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

Data structures and Algorithms - CO2003

Assignment 2

Hash-Map, Heap and Artificial Neural Networks

Ho Chi Minh City, 09/2024

SPECIFICATION OF THE MAJOR ASSIGNMENT

Version 1.0

1. Version 1.0: October 10, 2024

1 Introduction

1.1 Content

The second major assignment of the course *Data Structures and Algorithms* includes the following two tasks:

1. Task 1 (also referred to as **TASK-1**, accounting for 40% of the total score): This task requires students to develop the following two data structures:
   (a) **HashMap Structure**. The file to be implemented for this part is **./include/hash/xMap.h**. Students can find the implementation guide for this task in Section 2.
   (b) **Heap Structure**. The file to be implemented for this part is **./include/heap/Heap.h**. Students can find the implementation guide for this task in Section 3.
2. Task 2 (also referred to as **TASK-2**, accounting for 60% of the total score): This task requires students to use the learned data structures to develop a multi-layer feedforward neural network. Students can find the implementation guide for this task in Section 4.

1.2 Project Structure

The source code provided with this document is organized as a project structured as shown in Figures 1 and 2. Note that students must download and extract the files to view the project structure.

At the root directory of the project, students will find the following directories and files:

- **./include**: This directory contains all the header files (*.h) related to the project. Some notable subdirectories are:
  - **./include/list**: Contains files related to the List data structure, developed in Major Assignment 1.
  - **./include/loader**: Contains files related to data structures such as `Dataset`, `TensorDataset`, and `DataLoader`, developed in Major Assignment 1.
  - **./include/hash**: Contains files related to the HashMap data structure, part of **TASK-1** in Major Assignment 2.
  - **./include/heap**: Contains files related to the Heap data structure, part of **TASK-1** in Major Assignment 2.
  - **./include/ann**: Contains files related to **TASK-2** in Major Assignment 2.

  - **./include/tensor**: Contains files related to the **xtensor** library for multidimensional arrays.
  - **./include/sformat**: Contains files related to the **fmt** library for string formatting.
  - **./include/graph**: Contains files related to the Graph data structure, part of **TASK-1** in Major Assignment 3.
  - **./include/util**: Contains utility classes and functions used throughout the project.
  - **./include/dsaheader.h**: A header file that includes all the data structures developed in the course and provides shorthand references to them.

- **./src**: This directory contains all the "*.cpp" files related to the project, with the following subdirectories and files:

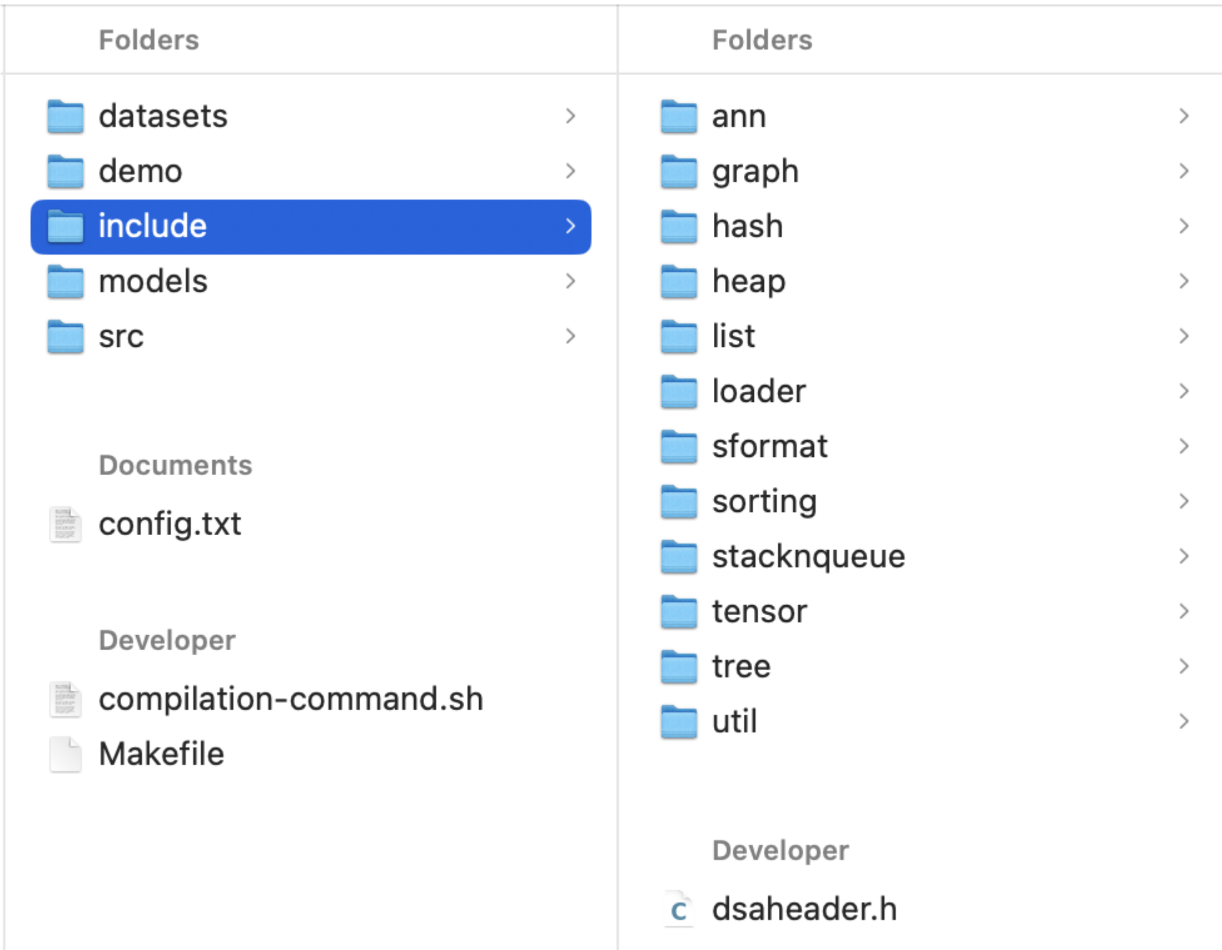
  - **./src/ann**: Contains the implementation of the classes for **TASK-2** in Major Assignment 2.
  - **./src/tensor**: Contains extended functions for the **xtensor** library, used in **TASK-2** of Major Assignment 2.
  - **./src/program.cpp**: This file contains the **main** function of the project.

