P1- What are tree data structure?illustrate 2 practical example where tree data structure is used?

A tree data structure is a non-linear data structure which is a natural choice to store hierarchical information.

More formally, a tree data structure is defined as follows.

Definition:
A tree data structure is a finite set T of nodes such that

1. There is a specially designated node called root
2. All other nodes can be partitioned into m sets T1, T2,..., Tm such that m >= 0 and all of these sets are also trees themselves

Implementation:

There are many ways to implement a tree.

A typical implementation of tree uses 2 fields, one is the data field(or key) and the other is a collection of children nodes. The collection of children nodes is usually implemented using a linear data structure like queue or linked-list.

Practical example of Tree data structure:

1. File System Hierarchy

Hierarchy of files in file systems of popular OS like linux based distributions is represented with tree like data structure.

Also the file system of databases, which require quick searching, is implemented with the help of a special kind of tree called B-tree which is a logical extension to binary search tree.

2. DOM representation

Modern web technologies use a tree-like in-memory representation of a web page. This is called Document Object Model or DOM.

Here, all the HTML tags are nodes of the DOM tree and a tag which is contained inside another tag becomes the child of the latter tag. This abstraction gives a convenient and efficient way to manipulate the web page.

P2-Explain different tree traversals Algorithm? With pseudocode and example.

There are many traversal algorithms for trees. Some of the popular ones are pre-order and post-order traversal.

For Binary tree, there is also one more traversal called in-order traversal however such traversal is not possible in case of a generic tree as there is no concept of left and right subtree in case of a generic tree.

1. Pre-order traversal:

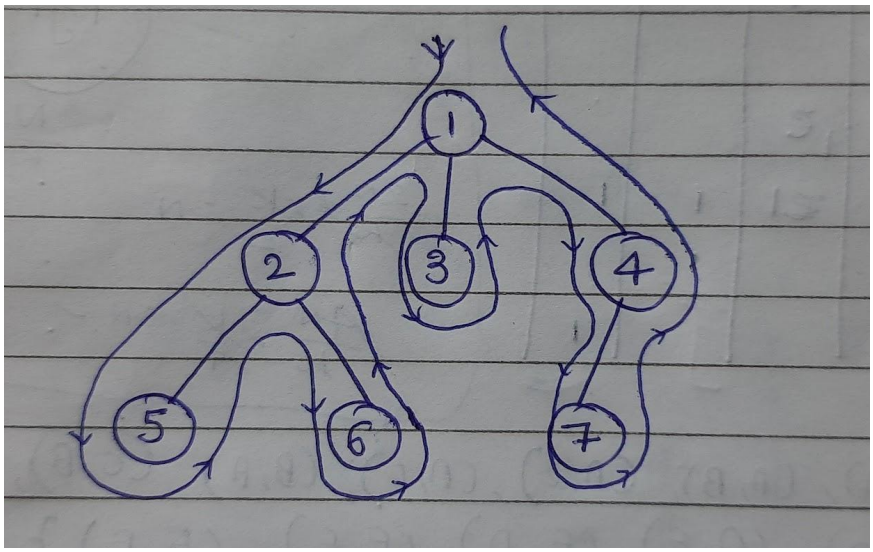
In this traversal, the parent node is visited before all of its children.

Its pseudocode is as follows.

preOrder(root)

1. Print root
2. For child in root.children
    a. Recursively call preOrder(child)

Let's take the below example of a tree.



Its pre-order traversal is 1, 2, 5, 6, 3, 4, 7.

2. Post-order traversal:

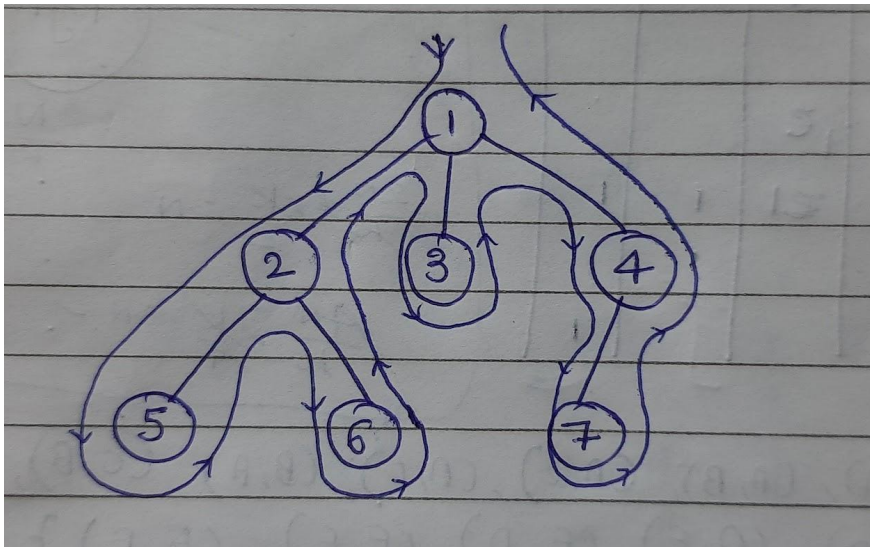
In this traversal, the child nodes are visited before the parent node.

