

Lecture 7: **Tree ADT**

01204212 Abstract Data Types and Problem Solving

Department of Computer Engineering
Faculty of Engineering, Kasetsart University
Bangkok, Thailand.



Department of
Computer Engineering
Kasetsart University



Outline

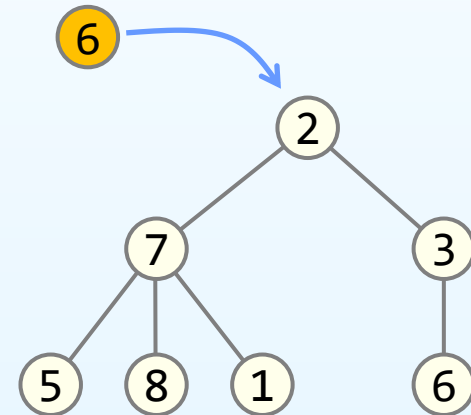
- Terminologies
- Implementation
- Tree Traversals
- Forests

What is a Tree ADT?



- Data:
 - A set of linked nodes (elements) that form a **hierarchical tree structure** with a **root** and its **subtrees**
 - This is a **recursive definition**
- Defined operations:
 - `attach(tree, parent, child)`
 - `detach(tree, node)`
 - `search(tree, node)`
 - `degree(tree, node)`
 - `is_root(tree, node)`
 - ...

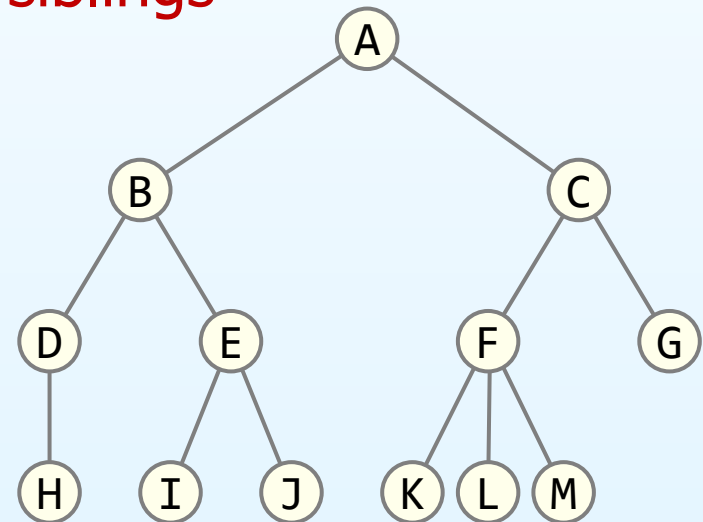
`attach(t, 3, 6)`



`detach(t, 6)`

Terminology: Parent, Child, and Sibling

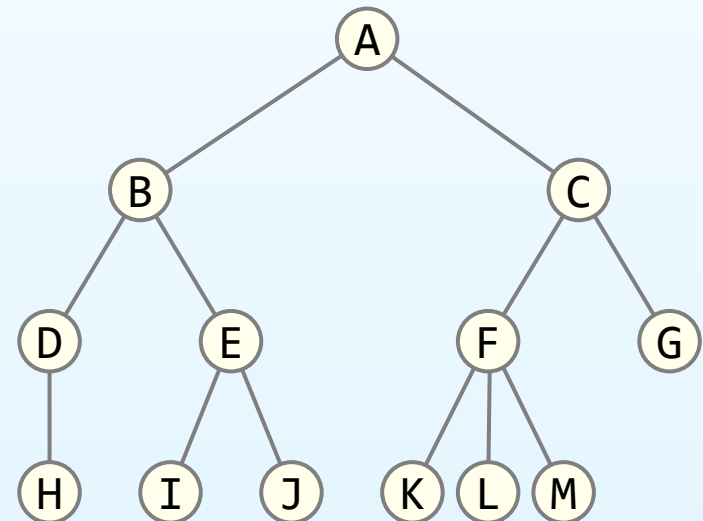
- A tree consists of nodes with a parent-child relation
- All nodes will have zero or more **child(ren)**
e.g., B has two children: D and E
- All nodes, except the first node, have only one **parent**
e.g., A is the parent of B
- Nodes with the same parent are **siblings**
e.g., B and C are siblings



Terminology: Root, Internal, and Leaf

- **Root** – node without parent
e.g., A is the root of the tree
- **Internal node** – node with at least one child
e.g., A, B, C, D, E, and F are internal nodes
- **External node** (a.k.a. **leaf**) – node without children
e.g., G, H, I, J, K, L, and M are leaves

rooted tree

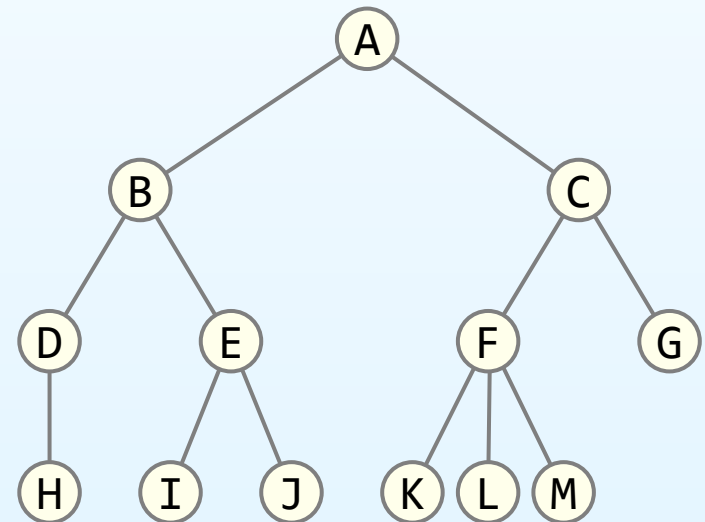


Terminology: Degree

- The **degree** of a node is defined as the number of its children

e.g., A has degree 2

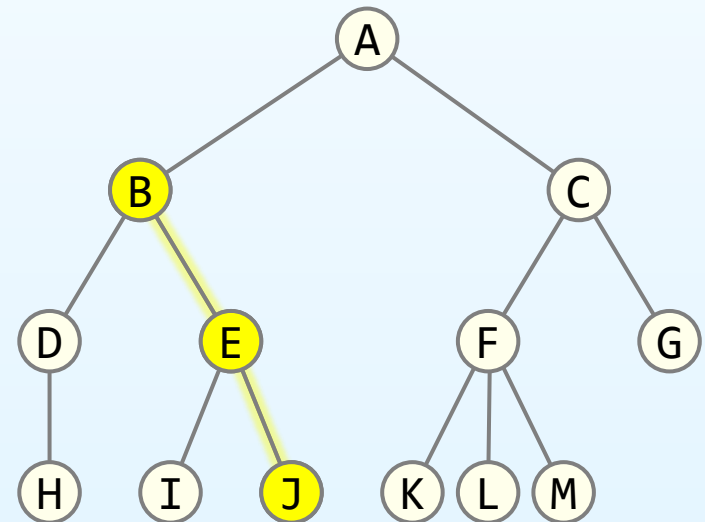
All leaf nodes have zero-degree



Terminology: Path and Path Length

- A **path** is a sequence of nodes $\langle a_i, a_{i+1}, a_{i+2}, \dots, a_j \rangle$ where a_{i+1} is a child of a_i
- The **length** of a path is defined as the number of links along that path

e.g., the path $\langle B, E, J \rangle$ has length 2

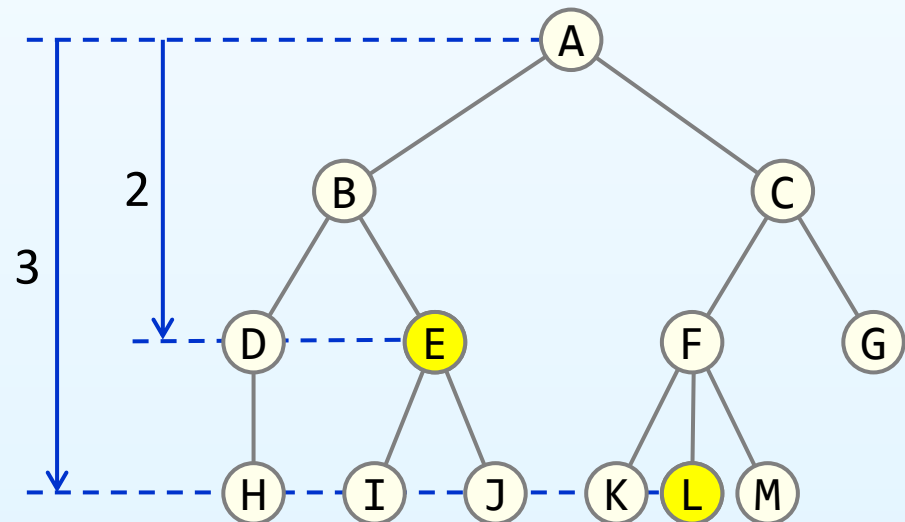


Terminology: Depth

- For each node in a tree, there exists a unique path from the root node to that node
- The length of this path is the depth of the node

