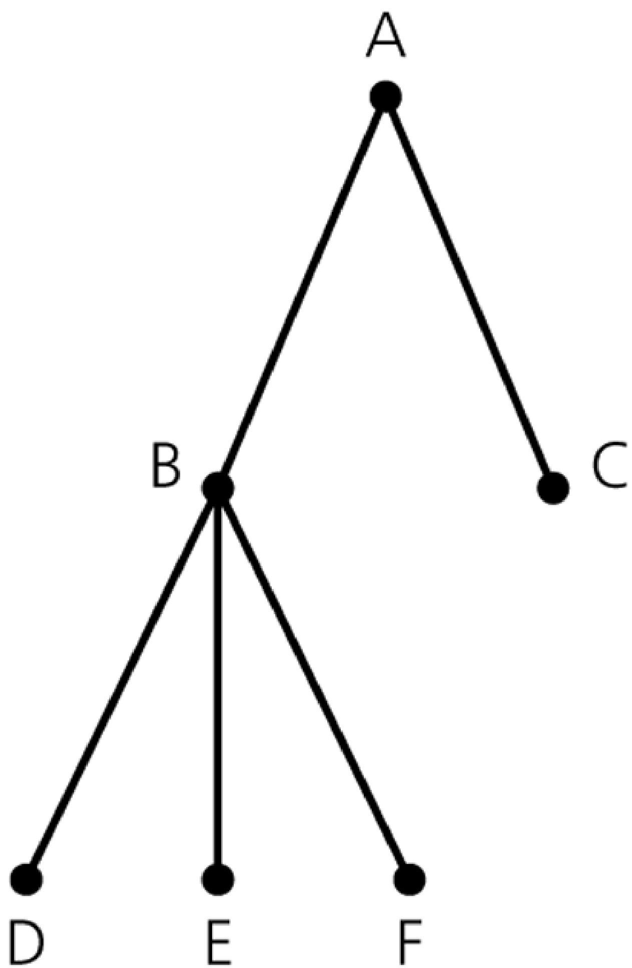
Ch. 11 Trees

사실을 많이 아는 것 보다는
이론적 틀이 중요하고,
기억력보다는
생각하는 법이 더 중요하다.

– 제임스 왓슨

A general tree

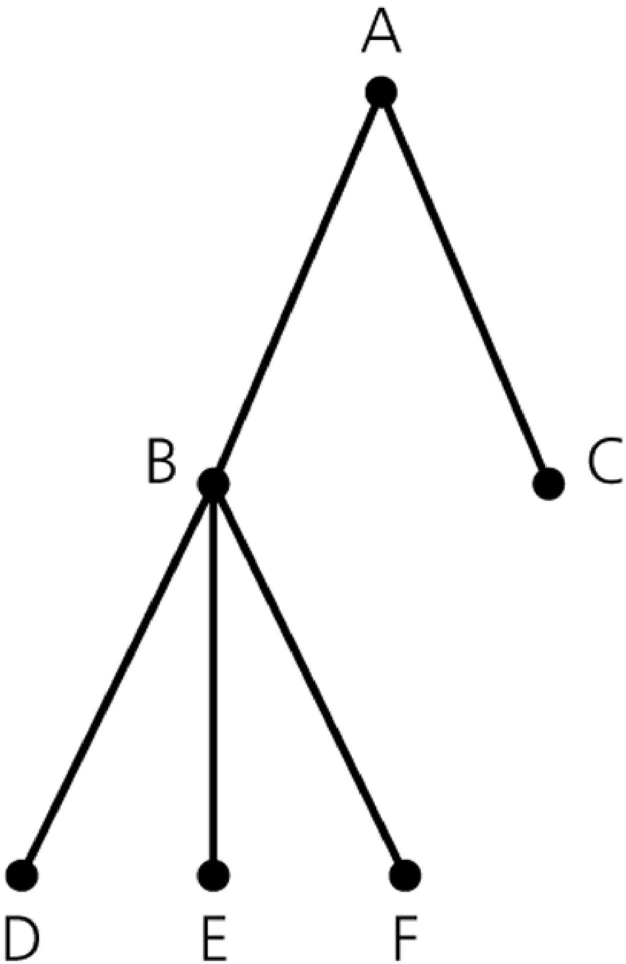


Terminology

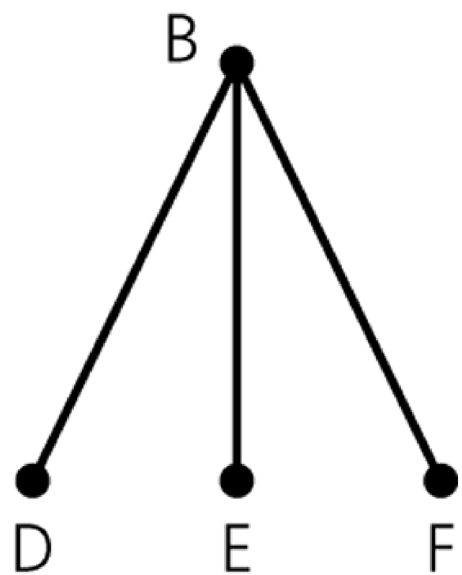
node or vertex
edge
parent
child
siblings
root
leaf
ancestor
descendant
subtree

Definition of Tree

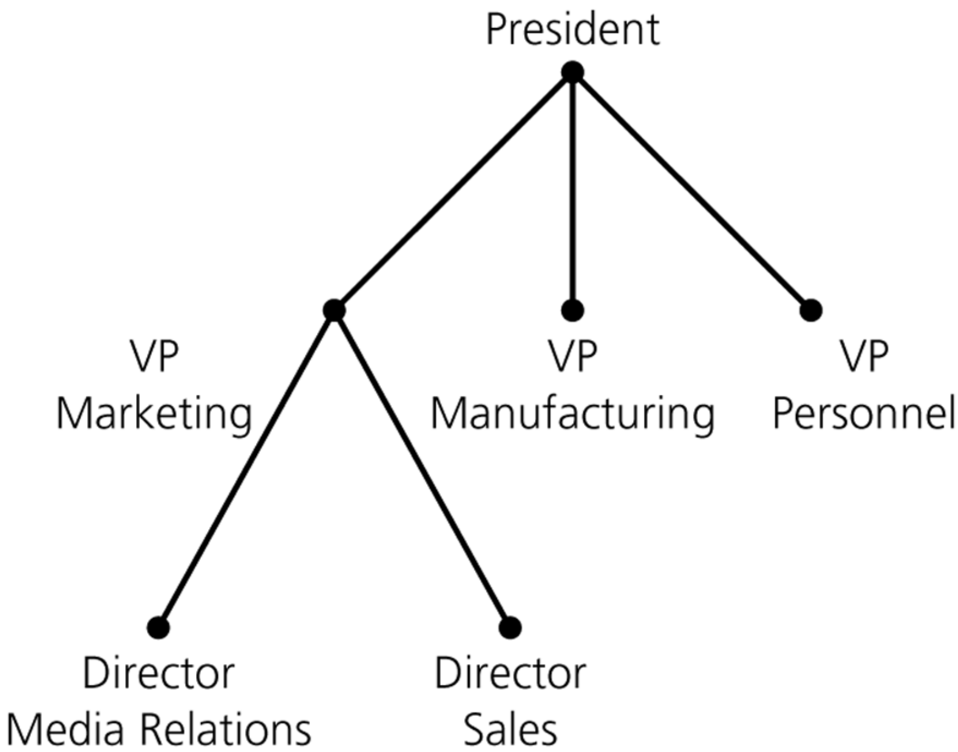
- General tree *T* is partitioned into disjoint subsets:
  - Empty or
  - Root node + sets of general trees



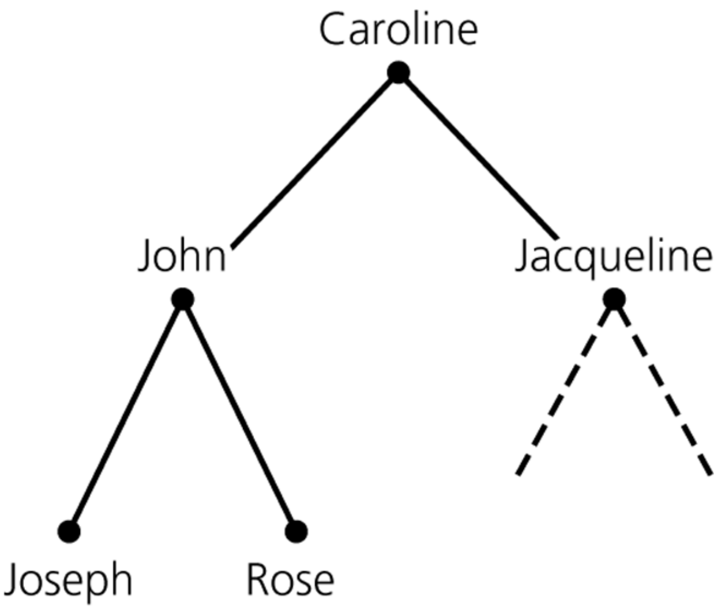
A subtree



An organization chart

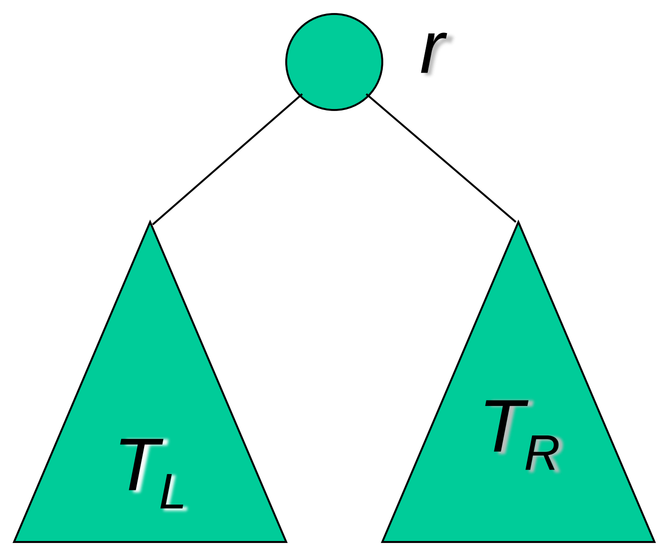


A family tree

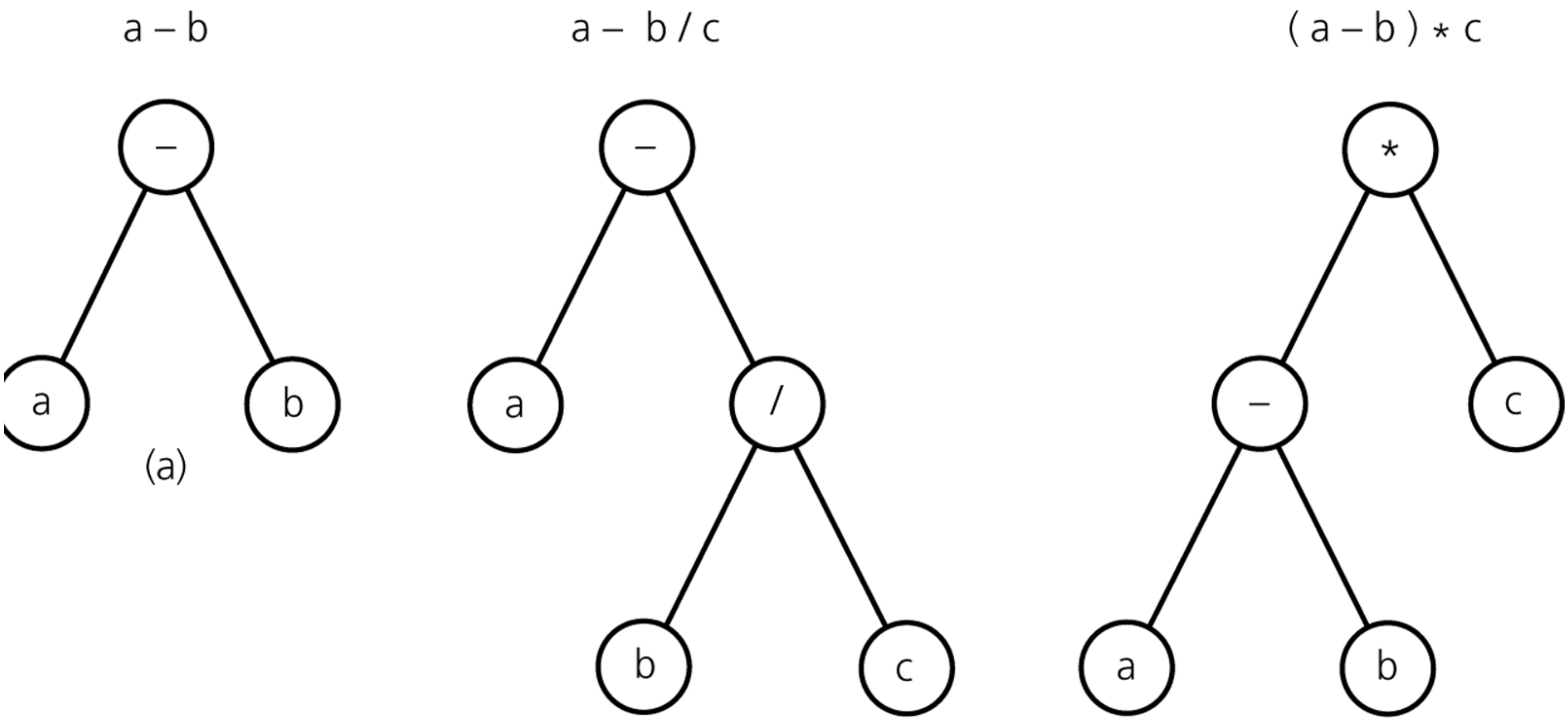


Binary Tree

- T is empty, or
- T is partitioned into three disjoint subsets:
  - A single node *r*, the root
  - Two sets of binary trees, called **left** and **right** **binary (sub)trees** of *r*



