
Chap 5. Trees (2)

Contents

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***

5.3 Binary Tree Traversal

- **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.

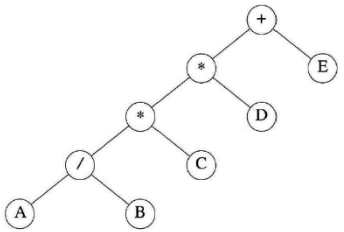


Figure 5.16: Binary tree with arithmetic expression

5.3.1 Inorder Traversal

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

```
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

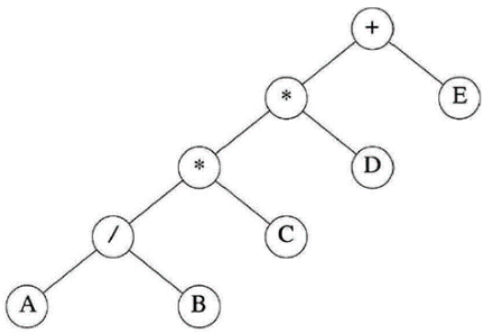
5.3.1 Inorder Traversal

```
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

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*	

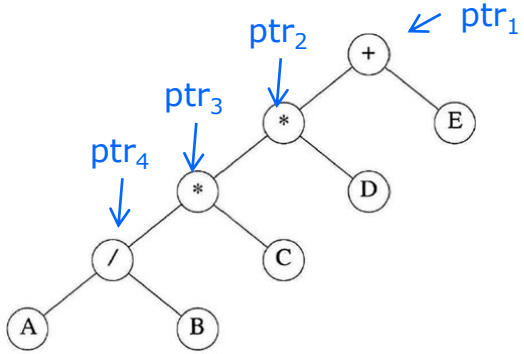
Trace of Program 5.1

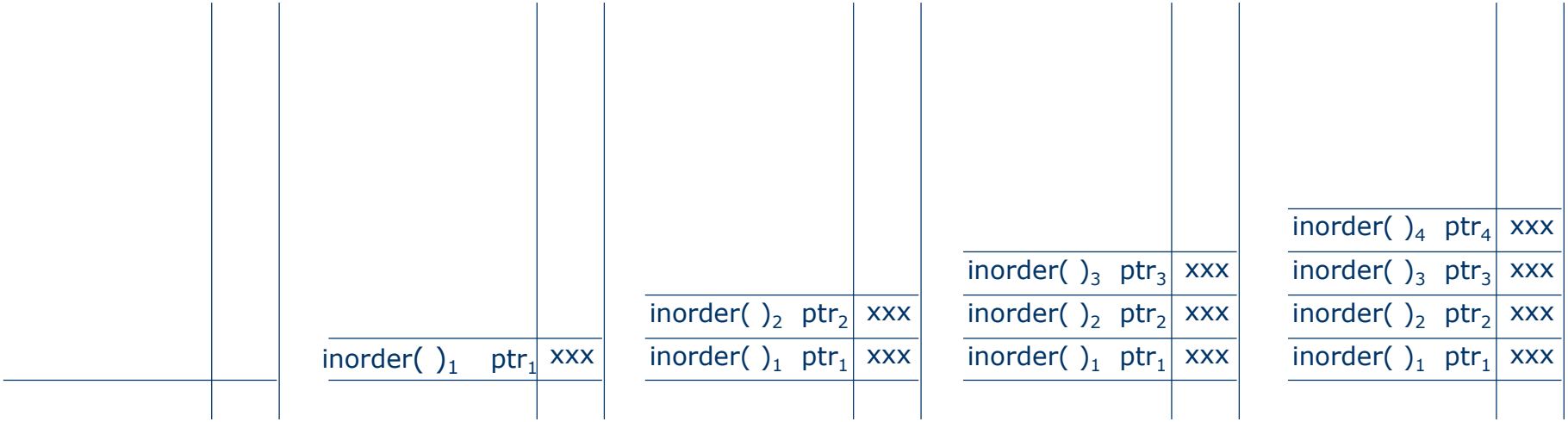


A/B*C*D+E

5.3.1 Inorder Traversal

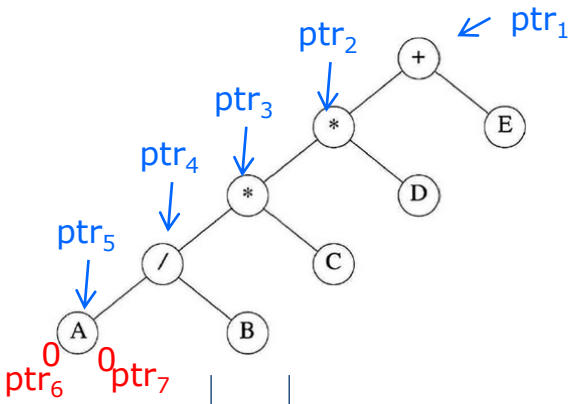
```
void inorder(treePointer ptr)
{/* inorder tree traversal */
   if (ptr) {
    ① inorder(ptr→leftChild);
      printf("%d",ptr→data);
    ② inorder(ptr→rightChild);
   }
}
```