- **./demo**: This directory contains "*.h" files demonstrating how to use the data structures and classes in the project. *Students are encouraged to review these examples to learn how to utilize the data structures and libraries.*
- **./datasets**: This directory contains datasets used in **TASK-2** of the major assignment.
- **./models**: This directory stores models generated in **TASK-2**. *Students should save their models in this directory.*
- **./config.txt**: This file contains configuration variables used in **TASK-2**.
- **./Makefile**: A file that supports compiling the project using the **make** command. To compile the project, students can run **make** in the **Terminal**.
- **./compilation-command.sh**: A script to compile the project using the **g++** command. To compile, students can run **./compilation-command.sh** in the **Terminal**.

1.3 Implementation Method

- **Important Note**:

  1. **TASK-1** of Major Assignment 2 requires (depends on) the doubly linked list data structure, specifically the `DLinkedList` from **TASK-1** of Assignment 1. Note that in TASK-2 of Assignment 2, we need to use the Backward-Iterator of `DLinkedList`. Therefore, students are also required to add the Backward-Iterator to `DLinkedList`



Hình 1: Project structure for Major Assignment 2 and the contents of the **./include** directory

before using it in TASK-2. The Backward-Iterator must support the following operators:

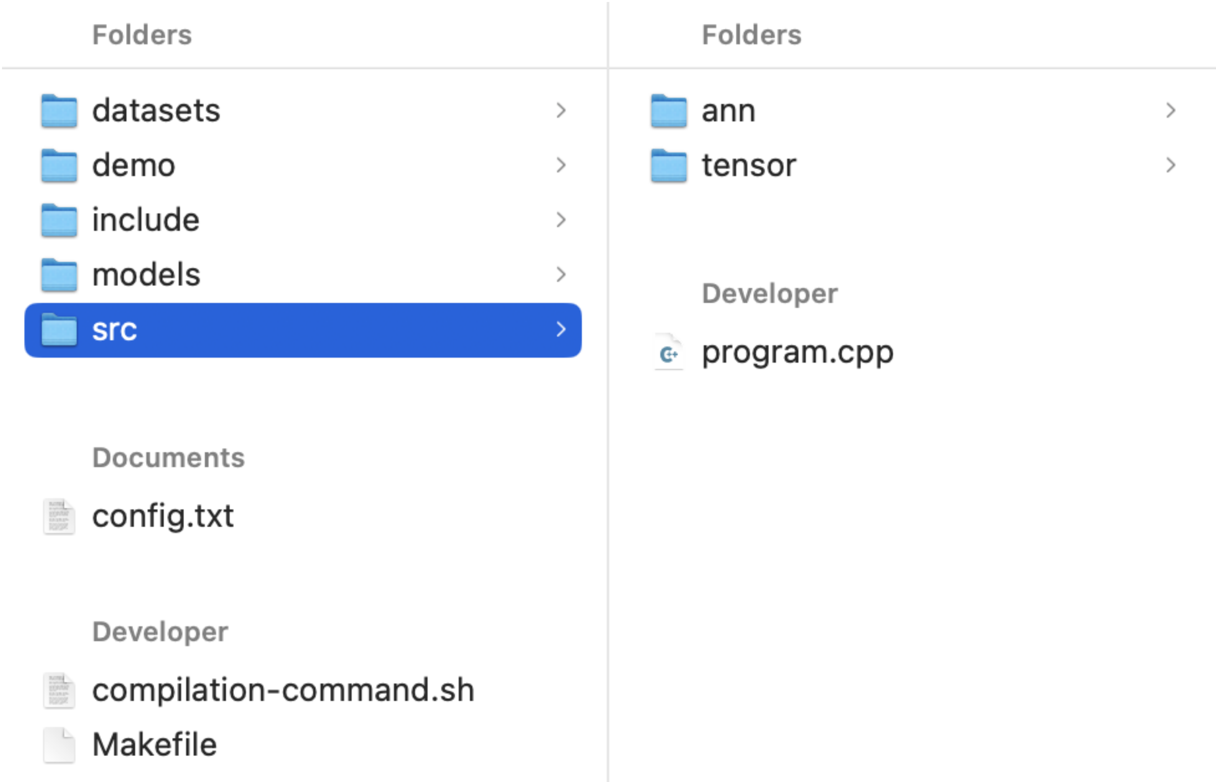
(a) Inequality comparison operator (`!=`).

(b) `++` operator: advances the iterator backward from the end to the beginning of the list.

