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

INVENTORY MANAGEMENT USING LISTS

Ho Chi Minh City, 02/2025

ASSIGNMENT'S SPECIFICATION

Version 1.1

1 Introduction

1.1 Objectives and Tasks

The objective of the assignment in this course is to help students develop and apply data structures to build real-world applications. In this semester, the chosen application theme is product inventory management. The assignments are interrelated (for example, Project 2 will inherit from Project 1), so students must ensure continuity among the assignments.

The tasks for Project 1 are presented as follows:

- **Develop** the list data structure, specifically an array-based list and a linked list (the two classes **XArrayList** and **DLinkedList** in the provided source code).
- **Utilize** the developed lists to build a library of classes supporting inventory management, including:
    1. `List1D`: Implementation of a one-dimensional list that stores values.
    2. `List2D`: Implementation of a two-dimensional list (matrix) that stores product attributes.
    3. `InventoryManager`: Manages product inventory information with components including the product attribute matrix, product names list, and inventory quantities list.

1.2 Methodology

1. Preparation: **Download** and **study** the provided source code. **Note**, it is mandatory that students compile the source code in the projects using **C++17**. The compiler has been tested with **g++**; it is recommended that students set up an environment using **g++**.
2. **Develop** the lists: Implement the classes **XArrayList** and **DLinkedList** in the folder **/include/list**.
3. **Utilize** the lists developed above to implement the application classes **List1D**, **List2D**, and **InventoryManager**.
4. **Test Run**: Verify that the program passes the provided sample test cases.

5. **Submission**: Submit the assignment on the system before the deadline.

1.3 Expected Outcomes of Project 1

After completing this major assignment, students will have reviewed and mastered:

- Advanced usage of the C/C++ programming language.
- Development of the list data structure, with two versions: array-based and linked-based.
- Selection and use of lists to develop the classes **List1D**, **List2D**, and **InventoryManager**.

1.4 Grading Components and Evaluation Method

1. List Implementation: (**60% of the points**); including the following files in the folder **/include/list**
   - **XArrayList.h**
   - **DLinkedList.h**
2. Implementation of data set and data loading classes: (**40% of the points**); including the following file in the folder **/include/app**:
   - **inventory.h**

Students will be evaluated using sample test cases provided at submission. The assignment, after being submitted to the system, will be graded based on hidden test cases.

2 Guidelines

2.1 Implmenting List Data Structures

2.1.1 Design Approach for Data Structures

In this course, the data structures are designed following a consistent template; namely:

1. Each data structure will have one or more abstract classes defining the operations supported by the structure. In the case of lists, this is the **IList** class located in the **/include/list** directory.

2. In practice, data structures should be able to contain elements of **any type**, such as **student objects**, **pointers to student objects**, or even just an **int**. Therefore, all data structures in this course utilize **templates** (in C++) to parameterize the element type.
3. All data structures are designed to encapsulate data and high-level details, separating the roles of **library developer** and **library user**. As such, all data structures, including lists, have been enhanced with an iteration feature (**iterator**) to facilitate element traversal from the user's perspective.

2.1.2 Overview of the List Data Structure

The list data structure in this assignment is designed with the following classes:

- The `IList` class, see Figure 1: this class defines a set of APIs supported by the list, whether implemented by arrays or by links. `IList` is the parent class of both `XArrayList` and `DLinkedList`. Some detailed points:
  - `IList` uses **templates** to parameterize the element type, allowing the list to contain elements of any type.
  - All APIs in `IList` are "pure virtual methods", meaning that derived classes from `IList` must override all these methods, supporting dynamic binding (polymorphism).
- The `XArrayList` and `DLinkedList` classes: these classes inherit from `IList` and implement all the APIs defined in `IList`, using arrays (as in `XArrayList`) and using links (as in `DLinkedList`).

