MSCS-532: Assignment 4: Heap Data Structures: Implementation, Analysis, and Applications

Oishani Ganguly

GitHub Repository:
https://github.com/Oishani/MSCS532_Assignment4

Part 1: Heapsort Implementation and Analysis

1. Implementation

The Heapsort implementation follows the classic in-place sorting algorithm, ensuring efficiency and robustness for various input cases. The code first transforms the input list into a max-heap structure, ensuring the largest element is always at the root. It then repeatedly swaps this root with the last unsorted element and reduces the considered heap size, maintaining the heap property through subsequent heapify operations. This approach guarantees a time complexity of $O(n \log n)$ and does not require additional space, aside from temporary variables. The implementation handles edge cases, including empty lists, single-element arrays, duplicate values, and already sorted or reverse-sorted data.

2. Analysis of Implementation

Time Complexity

Heapsort consists of two main phases: building the initial max-heap and repeatedly extracting the maximum element while maintaining the heap property.

1. Heap Construction

Building the heap requires converting an unordered array into a valid max-heap. This is done by calling `heapify` for all non-leaf nodes, starting from the last parent node and moving upward. For an array of size n, there are $\lfloor n/2 \rfloor$ non-leaf nodes.

The cost of `heapify` at a given node depends on its height in the heap. A node at height h (distance from the bottom) can move down up to h levels. The number of nodes at height h is at most $\lceil n/2^{h+1} \rceil$. So, the total work for building the heap is:

$$\sum_{h=0}^{\log n} \left(\frac{n}{2^{h+1}} \cdot O(h) \right)$$

This summation converges to $O(n)$, because most nodes are near the leaves and require little work (Cormen et al., 2009).

Even though a single call to `heapify` may take $O(\log n)$ time in the worst case, the majority of `heapify` calls are made on nodes low in the tree, which have small subtrees. Thus, the total cost is less than the naive $O(n \log n)$ estimate and is actually $O(n)$.

2. Sorting Phase (Extract-Max and Heapify)

After the heap is built, the sorting phase begins. The largest element (at the root) is swapped with the last element of the heap, reducing the effective size of the heap by one. The new root may violate the heap property, so `heapify` is called to restore the heap. This operation is repeated $n - 1$ times.

Each `heapify` call in this phase operates on a heap of size k, for $k = n, n-1, \ldots, 2$. Each such call takes $O(\log k) \leq O(\log n)$ time.

So, the total time for this phase is:

$$O(\log n) + O(\log(n-1)) + \cdots + O(\log 2) = O(n \log n)$$

3. Best, Average, and Worst Case

Unlike Quicksort, where performance depends on partition quality, Heapsort's time complexity is deterministic. Both phases depend on the structure of the heap, which is governed by the properties of a complete binary tree rather than the order of input data.

- Worst-case: $O(n \log n)$
- Average-case: $O(n \log n)$
- Best-case: $O(n \log n)$

There are no input arrangements that yield significantly better or worse performance, because the heap structure and extraction process require the same operations for all permutations of input data.

Heapsort does not exploit existing order in the input; every element must be compared and possibly moved multiple times to enforce the heap property. This necessity is what ensures Heapsort never exceeds $O(n \log n)$ or improves upon it for already sorted or nearly sorted data.

4. Recurrence Relation

For the sorting phase, each `heapify` operation can be modeled as:

$$T(n) = T(n-1) + O(\log n)$$

Unrolling this recurrence gives:

$$T(n) = O(\log n) + O(\log(n-1)) + \ldots + O(1) = O(n \log n)$$

Space Complexity

1. In-place Algorithm

Heapsort performs all operations within the input array, requiring only a constant amount of extra memory (for temporary variable swaps and index tracking). There is no need for auxiliary arrays or a call stack.

- Auxiliary space: $O(1)$
- Total space: $O(n)$, where n is for the array itself (input and output in place).

2. Overheads

The only overhead is a few integer variables used during swapping and iteration. There is no recursion (the algorithm is typically iterative) and no external data structures are used.

3. Comparison to Other Algorithms

- Quicksort (in-place, but recursive): space is $O(\log n)$ due to call stack.
- Mergesort (not in-place): space is $O(n)$ due to auxiliary arrays.
- Heapsort: minimal overhead, making it preferable when memory usage is critical.

Conclusion

Heapsort guarantees $O(n \log n)$ time complexity for all cases due to the rigid structure and process of heap operations, supported by a recurrence relation and sum analysis. Its space complexity is optimal for a comparison sort, with no more than constant auxiliary storage required.

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to algorithms (3rd ed.). *MIT Press*.

Knuth, D. E. (1998). The Art of Computer Programming, Volume 3: Sorting and Searching (2nd ed.). *Addison-Wesley*.

