Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

A Project Report on

"Console Based Text Editor with Undo/Redo using Stack"

[Code No.: COMP 202]

(For partial fulfillment of Year II / Semester I in Computer Engineering)

Submitted by

Nawaraj Pathak (036137-24)

Rollno: 65

Submitted to

Mr. Sagar Acharya

Department of Computer Science and Engineering

February 25, 2026

Abstract

This project is a simple console-based text editor developed using C++. The main objective of this project is to implement Undo and Redo operations using the Stack data structure. Two stacks are used to store the previous and undone states of the text. Whenever the user performs an action, the current state is saved, allowing it to be undone or redone later. This project demonstrates the practical use of stack and helps in understanding the Last-In-First-Out (LIFO) principle in real-life applications.

Contents

1

Introduction

1.1 Background

The Undo and Redo Text Editor is a simple console-based application developed using C++. The main purpose of this project is to understand the practical use of the Stack data structure in real-life applications.

In many software applications like text editors, the undo and redo features allow users to reverse or reapply their previous actions. These features are very useful because they help users correct mistakes easily. This project demonstrates how the Stack data structure can be used to implement these operations efficiently.

In this project, two stacks are used:

- One stack for storing previous states (Undo Stack)
- One stack for storing reverted states (Redo Stack)

Whenever the user adds text, the current state is saved in the undo stack. When undo is performed, the previous state is restored. Similarly, redo restores the undone state. Since stack follows the Last-In-First-Out (LIFO) principle, it is perfectly suitable for implementing undo and redo operations.

This project helps in understanding stack operations such as push and pop, and how they can be applied in real-world problem solving.

1.2 Objectives

The main objectives achieved during the implementation of this system are as follows:

- To understand the concept of Stack data structure.
- To implement Undo and Redo operations using stack.
- To learn how stack works using the LIFO (Last-In-First-Out) principle.
- To develop a simple console-based text editor using C++.
- To apply theoretical DSA concepts in a practical project.

1.3 Motivation and Significance

The main motivation behind this project is to understand how data structures are used in real-life applications. Undo and Redo features are commonly used in text editors and many software applications. By developing this project, we can learn how the Stack data structure works in a practical way.

The significance of this project is that it connects theoretical knowledge of stack with real implementation. It helps students understand the importance of LIFO (Last-In-First-Out) principle and how it is applied in real-world systems.

2. Related Works

2.1 Review of Existing Works

Many popular text editing software such as Microsoft Word and other text editors provide Undo and Redo functionality. These applications internally use stack-like mechanisms to store user actions and allow reversing previous steps.

However, these systems are complex and include many advanced features. In this project, a simplified version of Undo and Redo is implemented using stacks to clearly demonstrate the core concept without unnecessary complexity.

3

Design and Implementation

3.1 System Overview

The system is designed as a console-based application. It uses two stacks:

- Undo Stack → Stores previous states of text
- Redo Stack → Stores undone states

When the user adds new text:

1. Current text is pushed into the Undo stack.
2. Redo stack is cleared.
3. New text is appended.

When Undo is selected:

1. Current text is pushed into Redo stack.
2. Last saved state from Undo stack is restored.

When Redo is selected:

1. Current text is pushed into Undo stack.
2. Last saved state from Redo stack is restored.

The program runs in a loop until the user selects exit.

.

3.2 System Requirement Specifications

Hardware Requirements

- Computer or Laptop
- Minimum 4 GB RAM
- Basic keyboard input support

Software Requirements

- Operating System: Windows
- Compiler: GCC (MinGW / MSYS2)
- IDE/Editor: Visual Studio Code
- Programming Language: C++

Chapter 4

Results and Discussion

4.1 Implemented Features

- Add new text
- Undo last action
- Redo last undone action
- Display current text
- Exit program
- Error handling for empty undo/redo operations

4.2 Results and Performance Analysis

The program successfully performs Undo and Redo operations using stack. All stack operations such as push and pop work efficiently.

Time Complexity:

- Push operation → O(1)
- Pop operation → O(1)
- Undo → O(1)
- Redo → O(1)

Since stack operations take constant time, the system performs efficiently even with multiple operations.

