Project 5 – Priority Queue

CS 251, Spring 2023

Collaboration Policy

By submitting this assignment, you are acknowledging you have read the collaboration policy in the syllabus for projects. This project should be done individually. You may not receive any assistance from anyone outside of the CS 251 teaching staff.

Late Policy

You are given the grace period between the deadline and the first lecture of the following day (9am). You do not need to let anyone know you are using this grace period. Beyond this period, no late submissions will be accepted.

What & Where to Submit

1. Gradescope - priorityqueue.h and tests.cpp

Do not submit any additional files.

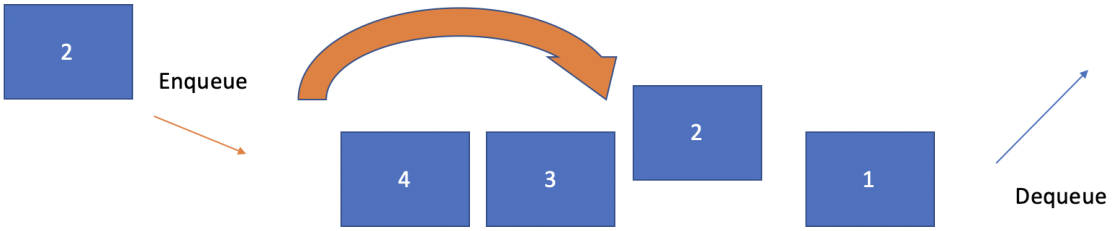
Table of Contents

Copyright Notice

Project originally crafted by Shanon Reckinger, University of Illinois Chicago.

Project 5 – Priority Queue

CS 251, Spring 2023



Project Summary

Queues process elements in a first-in, first-out (FIFO) order. But FIFO is not the best order for all problems. For example, the problem of assisting patients in a hospital, in this case, you will want to service the patients who have the most critical injuries before others with less serious needs. In this project, you will write an abstraction class for a priority queue.

For the priority queue class that you will write, you can imagine using it for a patient queue at a hospital. As each new patient checks into the hospital, the staff assesses the patient's injuries and gives them an integer priority rating. Smaller integers represent greater urgency, for example a patient of priority 1 is more urgent should receive care before a patient of priority 2. Once a doctor is ready to see a patient, the patient with the most urgent priority (i.e. with the lowest number rating) is seen first. However, you will be writing a more general purpose abstraction class. Therefore, it will work more generally for many types of priority queues.

The basic design of a priority queue is that, regardless of the order in which you add/enqueue the elements, when you remove/dequeue them, the one with the lowest priority number comes out first, then second-lowest, and so on, with the highest-numbered (i.e. least urgent) item coming out last. In your implementation of the priority queue, if two or more elements in the queue have the same priority, you should break ties by choosing the element who was placed in the queue first. This means that if an element is placed in the queue at priority K and there are already other elements in the queue with priority K, your new element should be placed after them. For example, imagine you have a patient queue for a hospital and patients arrive in this order:

- "Dolores" with priority 5
- "Bernard" with priority 4
- "Arnold" with priority 8
- "William" with priority 5
- "Teddy" with priority 5
- "Ford" with priority 2

If you were to dequeue the patients to process them, they would come out in this order: Ford, Bernard, Dolores, William, Teddy, Arnold.

A priority queue can internally store its elements in sorted or unsorted order; all that matters is that when the elements are dequeued, they come out in sorted order by priority. We will be specifying a very specific implementation in this project that you most adhere to. However, it is important to emphasize that abstraction classes can be implemented in lots of different ways.

Project 5 – Priority Queue

CS 251, Spring 2023

Starter Code

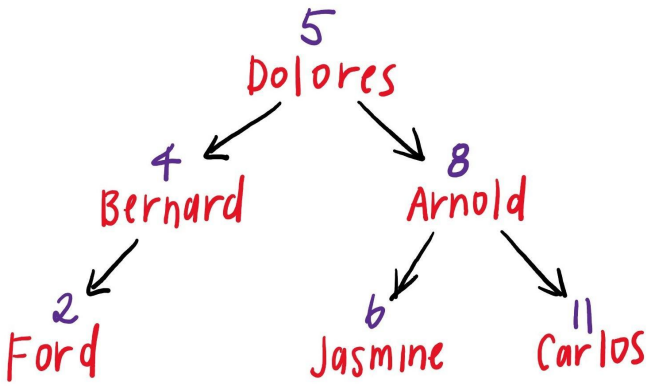
Priority Queue - Implementation Details

"Under the hood" of the priority queue abstraction could be a C array, a linked list, a vector, and many other options. We are going to implement a custom binary search tree. Later this semester, we are going to learn about a heap data structure (which, by the way, has nothing to do with the heap portion of memory). A priority queue is best and most naturally implemented with a heap. But we like to do things that are more painful (for learning). So, instead, we will implement a custom binary search tree.

