Assignment 4

COMP 2401

Date: November 9, 2021

Due: on November 22, 2021, before 23:55

Submission: Electronic submission on Brightspace.

1 Objectives

a. Dynamic memory handling (memory allocation and releasing).

b. Linked Lists

c. Working with pointers

d. Function pointers

e. Priority queues

f. Usage of the make utility

g. Recursion

2 Assignment points

Assignment Total points: 100

Part I (120 pts)

1. **Submission – 10 pts**
2. **Required coding - 85 pts**
3. **No memory leaks – 10 pts**
4. **Makefile Creation – 5 pts**

Bonus (45 pts)

1. **Saving to an Ascii file – 8 pts**
2. **Saving to a Binary file – 8 pts**
3. **Reversing a list - 14 pts**
4. **Releasing patients with low priority – 10 pts**
5. **Print the list of patients in reverse order - 5 pts**

3 Submission (10 points)

- Submission must be in Brightspace by the due date.
- (1 pt.) Submit a single tar file with all the c and h files Readme.txt file.
- (4 pts.) Submit a Readme.txt file explaining

  - Purpose of software
  - Revision information - Who the developer is and the development date
  - How the software is organized (partitioned into files)
  - Instruction on how to compile the program (give an example)
  - Any issues/limitations problem that the user must be aware of
  - Instructions explaining how to use the software
- (2 pts.) All pointers are initialized to NULL
- All pointers when not used are set to NULL
- (4 pts.) Code is readable and commented
- If you are coding any of the bonus functions, then clearly document which function were coded in the Readme file. Otherwise, these functions will not be teste

4 Grading:

You will be graded with respect to what you have submitted. Namely, you earn points for working code and functionality.

The grading will look at how the functions are coded, whether functions meet the required functionality, memory allocation and de-allocation, setting pointers to NULL as needed, no memory leaks, etc.

4.1 Testing

1. Do not change any of the function names – this may result in not receiving points for the functions ore receiving 0 for the assignment
2. Do not change any of the file names – this may result in receiving 0 for your assignment
3. You are provided with a function to populate the patients' records. Do not change the function. It is recommended that you use this function.
4. A short test program was created (see file emergencyRoomMain.c). You can use it to test your code. I will provide an executable code that will correspond to that file or another file.

5 Background

Your team was tasked to develop APIs (application programming interface functions) for managing patients arriving at a hospital emergency department (ER). The system assists the doctors and nurses in the ER to decide the next patient who will be treated. It does so by reviewing the severity of the illness and the time that the patient came to the ER.

Each patient that arrives at the hospital provides his name (first name, last name, and id) to the triage nurse, who also assesses the patient and then determines the medical problem and the priority level (i.e., how severe the medical issue is). Once accepted into the hospital the patient is tagged with an arrival order.

6 API Design

The design calls for the following structure

```
#define NAME_LENGTH 16
```

```
#define PROBLEM_LENGTH 24

typedef struct patientInfo {
      unsigned int id;             // patient id
      unsigned int arrivalTime;    // arrival order of patients
      unsigned char priorityLevel; // priority level
      char medicalProblem[PROBLEM_LENGTH];    // a description of the medical problem
      char firstName[NAME_LENGTH];      // patients first name
      char familyName[NAME_LENGTH];     // patients family name
} PatientInfo;
```

Patient's fields

firstName – a string that holds the patient first name

familyName – a string that holds the patient's family name

priorityLevel – determines the severity of the medical issue. A priority level of 10 is the most severe while priority level 1 is the least severe.

arrivalTime – the time in which the patients are admitted to the hospital. This value is assigned by the system for each patient as they are entered the system. The system is incrementing this number by 1 for each patient.

medicalProblem – a short description of the medical problem.

The system designers decided to implement the patients' management system using a singly linked list.

Each node in the linked list has the following structure

```
typedef struct listNode {
      PatientInfo *patient;
      struct listNode *next;
} ListNode;
```

Provided Files

linked_list_hospital.h – this file contains the definition of the linked link functions

linked_list_hospital.c – this file contains the code of the linked list functions

patient.h – this file contains the definitions of the patient functions

patient.c – this file contains the implementation of the patients' functions.

emergencyRoomMain.c – an example code for testing the functionality of the APIs. Note that this is not a comprehensive set. A more comprehensive test program may be provided later.

Makefile – a partial makefile for generating the code

Compiling the test program:

To test program, you will need to compile and link the test program with your code. For example use

```
gcc -g emergencyRoomMain.c linked_list_hospital.c patient.c
```

6.1 *Coding Instructions:*

1. Add comments to code – as provided in the slides given in class
2. No usage of global variables. All data must be passed or received via function parameters.
3. Each node in the linked list must be allocated independently of other nodes.
4. Test your program with valgrind to ensure that no memory leakage occurs.
5. This assignment will most likely be tested automatically using test program(s). Namely, your file linked_list_hospital.c and linked_list_hospital.h will be linked together with test programs. Therefore, you cannot deviate from the function definitions. You are allowed however, to add additional functions (helper functions) to the file linked_list_hospital.c. These functions will not be tested.
6. Make sure that all pointers are initialized to NULL or set to NULL as needed.
7. All functions that insert a node into the list must copy the input data onto the newly created node.
8. All functions that search for a node or delete a node must copy the data from the node to the output parameters

Suggestions –

1. Test functions – most functions will be quite short 2-10 lines of code. As you code design and implement test functions to ensure that your code covers all cases (e.g., empty list, i.e., head == NULL).
2. Add other functions, as needed, to simplify your code.
3. Do not start coding immediately – write the design/pseudo code on paper, then write it as comments in the function and last fill in the "blanks" between the comment with lines of code.

