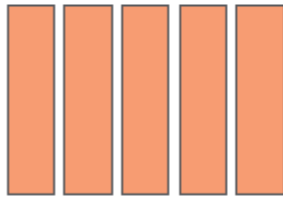


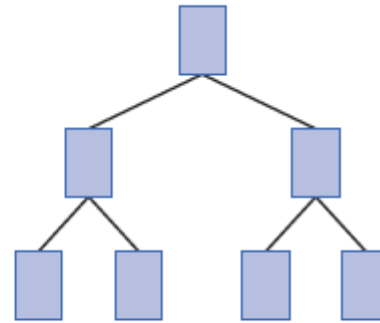
배
장
트
리

트리(TREE)

- 리스트, 스택, 큐 등은 선형 구조
- 트리: 계층적인 구조를 나타내는 자료구조



선형 자료구조

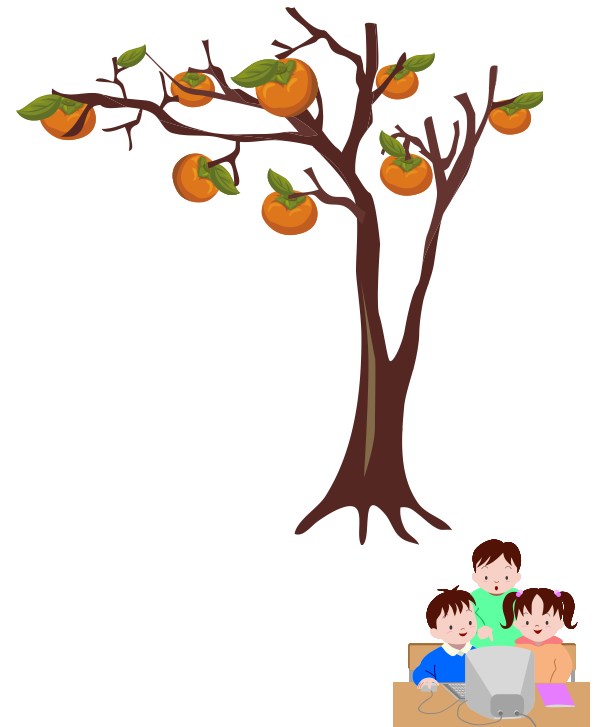


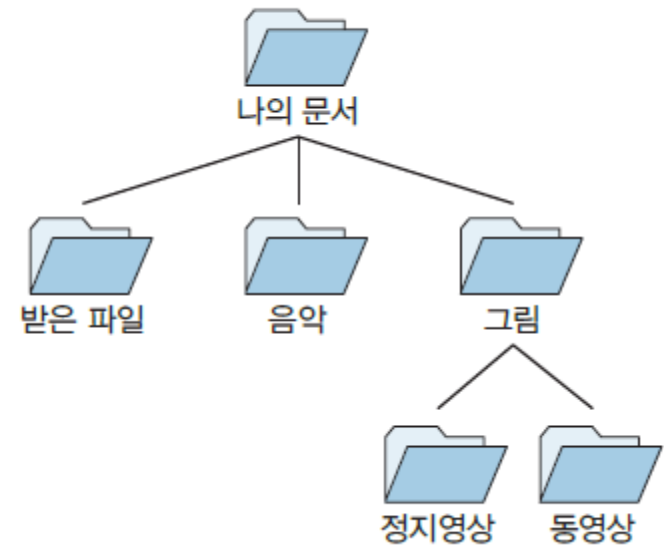
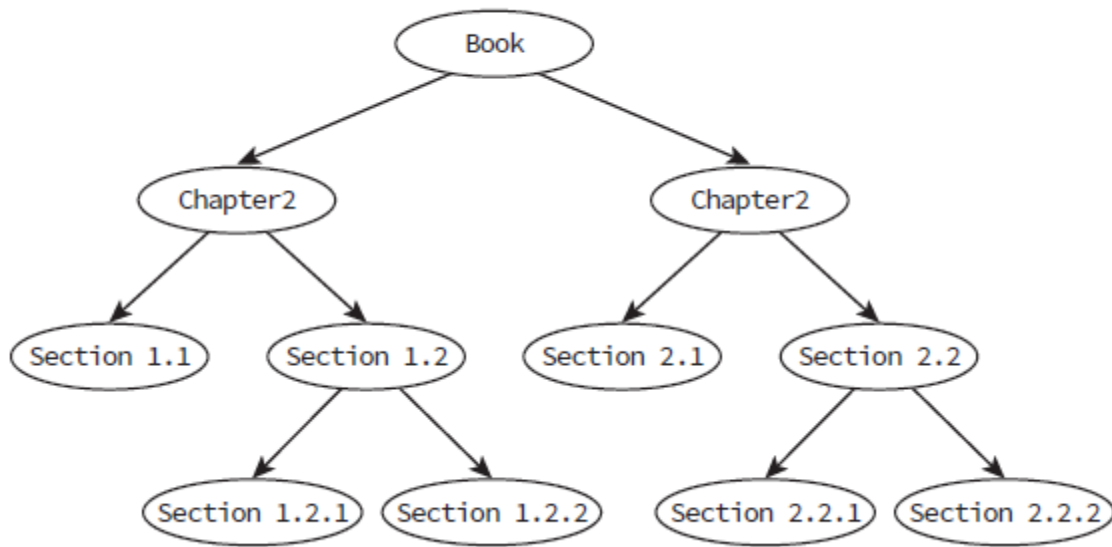
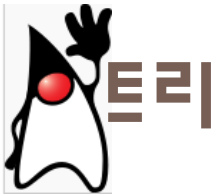
비선형 자료구조



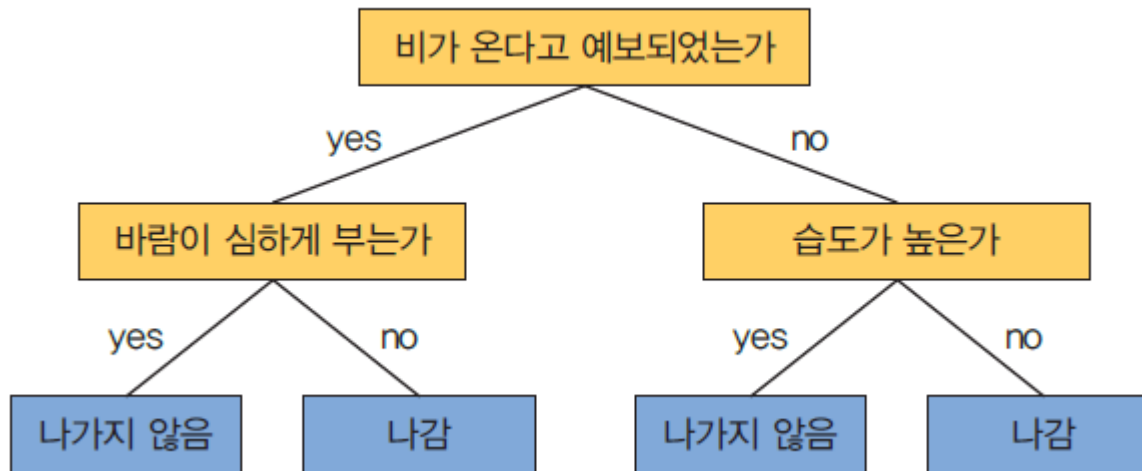
트리(TREE)

