Data Structure

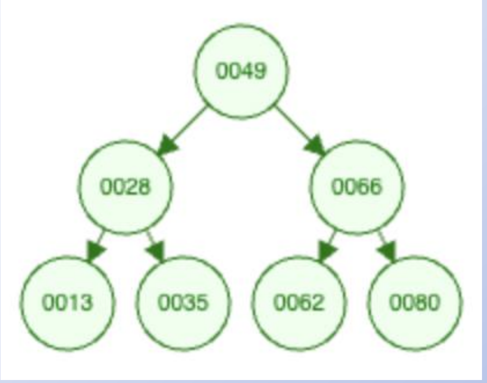
Chapter 6

Tree and Binary Trees

Lecture Contents

- Trees Data Structure
- Binary Tree
- Array representation of Binary tree
- Completed tree
- Linked list representation of Binary tree
- Binary tree traversal

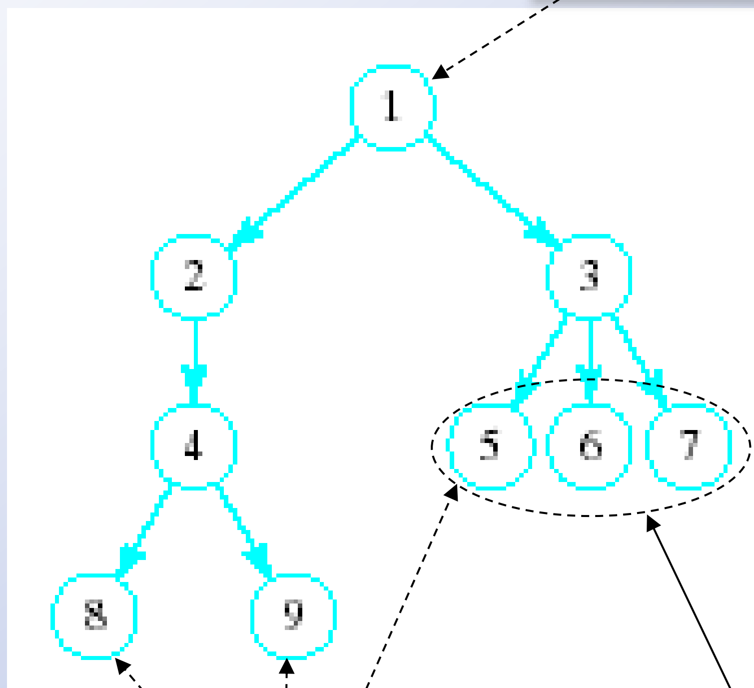
Trees



- A data structure which consists of
  - A finite set of elements called **nodes** or **vertices**
  - A finite set of **directed arcs** which connect pairs of nodes
- If the tree is nonempty
  - One of the nodes (the **root**) has no incoming arc
  - Every other node can be reached by following a unique sequence **(path)** of consecutive arcs

