CS202 – Data Structures

LECTURE-07

Trees and Binary Trees

Representation and traversal

Dr. Maryam Abdul Ghafoor
Assistant Professor
Department of Computer Science, SBASSE

Quick Recap

- Linear Data Structures Operations and Complexity
  - Insert(item): Θ(1)
  - Delete(item): Θ(1)
  - Search(item): Θ(n)

- Key Takeaway: Linear data structures are efficient for insertions and deletions but slow for searching.

Agenda

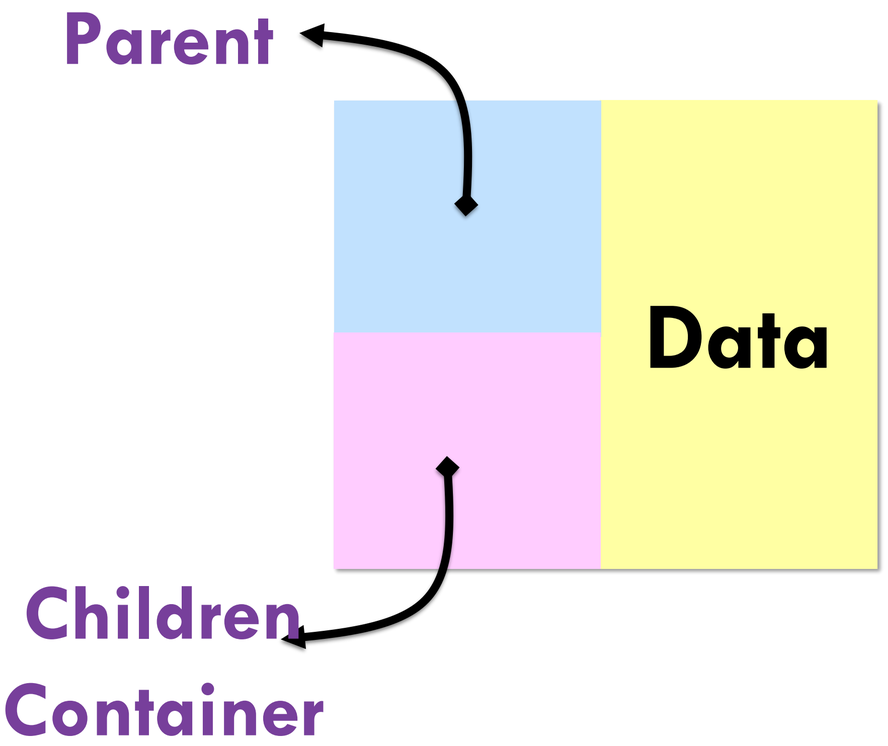
- Representing Trees
- Tree Traversals

Representing Trees

How do we represent trees where each node can have an arbitrary number of children nodes?

Representing General Trees

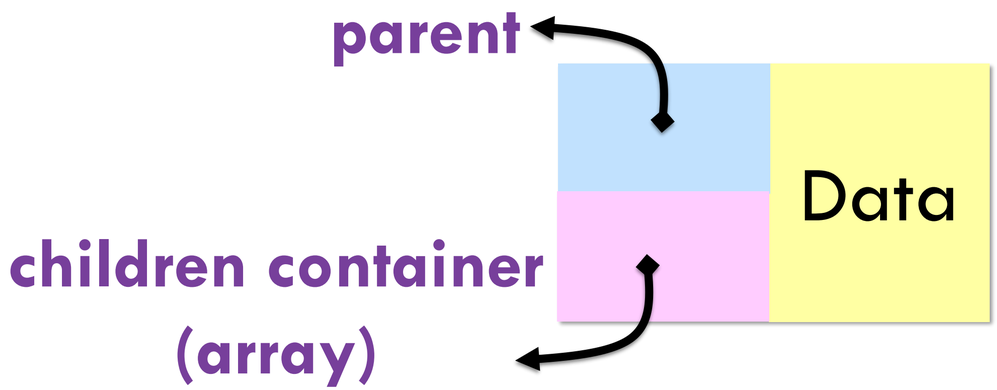
- Tree Node
  - Data
  - link to the parent
  - link to node's children
    - Array or Linked list



