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

CÂY HUFFMAN ĐA NHÁNH VÀ BỘ NÉN DỮ LIỆU KHO HÀNG

TP. HỒ CHÍ MINH, THÁNG 04/2025

ĐẶC TẢ BÀI TẬP LỚN

Phiên bản 1.1

1 Giới thiệu

1.1 Mục tiêu và nhiệm vụ

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 60% 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 40% cột điểm): Nội dung này yêu cầu sinh viên sử dụng các cấu trúc dữ liệu đã học để phát triển cây Huffman đa nhánh và ứng dụng thực tế.

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**. Sinh viên tái sử dụng hiện thực đã dùng cho BTL 1.
3. **Sử dụng** danh sách được hiện ở trên và phát triển các lớp ứng dụng **List1D**, **List2D** và **InventoryManager**. Sinh viên tái sử dụng hiện thực đã dùng cho BTL 1.
4. **Phát triển** cấu trúc bảng băm và Heap: Hiện thực lớp **xMap** và **Heap** trong thư mục **/include/hash** và **/include/heap**.
5. **Sử dụng** cấu trúc bảng băm và Heap để hiện thực các lớp ứng dụng **HuffmanTree** và **InventoryCompressor**.
6. **Chạy thử**: Kiểm tra chương trình phải qua được các testcases mẫu được cung cấp.

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

1.3 Chuẩn đầu ra của BTL-2

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 bảng băm và cấu trúc dữ liệu Hash.
- **Lựa chọn và sử dụng** được các cấu trúc dữ liệu trên để hiện thực các lớp **HuffmanTree** và **InventoryCompressor**.

1.4 Thành phần điểm và phương pháp chấm bài

1. Hiện thực (60% **điểm**): gồm tập tin sau đây
   - **/include/hash/xMap.h** (40%)
   - **/include/heap/Heap.h** (20%)
2. Hiện thực các lớp về Huffman Tree và InventoryCompressor: (40% **điểm**); gồm các tập tin sau đây trong thư mục **/include/app**:
   - **inventory_compressor.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 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 1), 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 1: 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 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 (colllision) 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 1: `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 (*deleteVal = 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 bằm 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 2.
   - `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 **đụng 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

```
template<class K, class V>
class xMap: public IMap<K,V>{
public:
    class Entry; //forward declaration
protected:
    DLinkedList<Entry* >* table;  //array of DLinkedList objects
    int capacity;   //size of table
    int count;      //number of entries stored hash-map
    float loadFactor; //define max number of entries can be stored (< (loadFactor * capacity))

    int (*hashCode)(K&,int); //hasCode(K key, int tableSize): tableSize means capacity
    bool (*keyEqual)(K&,K&);  //keyEqual(K& lhs, K& rhs): test if lhs == rhs
    bool (*valueEqual)(V&,V&); //valueEqual(V& lhs, V& rhs): test if lhs == rhs
    void (*deleteKeys)(xMap<K,V>*); //deleteKeys(xMap<K,V>* pMap): delete all keys stored in pMap
    void (*deleteValues)(xMap<K,V>*); //deleteValues(xMap<K,V>* pMap): delete all values stored in pMap

    //HIDDEN CODE
public:
    //Entry: BEGIN
    class Entry{
    private:
        K key;
        V value;
        friend class xMap<K,V>;

    public:
        Entry(K key, V value){
            this->key = key;
            this->value = value;
        }
    };
    //Entry: END
};
```

Hình 2: `xMap<T>`: Cấu trúc bảng băm; phần khai báo thuộc tính

(`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`$(0.75 \times 10) = 7$; 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 tuỳ vào nhu cầu của người sử dụng thư viện.