Tree terminologies

Root node

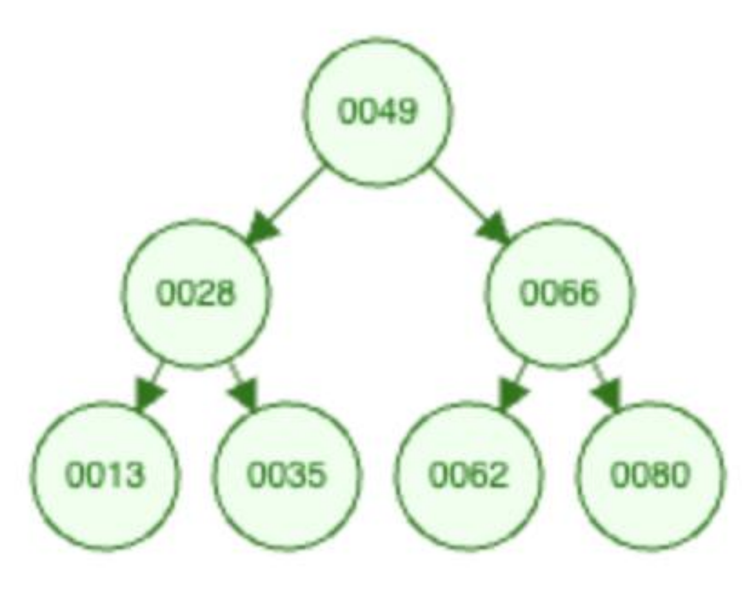


- Children of the parent (3)
- Siblings to each other

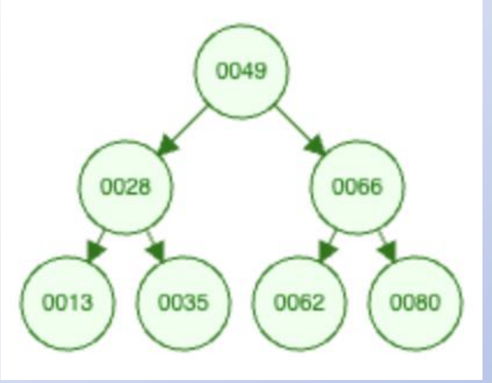
Leaf nodes

Binary Tree

- Redraw the previous structure so that it has a treelike shape – a binary tree

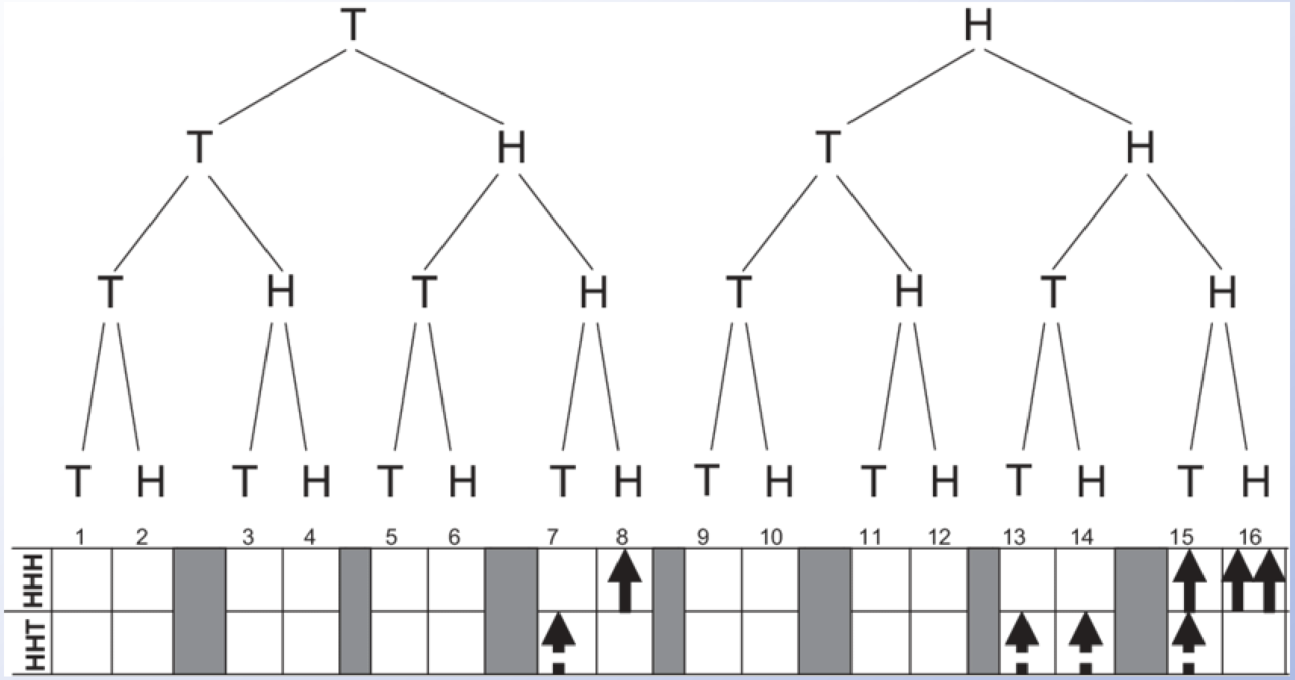


Binary Trees



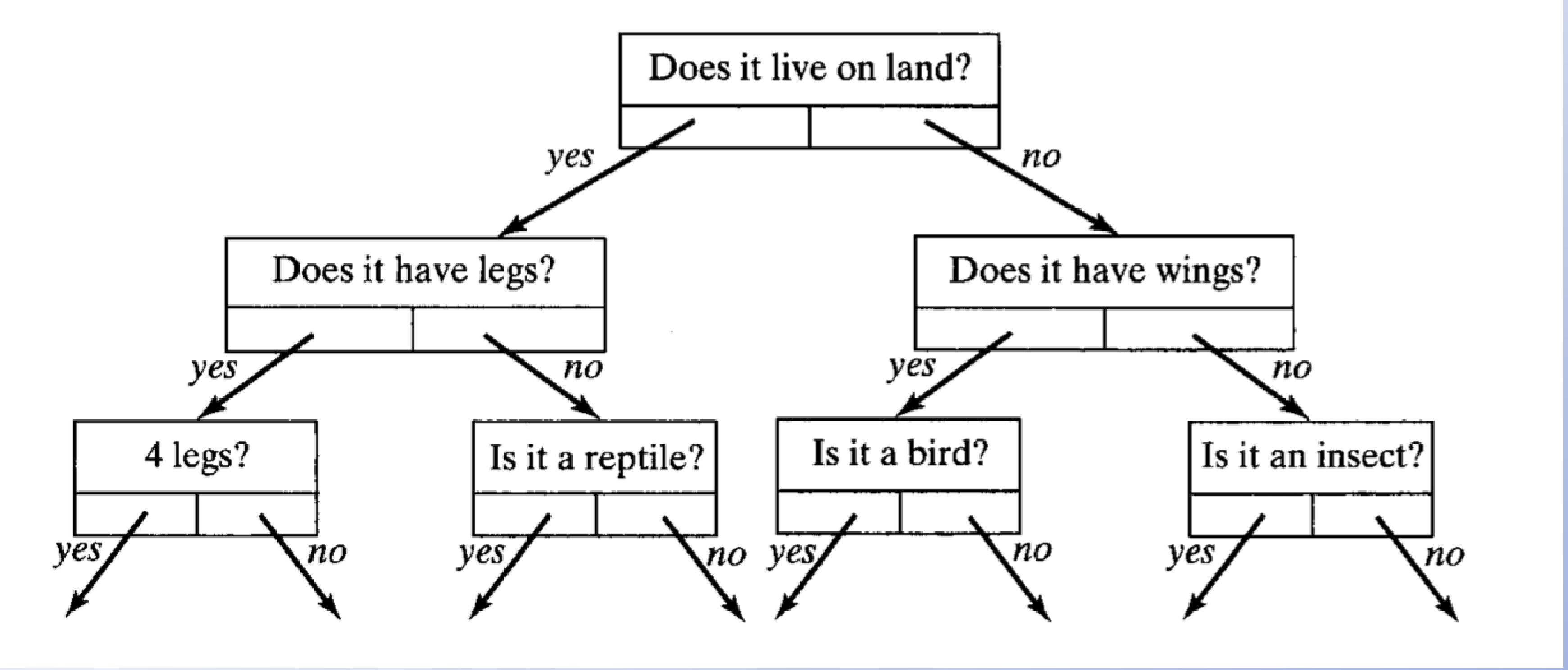
- Each node has at most two children
- Useful in modeling processes where
  - A comparison or experiment has exactly two possible outcomes
  - The test is performed repeatedly
- Example
  - Multiple coin tosses
  - Encoding/decoding messages in dots and dashes such as *Mores code*

Example: Multiple coin tosses



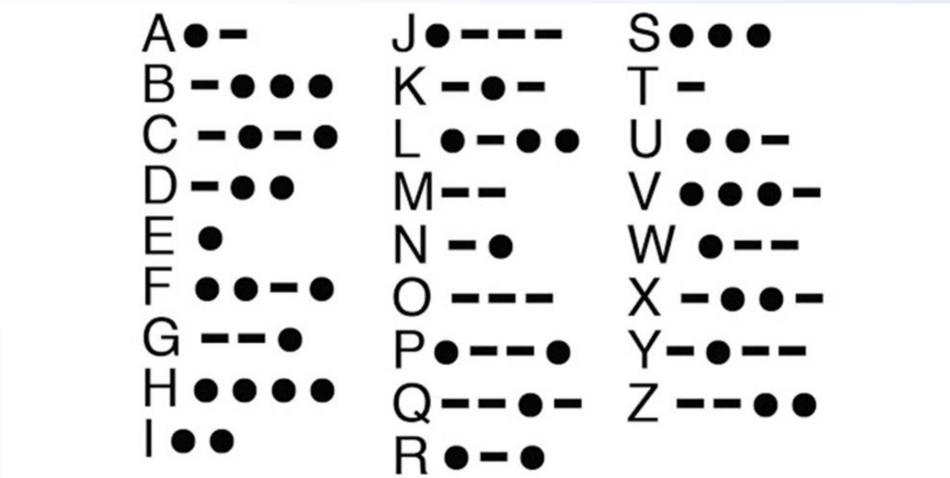
Each path from root to one of leaf nodes corresponds to a particular sequence of outcomes, such as HTH, a head followed by a tail followed by another head. Figure shows a probability tree indicating the 16 possible outcomes of a sequence of four-coin tosses.

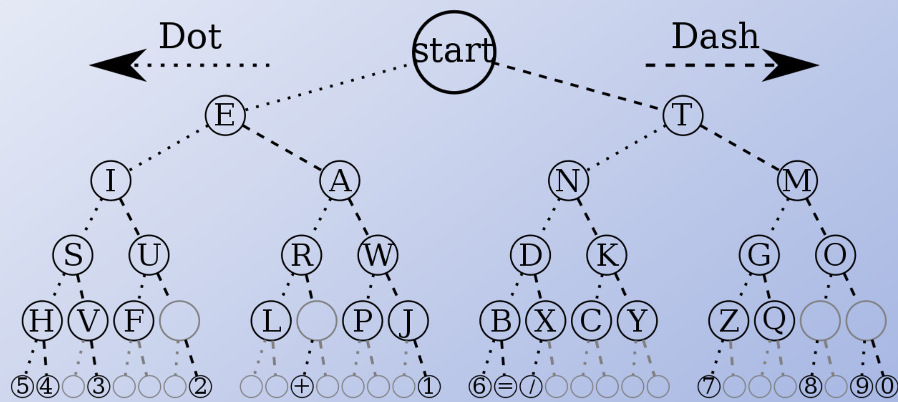
Example: Decision trees



each node contains information that requires a yes-no decision between its two subtrees. For example, the diagram, might be part of a decision tree in an *animal-guessing game* used to develop knowledge bases for programs called **expert systems**.

