

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



Cấu trúc dữ liệu và giải thuật - CO2003

Bài tập lớn 2

**BẢNG BĂM, CẤU TRÚC ĐỒNG và
MẠNG NƠN NHIỀU LỚP**

TP. HỒ CHÍ MINH, THÁNG 09/2024



ĐẶC TẢ BÀI TẬP LỚN

Phiên bản 1.0

1. Phiên bản 1.0: October 10, 2024

1 Giới thiệu

1.1 Nội dung

Bài tập lớn số hai của môn Cấu trúc dữ liệu và giải thuật có hai nội dung sau đây:

1. Nội dung 1 (còn được gọi là **TASK-1**, chiếm 40% cột điểm): Nội dung này yêu cầu sinh viên phát triển hai cấu trúc dữ liệu sau đây:
 - (a) **Cấu trúc Bảng băm (HashMap)**. Tập tin cần hiện thực cho phần này là `./include/hash/xMap.h`. Sinh viên đọc hướng dẫn thực hiện cho nội dung này trong Phần 2.
 - (b) **Cấu trúc Đống (Heap)**. Tập tin cần hiện thực cho phần này là `./include/heap/Heap.h`. Sinh viên đọc hướng dẫn thực hiện cho nội dung này trong Phần 3.
2. Nội dung 2 (còn được gọi là **TASK-2**, chiếm 60% cột điểm): Nội dung này yêu cầu sinh viên sử dụng các cấu trúc dữ liệu đã học để phát triển mạng nơ-ron truyền thẳng nhiều lớp. Sinh viên đọc hướng dẫn thực hiện cho nội dung này trong Phần 4.

1.2 Cấu trúc dự án

Mã nguồn được cung cấp kèm theo tài liệu này là một dự án có cấu trúc như Hình 1 và 2. Lưu ý, sinh viên phải download tập tin và giải nén để nhìn thấy cấu trúc dự án.






















Tại thư mục gốc của dự án, sinh viên sẽ nhìn thấy các thư mục và tập tin sau đây:

- **./include**: Thư mục này chứa tất cả các tập tin header (*.h) liên quan đến dự án. Một số thư mục con tiêu biểu:
 - **./include/list**: Chứa các tập tin liên quan cấu trúc dữ liệu Danh sách, đã phát triển ở Bài tập lớn 1.
 - **./include/loader**: Chứa các tập tin liên quan cấu trúc dữ liệu cho `Dataset`, `TensorDataset`, `DataLoader`, đã phát triển ở Bài tập lớn 1.
 - **./include/hash**: Chứa các tập tin liên quan cấu trúc dữ liệu Bảng băm (HashMap), thuộc **TASK-1** của Bài tập lớn 2.
 - **./include/heap**: Chứa các tập tin liên quan cấu trúc dữ liệu Đống (Heap), thuộc **TASK-1** của Bài tập lớn 2.
 - **./include/ann**: Chứa các tập tin liên quan đến **TASK-2** của Bài tập lớn 2.

- **./include/tensor**: Chứa các tập tin liên quan thư viện **xtensor**, thư viện cho ma trận nhiều chiều.
- **./include/sformat**: Chứa các tập tin liên quan thư viện **fmt**, dùng để định dạng chuỗi.
- **./include/graph**: Chứa các tập tin liên quan đến cấu trúc dữ liệu Đồ thị (Graph), thuộc **TASK-1** của Bài tập lớn 3.
- **./include/util**: Chứa các tập tin chứa các lớp và hàm tiện ích cho dự án.
- **./include/dsaheader.h**: Tập tin chứa phần “**#include**” cho các cấu trúc dữ liệu được phát triển trong môn học và tên gọi ngắn của chúng.
- **./src**: thư mục này chứa tất cả các tập tin “***.cpp**” liên quan đến dự án. Thư mục này có các thư mục và tập tin sau đây.
 - **./src/ann**: Chứa mã nguồn hiện thực của các lớp trong **TASK-2** của Bài tập lớn 2.
 - **./src/tensor**: Chứa mã nguồn hiện thực của các hàm mở rộng cho thư viện **xtensor**. Các hàm này sẽ được dùng cho **TASK-2** của Bài tập lớn 2.
 - **./src/program.cpp**: Đây là tập tin chứa hàm **main** của dự án.
- **./demo**: Thư mục này chứa tất cả các tập tin “***.h**” là demo về cách sử dụng của các cấu trúc dữ liệu và các lớp trong dự án. *Sinh viên nên tham khảo các ví dụ trong thư mục này để biết cách dùng các cấu trúc dữ liệu và các thư viện.*
- **./datasets**: Thư mục này chứa các tập dữ liệu dùng trong **TASK-2** của bài tập lớn.
- **./models**: thư mục này chứa các mô hình (model) được tạo ra trong **TASK-2**. *Sinh viên nên lưu các mô hình trong thư mục này.*
- **./config.txt**: Tập tin này chứa các biến cấu hình được dùng trong **TASK-2**.
- **./Makefile**: Tập tin để hỗ trợ biên dịch dự án theo cách dùng lệnh **make**. Nếu sinh viên cần biên dịch dự án thì gõ lệnh **make** tại cửa sổ **Terminal**.
- **./compilation-command.sh**: Tập tin để hỗ trợ biên dịch dự án theo cách dùng lệnh **g++**. Nếu sinh viên cần biên dịch dự án thì gõ lệnh **./compilation-command.sh** tại cửa sổ **Terminal**.

1.3 Phương pháp thực hiện

- **Lưu ý quan trọng**:
 1. **TASK-1** của Bài tập lớn 2 cần đến (phụ thuộc) cấu trúc dữ liệu danh sách liên kết kép, cụ thể là **DLinkedList** trong **TASK-1** của Bài 1. Lưu ý, trong **TASK-2** của Bài 2, chúng ta cần dùng đến **Backward-Iterator** của **DLinkedList**; do đó, sinh viên

Folders	Folders
 datasets >	 ann >
 demo >	 graph >
 include >	 hash >
 models >	 heap >
 src >	 list >
	 loader >
Documents	 sformat >
 config.txt	 sorting >
	 stacknqueue >
Developer	 tensor >
 compilation-command.sh	 tree >
 Makefile	 util >
	Developer
	 dsaheader.h

