VIETNAM NATIONAL UNIVERSITY - HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

DATA STRUCTURES AND ALGORITHMS - CO2003

ASSIGNMENT 1

IMPLEMENT A SIMPLE TEXT BUFFER USING LIST

HO CHI MINH CITY, 06/2025

ASSIGNMENT'S SPECIFICATION

Version 1.0

1 Assignment's outcome

After completing this assignment, students review and make good use of:

- Object-Oriented Programming (OOP)
- List data structures
- Sorting algorithms

2 Introduction

In Assignment 1, students are required to implement a Text Buffer using a Doubly Linked List (DLL) data structure to simulate the operation of a simple text editor.

Using a doubly linked list offers several significant advantages over managing strings with traditional static arrays, especially in its ability to efficiently insert or delete characters at arbitrary positions with lower complexity. This greatly improves performance when working with large documents or performing frequent modifications in the middle of the text.

Through this assignment, students not only practice implementing fundamental data structures, but also develop object-oriented programming (OOP) thinking, and gain a solid understanding of searching and sorting algorithms, as well as managing operation history. These are essential skills that form a foundation for building and developing real-world text processing applications.

3 Description

3.1 Doubly Linked List

A doubly linked list (*Doubly Linked List*, abbreviated as DLL) is a data structure in which each node stores a generic element of type `T`, along with two pointers: `next` (pointing to the next node) and `prev` (pointing to the previous node). DLLs support insertion and deletion at any position with low cost, making them well-suited for text buffer management problems.

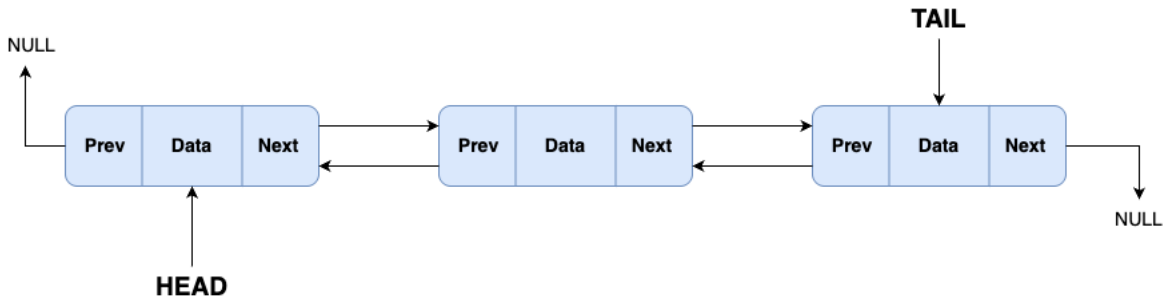

Figure 1: Illustration of a Doubly Linked List

Students are required to propose the member attributes of the `DoublyLinkedList` class. The following methods must be implemented:

- `DoublyLinkedList()`
  - **Function:** Initialize an empty list.
  - **Exception:** None.
  - **Complexity:** $O(1)$.
- `~DoublyLinkedList()`
  - **Function:** Free the memory of all elements to prevent memory leaks.
  - **Exception:** None.
  - **Complexity:** $O(n)$.
- `void insertAtHead(T data)`
  - **Function:** Insert an element containing `data` at the head of the list.
  - **Exception:** None.
  - **Complexity:** $O(1)$.
- `void insertAtTail(T data)`
  - **Function:** Insert an element containing `data` at the tail of the list.
  - **Exception:** None.
  - **Complexity:** $O(1)$.
- `void insertAt(int index, T data)`
  - **Function:** Insert an element containing `data` at the specified `index`.
  - **Exception:** Throw `out_of_range("Index is invalid!")` if the `index` is invalid (less than 0 or greater than the list size).
  - **Complexity:** $O(n)$.
- `void deleteAt(int index)`

  - **Function:** Remove the element at the specified `index`.
  - **Exception:** Throw `out_of_range("Index is invalid!")` if the `index` is invalid.
  - **Complexity:** $O(n)$.

