Project 4 - Priority Queue (BST)

CS 251, Spring 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 being submitted.
  + You may modify prqueue.h only to add additional **private** member functions. You may not add additional member variables, or additional public member functions.
  + You may not modify prqueue\_solution\_stub.h. This is only on zyBooks, and is only for compiling against the solution on zyBooks.
* The same memory restrictions as Project 3 apply. That is, memory errors will result in a 0 score on the test suite that hits them; and memory leaks are a flat score item.
* You will need to use classes, pointers, and new. Do not use malloc or NULL, we’re not writing C.

## Logistics

This project is larger, and runs through spring break. As such, we are splitting the project into **two parts**. The first part will be due before break, and the second part will be due after break. The two parts, together, are worth 1 project in the overall course grade. **Part 2 cannot be completed without Part 1.**

### Part 1 (35 points)

Due:

* **Gradescope: Friday 3/15**
  + prqueue.h
  + prqueue\_tests.cpp
* Grace tokens: <https://docs.google.com/forms/d/e/1FAIpQLSe7BdITKoqw6blYIUMVznjA1aPEAIRAZ5GkF6TS70pR85rU2w/viewform>
  + **Grace tokens should be requested by 5 PM the day of your current deadline.** For example, if you intend to submit the project by 11:59 PM on Saturday 3/16, you must submit the form by 5 PM on Friday 3/15. If you submit later, you will need to wait until we process it to be able to receive autograder feedback from Gradescope.
    - This is especially important because Ethan will be on break and responding less frequently.
  + This requires setting up a UIC Google account. If you have not yet done so, visit <https://learning.uic.edu/resources/virtual-collaboration/google-workspace/>.
* **Remember that course staff will be on break as well!** There will not be office hours over break; and Piazza response times will be significantly longer.

### Part 2 (65 points, Part 1 is a required prerequisite)

Due:

* **Gradescope: Tuesday 4/2**
  + prqueue.h
  + prqueue\_tests.cpp
* Grace tokens (will be created after break)

## Testing

We will continue to require you to write tests for your data structure. We’ll be taking the same approach as Project 3: we have many buggy implementations, and you’ll be writing tests that expose these buggy implementations!

You’ll receive credit for each buggy implementation that fails your tests. This will happen when you submit to Gradescope. Keep in mind that the correct implementation must pass your tests to receive any credit – no writing EXPECT\_TRUE(false), for example. To aid you in checking your own test cases, we’ve provided a solution “object file” in zyBooks: prqueue\_solution.o. This object file provides instantiations of the solution implementation for prqueue<int> and prqueue<string>.[[1]](#footnote-0) You will be able to test with values of those types, but not others.

In zyBooks, use make run\_solution\_tests to run your tests on the course staff’s correct solution.

In Part 1, the tests skip numbers. This is intentional, some of the broken solutions rely on behavior from Part 2.

Memory Safety & valgrind

The same requirements apply to this project as for Project 3: your programs may not have any errors reported by valgrind. We aren’t managing pointers as values this time, but we still need to be careful with the nodes themselves.

## 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). We also observed that this 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](https://en.wikipedia.org/wiki/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, 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!