(c) `*` operator: used to extract the current element.

2. **TASK-2** of Assignment 2 depends on `DLinkedList` and the classes in the two files **./include/loader/dataset.h** and **./include/loader/dataloader.h**. Additionally, the **Dataset**, **TensorDataset**, and **DataLoader** classes contain some new methods. Therefore, students need to update the implementation of these classes according to the new version (only minor updates are required).

- Method for implementing **TASK-1**:
  - Students **MUST complete TASK-1** first and independently from **TASK-2**, by moving the source code of **TASK-2** to a directory **outside the project**. The source



Hình 2: Project structure for Major Assignment 2 and the contents of the **./src** directory

code for **TASK-2** is located in:

* **./include/ann**
* **./include/tensor/tensor_lib.h** (move only this file).
* **./src/ann**
* **./src/tensor**

– Implement the functions in **./include/hash/xMap.h** and **./include/heap/Heap.h**: fill in the code at the places marked with **YOUR CODE IS HERE**.

– Compile and run tests. It is recommended to try the examples in **./demo/hash** and **./demo/heap**.

- Method for implementing **TASK-2**:
  - Students **CAN ONLY** start working on **TASK-2** after successfully completing **TASK-1** and the required structures from Assignment 1.
  - Move the provided code for **TASK-2** into the appropriate directories and proceed with the implementation.
  - Compile and run tests.

1.4 How to Compile the Project

Students can compile the project using one of the following three methods:

1. Using the **./compilation-command.sh** script:
   - Open a **Terminal** window and navigate to the root directory of the project using the **cd** command.
   - Execute the following command: **./compilation-command.sh**
   - If errors occur, resolve them and recompile the project.
2. Using the **Makefile**:
   - Open a **Terminal** window and navigate to the root directory of the project using the **cd** command.
   - Run one of the following commands:
     - **make** or
     - **make clean** followed by **make**
   - If errors occur, resolve them and recompile the project.
3. Using an Integrated Development Environment (IDE) such as **NetBeans**, **VSCode**, etc.:
   - Create a new project in the IDE.
   - Add the project files from the Major Assignment to the IDE project.
   - Configure the project to compile using **g++** with **C++17** standard.
   - Compile the project using the IDE's menu options.

2 TASK-1: Hash Table Structure

2.1 Design Principles

Similar to other data structures, the implementation of the hash table in this library involves two classes: (a) The `IMap` class (see Figure 3), which defines the APIs for the hash table; and (b) The `xMap` class (a subclass of `IMap`), which provides the concrete implementation of the hash table.

- `IMap` class (see Figure 3): This class defines a set of methods (APIs) supported by the hash table.

   - `IMap` uses **templates** to parameterize the data types of elements, specifically the types of **key** and **value**. Thus, the hash table can work with any key-value types.
   - All APIs in `IMap` are defined as **pure virtual methods**, meaning that any class inheriting from `IMap` must **override** all these methods. Due to their **virtual** nature, the APIs support dynamic binding (polymorphism).

- `xMap` class: This class inherits from `IMap`. It provides concrete implementations for all APIs defined in `IMap` by using a **doubly linked list (DLinkedList)** to store all `<key, value>` pairs that collide at each address in the hash table. In other words, the hash table in this library is a table of doubly linked lists.

```
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;
};
```

Hình 3: `IMap<T>`: Abstract class defining APIs for the hash table.

2.2 Explanation of APIs

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

- `virtual ~IMap() {};`
   - Virtual destructor to ensure that the destructor of derived classes is invoked when an object is deleted via a pointer to `IMap`.
- `virtual V put(K key, V value) = 0;`
   - **Parameters**:

        * `K key` — The key to add or update.
        * `V value` — The value to insert or replace.
    - **Return Value**:
        * Returns the old value if the key already exists; otherwise, returns the new value.
    - **Purpose**:
        * Adds a `<key, value>` pair to the hash table. If the key exists, the old value is replaced, and the old value is returned.

- `virtual V& get(K key) = 0;`
    - **Purpose**: Retrieves the value associated with the given key.
    - **Parameters**: `K key` — The key to retrieve.
    - **Return Value**: A reference to the value associated with the key.
    - **Exception**: Throws `KeyNotFound` if the key does not exist.

- `virtual V remove(K key, void (*deleteKeyInMap)(K) = 0) = 0;`
    - **Purpose**: Removes the key from the map and returns the associated value.
    - **Parameters**:
        * `K key` — The key to remove.
        * `void (*deleteKeyInMap)(K)` — A function pointer to delete the key (useful if the key is a pointer).
    - **Return Value**: The value associated with the key.
    - **Exception**: Throws `KeyNotFound` if the key does not exist.

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

- `virtual bool containsKey(K key) = 0;`
    - **Purpose**: Checks if the key exists in the map.
    - **Parameters**: `K key` — The key to check.

  - **Return Value**: `true` if the key exists; otherwise, `false`.

- `virtual bool containsValue(V value) = 0;`
  - **Purpose**: Checks if the value exists in the map at any address.
  - **Parameters**: `V value` — The value to check.
  - **Return Value**: `true` if the value exists; otherwise, `false`.

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

- `virtual int size() = 0;`
  - **Purpose**: Returns the number of key-value pairs in the map.
  - **Return Value**: The size of the map. A return value of `0` indicates an empty map.

- `virtual void clear() = 0;`
  - **Purpose**: Removes all key-value pairs from the map, resetting it to an empty state.

