Objectives

In this homework, you are expected to build a **Discrete Event Simulation (DES)** that models the operations of a bank. The goal is to simulate how customers arrive, wait in queues, receive service, and leave the system based on time-ordered events.

To make this possible, you are expected to first implement a **Singly Linked List** and use it to create a queue structure. You are also expected to implement a **Sorted Doubly Linked List**, which will be used to maintain both the **priority of customers** and the **ordering of events** by their time of occurrence. Using these data structures, you are expected to integrate them into a complete event-driven simulation of a banking system.

Details of the data structures will be explained in the following sections.

1 SinglyLinkedList Implementation

A **Singly Linked List (SLL)** is a dynamic data structure made up of nodes connected in a single direction. Each node is represented by the `SLLNode` structure, which stores a data element and a pointer to the next node.

The list uses a **dummy head node**, a design choice that simplifies insertion and deletion operations by providing a consistent starting point for the list.

```
template <class T>
struct SLLNode {
    T data;
    SLLNode<T>* next;

    SLLNode(const T& data, SLLNode<T>* next = NULL);
};

template <class T>
class SinglyLinkedList {
private:
    SLLNode<T>* head; // dummy node

public:
    // ---- Constructors & Destructor ----
    SinglyLinkedList();
    SinglyLinkedList(const SinglyLinkedList<T>& other);
    SinglyLinkedList<T>& operator=(const SinglyLinkedList<T>&
       other);
    ~SinglyLinkedList();

    // ---- Basic Operations ----
    void PushFront(const T& value);
    void PushBack(const T& value);
```

```
    T PopFront();
    T PopBack();

    // ---- Accessors ----
    bool IsEmpty() const;
    size_t GetSize() const;
    T& Front();
    const T& Front() const;
    T& Back();
    const T& Back() const;

    // ---- Searching & Removal ----
    bool Contains(const T& value) const;
    bool Remove(const T& value);

    // ---- Utility ----
    void Clear();
    void Reverse();

    // ---- Operator Overload for Printing ----
    template <class U>
    friend std::ostream& operator<<(std::ostream& os, const
       SinglyLinkedList<U>& list);
};
```

1.1 SLLNode

`SLLNode` structure holds the information about a node in the `SinglyLinkedList`. It is templated because `SinglyLinkedList` is a templated class.

1.2 SinglyLinkedList

The `SinglyLinkedList` class is a classical linked list implementation built using `SLLNode` structures.

> For this class to work properly, the template type `T` must have a **default constructor** and a **copy constructor**.

Hint

You may want to implement `Clear()` first that can be used in the destructor and other member functions.

1.2.1 `SinglyLinkedList()`

Default constructor of an empty linked list. The list uses a **dummy head node**, which is initialized using the **default constructor of** `T`. This design simplifies insertion and deletion operations by providing a consistent starting point for the list.

This function is implemented for you.

1.2.2 `SinglyLinkedList(const SinglyLinkedList<T>& other)`

The copy constructor of the class. It should follow **deep copy semantics**. Which will create a copy of all the nodes of "other", and wire them (setting the pointers) appropriately.

1.2.3 `SinglyLinkedList<T>& operator=(const SinglyLinkedList<T>& other)`

The copy assignment operator of the class. Most of the functionality of this function is similar to the copy constructor, the only difference is that the current object will have an existing node chain and it should be deleted beforehand

> Do not forget to check for self-assignment in the copy assignment operator.

1.2.4 `~SinglyLinkedList()`

The destructor of the class. It should delete the node chain so that there won't be any memory leaks.

1.2.5 `void PushFront(const T& value)`

Inserts a new node containing `value` immediately after the dummy head node, effectively adding the element to the beginning of the list.

1.2.6 `void PushBack(const T& value)`

Inserts a new node containing `value` at the end of the list by traversing from the dummy head node until the last element and linking the new node after it.

1.2.7 `T PopFront()`

Removes the first actual element (right after the dummy head) and returns the value. If the list is empty, this function should throw an `EmptyCollectionException`.

