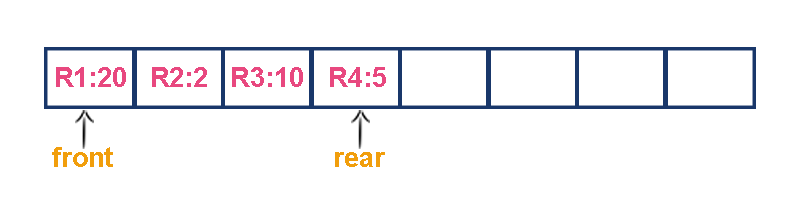
**Max Priority Queue**

In the normal queue data structure, insertion is performed at the end of the queue and deletion is performed based on the FIFO principle. This queue implementation may not be suitable for all applications.  
Consider a networking application where the server has to respond for requests from multiple clients using queue data structure. Assume four requests arrived at the queue in the order of R1, R2, R3 & R4 where R1 requires 20 units of time, R2 requires 2 units of time, R3 requires 10 units of time and R4 requires 5 units of time. A queue is as follows...



Now, check to wait time of each request that to be completed.

1. **R1 : 20 units of time**
2. **R2 : 22 units of time (R2 must wait until R1 completes 20 units and R2 itself requires 2 units. Total 22 units)**
3. **R3 : 32 units of time (R3 must wait until R2 completes 22 units and R3 itself requires 10 units. Total 32 units)**
4. **R4 : 37 units of time (R4 must wait until R3 completes 35 units and R4 itself requires 5 units. Total 37 units)**

**Here, the average waiting time for all requests (R1, R2, R3 and R4) is (20+22+32+37)/4 ≈ 27 units of time.**

That means, if we use a normal queue data structure to serve these requests the average waiting time for each request is 27 units of time.  
Now, consider another way of serving these requests. If we serve according to their required amount of time, first we serve R2 which has minimum time (2 units) requirement. Then serve R4 which has second minimum time (5 units) requirement and then serve R3 which has third minimum time (10 units) requirement and finally R1 is served which has maximum time (20 units) requirement.

Now, check to wait time of each request that to be completed.

1. **R2 : 2 units of time**
2. **R4 : 7 units of time (R4 must wait until R2 completes 2 units and R4 itself requires 5 units. Total 7 units)**
3. **R3 : 17 units of time (R3 must wait until R4 completes 7 units and R3 itself requires 10 units. Total 17 units)**
4. **R1 : 37 units of time (R1 must wait until R3 completes 17 units and R1 itself requires 20 units. Total 37 units)**

**Here, the average waiting time for all requests (R1, R2, R3 and R4) is (2+7+17+37)/4 ≈ 15 units of time.**

From the above two situations, it is very clear that the second method server can complete all four requests with very less time compared to the first method. This is what exactly done by the priority queue.

**Priority queue is a variant of a queue data structure in which insertion is performed in the order of arrival and deletion is performed based on the priority.**

There are two types of priority queues they are as follows...

1. **Max Priority Queue**
2. **Min Priority Queue**

**1. Max Priority Queue**

In a max priority queue, elements are inserted in the order in which they arrive the queue and the maximum value is always removed first from the queue. For example, assume that we insert in the order 8, 3, 2 & 5 and they are removed in the order 8, 5, 3, 2.  
The following are the operations performed in a Max priority queue...

1. **isEmpty() - Check whether queue is Empty.**
2. **insert() - Inserts a new value into the queue.**
3. **findMax() - Find maximum value in the queue.**
4. **remove() - Delete maximum value from the queue.**

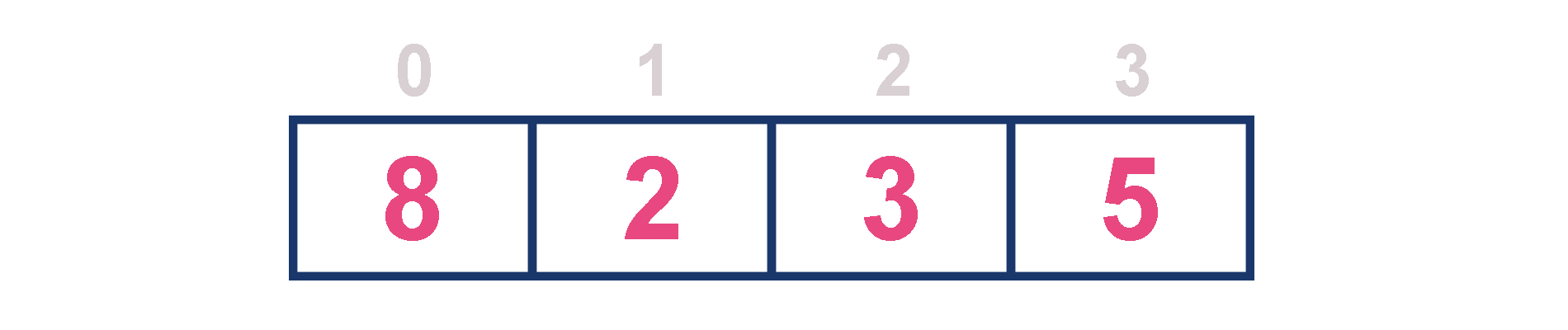
**Max Priority Queue Representations**

There are 6 representations of max priority queue.

1. **Using an Unordered Array**
2. **Using an Unordered Array with the index of the maximum value**
3. **Using an Array in Decreasing Order**
4. **Using an Array in Increasing Order**
5. **Using Linked List in Increasing Order**
6. **Using Unordered Linked List with reference to node with the maximum value**

**#1. Using an Unordered Array**

In this representation, elements are inserted according to their arrival order and the largest element is deleted first from the max priority queue.  
For example, assume that elements are inserted in the order of 8, 2, 3 and 5. And they are removed in the order 8, 5, 3 and 2.



Now, let us analyze each operation according to this representation...

**isEmpty()** - If '**front == -1**' queue is Empty. This operation requires **O(1)** time complexity which means constant time complexity.