Hình 1: Cấu trúc dự án cho Bài tập lớn 2 và nội dung thư mục `./include`

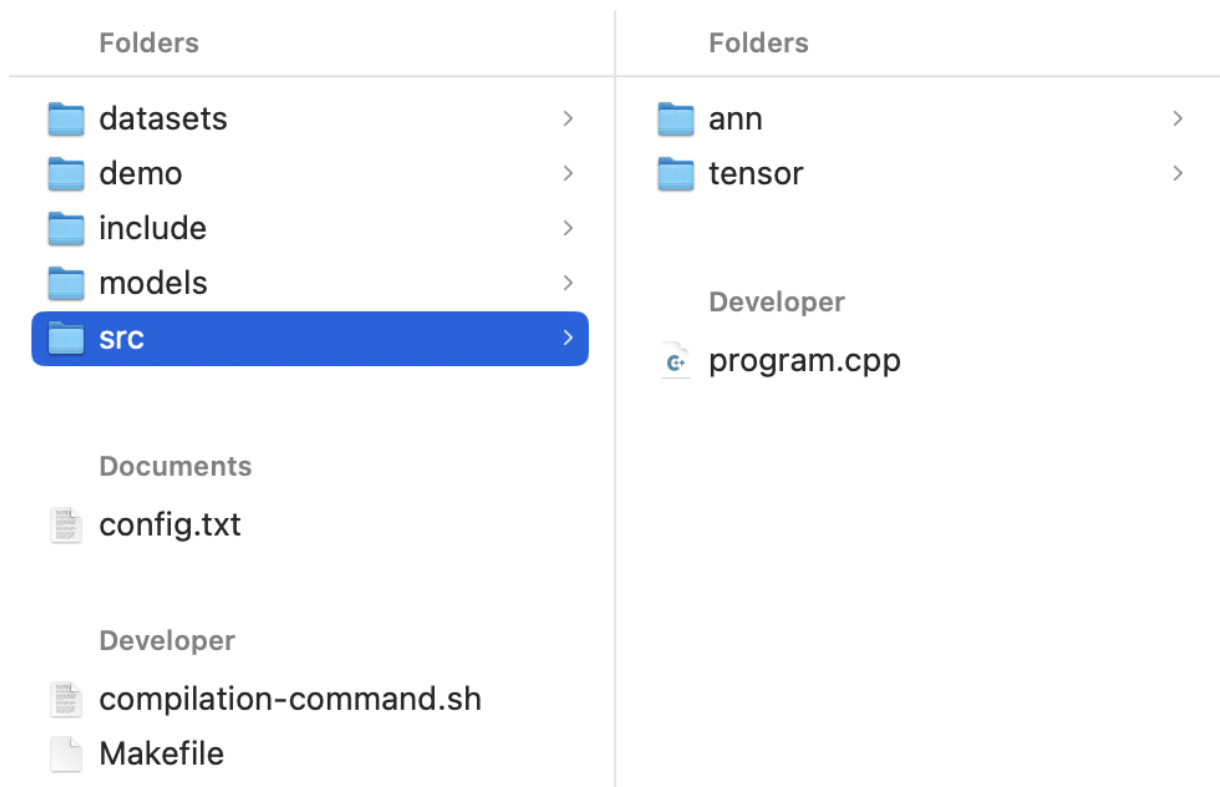
cũng được yêu cầu để bổ sung Backward-Iterator cho DLinkedList trước khi dùng trong TASK-2. Backward-Iterator cũng phải hỗ trợ các toán tử sau.

- (a) Toán tử so sánh khác (`!=`).
- (b) Toán tử `++`: khi `++` thì lùi theo hướng từ cuối về đầu danh sách.
- (c) Toán tử `*`: dùng để trích xuất phần tử.

2. **TASK-2** của Bài 2 thì phụ thuộc vào DLinkedList và các lớp trong hai tập tin `./include/loader/dataset.h` và `./include/loader/dataloader.h`. Mặt khác, lớp **Dataset**, **TensorDataset** và **DataLoader** cũng có bổ sung một số phương thức mới. Do đó, sinh viên cần cập nhật bản hiện thực cho các lớp này theo phiên bản mới (chỉ là cập nhật nhỏ).

• Cách thực hiện **TASK-1**:

- Sinh viên **PHẢI thực hiện TASK-1** trước và độc lập với **TASK-2**, bằng cách di chuyển mã nguồn của **TASK-2** sang một thư mục nào đó **nằm ngoài dự án**. Mã nguồn cho **TASK-2** nằm trong:



Hình 2: Cấu trúc dự án cho Bài tập lớn 2 và nội dung thư mục `./src`

- * `./include/ann`
- * `./include/tensor/tensor_lib.h` (chỉ di chuyển tập tin này).
- * `./src/ann`
- * `./src/tensor`
- Hiện thực các hàm trong `./include/hash/xMap.h` và `./include/heap/Heap.h`:
điền code vào những chỗ trống được đánh dấu bởi: **YOUR CODE IS HERE**.
- Biên dịch và chạy thử. Nên thử với các ví dụ trong `./demo/hash` và `./demo/heap`
- Cách thực hiện **TASK-2**:
 - Sinh viên **CHỈ CÓ THỂ** thực hiện **TASK-2** khi **TASK-1** và các cấu trúc trong Bài 1 đã được xây dựng thành công.
 - Di chuyển code được cung cấp cho **TASK-2** vào đúng thư mục và tiến hành thực hiện.
 - Biên dịch và chạy thử.

1.4 Cách biên dịch dự án

Sinh viên có thể theo một trong 3 cách sau đây để biên dịch dự án.

1. Dùng tập tin `./compilation-command.sh`:

- Mở cửa sổ **Terminal** và chuyển đến thư mục gốc của dự án, dùng lệnh `cd`;
- Thực hiện lệnh: `./compilation-command.sh`
- Nếu có lỗi xảy ra thì phải xử lý lỗi chương trình và biên dịch lại.

2. Sử dụng **Makefile**:

- Mở cửa sổ **Terminal** và chuyển đến thư mục gốc của dự án, dùng lệnh `cd`;
- Thực hiện lệnh:
 - `make` hay
 - `make clean` và `make`
- Nếu có lỗi xảy ra thì phải xử lý lỗi chương trình và biên dịch lại.

3. Sử dụng các môi trường phát triển tích hợp (IDE) như **NetBeans**, **VSCode**, v.v.:

- Tạo một dự án trên IDE;
- Thêm các tập tin của dự án trong Bài tập lớn vào dự án ở IDE;
- Cấu hình để biên dịch với `g++` và `C++17`.
- Biên dịch với menu trên IDE.

2 TASK-1: Cấu trúc bảng băm

2.1 Nguyên tắc thiết kế

Tương tự như ở các cấu trúc dữ liệu khác, phần hiện thực cho bảng băm trong thư viện này gồm hai lớp: (a) Lớp **IMap** (xem Hình 3), dùng để định nghĩa các APIs cho bảng băm; và (b) Lớp **xMap** (là lớp con của **IMap**) chứa hiện thực cụ thể cho bảng băm.

- Lớp **IMap**, xem Hình 3: lớp này định nghĩa một tập hợp các phương thức (API) được hỗ trợ bởi bảng băm.
 - **IMap** sử dụng **template** để tham số hoá kiểu dữ liệu phần tử; cụ thể là, kiểu của **key** và của **value**. Do đó, bảng băm có thể làm việc với bất kỳ kiểu **key** và **value** nào.
 - Tất cả các APIs trong **IMap** để ở dạng **pure virtual method**; nghĩa là các lớp kế thừa từ **IMap** cần phải **override** tất cả các phương thức này. Do có tính **virtual** nên các APIs sẽ hỗ trợ liên kết động (tính đa hình).
- Lớp **xMap**: được thừa kế từ **IMap**. Lớp này chứa hiện thực cụ thể cho tất cả các APIs được định nghĩa trong **IMap** bằng cách sử dụng **danh sách liên kết kép (DoublyLinkedList)**

để chứa tất cả các cặp <key, value> có xảy ra đụng độ (collision) tại từng địa chỉ trong bảng băm. Nói cách khác, bảng băm trong thư viện là bảng của các danh sách liên kết kép.

```
1
2 template<class K, class V>
3 class IMap {
4 public:
5     virtual ~IMap(){};
6     virtual V put(K key, V value)=0;
7     virtual V& get(K key)=0;
8     virtual V remove(K key, void (*deleteKeyInMap)(K)=0)=0;
9     virtual bool remove(K key, V value, void (*deleteKeyInMap)(K)=0, void (*deleteValueInMap)(V)=0)=0;
10    virtual bool containsKey(K key)=0;
11    virtual bool containsValue(V value)=0;
12    virtual bool empty()=0;
13    virtual int size()=0;
14    virtual void clear() = 0;
15    virtual string toString(string (*key2str)(K&)=0, string (*value2str)(V&)=0 )=0;
16    virtual DLinkedList<K> keys()=0;
17    virtual DLinkedList<V> values()=0;
18    virtual DLinkedList<int> clashes()=0;
19 };
```

Hình 3: IMap<T>: Lớp trừu tượng định nghĩa APIs cho bảng băm.