1.2.8 `T PopBack()`

Removes the last element of the list and returns the value. If the list is empty, this function should throw an `EmptyCollectionException`.

1.2.9 `bool IsEmpty() const`

Returns `true` if there is no element after the dummy head; otherwise returns `false`.

1.2.10 `size_t GetSize() const`

Returns the number of elements currently stored in the list (excluding the dummy head).

1.2.11 `T& Front()` and `const T& Front() const`

Returns a reference to the value of the first actual element (right after the dummy head). If the list is empty, throws an `EmptyCollectionException`. Both const and non-const overloads are provided for flexibility in use.

1.2.12 `T& Back()` and `const T& Back() const`

Returns a reference to the value of the last element in the list. If the list is empty, throws an `EmptyCollectionException`. Both const and non-const overloads are available.

1.2.13 `bool Contains(const T& value) const`

Returns `true` if an element equal to `value` exists in the list; otherwise returns `false`.

1.2.14 `bool Remove(const T& value)`

Removes the first occurrence of `value` from the list and returns `true` on success; otherwise returns `false`.

1.2.15 `void Clear()`

Removes all elements from the list, leaving only the dummy head node.

1.2.16 `void Reverse()`

Reverses the order of the elements stored in the list (dummy head remains at the front).

1.2.17 `std::ostream& operator<<(std::ostream& os, const SinglyLinkedList<T>& list)`

Overloads the output stream operator to print the elements of the list in their traversal order. Each element's value is written to the stream, separated by an arrow symbol (`->`), producing an output such as `10 -> 20 -> 30`.

This function is implemented for you.

2 Queue Implementation

The `Queue` class represents a standard first-in first-out (FIFO) or first-come first-served (FCFS) data structure built on top of the `SinglyLinkedList` which is a wrapper class. It supports basic queue operations such as enqueue, dequeue, and front element access.

```
template <class T>
struct SLLNode {
    T data;
    SLLNode<T>* next;

    SLLNode(const T& data, SLLNode<T>* next = NULL);
};
```

```
template <class T>
class Queue {
private:
    SinglyLinkedList<T> list; // underlying linked list

public:
    // ---- Constructors & Destructor ----
    Queue();
    Queue(const Queue<T>& other);
    Queue<T>& operator=(const Queue<T>& other);
    ~Queue();

    // ---- Core Operations ----
    void Enqueue(const T& value);
    T Dequeue();
    const T& Front() const;

    // ---- Utility ----
    bool IsEmpty() const;
    size_t GetSize() const;
    void Clear();

    // ---- Operator Overload for Printing ----
    template <class U>
    friend std::ostream& operator<<(std::ostream& os, const Queue
        <U>& q);
};
```

2.1 Queue Class

2.1.1 Queue()

Default constructor that creates an empty queue.

This function is implemented for you.

2.1.2 Queue(const Queue<T>& other)

Copy constructor that creates a new queue as a **deep copy** of `other`.

This function is implemented for you.

2.1.3 Queue<T>& operator=(const Queue<T>& other)

Assigns the contents of `other` to this queue, replacing any existing elements. It follows **deep copy semantics** to ensure the new queue owns separate copies of its elements.

This function is implemented for you.

2.1.4 ~Queue()

Destructor that deletes all elements in the queue and frees allocated memory.

This function is implemented for you.

2.1.5 `void Enqueue(const T& value)`

Adds a new element with the given `value` to the back of the queue.

2.1.6 `T Dequeue()`

Removes and returns the element at the front of the queue. If the queue is empty, throws an `EmptyCollectionException`.

2.1.7 `const T& Front() const`

Returns a reference to the value at the front of the queue without removing it. If the queue is empty, throws an `EmptyCollectionException`.

2.1.8 `bool IsEmpty() const`

Returns `true` if the queue has no elements; otherwise returns `false`.

2.1.9 `size_t GetSize() const`

Returns the number of elements currently stored in the queue.

2.1.10 `void Clear()`

Removes all elements from the queue and resets it to an empty state.

