VIETNAM NATIONAL UNIVERSITY - HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

DATA STRUCTURES AND ALGORITHMS - CO2003

Assignment 2

N-ARY HUFFMAN TREE AND THE INVENTORY DATA COMPRESSOR

Ho Chi Minh City, 04/2025

ASSIGNMENT'S SPECIFICATION

Version 1.1

1 Introduction

1.1 Objectives and Tasks

The second assignment for the Data Structures and Algorithms course consists of the following two parts:

1. **Task-1** (accounts for 60% of the total score): This part requires students to develop the following two data structures:
    (a) **Hash Table (HashMap)**. The file to implement for this part is **./include/hash/xMap.h**. Students should read the implementation guidelines for this part in Section 2.
    (b) **Heap**. The file to implement for this part is **./include/heap/Heap.h**. Students should refer to the implementation guidelines for this part in Section 3.
2. **Task-2** (accounts for 40% of the total score): This part requires students to use the developed data structures to implement a multi-way Huffman tree and its practical application.

1.2 Methodology

1. **Preparation: Download** and **review** the provided source code. **Note:** Students must compile the source code using **C++17**. The compiler has been tested with **g++**; it is recommended that students set up their environment to use **g++**.
2. **Development of lists:** Implement the classes **XArrayList** and **DLinkedList** in the directory /**include**/**list**. Students may reuse the implementations from Assignment 1.
3. **Utilization:** Use the above lists to develop the application classes **List1D**, **List2D**, and **InventoryManager**. Students may reuse the implementations from Assignment 1.
4. **Development of the Hash Table and Heap:** Implement the classes **xMap** and **Heap** in the directories /**include**/**hash** and /**include**/**heap**, respectively.
5. **Application:** Use the Hash Table and Heap structures to implement the application classes **HuffmanTree** and **InventoryCompressor**.
6. **Testing:** The program must pass the sample test cases provided.
7. **Submission:** The assignment must be submitted on the system before the deadline.

1.3 Expected Outcomes of Assignment-2

Upon completing this assignment, students will be able to:

- Proficiently use advanced features of the C/C++ programming languages.
- Develop data structures such as the Hash Table and Heap.
- Select and utilize the above data structures to implement the classes **HuffmanTree** and **InventoryCompressor**.

1.4 Grading Criteria and Method

1. **Implementation** (60% **of the total score**): Includes the following files:
   - **/include/hash/xMap.h** (40%)
   - **/include/heap/Heap.h** (20%)
2. **Implementation of the Huffman Tree and InventoryCompressor classes** (40% **of the total score**): Includes the following files in the **/include/app** directory:
   - **inventory_compressor.h**

Students will be evaluated using sample test cases announced at submission time. After submission, additional hidden test cases will be used for grading.

2 Hash Map Structure

2.1 Design Principles

Similar to other data structures, the implementation of the hash map in this library consists of two classes: (a) the `IMap` class (see Figure 1), used to define the APIs for the hash map; and (b) the `xMap` class (a subclass of `IMap`) that contains the concrete implementation of the hash map.

- **The `IMap` Class, see Figure 1:** This class defines a set of methods (APIs) supported by the hash map.
  - `IMap` uses **templates** to parameterize the element data types; specifically, the type of **key** and the type of **value**. Therefore, the hash map can work with any types for **key** and **value**.

  - All APIs in `IMap` are declared as **pure virtual methods**; that is, any class inheriting from `IMap` must **override** all of these methods. Because they are **virtual**, the APIs will support dynamic binding (i.e., polymorphism).

- **The `xMap` Class:** This class is inherited from `IMap`. It contains the concrete implementation for all the APIs defined in `IMap` by using a **doubly linked list (DoublyLinkedList)** to hold all key-value pairs that experience collisions at each index in the hash table. In other words, the hash map in this library is a table of doubly linked lists.

```cpp
template<class K, class V>
class IMap {
public:
    virtual ~IMap(){};
    virtual V put(K key, V value)=0;
    virtual V& get(K key)=0;
    virtual V remove(K key, void (*deleteKeyInMap)(K)=0)=0;
    virtual bool remove(K key, V value, void (*deleteKeyInMap)(K)=0, void (*
  deleteValueInMap)(V)=0)=0;
    virtual bool containsKey(K key)=0;
    virtual bool containsValue(V value)=0;
    virtual bool empty()=0;
    virtual int size()=0;
    virtual void clear() = 0;
    virtual string toString(string (*key2str)(K&)=0, string (*value2str)(V&)
  =0 )=0;
    virtual DLinkedList<K> keys()=0;
    virtual DLinkedList<V> values()=0;
    virtual DLinkedList<int> clashes()=0;
};
```

Figure 1: `IMap<T>`: Abstract class defining APIs for the hash map.