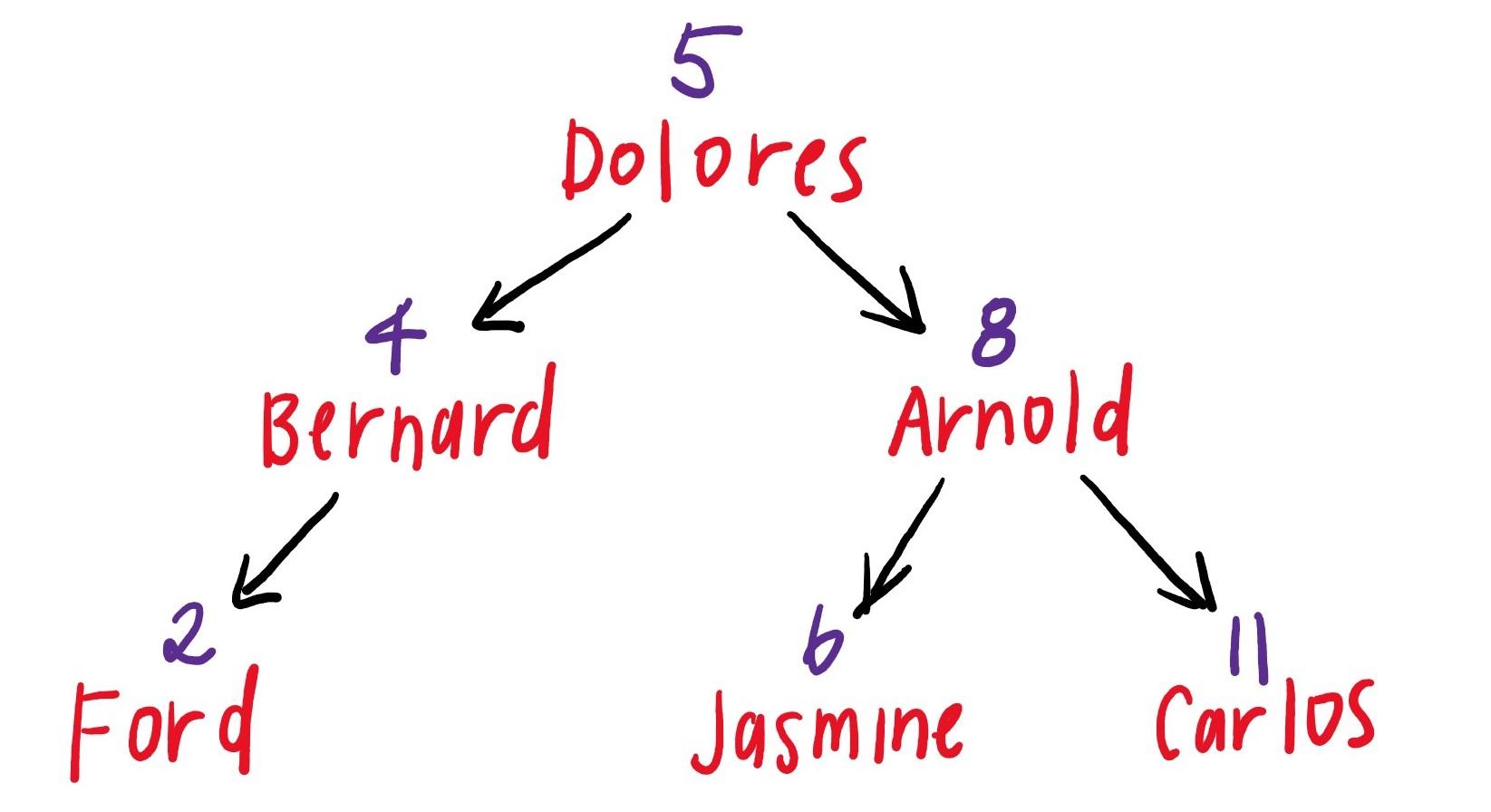
### Binary Search Tree as Priority Queue

The priority queue ADT can be implemented in many ways: an array, a vector, linked list, etc. 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:



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

The order in which elements would be dequeued is: Ford, Bernard, Dolores, Jasmine, Arnold, Carlos.

## 

## Part 1

### 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 create a prqueue similar to a vector, by providing the template parameter:

prqueue<int> pq;

We can use values of 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.[[2]](#footnote-1)

### Task: Testing

In Project 3 (and 2), course staff used STL containers to help write their tests, such as [deque](https://cplusplus.com/reference/deque/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.

In this project, we highly recommend using STL containers (vectors and maps) to help write your tests. One way of viewing 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.
* Compare the string that results from the map to the actual as\_string.

template <typename T>

string map\_as\_string(const map<T, vector<int>>& m) {

ostringstream ss;

for (const auto &e : mp) {

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));

}

### Internal Structure

The NODE struct you will use to represent your BST looks 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 might have seen before:

* Each node carries 2 pieces of data:
  + Priority (for BST organization and priority queue ordering)
  + Value
* Each node, in addition to its children, points to its parent.
* link is for Part 2, when we will handle duplicates. You can ignore it until then.

The 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 zyBooks HW 9 // Chapter 7.

### Task: Implementation

Many of your functions will be implemented recursively. **You can and should add private recursive helper functions in prqueue.h.**

Throughout Part 1, you can assume that no duplicate priorities are 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.

Implement the following functions. We recommend implementing them in this order:

1. Constructor, size, enqueue
2. peek, dequeue
   * Your code must support peek and dequeue on an empty priority queue. Since these return values instead of pointers, we can’t return nullptr. Instead, return the default value for the template parameter (return T{};).
   * Consider adding a private helper function to reduce code duplication.
3. as\_string
   * See the documentation in prqueue.h for details on the format. **Don’t call std::to\_string**, just feed the value directly into an output stream.  
       
