HEAPS AND PRIORITY QUEUES

Heaps Properties and Types

Heap Data Structure

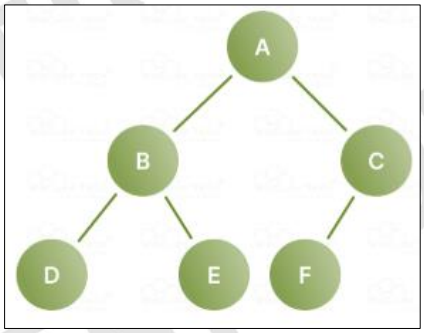
A binary tree in which every node is larger than the values associated with either child. This means the largest or smallest value is always at the top (root) of the tree.

It is widely used in computing for efficient management of priority-based operations, optimal selection, and real-time processing.

Properties of Heap Data Structure

Complete Binary Tree (Shape Property)

A heap is always a complete binary tree wherein all levels are filled except possibly the last, which is filled from left to right without skipping nodes. This allows representation using arrays (index-based), reducing pointer overhead.



This ensures that the tree remains balanced, which is crucial for maintaining the efficiency of heap operations.

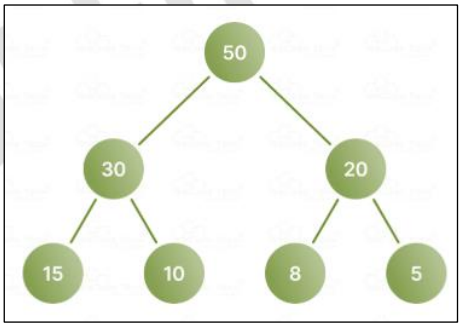
Heap Order Property

The heap must satisfy the heap order property, which differs for max-heaps and min-heaps. These are the two (2) types of heap data structure.

Types of Heap Data Structure

Max-Heap

In a max-heap, the parent node is always greater than or equal to its children. This ensures that the largest value is always at the root of the tree.



- The root node (50) is greater than its children (30 and 20).
- Each parent node (30, 20) is greater than its children (15, 10 for 30 and 8, 5 for 20).

Its use cases include the following:

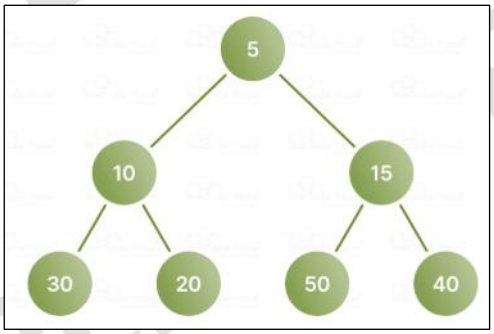
- **Heapsort**: A comparison-based sorting algorithm that uses a binary heap to repeatedly extract the maximum (or minimum) element and place it at the end of the array. It uses a max-heap to sort an array in ascending order by repeatedly extracting the maximum and placing it at the end.
- **Priority Queues**: In this case, the element with the highest priority is accessed first, such as real-time operating systems (RTOS) and emergency resource allocation systems.
- **Top-K Tracking**: The process of efficiently maintaining the K largest or K smallest elements from a continuous stream or large dataset, without sorting the entire input.

  In a max-heap, a max-heap size of K is used to find the K smallest elements. The new elements are compared to the root (largest); if a smaller value, it is replaced.

  This is applied in product ranking systems (bottom-K scores) and real-time performance analytics (slowest transactions).

Min-Heap

In a min-heap, the parent node is always less than or equal to its children. This ensures that the smallest value is always at the root of the tree.



- The root node (5) is smaller than its children (10 and 15).
- Each parent node (10, 15) is smaller than its children (30, 20 for 10 and 50, 40 for 15).

Its use cases include the following:

- **Dijkstra's Algorithm**: Uses a min-heap to extract the next closest vertex with the smallest tentative distance. It is applied in GPS routing, network routing protocols, and game AI pathfinding.
- **Priority Queues**: Min-heaps are the backbone of priority queues, where tasks with the lowest cost or priority are processed first. This is applied in CPU scheduling (shortest-job-first), task scheduling in operating systems, and printer job management.
- **Event-Driven Simulation**: This stores and retries the next event based on the timestamp. Min-heap ensures that the earlier event is always at the top. This is applied in network simulations, manufacturing process simulations, and logistics and supply chain modeling.

