**Assignment 4: Heap Data Structures: Implementation, Analysis, and  
Applications**

2024

* **name: Abdul raheman Gotori**
* **Student id: 005029919**
* **COURSE & TITLE: Algorithms and Data Structures (MSCS-532-A01)**
* **DATE:** **15TH september 2024**

Table of Contents

[**Heapsort Algorithm** 2](#_Toc177308646)

[**Priority Queue Implementation** 8](#_Toc177308647)

# **Heapsort Algorithm**

**ANALYSIS**Time Complexity Analysis  
  
1.Best-Case Time Complexity  
Heapsort takes O(n log n) time even in the best case. No matter how sorted the array is, heapification and extraction require the same operations. Unlike insertion sort, heapsort does not benefit from pre-sorted input. The array is already sorted, but the extraction process, including heapification after each extraction, takes O(n) time, and building the max-heap takes O(n log n).   
The best case time complexity = O(n log n).  
  
  
2. Average-Case Time Complexity  
  
The average heapsort time is O(n log n). The algorithm proceeds the same way regardless of input array arrangement. Whether the array is random or partially sorted max-heap construction takes O(n) time and extraction and heapification take O(log n). Since these steps are deterministic and independent of element arrangement, the average time complexity remains O(n log n).  
  
  
  
3. Worst-Case Time Complexity  
  
The worst-case Heapsort time complexity is O(n log n). Divided into two main phases:  
The max\_heapify function must be called on all non-leaf nodes to create a max-heap from an unsorted array. An array of size n has about n/2 non-leaf nodes. Time to heapify a subtree is proportional to node height. With log n nodes, the heap's height is logarithmic. Construction of the heap takes O(n).  
  
After the max-heap is finished, the algorithm extracts the root, the maximum element, and places it at the end of the array. This operation requires calling max\_heapify on the reduced heap to maintain the heap property. This phase takes O(n log n) time because there are n elements and heapification takes O(log n) time for each extraction.  
  
Why Heapsort is O(n log n) in All Cases  
In all cases, Heapsort consistently executes in O(n log n) time due to the nature of the heap operations:  
  
The max-heap is constructed during this phase by calling max\_heapify on each non-leaf node. The total time required to heapify a node is linear, defined as O(n), due to the fact that the time required to heapify a node is dependent on its height and there are more nodes at lower heights.  
  
Heapify operations during extraction: The root element (maximum) is exchanged with the last element in the heap in each extraction step, and the function max\_heapify is used to process the new root. For each heapify call, the time complexity is O(log n), as the heap has a logarithmic height (log n). Given that this is performed for each element (n times), the total time required for this phase is O(n log n).

Space Complexity and Overheads  
Heapsort is an in-place sorting algorithm which means that it necessitates only a constant amount of extra space in the form of O(1). In contrast to Merge Sort which necessitates additional memory for temporary arrays, Heapsort executes all operations directly on the input array without necessitating another array of storage. Heapsort's space complexity is O(1) due to minimal requirements for space which are restricted to a few auxiliary variables for index swapping and tracking.  
  
With Heapsort there are no substantial memory overheads. If max\_heapify is implemented recursively the recursion stack would contain the sole additional cost. However, in many implementations max\_heapify is written recursively to avoid this overhead thereby guaranteeing that the space complexity remains O(1).  
  
**COMPARISON**

Empirical Comparison CodeMeasuring the execution time of each sorting algorithm on arrays of different input sizes and distributions:- *import random*

