

Data Structures

IT 205

Dr. Manish Khare



Lecture – 29
10-Apr-2018

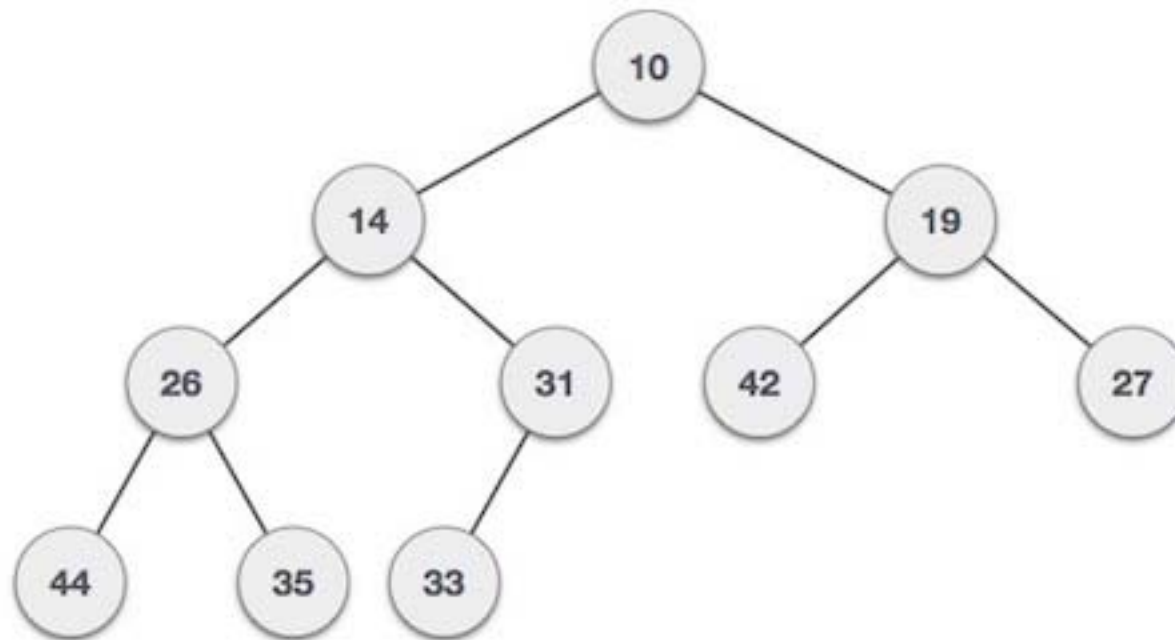
Heap Tree

- Heap is a special case of balanced binary tree data structure where the root-node key is compared with its children and arranged accordingly. If α has child node β then –

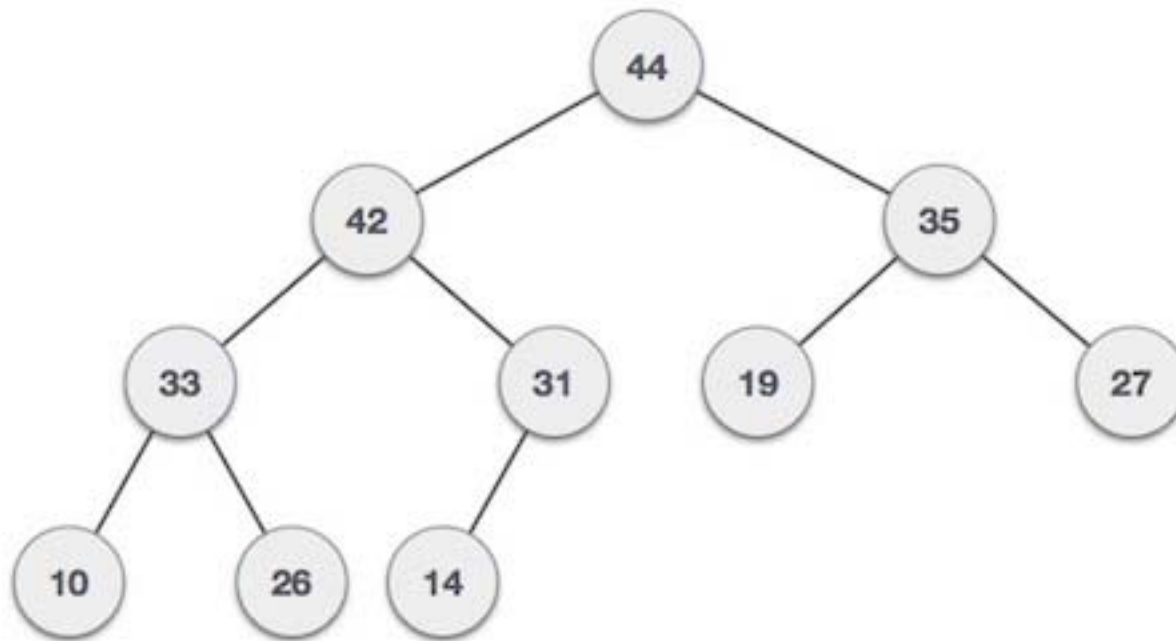
$$\text{key}(\alpha) \geq \text{key}(\beta)$$

- Based on this criteria, a heap can be of two types –
 - Min Heap
 - Max Heap

➤ **Min-Heap** – Where the value of the root node is less than or equal to either of its children.



➤ **Max-Heap** – Where the value of the root node is greater than or equal to either of its children.





➤ Every heap data structure has the following properties...

- **Property #1 (Ordering):** Nodes must be arranged in a order according to values based on Max heap or Min heap.
- **Property #2 (Structural):** All levels in a heap must full, except last level and nodes must be filled from left to right strictly.

Operations on Heap


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

- **Finding Maximum/Minimum**
- **Insertion**
- **Deletion**

Finding Max/Min Value Operation in Max/Min Heap

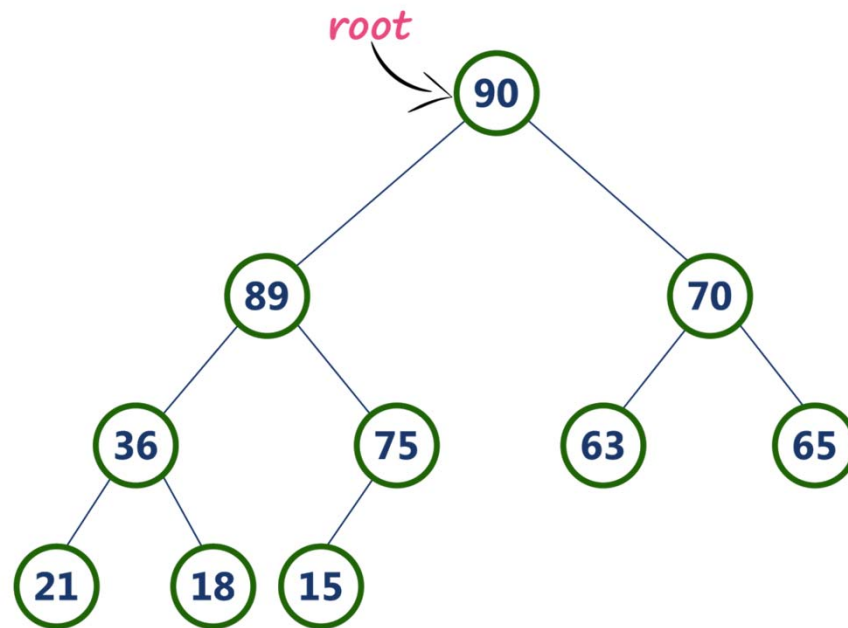
- Finding the node which has maximum/Minimum value in a max/min heap is very simple.
- In max/min heap, the root node has the maximum/minimum value than all other nodes in the max/min heap.
- So, directly we can display root node value as maximum/minimum value in max/min heap.

