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 1

Developing List Data Structures and Artificial Neural Networks

---

Ho Chi Minh City, 09/2024

ASSIGNMENT'S SPECIFICATION

Version 1.0

1 Introduction

1.1 Objectives and Tasks

The goal of the major assignments (BTLs) in this course is to help students develop data structures and use them to create various applications. In this semester, the application chosen as the main theme for all three major assignments is **neural networks and deep learning**; this is a topic with many applications in data analysis. All three assignments are based on the same theme and inherit from each other; that is, BTL-3 builds upon the source code from BTL-2 and BTL-1, and BTL-2 builds upon BTL-1. Therefore, students need to ensure this continuity. The tasks of the BTLs are presented as follows:

1. Major Assignment 1
   - **Develop** the data structure for lists; specifically, an array-based list and a linked list (the two classes **XArrayList** and **DLinkedList** in the provided source code).
   - **Utilize** the **xtensor** library to perform various tasks on multi-dimensional arrays (also known as tensors). This library has been downloaded and provided in the accompanying source code.
   - **Use** the developed lists to build a library of classes that allow users to create and use a multi-layer perceptron neural network (**M**ulti**L**ayer **P**erceptron, **MLP**). The **MLP** library includes many functions and features, but BTL-1 focuses only on the classes and methods that support the following two tasks:
     (a) Create and split datasets ready for processing by the **MLP**.
     (b) Create classes and methods to support inference by the **MLP**.
2. Major Assignment 2 (*detailed description will be released soon later.*)
   - **Develop** data structures for hash tables and heaps; specifically, the two classes **XHashMap** and **Heap** in the source code to be provided.
   - **Use** the hash table and heap data structures to implement classes and methods that support **training** neural networks. Specifically, the development focuses on the following two tasks for neural networks:
     (a) Implement training functionality for neural networks.

(b) Apply the **MLP** neural network to perform data analysis tasks such as classification and regression.

3. Major Assignment 3 (*detailed description will be released soon later.*)
   (a) **Develop** the data structure for graphs; specifically, the data structure for **directed graphs** and algorithms on graphs.
   (b) **Use** the graph data structure to build a library for **computational graphs** in the field of deep learning. Computational graphs in deep learning require many different computational nodes, many of which need to be accelerated by graphics cards (GPUs) for speed. Therefore, BTL-3 focuses only on a few basic nodes to help students understand the concepts and computational principles in deep learning.

1.2 Methodology

1. Preparation: **Download** and **review** the provided source code. **Note**, students are required to compile the source code in the assignments using **C++17**. The compiler tested by the course team is **g++**, and students are recommended to set up their environment to use **g++**.
2. **Utilizing** the **xtensor** library: Check the course website to download and run demos/guides on using this library.
3. **Developing** lists: Implement the **XArrayList** and **DLinkedList** classes in the **/include/list** directory.
4. **Developing and utilizing** the **MLP** library:
   (a) Refer to the course website for documentation on the **MLP** neural network.
   (b) Implement the classes in the **/include/ann** and **/src/ann** directory.
5. **Testing**: Ensure the program passes the provided sample test cases.
6. **Submission**: The assignments must be submitted on the system before the deadline.

1.3 Learning Outcomes for Assignment-1

Upon completing this assignment, students should be able to:

- **Use** the C/C++ programming language at an advanced level.
- **Utilize** available programming libraries, specifically the **xtensor** library.
- **Develop** the list data structure, in two versions: array-based and linked-based.

- **Select and apply** the list structure to develop a multilayer perceptron (MLP) neural network library.

1.4 Grading Components and Evaluation Method

The grading for Assignment-1 is divided into three components as detailed below. **Note**: students should complete the tasks in the order listed. Since the assignments are interdependent, if a student fails to complete some tasks, such as Task 3 (or even both 2 and 3, **Task 1 must be completed**), they will still need to complete them in Assignment-2 in order to **compile and run** the subsequent assignments successfully.

If students defer Task 2 and 3 to Assignment-2, they can only receive 50% of the points for those tasks.

1. Implementing the list: (50% **of the grade**); includes the following files in the **/include/list** directory:
   - **XArrayList.h**
   - **DLinkedList.h**
2. Implementing the classes for dataset and data loading: (20% **of the grade**); includes the following files in the **/include/ann** directory:
   - **dataset.h**
   - **dataloader.h**
3. Implementing other classes for the neural network: (30% **of the grade**); includes other files in the **/include/ann** and **/src/ann** directory.

