CS235102
Data Structures

Chapter 5 Trees

Chapter 5 Trees: Outline

- Introduction
  - Representation Of Trees
- Binary Trees
- Binary Tree Traversals
- Additional Binary Tree Operations
- Threaded Binary Trees
- Heaps
- Binary Search Trees
- Selection Trees
- Forests

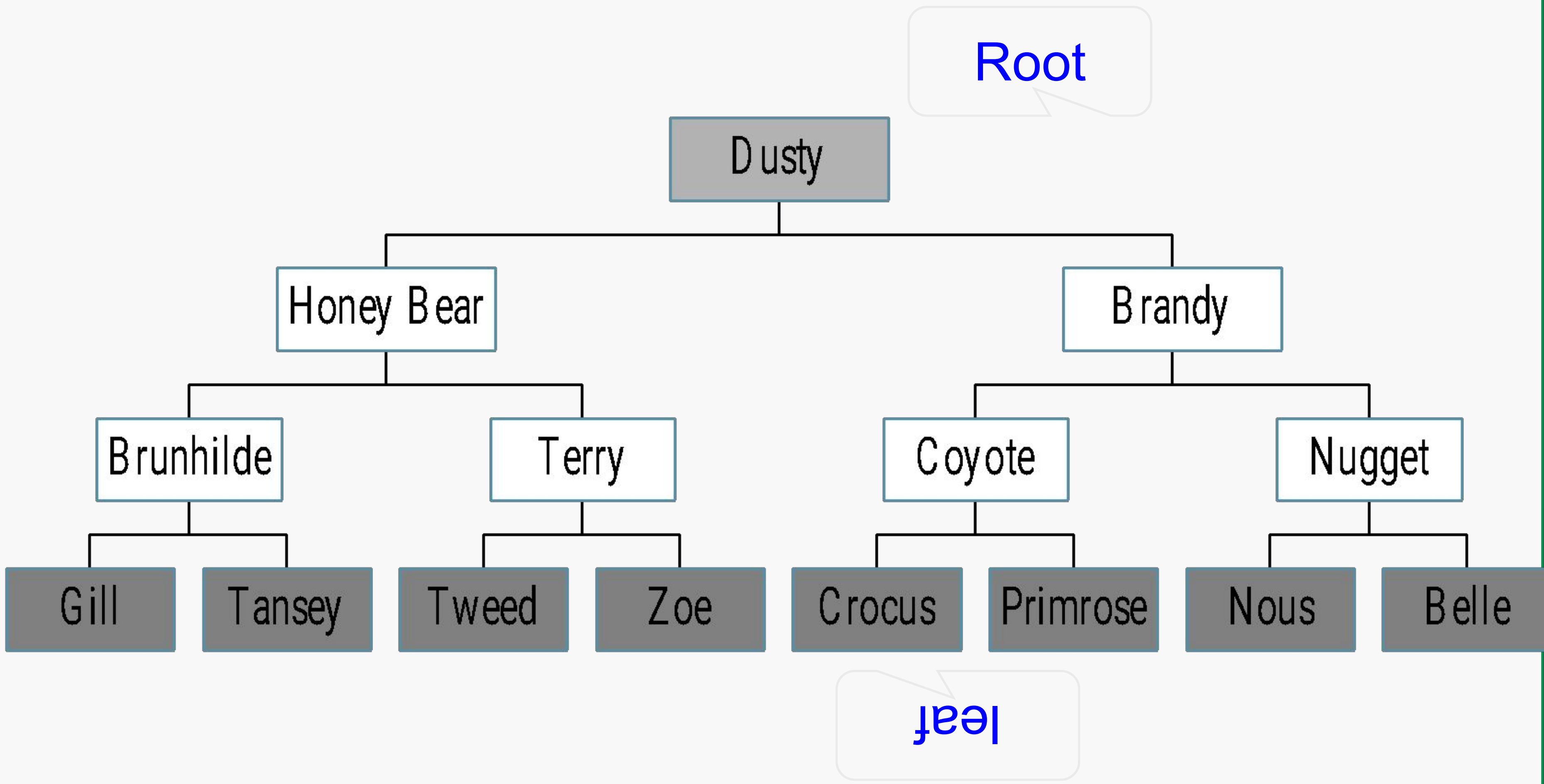
Introduction (1/8)

- A ***tree*** structure means that the data are organized so that items of information are related by branches

Introduction (2/8)

- **Definition** (recursively): A *tree* is a finite set of one or more nodes such that
  - There is a specially designated node called *root*.
  - The remaining nodes are partitioned into n>=0 disjoint set $T_1,\ldots,T_n$, where each of these sets is a tree. $T_1,\ldots,T_n$ are called the *subtrees* of the root.
- Every node in the tree is the root of some subtree

Trees


Root
Dusty
Honey Bear
Brandy
Brunhilde
Terry
Coyote
Nugget
Gill
Tansey
Tweed
Zoe
Crocus
Primrose
Nous
Belle
leaf

Introduction (3/8)

- Some **Terminology**
  - ***node***: the item of information plus the branches to each node.
  - ***degree***: the number of subtrees of a node
  - ***degree of a tree***: the **maximum of the degree** of the nodes in the tree.
  - ***terminal* nodes** (or ***leaf***): nodes that have degree zero
  - ***nonterminal* nodes**: nodes that don’t belong to terminal nodes.
  - ***children***: the roots of the subtrees of a node X are the *children* of X
  - ***parent***: X is the *parent* of its children.

Introduction (4/8)

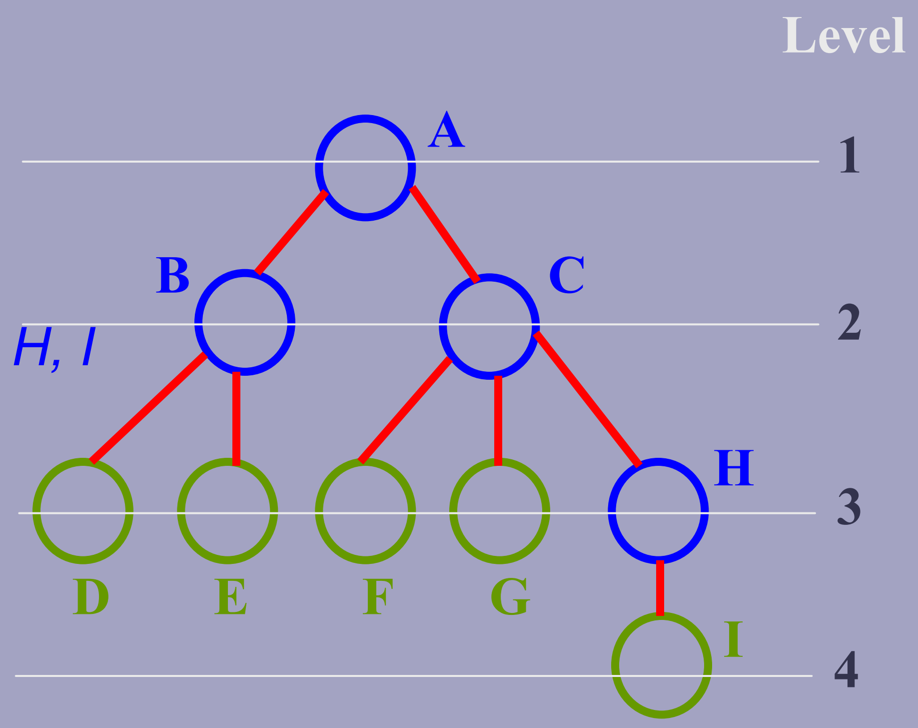
- Some **Terminology** (cont'd)
  - ***siblings***: children of the same parent are said to be siblings.
  - ***Ancestors of a node***: all the nodes along the path from the root to that node.
  - **The *level* of a node**: defined by letting the root be at level one. If a node is at level ***i***, then it children are at level ***i+1***.
  - ***Height* (*or depth*)**: the maximum level of any node in the tree

