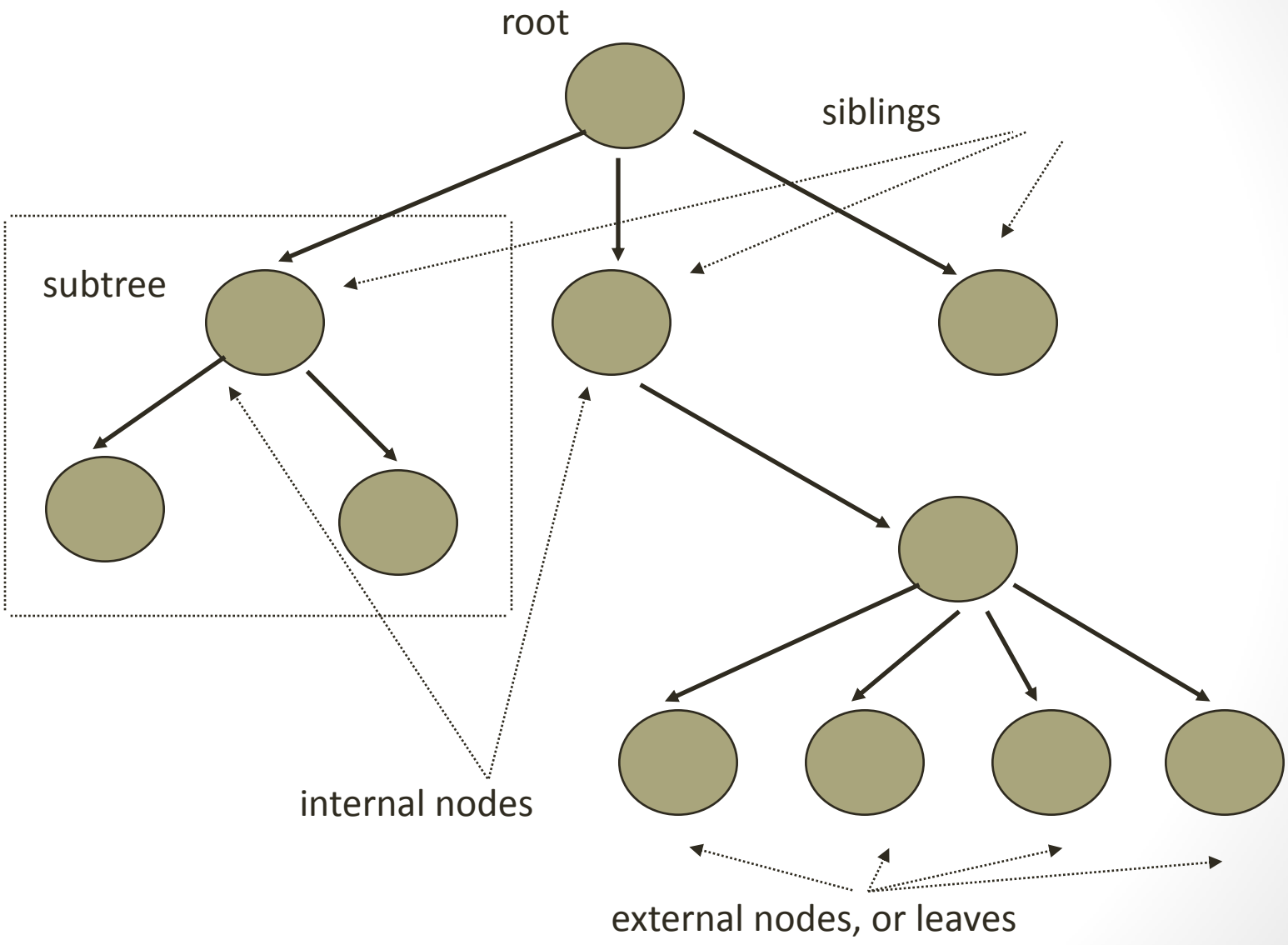
Trees

- Contrary to arrays, stacks, queues and sequences all of which are one-dimensional data structures.
- Tree is a non linear data structure
- Trees are **two-dimensional data structures with hierarchical relationship between data items.**

  One node in the tree is designated as the **root**. Each tree has exactly one path between the root and each of the other nodes. If there is more than one path between the root and some node, or no path at all, we have a **graph**.

Example of a tree:

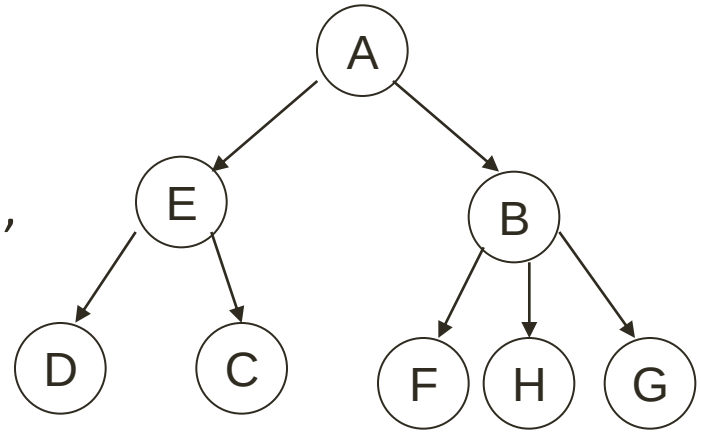


What is a Tree?

- A **tree** is a non-empty collection of vertices (nodes) and edges that satisfy certain requirements.
- Example:

Nodes={A,B,C,D,E,f,G,H}

Edges={(A,B),(A,E),(B,F),(B,G),(B,H),
(E,C),(E,D)}

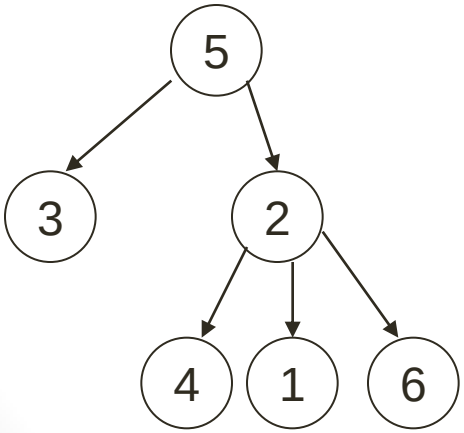


- A directed **path** from node m1 to node m_k is a list of nodes m_1, m_2, . . . , m_k such that each is the parent of the next node in the list. The length of such a path is k – 1 where k is the number of nodes in a path.
- Example: A, E, C is a directed path of length 2.

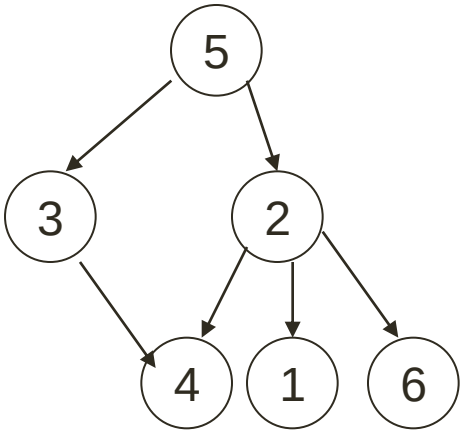
What is a Tree? (contd.)

- A tree satisfies the following properties:

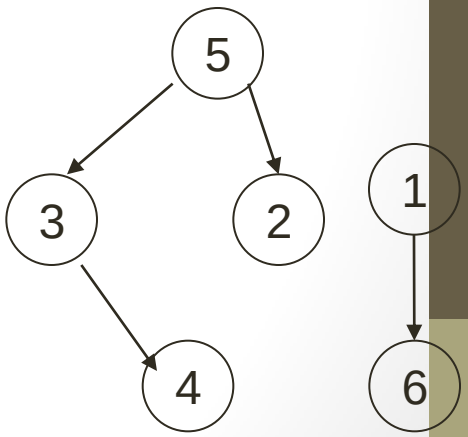
1. It has one designated node, called the root, that has no parent.
2. Every node, except the root, has exactly one parent.
3. A node may have zero or more children.
4. There is a unique directed path from the root to each node.