- `T& get(int index)`
  - **Function:** Return a reference to the element at the specified `index` in the list.
  - **Exception:** Throw `out_of_range("Index is invalid!")` if the `index` is invalid.
  - **Complexity:** $O(n)$.

- `int indexOf(T item)`
  - **Function:** Return the index of the first node whose value equals `item`. If not found, return `-1`.
  - **Exception:** None.
  - **Complexity:** $O(n)$.

- `bool contains(T item)`
  - **Function:** Check whether the list contains a node with value equal to `item`. Return `true` if found; otherwise, return `false`.
  - **Exception:** None.
  - **Complexity:** $O(n)$.

- `int size()`
  - **Function:** Return the current number of elements in the list.
  - **Exception:** None.
  - **Complexity:** $O(1)$.

- `void reverse()`
  - **Function:** Reverse the entire list.
  - **Exception:** None.
  - **Complexity:** $O(n)$.

- `string toString(string (*convert2str)(T&) = 0) const;`
  - **Function:** Return a string representation of the list's data. The function pointer `convert2str` is used to convert each element to a string. If no function pointer is provided, students should use the default string representation of the element.
  - **Exception:** None.
  - **Complexity:** $O(n)$.
  - **Output format:** `[<element 1>, <element 2>, <element 3>, ...]`.

Example 3.1

If the list includes: 1, 2, 3, 4, 5, and a function pointer is provided to format integers to strings.

Return value of toString: `[1, 2, 3, 4, 5]`

3.2 TextBuffer and HistoryManager

TextBuffer is a data structure that simulates a basic text editor. This structure supports inserting and deleting characters, moving the cursor, searching, and sorting the text content. TextBuffer stores character data (`char`) along with the current cursor position (`cursor`). The cursor allows users to perform editing operations at any position in the text, similar to real-world text editors.

Overview description:

- Each character in the buffer is stored as type `char`.
- The cursor (`cursor`) is used to determine the position for inserting or deleting a character in the buffer. It can move left, right, or jump to any arbitrary position, starting from 0. The final cursor position is right after the last character of the TextBuffer.
- TextBuffer supports undo/redo functionality, allowing users to revert or reapply editing operations.

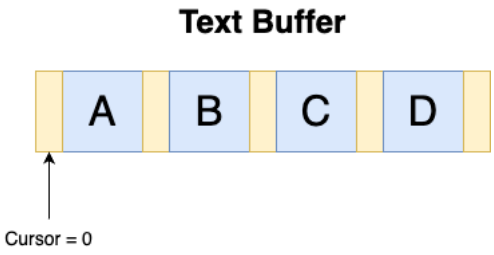

Figure 2: Illustration of TextBuffer with the cursor at the first position

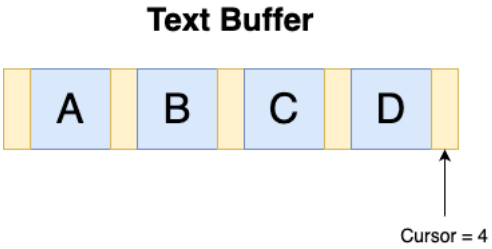

Figure 3: Illustration of TextBuffer with the cursor at the last position

The `TextBuffer` class includes the following methods:

- `TextBuffer()`
  - **Function:** Initialize an empty buffer with the cursor at the first position.
  - **Exception:** None.
  - **Complexity:** $O(1)$.
- `~TextBuffer()`
  - **Function:** Free all allocated memory to prevent memory leaks.
  - **Exception:** None.
  - **Complexity:** $O(n)$.
- `void insert(char c)`
  - **Function:** Insert character `c` immediately before the cursor position. For example, Figures 4 and 5 illustrate the buffer before and after inserting a character.
  - **Exception:** None.
  - **Complexity:** $O(n)$.

