UNIVERSITÀ DEGLI STUDI FIRENZE

DEPARTMENT OF INFORMATION ENGINEERING

AUTHOR: AWAD ELIA ***STUDENT ID:*** 7140815

Heap vs Unsorted Linked List vs Sorted Linked List

Contents

1 General Introduction

1.1 Laboratory Description

The goal of this project is to compare three different implementations of priority queues:

- Heap (binary max heap)
- Sorted linked list (in descending order)
- Unsorted linked list

In the following sections, I will first present the theoretical background of the problem being addressed, followed by the code documentation to facilitate understanding, and finally, based on the implemented code, I will conduct experiments to verify whether the theoretical assumptions hold in practice. At the same time, I will analyze the advantages and disadvantages of each implementation.

1.2 Testing Platform

It is important to specify the environment under which the tests were conducted, as performance results may vary depending on the platform used:

- **CPU:** Apple M2 SoC, 8 cores
- **RAM:** 8GB unified memory
- **Storage:** 256GB PCIe 4.0 SSD
- **Operating System:** macOS Sequoia 15.6
- **Programming Language:** Python 3.13.6

2 Theoretical Description of the Problem

2.1 Introduction to Priority Queues

A priority queue is a data structure used to maintain a set S of elements, each associated with a value called a *key*. Priority queues can be implemented in two main variants:

- **Max-Priority Queue:** where the element with the highest priority has the largest *key*.
- **Min-Priority Queue:** where the element with the highest priority has the smallest *key*.

Unlike a FIFO queue, where elements are removed in the order they arrived, in a priority queue, the element removed is always the one with the highest (or lowest) priority depending on the chosen implementation. Priority queues have several applications, such as sorting algorithms (Heap Sort) or process scheduling in operating systems.

In this project, we focus on the first variant — the **Max-Priority Queue**. The main **operations** it provides are:

- `Insert(S, x)` – inserts an element x into the set S.
- `Maximum(S)` – returns the element in S with the largest *key*.
- `Extract-Max(S)` – removes and returns the element in S with the largest *key*.
- `Increase-Key(S, x, k)` – increases the *key* of element x to a new value k, provided that $k > x.key$.

2.2 Heap

A possible implementation of a priority queue is through a **Binary Heap**. A heap is a nearly complete binary tree, meaning that all leaves have a distance from the root of either h or $h - 1$, where h is the height of the tree (the number of edges on the longest path from the root to a leaf). Because of this property, its height is $O(\log n)$.

A heap can be stored inside an array A as follows:

- Root of the tree: $A[0]$
- Parent of $A[i]$: $A[(i-1)//2]$
- Left child of $A[i]$: $A[2i+1]$
- Right child of $A[i]$: $A[2i+2]$

A heap must satisfy a fundamental **property**, depending on the implementation:

- $A[Parent(i)] \geq A[i]$ for Max-Heaps
- $A[Parent(i)] \leq A[i]$ for Min-Heaps

The fundamental operations supported by a Max-Heap and their theoretical **complexities** are:

- `Insert(S, x)`: inserts a new element at the bottom of the tree, then "bubbles up" the element until the heap property is restored — $O(\log n)$.
- `Maximum(S)`: the element with the largest key is at the root — $\Theta(1)$.
- `Extract-Max(S)`: retrieves and removes the root element (the largest key), then restores the heap property — $O(\log n)$.

2.3 Unsorted Linked List

A linked list is a data structure consisting of a sequence of nodes, where each node contains two fields:

- `value`: stores the node's value
- `next`: stores a pointer to the next node in the list

The first node of the list is called the *head*, and the last node is called the *tail*, whose `next` pointer is null, indicating the end of the list.

The theoretical time complexities for the main operations are:

- `Insert(List, value)`: can be performed in $\Theta(1)$ if the tail is tracked.
- `Maximum(List)`: requires scanning the entire list in the worst case — $O(n)$.
- `Extract-Max(List)`: also $O(n)$, since it must traverse the list to find and remove the node with the largest value.