tree



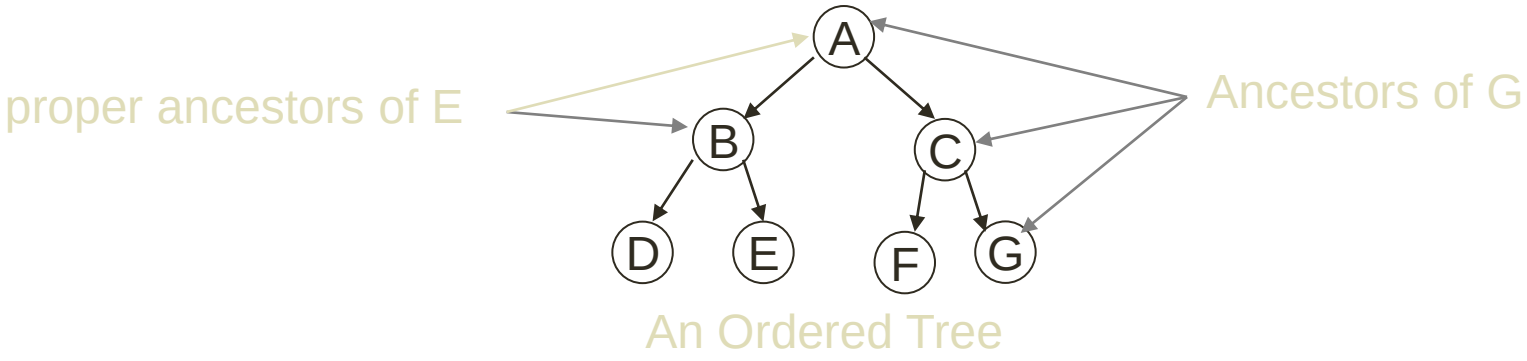
Not a tree



Not a tree

Tree Terminology

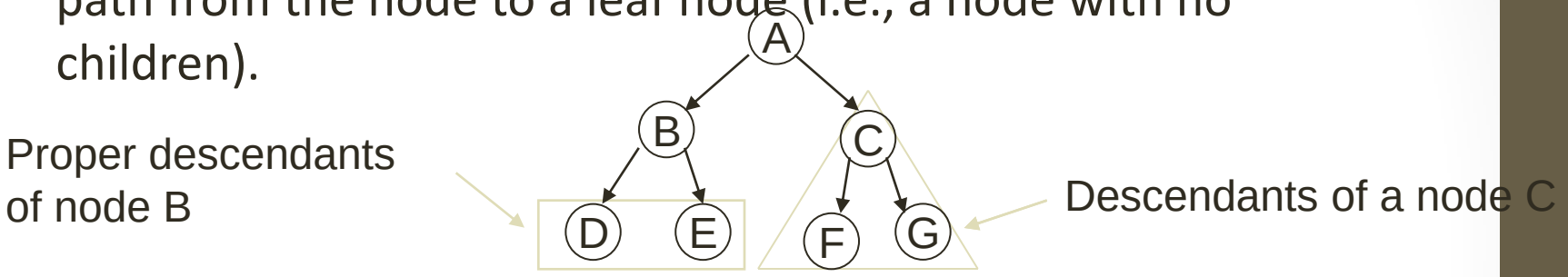
- Ordered tree: A tree in which the children of each node are linearly ordered (usually from left to right).



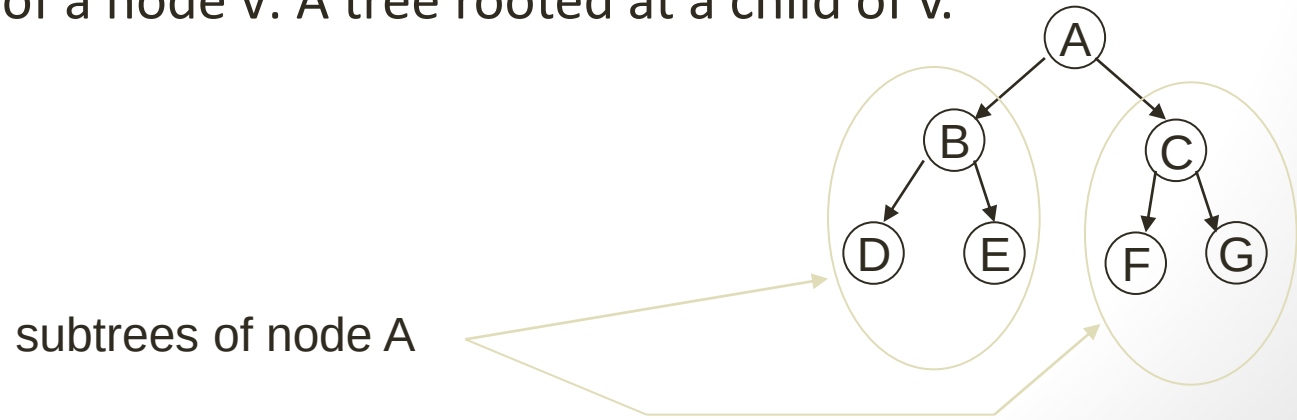
- **Ancestor** of a node v: Any node, including v itself, on the path from the root to the node.
- **Proper ancestor** of a node v: Any node, excluding v, on the path from the root to the node.

Tree Terminology (Contd.)

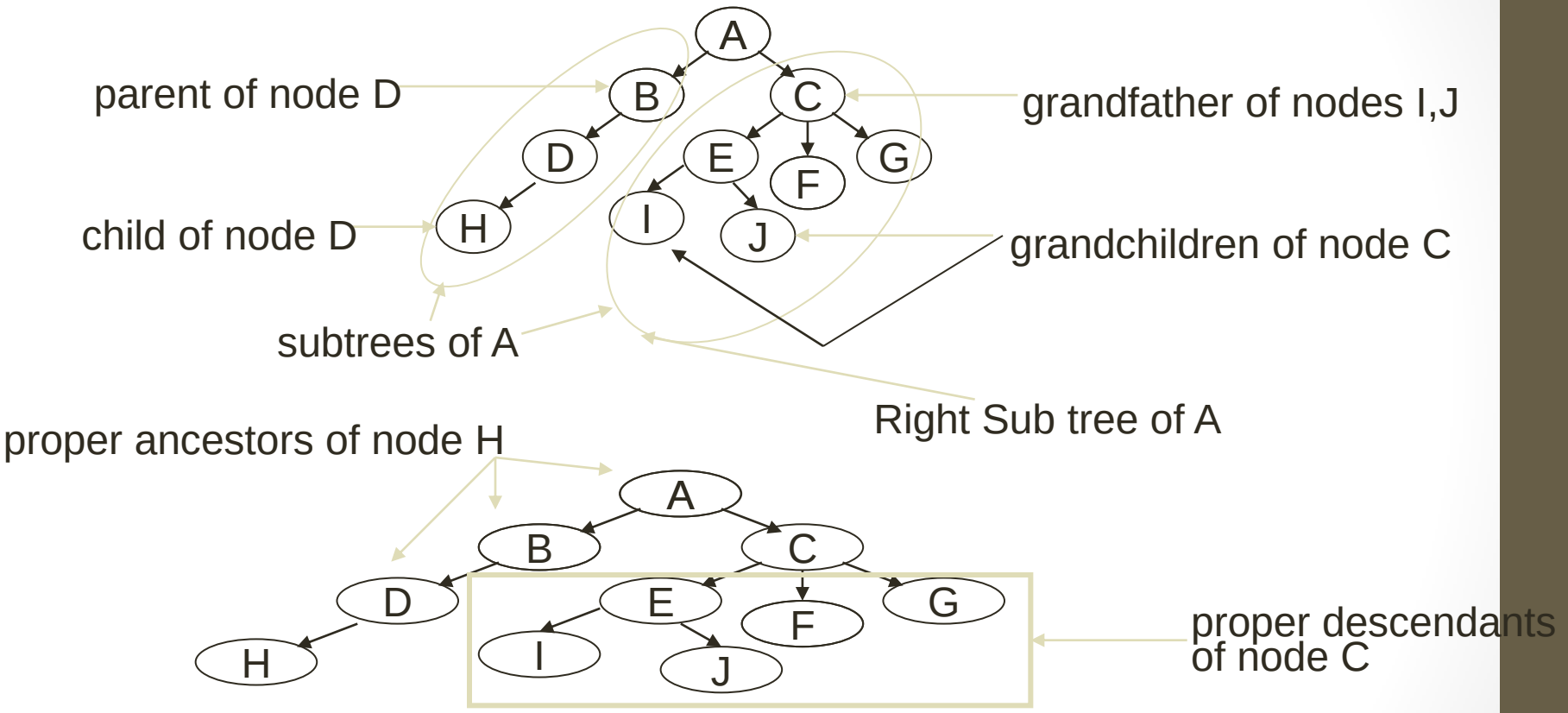
- **Descendant** of a node v: Any node, including v itself, on any path from the node to a leaf node (i.e., a node with no children).



- **Proper descendant** of a node v: Any node, excluding v, on any path from the node to a leaf node.
- **Subtree** of a node v: A tree rooted at a child of v.