For example, if add these elements to the queue:

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

Note that the BST is being created based on the priority number not the value, and the tree is:



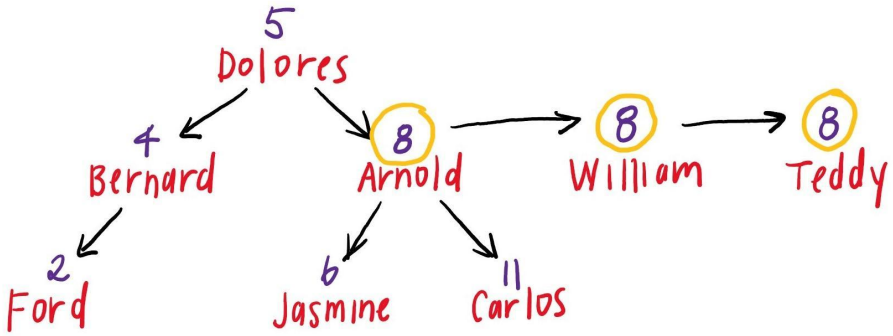
You might be wondering, how do we use a BST if we have priority ties? We know by now that BST's require unique values in order to stay in sorted order. Therefore, let's say we add these elements to the queue:

- "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

Project 5 – Priority Queue

CS 251, Spring 2023

We are going to implement our priority queue by adding a third dimension of linkages. Ahh! This is a visual of what it will look like:



The data structure you are going to use is a binary search tree. The node structure looks like this:

```
private:
    struct NODE {
        int priority;  // used to build BST
        T value;  // stored data for the p-queue
        bool dup;  // marked true when there are duplicate priorities
        NODE* parent;  // links back to parent
        NODE* link;  // links to linked list of NODES with duplicate priorities
        NODE* left;  // links to left child
        NODE* right;  // links to right child
    };
    NODE* root;  // pointer to root node of the BST
    int size;  // # of elements in the pqueue
    NODE* curr;  // pointer to next item in pqueue (see begin and next)
```

Therefore, your nodes keep track of more than a traditional BST node. First, keep in mind that you will build a BST based on the priority value (which is always an integer), but the node will also store the values (which are templated, so the priority queue can store integers, doubles, strings, etc.). Next, you will notice lots of node pointers. First, each node will link to its left and right children. Second, the nodes will link up to their parent. Finally, we will handle duplicate priorities by marking the nodes with dup assigned `true` and then creating a linked list using the link node pointer. In every function that searches, inserts, removes, or accesses the custom BST, make sure you take into consideration the duplicate linked list.

Project 5 – Priority Queue

CS 251, Spring 2023

The required parts of your tree design include:

- When we do an in-order traversal of your custom BST, we should find that each node has the correct **priority** and **value** when we navigate via the **left**, **right**, and **link** NODE*. The autograder does this so your tree must be built properly.

- Your linked lists must be set up in order, where the first node (the one linked to the tree) is the first duplicate element that was enqueued, the second node is the second duplicate element enqueued, etc.

- Your **parent** NODE* and **dup** members are not tested, so you are free to use those however you'd like. You will need them! But it is more fun to think about how to design those than for me to overspecify.

- When you are not using a NODE*, make sure you set it to ***nullptr***.

What about duplicate values? Are those allowed? Yes! No special case there, just allow them to be added, even with the same priority.

Getting Started

Overall, you should consider writing your functions first without considering duplicates and then test it thoroughly. Then, add in the linked structure to handle duplicates and test that next. It should be pretty straightforward to get things working for a priority queue with no duplicates, because the algorithms (and code) in many cases are often contained in the lectures. Therefore, testing that first will be a good first step.

If you need help getting started, here are some steps to take.

1. Write the **constructor**, **size**, and **enqueue** functions first. Write lots of tests. Remember that your priority queue is templated, so you will need to test on all primitive types. At this point, you will only be able to test that it compiles and keeps track of the size properly.

   Come up with lots of priority queues at this point. The simplest tests will be a set of unique priorities, but you should also test lots of duplicate priorities (make sure to test priorities that are duplicate in all spots in the priority queue/BST).

   Since the priority queue is a templated class, you will need to provide the type when you call the constructor, like this:
   ```
   priorityqueue<int> pq;
   ```

2. Write the **toString** method. The toString method will save the priority queue to a string. Please note that the string will have new line characters. You can do this a couple of ways. One is to use a stringstream and endl. Another way is to use a string, concatenation, and the new line character. Some people find it is confusing to recursively build a string. Instead, consider passing a stream recursively:
   ```
   void _recursiveFunciton(NODE* node, ostream& output);
   ```

You can then use the insertion operator in your recursion to fill the stream:

output << “string” << endl;

Note that when you call the recursive helper function (from toString()), you will pass in a stringstream type. You can then use one of the stringstream member functions to convert your stringstream object to a string.

For testing purposes, you might find it useful to pass “cout” in as your `ostream` instead of the stringstream object. Also, fun!

Once this is implemented, you should add toString() tests to all your tests.

