Data Structures

Lecture 7: Tree

Dongbo Min
Department of Computer Science and Engineering
Ewha Womans University, Korea
E-mail: dbmin@ewha.ac.kr

Tree

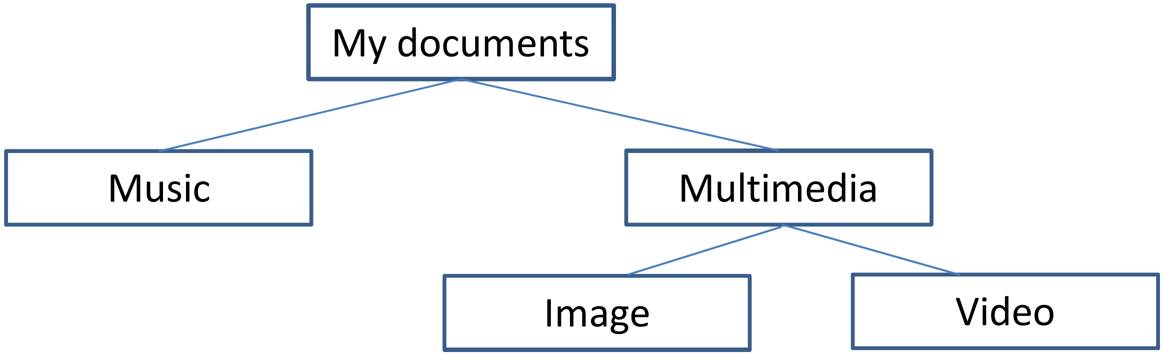
- Tree: Data structure representing a hierarchical structure
  Note) Lists, stacks, and queues are linear structures

- Tree consists of nodes of parent-child relationship.

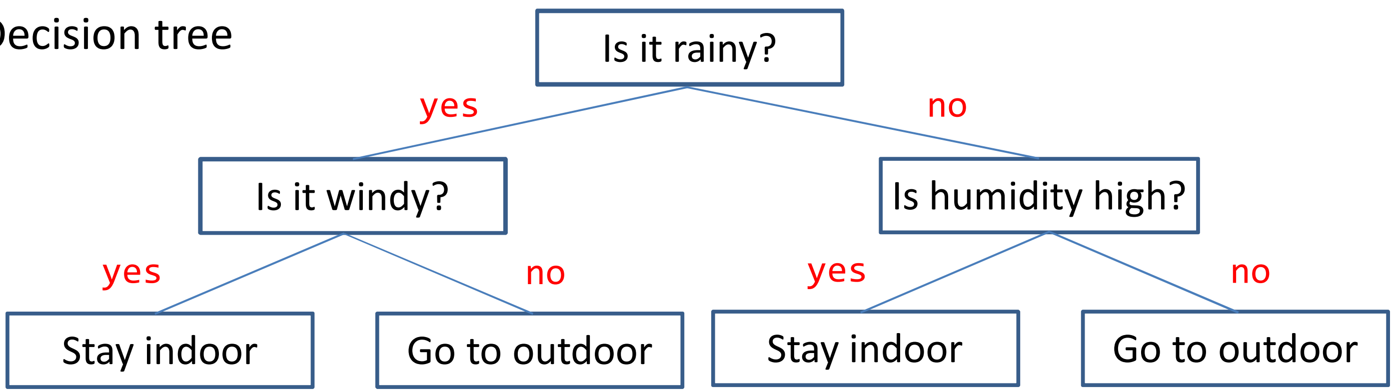
- Applications
  – Directory structure of computer disks
  – Decision tree in artificial intelligence

Tree

Directory structure

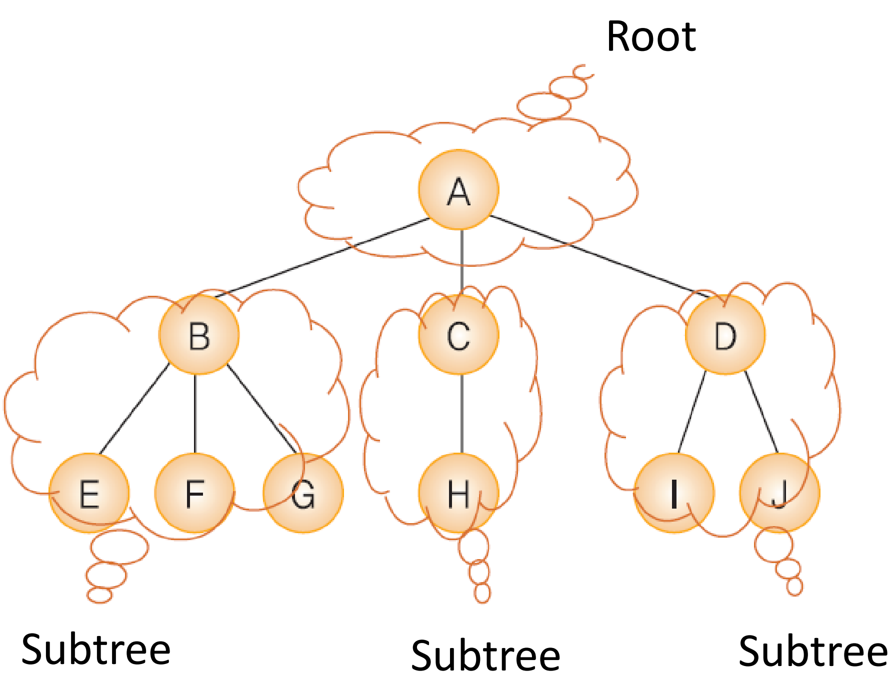


Decision tree



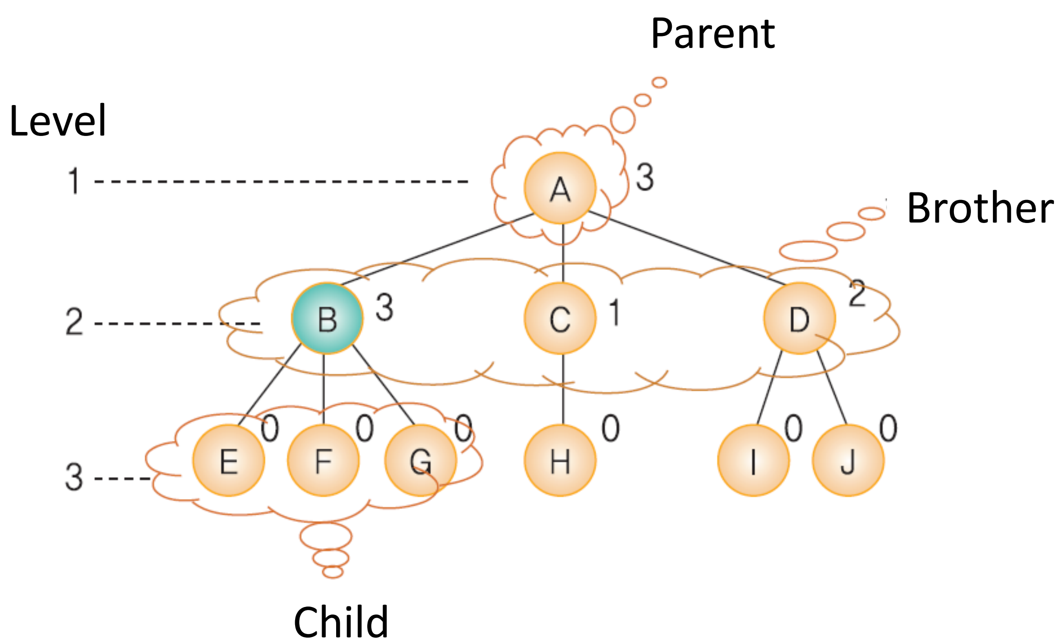
Terminology

- Node: Tree component
- Root: Node without parents
- Subtree: consists of one node and its descendants
- Terminal node: node without children (E, F, G, H, I, J)
- Non-terminal node: node with at least one child (A, B, C, D)

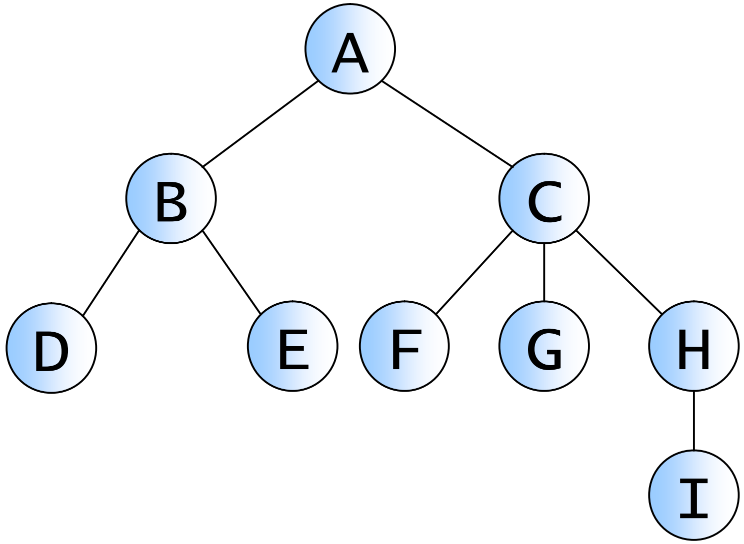


Terminology

- Level: the number of each layer in the tree
- Height: the maximum level of the tree
- Degree: the number of child nodes the node has
- Ancestor: parent, grandparent
- Offspring node: child, grandchild



Example



A is the root node.

B is the parent node of D and E.

C is the brother node of B.

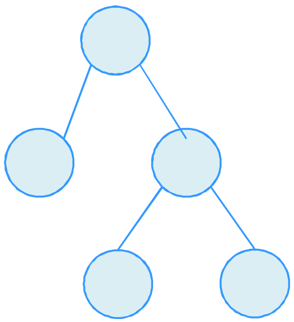
D and E are child nodes of B.

The degree of B is 2.

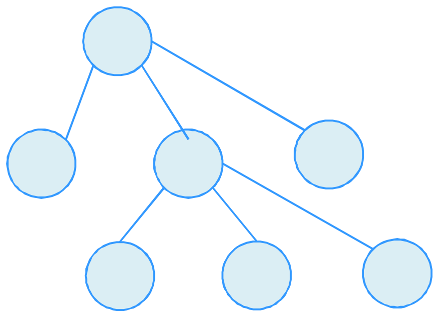
The height of the tree above is 4.

Tree Type

Binary tree

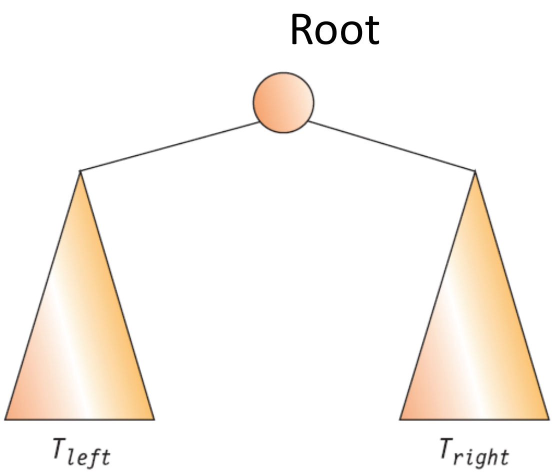


General tree



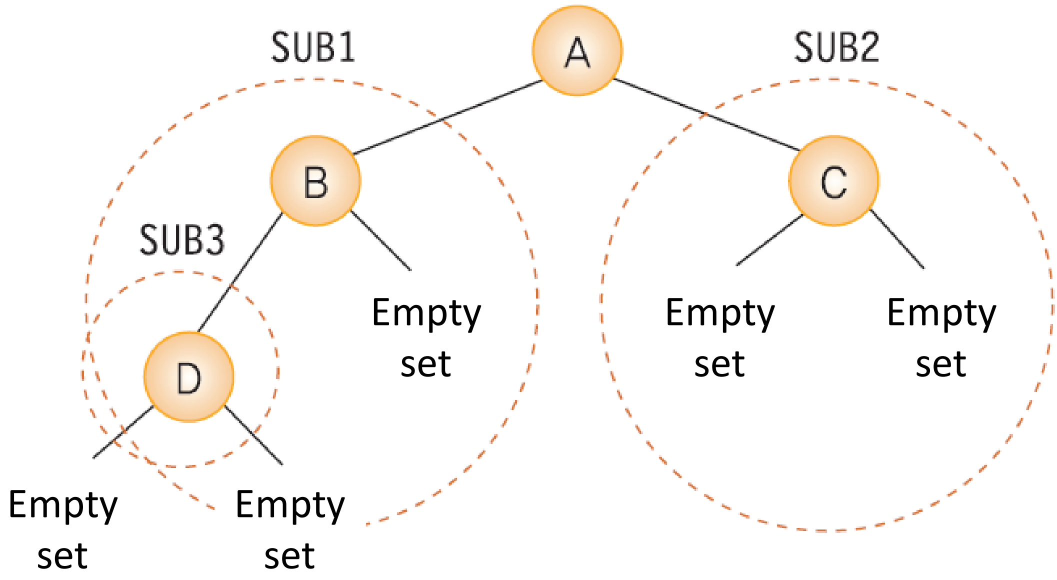
Binary Tree

- Binary tree
  - A tree in which all nodes have at most two subtrees.
  - Up to two child nodes exist in a node
  - The degree of all nodes is 2 or less -> Easy to implement
  - There is an order between the subtrees (left and right).



