

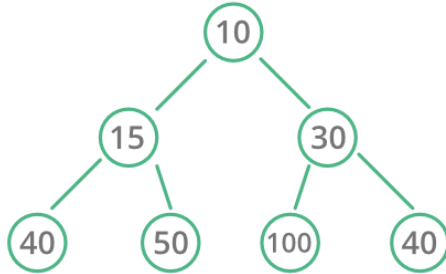


Islamic University of Gaza
Faculty of Engineering
Computer Engineering
Department

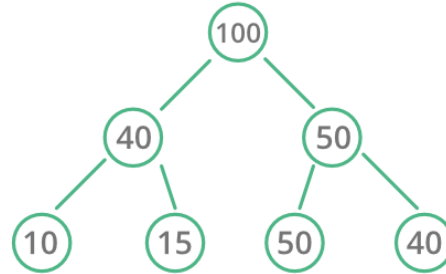


ECOM 3001 Data Structures and Algorithms Lab [Fall 2024]

Lab "08" Topic: Heaps and Priority Queue



Min Heap



Max Heap

Prepared by:
Eng. Waseem Alfarram

Supervised by:
Dr. Ahmad Mahdi

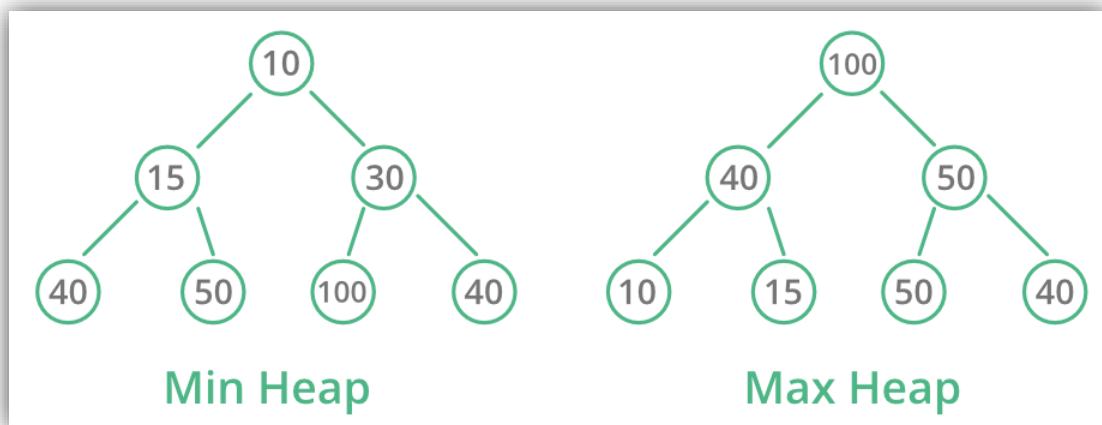
Lab Objectives:

- To define **Heap** data structure and **its Types**.
- To recognize some **Operations on Heap**.
- To **implement Max-Heap** using Array in Java.
- To get familiar with some **Advantages, Disadvantages and Applications of Heap** in real-life scenarios.
- To define **Priority Queue** data structure, **its Properties** and **How priority is assigned to the elements in the PQ**.
- To recognize **Types of Priority Queue**.
- To **differentiate between Normal Queue and Priority Queue**.
- To recognize **data structures we can use to build and implement a Priority Queue**.
- To recognize some **Operations on Priority Queue**.
- To **implement Priority Queue** using LinkedList in Java.
- To get familiar with a **Comparative Analysis between different implementations of Priority Queue**.
- To get familiar with some **Applications, Advantages and Disadvantages of Priority Queue** in real-life scenarios.
- To **solve some Exercises about Heap and Priority Queue** data structures to deepen your understanding for this lab topics.

What is Heap?

A **heap** is a complete binary tree 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 which generates two types of Heaps:

- **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.

