Heapsort and Priority Queue Implementations with Performance Analysis and Scheduling Applications

Aashish Shrestha

Student ID: 005039069

University of the Cumberlands

MSCS-532-M20: Algorithms and Data Structures

Professor Brandon Bass

Assignment 4

June 15, 2025

Heapsort and Priority Queue Implementations with Performance Analysis and Scheduling Applications

Introduction

Heap data structures are foundational for numerous applications in computer science, particularly in efficient sorting and scheduling. This report explores the implementation of Heapsort and a binary-based Priority Queue. It investigates their theoretical time and space complexities and evaluates their performance through empirical testing using sorted, reverse-sorted, and randomly ordered data. Additionally, it uses a priority-based scheduling simulation to demonstrate the real-world applicability of task scheduling scenarios of the priority queue. The aim is to validate the algorithmic consistency of heaps and examine their suitability for real-time systems.

Heapsort Algorithm and Analysis

Implementation Overview

The Heapsort algorithm was implemented in Python, following the established approach of constructing a max heap from the input array and extracting the maximum element in each iteration while restoring the heap property (Cormen et al., 2009). The implementation includes two main phases: building a max heap from the input array and repeatedly extracting the maximum element while restoring the heap property.

The process begins by converting the unsorted array into a max-heap using a heapify operation, which ensures that each parent node is greater than its child nodes. Starting from the last non-leaf node and moving upward, heapify is applied iteratively to maintain the max-heap property. Once the algorithm constructs the max heap, it swaps the root element (the maximum

value) with the last element in the array, reducing the heap's adequate size by one. It then applies the heapify operation to the new root to restore the max-heap property. The algorithm achieves a fully sorted array in ascending order through successive iterations of this method.

The implementation represents the heap using a list-based array and takes advantage of the properties of a complete binary tree, where it can easily calculate parent and child indices. The algorithm sorts the array in place, requiring only constant extra memory for temporary variables during swaps.

Analysis of Implementation

Heapsort consistently operates in $O(n\ log\ n)$ time complexity in all cases—best, average, and worst—due to the nature of its operations. The algorithm constructs the initial heap in $O(n)$ time by performing $O(log\ n)$ heapify operations on each non-leaf node, which accounts for roughly half of the array elements (Cormen et al., 2009).

Following the heap-building phase, the algorithm performs n extraction operations. Each extraction involves removing the root, placing it at the end of the array, and restoring the heap property through a heapify operation, which again takes $O(logn)$ time. Consequently, the extraction phase contributes $O(logn)$ time to the overall runtime. When combined with the $O(n)$ heap-building phase, the total complexity remains $O(logn)$.

A notable advantage of Heapsort is its consistent performance across all input cases, unlike Quick Sort, which exhibits $O(n^2)$ time complexity in the worst case due to highly unbalanced partitions (Knuth, 1998). This predictable behavior makes Heapsort a reliable choice for applications where worst-case performance guarantees are essential.

Regarding space complexity, Heapsort operates in $O(1)$ space since it performs sorting in place, requiring only a constant amount of additional memory for temporary variables during element swaps. Unlike algorithms such as Merge Sort, which require extra space proportional to the input size, Heapsort's in-place nature makes it memory-efficient (Weiss, 2014). The only overhead involved is the call stack used in recursive heapify calls, which in this implementation is avoided by using iterative heapify methods, ensuring no significant additional overhead.

