**Heap Data Structures - Implementation, Analysis, and Applications**

Muneeb Mohiuddin

Algorithms and Data Structures

University of the Cumberland’s

September 15, 2024

**Introduction**

Heap data structures are one of the most important data structures used in computer science, especially regarding algorithms and the way data is stored. They lie at the heart of significant numerical algorithms, such as Heapsort and priority queues. It is also a binary tree-based structure where it follows the heap property; for a max heap, none of the parent nodes are lesser than its child nodes, while in a min heap, none of the parent nodes are more significant than that of the child nodes (Wise et al., 2020). The Heapsort algorithm and the priority queue are discussed, and their applications and the time and space complexity of the implementation are discussed in this report.

**Heapsort Implementation**

The Heapsort algorithm is a simple and efficient comparison-based sorting algorithm that forms a max-heap from an unsorted array, gradually removes the largest (root) element, and puts the sorted array in place. The algorithm consists of two main phases:

1. **Constructing the max-heap** means rearranging the items in an array so that the max-heap remains invariant. Specifically, it can be observed that building the max-heap takes O(n) time.
2. **Extracting the maximum element –** Once the max heap has been built, the first element of the heap, which has the highest value, is exchanged with the last element of the heap. It is decreased by one, and the heap order is maintained by Heapifying the root node of that heap. This step is done n times; hence, the time complexity of this process is O(n log n ).

There are several benefits associated with the Heapsort algorithm. First, it invariably ensures O(n log n) time complexity, whether we use best-case, average-case, or worst-case; the other sorting algorithms, such as Quicksort, have a worst-case time complexity of O(n²). Second, Heapsort is an in-place sorting algorithm, meaning it needs only a constant amount of additional space, namely O(1). However, there is a disadvantage of Heapsort, which is used to sort the elements in a non-stable method. The equal elements can randomly be placed after sorting is done (Marcellino et al., 2021).

Thus, an executed Heapsort in Python was done, and the heap was implemented using an array to resemble the binary tree. The code starts from the middle of the array backward to construct the max-heap; it then continuously invokes the heapify procedure to reinstall the heap property each time the maximum element is removed. Concisely, the implementation has no serious issues and is highly structured and formalized to follow most algorithmic paradigms.

**Time Complexity of Heapsort**

* **Best Case –** When all items are sorted in descending order, the best case time complexity arising with Heapsort is O(n log n). To a certain extent, the input distribution does not depend on time building the heap or extracting the elements; in other words, the number of operations stays constant regardless of whether the input is already sorted or random.
* **Worst Case –** Even if it is in the worst case, Heapsort will still be O(n log n). The worst case is the maximum heapify, which can be n times where each heapify takes log base 2 of n; hence, the complexity of this algorithm remains O (n log n).
* **Average Case –** Both best-case and worst-case time complexity of Heapsort are optimal and efficient: The average case time complexity is O(n log n).

**Comparison with Other Sorting Algorithms**

To quickly compare Heapsort, Quicksort, and Merge Sort, I experimented with arrays of different sizes and sorted random and reverse-sorted distributions. The results verified that the time complexity of Heapsort arrived at O(n log n) in all tests conducted while Quicksort, regarded as one of the fastest sorting algorithms, was as fast as its promise for average cases; however, it was O(n²) for worst cases. Like the algorithm above, Merge Sort also achieved the time complexity of O(n log n). However, the recursive nature of the algorithm and the memory required to merge the parts made it necessary to use extra space.

The advantage of Heapsort is that it always takes the best of case time regardless of the size or distribution of the input; however, because of the overhead associated with heap operations, Heapsort is generally slower than Quicksort.

**Priority Queue Implementation**

A priority queue is one of the abstract data types that allows for the inserting of an element in a data structure along with its priority and always makes the element with the highest priority accessible. They are commonly employed in scheduling algorithms, resource management, and real-time systems. Heap is useful for implementing a Priority queue because it has high insertion and extraction operations (Petrovic, 2016).

In this problem, I coded Priority Queue using a max-heap accomplished by an array. I also defined a Task class with parameters that include the task's ID, priority, arrival time, and deadline. These operations are as follows: insert(task), extract\_max(), increase\_key() and is\_empty().

**Time Complexity Analysis of Priority Queue Operations**

* **Insert Operation –** The insert operation adds a new task to the heap and then rearranges the heap by calling heapify\_up, which will take O(log n) time because it rises through the height of the binary tree.
* **Extract Maximum –** The extract\_max function extracts the root node's value and deletes it, while heapify\_down rearranges the heap. The running time complexity is analyzed to be equal to log n.
* **Increase/Decrease Key-** This operation involves precisely rearranging the task heap by heapify\_up or heapify\_down operations, performed in O(logn).
* The **is\_empty()** operation only checks the size of the heap; hence, the time complexity of the operation is O(1).

**Design Choices**

This is because the priority queue requires the use of an array-based heap where the elements are accessed qualitatively by their index in instances of the heapify operation. Further, the required implementation of the array does not have the overhead the linked structure does. I chose max-heap since it is most beneficial to select tasks with the highest priority in scheduling systems.

**Conclusion**

The general discussion of Heapsort and work with priority queues having heaps as the data structure proves that heaps are instead helpful in sorting and scheduling tasks. Heapsort gives us confidence in O(n log n) particular sorting method, and priority queues allow efficient tasks managing operations with the time complexity of O(log n). The two algorithms are applicable in natural settings broadly, such as timing, resource management, and many others. Thus, the analysis and the findings from our experiments support the theoretical time complexities for these algorithms while giving an understanding of their effectiveness.

GitHub Link: <https://github.com/mmohiuddin08954/Assignment_4>

**References**

Marcellino, M., Pratama, D. W., Suntiarko, S. S., & Margi, K. (2021, October). Comparative of advanced sorting algorithms (quick sort, heap sort, merge sort, intro sort, radix sort) based on time and memory usage. In *2021 1st International Conference on Computer Science and Artificial Intelligence (ICCSAI)* (Vol. 1, pp. 154-160). IEEE.

Petrovic, D. (2016). Verification of selection and heap sort using locales.

Wise, J., Bader, J., Wong, C., Aldrich, J., Tanter, É., & Sunshine, J. (2020). Gradual verification of recursive heap data structures. *Proceedings of the ACM on Programming Languages*, *4*(OOPSLA), 1-28.