2.4 Sorted Linked List

This implementation is similar to a regular linked list, except that values are kept sorted (in ascending or descending order). Assuming a descending order, the theoretical time complexities are:

- `Insert(List, value)`: must find the correct position for insertion, which may require scanning the entire list — $O(n)$.
- `Maximum(List)`: simply returns the value of the head node — $\Theta(1)$.
- `Extract-Max(List)`: removes and returns the head node — $\Theta(1)$.

2.5 Summary of Theoretical Results

Summary:

Data Structure	**Insert**	**Extract-Max**	**Peek-Max**
Binary Heap	$O(\log n)$	$O(\log n)$	$\Theta(1)$
Unsorted Linked List	$\Theta(1)$	$O(n)$	$O(n)$
Sorted Linked List	$O(n)$	$\Theta(1)$	$\Theta(1)$

3 Code Documentation

3.1 File Overview and Module Interactions

The implementation consists of four files:

- `heap_priority_queue.py`: contains the implementation of the Heap data structure.
- `unsorted_list_priority_queue.py`: implements the unsorted linked list priority queue.
- `sorted_list_priority_queue.py`: implements the sorted linked list priority queue.
- `main.py`: runs the actual experiments, including data generation and plotting of results for each data structure.

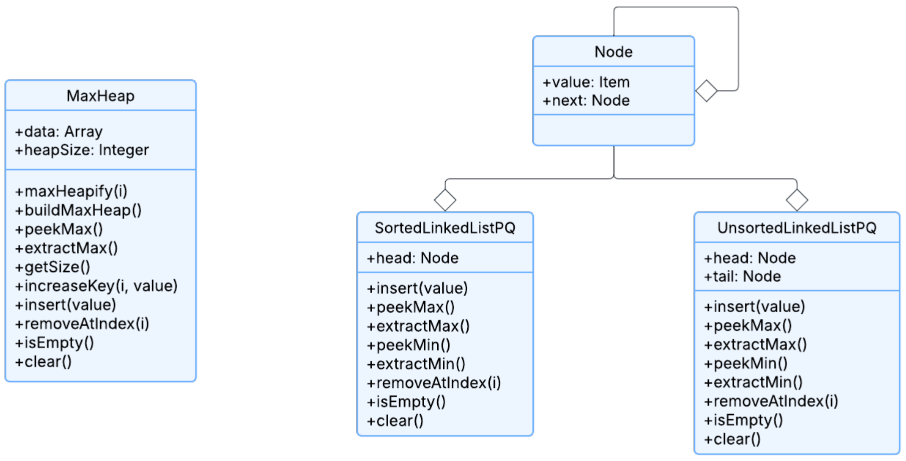


Figure 1: UML Class Diagram

The **MaxHeap** class implements a binary heap. The **Node** class is used via composition by the **UnsortedLinkedListPQ** and **SortedLinkedListPQ** classes, which implement the unsorted and sorted linked list priority queues respectively.

3.2 Implementation Choices

For simplicity, in all data structures the inserted items directly represent their priority values. Additionally, the project implements the *Max-Priority* variant of the priority queue. Several methods are included for completeness even if they are not used in the experiments.

3.3 Implemented Methods

A brief description of the main methods follows, to clarify the implementation:

- **MaxHeap**
  - `maxHeapify(i)`: restores the heap property by comparing the node at index i with its children, swapping if necessary, and recursing down the tree.
  - `buildMaxHeap()`: builds a max-heap from an arbitrary array using bottom-up `maxHeapify()`.
  - `peekMax()`: returns the item with the highest priority (root of the heap).
  - `extractMax()`: removes and returns the root item while maintaining the heap property.
  - `getSize()`: returns the heap size.
  - `increaseKey(i, value)`: increases the key of node i to `value` (if greater than the current one).
  - `insert(value)`: inserts a new item maintaining the heap property.
  - `removeAtIndex(i)`: removes the item at index i.
  - `isEmpty()`: returns whether the heap is empty.
  - `clear()`: clears the heap.
- **UnsortedLinkedListPQ**
  - `insert(value)`: appends a new item at the end of the list.
  - `peekMax()`: returns the maximum value by scanning the list.
  - `extractMax()`: removes and returns the maximum value by scanning the list.
  - `peekMin()`: returns the minimum value by scanning the list.
  - `extractMin()`: removes and returns the minimum value by scanning the list.
  - `removeAtIndex(i)`: removes the item at index i.