Example: Encoding/decoding Mores code



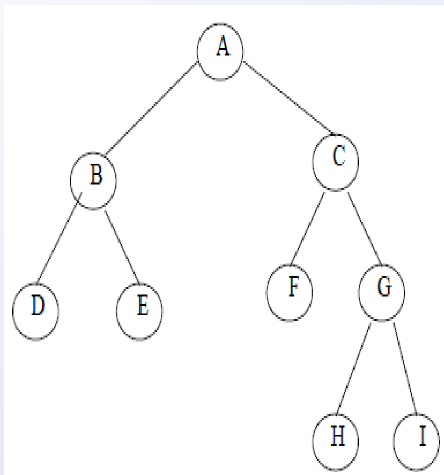


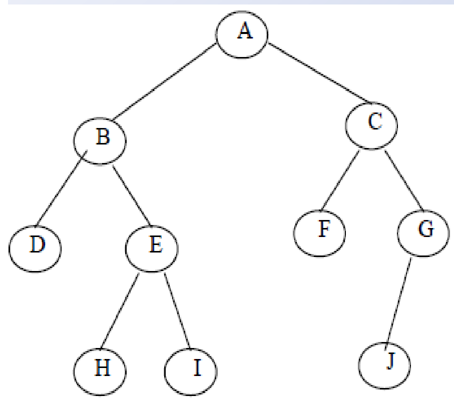
The sequence of dots and dashes labeling a path from the root to a particular node corresponds to the Morse code for that character; for example:

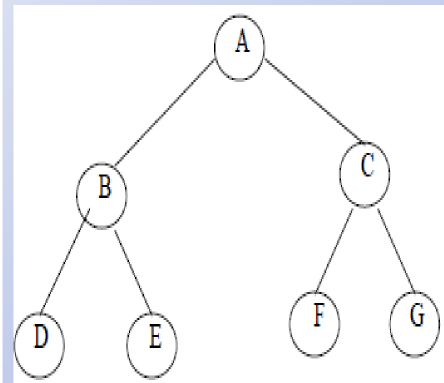
.. is the code for I and,

-. is the code for N.

Complete Binary Tree

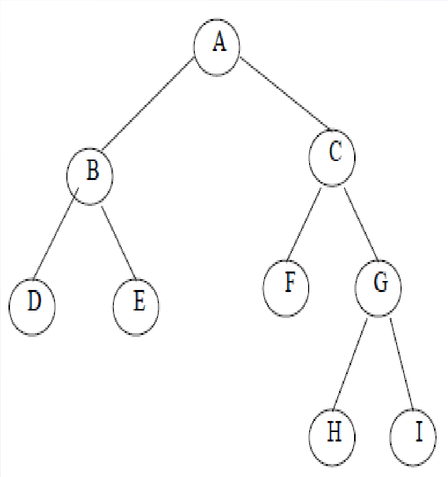


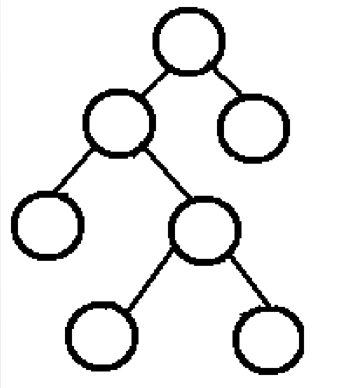


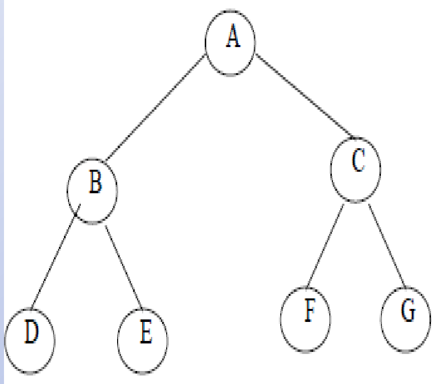


- a binary tree in which every level, except possibly the last, is completely filled- or has 2^L node.

Full (Strictly) Binary Tree

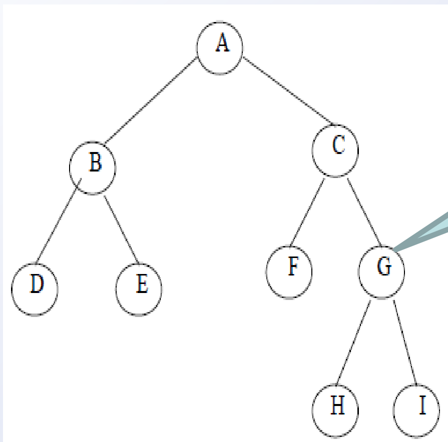




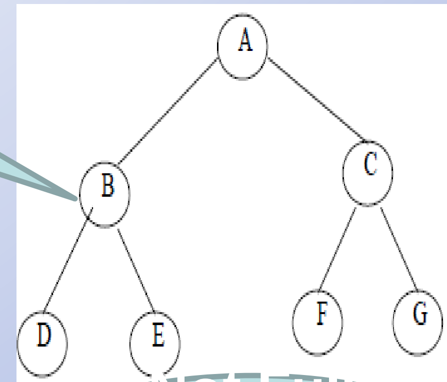


- A binary tree in which every node other than the leaves has exactly two children.

