

# Performance Analysis of Stack, Queue, and Priority Queue Implementations

In this project, I implemented different versions of three abstract data types (ADTs): Stack, Queue, and PriorityQueue. Each ADT was built using different underlying data structures so I could compare how they perform in real programs.

The goal was not just to understand the theory behind these data structures, but to actually measure how fast they run. To do this, I created a benchmark program that timed how long different operations took. The benchmark included warmup runs, multiple trials, and a checksum to make sure the code was actually executing the operations.

## Implementations

### Stack

I implemented two versions of a Stack:

- One using an ArrayList
- One using a doubly linked list

Both stacks support push and pop in constant time ( $O(1)$ ). Even though they have the same theoretical time complexity, they are implemented differently in memory.

### Queue

I implemented two versions of a Queue:

- An ArrayList-based queue using a circular buffer
- A doubly linked list-based queue

The circular buffer avoids shifting elements when removing items from the front. This allows enqueue and dequeue operations to stay  $O(1)$ .

### PriorityQueue

I tested three versions of a PriorityQueue:

- A sorted ArrayList implementation
- A sorted doubly linked list implementation
- A binary heap implementation

The sorted versions keep elements in order at all times, which makes removal fast but insertion slower. The heap version keeps partial order and balances both insertion and removal efficiently.

## Benchmark Design

The benchmark program used `System.nanoTime()` to measure execution time. Each structure was tested with:

- 15,000 warmup operations
- 60,000 measured operations
- 7 trials
- The median time reported

I also used a fixed random seed so each structure received the same inputs. A checksum was used to make sure the operations were not optimized away by the compiler.

## Stack Results

The ArrayListStack performed faster than the linked list stack. Even though both have O(1) push and pop operations, the ArrayList version stores elements in contiguous memory. This improves cache performance and reduces memory overhead.

The linked list version creates separate node objects and uses pointers, which adds extra memory and processing cost. This shows that constant factors matter even when Big-O complexity is the same.

## Queue Results

The circular ArrayListQueue performed better than the linked list queue. Since the circular buffer keeps elements in a single array, it benefits from better memory locality.

The linked list queue also has O(1) operations, but it requires allocating nodes and updating pointers, which makes it slightly slower in practice.

## Priority Queue Results

The binary heap performed the best for larger workloads. The sorted implementations require  $O(n)$  time to insert elements because they must maintain sorted order.

The heap only requires  $O(\log n)$  time for insertion and removal. As the number of elements increases, this difference becomes more noticeable. The heap also benefits from being array-based, which improves cache performance.

To wrap this up, this project helped me understand that theoretical time complexity is important, but it does not tell the whole story. Even when two data structures have the same Big-O complexity, their actual runtime can be different due to memory layout, cache performance, and object allocation.

Array-based implementations generally performed better because they use contiguous memory and have less overhead. The binary heap was the most efficient PriorityQueue implementation for larger inputs.

Overall, the benchmark results matched what I expected from the theoretical analysis, and this project showed how important it is to test performance in real code.