Part 2

Restrictions

Same as Part 1.

Logistics

The autograder for Part 2 will not run unless all tests for Part 1 pass.
Due:

- **Gradescope: Tuesday 11/19**
  - `prqueue.h`
  - `prqueue_tests.cpp`
- Grace tokens:
  - https://script.google.com/a/macros/uic.edu/s/AKfycbw_Pi4Pn-dxBnXAZrSPBY49m-w705uxyf0JIYGDXA0db52p86hby2Ki9vu4-b59vIQJcg/exec
    - This form requires your UIC Google Account. The permissions it requests are to know who you are, and for the script to edit our spreadsheet database in the back end.
    - The form will become active **once the project autograder closes**.
    - See [FA24] CS 251 Course Info for details.

For this part, you will use the same testing and implementation sequence you used for Part 1. We have split this part into 4 sections: `Duplicates`, `Copying`, `Equality`, and `Iterator`.

FAQ

[FA24] Priority Queue Part 2 FAQ

Running Code

- `make prqueue_tests`
  - This default target builds the `prqueue_tests` executable.
- `make test_duplicates`
- `make test_copying`
- `make test_equality`
- `make test_iterator`
  - These are the 4 milestones for this part of the project.
- `make test_all`
- `make prqueue_main`
- `make run_prqueue`

Priority Queues with Duplicates

If two elements in a priority queue have the same priority, which should be dequeued first? We will fall back to queue behavior: if two elements have the same priority, the element that was enqueued first should be dequeued first.

For example, patients that arrive in this order:

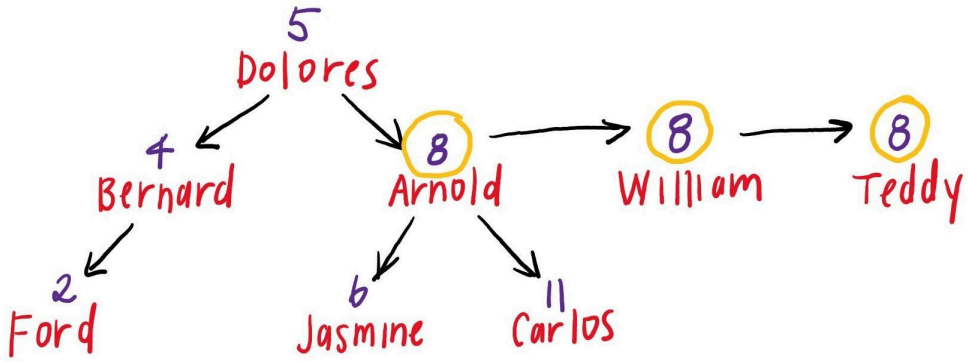
- "Dolores" with priority 3
- "Bernard" with priority 2
- "Arnold" with priority 4
- "William" with priority 3
- "Teddy" with priority 3
- "Ford" with priority 1

would be dequeued in the order: Ford, Bernard, Dolores, William, Teddy, Arnold. Note the order in which Dolores, William, and Teddy are dequeued.

However, BSTs as we know them don't *really* support duplicate values. To support this, we're going to implement a variation! Each node in the BST, rather than being a single node, will be a **linked list**. Let's say that we add elements in this order:

- "Dolores" with priority 5
- "Bernard" with priority 4
- "Arnold" with priority 8
- "Ford" with priority 2
- "Jasmine" with priority 6
- "Carlos" with priority 11
- "William" with priority 8
- "Teddy" with priority 8

Then, our BST would look like this:



This is an incomplete diagram and does not show all the pointers in use.

The node labeled “Arnold” with priority 8 is the head of a linked list. The elements of this linked list are all the nodes with priority 8, in the order they were inserted.

These elements would be dequeued in the order: Ford, Bernard, Dolores, Jasmine, Arnold, William, Teddy, Carlos. Nodes with duplicate values are, again, dequeued one-by-one in the order they were enqueued.