Introduction (5/8)

- Example

  A is the *root* node
  B is the *parent* of D and E
  C is the *sibling* of B
  D and *E* are the *children* of B
  D, E, F, G, I are *external nodes*, or *leaves*
  A, B, C, H are *internal nodes*
  The *level* of *E* is *3*
  The *height (depth)* of the tree is *4*
  The *degree* of node *B* is *2*
  The *degree* of the tree is *3*
  The *ancestors* of node *I* is *A, C, H*
  The *descendants* of node *C* is *F, G, H, I*

Property: (# edges) = (#nodes) - 1

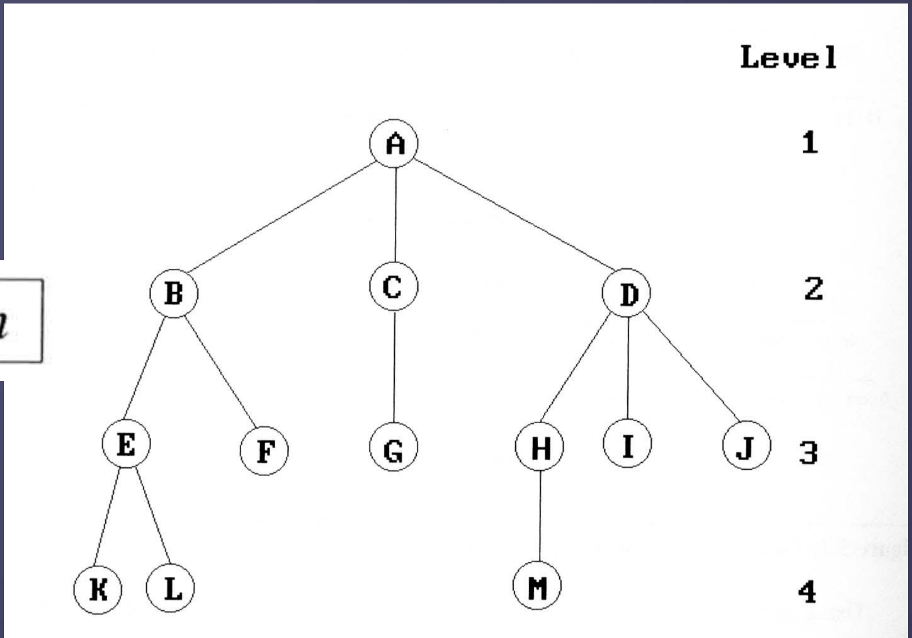


Introduction (6/8)

- Representation Of Trees
  - List Representation
    - we can write of Figure 5.2 as a list in which each of the subtrees is also a list

(**A** (**B** (E (K, L), F), **C** (G), **D** (H (M), I, J)))

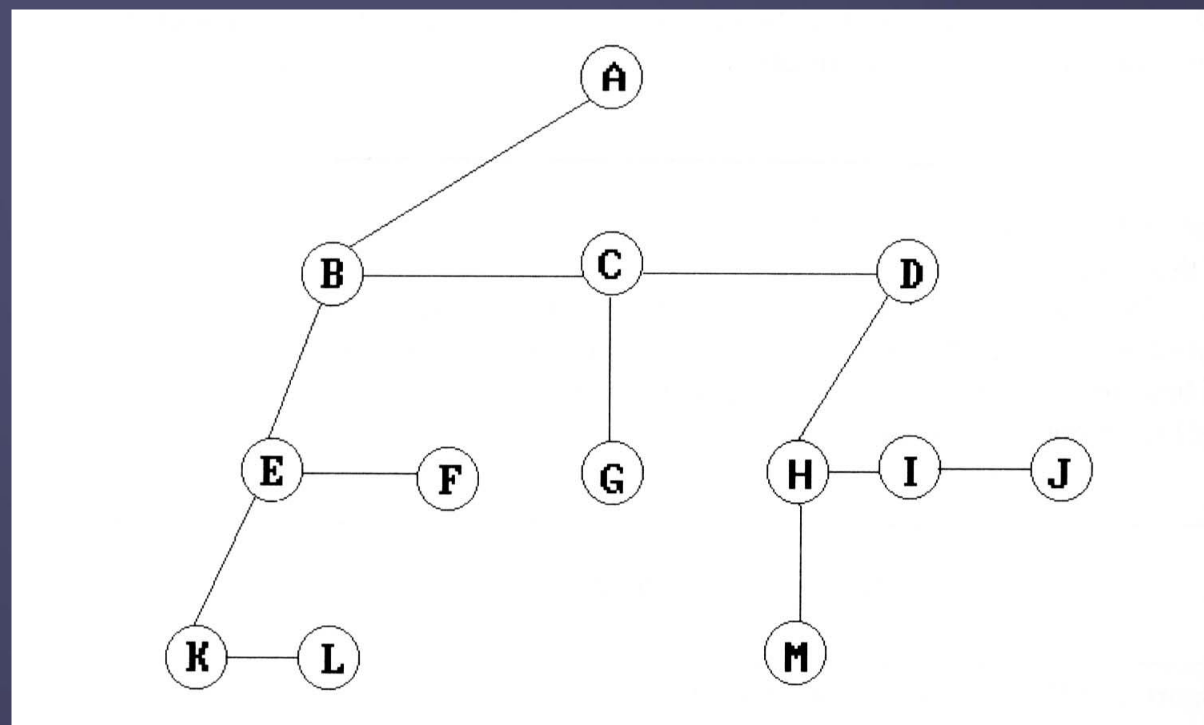
- The root comes first, followed by a list of sub-trees



Introduction (7/8)