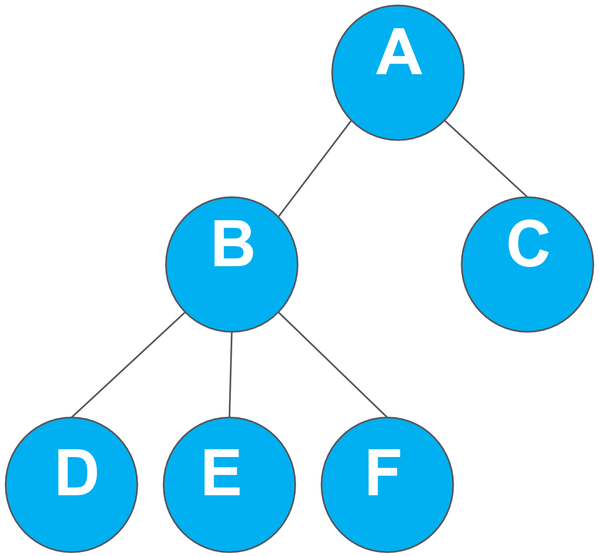
- Constraints
  - Variable number of children
  - No pre-defined maximum number of children
  - Design Decision?
    - Use **list** of children nodes

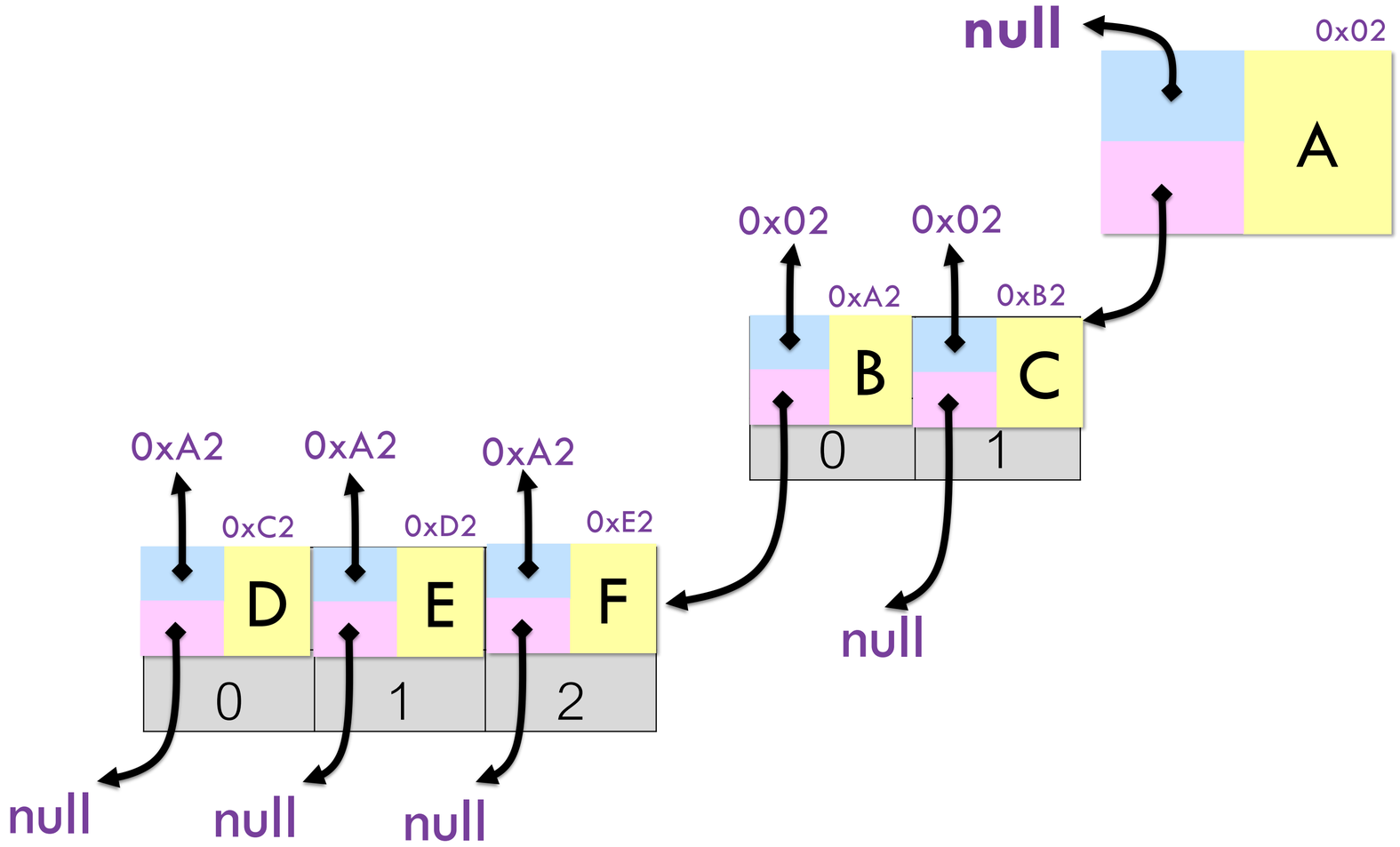
Representing General Trees - Arrays

```
class TreeNode{
public:
    TreeNode();
private:
   int data;
   TreeNode* parent;
   TreeNode** children;
/*For dynamic resizing, use vectors */
//std::vector<TreeNode*> children;
}
```



