Binary Trees (cont.)

Instructor: Dr. Sunho Lim (Ph.D., Assistant Professor)

Lecture 09

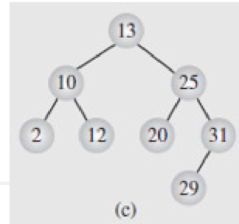
sunho.lim@ttu.edu

Adapted partially from Data Structures and Algorithms in C++, Adam Drozdek, 4th Edition, Cengage Learning; and Algorithms and Data Structures, Douglas Wilhelm Harder, Mmath

Tree Traversal

- The process of **visiting** each node in a tree data structure exactly one time
  - numerous possible traversals
  - e.g., in a tree of *n* nodes, there are *n*! traversals
- Two useful traversals
  - ***depth-first traversals***
  - ***breadth-first traversals***

Tree Traversal (cont.)



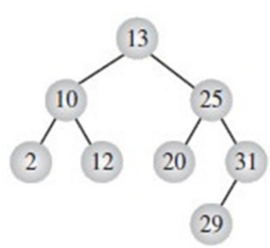
- **Breadth-First Traversal**
  - proceed **level-by-level** from top-down or bottom-up
  - visit each level's nodes left-to-right or right-to-left
  - e.g., 13, 10, 25, 2, 12, 20, 31, 29

- Implement using a **queue**, consider a **top-down**, **left-to-right** breadth-first traversal
  - start by placing the **root node** in the queue
  - then remove the node at the front of the queue
  - **after visiting it**, place its **children** (if any) in the queue
  - repeat until the queue is empty

Tree Traversal (cont.)

- Breadth-First Traversal (continued)
  - the queue-based breadth-first traversal

Tree



Queue				Output
13				
10	25			13
25	2	12		13, 10
2	12	20	31	13, 10, 25
12	20	31		13, 10, 25, 2
20	31			13, 10, 25, 2, 12
31				13, 10, 25, 2, 12, 20
29				13, 10, 25, 2, 12, 20, 31
				13, 10, 25, 2, 12, 20, 31, 29

Tree Traversal (cont.)

- Breadth-First Traversal (continued)

```
template<class T>
void BST<T>::breadthFirst() {
    Queue<BSTNode<T>*> queue;
    BSTNode<T> *p = root;
    if (p != 0) {
        queue.enqueue(p);
        while (!queue.empty()) {
            p = queue.dequeue();
            visit(p);
            if (p->left != 0)
                 queue.enqueue(p->left);
            if (p->right != 0)
                 queue.enqueue(p->right);
        }
    }
}
```


Tree Traversal (cont.)

- **Depth-First Traversal**
  - proceed by following **left- (or right-) hand branches** as far as possible
  - **backtrack** to the most recent fork and take the right- (or left-) hand branch to the next node
  - follow branches to the left (or right) again as far as possible
  - continue until all nodes have been visited
- Three activities:
  - traversing to the left (L)
  - traversing to the right (R)
  - visiting a node (V)

Tree Traversal

- Depth-First Traversal (continued)
  - follow the convention of traversing from **left to right**:
    - VLR – known as ***preorder traversal***
    - LVR – known as ***inorder traversal***
    - LRV – known as ***postorder traversal***

```
template<class T>
void BST<T>::preorder(BSTNode<T> *p) {
    if (p != 0) {
        visit(p);
        preorder(p->left);
        preorder(p->right);
    }
}
```

```
template<class T>
void BST<T>::inorder(BSTNode<T> *p) {
    if (p != 0) {
        inorder(p->left);
        visit(p);
        inorder(p->right);
    }
}
```

```
template<class T>
void BST<T>::postorder(BSTNode<T>* p) {
    if (p != 0) {
        postorder(p->left);
        postorder(p->right);
        visit(p);
    }
}
```

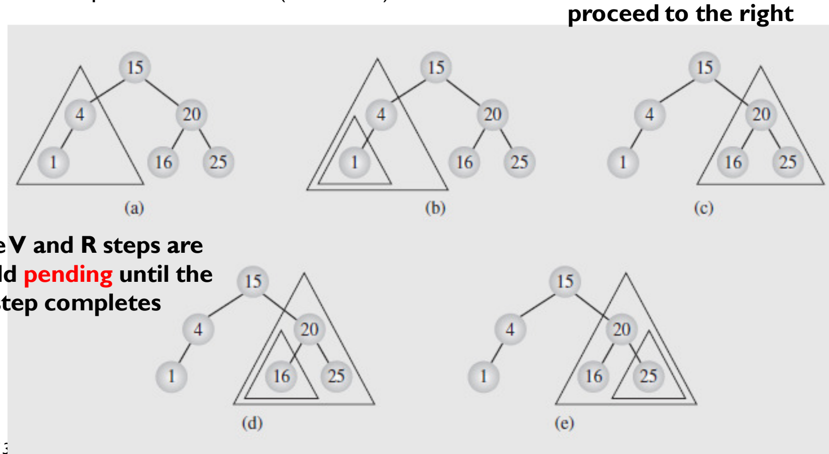

Tree Traversal (cont.)

