2296. Design a Text Editor

Hard | Stack | Design | Linked List | String | Doubly-Linked List | Simulation   Leetcode Link

Problem Description

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

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 than the last character of the text.

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 "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.

Solution Approach

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

- **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.

- **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 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.

- **Move cursor left (`cursorLeft`)**: Transfers `k` characters from the `self.left` stack to the `self.right` stack, if available, moving the 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 `self.right`. It then returns a string of up to the last 10 characters from the `self.left` stack.

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 editor design.

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:

1. Add text "hello".
2. Move cursor left by 2 positions.
3. Add text "X".
4. Delete text of 1 character.
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']`, `right = []` (Cursor is now at the end of "hello")

- **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"

- **Add text "X"**: The `addText` method adds the "X" character to `self.left`. `left = ['h', 'e', 'l', 'X']`, `right = ['l', 'o']` (Cursor is now after "X")

- **Delete text of 1 character**: The `deleteText(1)` method deletes 1 character from the left of the cursor by popping from `self.left`. `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 `self.right` and pushes it into `self.left`. `left = ['h', 'e', 'l', 'l']`, `right = ['o']` Last 10 characters of the left stack: "hell"

After all these operations, the state of the text editor and stacks are as follows:
Text Content: "hell|o" (with the cursor positioned at "|")
`left = ['h', 'e', 'l', 'l']`
`right = ['o']`

Each of these operations efficiently alters the state of the text editor by using the two-stack approach while keeping track of the cursor's position, as described in the solution approach.

Python Solution

```python
class TextEditor:
    def __init__(self):
        # Stack for the text on the left side of the cursor
        self.left_stack = []
        # Stack for the text on the right side of the cursor
        self.right_stack = []

    def addText(self, text: str) -> None:
        """Add text to the left side of the cursor."""
        # Extend the left stack with characters from the text
        self.left_stack.extend(text)

    def deleteText(self, k: int) -> int:
        """Delete 'k' characters to the left of the cursor."""
        # Ensure 'k' does not exceed the number of characters on the left
        k = min(k, len(self.left_stack))
        # Pop 'k' characters from the left stack
        for _ in range(k):
            self.left_stack.pop()
        # Return the number of characters deleted
        return k

    def cursorLeft(self, k: int) -> str:
        """Move the cursor 'k' positions to the left."""
        # Ensure 'k' does not exceed the number of characters on the left
        k = min(k, len(self.left_stack))
        # Move 'k' characters from the left stack to the right stack
        for _ in range(k):
            self.right_stack.append(self.left_stack.pop())
        # Return the last 10 characters on the left stack or all if less than 10
        return ''.join(self.left_stack[-10:])

    def cursorRight(self, k: int) -> str:
        """Move the cursor 'k' positions to the right."""
        # Ensure 'k' does not exceed the number of characters on the right
        k = min(k, len(self.right_stack))
        # Move 'k' characters from the right stack to the left stack
        for _ in range(k):
            self.left_stack.append(self.right_stack.pop())
        # Return the last 10 characters on the left stack or all if less than 10
        return ''.join(self.left_stack[-10:])


# Example usage:
# Initialize the text editor
text_editor = TextEditor()
# Add text
text_editor.addText("hello")
# Delete 3 characters
deleted_chars = text_editor.deleteText(3)
# Move cursor 2 positions to the left
display_text = text_editor.cursorLeft(2)
# Move cursor 2 positions to the right
display_text = text_editor.cursorRight(2)
```

Java Solution

```java
class TextEditor {
    private StringBuilder leftText = new StringBuilder();
    private StringBuilder rightText = new StringBuilder();

    // Constructor for the text editor.
    public TextEditor() {
        // Intentionally left empty.
    }

    // Adds the given text to the end of the current cursor position.
    public void addText(String text) {
        leftText.append(text);
    }

    // Deletes the 'k' characters before the cursor and returns the count of characters deleted.
    public int deleteText(int k) {
        int actualDeleteCount = Math.min(k, leftText.length());
        leftText.setLength(leftText.length() - actualDeleteCount);
        return actualDeleteCount;
    }

    // Moves the cursor to the left by 'k' positions and returns the text within 10 positions to the left of the cursor.
    public String cursorLeft(int k) {
        int actualMoveCount = Math.min(k, leftText.length());
        for (int i = 0; i < actualMoveCount; ++i) {
            rightText.append(leftText.charAt(leftText.length() - 1));
            leftText.deleteCharAt(leftText.length() - 1);
        }
        // Get the substring of the last 10 characters or entire text if it's shorter.
        return leftText.substring(Math.max(leftText.length() - 10, 0));
    }

    // 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) {
        int actualMoveCount = Math.min(k, rightText.length());
        for (int i = 0; i < actualMoveCount; ++i) {
            char ch = rightText.charAt(rightText.length() - 1);
            leftText.append(ch);
            rightText.deleteCharAt(rightText.length() - 1);
        }
        // Get the substring of the last 10 characters or entire text if it's shorter.
        return leftText.substring(Math.max(leftText.length() - 10, 0));
    }
}

/*
 * 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.
 */
```

