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 2

IMPLEMENT A SIMPLE TEXT BUFFER USING ROPE DATA STRUCTURE

HO CHI MINH CITY, 07/2025

ASSIGNMENT'S SPECIFICATION

Version 1.0

1 Assignment's outcome

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

- Be proficient in advanced C++ programming.
- Be able to develop the Rope data structure.
- Use the Rope data structure to implement a simple Text Buffer.

2 Introduction

In Assignment 2, students are required to implement a Text Buffer using the Rope data structure, aiming for a more optimized approach to managing and manipulating long strings that are frequently edited in the middle. Rope is an advanced data structure based on balanced AVL binary trees, designed to efficiently support operations such as insertion, deletion, character access, and string concatenation with low complexity.

Implementing the Text Buffer using Rope helps students understand how a Rope tree is organized, including leaf nodes that store actual data and internal nodes that store structure and weights. Students will also gain a solid understanding of tree navigation and rebalancing when modifications occur. Additionally, they will practice applying techniques such as tree splitting, concatenation, weight management, tree rebalancing, and organizing data efficiently in an object-oriented manner.

Through this assignment, students will not only review and enhance their object-oriented programming skills, but also become familiar with a modern and practically applicable data structure. This is an important step toward developing the ability to design flexible and efficient systems, especially for real-world problems involving complex string or text processing.

3 Description

3.1 Rope Data Structure

Rope is a binary tree-based data structure, designed to efficiently support string manipulation operations such as insertion, deletion, indexing, and concatenation. Rope is particularly suitable for scenarios where the string is frequently modified in the middle, such as in text editors. Students can refer to the theory of Rope at 1.

3.1.1 Rope Tree Organization

Rope is a binary tree in which:

- **Leaf nodes** contain the actual data as a string, with a maximum length defined as a constant in the design. In this assignment, each leaf node stores up to 8 characters (`chunkSize = 8`).
- **Internal nodes** do not store string data but maintain structural information. Each internal node has two children: `left` and `right`, and a `weight` attribute representing the total number of characters in the left subtree.
- Additionally, each node (both internal and leaf) has a `balance` attribute used to maintain AVL tree balance.

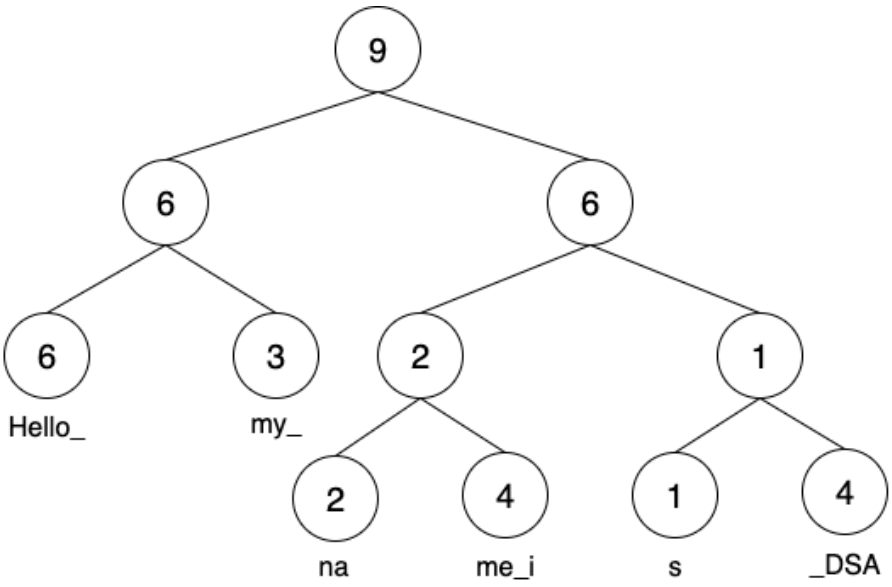

Figure 1: An illustration of a Rope storing the string "Hello_my_name_is_DSA"

3.1.2 Navigation Principle

To access the character at position i in the string, the algorithm traverses the tree from the root. At each node:

- If $i <$ `weight`, go to the left subtree.
- If $i \geq$ `weight`, go to the right subtree with $i' = i -$ `weight`.