2.2 Explanation of the APIs

Below is a description for each pure virtual method of `IMap`:

- `virtual ~IMap() {};`
  - Virtual destructor, ensuring that the destructors of derived classes are called when an object is deleted through a pointer to an `IMap` object.

- `virtual V put(K key, V value) = 0;`
  - **Parameters:**
    * `K key` — the key to be added or updated.

    - `V value` — the value to be added or replaced.
  - **Return:**
    - If the `key` is already present in the map, it returns the old value (currently in the map); otherwise, it returns the new value provided.
  - **Purpose:**
    - To add a key-value pair to the hash map. If the key already exists, update it with the new value and return the old value.

- `virtual V& get(K key) = 0;`
  - **Purpose:** Returns the value mapped by the given key.
  - **Parameter:** `K key` — the key for which the value is requested.
  - **Return:** A reference to the value corresponding to the key.
  - **Exception:** Throws the `KeyNotFound` exception if the key does not exist.

- `virtual V remove(K key, void (*deleteKeyInMap)(K) = 0) = 0;`
  - **Purpose:** Removes the entry containing `key` from the map and returns its associated value.
  - **Parameters:**
    - `K key` — the key to be removed.
    - `void (*deleteKeyInMap)(K)` — a function pointer (default is NULL) called to delete the key if `K` is a pointer.
  - **Return:** The value corresponding to the key.
  - **Exception:** Throws a `KeyNotFound` exception if the key is not found.

- `virtual bool remove(K key, V value, void (*deleteKeyInMap)(K) = 0, void (*deleteVal`
  `= 0) = 0;`
  - **Purpose:** Removes the `<key, value>` pair if it exists.
  - **Parameters:**
    - `K key` — the key to be removed.
    - `V value` — the value to be removed.
    - `void (*deleteKeyInMap)(K)` — function pointer to delete the key (default is NULL).
    - `void (*deleteValueInMap)(V)` — function pointer to delete the value (default is NULL).
  - **Return:** `true` if the `<key, value>` pair was removed, `false` otherwise.

- `virtual bool containsKey(K key) = 0;`

  - **Purpose:** Checks whether the map contains the given key.
  - **Parameter:** `K key` — the key to check.
  - **Return:** `true` if the key exists, otherwise `false`.

- `virtual bool containsValue(V value) = 0;`
  - **Purpose:** Checks whether the hash map contains the given value at any index.
  - **Parameter:** `V value` — the value to check.
  - **Return:** `true` if the value exists, otherwise `false`.

- `virtual bool empty() = 0;`
  - **Purpose:** Checks if the map is empty.
  - **Return:** `true` if the map is empty, otherwise `false`.

- `virtual int size() = 0;`
  - **Purpose:** Returns the number of key-value pairs in the map.
  - **Return:** The number of elements in the map. A return of 0 indicates an empty map. **Note:** An empty map consists of an array of linked lists with `capacity` (default value of `capacity` is 10), with each linked list being empty.

- `virtual void clear() = 0;`
  - **Purpose:** Removes all key-value pairs from the hash map and resets it to an empty state. **Note:** An empty map is one that contains `capacity` (default value of `capacity` is 10) linked lists, with each list being empty.

- `virtual string toString(string (*key2str)(K&) = 0, string (*value2str)(V&) = 0) = 0;`
  - **Purpose:** Returns a string that represents the map.
  - **Parameters:**
    * `string (*key2str)(K&)` — a function pointer to convert a key to a string. If this pointer is `NULL`, then `K` must support the output operator (`<<`) to convert the key to a string.
    * `string (*value2str)(V&)` — a function pointer to convert a value to a string. If this pointer is `NULL`, then `V` must support the output operator (`<<`) to convert the value to a string.
  - **Return:** A string describing the hash map.

- `virtual DLinkedList<K> keys() = 0;`
  - **Purpose:** Returns a doubly linked list containing all the keys in the map.
  - **Return:** A `DLinkedList<K>` containing the keys.

- `virtual DLinkedList<V> values() = 0;`
  - **Purpose:** Returns a doubly linked list containing all the values in the map.
  - **Return:** A `DLinkedList<V>` containing the values.
- `virtual DLinkedList<int> clashes() = 0;`
  - **Purpose:** Returns a doubly linked list containing the number of collisions at each index in the map.
  - **Return:** A `DLinkedList<int>` with the collision counts.

