

# Project 4 - Priority Queue (BST)

CS 251, Fall 2024

In this project, we will continue building our own version of data structures. By the end of this project, you will...

- Have created a templated binary search tree
- Designed a particular implementation of a priority queue
- Implemented multiple traversals through a binary search tree

## Restrictions

- **You may not include additional C++ libraries to implement `prqueue`.**
  - The only included libraries are `<sstream>` (for `as_string`) and `<iostream>` (for debugging).
  - You may (and should) include STL libraries to write tests.
- Files:
  - You may modify `prqueue_tests.cpp` freely.
  - You may also modify `prqueue_main.cpp`, which is not submitted.
  - You may modify `prqueue.h` only to add additional **private** member functions and to implement the functions that are left for you to do. You may not add additional member variables to `prqueue` or `NODE`, or additional public member functions.
    - If you modify this in an unexpected way, the autograder tests may fail to compile or behave extremely weirdly.
- As in Project 3, your program and tests **must** be memory-safe and may not have memory leaks. See [Memory Safety & AddressSanitizer](#).
- You will need to use classes, pointers, and `new`.
  - Do not use `malloc`, `realloc`, `free`, etc. – we're not writing C.

## Logistics

This project is larger. As such, we are splitting the project into **two parts**. The first part will be due 11/12, and the second part will be due a week after that. Each part can be individually extended with grace tokens. The two parts, together, are worth 1 project in the overall course grade.

**Part 2 cannot be completed without Part 1. Part 2's autograder will not run until you have completed all of Part 1.**

## Part 1 (40 points)

Due:

- **Gradescope: Tuesday 11/12**
  - `prqueue.h`
  - `prqueue_tests.cpp`
- Grace tokens:
  - Since the due date for this project is being split into two, we will add one additional grace token to the total you had for the semester, for use on Project 4.1 and later.
  - [https://script.google.com/a/macros/uic.edu/s/AKfycbw\\_Pi4Pn-dxBnXAZrSPBY49m-w705uxyf0JIYGDXA0db52p86hby2Ki9vu4-b59vIQJcg/exec](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.

## Part 2 (60 points, Part 1 is a required prerequisite)

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](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.

## FAQ

[\[FA24\] Priority Queue Part 1 FAQ](#)

## Running Code

- `make prqueue_tests`
  - This default target builds the `prqueue_tests` executable.
- `make test_core`

- `make test_using`
  - These are the two milestones for Part 1 of the project.
- `make test_all`
- `make prqueue_main`
- `make run_prqueue`
  - These two targets allow you to run the `prqueue_main.cpp` file, where you have a blank space to ad-hoc test your `prqueue`.

## Testing

We will continue to require you to write tests for the data structure you will implement in this project. We'll be taking the same approach as Project 3: we have many buggy implementations, and we will give you a vague description of how each implementation is broken. You will then write tests that expose each of these buggy implementations! The buggy solutions are only related to the functions you will implement in `prqueue.h`. As in the previous projects, testing each section is considered a milestone, and must be completed before you can earn credit for the implementation itself.

As with previous projects, we want you to write tests before implementing your own functions. You must complete the testing milestone for a specific section first, before your own implementation is tested. As with Project 3, you must then also pass your own tests, for the autograder for your implementation of that section to run.

**Start early, submit early, submit often.**

Keep in mind that the course staff's implementation must pass your tests to receive any credit – no writing `EXPECT_THAT(true, Eq(false))`, for example.

If at any point you see a message on Gradescope that says:  
**Your tests failed on the course staff solution, and are testing incorrect or unspecified behavior!**  
**You will need to fix your tests before proceeding.**

This means that your tests are incorrect, since the course staff solution (which is correct) is failing at least one of your tests, or is memory-unsafe. Thus, check and modify your tests that are checking incorrect or unspecified behavior.

## Memory Safety & AddressSanitizer

The same requirements apply to this project as for Project 3: your programs may not have any errors reported by AddressSanitizer. We aren't managing pointers as values this time, but we still need to be careful with the nodes themselves. Whenever a node is removed from the tree, we are responsible for deallocating the memory of that node.

## Priority Queues

We've talked about the queue ADT before, which processes elements in the order in which they were received: First-In, First-Out (FIFO). This FIFO model isn't appropriate for a lot of applications, such as hospital triage. The hospital wants to see patients with more critical injuries before others with less serious needs. The **priority queue** ADT is a way of describing these operations.

