Fatima Jinnah Women University

Data Structure Report

Stack Text Editor

Submitted To:

Sir Adeel Khalid

Submitted By:

Aima Malik (001)

Ayesha Nadeem (020)

Khadija Shafqat (051)

Mahnoor Sajjid (054)

Project: Stack Text Editor

what is basically stack text based editor

A stack-based text editor uses the stack data structure to manage the history of text changes. Here's a more detailed explanation:

Stack Data Structure

A stack is a last-in, first-out (LIFO) data structure. It means the last element added to the stack is the first one to be removed. The main operations of a stack are:

- **Push**: Add an element to the top of the stack.
- **Pop**: Remove the top element from the stack.
- **Peek/Top**: View the top element without removing it.

How It Works in a Text Editor

In a stack-based text editor, you typically have two stacks:

1. **Undo Stack**: This stack holds the history of states (or actions) before the current state. Each time you make a change, the current state is pushed onto the undo stack.
2. **Redo Stack**: This stack holds the states after you have undone an action. When you undo an action, the current state is pushed onto the redo stack.

Basic Operations

Here’s how the basic operations work:

1. **Append Text**:
    - Save the current text to the undo stack.
    - Append new text.
    - Clear the redo stack since you can no longer redo after making a new change.
2. **Undo**:
    - Save the current text to the redo stack.
    - Retrieve and restore the previous state from the undo stack.
3. **Redo**:
    - Save the current text to the undo stack.
    - Retrieve and restore the next state from the redo stack.

4. **Save to File**: Save the current text to a file.
5. **Load from File**: Load text from a file and save the current state to the undo stack.

Example

If you append "Hello", then " World", the editor saves each state:

- Initial state: ""
- After "Hello": undo stack has `[""]`, current text is "Hello"
- After " World": undo stack has `["", "Hello"]`, current text is "Hello World"

When you undo:

- Undo: redo stack has `["Hello World"]`, current text is "Hello"
- Undo again: redo stack has `["Hello World", "Hello"]`, current text is ""

When you redo:

- Redo: undo stack has `[""]`, current text is "Hello"
- Redo again: undo stack has `["", "Hello"]`, current text is "Hello World"

Benefits

- **Efficient Undo/Redo**: Managing text changes with stacks makes it easy to implement efficient undo and redo operations.
- **Simple Logic**: The use of stacks simplifies the logic for tracking the history of changes.

A stack-based approach helps maintain a clear and reversible history of text modifications, making it a reliable method for text editors.

Libraries Used:

 #include <iostream>: This includes the standard input-output stream library for using cin, cout, etc.

 #include <stack>: This includes the stack library, which provides the stack data structure.

 #include <fstream>: This includes the file stream library for file I/O operations.

 #include <string>: This includes the string library for using the string class.

 #include <iomanip>: This includes the library for better formatting of output.

 using namespace std;: This statement allows you to use all standard library names without the std:: prefix.

Statements Functions:

 stack<string> undoStack;: Defines a stack to keep track of the states for undo operations.

 stack<string> redoStack;: Defines a stack to keep track of the states for redo operations.

 string currentText;: Stores the current text in the editor.

Concept Used:

The Last In, First Out (LIFO) concept is primarily utilized in your program through the use of stacks for undo and redo operations. Here's how the LIFO principle is applied in the context of your `TextEditor` class:

Undo Operation

When you append text, the current state of `currentText` is pushed onto the `undoStack`. This means the most recent state (the one after the append operation) is at the top of the stack. When you call the `undo` function, it follows these steps:

1. **Push the current state of `currentText` onto the `redoStack`:**

   cpp

   ```
   redoStack.push(currentText);
   ```

   This saves the current state so you can redo the operation if needed.

2. **Retrieve the last state from the `undoStack` and set it as the current state**:

   cpp

   ```
   currentText = undoStack.top();
   undoStack.pop();
   ```

   This restores the most recent previous state, adhering to the LIFO principle where the last state saved (the most recent one before the current state) is the first one to be retrieved.

Redo Operation

Similarly, the redo operation leverages the LIFO concept. When you undo an action and want to redo it, the `redoStack` comes into play:

1. **Push the current state of `currentText` onto the `undoStack`:**

   cpp

   ```
   undoStack.push(currentText);
   ```

   This saves the current state so you can undo the redo if needed.

2. **Retrieve the last undone state from the `redoStack` and set it as the current state**:

   cpp

   ```
   currentText = redoStack.top();
   redoStack.pop();
   ```

   This restores the state that was undone last, again following the LIFO principle where the last undone state is the first to be redone.

Here's a visual summary of how LIFO is applied in both stacks:

Undo Operation:

```
undoStack:  [..., previousStateN-1, previousStateN]
Current Text: previousStateN
```

When `undo()` is called:

```
redoStack: [currentState]
undoStack:  [..., previousStateN-1]
Current Text: previousStateN-1
```

Redo Operation:

```
redoStack:  [previousStateN]
Current Text: previousStateN-1
```

When `redo()` is called:

```
undoStack: [currentState, previousStateN-1, previousStateN]
redoStack: []
Current Text: previousStateN
```