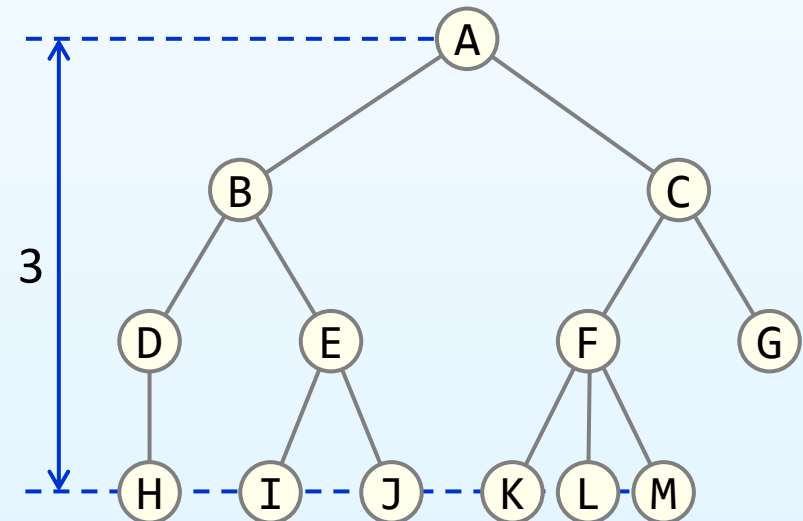
e.g., E has depth 2

L has depth 3



Terminology: Height

- The **height** of a tree is defined as the maximum depth of any node within the tree
e.g., the height of the example tree is 3
- The height of a tree with **one node** is 0
- For convenience, we define the height of the **empty tree** to be -1

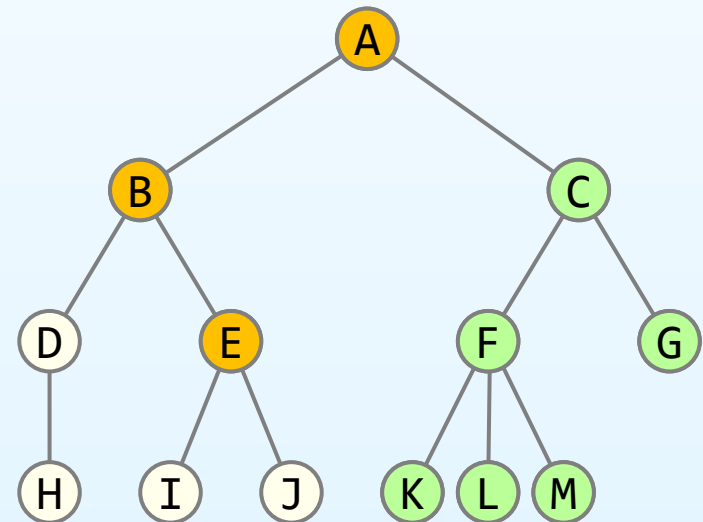


Terminology: Ancestor and Descendant

- **Ancestor** – the connected **higher-level** nodes
- **Descendant** – the connected **lower-level** nodes
- However, a node is **both** an ancestor and a descendant of **itself**

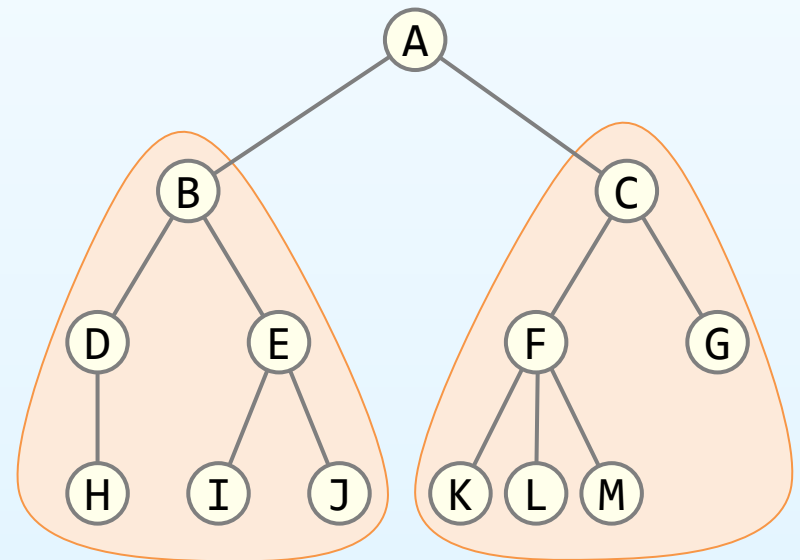
e.g., the ancestors of node E are A, B, and E

the descendants of node C are C, F, G, K, L and M



Terminology: Subtree

- Another approach is to define a tree **recursively**:
 - (Base case) A zero-degree node is a tree
 - (Recursion) A node with degree n is a tree if it has n children and all of its children are disjoint trees—**subtrees** with no intersecting nodes
- **Subtree** is a part of tree consisting of a node and its descendants



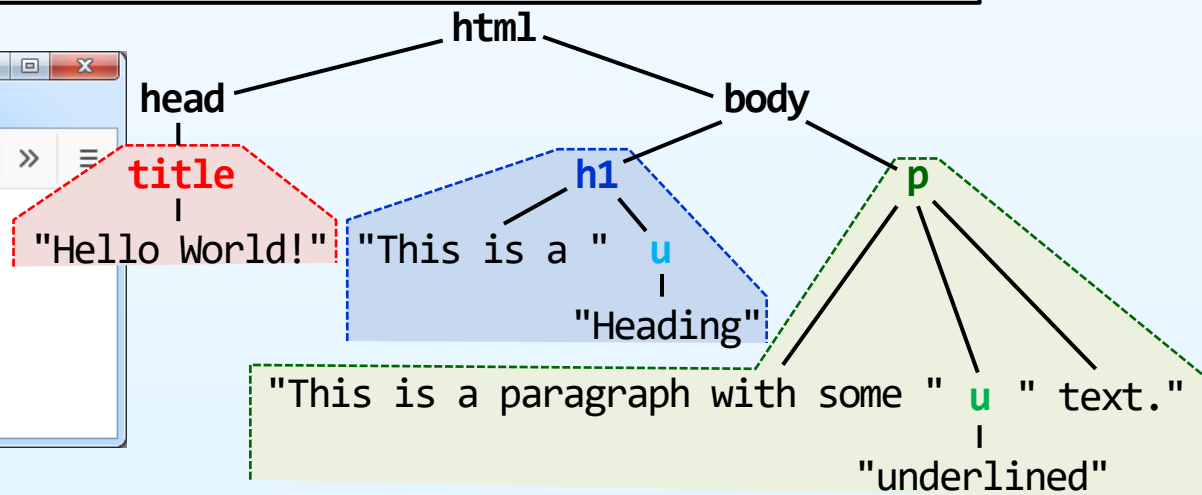
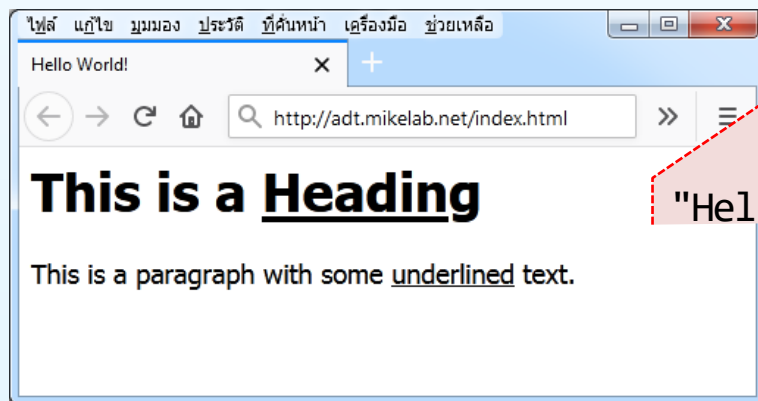
Important Properties

- A tree with n nodes always has $n - 1$ edges
- Any two nodes in a tree have at most one path between them

Application: HTML Structure

- The nested tags of HTML can be defined as a tree:

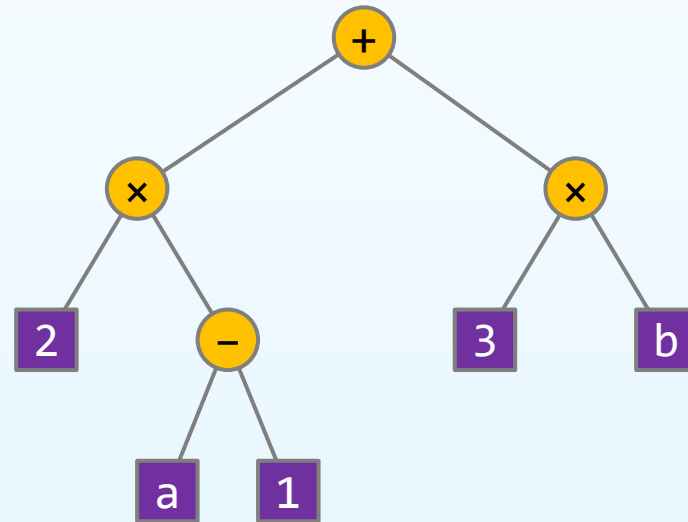
```
<html>
  <head>
    <title>Hello World!</title>
  </head>
  <body>
    <h1>This is a <u>Heading</u></h1>
    <p>This is a paragraph with some <u>underlined</u> text.</p>
  </body>
</html>
```



Application: Arithmetic Expression Tree

- A tree associated with an arithmetic expression
 - Internal nodes: operators
 - External nodes: operands

