# Lecture: Sorting Algorithms: Bubble Sort and Quick Sort

---

## 1. Introduction to Sorting

Sorting is a fundamental concept in computer science. It involves arranging data in a specific order, such as ascending or descending. Sorting algorithms are essential for optimizing search algorithms, organizing data, and solving complex problems efficiently.

## 2. Bubble Sort

Bubble Sort is one of the simplest sorting algorithms, working by repeatedly swapping adjacent elements that are out of order.

### Algorithm Steps

1. Compare adjacent elements in the array.
2. Swap them if they are in the wrong order.
3. Repeat the process for every pair until the array is sorted.
4. After each pass, the largest element "bubbles up" to its correct position.

### Example

**Unsorted Array**: [5, 3, 8, 4, 2]

| Pass | Array After Sorting | Comments |
|---|---|---|
| 1 | [3, 5, 4, 2, 8] | Largest element (8) sorted. |
| 2 | [3, 4, 2, 5, 8] | Second-largest sorted. |
| 3 | [3, 2, 4, 5, 8] | Third-largest sorted. |
| 4 | [2, 3, 4, 5, 8] | Fully sorted array. |

### Complexity

- **Time Complexity**:
  - Best Case: O(n) (when already sorted)
  - Worst Case: O($n^2$)(for reversed arrays)

```
namespace ConsoleApplication1
{
    class program
    {
        public static void bubble_sort()
        {
            Console.WriteLine("enter the total number of elements:");
             int max = int.Parse(Console.ReadLine());
            int[] ss = new int[max];
            for (int i = 0; i < max; i++)
            {
                Console.WriteLine("enter[{0}] element:", i);
                ss[i] = int.Parse(Console.ReadLine());
            }
            Console.WriteLine("befor sorting:");
            for (int k = 0; k < max; k++)
                Console.WriteLine(ss[k]);
            for (int i = 0; i < max; i++)
            {
                for (int j = 0; j < max - 1; j++)
                {
                    if (ss[j] > ss[j + 1])
                    {
                        int temp = ss[j];
                        ss[j] = ss[j + 1];
                        ss[j + 1] = temp;
                    }
                }
                Console.WriteLine("no. of pass "+ " "+(i+1));
                foreach (int num in ss)
                    Console.Write(num+ "");
                Console.WriteLine();
            }
            Console.WriteLine("the numbers in ascending orders are given below;");
            for (int i = 0; i < max; i++)
            {
                Console.WriteLine("sorted[{0}] element:", i);
                Console.WriteLine(ss[i]);
            }
        }
        static void Main(string[] args)
        {
            bubble_sort();
            Console.Read();
        }
    }
}
```

### 3. Quick Sort

Quick Sort is an efficient, divide-and-conquer algorithm that partitions the array and recursively sorts the subarrays.

#### Algorithm Steps

1. Choose a **pivot** element.
2. Partition the array into two halves:
    - Elements less than the pivot.
    - Elements greater than or equal to the pivot.

3. Recursively apply Quick Sort to the two halves.
4. Combine the sorted halves.

Example;

pivot

| | | | | | |
|---|---|---|---|---|---|
| 50 | 20 | 60 | 10 | 30 | 40 |
| i | | | | | j |

If(arr[i] < arr[j]) →NO → Swap (40,50) and i++

| | | | | | |
|---|---|---|---|---|---|
| 40 | 20 | 60 | 10 | 30 | 50 |
| | i | | | | j |

If(arr[i] <arr[j]) →yes →No Swap and i++

| | | | | | |
|---|---|---|---|---|---|
| 40 | 20 | 60 | 10 | 30 | 50 |
| | | i | | | j |

If(arr[i] < arr[j]) →NO → Swap (60,50) and j--

| | | | | | |
|---|---|---|---|---|---|
| 40 | 20 | 50 | 10 | 30 | 60 |
| | | i | | j | |

If(arr[i] < arr[j]) →NO → Swap (50,30) and i++

| | | | | | |
|---|---|---|---|---|---|
| 40 | 20 | 30 | 10 | 50 | 60 |
| | | | j | j | |

If(arr[i] <arr[j]) →yes →No Swap(10,50) and i++

| | | | | | |
|---|---|---|---|---|---|
| 40 | 20 | 30 | 10 | 50 | 60 |
| | | | | ij | |

