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

QUẢN LÝ KHO HÀNG SỬ DỤNG DANH SÁCH

TP. HỒ CHÍ MINH, THÁNG 02/2025

ĐẶC TẢ BÀI TẬP LỚN

Phiên bản 1.0

1 Giới thiệu

1.1 Mục tiêu và nhiệm vụ

Mục tiêu của các bài tập lớn (BTL) trong môn học này là giúp sinh viên phát triển và ứng dụng các cấu trúc dữ liệu để xây dựng các ứng dụng thực tế. Trong học kỳ này, ứng dụng được chọn làm chủ đề chính là quản lý kho hàng sản phẩm. Các bài tập lớn có tính liên kết với nhau (ví dụ, BTL-2 sẽ kế thừa từ BTL-1), do đó sinh viên cần lưu ý đảm bảo tính liên thông giữa các bài tập.

Nhiệm vụ của BTL-1 được trình bày như sau:

- **Phát triển** cấu trúc dữ liệu danh sách, cụ thể là danh sách dựa trên mảng và danh sách dựa trên liên kết (hai lớp **XArrayList** và **DLinkedList** trong mã nguồn được cung cấp).
- **Sử dụng** các danh sách đã phát triển để xây dựng thư viện các lớp hỗ trợ quản lý kho hàng, bao gồm:
  1. `List1D`: Hiện thực danh sách một chiều lưu trữ các giá trị.
  2. `List2D`: Hiện thực danh sách hai chiều (ma trận) lưu trữ các thuộc tính của sản phẩm.
  3. `InventoryManager`: Quản lý thông tin kho hàng sản phẩm với các thành phần gồm ma trận thuộc tính sản phẩm, danh sách tên sản phẩm và danh sách số lượng tồn kho.

1.2 Phương pháp tiến hành

1. Chuẩn bị: **Download** và **tìm hiểu** mã nguồn được cung cấp. **Lưu ý**, sinh viên cần biên dịch mã nguồn trong các BTL với **C++17** là bắt buộc. Bộ biên dịch, nhóm thực hiện đã kiểm tra với **g++**; khuyến nghị sinh viên dựng môi trường để sử dụng **g++**.
2. **Phát triển** danh sách: Hiện thực lớp **XArrayList** và **DLinkedList** trong thư mục **/include/list**
3. **Sử dụng** danh sách được hiện ở trên và phát triển các lớp ứng dụng **List1D**, **List2D** và **InventoryManager**
4. **Chạy thử**: Kiểm tra chương trình phải qua được các testcases mẫu được cung cấp.

5. **Nộp bài**: Phải nộp bài trên hệ thống trước deadline.

1.3 Chuẩn đầu ra của BTL-1

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/C++ ở mức nâng cao.
- **Phát triển** được cấu trúc dữ liệu danh sách, với hai phiên bản, dựa trên mảng và dựa vào liên kết.
- **Lựa chọn và sử dụng** được danh sách vào phát triển các lớp **List1D**, **List2D** và **InventoryManager** .

1.4 Thành phần điểm và phương pháp chấm bài

1. Hiện thực danh sách: (60% **điểm**); gồm tập tin sau đây trong thư mục **/include/list**
   - **XArrayList.h**
   - **DLinkedList.h**
2. Hiện thực các lớp về tập dữ liệu và tải dữ liệu: (40% **điểm**); gồm các tập tin sau đây trong thư mục **/include/app**:
   - **inventory.h**

Sinh viên được chạy đánh giá với các testcases được công bố mẫu khi nộp bài. Bài tập lớn sau khi nộp lên hệ thống sẽ được chấm điểm theo các testcases chưa công bố.

2 Hướng dẫn

2.1 Hiện thực cấu trúc dữ liệu danh sách

2.1.1 Phương pháp thiết kế của các cấu trúc dữ liệu

Trong môn học này các cấu trúc dữ liệu được thiết kế theo mẫu nhất quán; đó là,

1. Với mỗi cấu trúc dữ liệu sẽ có một hoặc vài lớp trừu tượng định nghĩa về các tác vụ sẽ được hỗ trợ bởi cấu trúc dữ liệu. Trong trường hợp danh sách, đó là lớp **IList** trong thư mục **/include/list**.

2. Trong thực tế, các cấu trúc dữ liệu sẽ chứa được các phần tử **có kiểu bất kỳ**, ví dụ như **đối tượng sinh viên**, **con trỏ đến đối tượng sinh viên**, hay chỉ là một con số kiểu **int**. Do đó, tất cả các cấu trúc dữ liệu trong môn học điều đã sử dụng **template** (trong C++) để tham số hoá kiểu phần tử.
3. Tất cả các cấu trúc dữ liệu điều được thiết kế để che dấu dữ liệu và các chi tiết ở mức cao; cũng như tách bạch hai vai trò, đó là, **người phát triển thư viện** và **người sử dụng thư viện**. Cũng chính vì vậy, tất cả các cấu trúc dữ liệu, kể cả danh sách, đã được bổ sung tính năng duyệt qua các phần tử (tính năng **iterator**) để thuận tiện cho việc duyệt qua phần tử từ phía người dùng.

