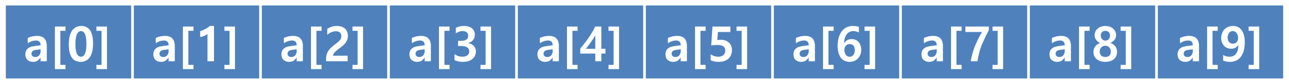
Trees

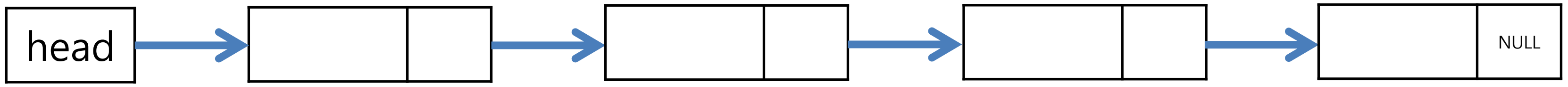
서승현 교수
Prof. Anes Seung-Hyun Seo
(seosh77@hanyang.ac.kr)

Division of Electrical Engineering
Hanyang University, ERICA Campus

Linear Data Structure

- 1D arrays, stack and lists are linear data structures

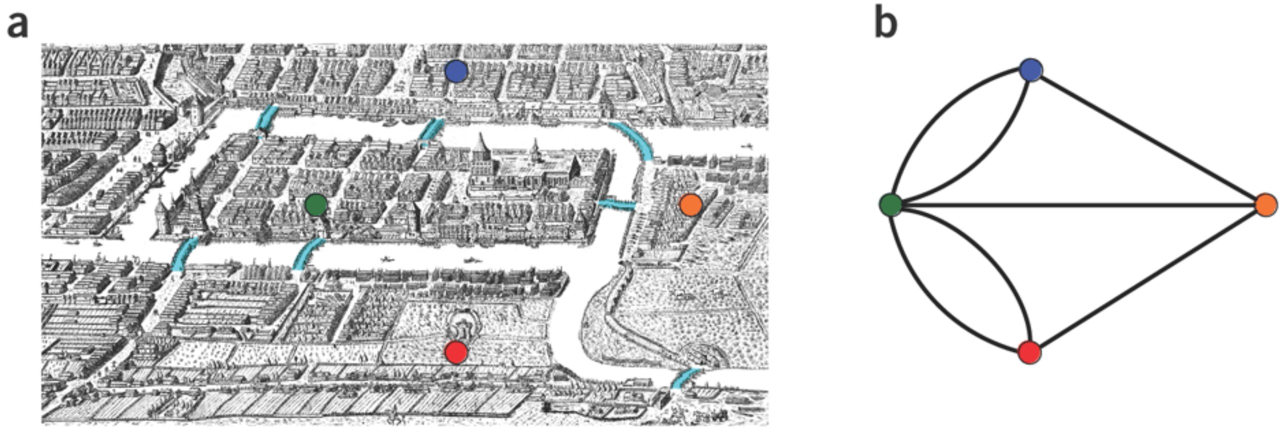


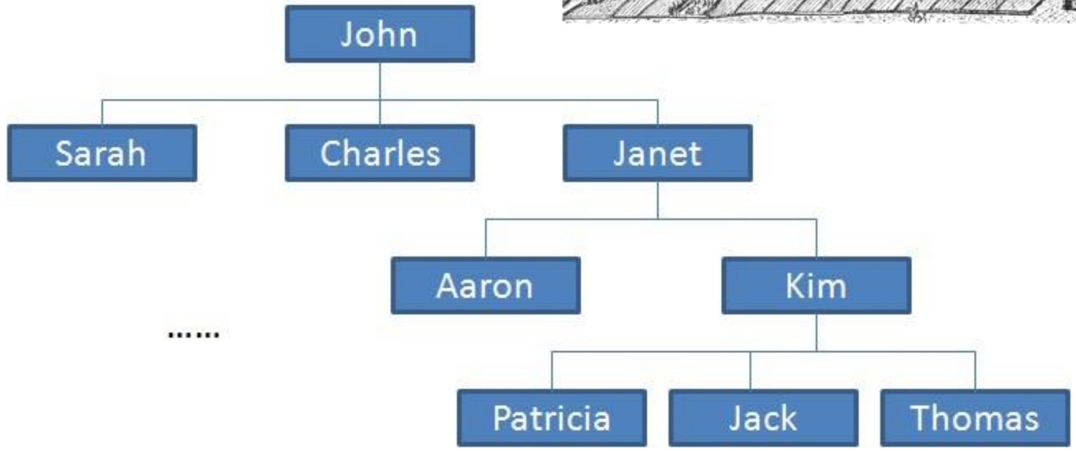


- What if we are dealing with a nonlinear data?

Examples of nonlinear data

■ **Family tree, roads in a map…**

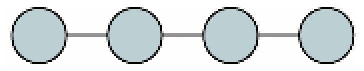




Basic Tree Concepts

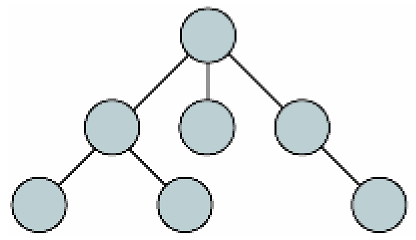
- **Logical structures**

Chap. 3~5



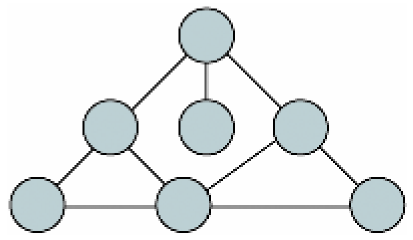
Linear list

Chap. 6



Tree

Chap. 7



Graph

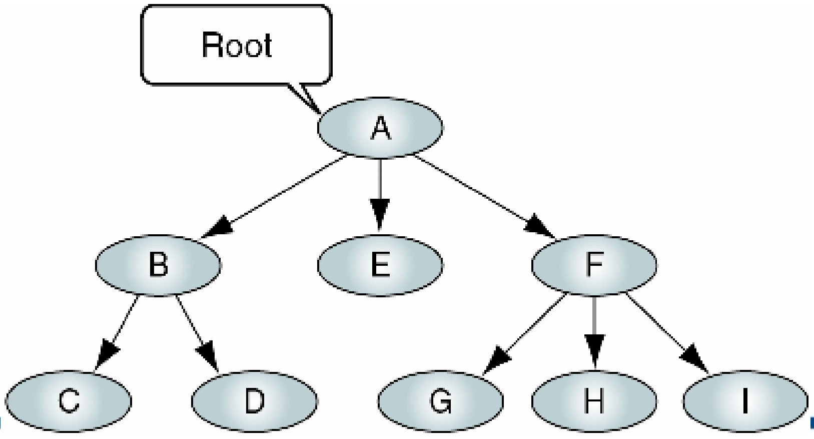
Linear structures

Non-linear structures

Basic Tree Concepts

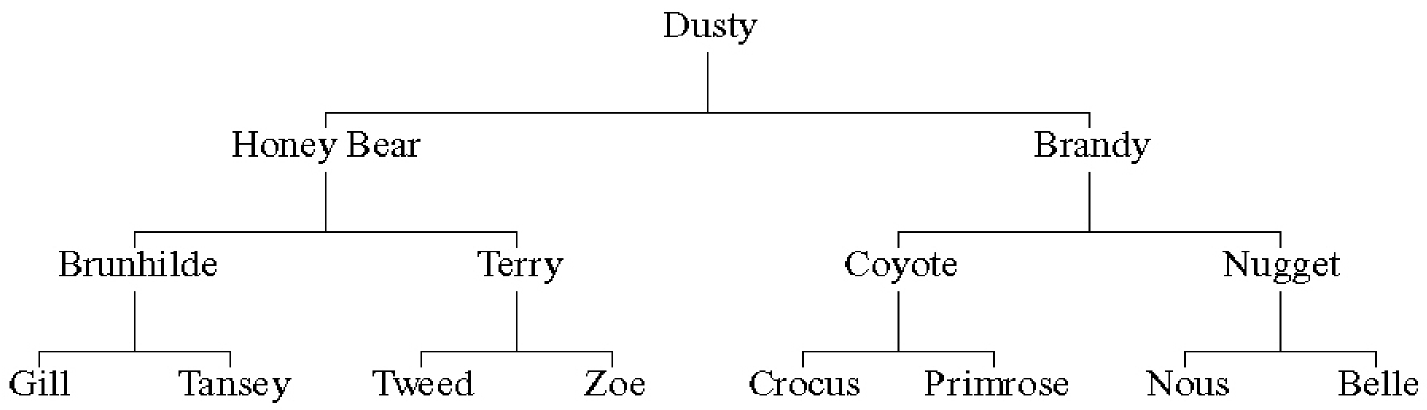
- **A Tree structure means that the data are organized in a hierarchical manner.**
- **Tree consists of**
  - A finite set of **nodes** (**vertices**)
  - A finite set of **branches** (**edges, arcs**) that connects the nodes
    - **Degree** of a node: # of branches
      - In-degree: # of branch toward the node
      - Out-degree: # of branch away from the node
  - Every non-empty tree has a **root node** whose in-degree is zero.

  In-degree of all other nodes is 1

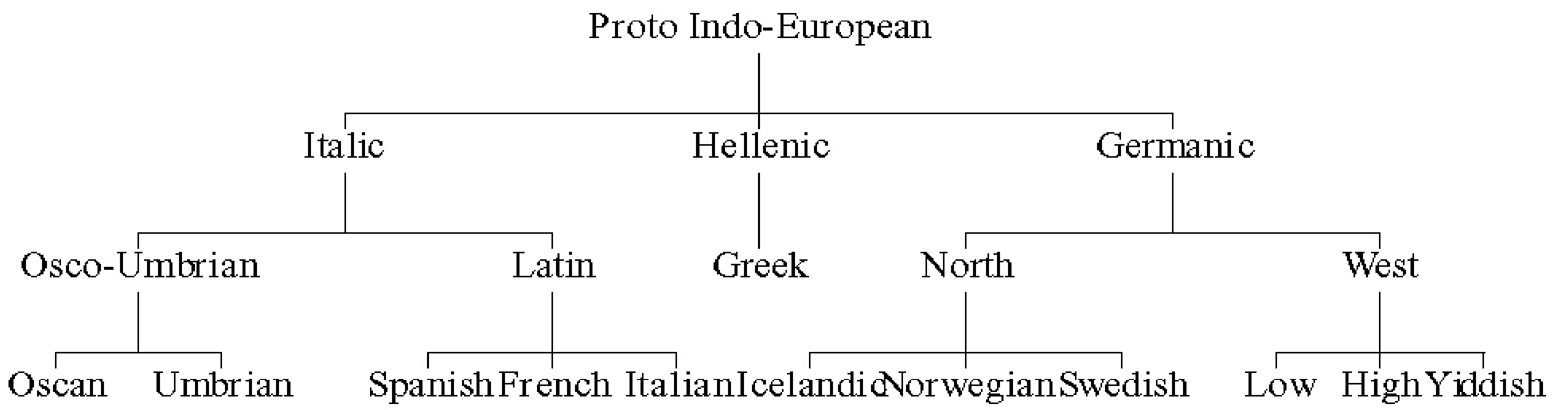


Why Trees?

- Tree is an effective representation for hierarchical structures



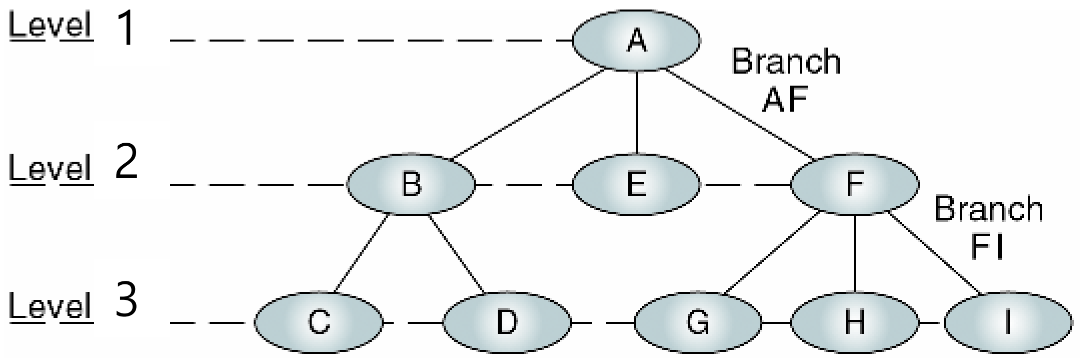
(a) Pedigree



(b) Lineal

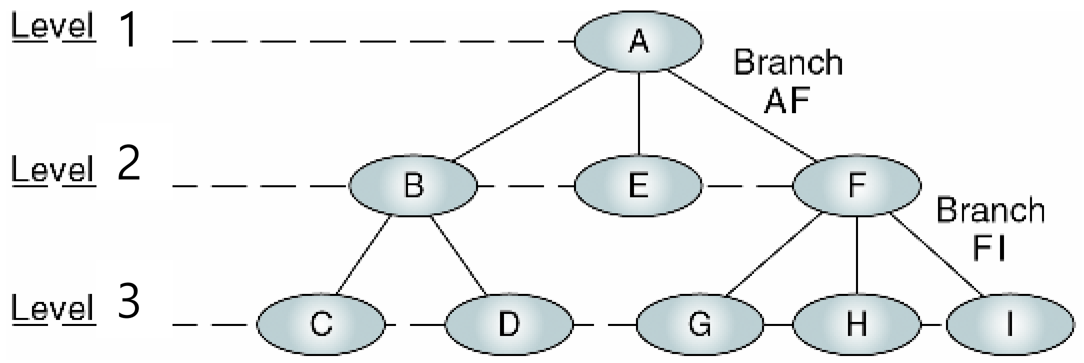
Terminologies

- Leaf node(terminal): node with out-degree zero
- Internal node: not a root or a leaf
- Parent node: A node which is the parent of its successor nodes
- Child node: A node which is the child of its predecessor
- Sibling: nodes with the same parent
- Path: a sequence of nodes in which each node is adjacent to the next one



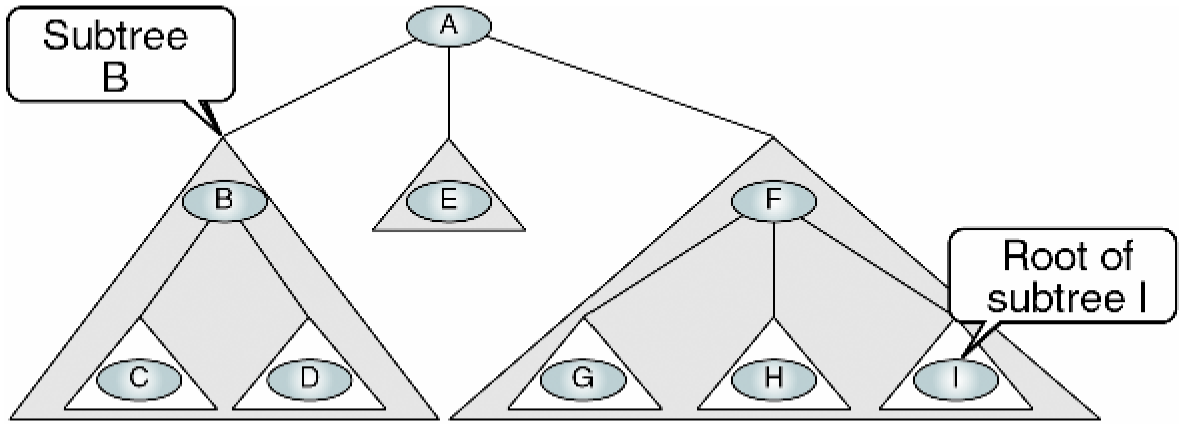
Terminologies

- **Ancestor: any node in the path from the root to the node**
- **Descendent: any node in the path below the parent node**
- **Level: distance from the root**
- **Height (depth) of a tree: the maximum level of any node in the tree**



Terminologies

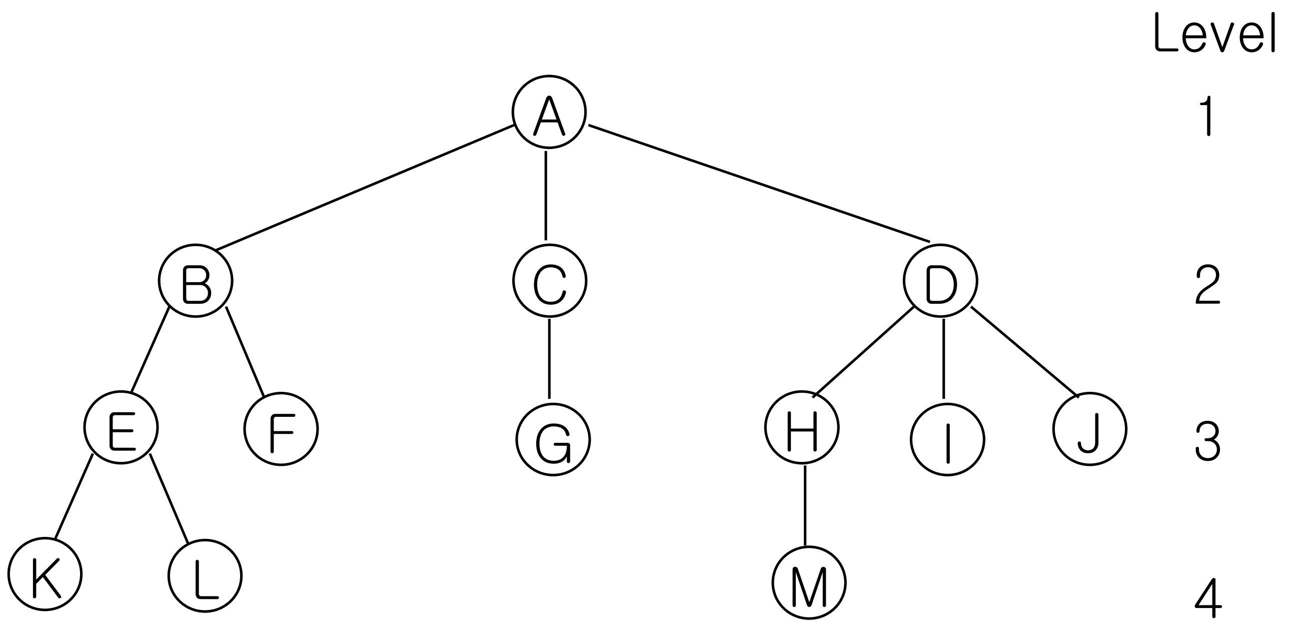
- **Subtree: any connected structure below the root**



Cf. Recursive definition of tree

- Empty or
- Has a designated node, root, from which hierarchically descend zero or more subtrees, which are also trees.