Binary Trees for Algebraic Expressions

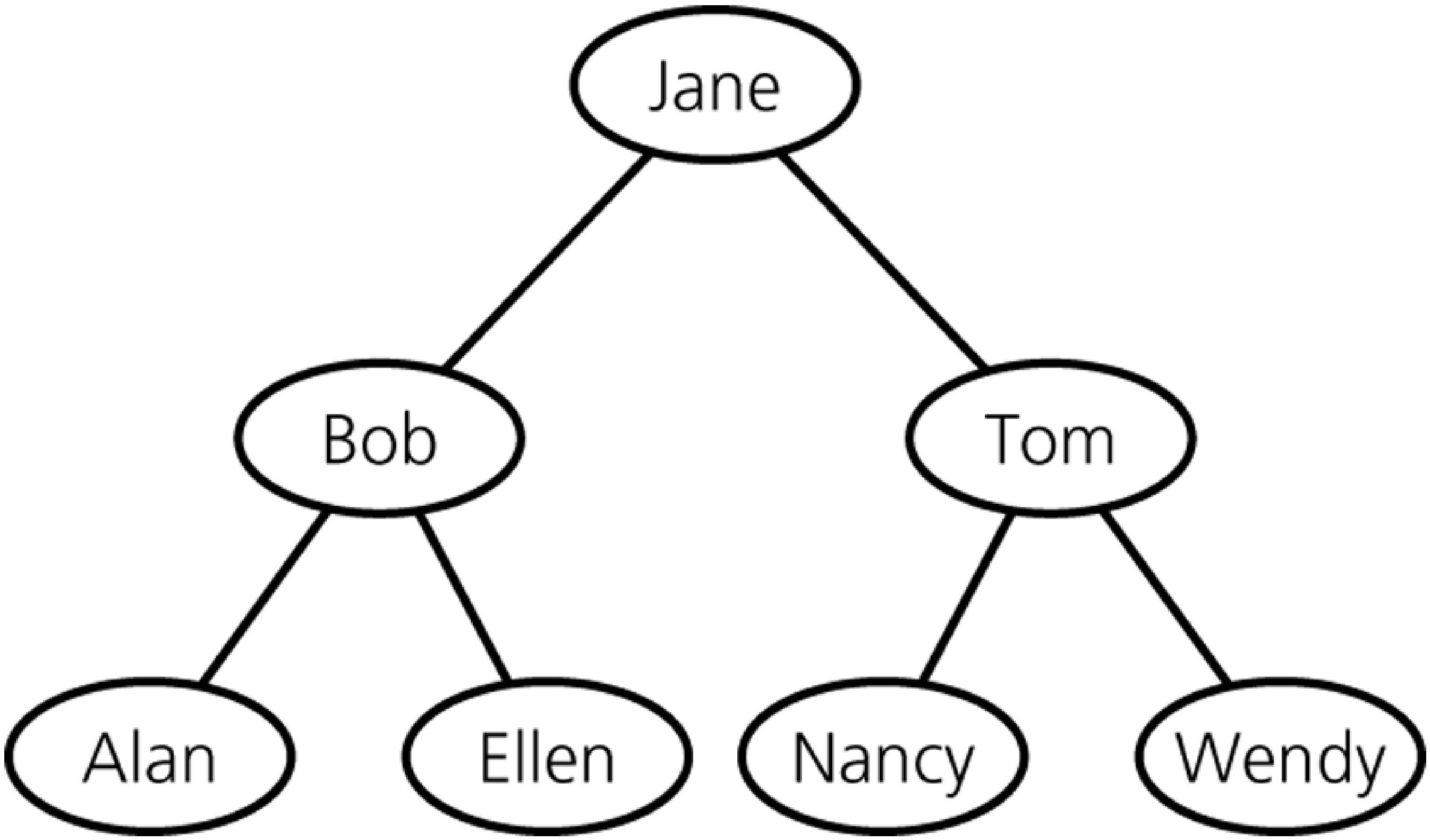


(a) (b) (c)

Binary Search Tree

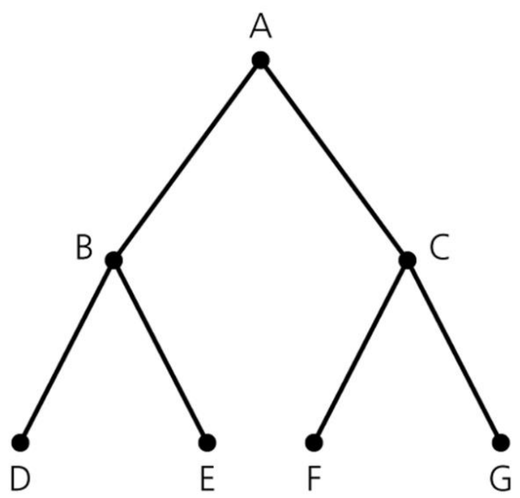
- A binary tree that is in a sense sorted according to the values in its nodes
- For each node ***n***, it satisfies:
  - ***n***'s value is greater than all values in its left subtree T_L
  - ***n***'s value is less than all values in its right subtree T_R
  - Both T_L and T_R are binary search trees

A Binary Search Tree of Names

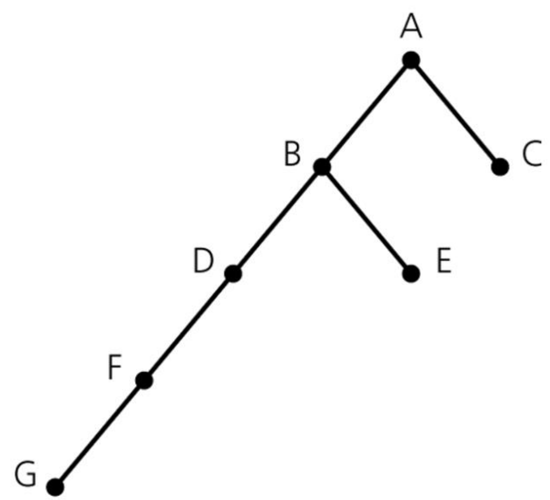


Height of a Tree

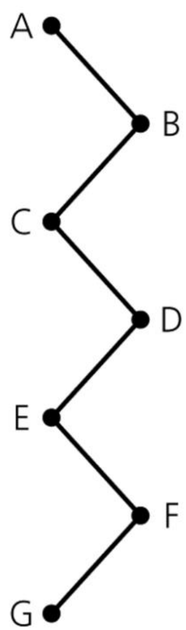
- The number of nodes on the longest path from the root to a leaf



Height 3



Height 5



Height 7

Full Binary Tree

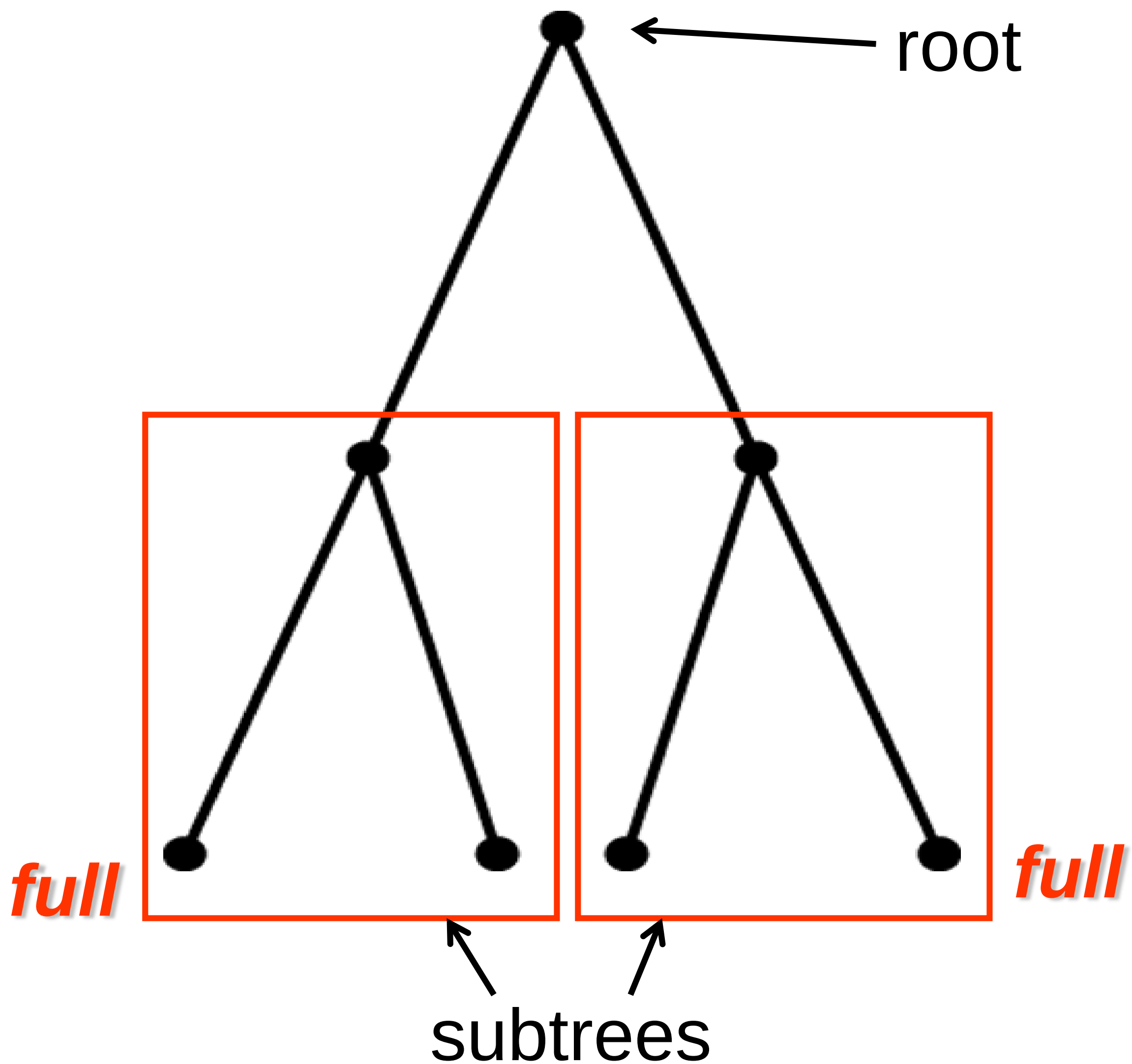
- If *T* is empty,

  T is a full binary tree of height 0
- If *T* is not empty and has height *h*,

  T is a full binary tree

  if the root's subtrees are both full binary trees of height h–1

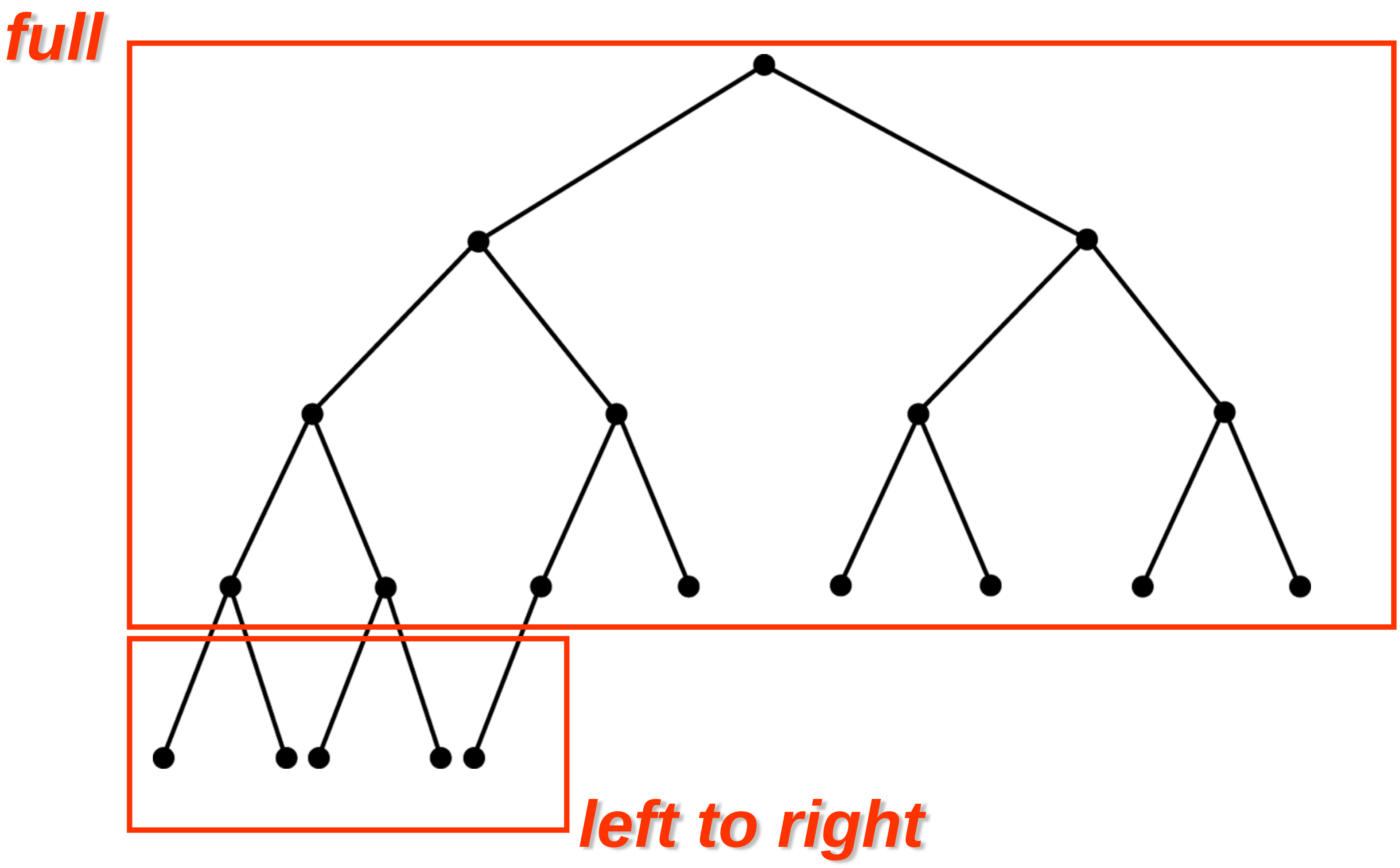
A Full Binary Tree of Height 3



Complete Binary Tree

- A complete binary tree of height h is
  a binary tree
  that is full down to level h–1
  with level h filled in from left to right

A Complete Binary Tree

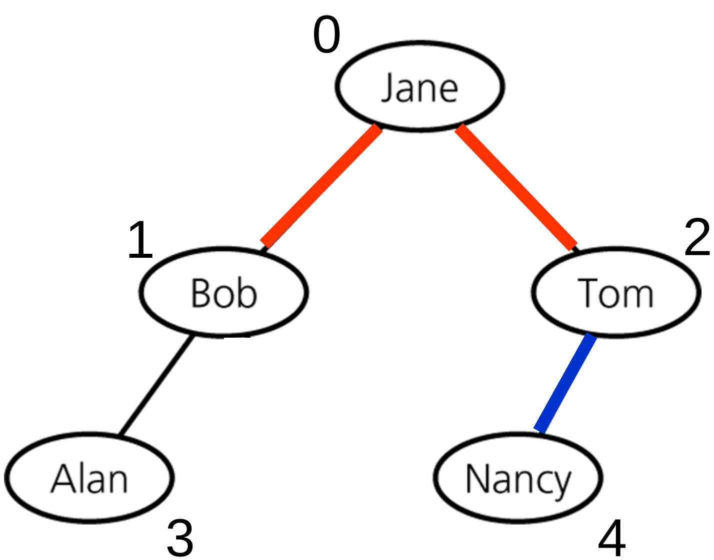


ADT *Binary Tree* Operations

- Create an empty binary tree
- Create a one-node binary tree
- Remove all nodes from a binary tree
- Determine whether a binary tree is empty
- Determine what data is the binary tree's root

Incomplete!

