



Trees

Dr. Megha Ummat

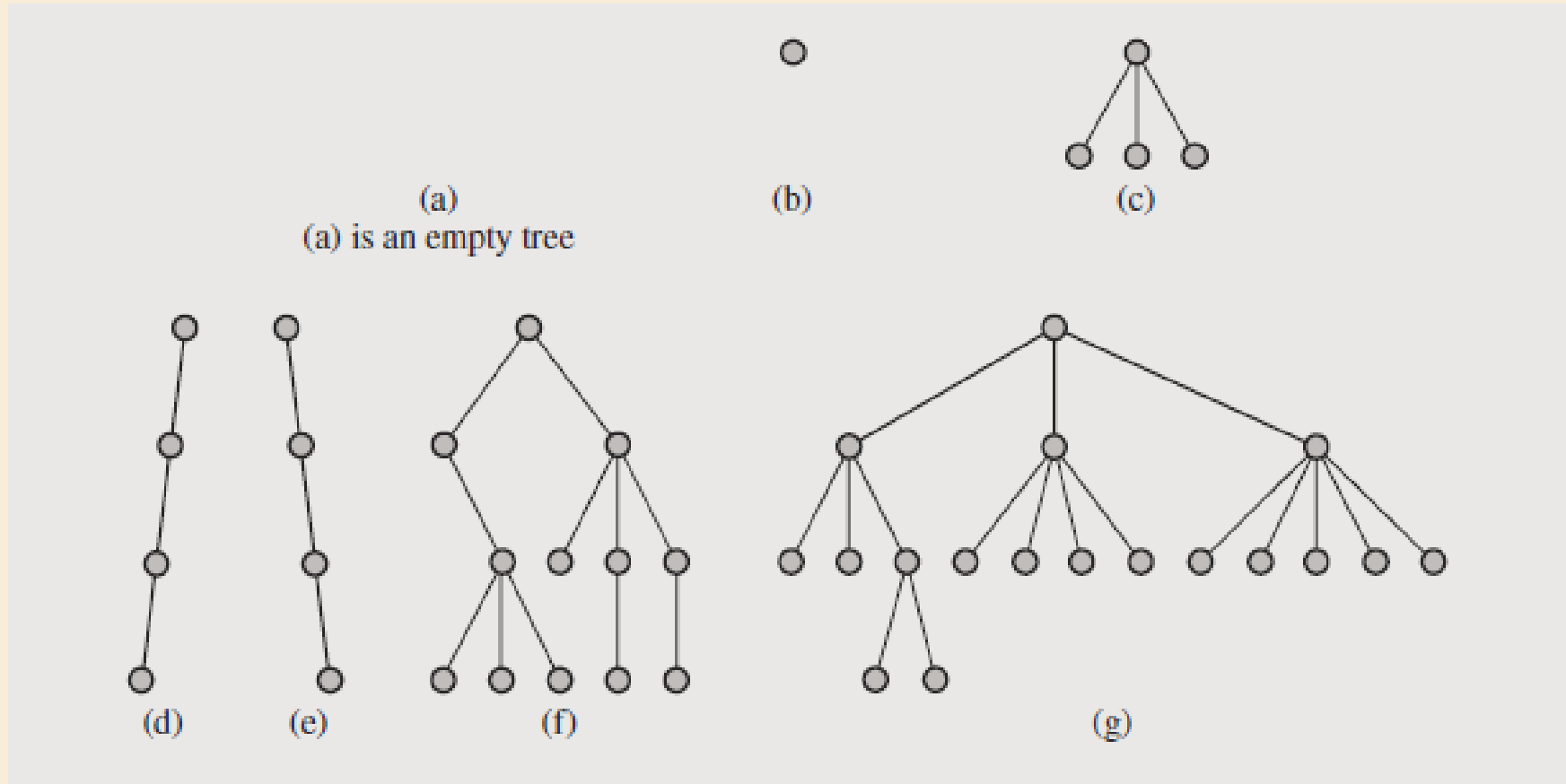
Trees

- Linked lists usually provide greater flexibility than arrays.
- But they are linear structures. Difficult to use them to organize a hierarchical representation of objects.
- To overcome this limitation, we create a new data type called a *tree* that consists of *nodes* and *arcs*.
- Unlike natural trees, these trees are depicted upside down with the *root* at the top and the *leaves (terminal nodes)* at the bottom.
- The root is a node that has no parent; it can have only child nodes. Leaves, on the other hand, have no children, or rather, their children are empty structures.

Trees

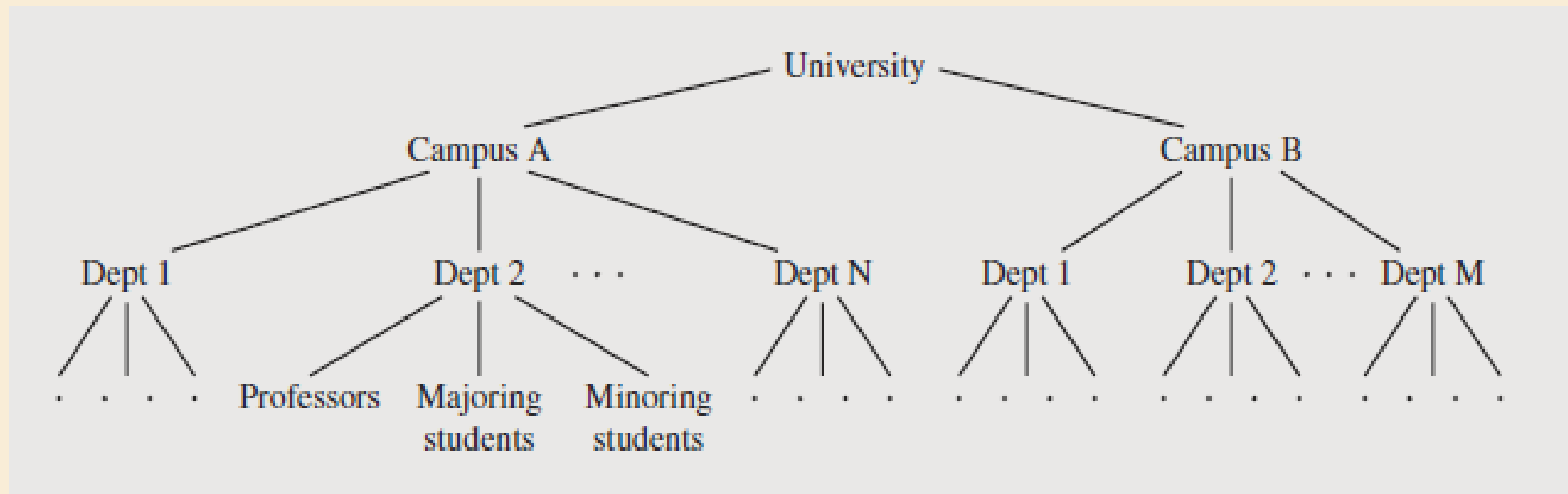
- A tree can be defined recursively as the following:
- **Rule 1:** An empty structure is an empty tree.
- **Rule 2:** If t_1, \dots, t_k are disjoint trees, then the structure whose root has as its children the roots of t_1, \dots, t_k is also a tree.
- **Rule 3:** Only structures generated by rules 1 and 2 are trees.

Example of Trees



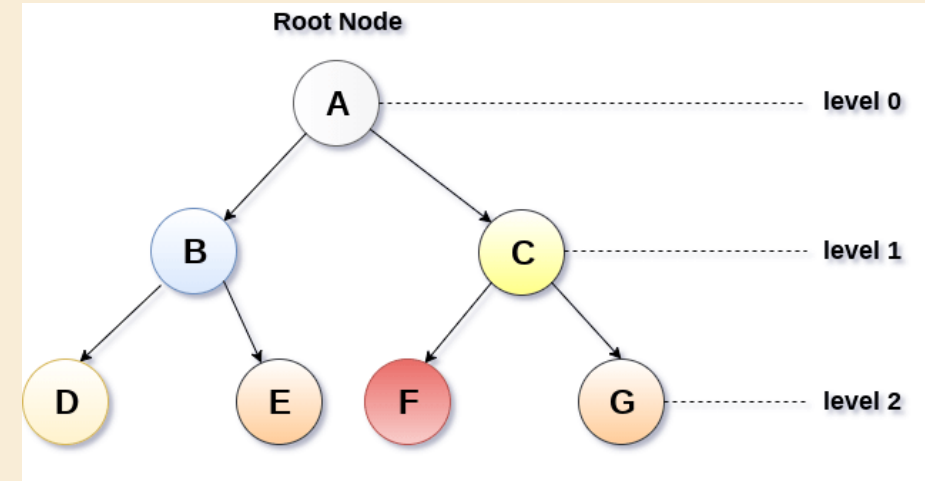
Example (d) and (e) are called degenerate trees as they resemble a linked list.

University Example



Some Terminologies

- Path
 - Each node has to be reachable from the root through a unique sequence of arcs, called a *path*.
 - The number of arcs in a path is called the *length* of the path.
- Level
 - The *level* of a node is the length of the path from the root to the node.

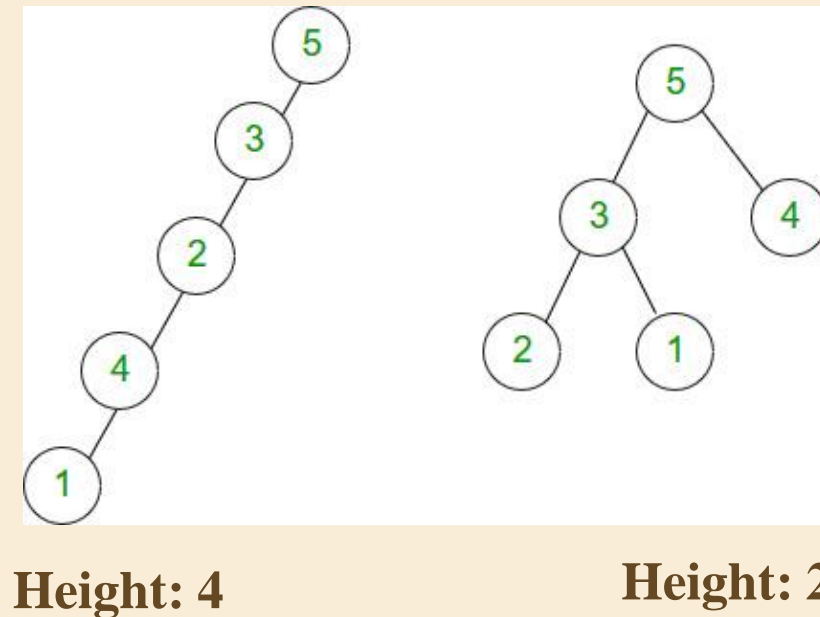
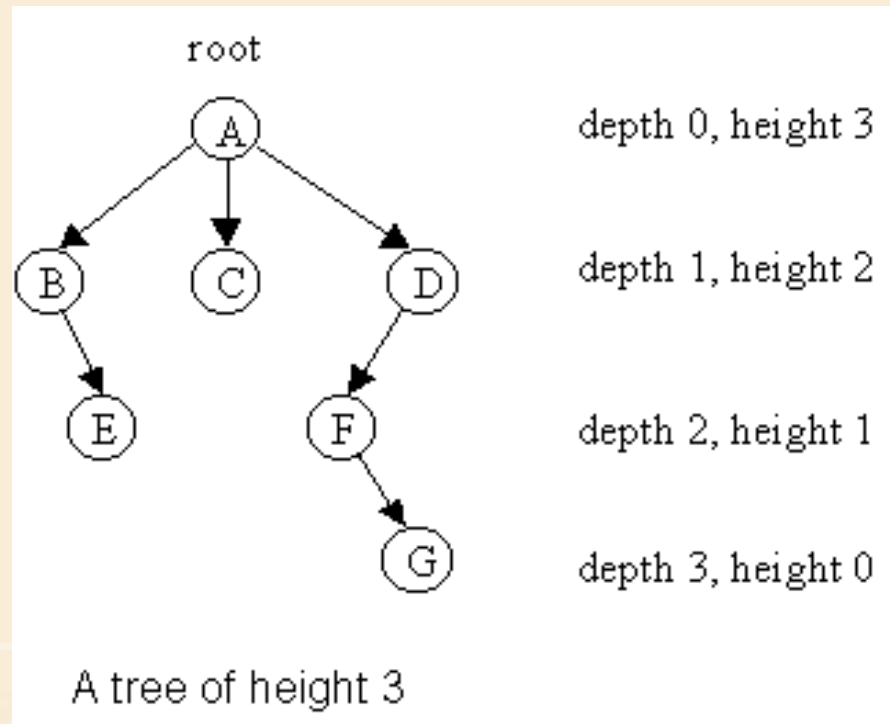


Some Terminologies

- Depth & Height
 - The depth of a node is the number of edges from the node to the tree's root node. A root node will have a depth of 0. Depth is same as level.
 - Depth of a tree is the maximum level of the tree.
 - The height of a node is the number of edges on the longest path from the node to a leaf.
 - The empty tree is a legitimate tree of height 0 (by definition), and a single node is a tree of height 1. This is the only case in which a node is both the root and a leaf.