  - `isEmpty()`: returns whether the list is empty.
  - `clear()`: clears the list.

- **SortedLinkedListPQ**
  - `insert(value)`: inserts a new item in the correct position to keep the list sorted.
  - `peekMax()`: returns the maximum item (head node).
  - `extractMax()`: removes and returns the head node.
  - `peekMin()`: returns the last node (minimum value).
  - `extractMin()`: removes and returns the last node.
  - `removeAtIndex(i)`: removes the item at index *i*.
  - `isEmpty()`: returns whether the list is empty.
  - `clear()`: clears the list.

- **tests.py**
  - `runTests(sizes, reps, results, structure, structureName)`: runs the experiments for each operation on a given structure.
  - `plotComparison(results, sizes, opName)`: plots and saves graphs comparing the three structures on a given operation.
  - `plotStructureGraphs(results, sizes, structureName)`: plots and saves graphs for each operation of a given structure.
  - `printTable(results, tableSizes)`: prints a summary table with the average execution times for various sizes.

4 Experiments Description

4.1 Data Used

Experiments were conducted on datasets up to 20,000 elements. For each test, random values were inserted to obtain an average over independent experiments.

4.2 Measurements

Measurements focus on the execution time of three main operations shared by all data structures:

- `insert(value)` – insert an element
- `peekMax()` – retrieve the maximum
- `extractMax()` – extract the maximum

For each operation, the average execution time was computed over multiple repetitions for increasing dataset sizes (up to 20,000 elements). The function `time.perf_counter()` was used to measure start and end times. The average times were stored in the `results` variable.

4.3 Experimental Results and Analysis

4.3.1 Insertion

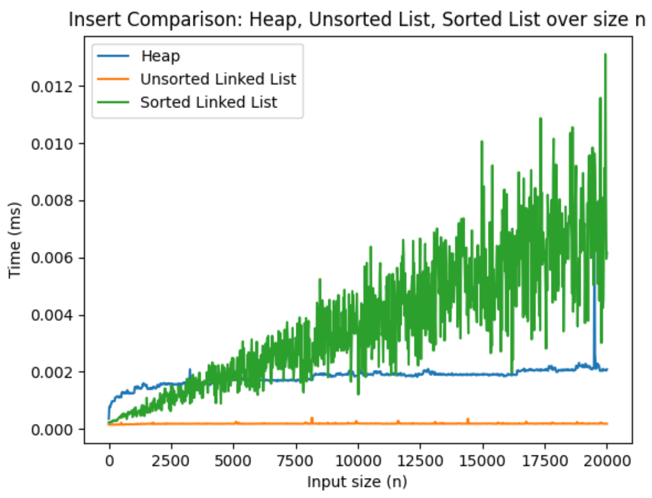


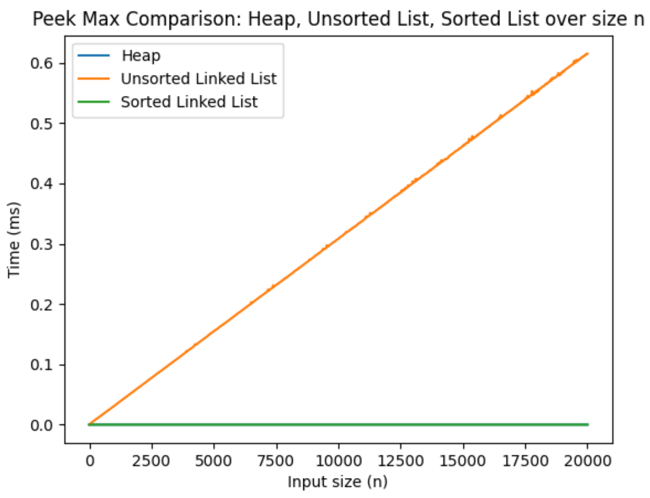
Figure 2: Insert comparison across data structures

Observing the graphs, we can see that the experimental results align with the previously stated theoretical hypotheses: Indeed, it can be observed that the **sorted linked list** grows linearly $O(n)$ as the number of elements increases, precisely because, in the worst-case scenario, it must traverse the entire list before inserting the element into the correct position. The reason for such high variance is that a random value is inserted each time; therefore, it could be inserted at the beginning or at the end of the list. In fact, it is noticeable that the variations are much more contained at the start because the number of elements to traverse in the worst case is relatively low, whereas for large sizes, the variations explode precisely because it might sometimes need to traverse 1,000 elements, or all 20,000.