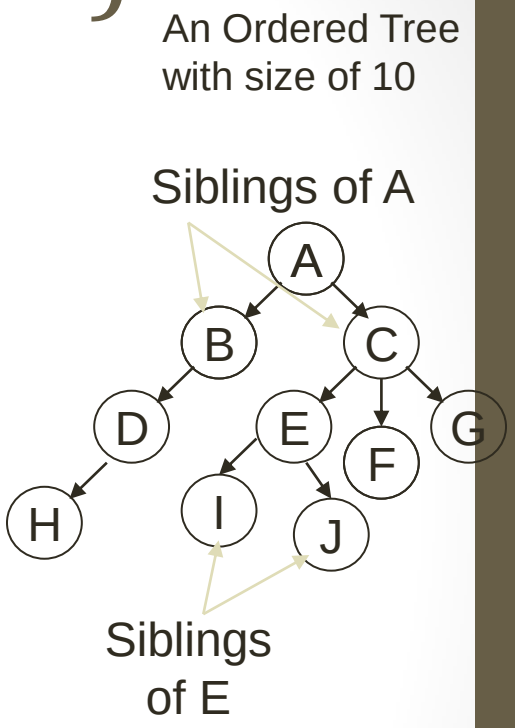


Tree Terminology (Contd.)



Tree Terminology (Contd.)

- Degree: The number of subtrees of a node
  - Each of node D and B has degree 1.
  - Each of node A and E has degree 2.
  - Node C has degree 3.
  - Each of node F,G,H,I,J has degree 0.



- **Leaf**: A node with degree 0.
- **Internal** or interior node: a node with degree greater than 0.
- **Siblings**: Nodes that have the same parent.
- **Size**: The number of nodes in a tree.

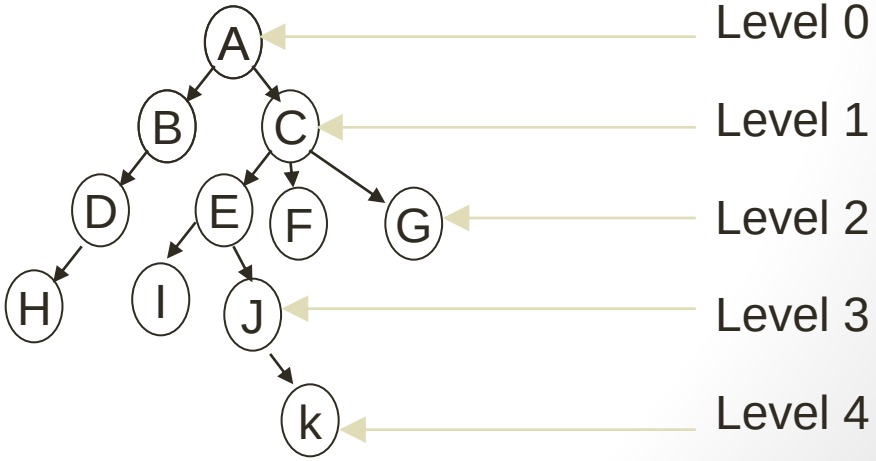
Tree Terminology (Contd.)

- **Level** (or depth) of a node v: The length of the path from the root to v

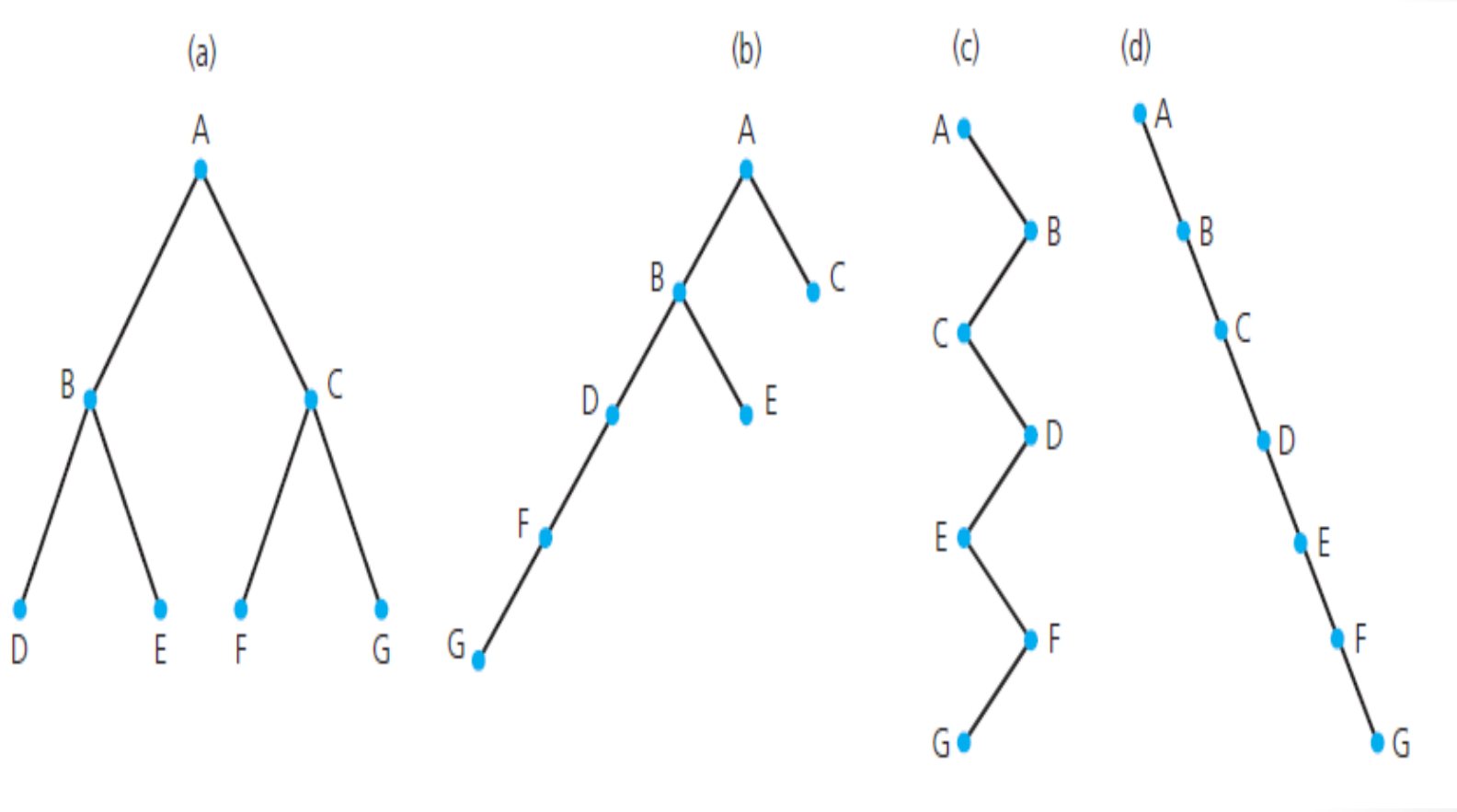
  OR

  It is the number of nodes on the path from that node to the root.
- **Height** of a node v: The length of the longest path from v to a leaf node.
- **Height** of a tree T (**maximum distance**) :It is the maximum level among all of the nodes in the tree.
  - The height of a tree is the max level number.
  - By definition the height of an empty tree is -1.

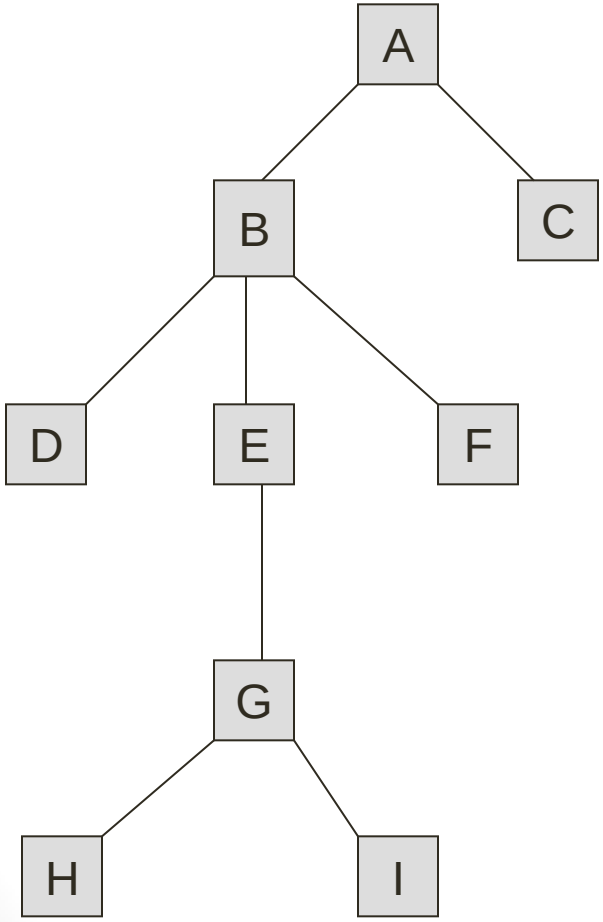
- The height of the tree is 4.
- The height of node C is 3.