2.2 Giải thích các APIs

Dưới đây là mô tả cho từng pure virtual method của IMap:

- virtual ~IMap() {};
 - Destructor ảo, đảm bảo rằng destructor của các lớp con được gọi khi xóa đối tượng thông qua con trỏ đến đối tượng kiểu IMap.
- virtual V put(K key, V value) = 0;
 - Tham số:
 - * K key — khóa cần thêm hoặc cập nhật.
 - * V value — giá trị cần thêm hoặc thay thế.
 - Trả về:
 - * Nếu key được tìm thấy có sẵn trong bảng thì trả về giá trị value cũ (đang có trong bảng); ngược lại, trả về giá trị value mới truyền vào.

– Mục đích:

- * Thêm một cặp **key** và **value** vào bảng băm. Nếu **key** đã tồn tại, gán giá trị mới và trả về giá trị cũ.

• virtual V& get(K key) = 0;

- Mục đích: Trả về giá trị được ánh xạ bởi **key**.
- Tham số: K key — khóa cần lấy giá trị.
- Giá trị trả về: Tham chiếu tới giá trị tương ứng với **key**.
- Ngoại lệ: Ném ngoại lệ **KeyNotFound** nếu **key** không tồn tại.

• virtual V remove(K key, void (*deleteKeyInMap)(K) = 0) = 0;

- Mục đích: Xóa **key** khỏi map và trả về giá trị tương ứng.
- Tham số:
 - * K key — khóa cần xóa.
 - * void (*deleteKeyInMap)(K) — con trỏ hàm (mặc định là NULL) được gọi để xóa khóa trong trường hợp K là con trỏ.
- Giá trị trả về: Giá trị tương ứng với **key**.
- Ngoại lệ: Ném ngoại lệ **KeyNotFound** nếu **key** không tồn tại.

• virtual bool remove(K key, V value, void (*deleteKeyInMap)(K) = 0, void (*deleteValueInMap)(V) = 0) = 0;

- Mục đích: Xóa cặp <key, value> nếu tồn tại.
- Tham số:
 - * K key — khóa cần xóa.
 - * V value — giá trị cần xóa.
 - * void (*deleteKeyInMap)(K) — con trỏ hàm xóa khóa (mặc định là NULL).
 - * void (*deleteValueInMap)(V) — con trỏ hàm xóa giá trị (mặc định là NULL).
- Giá trị trả về: true nếu cặp <key, value> được xóa, false nếu không tìm thấy.

• virtual bool containsKey(K key) = 0;

- Mục đích: Kiểm tra xem **key** có tồn tại trong map không.
- Tham số: K key — khóa cần kiểm tra.
- Giá trị trả về: true nếu **key** tồn tại, ngược lại false.

• virtual bool containsValue(V value) = 0;

- Mục đích: Kiểm tra xem **value** có tồn tại trong bảng băm không, tại bất kỳ địa chỉ nào của bảng băm.
- Tham số: V value — giá trị cần kiểm tra.

- **Giá trị trả về:** true nếu value tồn tại, ngược lại false.
- `virtual bool empty() = 0;`
 - **Mục đích:** Kiểm tra xem map có rỗng không.
 - **Giá trị trả về:** true nếu map rỗng, ngược lại false.
- `virtual int size() = 0;`
 - **Mục đích:** Trả về số lượng cặp **key-value** trong map.
 - **Giá trị trả về:** Số lượng phần tử trong map. Nếu trả về 0 thì nghĩa là bảng rỗng.
Bảng rỗng: là bảng có chứa **capacity** (giá trị mặc nhiên của **capacity** là 10) danh sách liên kết, nhưng từng danh sách trong đó là rỗng.
- `virtual void clear() = 0;`
 - **Mục đích:** Xóa tất cả các cặp **<key, value>** trong bảng băm và đưa bảng băm về trạng thái rỗng. **Lưu ý: Bảng rỗng:** là bảng có chứa **capacity** (giá trị mặc nhiên của **capacity** là 10) danh sách liên kết, nhưng từng danh sách trong đó là rỗng.
- `virtual string toString(string (*key2str)(K&) = 0, string (*value2str)(V&) = 0) = 0;`
 - **Mục đích:** Trả về chuỗi mô tả map.
 - **Tham số:**
 - * `string (*key2str)(K&)` — con trỏ hàm để chuyển khóa thành chuỗi. Trường hợp con trỏ này là NULL thì K phải hỗ trợ toán tử chèn (`<<`) để chuyển **key** sang chuỗi.
 - * `string (*value2str)(V&)` — con trỏ hàm để chuyển giá trị thành chuỗi. Trường hợp con trỏ này là NULL thì V phải hỗ trợ toán tử chèn (`<<`) để chuyển **value** sang chuỗi.
 - **Giá trị trả về:** Chuỗi mô tả bảng băm.
- `virtual DLinkedList<K> keys() = 0;`
 - **Mục đích:** Trả về danh sách các khóa trong map.
 - **Giá trị trả về:** `DLinkedList<K>` chứa các khóa.
- `virtual DLinkedList<V> values() = 0;`
 - **Mục đích:** Trả về danh sách các giá trị trong map.
 - **Giá trị trả về:** `DLinkedList<V>` chứa các giá trị.
- `virtual DLinkedList<int> clashes() = 0;`
 - **Mục đích:** Trả về danh sách số lần đụng độ tại mỗi địa chỉ trong map.
 - **Giá trị trả về:** `DLinkedList<int>` chứa số lần va chạm.

2.3 Bảng băm (Hash Map)

$xMap<K, V>$ là một phiên bản hiện thực của bảng băm (hash map), nơi các phần tử được lưu trữ dưới dạng cặp khóa-giá trị (K, V) trong một mảng có kích thước xác định trước, gọi là *table*. Nguyên lý hoạt động của $xMap<K, V>$ dựa trên việc sử dụng hàm băm để xác định vị trí lưu trữ các phần tử trong mảng.

Để đảm bảo hiệu quả hoạt động, $xMap<K, V>$ cần duy trì một mảng đủ lớn để chứa các phần tử khóa-giá trị và phải quản lý tốt tải trọng (load factor), nghĩa là tỷ lệ giữa số lượng phần tử hiện tại và kích thước của mảng. Nếu tỷ lệ này vượt quá giá trị `loadFactor` được chỉ định, bảng băm sẽ tự động thực hiện quá trình **rehash**, tức là tăng kích thước mảng và phân phối lại các phần tử.