- Depth-First Traversal (continued)
  - the recursion supported by the **run-time stack**
    - a heavy burden on the system
  - e.g., the **inorder** routine
    - traverse the left subtree of the node, then visit the node, then traverse the right subtree

Tree Traversal (cont.)

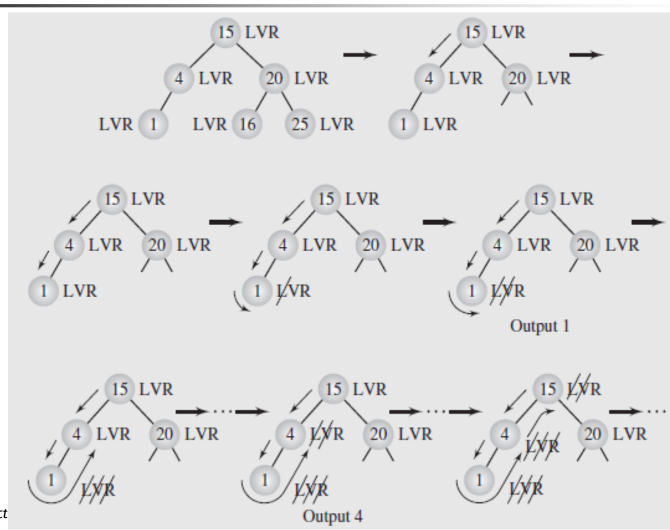
- Depth-First Traversal (continued)

the stack remembers the backtrack point, then visit the branch point node, and proceed to the right



the V and R steps are held pending until the L step completes

Tree Traversal (cont.)



Tree Traversal (cont.)

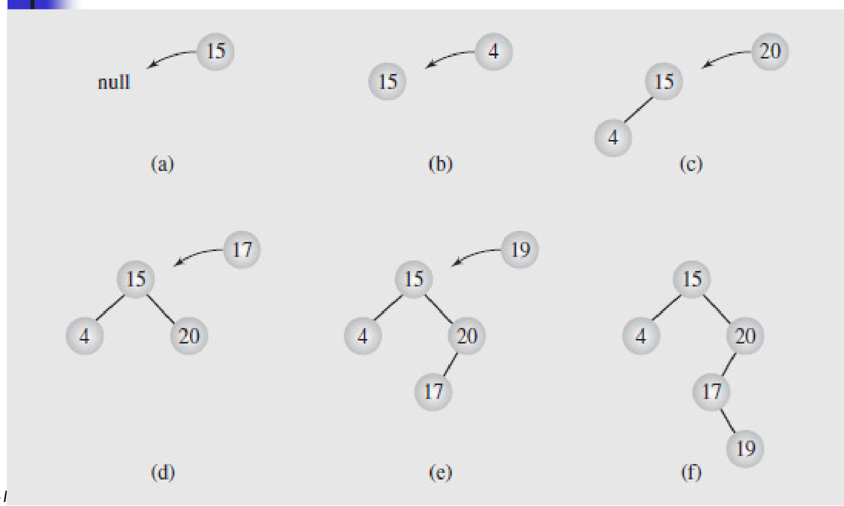
- Depth-First Traversal (continued)
  - consider nonrecursive implementations of the traversal algorithms
  - e.g., a nonrecursive version of the **preorder** algorithm

```
template<class T>
void BST<T>::iterativePreorder() {
    Stack<BSTNode<T>*> travStack;
    BSTNode<T> *p = root;
    if (p != 0) {
        travStack.push(p);
        while (!travStack.empty()) {
            p = travStack.pop();
            visit(p);
            if (p->right != 0)
                 travStack.push(p->right);
            if (p->left != 0)    // left child pushed after right
                 travStack.push(p->left); // to be on the top of
        }                       // the stack;
    }
}
```


Insertion

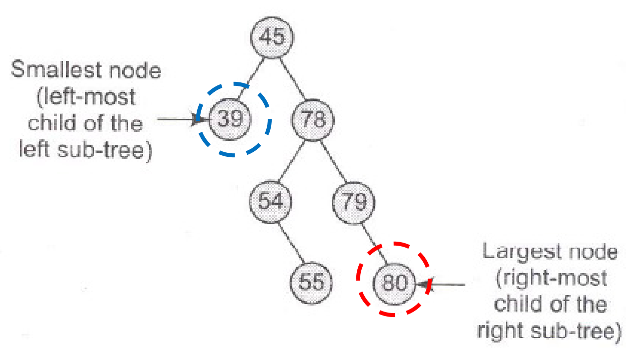
- Searching a binary tree
  - does not modify the tree
- Operations like insertions, deletions, modifying values, merging trees, and balancing trees
  - **alter the tree structure**
- Insert a new node in a binary tree??
  - perform in the same way as searching
  - compare the value of the node to be inserted to the current node
  - If the value to be inserted is smaller,
    - follow the left subtree;
  - if it is larger,
    - follow the right subtree
  - If the branch we are to follow is empty,
    - stop the search and insert the new node as that child

Insertion (cont.)