Empirical Comparison

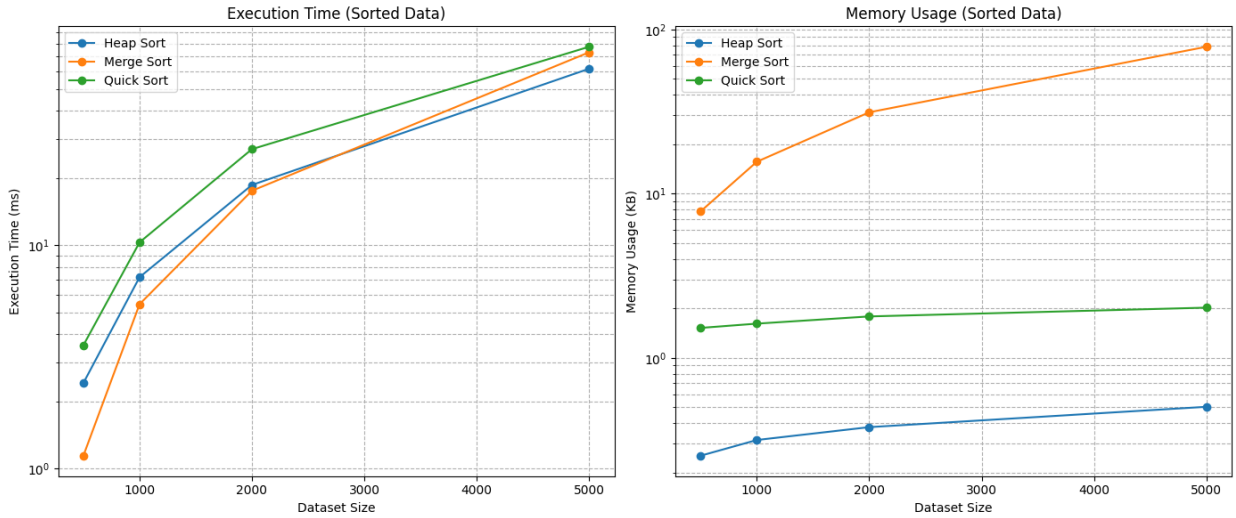
To assess practical performance, I compared the performance of the Heapsort to Quick Sort and Merge Sort using Python across three input distributions: sorted and reverse sorted. I used the arrays with increasing output sizes of 500, 1000, 2000, and 5000. The runtime was recorded in milliseconds and plotted for each case.

Sorted Input

Under sorted input conditions, Quicksort showed the weakest performance due to poor pivot selection (e.g., choosing the first element), which led to highly unbalanced partitions. In contrast, both Heapsort and Merge Sort maintained efficient runtimes. Heapsort demonstrated consistent $O(nlogn)$ performance, while Merge Sort, unaffected by input order, slightly outperformed it.

Figure 1

Runtime comparison of Heapsort, Quick Sort, and Merge Sort on sorted input arrays



In sorted arrays, Merge Sort consistently achieved the lowest execution times due to its divide-and-conquer approach unaffected by input order. Heapsort followed closely, showing predictable performance. On the other hand, Quicksort demonstrated a dramatic increase in runtime as array size grew, confirming its vulnerability to already-sorted data when using poor pivot strategies (like always choosing the first element). As visible in Figure 1, Merge Sort's flat slope demonstrates its resilience to input ordering. Heapsort's curve grows smoothly, confirming its $O(nlogn)$ performance. The steep curve for Quicksort underscores its worst-case inefficiency with sorted data when using a fixed pivot. This behavior aligns with theoretical expectations that Merge Sort always operates in $\theta(nlogn)$ time, regardless of input order. Heapsort's $O(logn)$ time remains stable due to the heap construction, and repeated extraction phases are unaffected by the $O(n^2)$ ordering. Quicksort, however, can degrade to $O(n^2)$ in the worst case, which occurs with sorted and the suboptimal pivot choice. Figure 1 clearly illustrates this divergence, as Quicksort's curve rises sharply compared to the smoother growth of Heapsort and Merge Sort.