2.1.2 Tổng quan về cấu trúc dữ liệu danh sách

Cấu trúc dữ liệu danh sách trong BTL này được thiết kế gồm các lớp như sau:

- Lớp `IList`, xem Hình 1: lớp này định nghĩa một danh mục các APIs được hỗ trợ bởi danh sách; bất kể là hiện thực bởi array hay bởi liên kết cũng sẽ phải hỗ trợ các APIs trong `IList`. `IList` là lớp cha của hai lớp `XArrayList` và `DLinkedList`. Một số ý chi tiết:
  - `IList` sử dụng **template** để tham số hoá kiểu dữ liệu phần tử, và cho phép chứa trong danh sách với kiểu bất kỳ.
  - Tất cả các APIs trong `IList` để ở dạng "pure virtual method"; nghĩa là các lớp kế thừa từ `IList` cần phải override tất cả các phương thức này và phương thức dạng này sẽ hỗ trợ liên kết động (tính đa hình).
- Lớp `XArrayList` và `DLinkedList`: được thừa kế từ `IList`, và sẽ hiện thực tất cả các APIs được định nghĩa trong `IList`, bằng cách sử dụng array (như trong `XArrayList`) và sử dụng liên kết (như trong `DLinkedList`).

Bên dưới là mô tả cho từng pure virtual method của `IList`:

- `virtual ~IList() {};`
  - Destructor ảo, được dùng để đảm bảo rằng destructor của các lớp con được gọi khi xóa đối tượng thông qua con trỏ lớp cơ sở.
- `virtual void add(T e) = 0;`
  - Thêm phần tử `e` vào cuối danh sách.
  - **Tham số**: `T e` — phần tử cần thêm.
- `virtual void add(int index, T e) = 0;`

```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;
};
```

Hình 1: `IList<T>`: Lớp trừu tượng định nghĩa APIs cho danh sách.

  - Chèn phần tử `e` vào vị trí `index` trong danh sách.
  - **Tham số**:
    * `int index` — vị trí muốn chèn phần tử.
    * `T e` — phần tử cần chèn.

- `virtual T removeAt(int index) = 0;`
  - **Mô tả**: Xóa và trả về phần tử tại vị trí `index`.
  - **Tham số**: `int index` — vị trí của phần tử cần xóa.
  - **Giá trị trả về**: Trả về phần tử tại vị trí `index`.
  - **Ngoại lệ**: Ném `std::out_of_range` nếu `index` không hợp lệ.
- `virtual bool removeItem(T item, void (*removeItemData)(T) = 0) = 0;`
  - **Mô tả**: Xóa phần tử `item` được lưu trữ trong danh sách.
  - **Tham số**:
    * `T item` — phần tử cần xóa.
    * `void (*removeItemData)(T) = 0` — con trỏ hàm (mặc định là NULL) được gọi để xóa dữ liệu của phần tử trong danh sách. Điều này cần thiết vì không xác định được kiểu `T` có phải là con trỏ hay không.
  - **Giá trị trả về**: `true` nếu phần tử `item` tồn tại và được xóa; ngược lại, `false`.
- `virtual bool empty() = 0;`
  - **Mô tả**: Kiểm tra xem danh sách có rỗng không.
  - **Giá trị trả về**: `true` nếu danh sách rỗng; ngược lại, `false`.
- `virtual int size() = 0;`

  - **Mô tả**: Trả về số lượng phần tử hiện có trong danh sách.
  - **Giá trị trả về**: Số lượng phần tử trong danh sách.

- `virtual void clear() = 0;`
  - **Mô tả**: Xóa tất cả các phần tử trong danh sách, đưa danh sách về trạng thái ban đầu.

- `virtual T& get(int index) = 0;`
  - **Mô tả**: Trả về tham chiếu đến phần tử tại vị trí `index`.
  - **Tham số**: `int index` — vị trí của phần tử cần lấy.
  - **Giá trị trả về**: Tham chiếu đến phần tử tại vị trí `index`.
  - **Ngoại lệ**: Ném `std::out_of_range` nếu `index` không hợp lệ.

- `virtual int indexOf(T item) = 0;`
  - **Mô tả**: Trả về chỉ số của phần tử `item` đầu tiên tìm thấy trong danh sách.
  - **Tham số**: `T item` — phần tử cần tìm chỉ số.
  - **Giá trị trả về**: Chỉ số của phần tử `item`; trả về `-1` nếu không tìm thấy.

- `virtual bool contains(T item) = 0;`
  - **Mô tả**: Kiểm tra xem danh sách có chứa phần tử `item` hay không.
  - **Tham số**: `T item` — phần tử cần kiểm tra.
  - **Giá trị trả về**: `true` nếu danh sách chứa phần tử `item`; ngược lại, `false`.

- `virtual string toString(string (*item2str)(T&) = 0) = 0;`
  - **Mô tả**: Trả về một chuỗi mô tả danh sách.
  - **Tham số**: `string (*item2str)(T&) = 0` — con trỏ hàm để chuyển đổi từng phần tử trong danh sách thành chuỗi.
  - **Giá trị trả về**: Chuỗi mô tả danh sách.