The program below demonstrates how to create and use both a Min-Heap and a Max-Heap in Java using the PriorityQueue class.

```
import java.util.PriorityQueue; //importing the PriorityQueue class
import java.util.Collections;

public class HeapExample {
    public static void main(String[] args) {
        // Min-Heap: Natural ordering (smallest element at the head)
        PriorityQueue<Integer> minHeap = new PriorityQueue<>(); //min-heap syntax
        minHeap.add(30);
        minHeap.add(10);
        minHeap.add(20);

        System.out.println("Min-Heap (ascending order):");
        while (!minHeap.isEmpty()) {
            System.out.print(minHeap.poll() + " ");
        }

        System.out.println("\n");

        //  Max-Heap: Reverse ordering (largest element at the head)
       PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Collections.reverseOrder());
        maxHeap.add(30);
        maxHeap.add(10);
        maxHeap.add(20);

        System.out.println("Max-Heap (descending order):");
        while (!maxHeap.isEmpty()) {
            System.out.print(maxHeap.poll() + " ");
        }
    }
}
```

This program helps visualize how **priority queues (heaps)** organize and retrieve values based on order, not insertion.

The output:

```
Min-Heap (ascending order):
10 20 30

Max-Heap (descending order):
30 20 10
```

Heap Operations and Heapify

These are the core operations used to manage a heap, whether it is a min-heap or max-heap:

- **Insert (Push)** – adds a new element to the heap by adding it at the bottom.
- **Extract Root (Pop)** – removes and returns the root element with the last element.
- **Peek (Get Root)** – returns the root without removing it.
- **Replace** – removes the root and inserts a new element in a single operation.

Heapify is the process of converting an arbitrary array (no specific structure or order) into a valid heap. It ensures that the heap property (parent ≥ or ≤ children) is maintained from a given node down to its descendants.

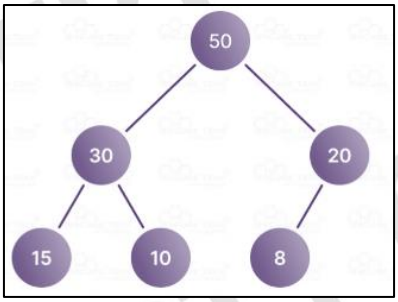
The two (2) heapify operations are Up-Heapify (Bubble-Up) and Down-Heapify (Bubble-Down).

- **Up-Heapify (Bubble-Up)**: Used after inserting a new element to restore the heap property by moving the element up the tree.

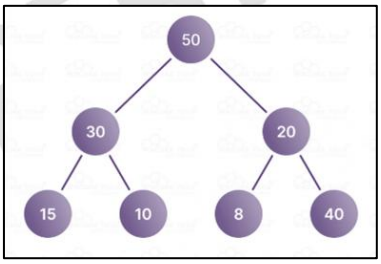
  Steps:
  1. Insert the new element at the end/bottom of the heap.
  2. Compare the inserted element with its parent.
  3. If the heap property is violated (the new element is greater than the parent in a max-heap or smaller than the parent in a min-heap), swap the new element with its parent.
  4. Repeat this process until the heap property is restored or the element becomes the root.

  Example (Max-Heap)

  Initial Heap:



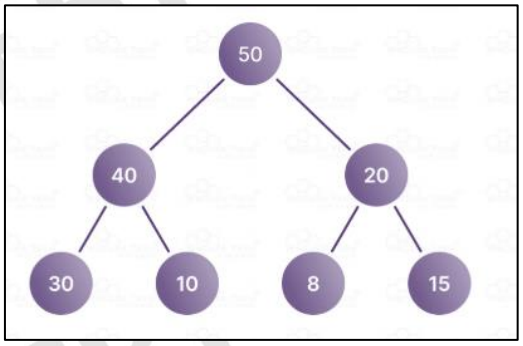
  Insert **40** at the bottom of the heap:



  - **40** is inserted as the **right child of node 20** (bottom right).
  - At this point:
    - Parent of 40 = 20
    - 40 > 20 → violates the max-heap rule
    - So, this can be fixed using up-heapify.

Applying Up-Heapify:

- Insert **40** as the right child of node **20** (bottom right of the heap).
- Compare **40** with parent **20** → **40** > **20** → swap them.
- Now, **40** becomes the parent of 8 and 20.
- Compare **40** with parent **30** → **40** > **30** → swap again.
- Now, **40** becomes the left child of **50**, and **30** becomes the left child of **40**.
  - After the swap, 30 becomes the left child of 40 because complete node positions (not just values) are exchanged in a heap. This ensures the heap maintains both the correct ordering (Heap Order Property) and the complete binary tree structure.
  - In heap operations, a swap affects the entire node and its children, not just values. This is why a former parent may become a child.
- After rotations, **20** is now repositioned and becomes the parent of **8** and **15**.
- Heap structure is now balanced, and the max-heap property is maintained.