     One way to build a string recursively is to use an ostringstream. In our recursive helper function, in addition to the NODE\*, we can add an argument ostream (of which ostringstream is a derived class):  
       
     void \_recursiveHelper(NODE\* node, ostream& output);  
       
     Inside this function, we can add data or endl to the ostream as follows:  
       
     output << "string" << endl;  
       
     Because output was passed by reference, we’re always adding to the same ostream! To get a string out of this, we use the str function: output.str();.
4. clear, destructor

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 your declarations to make them const:

void \_constRecursiveHelper(NODE\* node) const {

// ...

}

### 

### Grading Breakdown

Throughout, the prqueue will not have 2 keys with the same priority at the same time.

|  | **Points** |
| --- | --- |
| prqueue testing (catching bugs in broken implementations; tests must pass for a correct solution to receive credit) | 6 |
| Default prqueue constructor, size, enqueue | 8 |
| peek, dequeue | 8 |
| as\_string | 5 |
| clear | 3 |
| No valgrind errors or memory leaks (destructor + general hygiene); passes at least one prqueue test. | 5 |

**Total: 35**

There is no manual grading component for Part 1, but be kind to your post-break self. You’ll be building on the same code, so we recommend leaving meaningful comments to help you understand your own code after a week.

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

Part 2 is below. You can get started on it if you want, but it’s not due until after break (but is also due rather quickly after break).

## 

## Part 2

### 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:

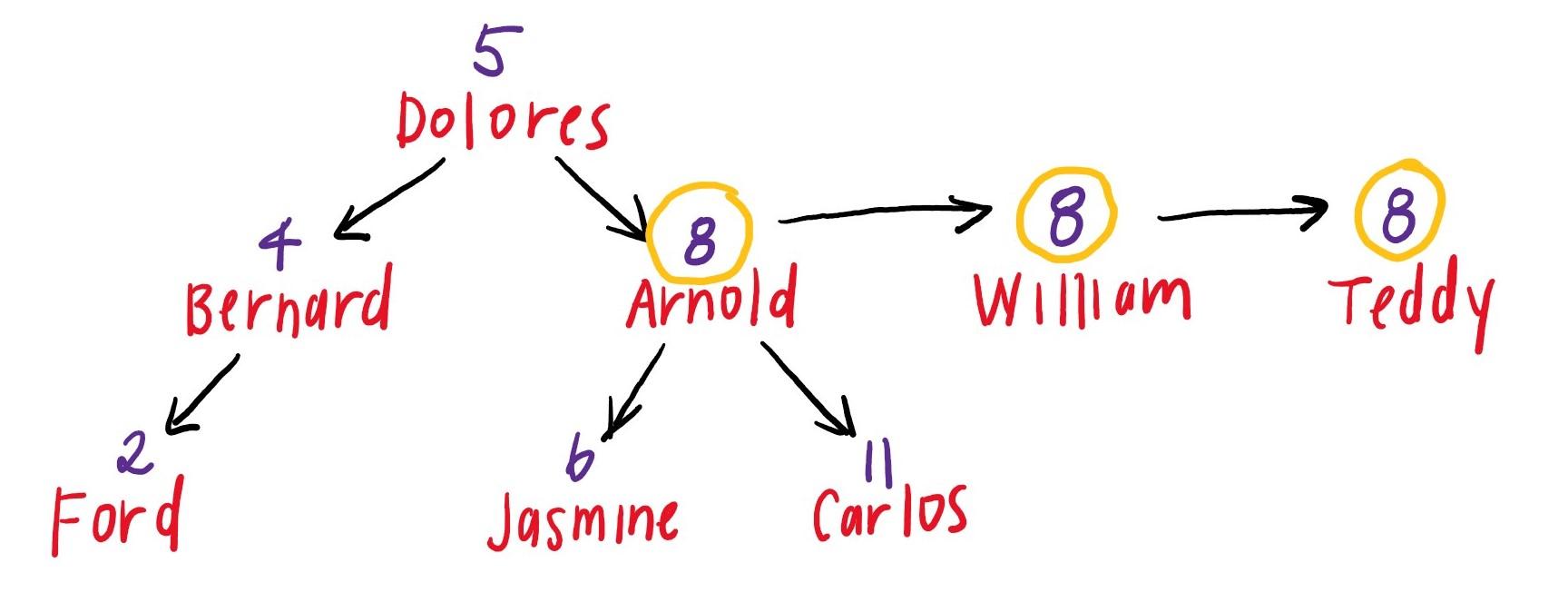
* "Dolores" with priority 3
* "Bernard" with priority 2
* "Arnold" with priority 4
* "William" with priority 3
* "Teddy" with priority 3
* "Ford" with priority 1

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.