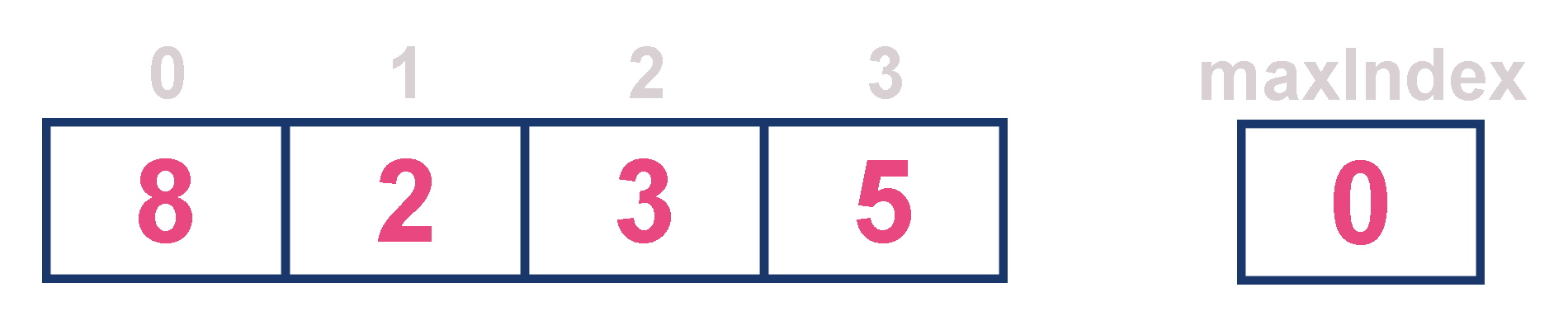
**insert()** - New element is added at the end of the queue. This operation requires **O(1)** time complexity which means constant time complexity.

**findMax()** - To find the maximum element in the queue, we need to compare it with all the elements in the queue. This operation requires **O(n)** time complexity.

**remove()** - To remove an element from the max priority queue, first we need to find the largest element using **findMax()** which requires **O(n)** time complexity, then that element is deleted with constant time complexity **O(1)**. The remove() operation requires **O(n) + O(1) ≈ O(n)** time complexity.

**#2. Using an Unordered Array with the index of the maximum value**

In this representation, elements are inserted according to their arrival order and the largest element is deleted first from max priority queue.  
For example, assume that elements are inserted in the order of 8, 2, 3 and 5. And they are removed in the order 8, 5, 3 and 2.



Now, let us analyze each operation according to this representation...

**isEmpty()** - If '**front == -1**' queue is Empty. This operation requires **O(1)** time complexity which means constant time complexity.

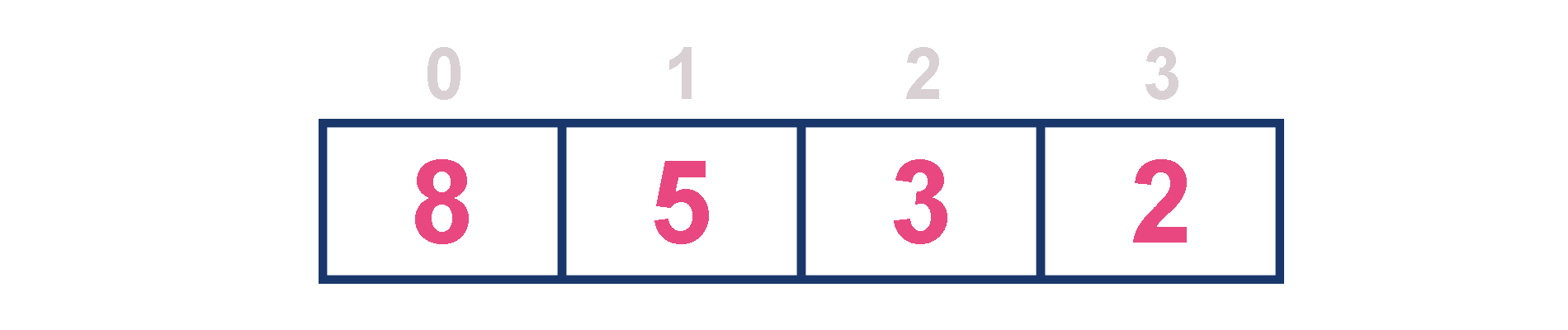
**insert()** - New element is added at the end of the queue with **O(1)** time complexity and for each insertion we need to update maxIndex with **O(1)** time complexity. This operation requires **O(1)** time complexity which means constant time complexity.

**findMax()** - Finding the maximum element in the queue is very simple because index of the maximum element is stored in maxIndex. This operation requires **O(1)** time complexity.

**remove()** - To remove an element from the queue, first we need to find the largest element using **findMax()** which requires **O(1)** time complexity, then that element is deleted with constant time complexity **O(1)** and finally we need to update the next largest element index value in maxIndex which requires **O(n)** time complexity. The remove() operation requires **O(1)+O(1)+O(n) ≈ O(n)** time complexity.

**#3. Using an Array in Decreasing Order**

In this representation, elements are inserted according to their value in decreasing order and largest element is deleted first from max priority queue.  
  
For example, assume that elements are inserted in the order of 8, 5, 3 and 2. And they are removed in the order 8, 5, 3 and 2.



Now, let us analyze each operation according to this representation...

**isEmpty()** - If '**front == -1**' queue is Empty. This operation requires **O(1)** time complexity which means constant time complexity.

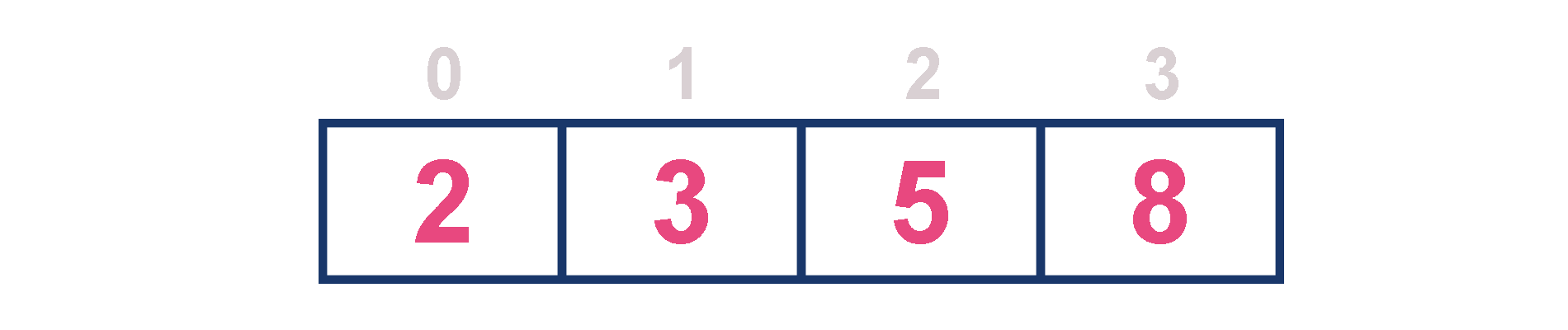
**insert()** - New element is added at a particular position based on the decreasing order of elements which requires **O(n)** time complexity as it needs to shift existing elements inorder to insert new element in decreasing order. This insert() operation requires **O(n)** time complexity.

