

Tree Traversals

Joseph Chuang-Chieh Lin (林莊傑)

Department of Computer Science & Engineering,
National Taiwan Ocean University

Fall 2024

Outline

- 1 Binary Tree Traversals (Using a Stack)
- 2 Level-Order Traversal
- 3 Additional Binary Tree Operations

Outline

1 Binary Tree Traversals (Using a Stack)

2 Level-Order Traversal

3 Additional Binary Tree Operations

Binary Tree Traversals

Question

How to visit each node of a tree exactly once?

Binary Tree Traversals

Question

How to visit each node of a tree exactly once?

- Let V , L , R stand for visiting the node, moving left, and moving right, resp.

Binary Tree Traversals

Question

How to visit each node of a tree exactly once?

- Let V , L , R stand for visiting the node, moving left, and moving right, resp.
 - Six possible combinations: LVR , LRV , VLR , VRL , RVL , RLV .

Binary Tree Traversals

Question

How to visit each node of a tree exactly once?

- Let V , L , R stand for visiting the node, moving left, and moving right, resp.
 - Six possible combinations: LVR , LRV , VLR , VRL , RVL , RLV .
- Adopting the convention that we traverse left before right, only three combinations of VLR remains:

Binary Tree Traversals

Question

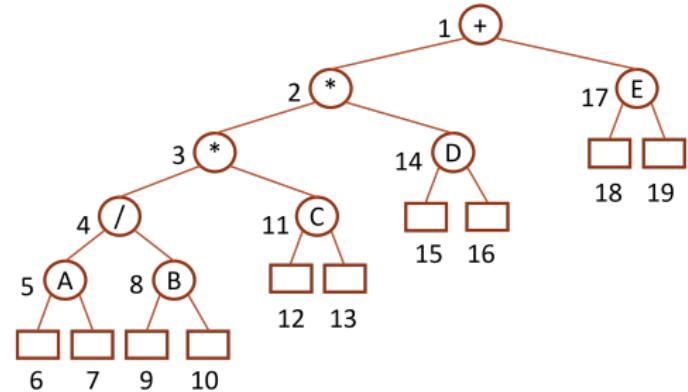
How to visit each node of a tree exactly once?

- Let V , L , R stand for visiting the node, moving left, and moving right, resp.
 - Six possible combinations: LVR , LRV , VLR , VRL , RVL , RLV .
- Adopting the convention that we traverse left before right, only three combinations of VLR remains:
 - **inorder** (中序走訪法)
 - **postorder** (後序走訪法)
 - **preorder** (先序走訪法)



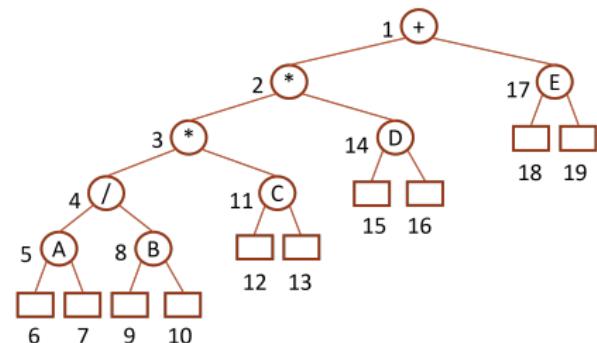
Tree Traversals

- Inorder traversal (LVR):
 $A/B*C*D+E$
- Preorder traversal (VLR):
 $+**/ABCDE$
- Postorder traversal (LRV):
 $AB/C*D*E+$



Code for Inorder Traversal

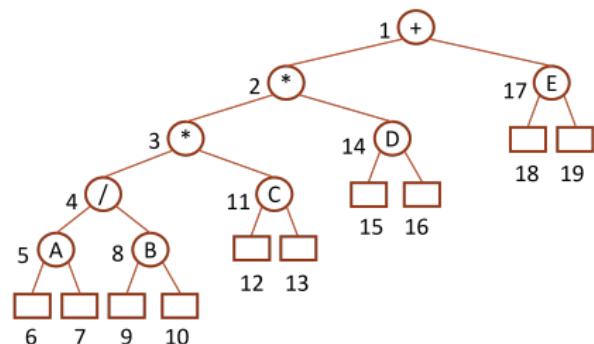
```
void inorder (treePointer ptr) {
    /* inorder tree traversal */
    if (ptr) {
        inorder (ptr->leftChild);
        printf ("%d", ptr->data);
        inorder (ptr->rightChild);
    }
}
```