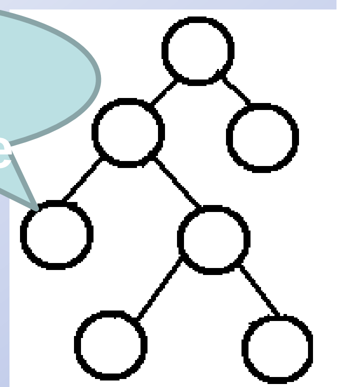
Binary Tree

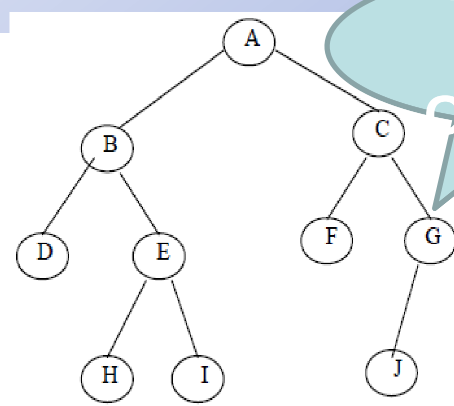


Full and Complete



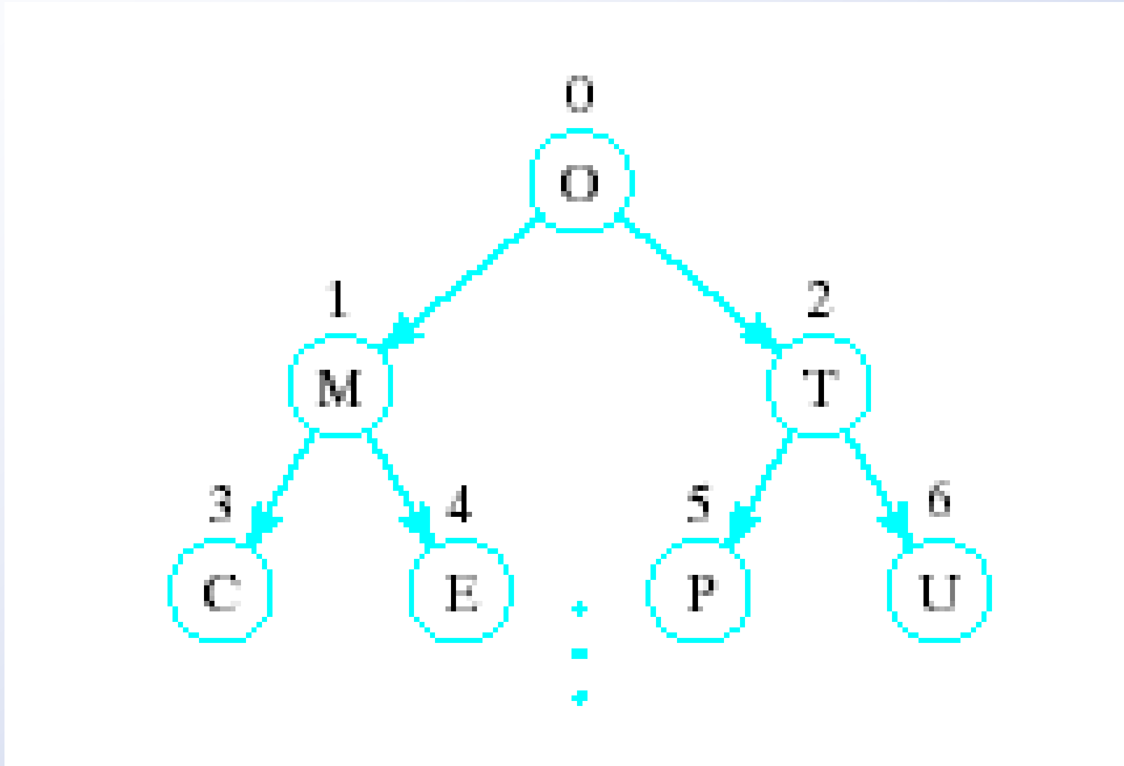
Full but not Complete





Not Full but Complete

Array Representation of Binary Trees

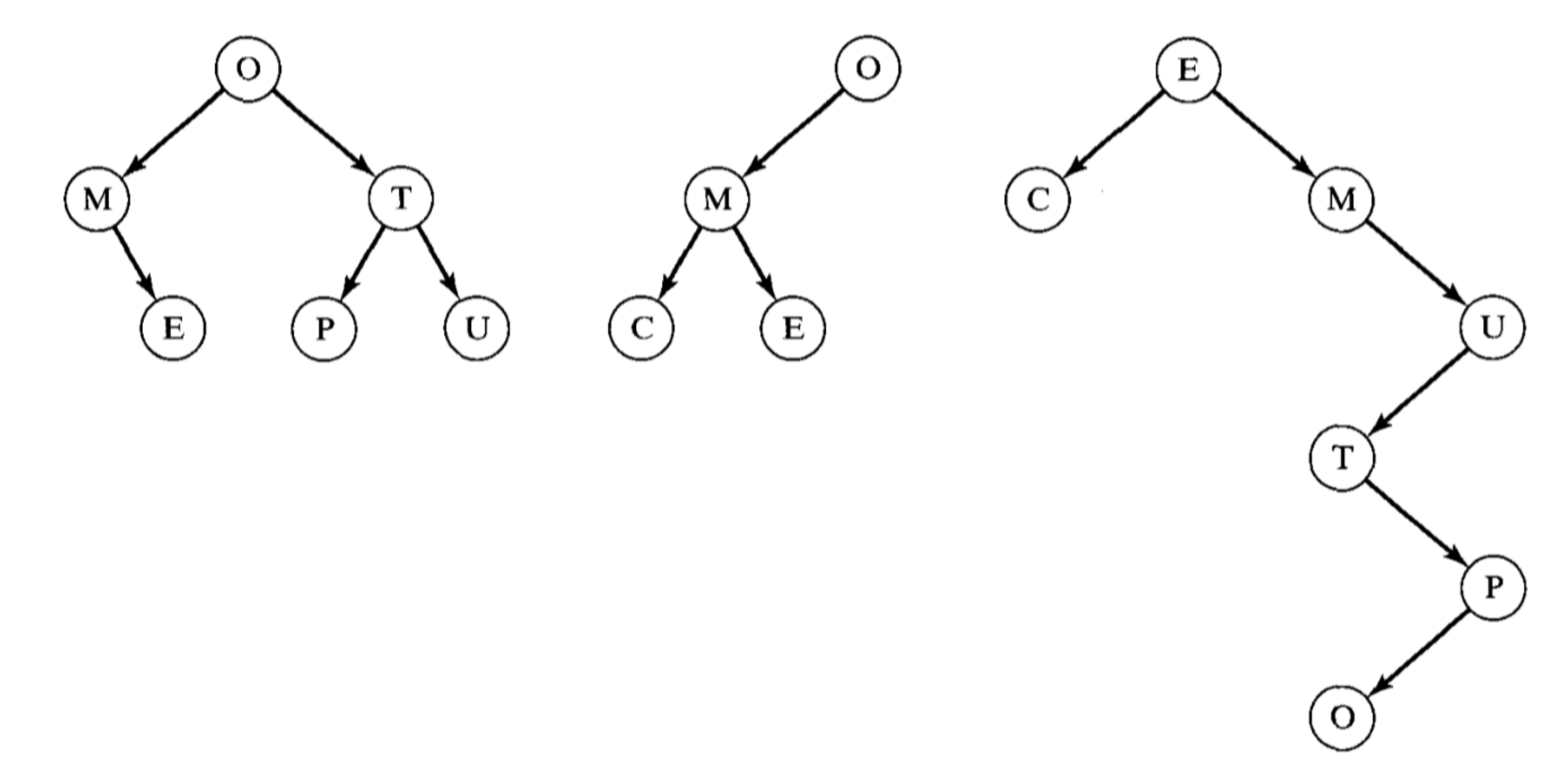


i	0	1	2	3	4	5	6	...
t[*i*]	O	M	T	C	E	P	U	...