Ngoài các phương thức kế thừa từ `IMap` để xử lý các thao tác cơ bản như `put`, `get`, `remove`, `containsKey`, và `clear`, $xMap<K, V>$ cũng hỗ trợ các phương thức tiện ích khác như `rehash` để mở rộng bảng băm, `ensureLoadFactor` để đảm bảo tỷ lệ tải trọng. Những phương thức này có thể được tìm thấy trong tập tin `xMap.h`, trong thư mục `/include/hash`.

1. Các thuộc tính: Xem Hình 4.

- `int capacity`: Sức chứa hiện tại của bảng băm.
- `int count`: Số lượng phần tử hiện có trong bảng băm.
- `DLinkedList<Entry* >* table`: Bảng băm là một mảng (động) của các phần tử có kiểu là danh sách liên kết kép (`DLinkedList`); và mỗi phần tử của danh sách là **con trỏ** đến `Entry`; ở đó, `Entry` là lớp chứa cặp `<key, value>`. Ta có thể hình dung, bảng băm là một dãy của các đối tượng kiểu danh sách; và danh sách sẽ chứa tất cả các cặp **đựng độ** tại một địa chỉ cụ thể.
- `float loadFactor`: Hệ số tải của bảng băm, cho biết mức độ sử dụng không gian trước khi phải mở rộng. Tại bất kỳ thời điểm nào thì số lượng phần tử trong bảng (`count`) **không được vượt quá** `loadFactor × capacity`. Ví dụ, nếu `capacity = 10` và `loadFactor = 0.75` thì số phần tử lớn nhất có thể chứa là `int(0.75 × 10) = 7`; khi chèn thêm phần tử thứ 8 thì cần phải gọi `ensureLoadFactor` để đảm bảo hệ số tải.
- Các thuộc tính là **con trỏ hàm**, được khởi gán qua hàm khởi tạo. **Lưu ý**: chỉ có `hashCode` là luôn luôn phải được truyền vào qua hàm khởi tạo nên chắc chắn phải khác `NULL`; các con trỏ hàm khác có thể `NULL` hay không tùy vào nhu cầu của người sử dụng thư viện.
 - `int (*hashCode)(K&, int)`: hàm này nhận vào **key** (truyền bằng tham khảo) và

```

1  template<class K, class V>
2  class xMap: public IMap<K,V>{
3  public:
4      class Entry; //forward declaration
5  protected:
6      DLinkedList<Entry* >* table; //array of DLinkedList objects
7      int capacity; //size of table
8      int count; //number of entries stored hash-map
9      float loadFactor; //define max number of entries can be stored (< (
10     loadFactor * capacity))
11
12     int (*hashCode)(K&,int); //hashCode(K key, int tableSize): tableSize
13     means capacity
14     bool (*keyEqual)(K&,K&); //keyEqual(K& lhs, K& rhs): test if lhs == rhs
15     bool (*valueEqual)(V&,V&); //valueEqual(V& lhs, V& rhs): test if lhs ==
16     rhs
17     void (*deleteKeys)(xMap<K,V>*); //deleteKeys(xMap<K,V>* pMap): delete
18     all keys stored in pMap
19     void (*deleteValues)(xMap<K,V>*); //deleteValues(xMap<K,V>* pMap):
20     delete all values stored in pMap
21
22     //HIDDEN CODE
23 public:
24     //Entry: BEGIN
25     class Entry{
26     private:
27         K key;
28         V value;
29         friend class xMap<K,V>;
30
31     public:
32         Entry(K key, V value){
33             this->key = key;
34             this->value = value;
35         }
36     };
37     //Entry: END
38 };

```

Hình 4: xMap<T>: Cấu trúc bảng băm; phần khai báo thuộc tính

kích thước bảng băm (số nguyên); nó sẽ tính ra địa chỉ của khoá trên bảng băm có kích thước truyền vào. **Lưu ý:** Nếu kích thước bảng băm truyền vào hàm là m thì giá trị trả về của hàm `hashCode` chỉ có thể nằm trong khoảng $[0, 1, \dots, m-1]$; để đảm bảo điều này, `hashCode` phải dùng đến toán tử $\%$ (modulo). Xem thêm các hàm `intKeyHash` và `stringKeyHash` có trong mã nguồn kèm theo.

- `bool (*keyEqual)(K&,K&)`: Trong trường hợp kiểu dữ liệu của khoá (K) không hỗ trợ việc so sánh tính bằng nhau giữa hai khoá qua toán tử `==`, thì người dùng cần truyền con trỏ đến một hàm có thể so sánh tính bằng nhau giữa hai

khoá. Hàm đó, nhận vào hai khoá kiểu `K` (truyền bằng tham khảo) và trả về `true` nếu hai khoá bằng nhau, và `false` nếu hai khoá không bằng nhau. **Chú ý:** Bên trong lớp `xMap` khi cần so sánh hai khoá thì hãy gọi phương thức `keyEQ`.

- `bool (*valueEqual)(V&,V&)`: Trong trường hợp kiểu dữ liệu của giá trị (`V`) không hỗ trợ việc so sánh tính bằng nhau giữa hai giá trị qua toán tử `==`, thì người dùng cần truyền con trỏ đến một hàm có thể so sánh tính bằng nhau giữa hai giá trị. Hàm đó, nhận vào hai giá trị kiểu `V` (truyền bằng tham khảo) và trả về `true` nếu hai giá trị bằng nhau, và `false` nếu hai giá trị không bằng nhau. **Chú ý:** Bên trong lớp `xMap` khi cần so sánh hai giá trị thì hãy gọi phương thức `valueEQ`.

- `void (*deleteKeys)(xMap<K,V>* pMap)`: Trong trường hợp `K` là con trỏ và người sử dụng thư viện cần `xMap` chủ động giải phóng bộ nhớ cho các khoá thì người sử dụng **CẦN** truyền vào con trỏ hàm thông qua `deleteKeys`. Người dùng cũng không cần phải định nghĩa hàm mới, mà chỉ cần truyền `xMap<K,V>::freeKey` vào cho `deleteKeys`. Hãy xem mã nguồn của `xMap<K,V>::freeKey` cho chi tiết.

Con trỏ hàm để xóa các khóa khi cần thiết.

- `void (*deleteValues)(xMap<K,V>* pMap)`: Trong trường hợp `V` là con trỏ và người sử dụng thư viện cần `xMap` chủ động giải phóng bộ nhớ cho các giá trị thì người sử dụng **CẦN** truyền vào con trỏ hàm thông qua `deleteValues`. Người dùng cũng không cần phải định nghĩa hàm mới, mà chỉ cần truyền `xMap<K,V>::freeValue` vào cho `deleteValues`. Hãy xem mã nguồn của `xMap<K,V>::freeValue` cho chi tiết.

2. Hàm khởi tạo và hàm hủy:

- `xMap(`
 `int (*hashCode)(K&,int),`
 `float loadFactor,`
 `bool (*valueEqual)(V&,V&),`
 `void (*deleteValues)(xMap<K,V>*),`
 `bool (*keyEqual)(K&,K&),`
 `void (*deleteKeys)(xMap<K,V>*)):`

- **Mục đích:** Hàm khởi tạo sẽ tạo ra một bảng băm rỗng, có `capacity` danh sách liên kết trong bảng. Giá trị `capacity` mặc nhiên là 10. Hàm này cũng khởi động các biến thành viên là con trỏ hàm đã được đề cập ở trên với các giá trị truyền

vào.

