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

Algorithms and Data Structures (MSCS-532-B01)

Shreya Sapkota

005026644

11/10/2024

Heapsort Implementation and Analysis

The Binary Heap Data Structure serves as the foundation for the comparison-based sorting method known as heap sort. It can be thought of as an optimization over selection sort in which the last (or first) element is swapped out for the maximum (or minimum) element first and follow the same process for the remaining elements.

Using the heapify function, we first create a max-heap from the input array by working upward from the last non-leaf node. This guarantees that every subtree preserves the max-heap property, which states that every parent node is larger than its offspring. The largest element in the max-heap is then switched with the array's final unsorted element to extract the maximum element. By doing this, the sorted elements are moved to the end of the array and the heap's effective size is decreased by one. In order to restore the max-heap structure prior to the subsequent extraction, we call heapify once more to preserve the heap property on the decreased heap. Until the array is completely sorted, these procedures are repeated.

Analysis of Implementation

In every situation i.e. worst, average, and best ones, Heapsort's time complexity is O(nlogn). Applying the heapify function from the bottom of the tree upwards while iterating over each non-leaf node makes building the max-heap an O(n) task. The total complexity is O(nlogn) since extracting the maximum element and rebalancing the heap (preserving the heap property) must be done n times, each taking O(logn) time. The space complexity is O(1) because Heapsort uses a fixed amount of space outside of the input array.

Empirical Comparison

Three different array types random, sorted, and reverse-sorted, each containing ten elements were tested with the Heapsort, Quicksort, and Mergesort algorithms using Python code. The following observations were made:

- Heapsort demonstrated its reliability across various input distributions by consistently sorting all three array types in ascending order. However, because of the extra complexity of preserving the heap structure throughout each extraction stage, Heapsort was typically slower than the other two algorithms.
- For the random input, Quicksort performed better than the other algorithms and had the fastest execution time because of its effective average-case performance. However, because Quicksort is recursive, it may work more slowly on arrays that are sorted or reverse-sorted. In these situations, the performance may deteriorate to O(n2) due to imbalanced partitions.
- Mergesort maintained O(nlogn) complexity while producing consistent and predictable results for all input types. It was almost as fast as Quicksort for bigger or more complicated inputs, but it was a little slower for smaller arrays because of recursion overhead. Another advantage of Mergesort is that, in contrast to Heapsort or Quicksort, it is a stable sort that maintains the relative order of equal components.

Priority Queue Implementation

Using Python's heapq module for effective max-heap operations, a list is used to represent the binary heap. By enabling list-based index management, where the parent and children nodes may be accessed with little overhead, this option simplifies implementation. A max-heap is used to ensure that the tasks with the highest priority are at the top of the heap, providing for constant access to the highest-priority task. The Task class makes it simple to organize and refer to each task within the priority queue by encapsulating its key characteristics, such as task ID, priority, arrival time, and deadline.

Implementation Details

- insert(task): This function "bubbles up" when necessary to preserve the heap structure while adding a new task to the heap. This function is effective for task management since it takes O(logn) time to insert a node into a binary heap while maintaining the heap property.
- extract_max(): By substituting the final element for the root and heapifying this function eliminates and returns the job with the highest priority from the heap. Because of the heap's reordering, this operation has an O(log n) time complexity.
- increase_key(task_id, new_priority): This function modifies a task's heap position to restore the max-heap property when a task's priority changes. The process requires discovering the target task (linear search, O(n)), changing its priority, and "bubbling up" as needed (log n).
- is_empty(): This helper function has an O(1) time complexity and returns True if the heap is empty.

Analysis of Scheduling Task

- insert: O(log n) because the newly added element must bubble up.
- extract_max: O(log n) because the root replacement requires reordering using heapify down.

- increase_key: O(n + log n), where locating the task is O(n) and reordering is O(log n) when the priority is raised.
- is_empty: O(1) since it simply determines the heap list's length.

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). The MIT Press.

Weiss, M. A. (2012). *Data Structures and Algorithm Analysis in Java* (3rd ed.). Pearson Education.