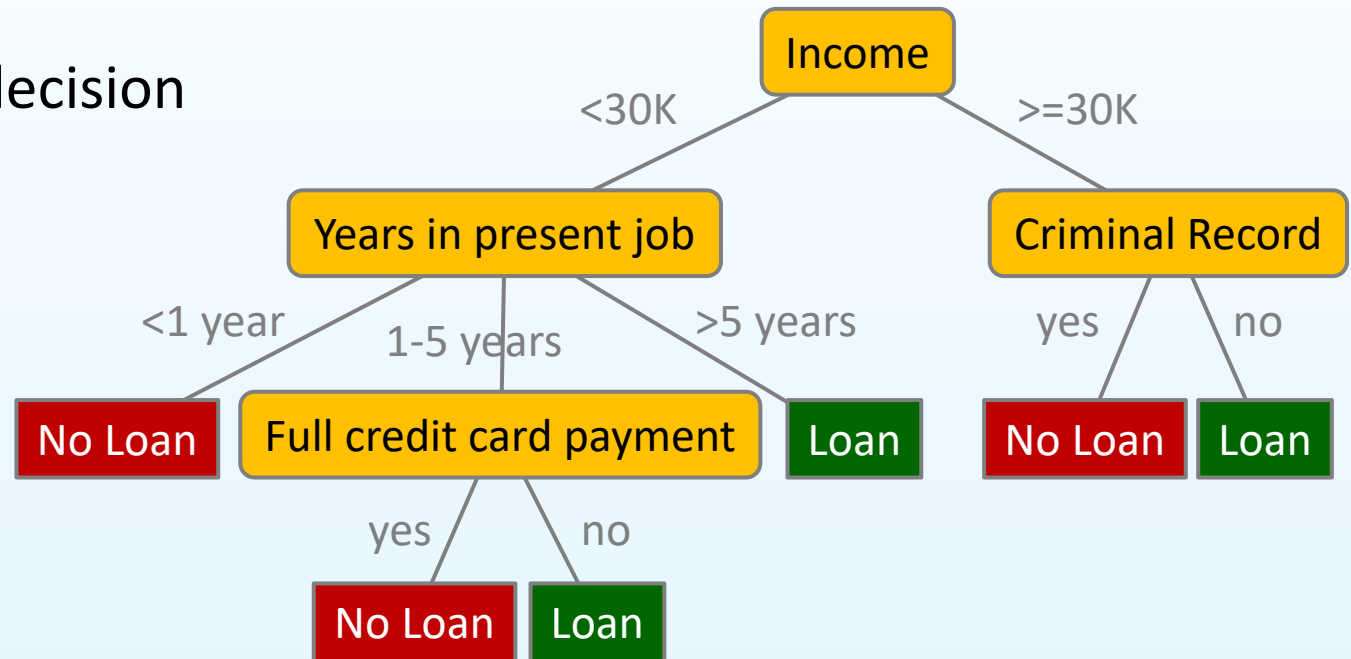
$(2 \times (a - 1) + (3 \times b))$



Application: Decision Tree

- A tree associated with a decision process
 - **Internal nodes:** questions (with yes/no answer)
 - **External nodes:** decisions

Loan approval decision



Outline

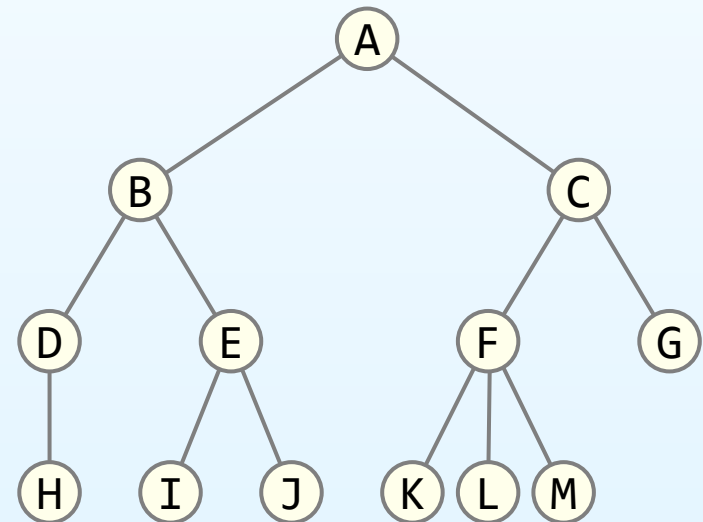
- Terminologies
- Implementation
- Tree Traversals
- Forests

Abstract Trees

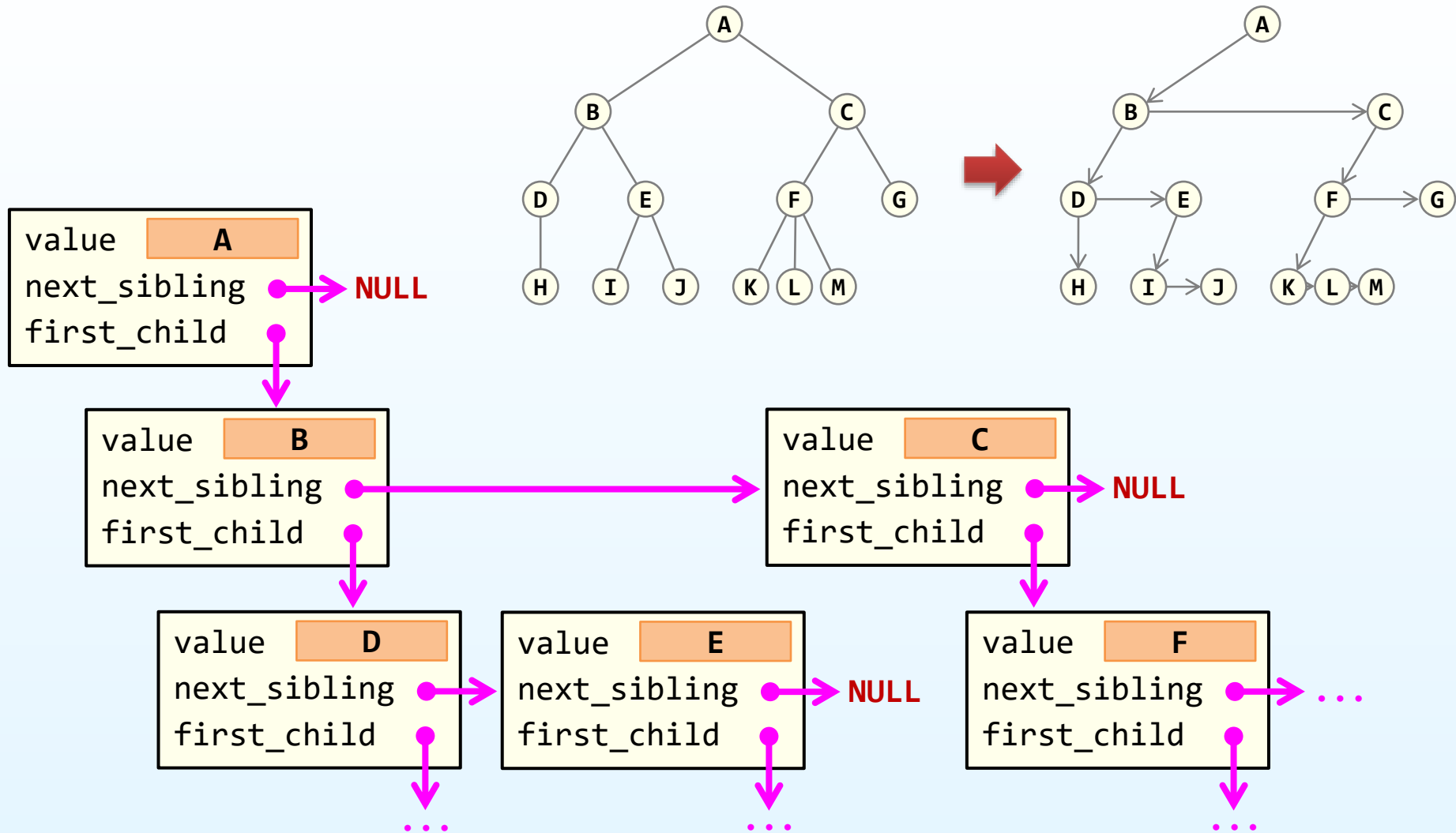
- A hierarchical ordering of a finite number of objects may be stored in a tree data structure
- Operations on a hierarchically stored container include:
 - Accessing the root
 - Given a node
 - Access its parent
 - Find the degree
 - Get a reference to a child
 - Attach a new subtree
 - Detach this subtree from its parent

Abstract Trees: Design

- A general tree does not strict the number of children
 - One possible **pointer-based implementation**
- What will be stored in an individual node?
 - A value
 - A pointer to next sibling
 - A pointer to the 1st child



Tree Implementation

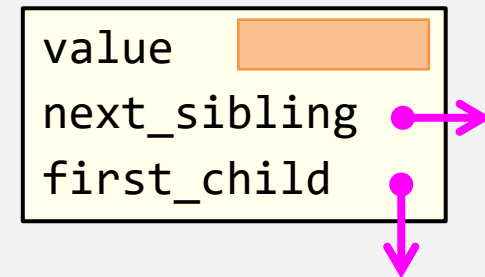


Tree Implementation

Assume that all data are distinct positive integers

```
1: #include <stdio.h>
2: #include <stdlib.h>
3:
4: typedef struct node {
5:     int value;
6:     struct node *next_sibling;
7:     struct node *first_child;
8: } node_t;
9:
10: typedef node_t tree_t;
11:
12: int main(void) {
13:     tree_t *t = NULL;
14:     return 0;
15: }
```

