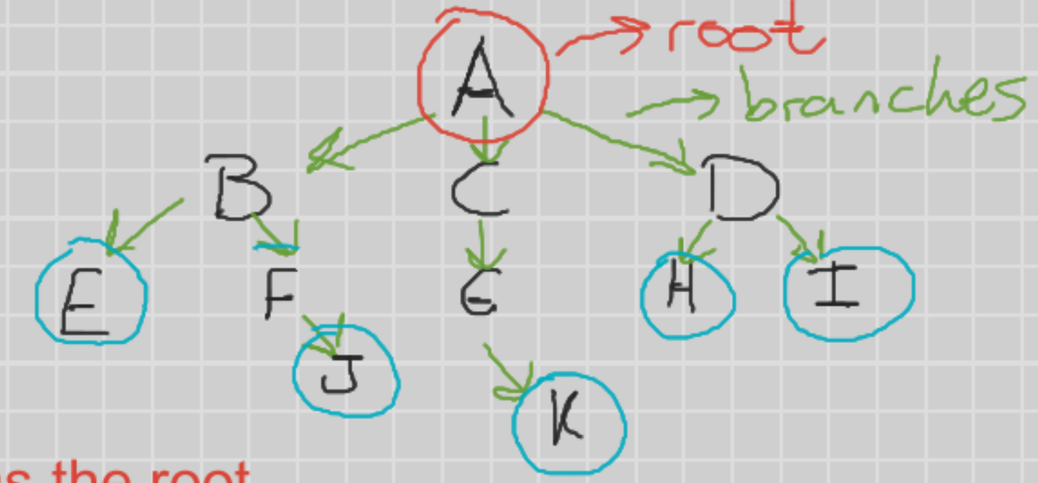
Trees



- Trees: are a non-linear data structure
  - they have a hierarchal organization

- The node at the very top of the tree is known as the root
  - Just like a linked list, the root is a pointer to the first element of the tree

- The nodes at the end of the tree are called leaves.

- The arrows (edges) from one node to another are called branches.

- The node at the top of an arrow is called a PARENT, while the node at the end of an arrow is called a CHILD. Example: We say that node A is the parent of B, C, and D. Nodes E & F are the children of B.

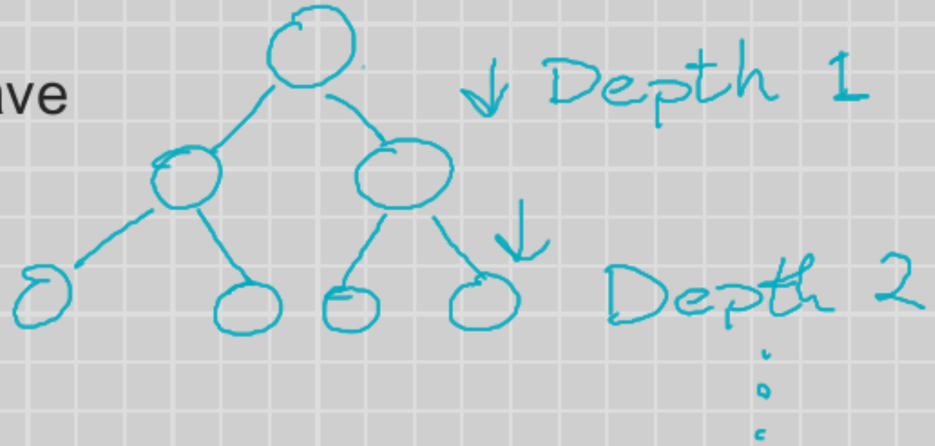
- All of the nodes on the way back from a node to the root, are known as ANCESTORS. Example: G, C, and A are the ancestors of K.

- All of the nodes on the way down form a node are known as DESCENDANTS. Example: E, F are descendants of B.

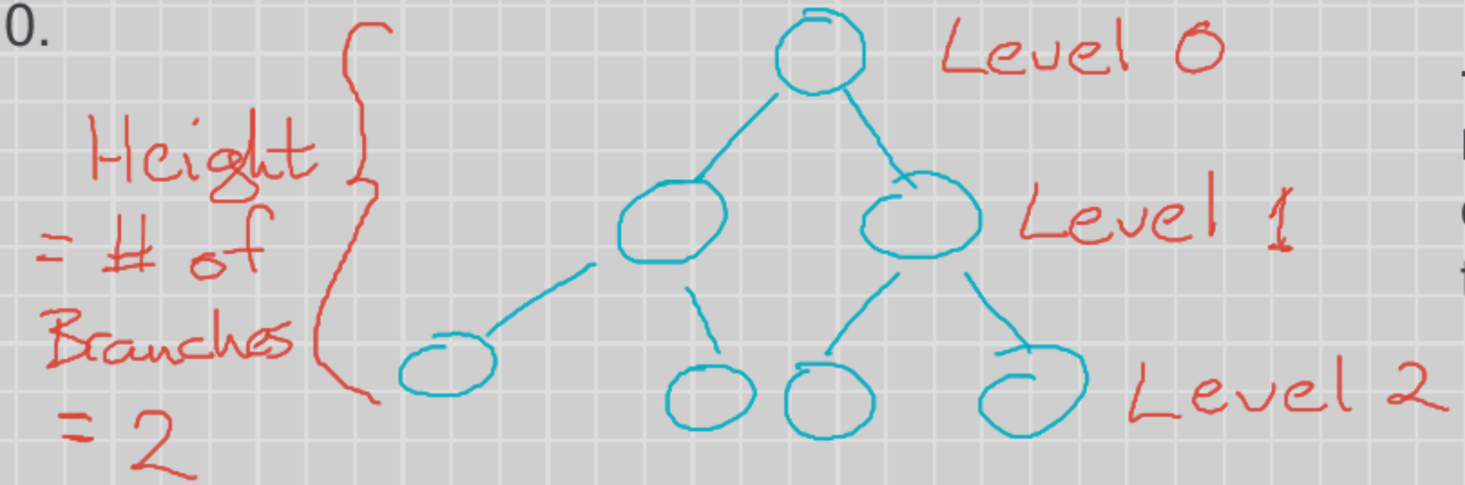
- Every node in a tree has a DEPTH, which is number of branches between the root and the node under inspection.

