[SWE2015-41] Introduction to Data Structures (자료구조개론)

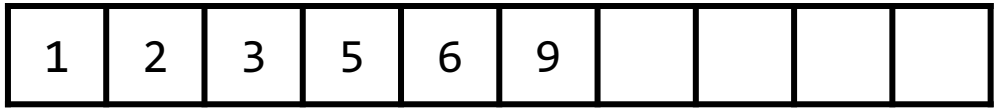
Trees

Department of Computer Science and Engineering

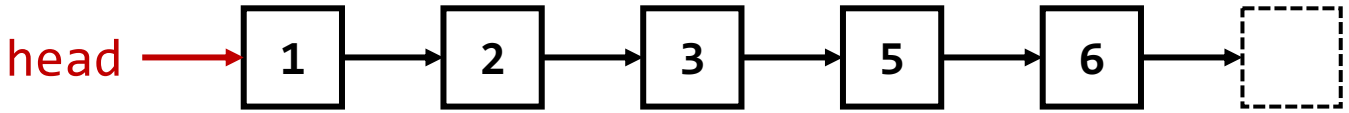
Instructor: Hankook Lee (이한국)

(Recap) Linear Structures

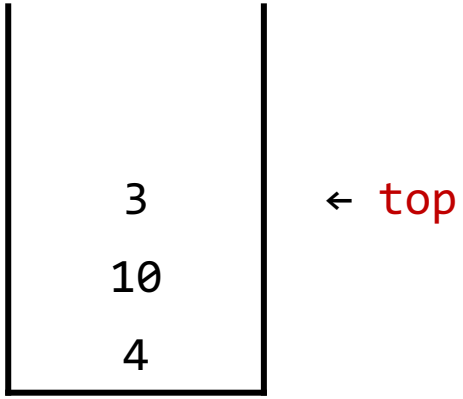
- **An array** is a collection of elements of **the same data type** in **a contiguous block of memory**

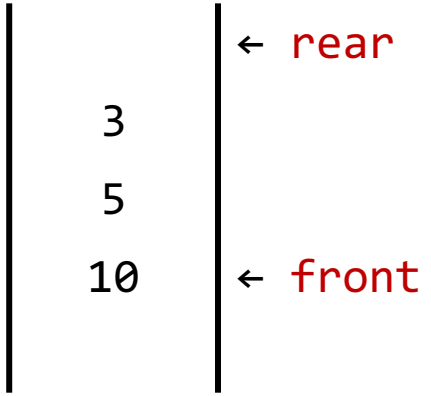


- **A linked list** is a collection of **sequentially-connected** elements



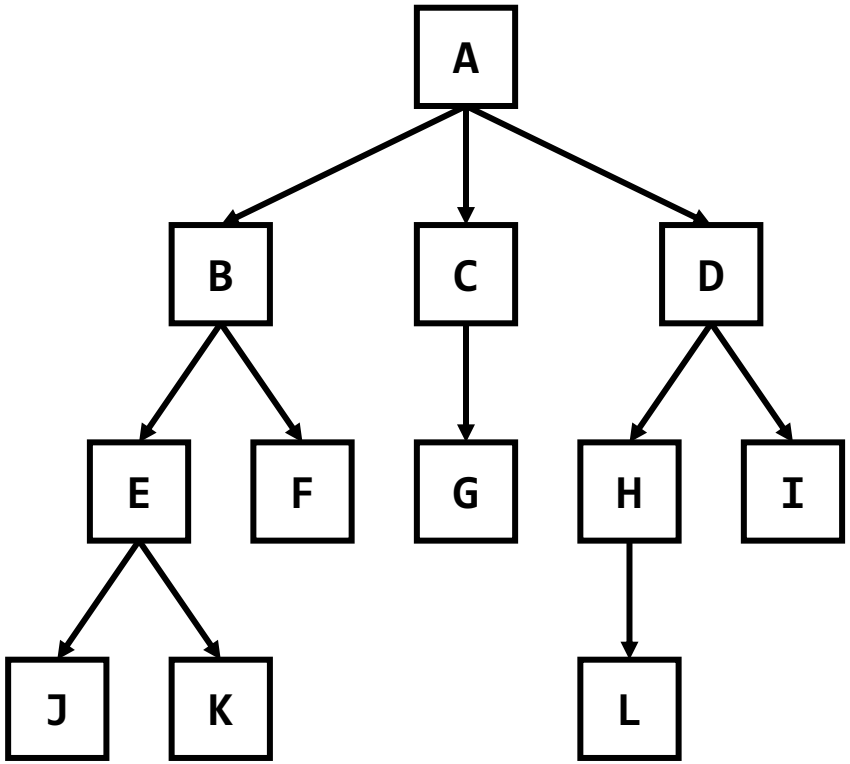
- **Stack** & **Queue** are linear structure based on LIFO & FIFO principles, resp.





What is Tree?

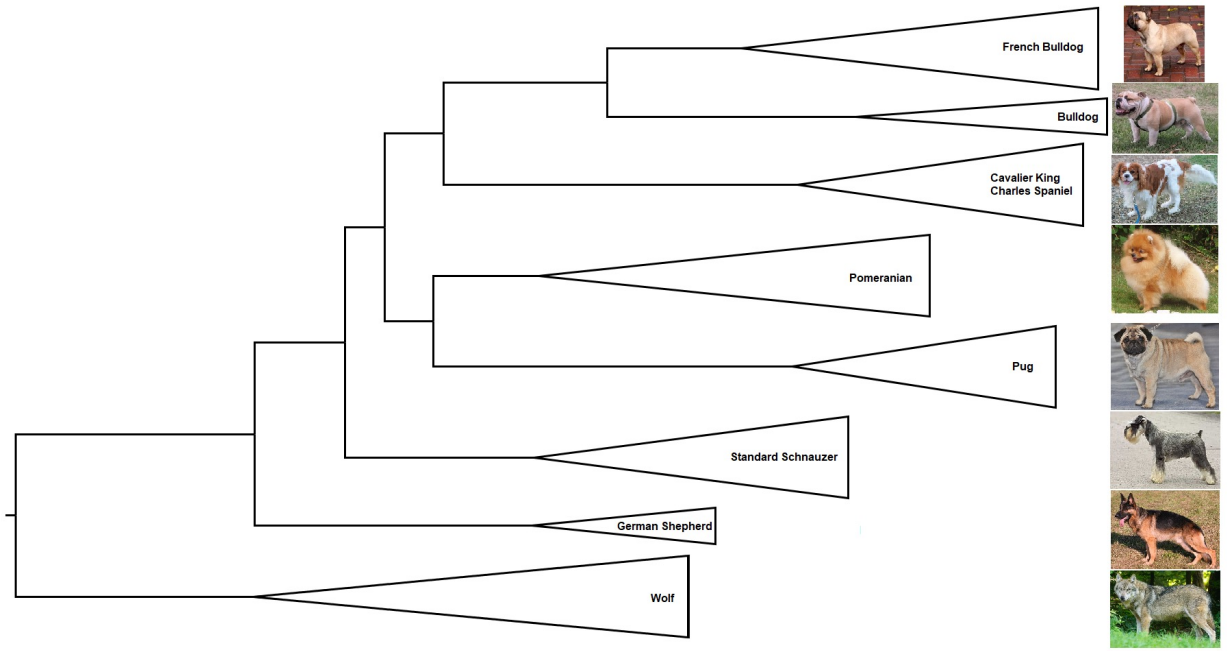
- Tree is a **hierarchical** structure with a set of connected nodes
  - Each node is composed with a parent-children relationship
  - There is no cycle (or loop) in the tree



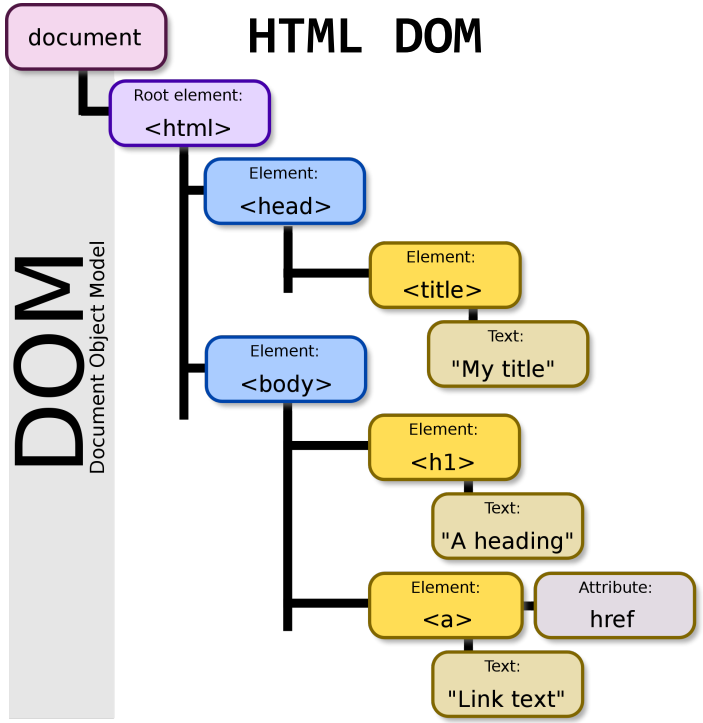
What is Tree?

- Tree is a **hierarchical** structure with a set of connected nodes
  - Each node is composed with a parent-children relationship
  - There is no cycle (or loop) in the tree

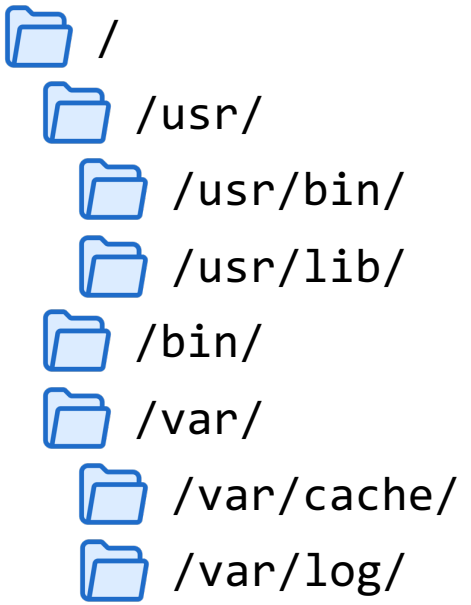
Phylogenetic tree (계통수)



HTML DOM



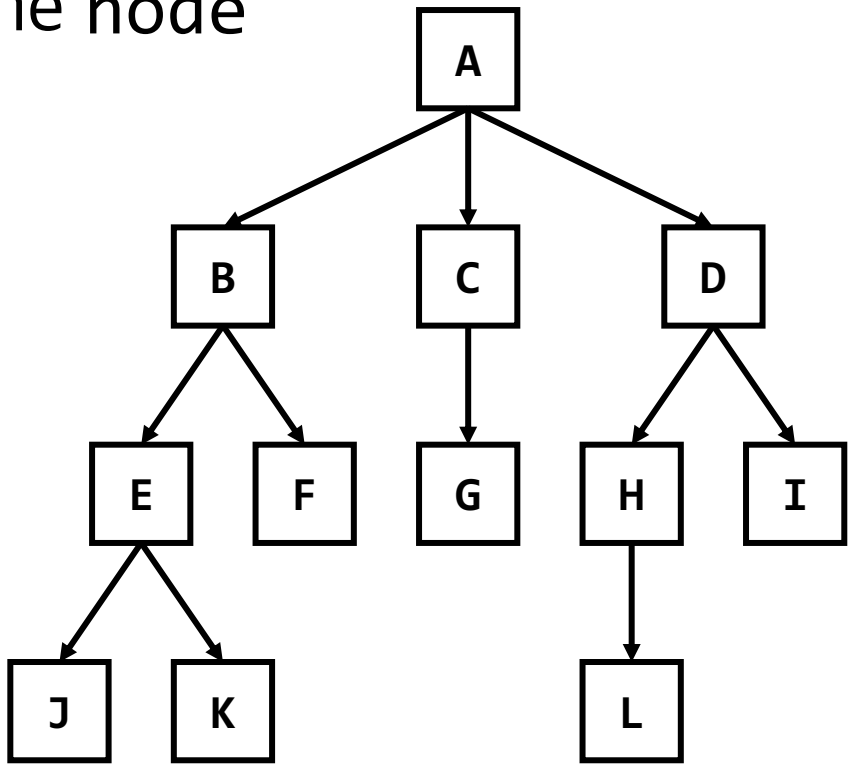
Linux Directory



Terminology (Basic)

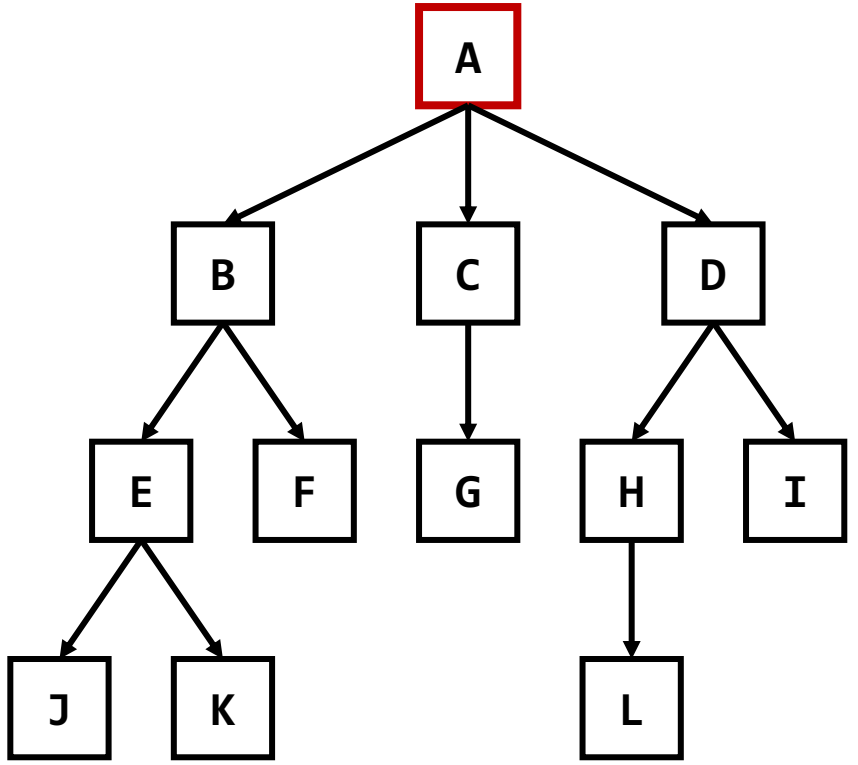
- **Node** represents an object
- **Edge** represents a connection between two nodes
  - If X → Y, say X is the **parent** of Y and Y is a **child** of X
- **Degree** of a node is the number of children of the node
  - It is equal to the number of outgoing edges

- Examples
  - B is the parent of E and F
  - H is a child of D
  - degree(A) = 3
  - degree(D) = 2
  - degree(J) = 0



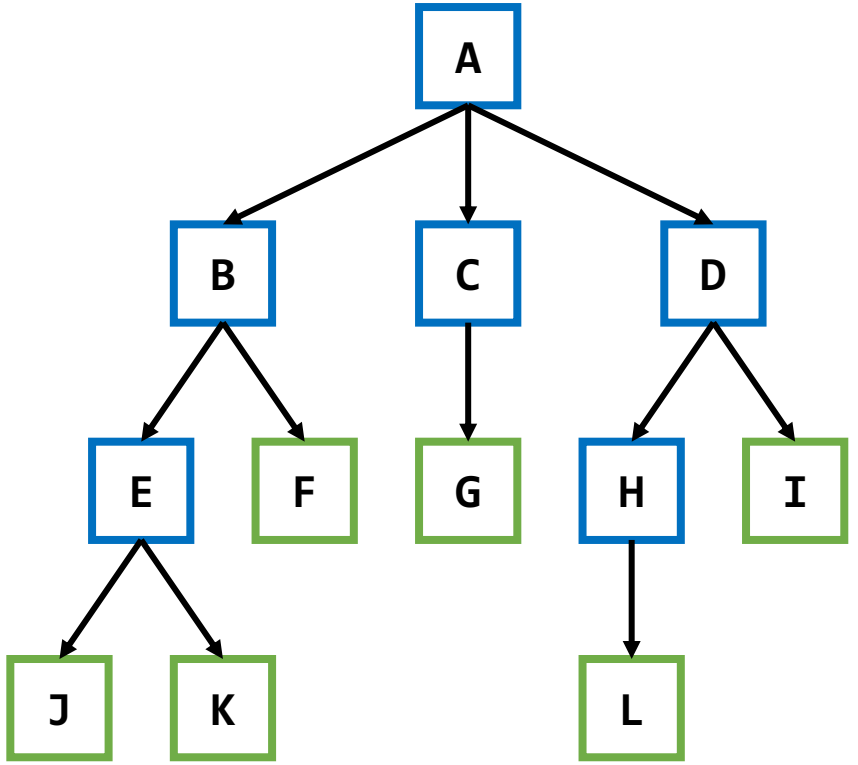
Terminology (Tree-Level)

- **Root** is the top node in a tree



Terminology (Tree-Level)

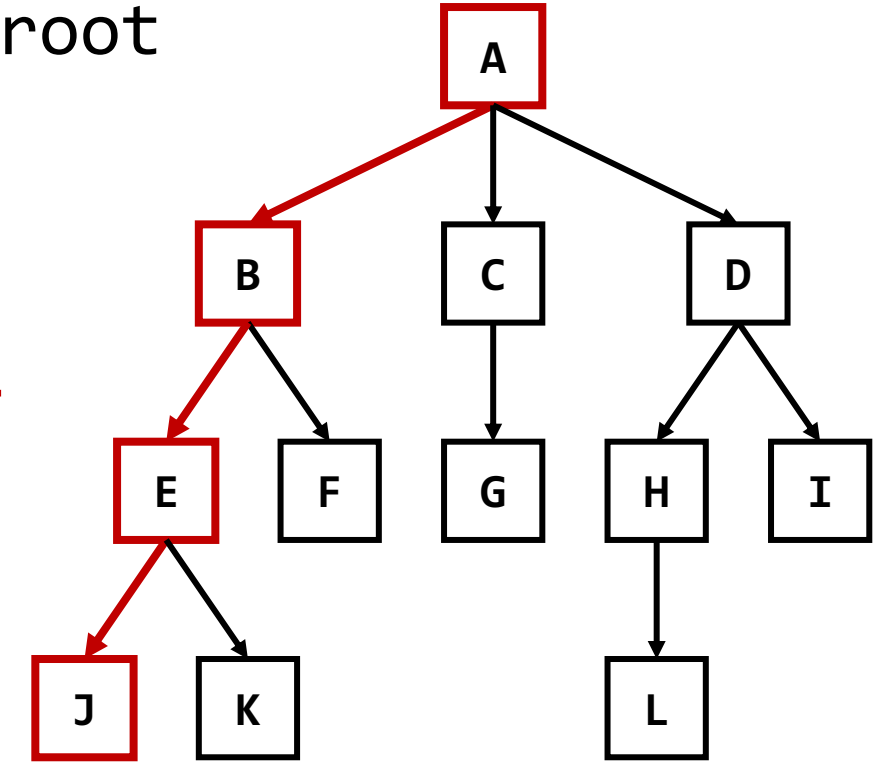
- **Root** is the top node in a tree
- **Internal** (or non-terminal) node: degree (# of children) ≥ 1
- **Leaf** (or terminal) node: degree (# of children) = 0



Terminology (Tree-Level)

- **Root** is the top node in a tree
- **Internal** (or non-terminal) node: degree ≥ 1
- **Leaf** (or terminal) node: degree = 0
- **Height** is # of nodes on the longest path from root

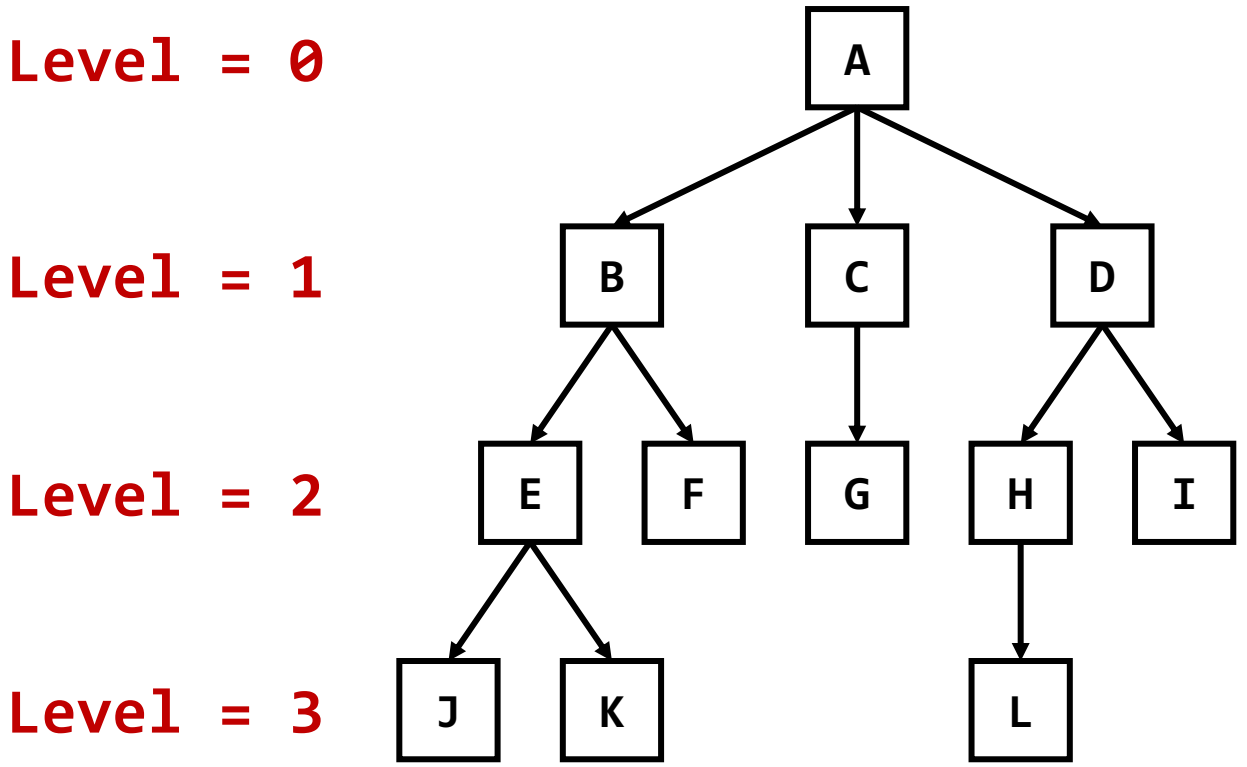
Height = 4



Terminology (Node-Level)

For a **node X**,

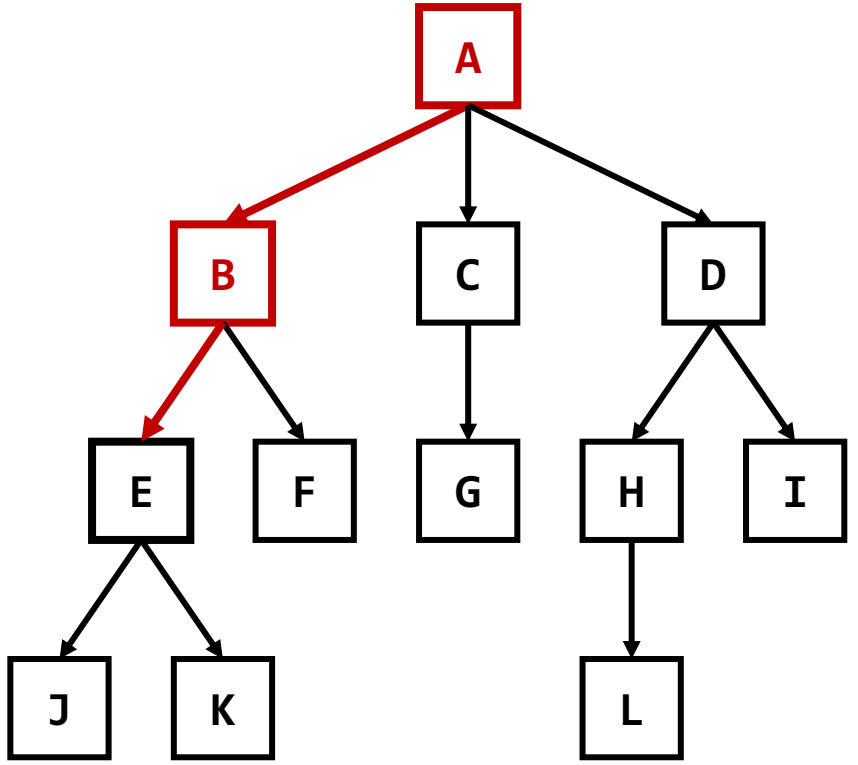
- **`Level`** or **`depth`** is the distance between **`root`** and **`X`**



Terminology (Node-Level)

For a **node X**,

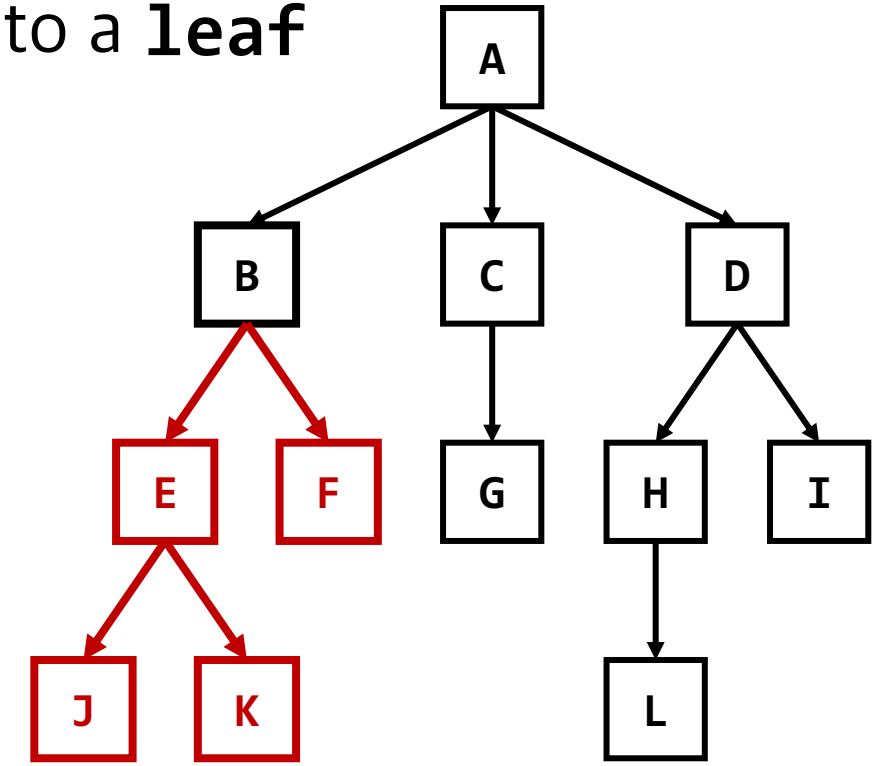
- **`Level`** or **`depth`** is the distance between **`root`** and **`X`**
- **`Ancestor`** is a predecessor on the path from **`root`** to **`X`**
  - For example, **`A`** and **`B`** are **`ancestor`**s of **`E`**