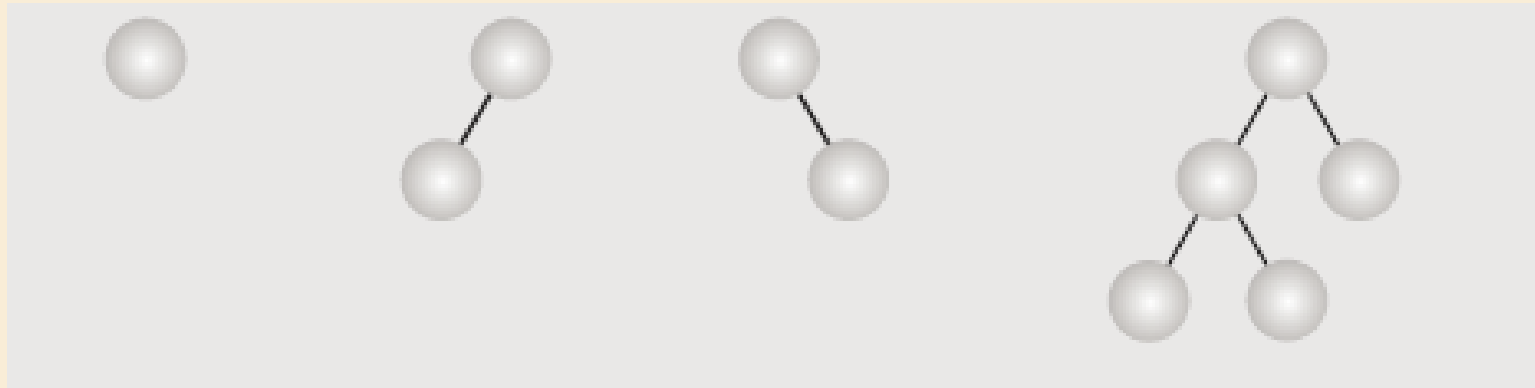
Some Terminologies

- Depth & Height



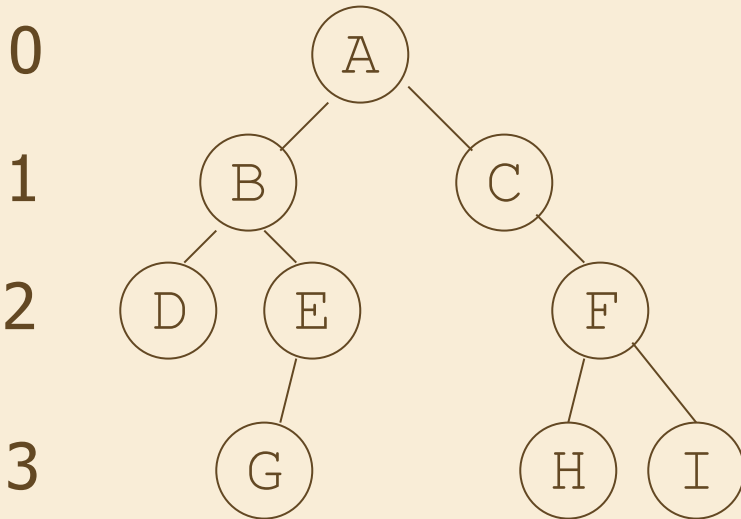
Binary Tree

- A *binary tree* is a tree whose nodes have two children (possibly empty), and each child is designated as either a left child or a right child.



An example of binary tree

level



root: A

node: A, B, C, ..., H, I

father of B: A

sons of B: D, E

left son of B: D

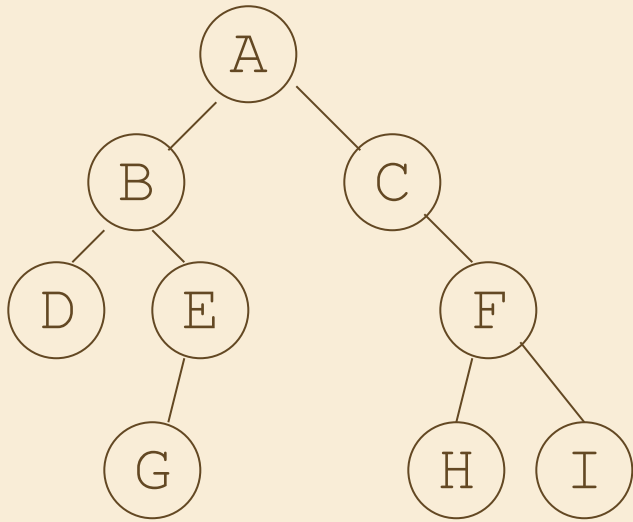
right son of B: E

depth: 3

ancestors of E: A, B

descendants of B: D, E, G

An example of binary tree



left descendant of B: D

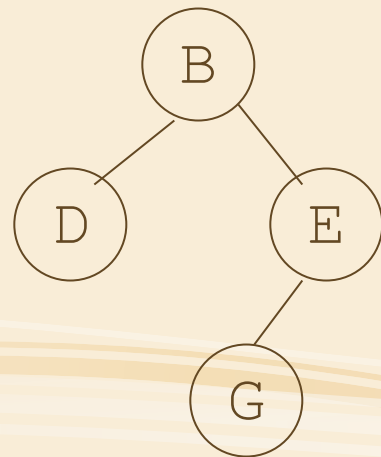
right descendant of B: E, G

brother: B and C are brothers

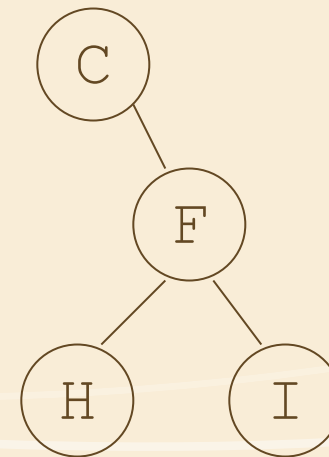
D and E are brothers

leaf: a node that has no sons e.g. D, G, H, I

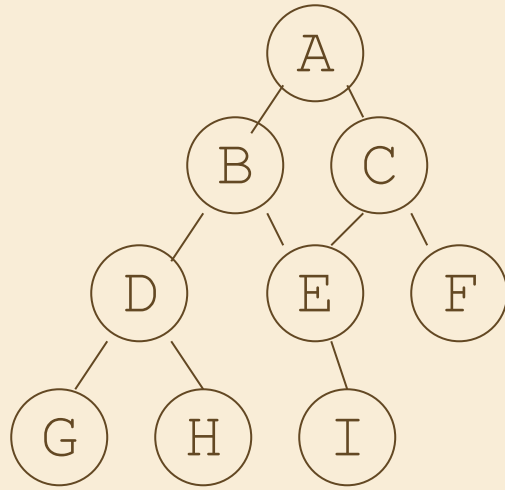
left subtree of A:



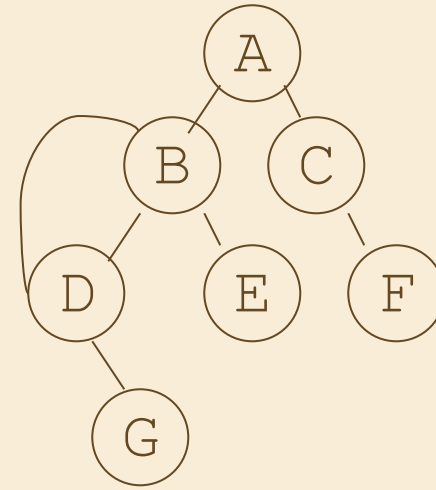
right subtree of A:



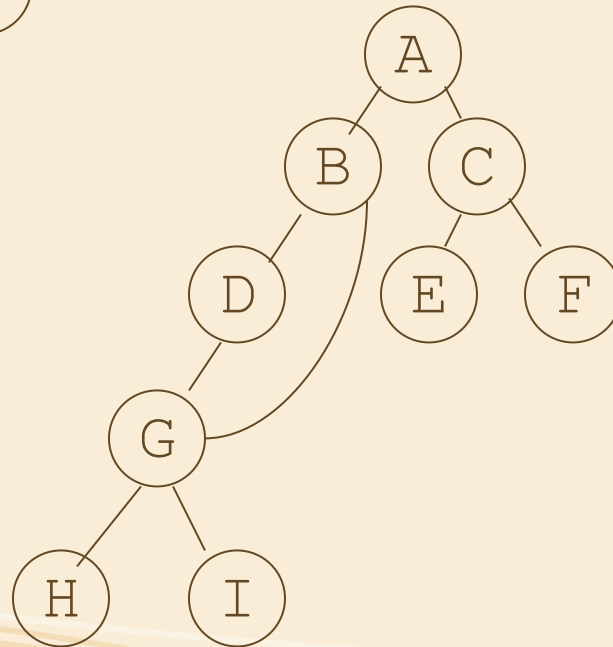
Not binary trees



(a)



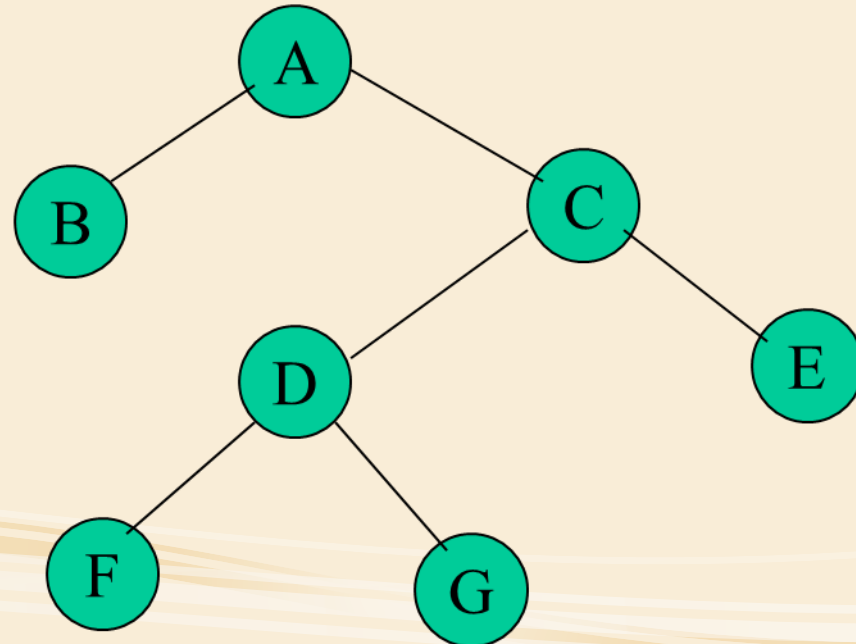
(b)



