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

Author: Raghul Krishnan
Date: 01-26-2025

Heapsort Implementation and Analysis

1. Implementation

Heapsort is implemented using a max-heap. Key steps include:

1. Building a max-heap from the input array.
2. Repeatedly extracting the maximum element and placing it at the end of the array.
3. Maintaining the heap property after each extraction.

GitHub implementation link

2. Analysis

- **Time Complexity**:
  - **Building the heap**: $O(n)$. The heapify operation runs in $O(\log n)$ time, and it is called for $n/2$ elements during heap construction.
  - **Extracting elements**: Each extraction requires $O(\log n)$, and this is performed n times.
  - Total complexity: $O(n) + O(n \log n) = O(n \log n)$.
  - **Best, Average, and Worst Case**: $O(n \log n)$ in all cases because heap operations do not depend on input order.
- **Space Complexity**:
  - $O(1)$ additional space since sorting is done in-place. No extra data structures are required beyond the input array.
- **Overhead**:
  - The overhead mainly comes from recursive or iterative calls to maintain the heap property during heapify.

3. Comparison with Quicksort and Merge Sort

- Running Heapsort, Quicksort, and Merge Sort on datasets of varying sizes and distributions (sorted, reverse-sorted, random) shows:
  - Quicksort is often faster on average for random data due to better cache locality.
  - Merge Sort provides consistent performance and excels in stable sorting scenarios.
  - Heapsort performs reliably but can be slower than Quicksort due to additional comparisons during heap operations.
- **Theoretical Analysis**:
  - Quicksort: Average $O(n\log n)$, Worst $O(n^2)$.
  - Merge Sort: $O(n\log n)$ in all cases, $O(n)$ space complexity.
  - Heapsort: Consistent $O(n\log n)$ with $O(1)$ space.

Priority Queue Implementation and Applications

Part A: Priority Queue Implementation

GitHub implementation link

1. **Data Structure**:
   - Used a Python list to represent the binary heap. This allows for efficient implementation of heap operations via array indexing.
   - Task objects are used to represent individual tasks, including fields like task ID, priority, and other metadata.
   - A max-heap is chosen as it suits applications like scheduling, where the highest priority task should be processed first.
2. **Core Operations**:
   - **insert(task)**:
     - Adds a new task to the heap, places it at the end, and adjusts the heap property by "bubbling up."
     - **Time Complexity**: $O(\log n)$.

- **extract_max()**:
    - Removes and returns the task with the highest priority (root of the heap). The last element replaces the root, and the heap property is restored by "bubbling down."
    - **Time Complexity**: $O(\log n)$.
- **increase_key(task, new_priority)**:
    - Updates a task's priority and ensures the heap property by bubbling up if the priority increases or down if it decreases.
    - **Time Complexity**: $O(\log n)$.
- **is_empty()**:
    - Simple check to see if the heap is empty.
    - **Time Complexity**: $O(1)$.

3. **Scheduling Simulation**:
    - Tasks are dynamically added and processed based on priority, demonstrating real-world scenarios like CPU scheduling or event management.
    - Observed behavior aligns with theoretical expectations; higher-priority tasks are consistently processed before lower-priority ones.

Design Choices and Observations

1. **Heap Representation**: A list-based heap is simple and efficient due to the direct mapping of parent-child relationships in a binary tree to array indices.
2. **Task Priority**: Using custom objects allows flexibility in storing additional metadata while prioritizing operations efficiently.
3. **Max-Heap**: Ideal for scenarios requiring the highest-priority task first, e.g., event management or job scheduling.

Conclusion

- Heapsort provides a reliable, in-place sorting solution with consistent $O(n \log n)$ performance. However, its real-world efficiency often lags behind Quicksort due to higher overheads.

- Priority queues, implemented using a heap, are versatile tools for efficiently managing dynamic, prioritized data. They find applications in real-time systems, networking, and more.
- The empirical comparison and analysis highlight the trade-offs between sorting and priority queue operations, emphasizing the need to choose the right tool for specific scenarios.