textRightOfCursor.slice(0, -1);

textRightOfCursor = textLeftOfCursor.charAt(textLeftOfCursor.length - 1) + textRightOfCursor;

35 // Function to move the cursor 'k' positions to the right, returning at most 10 characters before the cursor.

// Append the text to the left part (before cursor).

11 // Function to delete 'k' characters before the cursor's position.

// Delete the specified number of characters from the end.

// Remove the moved character from the left text.

textLeftOfCursor = textLeftOfCursor.slice(0, -1);

return textLeftOfCursor.substr(max(0, static_cast<int>(textLeftOfCursor.size()) - 10));

// Function to move the cursor 'k' positions to the right, returning at most 10 characters before the cursor

 $k = min(k, static_cast < int > (textRightOfCursor.size())); // Bound k to the number of characters available to move$

textLeftOfCursor += textRightOfCursor.back(); // Move last character of right text to the end of left text

20 21 // Function to move the cursor 'k' positions to the left, returning at most 10 characters before the cursor. function cursorLeft(k: number): string { 23 // Bound 'k' to the number of characters available to move left. k = Math.min(k, textLeftOfCursor.length); 24 25 while (k-->0) {

 Appending each character from the input text to the self.left list has a time complexity of O(n) where n is the length of text. 2. deleteText(k: int) -> int:

```
    Moving cursor left k times involves popping elements from self.left and appending to self.right, each operation takes

 0(1) so overall operation is 0(k). Then, joining the last 10 characters of self. left is 0(1), as it deals with at most 10
```

10 characters. So, overall complexity is O(k).

operations are performed.

3. cursorLeft(k: int) -> str:

characters regardless of the input size. 4. cursorRight(k: int) -> str: Similar to cursorLeft, moving the cursor right has a time complexity of O(k) for moving characters, and O(1) for joining up to

• Deleting k elements from the end of self.left list has a time complexity of O(k) since each pop operation is O(1) and k such

Space Complexity • The overall space complexity of the TextEditor class is 0(m + n) where m is the total number of characters to the left of cursor

and n is the total number of characters to the right of cursor. Since the input text is stored within these two lists, this accounts for the space taken up by the editor's content.