Students must run evaluations using the provided sample test cases. The final assignment submission will be graded based on unpublished test cases.

2 Guidelines

2.1 On List Data Structures

2.1.1 Design Approach for Data Structures

In this course, the data structures are designed following a consistent template; namely:

1. Each data structure will have one or more abstract classes defining the operations supported by the structure. In the case of lists, this is the **IList** class located in the **/include/list** directory.
2. In practice, data structures should be able to contain elements of **any type**, such as **student objects**, **pointers to student objects**, or even just an **int**. Therefore, all data structures in this course utilize **templates** (in C++) to parameterize the element type.
3. All data structures are designed to encapsulate data and high-level details, separating the roles of **library developer** and **library user**. As such, all data structures, including lists, have been enhanced with an iteration feature (**iterator**) to facilitate element traversal from the user's perspective.

2.1.2 Overview of the List Data Structure

The list data structure in this assignment is designed with the following classes:

- The `IList` class, see Figure 1: this class defines a set of APIs supported by the list, whether implemented by arrays or by links. `IList` is the parent class of both `XArrayList` and `DLinkedList`. Some detailed points:
  - `IList` uses **templates** to parameterize the element type, allowing the list to contain elements of any type.
  - All APIs in `IList` are "pure virtual methods", meaning that derived classes from `IList` must override all these methods, supporting dynamic binding (polymorphism).
- The `XArrayList` and `DLinkedList` classes: these classes inherit from `IList` and implement all the APIs defined in `IList`, using arrays (as in `XArrayList`) and using links (as in `DLinkedList`).

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

- `virtual ~IList() {};`
  - Virtual destructor, used to ensure that the destructors of derived classes are called when an object is deleted through a base class pointer.
- `virtual void add(T e) = 0;`
  - Adds element `e` to the end of the list.
  - **Parameter**: `T e` — the element to add.

```cpp
template<typename T>
class IList {
public:
virtual ~IList(){};
    virtual void    add(T e)=0;
    virtual void    add(int index, T e)=0;
    virtual T       removeAt(int index)=0;
    virtual bool    removeItem(T item, void (*removeItemData)(T)=0)=0;
    virtual bool    empty()=0;
    virtual int     size()=0;
    virtual void    clear()=0;
    virtual int     indexOf(T item)=0;
    virtual string  toString(string (*item2str)(T&)=0 )=0;
};
```

Figure 1: `IList<T>`: Abstract class defining APIs for the list.

- `virtual void add(int index, T e) = 0;`
  - Inserts element `e` at position `index` in the list.
  - **Parameters**:
    - `int index` — the position where the element should be inserted.
    - `T e` — the element to insert.
- `virtual T removeAt(int index) = 0;`
  - **Description**: Removes and returns the element at position `index`.
  - **Parameter**: `int index` — the position of the element to remove.
  - **Return Value**: Returns the element at position `index`.
  - **Exception**: Throws `std::out_of_range` if `index` is invalid.
- `virtual bool removeItem(T item, void (*removeItemData)(T) = 0) = 0;`
  - **Description**: Removes the element `item` stored in the list.
  - **Parameters**:
    - `T item` — the element to remove.
    - `void (*removeItemData)(T)` — a function pointer to handle the removal of dynamic memory associated with the element, if necessary.
  - **Return Value**: Returns `true` if the element is found and removed, `false` otherwise.
- `virtual bool empty() = 0;`
  - **Description**: Checks if the list is empty.
  - **Return Value**: Returns `true` if the list is empty, `false` otherwise.
- `virtual int size() = 0;`

  - **Description**: Returns the number of elements currently stored in the list.
  - **Return Value**: The number of elements in the list.

- `virtual void clear() = 0;`
  - **Description**: Removes all elements from the list.
- `virtual int indexOf(T item) = 0;`
  - **Description**: Returns the index of the first occurrence of `item` in the list, or `-1` if `item` is not in the list.
  - **Parameter**: `T item` — the element to search for.
  - **Return Value**: The index of the first occurrence of `item`, or `-1` if not found.
- `virtual string toString(string (*item2str)(T&) = 0) = 0;`
  - **Description**: Converts the list to a string, with elements separated by a comma.
  - **Parameter**: `string (*item2str)(T&)` — a function pointer that converts an element of type `T` to a string.
  - **Return Value**: A string representing the list.