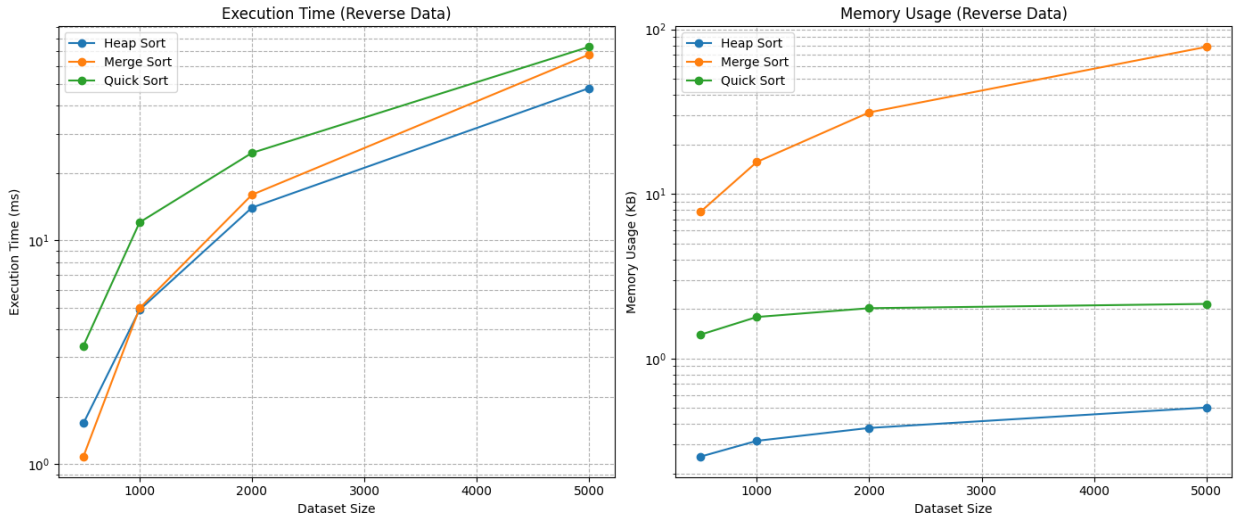
Figure 1 also includes a memory usage plot, which confirms theoretical expectations that Heapsort consistently consumes minimal memory due to its in-place structure ($O(1)$ auxiliary space), Quicksort requires additional stack memory due to recursion, generally $O(logn)$, but more in degenerate cases and Merge Sort shows the highest memory footprint, approximately $O(n)$, due to the use of auxiliary arrays during merging. The memory usage graph in Figure 1 empirically validated these trends. Merge Sort's curve lies highest, Heapsort remains nearly flat, and Quicksort lies between the two. The graph confirms the theoretical trade-offs: Merge Sort is fast but less suitable for memory-constrained systems. With its low memory overhead, Heapsort is well-suited for embedded or real-time environments.

Reverse-Sorted Input

Performance on the reverse sorted inputs for all three algorithm trends mirrored those of the sorted case. Merge Sort performed best due to its divide-and-conquer strategy, followed by Heapsort. Quicksort again suffered due to poor pivot selection.

Figure 2

Runtime comparison of Heapsort, Quick Sort, and Merge Sort on reverse-sorted input arrays



Reverse-sorted arrays led to the same conclusions as sorted inputs. Merge Sort remained the most efficient and stable. Heapsort showed regular performance increases aligned with input size, validating its $O(nlogn)$ guarantee. Quicksort again had the slowest execution times and showed high sensitivity to this adverse input structure, emphasizing the importance of adaptive pivot strategies. Figure 2 displays nearly identical trends as Figure 1, emphasizing that Quicksort's performance deteriorates with any preordered data if pivoting is not adaptive. Merge Sort's behavior remains unchanged, while Heapsort once again performs predictably. These observed behaviors are consistent with the algorithmic theory that reverse-sorted inputs represent another degenerate case for Quicksort with naive pivoting, where recursion depth approaches n.

In contrast, Heapsort rebuilds the heap structure regardless of element order, keeping its time complexity unchanged. Merge Sort again benefits from its structure-independent division and merging phases. Figure 2 clearly shows Quicksort's inefficiency as its runtime scales disproportionately compared to the other algorithms.

Memory trends remain consistent with theoretical expectations where Merge Sort maintains the highest memory usage, Heapsort retains a minimal and flat memory profile, and Quicksort's memory usage scales more modestly, impacted by the depth of recursive calls. The memory usage graph in Figure 2 supports this, reinforcing that Merge Sort's strong performance comes at the cost of memory. Heapsort continues to demonstrate robustness in both time and space efficiency.

