ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC VÀ KỸ THUẬT MÁY TÍNH

CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT - CO2003

BÀI TẬP LỚN 2

HIỆN THỰC TEXT BUFFER ĐƠN GIẢN SỬ DỤNG CẤU TRÚC DỮ LIỆU ROPE

TP. HỒ CHÍ MINH, THÁNG 07/2025

ĐẶC TẢ BÀI TẬP LỚN

Phiên bản 1.0

1 Chuẩn đầu ra

Sau khi hoàn thành bài tập lớn này, sinh viên ôn lại và sử dụng thành thục:

- Sử dụng được ngôn ngữ lập trình C++ ở mức nâng cao.
- Phát triển được cấu trúc dữ liệu Rope.
- Sử dụng cấu trúc dữ liệu Rope để phát triển một Text Buffer đơn giản.

2 Dẫn nhập

Trong Bài tập lớn 2, sinh viên được yêu cầu hiện thực Text Buffer sử dụng cấu trúc dữ liệu cây Rope, nhằm tiếp cận một cách tối ưu hơn để quản lý và thao tác trên chuỗi văn bản có độ dài lớn và thường xuyên bị chỉnh sửa ở giữa. Rope là một cấu trúc dữ liệu nâng cao dựa trên cây nhị phân cân bằng AVL, được thiết kế để hỗ trợ hiệu quả các thao tác như chèn, xoá, truy xuất ký tự và nối chuỗi với độ phức tạp thấp.

Việc hiện thực Text Buffer bằng Rope giúp sinh viên hiểu rõ cách tổ chức cây Rope, bao gồm các nút lá lưu dữ liệu thực tế và các nút trong lưu cấu trúc cùng trọng số, cũng như nắm vững cơ chế điều hướng và cân bằng cây khi có thay đổi. Bên cạnh đó, sinh viên sẽ rèn luyện khả năng áp dụng các kỹ thuật như tách cây, nối cây, kiểm soát trọng số, cân bằng lại cây, và tổ chức dữ liệu hiệu quả theo định hướng đối tượng.

Thông qua bài tập này, sinh viên không chỉ ôn lại và nâng cao kỹ năng lập trình hướng đối tượng, mà còn được tiếp cận với mô hình cấu trúc dữ liệu hiện đại và có tính ứng dụng thực tiễn cao. Đây là bước phát triển quan trọng giúp sinh viên làm chủ tư duy thiết kế hệ thống linh hoạt và hiệu quả, đặc biệt trong các bài toán xử lý văn bản hoặc chuỗi phức tạp trong thực tế.

3 Mô tả

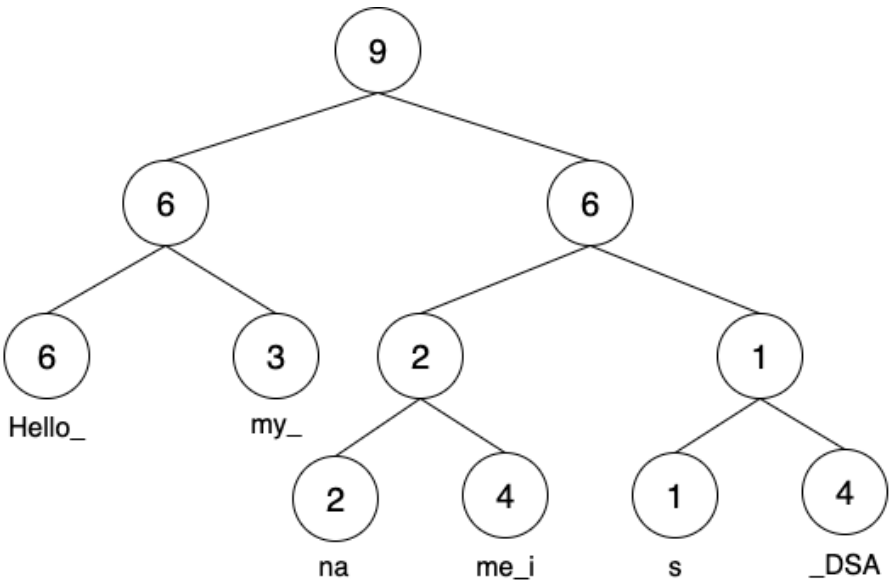
3.1 Cấu trúc dữ liệu Rope

Rope là một cấu trúc dữ liệu dạng cây nhị phân, được thiết kế để hỗ trợ hiệu quả các thao tác xử lý chuỗi như chèn, xoá, truy cập theo vị trí, nối chuỗi. Rope đặc biệt phù hợp với các tình huống trong đó chuỗi được chỉnh sửa thường xuyên tại vị trí giữa, ví dụ như trình soạn thảo văn bản. Sinh viên có thể tham khảo thêm lý thuyết về Rope 1

3.1.1 Tổ chức cây Rope

Rope là một cây nhị phân, trong đó:

- Các **nút lá (leaf nodes)** chứa dữ liệu thực tế dưới dạng chuỗi ký tự, với độ dài tối đa là một hằng số được cố định trong thiết kế. Trong bài tập này, mỗi nút lá chứa tối đa 8 ký tự (`chunkSize = 8`).
- Các **nút nội (internal nodes)** không chứa dữ liệu chuỗi, mà chỉ lưu thông tin về cấu trúc cây. Mỗi nút nội có 2 con `left` và `right`, và một thuộc tính `weight` biểu diễn tổng số ký tự có trong cây con trái.
- Ngoài ra, mỗi node (kể cả lá và trong) lưu thuộc tính `balance` để hỗ trợ cân bằng cây theo nguyên lý AVL.