Internal Structure

There’s more pointers that are in use in the `NODE`, so the autograder does some more complex checks for Part 2:

- For the *head node* of each linked list:
  - The `parent`, `left`, and `right` pointers must point to the head node of the correct linked list.
  - If there is another node with the same priority, the `link` pointer must point to the correct node.
- For *the remaining nodes* of each linked list:
  - The `link` pointer must be correct. The `link` pointer points to the next `NODE` in the linked list.
  - The `link` pointer of the last node in each linked list must be `nullptr`.
  - The other pointers for these non-head nodes (`parent`, `left`, `right`) are not checked. You can use them however you see fit, or ignore that they exist. This is a design decision for **you** to make!

Now we will go into more detail about testing and implementation of each section in Part 2. We recommend working on the sections and functions in the order that we present them here since tests for later functions may depend on previous functions working correctly.

Task: Testing Duplicates

Atr this point, you should write additional tests to test handling duplicates, with the functions that you have already implemented. You should check whether or not all your functions are working correctly with priority queues that have duplicate priorities.

This section will run tests in the `PRQueueDuplicates` suite. You can run your own tests on your code using `make test_duplicates`.

Buggy solution:

- `enqueue` doesn’t insert duplicates in the appropriate linked list.
- `enqueue` doesn’t insert duplicates in the right order in the linked list.

- `as_string` doesn't print duplicate values
- `dequeue` loses all other duplicates that are not returned

Task: Implementation Duplicates

Modify your earlier functions to handle duplicates. We recommend updating them in this order:

1. `enqueue`
2. `clear` and the destructor
3. `as_string`
4. `dequeue`
    a. `dequeue` can be tricky to implement correctly with duplicates. Depending on whether you're using extra pointers in your linked lists, make sure you update all of them!

Task: Testing Copying

1. Copy constructor
2. `operator=`

This section will run tests in the `PRQueueCopying` suite. You can run your own tests on your code using `make test_copying`.

Buggy solutions:

- `operator=` returns an empty priority queue
- `operator=` returns a priority queue with an incorrect size
- `operator=` does not copy duplicates
- copy constructor does not properly create new nodes
- copy constructor does not copy duplicates

Task: Implementing Copying

Implement the two Copying functions:

1. copy constructor for `prqueue`
2. `operator=`

For these functions, make sure to copy the exact tree structure, not just the values and priorities.

Task: Testing Equality

The Equality section has only one function for which you will write tests:

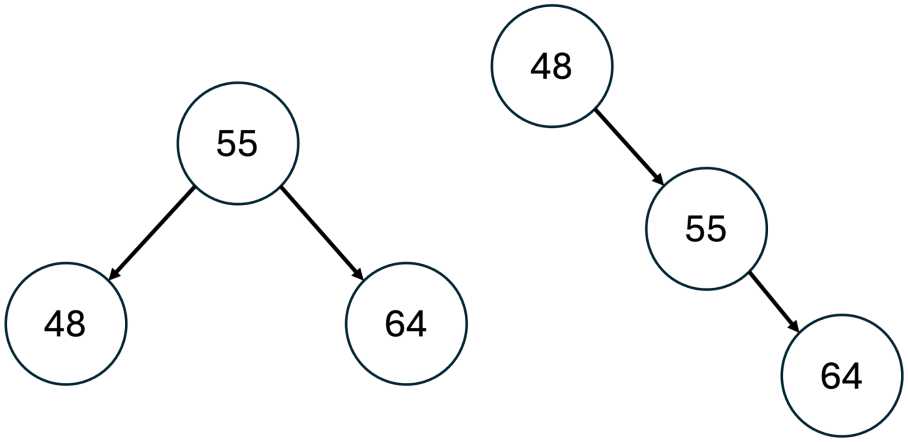
3. `operator==`

This section will run tests in the `PRQueueEquality` suite. You can run your own tests on your code using `make test_equality`.