(c)

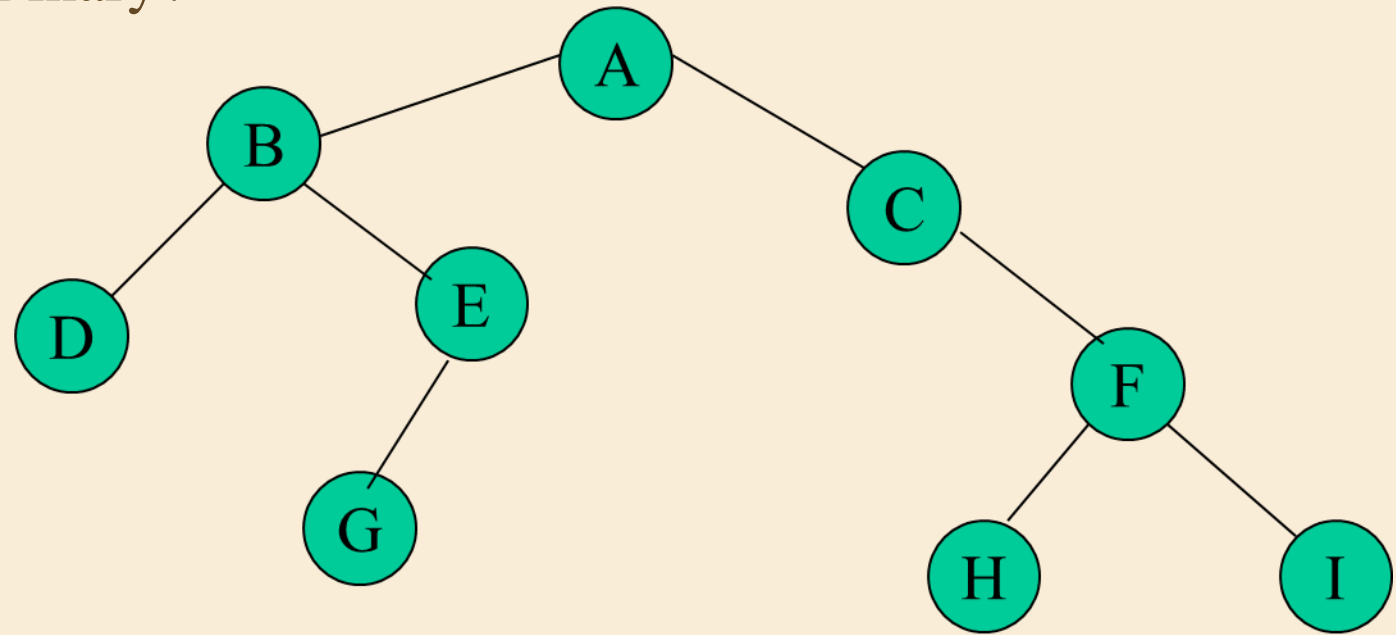
Strictly Binary Tree

- If every non-leaf node in a binary tree has nonempty left and right subtrees, the tree is called a strictly binary tree.
- A strictly binary tree with n leaves always contains $2n - 1$ nodes.



Strictly Binary Tree

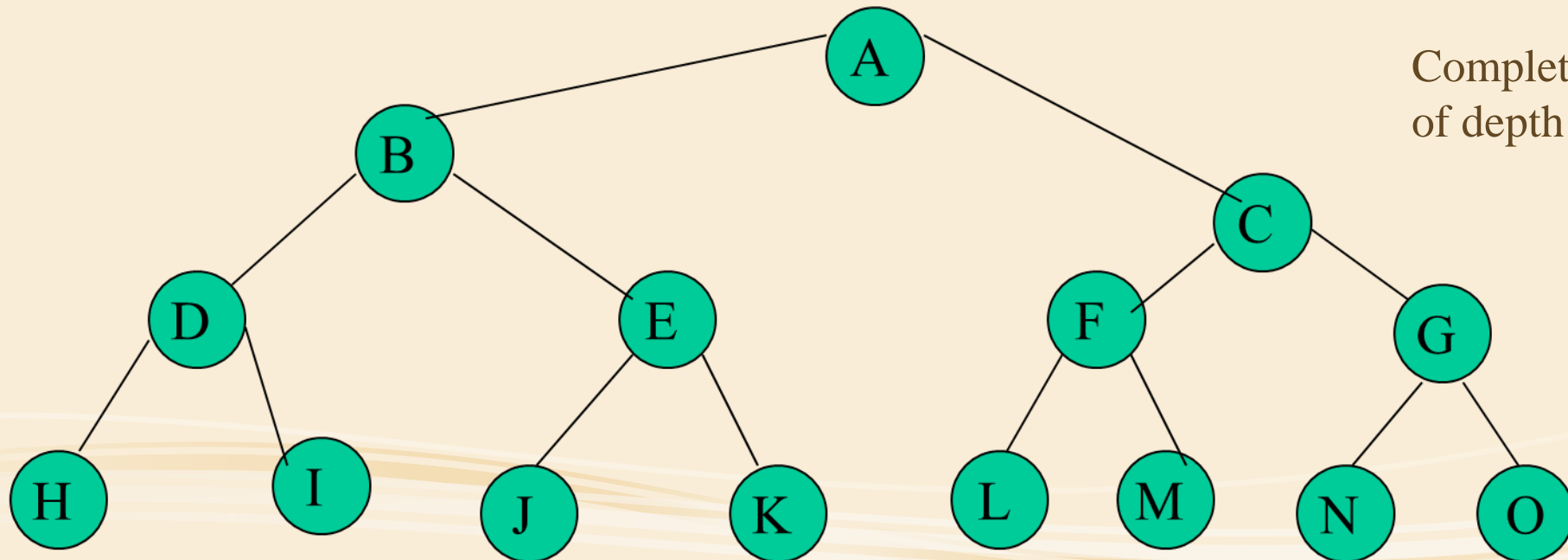
Is the given tree Strictly Binary?



Structure that is not a strictly binary tree:
because nodes C and E have one son each.

Complete Binary Tree

- A binary tree in which all non-terminal nodes have both their children, and all leaves are at the same level.
- A complete binary tree of depth d is the strictly binary tree all of whose leaves are at level d .



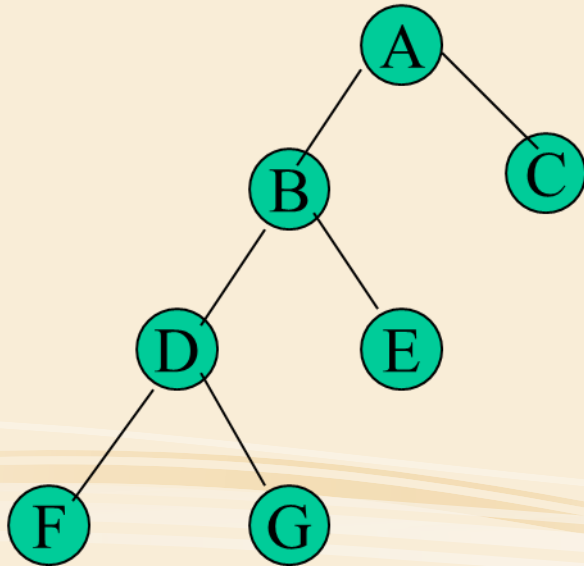
Complete binary tree
of depth 3

Complete Binary Tree

- A complete binary tree of depth d is the binary tree of depth d that contains exactly 2^i nodes at each level i between 0 and d .
- The total number of nodes = the sum of the number of nodes at each level between 0 and $d = 2^{d+1} - 1$

Almost Complete Binary Tree

- A binary tree of depth d is an almost complete binary tree if:
 - 1. A node n at level less than $d - 1$ has two sons.
 - 2. For any node n in the tree with a right descendant at level d , n must have a left son and every left descendant of n is either a leaf at level d or has two sons.

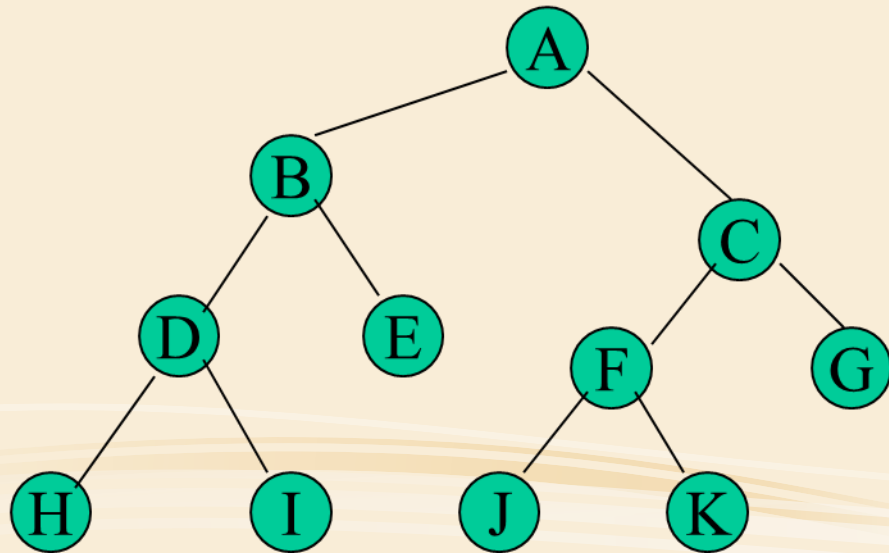


The strictly binary tree is not almost complete, since it contains leaves at levels 1, 2, and 3.

Violates condition 1

Almost Complete Binary Tree

- A binary tree of depth d is an almost complete binary tree if:
 - 1. A node n at level less than $d - 1$ has two sons.
 - 2. For any node n in the tree with a right descendant at level d , n must have a left son and every left descendant of n is either a leaf at level d or has two sons.



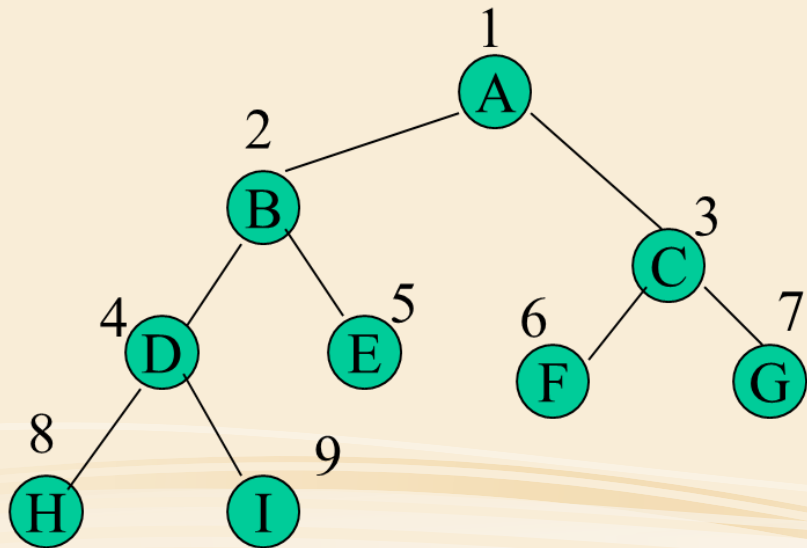
The strictly binary tree is not almost complete, since A has a right descendant at level 3 (J) but also has a left descendant that is a leaf at level 2 (E)

Violates condition 2

Satisfies the condition 1, since every leaf node is either at level 2 or at level 3.

Almost Complete Binary Tree

- A binary tree of depth d is an almost complete binary tree if:
 - 1. A node n at level less than $d - 1$ has two sons.
 - 2. For any node n in the tree with a right descendant at level d , n must have a left son and every left descendant of n is either a leaf at level d or has two sons.