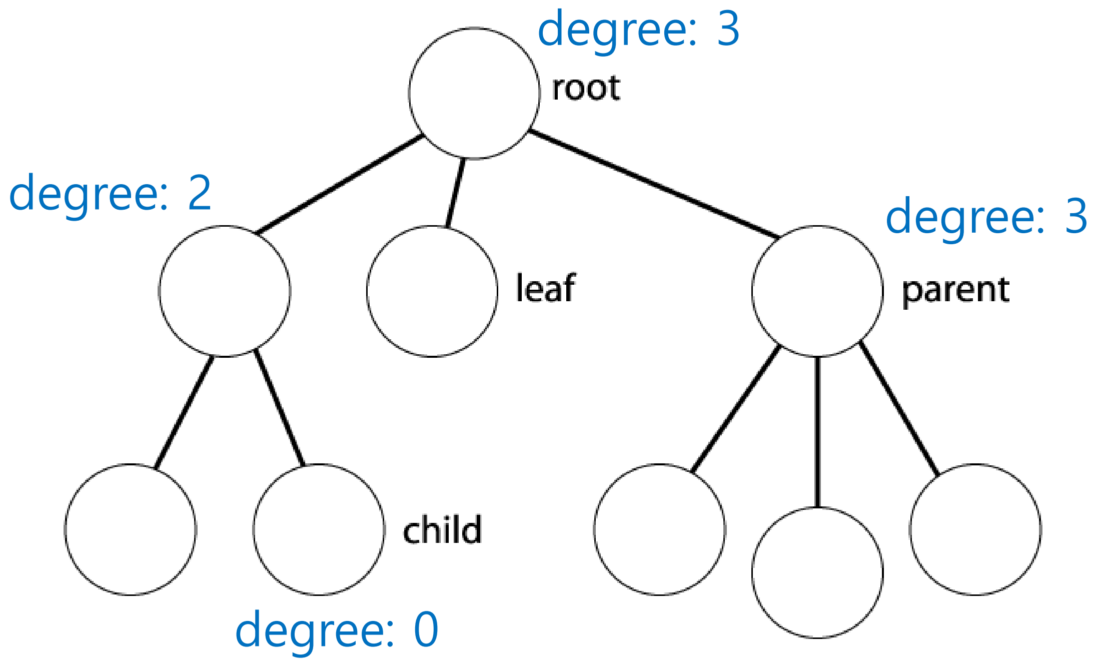
A sample tree



The number of subtrees of a node: degree

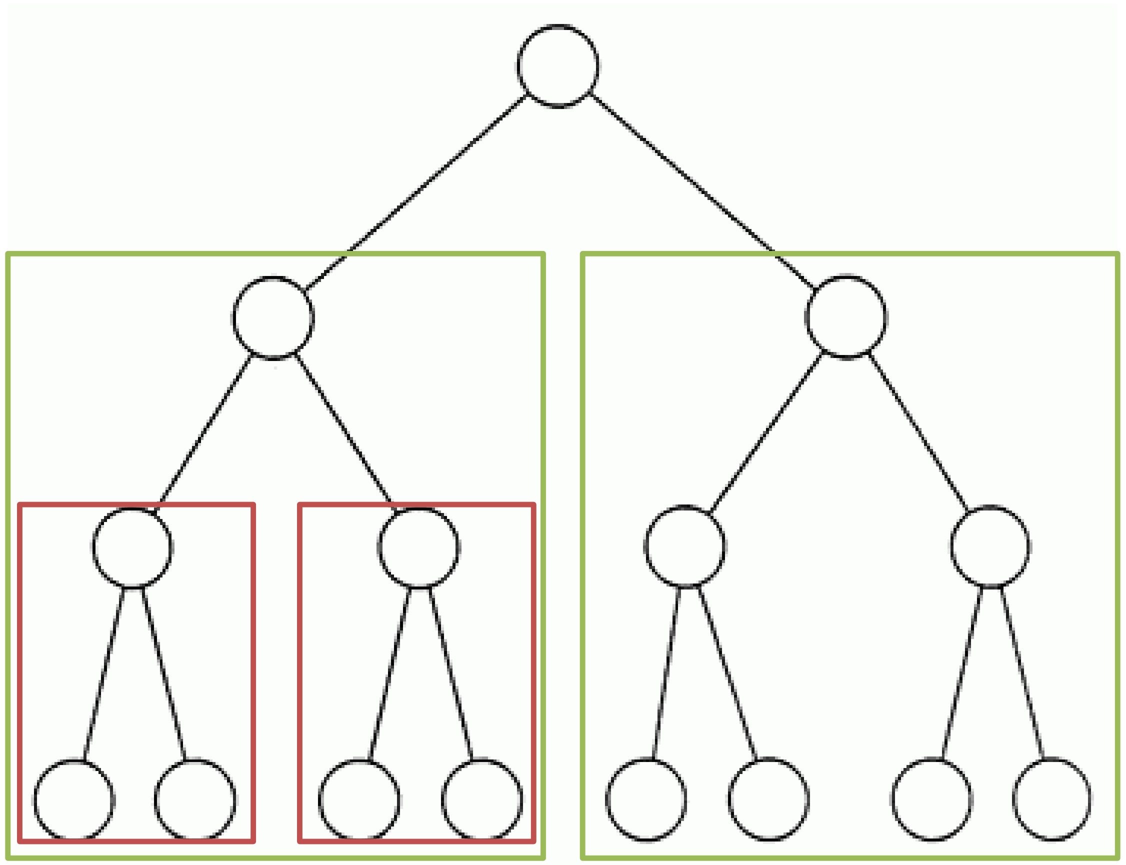
Degree

- The number of subtrees (= the number of children) is the degree of a node

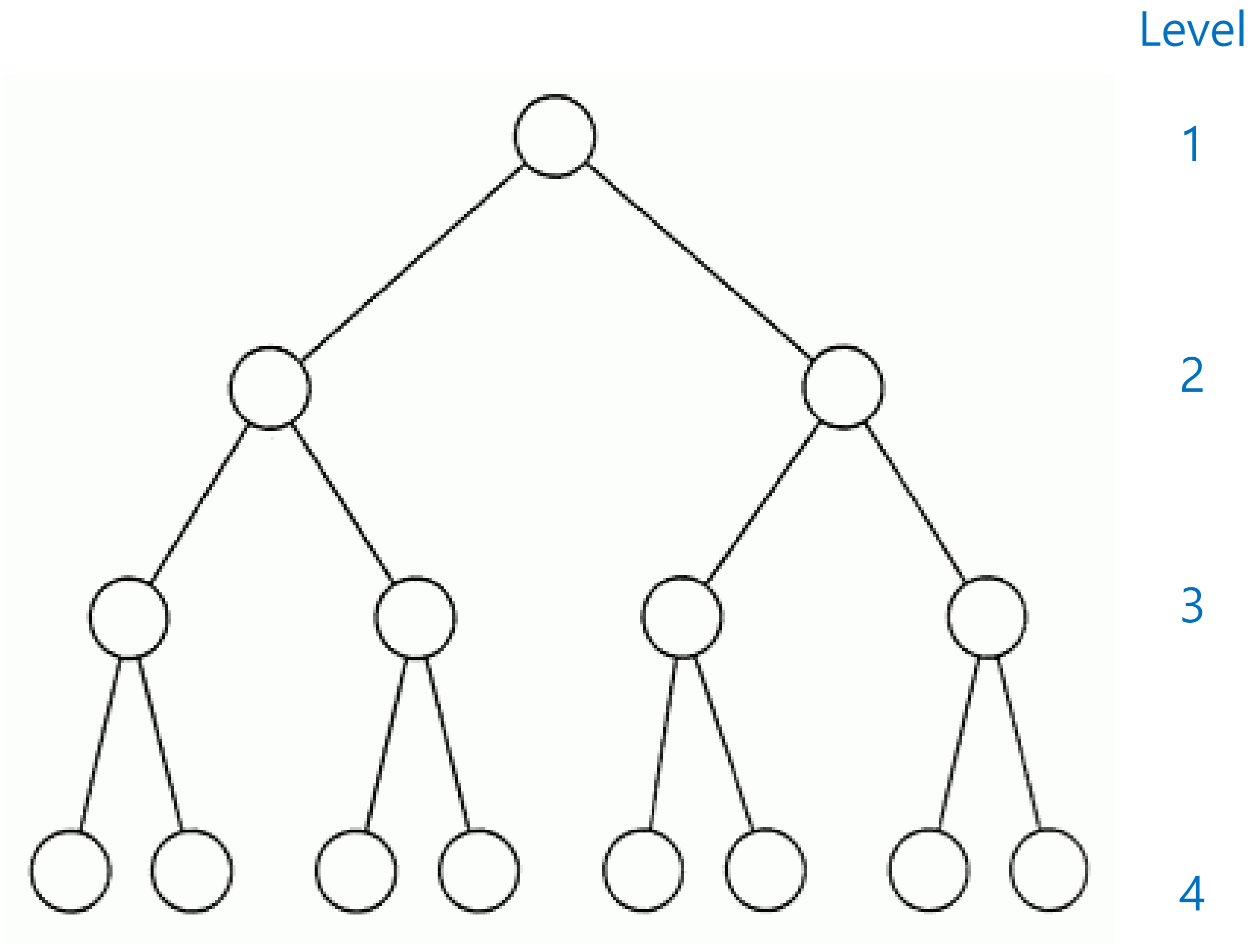


Subtree

- Subtrees of root = green boxes



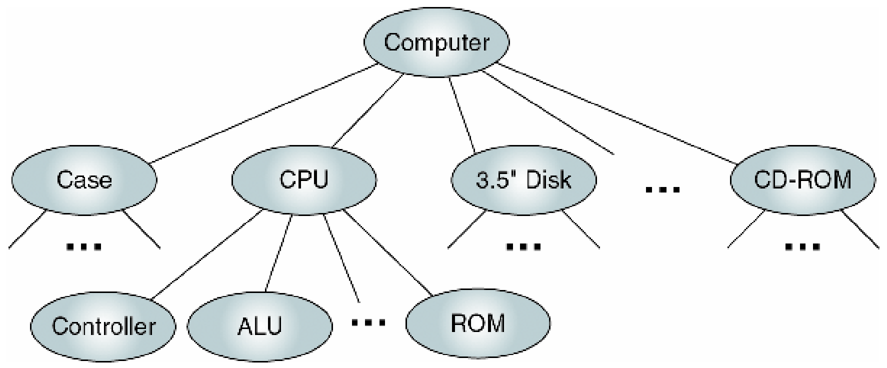
Level and Depth



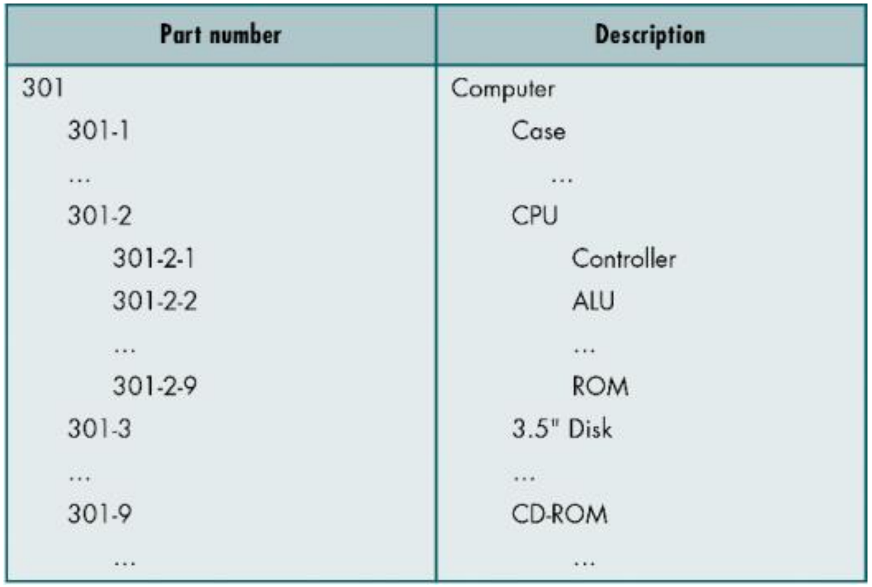
▶ Depth (=height): the maximum level of a tree

User Representations for Trees

- **General tree**



- Indented list

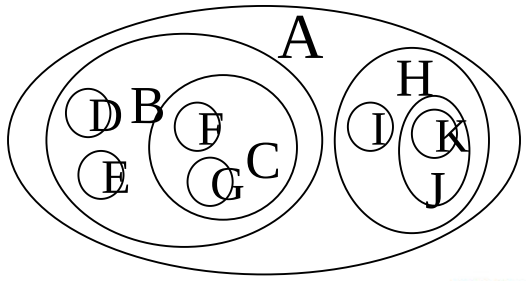


Part number	Description
301	Computer
301-1	Case
...	...
301-2	CPU
301-2-1	Controller
301-2-2	ALU
...	...
301-2-9	ROM
301-3	3.5" Disk
...	...
301-9	CD-ROM
...	...

- **Parenthetical listing**
  - Open parenthesis indicates the start of a new level

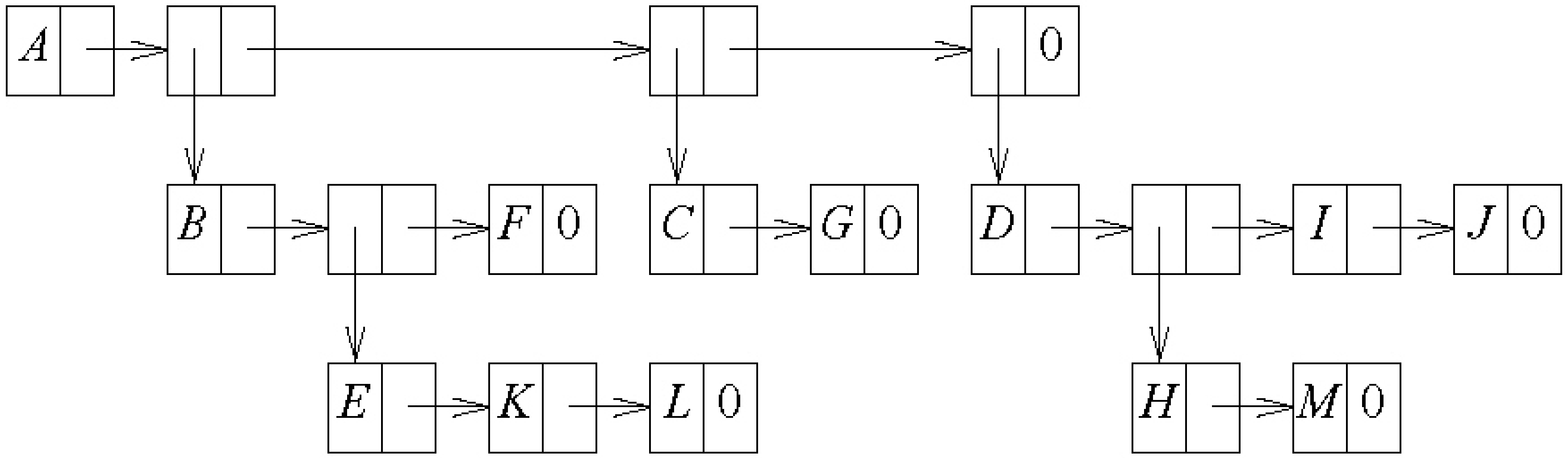
Ex) (A (B (C D) E F (G H I))

- Venn diagram



Representation of Trees

■ **List Representation of the tree**
(A(B(E(K,L),F),C(G),D(H(M),I,J)))



- Using this node structure is very wasteful of space.
- Lemma 5.1: If T is a k-ary tree(i.e., a tree of degree k) with n nodes, each having a fixed size as below, then n(k-1)+1 of the nk child fields are 0, n>=1.

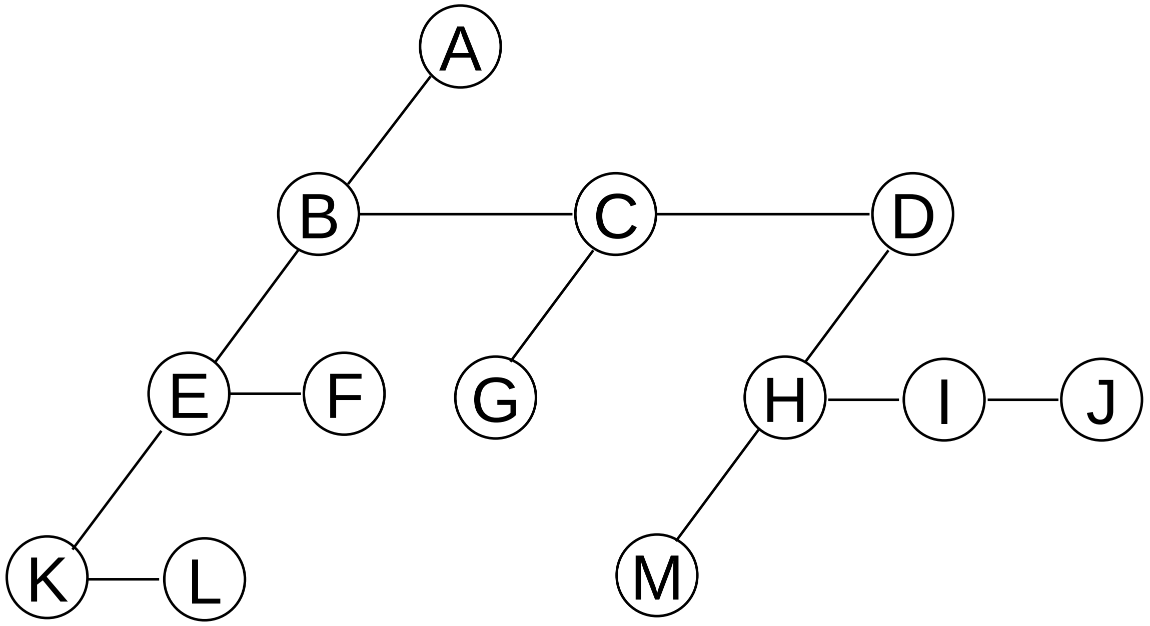
DATA	CHILD 1	CHILD 2	···	CHILD *k*

Left Child-Right Sibling Representation

■ **Node structure**

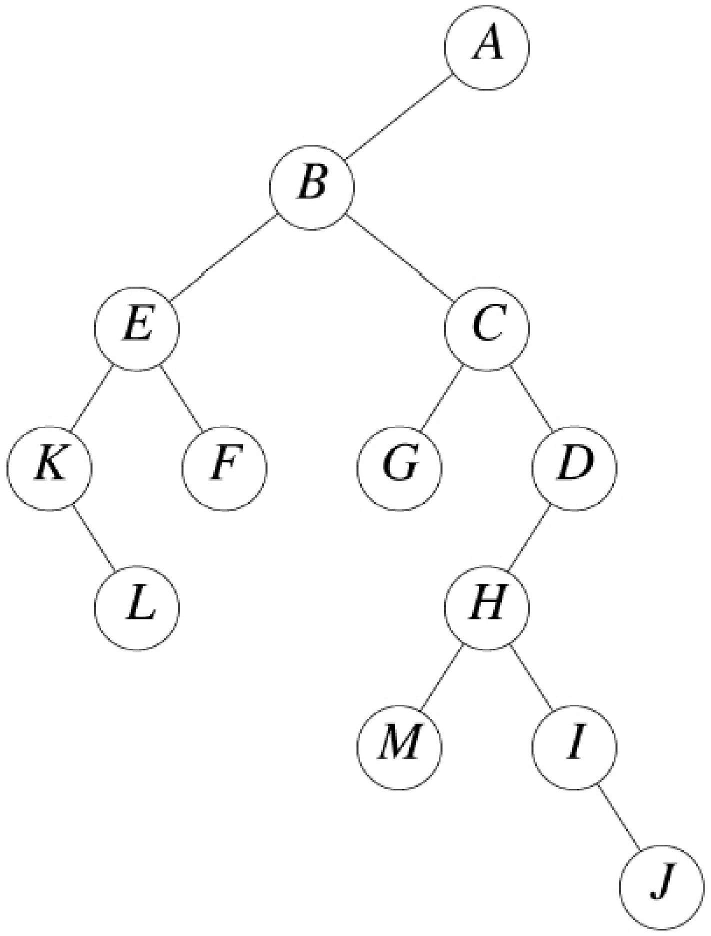
data	
left child	right sibling

■ **Tree**



Representation as a Degree-Two Tree

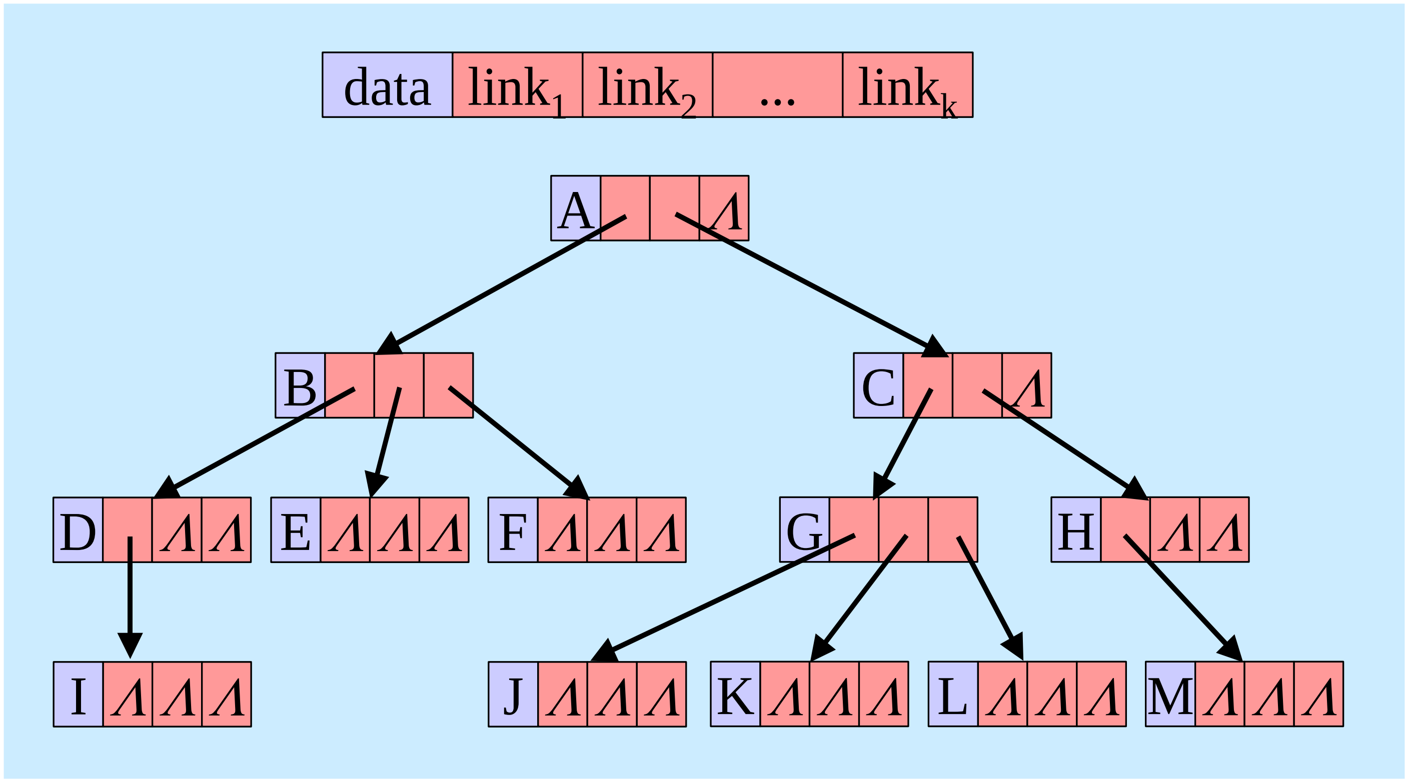
- **The degree-two tree representation of a tree**
  - Rotate the right-sibling pointers in a left child-right sibling tree clockwise by 45°
  - The right child of the root node of the tree is empty.