2.1.11 `std::ostream& operator<<(std::ostream& os, const Queue<T>& q)`

Overloads the output stream operator to print all elements of the queue in order from front to back, separated by arrows (`->`). For example: `5 -> 10 -> 15`. **This function is implemented for you.**

3 Multiple Queue Implementation

The `MultipleQueue Class` class manages multiple independent queues, each represented by a `Queue<T>` instance. It automatically assigns new elements to the queue with the smallest size, balancing the load among all queues.

```
template <class T>
class MultipleQueue {
private:
    Queue<T>* queues;
    const size_t count;

public:
    // ---- Constructors & Destructor ----
    MultipleQueue(size_t n);
    ~MultipleQueue();
```

```
    // ---- Core Operations ----
    int Enqueue(const T& value);
    T Dequeue(size_t index);
    const Queue<T>& GetQueue(size_t index) const;

    // ---- Utility ----
    size_t GetNumberOfQueues() const;
    size_t GetTotalSize() const;
    size_t GetSizeOfQueue(size_t index) const;

    // ---- Operator Overload for Printing ----
    template <class U>
    friend std::ostream& operator<<(std::ostream& os, const
       MultipleQueue<U>& mq);
};
```

3.1 MultipleQueue Class

3.1.1 MultipleQueue(size_t n)

Constructs a `MultipleQueue` with `n` internal queues, each initialized as empty.

This function is implemented for you.

3.1.2 ~MultipleQueue()

Deletes all dynamically allocated queues and frees the associated memory.

This function is implemented for you.

3.1.3 int Enqueue(const T& value)

Adds `value` to one of the internal queues. The element is always placed into the queue that currently has the smallest number of elements, keeping all queues approximately balanced in size.

If multiple queues have the same smallest size, the element is added to the queue with the smallest index. The function returns the index of the queue where the new element was inserted. If there are no queues available, it returns `-1`.

3.1.4 T Dequeue(size_t index)

Removes and returns the front element from the *queue at position* ***index*** (i.e., `index` specifies **which queue** to be dequeued). If all the queues are empty, throws an `EmptyCollectionException`; if `index` is out of range, throws an `IndexOutOfRangeException`.

3.1.5 const Queue<T>& GetQueue(size_t index) const

Returns a reference to the queue at the specified `index`. If the index is invalid, throws an `IndexOutOfRangeException`.

3.1.6 size_t GetNumberOfQueues() const

Returns the total number of queues managed by this object.

3.1.7 size_t GetTotalSize() const

Returns the total number of elements across all queues.

3.1.8 size_t GetSizeOfQueue(size_t index) const

Returns the number of elements in the queue at the given `index`. If the index is invalid, returns `0`.

3.1.9 std::ostream& operator<<(std::ostream& os, const MultipleQueue<T>& mq)

Prints the contents of each internal queue in order, prefixed with its queue number (e.g., `Queue 0: ...`).

This function is implemented for you.

4 SortedDoublyLinkedList Implementation

The `SortedDoublyLinkedList` class implements a **sorted** doubly linked list structure using `DLLNode` objects. The list automatically keeps its elements in **ascending order** based on comparisons between items as they are inserted. It maintains two dummy nodes, a **head** and a **tail**, to simplify insertion and deletion operations, ensuring that every element is located between these dummy nodes.

```
template <class T>
struct DLLNode {
    T item;
    DLLNode<T>* next;
    DLLNode<T>* prev;

    DLLNode(const T& item,
            DLLNode<T>* nextNode = NULL,
            DLLNode<T>* prevNode = NULL);
};

template <class T>
class SortedDoublyLinkedList {
private:
    DLLNode<T>* head; // dummy head
    DLLNode<T>* tail; // dummy tail

public:
    // ---- Constructors & Destructor ----
    SortedDoublyLinkedList();
    SortedDoublyLinkedList(const SortedDoublyLinkedList<T>& other
       );
```

```
    SortedDoublyLinkedList<T>& operator=(const
        SortedDoublyLinkedList<T>& other);
    ~SortedDoublyLinkedList();

    // ---- Core Operations ----
    void InsertItem(const T& item);
    void InsertItemPrior(const T& item);
    T RemoveFirstItem();
    T RemoveFirstItem(int priority);
    T RemoveLastItem(int priority);

    // ---- Query ----
    const T& FirstItem() const;
    const T& LastItem() const;
    bool IsEmpty() const;
    size_t GetSize() const;

    // ---- Modification ----
    void ChangePriority(int oldPriority, int newPriority);
    void SplitByPriority(SortedDoublyLinkedList<T>& low,
                         SortedDoublyLinkedList<T>& high,
                         int pivotPriority);
    void SplitAlternating(SortedDoublyLinkedList<T>& listA,
                          SortedDoublyLinkedList<T>& listB);

    static SortedDoublyLinkedList<T> Merge(const
        SortedDoublyLinkedList<T>& a,
    const SortedDoublyLinkedList<T>& b);

    // ---- Utility ----
    void Clear();

    // ---- Operator Overload for Printing ----
    template <typename U>
    friend std::ostream& operator<<(std::ostream& os, const
        SortedDoublyLinkedList<U>& list);
};
```

