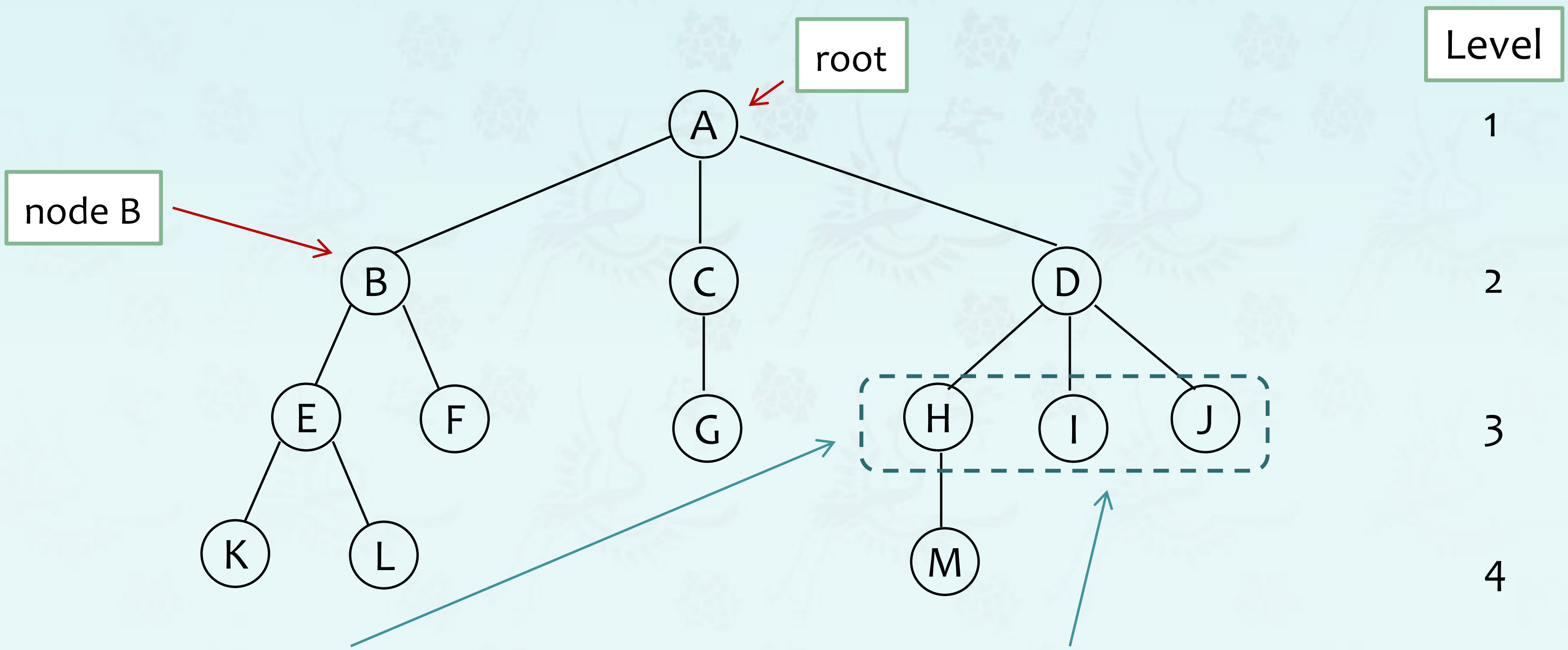
ITP20001/ECE20010 Data Structures

Chapter 5

- *Check your attendance – it matters!*

A tree data structure: it is like a linked list that has a **first** node, this node is called as the **root** of the tree.

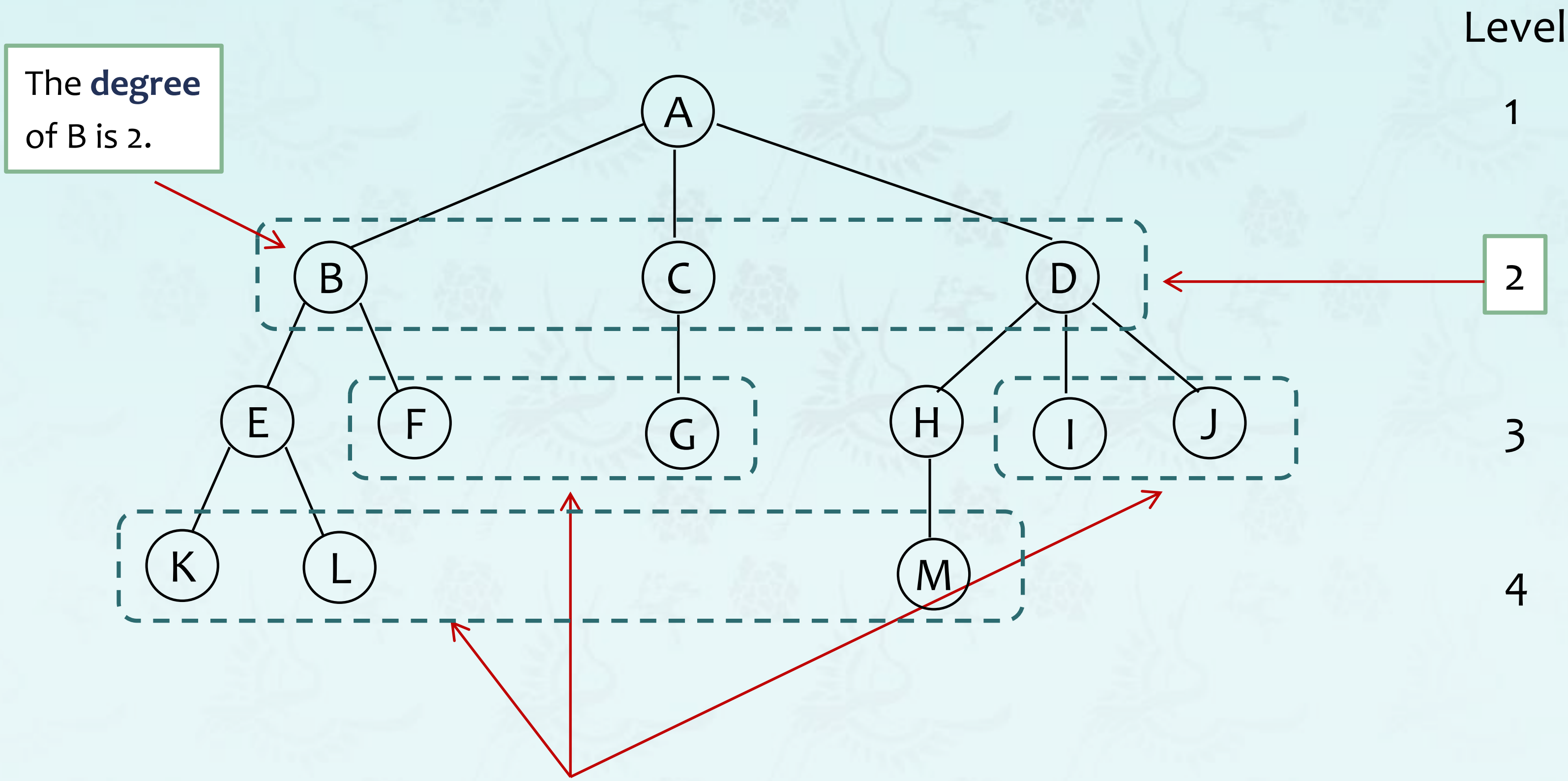
Example. A **tree** with a root storing the value 'A'



- The **children** of **D** are **H, I, and J**; **H, I, and J** are **siblings**.
- The **parent** of **D** is **A**.

Chapter 5.1 Introduction - Terminology

Definition. child, parent, sibling, degree, leaf nodes, level, height, internal node



- Zero degree nodes are **leaf nodes**, all others are **internal nodes.**
- The **degree** of a node is the number of children.
- The **degree of a tree** is the **maximum of the degree of the nodes** in the tree.
- The **height** or **depth** of a tree is the max level of any nodes in the tree.