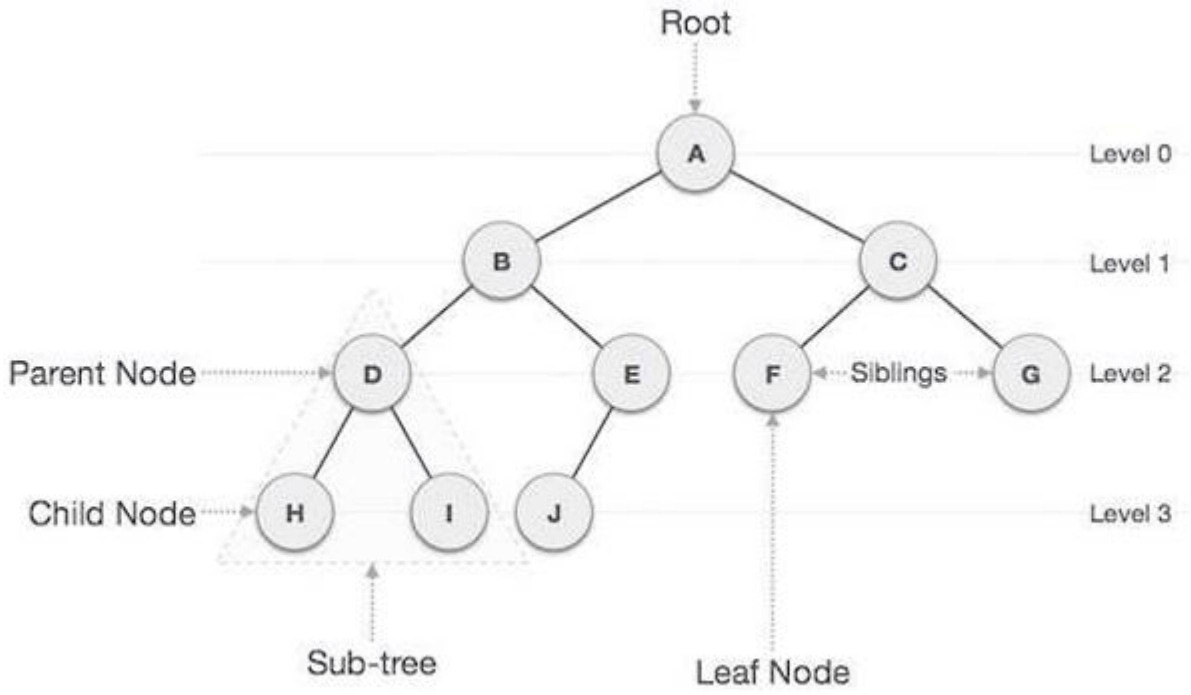
هنا pivot اصبح في مكانه الصحيح وقسم المصفوفة الى جزئين الجزء الايمن $\boxed{60}$ والجزء الايسر هو

| | | | |
|---|---|---|---|
| 40 | 20 | 30 | 10 |

نكرر نفس العملية السابقة للمصفوفة اليمنى واليسرى لغاية الوصول الى عنصر واحد في كل مصفوفة وتصبح المصفوفة مرتبة.

## Tree

Tree represents the nodes connected by edges. Binary Tree is a special data structure used for data storage purposes. A binary tree has a special condition that each node can have a maximum of two children. A binary tree has the benefits of both an ordered array and a linked list as search is as quick as in a sorted array and insertion or deletion operation are as fast as in linked list.



**Important Terms**

Following are the important terms with respect to tree.

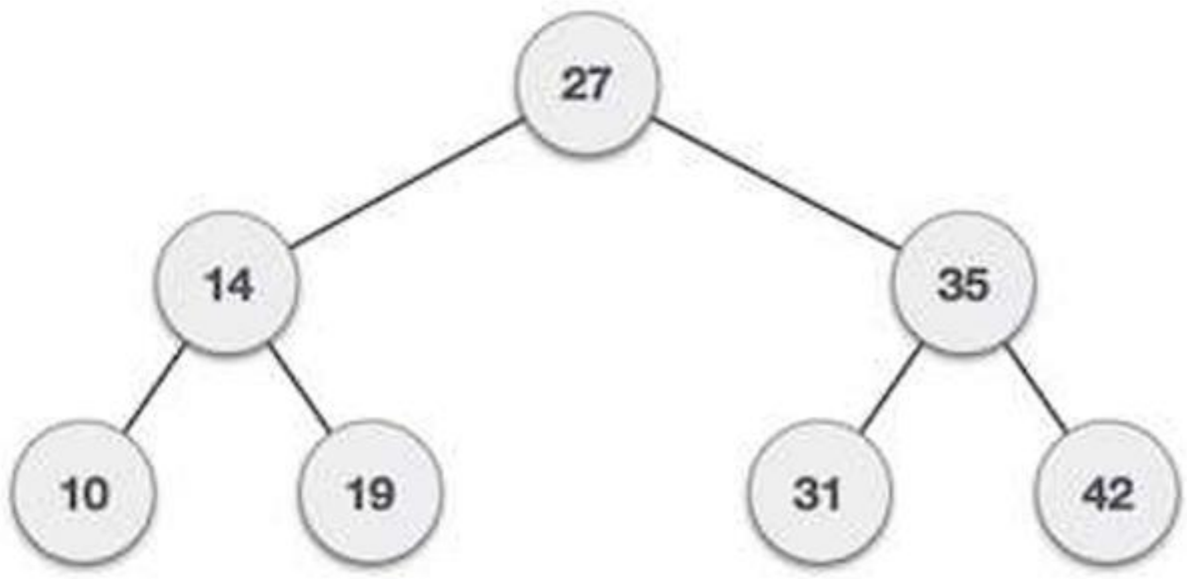
- Path − Path refers to the sequence of nodes along the edges of a tree.
- Root – The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
- Parent − Any node except the root node has one edge upward to a node called parent.
- Child – The node below a given node connected by its edge downward is called its child node.
- Leaf – The node which does not have any child node is called the leaf node.
- Subtree − Subtree represents the descendants of a node.
- Visiting − Visiting refers to checking the value of a node when control is on the node.