Up-Heapify can be used in the following instances:

- Inserting a new task into a priority queue
- Initial setup of a scheduling system or simulation event queue
- Maintaining top-K highest or lowest values from a stream

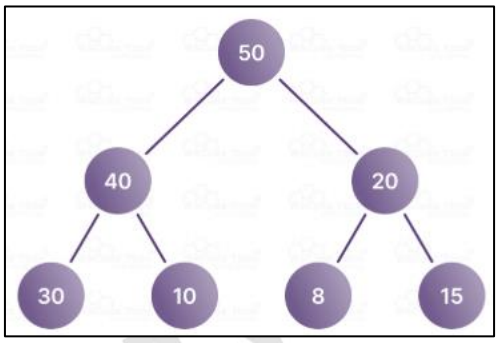
- **Down-Heapify (Bubble-Down):** Used after deleting the root element to restore the heap property by moving the new root element down the tree.

  Steps:

  1. Replace the root element with the last element in the heap.
  2. Remove the last element from the heap.
  3. Compare the new root with its children.
  4. If the heap property is violated (the new root is smaller than a child in a max-heap or larger than a child in a min-heap), swap the new root with the larger (in max-heap) or smaller (in min-heap) child.
  5. Repeat this process until the heap property is restored or the element reaches a leaf node.

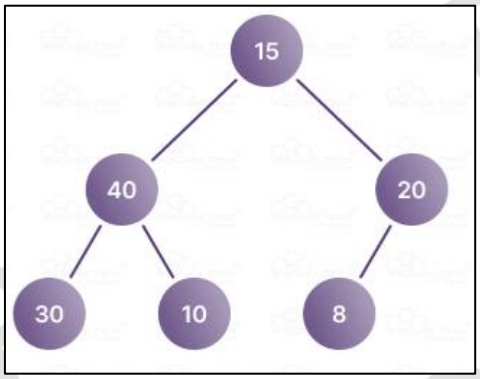
Example (Max-Heap)

Initial Heap:



Extract-Max (50):

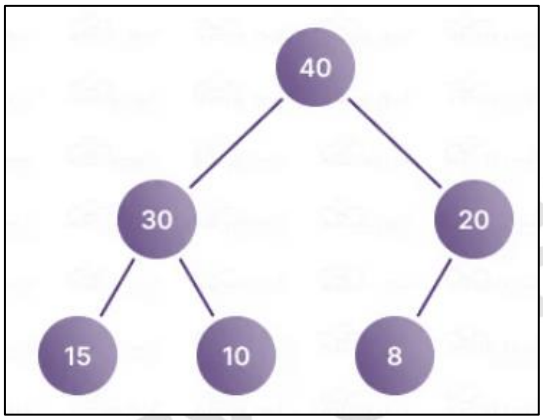
- Remove the root node **50** from the max-heap.
- Replace it with the last element, which is **15**.
- Now, **15** becomes the new root of the heap.



- The temporary tree becomes the illustration above.

Apply Down-Heapify:

- First comparison:
  - Compare **15** with its children: left = **40**, right = **20**.
  - Since **15 < 40** and **40 > 20**, swap **15** with **40** (larger child).
  - Now, **15** is in the left child position of **50**'s former left subtree.
- Second comparison:
  - Compare **15** with its new children: left = **30**, right = **10**.
  - Since **15 < 30** and **30 > 10**, swap **15** with **30** (larger child).
  - Now, **15** is a leaf node and has no children.
  - Heap is now restored to a valid max-heap structure.



Down-Heapify can be used in the following instances:

- Removing the highest-priority task in a job scheduler
- Sorting large datasets
- External merge sort or multi-file log merging

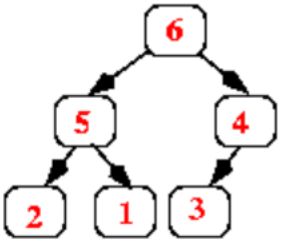
Priority Queue Implementations

A **priority queue** is an abstract data type that operates like a regular queue, except that each element has a priority level, and elements are dequeued based on their priority, not just their arrival order.