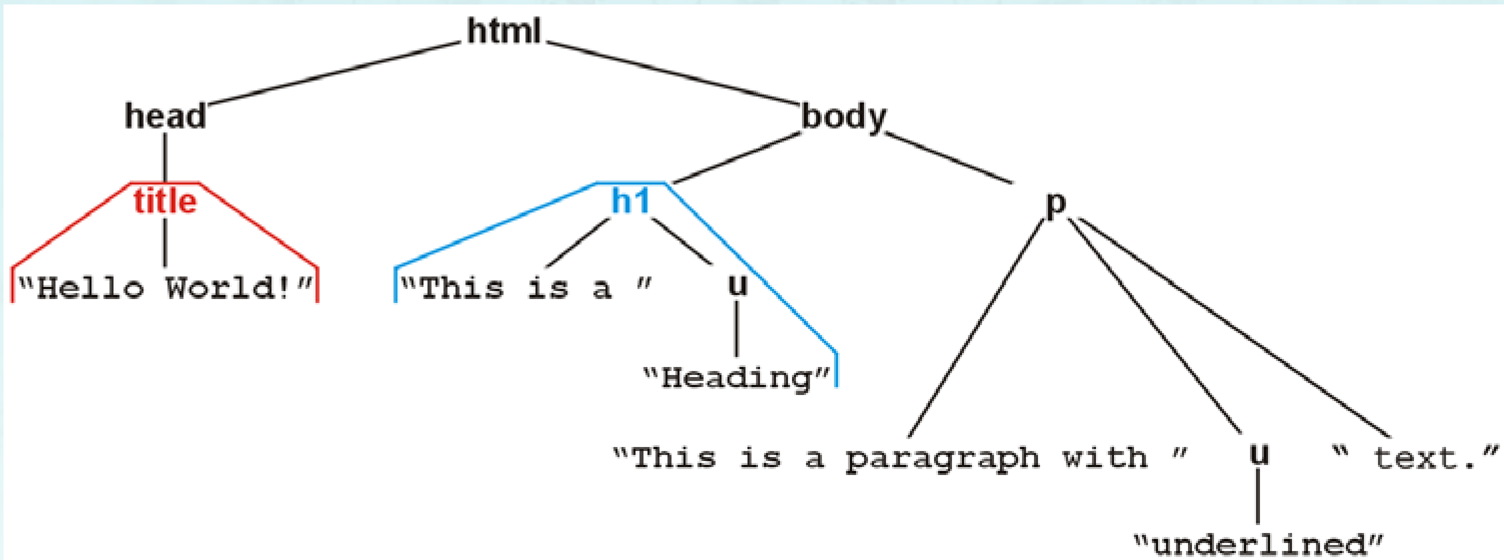
Chapter 5.1 Introduction – Representation of trees

Exercise. The tree representing the HTML document below?

```
<html>
   <head>
      <title>Hello World!</title>
   </head>
   <body>
      <h1>This is a <u>Heading</u></h1>
      <p>This is a paragraph with some <u>underlined</u> text.</p>
   </body>
</html>
```

Chapter 5.1 Introduction – Representation of trees

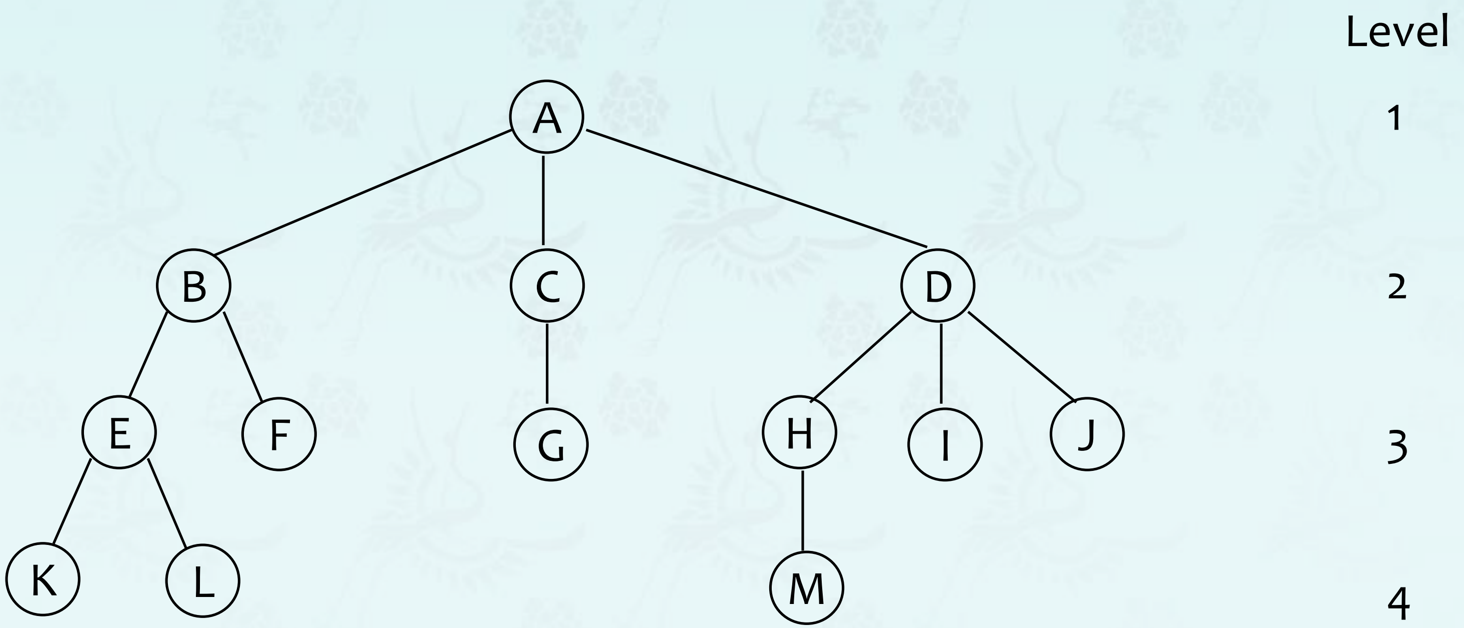
Exercise. The tree representing the HTML document below?




```
<html>
  <head>
    <title>Hello World!</title>
  </head>
  <body>
    <h1>This is a <u>Heading</u></h1>
    <p>This is a paragraph with some <u>underlined</u> text.</p>
  </body>
</html>
```

Chapter 5.1 Introduction – Representation of trees

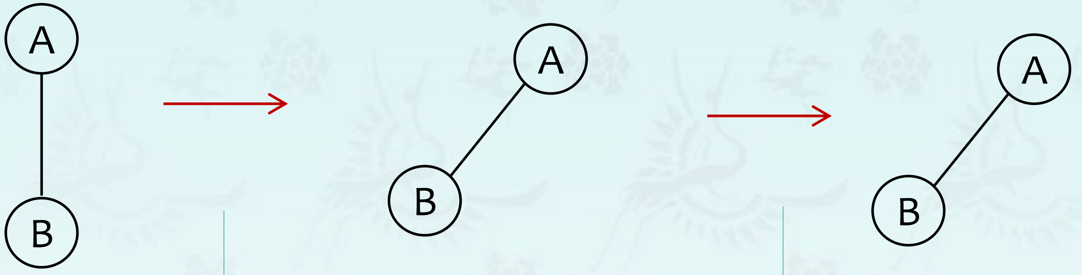
❖ **List representation:** **(A (B (E (K, L), F), C (G), D (H (M), I, J)))**



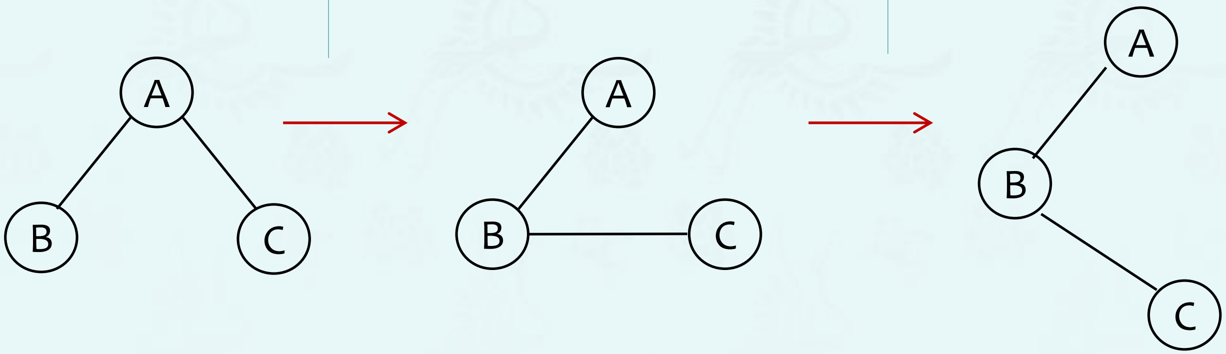
Chapter 5.1 Introduction – Representation of trees

❖ Left child-right child tree representation:

- Rotate the tree **clockwise by 45 degree.** Why?
- To obtain the degree-two tree.
- Note: The root of the tree can never have a sibling.



tree | Left child-right sibling tree | **binary tree**



Data Structures

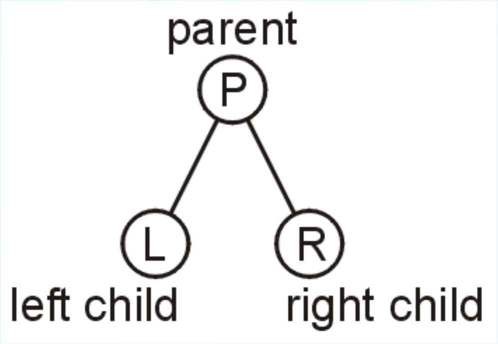
Chapter 5

- *introduction*
- ***binary tree***
- *priority queues & heaps*
- *binary search tree*

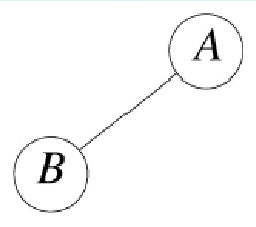
Chapter 5.2 Binary trees

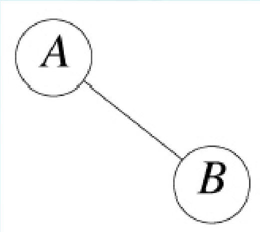
Definition: A tree such that each node has *exactly* two children.