– **Tham số:** xem phần mô tả thuộc tính.

- **xMap(const xMap<K,V>& map):** Hàm này sao chép dữ liệu từ một đối tượng bảng băm khác.
- **~xMap():** Hàm hủy, giải phóng bộ nhớ và tài nguyên đã được cấp phát cho bảng băm; bao gồm cả các **keys**, **values** và **entries** nếu có hoặc nếu người dùng yêu cầu.

3. Các phương thức:

- **V put(K key, V value)**

- **Chức năng:** Hàm này chèn một cặp **<key, value>** vào bảng băm. Nếu **key** đã tồn tại, giá trị cũ sẽ được cập nhật bởi **value**. Nếu **key** chưa tồn tại, một cặp **<key, value>** mới sẽ được thêm vào bảng băm. Hàm cũng có thể tự động mở rộng kích thước của bảng băm khi cần thiết (khi hệ số tải vượt quá ngưỡng cho phép).
- **Hướng dẫn hiện thực:** các ý chính như sau.
 - Sử dụng hàm băm **hashCode** để tính toán địa chỉ của **key**.
 - Lấy danh sách từ **table** bằng địa chỉ ở trên.
 - Tìm xem **key** đã có chứa trong danh sách hay chưa.
 - * Nếu có thì cập nhật **value** cũ bằng **value** mới. Nhớ backup **value** cũ để trả về.
 - * Nếu không (bảng băm chưa chứa **key**): tạo một **Entry** cho cặp **<key, value>** và đưa vào danh sách. Tăng số lượng phần tử trong bảng và gọi hàm **ensureLoadFactor** để đảm bảo hệ số tải.
 - (d) **Ngoại lệ:** Không có.

- **V get(K key)**

- **Chức năng:** Hàm này trả về giá trị liên kết với khóa **key** được truyền vào. Nếu khóa không tồn tại trong bảng băm, ném ra ngoại lệ **KeyNotFound**, đã được định nghĩa sẵn trong tập tin **IMap.h**.
- **Hướng dẫn hiện thực:** các ý chính như sau.
 - Sử dụng hàm băm **hashCode** để tính toán địa chỉ của **key** và lấy danh sách tại địa chỉ vừa tìm được.
 - Tìm xem **key** đã được chứa trong danh sách chưa.
 - * Nếu tìm thấy, trả về **value** tương ứng (chứa trong cùng **Entry**).
 - * Nếu không tìm thấy, ném ra ngoại lệ.
 - (c) **Ngoại lệ:** Ngoại lệ **KeyNotFound** khi không tìm thấy **key**.

- **V remove(K key, void (*deleteKeyInMap)(K) = 0)**
 - **Chức năng:** Xoá **Entry** chứa **key** ra khỏi bảng băm, khi tìm thấy **key**. Hàm này cũng gọi **deleteKeyInMap** để giải phóng vùng nhớ **key** khi **deleteKeyInMap** khác **nullptr**.
 - **Hướng dẫn hiện thực:** các ý chính như sau.
 - (a) Sử dụng hàm băm **hashCode** để tính toán địa chỉ của **key** và lấy danh sách tại địa chỉ vừa tìm được.
 - (b) Tìm xem **key** đã được chứa trong danh sách chưa.
 - * Nếu tìm thấy: (a) backup **value** để trả về; (b) Giải phóng **key** nếu **deleteKeyInMap** khác **NULL**; (c) gỡ **Entry** (chứa cặp **<key, value>**) ra khỏi danh sách và giải phóng vùng nhớ của **Entry**. Gợi ý: sử dụng hàm **removeItem** trên danh sách để vừa gỡ bỏ **Entry** và vừa giải phóng bộ nhớ của **Entry**, bằng cách truyền con trỏ đến hàm **xMap<K,V>::deleteEntry** vào hàm **removeItem**.
 - * Nếu không tìm thấy: ném ra ngoại lệ.
 - (c) **Ngoại lệ:** Ném ra ngoại lệ **KeyNotFound** nếu khóa **key** không tồn tại trong bảng.
- **bool remove(K key, V value, void (*deleteKeyInMap)(K), void (*deleteValueInMap)(V))**
 - **Chức năng:** Gỡ **Entry** chứa cặp **<key, value>** có trong bảng băm. Hàm này so sánh khớp cả **key** và **value** để xác định **Entry** nào đó được gỡ ra khỏi bảng hay không. Hàm này cũng giải phóng bộ nhớ cho **key** và **value** khi được yêu cầu; nghĩa là, khi **deleteKeyInMap** hay **deleteValueInMap** hay cả hai khác **NULL**. Hàm này chỉ trả về **true** nếu tìm thấy **<key, value>** và **false** nếu không tìm thấy cặp **<key, value>**.
 - **Hướng dẫn hiện thực:** các ý chính như sau.
 - (a) Tương tự như hàm **remove(K key, void (*deleteKeyInMap)(K))**. Khác nhau ở chỗ cần so sánh khớp cho cả **key** và **value**.
 - **Ngoại lệ:** Không có.
- **bool containsKey(K key)**
 - **Chức năng:** Kiểm tra xem bảng băm có chứa khóa **key** hay không.
 - **Hướng dẫn hiện thực:** các ý chính như sau.
 - (a) Sử dụng hàm băm **hashCode** để tính toán địa chỉ của **key** và lấy danh sách tại địa chỉ vừa tìm được.
 - (b) Tìm **key** trong danh sách ở trên và trả về kết quả tương ứng.

- **Ngoại lệ:** Không có.
- **bool containsValue(V value)**
 - **Chức năng:** Kiểm tra xem bảng băm có chứa giá trị *value* hay không.
 - **Hướng dẫn hiện thực:** các ý chính như sau.
 - (a) Tìm **value** trong tất cả các danh sách có trong bảng băm và trả về kết quả tương ứng.
 - **Ngoại lệ:** Không có.
- **bool empty()**
 - **Chức năng:** Kiểm tra xem bảng băm có rỗng hay không.
 - **Ngoại lệ:** Không có.
- **int size()**
 - **Chức năng:** Trả về số lượng phần tử hiện có trong bảng băm.
 - **Ngoại lệ:** Không có.
- **void clear()**
 - **Chức năng:** Xóa tất cả các phần tử trong bảng băm và đặt bảng băm về trạng thái ban đầu.
 - **Hướng dẫn hiện thực:** các ý chính như sau.
 - (a) Gọi hàm **removeInternalData** để giải phóng bộ nhớ.
 - (b) Khởi tạo lại bảng băm rỗng, với **capacity** là 10.
 - **Ngoại lệ:** Không có.
- **string toString(string (*key2str)(K&) = 0, string (*value2str)(V&) = 0)**
 - **Chức năng:** Trả về chuỗi biểu diễn của các phần tử trong bảng băm.
 - **Ngoại lệ:** Không có.
- **DLinkedList<K> keys()**
 - **Chức năng:** Trả về danh sách liên kết chứa tất cả các khóa trong bảng băm.
 - **Ngoại lệ:** Không có.
- **DLinkedList<V> values()**
 - **Chức năng:** Trả về danh sách liên kết chứa tất cả các giá trị trong bảng băm.
 - **Ngoại lệ:** Không có.
- **DLinkedList<int> clashes()**
 - **Chức năng:** Trả về danh sách liên kết chứa số phần tử trong danh sách tại từng địa chỉ.
 - **Hướng dẫn hiện thực:** các ý chính như sau.
 - **Ngoại lệ:** Không có.