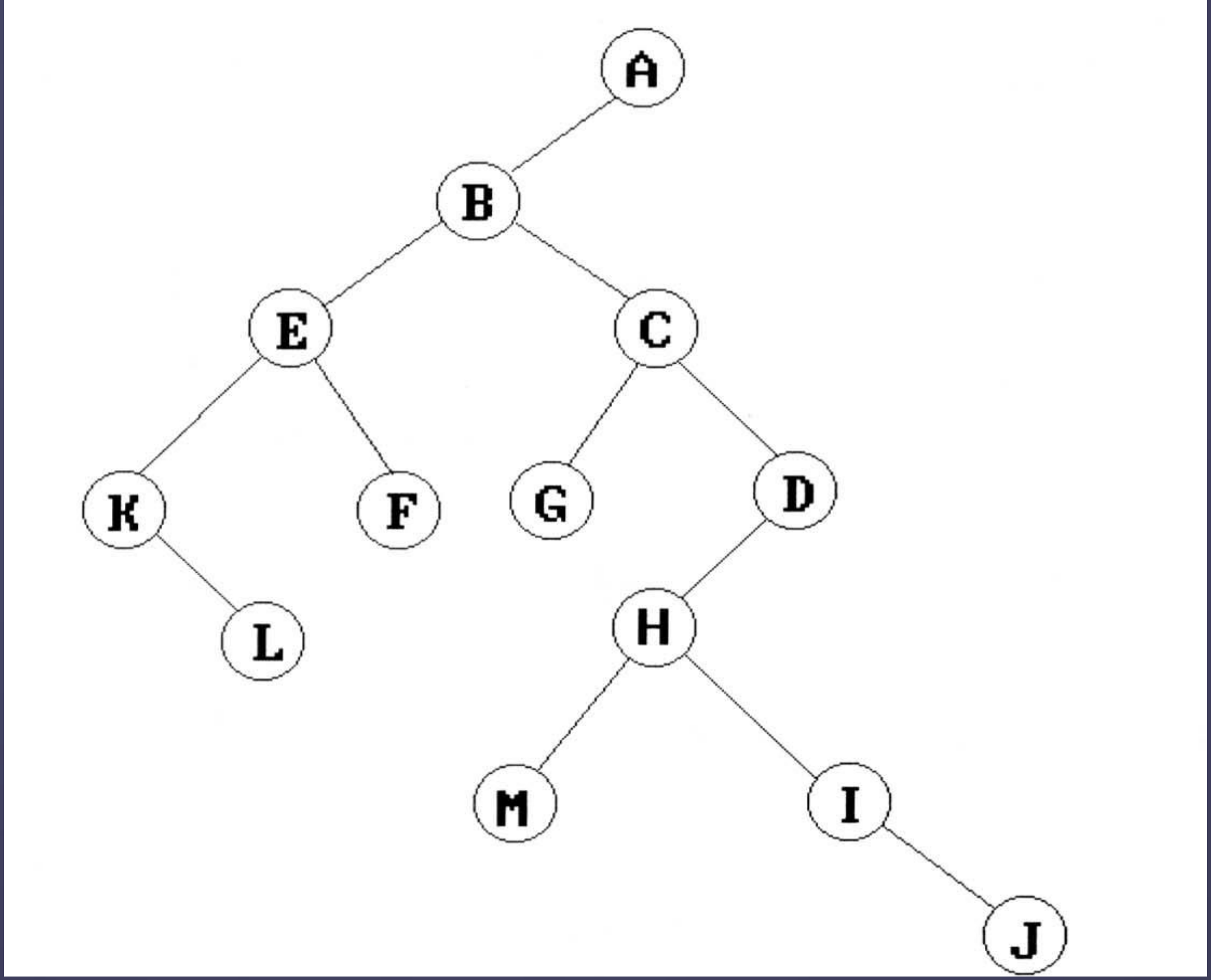
- Representation Of Trees (cont'd)
  - **Left Child-Right Sibling Representation**

<table>
<tr><td colspan="2">data</td></tr>
<tr><td>left child</td><td>right sibling</td></tr>
</table>



Introduction (8/8)

- Representation Of Trees (cont'd)
  - A Degree Two Tree



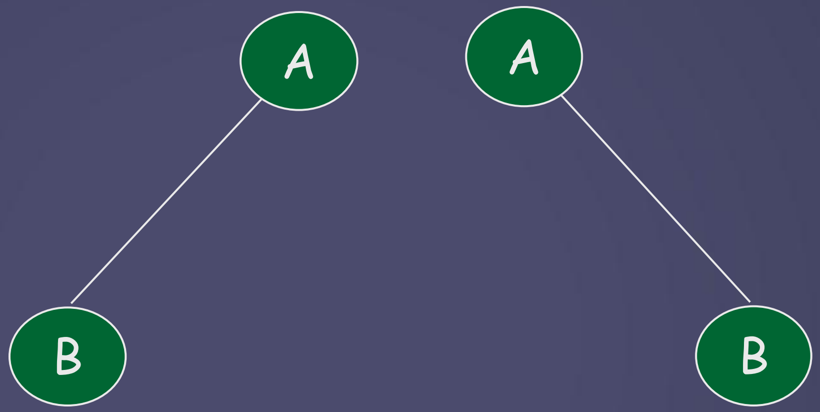
Binary Trees (1/9)

- Binary trees are characterized by the fact that any node can have at most two branches

- **Definition** (recursive):
  - A ***binary tree*** is a finite set of nodes that is either **empty** or **consists of a root and two disjoint binary trees** called the left subtree and the right subtree

Binary Trees (1/9)

- Thus the left subtree and the right subtree are distinguished



- Any tree can be transformed into binary tree
  - **by left child-right sibling representation**

Binary Trees (2/9)