- All children of the root have a depth of 1 because they are only one branch away.

- The grandchildren of the root have a depth of 2.

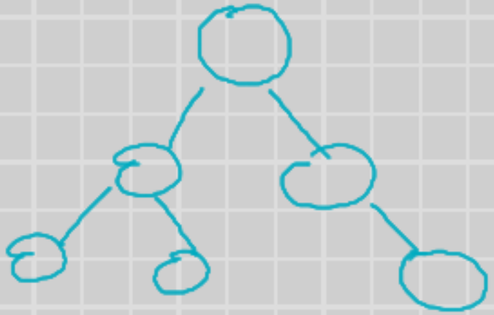


- The nodes that have the same depth belong the the same LEVEL of the tree, with the ROOT being LEVEL 0.



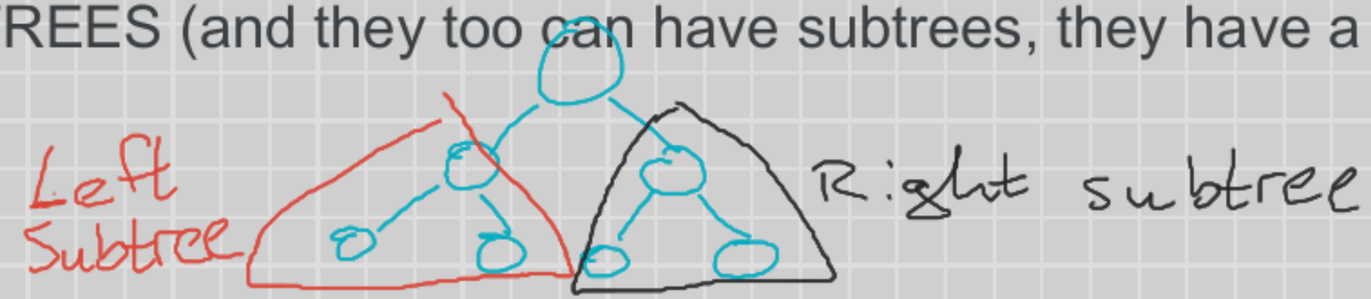
- The HEIGHT of a tree is measured by the number of branches from the root to the deepest leaf.

- Every node in a generic tree can have as many or as little children as needed.
- When the number of of children of every node is constrained to a maximum of 2, the tree is known as a binary tree.



- If every node has exactly two children, except the leaves, then it's known as a FULL BINARY TREE.

- In a binary tree since every node has two children they are drawn to the left, and right, we refer to them as the LEFT CHILD, and RIGHT CHILD. We can refer to them and their descendants as the left and right SUBTREES (and they too can have subtrees, they have a recursive structure).



Binary Tree Implementation

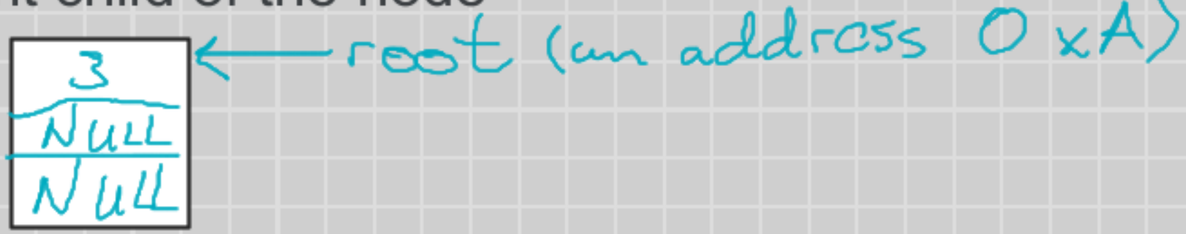
There are two ways that we will implement our binary trees:

1) Using memory pointers like a linked list.
2) Using arrays

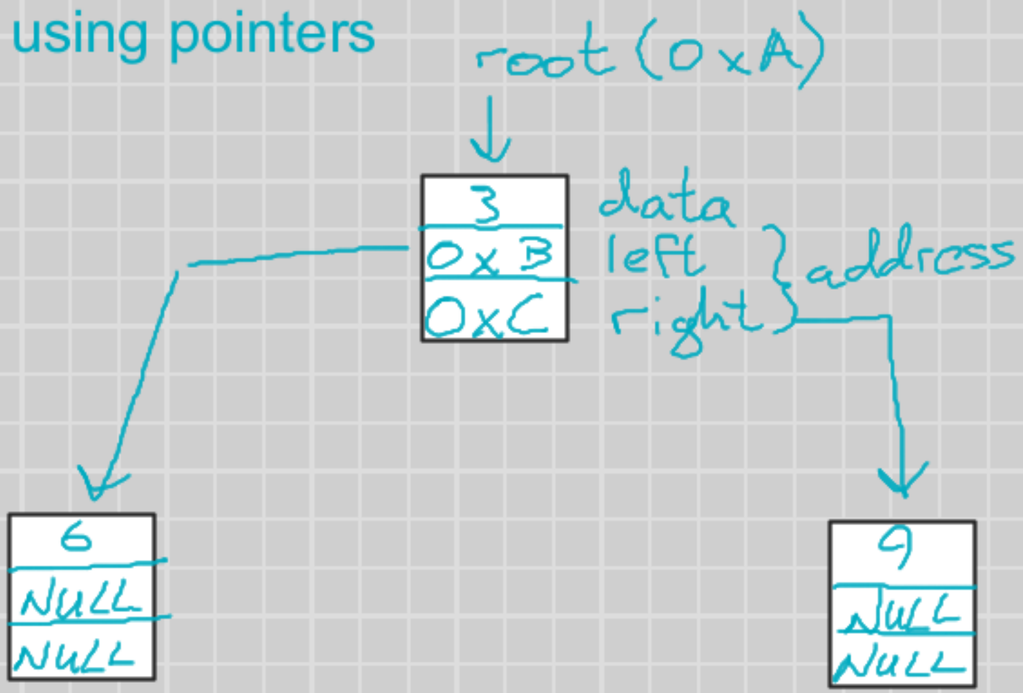
1) Using Pointers

When we use pointers we must define the nodes to store three pieces of info:

a) Data - the value being stored
b) Left memory address of the left child of the node
c) Right memory address of the right child of the node

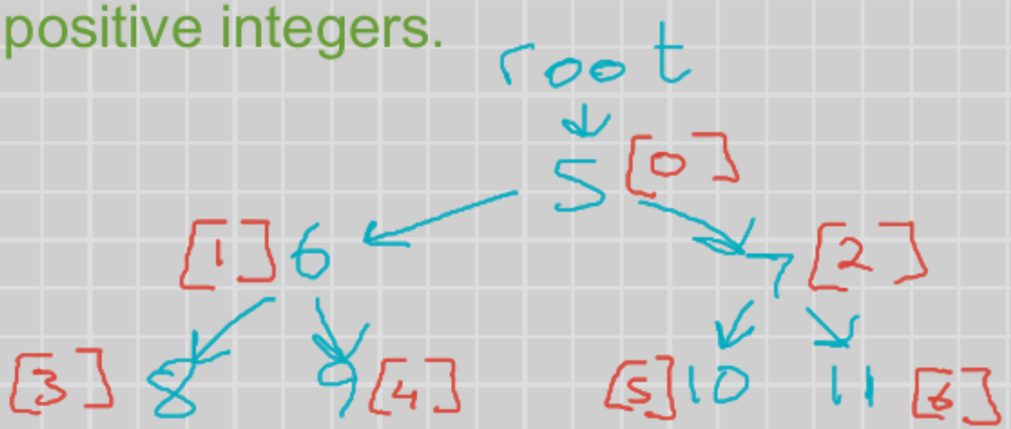


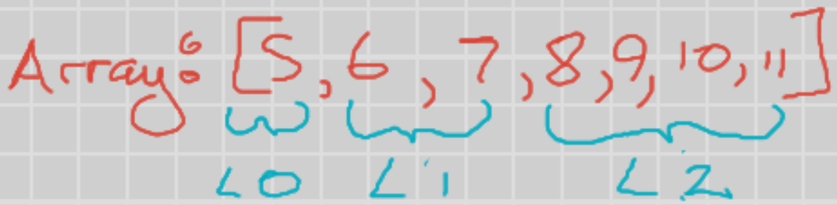
Adding nodes to a tree using pointers



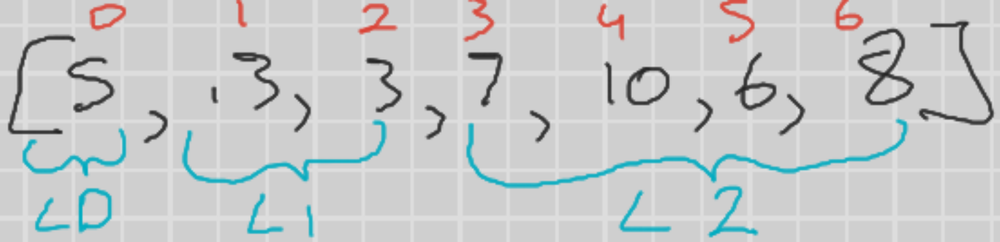
Binary Trees using arrays

1) The element at LEVEL 0 is stored at index 0 in the array. This is the ROOT of the tree.
2) The elements at LEVEL 1 of the tree are stored at indices 1 & 2 in the array.
3) The elements at LEVEL 2 of the tree are stored at indices 3, 4, 5, and 6 in the array.
4) To indicate the absense of a node at any given index, we will usually store a number outside of the range of acceptable values for the tree. We could use -1 to indicate the absence of a node in a tree of positive integers.





*In general element at LEVEL k of a tree are store using 2^k positions in an array, starting at position [(2^k) - 1] and ending at [2^(k+1) - 2].



-> Elements at level 0, one position needed (2^0) = 1. The element at level 0 will be placed at position [(2^k) - 1] = (2^0) - 1 = 0.

