Middle East Technical University - Department of Computer Engineering

CENG 213

Data Structures

Fall '2024-2025

Programming Assignment 1

Due Date: 11 November 2024, Monday, 23:55
Late Submission Policy will be explained below

Objectives

In this programming assignment, you are expected to implement a hospital simulation system. That relies on Discrete Event Simulation (DES).

To implement such an application, you will implement two data structures *FCFSList* (First Come First Serve List (hence the abbreviation FCFS)) and *Sorted List* data structures. Both of these data structures will have a Linked List basis. Specifically, the FCFS list will be implemented as a singly linked list, and the SortedList will be doubly linked.

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

Keywords: C++, Data Structures, Linked List, Discrete Event Simulation

1 FCFSList Implementation (20 pts)

The FCFS data structure used in this assignment follows a singly linked list implementation. It has only two methods; namely **Enqueue** and **Dequeue**. The implementation should follow the **dummy node** implementation. The data layout of the FCFSList class and its helper structures can be seen on Listing 1.

1.1 FCFSNode Struct

FCFSNode structure holds the information about a node in the FCFSList. It is templated because FCFSList is templated.

1.2 FCFSList Class

FCFSList class implements a restricted singly linked list. You can only add elements to the end of the queue, and remove elements from the beginning of the queue.

> The template type of the function is the type of elements that will be stored in this structure. The type "T" should have a **default constructor** and **copy constrcutor** for FCFSQueue to function properly.

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

    FCFSNode(const T&, FCFSNode* node = NULL);
};

template<class T>
class FCFSList
{
    template <typename U>
    friend std::ostream& operator<<(std::ostream&, const FCFSList<U>&);

    private:
    FCFSNode<T>*    head;

    public:
    // Constructors & Destructor
                FCFSList();
                FCFSList(const FCFSList&);
    FCFSList&   operator=(const FCFSList&);
                ~FCFSList();

    //
    void    Enqueue(const T& item);
    T       Dequeue();

};
```

Listing 1: FCFSList and Helper Structures

1.2.1 `FCFSList();`

The default constructor constructs an empty list. Since this class should be implemented with a dummy list implementation, it constructs an empty node, and default constructs the type T in the node.

This function is implemented for you.

1.2.2 `FCFSList(const FCFSList& 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. Set the head to the first element in the list.

1.2.3 `FCFSList& operator=(const FCFSList& 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.

1.2.4 `~ FCFSList();`

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

1.2.5 `void FCFSList<T>::Enqueue(const T& item);`

Enqueues an item to the list. In this case, the "enqueue" method will add the item to the end of the queue.

1.2.6 `T FCFSList<T>::Dequeue();`

Dequeues an item to the list. In this case, the "dequeue" method will remove the item from the beginning of the list the item to the end of the queue.

If there is no element in the list, it should return a default constructed object T.

1.2.7 `friend std::ostream& operator<<(std::ostream&, const FCFSList<U>&);`

std::ostream print method. This function is implemented internally. You can find the implementation in the student pack.

> This function requires the type T to be printable on std::cout. This means there should be a function (in this case, operator overload) in the system with a signature of `std::ostream& operator<<(std::ostream&, const T&);` For this assignment these functions are implemented for you. If you want to test your class with different types, you need to write this overload yourself.

2 Sorted List (50 pts)

In addition to the FCFS List data structure, the hospital simulation system will require a sorted list. This structure provides the core functionality of the hospital DES system.

It will be implemented as a doubly linked list with one crucial restriction. Given elements in the list will be stored as **sorted**. Items in the list will be **sorted in ascending fashion**.

> Similar to the FCFSList the template type "T" have restrictions. It should have an '`operator<`' and '`operator==`' for comparing with an integer and another object of type "T". These are implemented for you for the types that will be used for hospital simulation. If you want to test this class with a custom type, you need to implement these operator overloads.
>
> Please be careful, `node->item < otherNode->item` **will compare the pointers**. You should do `*node->item < *otherNode->item` for comparing the objects.

This means that when you add an element it may reside in an intermediate node where '`node->prev.item <= node.item`' and '`node->next.item > node.item`'. **The equivalences will change slightly depending on the "insert" functionality** . Please check the appropriate function explanations for further information.

Moreover, SortedList holds its node and data in the heap. The user will provide a pointer to an already allocated location in the heap (as `T*`). **This list should not be implemented with a dummy header**. The data layout of the list can be seen in Listing 2.

2.1 SortedListNode Struct

SortedListNode structure defined the node of the sorted list. It stores a pointer of the templated type inside. This pointer points to the allocated T in the heap.