➤ Heap Operations:

- **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 on that node. It takes **$O(\log n)$** to balance the tree.
- **Insert:**
When we insert a new element into the heap, it will distort the properties of the heap so we need to perform the heapify operations so that it maintains the property of the heap. This operation also takes **$O(\log n)$** time.
- **Delete:**
When we delete an element from the heap, it will always delete the root element of the tree as it is the maximum/minimum element. So, we can assign the intended element to be deleted the most maximum/minimum value it can afford and perform heapify operations so that the intended element to be deleted will be the root and then we can remove it easily and apply heapify operations. This operation takes **$O(\log n)$** time.
- **getMax or getMin:**
It finds the maximum element or minimum element for max-heap and min-heap respectively and as we know minimum and maximum elements will always be the root node itself for min-heap and max-heap respectively. This takes **$O(1)$** time.
- **removeMax or removeMin:**
This operation returns and deletes the maximum element and minimum element from the max-heap and min-heap respectively. In short, it deletes the root element of the heap binary tree. This takes **$O(\log n)$** time.

➤ Max-Heap Implementation in Java using Array:

```
// Max-Heap implementation in Java using Array

class MaxHeap {
    int[] arr;
    int maxSize;
    int heapSize;

    MaxHeap(int maxSize) {
        this.maxSize = maxSize;
        arr = new int[maxSize];
        heapSize = 0;
    }

    // Returns the index of the parent of the element at ith index
    int parent(int i) {
        return (i - 1) / 2;
    }

    // Inserts a new key 'x' in the Max Heap
    void insertKey(int x) {
        // To check whether the key can be inserted or not
        if (heapSize == maxSize) {
            System.out.println("\nOverflow: Could not insertKey\n");
            return;
        }

        // The new key is initially inserted at the end
        heapSize++;
        int i = heapSize - 1;
        arr[i] = x;

        // The max heap property is checked and if violation occurs,
        // it is restored
        while (i != 0 && arr[parent(i)] < arr[i]) {
            int temp = arr[i];
            arr[i] = arr[parent(i)];
            arr[parent(i)] = temp;
            i = parent(i);
        }
    }

    // Heapifies a sub-tree taking the given index as the root
    void MaxHeapify(int i) {
        int leftChild = 2*i + 1;
        int rightChild = 2*i + 2;
        int largest = i;
        if (leftChild < heapSize && arr[leftChild] > arr[i])
            largest = leftChild;
        if (rightChild < heapSize && arr[rightChild] > arr[largest])
            largest = rightChild;
        if (largest != i) {

```

```

        int temp = arr[i];
        arr[i] = arr[largest];
        arr[largest] = temp;
        MaxHeapify(largest);
    }
}

// Removes the root which in this case contains the maximum
// element
int removeMax() {
    // Checking whether the heap array is empty or not
    if (heapSize == 0)
        return Integer.MIN_VALUE;
    if (heapSize == 1) {
        heapSize--;
        return arr[0];
    }

    // Storing the maximum element to remove it
    int root = arr[0];
    arr[0] = arr[heapSize - 1];
    heapSize--;

    // To restore the property of the Max heap
    MaxHeapify(0);

    return root;
}

// Increases value of key at index 'i' to new_val
void increaseKey(int i, int newVal) {
    arr[i] = newVal;
    while (i != 0 && arr[parent(i)] < arr[i]) {
        int temp = arr[i];
        arr[i] = arr[parent(i)];
        arr[parent(i)] = temp;
        i = parent(i);
    }
}

// Deletes a key at given index i
void deleteKey(int i) {
    // It increases the value of the key to infinity and then
    // removes the maximum value
    increaseKey(i, Integer.MAX_VALUE);
    removeMax();
}

// Returns the maximum key (key at root) from max heap
int getMax() {
    return arr[0];
}

```

```

    int currentSize() {
        return heapSize;
    }

    void printHeap() {
        for (int i = 0; i < heapSize; i++) {
            System.out.print(arr[i] + " ");
        }
        System.out.println();
    }
}

public class MaxHeapTest {
    public static void main(String[] args) {
        // Assuming the maximum size of the heap to be 12
        MaxHeap h = new MaxHeap(12);

        h.insertKey(3);
        h.insertKey(9);
        h.insertKey(2);
        h.insertKey(1);
        h.insertKey(4);
        h.insertKey(5);
        System.out.println("The current size of the heap is " +
h.currentSize());
        h.printHeap();
        System.out.println("The current maximum element is " +
h.getMax());

        // Inserting a key value of 15 and 4
        h.insertKey(15);
        h.insertKey(4);
        System.out.println("\nThe current size of the heap after
insertion is " + h.currentSize());
        h.printHeap();

        // Deleting key at index 1
        h.deleteKey(1);
        System.out.println("\nThe current size of the heap after
deleting is " + h.currentSize());
        h.printHeap();
    }
}

