Binary Trees

Lecture 12

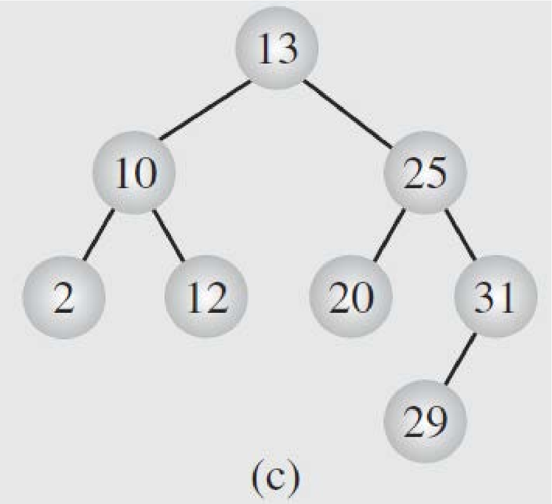
Instructor: Dr. Cong Pu, Ph.D.

cong.pu@okstate.edu

Adapted partially from Data Structures and Algorithms in Java, M.T. Goodrich, R.Tamassia and M. H. Goldwasser, Sixth Edition, Wiley; Data Structures and Algorithms in C++, Adam Drozdek, 4th Edition, Cengage Learning

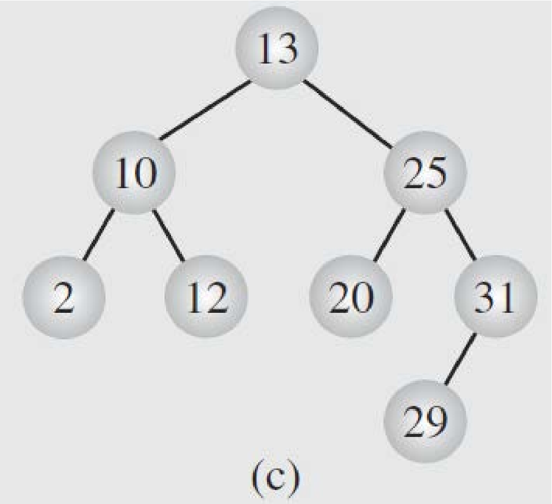
Tree Traversal

- ***Tree traversal***: the process of **visiting** each node in a tree data structure *exactly one time*
  - visiting nodes, but no visiting order specified
  - numerous possible tree traversals
  - e.g., in a tree of *n* nodes, there are *n*! traversals
    - most of them are chaotic and no regularity
    - two possible traversals
      - 2, 10, 12, 20, 13, 25, 29, 31
        - lists even numbers and then odd numbers in ascending order
      - 29, 31, 20, 12, 2, 25, 10, 13
        - lists all nodes from level to level right to left, starting from the lowest level up to the root



Tree Traversal

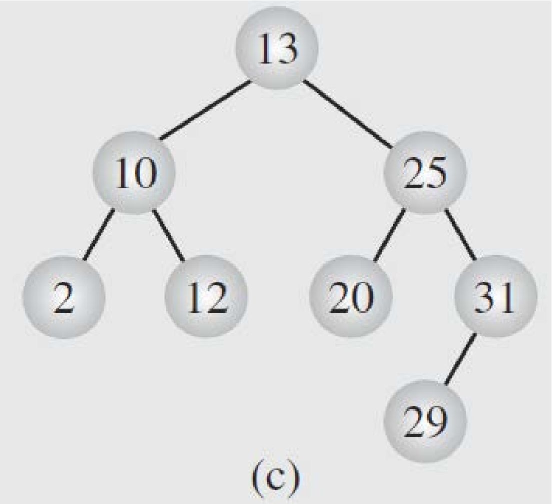
- ***Tree traversal***: the process of **visiting** each node in a tree data structure *exactly one time*
  - visiting nodes, but no visiting order specified
  - numerous possible tree traversals
  - e.g., in a tree of *n* nodes, there are *n*! traversals
    - most of them are chaotic and no regularity
    - another possible traversal
      - 13, 31, 12, 2, 10, 29, 20, 25
        - no regularity;
        - random jumping from node to node



(c)

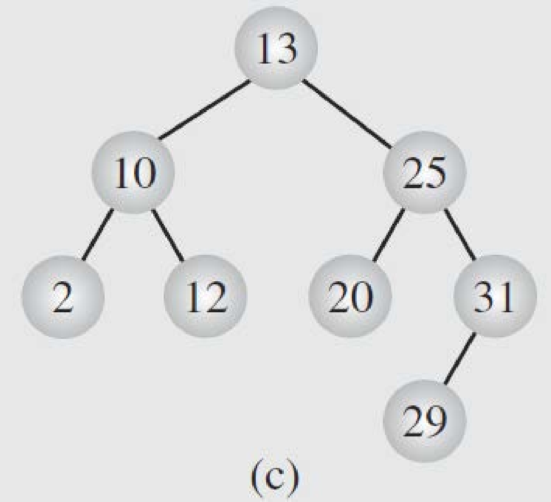
Tree Traversal

- ***Tree traversal***: the process of **visiting** each node in a tree data structure *exactly one time*
  - visiting nodes, but no visiting order specified
  - numerous possible tree traversals
  - e.g., in a tree of *n* nodes, there are *n*! traversals
    - most of them are chaotic and no regularity
- Two useful traversals
  - ***depth-first traversals***
  - ***breadth-first traversals***



(c)

Tree Traversal (cont.)

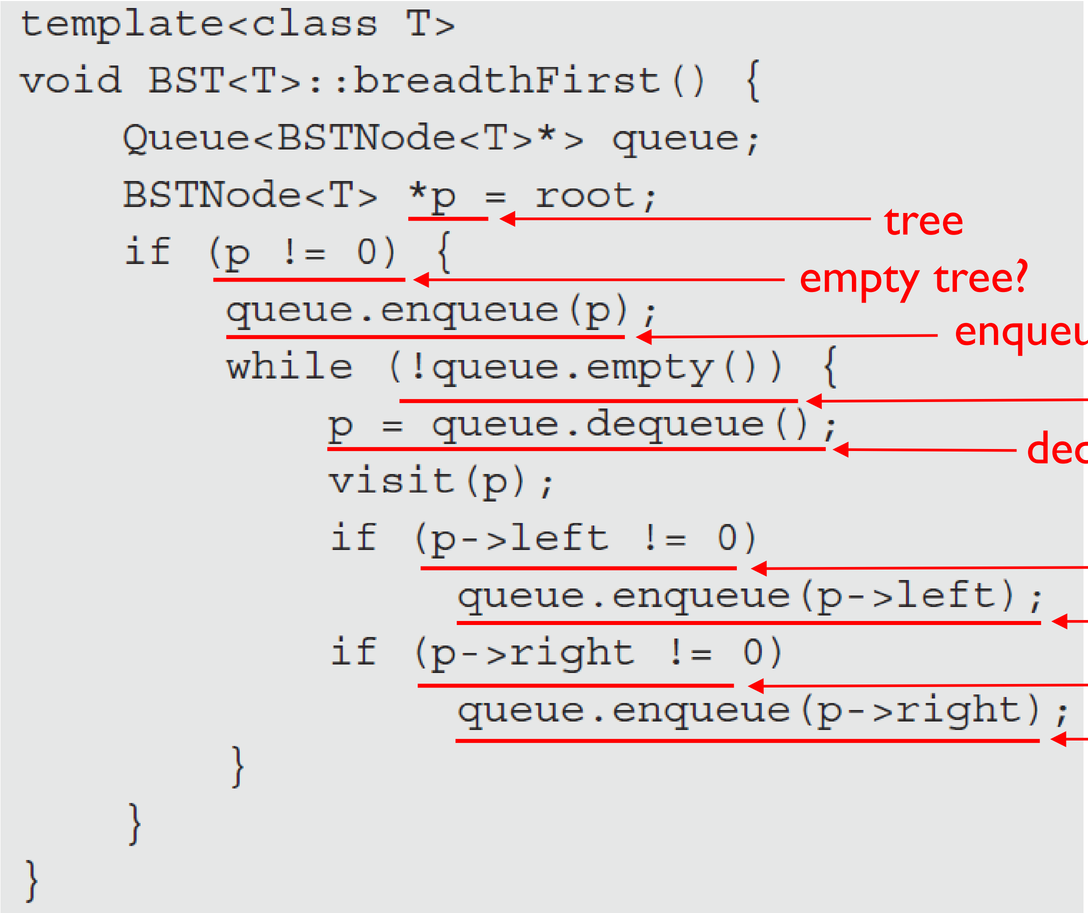


- **Breadth-First Traversal**
  - visit each node in the tree
  - start from lowest (or highest) level and move down (or up) level by level
    - on each level, visit node from left to right (or from right to left)
  - one of four possible traversals
    - e.g., 13, 10, 25, 2, 12, 20, 31, 29 (***top-down***, ***left-to-right***)
- 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) at the ***end*** of the queue
  - repeat until the queue is ***empty***

all nodes on level *n* must be visited before visiting nodes on level *n*+1

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

empty tree?