2.3 Hash Map (xMap<K, V>)

`xMap<K, V>` is a concrete implementation of the hash map, where elements are stored as key-value pairs (`K, V`) in a pre-determined sized array, called the *table*. The operation of `xMap<K, V>` is based on using a hash function to determine the storage location of each element in the array.

To ensure efficient performance, `xMap<K, V>` must maintain an array large enough to hold the key-value pairs and must properly manage the load factor, that is, the ratio between the current number of elements and the size of the array. If this ratio exceeds the specified `loadFactor`, the hash map will automatically perform rehashing; that is, increase the array size and redistribute the elements.

In addition to the methods inherited from `IMap` for basic operations such as `put`, `get`, `remove`, `containsKey`, and `clear`, `xMap<K, V>` also supports additional utility methods such as `rehash` for expanding the hash table, and `ensureLoadFactor` to maintain the load factor. These methods can be found in the file **xMap.h** in the directory /**include**/**hash**.

1. **Properties:** See Figure 2.
   - `int capacity`: The current capacity of the hash map.
   - `int count`: The number of elements currently in the hash map.
   - `DLinkedList<Entry* >* table`: The hash map is a dynamic array of doubly linked list objects, each element of which is a **pointer** to an `Entry` (where `Entry` is the class containing the `<key, value>` pair). One may visualize the hash map as an array of list objects, where each list contains all the colliding pairs at that particular index.
   - `float loadFactor`: The load factor of the hash map that indicates the level of space usage before rehashing is necessary. At any time, the number of elements in the map

```cpp
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 in the hash map
    float loadFactor;             // defines the maximum number of entries to
     be stored (< (loadFactor * capacity))

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

Figure 2: `xMap<T>`: Hash map structure; declaration of member variables.

(`count`) **must not exceed** `loadFactor` × `capacity`. For example, if `capacity` $= 10$ and `loadFactor` $= 0.75$, the maximum number of elements is `int`$(0.75 \times 10) = 7$; when the 8th element is inserted, `ensureLoadFactor` must be called to maintain the load factor.

- Function pointers: These properties are initialized via the constructor. **Note:** Only `hashCode` must always be passed into the constructor (and hence must not be `NULL`); the other function pointers may be `NULL` depending on the needs of the user.
  - `int (*hashCode)(K&,int)`: This function receives a **key** (passed by reference)

and the hash table size (an integer) and computes the index of the key in the hash map. **Note:** If the table size passed into the function is m, then the return value of `hashCode` must lie within the range $[0, 1, \cdots, m-1]$; to ensure this, `hashCode` must use the modulo operator (%). See also the functions `intKeyHash` and `stringKeyHash` provided in the source code.

- `bool (*keyEqual)(K&,K&)`: In case the key data type (`K`) does not support the equality operator (==) for comparing two keys, the user must provide a pointer to a function that can compare two keys for equality. This function takes two keys of type `K` (by reference) and returns `true` if they are equal, and `false` otherwise. **Note:** Within the **xMap** class, when comparing two keys, the `keyEQ` method should be used.
- `bool (*valueEqual)(V&,V&)`: In case the value data type (`V`) does not support comparison using the equality operator (==), the user must supply a pointer to a function that compares two values for equality. This function takes two values of type `V` (by reference) and returns `true` if they are equal, and `false` otherwise. **Note:** Within the **xMap** class, when comparing two values, the `valueEQ` method should be used.
- `void (*deleteKeys)(xMap<K,V>* pMap)`: In case `K` is a pointer and the user needs **xMap** to actively manage (release) the memory for keys, the user **MUST** pass a function pointer via `deleteKeys`. The user does not need to define a new function; it is sufficient to pass `xMap<K,V>::freeKey` to `deleteKeys`. Refer to the source code of `xMap<K,V>::freeKey` for details.
- `void (*deleteValues)(xMap<K,V>* pMap)`: Similarly, if `V` is a pointer and the user requires **xMap** to manage (release) the memory for values, the user **MUST** pass a function pointer via `deleteValues`. The user does not need to define a new function; simply pass `xMap<K,V>::freeValue` as `deleteValues`. See the source code of `xMap<K,V>::freeValue` for details.

2. **Constructor and Destructor:**