Below is a description of each pure virtual method in `IList`:

- `virtual ~IList() {};`
  - Virtual destructor, used to ensure that the destructors of derived classes are called when an object is deleted through a base class pointer.
- `virtual void add(T e) = 0;`
  - Adds element `e` to the end of the list.
  - **Parameter**: `T e` — the element to add.
- `virtual void add(int index, T e) = 0;`
  - Inserts element `e` at position `index` in the list.
  - **Parameters**:

```cpp
template<typename T>
class IList {
public:
virtual ~IList(){};
    virtual void    add(T e)=0;
    virtual void    add(int index, T e)=0;
    virtual T       removeAt(int index)=0;
    virtual bool    removeItem(T item, void (*removeItemData)(T)=0)=0;
    virtual bool    empty()=0;
    virtual int     size()=0;
    virtual void    clear()=0;
    virtual T&      get(int index)=0;
    virtual int     indexOf(T item)=0;
    virtual bool    contains(T item)=0;
    virtual string  toString(string (*item2str)(T&)=0 )=0;
};
```

Figure 1: `IList<T>`: Abstract class defining APIs for the list.

  * `int index` — the position where the element should be inserted.
  * `T e` — the element to insert.

- `virtual T removeAt(int index) = 0;`
  - **Description**: Removes and returns the element at position `index`.
  - **Parameter**: `int index` — the position of the element to remove.
  - **Return Value**: Returns the element at position `index`.
  - **Exception**: Throws `std::out_of_range` if `index` is invalid.

- `virtual bool removeItem(T item, void (*removeItemData)(T) = 0) = 0;`
  - **Description**: Removes the element `item` stored in the list.
  - **Parameters**:
    * `T item` — the element to remove.
    * `void (*removeItemData)(T)` — a function pointer to handle the removal of dynamic memory associated with the element, if necessary.
  - **Return Value**: Returns `true` if the element is found and removed, `false` otherwise.

- `virtual bool empty() = 0;`
  - **Description**: Checks if the list is empty.
  - **Return Value**: Returns `true` if the list is empty, `false` otherwise.

- `virtual int size() = 0;`
  - **Description**: Returns the number of elements currently stored in the list.
  - **Return Value**: The number of elements in the list.

- `virtual void clear() = 0;`
  - **Description**: Removes all elements from the list.
- `virtual int indexOf(T item) = 0;`
  - **Description**: Returns the index of the first occurrence of `item` in the list, or `-1` if `item` is not in the list.
  - **Parameter**: `T item` — the element to search for.
  - **Return Value**: The index of the first occurrence of `item`, or `-1` if not found.
- `virtual string toString(string (*item2str)(T&) = 0) = 0;`
  - **Description**: Converts the list to a string, with elements separated by a comma.
  - **Parameter**: `string (*item2str)(T&)` — a function pointer that converts an element of type `T` to a string.
  - **Return Value**: A string representing the list.

2.1.3 Array List

`XArrayList<T>` is an implementation of a list using an array to store elements of type `T`. In principle, `XArrayList<T>` must proactively maintain an array large enough to contain the elements within it. The manipulation of elements only pertains to APIs such as `add`, `remove`, and `removeItem`; therefore, when implementing these APIs, it is necessary to check the size of the current array to ensure sufficient space to store the elements.

In addition to methods for the APIs in `IList`, `XArrayList<T>` also require additional supporting methods, see in the file **XArrayList.h**, in the **/include/list** directory.

```cpp
template<typename T>
class XArrayList: public IList<T> {
protected:
    T*  data;
    int capacity;
    int count;
public:
    XArrayList(int capacity=10);
    ~XArrayList();

public:
    //Inherit from IList: BEGIN
    void    add(T e);
    void    add(int index, T e);
    T       removeAt(int index);
    bool     removeItem(T item, void (*removeItemData)(T)=0);
    bool     empty();
    int       size();
    void     clear();
    T&       get(int index);
    int       indexOf(T item);
    bool     contains(T item);
    string   toString(string (*item2str)(T&)=0 );
    //Inherit from IList: END
};
```

1. **Attributes:**
   - `T* data`: A dynamic array that stores the elements of the list.
   - `int capacity`: The current capacity of the dynamic array `data`.
   - `int count`: The number of elements currently in the list.
2. **Constructor and Destructor:**
   - `XArrayList(int capacity)`: Constructor that initializes the list with an initial capacity.
   - ∼`XArrayList()`: Destructor, frees the memory allocated for the dynamic array `data` and its elements.
3. **Methods:**
   - **void add(T e)**
     - **Function**: Adds an element `e` to the end of the list.
     - **Exceptions**: None.
   - **void add(int index, T e)**
     - **Function**: Adds element `e` at the specified position `index` in the list.
     - **Exceptions**: If `index` is invalid (out of range `[0, count]`), throws `out_of_range("Index is out of range!")`.
   - **T removeAt(int index)**
     - **Function**: Removes the element at position `index` and returns the removed element.
     - **Exceptions**: If `index` is invalid (out of range `[0, count-1]`), throws `out_of_range("Inde`
`is out of range!")`.
   - **bool removeItem(T item, void (*removeItemData)(T) = 0)**
     - **Function**: Removes the first element in the list with a value equal to `item`.