- Notice, exactly two children - not up to two children!
(because *exactly* two children means a left child **and/or** right child, no middle child.)
- Each child is either empty or another binary tree.
- Given this constraint, we can label the two children as left and right nodes or subtrees.



Example: two binary trees with two nodes

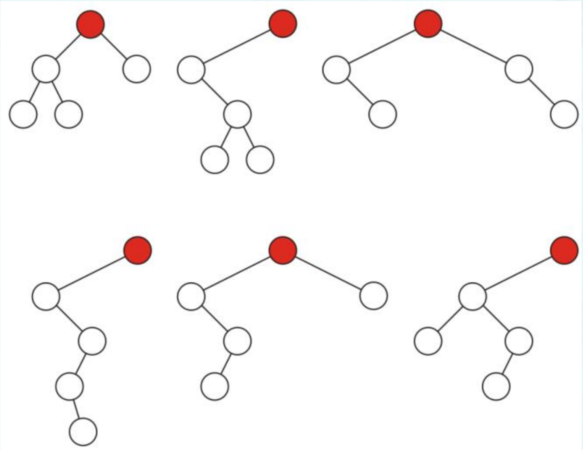




Q: are they two different **binary** trees?

A: Yes!

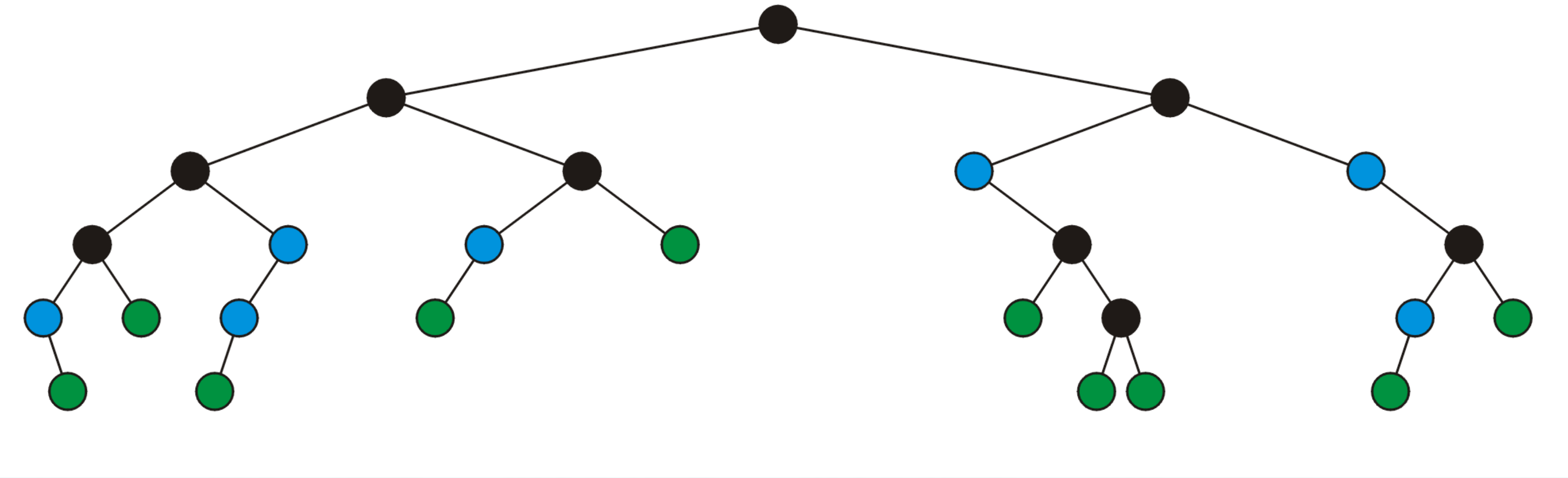
Example: five binary trees with five nodes.



Definition: A **full node** is a node where both left **and** right sub-trees are non-empty trees:

Q: how many full nodes are there?

Q: how many leaf nodes are there?

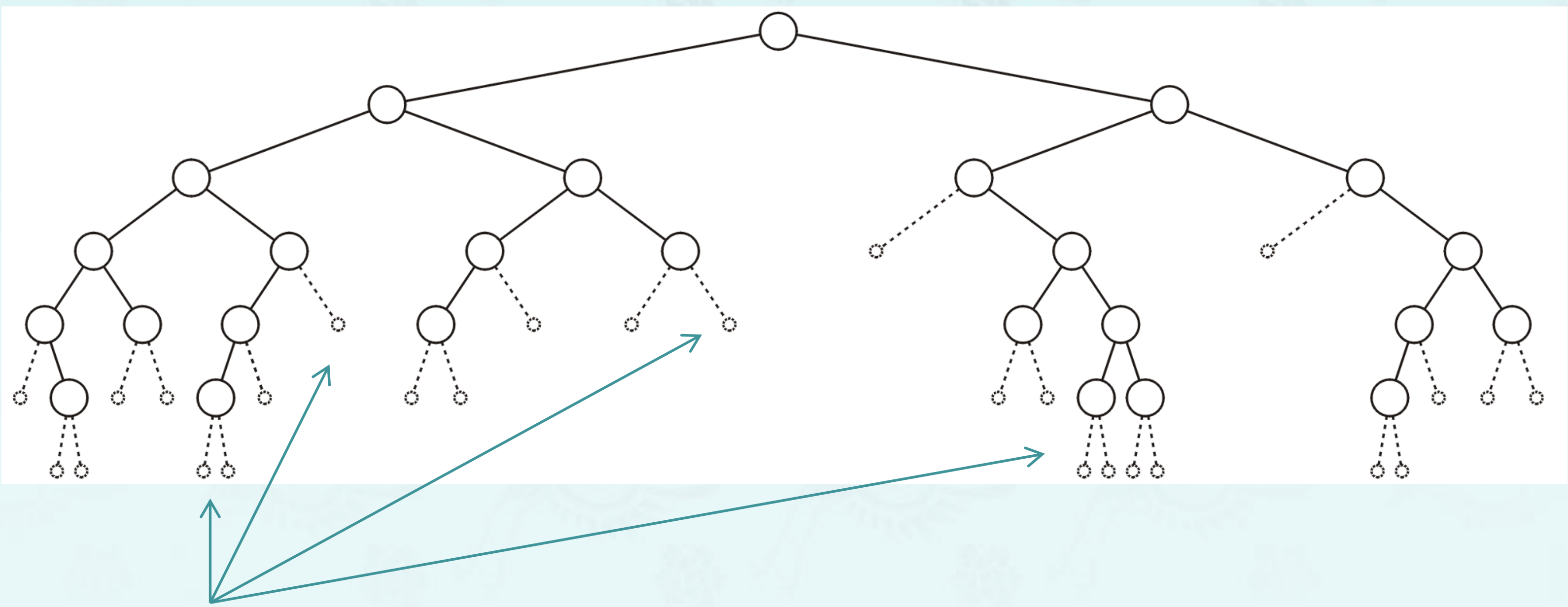


full nodes leaf nodes neither

Q: What is the height of the tree?

Q: What is the degree of the tree?

Definition: An ***empty node*** or ***null sub-tree*** is a location where a new leaf node (or a sub-tree) could be inserted.



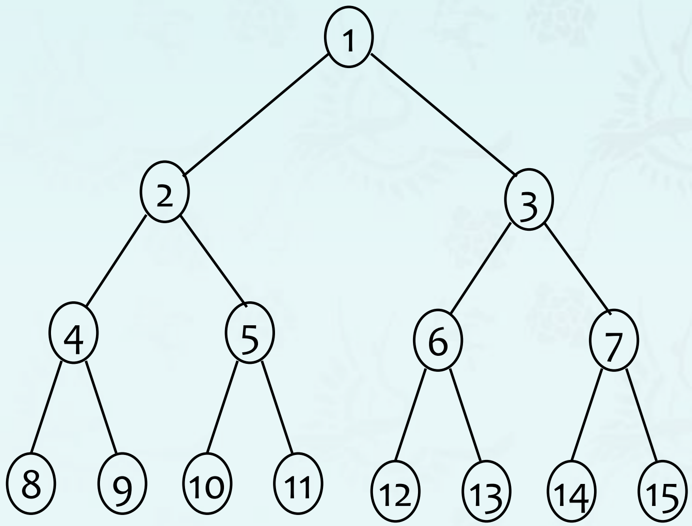
Graphically, the missing branches.

ADT BinaryTree
objects: a finite set of nodes either empty or consisting of a root node, leftBinaryTree, and rightBinaryTree.

functions:
`binaryTree newTree()`
`boolean isEmpty(bt)`
`binaryTree newNode(left, right, item)`
`binaryTree left(bt)`
`element getItem(bt)`
`binaryTree right(bt)`

Observation:

Maximum number of nodes in binary trees in each level and all levels?