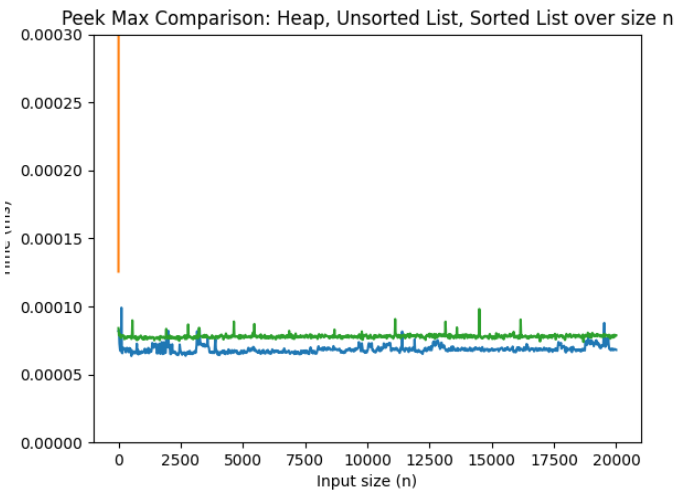
Regarding the **heap**, it is also consistent with the theory, exhibiting logarithmic growth $(O(logn))$. Unlike the sorted linked list, here we find much less variation because the new element is inserted at the bottom of the tree each time, then moves up the tree until it finds the correct position; therefore, the number of steps is proportional to the tree's height. Since the tree is almost complete, this height is $O(logn)$, thus resulting in smaller variations.

Finally, the **unsorted linked list**, as expected, shows a constant trend $(\Theta(1))$, presenting almost zero variation as it always performs the same pointer swap operation regardless of the data size or distribution.

