CL2001 Data Structures Lab [DS]	Lab 10 Heap and Types,Priority Queues, and Heaps as Priority Queues.

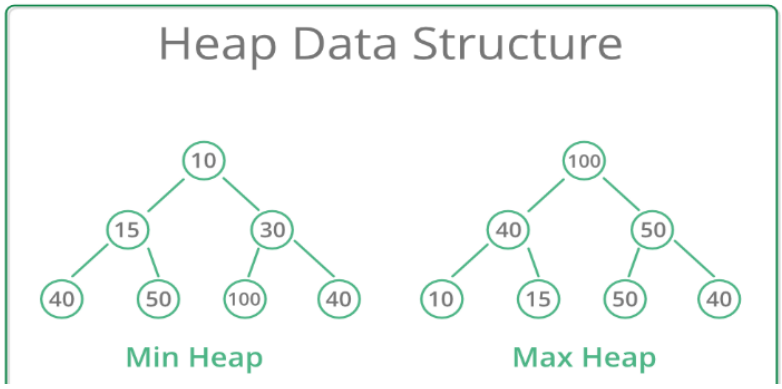
NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

Fall 2025

HEAP Data Structures

A Heap is a special Tree-based data structure in which the tree is a complete binary tree.
Operations of Heap Data Structure:

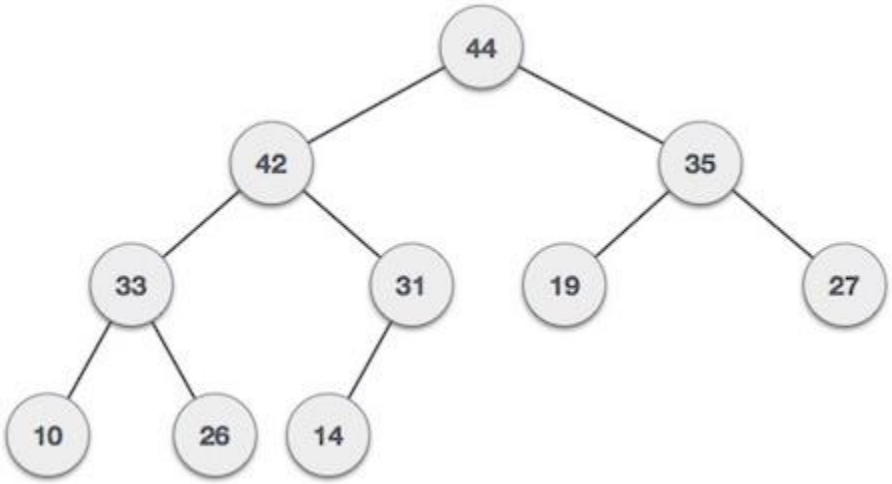
1. Heapify: a process of creating a heap from an array.
2. Insertion: process to insert an element in existing heap time complexity O(log N).
3. Deletion: deleting the top element of the heap or the highest priority element, and then organizing the heap and returning the element with time complexity O(log N).
4. Peek: to check or find the most prior element in the heap, (max or min element for max and min heap).



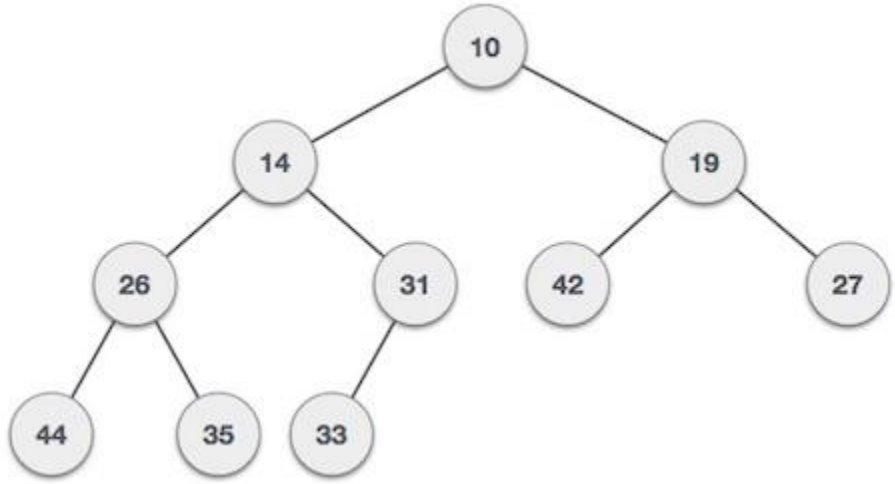
Types of Heap Data Structure

Generally, Heaps can be of two types:

Max-Heap: In a Max-Heap the key present at the root node must be greatest among the keys present at all of its children. The same property must be recursively true for all sub-trees in that Binary Tree.



Min-Heap: In a Min-Heap the key present at the root node must be minimum among the keys present at all of its children. The same property must be recursively true for all sub-trees in that Binary Tree.



How to construct a HEAP

We shall use the same example to demonstrate how a Max Heap is created. The procedure to create Min Heap is similar but we go for min values instead of max values.

We are going to derive an algorithm for max heap by inserting one element at a time. At any point of time, heap must maintain its property. While insertion, we also assume that we are inserting a node in an already heapified tree.

Step 1 − Create a new node at the end of heap.
Step 2 − Assign new value to the node.
Step 3 − Compare the value of this child node with its parent.
Step 4 − If value of parent is less than child, then swap them.
Step 5 − Repeat step 3 & 4 until Heap property holds.

Note − In Min Heap construction algorithm, we expect the value of the parent node to be less than that of the child node.

Let's understand Max Heap construction by an animated illustration. We consider the same input sample that we used earlier.

Heapify Algorithm

```
Heapify(array, size, i)
  set i as largest
  leftChild = 2i + 1
  rightChild = 2i + 2

  if leftChild > array[largest]
    set leftChildIndex as largest
  if rightChild > array[largest]
    set rightChildIndex as largest

  swap array[i] and array[largest]
```

Heapify:

- Start from the first index of non-leaf node whose index is
- given by n/2 – 1
- Set current element i as largest.
- The index of left child is given by 2i + 1 and the right child is
- given by 2i + 2.
- If leftChild is greater than currentElement (i.e. element
- at ith index), set leftChildIndex as largest.
- If rightChild is greater than element in largest,
- set rightChildIndex as largest.
- Swap largest with currentElement.

```
int parent(int i) { return (i - 1) / 2; }   // Parent index
int leftChild(int i) { return (2 * i) + 1; }   // Left child index
int rightChild(int i) { return (2 * i) + 2; } // Right child index
```

```
// Heapify the subtree rooted at index i
void minHeapify(int i) {
    int smallest = i;
    int left = leftChild(i);
    int right = rightChild(i);

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

    if (smallest != i) {
        swap(heap[i], heap[smallest]);
        minHeapify(smallest);
    }
}
```

Insertion In Min Heap:

```
Insertion Function (Pseudocode) -
void insertKey(int k)
{
    IF heap_size == capacity THEN ->
       print("Overflow : Heap full")
       return
    heap_size++
    i = heap_size - 1
    harr[i] = k
    WHILE ( i != 0 AND harr[parent(i)] > harr[i] )
       swap (harr[i] , harr[parent(i)])
       i = parent(i)  // return (i-1)/2
}
```

```
// Insert a new element into the heap
void insert(int val) {
    if (size == capacity) {
        cout << "Heap is full!" << endl;
        return;
    }

    // Insert the new element at the end of the heap
    heap[size] = val;
    size++;

    // Fix the min heap property if it is violated
    int i = size - 1;
    while (i != 0 && heap[parent(i)] > heap[i]) {
        swap(heap[i], heap[parent(i)]);
        i = parent(i);
    }
}
```

Remove

```
// Remove the minimum element (root) from the heap
void removeMin() {
    if (size <= 0) {
        cout << "Heap is empty!" << endl;
        return;
    }

    if (size == 1) {
        size--;
        return;
    }

    // Replace root with the last element
    heap[0] = heap[size - 1];
    size--;

    // Heapify the root
    minHeapify(0);
}
```