-> Elements at level 1, two positions needed (2^1) = 2. The elements at level 1 will be placed at positions:
1) [(2^k) - 1] = (2^1) - 1 = 1.
2) (2^1) = 2 === [2^(k+1) - 2] = (2^1+1) - 2 = (2^2) - 2 = 2

-> Elements at level 2, need 2^2 = 4 positions. The elements at level 2 will be placed at positions:
1) [(2^k) - 1] = (2^2) - 1 = 3
2) (2^2) = 4
3) (2^2) + 1 = 5
4) (2^2) + 2 = 6 === [2^(k+1) - 2] = (2^2+1) - 2 = (2^3) - 2 = 6

ASIDE: A node at the ith index (when indexing starts a 0) will have its: 1) Left child at 2i+1 2) Right child at 2i+2 and 3) Parent at FLOOR[(i - 1)/2]

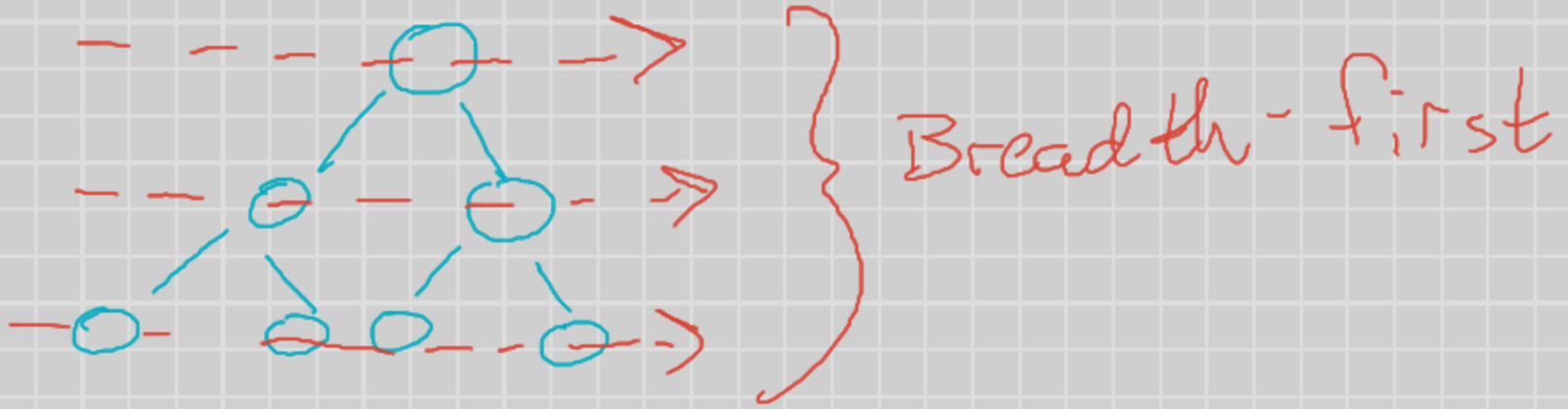
Binary Tree Traversals

Traversing a tree is important, because in order to INSERT, DELETE, or SEARCH for a specifi node, you must be able to get to it.

- TRAVERSAL is defined as visiting ALL the nodes of a tree. (Vist every single node)

TWO MAIN TRAVERSAL APPROACHES:

1) Breadth-first traversal: traverse the tree horizontally from left to right, startign from the root, then level 1 from left to right, and so on, visiting all siblings & counsins and the whole family in a given level before visiting their descendants.

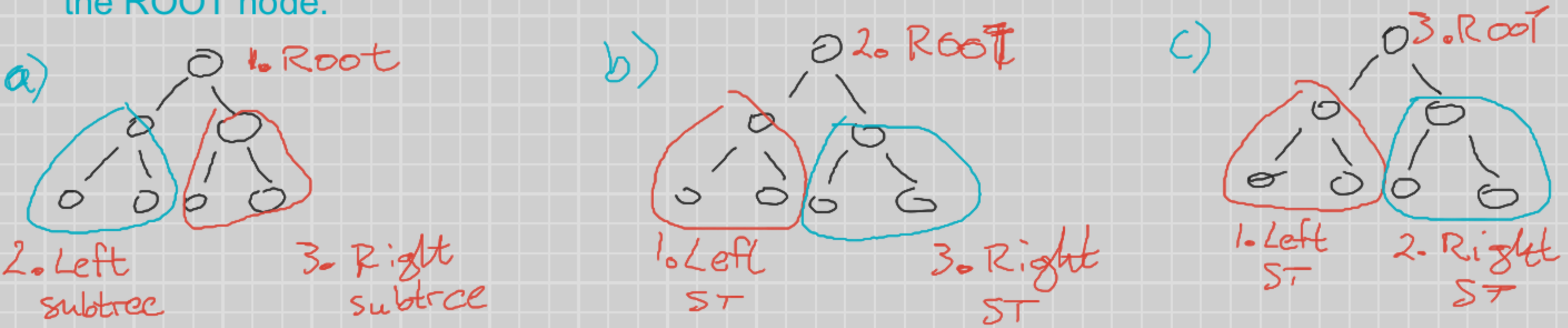


2) Depth-first Traversal: traverse the tree following its recursive structure vertically as if to divide the tree. There are three different types of dpeth-first traversal:

a) PRE-order: ROOT is the first node to visit, then visit the left subtree, and then the rright subtree.

b) IN-order: ROOT is visited in the middle of the traversal. First visit the left subtree, then the ROOT, and then the right subtree.

c) POST-order: ROOT is visited last. First visit the left subtree, then the right subtree, and finally the ROOT node.



```
function pre-order(T)
  if !ISEMPTY(T) then
    visit(root(T))
    pre-order(left(T))
    pre-order(right(T))
  end if
end function
```

