Heap Data Structures: Implementation, Analysis, and Applications

Minh B. Le

University of the Cumberlands

MSCS-532-B01: Algorithms and Data Structures

Dr. Vanessa Cooper

July 19, 2025

Introduction

The goal of this paper is to explore the Heap data structure in an in-depth manner by implementing and analyzing the Heapsort algorithm, as well as the priority queue structure. All source code is stored in this GitHub repository: https://github.com/leminh646/Algorithm-Efficiency-and-Scalability.

Heapsort Implementation and Analysis

This part of the paper aims to implement and analyze the heapsort algorithm to better understand its concept. All files being used for this part will be stored in the heapSort folder of the GitHub repository.

Analysis

In analyzing Heapsort's performance, we consider its two principal phases: heap construction and repeated extraction of the maximum element. Beginning with an unsorted array of size n, the build-max-heap operation reorganizes the array in linear time. Although each call to max-heapify can take up to O(log n) time in the worst case, one can show by summing over all nodes that the total cost of building the heap is $\sum_{i=1}^{\lfloor n/2 \rfloor} O(h_i) = O(n)$, where h_i is the height of node i in the heap (Cormen et al., 2022). Once the max-heap is in place, the algorithm performs $n - 1$ extraction steps, each consisting of swapping the root with the last element of the heap (constant work) and then restoring the heap property via a call to max-heapify, which again takes $O(log\ n)$ time. Thus, the extraction phase contributes $O(n\ log\ n)$ time. Combining these yields a worst-case runtime of $O(n\ +\ n\ log\ n) = O(n\ log\ n)$.

Heapsort's average-case and best-case runtimes likewise remain $\Theta(n\ log\ n)$. Unlike some comparison-based sorts that exploit existing order in the input, Heapsort's sequence of heapify and swap operations is largely data-agnostic: regardless of whether the array begins

nearly sorted or in reverse order, the algorithm still performs the same sequence of heap adjustments. Consequently, there is no input configuration that permits it to break the $\Theta(n \log n)$ barrier, nor is there a scenario in which it degrades beyond that bound (Sedgewick & Wayne, 2011). This uniformity makes Heapsort's performance predictable but prevents any best-case speedup over the average or worst cases.

Regarding space complexity, Heapsort sorts in place, requiring only $O(1)$ additional storage beyond the input array. Recursive implementations of max-heapify incur $O(\log n)$ stack space in the worst case, but this can be eliminated by rewriting the procedure iteratively, preserving constant extra space. Aside from a few loop indices and temporary variables for swapping, no auxiliary arrays or data structures are needed. In practice, this low overhead and memory locality make Heapsort attractive in environments where extra space is at a premium (Cormen et al., 2022).

Comparison

A benchmark program ('benchmark.py') is created to compare the running time of Heapsort with Mergesort and Quicksort. Tests are run on 3 different data sizes and 3 different types of distributions. The results are shown in Figure 1 below.

```
PS C:\Min\Codes\Heap Data Structures> & C:/Python313/python.exe "c:/Min/Codes/Heap
Data Structures/heapSort/benckmark.py"
=== Distribution: random ===
n=  1000  Heapsort:0.003503s  Mergesort:0.002785s  Quicksort:0.001735s
n=  2000  Heapsort:0.007833s  Mergesort:0.006122s  Quicksort:0.003606s
n=  5000  Heapsort:0.022964s  Mergesort:0.017501s  Quicksort:0.010343s

=== Distribution: sorted ===
n=  1000  Heapsort:0.003750s  Mergesort:0.001928s  Quicksort:0.045459s
n=  2000  Heapsort:0.008600s  Mergesort:0.004062s  Quicksort:0.182942s
n=  5000  Heapsort:0.024094s  Mergesort:0.011222s  Quicksort:1.146335s

=== Distribution: reversed ===
n=  1000  Heapsort:0.003233s  Mergesort:0.002057s  Quicksort:0.044884s
n=  2000  Heapsort:0.007855s  Mergesort:0.004808s  Quicksort:0.180760s
n=  5000  Heapsort:0.022094s  Mergesort:0.011414s  Quicksort:1.129815s

PS C:\Min\Codes\Heap Data Structures>
```

Figure 1: Heapsort vs Mergesort vs Quicksort

On random inputs, Quicksort clearly leads in raw speed, completing 5,000 elements in roughly 0.01 s compared with 0.0175 s for Mergesort and 0.023 s for Heapsort. This ordering reflects the fact that although all three algorithms run in Θ(n log n) on average, Quicksort typically has the smallest constant factor: its in-place partitioning and good cache locality mean fewer total comparisons and swaps than the two-phase merge of Mergesort or the repeated "sift-down" operations of Heapsort.