Hình 1: Minh hoạ một Rope lưu chuỗi "Hello_my_name_is_DSA"

3.1.2 Nguyên tắc điều hướng

Khi cần truy cập ký tự tại vị trí i bất kỳ trong chuỗi, thuật toán duyệt từ nút gốc. Tại mỗi nút:

- Nếu $i <$ `weight`, ta đi sang cây con trái.
- Nếu $i \geq$ `weight`, ta đi sang cây con phải với $i' = i -$ `weight`.

Khi gặp nút lá, việc truy cập ký tự là truy cập trực tiếp trong chuỗi lưu tại lá.

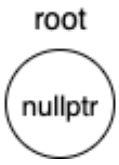
3.1.3 Cân bằng Rope

Để đảm bảo các thao tác có độ phức tạp $O(\log n)$, cây Rope được duy trì ở trạng thái cân bằng chiều cao nhờ sử dụng nguyên lý của cây AVL. Sau mỗi thao tác có thể làm thay đổi chiều cao của cây, phải đảm bảo hệ số cân bằng của các nút được cập nhật và tiến hành cân bằng lại nếu xảy ra tình trạng bị lệch.

3.1.4 Ví dụ minh hoạ về Rope

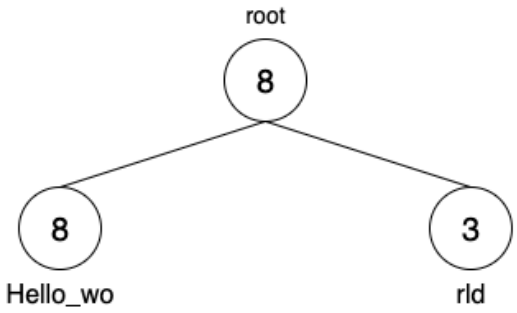
Trong ví dụ này, chúng ta sẽ thực hiện một số thao tác đơn giản trên Rope. Giả sử trong Rope nãy, mỗi nút lá cũng chứa tối đa 8 ký tự.

1. Khởi tạo một Rope rỗng



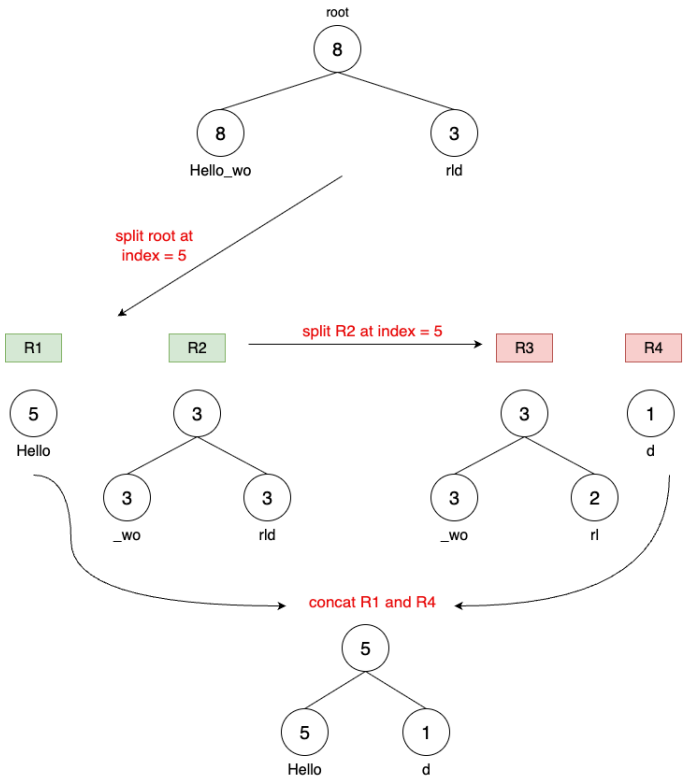
Hình 2: Minh hoạ một Rope rỗng

2. Thêm một chuỗi "Hello_World" vào Rope rỗng
    - **Bước 1:** Chuỗi "Hello_World" có 11 ký tự, chúng ta cần tách chuỗi ra làm 2 chuỗi con: "Hello_wo" (8 ký tự) và "rld" (3 ký tự).
    - **Bước 2:** Tạo 2 nút lá từ 2 chuỗi trên.
    - **Bước 3:** Lần lượt thêm 2 nút lá trên vào Rope. Lưu ý cần đảm bảo đúng thứ tự của chuỗi và nguyên lý cân bằng của cây AVL sau khi thêm.



