Report: Heap Data Structures: Implementation, Analysis, and Applications

Avinna Bhattarai

005028547

MSCS-532-A01

Assignment 4

Dr. Vanessa Ruth Cooper

09/15/2024

**Introduction**

This report explores the implementation and analysis of heap data structures and their applications. Specifically, we focus on Heapsort and priority queue operations. Heapsort is a comparison-based sorting algorithm with consistent time complexity across all cases. Priority queues, implemented using heaps, are widely used in scheduling algorithms, where the task with the highest or lowest priority is executed first.

**Heapsort Implementation**

**Design Choices**  
We implement the Heapsort algorithm using an array to represent the binary heap. An array is chosen due to its efficient memory layout and ease of index-based operations (parent and children). Heapsort works by first constructing a max-heap, then repeatedly extracting the maximum element to sort the array.

**Implementation Details**  
The Heapsort algorithm is divided into three main steps:

1. **Build Max-Heap**: Construct a max-heap from the unsorted array.
2. **Extract Maximum**: Swap the root of the heap (maximum element) with the last element of the array.
3. **Heapify**: Restore the heap property after each extraction.

**Time Complexity**

* **Building Max-Heap**:
* **Extracting Max and Restoring Heap**: for each of the n elements.
* **Total Time Complexity**: in the worst, average, and best cases.

**Space Complexity**  
Heapsort is an in-place sorting algorithm using no extra space other than the input array, resulting in a space complexity of ).

*Note: Please refer to* ***heapsort.py*** *in the repository*

**Priority Queue Implementation**

**Design Choices**  
We choose a max-heap to represent the priority queue, where tasks with the highest priority are dequeued first. For efficiency, the heap is represented using an array. A Task class is designed to store task attributes such as task\_id, priority, and deadline.

**Implementation Details**  
The priority queue supports the following operations:

* **Insert Task**: Adds a task to the heap while maintaining the heap property.
* **Extract Max**: Removes the task with the highest priority from the heap.
* **Increase/Decrease Priority**: Changes the priority of a task and adjusts its position in the heap accordingly.

**Time Complexity**

* **Insert**:
* **Extract Max**:
* **Increase/Decrease Priority**:
* **is\_empty**:

*Note: Please refer to* ***priority\_queue.py*** *in the repository*

**Empirical Comparison**

**Experimental Setup**

To compare the performance of Heapsort, Quicksort, and Merge Sort, we tested each algorithm with three types of input distributions:

* **Random**: An unsorted array of random integers.
* **Sorted**: An array where elements are already sorted in ascending order.
* **Reverse-Sorted**: An array sorted in descending order (the worst-case for some algorithms).

We used array sizes of n = [1000, 5000, 10000, 50000, 100000] and recorded the execution times for each sorting algorithm.

For the **priority queue**, we used a task scheduling scenario where tasks arrive with random priorities, and we measured the time taken to insert tasks and extract tasks in order of priority. We simulated a scenario where 1000 tasks are added and processed in batches of 10, measuring the insertion and extraction times.

The code for timing the sorting algorithms and priority queue operations can be found in ***comparison\_sort.py***in the repository.

**Results of Sorting Algorithms**

**Table 1**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Array Size** | **Algorithm** | **Random Array** | **Sorted Array** | **Reverse-Sorted Array** |
| **1000** | Heapsort | 0.00346s | 0.0049 s | 0.0051 s |
|  | Quicksort | 0.00015s | 0.0003 s | 0.0012 s |
|  | Merge Sort | 0.00014s | 0.0006 s | 0.0011 s |
| **5000** | Heapsort | 0.13424s | 0.0261 s | 0.0278 s |
|  | Quicksort | 0.00089s | 0.0025 s | 0.0064 s |
|  | Merge Sort | 0.00083s | 0.0036 s | 0.0048 s |
| **10000** | Heapsort | 0.07365s | 0.0513 s | 0.0557 s |
|  | Quicksort | 0.00243s | 0.0048 s | 0.0125 s |
|  | Merge Sort | 0.00271s | 0.0070 s | 0.0095 s |
| **50000** | Heapsort | 0.29995s | 0.2704 s | 0.2856 s |
|  | Quicksort | 0.01091s | 0.0363 s | 0.0891 s |
|  | Merge Sort | 0.01884s | 0.0435 s | 0.0594 s |
| **100000** | Heapsort | 1.32506s | 0.5635 s | 0.5928 s |
|  | Quicksort | 0.05769s | 0.0784 s | 0.1807 s |
|  | Merge Sort | 0.07552s | 0.0916 s | 0.1278 s |

**Analysis of Sorting Algorithm Performance**

* **Heapsort**:  
  Heapsort showed consistent performance across all input types due to its O(n log n) time complexity in all cases. It performed exceptionally well on sorted and reverse-sorted arrays, as the heap structure is built similarly regardless of initial order. However, it was generally slower than Quicksort and Merge Sort for all input sizes, particularly for random arrays. This can be attributed to its higher constant factor, especially in maintaining the heap property.
* **Quicksort**:  
  Quicksort was the fastest algorithm for random and sorted arrays, leveraging its average-case time complexity of O(n log n). However, it exhibited significantly slower performance on reverse-sorted arrays due to its worst-case time complexity of O(n^2) when the pivot selection is poor (as expected). Despite its theoretical drawbacks, Quicksort benefits from a better cache locality, which explains its superior performance on smaller input sizes.
* **Merge Sort**:  
  Merge Sort maintained a consistent performance across all input types, with results close to Quicksort in many cases. Its O(n log n) complexity in all cases and stable sorting behavior make it a reliable choice for larger input sizes, though it tends to consume more memory due to its space complexity of O(n).

**Priority Queue Task Scheduling**

To simulate task scheduling with a priority queue, we inserted 1000 tasks with random priorities and processed them in batches of 10. The results below show the time taken to insert tasks into the priority queue and extract the highest-priority tasks.

**Results of Priority Queue Operations**

**Table 2**

|  |  |  |
| --- | --- | --- |
| **Task Count** | **Average Insertion Time** | **Average Extraction Time (batch of 10)** |
| 1000 | 0.00045 s | 0.00078 s |

**Analysis of Priority Queue Performance**

* **Insertion Time**:  
  Insertion into the priority queue took time, as expected. For 1000 tasks, the average insertion time was relatively low, making the priority queue efficient for task scheduling applications.
* **Extraction Time**:  
  The extraction of the highest priority task also demonstrated an time complexity, with an average extraction time of 0.00078 seconds per batch of 10 tasks.

The results suggest that heaps are highly effective for managing priority queues, especially when scheduling large numbers of tasks with varying priorities.

**Conclusion**

This empirical comparison highlights the strengths and weaknesses of Heapsort, Quicksort, and Merge Sort. Heapsort is a reliable and stable algorithm for all input types but tends to be slower compared to Quicksort and Merge Sort for random data. Meanwhile, priority queues implemented using heaps show excellent performance in real-world scheduling scenarios, where both insertion and extraction are time efficient.