Data structures and algorithms Spring 2025

Lecture 8

Tree part I

Lecturer: Do Thuy Duong

Contents

TREE

What **it is** (<u>conceptual</u>)

Why **we use it** (<u>applications</u>)

How **we implement it** (<u>implementation</u>)

Definition [1]

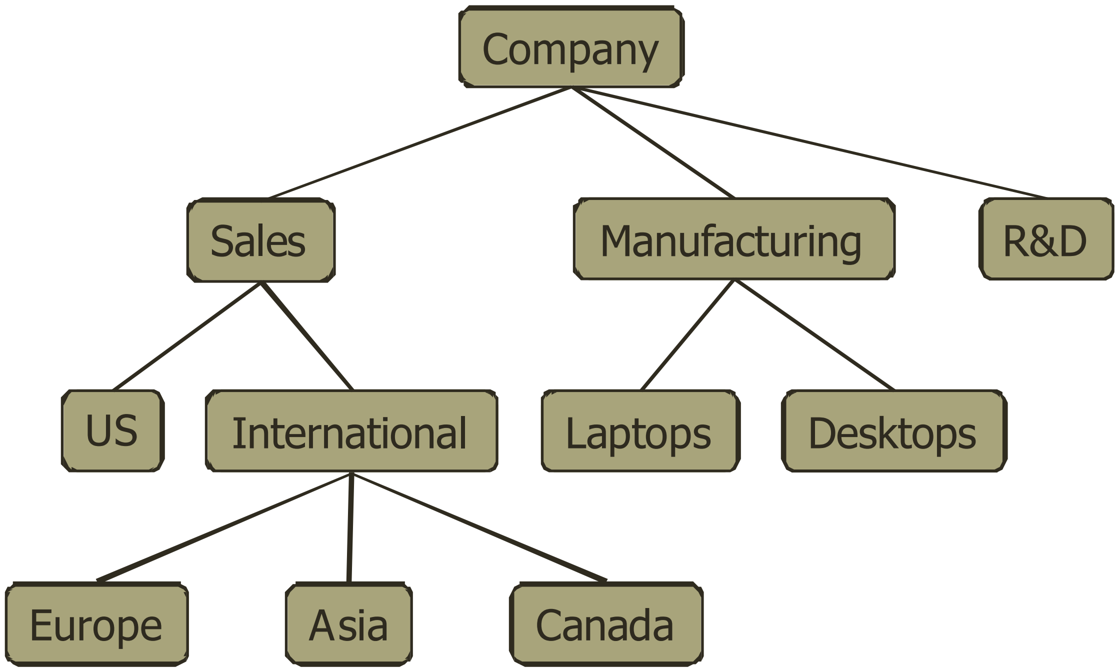
Nature View

Computer Scientists View

This might seem a little simplistic - but it can be quite helpful to keep simple pictures in mind to help you work through the concepts and terminology.

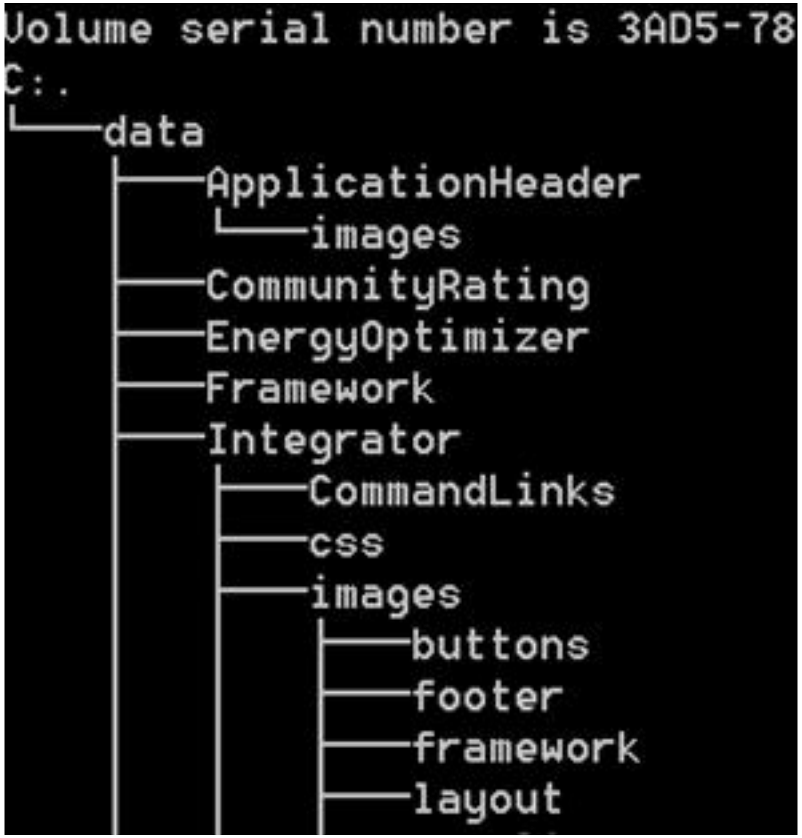
Definition [2]

- **Definition**
  - A tree (CS) is an abstract model of a hierarchical structure



Definition [3]

- A tree consists of **branches** and **nodes** with a **parent-child relation**
- Example:
  - Folder structures
  - Organization structure
  - Domain name structure