-
  ```
  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)*)):
  ```

- **Purpose**: The constructor creates an empty hash map with `capacity` linked lists in the table. The default value of `capacity` is 10. This constructor also initializes the function pointer member variables with the passed-in values.
- **Parameters**: See the property descriptions above.

- `xMap(const xMap<K,V>& map)`: This constructor copies data from another hash map object.
- `~xMap()`: The destructor frees all memory and resources allocated for the hash map, including all `keys`, `values` and `entries` if present or if required by the user.

3. **Methods:**

- **V put(K key, V value)**
  - **Purpose**: Inserts a key-value pair into the hash map. If *key* already exists, its old value is updated with *value*. If *key* does not exist, a new key-value pair is added to the hash map. The function may also automatically expand the size of the hash map when necessary (if the load factor exceeds the allowable threshold).
  - **Implementation Guidelines**: The main points are as follows:
    (a) Use the `hashCode` function to calculate the index of the `key`.
    (b) Retrieve the linked list from the `table` at the computed index.
    (c) Check whether the `key` already exists in the list.
      * If it does, update the existing `value` with the new `value` (remember to back up the old value to return it).
      * If not (i.e., the hash map does not contain `key`), create a new `Entry` for the `<key, value>` pair and insert it into the list. Increase the element count and call the `ensureLoadFactor` function to maintain the load factor.
    (d) **Exception**: None.

- **V get(K key)**
  - **Purpose**: Returns the value associated with the given key. If the key does not exist in the hash map, it throws the `KeyNotFound` exception, defined in **IMap.h**.
  - **Implementation Guidelines**: The main points are as follows:
    (a) Use the `hashCode` function to calculate the index of the `key` and retrieve the linked list at that index.
    (b) Search for the `key` in the list:
      * If found, return the corresponding `value` (stored in the same **Entry**).

  * If not found, throw an exception.

  (c) **Exception**: Throw the **KeyNotFound** exception if the `key` is not found.

- **V remove(K key, void (*deleteKeyInMap)(K) = 0)**
  - **Purpose**: Removes the `Entry` containing the `key` from the hash map, if found. This function also calls `deleteKeyInMap` to free the memory of `key` if `deleteKeyInMap` is not **nullptr**.
  - **Implementation Guidelines**: The main points are as follows:

    (a) Use the `hashCode` function to compute the index of the `key` and retrieve the list at that index.

    (b) Check whether the `key` exists in the list.

      * If found: (a) back up the corresponding `value` to return it; (b) free the `key` if `deleteKeyInMap` is not `NULL`; (c) remove the **Entry** (containing the `<key, value>` pair) from the list and free its memory. (Hint: use the **removeItem** function on the list, passing a pointer to the function `xMap<K,V>::deleteEntry` to both remove the **Entry** and free its memory.)
      * If not found: throw an exception.

    (c) **Exception**: Throws a **KeyNotFound** exception if the key does not exist in the map.

- **bool remove(K key, V value, void (*deleteKeyInMap)(K), void (*deleteValueInMap)(V))**
  - **Purpose**: Removes the `Entry` containing the `<key, value>` pair if it exists. This function compares both the key and value to determine which **Entry** to remove. It also frees the memory of `key` and `value` as required (i.e., when `deleteKeyInMap` or `deleteValueInMap` or both are not `NULL`). It returns `true` only if the `<key, value>` pair is found and removed; otherwise, it returns `false`.
  - **Implementation Guidelines**: The main idea is similar to the `remove(K key, void (*deleteKeyInMap)(K))` function. The difference is that both **key** and **value** must match.
  - **Exception**: None.

- **bool containsKey(K key)**
  - **Purpose**: Checks whether the hash map contains the given key.
  - **Implementation Guidelines**:

    (a) Use the `hashCode` function to compute the index of the `key` and retrieve the list at that index.

  (b) Search for the `key` in that list and return the corresponding result.
- **Exception**: None.

- **bool containsValue(V value)**
  - **Purpose**: Checks whether the hash map contains the given value.
  - **Implementation Guidelines**:
    (a) Search for the **value** in all the lists of the hash map and return the corresponding result.
  - **Exception**: None.
- **bool empty()**
  - **Purpose**: Checks if the map is empty.
  - **Exception**: None.
- **int size()**
  - **Purpose**: Returns the number of key-value pairs currently in the map.
  - **Exception**: None.
- **void clear()**
  - **Purpose**: Removes all key-value pairs from the hash map and resets it to its initial state.
  - **Implementation Guidelines**:
    (a) Call the **removeInternalData** function to free all memory.
    (b) Reinitialize the hash map as empty, with **capacity** equal to 10.
  - **Exception**: None.
- **string toString(string (*key2str)(K&) = 0, string (*value2str)(V&) = 0)**
  - **Purpose**: Returns a string representation of the hash map.
  - **Exception**: None.
- **DLinkedList<K> keys()**
  - **Purpose**: Returns a doubly linked list containing all the keys in the map.
  - **Exception**: None.
- **DLinkedList<V> values()**
  - **Purpose**: Returns a doubly linked list containing all the values in the hash map.
  - **Exception**: None.
- **DLinkedList<int> clashes()**
  - **Purpose**: Returns a doubly linked list containing the number of elements in the list at each index.

- **Implementation Guidelines**: Follow the main ideas described.
- **Exception**: None.

3 Heap Data Structure

3.1 Design Principles

Similar to other data structures, the implementation of the **Heap** in this library consists of two classes:

1. The class `IHeap` (see Figure 3), which is used to define the APIs for the Heap; and
2. The class `Heap` (a subclass of `IHeap`) that contains the concrete implementation for the Heap.

- **The `IHeap` Class, see Figure 3:**
  This class defines a set of methods (APIs) supported by the Heap. A few important notes about `IHeap` are as follows:
  - `IHeap` uses a **template** to parameterize the data type of the elements. Therefore, the Heap can store elements of any type `T`, provided that the type supports the necessary comparison operations to determine (a) equality and (b) ordering.
  - All APIs in `IHeap` are declared as **pure virtual methods**; meaning that any class inheriting from `IHeap` must **override** all these methods. Due to their virtual nature, the APIs will support dynamic binding (i.e., polymorphism).
- **The `Heap` Class:**
  This class is derived from `IHeap` and contains the concrete implementation of all the APIs defined in `IHeap`.

3.2 Explanation of the APIs

This section describes each **pure virtual method** of `IHeap`:

- `virtual ~IHeap() {};`
  - Virtual destructor, ensuring that the destructors of derived classes are called when a heap object is deleted via a base-class pointer.
- `virtual void push(T item) = 0;`
  - **Function:** Inserts an element `item` into the heap.
  - **Parameter:**

