# COMP 330
# Lab III - Synchronization

Rhodes College

—1848—

In this project, you will write a C program that uses POSIX threads, mutex locks, and condition variables to implement a variation of the classic producer/consumer problem. You will use GitHub classroom to get the starter code and submit your work. Your work will be compiled and graded on `cslogin.arc.rhodes.edu`.

## Project Description

In this project, you will be implementing a thread-safe bounded priority queue data structure. The priority queue will be implemented as an array-based heap. The primary operations that you will need to write are the `enqueue()` and `dequeue()` methods.

The basic operation of the lab goes as follows: Initially, we set a fixed number of entries to be added to the priority queue. In an operating system, these could be events, processes, etc. There will be a number of producer threads and a number fo consumer threads. Each producer will create a new entry that is associated with a randomly-generated priority. If there is room in the queue, the producer will enqueue this entry into the heap, ensuring that the heap property is preserved after insertion. If the buffer is full, the producer will wait until space has been freed up before proceeding. Concurrently, consumers will attempt to dequeue the highest priority entry from the heap. If there are no entries in the queue, the consumer will wait.

Within an operating system, there is a tradeoff between wanting to use dynamically data structures that can grow and shrink depending on usage, versus statically alloated data structures that have a fixed size. Internally, statically allocated data structures are often more efficient, since they don't require memory allocation calls or pointer dereferencing. In this assignment, you will implement a binary-tree based *max heap* structure with a fixed size bounded array. This approach works well since there is a straightforward mapping between the tree structure and the array indices.

Recall that a heap has some important properties that must be preserved at all times:

1. The heap is a complete binary tree. This means that all levels of the tree are fully filled, except possibly for the last level, which is filled from left to right.

2. The heap property must be maintained. In a max-heap, for any given node `N`, the value of `N` is greater than or equal to the values of its children. This ensures that the maximum value is always at the root of the tree.

At the top of Figure 1, we see a max-heap with the priority values stored in each node of the tree. Since a binary tree with the root labeled as index 0, the left child of a node at index `i` is located at index `2*i + 1`, and the right child is located at index `2*i + 2`. From any arbitrary node at index `i`, the parent node can be found at index `(i-1)/2` (using integer division). The array representation of this heap is also shown in Fig. 1.

To enqueue a new entry, it is added to the next spot that would keep the tree complete (i.e. the next unused entry in the array). The entry has been added to the tree, but the overall heap property may have been violated. To resolve this, you will need to implement a `heapify_up()` function that will start at the new node and work towards the root, swapping values until the heap property is ensured. As seen in Fig 1, adding a new entry with priority 7 to the tree would require that it be swapped with both the 4 and 6 entries until the heap property is restored.

Similarly, to dequeue the maximum value, we will remove the root node, by swapping the root with the last node in the tree (the rightmost node at the bottom of the tree). This effectively removes the maximum value from the tree, but, again, the heap property may have been violated. To resolve this, you will need to implement a `heapify_down()` function that works from the new root and goes down until the heap property is restored. An example of this heapify operation with dequeuing the maximum value is shown in Fig 1.

