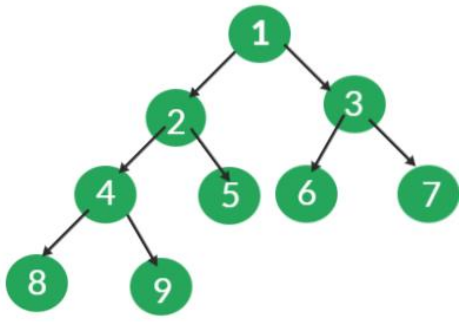
Data Structures Lab 10

Course: Data Structures (CL2001) **Semester:** Fall 2024
Instructor: Muhammad Nouman Hanif

Note:
- Lab manual covers following topics: **{Heaps, Min & Max Heap, Priority Queue, Heap Sort}**
- Maintain discipline during the lab.
- Just raise your hand if you have any problem.
- Completing all tasks of each lab is compulsory.
- Get your lab checked at the end of the session.

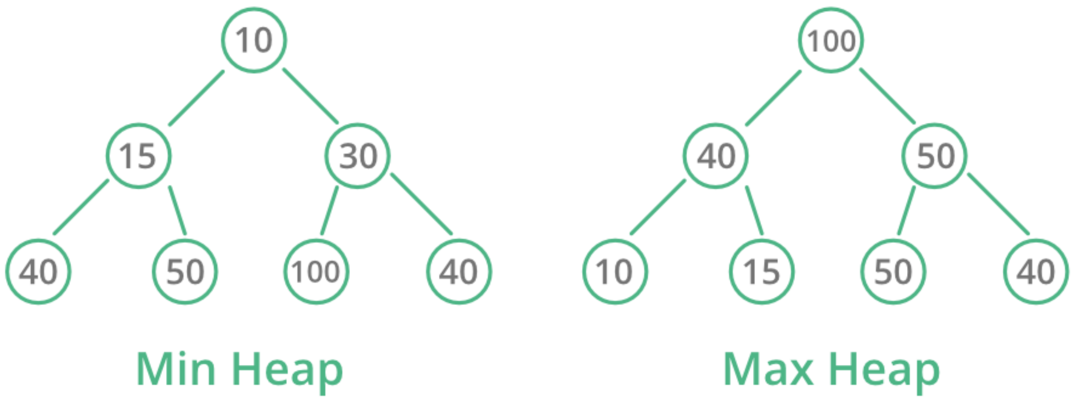
Heap Data Structure:

A Heap is a special Tree-based data structure in which the tree is a **complete binary tree** that satisfies the **heap property** (A complete binary tree is a binary tree in which every level, except possibly the last, is **completely filled**, and all nodes in the last level are **as far left as possible**.). It either follows **max heap** or **min heap** property.



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).

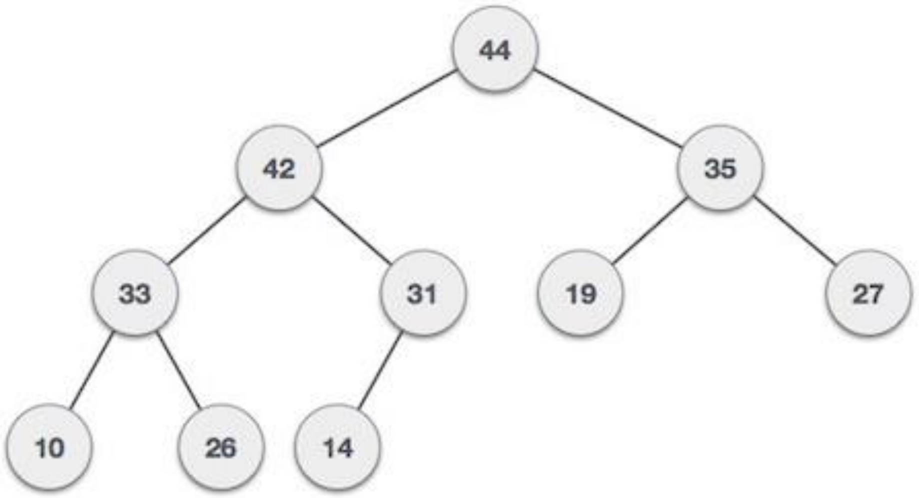


Types of Heap Data Structure:

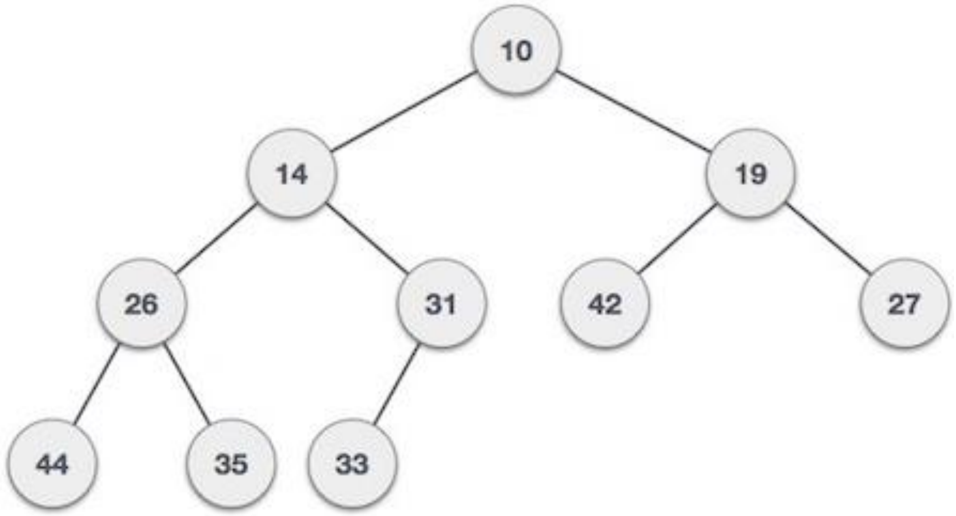
Generally, Heaps can be of **two types** (Practice)
Max Heap: 8 10 15 16 20 30 50 60
Min Heap: 125 100 70 60 50 45 10

For Input → 35 33 42 10 14 19 27 44 26 31

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.



<u>How to Construct Heap:</u>

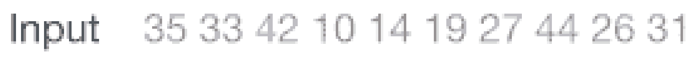
We shall use the same example to demonstrate how a Max Heap is created. The procedure to create Min Heap is similar but we go for min values instead of max values.

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

Steps:
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.

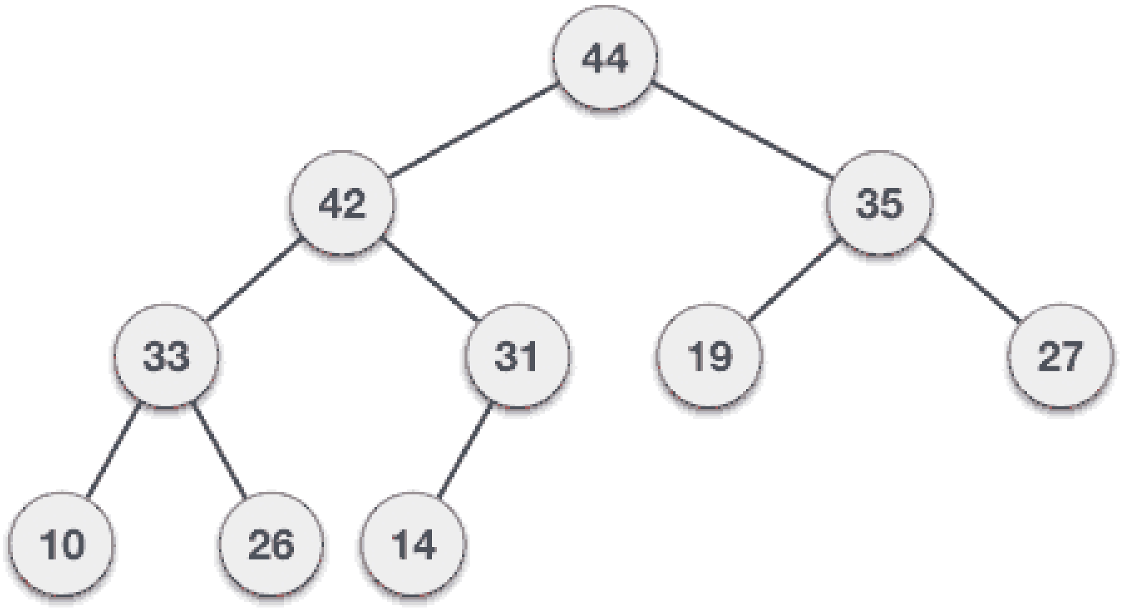
Let's understand Max Heap construction by an animated illustration. We consider the same input sample that we used earlier.