enqueue root in queue

still having node in queue

dequeue front node in queue

dequeued node has left child

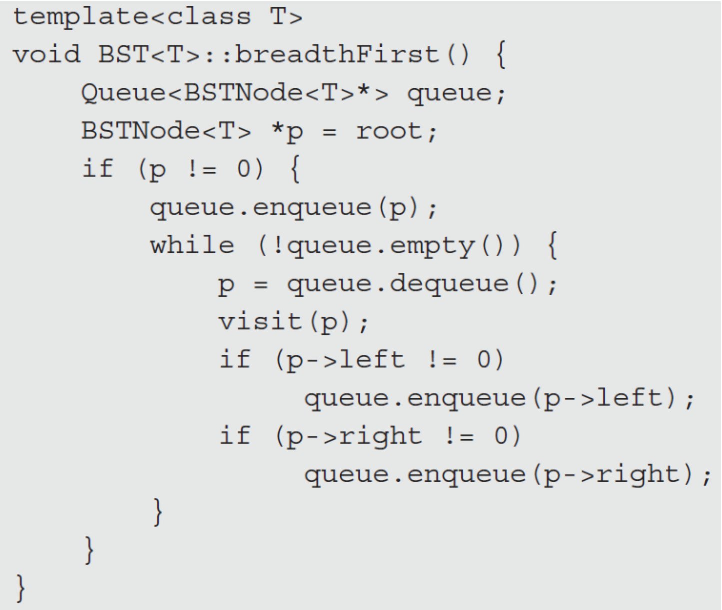
enqueue left child

dequeued node has right child

enqueue right child

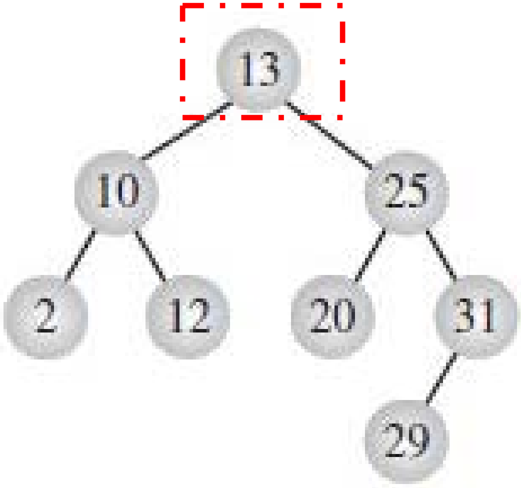
Tree Traversal (cont.)

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



```
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);
        }
    }
}
```

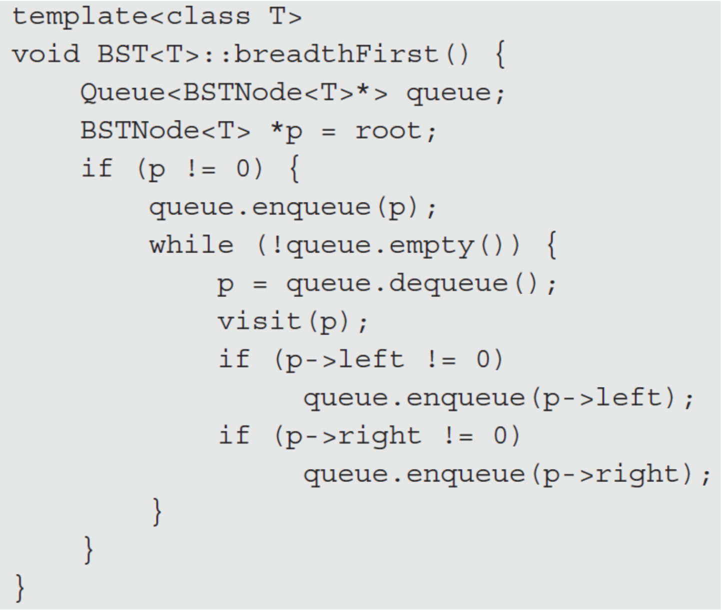
Tree



Queue

13			

Output

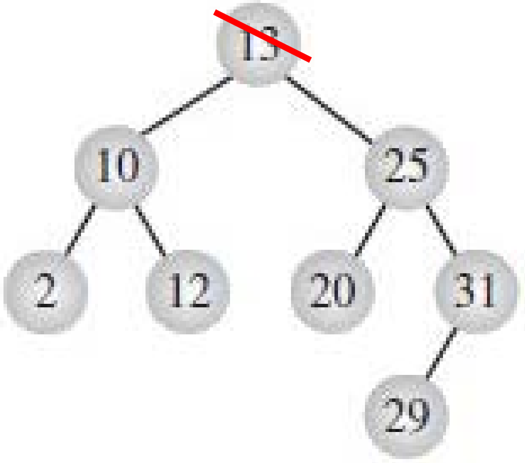


```
template<class T>
void BST<T>::breadthFirst() {
    Queue<BSTNode<T>*> queue;
    BSTNode<T> *p = root;
    if (p != 0) {
        queue.enqueue(p);
        while (!queue.empty()) {
            p = queue.dequeue();
            visit(p);
            if (p->left != 0)
                 queue.enqueue(p->left);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

Tree Traversal (cont.)

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

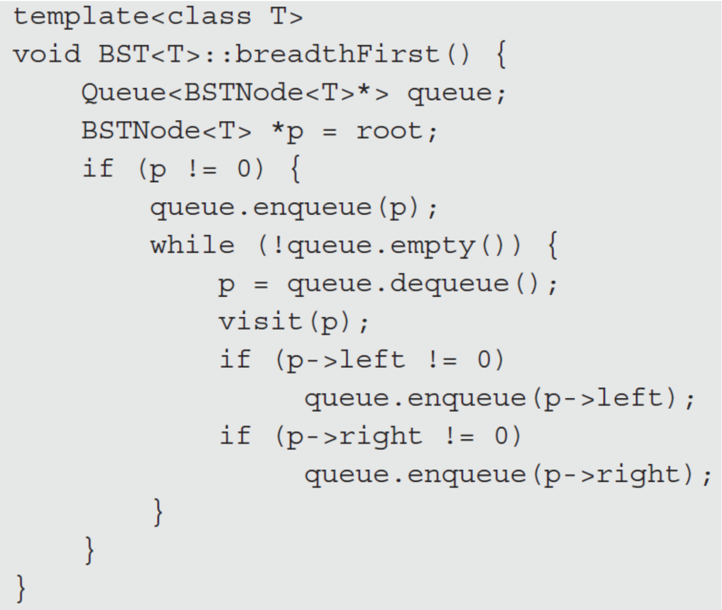
Tree



Queue				Output
13				
10	25			13

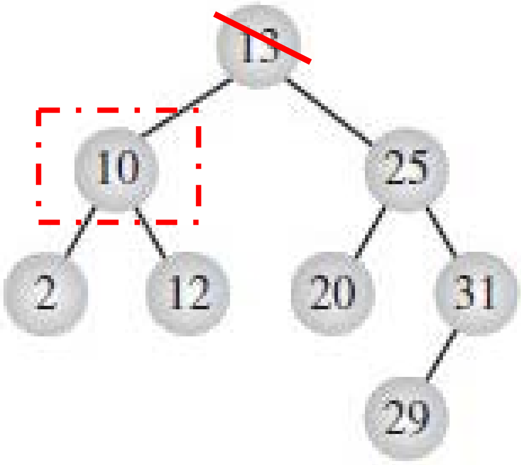
Tree Traversal (cont.)

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



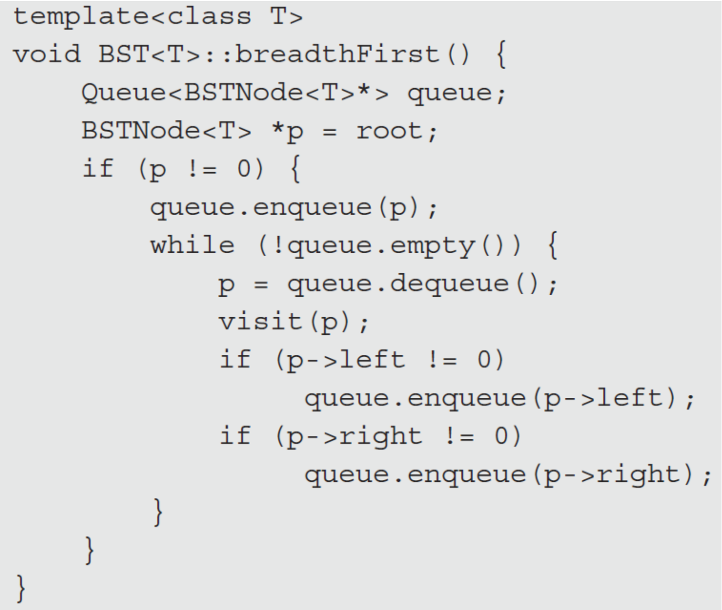
```
template<class T>
void BST<T>::breadthFirst() {
    Queue<BSTNode<T>*> queue;
    BSTNode<T> *p = root;
    if (p != 0) {
        queue.enqueue(p);
        while (!queue.empty()) {
            p = queue.dequeue();
            visit(p);
            if (p->left != 0)
                 queue.enqueue(p->left);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