A priority queue is similar to a queue or a stack, where we can put elements in, and get them out. Unlike a queue/stack, each element also has a **priority value**, which is a number. For example, a hospital emergency department might use the [Emergency Severity Index](#). A patient who requires immediate life-saving care would have a priority of 1; while a patient who would be fine for a while might have a priority of 4. This means that **lower priority values** should be seen first.

In Part 1 of this project, we will assume that we will never have to handle two elements with the same priority value. In Part 2, we will modify our Part 1 implementation to handle duplicate priorities!

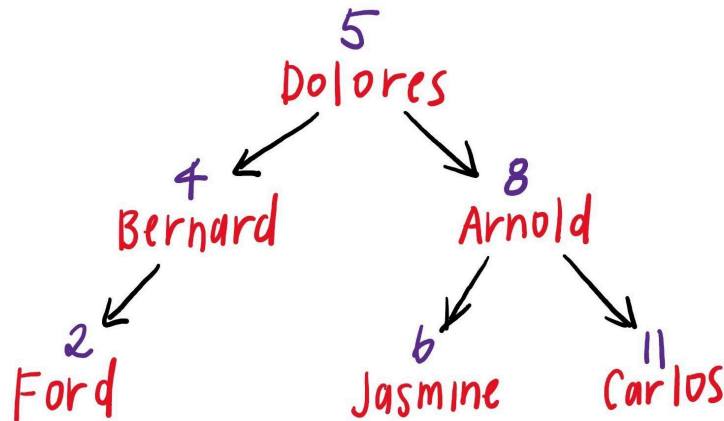
## Binary Search Tree as Priority Queue

The priority queue ADT can be implemented in many ways. There are different data structures to choose from as the underlying data structure, such as: an array, a vector, linked list, etc. For this project, we're going to use a binary search tree on the priorities, to allow us to quickly find the element with the lowest priority, barring the worst-case performance.

For example, for patients that arrive 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

Our BST would look like this:



**Note:** This is an incomplete diagram and does not show all the pointers that would be in use in our program's BST.

This priority queue is storing string values that represent the names of the patients. The order in which elements would be dequeued is ascending order by priority: "Ford", "Bernard", "Dolores", "Jasmine", "Arnold", "Carlos".

## Templates

Our `prqueue` is templated, which allows us to make a `prqueue` that stores integers, strings, or any type as the value by writing only one class definition. We can initialize a `prqueue` similarly to how we created a `vector`, by providing the template parameter:

```
prqueue<int> pq;
```

This creates a priority queue called `pq` that will store `int` values.

When we implement templated classes, we can use values of a generic type `T` like any other type. When the program is compiled, the compiler will make sure that whatever type replaces `T` supports all the operations we use with it.<sup>1</sup>

## Tip for Testing in this Project

In Project 2 and 3, course staff used STL containers to help write their tests, such as `deque`. This let us take advantage of STL functions to write large tests: we didn't have to hardcode the expected values for 100 items.

---

<sup>1</sup> We're omitting a lot of details here. If you're curious, see [LearnCPP 13.11](#). "Any type" is slightly inaccurate – it needs to be default-constructible, copy-constructible and support `<<`. This is still pretty broad!

In this project, we highly recommend using STL containers (`vectors` and `maps`) to help write your tests. One way of interpreting a `prqueue` is as a map from priorities to values. This gets us the sorted order of priorities for free! This is also straightforward to extend to duplicates in Part 2, by using a vector as the map value instead.

For example, to test `as_string`, we might go through the following steps:

- Generate the data for inserting into the `prqueue`.
- Use a `map` as the “source of truth”, and insert the data into the `map` and `prqueue`.
- Call a function that converts the `map` to the `prqueue<T>::as_string` format (see example `map_as_string` function below).
- Compare the string that results from the map to the actual `as_string`.

Here is an example implementation of this approach using integers as the values stored in the priority queue (note that this example places duplicate priorities in the priority queue, and won't work for Part 1):

```
string map_as_string(const map<int, vector<int>>& m) {
    stringstream ss;
    for (const auto &e : m) {
        int priority = e.first;
        vector<int> values = e.second;
        for (size_t j = 0; j < values.size(); j++) {
            ss << priority << " value: " << values[j] << endl;
        }
    }
    return ss.str();
}
```

```
TEST(Test, Test) {
    map<int, vector<int>> mp;
    vector<int> vals = {15, 16, 17, 6, 7, 8, 9, 2, 1};
    // Note that this uses duplicate priorities!
    vector<int> priorities = {1, 2, 3, 2, 2, 2, 2, 3, 3};
    prqueue<int> pq;
    for (int i = 0; i < vals.size(); i++) {
        pq.enqueue(vals[i], priorities[i]);
        mp[priorities[i]].push_back(vals[i]);
    }
```

```

    }
    EXPECT_EQ(pq.size(), vals.size());
    EXPECT_EQ(pq.as_string(), map_as_string(mp));
}