2.1.3 Danh sách đặc (Array List)

`XArrayList<T>` là một phiên bản hiện thực danh sách sử dụng mảng (array) để chứa các phần tử, kiểu `T`. Về nguyên tắc, `XArrayList<T>` phải chủ động duy trì một mảng đủ lớn để chứa các phần tử trong nó. Việc biến động phần tử chỉ liên quan đến APIs như `add`, `remove` và `removeItem`; do đó, khi hiện thực các APIs cần phải kiểm tra kích thước mảng đang có để đảm bảo đủ không gian lưu trữ các phần tử.

Ngoài các phương thức để hiện thực các APIs có trong `IList`, `XArrayList<T>` cũng còn cần các phương thức hỗ trợ khác, xem trong tập tin **XArrayList.h**, trong thư mục **/include/list**.

Lưu ý: Sinh viên chỉ cần hiện thực các phương thức có comment //TODO

```
        template <class T>
class XArrayList : public IList<T>
{
public:
    class Iterator; // forward declaration

protected:
    T *data;                                    // dynamic array to store the
    list's items
    int capacity;                               // size of the dynamic array
    int count;                                  // number of items stored in
    the array
    bool (*itemEqual)(T &lhs, T &rhs);          // function pointer: test if
    two items (type: T&) are equal or not
    void (*deleteUserData)(XArrayList<T> *); // function pointer: be called
    to remove items (if they are pointer type)

public:
    XArrayList(
        void (*deleteUserData)(XArrayList<T> *) = 0,
        bool (*itemEqual)(T &, T &) = 0,
        int capacity = 10);
    XArrayList(const XArrayList<T> &list);
    XArrayList<T> &operator=(const XArrayList<T> &list);
    ~XArrayList();

    // 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: BEGIN

    void println(string (*item2str)(T &) = 0)
    {
```

```
39          cout << toString(item2str) << endl;
40      }
41      void setDeleteUserDataPtr(void (*deleteUserData)(XArrayList<T> *) = 0)
42      {
43          this->deleteUserData = deleteUserData;
44      }
45
46      Iterator begin()
47      {
48          return Iterator(this, 0);
49      }
50      Iterator end()
51      {
52          return Iterator(this, count);
53      }
54
55      /** free:
56       * if T is pointer type:
57       *      pass THE address of method "free" to XArrayList<T>'s constructor:
58       *      to:  remove the user's data (if needed)
59       * Example:
60       *  XArrayList<Point*> list(&XArrayList<Point*>::free);
61       *  => Destructor will call free via function pointer "deleteUserData"
62       */
63      static void free(XArrayList<T> *list)
64      {
65          typename XArrayList<T>::Iterator it = list->begin();
66          while (it != list->end())
67          {
68              delete *it;
69              it++;
70          }
71      }
72
73  protected:
74      void checkIndex(int index);     // check validity of index for accessing
75      void ensureCapacity(int index); // auto-allocate if needed
76
77      /** equals:
78       * if T: primitive type:
79       *      indexOf, contains: will use native operator ==
80       *      to: compare two items of T type
81       * if T: object type:
```

```
     *       indexOf, contains: will use native operator ==
     *       to: compare two items of T type
     *       Therefore, class of type T MUST override operator ==
     * if T: pointer type:
     *       indexOf, contains: will use function pointer "itemEqual"
     *       to: compare two items of T type
     *       Therefore:
     *       (1): must pass itemEqual to the constructor of XArrayList
     *       (2): must define a method for comparing
     *            the content pointed by two pointers of type T
     *           See: definition of "equals" of class Point for more detail
     */
    static bool equals(T &lhs, T &rhs, bool (*itemEqual)(T &, T &))
    {
        if (itemEqual == 0)
            return lhs == rhs;
        else
            return itemEqual(lhs, rhs);
    }

    void copyFrom(const XArrayList<T> &list);

    void removeInternalData();

    ///////////////////////////////////////////////////////////////////
    ////////////////////////  INNER CLASSES DEFNITION ///////////////////
    ///////////////////////////////////////////////////////////////////
public:
    // Iterator: BEGIN
    class Iterator
    {
    private:
        int cursor;
        XArrayList<T> *pList;

    public:
        Iterator(XArrayList<T> *pList = 0, int index = 0)
        {
            this->pList = pList;
            this->cursor = index;
        }
        Iterator &operator=(const Iterator &iterator)
        {
```

```
125             cursor = iterator.cursor;
126             pList = iterator.pList;
127             return *this;
128         }
129         void remove(void (*removeItemData)(T) = 0)
130         {
131             T item = pList->removeAt(cursor);
132             if (removeItemData != 0)
133                 removeItemData(item);
134             cursor -= 1; // MUST keep index of previous, for ++ later
135         }
136 
137         T &operator*()
138         {
139             return pList->data[cursor];
140         }
141         bool operator!=(const Iterator &iterator)
142         {
143             return cursor != iterator.cursor;
144         }
145         // Prefix ++ overload
146         Iterator &operator++()
147         {
148             this->cursor++;
149             return *this;
150         }
151         // Postfix ++ overload
152         Iterator operator++(int)
153         {
154             Iterator iterator = *this;
155             ++*this;
156             return iterator;
157         }
158     };
159     // Iterator: END
160 };
161 
```

1. **Các thuộc tính:**
    - `T* data`: Mảng động lưu trữ các phần tử của danh sách.
    - `int capacity`: Sức chứa hiện tại của mảng động `data`.
    - `int count`: Số lượng phần tử hiện có trong danh sách.

