ADT Implementations + Performance Shootout

Warmup ops: 15,000
Measured ops: 60,000
Trials: 7 (median reported)

All implementations passed sanity checks and produced identical checksums per workload, confirming correctness.

Stack Results

Implementation	W1 Bulk (ns/op)	W2 Mixed (ns/op)
ArrayListStack	14.97	25.79
DLinkedListStack	18.15	80.18

Analysis

The **ArrayListStack** performed better than the linked list version, especially during the mixed workload.

Both stacks perform push and pop in constant time. However, the difference comes from how data is stored in memory.

- `ArrayList` keeps elements next to each other in memory.
- `DLinkedList` stores each element in a separate node connected by pointers.

Because ArrayList uses contiguous memory, it benefits from better cache usage. The linked list must follow pointers scattered around memory and create a new node object for each push. This adds extra overhead, especially when many operations are performed in sequence.

This explains why the linked list stack slows down significantly in the mixed workload.

Conclusion: The array-backed stack is faster in practice due to better memory efficiency.

Queue Results

Implementation	W1 Bulk (ns/op)	W2 Mixed (ns/op)
ArrayListQueue	14.34	30.73
DLinkedListQueue	14.20	38.95

Analysis

In the bulk workload, both queues perform nearly the same. Both add elements at one end and remove them from the other.

Under the mixed workload, the ArrayListQueue is somewhat faster. The circular buffer keeps elements in contiguous memory, while the linked list must allocate nodes and follow pointers.

Even though both implementations do the same logical work, the array-based queue benefits from fewer object allocations and better cache behavior.

Conclusion: The circular ArrayList implementation is slightly more efficient overall.

Priority Queue Results

Implementation	W1 Bulk (ns/op)	W2 Mixed (ns/op)	W3 Skewed (ns/op)
SortedArrayListPQ	3965.72	4124.32	4467.06
SortedDLinkedListPQ	23542.88	21629.61	25662.27
BinaryHeapPQ	74.09	62.22	59.41

Analysis

The binary heap is dramatically faster than both sorted implementations.

The sorted priority queues keep all elements fully ordered. When inserting a new item, they must search for the correct position and either shift elements (ArrayList) or walk through many nodes (DLinkedList). As the structure grows, this insertion work becomes expensive.

The heap takes a different approach. It keeps only enough order to guarantee that the smallest element is always accessible. Because of this, both insertion and removal require only a small number of adjustments. This makes the heap much faster for larger workloads.

Between the two sorted versions, the ArrayList implementation performs much better than the linked list version. Shifting elements in contiguous memory is surprisingly fast, while pointer traversal and node allocation are costly.

Skewed Priorities

With skewed priorities, the heap becomes slightly faster. When many elements have similar priorities, the heap often requires fewer adjustments during operations. The sorted structures still need to shift or traverse elements to maintain full ordering.

Overall Conclusions

Stack Winner: ArrayListStack

Faster due to contiguous memory and fewer allocations.

Queue Winner: ArrayListQueue

Circular buffer preserves contiguous layout and avoids unnecessary overhead.

Priority Queue Winner: BinaryHeapPriorityQueue

Maintains partial ordering efficiently, avoiding the heavy insertion cost required by fully sorted structures.

Overall, the results show that practical performance depends heavily on memory layout and object allocation, not just the abstract description of the data structure.