**findMax()** - Finding the maximum element in the queue is very simple because maximum element is at the beginning of the queue. This findMax() operation requires **O(1)** time complexity.

**remove()** - To remove an element from the max priority queue, first we need to find the largest element using **findMax()** operation which requires **O(1)** time complexity, then that element is deleted with constant time complexity **O(1)** and finally we need to rearrange the remaining elements in the list which requires **O(n)** time complexity. This remove() operation requires **O(1) + O(1) + O(n) ≈ O(n)** time complexity.

**#4. Using an Array in Increasing Order**

In this representation, elements are inserted according to their value in increasing order and maximum element is deleted first from max priority queue.  
For example, assume that elements are inserted in the order of 2, 3, 5 and 8. And they are removed in the order 8, 5, 3 and 2.



Now, let us analyze each operation according to this representation...

**isEmpty()** - If '**front == -1**' queue is Empty. This operation requires **O(1)** time complexity which means constant time complexity.

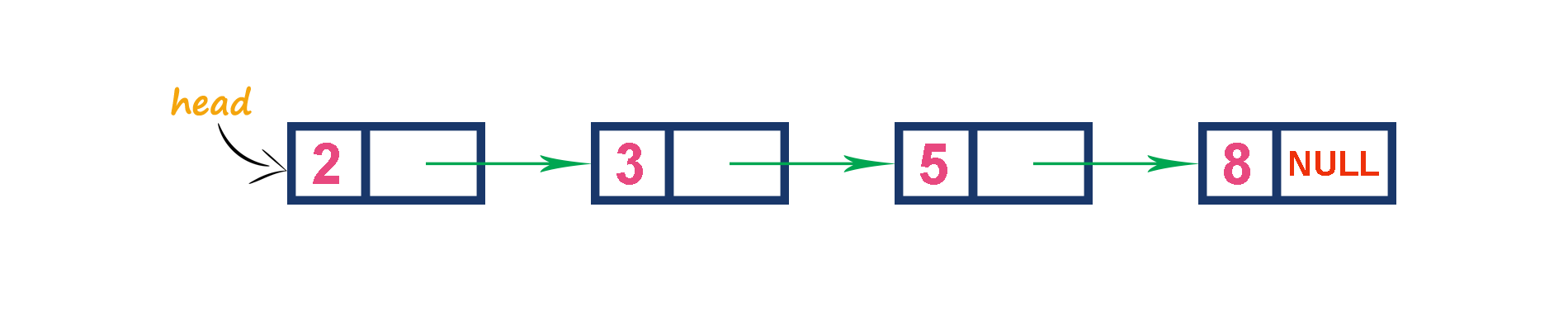
**insert()** - New element is added at a particular position in the increasing order of elements into the queue which requires **O(n)** time complexity as it needs to shift existing elements to maintain increasing order of elements. This insert() operation requires **O(n)** time complexity.

**findMax()** - Finding the maximum element in the queue is very simple becuase maximum element is at the end of the queue. This findMax() operation requires **O(1)** time complexity.

**remove()** - To remove an element from the queue first we need to find the largest element using **findMax()** which requires **O(1)** time complexity, then that element is deleted with constant time complexity **O(1)**. Finally, we need to rearrange the remaining elements to maintain increasing order of elements which requires **O(n)** time complexity. This remove() operation requires **O(1) + O(1) + O(n) ≈ O(n)** time complexity.

**#5. Using Linked List in Increasing Order**

In this representation, we use a single linked list to represent max priority queue. In this representation, elements are inserted according to their value in increasing order and a node with the maximum value is deleted first from the max priority queue.  
For example, assume that elements are inserted in the order of 2, 3, 5 and 8. And they are removed in the order of 8, 5, 3 and 2.



Now, let us analyze each operation according to this representation...

**isEmpty()** - If '**head == NULL**' queue is Empty. This operation requires **O(1)** time complexity which means constant time complexity.

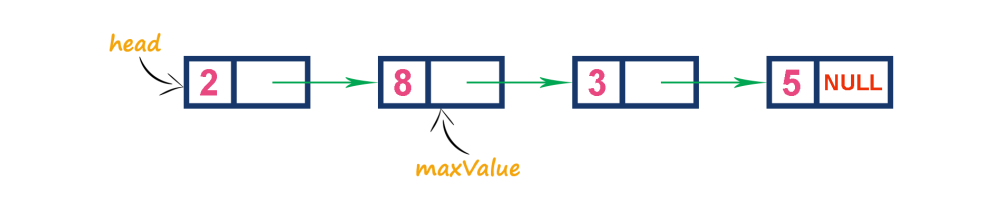
**insert()** - New element is added at a particular position in the increasing order of elements which requires **O(n)** time complexity. This insert() operation requires **O(n)** time complexity.

**findMax()** - Finding the maximum element in the queue is very simple because maximum element is at the end of the queue. This findMax() operation requires **O(1)** time complexity.

**remove()** - Removing an element from the queue is simple because the largest element is last node in the queue. This remove() operation requires **O(1)** time complexity.

**#6. Using Unordered Linked List with reference to node with the maximum value**

In this representation, we use a single linked list to represent max priority queue. We always maintain a reference (maxValue) to the node with the maximum value in the queue. In this representation, elements are inserted according to their arrival and the node with the maximum value is deleted first from the max priority queue.  
For example, assume that elements are inserted in the order of 2, 8, 3 and 5. And they are removed in the order of 8, 5, 3 and 2.



Now, let us analyze each operation according to this representation...

**isEmpty()** - If '**head == NULL**' queue is Empty. This operation requires **O(1)** time complexity which means constant time complexity.

**insert()** - New element is added at end of the queue which requires **O(1)** time complexity. And we need to update maxValue reference with address of largest element in the queue which requires **O(1)** time complexity. This insert() operation requires **O(1)** time complexity.

**findMax()** - Finding the maximum element in the queue is very simple because the address of largest element is stored at maxValue. This findMax() operation requires **O(1)** time complexity.

**remove()** - Removing an element from the queue is deleting the node which is referenced by maxValue which requires **O(1)** time complexity. And then we need to update maxValue reference to new node with maximum value in the queue which requires **O(n) time complexity**. This remove() operation requires **O(n)** time complexity.

**2. Min Priority Queue Representations**

Min Priority Queue is similar to max priority queue except for the removal of maximum element first. We remove minimum element first in the min-priority queue.  
The following operations are performed in Min Priority Queue...

