Algorithms Laboratory

Comparative Analysis of Priority Queue Implementations

Samuele Dell'Erba
Student ID: 7137648

8 August 2025

Contents

1 Introduction 3

2 Theoretical Description and Expected Performance 3

2.1 Max Heap . . . 3

2.2 Unsorted Linked List . . . 3

2.3 Sorted Linked List . . . 4

2.4 Summary Table of Expected Complexity . . . 4

3 Experiment Description 4

3.1 Test Environment . . . 4

3.2 Benchmark Methodology . . . 4

4 Implemented Code Documentation 5

5 Experimental Results and Critical Analysis 5

5.1 Insert Operation . . . 5

5.2 Maximum Operation . . . 6

5.3 Extract Max Operation . . . 7

5.4 Increment Key Operation . . . 7

6 Conclusions 8

1 Introduction

This report details an analysis and comparison of the performance of three different implementations of a max-priority queue, where the element with the highest priority corresponds to the largest value. As per the assignment requirements, the data structures examined are:

- **Max Heap**: An almost complete binary tree, stored in an array, satisfying the heap property.
- **Unsorted Linked List**: A simple, pointer-based list where new elements are added to the head.
- **Sorted Linked List**: A pointer-based list that is kept in decreasing order.

The fundamental priority queue operations (`insert`, `maximum`, `extract_max`, and `increment_key`) were implemented in Python for each structure. To this end, a series of experiments were run to measure the execution times of these operations on varying data sizes. This allows for an experimental verification of the theoretical computational complexities and helps in understanding the practical advantages and disadvantages of each approach.

2 Theoretical Description and Expected Performance

This section provides a brief overview of the theoretical characteristics of each data structure and establishes our expected performance in terms of asymptotic complexity (Big-O notation).

2.1 Max Heap

A Max Heap is a tree data structure that provides efficient access to its maximum element. The operations are implemented as follows:

- **Insert**: A new element is added to the end of the array. To maintain the heap property, the element "bubbles up" the tree (sift-up) until it reaches its correct position. This operation has a complexity of **O(log n)**.
- **Maximum**: The maximum element is always the root of the tree, making access a constant time operation, **O(1)**.
- **Extract Max**: The root is removed and replaced by the last element of the heap. This element then "bubbles down" the tree (sift-down or heapify) to restore the heap property. The complexity is **O(log n)**.
- **Increment Key**: A node's value is increased. Since this might violate the heap property, the node is bubbled up via a sift-up operation. The complexity is **O(log n)**.

2.2 Unsorted Linked List

This is the most straightforward implementation, as no order is maintained.

- **Insert**: A new element is simply added to the head of the list, which is a constant time operation, **O(1)**.
- **Maximum**: Finding the maximum element requires a full traversal of the list, resulting in linear complexity, **O(n)**.
- **Extract Max**: This operation also has a total complexity of **O(n)**, as it first requires finding the maximum element.
- **Increment Key**: Requires traversing the list to find the element at a given index, which has an average cost of O(n).

2.3 Sorted Linked List

This list is always maintained in decreasing order.

- **Insert**: To add a new element, the list must be traversed to find the correct position to preserve the order. This makes insertion an **O(n)** operation.
- **Maximum**: The maximum element is always the first in the list (the head), allowing for constant time access, **O(1)**.
- **Extract Max**: The head of the list is simply removed, which is also a constant time operation, **O(1)**.
- **Increment Key**: This involves finding the node (O(n)), removing it, and reinserting it in its new correct position (O(n)). The total complexity is **O(n)**.

2.4 Summary Table of Expected Complexity

Table 1 summarizes the expected computational complexities that we aim to verify with our experiments.

Table 1: Theoretical computational complexity of priority queue operations.

Operation	**Max Heap**	**Unsorted List**	**Sorted List**
`insert`	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$
`maximum`	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
`extract_max`	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
`increment_key`	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$

3 Experiment Description

3.1 Test Environment

The experiments were conducted on the following system:

- **Hardware**: 12th Gen Intel(R) Core(TM) i7-1255U CPU, 16.0 GB RAM
- **Operating System**: Windows 11 Home
- **Software**: Python 3.10.13, Matplotlib 3.10.5

3.2 Benchmark Methodology

The average execution time for each operation was measured as follows:

1. A set of input sizes N was defined: {10, 50, 100, 500, 1000, 2500, 5000}.
2. For each size N, the experiment was repeated $T = 10$ times (trials) to obtain a stable average and mitigate measurement noise.
3. In each trial, a data structure was first populated with N random integers sampled from an interval $[0, 10N]$.
4. The execution times for the individual operations were measured using Python's `time.perf_counter()`, which provides a high-precision timer.

