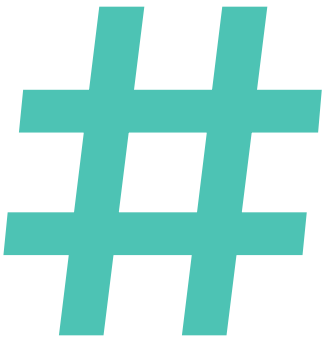


Heap

Step 01 What is a Heap Data Strucutre

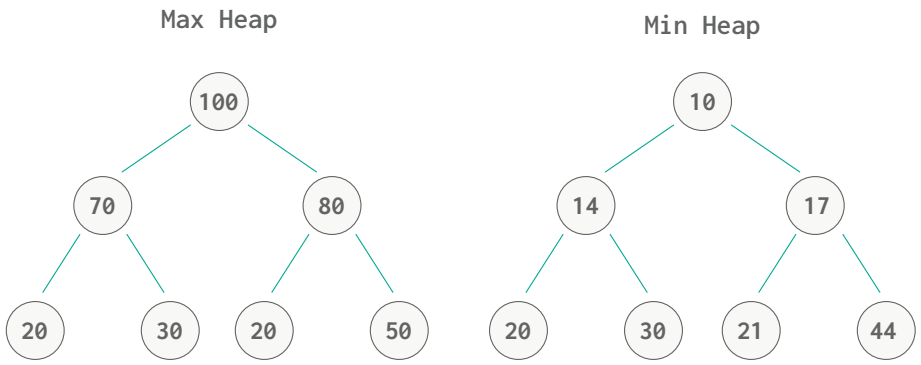
Heap is a hierarchical data structure that can be implemented using various underlying structures, such as arrays or trees. The heap property ensures that the highest/lowest value element is always at the root of the heap. However, there is no specific order or relationship between nodes at any level, so the heap is not sorted.

Step 02 Types of Heaps

There are two main types of heap:

01 **Max Heap:** The value of each node is less than or equal to the value of the parent, and the greatest value is at the root.

02 **Min Heap:** The value of each node is greater than or equal to the value of its parent, and the smallest value is at the root.



Step 03 Heap Operations

- **Heapify:** Constructs a heap from an unordered array.
- **Insert:** Inserts an element into the heap while ensuring the heap property.
- **Delete:** Removes an element from the heap while ensuring the heap property.
- **Peek (Find Max/Min):** Retrieves the highest/lowest root element without removing it.
- **Extract(Max/Min):** Retrieves the highest/lowest element and removes it.

Step 04 Example of Heaps applications

- Heap is used while implementing a priority queue.
- Heapsort is a sorting algorithm that uses a heap to sort the value in ascending or descending order.
- Binary Heap is used for pathfinding algorithms to find the shortest path between two points in a graph.

Step 05 How to Implement Heap Data strucutre

We are going to implement a max-heap class using an array.

Each element in an array at index i has:

- A parent at index (i-1)/2.
- A left child at index 2 * i + 1.
- A right child at index 2 * i + 2.

```
01  public class MaxHeap {
02      public int[] heap;
03      public int count;
04
05      public MaxHeap(int capacity) {
06          heap = new int[capacity];
07          count = 0;
08      }
```

CODE MaxHeap Class Fields and the Constructor.

- **heap**: An array to store the elements of the heap.
- **count**: The number of elements currently stored in the heap.
- MaxHeap(int capacity) Constructor.
  - Initializes the heap array with the specified capacity.
  - Sets the count to 0, indicating an empty heap.

```
01  public boolean isEmpty() {
02          return count == 0;
03      }
```

CODE isEmpty() Method: Returns true if the heap is empty (count is 0), false otherwise.

```
01      public void ensureCapacity() {
02          if (count == heap.length) {
03              in t[] newHeap = new int[heap.length * 2];
04              System.arraycopy(heap, 0, newHeap, 0, count);
05              heap = newHeap;
06          }
07      }
```

CODE ensureCapacity() Method: Resizes the array by doubling its length when it becomes full.

```
01    public void insert(int value) {
02          ensureCapacity();
03          heap[count++] = value;
04          heapifyUp(count - 1);
05      }
```

CODE insert(int value) Method: Inserts a new element into the heap.