5.3.1 Inorder Traversal

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



inorder( )5	ptr5	xxx
inorder( )4	ptr4	xxx
inorder( )3	ptr3	xxx
inorder( )2	ptr2	xxx
inorder( )1	ptr1	xxx

inorder( )6	ptr6	**0**
inorder( )5	ptr5	xxx
inorder( )4	ptr4	xxx
inorder( )3	ptr3	xxx
inorder( )2	ptr2	xxx
inorder( )1	ptr1	xxx

inorder( )5	ptr5	xxx
inorder( )4	ptr4	xxx
inorder( )3	ptr3	xxx
inorder( )2	ptr2	xxx
inorder( )1	ptr1	xxx

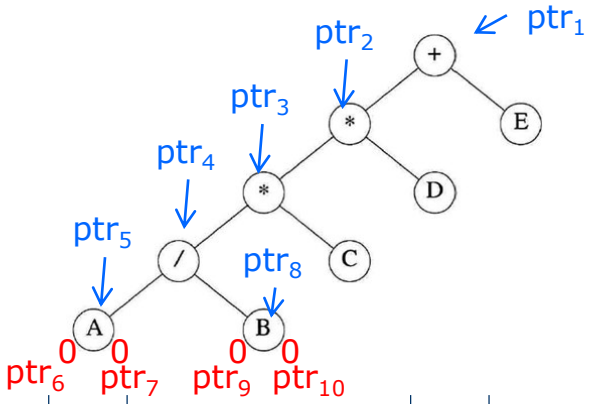
inorder( )7	ptr7	**0**
inorder( )5	ptr5	xxx
inorder( )4	ptr4	xxx
inorder( )3	ptr3	xxx
inorder( )2	ptr2	xxx
inorder( )1	ptr1	xxx

inorder( )5	ptr5	xxx
inorder( )4	ptr4	xxx
inorder( )3	ptr3	xxx
inorder( )2	ptr2	xxx
inorder( )1	ptr1	xxx

Output: A

5.3.1 Inorder Traversal

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



inorder( )$_4$	ptr$_4$	xxx
inorder( )$_3$	ptr$_3$	xxx
inorder( )$_2$	ptr$_2$	xxx
inorder( )$_1$	ptr$_1$	xxx

inorder( )$_8$	ptr$_8$	xxx
inorder( )$_4$	ptr$_4$	xxx
inorder( )$_3$	ptr$_3$	xxx
inorder( )$_2$	ptr$_2$	xxx
inorder( )$_1$	ptr$_1$	xxx

inorder( )$_9$	ptr$_9$	**0**
inorder( )$_8$	ptr$_8$	xxx
inorder( )$_4$	ptr$_4$	xxx
inorder( )$_3$	ptr$_3$	xxx
inorder( )$_2$	ptr$_2$	xxx
inorder( )$_1$	ptr$_1$	xxx

Output: /

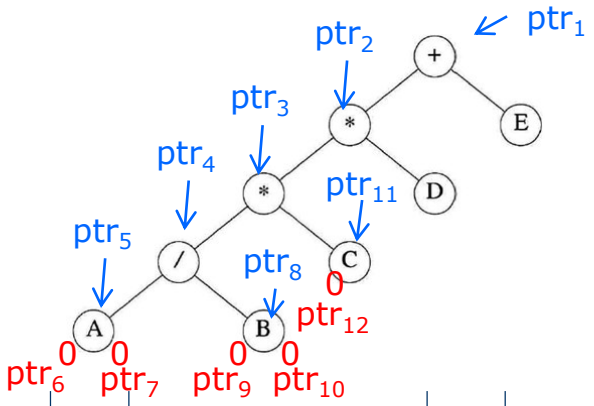
inorder( )$_8$	ptr$_8$	xxx
inorder( )$_4$	ptr$_4$	xxx
inorder( )$_3$	ptr$_3$	xxx
inorder( )$_2$	ptr$_2$	xxx
inorder( )$_1$	ptr$_1$	xxx

inorder( )$_{10}$	ptr$_{10}$	**0**
inorder( )$_8$	ptr$_8$	xxx
inorder( )$_4$	ptr$_4$	xxx
inorder( )$_3$	ptr$_3$	xxx
inorder( )$_2$	ptr$_2$	xxx
inorder( )$_1$	ptr$_1$	xxx

