Programming Assignment 2: Priority Queue

CSE 220: Systems Programming

Introduction

In this assignment, you will implement a *priority queue* from a predefined API described in this handout. You will use pointers and structures to implement the linked list. You will learn about simple pointer-based data structures, as well as how to read, understand, and implement an API from its specification.

Only three relatively simple tests are provided for this project, and you will have to write tests for your own code. Your tests will be validated as part of the assignment. You will learn how to write complete and sophisticated unit and functional tests, as well as how to validate the integrity of a data structure.

You will find the skills you used in the Testing lab useful for this assignment.

This handout is long and complicated, because this project is large and subtle. It is important that you read the handout carefully and understand what it is asking of you.

Academic Integrity

As detailed in the course syllabus, academic integrity is important for the value and durability of your degree from the University at Buffalo. Here are a few reminders to help you maintain the integrity of this assignment.

- It is a violation of the course academic integrity policy to share this assignment document or details about this assignment with any student at UB not enrolled in CSE 220 during this academic semester, with any student at any other institution, or with anyone else without permission from your instructor. This includes homework- and note-sharing Internet sites such as Course Hero and Chegg.
- It is a violation of the course academic integrity policy to share *any code* from this assignment with *any student* at any time during or after this semester. You may discuss your code with course staff if necessary.
- It is a violation of the course academic integrity policy to seek assistance from other students or *any online resource not specifically approved by your instructor*. Forbidden resources include: Stack Overflow/Stack Exchange, GitHub/GitLab/BitBucket, students who have previously taken this course, *etc.*
- It is a violation of the course academic integrity policy to discuss implementation details with anyone except course staff.

Students found violating any of these policies, or any other academic integrity policy in the syllabus, will be sanctioned. Sanctions may include failure in the course or expulsion from the University. Make sure that you are familiar with the course academic integrity policies, as well as the policies of the department and University.

1 Getting Started

You should read this entire handout and all of the given code, including the given tests, before you get started. You should have received a GitHub Classroom invitation for this project. Follow it and check out the resulting repository.

You will want to be caught up on your reading in *The C Programming Language* by Kernighan and Ritchie for this assignment. In particular, the assigned readings in Chapters 1-3 plus parts of chapters 5 and 6 are important for this project. If you have not read them already, you should read them now.

You should plan to *draw a lot of diagrams* for this assignment. You may find diagrams such as those found in Section 2.1 helpful, or perhaps diagrams of your own design. Understanding where individual pointers are pointing (and why) is critical to designing effective tests for this project.

Man page sections are noted as a number in square brackets after a keyword in this document. For example, calloc [3] indicates that the manual page for the `calloc()` function is found in section 3 of the Unix manual, and you can view it with the command `man 3 calloc`.

2 Requirements

You must implement a *priority queue* API for this project. The priority queue stores integer values of type `int` in a sorted structure with constant-time ($O(1)$) access for all routine operations. (If you have not yet completed CSE 250, asymptotic runtimes may not yet be familiar to you, and that's OK!) You have been given a header file, function prototypes, and an API description, and must implement the API as described.

Our priority queue contains a doubly-linked list consisting of nodes that contain some sort of data (in this case, an `int` value), a *priority*, and "links" to neighboring nodes in a chain — one link to the previous node in the chain and one link to the next node in the chain — in this case represented by C pointers. It also contains a table of pointers, one for each priority in the queue; each pointer in this table points to the *last list node of a given priority, if it exists*. (More on this later.)

You will also be responsible for writing tests for the API functionality and a data structure validator that verifies the correctness of a priority queue.

2.1 Structures

The structure for the priority queue itself is this:

```
typedef struct PriorityQueue {
    PQNode *head;
    PQNode **tails;
    int nprios;
} PriorityQueue;
```


The `head` pointer points to the first node of the associated linked list of queue items. The `tails` pointer is a pointer to an *array of pointers*, each one pointing to either `NULL` or a list node in the list starting at `head`. Finally, `nprios` specifies both *a)* the range of valid priorities in the list, and *b)* the number of entries in the `tails` array.

The structure for each node in your priority queue linked list will look like this:


```
typedef struct PQNode {
    int priority;
    int value;
    struct PQNode *next;
    struct PQNode *prev;
} PQNode;
```


The `priority` is an integer in the range $[0, \text{nprios})$. It represents the relative priority of this list item compared to other list items; items of "higher priority" (lower numeric priority values) appear *before* items of "lower priority" (higher numeric priority values) in the list. The `value` is a number that is provided by the user of the API when the item is inserted into the list; it is not used by the API itself for any reason. The `prev` and `next` pointers form the links between nodes in the list.