```cpp
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;
};
```

Figure 3: `IHeap<T>`: Abstract class defining APIs for Heap.

    * `T item` — the element to be added to the heap.

- `virtual T pop() = 0;`
  - **Function:** Removes and returns the **largest** or **smallest** element from the heap. In a **max-heap**, it returns the largest element; otherwise, in a **min-heap**, it returns the smallest element.
- `virtual const T peek() = 0;`
  - **Function:** Returns the largest/smallest element from the heap without removing it.
  - **Return Value:** The largest (or smallest) element in the heap.
- `virtual void remove(T item, void (*removeItemData)(T) = 0) = 0;`
  - **Function:** Removes the element `item` from the heap.
  - **Parameter:**
    * `T item` — the element to be removed.
    * `void (*removeItemData)(T)` — a function pointer (default is NULL) to process the data of the element to be removed. Typically, if the element type `T` is a pointer and the user needs the heap to free the memory, the user must supply a function pointer for deallocating the memory.
- `virtual bool contains(T item) = 0;`
  - **Function:** Checks whether the heap contains the element `item`.
  - **Parameter:** `T item` — the element to check.
  - **Return Value:** `true` if the element exists, otherwise `false`.

- `virtual int size() = 0;`
  - **Function:** Returns the number of elements currently in the heap.
  - **Return Value:** The number of elements in the heap.
- `virtual void heapify(T array[], int size) = 0;`
  - **Function:** Builds a heap from an array `array` with `size` elements.
  - **Parameters:**
    * `T array[]` — the array that contains the elements.
    * `int size` — the number of elements in the array.
- `virtual void clear() = 0;`
  - **Function:** Removes all elements from the heap and resets the heap to its initial state. **Note:** In its initial state, the heap is an array of size `capacity` (default capacity is 10) but contains no elements, i.e. it is **empty**.
- `virtual bool empty() = 0;`
  - **Function:** Checks whether the heap is empty.
  - **Return Value:** `true` if the heap is empty, otherwise `false`.
- `virtual string toString(string (*item2str)(T&) = 0) = 0;`
  - **Function:** Returns a string representation of the heap.
  - **Parameter:**
    * `string (*item2str)(T&)` — a function pointer to convert an element to a string.
  - **Return Value:** A string describing the heap.

3.3 Heap

The Heap has an important characteristic: physically, it is stored as an array of elements (i.e., the storage level of the heap); however, when working with the heap at the logical level, it should be viewed as a nearly complete or complete binary tree.

`Heap<T>` is a heap data structure where elements of type `T` are stored in a dynamic array whose size changes depending on the number of current elements. The operation of `Heap<T>` is based on maintaining the heap property, meaning that every parent node must have a value that is less than or equal to the values of its children in the case of a **min-heap** (or greater in the case of a **max-heap**).

To ensure this property, `Heap<T>` uses two main procedures: `reheapUp` and `reheapDown`, which rebalance the heap when elements are added (push) or removed (pop). When a new element is added, the `push` method inserts the element at the end of the array and then calls `reheapUp` to move the element to its correct position, ensuring the heap property. Similarly, when the root element is removed, the `pop` method moves the last element to the root position and calls `reheapDown` to maintain the heap property.

In addition to the methods inherited from `IHeap`, such as `push`, `pop`, `peek`, `contains`, and `clear`, `Heap<T>` also provides additional utility methods such as `heapify` to convert an array into a heap, `ensureCapacity` to automatically expand the array when necessary, and `free` to release user data if the type `T` is a pointer. These methods can be found in the file **Heap.h** in the directory /**include**/**heap**.

A former student from school B has implemented the Heap structure in the initial code. This code file is for reference only; students are required to modify the sample code to ensure it meets the descriptions below.