Unlike regular queues that process elements in a first-in, first-out (FIFO) manner, priority queues ensure that elements with higher priority are processed first. For example, when handling tasks sent to the Computer Science Department's printer, tasks sent by the department chair should be printed first, followed by tasks sent by professors, then those sent by graduate students, and finally those sent by undergraduates.

Implementing Priority Queues using Heaps

The following shows both the conceptual heap (the binary tree) and its array representation:



[0]	[1]	[2]	[3]	[4]	[5]	[6]
	6	5	4	2	1	3

The standard approach is to use an array (or an ArrayList), starting at position 1 (instead of 0), where each item in the array corresponds to one node in the heap:

- The root of the heap is always in array[1]
- Its left child is in array[2]
- Its right child is in array[3]

In general, if a node is in array[k], then its left child is in array[k*2], and its right child is in array[k*2 + 1].
If a node is in array[k], then its parent is in array[k/2] (using integer division, so that if k is odd, then the result is truncated; e.g., 3/2 = 1).

Implementing Insert

When a new value is inserted into a priority queue, add the value so that the heap still has its properties.

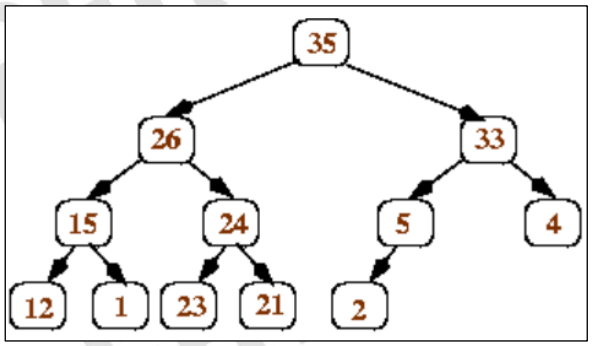
To achieve:

1. Add the new value at the **end** of the array; that corresponds to adding it as a new rightmost leaf in the tree (or, if the tree was a **complete** binary tree, i.e., all leaves were at the **same** depth d, then that corresponds to adding a new leaf at depth d+1).
2. Step 1 above ensures that the heap still has the **shape** property; however, it may not have the **order** property. Compare the new value to the value in its parent to check.

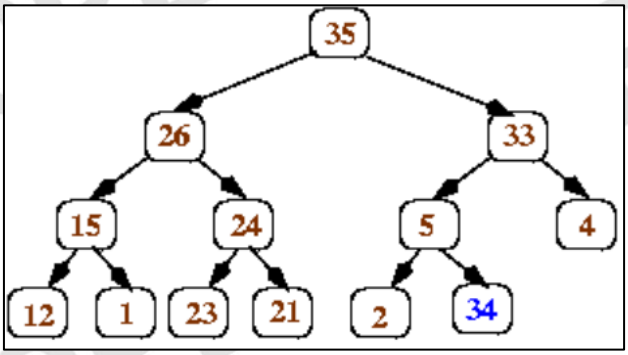
   If the parent is smaller, swap the values. Continue this check-and-swap procedure up the tree until the order property holds, or until it reaches the root.

For example, inserting a value of 34 into a heap:

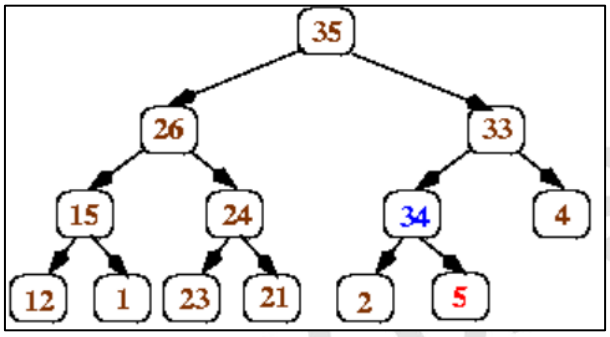
This is the initial heap:



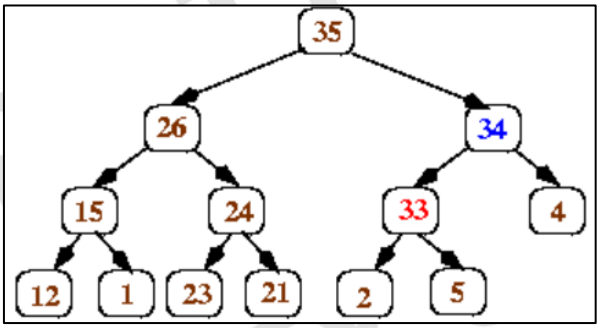
34 is then added to the rightmost leaf: (Step 1)



Swap 34 with the current parent (5): (Step 2)



Lastly, swap with the new parent (33): (Step 2). This is the last movement since its newer parent (35) is higher.