This list is maintained *sorted in priority order* (numerically) at all times, and new items are inserted *at the end of all items of their priority level*. This allows the earliest-inserted item of the highest priority in the list to be immediately found by looking at the head of the list.

The most complicated part of this data structure is the relationship between the `tails` pointer array and the linked list starting at `head`. Because the linked list contains multiple priorities, inserting a node at the end of all nodes of a given priority would require iterating the list from the head to the insertion point, if additional information were not provided. For a long list, this is an expensive linear-time ($O(n)$) operation. Thus, the `tails` array keeps track of the *last list node* of each priority, provided that it exists. The resulting structure for an example list having `nprios = 3` and containing the list items { (priority: 0, value: 33), (priority: 0, value: 12), (priority: 2, value: 20), (priority: 2, value: 27) } would look like Figure 1.

Note that an item inserted with priority 0 would be inserted exactly after the node pointed to by `tails[0]`, and an item inserted with priority 2 would be inserted exactly after the node pointed to by `tails[2]`. The (oldest) highest priority item in the list is pointed to by `head`, so it must be the case that value 33 was inserted *before* value 12, and that value 20 was inserted before value 27. The entry at `tails[1]` is `NULL`, because there are no items in the list having priority 1.

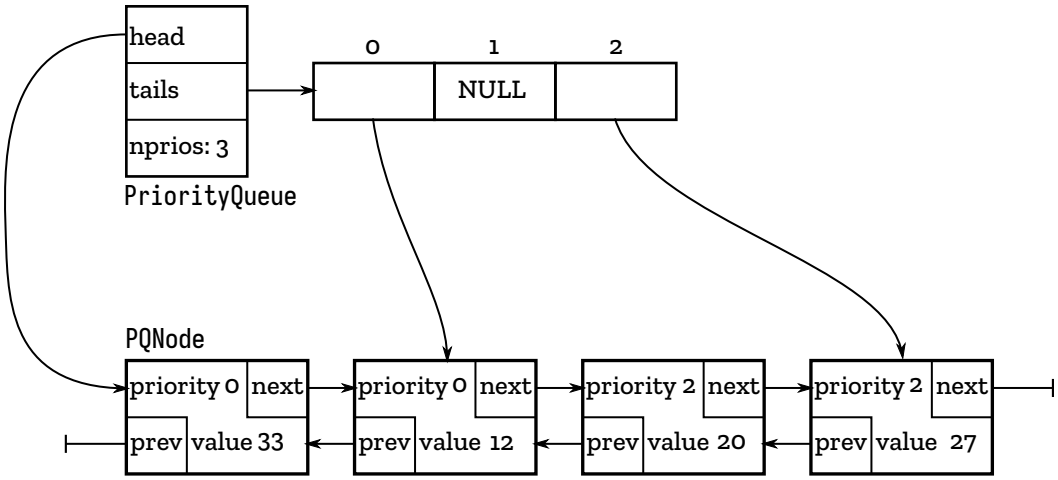

Figure 1: An example `PriorityQueue`

2.2 Invariants

This data structure maintains the following *invariants*. Recall that an invariant is a statement that is **always true** about a data structure. If your code returns from a function that manipulates a priority queue and any of the following statements are untrue, your code has a bug.

- An empty priority queue has `head == NULL` and every entry in `tails` is `NULL`. There are exactly as many entries in `tails` as the `nprios` entry.
- The priorities of the nodes in the list are sorted in *non-increasing* (*numerically non-decreasing*) *order*.
- Every node in the list has a priority between 0 and `nprios - 1`.
- The head of the list has `head->prev == NULL`.
- Every node except the head has a non-NULL `prev` pointer, and `node->prev->next == node`.
- The tail of the list has `node->next == NULL`.
- Every node with a non-`NULL` `next` pointer has `node->next->prev == node`.
- Every non-`NULL` entry in `tails` points to the last list item (farthest from `head`) having `priority` equal to the index of the entry.
- If an entry in `tails` is `NULL`, there are *no* list items having a priority equal to the index of the entry.

This means that there are a number of special cases to be checked, including:

- The empty queue
- A queue with one or more empty `tails` pointers

Full credit on this assignment will depend on identifying and correctly handling (and correctly testing for!) those special cases.

2.3 API

You must implement the following functions. All of the functions must adhere to the above invariants, *and all except pqueue_free() must run in constant time*. This means that **no function other than `pqueue_free()` can iterate the list**, and all functions must use the `tails` array correctly.

- `PriorityQueue *pqueue_init(int nprios);`:
  Create a new, empty priority queue and *return a pointer to that queue.* The memory for the queue structure and its tails table **MUST** be allocated using `malloc()` or `calloc()`. The `nprios` field in the created queue **MUST** be equal to the `nprios` argument, and the `tails` table **MUST** have `nprios` entries. Returns `NULL` if `nprios` is less than or equal to zero.

