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 1

HIỆN THỰC VECTORSTORE SỬ DỤNG DANH SÁCH

TP. HỒ CHÍ MINH, THÁNG 09/2025

ĐẶC TẢ BÀI TẬP LỚN

Phiên bản 1.0

1 Chuẩn đầu ra

Sau khi hoàn thành bài tập lớn này, sinh viên sẽ có khả năng:

- Thành thạo lập trình hướng đối tượng (OOP).
- Phát triển các cấu trúc dữ liệu danh sách.
- Sử dụng các cấu trúc dữ liệu dạng danh sách để hiện thực một VectorStore.

2 Dẫn nhập

Trong Bài tập lớn 1, sinh viên được yêu cầu hiện thực một **VectorStore** sử dụng hai cấu trúc dữ liệu cơ bản: **Danh sách liên kết đơn (Singly Linked List – SinglyLinkedList)** và **Danh sách mảng động (ArrayList)**. Mỗi vector được biểu diễn dưới dạng một danh sách liên kết đơn, trong khi toàn bộ tập hợp các vector sẽ được quản lý bởi một danh sách mảng động. Thiết kế này cho phép mô phỏng cách lưu trữ và tìm kiếm vector trong các hệ thống cơ sở dữ liệu hiện đại.

Việc sử dụng danh sách liên kết đơn giúp sinh viên luyện tập thao tác với cấu trúc dữ liệu tuyến tính động, đồng thời hiểu rõ cơ chế cấp phát bộ nhớ và quản lý con trỏ. Danh sách mảng động được tích hợp nhằm minh họa cách mảng động hoạt động khi cần mở rộng dung lượng, từ đó rèn luyện khả năng hiện thực các thao tác chèn, xoá, và truy cập theo chỉ số.

VectorStore là một trong những công nghệ có tính ứng dụng cao trong thực tế, được sử dụng trong các hệ thống tìm kiếm ngữ nghĩa, gợi ý nội dung, hoặc quản lý tri thức. Thông qua bài tập lớn này, sinh viên không chỉ rèn luyện kỹ năng lập trình hướng đối tượng (OOP) và củng cố kiến thức về cấu trúc dữ liệu tuyến tính, mà còn tiếp cận với ý tưởng xây dựng một ứng dụng cơ bản của **VectorStore**. Đây sẽ là nền tảng để sinh viên hiểu rõ hơn cách mà dữ liệu dạng vector được quản lý và khai thác trong các hệ thống hiện đại.

3 Mô tả

3.1 Các cấu trúc dữ liệu dạng danh sách

3.1.1 Danh sách mảng động - ArrayList

Danh sách mảng động (ArrayList) là một cấu trúc dữ liệu tuyến tính cho phép lưu trữ và quản lý tập hợp các phần tử cùng kiểu trong một mảng có khả năng thay đổi kích thước động. Khác với mảng tĩnh có kích thước cố định, ArrayList tự động mở rộng dung lượng khi số lượng phần tử vượt quá sức chứa hiện tại. Cơ chế này thường được hiện thực bằng cách cấp phát một mảng mới có dung lượng lớn hơn (thường gấp đôi dung lượng cũ), sau đó sao chép toàn bộ các phần tử cũ sang mảng mới.

Danh sách mảng động hỗ trợ hiệu quả các thao tác truy cập ngẫu nhiên theo chỉ số với độ phức tạp $O(1)$. Tuy nhiên, việc chèn hoặc xoá tại vị trí bất kỳ trong danh sách thường có độ phức tạp $O(n)$ do cần dịch chuyển các phần tử để bảo toàn thứ tự.

Thuộc tính của lớp ArrayList

Lớp `ArrayList<T>` được thiết kế với các thuộc tính cơ bản sau nhằm quản lý và tổ chức dữ liệu trong mảng động:

- `T* data`: Con trỏ đến vùng nhớ động dùng để lưu trữ các phần tử của danh sách. Vùng nhớ này có thể được cấp phát lại với dung lượng lớn hơn khi số phần tử vượt quá sức chứa hiện tại.
- `int capacity`: Sức chứa hiện tại của mảng động, biểu thị số lượng phần tử tối đa mà vùng nhớ `data` có thể lưu giữ tại một thời điểm. Khi số phần tử vượt quá `capacity`, lớp sẽ tự động mở rộng dung lượng theo cơ chế tăng trưởng.
- `int count`: Số lượng phần tử thực tế đang được lưu trong danh sách. Giá trị này luôn nhỏ hơn hoặc bằng `capacity`, và được cập nhật sau mỗi thao tác thêm hoặc xoá phần tử.