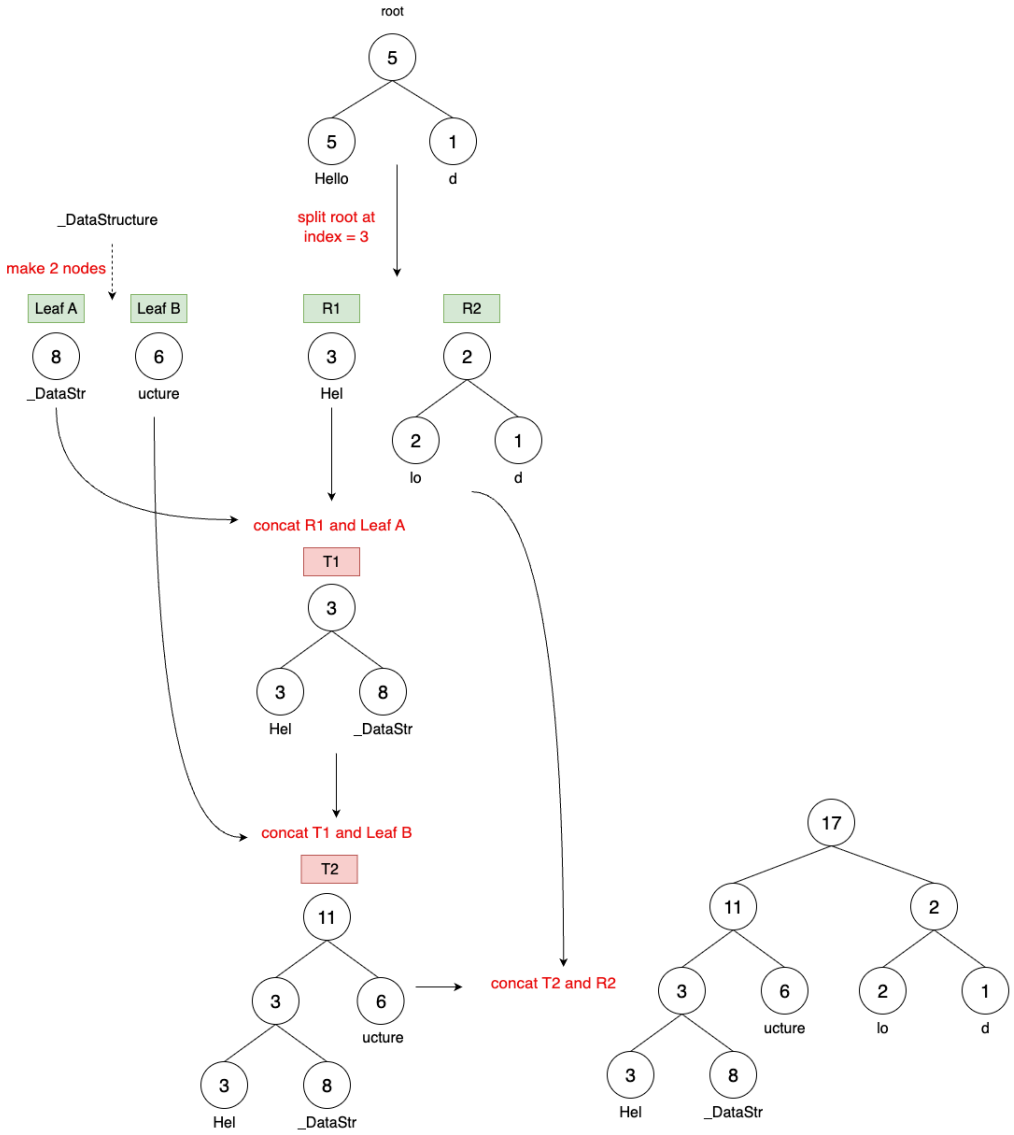
Hình 3: Minh hoạ sau khi thêm chuỗi "Hello_World" vào Rope rỗng

3. Xoá một đoạn trong Rope trên, bắt đầu từ vị trí thứ 5, xoá 5 ký tự.
   - **Bước 1:** Tách Rope ban đầu thành 2 Rope con ở vị trí 5. Giả sử nhận được cây bên trái là **R1** và cây bên phải là **R2**.
   - **Bước 2:** Tách tiếp cây **R2** ban đầu thành 2 Rope con ở vị trí 5. Giả sử nhận được cây bên trái là **R3** và cây bên phải là **R4**.
   - **Bước 3:** Nối Rope bên trái ở bước 1 (**R1**) với Rope bên phải ở bước 2 (**R4**).



Hình 4: Minh hoạ xoá một đoạn trên Rope, bắt đầu từ vị trí 5, xoá 5 ký tự

4. Chèn chuỗi "_DataStructure" vào Rope "Hellod" (kết quả ở ví dụ 3) tại vị trí 3.
    - **Bước 1:** Tách Rope ban đầu thành 2 rope con ở vị trí 3. Giả sử nhận được cây bên trái là **R1** và cây bên phải là **R2**.
    - **Bước 2:** Từ chuỗi "_DataStructure", tạo ra 2 nút lá: nút **Leaf A** (chứa chuỗi "_DataStr") và **Leaf B** (chứa chuỗi "ucture").
    - **Bước 3:** Nối cây **R1** với nút lá **Leaf A** (theo đúng thứ tự). Giả sử nhận được cây **T1**.
    - **Bước 4:** Nối cây **T1** với nút lá **Leaf B** (theo đúng thứ tự). Giả sử nhận được cây **T2**.
    - **Bước 5:** Nối cây **T2** với cây **R2** (theo đúng thứ tự). Kết quả nhận được là Rope sau khi thực hiện chèn chuỗi "_DataStructure" vào vị trí 3.



