DESIGN DOCUMENT

System Overview

- **Architecture:** The text editor follows a modular architecture, with the main components being the doubly linked list data structure and the text editor operations.
- **Technologies and Tools**: The text editor is implemented in Python, utilizing object-oriented programming concepts and standard libraries.

Data Structure Model

- **Doubly Linked List:** The chosen data structure to store the text content.
- **Structure:** Each character is represented by a node containing the character value and references to the previous and next nodes.

  Class Structure
  - ➔ *Node Class*: Represents a node in the doubly linked list.
  - ➔ *Attributes*: char, prev, next
  - ➔ *Methods*: None (acts as a simple container)
  - ➔ *TextEditor Class*: Implements the text editor functionalities.
  - ➔ *Attributes*: head, tail, cursor_position, capacity, output_texts, remaining_capacity
  - ➔ *Methods*: empty_text_editor(), handle_error(), add_text(text), delete_text(no_of_backspaces), move_left(no_of_arrow_strokes), move_right(no_of_arrow_strokes), print_text(), generate_output_file(output_file_path)

- **Justification:** The doubly linked list provides efficient insertion, deletion, and traversal operations, essential for text editing scenarios.

Operations and Time Complexity

Operation Name: Command	Operation Description	Time Complexity	Explanation
add_text(text): AddText <t>	This operation adds the provided text,<t> to the text editor at the current cursor position. The text is inserted character by character into the doubly linked list.	O(n), where n is the length of the input text	Adding text to the text editor requires iterating over each character of the input text and performing a constant number of operations for each character. Thus, the time complexity is directly proportional to the length of the input text.
delete_text(no_of_backspaces): DeleteText <n>	This operation deletes a specified nth character before the cursor position in the text editor.	O(k), where k is the number of backspaces	Deleting characters before the cursor involves traversing the linked list backward by the specified number of backspaces. Since the number of backspaces determines the number of iterations, the time complexity is directly proportional to the number of backspaces
move_left(no_of_arrow_strokes): MoveLeft <n>	This operation moves the cursor to the left by n number of positions	O(k), where k is the number of arrow strokes	Moving the cursor to the left requires updating the cursor position by subtracting the specified number of arrow strokes. As the number of arrow strokes increases, the time taken for the operation also increases linearly
move_right(no_of_arrow_strokes): MoveRight <n>	This operation moves the cursor to the right by n number of positions	O(k), where k is the number of arrow strokes	Moving the cursor to the right involves updating the cursor position by adding the specified number of arrow strokes. As the number of arrow strokes increases, the time taken for the operation also increases linearly.
print_text(): PrintText	This operation prints the text stored in the doubly linked list	O(n), where n is the length of the text	Printing the text requires traversing the linked list and printing each character. As the length of the text increases, the time taken for the operation also increases linearly
generate_output_file(output_file_path)	Internal operation which generates an output file containing all the texts generated in the text editor for each command	O(m), where m is the total length of the output texts	Generating the output file involves writing each output text to the file. The time taken is directly proportional to the total length of all the output texts combined
empty_text_editor(): EmptyEditor	This operation clears the entire doubly linked list within the same instance of the text editor and thereby sets the cursor position to 0.	O(1)	This function performs a constant number of operations to reset the attributes of the text editor. Regardless of the size of the text or the number of operations performed, the time taken remains constant.
handle_error(error_message)	Internal Operation which handles and prints the given error message.	O(1)	Handling and printing an error message involve a constant number of operations, resulting in constant time complexity.
process_text_editor()	Internal Operation which processes the commands in the interactive mode of the text editor		

process_input_files(input_file_path, output_file_path, text_editor)	Internal Operation which processes the commands from input file and generates output file		
select_capacity()	Internal Operation which allows user to assign the storage capacity of the data structure. Defaults to 40 characters.		
main()	Driver Code		

Implementation Details

- **Node Class Implementation:** Defines the attributes and methods for the Node class.
- **TextEditor Class Implementation:** Implements the text editor functionalities using the doubly linked list data structure.
- **Input Handling:** Provides interactive mode and file input processing.
- **Error Handling:** Properly handles and displays errors during text editor operations.

Testing Strategy

- **Unit Testing:** Conduct thorough unit tests to ensure the correctness of individual components and functions.
- **Integration Testing:** Perform integration testing to validate the interaction between the text editor components and operations.
- **Edge Cases and Error Handling:** Design and execute test cases to verify the handling of boundary cases and error scenarios.

Alternate Modeling Approach and Cost Implications

- Dynamic Array Model: An alternative modeling approach to the text editor using a dynamic array.
- Structure: The text content is stored as a sequence of characters within a resizable array.

Cost Implications:

- Advantages: Efficient append operation with a time complexity of O(1) amortized, compact memory usage.
- Limitations: Costly insertions and deletions in the middle of the array with a time complexity of O(n), potential memory reallocation when the array reaches capacity.
- Suitable Use Cases: Scenarios with frequent append operations at the end of the text content.
- Design Considerations: Implement efficient array resizing and management strategies to minimize reallocation and ensure optimal performance.