Finding Height



Tree Properties



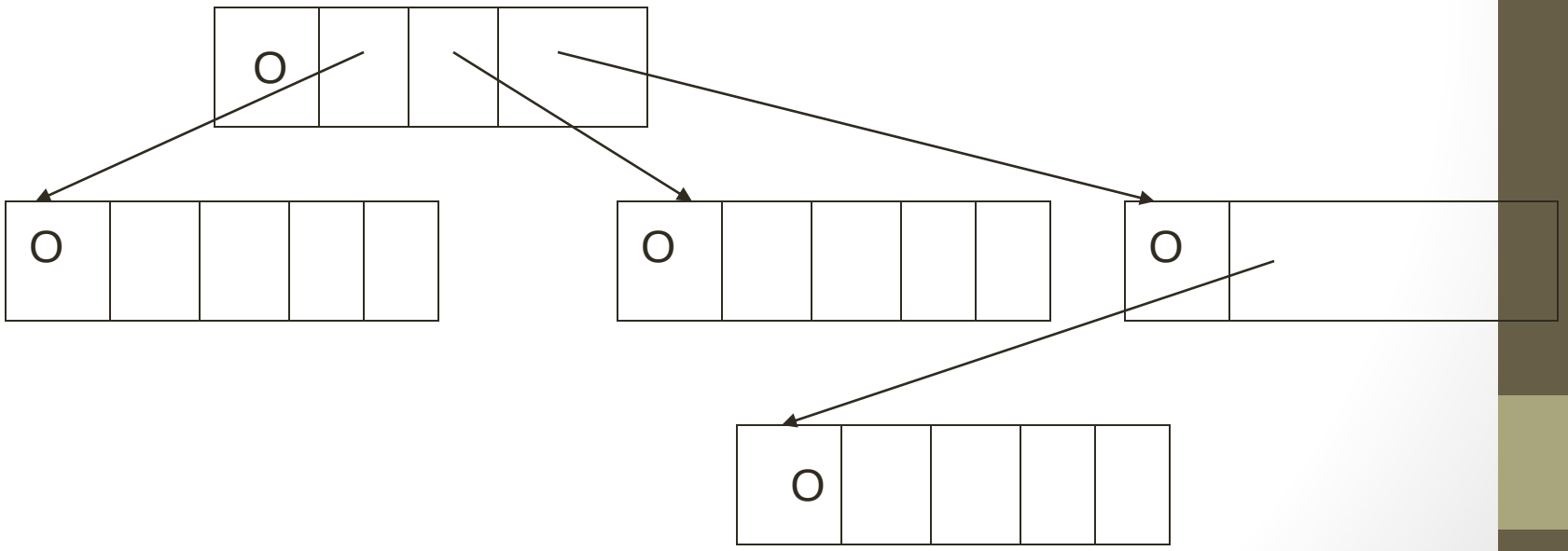
Property	Value
Number of nodes	
Height	
Root Node	
Leaves	
Interior nodes	
Number of levels	
Ancestors of H	
Descendants of B	
Siblings of E	
Right subtree	

Why Trees?

- Trees are very important data structures in computing.
- They are suitable for:
  - Hierarchical structure representation, e.g.,
    - File directory.
    - Organizational structure of an institution.
    - Class inheritance tree.
  - Problem representation, e.g.,
    - Expression tree.
    - Decision tree.
  - Efficient algorithmic solutions, e.g.,
    - Search trees.
    - Efficient priority queues via heaps.

A Tree Node

- Every tree node:
  - object – useful information
  - children – pointers to its children nodes



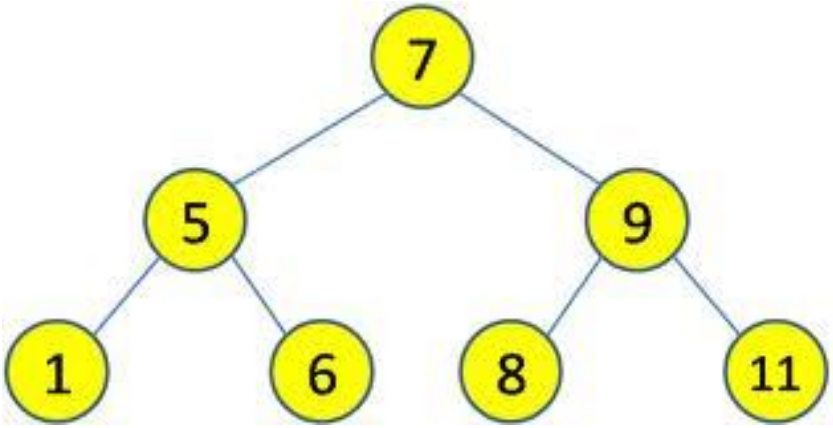
Kinds of Trees

- General Tree or a Rooted tree
  - Set T of one or more nodes such that T is partitioned into disjoint subsets
  - A single node r , the root
  - Sets that are general trees, called subtrees of r

Kinds of Trees

- Ordered Tree

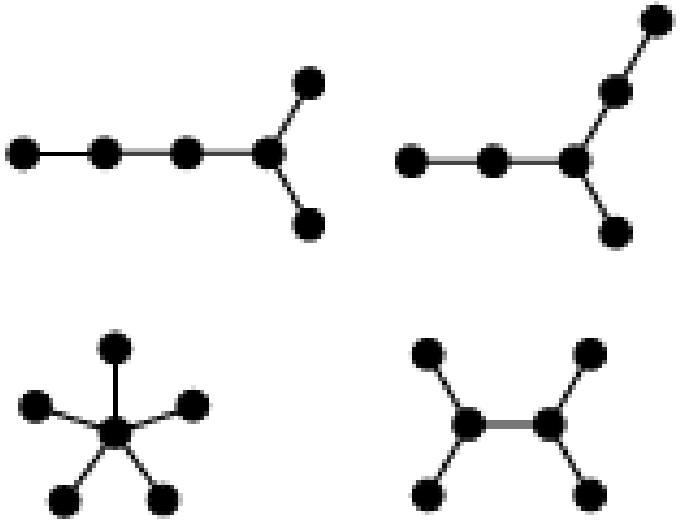
  “It is a rooted tree but in it the order of the children is specified.”



Kinds of Trees

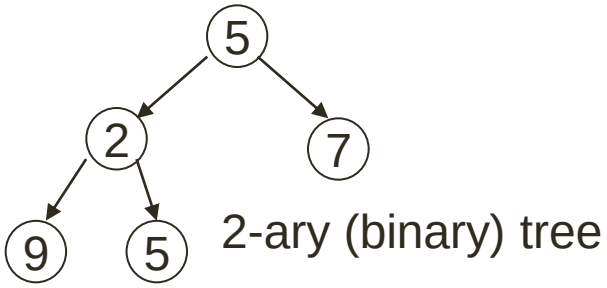
- Free Tree

  "A free tree is a tree that is not rooted"

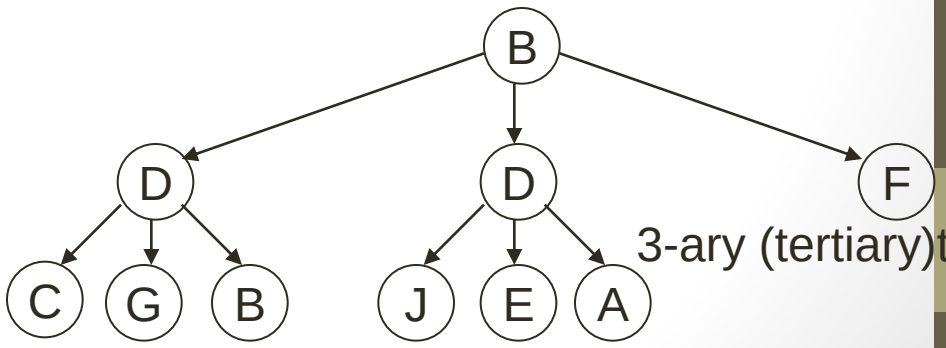


Kinds of Trees

- An N-ary tree is a tree that is either:
  - Empty, or
  - It consists of a root node and at most N non-empty N-ary subtrees.
- It follows that the degree of each node in an N-ary tree is at most N.
- Example of N-ary trees:



2-ary (binary) tree



3-ary (tertiary)tree

Kinds of Trees

- Binary tree

A binary tree is an N-ary tree for which N = 2.

Thus, a binary tree is either:

1. An empty tree, or
2. A tree consisting of a root node and at most two non-empty binary subtrees.

Example: Algebraic Expressions.