Height	Nodes at one level	Nodes at all levels
1	$2^0 = 1$	$1 = 2^1 - 1$
2	$2^1 = 2$	$3 = 2^2 - 1$
3	$2^2 = 4$	$7 = 2^3 - 1$
4	$2^3 = 8$	$15 = 2^4 - 1$
.	.	.
11	$2^{10} = 1024$	$2047 = 2^{11} - 1$
.	.	.
h		

Chapter 5.2 Binary trees - Properties

(1) The maximum number of **nodes on level i** of a binary tree is

$i \geq 1$

(2) The maximum number of **nodes in a binary tree of depth k** is

$k \geq 1$

(3) The depth(height) of a **complete binary tree** with **n** nodes is

$\lceil \quad \rceil$, $\lceil x \rceil$ is the smallest integer $\geq x$.

(1) The maximum number of **nodes on** $\boldsymbol{level\ i}$ of a binary tree is

$$2^{i-1}, \qquad i \geq 1$$

(2) The maximum number of **nodes in a binary tree of** $\boldsymbol{depth\ k}$ is

$$2^k - 1, \qquad k \geq 1$$

Proof (1) by induction on i.

Induction base:

On $\boldsymbol{level\ i\ =\ 1}$, the root is the only node. Hence, 2^{i-1} = 2^{1-1} = 2^0 = 1.
which is the maximum number of nodes on $level\ i = 1$ → 1
On $\boldsymbol{level\ i\ =\ 2}$, → $2^{2-1} = 2$

Induction hypothesis:

Assume that the maximum number of nodes on $\boldsymbol{level\ i-\ 1}$ is $\boldsymbol{2^{i-2}}$.

Induction step:

Since on $\boldsymbol{level\ i\ -1}$ → 2^{i-2} by hypothesis and
each node has a maximum $degree\ of\ 2$,
the maximum number of nodes on $\boldsymbol{level\ i}$ is $2 * 2^{i-2}$, or $\boldsymbol{2^{i-1}}$

(1) The maximum number of **nodes on *level i*** of a binary tree is

$$2^{i-1}, \qquad i \geq 1$$

(2) The maximum number of **nodes in a binary tree of *depth k*** is

$$2^k - 1, \qquad k \geq 1$$

Proof (2) Using geometric summation:

The maximum number of nodes in a binary tree of ***depth k*** is "the summation of the maximum number of nodes on every level".

$$\sum_{i=1}^{k} (maximum\ numer\ of\ nodes\ on\ level\ i) = \underbrace{2^0}_{level\ 1} + 2^1 + \ldots + \underbrace{2^{i-1}}_{level\ k}$$

(1) The maximum number of **nodes on $\boldsymbol{level\ i}$** of a binary tree is

$$2^{i-1}, \qquad i \geq 1$$

(2) The maximum number of **nodes in a binary tree of $\boldsymbol{depth\ k}$** is

$$2^{k} - 1, \qquad k \geq 1$$

Proof (2) Using geometric summation:
The maximum number of nodes in a binary tree of $\boldsymbol{depth\ k}$ is "the summation of the maximum number of nodes on every level".

$$\sum_{i=1}^{k} (maximum\ numer\ of\ nodes\ on\ level\ i) = 2^0 + 2^1 + \ \ldots + 2^{i-1} = \sum_{i=1}^{k} 2^{i-1} = \mathbf{2^k - 1}$$

$$\sum_{i=0}^{n} a^i = 1 + a + a^2 + \cdots + a^n = \frac{a^{n+1} - 1}{a - 1}$$

$$1 + 2 + 2^2 + \cdots + 2^{n-1} + 2^n = \frac{2^{n+1} - 1}{2 - 1}$$

$$\begin{aligned} 1 + 2 + 2^2 + \cdots + 2^{n-1} &= 2^{n+1} - 1 - 2^n \\ &= 2^n(2 - 1) - 1 \\ &= \mathbf{2^n - 1} \end{aligned}$$

Chapter 5.2 Binary trees - Properties

(1) The maximum number of **nodes on** $\boldsymbol{level\ i}$ of a binary tree is

$$2^{i-1}, \qquad i \geq 1$$

(2) The maximum number of **nodes in a binary tree of** $\boldsymbol{depth\ k}$ is

$$2^k - 1, \qquad k \geq 1$$

Something significant? The depth of a full binary tree of *n* nodes is $\Theta(\log n)$:

Many operations with trees have a run time that goes with the **depth** of some path within the tree; if we have a full binary tree (or something *close* to it), we know that those operations run in $\boldsymbol{O}(\log \boldsymbol{n})$.

Proof:

$n = 2^k - 1$

$n + 1 = 2^k$

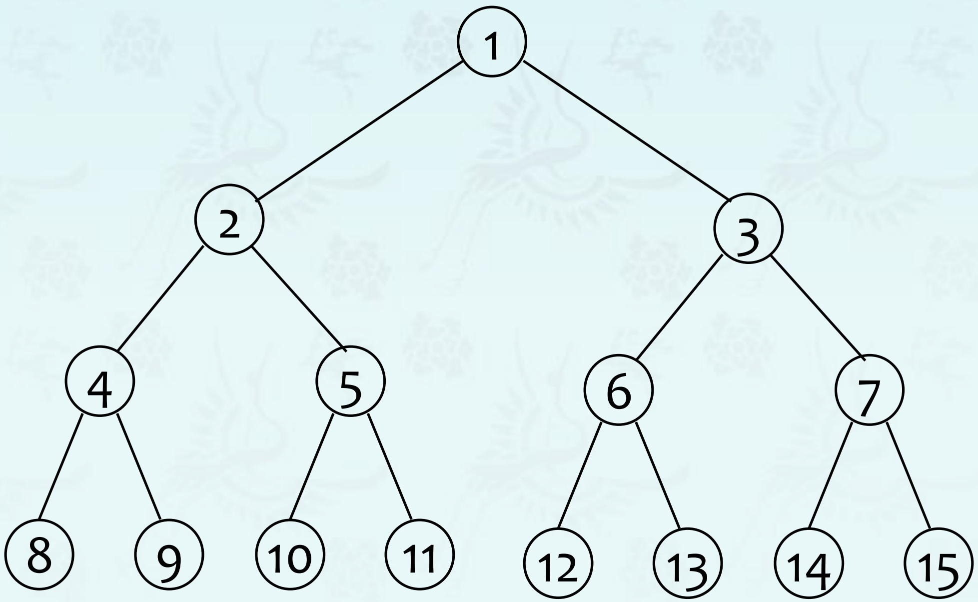
$log_2(n+1) = log_2(2^k)$

$log_2(n+1) = k$

$\Theta(\log n) = k$

Definition: *A **full** binary tree* of ***depth k*** is a binary tree having **$2^k – 1$ nodes**, $k \geq 0$.

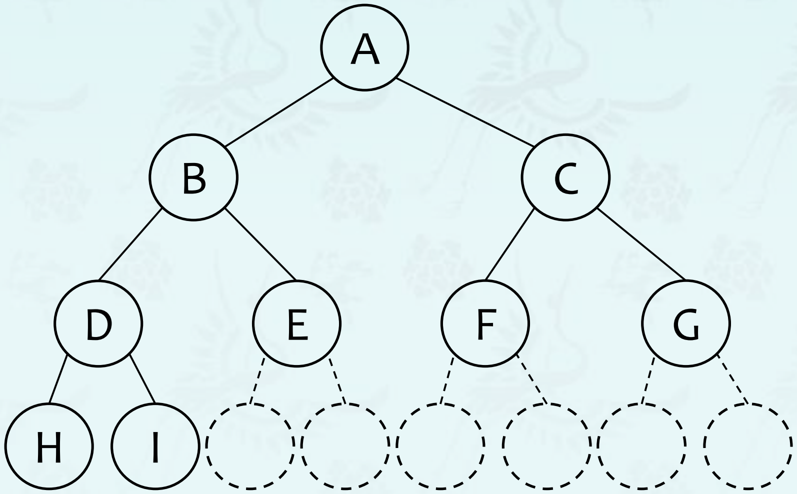
Definition: A binary tree with n nodes and $depth\ k$ is **complete** iff its nodes correspond to the nodes numbered from 1 to n in the full binary tree of $depth\ k$.



*A **full** binary tree*

Definition: *A **full** binary tree* of depth k is a binary tree of depth k having $\mathbf{2^k - 1}$ nodes, $k \geq 0$.

Definition: A binary tree with n nodes and $depth\ k$ is **complete iff** its nodes correspond to the nodes numbered from 1 to n in the full binary tree of $depth\ k$.



*A **complete** binary tree*

(3) The height of a **complete binary tree** with n nodes is

$\lceil log_2 (n+1) \rceil$, $\lceil x \rceil$ is the smallest integer $\geq x$.

Proof (3): The maximum number of **nodes** $\boldsymbol{n}$ of a binary tree with its $height\ k$ or $depth\ k$ is $2^k - 1$, $k \geq 1$.