- 트리는 부모-자식 관계의 노드들로 이루어진다.
- 응용분야:
 - ▣ 계층적인 조직 표현
 - ▣ 컴퓨터 디스크의 디렉토리 구조
 - ▣ 인공지능에서의 결정트리 (decision tree)



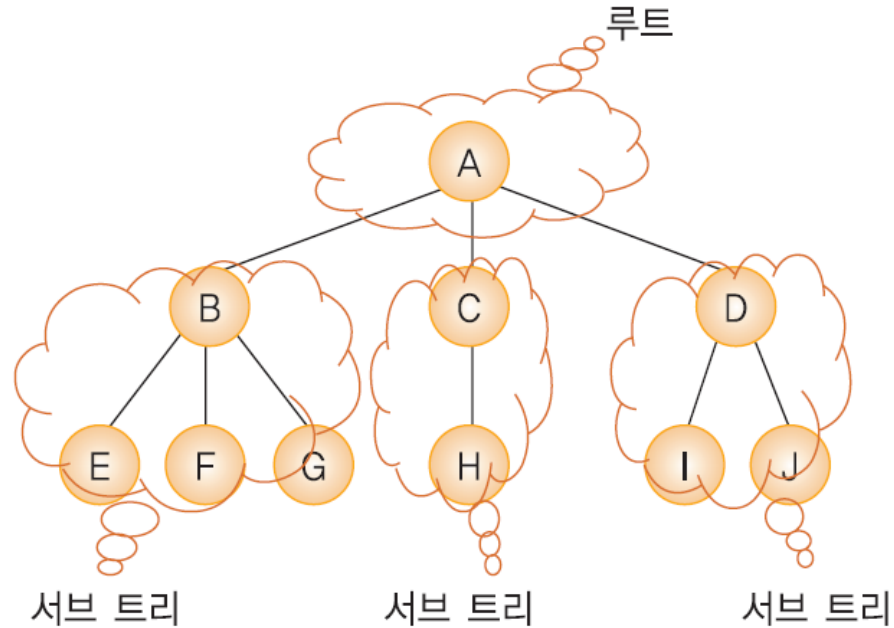


□ (예) 골프에 대한 결정 트리





트리의 용어



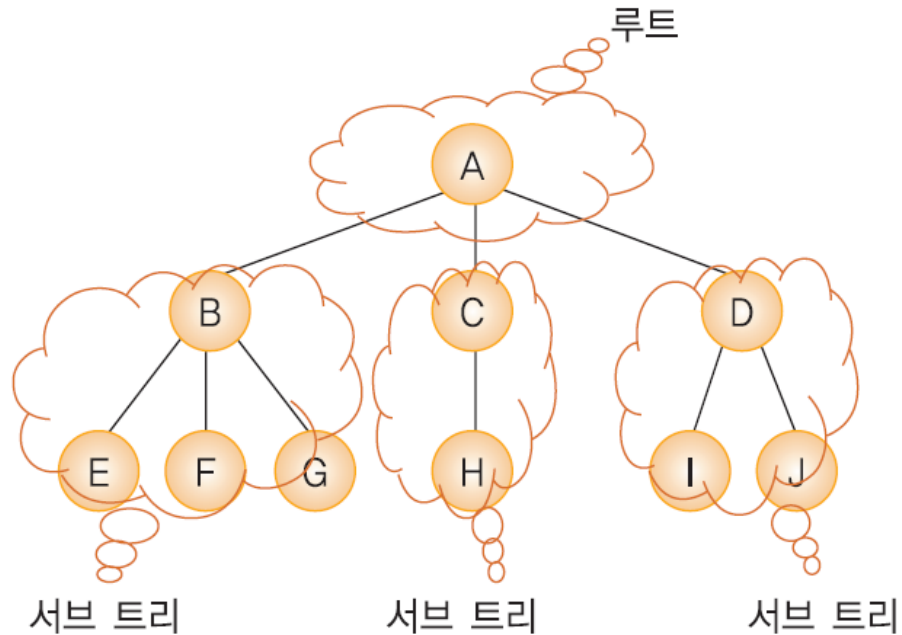
- 노드(node): 트리의 구성요소
- 루트(root): 부모가 없는 노드(A)
- 서브트리(subtree): 하나의 노드와 그 노드들의 자손들로 이루어진 트리

차수 개수의 제약을 뒤야 구현이 가능하다
ex. 이진트리





트리의 용어

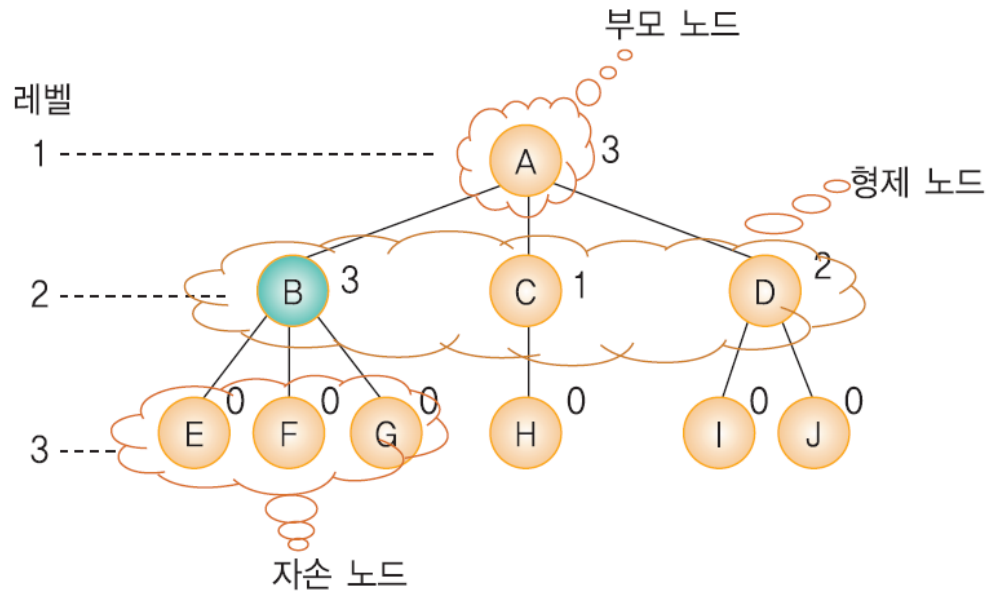


- 단말노드(**terminal node**): 자식이 없는 노드(A,B,C,D)
- 비단말노드: 적어도 하나의 자식을 가지는 노드(E,F,G,H,I,J)