Tree



Queue				Output
13				
10	25			13
25	2	12		13, 10

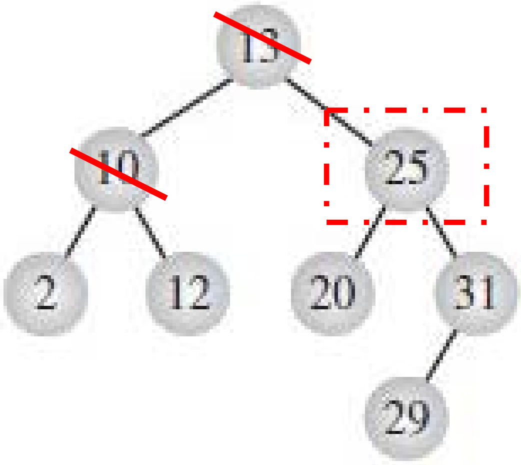
Tree Traversal (cont.)



```
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);
        }
    }
}
```

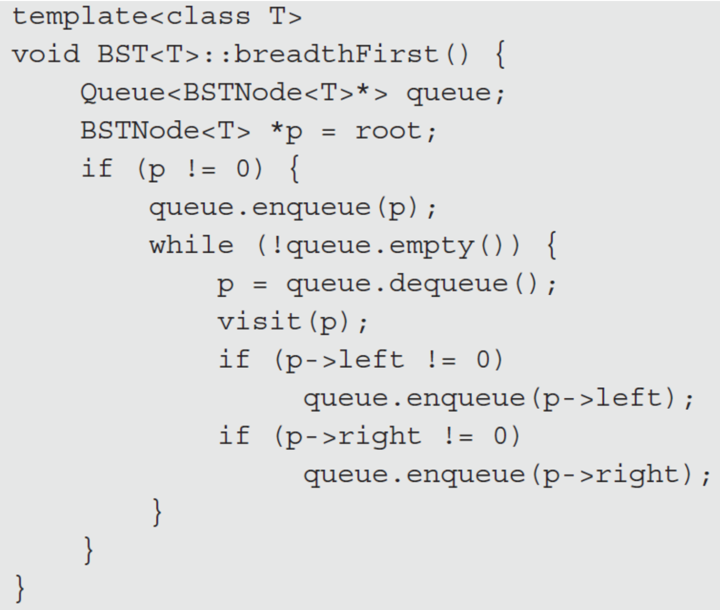
- Breadth-First Traversal (continued)
  - the queue-based breadth-first traversal

Tree



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

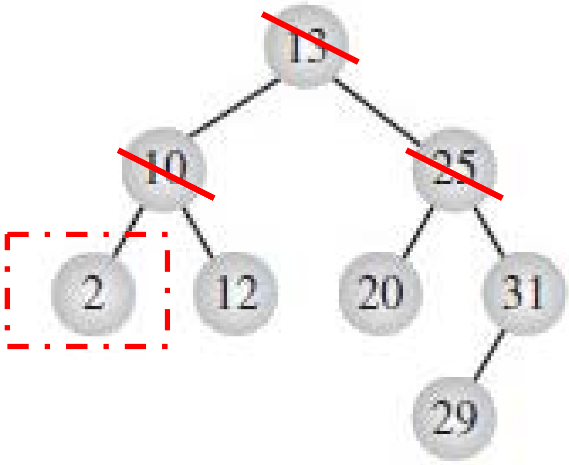
Tree Traversal (cont.)



```
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);
        }
    }
}
```

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

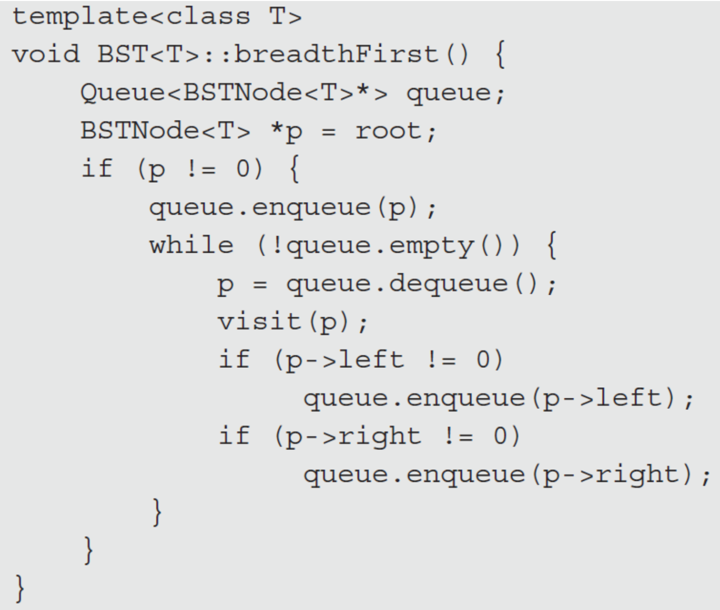
Tree



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

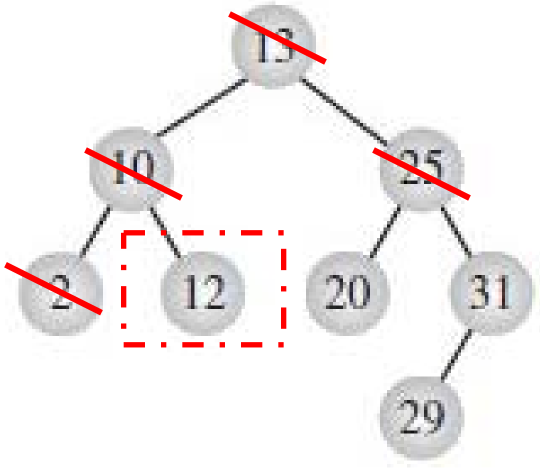
Tree Traversal (cont.)

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



```
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);
        }
    }
}
```

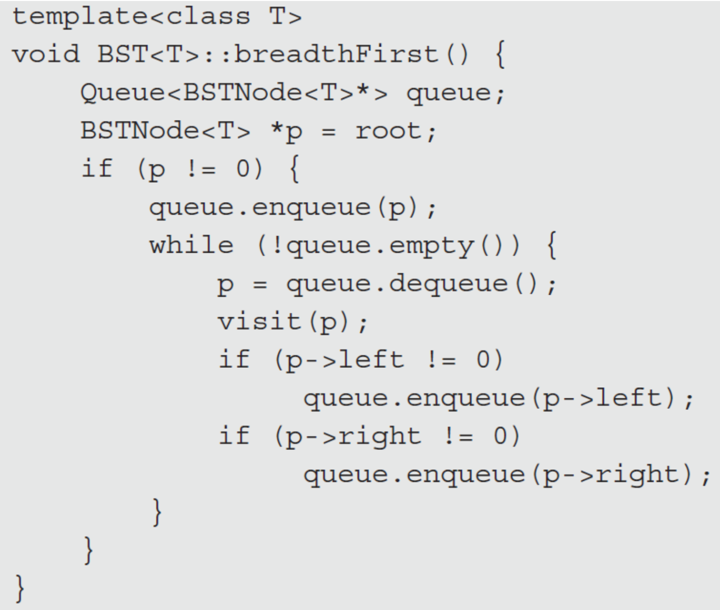
Tree



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

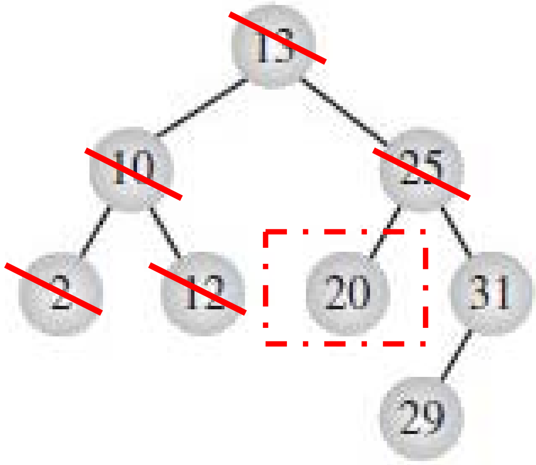
Tree Traversal (cont.)