- `int (*hashCode)(K&,int)`: hàm này nhận vào **key** (truyền bằng tham khảo) và 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 $[0, 1, \cdots, m-1]$; 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.
       (a) Sử dụng hàm băm `hashCode` để tính toán địa chỉ của `key`.
       (b) Lấy danh sách từ `table` bằng địa chỉ ở trên.
       (c) 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 dẫb 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, 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 chp `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 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 3), 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 3: 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 `T` 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`.

```
template<class T>
class IHeap {
public:
    virtual ~IHeap(){};
    virtual void push(T item)=0;
    virtual T pop()=0;
    virtual const T peek()=0;
    virtual void remove(T item, void (*removeItemData)(T)=0)=0;
    virtual bool contains(T item)=0;
    virtual int size()=0;
    virtual void heapify(T array[], int size)=0; //build heap from array
   having size items
    virtual void clear()=0;
    virtual bool empty()=0;
    virtual string toString(string (*item2str)(T&) =0)=0;
};
```

Hình 3: `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**.

Một cựu sinh viên trường B đã hiện thực cấu trúc Heap trong initial code. File code này chỉ dùng để tham khảo, sinh viên cần phải điều chỉnh file code mẫu để đảm bảo thỏa mãn các mô tả bên dưới.

```
template<class T>
class Heap: public IHeap<T>{
public:
    class Iterator; //forward declaration

protected:
    T *elements;    //a dynamic array to contain user's data
    int capacity;   //size of the dynamic array
    int count;      //current count of elements stored in this heap
    int (*comparator)(T& lhs, T& rhs);      //see above
    void (*deleteUserData)(Heap<T>* pHeap); //see above
    //HIDDEN CODE
};

```

Hình 4: `Heap<T>`: Cấu trúc Heap; phần khai báo biến thành viên.

1. **Các thuộc tính:** Xem Hình 4.
   - `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, tuỳ 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).
        - **Độ phức tạp**: O(log(n))
        - **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).
        - **Độ phức tạp**: O(log(n))
        - **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.

  - **Độ phức tạp**: O(1)
  - **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.
  - **Độ phức tạp**: O(log(n))
  - **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`.
  - **Độ phức tạp**: O(n)
  - **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.
- **void heapsort(XArrayList<T>& arrayList**
  - **Chức năng**: Sắp xếp danh sách arrayList sử dụng cấu trúc Heap. Nếu cấu trúc Heap đã có dữ liệu, thay dữ liệu hiện tại bằng dữ liệu của arrayList. Sau mỗi lần Heap Up, sinh viên in ra các phần tử hiện tại trong Heap.

4 Cây Huffman đa nhánh và Bộ Nén Kho Hàng

BTL này yêu cầu sinh viên hiện thực một cây Huffman N-nhánh và việc ứng dụng nó trong quá trình nén và giải nén dữ liệu kho hàng. Sinh viên có thể tìm đọc lý thuyết và giải thuật cây Huffman từ những nguồn sau trước khi đọc phần bên dưới: [1], [2].

4.1 Lý thuyết về cây Huffman đa nhánh

Cây Huffman đa nhánh là sự mở rộng của cây Huffman nhị phân truyền thống, cho phép mỗi nút nội có tới n nút con thay vì chỉ 2 nút. Một đặc điểm quan trọng của cây Huffman đa nhánh là trong trường hợp số lượng ký tự ban đầu không đủ để xây dựng cây hoàn chỉnh theo nhóm n, ta phải bổ sung các ký tự giả (dummy characters) (ký tự $'\backslash 0'$) với tần số bằng 0, nhằm đảm bảo rằng số lượng nút lá L sau khi bổ sung thỏa mãn điều kiện:

$$(L-1) \mod (n-1) = 0.$$

Các ký tự giả này không mang ý nghĩa dữ liệu mà chỉ dùng để đảm bảo mã hóa được sinh ra từ cây đạt tối ưu.