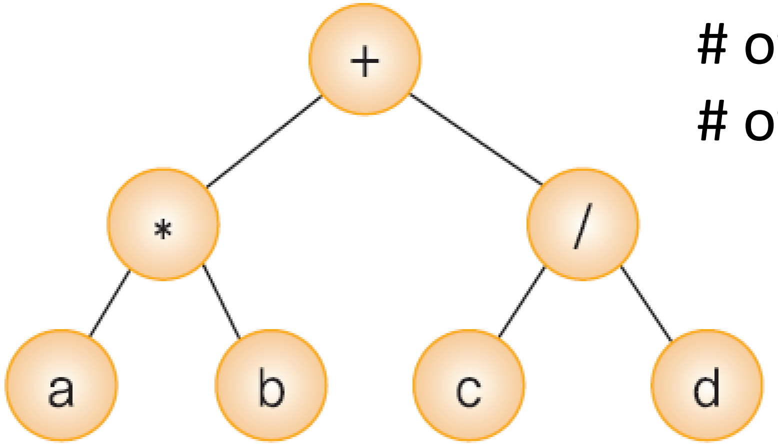
Binary Tree

- Binary tree
  - is either an empty set or
    a finite set of nodes consisting of a root, a left subtree, and a right subtree.
  - The subtrees of the binary tree should be binary trees.



Binary Tree

- If the number of nodes is n, the number of links is n-1



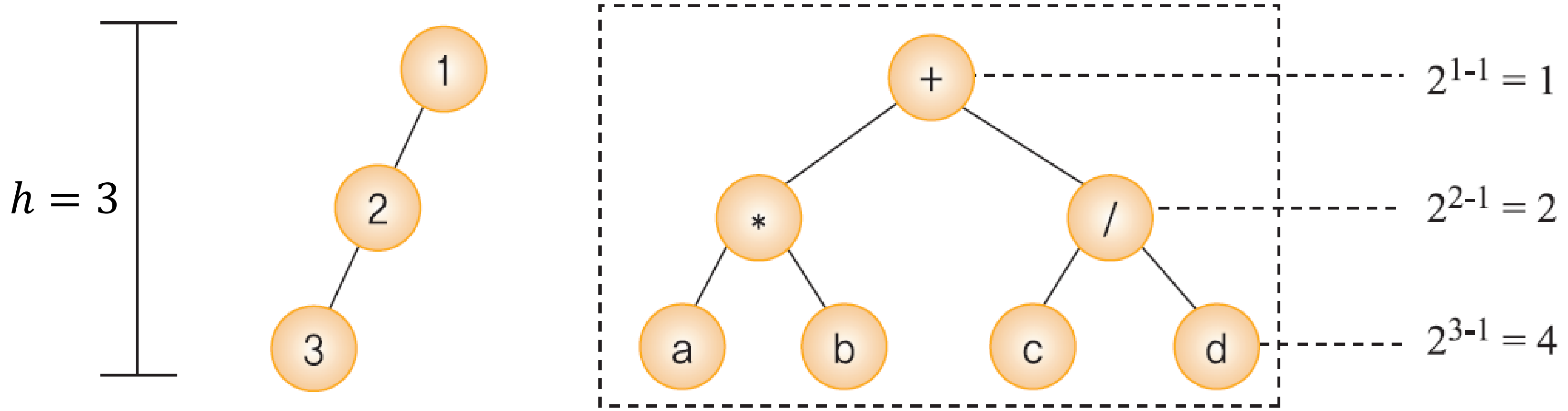
# of node: 7

# of links: 6

Binary Tree

- For a binary tree of height h,

$$h \leq \text{\# of nodes} \leq 2^h - 1$$



The minimum number of nodes $= 3$

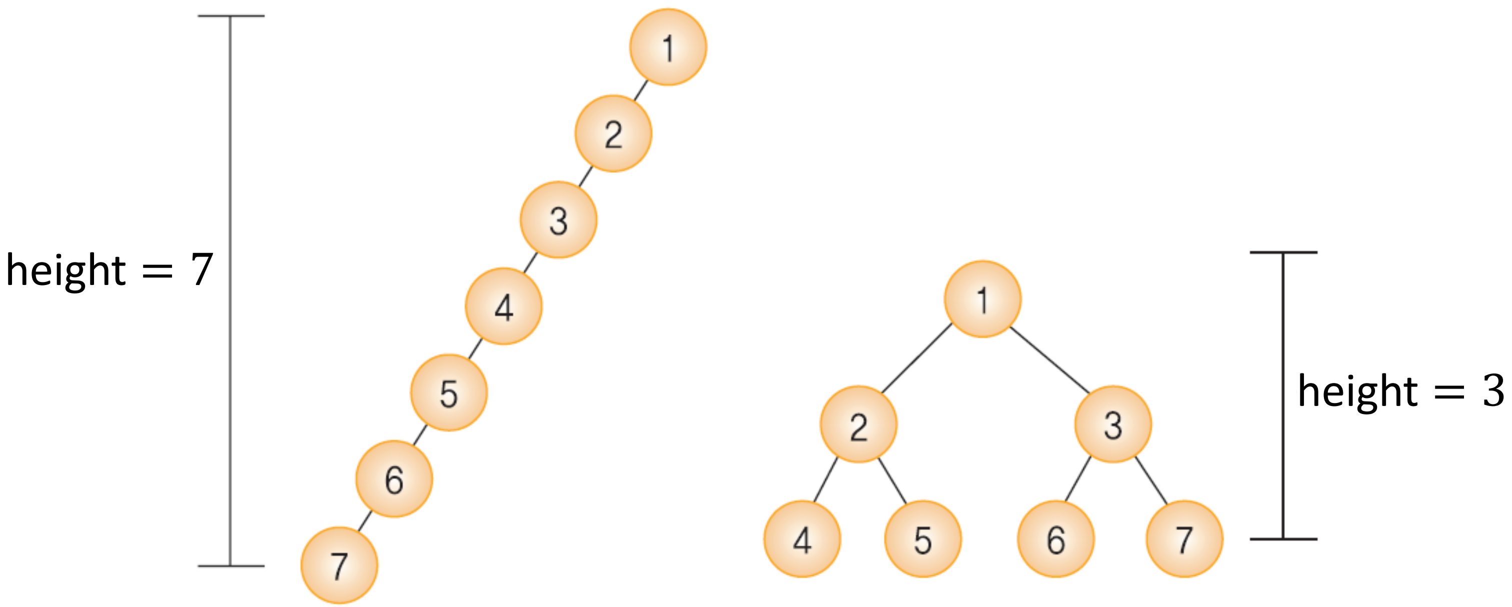
The maximum number of nodes $= 1 + 2 + 4 = 7$

Binary Tree

- For the binary tree with n nodes

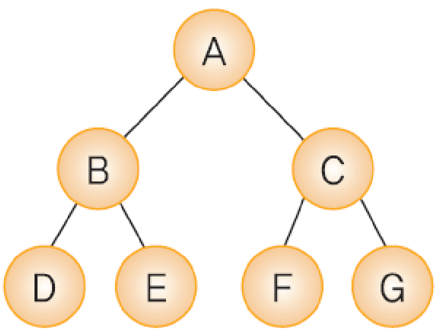
$$\lceil log_2(n+1) \rceil \leq \text{height of binary tree} \leq n$$

$\lceil x \rceil$: rounding operator
ex) $\lceil 3.1 \rceil = 4$

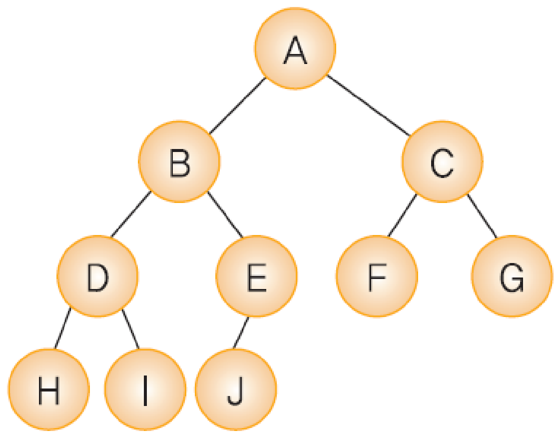


Binary Tree

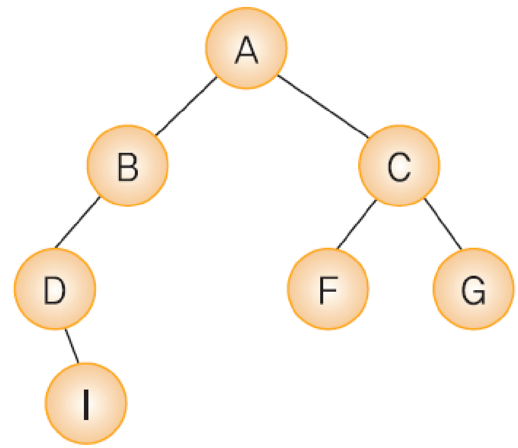
- Full binary tree
- Complete binary tree
- Other binary tree



Full binary tree



Complete binary tree



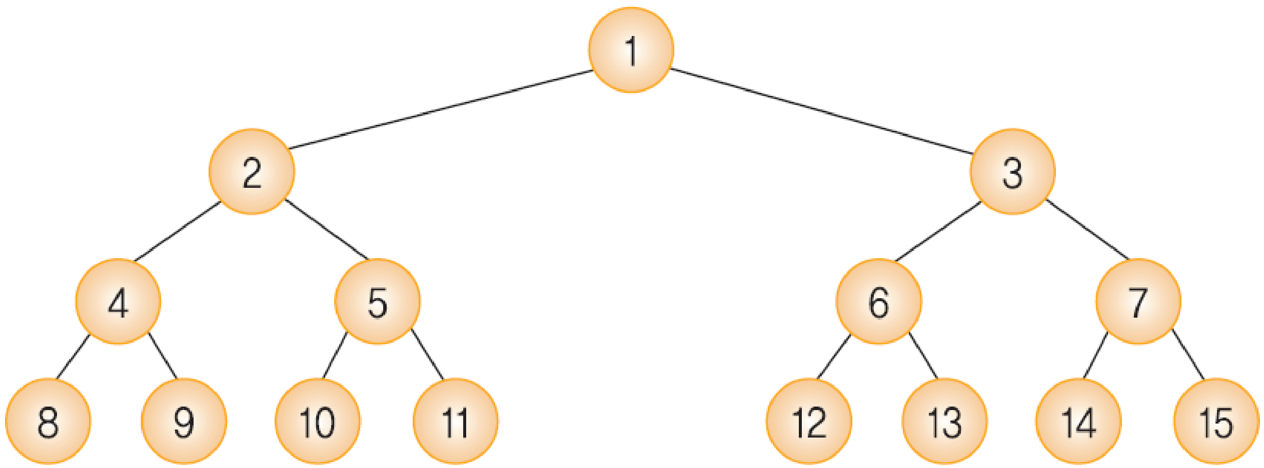
Other binary tree

Full Binary Tree

- Binary tree which is full of nodes at each level of the tree

# of nodes: $$\sum_{i=0}^{k-1} 2^i = 2^k - 1$$

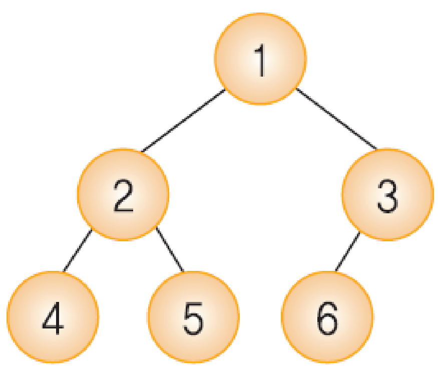
- A full binary tree can be numbered as follows.