Insertion Operation in Max Heap

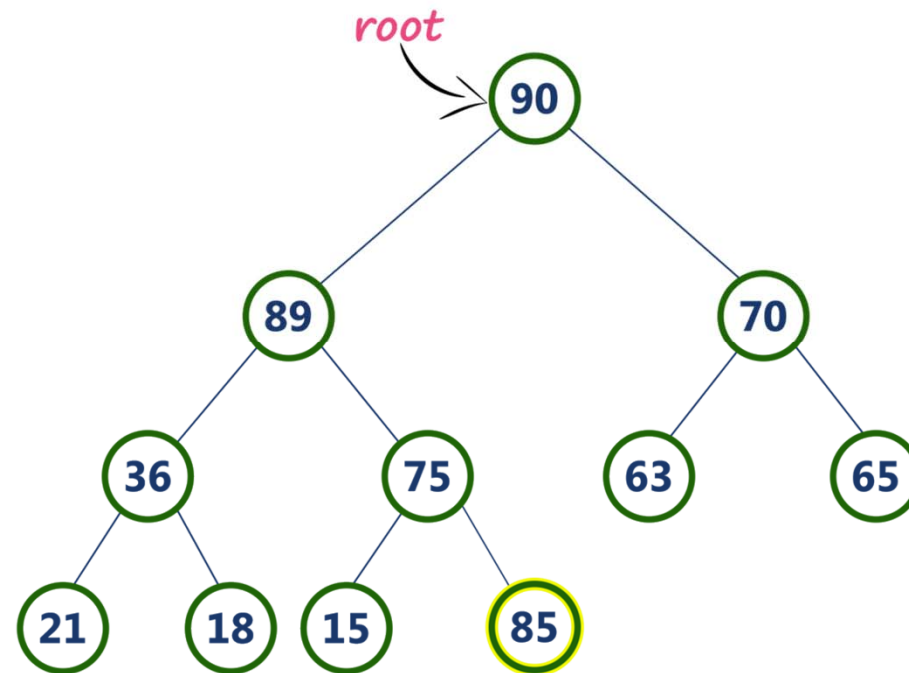
- 
- Insertion Operation in max heap is performed as follows...
- **Step 1:** Insert the **newNode** as **last leaf** from left to right.
 - **Step 2:** Compare **newNode value** with its **Parent node**.
 - **Step 3:** If **newNode value is greater** than its parent, then **swap** both of them.
 - **Step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

Insertion Operation in Max Heap - Example

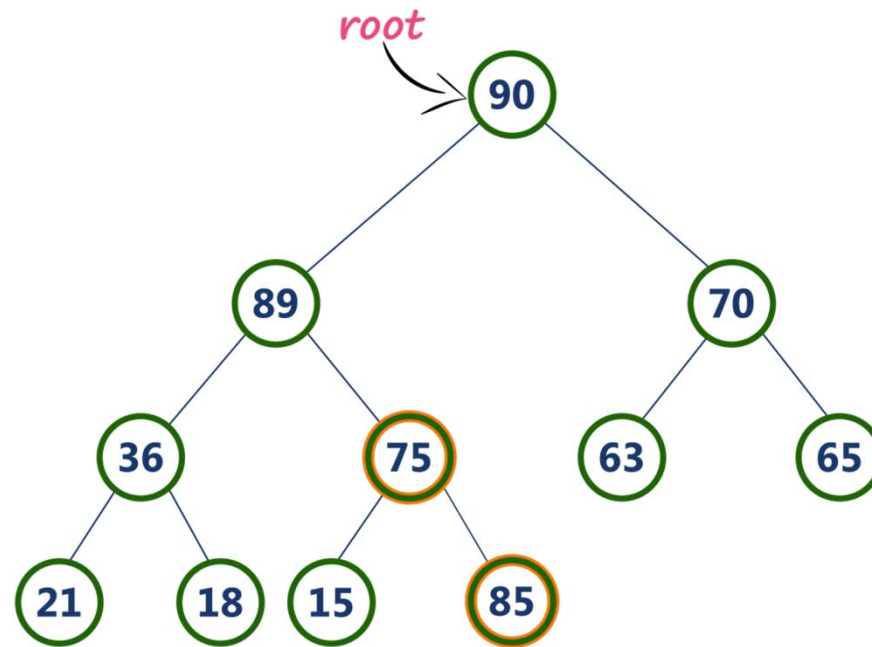
- Consider the following max heap. Insert a new node with value 85.



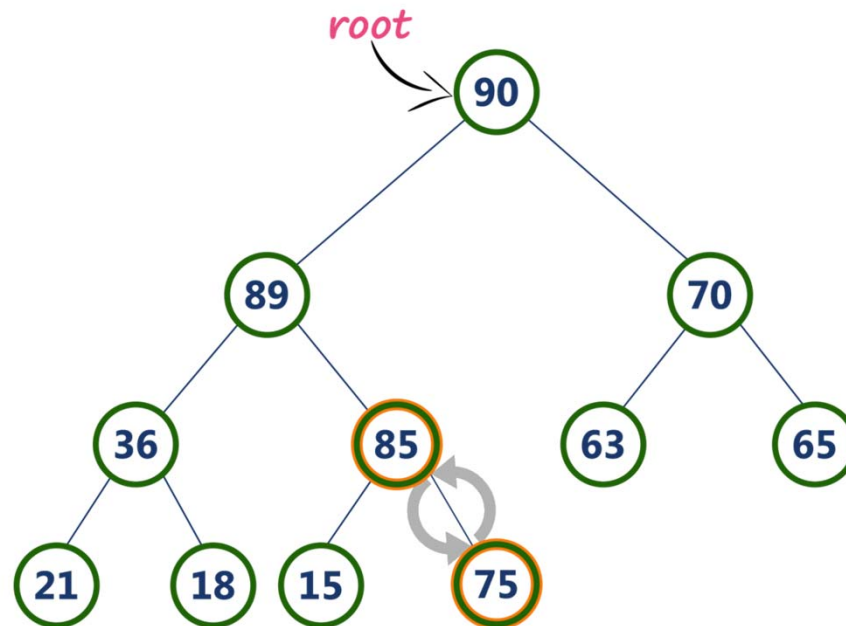
➤ **Step 1:** Insert the **newNode** with value 85 as **last leaf** from left to right. That means newNode is added as a right child of node with value 75. After adding max heap is as follows...



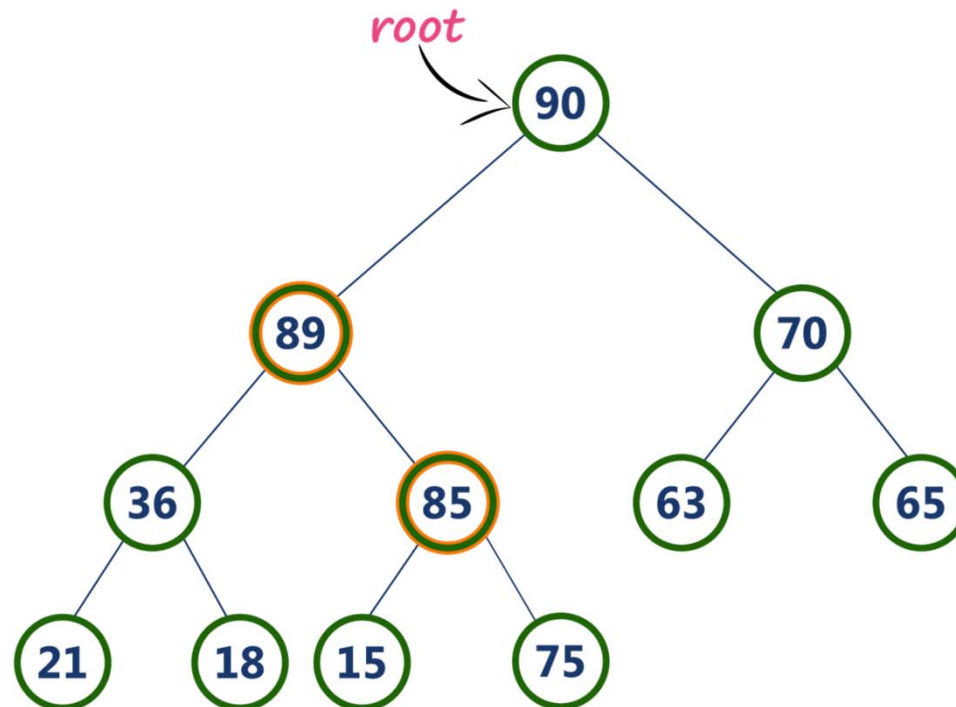
- **Step 2:** Compare **newNode** value (85) with its **Parent** node value (75). That means $85 > 75$