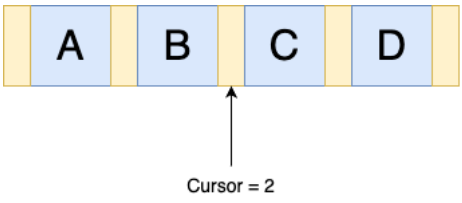

Figure 4: Illustration of TextBuffer before inserting a character

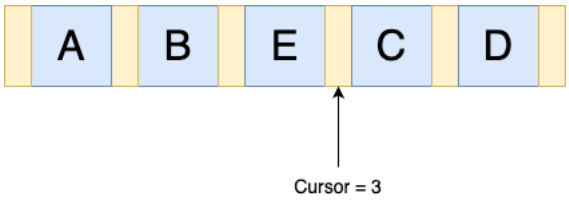

Figure 5: Illustration of TextBuffer after inserting a character

- `void deleteChar()`
  - **Function:** Delete the character immediately before the cursor position. Figures 6 and 7 illustrate the buffer before and after deleting a character.
  - **Exception:** None.
  - **Complexity:** $O(n)$.

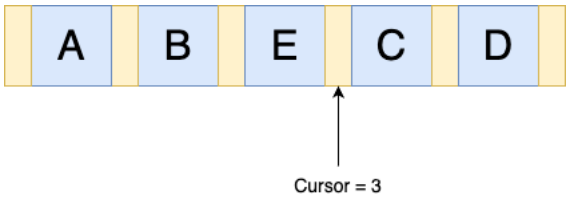

Figure 6: Illustration of TextBuffer before deleting a character

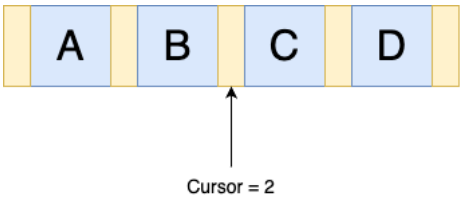

Figure 7: Illustration of TextBuffer after deleting a character

- `void moveCursorLeft()`
  - **Function:** Move the cursor one position to the left.
  - **Exception:** Throw `cursor_error()` if the cursor is already at the beginning.
  - **Complexity:** $O(1)$.
- `void moveCursorRight()`
  - **Function:** Move the cursor one position to the right.
  - **Exception:** Throw `cursor_error()` if the cursor is at the end.
  - **Complexity:** $O(1)$.
- `void moveCursorTo(int index)`

  - **Function:** Move the cursor to the specified `index`.
  - **Exception:** Throw `out_of_range("Index is invalid!")` if `index` is invalid.
  - **Complexity:** $O(1)$.

- `string getContent()`
  - **Function:** Return the entire buffer content as a string.
  - **Exception:** None.
  - **Complexity:** $O(n)$.

- `int getCursorPos()`
  - **Function:** Return the current cursor position.
  - **Exception:** None.
  - **Complexity:** $O(1)$.

- `int findFirstOccurrence(char c)`
  - **Function:** Return the index of the first occurrence of character `c`, or -1 if not found.
  - **Exception:** None.
  - **Complexity:** $O(n)$.

- `int* findAllOccurrences(char c, int &count)`
  - **Function:** Return an array of positions where character `c` occurs, and stores the number of occurrences in `count`.
  - **Exception:** None.
  - **Complexity:** $O(n)$.

- `void sortAscending()`
  - **Function:** Sort the entire buffer following the rule described below. After sorting, the cursor is moved to the first position.
  - Sorting rule: Sort characters in alphabetical order. If both uppercase and lowercase letters exist, uppercase letters are placed before lowercase letters, then followed by the next alphabet letters.

    Example 3.2

    Buffer contains: **abcdfDA**
    After calling `sortAscending`, buffer becomes: **AabcDdf**

  - **Exception:** None.
  - **Complexity:** $O(n \log n)$. Students are required to implement this method using algorithms with the specified complexity.