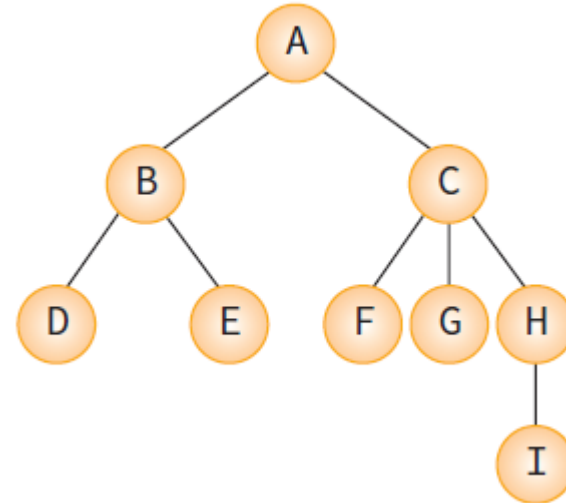


트리의 용어



- 자식, 부모, 형제, 조상, 자손 노드: 인간과 동일
- 레벨(level): 트리의 각층의 번호
- 높이(height): 트리의 최대 레벨(3)
- 차수(degree): 노드가 가지고 있는 자식 노드의 개수



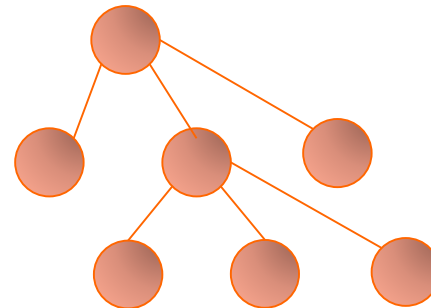
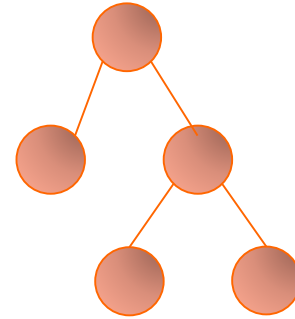
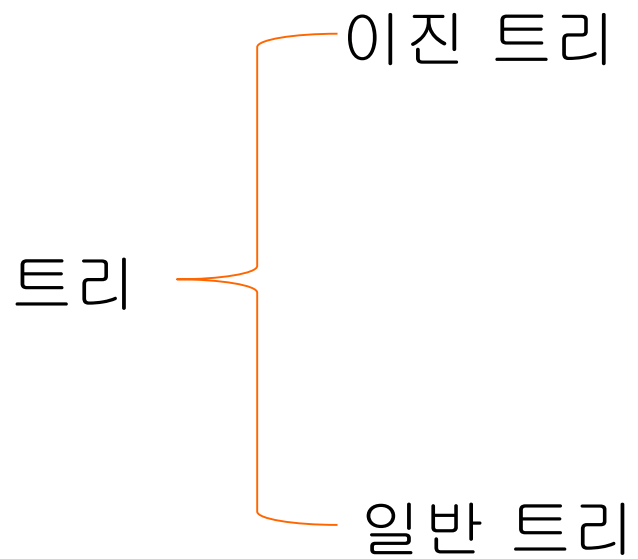


- A는 루트 노드이다.
- B는 D와 E의 부모노드이다.
- C는 B의 형제 노드이다.
- D와 E는 B의 자식노드이다.
- B의 차수는 2이다.
- 위의 트리의 높이는 4이다.





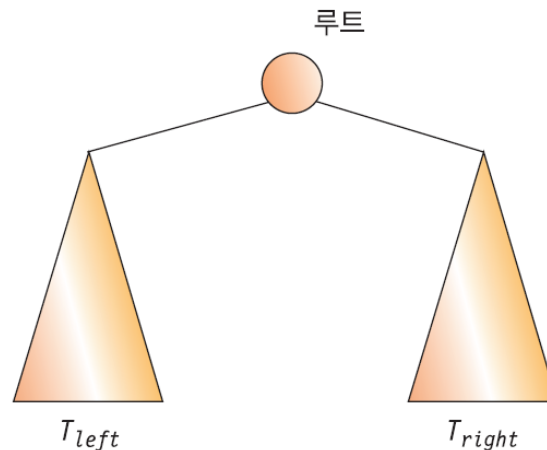
트리의 종류





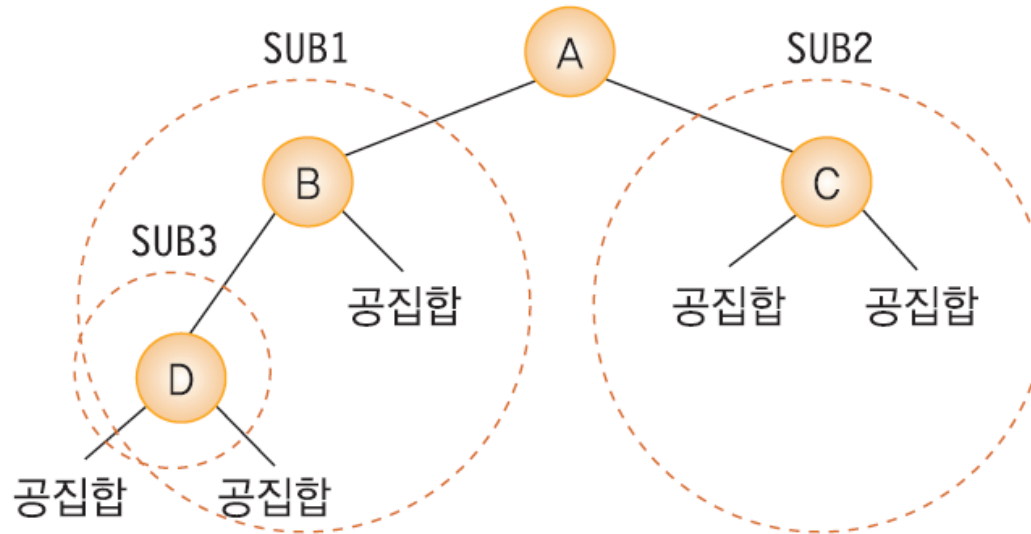
이진 트리 (binary tree)

- 이진 트리(binary tree) : 모든 노드가 2개의 서브 트리를 가지고 있는 트리
 - ▣ 서브트리는 공집합일수 있다.
- 이진트리의 노드에는 최대 2개까지의 자식 노드가 존재
- 모든 노드의 차수가 2 이하가 된다-> 구현하기가 편리함
- 이진 트리에는 서브 트리간의 순서가 존재





