# Data Structure Algorithms & Applications (CT-159)
# Lab 11

Heap and Priority Queue

## Objectives

The objective of the lab is to get students familiar with Heap and Priority Queue Data Structures.

## Tools Required

Dev C++ IDE

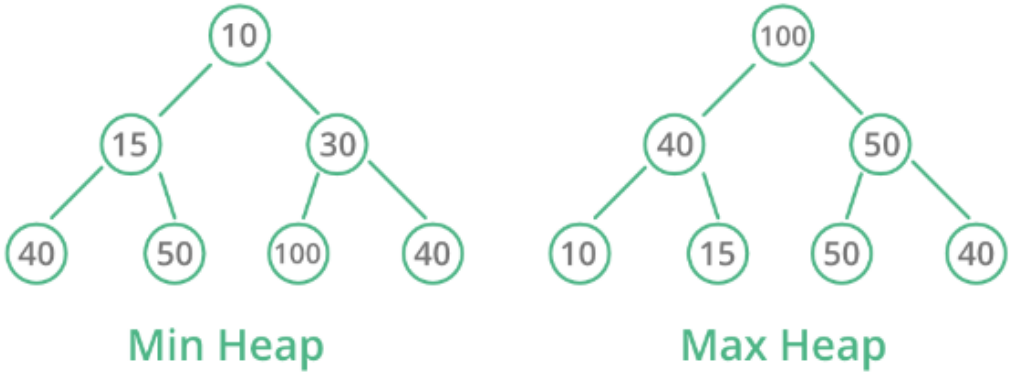
Course Coordinator –
Course Instructor –
Lab Instructor –
Prepared By Department of Computer Science and Information Technology
NED University of Engineering and Technology

## Heap

A **heap** is a binary tree-based **data structure** that follows the **heap property**. In a heap, the value of each node is compared to the values of its children in a specific way:

- **Max-Heap**: The value of each node is greater than or equal to the values of its children, ensuring that the root node contains the maximum value. As you move down the tree, the values decrease.
- **Min-Heap**: The value of each node is less than or equal to the values of its children, ensuring that the root node contains the minimum value. As you move down the tree, the values increase.

This property guarantees that the largest (in a max-heap) or smallest (in a min-heap) value is always at the root, and the tree maintains a specific order based on the type of heap. Heaps are usually used to implement priority queues, efficient sorting (heapsort), and finding the k-th largest or smallest element.




**Heap Implementation:**

A heap can be implemented using a dynamic array. The root of the heap is at index 0. For each element at index i, its:

- Parent is at index (i-1)/2
- Left child is at index 2*i + 1
- Right child is at index 2*i + 2

**Heapify:**

It is the process to rearrange the elements to maintain the property of heap data structure. It is done when a certain node creates an imbalance in the heap due to some operations like insertion and deletion. It takes **O(log N)** to balance the tree. **Heapify up** is used after inserting a new element at the end of the heap to restore the heap property by moving the element up to its correct position. **Heapify down** is used after removing the root (max/min element) from the heap. The last element is moved to the root, and heapify down restores the heap property by moving the element down to its correct position.

**Insertion:** If we insert a new element into the heap since we are adding a new element into the heap so it will distort the properties of the heap so we need to perform the **heapify-up** operation so that it maintains the property of the heap.

**Heapify Up Operation:**

Insert the new element at the last position in the heap (end of the array).
Set i as the index of the newly inserted element.
While i > 0 and heap[parent(i)] < heap[i] (for max-heap):

- Swap heap[i] with heap[parent(i)].
- Set i to parent(i).

Stop when the heap property is restored

```
void heapifyUp(int i) {
  while (i > 0 && heap[(i - 1) / 2] < heap[i]) {
    swap(heap[i], heap[(i - 1) / 2]);
    i = (i - 1) / 2; // Move to the parent
  }
}
```

**Deletion:** If we delete the element from the heap it always deletes the root element of the tree and replaces it with the last element of the tree. Since we delete the root element from the heap it will distort the properties of the heap so we need to perform heapify down operations so that it maintains the property of the heap.

**Heapify Down Operation:**
Replace heap[0] (root) with the last element and remove the last element.
Set i as the index of the new root.
While i has a left child (2 * i + 1 < heap_size):

- Let largest be the index of the left child.
- If i has a right child (2 * i + 2 < heap_size) and the right child is greater than the left child, set largest to the right child.
- If heap[i] is smaller than heap[largest], swap them.
- Set i to largest.