Terminology (Node-Level)

For a **node X**,

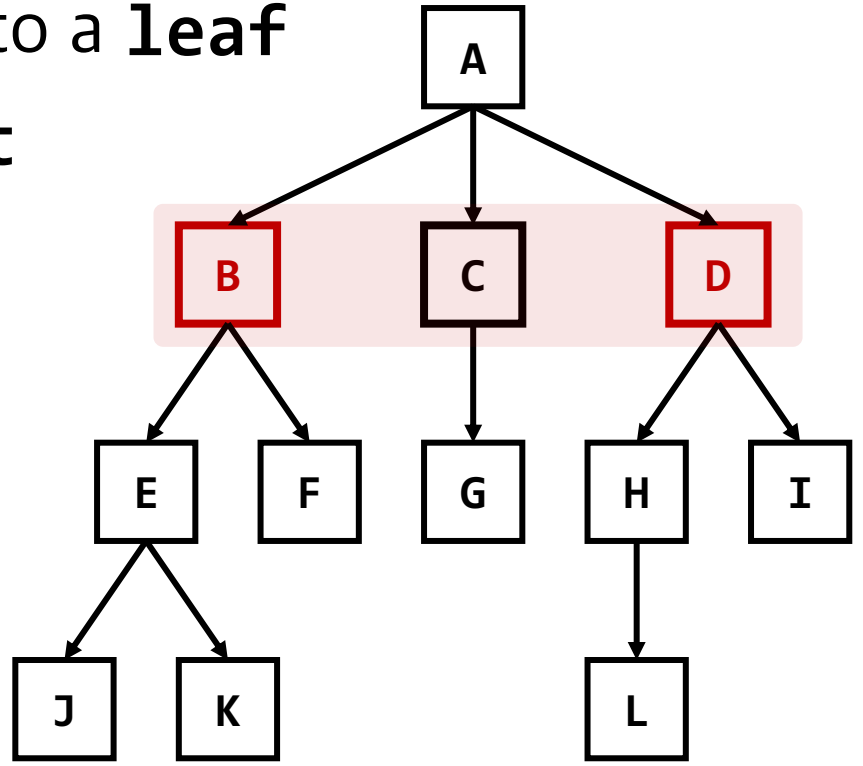
- **Level** or **depth** is the distance between **root** and **X**
- **Ancestor** is a predecessor on the path from **root** to **X**
- **Descendant** is a successor on any path from **X** to a **leaf**
  - For example, **E**, **F**, **J**, and **K** are **descendant**s of **B**



Terminology (Node-Level)

For a **node X**,

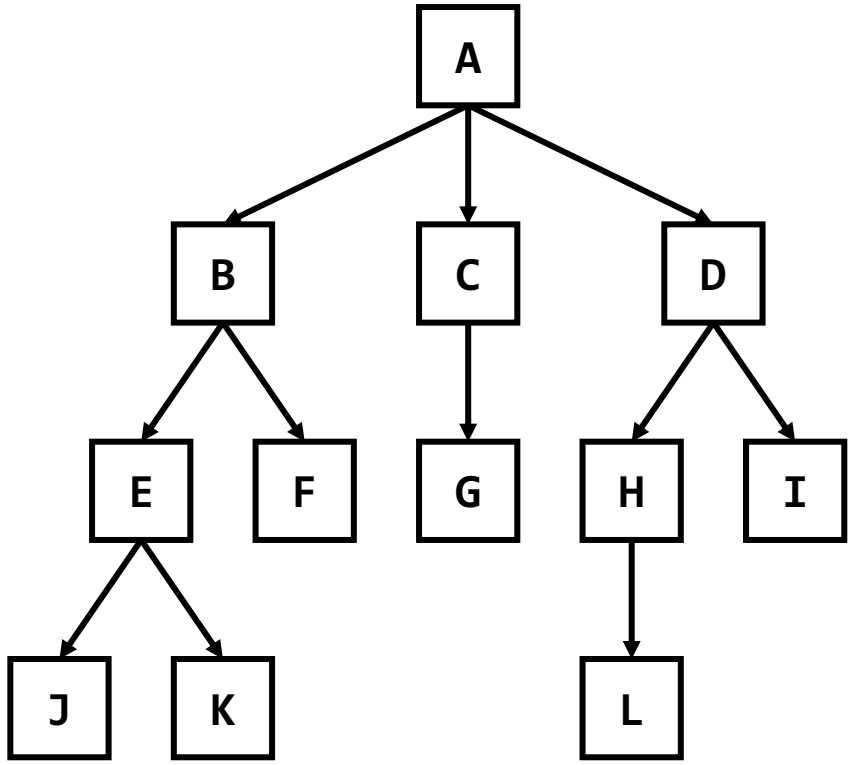
- **Level** or **depth** is the distance between **root** and **X**
- **Ancestor** is a predecessor on the path from **root** to **X**
- **Descendant** is a successor on any path from **X** to a **leaf**
- **Sibling** is another **node** with the same **parent**
  - For example, **B** and **D** are **sibling**s of **C**



Terminology (Node-Level)

Subtree rooted at a **node X**

- Any **node** can be treated as the **root node** of its own **subtree**
- The **subtree** includes **X** and all **descendant**s of **X**

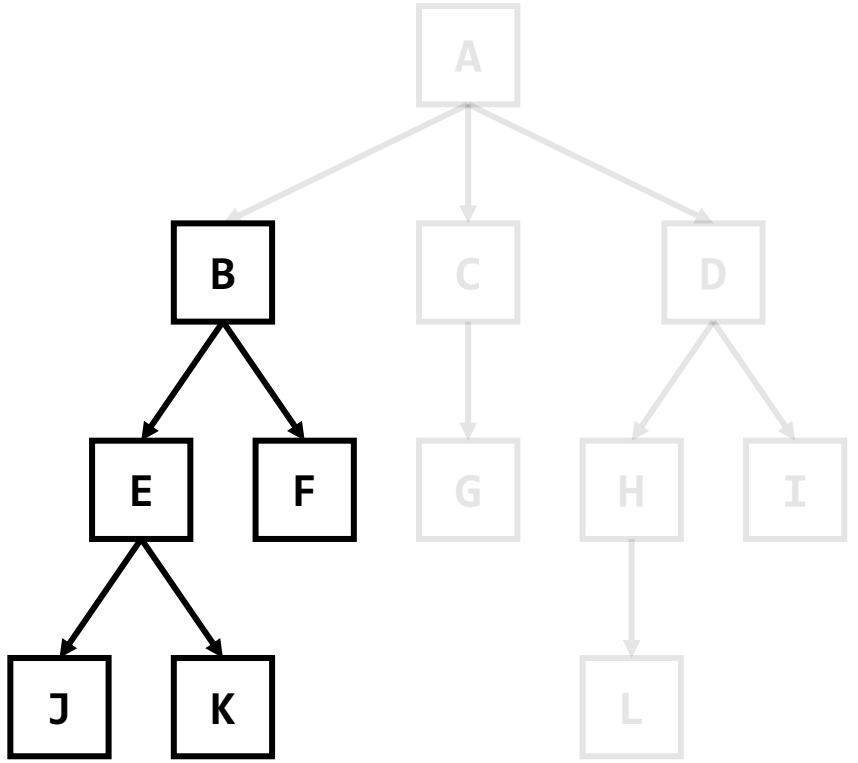


Terminology (Node-Level)

Subtree rooted at a **node X**

- Any **node** can be treated as the **root node** of its own **subtree**
- The **subtree** includes **X** and all **descendant**s of **X**

Subtree rooted at **node B**

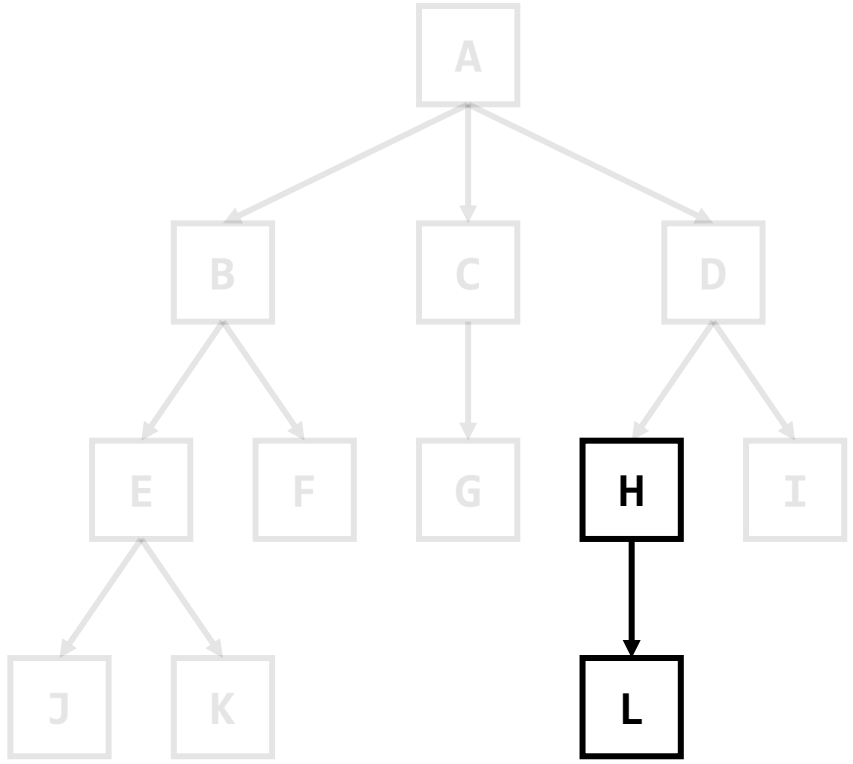


Terminology (Node-Level)

Subtree rooted at a **node X**

- Any **node** can be treated as the **root node** of its own **subtree**
- The **subtree** includes **X** and all **descendant**s of **X**

Subtree rooted at **node H**

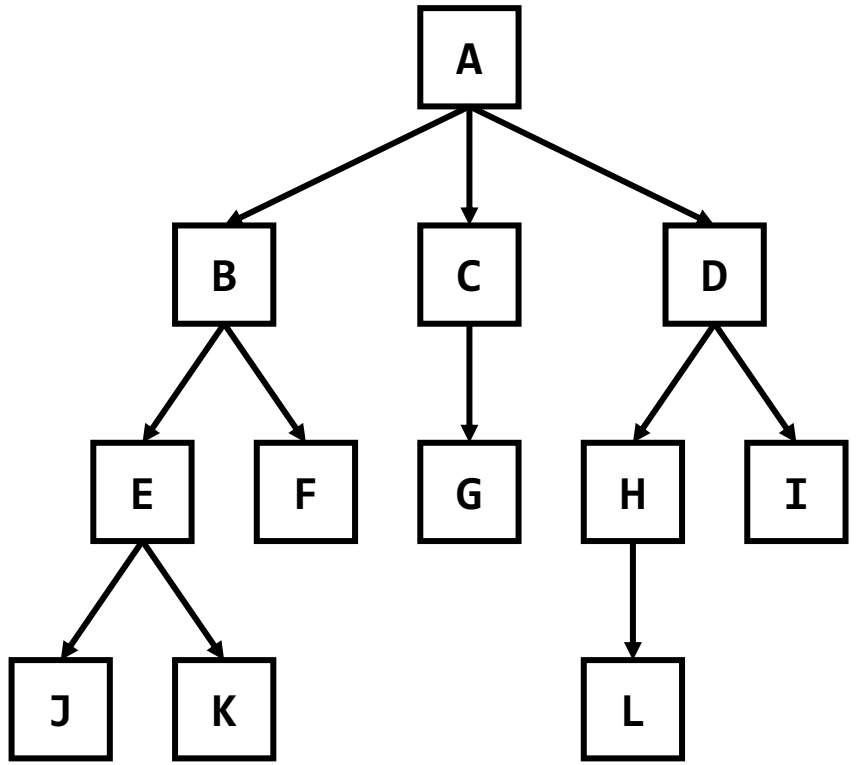


Implementation - Structure

- How to implement the tree structure?
  - **root** - the top node of the tree
  - **node** - each node contains **its item value** and **pointers for child nodes**
  - Use **the array structure** to store the child pointers

```
#define MAX_DEGREE 10

typedef struct _Node {
    int item;
    struct _Node *children[MAX_DEGREE];
} Node;

Node *root;
```

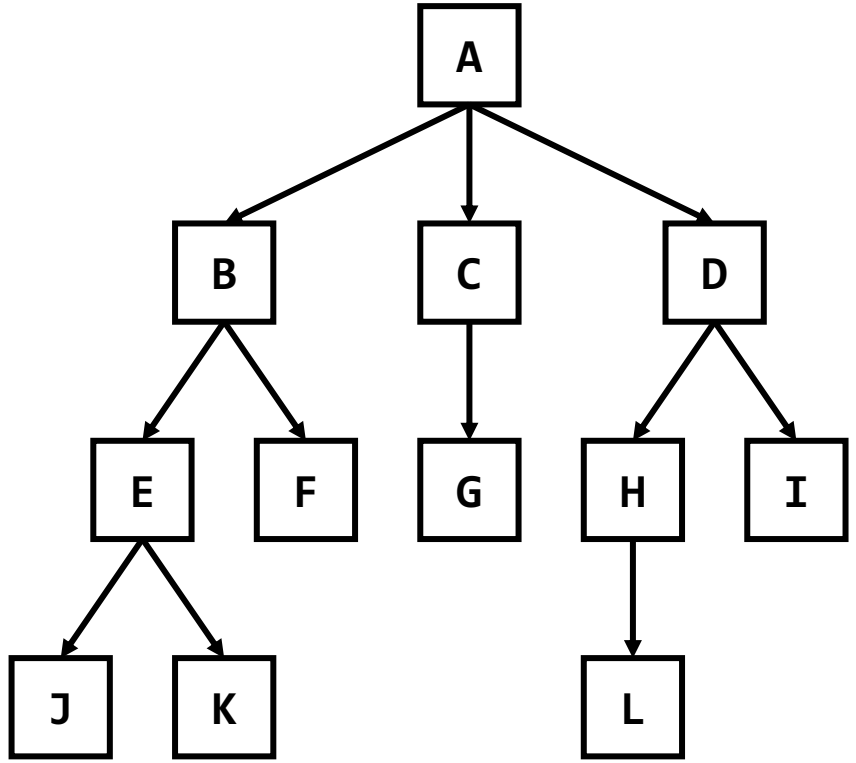
  - **Note.** This implementation limits the maximum degree



Implementation - Structure

- How to implement the tree structure?
  - **root** - the top node of the tree
  - **node** - each node contains **its item value** and **pointers for child nodes**
    - Use **the array structure** to store the child pointers

```
Node *A, *B, ..., *L; // require malloc()

root = A;
A->children[0] = B;
A->children[1] = C;
A->children[2] = D;
B->children[0] = E;
B->children[1] = F;
...
E->children[1] = K;
H->children[0] = L;
```

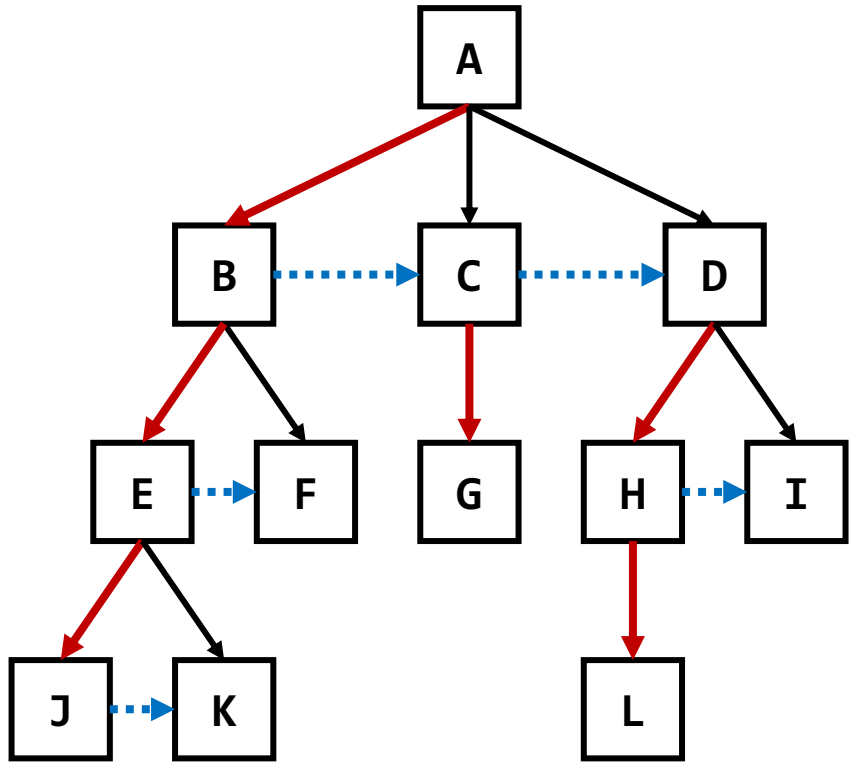


Implementation - Structure

- How to implement the tree structure?
  - **root** - the top node of the tree
  - **node** - each node contains **its item value** and **pointers for child nodes**
    - Use **the list structure** to store the child pointers

```
typedef struct _Node {
    int item;
    struct _Node *left_child, *right_sibling;
} Node;

Node *root;
```

    - **Note.** This is known as Left-Child Right-Sibling (LCRS) representation

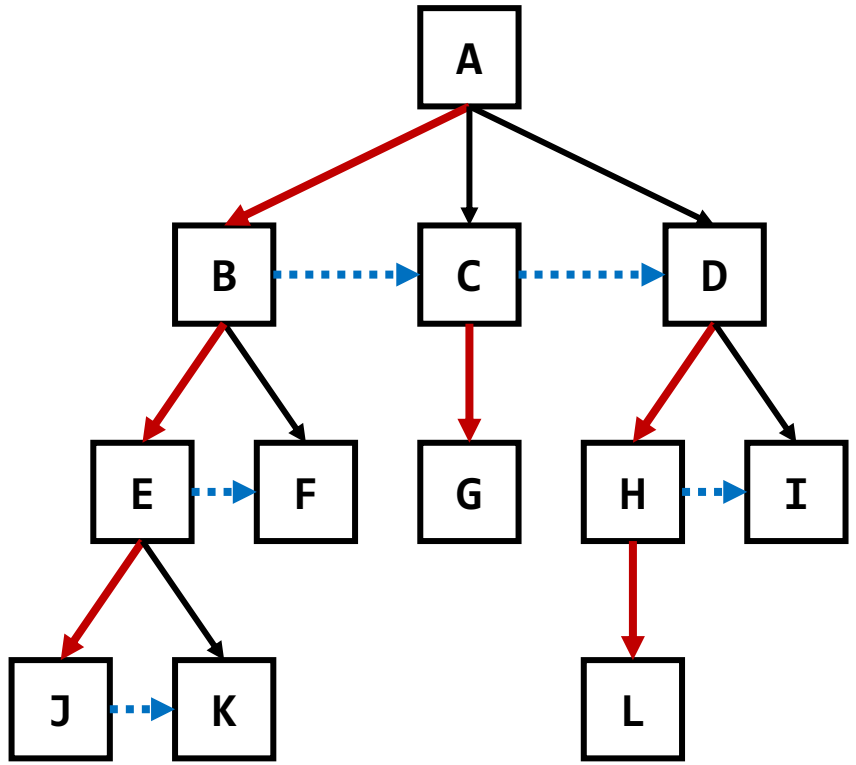


Implementation - Structure

- How to implement the tree structure?
  - **root** - the top node of the tree
  - **node** - each node contains **its item value** and **pointers for child nodes**

  - Use **the list structure** to store the child pointers

```
Node *A, *B, ..., *L; // require malloc()

root = A;
A->left_child = B;
B->left_child = E;
B->right_sibling = C;
C->left_child = G;
C->right_sibling = D;
...
H->right_sibling = I;
J->right_sibling = K;
```



Implementation - Traversal

- How to traverse **child nodes** of a node?
  - With the array structure

```
typedef struct _Node { int item; struct _Node *children[MAX_DEGREE]; } Node;
void printChildren(Node *node) {
    for (int i = 0; i < MAX_DEGREE; i ++) {
        if (node->children[i] != NULL)
            printf("%d", node->children[i]->item);
    }
}
```

  - With the list structure

```
typedef struct _Node { int item; struct _Node *left_child, *right_sibling; } Node;
void printChildren(Node *node) {

    ?

}
```

Implementation - Traversal

- How to traverse **all nodes in the subtree** rooted at a node?
  - With the array structure

```
typedef struct _Node { int item; struct _Node *children[MAX_DEGREE]; } Node;
void printSubtree(Node *node) {
    printf("%d", node->item); // Print information of the current node
    for (int i = 0; i < MAX_DEGREE; i ++)
        if (node->children[i] != NULL)
            printSubtree(node->children[i]); // Recursive function call
}
```

  - With the list structure

```
typedef struct _Node { int item; struct _Node *left_child, *right_sibling; } Node;
void printSubtree(Node *node, bool is_root) {

    ?

}
```

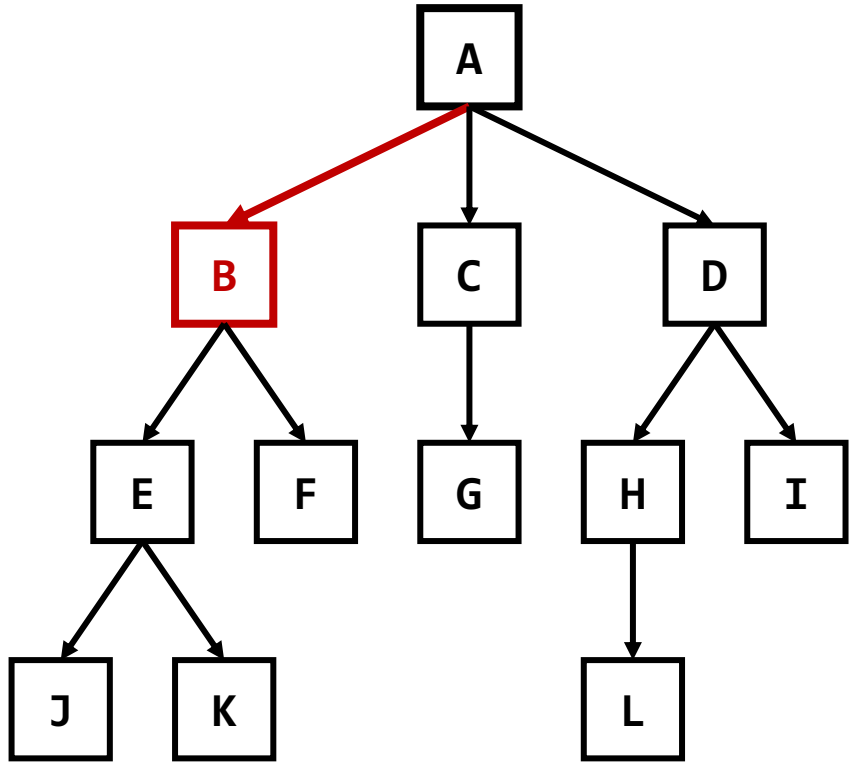
Implementation - Traversal

- What is the order of the traversal?

```
void printSubtree(Node *node) {
    printf("%d", node->item);
    for (int i = 0; i < MAX_DEGREE; i ++)
        if (node->children[i] != NULL)
            printSubtree(node->children[i]);
}
```

Recursive Function Calls:

- **A** → **B** (current)



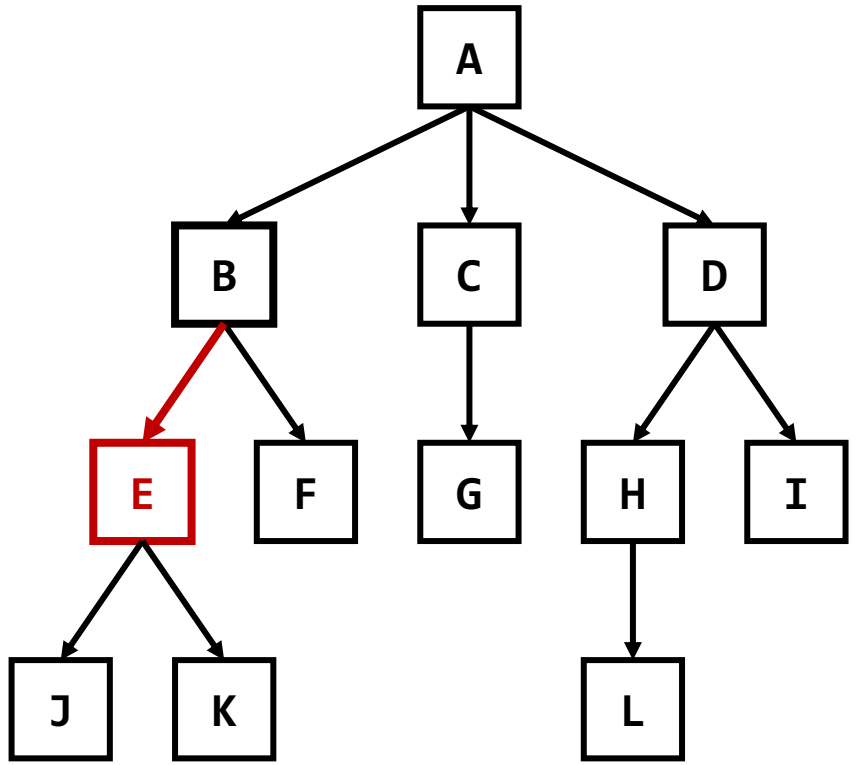
Implementation - Traversal

- What is the order of the traversal?

```
void printSubtree(Node *node) {
    printf("%d", node->item);
    for (int i = 0; i < MAX_DEGREE; i ++)
        if (node->children[i] != NULL)
            printSubtree(node->children[i]);
}
```

Recursive Function Calls:

- A → B
- **B** → **E** (current)