```

## Part 1

### Internal Structure of `prqueue`

Our `prqueue` implementation is a binary search tree, where the nodes are of type `NODE`. `NODE` is a struct that is defined like this:

```

struct NODE {
    int priority;
    T value;
    NODE* parent;
    NODE* left;
    NODE* right;
    NODE* link; // Link to duplicates -- Part 2 only
};

```

There's a bit more in here than the BST node we saw in class:

- Each node carries 2 pieces of data:
  - `priority` (for BST organization and priority queue ordering)
  - `value` - the value of type `T` that is stored in the node
- Each node, in addition to its `left` and `right` children, points to its `parent`.
- `link` is for Part 2, when we will handle duplicates. You can ignore it until then.

**The course staff autograder for Part 1 will check your internal tree structure.** Specifically, the autograder will check that the `parent`, `left`, and `right` pointers are all “correct”.

We strongly recommend reviewing Lectures 23-24 and Lab 9. If you need additional resources, see zyBooks HW 9.

### Testing and Implementation Sequence for this Project

As described above in the [Testing](#) section, we're evaluating your tests using the same approach from Project 3.

See `prqueue.h` for documentation and a description of what each function does. Write your tests in `prqueue_tests.cpp`. Remember to use `EXPECT_THAT` (keep going when it fails) or `ASSERT_THAT` (stop when it fails) where appropriate.

When you submit to Gradescope, we will run your tests on the course staff solution. Just like in Project 3, **your tests may fail on the course staff solution**. If this happens, you're testing incorrect or unspecified behavior, and will need to fix your tests! If the correct solution passes your tests, we will then run your tests on buggy solutions to check whether you tested each "thing". Then, we'll run your tests on your own code. When those pass, we'll finally run the autograder tests on your code.

We've broken down Part 1 of this `prqueue` implementation into 2 sections: `Core` and `Using`.

**For each of the 3 sections of Part 1 of `prqueue`...**

**First, write tests for the part's functions that catch the buggy solutions.**

We will give you a vague-ish description of how each buggy solution is incorrect. When you submit a test suite, we'll use your tests from a specific suite to try to detect the buggy solutions.

Buggy solutions are always "testable" by only using the functions implemented in that part or in previous parts. You can make a buggy solution fail a test by expecting or asserting the results of `prqueue` functions or `NODE` data, or by relying on ASan detection of a memory error or leak. In this project, the specific trees that you're performing operations on are just as important as the operations themselves. It's entirely possible to thoroughly check a single tree, and miss some bugs!

One test may catch many buggy solutions; one buggy solution may be caught by multiple tests; or a test might not catch any buggy solutions. Your tests must not time out (30s) on any solution. We do not have buggy solutions that create infinite loops.

**Second, submit these tests to the autograder** to verify that your tests (a) pass on a correct solution and that you're testing correct behavior, and (b) catch the described buggy solutions. You must catch all the buggy solutions to earn credit for the testing milestone for each section.

If your tests compile, pass on the correct solution, and catch all the buggy solutions, you'll see something like this:



```

The course staff solution passed on your tests!
Running your PRQueueCore tests on the buggy solutions...
CAUGHT buggy ctor_size: default constructor sets the starting size incorrectly
CAUGHT buggy ctor_root: default constructor sets the root of the tree incorrectly
CAUGHT buggy size_incorrect: size returns incorrect value
CAUGHT buggy enqueue_parent: enqueue finds an incorrect node to insert the new node as its child
CAUGHT buggy enqueue_size: enqueue does not properly update the size of the prqueue
CAUGHT buggy enqueue_child: enqueue finds the proper node to insert, but places it in the wrong place
CAUGHT buggy clear_size: clear resets the size incorrectly
CAUGHT buggy clear_root: clear does not properly reset the value of the root
-----
All buggy solutions caught!

```

If you didn't catch some of the buggy solutions, they will be listed with a red **MISSING**, and you'll need to update your test suite to test further behavior.


### Third, implement the section's functions in **prqueue.h**.

Your code **must** pass your own tests with ASan before the autograder's tests will run. We encourage you to go back and edit your tests to make their output clearer, or to add additional tests.

