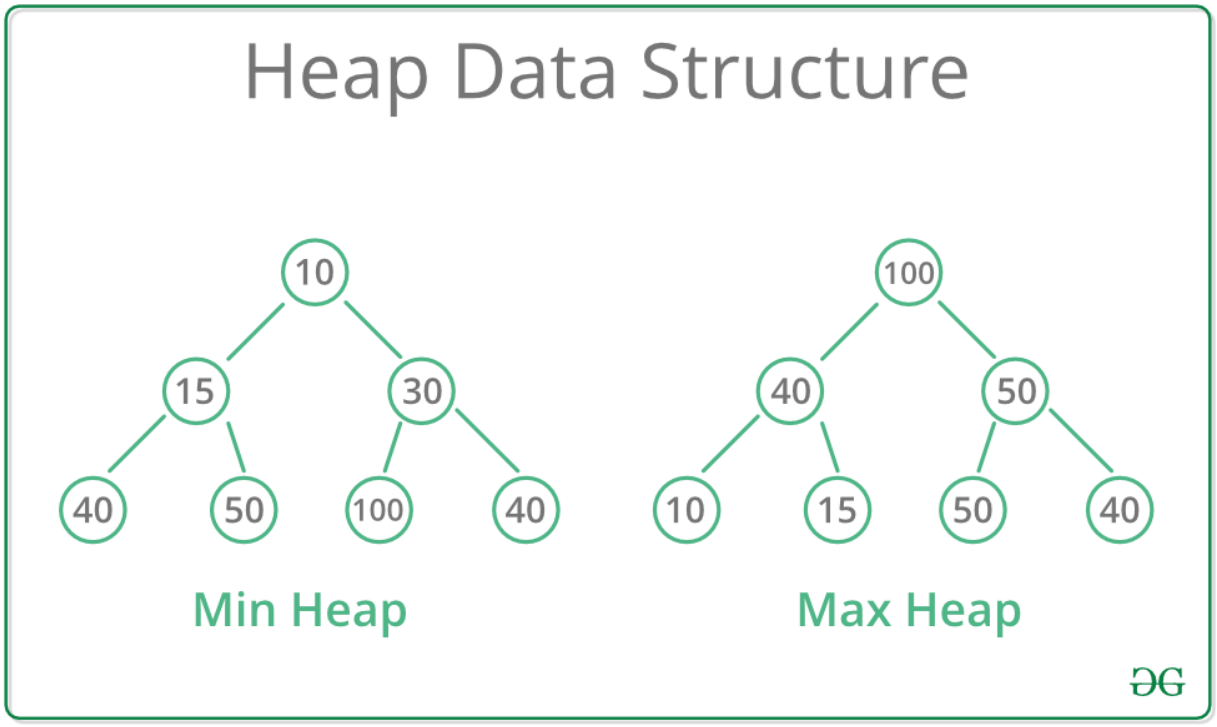
Heap

A **Heap** is a complete binary tree data structure that satisfies the heap property: for every node, the value of its children is greater than or equal to its own value. Heaps are usually used to implement priority queues, where the smallest (or largest) element is always at the root of the tree.



What is Heap Data Structure?

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.

Types of Heaps

There are two main types of heaps:

- **Max Heap:** The root node contains the maximum value, and the values decrease as you move down the tree.
- **Min Heap:** The root node contains the minimum value, and the values increase as you move down the tree.

Heap Operations

Common heap operations are:

- **Insert**: Adds a new element to the heap while maintaining the heap property.
- **Extract Max/Min:** Removes the maximum or minimum element from the heap and returns it.
- **Heapify**: Converts an arbitrary binary tree into a heap.

Heap Data Structure Applications

Heaps have various applications, like:

- 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.