  - **Exceptions**: None.

- **bool empty()**
  - **Function**: Checks if the list is empty.
  - **Exceptions**: None.
- **int size()**
  - **Function**: Returns the number of elements currently in the list.
  - **Exceptions**: None.
- **void clear()**
  - **Function**: Removes all elements in the list and resets the list to its initial state.
  - **Exceptions**: None.
- **T& get(int index)**
  - **Function**: Returns a reference to the element at position `index`.
  - **Exceptions**: If `index` is invalid (out of range `[0, count-1]`), throws `out_of_range("Inde` `is out of range!")`.
- **int indexOf(T item)**
  - **Function**: Returns the index of the first element with a value equal to `item` in the list, or `-1` if not found.
  - **Exceptions**: None.
- **bool contains(T item)**
  - **Function**: Checks if the list contains an element with a value equal to `item`.
  - **Exceptions**: None.
- **string toString(string (*item2str)(T&) = 0)**
  - **Function**: Returns a string representation of the elements in the list.
  - **Exceptions**: None.

2.1.4 Doubly Linked List

`DLinkedList<T>` is an implementation of a list using two links: **next** and **prev**. From the user's perspective, they do not need to be aware of the internal implementation details. Therefore, they are not required to be aware of the links. To achieve this, `DLinkedList<T>` needs a "**node**" object, which contains user data as well as two links: one to the next element and one to the previous element.

In addition to methods for the APIs in `IList`, `DLinkedList<T>` also require additional supporting methods, see in the file **DLinkedList.h**, in the **/include/list** directory.

```cpp
template<class T>
class DLinkedList: public IList<T> {
public:
    class Node; //Forward declaration
protected:
    Node *head;
    Node *tail;
    int count;

public:
    DLinkedList();
    ~DLinkedList();

    //Inherit from IList: BEGIN
    void    add(T e);
    void    add(int index, T e);
    T       removeAt(int index);
    bool    removeItem(T item, void (*removeItemData)(T)=0);
    bool    empty();
    int      size();
    void    clear();
    T&      get(int index);
    int      indexOf(T item);
    bool    contains(T item);
    string  toString(string (*item2str)(T&)=0 );
    //Inherit from IList: END

public:
    class Node{
    public:
        T data;
        Node *next;
        Node *prev;
        friend class DLinkedList<T>;

    public:
        Node(Node* next=0, Node* prev=0);
        Node(T data, Node* next=0, Node* prev=0);
    };
};
```

1. **class Node:**

- `T data`: The data of the node.
- `Node* next`: Pointer to the next node in the list.
- `Node* prev`: Pointer to the previous node in the list.

2. **Constructor and Destructor:**
   - `DLinkedList()`: Constructor that initializes an empty doubly linked list.
   - ∼`DLinkedList()`: Destructor, frees the memory allocated for the nodes in the list.

3. **Methods:**
   - **void add(T e)**
     - **Function**: Adds an element `e` to the end of the list.
     - **Exceptions**: None.
   - **void add(int index, T e)**
     - **Function**: Adds element `e` at the specified position `index` in the list.
     - **Exceptions**: If `index` is invalid (out of range `[0, count]`), throws `out_of_range("Index is out of range!")`.
   - **T removeAt(int index)**
     - **Function**: Removes the element at position `index` and returns the removed element.
     - **Exceptions**: If `index` is invalid (out of range `[0, count-1]`), throws `out_of_range("Inde is out of range!")`.
   - **bool removeItem(T item, void (*removeItemData)(T) = 0)**
     - **Function**: Removes the first element in the list with a value equal to `item`.
     - **Exceptions**: None.
   - **bool empty()**
     - **Function**: Checks if the list is empty.
     - **Exceptions**: None.
   - **int size()**
     - **Function**: Returns the number of elements currently in the list.
     - **Exceptions**: None.
   - **void clear()**
     - **Function**: Removes all elements in the list and resets the list to its initial state.
     - **Exceptions**: None.
   - **T& get(int index)**
     - **Function**: Returns a reference to the element at position `index`.