```
template<class T>
struct SortedListNode
{
    T*  item;
    SortedListNode<T>*  next;
    SortedListNode<T>*  prev;
    //
    SortedListNode(const T& item,
                   SortedListNode* nextNode = NULL,
                   SortedListNode* prevNode = NULL);
};

template<class T>
class SortedList
{
    private:
    SortedListNode<T>* head;
    SortedListNode<T>* back;

    public:
    // Constructors & Destructor
                    SortedList();
                    SortedList(const SortedList& list);
    SortedList&     operator=(const SortedList& list);
                    ~SortedList();

    ...
};
```

Listing 2: Sorted List Structures

2.2 SortedList Class

SortedList class has the core implementation. It stores the nodes and provides the interface.

2.2.1 `SortedList<T>::SortedList();`

Only constructor of the SortedList class. It constructs an empty list.

2.2.2 `SortedList<T>::SortedList(const SortedList&);`

Copy the constructor of the SortedList class. This function should do a deep copy. Please do not forget to copy the memory of that `node->item` points to as well.

2.2.3 `SortedList<T>& SortedList<T>::operator=(const SortedList&);`

The copy assignment operator. Do not forget to delete the already existing node chain of the assigned object. This operator should do a deep copy similar to the copy constructor.

2.2.4 `SortedList<T>::~SortedList();`

Destructor of the class. Deletes all the nodes and items from the heap.

2.2.5 `void SortedList<T>::InsertItem(T* i);`

Insert an already allocated item "i" into the appropriate position. item should be '`*node->prev.item <= *item`' and '`*node->next.item > *item`'. This means that i should be the last element of the nodes where `*node->item == *item`.

2.2.6 `void SortedList<T>::InsertItemPrior(T* i);`

Insert an already allocated item "i" into the appropriate position. Unlike the InsertItem function, the added item will be the first element where `*node->item == *item`.

2.2.7 `T* SortedList<T>::RemoveFirstItem();`

This function removes the very first item (which is pointed by the head) and returns it. You transfer ownership of the item "i" to the user, so you should not delete the item "i". However, you should delete the node.

If the list is empty this function should return NULL.

2.2.8 `T* SortedList<T>::RemoveFirstItem(int priority);`

Like the function above, the "RemoveFirstItem" function removes the **first item** that has the priority of "priority".

> Classes (in this case, Patient and Event) will have appropriate comparison operators so that you can compare the object with an integer. Fro example, '`*node->item < priority` and `*node->item == priority`' will work.

Similar to the function above, if no such item with this priority exists the function should return NULL.

2.2.9 `T* SortedList<T>::RemoveLastItem(int priority);`

In this case, the "RemoveLastItem" function removes the **last item** that has the priority of "priority". Otherwise, it has the same functionality compared to "RemoveFirstItem(int)".

2.2.10 `const T* SortedList<T>::FirstItem();`

Returns a pointer to the first item in the list. This does not transfer ownership to the user. This function is only for viewing purposes.

This function is implemented for you.

2.2.11 `const T* SortedList<T>::LastItem();`

This function is implemented for you.

Returns a pointer to the very last item in the list. This does not transfer ownership to the user. This function is only for viewing purposes.

2.2.12 `bool SortedList<T>::IsEmpty() const;`

This function returns true when the list is empty.

This function is implemented for you.

2.2.13 `bool SortedList<T>::ChangePriorityOf(int priority, int newPriority);`

This function should increase or decrease the priority of all items that have a priority "priority". If there are no items with such priority, this function should do nothing.

The relative ordering of the items should be preserved. For example, assume the priority of the pairs below is the second integer. The first integer is there to distinguish the elements with the same priority:

```
(0, 1)->(1, 1)->(2, 2)->(3, 3)->(4, 3)
```

When we call `ChangePriorityOf(1, 2)`, the result should be:

```
(0, 2)->(1, 2)->(2, 2)->(3, 3)->(4, 3)
```

Similarly; when we call `ChangePriorityOf(3, 2)`, the result should be

```
(0, 1)->(1, 1)->(2, 2)->(3, 2)->(4, 2)
```

2.2.14 `void SortedList<T>::Split(SortedList& l0, SortedList& l1, int priority);`

Splits the current list into two lists "l0" and "l1". The current list becomes empty. List "l0" contains items with a priority less than "priority". The rest of the items will be in "l1". The relative ordering of the items should be preserved.

2.2.15 `static SortedList<T> SortedList<T>::Merge(const SortedList& l1, const SortedList& l2);`

This static function will create a new SortedList<T> which will have all the items of "l1" and "l2". Since these lists are accessed via `const` reference. The newly created list should have copies of the items. Similar to the other functions relative ordering of the items should be preserved. In this case, items of "l1" should come **before** the elements of "l1".

For example with these two lists (same pair layout that is described in Section 2.2.13:

```
l0 = (0, 1)->(1, 1)->(2, 2)->(3, 3)->(4, 3)

l1 = (5, 1)->(6, 1)->(7, 2)
```

The result should be:

```
(0, 1)->(1, 1)->(5, 1)->(6, 1)->(2, 2)->(7, 2)->(3, 3)->(4, 3)
```

2.2.16 `friend std::ostream& operator<<(std::ostream&, const SortedList<U>&)`

Similar to the FCFS list this operator overload helps printing the entire linked list. Since this function should not be implemented with a dummy header, It does not skip the very first node but prints all the nodes.

This function is implemented for you but not available for editing. Implementation can be seen in the student pack.

3 Hospital (30 pts)

"Hospital" class is where it all comes together. It will simulate a patient/doctor/registration interaction using a Discrete Event Simulation (DES).

In such a simulation, there is an ordered event list that is sorted by time. Each event marks a change of the state in the system (in this case, Hospital). Since the events will be sorted by time,

we can advance the simulation by processing the very first item in the list. This event may trigger another event which will be added to this event list. Certain events that are specific to our system will be explained shortly.

> For further information about Discrete Event Simulation, you can check Wikipedia. Additionally this youtube video explains and shows a similar example (around 9:30 minutes in)

Meanwhile, the data Layout of the hospital class can be seen in Listing 3.

```
struct EventResult
{
    EventType type;
    int patientIdOrDoctorIndex;
    int timePoint;
};

class Hospital
{
    private:
    // Lifetime of patients are handled by these queues
    // Start on the reg queue,
    // Move to the exam queue
    // After the exam is done delete the patient
    FCFSList<Patient*>   regQueue;
    SortedList<Patient>  examQueue;
    // Priority Queue for events
    SortedList<Event*>   eventQueue;
    // Doctors in the hospital
    Doctor      doctors[2];
    Patient*    patientsInExam[2];

    int         currentTime;
    int         registrationTime;
    int         doctorCheckTime;

    ...

};
```

Listing 3: Hospital Class

3.1 Doctor, Patient, Event Classes

Our hospital has certain actors namely Patients and Doctors. The hospital has two doctors with differentiating examination times. In addition, the hospital has a "perfect" registration desk, meaning it can process infinitely many patients at the same time.

All these actors are already defined in Actor.h file(Except for one member function which will be explained shortly). Most of these classes only define properties, get/set functions, and print functionality. Additionally, some of these classes; namely Event and Patient, will have proper operator overloads and constructor/destructors to comply with the requirements of FCFSList and SortedList classes.

`Actor.h` file also defines the event class that will be used for DES. Events can have four types (see

Listing 4).