4.1.1 Quá trình xây dựng cây Huffman đa nhánh:

1. **Tính tần số:** Xác định tần số xuất hiện của mỗi ký tự trong dữ liệu cần mã hóa.
2. **Tạo nút lá:** Với mỗi ký tự thực, tạo một nút lá chứa ký tự và tần số tương ứng.
3. **Bổ sung ký tự giả (nếu cần):** Nếu số nút lá ban đầu không thỏa mãn điều kiện $(L-1) \mod (n-1) = 0$, thêm vào các ký tự giả với tần số 0 cho đến khi điều kiện trên được thỏa mãn.
4. **Xây dựng cây:** Lặp lại quá trình sau cho đến khi chỉ còn một nút trong heap:
   (a) Lấy ra n nút có tần số nhỏ nhất từ heap.
   (b) Tính tổng tần số của các nút được lấy ra.
   (c) Tạo một nút nội mới với tổng tần số vừa tính và gán các nút đó làm các nút con.
   (d) Đẩy nút nội mới vào heap.
5. **Xác định nút gốc:** Nút duy nhất còn lại trong heap là nút gốc của cây Huffman.

Sau khi xây dựng xong cây, ta có thể gán mã cho các ký tự bằng cách duyệt từ nút gốc xuống các nút lá. Trong thực tế, các mã này sẽ là chuỗi bit để giảm kích thước lưu trữ cho văn bản. Tuy nhiên, trong bài tập lớp này, để đơn giản hóa, ta biểu diễn mã dưới dạng một chuỗi

các kí tự từ '0' đến 'f' (theo hệ 16). Ví dụ, nếu cây huffman có 3 nhánh quy ước gán mã như sau:

- Nút con thứ nhất: mã '0'
- Nút con thứ hai: mã '1'
- Nút con thứ ba: mã '2'

Mã Huffman của các ký tự được tạo thành bằng cách kết hợp các mã số của các nút trên đường đi từ nút gốc đến nút chứa ký tự tương ứng. Sinh viên tham khảo ví dụ minh họa để hiểu rõ hơn

4.1.2 Ví dụ minh họa

Giả sử ta có 4 ký tự với tần số xuất hiện như sau:

- A: 5
- B: 9
- C: 12
- D: 13

Với cây Huffman 3-nhánh (mỗi nút nội có tối đa 3 nút con), ta cần số nút lá sao cho:

$$(L-1) \mod (3-1) = 0.$$

Với $L = 4$, ta có $(4-1) \mod 2 = 3 \mod 2 = 1 \neq 0$. Do đó, ta cần bổ sung $d = (2 - (3 \mod 2)) = 2 - 1 = 1$ ký tự giả có tần số 0. Sau khi bổ sung, số nút lá trở thành $L = 4 + 1 = 5$, và $(5-1) \mod 2 = 4 \mod 2 = 0$.

Danh sách các nút lá ban đầu gồm:

- \0 (ký tự giả): 0
- A: 5
- B: 9
- C: 12
- D: 13

Tiến hành xây dựng cây Huffman 3-nhánh như sau:

1. **Bước 1:** Lấy ra 3 nút có tần số nhỏ nhất:

- Dummy: 0
- A: 5
- B: 9

Tổng tần số $= 0 + 5 + 9 = 14$.

Tạo một nút nội mới với tần số 14, gán các nút `Dummy`, A và B làm các nút con.

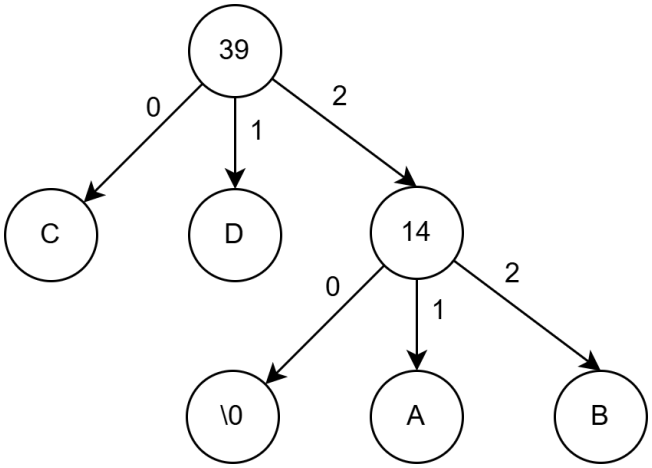
2. **Bước 2:** Sau khi loại bỏ 3 nút vừa xử lý, ta còn lại:

- Nút nội với tần số 14
- C: 12
- D: 13

Lấy ra 3 nút này, tổng tần số $= 14 + 12 + 13 = 39$.