### Task: Testing with Duplicates

Atr this point, you should write additional tests to test handling duplicates.

### Internal Structure

There’s more pointers, so the autograder does some more complex checks:

* 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 of the last node in each linked list is not checked.
  + The other pointers for these remaining nodes are not checked. You can use them however you see fit to handle duplicates, or ignore that they exist. This is a design decision for **you** to make!

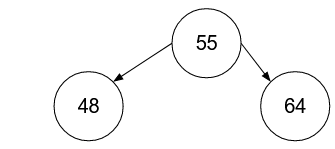
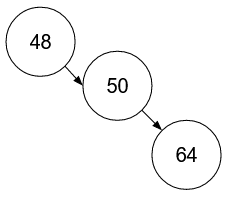
### Task: Implementation with Duplicates

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

1. enqueue
2. as\_string, clear + destructor
3. dequeue

### Task: Additional Implementation

Finally, we have some new functionality. Implement the following member functions, making sure to handle duplicates. We recommend implementing them in this order:

1. Copy constructor, operator=
   * Make sure to copy the exact tree structure, not just the values and priorities.
2. operator==
   * 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:  
        
       
     Even though they have the same priorities/values, the tree structure is different.
3. begin, next
   * **next is the most difficult function in the project. Plan your time accordingly.**
   * 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!  
       
     Instead of doing that, we’ll make an *iterative* version of the in-order traversal. This is sort of like a for loop: we use begin to start the in-order iteration, and next to advance the in-order iteration.  
       
     Given a node, how do you find the next node in an in-order traversal? Here’s one algorithm for a standard BST, given the current node:  
     + If the right subtree exists:
       - The next inorder node is the node with the smallest priority in the right subtree.
     + Else:
       - The next inorder 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 inorder node.
   * The algorithm as described is incomplete in 2 ways:
     + It will need to be modified for our BST with duplicates.
     + It does not say how to tell when it has reached the end.
   * 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.
   * 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

|  | **Points** |
| --- | --- |
| More prqueue testing, with duplicates and new functions (catching bugs in broken implementations; tests must pass for a correct solution to receive credit) | 10 |
| Handling duplicates in Part 1 functions (enqueue, as\_string, clear, destructor, dequeue) | 15 |
| Copy constructor, operator= | 10 |
| operator== | 5 |
| begin, next | 15 |
| No valgrind errors or memory leaks (destructor + general hygiene); passes at least one Part 2 prqueue test. | 5 |
| Documentation + style (manual, see below) | 5 |

**Total: 65**

**Style**

* 2 points: Code is styled consistently; for example, using the VSCode formatter.
  + (F1, type in “Format Document”)
* 1 point: Code is reasonably styled, but there are consistent significant stylistic issues (e.g. inconsistent indentation, line length > 120, spacing, etc.)
* 0 points: No credit (e.g. entire program is on one line)

**Documentation + Commenting**

* 3 points: Code is well-documented with descriptive variable names and comments, but not overly documented.
* 1.5 points: Code is partially documented, due to a lack of comments and/or poor naming; or code is overly documented with unnecessary comments.
* 0 points: Code has no documentation or appropriate names.

## Extra Challenge (Not for credit)

**If you want a really, really hard challenge, here you go!** You won’t submit this, but if you do it, please let us know, we’d love to see. Note that doing this will break the BST autograder tests because they verify BST tree structure.

Since we’ve implemented a BST, we’re actually most of the way to implementing an AVL tree! To complete an AVL implementation, you’ll need to modify the NODE\* to contain its height. Then, modify enqueue to track the node heights and rebalance the tree with AVL rotations. This guarantees O(log N) runtime on most of our queue operations by capping the tree height! (Ignoring duplicates…)

dequeue isn’t particularly instructive to write in an AVL context (and is frankly quite difficult), so feel free to skip that.

1. For technical reasons related to template classes and .h files, along with the need to hide the solution implementation, the solution compilation is rather hacky. [↑](#footnote-ref-0)
2. We’re omitting a lot of details here. If you’re curious, see zyBooks Ch. 31, or [LearnCPP 13.11](https://www.learncpp.com/cpp-tutorial/class-templates/). “Any type” is inaccurate – it needs to be copy-constructible and support <<, but “any” is close enough. [↑](#footnote-ref-1)