4.1 DLLNode

`DLLNode` structure holds the information about a node in the `SortedDoublyLinkedList`. It points to both the next and previous nodes in the list, allowing for bidirectional traversal.

4.2 SortedDoublyLinkedList Class

> For this class to function properly, the template type `T` must support **comparison operators** (`<`, `==`) so that items can be inserted in sorted order. In this homework, all classes used with `SortedDoublyLinkedList` will have these operators defined.

Be careful

The type `T` is only required to implement the `operator<` and `operator==`. This does **not** imply that `operator>`, `operator<=`, or `operator>=` must also be defined.

4.2.1 `SortedDoublyLinkedList()`

Default constructor that initializes an empty list with a dummy head and dummy tail. The dummy nodes are linked together, forming an empty but structurally valid list.

This function is implemented for you.

4.2.2 `~SortedDoublyLinkedList()`

Destroys the list and deallocates all dynamically created nodes.

4.2.3 `SortedDoublyLinkedList(const SortedDoublyLinkedList<T>& other)`

Creates a **deep copy** of another sorted doubly linked list, duplicating each node in order.

4.2.4 `SortedDoublyLinkedList<T>& operator=(const SortedDoublyLinkedList<T>& other)`

Replaces the contents of this list with those of `other`. Any existing nodes are deleted, and new ones are allocated to maintain a **deep copy**. Again, be careful about self-assignment.

4.2.5 `void InsertItem(const T& item)`

Inserts `item` into the list while preserving the overall ascending order according to the comparison operator of `T`. Items with lower priority values appear earlier in the list. If multiple elements share the same priority, the new element is inserted **after** all existing elements with the same priority.

4.2.6 `void InsertItemPrior(const T& item)`

Similar to `InsertItem`, this function also preserves ascending order based on priority. However, when multiple elements have the same priority, the new element is inserted **before** all of the existing elements with that priority.

4.2.7 `T RemoveFirstItem()`

Removes and returns the first real element (after the dummy head). Throws `EmptyCollectionException` if the list is empty.

4.2.8 `T RemoveFirstItem(int priority)`

Removes and returns the first item with the specified priority value. If the list is empty throws `EmptyCollectionException`. If no matching item is found, throws an `NotFoundException`.

4.2.9 `T RemoveLastItem(int priority)`

Removes and returns the last item with the specified priority value. If the list is empty throws `EmptyCollectionException`. If no matching item is found, throws an `NotFoundException`.

4.2.10 `const T& FirstItem() const`

Returns a reference to the first real item in the list (after the dummy head). Throws `EmptyCollectionException` if the list is empty.

4.2.11 `const T& LastItem() const`

Returns a reference to the last real item in the list (before the dummy tail). Throws `EmptyCollectionException` if the list is empty.

4.2.12 `bool IsEmpty() const`

Returns `true` if there are no elements between the dummy head and dummy tail.

4.2.13 `size_t GetSize() const`

Counts and returns the total number of elements in the list (excluding dummy nodes).