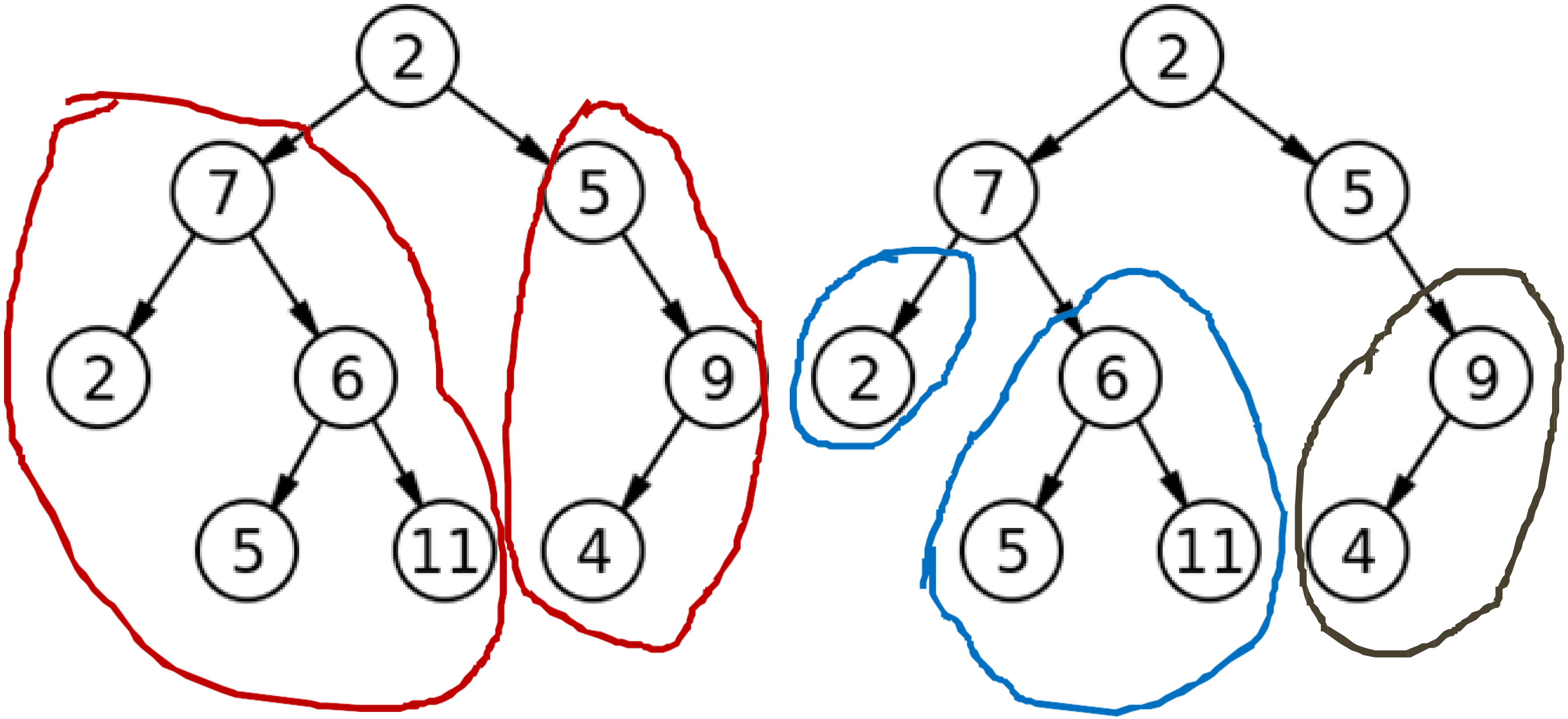


Definition [4]

- **Recursive definition:**
  Tree (T) is a finite set of **one or more nodes** such that:
  - There is one specially designated node called the **root** of the tree.
  - The remaining nodes (excluding the root) are partitioned into m disjoint sets **T1 , T2 , . . . , Tm** and each of these sets **in turn is a tree**.
  - The trees T1 , T2 , . . . , Tm are called the **sub-trees** of the root.

Definition [5]

- **Recursive definition – Tree example**



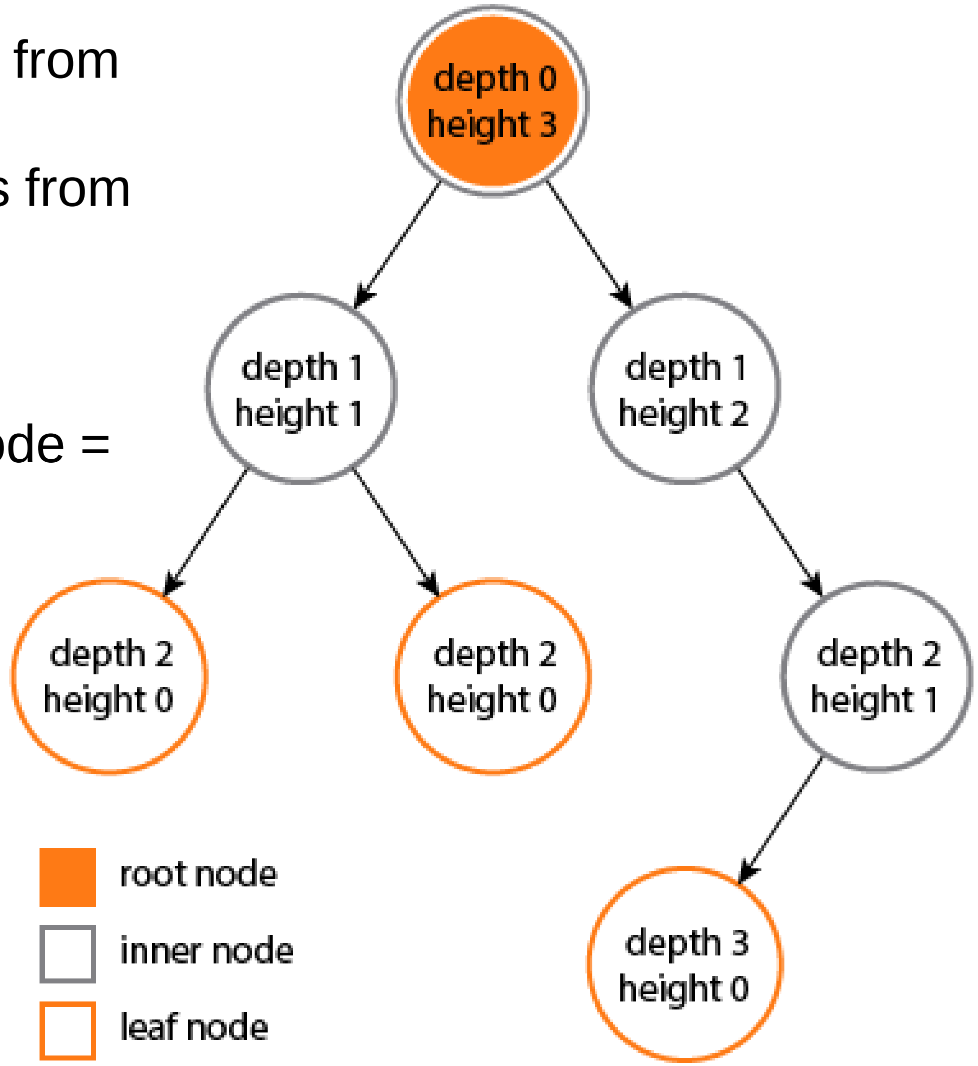
Terminologies [1]