5. To ensure that tests did not influence each other, the data structure was restored to its original size after any modifying operation.
6. The average results from the 10 trials were used to generate comparative graphs. All graphs are presented on a log-log scale. This choice is critical: without a logarithmic Y-axis, the fast-growing times of O(n) operations would "crush" the lines for O(1) and O(log n) operations against the horizontal axis, making them impossible to distinguish. The log-log plot therefore allows for a much clearer comparison of the growth rates.

4 Implemented Code Documentation

The project was implemented as a single Python script containing four classes:

- `Node`: An auxiliary class to represent the nodes of the linked lists.
- `MaxHeap`: Implements the priority queue using a heap stored in a Python list.
- `UnsortedLinkedList`: Implements the priority queue using an unsorted linked list.
- `SortedLinkedList`: Implements the priority queue using a linked list kept in decreasing order.

A key implementation choice for the `extract_max` operation of the unsorted list was to use an optimized single-pass approach to find and remove the maximum node. For the sorted list's `increment_key`, the node is simply removed and re-inserted, which ensures correctness while accepting an O(n) complexity.

5 Experimental Results and Critical Analysis

This section presents and analyzes the results from the experiments. For each operation, a comparative graph and a table with the raw numerical data are provided. All times are in seconds.

5.1 Insert Operation

The results for the insert operation are shown in Figure 1 and Table 2.

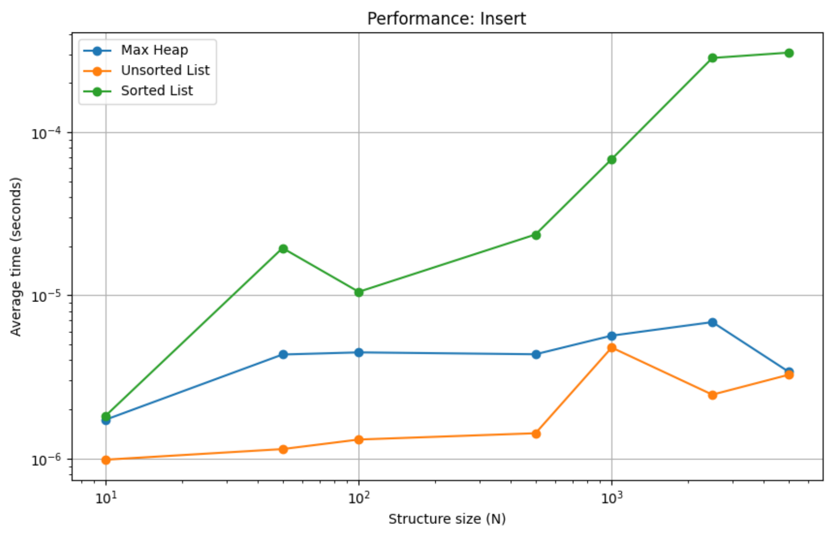

Figure 1: Comparison of average times for the `insert` operation (log-log scale).

Table 2: Average times (s) for the `insert` operation.

Size (N)	Max Heap	Unsorted List	Sorted List
10	1.73×10^{-6}	9.84×10^{-7}	1.83×10^{-6}
50	4.34×10^{-6}	1.14×10^{-6}	1.95×10^{-5}
100	4.47×10^{-6}	1.31×10^{-6}	1.05×10^{-5}
500	4.35×10^{-6}	1.43×10^{-6}	2.36×10^{-5}
1000	5.66×10^{-6}	4.79×10^{-6}	6.83×10^{-5}
2500	6.85×10^{-6}	2.46×10^{-6}	2.85×10^{-4}
5000	3.41×10^{-6}	3.26×10^{-6}	3.07×10^{-4}

Analysis: The experimental data strongly support the theoretical predictions. The **Unsorted List** (orange line) has a nearly flat execution time, consistent with its O(1) complexity, making it the fastest. The **Sorted List** (green line) shows clear linear growth, confirming its O(n) inefficiency. The **Max Heap** (blue line) sits in the middle, with the very slow growth characteristic of a logarithmic O(log n) trend.

5.2 Maximum Operation

The results for finding the maximum are shown in Figure 2 and Table 3.