이진 트리 검증



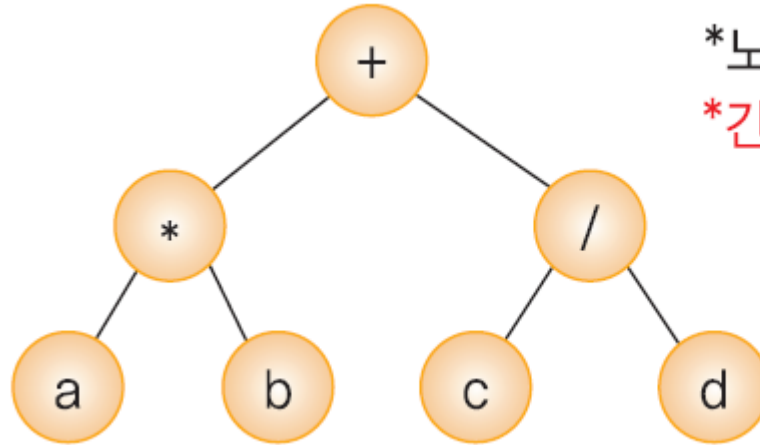
- 이진 트리는 공집합이거나
- 루트와 왼쪽 서브 트리, 오른쪽 서브 트리로 구성된 노드들의 유한 집합으로 정의된다. 이진 트리의 서브 트리들은 모두 이진 트리이어야 한다.





이진 트리의 성질

- 노드의 개수가 n 개이면 간선의 개수는 $n-1$



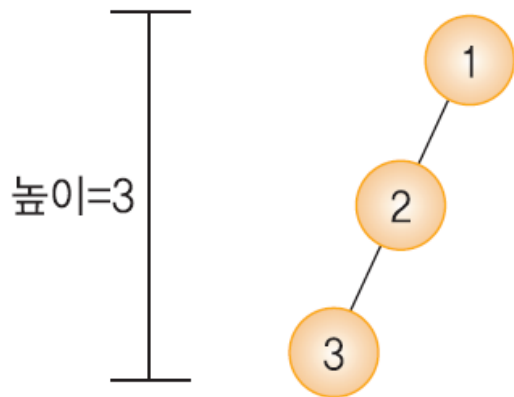
*노드의 개수: 7

*간선의 개수: 6

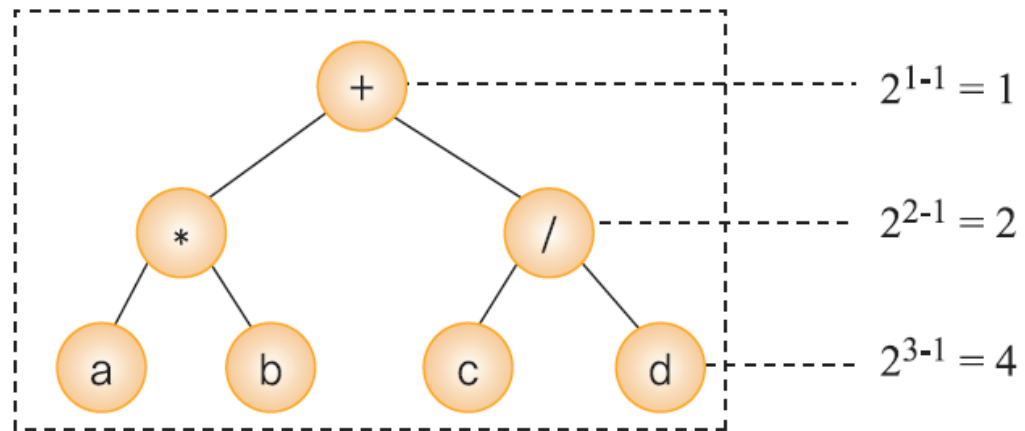


이진트리의 성질

- 높이가 h 인 이진트리의 경우, 최소 h 개의 노드를 가지며 최대 $2^h - 1$ 개의 노드를 가진다.



최소 노드 개수 = 3



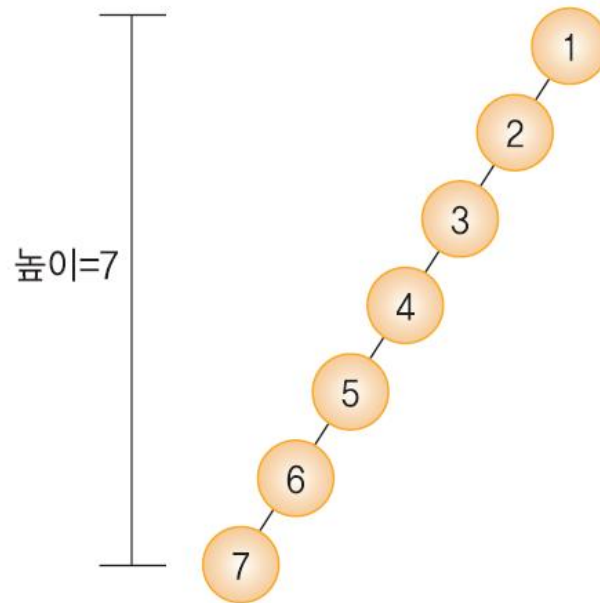
최대 노드 개수 = $2^{1-1} + 2^{2-1} + 2^{3-1} = 1 + 2 + 4 = 7$