Input 35 33 42 10 14 19 27 44 26 31

Heap Deletion Operation:

Let us derive an algorithm to **delete from max heap**. Deletion in Max (or Min) Heap always happens at the **root to remove** the Maximum (or minimum) value.

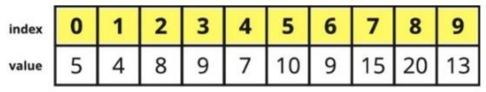
Steps:
Step 1 − Remove root node.
Step 2 − Move the last element of last level to root.
Step 3 − Compare the value of this child node with its parent.
Step 4 − If value of parent is less than child, then swap them.
Step 5 − Repeat step 3 & 4 until Heap property holds.



Algorithm:

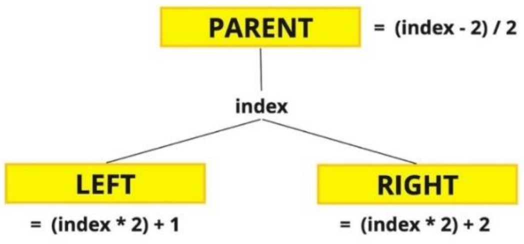
Heaps are commonly built on arrays. A heap represented as an array follows this simple rule.

- If a parent node is at index i, then its left child is at index 2i+1 and the right child is at index 2i+2.
- Similarly, for a given child node at index i, its parent node is at index (i-1)/2.
- This calculation is specific to 0-based arrays and is a common one among several variations.

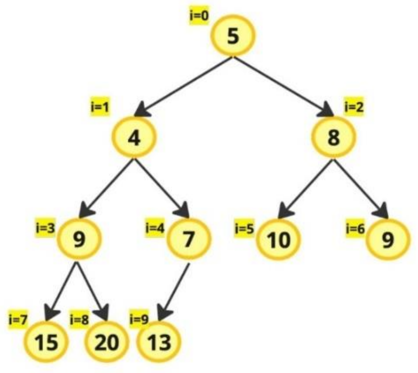


index	0	1	2	3	4	5	6	7	8	9
value	5	4	8	9	7	10	9	15	20	13

Pseudo Code:



Heap Index
Relationship Mapping



Left-Complete Tree

```
// Class for Max Heap
class MaxHeap:
    array: array of integers
    capacity: integer
    size: integer

// Constructor
    MaxHeap(cap):
        capacity = cap
        size = 0
        array = new array of integers with size cap

 // Function to swap two elements in the heap
    function swap(a, b):
        temp = a
        a = b
        b = temp

 // Function to heapify a subtree rooted at index i
    function heapify(i):
        largest = i
        left = 2 * i + 1
        right = 2 * i + 2
        // Check if the left child is larger than the root
        if left < size and array[left] > array[largest]:
            largest = left
        // Check if the right child is larger than the largest so far
        if right < size and array[right] > array[largest]:
            largest = right
        // If the largest is not the root, swap the root with the largest
        if largest != i:
            swap(array[i], array[largest])
            heapify(largest)
    // Function to insert a new element into the heap
    function insert(value):
        if size == capacity:
            print "Heap is full. Cannot insert more elements."
            return
        // Insert the new element at the end
        i = size
        size = size + 1
        array[i] = value
        // Fix the heap property if it is violated
        while i != 0 and array[(i - 1) / 2] < array[i]:
            swap(array[i], array[(i - 1) / 2])
            i = (i - 1) / 2

// Function to extract the maximum element from the heap
    function extractMax():
        if size == 0:
            print "Heap is empty. Cannot extract maximum element."
            return -1
        // Store the root element
        max = array[0]
        // Replace the root with the last element
        array[0] = array[size - 1]
        size = size - 1
        // Heapify the root
        heapify(0)
        return max

// Function to delete the maximum element from the heap (same as extractMax)
    function deleteMax():
        return extractMax()

// Function to print the elements of the heap
    function printHeap():
        print "Heap elements: "
        for i from 0 to size - 1:
            print array[i], " "

// Destructor to deallocate memory occupied by the heap
    destructor:
        delete array
```

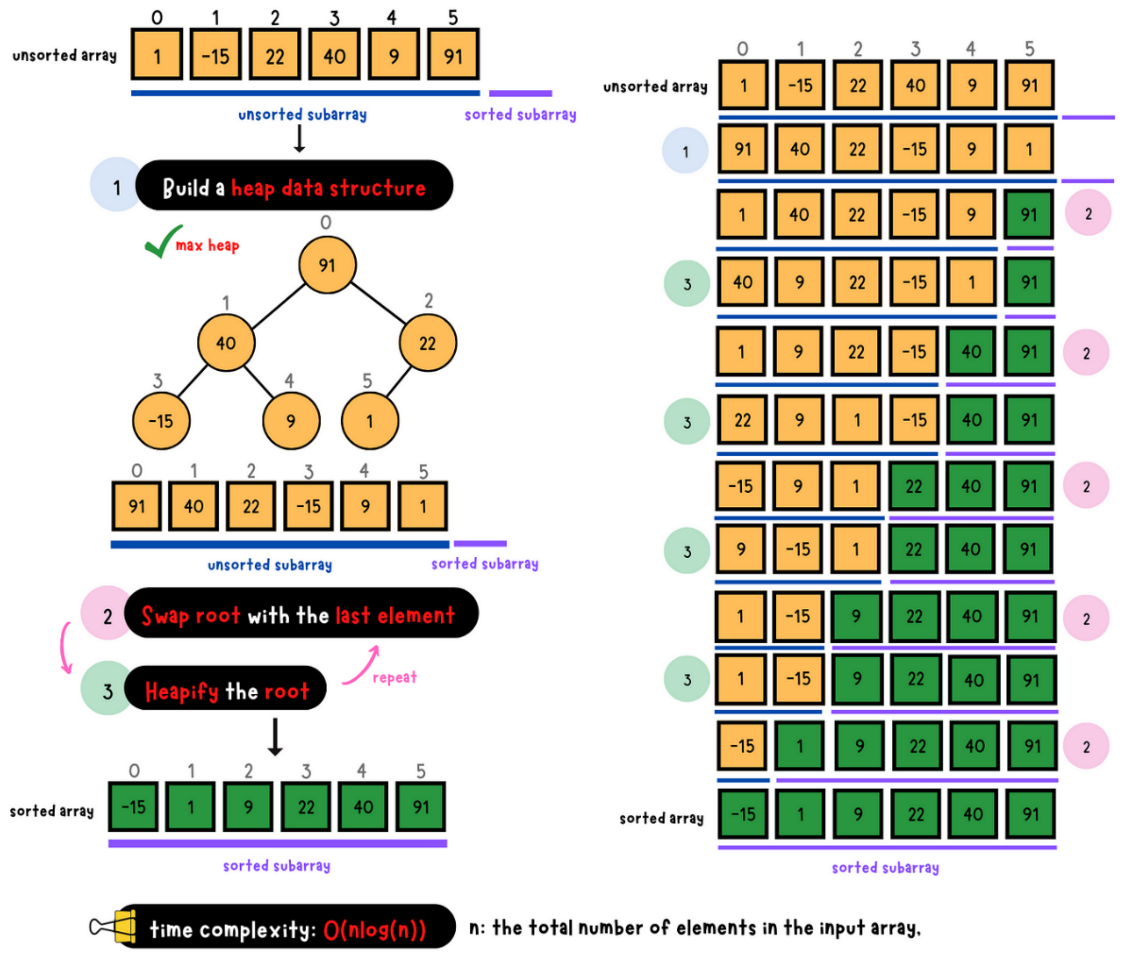
Heap Sort:

Heap sort is a comparison-based sorting technique based on **Binary Heap data structure**. It is similar to the **selection sort** where we first find **the minimum element and place the minimum element at the beginning**. Repeat the same process for the remaining elements. The time complexity is O(N log N)

Algorithm:

First convert the array into heap data structure using heapify, then one by one delete the root node of the Max-heap and replace it with the last node in the heap and then heapify the root of the heap. Repeat this process until size of heap is greater than 1.

```
HeapSort(arr)
BuildMaxHeap(arr)
for i = length(arr) to 2
        swap arr[1] with arr[i]
                heap_size[arr] = heap_size[arr] ? 1
                MaxHeapify(arr,1)
End
```



unsorted array
unsorted subarray
sorted subarray
1 Build a heap data structure
max heap
2 Swap root with the last element
3 Heapify the root
repeat
sorted array
time complexity: O(nlog(n))
n: the total number of elements in the input array.

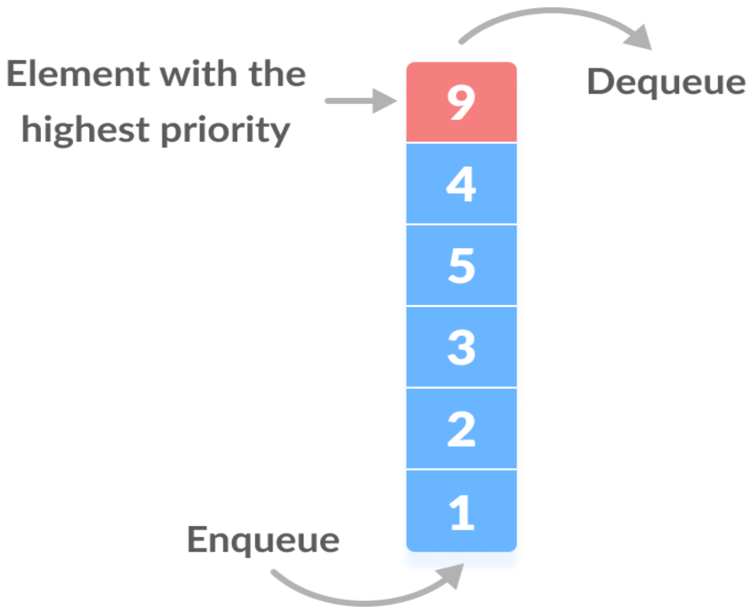
```
// Class for MaxHeapSort
class MaxHeapSort:
    array: array of integers
    size: integer
// Constructor to initialize the heap with an array
    MaxHeapSort(arr):
        array = arr
        size = length of arr
// Function to swap two elements in the heap
    function swap(a, b):
        temp = a
        a = b
        b = temp
// Function to heapify a subtree rooted at index i
    function heapify(i):
        largest = i
        left = 2 * i + 1
        right = 2 * i + 2
        // Check if the left child is larger than the root
        if left < size and array[left] > array[largest]:
            largest = left
        // Check if the right child is larger than the largest so far
        if right < size and array[right] > array[largest]:
            largest = right
        // If the largest is not the root, swap the root with the largest
        if largest != i:
            swap(array[i], array[largest])
            heapify(largest)
// Function to build the max heap from an unsorted array
    function buildMaxHeap():
        for i from (size / 2) - 1 to 0:
            heapify(i)
// Function to perform heap sort
    function heapSort():
        buildMaxHeap()
        for i from size - 1 down to 1:
            // Move the current root (largest element) to the end
            swap(array[0], array[i])
            size = size - 1  // Reduce the heap size
            // Heapify the root to maintain the max heap property
            heapify(0)
// Function to print the sorted array
    function printSortedArray():
        print "Sorted array: "
        for i from 0 to length of array - 1:
            print array[i], " "
```