Once a leaf node is reached, character access is performed directly on the stored string.

3.1.3 Rope Balancing

To ensure that operations are performed with complexity $O(\log n)$, the Rope must remain height-balanced by following AVL tree principles. After any modification that may change the height, the `balance` of affected nodes must be updated and rebalancing is performed if necessary.

3.1.4 Illustrative Example of Rope

In this example, we perform a few basic operations on a Rope. Assume that each leaf node also contains at most 8 characters.

1. Initialize an empty Rope

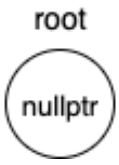

Figure 2: Illustration of an empty Rope

2. Add the string "Hello_World" to the empty Rope
    - **Step 1:** The string "Hello_World" has 11 characters, so we split it into two substrings: "Hello_wo" (8 characters) and "rld" (3 characters).
    - **Step 2:** Create two leaf nodes from the above substrings.
    - **Step 3:** Insert the two leaf nodes into the Rope in the correct order, ensuring AVL balancing after insertion.

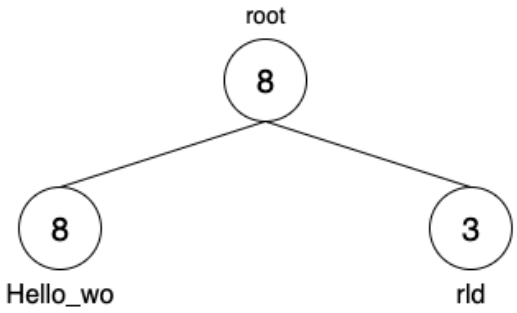

Figure 3: Illustration after adding "Hello_World" to the empty Rope

3. Delete a segment starting at position 5 with length 5 characters
   - **Step 1:** Split the original Rope at position 5. Suppose the left part is **R1** and the right part is **R2**.
   - **Step 2:** Further split **R2** at position 5. Suppose the left part is **R3** and the right part is **R4**.
   - **Step 3:** Concatenate **R1** and **R4** to form the new Rope.