- Calls ensureCapacity() to ensure enough space.
- Places the new element at the end of the heap and applies heapifyUp to maintain the heap property.

```
01  public void delete(int value) {
02          if (isEmpty()) {
03              System.out.println("the heap is empty ");
04          }
     // Search for the index of the required value
05          int index = -1;
06          for (int i = 0; i < count; i++) {
07              if (heap[i] == value) {
08                  index = i;
09                  break;
10              }
11          }
12          if (index == -1) {
13              System.out.println("the element is not found ");
14          }
     // Replaces the value with the last element in the heap and resizes the array.
15          heap[index] = heap[--count];
16          int[] newHeap = new int[heap.length -1];
17          System.arraycopy(heap, 0, newHeap, 0, count);
18          heap = newHeap;
19          heapifyDown(index);
20          System.out.println("Number " + value + " has been deleted");
21  }
```

CODE delete(int value) Method: Removes the specified value from the heap.

- Replace the element to be deleted with the last element in the heap.
- Resizes the array and calls heapifyDown to maintain the heap property.

```
01    public int peek() {
02          if (isEmpty()) {
03              return -1;
04          }
05          return heap[0];
06      }
```

CODE peek() Method: Returns the maximum element in the heap, or -1 if the heap is empty.

```
01    public int extract() {
02          int max = peek();
03          if (max != -1) {
04              System.out.print("Extract method: ");
05              delete(heap[0]);
06          }
07          return max;
08      }
```

CODE extract() Method: Retrieves and removes the maximum element from the heap.

- Calls delete(heap[0]) to remove the root element and maintain the heap property.
- Returns the extracted maximum value.

```
01      public void heapifyUp(int index) {
        // Calculates the index of the parent element.
02      int parent = (index - 1) / 2;
        //enters a loop that continues until the `index` reaches the root of the heap, or
        the current element is not greater than its parent.
03          while (index > 0 && heap[index] > heap[parent]) {
04              swap(index, parent);
05              index = parent;
06              heapifyUp(index);
07          }
08      }
```

CODE

- heapifyUp(int index) Method: Maintain the heap property.
- Repeatedly swapping the element with its parent until the heap property is satisfied.

```
01      public void heapifyDown(int index) {
        // Calculates the index of the left Child element.
02          int leftChild = 2 * index + 1;
        // Calculates the index of the right Child element
03          int rightChild = 2 * index + 2; .
        // Store the index of the largest element, initializing it as the current index.
04          int largest = index;
        // Checks if the leftchild index in the heap bounds, and the element in the leftchild
        index is larger than the element at the largest index, updating largest if necessary
05          if (leftChild < count && heap[leftChild] > heap[largest]) {
06              largest = leftChild;
07          }
        // Perform a similar check for the right child index, updating largest if necessary.
08          if (rightChild < count && heap[rightChild] > heap[largest]) {
09              largest = rightChild;
10          }
11
        // If the current index is not equal to the largest, the heap property is violated.
12          if (largest != index) {
13              swap(index, largest);
14              heapifyDown(largest);
15          }
16      }
```

CODE heapifyDown(int index) Method: Maintain the heap property.

- Repeatedly swaps the element with its larger child until the heap property is satisfied.

```
01      public void swap(int i, int j) {
02          int temp = heap[i];
03          heap[i] = heap[j];
04          heap[j] = temp;
05       }
06  }    public void swap(int i, int j) {
07          int temp = heap[i];
08          heap[i] = heap[j];
09          heap[j] = temp;
10       }
11  }
```

CODE swap(int i, int j) Method: Swaps two elements in the heap at indices i and j.

> **NOTE**
> For more information about `System.arraycopy()`

```
public class main {
    public static void main(String[] args) {
        MaxHeap heapNumbers = new MaxHeap(6);
        // Insert elements into the heap
        heapNumbers.insert(10);
        heapNumbers.insert(5);
        heapNumbers.insert(15);
        heapNumbers.insert(20);
        heapNumbers.insert(8);
        heapNumbers.insert(13);

        // Print the heap.
        System.out.print("Heap: ");
        for (int element : heapNumbers.heap) {
            System.out.print(element + " ");
        }
        System.out.println();

        // Print the peek of the heap.
        System.out.println("Heap peek: "+ heapNumbers.peek());
        // Call the extract.
        heapNumbers.extract();
        // Print the peek of the heap.
        System.out.println("Heap peek: "+ heapNumbers.peek());
       // Delete number 10 from the heap.
        heapNumbers.delete(10);
        // Print the heap.
        System.out.print("Heap: ");
        for (int element : heapNumbers.heap) {
            System.out.print(element + " ");
        }

    }
}
```