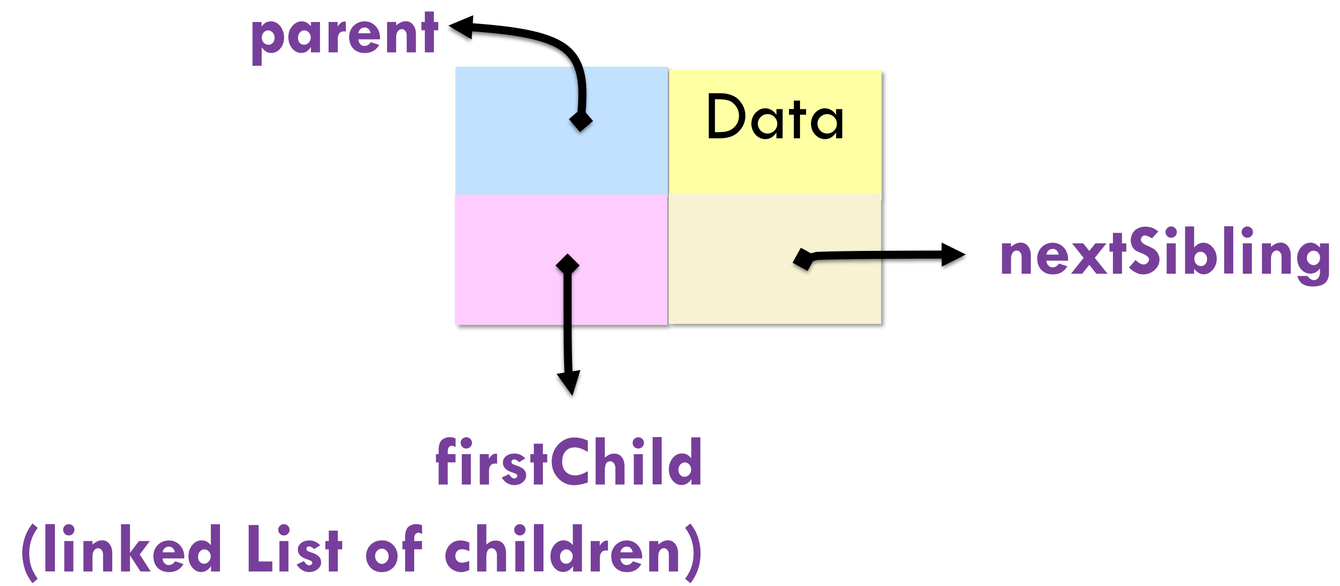
Representing General Trees – Arrays





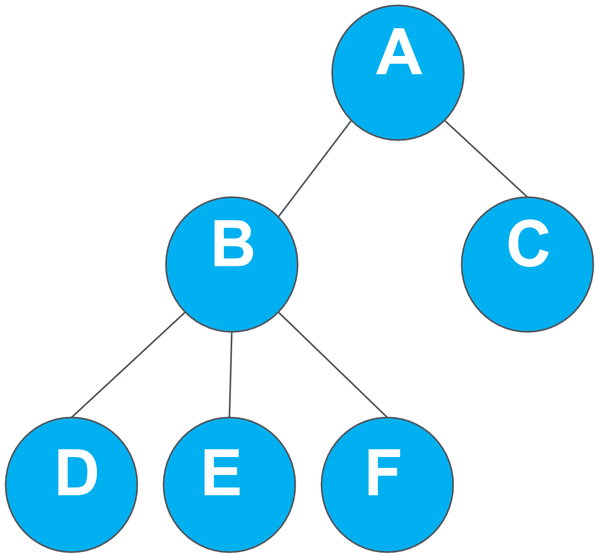
Representing General Trees – Linked List