Phương thức `private`

- `void ensureCapacity(int cap)`
  - **Chức năng**: Đảm bảo mảng động có đủ sức chứa ít nhất `cap` phần tử. Nếu `cap` lớn hơn giá trị `capacity` hiện tại, tăng `capacity` lên `1.5` lần, tạo một mảng động mới với giá trị `capacity` mới và sao chép toàn bộ phần tử cũ sang mảng mới.

  - **Ngoại lệ**: Không có ngoại lệ.
  - **Độ phức tạp**: O(n)

Các phương thức `public`

- `ArrayList(int initCapacity = 10)`
  - **Chức năng**: Khởi tạo một danh sách mảng động rỗng với sức chứa ban đầu bằng `initCapacity`.
  - **Ngoại lệ**: Không có ngoại lệ.
  - **Độ phức tạp**: O(1).
- `ArrayList(const ArrayList<T>& other)`
  - **Chức năng**: Khởi tạo một danh sách mới bằng cách sao chép toàn bộ dữ liệu từ danh sách `other`.
  - **Ngoại lệ**: Không có ngoại lệ.
  - **Độ phức tạp**: O(n).
- `~ArrayList()`
  - **Chức năng**: Giải phóng toàn bộ vùng nhớ đã cấp phát cho danh sách, tránh rò rỉ bộ nhớ.
  - **Ngoại lệ**: Không có ngoại lệ.
  - **Độ phức tạp**: O(1).
- `ArrayList<T>& operator=(const ArrayList<T>& other)`
  - **Chức năng**: Gán toàn bộ dữ liệu từ danh sách `other` sang danh sách hiện tại.
  - **Ngoại lệ**: Không có ngoại lệ.
  - **Độ phức tạp**: O(n).
- `void add(T e)`
  - **Chức năng**: Thêm phần tử `e` vào cuối mảng.
  - **Ngoại lệ**: Không có ngoại lệ.
  - **Độ phức tạp**: O(n).
- `void add(int index, T e)`
  - **Chức năng**: Thêm phần tử `e` vào vị trí chỉ định `index`, các phần tử sau đó sẽ được dịch phải một vị trí.
  - **Ngoại lệ**: Ném `out_of_range("Index is invalid!")` nếu `index` không hợp lệ.
  - **Độ phức tạp**: O(n).

- `T removeAt(int index)`
  - **Chức năng**: Xoá và trả về phần tử tại vị trí `index`, các phần tử phía sau sẽ dịch trái một vị trí.
  - **Ngoại lệ**: Ném `out_of_range("Index is invalid!")` nếu `index` không hợp lệ.
  - **Độ phức tạp**: O(n).
- `bool empty()`
  - **Chức năng**: Kiểm tra danh sách có rỗng hay không.
  - **Ngoại lệ**: Không có ngoại lệ.
  - **Độ phức tạp**: O(1).
- `int size()`
  - **Chức năng**: Trả về số lượng phần tử hiện tại trong danh sách.
  - **Ngoại lệ**: Không có ngoại lệ.
  - **Độ phức tạp**: O(1).
- `void clear()`
  - **Chức năng**: Xoá toàn bộ phần tử trong danh sách, đặt lại `count` về 0, `capacity` bằng 10.
  - **Ngoại lệ**: Không có ngoại lệ.
  - **Độ phức tạp**: O(1).
- `T& get(int index)`
  - **Chức năng**: Trả về tham chiếu đến phần tử tại vị trí `index`.
  - **Ngoại lệ**: Ném `out_of_range("Index is invalid!")` nếu `index` không hợp lệ.
  - **Độ phức tạp**: O(1).
- `void set(int index, T e)`
  - **Chức năng**: Gán lại giá trị phần tử tại vị trí `index` thành `e`.
  - **Ngoại lệ**: Ném `out_of_range("Index is invalid!")` nếu `index` không hợp lệ.
  - **Độ phức tạp**: O(1).
- `int indexOf(T item)`
  - **Chức năng**: Trả về chỉ số của phần tử đầu tiên có giá trị bằng `item`. Nếu không tìm thấy thì trả về -1.
  - **Ngoại lệ**: Không có ngoại lệ.
  - **Độ phức tạp**: O(n).
- `bool contains(T item)`

  - **Chức năng**: Kiểm tra xem danh sách có chứa phần tử có giá trị bằng `item` hay không.
  - **Ngoại lệ**: Không có ngoại lệ.
  - **Độ phức tạp**: O(n).

- `string toString(string (*item2str)(T&) = 0)`
  - **Chức năng**: Trả về biểu diễn chuỗi của toàn bộ danh sách. Nếu con trỏ hàm `item2str` được cung cấp, phương thức này sẽ dùng để chuyển từng phần tử sang chuỗi.
  - **Ngoại lệ**: Không có ngoại lệ.
  - **Độ phức tạp**: O(n).
  - **Định dạng in ra:** `[<element 1>, <element 2>, <element 3>, ...]`.