- **Exceptions**: If `index` is invalid (out of range `[0, count-1]`), throws `out_of_range("Inde`
`is out of range!")`.

- **int indexOf(T item)**
  - **Function**: Returns the index of the first element with a value equal to `item` in the list, or `-1` if not found.
  - **Exceptions**: None.
- **bool contains(T item)**
  - **Function**: Checks if the list contains an element with a value equal to `item`.
  - **Exceptions**: None.
- **string toString(string (*item2str)(T&) = 0)**
  - **Function**: Returns a string representation of the elements in the list.
  - **Exceptions**: None.

2.2 Application of the List Data Structure

2.2.1 Class `List1D<T>`

`List1D<T>` is a class that implements a one-dimensional list used to store an array of values of type `T` via the list data structure. This class provides basic APIs for operating on the list, including obtaining its size, accessing elements, assigning new values, adding elements, and converting the list into a string to facilitate printing.

1. **Constructors and Destructor:**
   - `List1D()`: Constructs an empty list.
   - `List1D(int num_elements)`: Constructs a list with the specified number of elements (initially, each element is initialized with a default value, for example, 0).
   - `List1D(const T* array, int num_elements)`: Constructs a list from an array of values.
   - `List1D(const List1D<T>& other)`: Copy constructor that creates a new list based on the content of an existing list.
   - `virtual ~List1D()`: Destructor that frees the allocated memory.
2. **Member Methods:**
   - `int size() const`: Returns the number of elements currently in the list.
   - `T get(int index) const`: Retrieves the element at position `index`.

- `void set(int index, T value)`: Assigns a new value to the element at position `index`.
- `void add(const T& value)`: Adds an element to the list.
- `string toString() const`: Returns a string representation of the list in the format: $[e_1, e_2, e_3, \ldots, e_n]$, where $e_1, e_2, e_3, \ldots$ are the elements of the list. For example: $[1, 2, 3, 4]$.
- The operator `<<`: Supports printing the list through an `ostream` object in the above format.

```cpp
// -------------------- List1D --------------------
template <typename T>
class List1D
{
private:
    IList<T> *pList;
public:
    List1D();
    List1D(int num_elements);
    List1D(const T *array, int num_elements);
    List1D(const List1D<T> &other);
    virtual ~List1D();

    int size() const;
    T get(int index) const;
    void set(int index, T value);
    void add(const T &value);
    string toString() const;

    friend ostream &operator<<(ostream &os, const List1D<T> &list);
};
```

2.2.2 Class `List2D<T>`

`List2D<T>` is a class that implements a two-dimensional list used to store a matrix of values of type `T` via the list data structure. In the new version of this class, the number of columns is not fixed; each row may contain a different number of elements. Each row of the matrix is represented by a `List1D<T>` object, which supports storing heterogeneous sets of attributes for each product.

1. **Constructors and Destructor:**

- `List2D()`: Constructs an empty matrix.
- `List2D(List1D<T>* array, int num_rows)`: Constructs a matrix from an array of `List1D<T>` objects with the specified number of rows. Since the number of columns in each row may vary, there is no need to specify a fixed number of columns.
- `List2D(const List2D<T>& other)`: Copy constructor that creates a new matrix based on the content of an existing matrix.
- `virtual ~List2D()`: Destructor that frees the memory allocated for the matrix.

2. **Member Methods:**

- `int rows() const`: Returns the number of rows in the matrix.
- `void setRow(int rowIndex, const List1D<T>& row)`: Replaces the row at the specified index `rowIndex` with a new row.
- `T get(int rowIndex, int colIndex) const`: Retrieves the element at the position specified by `rowIndex` and `colIndex`.
- `List1D<T> getRow(int rowIndex) const`: Returns the row at the specified index as a `List1D<T>` object.
- `string toString() const`: Returns a string representation of the matrix in the format:

$$[[e_{11}, e_{12}, \ldots], [e_{21}, e_{22}, \ldots], \ldots, [e_{m1}, e_{m2}, \ldots]]$$

  where each row may have a different number of elements (columns). For example: `[[1, 2], [3, 4, 5], [6]]`.
- The operator `<<`: Supports printing the matrix through an `ostream` object in the above format.

```
// -------------------- List2D --------------------
template <typename T>
class List2D
{
private:
    IList<IList<T> *> *pMatrix;
public:
    List2D();
    List2D(List1D<T>* array, int num_rows);
    List2D(const List2D<T> &other);
    virtual ~List2D();

    int rows() const;
    void setRow(int rowIndex, const List1D<T> &row);
    T get(int rowIndex, int colIndex) const;
    List1D<T> getRow(int rowIndex) const;
    string toString() const;

    friend ostream &operator<<(ostream &os, const List2D<T> &matrix);
};
```