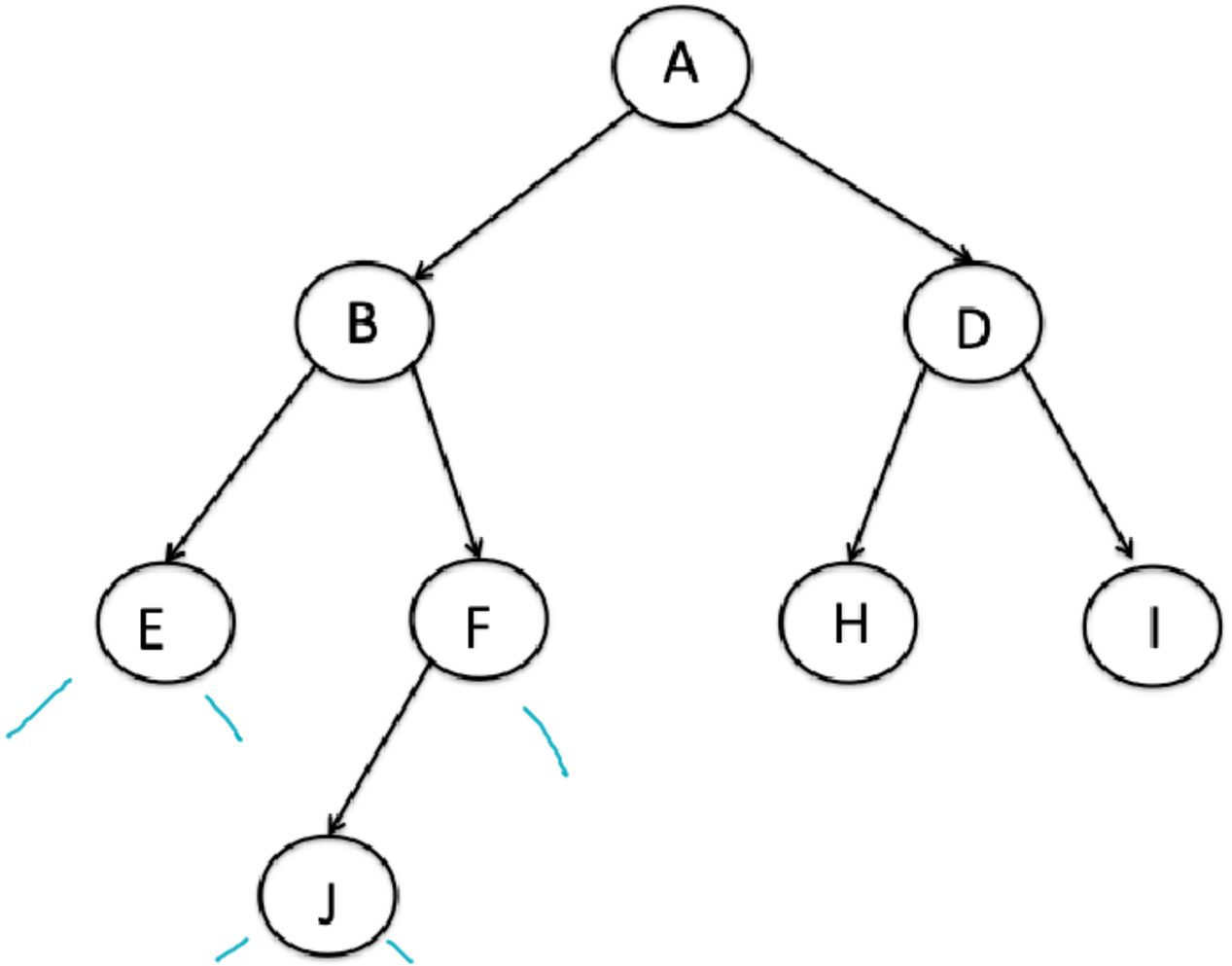
```
function in-order(T)
  if !ISEMPTY(T) then
    in-order(left(T))
    visit(root(T))
    in-order(right(T))
  end if
end function
```

```
function post-order(T)
  if !ISEMPTY(T) then
    post-order(left(T))
    post-order(right(T))
    visit(root(T))
  end if
end function
```

```
function pre-order(T)
  if !ISEMPTY(T) then
    visit(root(T))
    pre-order(left(T))
    pre-order(right(T))
  end if
end function
```

[A, B, E, F, J, D, H, I]