- `virtual string toString(string (*key2str)(K&) = 0, string (*value2str)(V&) = 0) = 0;`
  - **Purpose**: Returns a string representation of the map.
  - **Parameters**:
    * `string (*key2str)(K&)` — Function pointer to convert a key to a string.
    * `string (*value2str)(V&)` — Function pointer to convert a value to a string.
  - **Return Value**: A string representation of the map.

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

- `virtual DLinkedList<V> values() = 0;`
  - **Purpose**: Returns a list of all values in the map.
  - **Return Value**: `DLinkedList<V>` containing the values.

- `virtual DLinkedList<int> clashes() = 0;`
  - **Purpose**: Returns a list of collision counts at each address in the map.
  - **Return Value**: `DLinkedList<int>` containing the collision counts.

2.3 Hash Map

`xMap<K, V>` is an implementation of a hash map, where elements are stored as key-value pairs (`<K, V>`) in a predefined size array, called the *table*. The operation principle of `xMap<K, V>` relies on a hash function to determine the storage location of elements within the array.

Load Factor and the `rehash` Process: To ensure performance, `xMap<K, V>` needs to effectively manage the size of the array and maintain a proper *load factor*—the ratio of the current number of elements to the size of the array. If this ratio exceeds the specified `loadFactor`, the hash map will automatically perform a `rehash` (increase the array size) and redistribute the elements to minimize collisions.

Methods and Utility Extensions: In addition to inheriting methods from the `IMap` interface such as `put`, `get`, `remove`, `containsKey`, and `clear`, the `xMap<K, V>` class also provides additional utility methods like:

- `rehash`: Expands the size of the hash map.
- `ensureLoadFactor`: Ensures that the load ratio does not exceed the allowed threshold.

The detailed source code of the `xMap<K, V>` class can be found in the file `xMap.h`, located in the **/include/hash** directory.