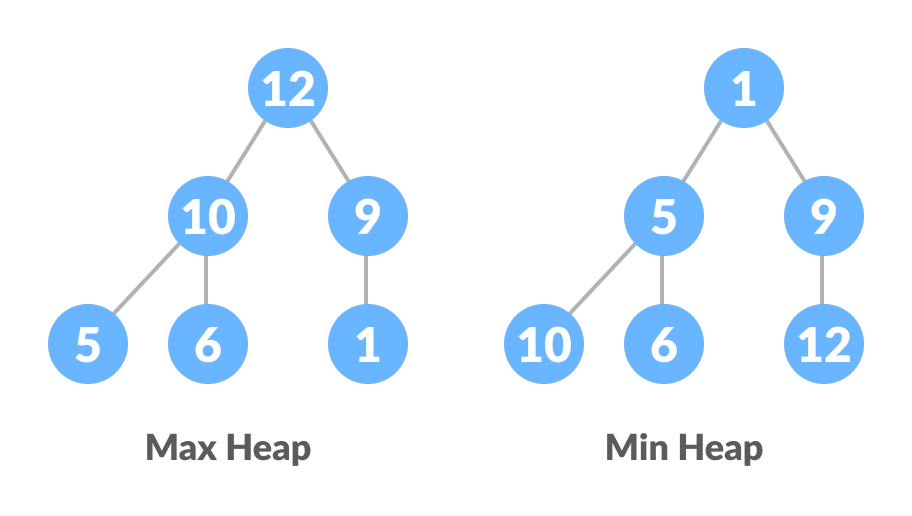
1. **isEmpty()** - Check whether queue is Empty.
2. **insert()** - Inserts a new value into the queue.
3. **findMin()** - Find minimum value in the queue.
4. **remove()** - Delete minimum value from the queue.

# Max Heap Data Structure

Heap data structure is a specialized binary tree-based data structure. Heap is a binary tree with special characteristics. In a heap data structure, nodes are arranged based on their values. A heap data structure sometimes also called as Binary Heap.  
There are two types of heap data structures and they are as follows...

1. **Max Heap**
2. **Min Heap**

The following example diagram shows Max-Heap and Min-Heap.



Every heap data structure has the following properties...

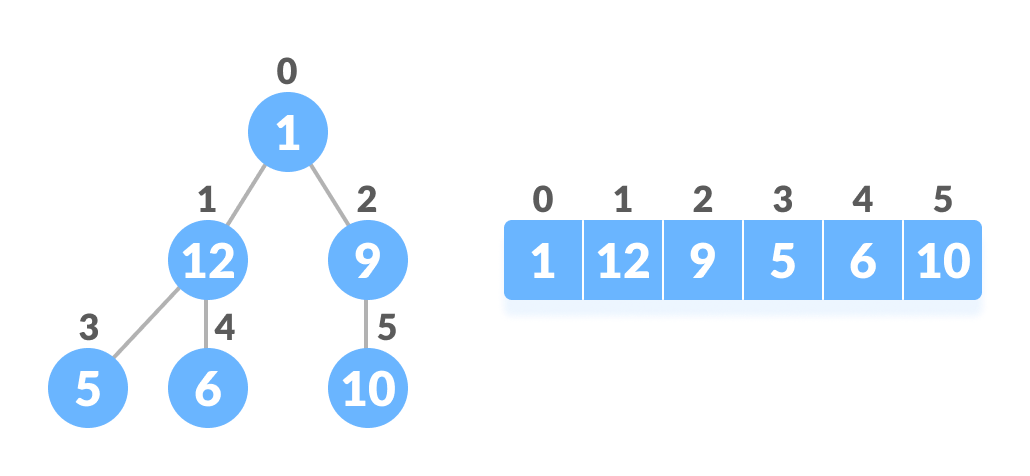
**Property #1 (Ordering):**Nodes must be arranged in an order according to their values based on Max heap or Min heap.

**Property #2 (Structural):**All levels in a heap must be full except the last level and all nodes must be filled from left to right strictly.

**Relationship between Array Indexes and Tree Elements**

A complete binary tree has an interesting property that we can use to find the children and parents of any node.

If the index of any element in the array is i, the element in the index 2i+1 will become the left child and element in 2i+2 index will become the right child. Also, the parent of any element at index i is given by the lower bound of (i-1)/2.



**Relationship between array and heap indices**

Let's test it out,

Left child of 1 (index 0)

= element in (2\*0+1) index

= element in 1 index

= 12

Right child of 1

= element in (2\*0+2) index

= element in 2 index

= 9

Similarly,

Left child of 12 (index 1)

= element in (2\*1+1) index

= element in 3 index

= 5

Right child of 12

= element in (2\*1+2) index

= element in 4 index

= 6

Let us also confirm that the rules hold for finding parent of any node

Parent of 9 (position 2)

= (2-1)/2

= ½

= 0.5

~ 0 index

= 1

Parent of 12 (position 1)

= (1-1)/2

= 0 index

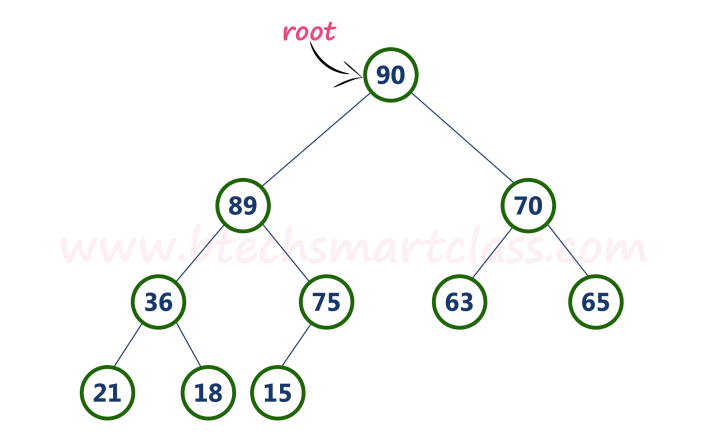
= 1

# Max Heap

Max heap data structure is a specialized complete binary tree data structure. In a max heap, nodes are arranged based on node value.  
Max heap is defined as follows...

**Max heap is a specialized complete binary tree in which every parent node contains greater or equal value than its child nodes.**

##### Example



Above tree is satisfying both Ordering property and Structural property according to the Max Heap data structure.

# Operations on Max Heap

The following operations are performed on a Max heap data structure...

1. **Finding Maximum**
2. **Insertion**
3. **Deletion**

# Finding Maximum Value Operation in Max Heap

Finding the node which has maximum value in a max heap is very simple. In a max heap, the root node has the maximum value than all other nodes. So, directly we can display root node value as the maximum value in max heap.

