Chap 5. Trees (1)

Contents

5.1 Introduction

5.2 Binary Trees

5.3 Binary Trees Traversals

5.4 Additional Binary Tree Operations

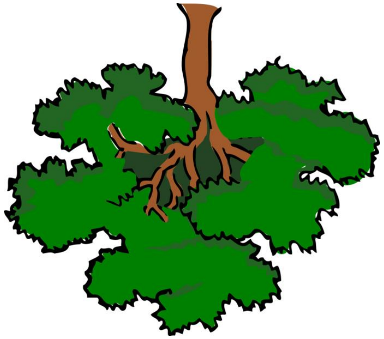
5.6 Heaps

5.7 Binary Search Trees

5.1 Introduction

5.1.1 Terminology

- ***Definition* :** A ***Tree*** is a finite set of *one or more nodes* such that
  1. There is a specially designated node called the *root*
  2. The remaining nodes are partitioned into *n* ≥ 0 *disjoint sets* T_1, …, T_n, where each of these sets is a tree. We call T_1,…, T_n the *subtrees* of the root.



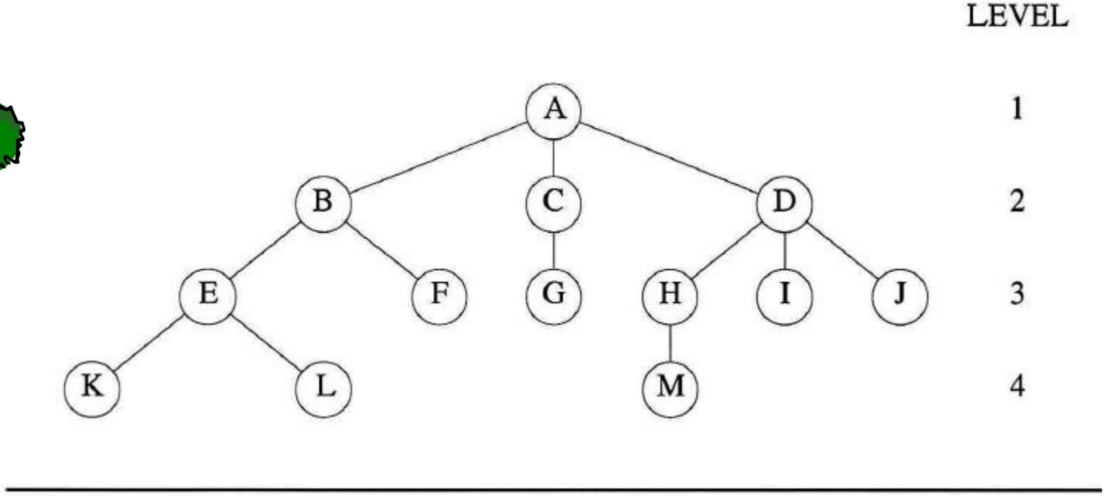
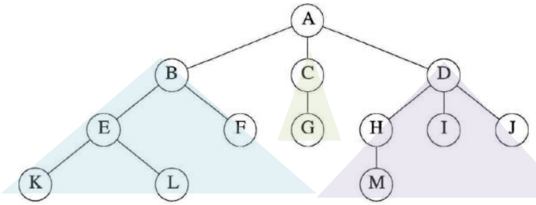


Figure 5.2: A sample tree

- degree of a node : number of subtrees of the node
- degree of a tree : maximum degree of the nodes in the tree
- leaf (terminal node) : a node with degree zero
- parent, children
- siblings : children of same parent
- grand parent, grand children
- ancestors of a node : all the nodes along the path from the root to the node
- descendants of a node : all the nodes that are in its subtrees
- level of a node
- height (depth) of a tree : maximum level of any node in the tree
- branch

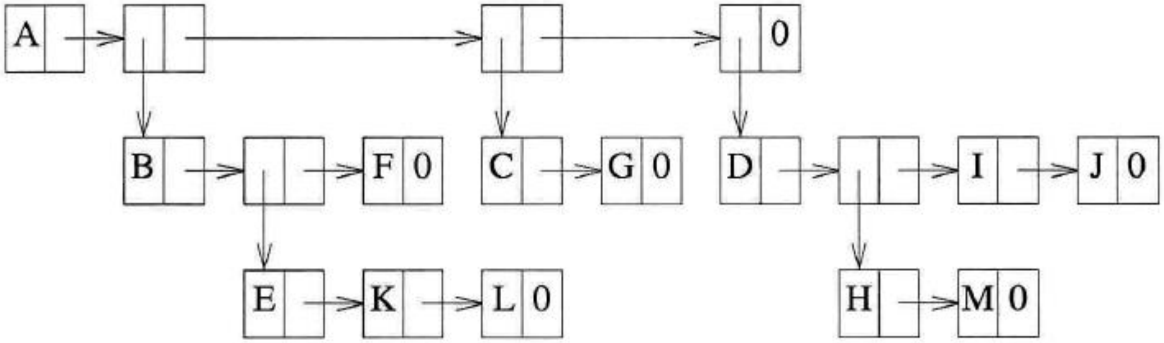
5.1.2 Representation of Trees



- List Representation

(root node (a list of the subtrees of that node))

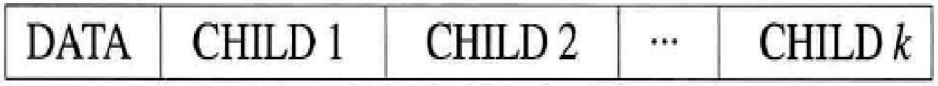
(A (B (E (K, L), F), C (G), D(H (M), I, J)))



tag fields not shown

Figure 5.3: List representation of the tree of Figure 5.2

- In Figure 5.4, a CHILD field is used to point to a subtree.
- In practice, we use only ***nodes of a fixed size*** to represent tree nodes.

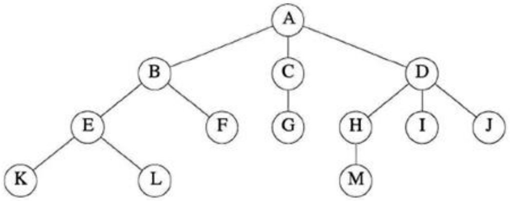


DATA	CHILD 1	CHILD 2	···	CHILD *k*

Figure 5.4: Possible node structure for a tree of degree *k*

- Left Child-Right Sibling Representation

<table>
<tr><td colspan="2">Data</td></tr>
<tr><td>Left Child</td><td>Right Sibling</td></tr>
</table>



→

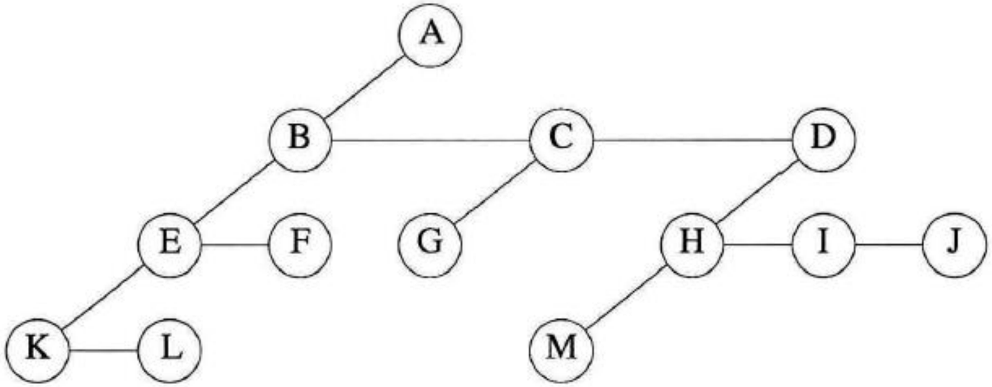
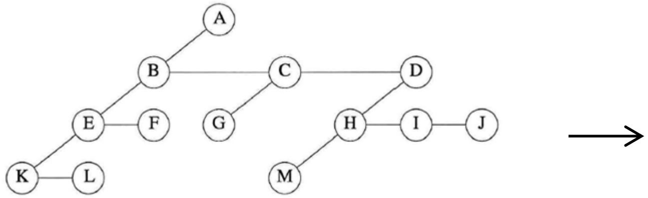