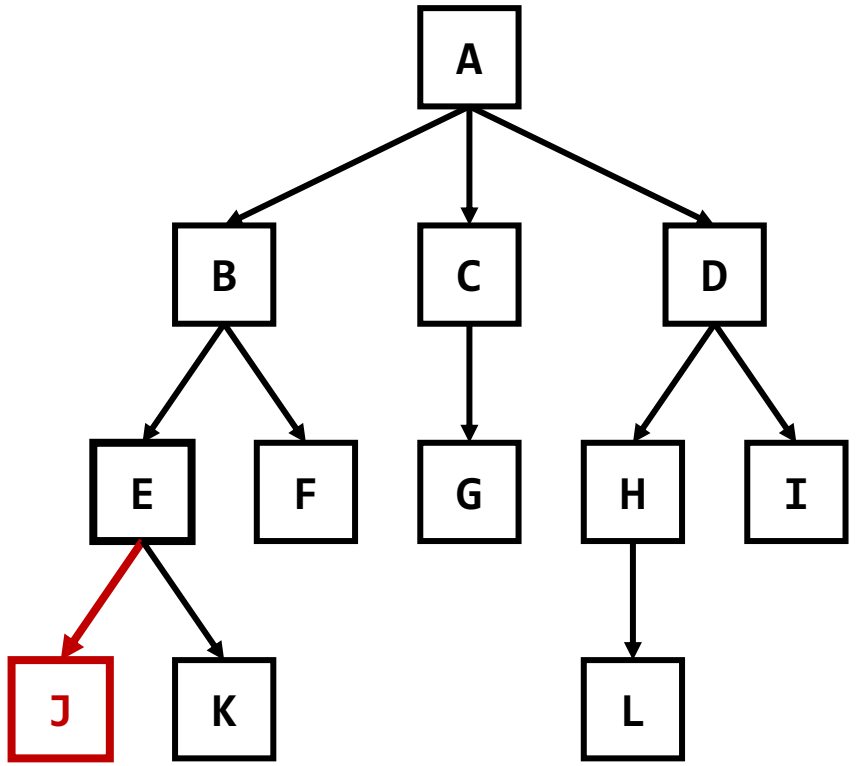
Implementation - Traversal

- What is the order of the traversal?

```
void printSubtree(Node *node) {
    printf("%d", node->item);
    for (int i = 0; i < MAX_DEGREE; i ++)
        if (node->children[i] != NULL)
            printSubtree(node->children[i]);
}
```

Recursive Function Calls:

- A → B
- B → E
- **E** → **J** (current)



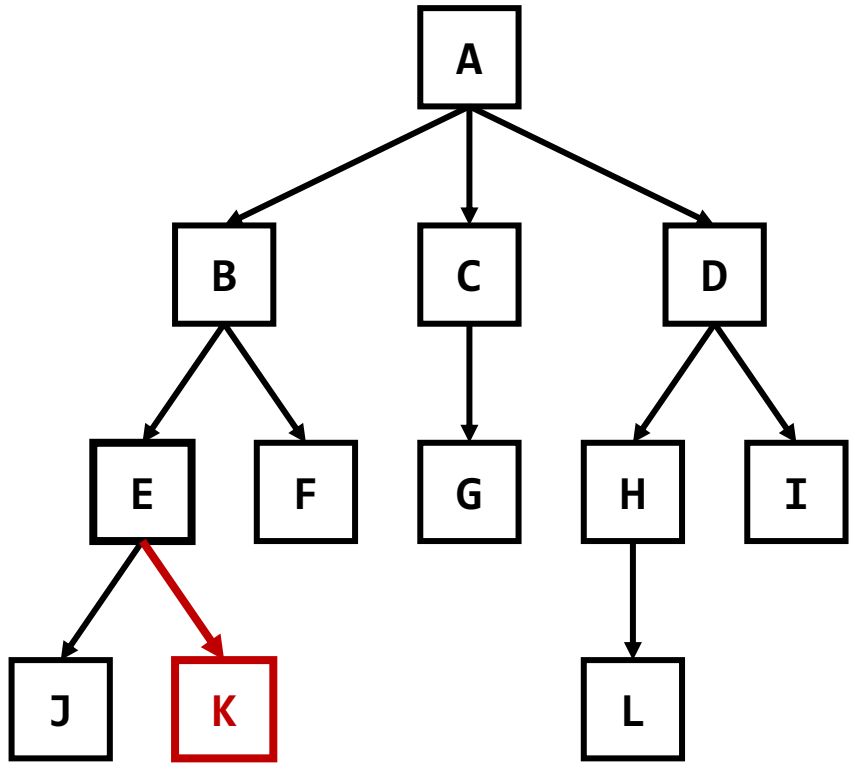
Implementation - Traversal

- What is the order of the traversal?

```
void printSubtree(Node *node) {
    printf("%d", node->item);
    for (int i = 0; i < MAX_DEGREE; i ++)
        if (node->children[i] != NULL)
            printSubtree(node->children[i]);
}
```

Recursive Function Calls:

- A → B
- B → E
- E → ~~J~~ **K** (current)



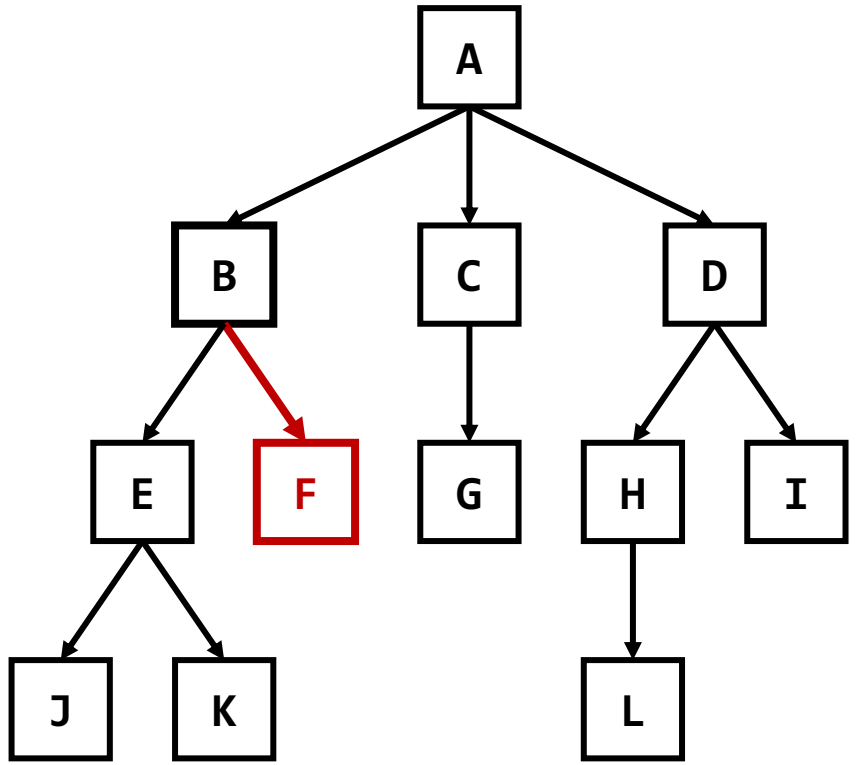
Implementation - Traversal

- What is the order of the traversal?

```
void printSubtree(Node *node) {
    printf("%d", node->item);
    for (int i = 0; i < MAX_DEGREE; i ++)
        if (node->children[i] != NULL)
            printSubtree(node->children[i]);
}
```

Recursive Function Calls:

- A → B
- B → E **F** (current)
- ~~E → J K~~



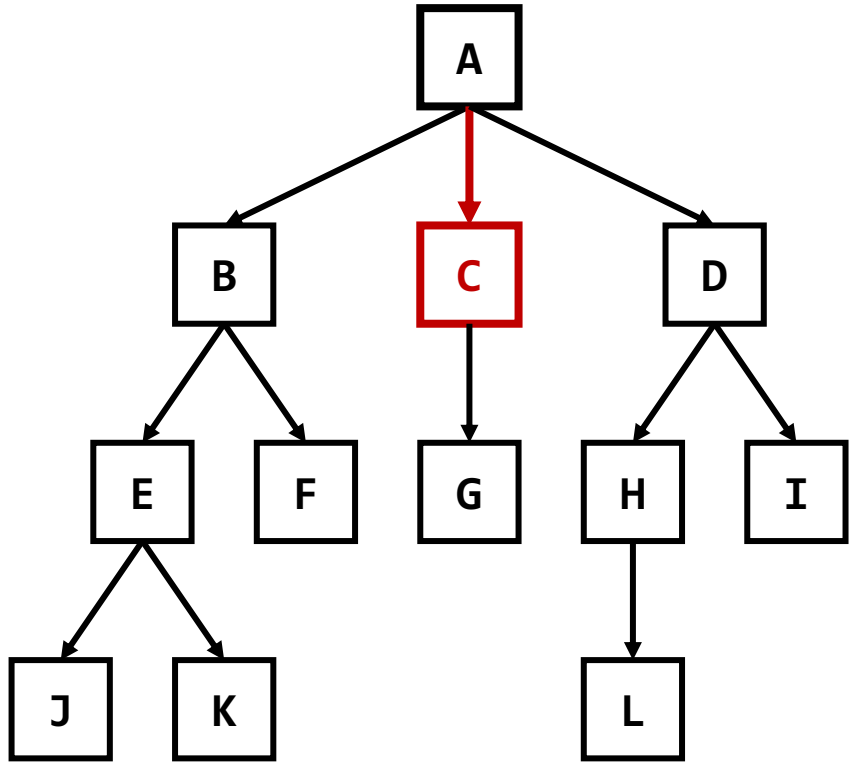
Implementation - Traversal

- What is the order of the traversal?

```
void printSubtree(Node *node) {
    printf("%d", node->item);
    for (int i = 0; i < MAX_DEGREE; i ++)
        if (node->children[i] != NULL)
            printSubtree(node->children[i]);
}
```

Recursive Function Calls:

- **A** → ~~B~~ **C** (current)
- ~~B → E F~~
- ~~E → J K~~



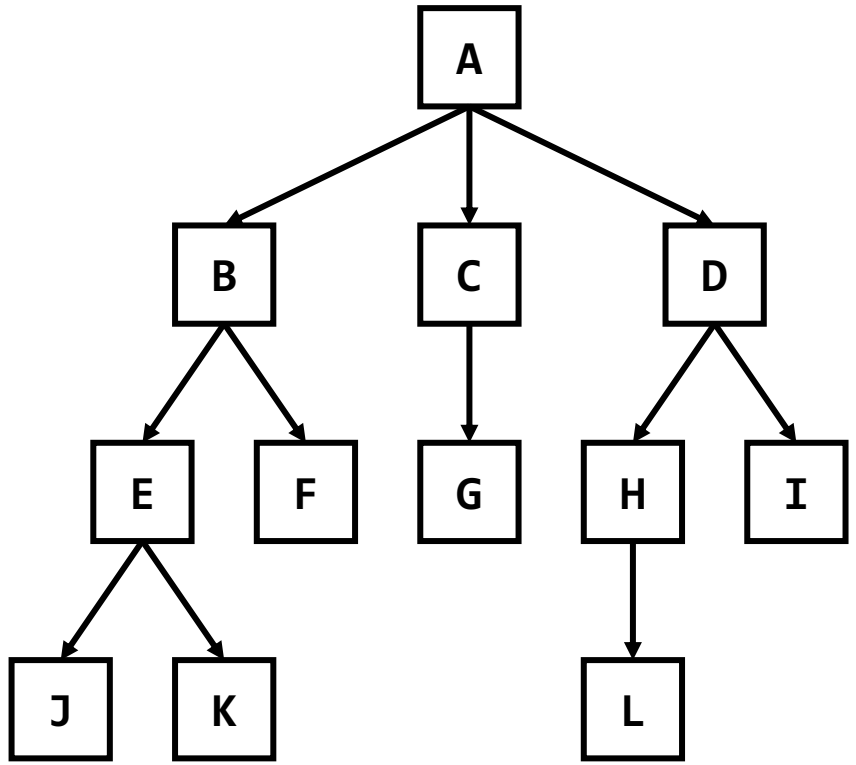
Implementation - Traversal

- What is the order of the traversal?

```
void printSubtree(Node *node) {
    printf("%d", node->item);
    for (int i = 0; i < MAX_DEGREE; i ++)
        if (node->children[i] != NULL)
            printSubtree(node->children[i]);
}
```

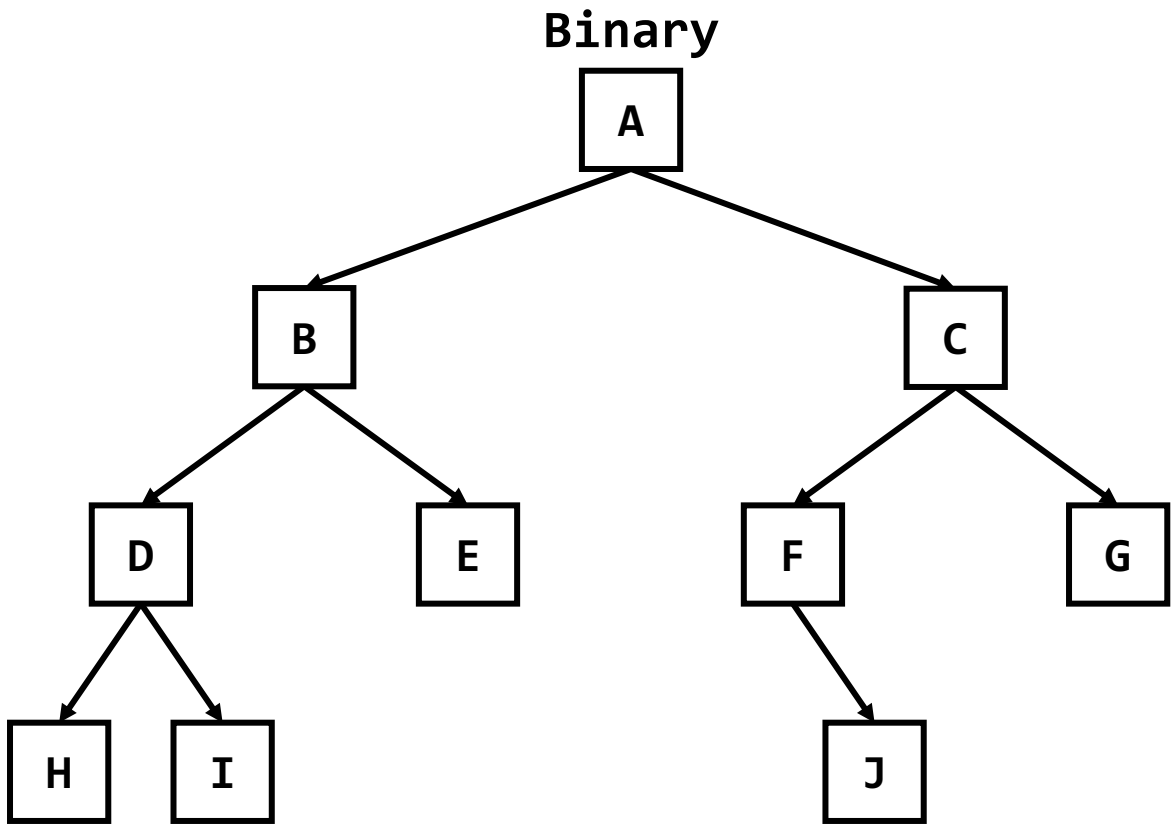
The order will be …

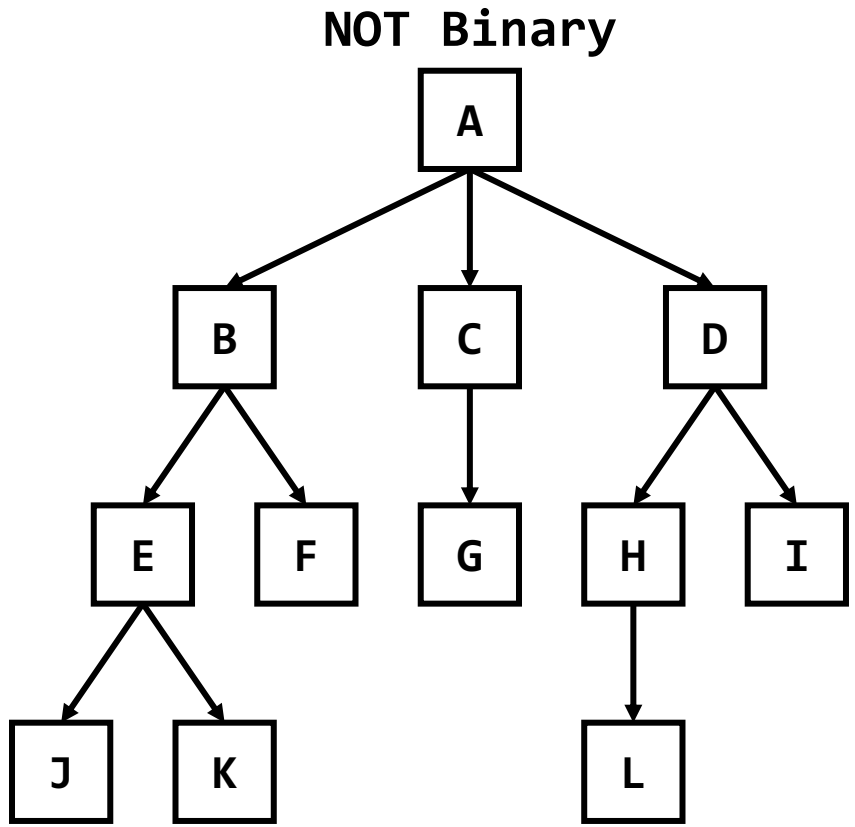
- A B E J K F C G D H L I
- This is known as **depth-first search** (**DFS**)



Binary Trees

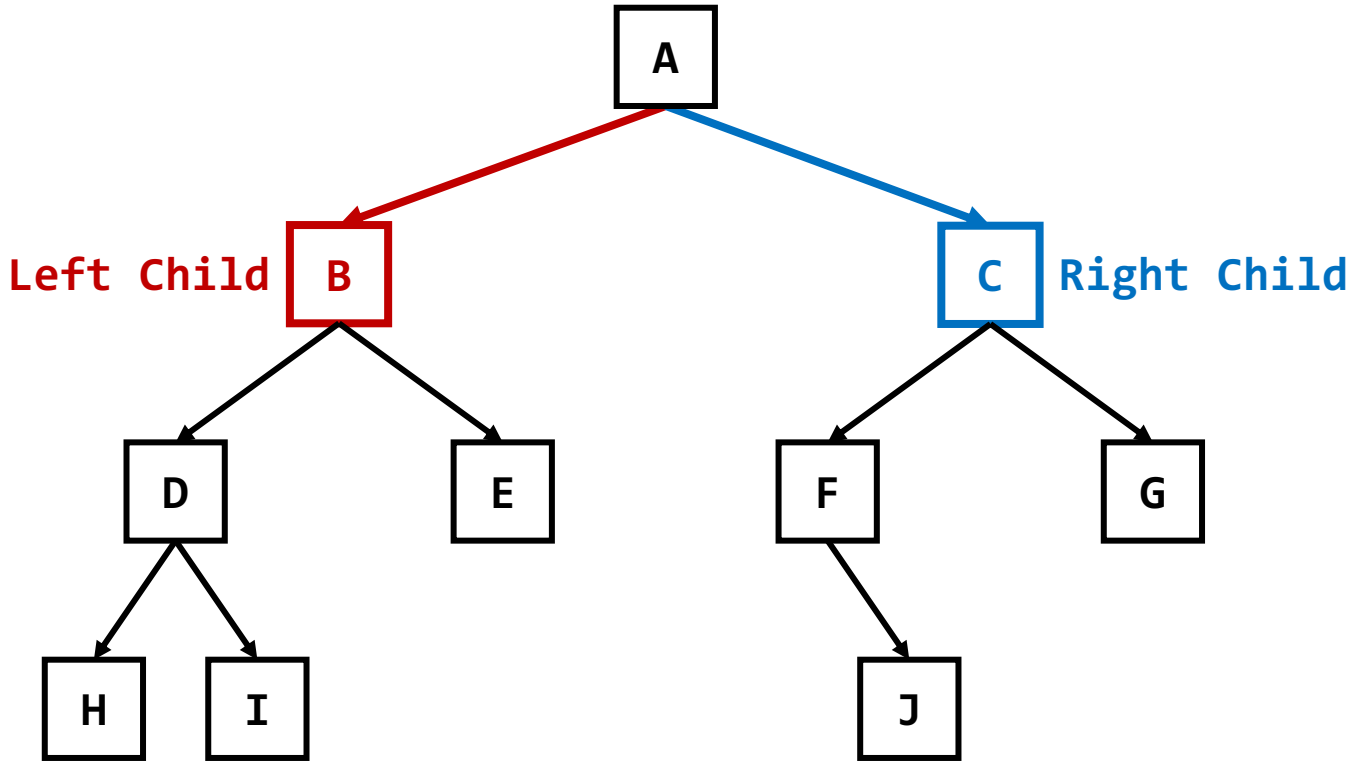
- **Binary Tree** is a tree in which each node has at most two children
  - degree(**X**) ≤ 2 for any node **X** in a binary tree





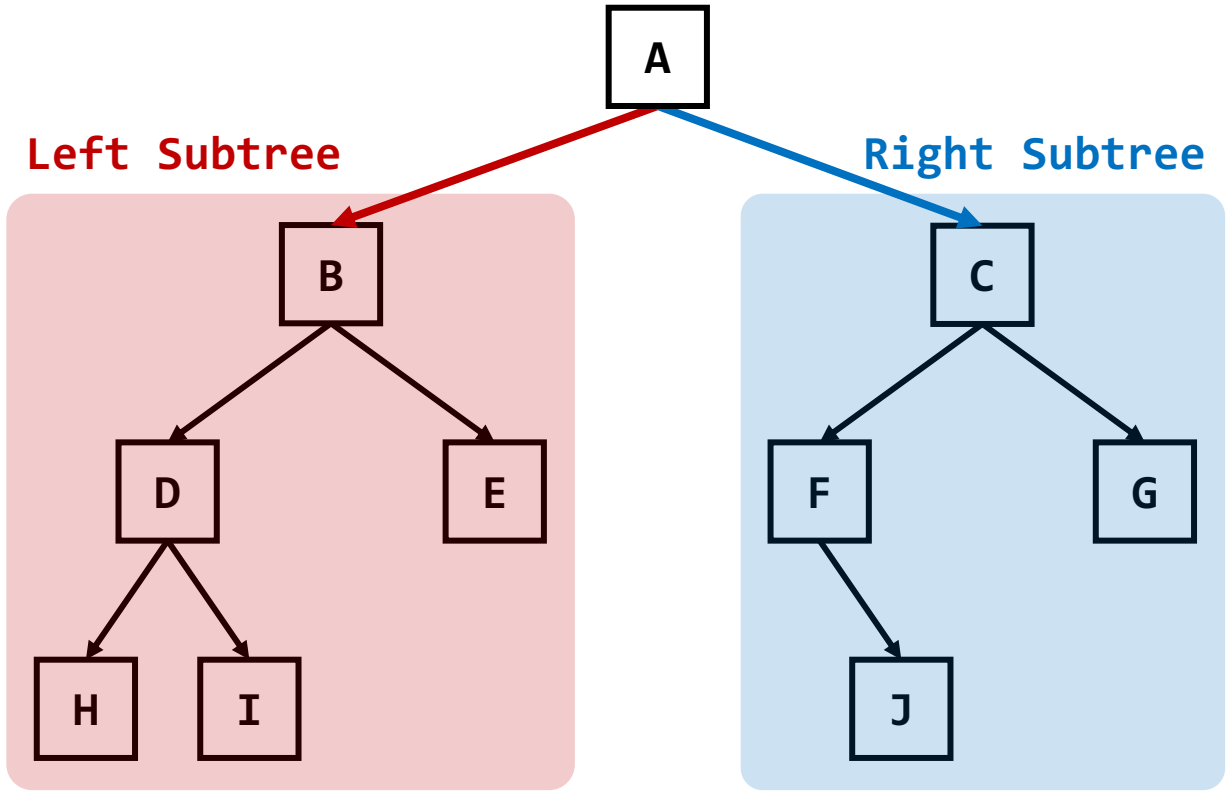
Binary Trees

- **Binary Tree** is a tree in which each node has at most two children
  - `degree(X)` ≤ 2 for any node **X** in a binary tree
  - Each node has **`left`** & **`right`** children



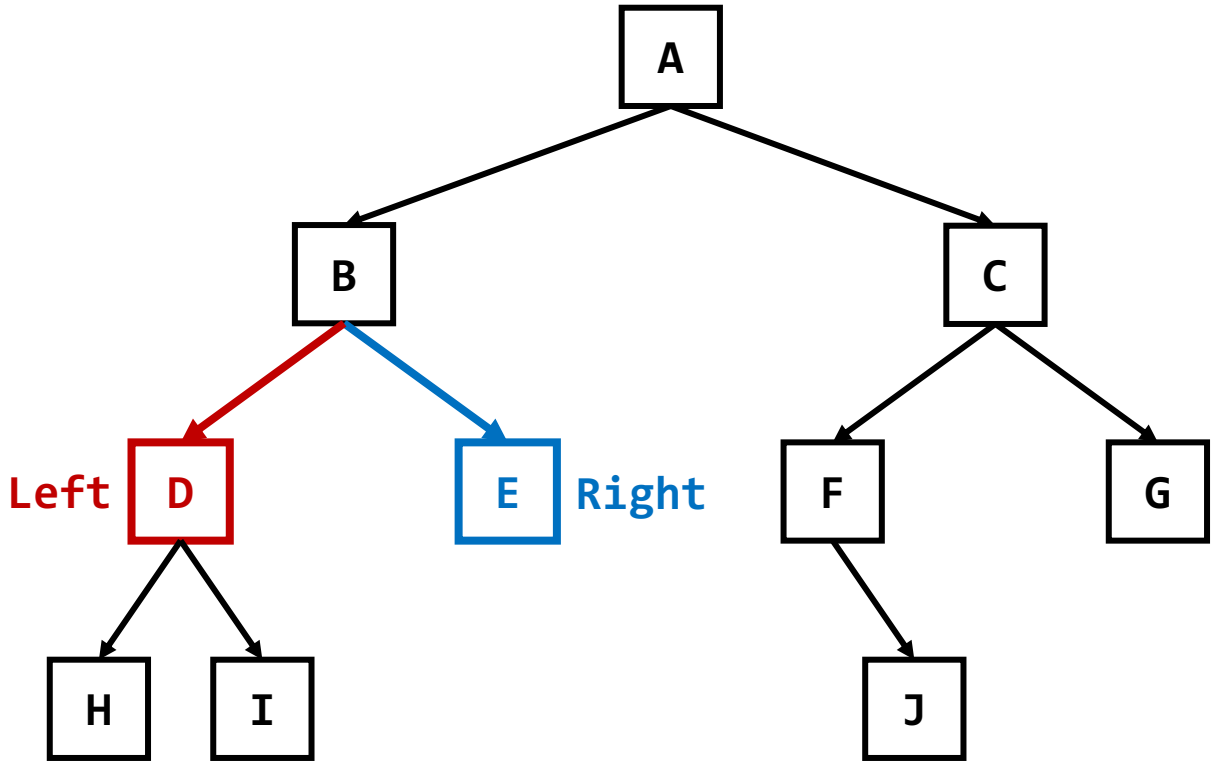
Binary Trees

- **Binary Tree** is a tree in which each node has at most two children
  - `degree(X)` ≤ 2 for any node **X** in a binary tree
  - Each node has **`left`** & **`right`** children