Ví dụ 3.1

Nếu mảng bao gồm: 1, 2, 3, 4, 5, con trỏ hàm được cung cấp để định dạng số nguyên thành chuỗi.
Kết quả trả về của toString: `[1, 2, 3, 4, 5]`

- `Iterator begin()`
  - **Chức năng**: Trả về iterator trỏ đến phần tử đầu tiên của danh sách. Lớp `Iterator` sẽ được mô tả ở bên dưới.
  - **Ngoại lệ**: Không có ngoại lệ.
  - **Độ phức tạp**: O(1).

- `Iterator end()`
  - **Chức năng**: Trả về iterator trỏ đến vị trí sau phần tử cuối cùng của danh sách. Lớp `Iterator` sẽ được mô tả ở bên dưới.
  - **Ngoại lệ**: Không có ngoại lệ.
  - **Độ phức tạp**: O(1).

Inner class: Iterator

`Iterator` là lớp lồng bên trong `ArrayList<T>` dùng để duyệt tuần tự các phần tử của danh sách mảng động. Mỗi `Iterator` duy trì một chỉ số hiện tại trong danh sách và cho phép truy cập phần tử tại vị trí đó, cũng như tiến tới phần tử kế tiếp. Sinh viên có thể tham khảo về iterator tại 1, 2

Quy ước: `begin()` trả về iterator tại vị trí đầu (chỉ số 0) và `end()` trả về iterator tại vị trí

sau phần tử cuối (chỉ số bằng `count`). Hai iterator được coi là khác nhau nếu khác `pList` hoặc khác chỉ số `cursor`.

Các thuộc tính

- `int cursor`: Chỉ số hiện tại trong danh sách. Hợp lệ trong khoảng $[0, \texttt{count}]$. Khi `cursor == count`, iterator ở trạng thái `end()` và không có phần tử để dereference.
- `ArrayList<T>* pList`: Con trỏ tới danh sách mảng động mà iterator đang duyệt.

Các phương thức

- `Iterator(ArrayList<T>* pList = nullptr, int index = 0)`
  - **Chức năng**: Khởi tạo iterator trỏ vào mảng `pList` tại vị trí `index`. Sinh viên cần dựa vào sự tương quan giữa `index` và mảng được truyền vào `pList` để kiểm tra tính hợp lệ của `index`.
  - **Ngoại lệ**: Ném `out_of_range("Index is invalid!")` nếu `index` ngoài miền hợp lệ.
  - **Độ phức tạp**: $O(1)$.
- `Iterator& operator=(const Iterator& other)`
  - **Chức năng**: Gán trạng thái từ iterator `other` cho iterator hiện tại.
  - **Ngoại lệ**: Không có ngoại lệ.
  - **Độ phức tạp**: $O(1)$.
- `T& operator*()`
  - **Chức năng**: Trả về tham chiếu tới phần tử tại vị trí `cursor` trong `pList`. Ném lỗi nếu dereference khi iterator đang ở `end()`.
  - **Ngoại lệ**: Ném `out_of_range("Iterator is out of range!")` nếu `cursor` không hợp lệ.
  - **Độ phức tạp**: $O(1)$.
- `bool operator!=(const Iterator& other)`
  - **Chức năng**: So sánh khác nhau giữa hai iterator. Hai iterator được xem là khác nhau nếu `pList` khác nhau hoặc `cursor` khác nhau.
  - **Ngoại lệ**: Không có ngoại lệ.
  - **Độ phức tạp**: $O(1)$.
- `Iterator& operator++()`
  - **Chức năng**: Tiến iterator đến phần tử kế tiếp (tiền tố, `++it`).

  - **Ngoại lệ**: Ném `out_of_range("Iterator cannot advance past end!")` nếu `cursor` đã ở vị trí `count`.
  - **Độ phức tạp**: $O(1)$.

- `Iterator operator++(int)`
  - **Chức năng**: Tiến iterator đến phần tử kế tiếp (hậu tố, `it++`) và trả về **bản sao** giá trị cũ.
  - **Ngoại lệ**: Ném `out_of_range("Iterator cannot advance past end!")` nếu `cursor` đã ở vị trí `count`.
  - **Độ phức tạp**: $O(1)$.

- `Iterator& operator--()`
  - **Chức năng**: Lùi iterator về phần tử trước đó (tiền tố, `--it`). Nếu iterator đang ở vị trí `end()`, phép giảm này sẽ đưa iterator về phần tử cuối cùng.
  - **Ngoại lệ**: Ném `out_of_range("Iterator cannot move before begin!")` nếu đang ở vị trí đầu tiên.
  - **Độ phức tạp**: $O(1)$.