- The abstract data type of binary tree

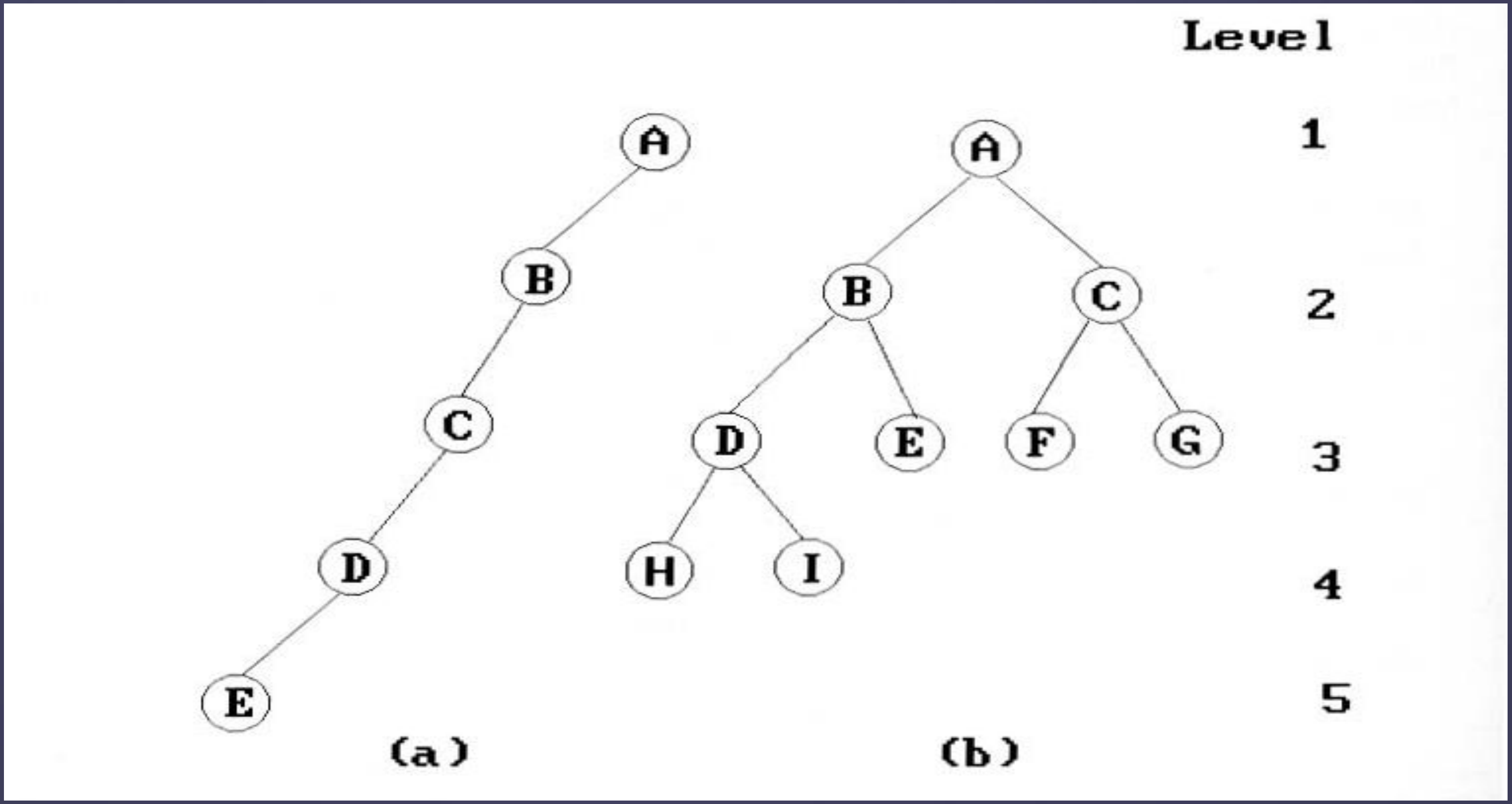
structure *Binary_Tree* (abbreviated *BinTree*) is
objects: a finite set of nodes either empty or consisting of a root node, left *Binary_Tree*, and right *Binary_Tree*.
functions:
for all $bt, bt1, bt2 \in BinTree$, $item \in element$

BinTree Create()	::=	creates an empty binary tree
Boolean IsEmpty(*bt*)	::=	**if** (*bt* == empty binary tree) **return** *TRUE* **else return** *FALSE*
BinTree MakeBT(*bt*1, *item*, *bt*2)	::=	**return** a binary tree whose left subtree is *bt*1, whose right subtree is *bt*2, and whose root node contains the data *item*.
BinTree Lchild(*bt*)	::=	**if** (IsEmpty(*bt*)) **return** error **else return** the left subtree of *bt*.
element Data(*bt*)	::=	**if** (IsEmpty(*bt*)) **return** error **else return** the data in the root node of *bt*.
BinTree Rchild(*bt*)	::=	**if** (IsEmpty(*bt*)) **return** error **else return** the right subtree of *bt*.

Structure 5.1: Abstract data type *Binary_Tree*

Binary Trees (3/9)

- Two special kinds of binary trees:
  (a) ***skewed tree***,
- (b) ***complete binary tree***
  - The all leaf nodes of these trees are on two adjacent levels



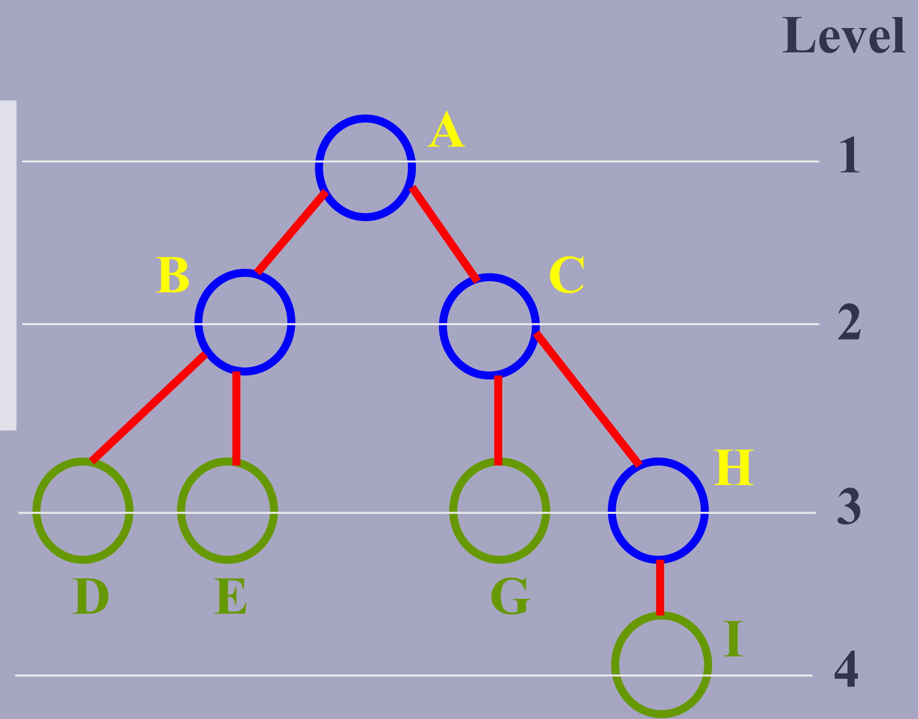
Binary Trees (4/9)

Lemma 5.1 [*Maximum number of nodes*]:

1. The maximum number of nodes **on level *i*** of a binary tree is $\mathbf{2^{i-1}}$, $i \geq 1$.
2. The maximum number of nodes in a **binary tree of depth *k*** is $\mathbf{2^k-1}$, where $k \geq 1$.

Prove by induction.

$$\sum_{i=1}^{k} 2^{i-1} = 2^k - 1$$



Binary Trees (4/9)

Lemma 5.2 [*Relation between number of leaf nodes and degree-2 nodes*]:

For any nonempty binary tree, T, if n_0 is the number of leaf nodes and n_2 is the number of nodes of degree 2, then $n_0 = n_2 + 1$.