CODE The main class

OUTPUT

Heap: 20 15 13 5 8 10

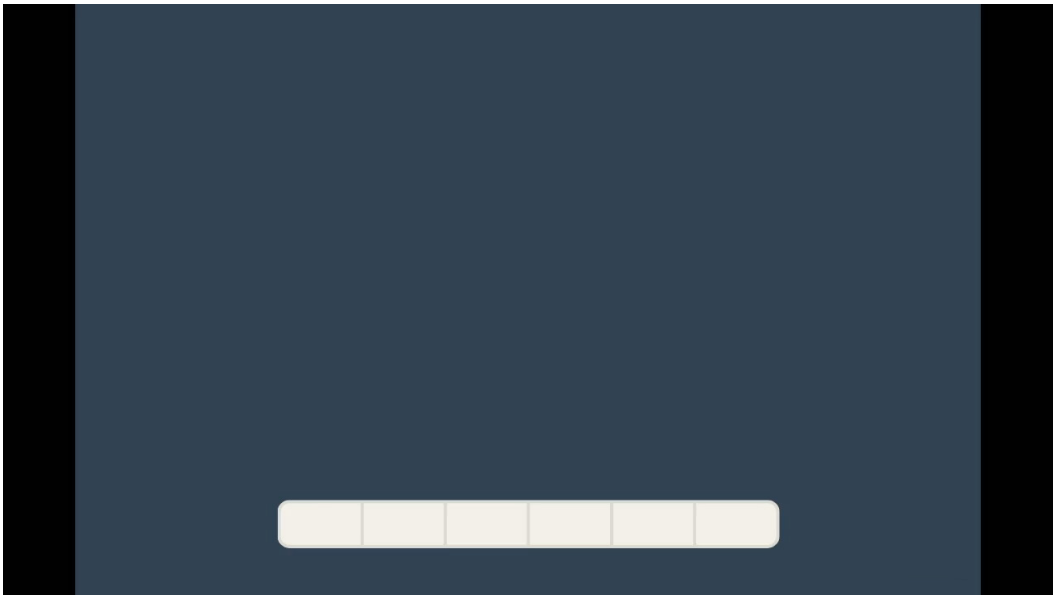
Heap peek: 20

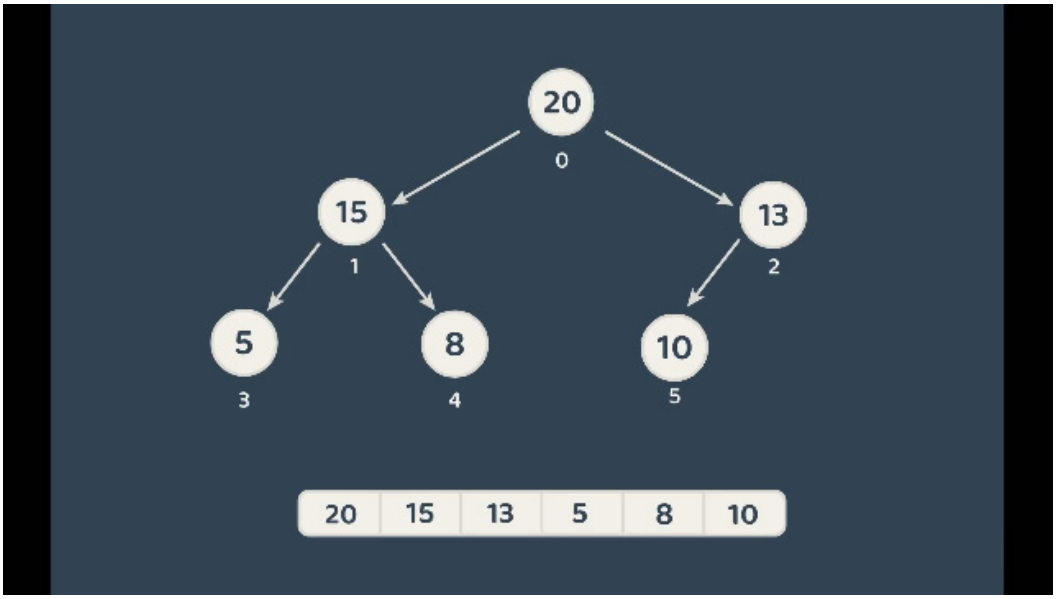
Extract method: Number 20 has been deleted

Heap peek: 15

Number 10 has been deleted

Heap: 15 8 13 5




20
0
15
1
13
2
5
3
8
4
10
5
20
15
13
5
8
10