Search

- Finding the **smallest** or **largest** node

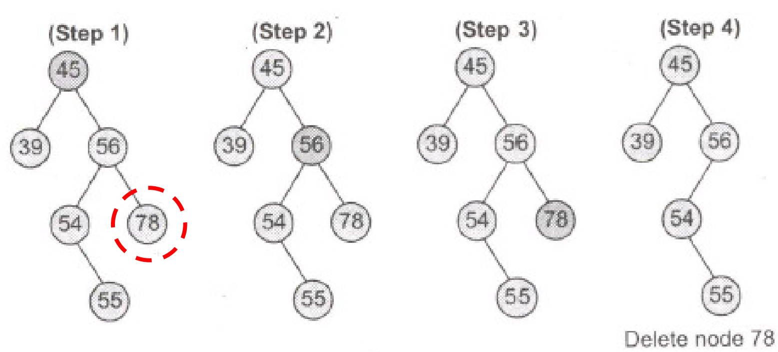


Deletion

- Can be a complex operation depending on the placement of the node to be deleted in the tree
  - more children a node has, more complex the deletion process
- **Three cases of deletion** that need to be handled:
  - deleting a node that has no children
  - deleting a node with one child
  - deleting a node with two children

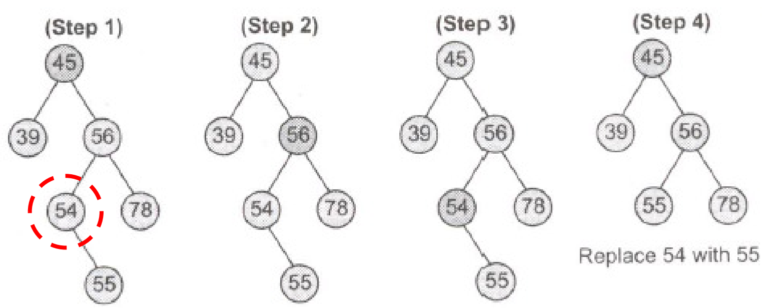
Deletion

- deleting a node that has no children (e.g., delete 78)



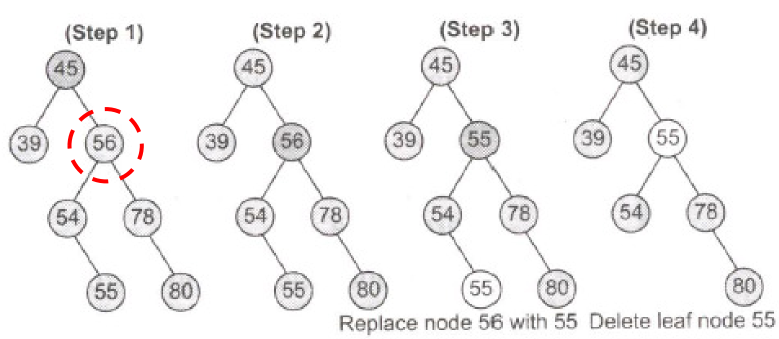
Deletion

- deleting a node with one child (e.g., delete 54)



Deletion

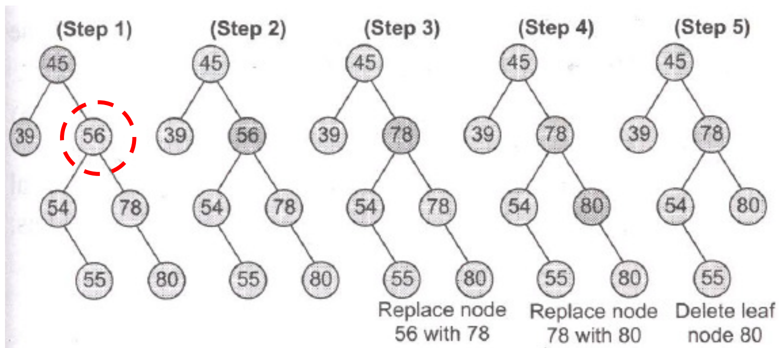
- deleting a node with two children (e.g., delete 56)



Find the **largest value** in the **left subtree**

Deletion

- deleting a node with two children (e.g., delete 56) (cont.)



Find the **smallest value** in the **right subtree**