Max Heap Implementation in Java:

```
import java.util.Arrays;

public class MaxHeap {

    private int[] heap;

    private int size;

    private int maxSize;


    public MaxHeap(int maxSize) {

        this.maxSize = maxSize;

        this.size = 0;

        this.heap = new int[this.maxSize + 1];
```

```
    // Fill the first element with a large value (sentinel value)
    this.heap[0] = Integer.MAX_VALUE;
}
// Return position of the parent
private int parent(int pos) {
    return pos / 2;
}
// Return position of the left child
private int leftChild(int pos) {
    return (2 * pos);
}
// Return position of the right child
private int rightChild(int pos) {
    return (2 * pos) + 1;
}
// Check if node is a leaf
private boolean isLeaf(int pos) {
    return pos >= (size / 2) && pos <= size;
}
```

```
    // Swap two nodes
    private void swap(int fpos, int spos) {
        int tmp;
        tmp = heap[fpos];
        heap[fpos] = heap[spos];
        heap[spos] = tmp;
    }
    // Max-heapify the node at pos
    private void maxHeapify(int pos) {
        if (!isLeaf(pos)) {
            if (heap[pos] < heap[leftChild(pos)] || heap[pos] < heap[rightChild(pos)]) {
                if (heap[leftChild(pos)] > heap[rightChild(pos)]) {
                    swap(pos, leftChild(pos));
                    maxHeapify(leftChild(pos));
                } else {
                    swap(pos, rightChild(pos));
                    maxHeapify(rightChild(pos));
                }
```

```
        }
    }
}
// Insert an element into the heap
public void insert(int element) {
    if (size >= maxSize) {
        return;
    }
    heap[++size] = element;
    int current = size;
    while (heap[current] > heap[parent(current)]) {
        swap(current, parent(current));
        current = parent(current);
    }
}
// Remove and return the maximum element (root)
public int extractMax() {
    int popped = heap[1];
    heap[1] = heap[size--];
```

```
        maxHeapify(1);
        return popped;
    }
    // Build the max heap
    public void maxHeap() {
        for (int pos = (size / 2); pos >= 1; pos--) {
            maxHeapify(pos);
        }
    }
    // Print the heap
    public void print() {
        for (int i = 1; i <= size / 2; i++) {
            System.out.print("Parent: " + heap[i] + " Left: " + heap[2 * i] + "
Right:" + heap[2 * i + 1]);
            System.out.println();
        }
    }
    public static void main(String[] args) {
        MaxHeap maxHeap = new MaxHeap(15);
```

```
        maxHeap.insert(5);
        maxHeap.insert(3);
        maxHeap.insert(17);
        maxHeap.insert(10);
        maxHeap.insert(84);
        maxHeap.insert(19);
        maxHeap.insert(6);
        maxHeap.insert(22);
        maxHeap.insert(9);

        maxHeap.maxHeap();
        maxHeap.print();
        System.out.println("The Max val is " + maxHeap.extractMax());
    }
}
```

Min Heap Implementation in Java:

```
import java.util.Arrays;
```

```
public class MinHeap {
    private int[] heap;
    private int size;
    private int maxSize;

    public MinHeap(int maxSize) {
        this.maxSize = maxSize;
        this.size = 0;
        this.heap = new int[this.maxSize + 1];
        // Fill the first element with a small value (sentinel value)
        this.heap[0] = Integer.MIN_VALUE;
    }
    // Return position of the parent
    private int parent(int pos) {
        return pos / 2;
    }
    // Return position of the left child
    private int leftChild(int pos) {
        return (2 * pos);
```

```
}
// Return position of the right child
private int rightChild(int pos) {
    return (2 * pos) + 1;
}

// Check if node is a leaf
private boolean isLeaf(int pos) {
    return pos >= (size / 2) && pos <= size;
}
// Swap two nodes
private void swap(int fpos, int spos) {
    int tmp;
    tmp = heap[fpos];
    heap[fpos] = heap[spos];
    heap[spos] = tmp;
}
// Min-heapify the node at pos
private void minHeapify(int pos) {
```

```
        if (!isLeaf(pos)) {
            if (heap[pos] > heap[leftChild(pos)] || heap[pos] > heap[rightChild(pos)]) {
                if (heap[leftChild(pos)] < heap[rightChild(pos)]) {
                    swap(pos, leftChild(pos));
                    minHeapify(leftChild(pos));
                } else {
                    swap(pos, rightChild(pos));
                    minHeapify(rightChild(pos));
                }
            }
        }
    }
    // Insert an element into the heap
    public void insert(int element) {
        if (size >= maxSize) {
            return;
        }
        heap[++size] = element;
```

```
    int current = size;


    while (heap[current] < heap[parent(current)]) {
        swap(current, parent(current));
        current = parent(current);
    }
}
// Remove and return the minimum element (root)
public int extractMin() {
    int popped = heap[1];
    heap[1] = heap[size--];
    minHeapify(1);
    return popped;
}
// Build the min heap
public void minHeap() {
    for (int pos = (size / 2); pos >= 1; pos--) {
        minHeapify(pos);
    }
```

```
    }
    // Print the heap
    public void print() {
        for (int i = 1; i <= size / 2; i++) {
            System.out.print("Parent: " + heap[i] + " Left: " + heap[2 * i] + "
Right:" + heap[2 * i + 1]);
            System.out.println();
        }
    }
    public static void main(String[] args) {
        MinHeap minHeap = new MinHeap(15);
        minHeap.insert(5);
        minHeap.insert(3);
        minHeap.insert(17);
        minHeap.insert(10);
        minHeap.insert(84);
        minHeap.insert(19);
        minHeap.insert(6);
        minHeap.insert(22);
```

```
        minHeap.insert(9);


        minHeap.minHeap();
        minHeap.print();
        System.out.println("The Min val is " + minHeap.extractMin());
    }
}
```