Figure 5.6: Left child-right sibling representation of tree of Figure 5.2

- Representation as a Degree Two Trees

Data	
Left Child	Right Child



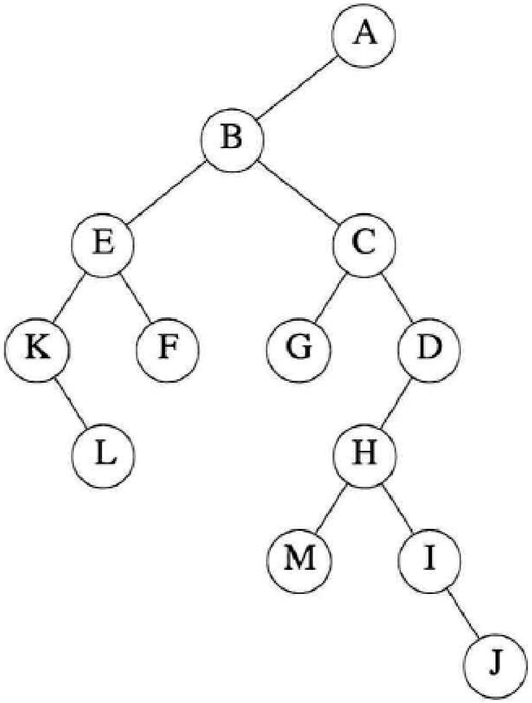


Figure 5.7: Left child-right child tree representation of tree of Figure 5.2

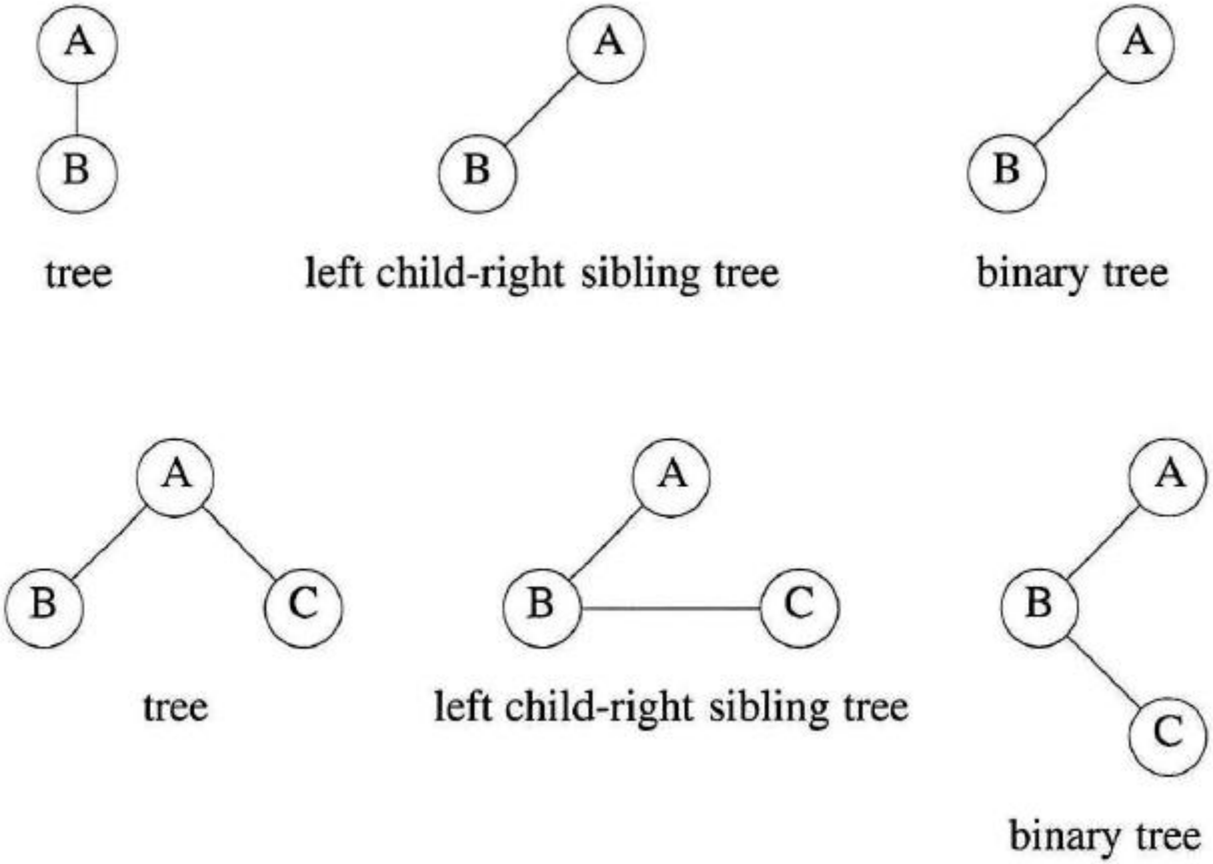


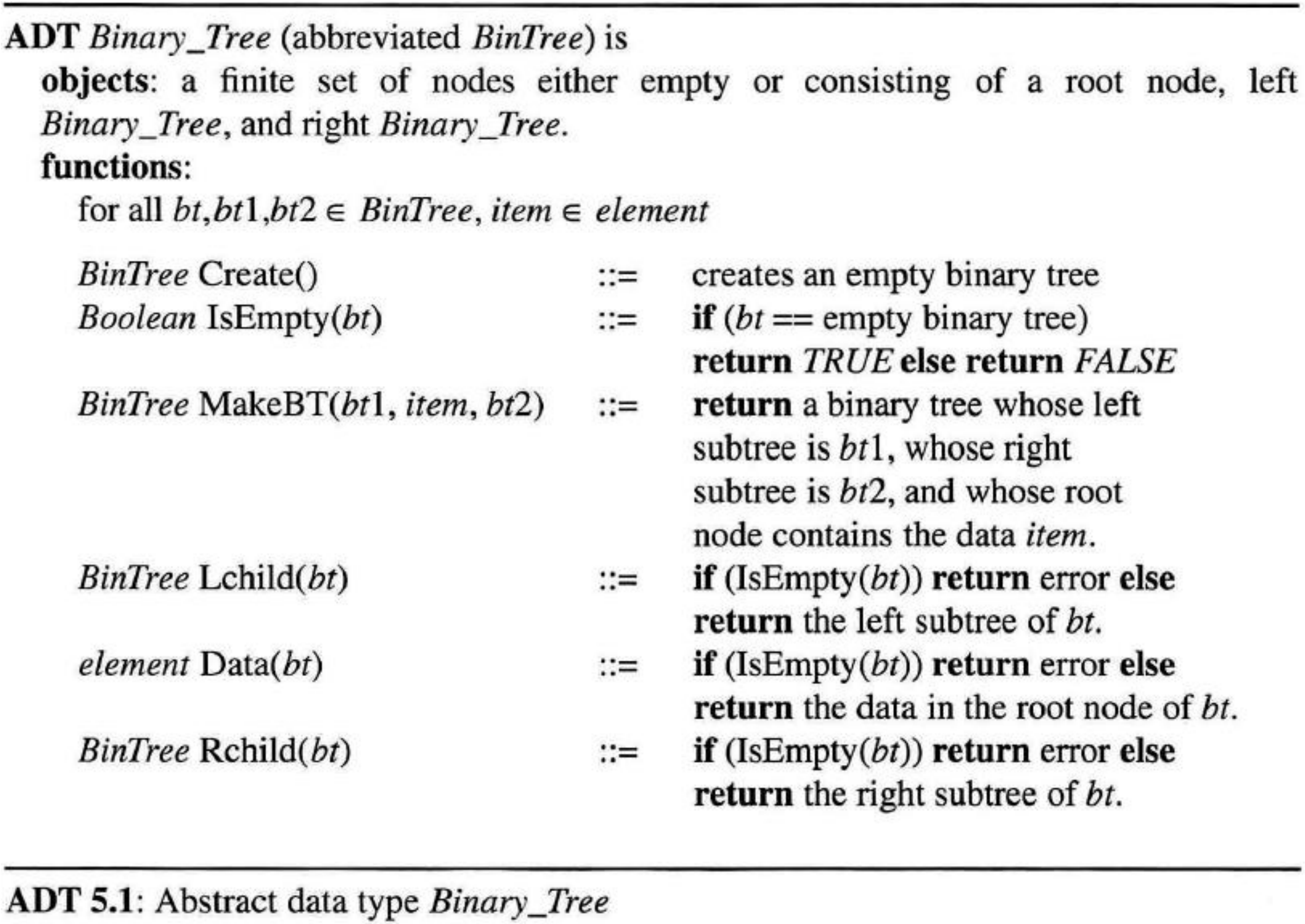
Figure 5.8: Tree representations

5.2 Binary Trees

5.2.1 The Abstract Data Type

Definition :

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*.



ADT *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*,*bt*1,*bt*2 ∈ *BinTree*, *item* ∈ *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*.

ADT 5.1: Abstract data type *Binary_Tree*

- Differences between a ***tree*** & a ***binary tree***
  1. There is no tree having zero nodes, but

     there is an empty binary tree.
  2. In a *binary tree*, we distinguish between *the order of the children* while in a tree we do not.

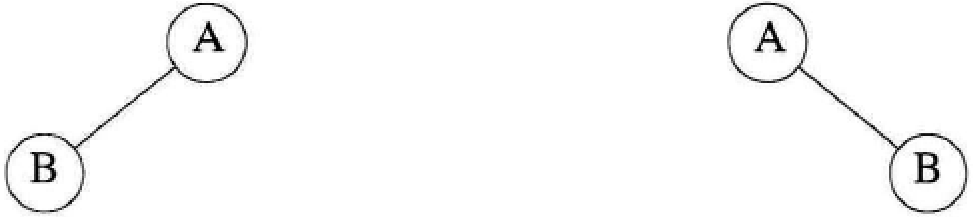


Figure 5.9: Two different binary trees

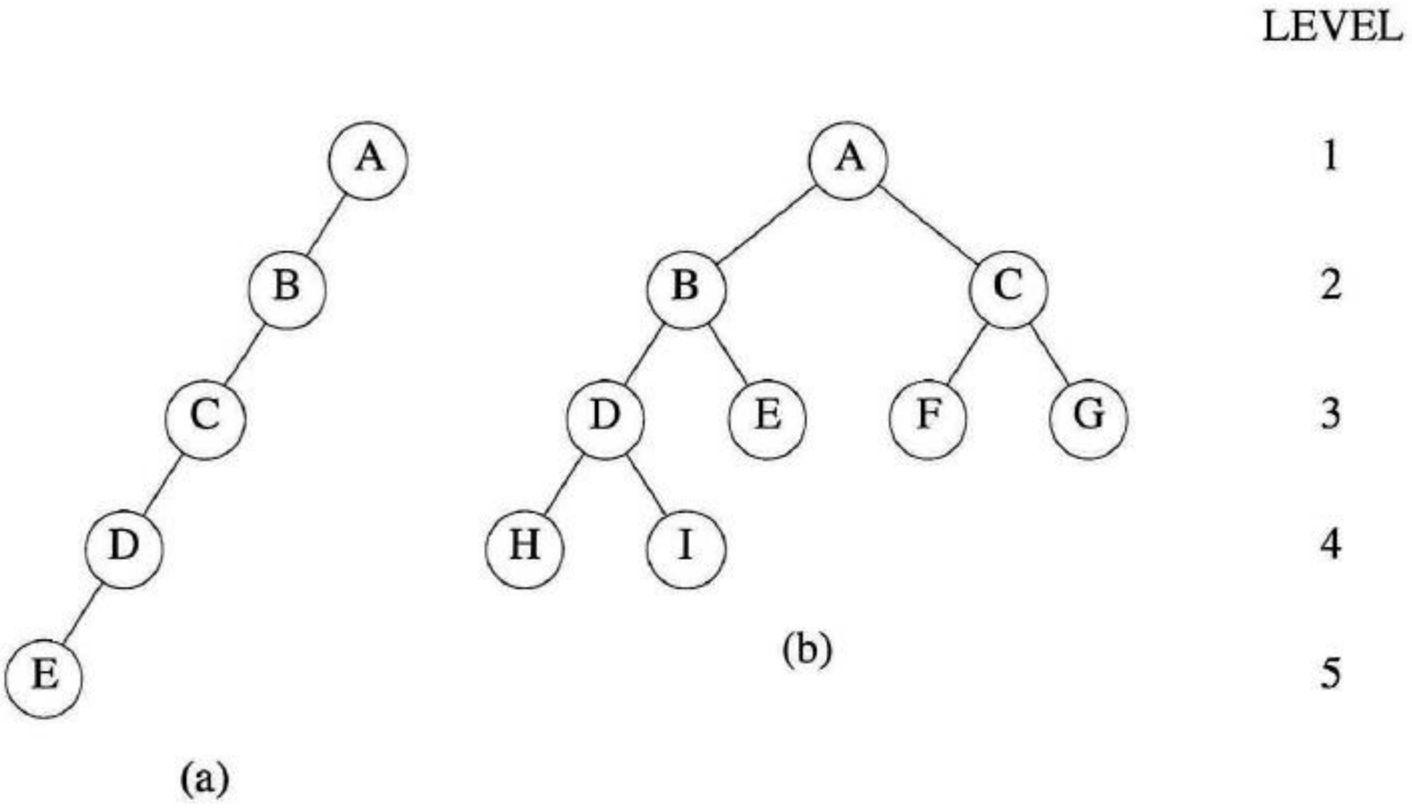


Figure 5.10: Skewed and complete binary trees

Caution

- Some texts start level numbers at 0.
  - Root is at level 0.
  - Its children are at level 1.
  - The grand children of the root are at level 2.
  - And so on.

