🌳 What Is a Partially Ordered Tree?

A **partially ordered tree** is a tree structure where the nodes follow a specific ordering constraint — but **not a total order** like in a binary search tree.

- In a **total order**, every pair of elements can be compared (like in a sorted array).
- In a **partial order**, only certain pairs have a defined relationship.

In tree terms:

- Each node has a relationship with its **children**, but not necessarily with its **siblings** or **descendants further down**.
- The ordering is **local**, not global.

🌳 What Makes a Tree "Partially Ordered"?

In a **partially ordered tree**, each node satisfies a local ordering constraint with its children — but not necessarily with other nodes in the tree.

For a **max-heap**:

- Every parent is **≥** its children.
- No guarantees about sibling relationships or across subtrees.

This local ordering is what makes it *partial* — unlike a binary search tree, which is totally ordered via in-order traversal.

🧠 Why POT Is Powerful

- It's **cheaper** than full sorting: O(log n) insert/delete vs O(n log n) sort.
- It's **flexible**: you can keep adding/removing while maintaining structure.
- It's **modular**: you only enforce order where it matters (parent-child), not everywhere.

✅ Heap Is a Data Structure

- A **heap** is a **partially ordered tree**, often implemented as an array.
- It maintains **local order** (parent ≥ children in max-heap), not global order.
- It's designed for **fast access to the max/min element**, not full sorting.

🔺 Why Is a Heap a Representation of a Partially Ordered Tree?

A **heap** (specifically a binary heap) is a classic example of a partially ordered tree. Here's why:

✅ Heap Property

- In a **max-heap**, every parent node is **greater than or equal to** its children.

- In a **min-heap**, every parent node is **less than or equal to** its children.
- This satisfies a **partial order**: each node is ordered with respect to its children, but not necessarily with other nodes.

🧠 **Local vs Global Ordering**

- The root of a max-heap is the **maximum element**, but the rest of the tree is not globally sorted.
- You can't say anything about the relative order of two sibling nodes or nodes in different subtrees.

🧱 **Structural Properties**

- Heaps are typically implemented as **complete binary trees** (every level filled except possibly the last).
- This structure allows efficient array-based representation and fast access to the root.

📦 **Array-Based Heap Implementation**

Heaps are typically stored in arrays because:

- They're **complete binary trees** (no gaps except possibly at the end).
- You can compute parent/child indices with simple arithmetic.

🧮 **Index Relationships**

For a node at index i:

- **Left child** → 2i + 1
- **Right child** → 2i + 2
- **Parent** → (i - 1) / 2 (integer division)

This lets you navigate the tree without pointers or structs — just raw array access.

🧠 **Why This Matters**

- **Efficiency**: Array-based heaps give O(log n) insert/delete and O(1) access to the max/min.
- **Compactness**: No pointers, no wasted space — just index math.
- **Partial order**: You only maintain local constraints, which is cheaper than full sorting.

🔧 **What Is Heapify?**

Heapify is the process of restoring the heap property in a binary heap — either **bottom-up** or **top-down** — after a change like insertion or deletion.

- Maintains the **heap property** (partial order) in a binary heap.
- Used during insertions (up-heapify) and deletions (down-heapify).

- Also used to **build a heap from an unsorted array** — this is called **heap construction**.

It ensures that:

- In a **max-heap**, every parent is ≥ its children.
- In a **min-heap**, every parent is ≤ its children.

🧱 **Two Types of Heapify**

1. Up-Heapify (a.k.a. "bubble up" or "sift up")

Restores the heap property by moving the new element upward until it's in the correct spot.

Used after **insertion**:

- Insert new element at the end of the array.
- Compare with its parent.
- If it violates the heap property, **swap** and repeat upward.
- Repeat until the heap property is restored or you reach the root.

```
void upHeapify(int heap[], int index) {
    while (index > 0) {
        int parent = (index - 1) / 2;
        if (heap[parent] < heap[index]) {
            swap(&heap[parent], &heap[index]);
            index = parent;
        } else break;
    }
}
```

2. Down-Heapify (a.k.a. "bubble down" or "sift down")

Used when you remove the root (usually the max or min), and replace it with the last element.

Used after **deletion** (usually of the root):

- Replace root with last element.
- Compare with children.
- If it violates the heap property, swap with the **larger child** (max-heap) or **smaller child** (min-heap)
- Repeat until the heap property is restored or you reach a leaf.

```
void downHeapify(int heap[], int size, int index) {
    while (index < size) {
        int left = 2 * index + 1;
        int right = 2 * index + 2;
        int largest = index;

        if (left < size && heap[left] > heap[largest]) largest = left;
        if (right < size && heap[right] > heap[largest]) largest = right;

        if (largest != index) {
            swap(&heap[index], &heap[largest]);
            index = largest;
        } else break;
    }
}
```

Operation	Direction	Triggered By	Typical Use Case
Up-Heapify	Upward	Insertion	Priority queue insert
Down-Heapify	Downward	Deletion (root)	Heap sort, pop max/min

🧠 **Why Heapify Matters**

- It's how heaps **self-correct** after changes.
- Keeps the **partial order** intact.
- Enables efficient O(log n) insertions and deletions.