Hình 5: Minh hoạ chèn chuỗi "_DataStructure" ở vị trí 3

3.1.5 Mô tả các thuộc tính và phương thức của lớp Rope

3.1.5.a Lớp `Node`

Lớp `Node` mô tả một nút trong cây Rope. Mỗi node có thể là:

- **Nút lá (leaf node)**: chứa một chuỗi ký tự (`data`) có độ dài tối đa `CHUNK_SIZE`.
- **Nút trong (internal node)**: không chứa dữ liệu (`data` rỗng), chỉ lưu trữ cấu trúc cây và thông tin phụ trợ.

Các thuộc tính chính:

- `left`, `right`: con trỏ tới cây con trái và phải.
- `data`: chuỗi ký tự tại node (chỉ dùng ở nút lá).
- `weight`: tổng số ký tự trong cây con bên trái. Nếu trong nút lá, `weight` lưu tổng số ký tự của chuỗi lưu tại nút lá đó.
- `height`: chiều cao của cây tại node hiện tại.
- `balance`: hệ số cân bằng AVL, được biểu diễn bằng kiểu `enum`:

Giá trị	Ý nghĩa
`LH`	Left Higher (Cây con trái cao hơn)
`EH`	Equal Height (Hai cây con bằng nhau)
`RH`	Right Higher (Cây con phải cao hơn)

Phương thức `isLeaf()` trả về `true` nếu node hiện tại là nút lá, ngược lại thì trả về `false`.

3.1.5.b Các phương thức `private`

Các phương thức `private` được sử dụng bên trong lớp Rope, hỗ trợ cho việc hiện thực các phương thức chính của lớp Rope.

- `int height(Node* node) const;`
  Trả về chiều cao của nút được truyền vào, giúp xác định tính cân bằng của cây tại node đó.
  - **Đầu vào:** Một node bất kỳ trong cây.
  - **Đầu ra:** Số nguyên biểu thị chiều cao của node. Trả về 0 nếu node rỗng.
  - **Độ phức tạp:** $O(1)$.
- `int getTotalLength(Node* node) const;`
  Tính tổng số ký tự được lưu trong cây con có gốc tại node đó.
  - **Đầu vào:** Một node bất kỳ trong cây.
  - **Đầu ra:** Tổng số ký tự của toàn bộ cây con gốc tại node đó.
  - **Độ phức tạp:** $O(\log n)$
- `void update(Node* node);`
  Kiểm tra và cập nhật lại các thông tin: `weight`, `height` và `balance` của node được truyền vào.
  - **Đầu vào:** Một node cần cập nhật.
  - **Độ phức tạp:** $O(\log n)$

- `Node* rotateLeft(Node* x);`
  Thực hiện phép xoay trái tại node `x`
  - **Đầu vào:** Node cần được xoay trái.
  - **Đầu ra:** Node mới làm gốc sau khi xoay trái.
  - **Độ phức tạp:** $O(1)$.
- `Node* rotateRight(Node* y);`
  Thực hiện phép xoay phải tại node `y`
  - **Đầu vào:** Node cần được xoay phải.
  - **Đầu ra:** Node mới làm gốc sau khi xoay phải.
  - **Độ phức tạp:** $O(1)$.
- `Node* rebalance(Node* node);`
  Kiểm tra tính cân bằng của node được truyền vào. Nếu xảy ra hiện tượng bị mất cân bằng, cần phải thực hiện các phép xoay để đưa cây về trạng thái cân bằng.
  - **Đầu vào:** Một node cần được kiểm tra tính chất cân bằng.
  - **Đầu ra:** Node mới làm gốc sau khi cân bằng.
  - **Độ phức tạp:** $O(1)$
- `void split(Node* node, int index, Node*& outLeft, Node*& outRight);`
  Tách cây tại `node` thành hai cây con dựa vào vị trí chỉ số của ký tự. Cây con bên trái chứa chuỗi từ đầu đến ký tự thứ `index - 1`, cây con bên phải chứa chuỗi từ ký tự thứ `index` đến hết.
  - **Đầu vào:** Một node gốc và vị trí tách (index).
  - **Đầu ra:** `outLeft` để lưu cây con bên trái và `outRight` để lưu cây con bên phải.
  - **Độ phức tạp:** $O(\log n)$.
- `Node* concatNodes(Node* left, Node* right);`
  Nối hai cây con trái và phải thành một cây Rope hoàn chỉnh mới. Phải tạo một nút gốc mới để nối của hai cây con này, không được chỉnh sửa trực tiếp cấu trúc của `left` hay `right` (tương tự các ví dụ về Rope)
  - **Đầu vào:** con trỏ đến 2 cây cần nối.
  - **Đầu ra:** Gốc của cây mới được tạo thành.
  - **Độ phức tạp:** $O(\log n)$
- `char charAt(Node* node, int index) const;`
  Trả về ký tự tại vị trí `index` trong cây Rope.
  - **Đầu vào:** Node gốc và chỉ số cần truy cập.

  - **Đầu ra:** Ký tự tại vị trí tương ứng.
  - **Độ phức tạp:** $O(\log n)$.

- `string toString(Node* node) const;`
  Trả về chuỗi dữ liệu được lưu trong cây Rope.
  - **Đầu vào:** Node gốc của cây.
  - **Đầu ra:** Chuỗi chữ liệu được lưu trong cây.
  - **Độ phức tạp:** $O(n)$.

