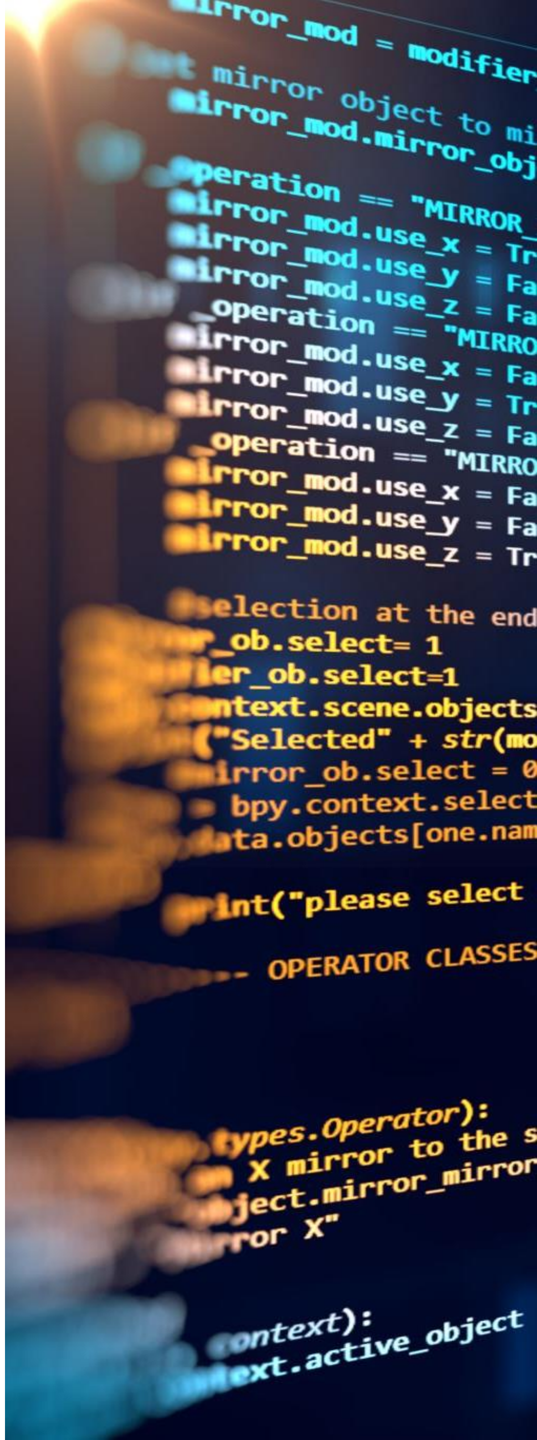
Please use the following QR code to check in and record your attendance.

CS 1037
Fundamentals of Computer Science II

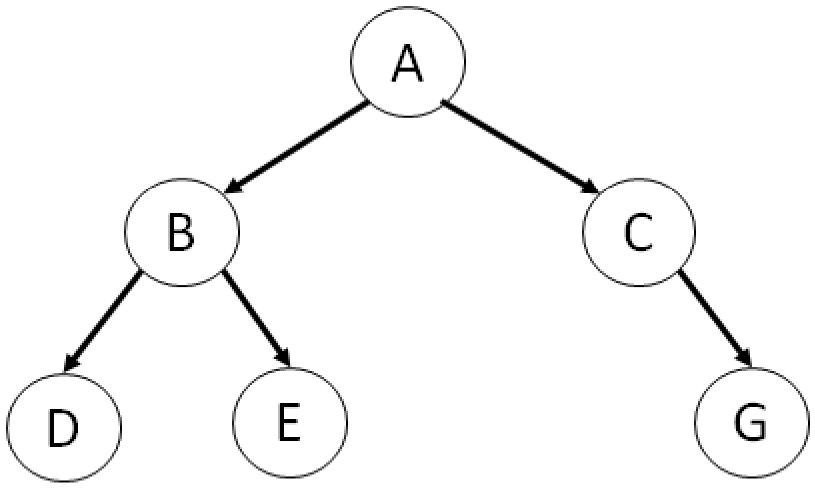
Tree ADT

Ahmed Ibrahim



The Concepts of Trees

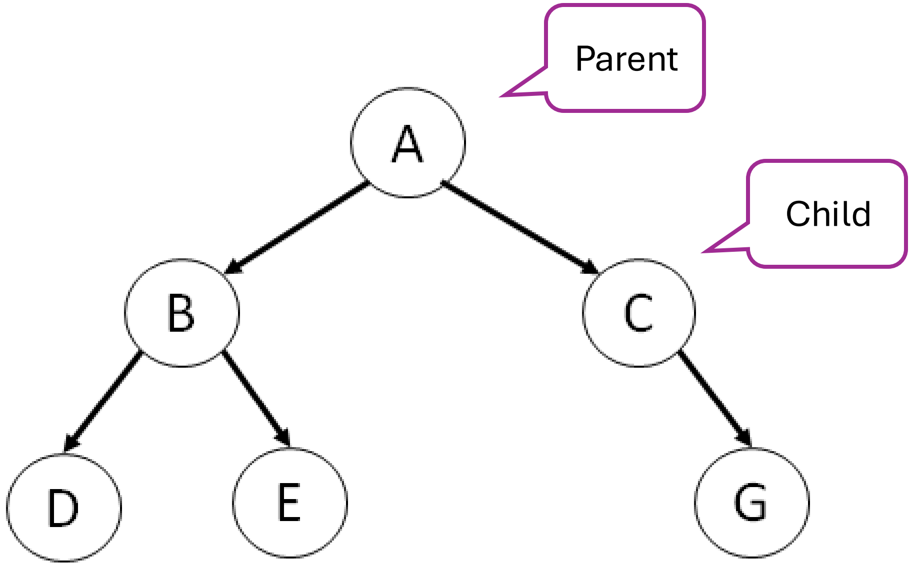
- The concepts of trees consist of abstract trees (or abstract data structures) and tree data structures (the implementations of abstract trees).
- An abstract tree is defined as a collection of nodes connected in a **tree structure** (or topology), together with a set of operations such as traversal, search, insertion, and deletion.



Tree diagram example

The Tree Structure

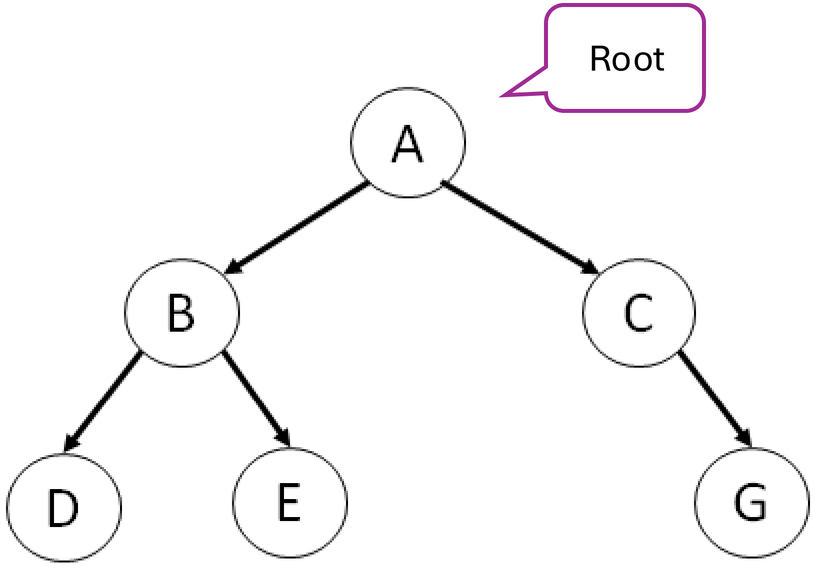
- A tree structure T consists of a set of nodes V (called vertex)and a set E of node-to-node relations (called edges), denoted as T = (V, E).
- Each edge in E represents a connection from a node **u** to a node **v**, written as (**u**, **v**), and **u** is called the parent of **v**, and **v** is the child of **u**. (V, E) satisfies the following conditions:
  - Each node has zero or more children.
  - There is a unique node called root, which is not a child of any node.
  - A unique route (path) from the root to any other node exists.



Tree diagram example

Unique Path from the Root

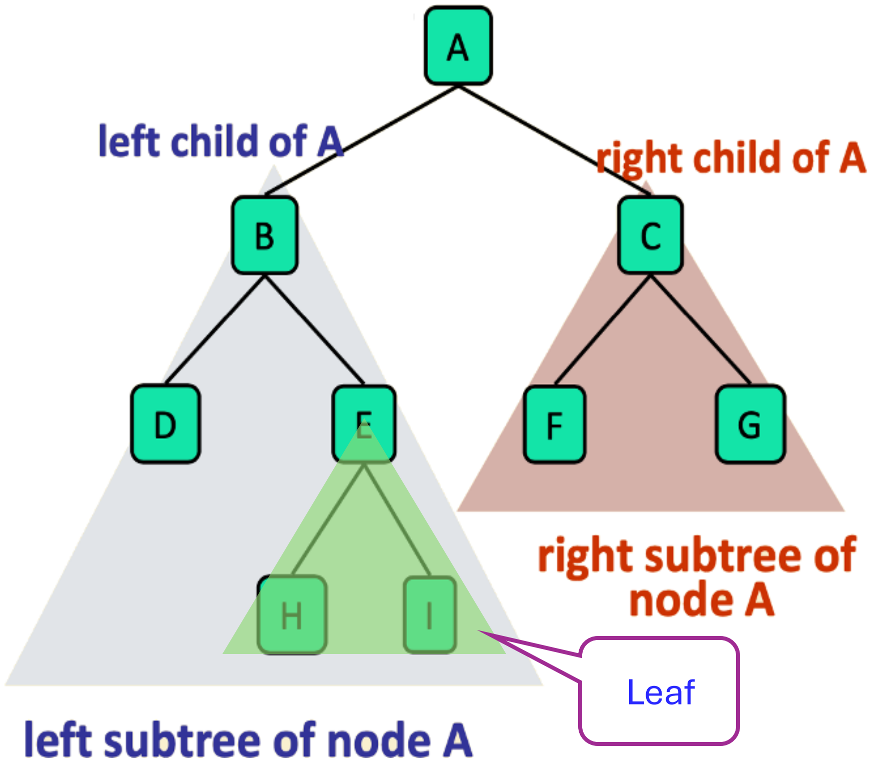
- In the following graph, the root node is A. Here are the possible unique paths from the root A to each other node:
  1. Path from A to B: A→B
  2. Path from A to C: A→C
  3. Path from A to D: A→B→D
  4. Path from A to E: A→B→E
  5. Path from A to G: A→C→G
- Each path follows the directed edges in the graph.