6.2 *Part I Simple Operation of the system (85 points)*

In this part of the assignment, you will code several basic functions used to manipulate a linked list. The files linked_list_hospital.c and linked_list_hospital.h contain the functions and their prototypes. You will have to complete the body of the function. The description of each function in the linked_list_hospital.c file provides detailed information about the functionality of each function including input and output parameters and return value.

Code the following functions

1. Insertion Functions (10 points)

1.1. (5 points) Insert to list as the first node
Purpose - create a new node and add it to the list as the first element
Function declaration:
`ListNode *insertToList(ListNode **head, PatientInfo *patient);`

1.2. (5 points) Insert to list after a given record
Purpose - create a new node and add it to the list after the given record

```
Function declaration:
ListNode *insertAfter(ListNode *node, PatientInfo *patient);
```

2. **Search Functions (15 points)**

2.1. (5 points) Search by priority
Purpose - search for the first node with the matching priority

```
Function declaration:
ListNode * searchFirstPatientByPriority(ListNode *head,
                                    unsigned char priority, PatientInfo *patient);
```

2.2. (10 points) Search by matching criteria (5 points)
Purpose - search for the first node with the matching criteria. A match will be determined by a function pointer parameter - findPatient. The function findPatient() will return a 0 if a match was found.

```
Function declaration:
ListNode * searchNextPatient(ListNode *head, int (*findPatient)(PatientInfo *), PatientInfo
*patient)
```

3. **(45 points) Delete Functions**

3.1. (5 points) Delete
Purpose - delete the first patient from the list

```
Function declaration:
int deleteFromList(ListNode **head, PatientInfo *patient);
```

3.2. (10 points) Search for the next patient to be treated by priority
Purpose – searches for the next patient to be treated. It searches for first patient with a matching priority, deletes it from the list and returns its information

```
Function declaration:
int retrieveNextPatientByPriority(ListNode **head, unsigned char priority,
                       PatientInfo *patient)
```

3.3. (25 points) Search for the patient with the highest priority criteria to be treated
Purpose - Retrieve the patient with the highest criteria to be treated. The maximum criteria are determined by a comparePatients function. Once the patient is found it should be deleted from the list and its information is returned.

```
Function declaration:
int retrieveNextPatientByCriteria(ListNode **head, int (*comparePatients)(PatientInfo *p1,
PatientInfo *p2, int currentTime), int currentTime, PatientInfo *patient);
```

3.4. (5 points) Delete the list and all the nodes
Purpose - deletes all nodes from the list

```
Function declaration:
void deleteList(PersonalInfo **head)
```

4. (5 points) Printing Functions

4.1. (0 points) Print a patient

Purpose – This function is already coded. Do not change it.

Function declaration:
```
void printPatient(PatientInfo *node);
```

4.2. (5 points) Print the list

Purpose - prints all the patient records in the list. The function accepts a function pointer to print the patient. Note you can use the existing function printPatient() as input.

Function declaration:
```
void printList(ListNode *head, void (*myPrint)(PatientInfo *patient));
```

5. (10 points) Counting Functions

5.1. (5 points) Count number of registered patients

Purpose - counts the number of patients that are waiting to be treated

Function declaration:
```
int numPatientsInEmergency(ListNode *head);
```

5.2. (5 points) Count patient with specific priority

Purpose - counts the number of patients with a matching priority

Function declaration:
```
int countPatients(ListNode *head, unsigned char priority)
```

6. (5 points) Make utility

6.1. (5 points) Complete the file Makefile so that the compilation can be carried out using the make utility

6.3 Part II Bonus questions (45 points)

7. (10 points)

7.1. (10 points) Remove patients with low priority

Purpose - removes all patients with a priority less than or equal to the given priority. Namely, remove all patients with priorityLevel <= priority.

Function declaration:
```
int releasePatients(ListNode **head, unsigned char priority);
```

8. (16 points) Backup

8.1. (8 points) Create a file backup in ASCII format

Purpose – store all the patients records in the file by printing out the information i.e., in ASCII readable format.

Function declaration:
```
int createAsciiBackup(char *fileName, ListNode *head);
```

8.2. (8 points) Create a file backup in binary format

Purpose – store all the patients records in the file in their binary format.

Function declaration:
```
int createBinaryBackup(char *fileName, ListNode *head)
```

9. (14 points) Reversing the list

9.1. (14 points) Reverse a list

Purpose – Reverse the list such that the last node in the list becomes the first node and the first node in the list becomes the last node.

Restrictions:
1. This operation must be achieved using recursive calls
2. No memory is to be allocated or released at any time during the execution of the function

Function declaration:
```
ListNode *reverseList(ListNode *head);
```

10. (5 points) Print list in reverse order

Purpose: prints all the patients' records in the list in reverse order using recursion

input
head - the head of the list

Function declaration:
```
void printListReverse(ListNode *head, void (*myPrint)(PatientInfo *patient));
```