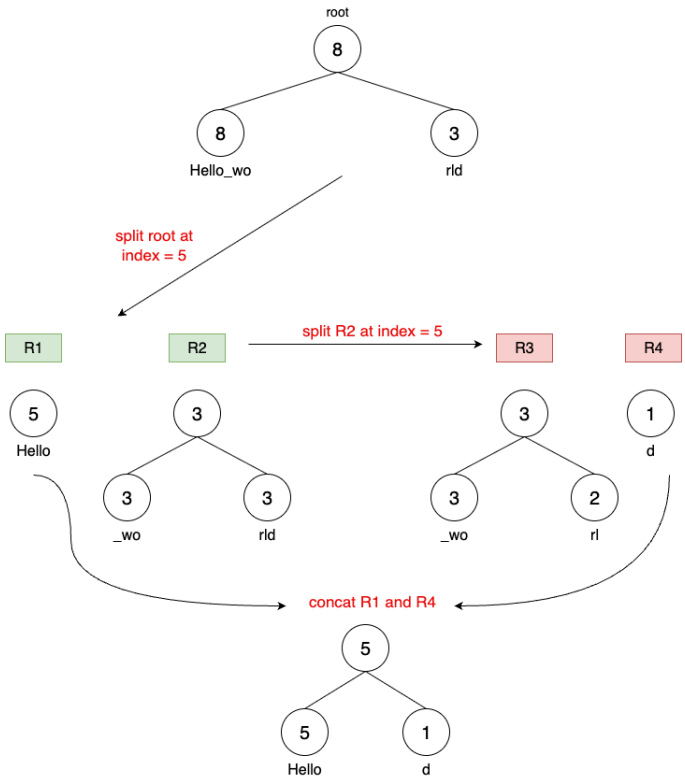

Figure 4: Illustration of deleting 5 characters starting at position 5

4. Insert the string "_DataStructure" into the Rope "Hellod" (result from example 3) at position 3

   - **Step 1:** Split the Rope at position 3. Let **R1** be the left subtree and **R2** be the right.
   - **Step 2:** From "_DataStructure", create two leaf nodes: **Leaf A** ("_DataStr") and **Leaf B** ("ucture").
   - **Step 3:** Concatenate **R1** with **Leaf A** (preserving order) to get **T1**.
   - **Step 4:** Concatenate **T1** with **Leaf B** (preserving order) to get **T2**.
   - **Step 5:** Concatenate **T2** with **R2** (preserving order) to get the final Rope after insertion.