- *We shall number levels with the root at level 1.*

5.2.2 Properties of Binary Trees

Lemma 5.2 [*Maximum number of nodes*]

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

Proof

1. Induction Base: $i = 1 \Rightarrow$ The max. # of nodes on level 1 is $2^{i-1} = 2^{0} = 1$

   Induction Hypothesis: $1 < i \Rightarrow$ The max. # of nodes on level *i*-1 is 2^{i-2}

   Induction Step: The max. # of nodes at level *i*
   = (The max. # of nodes at level *i*-1) $\times$ 2
   = $2^{i\text{-}2} \times 2 = 2^{i\text{-}1}$

2. $$\sum_{i=1}^{k} (\text{maximum number of nodes on level i}) = \sum_{i=1}^{k} 2^{i-1} = 2^{k} - 1$$

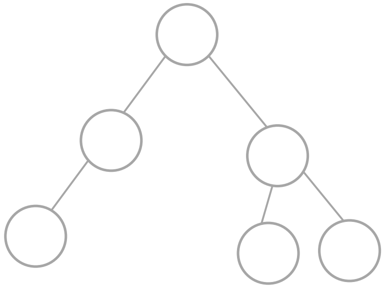
Lemma 5.3 [*Relation between number of leaf nodes and degree-2 nodes*]:

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

Proof

n: the total number of nodes

$$n = n_0 + n_1 + n_2 \qquad ①$$



B: the number of branches

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

$$n = B + 1 = n_1 + 2n_2 + 1 \qquad ②$$

$$n_0 = n_2 + 1 \qquad ①-②$$

Definition [*Full Binary Tree*] :

A *full binary tree* of depth k is a binary tree of depth k having $2^k - 1$ nodes, $k \geq 0$.

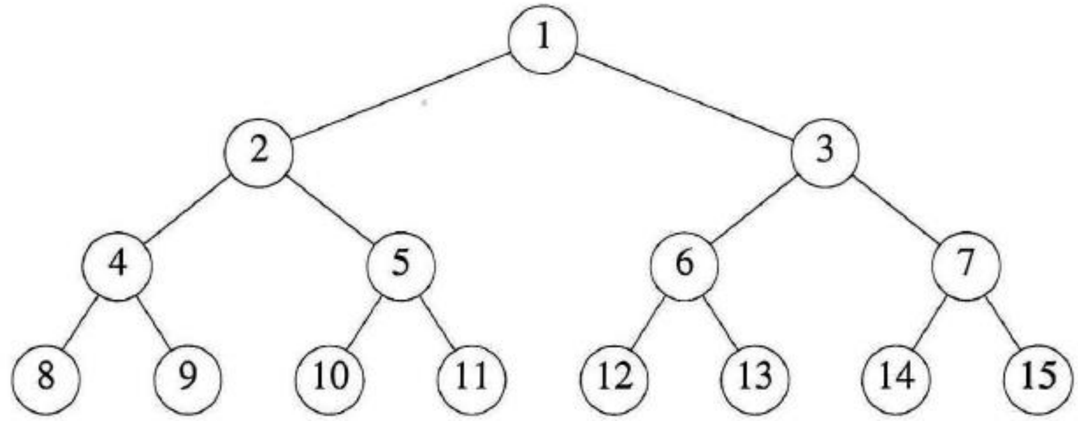
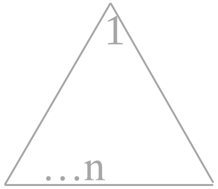
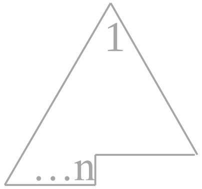


Figure 5.11: Full binary tree of depth 4 with sequential node numbers

Definition [*Complete Binary Tree*] :

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*.

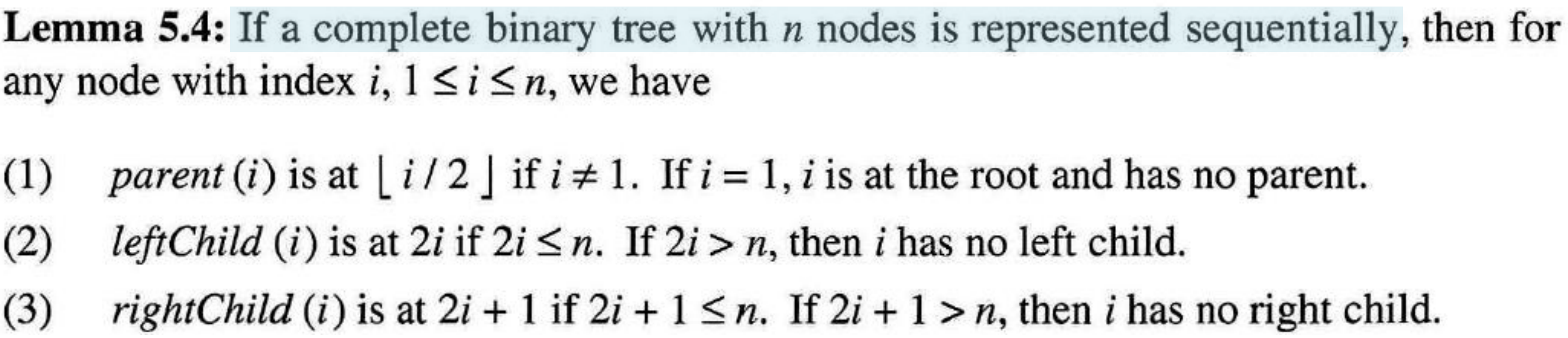




– The height of a complete binary tree with *n* nodes is $\lceil log_2(n+1) \rceil$

5.2.3 Binary Tree Representation

- **Array Representation**

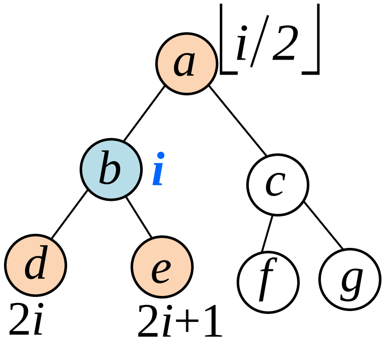


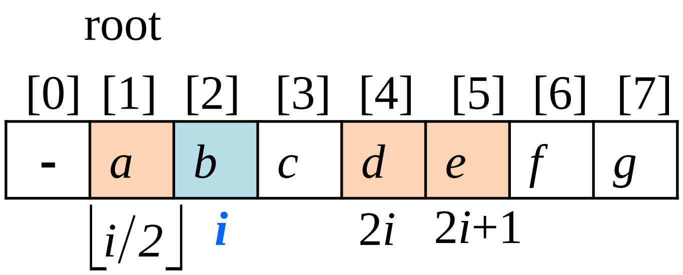
Lemma 5.4: If a complete binary tree with n nodes is represented sequentially, then for any node with index i, $1 \le i \le n$, we have

(1) *parent* (i) is at $\lfloor i/2 \rfloor$ if $i \ne 1$. 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.

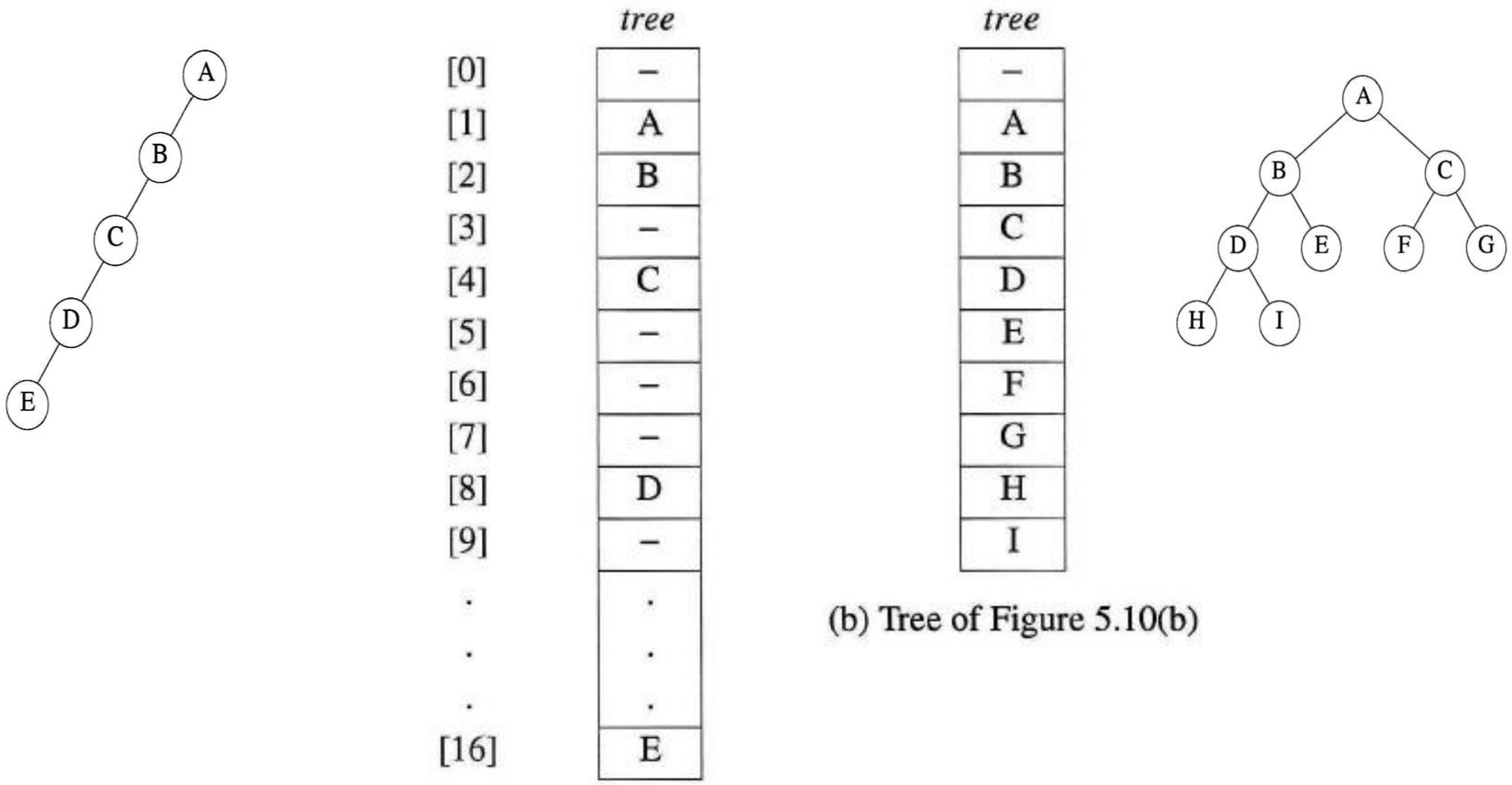
(3) *rightChild* (i) is at $2i + 1$ if $2i + 1 \le n$. If $2i + 1 > n$, then i has no right child.