1. **Attributes:** See Figure 4.
    - `int capacity`: The current capacity of the hash table.
    - `int count`: The number of elements currently in the hash table.
    - `DLinkedList<Entry* >* table`: The hash table is a (dynamic) array of elements of type doubly linked list (DLinkedList); each element of the list is a **pointer** to `Entry`, where `Entry` is the class containing the `<key, value>` pair. One can visualize the hash table as a series of list-like objects, and the list will contain all **collisions** at a specific address.
    - `float loadFactor`: The load factor of the hash table, indicating the level of space utilization before resizing is necessary. At any point, the number of elements in the table (`count`) **must not exceed** `loadFactor` $\times$ `capacity`. For example, if `capacity` $=10$ and `loadFactor` $=0.75$, then the maximum number of elements that can be stored is `int`$(0.75 \times 10) = 7$; when inserting the eighth element, `ensureLoadFactor` needs to be called to ensure the load factor.
    - The attributes are **function pointers**, initialized through the constructor. **Note**: Only `hashCode` must always be passed in through the constructor and must be

```
template<class K, class V>
class xMap: public IMap<K,V> {
public:
    class Entry; // Forward declaration

protected:
    DLinkedList<Entry*>* table;  // Array of doubly linked lists for
    collision handling
    int capacity;                // Size of the table (number of buckets)
    int count;                   // Number of entries currently stored
    float loadFactor;            // Maximum allowed load factor

    int (*hashCode)(K&, int);    // Hash function: takes key and table size
    bool (*keyEqual)(K&, K&);    // Function to compare two keys
    bool (*valueEqual)(V&, V&);  // Function to compare two values
    void (*deleteKeys)(xMap<K,V>*);   // Function to delete all keys
    void (*deleteValues)(xMap<K,V>*); // Function to delete all values

    // Other methods and helpers are hidden for brevity

public:
    // Definition of Entry class
    class Entry {
    private:
        K key;
        V value;
        friend class xMap<K, V>;

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

Hình 4: `xMap<T>`: Structure of the hash map and main attributes

different from `NULL`; other function pointers can be `NULL` or not, depending on the library user's needs.

- `int (*hashCode)(K&,int)`: This function takes a **key** (passed by reference) and the size of the hash table (an integer); it calculates the address of the key in the hash table of the given size. **Note**: If the size of the hash table passed to the function is m, the return value of the `hashCode` function can only lie in the range $[0, 1, \ldots, m-1]$; to ensure this, `hashCode` must use the modulo operator %. See also the functions `intKeyHash` and `stringKeyHash` in the accompanying source code.
- `bool (*keyEqual)(K&,K&)`: If the data type of the key (`K`) does not support

equality comparison between two keys using the == operator, the user must provide a pointer to a function that can compare the equality of two keys. This function takes two keys of type `K` (passed by reference) and returns `true` if the two keys are equal, and `false` if they are not equal. **Note:** Within the **xMap** class, when comparing two keys, use the `keyEQ` method.

  - `bool (*valueEqual)(V&,V&)`: If the data type of the value (`V`) does not support equality comparison between two values using the == operator, the user must provide a pointer to a function that can compare the equality of two values. This function takes two values of type `V` (passed by reference) and returns `true` if the two values are equal, and `false` if they are not equal. **Note:** Within the **xMap** class, when comparing two values, use the `valueEQ` method.
  - `void (*deleteKeys)(xMap<K,V>* pMap)`: If `K` is a pointer and the library user needs **xMap** to actively free memory for the keys, the user **MUST** provide a pointer to a function via `deleteKeys`. The user does not need to define a new function; they just need to pass `xMap<K,V>::freeKey` to `deleteKeys`. See the source code for `xMap<K,V>::freeKey` for details.
    This function pointer is used to delete keys when necessary.
  - `void (*deleteValues)(xMap<K,V>* pMap)`: If `V` is a pointer and the library user needs **xMap** to actively free memory for the values, the user **MUST** provide a pointer to a function via `deleteValues`. The user does not need to define a new function; they just need to pass `xMap<K,V>::freeValue` to `deleteValues`. See the source code for `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 table with `capacity` linked lists in the table. The default value of `capacity` is 10. This function also initializes the member variables, which are the function pointers mentioned above, with the provided values.
     ```

  - **Parameters**: See the attributes description.

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

3. **Methods:**

- **V put(K key, V value)**
  - **Functionality**: This function inserts a `<key, value>` pair into the hash table. If the *key* already exists, the old value will be updated with the *value*. If the *key* does not exist, a new `<key, value>` pair will be added to the hash table. The function can also automatically resize the hash table when necessary (when the load factor exceeds the allowed threshold).
  - **Implementation guidance**: The main points are as follows.
    (a) Use the hash function `hashCode` to compute the address of the `key`.
    (b) Retrieve the list from `table` using the above address.
    (c) Check if the `key` is already present in the list.
      * If it is, update the old `value` with the new `value`. Remember to backup the old `value` to return.
      * If not (the hash table does not contain the `key`): create an `Entry` for the `<key, value>` pair and insert it into the list. Increment the count of elements in the table and call the `ensureLoadFactor` function to ensure the load factor is maintained.
    (d) **Exception**: None.

- **V get(K key)**
  - **Functionality**: This function returns the value associated with the provided `key`. If the key does not exist in the hash table, it throws an exception `KeyNotFound`, which is predefined in the **IMap.h** file.
  - **Implementation guidance**: The main points are as follows.
    (a) Use the hash function `hashCode` to compute the address of the `key` and retrieve the list at the found address.
    (b) Check if the `key` is contained in the list.
      * If found, return the corresponding `value` (contained within the same **Entry**).
      * If not found, throw an exception.

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

- **V remove(K key, void (*deleteKeyInMap)(K) = 0)**
  - **Functionality**: Remove the **Entry** containing the `key` from the hash table when the `key` is found. This function also calls `deleteKeyInMap` to free the memory for `key` when `deleteKeyInMap` is not **nullptr**.
  - **Implementation guidance**: The main points are as follows.
    (a) Use the hash function `hashCode` to compute the address of the `key` and retrieve the list at the found address.
    (b) Check if the `key` is contained in the list.
      * If found: (a) backup the `value` to return; (b) free the `key` if `deleteKeyInMap` is not `NULL`; (c) remove the **Entry** (containing the `<key, value>` pair) from the list and free the memory of the **Entry**. Hint: use the **removeItem** function on the list to both remove the **Entry** and free the memory of the **Entry** by passing a pointer to the `xMap<K,V>::deleteEntry` function into the **removeItem** function.