- These lemmas allow us to define full and complete binary trees

Relations between Number of Leaf Nodes and Nodes of Degree 2

For any nonempty binary tree, T, if n_0 *is the number of leaf nodes and* n_2 *is the number of nodes of degree 2, then* $n_0 = n_2 + 1$

proof:

For any nonempty binary tree, T, if $\boldsymbol{n_0}$ *is the number of leaf nodes and* $\boldsymbol{n_2}$ *is the number of nodes of degree 2, then* $\boldsymbol{n_0=n_2+1}$

proof:

Let n and $\boldsymbol{B}$ denote the total number of nodes & **branches in T**.

Let n_0, n_1, n_2 represent the nodes with no children, single child, and two children respectively.

$B+1=n$, $B=n_1+2n_2$ ==> $n_1+2n_2+1=n$,

$n=n_0+n_1+n_2$, $n_1+2n_2+1=n_0+n_1+n_2$ ==> $n_0=n_2+1$

Binary Trees (5/9)

- Definition:
  - A ***full binary tree*** of depth ***k*** is a binary tree of death ***k*** having **2^k-1 nodes**, k ≥ 0.
  - A binary tree with n nodes and depth k is **complete** *iff* its nodes correspond to the nodes **numbered from 1 to n in the full binary tree** of depth k.
  - From Lemma 5.1, the **height** of a complete binary tree with *n* nodes is $\lceil \log_2(n+1) \rceil$

Binary Trees (6/9)

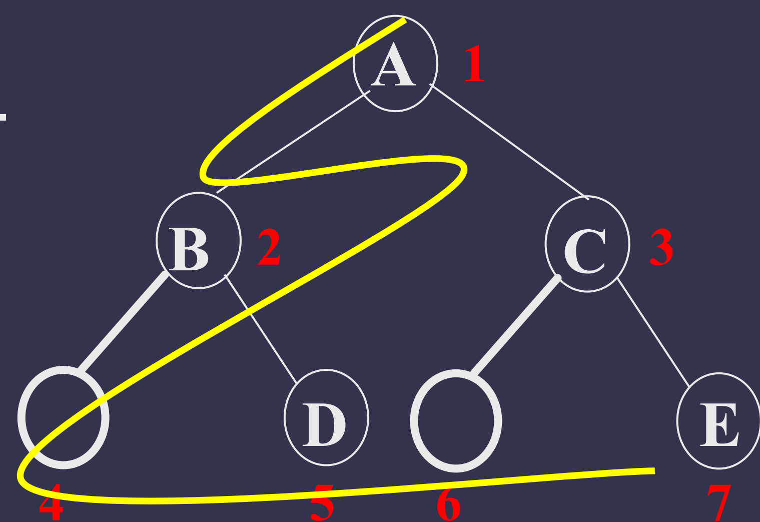
- **Binary tree representations (using array)**

Lemma 5.3: If a complete binary tree with n nodes is represented sequentially, then for any node with index i, $1 \le i \le n$, we have

1. ***parent(i)*** is at $\lfloor i/2 \rfloor$ if $i \neq 1$.
   If $i = 1$, i is at the root and has no parent.
2. ***LeftChild(i)*** is at **$2i$** if $2i \le n$.
   If $2i > n$, then i has no left child.
3. ***RightChild(i)*** is at **$2i+1$** if $2i+1 \le n$.
   If $2i + 1 > n$, then i has no left child

[1]	[2]	[3]	[4]	[5]	[6]	[7]
A	B	C	—	D	—	E

Level 1   Level 2   Level 3



Binary Trees (7/9)

- Binary tree representations (using array)
  - **Waste spaces**: in the worst case, a skewed tree of depth *k* requires 2^k-1 spaces. Of these, only *k* spaces will be occupied
  - Insertion or deletion of nodes from the middle of a tree requires the **movement of potentially many nodes to reflect the change in the level** of these nodes

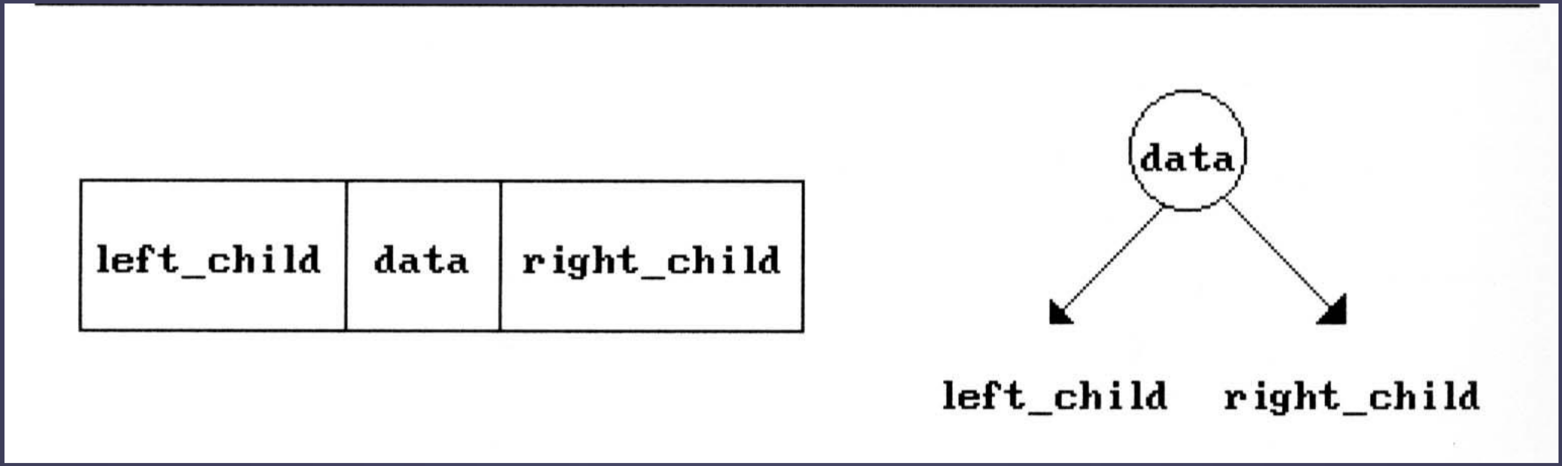
[1]	A
[2]	B
[3]	—
[4]	C
[5]	—
[6]	—
[7]	—
[8]	D
[9]	—
. . .	. . .
[16]	E

[1]	A
[2]	B
[3]	C
[4]	D
[5]	E
[6]	F
[7]	G
[8]	H
[9]	I

Binary Trees (8/9)