2.1.3 Array List

`XArrayList<T>` (see Figure 2) is an implementation of a list using an array to store elements of type `T`. In principle, `XArrayList<T>` must proactively maintain an array large enough to contain the elements within it. The manipulation of elements only pertains to APIs such as `add`, `remove`, and `removeItem`; therefore, when implementing these APIs, it is necessary to check the size of the current array to ensure sufficient space to store the elements.

In addition to methods as shown in Figure 2 for the APIs in `IList`, `XArrayList<T>` also requires additional supporting methods, see in the file **XArrayList.h**, in the **/include/list** directory.

1. **Attributes:**
   - `T* data`: A dynamic array that stores the elements of the list.
   - `int capacity`: The current capacity of the dynamic array `data`.
   - `int count`: The number of elements currently in the list.
2. **Constructor and Destructor:**
   - `XArrayList(int capacity)`: Constructor that initializes the list with an initial capacity.

```
template<typename T>
class XArrayList: public IList<T> {
protected:
    T*  data;
    int capacity;
    int count;
public:
    XArrayList(int capacity=10);
    ~XArrayList();

public:
    //Inherit from IList: BEGIN
    void    add(T e);
    void    add(int index, T e);
    T       removeAt(int index);
    bool    removeItem(T item, void (*removeItemData)(T)=0);
    bool    empty();
    int      size();
    void    clear();
    T&      get(int index);
    int      indexOf(T item);
    bool    contains(T item);
    string  toString(string (*item2str)(T&)=0 );
    //Inherit from IList: END
};

```

Figure 2: `XArrayList<T>`: List implemented using an array

- ~XArrayList(): Destructor, frees the memory allocated for the dynamic array data and its elements.

3. **Methods:**
   - **void add(T e)**
     - **Function**: Adds an element e to the end of the list.
     - **Exceptions**: None.
   - **void add(int index, T e)**
     - **Function**: Adds element e at the specified position index in the list.
     - **Exceptions**: If index is invalid (out of range [0, count]), throws out_of_range("Index is out of range!").
   - **T removeAt(int index)**
     - **Function**: Removes the element at position index and returns the removed element.
     - **Exceptions**: If index is invalid (out of range [0, count-1]), throws out_of_range("Index is out of range!").
   - **bool removeItem(T item, void (*removeItemData)(T) = 0)**
     - **Function**: Removes the first element in the list with a value equal to item.
     - **Exceptions**: None.
   - **bool empty()**
     - **Function**: Checks if the list is empty.
     - **Exceptions**: None.
   - **int size()**
     - **Function**: Returns the number of elements currently in the list.
     - **Exceptions**: None.
   - **void clear()**
     - **Function**: Removes all elements in the list and resets the list to its initial state.
     - **Exceptions**: None.
   - **T get(int index)**
     - **Function**: Returns a reference to the element at position index.
     - **Exceptions**: If index is invalid (out of range [0, count-1]), throws out_of_range("Index is out of range!").
   - **int indexOf(T item)**
     - **Function**: Returns the index of the first element with a value equal to item in the list, or -1 if not found.

  - **Exceptions**: None.
- **bool contains(T item)**
  - **Function**: Checks if the list contains an element with a value equal to `item`.
  - **Exceptions**: None.
- **string toString(string (*item2str)(T) = 0)**
  - **Function**: Returns a string representation of the elements in the list.
  - **Exceptions**: None.

2.1.4 Doubly Linked List

`DLinkedList<T>` (see Figure 3) is an implementation of a list using two links: **next** and **prev**. From the user's perspective, they do not need to be aware of the internal implementation details. Therefore, they are not required to be aware of the links. To achieve this, `DLinkedList<T>` needs a "**node**" object, which contains user data as well as two links: one to the next element and one to the previous element.

In addition to methods as shown in Figure 3 for the APIs in `IList`, `DLinkedList<T>` also requires additional supporting methods, see in the file **DLinkedList.h**, in the **/include/list** directory.