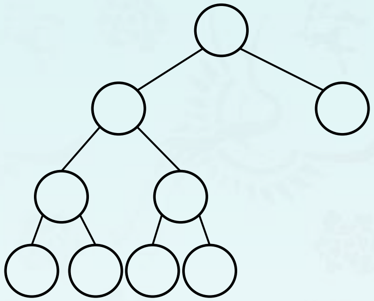
In a binary tree, it has the maximum number of nodes n of a $n = 2^k - 1$.

$$n = 2^k - 1, \quad for\ k \geq 1,$$
$$2^k = n + 1$$
$$log_2 (2^k) = log_2 (n+1)$$
$$k = log_2 (n+1)$$

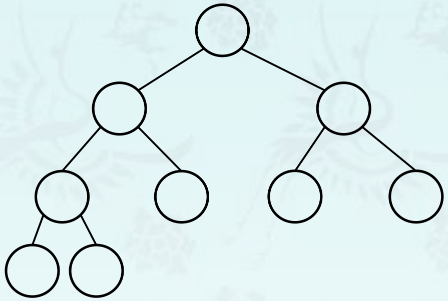
$k = \lceil log_2 (n+1) \rceil$ since k is an integer, to include incomplete trees.

Definition: A binary tree with n nodes and depth k is **complete iff** its nodes correspond to the nodes numbered from 1 to n in the full binary tree of depth k.

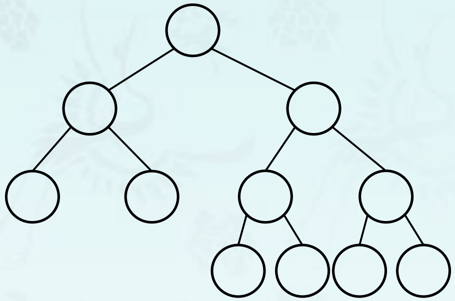
Exercise: identify a **complete** binary tree.



(1)



(2)

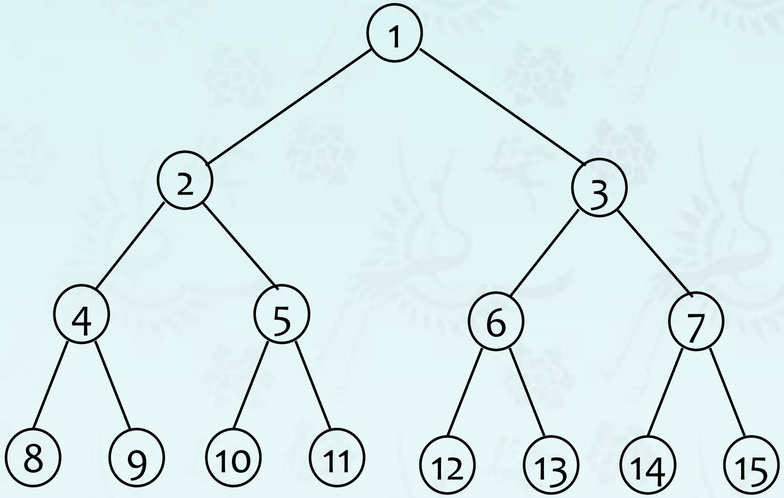


(3)

Q. Meanings of a complete tree in terms of ADT?

A. Removals of a node are only allowed from the "last" position.
There is one position available to insert a node every time!

Problem: representing a binary tree in memory
Hint: remembering a full binary tree with sequential node numbers



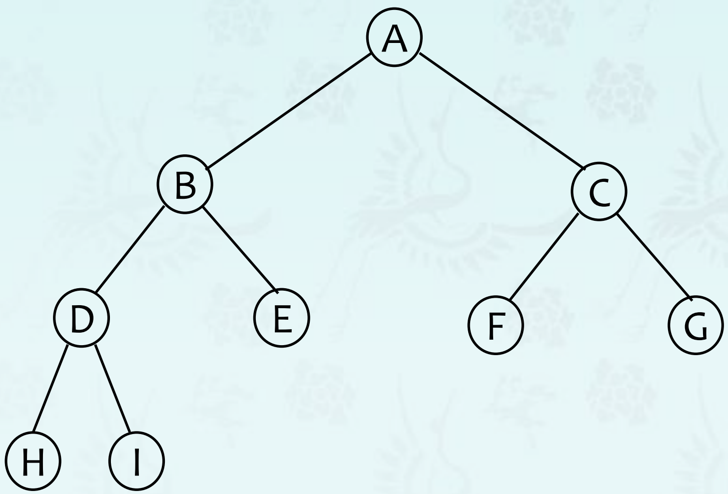
Solution: use one dimensional array to store nodes sequentially.

Any potential problems?

Problems remain: good for a full binary tree, but not good memory usage for a skewed or complete binary tree.

Chapter 5.2 Binary trees – Array representation

Problem: Let's suppose that you have a **complete binary tree** in an array, how can we locate node i's parent or child?
Example: Find its parent, left child and right child at node D.



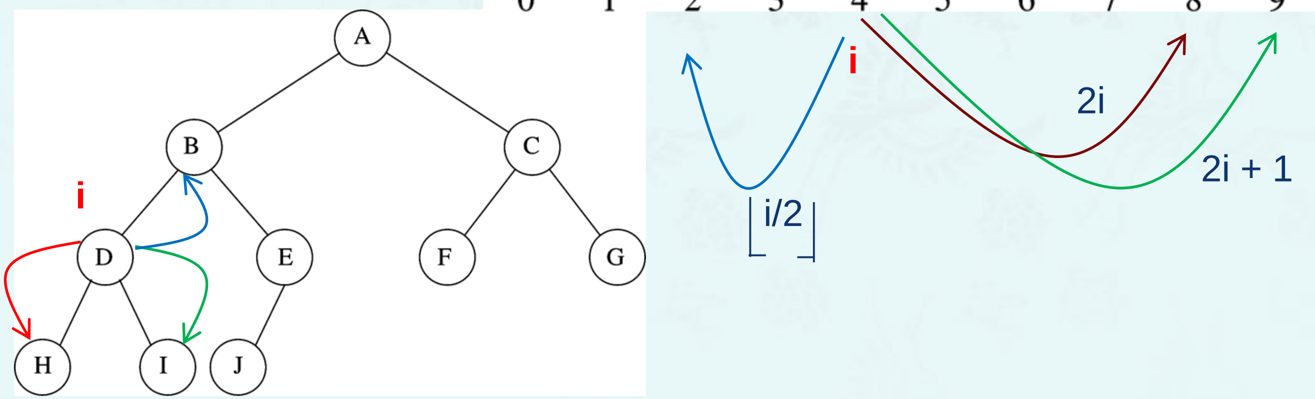
[0]	-
[1]	A
[2]	B
[3]	C
[4]	D
[5]	E
[6]	F
[7]	G
[8]	H
[9]	I

Example: Find its parent, left child and right child at node D.

Lemma 5.4 a **complete** binary tree with n nodes, any node index i, $1 \leq i \leq n$, we have

- Given element at position i in the array
  - Left $child(i)$ = at position $2i$
  - Right $child(i)$ = at position $2i + 1$
  - $Parent(i)$ = at position $\lfloor i/2 \rfloor$

	A	B	C	D	E	F	G	H	I	J			
0	1	2	3	4	5	6	7	8	9	10	11	12	13



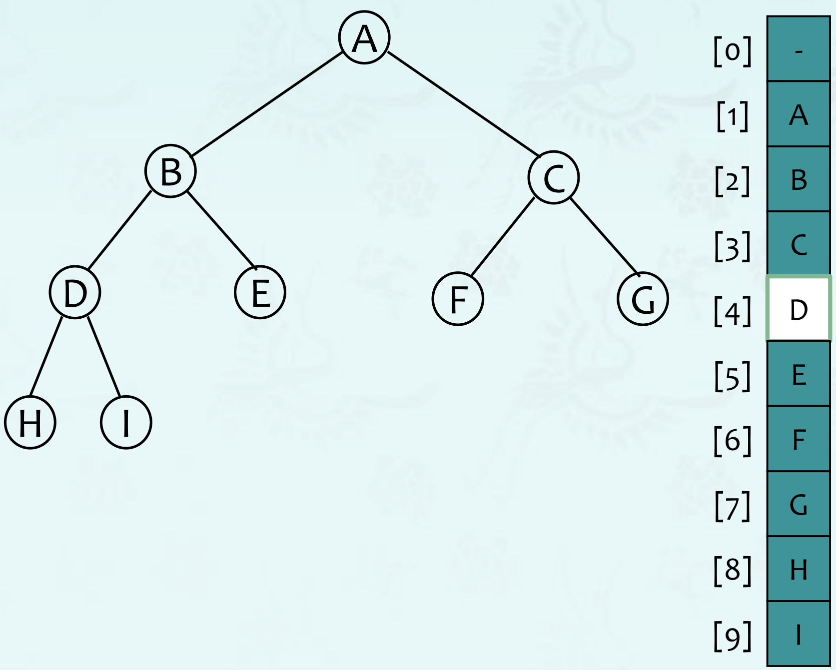
Example: Find its parent, left child and right child at node D.

Lemma 5.4 a **complete** binary tree with n nodes, any node index i, $1 \leq i \leq n$, we have

(1) $parent(i)$ is at $\lfloor i/2 \rfloor$ $if\ i = 1.$ $If\ i = 1,$ i is at the root and has no parent.

(2) $leftChild(i)$ is at $2i$ $if\ 2i <= n.$ $If\ 2i > n,$ then i has no left child.

(3) $rightChild(i)$ is at $2i + 1$ $if\ 2i + 1 <= n.$ $If\ 2i + 1 > n,$ then i has no right child.