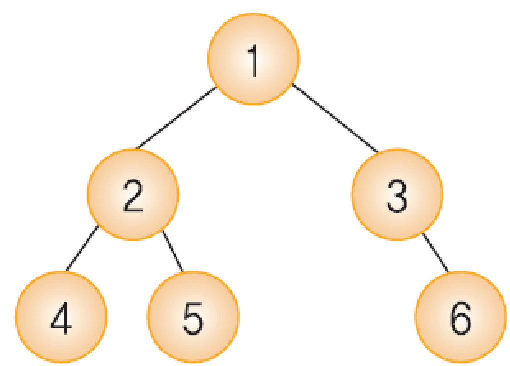
Complete Binary Tree

- Complete binary tree
  - Levels 1 to Level k-1 are filled with nodes.
  - At last level k, nodes are filled in order from left to right

Note) The node number is identical to that of full binary tree



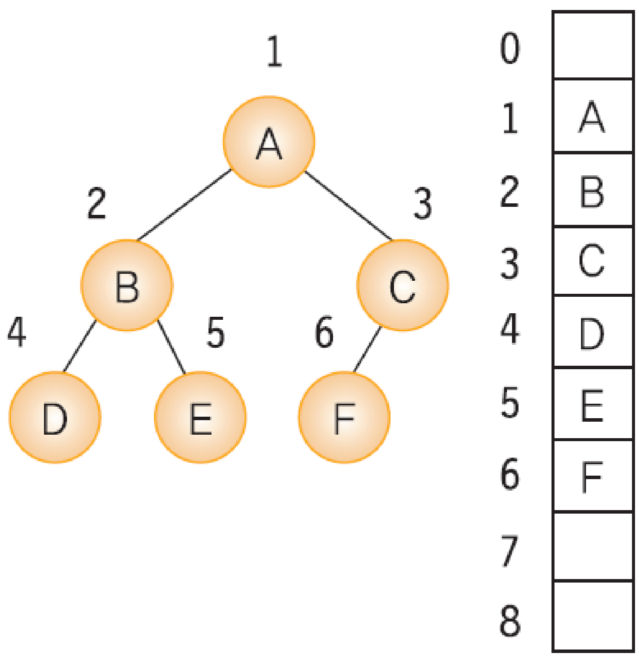
Complete binary tree



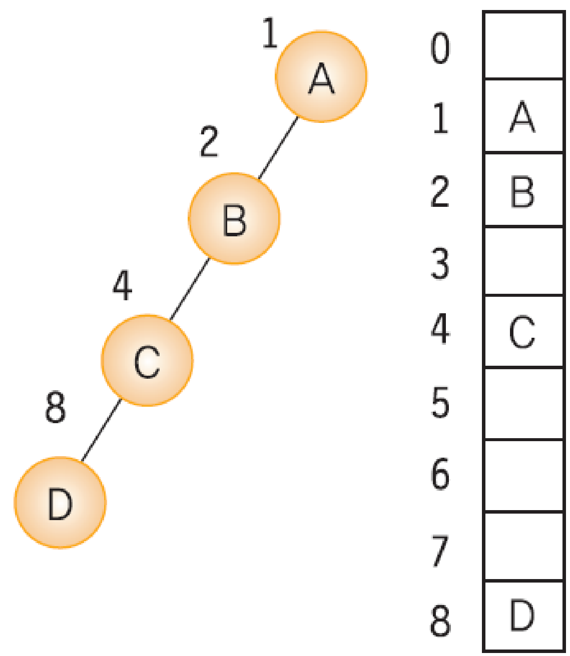
This is NOT complete binary tree

Binary Tree using Array

- Assumption: all binary trees are a full binary tree
- Each node is numbered and its number is used as an index of the array to store the node's data in the array
  - Pros: easy to implement
  - Cons: wastes memory spaces, except for full or complete binary trees



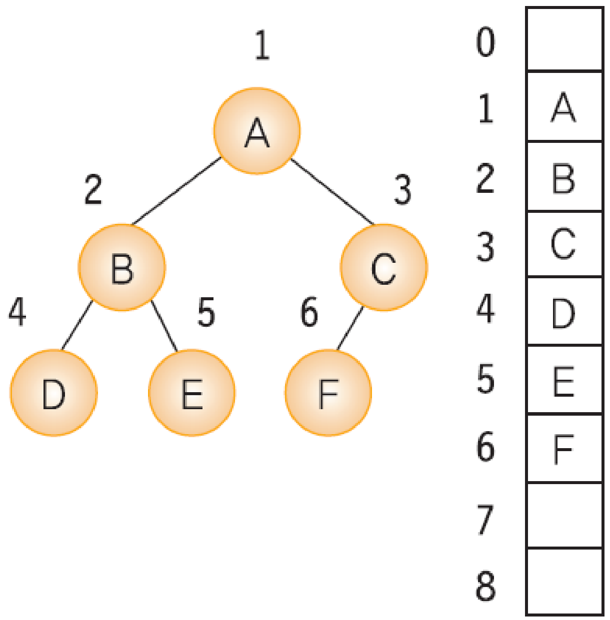
Complete binary tree



Slanted binary tree

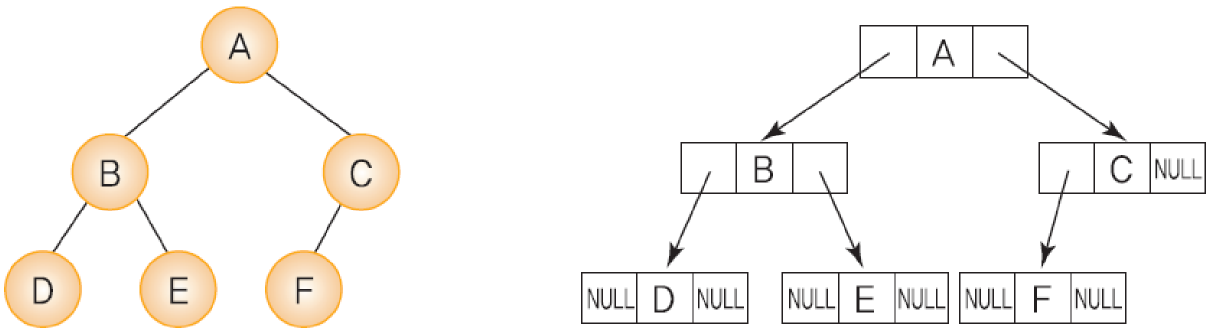
Binary Tree using Array

- Index of parent and child
  - Parent node of node i: $i/2$
  - Left child node of node i: $2i$
  - Right child node of node i: $2i + 1$

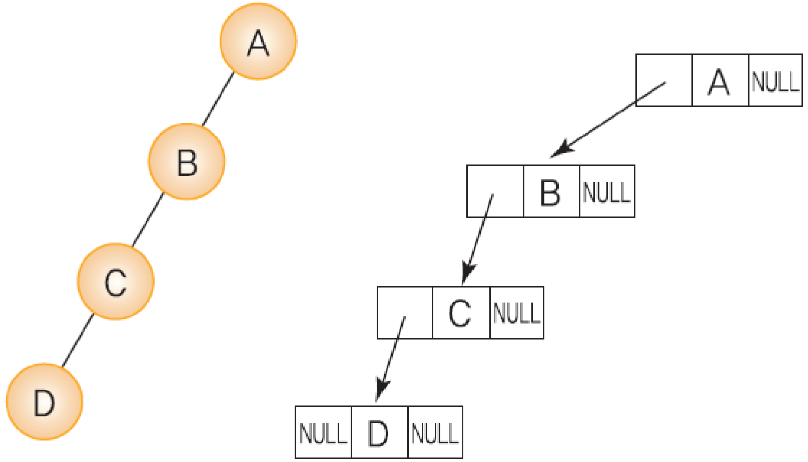


Binary Tree using Linked List

- Using a pointer, a parent node points a child node.



Complete binary tree



Slanted binary tree

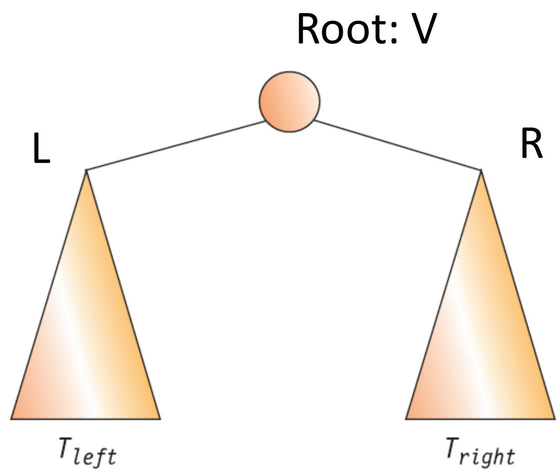
이화여자대학교
EWHA WOMANS UNIVERSITY

Binary Tree using Linked List

```
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;     // node n1
        n1->left = n2;
        n1->right = n3;

        n2->data = 20;     // node n2
        n2->right = NULL;

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

Traversal of Binary Tree

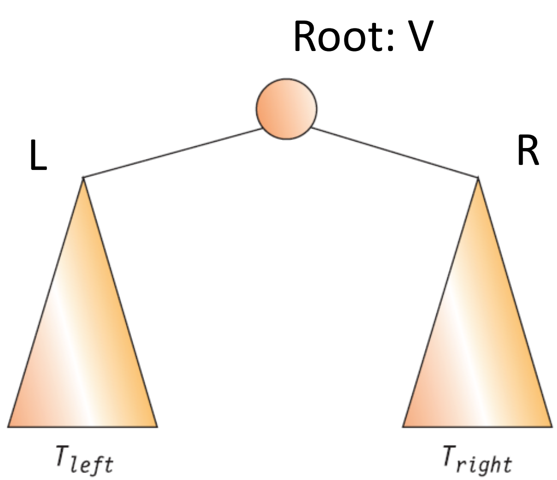
- Traversal: visiting all nodes of the tree
  - Preorder traversal: V -> L -> R
    - The root node is visited before child nodes (L/R).
  - Inorder traversal: L -> V -> R
    - Visit in an order of left descendant, root, right descendant.
  - Postorder traversal: L -> R -> V
    - Child nodes (L/R) are visited first from the root node.

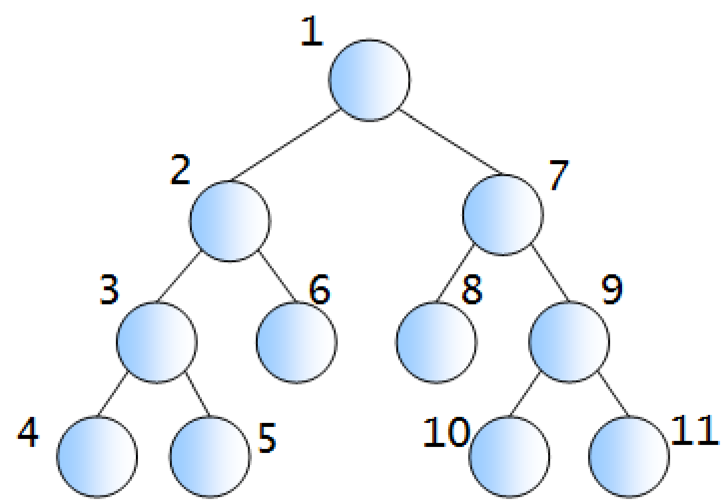


Preorder Traversal

- Procedure
  1. Visit the root node
  2. Visit the left subtree
  3. Visit the right subtree

```
preorder(x)
if x≠NULL
        print DATA(x);
        preorder(LEFT(x));
        preorder(RIGHT(x));
```

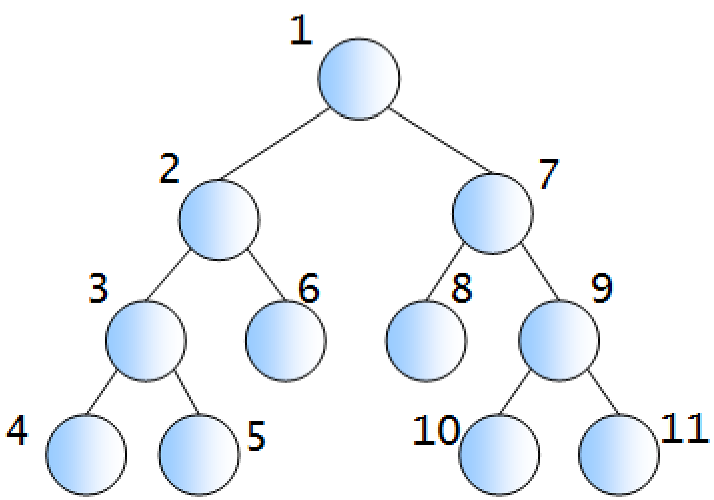




Preorder Traversal

- Call order of preorder traversal

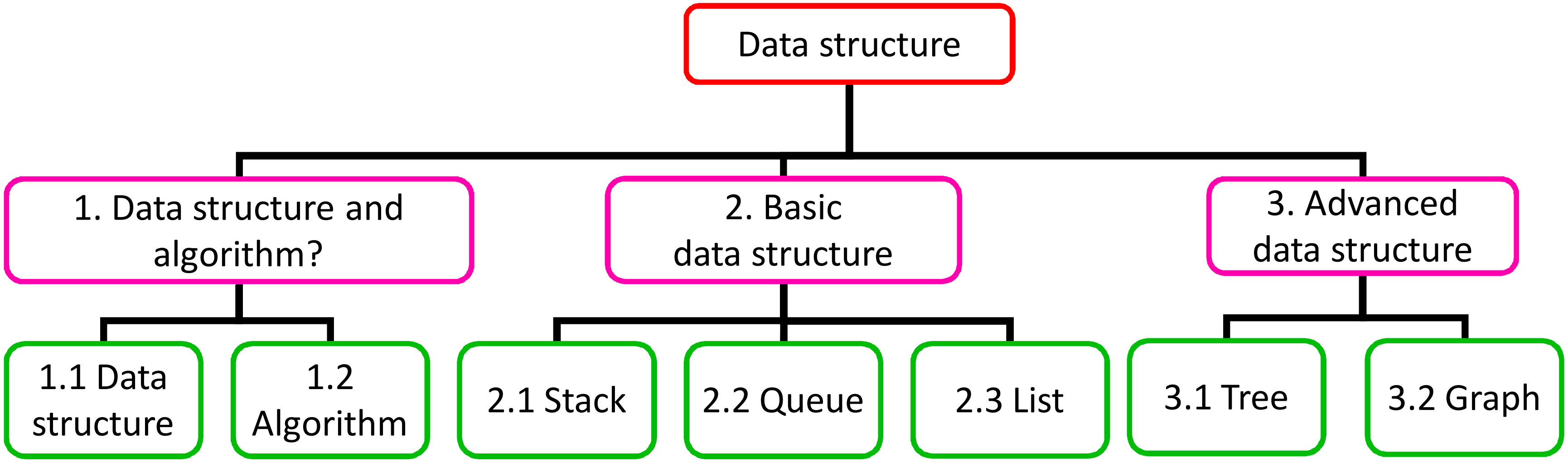
```
preorder(x)
if x≠NULL
        print DATA(x);
        preorder(LEFT(x));
        preorder(RIGHT(x));
```