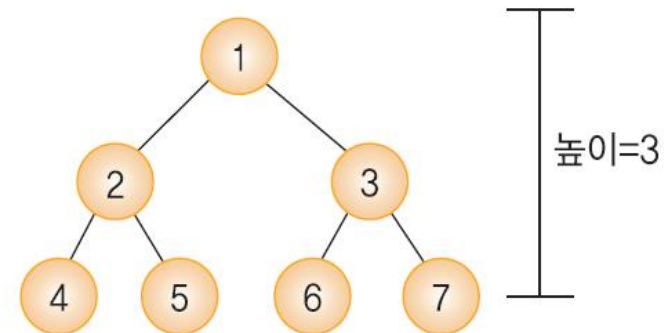


이진 트리의 성질

- n 개의 노드를 가지는 이진트리의 높이
 - ▣ 최대 n
 - ▣ 최소 $\lceil \log_2(n+1) \rceil$



(a) 최대 높이

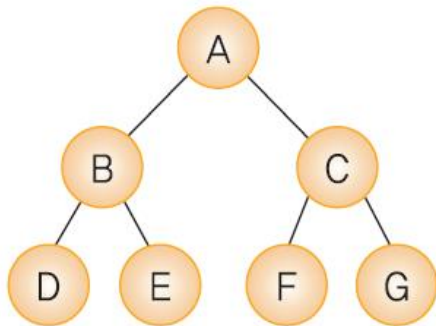


(b) 최소 높이

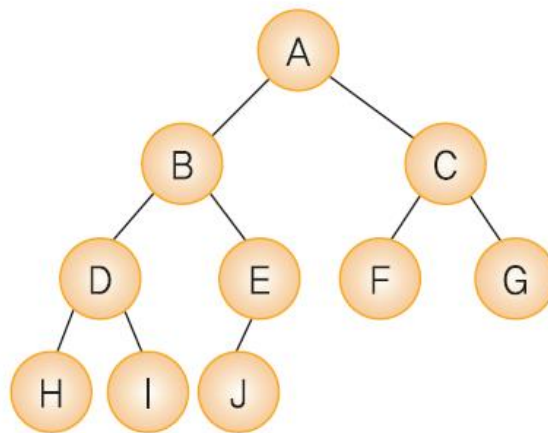


이진 트리의 분류

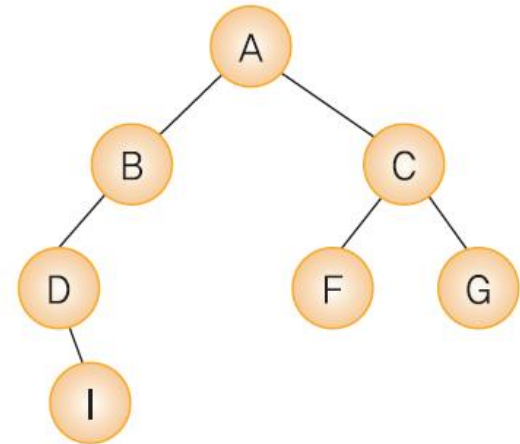
- 포화 이진 트리(full binary tree)
- 완전 이진 트리(complete binary tree)
- 기타 이진 트리



(a) 포화 이진 트리



(b) 완전 이진 트리



(c) 기타 이진 트리



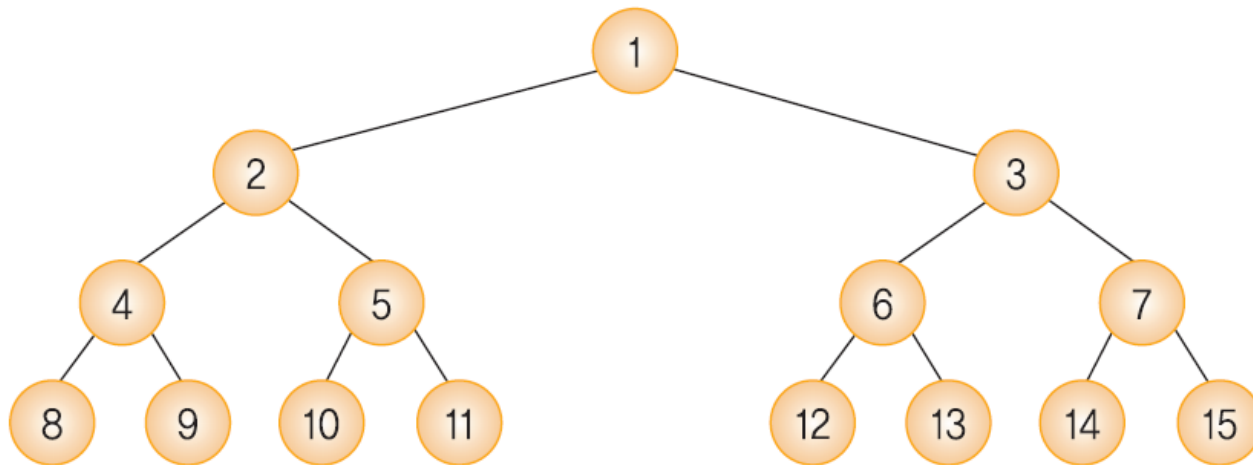


포화 이진 트리

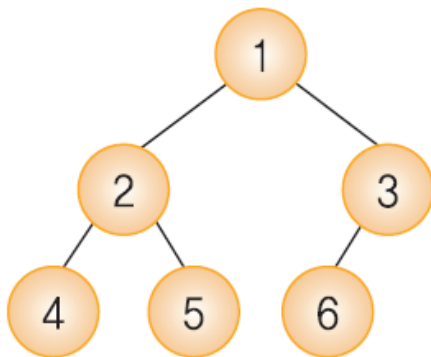
- 용어 그대로 트리의 각 레벨에 노드가 꽉 차있는 이진트리를 의미한다.

전체 노드 개수 : $2^{1-1} + 2^{2-1} + 2^{3-1} + \dots + 2^{k-1} = \sum_{i=0}^{k-1} 2^i = 2^k - 1$

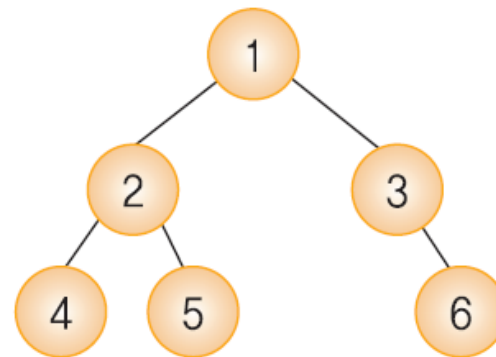
- 포화 이진 트리에는 다음과 같이 각 노드에 번호를 붙일 수 있다.



- 완전 이진 트리(**complete binary tree**): 레벨 1부터 $k-1$ 까지는 노드가 모두 채워져 있고 마지막 레벨 k 에서는 왼쪽부터 오른쪽으로 노드가 순서대로 채워져 있는 이진트리
- 포화 이진 트리와 노드 번호가 일치



(a) 완전 이진 트리



(b) 완전 이진 트리가 아님





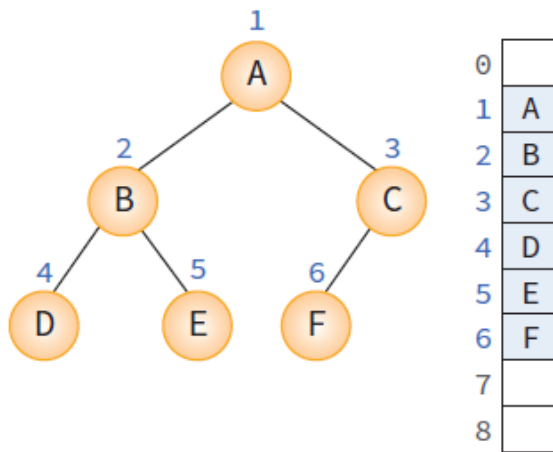
이진트리의 표현

- 배열을 이용하는 방법
- 포인터를 이용하는 방법

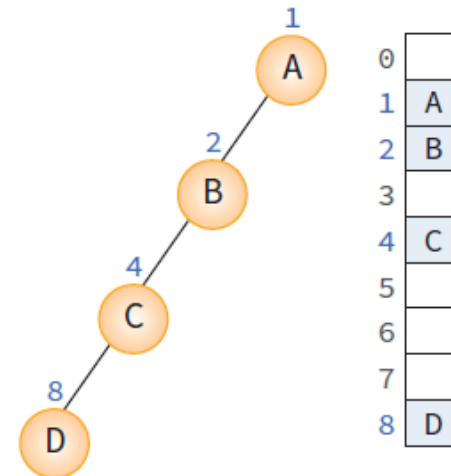
완전이진트리는 배열을 통한 구현이 훨씬 효율적



- 배열표현법: 모든 이진 트리를 포화 이진 트리라고 가정하고 각 노드에 번호를 붙여서 그 번호를 배열의 인덱스로 삼아 노드의 데이터를 배열에 저장하는 방법