- **binary tree**
  - Left child-right child trees
  - degree two trees

트리(tree)의 표현

- 괄호 : (A(B(D(I),E,F),C(G(J,K,L),H(M))))
- 컴퓨터상의 표현 : 데이터와 최대 차수만큼의 링크 필드

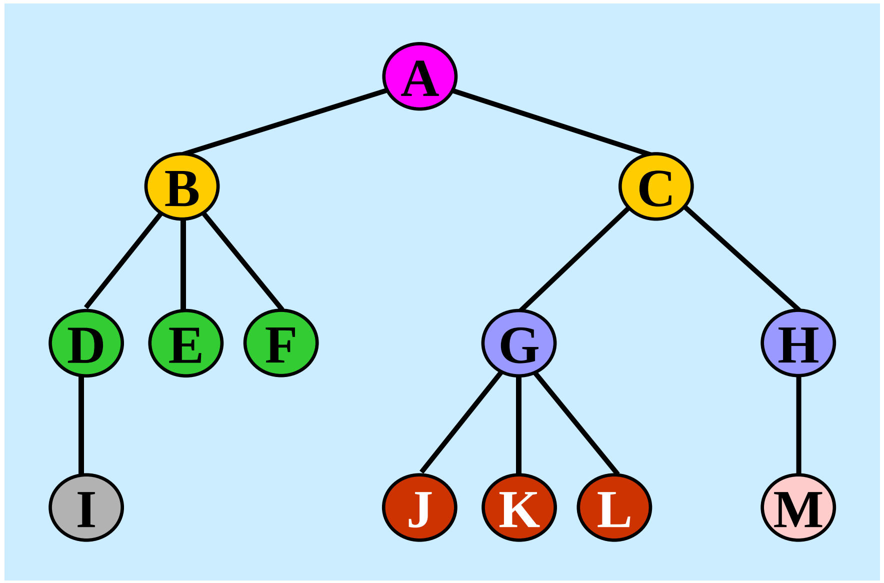


k차 트리의 2차 트리(이진 트리) 표현

- k 차 트리의 null 링크 = $nk - (n-1) = n(k-1) + 1$
  - k 가 클수록 메모리 낭비가 큼
    - 노드 수 : n, 전체 링크의 수 : n·k
    - Null이 아닌 링크 수: n - 1
  - k 차 트리를 이진 트리로 변환해서 표현
    - 모든 k 차 트리는 이진 트리로 변환 가능
    - 이진트리의 경우, 각 노드는 최대 2개의 자식 노드를 가짐
- k 차 트리의 이진 트리 변환 방법
  - 왼쪽 링크 : k 개의 자식노드 첫 번째 자식 노드를 포인트
  - 오른쪽 링크 : 오른쪽 첫 번째 형제 노드를 포인트

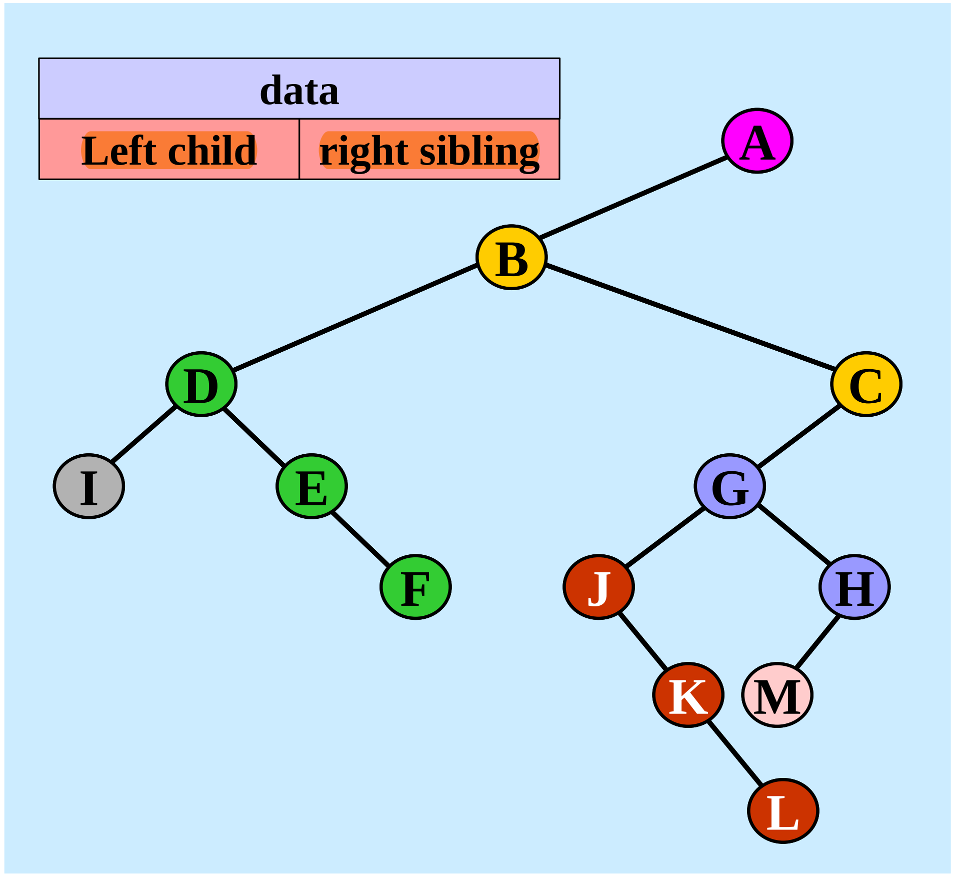
k차 트리의 2차 트리(이진 트리) 표현

- k차 트리 표현



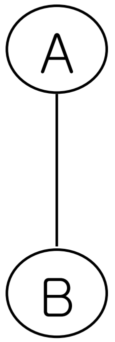
왼쪽 링크 : k 개의 자식노드 첫 번째 자식 노드를 포인트
오른쪽 링크 : 오른쪽 첫 번째 형제 노드를 포인트

- 이진 트리 표현

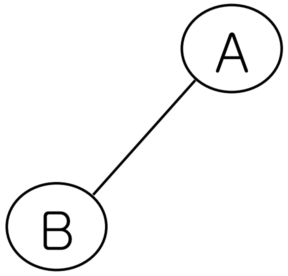


k차 트리 -> 이진 트리 로 표현하면 더 compact 하게 표현할 수 있다.

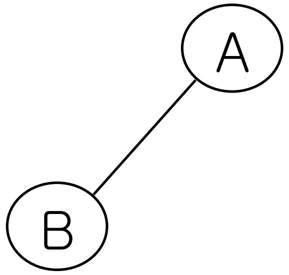
Tree representation



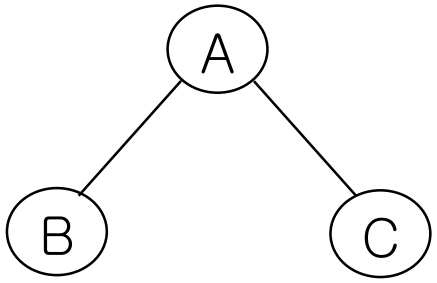
tree



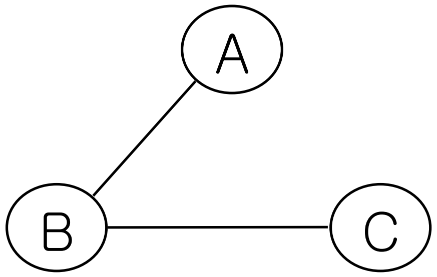
Left child-right sibling tree



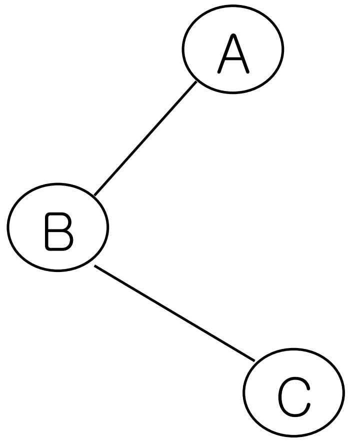
Binary tree



tree



Left child-right sibling tree



Binary tree

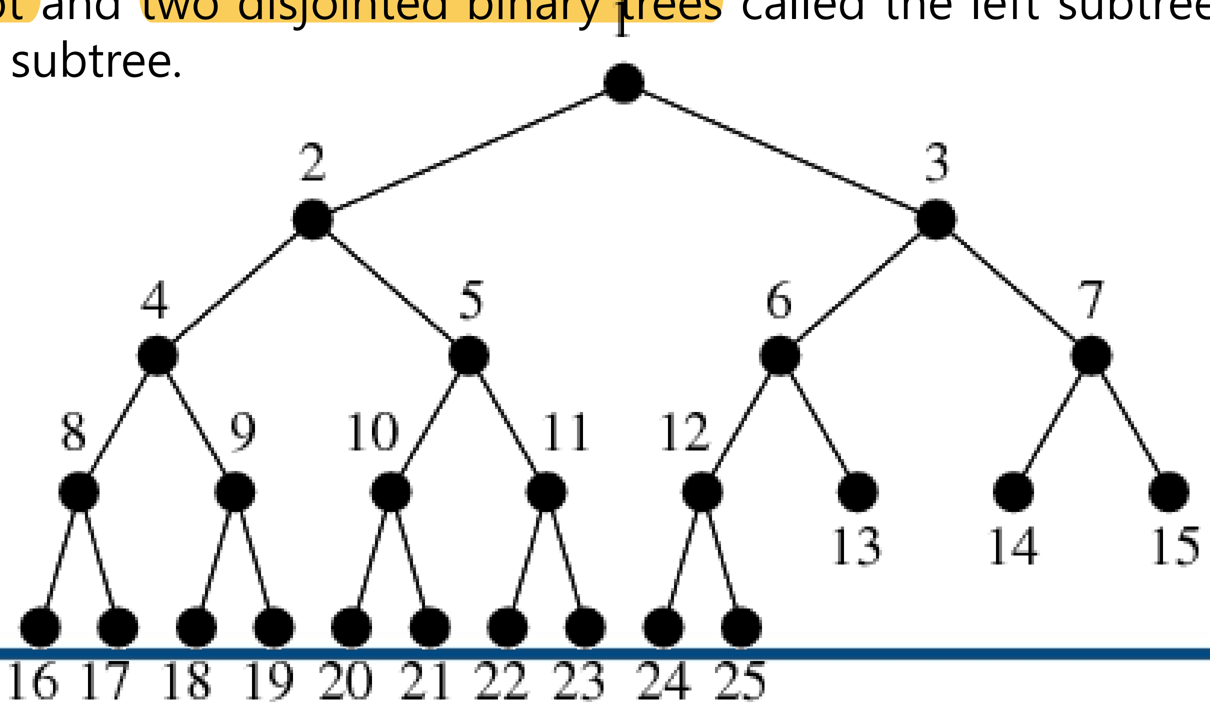
Abstract data type Binary_Tree

```
ADT Binary_Tree(abbreviated BinTree)
    object: a finite set of nodes either empty or consisting of a root node,
            left Binary_Tree and right Binary_Tree
    functions:
      for all bt, bt1, bt2 ∈ BinTree, item ∈ element
      BinTree Create()                   ::= creates an empty binary tree
      Boolean IsEmpty(bt)                ::= if (bt == empty binary tree)
                                              return TRUE else return FALSE
      BinTree MakeBT(bt1, item, bt2)     ::= return a binary tree whose left subtree
                                             bt1, whose right subtree is bt2, and
                                             whose root node contains the data
                                             item
      BinTree Lchild(bt)                 ::= if(IsEmpty(bt)) return error
                                              else return the left subtree of bt
      element Data(bt)                   ::= if(IsEmpty(bt)) return error
                                              else return the data in the root node
                                             of bt
      BinTree Rchild(bt)                 ::= if(IsEmpty(bt)) return error
                                              else return the right subtree of bt
```

Binary Trees (1)

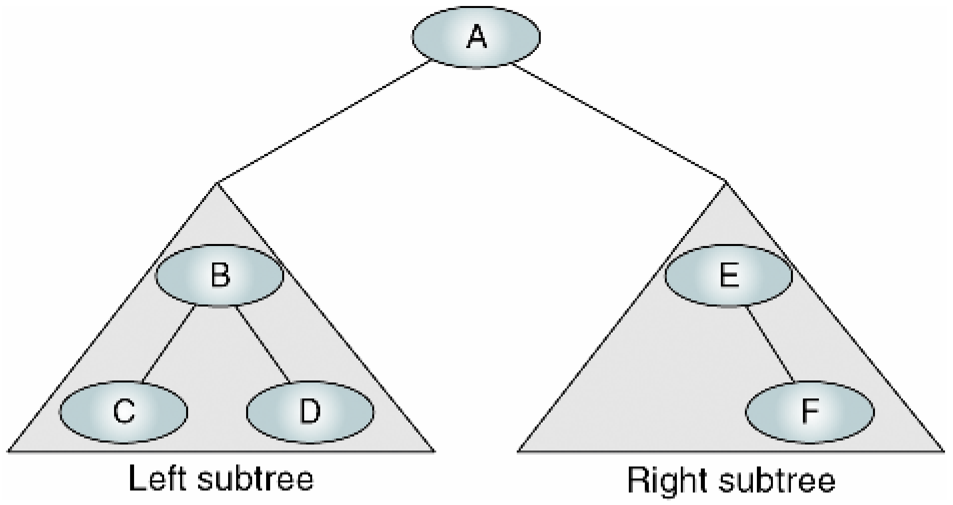
정의를 알아야한다.

- **Binary Tree**
  - The degree of any given node must not exceed two. =>At most 2 children
  - It is distinguished between the left subtree and the right subtree.
  - It may have zero nodes.
- **Definition of Binary Trees**
  - A binary tree is a finite set of nodes that is either empty or consists of a root and two disjointed binary trees called the left subtree and the right subtree.



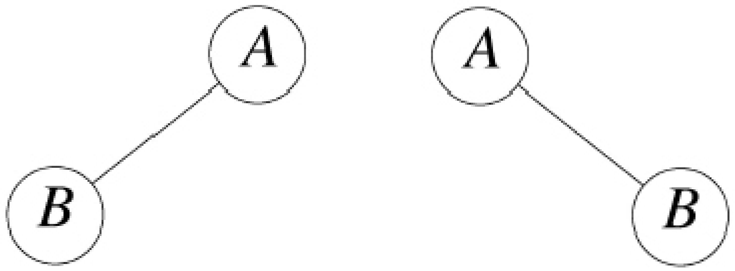
Binary Trees (2)

- **Binary tree: a tree in which no node can have more than two subtrees**
  - Left/right subtrees are also binary trees

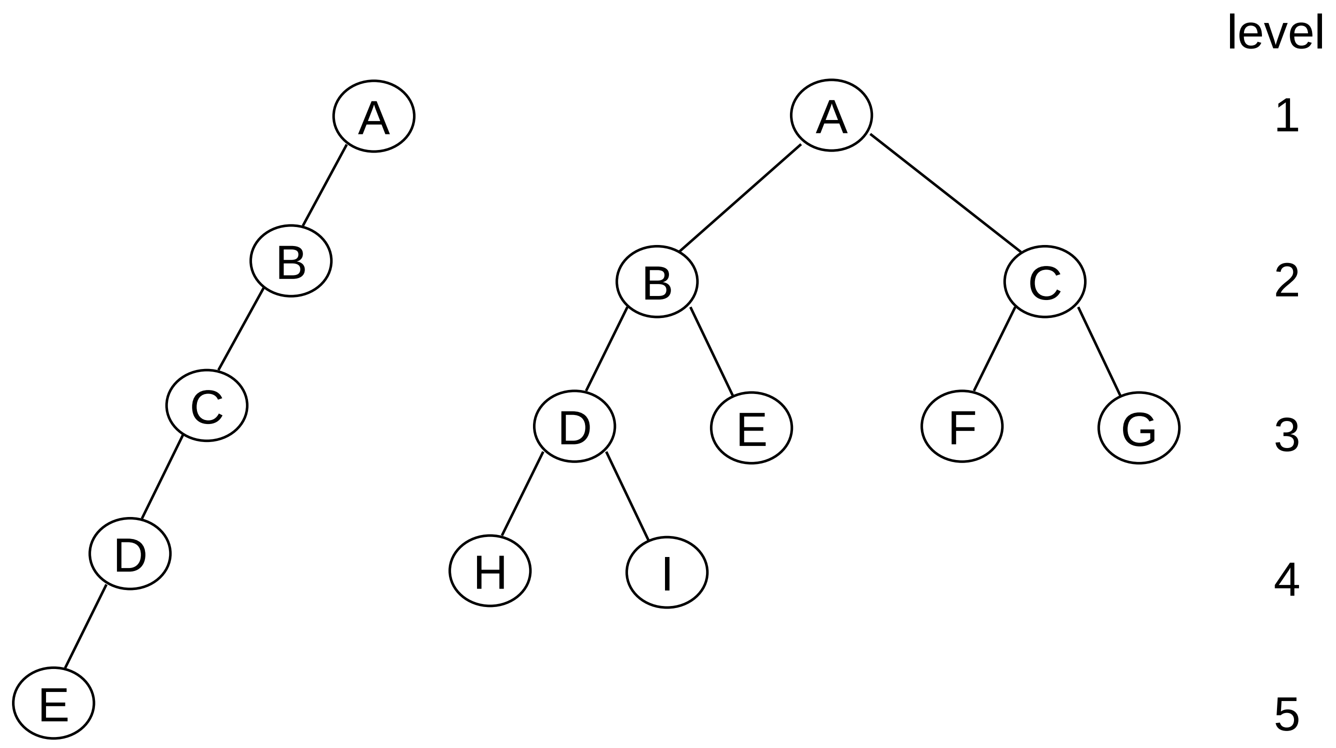


Binary Trees (3)

- **Different points between a binary tree and a tree**
  - There is an empty binary tree, but there is no tree having zero nodes.
  - In binary tree, we distinguish between the order of the children while in a tree we do not.
    - Two different binary trees