```
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);      // comparator function pointer
    void (*deleteUserData)(Heap<T>* pHeap);  // function pointer to delete
   user data
    // HIDDEN CODE
};

```

Figure 4: `Heap<T>`: Heap structure; declaration of member variables.

1. **Properties:** (See Figure 4)
   - `int capacity`: The current capacity of the heap, initially set to 10 by default.
   - `int count`: The number of elements currently stored in the heap.
   - `T* elements`: A dynamic array that stores the elements of the heap.
   - `int (*comparator)(T& lhs, T& rhs)`: A function pointer used to compare two elements of type `T` to determine their order in the heap.
     - If `comparator` is `NULL`: then (a) the type `T` must support the two comparison operators $>$ and $<$; and (b) `Heap<T>` behaves as a **min-heap**.

  - If `comparator` is not `NULL` and a **max-heap** is desired, then `comparator` returns:
    * $+1$ if `lhs` $<$ `rhs`
    * -1 if `lhs` $>$ `rhs`
    * 0 otherwise.
  - If `comparator` is not `NULL` and a **min-heap** is desired, then `comparator` returns:
    * -1 if `lhs` $<$ `rhs`
    * $+1$ if `lhs` $>$ `rhs`
    * 0 otherwise.
- `void (*deleteUserData)(Heap<T>* pHeap)`: A function pointer used to free user data when the heap is no longer used. In cases where `T` is a pointer and the user requires the heap to automatically free the memory for the elements, the user must supply a function for `deleteUserData` through the constructor. The user does not need to define a new function, simply pass the function `Heap<T>::free` into the constructor of `Heap<T>`.

2. **Constructors and Destructor:**

- `Heap(int (*comparator)(T&, T&)=0,`
  `void (*deleteUserData)(Heap<T>* )=0 ):`

  - **Purpose**: This constructor creates an empty heap, represented as an array of `capacity` (default value is 10) elements of type `T`, with `count` $= 0$. It initializes the member function pointers **comparator** and **deleteUserData** with the values provided.
  - **Parameters**: See the property descriptions above.
- `Heap(const Heap& heap)`: Copy constructor that duplicates the data from another heap object.
- `~Heap()`: Destructor that frees the memory and resources allocated for the heap.

3. **Methods:**

- **void push(T item)**
  - **Functionality**: This function inserts an element `item` into the heap and maintains the heap property (min or max).
  - **Complexity**: O(log(n))
  - **Exception**: None.
- **T pop()**

- **Functionality**: This function returns and removes the root element (the element at index 0 in `elements`, and also the largest or smallest element depending on the type of heap).
- **Complexity**: O(log(n))
- **Exception**: If the heap is empty, throw an exception `std::underflow_error("Calling to peek with the empty heap.")`.

- **T peek()**
  - **Functionality**: Returns the root element without removing it from the heap.
  - **Complexity**: O(1)
  - **Exception**: Throws an exception if the heap is empty: `throw std::underflow_error("Ca to peek with the empty heap.");`.
- **void remove(T item, void (*removeItemData)(T))**
  - **Functionality**: This method removes an element `item` from the heap. If the function `removeItemData` is provided, it will be called to free memory or perform any custom operations after the element is removed.
  - **Complexity**: O(log(n))
  - **Exception**: None.
- **bool contains(T item)**
  - **Functionality**: Checks whether the heap contains the element `value`.
  - **Exception**: None.
- **int size()**
  - **Functionality**: Returns the number of elements currently in the heap.
  - **Exception**: None.
- **void heapify(T array[], int size)**
  - **Functionality**: This method builds a heap from an array `array` of size `size`.
  - **Complexity**: O(n)
  - **Exception**: None.
- **bool empty()**
  - **Functionality**: Checks whether the heap is empty.
  - **Exception**: None.
- **void clear()**
  - **Functionality**: Removes all elements in the heap and resets it to the initial empty state.
  - **Exception**: None.

- **void heapsort(XArrayList<T>& arrayList)**
  - **Functionality**: Sorts the arrayList using the Heap structure. If the Heap already contains data, replace its current data with the data from arrayList. After each Heap Up, students should print the current elements in the Heap.

4 N-ary Huffman Tree and Inventory Compressor

This assignment requires students to implement an N-ary Huffman tree and apply it in compressing and decompressing inventory data. Students may consult theoretical resources and algorithms about Huffman trees from the following sources before continuing: [1], [2].

4.1 Theory of N-ary Huffman Trees

An N-ary Huffman tree is a generalization of the traditional binary Huffman tree, allowing each internal node to have up to n children instead of just 2. An important property of the N-ary Huffman tree is that if the number of initial characters is insufficient to build a complete tree in groups of n, we must add dummy characters (character $'\backslash 0'$) with frequency zero to ensure the number of leaf nodes L satisfies the condition:

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

These dummy characters do not carry any data significance and are only used to ensure that the encoding generated from the tree is optimal.

4.1.1 Steps to construct an N-ary Huffman Tree:

1. **Frequency count:** Determine the frequency of each character in the data to be encoded.
2. **Create leaf nodes:** For each real character, create a leaf node containing the character and its corresponding frequency.
3. **Add dummy characters (if needed):** If the initial number of leaf nodes does not satisfy $(L-1) \mod (n-1) = 0$, add dummy characters with frequency 0 until the condition is met.
4. **Build the tree:** Repeat the following process until only one node remains in the heap:
   (a) Remove the n nodes with the smallest frequencies from the heap.
   (b) Compute the total frequency of these nodes.
   (c) Create a new internal node with the computed frequency and assign the nodes as its children.
   (d) Push the new internal node back into the heap.
5. **Determine root node:** The only remaining node in the heap is the root of the Huffman tree.

Once the tree is built, we can assign codes to the characters by traversing from the root to the leaf nodes. In practice, these codes are bit strings to minimize text storage size. However, for simplicity in this classroom assignment, we represent codes as a sequence of characters from '0' to 'f' (base 16). For example, if the Huffman tree has 3 branches, we assign codes as follows:

- First child node: code '0'
- Second child node: code '1'
- Third child node: code '2'

The Huffman code for each character is formed by concatenating the numeric codes of the nodes along the path from the root to the corresponding character node. Refer to the illustrative example for better understanding.

4.1.2 Illustrative Example

Assume we have 4 characters with the following frequencies:

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

With a 3-ary Huffman tree (each internal node has at most 3 children), we need the number of leaf nodes such that:

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

With $L = 4$, we have $(4-1) \mod 2 = 3 \mod 2 = 1 \neq 0$. Hence, we need to add $d = (2 - (3 \mod 2)) = 2 - 1 = 1$ dummy character with frequency 0. After adding, the number of leaf nodes becomes $L = 4 + 1 = 5$, and $(5-1) \mod 2 = 4 \mod 2 = 0$.

Initial list of leaf nodes:

- \0 (dummy character): 0
- A: 5
- B: 9
- C: 12
- D: 13

Build the 3-ary Huffman tree as follows:

1. **Step 1:** Remove the 3 nodes with the lowest frequencies:
   - Dummy: 0
   - A: 5
   - B: 9

   Total frequency $= 0 + 5 + 9 = 14$.

   Create a new internal node with frequency 14 and assign `Dummy`, A, and B as its children.
2. **Step 2:** After removing the 3 processed nodes, we are left with:
   - Internal node with frequency 14
   - C: 12
   - D: 13

   Take these 3 nodes, total frequency $= 14 + 12 + 13 = 39$.

   Create a new internal node with frequency 39 and assign the nodes with frequencies 14, 12, and 13 as its children.

The internal node with frequency 39 is the only one left in the heap and thus becomes the root of the Huffman tree. Figure 5 shows the 3-ary Huffman tree constructed for the example above.

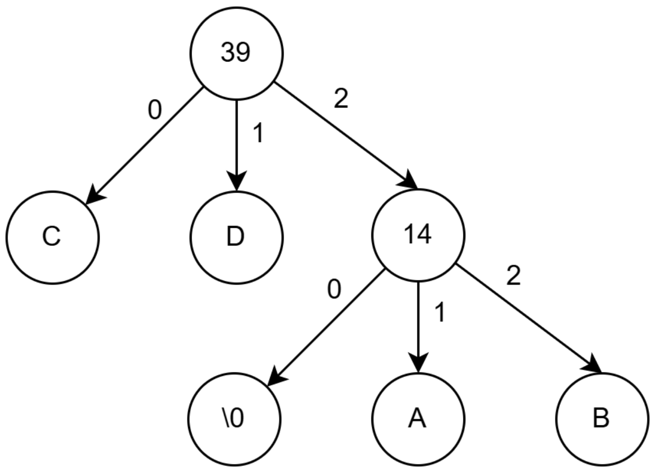

Figure 5: Constructed 3-ary Huffman Tree

Using the 3-ary Huffman tree, we get the encoded result for each character as follows:

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

4.2 HuffmanTree

`HuffmanTree<treeOrder>` is a template class that manages the N-ary Huffman tree used to encode and decode characters based on their frequencies.