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



```
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);
        }
    }
}
```

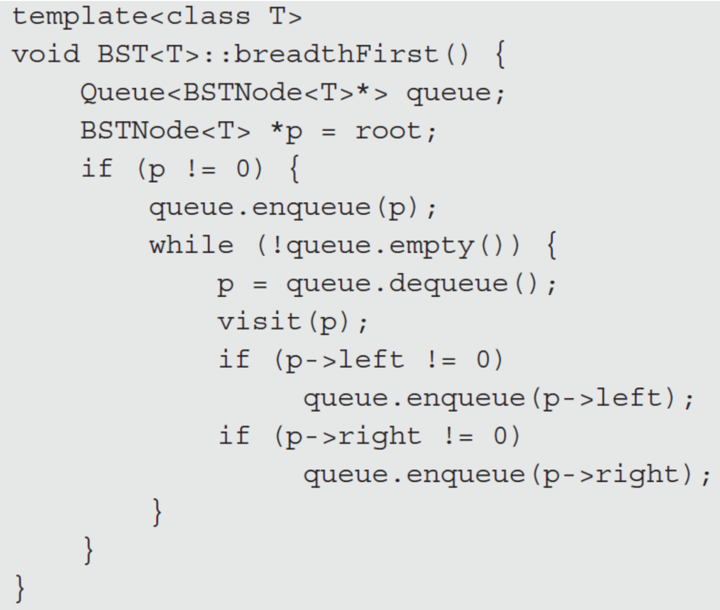
Tree



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

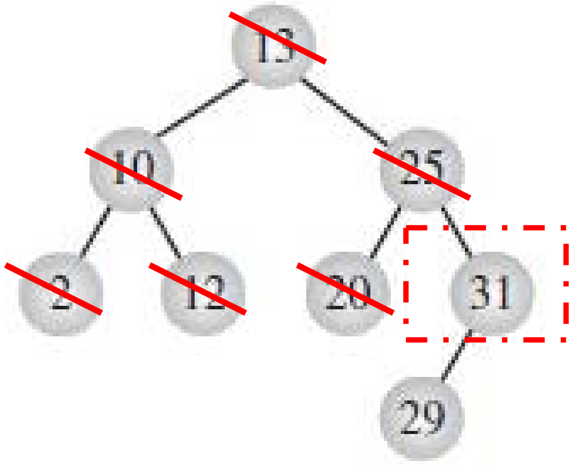
Tree Traversal (cont.)

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



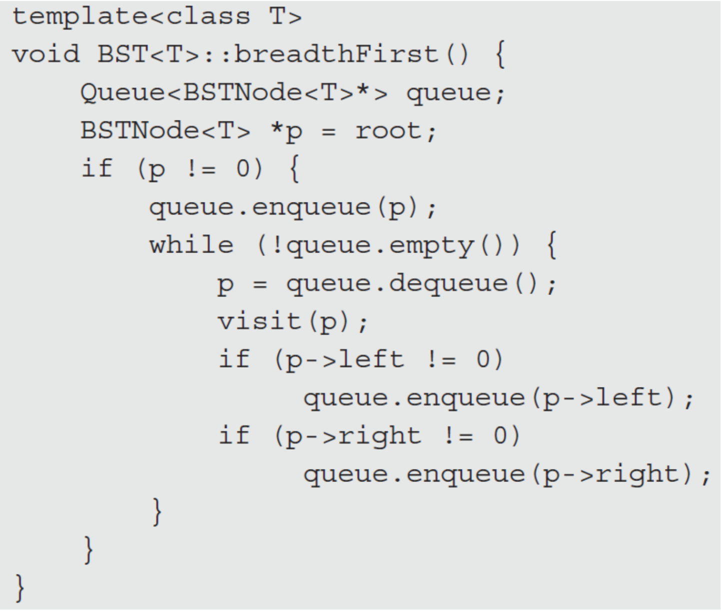
```
template<class T>
void BST<T>::breadthFirst() {
    Queue<BSTNode<T>*> queue;
    BSTNode<T> *p = root;
    if (p != 0) {
        queue.enqueue(p);
        while (!queue.empty()) {
            p = queue.dequeue();
            visit(p);
            if (p->left != 0)
                 queue.enqueue(p->left);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

Tree



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

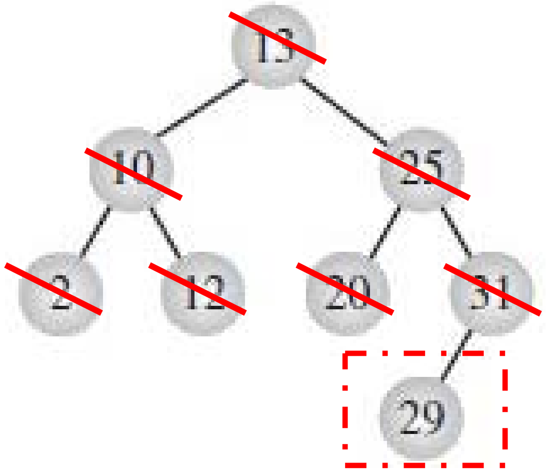
Tree Traversal (cont.)



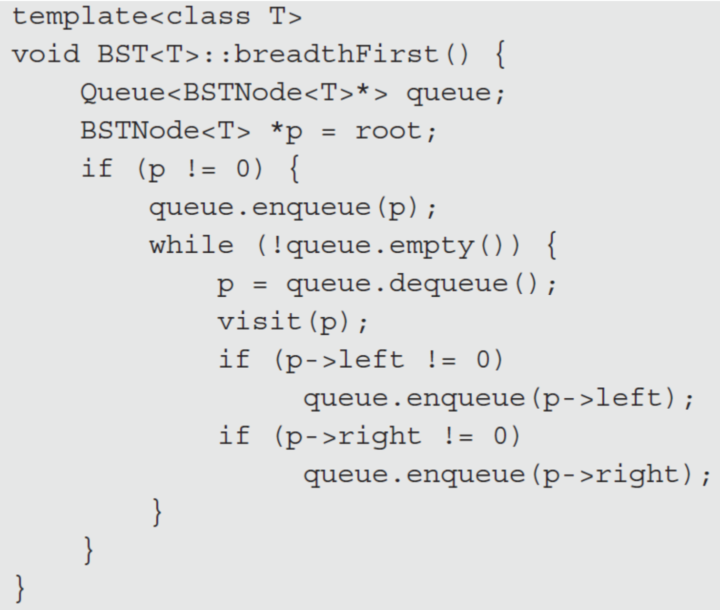
```
template<class T>
void BST<T>::breadthFirst() {
    Queue<BSTNode<T>*> queue;
    BSTNode<T> *p = root;
    if (p != 0) {
        queue.enqueue(p);
        while (!queue.empty()) {
            p = queue.dequeue();
            visit(p);
            if (p->left != 0)
                 queue.enqueue(p->left);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

