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 VECTORSTORE USING LIST

HO CHI MINH CITY, 09/2025

ASSIGNMENT'S SPECIFICATION

Version 1.0

1 Assignment's outcome

After completing this assignment, students will be able to:

- Demonstrate proficiency in Object-Oriented Programming (OOP).
- Develop and implement list-based data structures.
- Apply list-based data structures to implement a VectorStore.

2 Introduction

In Assignment 1, students are required to implement a **VectorStore** using two fundamental data structures: the **Singly Linked List (SinglyLinkedList)** and the **Dynamic Array List (ArrayList)**. Each vector is represented as a singly linked list, while the entire collection of vectors is managed by a dynamic array list. This design simulates the storage and retrieval mechanisms of vectors in modern database systems.

The use of a singly linked list allows students to practice operations on dynamic linear data structures, as well as to gain a deeper understanding of memory allocation and pointer management. The dynamic array list is incorporated to demonstrate how a resizable array functions when its capacity needs to be expanded, thereby reinforcing students' ability to implement insertion, deletion, and index-based access operations.

The **VectorStore** is a highly practical technology widely applied in real-world scenarios, including semantic search systems, content recommendation, and knowledge management. Through this assignment, students not only strengthen their skills in Object-Oriented Programming (OOP) and consolidate their knowledge of linear data structures, but also engage with the fundamental concept of building a basic **VectorStore** application. This will serve as a foundation for understanding how vector-based data is managed and utilized in modern systems.

3 Description

3.1 List-Based Data Structures

3.1.1 Dynamic Array List - ArrayList

A Dynamic Array List (ArrayList) is a linear data structure that allows storing and managing a collection of elements of the same type within an array whose size can be adjusted dynamically. Unlike a static array with a fixed capacity, an ArrayList automatically expands its capacity when the number of elements exceeds the current limit. This mechanism is typically implemented by allocating a new array with larger capacity (often double the previous size) and copying all existing elements into the new array.

The dynamic array list efficiently supports random access operations by index with a time complexity of $O(1)$. However, insertion or deletion at arbitrary positions generally incurs a time complexity of $O(n)$, as elements must be shifted to maintain order.

Attributes of the `ArrayList` Class

The `ArrayList<T>` class is designed with the following fundamental attributes to manage and organize data within a dynamic array:

- `T* data`: A pointer to the dynamically allocated memory used to store the elements of the list. This memory block can be reallocated with larger capacity when the number of elements exceeds the current limit.
- `int capacity`: The current storage capacity of the dynamic array, representing the maximum number of elements that `data` can hold at a given time. When the number of elements exceeds `capacity`, the class automatically expands the memory block according to the growth strategy.
- `int count`: The actual number of elements currently stored in the list. This value is always less than or equal to `capacity` and is updated after each insertion or deletion.

`private` Method

- `void ensureCapacity(int cap)`
  - **Function**: Ensures that the dynamic array has sufficient capacity to store at least `cap` elements. If `cap` exceeds the current `capacity`, the `capacity` is increased by a factor of 1.5, a new memory block is allocated, and all existing elements are copied into it.

- **Exception**: None.
- **Complexity**: O(n).

`public` **Methods**

- `ArrayList(int initCapacity = 10)`
  - **Function**: Initializes an empty dynamic array list with an initial capacity of `initCapacity`.
  - **Exception**: None.
  - **Complexity**: O(1).
- `ArrayList(const ArrayList<T>& other)`
  - **Function**: Constructs a new list by copying all elements from the list `other`.
  - **Exception**: None.
  - **Complexity**: O(n).
- `~ArrayList()`
  - **Function**: Releases all memory allocated for the list to avoid memory leaks.
  - **Exception**: None.
  - **Complexity**: O(1).
- `ArrayList<T>& operator=(const ArrayList<T>& other)`
  - **Function**: Assigns all elements from `other` to the current list.
  - **Exception**: None.
  - **Complexity**: O(n).
- `void add(T e)`
  - **Function**: Appends element `e` to the end of the array.
  - **Exception**: None.
  - **Complexity**: Amortized O(1); worst case O(n) if reallocation is required.
- `void add(int index, T e)`
  - **Function**: Inserts element `e` at position `index`, shifting subsequent elements one position to the right.
  - **Exception**: Throws `out_of_range("Index is invalid!")` if `index` is invalid.
  - **Complexity**: O(n).
- `T removeAt(int index)`
  - **Function**: Removes and returns the element at position `index`, shifting subsequent elements one position to the left.