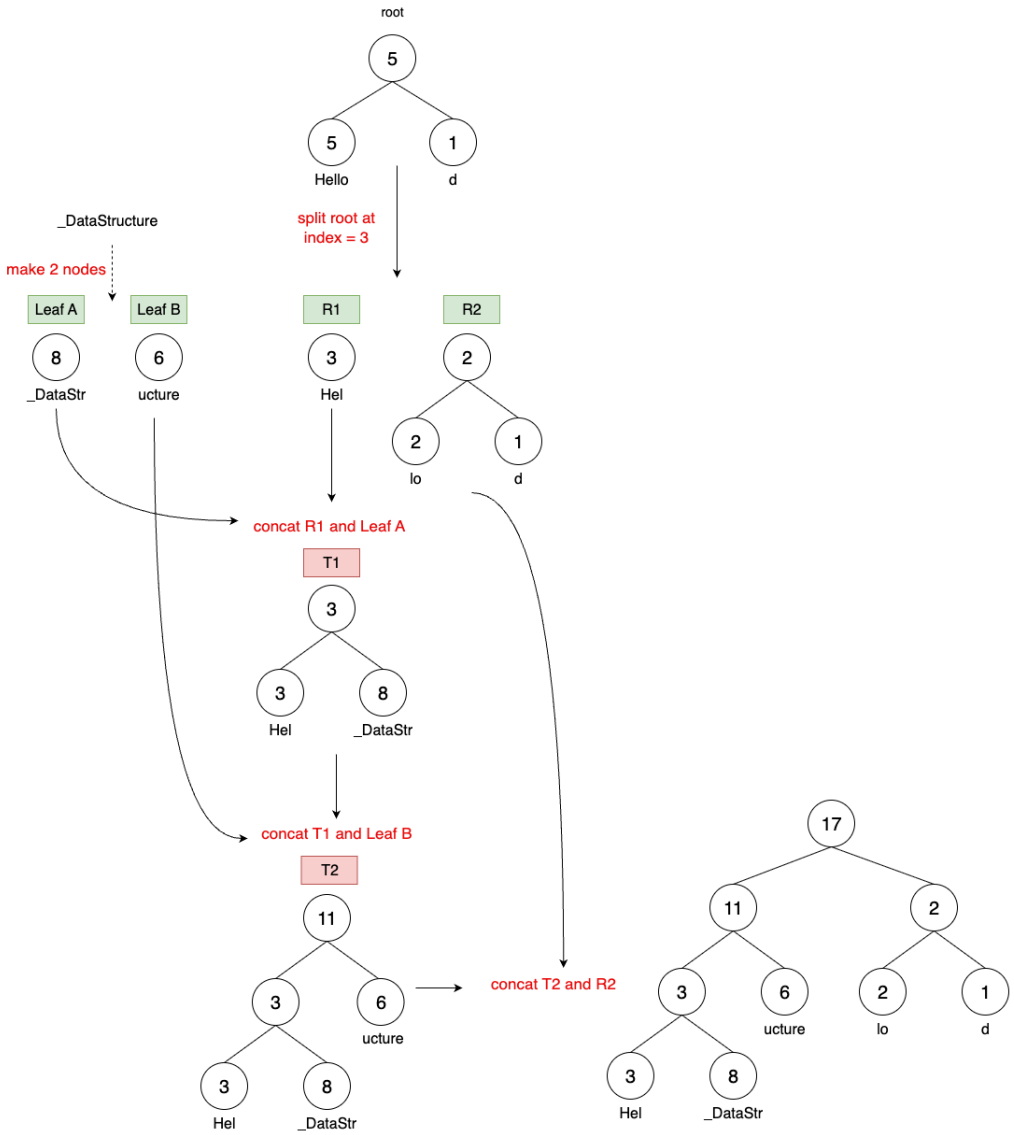

Figure 5: Illustration of inserting "_DataStructure" at position 3

3.1.5 Description of Rope Class Attributes and Methods

3.1.5.a `Node` Class

The `Node` class represents a node in the Rope tree. Each node can be:

- **Leaf node**: contains a string of characters (`data`) with a maximum length of `CHUNK_SIZE`.
- **Internal node**: contains no data (`data` is empty), and only stores structural and auxiliary information.

Main attributes:

- `left`, `right`: pointers to the left and right subtrees.
- `data`: the string stored at the node (only applicable to leaf nodes).
- `weight`: the total number of characters in the left subtree. For leaf nodes, `weight` is the length of the string in that node.
- `height`: the height of the subtree rooted at this node.
- `balance`: AVL balance factor, represented using an `enum` type:

Value	Meaning
`LH`	Left Higher (left subtree is taller)
`EH`	Equal Height (both subtrees have equal height)
`RH`	Right Higher (right subtree is taller)

The method `isLeaf()` returns `true` if the current node is a leaf, otherwise returns `false`.

3.1.5.b `private` Methods

The `private` methods are internal utilities used to implement the core methods of the Rope class.

- `int height(Node* node) const;`
  Returns the height of the given node, used to determine tree balance.
  - **Input:** Any node in the tree.
  - **Output:** Integer height of the node; returns 0 if node is null.
  - **Complexity:** $O(1)$.

- `int getTotalLength(Node* node) const;`
  Computes the total number of characters stored in the subtree rooted at the given node.
  - **Input:** Any node in the tree.
  - **Output:** Total number of characters in the subtree.
  - **Complexity:** $O(\log n)$.
- `void update(Node* node);`
  Check and update the information of the given node: `weight`, `height`, and `balance`.
  - **Input:** The node to update.
  - **Complexity:** $O(\log n)$.
- `Node* rotateLeft(Node* x);`
  Performs a left rotation at node `x`.
  - **Input:** The node to rotate.
  - **Output:** New root node after rotation.
  - **Complexity:** $O(1)$.
- `Node* rotateRight(Node* y);`
  Performs a right rotation at node `y`.
  - **Input:** The node to rotate.
  - **Output:** New root node after rotation.
  - **Complexity:** $O(1)$.
- `Node* rebalance(Node* node);`
  Check the balance status of the given node. If the node becomes unbalanced, perform appropriate rotations to restore the tree to a balanced state.
  - **Input:** The node to check and rebalance.
  - **Output:** New root node after rebalancing.
  - **Complexity:** $O(1)$.
- `void split(Node* node, int index, Node*& outLeft, Node*& outRight);`
  Splits the Rope at the given index into two subtrees. The left subtree contains characters from position 0 to `index - 1`, the right from `index` onward.
  - **Input:** The root node and the split index.
  - **Output:** `outLeft` and `outRight` as the resulting left and right subtrees.
  - **Complexity:** $O(\log n)$.

- `Node* concatNodes(Node* left, Node* right);`
  Concatenate the left and right subtrees into a new complete Rope. A new root node must be created to connect the two subtrees, and the structures of `left` and `right` must not be modified directly (as in the Rope examples).
  - **Input:** Pointers to the left and right subtrees.
  - **Output:** The root of the newly formed Rope.
  - **Complexity:** $O(\log n)$.
- `char charAt(Node* node, int index) const;`
  Returns the character at the specified `index` in the Rope.
  - **Input:** The root node and target index.
  - **Output:** Character at the given position.
  - **Complexity:** $O(\log n)$.
- `string toString(Node* node) const;`
  Return the string data stored in the Rope tree.
  - **Input:** Root node of the Rope.
  - **Output:** Full string stored in the Rope.
  - **Complexity:** $O(n)$.
- `void destroy(Node*& node);`
  Deallocates all dynamically allocated memory for the Rope rooted at `node`.
  - **Input:** Reference to the root node.
  - **Complexity:** $O(n)$.

