CS 271 Project 2 – Organized Heaps (Priority Queue Implementation)

Instructor: Flannery Currin

Due: October 1st, 2025 by 9:00 AM

1 Learning Goals

Heaps are often used to implement priority queues which are used in a wide range of applications from job scheduling to triage in emergency rooms. In this project, you will implement a templated priority queue as a min-heap and apply it to handle a waiting list for course registration. Specifically, after working on this project you should:

- Understand the key components making up a heap
- Know how to maintain a heap property and implement heapsort
- Understand how priority queues are used to maintain priority order as new elements are added and removed
- Be able to use heaps and priority queues with values other than primitive types

2 Project Overview

You should complete this project in a group as assigned on Canvas and in class. You are individually responsible for the learning goals above (dividing and conquering may not be the most effective strategy here). Use the Canvas guide on using Github to help manage version control. You may discuss this assignment with your partner(s), the course TA or instructor, but the work you submit must be your group members' own work. You may use your previous projects in this class as a reference for how to write a templated class and your first stop-gap project from CS 173 (dynamic array-based List).

Implement a MinQueue priority queue template class using a heap implemented with a private array member. We should be able to create a MinQueue of any type of element that can be compared with `<`, `>`, `==`, `!=`. Your class should have (at a minimum) the following two constructors:

- `MinQueue()`: empty constructor initializing an empty minimum priority queue (min heap)
- `MinQueue(T* A, int n)`: create a minimum priority queue (min heap) of the `n` elements in array `A`

Correct implementation of both constructors will be necessary to pass testing. *You will also need to implement a destructor to clean up dynamically-allocated memory.* Your MinQueue class should also implement (at a minimum) the following public methods:

- `insert(T x)`: `mq.insert(x)` should insert the element `x` into the priority queue `mq`. For example:

```
MinQueue<string> mq;
mq.insert(``hello'');
mq.insert(``world'');
```

  should result in the heap `[``hello'', ``world'']`

  N.B., for this project, we can skip setting $x.key$ to ∞ – it is possible to set some large value to represent ∞ if we know the specific type we are working with, but harder to find a good templated solution for this.

- `min()`: `mq.min()` should return the smallest value in the queue (throw an exception for an empty priority queue). For example, using the priority queue above:

  `cout << mq.min() << endl;`

  should print `` ``hello'' ``

- `extract_min()`: `mq.extract_min()` should remove and return the smallest value in the queue (throw an exception for an empty priority queue). For example, using the priority queue above:

```
string min = mq.extract_min();
cout << ``Removed: '' << min <<
     '', new min: '' << mq.min() << endl;
```

  should print `` ``Removed: hello, new min: world'' ``

- `decrease(int i, T k)`: `mq.decrease(i, k)` should decrease the value at index `i` to the new value `k`. For example:

```
MinQueue<int> mq2;
mq2.insert(7);
mq2.insert(9);
mq2.insert(2);
mq2.insert(8);
mq2.insert(3);
mq2.decrease_key(3, 1);
```

should result in the heap `[1, 2, 7, 3, 8]`

N.B., for this project, we can relax the check for $k < x.key$ to $k \leq x.key$ so we can skip setting $x.key$ to ∞ in `insert(T x)`.

- `min_heapify(int i)`: `mq.min_heapify(i)` should maintain the min-heap property by allowing the allowing element at index i to float down the heap so that, by the conclusion of the method, the subtree rooted at index i is a min-heap.
- `build_heap()`: `mq.build_heap()` should construct a min-heap from the potentially unordered values in the member array.
- `sort(T* A)`: `mq.sort(A)` should copy the values in the member array into array `A` in ascending order using heapsort. For example:

```
int* A = new int[10];
// populate A with integers
MinQueue<int> mq3(A, 10);
mq3.sort(A);
// print A
```

  should print the values in A in ascending order.

- `to_string()`: `mq.to_string()` should return a string representation of the items in the priority queue ***in the order in which the elements appear in the member array***. Use the `to_string()` method from `DoublyLinkedList.h` as reference.

You will also need to implement a public `set(int i, T val)` method which sets index `i` in the member array to `val` (update `q_size` iff `i` $\geq$ `q_size`) and an `allocate(int n)` method which ensures the member array has a capacity of at least `n`. **Your set and allocate methods will not be explicitly tested but will be necessary to pass unit testing for `min_heapify` and `build_heap`.** You may also find it useful to implement additonal helper methods like `clear()`, `parent(i)`, `left(i)`, etc.

Create a `MinQueue.h` file with these method declarations. You are to add any additional members necessary to `MinQueue.h` (the member array, etc.) and implement these methods in `MinQueue.h` or `MinQueue.cpp`.

Create unit tests for these methods that looks at boundary cases, edge cases, and disallowed inputs. An example test file `test_minqueue_example.cpp` has been provided and demonstrates (1) a general outline of what is expected in a test file and (2) a guide on how your projects will be tested after submission. The tests included in test minqueue example.cpp are not exhaustive. The unit testing in your submitted `test_minqueue.cpp` file should be much more complete. Additionally, for grading purposes, your code will be put through significantly more thorough testing than what is represented by `test_minqueue_example.cpp`. Passing the tests in this example file should be viewed as a lower bound.

In `test_minqueue_example.cpp`, you will notice a definition for a `StudentRecord` struct. Use the provided `StudentRecord` struct to populate a waitlist for course registration using a `MinQueue<StudentRecord>` to get a sense of how priority queues are used in real systems. Come up with at least one (1) of your own priority queue applications and implement a test for that application (a simplified representation of a real-world problem is fine). Think about situations in which a priority queue may be useful and situations where it may not appropriately capture certain nuances of a problem. If you find this type of question interesting, check out this dynamic priority case study and/or podcast episode.

3 Project Submission

On Canvas, before the deadline, you will submit **a zip** containing:

- Your `MinQueue.h`.
- Your `MinQueue.cpp` with your method implementations (if separate from `MinQueue.h`)
- `minqueue_test.cpp` with your unit tests and use-case practice
- A makefile to compile your project
- Any additional files you created that are needed to compile your project using your makefile.

4 Grading Scheme

The out-of-class submission will be graded using the following scheme:

Criteria	**Description**	**Points**
Completeness	Meets submission requirements (files, documentation, etc.)	2
Correctness	Passes Dr. Currin's extended tests and follows specs	4
Usecase	Contains required components and thorough testing	2
Testing	Tests should cover a range of types and cases, and logic should be sound	2
Total		**10**

The in-class component is worth 20 points. You should not need to study extra or memorize your out-of-class work for the in-class component. Working on the out-of-class component with the learning goals in mind is your best preparation. The in-class project component lets you test how prepared you are to apply the concepts covered by the project to a new scenario.