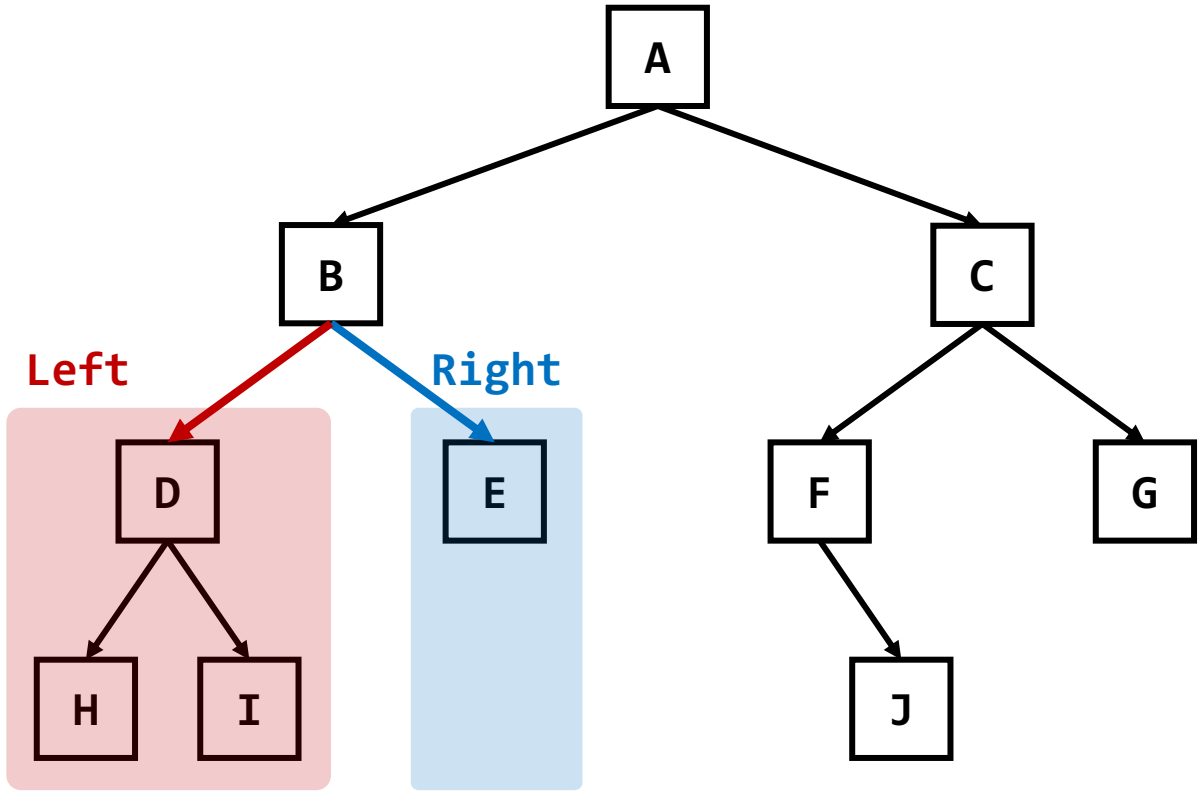
Binary Trees

- **Binary Tree** is a tree in which each node has at most two children
  - `degree(X)` ≤ 2 for any node **X** in a binary tree
  - Each node has **`left`** & **`right`** children



Binary Trees

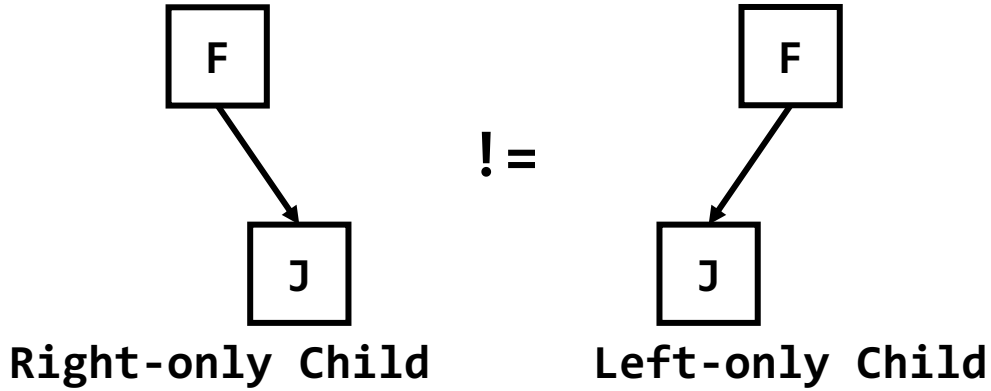
- **Binary Tree** is a tree in which each node has at most two children
  - `degree(X)` ≤ 2 for any node **X** in a binary tree
  - Each node has **`left`** & **`right`** children



Binary Trees

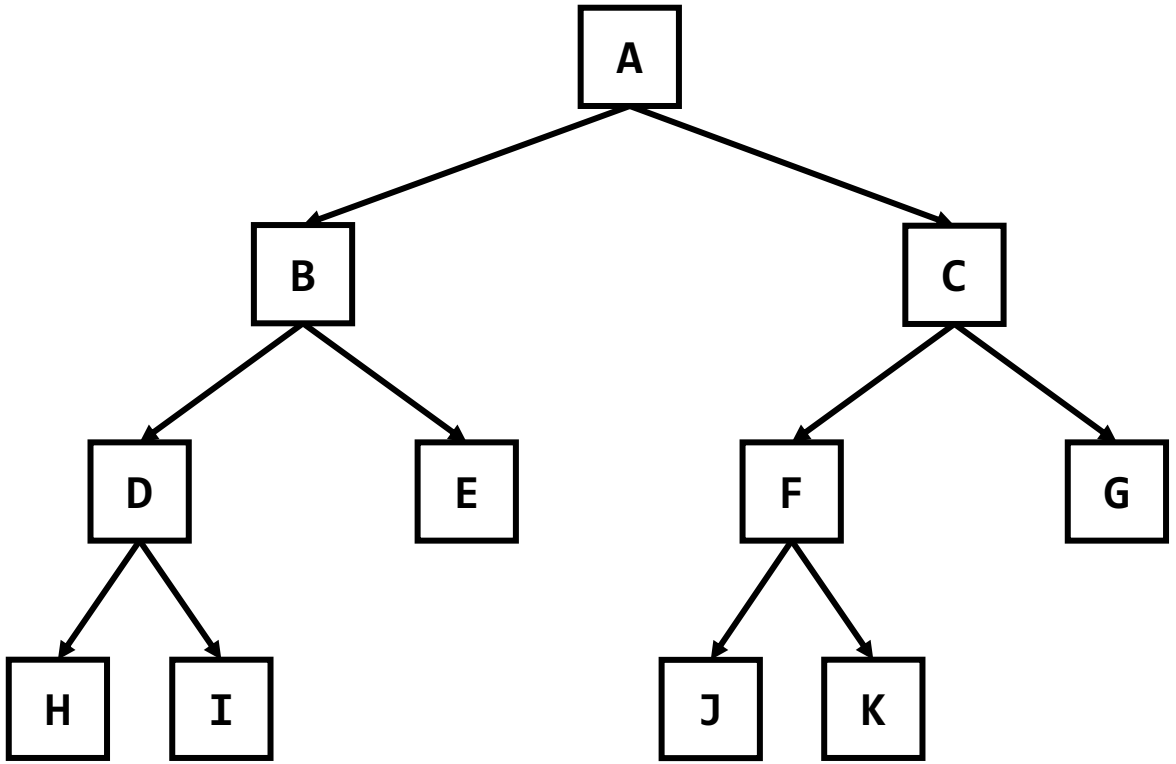
- **Binary Tree** is a tree in which each node has at most two children
  - `degree(X)` ≤ 2 for any node **`X`** in a binary tree
  - Each node has **`left`** & **`right`** children

  - In the binary tree structure, left and right directions are often considered differently



Binary Trees

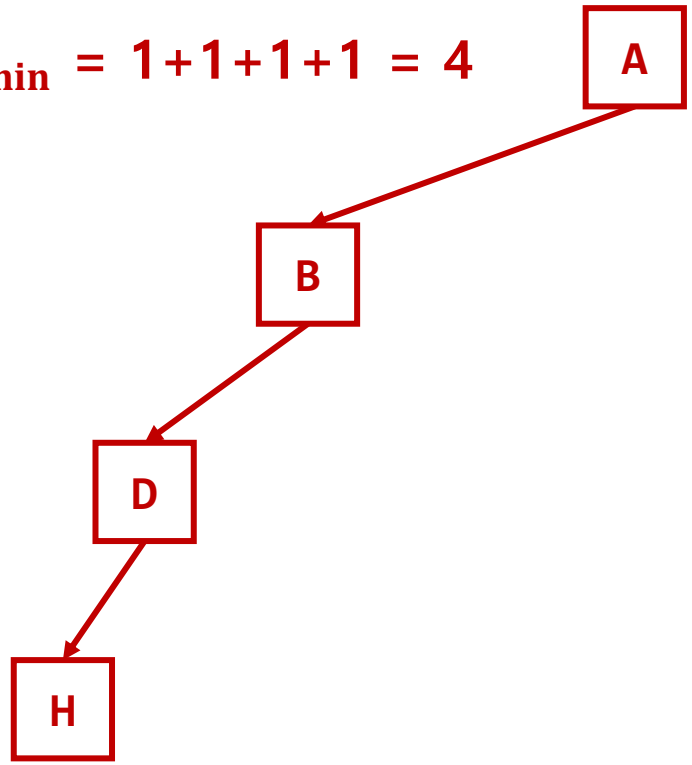
- **Extended Binary Tree** is a BT in which each node has zero or two children
  - `degree(X)` = `0` or `2` for any node **`X`** in an extended binary tree
  - Any binary tree can be easily extended by adding **`leaf`** nodes



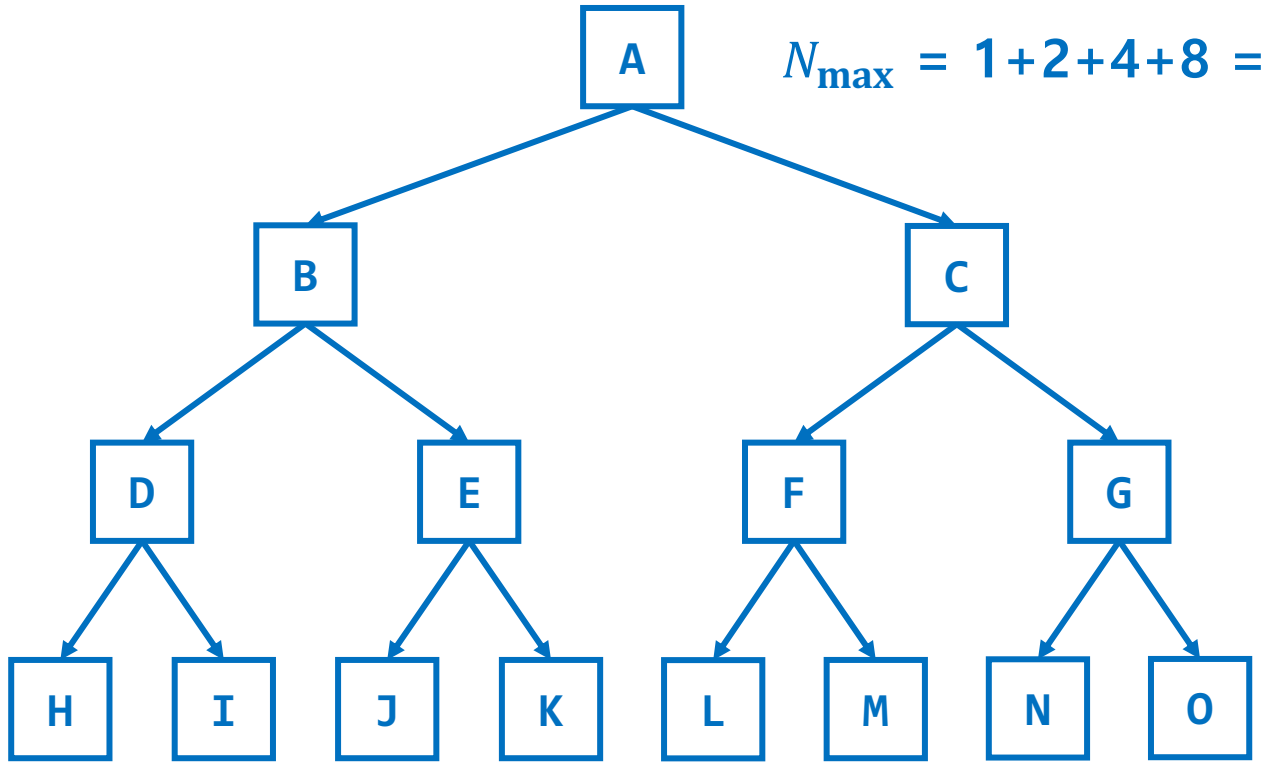
Properties of Binary Trees

- Let H be the **`height`** of a binary tree T
  - The `minimum` number of nodes in the binary tree is H
  - The `maximum` number of nodes in the binary tree is $2^H - 1$

N_{min} = 1+1+1+1 = 4

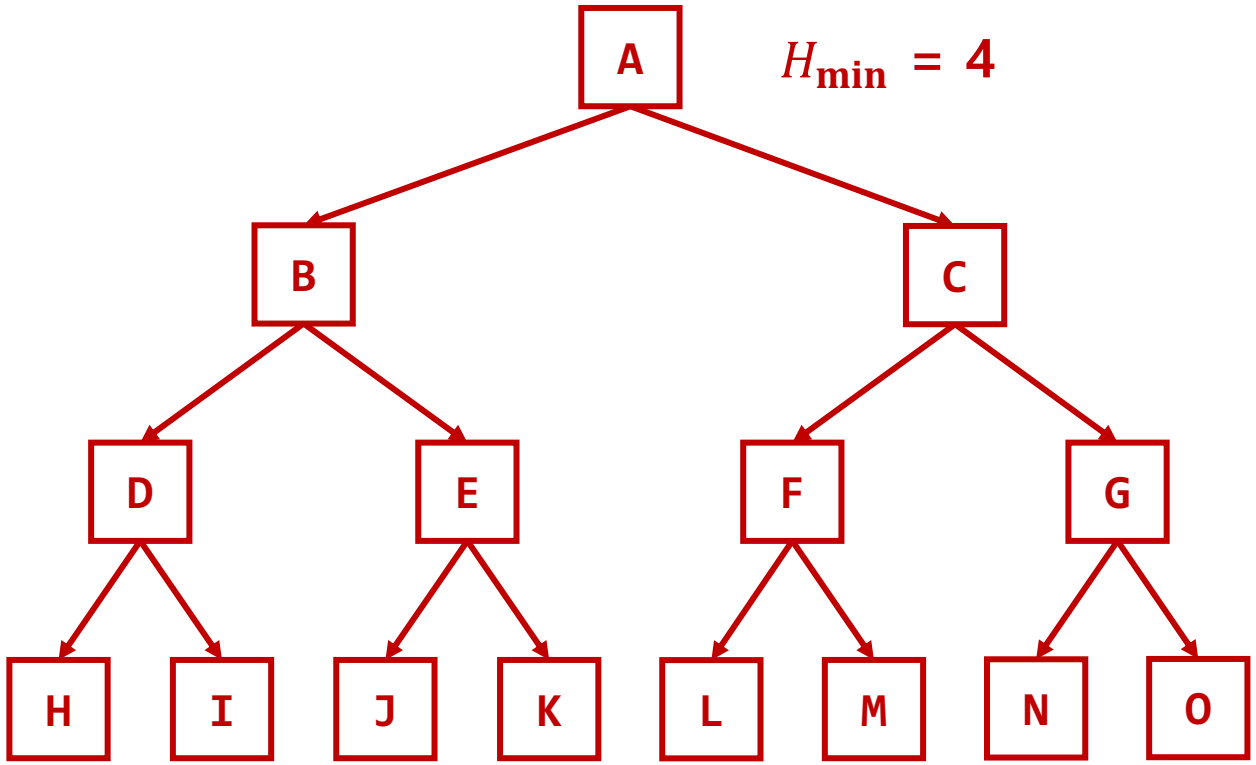


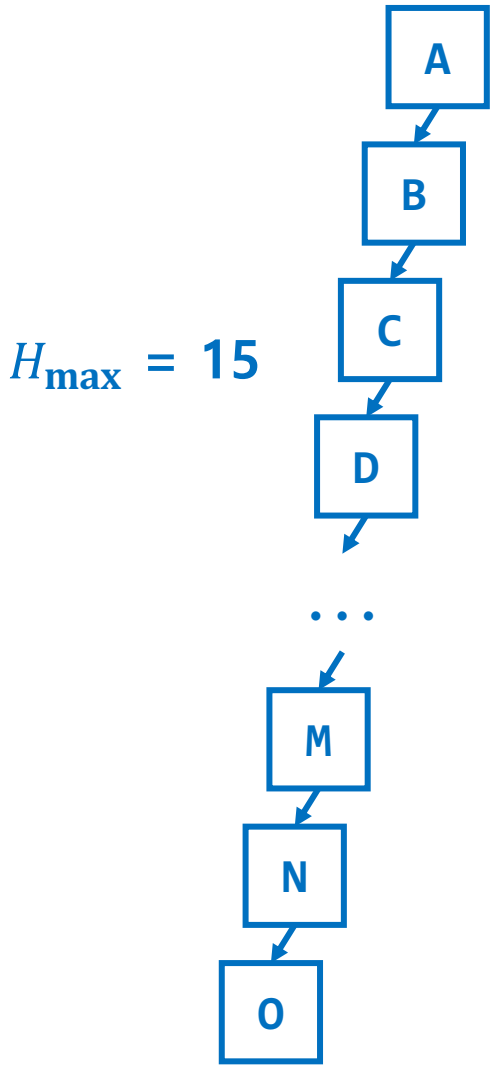
N_{max} = 1+2+4+8 = 15



Properties of Binary Trees

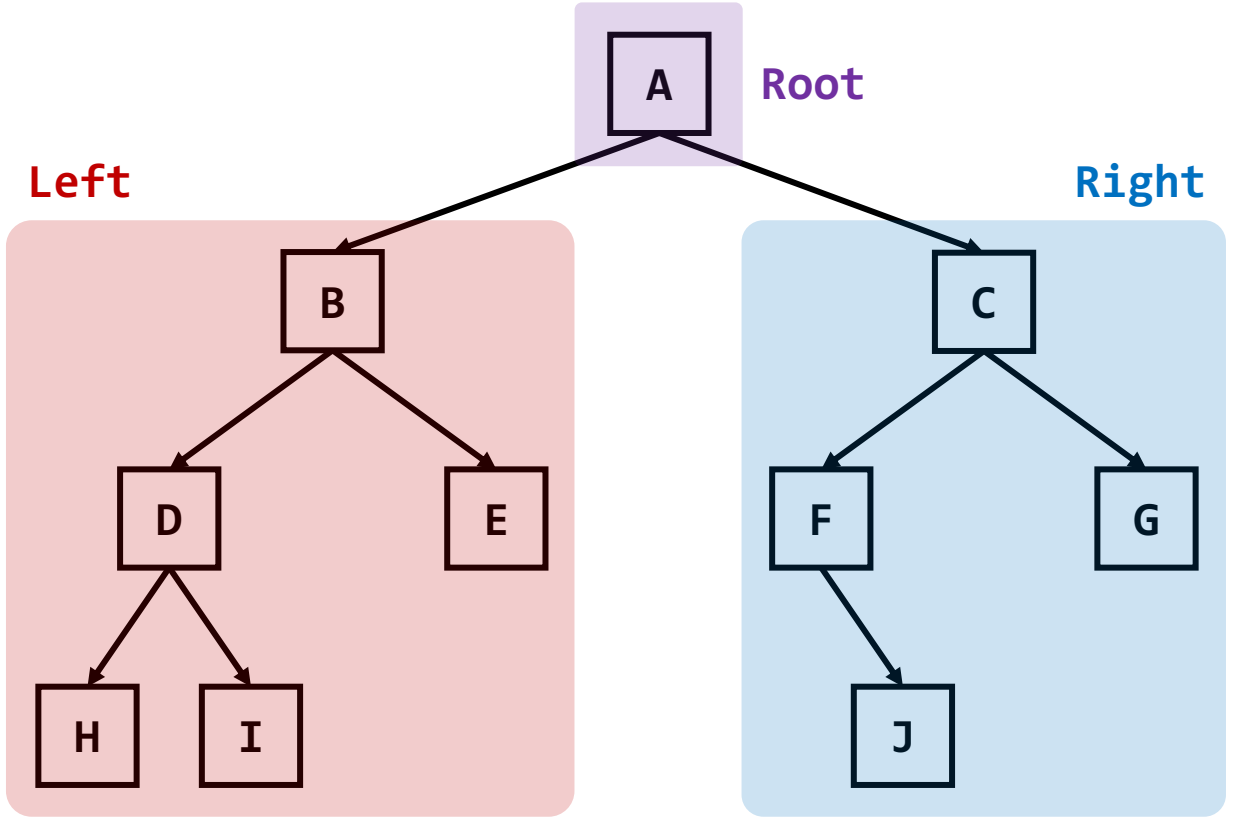
- Let N be the **number** of nodes in a binary tree T
  - The `minimum` height of the binary tree is $\lceil \log_2(N+1) \rceil$
  - The `maximum` height of the binary tree is N





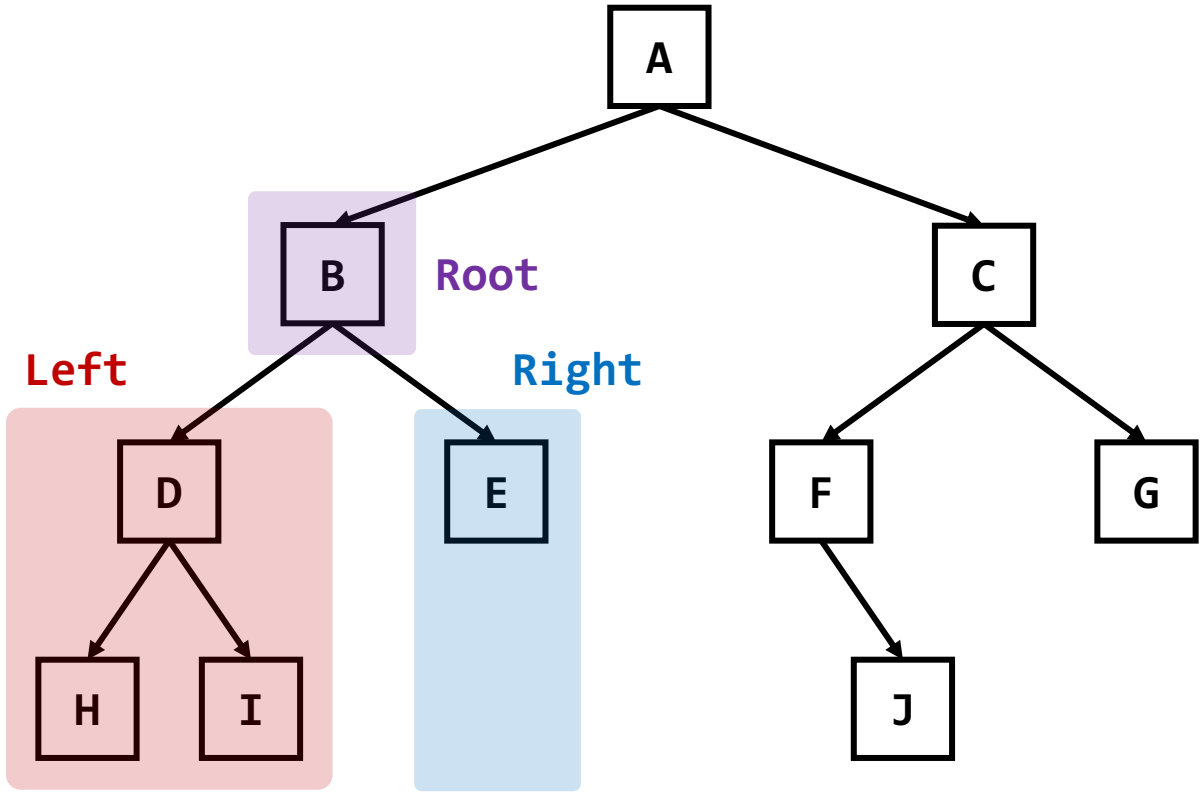
Binary Tree Traversal

- How to **traverse** all nodes in a binary tree?
  - In-order traversal : `Left Subtree` → `Root` → `Right Subtree`



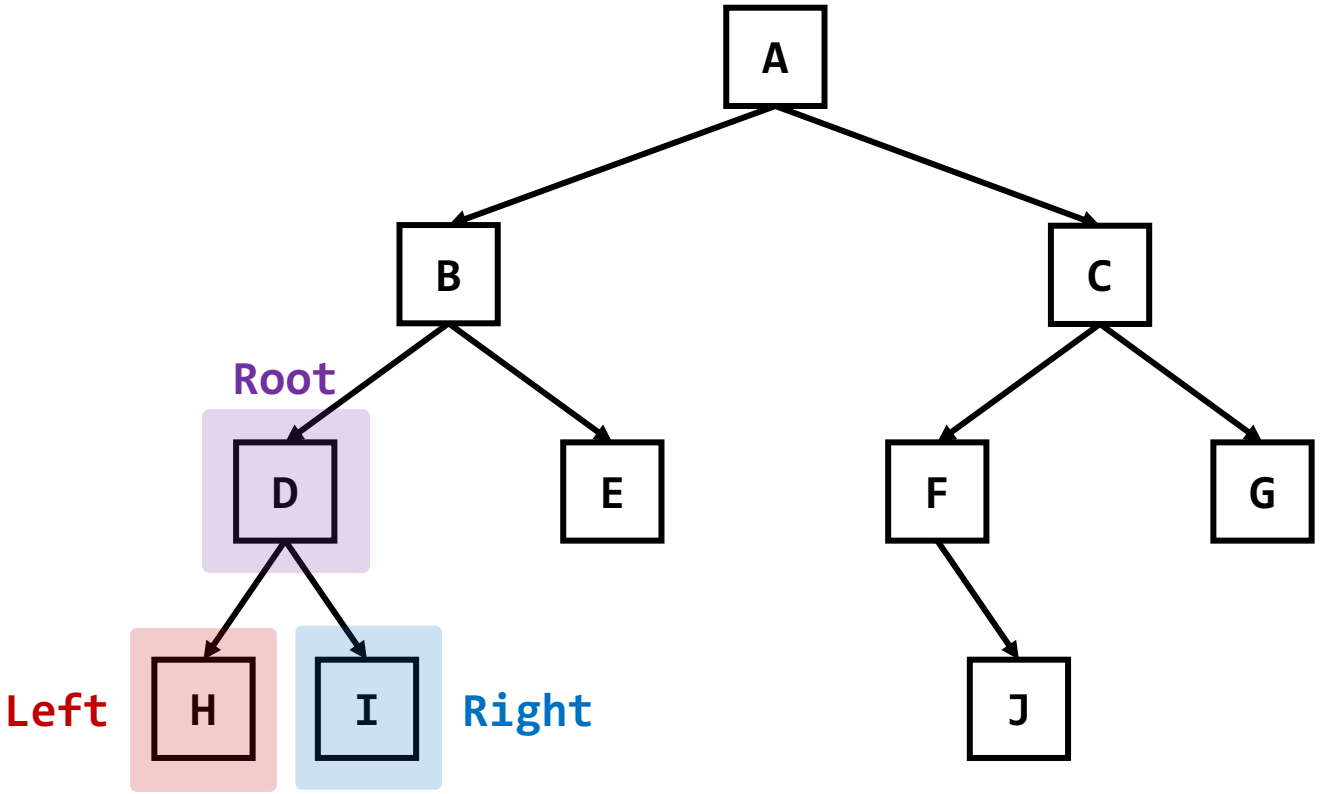
Binary Tree Traversal

- How to **traverse** all nodes in a binary tree?
  - In-order traversal : `Left Subtree` → `Root` → `Right Subtree`