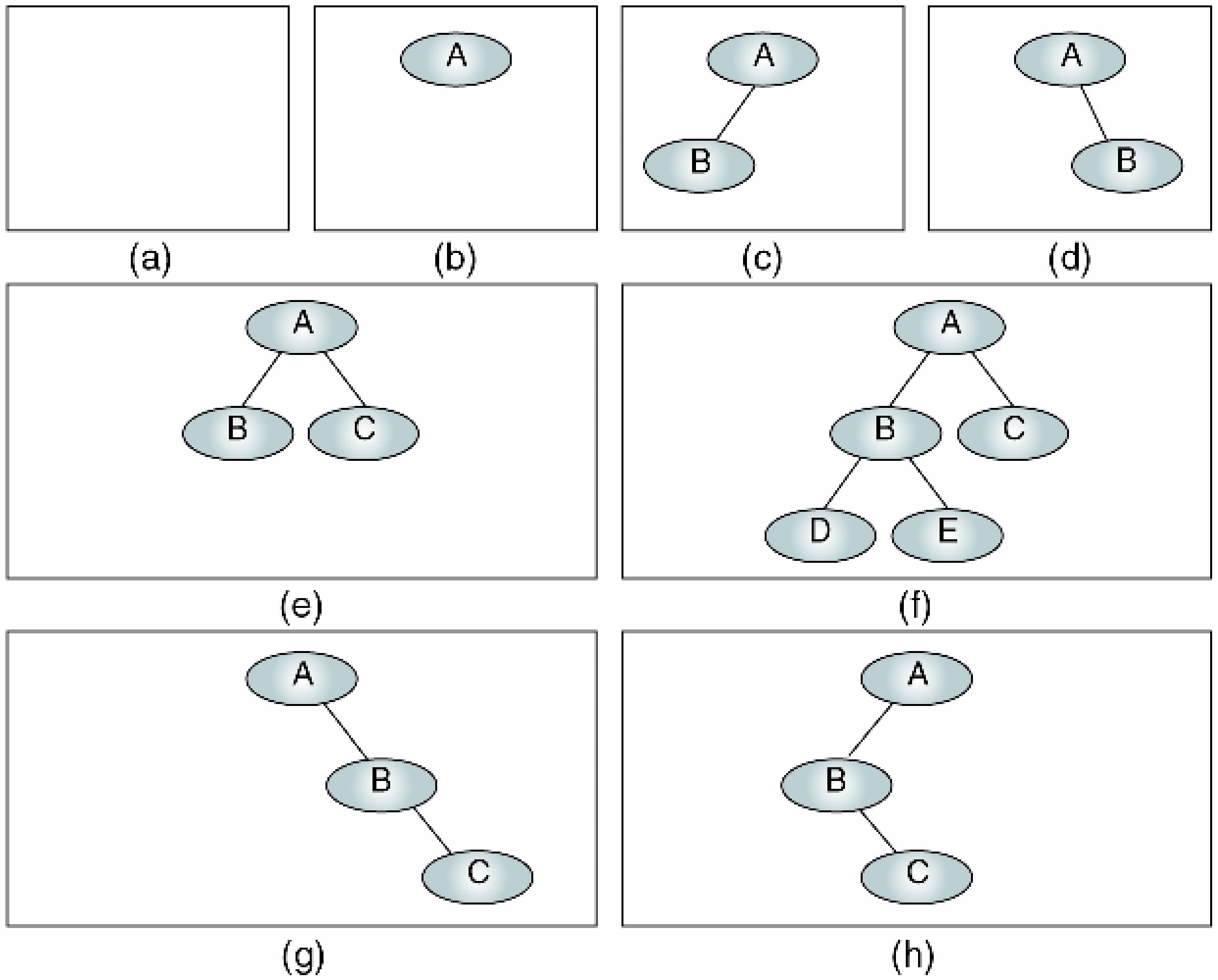


Binary Trees (4)



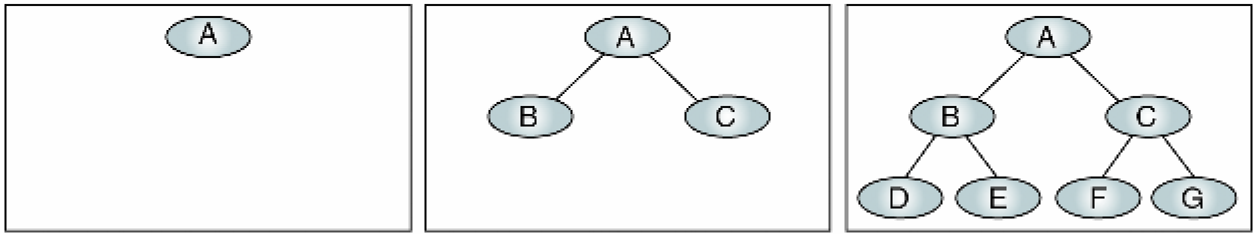
(a) Skewed binary tree (b) Complete binary tree

Examples of binary trees

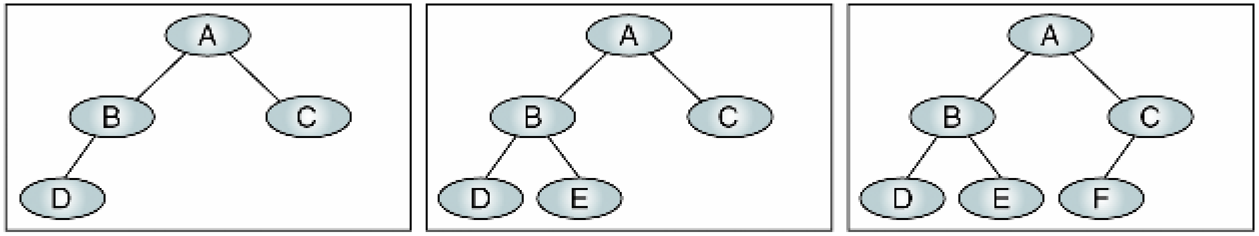


Examples of binary trees

- **Full binary tree (proper binary tree): a tree with the maximum # of entries for its height**



- **Complete binary tree: a tree that has minimum height for its nodes and all nodes in the last level are found on the left**



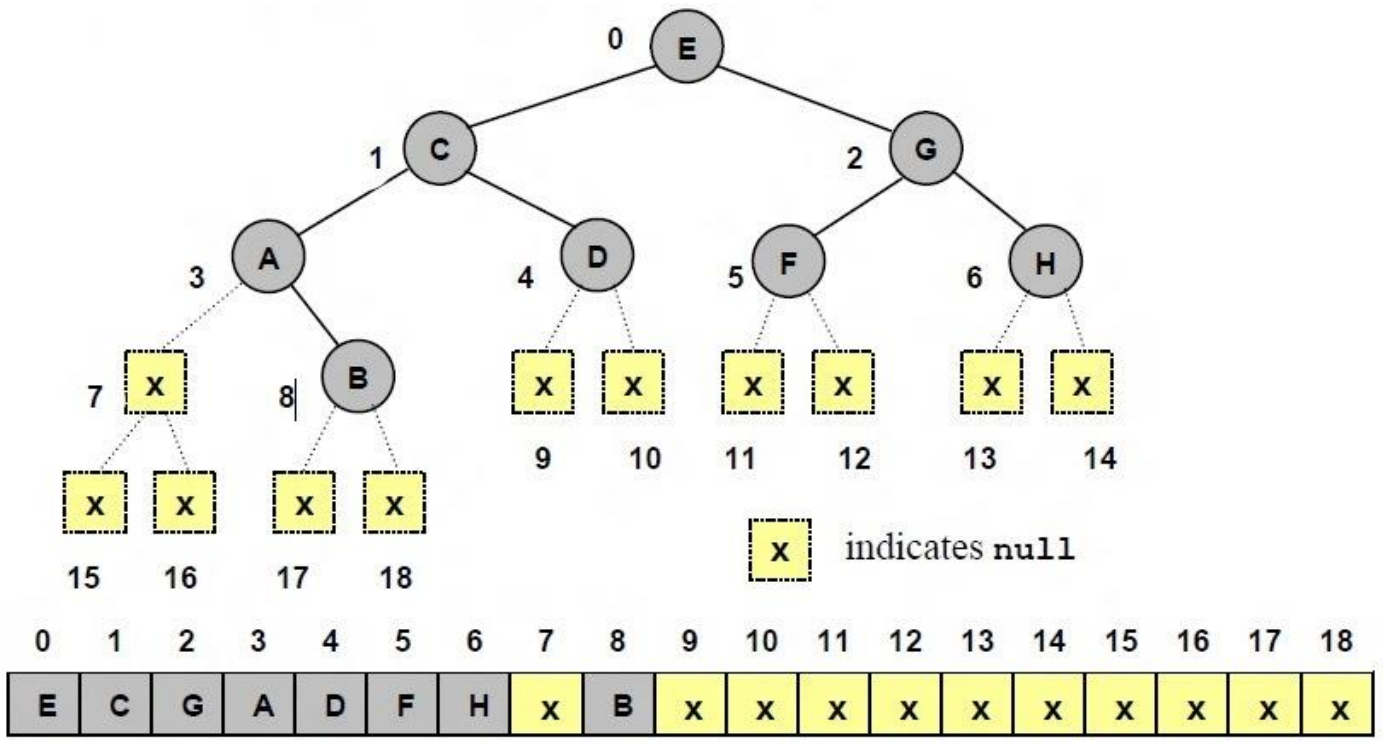
Binary Tree Representation(1)

■ Array Representation

- ▶ Using one-dimensional array to store the nodes
- ▶ Lemma 5.4
  - If a complete binary tree with n nodes is represented sequentially, then any node with index i, $1 \le i \le n$, we have
    ① *parent*(i) : $\lfloor i/2 \rfloor$ **if** $i \neq 1$
    ② *leftChild*(i) : $2i$ **if** $2i \le n$
    i has no left child **if** $2i > n$
    ③ *rightChild*(i) : $2i+1$ **if** $2i+1 \le n$
    i has no right child **if** $2i + 1 > n$
- ▶ For complete binary tree: ideal, no space is wasted.
- ▶ For a skewed tree: less than half the array is utilized.
  - In the worst case, a skewed tree of depth k will required 2^k-1 spaces. Only k will be used.

Binary Tree Representation(2)

- **Array Representation**
  - Level-by-level
  - Inefficient (Lots of empty spaces)

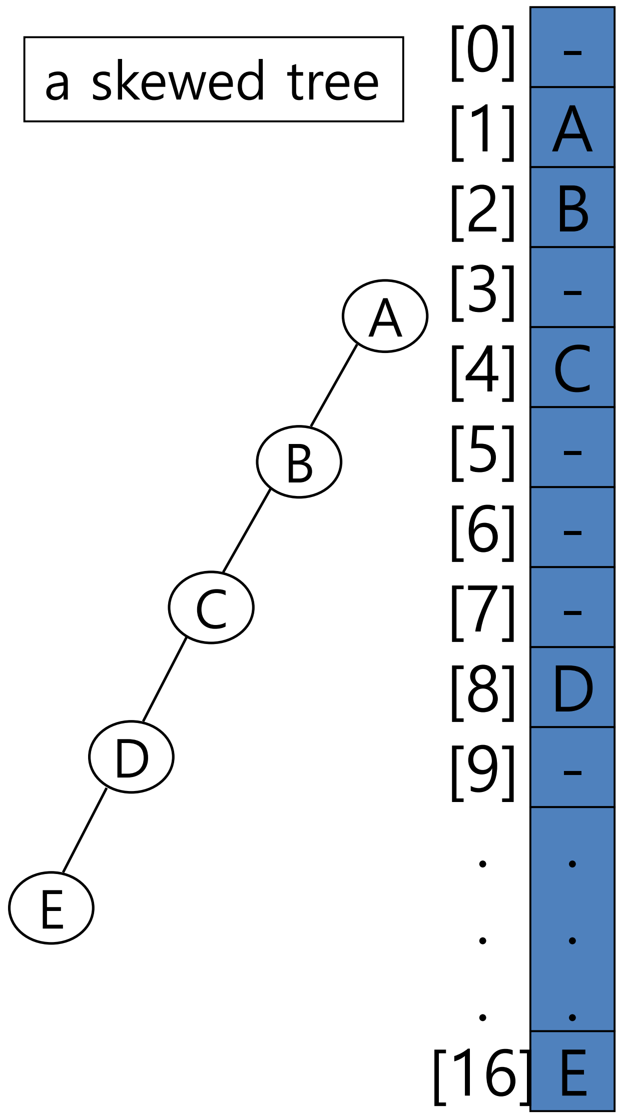


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
E	C	G	A	D	F	H	x	B	x	x	x	x	x	x	x	x	x	x

Array representation of a binary tree

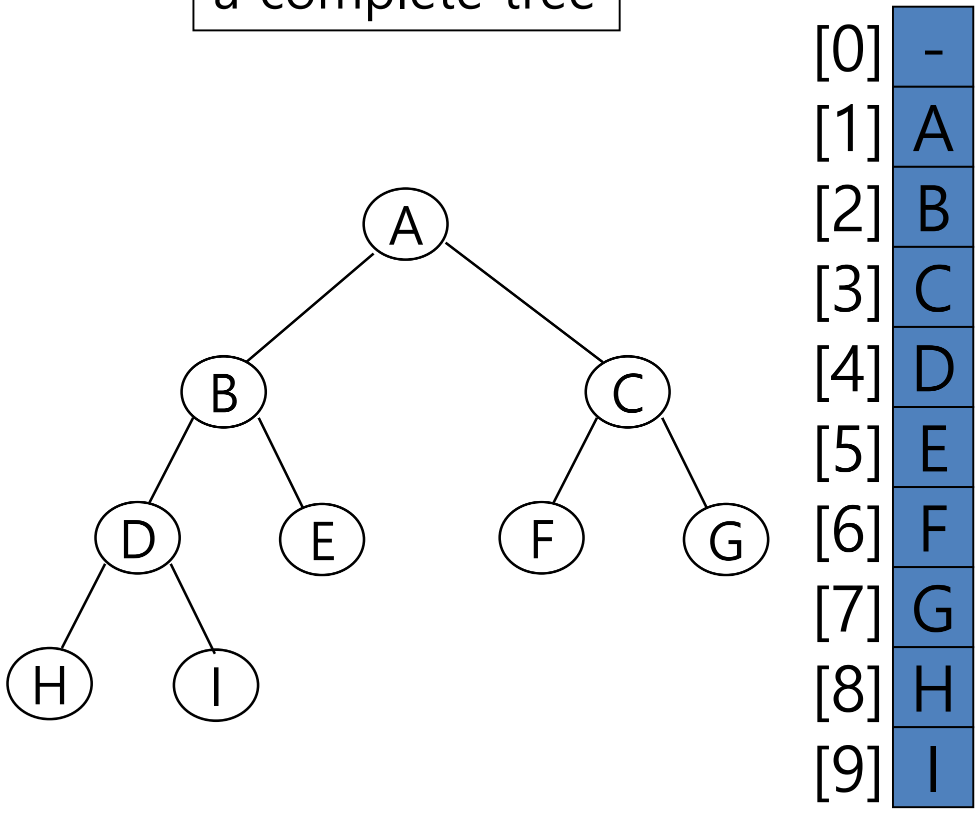
Binary Tree Representation(3)

a skewed tree



[0]	-
[1]	A
[2]	B
[3]	-
[4]	C
[5]	-
[6]	-
[7]	-
[8]	D
[9]	-
.	.
.	.
.	.
[16]	E

a complete tree



[0]	-
[1]	A
[2]	B
[3]	C
[4]	D
[5]	E
[6]	F
[7]	G
[8]	H
[9]	I

Binary Tree Representation(4)

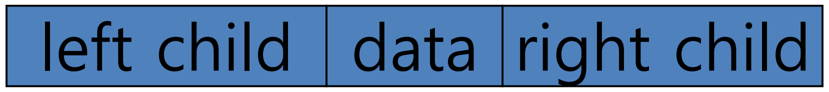
- **Disadvantages of array representation**
  - It is good only for complete binary trees.
  - It suffers from the general inadequacies of sequential representation.
  - Insertion and deletion of nodes from the middle of a tree require the movement of potentially many nodes to reflect the change in level number of these nodes.

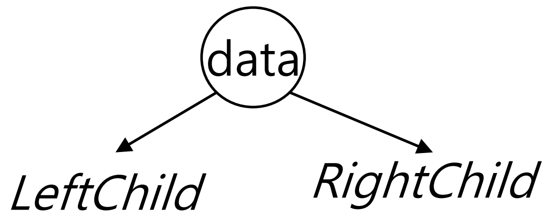
- **To overcome these problems, the use of a linked representation can be easily used.**

Binary Tree Representation(5)

Linked Representation

- Node



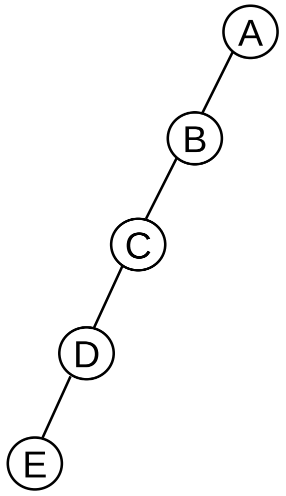


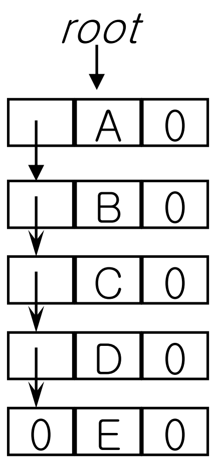
```
typedef struct node * treePointer;
typedef struct node{
            int data;
            treePointer leftChild, rightChild;
            };
```

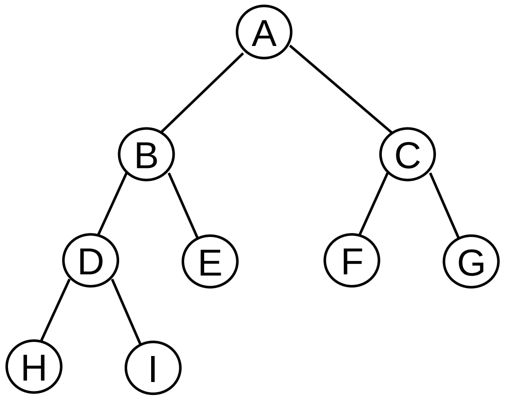
- It is difficult to determine the parent of a node
  - a fourth field, *parent may be included in the class, treePointer.*

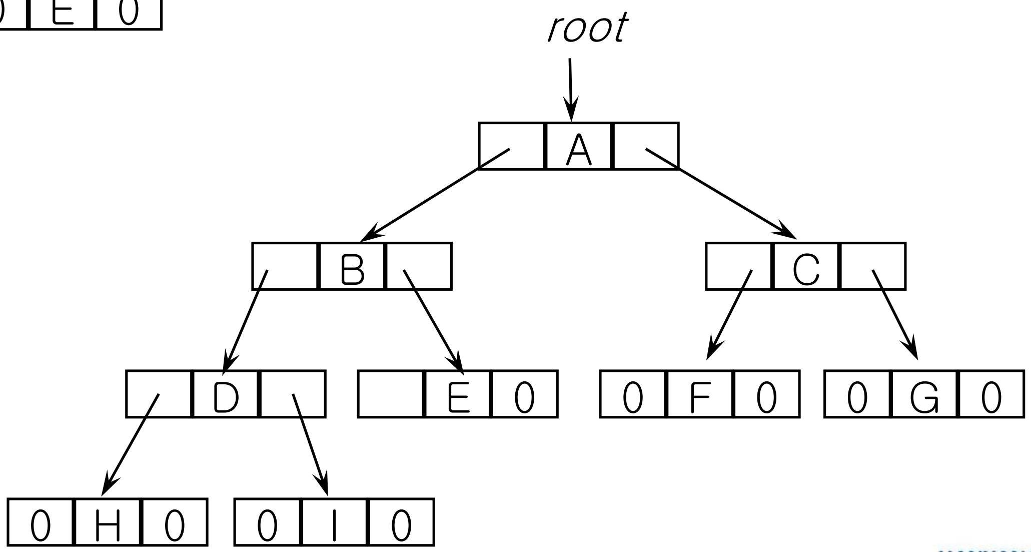
Binary Tree Representation(6)

◆ **Example of linked representation**









Properties of Binary Trees (1)

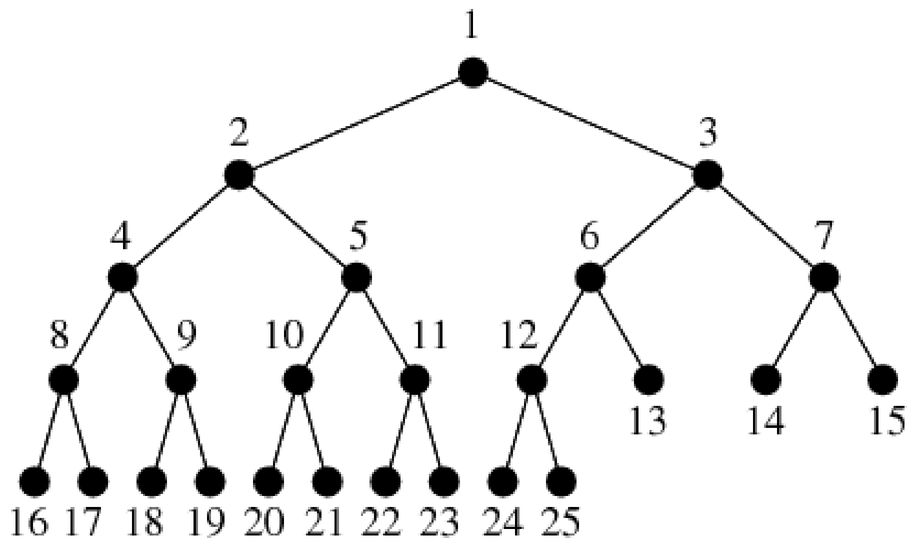
■ **Lemma 5.2 [Maximum number of nodes]**