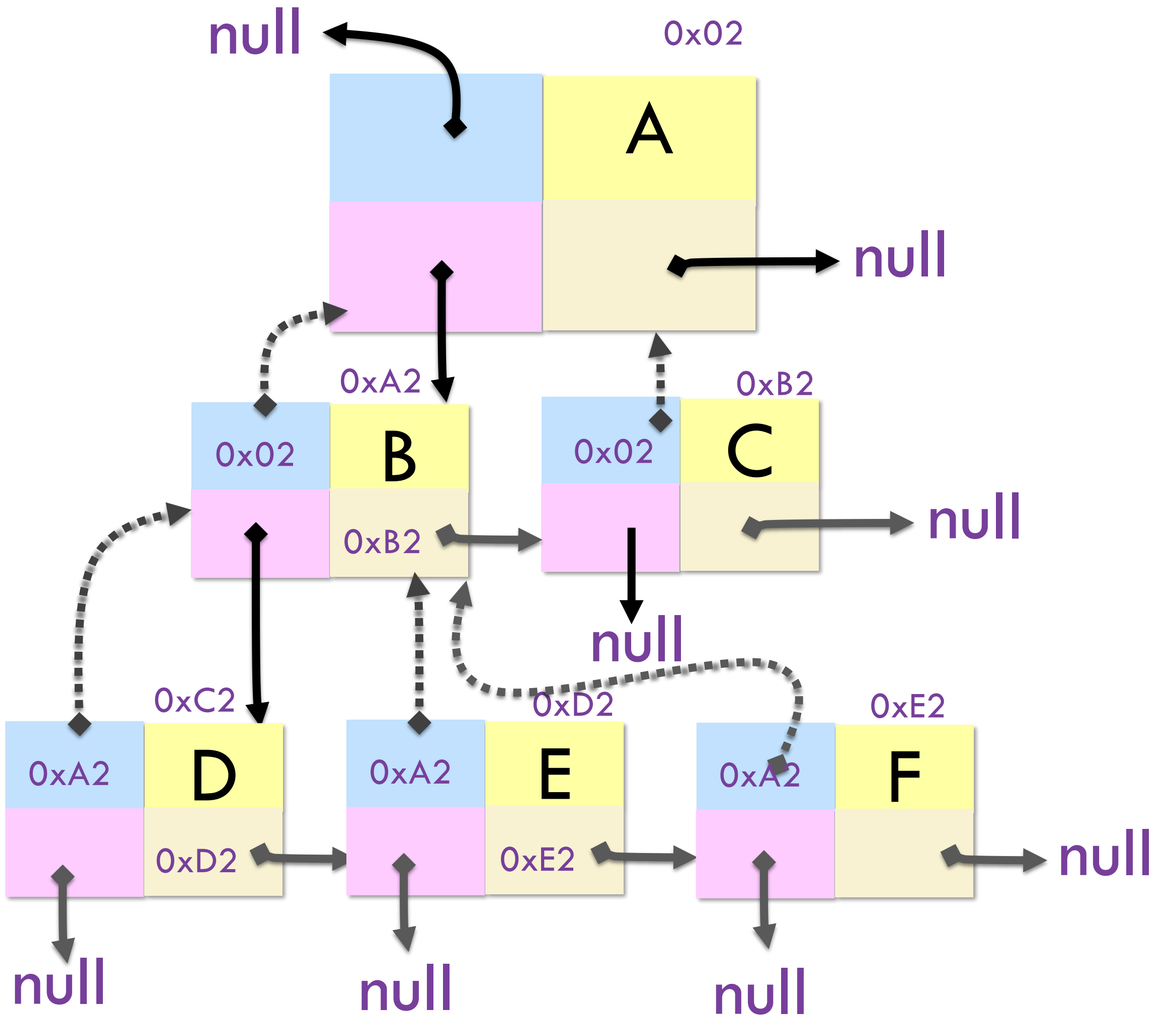
```
class TreeNode{
public:
   TreeNode();

private:
   int data;
   TreeNode*parent;
   TreeNode* firstChild; // ~head
   TreeNode* nextSibling; // ~next
};
class Tree{
   int size;
   TreeNode* root;
};
```



Representing General Trees – Linked List

One implementation strategy





Observation: Finding a child node involves doing a linear search on the children's linked list

Tree Operations

```
class Tree{
   public:
      Tree();
      int getsize();
      bool isEmpty();
      bool isRoot(TreeNode* node);
      bool isExternal(TreeNode* node);
      TreeNode* root();
      TreeNode* parent();
      TreeNode* children(TreeNode* node);
      void insert(string  data);
      TreeNode* search(int  data);

   private:
      int size; TreeNode*root;
};
```

Operation	Time Complexity
isRoot	O(1)
IsExternal	O(1)
Parent	O(1)
Size	O(1)
empty	O(1)
Root	O(1)
children	O(c)