  - **Exception**: Throws `out_of_range("Index is invalid!")` if `index` is invalid.
  - **Complexity**: O(n).

- `bool empty()`
  - **Function**: Checks whether the list is empty.
  - **Exception**: None.
  - **Complexity**: O(1).

- `int size()`
  - **Function**: Returns the number of elements currently stored in the list.
  - **Exception**: None.
  - **Complexity**: O(1).

- `void clear()`
  - **Function**: Removes all elements from the list, resets `count` to 0, and restores `capacity` to 10.
  - **Exception**: None.
  - **Complexity**: O(1).

- `T& get(int index)`
  - **Function**: Returns a reference to the element at position `index`.
  - **Exception**: Throws `out_of_range("Index is invalid!")` if `index` is invalid.
  - **Complexity**: O(1).

- `void set(int index, T e)`
  - **Function**: Reassigns the element at position `index` to value `e`.
  - **Exception**: Throws `out_of_range("Index is invalid!")` if `index` is invalid.
  - **Complexity**: O(1).

- `int indexOf(T item)`
  - **Function**: Returns the index of the first occurrence of `item`. If not found, returns -1.
  - **Exception**: None.
  - **Complexity**: O(n).

- `bool contains(T item)`
  - **Function**: Checks whether the list contains an element equal to `item`.
  - **Exception**: None.
  - **Complexity**: O(n).

- `string toString(string (*item2str)(T&) = 0)`
  - **Function**: Returns the string representation of the entire list. If the function pointer `item2str` is provided, it will be used to convert each element to a string.
  - **Exception**: None.
  - **Complexity**: O(n).
  - **Output format:** `[<element 1>, <element 2>, <element 3>, ...]`.

  Example 3.1

  If the array contains: $\{1, 2, 3, 4, 5\}$, and the function pointer provided converts integers to strings.

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

- `Iterator begin()`
  - **Function**: Returns an iterator pointing to the first element of the list. The `Iterator` class will be described below.
  - **Exception**: None.
  - **Complexity**: O(1).
- `Iterator end()`
  - **Function**: Returns an iterator pointing to the position after the last element of the list. The `Iterator` class will be described below.
  - **Exception**: None.
  - **Complexity**: O(1).

Inner class: Iterator

`Iterator` is an inner class of `ArrayList<T>` that supports sequential traversal of the elements in the dynamic array list. Each `Iterator` maintains a current index in the list and provides access to the element at that position, as well as the ability to advance to the next element. Students may refer to iterator concepts at 1, 2.

By convention, `begin()` returns an iterator pointing to the first element (index 0), and `end()` returns an iterator pointing to the position *after* the last element (index equal to `count`). Two iterators are considered different if either their `pList` pointers differ or their `cursor` indices differ.

Attributes

- `int cursor`: The current index in the list. Valid within the range $[0, \texttt{count}]$. When `cursor`

`== count`, the iterator is at the `end()` position and cannot be dereferenced.

- `ArrayList<T>* pList`: A pointer to the dynamic array list being traversed.

Methods

- `Iterator(ArrayList<T>* pList = nullptr, int index = 0)`
  - **Function**: Initializes an iterator pointing to the array `pList` at the given `index`. Students must verify the validity of `index` with respect to the list `pList`.
  - **Exception**: Throws `out_of_range("Index is invalid!")` if `index` is outside the valid range.
  - **Complexity**: $O(1)$.
- `Iterator& operator=(const Iterator& other)`
  - **Function**: Assigns the state from the iterator `other` to the current iterator.
  - **Exception**: None.
  - **Complexity**: $O(1)$.
- `T& operator*()`
  - **Function**: Returns a reference to the element at the position `cursor` in `pList`. Dereferencing at `end()` is invalid.
  - **Exception**: Throws `out_of_range("Iterator is out of range!")` if `cursor` is invalid.
  - **Complexity**: $O(1)$.
- `bool operator!=(const Iterator& other)`
  - **Function**: Compares two iterators for inequality. They are considered different if their `pList` differ or their `cursor` indices differ.
  - **Exception**: None.
  - **Complexity**: $O(1)$.
- `Iterator& operator++()`
  - **Function**: Advances the iterator to the next element (prefix, `++it`).
  - **Exception**: Throws `out_of_range("Iterator cannot advance past end!")` if `cursor` is already at `count`.
  - **Complexity**: $O(1)$.
- `Iterator operator++(int)`
  - **Function**: Advances the iterator to the next element (postfix, `it++`) and returns a **copy** of the old state.