- **A leaf node:**
  A node with no children.
- **A branch node (interior node):**
  A node which is not the root node and a leaf node.
- **Level of node:**
  The distance from the node to the root.
- **Depth of a tree:**
  The maximum level of any leaf in the tree
- **Degree of a node**:
  The number of its children

Terminologies [2]

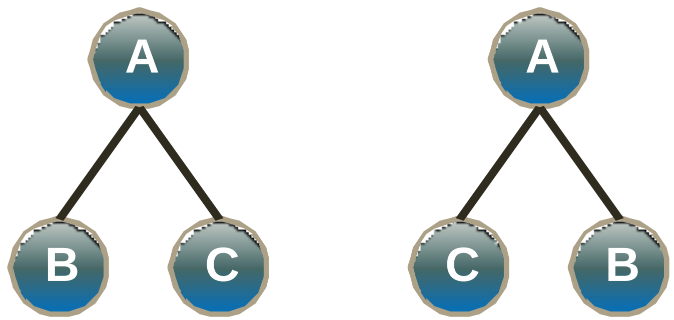
- Depth of a node: the number of edges from the node to the tree's root node.
- Height of a node: the number of edges from the node to the deepest possible leave.

- Height of a tree is height of root node = Depth of a tree is depth of deepest node = maximum level of any leaf in the tree.



Terminologies [3]

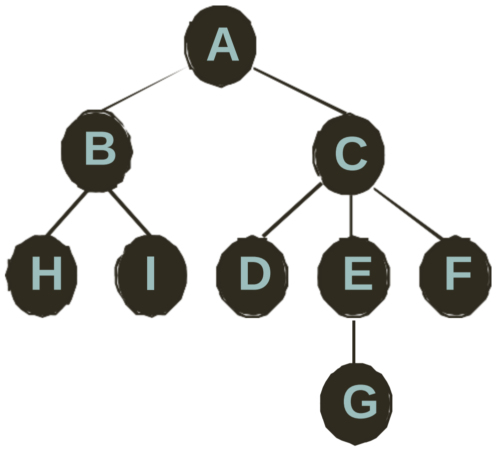
- **Parent and Siblings:**
  - Each root is said to be the parent of its sub-trees
  - Children of the same parent are siblings.
- **Ancestor and Descendant:**
  - Ancestor and descendant can also be used to denote the relationship that may span several level of tree.
- **Node order**
  - The order of the children nodes
  - Default is from left to right
- **Node label (data)**



Terminologies [4]

Example:

Level of node : State the levels of all the nodes:
A:____, B:____, C:____,
D:____, E:____, F:____,
G:____, H:____, I:____



Root of a tree: Root of the tree is: ______

Depth of a tree: Depth of the tree is: ______

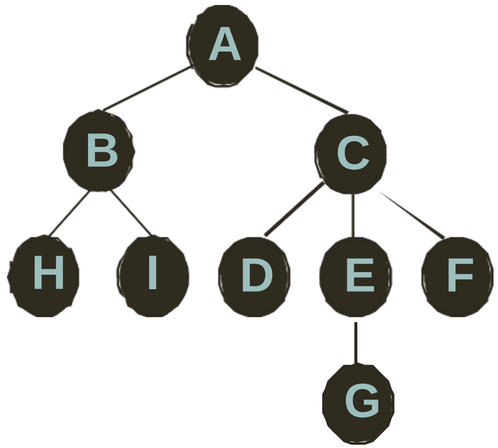
Degree of a node : State the degrees of:
A:____, B:____, C:____,
D:____, E:____, F:____,
G:____, H:____, I:____

Terminal node or leaf: State all the leaf nodes:
____

Branch (interior) node: State all the branch nodes:
____

Terminologies [5]

Parent and Siblings:



State the parents of: A:___, B:___, C:___,
D:___, E:___, F:___,
G:___, H:___, I:___

State the siblings of: A:_______, B:_______,
C:_______, D:_______, E:_______,
F:_______, G:_______, H:_______, I:_______