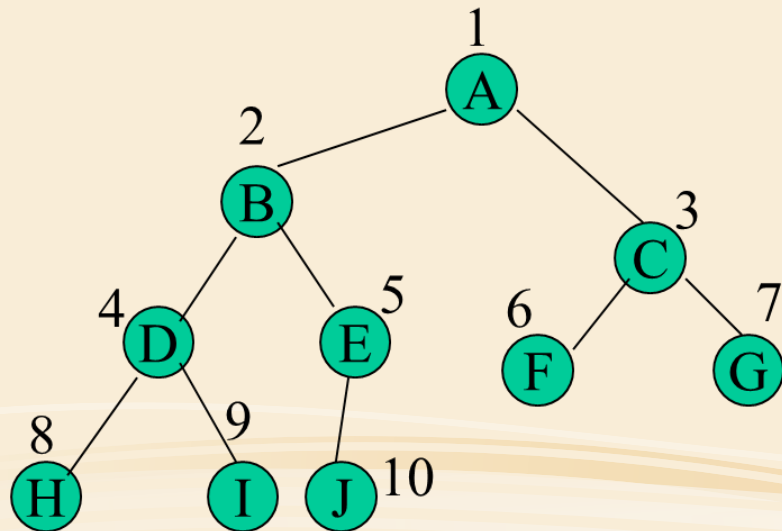


The binary tree is almost complete,
Satisfies the condition 1, since every
leaf node is either at level 2 or at
level 3.

Satisfies the condition 2

Almost Complete Binary Tree

- A binary tree of depth d is an almost complete binary tree if:
 - 1. A node n at level less than $d - 1$ has two sons.
 - 2. For any node n in the tree with a right descendant at level d , n must have a left son and every left descendant of n is either a leaf at level d or has two sons.



The binary tree is almost complete. Satisfies the condition 1, since every leaf node is either at level 2 or at level 3.

Satisfies the condition 2.

However, the binary tree is not strictly binary tree, since node E has a left son but not a right son

Observations

- An almost complete binary tree with n leaves has $2n-1$ nodes (Figure 1).
- An almost complete binary tree with n leaves that is not strictly binary has $2n$ nodes (Figure 2).
- There is only a single almost complete binary tree with n nodes. This tree is strictly binary if and only if n is odd.

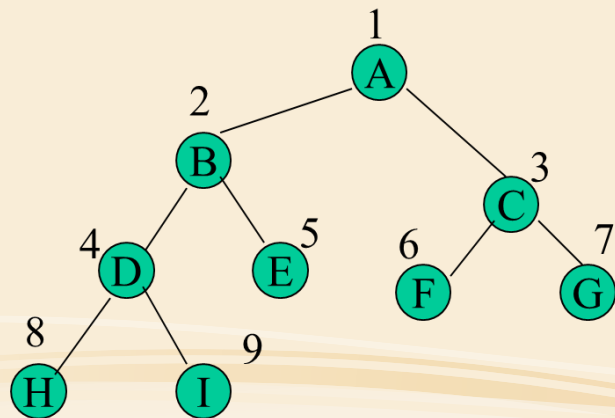


Figure 1

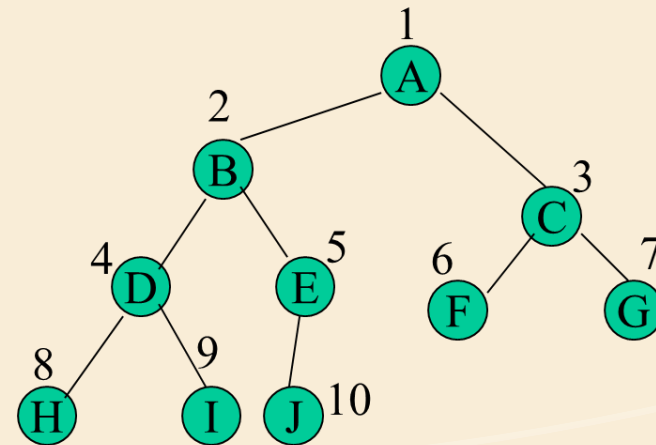


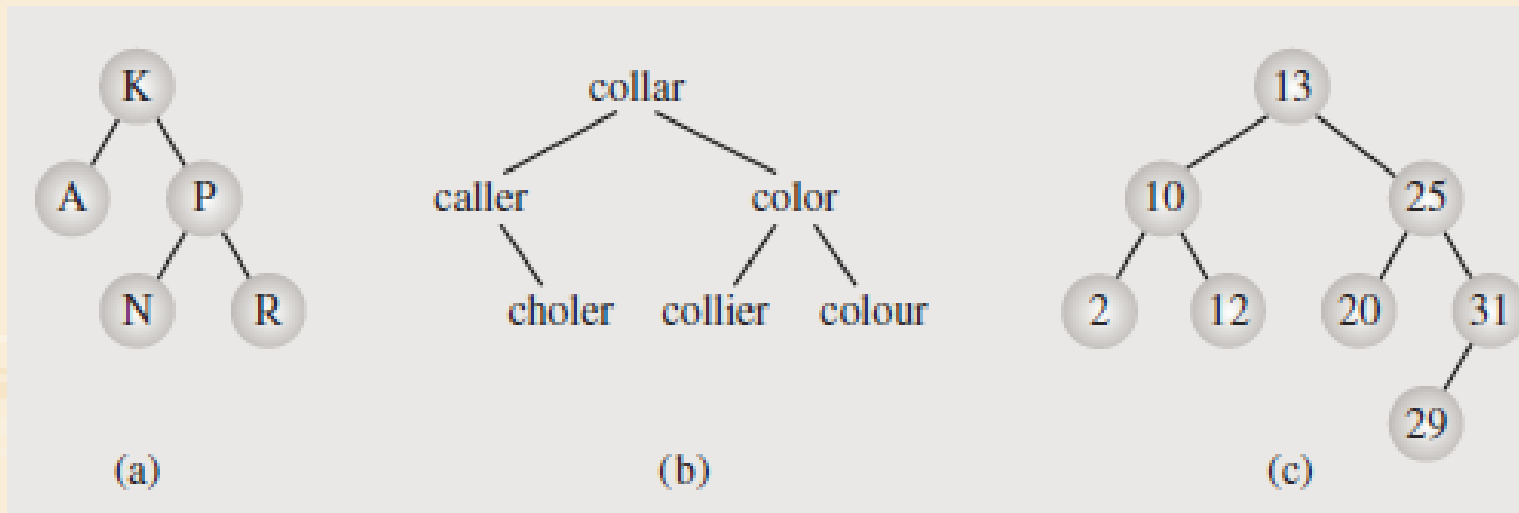
Figure 2

Observations

- For all the nonempty binary trees whose nonterminal nodes have exactly two nonempty children, the number of leaves m is greater than the number of nonterminal nodes k and $m = k + 1$.
- It implies that an i –level complete decision tree has 2^i leaves, and due to the preceding observation, it also has $2^i - 1$ nonterminal nodes, which makes $2^i + 2^i - 1 = 2^{i+1} - 1$ nodes in total.

Binary Search Tree

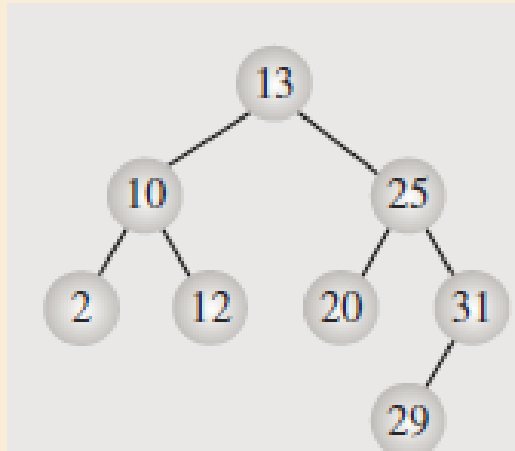
- Binary search tree has the following property:
 - For each node n of the tree, all values stored in its left subtree (the tree whose root is the left child) are less than value v stored in n , and all values stored in the right subtree are greater than or equal to v .
- Also called Ordered Binary Trees
- Multiple copies of the same value in the same tree is avoided.



Implementing Binary Trees

- Binary trees can be implemented as arrays or as linked structures.
- To implement a tree as an array, a node is declared as a structure with an information field and two “pointer” fields.
- These pointer fields contain the indexes of the array cells in which the left and right children are stored, if there are any.
- The root is always located in the first cell, cell 0, and -1 indicates a null child.

Implementing Binary Trees



Index	Info	Left	Right
0	13	4	2
1	31	6	-1
2	25	7	1
3	12	-1	-1
4	10	5	3
5	2	-1	-1
6	29	-1	-1
7	20	-1	-1

Implementing Binary Trees

- Locations of children must be known to insert a new node, and these locations may need to be located sequentially.
- After deleting a node from the tree, a hole in the array would have to be eliminated. This can be done either by using a special marker for an unused cell, which may lead to populating the array with many unused cells, or by moving elements by one position, which also requires updating references to the elements that have been moved.

BST Node

```
template<class T>
class BSTNode
{
    public:

    T key;
    BSTNode *left,*right;

    BSTNode()
    {
        left = right = 0;
    }

    BSTNode(const T& el, BSTNode *l=0, BSTNode *r = 0)
    {
        key = el;
        left = l;
        right = r;
    }
};
```

```

template<class T>
class BST
{
    BSTNode<T> *root;
    T* search(BSTNode<T>*,const T&)const;
    void preorder(BSTNode<T>*);
    void inorder(BSTNode<T>*);
    void postorder(BSTNode<T>*);
    void clear (BSTNode<T> *);
    void visit(BSTNode<T>* p)
    {
        cout<<p->key<<" ";
    }

    public:
    BST()
    {
        root = 0;
    }

    ~BST() {      clear();
}

```

```

void clear()
{
    clear(root);
    root = 0;
}

bool isempty() const
{
    return (root == 0);
}

void preorder()
{
    preorder(root);
}

void inorder()
{
    inorder(root);
}

```

```
void postorder()
{
    postorder(root);
}

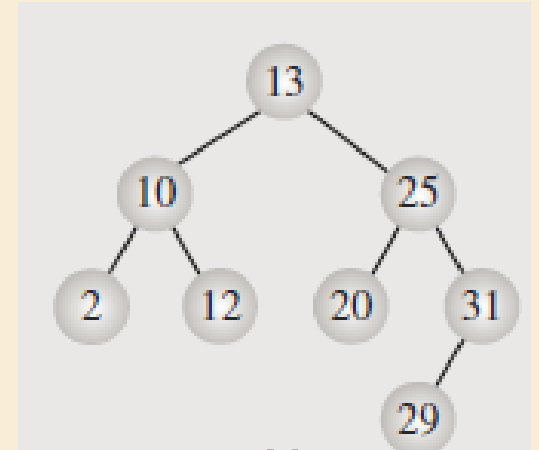
T* search(const T& el)const
{
    return search(root,el);
}

void insert(const T&);
void del_copy(BSTNode<T>*& n);
void del_merge(BSTNode<T>*& n);
void find_del_copy(const T& el);
void find_del_merge(const T& el);
void breadthFirst();
};
```

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

Searching a BST

- For every node, compare the element to be located with the value stored in the node currently pointed at.
- If the element is less than the value, go to the left subtree and try again.
- If it is greater than that value, try the right subtree.
- If it is the same, the search can be discontinued.
- The search is also aborted if there is no way to go, indicating that the element is not in the tree.




For locating 31, only 3 tests need to be performed.