- `Iterator operator--(int)`
  - **Chức năng**: Lùi iterator về phần tử trước đó (hậu tố, `it--`) và trả về **bản sao** giá trị cũ.
  - **Ngoại lệ**: Ném `out_of_range("Iterator cannot move before begin!")` nếu đang ở vị trí đầu tiên.
  - **Độ phức tạp**: $O(1)$.

3.1.2 Danh sách liên kết đơn - SinglyLinkedList

Danh sách liên kết đơn (Singly Linked List, viết tắt SinglyLinkedList) là một cấu trúc dữ liệu tuyến tính trong đó các phần tử được lưu trữ dưới dạng các nút (node). Mỗi nút bao gồm hai thành phần: dữ liệu (data) và một con trỏ (next) trỏ đến nút kế tiếp trong danh sách. Nút cuối cùng của danh sách có con trỏ `next` bằng `nullptr`, biểu thị kết thúc danh sách.

Khác với mảng tĩnh hoặc danh sách mảng động, SinglyLinkedList không yêu cầu cấp phát bộ nhớ liên tục cho toàn bộ phần tử. Thay vào đó, mỗi phần tử được cấp phát riêng biệt và liên kết lại thông qua các con trỏ. Nhờ vậy, việc chèn hoặc xoá phần tử tại đầu hoặc giữa danh sách trở nên linh hoạt hơn, với chi phí thao tác trung bình là $O(1)$ cho các thao tác tại đầu danh sách và $O(n)$ cho thao tác tại vị trí bất kỳ.

Tuy nhiên, do đặc thù chỉ hỗ trợ duyệt theo một chiều (từ đầu đến cuối), việc truy cập

ngẫu nhiên tới phần tử tại vị trí bất kỳ sẽ tốn chi phí $O(n)$. Điều này khiến danh sách liên kết đơn kém hiệu quả hơn so với danh sách mảng động trong các bài toán yêu cầu truy cập trực tiếp theo chỉ số.

Lớp `Node`

Thuộc tính:

- `T data`: Dữ liệu của phần tử được lưu trong nút.
- `Node* next`: Con trỏ trỏ đến nút kế tiếp trong danh sách.

Các thuộc tính của lớp `SinglyLinkedList`

- `Node* head`: Con trỏ tới nút đầu tiên trong danh sách.
- `Node* tail`: Con trỏ tới nút trong danh sách.
- `int count`: Số lượng phần tử hiện có trong danh sách.

Các phương thức `public` của lớp `SinglyLinkedList`

- `SinglyLinkedList()`
  - **Chức năng**: Khởi tạo một danh sách liên kết đơn rỗng.
  - **Ngoại lệ**: Không có ngoại lệ.
  - **Độ phức tạp**: O(1).
- `~SinglyLinkedList()`
  - **Chức năng**: Giải phóng toàn bộ vùng nhớ các nút đã cấp phát, tránh rò rỉ bộ nhớ.
  - **Ngoại lệ**: Không có ngoại lệ.
  - **Độ phức tạp**: O(n).
- `void add(T e)`
  - **Chức năng**: Thêm phần tử `e` vào cuối danh sách.
  - **Ngoại lệ**: Không có ngoại lệ.
  - **Độ phức tạp**: O(n).
- `void add(int index, T e)`
  - **Chức năng**: Thêm phần tử `e` vào vị trí `index`.
  - **Ngoại lệ**: Ném `out_of_range("Index is invalid!")` nếu `index` không hợp lệ
  - **Độ phức tạp**: O(n).
- `T removeAt(int index)`

  - **Chức năng**: Xoá và trả về phần tử tại vị trí `index`.
  - **Ngoại lệ**: Ném `out_of_range("Index is invalid!")` nếu `index` không hợp lệ.
  - **Độ phức tạp**: O(n).

- `bool removeItem(T item)`
  - **Chức năng**: Xoá nút đầu tiên có giá trị bằng `item` trong danh sách. Trả về `true` nếu xoá thành công, `false` nếu không tìm thấy.
  - **Ngoại lệ**: Không có ngoại lệ.
  - **Độ phức tạp**: O(n).

- `bool empty()`
  - **Chức năng**: Kiểm tra danh sách có rỗng hay không.
  - **Ngoại lệ**: Không có ngoại lệ.
  - **Độ phức tạp**: O(1).

- `int size()`
  - **Chức năng**: Trả về số lượng phần tử hiện tại trong danh sách.
  - **Ngoại lệ**: Không có ngoại lệ.
  - **Độ phức tạp**: O(1).

- `void clear()`
  - **Chức năng**: Xoá toàn bộ nút trong danh sách.
  - **Ngoại lệ**: Không có ngoại lệ.
  - **Độ phức tạp**: O(n).