```

Output:

The current size of the heap is 6

9 4 5 1 3 2

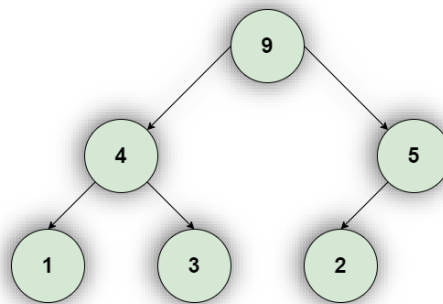
The current maximum element is 9

The current size of the heap after insertion is 8

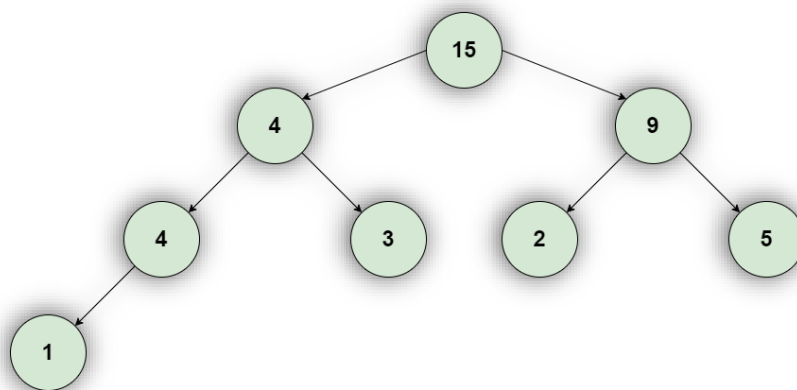
15 4 9 4 3 2 5 1

The current size of the heap after deleting is 7

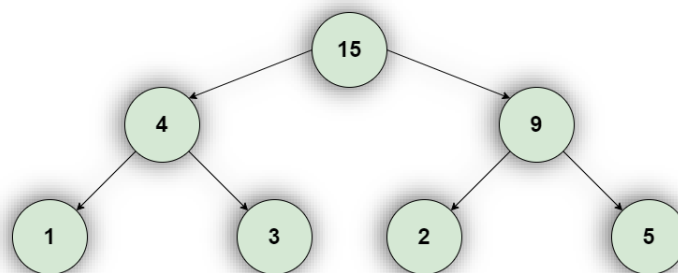
15 4 9 1 3 2 5



(Max-Heap)



(Max-Heap after insertion)



(Max-Heap after deletion)

➤ Advantages of Heaps:

- Fast access to maximum/minimum element ($O(1)$).
- Efficient Insertion and Deletion operations ($O(\log n)$) Flexible size.
- Can be efficiently implemented as an array
- Suitable for real-time applications.

➤ Disadvantages of Heaps:

- Not suitable for searching for an element other than maximum/minimum ($O(n)$ in worst case).
- Extra memory overhead to maintain heap structure.
- Slower than other data structures like arrays and linked lists for non-priority queue operations.

➤ Heap Data Structure Applications:

- Heaps are commonly used to **implement priority queues**, where elements are retrieved based on their priority (maximum or minimum value).
- **Heapsort** is a sorting algorithm that uses a heap to sort an array in ascending or descending order.
- Heaps are used in **graph algorithms** like Dijkstra's algorithm and Prim's algorithm for finding the shortest paths and minimum spanning trees.

What is Priority Queue?

A **priority queue** is a type of queue that arranges elements based on their priority values. Elements with higher priority values are typically retrieved or removed before elements with lower priority values. Each element has a priority value associated with it. When we add an item, it is inserted in a position based on its priority value.

There are several ways to implement a priority queue, including using an array, linked list, heap, or binary search tree, **binary heap** being the most common method to implement. The reason for using Binary Heap is simple, in binary heaps, we have easy access to the min (in min heap) or max (in max heap) and binary heap being a complete binary tree are easily implemented using arrays. Since we use arrays, we have cache friendliness advantage also.