• Traversing − Traversing means passing through nodes in a specific order.

• Levels − Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.

**Binary Search Tree**

Representation Binary Search tree exhibits a special behavior. A node's left child must have a value less than its parent's value and the node's right child must have a value greater than its parent value.



**There are three types of binary tree traversals.**

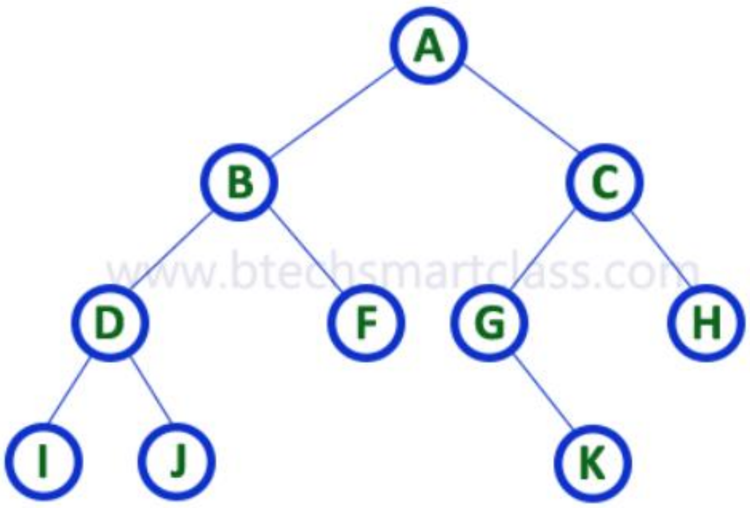
**In - Order Traversal**

**Pre - Order Traversal**

**Post - Order Traversal**

## 1. In - Order Traversal ( leftChild - root - rightChild )

In In-Order traversal, the root node is visited between left child and right child. In this traversal, the left child node is visited first, then the root node is visited and later we go for visiting right child node. This in-order traversal is applicable for every root node of all subtrees in the tree. This is performed recursively for all nodes in the tree.



In the above example of binary tree, first we try to visit left child of root node 'A', but A's left child is a root node for left subtree. so we try to visit its (B's) left child 'D' and again D is a root for subtree with nodes D, I and J. So we try to visit its left child 'I' and it is the left most child. So first we visit 'I' then go for its root node 'D' and later we visit D's right child 'J'. With this we have completed the left part of node B. Then visit 'B' and next B's right child 'F' is visited. With this we have completed left part of node A. Then visit root node 'A'. With this we have completed left and root parts of node A. Then we go for right part of the node A. In right of A again there is a subtree with root C. So go for left child of C and again it is a subtree with root G. But G does not have left part so we visit 'G' and then visit G's right child K.

With this we have completed the left part of node C. Then visit root node **'C'** and next visit C's right child **'H'** which is the right most child in the tree so we stop the process.

That means here we have visited in the order of **I - D - J - B - F - A - G - K - C - H** using In-Order Traversal.

In-Order Traversal for above example of binary tree is

I - D - J - B - F - A - G - K - C - H

2. **Pre - Order Traversal ( root - leftChild - rightChild )**

In Pre-Order traversal, the root node is visited before left child and right child nodes. In this traversal, the root node is visited first, then its left child and later its right child. This pre-order traversal is applicable for every root node of all subtrees in the tree.

In the above example of binary tree, first we visit root node 'A' then visit its left child 'B' which is a root for D and F. So we visit B's left child 'D' and again D is a root for I and J. So we visit D's left child 'I' which is the left most child. So next we go for visiting D's right child 'J'. With this we have completed root, left and right parts of node D and root, left parts of node B. Next visit B's right child 'F'. With this we have completed root and left parts of node A. So we go for A's right child 'C' which is a root node for G and H. After visiting C, we go for its left child 'G' which is a root for node K. So next we visit left of G, but it does not have left child so we go for G's right child 'K'. With this we have completed node C's root and left parts. Next visit C's right child 'H' which is the right most child in the tree. So we stop the process.

That means here we have visited in the order of A-B-D-I-J-F-C-G-K-H using Pre-Order Traversal.

Pre-Order Traversal for above example binary tree is

A - B - D - I - J - F - C - G - K - H

**3-Post - Order Traversal ( leftChild - rightChild - root )**

In Post-Order traversal, the root node is visited after left child and right child. In this traversal, left child node is visited first, then its right child and then its root node. This is recursively performed until the right most node is visited.

Here we have visited in the order of I - J - D - F - B - K - G - H - C - A using Post-Order Traversal.

Post-Order Traversal for above example binary tree is

I - J - D - F - B - K - G - H - C - A