- `T& get(int index)`
  - **Chức năng**: Trả về tham chiếu tới dữ liệu tại vị trí `index` trong danh sách.
  - **Ngoại lệ**: Ném `out_of_range("Index is invalid!")` nếu `index` không hợp lệ.
  - **Độ phức tạp**: O(n).

- `int indexOf(T item)`
  - **Chức năng**: Trả về chỉ số của phần tử đầu tiên có giá trị bằng `item`. Nếu không tìm thấy thì trả về -1.
  - **Ngoại lệ**: Không có ngoại lệ.
  - **Độ phức tạp**: O(n).

- `bool contains(T item)`
  - **Chức năng**: Kiểm tra xem danh sách có chứa phần tử bằng `item` hay không.
  - **Ngoại lệ**: Không có ngoại lệ.

  - **Độ phức tạp**: O(n).

- `string toString(string (*item2str)(T&) = 0)`
  - **Chức năng**: Trả về chuỗi biểu diễn toàn bộ danh sách. Nếu con trỏ hàm `item2str` được cung cấp, hàm sẽ được dùng để chuyển đổi phần tử sang chuỗi. Nếu không, sử dụng biểu diễn mặc định của kiểu dữ liệu.
  - **Ngoại lệ**: Không có ngoại lệ.
  - **Độ phức tạp**: O(n).
  - **Định dạng in ra:** `[<element 1>]->[<element 2>]->[<element 3>]...`

Ví dụ 3.2

Nếu mảng bao gồm: 1, 2, 3, 4, 5, con trỏ hàm được cung cấp để định dạng số nguyên thành chuỗi.
Kết quả trả về của toString: `[1]->[2]->[3]->[4]->[5]`

- `Iterator begin()`
  - **Chức năng**: Trả về iterator trỏ đến phần tử đầu tiên của danh sách. Lớp `Iterator` sẽ được mô tả ở bên dưới.
  - **Ngoại lệ**: Không có ngoại lệ.
  - **Độ phức tạp**: O(1).
- `Iterator end()`
  - **Chức năng**: Trả về iterator trỏ đến vị trí sau phần tử cuối cùng của danh sách. Lớp `Iterator` sẽ được mô tả ở bên dưới.
  - **Ngoại lệ**: Không có ngoại lệ.
  - **Độ phức tạp**: O(1).

Inner class: Iterator

`Iterator` là lớp lồng bên trong `SinglyLinkedList<T>` dùng để duyệt tuần tự các phần tử của danh sách liên kết đơn. Mỗi `Iterator` duy trì một con trỏ đến nút (Node) hiện tại, cho phép truy cập dữ liệu của nút này và tiến sang nút kế tiếp bằng phép tăng (`++`). Quy ước: `begin()` trả về iterator trỏ đến nút đầu tiên, còn `end()` trả về iterator có con trỏ bằng `nullptr`, biểu diễn trạng thái kết thúc.

Thuộc tính

- `Node* current`: Con trỏ trỏ đến nút hiện tại trong danh sách.

Phương thức

- `Iterator(Node* node = nullptr)`
  - **Chức năng**: Khởi tạo iterator trỏ đến nút `node` trong danh sách.
  - **Ngoại lệ**: Không có ngoại lệ.
  - **Độ phức tạp**: O(1).
- `Iterator& operator=(const Iterator& other)`
  - **Chức năng**: Gán iterator hiện tại bằng iterator `other`.
  - **Ngoại lệ**: Không có ngoại lệ.
  - **Độ phức tạp**: O(1).
- `T& operator*()`
  - **Chức năng**: Trả về tham chiếu đến dữ liệu (`data`) của nút hiện tại.
  - **Ngoại lệ**: Ném `out_of_range("Iterator is out of range!")` nếu `current == nullptr`.
  - **Độ phức tạp**: O(1).
- `bool operator!=(const Iterator& other)`
  - **Chức năng**: So sánh sự khác nhau giữa hai iterator.
  - **Ngoại lệ**: Không có ngoại lệ.
  - **Độ phức tạp**: O(1).
- `Iterator& operator++()`
  - **Chức năng**: Tiến iterator sang nút kế tiếp trong danh sách (tiền tố, `++it`) và trả về nút mới nhất.
  - **Ngoại lệ**: Ném `out_of_range("Iterator cannot advance past end!")` nếu không thể tăng.
  - **Độ phức tạp**: O(1).
- `Iterator operator++(int)`
  - **Chức năng**: Tiến iterator sang nút kế tiếp (hậu tố, `it++`) và trả về **bản sao** giá trị cũ.
  - **Ngoại lệ**: Ném `out_of_range("Iterator cannot advance past end!")` nếu không thể tăng.
  - **Độ phức tạp**: O(1).