C++ Solution

```cpp
#include <string>
#include <algorithm>

using std::string;
using std::min;
using std::max;

class TextEditor {
public:
    // Constructor initializes an empty text editor
    TextEditor() {
        // Both strings are initially empty
        textLeftOfCursor = "";
        textRightOfCursor = "";
    }

    // Function to add text directly before the cursor's current position
    void addText(string text) {
        textLeftOfCursor += text; // Append the text to the left part (before cursor)
    }

    // Function to delete 'k' characters before the cursor's position
    int deleteText(int k) {
        int actualDeleteCount = min(k, static_cast<int>(textLeftOfCursor.size()));
        textLeftOfCursor.resize(textLeftOfCursor.size() - actualDeleteCount); // Delete characters from the end
        return actualDeleteCount; // Return the number of characters that were actually deleted
    }

    // Function to move the cursor 'k' positions to the left, returning at most 10 characters before the cursor
    string cursorLeft(int k) {
        k = min(k, static_cast<int>(textLeftOfCursor.size())); // Bound k to the number of characters available to move
        while (k--) {
            textRightOfCursor += textLeftOfCursor.back(); // Move last character of left text to the start of right text
            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));
    }

    // Function to move the cursor 'k' positions to the right, returning at most 10 characters before the cursor
    string cursorRight(int k) {
        k = min(k, static_cast<int>(textRightOfCursor.size())); // Bound k to the number of characters available to move
        while (k--) {
            textLeftOfCursor += textRightOfCursor.back(); // Move last character of right text to the end of left text
            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
        return textLeftOfCursor.substr(max(0, static_cast<int>(textLeftOfCursor.size()) - 10));
    }

private:
    string textLeftOfCursor;   // Stores the text left of the cursor
    string textRightOfCursor;  // Stores the text right of the cursor
};

// Example of how to use the TextEditor class
/*
TextEditor* editor = new TextEditor();
editor->addText("hello");
int deletedChars = editor->deleteText(3);
string leftSubstring = editor->cursorLeft(2);
string rightSubstring = editor->cursorRight(2);
delete editor; // Release the allocated memory
*/
```

Typescript Solution

```typescript
// Initialize two strings to simulate the text to the left and right of the cursor.
let textLeftOfCursor: string = "";
let textRightOfCursor: string = "";

// Function to add text directly before the cursor's current position.
function addText(text: string): void {
    // Append the text to the left part (before cursor).
    textLeftOfCursor += text;
}

// Function to delete 'k' characters before the cursor's position.
function deleteText(k: number): number {
    // Determine the actual number of characters that can be deleted.
    const actualDeleteCount: number = Math.min(k, textLeftOfCursor.length);
    // Delete the specified number of characters from the end.
    textLeftOfCursor = textLeftOfCursor.slice(0, -actualDeleteCount);
    // Return the number of characters that were actually deleted.
    return actualDeleteCount;
}

// Function to move the cursor 'k' positions to the left, returning at most 10 characters before the cursor.
function cursorLeft(k: number): string {
    // Bound 'k' to the number of characters available to move left.
    k = Math.min(k, textLeftOfCursor.length);
    while (k-- > 0) {
        // Move the last character of the left text to the start of the right text.
        textRightOfCursor = textLeftOfCursor.charAt(textLeftOfCursor.length - 1) + textRightOfCursor;
        // Remove the moved character from the left text.
        textLeftOfCursor = textLeftOfCursor.slice(0, -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));
}

// Function to move the cursor 'k' positions to the right, returning at most 10 characters before the cursor.
function cursorRight(k: number): string {
    // Bound 'k' to the number of characters available to move right.
    k = Math.min(k, textRightOfCursor.length);
    while (k-- > 0) {
        // Move the last character of the right text to the end of the left text.
        textLeftOfCursor += textRightOfCursor.charAt(textRightOfCursor.length - 1);
        // Remove the moved character from the right text.
        textRightOfCursor = textRightOfCursor.slice(0, -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));
}

// Example usage:
// addText("hello");
// let deletedChars: number = deleteText(3);
// let leftSubstring: string = cursorLeft(2);
// let rightSubstring: string = cursorRight(2);
```

Time and Space Complexity

Time Complexity

1. `addText(text: str) -> None`:
   - 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`:
   - 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 operations are performed.
3. `cursorLeft(k: int) -> str`:
   - Moving cursor left k times involves popping elements from `self.left` and appending to `self.right`, each operation takes $O(1)$ so overall operation is $O(k)$. Then, joining the last 10 characters of `self.left` is $O(1)$, as it deals with at most 10 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 10 characters. So, overall complexity is $O(k)$.

Space Complexity

- The overall space complexity of the `TextEditor` class is $O(m + n)$ 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.