Module 5 - Advanced Sorting Algorithms & The Heap Data Structure

1. Motivation: The Quest for the Ideal Sorting Algorithm

In our study of advanced sorting algorithms, we have explored powerful techniques that offer significant performance improvements over basic $O(n^2)$ methods. However, these advanced algorithms often come with their own set of trade-offs. Let's recap two of the most prominent examples: Merge Sort and Quick Sort.

A Recap of Trade-offs

Merge Sort

- **Strength:** Its greatest advantage is its **guaranteed $\Theta(n \log n)$ performance**. This efficiency holds true for all cases—worst, average, and best—making it exceptionally reliable and predictable.
- **Weakness:** It is not an in-place algorithm. Merge Sort requires auxiliary memory proportional to the size of the input array, giving it a space complexity of **$O(n)$**. This can be a significant drawback when memory is limited.

Quick Sort

- **Strength:** It is an **in-place** algorithm, requiring only a small, logarithmic amount of space on the recursion stack (**$O(\log n)$**). In the average case, it is often faster in practice than other $O(n \log n)$ algorithms due to its low constant factors.
- **Weakness:** Its primary disadvantage is its worst-case performance. If the pivot selection is poor (e.g., on an already sorted array), its performance degrades significantly to a slow **$\Theta(n^2)$**, which is no better than basic sorting methods like Bubble Sort.

The Key Question

This analysis of trade-offs leads to a crucial question: **Is there an algorithm that offers the "best of both worlds"?** Can we find a sorting method that combines:

1. The guaranteed **$\Theta(n \log n)$** worst-case performance of Merge Sort.
2. The **$O(1)$** space efficiency (in-place nature) of Quick Sort.

The Answer: Heap Sort

The answer is **yes**, and one of the most classic algorithms that achieves this powerful combination is **Heap Sort**. It stands as a testament to how the right choice of an underlying data structure can lead to an algorithm with an excellent performance profile.

To fully understand how Heap Sort achieves this, we must first dive into the data structure that powers it: the **Heap**. The following sub-modules will build our understanding from the ground up, starting with the basic concepts of trees and leading to the full implementation and analysis of Heap Sort.

2. Prerequisite: An Introduction to the Tree Data Structure

Before we can understand the Heap, we must first be familiar with its underlying structure: the Tree. In computer science, a **tree** is a widely used data structure that simulates a hierarchical structure, with a set of connected nodes.

Core Components

Every tree is composed of a few fundamental components:

- **Node:** The primary entity in a tree that contains data. It is also known as a "vertex".
- **Edge:** A connection or link between two nodes.
- **Root:** The topmost node in a tree. It is the only node that does not have a parent.

Consider the tree structure below:

- Nodes: {A, B, C, D, E}
- Edges: {(A-B), (A-C), (B-D), (B-E)}
- Root: Node A

Hierarchical Terminology

The relationships between nodes in a tree are described using terminology borrowed from family trees:

- **Parent:** A node that is directly above another node and connected to it. In our example, A is the parent of B and C.
- **Child:** A node that is directly below another node and connected to it. In our example, D and E are children of B.
- **Leaf:** A node that has no children. In our example, C, D, and E are leaf nodes.

Structure and Depth

We can measure the structure of a tree in several ways:

- **Level:** The level of a node is defined by its distance from the root. The root is at **Level 0**. Its children are at Level 1, their children are at Level 2, and so on.
- **Depth/Height:** The height of a tree is the number of edges on the longest path from the root to a leaf. Our example tree has a height of 2.

Focus on the Binary Tree

While a node in a tree can have any number of children, our focus for understanding heaps will be on a specific type: the **Binary Tree**.

A **Binary Tree** is a tree data structure in which each node has **at most two children**, which are referred to as the left child and the right child.

The reason we focus on Binary Trees is that the **Heap** data structure, which is the engine of Heap Sort, is a specialized type of Binary Tree. Mastering this concept is a fundamental step toward understanding heaps.