Stop when the heap property is restored

```
void heapifyDown(int i) {
   int size = heap.size();
   while (2 * i + 1 < size) { // While there's a left child
      int largest = 2 * i + 1; // Assume left child is largest
      int right = 2 * i + 2;   // Right child index
      if (right < size && heap[right] > heap[largest]) {
         largest = right; // Right child is larger
      }
      if (heap[i] >= heap[largest]) {
         break; // Heap property is satisfied
      }
      swap(heap[i], heap[largest]); // Swap with larger child
      i = largest; // Move to the larger child
   }
}
```

Example 01: Implementation of a Max Heap Class

```
#include <iostream>
#include <vector>
using namespace std;
class MaxHeap {
private:
   vector<int> heap;

   // Helper method to heapify up (for insertion)
   void heapifyUp(int index) {
      while (index > 0 && heap[parent(index)] < heap[index]) {
         swap(heap[parent(index)], heap[index]);
         index = parent(index);
      }
   }
   // Helper method to heapify down (for deletion)
   void heapifyDown(int index) {
      int size = heap.size();
      while (leftChild(index) < size) { // while there is a left child
         int largest = index;
         int left = leftChild(index);
         int right = rightChild(index);

         if (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;  // heap property satisfied
         }
      }
   }
```

```
    // Helper functions to get parent and children indices
    int parent(int i) { return (i - 1) / 2; }
    int leftChild(int i) { return 2 * i + 1; }
    int rightChild(int i) { return 2 * i + 2; }
public:
    // Insert a new value into the heap
    void insert(int value) {
        heap.push_back(value); // Add value at the end
        heapifyUp(heap.size() - 1); // Restore heap property
    }
    // Remove the maximum element (the root)
    void extractMax() {
        if (heap.empty()) {
            throw out_of_range("Heap is empty");
        }
        int maxValue = heap[0];
        heap[0] = heap.back();  // Move last element to root
        heap.pop_back();  // Remove the last element
        heapifyDown(0);  // Restore heap property
    }
    // Get the maximum element without removing it
    int getMax() const {
        if (heap.empty()) {
            throw out_of_range("Heap is empty");
        }
        return heap[0];
    }
    // Check if the heap is empty
    bool isEmpty() const {
        return heap.empty();
    }
};

// Main function to demonstrate the MaxHeap class
int main() {
    MaxHeap heap;
    heap.insert(20);
    heap.insert(15);
    heap.insert(30);
    heap.insert(40);
    heap.insert(10);
    cout << "Max element: " << heap.getMax() << endl;
    cout << "Extracting max element: " << heap.extractMax() << endl;
    cout << "Max-element after extraction: ";
    cout << "Max element: " << heap.getMax() << endl;
}
```

## Heap Sort

**Heap Sort** is an efficient comparison-based sorting algorithm that uses the **heap data structure** to sort an array. The key idea is to first build a max-heap from the input array and then repeatedly extract the maximum element (root of the heap), placing it at the end of the array. The process is repeated until the array is sorted.

**Steps for Heap Sort:**

1. **Build a Max-Heap** from a given unsorted input array. Use from-bottom-to-top approach.
2. **Extract the maximum element** (the root) from the heap, and move it to the end of the array.
3. **Heapify the root** to maintain the heap property for the reduced heap (ignoring the last element which is now sorted).
4. Repeat the process until all elements are extracted and placed in their correct position.

### Priority Queue

A **priority queue** is a data structure that allows efficient retrieval of the "highest priority" element. The most common implementation of a priority queue is using a **heap** because heaps provide efficient insertion and deletion (retrieval of the highest or lowest element, depending on whether it's a max-heap or min-heap).

**Time Complexity:**

- **Insertion**: O(log n) — Heapify up takes logarithmic time in the worst case.
- **Deletion (pop)**: O(log n) — Heapify down takes logarithmic time.

- **Top (peek)**: O(1) — Simply accessing the root element.
- **Space Complexity**: O(n) — Space to store the elements in the heap.

## Priority Queue Library

In C++, the Standard Template Library (STL) provides a priority_queue container, which is essentially a max-heap by default. This means the largest element is always at the top. The priority_queue container is part of the <queue> header.