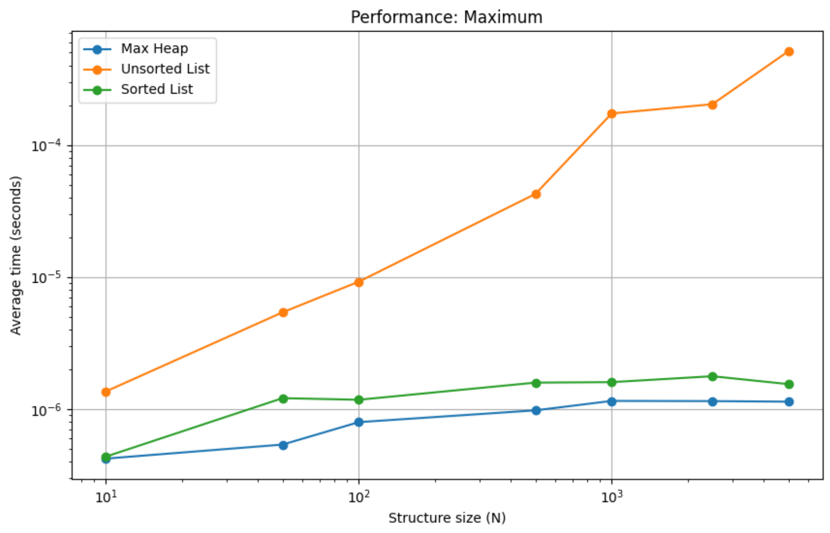

Figure 2: Comparison of average times for the `maximum` operation (log-log scale).

Table 3: Average times (s) for the `maximum` operation.

Size (N)	Max Heap	Unsorted List	Sorted List
10	4.22×10^{-7}	1.36×10^{-6}	4.36×10^{-7}
50	5.38×10^{-7}	5.40×10^{-6}	1.21×10^{-6}
100	7.95×10^{-7}	9.20×10^{-6}	1.18×10^{-6}
500	9.78×10^{-7}	4.25×10^{-5}	1.58×10^{-6}
1000	1.15×10^{-6}	1.73×10^{-4}	1.60×10^{-6}
2500	1.15×10^{-6}	2.03×10^{-4}	1.77×10^{-6}
5000	1.14×10^{-6}	5.11×10^{-4}	1.54×10^{-6}

Analysis: As expected, the data show that both the **Max Heap** and **Sorted List** (blue and green lines) are extremely fast, with low, constant execution times that confirm their O(1) complexity. In contrast, the **Unsorted List** (orange line) is clearly the slowest, showing linear growth in line with its O(n) complexity.

5.3 Extract Max Operation

The results for extracting the maximum are shown in Figure 3 and Table 4.

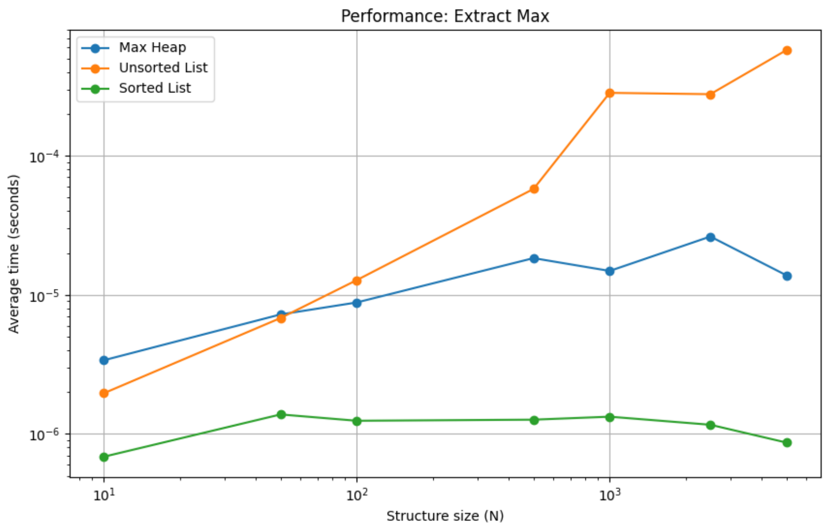

Figure 3: Comparison of average times for the `extract_max` operation (log-log scale).

Table 4: Average times (s) for the `extract_max` operation.