4.2.14 `void SplitByPriority(SortedDoublyLinkedList<T>& low, SortedDoublyLinkedList<T>& high, int pivotPriority)`

Divides the current list into two separate lists based on the given `pivotPriority`. All items whose priority is **less than or equal to** `pivotPriority` are moved into the list `low`, while all items with **greater priority** are moved into the list `high`. After this operation, the original list **does not change**, and both `low` and `high` preserve their internal sorted order.

4.2.15 `void SplitAlternating(SortedDoublyLinkedList<T>& listA, SortedDoublyLinkedList<T>& listB)`

Distributes the elements of the current list alternately between `listA` and `listB` without modifying the original list. Specifically, the 1st, 3rd, 5th, and so on elements are copied into `listA`, while the 2nd, 4th, 6th, and so on elements are copied into `listB`. Both resulting lists preserve the relative order of the elements as they appear in the original list.

4.2.16 `static SortedDoublyLinkedList<T> Merge(const SortedDoublyLinkedList<T>& a, const SortedDoublyLinkedList<T>& b)`

Creates and returns a new `SortedDoublyLinkedList` that contains all elements from both input lists `a` and `b`. The resulting list preserves ascending order by merging the elements as in the merge step of the merge sort algorithm. Neither of the original lists is modified during this operation — new nodes are created for the merged list. If both input lists are empty, the returned list is also empty.

4.2.17 `void Clear()`

Removes all elements from the list, leaving only the dummy head and dummy tail nodes.

4.2.18 `std::ostream& operator<<(std::ostream& os, const SortedDoublyLinkedList<T>& list)`

Prints the contents of the list from head to tail, skipping dummy nodes. Each element is separated by an arrow (`->`).

This function is implemented for you.

5 Entities

This section introduces the main entities used in the simulation, each representing a real-world role in the banking system. These classes are already implemented for you and should be used as they are. You can refer to their definitions and implementations in `Entities.h` and `Entities.cpp`.

5.1 `Customer`

The `Customer` class models a bank customer waiting to receive service. Each customer has an **ID**, a **priority level**, and a **service time**. Customers are compared primarily by their priority levels, which determine their order in the sorted data structures. Lower priority values indicate higher service priority. This class also overloads comparison and assignment operators to be used by the `SortedDoublyLinkedList`.

5.2 `Event`

The `Event` class represents a time-based occurrence in the simulation, such as a customer arrival, joining or leaving a queue, starting service, or completing service. Each event contains a **timestamp**, an **event type**, a related `Customer` object, and an index representing the banker or officer responsible for handling it. Events are automatically ordered by their time points in the simulation's event queue.

5.3 `SecurityOfficer`

The `SecurityOfficer` class represents the bank employee who initially interacts with arriving customers. In a real-world scenario, security officers are responsible for understanding the customer's purpose, for example, whether they want to open an account, withdraw cash, or make a transfer, and assigning an appropriate priority or directing them to the correct service desk. They act as the first stage of organization within the bank, ensuring that customers are served in a fair and efficient order. Each officer has a unique **ID** and a **dispatch time**, representing how long it takes for them to process and classify a newly arrived customer.

5.4 `Banker`

The `Banker` class models a bank clerk who serves customers assigned by the simulation. Each banker has a unique **ID** and a **service time**, representing how long it takes them to complete a customer's transaction. Bankers interact with customers through events such as `SERVICE_STARTED` and `SERVICE_COMPLETED`.

6 Bank Simulation and Discrete Event System

In this part of the Homework, you will integrate the previously implemented data structures into a complete **Discrete Event Simulation (DES)** that models how a bank operates over time.

A Discrete Event Simulation is a computational model where the state of the system changes only at specific moments, called **events**. Each event represents a significant action, such as a customer arriving, joining a queue, or being served by a banker. Between these events, no changes occur. Time simply advances to the next scheduled event.