(a) 완전 이진트리



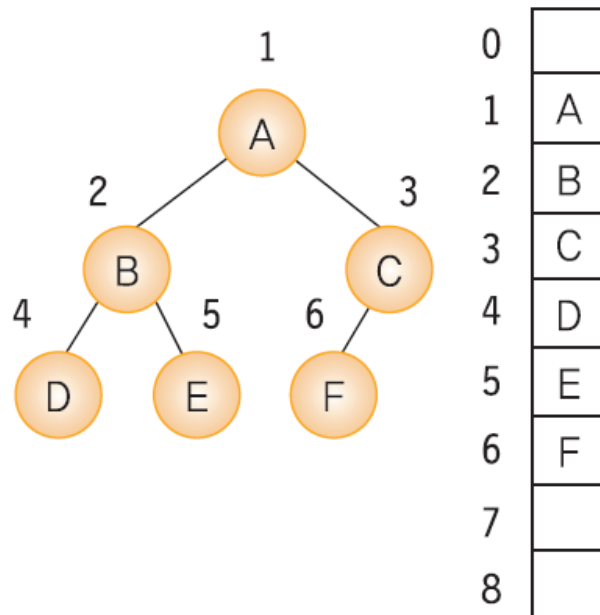
(b) 경사 이진트리



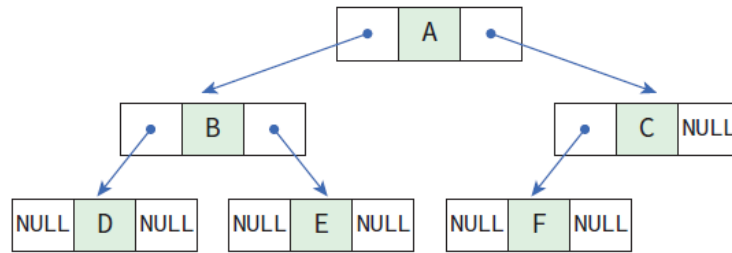
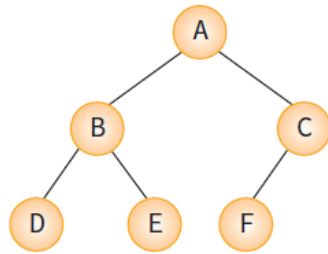


부모와 자식 인덱스 관계

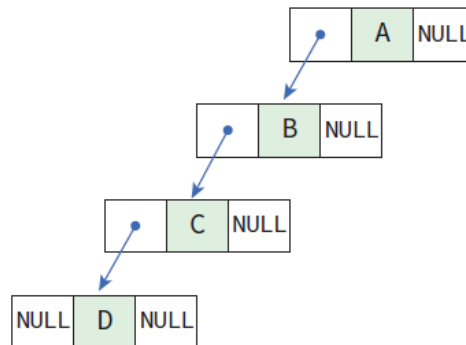
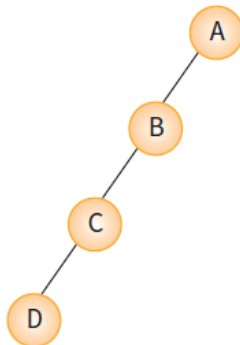
- 노드 i 의 부모 노드 인덱스 = $i/2$
- 노드 i 의 왼쪽 자식 노드 인덱스 = $2i$
- 노드 i 의 오른쪽 자식 노드 인덱스 = $2i+1$



- 링크 표현법: 포인터를 이용하여 부모노드가 자식노드를 가리키게 하는 방법



(a) 완전 이진트리



(b) 경사 이진트리





- 노드는 구조체로 표현
- 링크는 포인터로 표현

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





링크 표현 프로그램

```
#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));
```





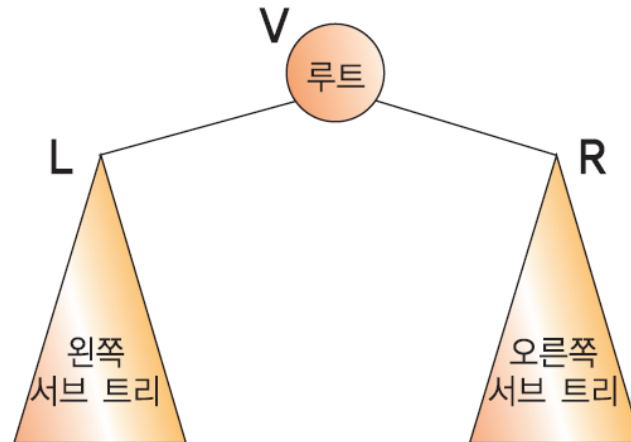
```
n1->data = 10;           // 첫 번째 노드를 설정한다.  
n1->left = n2;  
n1->right = n3;  
n2->data = 20;           // 두 번째 노드를 설정한다.  
n2->left = NULL;  
n2->right = NULL;  
n3->data = 30;           // 세 번째 노드를 설정한다.  
n3->left = NULL;  
n3->right = NULL;  
free(n1); free(n2); free(n3);  
return 0;  
}
```