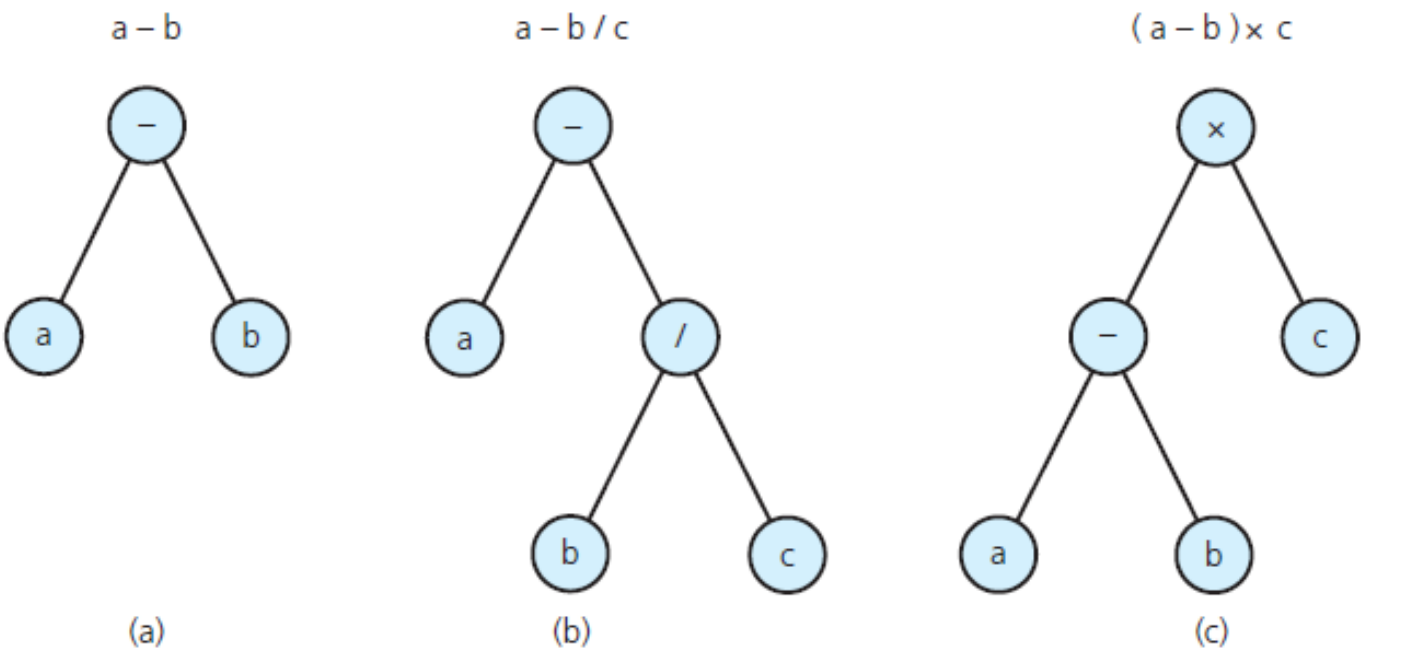


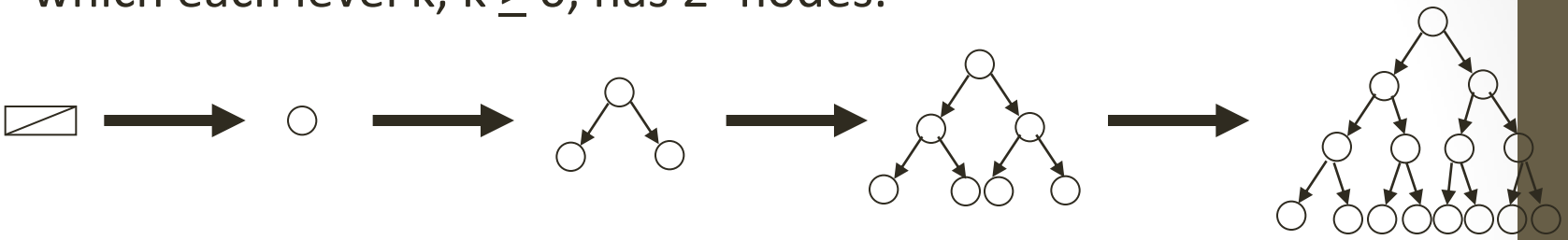
FIGURE 15-3 Binary trees that represent algebraic expressions

Generalization

- Binary or n-ary tree is a special type of ordered tree.
- An ordered tree is a special type of rooted tree.
- A rooted tree is a special type of free tree.

Binary Trees (Contd.)

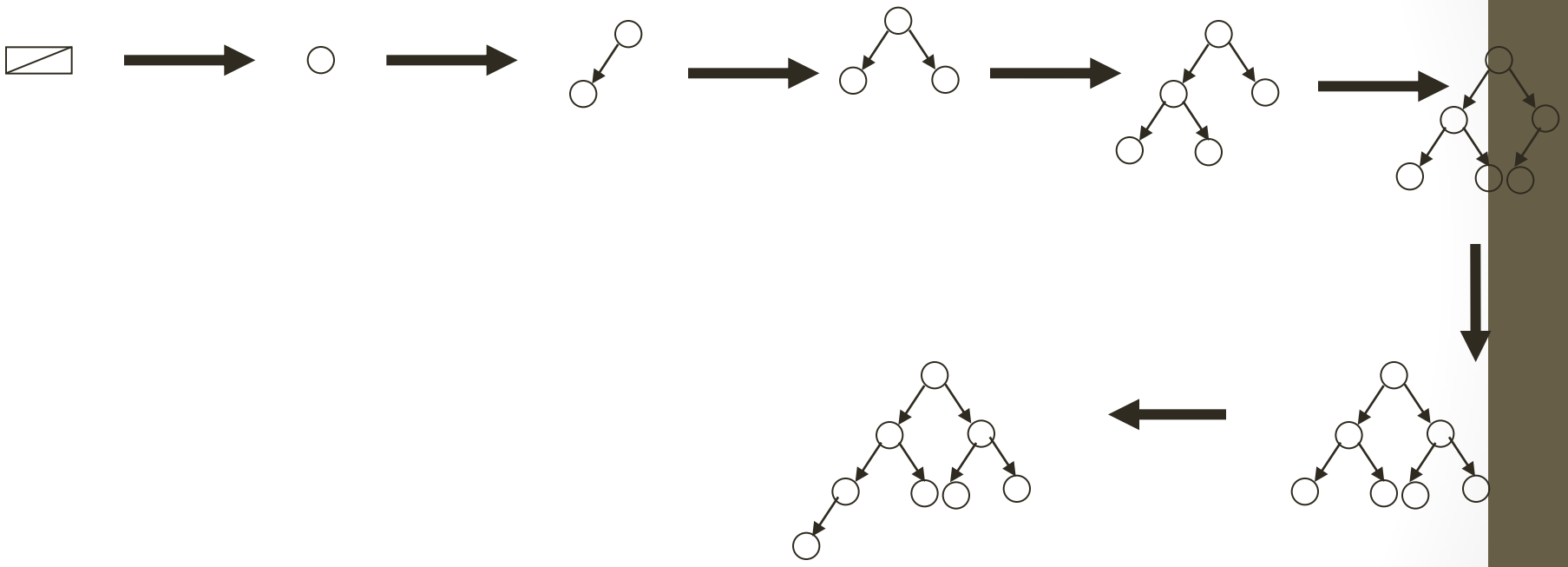
- A **full binary tree** is either an empty binary tree or a binary tree in which each level k, $k \geq 0$, has 2^k nodes.



- A **complete binary tree** is either an empty binary tree or a binary tree in which:
  1. Each level k, $k \geq 0$, other than the last level contains the maximum number of nodes for that level, that is 2^k.
  2. The last level may or may not contain the maximum number of nodes.
  3. If a slot with a missing node is encountered when scanning the last level in a left to right direction, then all remaining slots in the level must be empty.
- Thus, every full binary tree is a complete binary tree, but the opposite is not true.

Binary Trees (Contd.)

- Example showing the growth of a complete binary tree:



Properties of Binary Tree

- The number of external nodes is at least **h + 1**, where **h** is the height of the tree, and at most $\mathbf{2^h}$. The later holds for a **full binary tree**, which is a tree where internal nodes completely fill every level.
- The number of internal nodes is at least **h** and at most $\mathbf{2^h - 1.}$
- The total number of nodes in a binary tree is at least **2*h + 1** and at most $\mathbf{2^{h+1} - 1.}$
- The height, **h**, of a binary tree with **n** nodes is at least **log n** and at most **n -1** .
- A binary tree with **n** nodes has exactly **n - 1** edges.

Properties of a Complete Binary Tree

The following property holds for a complete binary tree.