      * If not found: throw an exception.
    (c) **Exception**: Throw a **KeyNotFound** exception if the key does not exist in the table.
- **bool remove(K key, V value, void (*deleteKeyInMap)(K), void (*deleteValueInMap)(V))**
  - **Functionality**: Remove the **Entry** containing the `<key, value>` pair from the hash table. This function matches both the **key** and **value** to determine which **Entry** to remove from the table. This function also frees memory for `key` and `value` when requested; that is, when `deleteKeyInMap` or `deleteValueInMap` or both are not `NULL`. This function returns `true` only if the `<key, value>` pair is found and `false` if the `<key, value>` pair is not found.
  - **Implementation guidance**: The main points are as follows.
    (a) Similar to the `remove(K key, void (*deleteKeyInMap)(K))` function. The difference is that it needs to match both the **key** and **value**.
  - **Exception**: None.
- **bool containsKey(K key)**
  - **Functionality**: Check if the hash table contains the *key*.
  - **Implementation guidance**: The main points are as follows.
    (a) Use the hash function `hashCode` to compute the address of the `key` and retrieve the list at the found address.

(b) Search for the **key** in the list and return the corresponding result.

- **Exception**: None.

- **bool containsValue(V value)**
  - **Functionality**: Check if the hash table contains the *value*.
  - **Implementation guidance**: The main points are as follows.
    (a) Search for the **value** in all lists in the hash table and return the corresponding result.
  - **Exception**: None.
- **bool empty()**
  - **Functionality**: Check if the hash table is empty.
  - **Exception**: None.
- **int size()**
  - **Functionality**: Return the number of elements currently in the hash table.
  - **Exception**: None.
- **void clear()**
  - **Functionality**: Remove all elements in the hash table and reset the hash table to its initial state.
  - **Implementation guidance**: The main points are as follows.
    (a) Call the **removeInternalData** function to free memory.
    (b) Reinitialize the empty hash table with a **capacity** of 10.
  - **Exception**: None.
- **string toString(string (*key2str)(K&) = 0, string (*value2str)(V&) = 0)**
  - **Functionality**: Return a string representation of the elements in the hash table.
  - **Exception**: None.
- **DLinkedList<K> keys()**
  - **Functionality**: Return a linked list containing all the keys in the hash table.
  - **Exception**: None.
- **DLinkedList<V> values()**
  - **Functionality**: Return a linked list containing all the values in the hash table.
  - **Exception**: None.
- **DLinkedList<int> clashes()**
  - **Functionality**: Return a linked list containing the number of elements in the list at each address.

- **Implementation guidance**: The main points are as follows.
- **Exception**: None.

3 TASK-1: Data Structure Heap

3.1 Design Principles

Similar to other data structures, the implementation of **Heap** in this library consists of two classes: (a) the `IHeap` class (see Figure 5), which is used to define the APIs for the **Heap** structure; and (b) the `Heap` class (which is a subclass of `IHeap`) that contains the specific implementation for the **Heap**.

- The `IHeap` class, see Figure 5: this class defines a set of methods (APIs) supported by the Heap. Some notes about `IHeap` are as follows:
  - `IHeap` uses **templates** to parameterize the data type of the elements. Therefore, **Heap** can contain elements of any type `T`, as long as the data type supports comparison to determine: (a) equality and (b) order.
  - All APIs in `IHeap` are defined as **pure virtual methods**; meaning that classes inheriting from `IHeap` must **override** all of these methods. Due to the **virtual** nature, these APIs will support dynamic binding (polymorphism).
- The `Heap` class: inherits from `IHeap`. This class contains the specific implementation for all APIs defined in `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 5: `IHeap<T>`: Abstract class defining APIs for Heap.

3.2 Explanation of APIs

This section will describe each **pure virtual method** of `IHeap`:

- `virtual ~IHeap() {};`
  - The virtual destructor ensures that the destructors of subclasses are called when the heap object is deleted through a base class pointer.
- `virtual void push(T item) = 0;`
  - Adds an element `item` to the heap.
  - **Parameters**:
    * `T item` — the element to be added to the heap.
- `virtual T pop() = 0;`
  - Removes and returns the **largest** or **smallest** element from the heap. If it is a **max-heap**, it returns the largest element; otherwise, it returns the smallest element.
- `virtual const T peek() = 0;`
  - 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;`
  - Removes the element `item` from the heap.
  - **Parameters**:
    * `T item` — the element to be removed.
    * `void (*removeItemData)(T)` — a function pointer (default is NULL) to handle the data of the element to be removed. Typically, if the data type `T` is a pointer and the user needs to free the memory of the element, they should pass in the address of the function to free the memory for the element.
- `virtual bool contains(T item) = 0;`
  - Checks whether the element `item` exists in the heap.
  - **Parameters**: `T item` — the element to check.
  - **Return value**: `true` if the element exists, otherwise `false`.
- `virtual int size() = 0;`
  - 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;`
  - Builds a heap from an array `array` with size `size`.
  - **Parameters**:
    * `T array[]` — the array containing the elements.

    * `int size` — the number of elements in the array.

- `virtual void clear() = 0;`
  - Removes all elements in the heap and resets the heap to its initial state. Note: in the initial state, the heap is an array of size `capacity` containing elements of type `T`; the default of `capacity` is 10. In the initial state, the heap contains no elements, meaning it is **empty**.
- `virtual bool empty() = 0;`
  - Checks whether the heap is empty or not.
  - **Return value**: `true` if the heap is empty, otherwise `false`.
- `virtual string toString(string (*item2str)(T&) = 0) = 0;`
  - Returns a string representation of the heap.
  - **Parameters**:
    * `string (*item2str)(T&)` — a function pointer to convert the element to a string.
  - **Return value**: A string describing the heap.

3.3 Heap

A heap has an important property; it is an array of elements when viewed at the **physical** level, which means 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 data structure of type heap, where elements of type `T` are stored in a dynamic array that changes size according to the current number of elements. The operation principle of `Heap<T>` is based on maintaining the property of a heap, meaning every parent node must have a value less than or equal to its child nodes 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 processes: `reheapUp` and `reheapDown`, which help balance the heap when adding (push) or removing (pop) elements. When a new element is added, the `push` method adds the element to the end of the array and performs `reheapUp` to move this element up to the correct position. Similarly, when removing the root element, the `pop` method moves the last element to the top and performs `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 supports other utility methods like `heapify` to turn an array into a heap, `ensureCapacity` to automatically resize the array when necessary, and `free` to free user data if the type `T` is a pointer. These methods can be found in the **Heap.h** file, located in the **/include/heap** directory.