```cpp
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;
};
```

Attributes:

- `HuffmanNode* root`: Pointer to the root node of the Huffman tree.
- `struct HuffmanNode`: Inner structure representing a node in the Huffman tree.
  - `char symbol`: Stores the character (for leaf nodes).
  - `int freq`: Frequency of the character or the sum frequency of its child nodes (for internal nodes).
  - `XArrayList<HuffmanNode*> children`: List containing child nodes; for internal nodes, this list contains merged nodes from the heap.

Methods:

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

- **Functionality:** Builds the N-ary Huffman tree from a list of character-frequency pairs.
- **Implementation Requirements:**
  (a) Create a heap from the given list.
  (b) While the heap has more than one node, select up to `treeOrder` nodes with the lowest frequencies.
  (c) Compute the total frequency and group selected nodes into a list.
  (d) Create a new internal node with the computed frequency and list of children.
  (e) Push the new internal node into the heap.
  (f) After the loop, the last remaining node becomes the `root` of the tree.
- **Note**: To ensure consistency, when two nodes in the heap have the same frequency, the one added earlier has higher priority.

2. **void generateCodes(xMap<char, std::string>& table)**
   - **Functionality:** Generates Huffman codes for each character in the tree. The accumulated codes are stored in `table`, with the character as the key.
3. **std::string decode(const std::string& huffmanCode)**
   - **Functionality:** Decodes a Huffman code string back to the original text.

4.3 InventoryCompressor<treeOrder>

`InventoryCompressor<treeOrder>` is a template class that uses the Huffman tree to compress and decompress inventory data. The number of branches `treeOrder` is used consistently during Huffman tree construction.