Tree diagram example

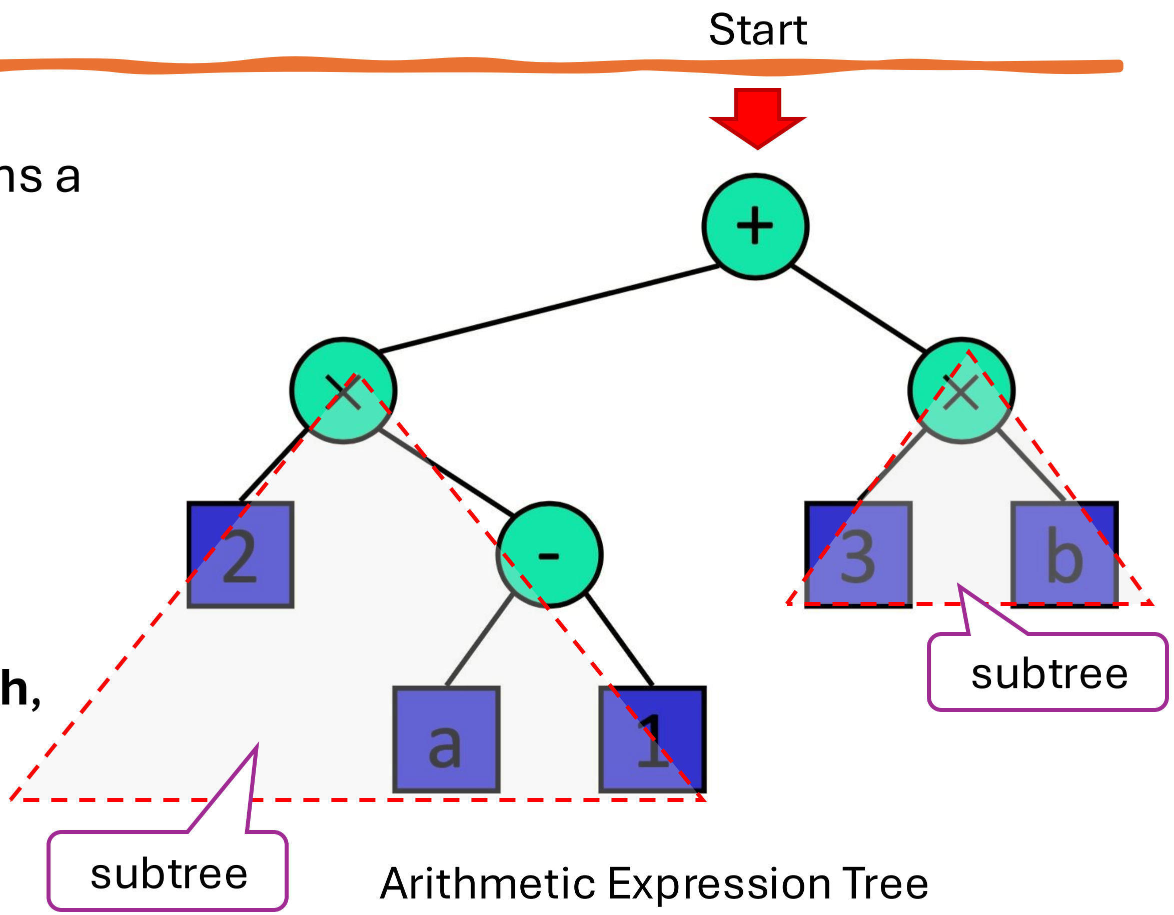
Tree Definition

- A recursive tree definition defines a tree in terms of itself. A tree is defined recursively as a collection of nodes with the following properties:
    1. A single node (leaf node) — this is the **simplest tree**.
    2. Or, a root node connected to a collection of smaller trees (subtrees), each following the same recursive structure.



Tree ADT

- By tree definition, each non-empty tree contains a unique root. Each child of the root node is the root of a subtree.
- Data values are stored in tree nodes.
- Tree operations access a tree from its root.
- Basic tree operations include **traversal**, **search**, **insertion**, and **deletion**.



Arithmetic Expression Tree

Tree Data Structures

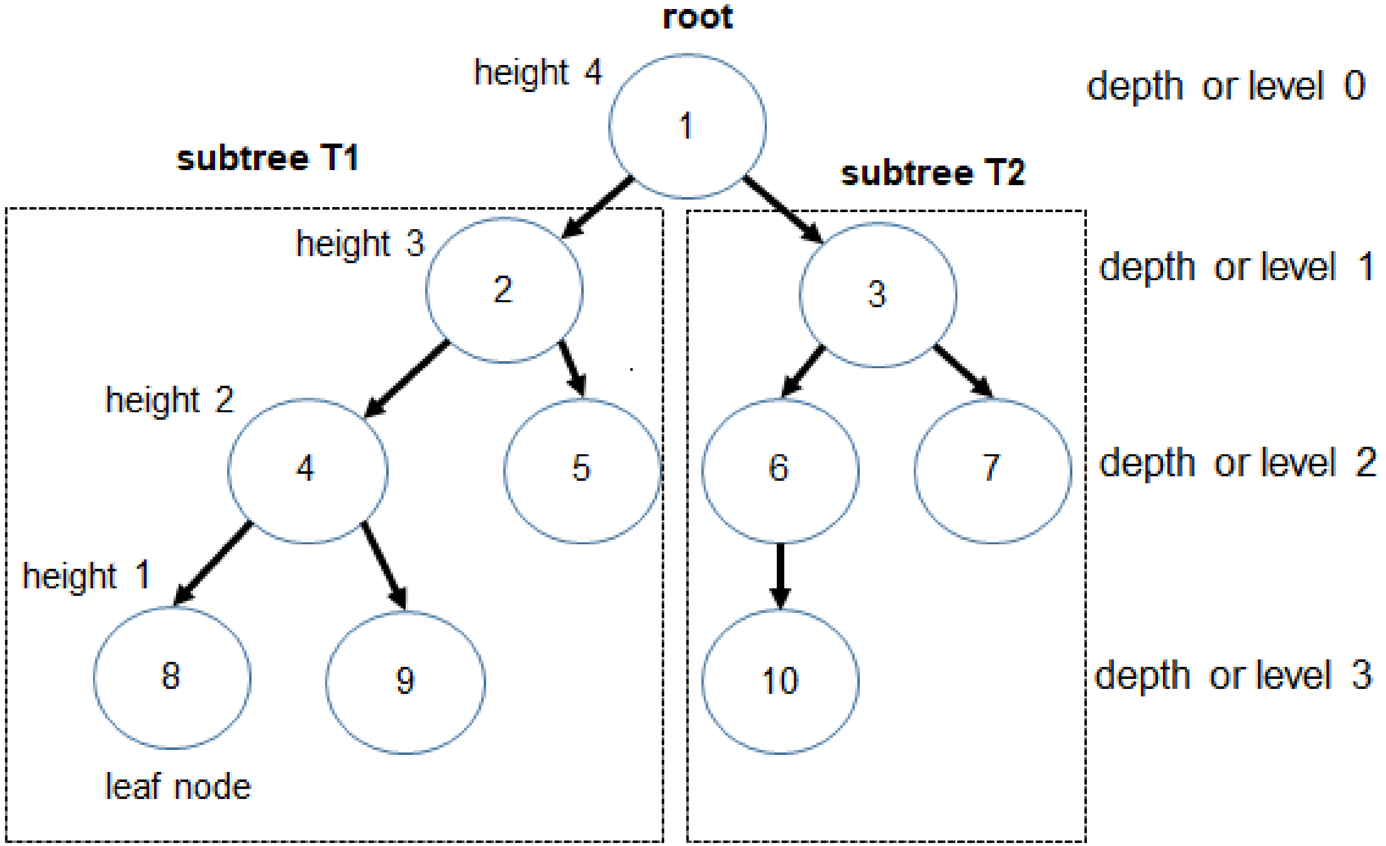
- A **tree data structure** (or simply a tree) is an implementation of an abstract tree in a programming language in which the parent-child relations of nodes are represented by a specific method.
- For example, a tree can be represented by the **linked node representation** in C using a node structure consisting of a data part and a link part containing a list of pointers to its children.

```c
typedef struct node {
int data;
struct node *left;
struct node *right;
} TNODE;
```

The following structure defines a tree node, where each node can have up to two children: a left child and a right child.