Let **i** be a number assigned to a node in a complete binary tree. Then:

1. If i = 1, then this node is the root of the tree. If i > 1, then the parent of this node is assigned the number (i / 2).
2. If 2*i > n, then the corresponding node has no left child. Otherwise, the left child of that node is assigned the number 2*i.
3. If 2*i + 1 > n, then the corresponding node has no right child. Otherwise, the right child of that node is assigned the number 2*i + 1.

Operations (methods) on binary trees

empty () - Returns true if the binary tree is empty

getRoot () - Returns the root node of the tree

leftChild (node) - Returns the left child of node.

rightChild (node) -Returns the right child of node.

expandExternal(node) - Makes node internal by creating its left and right children

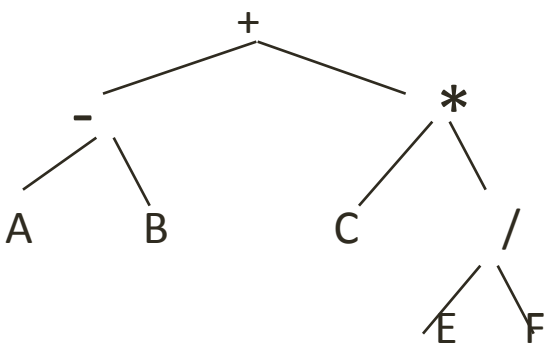
removeAboveExternal(node) - Removes an external node together with its parent

insert (node) - Inserts node in the appropriate position in the tree

delete (node) - Deletes node

Linear (or sequence-based) representation of a binary tree

Linear representation of a binary tree utilizes one-dimensional array of size $2^{h+1} - 1$. Consider the following tree:



level 0 (d = 0)

level 1 (d = 1)

level 2 (d = 2)

level 3 (d = 3)

To represent this tree, we need an array of size $2^{3+1} - 1 = 15$

The tree is represented as follows:

1. The root is stored in BinaryTree[1].
2. For node BinaryTree[n], the left child is stored in BinaryTree[2*n], and the right child is stored in BinaryTree[2*n+1]

i:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
BinaryTree[i]:	+	-	*	A	B	C	/							E	F

Linear representation of a binary tree (cont.)

Advantages of linear representation:

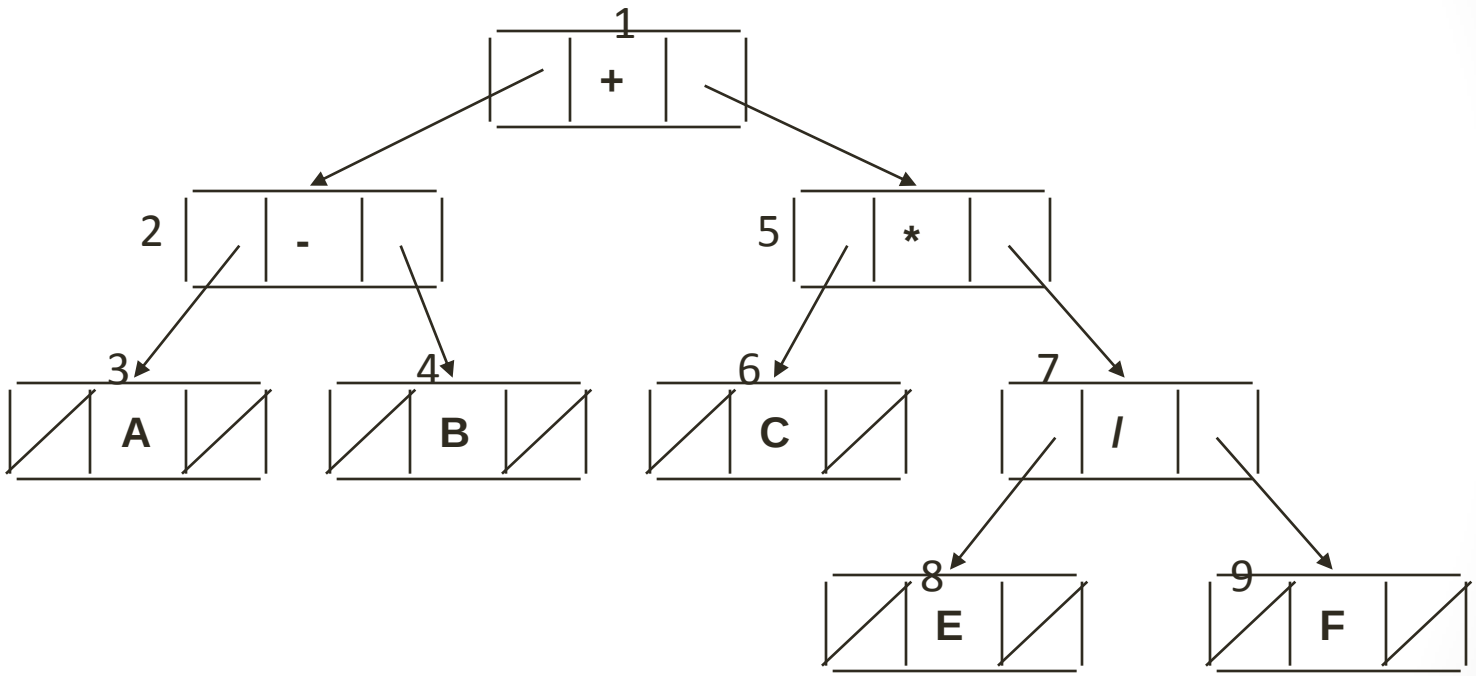
1. Simplicity.
2. Given the location of the child (say, k), the location of the parent is easy to determine (k / 2).

Disadvantages of linear representation:

1. Additions and deletions of nodes are inefficient, because of the data movements in the array.
2. Space is wasted if the binary tree is not complete. That is, the linear representation is useful if the number of missing nodes is small.

Linked representation of a binary tree

Linked representation uses explicit links to connect the nodes. Example:



Nodes in this tree can be viewed as positions in a sequence (numbered 1 through 9).

Binary tree nodes (linked representation)

```
class BTNode {
  char data;
  BTNode leftChild;
  BTNode rightChild;
  BTNode parent;
  int pos;
  public BTNode () { }
  public BTNode (char newData) {
     data = newData;                    }
  public BTNode (char newData, BTNode newLeftChild, BTNode
   newRightChild)
  {   data = newData;
     leftChild = newLeftChild;
     rightChild = newRightChild; }
  ... methods setData, setLeftChild, setRightChild, getData, getLeftChild,
   getRightChild,
     displayBTNode follow next ...
```

Binary tree (linked representation)

We can use a positional sequence ADT to implement a binary tree. Our example tree, in this case, we be represented as follows:

position	1	2	3	4	5	6	7	8	9
data	+	-	A	B	*	C	/	E	F
leftChild	2	3	null	null	6	null	8	null	null
rightChild	5	4	null	null	7	null	9	null	null
parent	null	1	2	2	1	5	5	7	7

Tree Traversal

preOrder() - Visit the root, then the left subtree, then the right subtree

postOrder () - Visit the left subtree, then the right subtree, then the root

inOrder() - Visit the left subtree, then the root, then the right subtree

levelOrder () - Starting from the root, visit tree nodes level by level

Pre Order Traversal

```
public void preOrder (BTNode localRoot)
{
        if (localRoot != null)
          {
           localRoot.displayBTNode();
           preOrder(localRoot.leftChild);
           preOrder(localRoot.rightChild);
          }
}
```