(1) The maximum number of nodes on level i of a binary tree is

$2^{i-1}(i \geq 1)$

(2) The maximum number of nodes in a binary tree of depth k is 2^k-1(k ≥ 1)

$$\sum_{i=1}^{k}(\text{The maximum number of nodes in level i.}) = \sum_{i=1}^{k} 2^{i-1} = 2^k - 1$$



Properties of Binary Trees (2)

- **Lemma 5.3 [Relation between number of leaf nodes and degree-2 nodes]**

  For any nonempty binary tree, T,

  if

  n : The number of nodes in a tree

  n_0 : The number of leaves

  n_1 : The number of nodes with one child

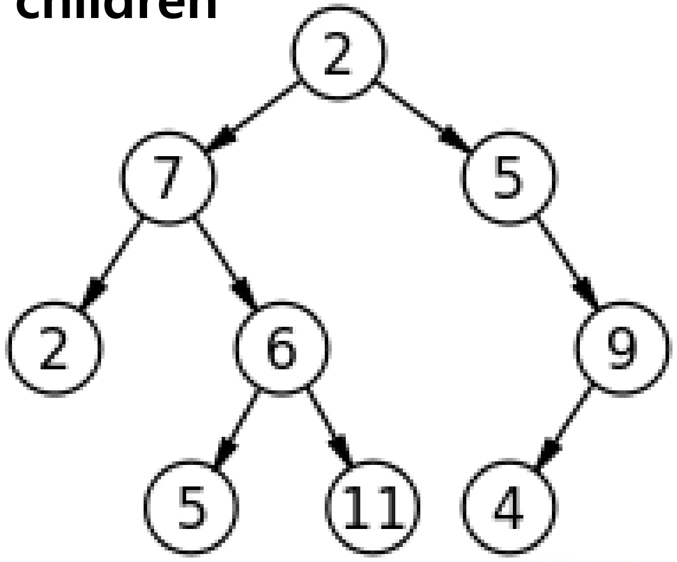
  n_2 : The number of nodes with two children

  E : The number of edges

  then

  $$n = n_0 + n_1 + n_2$$

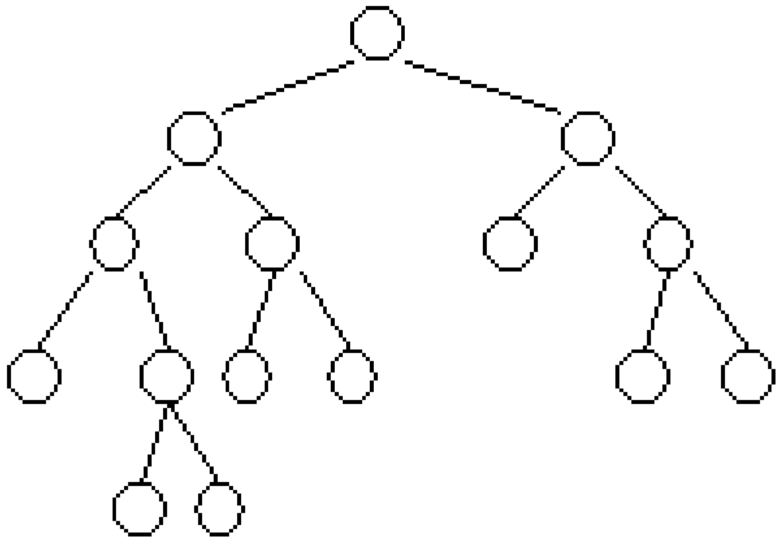
- **Why?**
  - $n = E + 1 = n_1 + 2\, n_2 + 1$
  - Therefore, $n_0 = n_2 + 1$



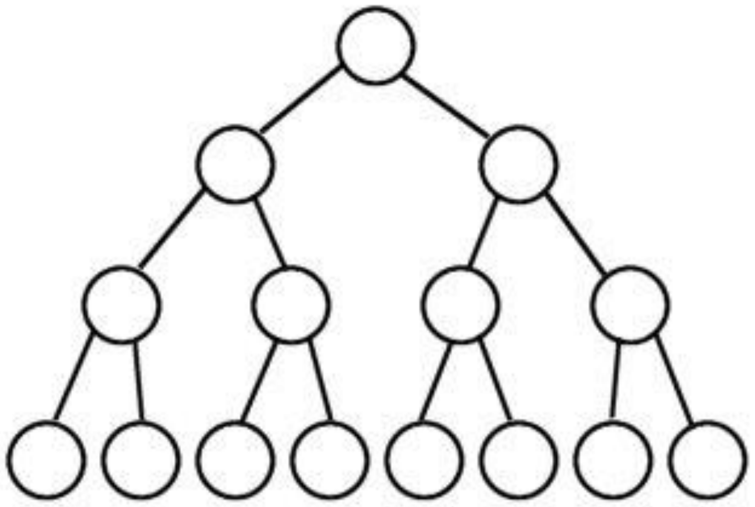
Special cases of binary trees

- Strict binary tree: 0 or 2 children for every node
- Full binary tree: All levels are completely filled

Strict binary tree

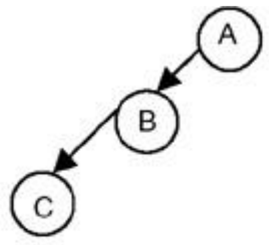


Full Binary Tree

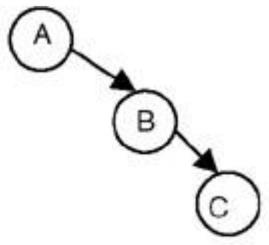


Special cases of binary trees

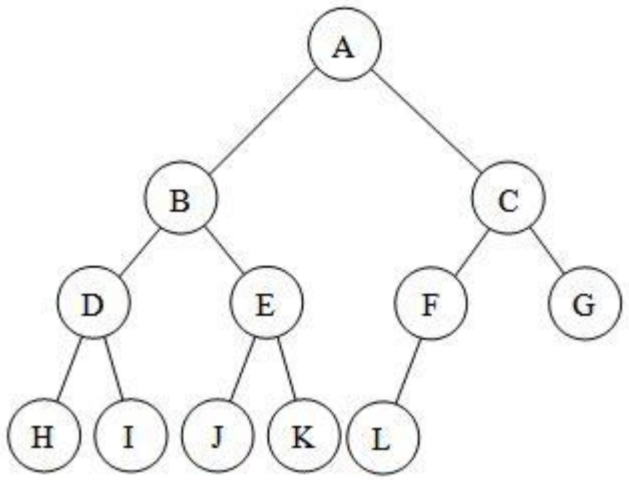
- **Skewed binary tree**
- **Complete binary tree**
  - All levels except the last are filled
  - The nodes are positioned as far <u>left</u> as possible



Left Skewed



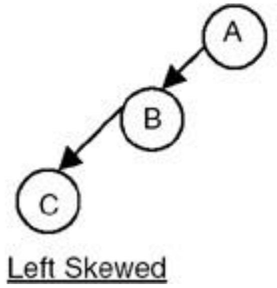
Right Skewed

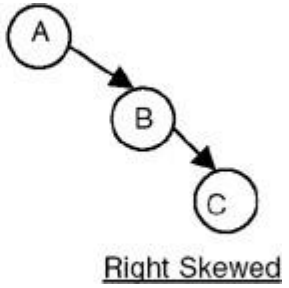


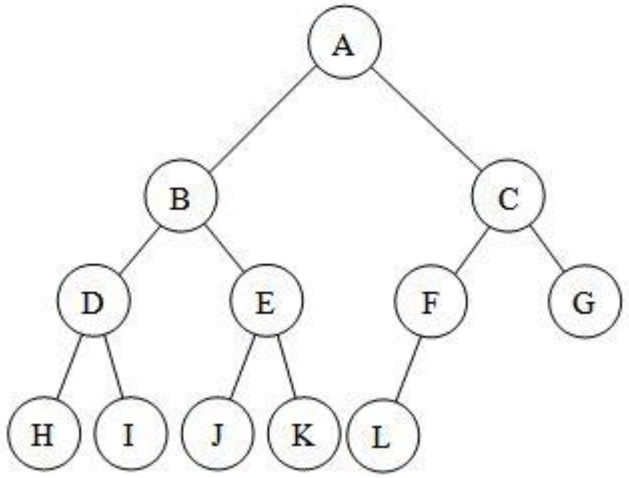
complete

More relations

- **n : the number of nodes** (k for depth, usually)
- **Maximum depth : n**
  - = depth of a skewed tree
- **Minimum depth :** $\lceil \log_2(n+1) \rceil$
  - = depth of a complete tree



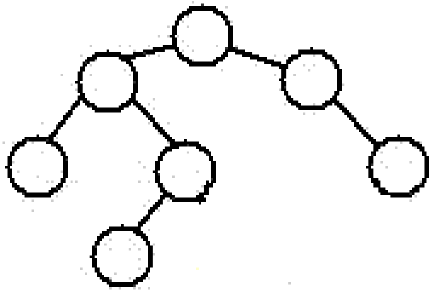


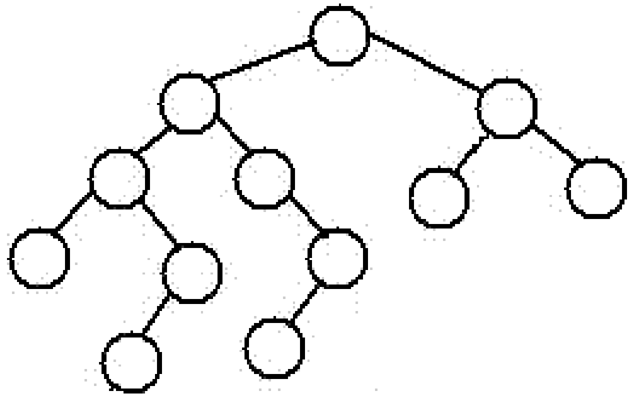


complete

Balanced Binary Trees(1)

- The difference between the heights of every two subtrees is less than 1

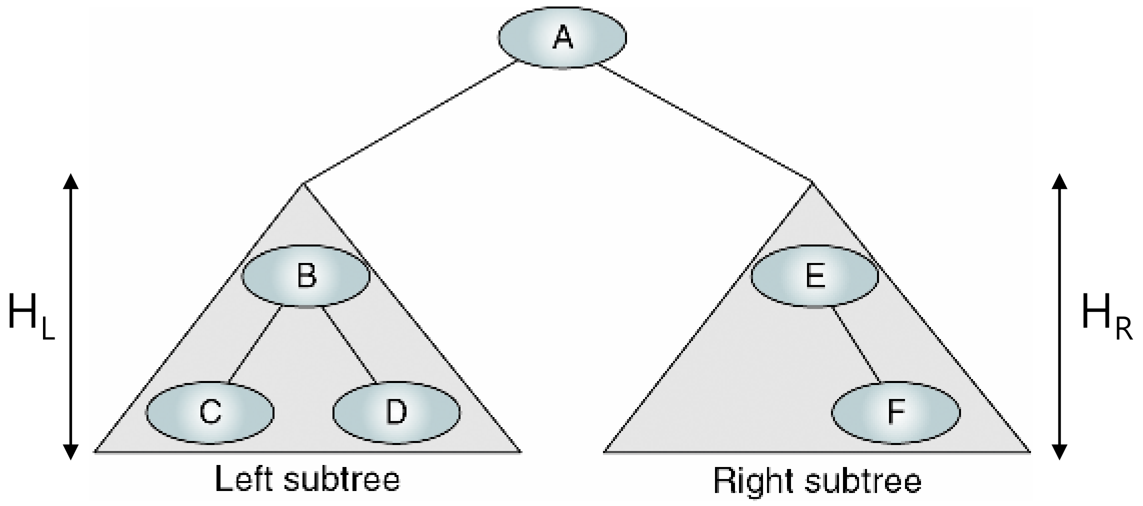

A height-balanced Tree


Not a height-balanced tree

▶ Balanced trees are efficient in data processing

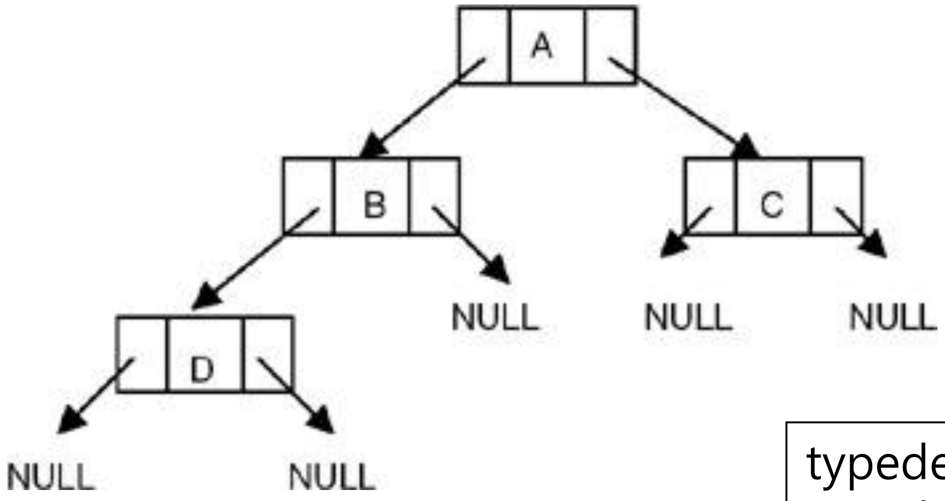
Balanced Binary Trees(2)

- **Balance**
  - Balance factor (B) = $H_L - H_R$
  - Balanced binary tree: $|B| \le 1$



Linked representation

- Each node has two link fields
  - Left child and right child



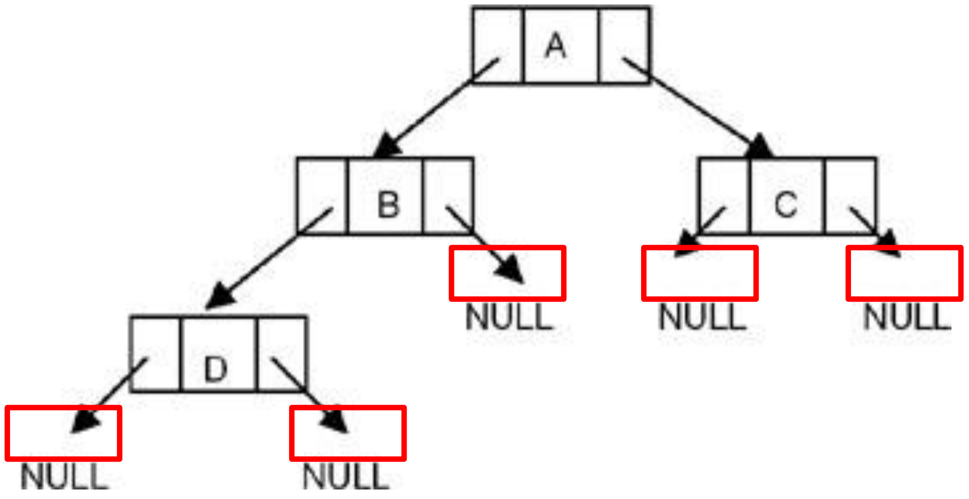
```
typedef struct node * treePointer;
typedef struct node{
          int data;
          treePointer leftChild, rightChild;
          };
```

Linked representation – Program

```
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>

typedef struct TreeNode {
            int data;
            struct TreeNode *left, *right;
} TreeNode;

//      n1
//     /  |
//  n2   n3

void  main()
{
  TreeNode *n1, *n2, *n3;

  n1= (TreeNode *)malloc(sizeof(TreeNode));
  n2= (TreeNode *)malloc(sizeof(TreeNode));
  n3= (TreeNode *)malloc(sizeof(TreeNode));
  n1->data = 10;      // the first node setting
  n1->left = n2;
  n1->right = n3;

  n2->data = 20;      // the second node setting
  n2->right = NULL;

  n3->data = 30;      // the third node setting
  n3->right = NULL;
}
```

Linked representation

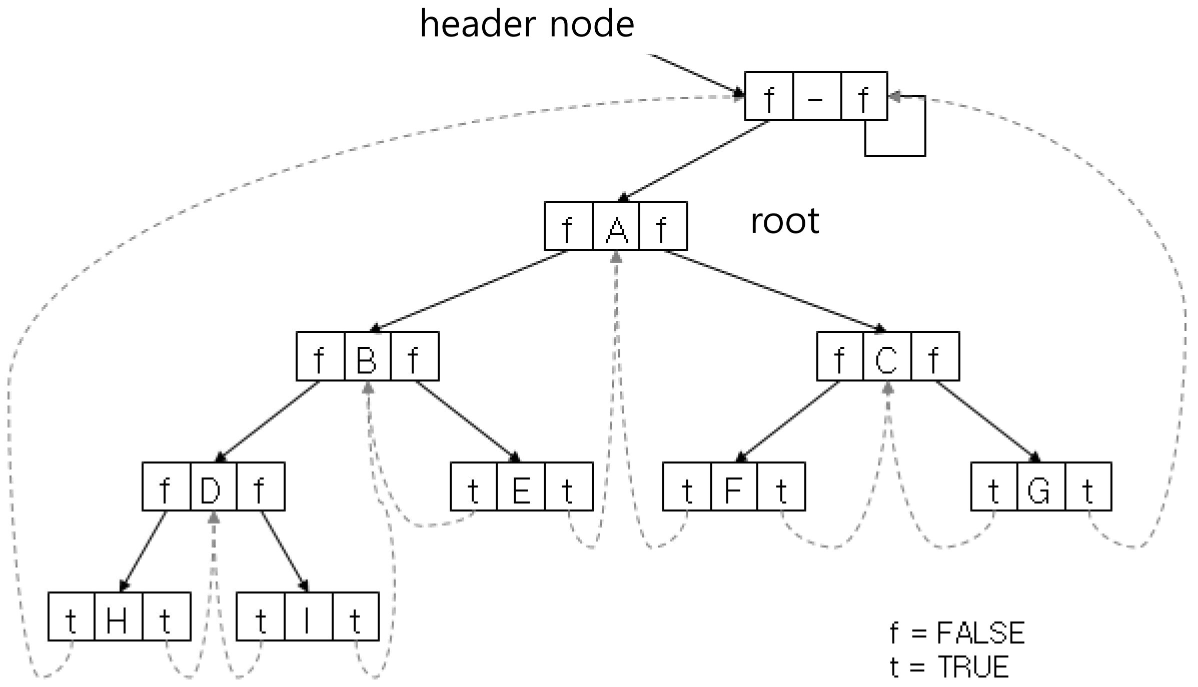
- **Still, a lot of links are wasted**
  - Lots of null pointers
    - There are n+1 null links out of 2n total links.



  - What if we can put them in good use?

Thread Binary tree

- **Let the empty link point to upper-level nodes**
  - A.J.Perlis and C.Thornton have devised a clever way to make use of these null links.
  - They replace the null links by pointers, called threads, to other nodes in the tree.



Binary Tree Traversals

- **What if we want to perform a certain process on every node of a tree?**
  - We need to visit each node in the tree once = traversal
  - When a node is visited, some operation (such as outputting its data field) is performed on it.