3. Comparison

Experimental Setup

To empirically compare the performance of Heapsort, Quicksort, and Merge Sort, we implemented each algorithm in its own Python file and benchmarked them side-by-side using a unified script. The benchmarking was performed on arrays of various sizes (1,000, 5,000, 10,000, and 50,000 elements) and across four input distributions:

- **Random arrays:** Elements drawn uniformly at random.
- **Sorted arrays:** Elements already in ascending order.
- **Reverse-sorted arrays:** Elements in descending order.
- **Arrays with repeated elements:** Arrays where a small set of values dominates.

To prevent premature recursion errors with recursive algorithms on large inputs, we explicitly set Python's recursion limit to 10,000 at the start of the script (`sys.setrecursionlimit(10000)`). Each test was run three times, and the average execution time was recorded. This approach minimizes noise due to system processes and interpreter fluctuations.

Results Table

The following table summarizes the average execution times (in seconds) for each configuration:

Algorithm	**Distribution**	**1000**	**5000**	**10000**	**50000**
Heapsort	Random	0.00223 s	0.01433 s	0.03134 s	0.18501 s
Heapsort	Sorted	0.00231 s	0.01506 s	0.03272 s	0.19238 s
Heapsort	Reverse	0.00201 s	0.01345 s	0.02937 s	0.17525 s
Heapsort	Repeated	0.00190 s	0.01291 s	0.02851 s	0.16724 s
Quicksort	Random	0.00120 s	0.00677 s	0.01447 s	0.08024 s
Quicksort	Sorted	0.00111 s	0.00637 s	0.01390 s	0.07766 s

Quicksort	Reverse	0.00110 s	0.00657 s	0.01362 s	0.07587 s
Quicksort	Repeated	0.00238 s	0.03675 s	0.13479 s	3.14621 s
Merge Sort	Random	0.00152 s	0.00886 s	0.01897 s	0.11012 s
Merge Sort	Sorted	0.00122 s	0.00727 s	0.01553 s	0.08944 s
Merge Sort	Reverse	0.00125 s	0.00753 s	0.01609 s	0.09152 s
Merge Sort	Repeated	0.00146 s	0.00883 s	0.01887 s	0.10920 s

Random Arrays

On randomly generated arrays, Quicksort was the fastest algorithm, especially as input size increased, completing the largest test (50,000 elements) in 0.08 seconds. Merge Sort followed closely, with times that were slightly higher but remained under 0.12 seconds for all sizes. Heapsort lagged behind, taking over twice as long as Quicksort for the largest input. These results reflect the theoretical efficiency of Quicksort's in-place partitioning and the generally small constant factors in its implementation. Heapsort's slower times are attributable to less cache-friendly memory access patterns and the overhead of maintaining the heap structure.

Sorted and Reverse-Sorted Arrays

Both Heapsort and Merge Sort demonstrated consistent performance regardless of input order. Their times were nearly identical to those observed for random input, confirming that neither algorithm is sensitive to initial order—consistent with their $O(n \log n)$ complexity for all cases. Quicksort, using randomized pivots, also maintained its speed and did not degrade on sorted or reverse-sorted arrays. This contrasts with deterministic pivot choices, which would result in $O(n^2)$ behavior; the randomized pivot ensures Quicksort remains robust across distributions.

Repeated Elements

The repeated elements case revealed a stark difference. While Heapsort and Merge Sort remained efficient, Quicksort's performance deteriorated rapidly with input size, taking over 3 seconds for 50,000 elements (compared to under 0.2 seconds for Heapsort and Merge Sort). This is due to the poor partitioning behavior of the classic Quicksort implementation on arrays with many duplicates, causing deep recursion and redundant work. In practice, optimized versions of Quicksort (e.g., with three-way partitioning) or switching to other sorts for highly redundant data are common solutions.

Theory vs. Empirical Results

The empirical results strongly confirm theoretical predictions. Heapsort and Merge Sort reliably achieved $O(n \log n)$ performance for all tested cases, showing little variation with input order or

value distribution. Quicksort was consistently fastest except on inputs with many repeated elements, where its performance deteriorated significantly—an example of the classic pitfalls of the standard algorithm on such distributions. The robust performance of Merge Sort and Heapsort on all distributions highlights their suitability for scenarios where worst-case guarantees are needed, while Quicksort is often preferred for typical cases due to its low overhead.

Additionally, the increase in the recursion limit ensured that Quicksort and Merge Sort could process larger arrays without failing due to Python's default stack depth, allowing a more accurate measurement of their true algorithmic performance.

Conclusion

Quicksort is the preferred algorithm for random and ordered data, achieving the fastest sorting times in the majority of cases. However, its poor performance on inputs with many repeated elements underscores the importance of algorithm selection based on input characteristics. Merge Sort and Heapsort offer consistently reliable performance, unaffected by input order or redundancy, making them strong choices where worst-case performance or stability is a priority. These empirical results validate the theoretical analyses of each algorithm’s strengths and limitations.

