Data Structures (Fall 2024)

Assignment # 02: Console-Based Notepad Application

1. Project Overview and Objectives

This report details the design and implementation of a console-based notepad application as required for this assignment. The primary objective of this project was to build a functional text editor that operates entirely within the console, utilizing fundamental data structures to manage text manipulation, cursor movement, and other core editor functionalities.

The application is built around a custom **two-dimensional (2D) doubly linked list** to store and manage text, allowing for dynamic and efficient insertion and deletion of characters at any point in the document. To enhance user experience, the project also incorporates **stack-based undo and redo functionality**, enabling users to revert and reapply their last five word-based actions.

The program adheres strictly to the assignment constraints, avoiding the use of standard arrays, strings, and other STL containers in favor of custom-built data structures. All functionalities, from character input and cursor navigation to file I/O and memory management, have been implemented from the ground up.

2. Core Data Structure: The 2D Linked List

The foundation of the notepad is a 2D doubly linked list, which provides a flexible grid of characters. Each element in this grid is a `Node`.

Node Structure (`class Node`)

Each `Node` in the linked list is designed to hold a single character and maintain connections to its immediate neighbors. The structure is defined as follows:

- **`character`**: Stores the character data.
- **`left`, `right`, up, down**: Pointers to adjacent nodes, forming the 2D grid.
- **`positionX`, `positionY`**: Integer coordinates to keep track of the node's logical position, which is crucial for cursor mapping.
- **`backSlashN`**: A boolean flag to signify if this node represents the start of a new line (created by an Enter key press or word wrap).

Line Management (`class PointersNode`)

To efficiently manage lines and navigate vertically, a secondary linked list of `PointersNode` is used. This structure holds pointers to the start and end of each line in the main 2D text grid. This approach simplifies line-based operations like moving between lines and re-aligning text after modifications.

3. Core Functionalities Implemented

3.1. Text Insertion and Word Wrap

- **Character-by-Character Input**: The application captures keyboard input directly, processing alphabetic characters and special keys. Non-alphabetic characters are ignored as per the requirements.
- **Dynamic Insertion**: When a character is inserted, it is placed into a new `Node` at the cursor's current position. Existing text is not overwritten; instead, the pointers of the surrounding nodes are updated to shift the subsequent text to the right, making space for the new character.
- **Word Wrap Logic**: A key feature is the automatic word wrapping. The `fixAlignment()` function continuously monitors the text. If a word exceeds the defined `MAIN_AREA_WIDTH` (100 characters), the function locates the beginning of that word (the last space character) and moves the entire word and all subsequent text to a new line below. This is achieved by manipulating the `down` and `up` pointers and creating new line entries in the `PointersNode` list.
- **Enter Key Handling**: Pressing `Enter` inserts a new line. If pressed mid-line, the text after the cursor is moved to the new line. This is handled by setting the `backSlashN` flag on a new node, which signals the `fixAlignment()` function to create a line break.

3.2. Text Deletion

- **Backspace Functionality**: The `deleteCharacter()` function handles deletions. When the backspace key is pressed, the `Node` to the left of the cursor is identified and removed.
- **Pointer Realignment**: The `left` and `right` pointers of the nodes adjacent to the deleted node are re-linked to bridge the gap, effectively shifting all subsequent text to the left.
- **Line Deletion**: If a backspace is pressed at the beginning of a line, it effectively merges the current line with the one above it, handled by the `fixAlignment()` and pointer manipulation logic.

3.3. Cursor Navigation

- **`gotoxy(int x, int y)`**: This standard console function is used to physically move the console cursor to a specific (x, y) coordinate on the screen.

- **Logical Cursor (`Node* cursor`)**: Internally, a `Node*` named `cursor` tracks the current editing position within the 2D linked list.
- **Arrow Key Movement**: The `moveUp()`, `moveDown()`, `moveLeft()`, and `moveRight()` methods update the `cursor` pointer by traversing to the corresponding adjacent node (`up`, `down`, `left`, or `right`). After the logical cursor is updated, `gotoxy()` is called with the node's stored `positionX` and `positionY` to synchronize the visual cursor on the console.

3.4. Undo and Redo Functionality

- **Stack Implementation**: The undo/redo feature is implemented using two custom stack-like linked lists (`undoImplementation` and `redoImplementation`).
- **Word-Based Actions**: Actions are tracked on a per-word basis. When a new word is inserted (detected by a space or new line), a reference to its starting position is pushed onto the `undo` stack. When a word is deleted, the word itself is stored and pushed onto the `redo` stack.
- **Limited History**: Both stacks are capped at a maximum of 5 actions. If a sixth action is performed, the oldest action is discarded to maintain the limit.
- **`undo()`**: Pops an action from the `undo` stack. If the action was an insertion, the corresponding word is located in the 2D list and "deleted" by moving it to the `redo` stack.
- **`redo()`**: Pops a word from the `redo` stack and re-inserts it back into the text at its original position. The action is then pushed back onto the `undo` stack.

3.5. File I/O Operations

- **`saveToFile(const string& filename)`**: This function traverses the entire 2D linked list from the head to the tail, line by line, and writes the `character` from each `Node` into the specified output file.
- **`loadFromFile(const string& filename)`**: This function reads a file character by character. For each character read, it calls the `insertCharacter()` method, effectively building the 2D linked list from the file's content.
- **Automatic File Creation**: If a user tries to load a file that does not exist, the program automatically creates an empty file with that name before opening it for editing, preventing errors.

4. User Interface and Window Layout

As per the assignment specification, the console window is partitioned into three distinct areas using simple character-based borders drawn by the `drawLayout()` function:

1. **Main Text Area (60%)**: The primary area for text editing.

2. **Suggestions Area (20%)**: A reserved space at the bottom, prepared for future implementation of word suggestions.
3. **Search Area (20%)**: A reserved space on the right, prepared for future implementation of search functionality.

This layout is established at the start and maintained throughout the application's lifecycle.

5. Memory Management

The application relies heavily on dynamic memory allocation for creating `Node` objects. To prevent memory leaks, a destructor or a dedicated cleanup function would be responsible for traversing the entire 2D linked list and `delete`-ing every node upon program exit. In the provided `CursorMovement.cpp`, explicit cleanup is managed during operations like `deleteCharacter`, ensuring that removed nodes are deallocated immediately. A comprehensive cleanup on exit is a critical final step for robust memory management.

6. Conclusion

This project successfully implements a feature-rich, console-based notepad application using a 2D linked list and other custom data structures. The implementation meets all the core requirements of the assignment, including character-by-character insertion, deletion, word wrapping, undo/redo functionality, and file I/O.

The development process provided deep practical insight into the complexities of managing dynamic, interconnected data structures, handling low-level console I/O, and implementing foundational text editor features from scratch. The resulting application serves as a strong demonstration of the power and flexibility of linked lists in real-world scenarios.