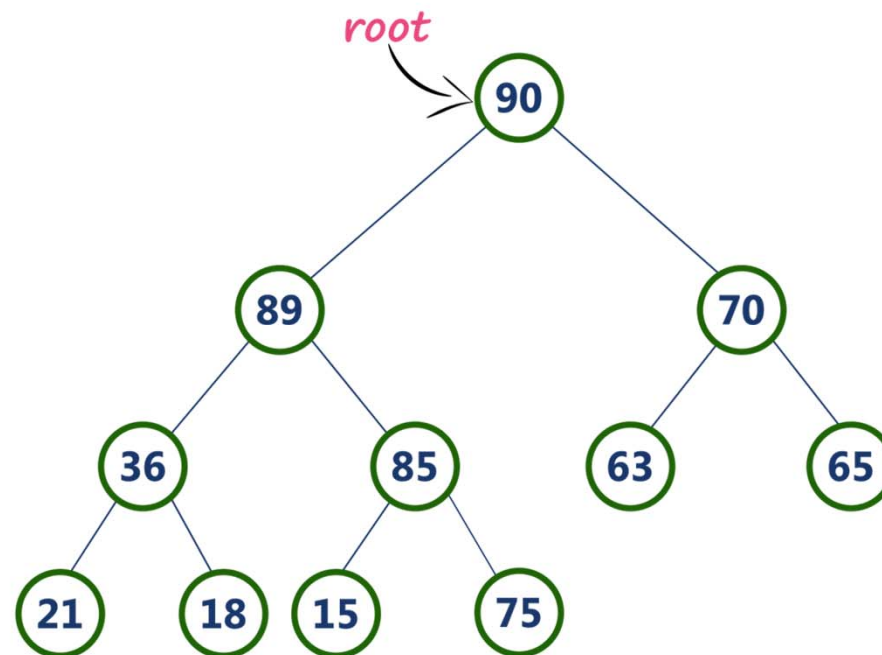
- **Step 3:** Here **newNode** value (**85**) is **greater** than its **parent** value (**75**), then **swap** both of them. After swapping, max heap is as follows...


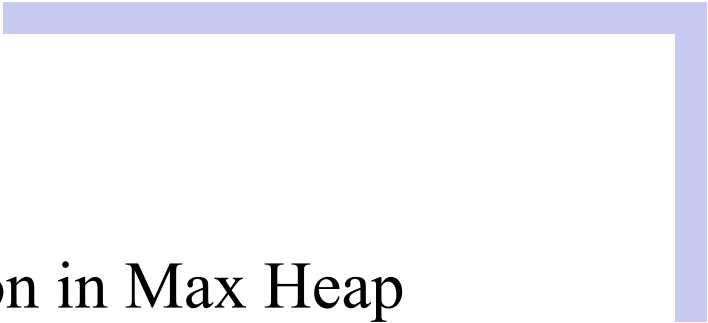


➤ **Step 4:** Now, again compare newNode value (85) with its parent node value (89).




➤ 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...





Animation video for insertion operation in Max Heap

Input 35 33 42 10 14 19 27 44 26 31



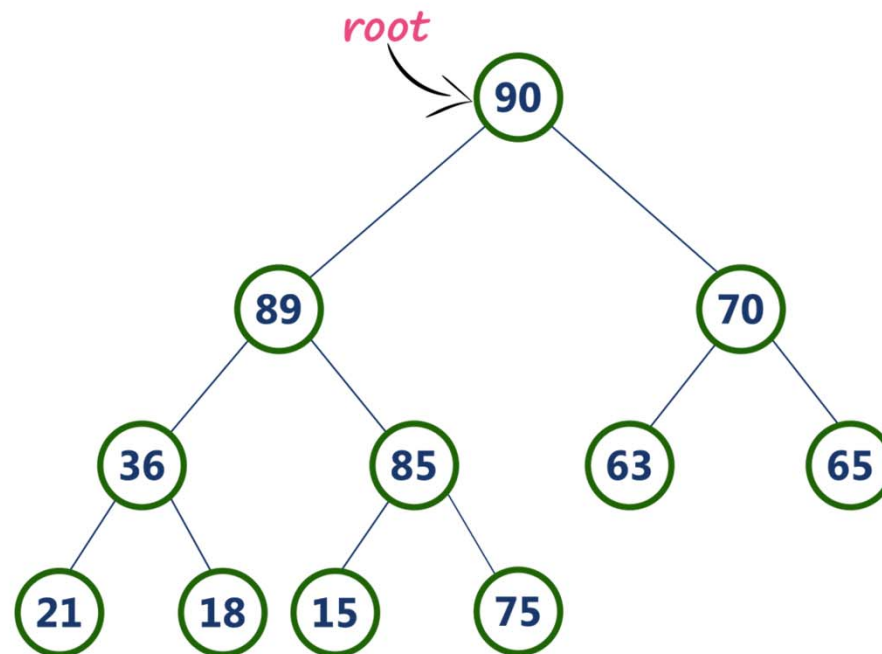
➤ **Note** – In Min Heap construction algorithm, we expect the value of the parent node to be less than that of the child node.

Deletion Operation in Max Heap

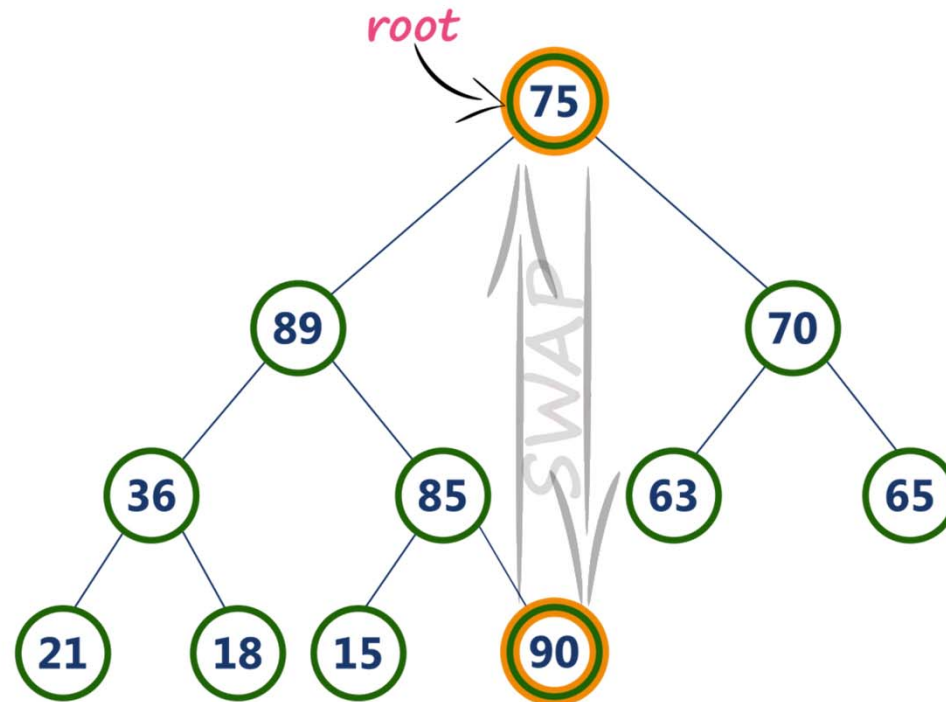
- Deletion in Max (or Min) Heap always happens at the root to remove the Maximum (or minimum) value.