- `bool (*itemEqual)(T& lhs, T& rhs)`: Con trỏ hàm để kiểm tra xem hai phần tử có bằng nhau hay không.
- `void (*deleteUserData)(XArrayList<T>*)`: Con trỏ hàm để xóa dữ liệu người dùng khi cần thiết, đặc biệt hữu ích khi các phần tử là con trỏ.

2. **Hàm khởi tạo và hàm hủy:**
   - `XArrayList(int capacity)`: Hàm khởi tạo cho phép khởi tạo danh sách với sức chứa ban đầu
   - `XArrayList(const XArrayList<T>& list)`: Hàm sao chép khởi tạo, sao chép dữ liệu từ một đối tượng `XArrayList` khác.
   - `XArrayList<T>& operator=(const XArrayList<T>& list)`: Toán tử gán sao chép, dùng để sao chép dữ liệu từ một đối tượng `XArrayList` khác.
   - `~XArrayList()`: Hàm hủy, giải phóng bộ nhớ đã cấp phát cho mảng động `data` và các phần tử.
3. **Các phương thức:**
   - **void add(T e)**
     - **Chức năng**: Thêm một phần tử e vào cuối danh sách.
     - **Ngoại lệ**: Không có.
   - **void add(int index, T e)**
     - **Chức năng**: Thêm phần tử e vào vị trí chỉ định *index* trong danh sách.
     - **Ngoại lệ**: Nếu *index* không hợp lệ (ngoài đoạn $[0, \text{count}]$), ném ra ngoại lệ `out_of_range("Index is out of range!")`.
   - **T removeAt(int index)**
     - **Chức năng**: Xóa phần tử tại vị trí *index* và trả về phần tử bị xóa.
     - **Ngoại lệ**: Nếu *index* không hợp lệ (ngoài đoạn $[0, \text{count} - 1]$), ném ra ngoại lệ `out_of_range("Index is out of range!")`.
   - **bool removeItem(T item, void (*removeItemData)(T) = 0)**
     - **Chức năng**: Xóa phần tử đầu tiên trong danh sách có giá trị bằng *item*.
     - **Ngoại lệ**: Không có.
   - **bool empty()**
     - **Chức năng**: Kiểm tra xem danh sách có rỗng hay không.
     - **Ngoại lệ**: Không có.
   - **int size()**
     - **Chức năng**: Trả về số lượng phần tử hiện có trong danh sách.
     - **Ngoại lệ**: Không có.

- **void clear()**
  - **Chức năng**: Xóa tất cả các phần tử trong danh sách và đặt danh sách về trạng thái ban đầu.
  - **Ngoại lệ**: Không có.
- **T& get(int index)**
  - **Chức năng**: Trả về tham chiếu đến phần tử tại vị trí *index*.
  - **Ngoại lệ**: Nếu *index* không hợp lệ (ngoài đoạn $[0, \text{count} - 1]$), ném ra ngoại lệ `out_of_range("Index is out of range!")`.
- **int indexOf(T item)**
  - **Chức năng**: Trả về chỉ số của phần tử đầu tiên có giá trị bằng *item* trong danh sách, hoặc -1 nếu không tìm thấy.
  - **Ngoại lệ**: Không có.
- **bool contains(T item)**
  - **Chức năng**: Kiểm tra xem danh sách có chứa phần tử *item* hay không.
  - **Ngoại lệ**: Không có.
- **string toString(string (*item2str)(T&) = 0)**
  - **Chức năng**: Trả về chuỗi biểu diễn các phần tử trong danh sách.
  - **Ngoại lệ**: Không có.

2.1.4 Danh sách liên kết đôi (Doubly Linked List)

`DLinkedList<T>` là một phiên bản hiện thực danh sách sử dụng hai liên kết **next** và **prev**. Về phía người sử dụng của danh sách, họ không quan tâm về hiện thực chi tiết bên trong. Do đó, họ cũng không phải nhận thức rằng có liên kết hay không. Để làm được việc này, về nguyên tắc, `DLinkedList<T>` cần có đối tượng "**node**", và đối tượng này sẽ chứa dữ liệu của người dùng, cũng như hai liên kết, một đến phần tử kế tiếp, một đến phần tử theo sau.

Ngoài các phương thức để hiện thực các APIs có trong `IList`, `DLinkedList<T>` còn cần các phương thức hỗ trợ khác, xem trong tập tin **DLinkedList.h**, trong thư mục **/include/list**.