2.2.3 Inventory Management (`InventoryManager`)

`InventoryManager` is a class that manages product inventory and provides APIs for storing and processing product information. This class manages data through three main components:

- `attributesMatrix`: A matrix of product attributes (of type `List2D<InventoryAttribute>`); each row represents the attributes of a product. Note: The number of attributes for each product may vary.
- `productNames`: A list of product names (of type `List1D<string>`).
- `quantities`: A list of inventory quantities for each product (of type `List1D<int>`).

Example 2.1

Suppose there are 3 products in the inventory with the following information:

1. **Product A:**
   - **Attributes:** [("weight", 10), ("height", 156)]
   - **Product Name:** "Product A"
   - **Inventory Quantity:** 50
2. **Product B:**
   - **Attributes:** [("weight", 20), ("depth", 24), ("height", 100)]
   - **Product Name:** "Product B"
   - **Inventory Quantity:** 30
3. **Product C:**
   - **Attributes:** [("color", 2)]
   - **Product Name:** "Product C"
   - **Inventory Quantity:** 20

This information is managed through the three components of the `InventoryManager` class as follows:

- `attributesMatrix` (of type `List2D<InventoryAttribute>`) contains:

$$\begin{aligned}&[[(\text{"weight"}, 10),\ (\text{"height"}, 156)],\\ &\ [(\text{"weight"}, 20),\ (\text{"depth"}, 24),\ (\text{"height"}, 100)],\\ &\ [(\text{"color"}, 2)]]\end{aligned}$$

- `productNames` (of type `List1D<string>`) contains:

$$[\text{"Product A"},\ \text{"Product B"},\ \text{"Product C"}]$$

- `quantities` (of type `List1D<int>`) contains:

$$[50,\ 30,\ 20]$$

The methods to be implemented for the `InventoryManager` class are described in detail as follows:

1. `InventoryManager()`

   Function: Constructs an empty `InventoryManager` object.

2. `InventoryManager(const List2D<InventoryAttribute>& matrix, const List1D<string>& names, const List1D<int>& quantities)`

   Function: Constructs an `InventoryManager` object using the provided attribute matrix, product names list, and quantities list.

3. `InventoryManager(const InventoryManager& other)`

   Function: Creates a new `InventoryManager` object by copying all data from the existing `other` object.

4. `int size() const`

   Function: Returns the number of products currently in the inventory.

5. `List1D<InventoryAttribute> getProductAttributes(int index) const`

   Function: Returns the list of attributes for the product at position `index`.

   Exception: If `index` is invalid (not in the range $[0, \texttt{size()} - 1]$), throws an exception `out_of_range("Index is invalid!")`.

6. `string getProductName(int index) const`

   Function: Returns the name of the product at position `index`.

   Exception: If `index` is invalid, throws an exception `out_of_range("Index is invalid!")`.

7. `int getProductQuantity(int index) const`

   Function: Returns the inventory quantity for the product at position `index`.

   Exception: If `index` is invalid, throws an exception `out_of_range("Index is invalid!")`.

8. `void updateQuantity(int index, int newQuantity)`

   Function: Updates the inventory quantity for the product at position `index` with the new value `newQuantity`.

   Exception: If `index` is invalid, throws an exception `out_of_range("Index is invalid!")`.

9. `void addProduct(const List1D<InventoryAttribute>& attributes, const string& name, int quantity)`

   Function: Adds a new product to the inventory with the specified attributes, name, and quantity.

10. `void removeProduct(int index)`

    Function: Removes the product at position `index` from the inventory.

    Exception: If `index` is invalid, throws an exception `out_of_range("Index is invalid!")`.

11. `List1D<string> query(int attributeIndex, const double &minValue, const double &maxValue, int minQuantity, bool ascending) const`

    Function: Queries and returns a list of product names for which the attribute with the name `attributeName` is within the range `[minValue, maxValue]` and whose inventory quantity is at least `minQuantity`. The results are sorted in ascending order if `ascending` is `true`.