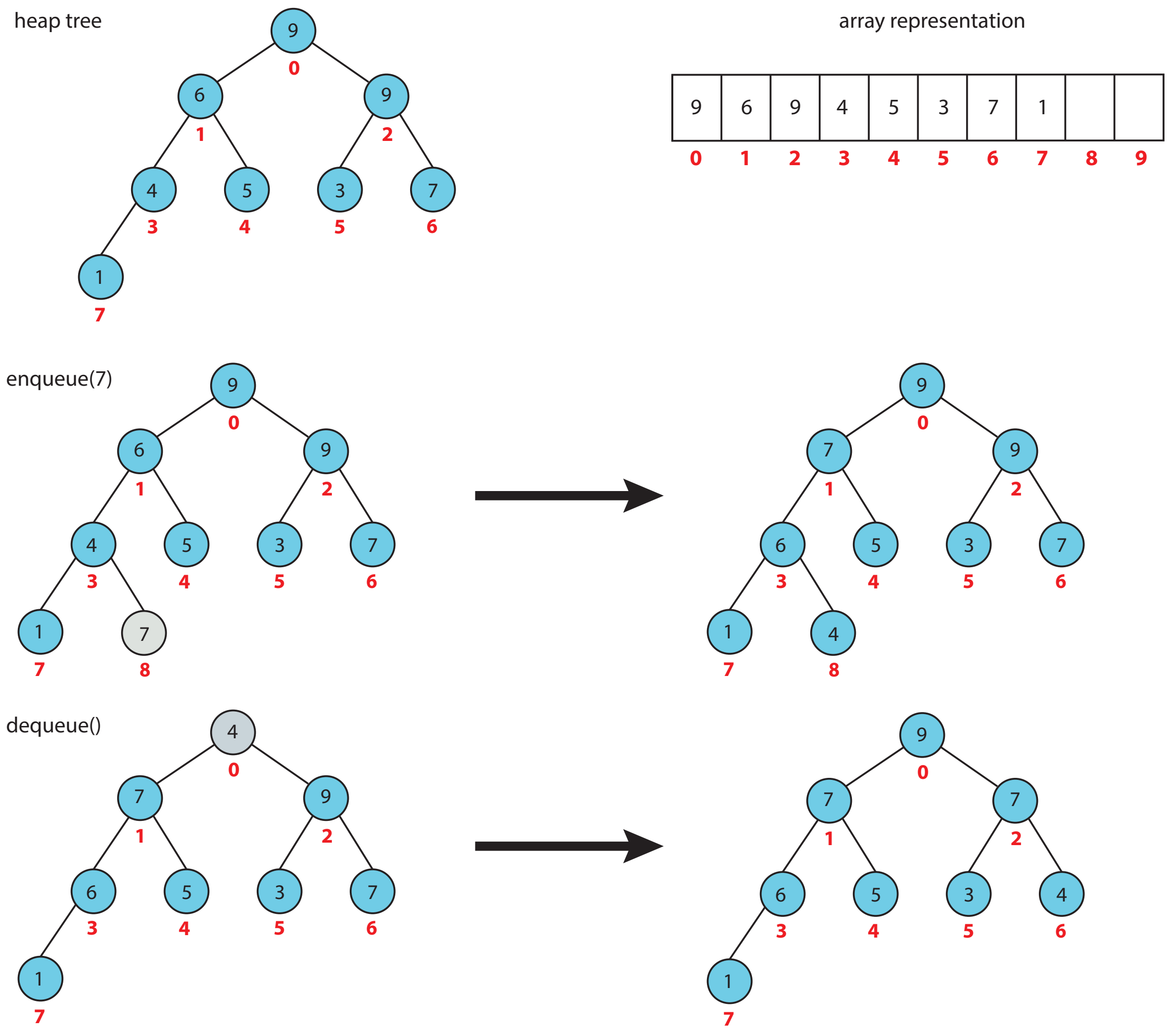

Figure 1: Heap Structure and Operations

## Synchronization

The tricky part of implementing this data structure is that access to the heap must be carefully synchronized. Since multiple threads will be accessing the heap, you must ensure that only one thread can modify the heap at a time. You will need to use mutex locks to protect access to the heap and condition variables to signal when the heap is not full (for producers) or not empty (for consumers).

When using condition variables, it is *critical* that you pair `wait()`/`signal()` actions with some other way to identify whether it is safe or not to proceed. There are two issues that can be troublesome with condition variables: First, if a `signal()` operation happens *before* the corresponding `wait()`, the signal will be lost. Second, sometimes it's possible for a thread to be awoken due to how process/thread scheduling is implemented. Your solution should be thread-safe for any number of producers/consumers, not just one of each or where they are symmetric.

## Implementation Hints

- You should implement the six functions marked in `priority_queue.c`.
- There is a `dprint()` function which can be used to add diagnostic messages. This is turned on by using the `-d`

flag to the program.