Tạo một nút nội mới với tần số 39, gán các nút có tần số 14, 12, và 13 làm các nút con.

Nút nội có tần số 39 là nút duy nhất còn lại trong heap, do đó nó trở thành nút gốc của cây Huffman. Hình 5 thể hiện cây Huffman 3-nhánh sau khi xây với ví dụ trên.



Hình 5: Cây huffman 3-nhánh sau khi xây

Thông qua cây Huffman 3 nhánh, ta thu được đoạn mã của mỗi ký tự sau khi mã hóa như sau:

- A: "21"
- B: "22"
- C: "0"
- D: "1"

4.2 HuffmanTree

`HuffmanTree<treeOrder>` là template class quản lý cây Huffman đa nhánh dùng để mã hóa và giải mã các ký tự dựa trên tần suất xuất hiện của chúng.

```
template<int treeOrder>
class HuffmanTree {
public:
    struct HuffmanNode {
        char symbol;
        int freq;
        XArrayList<HuffmanNode*> children;

        HuffmanNode(char s, int f); //Leaf node
        HuffmanNode(int f, const  XArrayList<HuffmanNode*>& childs); //
    Internal node
    };

    HuffmanTree();
    ~HuffmanTree();

    void build(XArrayList<pair<char, int>>& symbolsFreqs);
    void generateCodes(xMap<char, std::string>& table);
    std::string decode(const std::string& huffmanCode);

private:
    HuffmanNode* root;
};
```

Các thuộc tính:

- `HuffmanNode* root`: Con trỏ đến nút gốc của cây Huffman.
- `struct HuffmanNode`: Cấu trúc lồng bên trong đại diện cho một nút trong cây Huffman.
  - `char symbol`: Lưu trữ ký tự (áp dụng cho nút lá).
  - `int freq`: Tần số xuất hiện của ký tự, hoặc tổng tần số của các nút con (áp dụng cho nút nội).
  - `XArrayList<HuffmanNode*> children`: Danh sách chứa các nút con; với nút nội, danh sách này chứa các nút được hợp nhất từ heap.

Các phương thức:

1. **void build(XArrayList<pair<char, int»& symbolsFreqs)**

- **Chức năng:** Xây dựng cây Huffman đa nhánh từ danh sách các cặp ký tự – tần số.
- **Yêu cầu hiện thực:**
  (a) Tạo một heap từ danh sách truyền vào.
  (b) Trong khi heap có nhiều hơn một nút, chọn ra tối đa `treeOrder` nút có tần số nhỏ nhất.
  (c) Tính tổng tần số của các nút được chọn và gom chúng lại thành danh sách con.
  (d) Tạo một nút nội mới với tổng tần số vừa tính và danh sách các nút con.
  (e) Đẩy nút nội vừa tạo vào heap.
  (f) Sau vòng lặp, nút cuối cùng còn lại sẽ trở thành nút `root` của cây.
- **Lưu ý**: Để đảm bảo tính nhất quán, khi 2 nút trong Heap có cùng tần số, nút nhỏ hơn là nút có thời điểm được đưa vào Heap sớm hơn.

2. **void generateCodes(xMap<char, std::string>& table)**
   - **Chức năng:** Tạo mã Huffman cho mỗi ký tự có trong cây. Lưu chuỗi mã tích lũy vào `table` với khóa là ký tự của nút.
3. **std::string decode(const std::string& huffmanCode)**
   - **Chức năng:** Giải mã chuỗi mã Huffman về văn bản ban đầu.