- Deleting root node from a max heap is quite difficult as it disturbs the max heap properties. We use the following steps to delete 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 is fixed at its exact position.

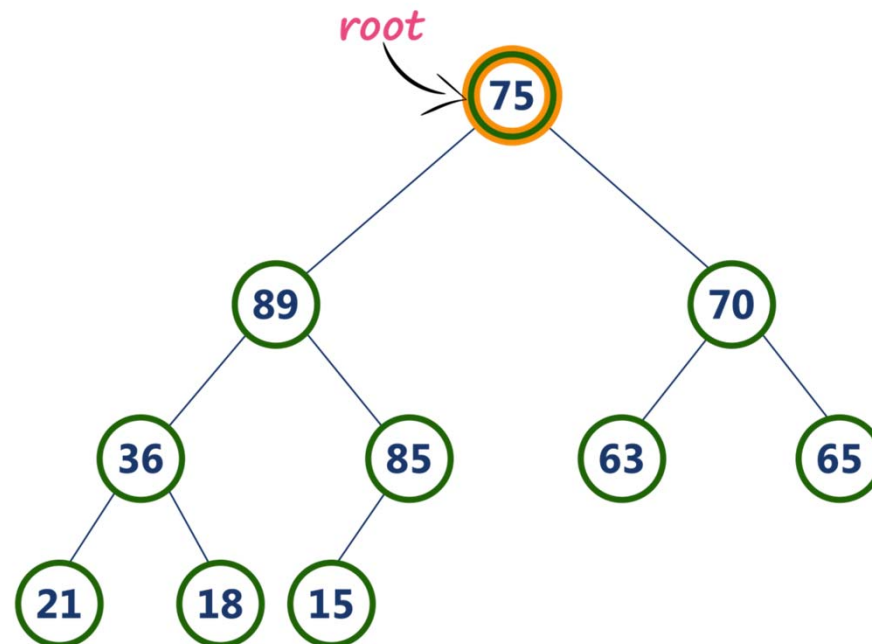
- Consider the following max heap. **Delete root node (90) from the max heap.**



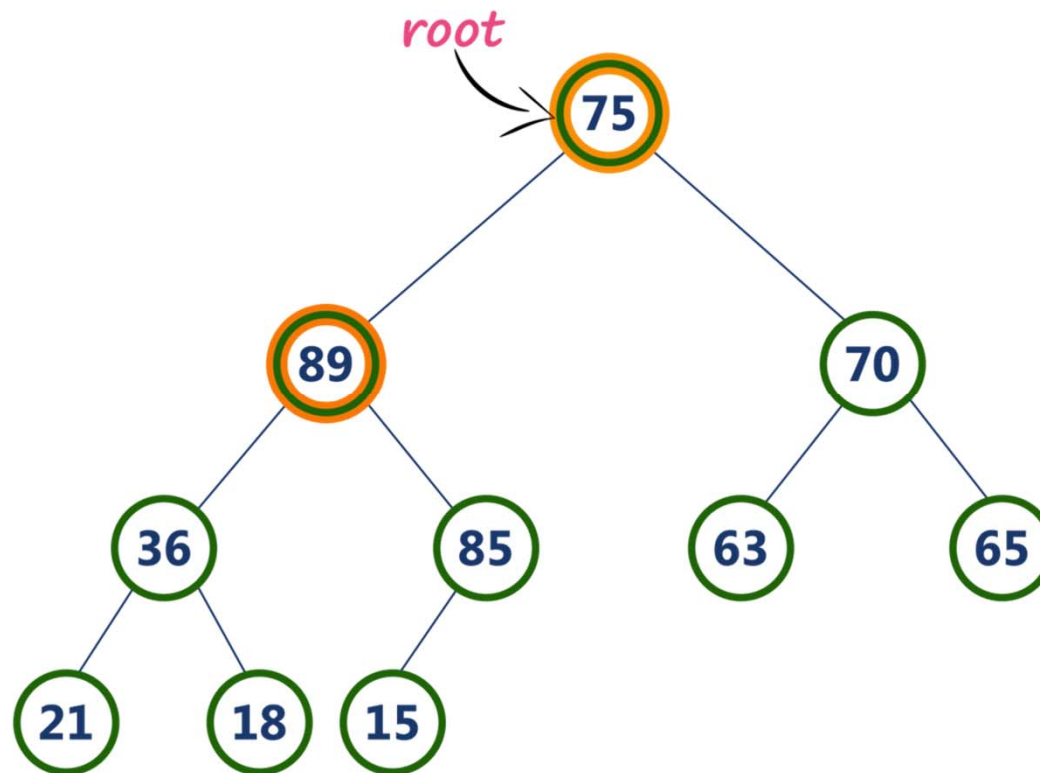
- **Step 1: Swap the root node (90) with last node 75** in max heap After swapping max heap is as follows...



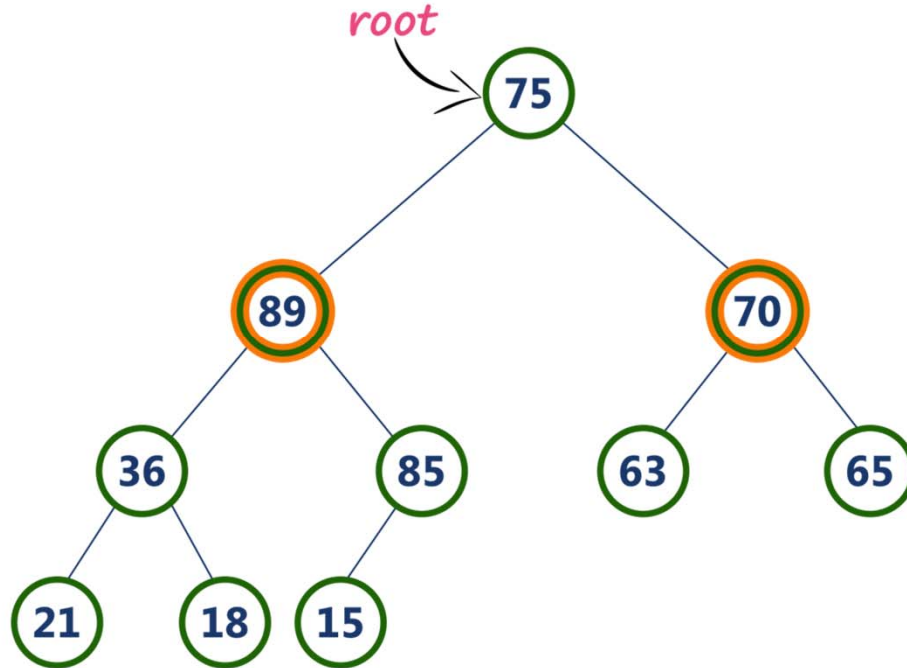
➤ **Step 2: Delete** last node. Here node with value 90. After deleting node with value 90 from heap, max heap is as follows...