Solution:

$parent(i = 4)$ is at 4/2 = 2

$leftChild(4)$ is at 2x4 = 8

$rightChild(4)$ is at 2x4 + 1 = 9

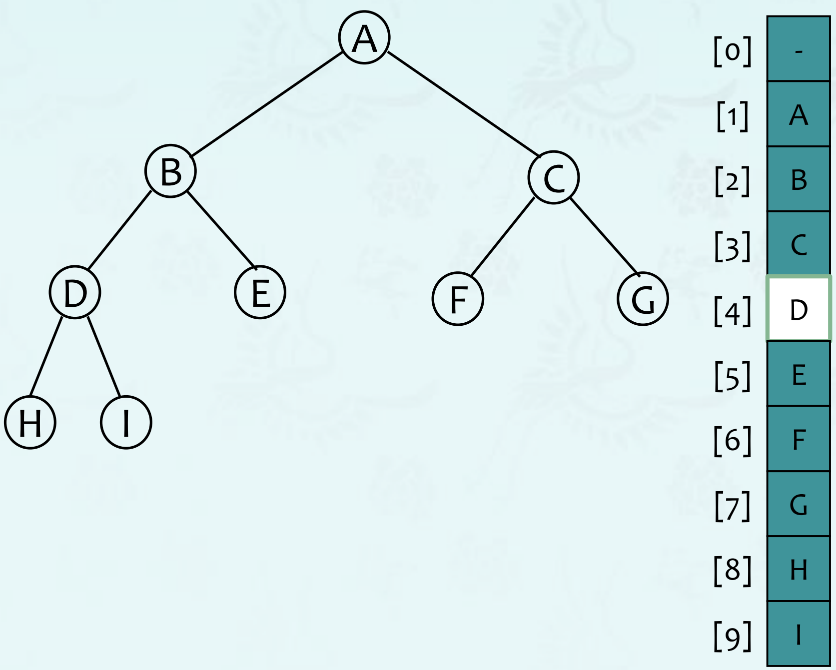
Wow!
Can we use this to all binary trees?
Why not?

Example: Find its parent, left child and right child at node D.

Lemma 5.4 a **complete** binary tree with n nodes, any node index i, $1 \leq i \leq n$, we have

Wow!
Can we use this to all binary trees?
Why not?

Problem remains:
The problem with storing an arbitrary binary tree using an array is the inefficiency *in memory usage*.

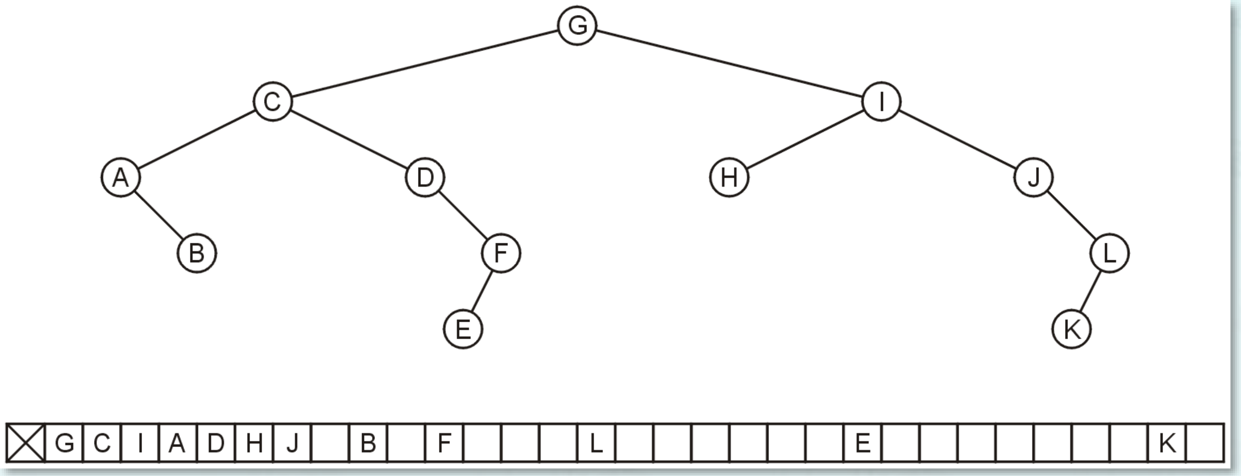


Chapter 5.2 Binary trees – Array representation

Q. Can we use this array rep. to store all binary trees? Why not?

Example: This tree has 12 nodes, and requires an array of 32 elements.

A. Adding one extra node, as a child of node K **doubles** the required memory for the array!



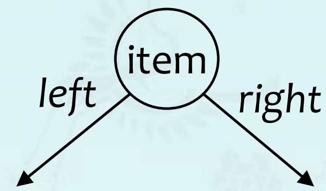
A. In the worst case a skewed tree of depth k will require **$2^k - 1$** space which is O(**2^k**). Of these, **only k** will be used.

Q. What happens when k = n? (Is there such a tree?)

Chapter 5.2 Binary trees – Linked representation

Node representations:

left	item	right



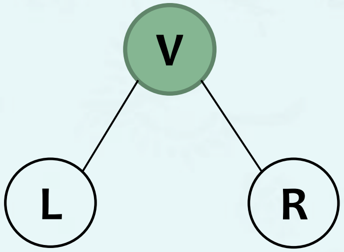

```
typedef struct node *pTree;
typedef struct node{
    int   item;
    pTree left
    pTree right;
}node;
```

Q. Is this node structure in C good enough?

A. Not easy to find its parent node.
Parent field could be added if necessary

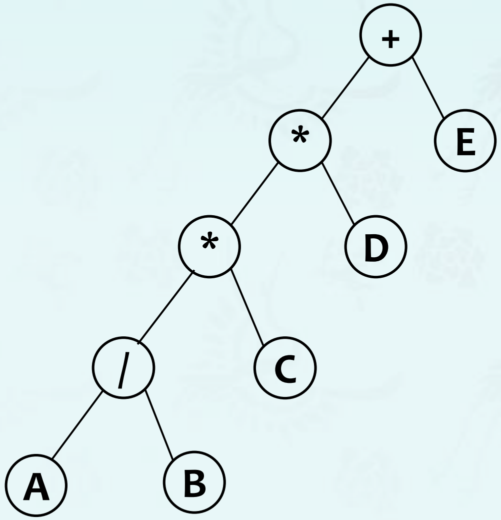
Tree traversal (known as **tree search**) refers to the process of visiting each node in a tree, exactly once, in a systematic way.

- There are three possible moves if we traverse left before right: **LVR, LRV, VLR.**
- These are named **inorder, postorder,** and **preorder** because of the position of the **V** (visiting node) with respect to the L and R.
- There are three types of **depth-first traversal.**



Example: inorder traversal(LVR)

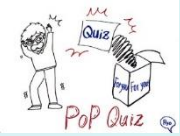
- Moving down the tree toward the left until you can go no farther. Then you "visit" the node, move one node to the right and continue. If you cannot move to the right, go back one more node.




```
void inorder(pTree ptr) {
   if (ptr) {
       inorder(ptr->left);
       printf("%c", ptr->item);
       inorder(ptr->right);
   }
}
```

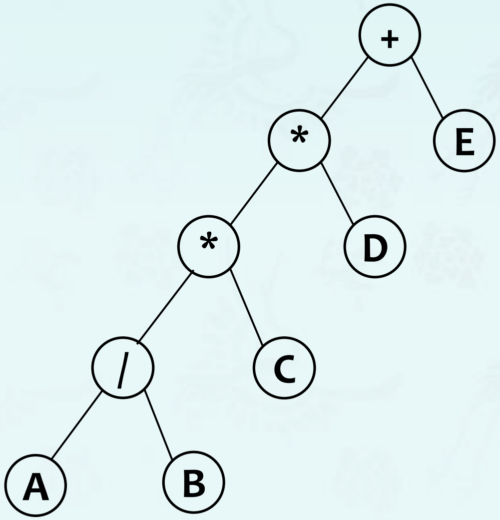
Output:

Output(LVR) : A / B * C * D + E



Example: inorder traversal(LVR)

Q: How many times is inorder( ) invoked for the complete traversal?