Priority Queue:

A priority queue is a special type of queue in which each element is associated with a priority value having 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**.



Difference between Priority Queue and Normal Queue:
In a queue, the first-in-first-out rule is implemented whereas, in a priority queue, the values are removed based on priority. The element with the **highest priority is removed first**.

Algorithm:

```
// Constants
const MAX_SIZE = 100000

// Class for Priority Queue
class PriorityQueue:
   values: array of integers
   priorities: array of integers
   size: integer

   // Constructor
   PriorityQueue():
      size = -1

   // Function to insert a new element into the priority queue
   function enqueue(value, priority):
      size = size + 1
      values[size] = value
      priorities[size] = priority

   // Function to check the index of the element with the highest priority
   function peek():
      highestPriority = INT_MIN
      ind = -1

     // Find the element with the highest priority
      for i from 0 to size:
         if highestPriority == priorities[i] and ind > -1 and values[ind] < values[i]:
            highestPriority = priorities[i]
            ind = i
         else if highestPriority < priorities[i]:
            highestPriority = priorities[i]
            ind = i

      return ind

  // Function to remove the element with the highest priority
   function dequeue():
      int ind = peek()
      // Shift elements after the index to fill the gap
      for i from ind to size - 1:
         values[i] = values[i + 1]
         priorities[i] = priorities[i + 1]

      size = size – 1
```

Exercise

1. Consider a tree 7,1,6,2,5,9,10,2. You are tasked with finding both the min heap and max heap of this tree. Now, perform insertion and deletion operations on both min and max heap by inserting 99.

2. Consider another tree 35,33,42,10,14,19,27,44,26,31 Make the tree into a min heap and delete its root node and rebalance the tree to max heap and print the tree in a sorted output.

3. Given an array of size N. The task is to sort the array elements by completing functions heapify() and buildHeap() which are used to implement Heap Sort.
   Input:
   N = 5
   arr[] = {4,1,3,9,7}
   Output:
   1 3 4 7 9
   Explanation:
   After sorting elements using heap sort, elements will be in order as 1,3,4,7,9.

4. Assume you are tasked to schedule computer tasks. Given task from t1 to tn you must schedule them in the given manner down below.
   Note:
   Each task that is being pooled comes with a priority (generate this priority randomly between values 1 to 10). The value with the highest priority gets the first treatment and then subsequent nodes will get later priorities. Once the tree is built up delete the nodes accordingly to the priority (Max to min) while also printing the order.

5. Given the elements 1, 2, 3, 5, 4, 9 implement a priority Queue. With the highest being 9 and the lowest being 1 (Note: Do not use the built-In Queue function use heap trees for its implementation).

6. Given an array a of length n, find the minimum number of operations required to make it non-increasing. You can select any one of the following operations and perform it any number of times on an array element.
   - increment the array element by 1.
   - decrement the array element by 1.

   Input:
   N = 4
   array = {3, 1, 2, 1}
   Output:
   1
   Explanation:
   Decrement array[2] by 1. New array becomes {3,1,1,1} which is non-increasing. Therefore, only 1 step is required.
   Note: Use priority Queue for this question