Assignment 4: Heap Data Structures Implementation, Analysis and Applications

Pavani Chavali

Department of Computer Science, University of The Cumberlands

MSCS-532-B01: Algorithms and Data Structures

Professor. Brandon Bass

Date: 11/09/2025

Analysis of Heap sort:

The time complexity for Heapsort is O(n log n) for the worst, average, and best cases. It's one of the few comparison-based sorting algorithms with a consistent and optimal time complexity.

This consistency comes from the two main phases of the algorithm:

Max Heap (max_heap):

This step rearranges the initial array into a max-heap structure.

It does this by calling heapify on the n/2 non-leaf nodes.

While a single heapify call is O(log n), a tighter analysis of all calls during the build phase shows that the total work is linear.

Time: O(n)

Sort (Extraction Phase)

This phase involves a loop that runs n-1 times.

In each iteration, it swaps the root (max element) with the last element of the heap (O(1)).

It then reduces the heap size by one.

It calls heapify on the new root to restore the max-heap property. The heap size k decreases from n-1 down to 1.

Each heapify call takes O(log k) time.

The total time for this phase is the sum of these calls: O(log(n-1)) + O(log(n-2)) + ... + O(log 1).This sum is mathematically proven to be O(n \log n).

Time Complexity Analysis:

Basically, the time complexity is the sum of the two phases which is build an sort

O(n) (build) + O(n logn)(sort) = O(n log n)

Worst Case: O(n logn) The O(n log n) extraction phase is the bottleneck. This time complexity occurs regardless of the initial data order. The heapify calls will always take time proportional to the heap's height.

Average Case: O(n log n) On average, the heapify calls still require logarithmic time to bubble elements down, leading to the same O(n log n) complexity.

Best case: O(n log n) This is a key characteristic of Heapsort. Unlike algorithms like Insertion Sort, it does not get faster is the array is already sorted.

Build_heap phase will still run in O(n)

The extraction phase will still perform all n-1 swaps and heapify call. Even if the array is sorted the algorithm must execute its full logic.

Space Complexity: O(1)

Explanation for O(n log n) for all the cases:

The time complexity for Heapsort is always O(n log n) because its operation is not data dependent it performs the same set of steps regardless of the input order. The sorting phase of the algorithm consists of a loop that must run n-1 times, In every iteration it performs a swap (O(1)) and then calls heapify to restore the heap property. The heapify operation's cost is proportional to the height of the heap, which is **O(log k)** (where k is the current heap size).

Overheads of heapsort:

Recursive Overhead:

The recursive implementation of heapify does incur a small overhead. It uses the function call stack. In the worst case, the recursion depth is equal to the height of the heap, which is O(log n). Therefore, the recursive version technically has an O(log n) space complexity. An iterative version of heapify avoids this and achieves true O(1) space.

Poor Cache Locality:

Modern CPUs are fastest when accessing memory locations that are close together. Heapsort's heapify function breaks this by jumping around the array, comparing a parent at index i with its distant children at 2*i + 1 and 2*i + 2. This random-access pattern leads to many cache misses, forcing the CPU to wait for data from slower main memory. Algorithms like Quicksort and Merge sort, which scan memory sequentially, perform better in practice because they are more cache-friendly.

Stability:

In terms of stability heap sort is not stable because, after sorting it doesn’t maintain any original relative order.

Observations:

From my observations after taking different input data like Random data , sorted and Reverse sorted data, on Random data quick sort is fastest in executing the program in sorting sense, next comes the Merge and slowest is Heapsort. Despite Heapsort have the same O(n log n) time complexity it is practically slowest of all three. In case of sorted and reverse sorted data (Worst Case) Heapsort and Merge sort will be on same position and quick sort will be the slowest one, this scenario is when the pivot is taken as first or last element, making ot exceptionally slower than the others. Heapsort and Merge sort don't have any effect by the input order and maintain their stable O(n log n) performance.

Priority Queue Implementation GitHub Link:

I choose to implement priority queue using list in python as it most efficient way to implement priority queue as a binary heap. Mainly I want to discuss on two advantages Efficiency and simplicity of writing and understanding code.

High Efficiency:

Excellent Cache Locality: The most important performance reason. In array we can see all the elements are stores right next to each other in a single and continuous block of memory. When the algorithm sifts element up and down the next element needs to access are often already in the CPU'S fast cache. This is much faster than following pointers, which can lead to cache misses by jumping to random, scattered memory locations.

O(1) Space Overhead: The array uses exactly O(n) space ofr n elements. There is no extra memory waster on storing pointers for every single node which would be required in a linked tree structure.

Simplicity of implementation:

No pointers: All tree navigation is done with simple arithmetic and for any element at index i:

Parent is in (i-1)//2 and left child 2*i+1 position and right child 2*i+2 position

This completely eliminates the complexity of managing node objects, pointers and manual memory allocation, making the code much cleaner and less bugs.

Time Complexity:

The program has different functions created to perform different tasks like inserting, checking for empty table, deletion and so on

For peeking and check for empty function, time complexity would be everytime O(1).

Logarithmic Time Operations like insert, extract_min has O(logn) since insert function appends to end(O(1)) then calls heapify_up (O(log n)) time complexity and extract_min swaps root with last element (O(1)), then calls heapify_down (O(logn)).

Only heapify_up and down will O(logn) time complexity.

GitHubLink:

https://github.com/PavaniChavali135/AlgoandDatastructures/tree/main/week_3/Asgmnt4