- **Binary tree traversal methods**
  - In-order
  - Pre-order
  - Post-order
  - Level-order

Binary Tree Traversals

■ **Elementary operations (for each node)**

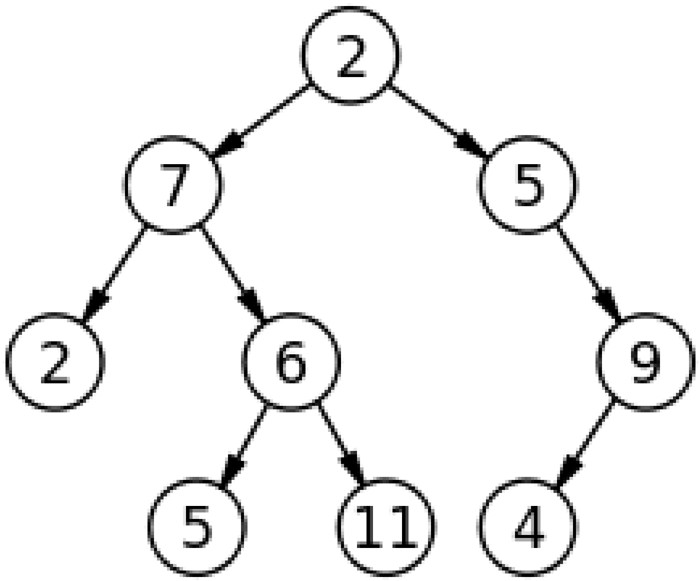
- L : moving left
- V : visit
- R : moving right
- These are performed recursively

■ **Six possible combinations**

- LVR, LRV, VLR, VRL, RVL, RLV

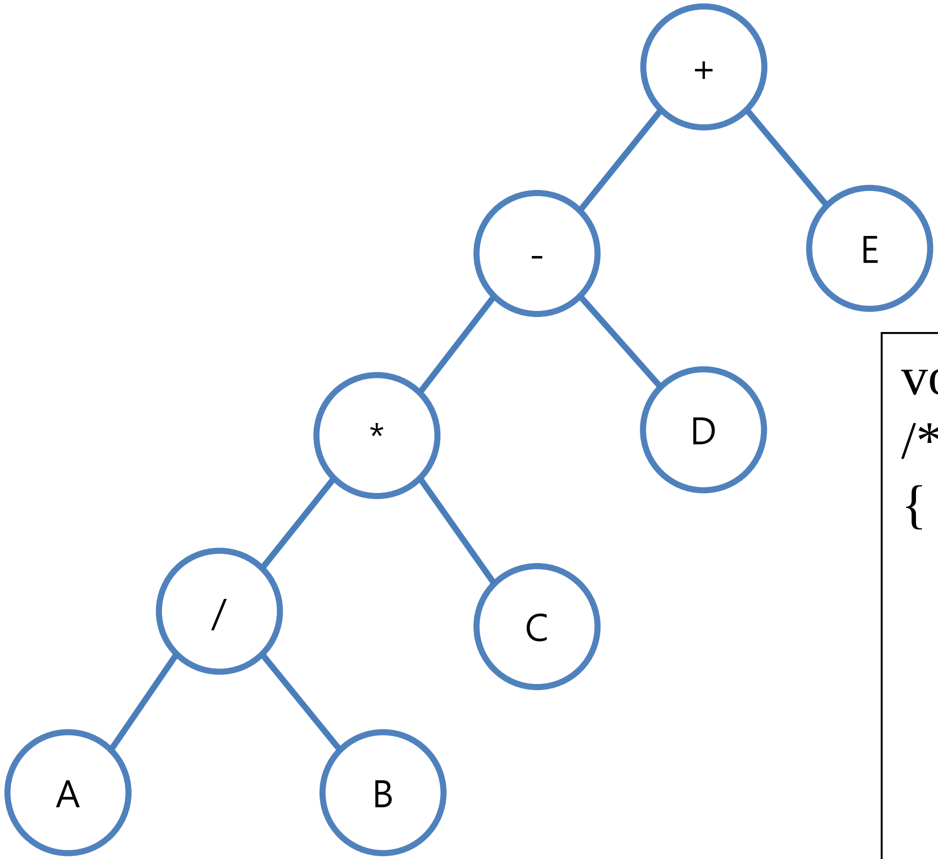
■ **Assume "R after L" is fixed**

- LVR : in-order
- VLR : pre-order
- LRV : post-order



In-order (LVR)

- Left, Visit, Right



A/B*C-D+E

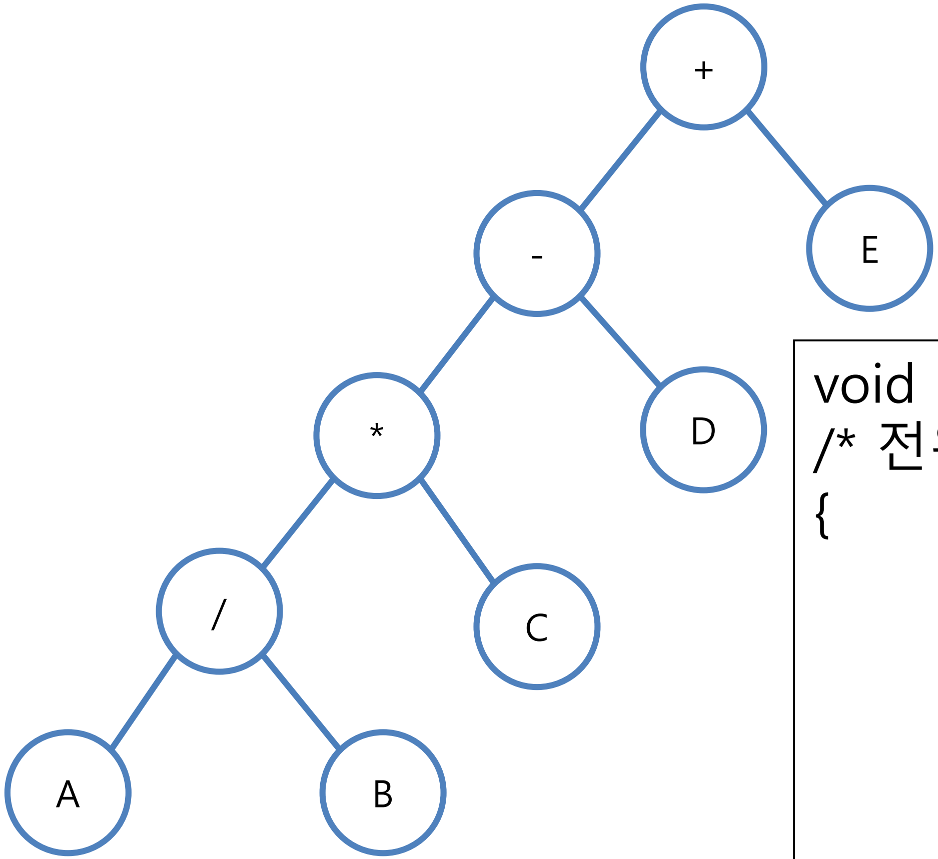
```
void inorder(treePointer ptr)
/* In-order  traverse */
{
    if (ptr) {
            inorder(ptr->leftChild);
            printf("%d", ptr->data);
            inorder(ptr->rightChild);
    }
}
```

Call of inorder	Value in root	Action	inorder	Value in root	Action
1	+		11	C	
2	-		12	NULL	
3	*		11	C	**printf**
4	/		13	NULL	
5	A		2	-	**printf**
6	NULL		14	D	
5	A	**printf**	15	NULL	
7	NULL		14	D	**printf**
4	/	**printf**	16	NULL	
8	B		1	+	**printf**
9	NULL		17	E	
8	B	**printf**	18	NULL	
10	NULL		17	E	**printf**
3	*	**printf**	19	NULL	

출력 : A/B*C -D+E

Pre-order (VLR)

- Visit, Left, Right

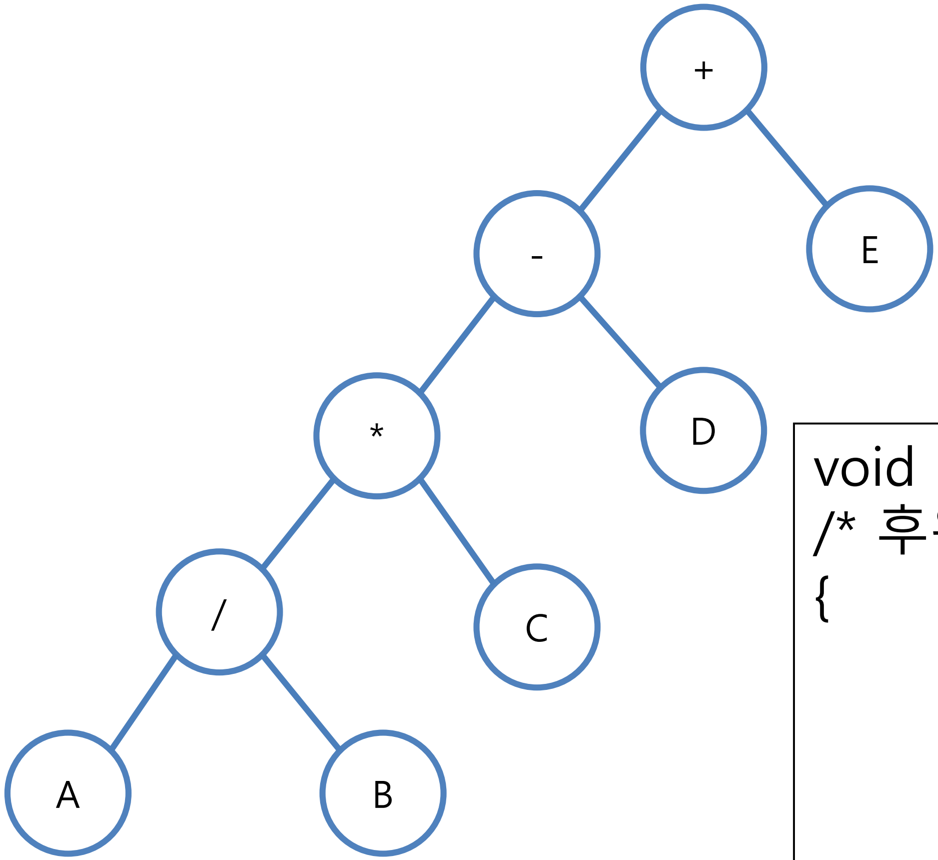


+-*/ABCDE

```
void preorder(treePointer ptr)
/* 전위 트리 순회 */
{
        if (ptr) {
                printf("%d", ptr->data);
                preorder(ptr->leftChild);
                preorder(ptr->rightChild);
        }
}
```

Post-order (LRV)

■ **Left, Right, Visit**



AB/C*D-E+

```
void postorder(treePointer ptr)
/* 후위 트리 순회 */
{
        if (ptr) {
                postorder(ptr->leftChild);
                postorder(ptr->rightChild);
                printf("%d", ptr->data);
```

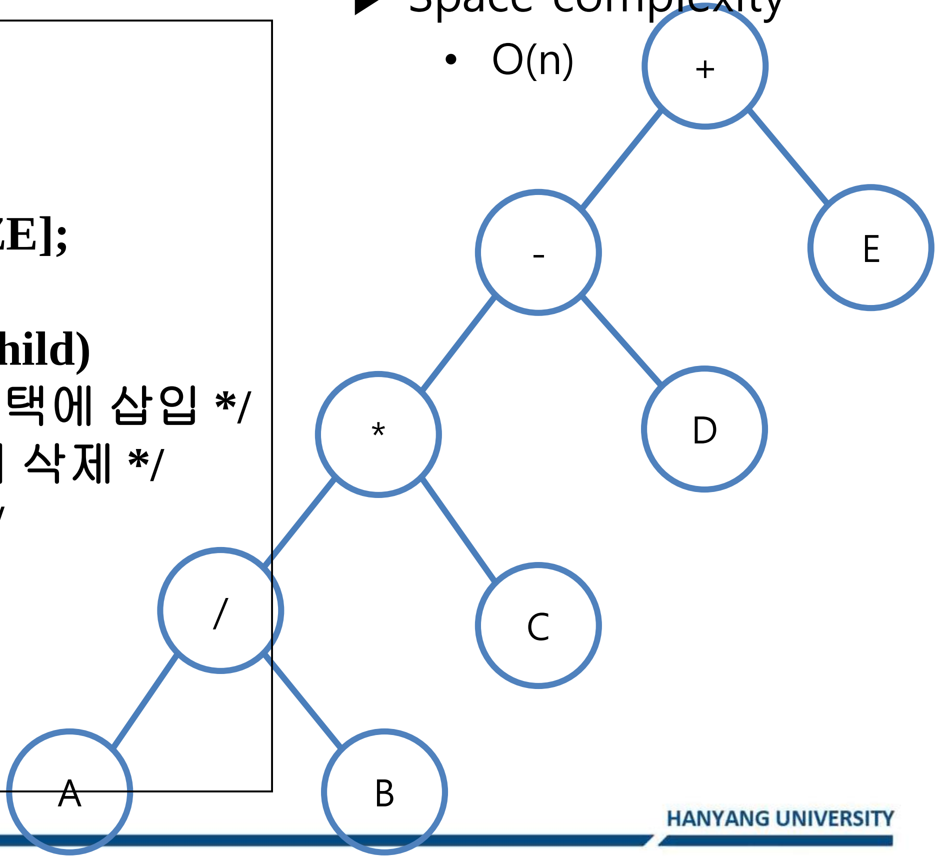
Iterative implementation

- In-order / pre-order / post-order can be implemented iteratively using a stack.

A/B*C-D+E

- Time complexity
  - O(n)
- Space complexity
  - O(n)

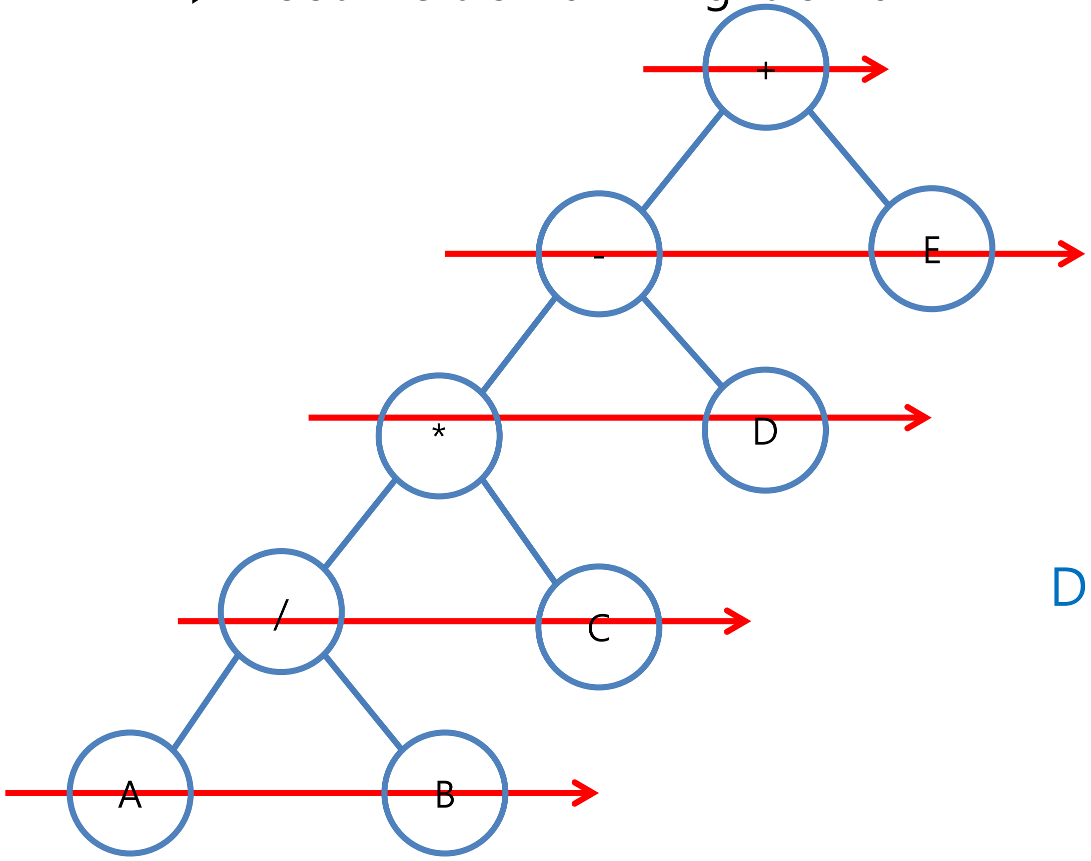
```
void iterInorder(treePointer node)
{
    int top = -1; /* 스택 초기화 */
    treePointer stack[MAX_STACK_SIZE];
    for (;;) {
        for (; node; node = node->leftChild)
            add(&top, node); /*스택에 삽입 */
        node = delete(^top); /*스택에서 삭제 */
        if (!node) break; /* 공백 스택 */
        printf("%d", node->data);
        node = node->rightChild;
    }
}
```



Level order

- **Level-by-level order**
  - Using a queue
  - Root->left child -> right child

Queue로 구현 가능하다.



+-E*D/CAB

Difficult to do recursively

Level order

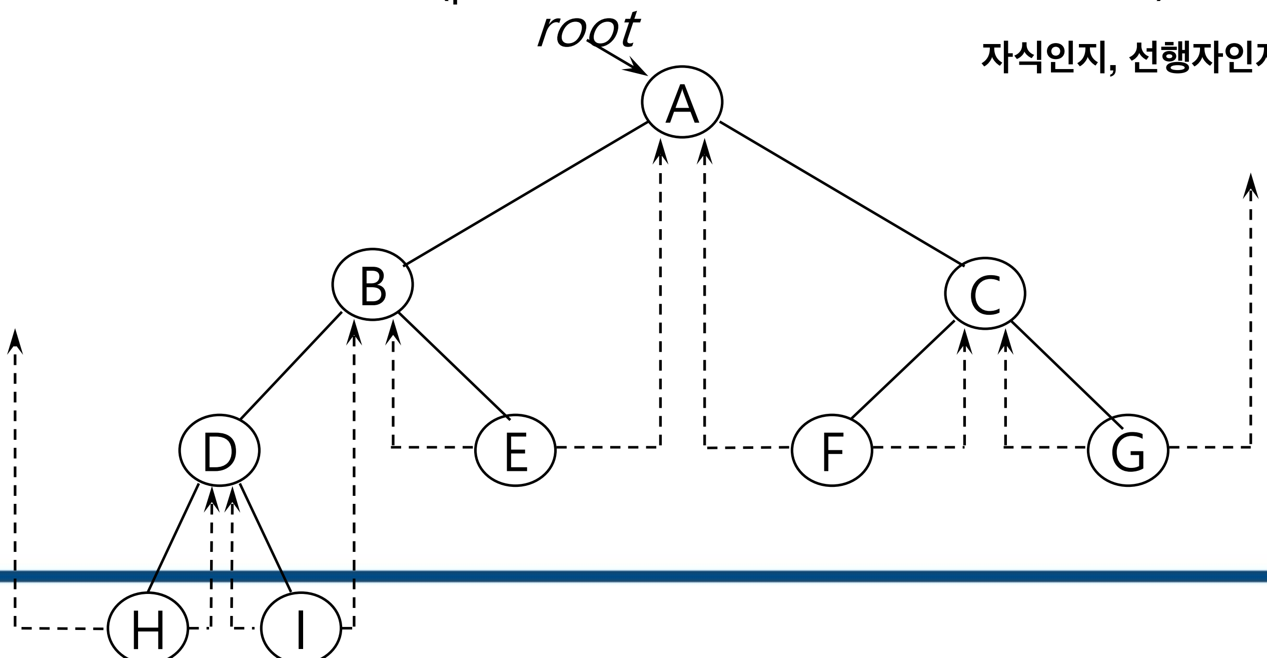
```
void levelOrder(treePointer ptr)
/* 레벨 순서 트리 순회 */
{
        int front = rear = 0;
        treePointer queue[MAX_QUEUE_SIZE];
        if (!ptr) return; /* 공백 트리 */
        addq(front, &rear, ptr);
        for ( ; ; ) {
                ptr = deleteq(&front, rear);
                if (ptr) {
                        printf("%d", ptr->data);
                        if (ptr->leftChild)
                                addq(front, &rear, ptr->leftChild);
                        if (ptr->rightChild)
                                addq(front, &rear, ptr->rightChild);
                }
                else break;
        }
}
```

Use of traversals

- Copying trees
- Testing equality
- Find the number of nodes
- Find the depth of the tree
- Parsing, etc.

Threaded Binary Tree

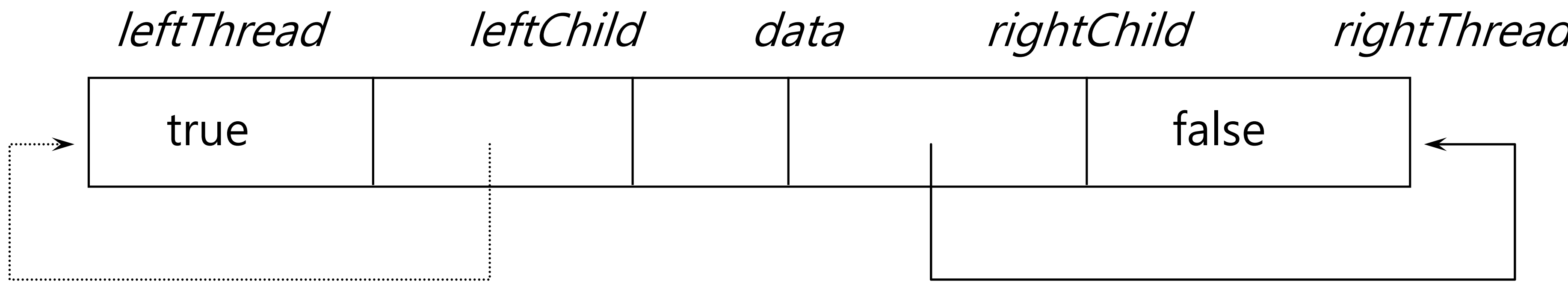
- **스레드(Thread)**
  - Replacing the null links with pointers, called threads, to other nodes in the tree.
  - if ptr->*rightChild* == NULL,
    ptr->*rightChild* = a pointer to the inorder successor of ptr
    (ptr의 중위 후속자에 대한 포인터)
  - if ptr->*leftChild* == NULL,
    ptr->*leftChild* = a pointer to the inorder predecessor of ptr
    (ptr의 중위 선행자에 대한 포인터)



자식인지, 선행자인지 후행자인지 모른다.