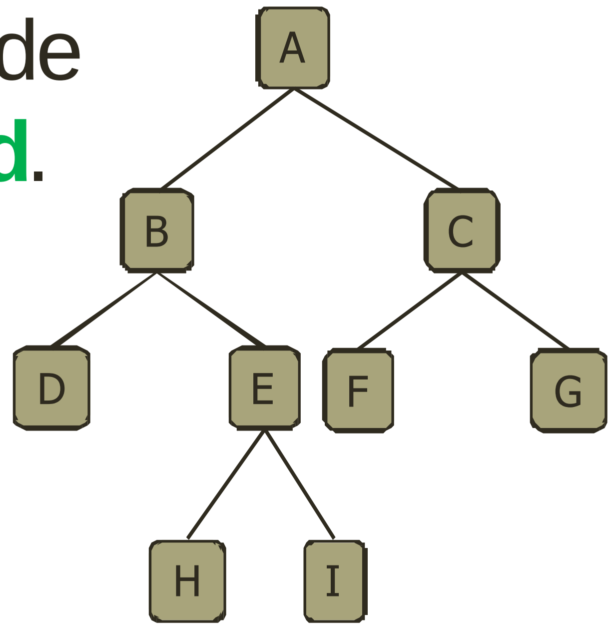
Ancestor and Descendant:

State the ancestors of: A:______, B:______, C:______, D:___________,
E:___________, F:___________, G:___________,
H:___________, I:___________

State the descendants of: A:__________________________,
B:___________, C:___________,
D:____, E:____, F:____, G:____, H:____, I:____

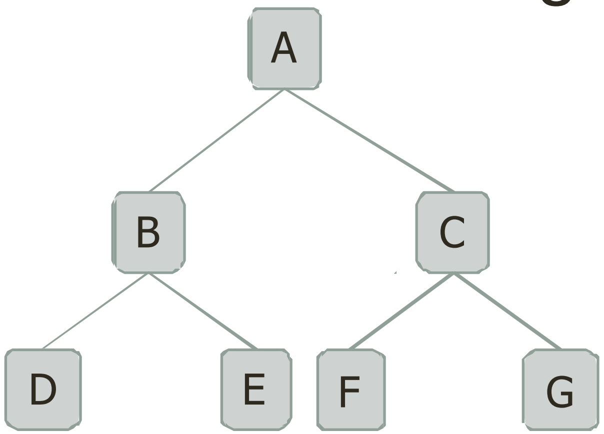
Binarytree [1]

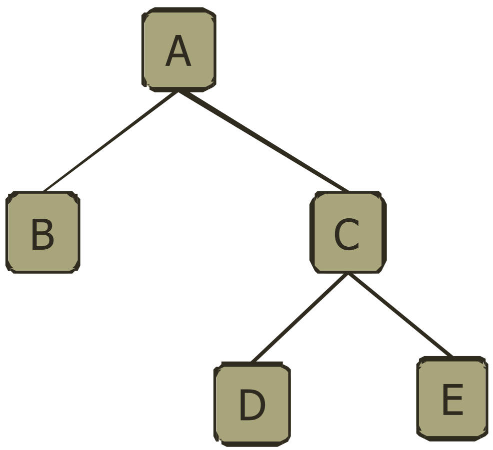
- **Definition:** A **binary tree** is a tree that:
  - Root node and each interior node has at most two children.
  - The children of a node are an ordered pair.
- The children of an internal node called **left child** and **right child**.

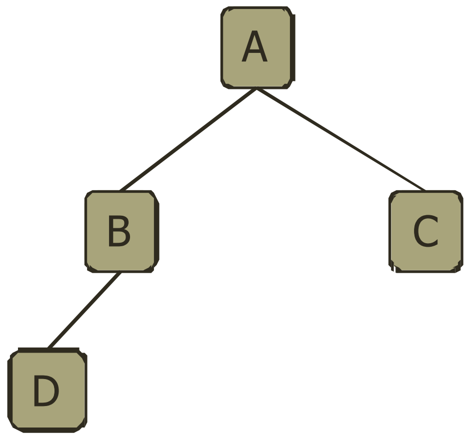


Binarytree [2]

- **Some typical types of Binary tree**
  - **Perfect Binary tree**: every level is completely filled.
  - **Full Binary tree**: each node is either a leaf or has exactly two children.
  - **Complete Binary tree**: a perfect binary tree except perhaps for the final level which is filled from left to right.

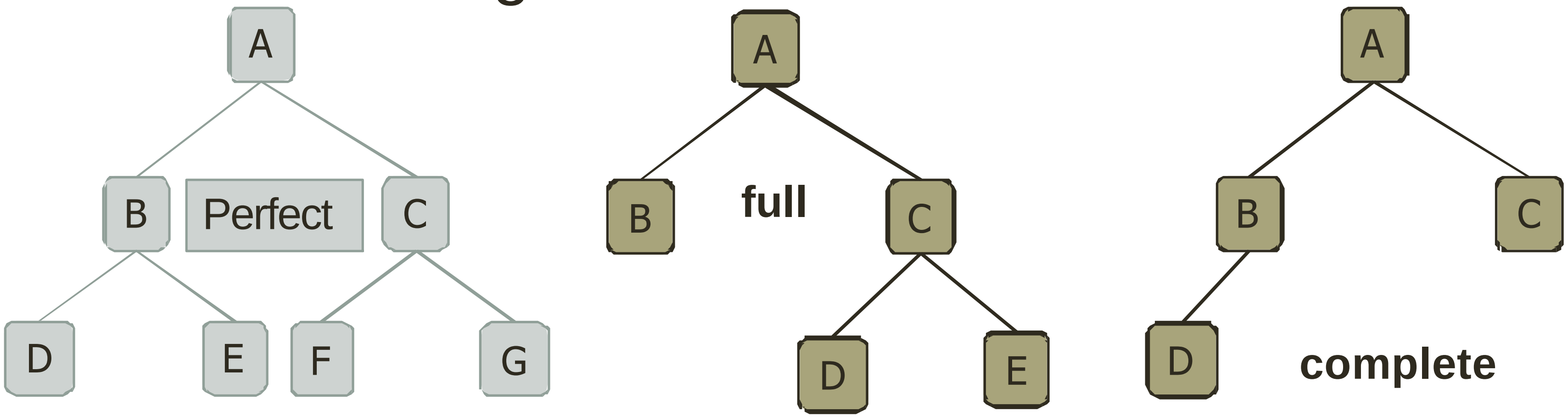






Binarytree [2]

- **Some typical types of Binary tree**
  - **Perfect Binary tree**: every level is completely filled.
  - **Full Binary tree**: each node is either a leaf or has exactly two children.
  - **Complete Binary tree**: a perfect binary tree except perhaps for the final level which is filled from left to right.