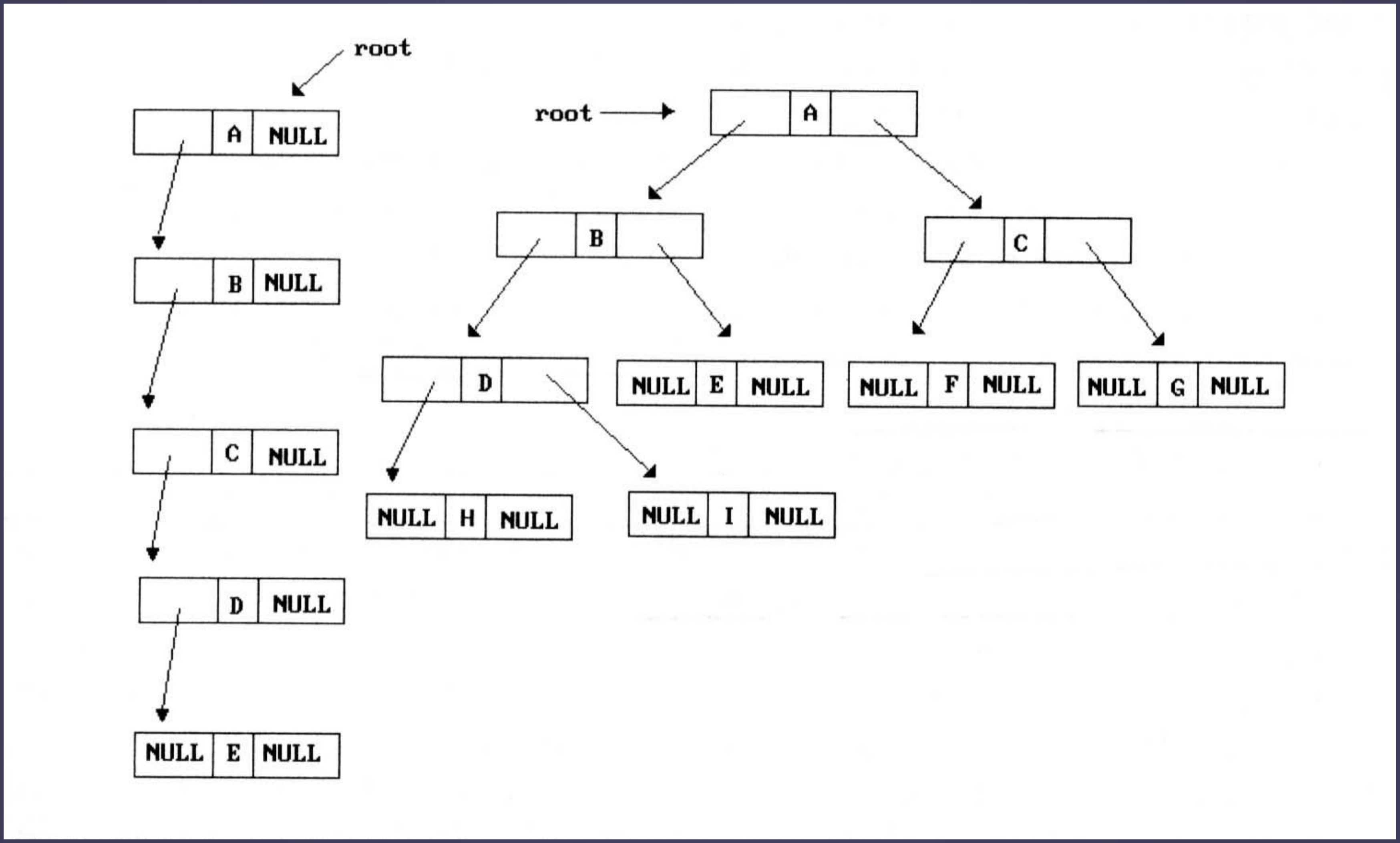
- Binary tree representations (using link)

```
typedef struct node *tree_pointer;
typedef struct node {
        int data;
        tree_pointer left_child, right_child;
        };
```



Binary Trees (9/9)

- Binary tree representations (using link)



Binary Tree Traversals (1/9)

- How to traverse a tree or visit each node in the tree exactly once?
  - There are six possible combinations of traversal

    LVR, LRV, VLR, VRL, RVL, RLV

  - Adopt convention that we traverse left before right, only 3 traversals remain

L**V**R (inorder), LR**V** (postorder), **V**LR (preorder)

left_child	data	right_child

L: **moving left**

V: **visiting node**

R: **moving right**

Binary Tree Traversals (2/9)

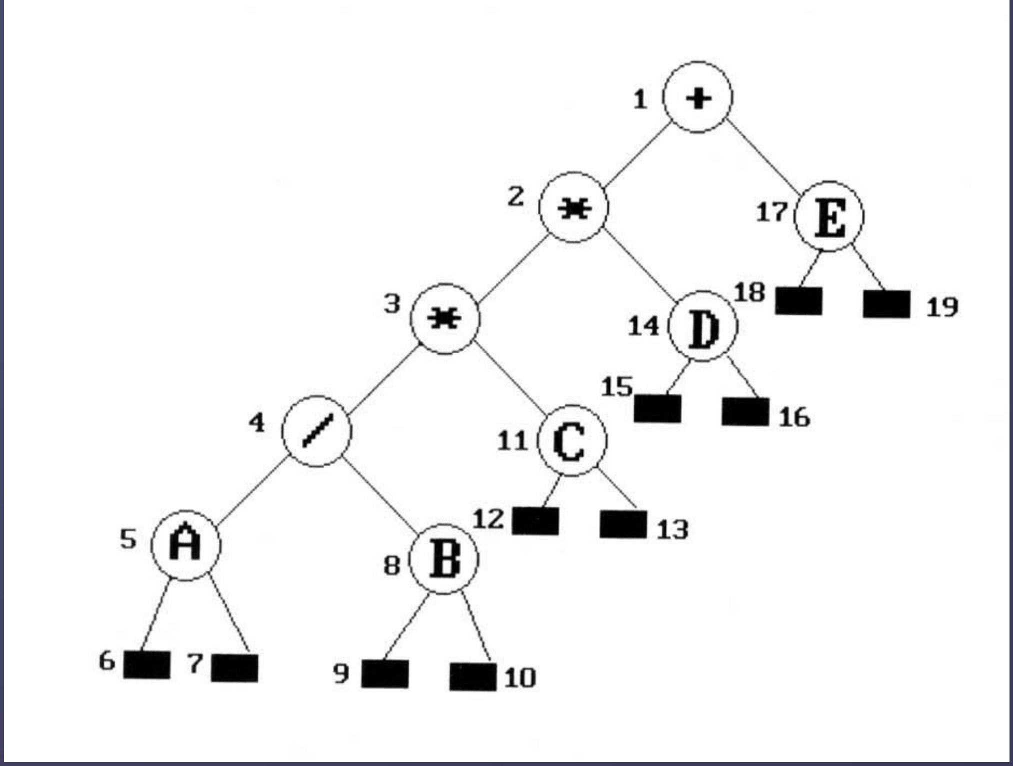
- Arithmetic Expression using binary tree
  - **inorder** traversal (infix expression)

    A / B * C * D + E
  - **preorder** traversal (prefix expression)

    + * * / A B C D E
  - **postorder** traversal (postfix expression)