3.2 VectorStore

3.2.1 Tổng quan về VectorStore

VectorStore là một kho dữ liệu đặc biệt dùng để lưu trữ và truy vấn các vector nhiều chiều. Trong khoa học máy tính, vector thường được dùng để biểu diễn thông tin (ví dụ: một câu văn, một đoạn hội thoại, một hình ảnh) dưới dạng một dãy số thực. Mục tiêu của VectorStore là cho phép lưu trữ nhiều vector cùng lúc, đồng thời hỗ trợ các phép truy vấn tìm kiếm dựa trên **độ tương đồng** giữa vector truy vấn và các vector trong kho.

Cơ chế hoạt động: Một **VectorStore** có các cơ chế cơ bản để cung cấp những tính năng xử lý, lưu trữ và truy vấn dữ liệu, cụ thể:

- **Nhúng dữ liệu (Embedding):** Dữ liệu thô (raw data), chẳng hạn một chuỗi văn bản, trước tiên sẽ được biến đổi thành một vector số thực nhờ một hàm nhúng (*embedding function*), vector này thường sẽ có số chiều cố định đối với từng loại VectorStore.
  Ví dụ: Câu "Con mèo trắng" có thể được ánh xạ thành vector $[0.2, -0.1, 0.5, 0.0, \dots]$.
- **Lưu trữ:** Sau khi một vector được sinh ra, VectorStore sẽ lưu trữ vector đó kèm với các dữ liệu mô tả (*metadata*), ví dụ như: mã định danh, dữ liệu gốc, timestamp, ...
  Ví dụ: id = 1, rawText = "Con mèo trắng", vector = $[0.2, -0.1, 0.5, 0.0, \dots]$.
- **Duyệt và xử lý hàng loạt:** VectorStore cho phép duyệt qua toàn bộ vector đã lưu để áp dụng các thao tác chung, ví dụ chuẩn hoá (normalization), in thông tin, hay tính toán thống kê.
- **Tìm kiếm tương tự (Similarity Search):** Khi nhận một truy vấn (query), trước tiên query cũng được biến đổi thành vector số thực. Sau đó, VectorStore sẽ so sánh query với toàn bộ vector trong kho bằng một thước đo độ tương đồng. Cuối cùng, VectorStore sẽ trả về vector gần nhất, hoặc danh sách k vector gần nhất.
  Ví dụ: Query "Con chó đen" ánh xạ thành vector, so sánh với các vector trong kho, và VectorStore trả về 3 câu văn có vector gần nhất.

VectorStore cung cấp một ví dụ thực tiễn cho việc kết hợp **lưu trữ dữ liệu** và **giải thuật tìm kiếm**. Nó cho thấy rằng thay vì so sánh dữ liệu thô (văn bản), ta có thể biến đổi chúng sang dạng số học (vector nhiều chiều), từ đó việc tính toán độ tương đồng trở nên khả thi bằng các phép toán cơ bản như tích vô hướng, chuẩn vector, hoặc khoảng cách. Cơ chế này là nền tảng cho nhiều hệ thống hiện đại như tìm kiếm ngữ nghĩa (semantic search), hệ gợi ý (recommendation systems) và hệ thống hỏi đáp dựa trên tri thức (QA systems).

3.2.2 Lớp VectorStore

Lớp `VectorStore` là lớp mô phỏng một kho lưu trữ vector nhiều chiều cùng với siêu dữ liệu (metadata) đi kèm. Mục tiêu của lớp là cho phép sinh viên làm quen với cơ chế cơ bản của các hệ thống tìm kiếm ngữ nghĩa: dữ liệu thô (`rawText`) được ánh xạ thành một vector số thực nhiều chiều nhờ một *hàm nhúng* (`embeddingFunction`), sau đó vector này được lưu trữ và có thể truy vấn lại dựa trên độ tương đồng.

Cấu trúc `VectorRecord`

Để đảm bảo tính đồng bộ khi lưu trữ, lớp `VectorStore` sử dụng một cấu trúc trung gian gọi là `VectorRecord`. Mỗi `VectorRecord` đại diện cho một bản ghi trong kho, bao gồm:

- `int id`: Định danh duy nhất của vector.
- `string rawText`: Chuỗi gốc trước khi được ánh xạ.
- `int rawLength`: Chiều dài của chuỗi `rawText`.
- `SinglyLinkedList<float>* vector`: Con trỏ tới danh sách liên kết đơn chứa vector số thực nhiều chiều.

Các thuộc tính của lớp `VectorStore`

- `ArrayList<VectorRecord*> records`: Danh sách mảng động chứa toàn bộ các bản ghi vector và siêu dữ liệu đi kèm.
- `int dimension`: Số chiều cố định của mọi vector trong kho.
- `int count`: Số lượng bản ghi hiện có trong kho.
- `SinglyLinkedList<float>* (*embeddingFunction)(const string&)`: Con trỏ hàm ánh xạ từ chuỗi văn bản sang vector nhiều chiều.