A/B*C*D+E

Code for Preorder Traversal

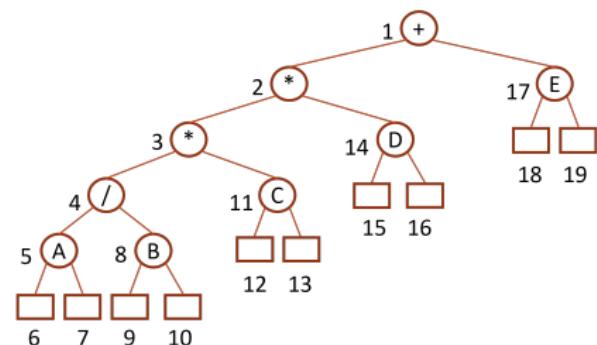
```
void Preorder (treePointer ptr) {
    /* inorder tree traversal */
    if (ptr) {
        printf ("%d", ptr->data);
        inorder (ptr->leftChild);
        inorder (ptr->rightChild);
    }
}
```



+**/ABCDE

Code for Postorder Traversal

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



AB/C*D*E+

Outline

1 Binary Tree Traversals (Using a Stack)

2 Level-Order Traversal

3 Additional Binary Tree Operations

Level-Order Traversal

- When written recursively, the inorder, preorder, and postorder traversals all require a **stack**.

Level-Order Traversal

- When written recursively, the inorder, preorder, and postorder traversals all require a **stack**.
- We now turn to a traversal that requires a **queue**. This traversal called **level-order traversal**.

Level-Order Traversal

- When written recursively, the inorder, preorder, and postorder traversals all require a **stack**.
- We now turn to a traversal that requires a **queue**. This traversal called **level-order traversal**.

Steps of a Level-Order Traversal

- visit the root first.
- then the root's left child followed by the right child.
- visit next level from leftmost node to right most node.

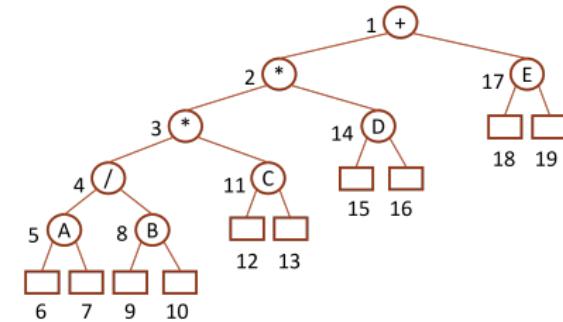


Code for the Level-Order Traversal

```

void levelOrder(treePointer ptr) {
    int front = rear = 0;
    treePointer queue[MAX_QUEUE_SIZE];
    if (!ptr) return; /* empty tree */
    add(ptr); // enqueue
    while (1) {
        ptr = delete(); // dequeue
        if (ptr) {
            printf("%d", ptr->data);
            if (ptr->leftChild)
                // leftChild exists
                add(ptr->leftChild);
                // enqueue
            if (ptr->rightChild)
                // rightChild exists
                add(ptr->rightChild);
                // enqueue
        } else break;
    }
}

```



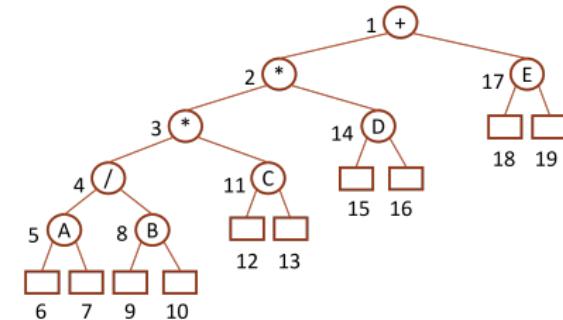
+

Code for the Level-Order Traversal

```

void levelOrder(treePointer ptr) {
    int front = rear = 0;
    treePointer queue[MAX_QUEUE_SIZE];
    if (!ptr) return; /* empty tree */
    add(ptr); // enqueue
    while (1) {
        ptr = delete(); // dequeue
        if (ptr) {
            printf("%d", ptr->data);
            if (ptr->leftChild)
                // leftChild exists
                add(ptr->leftChild);
                // enqueue
            if (ptr->rightChild)
                // rightChild exists
                add(ptr->rightChild);
                // enqueue
        } else break;
    }
}