  - **Exception**: Throws `out_of_range("Iterator cannot advance past end!")` if `cursor` is already at `count`.
  - **Complexity**: $O(1)$.

- `Iterator& operator--()`
  - **Function**: Moves the iterator to the previous element (prefix, `--it`). If the iterator is currently at `end()`, this operation moves it to the last element.
  - **Exception**: Throws `out_of_range("Iterator cannot move before begin!")` if already at the first element.
  - **Complexity**: $O(1)$.
- `Iterator operator--(int)`
  - **Function**: Moves the iterator to the previous element (postfix, `it--`) and returns a **copy** of the old state.
  - **Exception**: Throws `out_of_range("Iterator cannot move before begin!")` if already at the first element.
  - **Complexity**: $O(1)$.

3.1.2 Singly Linked List - SinglyLinkedList

A Singly Linked List (abbreviated as `SinglyLinkedList`) is a linear data structure in which elements are stored in the form of nodes. Each node consists of two components: the data field and a pointer (`next`) that references the subsequent node in the list. The last node of the list has its `next` pointer set to `nullptr`, indicating the end of the list.

Unlike static arrays or dynamic array lists, a `SinglyLinkedList` does not require contiguous memory allocation for all its elements. Instead, each element is individually allocated and linked together through pointers. As a result, insertion and deletion operations at the head or in the middle of the list are more flexible, with an average cost of $O(1)$ for operations at the head of the list and $O(n)$ for operations at arbitrary positions.

However, due to its nature of supporting traversal in only one direction (from head to tail), random access to an element at an arbitrary position incurs a cost of $O(n)$. This makes the singly linked list less efficient than the dynamic array list in scenarios where direct index-based access is frequently required.

Class `Node`

Attributes:

- `T data`: The data stored in the node.
- `Node* next`: A pointer to the next node in the list.

Attributes of `SinglyLinkedList`

- `Node* head`: Pointer to the first node in the list.
- `Node* tail`: Pointer to the last node in the list.
- `int count`: The number of elements currently in the list.

`public` Methods of `SinglyLinkedList`

- `SinglyLinkedList()`
  - **Function**: Initializes an empty singly linked list.
  - **Exception**: None.
  - **Complexity**: O(1).
- `~SinglyLinkedList()`
  - **Function**: Releases all allocated nodes to avoid memory leaks.
  - **Exception**: None.
  - **Complexity**: O(n).
- `void add(T e)`
  - **Function**: Adds element `e` to the end of the list.
  - **Exception**: None.
  - **Complexity**: O(n).
- `void add(int index, T e)`
  - **Function**: Inserts element `e` at position `index`.
  - **Exception**: Throws `out_of_range("Index is invalid!")` if `index` is invalid.
  - **Complexity**: O(n).
- `T removeAt(int index)`
  - **Function**: Removes and returns the element at position `index`.
  - **Exception**: Throws `out_of_range("Index is invalid!")` if `index` is invalid.
  - **Complexity**: O(n).
- `bool removeItem(T item)`
  - **Function**: Removes the first node whose value equals `item`. Returns `true` if removal is successful, `false` otherwise.

  - **Exception**: None.
  - **Complexity**: O(n).

- `bool empty()`
  - **Function**: Checks whether the list is empty.
  - **Exception**: None.
  - **Complexity**: O(1).

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

- `void clear()`
  - **Function**: Removes all nodes from the list.
  - **Exception**: None.
  - **Complexity**: O(n).

- `T& get(int index)`
  - **Function**: Returns a reference to the data at position `index` in the list.
  - **Exception**: Throws `out_of_range("Index is invalid!")` if `index` is invalid.
  - **Complexity**: O(n).