Searching a BST

```
template<class T>
T* BST<T>::search(BSTNode<T> *p,const T& el)const
{
    while (p!=0)
    {
        if (el == p->key)
            return &p->key;
        else if (el < p->key)
            p = p->left;
        else p = p->right;
    }
    return 0;
}
```

Run Time of Search

- The complexity of searching is measured by the number of comparisons performed during the searching process.
 - This number depends on the number of nodes encountered on the unique path leading from the root to the node being searched for.
 - Therefore, the complexity is the length of the path leading to this node plus 1.
 - Complexity depends on the shape of the tree and the position of the node in the tree.
- 

Internal Path Length

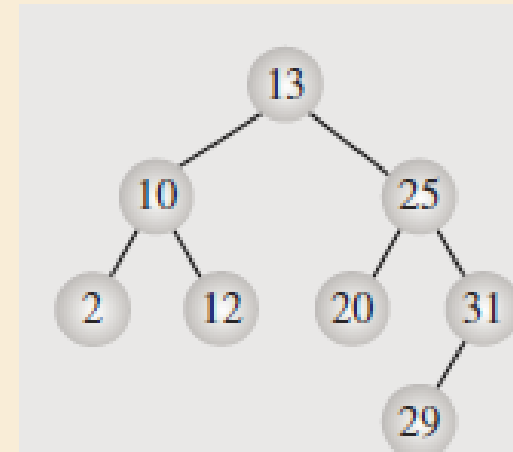
- The *internal path length (IPL)* is the sum of all path lengths of all nodes, which is calculated by summing $\sum (i - 1)l_i$ over all levels i , where l_i is the number of nodes on level i . (Assuming level starts from 1)
- A depth of a node in the tree is determined by the path length.
- An average depth, called an *average path length*, is given by the formula IPL/n , which depends on the shape of the tree.
- In the worst case, when the tree turns into a linked list,

$$pathworst = \frac{1}{n} \sum_{i=1}^n (i - 1) = \frac{n-1}{2} = O(n)$$

and a search can take n time units.

Internal Path Length

- Compute Internal Path Length.



Searching: Best Case

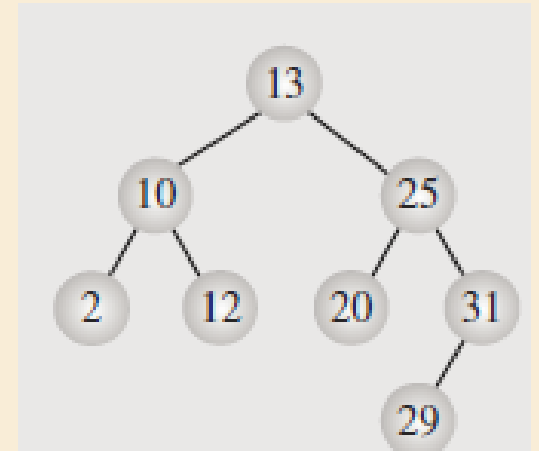
- The best case occurs when all leaves in the tree of height h are in at most two levels, and only nodes in the next to last level can have one child.
- To simplify the computation, we approximate the average path length for such a tree, *pathbest*, by the average path of a complete binary tree of the same height.
- Height of Tree = $O(\lg n)$

Tree Traversal

- *Tree traversal* is the process of visiting each node in the tree exactly one time.
- Traversal may be interpreted as putting all nodes on one line or linearizing a tree.
- For a tree with n nodes there are $n!$ different traversals.
- Two classes of traversals seem of use:
 - Breadth First Traversal
 - Depth First Traversal

Breadth First Traversal

- *Breadth-first traversal* is visiting each node starting from the lowest (or highest) level and moving down (or up) level by level, visiting nodes on each level from left to right (or from right to left).
- Consider a top-down, left-to-right, breadth-first traversal.
- Implementation of this kind of traversal is straightforward when a queue is used.
- After a node is visited, its children, if any, are placed at the end of the queue, and the node at the beginning of the queue is visited.



```

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

```


Breadth First Traversal

- Considering that for a node on level n , its children are on level $n + 1$, by placing these children at the end of the queue, they are visited after all nodes from level n are visited.
- Thus, the restriction that all nodes on level n must be visited before visiting any nodes on level $n + 1$ is met.

Depth First Traversal

- *Depth-first traversal* proceeds as far as possible to the left (or right), then backs up until the first crossroad, goes one step to the right (or left), and again as far as possible to the left (or right).
- We repeat this process until all nodes are visited.
- There are three tasks of interest in this type of traversal:
 - V—visiting a node
 - L—traversing the left subtree
 - R—traversing the right subtree

Depth First Traversal

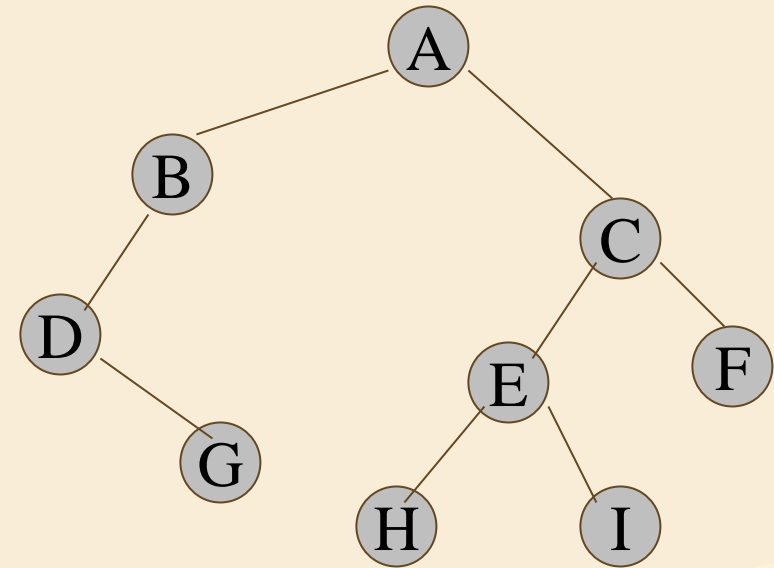
- VLR—preorder tree traversal
 - LVR—inorder tree traversal
 - LRV—postorder tree traversal
- 

Inorder

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

1. Traverse the left subtree in inorder
2. Visit the root
3. Traverse the right subtree in inorder

inorder: DGBAHEICF

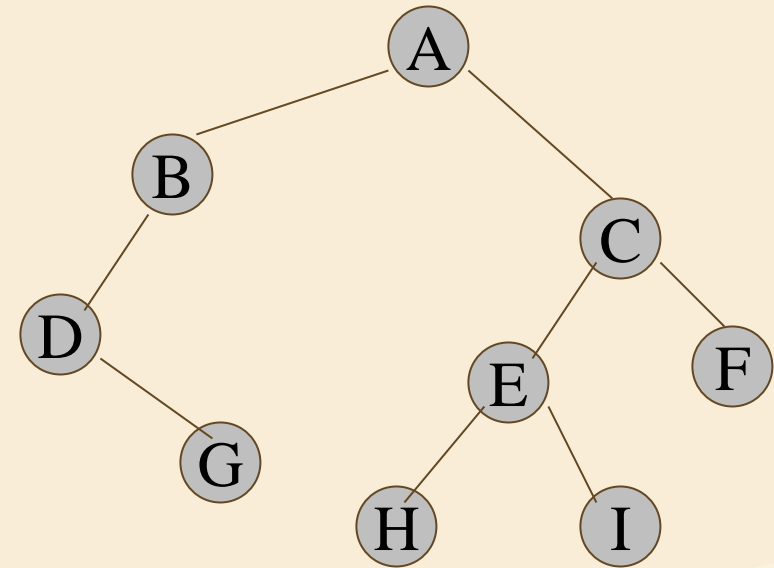


Preorder

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

1. Visit the root
2. Traverse the left subtree in preorder
3. Traverse the right subtree in preorder

preorder: ABDGCEHIF

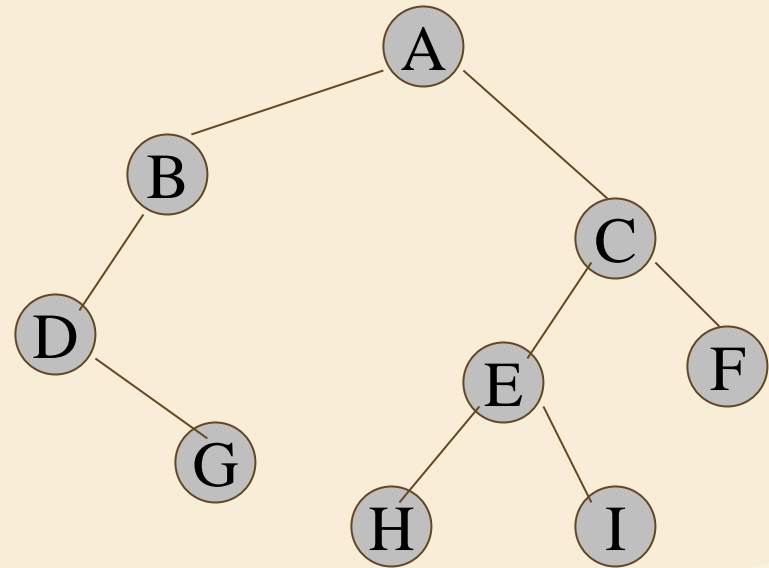


Postorder

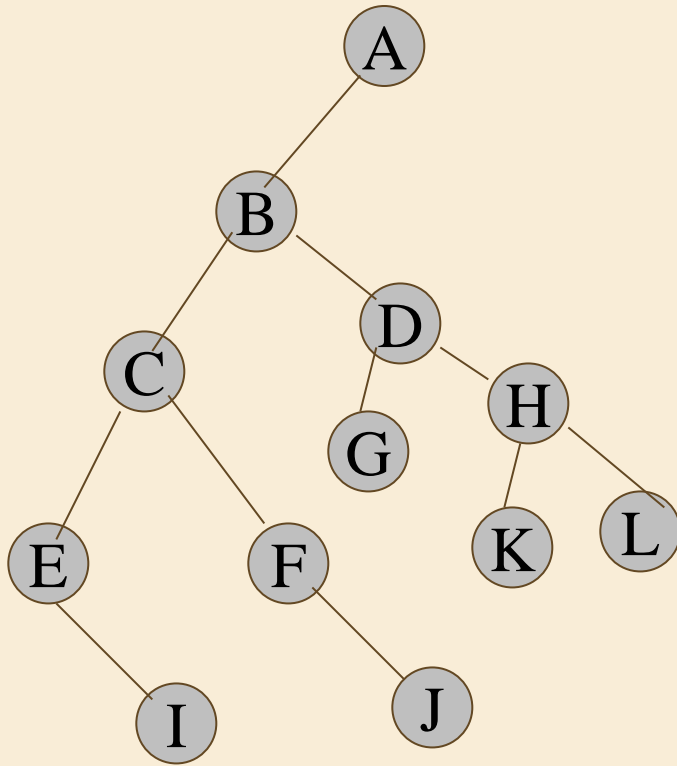
```
void BST<T>::postorder(BSTNode<T> *p)
{
    if (p != 0 )
    {
        postorder(p->left);
        postorder(p->right);
        visit(p);
    }
}
```