3 TASK-1: Cấu trúc dữ liệu Heap

3.1 Nguyên tắc thiết kế

Tương tự như ở các cấu trúc dữ liệu khác, phần hiện thực cho **Heap** trong thư viện này gồm hai lớp: (a) Lớp **IHeap** (xem Hình 5), dùng để định nghĩa các APIs cho cấu trúc **Heap**; và (b) Lớp **Heap** (là lớp con của **IHeap**) chứa hiện thực cụ thể cho **Heap**.

- Lớp **IHeap**, xem Hình 5: lớp này định nghĩa một tập hợp các phương thức (API) được hỗ trợ bởi **Heap**. Một số lưu ý về **IHeap** như sau:
 - **IHeap** sử dụng **template** để tham số hoá kiểu dữ liệu phần tử. Do đó, **Heap** có thể chứa các phần tử có kiểu bất kỳ, miễn sao kiểu dữ liệu đó hỗ trợ phép so sánh để cho biết: (a) về tính bằng nhau và (b) về trật tự trước sau.
 - Tất cả các APIs trong **IHeap** để ở dạng **pure virtual method**; nghĩa là các lớp kế thừa từ **IHeap** cần phải **override** tất cả các phương thức này. Do có tính **virtual** nên các APIs sẽ hỗ trợ liên kết động (tính đa hình).
- Lớp **Heap**: được thừa kế từ **IHeap**. Lớp này chứa hiện thực cụ thể cho tất cả các APIs được định nghĩa trong **IHeap**.

```
1 template<class T>
2 class IHeap {
3 public:
4     virtual ~IHeap(){};
5     virtual void push(T item)=0;
6     virtual T pop()=0;
7     virtual const T peek()=0;
8     virtual void remove(T item, void (*removeItemData)(T)=0)=0;
9     virtual bool contains(T item)=0;
10    virtual int size()=0;
11    virtual void heapify(T array[], int size)=0; //build heap from array
        having size items
12    virtual void clear()=0;
13    virtual bool empty()=0;
14    virtual string toString(string (*item2str)(T&) =0)=0;
15 };
```

Hình 5: **IHeap<T>**: Lớp trừu tượng định nghĩa APIs cho **Heap**.

3.2 Giải thích các APIs

Phần này sẽ mô tả cho từng **pure virtual method** của **IHeap**:

- `virtual ~IHeap() {};`
 - Destructor ảo đảm bảo rằng destructor của các lớp con được gọi khi đối tượng heap bị xóa thông qua con trỏ lớp cơ sở.
- `virtual void push(T item) = 0;`
 - Thêm một phần tử `item` vào heap.
 - **Tham số:**
 - * `T item` — phần tử cần thêm vào heap.
- `virtual T pop() = 0;`
 - Loại bỏ và trả về phần tử **lớn nhất** hoặc **nhỏ nhất** từ heap. Nếu là **max-heap** thì trả về phần tử lớn nhất; ngược lại trả về phần tử nhỏ nhất.
- `virtual const T peek() = 0;`
 - Trả về phần tử lớn nhất/nhỏ nhất từ heap mà không loại bỏ nó.
 - **Giá trị trả về:** Phần tử lớn nhất hoặc nhỏ nhất trong heap.
- `virtual void remove(T item, void (*removeItemData)(T) = 0) = 0;`
 - Xóa phần tử `item` khỏi heap.
 - **Tham số:**
 - * `T item` — phần tử cần xóa.
 - * `void (*removeItemData)(T)` — con trỏ hàm (mặc định là NULL) để xử lý dữ liệu của phần tử cần xóa. Thông thường, nếu kiểu phần tử `T` là con trỏ và người dùng cần giải phóng bộ nhớ của phần tử, thì họ cần truyền vào địa chỉ hàm để giải phóng bộ nhớ cho phần tử.
- `virtual bool contains(T item) = 0;`
 - Kiểm tra xem phần tử `item` có tồn tại trong heap không.
 - **Tham số:** `T item` — phần tử cần kiểm tra.
 - **Giá trị trả về:** `true` nếu phần tử tồn tại, ngược lại `false`.
- `virtual int size() = 0;`
 - Trả về số lượng phần tử hiện có trong heap.
 - **Giá trị trả về:** Số lượng phần tử trong heap.
- `virtual void heapify(T array[], int size) = 0;`
 - Xây dựng heap từ một mảng `array` với kích thước `size`.
 - **Tham số:**
 - * `T array[]` — mảng chứa các phần tử.

- * `int size` — số phần tử trong mảng.
- `virtual void clear() = 0;`
 - Xóa tất cả các phần tử trong heap và đưa heap về trạng thái khởi động. Lưu ý: ở trạng thái khởi động, heap là một array có kích thước là `capacity` phần tử kiểu `T`; mặc nhiên của `capacity` là 10. Ở trạng thái khởi động heap không chứa phần tử nào, nghĩa là **empty**.
- `virtual bool empty() = 0;`
 - Kiểm tra xem heap có rỗng hay không.
 - **Giá trị trả về:** `true` nếu heap rỗng, ngược lại `false`.
- `virtual string toString(string (*item2str)(T&) = 0) = 0;`
 - Trả về chuỗi đại diện cho heap.
 - **Tham số:**
 - * `string (*item2str)(T&)` — con trỏ hàm để chuyển phần tử thành chuỗi.
 - **Giá trị trả về:** Chuỗi mô tả heap.

3.3 Heap

Heap có một tính chất quan trọng; đó là, nó là một mảng (array) của các phần tử khi nhìn ở mức **vật lý**; nghĩa là mức lưu trữ của heap. Tuy nhiên, khi cần làm việc với heap ở mức luận lý, thì nên xem heap là một cây nhị phân **gần đầy đủ (nearly complete)** hoặc **đầy đủ (complete)**.

`Heap<T>` là một cấu trúc dữ liệu dạng heap, nơi các phần tử kiểu `T` được lưu trữ trong một mảng động có kích thước thay đổi tùy theo số lượng phần tử hiện tại. Nguyên lý hoạt động của `Heap<T>` dựa trên việc duy trì tính chất của một heap, nghĩa là mọi nút cha đều phải có giá trị nhỏ hơn hoặc bằng các nút con của nó trong trường hợp **min-heap** (hoặc lớn hơn trong trường hợp **max-heap**).

Để đảm bảo tính chất này, `Heap<T>` sử dụng hai quá trình chính là **reheapUp** và **reheapDown**, giúp cân bằng lại heap khi thêm (push) hoặc xóa (pop) phần tử. Khi thêm phần tử mới vào, phương thức **push** sẽ thêm phần tử vào vị trí cuối của mảng và thực hiện **reheapUp** để di chuyển phần tử này lên đúng vị trí. Tương tự, khi xóa phần tử gốc, phương thức **pop** sẽ di chuyển phần tử cuối lên đầu và thực hiện **reheapDown** để duy trì tính chất heap.