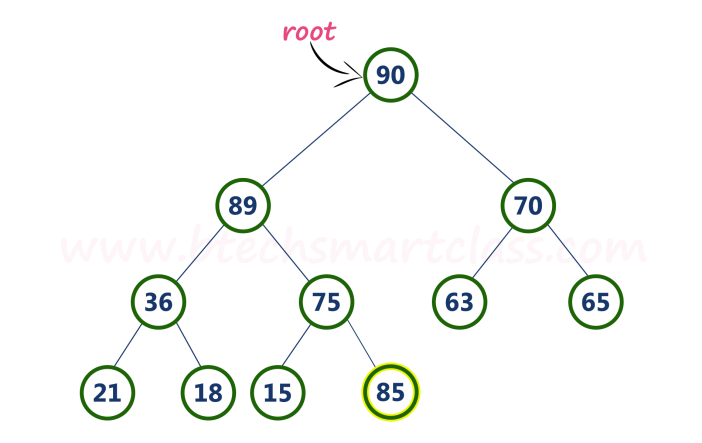
# Insertion Operation in Max Heap

Insertion Operation in max heap is performed as follows...

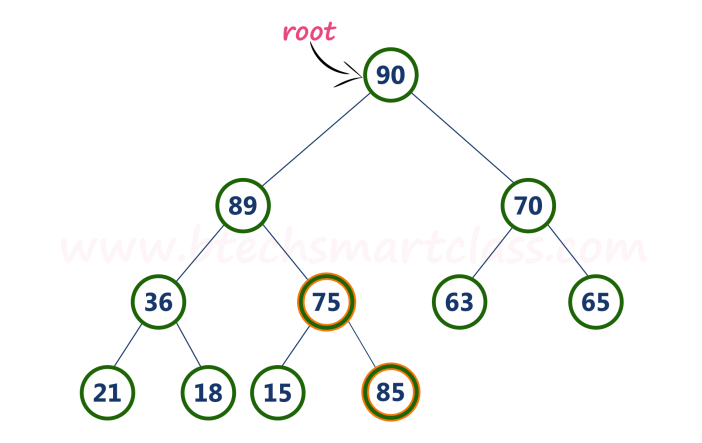
* **Step 1 -**Insert the **newNode** as **last leaf** from left to right.
* **Step 2 -**Compare **newNode value** with its **Parent node**.
* **Step 3 -**If **newNode value is greater** than its parent, then **swap** both of them.
* **Step 4 -**Repeat step 2 and step 3 until newNode value is less than its parent node (or) newNode reaches the root.

**Example**  
Consider the above max heap. **Insert a new node with value 85.**

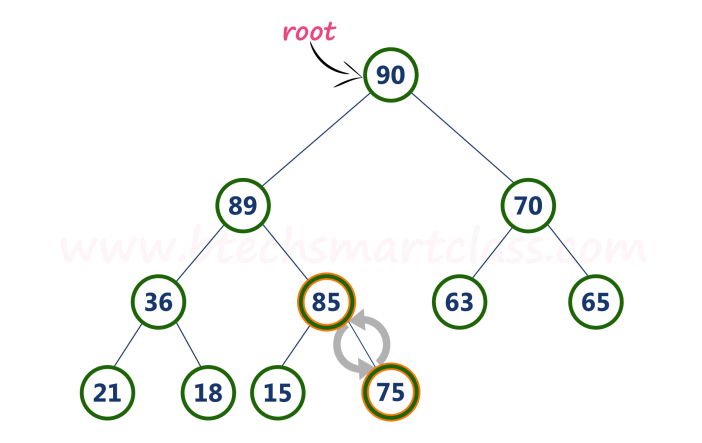
* **Step 1 -**Insert the **newNode** with value 85 as **last leaf** from left to right. That means newNode is added as a right child of node with value 75. After adding max heap is as follows...



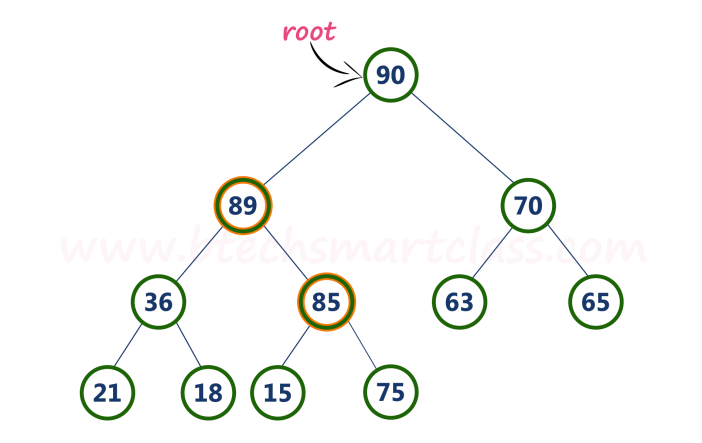
* **Step 2 -**Compare **newNode value (85)** with its **Parent node value (75)**. That means **85 > 75**



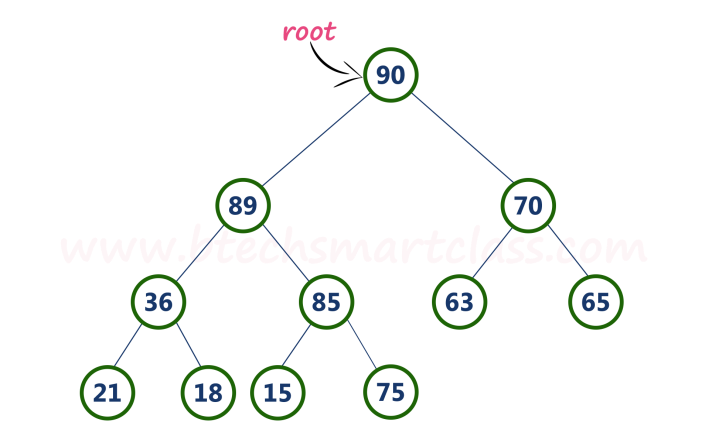
* **Step 3 -**Here **newNode value (85) is greater** than its **parent value (75)**, then **swap** both of them. After swapping, max heap is as follows...



* **Step 4 -**Now, again compare newNode value (85) with its parent node value (89).



Here, newNode value (85) is smaller than its parent node value (89). So, we stop insertion process. Finally, max heap after insertion of a new node with value 85 is as follows...



