CS 141 : Program Design 2
Spring 2021

UIC COMPUTER SCIENCE

Project 05 : Linked-lists and Priority Queues (doc v1.2)

Assignment: C++ libraries for linked-list and priority queue
Evaluation: Gradescope followed by manual review
Policy: Individual work only
Complete By:
Part 01 (40pts): Friday April 9th @ 11:59pm CDT, late day Sat 4/10 @ 11:59pm
Part 02 (60pts): Sunday April 11th @ 11:59pm CDT, late day Mon 4/12 @ 11:59pm

Pre-requisites: HW 12-13, Lab 12, Panopto recordings 28-32

Overview

The assignment has 2 parts. In part 01, you'll implement a set of functions to create and manipulate one-way linked-lists. In part 02, you'll implement an additional set of functions to implement a priority queue using your linked-list functions.

Part 01: One-way linked-lists

In part 01 you're going to implement the functions defined in a provided "list.h" header file. These functions implement a one-way linked-list, and are similar to the functions you used in lab 11. The header file is listed in Appendix A of this document, or can be downloaded from the Codio project "cs141-project05".

Your implementation should reside in the file "list.cpp". The functions are designed to support the priority queue you'll be implementing in part 02. That's why each node of the linked-list is designed to store a **NodeData** struct containing an **ID** and a **Priority**. You are required to implement the functions as designed, using the structs as given; you cannot change the structs nor the functions. You can *add* additional functions if you want, but you cannot change what is provided. When implementing the functions, do not use any built-in C++ data structures or algorithms --- implement all functions yourself. No global variables (i.e. no variables declared outside of your functions). You will be limited to a total of **10** Gradescope submissions.

To get started, create a new file "list.cpp" and #include "list.h". Add a header comment, and other header files that you need to #include. Now one by one, copy a function's comment and header from "list.h" over to "list.cpp". Implement the function, and then test using a separate "main.cpp" to call the function. Your testing code in main() can be a series of function calls, or you can write more of a test program like we did in lab 11. When you compile your program, be sure to include both "main.cpp" and "list.cpp" like this:

```
g++ -g -Wall main.cpp list.cpp -o main
```

You'll notice that some of the functions *throw an exception* when an error condition occurs; exceptions are the error handling technique adopted by modern programming languages like C++, C#, Java and Python. The idea is that an exception must be caught (and hopefully fixed) otherwise the program ends. For example, the **retrieve_from_list(L, position)** function is supposed to return the NodeData from the node at the specified position. What should the function return if position is invalid? There's no good return value that works in all cases. Instead, what modern languages do is have the function **throw** an *invalid_argument* exception like this:

```
if (the position is invalid)
{
   throw invalid_argument("retrieve_from_list: invalid position");
}
```

How do you test to make sure your function is throwing the exception properly? By using **try-catch** in your testing code as follows:

```
try
{
   retrieve_from_list(L, 1000000);  // definitely an invalid position

   cout << "**Error: no exception was thrown, why not?!" << endl;
}
catch (invalid_argument& e)
{
   cout << "Correct, retrieve_from_list failed, msg=" << e.what() << endl;
}
catch (...)
{
   cout << "**Error: wrong type of exception was thrown?!" << endl;
}
```

You should use a similar technique to test any function that can possibly throw an exception. [Note: exceptions are expensive to implement and should only be used for *exceptional* circumstances. Exceptions are not meant to replace the normal function return process. In other words, exceptions are used as little as possible when programming.]

Part 01: submission, early/late policy, and grading

Submission: submit your "list.h" and "list.cpp" files to Gradescope under "Project 05: part 01". You will be limited to a total of **10** submissions, so you must test locally. If necessary, you may submit up to 24 hours late for a 10% penalty (5 additional submissions will be provided); if you submit 24+ hours before the deadline, you can earn a 10% "early bonus" to offset a future late penalty.

Your score on part 01 is based solely on correctness, as determined by Gradescope; correctness will be worth 40 points out of the total 100 project points. Note that the TAs may manually review to ensure adherence to project restrictions, and your score could change as a result.

In terms of grading, note that we expect all submissions to compile, run, and pass at least some of the test cases; do not expect partial credit with regards to correctness unless your submission compiles and runs. By default, we grade your **last** submission. Gradescope keeps a complete submission history, so you can **activate** an earlier submission if you want us to grade a different one; this must be done before the due date. We assume *every* submission on your Gradescope account is your own work; do not submit someone else's work for any reason, otherwise it is considered academic misconduct (how else did you end up with someone else's program?).