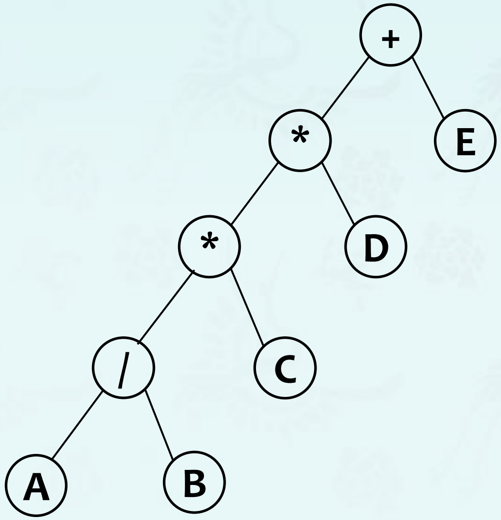

```
void inorder(pTree ptr) {
   if (ptr) {
       inorder(ptr->left);
       printf("%c", ptr->item);
       inorder(ptr->right);
   }
}
```

Output:

Output(LVR) : A / B * C * D + E

Example: inorder traversal(LVR)

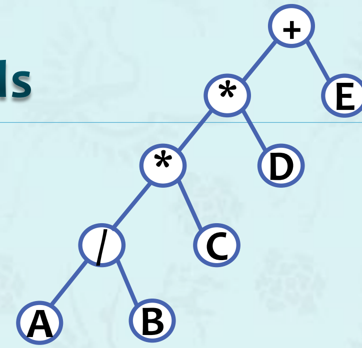
Note: "Since there are 9 nodes in the tree, inorder is invoked 19 times for the complete traversal." (p.207) This is not a typo.




```
void inorder(pTree ptr) {
   if (ptr) {
        inorder(ptr->left);
        printf("%c", ptr->item);
        inorder(ptr->right);
   }
}
```

A. **Every leaf node** node must visit (call the function) its left child and right child to make sure they don't have the child. 9 + 5 * 2 = 19

Chapter 5.3 Binary tree traversals

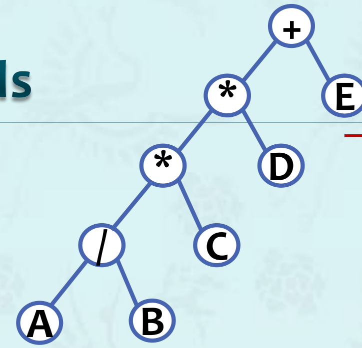



```
void inorder(pTree ptr){
  if (ptr) {
    inorder(ptr->left);
    printf("%c", ptr->item);
    inorder(ptr->right);
  }
}
```

Example: inorder traversal(LVR)

Call of inorder	**ptr** or **ptr->item**	Action	inorder	**ptr** or **ptr->item**	Value Action
1	+		11	C	
2	*		12	NULL	
3	*		11	C	**printf**
4	/		13	NULL	
5	A		2	*	**printf**
6	NULL		14	D	
5	A	**printf**	15	NULL	
7	NULL		14	D	**printf**
4	/	**printf**	16	NULL	
8	B		1	+	**printf**
9	NULL		17	E	
8	B	**printf**	18	NULL	
10	NULL		17	E	**printf**
3	*	**printf**	19	NULL	

Chapter 5.3 Binary tree traversals




```
void inorder(pTree ptr){
  if (ptr) {
    inorder(ptr->left);
    printf("%d", ptr->item);
    inorder(ptr->right);
  }
}
```

Example: inorder traversal(LVR)

Call of inorder		ptr or ptr->item		Action	inorder	ptr or ptr->item	Value Action
1		+	1.push				
2		*	2.push				
3		*	3.push				
4		/	4.push				
5		A	5.push				
6		NULL	return				
5	1.pop	A		printf			
7		NULL	return				
4	2.pop	/		printf			
8		B	6.push				
9		NULL	return				
8	3.pop	B		printf			
10		NULL	return				
3	4.pop	*		printf			

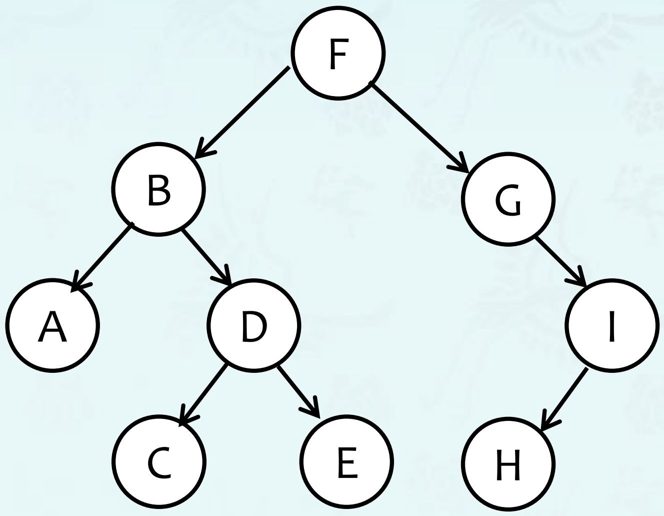
System Stack

5.push(A) 1.pop
4.push(/) 2.pop 6.push(B) 3.pop
3.push(*) 4.pop
2.push(*)
1.push(+)

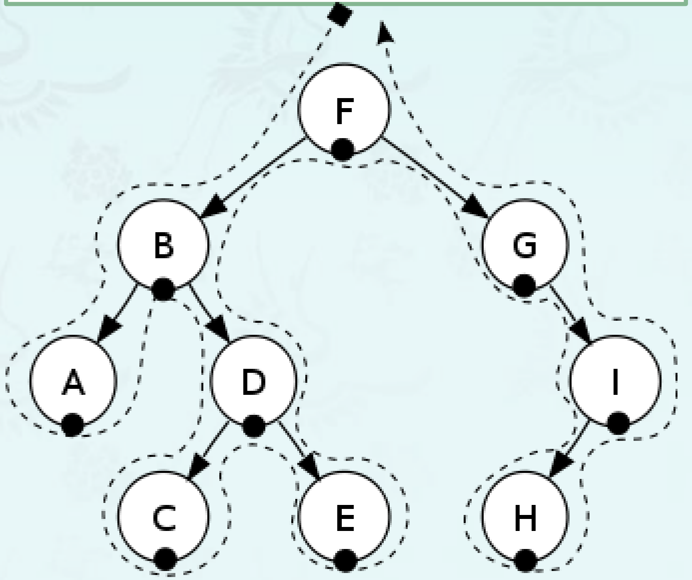
Example: inorder traversal(LVR)

1. Traverse the left subtree.
2. Visit the root.
3. Traverse the right subtree.

Exercise: Output?

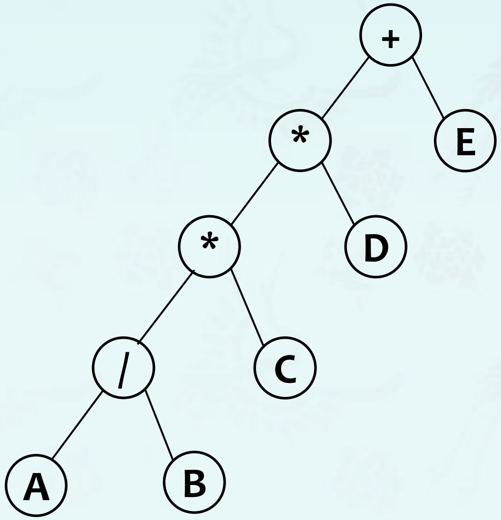


A: A, B, C, D, E, F, G, H, I



Example: preorder traversal(VLR)

- Visit a node, traverse left, and continue. When you cannot continue, move right and begin again or move back until you can move right and resume.




```
void preorder(pTree ptr) {
   if (ptr) {
        printf("%c", ptr->item);
        preorder(ptr->left);
        preorder(ptr->right);
   }
}
```

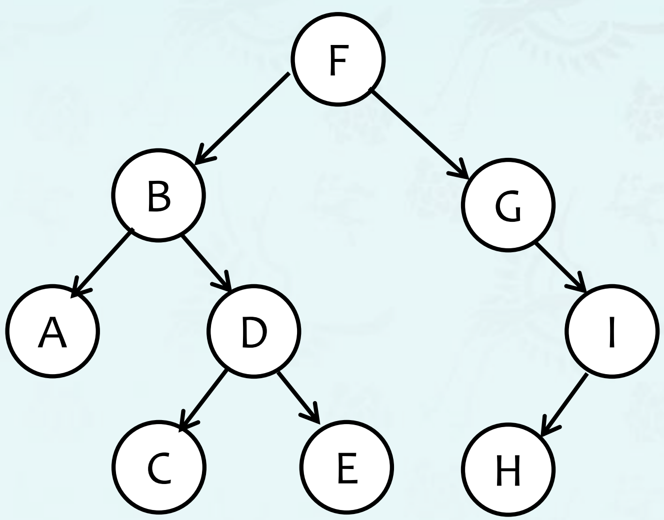
Output:

Output(LVR) : + * * / A B C D E

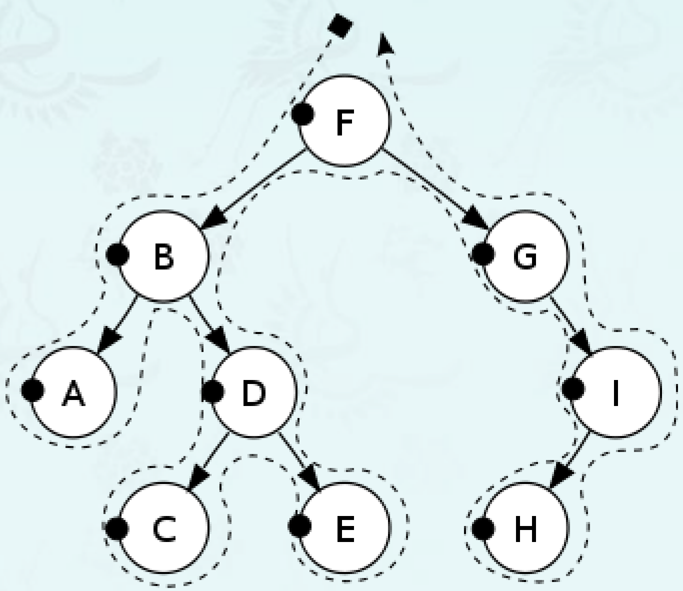
Example: preorder Traversal(VLR)

1. Visit the root.
2. Traverse the left subtree.
3. Traverse the right subtree

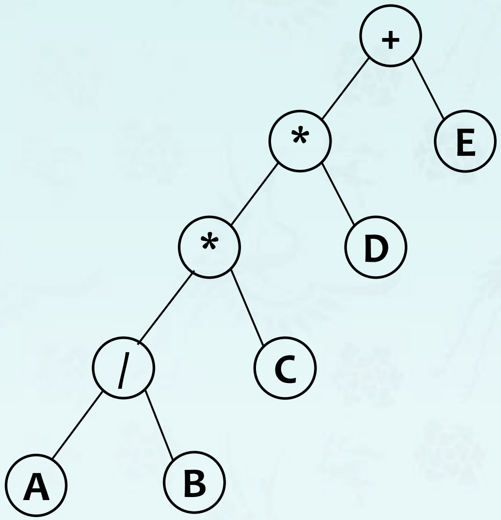
Exercise: Output?



