CENG 213 - Data Structures

Assignment 1: Hospital Emergency Simulation

Due: April 15, 2025 23:59

Objective

Implement a hospital emergency room simulation using custom data structures without using STL. Patients arrive and go to a quick check-up (triage) to assess the severity of their condition. Then, they wait to see a doctor. If a patient waits in a queue longer than a predetermined time (and is at the back of the queue exactly at that time), they become bored and leave.

Required Data Structures

- **LinkedList**: Base for double ended queues and monotonic stack
- **SortedLinkedList**: Base for priority queue (sorted by event time)
- **PriorityQueue**: For discrete event simulation (DES)
- **TieredFCFSQueue**: K-tiered dequeue system for patient urgency levels. A k-tiered FCFS dequeue consists of 'k' sub-dequeues where each sub-queue is a FCFS queue. Items are added to a specific sub-dequeue, but removed from the first non-empty sub-dequeue, proceeding sequentially through the tiers. Each sub-dequeue operates on a first-come, first-served basis, and the removal order prioritizes the sub-dequeues from 0 to k-1.
- **MonotonicStack**: Increasing stack for doctor patient records. An increasing monotonic stack is a data structure that maintains its elements in ascending order. Essentially, when you add a new element:
  - If it's greater than or equal to the top element, you push it onto the stack.
  - If it's smaller, you pop elements from the top until the stack is empty or the top element is smaller than or equal to the new element, then you push the new element.

Class Specifications

1. Event Class

```
#ifndef EVENT_H
#define EVENT_H

#include <iostream>

enum EventType
{
    TriageQueueEntrance, // a patient enters to triage queue
    TriageEntrance, /* a patient enters to triage (triage queue wait is finished) */
    TriageLeave, // a patient leaves triage
    DoctorQueueEntrance,
    DoctorEntrance, /* a patient enters to doctor exemination (doctor queue wait is
        finished) */
    PatientLeaveHospital, /* a patients compelete doctor visit and leaves the
        hospital */
    TriageQueueBoringStart, /* a patient in triage queue gets bored and leaves the
        hospital */
```

```
    DoctorQueueBoringStart /* a patient in doctor queue gets bored and leaves the
        hospital */
};

class Event
{
public:
    int time; // -1, if not applicable
    EventType type;
    int patientId; // -1 if not applicable
    int resourceId; // Triage/Doctor ID, -1 if not applicable
    Event(int t, EventType et, int pid, int rid);
    bool operator<(const Event& other) const;

    Event();
    Event(const Event& other); // copy constructor
    Event& operator=(const Event& other); // assignment operator

    // Overloading the << operator to enable easy printing of Event objects
    // This allows us to use 'std::cout << event;' instead of writing a separate
        print() function.
    /*
    example output:
    [TIME 11] Event Type: 3, Patient Id: 2, Resource Id: 0
    */
    friend std::ostream& operator<<(std::ostream& os, const Event& event);

};

#endif
```

2. LinkedList

```
#ifndef LINKEDLIST_H
#define LINKEDLIST_H

#include <iostream>

class MonotonicStack;
class SortedLinkedList;

// Template Node class
template <typename T>
class Node {
public:
    T data;
    Node* next;
    Node* prev;
    Node(const T& d) {}
};

// Template LinkedList class
template <typename T>
class LinkedList {
```

```
private:
    Node<T>* head;
    Node<T>* tail;

public:
    LinkedList() {}
    ~LinkedList();
    void addFront(const T& data);
    void addBack(const T& data);
    T removeFront();
    T removeBack();
    bool isEmpty() const;
    int length() const;
    T getFront() const;
    T getBack() const;

    friend class SortedLinkedList;
    friend class MonotonicStack;
    friend class FCFSQueue;
    friend class TieredFCFSQueue;
    friend class priorityQueue;

    // Overloading the << operator to enable easy printing of MonotonicStack objects
    /* This allows us to use 'std::cout << s;' instead of writing a separate print()
         function.*/
    friend std::ostream& operator<<(std::ostream& os, const MonotonicStack& event);
};
#endif
```

3. SortedLinkedList

```
#ifndef SORTEDLINKEDLIST_H
#define SORTEDLINKEDLIST_H

#include "LinkedList.h"
#include "Event.h"

class SortedLinkedList {
private:
    LinkedList<Event> list;

public:
    SortedLinkedList();
    ~SortedLinkedList();
    void add(const Event& data);
    Event removeSmallest();
    bool isEmpty() const;
};

#endif
```

4. FCFSQueue (LinkedList based Double-Ended Queue)

```
#ifndef FCFSQUEUE_H
#define FCFSQUEUE_H

#include "LinkedList.h"

class FCFSQueue {
private:
    LinkedList<int> patients; // stores patient IDs

public:
    void enqueue(int patientId);
    int dequeue();
    bool isEmpty() const;
    int getFirst() const;
    int getLast() const;
};

#endif
```