Part 02: priority queues

A **priority queue** is a data structure that maintains elements in order by priority. A simple example is a line where those 65 and older are allowed to move to the front of the line. Here in part 02 the priority queue will be more general in that it stores (ID, Priority) pairs in a one-way linked-list, in priority order low to high. For example, here are 4 pairs in priority order:

```
(45,7) (22,15) (4,15) (18,99)
```

The pairs will be stored in a one-way linked-list, using the functions you created in part 01. Note that IDs are unique --- no pairs will have the same ID. However, different pairs can have the same priority. If two pairs have the same priority, they are ordered based on arrival: the pair that was inserted first is prioritized over the pair inserted second. In the example above, note that IDs 22 and 4 have the same priority.

In part 02 you're going to implement the functions defined in a provided "pqueue.h" header file. These functions implement our notion of a priority queue. The header file is listed in Appendix B of this document, or can be downloaded from the Codio project "cs141-project05".

Your implementation should reside in the file "pqueue.cpp". You must implement the "pq" functions as given, you cannot change the design of these functions. Likewise, you cannot change the design of the structs or functions that were required in part 01. In other words, the same rules apply here in part 02 that applied in part 01: you can *add* additional functions if you want, but you cannot change what is provided. When implementing the functions, do not use any built-in C++ data structures or algorithms --- implement all functions yourself. No global variables (i.e. no variables declared outside of your functions). You will be limited to a total of **10** Gradescope submissions.

To get started, create a new file "pqueue.cpp" and #include "pqueue.h". Add a header comment, and other header files that you need to #include. Now one by one, copy a function's comment and header from "pqueue.h" over to "pqueue.cpp". You have three choices when implementing the function: (1) call one or more of the existing list functions from part 01, (2) implement the function from scratch by working with the linked-list directly, or (3) a combination of #1 and #2. Most of the "pq" functions can be implemented using options 1 or 3; in other words, take advantage of the existing list functions as much as possible.

As in part 01, you'll need to test your work using a separate "main.cpp". When you compile your program, be sure to include all three C++ files:

```
g++ -g -Wall main.cpp pqueue.cpp list.cpp -o main
```

Part 02: submission, early/late policy, and grading

Submission: submit your "list" (.h, .cpp) and "pqueue"(.h, .cpp) files to Gradescope under "Project 05: part 02". You will be limited to a total of **10** submissions, so you must test locally. If necessary, you may submit up to 24 hours late for a 10% penalty (5 additional submissions will be provided); if you submit 24+ hours before the deadline, you can earn a 10% "early bonus" to offset a future late penalty.

Your score on part 02 is based on two factors: (1) correctness as determined by Gradescope, and (2) manual review of "pqueue.cpp" (and potentially "list.cpp") for commenting, readability (consistent indentation and whitespace), appropriate variable names, use of functions, and adherence to project restrictions. Deductions are made based on lack of commenting / readability, bad variable names, and breaking project restrictions.

In terms of grading, note that we expect all submissions to compile, run, and pass at least some of the test cases; do not expect partial credit with regards to correctness unless your submission compiles and runs. By default, we grade your **last** submission. Gradescope keeps a complete submission history, so you can **activate** an earlier submission if you want us to grade a different one; this must be done before the due date. We assume *every* submission on your Gradescope account is your own work; do not submit someone else's work for any reason, otherwise it is considered academic misconduct (how else did you end up with someone else's program?).

Academic Honesty

In this assignment, all work submitted for grading *must* be done individually. While we encourage you to talk to your peers and learn from them, this interaction must be superficial with regards to all work submitted for grading. This means you *cannot* work in teams, you cannot work side-by-side, you cannot submit someone else's work (partial or complete) as your own. The University's policy is available here:

https://dos.uic.edu/conductforstudents.shtml

In particular, note that you are guilty of academic dishonesty if you **extend or receive any kind of unauthorized assistance**. Absolutely no transfer of program code between students is permitted (paper or electronic), and you may not solicit code from family, friends, or online forums. Other examples of academic dishonesty include emailing your program to another student, screen sharing, copying-pasting code from the internet, working in a group on a homework assignment, and allowing a tutor, TA, or another individual to write an answer for you. Academic dishonesty is unacceptable, and penalties range from a letter grade drop to expulsion from the university; cases are handled via the official student conduct process described at https://dos.uic.edu/conductforstudents.shtml .

