**Name: Sameen Noor**

**Reg No: 2023-BCS-077**

**Project: Stack Based Text Editor**

**Code:**

#include <iostream>

#include <string>

#include <fstream>

#include <stack>

using namespace std;

// Command structure to store undo information

struct UndoCmd {

int lineNumber;

string text; int commandNumber; // 1: Insert, 2: Delete, 3: Replace

};

// Node structure for the linked list

struct Node {

string data;

Node \*next;

Node(const string &data) : data(data), next(NULL) {} // Constructor for initialization

};

// Class for text editor operations using data structures

class TextEditor {

private:

Node \*head;

Node \*tail;

int numOfLines;

stack<UndoCmd> undoStack; // Stack to store undo operations

stack<UndoCmd> redoStack; // Stack to store redo operations

public:

TextEditor() : head(NULL), tail(NULL), numOfLines(0) {}

void run() {

int choice = -1;

while (choice != 0) {

displayHeading();

displayMenu();

cin >> choice;

handleChoice(choice);

}

}

private:

void displayHeading() {

cout << "\n==== STACK-BASED TEXT EDITOR ====\n";

}

void displayMenu() {

cout << "1. Insert text into Line N\n";

cout << "2. Delete line N\n";

cout << "3. Replace text in Line N\n";

cout << "4. Print all lines\n";

cout << "5. Undo\n"; // Undo option before Save to .txt file

cout << "6. Redo\n"; // Redo option before Save to .txt file

cout << "7. Save to .txt file\n"; // Save to file option after Undo and Redo

cout << "8. Clear editor\n";

cout << "0. Exit\n";

cout << "Enter your choice: ";

}

void handleChoice(int choice) {

switch (choice) {

case 1: insertText(); break;

case 2: deleteLine(); break;

case 3: replaceLine(); break;

case 4: printAllLines(); break;

case 5: undo(); break; // Fixed undo

case 6: redo(); break; // Fixed redo

case 7: saveToFile(); break;

case 8: clearEditor(); break;

case 0: cout << "Exiting editor.\n"; break;

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

}

}

void insertText() {

int lineNum;

string text;

cout << "Enter line number: ";

cin >> lineNum;

cout << "Enter text: ";

cin.ignore();

getline(cin, text);

if (lineNum == 1)

addToHead(text);

else if (lineNum > numOfLines)

addToTail(text);

else

insertAtPosition(text, lineNum);

cout << "Data entered successfully in line " << lineNum << ".\n";

}

void deleteLine() {

int lineNum;

cout << "Enter line number to delete: ";

cin >> lineNum;

if (lineNum < 1 || lineNum > numOfLines) {

cout << "Invalid line number.\n";

return;

}

if (lineNum == 1)

deleteHead();

else if (lineNum == numOfLines)

deleteTail();

else

deleteAtPosition(lineNum);

cout << "Data deleted successfully from line " << lineNum << ".\n";

}

void addToHead(const string &text) {

Node \*newNode = new Node(text);

newNode->next = head; // New node points to the current head

head = newNode; // Update head to the new node

if (!tail) tail = head; // If list was empty, update tail as well

numOfLines++;

undoStack.push({1, text, 1}); // Save to undo stack

while (!redoStack.empty()) redoStack.pop(); // Clear redo stack

}

void addToTail(const string &text) {

Node \*newNode = new Node(text);

if (!tail) {

head = tail = newNode;

} else {

tail->next = newNode;

tail = newNode;

}

numOfLines++;

undoStack.push({numOfLines, text, 1});

while (!redoStack.empty()) redoStack.pop(); // Clear redo stack

}

void insertAtPosition(const string &text, int lineNum) {

Node \*newNode = new Node(text);

Node \*prev = head;

for (int i = 1; i < lineNum - 1 && prev; ++i) {

prev = prev->next;

}

newNode->next = prev->next;

prev->next = newNode;

numOfLines++;

undoStack.push({lineNum, text, 1});

while (!redoStack.empty()) redoStack.pop(); // Clear redo stack

}

void deleteHead() {

if (!head) return;