More about trees

Depth of a node is its distance form the root

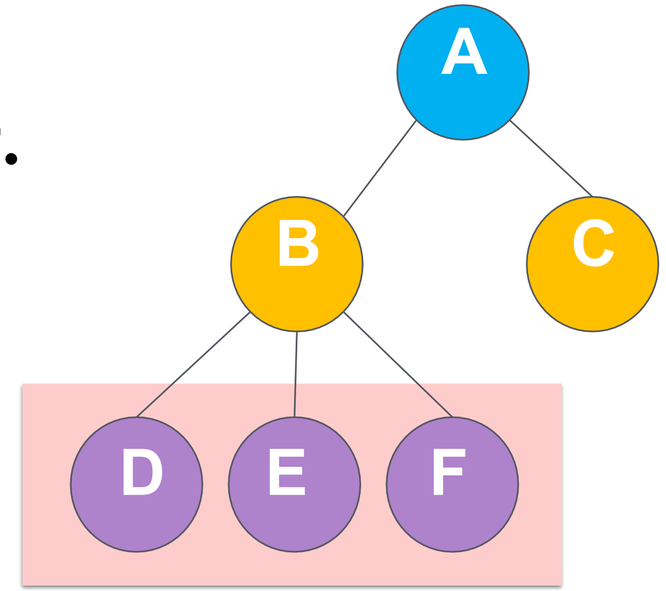
- The depth of the root 0, then depth of a node in a tree can be computed using following relation
  - Depth(node) = 1+ depth(node → parent)

Height of the node is the distance of a node from its deepest descendant. Height of the null node is -1.

- Height(root) = height of the tree
- Height(node) = 1 + max height of its descendants
  = 1 + max (height(node→left), height(node → right)

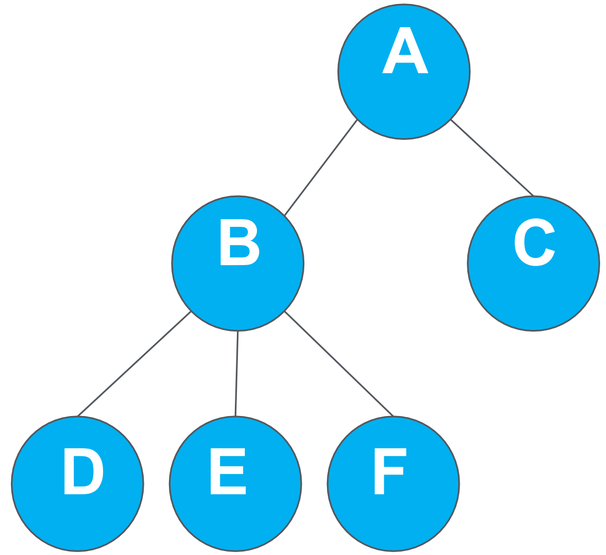
More about trees

- **Root Node**
  - The **topmost node** of a tree. It has no parent.
  - Every tree has exactly **one root.**
- **External Node (Leaf Node)**
  - A node with **no children.**
  - It is at the "end" of a branch.
- **Internal Node**
  - A node with **at least one child.** (in above: A, B and C)
  - Lies "inside" the tree structure, not at the edge.
  - **Includes the root (unless the tree has only one node)**



What does this code do?

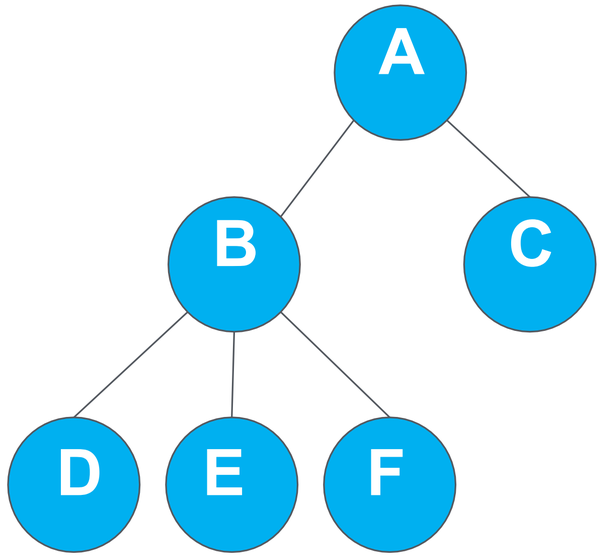
```
int mystery(Node *node) {
if (node.isRoot())
    return 0;
else
    return 1 + mystery(node.parent());
}
```



What will it return, if node E is passed to it?