root

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
-	*a*	*b*	*c*	*d*	*e*	*f*	*g*
	$\lfloor i/2 \rfloor$	i		$2i$	$2i$+1		



	tree
[0]	–
[1]	A
[2]	B
[3]	–
[4]	C
[5]	–
[6]	–
[7]	–
[8]	D
[9]	–
.	.
.	.
.	.
[16]	E

(a) Tree of Figure 5.10(a)

tree
–
A
B
C
D
E
F
G
H
I

(b) Tree of Figure 5.10(b)

Figure 5.12: Array representation of the binary trees of Figure 5.10

- **Linked Representation**

```
typedef struct node *treePointer;
typedef struct node {
          int data;
          treePointer leftChild, rightChild;
          } node;
```

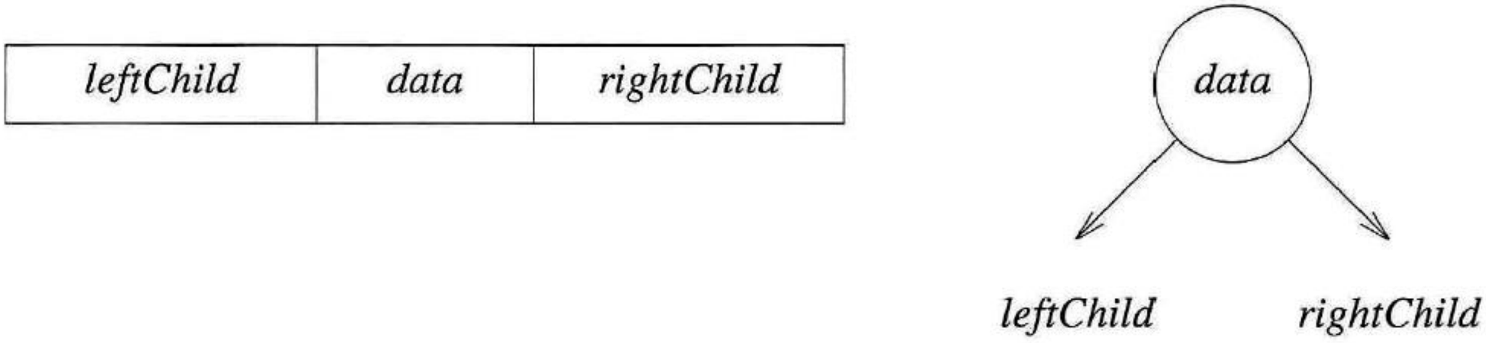


Figure 5.13: Node representations

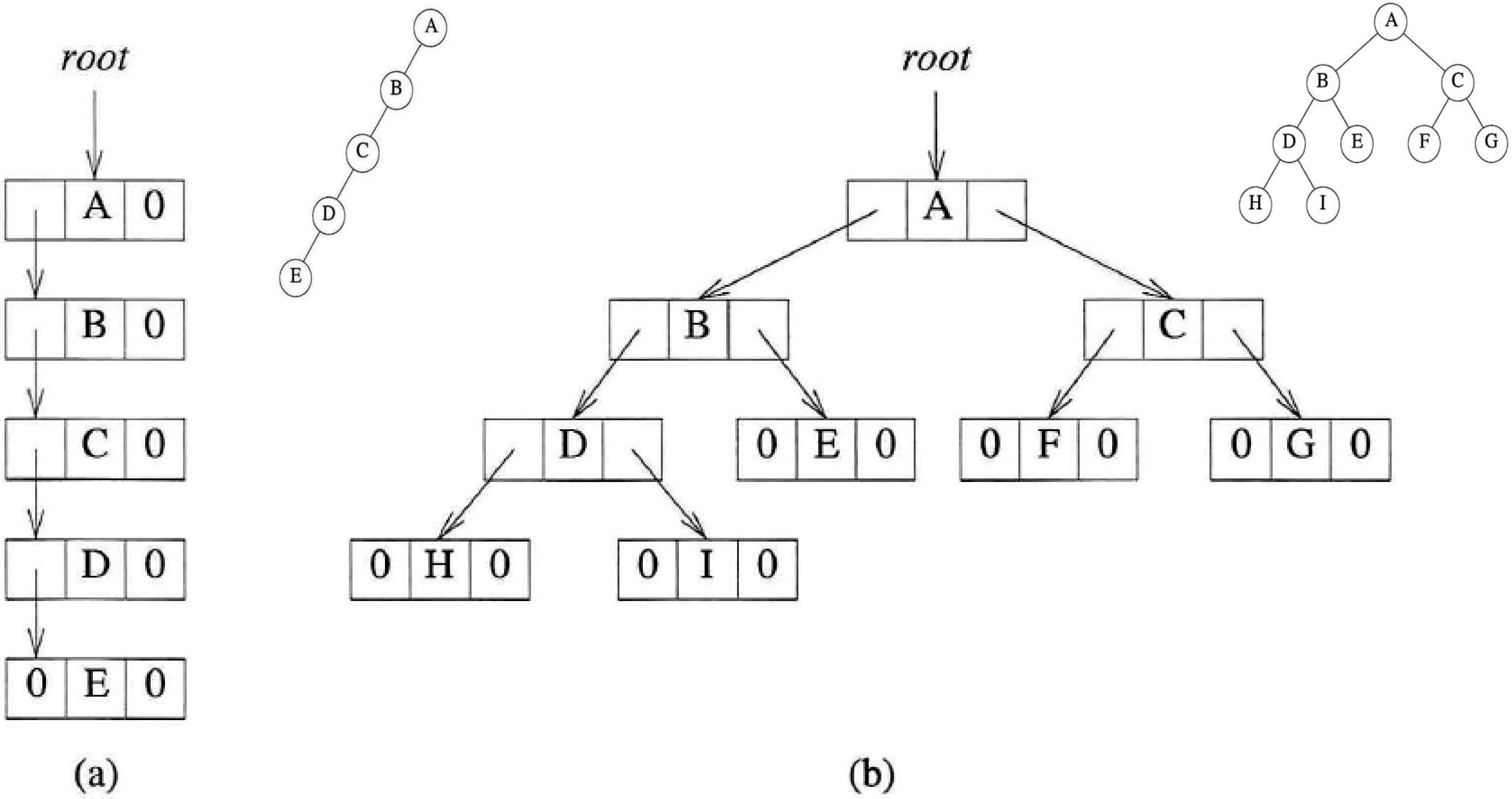


Figure 5.14: Linked representation for the binary trees of Figure 5.10

5.3 Binary Tree Traversal

- Traversing a tree
  - Visiting each node in the tree exactly once

- When traversing a binary tree,
  - L, V, R : *moving left*, *visiting the node*, *moving right*
  - Six possible combinations of traversal
    - LVR, LRV, VLR, VRL, RVL, RLV
  - If we traverse left before right, only tree remains
    - LVR: ***inorder***
    - LRV: ***postorder***
    - VLR: ***preorder***

- There is a natural correspondence between
  - *these traversals* and *producing the infix, postfix, and prefix forms of an expression*.

- Consider a binary tree for *A/B*C*D+E*
  - For each node that contains an operator,
    - its left subtree gives the left operand and
    - its right subtree the right operand.

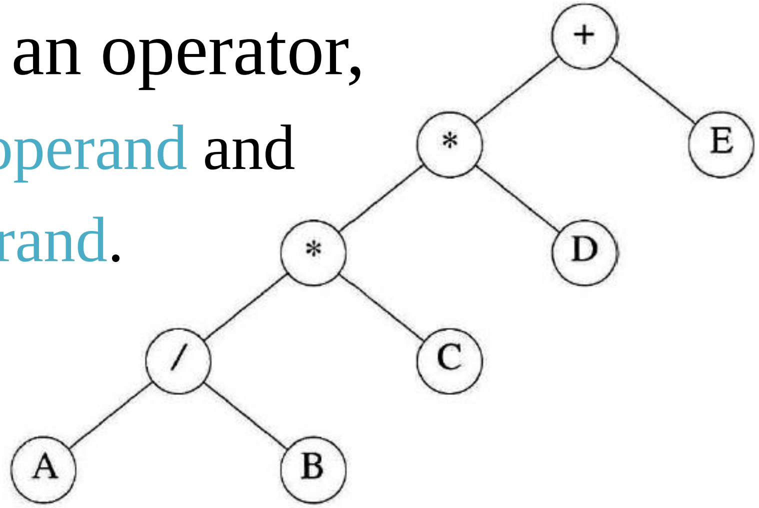
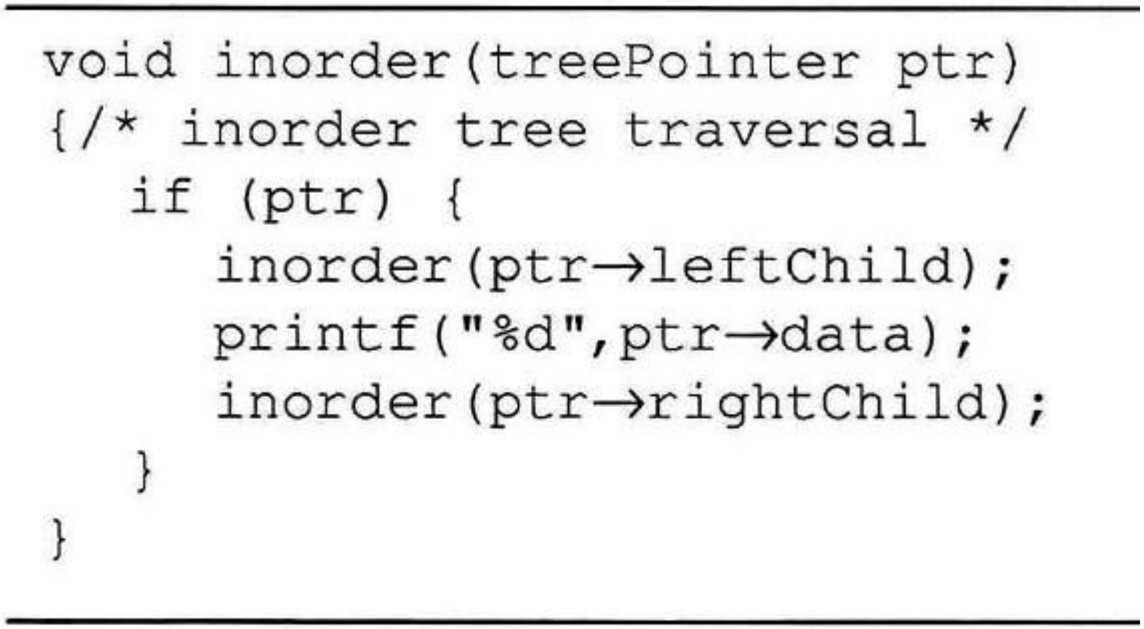