**Key Features of priority_queue:**

- **Automatic ordering**: It automatically orders elements such that the largest (by default) or smallest (if using a custom comparator) element is accessible at the top.
- **Custom ordering**: By providing a custom comparator, you can turn it into a min-heap or create custom priority rules.
- **Operations**:
  - Insertion (push)
  - Access to the top element (top)
  - Removal of the top element (pop)
  - Check size (size), empty state (empty)

**Example 02: Using Priority queue library as a max-heap.**

```
#include <iostream>
#include <queue>
using namespace std;
int main() {
   priority_queue<int> pq;
   // Insert elements
   pq.push(10);
   pq.push(30);
   pq.push(20);
   pq.push(5);
   // Display the top element (max element)
   cout << "Top element: " << pq.top() << endl; // 30
   // Remove elements (pops the largest element)
   pq.pop();
   cout << "Top element after pop: " << pq.top() << endl; // 20
   return 0;
}
```

**Example 02: Using Priority queue library as a min-heap**.

```
#include <iostream>
#include <queue>
#include <vector>
using namespace std;

// Custom comparator for a max-heap
//functor : a class that overloads function operator (). it invokes function whenever its instance/object is created
class CustomCompare {
   public:
    bool operator()(int a, int b) {
      return a < b;  // Max-heap: larger elements should have higher priority
   }
};
int main() {
   // Max-heap using a custom comparator and vector as the container
   priority_queue<int, vector<int>, CustomCompare> pq;
   pq.push(10);
   pq.push(30);
   pq.push(20);
   pq.push(5);
   while (!pq.empty()) {
      cout << pq.top() << " ";  // Output: 30 20 10 5
      pq.pop();
   }
   return 0;
}
```

# Exercise

1. Implement Heap Sort by using the steps listed in the Lab manual on Page#05.
2. Given an integer array nums and an integer k, return the k most frequent elements. You may return the answer in any order.
   Example 1: Input: nums = [1,1,1,2,2,3], k = 2, Output: [1,2]
   Example 2: Input: nums = [1], k = 1, Output: [1]
3. Given a string s, sort it in decreasing order based on the frequency of the characters. The frequency of a character is the number of times it appears in the string. Return the sorted string. If there are multiple answers, return any of them.
   Example 1: Input: s = "tree", Output: "eert"
   Example 2: Input: s = "cccaaa", Output: "aaaccc"
4. There are n workers. You are given two integer arrays quality and wage where quality[i] is the quality of the ith worker and wage[i] is the minimum wage expectation for the ith worker. We want to hire exactly k workers to form a paid group. To hire a group of k workers, we must pay them according to the following rules: Every worker in the paid group must be paid at least their minimum wage expectation. In the group, each worker's pay must be directly proportional to their quality. This means if a worker's quality is double that of another worker in the group, then they must be paid twice as much as the other worker. Given the integer k, return the least amount of money needed to form a paid group satisfying the above conditions. Answers within 10^-5 of the actual answer will be accepted.
   Example 1: Input: quality = [10,20,5], wage = [70,50,30], k = 2, Output: 105.00000
   Explanation: We pay 70 to 0th worker and 35 to 2nd worker.
   Example 2: Input: quality = [3,1,10,10,1], wage = [4,8,2,2,7], k = 3, Output: 30.66667
   Explanation: We pay 4 to 0th worker, 13.33333 to 2nd and 3rd workers separately.
5. The median is the middle value in an ordered integer list. If the size of the list is even, there is no middle value. So the median is the mean of the two middle values. For examples, if arr = [2,3,4], the median is 3. For examples, if arr = [1,2,3,4], the median is (2 + 3) / 2 = 2.5. You are given an integer array nums and an integer k. There is a sliding window of size k which is moving from the very left of the array to the very right. You can only see the k numbers in the window. Each time the sliding window moves right by one position. Return the median array for each window in the original array. Answers within 10^-5 of the actual value will be accepted.
   Example 1: Input: nums = [1,3,-1,-3,5,3,6,7], k = 3,
   Output: [1.00000,-1.00000,-1.00000,3.00000,5.00000,6.00000]

| **Lab 11 Evaluation** | | |
|---|---|---|
| **Student Name:** | | **Student ID:** **Date:** |
| **Rubric** | **Marks (25)** | **Remarks by teacher in accordance with the rubrics** |
| **R1** | | |
| **R2** | | |
| **R3** | | |
| **R4** | | |
| **R5** | | |