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

```
9
10 public:
11     DLinkedList();
12     ~DLinkedList();
13
14     //Inherit from IList: BEGIN
15     void    add(T e);
16     void    add(int index, T e);
17     T       removeAt(int index);
18     bool    removeItem(T item, void (*removeItemData)(T)=0);
19     bool    empty();
20     int      size();
21     void    clear();
22     T&      get(int index);
23     int      indexOf(T item);
24     bool    contains(T item);
25     string  toString(string (*item2str)(T&)=0 );
26     //Inherit from IList: END
27
28 public:
29     class Node{
30     public:
31         T data;
32         Node *next;
33         Node *prev;
34         friend class DLinkedList<T>;
35
36     public:
37         Node(Node* next=0, Node* prev=0);
38         Node(T data, Node* next=0, Node* prev=0);
39     };
40 };
```

1. **class Node:**
    - `T data`: Biến kiểu `T`, lưu trữ dữ liệu của node. `T` là kiểu dữ liệu tổng quát (sử dụng template).
    - `Node* next`: Con trỏ trỏ tới node tiếp theo trong danh sách liên kết đôi.
    - `Node* prev`: Con trỏ trỏ tới node trước đó trong danh sách liên kết đôi.
    - `Node(Node* next = 0, Node* prev = 0)`: Constructor mặc định cho phép khởi tạo một node với các con trỏ `next` và `prev` nhận giá trị mặc định là `nullptr`.
    - `Node(T data, Node* next = 0, Node* prev = 0)`: Constructor cho phép khởi tạo

một node với dữ liệu `data`, cùng với các con trỏ `next` và `prev`.

2. **Hàm khởi tạo và hàm hủy:**
   - `DLinkedList()`: Khởi tạo một đối tượng danh sách liên kết đôi rỗng. Gợi ý: Sinh viên nên tham khảo phương pháp hiện thực sử dụng dummy node (Gán head và tail cho một Node rỗng) để đồng bộ với cách hiện thực class Iterator cung cấp sẵn.
   - `~DLinkedList()`: Hủy đối tượng danh sách liên kết đôi, giải phóng tất cả các nút trong danh sách.
3. **Các thuộc tính:**
   - `Node* head`: Con trỏ đến đầu danh sách liên kết đôi.
   - `Node* tail`: Con trỏ đến cuối danh sách liên kết đôi.
   - `int count`: Biến số nguyên lưu trữ số lượng node chứa dữ liệu của người dùng trong danh sách.
4. **Các phương thức:**
   - `void add(T e)`
     - **Chức năng:** Thêm phần tử `e` vào cuối danh sách.
     - **Ngoại lệ:** Không có ngoại lệ.
   - `void add(int index, T e)`
     - **Chức năng:** Thêm phần tử `e` vào vị trí chỉ định `index` trong danh sách.
     - **Ngoại lệ:** Nếu `index` không hợp lệ (ví dụ: nhỏ hơn 0 hoặc lớn hơn kích thước danh sách), sẽ ném ra ngoại lệ `std::out_of_range`.
   - `T removeAt(int index)`
     - **Chức năng:** Xóa và trả về phần tử tại vị trí chỉ định `index` trong danh sách.
     - **Ngoại lệ:** Nếu `index` không hợp lệ (ví dụ: nhỏ hơn 0 hoặc lớn hơn hoặc bằng kích thước danh sách), sẽ ném ra ngoại lệ `std::out_of_range`.
   - `bool removeItem(T item, void (*removeItemData)(T)=0)`
     - **Chức năng:** Xóa phần tử đầu tiên trong danh sách mà có giá trị bằng với `item`. Nếu tìm thấy và xóa thành công, trả về `true`; ngược lại trả về `false`.
     - **Ngoại lệ:** Nếu con trỏ hàm `removeItemData` được cung cấp, hàm này sẽ được gọi để xử lý dữ liệu trước khi xóa. Không có ngoại lệ khác.
   - `bool empty()`
     - **Chức năng:** Kiểm tra xem danh sách có rỗng hay không. Trả về `true` nếu rỗng, ngược lại trả về `false`.
     - **Ngoại lệ:** Không có ngoại lệ.

- `int size()`
  - **Chức năng:** Trả về số lượng phần tử hiện tại trong danh sách.
  - **Ngoại lệ:** Không có ngoại lệ.
- `void clear()`
  - **Chức năng:** Xóa tất cả các phần tử trong danh sách, làm cho danh sách trở thành rỗng.
  - **Ngoại lệ:** Không có ngoại lệ.
- `T& get(int index)`
  - **Chức năng:** Trả về tham chiếu đến phần tử tại vị trí chỉ định `index` trong danh sách.
  - **Ngoại lệ:** Nếu `index` không hợp lệ (ví dụ: nhỏ hơn 0 hoặc lớn hơn hoặc bằng kích thước danh sách), sẽ ném ra ngoại lệ `std::out_of_range`.
- `int indexOf(T item)`
  - **Chức năng:** Trả về chỉ số của phần tử đầu tiên có giá trị bằng với `item` trong danh sách. Nếu không tìm thấy, trả về -1.
  - **Ngoại lệ:** Không có ngoại lệ.
- `bool contains(T item)`
  - **Chức năng:** Kiểm tra xem danh sách có chứa phần tử có giá trị bằng với `item` hay không. Trả về `true` nếu có, ngược lại trả về `false`.
  - **Ngoại lệ:** Không có ngoại lệ.
- `string toString(string (*item2str)(T&)=0)`
  - **Chức năng:** Trả về một chuỗi ký tự đại diện cho danh sách. Nếu con trỏ hàm `item2str` được cung cấp, hàm này sẽ được gọi để chuyển đổi mỗi phần tử sang chuỗi ký tự.
  - **Ngoại lệ:** Không có ngoại lệ.