Output: B

5.3.1 Inorder Traversal

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



inorder( )$_8$	ptr$_8$	xxx
inorder( )$_4$	ptr$_4$	xxx
inorder( )$_3$	ptr$_3$	xxx
inorder( )$_2$	ptr$_2$	xxx
inorder( )$_1$	ptr$_1$	xxx

inorder( )$_4$	ptr$_4$	xxx
inorder( )$_3$	ptr$_3$	xxx
inorder( )$_2$	ptr$_2$	xxx
inorder( )$_1$	ptr$_1$	xxx

inorder( )$_3$	ptr$_3$	xxx
inorder( )$_2$	ptr$_2$	xxx
inorder( )$_1$	ptr$_1$	xxx

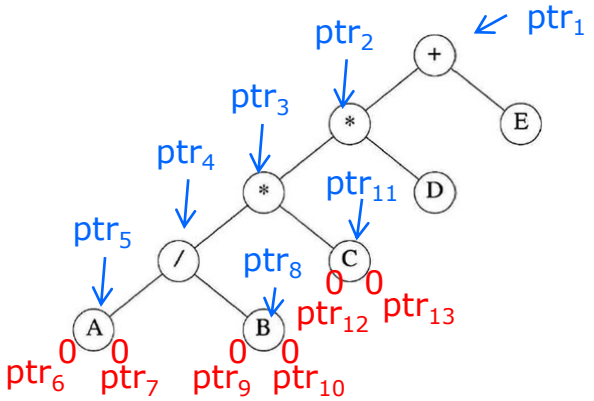
inorder( )$_{11}$	ptr$_{11}$	xxx
inorder( )$_3$	ptr$_3$	xxx
inorder( )$_2$	ptr$_2$	xxx
inorder( )$_1$	ptr$_1$	xxx

inorder( )$_{12}$	ptr$_{12}$	**0**
inorder( )$_{11}$	ptr$_{11}$	xxx
inorder( )$_3$	ptr$_3$	xxx
inorder( )$_2$	ptr$_2$	xxx
inorder( )$_1$	ptr$_1$	xxx

Output: *

5.3.1 Inorder Traversal

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



inorder( )$_{11}$ ptr$_{11}$	xxx
inorder( )$_3$ ptr$_3$	xxx
inorder( )$_2$ ptr$_2$	xxx
inorder( )$_1$ ptr$_1$	xxx

inorder( )$_{13}$ ptr$_{13}$	**0**
inorder( )$_{11}$ ptr$_{11}$	xxx
inorder( )$_3$ ptr$_3$	xxx
inorder( )$_2$ ptr$_2$	xxx
inorder( )$_1$ ptr$_1$	xxx

inorder( )$_{11}$ ptr$_{11}$	xxx
inorder( )$_3$ ptr$_3$	xxx
inorder( )$_2$ ptr$_2$	xxx
inorder( )$_1$ ptr$_1$	xxx

inorder( )$_3$ ptr$_3$	xxx
inorder( )$_2$ ptr$_2$	xxx
inorder( )$_1$ ptr$_1$	xxx

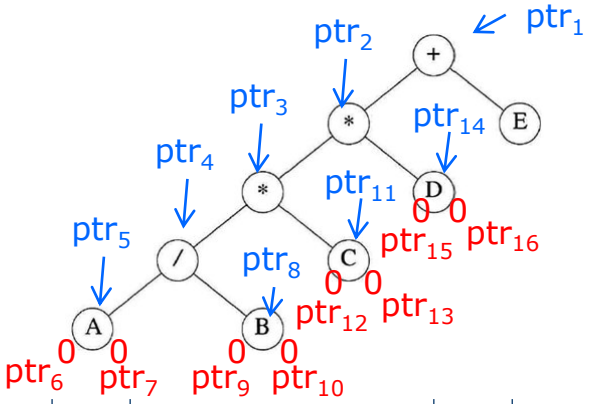
inorder( )$_2$ ptr$_2$	xxx
inorder( )$_1$ ptr$_1$	xxx