Its pseudocode is as follows.

postOrder(root)

1. If root.children is empty print root
2. Return
3. For child in root.children
    a. Recursively call postOrder(child)
4. print root

Let’s take the below example of a tree.



Its post-order traversal is 5, 6, 2, 3, 7, 4, 1.

P3- How can the bounds be determined for any sorting algorithm? Can a better bound than nlg(n) be obtained?

To find the upper or lower bound for any sorting algorithm, we first need to know how the algorithm finds the relative order between any two elements.

Some algorithms like quick sort, insertion sort, merge sort etc. uses comparison between elements to know this relative order while some other algorithms like counting sort uses extra information like the range of values of elements along with comparison to sort the elements.

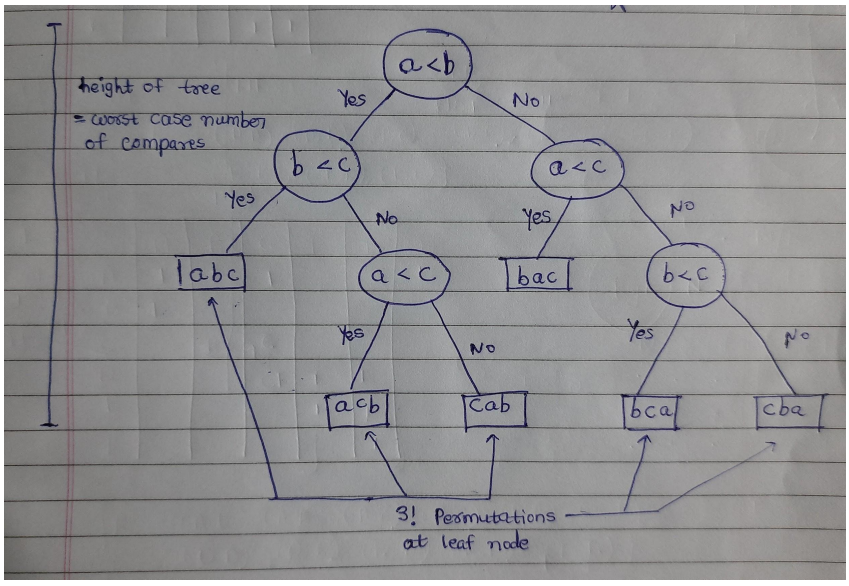
In few input cases, non-comparison based algorithms like counting sort can outperform comparison based algorithms. But for the general case, it might take very large auxiliary space to do that. For example, counting sort is preferable when the range of values of elements is very less than the number of elements. But for the general case, comparison based sorting algorithms are better.

We can also prove that there is no comparison based algorithm which provides better than O(nlogn) time complexity.

Proof:

Let's say we have n elements to sort. The possible sorted order could be any one of all possible permutations of n elements which is n!.

A comparison based sorting algorithm will need to do a series of comparisons involving all n elements in order to find the relative order among these n elements. Let's take a look on such a series of comparisons for 3 elements



So, we can see that after series of comparisons, all possible permutations will be present as the leaf nodes of the tree. The number of comparisons needed to sort the elements in some order is equal to the length of the path from root to that leaf node.

The decision tree is a binary tree. Let h be the maximum height of the tree. This will give the maximum number of comparisons needed to sort the element.

A binary tree with height h can have at most 2^h leaves. We already know that, there will be n! different permutations at the leaf nodes.

So $2^h >= n!$.
$h >= \log(n!)$
We can approximate the value of h by taking the integral of log(n!) from n = 1 to n. Hence, h will be approximately equal to nlogn.

P4.-Explain the Double ended queue along with its Pseudo code?

A double ended queue is similar to a “normal” queue just with the modification that enqueue and dequeue operations can be performed at both the ends.

We will maintain two pointers namely front and back which will point to the front and back of the queue. Also, we are assuming here that the space for the elements are allocated dynamically. Also, we will maintain a field called size which will store the number of elements currently present in the queue.

Structure of the node will be as follows.

Node:

data : it will store the data
next : it will store the pointer to next node
prev : it will store the pointer to previous node

Pseudo code for enqueue, dequeue at the front and rear are as follows.

enqueueFront(data):

```
if front == null:
        node = allocate space for data
        front = node
        back = node

else:
        node = allocate space for data
        node.next = front
        front.prev = node
        front = node
size++
```

enqueueBack(data):

```
if back == null:
        node = allocate space for data
        back = node
        front = node

else:
        node = allocate space for data
        back.next = node
        node.prev = back
        back = node
size++
```

dequeueFront():

```
if front == null:
        Return
node = front
front = front.next
front.prev = null
size--;
Return node.data
```

dequeueBack():

```
if back == null:
        Return
node = back
back = back.prev
back.next = null
size--;
Return node.data
```

size():

```
Return size
```

isEmpty():

```
Return size == 0
```