Buggy solutions:

- `operator==` returns an incorrect result when priority queues are equal.
- `operator==` returns an incorrect result when priority queues are not equal
- `operator==` may return an incorrect result when priority queue sizes are equal

Two trees are considered equivalent if and only if their internal structure (values, priorities, and nodes) is the same as well. So, these trees would not be equivalent:



Task: Implementing Equality

Implement the one Equality function:

3. `operator==`

Task: Testing Iterator

The Iterator section has only two functions for which you will write tests:

4. `begin`
5. `next`

This section will run tests in the `PRQueueIterator` suite. You can run your own tests on your code using `make test_iterator`.

You may assume that these functions are used “properly”. That is, don’t worry about the `prqueue` being modified while someone is using `begin` and `next` to iterate the tree. You can also assume that `begin` is called before `next`, and that we won't copy or assign a tree in the middle of iteration.

While we talk about internal implementation details, such as `curr`, it's possible to reveal bugs here just by checking the outputs of `begin` and `next`. You won't need to (and can't) access `curr` directly, since it's a private member! You will also need to pay attention to the trees that you're setting up, in addition to making sure that you check the results of *all* the "returns".

Buggy solutions:

- `begin` doesn't properly set value of `curr` when the priority queue is empty
- `begin` doesn't properly set the value of `curr` when the priority queue is not empty
- On some trees, `next` doesn't properly set `curr` to the next value in the priority queue
- On some trees, `next` returns `false` even though there are more nodes to traverse
- On some trees, `next` returns an incorrect value by reference.
- On some trees, `next` returns an incorrect priority by reference.

Task: Implementing Iterator

Implement the two functions in the Iterator section:

- `begin`
- `next`

`next` is the most difficult function in the project. Plan your time accordingly.

Both of these functions work with the `NODE* curr` that is a member of the `prqueue`. `curr` is a utility pointer that should point to the next `NODE` that will be returned by the `next` function. Optionally, you can use `NODE* temp` to keep track of additional info, but you're not required to.

When we implemented `as_string`, we wrote an in-order traversal. However, we had to do this in-order traversal "all at once" – it's wrapped up and abstracted away inside the `as_string` function and its helper. What if we wanted to perform a different operation on the nodes in-order? We'd have to reimplement the in-order traversal for each operation![1]

Instead of doing that, for this section, we'll make an *iterative* version of the in-order traversal. This is sort of like a `while` loop for linked lists: we use `begin` to start the in-order iteration, and `next` to advance the in-order iteration. See the documentation for `next` for an example of how it might be used.

Given a node, how do you find the next node in an in-order traversal? Here's an algorithm for a standard BST. Given the current node:

[1] If you know a lot more about C++, you might point out lambdas – but these are a good deal more complicated than we want to deal with right now.

- If the right subtree exists:
    - The next in-order node is the node with the smallest priority in the right subtree.
- Else:
    - The next in-order node is an ancestor of the current node.
    - While the current node is a right child:
        - Set the current node to its parent
    - Set the current node to its parent one more time. This is the next in-order node.

The algorithm as described is incomplete in 2 ways:

- It will need to be modified for our BST with duplicates.
    - How do we make sure to iterate duplicates?
    - How do we continue after reaching the end of one priority's duplicates?
- It does not say how to tell when it has reached the end of iterating.

These two functions must not modify any member pointers in the tree or in nodes aside from `curr` and `temp`**.** Depending on your implementation, you might not use `temp`. This is fine.

Grading Breakdown

Milestone	**Total Points**
`prqueue` Duplicate Testing	4
`prqueue` Duplicate Implementation constructor, `size`,`enqueue`, `peek`, `dequeue`, `as_string`, `clear`, destructor	14
`prqueue` Copying Testing	20
`prqueue` Copying Implementation copy constructor, `operator=`	28
`prqueue` Equality Testing	32
`prqueue` Equality Implementation `operator==`	36
`prqueue` Iterator Testing	48
`prqueue` Iterator Implementation `begin`, `next`	**60**