5. PriorityQueue (SortedLinkedList Based)

```
#ifndef PRIORITYQUEUE_H
#define PRIORITYQUEUE_H

#include "SortedLinkedList.h"

class PriorityQueue {
private:
    SortedLinkedList events;

public:
    void enqueue(const Event& e);
    Event dequeue();
    bool isEmpty() const;

    int getFirst() const;
    int getLast() const;
};

#endif
```

6. TieredFCFSQueue

```
#ifndef TIEREDFCFSQUEUE_H
#define TIEREDFCFSQUEUE_H

#include "FCFSQueue.h"

class TieredFCFSQueue {
private:
    FCFSQueue* tiers;
    int numTiers;
```

```
public:
    TieredFCFSQueue(int k);
    ~TieredFCFSQueue();
    void enqueue(int patientId, int tier);
    int dequeue();
    int getFirst() const;
    int getLast() const;
    bool isEmpty() const;
};

#endif
```

7. MonotonicStack

```
#ifndef MONOTONICSTACK_H
#define MONOTONICSTACK_H

#include "LinkedList.h"

class MonotonicStack {
private:
    LinkedList<int> data; // stores patient IDs

public:
    MonotonicStack();
    ~MonotonicStack();
    void push(int patientId);
    int pop();
    int top() const;
    bool isEmpty() const;

    // Overloading the << operator to enable easy printing of MonotonicStack objects
    // This allows us to use `std::cout << s;` instead of writing a separate print()
        function.
    /*
    Example Output:
    {1,3,5,7}
    */
    friend std::ostream& operator<<(std::ostream& os, const MonotonicStack& event);
};

#endif
```

8. DES (Discrete Event Simulation)

```
#ifndef DES_H
#define DES_H

#include "PriorityQueue.h"
#include "TieredFCFSQueue.h"
#include "FCFSQueue.h"
#include "MonotonicStack.h"
```

```
using namespace std;

class DES {
private:
    int numTriages, numDoctors, numTiers;
    int triageDuration, doctorVisitDuration, boringDuration;
    FCFSQueue triageQueue;
    TieredFCFSQueue doctorQueue;
    PriorityQueue eventQueue;
    bool* triageAvailable;
    bool* doctorAvailable;
    MonotonicStack* doctorStacks;
    int *urgencyLevels;


public:
    DES(int numTriages, int numDoctors, int numTiers, int tDuration, int dDuration,
        int bDuration,
    int numPatients, int* urgencyLevels, int *patientArrivalTimes);
    ~DES();
    void run();
    void processEvent(const Event& e);
};

#endif
```

Simulation Rules

1. **Priority of Events in Priority Queue**:
   - Event with smaller ID has higher priority
   - If the rules upside is not applicable, event with smaller patientId has higher priority.
   - If the rules upside is not applicable, event with smaller resourceId has higher priority.
   - If the rules upside is not applicable, event with smaller type (since EventType is an enum, it can be behaved as a number) has higher priority.
2. **Triage Selection**: Always choose smallest available triage ID
3. **Doctor Selection**: Choose smallest available doctor ID
4. **Time Handling**:
   - Triage duration: Given as argument of DES class.
   - Doctor visit: Given as argument of DES class.
   - Patients may get bored in queues: After waiting a predetermined time (given as argument of DES class) in any queue, patients get bored. At the moment they get bored, they check whether they are at the end of the queue. If they are, they leave. If a patient leaves due to boredom, they leave only at the boredom start time, not afterward.
5. **Necessity of Multiple Queue/Stack Types**:
   - FCFS Queue is used for triage entrance queue.
   - Tiered FCFS Queue is used to prioritize patients in the doctors' entrance queue based on urgency.
   - FCFS Queue and Tiered FCFS Queue are double ended queues to handle patients who get bored in the queue and leave.
   - Priority Queue is used for Discrete Event Simulation.

- Doctors push the patient IDs of examined patients onto monotonic stacks to maintain a chronological record.

Small Hints

- At the beginning of the simulation, enqueue triage queue entrance events into eventQueue.
- After processing each event, check whether any patients are waiting in any queue and if there is an available resource (doctor/triage). If so, enqueue corresponding TriageEntrance or DoctorEntrance events.
- Enqueue a new DoctorQueueEntrance event right after processing a TriageLeave event, as each TriageLeave is followed by a DoctorQueueEntrance.
- After processing TriageQueueEntrance/DoctorQueueEntrance events, enqueue TriageQueueBoringStart/DoctorQueueBoringStart events.
- When processing TriageQueueBoringStart and DoctorQueueBoringStart, check whether the corresponding patient is at the end of the queue. If they are at the back, they leave at that time.
- If a patient leaves any queue due to boredom, enqueue a PatientLeaveHospital event.
- Enums can be treated as numerical values.
- Carefully follow "Simulation Rules" and "Implementation Notes."