Figure 5.16: Binary tree with arithmetic expression

5.3.1 Inorder Traversal



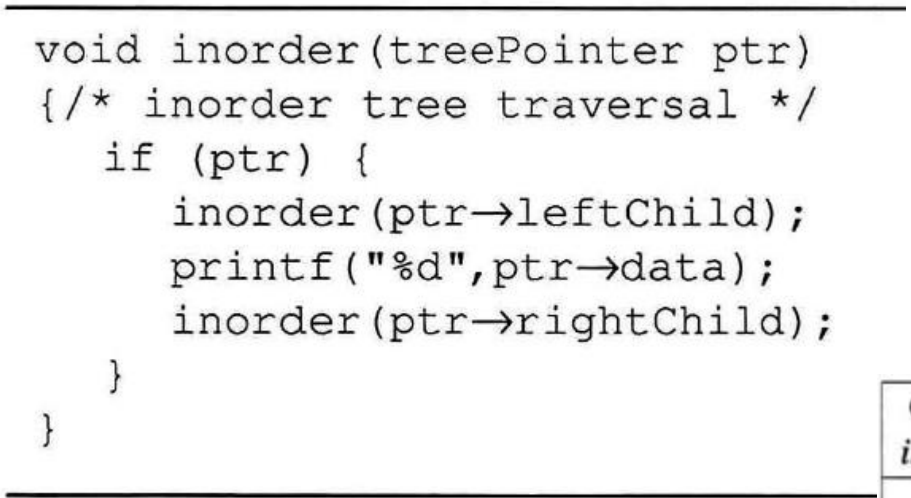
```
void inorder(treePointer ptr)
{/* inorder tree traversal */
   if (ptr) {
      inorder(ptr→leftChild);
      printf("%d",ptr→data);
      inorder(ptr→rightChild);
   }
}
```

Program 5.1: Inorder traversal of a binary tree

1. Return if the tree is null
2. Inorder traversal of the left subtree
3. Print the value
4. Inorder traversal of the right subtree

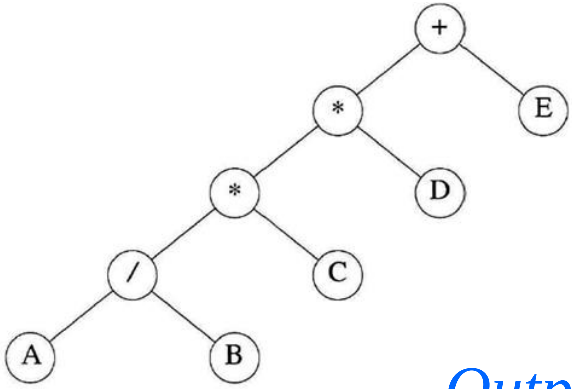
※ 보충자료 참고

Example



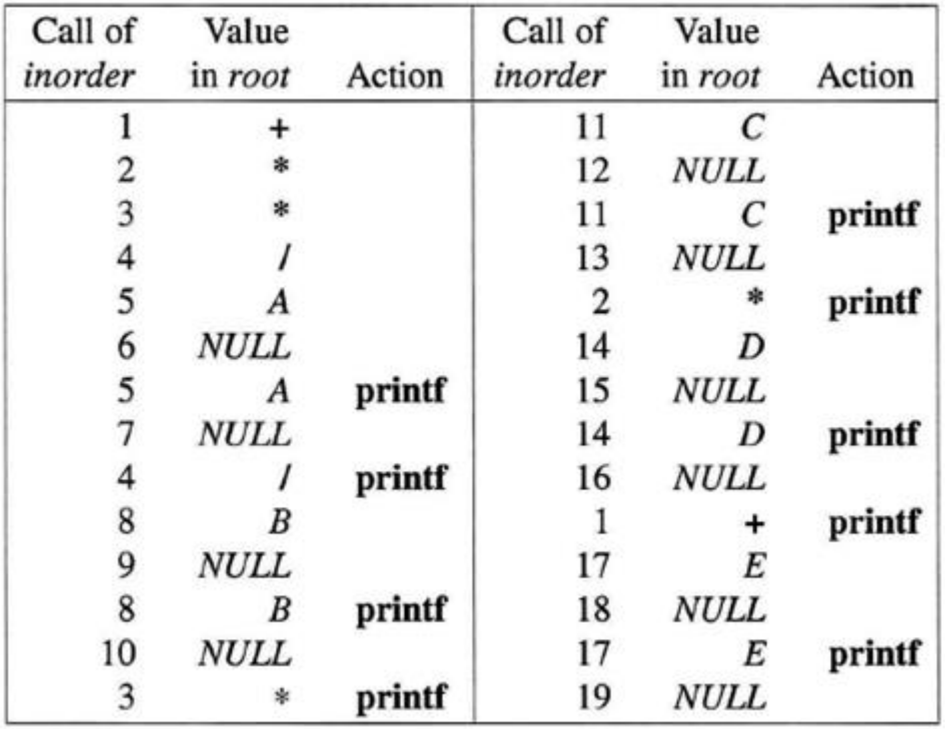
```
void inorder(treePointer ptr)
{/* inorder tree traversal */
   if (ptr) {
      inorder(ptr→leftChild);
      printf("%d",ptr→data);
      inorder(ptr→rightChild);
   }
}
```

Program 5.1: Inorder traversal of a binary tree



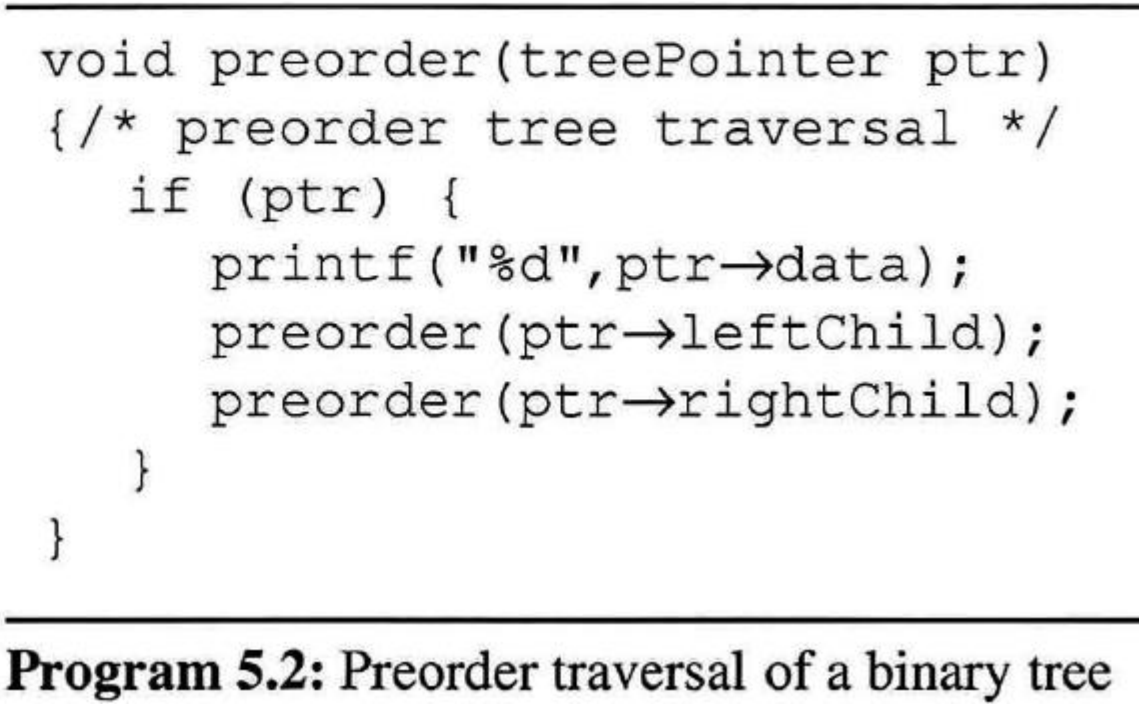
Output ?

A/B*C*D+E



Call of *inorder*	Value in *root*	Action	Call of *inorder*	Value in *root*	Action
1	+		11	*C*	
2	*		12	*NULL*	
3	*		11	*C*	**printf**
4	/		13	*NULL*	
5	*A*		2	*	**printf**
6	*NULL*		14	*D*	
5	*A*	**printf**	15	*NULL*	
7	*NULL*		14	*D*	**printf**
4	/	**printf**	16	*NULL*	
8	*B*		1	+	**printf**
9	*NULL*		17	*E*	
8	*B*	**printf**	18	*NULL*	
10	*NULL*		17	*E*	**printf**
3	*	**printf**	19	*NULL*	

5.3.2 Preorder Traversal

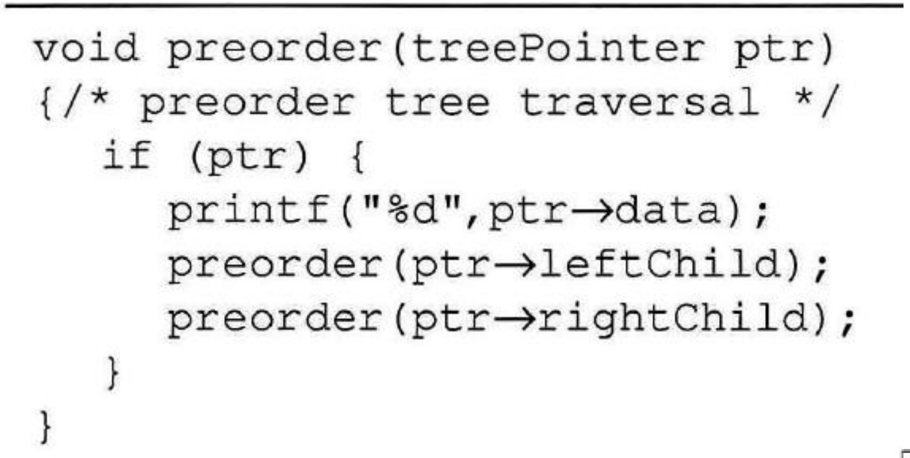


```
void preorder(treePointer ptr)
{/* preorder tree traversal */
   if (ptr) {
      printf("%d",ptr→data);
      preorder(ptr→leftChild);
      preorder(ptr→rightChild);
   }
}
```