```
preorder(n1)
    print(n1)
    preorder(n1->left: n2)
        print(n2)
        preorder(n2->left: n3)
            print(n3)
            preorder(n3->left: n4)
                print(n4)
                preorder(n4->left: null)
                preorder(n4->right: null)
            preorder(n3->right: n5)
                print(n5)
                preorder(n5->left: null)
                preorder(n5->right: null)
        preorder(n2->right: n6)
                print(n6)
                preorder(n6->left: null)
                preorder(n6->right: null)
    preorder(n1->right: n7)
        print(n7)
        preorder(n7->left: n8)
.....
```

Preorder Traversal

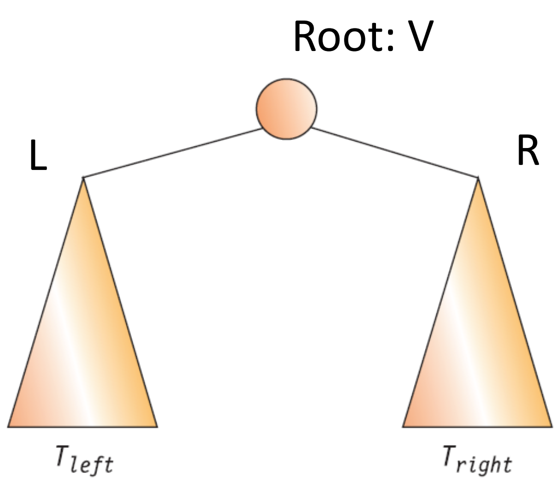
- Example) Output of structured documents

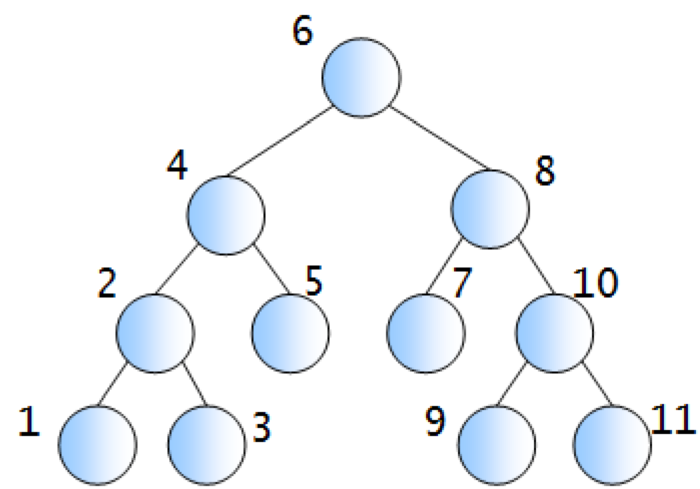


Inorder Traversal

- Procedure
  1. Visit the left subtree
  2. Visit the root node
  3. Visit the right subtree

```
inorder(x)
if x≠NULL
        inorder(LEFT(x));
        print DATA(x);
        inorder(RIGHT(x));
```



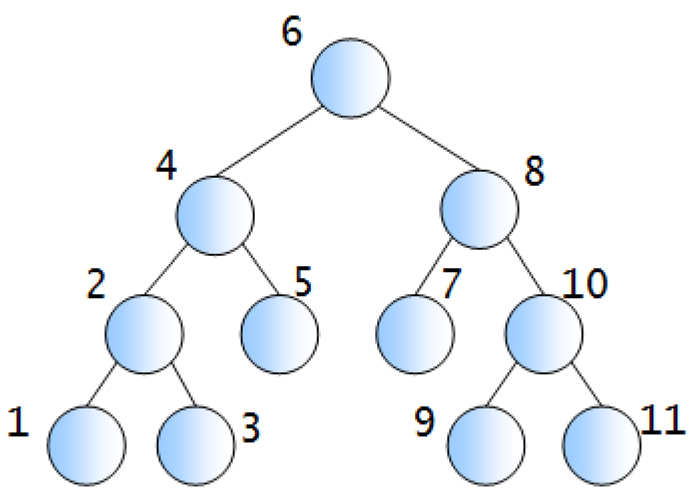


Inorder Traversal

- Call order of inorder traversal

```
inorder(x)
if x≠NULL
          inorder(LEFT(x));
          print DATA(x);
          inorder(RIGHT(x));
```

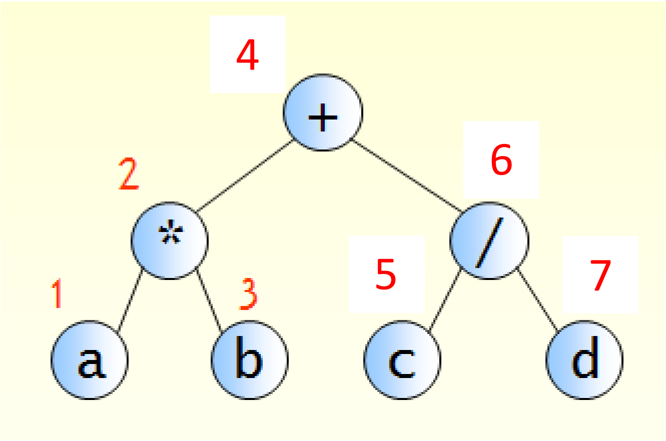
```
inorder(n6)
     inorder(n6->left: n4)
          inorder(n4->left: n2)
               inorder(n2->left: n1)
                    inorder(n1->left: null)
                    print(n1)
                    inorder(n1->right: null)
               print(n2)
               inorder(n2->right: n3)
                    inorder(n3->left: null)
                    print(n3)
                    inorder(n3->right: null)
          print(n4)
          inorder(n4->right: n5)
               inorder(n5->left: null)
               print(n5)
               inorder(n5->right: null)
     print(n6)
     inorder(n6->right: n8)
.....
```



이화여자대학교 EWHA WOMANS UNIVERSITY

Inorder Traversal

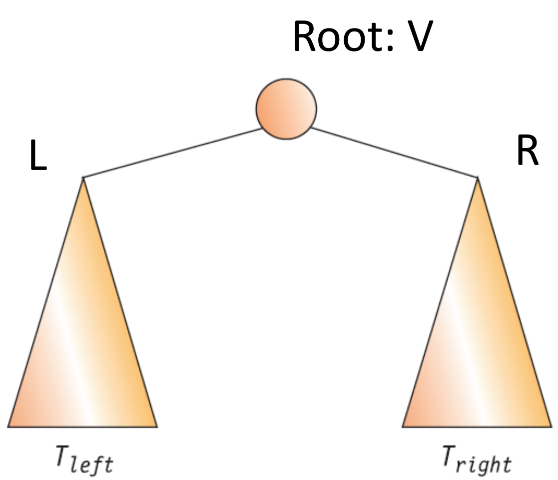
- Example) Formula tree

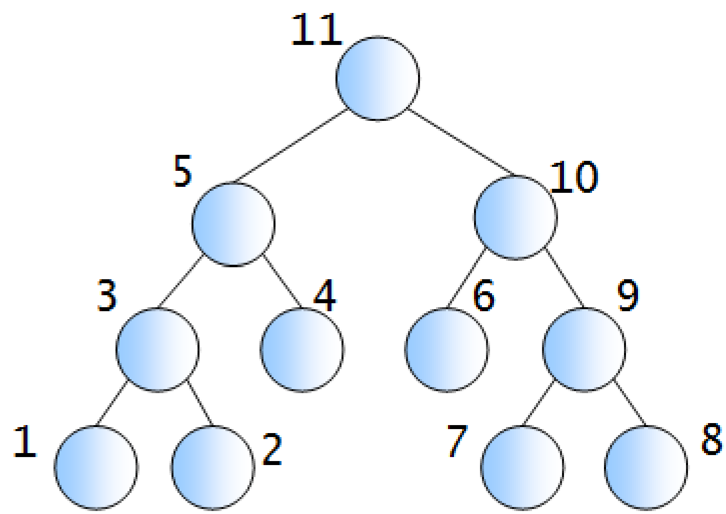


Postorder Traversal

- Procedure
  1. Visit the left subtree
  2. Visit the right subtree
  3. Visit the root node

```
postorder(x)
if x≠NULL
        postorder(LEFT(x));
        postorder(RIGHT(x));
        print DATA(x);
```

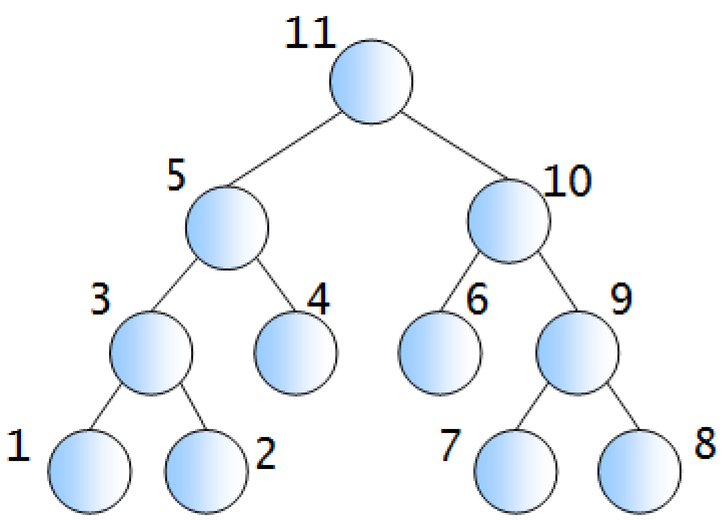




Postorder Traversal

- Call order of postorder traversal

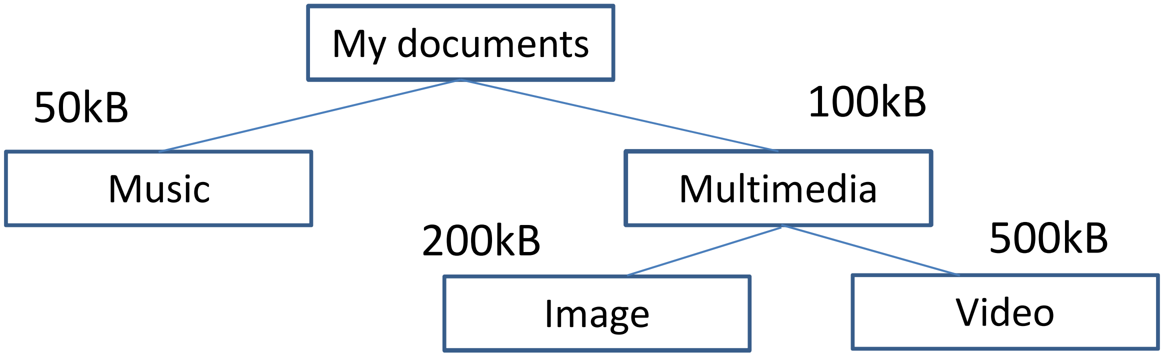
```
postorder(x)
if x≠NULL
           postorder(LEFT(x));
           postorder(RIGHT(x));
           print DATA(x);
```



?

Postorder Traversal

- Example) Calculation of directory size



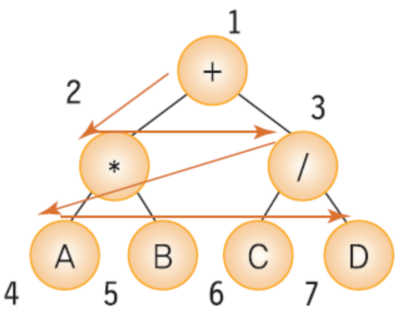
```c
TreeNode n1 = { 1, NULL, NULL };
TreeNode n2 = { 4, &n1, NULL };
TreeNode n3 = { 16, NULL, NULL };
TreeNode n4 = { 25, NULL, NULL };
TreeNode n5 = { 20, &n3, &n4 };
TreeNode n6 = { 15, &n2, &n5 };
TreeNode *root = &n6;

preorder(TreeNode *root) {
          if (root) {
                    printf("%d\n", root->data); // Visit root node
                    preorder(root->left); // Left subtree
                    preorder(root->right); // Right subtree
          }
}
inorder(TreeNode *root) {
          if (root) {
                    inorder(root->left); // Left subtree
                    printf("%d\n", root->data); // Visit root node
                    inorder(root->right);// Right subtree
          }
}
postorder(TreeNode *root) {
          if (root) {
                    postorder(root->left); // Left subtree
                    postorder(root->right); // Right subtree
                    printf("%d\n", root->data); // Visit root node
          }
}
void main()
{
          inorder(root);
          preorder(root);
          postorder(root);
}
```

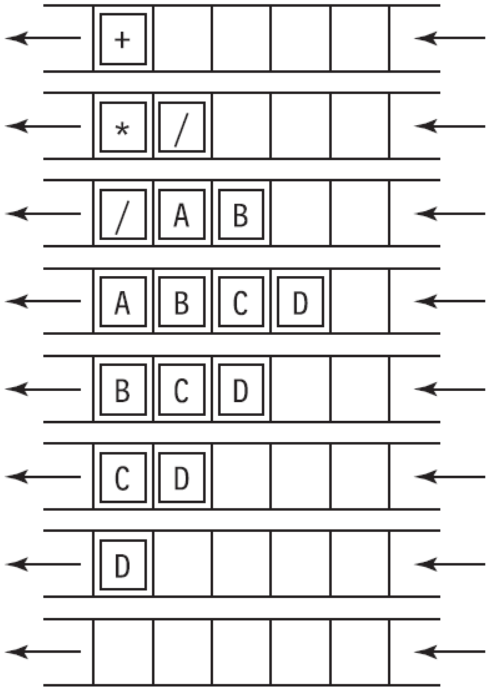
```c
typedef struct TreeNode {
          int data;
          struct TreeNode *left, *right;
} TreeNode;