- 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

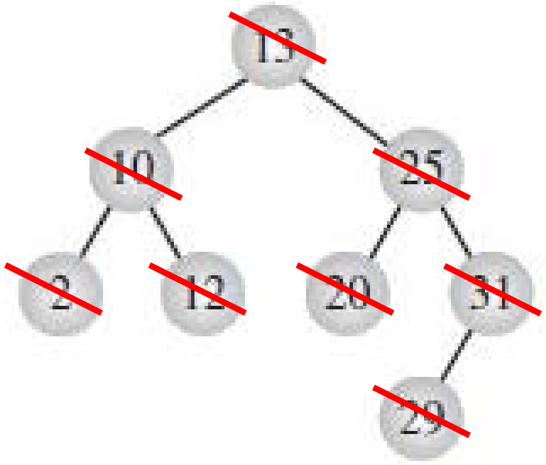


```
template<class T>
void BST<T>::breadthFirst() {
    Queue<BSTNode<T>*> queue;
    BSTNode<T> *p = root;
    if (p != 0) {
        queue.enqueue(p);
        while (!queue.empty()) {
            p = queue.dequeue();
            visit(p);
            if (p->left != 0)
                 queue.enqueue(p->left);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

Tree Traversal (cont.)

- 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

(empty queue!)

Tree Traversal (cont.)

- Depth-First Traversal
  - proceed by following **left- (or right-) hand branches** ***as far as possible***
  - **backtrack** to the ***most recent crossroad*** 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

  (when are nodes visited?? before proceeding down or after backing up??)
- Three activities:
  - traversing to the left subtree (L)
  - traversing to the right subtree (R)
  - visiting a node (V)

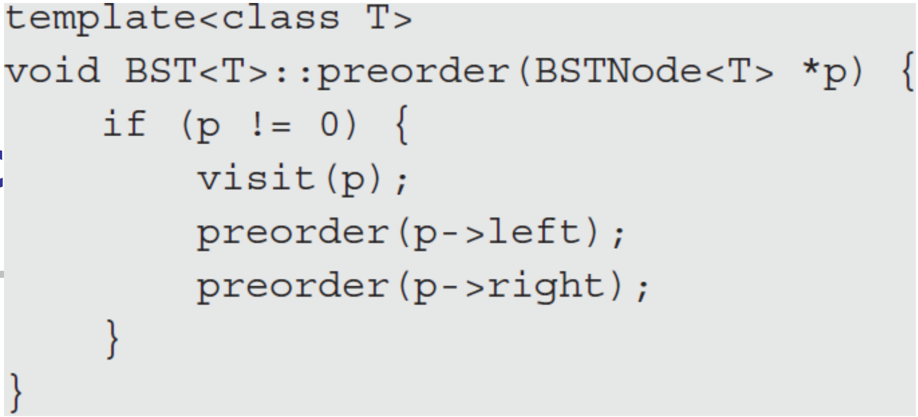