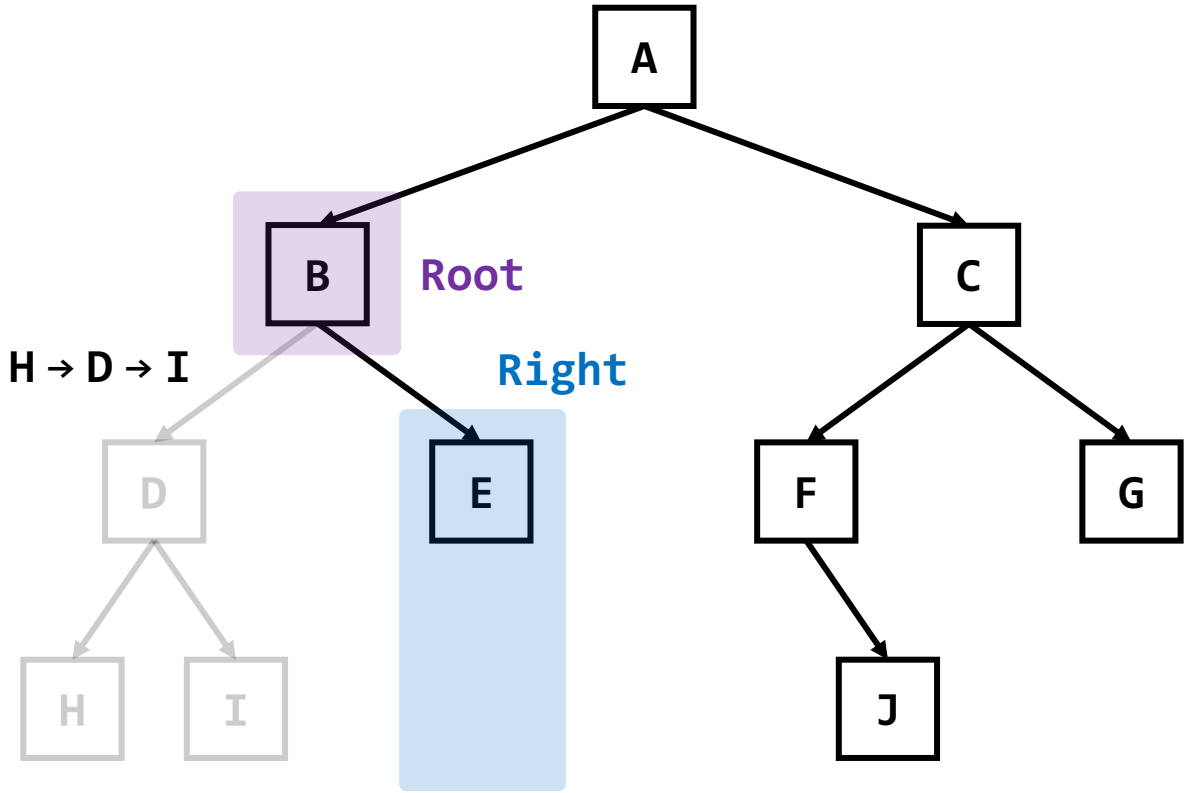
Binary Tree Traversal

- How to **traverse** all nodes in a binary tree?
  - In-order traversal : **`Left Subtree`** → **`Root`** → **`Right Subtree`**



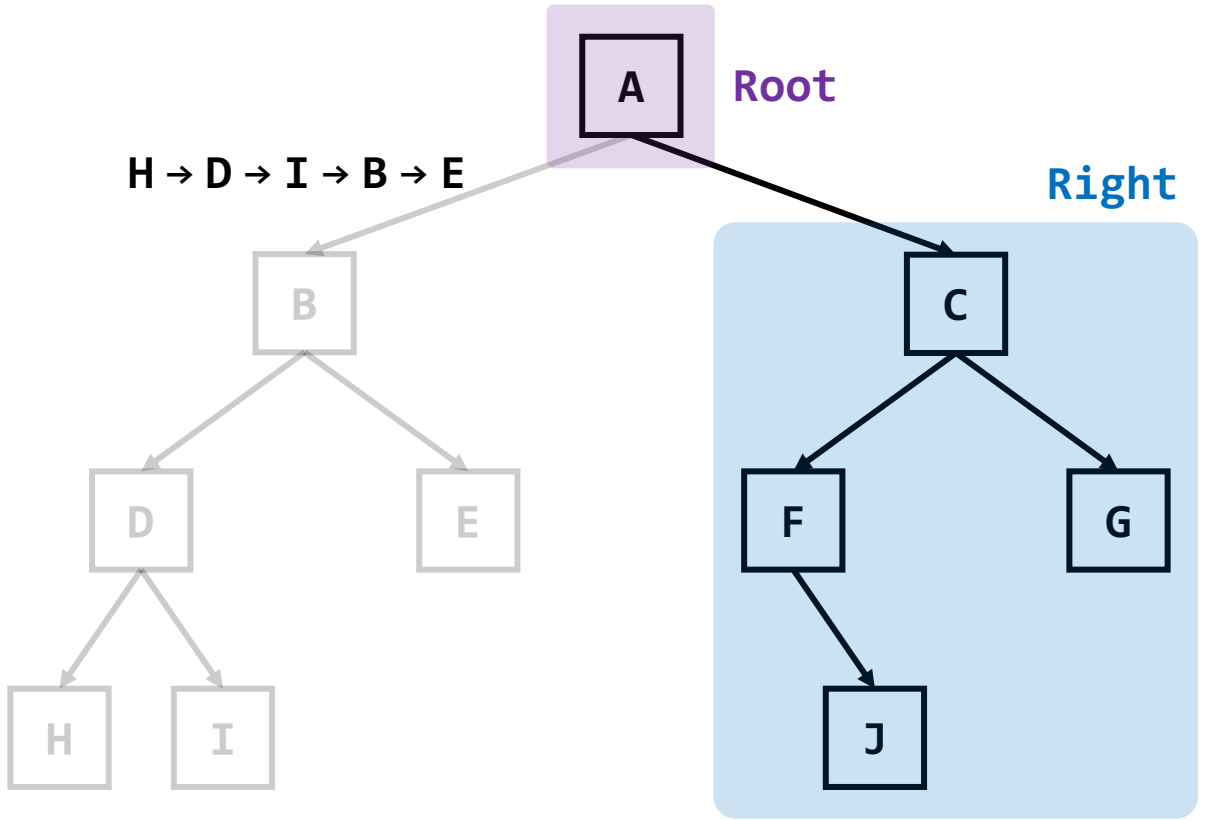
Binary Tree Traversal

- How to **traverse** all nodes in a binary tree?
  - In-order traversal : `Left Subtree` → `Root` → `Right Subtree`



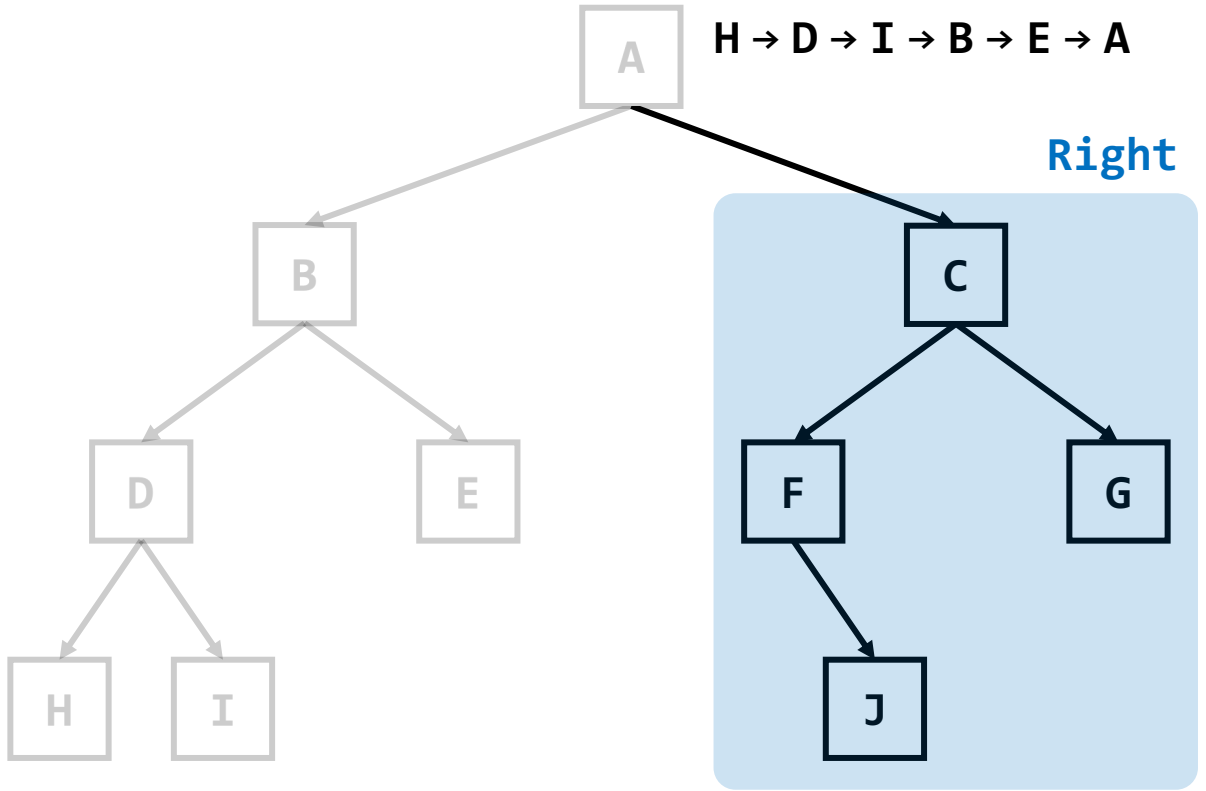
Binary Tree Traversal

- How to **traverse** all nodes in a binary tree?
  - In-order traversal : **`Left Subtree → Root → Right Subtree`**



Binary Tree Traversal

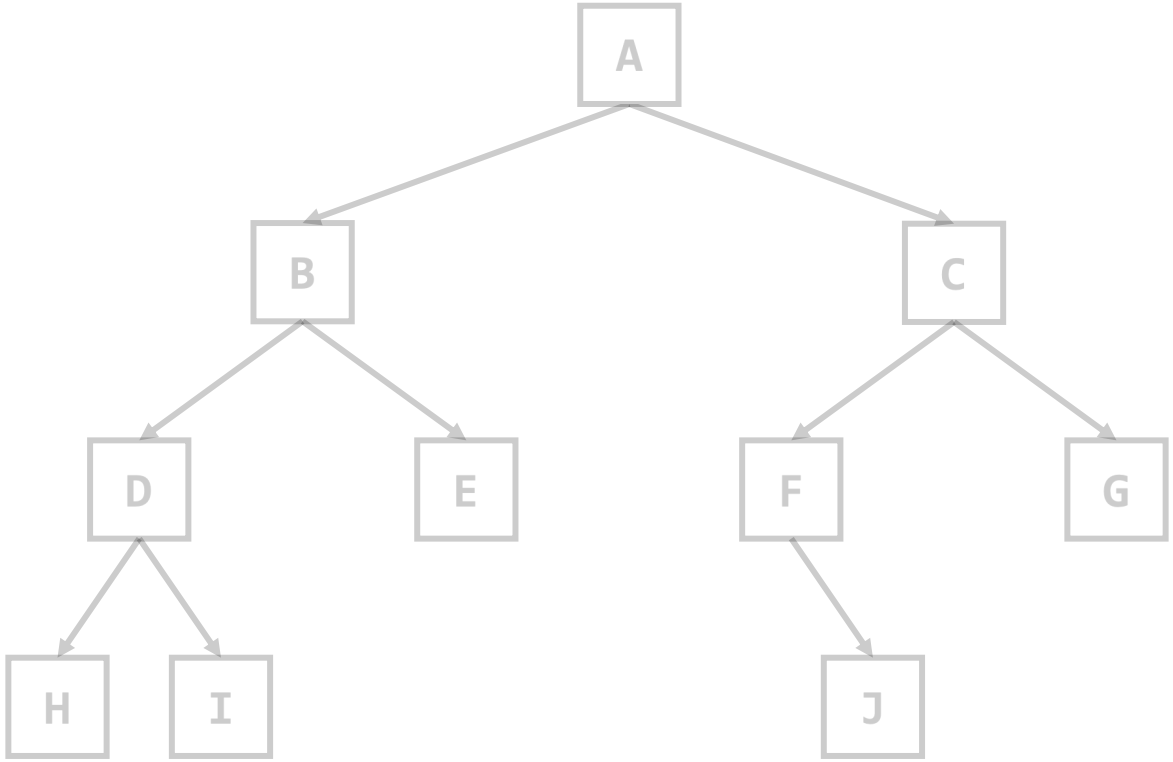
- How to **traverse** all nodes in a binary tree?
  - In-order traversal : **`Left Subtree`** → **`Root`** → **`Right Subtree`**



Binary Tree Traversal

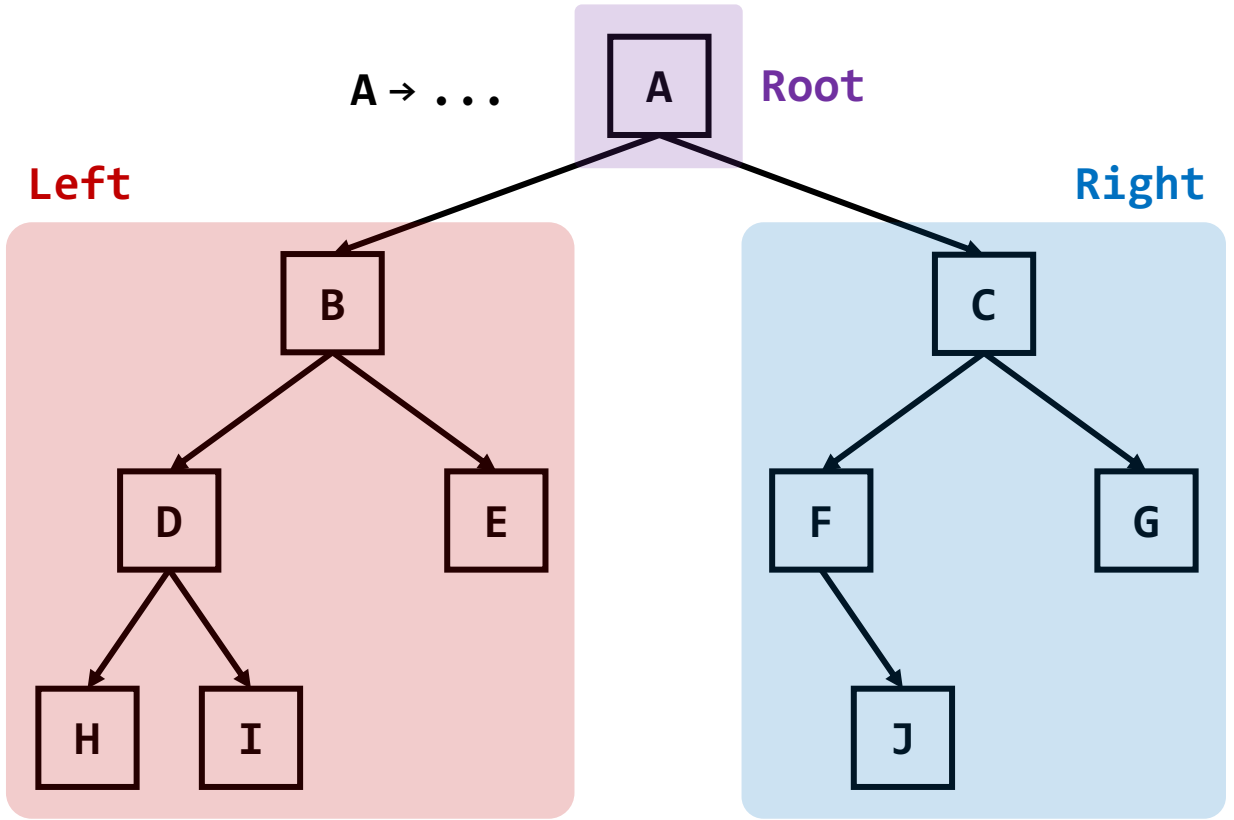
- How to **traverse** all nodes in a binary tree?
  - In-order traversal : **`Left Subtree → Root → Right Subtree`**

H → D → I → B → E → A → F → J → C → G



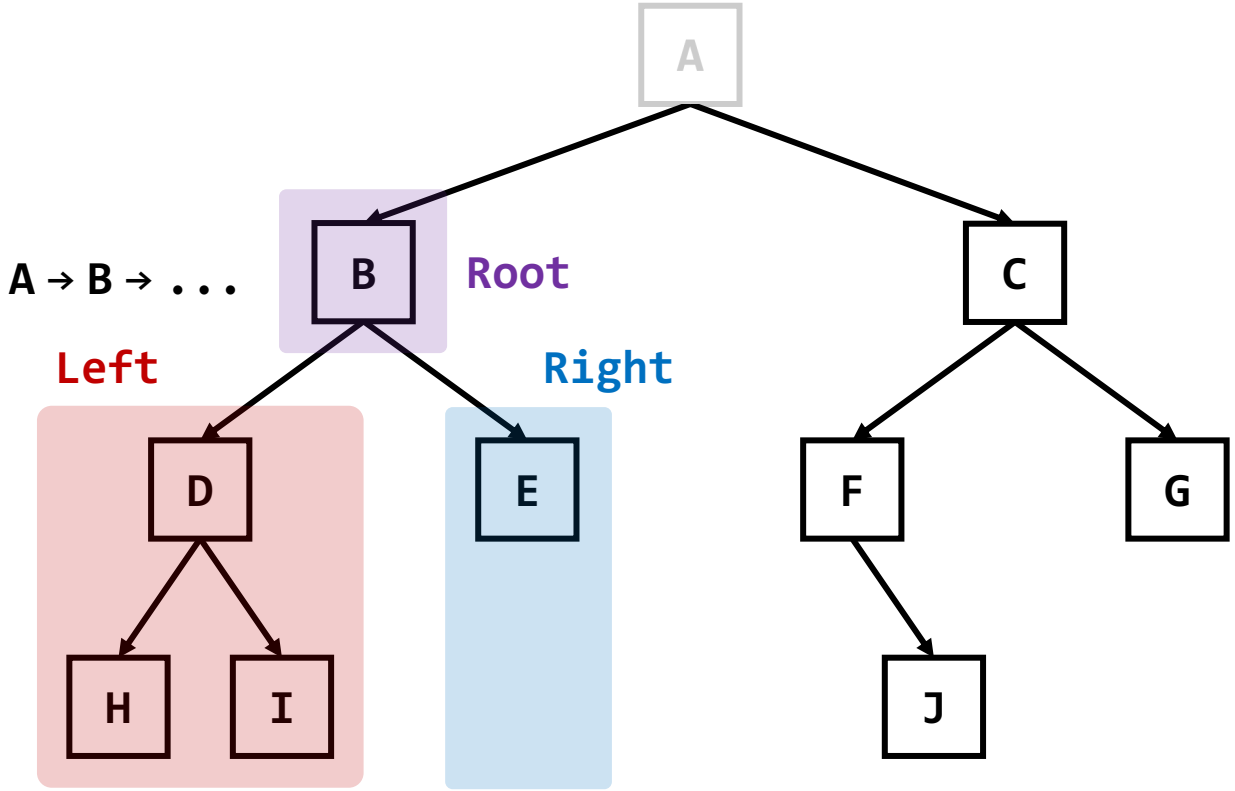
Binary Tree Traversal

- How to **traverse** all nodes in a binary tree?
  - Pre-order traversal : `Root` → `Left Subtree` → `Right Subtree`



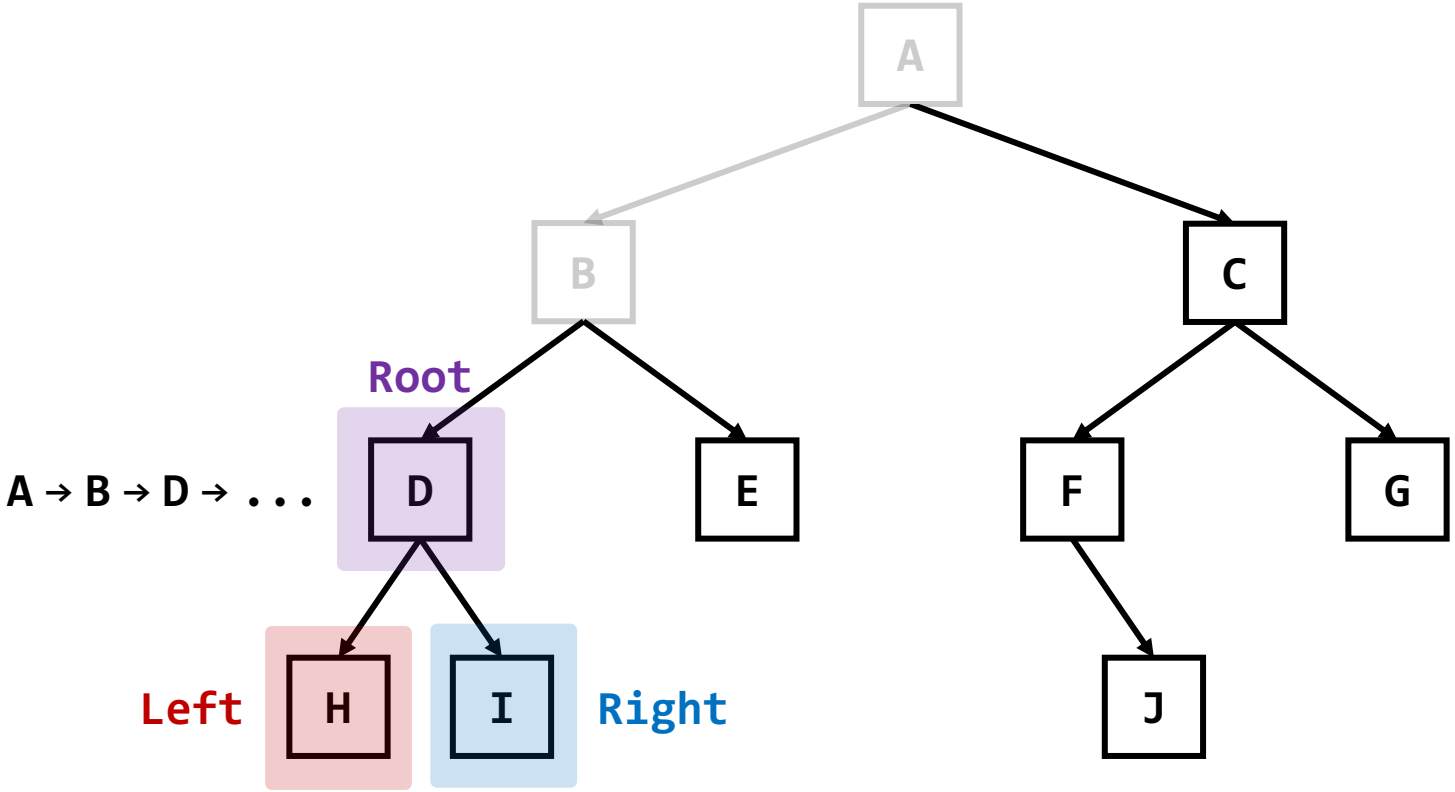
Binary Tree Traversal

- How to **traverse** all nodes in a binary tree?
  - Pre-order traversal : `Root` → `Left Subtree` → `Right Subtree`