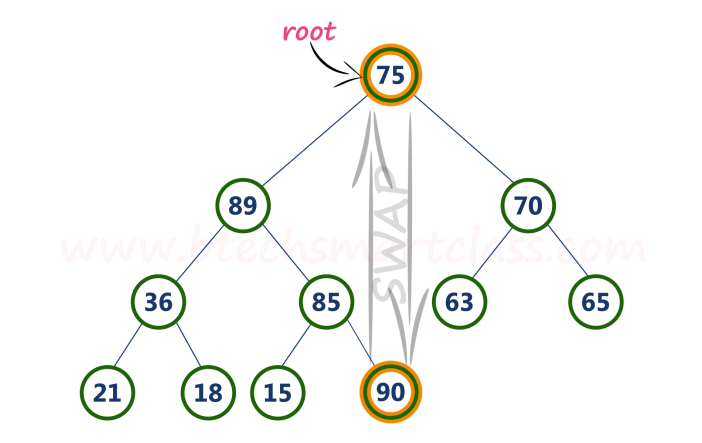
# Deletion Operation in Max Heap

In a max heap, deleting the last node is very simple as it does not disturb max heap properties.  
  
Deleting root node from a max heap is little difficult as it disturbs the max heap properties. We use the following steps to delete the root node from a max heap...

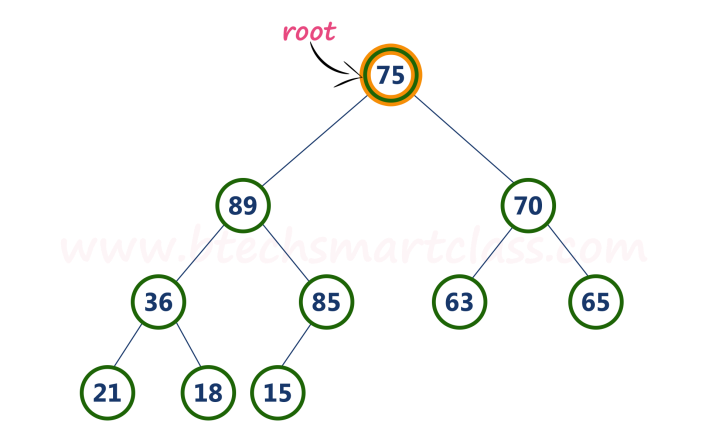
* **Step 1 - Swap** the **root** node with **last** node in max heap
* **Step 2 - Delete** last node.
* **Step 3 -**Now, compare **root value** with its **left child value**.
* **Step 4 -**If **root value is smaller** than its left child, then compare **left child** with its **right sibling**. Else goto **Step 6**
* **Step 5 -**If **left child value is larger** than its **right sibling**, then **swap root** with **left child** otherwise **swap root** with its **right child**.
* **Step 6 -**If **root value is larger** than its left child, then compare **root value** with its **right child** value.
* **Step 7 -**If **root value is smaller** than its **right child**, then **swap root** with **right child** otherwise **stop the process**.
* **Step 8 -**Repeat the same until root node fixes at its exact position.

**Example**  
Consider the above max heap. **Delete root node (90) from the max heap.**

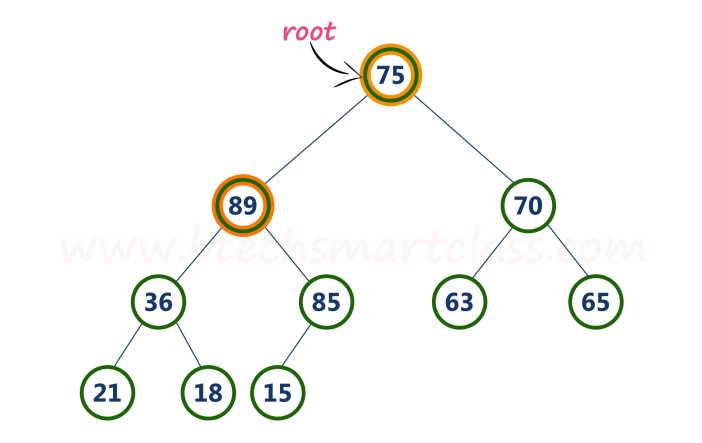
* **Step 1 - Swap** the **root node (90)** with **last node 75** in max heap. After swapping max heap is as follows...



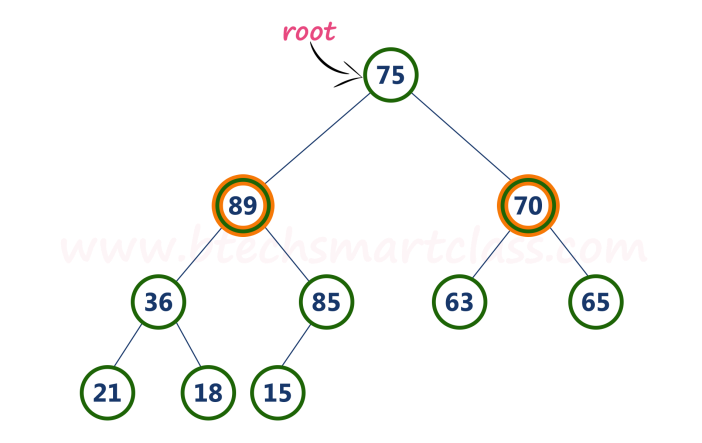
* **Step 2 - Delete** last node. Here the last node is 90. After deleting node with value 90 from heap, max heap is as follows...



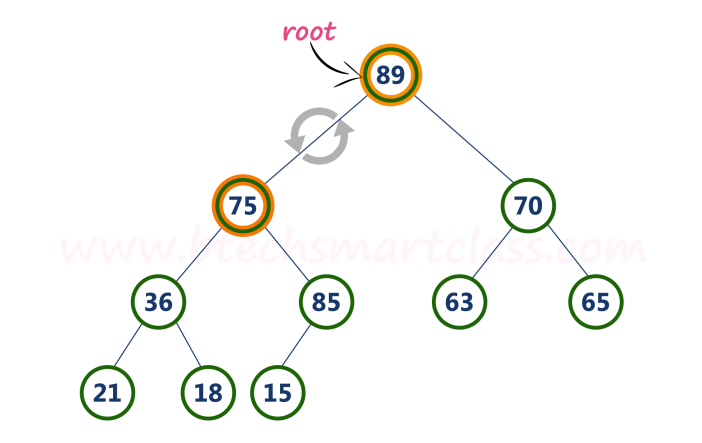
* **Step 3 -**Compare **root node (75)** with its **left child (89)**.