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



Example: postorder traversal(LRV)




```
void postorder(pTree ptr) {
   if (ptr) {
       postorder(ptr->left);
       postorder(ptr->right);
       printf("%c", ptr->item);
   }
}
```

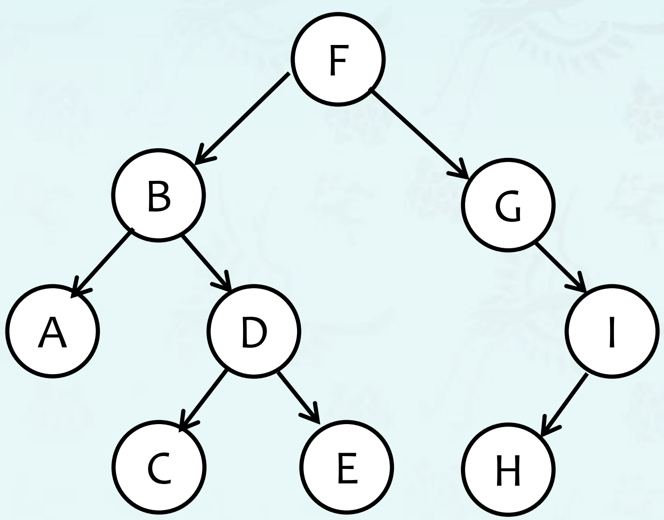
Output:

Output(LVR) : A B / C * D * E +

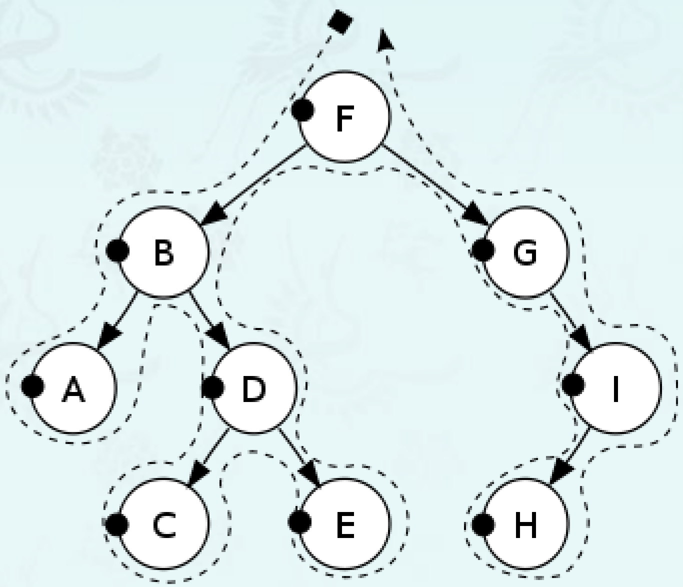
Example: postorder traversal(LR V)

1. Traverse the left subtree.
2. Traverse the right subtree.
3. Visit the root.

Exercise: Output?



A: A C E D B H I G F



Iterative inorder traversal:
Using a **stack** is the obvious way to traverse tree without recursion. Below is an algorithm for traversing binary tree using stack.

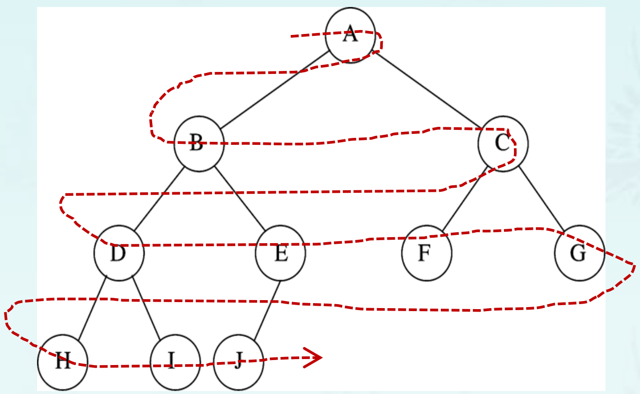
1) Get an empty stack S.
2) Set a node to start to traverse.
3) Push the node to S and set **node = node->left** until the node is NULL
4) If the node is NULL and stack is not empty then
   a) Pop the top item from stack.
   b) Print the popped item, set **node = node ->right**
   c) Go to step 3.
5) If node is NULL and stack is empty then we are done

Iterative inorder traversal: **p.210**

```
void iterativeInorder(pTree node) {  // node to start
   int top = -1;                     // initialize stack
   pTree stack[MAX_STACK_SIZE];// get a stack
   for (;;) {
        for (; node; node = node->left)
            push(node);
        node = pop();
        if (!node) break;
        printf("%d", node->item);
        node = node->right;
   }
}
```

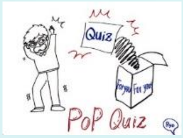
Chapter 5.3 Binary tree traversals

1. **Depth first** search(DFS) – preorder, inorder, postorder traversal
2. **Breadth first** search(BFS) - **level-order** traversal



level-order traversal

1. Visit the root first.
2. The root's left child followed by the root's right child
3. Visit the nodes at each new level from the left most node to the rightmost node.



Summary

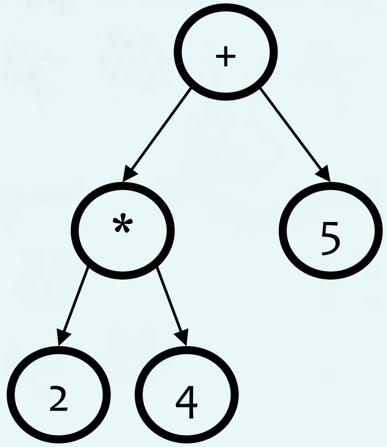
A *traversal* is an order for visiting all the nodes of a tree

- *Pre-order*: + * 2 4 5
- *In-order*: 2 * 4 + 5
- *Post-order*: 2 4 * 5 +

Pre-order: root, left subtree, right subtree

In-order: left subtree, root, right subtree

Post-order: left subtree, right subtree, root

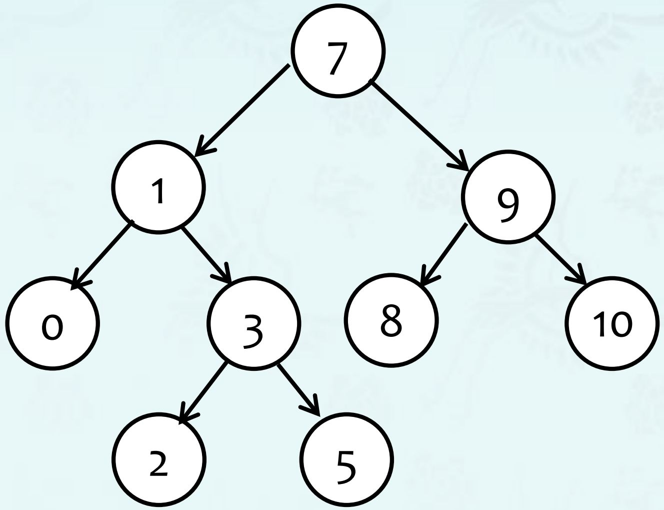


Example:

preorder Traversal(VLR)

inorder traversal(LVR)

postorder traversal(LRV)



Observations:

1. If you know you need to explore the roots before inspecting any leaves, you pick **pre-order** because you will encounter all the roots before all of the leaves.
2. If you know you need to explore all the leaves before any nodes, you select **post-order** because you don't waste any time inspecting roots in search for leaves.
3. If you know that the tree has an inherent sequence in the nodes, and you want to flatten the tree back into its original sequence, than an **in-order** traversal should be used. The tree would be flattened in the same way it was created. A pre-order or post-order traversal might not unwind the tree back into the sequence which was used to create it.

If I wanted to simply print out the hierarchical format of the tree in a linear format, I'd probably use preorder traversal. For example:

```
- ROOT
  - A
    - B
    - C
  - D
    - E
    - F
      - G
```

ECE20010 Data Structures

Chapter 5

- *introduction*
- *binary tree*
- ***priority queues & heaps***
- *binary search tree*