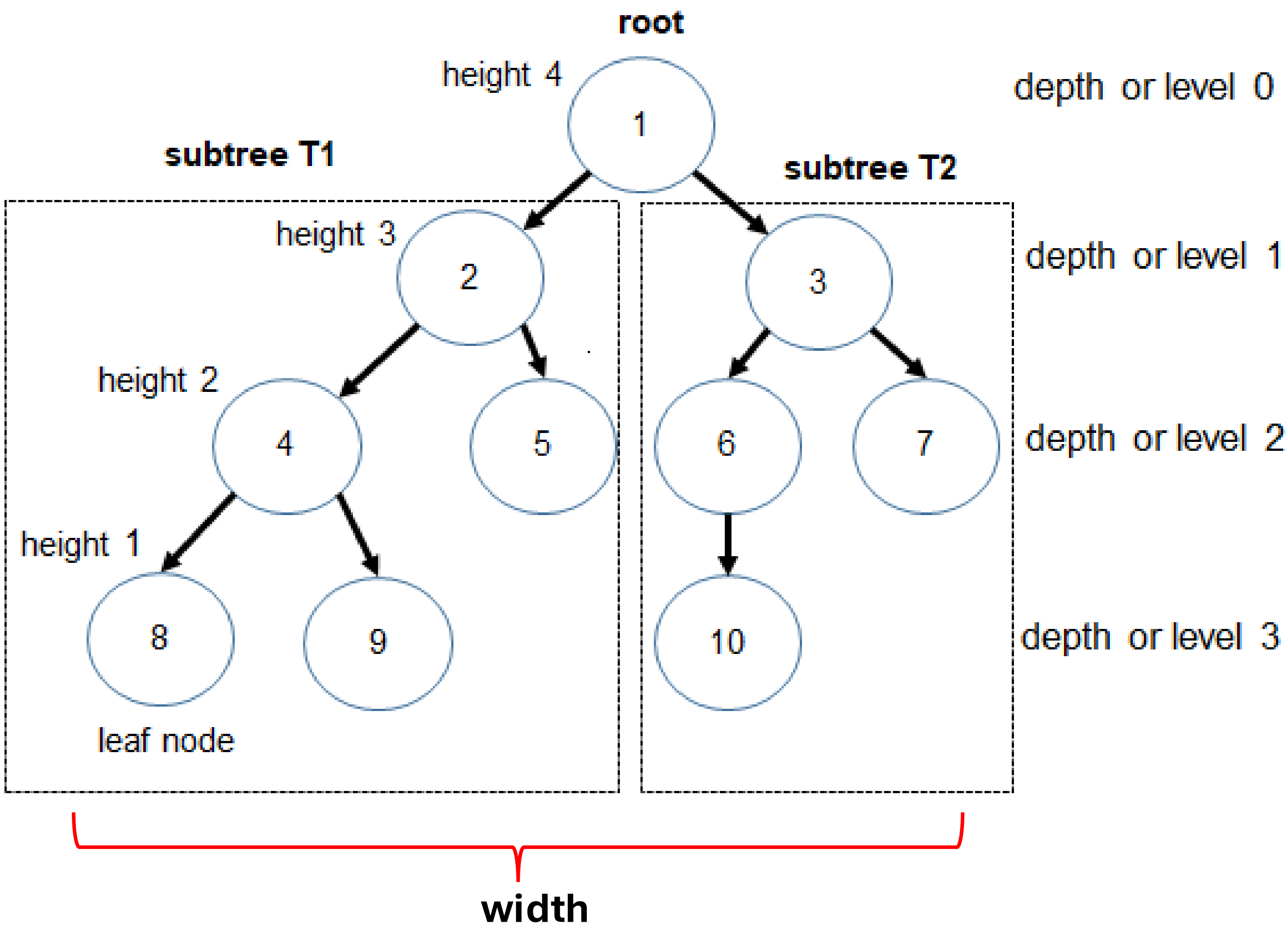
Tree Terminology

- The **depth** of a node is the number of ancestors
- The **height** of a tree is the maximum depth from the root
  - 3 in this example
- **Descendant** of a node is a child, grandchild, grand-grandchild, etc.
- **Siblings** are two nodes that are children of the same parent



Tree terms

The **width** of a tree refers to the maximum number of nodes overall levels of a tree.
Note – A tree does not contain a cycle.



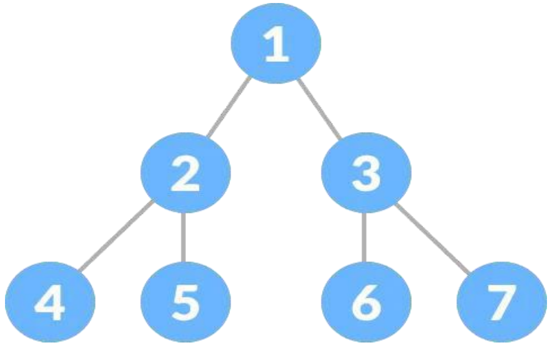
Classification of Trees

- Trees can be classified into many categories by their **properties** and **applications**. We will look into the following categories.
  - General trees – no restriction
  - Binary trees – each node has at most two children.
  - Binary search trees – binary trees for efficient searching
    - Ex. **AVL trees:** height-balanced binary search trees
  - Multi-way search trees – a generalization to binary search trees
    - Ex. **B-trees** – balanced multi-way search trees

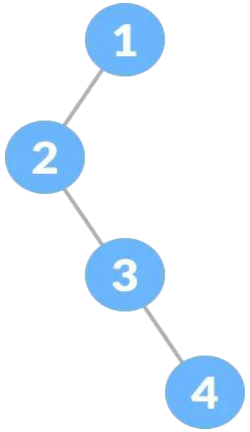
Classification of Trees (cont.)

```
typedef struct node {
int data;
struct node *child[k];
} TNODE
```

defines an array of pointers to child nodes



A binary tree



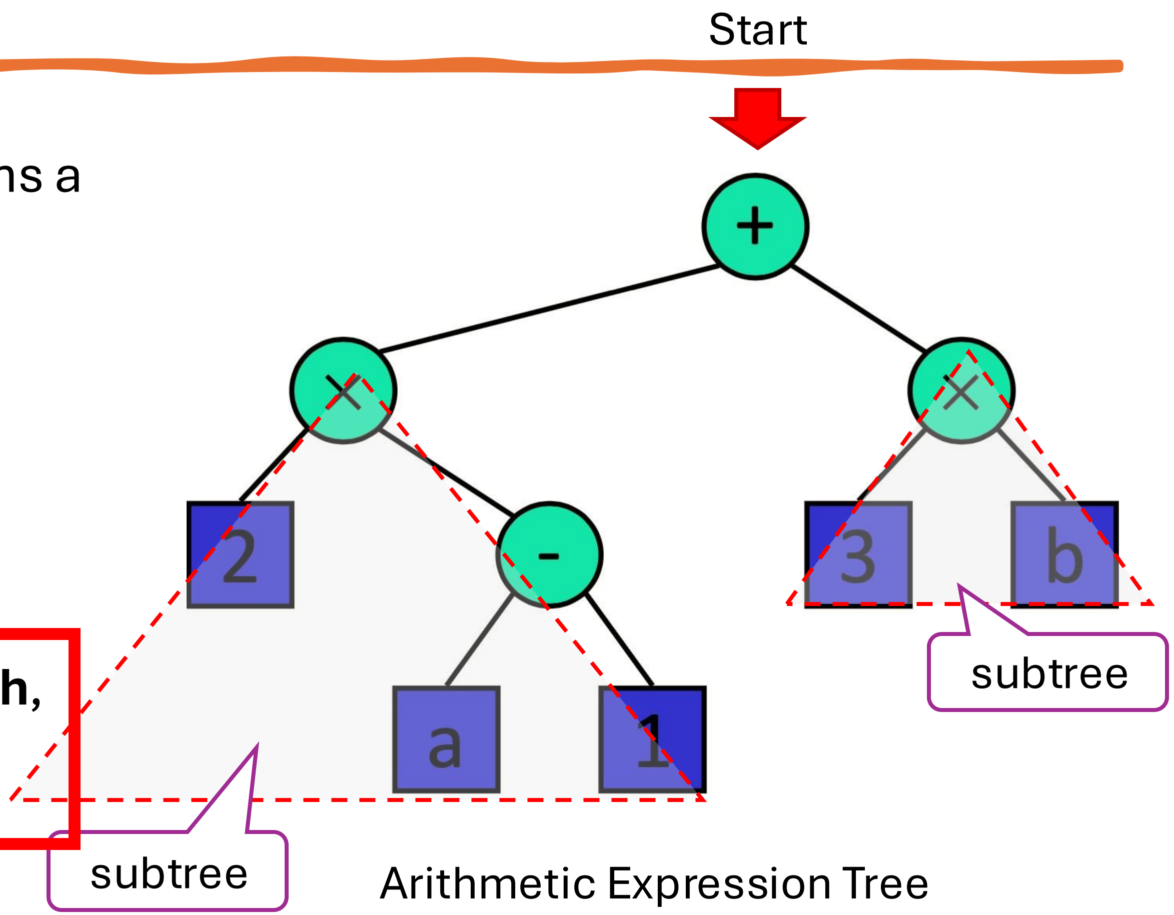
A degenerate tree is a tree that has a single child, either left or right.

- A node in a **general tree** has zero or more children.
- If a node is allowed to have at most two children, then it is called a **binary tree**.
- If a node is allowed to have at most three children, then it is called a **ternary tree**.
- In general, a node is allowed to have k children, it is called an **k-way tree** or **k-tree**.

Tree **Traversal** Techniques

Recall: Tree ADT

- By tree definition, each non-empty tree contains a unique root. Each child of the root node is the root of a subtree.
- Data values are stored in tree nodes.
- Tree operations access a tree from its root.
- Basic tree operations include **traversal**, **search**, **insertion**, and **deletion**.