1. **class Node:**
   - `T data`: The data of the node.
   - `Node* next`: Pointer to the next node in the list.
   - `Node* prev`: Pointer to the previous node in the list.
2. **Constructor and Destructor:**
   - `DLinkedList()`: Constructor that initializes an empty doubly linked list.
   - `~DLinkedList()`: Destructor, frees the memory allocated for the nodes in the list.
3. **Methods:**
   - **void add(T e)**
     - **Function**: Adds an element `e` to the end of the list.
     - **Exceptions**: None.
   - **void add(int index, T e)**
     - **Function**: Adds element `e` at the specified position `index` in the list.
     - **Exceptions**: If `index` is invalid (out of range `[0, count]`), throws `out_of_range("Index is out of range!")`.

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

public:
    DLinkedList();
    ~DLinkedList();

    //Inherit from IList: BEGIN
    void    add(T e);
    void    add(int index, T e);
    T       removeAt(int index);
    bool    removeItem(T item, void (*removeItemData)(T)=0);
    bool    empty();
    int      size();
    void    clear();
    T&      get(int index);
    int      indexOf(T item);
    bool    contains(T item);
    string  toString(string (*item2str)(T&)=0 );
    //Inherit from IList: END

public:
    class Node{
    public:
        T data;
        Node *next;
        Node *prev;
        friend class DLinkedList<T>;

    public:
        Node(Node* next=0, Node* prev=0);
        Node(T data, Node* next=0, Node* prev=0);
    };
};

```

Figure 3: `DLinkedList<T>`: List implemented using doubly linked nodes (**next** and **prev**)

- **T removeAt(int index)**
  - **Function**: Removes the element at position `index` and returns the removed element.
  - **Exceptions**: If `index` is invalid (out of range `[0, count-1]`), throws `out_of_range("Index is out of range!")`.
- **bool removeItem(T item, void (*removeItemData)(T) = 0)**
  - **Function**: Removes the first element in the list with a value equal to `item`.
  - **Exceptions**: None.
- **bool empty()**
  - **Function**: Checks if the list is empty.
  - **Exceptions**: None.
- **int size()**
  - **Function**: Returns the number of elements currently in the list.
  - **Exceptions**: None.
- **void clear()**
  - **Function**: Removes all elements in the list and resets the list to its initial state.
  - **Exceptions**: None.
- **T& get(int index)**
  - **Function**: Returns a reference to the element at position `index`.
  - **Exceptions**: If `index` is invalid (out of range `[0, count-1]`), throws `out_of_range("Index is out of range!")`.
- **int indexOf(T item)**
  - **Function**: Returns the index of the first element with a value equal to `item` in the list, or `-1` if not found.
  - **Exceptions**: None.
- **bool contains(T item)**
  - **Function**: Checks if the list contains an element with a value equal to `item`.
  - **Exceptions**: None.
- **string toString(string (*item2str)(T&) = 0)**
  - **Function**: Returns a string representation of the elements in the list.
  - **Exceptions**: None.

2.2 Classes Related to Datasets and Data Loading

This implementation pertains to two groups of classes as follows.

1. The group of classes implementing datasets, which includes the following classes:
   (a) Class `Dataset`: in file **/include/ann/dataset.h**
   (b) Class `TensorDataset`: in file **/include/ann/dataset.h**
   (c) Class `ImageFolderDataset`: *this class is to be designed by the student.* Hint: this class is a subclass of `Dataset` and its constructor must take the name of the directory containing the data and labels.
2. The group of classes implementing data loading from datasets, which includes the following class:
   (a) Class `DataLoader`: in file **/include/ann/dataloader.h**

2.2.1 Class `Dataset`

To be compatible with data loading in `DataLoader`, any dataset must derive from `Dataset<DT, LT>`. All four methods in `Dataset<DT, LT>` are "virtual pure methods"; therefore, they must be overridden in derived classes. `Dataset<DT, LT>` is described in more detail as follows:

1. **Type Parameters `DT` and `LT`**:
   - `DT`: Data before being loaded into the neural network is described in numeric form; specifically, as a two-dimensional or multi-dimensional array. `DT` will be the data type of the elements in the array; typically `double` or `float`.
   - `LT`: Each unit of observed data, also known as an observation or sample, is usually labeled. For example, an image in an image classification problem has a label that is the name of the class; however, computers use numbers to indicate the class. `LT` is the type of the number representing the label.
2. **Methods**:
   - Method `len`: This method returns the size of the dataset; that is, how many samples (observations) are in the dataset. Some examples:
     – If the dataset is a matrix, then rows are data samples, also called feature vectors. Therefore, the size of the dataset will be the number of rows of the matrix.

  - If the dataset is a multi-dimensional matrix, then the size of the first dimension (dimension 0) is the number of samples in the dataset. This is also the case for `TensorDataset` discussed below.
  - If the dataset is the number of files in a directory, then the number of files is the size of the dataset. This is also the case for `ImageFolderDataset`.
- Method `getitem`: This method returns an object of type `DataLabel`. The `DataLabel` class is provided in the source file **dataset.h**.
- Methods `get_data_shape` and `get_label_shape`: These two methods return the shape of the data and labels. See more about multi-dimensional arrays in the **xtensor** library to better understand this property.