Part 2: Priority Queue Implementation and Applications

1. Implementation

Data Structure

For the priority queue’s underlying binary heap, we chose Python’s built-in dynamic list (`list`) as the storage container. The array-based representation of a binary heap is standard in computer science due to its space efficiency and fast, predictable access times. Each node’s children and parent can be computed directly via simple index arithmetic (e.g., left child at `2i+1`, right child at `2i+2`, parent at `(i-1)//2`), enabling all essential heap operations without pointer overhead or memory fragmentation. In Python, lists are particularly convenient due to dynamic resizing and optimized memory management, making them ideal for this context.

Task Class Design

Each task in the priority queue is encapsulated in a `Task` class, storing attributes essential for real-world scheduling, such as:

- task ID (for unique identification),
- priority (an integer or float, determining order in the heap),
- arrival time (when the task entered the system),
- deadline (if applicable), and
- any descriptive metadata.

The class implements comparison methods so that the heap can order tasks according to their priority, and provides clear string representations for debugging and logging. This modular approach allows flexibility in adapting to various scheduling policies.

Heap Type Selection

For this implementation, we selected a min-heap structure. This choice reflects typical scheduling scenarios—such as earliest deadline first or shortest job first—where tasks with the lowest numerical priority are serviced first. A min-heap efficiently provides quick access to the task with the smallest priority value. However, the underlying logic could be easily adapted to a max-heap if the scheduling algorithm requires servicing the highest priority task first.

Core Operations

a. insert(task)

The `insert` operation appends a new task to the end of the list, then restores the heap property by “bubbling up” the new element to its correct position. This process involves comparing the new task with its parent and swapping them if the heap property is violated.

Time Complexity: Insertion has a worst-case time complexity of $O(\log n)$, where n is the number of tasks. This is because the task may need to travel from a leaf to the root, which in a binary heap takes at most logarithmic steps.

b. extract_min() (or extract_max() for max-heap)

This operation removes and returns the root task (which is always the task with the highest or lowest priority, depending on heap type). The last element in the heap replaces the root, and the heap property is restored by “bubbling down” this element through successive swaps with its smallest child.

Time Complexity: Extracting the minimum (or maximum) task requires $O(\log n)$ time in the worst case, corresponding to the height of the heap.

c. increase_key(task, new_priority) / decrease_key(task, new_priority)

When a task’s priority changes (e.g., due to a system event or user action), the priority queue updates the value and repositions the task in the heap. If the priority increases (for a min-heap),

the task is bubbled down; if it decreases, it is bubbled up. In practice, efficiently locating the task may require auxiliary indexing (e.g., a hash map from task ID to index) to avoid a linear search.

Time Complexity: If the task's index is known, changing its priority and restoring heap order requires $O(\log n)$ time. If not, finding the task can take $O(n)$ time, so best practice is to maintain an index for efficient access.

d. is_empty()

This operation simply checks whether the heap/list is empty, returning a boolean. It is implemented with a constant-time check of the heap's length.

Time Complexity: Checking for emptiness is $O(1)$.

Design Justification and Edge Cases

- **Efficiency:** All primary operations—insert, extract, and key change—are logarithmic in the number of tasks due to the complete binary tree structure. Python's built-in `heapq` module and idiomatic tuple comparison are leveraged for efficiency and simplicity where appropriate.
- **Robustness:** The design checks for attempts to extract from or peek at an empty queue and handles duplicate priorities gracefully by using a tie-breaker (such as an insertion counter).
- **Scalability:** The array-based heap grows dynamically as tasks are added.

Scheduler Simulation Overview

To demonstrate the practical application of the priority queue, we implemented a simple task scheduler simulation. The scheduler receives tasks with varying priorities, arrival times, and (optionally) deadlines, and schedules their execution according to their priority. This setup reflects real-world systems—such as operating systems or job queues—where tasks must be dynamically managed and executed based on changing priorities and arrival patterns.

Simulation Design

- **Task Arrival:** At each simulated time step, one or more tasks may "arrive" and are inserted into the priority queue.
- **Scheduling Policy:** The scheduler always selects the available task with the highest priority (i.e., lowest priority value in a min-heap) to execute next.
- **Execution:** The selected task is removed from the queue and considered executed at the current time step.
- **Dynamic Updates:** The simulation can demonstrate updating task priorities mid-simulation, reflecting scenarios such as task aging, changes in urgency, or external events.

- **Edge Cases:** The simulation handles cases where the queue is empty (idle periods), and where multiple tasks have the same priority (using a counter to break ties).

Analysis

This simulation validates the priority queue's functionality and efficiency in a dynamic scheduling context. All core operations (`insert`, `extract_min`, `decrease_key`, and `is_empty`) are exercised as they would be in real systems. The $O(\log n)$ complexity for insertion, extraction, and priority updates ensures the scheduler remains responsive even as the number of tasks increases. The simulation also demonstrates robust handling of ties, dynamic priority changes, and system idling, showcasing the reliability and flexibility of the implementation for real-world scheduling tasks.