Arithmetic Expression Tree

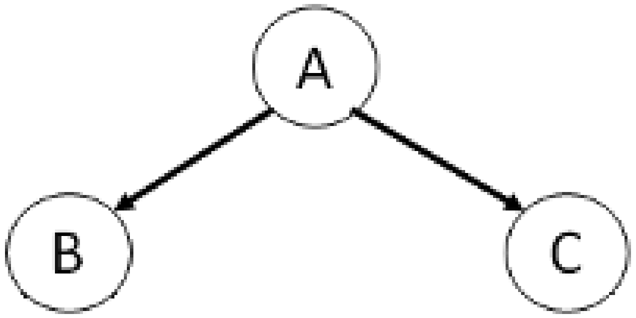
Tree Traversal

Traversal refers to visiting each node in the tree in a specific order.

There are several types of traversal methods:

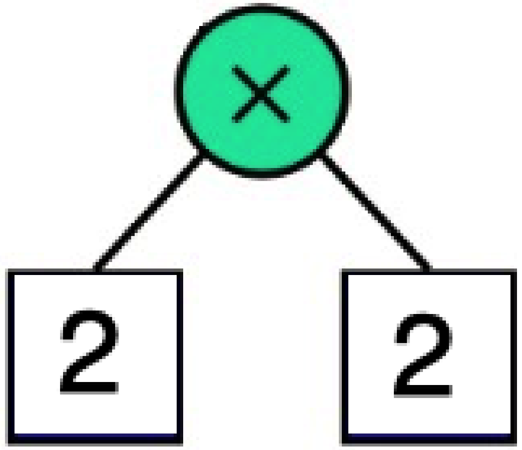
- **Pre-order Traversal** (Root -> Left -> Right): Visit the root node first, then recursively traverse the left subtree, followed by the right subtree.
- **In-order Traversal** (Left -> Root -> Right): Recursively traverse the left subtree, visit the root node, and then traverse the right subtree. This is commonly used in binary search trees to **retrieve values in sorted order**.

what does that mean?



Tree Traversal (cont.)

- **Post-order Traversal** (Left -> Right -> Root): Recursively traverse the left and right subtrees before visiting the root node. This is useful for **deleting nodes** in a tree or **evaluating expression trees**.
- **Level-order Traversal** (Breadth-First): Traverse the tree level by level, visiting all nodes at each level before moving to the next. It begins at the root node. This is typically implemented using a **queue** and is used in the **breadth-first search (BFS)** algorithm.



Recursion

- Recursion is a programming technique where a function calls **itself** in order to solve a problem.
- Each recursive call works on a **smaller part** of the problem until a base condition is met, which stops the recursion.
- Recursion is commonly used in algorithms with hierarchical or **repetitive** data structures, such as **trees**, linked lists, and mathematical sequences.

Key Components of Recursion

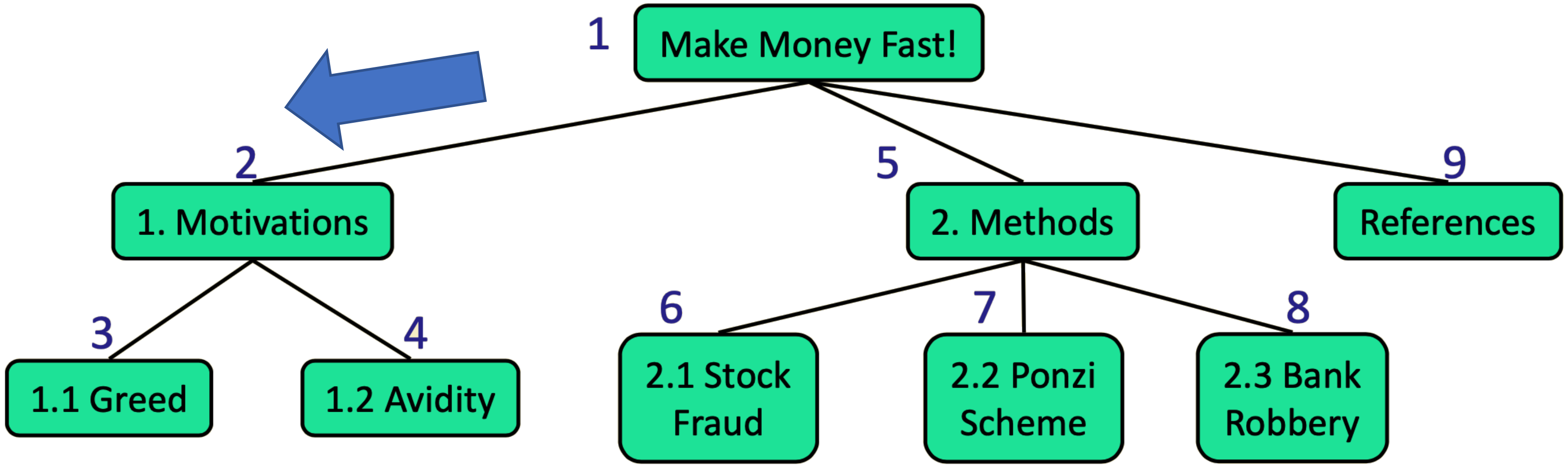
Base Case: The condition under which the recursive calls stop. This prevents infinite recursion by providing a condition that leads to an immediate answer.

Recursive Case: The part of the function that breaks down the problem and makes the recursive call(s). Each recursive call should bring the problem closer to the base case.

```
int factorial(int n) {
// Base Case: If n is 0, return 1
if (n == 0) {return 1;}
// Recursive Case: Multiply n by the
factorial of (n - 1)
else {return n * factorial(n - 1);}
}
```

Tree Traversal: Preorder Traversal

- A node is visited before its descendants.
- When is it applied?
  - Use when must perform computations for a node before any computations for its descendants



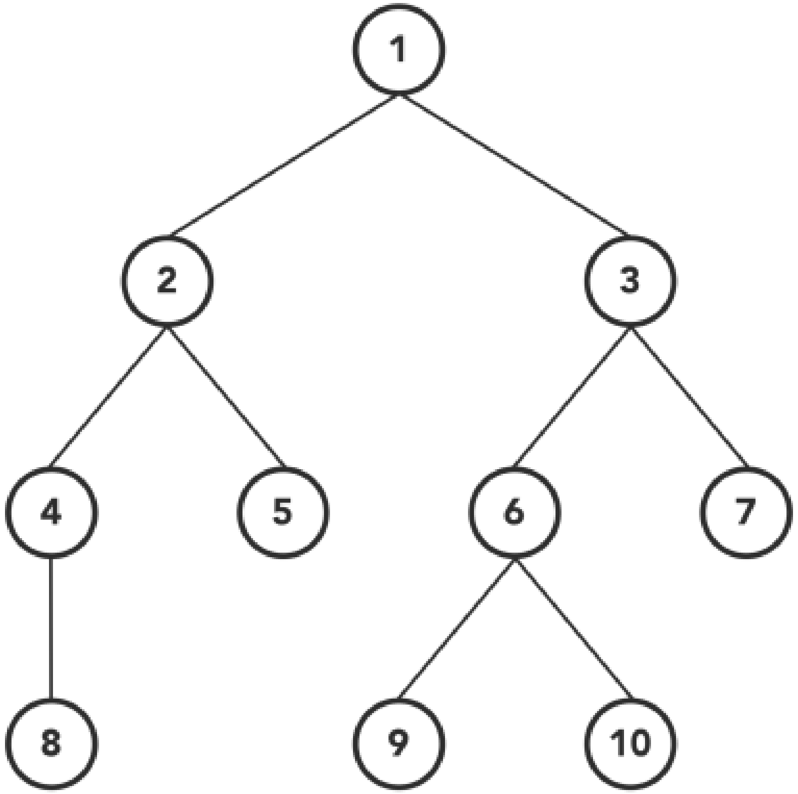

```
void preorder(Node* node) {
if (node == NULL)
return;

//root node
printf("%d ", node->data);
// Traverse left subtree
preorder(node->left);
// Traverse right subtree
preorder(node->right);
}
```

Preorder Traversal

```
void preorder(Node* node) {
if (node == NULL)
return;

//root node
printf("%d ", node->data);
// Traverse left subtree
preorder(node->left);
// Traverse right subtree
preorder(node->right);
}
```



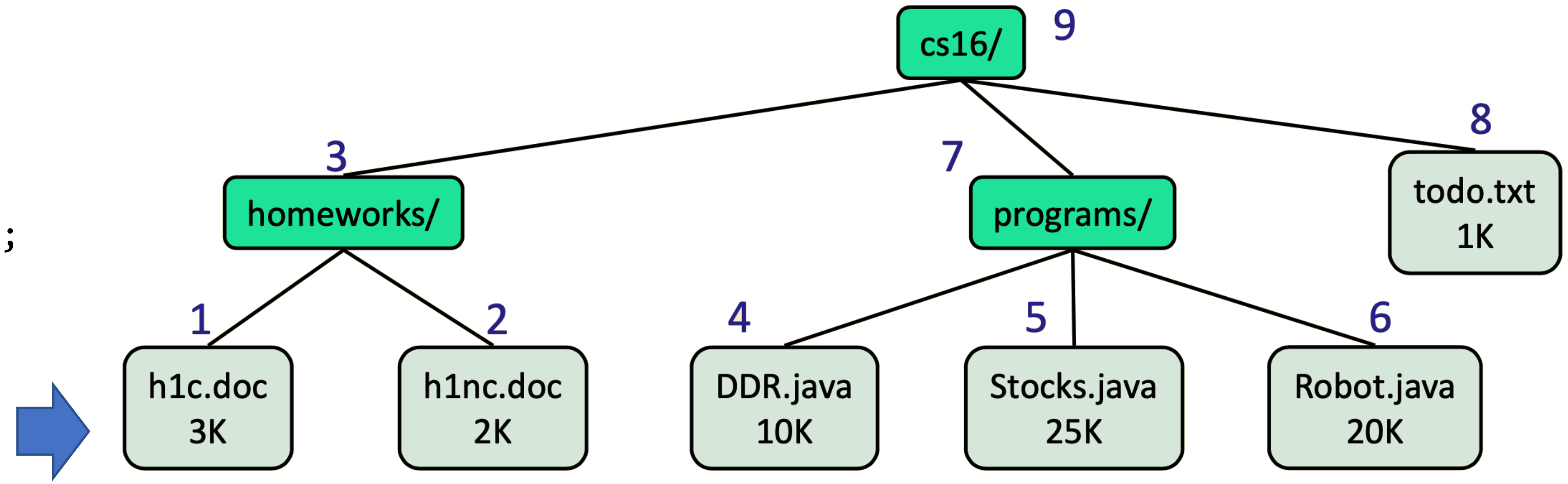
Source: https://towardsdatascience.com/4-types-of-tree-traversal-algorithms-d56328450846