Note that there should be a new line after each item in priority queue is added to the string (including the last one):

```
“1 value: Ben\n
2 value: Jen\n
2 value: Sven\n
3 value: Gwen\n”
```

3. Write the **clear** function. What is the appropriate traversal for deleting a BST? Once you determine that, remember that we have a custom BST, so some customization of the traversal will need to take place. After implementing, add to your tests and also begin your valgrind testing.

4. Overload the **equals operator**. Write and test!

5. Write the **begin** and **next** functions. This is a new idea for our classes. Read the comments carefully for begin and next and implement. You will want to test this thoroughly before moving on. Memory should not be affected by these two functions. The basic idea of these functions is that when you call begin your object should point curr to the first inorder node. Then, each time you call next it should return curr’s priority and value and then advance curr to the next in order node.

   Here is a potential algorithm for finding the next inorder node in a BST:
   - If the right subtree of the node is not empty, then the next inorder node lies in the right subtree. Go to the right subtree and return the node with the minimum priority in the right subtree.
   - If the right subtree is empty, then the next inorder node is one of its ancestors. Travel up the tree using the parent pointer until you see a node which is the left child of its parent. The parent of such a node is the next inorder node.

   Remember, we have a custom BST so some modifications of this algorithm will need to be made.

6. Write the **dequeue** function. This function will need to find the first node in the priority queue, return the value, and delete the memory associated with that node. Make sure to test this thoroughly (and on all types) before moving on. Also, make sure to run valgrind on all new dequeue tests. Make sure your code considers calling deque on an empty priority queue, however, you can decide how to handle it.

7. Write the **peek** function. This function will need to find the first node in the priority queue, return the value, but it will not delete the memory associated with that node. This function will be an easier version of dequeue. Consider code reuse setting up a helper function. Test, test, test! You can't really test peek unless you have dequeue set up.

8. Write the **equality operator**. Check out the traversal video for an example of how to write a code that determines if two BSTs are equal. You will need to modify this code to work for our custom BST. Don't forget to test priority queues that are and are not equal. Also, we will define equal priorityqueues as two priorityqueues are equal if their underlying custom BSTs are equal (same size, same shape, same values/priorities, etc).

9. Write the **destructor**. It is good to leave this for last in case you write it wrong. If you have an incorrect destructor, lots of bizarre segmentation faults and error messages can occur when writing the other pieces. Once you have implemented the destructor and the rest of your code is well tested, you should re-run valgrind or similar memory leak detection software on all your tests.

Testing

Spend some time developing scalable tests, using the Google Tests Framework or the Catch Framework.

It is really useful to use other containers to test the priority queue. You may even be able to use the C++ priority queue container! However, it is tricky to use so I might suggest using vectors and maps instead. For example, how can you test toString() on a priority queue with 100 items? You definitely don't want to hard-code that toString to test it. Instead, fill a map with the same items you are enqueuing. Then, write a function that can "toString" a map. This will also be super handy for testing dequeue, front, and next. You can iterate through your vector or map to ensure that you are getting what you intended. You might wonder how you can use a map to test a priority queue with duplicate priorities? You can create a map that maps priorities to a vector of values. Works really nice!

Project 5 – Priority Queue

CS 251, Spring 2023

Here is an example:

```
map<int, vector<int> > map;
int n = 9;
int vals[] = {15, 16, 17, 6, 7, 8, 9, 2, 1};
int prs[] = {1, 2, 3, 2, 2, 2, 2, 3, 3};
priorityqueue<int> pq;
for (int i = 0; i < n; i++) {
    pq.enqueue(vals[i], prs[i]);
    map[prs[i]].push_back(vals[i]);
}
EXPECT_EQ(pq.Size(), 9);
stringstream ss;
for (auto e: map) {
    int priority = e.first;
    vector <int> values = e.second;
    for (size_t j = 0; j < values.size(); j++){
        ss << priority << " value: " << values[j] << endl;
    }
}
EXPECT_EQ(pq.toString(), ss.str());
```

Additionally, consider what types of priority queues you should test. Does it matter what the values are? Should they be unique? What about priorities? When you determine that, it is easier to build lots of priority queues to test on. Consider adding the myrandom.h header file.

getRoot()

You will notice there is a public member function called getRoot(). It is already written for you and you do not need to edit it and you also do not need to call it. We are using it in the autograding scripts. We use it to make sure you have built the custom BST correctly. This will test that your tree has all the correct nodes (with the correct priorities and correct values). It will also check that each node has the correct left and right NODE* and correct link NODE*. It will not check anything else about the linked structure, so you have some flexibility in the design beyond that.

Requirements Recap

1. You may not change the API provided. You must keep all private and public member functions and variables as they are given to you. If you make changes to these, your code will not pass the test cases. You may not add any private member variables, public member variables, or public member functions. But you may add private member helper functions (and you should).
2. You must have a clean valgrind report when your code is run. Check out the makefile to run valgrind on your code. All memory allocated (call new) must be freed (call delete). The test cases run valgrind on your code.
3. Your tests.cpp should have 100s of assertions for each public member function. You should be placing significant effort into testing.
4. No global or static variables.
5. You can and should use lots of containers in your tests.cpp. You may also use other useful libraries like those that allow for random generations or file reading.