Array-Based Representation



tree

	item	leftChild	rightChild
0	Jane	1	2
1	Bob	3	-1
2	Tom	4	–1
3	Alan	–1	–1
4	Nancy	–1	–1
5	?	–1	–1
6	?	–1	7
7	?	–1	8
8	?	–1	9
⋮	⋮	⋮	⋮

root

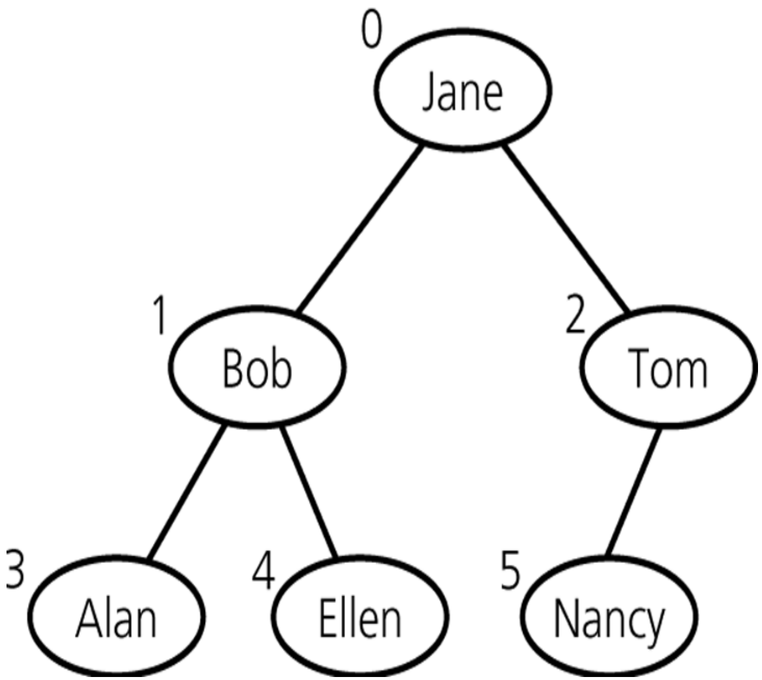
0

free

6

Free list

In Case of Complete Binary Trees



0	Jane
1	Bob
2	Tom
3	Alan
4	Ellen
5	Nancy
6	
7	

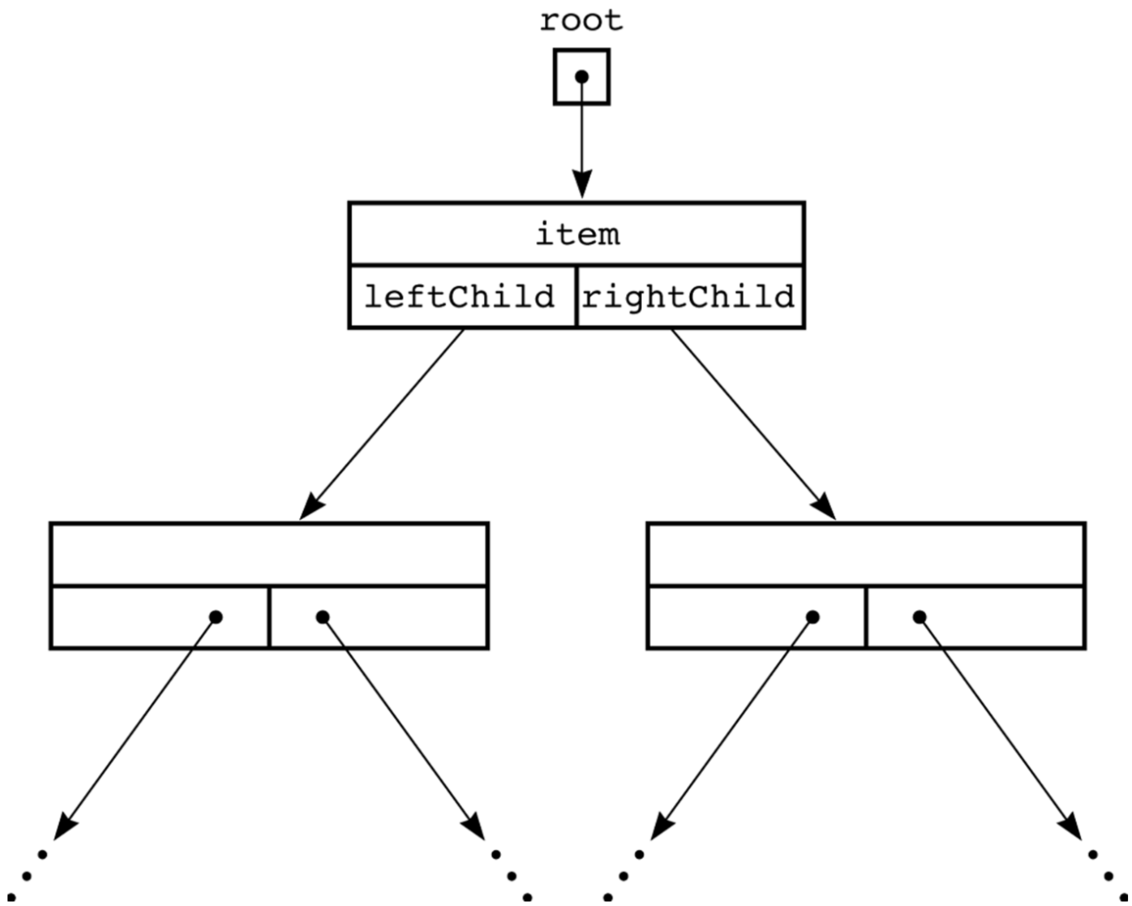
Node i's children:

$$2i + 1, 2i + 2$$

Node i's parent: $\left\lfloor \frac{i-1}{2} \right\rfloor$

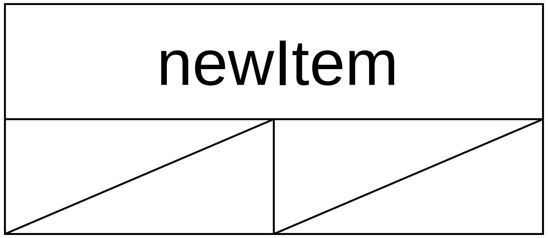
No link needed!

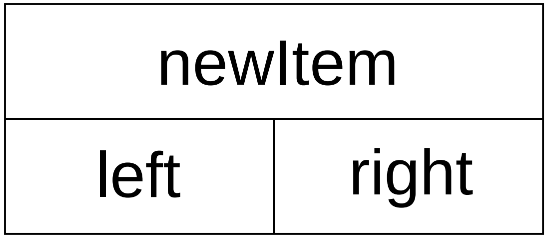
Reference-Based Representation



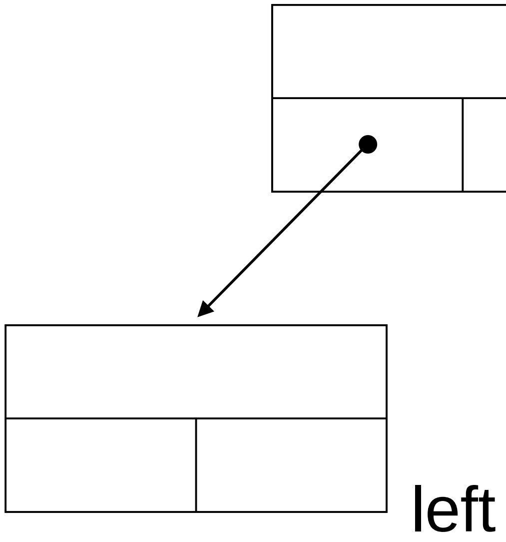
Reference-Based Implementation of Binary Tree

```
public class TreeNode {
    private Object item;
    private TreeNode leftChild;
    private TreeNode rightChild;
    public TreeNode(Object newItem) {
        item = newItem;
        leftChild = rightChild = null;
    }
    public TreeNode(Object newItem, TreeNode left, TreeNode right) {
        item = newItem;
        leftChild = left;
        rightChild = right;
    }
```





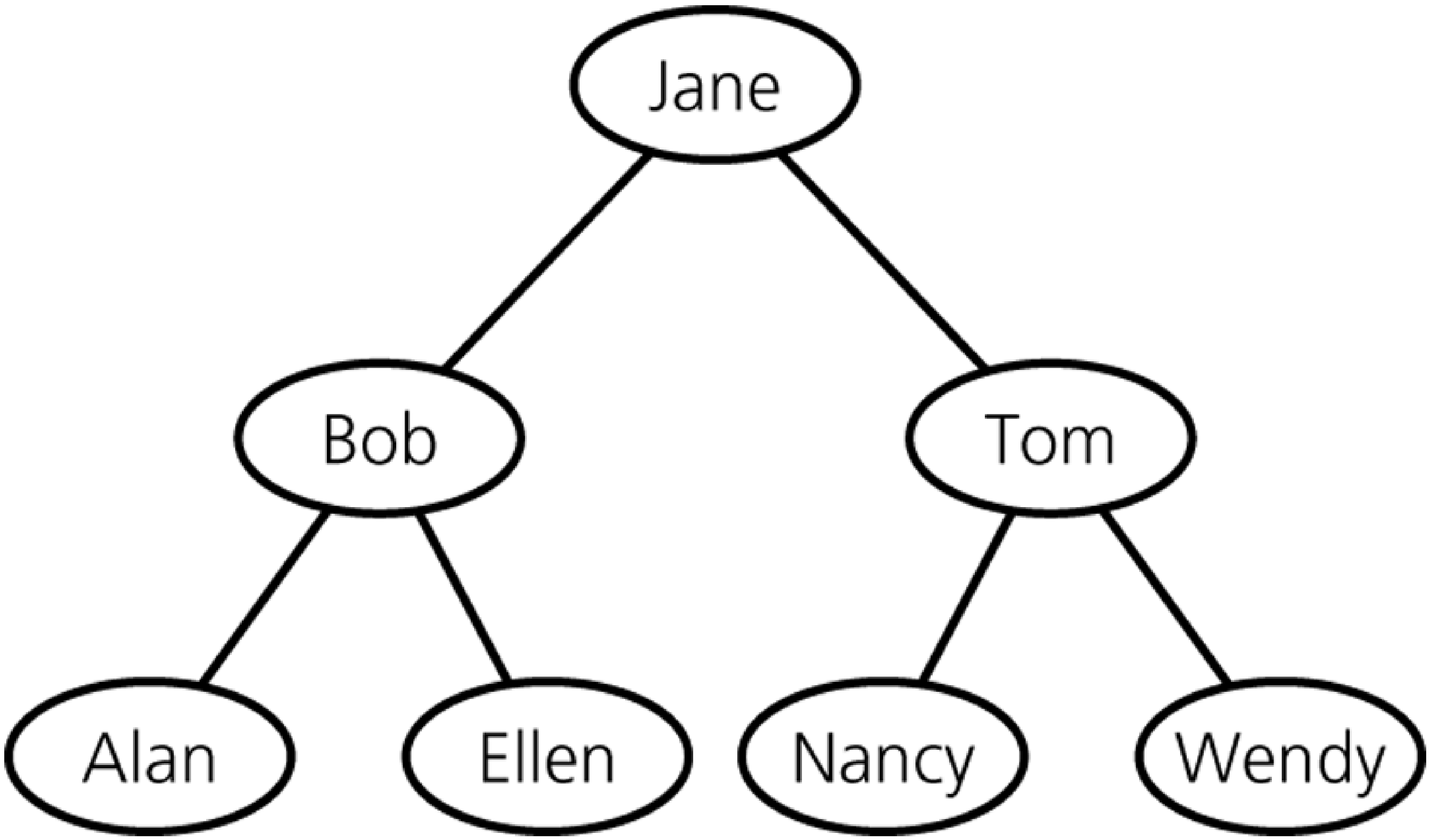
```
    public Object getItem( ) {
            return item;
    }
    public void setItem(Object newItem) {
            item = newItem;
    }
    public TreeNode getLeft( ) {
            return leftChild;
    }
    public TreeNode getRight( ) {
            return rightChild;
    }
    public setLeft(TreeNode left) {
            leftChild = left;
    }
    public setRight(TreeNode right) {
            rightChild = right;
    }
} // end TreeNode
```



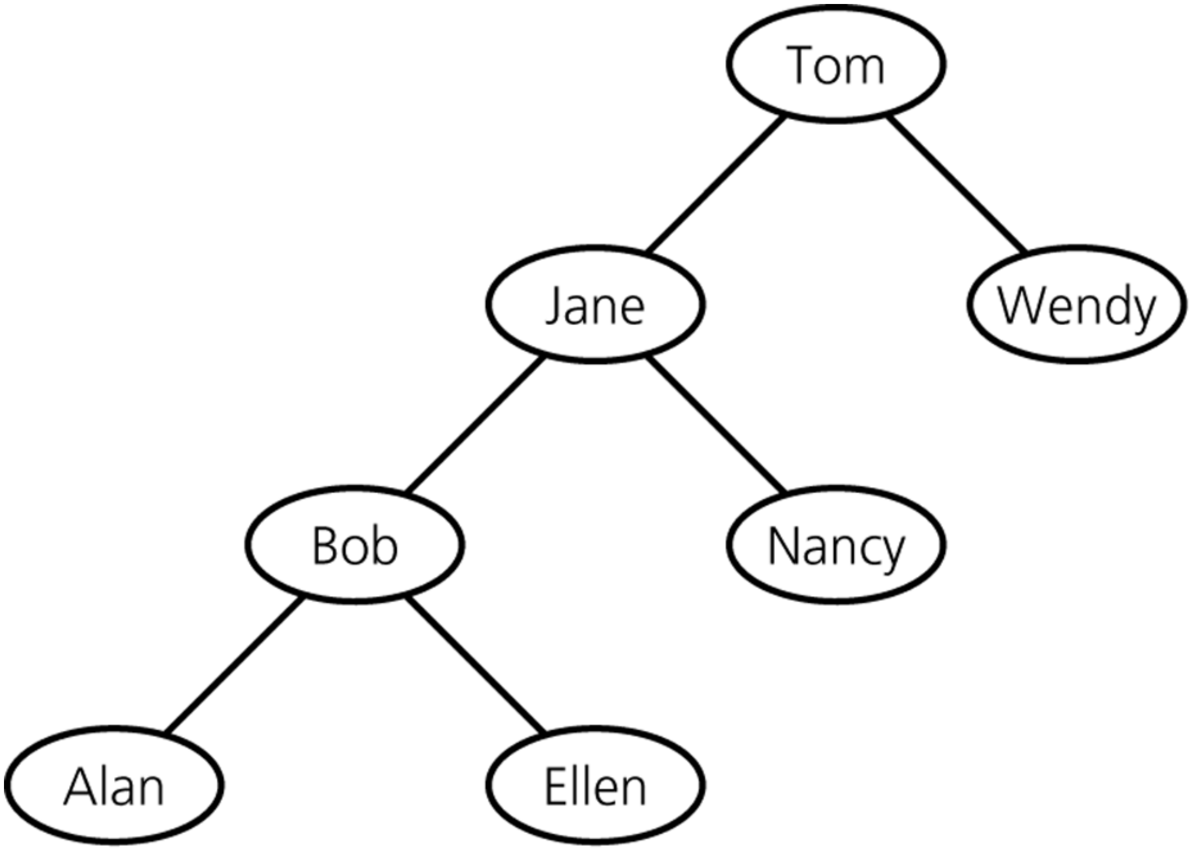
Binary Search Tree

- Each node has a search key
  - There are no duplications among the search keys in a binary search tree
- For each node ***n***, it satisfies:
  - ***n***'s key is greater than all keys in its left subtree T_L
  - ***n***'s key is less than all keys in its right subtree T_R
  - Both T_L and T_R are binary search trees