- Store the i^{th} node in the i^{th} location of the array

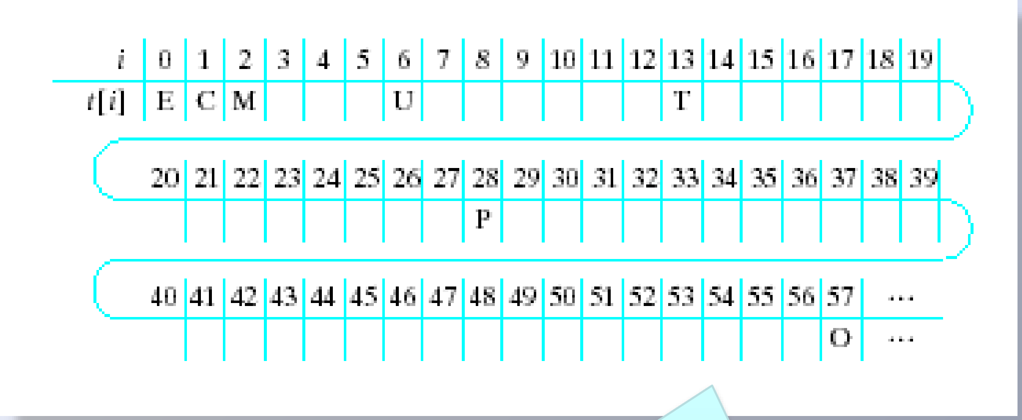
Array Representation of Binary Trees

Not-Completed *trees*



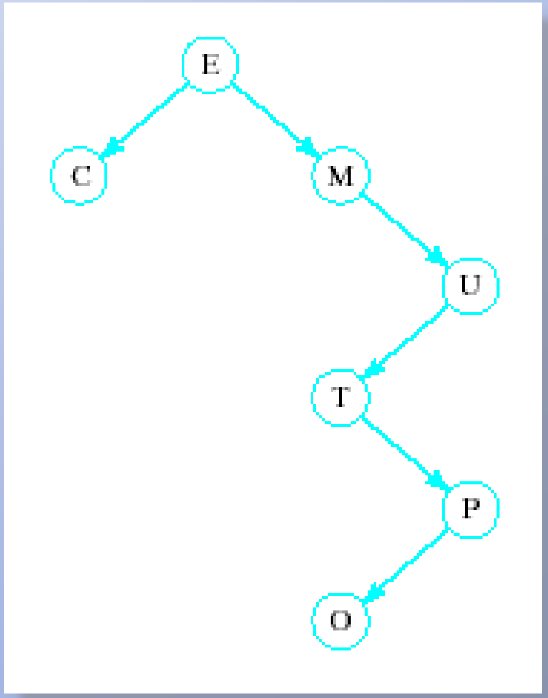
Array Representation of Binary Trees

- Works OK for complete trees, not for sparse trees



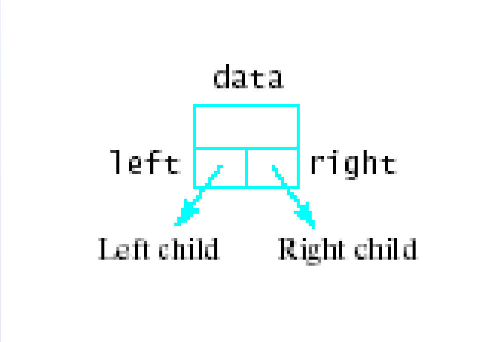
58 array elements to host 7 tree nodes

Waste of space

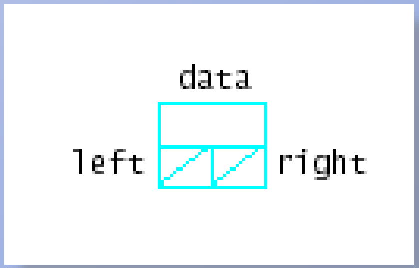


Linked List Representation of Binary Trees

- Uses space more efficiently

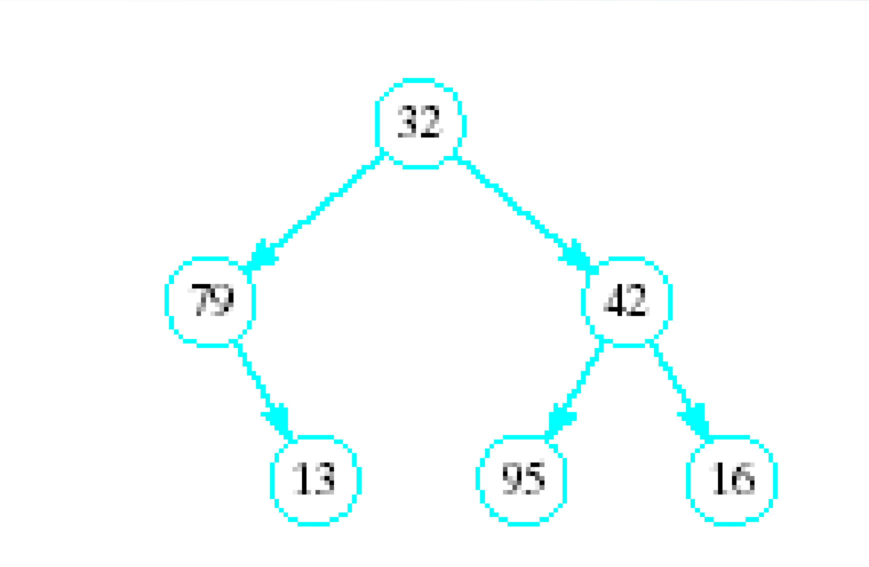


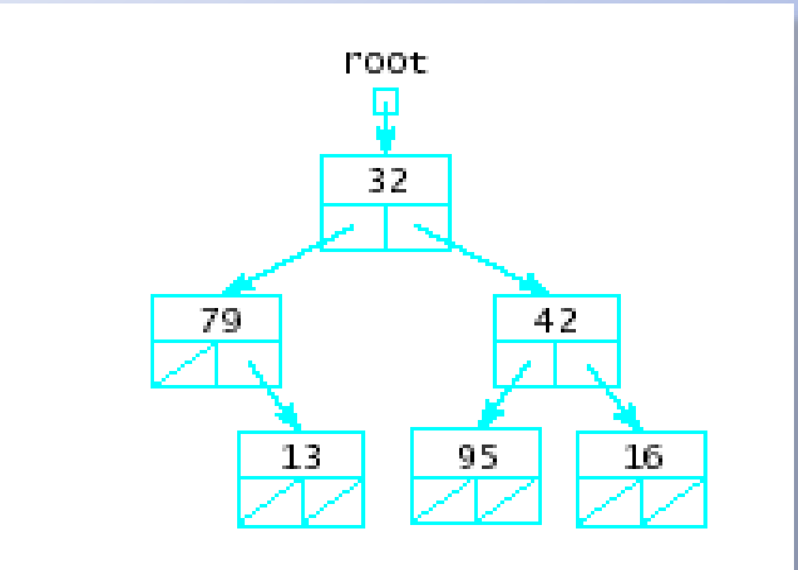
- Provides additional flexibility
- Each node has two links
  - one to the left child of the node
  - one to the right child of the node
  - if no child node exists for a node, the link is set to NULL



Linked List Representation of Binary Trees

Example





Linked Binary Tree representation

```
class BinNode
{
 public:
  DataType data;
  BinNode * left;
  BinNode * right;

  // BinNode constructors
  // Default -- data is default DataType value;
  //         -- both links are null
  BinNode()
  : left(0), right(0)
  {}

  // Explicit Value -- data part contains item;
  //                -- both links are null
  BinNode(DataType item)
  : data(item), left(0), right(0)
  {}
};// end of class BinNode declaration
```

Binary Trees as Recursive Data Structures

- A binary tree is either empty …

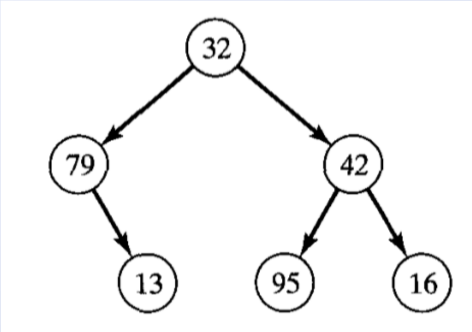
  Anchor

  or

- Consists of
  - a node called the root
  - root has pointers to two disjoint binary (sub)<u>trees</u> called …
    - ✓ right (sub)<u>tree</u>
    - ✓ left (sub)<u>tree</u>

  Inductive step

Tree Traversal is Recursive



If the binary tree is empty, then do nothing

Else

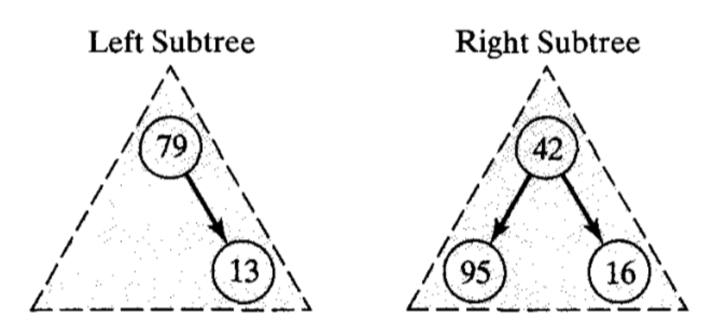
N: Visit the *root*, process data

L: Traverse the *left subtree*

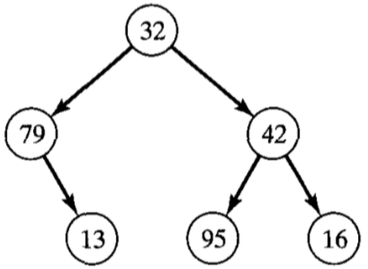
R: Traverse the *right subtree*

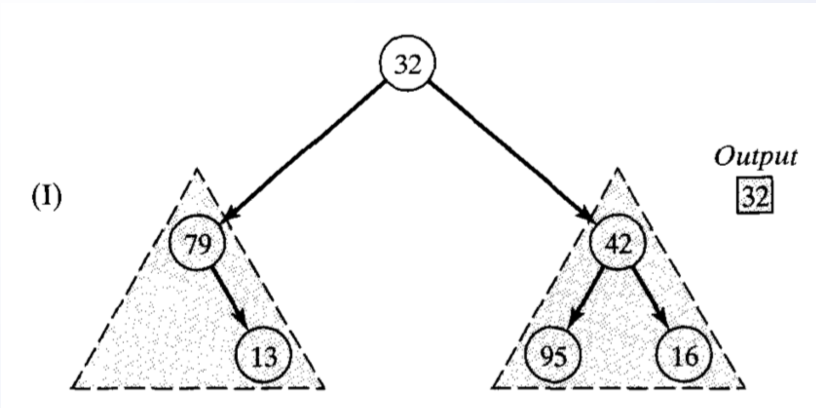
The "anchor"