4.3.2 Maximum Search



(a) Maximum Search Comparison



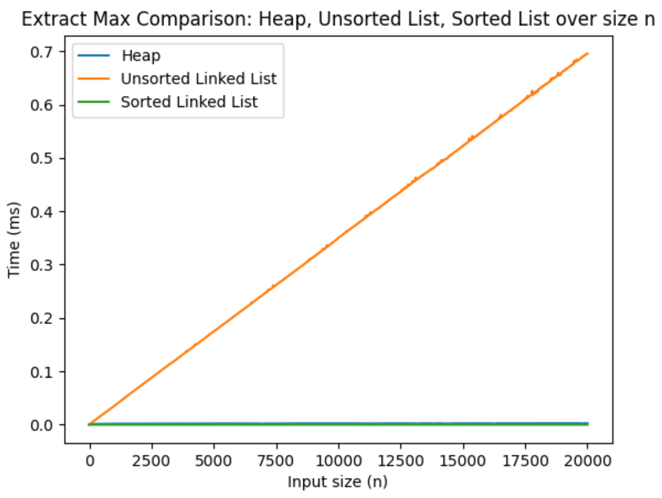
(b) Zoomed Maximum Search Comparison

Figure 3: Maximum search comparison across data structures

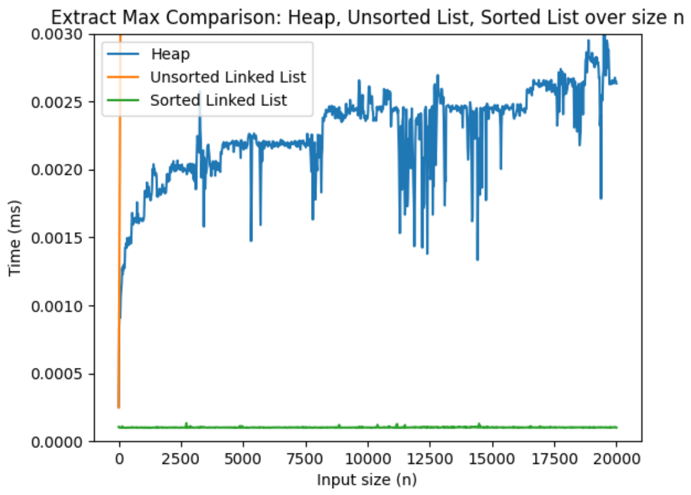
Regarding the **unsorted linked list**, execution times increase linearly as the number of elements grows ($O(n)$); this occurs because, in the worst-case scenario, the list must be traversed to the very end.

While it is nearly impossible to distinguish the difference between finding the maximum in the heap versus the sorted linked list in Figure 3(a), Figure 3(b)—which depicts a zoomed-in view of the same comparison—shows that both the **heap** and the **sorted linked list** exhibit a constant trend (consistent with our hypotheses). However, the sorted linked list has slightly higher execution times due to the requirement of a double memory access: first accessing the pointer to the node instance, and then accessing the node's value field.

4.3.3 Maximum Extraction



(a) Exract Max Comparison



(b) Zoomed Extract Max Comparison

Figure 4: Maximum extraction comparison across data structures

The **unsorted linked list** exhibits a linear trend ($O(n)$); indeed, it must traverse the entire list to find the maximum.

In Figure 4(b), we find the comparison on a smaller scale, where it is clearly noticeable that the **heap** follows a logarithmic trend ($O(logn)$), as hypothesized.Still looking at Figure 4(b), we see how for the **sorted linked list**, extracting the maximum is extremely rapid and constant time ($\Theta(1)$), as it must access the head node of the list regardless of its size.

4.4 Final Conclusions

Comparing both graphical and tabular results, it is evident that each priority queue implementation has its own strengths and weaknesses:

- **Heap:** the most balanced solution, offering logarithmic complexity for both insertion and extraction. Efficient for large datasets with frequent mixed operations.
- **Unsorted Linked List:** optimal for insertions but inefficient for all other operations.
- **Sorted Linked List:** opposite behavior — excellent for retrieval and extraction but poor for insertions. Useful in contexts where frequent maximum retrieval is needed.

Table 1: Average Operation Times

Size:	**Data Structure**	**Operation**	**Average Time (ms)**
101	Heap	Insert Peek Max Extract Max	0.000926 0.000089 0.001128
101	Unsorted Linked List	Insert Peek Max Extract Max	0.000145 0.003640 0.004103
101	Sorted Linked List	Insert Peek Max Extract Max	0.000241 0.000077 0.000101
1001	Heap	Insert Peek Max Extract Max	0.001264 0.000068 0.001674
1001	Unsorted Linked List	Insert Peek Max Extract Max	0.000154 0.030938 0.035021
1001	Sorted Linked List	Insert Peek Max Extract Max	0.000450 0.000076 0.000101
5001	Heap	Insert Peek Max Extract Max	0.001741 0.000067 0.002248
5001	Unsorted Linked List	Insert Peek Max Extract Max	0.000176 0.154720 0.174762
5001	Sorted Linked List	Insert Peek Max Extract Max	0.001518 0.000077 0.000101
10001	Heap	Insert Peek Max Extract Max	0.001906 0.000067 0.002447
10001	Unsorted Linked List	Insert Peek Max Extract Max	0.000183 0.307884 0.350318
10001	Sorted Linked List	Insert Peek Max Extract Max	0.005373 0.000078 0.000103
20001	Heap	Insert Peek Max Extract Max	0.002051 0.000069 0.002633
20001	Unsorted Linked List	Insert Peek Max Extract Max	0.000179 0.615134 0.695486
20001	Sorted Linked List	Insert Peek Max Extract Max	0.004529 0.000077 0.000100

In conclusion, experimental results fully confirm the theoretical predictions, demonstrating how different data structures excel under different operational constraints.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein (2009). *Introduction to Algorithms*, 3rd Edition, McGraw Hill.