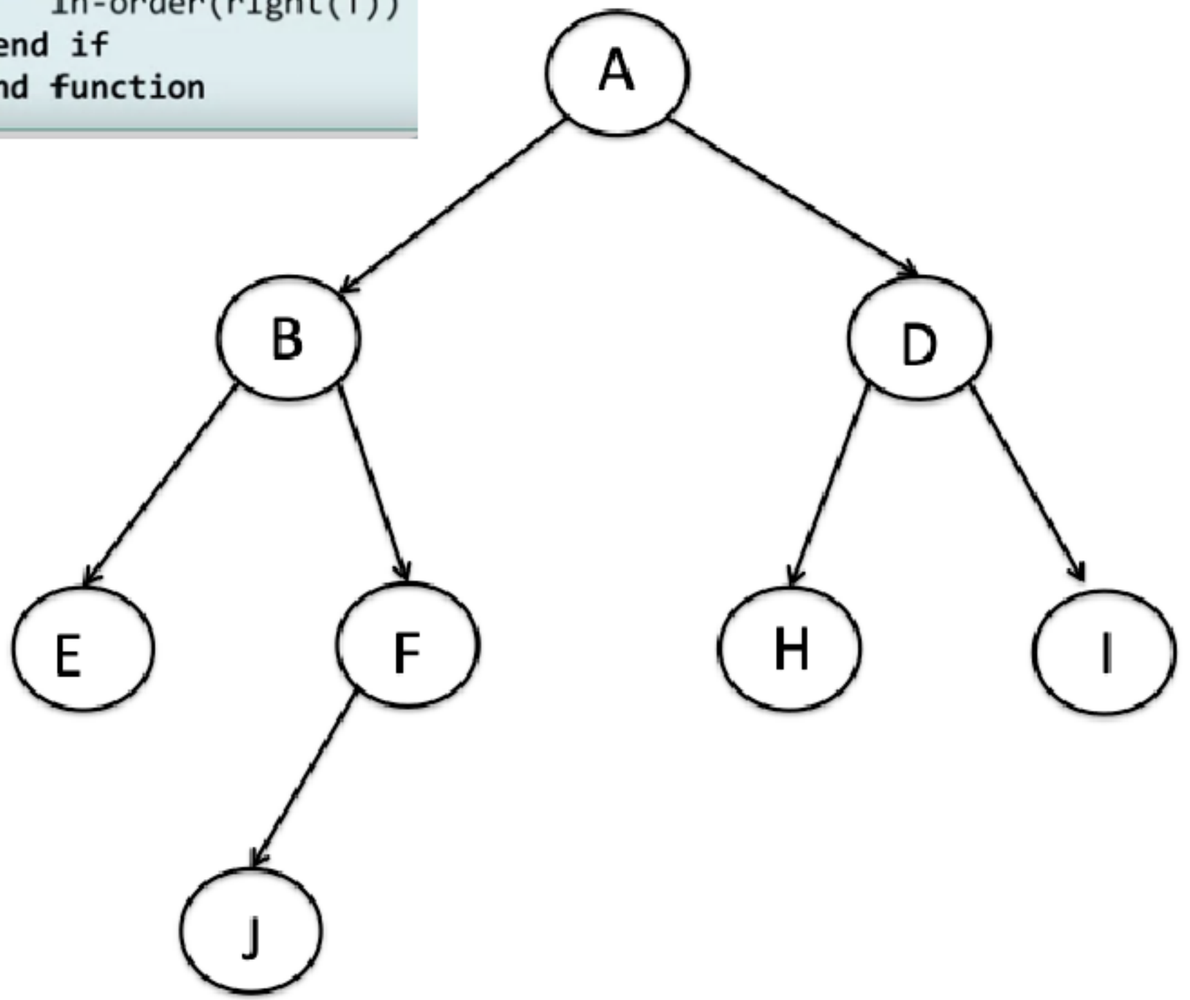
1. If we traverse this tree using pre-order traversal and we print the content of each node when visiting it, what is printed on screen?



[E, B, J, F, A, H, D, I]

1. If we raversal and we print the content of each node when visiting it, what is printed on scre

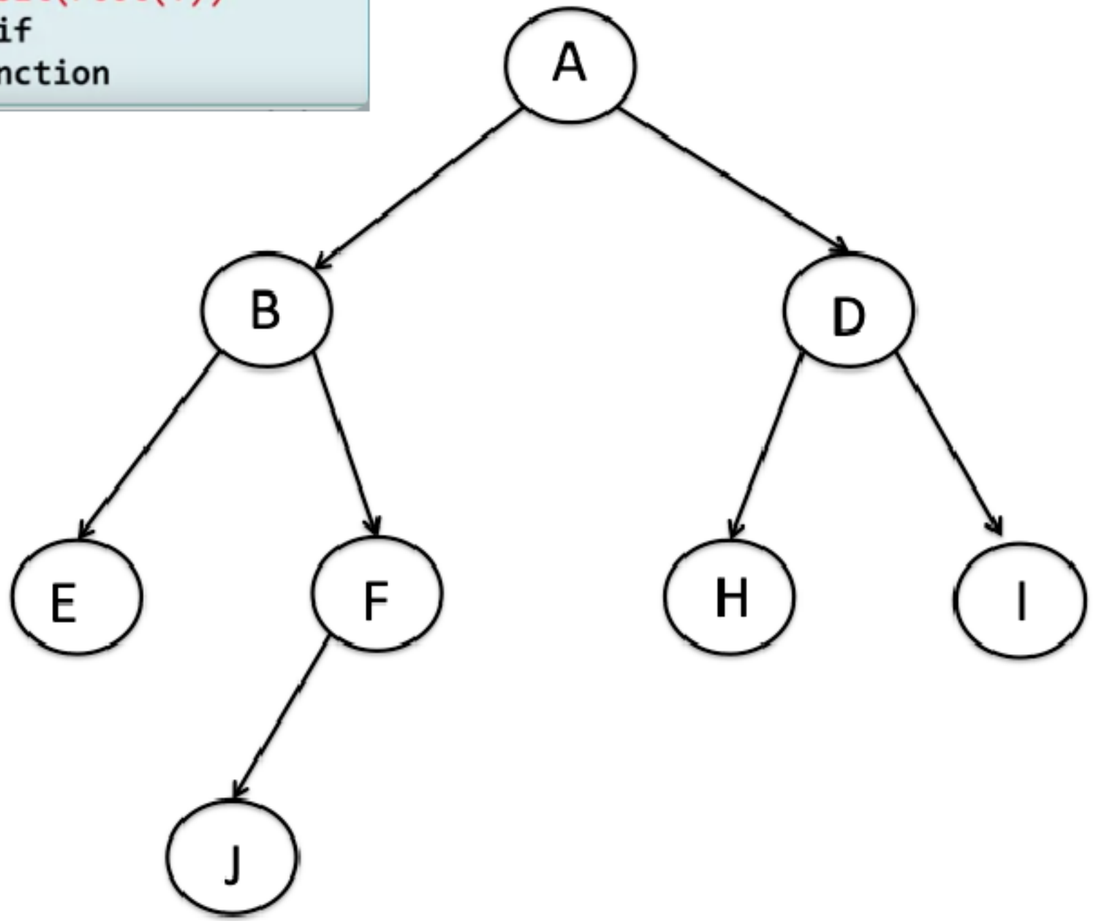
```
function in-order(T)
  if !ISEMPTY(T) then
    in-order(left(T))
    visit(root(T))
    in-order(right(T))
 end if
end function
```



