

Ahmad Albarasy · Follow

Last edited by Ahmad Albarasy on Jan 15, 2026

DSaA Lab 8 - Heaps and Priority Queues

The Islamic University of Gaza
Engineering Faculty
Department of Computer Engineering

Prepared by: Eng. Ahmad Albarasy

Under the supervision of: Prof. Ahmad Mahdi

In this lab, we are going to explore heaps, one of the most important applications of tree-based data structures which provides an efficient way to organize data based on priority, making them a fundamental component in many real-world systems and algorithms. We are also going to study priority queues, which rely on heaps to support fast insertion and removal of the highest (or lowest) priority elements.

The Priority Queue

Definition

A priority queue is **an abstract data type (ADT) in which each element is associated with a priority**, and elements are accessed or removed based on their priority rather than their insertion order.

Unlike a regular queue (FIFO), where the first inserted element is the first removed, a priority queue always removes the element with the highest priority (or the lowest priority, depending on the convention).

```
1  /**
2   * An interface for a priority queue, where each element has an associated key
3   * that determines its priority.
4   */
5  public interface PriorityQueue<K, V> {
6
7      /**
8       * Returns the number of entries in the priority queue.
9       */
10     int size();
11
12     /**
13      * Tests whether the priority queue is empty.
14      */
15     boolean isEmpty();
16
17     /**
18      * Inserts a key-value pair into the priority queue
19      * and returns the entry created.
20      */
21     Entry<K, V> insert(K key, V value) throws IllegalArgumentException;
22
23     /**
24      * Returns (but does not remove) an entry with minimal key.
25      */
26     Entry<K, V> min();
27
28     /**
29      * Removes and returns an entry with minimal key.
30      */
31     Entry<K, V> removeMin();
32 }
33
```

Implementation

A priority queue is an abstract data type and can be implemented using several underlying data structures. Each implementation offers different performance characteristics for insertion, access, and removal operations.

1. Unsorted List or Array

In this approach, elements are stored in an unsorted collection.

- **Insertion:** $O(1)$, since the element can be appended at the end.
- **Find-Min / Remove-Min:** $O(n)$, since the entire list must be scanned to find the minimum.

This implementation is simple but inefficient for frequent removals.

2. Sorted List or Array

Here, elements are maintained in sorted order.

- **Insertion:** $O(n)$, since elements must be shifted to maintain order.
- **Find-Min:** $O(1)$, if the smallest element is kept at one end.
- **Remove-Min:** $O(1)$ or $O(n)$, depending on whether shifting is required.

This approach is efficient when insertions are rare.

3. Binary Search Tree

A **balanced binary search tree** (such as a red-black tree or AVL tree) can be used to implement a priority queue.

- **Insertion:** $O(\log n)$
- **Find-Min:** $O(\log n)$
- **Remove-Min:** $O(\log n)$

This provides good performance but requires a more complex structure.

4. Binary Heap (Most Common)

The most common and efficient implementation of a priority queue is the heap implementation.

- **Insertion:** $O(\log n)$
- **Find-Min:** $O(1)$
- **Remove-Min:** $O(\log n)$

The heap offers an excellent balance between simplicity and efficiency, which is why it is widely used in practice.

Real-World Use Cases

Priority queues are used in many real-world systems where elements must be processed according to importance, urgency, or cost rather than arrival time.

1. Task Scheduling in Operating Systems

Operating systems use priority queues to decide which process should run next. Processes with higher priority (such as system-critical tasks) are scheduled before lower-priority background tasks.

2. Network Packet Routing

Routers and switches use priority queues to process network packets. Time-sensitive data such as video calls or voice traffic is given higher priority than less urgent data like file downloads.

3. Graph Algorithms

Algorithms such as Dijkstra's shortest path and Prim's minimum spanning tree rely on priority queues to efficiently select the next node with the smallest distance or cost.

4. Emergency and Service Systems

Hospitals and emergency response systems use priority queues to treat patients or incidents based on severity rather than arrival time.

5. Artificial Intelligence and Search Algorithms

AI search strategies such as A* use priority queues to explore the most promising states first based on heuristic values.