```
template<typename DType, typename LType>
class Dataset{
private:
public:
    Dataset(){};
    virtual ~Dataset(){};

    virtual int len()=0;
    virtual DataLabel<DType, LType> getitem(int index)=0;
    virtual xt::svector<unsigned long> get_data_shape()=0;
    virtual xt::svector<unsigned long> get_label_shape()=0;

};
```

Figure 4: `Dataset<T>`: The base class for any dataset.

2.2.2 Class TensorDataset

`TensorDataset<DT, LT>` (see Figure 5) is a subclass of `Dataset<DT, LT>`; it represents datasets **already in** tensor format. This means that the data of `TensorDataset<DT, LT>` has the type `xt::xarray<DT>` and the dataset labels have the type `xt::xarray<LT>`. The meaning of member variables and methods is presented as follows.

1. **Member Variables**:
   - `data` and `label`: these two tensors have element types `DT` and `LT` respectively. They contain the data and labels in the dataset.
   - `data_shape` and `label_shape`: these two tensors contain the shape of the data for the `data` and `label` variables respectively. With the **xtensor** library, the shape of the data is of type `xt::svector`.

2. **Constructor**: The constructor `TensorDataset(.,.)` has two parameters. Students must assign to the member variables and store the data shape of the labels in the corresponding variables.
3. **Other methods overridden from** `Dataset`: Implement these methods according to the description in the `Dataset` class.

```
template<typename DType, typename LType>
class TensorDataset: public Dataset<DType, LType>{
private:
    xt::xarray<DType> data;
    xt::xarray<LType> label;
    xt::svector<unsigned long> data_shape, label_shape;

public:
    /* TensorDataset:
     * need to initialize:
     * 1. data, label;
     * 2. data_shape, label_shape
    */
    TensorDataset(xt::xarray<DType> data, xt::xarray<LType> label){
        /* TODO: your code is here for the initialization
         */
    }
    /* len():
     *  return the size of dimension 0
    */
    int len(){
        /* TODO: your code is here to return the dataset's length
         */
        return 0; //remove it when complete
    }

    /* getitem:
     * return the data item (of type: DataLabel) that is specified by index
     */
    DataLabel<DType, LType> getitem(int index){
        /* TODO: your code is here
         */
    }

    xt::svector<unsigned long> get_data_shape(){
        /* TODO: your code is here to return data_shape
         */
    }
    xt::svector<unsigned long> get_label_shape(){
        /* TODO: your code is here to return label_shape
         */
    }
};
```

Figure 5: `TensorDataset<DT, LT>`: The class representing datasets already in tensor format.

2.2.3 The Dataloader Class

The purpose of developing the dataset and dataloader classes is to support running the code snippet shown in Figure 6. This code snippet can be understood as follows:

- **Line 15**: This is used to create a dataset of type TensorDataset. This dataset takes tensors created in the previous lines.
- **Line 16**: This creates a dataloader for the dataset with the following information:
  1. **&ds**: The dataloader for the Tensor dataset mentioned above. Note that it takes the address of the loader to use polymorphism.
  2. **batch_size=30**: The dataset has a total of `100` samples; the loader will divide them into batches of `30` samples. Thus, there will be `3` batches with `10` samples remaining. Since the last parameter is `drop_last=false`, the remaining `10` samples will be included in the final batch.
  3. **shuffle=true**: This means **begin batching**, the loader will **shuffle the data** (i.e., shuffle the order of the data samples) randomly (uniform distribution). If this parameter is `false`, the data will not be shuffled randomly.
  4. **drop_last=false**: As described above.
- **Lines 17-19**: These lines signify "for each batch loaded by the loader, process this batch of data". The processing in lines `18-19` involves printing the shapes of the data and labels in the batch.

Based on the above example and the implementation method to support the "foreach" syntax as done in the `XArrayList` and `DLinkedList` classes, students should research how to implement the `DataLoader` class. The source code for this class is provided in Figure 7.

2.3 Classes for MLP Neural Networks

2.3.1 Definition and Implementation Files

All classes implemented in this section have their source code located in the following two places within the project (relative to the project directory).

- **/include/ann/**: contains definition files (‘‘.h‘‘, header).
- **/src/ann/**: contains implementation files (“.cpp“, source)