- Make sure to properly initialize and clean up your synchronization primitives and/or any dynamically allocated memory.
- There is program logic for logging queue operations to a CSV file. This may be helpful with debugging and can be enabled by the `-l` option.
- Several global environment settings are in a `global_t` structure, `g`. This is considered good form when using global variables. This contains things like the debugging status, number of producers/consumers, etc.
- Submit your work via GitHub, ensuring only to commit C source/headers and the Makefile.

## Sample Output

```
brian@cslogin $ ./lab3 -n 5 -p 2 -c 3 -q 10
    [Producer 0] Enqueued 88 (priority 6)
    [Producer 1] Enqueued 52 (priority 0)
    [Consumer 0] Dequeued 88 (priority 6)
    [Consumer 1] Dequeued 52 (priority 0)
    [Producer 0] Enqueued 26 (priority 6)
    [Consumer 2] Dequeued 26 (priority 6)
    [Producer 1] Enqueued 57 (priority 1)
    [Consumer 0] Dequeued 57 (priority 1)
    [Producer 0] Enqueued 1 (priority 4)
    [Consumer 1] Dequeued 1 (priority 4)
    [Producer 1] Enqueued 10 (priority 8)
    [Consumer 2] Dequeued 10 (priority 8)
    [Producer 0] Enqueued 70 (priority 8)
    [Producer 0] Enqueued 34 (priority 1)
    [Consumer 1] Dequeued 70 (priority 8)
    [Producer 0] Enqueued 39 (priority 3)
    [Producer 1] Enqueued 31 (priority 4)
    [Consumer 0] Dequeued 31 (priority 4)
    [Consumer 2] Dequeued 39 (priority 3)
    [Producer 0] Enqueued 80 (priority 2)
    [Producer 1] Enqueued 84 (priority 1)
    [Producer 0] Enqueued 83 (priority 9)
    [Consumer 2] Dequeued 83 (priority 9)
    [Consumer 0] Dequeued 80 (priority 2)
    [Consumer 2] Dequeued 84 (priority 1)
    [Consumer 1] Dequeued 34 (priority 1)
    [Producer 0] Enqueued 82 (priority 7)
    [Consumer 1] Dequeued 82 (priority 7)
    [Producer 1] Enqueued 57 (priority 8)
    [Consumer 0] Dequeued 57 (priority 8)
    [Producer 1] Enqueued 95 (priority 1)
    [Consumer 1] Dequeued 95 (priority 1)
    [Producer 0] Enqueued 55 (priority 0)
    [Consumer 2] Dequeued 55 (priority 0)
    [Producer 1] Enqueued 76 (priority 4)
    [Consumer 1] Dequeued 76 (priority 4)
    [Producer 0] Enqueued 26 (priority 5)
    [Consumer 2] Dequeued 26 (priority 5)
    [Producer 1] Enqueued 61 (priority 2)
    [Consumer 0] Dequeued 61 (priority 2)
    [Producer 0] Enqueued 16 (priority 9)
```

```
[Producer 1] Enqueued 50 (priority 8)
[Consumer 1] Dequeued 16 (priority 9)
[Consumer 0] Dequeued 50 (priority 8)
[Producer 1] Enqueued 72 (priority 4)
[Producer 0] Enqueued 19 (priority 5)
[Consumer 2] Dequeued 19 (priority 5)
[Consumer 1] Dequeued 72 (priority 4)
[Producer 1] Enqueued 58 (priority 6)
[Consumer 1] Dequeued 58 (priority 6)
[Producer 0] Enqueued 12 (priority 8)
[Producer 0] Enqueued 38 (priority 5)
[Consumer 2] Dequeued 12 (priority 8)
[Consumer 0] Dequeued 38 (priority 5)
[Producer 1] Enqueued 79 (priority 3)
[Consumer 0] Dequeued 79 (priority 3)
[Producer 1] Enqueued 22 (priority 9)
[Producer 1] Enqueued 81 (priority 5)
[Consumer 2] Dequeued 22 (priority 9)
[Consumer 0] Dequeued 81 (priority 5)
```