The Heap

Definition

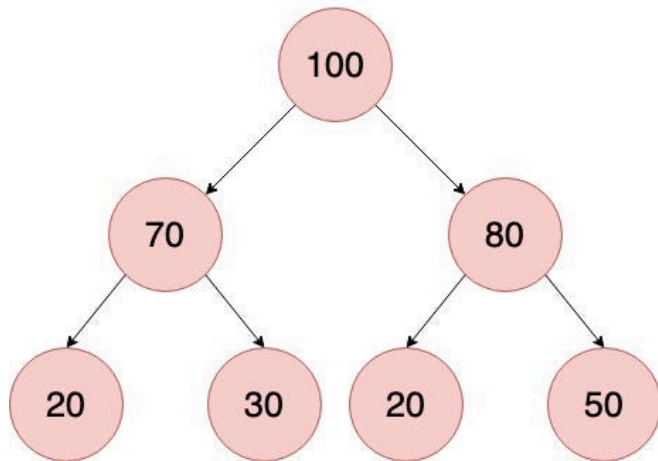
A heap is a **specialized complete binary tree** that ensures that the maximum (in a max-heap) or minimum (in a min-heap) element is always stored at the root of the tree, which allows efficient access to the highest- or lowest-priority element.

The way heaps maintain such a property is achieved as follows:

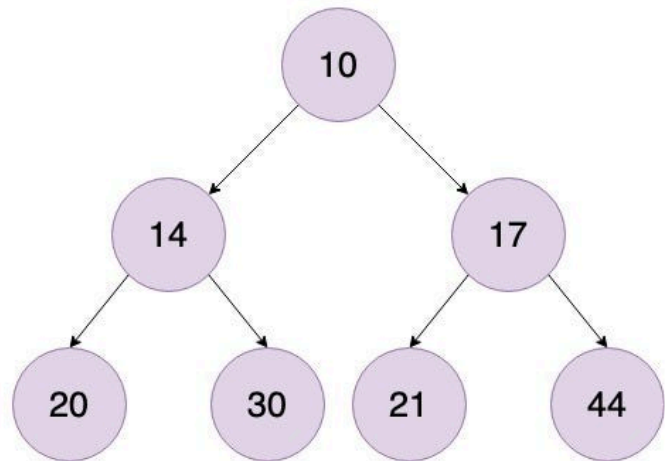
1. Every node satisfies the heap property:

- In a max-heap, the value of each node is greater than or equal to the values of its children
- In a min-heap, the value of each node is less than or equal to the values of its children.

Max Heap



Min Heap



2. **The heap is always kept complete**, meaning that all levels are fully filled except possibly the last, which is filled from left to right.

Fundamental and Auxiliary Operations

Heaps support a small set of fundamental operations that define their main functionality and a set of auxiliary operations that are used internally to maintain the heap structure and its properties.

Fundamental Operations

- **Insertion:** Adds a new element x to the heap while preserving the heap structure and heap property.
- **Find-Min / Find-Max:** Returns the minimum element in a min-heap or the maximum element in a max-heap without removing it.
- **Remove-Min / Remove-Max:** Removes and returns the root element (minimum or maximum), then restores the heap structure.

Auxiliary Operations

- **Upheap (Bubble-Up / Percolate-Up):** Moves a newly inserted element upward until the heap property is restored.
- **Downheap (Bubble-Down / Percolate-Down):** Moves the root element downward after removal until the heap property is restored.
- **Swap:** Exchanges two elements in the underlying array representation.

The efficiency of these operations is closely related to the height of the heap. Since a heap is a complete binary tree, its height is $O(\log n)$, where n is the number of elements. As a result, insertion and removal operations run in $O(\log n)$ time due to the

upheap and downheap processes, while accessing the minimum or maximum element takes $O(1)$ time because it is always stored at the root.