Ngoài các phương thức kế thừa từ `IHeap`, như **push**, **pop**, **peek**, **contains**, và **clear**, `Heap<T>` còn hỗ trợ các phương thức tiện ích khác như **heapify** để biến một mảng thành heap,

`ensureCapacity` để tự động mở rộng mảng khi cần thiết, và `free` để giải phóng dữ liệu người dùng nếu kiểu `T` là con trỏ. Những phương thức này có thể được tìm thấy trong tập tin **Heap.h**, trong thư mục `/include/heap`.

```
1 template<class T>
2 class Heap: public IHeap<T>{
3 public:
4     class Iterator; //forward declaration
5
6 protected:
7     T *elements;    //a dynamic array to contain user's data
8     int capacity;    //size of the dynamic array
9     int count;       //current count of elements stored in this heap
10    int (*comparator)(T& lhs, T& rhs);    //see above
11    void (*deleteUserData)(Heap<T>* pHeap); //see above
12    //HIDDEN CODE
13 };
14
```

Hình 6: `Heap<T>`: Cấu trúc Heap; phần khai báo biến thành viên.

1. Các thuộc tính: Xem Hình 6.

- `int capacity`: Sức chứa hiện tại của heap, khởi động mặc nhiên là 10.
- `int count`: Số lượng phần tử hiện có trong heap.
- `T* elements`: Mảng động lưu trữ các phần tử của heap.
- `int (*comparator)(T& lhs, T& rhs)`: Con trỏ hàm so sánh hai phần tử kiểu `T` để xác định thứ tự của chúng trong heap.
 - Trường hợp `comparator` là `NULL`: lúc đó, (a) kiểu `T` phải hỗ trợ hai phép toán so sánh là `>` và `<`; và (b) `Heap<T>` là một **min-heap**.
 - Trường hợp `comparator` khác `NULL`; muốn `Heap<T>` là **max-heap** thì hàm `comparator` trả về ba giá trị theo quy luật sau:
 - * `+1`: nếu `lhs < rhs`
 - * `-1`: nếu `lhs > rhs`
 - * `0`: trường hợp khác.
 - Trường hợp `comparator` khác `NULL`; muốn `Heap<T>` là **min-heap** thì hàm `comparator` trả về ba giá trị theo quy luật sau:
 - * `-1`: nếu `lhs < rhs`
 - * `+1`: nếu `lhs > rhs`
 - * `0`: trường hợp khác.
- `void (deleteUserData)(Heap<T> pHeap)`: Con trỏ hàm dùng để giải phóng dữ liệu người dùng khi heap không còn được sử dụng. Trong trường hợp kiểu `T` là con trỏ và

người dùng có nhu cầu để heap phải tự giải phóng bộ nhớ của các phần tử thì người dùng cần phải truyền một hàm cho `deleteUserData`, thông qua hàm khởi tạo. Người dùng cũng không cần định nghĩa hàm mới, chỉ cần truyền hàm `Heap<T>::free` vào hàm khởi tạo của `Heap<T>`.

2. Hàm khởi tạo và hàm hủy:

- `Heap(int (*comparator)(T&, T&)=0, void (*deleteUserData)(Heap<T>*)=0)`: Khởi tạo heap rỗng. Heap rỗng là một mảng của `capacity` (10) phần tử kiểu `T`, nhưng có `count = 0`. Khởi gán các biến thành viên `comparator` và `deleteUserData` với các tham số nhận được từ hàm khởi tạo. Lưu ý, tùy vào `comparator` mà heap có thể là **min-heap** hoặc **max-heap**. Nếu không truyền vào `comparator` thì Heap mặc nhiên là **min-heap**. Xem thêm phần giải thích cho các biến thành viên là con trỏ để hiểu rõ về các tham số `comparator` và `deleteUserData`.
- `Heap(const Heap& heap)`: Hàm sao chép, sao chép dữ liệu từ một đối tượng heap khác.
- `~Heap()`: Hàm hủy, giải phóng bộ nhớ và tài nguyên được cấp phát cho heap.

3. Các phương thức:

- `void push(T item)`
 - **Chức năng**: Hàm này chèn một phần tử `item` vào heap và duy trì tính chất của heap (min hoặc max).
 - **Hướng dẫn hiện thực**: các ý chính như sau.
 - (a) Phải gọi hàm `ensureCapacity` để đảm bảo mảng động `elements` có thể chứa `(count + 1)` phần tử.
 - (b) Thêm phần tử vào vị trí cuối của mảng `elements`.
 - (c) Thực hiện quá trình `reheapUp` để đưa phần tử về đúng vị trí, đảm bảo tính chất heap.
 - (d) Tăng giá trị biến `count`.
 - **Ngoại lệ**: Không có.
- `T pop()`
 - **Chức năng**: Hàm này trả về và xóa phần tử gốc (phần tử tại chỉ số 0 trên `elements`, và cũng là phần tử lớn nhất hoặc nhỏ nhất tùy thuộc vào loại heap).
 - **Hướng dẫn hiện thực**: các ý chính như sau.
 - (a) Đưa phần tử ở cuối cùng trên mảng `elements` lên vị trí số 0, phải backup lại phần tử tại 0 để trả về.

- (b) Thực hiện quá trình `reheapDown` để duy trì tính chất heap.
- (c) Cập nhật biến `count`.
- **Ngoại lệ:** Nếu heap rỗng, ném ra ngoại lệ `std::underflow_error("Calling to peek with the empty heap.")`.
- **T peek()**
 - **Chức năng:** Trả về phần tử gốc mà không xóa nó khỏi heap.
 - **Ngoại lệ:** Ném ra ngoại lệ nếu heap rỗng: `throw std::underflow_error("Calling to peek with the empty heap.");`.
- **void remove(T item, void (*removeItemData)(T))**
 - **Chức năng:** Phương thức này xóa một phần tử `item` khỏi heap. Nếu cung cấp hàm `removeItemData`, hàm này sẽ được gọi để giải phóng bộ nhớ hoặc thực hiện các thao tác tùy ý sau khi phần tử bị xóa.
 - **Hướng hiện thực:** các ý chính như sau.
 - (a) Gọi phương thức `getItem` để tìm vị trí của phần tử `item` trong heap. Nếu không tìm thấy phần tử, thoát khỏi phương thức.
 - (b) Nếu phần tử được tìm thấy tại vị trí `foundIdx` trên `elements`: thay thế phần tử tại vị trí `foundIdx` bằng phần tử cuối cùng trong heap (`elements[count - 1]`).
 - (c) Giảm số lượng phần tử (`count`) đi 1.
 - (d) Gọi phương thức `reheapDown` từ vị trí `foundIdx` để khôi phục tính chất của heap.
 - (e) Nếu con trỏ hàm `removeItemData` được cung cấp, gọi hàm này để giải phóng bộ nhớ hoặc xử lý dữ liệu của phần tử đã bị xóa.
 - **Ngoại lệ:** Không có.
- **bool contains(T item)**
 - **Chức năng:** Kiểm tra xem heap có chứa phần tử `value` hay không.
 - **Ngoại lệ:** Không có.
- **int size()**
 - **Chức năng:** Trả về số lượng phần tử hiện có trong heap.
 - **Ngoại lệ:** Không có.
- **void heapify(T array[], int size)**
 - **Chức năng:** Phương thức này xây dựng heap từ một mảng `array` có kích thước `size`.
 - **Hướng hiện thực:**

- (a) Duyệt qua từng phần tử trong mảng `array`.
- (b) Gọi phương thức `push` để thêm từng phần tử vào heap và duy trì tính chất của heap.
 - **Ngoại lệ:** Không có.
- `bool empty()`
 - **Chức năng:** Kiểm tra xem heap có rỗng hay không.
 - **Ngoại lệ:** Không có.
- `void clear()`
 - **Chức năng:** Xóa tất cả các phần tử trong heap và đặt heap về trạng thái rỗng ban đầu.
 - **Ngoại lệ:** Không có.