- `PQNode *pqueue_peek(PriorityQueue *pqueue);`:
  Return a pointer to the `head` node of the specified queue (or `NULL` if the queue is empty).

- `PQNode *pqueue_get(PriorityQueue *pqueue);`:
  Remove the head node from the specified queue and return the removed node. This function **MUST** update `head` and the `tails` table appropriately if necessary. If the queue is empty, this function **MUST** return `NULL`. The `prev` and `next` members of the returned node will not be examined. The removed node **MUST NOT** be freed by this function.

- `void pqueue_insert(PriorityQueue *pqueue, int value, int priority);`:
  Create a new queue node with the given `value` and `priority` fields and insert it into the queue, updating `head` and `tails` as appropriate. The new node **MUST** be allocated using `malloc()` or `calloc()`. This function should do nothing if the specified priority is not in the allowable range for the specified queue.

- `void pqueue_free(PriorityQueue *pqueue);`:
  Free all memory allocated to the priority queue by using `free()`. (Note that this requires freeing *each of the list nodes individually*, the `tails` table, and the queue structure itself.) Because this function must access every list node, it will not run in constant time.

These functions *are not required to work properly on malformed queues*. For example, if a queue contains a priority greater than its `nprios`, your function can do anything at all, including detecting the problem and returning early or simply crashing the program. Your queue API should do "reasonable things" (that is, it should not arbitrarily crash or corrupt the data structure if this is avoidable) when presented with invalid arguments, but it will not be tested with any invalid arguments that are not explicitly described above.

The API functions **MUST** be implemented in `src/priority_queue.c`, and **MUST NOT** require any other files to link and run correctly.

2.4 Tests and the Queue Validator

The only tests provided for these functions are found in `tests`, and test `pqueue_init()` for basic appearance of functionality and `pqueue_insert()` for a simple insert operations on a queue. A large part of your assignment will be *writing tests to verify the correctness of your API functions*. You will be graded on the tests that you implement and submit. The quality of these tests will also show up indirectly in the quality of your implementation of the API. You should **take this testing very seriously**, as it will ensure a satisfactory project grade.

You must write one or more tests to identify whether or not an implementation of this API is correct. Much like the tests you wrote for the testing lab, these tests should *call the specified API functions*, and determine whether or not they work correctly. Each test should be self-contained in a `.c` file that depends only on `src/priority_queue.h` and standard C library functions. When run, it may print any diagnostic information you like (which will be ignored by the grading framework) to either standard output or standard error, and it **MUST** exit with a zero status (returning 0 from main or calling `exit()` [3] with a zero argument) if it does not detect an error in the implementation, and with a non-zero status if it does detect an error in the implementation.

Your goal in writing these tests is to differentiate between a working API and a broken API. You may write one large test, or many small tests, or whatever combination you choose. I recommend that you write a number of small tests. Your tests will be evaluated for correctness by:

- Expecting *every test* to return a zero status for a correct implementation of the API, and
- Expecting *one or more tests* to return a non-zero status for a broken implementation of the API.

It is reasonable, for example, for a test that checks only the functionality of `pqueue_insert()` to pass an implementation that has a working `pqueue_insert()`, but a buggy `pqueue_get()`. However, you would expect *some other test in the suite* to fail the buggy `pqueue_get()`.

In addition to these API functions and whatever tests you may write to test the API, you **MUST** implement a single testing function in `src/validate.c`:

- `bool pqueue_validate(PriorityQueue *pqueue);`:
  Return `true` if `pqueue` is a valid priority queue that meets all of the invariants, or `false` otherwise. It **MUST** handle a malformed priority queue with `NULL` in any location, or with a pointer to a correctly formed but invalid data item in any location, without crashing. It *need not handle pointers to invalid locations* without crashing.

The queue validator **MUST** be implemented in `src/validator.c`.

The tests that you write *may* provide any functionality that you need, and **MUST NOT** assume that they are run against your implementation of the API. Your tests will be evaluated by running them against known good and known bad implementations of the required API, and should be able to distinguish the two.

Neither tests nor the queue validator must run in constant time; in particular, the validator is *expected* to walk the entire queue in order to perform its duties!

3 Allocation

You should use `malloc()` or `calloc()` to allocate the queue structures and list nodes, and `free()` to free them when they are no longer needed. The `calloc()` function is particularly useful for this since it clears the memory (sets all bytes to 0) before returning it. See `calloc()` [3] and `malloc()` [3], as well as lecture material and K&R, for more information.