Output: C

Output: *

5.3.1 Inorder Traversal

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



inorder( )$_{14}$ptr$_{14}$	xxx
inorder( )$_2$ ptr$_2$	xxx
inorder( )$_1$ ptr$_1$	xxx

inorder( )$_{15}$ptr$_{15}$	**0**
inorder( )$_{14}$ptr$_{14}$	xxx
inorder( )$_2$ ptr$_2$	xxx
inorder( )$_1$ ptr$_1$	xxx

inorder( )$_{14}$ptr$_{14}$	xxx
inorder( )$_2$ ptr$_2$	xxx
inorder( )$_1$ ptr$_1$	xxx

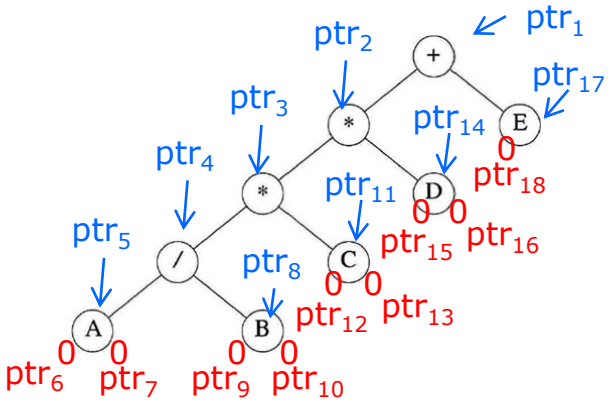
inorder( )$_{16}$ptr$_{16}$	**0**
inorder( )$_{14}$ptr$_{14}$	xxx
inorder( )$_2$ ptr$_2$	xxx
inorder( )$_1$ ptr$_1$	xxx

inorder( )$_{14}$ptr$_{14}$	xxx
inorder( )$_2$ ptr$_2$	xxx
inorder( )$_1$ ptr$_1$	xxx

Output: D

5.3.1 Inorder Traversal

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



inorder( )$_2$	ptr$_2$	xxx
inorder( )$_1$	ptr$_1$	xxx

inorder( )$_1$	ptr$_1$	xxx

inorder( )$_{17}$ptr$_{17}$		xxx
inorder( )$_1$	ptr$_1$	xxx

inorder( )$_{18}$ptr$_{18}$		**0**
inorder( )$_{17}$ptr$_{17}$		xxx
inorder( )$_1$	ptr$_1$	xxx

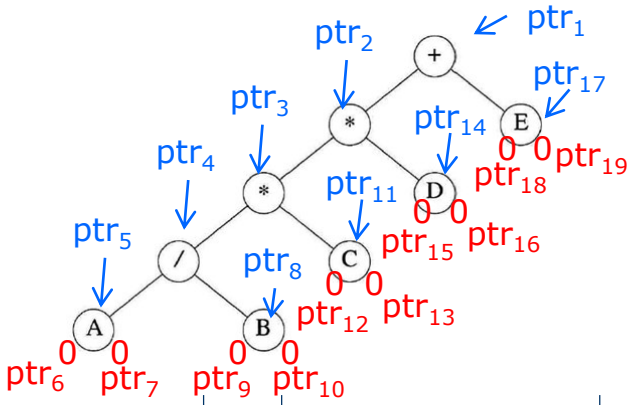
inorder( )$_{17}$ptr$_{17}$		xxx
inorder( )$_1$	ptr$_1$	xxx