Program 5.2: Preorder traversal of a binary tree

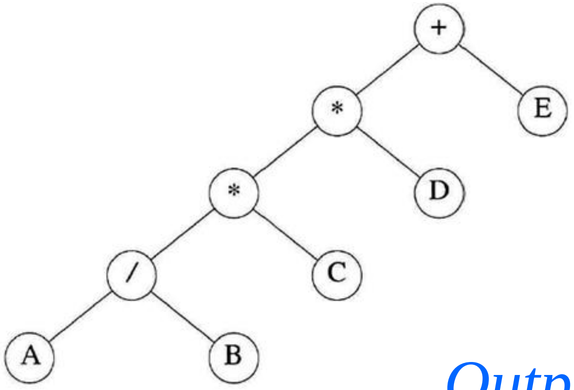
1. Return if the tree is null
2. Print the value
3. Preorder traversal of the left subtree
4. Preorder traversal of the right subtree

Example



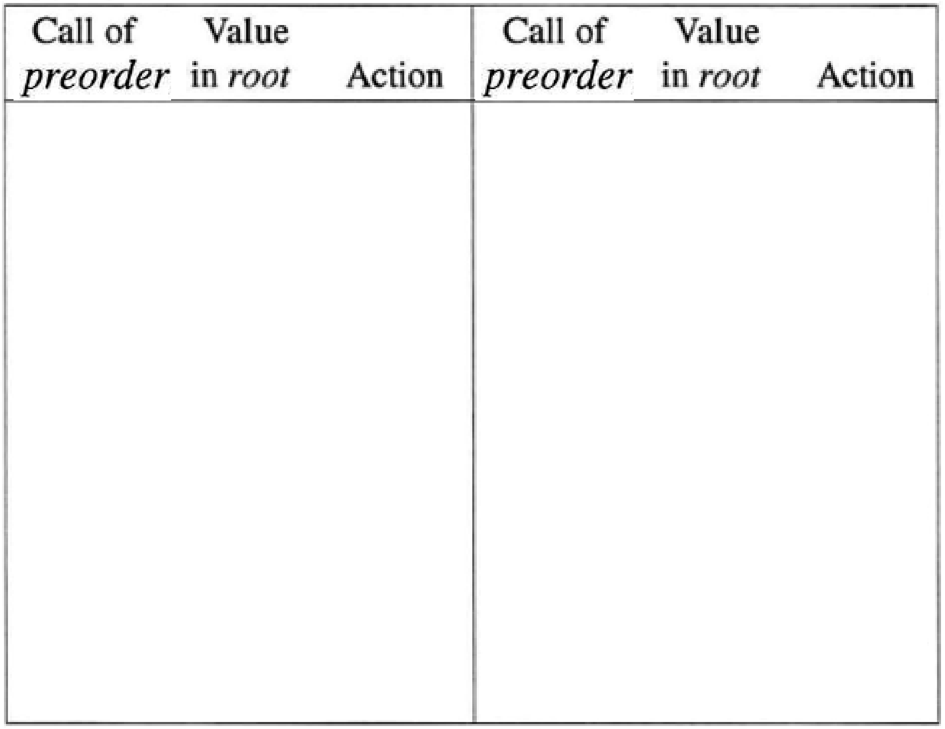
```
void preorder(treePointer ptr)
{/* preorder tree traversal */
   if (ptr) {
      printf("%d",ptr→data);
      preorder(ptr→leftChild);
      preorder(ptr→rightChild);
   }
}
```

Program 5.2: Preorder traversal of a binary tree



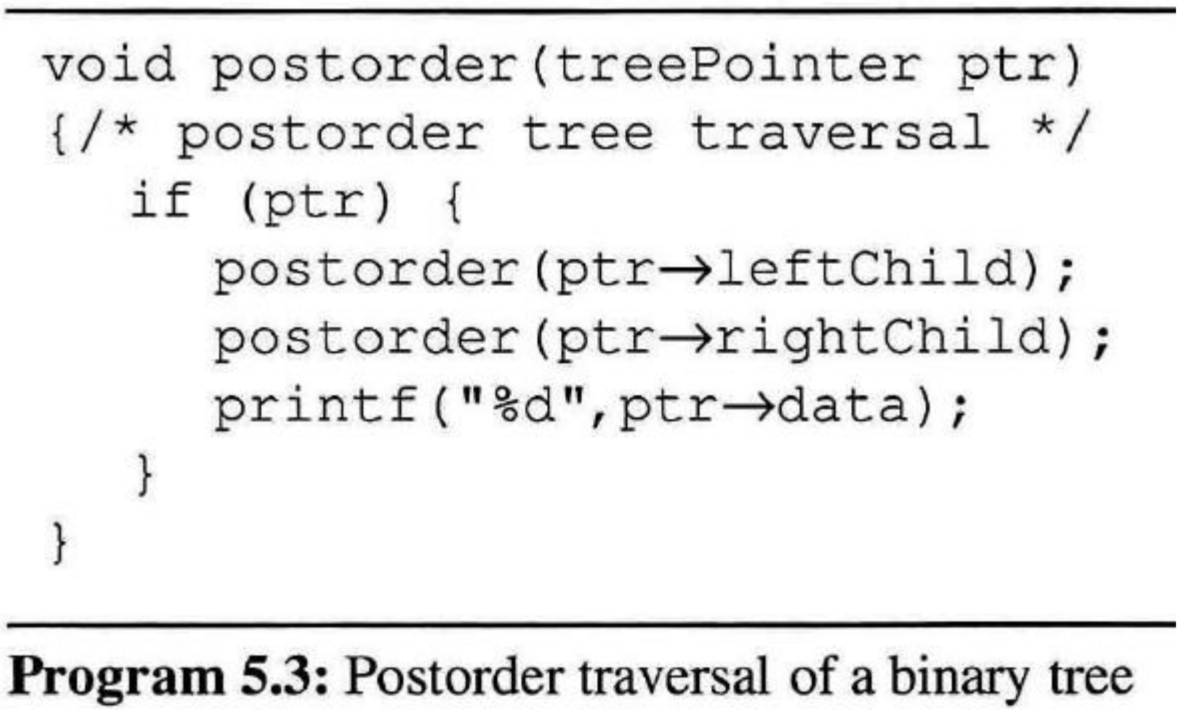
Output ?

+/ABCDE**



Call of *preorder*	Value in *root*	Action	Call of *preorder*	Value in *root*	Action

5.3.3 Postorder Traversal



```
void postorder(treePointer ptr)
{/* postorder tree traversal */
   if (ptr) {
      postorder(ptr→leftChild);
      postorder(ptr→rightChild);
      printf("%d",ptr→data);
   }
}
```

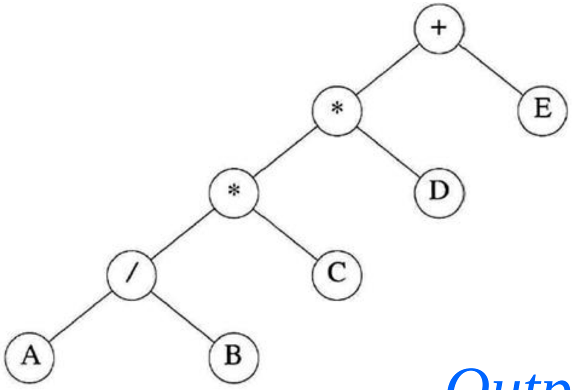
Program 5.3: Postorder traversal of a binary tree

1. Return if the tree is null
2. Postorder traversal of the left subtree
3. Postorder traversal of the right subtree
4. Print the value

Example

```
void postorder(treePointer ptr)
{/* postorder tree traversal */
   if (ptr) {
      postorder(ptr→leftChild);
      postorder(ptr→rightChild);
      printf("%d",ptr→data);
   }
}
```

Program 5.3: Postorder traversal of a binary tree



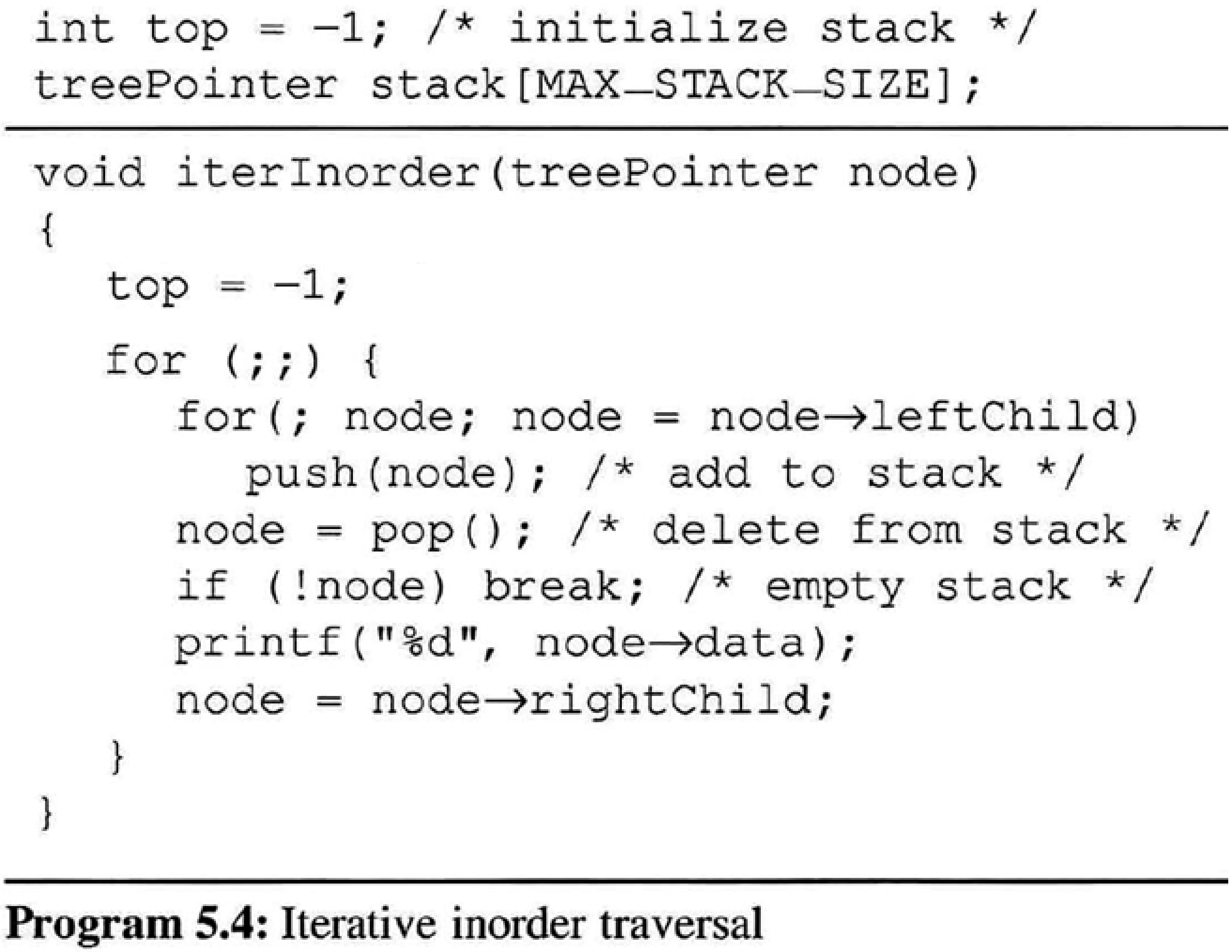
Output ?