Các phương thức `public` của lớp `VectorStore`

- `VectorStore(int dimension = 512, SinglyLinkedList<float>* (*embeddingFunction)(const string&))`
  - **Chức năng**: Khởi tạo một kho vector rỗng với số chiều cố định và gắn hàm nhúng ban đầu.
  - **Ngoại lệ**: Không có.
  - **Độ phức tạp**: O(1).
- `int size()`
  - **Chức năng**: Trả về số lượng vector hiện có trong kho.

  - **Ngoại lệ**: Không có.
  - **Độ phức tạp**: O(1).

- `bool empty()`
  - **Chức năng**: Kiểm tra kho có rỗng hay không.
  - **Ngoại lệ**: Không có.
  - **Độ phức tạp**: O(1).

- `void clear()`
  - **Chức năng**: Xoá toàn bộ vector và siêu dữ liệu đi kèm.
  - **Ngoại lệ**: Không có.
  - **Độ phức tạp**: O(n).

- `SinglyLinkedList<float>* preprocessing(string rawText)`
  - **Chức năng**: Tiền xử lý dữ liệu văn bản đầu vào và chuyển đổi thành vector số thực nhiều chiều.
  - **Yêu cầu**:
    1. Gọi `embeddingFunction` để ánh xạ chuỗi `rawText` thành vector.
    2. Kiểm tra số chiều của vector kết quả:
       * Nếu số chiều lớn hơn `dimension`, cắt bớt các phần tử dư thừa ở cuối.
       * Nếu số chiều nhỏ hơn `dimension`, thêm các phần tử có giá trị `0.0` vào cuối (post-padding) cho đến khi đủ số chiều.
  - **Ngoại lệ**: Không có.
  - **Độ phức tạp**: O(d).

- `void addText(string rawText)`
  - **Chức năng**: Thêm một bản ghi mới vào kho từ chuỗi văn bản gốc. Cần phải sử dụng method `preprocessing` để tiền xử lý `rawText`.
  - **Ngoại lệ**: Không có.
  - **Độ phức tạp**: O(n).

- `SinglyLinkedList<float>& getVector(int index)`
  - **Chức năng**: Truy xuất tham chiếu tới vector tại vị trí `index`.
  - **Ngoại lệ**: Ném `out_of_range("Index is invalid!")` nếu chỉ số không hợp lệ.
  - **Độ phức tạp**: O(1).

- `string getRawText(int index)`
  - **Chức năng**: Lấy chuỗi gốc đi kèm vector tại vị trí `index`.

  - **Ngoại lệ**: Ném `out_of_range("Index is invalid!")` nếu chỉ số không hợp lệ.
  - **Độ phức tạp**: O(1).

- `int getId(int index)`
  - **Chức năng**: Lấy `id` của vector tại vị trí `index`.
  - **Ngoại lệ**: Ném `out_of_range("Index is invalid!")` nếu chỉ số không hợp lệ.
  - **Độ phức tạp**: O(1).

- `bool removeAt(int index)`
  - **Chức năng**: Xoá vector và metadata tại vị trí `index`.
  - **Yêu cầu:** Giải phóng vector được khai báo động và xoá `record` của vector đó khỏi VectorStore.
  - **Ngoại lệ**: Ném `out_of_range("Index is invalid!")` nếu chỉ số không hợp lệ.
  - **Độ phức tạp**: O(n).

- `bool updateText(int index, string newRawText)`
  - **Chức năng**: Cập nhật bản ghi bằng chuỗi mới, giữ nguyên `id`, cập nhật lại các thuộc tính khác của `record`. Cần phải tiền xử lý chuỗi mới.
  - **Ngoại lệ**: Ném `out_of_range("Index is invalid!")` nếu chỉ số không hợp lệ.
  - **Độ phức tạp**: O(n).

- `void setEmbeddingFunction(SinglyLinkedList<float>* (*newEmbeddingFunction)(const string&))`
  - **Chức năng**: Thay đổi hàm nhúng được sử dụng.
  - **Ngoại lệ**: Không có.
  - **Độ phức tạp**: O(1).

- `void forEach(void (*action)(SinglyLinkedList<float>&, int, string&))`
  - **Chức năng**: Duyệt toàn bộ bản ghi trong kho và áp dụng hàm `action`.
  - **Ngoại lệ**: Không có.
  - **Độ phức tạp**: O(n).

