University of Waterloo
CS240 Spring 2022
Programming 1

Due Date: Wednesday, June 8 at 5:00pm

The integrity of the grade you receive in this course is very important to you and the University of Waterloo. As part of every assessment in this course you must read and sign an Academic Integrity Declaration before you start working on the assessment and submit it **before the deadline of June 8** along with your answers to the assignment; i.e. **read, sign and submit P01-AID.txt now or as soon as possible**. The agreement will indicate what you must do to ensure the integrity of your grade.

Note: All code submitted must be your own. Also, at this point, you are expected to be able to debug your code yourself.

The Academic Integrity Declaration must be signed and submitted on time or the assessment will not be marked.

Please read `https://student.cs.uwaterloo.ca/~cs240/s22/assignments.phtml#guidelines` for guidelines on submission. Submit the file `pqueue.cpp` to Marmoset.

Late Policy: Programming Questions are due at 5:00pm and **no lates are accepted**. Submissions after the deadline will not be accepted but may be reviewed (by request) for feedback purposes only.

Problem 1 [20 marks]

In this programming question, you are asked to implement a priority queue in 3 ways, using C++ and compiled using `g++ -std=c++17` in the `linux.student.cs.uwaterloo.ca` environment. An item in the priority queue will have a non-negative integer priority (key) and a positive integer value. The value is a timestamp recording the order in which items have been inserted; the first item inserted will have value 1, the second item value 2, and so on. The priority will be provided by the user while the timestamp will be computed by your program. The timestamp is initially 1 and is then incremented as each new item is inserted. One "clock" is used for all implementations; i.e. each item inserted (into any of the priority queues) will have a unique timestamp.

Each implementation of the priority queue must support the insert and deleteMax operations, as well as, findMax which returns the item with the highest priority (but does not

remove it from the data structure) and lookup which accesses an internal component of the data structure.
Implementation 1: A singly linked list that is ordered by priority where the largest priority is in the front of the list. If two items have the same priority, then the tie is broken by ordering the item with the smallest value (timestamp) first.

For this implementation, you must dynamically allocate and link the nodes yourself; i.e. you may not use the built in linked lists. You may however, use smart pointers.

Implementation 2: A max-heap implemented with a dynamic array (as described in the course notes). When the first item is inserted, an array of length 1 is dynamically allocated to accommodate the first item. You must then use the doubling strategy to reallocate the dynamic array when more space is required; i.e. if there is no space to insert an item, double the size of the array, then perform the insert. Also, when the last item has been removed, dynamic array should be reallocated back to size 1.

For this implementation, you may not use a vector or smart pointers. You must use `malloc`, `realloc`, and `free` (do not use `new` and `delete`) to implement the reallocation and doubling strategy yourself.

Implementation 3: A vector of (pointers to) queues. Each index i of the vector points to a queue of items where each item in the queue has priority i; if no items of priority i exist, this pointer must be assigned `nullptr` (i.e. **do not create empty queues if they are not needed**). If there are no items in the priority queue, the vector may be of size 0 or 1. At all other times, the vector must be sized to be 1 + largest priority stored in the priority queue. You are free to choose the implementation of the queue used at a vector index - a circular vector, linked list with back pointer, or std::queue is okay. The insert/delete (enqueue/dequeue, pushback/popfront) queue functions must run in O(1) time.

You may not use any built in data structures (unless specified) that may trivialize the implementations. You may use std::pair in any of the three implementations and std::vector may only be used in implementation 3. You may not use std:swap or anything from the algorithms library or math library. If in doubt, please ask. You must also manage the dynamically allocated memory yourself (where applicable) and must free all memory before your program terminates.

In all implementations, findMax must have runtime O(1).

Implement your program in C++ and provide a main function that accepts the following commands from stdin (you may assume that all inputs are valid). You may assume the priority queues described above are numbered 1, 2 and 3, respectively.

- `i num priority` - inserts an item (`priority, timestamp`) into priority queue `num`.
- `d num` - removes the item with the highest priority from priority queue `num` and prints the priority and timestamp to stdout - the two integers are separated by a space and a newline follows the second integer. If the priority queue is empty, does nothing.
- `f num` - prints (but does not remove) the item with the highest priority from priority queue `num`. Prints the priority and timestamp to stdout - the two integers are separated by a space and a newline follows the second integer. If the priority queue is empty, does nothing.
- `l num i` - performs a lookup of the `i`-th item of the data structure used to implement priority queue `num` and prints the following:
  - If `num` is 1, prints the `timestamp` of the `i`-th item of the linked list where the first item is the 0-th item.
  - If `num` is 2, prints the `timestamp` of the item at index `i` of the dynamic array.
  - If `num` is 3, prints the `timestamp`s of all the items in the queue at index `i` of the vector. Each timestamp is separated by a space and the last is followed by a newline. If the queue is empty, prints nothing.

  This operation does not remove items from the priority queue. You may assume that `i` will be valid for the data structure; ie. do not need to check if it is out-of-bounds.
- `r` - initializes or resets all priority queues to be empty.
- `x` - terminates the program.

Place your entire program in the file `pqueue.cpp` and **clearly label (with comments) each of the implementations so it is easy for the markers to find them in your file**.