4.4 Challenges and Limitations

Challenges Faced

- Handling variable declaration inside switch-case
- Fixing compiler and linking errors
- Managing input using getline() and cin properly

Limitations

- Console-based interface only
- Does not support advanced text editing
- Stores entire string state instead of character-by-character changes
- No file saving feature

4.5 Discussion

This project clearly demonstrates how stack can be used in practical applications. The LIFO principle makes stack suitable for implementing Undo and Redo functionality.

Although the system is simple, it effectively shows how theoretical data structures can solve real problems. With further development, features such as file handling, graphical interface, and advanced editing can be added.

Chapter 5

Conclusion and Future Works

This project clearly demonstrates how stack can be used in practical applications. The LIFO principle makes stack suitable for implementing Undo and Redo functionality.

Although the system is simple, it effectively shows how theoretical data structures can solve real problems. With further development, features such as file handling, graphical interface, and advanced editing can be added.

In this project, two stacks were used to manage text states, one for undo operations and another for redo operations. Whenever the user performed an action, the current state was stored in the undo stack. When undo was selected, the previous state was restored and stored in the redo stack. This mechanism clearly shows how stack helps in handling backtracking operations efficiently.

The project also improved understanding of important programming concepts such as loops, switch-case statements, stack operations (push and pop), and user input handling. The time complexity of stack operations is O(1), which makes the undo and redo operations efficient and fast.

Overall, this project successfully connects theoretical knowledge of data structures with practical implementation and provides a strong foundation for understanding advanced applications of stacks.

5.1 Limitations

Although the project works successfully, it has some limitations:

1. Console-Based Interface

Limitation:
The project is console-based and does not have a graphical user interface (GUI), which makes it less user-friendly.

Future Solution:
In future, a GUI-based version can be developed using frameworks such as Qt or other graphical libraries. This would make the application more interactive and visually appealing.

2. Full Text State Storage

Limitation:
The program stores the entire text string in the stack after every change. This can consume more memory if the text becomes very large.

Future Solution:
Instead of storing the whole string, future versions can store only the changes (like characters added or removed). This will reduce memory usage and improve efficiency.

3. No File Handling

Limitation:
The project does not allow saving text to a file or loading text from a file.

Future Solution:
File handling can be added so that users can save their work and reopen it later. This will make the application more practical.

4. Limited Editing Features

Limitation:
The editor only supports adding text and undo/redo operations. It does not support delete, replace, copy, or paste functions.

Future Solution:
Additional editing features can be implemented to make the application more advanced and similar to real text editors.

5.2 Future Enhancements

The project can be further improved in several ways:

1. Developing a graphical user interface for better usability.
2. Implementing file saving and loading functionality.
3. Optimizing memory usage by storing only changes instead of full text states.
4. Adding more editing features such as delete, replace, and clear text.
5. Implementing character-level undo/redo instead of full string-based undo.
6. Extending the project into a complete mini text editor application.

In the future, this simple project can be expanded into a fully functional text editing software by combining data structures with advanced programming concepts.

References

C++ Programming Language – Bjarne Stroustrup Data Structures and Algorithms using C++ – Mark Allen Weiss

GeeksforGeeks, "Stack Data Structure," Available at: https://www.geeksforgeeks.org

- TutorialsPoint, "Stack in C++," Available at: https://www.tutorialspoint.com
- Class Notes and Lecture Materials on Data Structures