- `void destroy(Node*& node);`
  Giải phóng toàn bộ bộ nhớ được cấp phát cho cây có gốc tại `node`.
  - **Đầu vào:** Một con trỏ tham chiếu tới node cần huỷ.
  - **Độ phức tạp:** $O(n)$

3.1.5.c Các phương thức `public`

- `Rope();`
  Constructor của lớp Rope, khởi tạo 1 Rope rỗng.
- `~Rope();`
  Destructor của lớp Rope, giải phóng hết tất cả vùng nhớ được cấp phát động.
  - **Độ phức tạp:** $O(n)$

- `int length() const;`
  Trả về tổng số ký tự đang được lưu trong Rope.
  - **Đầu ra:** số lượng ký tự của chuỗi trong Rope.
  - **Độ phức tạp:** $O(1)$.

- `bool empty() const;`
  Kiểm tra Rope có rỗng hay không.
  - **Đầu ra:** `true` nếu rỗng, ngược lại `false`.
  - **Độ phức tạp:** $O(1)$.

- `char charAt(int index) const;`
  Trả về ký tự tại vị trí `index` trong chuỗi.
  - **Đầu vào:** `index` là chỉ số.
  - **Đầu ra:** Ký tự tại vị trí tương ứng.
  - **Độ phức tạp:** $O(\log n)$.
  - **Ngoại lệ:** Ném `out_of_range("Index is invalid!")` nếu `index` không hợp lệ.

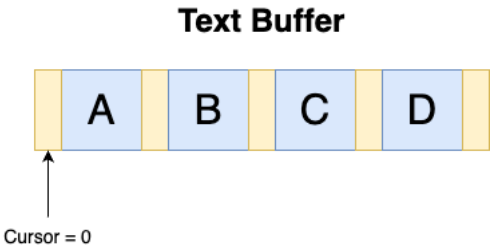
- `string substring(int start, int length) const;`
  Trả về chuỗi con gồm `length` ký tự bắt đầu từ vị trí `start`.
  - **Đầu vào:** vị trí bắt đầu `start` và chiều dài của chuỗi con cần lấy `length`.
  - **Đầu ra:** Chuỗi con.
  - **Độ phức tạp:** $O(\log n)$
  - **Ngoại lệ:** Ném ra `out_of_range("Index is invalid!")` nếu `start` không hợp lệ. Ném ra `out_of_range("Length is invalid!")` nếu `length` vượt quá giới hạn chuỗi.
- `void insert(int index, const string& s);`
  Chèn chuỗi `s` vào vị trí `index`. Nếu chuỗi rỗng thì không thực hiện gì cả.
  - **Đầu vào:** Vị trí chèn và chuỗi cần chèn.
  - **Độ phức tạp:** $O(\log n)$.
  - **Ngoại lệ:** Ném `out_of_range("Index is invalid!")` nếu `index` vượt quá giới hạn của chuỗi.
- `void deleteRange(int start, int length);`
  Xoá `length` ký tự bắt đầu từ vị trí `start`.
  - **Đầu vào:** vị trí bắt đầu `start` và chiều dài cần xoá `len`.
  - **Độ phức tạp:** $O(\log n)$.
  - **Ngoại lệ:** Ném ra `out_of_range("Index is invalid!")` nếu `start` không hợp lệ. Ném ra `out_of_range("Length is invalid!")` nếu `length` vượt quá giới hạn chuỗi.
- `string toString() const;`
  Chuyển toàn bộ Rope thành một chuỗi liên tục.
  - **Đầu ra:** Chuỗi hoàn chỉnh.
  - **Độ phức tạp:** $O(n)$.

3.2 RopeTextBuffer - TextBuffer hiện thực bằng Rope

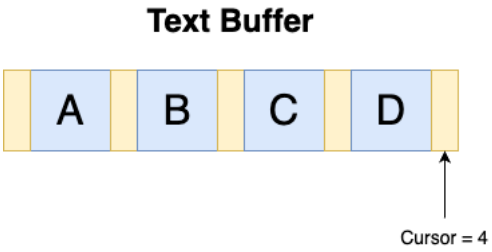
3.2.1 Lớp RopeTextBuffer

Lớp `RopeTextBuffer` là một trình quản lý văn bản được xây dựng dựa trên cấu trúc dữ liệu Rope, nhằm hỗ trợ hiệu quả các thao tác chỉnh sửa văn bản động như chèn, xoá, thay thế, di chuyển con trỏ, tìm kiếm.

Bên cạnh khả năng thao tác chuỗi phức tạp với độ phức tạp thấp, lớp này còn hỗ trợ tính năng quay lui (undo), phục hồi (redo). Tất cả các thao tác đều được hiện thực sao cho đảm bảo tính đúng đắn, hiệu quả và dễ mở rộng.



Hình 6: Minh hoạ một Text Buffer với cursor ở vị trí đầu tiên



Hình 7: Minh hoạ một Text Buffer với cursor ở vị trí cuối cùng

Trong hình 6, vị trí của ký tự "A" gọi là vị trí ngay sau con trỏ. Trong hình 7, vị trí của ký tự "D" gọi là vị trí ngay trước con trỏ.

Các thuộc tính chính:

- `Rope rope`: Cây Rope dùng để lưu trữ nội dung văn bản.
- `int cursorPos`: Vị trí con trỏ hiện tại trong văn bản (tính từ đầu chuỗi). Cursor bắt đầu bằng 0, vị trí cuối cùng của cursor là sau ký tự cuối cùng của chuỗi.
- `HistoryManager* history`: Con trỏ đến bộ quản lý lịch sử thao tác.

Lớp `HistoryManager` sẽ được mô tả ở bên dưới.