- `void deleteAllOccurrences(char c)`
  - **Function:** Delete all occurrences of character `c` in the buffer and move the cursor to the beginning of the buffer. If the character to be deleted is not present in the buffer, the cursor position remains unchanged.
  - **Exception:** None.
  - **Complexity:** $O(n)$.
- `void undo()`
  - **Function:** Perform an undo operation, restoring the previous state. Only applicable to certain operations as described below.
  - **Exception:** None.
  - **Complexity:** $O(n)$.
- `void redo()`
  - **Function:** Perform a redo operation (re-execute the last undone action). **After inserting/deleting (`insert`/`delete` action)** a character, **previous redo actions cannot be performed** and must wait for new undo actions to become available.
  - **Exception:** None.
  - **Complexity:** $O(n)$.

  Note: For operations that change the buffer length, the cursor position must be updated accordingly.

Example 3.3

```
Assume the initial buffer is empty:
|
Action 1: insert('A')
A|
Action 2: insert('B')
AB|
Action 3: insert('C')
ABC|
Action 4: moveCursorLeft()
AB|C
Action 5: insert('X')
ABX|C
Action 6: moveCursorRight()
ABXC|
Action 7: deleteChar()
ABX|
Action 8: undo()  (restore character C)
ABXC|
Action 9: undo()  (move cursor to the left)
ABX|C
Action 10: undo()  (remove character X)
AB|C
Action 10: redo()  (re-insert character X)
ABX|C
Action 11: redo()  (move cursor to the right again)
ABXC|
Action 11: redo()  (delete character C again)
ABX|
```

The `HistoryManager` class is a nested class inside the `TextBuffer` class. It is used to manage the history of operations performed on the buffer, and also supports the undo/redo functionality of the buffer.

Main functions:

- Store a list of executed actions, including information about the action name, cursor position, and related character (if any).
- Allow printing the entire action history for verification and illustration purposes.
- Support checking the number of currently stored actions.

Main methods:

- `HistoryManager()`
  - **Function:** Initialize an empty action history list.
  - **Exception:** None.
  - **Complexity:** $O(1)$.
- `~HistoryManager()`
  - **Function:** Free the memory used by the history, deleting all stored actions.
  - **Exception:** None.
  - **Complexity:** $O(n)$.
- `void addAction(const string &actionName, int cursorPos, char c)`
  - **Function:** Record a new action into the history list, storing the action name, the cursor position before the action, and the related character.
  - **Exception:** None.
  - The action format rule is presented below.
  - **Complexity:** $O(1)$.
- `void printHistory()`
  - **Function:** Print the entire list of stored actions to the screen for verification and illustration.
  - **Format:** `[(<action name>, <cursor position>, <char>), (<action name>, <cursor position>, <char>), ...]`
  - **Exception:** None.
  - **Complexity:** $O(n)$.

Example 3.4

With example 3.3, when calling `printHistory()`, the program will print:
`[(insert, 0, A), (insert, 1, B), (insert, 2, C), (move, 3, L), (insert, 2, X), (move, 3, R), (delete, 4, C)]`

- `int size()`

  - **Function:** Return the number of actions currently stored in the history.
  - **Exception:** None.
  - **Complexity:** $O(1)$.

In the buffer, only certain actions can be undone or redone. These are also the actions that need to be recorded in the action history, specifically:

- Character insertion action:
  - **Action name:** `insert`
  - **Related character:** the character being inserted.
- Character deletion action:
  - **Action name:** `delete`
  - **Related character:** the character being deleted.
  - **Note:** Only actions that delete one character are recorded.
- Cursor movement action:
  - **Action name:** `move`
  - **Related character:**
    - If moving left, store character `'L'`.
    - If moving right, store character `'R'`.
    - If jumping to a specific `index`, store character `'J'`.
- Buffer sorting action:
  - **Action name:** `sort`
  - **Related character:** store character `'\0'`.