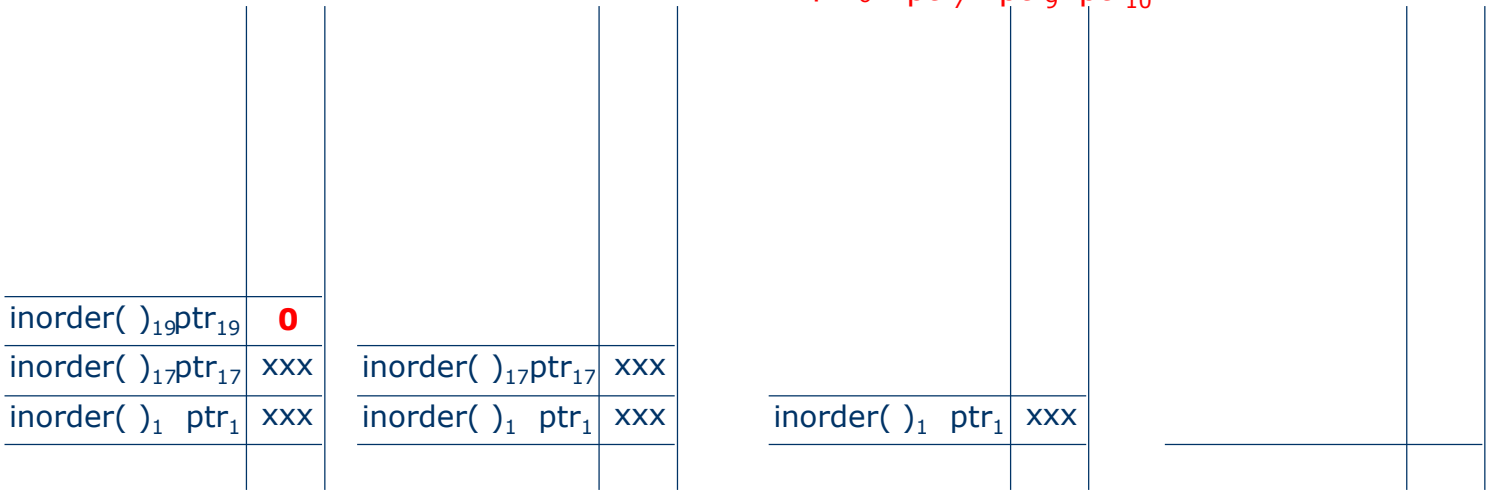
Output: +

Output: E

5.3.1 Inorder Traversal

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





The number of calls of inorder?

19

5.3.2 Preorder Traversal

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

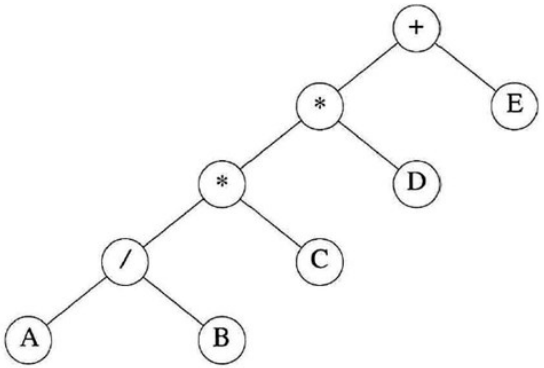
```
void preorder(treePointer ptr)
{/* preorder tree traversal */
   if (ptr) {
      printf("%d",ptr→data);
      preorder(ptr→leftChild);
      preorder(ptr→rightChild);
   }
}
```

Program 5.2: Preorder traversal of a binary tree

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



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

+/ABCDE**

5.3.3 Postorder Traversal

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

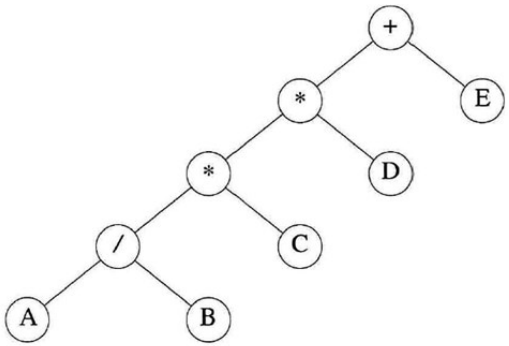
```
void postorder(treePointer ptr)
{/* postorder tree traversal */
   if (ptr) {
      postorder(ptr→leftChild);
      postorder(ptr→rightChild);
      printf("%d",ptr→data);
   }
}
```

Program 5.3: Postorder traversal of a binary tree

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



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

AB/C*D*E+

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*.