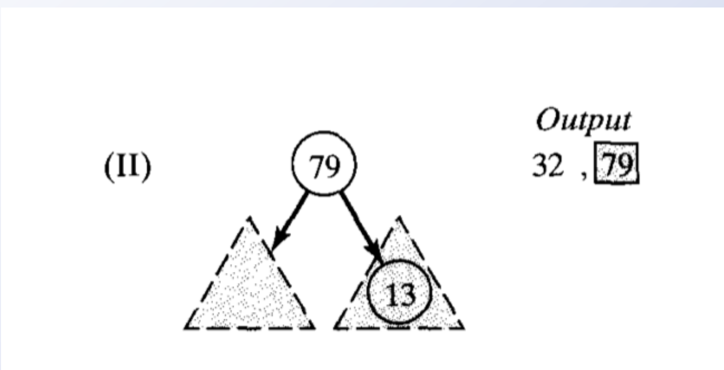
The inductive step

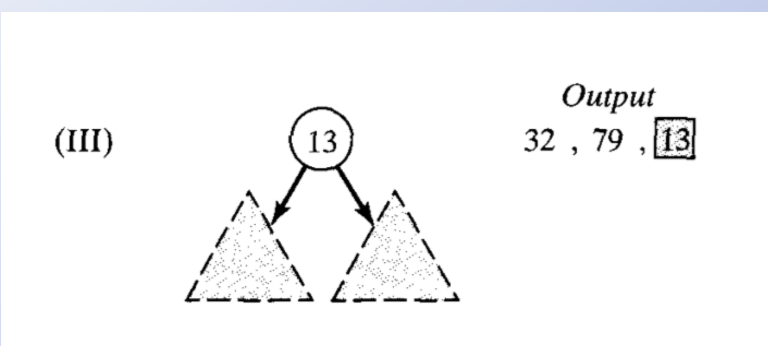


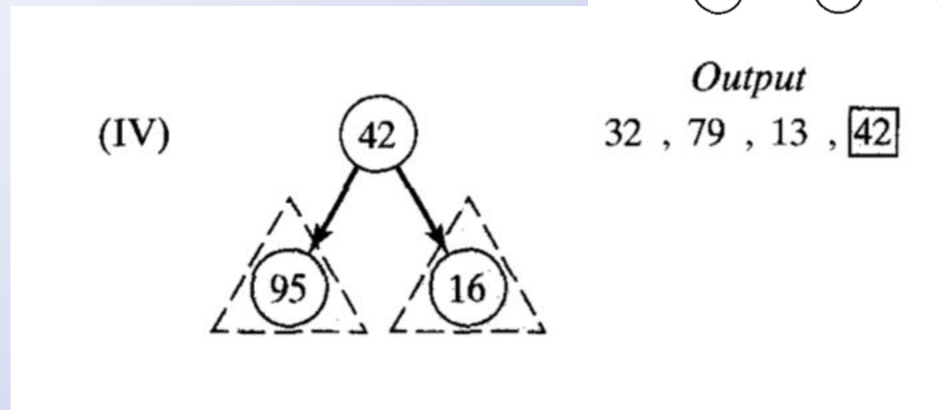
Tree Traversal is Recursive

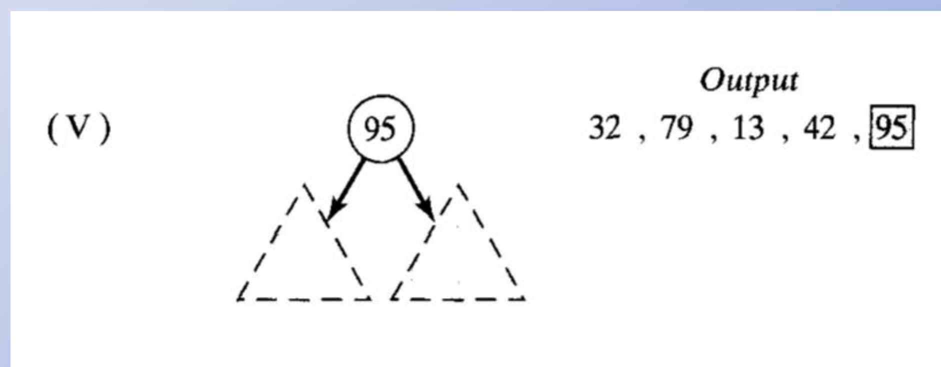


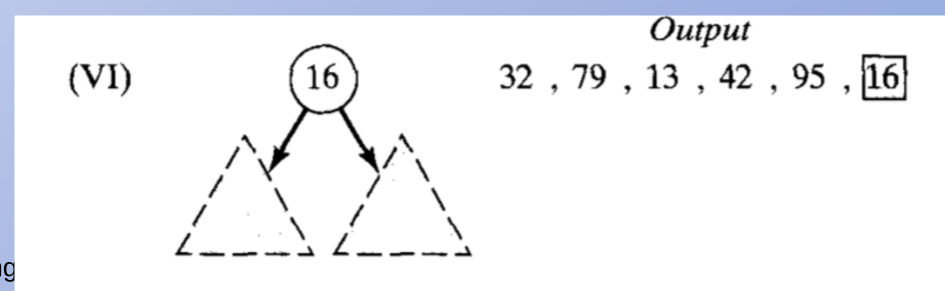




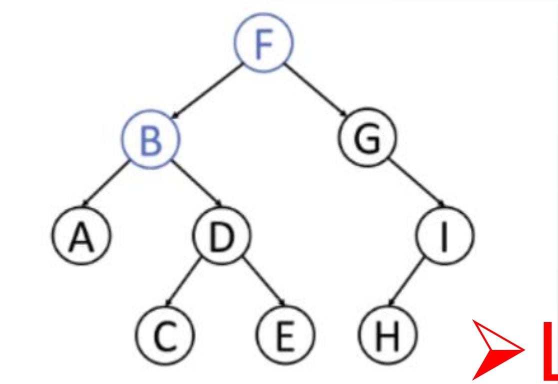






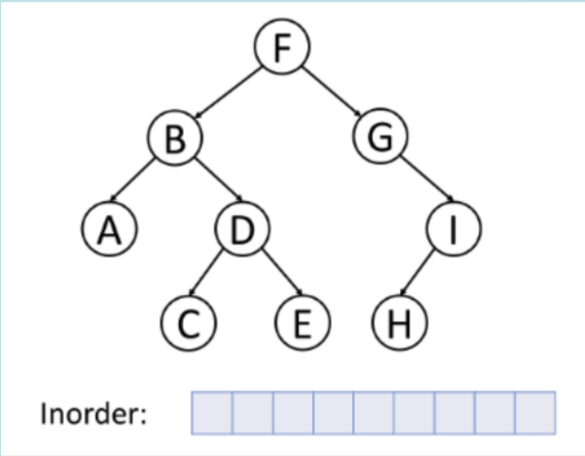


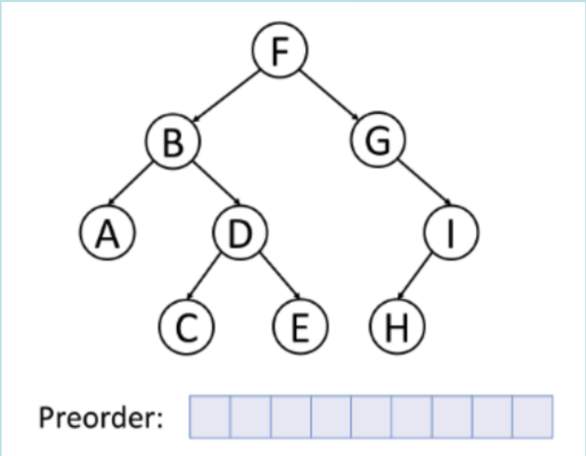
Traversal Order



➢ Left, Node, Right

the inorder traversal



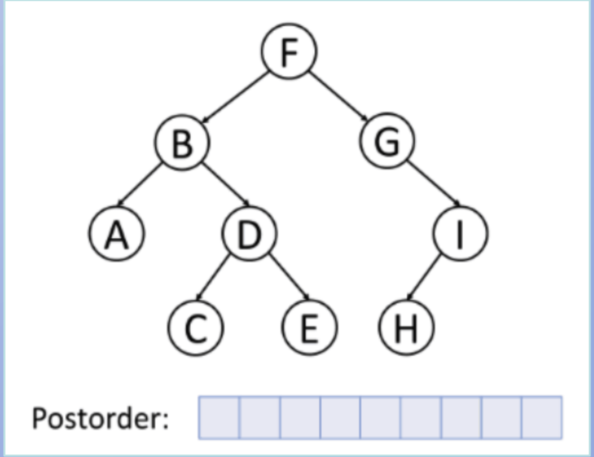


➢ Node, Left, Right

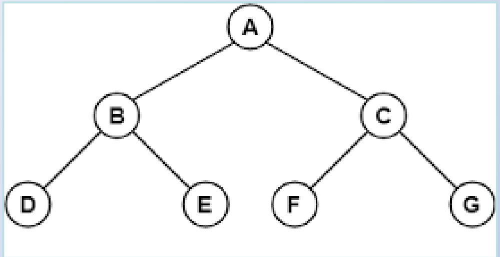
the preorder traversal

➢ Left, Right, Node

the postorder traversal



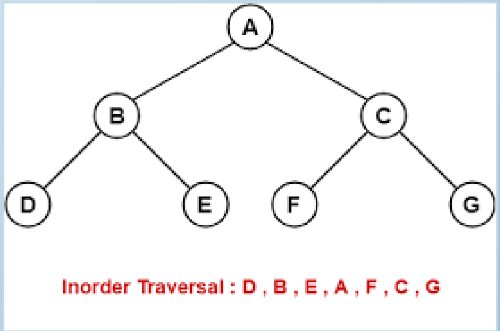
Traversal Order



Three possibilities for inductive step …

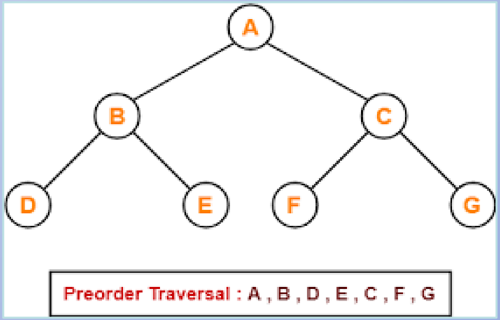
- **L**eft subtree, **N**ode, **R**ight subtree