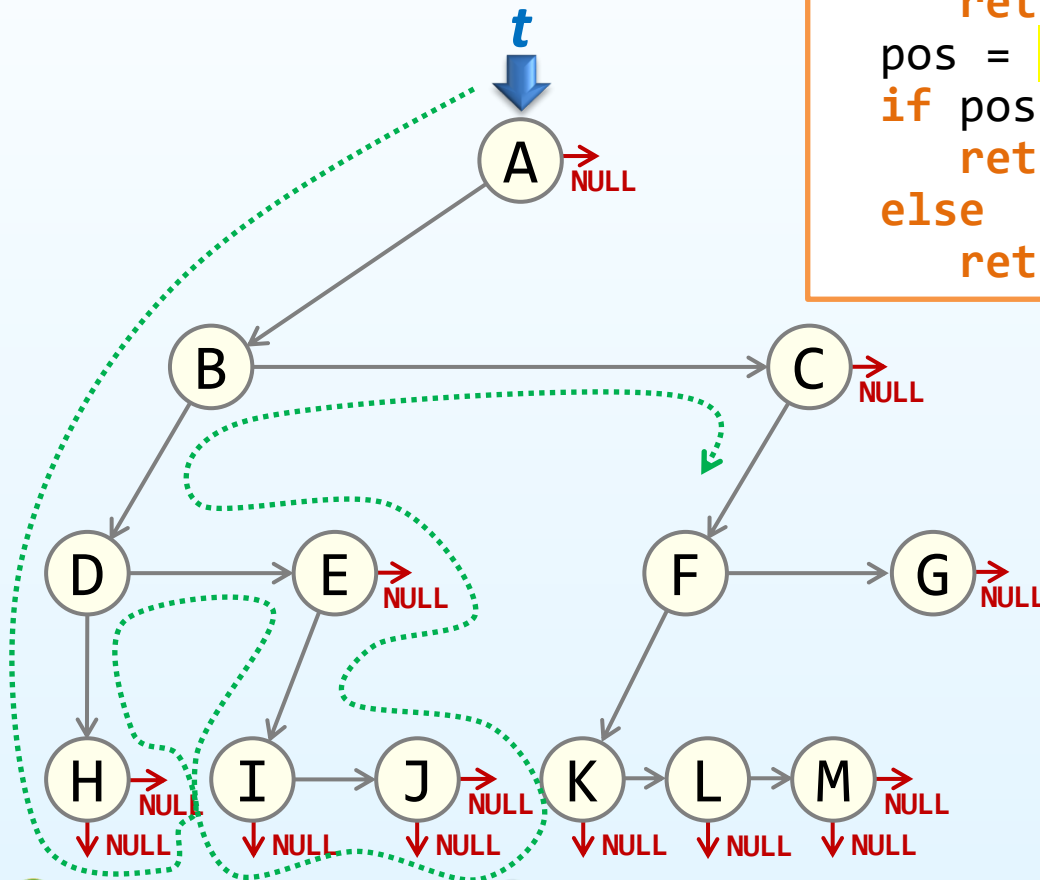
node



The search() Operation

Return position of node v in tree t if found, otherwise NULL

search(t , 'F')



Algorithm: search(t , v)

if $t \rightarrow \text{value} == v$ || $t == \text{NULL}$

return t

pos = search($t \rightarrow \text{first_child}$, v)

if pos != NULL

return pos

else

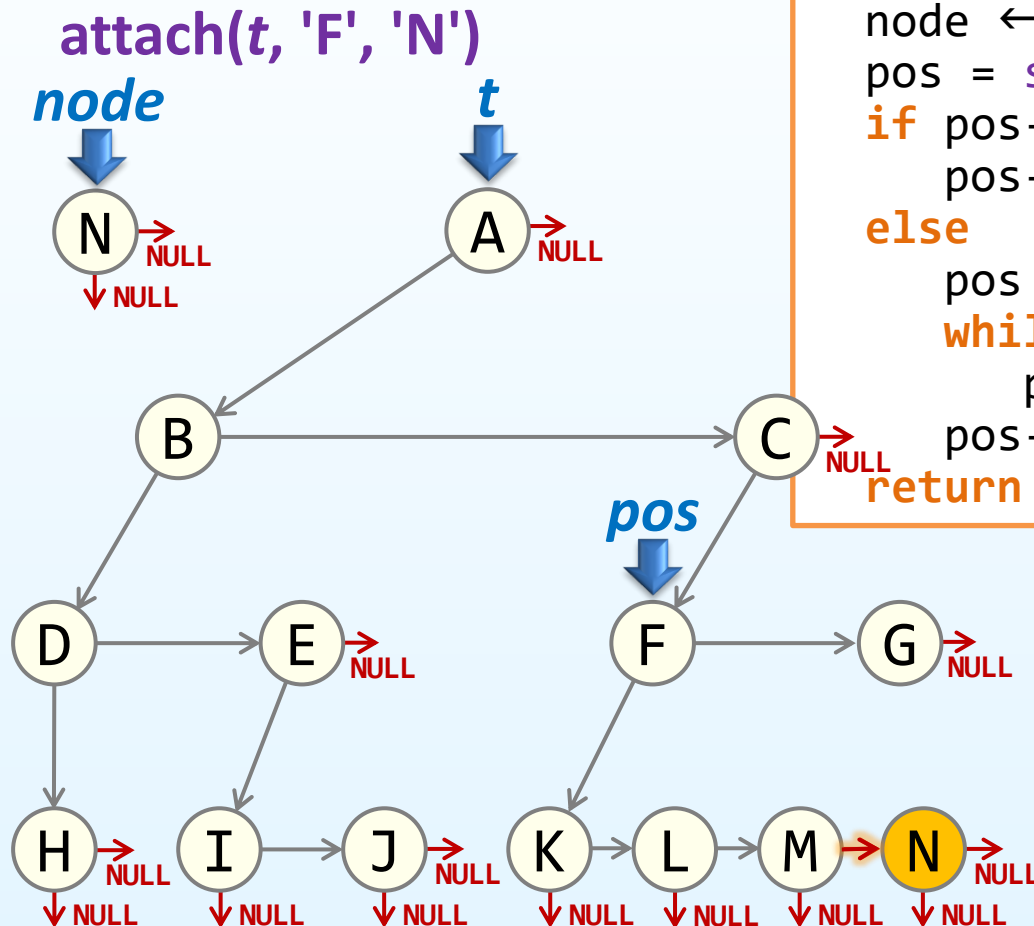
return search($t \rightarrow \text{next_sibling}$, v)

What is the running time?

The attach() Operation

Insert a node/subtree c as a child of p in tree t

```
Algorithm: attach( $t$ ,  $p$ ,  $c$ )  
  node  $\leftarrow$  allocate a new node  $c$   
  pos = search( $t$ ,  $p$ )  
  if pos->first_child == NULL  
    pos->first_child = node  
  else  
    pos = pos->first_child  
    while pos->next_sibling != NULL  
      pos = pos->next_sibling  
    pos->next_sibling = node  
  return  $t$ 
```



What is the running time?

Exercise 1: Other Operations

Implement the following functions for a **rooted tree**

- $\text{detach}(t, n)$ – delete a node/subtree n from tree t
 - Return t after the deletion
- $\text{degree}(t, n)$ – find the number of children of a node n
 - Return the number of children
- $\text{is_root}(t, n)$ – check whether a node n is root
 - return 1 if that node is the root node, otherwise 0
- $\text{is_leaf}(t, n)$ – check whether a node n is leaf
 - return 1 if that node is a leaf node, otherwise 0

Outline

- Terminologies
- Implementation
- Tree Traversals
- Forests

What is a traversal?

Once the objects are stored in a data structure,
how do we access them all?