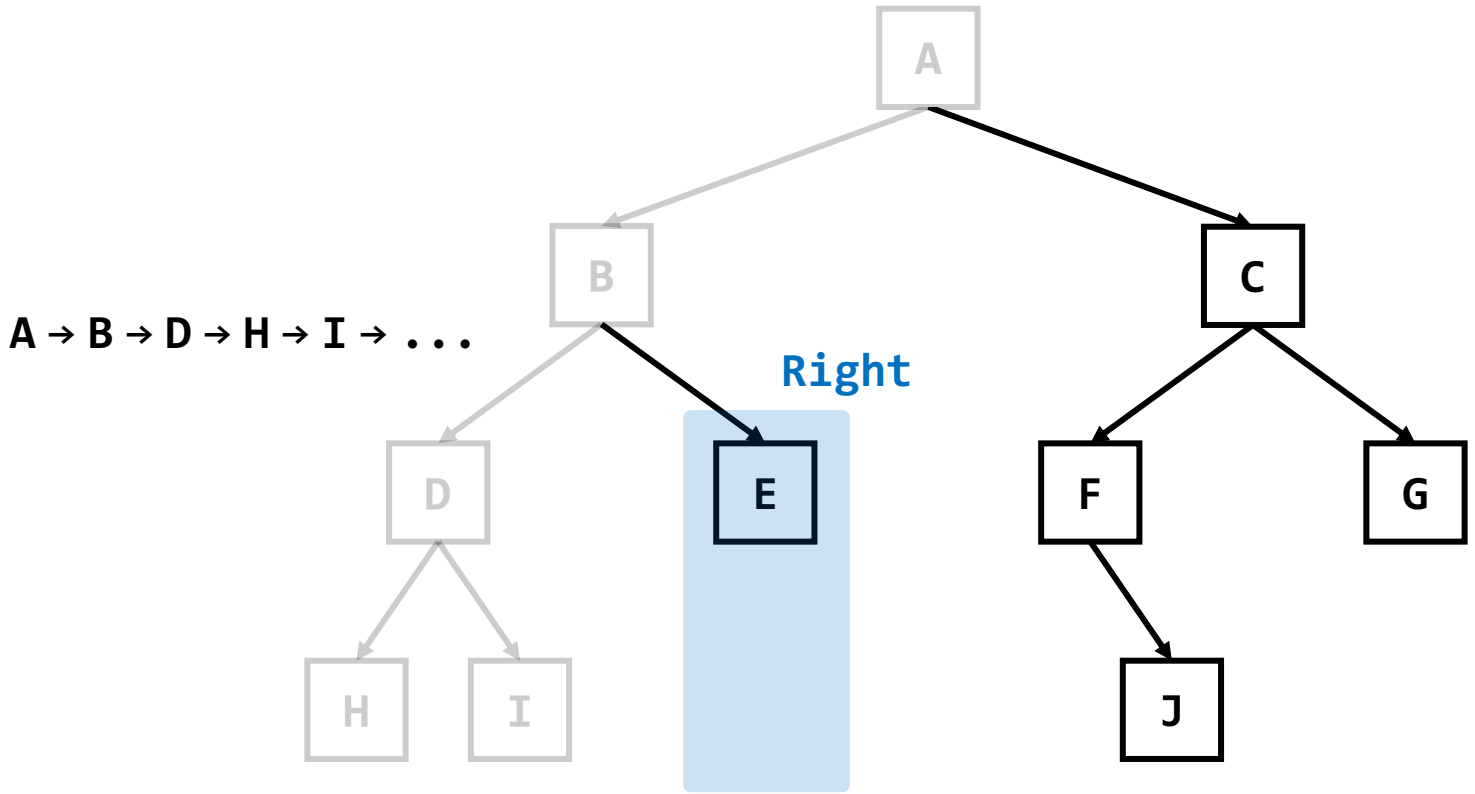


Binary Tree Traversal

- How to **traverse** all nodes in a binary tree?
  - Pre-order traversal : **Root** → **Left Subtree** → **Right Subtree**



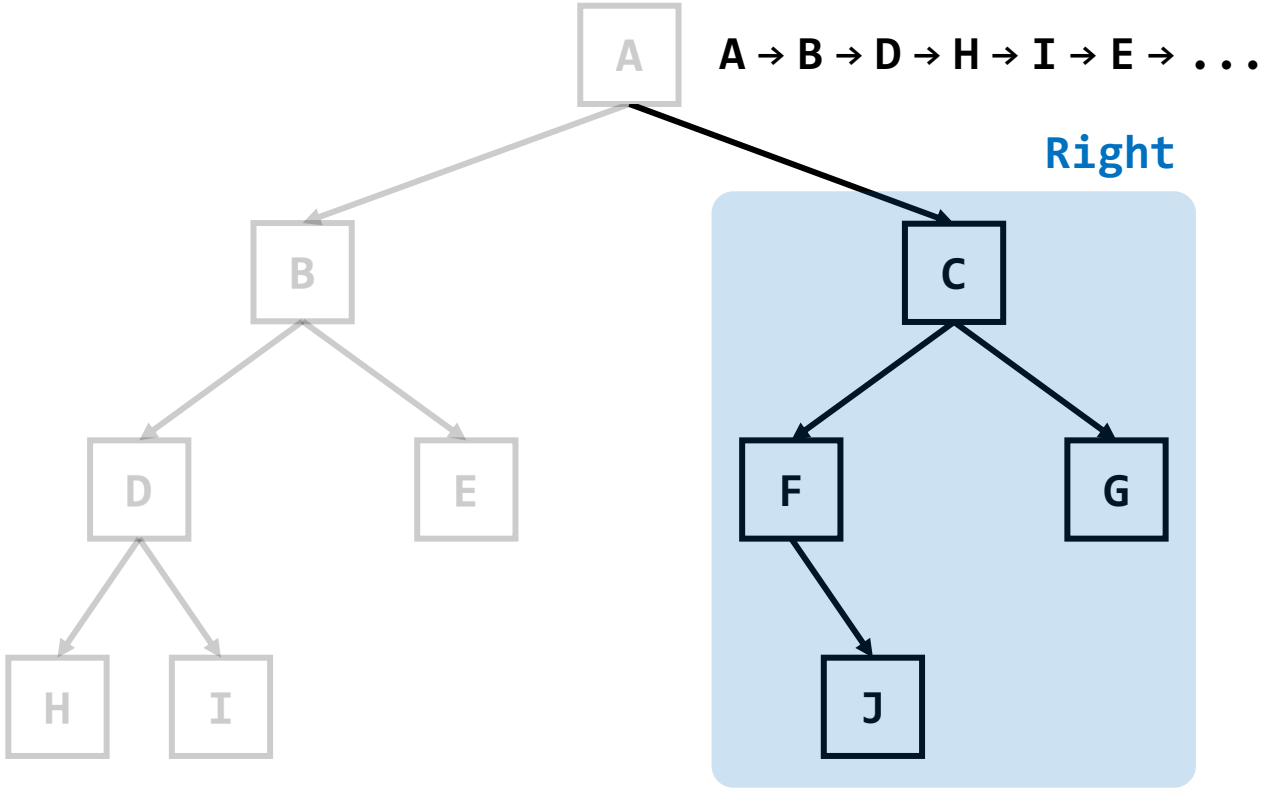
Binary Tree Traversal



- How to **traverse** all nodes in a binary tree?
  - Pre-order traversal : `Root` → `Left Subtree` → `Right Subtree`

Binary Tree Traversal

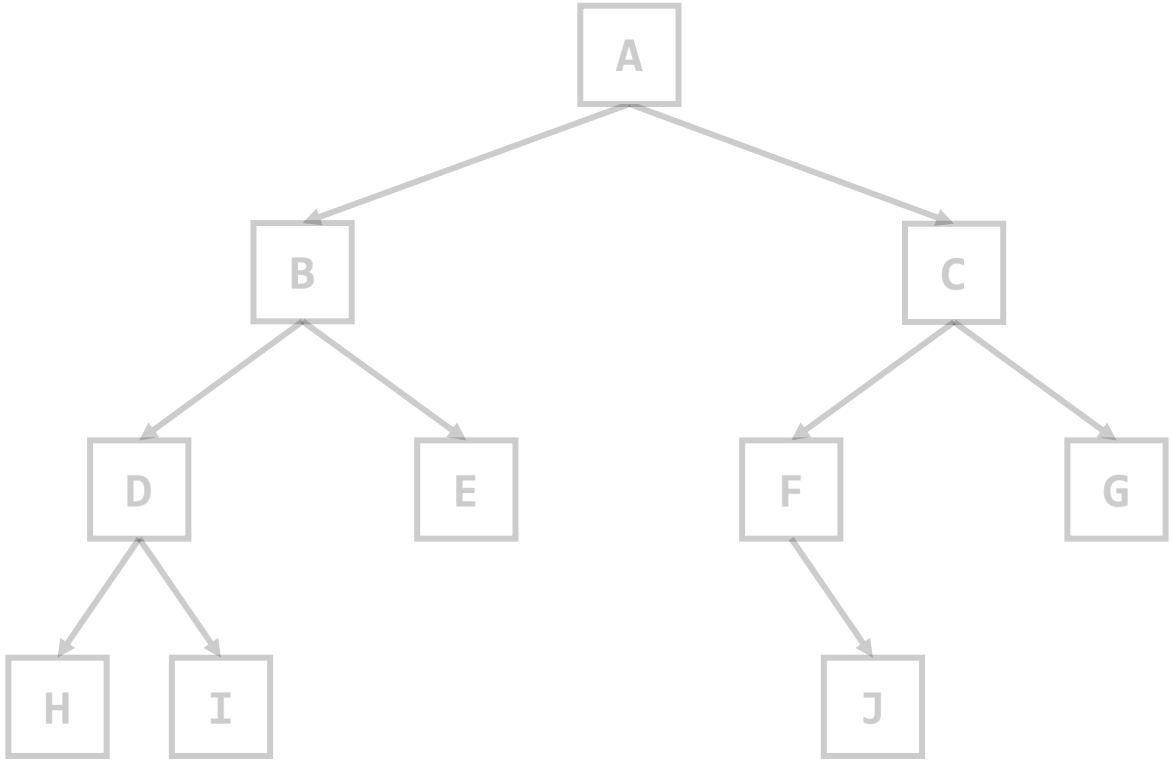
- How to **traverse** all nodes in a binary tree?
  - Pre-order traversal : `Root` → `Left Subtree` → `Right Subtree`



Binary Tree Traversal

- How to **traverse** all nodes in a binary tree?
  - Pre-order traversal : `Root` → `Left Subtree` → `Right Subtree`

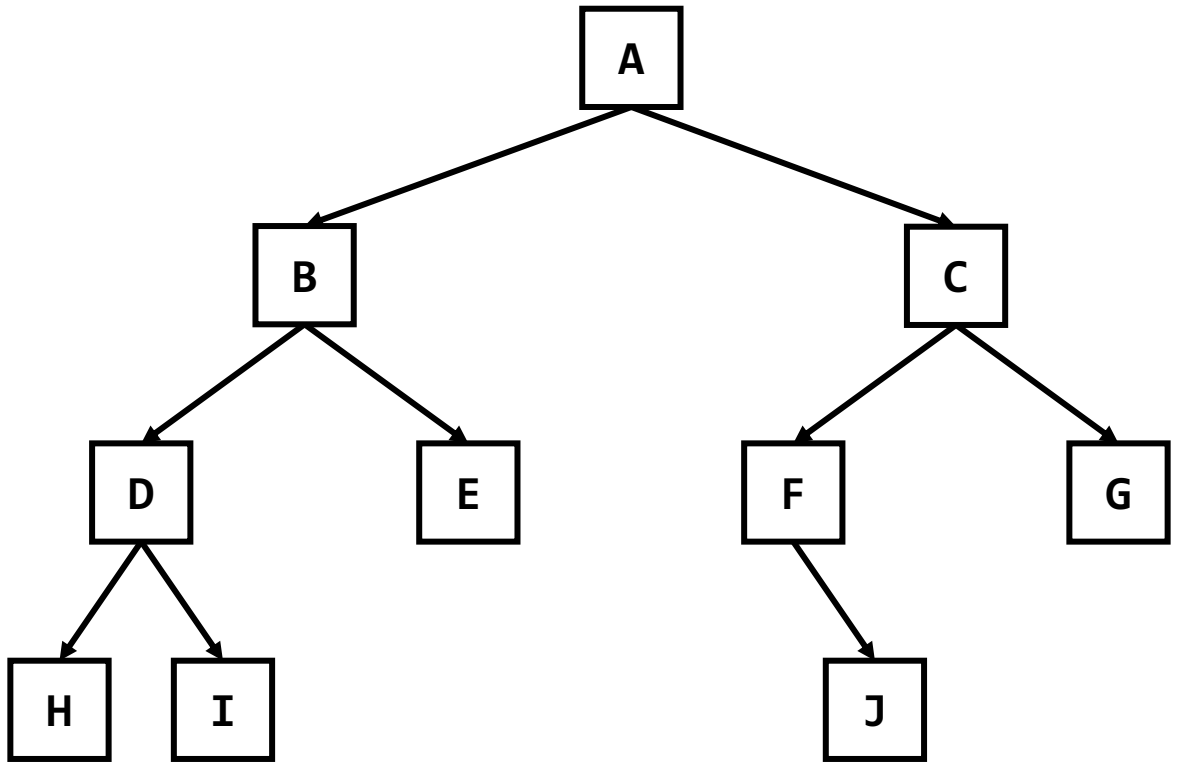
A → B → D → H → I → E → C → F → J → G



Binary Tree Traversal

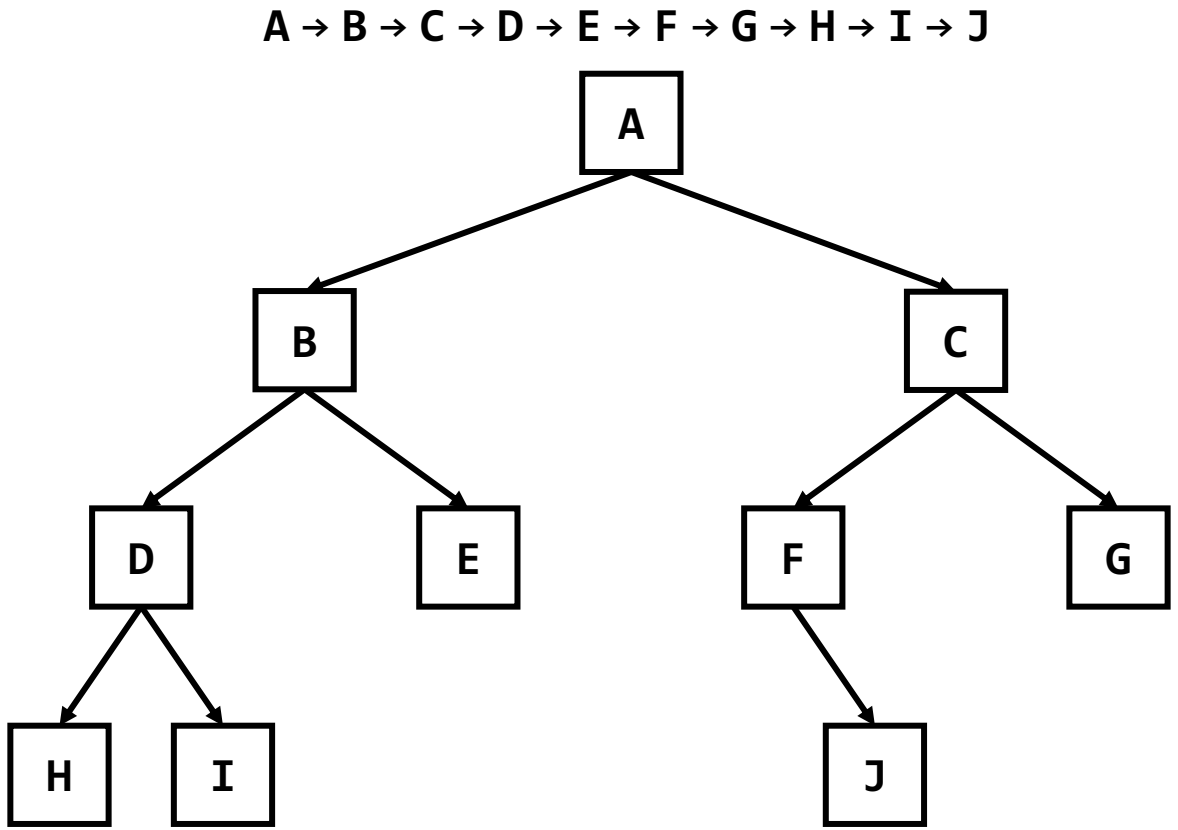
- How to **traverse** all nodes in a binary tree?
  - Post-order traversal : **`Left Subtree → Right Subtree → Root`**

H → I → D → E → B → J → F → G → C → A

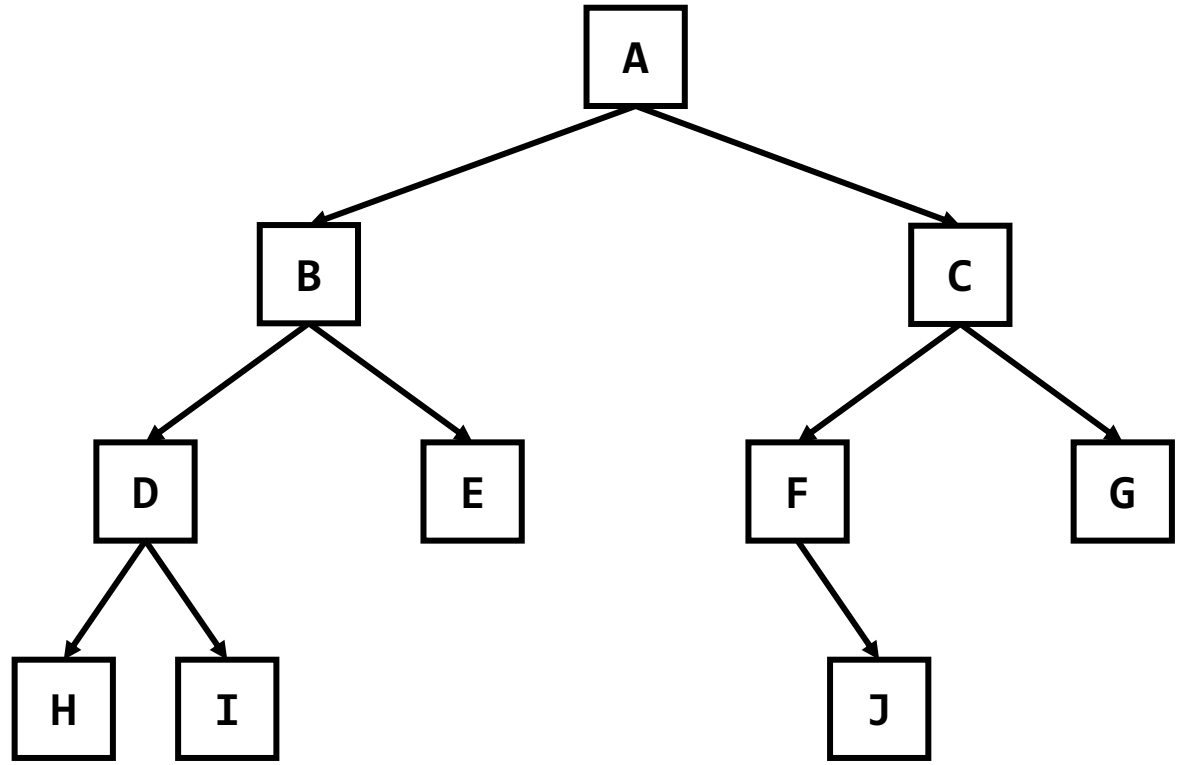


Binary Tree Traversal

- How to **traverse** all nodes in a binary tree?
  - Level-order traversal : from top (`level=0`) to bottom (`level=height-1`)

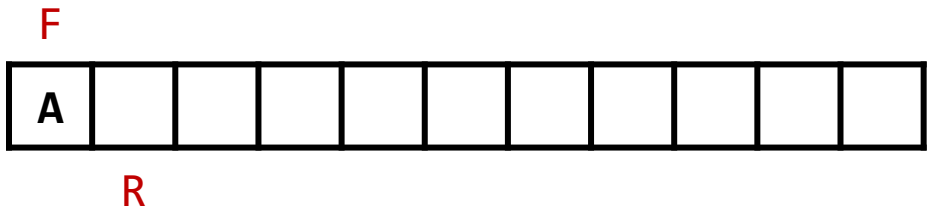


Binary Tree Traversal



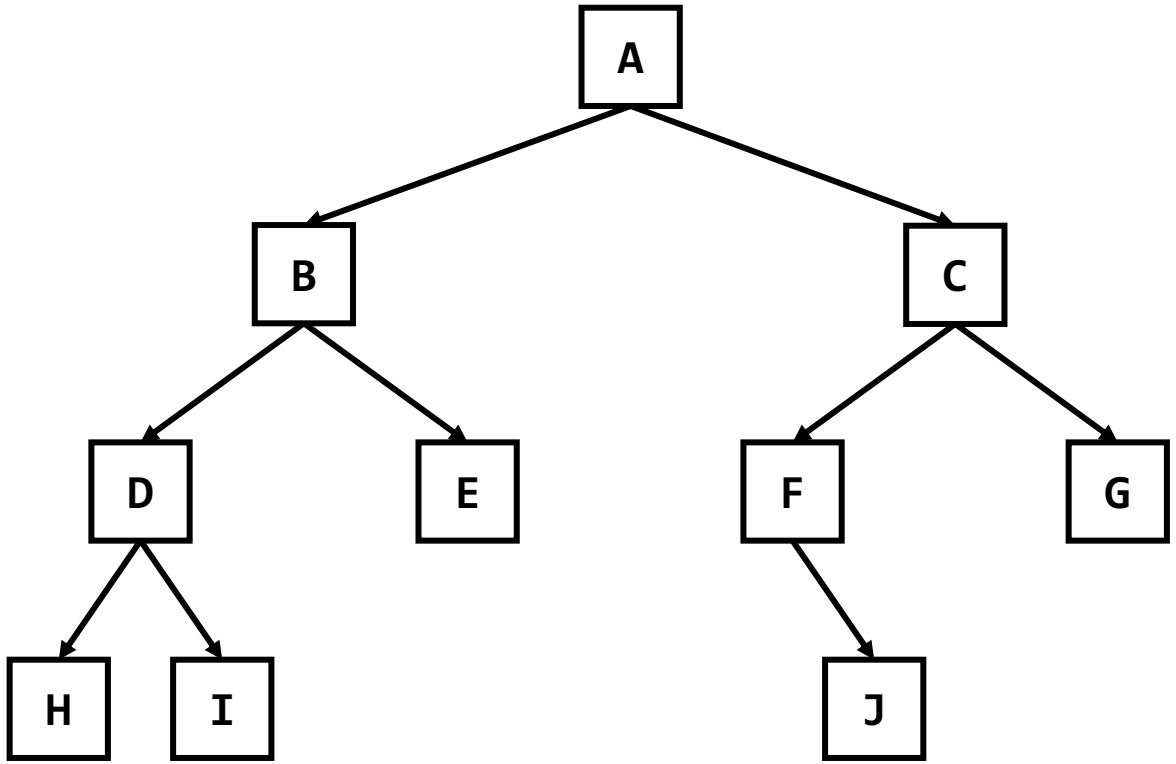
Queue-based level-order traversal

1. Print the value of the front
2. Enqueue two children of the front
3. Dequeue the front



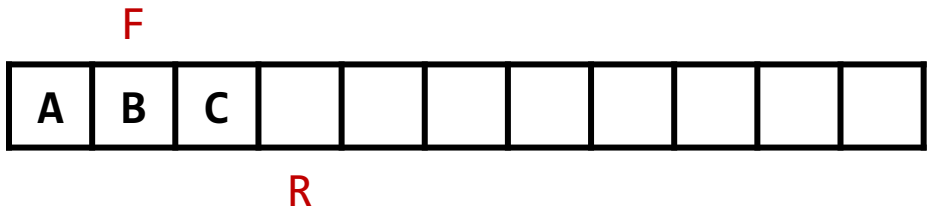
Binary Tree Traversal

- How to **traverse** all nodes in a binary tree?
  - Level-order traversal : from top (`level=0`) to bottom (`level=height-1`)



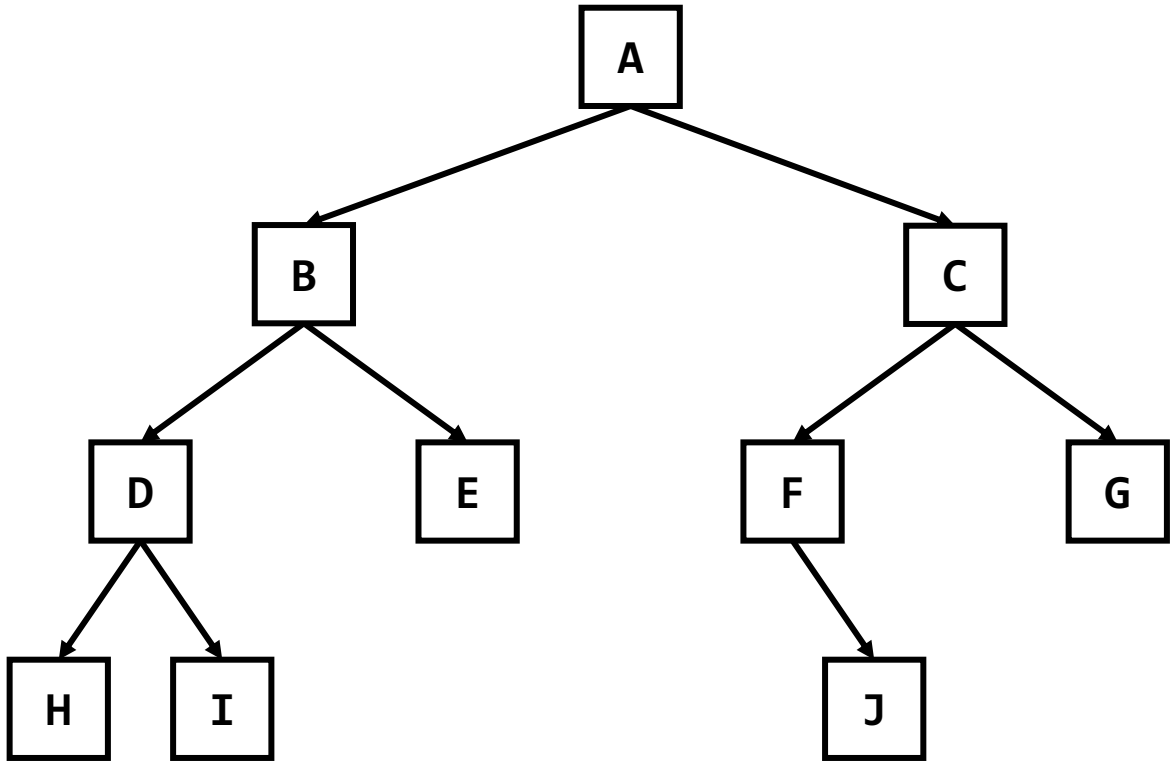
Queue-based level-order traversal

1. Print the value of the front
2. Enqueue two children of the front
3. Dequeue the front



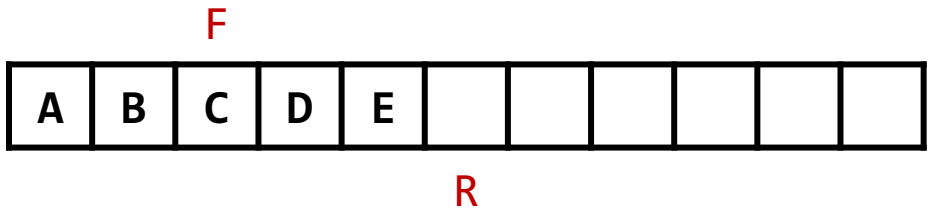
Binary Tree Traversal

- How to **traverse** all nodes in a binary tree?
  - Level-order traversal : from top (`level=0`) to bottom (`level=height-1`)