2.2 Ứng dụng cấu trúc dữ liệu danh sách

2.2.1 Lớp `List1D<T>`

`List1D<T>` là lớp hiện thực danh sách một chiều dùng để lưu trữ một mảng các giá trị thuộc kiểu `T` thông qua cấu trúc dữ liệu danh sách. Lớp này cung cấp các API cơ bản nhằm thao tác trên danh sách, bao gồm việc lấy kích thước, truy xuất, gán, thêm phần tử và chuyển đổi danh sách thành chuỗi để hỗ trợ in ra.

1. **Hàm khởi tạo và hàm hủy:**
   - `List1D()`: Khởi tạo danh sách rỗng.
   - `List1D(int num_elements)`: Khởi tạo danh sách với số lượng phần tử được chỉ định (ban đầu, mỗi phần tử sẽ được khởi tạo với giá trị mặc định, ví dụ như 0).
   - `List1D(const T* array, int num_elements)`: Khởi tạo danh sách từ một mảng các giá trị.
   - `List1D(const List1D<T>& other)`: Copy constructor, tạo danh sách mới dựa trên nội dung của danh sách hiện có.
   - `virtual ~List1D()`: Destructor, giải phóng bộ nhớ đã cấp phát.
2. **Các phương thức thành viên:**
   - `int size() const`: Trả về số lượng phần tử hiện có trong danh sách.
   - `T get(int index) const`: Truy xuất phần tử tại vị trí `index`.
   - `void set(int index, T value)`: Gán giá trị mới cho phần tử tại vị trí `index`.
   - `void add(const T& value)`: Thêm một phần tử vào danh sách.
   - `string toString() const`: Trả về chuỗi biểu diễn của danh sách theo định dạng:

     $$[e_1, e_2, e_3, ..., e_n]$$

     với $e_1, e_2, e_3, \ldots$ là các phần tử của danh sách. Ví dụ: `[1, 2, 3, 4]`.
   - Toán tử `<<`: Hỗ trợ in ra danh sách thông qua đối tượng `ostream` với định dạng như trên.

```
// -------------------- 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;
```

```
17      void set(int index, T value);
18      void add(const T &value);
19      string toString() const;
20
21      friend ostream &operator<<(ostream &os, const List1D<T> &list);
22  };
```

2.2.2 Lớp `List2D<T>`

`List2D<T>` là lớp hiện thực danh sách hai chiều dùng để lưu trữ một ma trận các giá trị thuộc kiểu `T` thông qua cấu trúc danh sách. Phiên bản mới của lớp này không cố định số cột, mà mỗi hàng có thể chứa số lượng phần tử khác nhau. Mỗi hàng của ma trận được biểu diễn bởi một đối tượng `List1D<T>`, từ đó hỗ trợ lưu trữ các tập hợp thuộc tính không đồng nhất cho từng sản phẩm.

1. **Hàm khởi tạo và hàm hủy:**
   - `List2D()`: Khởi tạo một ma trận rỗng.
   - `List2D(List1D<T>* array, int num_rows)`: Khởi tạo ma trận từ một mảng các đối tượng `List1D<T>` với số hàng được chỉ định. Vì số cột của mỗi hàng có thể khác nhau nên không cần chỉ định số cột cố định.
   - `List2D(const List2D<T>& other)`: Hàm sao chép, tạo một ma trận mới dựa trên nội dung của ma trận hiện có.
   - `virtual ~List2D()`: Hàm hủy, giải phóng bộ nhớ đã cấp phát cho ma trận.
2. **Các phương thức thành viên:**
   - `int rows() const`: Trả về số lượng hàng của ma trận.
   - `void setRow(int rowIndex, const List1D<T>& row)`: Gán một hàng mới cho ma trận tại vị trí `rowIndex`.
   - `T get(int rowIndex, int colIndex) const`: Truy xuất giá trị tại vị trí xác định bởi `rowIndex` và `colIndex`.
   - `List1D<T> getRow(int rowIndex) const`: Trả về hàng của ma trận dưới dạng đối tượng `List1D<T>`.
   - `string toString() const`: Trả về chuỗi biểu diễn của ma trận theo định dạng:

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

     với mỗi hàng có thể có số lượng phần tử (cột) khác nhau. Ví dụ: `[[1, 2], [3, 4, 5], [6]]`.