Postorder Traversal

```
void postorder(Node* node) {
if (node == NULL)
return;

 // Traverse left subtree
 postorder(node->left);
 // Traverse right subtree
 postorder(node->right);
 // Visit node (Root)
 printf("%d ", node->data);
}
```

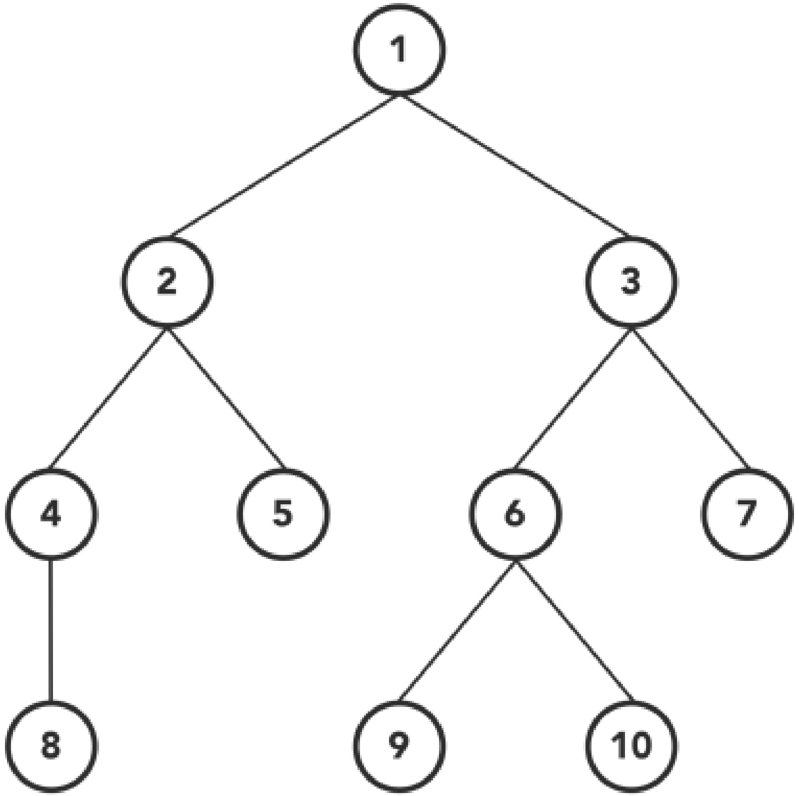
- Node is visited after its descendants
- When is it applied?
  - Visit leaf nodes first
  - trying to delete a tree



Postorder Traversal

```
void postorder(Node* node) {
if (node == NULL)
return;

 // Traverse left subtree
 postorder(node->left);
 // Traverse right subtree
 postorder(node->right);
 // Visit node (Root)
 printf("%d ", node->data);
}
```



Inorder Traversal

- In order traversal, the left subtree is visited first, followed by the root node and finally the right subtree.
- When is it applied? when we're trying to get the nodes in a **sorted order**.
- Inorder Traversal Application
  - Binary Search Trees (BST): Inorder traversal of a BST gives the nodes in a sorted order.
  - Creating a copy of a tree

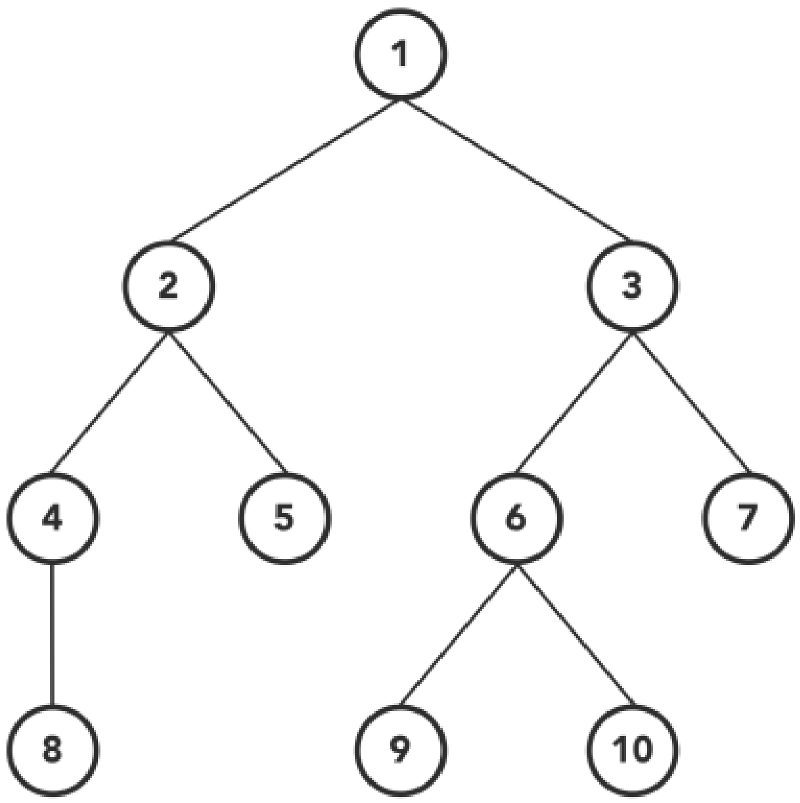
```
void inorder(Node* node) {
if (node == NULL)
return;

// Traverse left subtree
inorder(node->left);
// Visit node
printf("%d ", node->data);
// Traverse right subtree
inorder(node->right);
}
```

Inorder Traversal

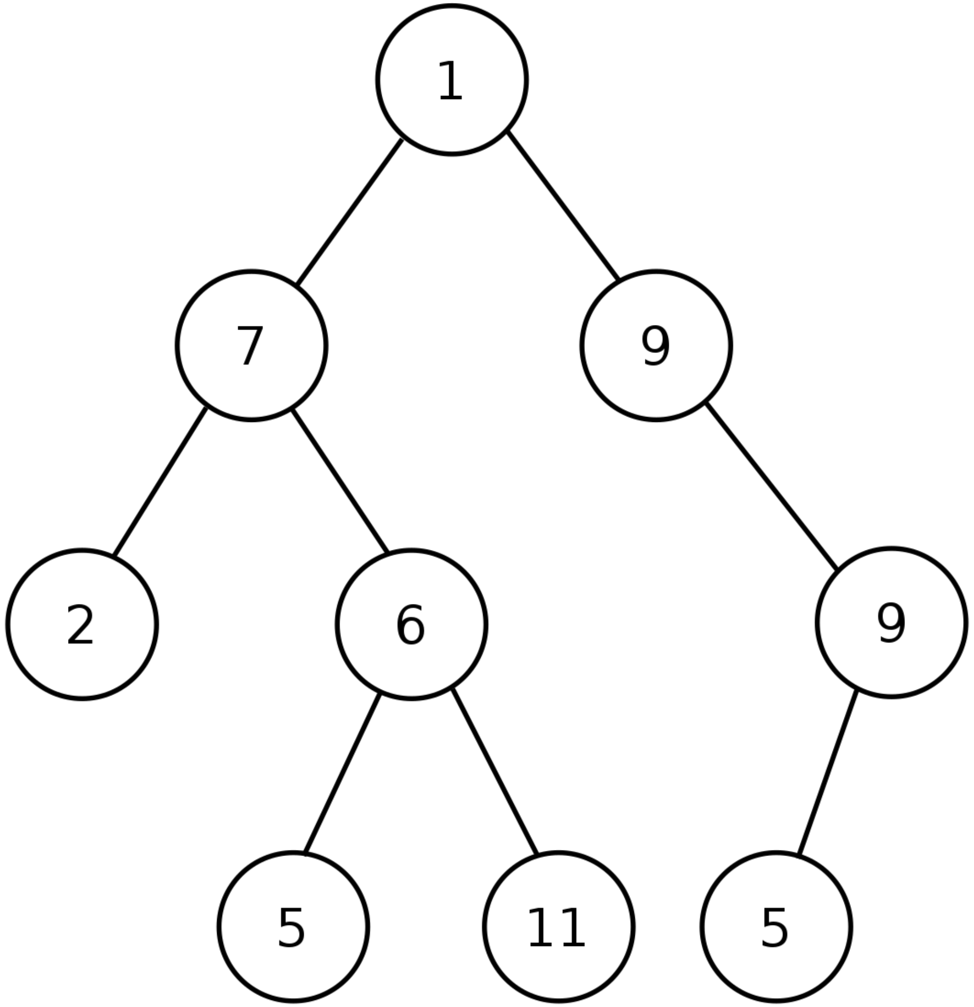
```
void inorder(Node* node) {
if (node == NULL)
return;

// Traverse left subtree
inorder(node->left);
// Visit node
printf("%d ", node->data);
// Traverse right subtree
inorder(node->right);
}
```