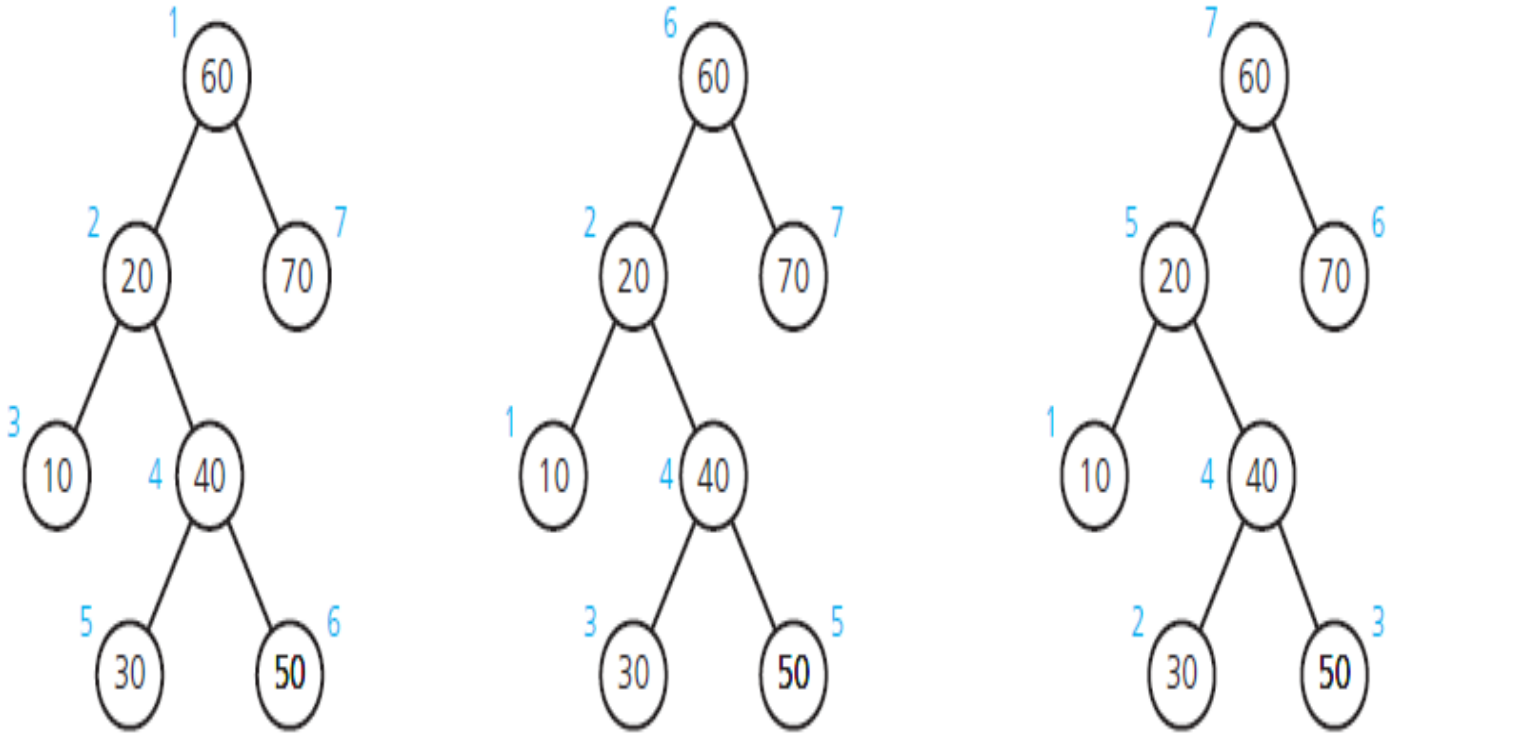
Post Order Traversal

```
public void preOrder (BTNode localRoot)
{
        if (localRoot != null)
          {
           preOrder(localRoot.leftChild);
           preOrder(localRoot.rightChild);
           localRoot.displayBTNode();
          }
}
```

In Order Traversal

```
public void preOrder (BTNode localRoot)
{
        if (localRoot != null)
          {
           preOrder(localRoot.leftChild);
           localRoot.displayBTNode();
           preOrder(localRoot.rightChild);
          }
}
```

Traversing



(a) Preorder: 60, 20, 10, 40, 30, 50, 70 (b) Inorder: 10, 20, 30, 40, 50, 60, 70 (c) Postorder: 10, 30, 50, 40, 20, 70, 60

(Numbers beside nodes indicate traversal order.)

Level-order Traversal

- Implementation: not recursive.
- Use a queue: which initially contains only the root.
- Repeat, dequeue a node from the list
  , visit it
  ,enqueue its children (left to right)
  Until queue is empty

Level Order Traversal

```
public void levelOrder (BTNode localRoot) {
    BTNode[] queue = new BTNode[20];
    int front = 0; int rear = -1;
    while (localRoot != null) {
       localRoot.displayBTNode();
       if (localRoot.leftChild != null) {
          rear++;
          queue[rear] = localRoot.leftChild; }
       if (localRoot.rightChild != null) {
          rear++;
          queue[rear] = localRoot.rightChild;  }
       localRoot = queue[front];
       front++;  }   }
```

Binary Tree Operations

- Test whether a binary tree is empty.
- Get the height of a binary tree.
- Get the number of nodes in a binary tree.
- Get the data in a binary tree's root.
- Set the data in a binary tree's root.
- Add a new node containing a given data item to a binary tree.

Binary Tree Operations

- Remove the node containing a given data item from a binary tree.
- Remove all nodes from a binary tree.
- Retrieve a specific entry in a binary tree.
- Test whether a binary tree contains a specific entry.
- Traverse the nodes in a binary tree in preorder, inorder, or postorder.

Binary Search Tree

- For each node n, a binary search tree satisfies the following three properties:
  - n ’s value is greater than all values in its left subtree T_L.
  - n ’s value is less than all values in its right subtree T_R.
  - Both T_L and T_R are binary search trees.