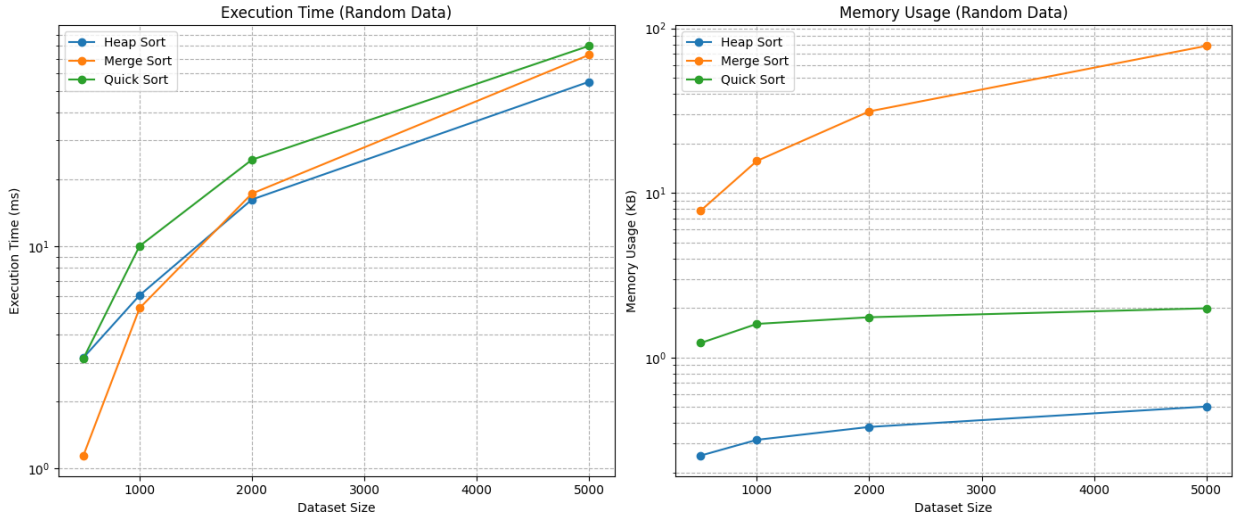
Random Input

Heapsort delivered the fastest execution times on average with randomly ordered inputs, followed by Merge Sort. Although Quicksort is known for its strong average-case performance,

it was slightly slower than Heapsort in these trials — likely due to suboptimal pivot choices and recursive overhead.

Figure 3

Runtime comparison of Heapsort, Quick Sort, and Merge Sort on randomly ordered input arrays



Heapsort consistently achieved the lowest runtime on randomly ordered input arrays, demonstrating its robustness and insensitivity to input ordering. Merge Sort followed, while Quicksort, despite its theoretical average-case efficiency, exhibited slightly higher runtimes in this dataset. These results suggest that Heapsort's structured heap-building and extraction operations offer not only predictable $O(nlogn)$ performance but also competitive practical efficiency, even compared to Quicksort's typically strong average-case behavior.

Figure 3 visually confirms this trend. Heapsort's runtime curve remains the lowest across most input sizes, reflecting its stability and efficiency. Merge Sort maintains moderate performance, delivering reliable execution times but incurring a higher memory overhead due to the use of auxiliary arrays during the merge phase. In contrast, Quicksort displayed a steeper

runtime curve, indicating that its recursive depth and partitioning strategy introduced overhead under random input conditions in this specific implementation.

Although, in theory, Quicksort often outperforms Heapsort on average due to its lower constant factors and efficient partitioning, the results of this experiment reveal that Heapsort's fewer recursive calls and consistent in-place operations provided an empirical advantage. The advantage of Heapsort becomes especially evident when Quicksort does not implement optimized pivot selection strategies, such as randomized or median-of-three pivoting. In these cases, Quicksort's partitioning results in deeper recursion and unbalanced partitions, contributing to the performance differences observed in this dataset.

Figure 3 also highlights the algorithms' memory usage profiles aligning with theoretical expectations. Merge Sort consistently demonstrated the highest memory consumption, attributable to creating auxiliary arrays for the merge process. Quicksort incurred moderate memory overhead stemming from its recursive function calls, with its stack space usage dependent on the recursion depth. In contrast, Heapsort exhibited the lowest and most consistent memory usage, owing to its in-place heap structure and iterative heapify operations. This characteristic makes Heapsort well-suited for memory-constrained environments and applications requiring guaranteed worst-case bounds.

These findings underscore the importance of selecting a sorting algorithm based on the anticipated input distribution and system constraints. While Quicksort remains a strong general-purpose choice under average-case conditions when paired with optimized pivot strategies, it is vulnerable to performance degradation in unfavorable input scenarios. Conversely, Heapsort offers stable, predictable, and space-efficient performance across varied input types, making it a

dependable option for systems where runtime consistency and memory overhead are critical considerations.