Appendix

main.cpp > main()

```cpp
#include <iostream>
#include <stack>
#include <string>

using namespace std;

int main() {

    stack<string> undoStack;
    stack<string> redoStack;
    string currentText = "";

    int choice;

    while (true) {
        cout << "\n------ TEXT EDITOR ------\n";
        cout << "1. Add Text\n";
        cout << "2. Undo\n";
        cout << "3. Redo\n";
        cout << "4. Display Current Text\n";
        cout << "5. Exit\n";
        cout << "Enter your choice: ";

        cin >> choice;
        cin.ignore();

        switch (choice) {

        case 1: {
            cout << "Enter text to add: ";
            string newText;
            getline(cin, newText);

            undoStack.push(currentText);
            currentText += newText;

            while (!redoStack.empty()) {
                redoStack.pop();
            }
            break;
        }

        case 2:
            if (!undoStack.empty()) {
                redoStack.push(currentText);
                currentText = undoStack.top();
                undoStack.pop();
                cout << "Undo successful.\n";
            } else {
                cout << "Nothing to undo.\n";
            }
            break;

        case 3:
            if (!redoStack.empty()) {
                undoStack.push(currentText);
                currentText = redoStack.top();
                redoStack.pop();
                cout << "Redo successful.\n";
            } else {
                cout << "Nothing to redo.\n";
            }
            break;

        case 4:
            cout << "Current Text: " << currentText << endl;
            break;

        case 5:
            cout << "Exiting program.\n";
            return 0;

        default:
            cout << "Invalid choice.\n";
        }
    }

    return 0;
}
```

Figure 1: Source code

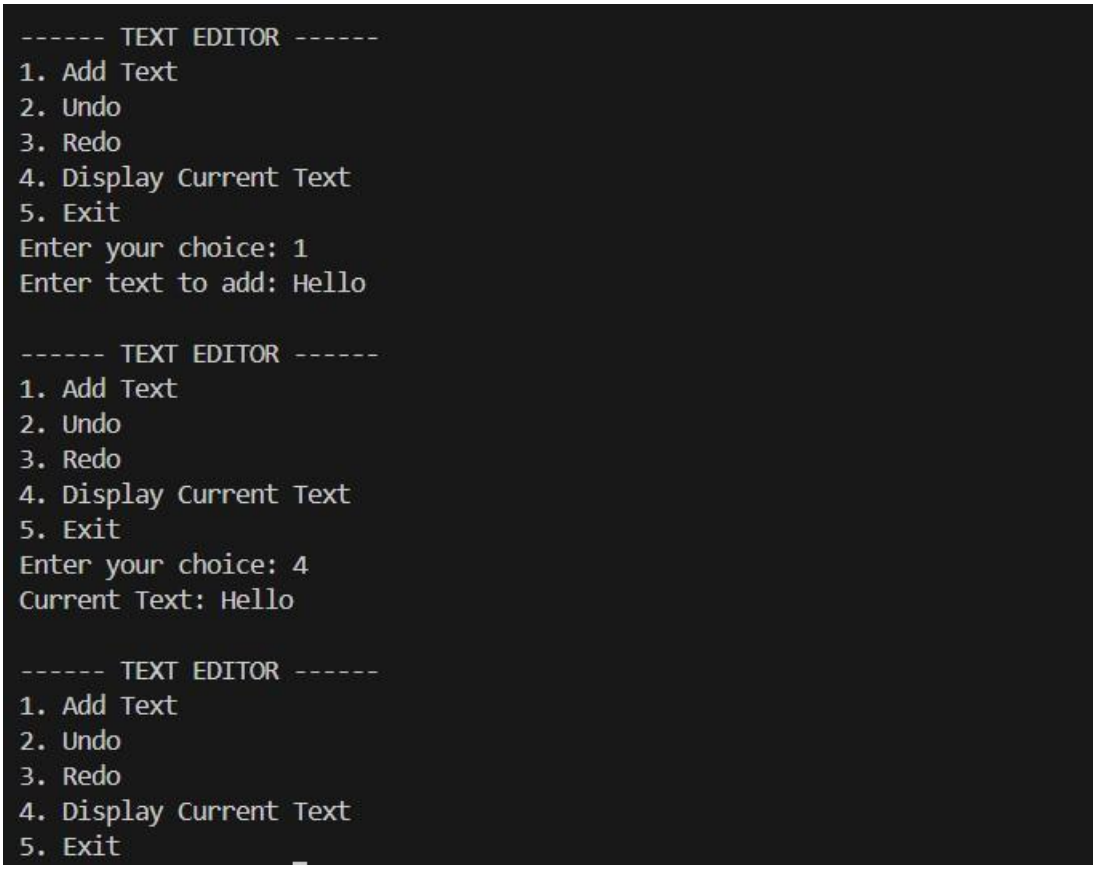

Figure 2: Output