- `int indexOf(T item)`
  - **Function**: Returns the index of the first element equal to `item`. Returns -1 if not found.
  - **Exception**: None.
  - **Complexity**: O(n).

- `bool contains(T item)`
  - **Function**: Checks whether the list contains an element equal to `item`.
  - **Exception**: None.
  - **Complexity**: O(n).

- `string toString(string (*item2str)(T&) = 0)`
  - **Function**: Returns a string representation of the entire list. If the function pointer `item2str` is provided, it will be used to convert each element into a string. Otherwise, the default representation of the data type will be used.
  - **Exception**: None.

  - **Complexity**: O(n).
  - **Output format:** `[<element 1>]->[<element 2>]->[<element 3>]....`

> **Example 3.2**
>
> If the list contains: $\{1, 2, 3, 4, 5\}$, and the function pointer provided converts integers into strings.
>
> **Result of toString:** `[1]->[2]->[3]->[4]->[5]`

- `Iterator begin()`
  - **Function**: Returns an iterator pointing to the first element of the list. The `Iterator` class will be described below.
  - **Exception**: None.
  - **Complexity**: O(1).
- `Iterator end()`
  - **Function**: Returns an iterator pointing to the position after the last element of the list. The `Iterator` class will be described below.
  - **Exception**: None.
  - **Complexity**: O(1).

Inner class: Iterator

`Iterator` is an inner class of `SinglyLinkedList<T>` used to sequentially traverse the elements of the singly linked list. Each `Iterator` maintains a pointer to the current node, allowing access to its data and advancing to the next node via the increment operator (`++`). By convention, `begin()` returns an iterator pointing to the first node, while `end()` returns an iterator whose pointer is `nullptr`, representing the end state.

Attribute

- `Node* current`: A pointer to the current node in the list.

Methods

- `Iterator(Node* node = nullptr)`
  - **Function**: Initializes the iterator to point to the given node in the list.
  - **Exception**: None.
  - **Complexity**: O(1).

- `Iterator& operator=(const Iterator& other)`
  - **Function**: Assigns the current iterator to the state of `other`.
  - **Exception**: None.
  - **Complexity**: O(1).
- `T& operator*()`
  - **Function**: Returns a reference to the data (`data`) of the current node.
  - **Exception**: Throws `out_of_range("Iterator is out of range!")` if `current == nullptr`.
  - **Complexity**: O(1).
- `bool operator!=(const Iterator& other)`
  - **Function**: Compares two iterators for inequality.
  - **Exception**: None.
  - **Complexity**: O(1).
- `Iterator& operator++()`
  - **Function**: Advances the iterator to the next node in the list (prefix, `++it`) and returns the updated iterator.
  - **Exception**: Throws `out_of_range("Iterator cannot advance past end!")` if the iterator is already at the end.
  - **Complexity**: O(1).
- `Iterator operator++(int)`
  - **Function**: Advances the iterator to the next node (postfix, `it++`) and returns a **copy** of the old state.
  - **Exception**: Throws `out_of_range("Iterator cannot advance past end!")` if the iterator is already at the end.
  - **Complexity**: O(1).

3.2 VectorStore

3.2.1 Overview of VectorStore

A **VectorStore** is a specialized data repository used for storing and querying high-dimensional vectors. In computer science, vectors are commonly used to represent information (e.g., a sentence, a dialogue, or an image) as a sequence of real numbers. The goal of a VectorStore is to allow simultaneous storage of multiple vectors and to support query operations based on the **similarity** between the query vector and the stored vectors.

Mechanism of Operation: A **VectorStore** provides fundamental mechanisms to enable processing, storage, and query functionalities, specifically:

- **Embedding:** Raw data (e.g., a text string) is first transformed into a real-valued vector through an *embedding function*. The resulting vector typically has a fixed dimensionality for each type of VectorStore.
  Example: The sentence "White cat" may be mapped into the vector $[0.2, -0.1, 0.5, 0.0, \dots]$.
- **Storage:** Once generated, the vector is stored in the VectorStore together with its descriptive metadata, such as identifier, raw text, timestamp, etc.
  Example: id = 1, rawText = "White cat", vector = $[0.2, -0.1, 0.5, 0.0, \dots]$.
- **Iteration and Batch Processing:** The VectorStore allows traversal over all stored vectors to apply common operations, such as normalization, printing, or statistical analysis.
- **Similarity Search:** When receiving a query, the input is also transformed into a real-valued vector. The VectorStore then compares this query vector against all stored vectors using a similarity metric. Finally, it returns the nearest vector or a list of the top k nearest vectors.
  Example: The query "Black dog" is embedded into a vector, compared with vectors in the store, and the VectorStore returns the top 3 most similar sentences.