//            15
//    4                 20
// 1               16     25
```

Level Traversal

- The level traversal is a method for visiting each node in order of level.

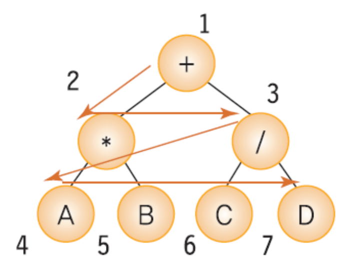
- Level traversal uses *queue*, whereas conventional traversal methods use *stack*.

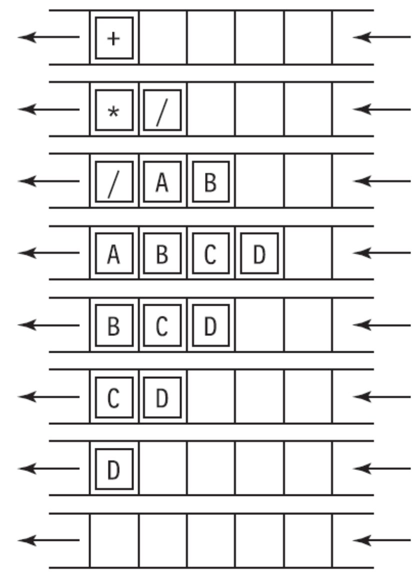




Level Traversal

```
level_order(root)

initialize queue;
if(root==NULL) then return;
enqueue(queue, root);
while is_empty(queue) != TRUE do
    x ← dequeue(queue);
    print x->data
    if(x->left != NULL)
            enqueue(queue, LEFT(x));
    if(x->right != NULL)
            enqueue(queue, RIGHT(x));
```





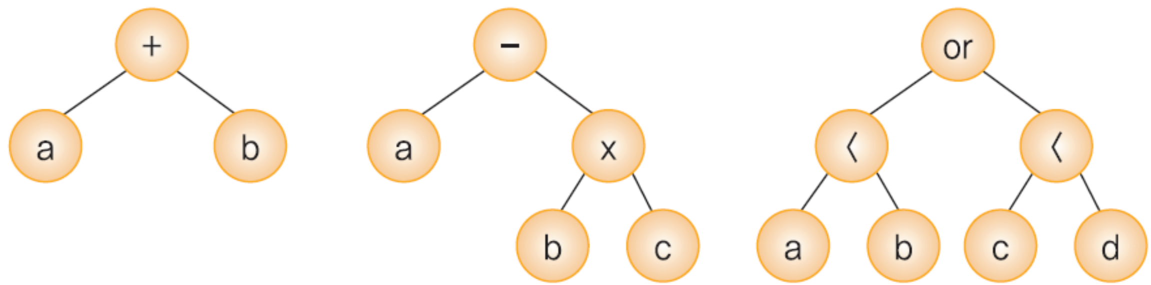
C code of ‘level_order(root)’:
will be a part of homework

```c
void level_order(TreeNode *ptr) {
	QueueType q;

	init(&q);
	if (ptr == NULL) return;

	enqueue(&q, ptr);
	while(!is_empty(&q)){
		ptr = dequeue(&q);
		printf("%d\n", ptr->data);
		if (ptr->left)
			enqueue(&q, ptr->left);
		if (ptr->right)
			enqueue(&q, ptr->right);
	}
}

void main()
{
	printf("level traversal\n");
	level_order(root);
	printf("\n");
}
```

```c
//        15
//    4        20
// 1        16    25
TreeNode n1 = { 1, NULL, NULL };
TreeNode n2 = { 4, &n1, NULL };
TreeNode n3 = { 16, NULL, NULL };
TreeNode n4 = { 25, NULL, NULL };
TreeNode n5 = { 20, &n3, &n4 };
TreeNode n6 = { 15, &n2, &n5 };
TreeNode *root = &n6;

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

typedef TreeNode * element;
typedef struct QueueNode {
	element item;
	struct QueueNode *link;
} QueueNode;

typedef struct QueueType {
	QueueNode *front;
	QueueNode *rear;
} QueueType;
```

Formula Tree

- Formula tree: represents an arithmetic equation as tree
  - Non-terminal node: operator
  - Terminal node: operand

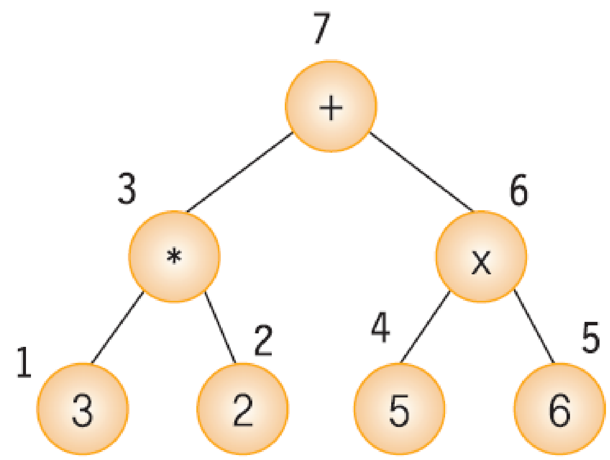


(a) (b) (c)

	a+b	a−(b × c)	(a<b) or (c<d)
Prefix	+ab	−a × bc	or<ab<cd
Infix	a+b	a−(b × c)	(a<b) or (c<d)
Postfix	ab+	abc × −	ab<cd<or

Formula Tree

- Calculation of formula tree
  - Using postorder traversal
  - Calculate the value of the subtree as a recursive call
  - When visiting a non-terminal node, the values of both subtrees are calculated using the operator stored in the node



Formula Tree

- Pseudo code

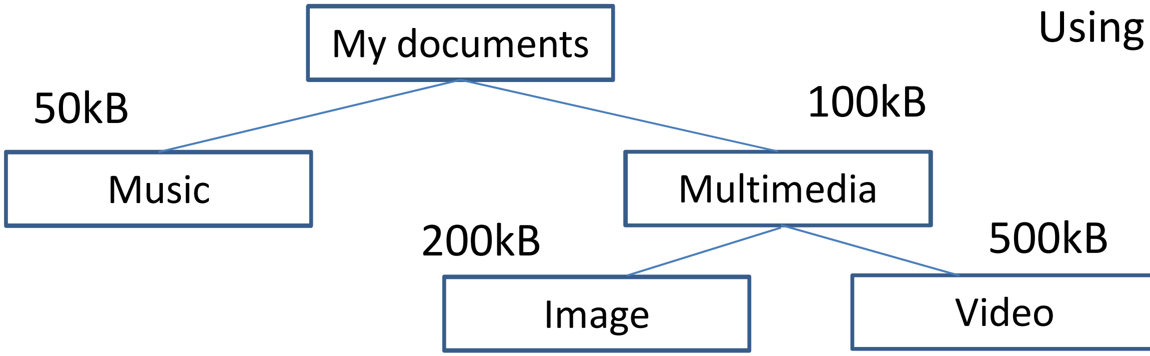
```
evaluate(exp)

if exp = NULL
        then return 0;
if exp->left = NULL and exp->right = NULL
        then return exp->data;
x ← evaluate(exp->left);
y ← evaluate(exp->right);
op ← exp->data;
return (x op y);
```

```c
typedef struct TreeNode {
        int data;
        struct TreeNode *left, *right;
} TreeNode;
//         +
//     *       +
//  1    4   16   25
```

```c
TreeNode n1 = { 1, NULL, NULL };
TreeNode n2 = { 4, NULL, NULL };
TreeNode n3 = { '*', &n1, &n2 };
TreeNode n4 = { 16, NULL, NULL };
TreeNode n5 = { 25, NULL, NULL };
TreeNode n6 = { '+', &n4, &n5 };
TreeNode n7 = { '+', &n3, &n6 };
TreeNode *exp = &n7;

int evaluate(TreeNode *root)
{
        if (root == NULL)
                return 0;
        if (root->left == NULL && root->right == NULL)
                return root->data;
        else {
                int op1 = evaluate(root->left);
                int op2 = evaluate(root->right);
                switch (root->data) {
                case '+':return op1 + op2;
                case '-':return op1 - op2;
                case '*':return op1*op2;
                case '/':return op1 / op2;
                }
        }
        return 0;
}
void main()
{
        printf("%d", evaluate(exp));
}
```

Calculation of Directory Size

Using postorder traversal

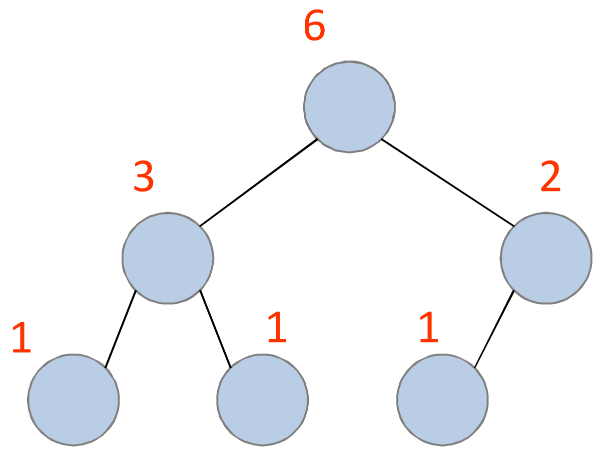



```c
int calc_dir_size(TreeNode *root)
{
            int left_size, right_size;
            if (root) {
                        left_size = calc_dir_size(root->left);
                        right_size = calc_dir_size(root->right);
                        return (root->data + left_size + right_size);
            }
            return 0
}
void main()
{
            TreeNode n4 = { 500, NULL, NULL };
            TreeNode n5 = { 200, NULL, NULL };
            TreeNode n3 = { 100, &n4, &n5 };
            TreeNode n2 = { 50, NULL, NULL };
            TreeNode n1 = { 0, &n2, &n3 };
            printf("Directory Size = %d\n", calc_direc_size(&n1));
}
```

Binary Tree Operation: Number of Nodes

- Calculate the number of nodes in the tree
- It recursively calls each subtree, adds 1 to the returned value, and returns

```
int get_node_count(TreeNode *node)
{
        int count = 0;
        if(node != NULL)
                count = 1 + get_node_count(node->left) + get_node_count(node->right);
        return count;
}
```



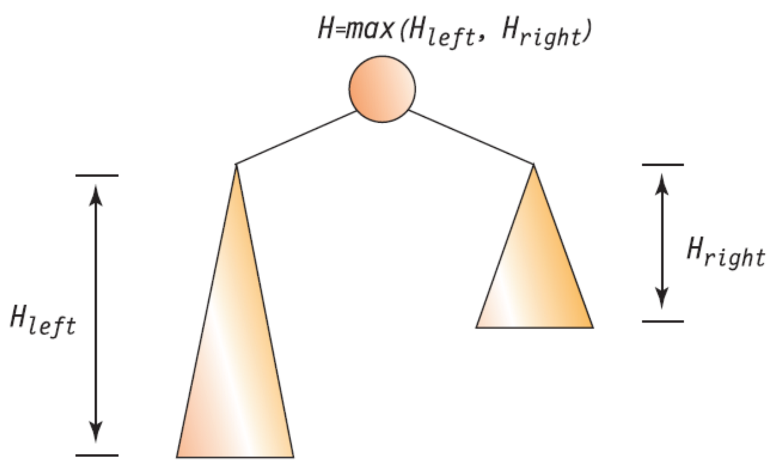
Binary Tree Operation: Number of Leaf Nodes