More Operations on Trees

Operation	Time Complexity
Depth	O(n)
Height(Tree)	O(n)



Assume **n** is the total number of node.

What is the **average time complexity** of these operations?

Why Traversals are important?

Traversal is a manner of visiting each node in a tree once.

Traversals are …

- …used for **searching, sorting, expression evaluation, and tree modifications.**

- …essential for **constructing and interpreting hierarchical data structures.**

Tree Traversals

- Preorder Traversal → root – left – right
- Post order traversal → left – right – root
- In-order Traversal → left – root – right
- Level order traversals → level by level

More Applications of Traversal

- Tree Duplication or Tree Cloning
- Tree Deletion
- Expression Evaluation
- Sorting over trees in BST
- Distance of a node from given root (#edges)

Traversals

Pre-order: D B A H C F E G

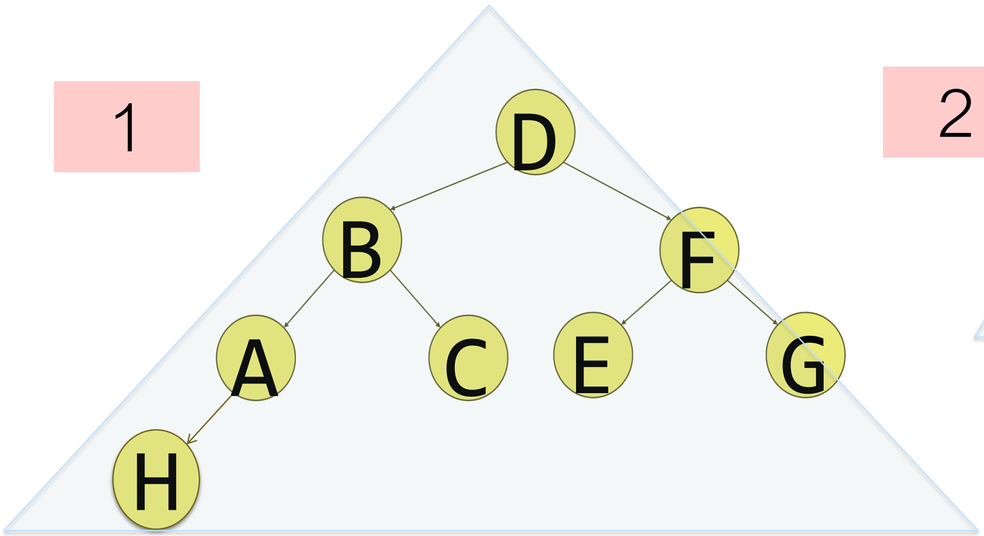
Post-order: H A C B E G F D

In-order: H A B C D E F G

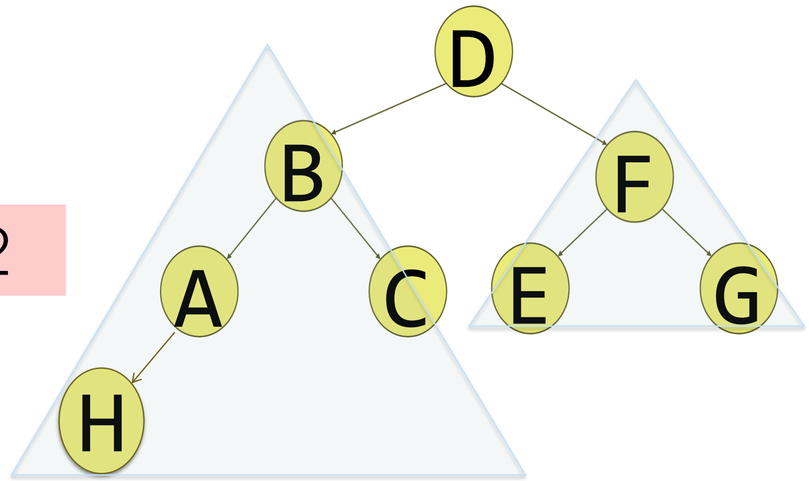
Level-order: D B F A C E G H

Trees are recursive: each tree has a left subtree and a right subtree

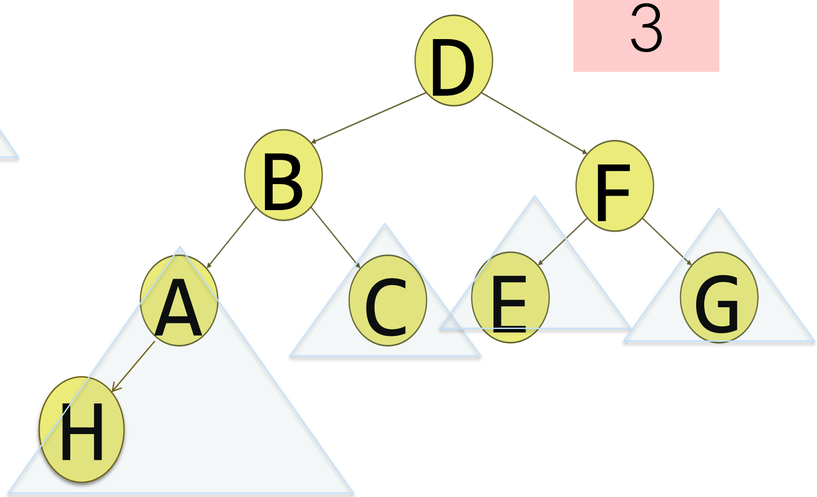
1



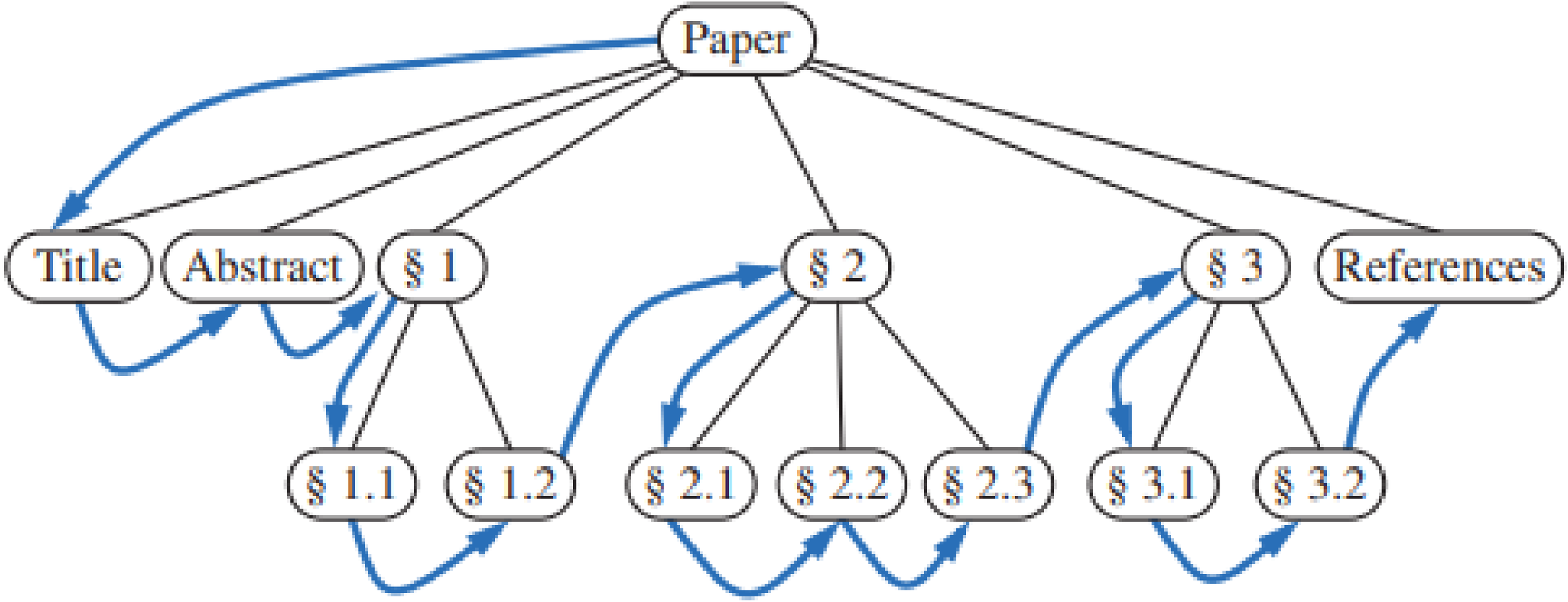
2



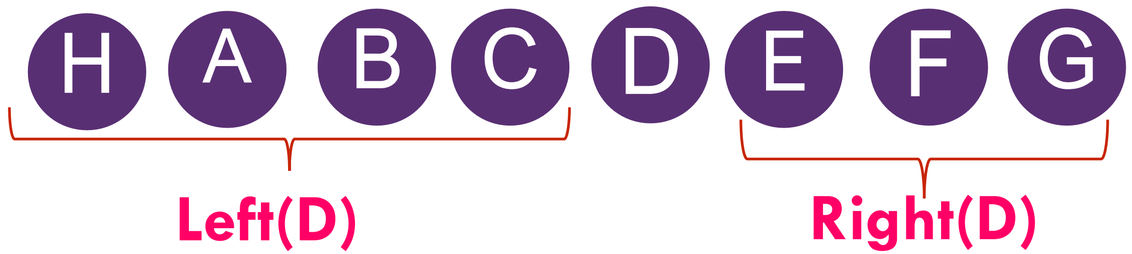
3

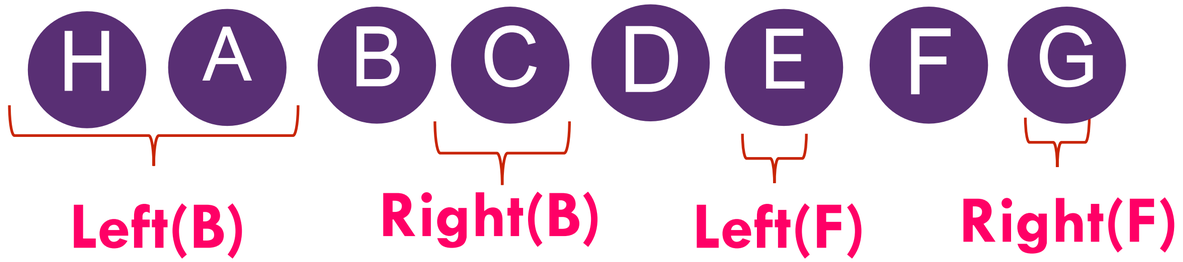


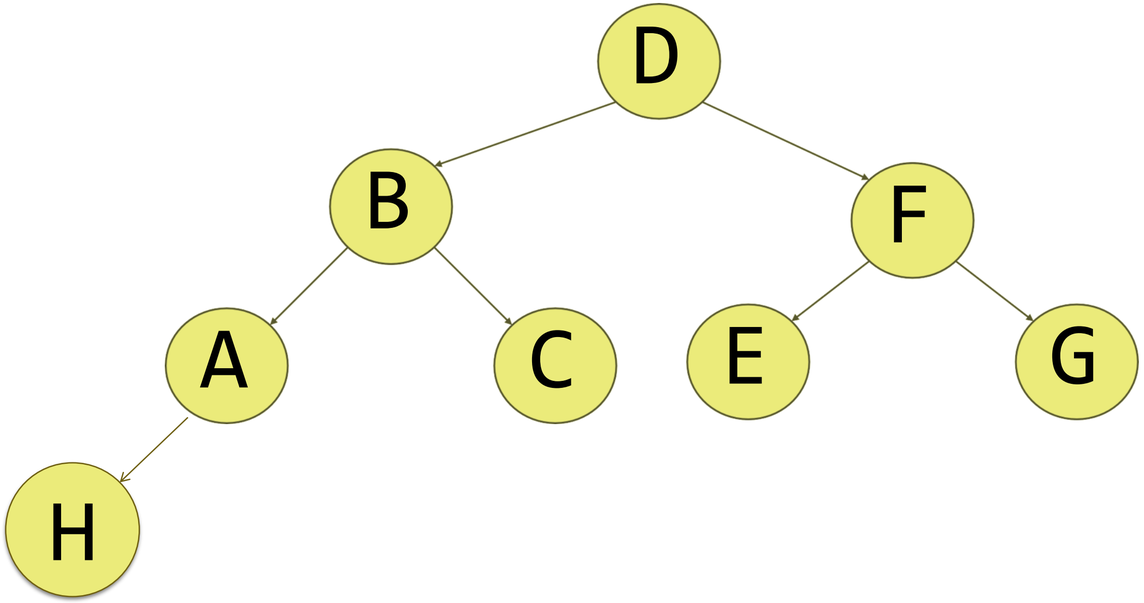
Preorder Traversal – Example



Inorder Traversal







- For every node X
  - Nodes in X.left are before X
  - Nodes in X.right are after X

```
traverse(X)
  traverse(X.left)
  output(X)
  traverse(X.right)
```

Traversals – Time Complexity

- Total number of recursive calls $\in \Theta(N)$, where N is the number of nodes in the tree
- Each call itself takes $\Theta(1)$ operations
- Total time taken by a traversal: $\Theta(N)$

Note: *In terms of actual time taken, recursive code is often slower than iterative code due to large number of function calls and the need to maintain activation records on the program call stack. However, recrusive codes are often easier to write and result in succint code. In case of trees, it is fine to use them if the tree depth isn't too large.*

Questions