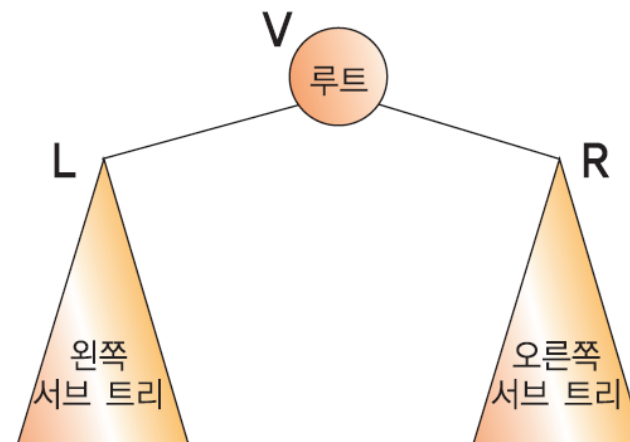
이진 트리의 순회

- 순회(traversal): 트리의 노드들을 체계적으로 방문하는 것

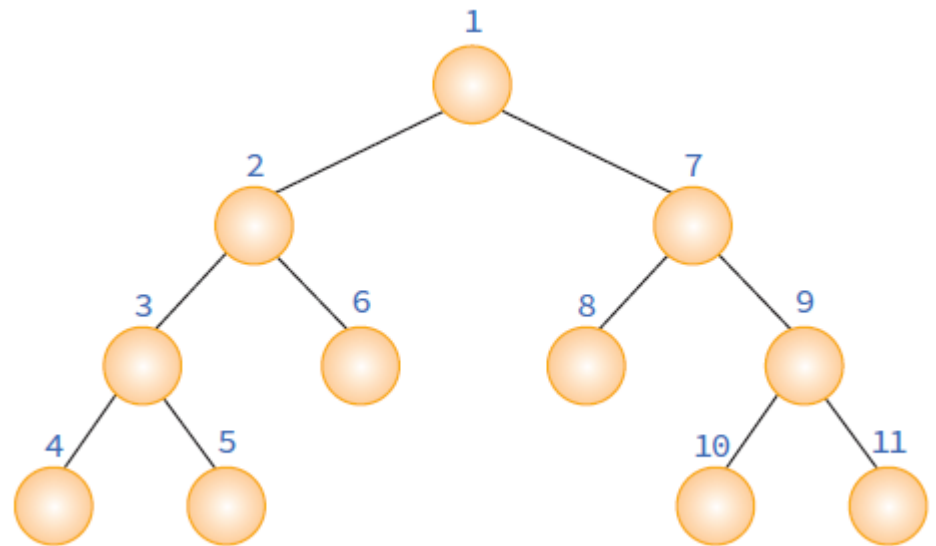
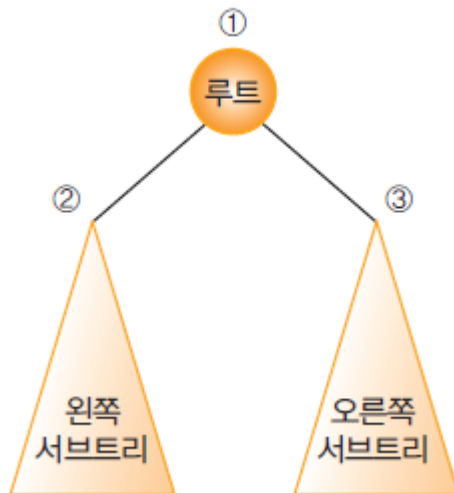


이진 트리의 순회

- 3가지의 기본적인 순회방법
 - ▣ 전위순회(preorder traversal) : VLR
 - 자손노드보다 루트노드를 먼저 방문한다.
 - ▣ 중위순회(inorder traversal) : LVR
 - 왼쪽 자손, 루트, 오른쪽 자손 순으로 방문한다.
 - ▣ 후위순회(postorder traversal) : LRV
 - 루트노드보다 자손을 먼저 방문한다.



1. 루트 노드를 방문한다
2. 왼쪽 서브트리를 방문한다
3. 오른쪽 서브트리를 방문한다



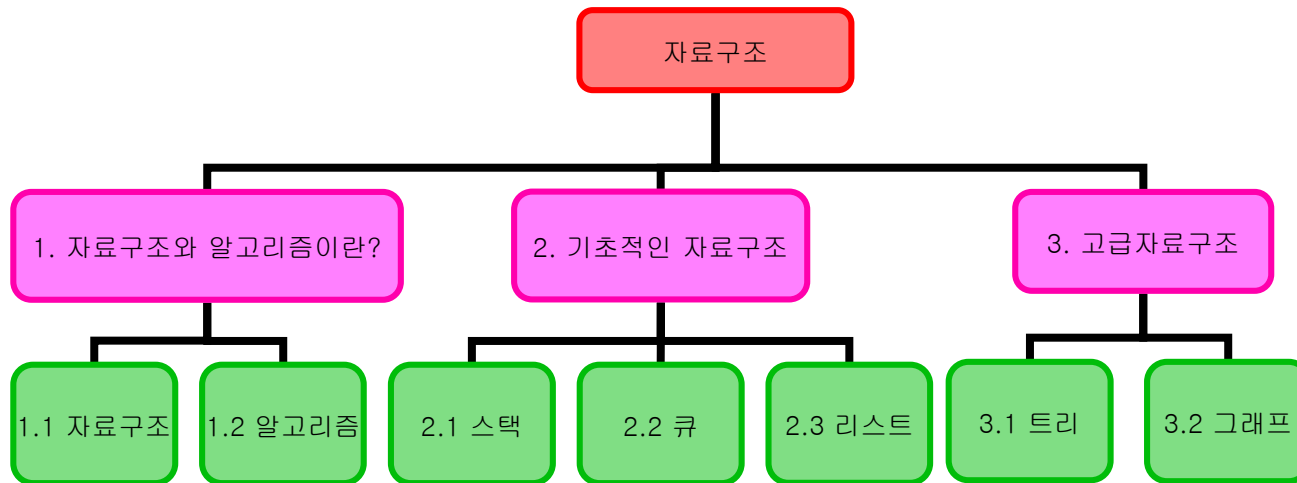


- 순환 호출을 이용한다.

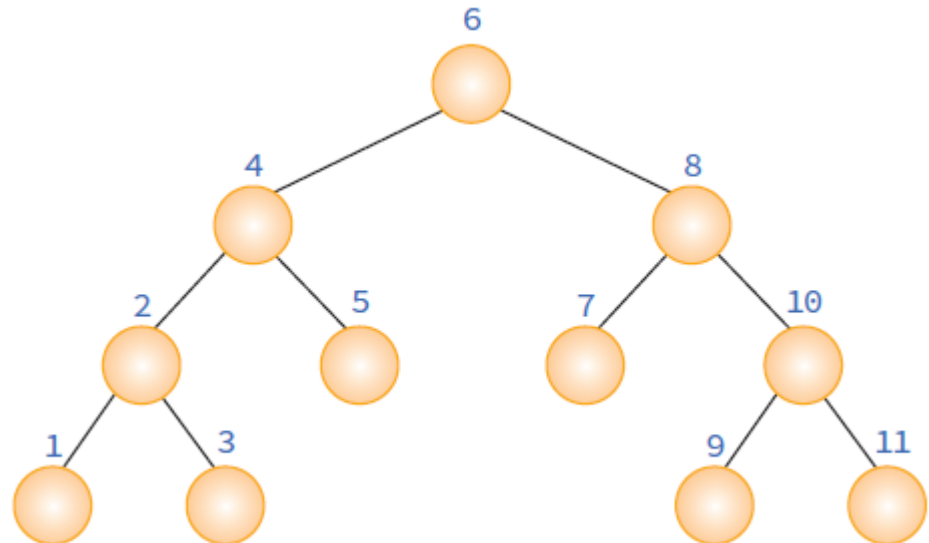
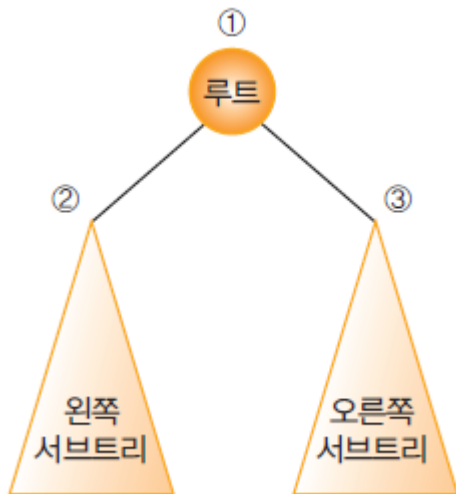
```
preorder(x)  
if  $x \neq \text{NULL}$   
    then    print DATA(x);  
            preorder(LEFT(x));  
            preorder(RIGHT(x));
```