Implementation Notes

- **Time Handling**: All durations start immediately when resources become available
- **Monotonic Stack**: Must reject pushes that violate increasing order
- **Tier Management**:
  - Patients stay in their assigned tier until dequeued
  - Dequeue always from lowest-numbered non-empty tier

Example Test Scenario

```
#include <iostream>
#include <string>
#include "DES.h"


int main() {
    int patient_arrival_times[2] = {2,2};
    int urgency_levels[2] = {2,1};

    // Create the DES simulation.
    /* Arguments: numTriages, numDoctors, numTiers, triageDuration,
        doctorVisitDuration, boringDuration, numPatients, patients array. */
    DES sim(2, 2, 3, 4, 5, 6, 2, urgency_levels, patient_arrival_times);

    // Run the simulation.
    sim.run();

    return 0;
}
```

Expected Output

```
[TIME 2] Event Type: 0, Patient Id: 0, Resource Id: -1
[TIME 2] Event Type: 1, Patient Id: 0, Resource Id: 0
[TIME 2] Event Type: 0, Patient Id: 1, Resource Id: -1
[TIME 2] Event Type: 1, Patient Id: 1, Resource Id: 1
[TIME 6] Event Type: 2, Patient Id: 0, Resource Id: 0
[TIME 6] Event Type: 3, Patient Id: 0, Resource Id: -1
[TIME 6] Event Type: 4, Patient Id: 0, Resource Id: 0
[TIME 6] Event Type: 2, Patient Id: 1, Resource Id: 1
[TIME 6] Event Type: 3, Patient Id: 1, Resource Id: -1
[TIME 6] Event Type: 4, Patient Id: 1, Resource Id: 1
[TIME 8] Event Type: 6, Patient Id: 0, Resource Id: -1
[TIME 8] Event Type: 6, Patient Id: 1, Resource Id: -1
[TIME 11] Event Type: 5, Patient Id: 0, Resource Id: -1
[TIME 11] Event Type: 5, Patient Id: 1, Resource Id: -1
[TIME 12] Event Type: 7, Patient Id: 0, Resource Id: -1
[TIME 12] Event Type: 7, Patient Id: 1, Resource Id: -1
Simulation finished.
Monotonic Stack of Doctor 0 is {0}
Monotonic Stack of Doctor 1 is {1}
```

Grading Rubric

Component	**Points**
LinkedList, and FCFSQueue (Double Ended Queue Based), and Event	20
SortedLinkedList and PriorityQueue	20
TieredFCFSQueue	10
MonotonicStack	15
Descrete Event Simulation	35

IMPORTANT NOTES:

Do not start your homework before reading these notes!!!

NOTES ABOUT IMPLEMENTATION:

1. You ARE NOT ALLOWED to modify the given parts of the header file.
2. Moreover, you ARE NOT ALLOWED to use any global variables or any global functions.
3. The output message for each operation MUST exactly match the format shown in the output of the example code. Otherwise, you cannot receive points.
4. Using STL (Standard Template Library) is completely forbidden for this assignment.

NOTES ABOUT SUBMISSION:

1. Cheating and plagiarism are strictly forbidden.
2. In this assignment, you must have separate interface and implementation files (i.e., separate `.h` and `.cpp` files) for your class. Your class names MUST BE `LinkedList, SortedLinkedList, FCFSQueue, PriorityQueue, TieredFCFSQueue, MonotonicStack, DES, Event`. Your file names MUST BE `Event.h`, `Event.cpp`, `PriorityQueue.h`, `PriorityQueue.cpp`, `FCFSQueue.h`, `FCFSQueue.cpp` , `TieredFCFSQueue.h`, `TieredFCFSQueue.cpp` , `MonotonicStack.h`, `MonotonicStack.cpp`, `DES.h`, `DES.cpp`, `LinkedList.h`, `SortedLinkedList.h`, `SortedLinkedList.cpp`.

3. The code (`main` function) given above is just an example. We will test your implementation using different scenarios, which will contain different function calls. Thus, do not test your implementation only by using this example code. We recommend you to write your own driver files to make extra tests. However, you MUST NOT submit these test codes (we will use our own test code). In other words, do not submit a file that contains a function called `main`.
4. In the "edit" section of VPL, you can evaluate your code with a few example test cases. You can see whether you pass or fail them. We recommend you to use it. It may help you pinpoint the errors in your codes.
5. This assignment is due by 23:59 on Monday, April 15, 2025. You should upload your work to the corresponding VPL activity in ODTUCLass before the deadline. No hardcopy submission is needed.
6. This homework will be graded by your TA **Burak Ferit Aktan** (aktan@ceng.metu.edu.tr). Thus, you may ask your homework related questions directly to him. There will also be a forum on ODTUClass for questions.