6.1 Bank Simulation Overview

In this simulation, the bank operates with the following main components:

- **Customers:** Individuals arriving at the bank to receive service. Each customer has a unique ID, a priority level, and a service time.
- **Security Officers:** Employees who process incoming customers. They simulate the time it takes for a customer to be dispatched to a service queue.
- **Bankers:** Employees who serve customers after they are dispatched by the security officers.
- **Queues:** The simulation involves three different queue structures, each serving a distinct purpose:
  - **Waiting Queues** implemented using `MultipleQueue<Customer>`, these represent the physical waiting lines in front of security officers. Each officer manages one queue, and arriving customers selects the queue with the fewest people.
  - **Service Queue** implemented using `SortedDoublyLinkedList<Customer>`, this structure holds customers who are ready to be served by bankers. It maintains ascending order based on priority values, ensuring that higher-priority customers (lower priority numbers) are served first.
  - **Event Queue** implemented using `SortedDoublyLinkedList<Event>`, this is the core of the discrete event system. It keeps all upcoming events sorted by their scheduled **time of occurrence**, determining the order in which actions happen in the simulation.

To better understand the logic of this simulation, imagine a busy bank in **Kızılay** during lunch hour. Around fifty to a hundred people rush in after their break, all trying to finish their banking tasks before returning to work. Each **security officer** stands in front of their own line, serving customers one by one. When a new customer arrives, they

look around and join the line with the fewest people (just like in a supermarket); this represents the **waiting queues**. After waiting for their turn, customers are dispatched into the **service queue**, where their **priority level** determines who will be served first by the available bankers. Inside, **bankers** serve each customer according to the service queue order. The duration of the service depends on both the customer's required banking task and the banker's individual processing time. Meanwhile, the simulation keeps track of every step arrivals, queue movements, and completed services in the **event queue**.

6.2 Event Types

Every event in the simulation has a specific **type**, representing a distinct real-world activity within the bank:

- `ARRIVAL`: A customer arrives at the bank and is ready to be processed by a security officer.
- `QUEUE_ENTERED`: A customer joins the waiting queue that is handled by a specific security officer.
- `QUEUE_EXITED`: A customer finishes being processed by a security officer and moves to the service area.
- `SERVICE_STARTED`: A banker starts serving a customer.
- `SERVICE_COMPLETED`: A banker finishes serving a customer.
- `INVALID_EVENT`: A fallback type used for uninitialized or invalid events.
- `FINISHED`: The simulation has completed all events.

Each event is represented by an instance of the `Event` class and stored inside the event queue. As the simulation runs, the next event (the one with the smallest time value) is removed from the queue and handled accordingly.

6.3 `Bank` Class

The `Bank` class represents the main simulation environment for the discrete event system (DES). It integrates all previously implemented data structures and entities, including `MultipleQueue<Customer>` for waiting customers, `SortedDoublyLinkedList<Customer>` for prioritized service order, and `SortedDoublyLinkedList<Event>` for time-ordered event management.

6.3.1 `Bank(int numBankers, const int* bankerServiceTimes, int numSecurityOfficers, const int* officerDispatchTimes, int bankersStartTime)`

Initializes the simulation environment by creating bankers and security officers with the given service and dispatch times. Also populates the event queue with initial `SERVICE_STARTED` events for each banker, simulating the system's starting state.

This function is implemented for you.

6.3.2 ~Bank()

Destroys the bank object and releases all dynamically allocated memory, including bankers and security officers.

6.3.3 void AddCustomer(int id, int priority)

Creates a new `Customer` with the given `id` and `priority`. Then, it inserts an `ARRIVAL` event into the event queue to represent the moment this customer enters the system.

6.3.4 EventResult DoSingleEventIteration()

Processes the next event in the simulation by removing the first (earliest) event from the event queue and executing the corresponding action. Depending on the event type, this may cause new events such as `QUEUE_ENTERED`, `QUEUE_EXITED`, `SERVICE_STARTED`, or `SERVICE_COMPLETED` to be generated. Returns an `EventResult` describing the executed event and its outcomes.

6.3.5 std::ostream& operator<<(std::ostream& stream, const Bank& b)

Overloaded output operator that prints the current state of the bank simulation, including the contents of the waiting queues, service queue, and event queue.

This function is implemented for you.