Binarytree [3]

- **Number of nodes in a Binary tree**
  - Consider a binary tree **T** with the depth **h**, **n** is the total nodes of T, **l** is the total leaves of T
  - If T is a perfect binary tree, then:

$$\boldsymbol{l} = \mathbf{2}^{\boldsymbol{h}} \; and \; \boldsymbol{n} = \mathbf{2}^{\boldsymbol{h}+\mathbf{1}} - \mathbf{1} = \mathbf{2}\boldsymbol{l} - \mathbf{1}$$

  - If T is a full binary tree, then:

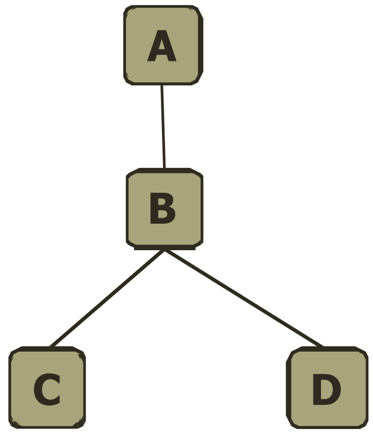
$$\mathbf{2}^{\boldsymbol{h}} \leq \boldsymbol{n} \leq \mathbf{2}^{\boldsymbol{h}+\mathbf{1}} - \mathbf{1}$$

Tree traversal [1]

- **Definition:** A **traversal** of a tree:
  - **Start from the root of the tree**
  - **Visit every node** of the tree.
  - Each node is **visited once**.
- **Traversal order**
  - Order of visiting
- **Visit**:
  - Do something with the node
    - Print node's label
    - Perform an operation on node's data
  - **Visit a node <> go to a node**

Tree traversal [2]

- **Example**
  - Tree traversal to print all node's label



- **Result**
  - **A, B, C, D**
    - Start from A, visit A → go to B, visit B → go to C, visit C → go to B → go to D, visit D.
  - **C, D, B, A**
    - Start from A → go to B → go to C, visit C → go to B → go to D, visit D → go to B, visit B → go to A, visit A.
  - **C, B, D, A**
    - Start from A → go to B → go to C, visit C → go to B, visit B → go to D, visit D → go to B → go to A, visit A.

Tree traversal [3]

- **Level-order**

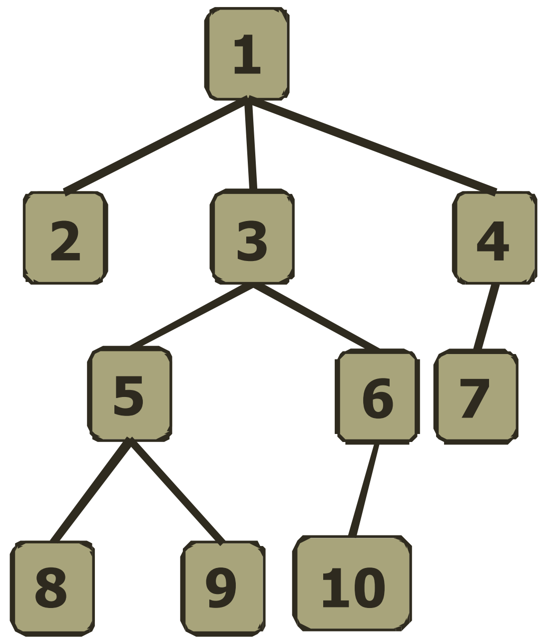
  Start at root, visit root.

  Visit the nodes at each level, from left to right.

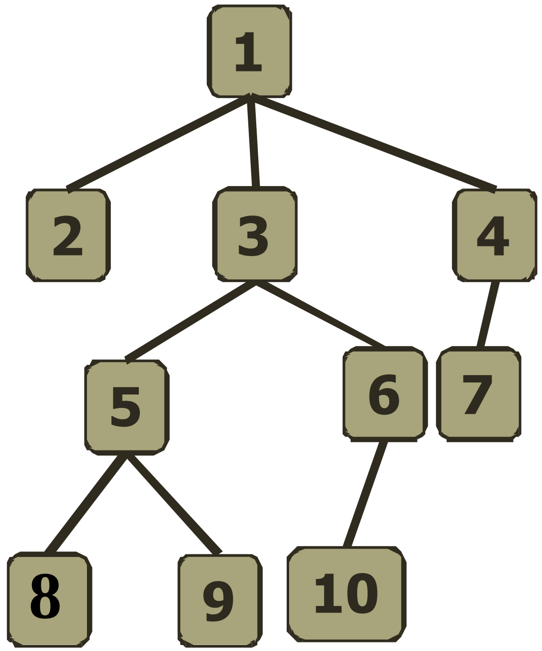
  1→2 →3 →4 →5 →6 →7 →8 →9 →10

- **Pre-order**

  Visit each node, followed by its children (in pre-order) from left to right.
  1 →2 →3 →5 →8 →9 →6 →10 →4 →7



Tree traversal [4]



- **In-order**

  Visit the left-most child, followed by the root, followed by the remaining children from left to right.