  the inorder traversal



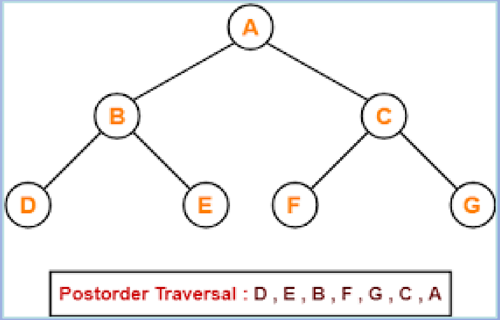
- **N**ode, **L**eft subtree, **R**ight subtree

  the preorder traversal



- **L**eft subtree, **R**ight subtree, **N**ode

  the postorder traversal

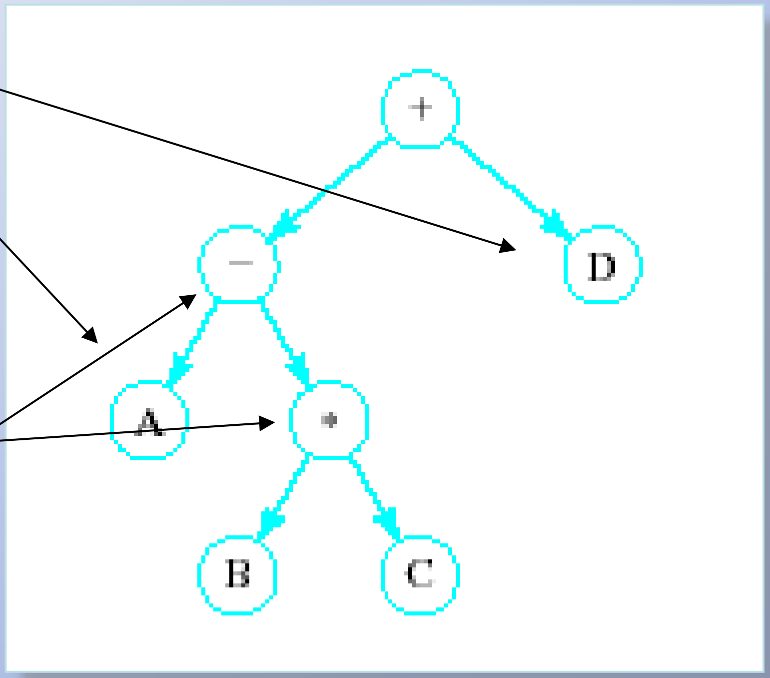


Traversal Order

- Given expression

  A – B * C + D

- Represent each operand as
  - The child of a parent node
- Parent node, representing the corresponding operator



Traversal Order

- Inorder traversal produces infix expression (LNR)

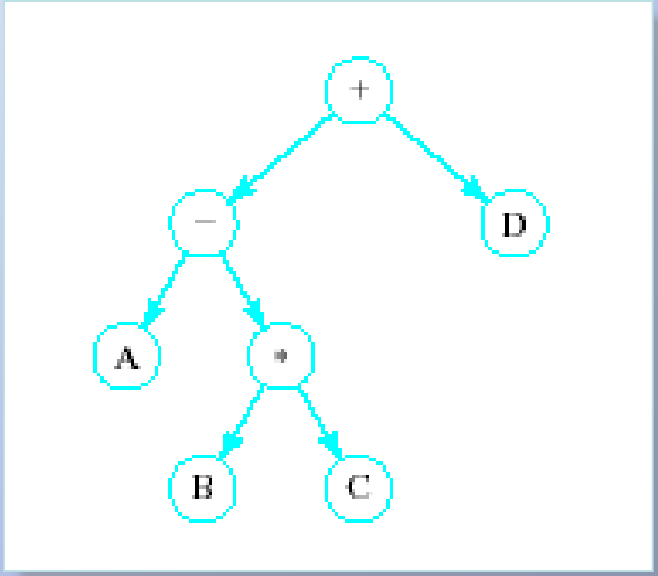
  A – B * C + D

- Preorder traversal produces the prefix expression (NLR)

  + – A * B C D

- Postorder traversal produces the postfix or RPN expression (LRN)

  A B C * – D +



Example: Traversal Order

- <u>Inorder</u> traversal produces <u>infix</u> expression (LNR)