Priority queues are often used in real-time systems, where the order in which elements are processed is not simply based on the fact who came first (or inserted first), but based on priority.

➤ Properties of Priority Queue Data Structure:

- Every item has a priority associated with it.
- An element with high priority is dequeued before an element with low priority.
- If two elements have the same priority, they are served according to their order in the queue.

Priority Queue		
Initial Queue = { }		
Operation	Return value	Queue Content
insert (C)		C
insert (O)		C O
insert (D)		C O D
remove max	O	C D
insert (I)		C D I
insert (N)		C D I N
remove max	N	C D I
insert (G)		C D I G

(**Priority Queue**, an element with a maximum ASCII value will have the highest priority)

➤ How is Priority assigned to the elements in a PQ?

In a priority queue, generally, the value of an element is considered for assigning the priority.

For example, the element with the highest value is assigned the highest priority and the element with the lowest value is assigned the lowest priority. The reverse case can also be used i.e., the element with the lowest value can be assigned the highest priority. Also, the priority can be assigned according to our needs.

➤ Types of Priority Queue:

1. Ascending Order Priority Queue:

As the name suggests, in ascending order priority queue, the element with a lower priority value is given a higher priority in the priority list. For example, if we have the following elements in a priority queue arranged in ascending order like 4,6,8,9,10. Here, 4 is the smallest number, therefore, it will get the highest priority in a priority queue and so when we dequeue from this type of priority queue, 4 will remove from the queue and dequeue returns 4.

2. Descending Order Priority Queue:

The root node is the maximum element in a max heap, as you may know. It will also remove the element with the highest priority first. As a result, the root node is removed from the queue. This deletion leaves an empty space, which will be filled with fresh insertions in the future. The heap invariant is then maintained by comparing the newly inserted element to all other entries in the queue.

➤ Difference between PQ and Normal Queue:

There is no priority attached to elements in a queue, the rule of **First-In-First-Out (FIFO)** is implemented whereas, in a priority queue, the **elements have a priority**. The elements with higher priority are served first.

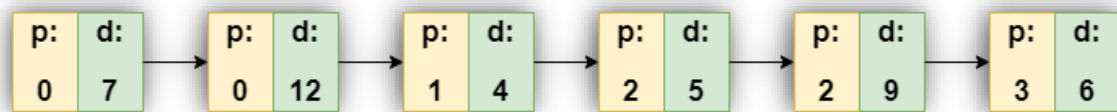
➤ How to implement Priority Queue?

- Arrays
- Heap
- Linked List
- Binary Search Tree

➤ Operations of a Priority Queue:

- **Insertion:**
When a new element is inserted in a priority queue, it moves to the empty slot from top to bottom and left to right. However, if the element is not in the correct place, then it will be compared with the parent node. If the element is not in the correct order, the elements are swapped. The swapping process continues until all the elements are placed in the correct position.
- **Deletion:**
As you know that in a max heap, the maximum element is the root node. And it will remove the element which has maximum priority first. Thus, you remove the root node from the queue. This removal creates an empty slot, which will be further filled with new insertion. Then, it compares the newly inserted element with all the elements inside the queue to maintain the heap invariant.
- **Peek:**
This operation helps to return the maximum element from Max Heap or the minimum element from Min Heap without deleting the node from the priority queue.
- **Pop:**
This operation helps to return the maximum element from Max Heap or the minimum element from Min Heap with deleting the node from the priority queue.