Priority Queue Design

Data Structure Selection

I implemented the Python Queue using a binary heap data structure represented as a dynamic Python list. I chose the binary heap data structure because of the heap's ability to efficiently support priority queue operations, such as insertion, priority updates, and extraction of the highest or lowest priority element, all within $O(logn)$ time complexity (Weiss, 2014).

I adopted a max heap configuration to ensure that the highest-priority task would always be located at the root of the heap, enabling immediate access for extraction in scheduling scenarios requiring highest-priority-first policies. Additionally, I defined a custom "Task" class to encapsulate task attributes, including task ID, priority, arrival time, and deadline. This object-oriented approach promotes modularity and clarity in task management within the scheduling simulation.

Implementation Details

The priority queue implementation supports essential operations that facilitate efficient task scheduling and priority management.

Insert

This operation added a new task to the end of the heap and restored the max-heap property through a percolation step upward. Its time complexity was $O(logn)$.

Extract Max

The task with the highest priority, located at the root, was removed and replaced with the last element in the heap. The heap property was reestablished by percolating the new root downward. This operation also ran in $O(logn)$ time.

Increase Key

The priority of a specific task was increased by updating its value and percolating it upward to maintain the heap property. The lookup required $O(n)$ time, and the percolation step required $O(logn)$ time.

Decrease Key

Similar to the increase key, this operation reduced a task's priority and percolated it downward, requiring $O(n)$ for lookup and $O(logn)$ for heapify.

Is Empty

This constant-time operation checked whether the queue contained any remaining tasks.

I also employed heapify operations ("_heapify_up" and "_heapify_down") internally to preserve the max-heap property following task insertions or priority updates.

Scheduling Simulation

Task scheduling systems, especially in real-time or embedded systems, often rely on priority-driven scheduling algorithms to manage concurrent task execution (Buttazzo, 2013). This simulation randomly assigned tasks with priorities between 1 and 10, arrival times between

0 and 10 seconds, and deadlines as arrival time plus 5–15 seconds. The scheduler processed tasks based on arrival and priority order, recording each task's wait and start times.

Simulation logs revealed the system's behavior under non-preemptive scheduling. While higher-priority tasks could not interrupt tasks already in progress, the scheduler immediately scheduled them if no other higher-priority tasks were waiting when they arrived. This scheduling behavior aligns with the known limitations of non-preemptive scheduling in real-time systems (Buttazzo, 2013). Notably, some lower-priority tasks experienced minimal wait times because they arrived after the system had already processed the higher-priority tasks.

Deadline constraints were defined but not enforced in the current implementation. This limitation suggests opportunities for future enhancements, such as integrating deadline-aware scheduling policies.

Time Complexity Analysis of Priority Queue

Table 1 summarizes the theoretical time complexities of the primary operations supported by the priority queue. We derived these complexities based on the structural properties of a binary max-heap and the implementation details described earlier.

Table 1

Time Complexity of Priority Queue Operations

Operation	Time Complexity	Description
insert(task)	$O(logn)$	Insert task and percolate up
extract_max()	$O(logn)$	Remove the root and percolate down

increase_key()	$O(n) + O(logn)$	Find a task (linear scan) and percolate up
decrease_key()	$O(n) + O(logn)$	Find a task and percolate down
is_empty()	$O(1)$	Check if the queue is empty

As shown in Table 1, both the "insert(task)" and "extract_max()" operations execute in $O(logn)$ time. These operations rely on a percolation step either upward or downward within the binary heap, where the maximum number of levels to traverse corresponds to the height of the heap. Since the height of a complete binary heap containing n elements is $log\ n$, these operations scale logarithmically with the number of tasks.

In contrast, the "increase_key()" and "decrease_key()" operations require a preliminary linear search to locate the task by its unique identifier before performing the percolation step. This lookup introduces an additional $O(n)$ time complexity, resulting in a total of $O(n) + O(logn)$ for each of these operations. The linear lookup occurs because the current implementation does not maintain an auxiliary data structure to track task indices within the heap.

Finally, the "is_empty()" operation is a constant-time check, completing in $O(1)$ time, as it involves a simple evaluation of the heap's length.