Binary Search Tree

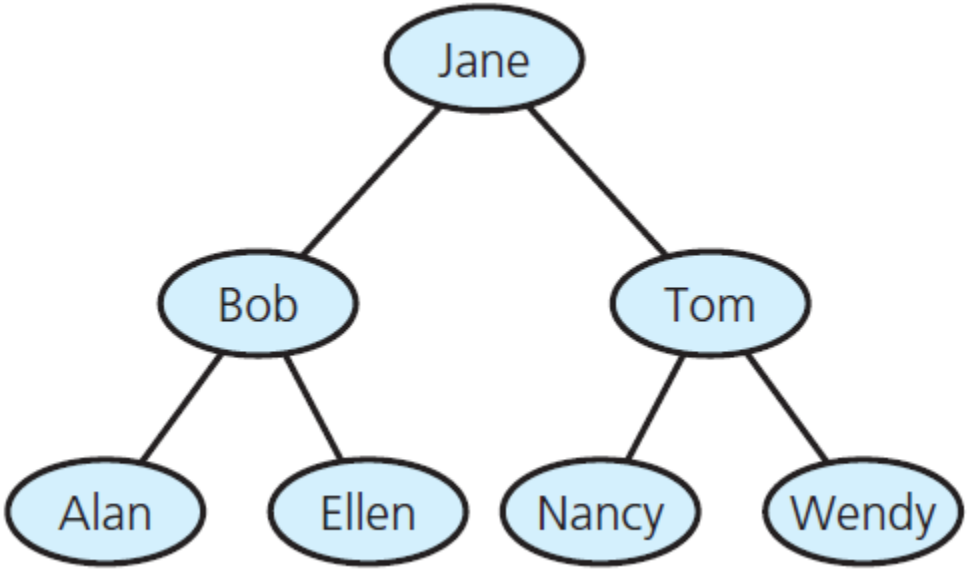


FIGURE 15-4 A binary search tree of names

The ADT Binary Search Tree

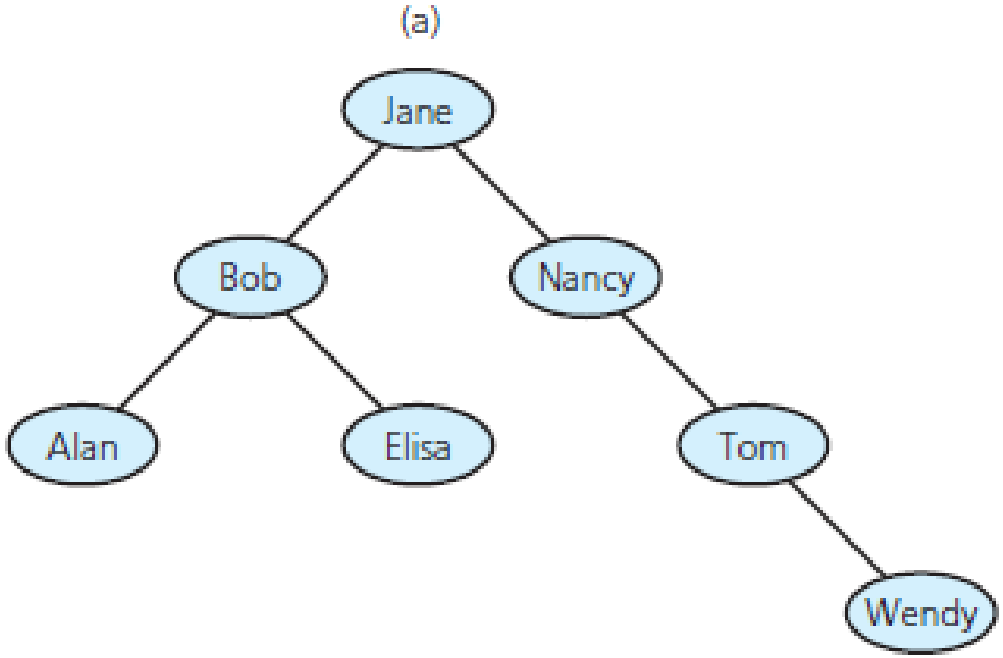


FIGURE 15-14 Binary search trees with the same data as in Figure 15-13

The ADT Binary Search Tree

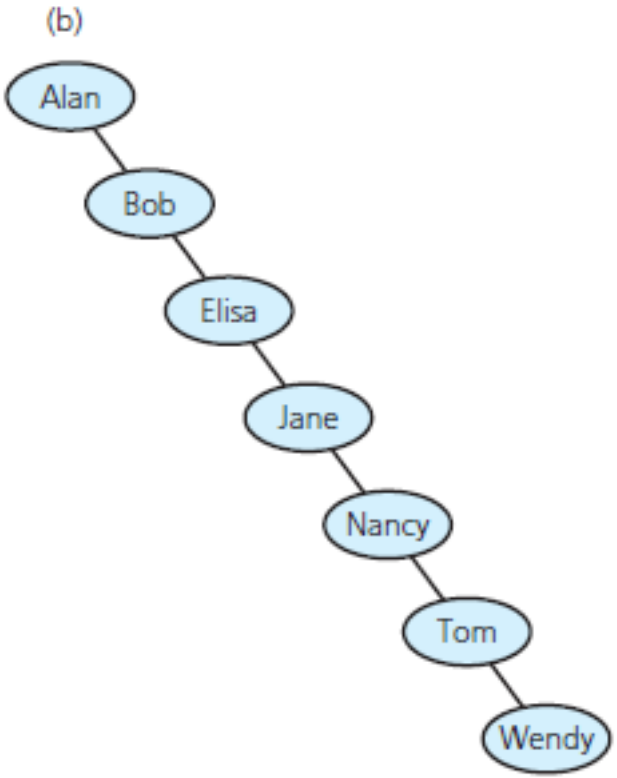


FIGURE 15-14 Binary search trees with the same data as in Figure 15-13

The ADT Binary Search Tree

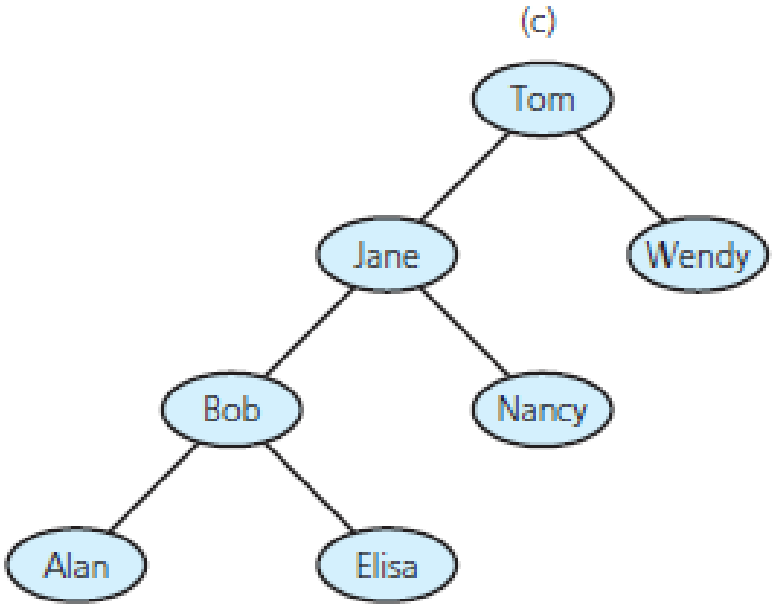


FIGURE 15-14 Binary search trees with the same data as in Figure 15-13

Binary Search Tree Operations

- Test whether binary search tree is empty.
- Get height of binary search tree.
- Get number of nodes in binary search tree.
- Get data in binary search tree's root.
- Insert new item into binary search tree.
- Remove given item from binary search tree.

Binary Search Tree Operations

- Remove all entries from binary search tree.
- Retrieve given item from binary search tree.
- Test whether binary search tree contains specific entry.
- Traverse items in binary search tree in
  - Preorder
  - Inorder
  - Postorder.

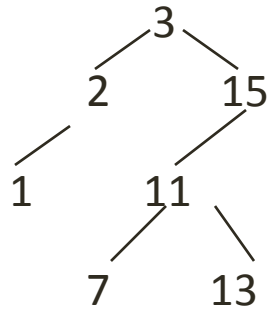
Searching a Binary Search Tree

- Search algorithm for binary search tree

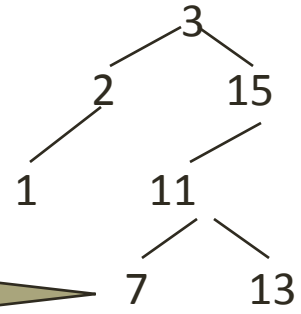
```
// Searches the binary search tree for a given target value.
search(bstTree: BinarySearchTree, target: ItemType)

    if (bstTree is empty)
        The desired item is not found
    else if (target == data item in the root of bstTree)
        The desired item is found
    else if (target < data item in the root of bstTree)
        search(Left subtree of bstTree, target)
    else
        search(Right subtree of bstTree, target)
```

Insertion in binary search tree

Insert **9** in the the following tree:

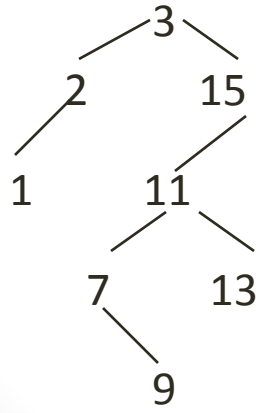


Step 1: search for **9**



search stops here

Step 2: insert **9** at the point where the search terminates unsuccessfully



That is, new nodes are always inserted at the leaf level.

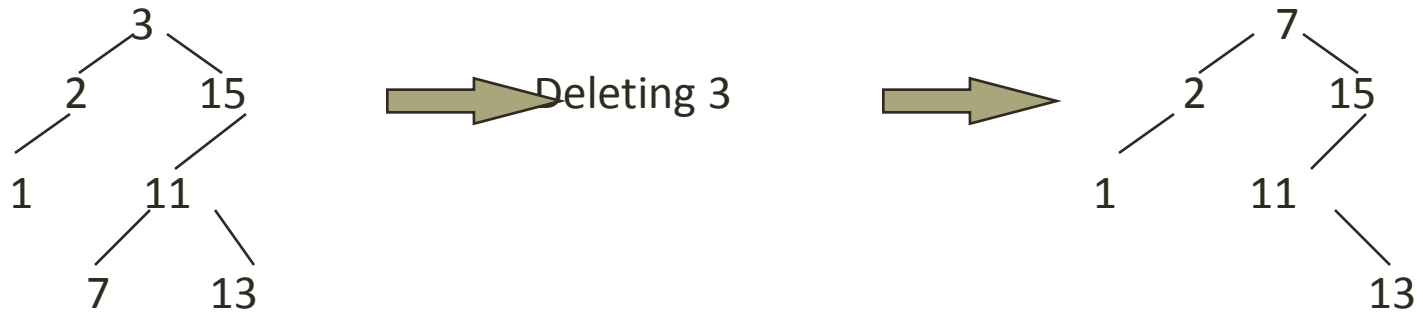
Binary Tree with an ordering property: the insert method

```
class BTLRADT {

  BTNode root;

  public BTLRADT () { }

  public BTNode getRoot () {
    return root; }

  public void insert (char newData) {
    BTNode newNode = new BTNode
   ();
    newNode.data = newData;
    if (root == null)
       root = newNode;
    else {
       BTNode temp = root;
       BTNode parent;
```

```
while (true) {
          parent  = temp;
          if (newData < temp.data)  {   //go
    left
            temp = temp.leftChild;
            if (temp == null) {
              parent.leftChild = newNode;
              return;    }
          }
else {                    // go right
            temp = temp.rightChild;
            if (temp == null) {
              parent.rightChild = newNode;
              return;
            }
          }
        }
      }
  }
```

Deletion in binary search tree

Consider the tree:



The following cases of deletions are possible:

1. Delete a note with no children, for example 1. This only requires the appropriate link in the parent node to be made null.
2. Delete a node which has only one child, for example 15. In this case, we must set the corresponding child link of the parent's parent to point to the only child of the node being deleted.
3. Delete a node with two children, for example 3. The delete method is based on the following consideration: in-order traversal of the resulting tree (after delete operation) must yield an ordered list. To ensure this, the following steps are carried out:

   Step 1: Replace **3** with the node with the next largest datum, i.e. **7**.

   Step 2: Make the left link of **11** point to the right child of **7** (which is null here).

   Step 3: Copy the links from the node containing **3** to the node containing **7**, and make the parent node of **3** point to **7**.

Efficiency of Binary Search Tree Operations

Operation	Average case	Worst case
Retrieval	O(log *n*)	O(*n*)
Insertion	O(log *n*)	O(*n*)
Removal	O(log *n*)	O(*n*)
Traversal	O(*n*)	O(*n*)