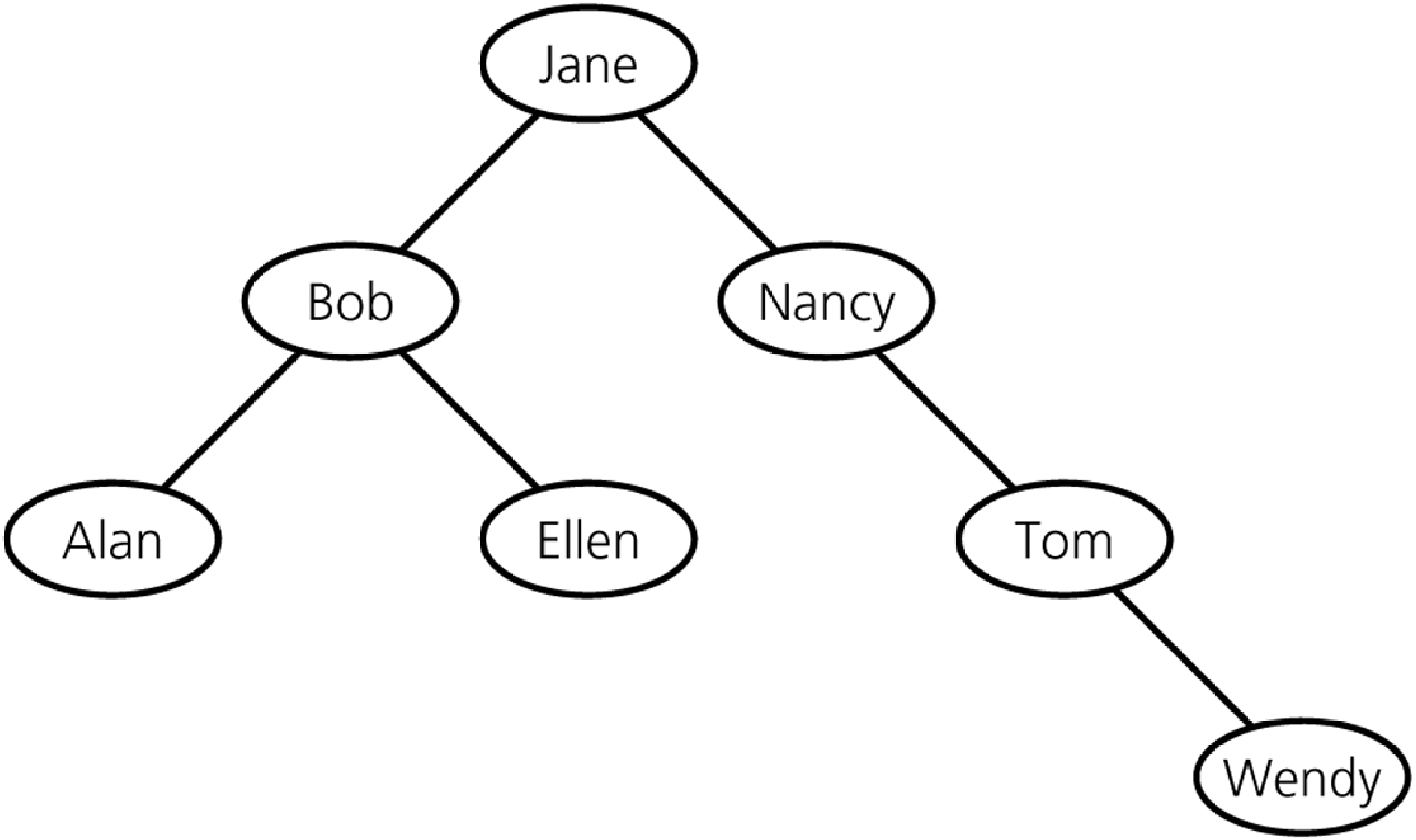
A Binary Search Tree of Names



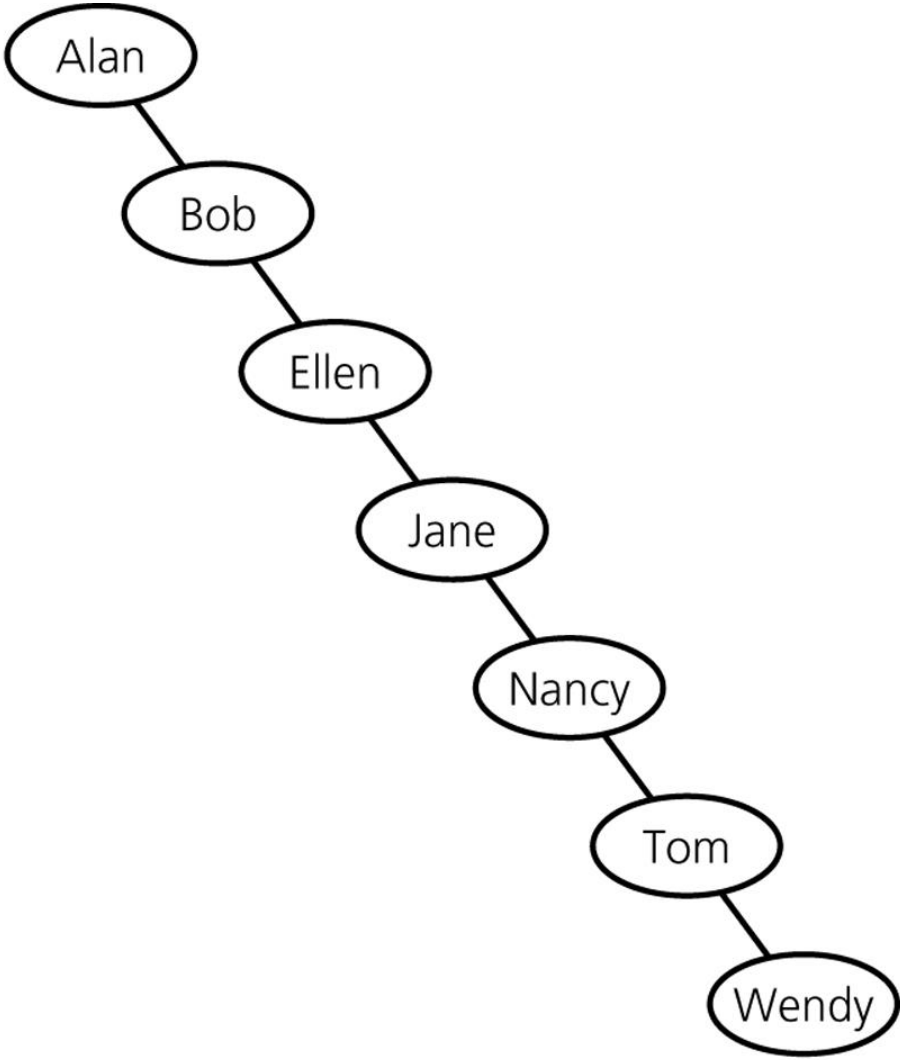
Another Binary Search Tree w/ the Same Data



Yet Another



Yet Another



ADT *Binary Search Tree* Operations

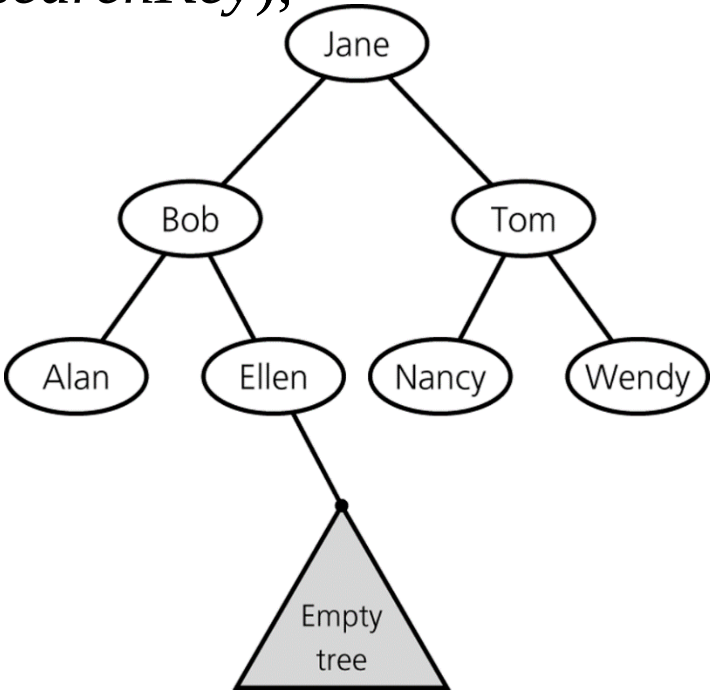
- Insert a new item into a binary search tree
- Delete the item w/ a given search key from a binary search tree
- Retrieve the item w/ a given search key from a binary search tree

✓ Binary search tree는 index(색인, 찾아보기)용으로 유용하다

Search in a Binary Search Tree

```
search(root, searchKey) {
    if (root is empty) return "Not found!";
    else if (searchKey == root's key) return root;
    else if (searchKey < root's key)
        return search(root's left child, searchKey);
    else
        return search(root's right child, searchKey);
}
```

```
search(root, searchKey) {
    if (root is empty) return "Not found!";
    else if (searchKey == root's key) return root;
    else if (searchKey < root's key)
        return search(root's left child, searchKey);
    else
        return search(root's right child, searchKey);
}
```



Insertion in a Binary Search Tree

```
insert (root, newItem) {
        if (root is null) {
                newItem을 key로 가진 새 node를 매단다;
        }
        else if (newItem < root's key)
                insert(root's left child, newItem);
        else
                insert(root's right child, newItem);
}
```

✓ Search()와 구조가 거의 같다

Deletion in a Binary Search Tree

deleteItem (*root*, searchKey) {
        dNode = search(*root*, searchKey);
        deleteNode(*dNode*);
}

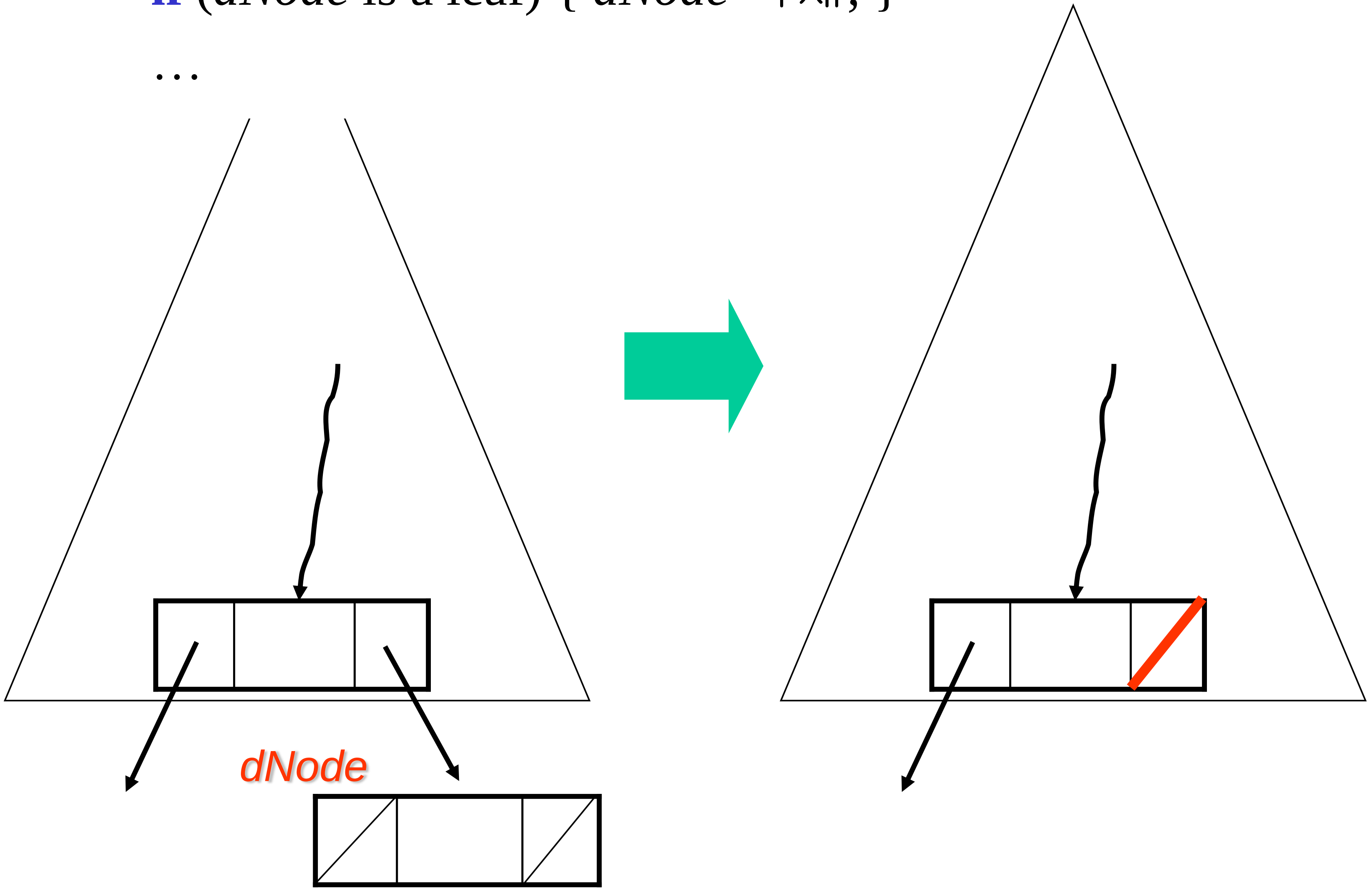
✓ Binary search tree의 operation들 중 상대적으로 복잡

```
deleteNode (dNode) {
        if (dNode is a leaf) { dNode 삭제; } // case 1
        else if (dNode has only one child c) {   // case 2
                c replaces dNode;
        } else { // dNode has two children          // case 3
                minNode =  dNode' right subtree의 leftmost node;
                // minNode has at most one right child
                minNode replaces dNode;
                deleteNode(minNode);
        }
}
```