Size (N)	Max Heap	Unsorted List	Sorted List
10	3.39×10^{-6}	1.96×10^{-6}	6.86×10^{-7}
50	7.23×10^{-6}	6.81×10^{-6}	1.38×10^{-6}
100	8.81×10^{-6}	1.28×10^{-5}	1.24×10^{-6}
500	1.84×10^{-5}	5.77×10^{-5}	1.27×10^{-6}
1000	1.48×10^{-5}	2.83×10^{-4}	1.33×10^{-6}
2500	2.62×10^{-5}	2.77×10^{-4}	1.16×10^{-6}
5000	1.38×10^{-5}	5.74×10^{-4}	8.67×10^{-7}

Analysis: The **Sorted List** (green line) is the most efficient structure for this task, with a constant and very low execution time, as expected from its O(1) complexity. The **Unsorted List** (orange line) is again the slowest. The **Max Heap** (blue line) offers excellent performance with very limited growth (O(log n)), making it a great alternative.

5.4 Increment Key Operation

The results for the increment key operation are shown in Figure 4 and Table 5.

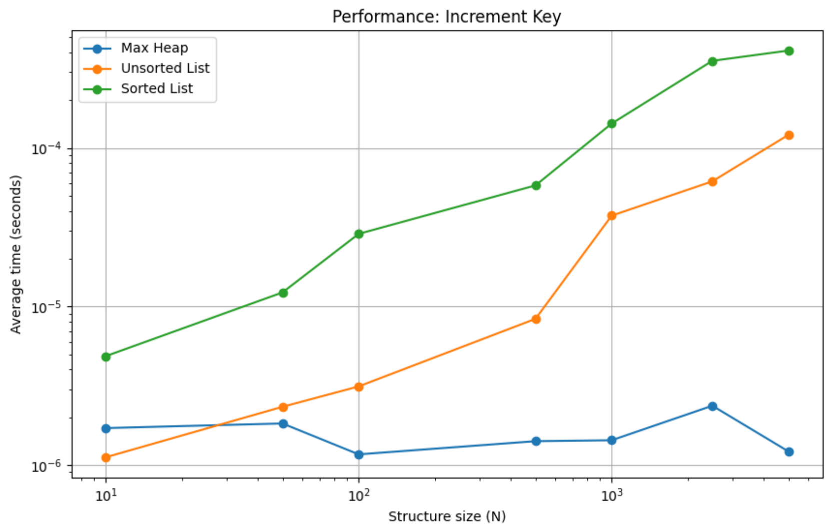

Figure 4: Comparison of average times for the `increment_key` operation (log-log scale).

Table 5: Average times (s) for the `increment_key` operation.

Size (N)	Max Heap	Unsorted List	Sorted List
10	1.71×10^{-6}	1.12×10^{-6}	4.86×10^{-6}
50	1.83×10^{-6}	2.33×10^{-6}	1.23×10^{-5}
100	1.17×10^{-6}	3.13×10^{-6}	2.87×10^{-5}
500	1.41×10^{-6}	8.36×10^{-6}	5.81×10^{-5}
1000	1.43×10^{-6}	3.74×10^{-5}	1.43×10^{-4}
2500	2.37×10^{-6}	6.16×10^{-5}	3.54×10^{-4}
5000	1.22×10^{-6}	1.21×10^{-4}	4.12×10^{-4}

Analysis: For this operation, the **Max Heap** (blue line) is clearly superior. Its execution time remains almost constant, reflecting its efficient O(log n) complexity. In contrast, both the **Sorted List** and **Unsorted List** (green and orange lines) show a linear O(n) trend, since they both require a scan to find the element to be modified.

6 Conclusions

The experimental analysis has clearly confirmed the theoretical predictions about the complexity of these priority queue implementations. The results show that there is no single "best" data structure; the optimal choice depends on the expected usage profile and the relative frequency of operations.

- The **Max Heap** proved to be the most versatile and balanced implementation. It offers excellent (logarithmic or constant) performance across all key operations, making it the ideal default choice for a general-purpose priority queue.
- The **Unsorted Linked List** excels only at insertion (O(1)) and pays a high price for all other operations that need to find the maximum element (O(n)). Its use is therefore limited to scenarios that are heavily dominated by insertions.

- The **Sorted Linked List** offers unbeatable performance for accessing and extracting the maximum (O(1)), but this comes at the cost of a very slow insert operation (O(n)). It is well-suited for applications where the queue is frequently queried for its maximum element but rarely modified.

In summary, this analysis reinforced the understanding of the intrinsic trade-offs in data structure design, demonstrating that both theoretical and experimental analysis are fundamental to making informed implementation choices.