*import time*

*# Function to generate different types of input*

*def generate\_input(size, case='random'):*

*if case == 'sorted':*

*return list(range(size))*

*elif case == 'reverse-sorted':*

*return list(range(size, 0, -1))*

*else:*

*return [random.randint(0, size) for \_ in range(size)]*

*# Function to measure execution time*

*def measure\_time(sort\_func, A):*

*start = time.time()*

*sort\_func(A)*

*return time.time() - start*

*# Testing each algorithm on different input cases*

*sizes = [1000, 5000, 10000, 50000]*

*cases = ['random', 'sorted', 'reverse-sorted']*

*for size in sizes:*

*for case in cases:*

*A\_quick = generate\_input(size, case)*

*A\_merge = A\_quick[:]*

*A\_heap = A\_quick[:]*

*#Measure times*

*time\_quick = measure\_time(lambda A: quicksort(A, 0, len(A) - 1), A\_quick)*

*time\_merge = measure\_time(lambda A: merge\_sort(A, 0, len(A) - 1), A\_merge)*

*time\_heap = measure\_time(heapsort, A\_heap)*

*#Display results*

*print(f"Input Size: {size}, Case: {case}")*

*print(f"Quicksort Time: {time\_quick:.4f} seconds")*

*print(f"Merge Sort Time: {time\_merge:.4f} seconds")*

*print(f"Heapsort Time: {time\_heap:.4f} seconds")*

*print()*

Analysis of Results  
1. Quicksort  
Best Case - The time complexity is O(n log n) when the pivot element divides the array into nearly equal parts.   
In the event that the pivot is consistently the smallest or largest element (as in sorted or reverse-sorted arrays without optimization), Quicksort degenerates to O(n^2).   
On random data, Quicksort operates efficiently with an average time complexity of O(n log n). It is more efficient than Heapsort and Merge Sort in practice because of its low overhead in partitioning and fewer memory accesses.  
  
  
2. Merge Sort  
Best, Worst, and Average Case: Merge Sort consistently executes in O(n log n) time, whatever the input distribution. This yields a greater degree of predictability than Quicksort, particularly in the most dire scenarios. However, its temporary arrays necessitate additional memory, which may prove detrimental.   
As a result of the constant factor overhead associated with merging, Merge Sort may be slower than Quicksort, despite its consistent performance across all input types.  
  
3. Heapsort  
Best, Worst and Average Case - Heapsort, similar to Merge Sort executes O(n log n) in all scenarios. As it consistently executes heap construction and extraction operations in the same manner, it does not capitalize on pre-sorted data.  
In practical terms, heapsort is slightly slower than Quicksort due to the fact that heap operations (insertion and extraction) necessitate a greater number of comparisons and swaps than Quicksort's partitioning methodology.  
Heapsort may exhibit superior performance over Quicksort when dealing with reverse-sorted data, particularly when Quicksort's pivot selection is suboptimal. Nevertheless, the time complexity of maintaining the heap structure will result in a generally slower performance than Merge Sort for all types of inputs.

# **Priority Queue & Scheduler Implementation**

**IMPLEMENTATION  
  
Python code which has Priority Queue & Sceduler combined in one py file and can be compiled.****ANALYSIS**  
**1. Data Structure**

* A binary heap is represented by (self.heap = ()). This is chosen for its simplicity and efficiency in managing the heap operations (insert, extract etc.).

**2. Task Representation**

* The **Task class** represents individual tasks with attributes such as task\_id, priority, arrival\_time and deadline. Tasks are scheduled based on their priority. Here we are using a max-heap where tasks with the highest priority are to be processed first.

**3. Core Operations**

a. insert(task)

* Insertion into the heap is handled by joining new task to the end of the list and then bubbling it to maintain max-heap property. This ensures that the parent node has the highest priority than it’s children.
* Time Complexity - O(log n) because the task needs to bubble up.

*def insert(self, task):*

*self.heap.append(task)*

*self.\_bubble\_up(len(self.heap) - 1)*

b. extract\_max():

* This operation removes & returns task with the highest priorit. The last task in the heap replaces the root and then heap is bubbled down.
* Time Complexity - O(log n) because the task needs to bubble down the heap.

*def extract\_max(self):*

*if self.is\_empty():*

*raise IndexError("Priority queue is empty")*

*max\_task = self.heap[0]*

*self.heap[0] = self.heap.pop()*

*self.\_bubble\_down(0)*

*return max\_task*

c. increase\_key(task\_id, new\_priority):

* This operation adjusts the priority of the task. First it finds the task by task\_id, then it updates the priority and then it will bubble the task up or bubble it down.
* Time Complexity - O(n) to find the task in the list O(log n) to reposition it in heap. And in the worst-case time, time complexity is O(n + logn).

*def increase\_priority(self, task\_id, new\_priority):*

*index = self.\_find\_task\_index(task\_id)*

*if index is None:*

*raise ValueError("Task not found")*

*old\_priority = self.heap[index].priority*

*self.heap[index].priority = new\_priority*

*if new\_priority > old\_priority:*

*self.\_bubble\_up(index)*

*else:*

*self.\_bubble\_down(index)*

d. is\_empty():

* An operation to check if the priority queue is empty or not.
* Time Complexity - O(1). It only checks length of the list.

*def is\_empty(self):*

*return len(self.heap) == 0***Time Complexities summarized below for priority queue operation**

| **Operation** | **Worst-Case Time Complexity** | **Best-Case Time Complexity** |
| --- | --- | --- |
| insert(task) | O(log n) | O(1) |
| extract\_max() | O(log n) | O(1) |
| increase\_key(task\_id, new\_priority) | O(n + log n) | O(n) |
| is\_empty() | O(1) | O(1) |