deleteNode (*dNode*) { // case 1

if (*dNode* is a leaf) { *dNode* 삭제; }

…



deleteNode (*dNode*) { // case 2

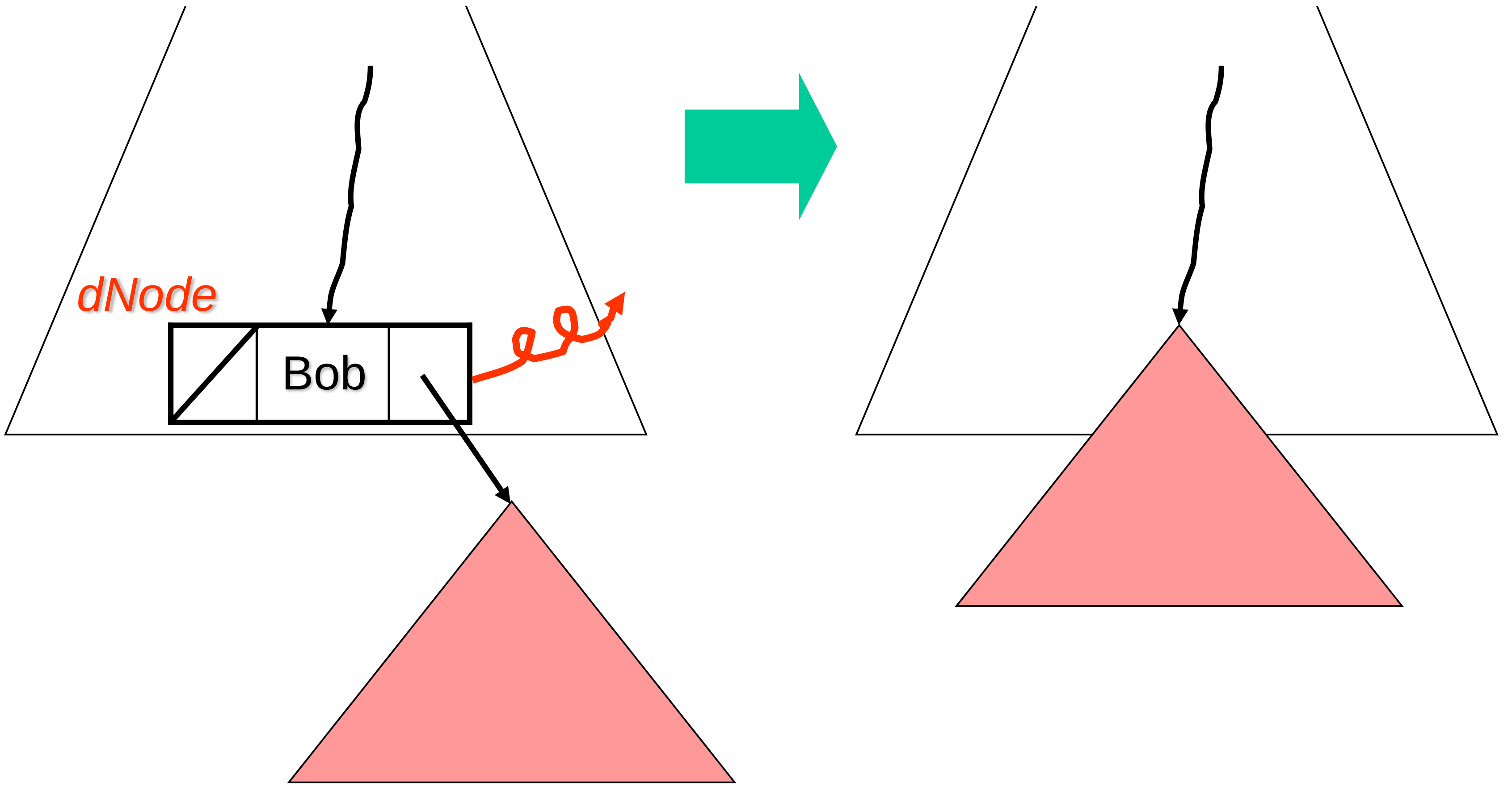
. . .

else if (*dNode* has only one child ***c***) {

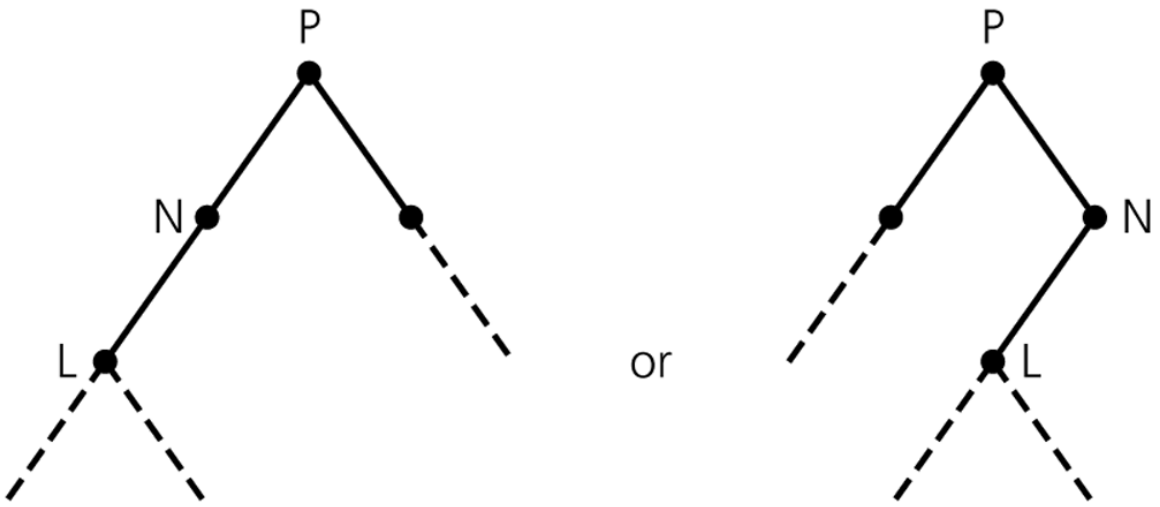
c replaces *dNode*;

}

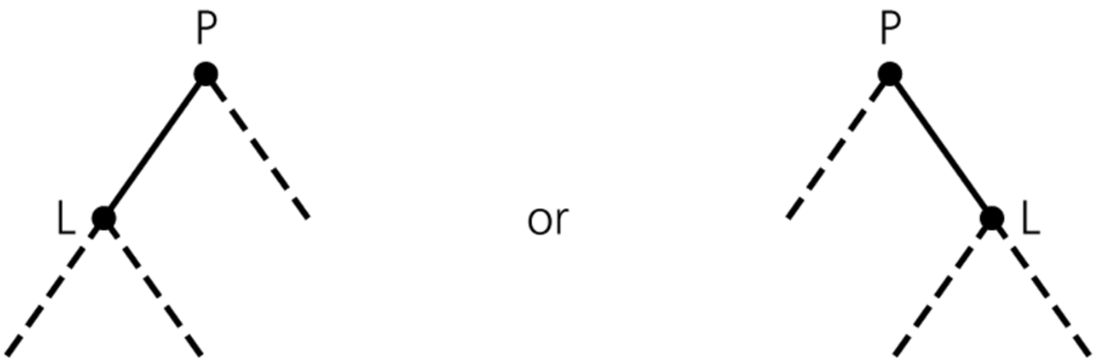
. . .



N with only a left child

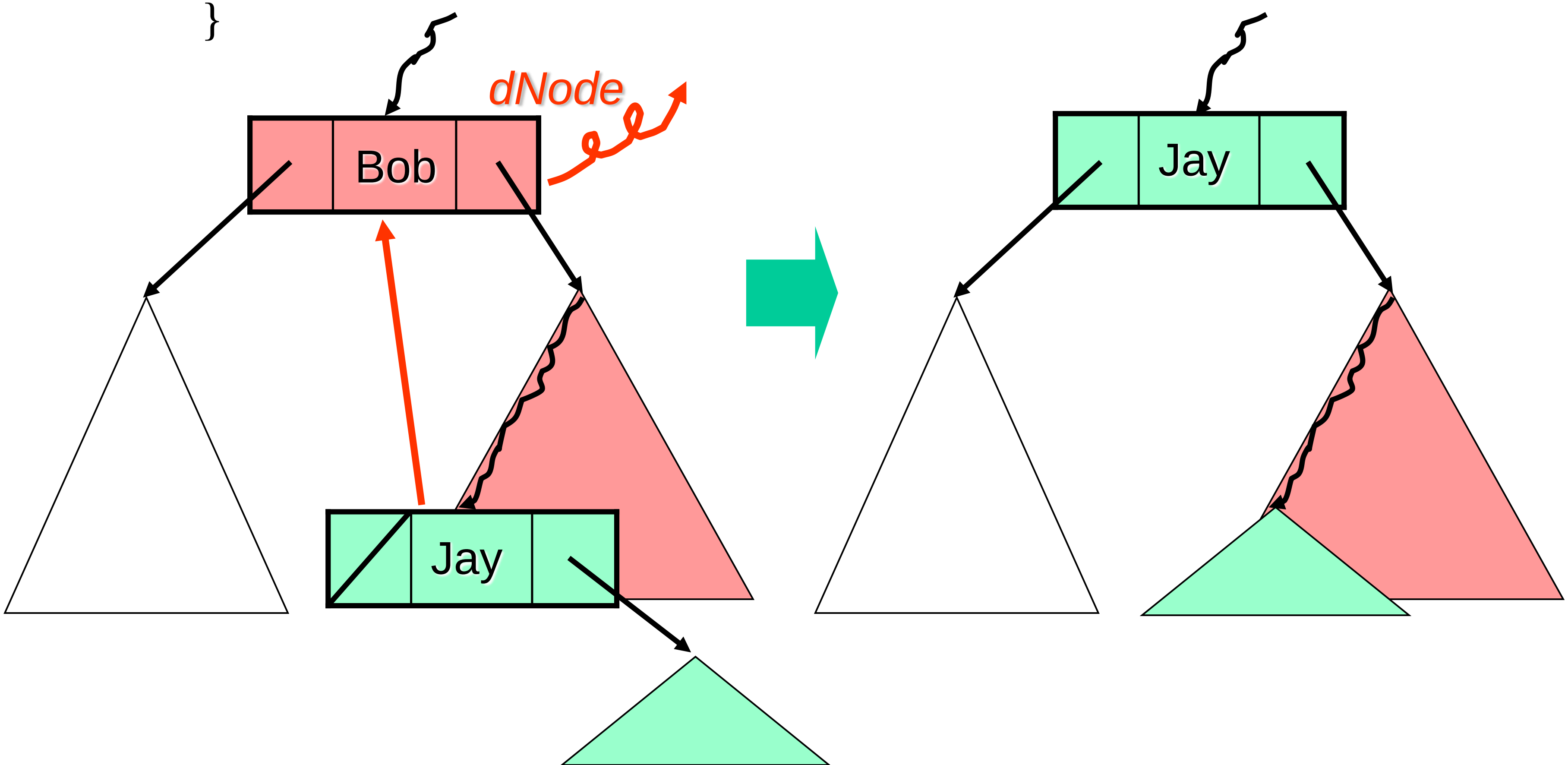


N can be either the left or right child of *P*



After deleting node *N*

```
deleteNode (dNode) {  // case 3
        ...
        } else { // dNode has two children
                minNode = dNode' right subtree의 leftmost node;
                // minNode has at most one right child
                 minNode replaces dNode;
                deleteNode(minNode);
        }
```

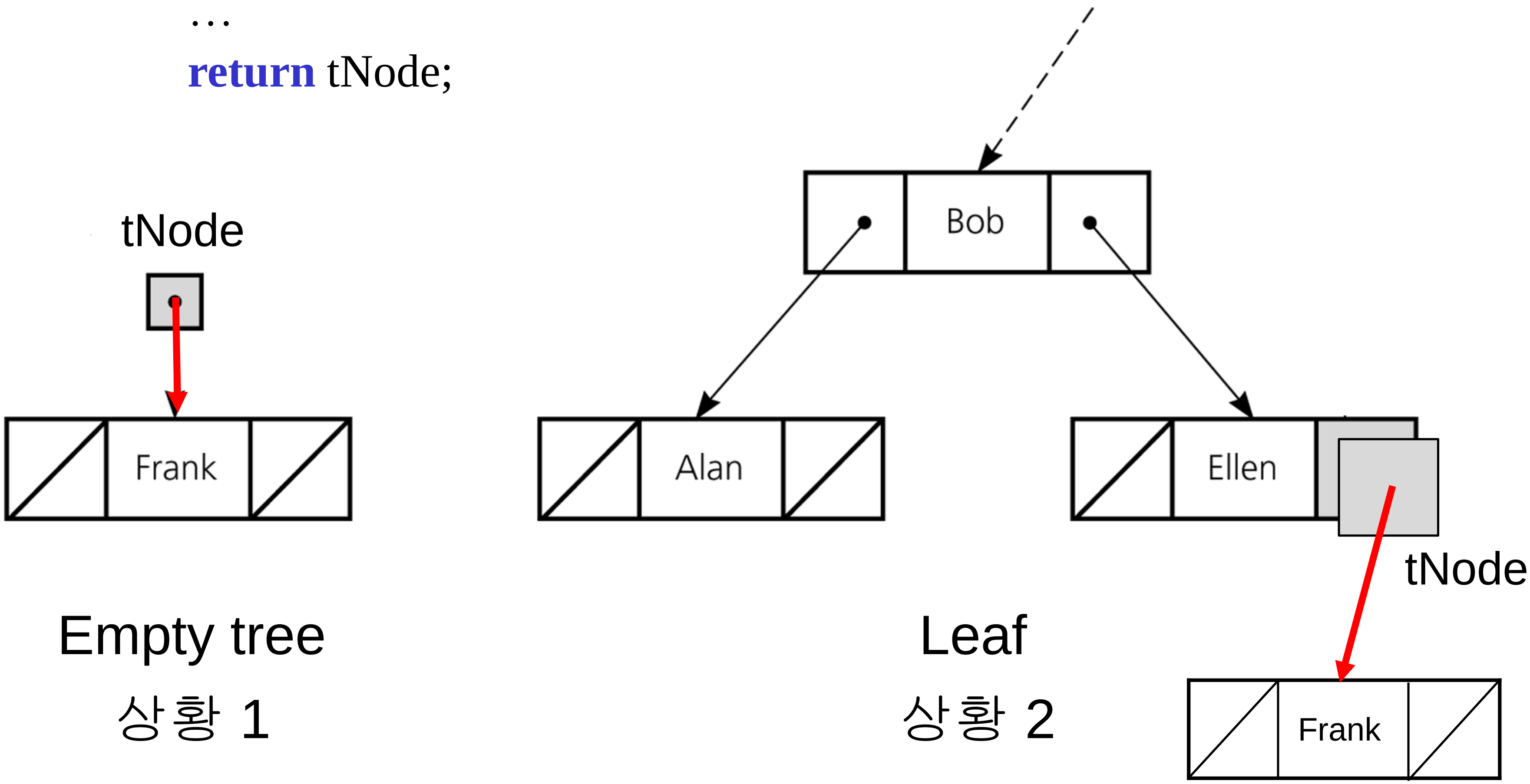


More Detailed Pseudo-Code (Reference-Based)