- `double cosineSimilarity(const SinglyLinkedList<float>& v1, const SinglyLinkedList<float>& v2)`
  - **Chức năng**: Tính độ tương đồng cosine giữa hai vector.
  - **Yêu cầu**: tính và trả về giá trị `cosin` giữa 2 vector được truyền vào theo công thức:

$$\cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|}$$

.

- **Độ phức tạp**: O(d).

- `double l1Distance(const SinglyLinkedList<float>& v1, const SinglyLinkedList<float>& v2)`
  - **Chức năng**: Tính khoảng cách Manhattan giữa hai vector.
  - **Yêu cầu**: tính và trả về khoảng cách Manhattan giữa 2 vector theo công thức:

$$d(\vec{A}, \vec{B}) = \sum_{i=1}^{d} |A_i - B_i|$$

    Với d là số chiều của vector.
  - **Ngoại lệ**: Không có.
  - **Độ phức tạp**: O(d).

- `double l2Distance(const SinglyLinkedList<float>& v1, const SinglyLinkedList<float>& v2)`
  - **Chức năng**: Tính khoảng cách Euclidean giữa hai vector.
  - **Yêu cầu**: tính và trả về khoảng cách Euclidean giữa 2 vector theo công thức:

$$d(\vec{A}, \vec{B}) = \sqrt{\sum_{i=1}^{d} (A_i - B_i)^2}$$

    Với d là só chiều của vector.
  - **Ngoại lệ**: Không có.
  - **Độ phức tạp**: O(d).

- `int findNearest(const SinglyLinkedList<float>& query, metric = "cosine")`
  - **Chức năng**: Tìm vector gần nhất với truy vấn, sử dụng độ đo `metric` để tính sự tương đồng giữa các vector. Giá trị `metric` sẽ có 3 giá trị: `cosine, euclidean` và `manhattan`.
  - **Ngoại lệ**: Ném `metric_error()` nếu không thuộc các `metric` trên.
  - **Độ phức tạp**: O(n × d).

- `int* topKNearest(const SinglyLinkedList<float>& query, int k, metric = "cosine")`
  - **Chức năng**: Tìm `k` vector gần nhất với truy vấn, sử dụng độ đo `metric` để tính sự tương đồng giữa các vector. Giá trị `metric` sẽ có 3 giá trị: `cosine, euclidean` và `manhattan`.
  - **Ngoại lệ:**

    * Ném `metric_error()` nếu không thuộc các `metric` trên.
    * Ném `invalid_k_value()` nếu giá trị `k` không hợp lệ.
  - **Trả về:** Mảng động `int` chứa chỉ số của `k` vector gần nhất.
  - **Độ phức tạp**: O(n × d + $n\log(n)$).
  - **Lưu ý:** Cần phải đảm bảo thực hiện đúng độ phức tạp.

4 Yêu cầu và chấm điểm

4.1 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: `utils.h`, `main.cpp`, `main.h`, `VectorStore.h`, `VectorStore.cpp`. Sinh viên chỉ nộp 2 file, đó là `VectorStore.h` và `VectorStore.cpp`. Do đó, không được phép chỉnh sửa file `main.h` khi kiểm tra chương trình.
3. Sinh viên sử dụng lệnh sau để biên dịch:
   `g++ -o main main.cpp VectorStore.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 `VectorStore.cpp`, thêm tùy chọn `-std=c++17`, và `-I .`
4. 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.
5. Chỉnh sửa file `VectorStore.h` và `VectorStore.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 `VectorStore.h` chỉ được phép có đúng một dòng `#include "main.h"`, và trong file `VectorStore.cpp` chỉ được phép có một dòng `#include "VectorStore.h"`. Ngoài hai dòng này, không được phép thêm bất kỳ lệnh `#include` nào khác trong các file này.
- Sinh viên không được sử dụng lệnh `#define TESTING` trong 2 file được yêu cầu chỉnh sửa.

6. 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.
7. Sinh viên phải thiết kế và sử dụng các cấu trúc dữ liệu đã học.
8. Sinh viên bắt buộc phải giải phóng toàn bộ vùng nhớ cấp phát động khi chương trình kết thúc.

4.2 Thời hạn nộp bài

Thời hạn **theo thông báo trên LMS**. Sinh viên vui lòng nộp bài lên hệ thống trước thời hạn thông báo. Sinh viên tự chịu trách nhiệm nếu xuất hiện các lỗi trên hệ thống do nộp gần sát giờ quy định.

4.3 Chấm điểm

Toàn bộ mã nguồn của sinh viên sẽ được chấm trên bộ testcases ẩn, điểm được tính theo từng yêu cầu, cụ thể:

- Hiện thực danh sách mảng động: **3 điểm**.
- Hiện thực danh sách liên kết đơn: **3 điểm**.
- Hiện thực danh sách VectorStore: **4 điểm**.

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.

——————————HẾT——————————