4 TASK-2: Mạng nơron MLP

4.1 Phương pháp thực hiện

Để hiện thực được mạng nơron, sinh viên cần phải tiến hành **tuần tự** theo các bước sau đây:

1. (**Bắt buộc**): Nghiên cứu tài liệu hướng dẫn về **Mạng nơron nhân tạo**, đã công bố trước.
2. Kết hợp với các hướng dẫn trong tài liệu này, ở các phần sau đây.

4.2 Cấu trúc mã nguồn của phần mạng nơron

Mã nguồn của mạng nơron (ANN) gồm hai dạng tập tin, “.h” và “.cpp”, chứa trong `./include/ann` và `./src/ann` tương ứng. Để hiện thực ANN chúng ta cần nhiều lớp; do đó, nếu đưa hết chúng vào một thư mục sẽ rất khó quản lý. Trong dự án này, các lớp hiện thực cho ANN được tổ chức vào các thư mục con của `./include/ann` và `./src/ann`. Hình 7 và 8 minh hoạ cho nội dung của các lớp nằm trong nhóm “**layer**”. Các thư mục của ANN có nhiệm vụ như được trình bày sau đây. **Lưu ý**, các thư mục có tên sau đây nếu ở `./include/ann` thì chúng chứa tập tin “.h”, còn khi ở `./src/ann` chúng chứa tập tin “.cpp”.

- **layer:** Thư mục này chứa định nghĩa các lớp tính toán để xây dựng mạng nơron. Các lớp được hiện thực trong **TASK-2** gồm:

Folders	Folders	Folders	Developer
<ul style="list-style-type: none"> datasets demo include models src 	<ul style="list-style-type: none"> ann graph hash heap list loader sformat sorting stacknqueue tensor tree util 	<ul style="list-style-type: none"> config dataset layer loss metrics model modelzoo optim 	<ul style="list-style-type: none"> FCLayer.h ILayer.h ReLU.h Sigmoid.h Softmax.h Tanh.h
<p>Documents</p> <ul style="list-style-type: none"> config.txt 	<p>Developer</p> <ul style="list-style-type: none"> dsaheader.h 	<p>Developer</p> <ul style="list-style-type: none"> annheader.h functions.h 	

Hình 7: Tổ chức mã nguồn của ANN, tập tin “.h”

Folders	Folders	Folders	Developer
<ul style="list-style-type: none"> datasets demo include models src 	<ul style="list-style-type: none"> ann tensor 	<ul style="list-style-type: none"> config dataset layer loss metrics model modelzoo optim 	<ul style="list-style-type: none"> FCLayer.cpp Layer.cpp ReLU.cpp Sigmoid.cpp Softmax.cpp Tanh.cpp
<p>Documents</p> <ul style="list-style-type: none"> config.txt 	<p>Developer</p> <ul style="list-style-type: none"> program.cpp 	<p>Developer</p> <ul style="list-style-type: none"> functions.cpp 	

Hình 8: Tổ chức mã nguồn của ANN, tập tin “.cpp”

1. Lớp **ILayer**: là lớp trừu tượng, định nghĩa các phương thức **virtual** để các lớp con **override**. Lớp này là lớp cha của tất cả các lớp sau đây.
 2. Lớp kết nối đầy đủ, **FCLayer**.
 3. Lớp **ReLU**.
 4. Lớp **Sigmoid**.
 5. Lớp **Tanh**.
 6. Lớp **Softmax**.
- **loss**: Thư mục này chứa tất cả các lớp định nghĩa các hàm tổn thất khác nhau. Các lớp được hiện thực trong **TASK-2** gồm:
 1. Lớp **ILossLayer**: là lớp trừu tượng, định nghĩa các phương thức **virtual** để các lớp con **override**. Lớp này là lớp cha của tất cả các lớp sau đây.
 2. Lớp **CrossEntropy**.

- **metrics**: Thư mục này chứa tất cả các lớp định nghĩa các loại metrics (độ đo) khác nhau cho các bài toán khác nhau. Các lớp được hiện thực trong **TASK-2** gồm:
 1. Lớp **IMetrics**: là lớp trừu tượng, định nghĩa các phương thức **virtual** để các lớp con **override**. Lớp này là lớp cha của tất cả các lớp sau đây.
 2. Lớp **ClassMetrics**: **ClassMetrics** là tên viết tắt của **Classification Metrics**, là lớp tính toán các độ đo phổ thông để đánh giá hiệu quả cho **bài toán phân loại**.
- **model**: Thư mục này chứa tất cả các lớp định nghĩa các kiểu mô hình mạng nơron khác nhau; ví dụ như, mô hình phân loại đơn nhãn, phân loại đa nhãn, hồi quy, v.v. Các lớp được hiện thực trong **TASK-2** gồm:
 1. Lớp **IModel**: là lớp trừu tượng, định nghĩa các phương thức **virtual** để các lớp con **override**. Lớp này là lớp cha của tất cả các lớp sau đây.
 2. Lớp **MLPClassifier**: là mô hình mạng nơron nhiều lớp, làm nhiệm vụ phân loại đơn nhãn.
- **optim**: Thư mục này chứa tất cả các lớp định nghĩa cho các giải thuật huấn luyện mạng nơron khác nhau, v.v. Các lớp được hiện thực trong **TASK-2** gồm:
 1. Lớp **IParamGroup**: là lớp trừu tượng, định nghĩa các phương thức **virtual** để các lớp con **override**. Lớp này là lớp cha của tất cả các lớp sau đây. Hàm **step** trong các lớp sau đây chính là hàm trực tiếp thực hiện việc cập nhật các tham số trong nhóm trong quá trình học.
 - **SGDParamGroup**
 - **AdaParamGroup**
 - **AdamParamGroup**
 2. Lớp **IOptimizer**: là lớp trừu tượng, định nghĩa các phương thức **virtual** để các lớp con **override**. Lớp này là lớp cha của tất cả các lớp sau đây. Hàm **create_group** trong các lớp sau đây sẽ tạo ra các đối kiểu **SGDParamGroup**, **AdaParamGroup** hay **AdamParamGroup** để quản lý tham số, tùy theo chiến lược cập nhật của mình.
 - **SGD**
 - **Adagrad**
 - **Adam**
- **dataset**: Thư mục này chứa lớp **DSFactory** dùng để tạo ra một số dataset phổ biến.
- **config**: Thư mục này chứa lớp **Config** dùng để quản lý các tham số cấu hình trong quá trình tạo ra mạng nơron; như, thư mục chứa mô hình, tên checkpoint để các mô hình trong lúc huấn luyện, v.v.

- **modelzoo**: Thư mục này chứa một số ví dụ minh họa về việc tạo mô hình và huấn luyện mô hình.

4.3 Thứ tự hiện thực

Sinh viên nên hiện thực các lớp theo thứ tự được gợi ý sau đây:

1. Các lớp trong thư mục **layer**.
2. Các lớp trong thư mục **loss**.
3. Các lớp trong thư mục **metrics**.
4. Các lớp trong thư mục **model**.
5. Các lớp trong thư mục **optim**.