4 Testing

The given code contains three example tests, and the Makefile contains rules that you can use to create additional tests with minimal effort. If you have been keeping up on assignments and readings up to this point (including reading the Make documentation for Lab 02), you should find it approachable to create and run tests of your own. If you add them to the appropriate variable in the Makefile, they will be run when you invoke `make test`, which will help you run them often and use them to catch bugs and regressions. You may even find it helpful to implement `pqueue_validate()` and some additional tests *before* you do the majority of your queue implementation.

You should certainly read the given tests, and their comments, for guidance on how to implement and test your project.

5 Guidelines

You may find it helpful to implement the functions in roughly this order, *alongside their tests*, returning to functions that appear to be incomplete or incorrect as necessary:

- `pqueue_validate()`
- `pqueue_init()`
- `pqueue_insert()`
- `pqueue_peek()`

- `pqueue_get()`
- `pqueue_free()`

Remember to test as you go! It is very difficult to write an entire project and *only then* start testing. Starting with even tests for the well-formedness of the queue in `pqueue_validate()` and running it regularly (perhaps via a `make test` rule) will be very valuable. *Add your tests to `make test`!* (You must do this for submission to Autograder, anyway.)

5.1 Understanding the `tails` Array

The `tails` array enables an item to be inserted in the queue very rapidly even if there are many items already enqueued. Consider a queue that does not have a tails table inserting an item of priority 0 when there are already 3,000 items at priority 0. This queue would have to iterate all 3,000 enqueued items of priority 0 to find the *last* such item, then insert the new item between this and the next item (of some priority greater than 0, or at the tail of the list).

Considering Figure 1, you can see that the `tails[0]` pointer directly indicates the last node having priority 0. In our hypothetical queue containing 3,000 nodes of priority 0, this would allow insertion of a new node after the 3,000th node *without walking the list*. The insert function can simply consult the `tails` array and immediately locate the correct node to begin the assertion. It then simply has to fix up the `tails[0]` pointer after insertion and the structure invariant remains intact!

The only place that this gets tricky is where there *are nodes in the queue*, but there *are no nodes at the target priority*. Looking at Figure 1 again, we see that `tails[1]` is `NULL`. This means that there are no nodes in the list at priority 1, so any newly-inserted node at priority 1 would need to follow the node at `tails[0]`, instead. It would, of course, then become the tail of the nodes at priority 1.

(**Warning:** Algorithm analysis ahead! Try to follow along if you have had or are taking 250!) This means that the complexity of insertion of a node is independent of the number of nodes in the list, but *dependent on the number of priorities in the queue*. In the worst case scenario, you have nodes of priority 0, but no nodes of priorities from 1 through `nprios - 1`, and you are inserting at priority `nprios - 1`. This requires checking every entry in the `tails` array, starting at `nprios - 1`, to determine that they are all `NULL` until you reach 0. If we denote the number of nodes in the list as n and the number of priorities in the queue as p, then inserting into a priority queue *without* a tails list is $O(n)$, but inserting into a priority queue *with* a tails list is $O(p)$. Since we expect that $p \ll n$, this is much faster. Furthermore, if we assume that p is always bounded above by a constant (which is reasonable; most priority systems have no more than on the order of a few dozen priorities, and many have only 2–4), then the operation using the `tails` array is $O(1)$!

6 Submission

Use the command `make submission` to build the file `pqueue.tar`, which is uploaded to Autograder. You will turn this assignment in in **two parts**. The first part will be due after one week, and the second part in two weeks. (Check Piazza and the Autograder for specific dates and times.) The first submission will test your tests and your queue validator, and the second submission will be for your final API implementation. The Autograder for both parts will be the same. It will provide minimal testing of your API to verify that it compiles, and will evaluate *only your tests and validator*. You must perform correctness testing of your API yourself (using the tests you submit).

7 Grading

This assignment is worth 5% of your course grade. Points for this project will be assigned as follows. The points for Part 1 will be evaluated by the autograder during the first week, and for Part 2 after the final submission at the end of the assignment. Improvements on your Part 1 score during the second week will not directly affect the score for Part 1, but *will* indirectly affect the correctness of your API (and therefore your total project score).

Points	Part	Description
2	*N/A*	Handout quiz
1	1	Tests identify correct API functions
2	1	Tests for `pqueue_insert()` identify bugs
1	1	Tests for `pqueue_peek()` identify bugs
1	1	Tests for `pqueue_get()` identify bugs
2	1	Queue validator is correct and complete
0.5	2	Queue init is correct
2	2	Queue insert is correct
1	2	Queue insert is $O(1)$
0.5	2	Queue peek is correct
2	2	Queue get is correct
1	2	Queue free is correct
2	2	Queue operations and free leak no memory
2	2	Stress tests pass

The "stress tests" will perform many and varied queue operations. Each of the other tests will attempt to test the specific required functionality as closely as possible. Some tests may unavoidably require more than one function to operate, however.