- For an array or linked list, those objects can be accessed sequentially
 - ➡ $\Theta(n)$
- For a stack or queue, we can run multiple `pop()` or `dequeue()` operations
 - ➡ $\Theta(n)$
- However, how can we iterate through all the objects in a tree in an efficient manner
 - ➡ Require $\Theta(n)$ in running time

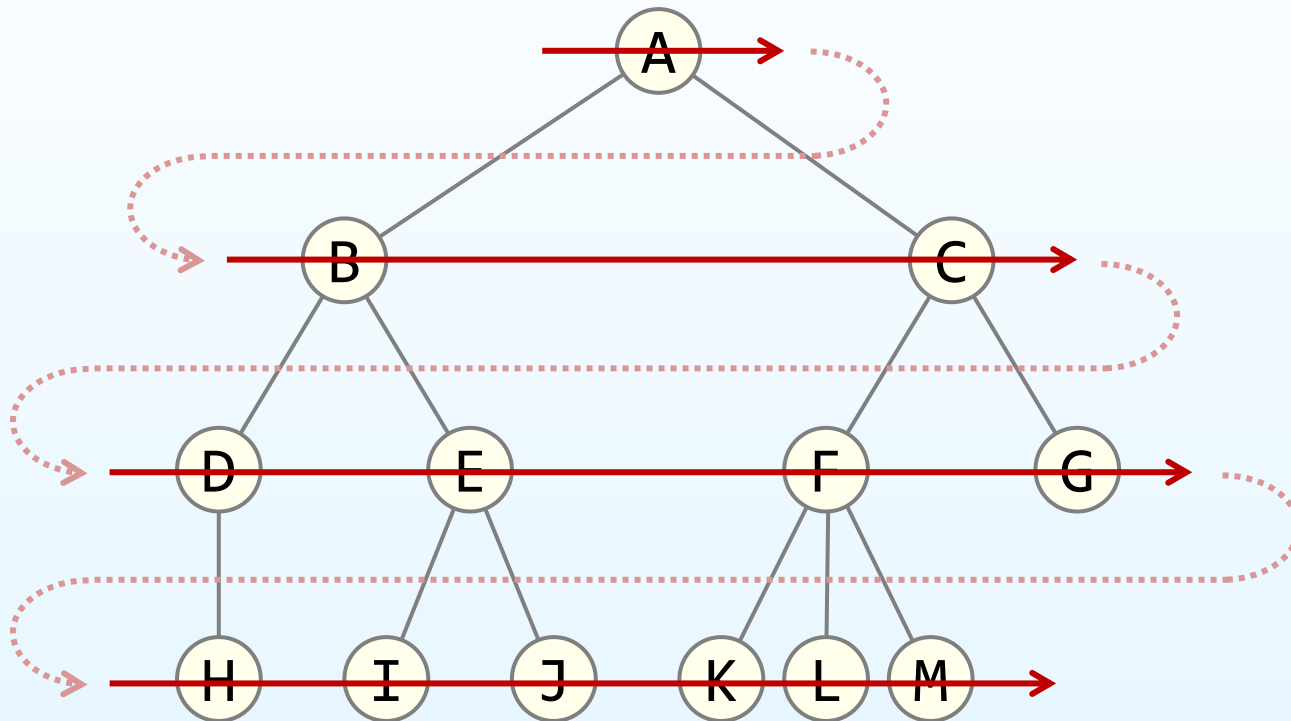
Types of Tree Traversal

Tree traversal algorithms can be classified broadly in **two categories** by the **order** in which the nodes are visited:

- Breadth-First Search (BFS)
- Depth-First Search (DFS)

Breadth-First Search (BFS)

“It starts from the root node and visits all nodes of **current depth** before moving to the next depth of tree.”

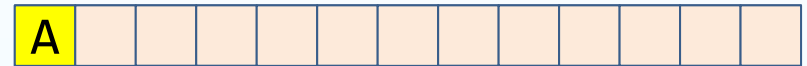


Breadth-First Search (BFS)

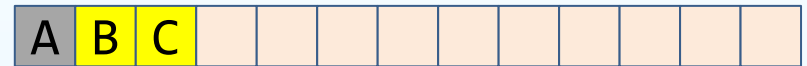
- BFS can be implemented using a **queue**

Algorithm for BFS

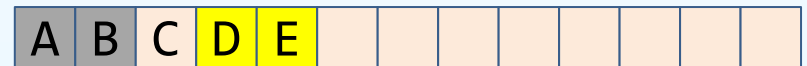
Step 1: enqueue({A})



Step 2: dequeue() and then enqueue({B,C})



Step 3: dequeue() and then enqueue({D,E})

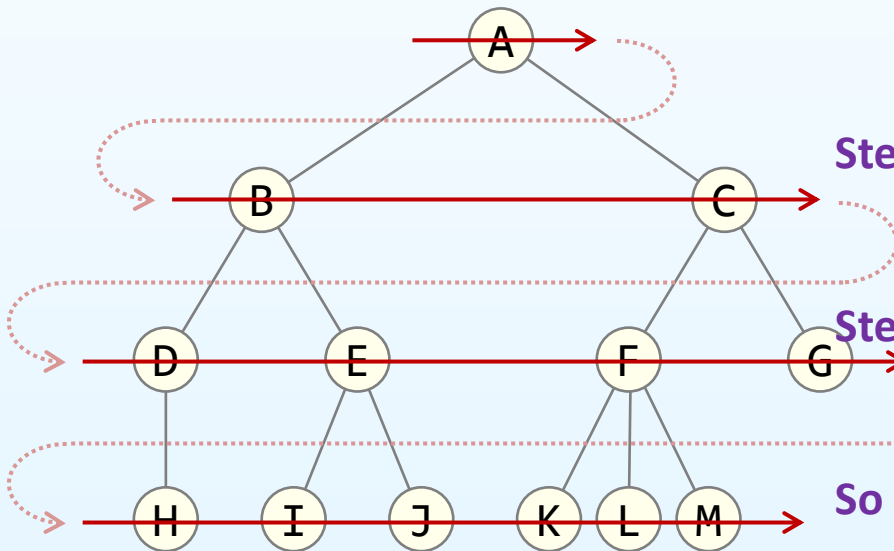


Step 4: dequeue() and then enqueue({F,G})



So on ...

Output: A B C ...



Breadth-First Search (BFS)

- BFS can be implemented using a **queue**

Algorithm: BFS(t)

$q \leftarrow$ allocate a queue

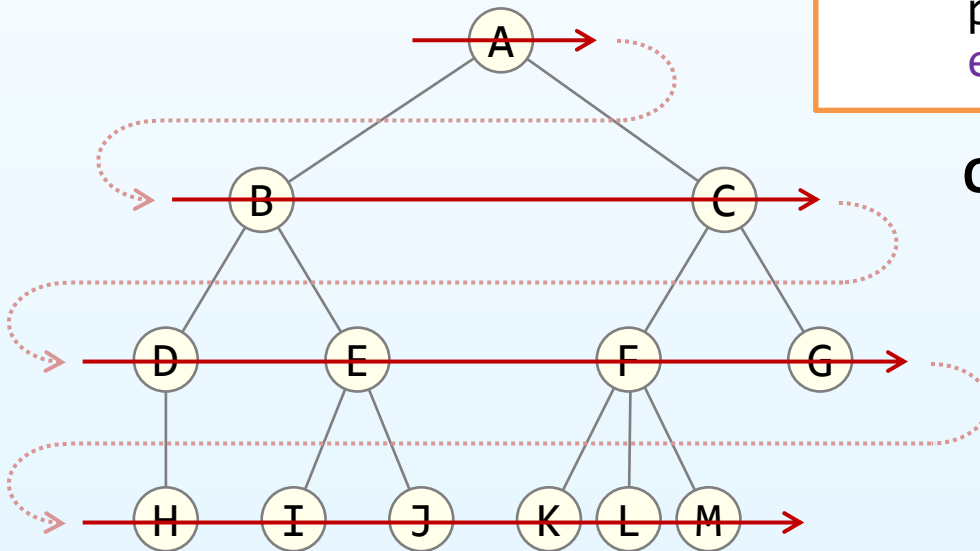
enqueue(q , root)

while q is not empty

node \leftarrow dequeue(q)

print(node)

enqueue(q , node's all children)



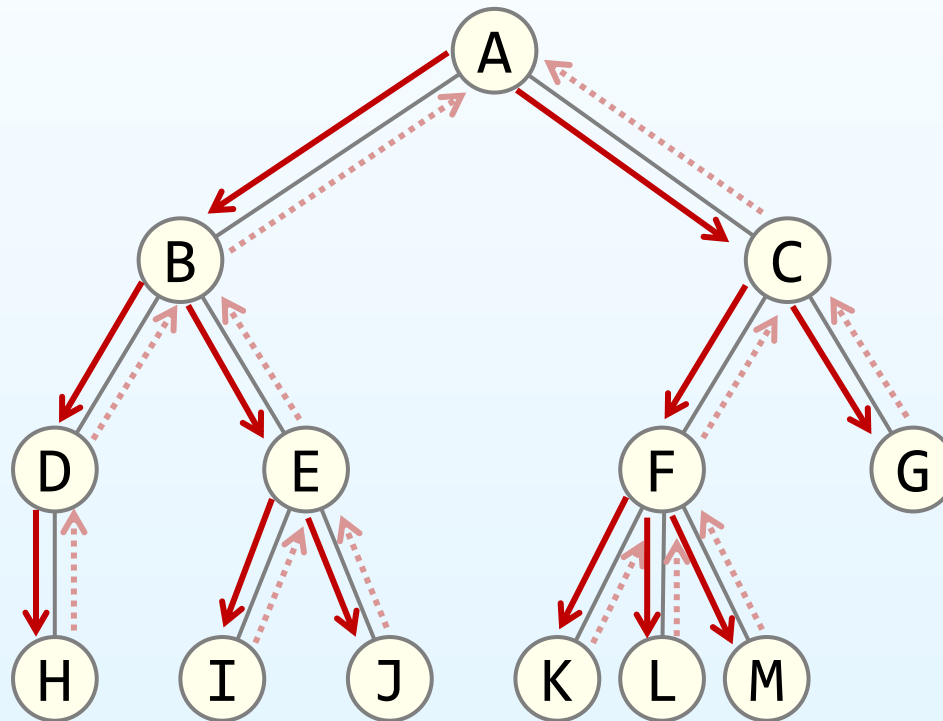
Output: A B C D E F G H I J K L M

What is the
running time?