- Toán tử <<: Hỗ trợ in ra ma trận thông qua đối tượng `ostream` với định dạng như trên.

```
// -------------------- 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 Quản lý tồn kho (`InventoryManager`)

`InventoryManager` là lớp quản lý tồn kho sản phẩm, cung cấp các API để lưu trữ và xử lý thông tin của các sản phẩm trong kho. Lớp này quản lý dữ liệu thông qua ba thành phần chính:

- `attributesMatrix`: Ma trận các thuộc tính của sản phẩm (kiểu `List2D<InventoryAttribute>`); mỗi hàng biểu diễn các thuộc tính của một sản phẩm. Lưu ý: Số lượng thuộc tính của từng sản phẩm có thể khác nhau.
- `productNames`: Danh sách tên sản phẩm (kiểu `List1D<string>`).
- `quantities`: Danh sách số lượng tồn kho của từng sản phẩm (kiểu `List1D<int>`).

Ví dụ 2.1

Giả sử trong kho có 3 sản phẩm với thông tin như sau:

1. **Sản phẩm A:**
   - **Thuộc tính:** [("weight", 10), ("height", 156)]
   - **Tên sản phẩm:** "Product A"
   - **Số lượng tồn kho:** 50
2. **Sản phẩm B:**
   - **Thuộc tính:** [("weight", 20),("depth", 24), ("height", 100)]
   - **Tên sản phẩm:** "Product B"
   - **Số lượng tồn kho:** 30
3. **Sản phẩm C:**
   - **Thuộc tính:** [("color", 2)]
   - **Tên sản phẩm:** "Product C"
   - **Số lượng tồn kho:** 20

Thông tin này sẽ được quản lý thông qua ba thành phần của lớp `InventoryManager` như sau:

- `attributesMatrix` (kiểu `List2D<InventoryAttribute>`) chứa:

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

- `productNames` (kiểu `List1D<string>`) chứa:

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

- `quantities` (kiểu `List1D<int>`) chứa:

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

Các phương thức cần hiện thực cho lớp `InventoryManager` được mô tả chi tiết như sau:

1. `InventoryManager()`

- **Chức năng:** Khởi tạo đối tượng `InventoryManager` rỗng.

2. `InventoryManager(const List2D<InventoryAttribute>& matrix, const List1D<string>& names, const List1D<int>& quantities)`
   - **Chức năng:** Khởi tạo đối tượng `InventoryManager` sử dụng ma trận thuộc tính, danh sách tên sản phẩm và danh sách số lượng được cung cấp.
3. `InventoryManager(const InventoryManager& other)`
   - **Chức năng:** Tạo một đối tượng `InventoryManager` mới bằng cách copy toàn bộ dữ liệu từ đối tượng `other`.
4. `int size() const`
   - **Chức năng:** Trả về số lượng sản phẩm hiện có trong kho.
5. `List1D<InventoryAttribute> getProductAttributes(int index) const`
   - **Chức năng:** Trả về danh sách các thuộc tính của sản phẩm tại vị trí `index`.
   - **Ngoại lệ:** Nếu `index` không hợp lệ (không nằm trong đoạn $[0, \texttt{size()} - 1]$), ném ra ngoại lệ `out_of_range("Index is invalid!")`.
6. `string getProductName(int index) const`
   - **Chức năng:** Trả về tên của sản phẩm tại vị trí `index`.
   - **Ngoại lệ:** Nếu `index` không hợp lệ, ném ra ngoại lệ `out_of_range("Index is invalid!")`.
7. `int getProductQuantity(int index) const`
   - **Chức năng:** Trả về số lượng tồn kho của sản phẩm tại vị trí `index`.
   - **Ngoại lệ:** Nếu `index` không hợp lệ, ném ra ngoại lệ `out_of_range("Index is invalid!")`.
8. `void updateQuantity(int index, int newQuantity)`
   - **Chức năng:** Cập nhật số lượng tồn kho của sản phẩm tại vị trí `index` với giá trị mới `newQuantity`.
   - **Ngoại lệ:** Nếu `index` không hợp lệ, ném ra ngoại lệ `out_of_range("Index is invalid!")`.
9. `void addProduct(const List1D<InventoryAttribute>& attributes, const string& name, int quantity)`
   - **Chức năng:** Thêm một sản phẩm mới vào kho với các thuộc tính, tên và số lượng được chỉ định.
10. `void removeProduct(int index)`

- **Chức năng:** Xóa sản phẩm tại vị trí `index` khỏi kho.
- **Ngoại lệ:** Nếu `index` không hợp lệ, ném ra ngoại lệ `out_of_range("Index is invalid!")`.

11. `List1D<string> query(int attributeName, const double &minValue, const double &maxValue, int minQuantity, bool ascending) const`
    - **Chức năng:** Truy vấn và trả về danh sách tên sản phẩm có giá trị thuộc tính tên `attributeName` nằm trong khoảng `[minValue, maxValue]` và số lượng tồn kho không nhỏ hơn `minQuantity`. Kết quả được sắp xếp theo thứ tự tăng dần nếu `ascending` là `true`.
    - **Ví dụ:** Giả sử thuộc tính thứ 1 của sản phẩm biểu diễn trọng lượng, nếu gọi `query(1, 5, 10, 3, true)`, hàm sẽ trả về danh sách tên của các sản phẩm có trọng lượng từ 5 đến 10 và số lượng tồn kho ít nhất là 3, sắp xếp theo thứ tự tăng dần.
12. `void removeDuplicates()`
    - **Chức năng:** Loại bỏ các sản phẩm trùng lặp trong kho. Nếu kho chứa hai sản phẩm có thông tin giống hệt nhau (cùng thuộc tính và tên), gọi `removeDuplicates()` sẽ chỉ giữ lại một sản phẩm với số lượng là tổng của tất cả các sản phẩm trùng lặp.
    - **Lưu ý:** Phương thức này nên được hiện thực với độ phức tạp thời gian là $O(n^2)$ để vượt qua tất cả testcase.
13. `static InventoryManager merge(const InventoryManager &inv1, const InventoryManager &inv2)`
    - **Chức năng:** Hợp nhất hai đối tượng tồn kho `inv1` và `inv2` thành một kho duy nhất.
14. `void split(InventoryManager &section1, InventoryManager &section2, double ratio) const`
    - **Chức năng:** Phân chia kho hiện tại thành hai phần, lưu vào `section1` và `section2` theo tỉ lệ `ratio`. Nếu sau khi chia, số lượng sản phẩm của `section1` không là số nguyên thì làm tròn lên.
    - **Ví dụ:** Nếu gọi `split(sec1, sec2, 0.6)` trên kho có 11 sản phẩm, thì `sec1` sẽ chứa 7 sản phẩm và `sec2` sẽ chứa 4 sản phẩm.
15. `List2D<InventoryAttribute> getAttributesMatrix() const`
    - **Chức năng:** Trả về ma trận các thuộc tính của toàn bộ sản phẩm trong kho.
16. `List1D<string> getProductNames() const`
    - **Chức năng:** Trả về danh sách tên của tất cả các sản phẩm trong kho.

17. `List1D<int> getQuantities() const`
    - **Chức năng:** Trả về danh sách số lượng tồn kho của tất cả các sản phẩm.
18. `string toString() const`
    - **Chức năng:** Trả về chuỗi ký tự biểu diễn đầy đủ thông tin tồn kho, bao gồm ma trận thuộc tính, danh sách tên sản phẩm và số lượng tồn kho.
    - **Ví dụ:** Hàm trả về chuỗi:

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