5.3.4 Iterative Inorder Traversal

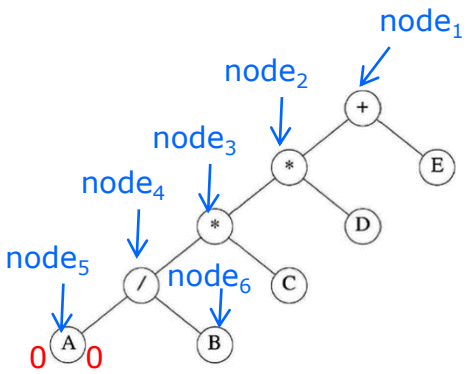
```
int top = -1; /* initialize stack */
treePointer stack[MAX-STACK-SIZE];

void iterinorder(treePointer node)
{
    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

5.3.4 Iterative Inorder Traversal

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

Program 5.4: Iterative inorder traversal

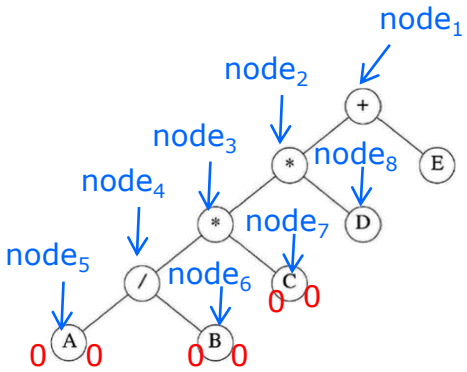


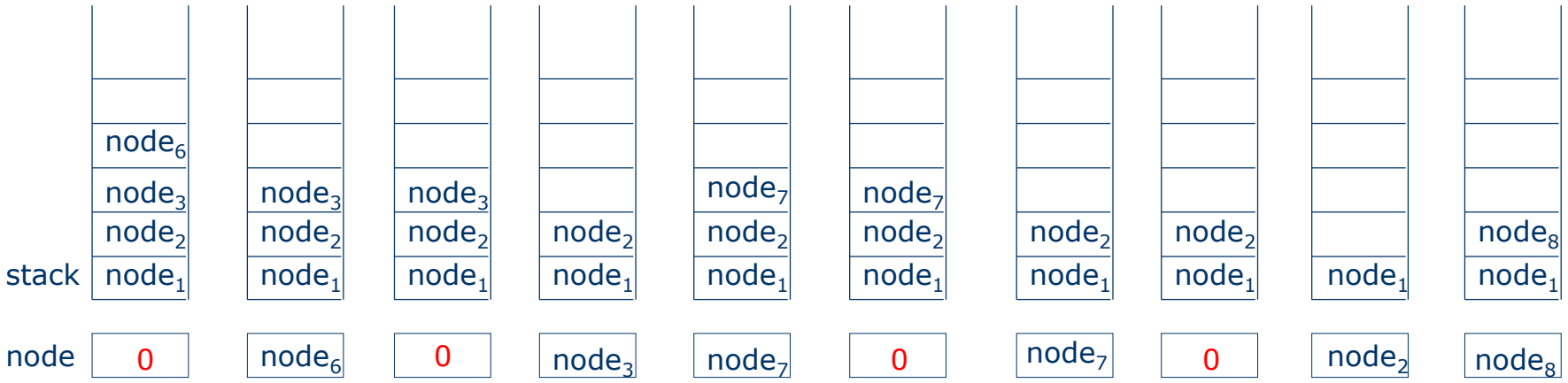
stack	node$_1$	node$_1$ node$_2$	node$_1$ node$_2$ node$_3$	node$_1$ node$_2$ node$_3$ node$_4$	node$_1$ node$_2$ node$_3$ node$_4$ node$_5$	node$_1$ node$_2$ node$_3$ node$_4$ node$_5$	node$_1$ node$_2$ node$_3$ node$_4$	node$_1$ node$_2$ node$_3$ node$_4$	node$_1$ node$_2$ node$_3$ node$_6$
node	node$_1$	node$_2$	node$_3$	node$_4$	node$_5$	0	node$_5$	0	node$_4$	node$_6$
output							A		/	

5.3.4 Iterative Inorder Traversal

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

Program 5.4: Iterative inorder traversal