Implementation


```
1  /**
2   * An implementation of a priority queue using an array-based heap.
3   */
4  public class HeapPriorityQueue<K, V> extends AbstractPriorityQueue<K, V> {
5
6      /** Primary collection of priority queue entries */
7      protected ArrayList<Entry<K, V>> heap = new ArrayList<>();
8
9      /** Creates an empty priority queue
10       * based on the natural ordering of its keys. */
11     public HeapPriorityQueue() {
12         super();
13     }
14
15     /** Creates an empty priority queue
16      * using the given comparator to order keys. */
17     public HeapPriorityQueue(Comparator<K> comp) {
18         super(comp);
19     }
20
21     // protected utilities
22
23     protected int parent(int j) {
24         return (j - 1) / 2;
25     }
26
27     protected int left(int j) {
28         return 2 * j + 1;
29     }
30
31     protected int right(int j) {
32         return 2 * j + 2;
33     }
34
35     protected boolean hasLeft(int j) {
36         return left(j) < heap.size();
37     }
38
39     protected boolean hasRight(int j) {
40         return right(j) < heap.size();
41     }
42
43     /** Exchanges the entries at indices i and j of the array list. */
44     protected void swap(int i, int j) {
45         Entry<K, V> temp = heap.get(i);
46         heap.set(i, heap.get(j));
47         heap.set(j, temp);
48     }
49
50     /** Moves the entry at index j higher,
51      * if necessary, to restore the heap property. */
52     protected void upheap(int j) {
53         while (j > 0) {
```

```

54     int p = parent(j);
55     if (compare(heap.get(j), heap.get(p)) >= 0) break;
56     swap(j, p);
57     j = p;
58 }
59 }
60
61 /** Moves the entry at index j lower,
62  * if necessary, to restore the heap property. */
63 protected void downheap(int j) {
64     while (hasLeft(j)) {
65         int leftIndex = left(j);
66         int smallChildIndex = leftIndex;
67
68         if (hasRight(j)) {
69             int rightIndex = right(j);
70             if (compare(heap.get(leftIndex), heap.get(rightIndex)) > 0) {
71                 smallChildIndex = rightIndex;
72             }
73         }
74
75         if (compare(heap.get(smallChildIndex), heap.get(j)) >= 0) break;
76
77         swap(j, smallChildIndex);
78         j = smallChildIndex;
79     }
80 }
81
82 // public methods
83
84 /** Returns the number of items in the priority queue. */
85 public int size() {
86     return heap.size();
87 }
88
89 /** Returns (but does not remove) an entry with minimal key (if any). */
90 public Entry<K, V> min() {
91     if (heap.isEmpty()) return null;
92     return heap.get(0);
93 }
94
95 /** Inserts a key-value pair and returns the entry created. */
96 public Entry<K, V> insert(K key, V value)
97     throws IllegalArgumentException {
98     checkKey(key);
99     Entry<K, V> newest = new PQEntry<>(key, value);
100    heap.add(newest);
101    upheap(heap.size() - 1);
102    return newest;
103 }
104
105 /** Removes and returns an entry with minimal key (if any). */
106 public Entry<K, V> removeMin() {

```

```
107         if (heap.isEmpty()) return null;
108         Entry<K, V> answer = heap.get(0);
109         swap(0, heap.size() - 1);
110         heap.remove(heap.size() - 1);
111         downheap(0);
112         return answer;
113     }
114 }
115
```

The Heap Sort

Definition

Heap sort is a **comparison-based sorting algorithm** that uses a heap to efficiently sort an array. It works by first building a max-heap from the array, then repeatedly removing the maximum element from the heap and placing it at the end of the array. Building a max-heap from an existing linear sequence (typically an array) happens in-place without using an extra container, making the process memory-efficient.

Algorithm Steps

1. **Build a max-heap** from the input array.
2. **Swap the root (maximum element) with the last element** of the heap.
3. **Reduce the heap size** by one and restore the heap property using downheap (heapify) on the new root.
4. Repeat steps 2–3 until the heap is empty.

```
public class HeapSort {

    // Main function to perform heap sort
    public static void heapSort(int[] arr) {
        int n = arr.length;

        // Build max heap
        for (int i = n / 2 - 1; i >= 0; i--) {
            heapify(arr, n, i);
        }

        // Extract elements from heap one by one
        for (int i = n - 1; i >= 0; i--) {
            int temp = arr[0];
            arr[0] = arr[i];
            arr[i] = temp;

            // Heapify reduced heap
            heapify(arr, i, 0);
        }
    }
}
```

```
    }  
}  
  
// Heapify a subtree rooted at index i  
private static void heapify(int[] arr, int n, int i) {  
    int largest = i;  
    int left = 2 * i + 1;  
    int right = 2 * i + 2;  
  
    if (left < n && arr[left] > arr[largest])  
        largest = left;  
  
    if (right < n && arr[right] > arr[largest])  
        largest = right;  
  
    if (largest != i) {  
        int swap = arr[i];  
        arr[i] = arr[largest];  
        arr[largest] = swap;  
  
        // Recursively heapify affected sub-tree  
        heapify(arr, n, largest);  
    }  
}
```

Ahmad Albarasy /  Data Structures and Alg... Edit

Complexity

- **Time Complexity:** $O(n \log n)$ for all cases.
- **Space Complexity:** $O(1)$ extra space for array-based implementation (in-place sort).