```



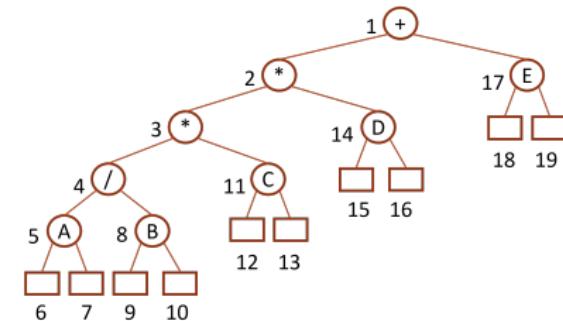
+*

Code for the Level-Order Traversal

```

void levelOrder(treePointer ptr) {
    int front = rear = 0;
    treePointer queue[MAX_QUEUE_SIZE];
    if (!ptr) return; /* empty tree */
    add(ptr); // enqueue
    while (1) {
        ptr = delete(); // dequeue
        if (ptr) {
            printf("%d", ptr->data);
            if (ptr->leftChild)
                // leftChild exists
                add(ptr->leftChild);
                // enqueue
            if (ptr->rightChild)
                // rightChild exists
                add(ptr->rightChild);
                // enqueue
        } else break;
    }
}

```



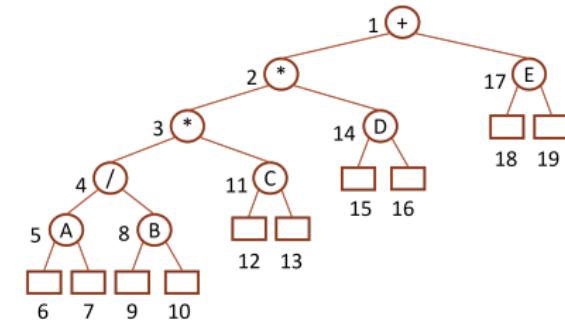
+*E

Code for the Level-Order Traversal

```

void levelOrder(treePointer ptr) {
    int front = rear = 0;
    treePointer queue[MAX_QUEUE_SIZE];
    if (!ptr) return; /* empty tree */
    add(ptr); // enqueue
    while (1) {
        ptr = delete(); // dequeue
        if (ptr) {
            printf("%d", ptr->data);
            if (ptr->leftChild)
                // leftChild exists
                add(ptr->leftChild);
                // enqueue
            if (ptr->rightChild)
                // rightChild exists
                add(ptr->rightChild);
                // enqueue
        } else break;
    }
}

```



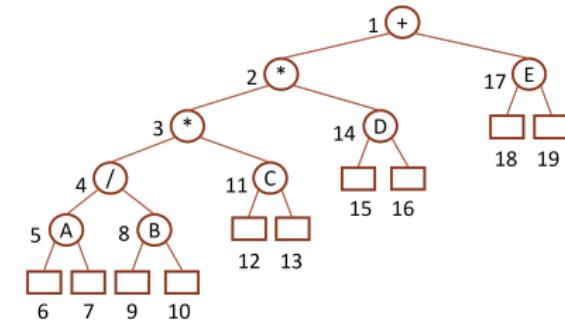
**E*

Code for the Level-Order Traversal

```

void levelOrder(treePointer ptr) {
    int front = rear = 0;
    treePointer queue[MAX_QUEUE_SIZE];
    if (!ptr) return; /* empty tree */
    add(ptr); // enqueue
    while (1) {
        ptr = delete(); // dequeue
        if (ptr) {
            printf("%d", ptr->data);
            if (ptr->leftChild)
                // leftChild exists
                add(ptr->leftChild);
                // enqueue
            if (ptr->rightChild)
                // rightChild exists
                add(ptr->rightChild);
                // enqueue
        } else break;
    }
}

```



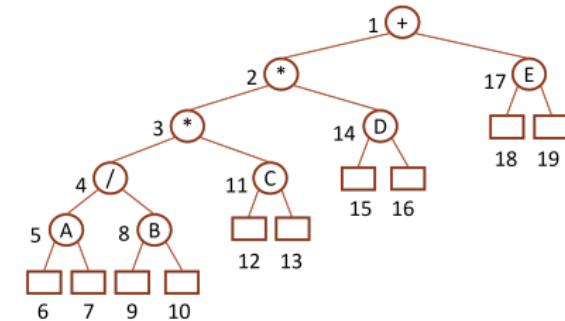
**E*D

Code for the Level-Order Traversal

```

void levelOrder(treePointer ptr) {
    int front = rear = 0;
    treePointer queue[MAX_QUEUE_SIZE];
    if (!ptr) return; /* empty tree */
    add(ptr); // enqueue
    while (1) {
        ptr = delete(); // dequeue
        if (ptr) {
            printf("%d", ptr->data);
            if (ptr->leftChild)
                // leftChild exists
                add(ptr->leftChild);
                // enqueue
            if (ptr->rightChild)
                // rightChild exists
                add(ptr->rightChild);
                // enqueue
        } else break;
    }
}

