CS271: Data Structures

Instructor: Dr. Stacey Truex

Project #2

Submission

This project is meant to be completed in groups. You should work in your Unit 1 groups. Implementation solutions should be written in `C++`. Only one submission (the last submission uploaded to canvas) will be graded per group. Submissions should be a compressed file following the naming convention: `NAMES_cs271_project2.zip` where `NAMES` is replaced by the first initial and last name of each group member. For example, if Dr. Truex and Dr. Law were in a group they would submit one file titled `STruexMLaw_cs271_project2.zip`. **You will lose points if you do not follow the course naming convention**. Your `.zip` file should contain a *minimum* of 6 files:

1. `makefile`
2. `minqueue.cpp`
3. `test_minqueue.cpp`
4. `usecase.cpp`
5. `main.cpp`
6. `commits.pdf`: a commit history for your GitHub project

Additional files such as a `minqueue.h` header file or `README.md` are welcome. The above merely represent the minimum files required for project completion. Details for each are as follows.

Specifications

Priority Queue

Implement a `MinQueue` priority queue template class **using heaps**. 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 `A`

Correct implementation of both constructors will be necessary to pass testing.

MinQueue class methods

In addition, recall that min-priority queues support (at a minimum) the following operations:

- insert(`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"]`

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

```
cout « mq.min() « endl;
```

  should result in the printing of the string `"hello"`

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

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

  should result in the printing of the string `"Removed:  hello, new min:  world"`

- decrease_key(`i, k`): `mq.decrease_key(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]`

Heap methods

A min-heap should also support (at a minimum) the following operations:

- heapify(`i`): should maintain the min-heap property by 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. Your heapify method should be accessible via `min_heapify` in your `MinQueue`.

- build_min_heap(): should produce a min-heap from the potentially unordered values in the member array. Your build_min_heap method should be accessible via `build_heap` in your `MinQueue`.

- heapsort(`A`): should result in the array `A` containing the values in the min heap member array in *ascending* order. Your heapsort method should be accessible via a `sort` method in your `MinQueue`. For example:

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

should result in the printing of the values in `A` in *ascending* order.

Be sure to include suitable preconditions and postconditions in the comments before each method. Code implementing your `MinQueue` should be written in the file `minqueue.cpp`.

Unit Testing

In addition to the functionality above, you are expected to implement a `to_string()` method which returns a string with the elements in your priority queue *separated by a single space* in the order in which they appear in the member array. For example, using the queue `mq2` generated in the above specifications:

```
cout << mq2.to_string() << endl;
```

should result in the printing of the string `"1 2 7 3 8"`. Your `to_string()` method will be required for your class to pass testing.

Finally, you will need to implement a method `set(i, val)` method which sets index `i` in the member array to `val` and a method `allocate(n)` 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 `heapify` and `build_min_heap`.

For each method you write, you should also be writing a unit test method in a separate unit test file that *thoroughly* tests that method. Think, in addition to common cases: what are my boundary cases? edge cases? disallowed input? Each method should have *its own* test method. Note that your test file will be run against other class submissions.

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 `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.

Documentation

The expectation of all coding assignments in this course is that they are well-documented. This means that logic is documented with line comments and method pre- and post- conditions are properly documented immediately after the method's parameter list.

Pre-conditions and post-conditions are used to specify precisely what a method does. However, a pre-condition/post-condition specification does not indicate how that method accomplishes its task (if such commenting is necessary it should be done through line level comments). Instead, pre-conditions indicate what must be true before the method is called while the post-condition indicates what will be true when the method is finished.

Usecase

Finally, use your `MinQueue` to solve the following problem:

You are given an array of integers `nums`, there is a sliding window of size `k` which is moving from the very left of the array to the very right. You can only see the `k` numbers in the window. Each time the sliding window moves right by one position. Return the min sliding window.

Example 1

Input: `nums = [1,3,-1,-3,5,3,6,7]`, `k = 3`
Output: `[-1,-3,-3,-3,3,3]`
Explanation:

Window Position	Min
[1 3 -1] -3 5 3 6 7	-1
1 [3 -1 -3] 5 3 6 7	-3
1 3 [-1 -3 5] 3 6 7	-3
1 3 -1 [-3 5 3] 6 7	-3
1 3 -1 -3 [5 3 6] 7	3
1 3 -1 -3 5 [3 6 7]	3

Example 2

Input: `nums = [1]`, `k = 1`
Output: `[1]`

Your solution should be implemented in `usecase.cpp` in the function

```
string sliding_window(T arr[], int len, int window)
```

where `len` is the length of the array `nums` and `k` is the size of the sliding window. Your `sliding_window` function should return a string with the values in the output of the sliding window problem separated by a single space. For example, the above examples would return the strings `"-1 -3 -3 -3 3 3"` and `"1"` from `sliding_window` respectively. In your `main.cpp` file, your `main` function should include at least one example test case demonstrating the accuracy of your usecase solution. Note that your use case will only be tested when the template is set to `int`.

Makefile

With each project you should be submitting a corresponding makefile. Once unpacking your `.zip` file, the single command `make` should create a `test` executable **and** a `usecase` executable. The command `./test` should then run all the unit tests in your `test_minqueue.cpp` file evaluating your `MinQueue` class. The command `./usecase` should run the example test case in your `main.cpp` file demonstrating the accuracy of your sliding window solution.

Efficiency

Each project in this course will additionally be evaluated for efficiency. Each method detailed in the MinQueue class methods section of this document will be called 1 time using a very large example. The total time to execute all methods will be clocked.

Rubric

Note that any coding projects that do not compile with the provided `test_minqueue_example.cpp` file will be given a 0. All projects that are able to be successfully compiled will be graded using the following rubric.

C++ Implementation			
	does not compile: 0/40		
	40 Total Points		
	Code	**Completeness** met submission requirements	10 pts
		Correctness passes unit testing	17 pts
		Validation implementation deductions ex: member array is public	——
	Usecase	**Correctness** passes unit testing	5 pts
	Efficiency	**Time Test**	2 pts
		encountered error - could not complete time test	0/2
		takes over 2x fastest submission	1/2
		within 2x fastest submission	2/2
		fastest submission	3/2
	Documentation	**Documentation**	3 pts
		extremely sparse documentation	0/3
		missing comments or pre- and post-conditions	1/3
		documentation lacks detail in areas	2/3
		detailed comments & pre- and post-conditions	3/3
	Testing	**Unit tests**	3 pts
		does not expand on example test file	0/3
		not all functions tested *or* testing not implemented as unit testing *or* no variation in templates	1/3
		caught some of the bugs in classmates' code	2/3
		caught most bugs in classmates' code	3/3