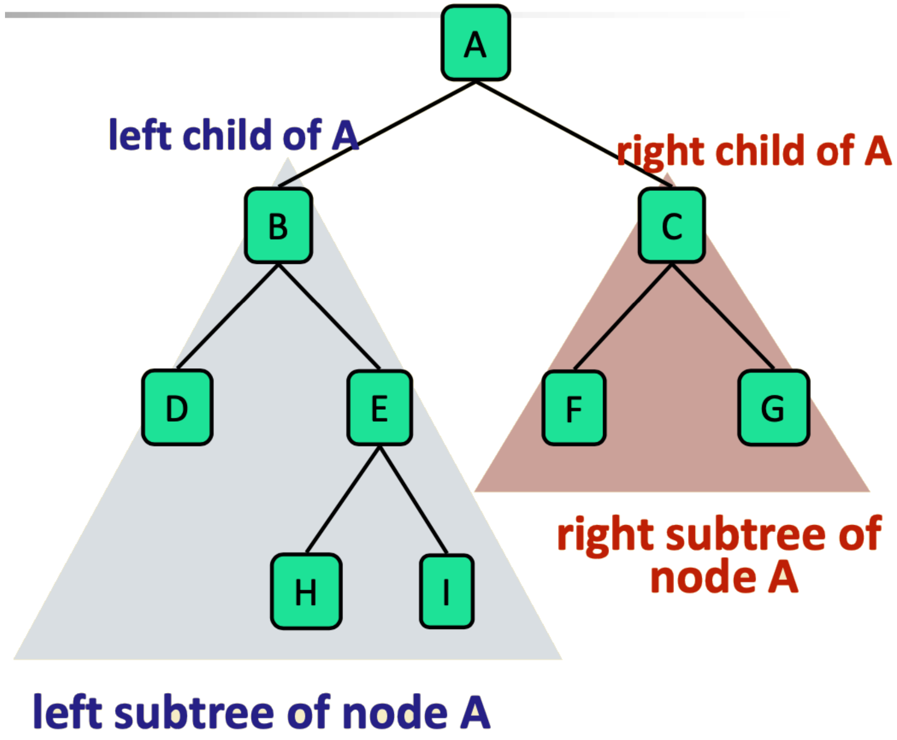
Source: https://towardsdatascience.com/4-types-of-tree-traversal-algorithms-d56328450846

Binary Trees



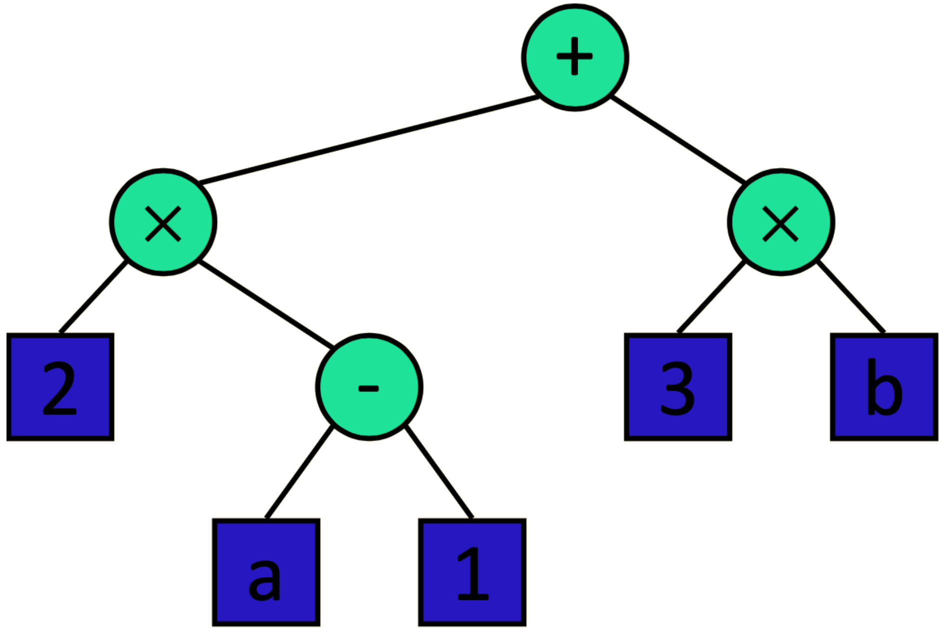
Binary Trees (BTs)

- In a **Binary tree,** a node has at most two children
- Children are an ordered pair
  - **left child** and **right child**
  - corresponding subtrees are the **left subtree** and **right subtree**
- In a binary tree, each internal node has exactly two children
- Applications
  - Arithmetic expressions
  - Decision processes



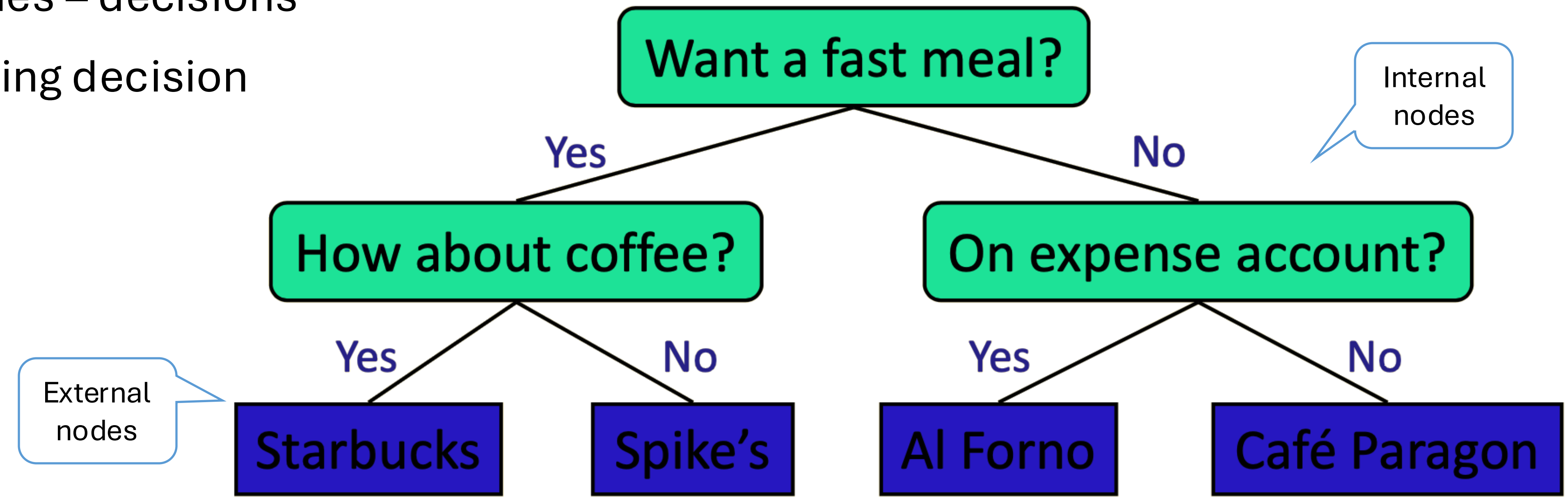
Arithmetic Expression Tree

- The binary tree associated with an arithmetic expression
  - internal nodes store operators
  - external nodes store operands
- Example: arithmetic expression tree for the expression: $(2 \times (a - 1) + (3 \times b))$



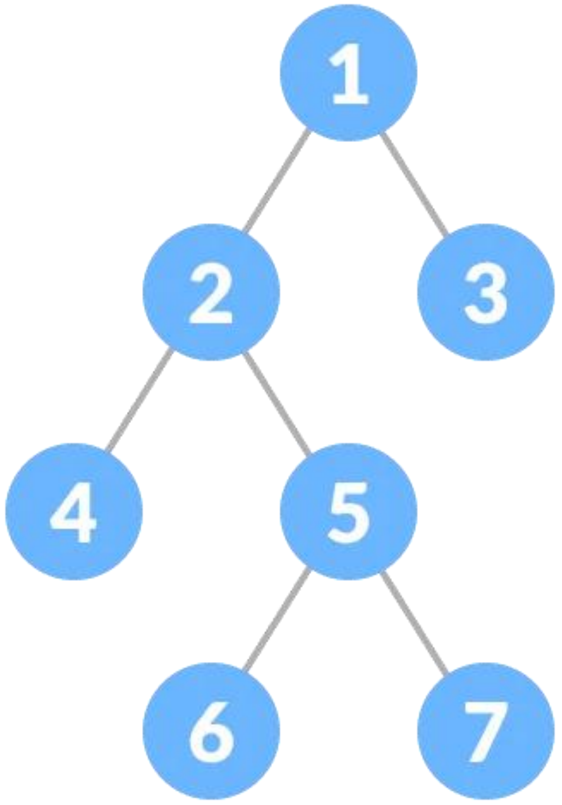
Decision Tree

- Binary tree associated with a decision process
- Internal nodes – questions with yes/no answer
- External nodes – decisions
- Example dining decision



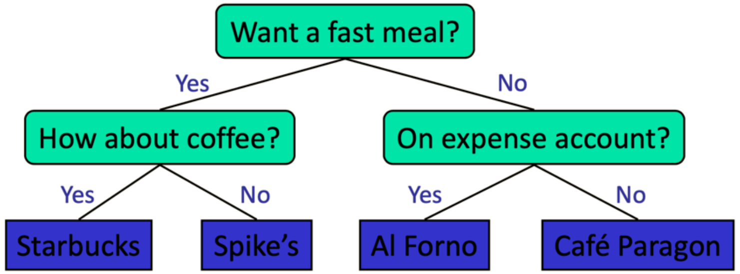
Types of BTs: Full Binary Tree

- A **full** Binary tree is a special type of binary tree in which every parent node/internal node has either **two** or **no** children.
- It is also known as a **proper** binary tree.