Node \*temp = head;

head = head->next;

undoStack.push({1, temp->data, 2}); // Save to undo stack

if (!head) tail = NULL; // If list is empty, update tail

delete temp;

numOfLines--;

while (!redoStack.empty()) redoStack.pop(); // Clear redo stack

}

void deleteTail() {

if (!head) return;

if (head == tail) { // Only one node

undoStack.push({1, head->data, 2}); // Save to undo stack

delete head;

head = tail = NULL;

} else {

Node \*current = head;

while (current->next != tail) {

current = current->next;

}

undoStack.push({numOfLines, tail->data, 2}); // Save to undo stack

delete tail;

tail = current;

tail->next = NULL;

}

numOfLines--;

while (!redoStack.empty()) redoStack.pop(); // Clear redo stack

}

void deleteAtPosition(int lineNum) {

Node \*prev = head;

for (int i = 1; i < lineNum - 1 && prev; ++i) {

prev = prev->next;

}

Node \*toDelete = prev->next;

undoStack.push({lineNum, toDelete->data, 2}); // Save to undo stack

prev->next = toDelete->next;

if (lineNum == numOfLines) tail = prev; // Update tail if last line is deleted

delete toDelete;

numOfLines--;

while (!redoStack.empty()) redoStack.pop(); // Clear redo stack

}

void replaceLine() {

int lineNum;

string newText;

cout << "Enter line number to replace: ";

cin >> lineNum;

cout << "Enter new text: ";

cin.ignore();

getline(cin, newText);

if (lineNum < 1 || lineNum > numOfLines) {

cout << "Invalid line number.\n";

return;

}

Node \*current = head;

for (int i = 1; i < lineNum && current; ++i) {

current = current->next;

}

undoStack.push({lineNum, current->data, 3}); // Save undo info

current->data = newText;

cout << "Line " << lineNum << " replaced successfully.\n";

while (!redoStack.empty()) redoStack.pop(); // Clear redo stack

}

void printAllLines() {

if (!head) {

cout << "No lines to display.\n";

return;

}

Node \*current = head;

int lineNum = 1;

while (current) {

cout << lineNum++ << ": " << current->data << endl;

current = current->next;

}

}

void saveToFile() {

ofstream outFile("output.txt"); // Saving to output.txt

Node \*current = head;

while (current) {

outFile << current->data << "\n";

current = current->next;

}

cout << "Saved data to 'output.txt' successfully.\n";

}

void undo() {

if (undoStack.empty()) {

cout << "No commands to undo.\n";

return;

}

UndoCmd lastCmd = undoStack.top();

undoStack.pop();

if (lastCmd.commandNumber == 1) { // Undo insert

deleteAtPosition(lastCmd.lineNumber);

} else if (lastCmd.commandNumber == 2) { // Undo delete

insertAtPosition(lastCmd.text, lastCmd.lineNumber);

} else if (lastCmd.commandNumber == 3) { // Undo replace

replaceAtPosition(lastCmd.text, lastCmd.lineNumber);

}

redoStack.push(lastCmd); // Push the undone command to redo stack

}

void redo() {

if (redoStack.empty()) {

cout << "No commands to redo.\n";

return;

}

UndoCmd lastCmd = redoStack.top();

redoStack.pop();

if (lastCmd.commandNumber == 1) { // Redo insert

insertAtPosition(lastCmd.text, lastCmd.lineNumber);

} else if (lastCmd.commandNumber == 2) { // Redo delete

deleteAtPosition(lastCmd.lineNumber);

} else if (lastCmd.commandNumber == 3) { // Redo replace

replaceAtPosition(lastCmd.text, lastCmd.lineNumber);

}

undoStack.push(lastCmd); // Push the redone command back to undo stack

}

void replaceAtPosition(const string &text, int lineNum) {

Node \*current = head;

for (int i = 1; i < lineNum && current; ++i) {

current = current->next;

}

current->data = text;

}

void clearEditor() {

while (head) {

deleteHead();

}

cout << "Editor cleared.\n";

}

};

int main() {

TextEditor editor;

editor.run();

return 0;

}