```
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 6: `Heap<T>`: Heap structure; member variable declaration.

1. **Attributes:** See Figure 6.
   - `int capacity`: The current capacity of the heap, initialized by default to 10.
   - `int count`: The number of elements currently in the heap.
   - `T* elements`: A dynamic array that stores the elements of the heap.
   - `int (*comparator)(T& lhs, T& rhs)`: A function pointer that compares two elements of type `T` to determine their order in the heap.
     - If `comparator` is `NULL`: then, (a) type `T` must support two comparison operators: $>$ and $<$; and (b) `Heap<T>` is a **min-heap**.
     - If `comparator` is not `NULL`; to make `Heap<T>` a **max-heap**, the `comparator` function should return three values according to the following rules:
       * $+1$: if `lhs` $<$ `rhs`
       * -1: if `lhs` $>$ `rhs`
       * 0: otherwise.
     - If `comparator` is not `NULL`; to make `Heap<T>` a **min-heap**, the `comparator` function should return three values according to the following rules:
       * -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 in use. If the type `T` is a pointer and the user requires the heap to automatically free the memory of its elements, the user must pass a function to `deleteUserData` through the constructor. The user does not need to define a new function; they can simply pass the function `Heap<T>::free` to the constructor of `Heap<T>`.

2. **Constructor and Destructor:**

- `Heap(int (*comparator)(T& , T&)=0,`
`void (*deleteUserData)(Heap<T>*)=0 )`: Initializes an empty heap. An empty heap is an array of `capacity` (10) elements of type `T`, but with `count` = 0. Initializes the member variables **comparator** and **deleteUserData** with the parameters received from the constructor. Note that depending on the **comparator**, the heap can be either a **min-heap** or a **max-heap**. If no **comparator** is provided, the Heap defaults to a **min-heap**. See the explanation for the member variables that are pointers for a better understanding of the **comparator** and **deleteUserData** parameters.
- `Heap(const Heap& heap)`: Copy constructor that copies 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 `item` into the heap and maintains the heap property (min or max).
  - **Implementation Guide**: The main points are as follows.
    (a) Call the **ensureCapacity** function to ensure that the dynamic array **elements** can hold `(count + 1)` elements.
    (b) Add the element to the end of the `elements` array.
    (c) Perform the `reheapUp` process to move the element to its correct position, ensuring the heap property.
    (d) Increase the value of the `count` variable.
  - **Exceptions**: None.
- **T pop()**
  - **Functionality**: This function returns and removes the root element (the element at index 0 on `elements`, which is also the largest or smallest element depending

on the type of heap).

  - **Implementation Guide**: The main points are as follows.
    (a) Move the last element in the `elements` array to position 0, backing up the element at 0 to return it.
    (b) Perform the `reheapDown` process to maintain the heap property.
    (c) Update the `count` variable.
  - **Exceptions**: If the heap is empty, throw the exception `std::underflow_error(‘‘Calling to peek with the empty heap.‘‘)`.

- **T peek()**
  - **Functionality**: Returns the root element without removing it from the heap.
  - **Exceptions**: 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 `item` from the heap. If a `removeItemData` function is provided, it will be called to free memory or perform custom operations after the element is removed.
  - **Implementation Guide**: The main points are as follows.
    (a) Call the `getItem` method to find the position of the `item` in the heap. If the element is not found, exit the method.
    (b) If the element is found at position `foundIdx` in `elements`: replace the element at `foundIdx` with the last element in the heap (`elements[count - 1]`).
    (c) Decrease the number of elements (`count`) by 1.
    (d) Call the `reheapDown` method from position `foundIdx` to restore the heap property.
    (e) If the function pointer `removeItemData` is provided, call this function to free memory or handle the data of the removed element.
  - **Exceptions**: None.

- **bool contains(T item)**
  - **Functionality**: Checks whether the heap contains the `item`.
  - **Exceptions**: None.

- **int size()**
  - **Functionality**: Returns the number of elements currently in the heap.
  - **Exceptions**: None.

- **void heapify(T array[], int size)**

- **Functionality**: This method builds a heap from an array `array` of size `size`.
- **Implementation Guide**:
  (a) Iterate through each element in the `array`.
  (b) Call the `push` method to add each element to the heap and maintain the heap property.
- **Exceptions**: None.

• **bool empty()**
- **Functionality**: Checks whether the heap is empty.
- **Exceptions**: None.

• **void clear()**
- **Functionality**: Removes all elements from the heap and resets the heap to its initial empty state.
- **Exceptions**: None.

4 TASK-2: Multi-Layer Perceptron MLP

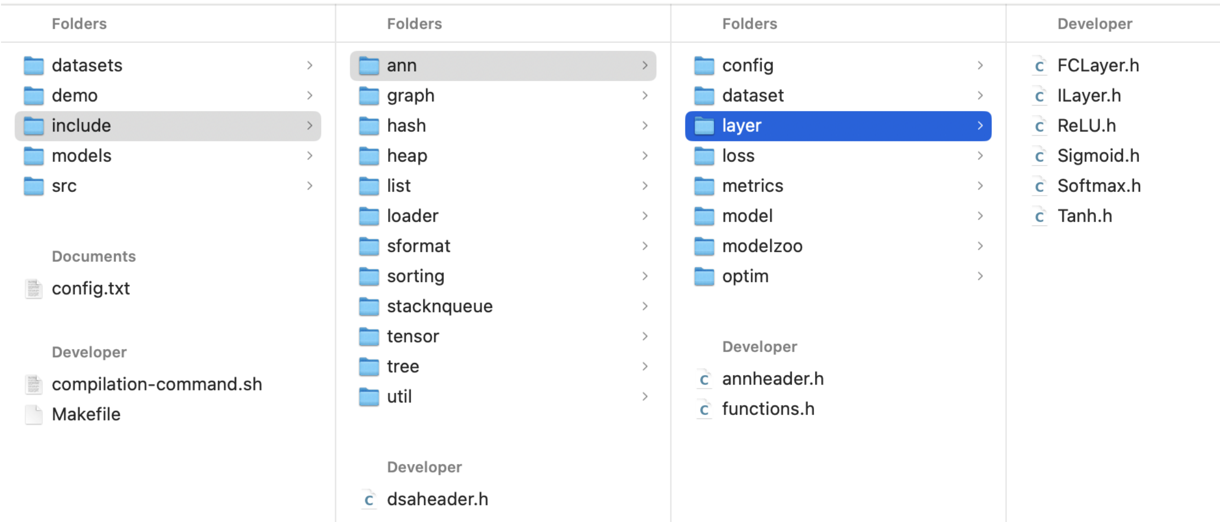
4.1 Implementation Method

To implement the neural network, students need to proceed **sequentially** through the following steps:

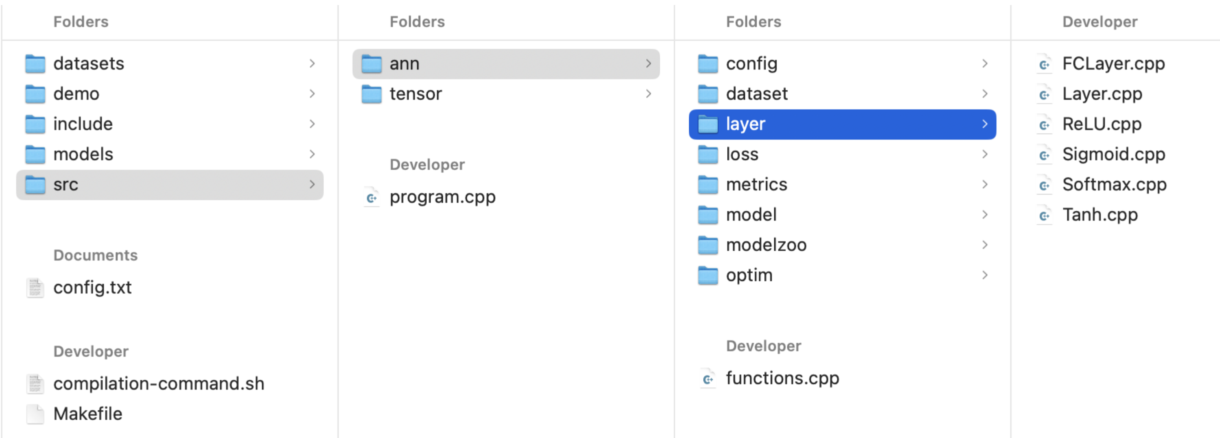
1. **(Mandatory)**: Study the guide on **Artificial Neural Networks** that has been published earlier.
2. Combine with the instructions in this document in the following sections.

4.2 Source Code Structure of the Neural Network

The source code of the neural network (ANN) consists of two types of files: **".h"** and **".cpp"**, located in **./include/ann** and **./src/ann**, respectively. To implement the ANN, we need multiple layers; therefore, if we put them all into one directory, it will be very difficult to manage. In this project, the classes for the ANN are organized into subdirectories of **./include/ann** and **./src/ann**. Figures 7 and 8 illustrate the contents of the classes in the **"layer"** group. The directories of the ANN serve the following purposes. **Note**, the directories with the following names in **./include/ann** contain **".h"** files, while those in **./src/ann** contain **".cpp"** files.



Hình 7: Source code organization of ANN, “.h“ file



Hình 8: Source code organization of ANN, “.cpp“ file

- **layer**: This directory contains the definitions of computational classes for building neural networks. The classes implemented in **TASK-2** include:
  1. Class **ILayer**: an abstract class that defines **virtual** methods for subclasses to **override**. This class is the parent of all the following classes.
  2. Fully connected layer, **FCLayer**.
  3. Class **ReLU**.
  4. Class **Sigmoid**.
  5. Class **Tanh**.
  6. Class **Softmax**.
- **loss**: This directory contains all classes defining different loss functions. The classes implemented in **TASK-2** include:

1. Class **ILossLayer**: an abstract class that defines **virtual** methods for subclasses to **override**. This class is the parent of all the following classes.
2. Class **CrossEntropy**.

- **metrics**: This directory contains all classes defining different types of metrics for various problems. The classes implemented in **TASK-2** include:
  1. Class **IMetrics**: an abstract class that defines **virtual** methods for subclasses to **override**. This class is the parent of all the following classes.
  2. Class **ClassMetrics**: **ClassMetrics** stands for **Classification Metrics**, which calculates common metrics to evaluate performance for **classification problems**.
- **model**: This directory contains all classes defining different types of neural network models; for example, single-label classification, multi-label classification, regression, etc. The classes implemented in **TASK-2** include:
  1. Class **IModel**: an abstract class that defines **virtual** methods for subclasses to **override**. This class is the parent of all the following classes.
  2. Class **MLPClassifier**: a multi-layer neural network model responsible for single-label classification.
- **optim**: This directory contains all classes defining various training algorithms for neural networks, etc. The classes implemented in **TASK-2** include:
  1. Class **IParamGroup**: an abstract class that defines **virtual** methods for subclasses to **override**. This class is the parent of all the following classes. The **step** function in the following classes directly updates the parameters in the group during the learning process.
     - **SGDParamGroup**
     - **AdaParamGroup**
     - **AdamParamGroup**
  2. Class **IOptimizer**: an abstract class that defines **virtual** methods for subclasses to **override**. This class is the parent of all the following classes. The **create_group** function in the following classes creates instances of **SGDParamGroup**, **AdaParamGroup**, or **AdamParamGroup** to manage parameters, depending on their update strategy.
     - **SGD**
     - **Adagrad**
     - **Adam**

- **dataset**: This directory contains the class **DSFactory** used to create some popular datasets.
- **config**: This directory contains the class **Config** used to manage configuration parameters during the creation of the neural network; such as, the directory containing the model, checkpoint names for models during training, etc.
- **modelzoo**: This directory contains some illustrative examples of creating and training models.

4.3 Implementation Order

Students should implement the classes in the following suggested order:

1. Classes in the **layer** directory.
2. Classes in the **loss** directory.
3. Classes in the **metrics** directory.
4. Classes in the **model** directory.
5. Classes in the **optim** directory.