1. Traverse the left subtree in postorder
2. Traverse the right subtree in postorder
3. Visit the root

postorder: GDBHIEFCA



Traversing a binary tree

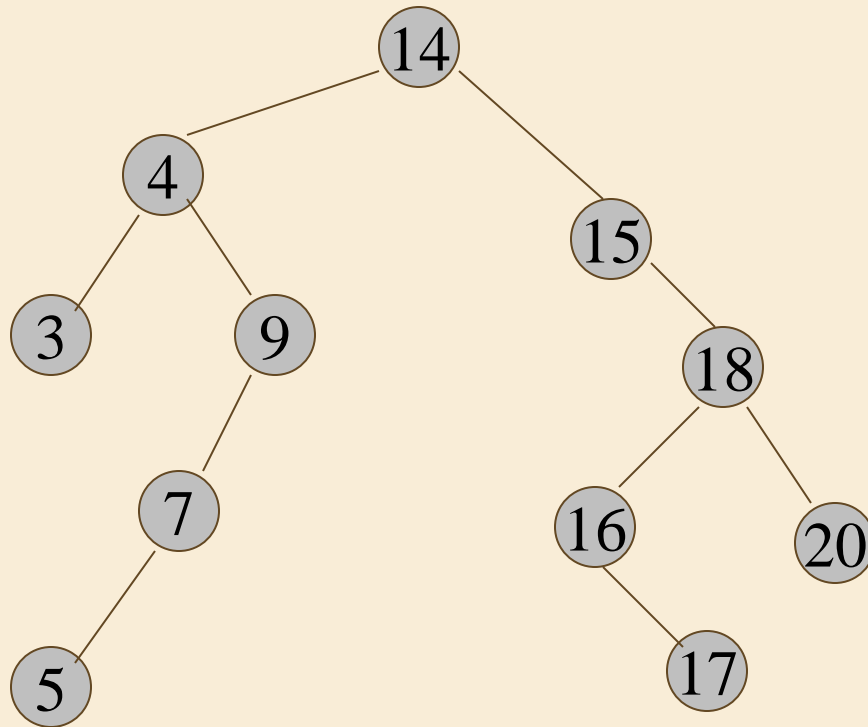


preorder: ABCEIFJDGHKL

inorder: EICFJBGDKHLA

postorder: IEJFCGKLHDBA

Traversing a binary tree



Preorder: 14, 4, 3, 9, 7, 5, 15, 18, 16, 17, 20

Inorder: 3, 4, 5, 7, 9, 14, 15, 16, 17, 18, 20

Postorder: 3, 5, 7, 9, 4, 17, 16, 20, 18, 15, 14

Run Time Stack during Inorder Traversal

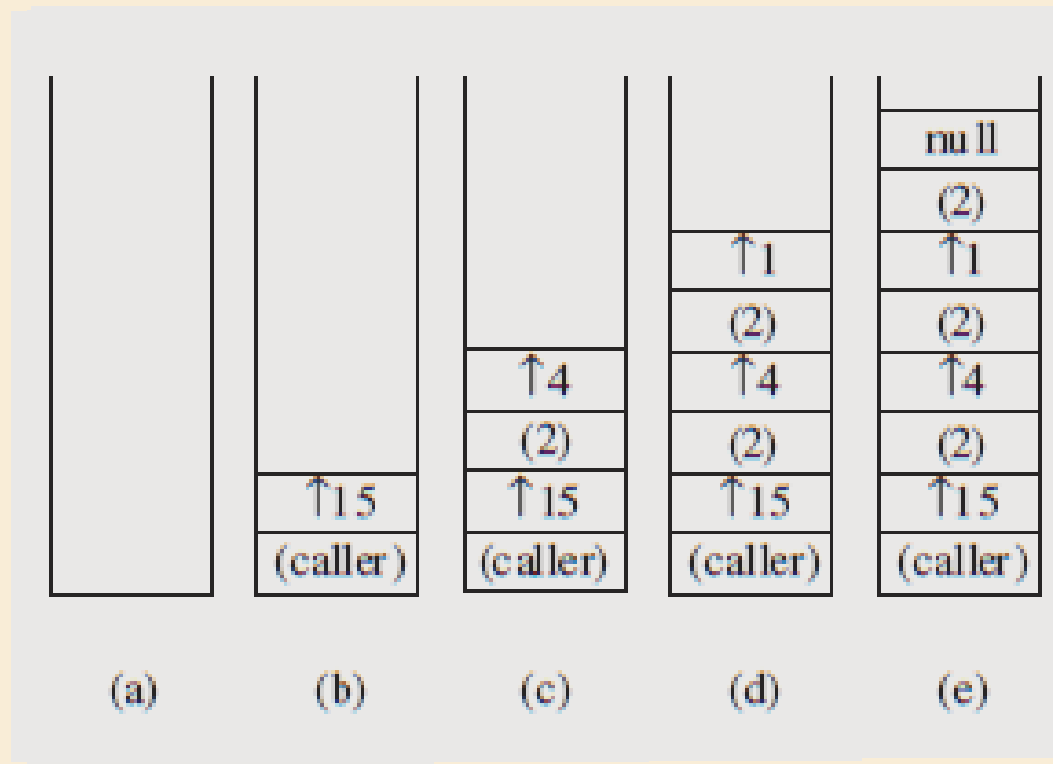
- Let us understand the way inorder() works by observing the behavior of the runtime stack.
- **Some Notations**
 - Numbers in parenthesis - Return Addresses
 - ↑4 – Node points to the node of the tree whose value is number 4

```
template<class T>
void BST<T>::inorder(BSTnode<T> *node) {
    if (node != 0) {
        /* 1 */      inorder(node->left);
        /* 2 */      visit(node);
        /* 3 */      inorder(node->right);
        /* 4 */      }
    }
```

Run Time Stack during Inorder Traversal

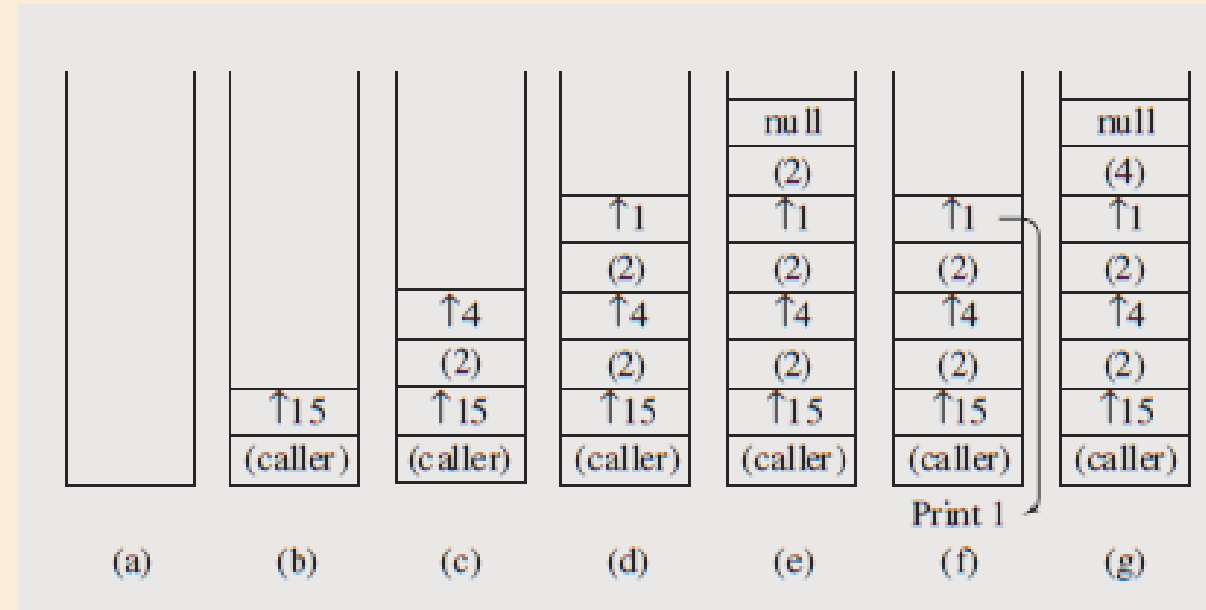
- a) Initially, we assume that the run-time stack is empty.
- b) Upon the first call, the return address of `inorder()` and the value of node, $\uparrow 15$, are pushed onto the run-time stack. The tree, pointed to by node, is not empty, the condition in the if statement is satisfied, and `inorder()` is called again with node 4.
- c) Before it is executed, the return address, (2), and current value of node, $\uparrow 4$, are pushed onto the stack. Because node is not null, `inorder()` is about to be invoked for node's left child, $\uparrow 1$.
- d) First, the return address, (2), and the node's value are stored on the stack.
- e) `inorder()` is called with node 1's left child. The address (2) and the current value of parameter node, null, are stored on the stack. Because node is null, `inorder()` is exited immediately; upon exit, the activation record is removed from the stack.

Run Time Stack during Inorder Traversal



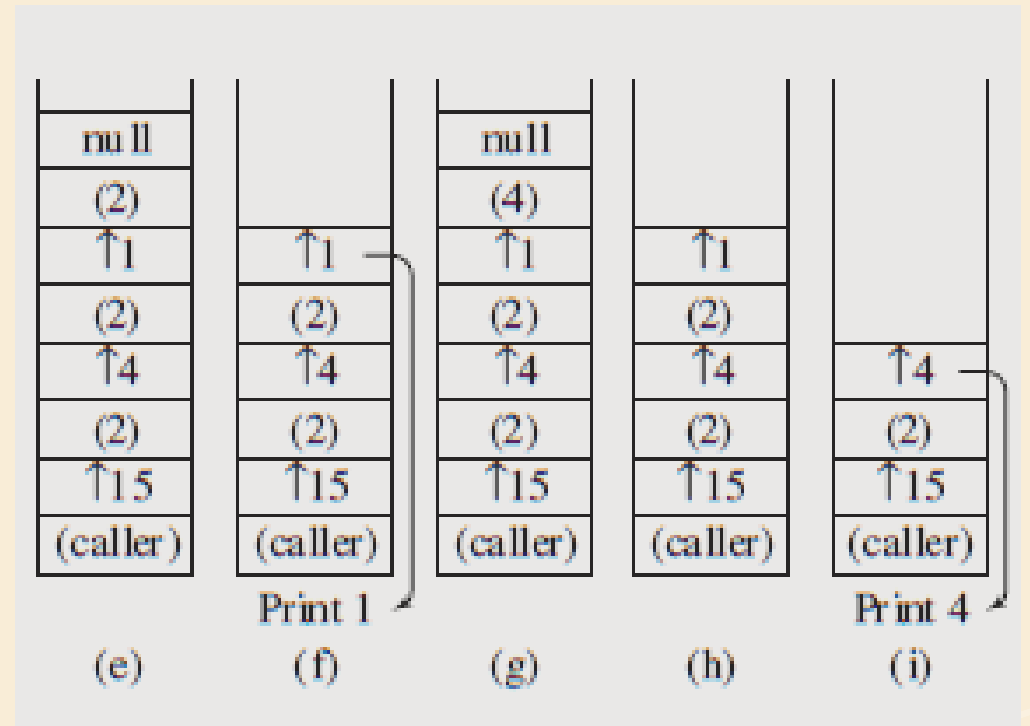
Run Time Stack during Inorder Traversal

- f) The system goes now to its run-time stack, restores the value of the node, $\uparrow 1$, executes the statement under (2), and prints the number 1. Because node is not completely processed, the value of node and address (2) are still on the stack.
- g) With the right child of node $\uparrow 1$, the statement under (3) is executed, which is the next call to `inorder()`. First, however, the address (4) and node's current value, null, are pushed onto the stack. Because node is null, `inorder()` is exited; upon exit, the stack is updated.

