Cairo University, Faculty of Engineering
Computer Engineering Department
Data Structures and Algorithms

Project Phases, deliverables, and Guidelines

Phase 1.1: Selecting the appropriate data structures that should be used to store different lists in the project.
Phase 1.2: Implementing and testing the selected data structures to make sure they work as required to proceed to phase 2.
Phase 2: Implementing the algorithms related to the project logic to simulate different scenarios and generate the output file containing system operation statistics.

Phase 1

This phase is concerned with selecting the appropriate data structures **for the whole project,** not just for Phase 1.

Phase 1.1: Choose the data structures.

- Decide the **appropriate data structures** you should use to store each list in the project. For example, what DS is suitable to store early patients, U-therapy waiting patients, etc.

Deliverables: You should deliver a report in the following updated format.
You are allowed to use the following **data structures ONLY:**

- Stack
- Queue
- Pri-queue
- or any class derived from them having the needed additional functions

Phase 1.2: Implement & Test the data structures.

ALL data structures in the project MUST make use of the Stack, Queue, or Priority Queue classes **given in the labs** or classes derived from them. Choose the appropriate DS for a lower complexity and overall time to frequent operations.

NOT ALLOWED:

- **Global variables.**
- **Use of friendship.**
- **C++ STL.**

SHARE, MOVE, Don't Copy

Don't allocate the same unit more than once in the program. So all lists should be **lists of pointers** so that when a unit moves from one list to another, you just make the new list point to it, not to take a new copy of its object.

Phase 1.2 Code

You have to finalize the following in phase 1.2.

1. *Full data members of classes (See Appendix A below)*
2. *Full implementation for **ALL** data structures that you will use in the project.*
3. ***File loading function** which reads from the mentioned input file format.*
4. *Simple **Simulator** function for Phase 1. The main purpose of this function is to test different lists and make sure all operations on them are working properly. It is a demo function for testing.*

The Simulation function performs the following operations:

1. *Perform any needed initializations.*
2. *Call the file loading function to load all data and save ALL system data and load all patients to the **ALL-Patients** List.*
3. *At each **timestep,***
   a. *Check the ALL-Patients list and if patient**(s)** arrive at the current timestep, move them to either Early list (if VT < PT), Late list (if VT > PT), or One of the waiting lists (if VT == PT) as follows.*
   *The following procedure is called **RandomWaiting**:*
   *Generate a random number **N** from 0 to 100. If **N** < 33, then E-Waiting, else if **N** < 66, then U-Waiting, else X-Waiting.*
   b. *Generate a random number **X** from 0 to 100 then:*
      i. If ***X***<10, move next patient from Early to a *RandomWaiting* list.
      ii. If 10<=***X***<20, move next patient from Late to a *RandomWaiting* list and insert him using **PT+penalty** time.
      iii. If 20<=***X***<40, move **2** next patients from a *RandomWaiting* to In-treatment list.
      iv. If 40<=***X***<50, move next patient from In-treatment to a *RandomWaiting* list.
      v. If 50<=***X***<60, move next patient from In-treatment to **Finish**.
      vi. If 60<=***X***<70, move random patient from *X-Waiting* to **Finish**, to simulate the **cancel** process.
      vii. If 70<=***X***<80, choose random patient from *Early* list to the appropriate list to perform an accepted **reschedule** process.
      viii. **Note:** In each *RandomWaiting,* a new random ***N*** is generated.
4. *Print all applicable info to console as in the "**Program Interface**" section.*
5. ***End the simulation when ALL patients are moved to the finish list.***

Notes:

- *In the above simulation function, the "next" patient of any list means the patient that has the turn to go out of the list (e.g., pop if stack, etc.).*
- *NO actual **patient-treatment assignment logic** is required in Phase 1. This includes the order in both types of patients and handling durations.*
- *NO actual **treatment resources availability check** is required in Phase 1, but the full implementation of their lists and list objects is required.*
- *NO **output file** in Phase1.*

Phase1 Video

You should record a short video (less than 10 minutes) showing a demo for Phase1. In the video, you should do the following:

- Give a **very** brief explanation for the types of data structures lists you have chosen for each list in the project. (max 2 min)
- Show an input file with at least 30 patients of non-trivial scheduling requirements
- Run the code and explain (for a part of the input) how patients are moved from one list to another. Map your explanation to the data loaded from input file

Phase 1.2 Deliverables:

Each team is required to submit:

1. A text file named ***ID.txt*** containing team members' names, IDs, and emails.
2. **Phase1 code** [Do not include executable files].
3. One sample , non-trivial, input file (test case).
4. **Phase1 Video (upload to Gdrive, Give access to everyone, share link)**

Phase 2

In this phase, you should extend the code of phase 1 to build the full application, produce the final output file, and support the different operation modes.

Load must be distributed equally between team members.