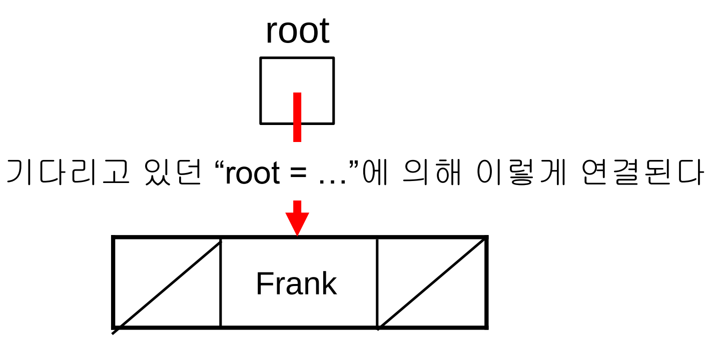
```
insert(Comparable newItem) {
                root = insertItem(root, newItem);
}
TreeNode insertItem(TreeNode tNode, Comparable newItem) {
        if (tNode == null) { // insert after a leaf  (or into an empty tree)
                tNode = new TreeNode(newItem, null, null);
        } else  if (newItem < tNode's item) { // branch left
                tNode.setLeft( insertItem(tNode.getLeft( ), newItem) );
        } else { // branch right
                tNode.setRight( insertItem(tNode.getRight( ), newItem) );
        }
        return tNode;
} // end insertItem
```

✓ tNode는 null일 때만 값이 바뀐다

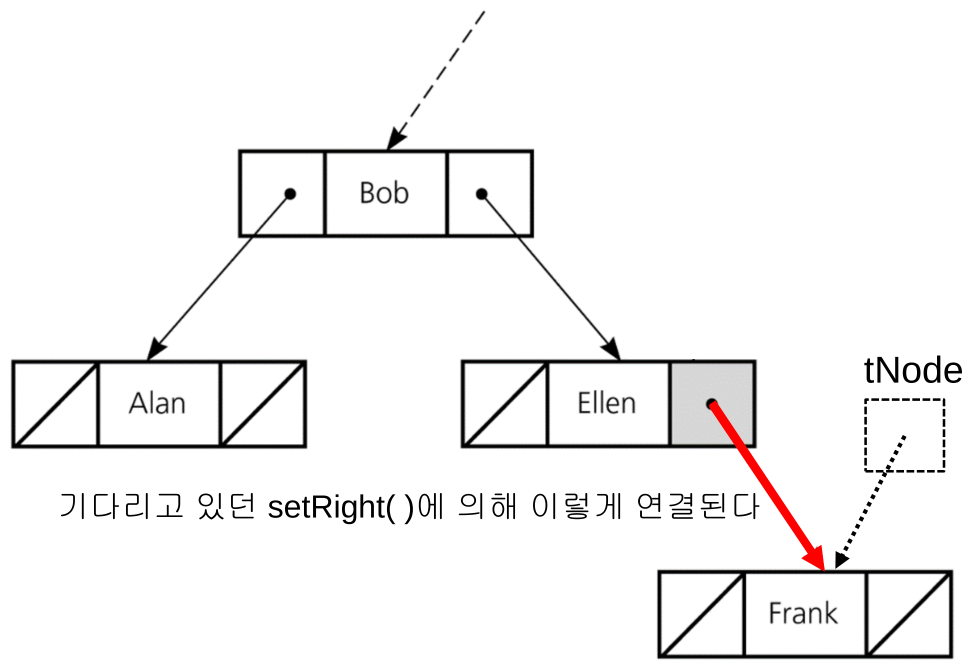
```
TreeNode insertItem(TreeNode tNode, Comparable newItem) {
        if (tNode == null) { // insert after a leaf (or into an empty tree)
                tNode = new TreeNode(newItem, null, null);
        }
        …
        return tNode;
```



```
TreeNode insertItem(TreeNode tNode, Comparable newItem) {
        if (tNode == null) { // insert after a leaf (or into an empty tree)
                tNode = new TreeNode(newItem, null, null);
        }
        …
        return tNode;
```

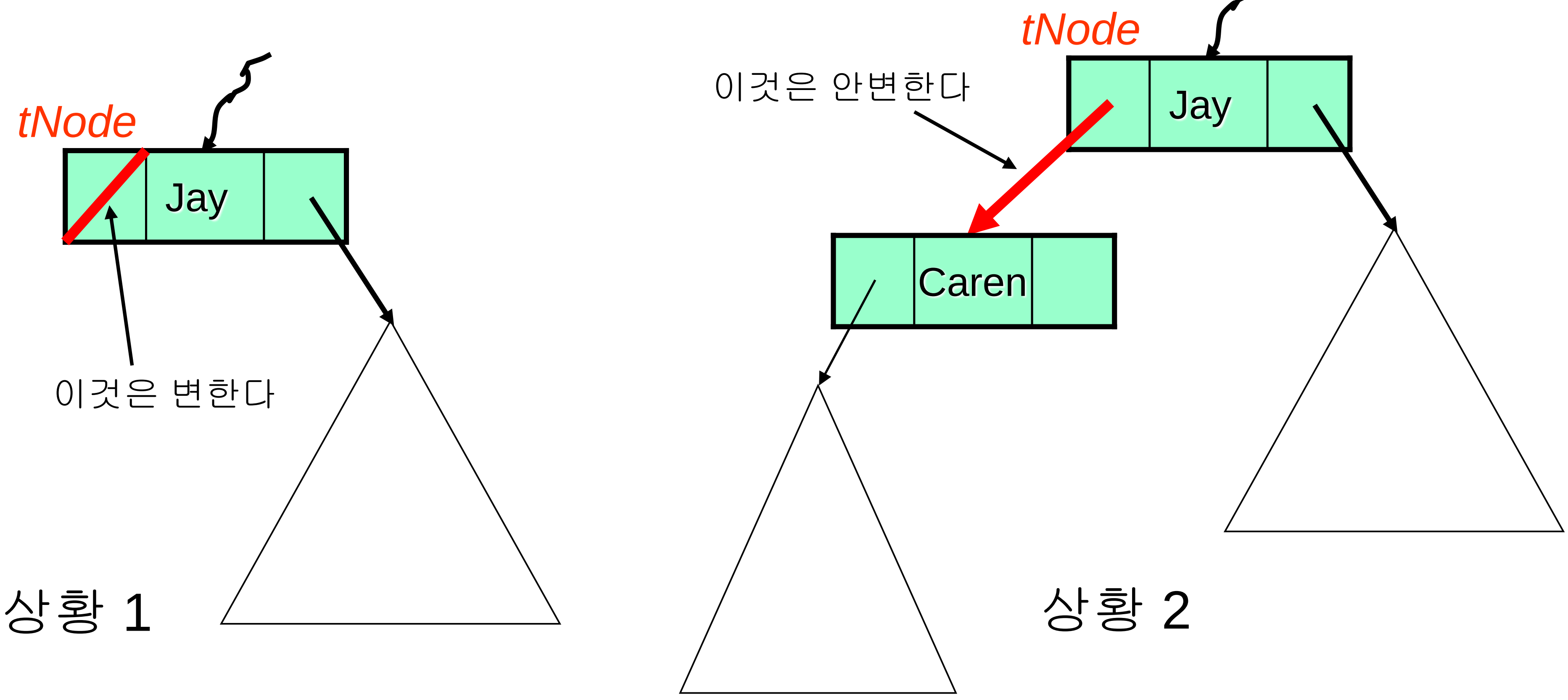


상황 1



상황 2

```
TreeNode insertItem(TreeNode tNode, Comparable newItem) {
        …
        if (newItem <  tNode's item) { // branch left
                tNode.setLeft( insertItem(tNode.getLeft( ), newItem) );
        }
        …
        return tNode;
```



```
TreeNode retrieve (Comparable searchKey) {
        return retrieveItem(root, searchKey);
}
TreeNode retrieveItem (TreeNode tNode, Comparable searchKey) {
        if (tNode == null) return null;  // not exist!
        else {
                if (searchKey == tNode’s key) return tNode;
                else if (searchKey < tNode’s key)
                        return retrieveItem(tNode.getLeft( ), searchKey);
                else
                        return retrieveItem(tNode.getRight( ), searchKey);
        }
}
```

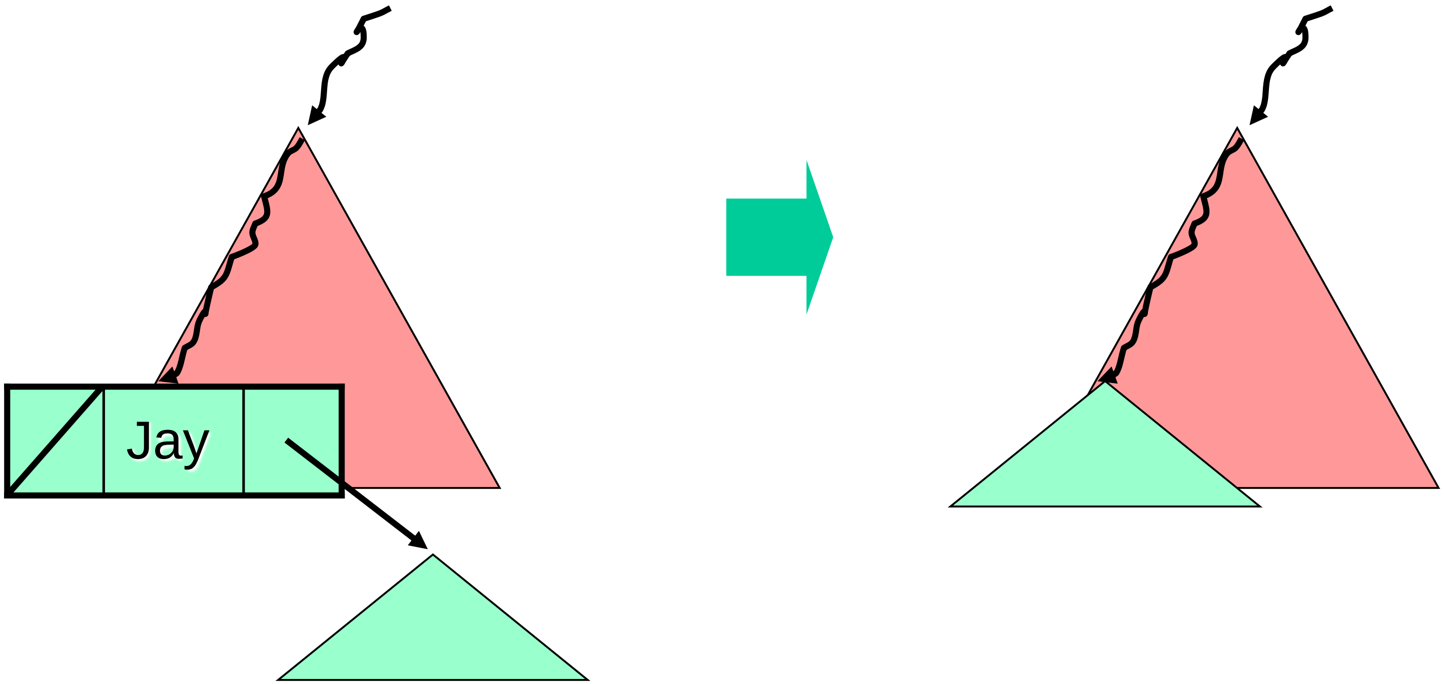
```
TreeNode deleteItem (TreeNode tNode, Comparable searchKey) {

    if (tNode == null) {exception 처리}; // item not found!
    else {
        if (searchKey == tNode's key) { // item found!
            tNode = deleteNode(tNode);
        } else if (searchKey < tNode's key) {
            tNode.setLeft(deleteItem(tNode.getLeft( ), searchKey));
        } else {
            tNode.setRight(deleteItem(tNode.getRight( ), searchKey) );
        }
    }
    return tNode; // tNode: parent에 매달리는 노드
}
```

```
TreeNode deleteNode (TreeNode tNode) {
    // Three cases
    //   1. tNode is a leaf
    //   2. tNode has only one child
    //   3. tNode has two children

    if ( (tNode.getLeft( ) == null)  && (tNode.getRight( ) == null)) { // case 1
            return null;
    } else if (tNode.getLeft( ) == null ) {  // case 2 (only right child)
            return tNode.getRight( );
    } else if (tNode.getRight( ) == null) {  // case 2 (only left child)
            return tNode.getLeft( );
    } else {  // case 3 – two children
            tNode.setItem(minimum item of tNode's right subtree);
            tNode.setRight(deleteMin(tNode.getRight( ));
            return tNode; // tNode survived
    }
}
```