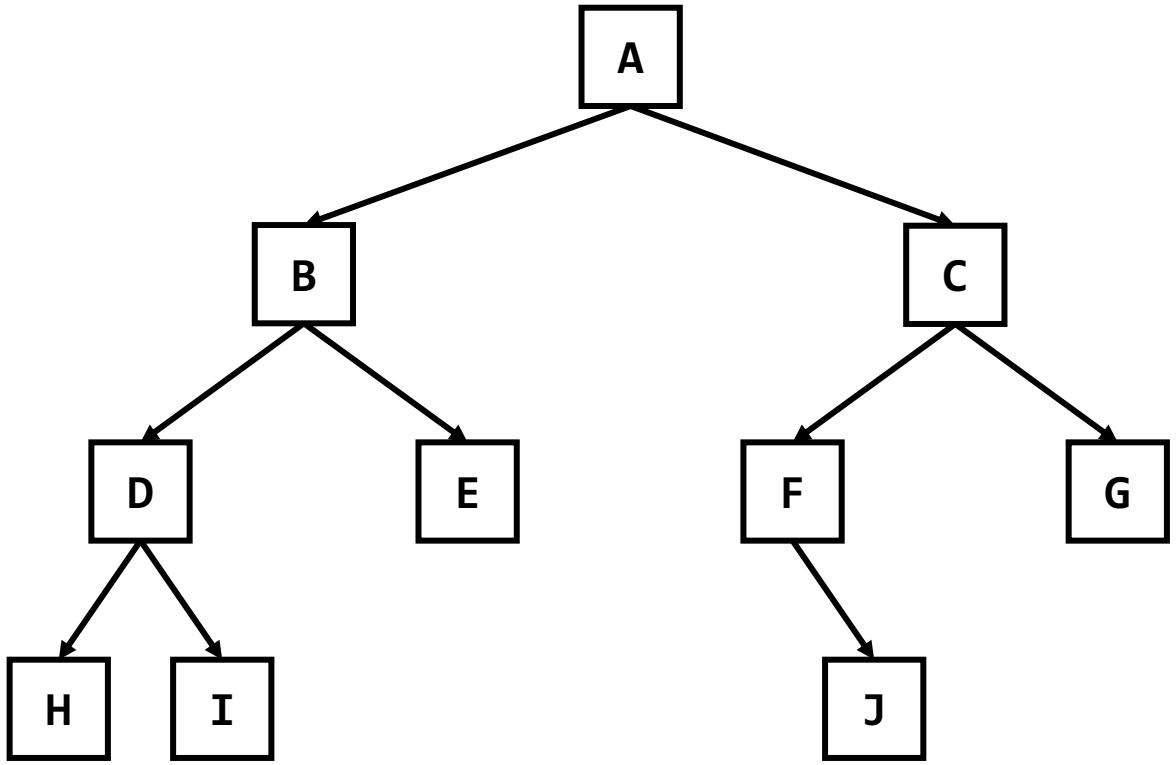
Queue-based level-order traversal

1. Print the value of the front
2. Enqueue two children of the front
3. Dequeue the front



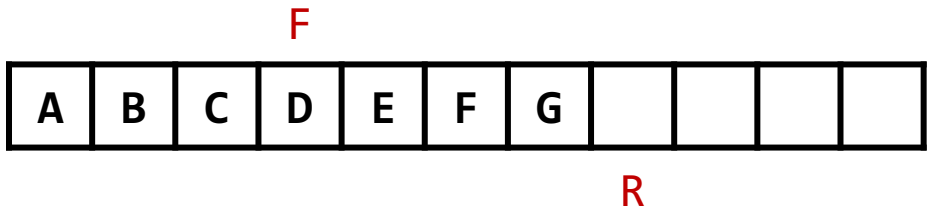
Binary Tree Traversal

- How to **traverse** all nodes in a binary tree?
  - Level-order traversal : from top (`level=0`) to bottom (`level=height-1`)

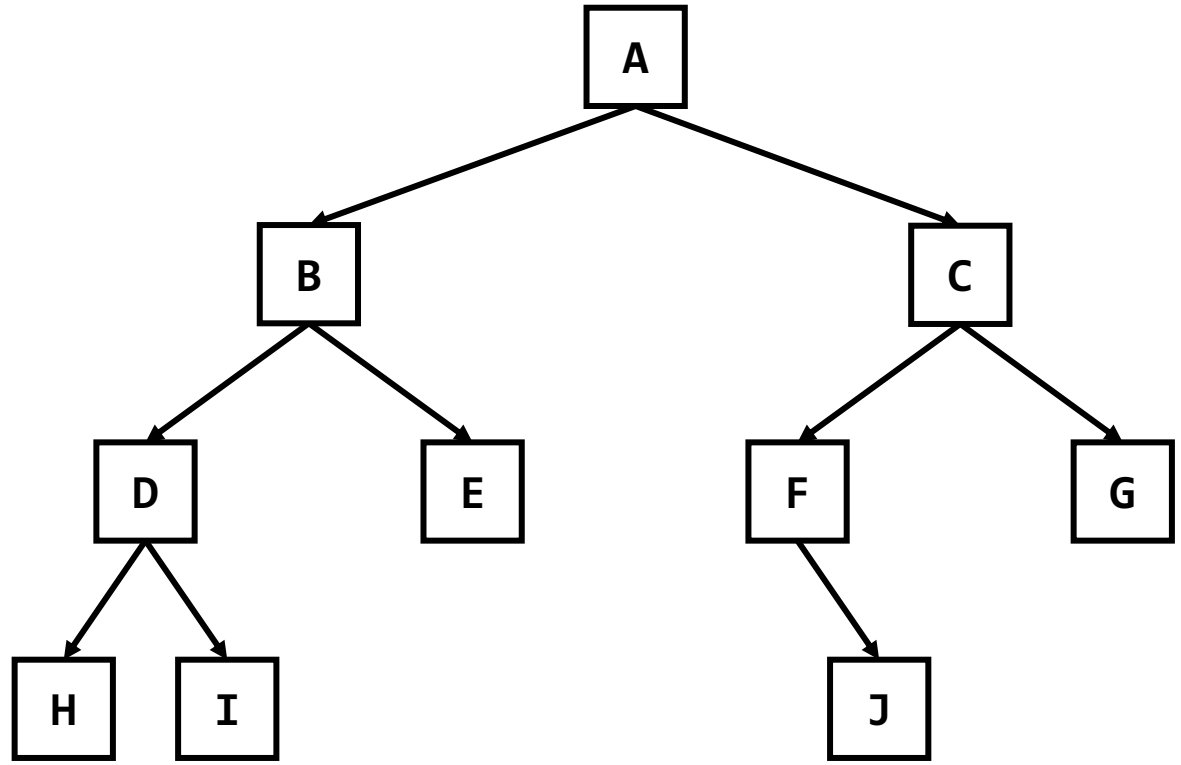


Queue-based level-order traversal

1. Print the value of the front
2. Enqueue two children of the front
3. Dequeue the front

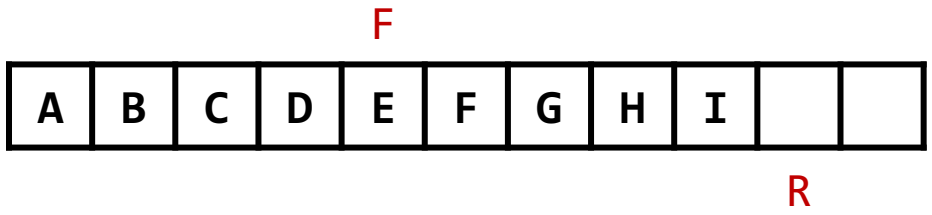


Binary Tree Traversal

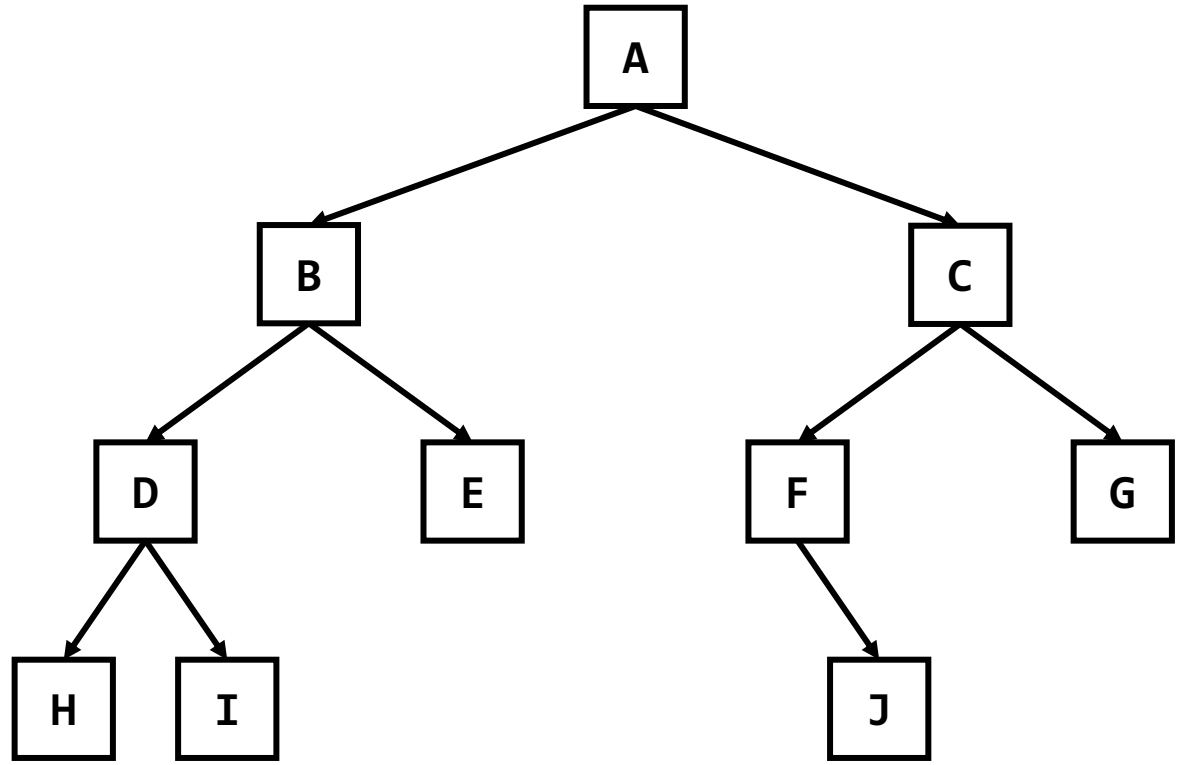


Queue-based level-order traversal

1. Print the value of the front
2. Enqueue two children of the front
3. Dequeue the front

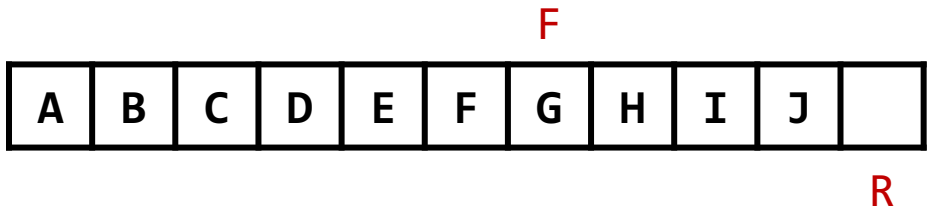


Binary Tree Traversal



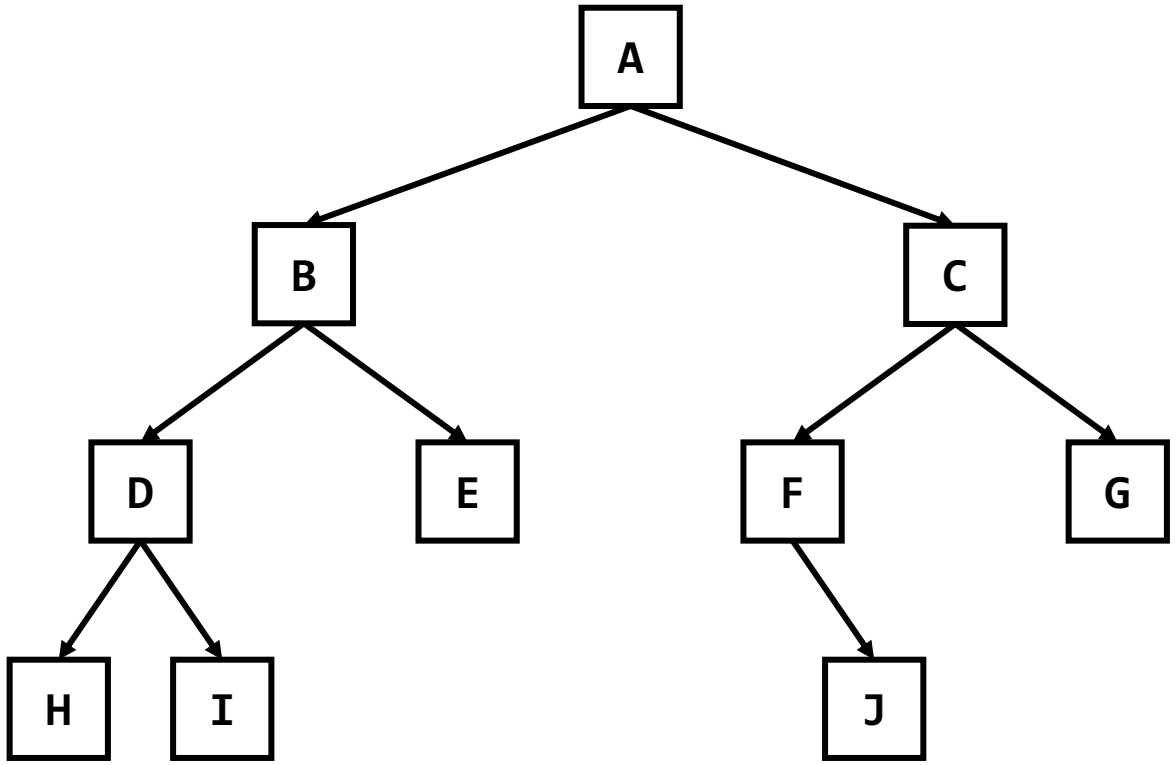
Queue-based level-order traversal

1. Print the value of the front
2. Enqueue two children of the front
3. Dequeue the front



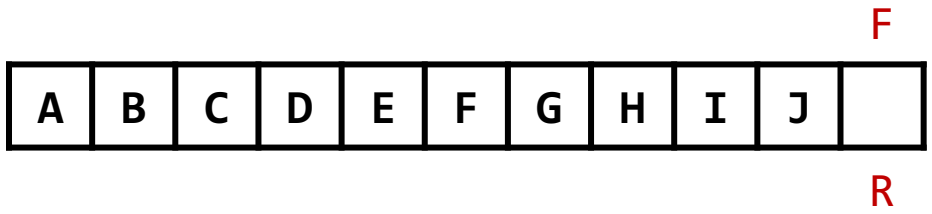
Binary Tree Traversal

- How to **traverse** all nodes in a binary tree?
  - Level-order traversal : from top (`level=0`) to bottom (`level=height-1`)



Queue-based level-order traversal

1. Print the value of the front
2. Enqueue two children of the front
3. Dequeue the front



Binary Tree Traversal

- How to **traverse** all nodes in a binary tree?

  Depth-First Search (DFS)

  - In-order traversal : `Left Subtree → Root → Right Subtree`
  - Pre-order traversal : `Root → Left Subtree → Right Subtree`
  - Post-order traversal : `Left Subtree → Right Subtree → Root`

  Breadth-First Search (BFS)

  - Level-order traversal : from top (`level=0`) to bottom (`level=height-1`)

Binary Tree Implementation


```
typedef struct _Node {
    int item;
    struct _Node *left, *right;
} Node;
```

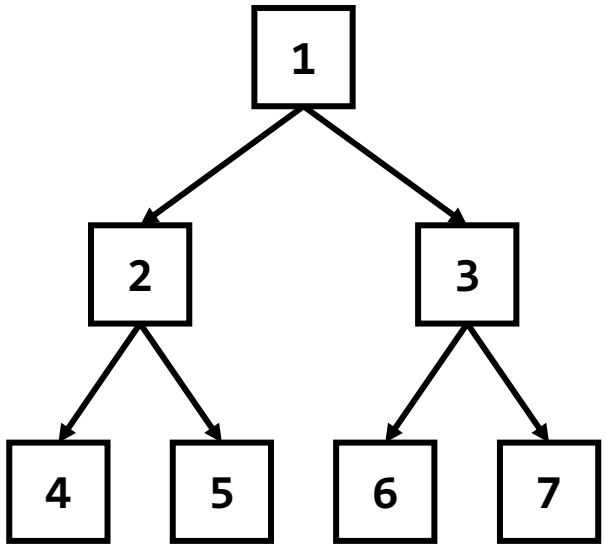
- In general, **the (linked-)list structure** is suitable for BT implementation
  - The tree is **non-linear** structure, which is not fit with the array structure
  - Since degree ≤ 2, the node structure can be easily implemented
  - Insertion and deletion are easier to implement

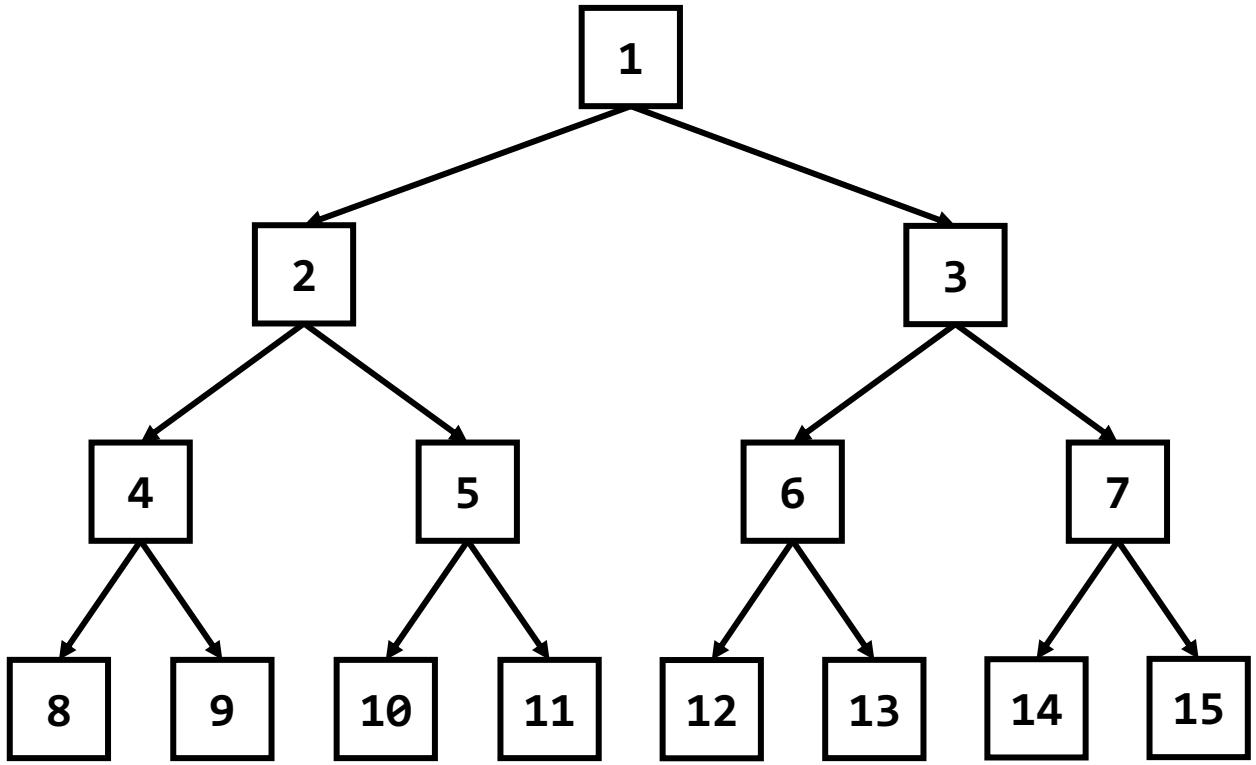
Binary Tree Implementation


```c
typedef struct _Node {
    int item;
    struct _Node *left, *right;
} Node;

Node* createNode(int item, Node *left, Node *right); // Create a node with subtrees
void removeNode(Node *node); // Delete the node and its all descendants
int computeHeight(Node *node); // Compute height of the subtree rooted at the node

void traverseInOrder(Node *node);
void traversePreOrder(Node *node);
void traversePostOrder(Node *node);
void traverseLevelOrder(Node *node);
```

Full & Complete Binary Trees

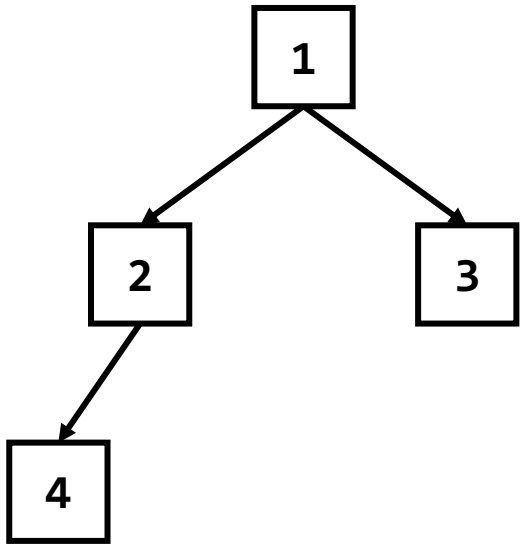
- **Full Binary Tree** is a BT of height H has $2^H - 1$ nodes
  - Node numbering from lower to higher levels, from left to right

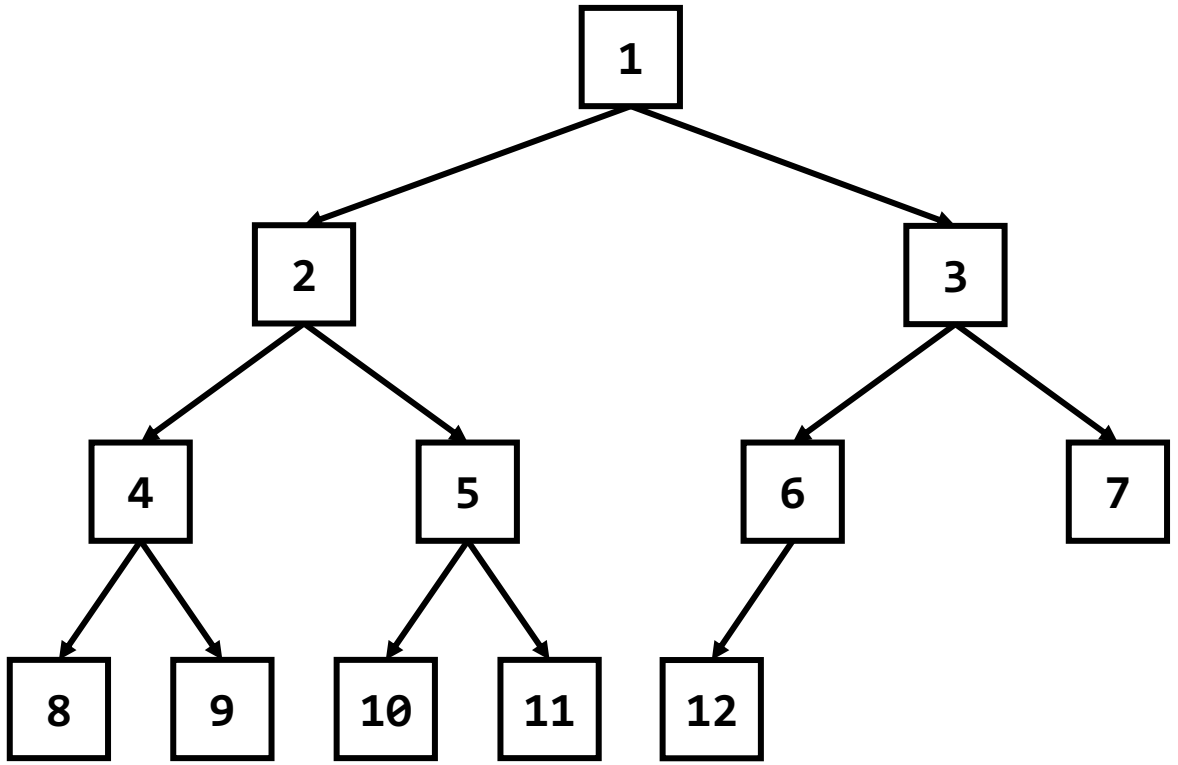




Full & Complete Binary Trees

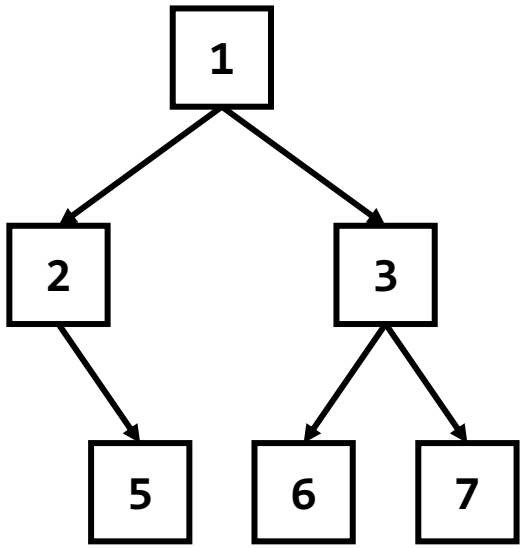
- **Complete Binary Tree** is a BT satisfying …
  - All nodes are sequentially filled from lower to higher levels, from left to right
  - The same node numbering to the full binary tree

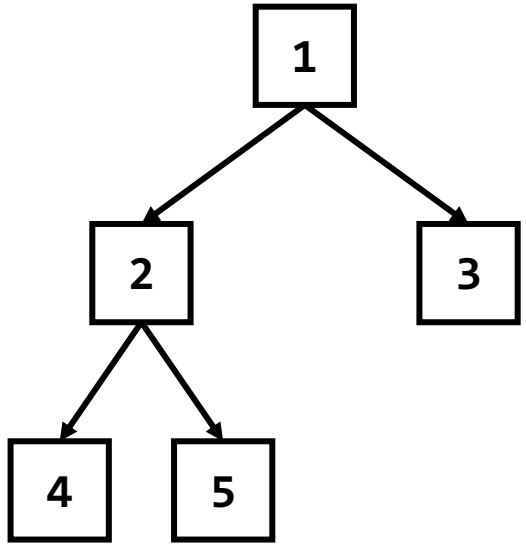


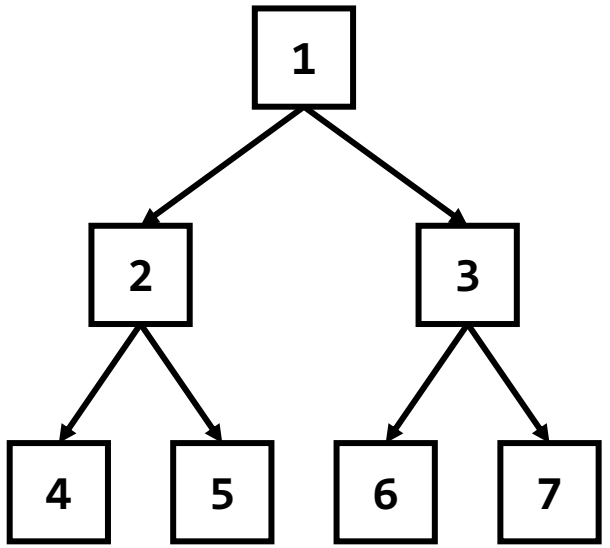


Full & Complete Binary Trees

(Q1) Is it a full or complete binary tree? Or none of both?