Tree Traversal (cont.)

- Three activities:
  - traversing to the left subtree (L)
  - traversing to the right subtree (R)
  - visiting a node (V)
- An ***orderly traversal***: the tasks are performed in ***the same order*** for each node
- Six possible ordered depth-first traversals

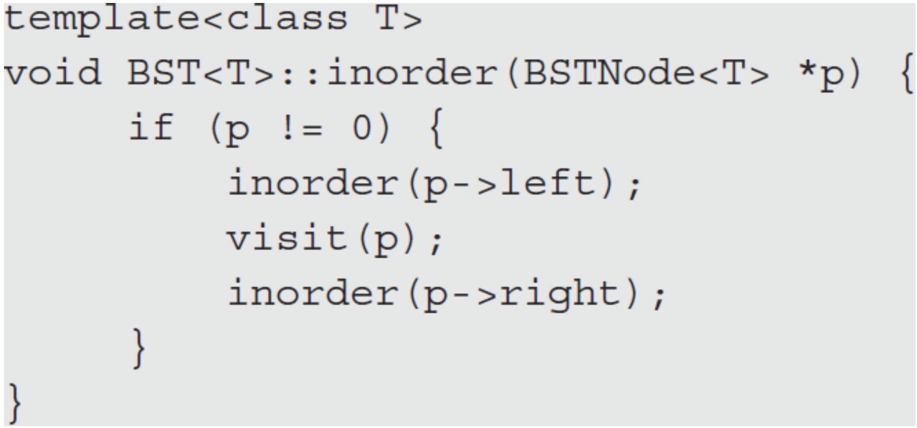
VLR VRL LVR RVL LRV RLV

Tree Traversal (c

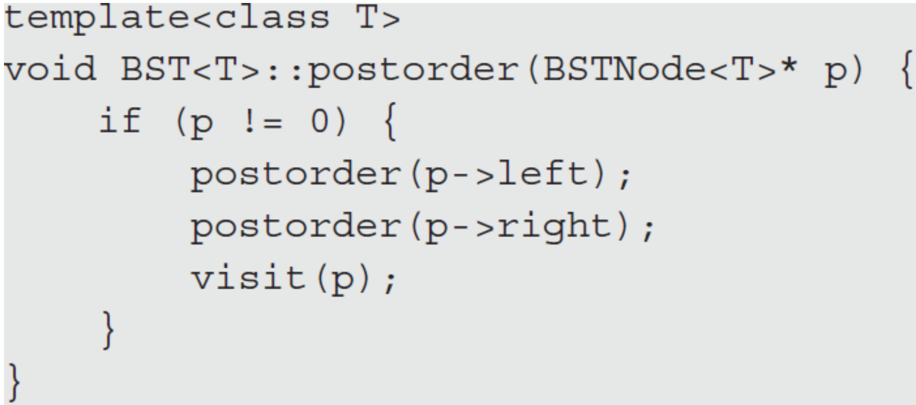
- Depth-First Traversal (continued)
  - follow the convention of traversing from **left to right**:
    - VLR – known as ***preorder traversal***
    - LVR – known as ***inorder traversal***
    - LRV – known as ***postorder traversal***



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



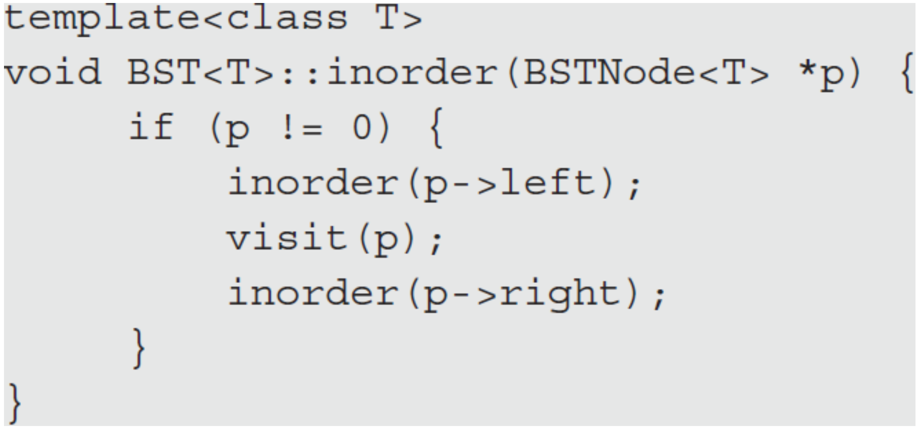
```
template<class T>
void BST<T>::inorder(BSTNode<T> *p) {
     if (p != 0) {
         inorder(p->left);
         visit(p);
         inorder(p->right);
     }
}
```



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

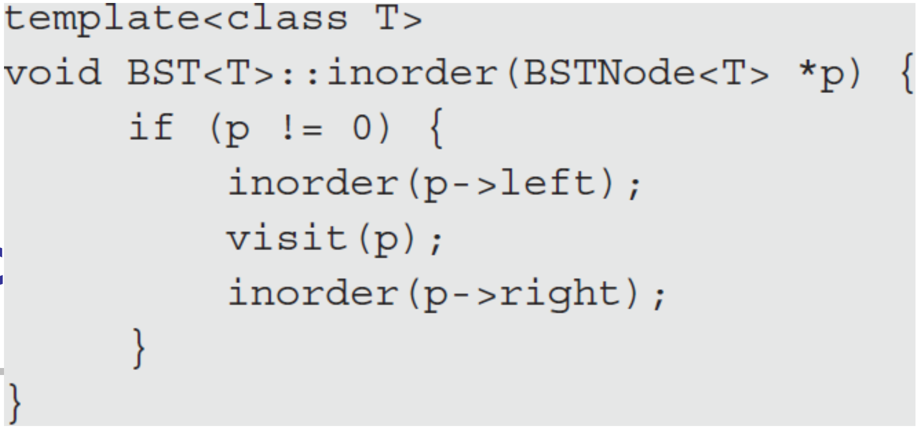