4.3 InventoryCompressor<treeOrder>

`InventoryCompressor<treeOrder>` là class template sử dụng cây Huffman để nén và giải nén dữ liệu kho hàng. Số nhánh `treeOrder` được sử dụng thống nhất trong quá trình xây dựng cây Huffman.

```
template<int treeOrder>
class InventoryCompressor {
public:
    InventoryCompressor(InventoryManager* manager);
    ~InventoryCompressor();

    void buildHuffman();
    void printHuffmanTable();
    std::string productToString(const List1D<InventoryAttribute>& attributes
    , const std::string& name);
    std::string encodeHuffman(const List1D<InventoryAttribute>& attributes,
    const std::string& name);
    std::string decodeHuffman(const std::string& huffmanCode, List1D<
    InventoryAttribute>& attributesOutput, std::string& nameOutput);

```

```
13 private:
14     xMap<char, std::string> huffmanTable;
15     InventoryManager* invManager;
16     HuffmanTree<treeOrder>* tree;
17 };
```

Các thuộc tính:

- `InventoryManager* invManager`: Con trỏ trỏ đến đối tượng `InventoryManager` cung cấp thông tin tên sản phẩm và các thuộc tính của kho hàng.
- `xMap<char, std::string> huffmanTable`: Bảng băm chứa các mã Huffman tương ứng với từng ký tự.
- `HuffmanTree<treeOrder>* tree`: Con trỏ trỏ đến cây Huffman được xây dựng dùng để mã hóa và giải mã.

Hàm khởi tạo và hàm hủy:

- `InventoryCompressor(InventoryManager* manager)`: Khởi tạo bộ nén với đối tượng `InventoryManager`.
- `~InventoryCompressor()`: Hủy đối tượng và giải phóng bộ nhớ được cấp phát cho cây Huffman.

Các phương thức:

1. **void buildHuffman()**
   - **Chức năng:** Xây dựng cây Huffman và tạo các mã Huffman dựa trên tần số xuất hiện của các ký tự trong kho hàng.
   - **Yêu cầu hiện thực:**
     (a) Duyệt qua tất cả sản phẩm để xây dựng bảng tần số cho từng ký tự trong biểu diễn dạng chuỗi của sản phẩm thông qua phương thức `productToString`.
     (b) Tạo bảng mã `huffmanTable` bằng cách xây dựng cây Huffman với bảng tần số trên.
2. **std::string productToString(const List1D<InventoryAttribute>& attributes, const std::string& name)**
   - **Chức năng:** Chuyển đổi dữ liệu sản phẩm thành chuỗi với định dạng: *"tên sản phẩm:(thuộc tính1:giá trị1), (thuộc tính2:giá trị2), ..."*.
3. **std::string encodeHuffman(const List1D<InventoryAttribute>& attributes, const std::string& name)**

- **Chức năng:** Mã hóa chuỗi sản phẩm thành chuỗi mã Huffman.
- **Yêu cầu hiện thực:** Chuyển đổi dữ liệu sản phẩm thành chuỗi dùng `productToString` sau đó tạo chuỗi mã từ `huffmanTable`

4. **std::string decodeHuffman(const std::string& huffmanCode, List1D<InventoryAttribute attributesOutput, std::string& nameOutput)**
   - **Chức năng:** Giải mã chuỗi mã Huffman để khôi phục dữ liệu sản phẩm ban đầu.
   - **Yêu cầu hiện thực:**
     (a) Giải mã chuỗi sử dụng cây Huffman.
     (b) Phân tích chuỗi kết quả để lấy ra tên sản phẩm và danh sách các thuộc tính. Format của chuỗi trả về là: *"tên sản phẩm: (thuộc tính1: giá trị1), (thuộc tính2: giá trị2), ..."*.
     (c) Phương thức trả về chuỗi sau khi mã hóa. Đồng thời, tên sản phẩm và danh sách thuộc tính lần lượt được gán vào nameOutput và attributesOutput

5 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.

5.1 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/test/* src/main.cpp -o main**

5.2 Nộp bài

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

6 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.

7 Theo dõi các thay đổi qua các phiên bản

(v1.1)

- Cập nhật mô tả cho cấu trúc dữ liệu Heap.
- Cập nhật lại initial code cho cấu trúc dữ liệu Heap
- Cập nhật thành phần điểm

——————————HẾT——————————