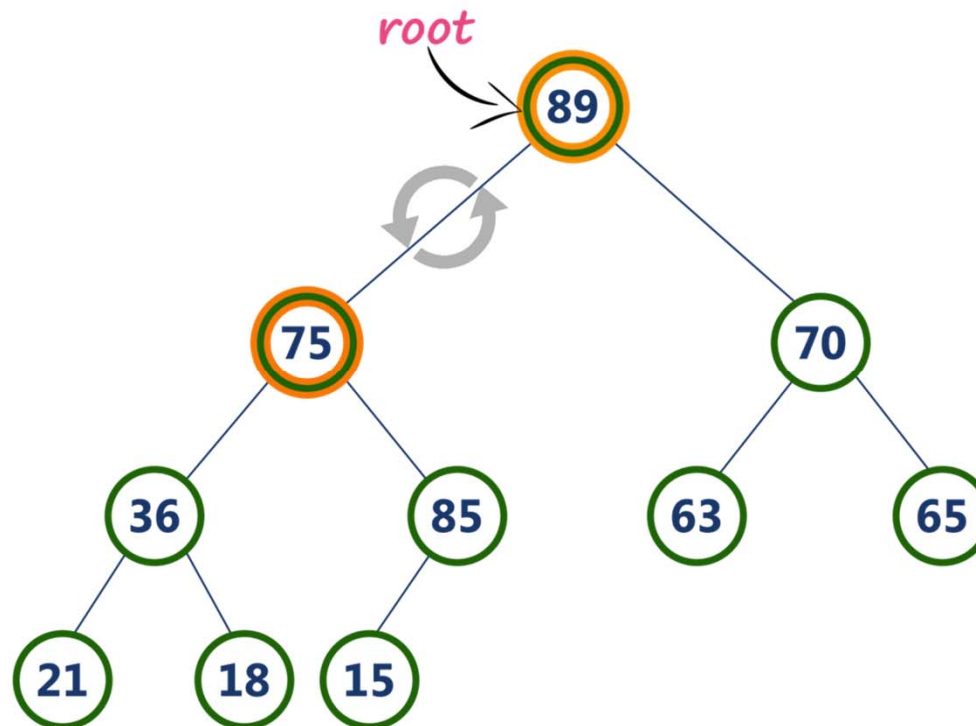
➤ **Step 3: Compare root node (75) with its left child (89).**



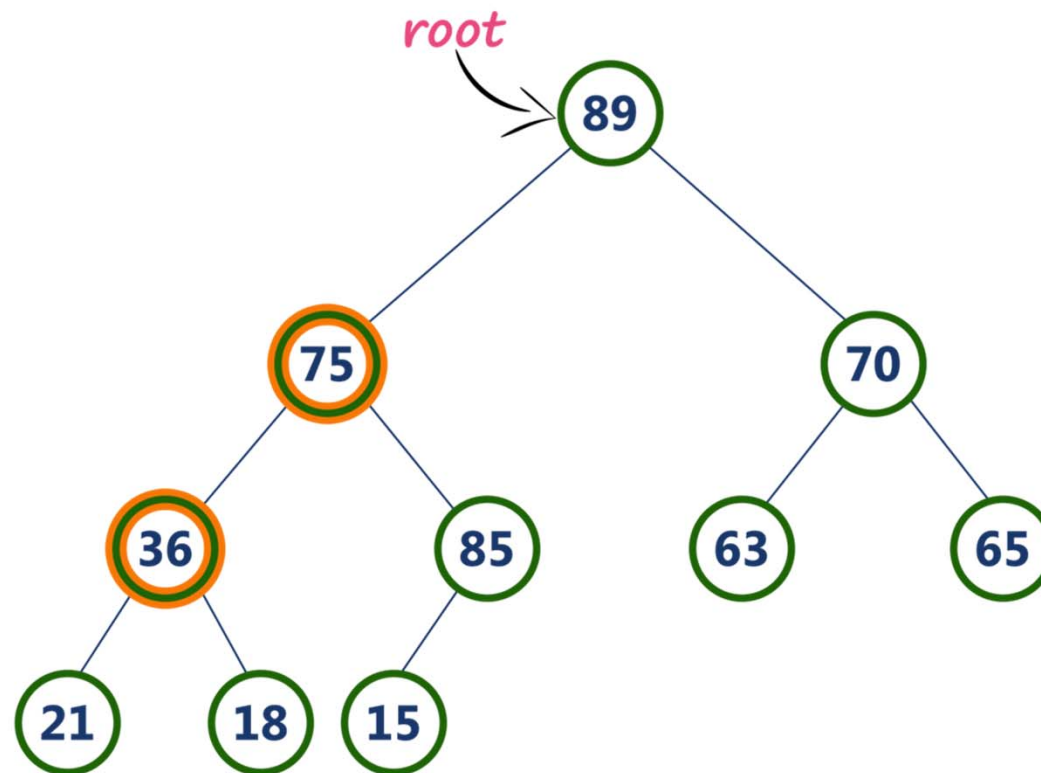
- Here, **root value (75)** is **smaller** than its left child value (89).
So, compare left child (89) with its right sibling (70).



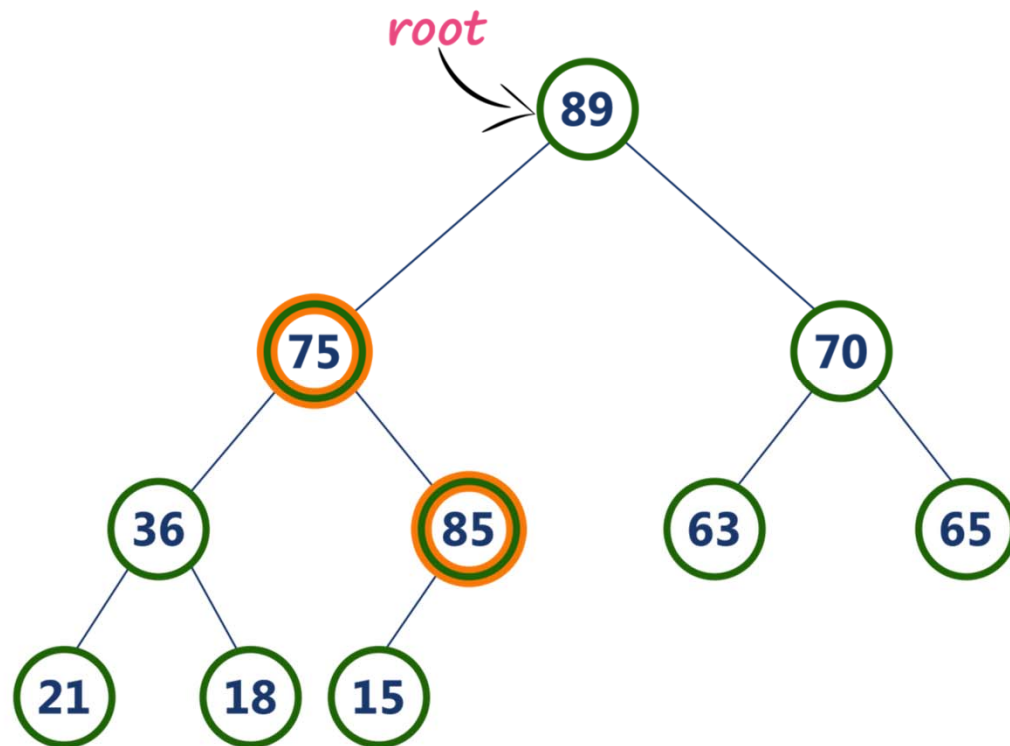
- **Step 4:** Here, left child value (89) is larger than its right sibling (70), So, swap root (75) with left child (89).



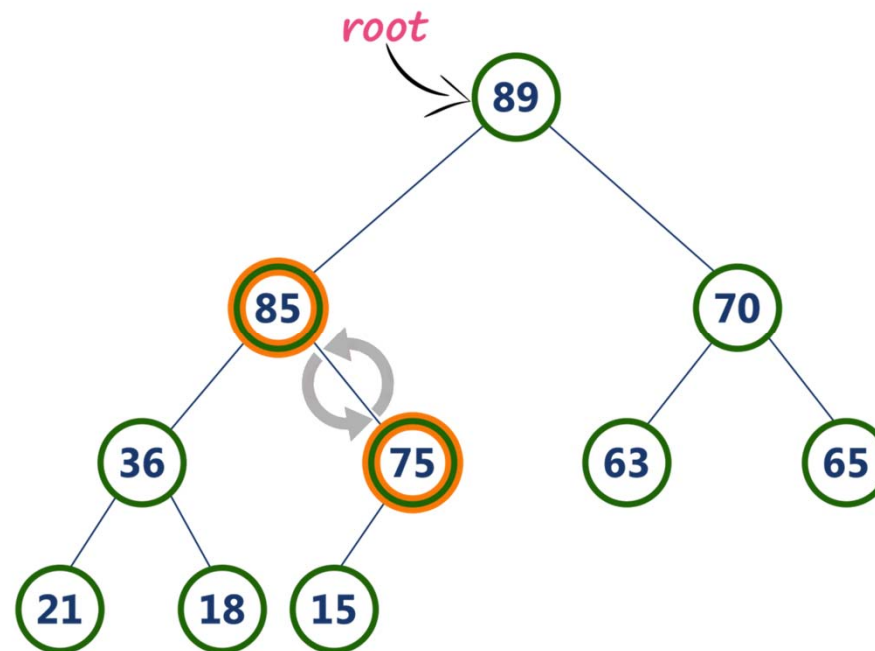
➤ **Step 5:** Now, again compare **75** with its **left child (36)**.



