Code

#include <iostream>

#include <stack>

#include <conio.h>

#include <string>

#include <fstream>

#include <vector>

#include <string>

using namespace std;

string filename1 = "words1.txt";

string filename2 = "words2.txt";

const int ALPHABET\_SIZE = 26;

// TrieNode structure

struct TrieNode {

TrieNode\* children[ALPHABET\_SIZE];

bool isEndOfWord;

};

// Create a new TrieNode

TrieNode\* createNode() {

TrieNode\* newNode = new TrieNode;

newNode->isEndOfWord = false;

for (int i = 0; i < ALPHABET\_SIZE; i++) {

newNode->children[i] = nullptr;

}

return newNode;

}

// Insert a word into the trie

void insertWord(TrieNode\* root, const string& word) {

TrieNode\* node = root;

for (char c : word) {

int index = c - 'a';

if (index < 0 || index >= ALPHABET\_SIZE) {

// Skip invalid characters

continue;

}

if (node->children[index] == nullptr) {

node->children[index] = createNode();

}

node = node->children[index];

}

node->isEndOfWord = true;

}

// Search for a prefix in the trie

bool searchPrefix(TrieNode\* root, const string& prefix) {

TrieNode\* node = root;

for (char c : prefix) {

int index = c - 'a';

if (index < 0 || index >= ALPHABET\_SIZE || node->children[index] == nullptr) {

return false;

}

node = node->children[index];

}

return true;

}

// Get suggestions for a given prefix

void getSuggestions(TrieNode\* root, const string& prefix, string currentWord, vector<string>& suggestions) {

TrieNode\* node = root;

for (char c : prefix) {

int index = c - 'a';

if (index < 0 || index >= ALPHABET\_SIZE || node->children[index] == nullptr) {

return;

}

node = node->children[index];

currentWord += c;

}

if (node->isEndOfWord) {

suggestions.push\_back(currentWord);

}

for (int i = 0; i < ALPHABET\_SIZE; i++) {

if (node->children[i] != nullptr) {

char c = 'a' + i;

getSuggestions(node->children[i], "", currentWord + c, suggestions);

}

}

}

// Clear the console screen

void clearConsole() {

cout << "\033[2J\033[1;1H";

}

// Set the text color to green

void setGreenColor() {

cout << "\033[0;32m";

}

// Reset the text color

void resetColor() {

cout << "\033[0m";

}

struct Node {

char data;

Node\* next;

Node\* prev;

};

Node\* head;

Node\* tail;

class LinkedList {

private:

int size;

public:

LinkedList() {

head = nullptr;

tail = nullptr;

size = 0;

}

Node\* getHead() {

return head;

}

Node\* getTail() {

return tail;

}

void insert(char c, int position) {

if (position < 0 || position > size)

return;

Node\* newNode = new Node();

newNode->data = c;

if (position == 0) {

newNode->next = head;

newNode->prev = nullptr;

if (head)

head->prev = newNode;

else

tail = newNode;

head = newNode;

}

else if (position == size) {

newNode->next = nullptr;

newNode->prev = tail;

if (tail)

tail->next = newNode;

else

head = newNode;

tail = newNode;

}

else {

Node\* current = head;

for (int i = 0; i < position; i++) {

current = current->next;

}

newNode->next = current;

newNode->prev = current->prev;

if (current->prev)

current->prev->next = newNode;

else

head = newNode;

current->prev = newNode;

}

// Increment line count when a newline character is inserted

if (c == '\n')

size++;

size++;

}

void remove(int position) {

if (position < 0 || position >= size)

return;

Node\* current = head;

for (int i = 0; i < position; i++) {

current = current->next;

}

if (current->prev)

current->prev->next = current->next;

else

head = current->next;

if (current->next)

current->next->prev = current->prev;

else

tail = current->prev;

delete current;

size--;

}

void print() {

Node\* current = head;

int count = 0;

while (current) {

if (count >= 60 && current->data != ' ' && current->next && current->next->data != ' ') {

cout << endl;

count = 0;

