Lab Exercise #5
Due Date: Mar 20th 2024
Priority Queues & Heaps

Description:
For this lab exercise, you will implement a **Priority Queue.** Heaps are specialized data structures where each parent is greater or equal to (for a max heap), or smaller or equal to (for a min heap) than its children. They are commonly used to implement priority queues, efficient for extracting the maximum or minimum element in logarithmic time. Priority queues, built on heaps, prioritize elements based on their assigned priority, making them suitable for tasks requiring efficient retrieval of the highest-priority item.

starter.zip (Starter Code):
The following is a brief description of the starter code provided to you:

- **PriorityQueueHeap.h**
    - Declares the **PriorityQueueHeap** class which contains the following private member variables:
        - **Type* arr:** This is the pointer to the array which will hold the heap
        - **int capacity:** This variable will hold the capacity of the array
        - **int size:** This variable will hold the number of filled elements in the array
        - **Compare compare:** This variable will hold the Comparison function object which will determine whether the priority queue instantiated is a min or a max heap. The comparison function can be of two types:
            - std::greater<Type>
                - This is the default comparison function which is used to implement a **min heap**
                - An example use case when compare is std::greater<Type>
                    - compare(a, b) returns 1 if a > b and 0 if a <= b
            - std::less<Type>
                - This is used to implement a **max heap**
                - An example use case when compare is of std::less<Type>
                    - compare(a, b) returns 1 if a < b and 0 if a >= b
    - The PriorityQueueHeap class declares the following public methods:
        - **Default Constructor, Constructor using Comparison Function, Copy Constructor, Copy Assignment, Destructor**
        - **int pq_size():** returns the number of elements in the heap (this is not the capacity of the array).
        - **bool is_pq_empty():** returns true if the priority queue is empty and false otherwise

        - **Type pq_top():** returns the minimum or maximum element (depending on whether the object is a min heap or a max heap). If the priority queue is empty, it throws a std::out_of_range exception.
        - **void pq_insert(Type x):** inserts the element x onto the heap such that the heap property is not violated. When the number of elements in the heap is equal to the capacity of the array, you must resize the array such that it doubles in capacity to accommodate additional elements on the heap.
            - The pq_insert method will use the private bubbleUp and pq_parent helper functions to implement its functionality
        - **Type pq_delete():** removes and returns the minimum/maximum element of the heap (depending on whether the object is a min heap or a max heap) such that the heap property is not violated. If the priority queue is empty, it throws a std::out_of_range exception.
            - The pq_delete method will use the heapify helper method to implement its functionality
- **main.cpp**
    - You can use this file to instantiate PriorityQueueHeap objects and test your code.

Contract (TASK):
Your task is to implement the methods of the PriorityQueueHeap class. You have been provided with non-functional definitions of these methods. **You must implement these method definitions in the given header file,** by replacing the current non-functional definitions with your functional implementations. **Do not create additional cpp files to define these functions, make sure that they are part of the header files.**

[Advisory]:
- You may choose **10** to be the initial capacity of the array when allocating memory in the constructor
- You must throw a **std::out_of_range** exception when you come across invalid calls to the pq_top and pq_delete method

Deliverables and Submissions:

In a zip folder named using the format **<FirstName>-<LastName>-<UIN>-<LE5>.zip** you must submit the following file(s) to Canvas:
- **PriorityQueueHeap.h** (containing the functional method definitions of the PriorityQueueHeap class)

Do not use angular brackets in the name of the submission folder.

Grading:
Coding (20 points)

Each method of the PriorityQueueHeap will be tested using an automatic testing infrastructure. You will be provided with the testing infrastructure which you can use to evaluate your implementations.

The grading rubric is as follows:

- Each of the following methods will be tested out:
  - Copy Constructor: 2 points
  - Copy Assignment: 2 points
  - pq_insert: 7 points
  - pq_empty: 1 point
  - pq_delete: 7 points
  - pq_delete and pq_top exceptions: 1 point

In case of any memory leaks in your program, we will subtract 2 points.

testInfrastructure.zip (Test Methodology):
In order to familiarize you with the auto-grading environment on our end we are providing you with an automated test infrastructure that can be used to check the correctness of your code. The testing infrastructure is constructed as follows:

- **pq_test.cpp**: This cpp file defines all the test functions that have been mentioned in the grading rubric above. It also runs these tests and outputs your final score.
- **test.py**: This file compiles pq_test.cpp in order to produce the executable. It then runs this executable which will indicate your score (as well as the test cases you have passed or failed). It will also run valgrind on the executable file to indicate the presence of memory leaks if they exist.

Note: There is a possibility that one or more of your methods causes a segmentation fault which in turn causes the test script to crash when calling that particular method. In such a case, we will not be able to calculate your score for the assignment and you will be assigned a default grade of 0. After the grades are posted, you will be given 1 week to rectify your error(s) and resubmit the assignment. You will be subject to a 25% penalty on the grade obtained from the resulting resubmission of the assignment.

How to test your program natively (run your own test cases):
You can use main.cpp provided in the starter.zip folder to instantiate PriorityQueueHeap objects. You can then test the functionality of your methods by calling these methods and checking whether what they return is what you expect.
To compile your own tests in main.cpp, **you can use the makefile provided within starter.zip to compile your program**. In your terminal:

- Run **make** on the terminal (this will create an executable named main)
- Next, run **./main** on your terminal to run your native tests
- Finally run **valgrind --leak-check=full ./main** to check whether there are memory leaks.

How to use testInfrastructure:
To check your score using testInfrastructure, you can follow either of the two options mentioned below:

- **Option 1:** Move your PriorityQueueHeap.h file into the testInfrastructure directory. After this, run **python3 test.py**
- **Option 2:** Move your PriorityQueueHeap.h file into the testInfrastructure directory. **Ensure that the makefile in this directory is the one given to you originally in the testInfrastructure.zip folder and not in the starter.zip folder.** In your terminal:
  - Run **make** in order to compile the program (which will compile pq_test.cpp and create an executable named pq_test).
  - Next, run **./pq_test** on your terminal to actually run the tests and output your score.
  - Finally run **valgrind --leak-check=full ./pq_test** to check whether there are memory leaks.

Please also make sure that you run the test script a few times (3 - 4 times should be sufficient) to ensure that your code is consistently passing test cases in different random runs of the test script.