```
int get_leaf_count(TreeNode *node)
{
	int count = 0;
	if(node != NULL) {
		if( node->left==NULL && node->right==NULL )
			return 1;
		else
			count = get_leaf_count(node->left)+get_leaf_count(node->right)
	}
	return count;
}
```

Binary Tree Operation: Height

- Recursive call to the subtree, and returns the maximum value among the return values of the subtrees

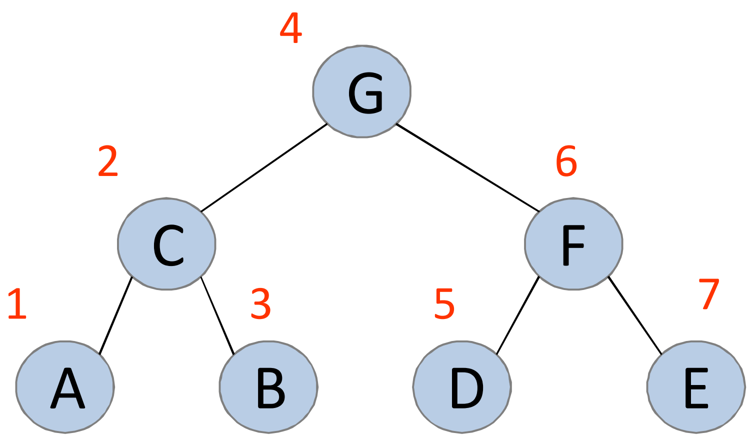
```c
int get_height(TreeNode *node)
{
        int height = 0;
        if (node != NULL)
        height = 1 + max(get_height(node->left), get_height(node->right));
        return height;
}
```



Predecessor/Successor in Binary Tree

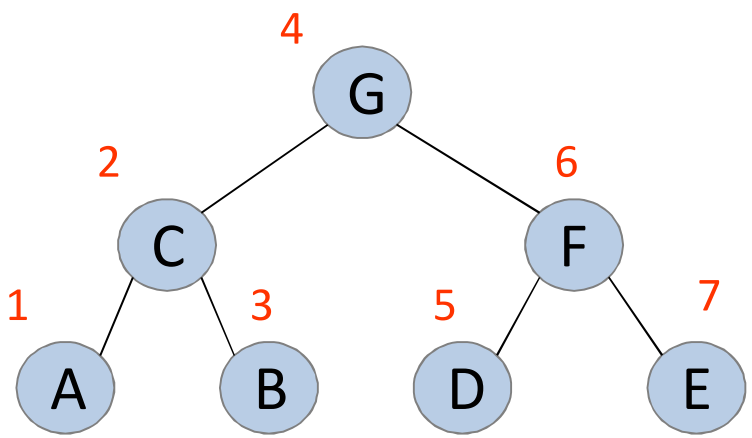
- Predecessor and successor are important in binary tree
  - They are defined depending on the type of traversal
  - For instance,
    inorder predecessor: previous node at the inorder traversal
    inorder successor: next node at the inorder traversal

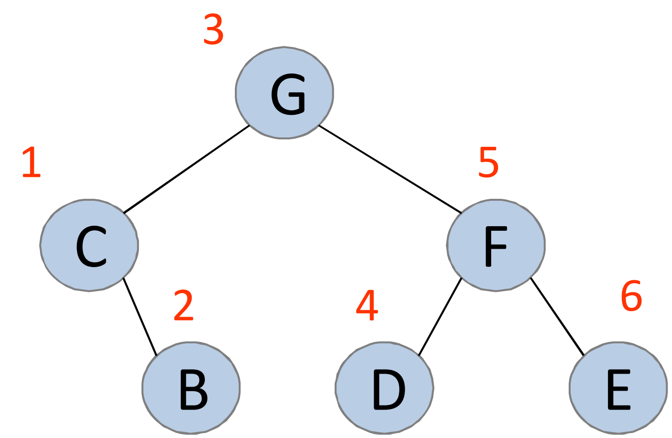
```
Node -> Predecessor

    A -> NULL
    C -> A
    B -> C
    G -> B
    D -> G
    F -> D
    E -> F
```

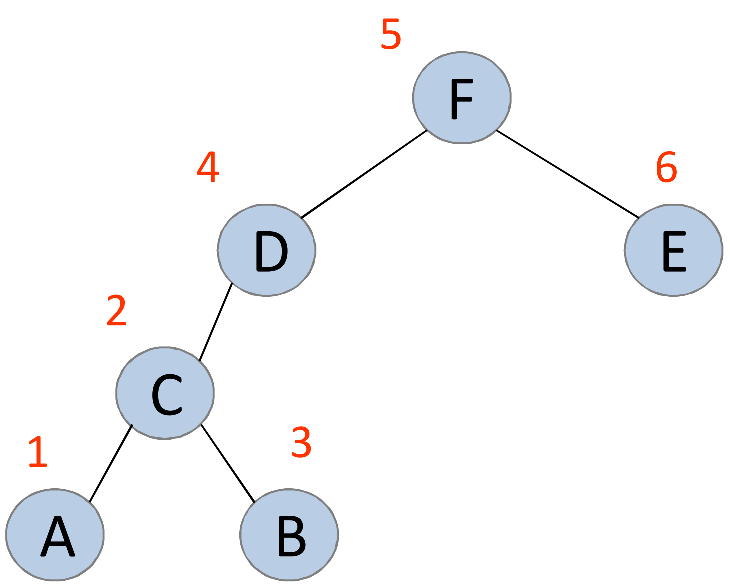


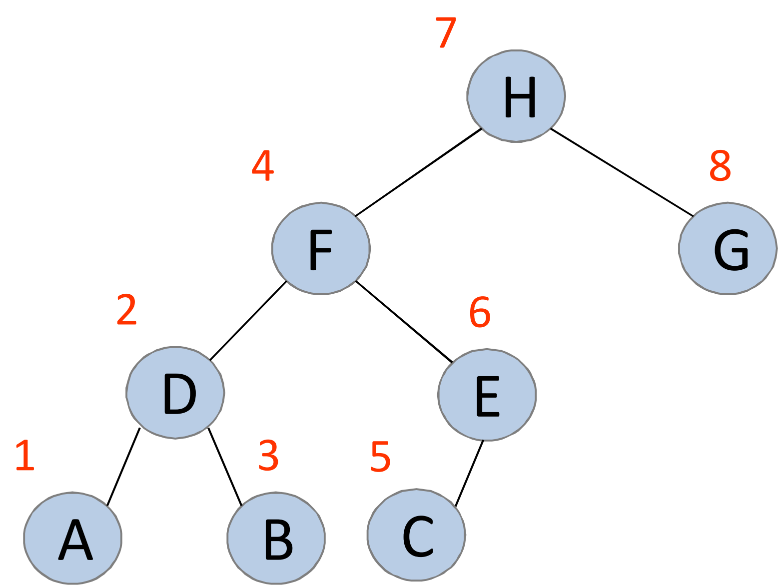

```
Node -> Successor

    A -> C
    C -> B
    B -> G
    G -> D
    D -> F
    F -> E
    E -> NULL
```

Question: how to find the successor in the inorder traversal?

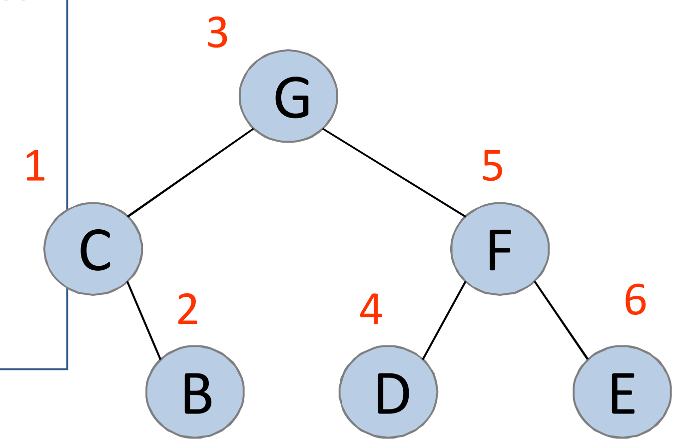


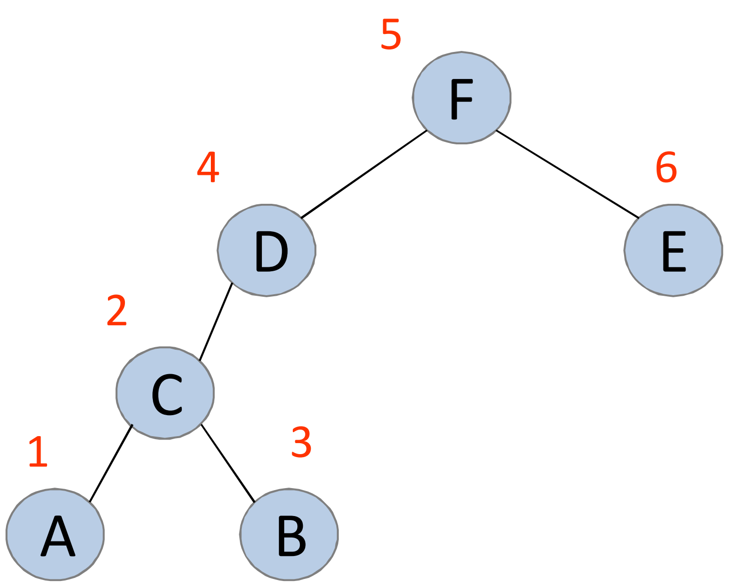


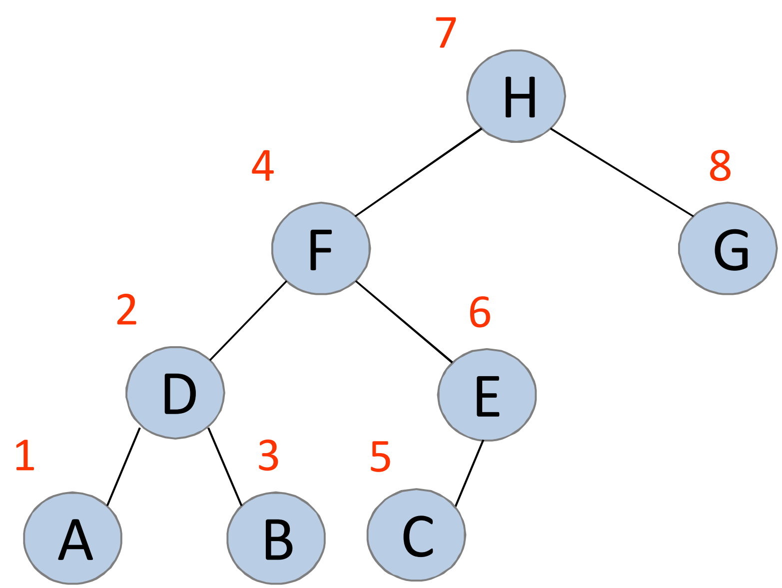




```
Tree_successor(x)
{
        if x->right != NULL //x's right subtree is not null
                return the leftmost node of right subtree

        //x's right subtree is null
        y = x->parent
        while (y != NULL and x == y->right) {
                x = y;
                y = y->parent;
        }
        return y;
}
```





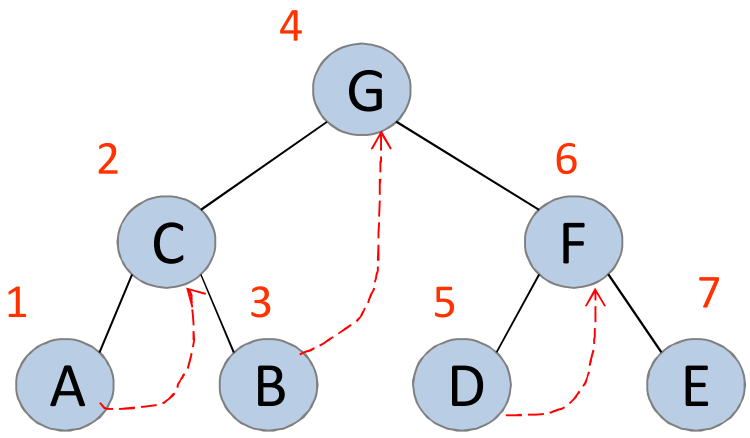


Threaded Binary Tree

- Problem of recursive traversal in binary tree
  - Recursive calls may be time-consuming for large scale tree.

⇨ To address the above issue, we can use the successor.

- Threaded binary tree
  - It saves the successor in the NULL link for traversal
  - Without recursive calls, we can traverse the nodes of the tree.



Example) inorder traversal

In the leaf nodes (A, B, D), their successors are stored in the *right links* (which are originally NULL).

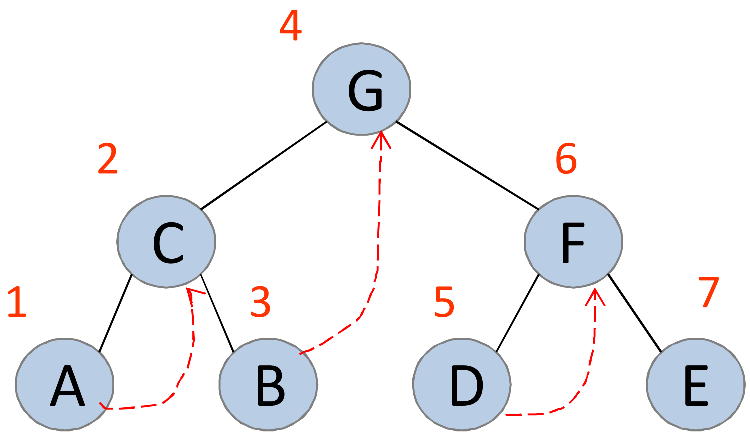
The leaf node (E) has no successor.

The right links of the nodes (C, G, F) are not null. Thus, compute their successors *on-the-fly*.

Threaded Binary Tree

- ‘is_thread’
  - To distinguish whether the links of nodes indicate the inorder successor or the child