Properties of Proper BT

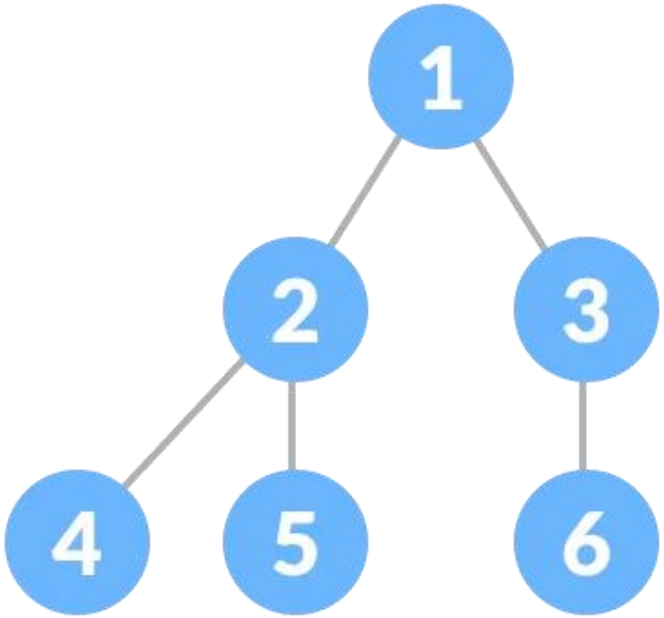
- Notations
  - **n** number of all nodes
  - **e** number of external nodes
  - *i* number of internal nodes
  - ***d*** depth
  - ***h*** height



1. The number of leaves is $i + 1$.
2. The total number of nodes is $2i + 1$.
3. The number of internal nodes is $(n - 1) / 2$.
4. The number of leaves is $(n + 1) / 2$.
5. The total number of nodes is $2e - 1$.
6. The number of internal nodes is $e - 1$.
7. The number of leaves is at most 2^{h-1}

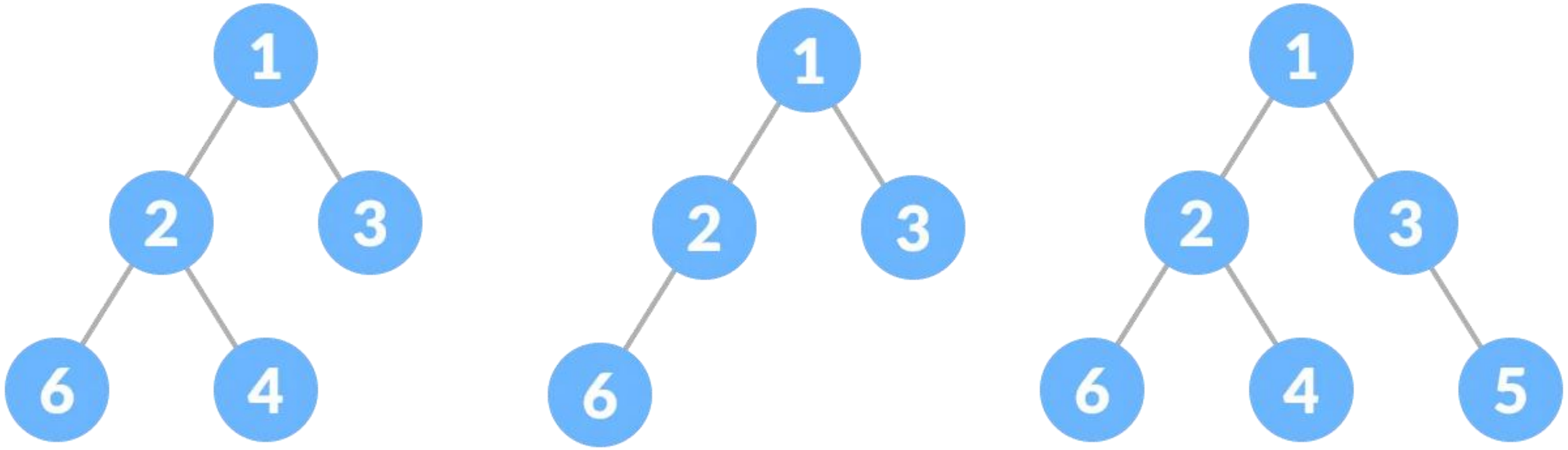
Types of BTs: Complete Binary Tree

- A complete binary tree is a binary tree in which every level, except possibly the last, is **completely filled**, and all nodes are as far left as possible.
- This means that:
  - All levels above the last level are fully filled.
  - The last level may not be fully filled, but if it has missing nodes, those nodes are only on the right side (i.e., **all leaf nodes lean to the left**).



Question!

- Which of the following options match the type of each binary search tree?



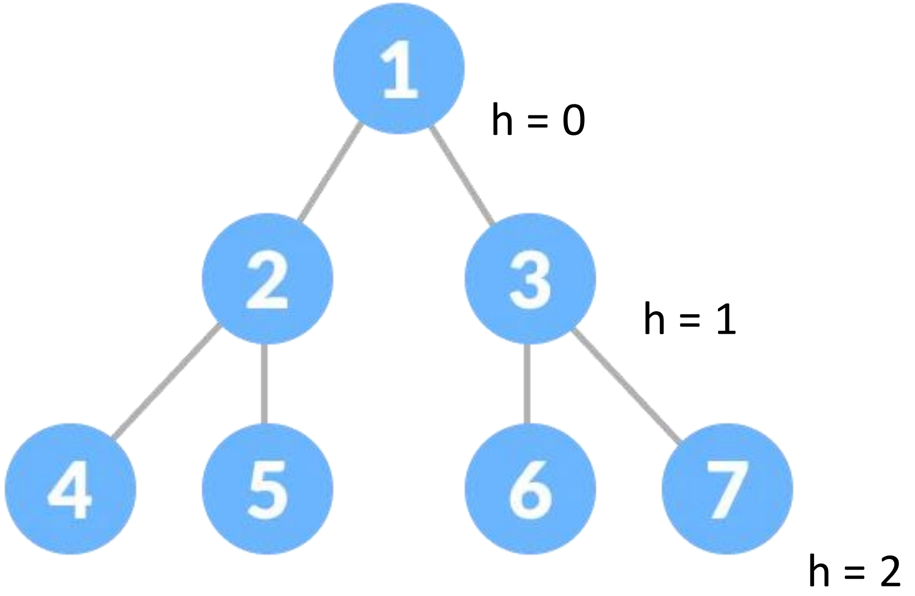
Tree 1 Tree 2 Tree 3

A) Full Binary Tree, B) Complete Binary Tree, C) Both, D) Neither

Tree 1: Both
Tree 2: Both
Tree 3: Neither

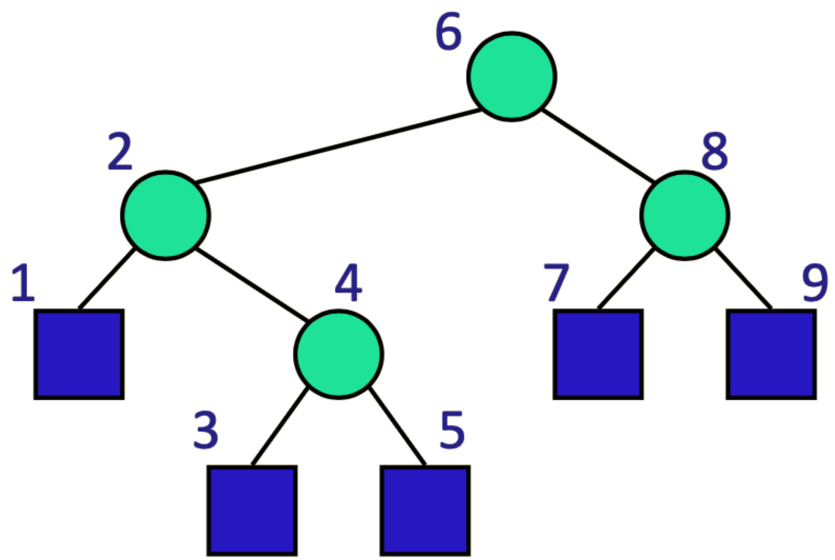
Types of BTs: Perfect Binary Tree

- A perfect binary tree is a binary tree in which every internal node has exactly **two child nodes** and all the leaf nodes are at the same level.
- A perfect binary tree of height h has $2^{h+1} - 1$ node.
- A perfect binary tree of height $\boldsymbol{h}$ has 2^{h} leaf nodes.