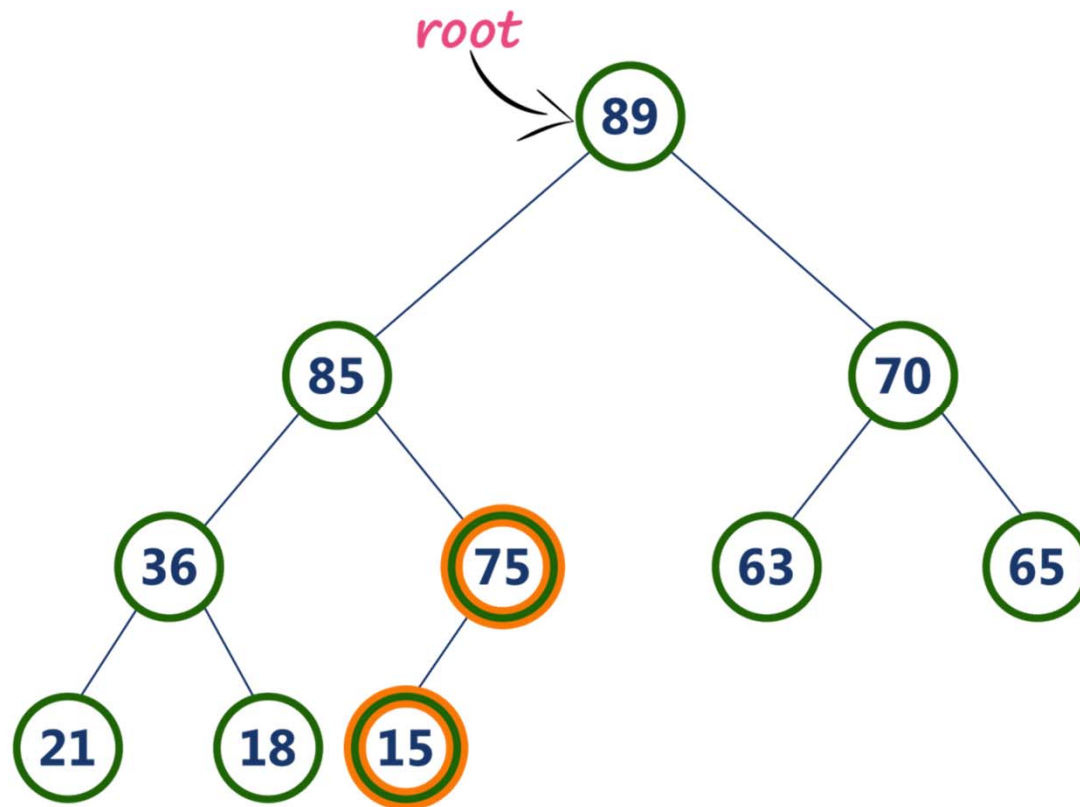
- Here, node with value **75** is larger than its left child. So, we compare node with value **75** is compared with its right child **85**.



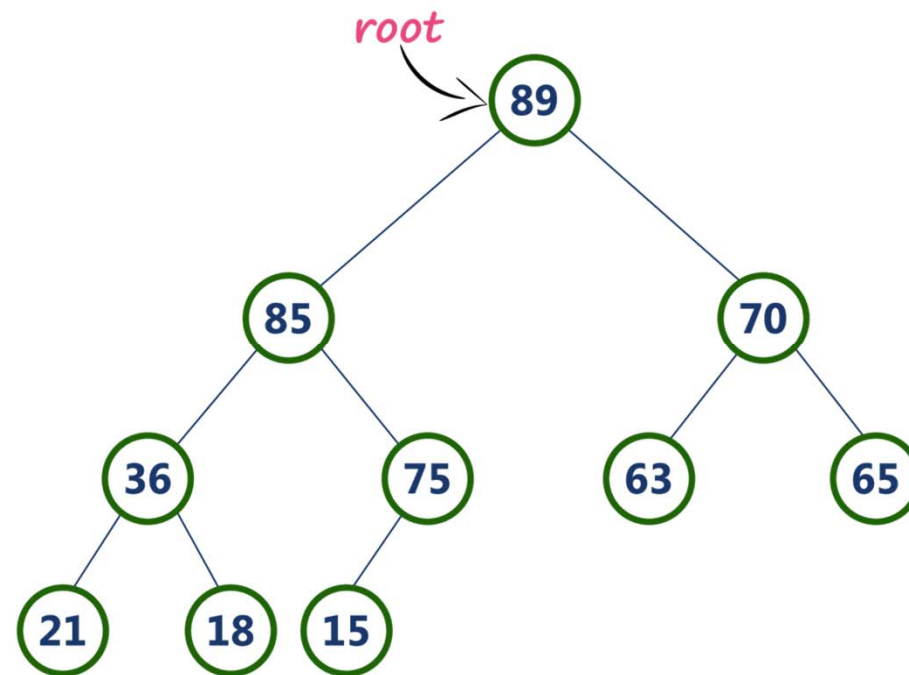
➤ **Step 6:** Here, node with value **75** is smaller than its **right child (85)**. So, we swap both of them. After swapping max heap is as follows...



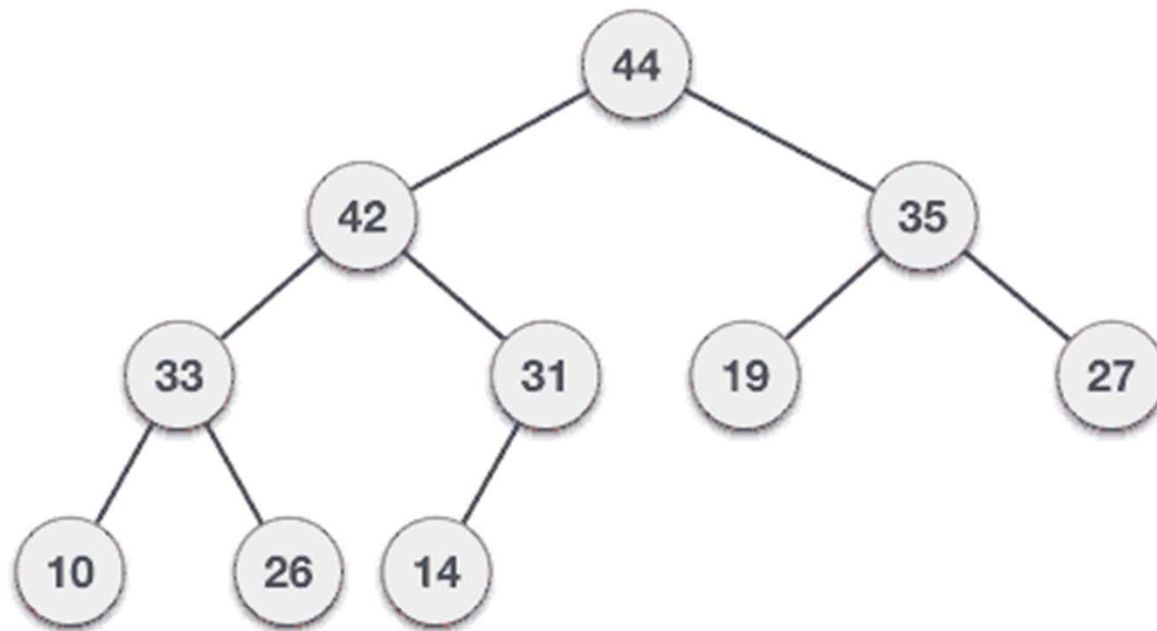
➤ **Step 7:** Now, compare node with value **75** with its left child (**15**).