Drawback: Memory is potentially expensive – maximum nodes at a given depth

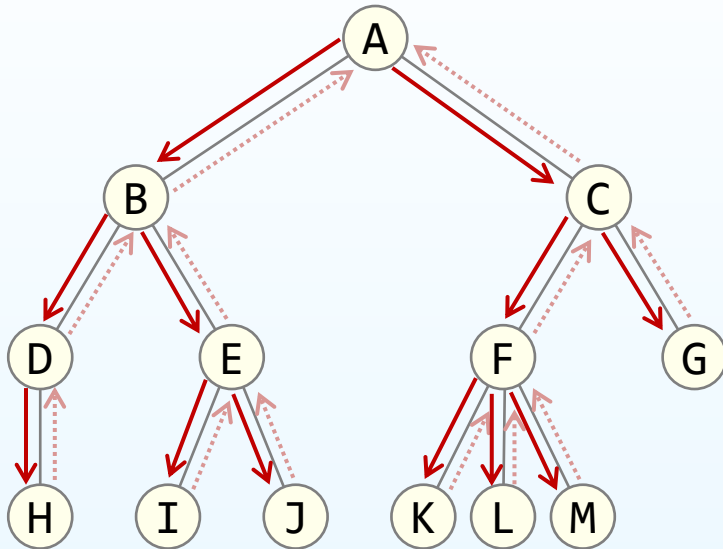
Depth-First Search (DFS)

“It starts with the root node and first visits all nodes of one branch **as deep as possible** before **backtracking**, it visits all other branches in a similar fashion.”



Depth-First Search (DFS)

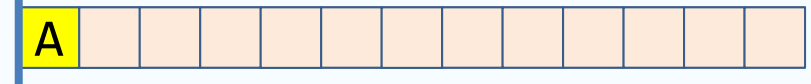
- DFS can be implemented using a **stack**



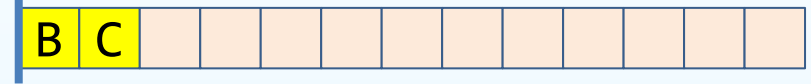
Output: A C G F M ...

Algorithm for DFS

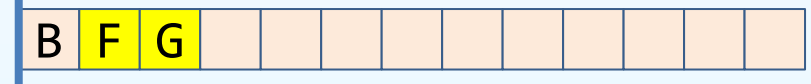
Step 1: `push({A})`



Step 2: `pop()` and then `push({B,C})`



Step 3: `pop()` and then `push({F,G})`



Step 4: `pop()` and then `push({})`

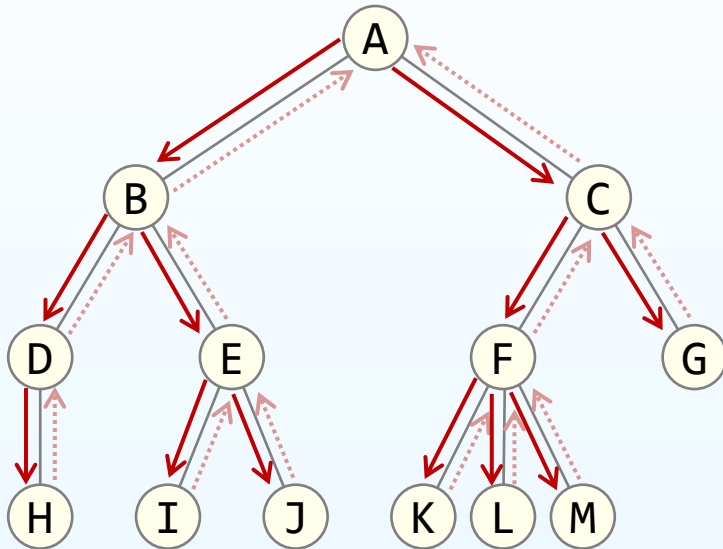


So on ...

The sequence is not correct as expected!
What should we do?

Depth-First Search (DFS)

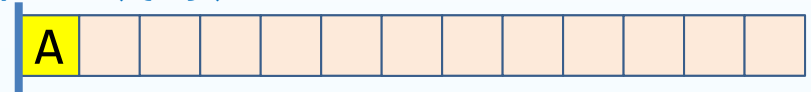
- DFS can be implemented using a **stack**



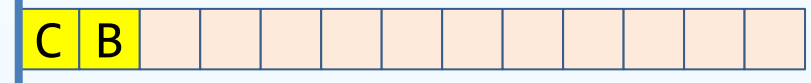
Output: A B D H E ...

Algorithm for DFS

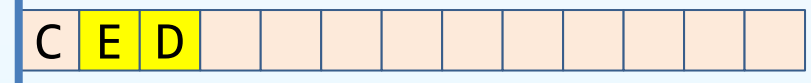
Step 1: **push**({A})



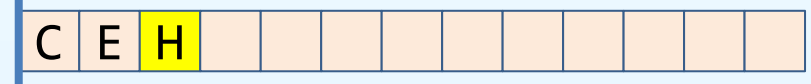
Step 2: **pop**() and then **push**({C,B})



Step 3: **pop**() and then **push**({E,D})



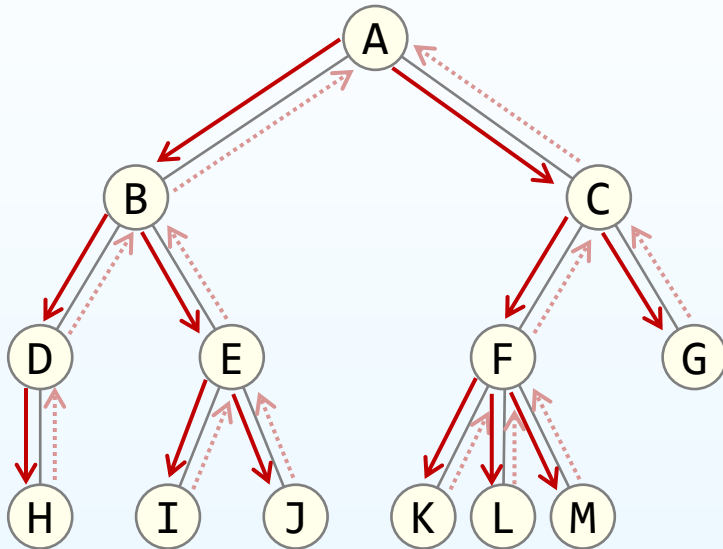
Step 4: **pop**() and then **push**({H})



So on ...

Depth-First Search (DFS)

- DFS can be implemented using a **stack**



Algorithm: DFS(t)

$s \leftarrow$ allocate a stack

push(s , root)

while s is not empty

node \leftarrow pop(s)

print(node)

push(s , node's all children in
reverse order)

Output: A B D H E I J C F K L M G

What is the running time?

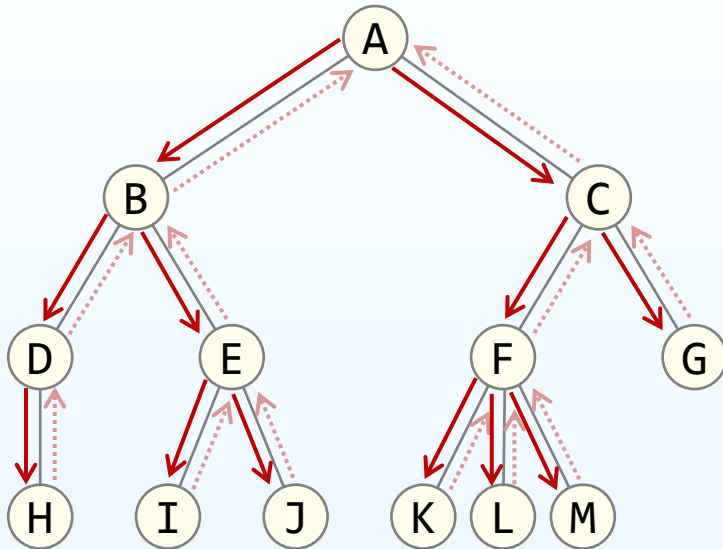
Note: The memory required is the height of the tree $\Theta(h)$



Depth-First Search (DFS)

- DFS can also be implemented using a **recursion**

We have already seen
this approach!!!