```
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);

private:
    xMap<char, std::string> huffmanTable;
    InventoryManager* invManager;
    HuffmanTree<treeOrder>* tree;
};
```

Attributes:

- `InventoryManager* invManager`: Pointer to the `InventoryManager` object providing product names and inventory attributes.
- `xMap<char, std::string> huffmanTable`: Hash table containing Huffman codes for each character.
- `HuffmanTree<treeOrder>* tree`: Pointer to the constructed Huffman tree used for encoding and decoding.

Constructor and Destructor:

- `InventoryCompressor(InventoryManager* manager)`: Initializes the compressor with the `InventoryManager` object.
- ∼`InventoryCompressor()`: Destroys the object and frees the memory allocated for the Huffman tree.

Methods:

1. **void buildHuffman()**
   - **Functionality:** Builds the Huffman tree and generates Huffman codes based on character frequencies in the inventory data.
   - **Implementation Requirements:**
     (a) Traverse all products to build a frequency table for each character in the string representation of a product using `productToString`.
     (b) Create the `huffmanTable` by building a Huffman tree using the frequency table.
2. **std::string productToString(const List1D<InventoryAttribute>& attributes, const std::string& name)**
   - **Functionality:** Converts product data to a string with the format: *"product name:(attribute1:va (attribute2:value2), ..."*.
3. **std::string encodeHuffman(const List1D<InventoryAttribute>& attributes, const std::string& name)**

- **Functionality:** Encodes the product string into a Huffman code.
- **Implementation Requirements:** Convert product data to a string using `productToString`, then generate the code using `huffmanTable`.

4. **std::string decodeHuffman(const std::string& huffmanCode, List1D<InventoryAttribute attributesOutput, std::string& nameOutput)**

   - **Functionality:** Decodes the Huffman code to recover the original product data.
   - **Implementation Requirements:**
     (a) Decode the string using the Huffman tree.
     (b) Parse the result string to extract the product name and attribute list. The expected format is: *"product name: (attribute1: value1), (attribute2: value2), ..."*.
     (c) The method returns the decoded string. At the same time, the product name and attribute list are assigned to `nameOutput` and `attributesOutput`, respectively.

5 Requirements

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

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

5.1 Compilation

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

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

5.2 Submission

Detailed submission instructions will be provided later.

6 Other Regulations

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

7 Tracking Changes Across Versions

(v1.1)

- Updated the description for the Heap data structure.
- Updated the initial code for the Heap data structure.
- Updated the grading criteria.

———————END———————