```
typedef struct TreeNode {
        int data;
        struct TreeNode *left, *right;
        int is_thread; //TRUE, if right link is a thread
} TreeNode;
```

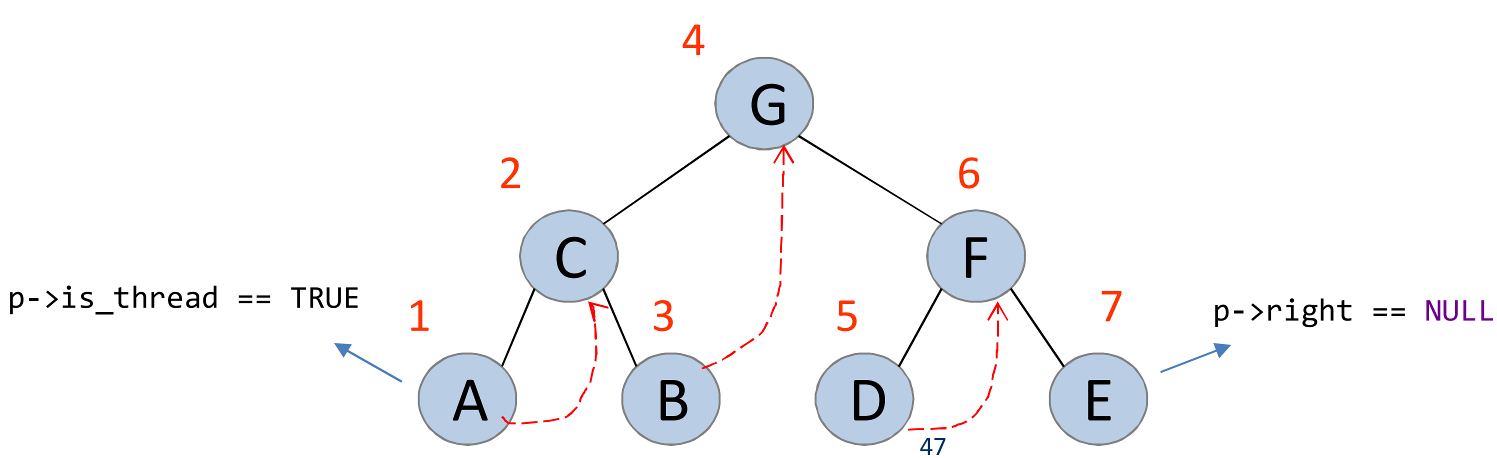



```
TreeNode n1 = { 'A', NULL, n3, 1 };
TreeNode n2 = { 'B', NULL, n7, 1 };
TreeNode n3 = { 'C', &n1, &n2, 0 };
TreeNode n4 = { 'D', NULL, n6, 1 };
TreeNode n5 = { 'E', NULL, NULL, 0 };
TreeNode n6 = { 'F', &n4, &n5, 0 };
TreeNode n7 = { 'G', &n3, &n6, 0 };
TreeNode *exp = &n7;
```

Threaded Binary Tree

- 'find_successor': function that finds the inorder successor

```
TreeNode *find_successor(TreeNode *p)
{
        // q: right pointer of p
        TreeNode *q = p->right;

        // Returns the right child if the right child is NULL or p is a thread
        if (q == NULL || p->is_thread == true)
                return q;

        // If the right child is not NULL, compute its successor on-the-fly
        // by moving to the leftmost node of the right subtree
        while (q->left != NULL) q = q->left;
        return q;
}
```



Threaded Binary Tree

- Iterative inorder traversal function using thread
  - Since the inorder traversal starts with the leftmost node, find the leftmost node first.

```
void thread_inorder(TreeNode *t)
{
		TreeNode *q;
		q = t;
		while (q->left) q = q->left;	// Go to the leftmost node
		do
		{
			printf("%c", q->data); // Output data
			q = find_successor(q); // Call the successor
		} while (q);			// If not null
}
```

```c
TreeNode *find_successor(TreeNode *p)
{
	// q: right pointer of p
	TreeNode *q = p->right;

	// Returns the right child if the right child is NULL or p is a thread
	if (q == NULL || p->is_thread == true)
		return q;

	// If the right child is not NULL, compute its successor on-the-fly
	// by moving to the leftmost node of the right subtree
	while (q->left != NULL) q = q->left;
	return q;
}
void thread_inorder(TreeNode *t)
{
	TreeNode *q;
	q = t;
	// Go to the leftmost node
	while (q->left) q = q->left;
	do
	{
		printf("%c\n", q->data); // Output data
		// Call the successor
		q = find_successor(q);
	} while (q);    // If not null
}
void main()
{
	//Set up thread using the successor
	n1.right = &n3;
	n2.right = &n7;
	n4.right = &n6;

	thread_inorder(exp);
}
```

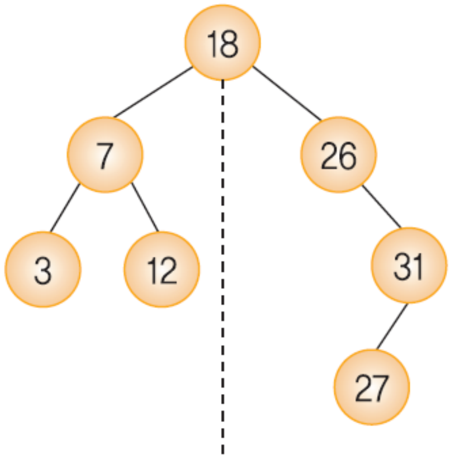
```c
typedef struct TreeNode {
int data;
struct TreeNode *left, *right;
int is_thread;
} TreeNode;

//         G
//     C       F
//   A   B   D   E
TreeNode n1 = { 'A', NULL, NULL, 1 };
TreeNode n2 = { 'B', NULL, NULL, 1 };
TreeNode n3 = { 'C', &n1, &n2, 0 };
TreeNode n4 = { 'D', NULL, NULL, 1 };
TreeNode n5 = { 'E', NULL, NULL, 0 };
TreeNode n6 = { 'F', &n4, &n5, 0 };
TreeNode n7 = { 'G', &n3, &n6, 0 };
TreeNode *exp = &n7;
```

Binary Search Tree

- Data structure for efficient search operation

```
key(left subtree) ≤ key(root node) ≤ key(right subtree)
```

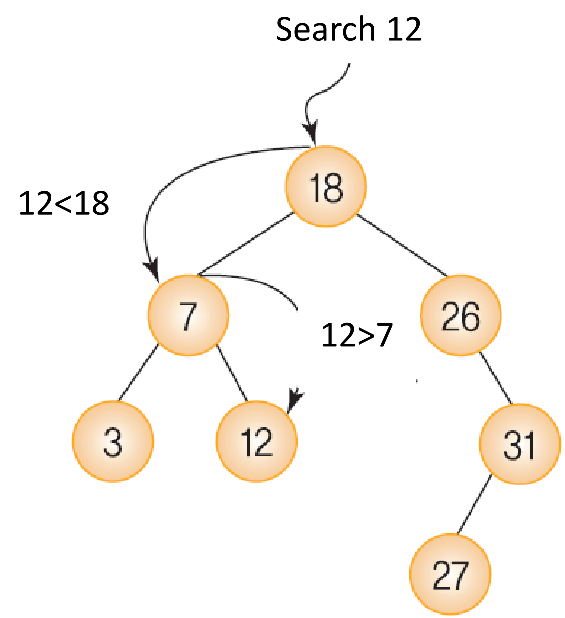
- You can get sorted values in ascending order through the inorder traversal



Search Operation in Binary Search Tree

- Three cases
  1. If the results are the same, the search ends successfully.
  2. If the given value < the value of the root node,
     the search restarts for the left child of this root node.
  3. If the given value > the value of the root node,
     the search restarts for the right child of this root node.

```
search(x, k)

if x = NULL
        then return NULL;
if k = x->key
        then return x;
else if k<x->key
        then return search(x->left, k);
else    return search(x->right, k);
```



Search Operation in Binary Search Tree

Recursion

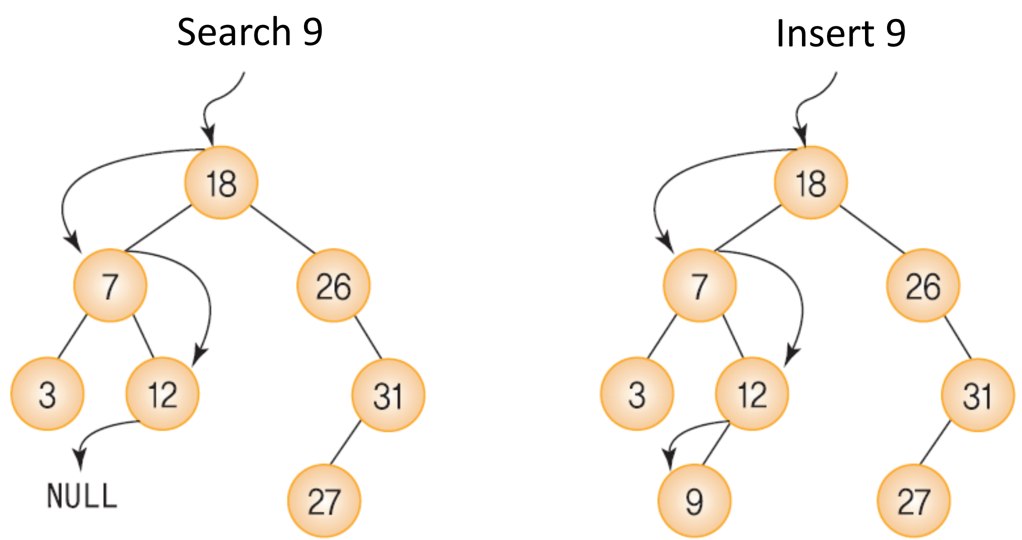
```
TreeNode  *search(TreeNode  *node, int  key)
{
        if (node == NULL)  return NULL;
        if (key == node->key) return node;
        else if (key < node->key)
                return  search(node->left, key);
        else
                return  search(node->right, key);
}
```

Iteration

```
TreeNode *search(TreeNode *node, int key)
{
        while (node != NULL) {
                if (key == node->key) return node;
                else if (key < node->key)
                        node = node->left;
                else
                        node = node->right;
        }
        return NULL;
}
```

Insertion in Binary Search Tree

- In order to insert an element into the binary search tree, it is necessary to perform the search first
  - The binary search tree should not contain the node with the same key value.
  - The location where the search failed is the location where the new node is inserted.



(a) (b)

Insertion in Binary Search Tree

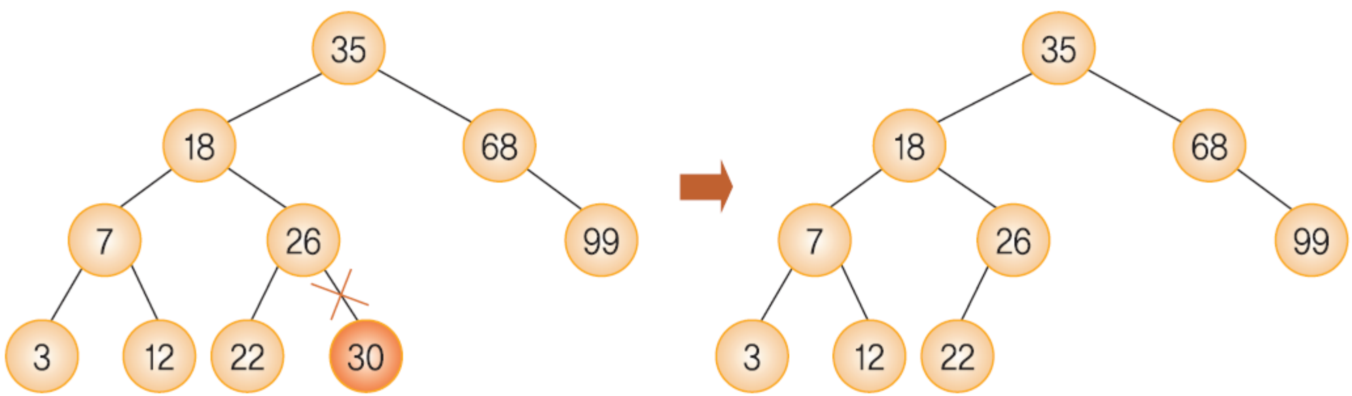
```
insert_node(T,z)

p ← NULL;
t ← root;
while t!=NULL do
        p ← t;
        if z->key < p->key
                then t ← p->left;
        else t ← p->right;
z ← make_node(key);      // create the node to be inserted
if p=NULL
        then root ← z;   // if tree is empty
else if z->key < p->key
        then p->left ← z
else p->right ← z
```