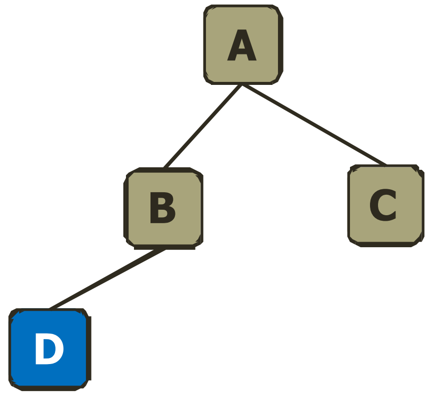
  2 →1 →8 →5 →9 →3 →10 →6 →7 → 4

- **Post-order**

  Visit the left-most child, followed by the remaining children from left to right, followed by the root.

  2 →8 →9 →5 →10 →6 →3 →7 →4 →1

Tree operations [1]



- **Create an empty tree**
- **Add a new node to the tree**
  - A **new node** is added to the tree **as a child of a parent node.**
  - If the parent is null, new node is the root of the tree.
  - Node is added from **left to right.**
  - Example: **addNode(D,B)**

Tree operations [2]

- **Get root node of the tree**
  - Return the root node of the tree.
  - Return null if the tree is empty.
- **Get the parent node of a node in the tree**
  - Return the parent node of a node n in the tree.
  - Return null if the tree is empty or n is the root
- **Get the left most child of a node in the tree**
  - Return the left most child of a node n in the tree
  - Return null if the tree is empty or n has no child

Tree operations [3]

- **Get the nearest right sibling of a node in the tree**
  - Return the nearest right sibling of a node n in the tree.
  - Return null if the tree is empty or n has no right sibling.
- **Get/Set node's label**
  - Get/set the label of a node in the tree.
- **Check if the tree is empty**
  - Return true if the number of nodes =0, otherwise return false

Tree ADT

Tree

+addNode(NodeType newNode, NodeTypeparent): void
+getRoot(): NodeType
+getParent(NodeType n): NodeType
+leftMostChild(NodeType n): NodeType
+rightSibling(NodeType n): NodeType
+getNodeLabel(NodeType n): LabelType
+setNodeLabel(NodeType n, LabelType label): void
+isEmpty(): boolean
…

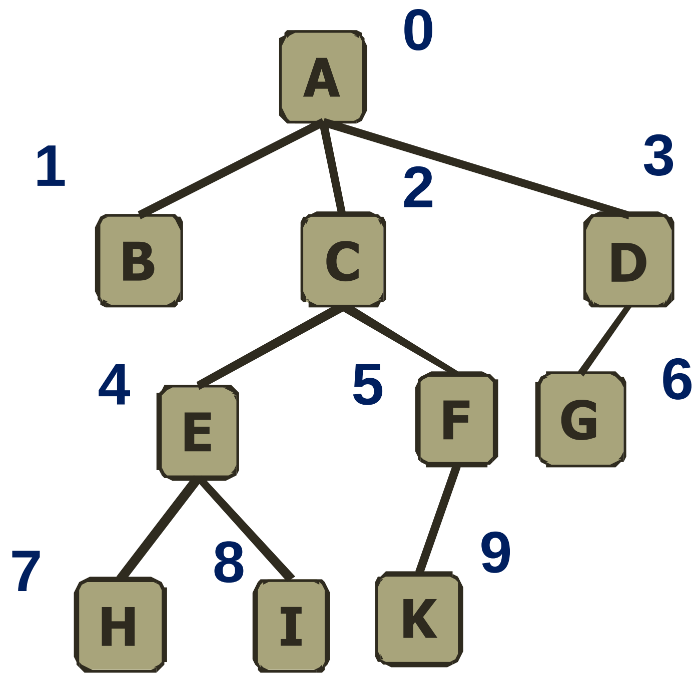
Tree Implementation

Array-based tree[1]

- **Array is used to represent a tree**
  - A tree T has N nodes, **each node is indexed by a number from 0 to N-1**.
  - With **node i** in the tree
    - **L[i]** is the label of node i
    - **P[i]** is the parent of node I
- **Indexing scheme:**
  - **Incremental indexing** started from root
  - **Root is 0**, parent is **smaller** than children
  - Children of the same parent are indexed from **left to right**

Array-based tree[2]

- **Example**



	0	1	2	3	4	5	6	7	8	9
L:	A	B	C	D	E	F	G	H	I	K
P:	-1	0	0	0	2	2	3	4	4	5

Array-based tree[3]

Tree

-maxSize: int //maximum possible number of nodes
-n: int //current number of nodes in the tree
-l: String[] //Label array
-p: int[] //Parent array

+addNode(String label, int parent): void
+getParent(int node): int
+leftMostChild(int node): int
+rightSibling(int node): int
+getNodeLabel(int node): String
+setNodeLabel(int node, String label): void
…

Array-based tree[4]

- addNode(String **label**, int **parent**)
  - If parent is -1, then we will add a root node

    ```
    l[0]=label;  p[0]=-1;
    ```

  - If parent is different from -1
    - Must **find a correct position** for the new node
    - **Shifting the array to the right** if it is necessary

```
addNode("G",3);
```

0	1	2	3
A	B	C	D
-1	0	0	0

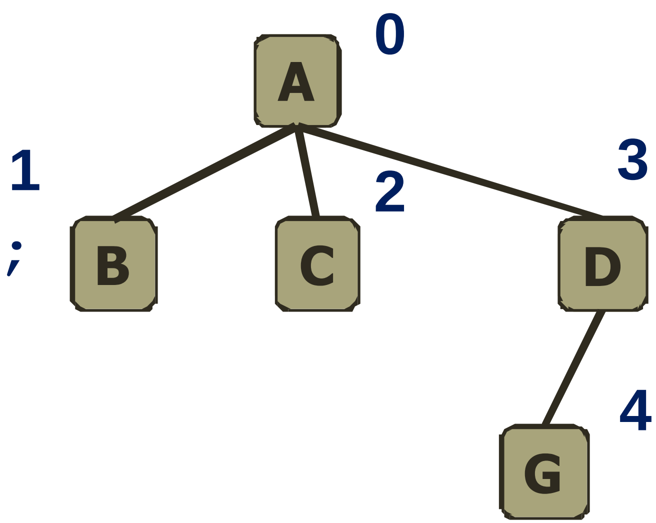
Array-based tree[4]

