SAN DIEGO STATE UNIVERSITY

Department of Computer Science
CS 210 Data Structures

Program 1 Phase 2 Instructions

Student Objective

In this assignment, you will develop two implementations of a Priority Queue: one ordered and one unordered. Both implementations must adhere to specific criteria, ensuring removal operations consistently return the object of the highest priority that has been in the queue the longest. Additionally, the FIFO order must be maintained within each priority level.

This assignment will progress through three distinct phases. Phase 1 involves the implementation of two Array classes: `UnorderedArrayList` and `OrderedArrayList`. Following Phase 1, Phase 2 focuses on the development of a Priority Queue specific to each array class type. Finally, Phase 3 will entail conducting an analysis of your written code, evaluating aspects such as time complexity, space complexity, and overall clarity.

As you tackle this first program, remember it's your chance to grasp the basics of testing, finding edge cases, and mastering debugging. Embrace the opportunity to learn test-driven development—it's the perfect starting point for your journey in this course!

Phase 2 Instructions

You should have already read *Separate Compilation*. If you skip this step, you may struggle to understand how the program files work together.

Class prototypes defined in `UnorderedPQ.h` and `OrderedPQ.h` are provided. Both of these classes are subclasses of the pure virtual `PriorityQueue` class. You **must** review `PriorityQueue.h` for reference to understand the functionality of each method.

Phase 2 of the project will consist of the following files *in addition to phase 1 files.* You must use exactly these filenames.

`PriorityQueue.h`	The PQ ADT interface. (provided)
`OrderedPQ.h`	The ordered PQ definition. (provided)
`UnorderedPQ.h`	The unordered PQ definition. (provided)
`OrderedPQ.cpp`	The ordered array implementation.
`UnorderedPQ.cpp`	The unordered array implementation.

Your task is to write your own code for the public functions found in `OrderedPQ.h` and `UnorderedPQ.h`. You may not modify `PriorityQueue.h`.

Finally, this phase marks the culmination of the coding aspect of the project. Ensure that your code is thoroughly commented, well-formatted according to C++ standards, and adheres to best practices such as meaningful variable names and using camelCase.

Additional Details

- You may not modify any ADT interface.
- You may add private members as needed.
- You may add public functions strictly for debugging purposes. These must be removed before submitting.
- You may only submit source files specified in the instructions.
- All source code file must have your **name** and **RedID** at the beginning of the file.
- Each method must be as efficient as possible.
  - For example: $O(N)$ complexity is unacceptable if the method could be written with $O(\log N)$ complexity.
  - Hint: For the ordered array, certain methods must use binary search where possible.
- By convention, a lower number means a higher priority.
- Your implementations may only `#include` the following files/libraries: `"OrderedArrayList.h"`, `"UnorderedArrayList.h"`, and `<stdexcept>`. `<iostream>` may be used for debugging purposes but should be removed before submitting.
- You are expected to write all the code yourself, you may not use the Standard Library API for any containers.
- Your code must not print anything.
- Your code should handle all error conditions gracefully. It should never crash.
  - Follow the outline in the ADT interface which specify what exceptions to throw.
- Your code must compile and run correctly on Gradescope to receive any credit. Testing on edoras may prove to be a helpful exercise.
- Minimal tester/driver programs may be provided to help you test your code. You should add more tests and focus heavily on the edge cases.
- Allow sufficient time for testing on Gradescope.

Hints

- The implementation for `UnorderedPQ` and `OrderedPQ` should be done by calling methods from the phase 1 ArrayList classes you implemented. If you are rewriting the same/similar code from phase 1 then you're on the wrong track. Focus on how to use phase 1 objects to implement phase 2 functions with very little new code.
- If you are getting unexpected output, use `gdb` and step through your program. Failure to learn how to use debugging tools will result in many wasted hours.

Turning in Phase 2

To submit phase 2, you must upload the following C++ files to your `Gradescope` account through Canvas. The automated grading platform will run. You're permitted unlimited submissions, but *don't* use Gradescope as your testing platform. If we see this happening, we will enforce limited submissions.

```
OrderedArrayList.h

UnorderedArrayList.h

OrderedArrayList.cpp

UnorderedArrayList.cpp

OrderedPQ.h

UnorderedPQ.h

OrderedPQ.cpp

UnorderedPQ.cpp
```

Code Format and Comments

- Proper indentation:
  - Consistent use of spaces or tabs for indentation (typically 2 or 4 spaces).
  - Nested code blocks should be indented accordingly.
- Consistent use of braces:
  - Braces should be used consistently for code blocks, including if statements, loops, and functions.
- Descriptive variable and function names:
  - Variable and function names should be meaningful and descriptive of their purpose.
  - Use of camelCase for naming.
- Consistent naming conventions:
  - Follow consistent naming conventions for variables, functions, classes, and other identifiers.
- Proper spacing:
  - Use spaces around operators for readability.
  - Use consistent spacing within expressions and statements.
- Use of appropriate data types:
  - Choose appropriate data types for variables and functions based on their usage and requirements. (We are not concerned about memory usage, so do not worry too much about this one.)
- Avoidance of magic numbers and hardcoded values:
  - Use named constants instead of hardcoded values.
  - Magic numbers should be avoided or properly explained with comments.
- Proper use of comments:
  - Every file should have a header comment with your name and RedID. (Look at example provided in the appendix.)
  - Every function should have a leading comment. (Look at example provided in the appendix.)
  - Include comments to explain complex logic, algorithms, or non-trivial code blocks.
  - Comments should be clear and concise, adding value to the code understanding.

Cheating Policy

There is a zero tolerance policy on cheating in this course. You are expected to complete all programming assignments on your own. Collaboration with other students in the course is not permitted. You may discuss ideas or solutions in general terms with other students, but you must not exchange code. During the grading process I will examine your code carefully. Anyone caught cheating on a programming assignment (or on an exam) will receive an ”F” in the course, and a referral to Judicial Procedures.

Appendix

```
/**
* @file sample.cpp
* @author <Your Name> <Your RedID>
* @brief This is a sample program to show how to comment
* @date YYYY-MM-DD
*/
#include <stdio.h>

int main() {
    printf("Hello, world!");
    return 0;
}
```

Listing 1: Example Header Comment

```
/**
* Returns the Object at the specified position in this list
* @param index
* @return Object
* @throws invalid_argument if index out of range
*/
int get(int index) {
    function definition...
}
```

Listing 2: Example Function Comment