```c
// Insert the key into the binary search tree root.
// If key is already in root, it is not inserted.
void insert_node(TreeNode **root, int key)
{
        TreeNode *p, *t;// p: parent node, t: current node
        TreeNode *n;// n: new node to be inserted
        t = *root;
        p = NULL;
        // Search first
        while (t != NULL) {
                if (key == t->key) {
                        printf("The same key exists in the tree.\n");
                        return;
                }
                p = t;
                if (key < t->key) t = t->left;
                else t = t->right;
        }
        // Since the key is not in the tree, insertion is possible.
        n = (TreeNode *)malloc(sizeof(TreeNode));
        if (n == NULL) return;
        n->key = key;
        n->left = n->right = NULL;

        if (p != NULL)
                if (key < p->key)
                        p->left = n;
                else p->right = n;
        else *root = n;
}
```

Deletion in Binary Search Tree

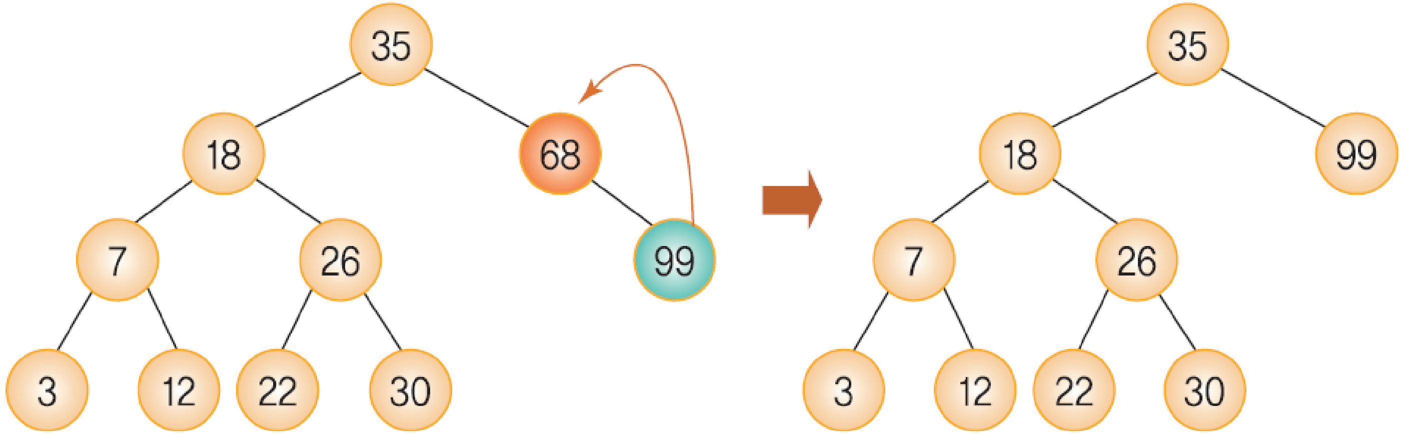
- Three cases
  1. If the node to be deleted is a leaf node
  2. If the node to be deleted has only one left or right subtree
  3. If the node to be deleted has both subtrees

- CASE 1: If the node to be deleted is a leaf node
  Find the parent node of the leaf node and disconnect it.



Deletion in Binary Search Tree

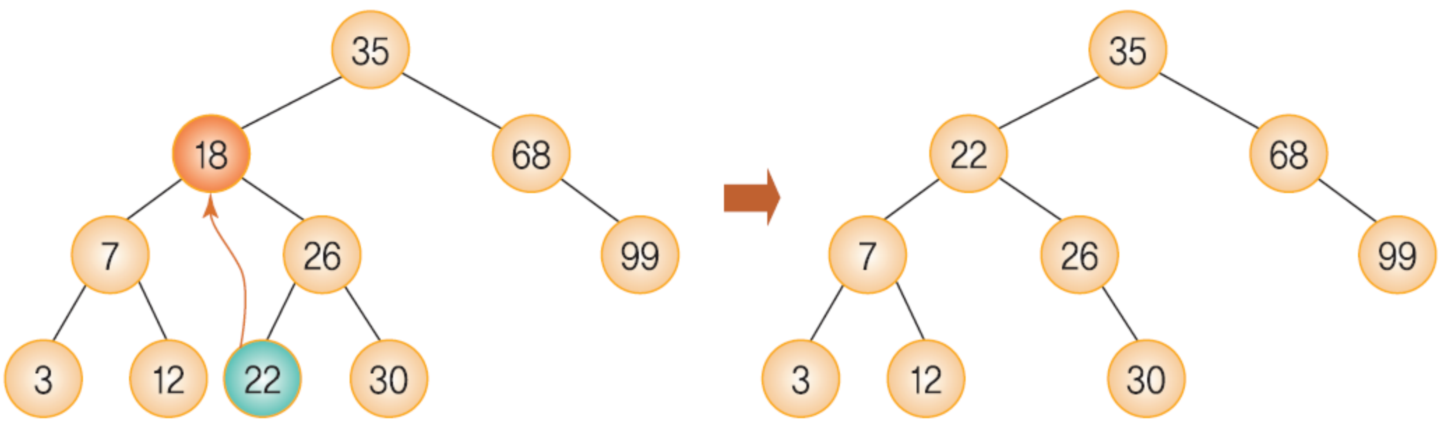
- CASE 2: If the node to be deleted has only one left or right subtree
  The node is deleted and the subtree is attached to the parent node.



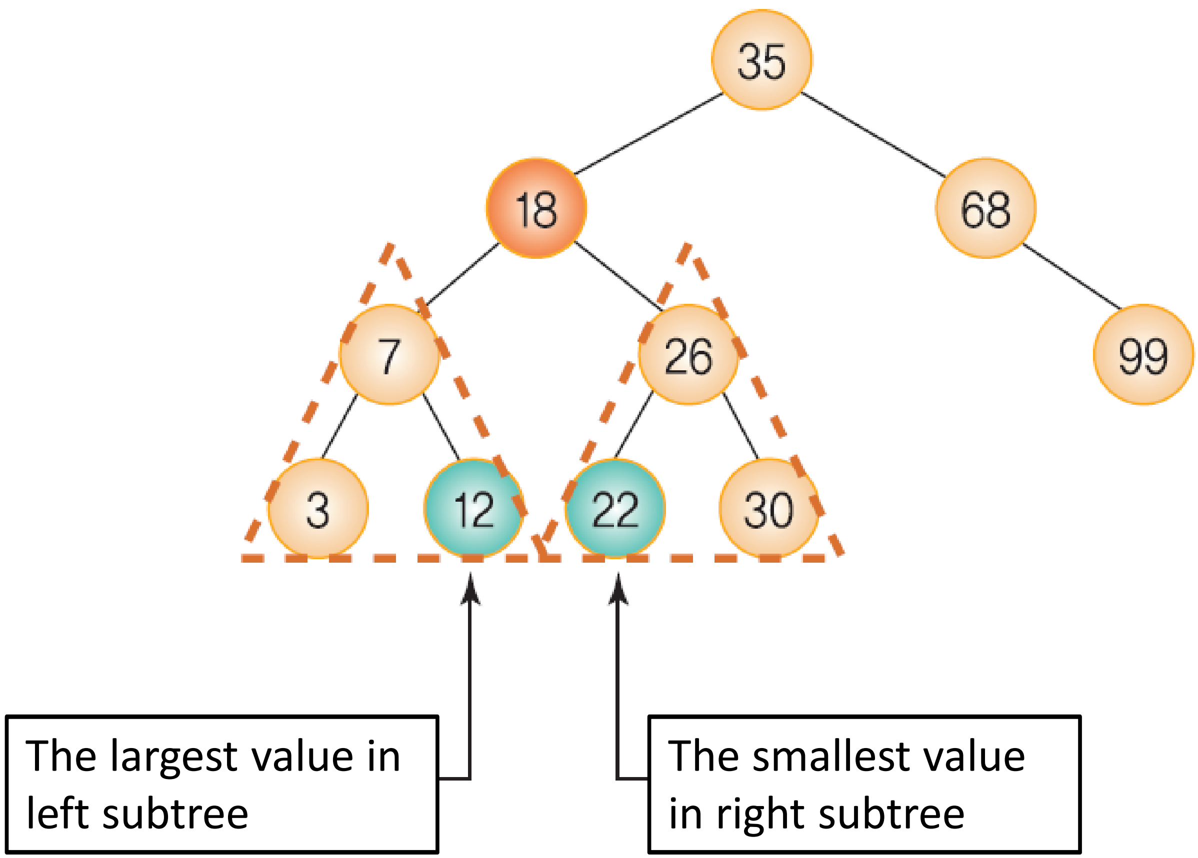
Deletion in Binary Search Tree

- CASE 3: If the node to be deleted has both subtrees

The predecessor or successor are brought to the delete node position.



Deletion in Binary Search Tree

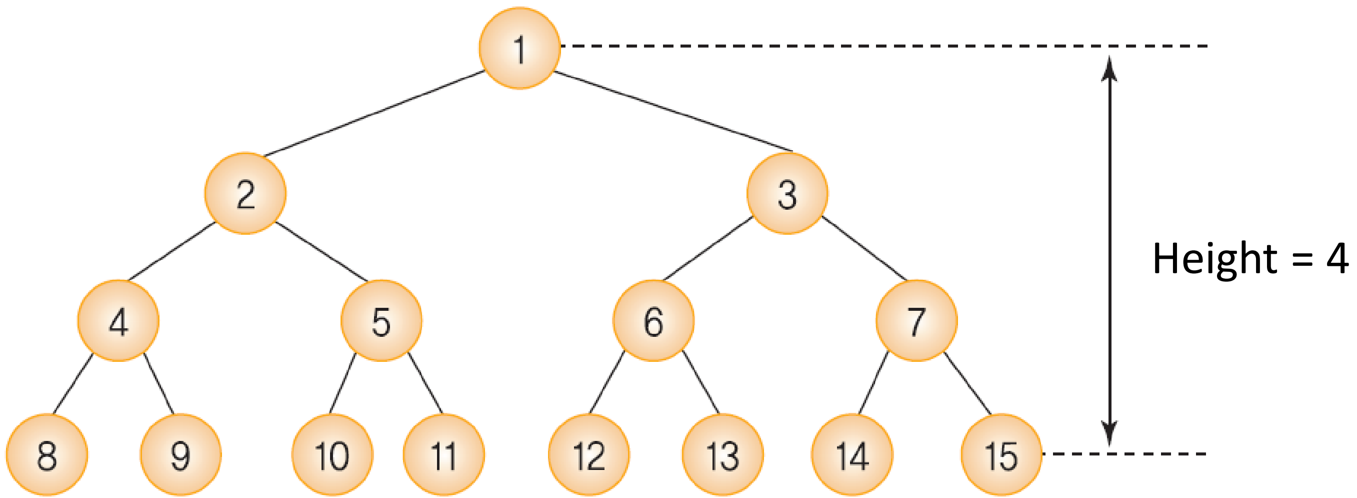


```c
// Deletion in binary search tree
void delete_node(TreeNode **root, int key)
{
	TreeNode *p, *child, *succ, *succ_p, *t;
	// search node t with key, p: t's parent
	p = NULL;
	t = *root;
	while (t != NULL && t->key != key) {
		p = t;
		t = (key < t->key) ? t->left : t->right;
	}
	// If t is NULL at the end, there is no key in the tree
	if (t == NULL) {
		printf("key is not in the tree");
		return;
	}

	// Case 1
	if ((t->left == NULL) && (t->right == NULL)) {
		if (p != NULL) {
			if (p->left == t)
				p->left = NULL;
			else p->right = NULL;
		}
		// If the parent node is NULL, the node to be deleted is the root
		else
			*root = NULL;
	}
```

```
        // Case 2
        else if ((t->left == NULL) || (t->right == NULL)) {
                child = (t->left != NULL) ? t->left : t->right;
                if (p != NULL) {
                        if (p->left == t)
                                p->left = child;
                        else p->right = child;
                }
                // If the parent node is NULL, the node to be deleted is the root
                else
                        *root = child;
        }
        // Case 3
        else {
                // Find the successor at right subtree
                succ_p = t;
                succ = t->right;
                // Keep moving to the left and find the successor
                while (succ->left != NULL) {
                        succ_p = succ;
                        succ = succ->left;
                }
                if (succ_p->left == succ)
                        succ_p->left = succ->right;
                else
                        succ_p->right = succ->right;
                t->key = succ->key;
                t = succ;
        }
        free(t);
}
```

Performance Analysis in Binary Search Tree

- The time complexity of the search, insertion, and deletion in the binary search tree is proportional to the tree height h



Performance Analysis in Binary Search Tree

- The best case
  - If the binary tree is balanced
  - $h = \log_2 n$

- The worst case
  - For one-sided, oblique binary trees
  - $h = n$
  - The time complexity are the same as that of sequential search.