Appendix A: “list.h”

```
/*list.h*/

//
// Author: Prof. Hummel, U. of Illinois Chicago, Spring 2021
//
// This library implements a one-way linked-list.
//

#pragma once

#include <iostream>
#include <string>
#include <exception>
#include <stdexcept>

using namespace std;

//
// The data stored in each node:
//
struct NodeData
{
   int ID;
   int Priority;
};

//
// Node in a one-way linked-list:
//
struct Node
{
   NodeData Data;
   Node*    Next;
};

//
// List defines a linked-list:
//
struct List
{
   Node* Head;
   int   Count;
};

//
// Functions:
//

//
// init_list
//
```

```
// Initializes list to empty; must be called before list can
// be used with other functions.
//
void init_list(List& L);

//
// print_list
//
// Outputs "List: " followed by each data element and a space. Output
// is directed to the console. The format of each data element is
// (field1,field2,...), e.g. (ID,Priority).
//
void print_list(List L);

//
// push_back_list
//
// Pushes the data onto the end of the list.
//
void push_back_list(List& L, NodeData d);

//
// push_front_list
//
// Pushes the data onto the front of the list.
//
void push_front_list(List& L, NodeData d);

//
// free_list
//
// Deletes all nodes from the list, frees the memory, and resets
// the list to empty.
//
void free_list(List& L);

//
// search_list
//
// Search the list for the first occurrence of the given ID. If found,
// its position in the list is returned; positions are 0-based,
// meaning the first node is position 0. If not found, -1 is
// returned.
//
int search_list(List L, int ID);

//
// retrieve_from_list
//
// Retrieve's the data from node at the given position; the list
// remains unchanged. Positions are 0-based, meaning the first node
// is position 0. Throws an "invalid_argument" exception if the
// position is invalid, i.e.
// throw invalid_argument("retrieve_from_list: invalid position");
```

```
//
NodeData retrieve_from_list(List L, int pos);

//
// remove_from_list
//
// Removes and frees the node at the given position, returning
// the node's data. Positions are 0-based, meaning the first node
// is position 0. Throws an "invalid_argument" exception if the
// position is invalid, i.e.
// throw invalid_argument("remove_from_list: invalid position");
//
NodeData remove_from_list(List& L, int pos);
```

Appendix B: “pqueue.h”

```
/*pqueue.h*/

//
// Author: Prof. Hummel, U. of Illinois Chicago, Spring 2021
//
// This library builds upon list library to implement a Priority Queue.
//

#pragma once

#include "list.h"

using namespace std;

//
// pq_init
//
// Initializes list to empty; must be called before list can
// be used with other queue functions.
//
void pq_init(List& L);

//
// pq_print
//
// Outputs "List: " followed by each data element and a space. Output
// is directed to the console. The format of each data element is
// (field1,field2,...), e.g. (ID,Priority).
//
void pq_print(List L);

//
// pq_front
//
// Returns the ID at the front of the priority queue. Throws
// an "invalid_argument" exception if the queue is empty, i.e.
// throw invalid_argument("pq_front: queue empty");
//
int pq_front(List L);

//
// pq_dequeue
//
// Removes the data element at the front of the queue, removing the
// node as well and freeing this memory. Nothing is returned.
// Throws an "invalid_argument" exception if the queue is empty, i.e.
// throw invalid_argument("pq_dequeue: queue empty");
//
void pq_dequeue(List& L);

//
```

```
// pq_enqueue
//
// Inserts the given (ID,priority) pair into the queue based on
// the priority, ordered low to high. If elements already exist with
// the same priority, this new element comes after those elements.
// Example: suppose the queue currently contains the following
// (ID,Priority) pairs:
//
//   (10,19) (84,25) (21,50)
//
// A call to enqueue (79,25) yields the following result:
//
//   (10,19) (84,25) (79,25) (21,50)
//
// If the ID is already in the queue, then the pair is repositioned
// based on the new priority. For example, suppose the queue is as
// shown above, and we enqueue (84,50). The result is now
//
//   (10,19) (79,25) (21,50) (84,50)
//
void pq_enqueue(List& L, int ID, int priority);

//
// pq_nudge
//
// Nudges the element with the given ID forward one element in the
// queue. The element takes on the same priority as the element
// that now follows it.  Example: suppose the queue currently contains
// the following (ID,Priority) pairs:
//
//   (10,19) (84,25) (21,50)
//
// A call to "nudge" 21 forward produces this result:
//
//   (10,19) (21,25) (84,25)
//
// If the ID is already at the front of the queue, nothing happens.
// If the ID is not in the queue, throws "invalid_argument" exception, i.e.
// throw invalid_argument("pq_nudge: ID not found");
//
void pq_nudge(List& L, int ID);

//
// pq_clear
//
// Deletes all nodes from the list, frees the memory, and resets
// the list to empty.
//
void pq_clear(List& L);

//
// pq_duplicate
//
// Makes a complete copy of the given list L and returns this duplicate.
// A "deep" copy is made, meaning all nodes and data are duplicated in
// the new, returned list.
//
List pq_duplicate(List L);
```