- addNode(String **label**, int **parent**)
  - If parent is -1, then we will add a root node

    ```
    l[0]=label;  p[0]=-1;
    ```

  - If parent is different from -1
    - Must **find a correct position** for the new node
    - **Shifting the array to the right** if it is necessary

```
addNode("G",3);
```



0	1	2	3
A	B	C	D
-1	0	0	0

0	1	2	3	4
A	B	C	D	G
-1	0	0	0	3

Array-based tree[5]

- addNode(String **label**, int **parent**)
  - If parent is different from -1
    - Must **find a correct position** for the new node
    - **Shifting the array to the right** if it is necessary

0	1	2	3	4
A	B	C	D	G
-1	0	0	0	3

```
addNode("E",2);
```

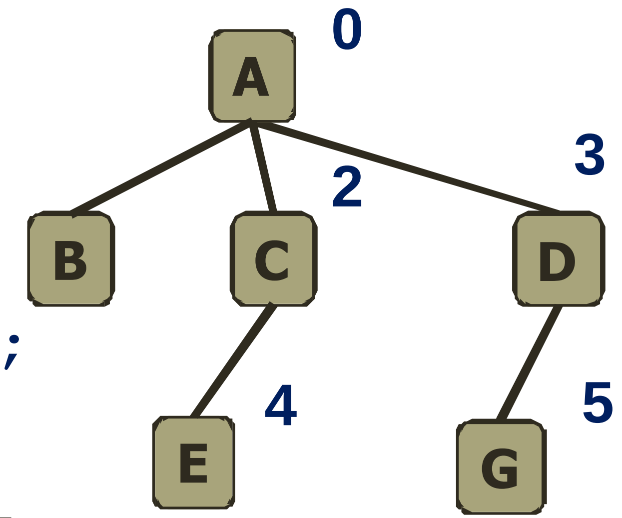
Array-based tree[5]

- addNode(String **label**, int **parent**)
  - If parent is different from -1
    - Must **find a correct position** for the new node
    - **Shifting the array to the right** if it is necessary

0	1	2	3	4
A	B	C	D	G
-1	0	0	0	3

```
addNode("E",2);
```

0	1	2	3	4	5
A	B	C	D	E	G
-1	0	0	0	2	3



Array-based tree[6]

- leftMostChild(int **node**)
  - Example:

```
leftMostChild(0)  is 1
leftMostChild(3) is 4
leftMostChild(2) is -1
```

- Start from **node+1**

  - Find the first node i that:

  p[i]==node

  - Return -1 if could not find i

0	1	2	3	4
A	B	C	D	G
-1	0	0	0	3

Array-based tree[6]

- leftMostChild(int **node**)
  - Example:

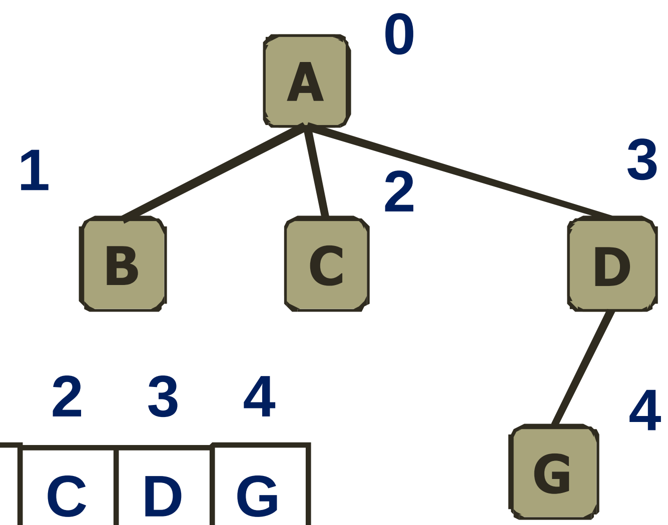
```
leftMostChild(0)  is 1
leftMostChild(3) is 4
leftMostChild(2) is -1
```

- Start from **node+1**

- Find the first node i that:

p[i]==node

- Return -1 if could not find i



0	1	2	3	4
A	B	C	D	G
-1	0	0	0	3

Array-based tree[7]

- nearestRightSibling(int **node**)
  - Example:

```
nearestRightSibling(1) is   2
nearestRightSibling(2) is   3
nearestRightSibling(3) is   -1
```

  - Check **node+1**

    - Return **node+1** if:

    p[node+1]==p[node]

    - Return -1 otherwise

0	1	2	3	4
A	B	C	D	G
-1	0	0	0	3

Array-based tree[7]

- nearestRightSibling(int **node**)
  - Example:

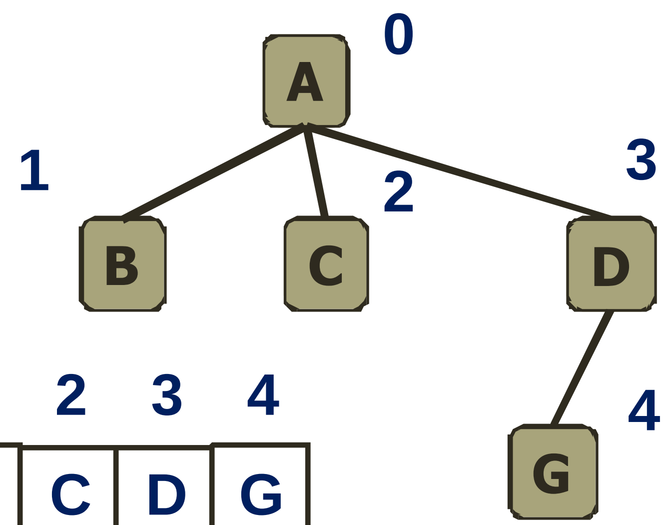
```
nearestRightSibling(1)  is   2
nearestRightSibling(2) is   3
nearestRightSibling(3) is   -1
```