Threaded Binary Tree

■ **Node structure**



▶ leftThread == false if leftChild is a pointer
== true if leftChild is a thread

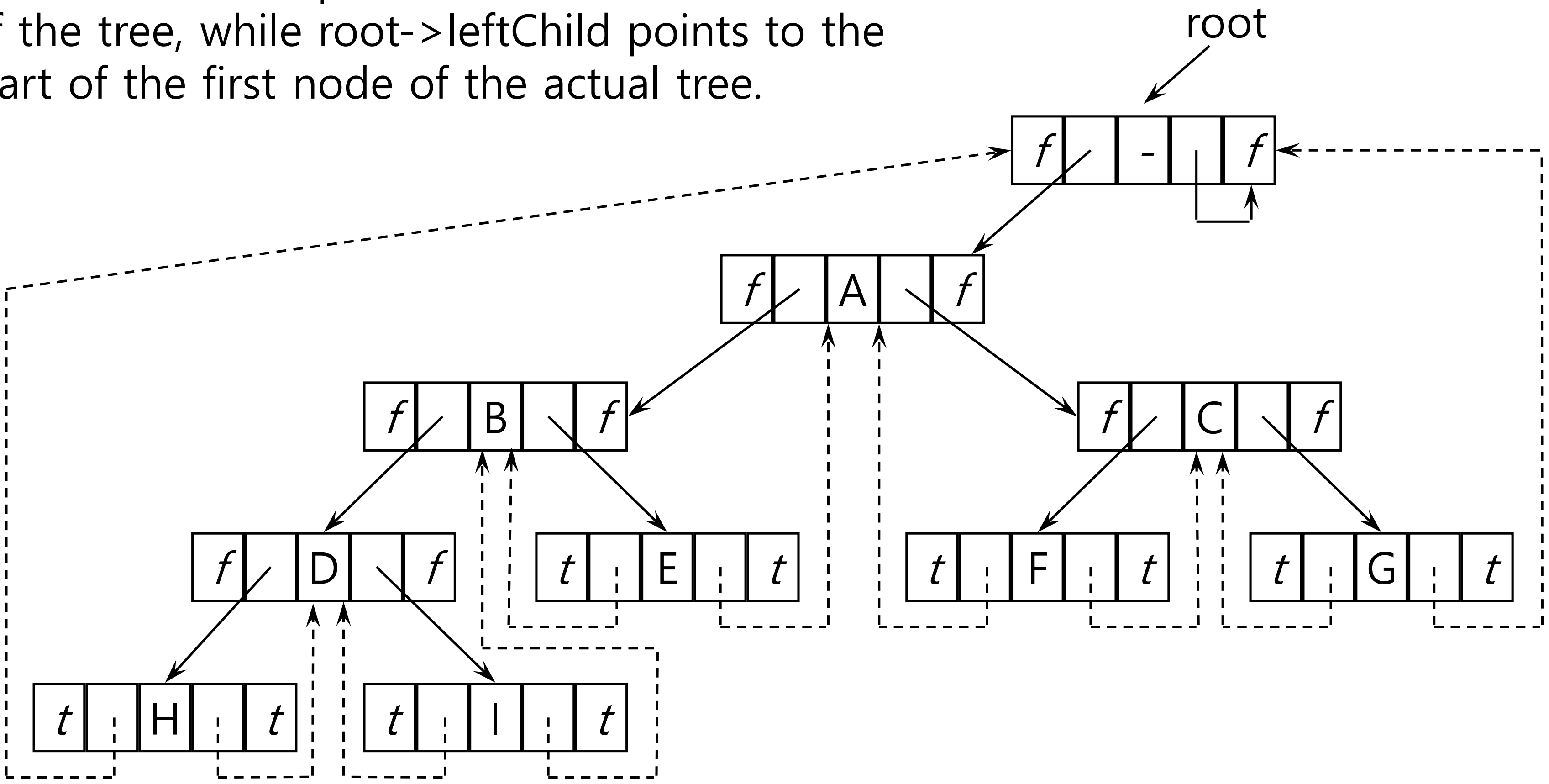
▶ rightThread == false if rightChild is a pointer
== true if rightChild is a thread

■ **Head Node, root**

▶ Solve the problem of the loose threads by having them point to the head node

Memory representation of threaded Binary Tree

The variable root points to the header node of the tree, while root->leftChild points to the start of the first node of the actual tree.



f = **false**; *t* = **true**

Inorder Traversal of a Threaded Binary Tree

- **Inorder traversal without using a stack.**
- **The inorder successor of x (중위 순회의 후속자)**
  - x->rightThread == true : x->rightChild

    == false : by following a path of left-child links from the right-child of x until we reach a node with LeftThread==true.
- **Finding the inorder successor of a node**

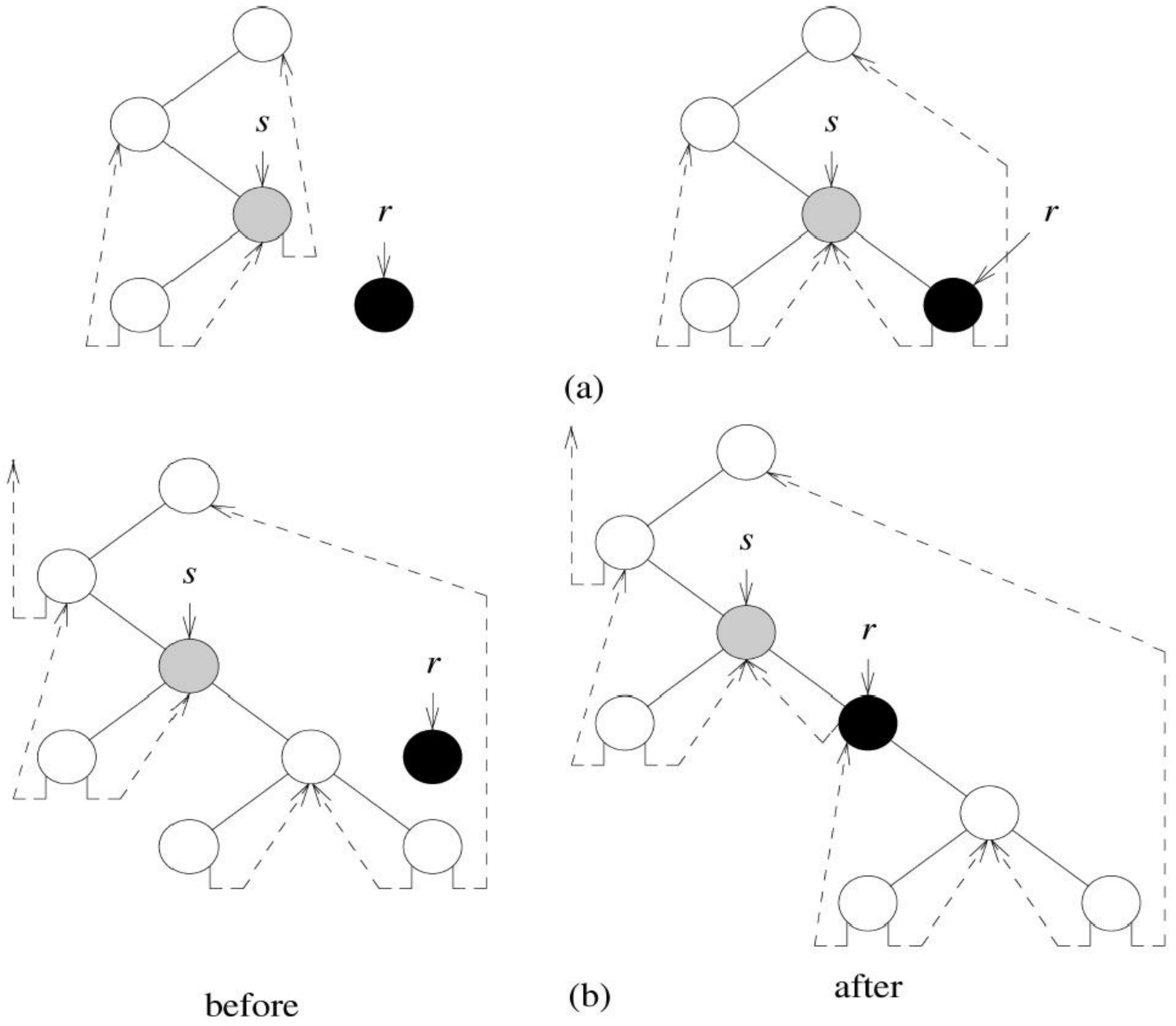
```
threadedPointer insucc(threadedPointer tree)
{/* 스레드 이진 트리에서 중위 후속자를 찾는다. */
        threadedPointer temp;
        temp = tree->rightChild;
        if (!tree->rightThread)
                while (!temp->leftThread)
                        temp = temp->leftChild;
        return temp;
}
```

Inorder Traversal of a Threaded Binary Tree

```
void tinorder(threadedPointer tree)
{/* 스레드 이진 트리의 중위 순회 */
        threadedPointer temp = tree;
        for ( ; ; ) {
                temp = insucc(temp);
                if (temp = tree) break;
                printf(“%3c”, temp->data);
        }
}
```

Inserting a Node into a Threaded Binary Tree

- The case of inserting r as the right child of a node s



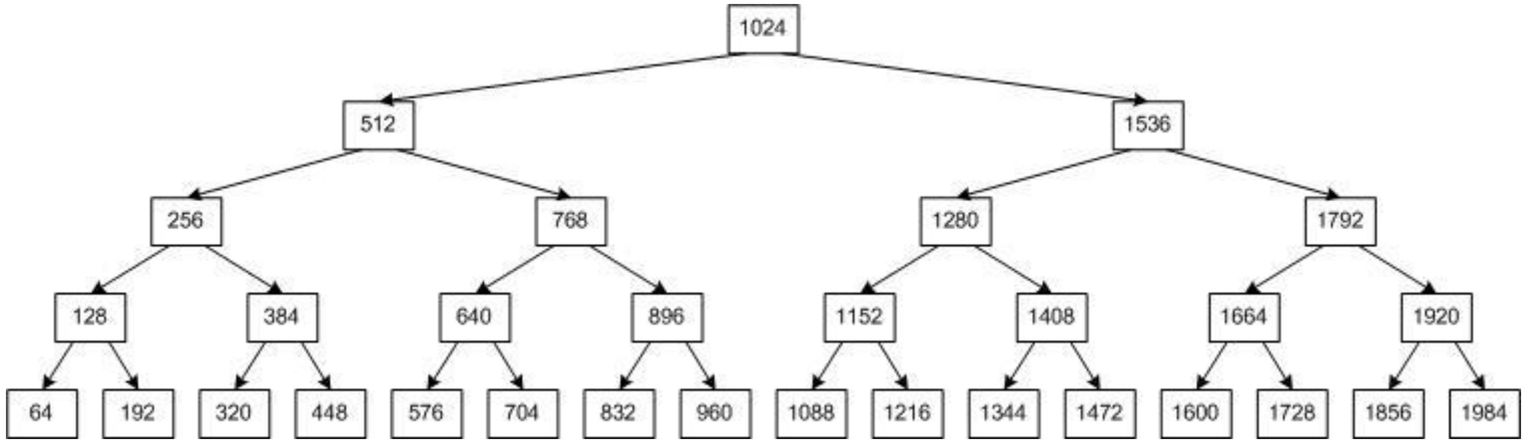
Inserting a Node into a Threaded Binary Tree

- **The case of inserting r as the right child of a node s**

```
void insertRight(threadedPointer s, threadedPointer r)
{/* 스레드 이진 트리에서 r을 s의 오른쪽 자식으로 삽입 */
        threadedPointer temp;
        r->rightChild = parent->rightChild;
        r->rightThread = parent->rightThread;
        r->leftChild = parent;
        r->leftThread = TRUE;
        s->rightChild = child;
        s->rightThread = FALSE;
        if (!r->rightThread) {
                temp = insucc(r);
                temp->leftChild = r;
        }
}
```

Binary Search Tree (BST)

- **Efficient data structure for <u>ordered</u> sequence**
  - Search
  - Inserting/removing



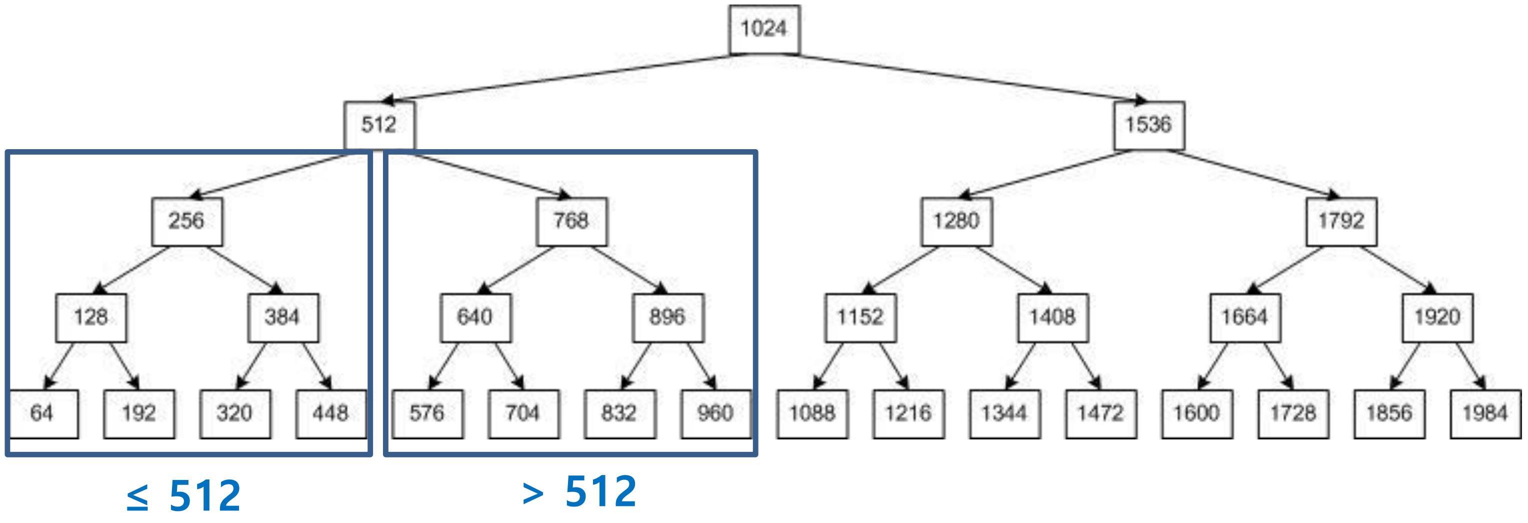
Binary Search Tree (BST)

- **Dictionary**
  - A set of pairs<key, item>

```
ADT Dictionary is
      objects: a collection of n>0 pairs, each pair has a key and an associated item
      functions:
            for all d∈Dictionary, item ∈ Item, k ∈ Key, n∈ integer
            Dictionary Create(max_size)        ::= create an empty dictionary
            Boolean IsEmpty(d,n)     ::= if(n>0) return TRUE
                                          else return FALSE
            Element Search(d,k)      ::= return item with key k,
                                          return NULL if no such element.
            Element Delete(d,k)      ::= delete and return item (if any) with key k;
            void Insert(d, item, k)  ::= insert item with key k into d.
```

Definition of BST

- For each node, the left subtree has smaller values than the node and the right subtree larger



Binary Search Tree

Definition of BST

- A binary search tree is a binary tree. It may be empty. If it is not empty then it satisfies the following properties:

(1) Each node has exactly one key and the keys in the tree are distinct.

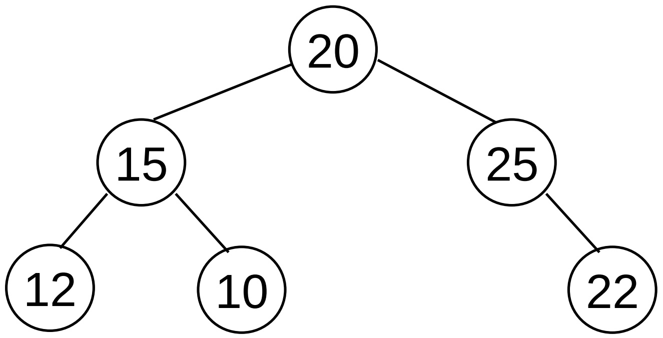
(2) The keys (if any) in the left subtree are smaller than the key in the root.

(3) The keys (if any) in the right subtree are larger than the key in the root.

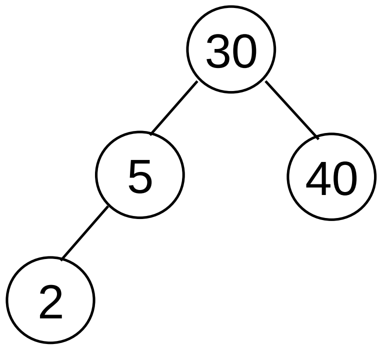
(4) The left and right subtrees are also binary search trees.

The keys must be distinct.