Các phương thức công khai:

- `RopeTextBuffer();`
  Khởi tạo một buffer văn bản rỗng, con trỏ đặt tại vị trí đầu tiên.
- `~RopeTextBuffer();`
  Giải phóng toàn bộ bộ nhớ sử dụng bởi buffer và các thành phần liên quan.
  - **Độ phức tạp:** $O(n)$.
- `void insert(const string& s);`
  Chèn chuỗi mới tại vị trí ngay trước vị trí con trỏ hiện tại.
  - **Đầu vào:** Chuỗi `s` cần chèn.
  - **Độ phức tạp:** $O(\log n)$.
- `void deleteRange(int length);`
  Xoá một đoạn văn bản bắt đầu từ vị trí ngay sau con trỏ, với độ dài cho trước.
  - **Đầu vào:** `length` là số ký tự cần xoá.
  - **Độ phức tạp:** $O(\log n)$.
  - **Ngoại lệ:** Ném `out_of_range("Length is invalid!")` nếu `length` vượt quá giới hạn của chuỗi.
- `void replace(int length, const string& s);`
  Thay thế hoàn toàn `length` ký tự bắt đầu từ vị trí ngay sau con trỏ bằng chuỗi `s`, kể cả khi độ dài chuỗi `s` lớn hơn hay nhỏ hơn `length`. Sau khi thay thế, con trỏ nằm ngay sau chuỗi vừa được thay thế vào.
  - **Đầu vào:** độ dài cần thay thế`length` và chuỗi được thay thế `s`.
  - **Độ phức tạp:** $O(\log n)$.
  - **Ngoại lệ:** Ném `out_of_range("Length is invalid!")` nếu `length` vượt quá giới hạn của chuỗi.

- `void moveCursorTo(int index);`
  Di chuyển con trỏ đến vị trí chỉ định.
  - **Đầu vào:** `index` là chỉ số vị trí mới.
  - **Độ phức tạp:** $O(1)$.
  - **Ngoại lệ:** Ném `out_of_range("Index is invalid!")` nếu `index` vượt quá độ dài của chuỗi.
- `void moveCursorLeft();`
  Di chuyển con trỏ sang trái một ký tự.
  - **Độ phức tạp:** $O(1)$.
  - **Ngoại lệ:** Ném `cursor_error()` nếu con trỏ đang ở đầu.
- `void moveCursorRight();`
  Di chuyển con trỏ sang phải một ký tự.
  - **Độ phức tạp:** $O(1)$.
  - **Ngoại lệ:** Ném `cursor_error()` nếu con trỏ đang ở cuối.
- `int getCursorPos() const;`
  Trả về vị trí hiện tại của con trỏ.
  - **Đầu ra:** vị trí hiện tại của con trỏ.
  - **Độ phức tạp:** $O(1)$.
- `string getContent() const;`
  Trả về toàn bộ nội dung buffer hiện tại.
  - **Đầu ra:** Chuỗi kết quả.
  - **Độ phức tạp:** $O(n)$.
- `int findFirst(char c) const;`
  Tìm vị trí xuất hiện đầu tiên của ký tự `c`.
  - **Đầu vào:** ký tự `c`.
  - **Đầu ra:** vị trí đầu tiên xuất hiện ký tự `c`, nếu không tìm thấy thì trả về -1.
- `int* findAll(char c) const;`
  Tìm tất cả vị trí xuất hiện của ký tự `c`.
  - **Đầu vào:** ký tự `c`.
  - **Đầu ra:** Mảng động chứa các vị trí xuất hiện ký tự `c`. Nếu không tồn tại, trả về `nullptr`.

- `void undo();`
  Thực hiện thao tác undo, khôi phục trạng thái trước đó. Chỉ áp dụng trên một số thao tác được mô tả kỹ bên dưới.
  - **Độ phức tạp:** $O(\log n)$.
- `void redo();`
  Thực hiện thao tác redo (làm lại thao tác đã undo). Sau khi chèn/xoá (hành động `insert`/`delete`) một ký tự thì **không thể redo các hành động trước đó**, mà bắt buộc phải chờ đến khi có các thao tác undo tiếp theo được thực hiện.
  - **Độ phức tạp:** $O(\log n)$.
- `void printHistory() const;`
  In ra lịch sử thao tác đã thực hiện.
  - **Độ phức tạp:** $O(n)$.
- `void clear();`
  Xoá toàn bộ nội dung văn bản, đưa buffer về trạng thái rỗng.
  - **Độ phức tạp:** $O(n)$.

Lưu ý: Với những thao tác có thay đổi chiều dài của buffer, phải cập nhật vị trí con trỏ phù hợp.