3.1.5.c `public` Methods

- `Rope();`
  Constructor that initializes an empty Rope.
- `~Rope();`
  Destructor that frees all dynamically allocated memory.
  - **Complexity:** $O(n)$, where n is the number of nodes.
- `int length() const;`
  Returns the total number of characters stored in the Rope.
  - **Output:** Character count.
  - **Complexity:** $O(1)$.

- `bool empty() const;`
  Checks if the Rope is empty.
  - **Output:** `true` if empty, otherwise `false`.
  - **Complexity:** $O(1)$.
- `char charAt(int index) const;`
  Returns the character at the specified index.
  - **Input:** The target index.
  - **Output:** Character at the index.
  - **Complexity:** $O(\log n)$.
  - **Exception:** Throws `out_of_range("Index is invalid!")` if `index` is out of bounds.
- `string substring(int start, int length) const;`
  Returns a substring of length `length` starting at position `start`.
  - **Input:** Start index and length.
  - **Output:** Substring.
  - **Complexity:** $O(\log n)$.
  - **Exception:** Throws `out_of_range("Index is invalid!")` if `start` is out of bounds. Throws `out_of_range("Length is invalid!")` if length exceeds the string's length.
- `void insert(int index, const string& s);`
  Inserts the string `s` at position `index`. Does nothing if `s` is empty.
  - **Input:** Insertion index and string.
  - **Complexity:** $O(\log n)$.
  - **Exception:** Throws `out_of_range("Index is invalid!")` if `index` is out of bounds.
- `void deleteRange(int start, int length);`
  Deletes `length` characters starting from position `start`.
  - **Input:** Start index and number of characters to delete.
  - **Complexity:** $O(\log n)$.
  - **Exception:** Throws `out_of_range("Index is invalid!")` if `start` is out of bounds. Throws `out_of_range("Length is invalid!")` if length exceeds the string's length.
- `string toString() const;`
  Converts the entire Rope into a continuous string.
  - **Output:** The complete string.
  - **Complexity:** $O(n)$.

3.2 RopeTextBuffer - A TextBuffer Implemented with Rope

3.2.1 The RopeTextBuffer Class

The `RopeTextBuffer` class is a text management module built upon the Rope data structure, aiming to efficiently support dynamic text editing operations such as insertion, deletion, replacement, cursor movement, and search.

In addition to efficiently handling complex string operations, this class also supports features like undo, redo. All operations are implemented to ensure correctness, efficiency, and extensibility.

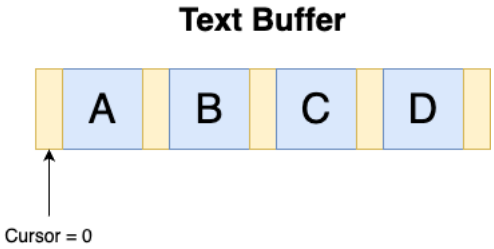

Figure 6: Illustration of a Text Buffer with the cursor at the beginning

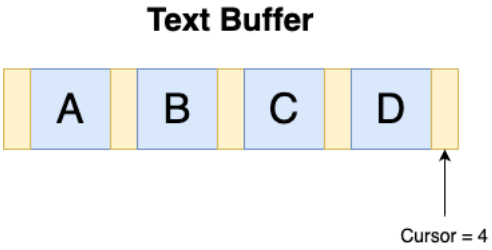

Figure 7: Illustration of a Text Buffer with the cursor at the end

In Figure 6, the position of the character `"A"` is referred to as the position immediately after the cursor. In Figure 7, the position of the character `"D"` is referred to as the position immediately before the cursor.