AB/C*D*E+

Call of *postorder*	Value in *root*	Action	Call of *postorder*	Value in *root*	Action

5.3.4 Iterative Inorder Traversal

- We can develop equivalent iterative functions instead of using recursion.
- To simulate recursion, we must create *our own stack*.



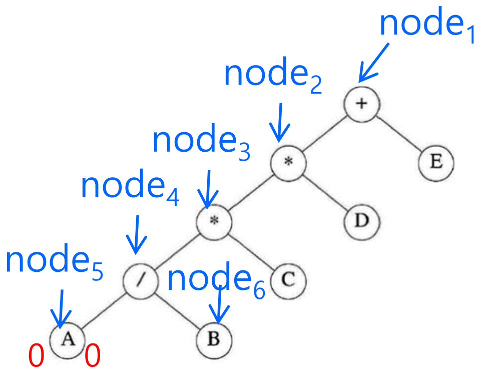
```
int top = -1; /* initialize stack */
treePointer stack[MAX_STACK_SIZE];
```

```
void iterInorder(treePointer node)
{
   top = -1;
   for (;;) {
      for(; node; node = node→leftChild)
         push(node); /* add to stack */
      node = pop(); /* delete from stack */
      if (!node) break; /* empty stack */
      printf("%d", node→data);
      node = node→rightChild;
   }
}
```

Program 5.4: Iterative inorder traversal

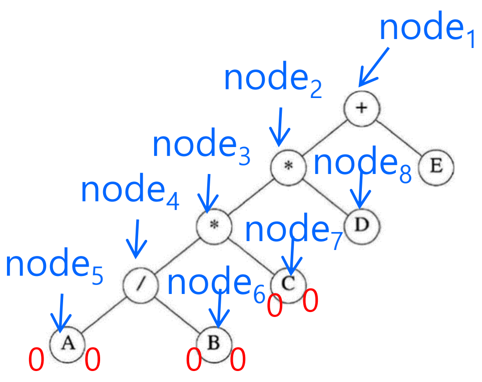
User-Defined Stack

```
void iterInorder(treePointer node)
{
  top = -1;
  for (;;) {
    for(; node; node = node→leftChild)
      push(node); /* add to stack */
    node = pop(); /* delete from stack */
    if (!node) break; /* empty stack */
    printf("%d", node→data);
    node = node→rightChild;
  }
}
```



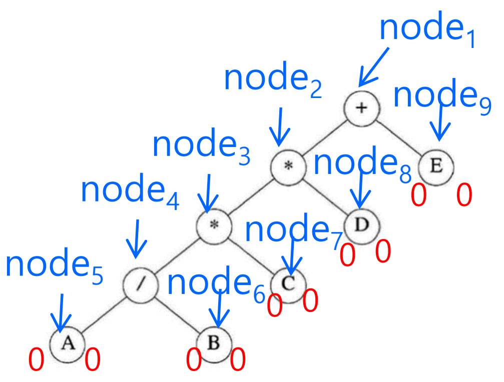
					$node_5$	$node_5$				
				$node_4$	$node_4$	$node_4$	$node_4$	$node_4$		$node_6$
			$node_3$	$node_3$	$node_3$	$node_3$	$node_3$	$node_3$	$node_3$	$node_3$
		$node_2$	$node_2$	$node_2$	$node_2$	$node_2$	$node_2$	$node_2$	$node_2$	$node_2$
stack	$node_1$	$node_1$	$node_1$	$node_1$	$node_1$	$node_1$	$node_1$	$node_1$	$node_1$	$node_1$
node	$node_1$	$node_2$	$node_3$	$node_4$	$node_5$	0	$node_5$	0	$node_4$	$node_6$
output							A		/	

```
void iterInorder(treePointer node)
{
  top = -1;
  for (;;) {
    for(; node; node = node→leftChild)
      push(node); /* add to stack */
    node = pop(); /* delete from stack */
    if (!node) break; /* empty stack */
    printf("%d", node→data);
    node = node→rightChild;
  }
}
```



stack	node$_6$ node$_3$ node$_2$ node$_1$	node$_3$ node$_2$ node$_1$	node$_3$ node$_2$ node$_1$	node$_2$ node$_1$	node$_7$ node$_2$ node$_1$	node$_7$ node$_2$ node$_1$	node$_2$ node$_1$	node$_2$ node$_1$	node$_1$	node$_8$ node$_1$
node	0	node$_6$	0	node$_3$	node$_7$	0	node$_7$	0	node$_2$	node$_8$
output		B		*			C		*	

```
void iterInorder(treePointer node)
{
  top = -1;
  for (;;) {
    for(; node; node = node→leftChild)
      push(node); /* add to stack */
    node = pop(); /* delete from stack */
    if (!node) break; /* empty stack */
    printf("%d", node→data);
    node = node→rightChild;
  }
}
```



stack	node$_8$ node$_1$	node$_1$	node$_1$		node$_9$	node$_9$			pop → returns null	
node	0	node$_8$	0	node$_1$	node$_9$	0	node$_9$	0		**NULL**
output		D		+			E			

The number of calls of push?

5.3.5 Level-Order Traversal

- A traversal that requires a *queue*.
- Visit the root first, the root's left child, followed by the root's right child
- Continue, visiting the node at each new level from the leftmost node to the rightmost node

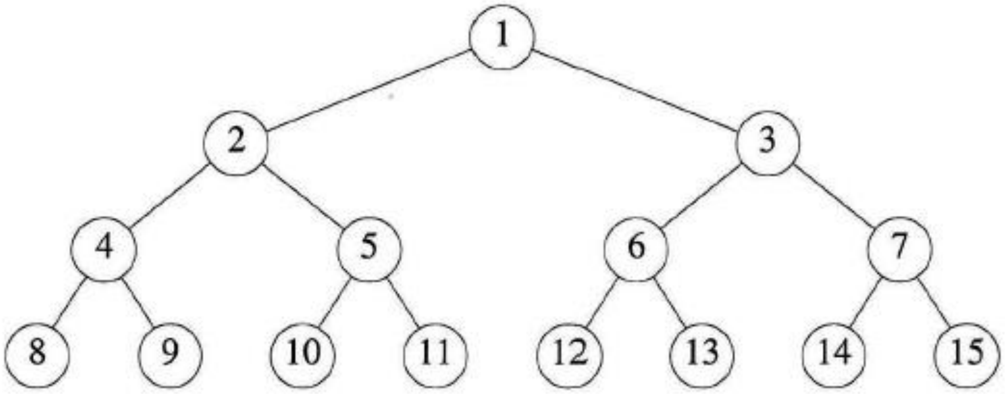
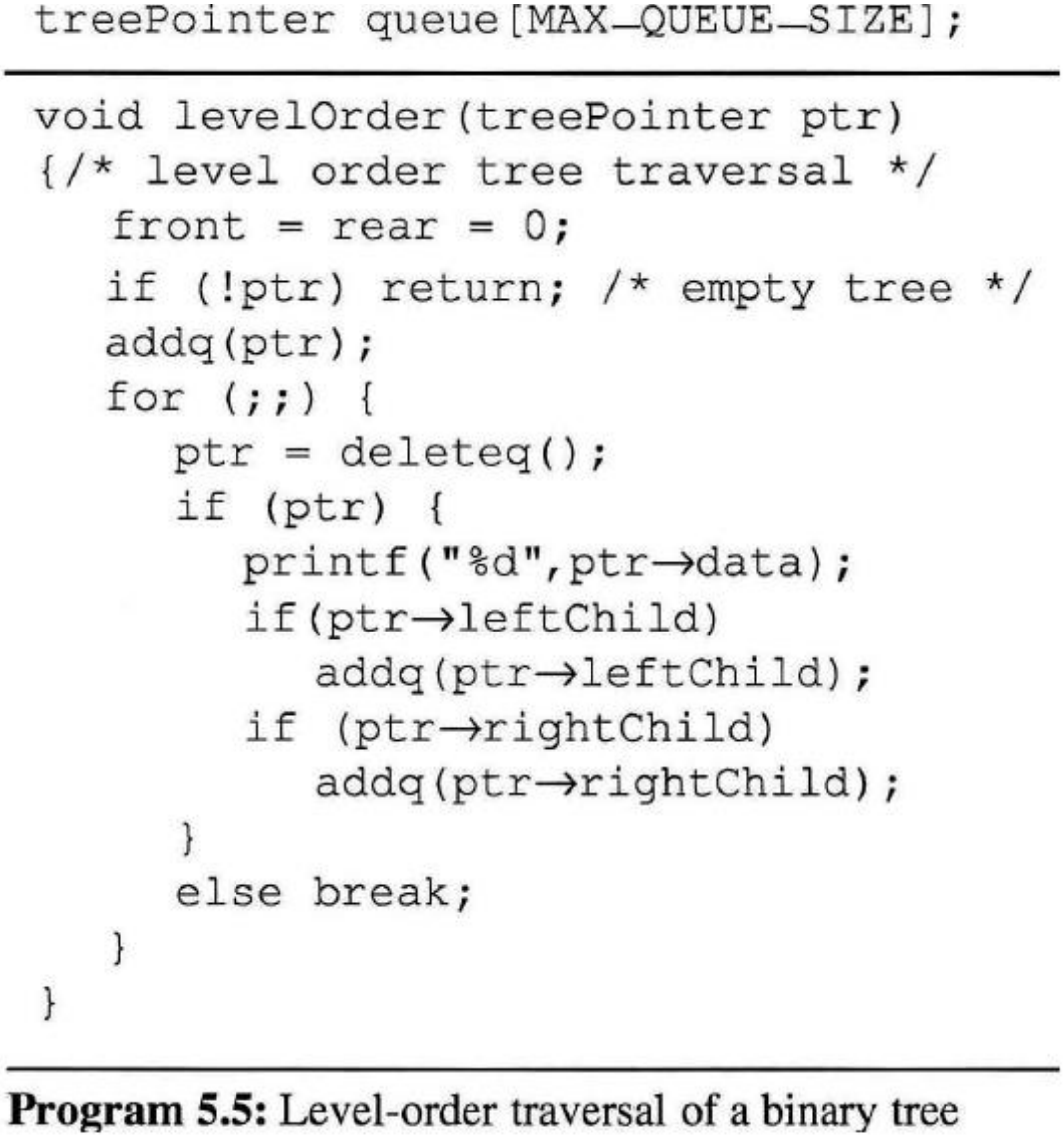