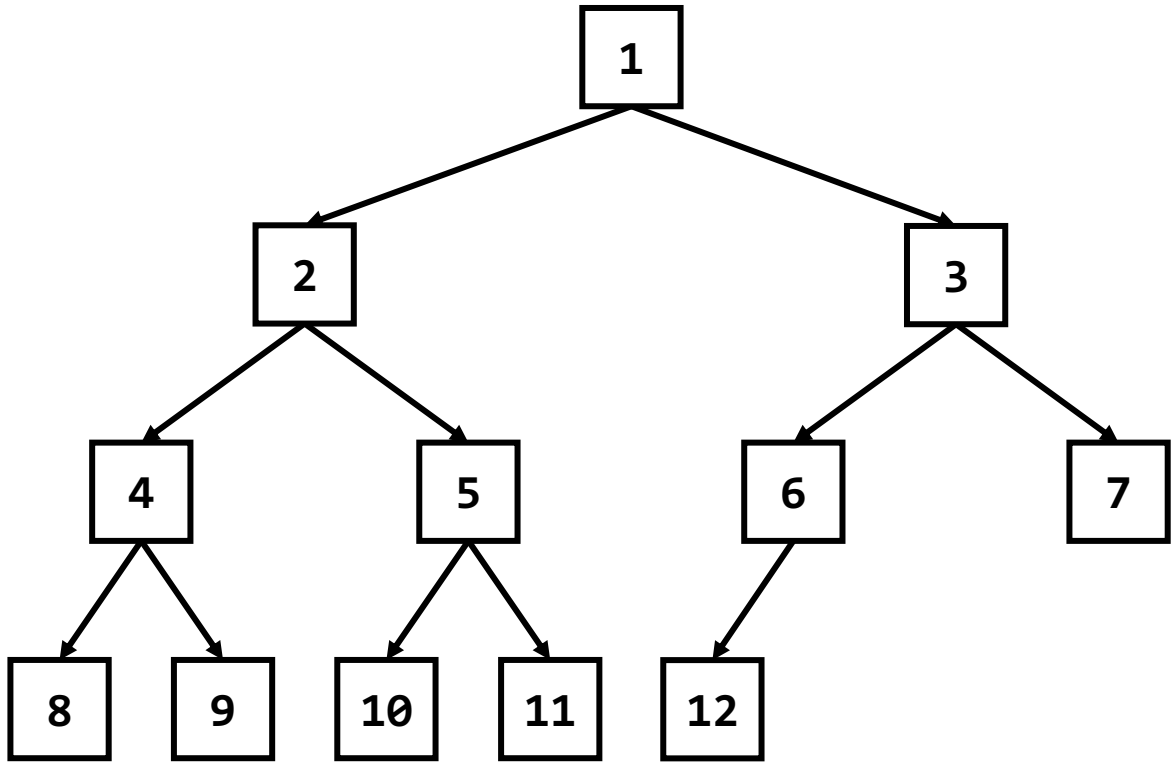
(Q2) Is any full binary tree a complete binary tree?

(Q3) How many nodes are required to make a complete BT to a full one?

- Assume N is the number of the complete BT

Complete Binary Tree Implementation

- The nodes in a complete binary tree are **sequentially filled**
  - There exists the **unique node numbering**
  - You can efficiently implement a complete BT using the array structure

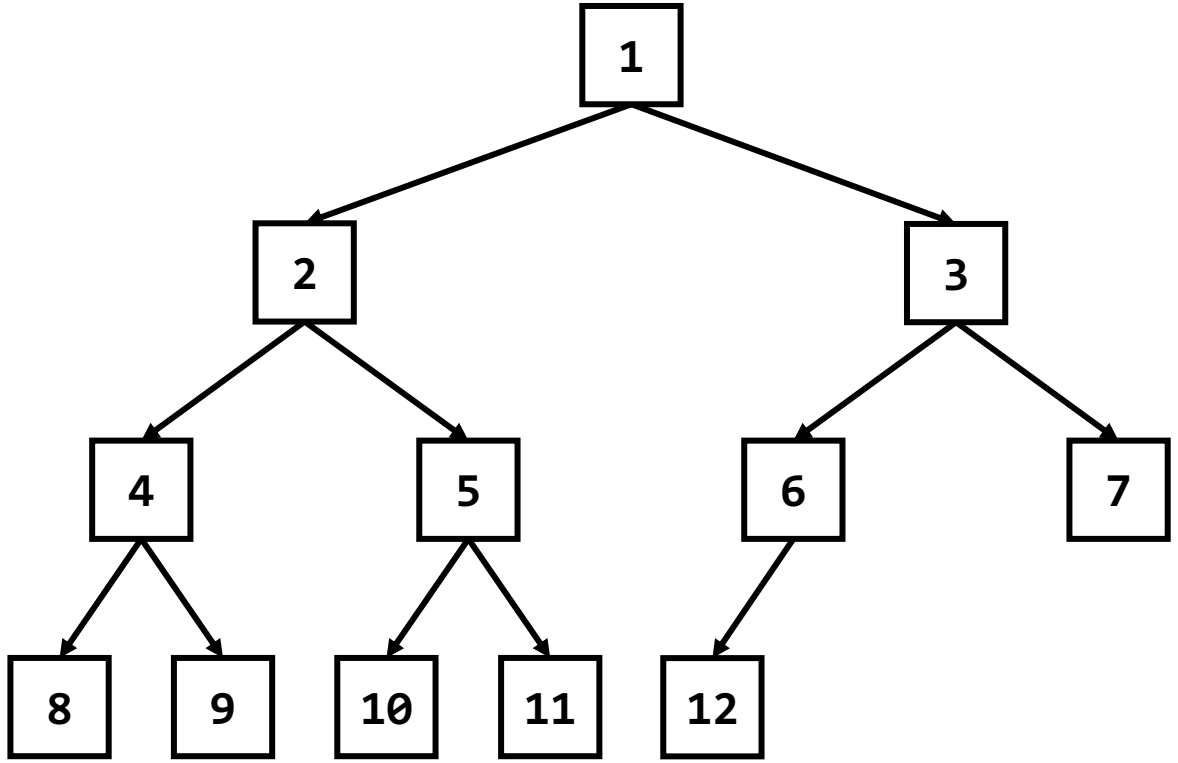


Complete Binary Tree Implementation

- The nodes in a complete binary tree are **sequentially filled**
  - There exists the **unique node numbering**
  - You can efficiently implement a complete BT using the array structure

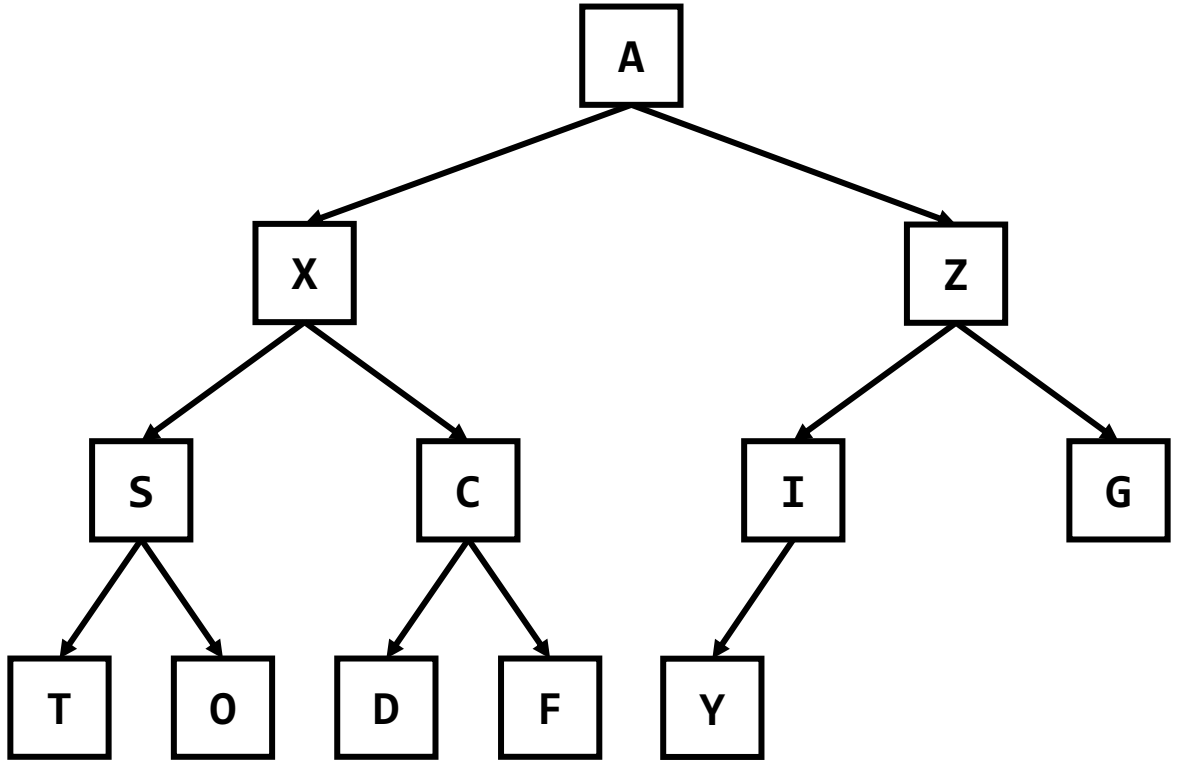
Interesting property of the numbering:

- Let `i` be a node number
- The parent number of `i` is `i/2` (floor)
- The left child number of `i` is `i*2`
- The right child number of `i` is `i*2+1`
- Check the node is left or right by `i%2==0`
- Move to its sibling by `i+1` or `i-1`



Complete Binary Tree Implementation

- Implement the following functions for traversal



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	X	Z	S	C	I	G	T	O	D	F	Y				

Complete Binary Tree Implementation

- Implement the following functions for traversal

```
int items[MAX_SIZE]; // Array for nodes in a complete binary tree
int N; // The number of nodes in the tree

void printParent(char tree[], int n);
void printChildren(char tree[], int n);
void printSibling(char tree[], int n);
void printAncestors(char tree[], int n);
void printDescendants(char tree[], int n);
```

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	X	Z	S	C	I	G	T	O	D	F	Y				

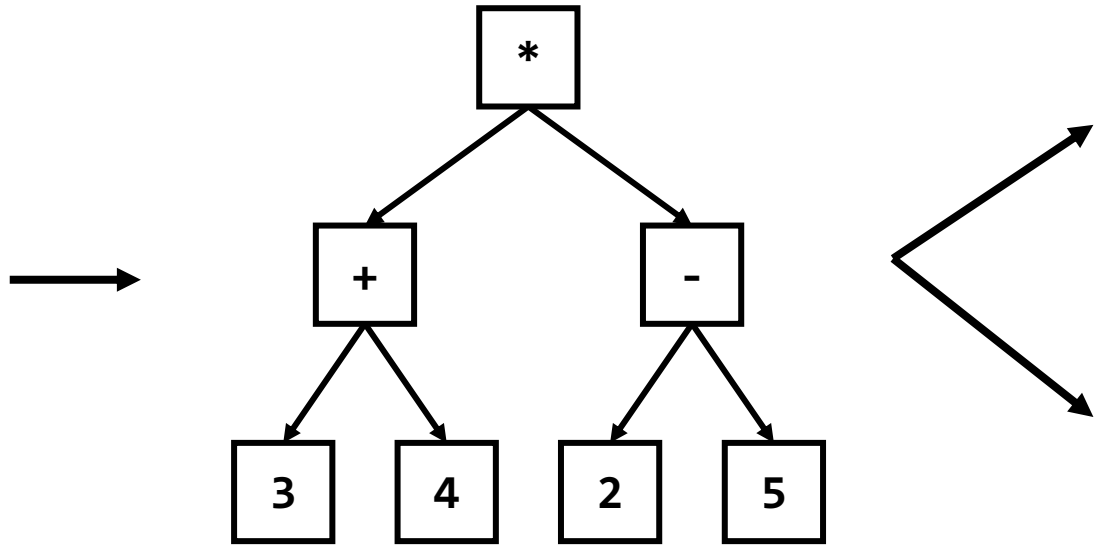
Array-based Complete BT Representation

Binary Trees - Problem Solving Practice

- Problem: **Expression Tree**
  - **(Q1)** Given a postfix expression, construct a binary tree to represent the expression
  - **(Q2)** Implement a function that evaluate the expression and print its infix expression

Postfix Expression

```
3 4 + 2 5 - *
```



Evaluation Result

```
-21
```

Infix Expression

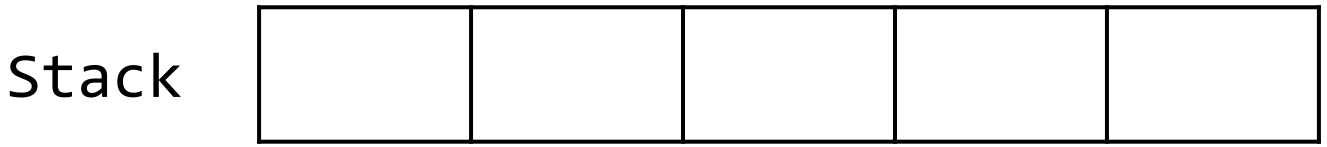
```
(3 + 4) * (2 – 5)
```

Binary Trees - Problem Solving Practice

- Problem: **Expression Tree**
  - **(Q1)** Given a postfix expression, construct a binary tree to represent the expression
  - **(A1)** Read operands and construct subtrees from left to right

Postfix Expression

3 4 + 2 5 - *

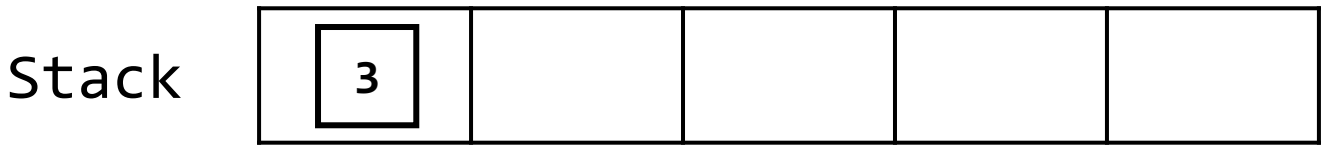


Binary Trees - Problem Solving Practice

- Problem: **Expression Tree**
  - **(Q1)** Given a postfix expression, construct a binary tree to represent the expression
  - **(A1)** Read operands and construct subtrees from left to right

Postfix Expression

3 4 + 2 5 - *

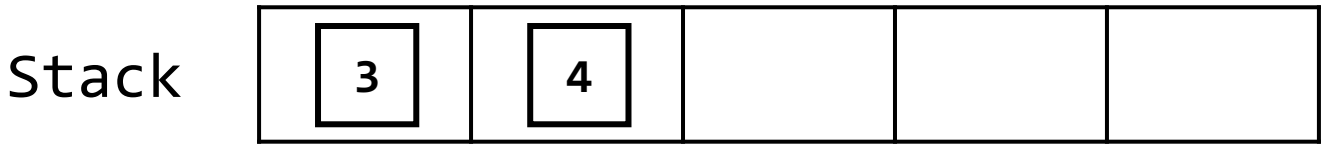


Binary Trees - Problem Solving Practice

- Problem: **Expression Tree**
  - **(Q1)** Given a postfix expression, construct a binary tree to represent the expression
  - **(A1)** Read operands and construct subtrees from left to right

Postfix Expression

3 4 + 2 5 - *



Binary Trees - Problem Solving Practice

- Problem: **Expression Tree**
  - **(Q1)** Given a postfix expression, construct a binary tree to represent the expression
  - **(A1)** Read operands and construct subtrees from left to right

Postfix Expression

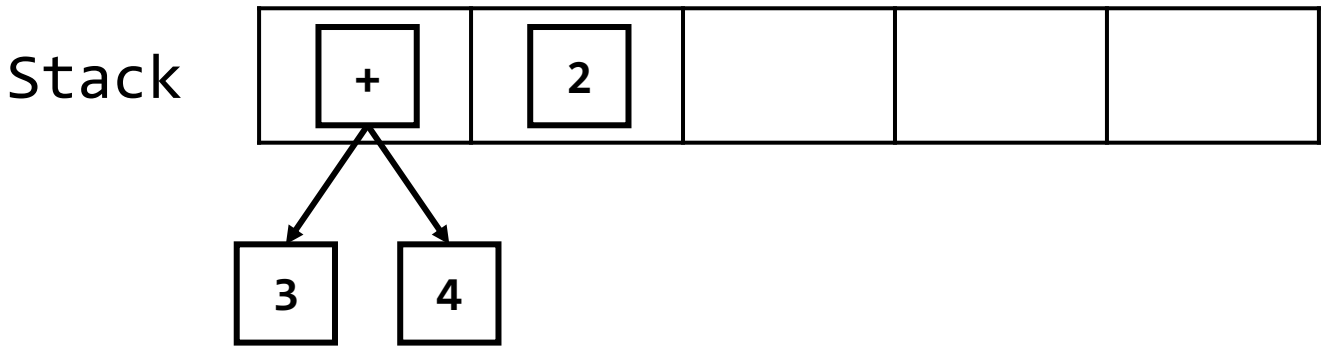
3 4 + 2 5 - *

Binary Trees - Problem Solving Practice

- Problem: **Expression Tree**
  - **(Q1)** Given a postfix expression, construct a binary tree to represent the expression
  - **(A1)** Read operands and construct subtrees from left to right

Postfix Expression

3 4 + 2 5 - *

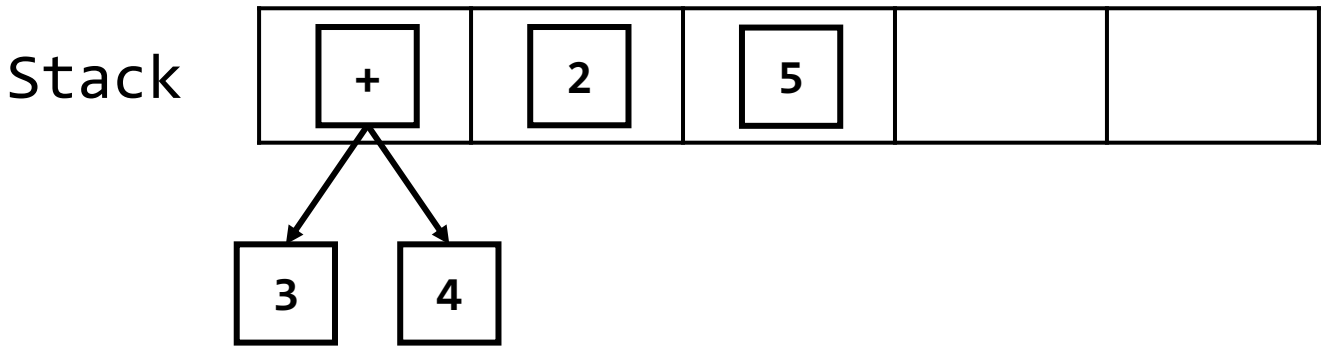


Binary Trees - Problem Solving Practice

- Problem: **Expression Tree**
  - **(Q1)** Given a postfix expression, construct a binary tree to represent the expression
  - **(A1)** Read operands and construct subtrees from left to right

Postfix Expression

3 4 + 2 5 - *

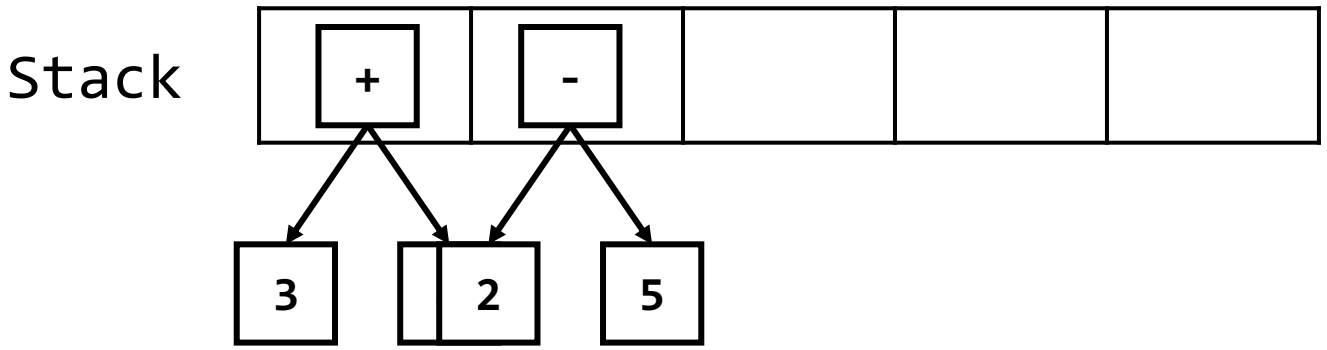


Binary Trees - Problem Solving Practice

- Problem: **Expression Tree**
  - **(Q1)** Given a postfix expression, construct a binary tree to represent the expression
  - **(A1)** Read operands and construct subtrees from left to right

Postfix Expression

3 4 + 2 5 - *

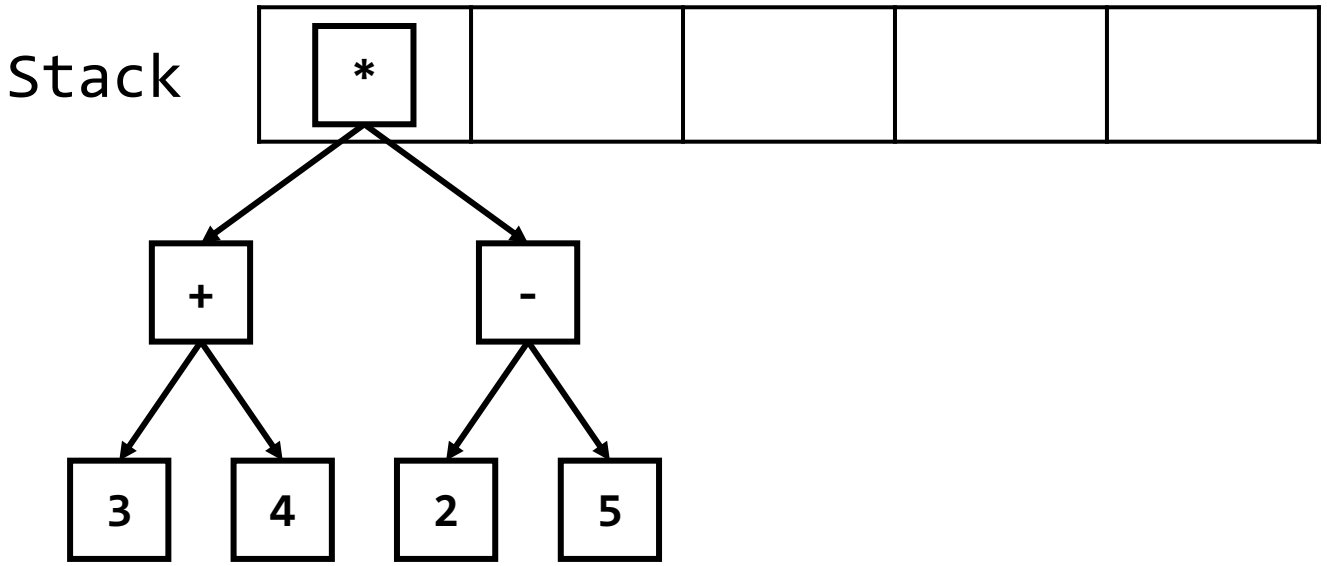


Binary Trees - Problem Solving Practice

- Problem: **Expression Tree**
  - **(Q1)** Given a postfix expression, construct a binary tree to represent the expression
  - **(A1)** Read operands and construct subtrees from left to right

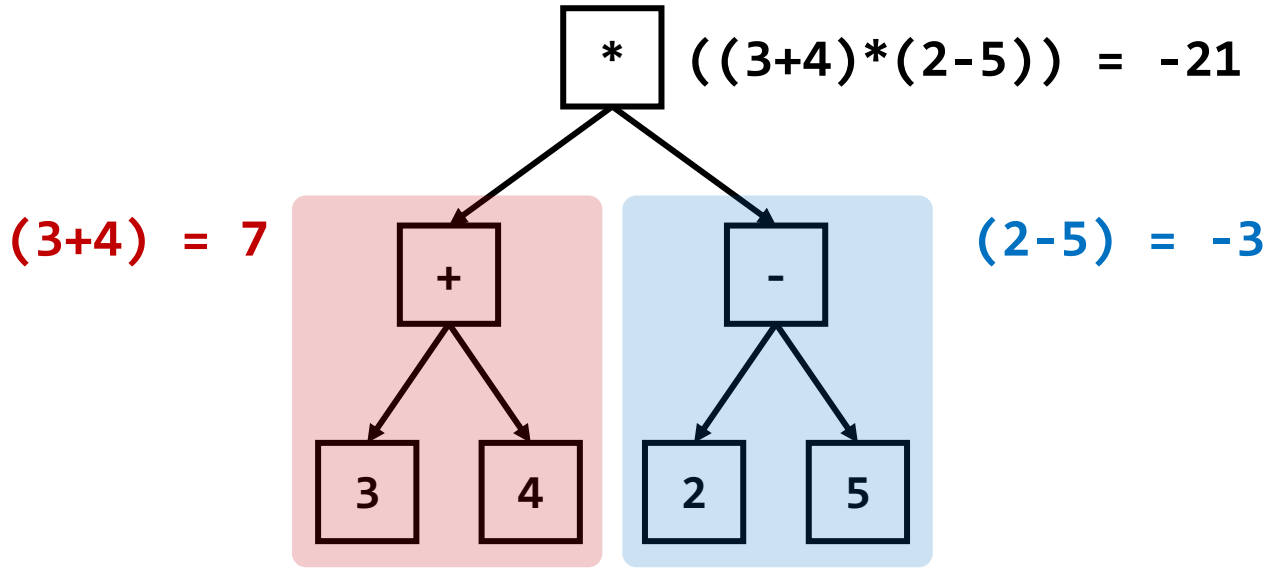
Postfix Expression

3 4 + 2 5 - *



Binary Trees - Problem Solving Practice

- Problem: **Expression Tree**
  - **(Q2)** Implement a function that evaluate the expression and print its infix expression
  - **(A2)** It can be easily implemented using recursion



Any Questions?