Max Heap

Insertion

The Max-Heap insertion operation places a new element at the end of the heap array and then "heapifies up," comparing the new element with its parent and swapping if necessary to maintain the max-heap property.

```
void insert(int val) {
        if (size == capacity) {
            cout << "Heap is full!" << endl;
            return;
        }

        heap[size] = val;
        size++;

        int i = size - 1;

        // Maintain Max-Heap property
        while (i != 0 && heap[parent(i)] < heap[i]) {
            swap(heap[i], heap[parent(i)]);
            i = parent(i);
        }
    }
```

The deletion operation removes the root (maximum element), then replaces it with the last element in the heap array. It then performs "heapify down," comparing the new root with its children and swapping with the larger child until the max-heap property is restored.

Heapify

```
// Heapify the subtree rooted at index i
void maxHeapify(int i) {
    int largest = i;
    int left = leftChild(i);
    int right = rightChild(i);

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

    if (largest != i) {
        swap(heap[i], heap[largest]);
        maxHeapify(largest);
    }
}

// Get the maximum element (root)
int getMax() {
    if (size > 0)
        return heap[0];
    cout << "Heap is empty!" << endl;
    return -1;
}
```

Priority Queue:

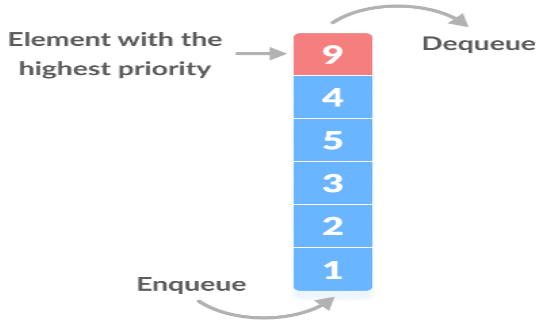
A priority queue is a special type of queue in which each element is associated with a priority value. And, elements are served on the basis of their priority. That is, higher priority elements are served first.

However, if elements with the same priority occur, they are served according to their order in the queue.

Assigning Priority Value

Generally, the value of the element itself is considered for assigning the priority. For example, The element with the highest value is considered the highest priority element. However, in other cases, we can assume the element with the lowest value as the highest priority element.

We can also set priorities according to our needs.




```
// Node structure for priority queue
struct Node {
    int value;      // The value of the element
    int priority;   // The priority of the element
    Node* next;     // Pointer to the next node in the queue

    Node(int val, int pri) : value(val), priority(pri), next(nullptr) {}
};

// PriorityQueue class
class PriorityQueue {
private:
    Node* front; // The front of the queue

public:
    // Constructor
    PriorityQueue() : front(nullptr) {}

    // Destructor to free the memory
    ~PriorityQueue() {
        while (front) {
            Node* temp = front;
            front = front->next;
            delete temp;
        }
    }
```

```
    // Insert a new element with a given priority
    void insert(int val, int pri) {
        Node* newNode = new Node(val, pri);
```

```
    // If the queue is empty or the new node has higher priority than the front
    if (!front || pri > front->priority) {
        newNode->next = front;
        front = newNode;
    } else {
        // Otherwise, find the correct position based on priority
        Node* current = front;
        while (current->next && current->next->priority >= pri) {
            current = current->next;
        }
        newNode->next = current->next;
        current->next = newNode;
    }
}
```

```
// Remove the element with the highest priority (front of the queue)
void remove() {
    if (!front) {
        cout << "Queue is empty!" << endl;
        return;
    }
    Node* temp = front;
    front = front->next;
    delete temp;
}
```

```
// Get the element with the highest priority (front of the queue)
int peek() {
    if (!front) {
        cout << "Queue is empty!" << endl;
        return -1; // Error value
    }
    return front->value;
}
```

```
// Print the entire queue (from front to back)
void printQueue() {
    if (!front) {
        cout << "Queue is empty!" << endl;
        return;
```

```
        }
        Node* current = front;
        while (current) {
            cout << "(" << current->value << ", " << current->priority << ") ";
            current = current->next;
        }
        cout << endl;
    }
};
```