    A B / C * D * E +
  - **level order** traversal

    + * E * D / C A B



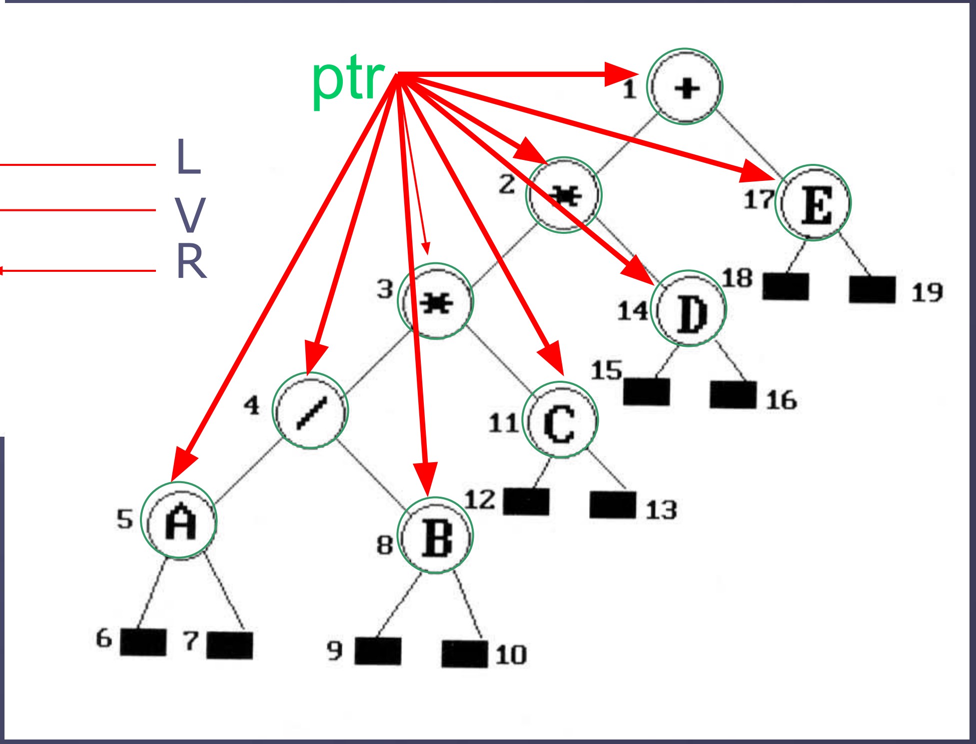
Binary Tree Traversals (3/9)

- **Inorder** traversal (*LVR*) (recursive version)

output **A / B* C * D + E**

:

```
void inorder(tree_pointer ptr)
/* inorder tree traversal */
{
    if (ptr) {
        inorder(ptr->left_child);
        printf("%d",ptr->data);
        inorder(ptr->right_child);
    }
}
```



Binary Tree Traversals (4/9)

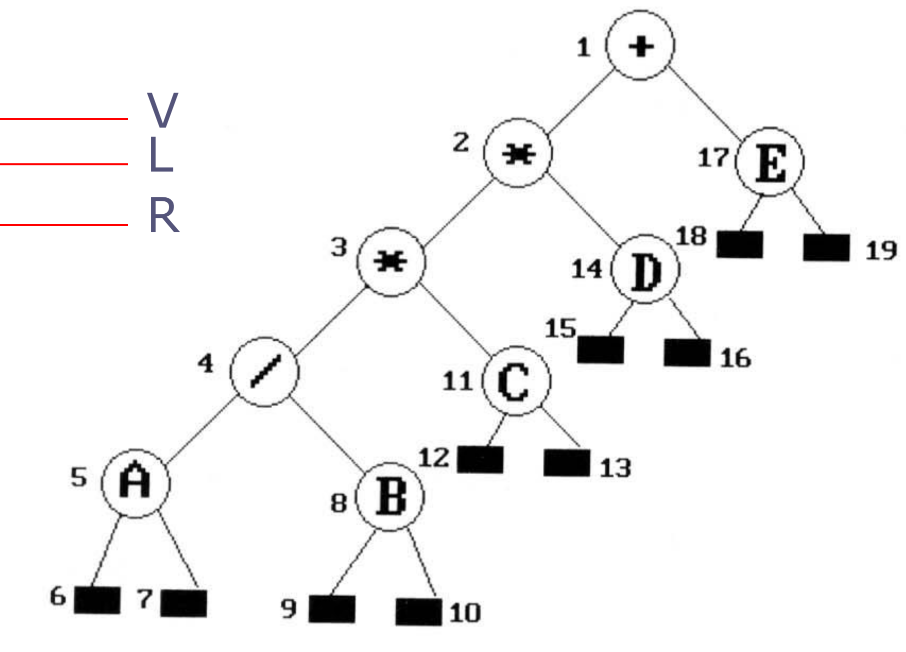
- Preorder traversal (*VLR*) (recursive version)

output + * * / A B C D E

:

```
void preorder(tree_pointer ptr)
/* preorder tree traversal */
{
   if (ptr) {
      printf("%d",ptr->data);
      preorder(ptr->left_child);
      preorder(ptr->right_child);
   }
}
```

V
L
R



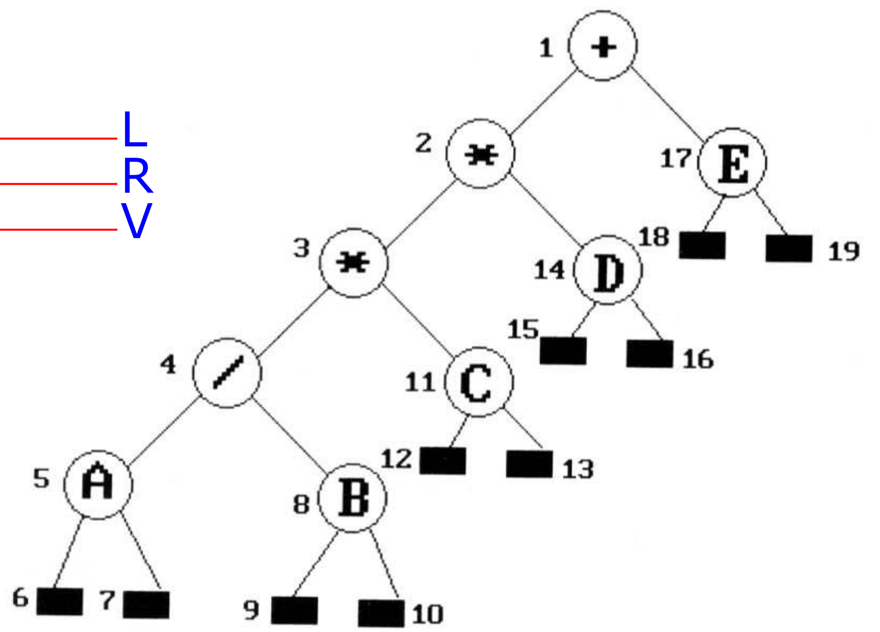
Binary Tree Traversals (5/9)