Example: Suppose the first attribute of a product represents weight; if `query(1, 5, 10, 3, true)` is called, the function will return a list of product names with weight between 5 and 10 and with an inventory quantity of at least 3, sorted in ascending order.

12. `void removeDuplicates()`

    Function: Removes duplicate products from the inventory. If the inventory contains two products with identical attributes and names, calling `removeDuplicates()` will retain only one product with its quantity set to the sum of all duplicate products.

    Note: You should implement this method with O(n^2) time complexity to fully passed all the testcase.

13. `static InventoryManager merge(const InventoryManager &inv1, const InventoryManager &inv2)`

    Function: Merges the two inventory objects `inv1` and `inv2` into a single inventory.

14. `void split(InventoryManager &section1, InventoryManager &section2, double ratio) const`

    Function: Splits the current inventory into two sections, storing them in `section1` and `section2` according to the ratio `ratio`. If, after splitting, the number of products in `section1` is not an integer, round it up.

    Example: If `split(sec1, sec2, 0.6)` is called on an inventory of 11 products, then `sec1` will contain 7 products and `sec2` will contain 4 products.

15. `List2D<InventoryAttribute> getAttributesMatrix() const`

    Function: Returns the attribute matrix of all products in the inventory.

16. `List1D<string> getProductNames() const`

    Function: Returns the list of product names in the inventory.

17. `List1D<int> getQuantities() const`

    Function: Returns the list of inventory quantities for all products.

18. `string toString() const`

    Function: Returns a string representing the complete inventory information, including the attribute matrix, product names, and inventory quantities.

    Example: The function returns a string like:

```
InventoryManager[
  AttributesMatrix: [[...], [...], ...],
  ProductNames: ["ProdA", "ProdB", ...],
  Quantities: [10, 5, ...]
```

```
    ]
```

```
// -------------------- InventoryManager --------------------
class InventoryManager
{
private:
    List2D<InventoryAttribute> attributesMatrix;
    List1D<string> productNames;
    List1D<int> quantities;

public:
    InventoryManager();
    InventoryManager(const List2D<InventoryAttribute> &matrix,
                     const List1D<string> &names,
                     const List1D<int> &quantities);
    InventoryManager(const InventoryManager &other);

    int size() const;
    List1D<InventoryAttribute> getProductAttributes(int index) const;
    string getProductName(int index) const;
    int getProductQuantity(int index) const;
    void updateQuantity(int index, int newQuantity);
    void addProduct(const List1D<InventoryAttribute> &attributes, const
   string &name, int quantity);
    void removeProduct(int index);

    List1D<string> query(int attributeName, const double &minValue,
                         const double &maxValue, int minQuantity, bool
   ascending) const;

    void removeDuplicates();

    static InventoryManager merge(const InventoryManager &inv1,
                                  const InventoryManager &inv2);

    void split(InventoryManager &section1,
               InventoryManager &section2,
               double ratio) const;

    List2D<InventoryAttribute> getAttributesMatrix() const;
    List1D<string> getProductNames() const;
    List1D<int> getQuantities() const;
    string toString() const;
};
```

3 Requirements

Students must complete the above classes according to the listed interfaces, ensuring:

- Implement the methods marked with //TODO.
- Students are allowed to add additional methods, member variables, and functions to support the above classes.
- Students are responsible for modifying the original source code for behaviors not covered in the two guidelines above.
- Students are not permitted to include any other libraries. If detected, the student will receive a zero grade for the project.

3.1 Compilation

Students **should** integrate the code into an IDE of their choice for convenience and use the interface for compilation.

If you need to compile via the command line, you may refer to the following command: **g++ -g -I include -I src -std=c++17 src/main.cpp -o main**

3.2 Submission

Detailed submission instructions will be provided later.

4 Other Regulations

- Students must complete this project independently and prevent others from copying their results. Failure to do so will result in disciplinary action for academic misconduct.
- All decisions made by the project supervisor are final.
- Students are not allowed to provide test cases after grading, though they may provide information on test case design strategy and the distribution of student numbers per test case.

- The content of the project will be synchronized with a similar question in the exam.

5 Tracking Changes Across Versions

(v1.1)

- Adjusted the prototype of the query() function to sort by attributeName.
- Updated the description of the Inventory Manager source code to match the initial code.

END