LECTURE 23

Heap and Priority Queue

CSC211 · Algorithms and Data Structures

COMSATS University Islamabad, Lahore Campus

LECTURE 23

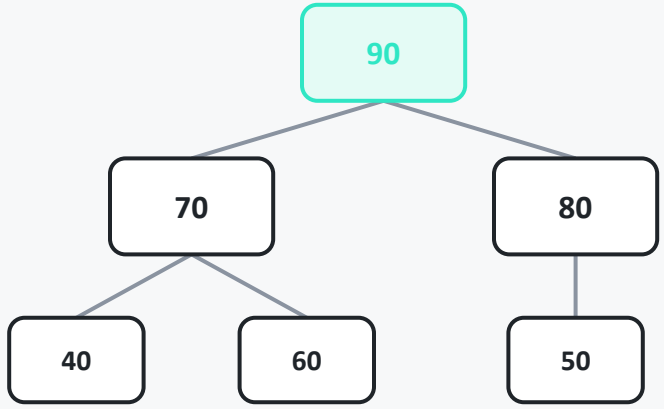
Agenda / Learning Objectives

1. The heap concept and the heap property (min-heap and max-heap)
2. The array representation of a heap, and why completeness makes it work
3. Insertion (heapify up) and deletion (heapify down)
4. Building an entire heap from an unsorted array in O(n)
5. Priority queues on top of a heap, and heap sort (a preview)
6. Common pitfalls when implementing heap operations by hand

CORE CONCEPT

The Heap Concept and Heap Property

A heap is a complete binary tree that additionally satisfies the heap property:



A valid MAX-HEAP — even though 70 > 50, never allowed in a BST

Max-Heap: every parent's value is ≥ both its children's.

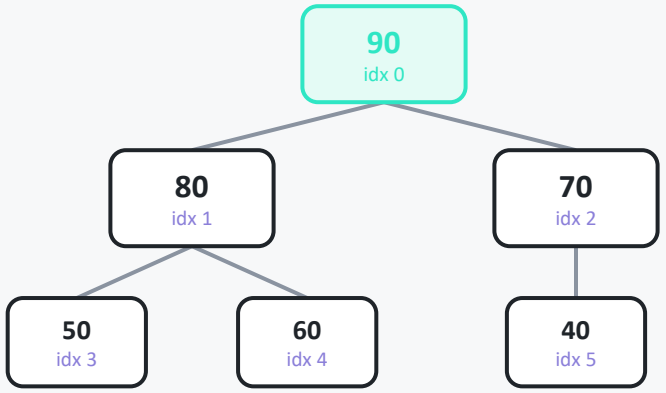
Min-Heap: every parent's value is ≤ both its children's.

What the heap property does NOT promise: unlike a BST, there's no left-vs-right ordering rule — only parent-vs-child. That relaxation is exactly what makes heap insertion and removal faster and simpler than a BST's.

STORAGE

The Array Representation of a Heap

Because a heap is always complete, it needs no pointers at all — the same heap, two representations:



ARRAY (same heap)

90	80	70	50	60	40
0	1	2	3	4	5

```
parent(i) = (i - 1) / 2
left(i)   = 2i + 1
right(i)  = 2i + 2
```

Completeness is what makes this work: no gaps, no wasted space, and a child/parent index can be computed with plain arithmetic — no pointer chasing needed.

INSERTION

Insert, Then Heapify Up

- Add the new value at the very next open array slot (preserving completeness).
- Heapify up: repeatedly swap it with its parent for as long as it violates the heap property.
- Cost: O(log n) — heapify-up walks at most the tree's height.

max_heap.cpp

```
void heapifyUp(int i) {
    while (i > 0 &&
           data[parentIndex(i)] < data[i]) {
        swap(data[parentIndex(i)], data[i]);
        i = parentIndex(i);
    }
}
void insert(int value) {
    data.push_back(value);
    heapifyUp(data.size() - 1);
}
```

Verified output: inserting 50, 80, 40, 90, 60, 70 one at a time produces array [90, 80, 70, 50, 60, 40] — the exact heap on the previous slide.