[E, J, F, B, H, I, D, A]

```
function post-order(T)
  if !ISEMPTY(T) then
    post-order(left(T))
    post-order(right(T))
    visit(root(T))
  end if
end function
```

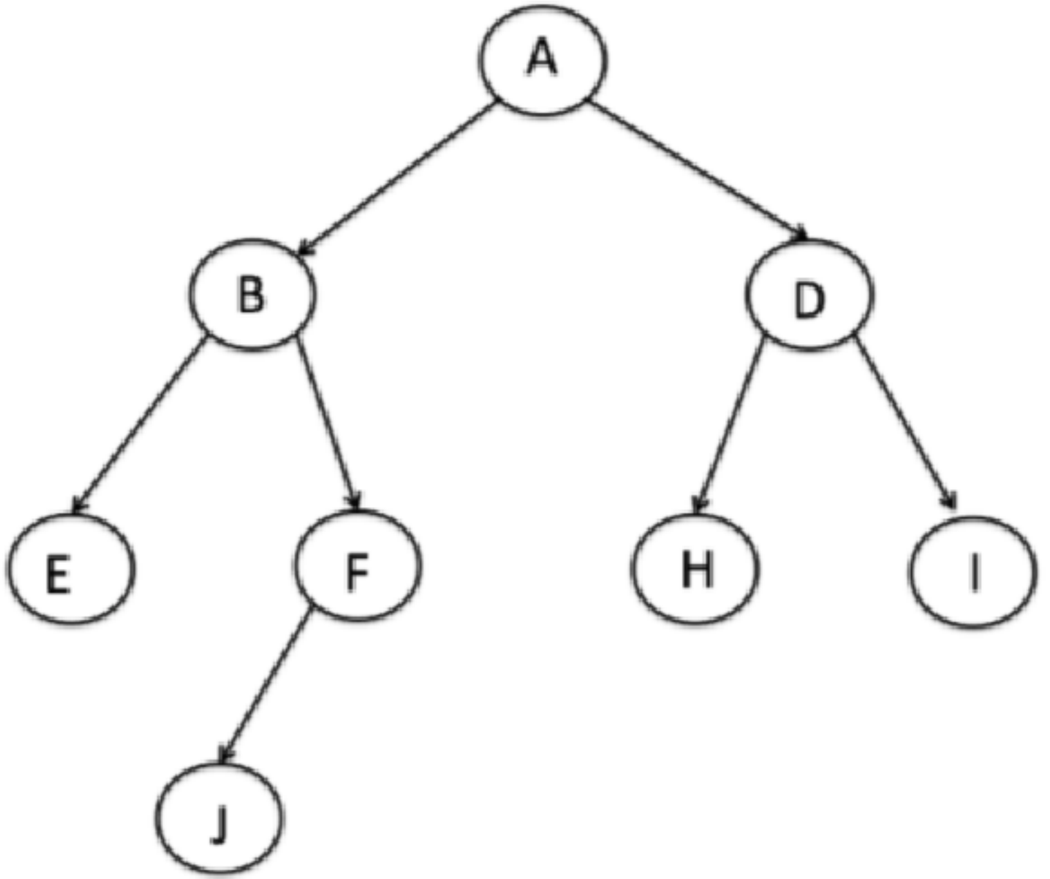
traversal and we print the content of each node when visiting it, what is printed on



[I, D, H, A, F, J, B, E]

4. If we traverse the tree shown in the figure using this traversal:

```
1  function no-order(T)
2      if !ISEMPTY(T) then
3          no-order(right(T))
4          print(root(T))
5          no-order(left(T))
6  end function
```