4 Requirements

To complete this assignment, students need to:

1. Read this entire description file carefully.
2. Download the **initial.zip** file and extract it. After extracting, students will obtain the following files: `main.cpp`, `main.h`, `TextBuffer.h`, `TextBuffer.cpp`, and a folder containing sample outputs. Students only need to submit two files: `TextBuffer.h` and `TextBuffer.cpp`. Therefore, students are not allowed to modify the `main.h` file when testing the program.

3. Use the following command to compile:
   `g++ -o main main.cpp TextBuffer.cpp -I . -std=c++17`
   This command should be used in Command Prompt/Terminal to compile the program. If students use an IDE to run the program, note that they must: add all files to the IDE's project/workspace; modify the build command in the IDE accordingly. IDEs usually provide a Build button and a Run button. When clicking Build, the IDE runs the corresponding compile command, which typically compiles only `main.cpp`. Students must configure the compile command to include `TextBuffer.cpp`, and add the options `-std=c++17` and `-I .`
4. The program will be graded on a Unix-based platform. Students' environments and compilers may differ from the actual grading environment. The submission area on LMS is configured similarly to the grading environment. Students must check their program on the submission page and fix all errors reported by LMS to ensure correct final results.
5. Edit the `TextBuffer.h` and `TextBuffer.cpp` files to complete the assignment, while ensuring the following two requirements:
   - All methods described in this guide must be implemented so that the program can compile successfully. If a method has not yet been implemented, students must provide an empty implementation for that method. Each test case will call certain methods to check their return values.
   - The file `TextBuffer.h` must contain exactly one line `#include "main.h"`, and the file `TextBuffer.cpp` must contain exactly one line `#include "TextBuffer.h"`. Apart from these, no other `#include` statements are allowed in these files.
6. Students are encouraged to write additional supporting classes, methods, and attributes within the classes they are required to implement. However, these additions must not change the requirements of the methods described in the assignment.
7. Students must design and use data structures learned in the course.
8. Students must ensure that all dynamically allocated memory is properly freed when the program terminates.

5 Harmony Questions

The final exam for the course will include several "Harmony" questions related to the content of the Assignment.

Students must complete the Assignment by their own ability. If a student cheats in the

Assignment, they will not be able to answer the Harmony questions and will receive a score of 0 for the Assignment.

Students **must** pay attention to completing the Harmony questions in the final exam. Failing to do so will result in a score of 0 for the Assignment, and the student will fail the course. **No explanations and no exceptions.**

6 Regulations and Handling of Cheating

The Assignment must be done by the student THEMSELVES. A student will be considered cheating if:

- There is an unusual similarity between the source code of submitted projects. In this case, ALL submissions will be considered as cheating. Therefore, students must protect their project source code.
- The student does not understand the source code they have written, except for the parts of code provided in the initialization program. Students can refer to any source of material, but they must ensure they understand the meaning of every line of code they write. If they do not understand the source code from where they referred, the student will be specifically warned NOT to use this code; instead, they should use what has been taught to write the program.
- Submitting someone else's work under their own account.
- Students use AI tools during the Assignment process, resulting in identical source code.

If the student is concluded to be cheating, they will receive a score of 0 for the entire course (not just the assignment).

NO EXPLANATIONS WILL BE ACCEPTED AND THERE WILL BE NO EXCEPTIONS!

After the final submission, some students will be randomly selected for an interview to prove that the submitted project was done by them.

Other regulations:

- All decisions made by the lecturer in charge of the assignment are final decisions.
- Students are not provided with test cases after the grading of their project.
- The content of the Assignment will be harmonized with questions in the exam that has similar content.

7 Changelog

- Example 3.3, update the results of `undo` and `redo` methods.
- Update the `redo` method: Delete character (`delete` action) also cannot perform previous redo actions.
- Update the `deleteAllOccurrences` method: The cursor position is updated as follows:
  - If the character to be deleted exists in the buffer: Move to the beginning of the buffer.
  - If the character to be deleted does not exist in the buffer: Keep the cursor position unchanged.

——————————THE END——————————