```cpp
// -------------------- 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;

```

```
27      void removeDuplicates();
28
29      static InventoryManager merge(const InventoryManager &inv1,
30                                    const InventoryManager &inv2);
31
32      void split(InventoryManager &section1,
33                 InventoryManager &section2,
34                 double ratio) const;
35
36      List2D<InventoryAttribute> getAttributesMatrix() const;
37      List1D<string> getProductNames() const;
38      List1D<int> getQuantities() const;
39      string toString() const;
40  };
```

3 Yêu cầu

Sinh viên cần hoàn thiện các lớp trên theo các giao diện được liệt kê, đảm bảo:

- Hiện thực các phương thức được comment //TODO
- Sinh viên được phép bổ sung các phương thức, biến thành viên, hàm để hỗ trợ cho các lớp trên.
- Sinh viên tự chịu trách nhiệm với việc sửa mã nguồn ban đầu cho những hành vi không nằm trong 2 hướng dẫn trên.
- Sinh viên không được phép include bất cứ thư viện nào khác. Nếu phát hiện, sinh viên sẽ bị điểm 0 cho BTL.

3.1 Biên dịch

Sinh viên **Nên** tích hợp code vào một trong các IDE nào đó cảm thấy thuận tiện cho cá nhân, và sử dụng giao diện để biên dịch.

Nếu cần biên dịch bằng dòng, sinh viên có thể tham khảo dòng lệnh sau đây: **g++ -g -I include -I src -std=c++17 src/main.cpp -o main**

3.2 Nộp bài

Chỉ dẫn nộp bài sẽ được công bố chi tiết sau.

4 Các quy định khác

- Sinh viên phải hoàn thành dự án này một cách độc lập và ngăn chặn người khác sao chép kết quả của mình. Nếu không làm được điều này sẽ dẫn đến hành động kỷ luật vì gian lận học thuật.
- Tất cả các quyết định của giảng viên phụ trách dự án là quyết định cuối cùng.
- Sinh viên không được phép cung cấp testcase sau khi đã chấm điểm nhưng có thể cung cấp thông tin về chiến lược thiết kế testcase và phân phối số lượng sinh viên cho từng testcase.
- Nội dung của dự án sẽ được đồng bộ với một câu hỏi trong kỳ thi có nội dung tương tự.

5 Theo dõi các thay đổi qua các phiên bản

(v1.1)

- Điều chỉnh lại prototype của hàm query() thành sắp xếp theo attributeName
- Cập nhật lại mô tả source code Inventory Manager cho khớp với initial code
-

——————————HẾT——————————