Tree Traversal (cont.)

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



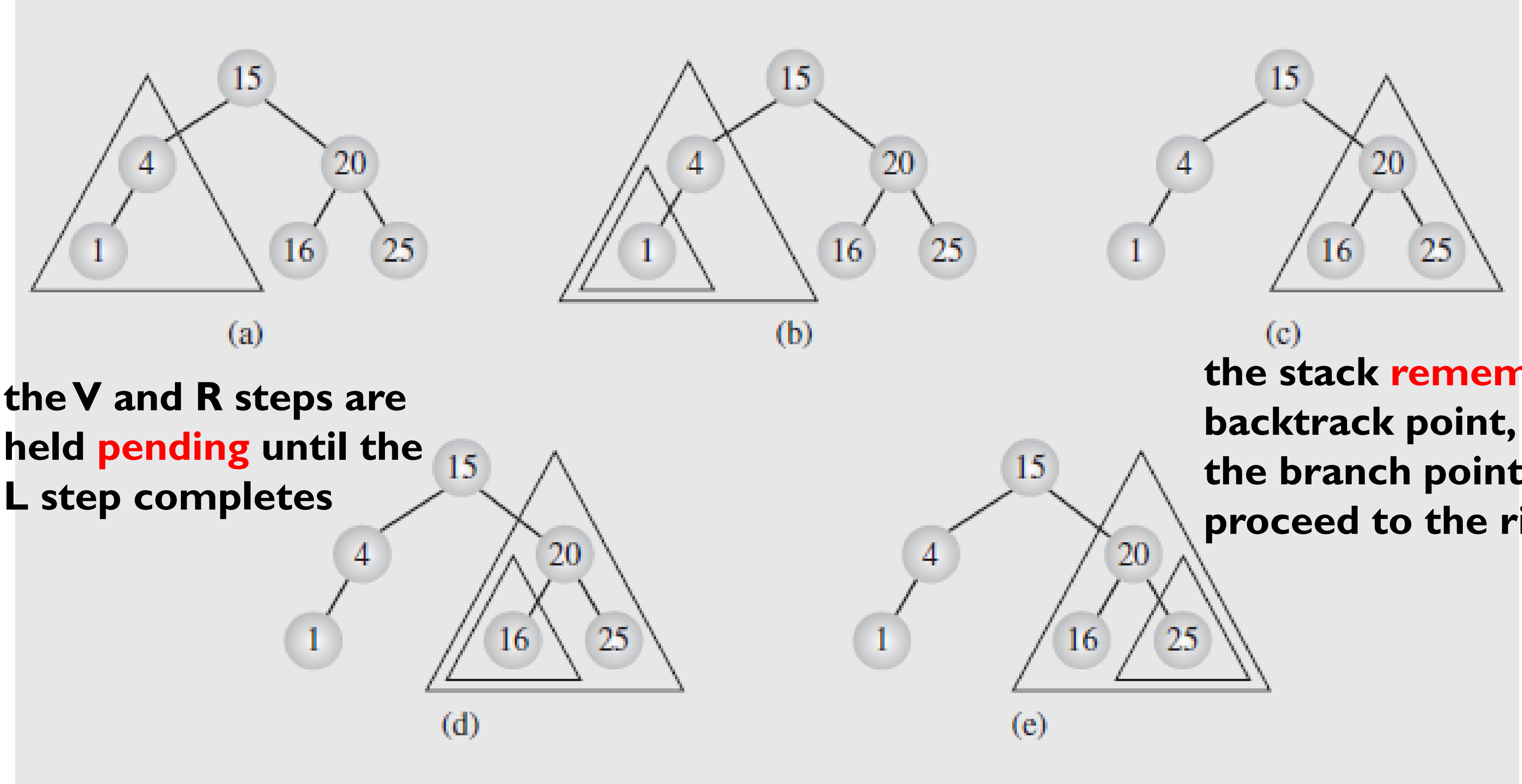
```
template<class T>
void BST<T>::inorder(BSTNode<T> *p) {
    if (p != 0) {
        inorder(p->left);
        visit(p);
        inorder(p->right);
    }
}
```

Tree Traversal (c



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

- Depth-First Traversal (continued) – L V R



(a)

(b)

(c)

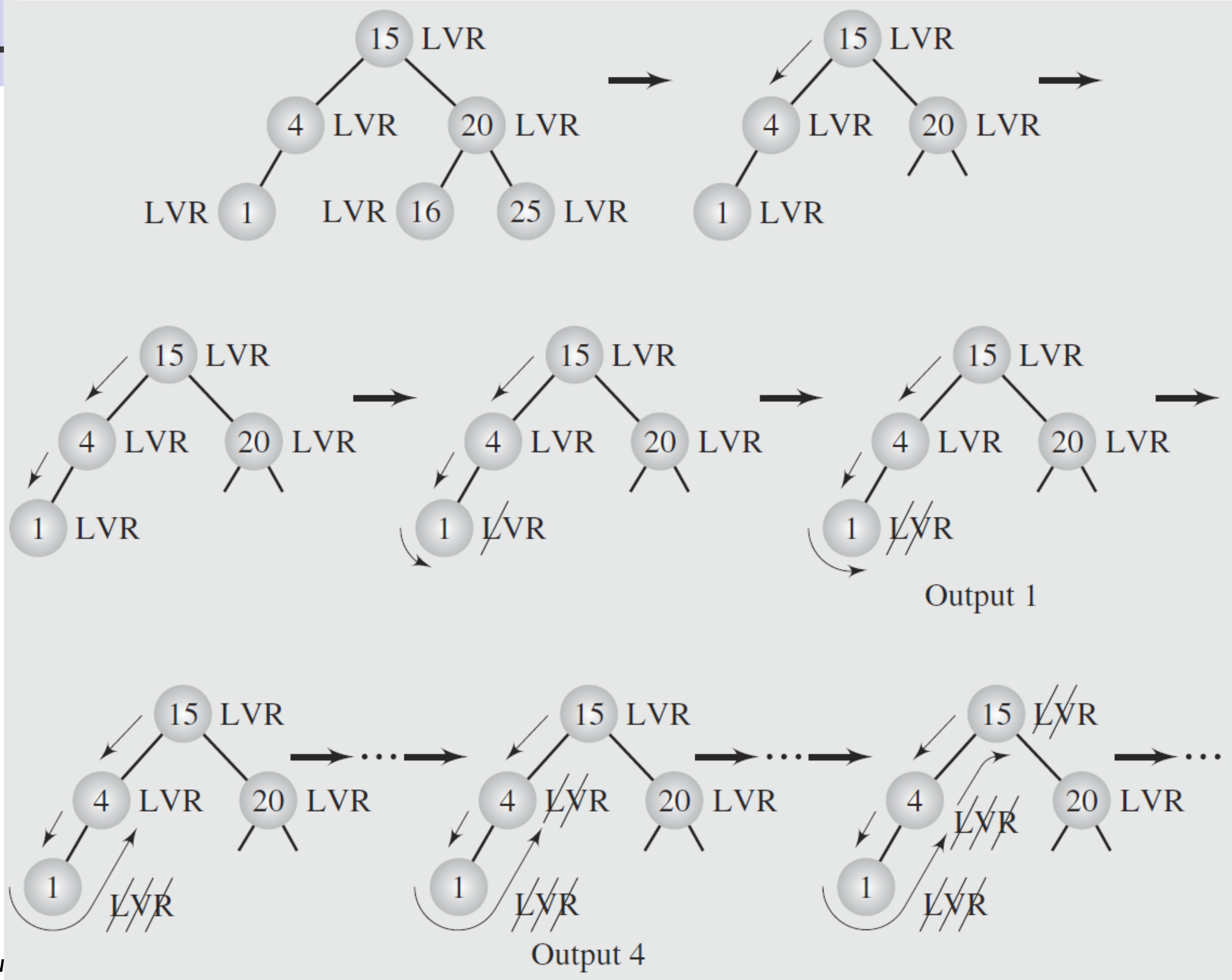
(d)

(e)

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

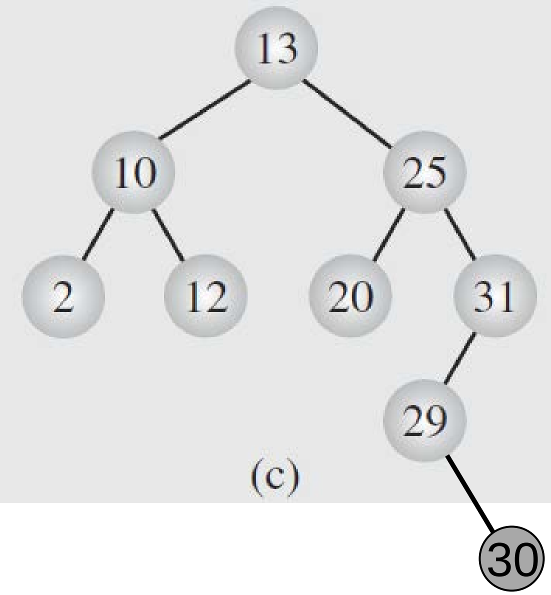
the stack remembers the backtrack point, then visit the branch point node, and proceed to the right

Tree Traversal (cont.)



Insertion

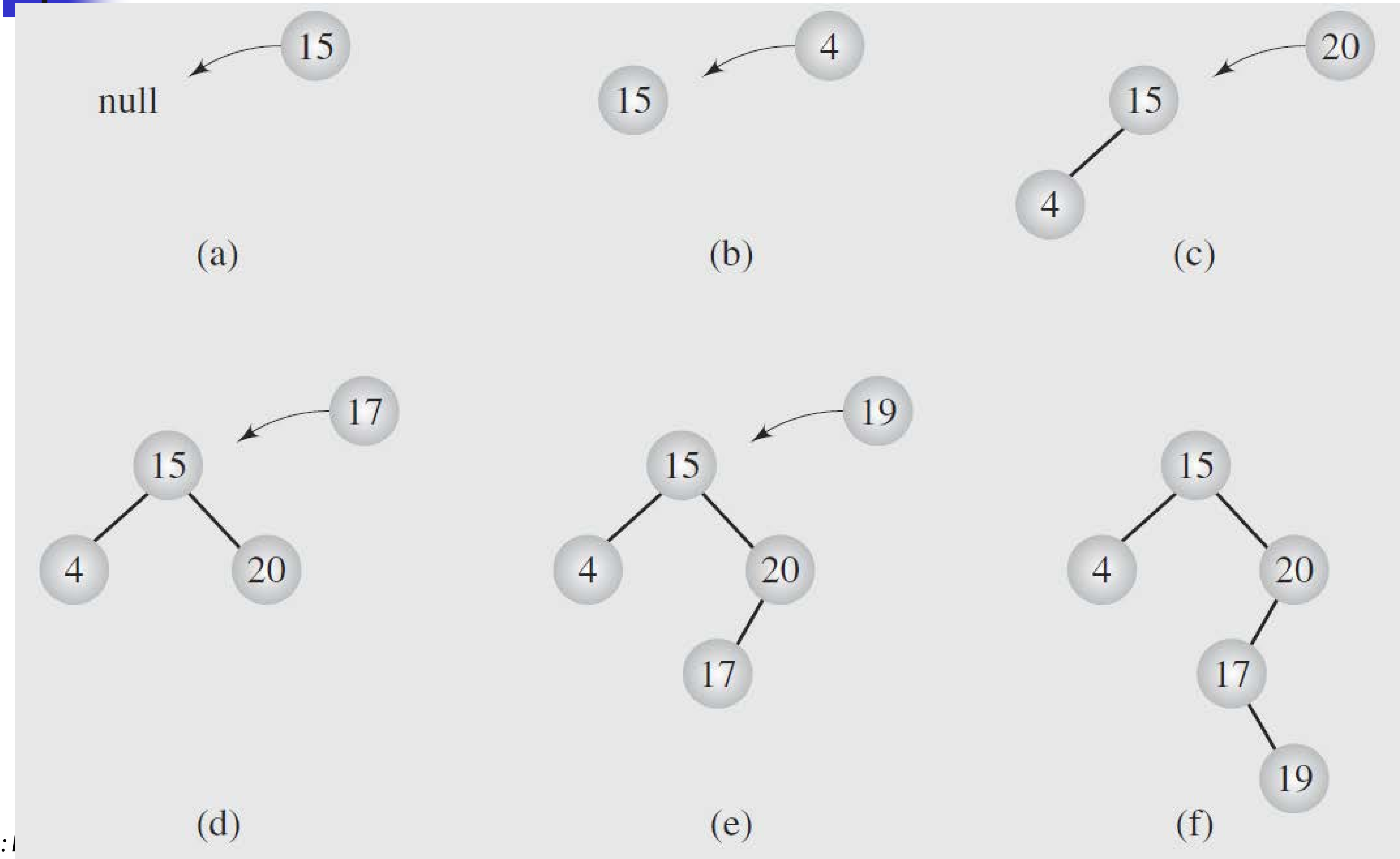
- Searching a binary tree
  - does not modify the tree
- Tree traversals can change the tree
  - depending on visit()
  - operations like insertions, deletions, modifying values, etc.
    - **alter the tree structure**



Insertion

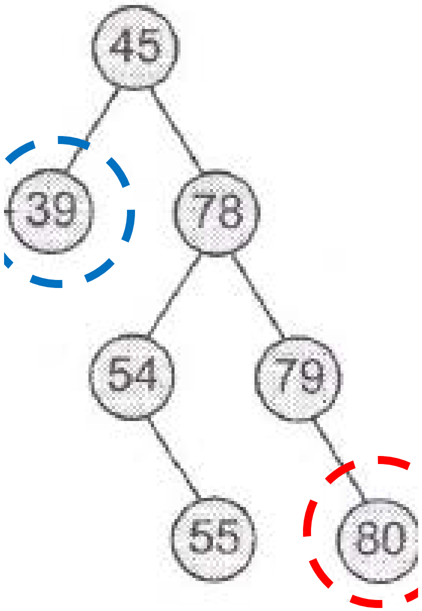