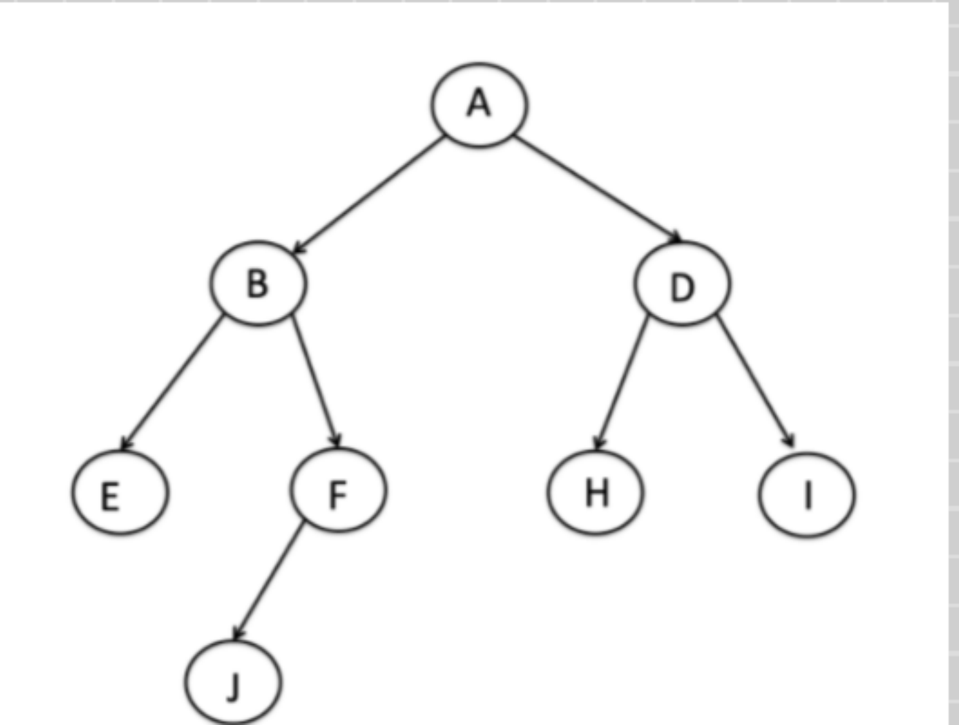
Breadth - first traversal

Here we can't take advantage of the vertical recursive structure of the tree, because the traversal takes place horizontally, where it doesn't have a recursive structure.

*Every iterative function can be transformed into a resursive function, but there are times where that tranformation is not convenient, breadth-first traversal is an example of that.

We can use a queue to keep track and print nodes.

[A, B, D, E, F, H, I, J] - result of breadth-first traversal



```
1  function breadth-first(root)
2     Q ← new Queue()
3     enqueue-if(Q,root)
4     while !ISEMPTY(Q)  do
5        t ← PEEK(Q)
6        visit(t)
7        enqueue-if(Q,left(t))
8        enqueue-if(Q,right(t))
9        DEQUEUE(Q)
10    end while
11 end function
```

Create Empty Queue

Enqueue Root Node

Check if Queue is Empty

Repeated While there is no in the Tree to Visit

Obtain Information of Element at front of the queue

Visit Node

Insert information of nodes Children

Node Removed once it is visited and children info Stored

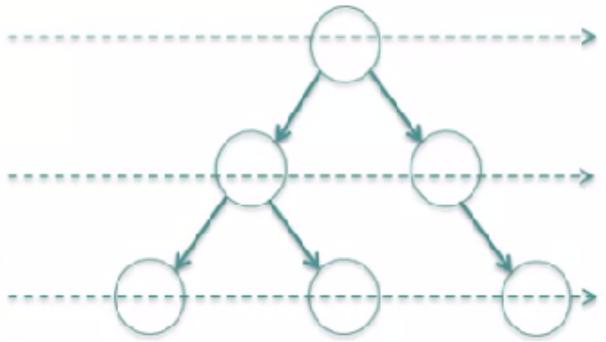
```
function enqueue-if(Q,t)
   if !NULL(t)  then
      ENQUEUE(Q,t)
end function
```

```
left(t) {
    return t.left;
}
```

visit(T.root)

visit(T.root.left); visit(T.root.right)

visit(T.root.left.left); visit(T.root.left.right);
visit(T.root.right.left); visit(T.root.right.right)



```
function breadth-first(root)
    Q ← new Queue()
    enqueue-if(Q,root)
    while !ISEMPTY(Q)  do
        t ← PEEK(Q)
        visit(t)
        enqueue-if(Q,left(t))
        enqueue-if(Q,right(t))
        DEQUEUE(Q)
    end while
end function
```

Create Empty Queue

Enqueue Root Node

Check if Queue is Empty

Repeated While there is no in the Tree to Visit

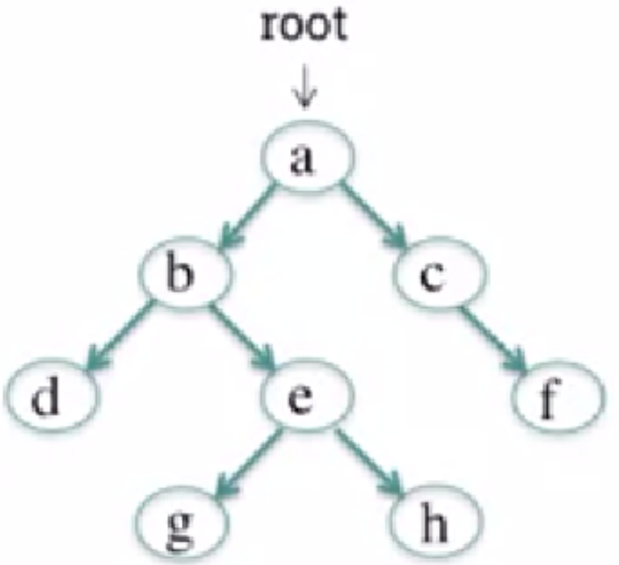
Obtain Information of Element at front of the queue

Visit Node print

Insert information of nodes Children

Node Removed once it is visited and children info Stored

```
function enqueue-if(Q,t)
    if !NULL(t)  then
        ENQUEUE(Q,t)
end function
```



Q = [A, B, C, D, E, F, G, H]

Print: A, B, C, D, E, F, G, H