When the array is already sorted, however, the naive first-element pivot choice sends Quicksort into its quadratic worst case. At n = 5,000, its time balloons to over 1 s, more than fifty times slower than its random-input performance. In contrast, both Mergesort and Heapsort remain squarely in the n log n regime, taking about 0.011 s and 0.024 s respectively for n = 5,000. This divergence illustrates the theoretical fact that Quicksort's worst-case time is $\Theta(n^2)$ when partitions are maximally unbalanced, whereas Mergesort and Heapsort guarantee $\Theta(n\ log\ n)$ regardless of input order.

The reversed-order results mirror the sorted-case behavior: Quicksort again degrades to over a second at n = 5,000, while Mergesort and Heapsort show nearly identical timings to their sorted-input runs. This stability in Mergesort and Heapsort aligns with their lack of dependency on input distribution—Mergesort always performs the same sequence of splits and merges, and Heapsort always executes the same heap-building and extract steps, yielding a uniform $\Theta(n\ log\ n)$ cost.

Finally, though Heapsort avoids Mergesort's extra $O(n)$ memory footprint and Quicksort's pathological cases, it pays for that in a higher constant factor: even on sorted or random data, it runs roughly twice as slowly as Mergesort under this implementation. This overhead comes from the repeated swapping and branching of the heap operations. Overall, the empirical timings match the theoretical analysis exactly: Quicksort shines on average but suffers quadratic collapse in worst cases, whereas Mergesort and Heapsort deliver predictable $n\ log\ n$ performance, trading off memory or constant‑factor cost.

Priority Queue Implementation and Applications

The concept of priority queues will be thoroughly explored in this part of the paper, with a proper implementation in Python and a complexity analysis. Source code can be found in the priorityQueue folder of the GitHub repository under the file 'pq.py'.

Design Choice

In designing the priority‑queue-based scheduler, an array‑backed binary heap was chosen for its simplicity and performance. An array (or dynamic list) representation allows constant‑time index arithmetic to navigate parent and child relationships. Specifically, for a node at index i, its children reside at indices $2i\ +\ 1$ and $2i\ +\ 2$, and its parent at $\lfloor(i–1)/2\rfloor$. This contiguous layout yields excellent cache locality and avoids the pointer overhead inherent in

tree-based structures (Cormen et al., 2022). Encapsulating each work unit in a small Task object carries all relevant metadata, such as ID, priority, arrival time, and optional deadline, in a single entity, while a supplementary index map grants $O(1)$ access to any task's position in the heap, which is crucial for efficient priority adjustments.

Analysis

The implementation hinges on two fundamental operations: sift_up and sift_down. Insertion appends the new Task at the end of the array and repeatedly compares it with its parent, swapping when its priority is higher; this "heapify-up" process terminates once the heap property is restored. Extraction of the highest-priority task swaps the root with the last element, removes it, and then "heapifies-down" the new root by swapping it with the larger of its two children until proper order is achieved. Priority updates leverage the index map to locate the task in $O(1)$ time, then selectively apply sift_up or sift_down depending on whether the priority increased or decreased. A simple length check completes the is_empty operation.

Each of these operations runs in $O(log\ n)$ time in the worst case because a node may traverse from a leaf to the root (or vice versa) over at most the height of the heap, which is $\lfloor log_2\ n \rfloor$ (Weiss, 2013). The auxiliary index map adds only $O(1)$ overhead per update, preserving these bounds. Checking emptiness is $O(1)$ since it merely compares the array's length to zero. Space complexity remains $O(n)$ for storing n tasks plus an additional $O(n)$ for the index map, which is acceptable in most scheduling contexts where task counts are bounded by system resources.

When applied to an operating-system-style scheduler or a real-time job manager, these priority-queue operations ensure that task arrival, dispatch, and priority changes all incur only logarithmic penalties, scaling gracefully even under high loads. This predictable performance

makes the array-based max-heap an attractive foundation for systems requiring timely, priority-driven execution without sacrificing efficiency or simplicity.

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to Algorithms (4th ed.). Random House Publishing Services. https://reader2.yuzu.com/books/9780262367509

Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley Professional.

Weiss, M. A. (2013). *Data Structures and Algorithm Analysis in Java* (3rd ed.). Pearson.