Ví dụ 3.1

```
Giả sử buffer ban đầu rỗng:
|
Thao tác 1: insert("A")
A|
Thao tác 2: insert("CSE")
ACSE|
Thao tác 3: insert("HCMUT")
ACSEHCMUT|
Thao tác 4: moveCursorLeft()
ACSEHCMU|T
Thao tác 5: insert("123")
ACSEHCMU123|T
Thao tác 6: moveCursorTo(4)
ACSE|HCMU123T
Thao tác 7: deleteRange(3)
ACSE|U123T
Thao tác 8: undo()  (khôi phục chuỗi đã xóa)
ACSE|HCMU123T
Thao tác 9: undo()  (di chuyển cursor về lại vị trí trước thao tác 6)
ACSEHCMU123|T
Thao tác 10: undo()  (xóa chuỗi "123" đã thêm)
ACSEHCMU|T
Thao tác 11: redo()  (thêm lại chuỗi "123")
ACSEHCMU123|T
Thao tác 12: redo() (di chuyển cursor về lại vị trí 4)
ACSE|HCMU123T
Thao tác 13: redo() (xóa 3 ký tự)
ACSE|U123T
```

3.2.2 Lớp HistoryManager

Lớp `HistoryManager` chịu trách nhiệm lưu trữ và quản lý lịch sử các thao tác chỉnh sửa văn bản. Nhờ đó, người dùng có thể quay lại trạng thái trước đó (undo) hoặc phục hồi lại thao tác

vừa huỷ (redo).

Mỗi thao tác được lưu dưới dạng một đối tượng `Action` gồm thông tin mô tả ngắn, vị trí con trỏ trước và sau thao tác, cùng với nội dung văn bản có liên quan. Sinh viên tự đề xuất các thuộc tính phù hợp để lưu trữ lịch sử.

Cấu trúc `Action`:

- `string actionName`: mô tả tên thao tác.
- `int cursorBefore`: vị trí con trỏ trước khi thực hiện thao tác.
- `int cursorAfter`: vị trí con trỏ sau khi thực hiện thao tác.
- `string data`: chuỗi liên quan tới thao tác.

Các phương thức của lớp `HistoryManager`:

- `HistoryManager();`
  Hàm khởi tạo bộ quản lý lịch sử, thiết lập trạng thái ban đầu rỗng.
- `~HistoryManager();`
  Giải phóng toàn bộ vùng nhớ sử dụng trong quá trình quản lý lịch sử.
  - **Độ phức tạp:** $O(n)$
- `void addAction(const Action& a);`
  Lưu lại một hành động vào lịch sử thao tác.
  - **Đầu vào:** hành động `a` cần lưu.
  - **Độ phức tạp:** $O(1)$.
- `bool canUndo() const;`
  Kiểm tra xem có thể thực hiện thao tác undo hay không.
  - **Đầu ra:** `true` nếu có thể undo, ngược lại `false`.
  - **Độ phức tạp:** $O(1)$.
- `bool canRedo() const;`
  Kiểm tra xem có thể thực hiện thao tác redo hay không.
  - **Đầu ra:** `true` nếu có thể redo, ngược lại `false`.
  - **Độ phức tạp:** $O(1)$.
- `void printHistory() const;`
  In ra danh sách tất cả các hành động đã được lưu trong lịch sử thao tác.
  - **Định dạng:** `[(<action name>, <cursor before>, <cursor after>, <string data>), (<action name>, <cursor before>, <cursor after>, <string data>), ...]`

- **Độ phức tạp:** $O(n)$

Ví dụ 3.2

Với ví dụ 3.1, khi gọi hàm `printHistory()`, chương trình sẽ in ra:
`[(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)]`

Trong buffer, chỉ một số hành động có thể thực hiện undo/redo, đây cũng là các hành động cần được ghi vào lịch sử hành động, cụ thể:

- Hành động chèn ký tự:
  - **Tên hành động:** `insert`
  - **Chuỗi liên quan:** ký tự được chèn vào.
- Hành động xoá ký tự:
  - **Tên hành động:** `delete`.
  - **Chuỗi liên quan:** ký tự bị xoá đi.
- Hành động di chuyển con trỏ:
  - **Tên hành động:** `move`.
  - **Chuỗi liên quan:**
    - Nếu di chuyển sang trái, lưu ký tự `"L"`.
    - Nếu di chuyển sang phải, lưu ký tự `"R"`.
    - Nếu di chuyển đến một vị trí `index`, lưu ký tự `"J"`.
- Hành động thay thế chuỗi:
  - **Tên hành động:** `replace`.
  - **Chuỗi liên quan:** chuỗi **bị thay thế**.

4 Yêu cầu

Để hoàn thành bài tập này, sinh viên cần:

1. Đọc toàn bộ file mô tả này.