3. The Heap Data Structure

Now that we understand the concept of a binary tree, we can define a Heap. A **Heap** is a specialized tree-based data structure that satisfies two specific properties.

The Two Defining Properties of a Heap

For a binary tree to be considered a Heap, it must adhere to the following rules:

1. **Structure Property: It must be an Essentially Complete Binary Tree.**
   This means that the tree is completely filled on all levels, with the possible exception of the last level, which must be filled from **left to right** without any gaps. This "no gaps" property is crucial, as it allows a heap to be stored efficiently in an array.
2. **Heap Property (Order Property):** The nodes must be ordered in a specific way relative to their children. This ordering defines the type of heap.

Types of Heaps

There are two main types of heaps, distinguished by the Heap Property they enforce:

- **Max-Heap:** The value of every node is **greater than or equal to** the value of each of its children.
  - **Implication:** In a Max-Heap, the **largest element** in the entire collection is always located at the **root** of the tree. This is the type of heap used for Heap Sort.
  - Example: A tree with root 10 and children 8 and 9 is a valid Max-Heap at the top level.
- **Min-Heap:** The value of every node is **less than or equal to** the value of each of its children.
  - **Implication:** In a Min-Heap, the **smallest element** is always located at the **root**. This structure is commonly used to implement Priority Queues.
  - Example: A tree with root 2 and children 5 and 3 is a valid Min-Heap at the top level.

The Array Representation

The complete binary tree property is precisely what makes an array a perfect and highly efficient way to store a heap. The hierarchical relationships are not stored with pointers, but are instead calculated mathematically based on an element's index.

For an element at a **zero-based index i** in an array representing a heap:

- The index of its **parent** is calculated as: floor((i - 1) / 2)
- The index of its **left child** is calculated as: 2 * i + 1
- The index of its **right child** is calculated as: 2 * i + 2

For example, for the node at index i = 3:

- Its parent is at index floor((3 - 1) / 2) = floor(1) = 1.
- Its left child would be at index 2 * 3 + 1 = 7.

4. Core Operations and the Heap Sort Algorithm

The Heap Sort algorithm is a two-phase process that masterfully uses the properties of the Max-Heap. Both phases rely on a core "helper" operation that maintains the heap property.

The Engine of the Heap: The siftdown Operation

To build and maintain a heap, we need a procedure to fix any violations of the heap property. The primary operation used in Heap Sort is siftdown (also known as heapify-down).

- **siftdown (or Heapify-Down):** This operation is used when a node's value is **smaller than** one of its children, violating the Max-Heap property. The siftdown process corrects this by repeatedly swapping the parent with its **largest child**, effectively "sinking" the smaller element down the tree until the heap property is restored for that subtree. The time complexity for a single siftdown operation is proportional to the height of the tree, making it **O(log n)**.
- **siftup (or Heapify-Up):** While less critical for our Heap Sort implementation, this complementary operation is used when a newly inserted node is **larger than** its parent. It "bubbles" the element up the tree by swapping it with its parent until the heap property is restored. This is the key operation used when adding elements to a std::priority_queue.

With the siftdown operation as our main tool, we can now construct the two phases of Heap Sort.

Phase 1: makeheap (The Heapify Process)

The first step is to convert the unsorted input array into a valid Max-Heap.

- **Goal:** Rearrange the elements of the array so that they satisfy the Max-Heap property.
- **Method (Bottom-Up):** The most efficient way to do this is with a bottom-up approach. We treat the entire array as a complete binary tree and then iteratively fix it. The process starts from the **last non-leaf node** and moves upwards towards the root. For each node, we call siftdown to ensure its subtree is a valid Max-Heap. By the time we reach the root, the entire array is guaranteed to be a Max-Heap.
- **Analysis:** While it involves multiple calls to siftdown (an O(log n) operation), a tight analysis shows that the makeheap phase can be completed in **O(n) linear time**, which is remarkably efficient.

Phase 2: The Sorting Process

Once the array is a Max-Heap, the largest element is at the root (array[0]). The sorting phase systematically extracts this largest element and places it in its correct final position.