Algorithm: DFS(t)

node \leftarrow root of t

print(node)

DFS(node's first child)

DFS(node's sibling)

Output: A B D H E I J C F K L M G

Applications of DFS

- Finding the height of a tree
- Printing a hierarchical structure
- Calculating the total value of subtrees

Finding the Height

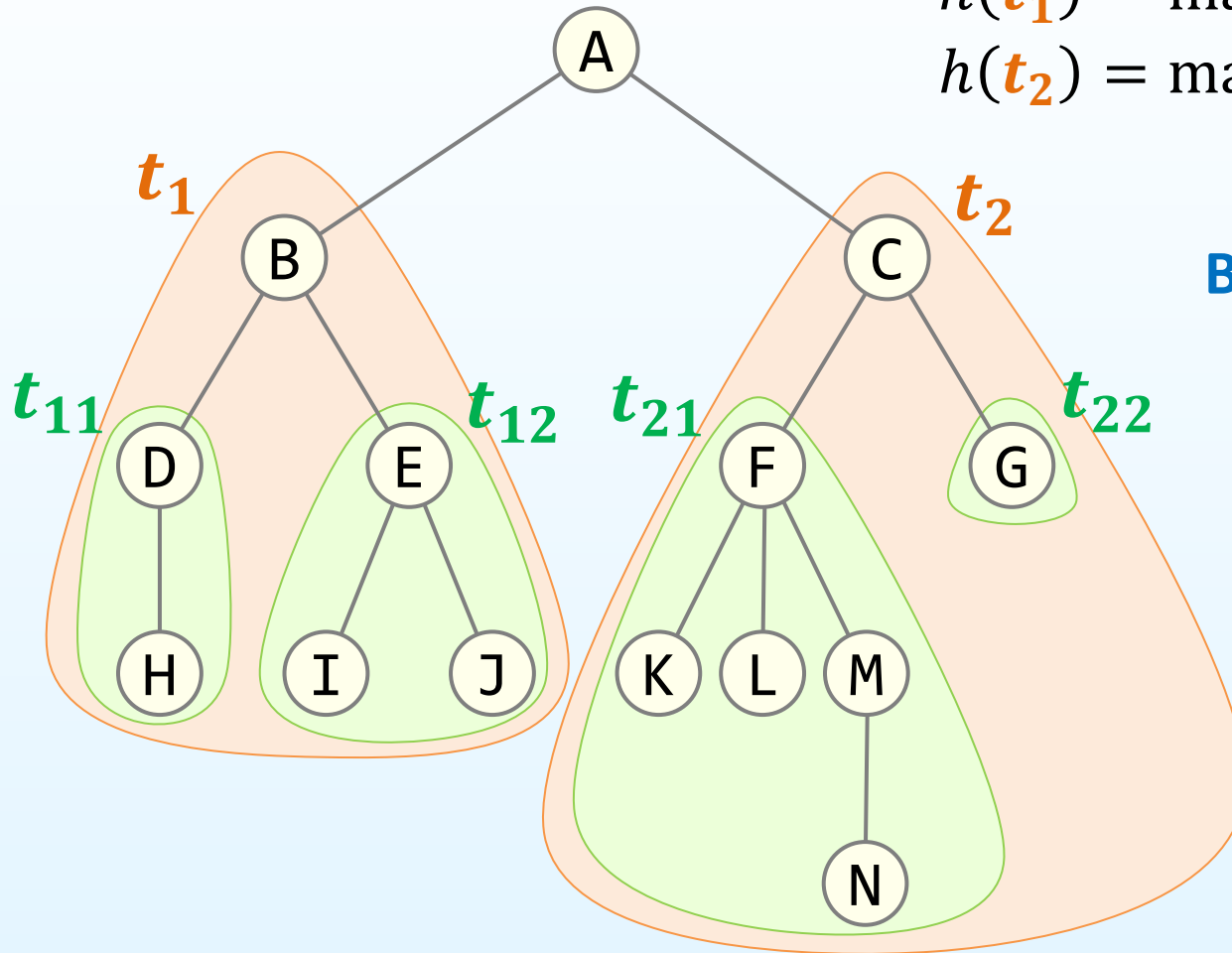
$$h(t) = \max(h(t_1), h(t_2)) + 1$$

$$h(t_1) = \max(h(t_{11}), h(t_{12})) + 1$$

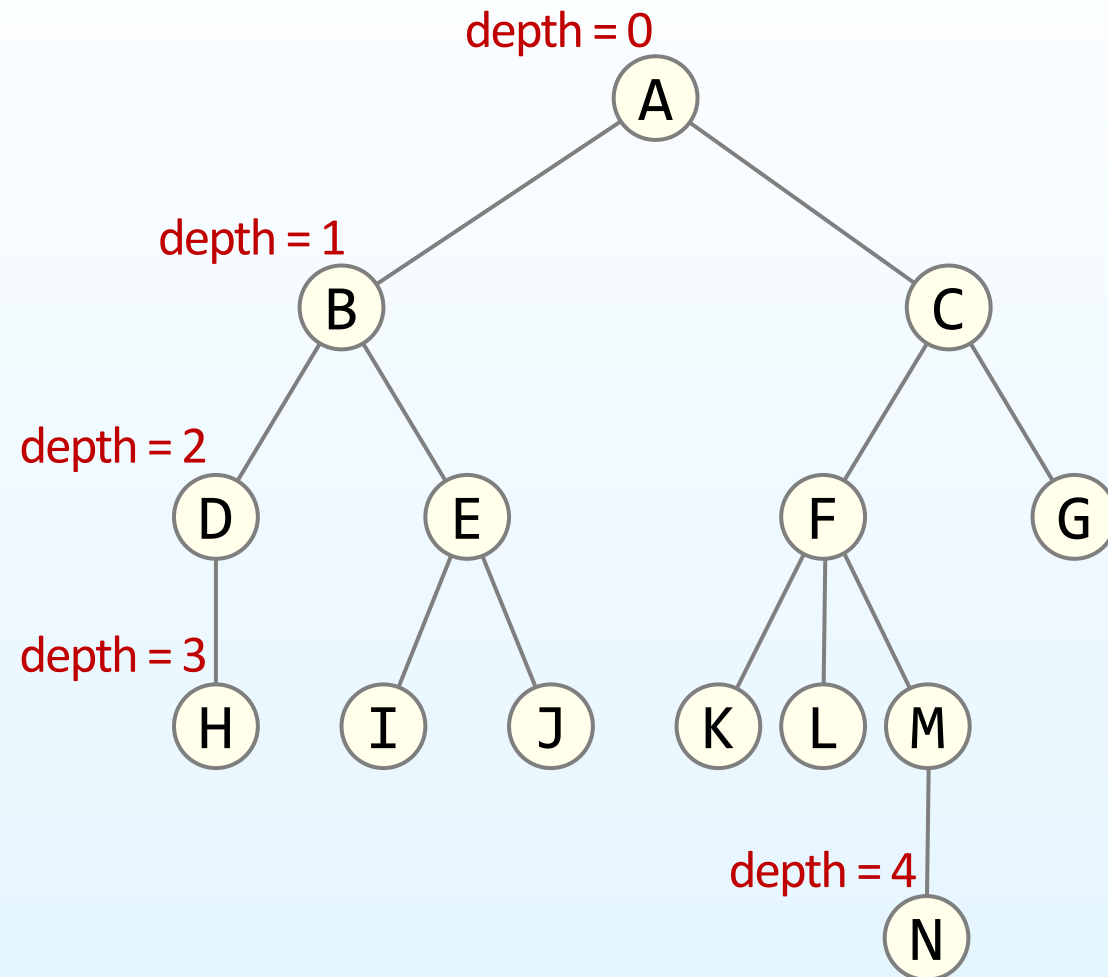
$$h(t_2) = \max(h(t_{21}), h(t_{22})) + 1$$

⋮

Base case ?



Printing a Hierarchical Structure

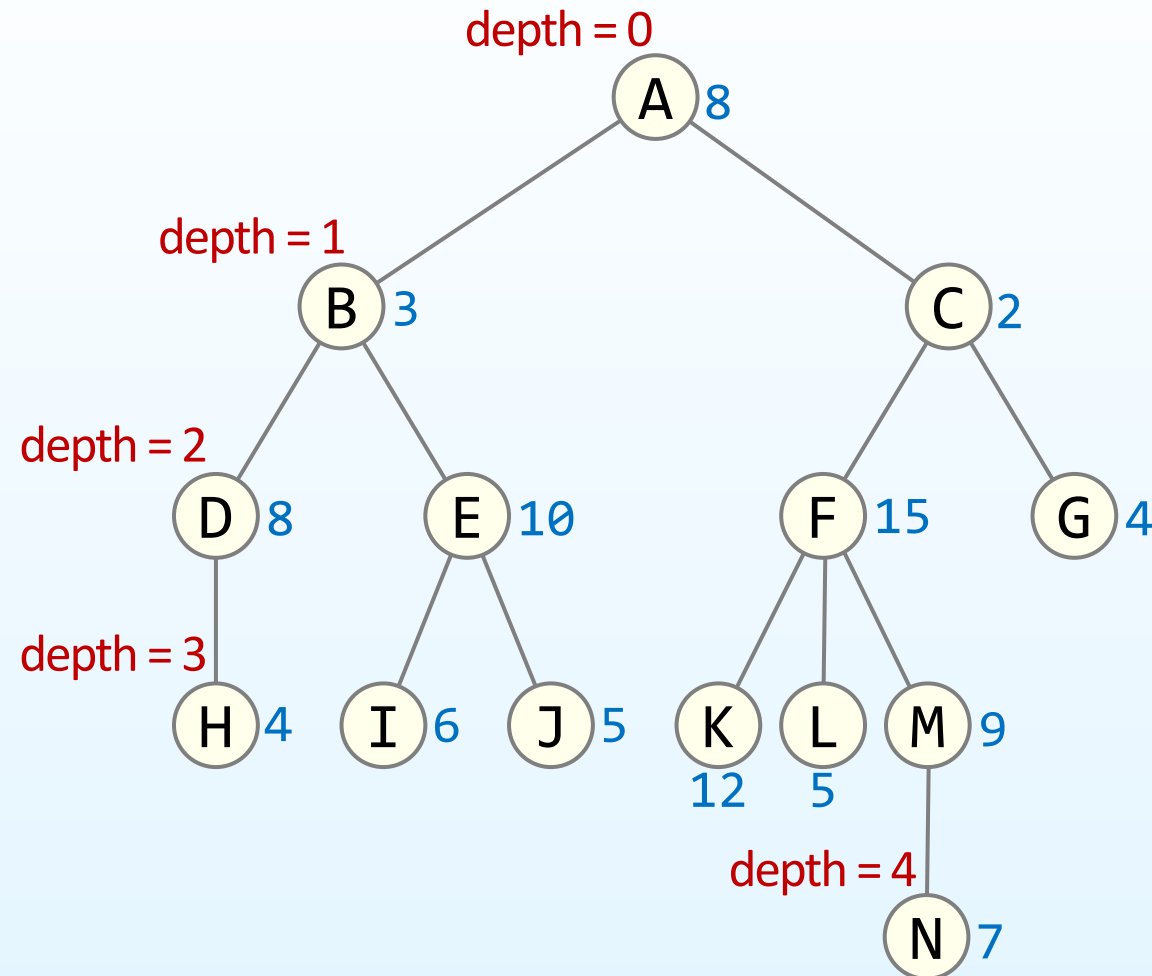


Output:

```
A
  B
    D
      H
    E
      I
      J
  C
    F
      K
      L
      M
        N
  G
```