Tasks

Task 1 (4 points)

Given an integer array `nums` and an integer `k`, return the `k`th largest element in the array.

Note that it is the `k`th largest element in the sorted order, not the `k`th distinct element.

Question link on [LeetCode](#)

Example 1:

Input: `nums = [3,2,1,5,6,4]`, `k = 2`

Output: `5`

Example 2:

Input: `nums = [3,2,3,1,2,4,5,5,6]` , `k = 4`

Output: `4`

Constraints:

- `1 <= k <= nums.length <= 105`
- `-104 <= nums[i] <= 104`

Task 2 (6 points)

Given an integer array `nums` and an integer `k` , return the `k` most frequent elements. You may return the answer in any order.

Question link on [LeetCode](#)

Example 1:

Input: `nums = [1,1,1,2,2,3]` , `k = 2`

Output: `[1,2]`

Example 2:

Input: `nums = [1]` , `k = 1`

Output: `[1]`

Example 3:

Input: `nums = [1,2,1,2,1,2,3,1,3,2]` , `k = 2`

Output: `[1,2]`

Constraints:

- `1 <= nums.length <= 105`
- `-104 <= nums[i] <= 104`
- `k` is in the range `[1, the number of unique elements in the array]`
- It is guaranteed that the answer is unique

Tasks Submission Guide

You will solve both problems on LeetCode and you have to make sure that your solution passes all test cases and that LeetCode accepts it. Once you are done, its time to submit your solution on GradeScope.

Firstly, make sure to submit only two files:

1. TaskOne.java

2. TaskTwo.java

Secondly, for each question, you have to follow these steps to make sure GradeScope's autograder runs without issues:

For Task 1:

1. Copy your whole solution into a file named `TaskOne.java`
2. Change the class name from `Solution` to `TaskOne` , and make sure that the class is `public`
3. add the following package identifier to the beginning of the file. Make sure to copy it and **don't write it manually to avoid typos**:

```
1 | package com.gradescope.labEight;
```

Here's an example solution for reference:

```
1 | package com.gradescope.labEight;
2 |
3 | public class TaskOne {
4 |     public int findKthLargest(int[] nums, int k) {
5 |
6 |     }
7 |     /* if any helper methods are needed, make sure to include them */
8 | }
```

For Task 2:

1. Copy your whole solution into a file named `TaskTwo.java`
2. Change the class name from `Solution` to `TaskTwo` , and make sure that the class is `public`
3. add the following package identifier to the beginning of the file. Make sure to copy it and **don't write it manually to avoid typos**:

```
1 | package com.gradescope.labEight;
```

Here's an example solution for reference:

```
1 | package com.gradescope.labEight;
2 |
3 | public class TaskTwo {
4 |     public int[] topKFrequent(int[] nums, int k) {
5 |
6 |     }
7 |     /* if any helper methods are needed, make sure to include them */
8 | }
```

Happy Coding 😊

Last changed by

Ahmad Albarasy · Follow

0 16

Add a comment

Read more

Data Structures and Algorithms Lab 7 - Hashing and Maps

Hashing is a process of converting data
Jan 3, 2026

Data Structures and Algorithms Lab 6 - Trees

In this lab, we are going to explore trees, one of the most important and interesting data structures that ar...
Dec 26, 2025

Data Structures and Algorithms Lab 5 - Stacks and Queues

In this lab, we are going about stacks and queues, which are
Dec 13, 2025

Data Structures and Algorithms Lab 4 - Recursion

In this lab, we are going to learn about recursion and how can we use it to solve interesting problems. We ar...
Dec 6, 2025

[Read more from Ahmad Albarasy](#)

Published on  **HackMD**