2. Tải file **initial.zip** và giải nén. Sau khi giải nén, sinh viên sẽ nhận được các file bao gồm: `main.cpp`, `main.h`, `RopeTextBuffer.h` và `RopeTextBuffer.cpp`. Sinh viên chỉ nộp 2 file, đó là `RopeTextBuffer.h` và `RopeTextBuffer.cpp`. Do đó, không được phép chỉnh sửa file `main.h` khi kiểm tra chương trình.
3. Trong bài tập lớn này không cho phép sử dụng các thư viện hỗ trợ danh sách có sẵn. Nếu muốn dùng các kiểu dữ liệu danh sách, sinh viên có thể tái sử dụng lớp `DoublyLinkedList` ở Bài tập lớn 1 hoặc tự định nghĩa lại.
4. Sinh viên sử dụng lệnh sau để biên dịch:
   `g++ -o main main.cpp RopeTextBuffer.cpp -I . -std=c++17`
   Lệnh trên được sử dụng trong Command Prompt/Terminal để biên dịch chương trình. Nếu sinh viên sử dụng IDE để chạy chương trình, cần lưu ý: thêm tất cả các file vào project/workspace của IDE; chỉnh lại lệnh biên dịch trong IDE cho phù hợp. IDE thường cung cấp nút Build (biên dịch) và Run (chạy). Khi bấm Build, IDE sẽ chạy câu lệnh biên dịch tương ứng, thông thường chỉ biên dịch file `main.cpp`. Sinh viên cần tìm cách cấu hình để thay đổi câu lệnh biên dịch, cụ thể: thêm file `RopeTextBuffer.cpp`, thêm tùy chọn `-std=c++17`, và `-I .`
5. Chương trình sẽ được chấm trên nền tảng Unix. Môi trường của sinh viên và trình biên dịch có thể khác với môi trường chấm thực tế. Khu vực nộp bài trên LMS được thiết lập tương tự môi trường chấm thực tế. Sinh viên bắt buộc phải kiểm tra chương trình trên trang nộp bài, đồng thời sửa tất cả lỗi phát sinh trên hệ thống LMS để đảm bảo kết quả chính xác khi chấm cuối cùng.
6. Chỉnh sửa file `RopeTextBuffer.h` và `RopeTextBuffer.cpp` để hoàn thành bài tập, đồng thời đảm bảo hai yêu cầu sau:
   - Tất cả các phương thức được mô tả trong file hướng dẫn phải được cài đặt để chương trình có thể biên dịch thành công. Nếu sinh viên chưa hiện thực một phương thức nào đó, cần cung cấp phần hiện thực rỗng cho phương thức đó. Mỗi testcase sẽ gọi một số phương thức để kiểm tra kết quả trả về.
   - Trong file `RopeTextBuffer.h` chỉ được phép có đúng một dòng `#include "main.h"`, và trong file `RopeTextBuffer.cpp` chỉ được phép có một dòng `#include "RopeTextBuffer.h"`. Ngoài hai dòng này, không được phép thêm bất kỳ lệnh `#include` nào khác trong các file này.
7. Khuyến khích sinh viên được phép viết thêm các lớp, phương thức và thuộc tính phụ trợ trong các lớp yêu cầu hiện thực. Nhưng sinh viên phải đảm bảo các lớp, phương thức này không làm thay đổi yêu cầu của các phương thức được mô tả trong đề bài.
8. Sinh viên không được chỉnh sửa các prototype, các định nghĩa có sẵn trong các file.

9. Sinh viên phải thiết kế và sử dụng các cấu trúc dữ liệu đã học.
10. Sinh viên bắt buộc phải giải phóng toàn bộ vùng nhớ cấp phát động khi chương trình kết thúc.

5 Harmony cho Bài tập lớn

Bài kiểm tra cuối kì của môn học sẽ có một số câu hỏi Harmony với nội dung của BTL.

Sinh viên phải giải quyết BTL bằng khả năng của chính mình. Nếu sinh viên gian lận trong BTL, sinh viên sẽ không thể trả lời câu hỏi Harmony và nhận điểm 0 cho BTL.

Sinh viên **phải** chú ý làm câu hỏi Harmony trong bài kiểm tra cuối kỳ. Các trường hợp không làm sẽ tính là 0 điểm cho BTL, và bị không đạt cho môn học. **Không chấp nhận giải thích và không có ngoại lệ.**

6 Quy định và xử lý gian lận

Bài tập lớn phải được sinh viên TỰ LÀM. Sinh viên sẽ bị coi là gian lận nếu:

- Có sự giống nhau bất thường giữa mã nguồn của các bài nộp. Trong trường hợp này, TẤT CẢ các bài nộp đều bị coi là gian lận. Do vậy sinh viên phải bảo vệ mã nguồn bài tập lớn của mình.
- Sinh viên không hiểu mã nguồn do chính mình viết, trừ những phần mã được cung cấp sẵn trong chương trình khởi tạo. Sinh viên có thể tham khảo từ bất kỳ nguồn tài liệu nào, tuy nhiên phải đảm bảo rằng mình hiểu rõ ý nghĩa của tất cả những dòng lệnh mà mình viết. Trong trường hợp không hiểu rõ mã nguồn của nơi mình tham khảo, sinh viên được đặc biệt cảnh báo là KHÔNG ĐƯỢC sử dụng mã nguồn này; thay vào đó nên sử dụng những gì đã được học để viết chương trình.
- Nộp nhầm bài của sinh viên khác trên tài khoản cá nhân của mình.
- Sinh viên sử dụng các công cụ AI trong quá trình làm bài tập lớn dẫn đến các mã nguồn giống nhau.

Trong trường hợp bị kết luận là gian lận, sinh viên sẽ bị điểm 0 cho toàn bộ môn học (không chỉ bài tập lớn).

KHÔNG CHẤP NHẬN BẤT KỲ GIẢI THÍCH NÀO VÀ KHÔNG CÓ BẤT KỲ NGOẠI LỆ NÀO!

Sau mỗi bài tập lớn được nộp, sẽ có một số sinh viên được gọi phỏng vấn ngẫu nhiên để chứng minh rằng bài tập lớn vừa được nộp là do chính mình làm.

Một số quy định khác:

- Mọi quyết định của giảng viên phụ trách bài tập lớn là quyết định cuối cùng.
- Sinh viên không được cung cấp testcase sau khi chấm bài.
- Nội dung Bài tập lớn sẽ được Harmony với các câu hỏi trong bài kiểm tra cuối kì với nội dung tương tự.

7 Thay đổi so với phiên bản trước

- Thêm ngoại lệ cho các phương thức `charAt, substring, deleteRange`

——————————HẾT——————————