```



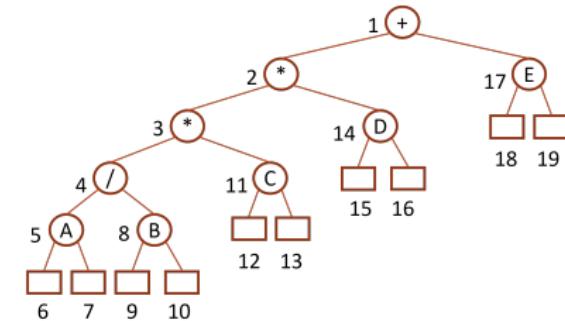
**E*D/

Code for the Level-Order Traversal

```

void levelOrder(treePointer ptr) {
    int front = rear = 0;
    treePointer queue[MAX_QUEUE_SIZE];
    if (!ptr) return; /* empty tree */
    add(ptr); // enqueue
    while (1) {
        ptr = delete(); // dequeue
        if (ptr) {
            printf("%d", ptr->data);
            if (ptr->leftChild)
                // leftChild exists
                add(ptr->leftChild);
                // enqueue
            if (ptr->rightChild)
                // rightChild exists
                add(ptr->rightChild);
                // enqueue
        } else break;
    }
}

```



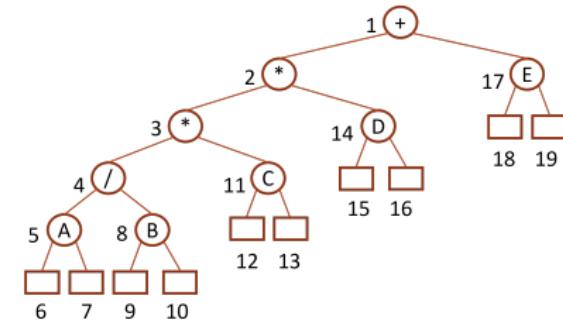
**E*D/C

Code for the Level-Order Traversal

```

void levelOrder(treePointer ptr) {
    int front = rear = 0;
    treePointer queue[MAX_QUEUE_SIZE];
    if (!ptr) return; /* empty tree */
    add(ptr); // enqueue
    while (1) {
        ptr = delete(); // dequeue
        if (ptr) {
            printf("%d", ptr->data);
            if (ptr->leftChild)
                // leftChild exists
                add(ptr->leftChild);
                // enqueue
            if (ptr->rightChild)
                // rightChild exists
                add(ptr->rightChild);
                // enqueue
        } else break;
    }
}

```



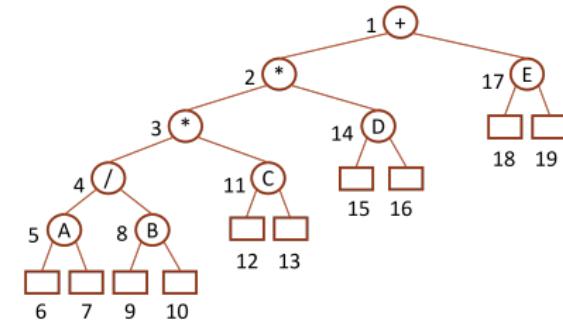
**E*D/CA

Code for the Level-Order Traversal

```

void levelOrder(treePointer ptr) {
    int front = rear = 0;
    treePointer queue[MAX_QUEUE_SIZE];
    if (!ptr) return; /* empty tree */
    add(ptr); // enqueue
    while (1) {
        ptr = delete(); // dequeue
        if (ptr) {
            printf("%d", ptr->data);
            if (ptr->leftChild)
                // leftChild exists
                add(ptr->leftChild);
                // enqueue
            if (ptr->rightChild)
                // rightChild exists
                add(ptr->rightChild);
                // enqueue
        } else break;
    }
}

```



**E*D/CAB

Outline

1 Binary Tree Traversals (Using a Stack)

2 Level-Order Traversal

3 Additional Binary Tree Operations

Copying a Binary Tree

```
treePointer copy(treePointer original) {
    /* return a tree_pointer to an exact copy of the original tree */
    treePointer temp;
    if (original) {
        MALLOC(temp, sizeof(*temp));
        temp->leftChild = copy(original->leftChild);
        temp->rightChild = copy(original->rightChild);
        temp->data = original->data;
        return temp;
    }
    return NULL;
}
```

Testing for Equality of Binary Trees

```
int equal(treePointer first, treePointer second) {
    /* function returns FALSE if the binary trees first and second are not equal
    Otherwise it returns TRUE */
    return (
        (!first && !second) || (first && second &&
        (first->data == second->data) &&
        equal(first->leftChild, second->leftChild) &&
        equal(first->rightChild, second->rightChild))
    )
}
```



Discussions