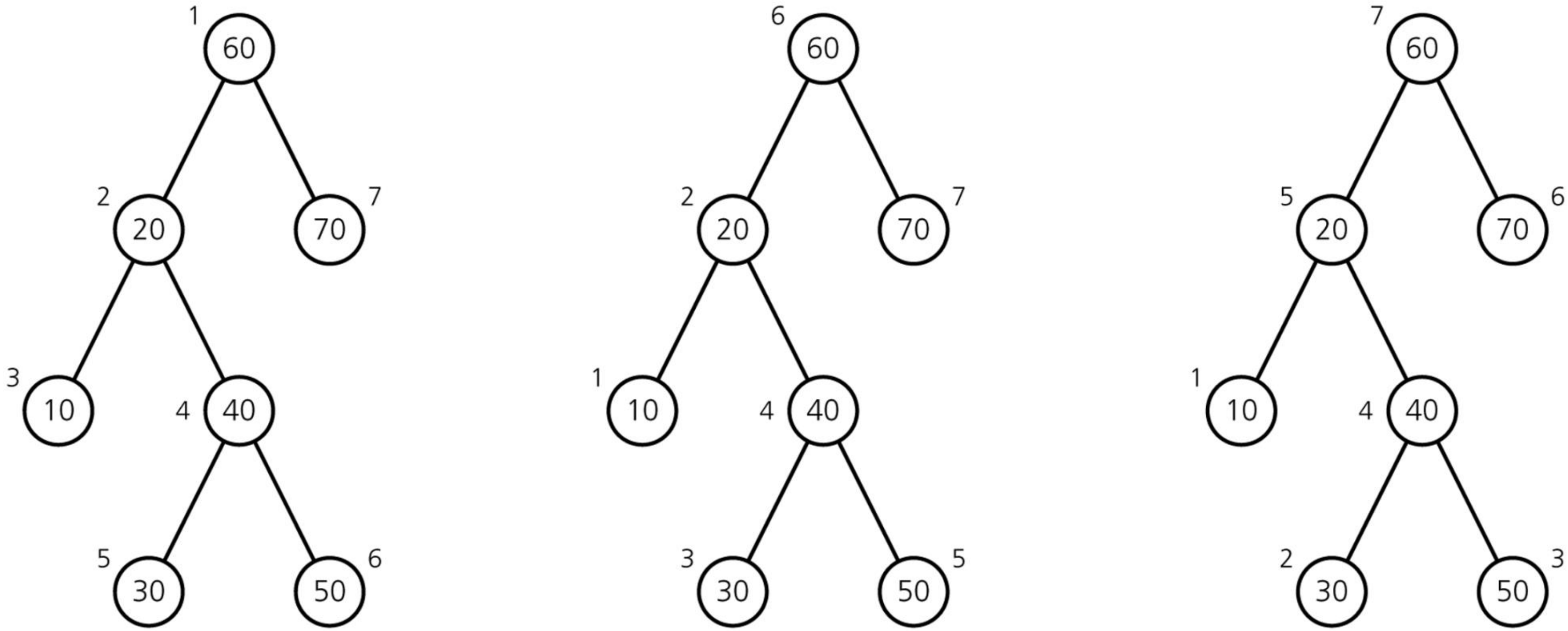
```
TreeNode deleteMin (TreeNode tNode) {
        if (tNode.getLeft( ) == null) { // found min
                return tNode.getRight( ); // right child moves to min's place
        } else { // branch left, then backtrack
                tNode.setLeft(deleteMin(tNode.getLeft( ));
                return tNode;
        }
}
```



Traversal of Binary Tree

- A traversal algorithm visits every node in the tree
- There are three representative traversal algorithms for binary trees
  - Preorder traversal
  - Inorder traversal
  - Postorder traversal

Preorder, Inorder, Postorder



(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.)

Preorder Traversal

```
preorder(root)
{
      if (root is not empty) {
            Mark root;
            preorder(Left subtree of root);
            preorder(Right subtree of root);
      }
}
```

Inorder Traversal

```
inorder(root)
{
    if (root is not empty) {
        inorder(Left subtree of root);
        Mark root;
        inorder(Right subtree of root);
    }
}
```

Postorder Traversal

```
postorder(root)
{
    if (root is not empty) {
        postorder(Left subtree of root);
        postorder(Right subtree of root);
        Mark root;
    }
}
```

Operations' Efficiency on B.S.T.

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

Properties of Binary Trees

Theorem 1

The *inorder* traversal of a binary search tree T visits its nodes in sorted search-key order.

<Proof>

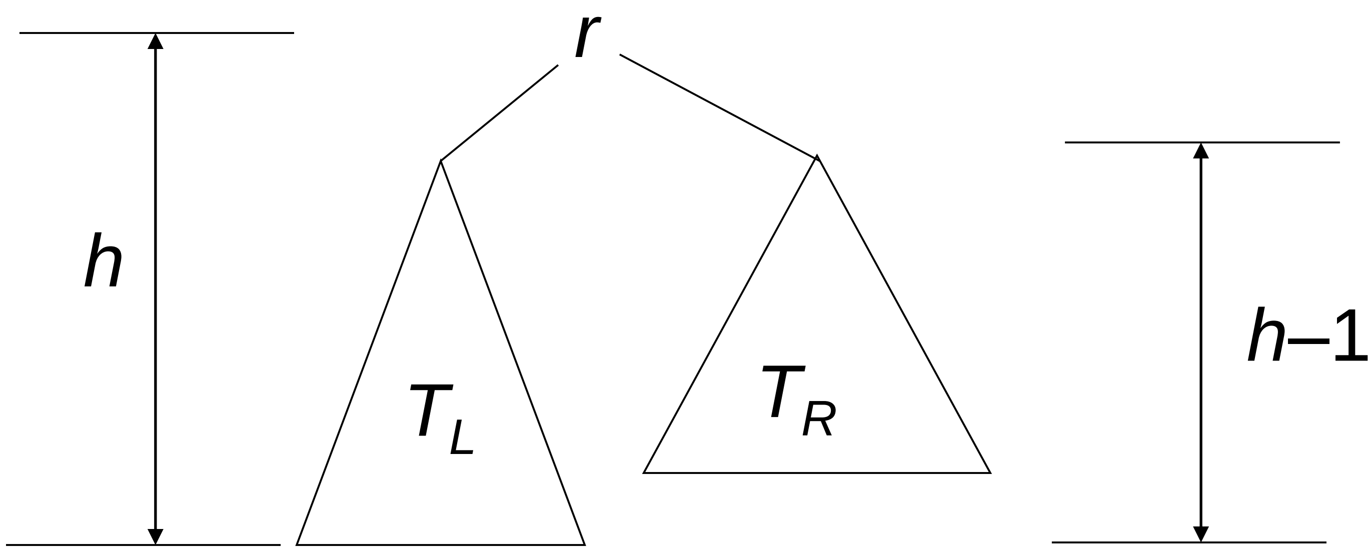
Basis: $h = 1$.

T consists of only one node, the root.

Visiting the only node is obviously in sorted order.

Inductive hypothesis: Assume that the theorem is true for all $k < h$.

Inductive conclusion: Want to show that the theorem is true for $k = h$. T is of the form



Inorder visits T_L and T_R in sorted order, respectively, by the inductive hypothesis. Because keys in T_L < r's key and keys in T_R > r's key, the *inorder* traversal of $T_L \rightarrow r \rightarrow T_R$ is in sorted order. ■

Height

Theorem 2

A full binary tree of height $h \geq 0$ has $2^h - 1$ nodes.

Corollary 1

The number of nodes in a binary tree of height h is at most $2^h - 1$.

Proofs are simple!

Theorem 3

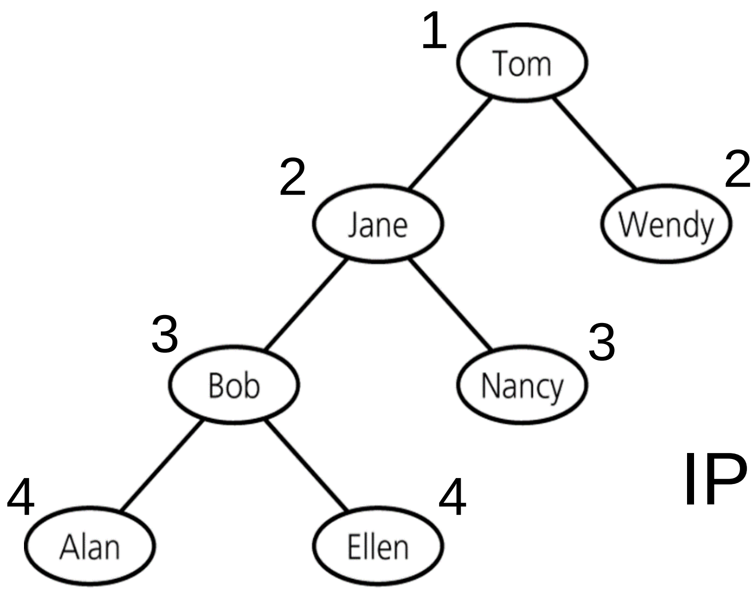
The minimum height of a binary tree with n nodes is $\lceil \log_2(n+1) \rceil$.

<Proof>

Straightforward by Corollary 1

Depth

- Definition: Internal Path Length (IPL)
  - Sum of depths of its nodes



IPL = 19

Theorem 4

The expected IPL of a binary tree with n nodes is $O(n \log n)$ under the assumption that all permutations are equally likely.

<Proof> Chapter11-IPL증명.doc

✓ Meaning: Average search time for an item is $O(\log n)$.

Tree Size 구하기

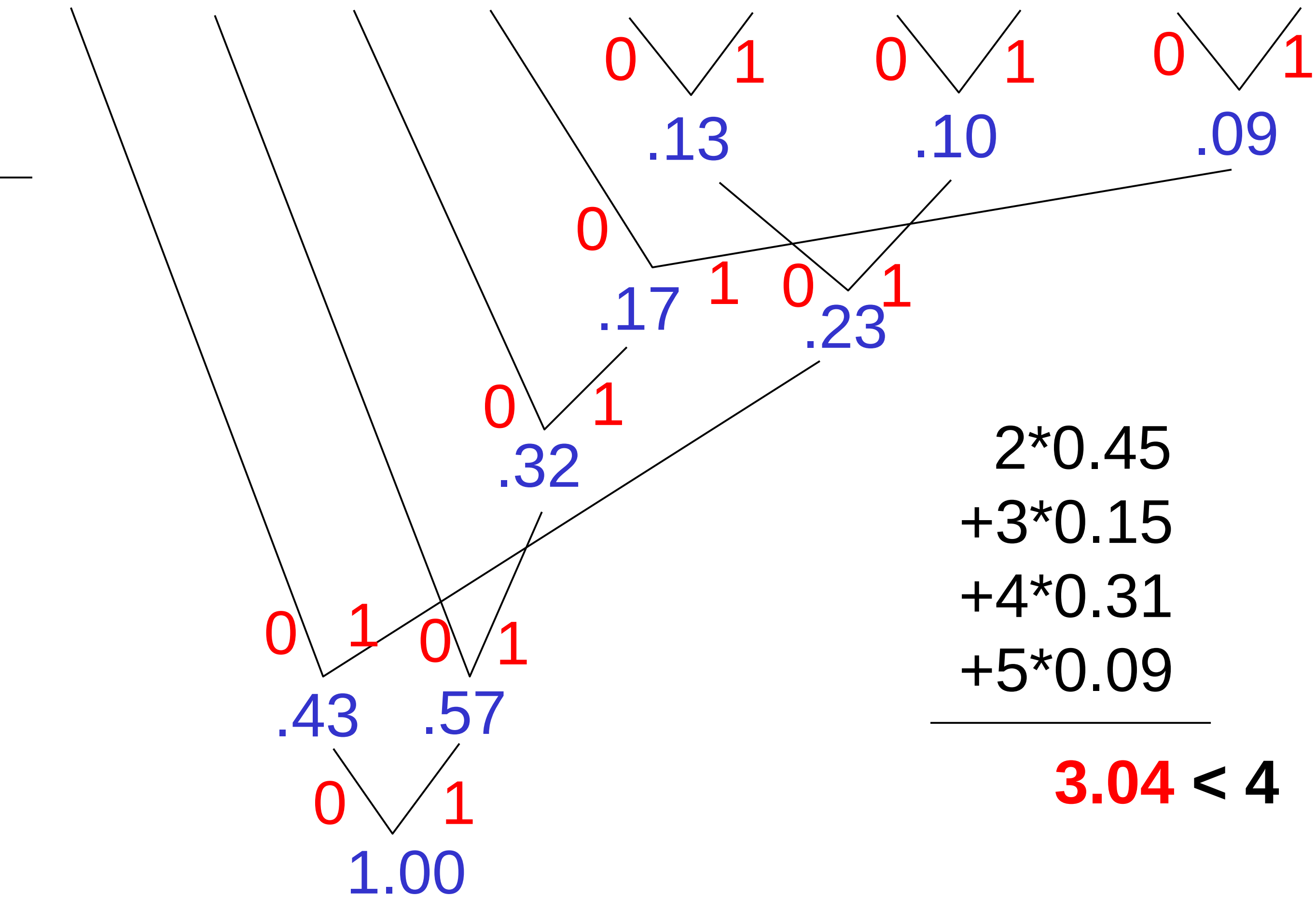
```
int size(TreeNode t)
{
        if (t == null) return 0;
        else return (1 + size(t.getLeft( )) + size(t.getRight( )));
}
```

An Example Use: Huffman Code

- A Simple data compression
- Examine the frequencies of each digit in the file
- Then, determine the code for each digit w/ a binary tree
    - ✓ Optimal in symbol-by-symbol encoding with given probabilities
    - ✓ Cf: an interesting history in relation to Shannon-Fano algorithm (top-down)

- E.g., Want to handle a file w/ only 10 digits

Digits	0	1	2	3	4	5	6	7	8	9
Freq.	.20	.25	.15	.08	.07	.06	.05	.05	.05	.04



0	00
1	10
2	110
3	1110
4	0100
5	0101
6	0110
7	0111
8	11110
9	11111

2*0.45
+3*0.15
+4*0.31
+5*0.09

3.04 < 4

Treesort

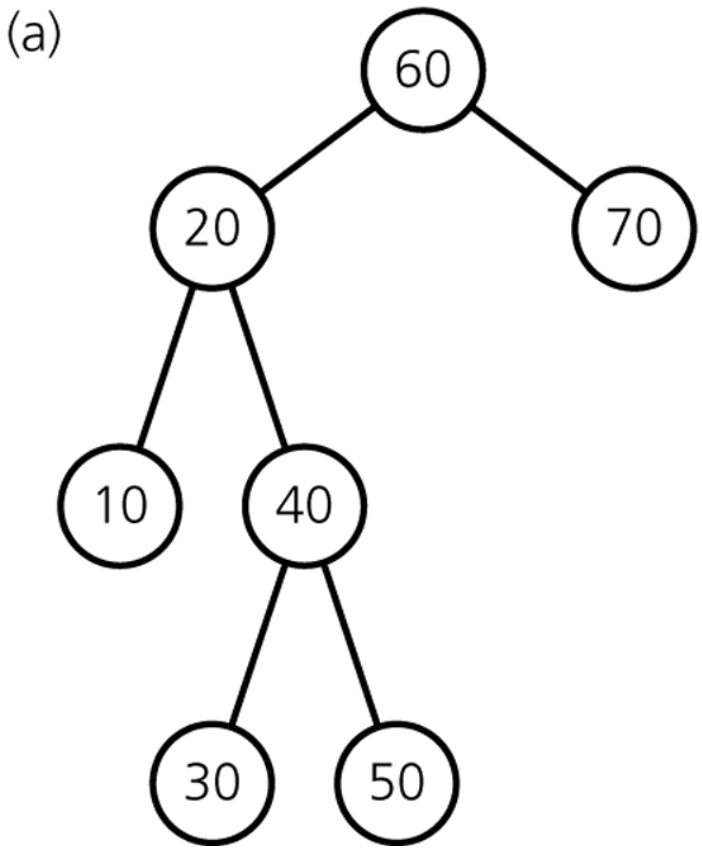
- Inorder traversal을 이용한 sorting 방법
  1. Element들을 모두 binary search tree로 넣는다
  2. Inorder traversal 순서대로 print 한다
- Performance
  - Average case: $O(n \log n)$
  - Worst case: $O(n^2)$