The VectorStore provides a practical example of combining **data storage** and **search algorithms**. It illustrates that instead of directly comparing raw data (e.g., text), data can be transformed into numerical form (high-dimensional vectors), enabling similarity computation through basic mathematical operations such as dot product, vector norm, or distance measures. This mechanism forms the foundation of many modern systems, including semantic search engines, recommendation systems, and knowledge-based question answering systems.

3.2.2 The VectorStore Class

The `VectorStore` class simulates a storage system for multi-dimensional vectors together with their associated metadata. Its objective is to help students become familiar with the basic mechanism of semantic search systems: raw data (`rawText`) is mapped into a high-dimensional real-valued vector through an *embedding function* (`embeddingFunction`), after which the vector is stored and can later be queried based on similarity.

Structure `VectorRecord`

To ensure synchronization during storage, the `VectorStore` class uses an intermediate structure called `VectorRecord`. Each `VectorRecord` represents one entry in the store, consisting of:

- `int id`: The unique identifier of the vector.
- `string rawText`: The original string before embedding.
- `int rawLength`: The length of the `rawText`.
- `SinglyLinkedList<float>* vector`: A pointer to a singly linked list storing the high-dimensional real-valued vector.

Attributes of the `VectorStore` Class

- `ArrayList<VectorRecord*> records`: A dynamic array list containing all vector records and their associated metadata.
- `int dimension`: The fixed dimensionality of all vectors in the store.
- `int count`: The number of records currently stored.
- `SinglyLinkedList<float>* (*embeddingFunction)(const string&)`: A function pointer used to map a raw text string into a high-dimensional vector.

`public` **Methods of the `VectorStore` Class**

- `VectorStore(int dimension = 512, SinglyLinkedList<float>* (*embeddingFunction)(cons string&))`
  - **Function**: Initializes an empty vector store with a fixed dimensionality and assigns the initial embedding function.
  - **Exception**: None.
  - **Complexity**: O(1).
- `int size()`
  - **Function**: Returns the number of vectors currently stored.
  - **Exception**: None.
  - **Complexity**: O(1).
- `bool empty()`
  - **Function**: Checks whether the store is empty.
  - **Exception**: None.
  - **Complexity**: O(1).

- `void clear()`
  - **Function**: Removes all vectors and their associated metadata.
  - **Exception**: None.
  - **Complexity**: O(n).
- `SinglyLinkedList<float>* preprocessing(string rawText)`
  - **Function**: Preprocesses the input text and converts it into a high-dimensional vector.
  - **Requirements**:
    1. Invoke `embeddingFunction` to map `rawText` into a vector.
    2. Check the dimensionality of the resulting vector:
       * If greater than `dimension`, truncate excess elements at the end.
       * If smaller than `dimension`, append zeros (post-padding) until the vector reaches the required size.
  - **Exception**: None.
  - **Complexity**: O(d).
- `void addText(string rawText)`
  - **Function**: Adds a new record to the store from the raw text. Must invoke `preprocessing` to transform the input `rawText`.
  - **Exception**: None.
  - **Complexity**: O(n).
- `SinglyLinkedList<float>& getVector(int index)`
  - **Function**: Retrieves a reference to the vector at the specified `index`.
  - **Exception**: Throws `out_of_range("Index is invalid!")` if the index is invalid.
  - **Complexity**: O(1).
- `string getRawText(int index)`
  - **Function**: Retrieves the raw text associated with the vector at `index`.
  - **Exception**: Throws `out_of_range("Index is invalid!")` if the index is invalid.
  - **Complexity**: O(1).
- `int getId(int index)`
  - **Function**: Retrieves the `id` of the vector at `index`.
  - **Exception**: Throws `out_of_range("Index is invalid!")` if the index is invalid.
  - **Complexity**: O(1).