}

if (current->data == '\n') {

cout << endl;

count = 0;

}

else {

cout << current->data;

count++;

}

current = current->next;

}

cout << endl;

}

int getSize() {

return size;

}

char getChar(int position) {

if (position < 0 || position >= size)

return '\0'; // Return null character if position is out of range

Node\* current = head;

for (int i = 0; i < position; i++) {

current = current->next;

}

return current->data;

}

string getText() {

string text;

Node\* current = head;

while (current) {

text += current->data;

current = current->next;

}

return text;

}

void setText(const string& text) {

clear();

for (char c : text) {

insert(c, size);

}

}

void clear() {

while (head) {

Node\* temp = head;

head = head->next;

delete temp;

}

tail = nullptr;

size = 0;

}

};

class Notepad {

private:

LinkedList lines;

stack<pair<char, int>> undoStack;

stack<pair<char, int>> redoStack;

int cursorPosition;

string addedWord;

int wordStartPosition;

string clipboard;

void pushUndoStack(char c, int position) {

undoStack.push(make\_pair(c, position));

}

void pushRedoStack(char c, int position) {

redoStack.push(make\_pair(c, position));

}

public:

Notepad() {

cursorPosition = 0;

}

void tireloader()

{

TrieNode\* root = createNode();

int asd = 0;

if (asd == 0)

{

// Load words from a file

ifstream inputFile("otfile.txt");

if (!inputFile.is\_open()) {

cout << "Failed to open file." << endl;

return;

}

string word;

while (getline(inputFile, word)) {

insertWord(root, word);

}

inputFile.close();

// Run program until escape is entered

string prefix;

while (true) {

clearConsole();

cout << "Enter a prefix (enter 'escape' to exit): ";

cin >> prefix;

if (prefix == "escape") {

break;

}

bool hasSuggestions = searchPrefix(root, prefix);

if (hasSuggestions) {

vector<string> suggestions;

getSuggestions(root, prefix, prefix, suggestions);

clearConsole();

setGreenColor();

cout << "Suggestions:" << endl;

resetColor();

for (int i = 0; i < suggestions.size(); i++) {

cout << i + 1 << ". " << suggestions[i] << endl;

}

cout << endl << "Enter the number of the word to add or 0 to skip: ";

int choice;

cin >> choice;

if (choice >= 1 && choice <= suggestions.size()) {

string selectedWord = suggestions[choice - 1];

cout << "Selected word: " << selectedWord << endl;

prefix += selectedWord;

}

else {

cout << "Skipped adding a word." << endl;

}

}

else {

clearConsole();

cout << "No suggestions found. Do you want to add this word? (y/n): ";

char choice;

cin >> choice;

if (choice == 'y' || choice == 'Y') {

insertWord(root, prefix);

cout << "Word added successfully." << endl;

}

}

// Wait for user input to continue

cout << endl << "Press Enter to continue...";

cin.ignore();

cin.get();

}

clearConsole();

}

}

void saveWords(const string& filename) {

ofstream file(filename);

if (file.is\_open()) {

file << lines.getText();

file.close();

cout << "Words saved to " << filename << endl;

}

else {

cout << "Unable to open file for saving: " << filename << endl;

}

}

void insert(char c) {

pushUndoStack(c, cursorPosition);

lines.insert(c, lines.getSize()); // Insert at the end of the line

cursorPosition++;

redoStack = stack<pair<char, int>>(); // Clear redo stack

system("cls");

lines.print();

}

void removeLast() {

if (cursorPosition == 0)

return;

char removedChar = lines.getChar(cursorPosition - 1);

pushUndoStack(removedChar, cursorPosition - 1);

lines.remove(cursorPosition - 1);

cursorPosition--;

// If the deleted character was part of a word added by the user, adjust the cursor position

if (cursorPosition >= wordStartPosition && cursorPosition < wordStartPosition + addedWord.length()) {

cursorPosition = wordStartPosition;