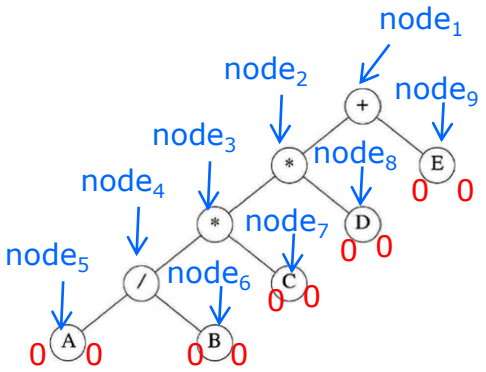


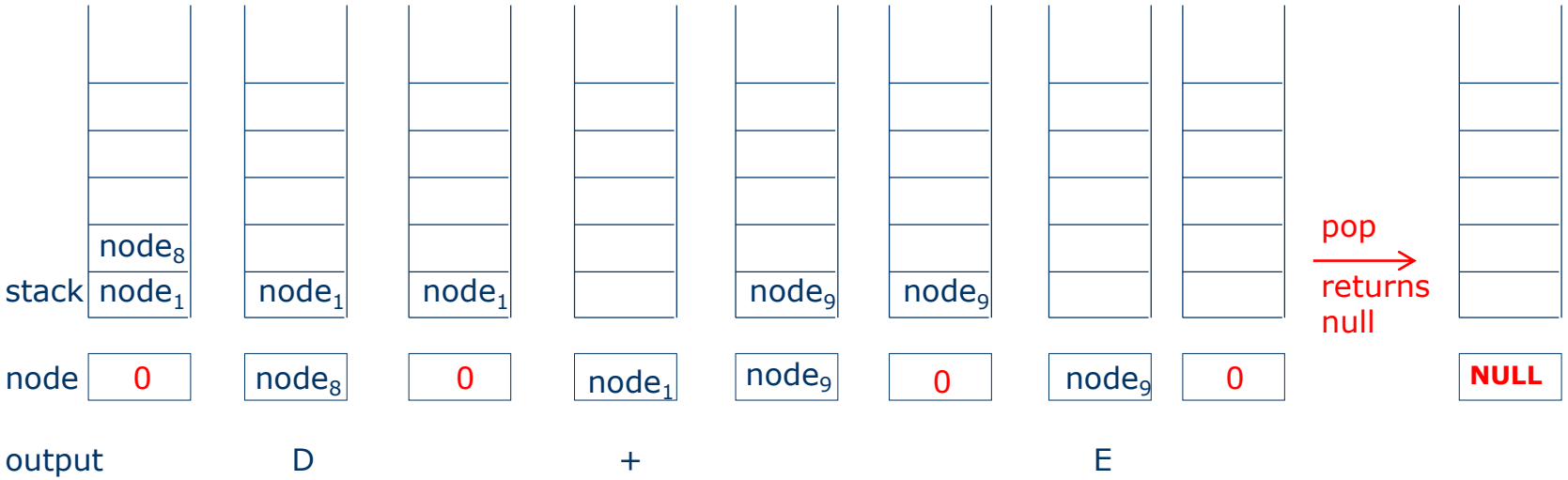


5.3.4 Iterative Inorder Traversal

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

Program 5.4: Iterative inorder traversal





The number of calls of push?

Binary tree for postfix expression

1. Scan the expression from left to right
2. If the token is an operand,

   Create a node with the value and NULL link.

   Push the node onto the stack.
3. If the token is an operator,

   Create a node for the operator.

   Pop a node from the stack and set it as the right child of the operator node.

   Pop another node from the stack and set it as the left child of the operator node.

   Push the operator node onto the stack.
4. The last remaining node on the stack is the root node.

*AB/C*D*E+*

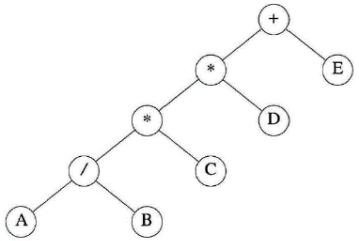


Figure 5.16: Binary tree with arithmetic expression

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