Main attributes:

- `Rope rope`: A Rope tree used to store the text content.
- `int cursorPos`: Current position of the cursor in the text (measured from the beginning). The cursor starts at 0, and its last position is right after the last character.
- `HistoryManager* history`: Pointer to the operation history manager.

The `HistoryManager` class will be described later.

Public methods:

- `RopeTextBuffer();`
  Initializes an empty text buffer with the cursor positioned at the beginning.
- `~RopeTextBuffer();`
  Frees all memory used by the buffer and its components.
  - **Complexity:** $O(n)$.
- `void insert(const string& s);`
  Inserts a string at the position immediately before the current cursor.
  - **Input:** String `s` to be inserted.
  - **Complexity:** $O(\log n)$.
- `void deleteRange(int length);`
  Deletes a segment of text starting from the position immediately after the cursor, with the given length.
  - **Input:** `length` is the number of characters to delete.
  - **Complexity:** $O(\log n)$.
  - **Exception:** Throws `out_of_range("Length is invalid!")` if `length` exceeds the string's length.
- `void replace(int length, const string& s);`
  Replaces exactly `length` characters starting immediately after the cursor with string `s`, regardless of whether `s` is longer or shorter than `length`. After replacement, the cursor moves to the end of the inserted string.
  - **Input:** Length to replace and the replacement string `s`.
  - **Complexity:** $O(\log n)$.
  - **Exception:** Throws `out_of_range("Length is invalid!")` if `length` exceeds the string's length.
- `void moveCursorTo(int index);`
  Moves the cursor to the specified position.
  - **Input:** `index` is the new cursor position.
  - **Complexity:** $O(1)$.
  - **Exception:** Throws `out_of_range("Index is invalid!")` if `index` exceeds the string's length.
- `void moveCursorLeft();`
  Moves the cursor one character to the left.

  - **Complexity:** $O(1)$.
  - **Exception:** Throws `cursor_error()` if the cursor is already at the beginning.

- `void moveCursorRight();`
  Moves the cursor one character to the right.
  - **Complexity:** $O(1)$.
  - **Exception:** Throws `cursor_error()` if the cursor is already at the end.

- `int getContent() const;`
  Return the entire buffer content as a string.
  - **Output:** The content's output string.
  - **Complexity:** $O(n)$.

- `int getCursorPos() const;`
  Returns the current position of the cursor.
  - **Output:** The current cursor index.
  - **Complexity:** $O(1)$.

- `int findFirst(char c) const;`
  Finds the first occurrence of character `c`.
  - **Input:** Character `c`.
  - **Output:** Index of the first occurrence, or -1 if not found.

- `int* findAll(char c) const;`
  Finds all occurrences of character `c`.
  - **Input:** Character `c`.
  - **Output:** A dynamic array containing all positions of `c`, or `nullptr` if none found.

- `void undo();`
  Performs an undo operation, restoring the previous state. Only applicable to certain actions described later.
  - **Complexity:** $O(\log n)$.

- `void redo();`
  Performs a redo operation (reapplies the last undone action). After inserting or deleting a character, it is **not possible to redo previous actions** unless new undo actions are performed.
  - **Complexity:** $O(\log n)$.

- `void printHistory() const;`
  Prints the operation history.
  - **Complexity:** $O(n)$.
- `void clear();`
  Clears the entire content of the buffer, resetting it to an empty state.
  - **Complexity:** $O(n)$.

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

Example 3.1

```
Assume the initial buffer is empty:
|
Step 1: insert("A")
A|
Step 2: insert("CSE")
ACSE|
Step 3: insert("HCMUT")
ACSEHCMUT|
Step 4: moveCursorLeft()
ACSEHCMU|T
Step 5: insert("123")
ACSEHCMU123|T
Step 6: moveCursorTo(4)
ACSE|HCMU123T
Step 7: deleteRange(3)
ACSE|U123T
Step 8: undo()  (restores the deleted string)
ACSE|HCMU123T
Step 9: undo()  (moves cursor back to position before step 6)
ACSEHCMU123|T
Step 10: undo()  (removes the inserted "123")
ACSEHCMU|T
Step 11: redo()  (reinserts "123")
ACSEHCMU123|T
Step 12: redo() (moves cursor to position 4 again)
ACSE|HCMU123T
Step 13: redo() (deletes 3 characters)
ACSE|U123T
```

3.2.2 The HistoryManager Class

The `HistoryManager` class is responsible for storing and managing the history of text editing operations. This allows users to revert to a previous state (undo) or restore a previously undone operation (redo).