```cpp
#include <iostream>
#include <iomanip>
#include <sstream>
using namespace std;

#include "ann/functions.h"
#include "ann/xtensor_lib.h"
#include "ann/dataset.h"
#include "ann/dataloader.h"

int main(int argc, char** argv) {
    int nsamples = 100;
    xt::xarray<double> X = xt::random::randn<double>({nsamples, 10});
    xt::xarray<double> T = xt::random::randn<double>({nsamples, 5});
    TensorDataset<double, double> ds(X, T);
    DataLoader<double, double> loader(&ds, 30, true, false);
    for(auto batch: loader){
        cout <<  shape2str(batch.getData().shape()) << endl;
        cout <<  shape2str(batch.getLabel().shape()) << endl;
    }
    return 0;
}
```

Figure 6: Using `DataLoader`.

2.3.2 List of Classes to Implement

The list of classes and functions to implement is provided in Table 1. Students should refer to the source code for details of the functions that need to be implemented. More detailed documentation on this implementation will be added subsequently.

Table 1: Table of Classes and Files to Implement

Class	Header File	Source File
Layer	Layer.h	Layer.cpp
FCLayer	FCLayer.h	FCLayer.cpp
ReLU	ReLU.h	ReLU.cpp
Softmax	Softmax.h	Softmax.cpp
BaseModel	BaseModel.h	BaseModel.cpp
Supplementary Functions for `xtensor`	xtensor_lib.h	xtensor_lib.cpp
Other Functions	funtions.h	funtions.cpp

```
template<typename DType, typename LType>
class DataLoader{
public:
private:
    Dataset<DType, LType>* ptr_dataset;
    int batch_size;
    bool shuffle;
    bool drop_last;
    /*TODO: add more member variables to support the iteration*/
public:
    DataLoader(Dataset<DType, LType>* ptr_dataset,
            int batch_size,
            bool shuffle=true,
            bool drop_last=false){
        /*TODO: Add your code to do the initialization */
    }
    virtual ~DataLoader(){}

    ////////////////////////////////////////////////////////////
    // The section for supporting the iteration and for-each to DataLoader
    //
    /// START: Section
    ////////////////////////////////////////////////////////////

    /*TODO: Add your code here to support iteration on batch*/

    ////////////////////////////////////////////////////////////
    // The section for supporting the iteration and for-each to DataLoader
    //
    /// END: Section
    ////////////////////////////////////////////////////////////
};
```

Figure 7: `DataLoader<DT, LT>`: The class that splits data into batches and supports the "**foreach batch in the-dataset**" syntax.

3 Coding and Compilation

3.1 Development Strategy

BTL-1 consists of three components with point distributions as detailed in Section **??**. Students should follow the weekly tasks to complete the work.

To simplify compilation of each part, students should split the project into smaller projects. Specifically, the initial project should focus only on developing the list, which includes just two files: `XArrayList.h` and `DLinkedList.h`.

After successfully compiling the list project, students should gradually proceed with the remaining two tasks.

3.2 Compilation

Students **should** integrate the code into an IDE that they find convenient and use the interface to compile. For example, the instructor used **Apache NetBeans IDE 13** for development.

If compilation from the command line is needed, students can use the following command: **g++ -Iinclude -Isrc -std=c++17 your_main.cpp**

4 Submission

Students will fill in the code into the provided files, compress them, and then submit them to the system. Submission instructions will be detailed later.

5 Additional Regulations

- Students must complete this project independently and prevent others from stealing their results. Failure to do so will result in disciplinary action for academic dishonesty.
- All decisions made by the instructor in charge of the project are final.
- Students are not allowed to provide testcases after grading but may provide information on testcase design strategies and the distribution of student counts for each testcase.
- The content of the project will be harmonized with a question in the exam with similar content.

6 Tracking Changes Across Versions

- September 09, 2024: Version 1.0 has been released.

——————————END——————————