- Postorder traversal (*LRV*) (recursive version)

output : A B / C * D * E +

```
void postorder(tree_pointer ptr)
/* postorder tree traversal */
{
  if (ptr) {
    postorder(ptr->left_child);
    postorder(ptr->right_child);
    printf("%d",ptr->data);
  }
}
```

L
R
V



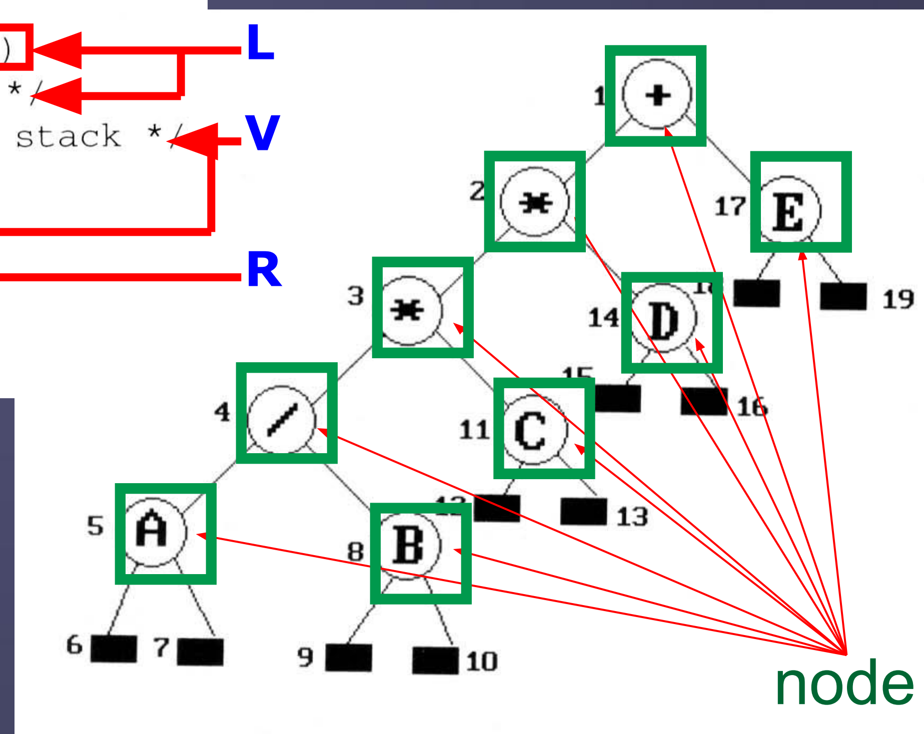
Binary Tree Traversals (6/9)

- **Iterative inorder traversal**
  - we use a **stack** to simulate recursion

```
void iter_inorder(tree_pointer node)
{
   int top = -1; /* initialize stack */
   tree_pointer stack[MAX_STACK_SIZE];
   for (;;) {
      for(; node; node = node->left_child)
         add(&top, node); /* add to stack */
      node = delete(&top); /* delete from stack */
      if (!node) break; /* empty stack */
      printf("%d", node->data);
      node = node->right_child;
   }
}
```

L
V
R

	5	8	11	14	17
	A	B	C	D	E



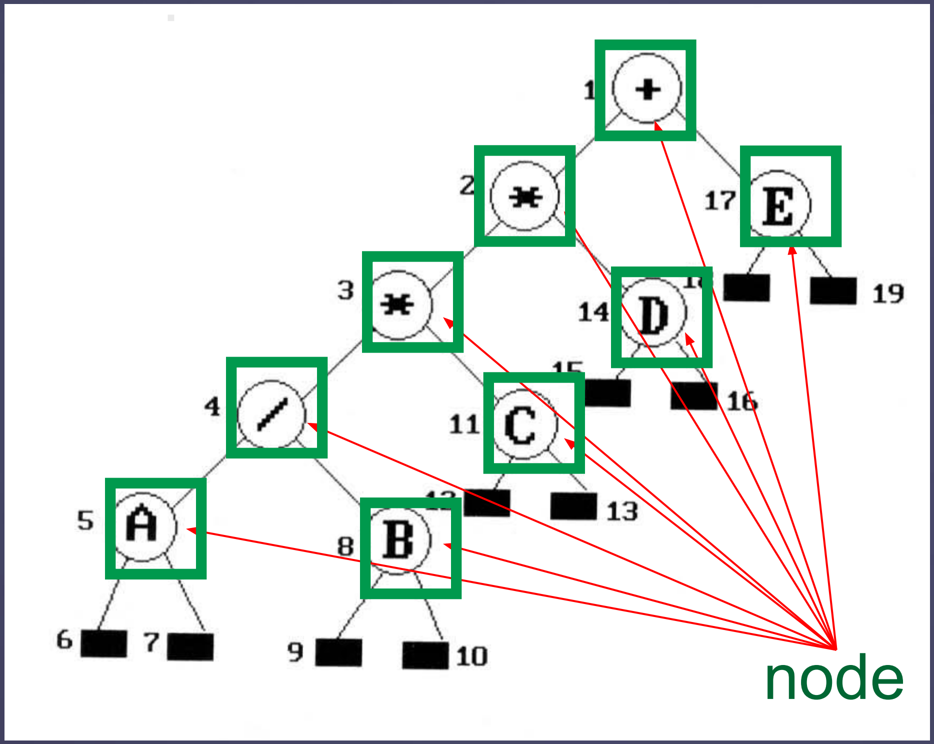
output A / B * C * D + E

:

Binary Tree Traversals (6/9)

- **Iterative inorder traversal**

output A / B * C * D + E



Binary Tree Traversals (7/9)

- Analysis of inorder2 (Non-recursive Inorder traversal)
  - Let ***n*** be the number of nodes in the tree
  - **Time complexity: O(*n*)**
    - Every node of the tree is placed on and removed from the stack exactly once
  - **Space complexity: O(*n*)**
    - equal to the depth of the tree which (skewed tree is the worst case)

Binary Tree Traversals (8/9)

- Level-order traversal
  - method:
    - We **visit the root first, then the root's left child, followed by the root's right child.**
    - We continue in this manner, visiting the nodes at each new level from the leftmost node to the rightmost nodes
  - This traversal requires a queue to implement

Binary Tree Traversals (9/9)

- Level-order traversal **(using queue)**

```
void level_order(tree_pointer ptr)
/* level order tree traversal */
{
   int front = rear = 0;
   tree_pointer queue[MAX_QUEUE_SIZE];
   if (!ptr) return; /* empty tree */
   addq(front, &rear, ptr)
   for (;;) {
      ptr = deleteq(&front, rear);
      if (ptr) {
         printf("%d",ptr->data);
         if(ptr->left_child)
            addq(front,&rear,ptr->left_child);
         if (ptr->right_child)
            addq(front,&rear,ptr->right_child);
      }
      else break;
   }
}
```

FIFO

output + * E * D / C A B

.

2	17	3	14	4	11	5	8	
*	E	*	D	/	C	A	B