🔄 **Heap Sort Is a Sorting Algorithm**

- **Heap sort** uses a heap to sort data.
- It builds a heap (partial order), then repeatedly extracts the root (max/min) and reheapifies.
- This gives a **fully sorted array**, but only after **destroying the heap structure**.
- Uses heapify to **build a heap**, then repeatedly:
    1. Swap the root (max/min) with the last element.
    2. Shrink the heap size.
    3. Down-heapify the new root.
- This gradually moves the largest (or smallest) elements to the end of the array.

```
void heapSort(int arr[], int n) {
    // Step 1: Build max-heap
    for (int i = n/2 - 1; i >= 0; i--)
        downHeapify(arr, n, i);

    // Step 2: Extract elements one by one
    for (int i = n - 1; i > 0; i--) {
        swap(&arr[0], &arr[i]);
        downHeapify(arr, i, 0);
    }
}
```

Note. heapify is a **building block** of heap sort — but it's also used independently to maintain the heap structure.

🧱 **Insert vs Heapify in an Empty Heap**

Both operations help build a heap, but they differ in **intent**, **efficiency**, and **use case**.

🔁 **Insert (One-by-One)**

- You start with an empty array.
- Insert elements **one at a time**.
- After each insert, you **up-heapify** to restore the heap property.

✅ **Pros**

- Simple and intuitive.
- Good for **streaming data** or **priority queues**.

❌ **Cons**

- Each insert takes O(log n), so total cost is **O(n log n)**.

```
for (int i = 0; i < n; i++) {
    heap[size++] = input[i];
    upHeapify(heap, size - 1);
}
```

⚙️ **Heapify (Bulk Build)**

- You start with an **unsorted array** of n elements.
- Treat it as a complete binary tree.

- Apply **down-heapify** from the last non-leaf node up to the root.

✅ **Pros**

- Much faster: total cost is **O(n)**.
- Ideal for **heap sort** or **batch construction**.

❌ **Cons**

- Only works when you already have all the data.

```
for (int i = n/2 - 1; i >= 0; i--) {
    downHeapify(arr, n, i);
}
```

🔼 **Up-Heapify and** 🔽 **Down-Heapify: Mechanisms**

These are **generic operations** that apply to both min-heaps and max-heaps — the difference lies in the **comparison logic**.

✅ **In a max-heap:**

- **Up-heapify**: bubble up if the child is **greater than** the parent.
- **Down-heapify**: bubble down by swapping with the **larger** child if the parent is **less than** either child.

✅ **In a min-heap:**

- **Up-heapify**: bubble up if the child is **less than** the parent.
- **Down-heapify**: bubble down by swapping with the **smaller** child if the parent is **greater than** either child.

🖌️ **What Is deletemin?**

- **Used in a min-heap**
- Removes the **minimum element**, which is always at the **root** (index 0)
- After removal:
    1. Replace root with the last element in the array.
    2. Apply **down-heapify** to restore the min-heap property.

Example:

Min-heap array: [2, 5, 3, 10, 8]

- deletemin() removes 2
- Replace root with 8: [8, 5, 3, 10]
- Down-heapify → swap 8 with 3, then with 5 if needed
- Result: [3, 5, 8, 10]

🧹 **What Is deletemax?**

- **Used in a max-heap**
- Removes the **maximum element**, which is also at the **root** (index 0)
- After removal:
    1. Replace root with the last element.
    2. Apply **down-heapify** to restore the max-heap property.

Example:

Max-heap array: [9, 6, 7, 2, 3]

- deletemax() removes 9
- Replace root with 3: [3, 6, 7, 2]
- Down-heapify → swap 3 with 7, then with 6 if needed
- Result: [7, 6, 3, 2]

🧠 Why Use POT for Priority Queues?

A **priority queue** is a data structure where each element has a priority, and you always remove the element with the **highest (or lowest) priority** first.

To support this efficiently, we need:

- **Fast access to the highest-priority element** → must be at a known location.
- **Fast insertion and deletion** → must maintain structure without full sorting.

This is exactly what a **heap (POT)** gives us.

🔺 How POT Helps (in Array Form)

A **binary heap** is a partially ordered tree stored in an array. Here's why it's perfect:

✅ 1. Root Is Always the Highest Priority

- In a **max-heap**, the root (index 0) is the largest element.
- You can access it in **O(1)** time.

✅ 2. Partial Order = Efficient Maintenance

- Only parent-child relationships are ordered.
- You don’t need to sort the entire array — just maintain local order.
- Insertions and deletions cost **O(log n)** thanks to up-heapify and down-heapify.

✅ 3. Array Indexing = No Pointers, No Overhead

- Parent and child indices are computed with simple math:
    - Parent: (i - 1) / 2
    - Children: 2i + 1, 2i + 2
- This makes the structure **compact** and **cache-friendly**.

Simulation of Priority Queue in regards to P.O.T.

Initial Heap After All Insertions

Patients and their priorities:

Patient	Priority
Alice	5
Bob	2
Carol	8
Dave	3
Eve	6

After inserting all and applying **up-heapify**, the array-based max-heap looks like:

Code Copy

```
Index:   0   1   2   3   4
Values: [8, 6, 5, 2, 3]   // Carol, Eve, Alice, Bob, Dave
```

Step-by-Step Serving Simulation

Serve 1: Carol (priority 8)

- Replace root with last: [3, 6, 5, 2]
- Down-heapify:
    - Compare 3 with 6 and 5 → swap with 6
- Result: [6, 3, 5, 2]

Serve 2: Eve (priority 6)

- Replace root with last: [2, 3, 5]
- Down-heapify:
    - Compare 2 with 3 and 5 → swap with 5
    - Compare 2 with no children → done

- Result: [5, 3, 2]

✅ **Serve 3: Alice (priority 5)**

- Replace root with last: [2, 3]
- Down-heapify:
    - Compare 2 with 3 → swap
- Result: [3, 2]

✅ **Serve 4: Dave (priority 3)**

- Replace root with last: [2]
- No heapify needed — only one element

✅ **Serve 5: Bob (priority 2)**

- Remove the last element
- Heap is now empty: []