```
enum EventType
{
    RegistrationQueueEntrance,
    RegistrationEnd,
    //
    DoctorQueueCheck,
    ExaminationEnd,
    //
    InvalidEvent
};
```

Listing 4: Hospital Class

Almost all of the functionality of these functions is implemented for you. Except for the Doctor's member function `ExamTimeOf`.

3.1.1 `Doctor::ExamTimeOf(const Patient& patient) const);`

Given a patient, this function should return the examination time of that specific Doctor. When the patient's priority number is n, with maximum priority N and the doctor's baseline examination time is t function should process the patient's examination time r as follows:

$$r = (N - n + 1) \times t \tag{1}$$

This means that patients with higher priority probably have a serious condition that takes more time. Priority of a patient n is $n \in [0, N]$. N is already declared as `MAX_PATIENT_PRIORTIY` in `Actors.h` file.

3.2 Discrete Event Simulation and Events

As described above DES is a series of timestamped events that may trigger new events. Events can be:

- `RegistrationQueueEntrance`: This event will be added to the system when a new patient is added. The timestamp of this event will be '`currentTime`'. When this event is added to the system, the patient will be added to the `regQueue`.
- `RegistrationEnd`: This event will be added to the system when a `RegistrationQueueEntrance` event is processed. The timestamp of this event will be '`currentTime + registrationTime`'. When this event occurs, the patient will be removed from `regQueue` and added to the `examQueue`.
- `DoctorQueueCheck`: This is a special event that is related to Doctors. Doctors will periodically check the `examQueue`, if it is not empty they will transfer that patient to the `patientsInExam` and add `ExaminationEnd` to the event system. The timestamp of this event will be '`currentTime + doctor.ExamTimeOf(patient)`'. **If `examQueue` is empty, the system will trigger the `DoctorQueueCheck` event again for the checking doctor.** The timestamp of this new `DoctorQueueCheck` event will have '`currentTime + doctorCheckTime`'.
- `ExaminationEnd`: The examined patient will be removed from the system. Then the corresponding doctor of that patient will search new patient similar to the `DoctorQueueCheck`. If he/she finds a patient, it will immediately start examining that patient. This means that the patient will be moved from the `examQueue` to the corresponding slot `patientInExam`. Similarly, if `examList` is empty, a `DoctorQueueCheck` event will be added to the system.

Initially hospital class will issue two events. `DoctorQueueCheck` for both Doctors. Events can either be constructed with a patient pointer or with a doctor index. Events `RegistrationQueueEntrance, RegistrationEnd, ExaminationEnd` should be constructed with a patient pointer and `DoctorQueueCheck` event should be constructed with the doctorIndex. Doctor index is the index of the doctor in the 2-element array `doctors`.

When patients move from `regQueue` to `examQueue`. Patients will be reordered according to their `priority`. When two patients have the same priority, newly added patients should not come after the old patients with the same priority. Priorities are sorted in ascending order as well meaning priority zero will have the highest precedence.

> This is why the `SortedList` class holds its elements in the heap. Most of the events will require a `Patient` reference (in this case reference is handled with a pointer) and the patient will move around between data structures. When a patient moves, this will prevent invalidation of the patient pointers inside the events.

3.3 Hospital Class

3.3.1 `Hospital::Hospital(const Doctor& d0, const Doctor& d1, int registrationTime, int doctorCheckTime);`

Issues two doctor check events, and initializes `registrationTime, doctorCheckTime` properties of the class. Additionally, it sets the `currentTime` to zero.

This function is implemented for you.

3.3.2 `Hospital::~Hospital();`

Most of the deletions of Patients/Evens are handled automatically. However, **patients that are in the `FCFSList<Patient*>` won't be deleted by the destructor of `FCFSList<Patient*>`.** FCFSList will only delete the nodes but not the patients. You should manually delete patients that remained in this list.

> You may want to implement `IsEmpty()` member to the FCFSList.

3.3.3 `void Hospital::AddPatient(int id, int priority);`

This function will create a new patient with `id` and `priority`. A `RegistrationQueueEntrance` event will be added to the system. Newly created patients should be added to the `regQueue`.

3.3.4 `EventResult Hospital::DoSingleEventIteration();`

This is where DES happens. A single item will be removed from the event list, and processed according to Section 3.2. `currentTime` should be updated with the time of the event. Finally, `EventResult` will be returned to the user. Testing functions will print this `EventResult` object. This struct has the information of the currently processed event, not the newly issued event (if any). The result will have:

- `type` should be the type of the processed event
- `timePoint` should have the updated current time
- `patientIdOrDoctorIndex` should either have index the of the doctor when `type = DoctorQueueCheck` or patient id otherwise

Regulations

1. **Programming Language:** You will use C++.
2. Standard Template Library is **not** allowed at all!
3. Using external libraries other than those already included is not allowed.
4. Changing or modifying already implemented functions is not allowed
5. You can add any private member functions unless it is explicitly stated that you should not.
6. **Late Submission Policy:** Each student receives 5 late days for the entire semester. You may use late days on programming assignments, and each allows you to submit up to 24 hours late without penalty. For example, if an assignment is due on Thursday at 11:30pm, you could use 2 late days to submit it on Saturday by 11:30pm with no penalty. Once a student has used up all their late days, each successive day that an assignment is late will result in a loss of 5% on that assignment.

   No assignment may be submitted more than 3 days (72 hours) late without permission from the course instructor. In other words, this means there is a practical upper limit of 3 late days usable per assignment. If unusual circumstances truly beyond your control prevent you from submitting an assignment, you should discuss this with the course staff as soon as possible. If you contact us well in advance of the deadline, we may be able to show more flexibility in some cases.
7. **Cheating:** We have a zero tolerance policy for cheating. In case of cheating, all parts involved (source(s) and receiver(s)) get zero. People involved in cheating will be punished according to the university regulations. Remember that students of this course are bound to the code of honor and its violation is subject to severe punishment.
8. **Newsgroup:** You must follow the Forum (`odtuclass.metu.edu.tr`) for discussions and possible updates on a daily basis.

Submission

- Submission will be done via Class, (`class.ceng.metu.edu.tr`).
- Don't write a "main" function in any of your source files. It will clash with the test case(s) main function and your code will not compile.
- A test environment will be ready in Class :
  - You can submit your source files to Class and test your work with a subset of evaluation inputs and outputs.
  - Additional test cases will be used for the evaluation of your final grade. So, your actual grades may be different than the ones you get in Class.
  - Only the last submission before the deadline will be graded.