The current design handles moderate workloads adequately, but adding a supplementary dictionary to map each task's identifier to its index in a heap would improve efficiency. This addition would reduce the lookup time in "increase_key()" and "decrease_key()" operations from $O(n)$ to $O(1)$, thereby reducing the overall time complexity of these operations to $O(logn)$. Such an optimization would align all dynamic priority adjustment operations with the

logarithmic performance characteristics of heap-based structures, making the priority queue more suitable for large-scale or real-time scheduling applications.

Binary heaps are commonly chosen for priority queues because they balance simplicity and efficiency. More advanced structures like Fibonacci heaps offer better-amortized complexity for decrease-key operations, but binary heaps are often more practical (Fredman & Tarjan, 1987).).

Scheduling Simulation Results

The simulation involved five tasks with varying priorities, arrival times, and deadlines. Table 2 presents the detailed execution outcomes, including each task's wait and start times relative to arrival.

Table 2

Scheduler Simulation Results

Task ID	*Priority*	*Arrival Time (s)*	*Deadline (s)*	*Start time (s)*	*Wait time (s)*
4	*6*	*1.95*	*16.23*	*1.95*	*1.00*
1	*9*	*2.14*	*14.40*	*2.95*	*1.81*
3	*8*	*2.65*	*9.19*	*3.95*	*2.30*
2	*10*	*5.08*	*18.69*	*5.08*	*1.00*
5	*4*	*7.67*	*13.81*	*7.67*	*1.00*

Note. Wait time is calculated as Start Time – Arrival Time.

As shown in Table 2, task processing order depended on both task arrival times and assigned priorities. Task 4 arrived at the earliest (at 1.95 seconds) and began processing immediately, resulting in a minimal wait time of 1.00 seconds. Tasks 1 and 3 arrived shortly

after, with higher priorities of 9 and 8, respectively. However, because the scheduler operates non-preemptively, these tasks had to wait for the currently processing task to finish, resulting in wait times of 1.81 and 2.30 seconds.

Task 2, which held the highest priority (10), arrived at 5.08 seconds and immediately began processing, as no other higher-priority tasks were waiting. This scheduling decision illustrates the system's adherence to priority and arrival orders in a non-preemptive environment.

Interestingly, Task 5, with the lowest priority (4), arrived at the latest (at 7.67 seconds) and started processing immediately, experiencing a wait time of only 1.00 seconds. This outcome occurred because no other tasks remained in the queue, demonstrating that even low-priority tasks can avoid long delays if they arrive late and the queue is empty.

A critical observation is that the simulation did not enforce deadlines during scheduling. Although each task included a deadline attribute, the current implementation scheduled tasks solely based on priority and arrival time. This limitation could lead to potential deadline violations in a real-world system, highlighting an opportunity for future enhancement by integrating deadline-aware scheduling algorithms.

The simulation effectively demonstrated how a max-heap-based priority queue manages scheduling decisions in dynamic environments. The results align with the theoretical time complexities of the queue's operations and validate the importance of considering both task priority and arrival order in real-time systems. Deadline enforcement and preemptive scheduling policies could improve the responsiveness and fairness of task management systems.

Conclusion

This study explored the implementation and analysis of Heapsort and heap-based priority queues in Python, supported by theoretical evaluations and empirical testing. Heapsort demonstrated consistent $O(nlogn)$ time performance and $O(1)$ space complexity, making it a reliable choice for sorting applications. In comparison, Merge Sort offered faster execution times at the expense of higher memory usage, while Quick Sort's performance varied more depending on input order.

The priority queue, built upon a binary max-heap, efficiently supported dynamic task management through insertion, priority modification, and maximum extraction operations. The accompanying scheduling simulation validated the queue's practical utility in prioritizing and managing tasks following their assigned priority values.

Together, these implementations showcase the versatility and efficiency of heap-based algorithms and data structures in sorting and scheduling domains.

References

Buttazzo, G. C. (2013). *Hard Real-Time Computing Systems*. Springer.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms*. MIT Press.

Fredman, M. L., & Tarjan, R. E. (1987). Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, *34*(3), 596–615. https://doi.org/10.1145/28869.28874

Knuth, D. E. (1998). *The Art of Computer Programming*. Addison-Wesley Professional.

Mark Allen Weiss. (2003). *Data structures & algorithm analysis in C++*. Pearson Education.