  - Check **node+1**

    - Return **node+1** if:

    p[node+1]==p[node]

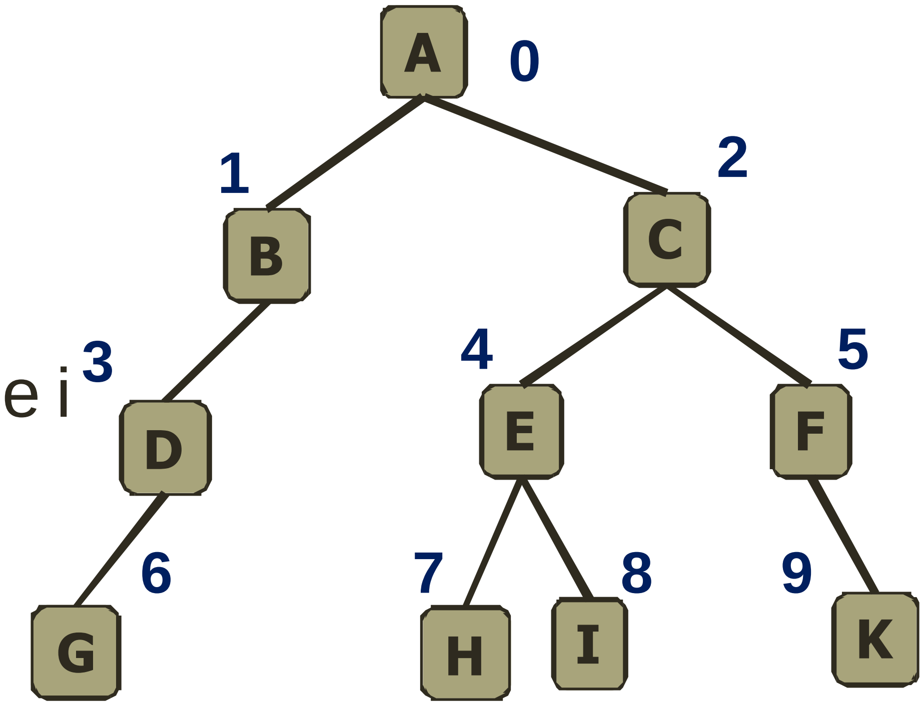
    - Return -1 otherwise



0	1	2	3	4
A	B	C	D	G
-1	0	0	0	3

Array-based binarytree [1]

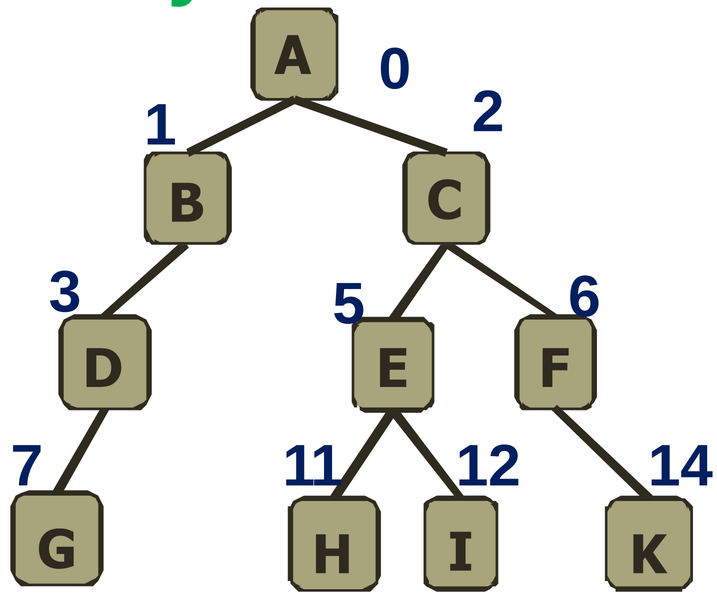
- **Array is used to represent a binary tree**
  - Using the above same indexing scheme
  - With **node i** in the tree
    - **L[i]** is the label of node i
    - **P[i]** is the parent of node i



	0	1	2	3	4	5	6	7	8	9
L:	A	B	C	D	E	F	G	H	I	K
P:	-1	0	0	1	2	2	3	4	4	5

Array-based binarytree [2]

- **Array is used to represent a binary tree**
  - Using **perfect binary tree indexing scheme**
  - With **node i** in the tree
    - **L[i]** is the label of node i
    - Left child: **2i+1**
    - Right child: **2i+2**
    - Parent: $\lfloor (i-1)/2 \rfloor$ $\lfloor x \rfloor$: "Floor" ⓪ The greatest integer less than x



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
L:	A	B	C	D	-	E	F	G	-	-	-	H	I	-	K

Array-based binarytree [3]

BinaryTree

-maxSize: int //maximum possible number of nodes

-l: String[]   //Label array

-n: int   / / Current number of nodes

___

+addRoot(String label) : void

+getLeftChild(int node) : int

+getRightChild(int node): int

+getParent(int node): int

+addLeftChild(int node): int

+addRightChild(int node): int

+getNodeLabel(int node): String

+setNodeLabel(int node, String label): void

+preOrderTravel(int node) : void

...

Tutorial & next topic

- **Preparing for the tutorial:**
  - Practice with examples and exercises in Tutorial 8 Example Code
- **Preparing for next topic:**
  - Read textbook chapter 4 (4.1 – 4.3): Tree data structure.
  - Read supplementary book chapter 10 (10.4) and chapter 12