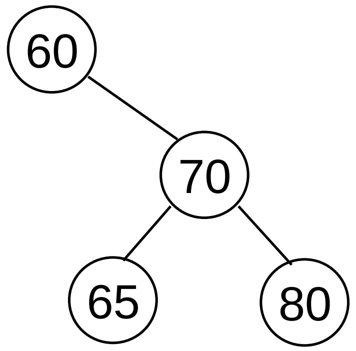
Binary Search Tree



(a) X



(b) O



(c) O

Search of BST

- **k = the root's key : the search terminates successfully.**
- **k < the root's key : search the right subtree of the root.**
- **k > the root's key : search the left subtree of the root.**

<Recursive search of a BST>

```
element* search(treePointer tree, int key)
{
/* Return a pointer to the element whose key is k, if there is no such element,
   return NULL. */
    if (!root) return NULL;
    if (key == root->data) return root;
    if (key < root->data)
             return search(root->leftChild, key);
    return search(root->rightChild, key);
}
```

```
typedef struct node * treePointer;
typedef struct node{
          int data;
          treePointer leftChild, rightChild;
          };
```

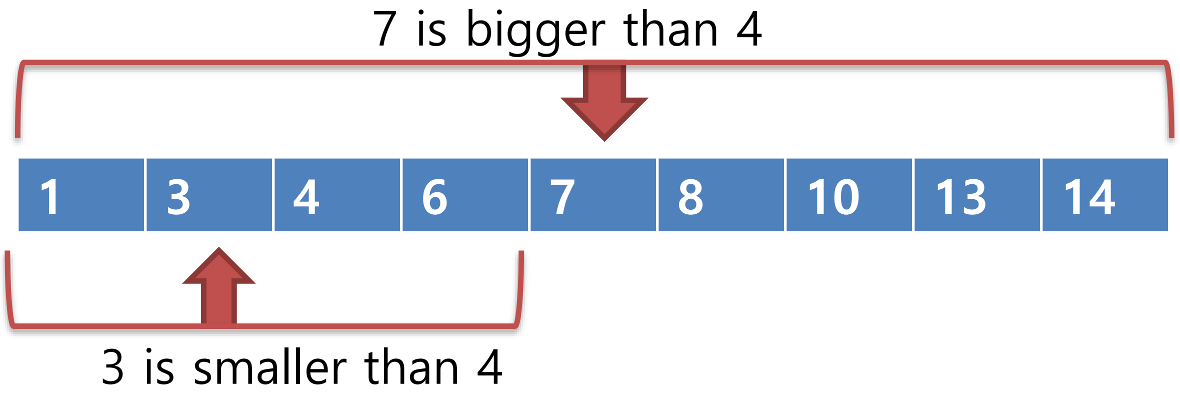
Search of BST

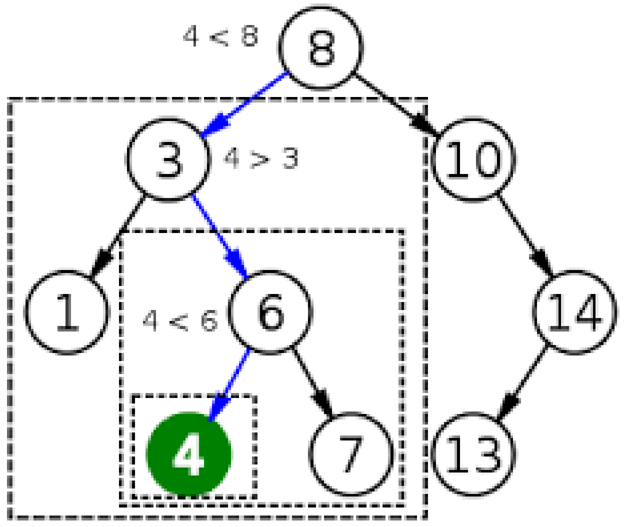
<Iterative search of a BST>

```
element* iterSearch(treePointer tree, int key)
{
/* return a pointer to the element whose key is k, if there is no element,
   return NULL */
while (tree) {
                 if (key == tree->data) return tree;
                 if (key < tree->data)
                               tree = tree->leftChild;
                 else
                               tree = tree->rightChild;
                 return NULL;
}
```

Search

■ **Binary search is native to BST**

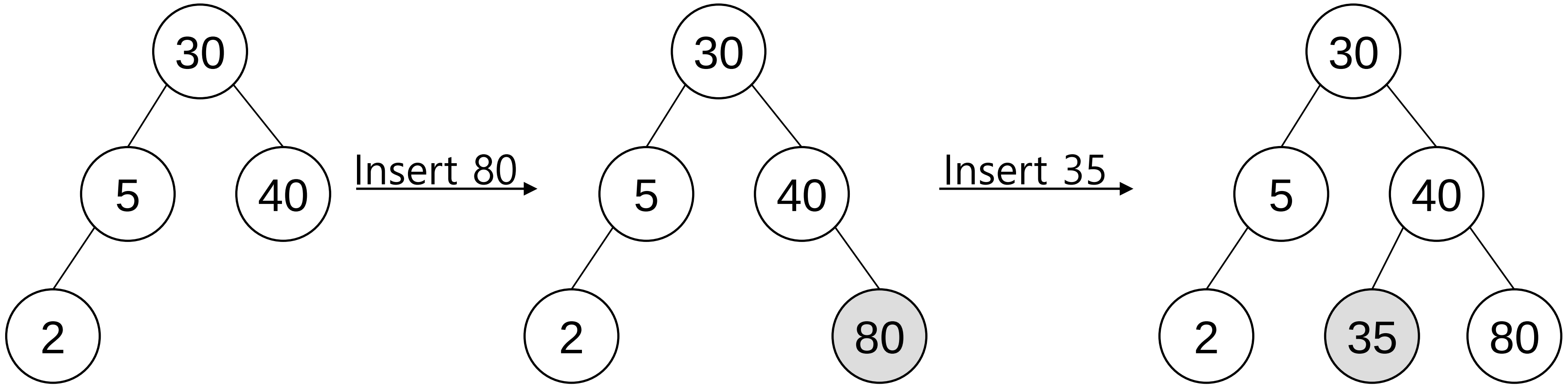




- h : the height of the BST
- Performing the search operation using *search()* or *iterSearch()* in O(h)
- *search()* has an additional stack space requirement which is O(h).

Inserting into a BST

- **To insert a dictionary pair whose key is k,**
  We must first verify that the key is different from those of existing pairs.
  - Search the tree for a node with k.
  - If this search terminates unsuccessfully, then we insert the pair at the point the search terminated.

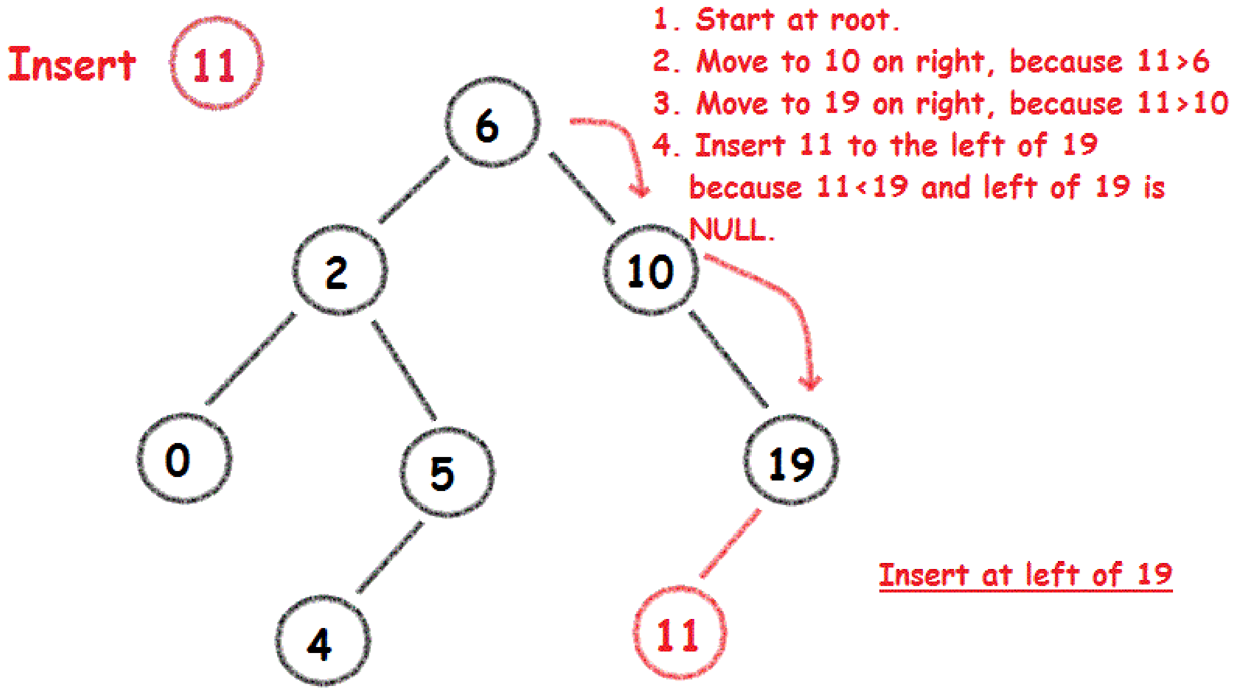


Inserting into a BST

```
void insert (treePointer *node, int k, iType theItem)
{/* If a node in the tree points to k, the node does nothing;
    Otherwise add a new node with data=(k, theItem) */
        treePointer ptr, temp = modifiedSearch(*node, k);
        if(temp || !(*node)) {
                /* k is not in the tree */
                MALLOC(ptr, sizeof(*ptr));
                ptr->data.key   = k;
                ptr->data.item = theItem;
                ptr->leftChild = ptr->rightChild = NULL;
                if(*node) /* insert as child of temp */
                        if(k<temp->data.key) temp->leftChild = ptr;
                        else temp->rightChild = ptr;
                else *node = ptr;
        }
}
```

Inserting into a BST

■ Search, and insert


Insert 11
1. Start at root.
2. Move to 10 on right, because 11>6
3. Move to 19 on right, because 11>10
4. Insert 11 to the left of 19 because 11<19 and left of 19 is NULL.
6
2
10
0
5
19
4
11
Insert at left of 19

▶ What if there is already 11?

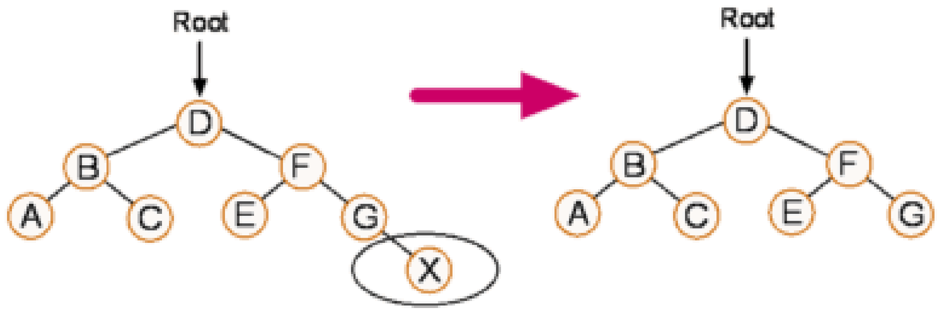
Remove operation in BST

- **Deletion of a leaf node**
  - The left-child field of its parent is set to 0 (NULL) and the node freed.
- **Deletion of a nonleaf node that has only one child**
  - The node containing the dictionary pair to be deleted is freed, and its single-child takes the place of the freed node. (삭제된 노드의 자식을 삭제된 노드의 자리에 위치)
- **Deletion of a nonleaf node that has two children**
  - The pair to be deleted is replaced by either the largest pair in its left subtree or the smallest one in its right subtree.
  - We proceed to delete this replacement pair from the subtree from which it was taken.
- **Time complexity : O(h)**

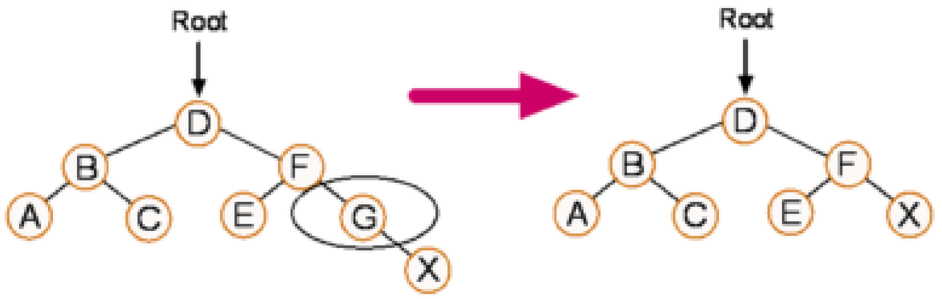
Remove operation in BST (1)

- If the node has less than 2 children.

Leaf Deletion

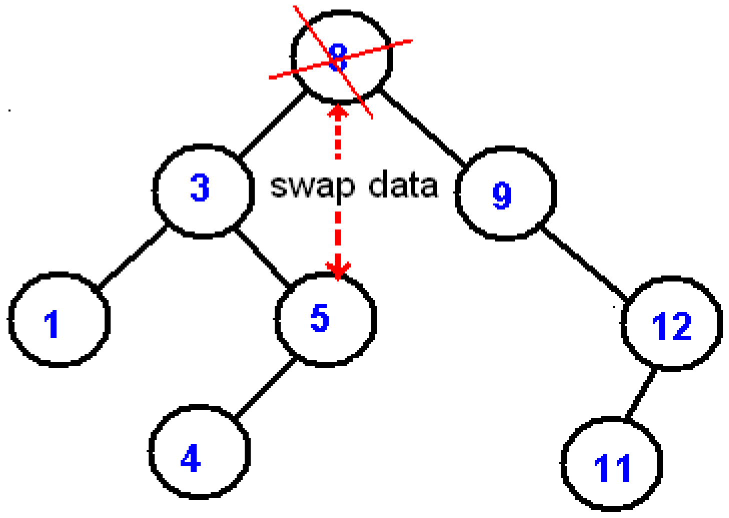


Deleting a node with a single child

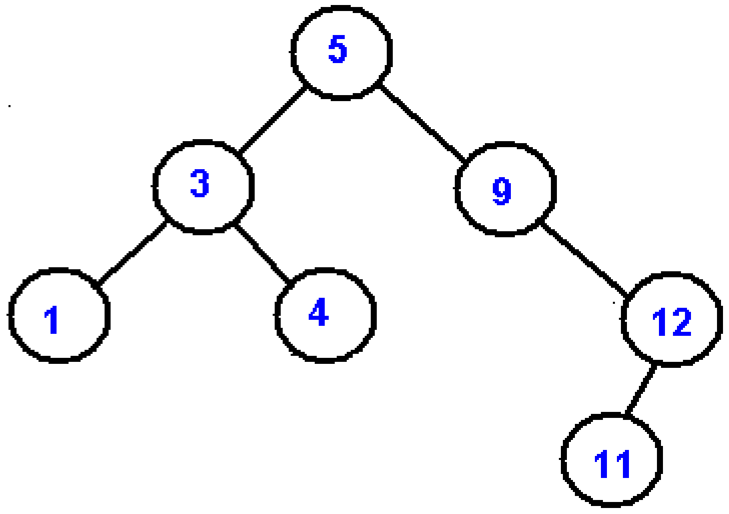


Remove operation in BST (2)

- **If the node has 2 children**
  - If we substitute the deleted node A with the largest node B in the left subtree, then B will not have a right child.



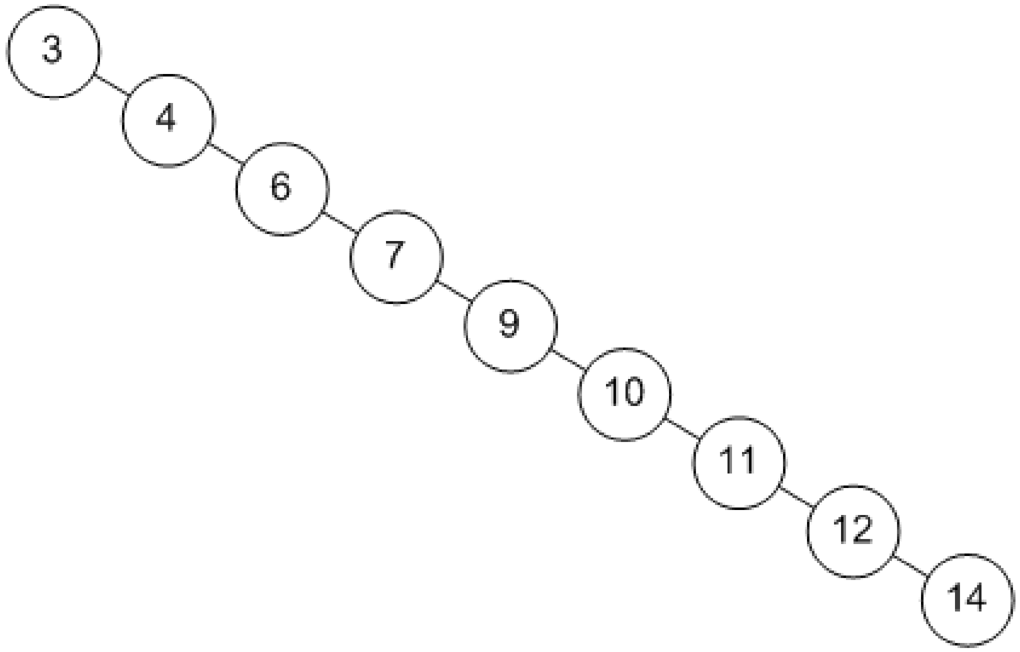
before deletion



after deletion

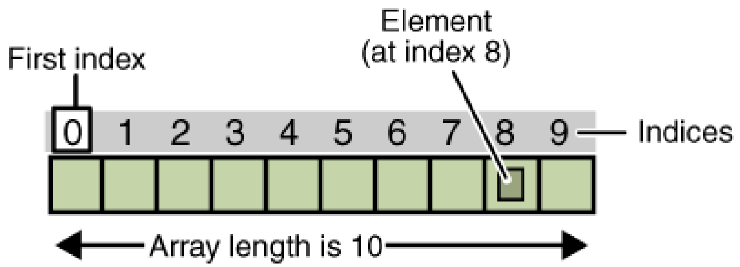
Binary Search Tree

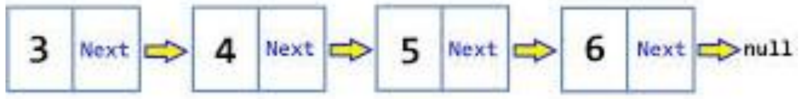
- **… is efficient only when it is balanced**
  - Unbalanced trees will be useless

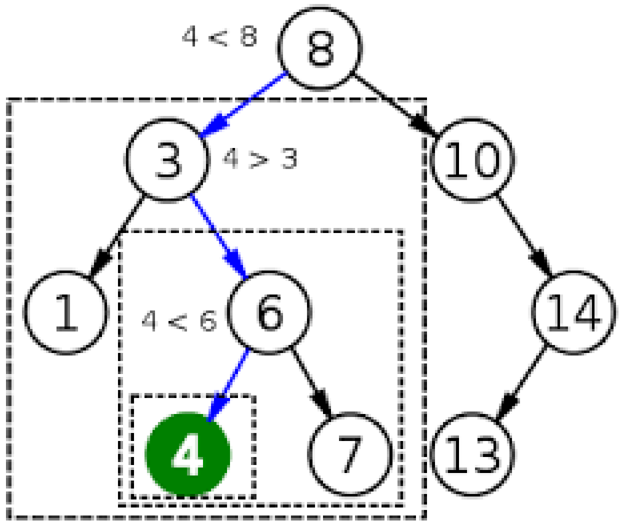


Time Complexity

	Array (unsorted)	Linked list	Array (sorted)	BST (balanced)
Search	O(n)	O(n)	O(log n)	O(log n)
Insert	O(1)	O(1)	O(n)	O(log n)
Remove	O(n)	O(n)	O(n)	O(log n)

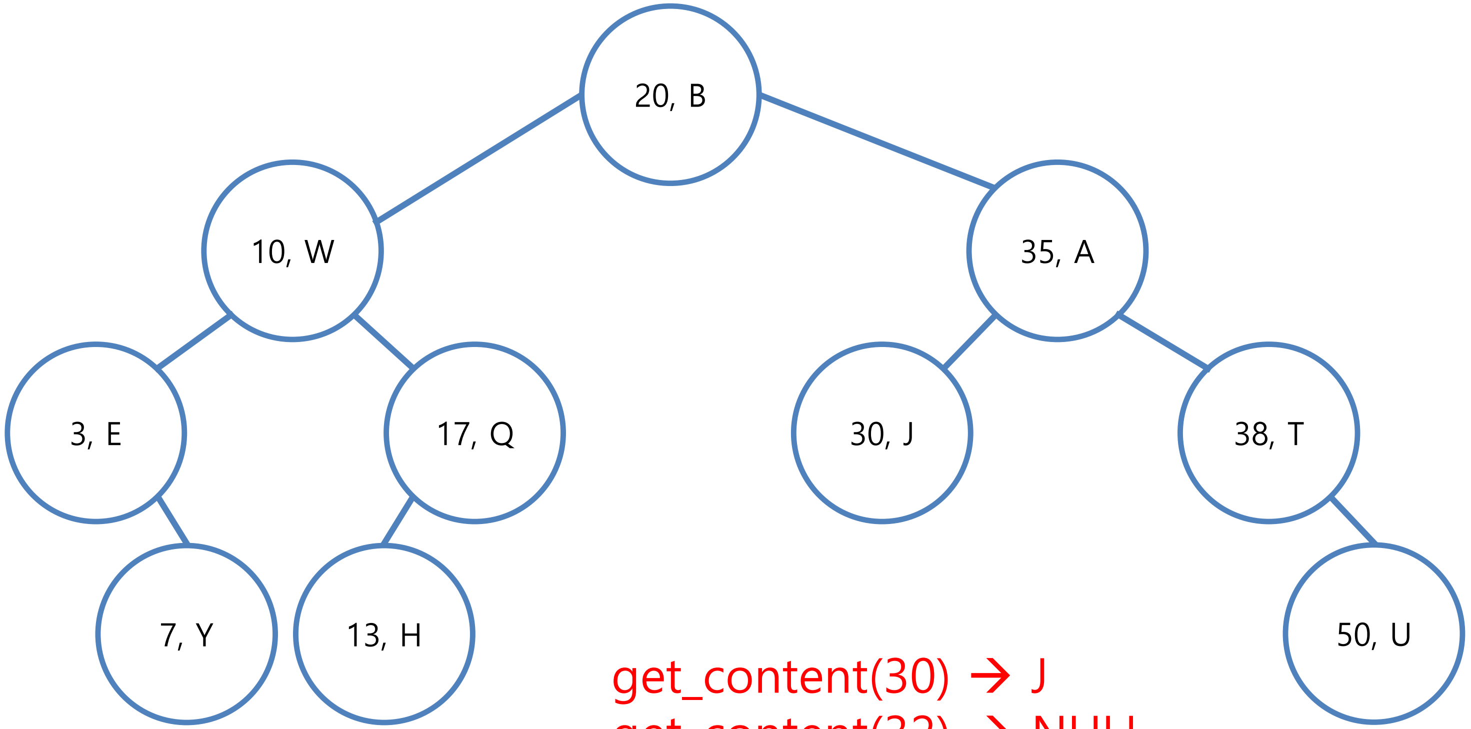






Application of BST

- **Dictionary**
  - Data fields of a node: (key, content)



get_content(30) → J
get_content(32) → NULL