EXTRACTION

Extract Max, Then Heapify Down

- extractMax always returns the root — guaranteed to be the maximum, by the heap property.
- Move the last element to the root (patching the hole), then let it sink down (heapify down).
- Cost: O(log n) — heapify-down walks at most the tree's height.

max_heap.cpp

```
int extractMax() {
    int maxValue = data[0];
    data[0] = data.back();  // last → root
    data.pop_back();
    if (!data.empty())
        heapifyDown(0);   // sink to place
    return maxValue;
}
```

Common pitfall: heapifyDown must check bounds against the NEW, shrunken size after pop_back — comparing against a stale size risks reading past the end of the array.

BUILD-HEAP

Turning an Array Into a Heap in O(n)

Call heapifyDown on every non-leaf node, from the last non-leaf (index n/2 − 1) back to the root:

BEFORE (index order = insertion order)

0	1	2	3	4	5	
5	3	17	10	84	19	
6	22	9	1	100	12	*(indices 6–11)*

AFTER (valid max-heap)

0	1	2	3	4	5
100	84	19	22	5	17
6	10	9	1	3	12

Watch index 1 (value 3): it takes two swaps to settle — first with index 4, then index 10 — sifting down can cascade through multiple levels.

WHY START FROM THE LAST NON-LEAF?

- Every leaf is already a valid “heap” of size one — nothing below it can violate the heap property.
- For an array of size n, the last non-leaf sits at index n/2 − 1; everything after that is a leaf.

MEASURED

Repeated Insertion vs. Bottom-Up Heapify

The claim “build-heap is O(n), not O(n log n)” is easy to get wrong by intuition. The practical difference, measured directly on the same 12 values:

11 swaps

12 repeated insertions (heapify-up each time)

7 swaps

bottom-up heapify (buildHeap) on the same array

Both produce a valid max-heap (there's no single “correct” heap shape for a given set of values), but the two approaches scale differently: repeated insertion's total swap count grows like **O(n log n)** in the worst case, while build-heap stays **O(n)** — the gap widens as n grows, even though both counts are small enough here to verify by hand.

APPLICATION

Heap-Based Priority Queue

pq_scheduler.cpp

```
struct CompareTask {
  bool operator()(const Task& a,
                  const Task& b) {
    return a.priority < b.priority;
  }   // max-heap by priority
};

priority_queue<Task, vector<Task>,
               CompareTask> scheduler;
```

PROCESSING ORDER (real output)

10	Server is down
9	Patch security vulnerability
3	Update documentation
2	Send weekly report
1	Reply to email

std::priority_queue is a max-heap by default — insert is identical to a heap's, and top()/pop() is extractMax()/dequeue. For min-heap behavior with built-in types, pass greater<int> instead of writing a custom comparator.

ANALYSIS

Heap Sort Preview — and Heap Complexity

Heap sort (preview): repeatedly calling extractMax on a max-heap built from an array, writing each result into a new array from the back forward, produces a fully sorted array — covered fully in Lecture 31.

Operation	Complexity	Why
insert	O(log n)	Heapify-up walks at most the tree's height
extractMax / extractMin	O(log n)	Heapify-down walks at most the tree's height
Peek at max/min	O(1)	Always sitting at index 0
Build-heap from an array	O(n)	Cheaper than n individual O(log n) inserts

WATCH OUT

Common Pitfalls With Heaps

!

Checking bounds against the wrong size after pop_back — use the new, shrunken size, not the old one.

↔

Flipping only one comparison when converting max-heap to min-heap — both heapifyUp's < and heapifyDown's two > comparisons must flip together.

≠

Assuming a heap is sorted — indexing straight across the array is NOT sorted order; only repeated extraction produces one.

△

Forgetting std::priority_queue's inverted comparator convention — it answers “is a lower priority than b,” which is why < gives a max-heap.

Key Takeaways

1. A heap is a complete binary tree with the heap property: parent ≥ (max) or ≤ (min) both children — weaker than a BST's, which is why heap operations are simpler and O(log n).

2. Because a heap is always complete, it's stored efficiently as a plain array — no pointers needed.

3. Insertion appends and heapifies up; extraction removes the root, moves the last element there, and heapifies down — both O(log n).

4. Build-heap turns an unsorted array into a valid heap in O(n), measurably fewer swaps than n repeated insertions.

5. A priority queue is a heap — std::priority_queue is exactly this structure wrapped in a container interface.