The autograder's tests on Gradescope are likely more detailed than yours, so they may catch things that yours don't!

Now we will go into more detail about testing and implementation of each section. We recommend working on the sections and functions in the order that we present them here.

## Task: Testing Core

 [FA24] Priority Queue Part 1 FAQ

The **Core** section has six functions for which you will write tests:

1. Default constructor for the **prqueue**
2. **size**, **getRoot**
3. **enqueue**
4. **clear**
5. destructor for **prqueue**

This section will run tests in the **PRQueueCore** suite. You can run your own tests on your code using **make test\_core**.

How can you test that `enqueue` worked correctly without any sort of function that lets you see inside the tree? Unfortunately, we'll have to use `getRoot` here to look directly at the internal structure, as annoying as that is. See the FAQ for more details!

**If you downloaded the starter code before 2:30 PM on 10/25, you will need to update the signature of `getRoot`, and add return statements to the necessary functions. We recommend just downloading the starter code again.**

Buggy solutions:

- Default constructor sets the starting size incorrectly
- Default constructor sets the root of the tree incorrectly
- `size` returns incorrect value
- On some trees, `enqueue` finds an incorrect node to insert the new node as its child
- `enqueue` does not properly update the size of the `prqueue`
- On some trees, `enqueue` finds the proper node to insert the new node but places it in the wrong place
- `clear` resets the size incorrectly
- `clear` does not properly reset the value of the root

## Task: Implementing Core functions

Implement the six `Core` functions (we recommend implementing them in this order):

1. Default constructor
2. `size`, `getRoot`
  - a. Your implementation for each of these **must** be one statement long.
3. `enqueue`
4. `clear`
5. destructor for `prqueue`

Throughout Part 1, you can assume that no duplicate priorities will be enqueued. **You will handle duplicates in Part 2**, in the same codebase, so we recommend leaving TODOs where you plan to add code in the future.

## Task: Testing Using

 [FA24] Priority Queue Part 1 FAQ

The `Using` section has two functions for which you will write tests:

6. `as_string`

7. `peek`
8. `dequeue`

In both these functions depend on computing the current “front” of the priority queue. This section will run tests in the `PRQueueUsing` suite. You can run your own tests on your code using `make test_using`.

Buggy solutions:

- `as_string` does not include all elements of the `prqueue` in the returned string
- `as_string` returns an improperly formatted string
- `peek` does not return the correct value that is in the front of the priority queue
- On some trees, `dequeue` does not return the correct value that is currently saved in the front node
- `dequeue` does not properly update the size of the `prqueue`
- On some trees, `dequeue` does not remove the node from the tree
- On some trees, `dequeue` loses part of the tree when the node is removed

## Task: Implementing Using

The functions in this section (and most, if not all of the remaining functions in the project) will be implemented recursively. **You can and should add private recursive helper functions in `prqueue.h`.**

Implement the two `Using` functions:

6. `as_string`
7. `peek`
8. `dequeue`

For this section, we strongly recommend adding private helper functions.

If you're getting a compilation error "error: passing `const...` as `this` argument discards qualifiers", you're probably calling a helper function that is not declared `const`.

You'll need to change the helper's declaration to make it `const`:

```
void _constRecursiveHelper(NODE* node) const {  
    // ...  
}
```

See Lab 9's `serialize` function for suggestions on implementing `as_string`.

Your code must not crash for `peek` and `dequeue` on an empty priority queue. Since these functions return values instead of pointers, we can't return `nullptr`. Instead, return the default value for the template parameter (`return T{};`).

## Grading Breakdown

This project is graded based on milestones. To complete a milestone, you'll need to both:

- Complete **every** milestone before it, and
- Pass *all* of its tests without any memory errors or leaks.
  - For testing milestones, this means writing tests that pass on a correct solution and catch all broken solutions.

Milestone	Total Points
<code>prqueue</code> Core Testing	8
<code>prqueue</code> Core Implementation constructor, <code>size</code> , <code>getRoot</code> , <code>enqueue</code> , <code>clear</code> , destructor	20
<code>prqueue</code> Using Testing	28
<code>prqueue</code> Using Implementation <code>as_string</code> , <code>peek</code> , <code>dequeue</code>	40

Note that Part 1 is worth a total of 40 points and Part 2 will be worth 60 points.

**Once you have completed the above, you are done with Part 1.**

Part 2 is on the "Part 2" tab in Google Docs, see the left sidebar. We strongly encourage starting on it early, but you should not start it until you have completely finished Part 1. It is due 1 week after the Part 1 deadline.