Max-Priority Queue using Max Heap:

```
import java.util.Arrays;
class PriorityQueue {
    private int[] heap;
    private int size;
    private int maxSize;
    // Constructor to initialize the priority queue
    public PriorityQueue(int maxSize) {
        this.maxSize = maxSize;
        this.size = 0;
```

```
        heap = new int[this.maxSize + 1]; // Extra space for easier index management
        heap[0] = Integer.MAX_VALUE; // Sentinel value at index 0 for easier calculations
    }
    // Return position of parent
    private int parent(int pos) {
        return pos / 2;
    }
    // Return position of left child
    private int leftChild(int pos) {
        return 2 * pos;
    }
    // Return position of right child
    private int rightChild(int pos) {
        return (2 * pos) + 1;
    }
    // Check if a node is a leaf node
    private boolean isLeaf(int pos) {
```

```
        return pos > size / 2 && pos <= size;
    }
    // Swap two nodes of the heap
    private void swap(int fpos, int spos) {
        int tmp = heap[fpos];
        heap[fpos] = heap[spos];
        heap[spos] = tmp;
    }
    // Heapify the node at position `pos`
    private void maxHeapify(int pos) {
        if (!isLeaf(pos)) {
            if (heap[pos] < heap[leftChild(pos)] || heap[pos] < heap[rightChild(pos)]) {
                if (heap[leftChild(pos)] > heap[rightChild(pos)]) {
                    swap(pos, leftChild(pos));
                    maxHeapify(leftChild(pos));
                } else {
                    swap(pos, rightChild(pos));
                    maxHeapify(rightChild(pos));
```

```
            }
        }
    }
}
// Insert an element into the priority queue
public void insert(int element) {
    if (size >= maxSize) {
        System.out.println("Priority queue is full.");
        return;
    }
    heap[++size] = element;  // Insert element at the last position
    int current = size;
    // Rebalance the heap
    while (heap[current] > heap[parent(current)]) {
        swap(current, parent(current));
        current = parent(current);
    }
}
```

```
    // Remove and return the element with the highest priority (root of the heap)
    public int extractMax() {
        int max = heap[1]; // The max element is at the root
        heap[1] = heap[size--]; // Replace root with last element and reduce size
        maxHeapify(1);  // Restore heap property from the root
        return max;
    }
    // Print the priority queue (heap structure)
    public void print() {
        for (int i = 1; i <= size / 2; i++) {
            System.out.println("Parent: " + heap[i] + " Left: " + heap[2 * i] + " Right: " + heap[2 * i + 1]);
        }
    }
    // Build a max heap
    public void buildMaxHeap() {
        for (int pos = size / 2; pos >= 1; pos--) {
            maxHeapify(pos);
```

```
    }
}
public static void main(String[] args) {
    PriorityQueue priorityQueue = new PriorityQueue(10);
    priorityQueue.insert(5);
    priorityQueue.insert(3);
    priorityQueue.insert(17);
    priorityQueue.insert(10);
    priorityQueue.insert(84);
    priorityQueue.insert(19);
    priorityQueue.insert(6);
    priorityQueue.insert(22);
    priorityQueue.insert(9);

    System.out.println("Max Heap built from inserted elements:");
    priorityQueue.buildMaxHeap();
    priorityQueue.print();
```

```
        System.out.println("\nMax Element: " +
priorityQueue.extractMax());
        System.out.println("Max Element: " + priorityQueue.extractMax());
    }
}
```

Home task: Implement Min-Priority Queue using Min Heap.