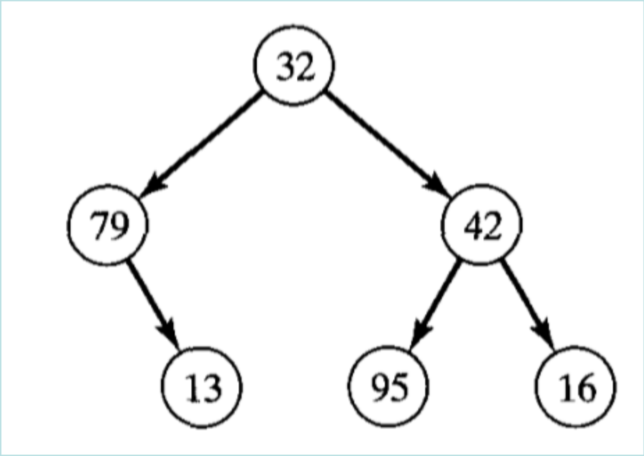
79, 13, 32, 95, 42, 16

- <u>Preorder</u> traversal produces the <u>prefix</u> expression (NLR)

32, 79, 13, 42, 95, 16

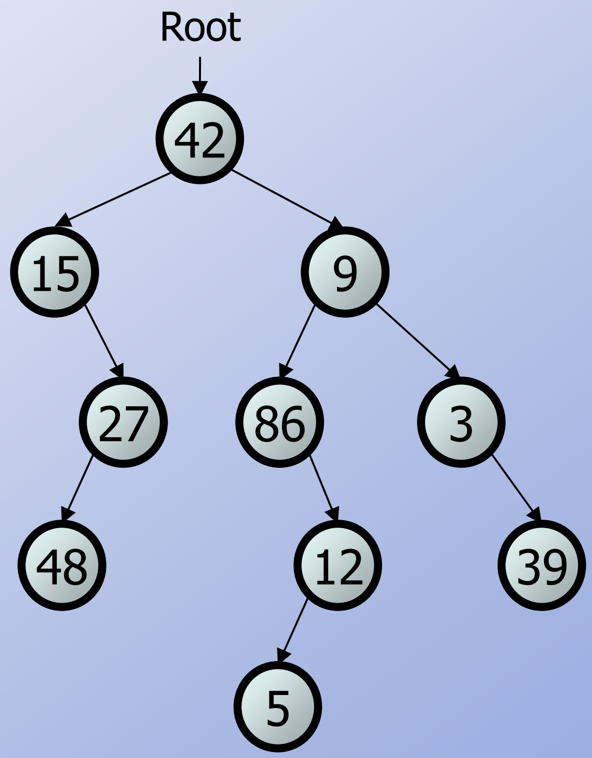
- <u>Postorder</u> traversal produces the <u>postfix</u> or RPN expression (LRN)

13, 79, 95, 16, 42, 32



Exercise

- **Pre-order:**
- **In-order:**
- **Post-order:**



Exercise

- **Pre-order:**

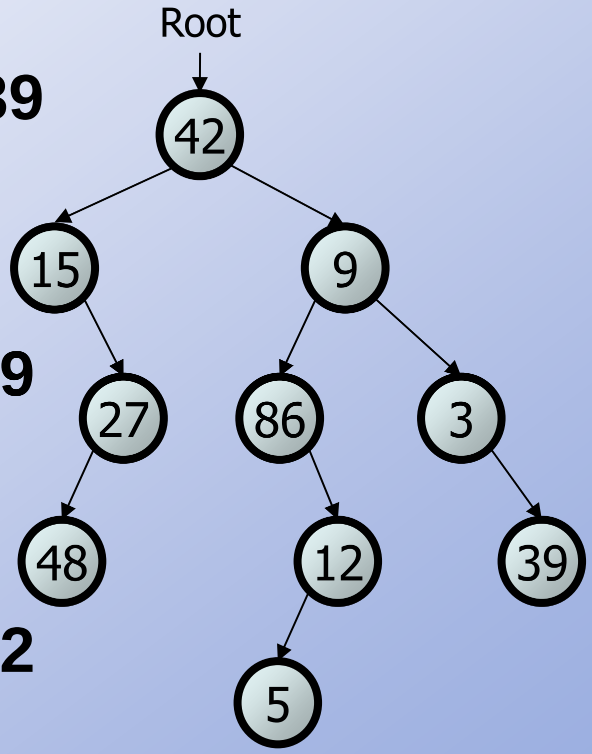
  42 15 27 48 9 86 12 5 3 39

- **In-order:**

  15 48 27 42 86 5 12 9 3 39

- **Post-order:**

  48 27 15 5 12 86 39 3 9 42



Tree Implementation

Pre: The tree is initialized.

Post: The tree has been been traversed in prefix order sequence.

```
void Preorder(TreeType t){
      if(t){
            print(t.info);
            Preorder(t.left);
            Preorder(t.right);
      }
}
```

Tree Implementation

Pre: The tree is initialized.

Post: The tree has been been traversed in Postfix order sequence.

```
void Postorder(TreeType t){
      if(t){
            Preorder(t.left);
            Preorder(t.right);
            print(t.info);
      }
}
```

Tree Implementation

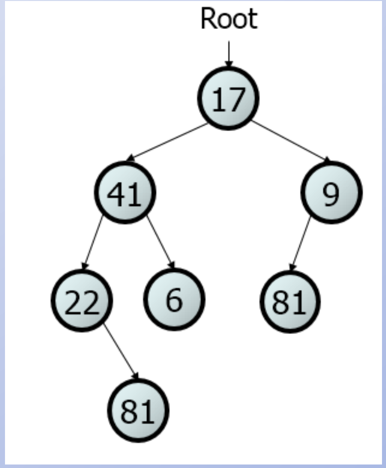
Pre: The tree is initialized.

Post: The tree has been been traversed in inorder sequence.

```
void Inorder(TreeType t){
      if(t){
            Preorder(t.left);
            print(t.info);
            Preorder(t.right);
      }
}
```

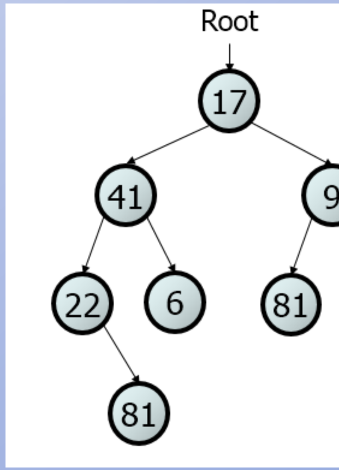
Tree Implementation

```
int Size(TreeType t){
   if (!t)
      return 0;
   return (1+Size(t.left)+Size(t.right));
}
```

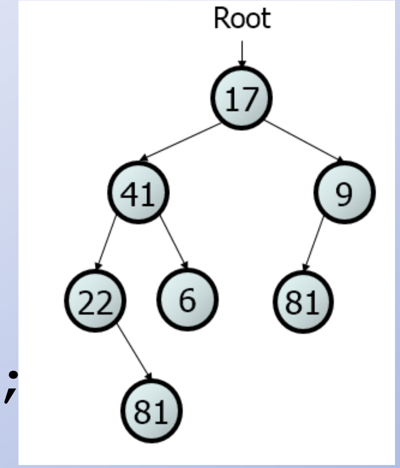


Tree Implementation

```
int height(TreeType t){
      if (!t)
            return -1;
    int a=height(t.left);
    int b=heigth(t.right);
    return (a>b)? 1+a : 1+b;
}
```



Tree Implementation




```
void ClearTree(Tree *t){
  if (*t){
      ClearTree(&(*t).left);
      ClearTree(&(*t).right);
      free(*t);
      *t=NULL;
  }
}
```