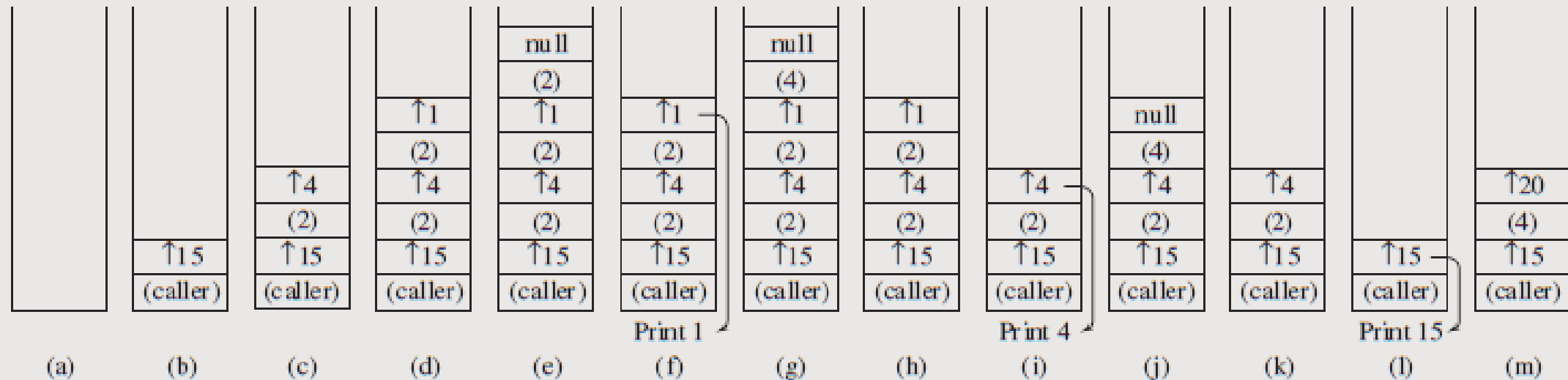


Run Time Stack during Inorder Traversal

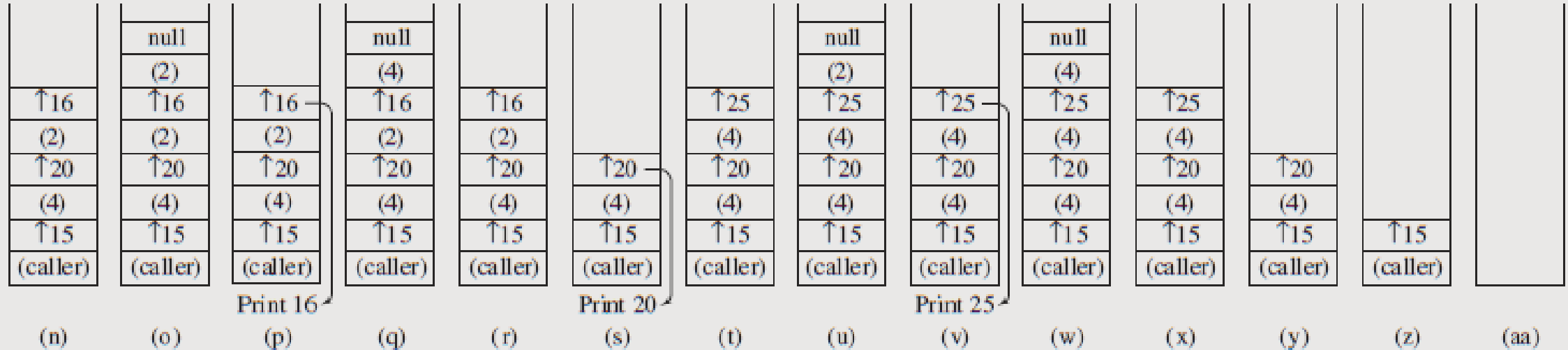
- h) The system now restores the old value of the `node`, $\uparrow 1$, and executes statement (4).
- i) Because this is `inorder()`'s exit, the system removes the current activation record and refers again to the stack; restores the `node`'s value, $\uparrow 4$; and resumes execution from statement (2). This prints the number 4 and then calls `inorder()` for the right child of `node`, which is null.



Run Time Stack during Inorder Traversal



Run Time Stack during Inorder Traversal



Non-Recursive Preorder

```
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;
    }
}
```

Non-Recursive Postorder

```
template<class T>
void BST<T>::iterativePostorder() {
    Stack<BSTNode<T>*> travStack;
    BSTNode<T>* p = root, *q = root;
    while (p != 0) {
        for ( ; p->left != 0; p = p->left)
            travStack.push(p);
        while (p->right == 0 || p->right == q) {
            visit(p);
            q = p;
            if (travStack.empty())
                return;
            p = travStack.pop();
        }
        travStack.push(p);
        p = p->right;
    }
}
```

Iterative Inorder

```
template<class T>
void BST<T>::iterativeInorder() {
    Stack<BSTNode<T>*> travStack;
    BSTNode<T> *p = root;
    while (p != 0) {
        while (p != 0) {           // stack the right child (if any)
            if (p->right)           // and the node itself when going
                travStack.push(p->right); // to the left;
            travStack.push(p);
            p = p->left;
        }
        p = travStack.pop();        // pop a node with no left child
        while (!travStack.empty() && p->right == 0) { // visit it
            visit(p);               // and all nodes with no right
            p = travStack.pop();    // child;
        }
        visit(p);                  // visit also the first node with
        if (!travStack.empty())    // a right child (if any);
            p = travStack.pop();
        else p = 0;
    }
}
```

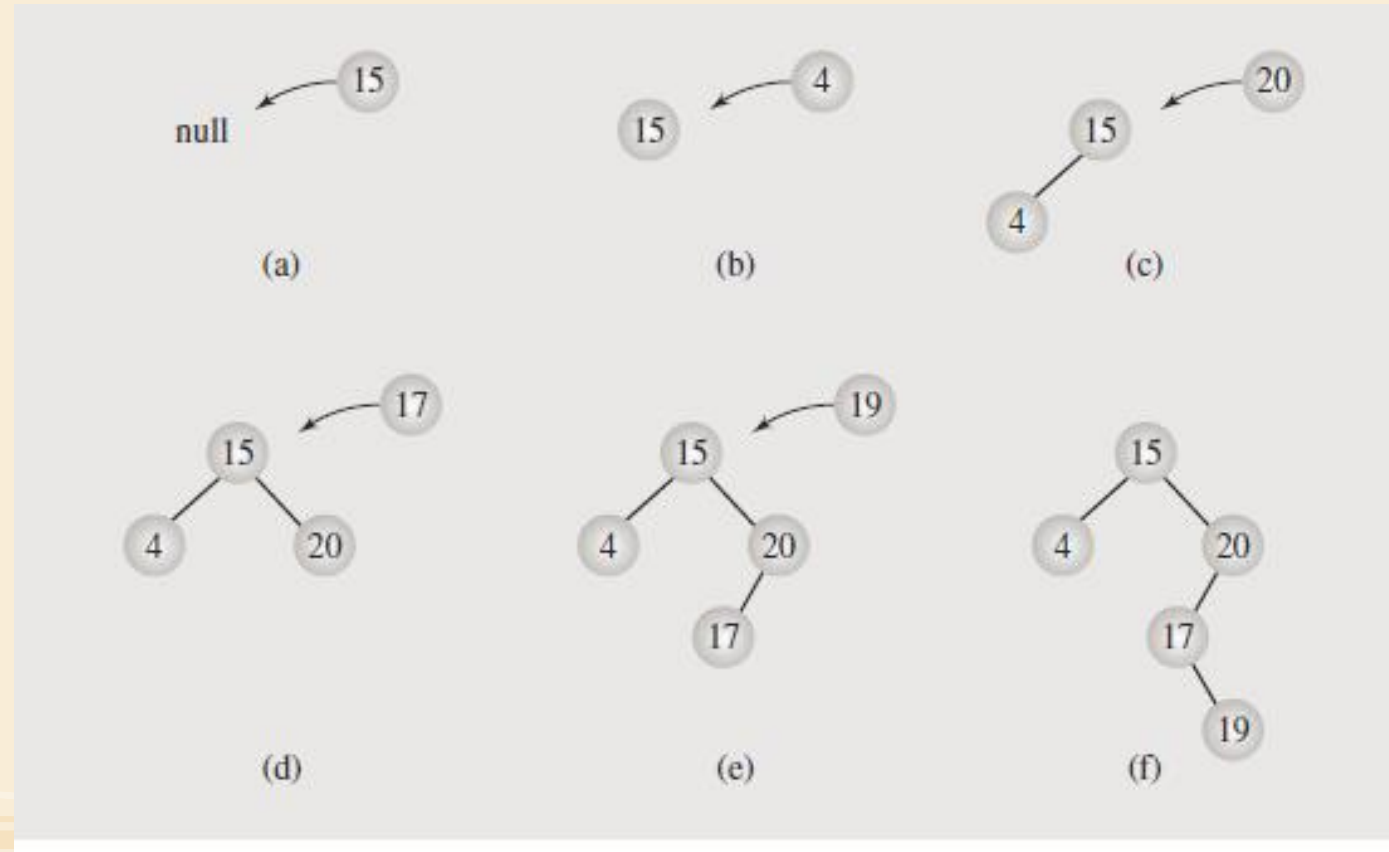
BST Insertion

- To insert a new node with key el , a tree node with a dead end has to be reached, and the new node has to be attached to it.
- Such a tree node is found using the same technique that tree searching used: the key el is compared to the key of a node currently being examined during a tree scan.
- If el is less than that key, the left child (if any) of p is tried; otherwise, the right child (if any) is tested.
- If the child of p to be tested is empty, the scanning is discontinued and the new node becomes this child.

BST Insertion

```
template<class T>
void BST<T>::insert(const T& el)
{
    BSTNode<T> *p=root,*prev = 0;
    while(p != 0)
    {
        prev = p;
        if (p->key < el)
            p=p->right;
        else
            p=p->left;
    }
    if (root == 0)
        root = new BSTNode<T>(el);
    else if (prev->key < el)
        prev->right = new BSTNode<T>(el);
    else prev->left = new BSTNode<T>(el);
}
```

BST Insertion

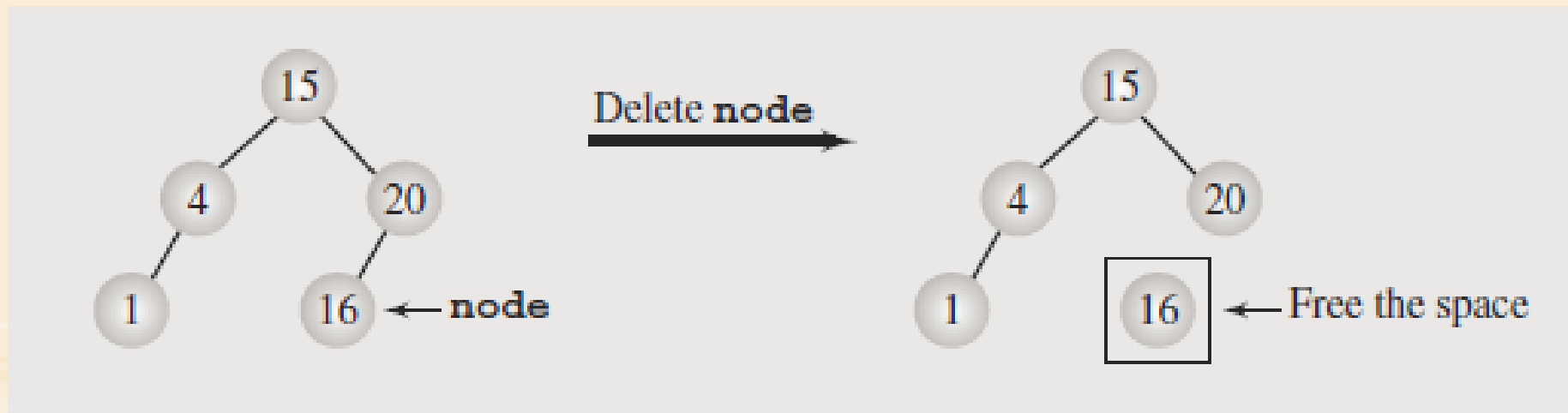


Deleting a Node in BST

- Deleting a node is another operation necessary to maintain a binary search tree.
- The level of complexity in performing the operation depends on the position of the node to be deleted in the tree.
- Three Cases for the node to be deleted
 - Node is a leaf.
 - Node has one child.
 - Node has two children.