}

redoStack = stack<pair<char, int>>(); // Clear redo stack

system("cls");

lines.print();

}

bool canUndo() {

return !undoStack.empty();

}

bool canRedo() {

return !redoStack.empty();

}

void undo() {

if (!canUndo())

return;

pair<char, int> undoData = undoStack.top();

undoStack.pop();

char undoChar = undoData.first;

int undoPosition = undoData.second;

if (undoPosition < cursorPosition)

cursorPosition--; // Decrement cursor position

pushRedoStack(undoChar, undoPosition);

lines.remove(undoPosition);

system("cls");

lines.print();

}

void redo() {

if (!canRedo())

return;

pair<char, int> redoData = redoStack.top();

redoStack.pop();

char redoChar = redoData.first;

int redoPosition = redoData.second;

if (redoPosition <= cursorPosition)

cursorPosition++; // Increment cursor position

pushUndoStack(redoChar, redoPosition);

lines.insert(redoChar, redoPosition);

system("cls");

lines.print();

}

void search(const string& query) {

Node\* current = lines.getHead();

int position = 0;

bool found = false;

while (current) {

if (current->data == query[0]) {

Node\* temp = current;

int i = 0;

while (temp && i < query.length()) {

if (temp->data != query[i])

break;

temp = temp->next;

i++;

}

if (i == query.length()) {

found = true;

break;

}

}

current = current->next;

position++;

}

if (found) {

cursorPosition = position + query.length();

system("cls");

lines.print();

cout << "Found at position: " << position << endl;

}

else {

system("cls");

lines.print();

cout << "Not found." << endl;

}

}

void searchAndReplace(const string& query, const string& replacement) {

Node\* current = lines.getHead();

int position = 0;

bool found = false;

int length = 0; // Variable to store the length of the list

while (current) {

if (current->data == query[0]) {

Node\* temp = current;

int i = 0;

while (temp && i < query.length()) {

if (temp->data != query[i])

break;

temp = temp->next;

i++;

}

if (i == query.length()) {

found = true;

break;

}

}

current = current->next;

position++;

length++; // Increment the length during traversal

}

if (found) {

for (int i = 0; i < query.length(); i++) {

pushUndoStack(lines.getChar(position), position);

lines.remove(position);

}

for (int i = 0; i < replacement.length(); i++) {

pushUndoStack(replacement[i], position + i);

lines.insert(replacement[i], position + i);

}

cursorPosition = position + replacement.length();

redoStack = stack<pair<char, int>>(); // Clear redo stack

system("cls");

lines.print();

cout << "Search and replace completed." << endl;

}

else {

system("cls");

lines.print();

cout << "Query not found." << endl;

}

// Move cursor position to the end of the line

Node\* endNode = lines.getHead();

for (int i = 0; i < length - 1; i++) { // Use the calculated length

endNode = endNode->next;

}

cursorPosition = length - 1;

}

void copyText(int line, int start, int end) {

clipboard.clear(); // Clear the clipboard before copying

Node\* currentLine = lines.getHead();

int currentLineIndex = 0;

int currentIndex = 0;

while (currentLine && currentLineIndex <= line) {

if (currentLineIndex == line) {

Node\* currentChar = currentLine;

int currentCharIndex = 0;

while (currentChar && currentCharIndex <= end) {

if (currentCharIndex >= start) {

clipboard += currentChar->data;

currentIndex++;

}

currentChar = currentChar->next;

currentCharIndex++;

}

break;

}

currentLine = currentLine->next;

currentLineIndex++;

}

cout << "Text copied." << endl;

}

void pasteText(int line, int pos) {

if (clipboard.empty()) {

cout << "Clipboard is empty. Nothing to paste." << endl;

return;

}

int clipboardLength = clipboard.length();

Node\* currentLine = lines.getHead();

int currentLineIndex = 0;

int currentIndex = 0;

while (currentLine && currentLineIndex <= line) {

if (currentLineIndex == line) {

Node\* currentChar = currentLine;

int currentCharIndex = 0;

while (currentChar) {

if (currentCharIndex == pos) {

for (int i = 0; i < clipboardLength; i++) {

lines.insert(clipboard[i], currentIndex);

currentIndex++;