Inorder Traversal for Binary Trees

- **Inorder** traversal visits node after visiting left subtree but before visiting right subtree

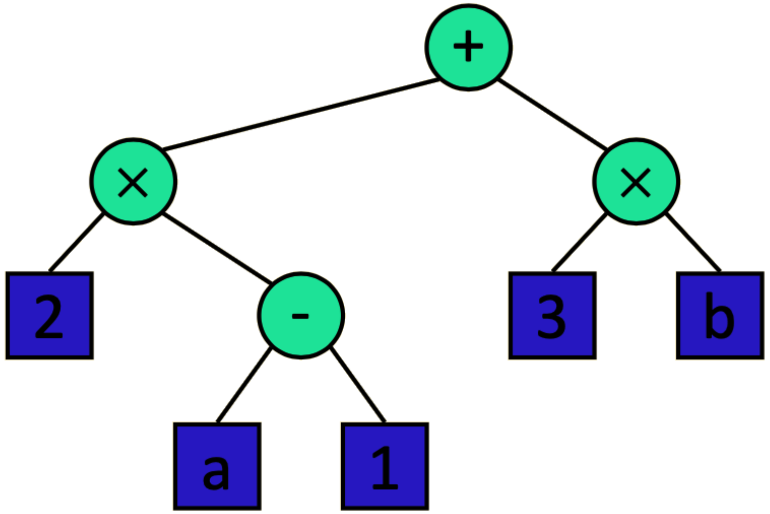



```
void inorder(Node* node) {
if (node == NULL)
return;

// Traverse left subtree
inorder(node->left);
// Visit node
printf("%d ", node->data);
// Traverse right subtree
inorder(node->right);
}
```

Print Arithmetic Expressions

- Specialization of inorder traversal
  - print (before traversing the left subtree
  - print operand or operator when visiting the node
  - print) after traversing right



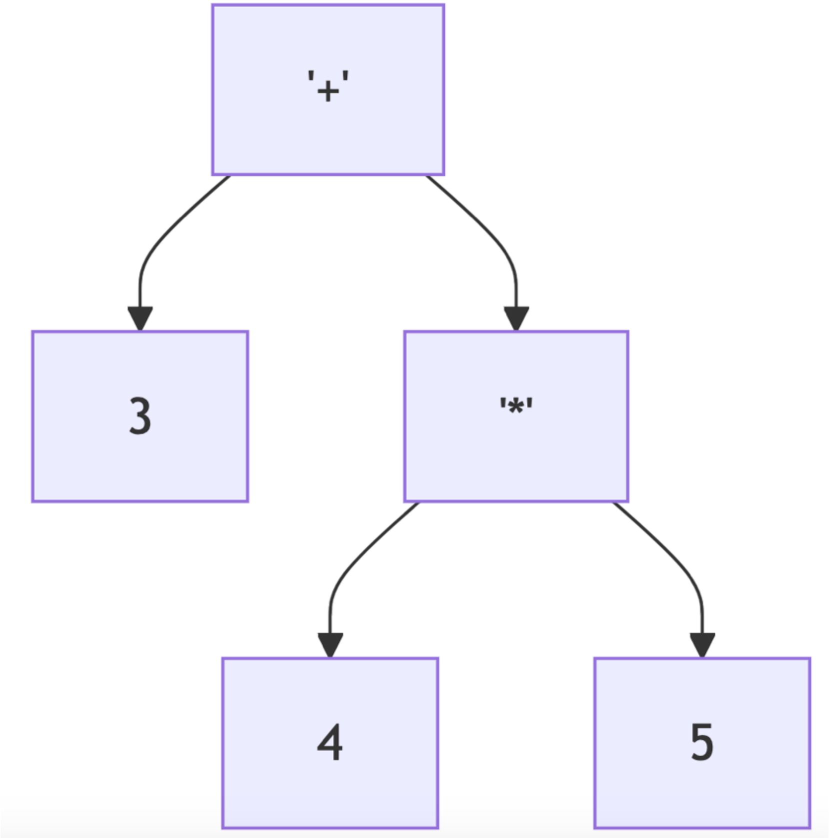

```
Algorithm printExpression(v)
    if hasLeft (v)
        print "("
        printExpression(left(v))
    print v.element()
    if hasRight(v)
        printExpression(right(v))
        print ")"
```

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

Evaluate Arithmetic Expressions

- Specialization of **postorder** traversal returns the value of a subtree
- When visiting an internal node, combine the values of the subtrees

```c
typedef struct Node {
int isExternal;
double value;
char operator;
struct Node* left;
struct Node* right;
} Node;
```



Search Operations

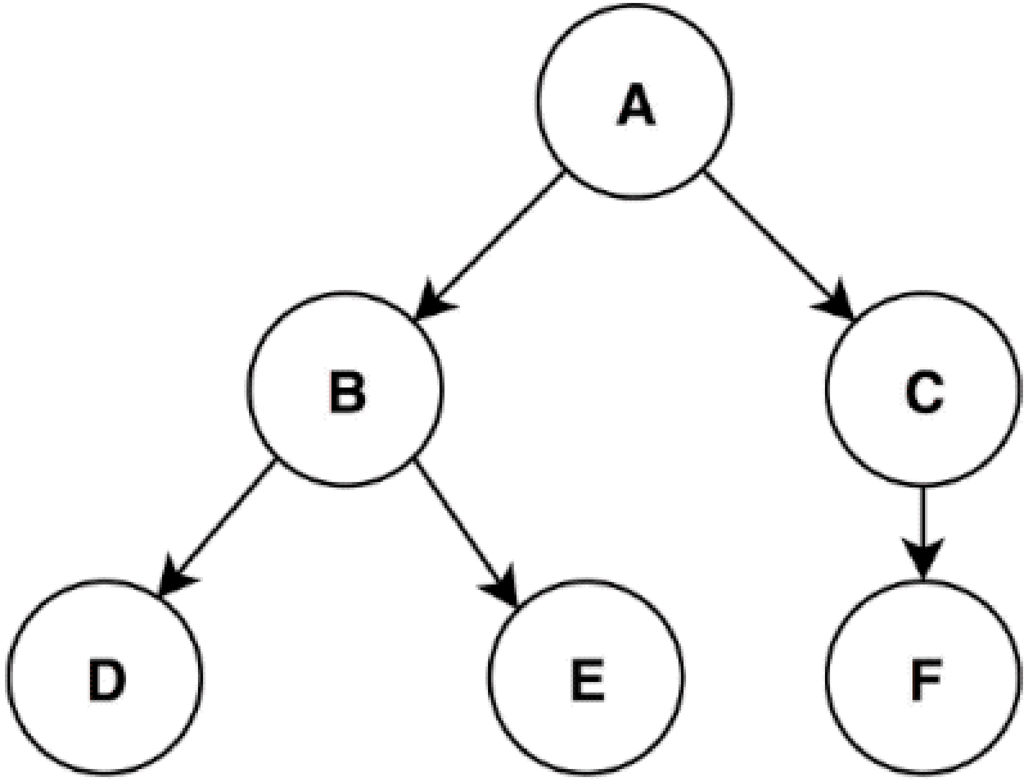
Depth-First Searching

- Searching a tree involves finding a node with a specific property matching a given search key. The search operation returns the found node or NULL if it is not found.
- The **preorder** traversal can be used to search a tree. It checks a node's data value against the search key, and if it matches, it returns the node and stops the traversal.
- This is a so-called **Depth-First Search (DFS) algorithm**.

```
NODE *search(TNODE *root, int key) {
if (root == NULL) return NULL;
if (key == root->data) return root;
else {
NODE *p = search(root->left, key);
if (p != NULL) return p;
else
return search(root->right, key); }
}
```

Depth-First Search

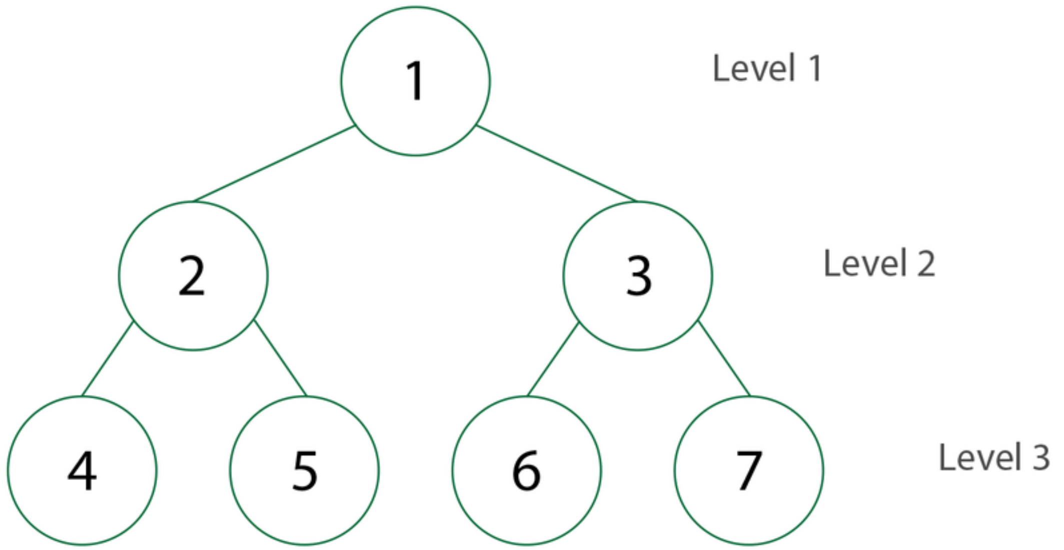
```
NODE *search(TNODE *root, int key) {
if (root == NULL) return NULL;
if (key == root->data) return root;
else {
NODE *p = search(root->left, key);
if (p != NULL) return p;
else
return search(root->right, key); }
}
```



Source: https://medium.com/@elfrmkr98/getting-started-with-trees-depth-first-search-dfs-cbc294a74bd0

Breadth-First Searching

- The **level-order traversal** (or **breadth-first** traversal) can be used to search a tree. It checks each node's data value against the search key level by level, starting from the root. If it finds a match, it returns the node and stops the traversal.
- This is a so-called Breadth-First Search (BFS) algorithm.



Breadth-First Searching

Source: https://medium.com/@elfrmkr98/getting-started-with-trees-breadth-first-search-bfs-344dc92cbd30


Thank you