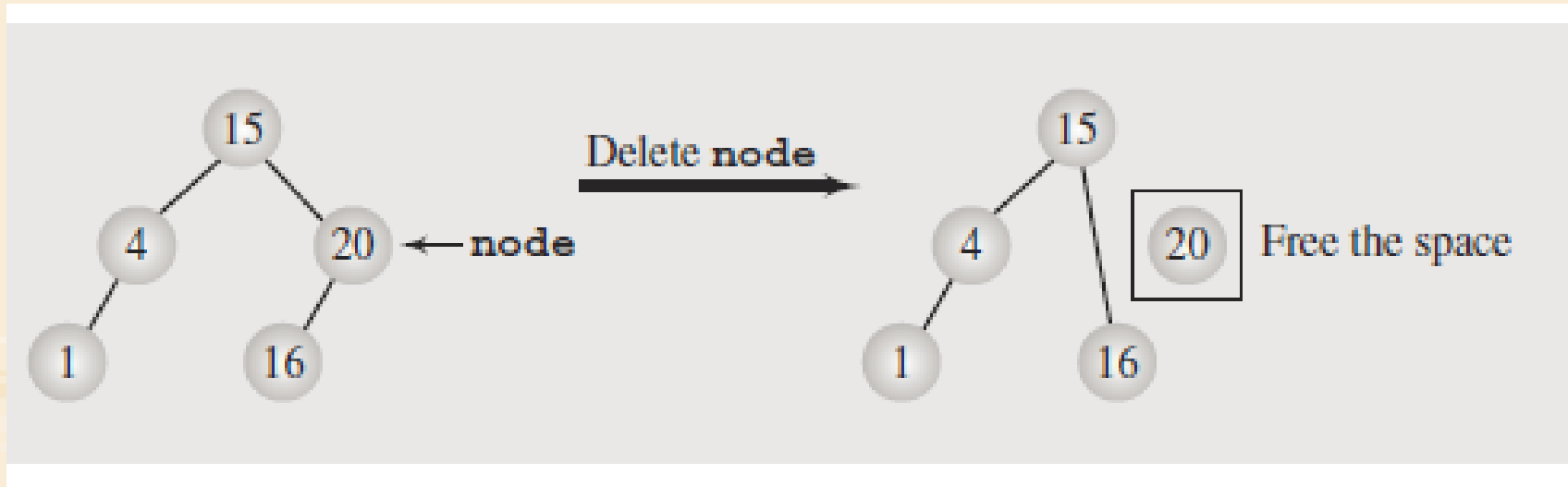
Case 1

- The node is a leaf; it has no children.
- This is the easiest case to deal with.
- The appropriate pointer of its parent is set to null and the node is disposed of by delete as in Figure.



Case 2

- The node has one child.
- The parent's pointer to the node is reset to point to the node's child.
- In this way, the node's children are lifted up by one level and all great-great-. . . grandchildren lose one “great” from their kinship designations.




Case 3

- The node has two children.
- In this case, no one-step operation can be performed because the parent's right or left pointer cannot point to both the node's children at the same time.
- Two Variations
 - **Deletion by Merging**
 - **Deletion by Copying**

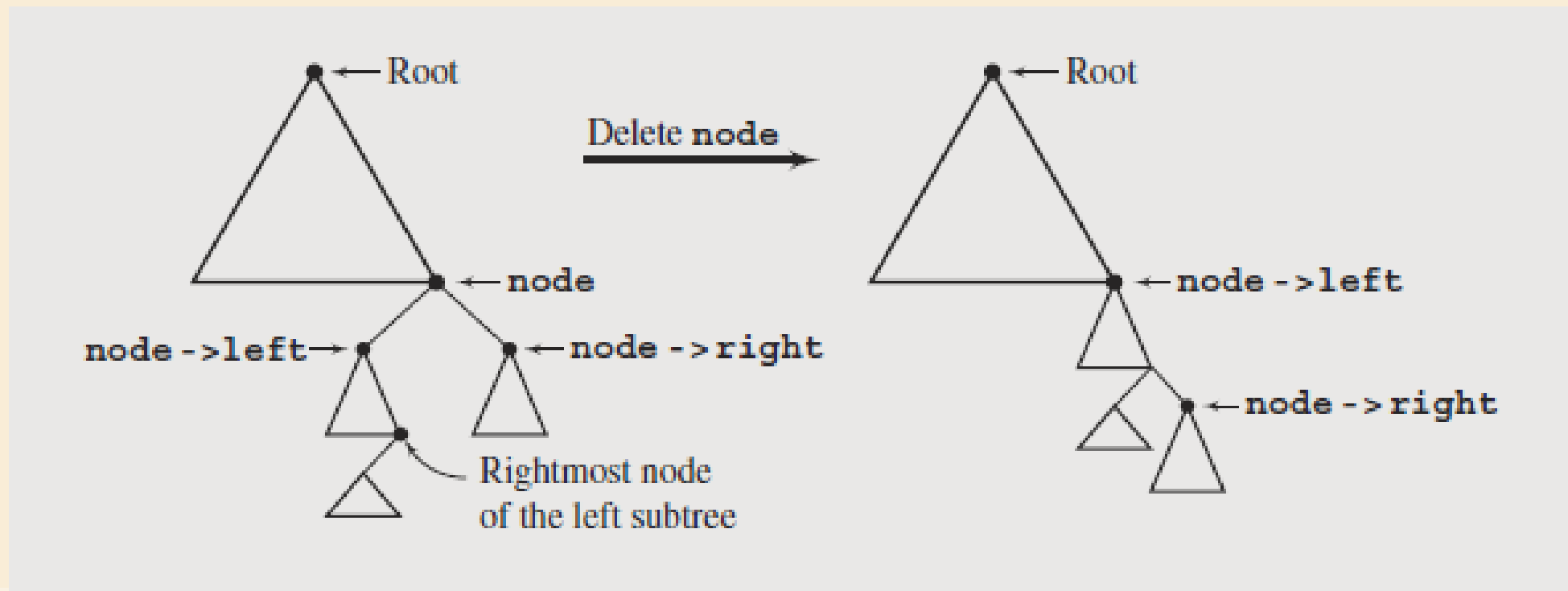
Deletion by Merging

- This solution makes one tree out of the two subtrees of the node and then attaches it to the node's parent. This technique is called *deleting by merging*.
- But how can we merge these subtrees?
- By the nature of binary search trees, every key of the right subtree is greater than every key of the left subtree, so the best thing to do is to find in the left subtree the node with the greatest key and make it a parent of the right subtree.
- Symmetrically, the node with the lowest key can be found in the right subtree and made a parent of the left subtree.

Deletion by Merging

- The desired node is the rightmost node of the left subtree.
 - It can be located by moving along this subtree and taking right pointers until null is encountered.
 - This means that this node will not have a right child, and there is no danger of violating the property of binary search trees in the original tree by setting that rightmost node's right pointer to the right subtree.
 - The same could be done by setting the left pointer of the leftmost node of the right subtree to the left subtree.
- 

Deletion by Merging



```

template<class T>
void BST<T>::findAndDeleteByMerging(const T& el) {
    BSTNode<T> *node = root, *prev = 0;
    while (node != 0) {
        if (node->el == el)
            break;
        prev = node;
        if (el < node->el)
            node = node->left;
        else node = node->right;
    }
    if (node != 0 && node->el == el)
        if (node == root)
            deleteByMerging(root);
        else if (prev->left == node)
            deleteByMerging(prev->left);
        else deleteByMerging(prev->right);
    else if (root != 0)
        cout << "element" << el << "is not in the tree\n";
    else cout << "the tree is empty\n";
}

```

Deletion by Merging

Deletion by Merging

- Instead of calling `search()` before invoking `deleteByMerging()`, `findAndDeleteByMerging()` seems to forget about `search()` and searches for the node to be deleted using its private code.
- `search()` returns a pointer to the node containing el.
- In `findAndDeleteByMerging()`, it is important to have this pointer stored specifically in one of the pointers of the node's parent.
- In other words, a caller to `search()` is satisfied if it can access the node from any direction, whereas `findAndDeleteByMerging()` wants to access it from either its parent's left or right pointer data member.
- Otherwise, access to the entire subtree having this node as its root would be lost.

```

template<class T>
void BST<T>::deleteByMerging(BSTNode<T>*& node) {
    BSTNode<T> *tmp = node;
    if (node != 0) {
        if (!node->right)           // node has no right child: its left
            node = node->left;       // child (if any) is attached to its
                                    // parent;
        else if (node->left == 0)    // node has no left child: its right
            node = node->right;       // child is attached to its parent;
        else {                      // be ready for merging subtrees;
            tmp = node->left;         // 1. move left
            while (tmp->right != 0)   // 2. and then right as far as
                                    // possible;
                tmp = tmp->right;
            tmp->right =              // 3. establish the link between
                node->right;          // the rightmost node of the left
                                    // subtree and the right subtree;
            tmp = node;              // 4.
            node = node->left;        // 5.
        }
        delete tmp;                 // 6.
    }
}

```

Deletion by Copying

- Another solution, called *deletion by copying*, was proposed by Thomas Hibbard and Donald Knuth.
- If the node has two children, the problem can be reduced to one of two simple cases: the node is a leaf or the node has only one nonempty child.
- This can be done by replacing the key being deleted with its immediate predecessor (or successor).
- As already indicated in the algorithm deletion by merging, a key's predecessor is the key in the rightmost node in the left subtree (and analogically, its immediate successor is the key in the leftmost node in the right subtree).

Deletion by Copying

- First, the predecessor has to be located.
- This is done, again, by moving one step to the left by first reaching the root of the node's left subtree and then moving as far to the right as possible.
- Next, the key of the located node replaces the key to be deleted. And that is where one of two simple cases comes into play.
- If the rightmost node is a leaf, the first case applies; however, if it has one child, the second case is relevant.
- In this way, deletion by copying removes a key k_1 by overwriting it by another key k_2 and then removing the node that holds k_2 , whereas deletion by merging consisted of removing a key k_1 along with the node that holds it.

Deletion by Copying

- To implement the algorithm, two functions can be used. One function, `deleteByCopying()`.
- The second function, `findAndDeleteByCopying()`, is just like `findAndDeleteByMerging()`, but it calls `deleteByCopying()` instead of `deleteByMerging()`.

```

template<class T>
void BST<T>::deleteByCopying(BSTNode<T>*& node) {
    BSTNode<T> *previous, *tmp = node;
    if (node->right == 0)                // node has no right child;
        node = node->left;
    else if (node->left == 0)            // node has no left child;
        node = node->right;
    else {
        tmp = node->left;                // node has both children;
        previous = node;                // 1.
        while (tmp->right != 0) {        // 2.
            previous = tmp;
            tmp = tmp->right;
        }
        node->el = tmp->el;                // 3.
        if (previous == node)
            previous ->left = tmp->left;
        else previous ->right = tmp->left; // 4.
    }
    delete tmp;                          // 5.
}

```


Deletion by Copying

- This algorithm does not increase the height of the tree, but it still causes a problem if it is applied many times along with insertion.
- The algorithm is asymmetric; it always deletes the node of the immediate predecessor of the key in node, possibly reducing the height of the left subtree and leaving the right subtree unaffected.
- Therefore, the right subtree of node can grow after later insertions, and if the key in node is again deleted, the height of the right tree remains the same.

Deletion by Copying

- After many insertions and deletions, the entire tree becomes right unbalanced, with the right subtree bushier and larger than the left subtree.
- To circumvent this problem, a simple improvement can make the algorithm symmetrical. The algorithm can alternately delete the predecessor of the key in node from the left subtree and delete its successor from the right subtree.

Counting Leaves

```
int count_leaf()
{
    return count_leaf(root);
}

template<class T>
int BST<T>::count_leaf(BSTNode<T>* p)
{
    if (p == 0)
        return 0;
    else if (p->left == 0 && p->right == 0)
        return 1;
    return count_leaf(p->left) + count_leaf(p->right);
}
```

Counting Non-leaf nodes

```
template<class T>
int BST<T>::count_nonleaf(BSTNode<T>* p)
{
    if (p == 0)
        return 0;
    else if (p->left == 0 && p->right == 0)
        return 0;
    else
        return count_nonleaf(p->left) + count_nonleaf(p->right) + 1;
}
```

Computing Height

```
template<class T>
int BST<T> ::Tree_height(BSTNode<T>* p)
{
    if (p == 0)
        return 0;
    else
        return max(Tree_height(p->left),Tree_height(p->right))+1;
}
```

Creating Mirror Image

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