Figure 5.11: Full binary tree of depth 4 with sequential node numbers

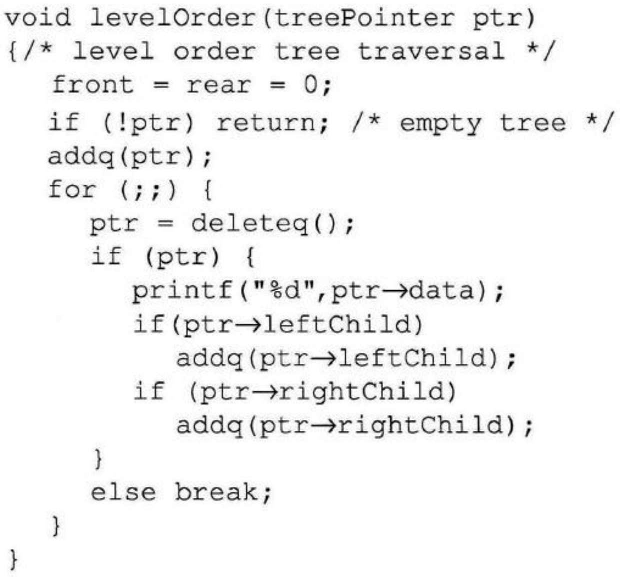
int front = 0, rear = 0; // circular queue;



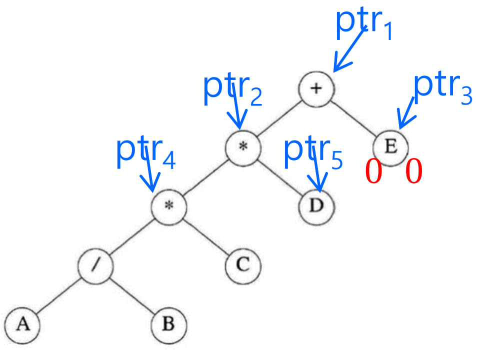
```
treePointer queue[MAX_QUEUE_SIZE];
```

```
void levelOrder(treePointer ptr)
{/* level order tree traversal */
   front = rear = 0;
   if (!ptr) return; /* empty tree */
   addq(ptr);
   for (;;) {
      ptr = deleteq();
      if (ptr) {
         printf("%d",ptr→data);
         if(ptr→leftChild)
            addq(ptr→leftChild);
         if (ptr→rightChild)
            addq(ptr→rightChild);
      }
      else break;
   }
}
```

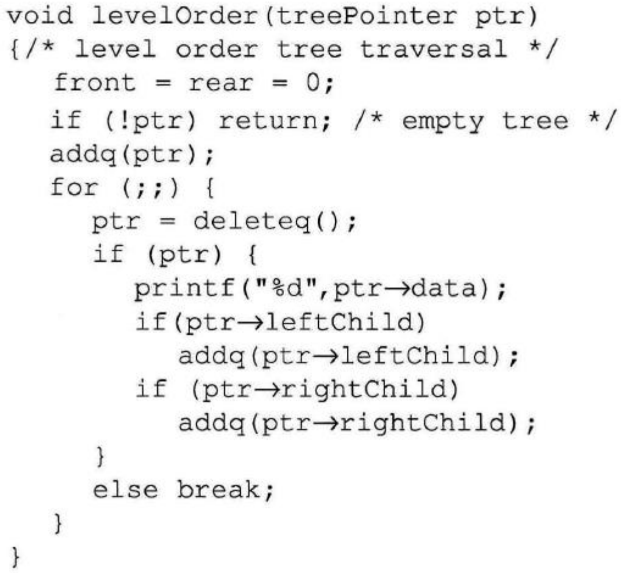
Program 5.5: Level-order traversal of a binary tree



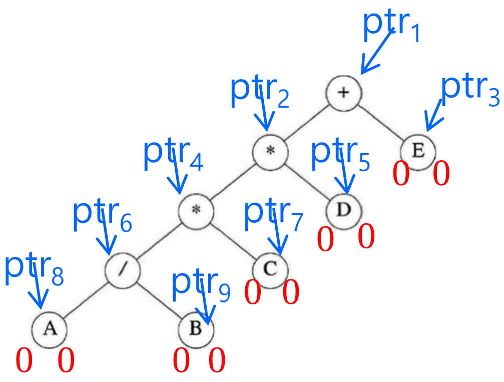
```
void levelOrder(treePointer ptr)
{/* level order tree traversal */
   front = rear = 0;
   if (!ptr) return; /* empty tree */
   addq(ptr);
   for (;;) {
      ptr = deleteq();
      if (ptr) {
         printf("%d",ptr→data);
         if(ptr→leftChild)
            addq(ptr→leftChild);
         if (ptr→rightChild)
            addq(ptr→rightChild);
      }
      else break;
   }
}
```

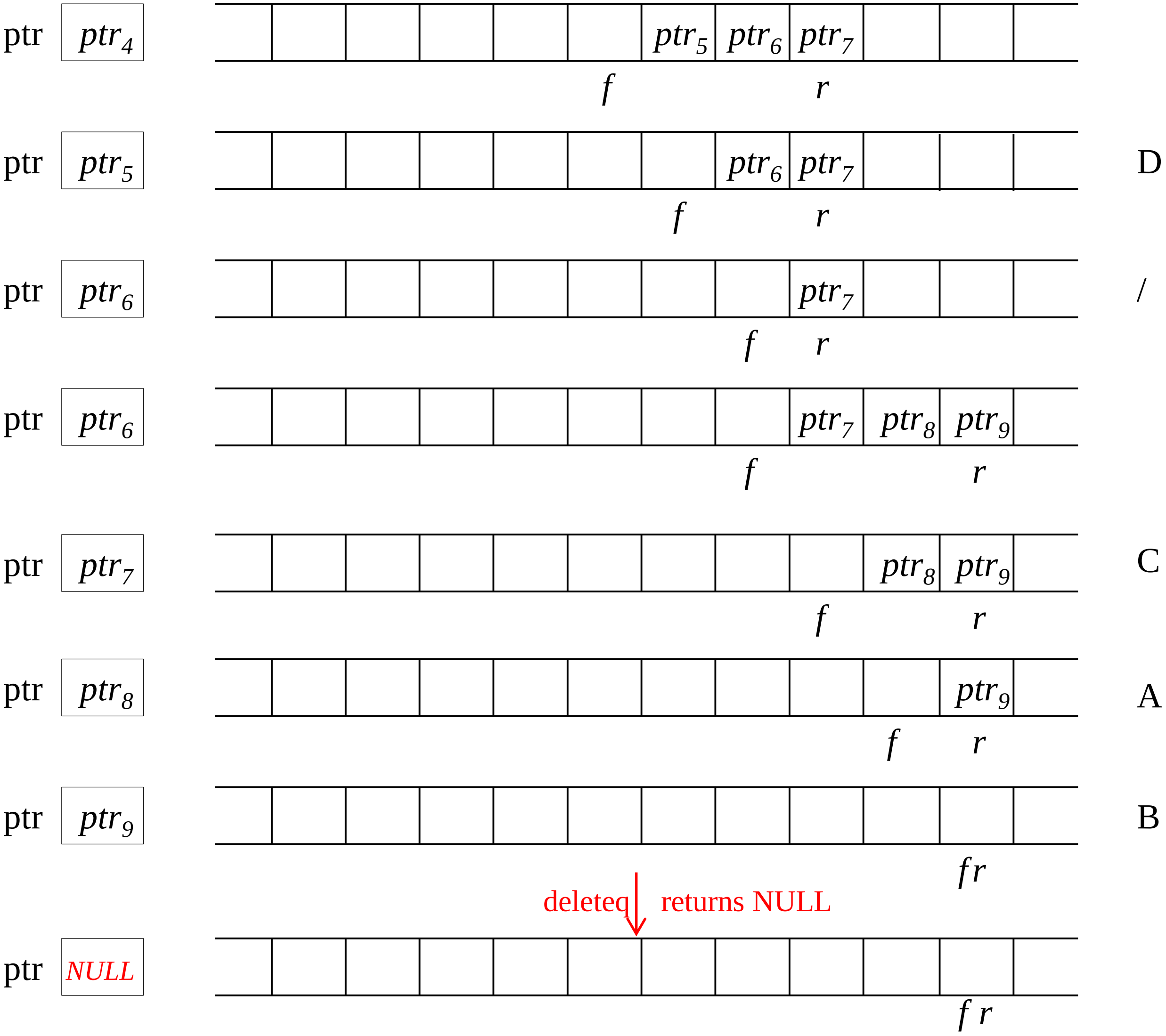


ptr	ptr_1													output
			f r											
ptr	ptr_1			ptr_1										
			f	r										
ptr	ptr_1													+
				fr										
ptr	ptr_1				ptr_2	ptr_3								
				f		r								
ptr	ptr_2					ptr_3								*
					f	r								
ptr	ptr_2					ptr_3	ptr_4	ptr_5						
					f			r						
ptr	ptr_3						ptr_4	ptr_5						E
						f		r						
ptr	ptr_4							ptr_5						*
							f	r						



```
void levelOrder(treePointer ptr)
{/* level order tree traversal */
   front = rear = 0;
   if (!ptr) return; /* empty tree */
   addq(ptr);
   for (;;) {
      ptr = deleteq();
      if (ptr) {
         printf("%d",ptr→data);
         if(ptr→leftChild)
            addq(ptr→leftChild);
         if (ptr→rightChild)
            addq(ptr→rightChild);
      }
      else break;
   }
}
```





ptr ptr_4 — ptr_5 ptr_6 ptr_7 — f r

ptr ptr_5 — ptr_6 ptr_7 — f r — D

ptr ptr_6 — ptr_7 — f r — /

ptr ptr_6 — ptr_7 ptr_8 ptr_9 — f r

ptr ptr_7 — ptr_8 ptr_9 — f r — C

ptr ptr_8 — ptr_9 — f r — A

ptr ptr_9 — $f r$ — B

deleteq returns NULL

ptr *NULL* — f r