Saving a B.S.T. in a File

- Preserving the original shape
  - Use preorder for saving
- Restoring to a balanced shape
  - Use inorder for saving
  - Restoring

recursiveRestore (*L*) { // *L*: an array
    Set the median item ***r*** to be the root;
    r.leftChild = recursiveRestore(the left part of median);
    r.rightChild = recursiveRestore(the right part of median);
    return ***r*** ;
}

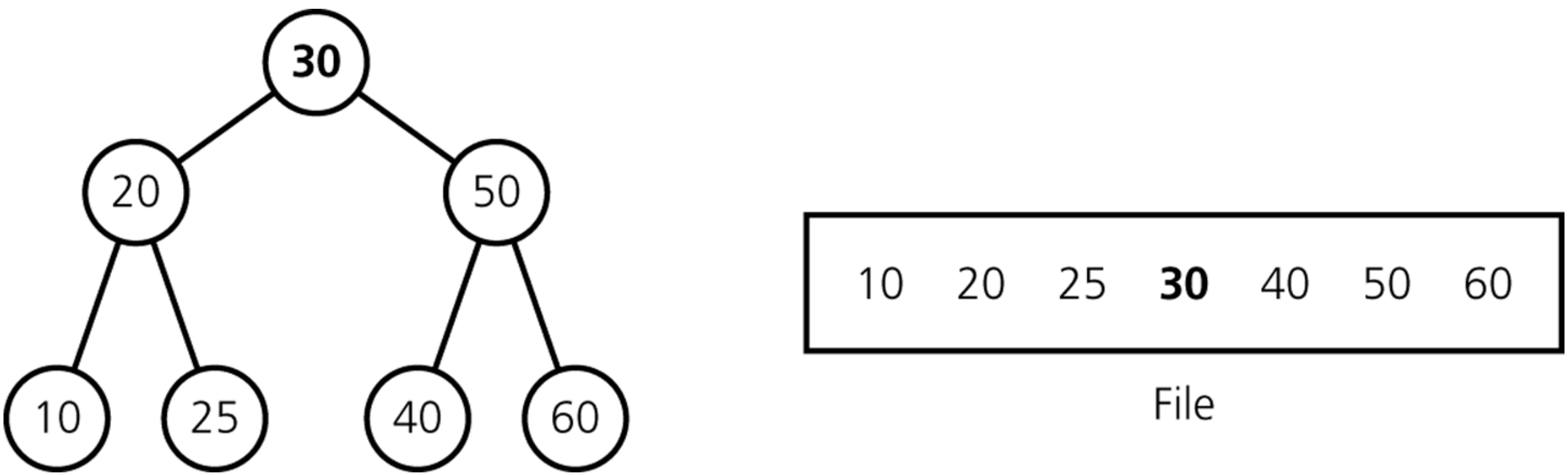
a) A binary search tree ***bst***; b) the sequence of insertions that result in this tree



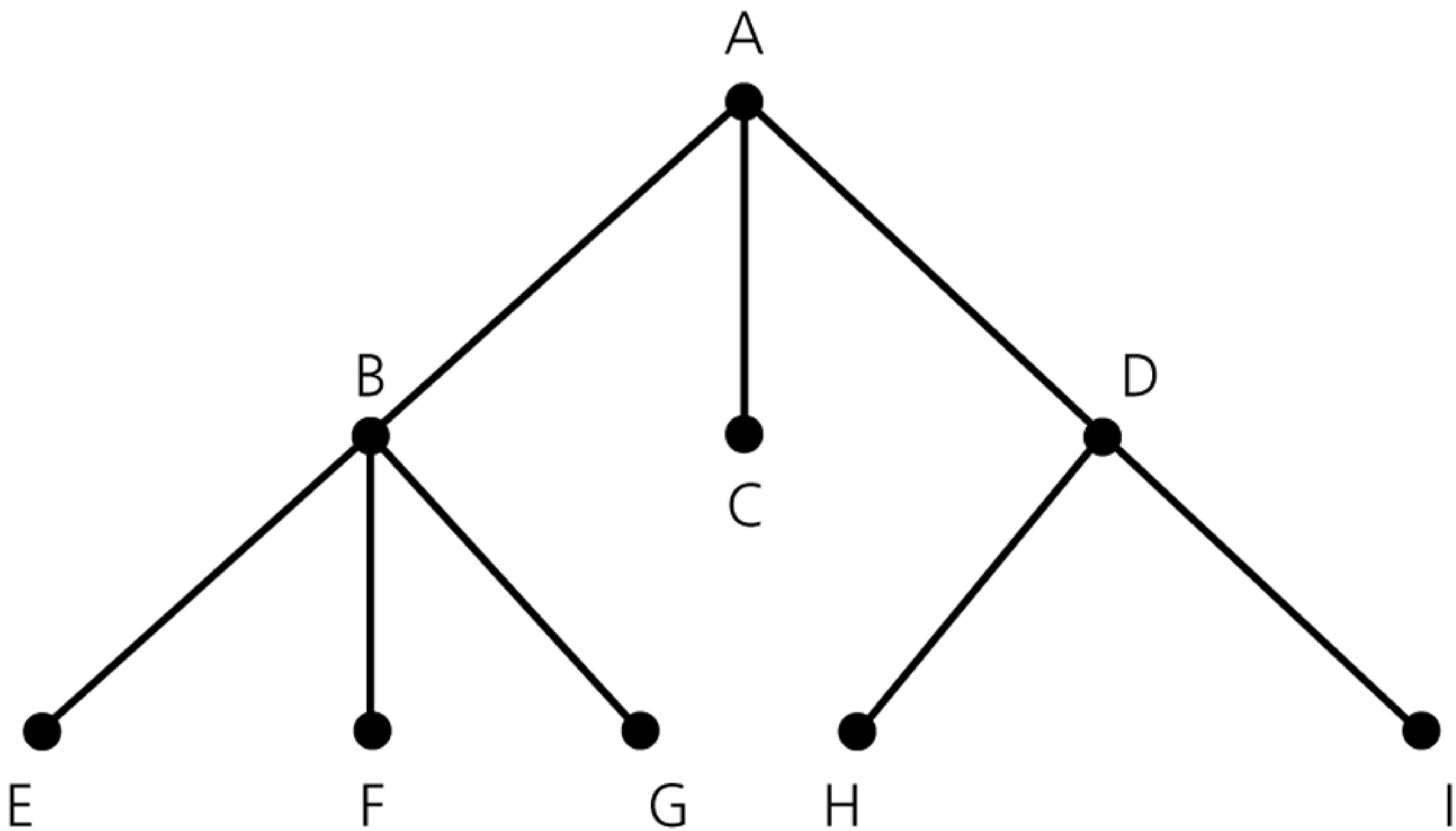
(b)

```
bst.insert(60);
bst.insert(20);
bst.insert(10);
bst.insert(40);
bst.insert(30);
bst.insert(50);
bst.insert(70);
```

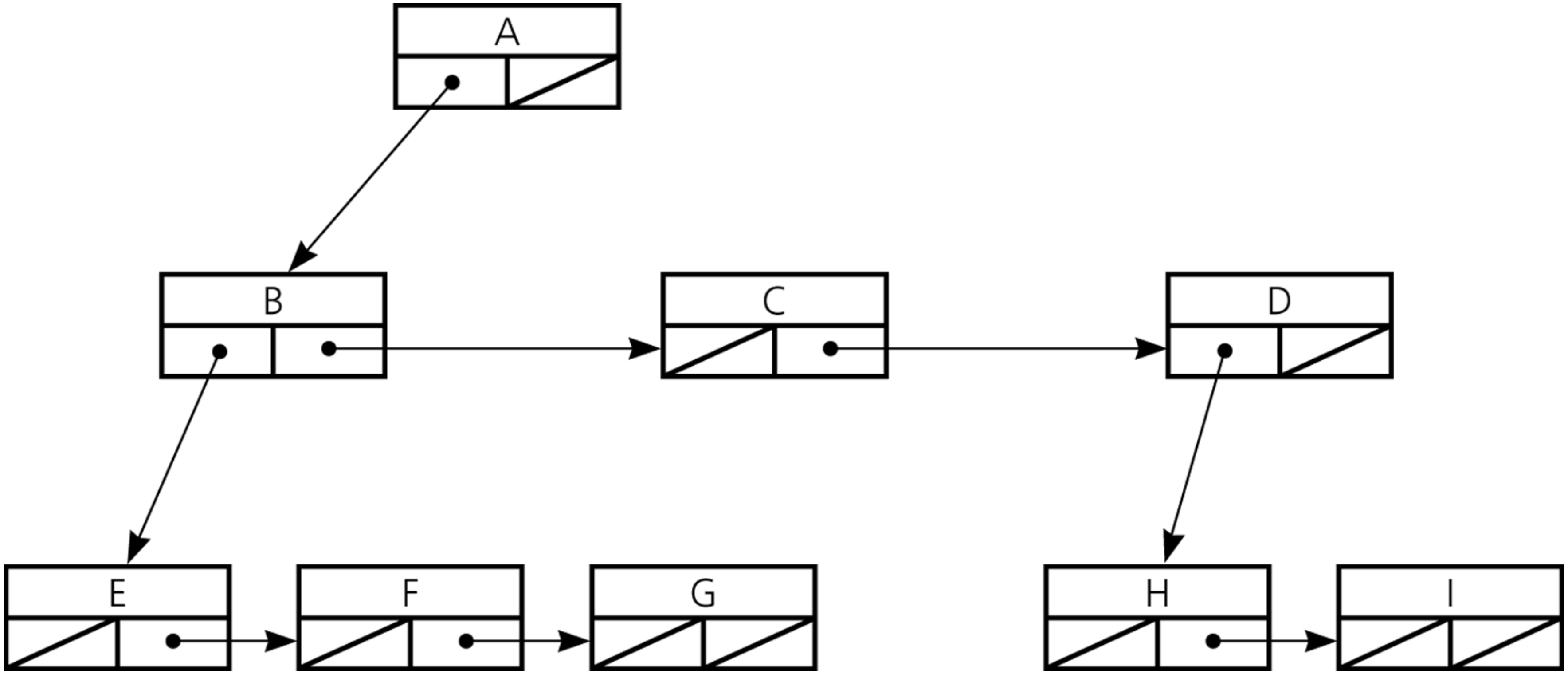
A full tree saved in a file by using inorder traversal



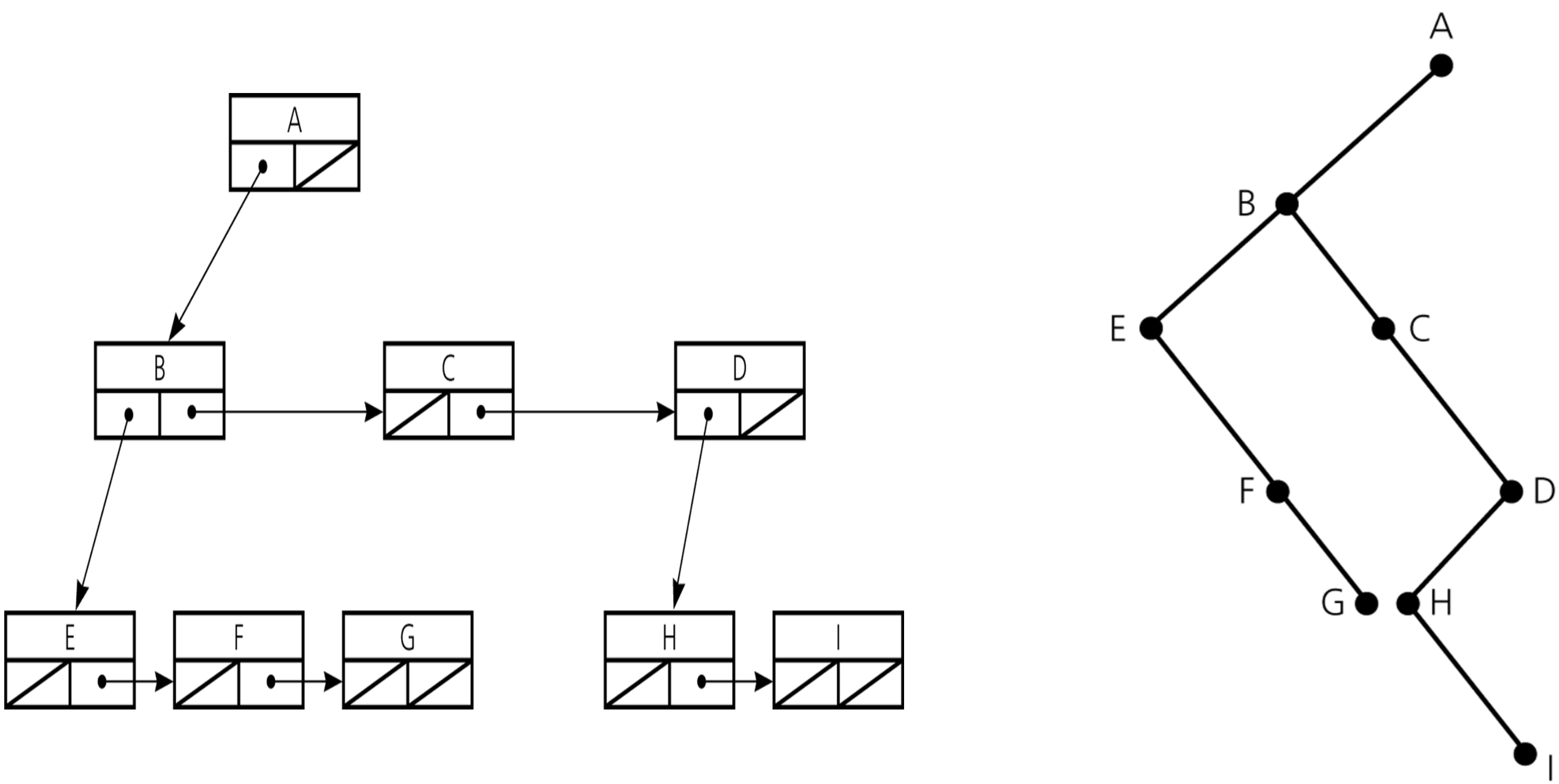
General Trees



A Reference-Based Implementation of General Trees



A General Tree and Corresponding Binary Tree



n-ary Tree