Data Structures - Binary Trees
and Operations on Binary Trees

MCS 275
A **binary tree** is a finite set of elements. It can be
- empty
- partitioned into three disjoint subsets

The structure of the three disjoint subsets is as follows:
- a single element, called the root of the tree
- subsets two and three are themselves binary trees, called
  - **left** subtree and **right** subtree
  - that is, left and right subtrees of the original tree
- each element in a binary tree is called a **node**

**Figure**: A binary tree
A binary tree has a specific structure

- each node can have at most two branches

Figure: This is still a binary tree
A **binary tree** has a specific structure:
- each node can have at most two branches

**Figure:** Not a binary tree: fails the **binary** condition
Not a Binary Trees

A **binary tree** has a specific structure

- each node can have at most two branches
- trees do not contain loops

Figure: Not a binary tree: fails the **tree** condition. Trees do not contain loops!
nodes $C, D, E, E$ are called **leaves**

a strictly binary tree with $n$ leaves has $2n - 1$ nodes

**Figure:** A complete binary tree
Nodes in a tree can be categorized by their level:

- the root (R) has level 0
- the nodes A and B have level 1
- leaves (C, D, E, F) form the last level, in this case 2
- the depth of a binary tree is the maximum of any leaf in the tree

Figure: A complete binary tree of depth 2
A complete Binary Tree

If a binary tree contains \( m \) nodes at level \( l \)
- it contains at most \( 2m \) nodes at level \( l + 1 \)

Furthermore,
- a binary tree can contain at most \( 2^l \) nodes at level \( l \)
- the total number of nodes in a complete binary tree of depth \( d \), denoted \( TN \), equals the sum of nodes at each level between 0 and \( d \)

\[
TN = 2^0 + 2^1 + \cdots + 2^d = \sum_{k=0}^{d} 2^k
\]

![Binary Tree Diagram]

**Figure:** A complete binary tree of depth 2 with \( 2^0 + 2^1 + 2^2 = 7 \) nodes.
A common operation on a **binary tree** is **traversal**. That is,

- nodes in the tree often contain some useful information
- traverse (pass) through the tree
- we may want to print the content of each node
- we may want to process the data contained in the nodes in some other way

In case of lists: \([1,2,43,23,4,56,6,4,54,43,17,4,2]\)

- traversal is straight forward - we traverse it in a linear fashion
- we loop through the list and visit each element
- we may want to start at a specific index
- and may end at specific index
- but in general we visit all elements from first to last within a specified range
- there is an **obvious** or **natural** way to traverse a list
Operations on Binary Trees: Traversal

Figure: Traversal of a binary tree is not obvious.

For a tree or a binary tree:

- there is no natural or obvious way of how to traverse
Operations on Binary Trees: Traversal

Since there are no natural ways to traverse a binary tree:

- we will define three methods of traversal
- traversal methods will be recursive
- many operations on binary trees are recursive

The three methods of traversal are

- preorder (depth-first order)
- inorder (symmetric order)
- postorder
Operations on Binary Trees: Preorder Traversal

Recursive traversal in **preorder** performs the following operations:

1. visits the root node (read/write)
2. traverses the left subtree in preorder (recursive call)
3. traverses the right subtree in preorder (recursive call)
Recursive traversal in **inorder** performs the following operations:

1. traverses the left subtree in inorder (recursive call)
2. visits the root node (read/write)
3. traverses the right subtree in inorder (recursive call)
Recursive traversal in **postorder** performs the following operations:

1. traverses the left subtree in postorder (recursive call)
2. traverses the right subtree in postorder (recursive call)
3. visits the root node (read/write)
Operations on Binary Trees: Preorder Traversal

1. visit the root node (read/write)
2. traverse the left subtree in preorder (recursive call)
3. traverse the right subtree in preorder (recursive call)

Figure: preorder: ABDGCEHIF
Operations on Binary Trees: Inorder Traversal

1. traverse the left subtree in inorder (recursive call)
2. visit the root node (read/write)
3. traverse the right subtree in inorder (recursive call)

Figure: inorder: DGBAHEICF
Operations on Binary Trees: Postorder Traversal

1. traverse the left subtree in postorder (recursive call)
2. traverse the right subtree in postorder (recursive call)
3. visit the root node (read/write)

Figure: postorder: GDBHIEFCA
Operations on Binary Trees: Preorder Traversal

1. visit the root node (read/write)
2. traverse the left subtree in preorder (recursive call)
3. traverse the right subtree in preorder (recursive call)

![Diagram of a binary tree with nodes A, B, C, D, E, F, G, H, I, J, K, L.]

Figure:
Operations on Binary Trees: Preorder Traversal

1. visit the root node (read/write)
2. traverse the left subtree in preorder (recursive call)
3. traverse the right subtree in preorder (recursive call)

Figure: preorder: ABCEIFJDGHKL
Operations on Binary Trees: Inorder Traversal

1. traverse the left subtree in inorder (recursive call)
2. visit the root node (read/write)
3. traverse the right subtree in inorder (recursive call)

Figure:
Operations on Binary Trees: Inorder Traversal

1. Traverse the left subtree in inorder (recursive call)
2. Visit the root node (read/write)
3. Traverse the right subtree in inorder (recursive call)

Figure: inorder: EICFJBGDKHLA
Operations on Binary Trees: Postorder Traversal

1. traverse the left subtree in postorder (recursive call)
2. traverse the right subtree in postorder (recursive call)
3. visit the root node (read/write)

Figure:
1. traverse the left subtree in postorder (recursive call)
2. traverse the right subtree in postorder (recursive call)
3. visit the root node (read/write)

Figure: postorder: IEJFCGKLHDBA