- `bool removeAt(int index)`
  - **Function**: Removes the vector and its metadata at `index`.
  - **Requirement**: Release the dynamically allocated vector and delete the record from the `VectorStore`.
  - **Exception**: Throws `out_of_range("Index is invalid!")` if the index is invalid.
  - **Complexity**: O(n).
- `bool updateText(int index, string newRawText)`
  - **Function**: Updates the record with new text while preserving the `id`, and refreshes other attributes of the record. The new text must be preprocessed.
  - **Exception**: Throws `out_of_range("Index is invalid!")` if the index is invalid.
  - **Complexity**: O(n).
- `void setEmbeddingFunction(SinglyLinkedList<float>* (*newEmbeddingFunction)(const string&))`
  - **Function**: Changes the embedding function being used.
  - **Exception**: None.
  - **Complexity**: O(1).
- `void forEach(void (*action)(SinglyLinkedList<float>&, int, string&))`
  - **Function**: Iterates through all records in the store and applies the function `action`.
  - **Exception**: None.
  - **Complexity**: O(n).
- `double cosineSimilarity(const SinglyLinkedList<float>& v1, const SinglyLinkedList<float>& v2)`
  - **Function**: Computes the cosine similarity between two vectors.
  - **Requirement**: Compute and return the cosine similarity using the formula:

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

  - **Complexity**: O(d).
- `double l1Distance(const SinglyLinkedList<float>& v1, const SinglyLinkedList<float>& v2)`
  - **Function**: Computes the Manhattan distance between two vectors.

- **Requirement**: Compute and return the Manhattan distance using the formula:

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

  where d is the dimensionality of the vector.
- **Exception**: None.
- **Complexity**: O(d).

- `double l2Distance(const SinglyLinkedList<float>& v1, const SinglyLinkedList<float>& v2)`
  - **Function**: Computes the Euclidean distance between two vectors.
  - **Requirement**: Compute and return the Euclidean distance using the formula:

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

    where d is the dimensionality of the vector.
  - **Exception**: None.
  - **Complexity**: O(d).

- `int findNearest(const SinglyLinkedList<float>& query, metric = "cosine")`
  - **Function**: Finds the nearest vector to the query using the specified `metric`. Supported values: `cosine`, `euclidean`, and `manhattan`.
  - **Exception**: Throws `metric_error()` if the metric is invalid.
  - **Complexity**: O(n × d).

- `int* topKNearest(const SinglyLinkedList<float>& query, int k, metric = "cosine")`
  - **Function**: Finds the `k` nearest vectors to the query using the specified `metric`. Supported values: `cosine`, `euclidean`, and `manhattan`.
  - **Exception**:
    * Throws `metric_error()` if the metric is invalid.
    * Throws `invalid_k_value()` if `k` is not valid.
  - **Return**: A dynamically allocated array of `int` containing the indices of the `k` nearest vectors.
  - **Complexity**: O(n × d + $n\log(n)$).
  - **Note**: Students must ensure that the time complexity requirement is satisfied.

4 Requirements and Grading

4.1 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: `utils.h`, `main.cpp`, `main.h`, `VectorStore.h`, `VectorStore.cpp`, and a folder containing sample outputs. Students only need to submit two files: `VectorStore.h` and `VectorStore.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 VectorStore.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 `VectorStore.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 `VectorStore.h` and `VectorStore.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 `VectorStore.h` must contain exactly one line `#include "main.h"`, and the file `VectorStore.cpp` must contain exactly one line `#include "VectorStore.h"`. Apart from these, no other `#include` statements are allowed in these files.
   - Students are not allowed to use the directive `#define TESTING` in the two files that are required to be modified.

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.

4.2 Submission Deadline

The deadline is **as announced on LMS**. Students must submit their assignments to the system before the specified deadline. Students are fully responsible for any issues arising from submissions made too close to the deadline.

4.3 Grading

All student source code will be evaluated against a hidden set of test cases. The final score will be computed based on the following criteria:

- Implementation of the dynamic array list: **3 points**.
- Implementation of the singly linked list: **3 points**.
- Implementation of the VectorStore: **4 points**.

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.

———————THE END———————