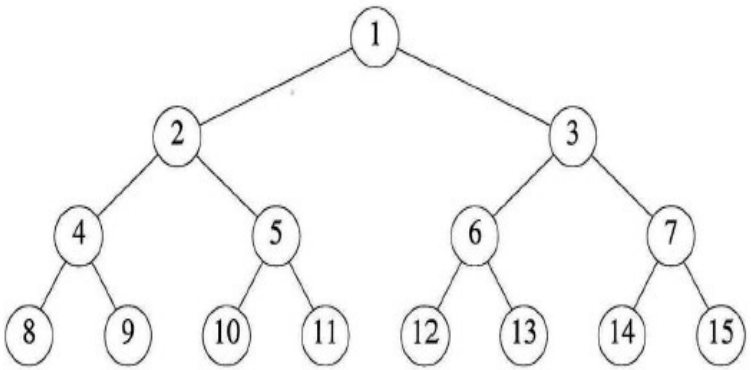


Figure 5.11: Full binary tree of depth 4 with sequential node numbers

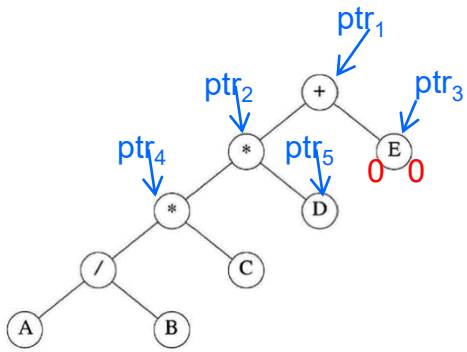
5.3.5 Level-Order Traversal

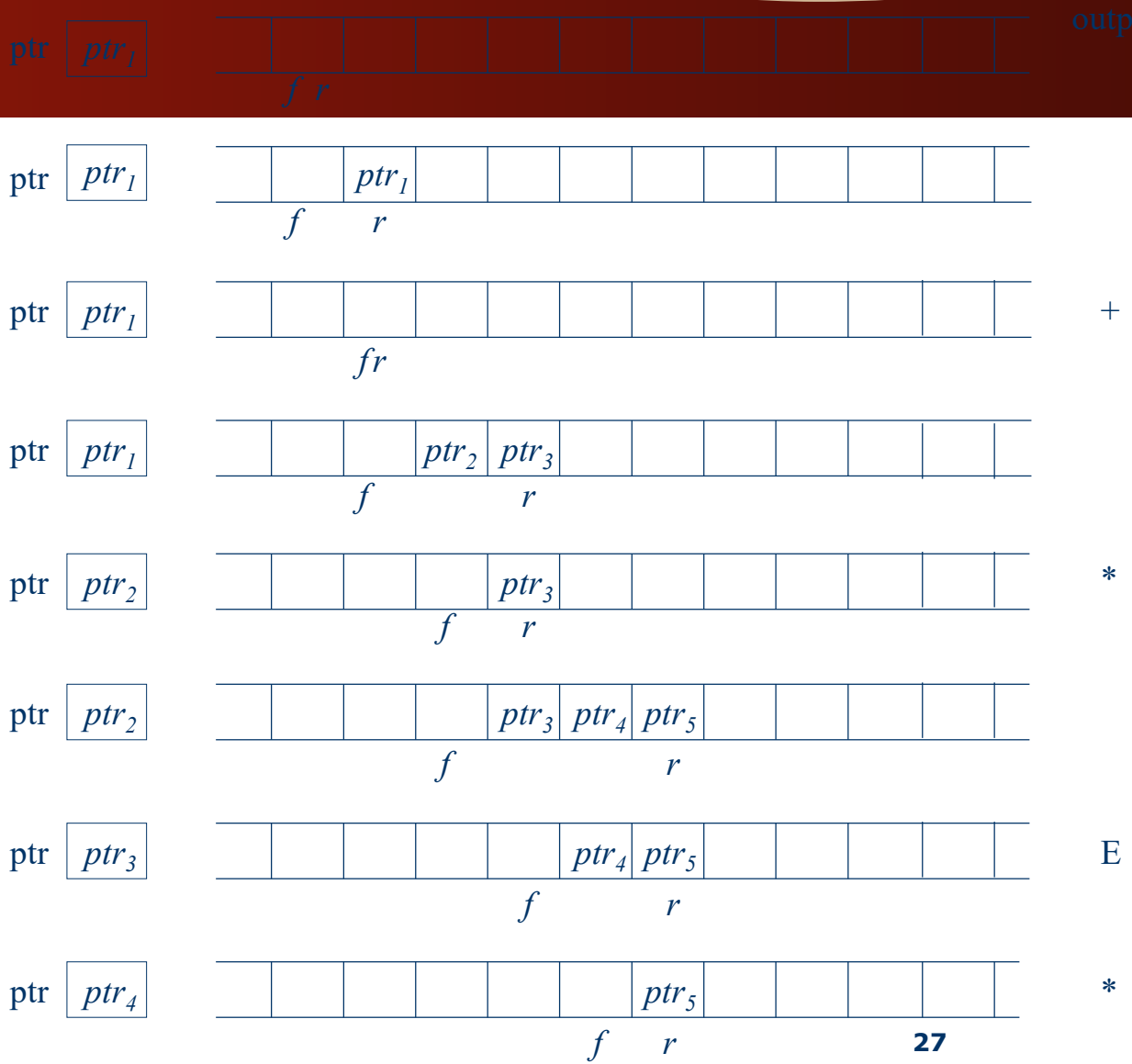
```
int front = rear = 0;
void levelOrder(treePointer ptr)
{/ * level order tree traversal */
        treePointer queue[MAX-QUEUE-SIZE];
        if (!ptr) return; / *empty tree* /
        addq (ptr);
        for (;;) {
                ptr = deleteq();
                if (ptr) {
                        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

```
int front = rear = 0;
void levelOrder(treePointer ptr)
{/ * level order tree traversal */
        treePointer queue[MAX-QUEUE-SIZE];
        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





```
int front = rear = 0;
void levelOrder(treePointer ptr)
{/ * level order tree traversal */
        treePointer queue[MAX-QUEUE-SIZE];
        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