Each operation is stored as an `Action` object containing a short description, cursor positions before and after the operation, and the relevant text content. Students are expected to design and propose appropriate attributes to store action history.

`Action` Structure:

- `string actionName`: the name of the operation.
- `int cursorBefore`: cursor position before the operation.
- `int cursorAfter`: cursor position after the operation.
- `string data`: string related to the operation.

Methods of the `HistoryManager` Class:

- `HistoryManager();`
  Constructor that initializes an empty history manager.
- `~HistoryManager();`
  Destructor that releases all memory used during history management.
  - **Complexity:** $O(n)$
- `void addAction(const Action& a);`
  Saves an action into the history.
  - **Input:** the action `a` to be recorded.
  - **Complexity:** $O(1)$.
- `bool canUndo() const;`
  Checks if an undo operation can be performed.
  - **Output:** `true` if undo is possible; otherwise `false`.
  - **Complexity:** $O(1)$.
- `bool canRedo() const;`
  Checks if a redo operation can be performed.
  - **Output:** `true` if redo is possible; otherwise `false`.
  - **Complexity:** $O(1)$.
- `void printHistory() const;`
  Prints the list of all recorded actions in the operation history.
  - **Format:** `[(<action name>, <cursor before>, <cursor after>, <string data>), (<action name>, <cursor before>, <cursor after>, <string data>), ...]`
  - **Complexity:** $O(n)$

Example 3.2

For example 3.1, calling `printHistory()` will output:

```
[(insert, 0, 1, A), (insert, 1, 4, CSE), (insert, 4, 9, HCMUT),
(move, 9, 8, L), (insert, 8, 11, 123), (move, 11, 4, J), (delete, 4,
4, HCM)]
```

In the buffer, only a specific set of operations are recorded in history and support undo/redo functionality. These operations include:

- Character insertion:
  - **Action name:** `insert`
  - **Relevant string:** the inserted characters.
- Character deletion:
  - **Action name:** `delete`
  - **Relevant string:** the deleted characters.
- Cursor movement:
  - **Action name:** `move`
  - **Relevant string:**
    - If moved left, store character `"L"`.
    - If moved right, store character `"R"`.
    - If jumped to a specific `index`, store character `"J"`.
- String replacement:
  - **Action name:** `replace`
  - **Relevant string:** the string **being replaced**.

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`, `RopeTextBuffer.h` and `RopeTextBuffer.cpp`. Students only need to submit two files: `RopeTextBuffer.h` and `RopeTextBuffer.cpp`. Therefore, students are not allowed to modify the `main.h` file when testing the program.

3. In this assignment, the use of built-in list libraries is not allowed. If students need to use list data structures, they may either reuse the `DoublyLinkedList` class from Assignment 1 or define their own.
4. Use the following command to compile:
   `g++ -o main main.cpp RopeTextBuffer.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 `RopeTextBuffer.cpp`, and add the options `-std=c++17` and `-I .`
5. 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.
6. Edit the `RopeTextBuffer.h` and `RopeTextBuffer.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 `RopeTextBuffer.h` must contain exactly one line `#include "main.h"`, and the file `RopeTextBuffer.cpp` must contain exactly one line `#include "RopeTextBuffer.h"`. Apart from these, no other `#include` statements are allowed in these files.
7. 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.
8. Students must design and use data structures learned in the course.
9. 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

- Add exceptions for `charAt`, `deleteRange` and `substring` methods.

——————————THE END——————————