- 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...



➤ Animation video for deletion operation in Max Heap





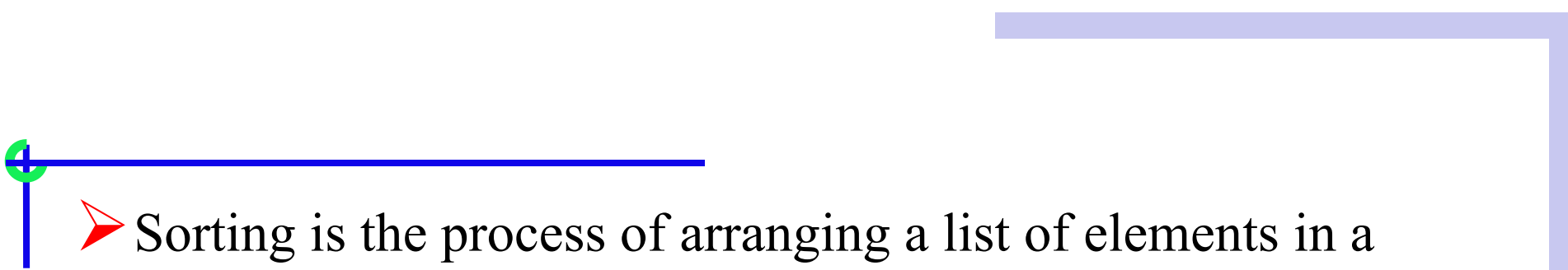
➤ **Note** – In Min Heap deletion algorithm, we expect the value of the parent node to be less than that of the child node.



Sorting Algorithms

Sorting







➤ Sorting is the process of arranging a list of elements in a particular order (Ascending or Descending).

➤ Sorting is categorized as

- Internal Sorting
- External Sorting

➤ Internal sorting means we are arranging the numbers within the array only which is in computer primary memory.

➤ External sorting is the sorting of numbers from the external file by reading it from secondary memory.

- 
- 
- Why do we need to sort data ?:
 - to arrange names in alphabetical order
 - arrange students by grade, etc.
 - preliminary step to searching data.
 - Basic steps involved:
 - compare two items
 - swap the two items or copy one item.



➤ Type of Sorting algorithm

- Bubble Sort
- Selection Sort
- Insertion Sort
- Quick Sort
- Radix Sort
- Merge Sort
- Heap Sort
- Shell Sort

Bubble Sort

- Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.
- This algorithm is not suitable for large data sets

Bubble Sort

- In bubble sort, each element is compared with its adjacent element. If the first element is larger than the second one then the position of the elements are interchanged, otherwise it is not changed. Then the next element is compared with its adjacent element and the same process is repeated for all the elements in the array. At last largest element is at last position.
- During the next all passes the same process is repeated until no more elements are left for comparison.

Function for Bubble Sort

```
void bubbleSort(int arr[], int n)
{
    int i, j;
    for (i = 0; i < n-1; i++)

        // Last i elements are already in place
        for (j = 0; j < n-i-1; j++)
            if (arr[j] > arr[j+1])
                swap(&arr[j], &arr[j+1]);
}
```

Bubble Sort - Example 1

➤ **Example:** (5 1 4 2 8)

First Pass:

(5 1 4 2 8) \rightarrow (1 5 4 2 8), Here, algorithm compares the first two elements, and swaps since $5 > 1$.

(1 5 4 2 8) \rightarrow (1 4 5 2 8), Swap since $5 > 4$

(1 4 5 2 8) \rightarrow (1 4 2 5 8), Swap since $5 > 2$

(1 4 2 5 8) \rightarrow (1 4 2 5 8), Now, since these elements are already in order ($8 > 5$), algorithm does not swap them.

➤ **Second Pass:**

(1 4 2 5 8) \rightarrow (1 4 2 5 8)

(1 4 2 5 8) \rightarrow (1 2 4 5 8), Swap since $4 > 2$

(1 2 4 5 8) \rightarrow (1 2 4 5 8)

(1 2 4 5 8) \rightarrow (1 2 4 5 8)

Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

➤ **Third Pass:**

(1 2 4 5 8) \rightarrow (1 2 4 5 8)

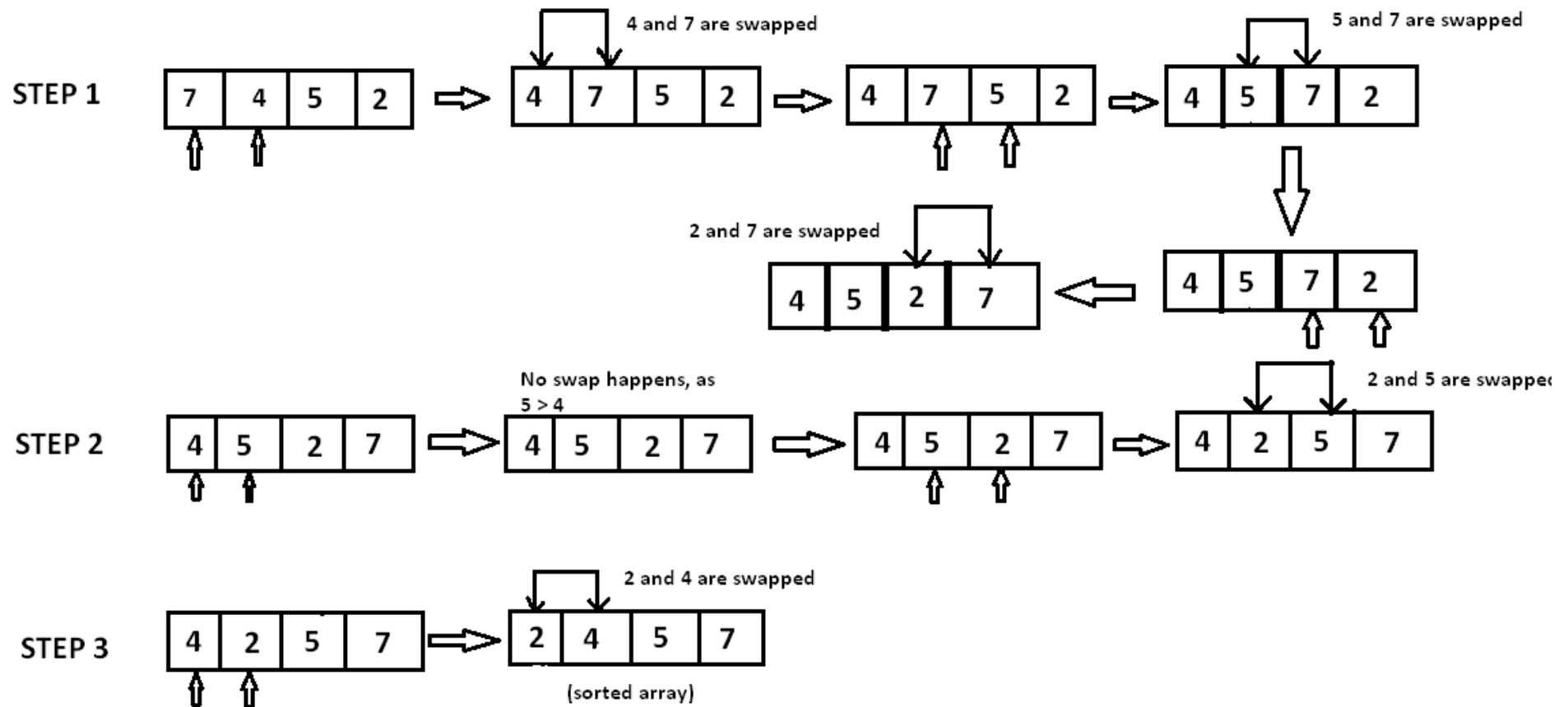
(1 2 4 5 8) \rightarrow (1 2 4 5 8)

(1 2 4 5 8) \rightarrow (1 2 4 5 8)

(1 2 4 5 8) \rightarrow (1 2 4 5 8)

Bubble Sort - Example 2

Example: { 7, 4, 5, 2 }



Bubble Sort - Exercise

Sort following array using Bubble Sort.