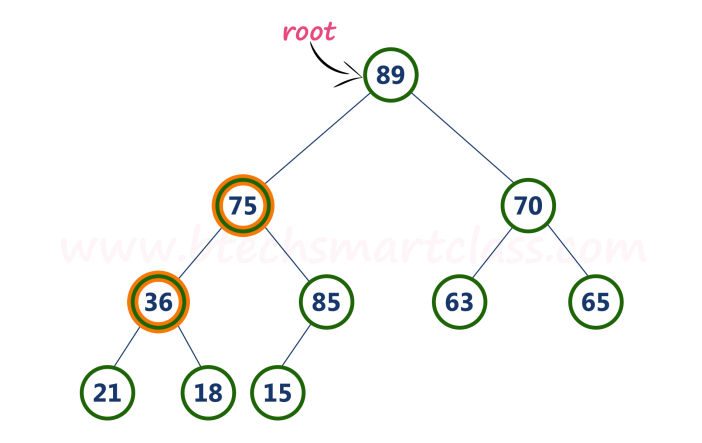
Here, **root value (75) is smaller** than its left child value (89). So, compare left child (89) with its right sibling (70).



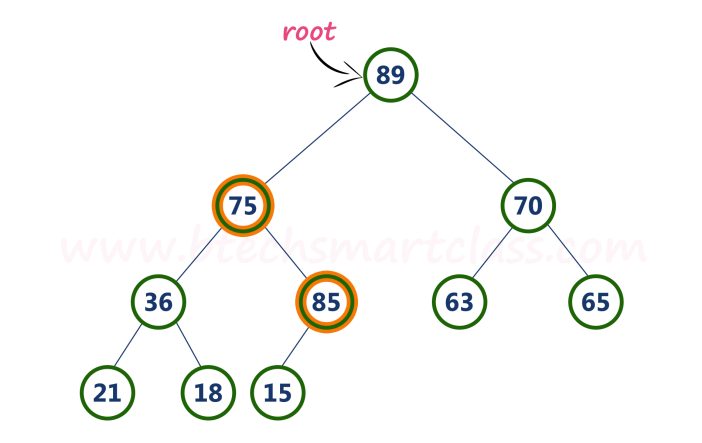
* **Step 4 -**Here, **left child value (89) is larger** than its **right sibling (70)**, So, **swap root (75)** with **left child (89)**.



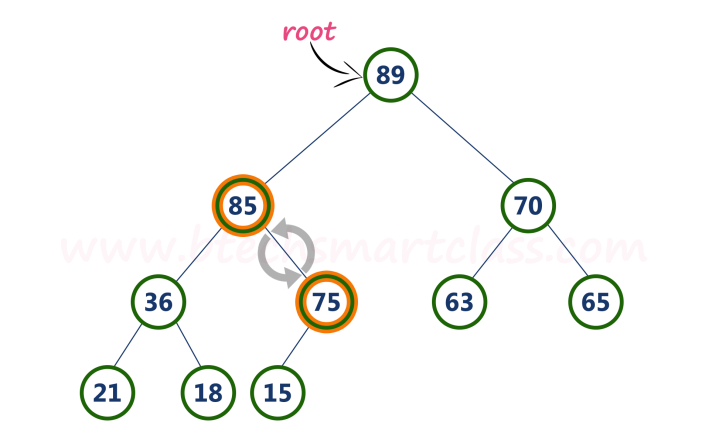
* **Step 5 -**Now, again compare **75** with its **left child (36)**.



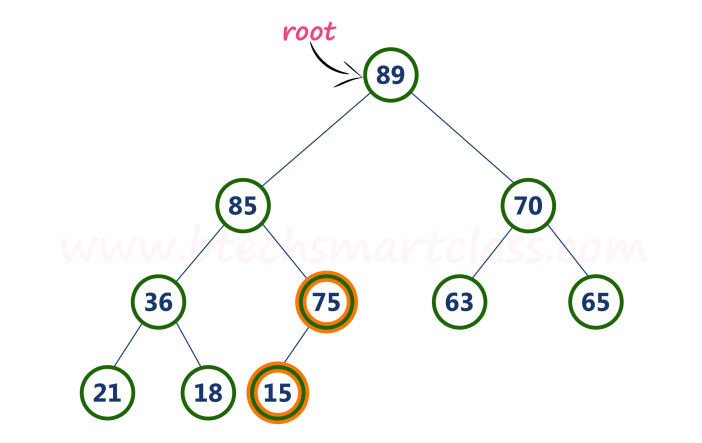
Here, node with value **75** is larger than its left child. So, we compare node **75** with its right child **85**.



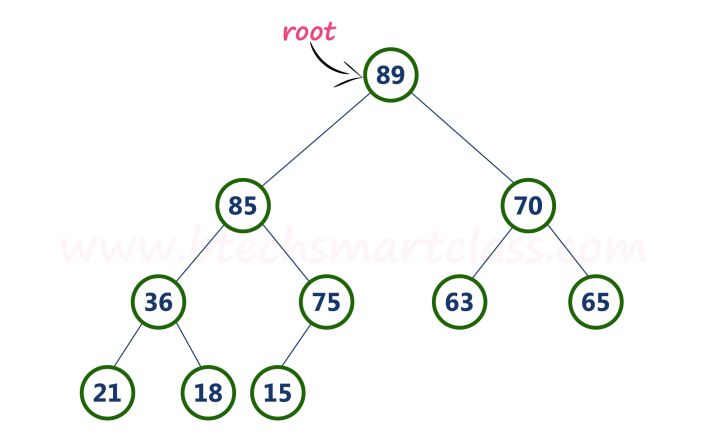
* **Step 6 -**Here, node with value **75** is smaller than its **right child (85)**. So, we swap both of them. After swapping max heap is as follows...



* **Step 7 -**Now, compare node with value **75** with its left child (**15**).



Here, node with value **75** is larger than its left child (**15**) and it does not have right child. So we stop the process.  
  
Finally, max heap after deleting root node (**90**) is as follows...