Implementing removeMax

As heaps have an order property, the largest value is always at the root. Therefore, the **removeMax** operation will always remove and return the root value.

This part shows how to replace the root node so that the heap still has the order and shape properties.

To achieve:

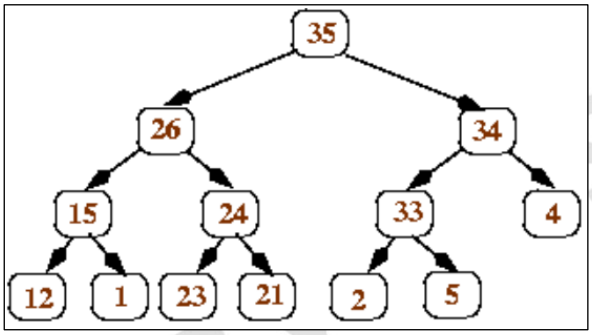
1. Replace the value in the root with the value at the end of the array (which corresponds to the heap's rightmost leaf at depth d).

   Remove that leaf from the tree.

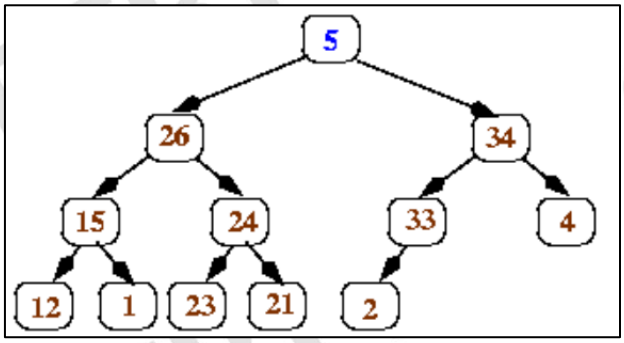
2. From down the tree, swap the values to restore the order property: each time, if the value in the current node is less than one of its children, then swap its value with the larger child. This ensures that the new root value is larger than both of its children.

For example:

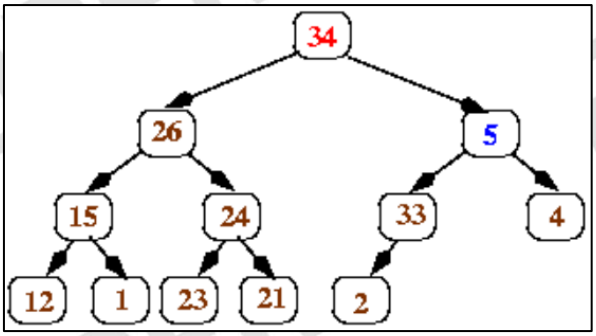
This is the initial heap:



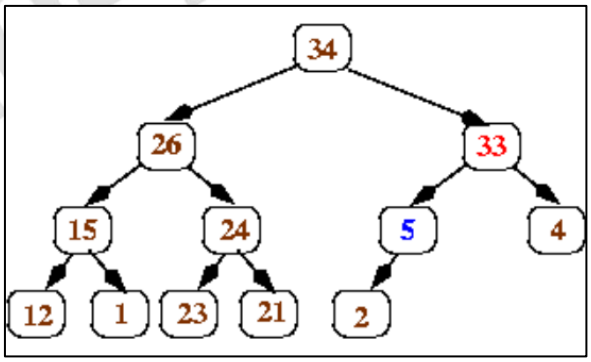
Extracting the root value (35) and replacing it with the last leaf (35): (Step 1)



Swapping the current root (5) with the larger child (35): (Step 2)



Lastly, swap value 5 with its children: (Step 2)



References:

Oracle. (n.d.a.). *Heap* operations. Retrieved on August 6, 2025, from https://docs.oracle.com/cd/E19205-01/819-3703/14_8.htm

University of Wisconsin-Madison. (n.d.a.). *Priority queues.* Retrieved on August 7, 2025, from https://pages.cs.wisc.edu/~vernon/cs367/notes/11.PRIORITY-Q.html

WSCube Tech. (2025). *What is heap data structure? Types, examples, operations, full guide*. Retrieved on August 6, 2025, from https://www.wscubetech.com/resources/dsa/heap-data-structure