Calculating the Total Value of Subtrees



Output:

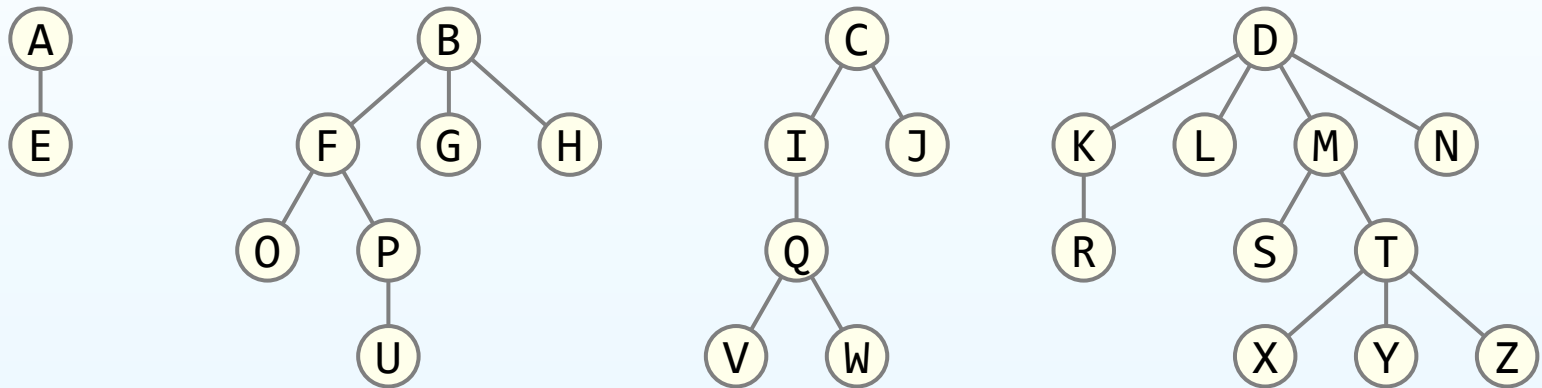
```
H(4) 4
D(8) 12
I(6) 6
J(5) 5
E(10) 21
B(3) 36
K(12) 12
L(5) 5
N(7) 7
M(9) 16
F(15) 48
G(4) 4
C(2) 54
A(8) 98
```

Outline

- Terminologies
- Implementation
- Tree Traversals
- Forests

Definition

A Forest is a data structure that is a collection of **disjoint rooted trees**

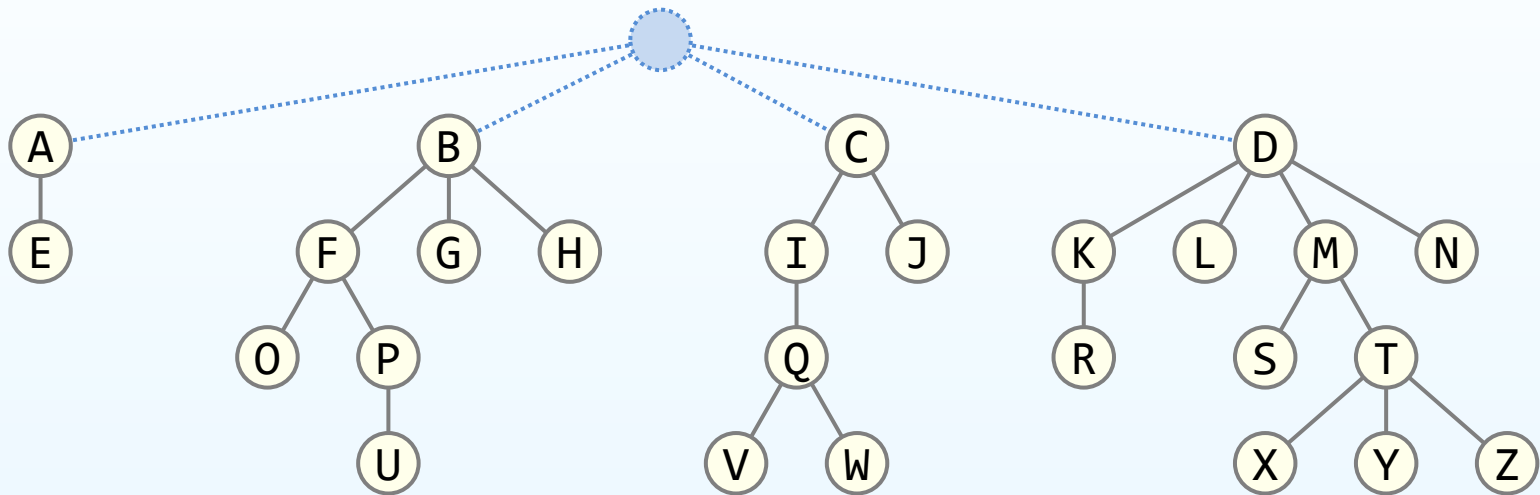


Note that:

- Any tree can be converted into a forest by **removing** the root node
- Any forest can be converted into a tree by **adding** a root node that has the roots of all the trees in the forest as children

Traversals

Traversals on forests can be achieved by treating the roots as children of a notional root

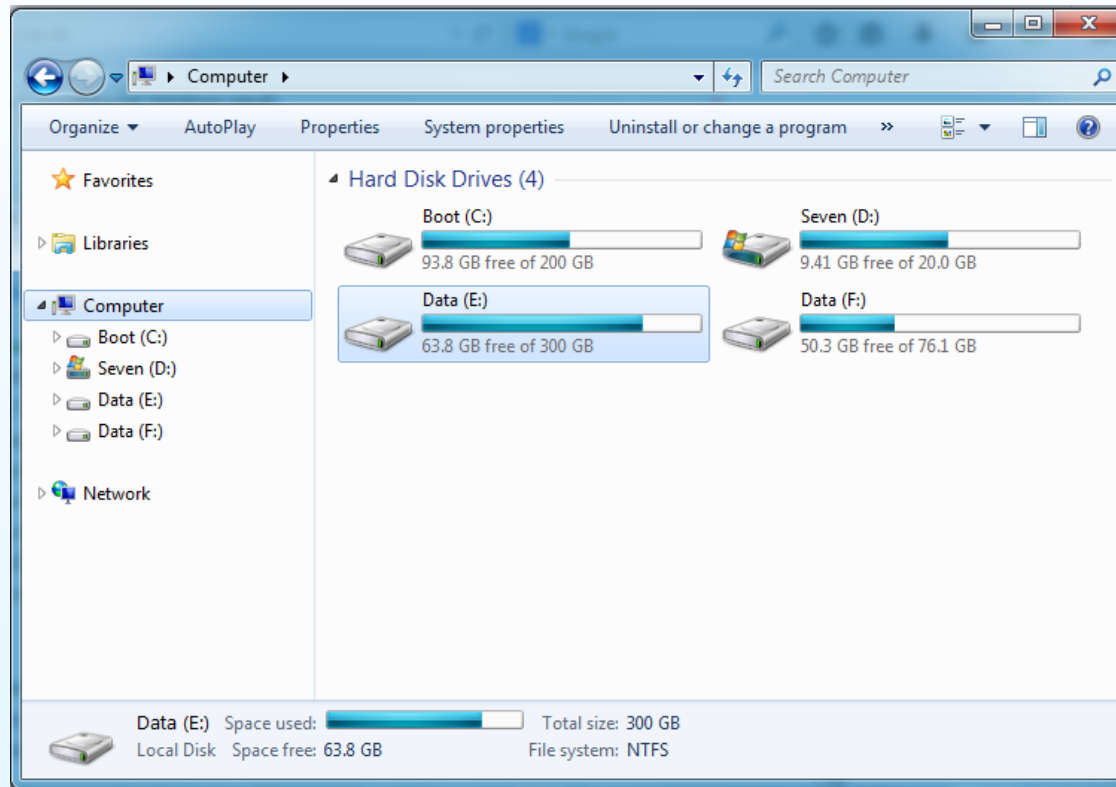


- **Breadth-first traversal:** A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
- **Depth-first traversal:** A E B F O P U G H C I Q V W J D K R L M S T X Y Z N

Application

In Windows, each drive form the root if its own directory structure

- Each of the directories is hierarchical—that is, a rooted tree



Any Question?