Phase 2 Deliverables:

Each team is required to deliver the following:

1. A text file named ***ID.txt*** containing team members' names, IDs, and emails.
2. **Team Workload document**. Showing load of each member.
3. **Final project code** [Do not include executable files].
4. **Six <u>comprehensive</u> sample input files (test cases) and their output files. Sample input files must cover simple to complex scenarios.**
   The sample input files must <u>clearly</u> show at least the following scenarios:

   - The normal flow and different durations of elements in each list.
   - Early and Late patients with different penalties.
   - Available device resources, becoming unavailable, then available.
   - Room capacity > 1 and going available, full, and available again.
   - How the treatment order is handled in N-patients and R-patients.
   - Accepted and not accepted cancel operations.
   - Accepted rescheduling operation.

   Note: In **test_description.txt**, write which input files test which scenarios and specify the input file testing most scenarios.

Project Evaluation

- ✔ **Project Grade:** 40% (phase 1.1, phase 1.2) and 60% (Phase 2)
- ✔ **Work Load:** Each member must be responsible for writing some project modules and answer some questions showing understanding of the logic and implementation of his part. The workload of team members must be equal.
- ✔ **Individual grading for EACH item (% is from Phase 2 Grade) :**
  - **[60%]** Code understanding and quality
  - **[20%]** Code run
  - **[20%]** Integrating work with members who finished or nearly finished.
- ✔ **Team Grade:** If there are no ***major*** problems in the team (a big part not implemented, integration problems, etc.), load is divided equally, and all members understand their code, then we give the same ***team grade*** to all members (without performing a detailed individual grading).
- ✔ You should **inform the TAs** before the deadline **with sufficient time (some weeks before)** if any problems happen between team members.
- ✔ If one or more members don't do their work, the other members will NOT be affected **as long as** (1) they inform us beforehand about the problem, (2) all other parts are integrated, AND(3) *their part can be easily tested*.

Penalties:

The cheating penalty (giving or taking) any project part from any source is taking ZERO in the project AND also taking MINUS FIVE (-5) or more from other class work grades, so it is better to deliver it incomplete than cheated.

It is totally your responsibility to keep your code away from others.

Appendix A – Guidelines for Project Code

The classes of the project are described below. In addition, you need **lists of appropriate data structures** to hold patients, required treatments, available resources, etc. You may add extra data members and functionality as needed.

Resource

A class to represent the resource needed for a treatment. This includes electro/ultrasound devices or a room. It should have a resource type and ID.

Treatment

Base class for all treatment types required by the patient. Each treatment has a duration, an assignment Time (the time it is assigned to a resource), and a pointer to the assigned resource. It should have 2 virtual functions.

- **CanAssign:** checks if the required resource is available or not
- **MovetoWait:** moves the patient to the appropriate waiting list. It should call the appropriate "**AddToWait**" function from the scheduler class.

Derive the treatment types from this class and override its virtual functions.

Patient

This class should have data members: ID, PT, VT, required treatments **list**, etc.

Scheduler

This class is the maestro class that manages the system. It should hold **objects** of appropriate **lists** of patients and resources.

It should have member functions to:

- ➔ At program startup, open the input file and load patients to the "ALL" list
- ➔ A main Simulation function that:
  At each timestep, (**and among other things**) ⇒ This is a Phase 2 function.
  - ◆ Checks "ALL" list and if a patient has arrived, move him to Early/Late/Wait
  - ◆ Functions AddToWait_U to add a patient to U-Wait list and similar functions to add to E-Wait and X-Wait. These functions are to be called by *Treatment::MovetoWait* function mentioned above
  - ◆ Assign patients to resources
  - ◆ When a patient is done with a treatment, move him to Wait/Finish list
  - ◆ Collect statistics that are needed to create output file
  - ◆ Calls UI class functions to print details on the output screen
- ➔ Produce the output file at the end of simulation ⇒ This is a Phase 2 function.

UI

It should be responsible for reading any inputs from the user and printing any information on the **console**. It contains the input and output functions that you should use to show the status of the system at the end of each timestep.

How to print the output screen described in "Program Interface" section.

- In each data structure (DS) you will use (e.g. queue, priority queue, etc.) , you should add a **print member function** that loops on its data and prints it formatted as shown in the Program Interface section. You are also allowed to add a **counter data member and GetCount()** in different DS to be able to print their count to the O/P window. These are the ONLY allowed edits in the stack, queue, and priority queue classes of the lab code.
- You should also overload << operator for ***patient**** (and any other classes as needed) to make it print their info. You may use friendship here.

The Role of UI class when printing the o/p screen.

Then in UI class, add a member function that takes as parameters pointers/references to these lists. This function prints the output screen in the required format. When it needs to print the lists passed to it, it should just call the print function of each list.

Another important role of the UI class is to provide functions to be called to read the interface mode, input and output file **names.**

The UI class is **NOT** responsible for reading/writing the input/output files