}

}

currentChar = currentChar->next;

currentCharIndex++;

currentIndex++;

}

break;

}

currentLine = currentLine->next;

currentLineIndex++;

}

cout << "Text pasted." << endl;

}

void addWordAtPosition() {

cout << "Enter word to add: ";

string word;

cin >> word;

cout << "Enter position (0 for start, -1 for end): ";

int position;

cin >> position;

if (position == 0) {

// Add at the start

for (int i = word.length() - 1; i >= 0; i--) {

lines.insert(word[i], 0);

}

cursorPosition += word.length();

}

else if (position == -1) {

// Add at the end

int size = lines.getSize();

for (char c : word) {

lines.insert(c, size);

size++;

}

cursorPosition += word.length();

}

else {

// Add at the specified position

Node\* current = lines.getHead();

int currentPosition = 0;

int lineLength = 0;

while (current && currentPosition < position) {

current = current->next;

currentPosition++;

// Increment line length if not at the end of the line

if (current && current->data != '\n') {

lineLength++;

}

}

if (current) {

for (char c : word) {

lines.insert(c, currentPosition);

currentPosition++;

lineLength++;

}

cursorPosition += lineLength + word.length();

}

}

system("cls");

lines.print();

}

void handleKeyPress(char key) {

switch (key) {

case 8: {

// Backspace

removeLast();

break;

}

case 48: {

// 0 to undo

undo();

break;

}

case 49: {

// 1 to redo

redo();

break;

}

case 50: {

// 2 to search

cout << "Enter query to search: ";

string query;

cin >> query;

search(query);

break;

}

case 51: {

// 3 to search and replace

cout << "Enter query to search: ";

string query2;

cin >> query2;

cout << "Enter replacement: ";

string replacement;

cin >> replacement;

searchAndReplace(query2, replacement);

break;

}

case 52: {

// 4 to copy

cout << "Enter 'from' position: ";

int fromPos;

cin >> fromPos;

cout << "Enter 'to' position: ";

int toPos;

cin >> toPos;

copyText(0, fromPos, toPos);

break;

}

case 53: // 5 to paste

{

cout << "Enter position to paste: ";

int pastePos;

cin >> pastePos;

cout << "Enter copied text: ";

string copiedText;

cin.ignore(); // Ignore the newline character left in the input buffer

getline(cin, copiedText);

pasteText(0, pastePos);

break;

}

case 54:

{

addWordAtPosition();

break;

}

case 55: {

// 7 to display the number of characters entered

cout << "Number of characters entered: " << lines.getSize() << endl;

break;

}

case 56: {

// 8 to save words in a file

saveWords(filename1);

break;

}

case 57: {

// 9 to compare the two files

saveWords(filename2);

ifstream file1(filename1);

ifstream file2(filename2);

if (file1.is\_open() && file2.is\_open()) {

string content1((istreambuf\_iterator<char>(file1)), (istreambuf\_iterator<char>()));

string content2((istreambuf\_iterator<char>(file2)), (istreambuf\_iterator<char>()));

if (content1 == content2) {

cout << "Both files are the same." << endl;

}

else {

cout << "Files are different." << endl;

}

file1.close();

file2.close();

}

else {

cout << "Unable to open files for comparison." << endl;

}

break;

}

case 13: {

// Enter key

insert('\n'); // Insert a new line character

break;

}

case 92: {

//trietree

tireloader();

break;

}

default: {

insert(key);

// Call tireloader() outside the switch statement

break;

}

}

}

};

int main() {

Notepad notepad;

char key = NULL;

cout << "Welcome to Notepad!" << endl;

cout << "Press Enter to start." << endl;

\_getch();

do {

system("cls");

notepad.handleKeyPress(key);

key = \_getch();

} while (key != 27); // 27 is the ASCII code for ESC

return 0;

}