6.4 Processing Events

- When a customer arrives (by `AddCustomer()` function), they are immediately placed into the event queue with an `ARRIVAL` event.
- Upon processing the `ARRIVAL` event, the customer selects the queue with the fewest people and joins it (is placed by MultipleQueue), generating a `QUEUE_ENTERED` event at the same time as `ARRIVAL` event.
- When `QUEUE_ENTERED` event is processed, if there is no other customer in the queue, the security officer begins dispatching the customer and generating a `QUEUE_EXITED` event at `currentTime + officerDispatchTime`; otherwise no event is generated.
- When `QUEUE_EXITED` event is processed, the customer is removed from the waiting queue and added to the service queue. If there are still customers waiting in the waiting queue, a new `QUEUE_EXITED` event is generated for the next customer.
- `SERVICE_STARTED` events are pre-populated for each banker at `bankersStartTime` when the bank class is created. When processed, if there are customers in the service queue, the banker begins serving the highest-priority customer, generating a `SERVICE_COMPLETED` event at `currentTime + bankerServiceTime + customer's serviceTime`.
- When `SERVICE_COMPLETED` event is processed, the banker finishes serving the customer. A new `SERVICE_STARTED` event is generated for the banker if there are still customers at the service queue.

- After all events are processed, a final `FINISHED` event is added to the event queue to indicate the end of the simulation.

Above is a high-level overview of how events are processed in the bank simulation. It simplifies most cases and provides sufficient detail for this homework. You don't need to consider edge cases beyond what is described here.

6.4.1 Example Run

```
int bankerServiceTimes[3] = {5, 7, 6};
int officerDispatchTimes[2] = {2, 3};

Bank bank(3, bankerServiceTimes, 2, officerDispatchTimes, 10);

bank.AddCustomer(1, 5);
bank.AddCustomer(2, 3);
bank.AddCustomer(3, 4);
bank.AddCustomer(4, 4);
bank.AddCustomer(5, 2);
bank.AddCustomer(6, 1);
```

This code initializes a bank simulation with 3 bankers and 2 security officers, where banker services start at time 10, and each employee has their respective service and dispatch times. It then adds 6 customers with varying priorities to the simulation. If you run the simulation by repeatedly calling `DoSingleEventIteration()`, you will get below output:

```
Time[0]: Customer(1) arrived at the bank.
Time[0]: Customer(2) arrived at the bank.
Time[0]: Customer(3) arrived at the bank.
Time[0]: Customer(4) arrived at the bank.
Time[0]: Customer(5) arrived at the bank.
Time[0]: Customer(6) arrived at the bank.
Time[0]: Customer(1) entered the queue at position 0.
Time[0]: Customer(2) entered the queue at position 1.
Time[0]: Customer(3) entered the queue at position 0.
Time[0]: Customer(4) entered the queue at position 1.
Time[0]: Customer(5) entered the queue at position 0.
Time[0]: Customer(6) entered the queue at position 1.
Time[2]: Customer(1) exited the queue and was dispatched by Officer(0).
Time[3]: Customer(2) exited the queue and was dispatched by Officer(1).
Time[4]: Customer(3) exited the queue and was dispatched by Officer(0).
Time[6]: Customer(4) exited the queue and was dispatched by Officer(1).
Time[6]: Customer(5) exited the queue and was dispatched by Officer(0).
Time[9]: Customer(6) exited the queue and was dispatched by Officer(1).
Time[10]: Banker(0) started serving Customer(6).
Time[10]: Banker(1) started serving Customer(5).
Time[10]: Banker(2) started serving Customer(2).
Time[15]: Service for Customer(6) by Banker(0) is completed.
Time[15]: Banker(0) started serving Customer(4).
Time[16]: Service for Customer(2) by Banker(2) is completed.
Time[16]: Banker(2) started serving Customer(3).
```

```
Time[17]: Service for Customer(5) by Banker(1) is completed.
Time[17]: Banker(1) started serving Customer(1).
Time[20]: Service for Customer(4) by Banker(0) is completed.
Time[22]: Service for Customer(3) by Banker(2) is completed.
Time[24]: Service for Customer(1) by Banker(1) is completed.
Time[24]: Simulation has finished.
```