1. {54,26,93,17,77,31,44,55,20}
2. {56,47,41,35,22,11}

Selection Sort

- Selection Sort algorithm is used to arrange a list of elements in a particular order (Ascending or Descending).
- In selection sort, the first element in the list is selected and it is compared repeatedly with remaining all the elements in the list. If any element is smaller than the selected element (for Ascending order), then both are swapped.
- Then we select the element at second position in the list and it is compared with remaining all elements in the list. If any element is smaller than the selected element, then both are swapped. This procedure is repeated till the entire list is sorted.



The selection sort algorithm is performed using following steps...

- **Step 1:** Select the first element of the list (i.e., Element at first position in the list).
- **Step 2:** Compare the selected element with all other elements in the list.
- **Step 3:** For every comparsion, if any element is smaller than selected element (for Ascending order), then these two are swapped.
- **Step 4:** Repeat the same procedure with next position in the list till the entire list is sorted.

Function for Selection Sort

//Selection sort logic

```
for(i=0; i<size; i++){  
    for(j=i+1; j<size; j++){  
        if(list[i] > list[j])  
        {  
            temp=list[i];  
            list[i]=list[j];  
            list[j]=temp;  
        }  
    }  
}
```

Selection Sort – Example 1

Consider the following unsorted list of elements...

15	20	10	30	50	18	5	45
----	----	----	----	----	----	---	----

Iteration #1

Select the first position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

15	20	10	30	50	18	5	45
----	----	----	----	----	----	---	----

$15 > 20$
FALSE

15	20	10	30	50	18	5	45
----	----	----	----	----	----	---	----

$15 > 10$
TRUE
SWAP

Selection Sort – Example 1



$10 > 30$
FALSE



$10 > 50$
FALSE

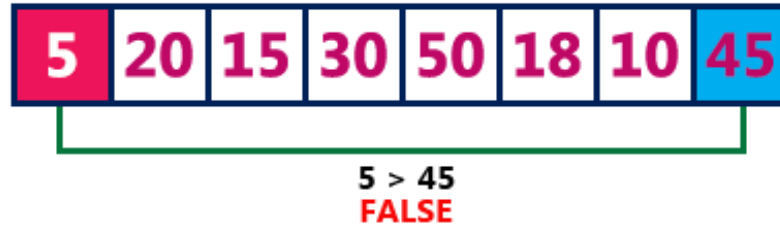


$10 > 18$
FALSE



$10 > 5$
TRUE
SWAP

Selection Sort – Example 1



List after 1st iteration



Iteration #2

Select the second position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 2nd iteration



Selection Sort – Example 1

Iteration #3

Select the third position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 3rd iteration

5	10	15	30	50	20	18	45
---	----	----	----	----	----	----	----

Iteration #4

Select the fourth position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 4th iteration

5	10	15	18	50	30	20	45
---	----	----	----	----	----	----	----

Iteration #5

Select the fifth position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 5th iteration

5	10	15	18	20	50	30	45
---	----	----	----	----	----	----	----

Selection Sort – Example 1

Iteration #6

Select the sixth position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 6th iteration

5	10	15	18	20	30	50	45
---	----	----	----	----	----	----	----

Iteration #7

Select the seventh position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 7th iteration

5	10	15	18	20	30	45	50
---	----	----	----	----	----	----	----

Final sorted list

Selection Sort - Exercise


Sort following array using Insertion Sort.

1. {16,15,2,13,6}

2. {44,77,22,88,33,66,11,55}

Insertion Sort

- Sorting is the process of arranging a list of elements in a particular order (Ascending or Descending).
- Insertion sort algorithm arranges a list of elements in a particular order. In insertion sort algorithm, every iteration moves an element from unsorted portion to sorted portion until all the elements are sorted in the list.



➤ The insertion sort algorithm is performed using following steps...

- **Step 1:** Assume that first element in the list is in sorted portion of the list and remaining all elements are in unsorted portion.
- **Step 2:** Consider first element from the unsorted list and insert that element into the sorted list in order specified.
- **Step 3:** Repeat the above process until all the elements from the unsorted list are moved into the sorted list.

Function for Insertion Sort

➤ //Insertion sort logic

```
for i = 1 to size-1 {  
    temp = list[i];  
    j = i;  
    while ((temp < list[j]) && (j > 0)) {  
        list[j] = list[j-1];  
        j = j - 1;  
    }  
    list[j] = temp;  
}
```

Insertion Sort – Example 1

Consider the following unsorted list of elements...

15	20	10	30	50	18	5	45
----	----	----	----	----	----	---	----

Assume that sorted portion of the list empty and all elements in the list are in unsorted portion of the list as shown in the figure below...

Sorted

Unsorted

15	20	10	30	50	18	5	45
----	----	----	----	----	----	---	----

Insertion Sort – Example 1

Move the first element 15 from unsorted portion to sorted portion of the list.

Sorted		Unsorted					
15	20	10	30	50	18	5	45

To move element 20 from unsorted to sorted portion, Compare 20 with 15 and insert it at correct position

Sorted		Unsorted					
15	20	10	30	50	18	5	45

Insertion Sort – Example 1

To move element 10 from unsorted to sorted portion, Compare 10 with 20 and it is smaller so swap. Then compare 10 with 15 again smaller swap. And 10 is insert at its correct position in sorted portion of the list.

Sorted			Unsorted				
10	15	20	30	50	18	5	45

To move element 30 from unsorted to sorted portion, Compare 30 with 20, 15 and 10. And it is larger than all these so 30 is directly inserted at last position in sorted portion of the list.

Sorted				Unsorted			
10	15	20	30	50	18	5	45

Insertion Sort – Example 1

To move element 50 from unsorted to sorted portion, Compare 50 with 30, 20, 15 and 10. And it is larger than all these so 50 is directly inserted at last position in sorted portion of the list.

Sorted					Unsorted		
10	15	20	30	50	18	5	45

To move element 18 from unsorted to sorted portion, Compare 18 with 30, 20 and 15. Since 18 is larger than 15, move 20, 30 and 50 one position to the right in the list and insert 18 after 15 in the sorted portion.

Sorted					Unsorted	
10	15	18	20	30	50	5 45

Insertion Sort – Example 1

To move element 5 from unsorted to sorted portion, Compare 5 with 50, 30, 20, 18, 15 and 10. Since 5 is smaller than all these element, move 10, 15, 18, 20, 30 and 50 one position to the right in the list and insert 5 at first position in the sorted list.

Sorted							Unsorted
5	10	15	18	20	30	50	45

To move element 45 from unsorted to sorted portion, Compare 45 with 50 and 30. Since 45 is larger than 30, move 50 one position to the right in the list and insert 45 after 30 in the sorted list.

Sorted							Unsorted
5	10	15	18	20	30	45	50

Insertion Sort – Example 1

Unsorted portion of the list has become empty. So we stop the process. And the final sorted list of elements is as follows...

5	10	15	18	20	30	45	50
---	----	----	----	----	----	----	----

Insertion Sort – Example 2

54	26	93	17	77	31	44	55	20	Assume 54 is a sorted list of 1 item
26	54	93	17	77	31	44	55	20	inserted 26
26	54	93	17	77	31	44	55	20	inserted 93
17	26	54	93	77	31	44	55	20	inserted 17
17	26	54	77	93	31	44	55	20	inserted 77
17	26	31	54	77	93	44	55	20	inserted 31
17	26	31	44	54	77	93	55	20	inserted 44
17	26	31	44	54	55	77	93	20	inserted 55
17	20	26	31	44	54	55	77	93	inserted 20

Insertion Sort – Exercise

Sort following array using Insertion Sort.

1. {25,15,30,9,99,20,26}

2. {44,77,22,88,33,66,11,55}