- Insert a new node in a binary tree??
  - perform in the same way as searching
  - compare the value of the node to be inserted to the current node
  - if the value to be inserted is smaller,
    - follow the left subtree;
  - if it is larger,
    - follow the right subtree;
  - if the child branch we are to follow is empty,
    - stop the search and insert the new node as that child
  - E.g., insert node 30

Insertion (cont.)



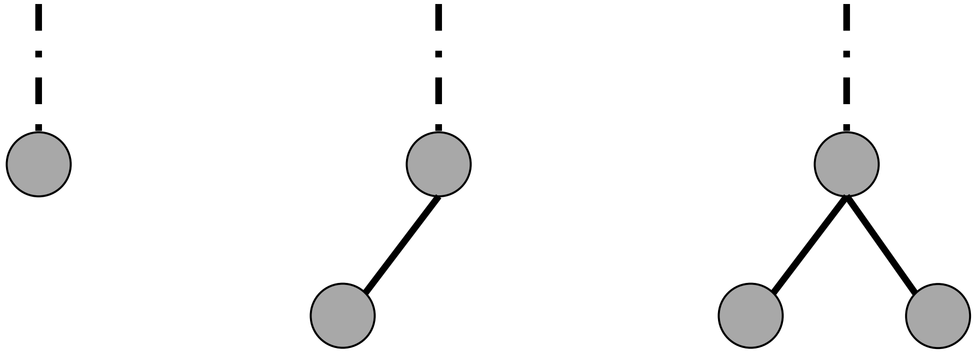
Search

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



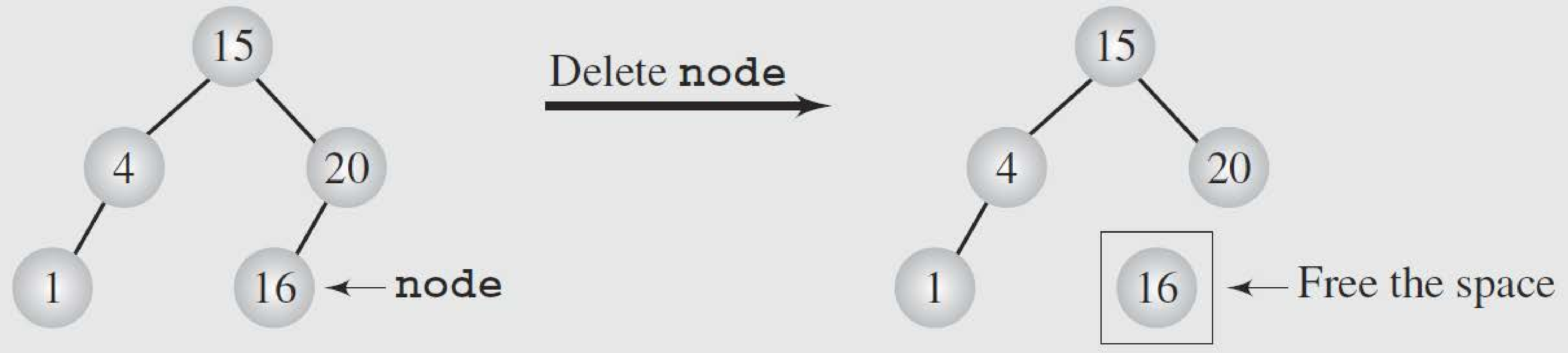
Deletion

- 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



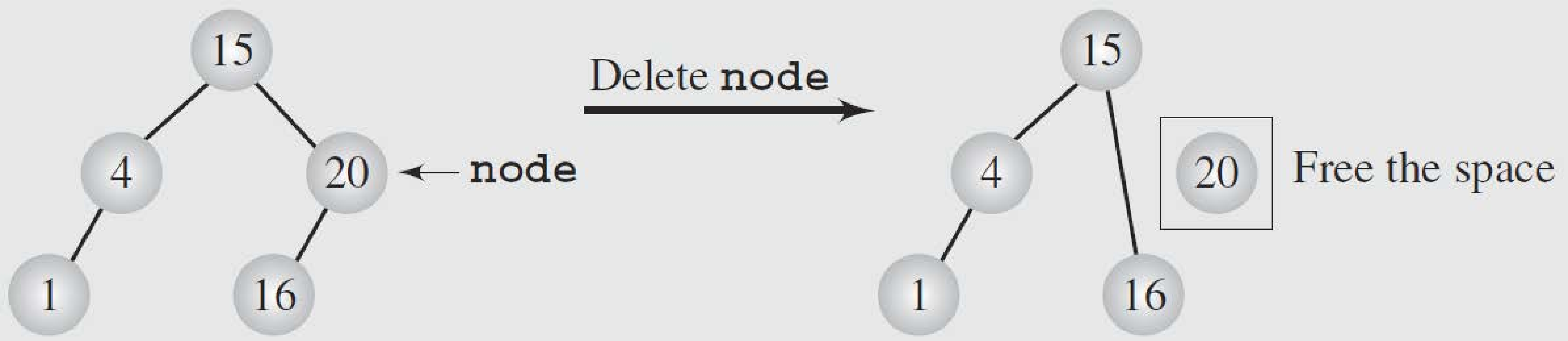
Deletion

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



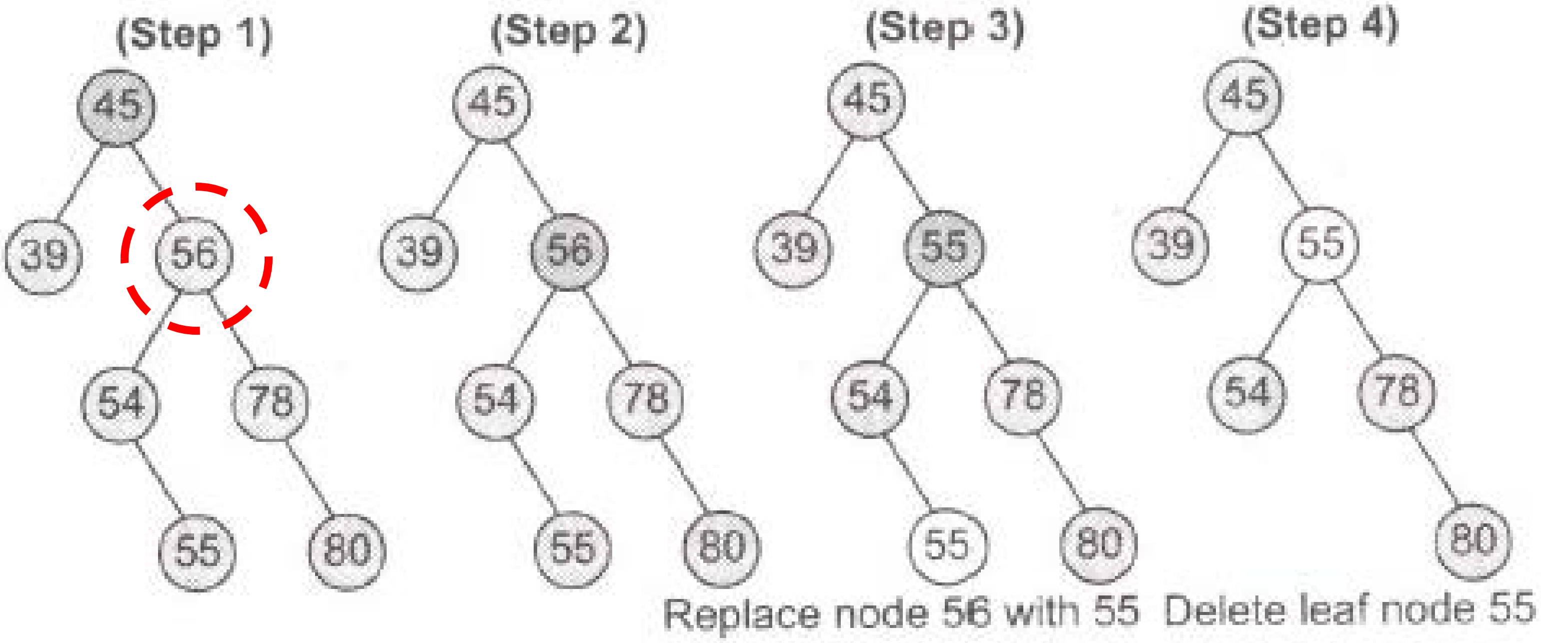
Deletion

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



Deletion

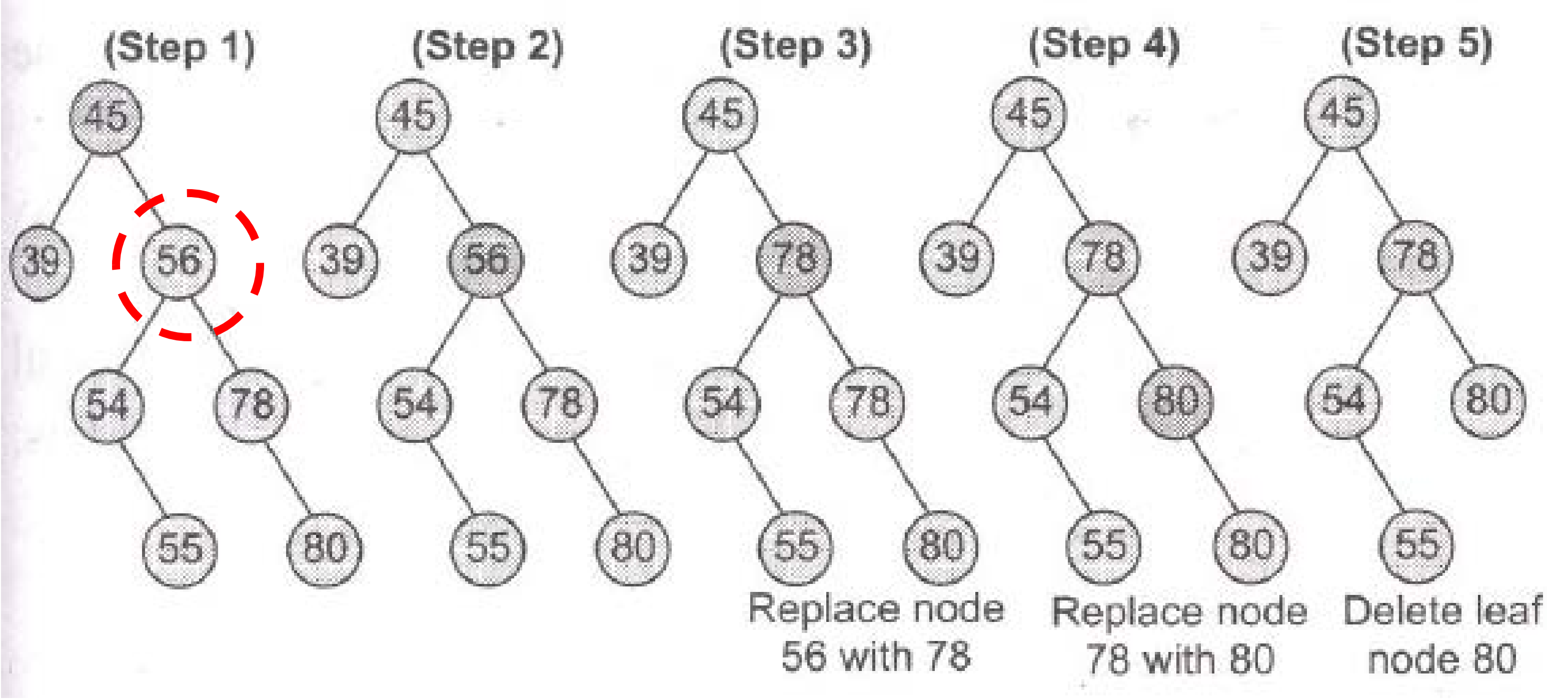
- deleting a node with **two children** (e.g., delete 56)



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**