➤ Priority Queue Implementation using LinkedList in Java:



```
// Priority Queue implementation in Java using LinkedList
class PriorityQueue {
    class Node {
        int data;
        // Lower values indicate higher priority
        int priority;
        Node next;

        public Node(int d, int p) {
            this.data = d;
            this.priority = p;
            this.next = null;
        }
    }

    Node pqHead;
    int size = 0;

    public PriorityQueue() {
        this.pqHead = null;
    }

    public PriorityQueue(int d, int p) {
        Node newNode = new Node(d, p);
        this.pqHead = newNode;
        size++;
    }

    // Function to check if the PQ is empty
    boolean isEmpty() {
        return (size == 0);
    }

    // Function to push according to priority
    void push(int d, int p) {
        // Create new Node
        Node newNode = new Node(d, p);

        if (this.size == 0) {
            this.pqHead = newNode;
        }
        // Special Case: The head of list has lesser
        // priority than new node. So insert new
        // node before head node and change head node.
    }
}
```

```

        else if (p < this.pqHead.priority) {
            // Insert New Node before head
            newNode.next = this.pqHead;
            this.pqHead = newNode;
        }
        else {
            Node start = this.pqHead;
            // Traverse the list and find a position to insert new node
            while (start.next != null && p >= start.next.priority) {
                start = start.next;
            }
            // Either at the ends of the list or at required position
            newNode.next = start.next;
            start.next = newNode;
        }
        size++;
    }

    // Return the value at head
    int peek() {
        return this.pqHead.data;
    }

    // Removes the element with the highest priority from the list
    int pop() {
        int removedData = this.pqHead.data;
        this.pqHead = this.pqHead.next;
        size--;
        return removedData;
    }
}

public class test {
    public static void main(String args[]) {
        // Create a Priority Queue
        // 7 >> 12 >> 4 >> 5 >> 9 >> 6
        PriorityQueue pq = new PriorityQueue();
        pq.push(4, 1);
        pq.push(5, 2);
        pq.push(6, 3);
        pq.push(7, 0);
        pq.push(9, 2);
        pq.push(12, 0);

        while (!pq.isEmpty()) {
            System.out.printf("%d ", pq.peek());
            pq.pop();
        }
    }
}

```

Output:

7 12 4 5 9 6

➤ Comparative Analysis of Different Implementations of PQ:

Arrays	enqueue()	dequeue()	peek()
Time Complexity	$O(1)$	$O(n)$	$O(n)$

Linked List	push()	pop()	peek()
Time Complexity	$O(n)$	$O(1)$	$O(1)$

Binary Heap	insert()	remove()	peek()
Time Complexity	$O(\log n)$	$O(\log n)$	$O(1)$

Binary Search Tree	peek()	insert()	delete()
Time Complexity	$O(1)$	$O(\log n)$	$O(\log n)$

➤ Applications of Priority Queue:

- CPU Scheduling.
- Graph algorithms like [Dijkstra's shortest path algorithm](#), [Prim's Minimum Spanning Tree](#), etc.
- All queue applications where priority is involved.
- Data compression in Huffman code.
- Event-driven simulation such as customers waiting in a queue.
- Finding Kth largest/smallest element.

➤ Advantages of Priority Queue:

- **It helps to access the elements in a faster way:** This is because elements in a priority queue are ordered by priority, one can easily retrieve the highest priority element without having to search through the entire queue.
- **The ordering of elements in a Priority Queue is done dynamically:** Elements in a priority queue can have their priority values updated, which allows the queue to dynamically reorder itself as priorities change.
- **Efficient algorithms can be implemented:** Priority queues are used in many algorithms to improve their efficiency, such as Dijkstra's algorithm for finding the shortest path in a graph and the A* search algorithm for pathfinding.
- **Included in real-time systems:** This is because priority queues allow you to quickly retrieve the highest priority element, they are often used in real-time systems where time is of the essence.

➤ Disadvantages of Priority Queue:

- **High complexity:** Priority queues are more complex than simple data structures like arrays and linked lists, and may be more difficult to implement and maintain.
- **High consumption of memory:** Storing the priority value for each element in a priority queue can take up additional memory, which may be a concern in systems with limited resources.
- **It is not always the most efficient data structure:** In some cases, other data structures like heaps or binary search trees may be more efficient for certain operations, such as finding the minimum or maximum element in the queue.
- **At times it is less predictable:** This is because the order of elements in a priority queue is determined by their priority values, the order in which elements are retrieved may be less predictable than with other data structures like stacks or queues, which follow a First-In, First-Out (FIFO) or Last-In, First-Out (LIFO) order.

Leetcode Exercises:

- leetcode.com/problems/delete-greatest-value-in-each-row/
- leetcode.com/problems/maximum-product-of-two-elements-in-an-array/
- leetcode.com/problems/minimum-amount-of-time-to-fill-cups/
- leetcode.com/problems/kth-largest-element-in-an-array/
- leetcode.com/problems/single-threaded-cpu/

End of Lab 😊