**Heap Sort Algorithm**

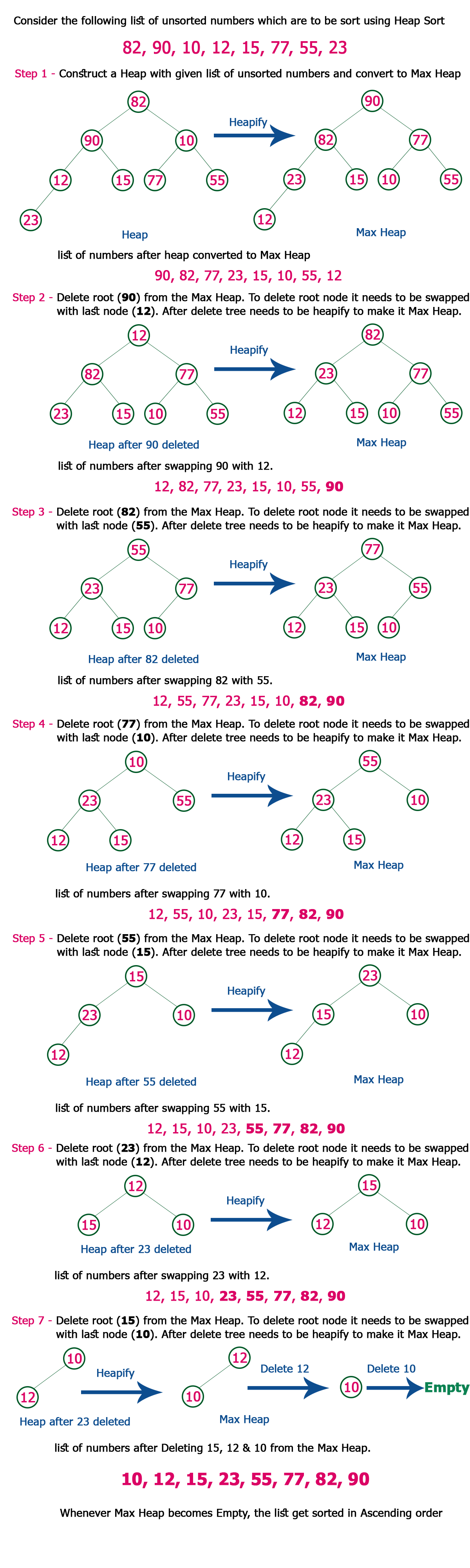
Heap sort is one of the sorting algorithms used to arrange a list of elements in order. Heap sort algorithm uses one of the tree concepts called **Heap Tree**. In this sorting algorithm, we use **Max Heap** to arrange list of elements in Descending order and **Min Heap** to arrange list elements in Ascending order.

**Step by Step Process**

The Heap sort algorithm to arrange a list of elements in ascending order is performed using following steps...

* **Step 1 -**Construct a **Binary Tree** with given list of Elements.
* **Step 2 -**Transform the Binary Tree into **Min Heap.**
* **Step 3 -**Delete the root element from Min Heap using **Heapify** method.
* **Step 4 -**Put the deleted element into the Sorted list.
* **Step 5 -**Repeat the same until Min Heap becomes empty.
* **Step 6 -**Display the sorted list.

**Example**



**Complexity of the Heap Sort Algorithm**

To sort an unsorted list with **'n'** number of elements, following are the complexities...

**Worst Case : O(n log n)**  
**Best Case : O(n log n)**  
**Average Case : O(n log n)**

**Heap Sort**

void adjust(int list[], int root, int n)

{

int child, rootkey;

int temp;

temp=list[root];

rootkey=list[root].key;

child=2\*root;

while (child <= n)

{

if ((child < n) && (list[child].key < list[child+1].key))

child++;

if (rootkey > list[child].key)

break;

else

{

list[child/2] = list[child];

child \*= 2;

}

}

list[child/2] = temp;

}

void heapsort(int list[], int n)

{

int i, j;

int temp;

for (i=n/2; i>0; i--)

adjust(list, i, n);

for (i=n-1; i>0; i--)

{

SWAP(list[1], list[i+1], temp);

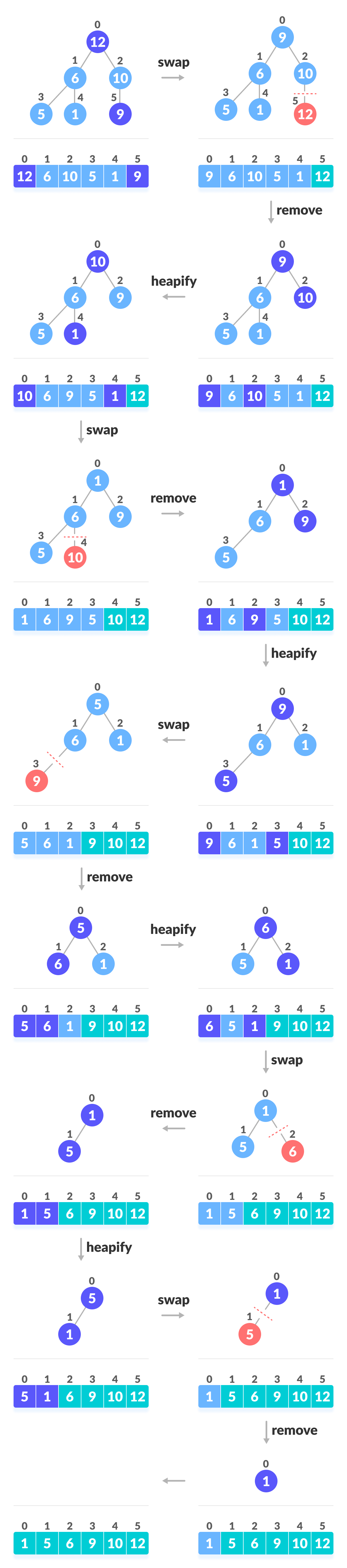
adjust(list, 1, i);

}

}

## How Heap Sort Works?

1. Since the tree satisfies Max-Heap property, then the largest item is stored at the root node.
2. **Swap:** Remove the root element and put at the end of the array (n-1 position) Put the last item of the tree (heap) at the vacant place.
3. **Remove:** Reduce the size of the heap by 1.
4. **Heapify:** Heapify the root element again so that we have the highest element at root.
5. The process is repeated until all the items of the list are sorted.



// Heap Sort in C

#include <stdio.h>

void heapify(int arr[], int n, int i)

{

// Find largest among root, left child and right child

int largest = i;

int left = 2 \* i + 1;

int right = 2 \* i + 2;

if (left < n && arr[left] > arr[largest])

largest = left;

if (right < n && arr[right] > arr[largest])

largest = right;

// Swap and continue heapifying if root is not largest

if (largest != i)

{

int temp = arr[i];

arr[i] = arr[largest];

arr[largest] = temp;

heapify(arr, n, largest);

}

}

// Main function to do heap sort

void heapSort(int arr[], int n) {

// Build max heap

for (int i = (n / 2) - 1; i >= 0; i--)

heapify(arr, n, i);

// Heap sort

for (int i = n - 1; i >= 0; i--) {

int temp = arr[i];

arr[i] = arr[0];

arr[0] = temp;

// Heapify root element to get highest element at root again

heapify(arr, i, 0);

}

}

int main()

{

int arr[10],n,i;

printf(“Enter the number of elements”);

scanf(“%d”,&n);

printf(“Enter the values”);

for (i = 0; i < n; ++i)

scanf(“%d”,&arr[i]);

heapSort(arr, n);

printf("Sorted array is \n");

for (i = 0; i < n; ++i)

{

printf("%d ", arr[i]);

printf("\n");

}

}