This is done through a repeated process:

1. **Swap:** Swap the root element (array[0], the current maximum) with the **last element** in the heap portion of the array. The largest element is now in its final, sorted position at the end of the array.
2. **Shrink:** The effective size of the heap is reduced by one, "locking in" the sorted element at the end so it is no longer considered part of the heap.
3. **Repair:** The new root element (which was previously the last element) likely violates the Max-Heap property. Call siftdown on the root (array[0]) to repair the heap, ensuring the next largest element rises to the top.

This cycle is repeated n-1 times, until the entire array is sorted.

Complexity Analysis of Heap Sort

- **Time Complexity:**
  - Phase 1 (makeheap): **O(n)**
  - Phase 2 (Sorting): Consists of n-1 calls to siftdown, each taking **O(log n)** time. Total time for this phase is **O(n log n)**.
  - The overall complexity is dominated by the sorting phase, giving Heap Sort a guaranteed **Θ(n log n)** time complexity for worst-case, average-case, and best-case scenarios.
- **Space Complexity:**
  - The algorithm operates directly on the input array, swapping elements within it. It requires no significant extra storage. Therefore, Heap Sort is an **in-place** algorithm with a space complexity of **O(1)**.

5. The Heap in Practice: The C++ Standard Template Library (STL)

While understanding how to build Heap Sort from scratch is crucial for algorithmic knowledge, in modern C++, we often leverage the powerful abstractions provided by the Standard Template Library (STL). The concepts of a heap are primarily exposed through std::priority_queue.

std::priority_queue: A Ready-to-Use Heap

The C++ STL provides a container adapter called std::priority_queue that is an implementation of a Heap.

- **Default Behavior:** By default, std::priority_queue<int> is a **Max-Heap**. This means that when you call top(), it returns the largest element, and when you pop(), it removes the largest element while maintaining the heap property.
- **Core Operations:**
  - push(): Adds an element to the queue (heap). This internally performs a sift-up operation. Complexity: **O(log n)**.
  - pop(): Removes the top element. Complexity: **O(log n)**.
  - top(): Returns a reference to the top element without removing it. Complexity: **O(1)**.

Working with Custom Data Types

What happens if we want to create a priority queue of custom objects, like a struct for patients in a hospital?

```
struct Patient {
    std::string name;
    int triage_level; // Level 1 is highest priority
};

// This will cause a compile error!
std::priority_queue<Patient> er_queue;
```

The compiler doesn't know how to compare two Patient objects. To solve this, we must provide our own custom comparison logic.

Custom Comparators: Functors and Lambda Expressions

We can tell std::priority_queue how to order our custom types using a **custom comparator**. There are two common ways to do this in modern C++:

1. **Functor (Function Object):** A struct or class that overloads the function call operator operator(). The priority_queue will create an object of this type and use its operator() to compare elements. This is a powerful, stateful way to define comparison logic.

```
struct ComparePatients {
    bool operator()(const Patient& a, const Patient& b) {
        // Because priority_queue is a Max-Heap, it puts the "larger"
        // element on top. We want level 1 to be the highest priority,
        // so we tell the queue that 'a' is "less than" 'b' if its
        // triage level is numerically greater.
        return a.triage_level > b.triage_level;
    }
};

// Usage:
std::priority_queue<Patient, std::vector<Patient>, ComparePatients> er_queue;
```

2. **Lambda Expression:** An inline, anonymous function that can be defined on the spot. Lambdas are often more concise and readable for simple, stateless comparison logic. They are commonly used with algorithms like std::sort.

```
auto compare_lambda = [](const Patient& a, const Patient& b) {
    return a.triage_level > b.triage_level;
};

// Usage with priority_queue is slightly more verbose
std::priority_queue<Patient, std::vector<Patient>, decltype(compare_lambda)>
er_queue(compare_lambda);
```

By providing these comparators, we can leverage the highly optimized and safe implementation of a heap provided by the STL for any data type we need.