□ (예) 구조화된 문서출력



1. 왼쪽 서브트리를 방문한다
2. 루트 노드를 방문한다
3. 오른쪽 서브트리를 방문한다





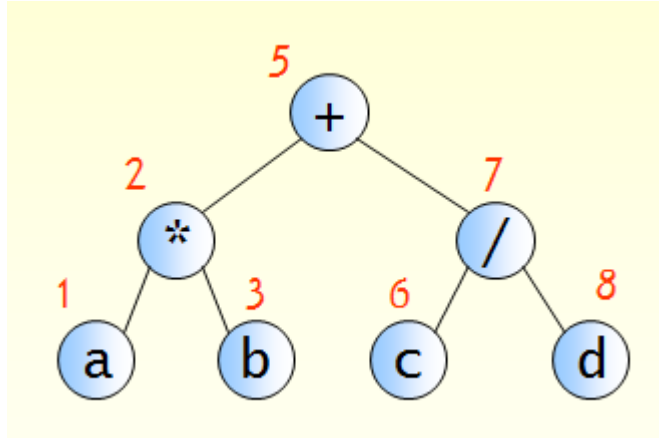
중위 순회 알고리즘

- 순환 호출을 이용한다.

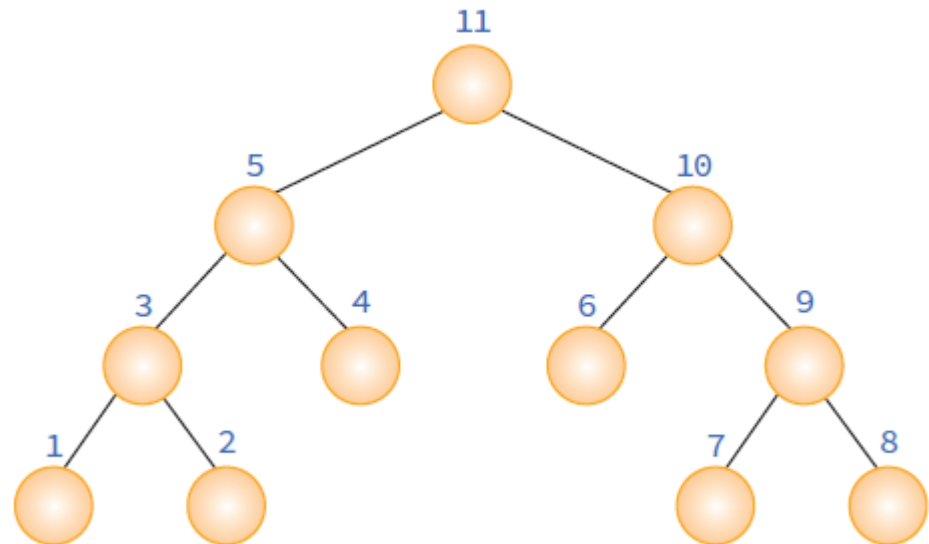
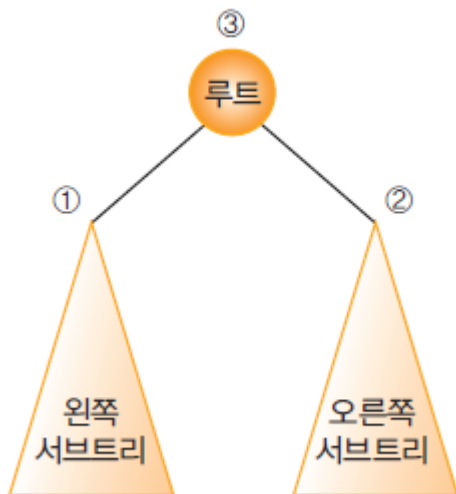
```
inorder(x)  
if  $x \neq \text{NULL}$   
    then      inorder(LEFT(x));  
              print DATA(x);  
              inorder(RIGHT(x));
```



□ (예) 수식 트리



1. 왼쪽 서브트리를 방문한다
2. 오른쪽 서브트리를 방문한다
3. 루트 노드를 방문한다





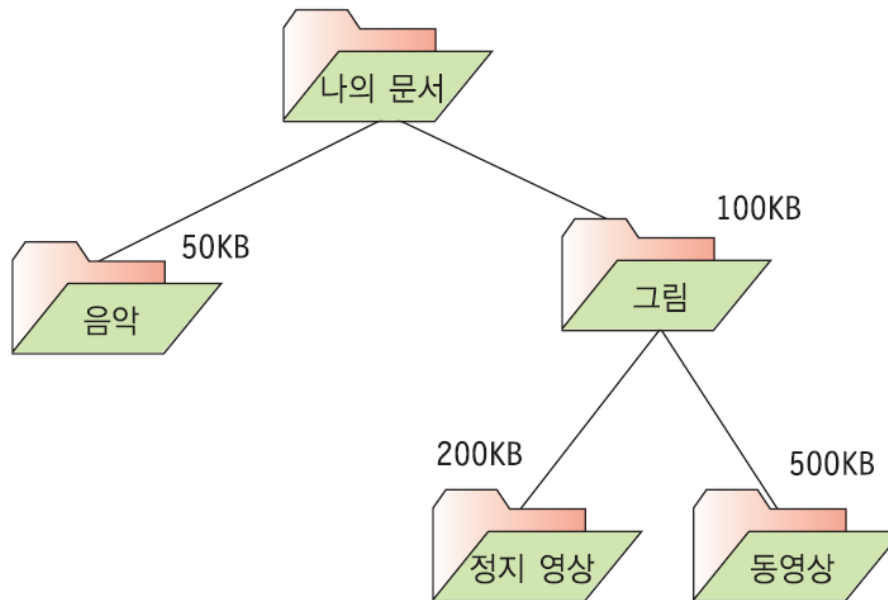
후위 순회 알고리즘

- 순환 호출을 이용한다.

```
postorder(x)  
if  $x \neq \text{NULL}$   
    then    postorder(LEFT(x));  
            postorder(RIGHT(x));  
            print DATA(x);
```



□ (예) 디렉토리 용량 계산



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



// 중위 순회

```
inorder( TreeNode *root ){
```

```
    if ( root ){
```

```
        inorder( root->left );
```

// 왼쪽서브트리 순회

```
        printf("%d", root->data );
```

// 노드 방문

```
        inorder( root->right );
```

// 오른쪽서브트리 순회

```
    }
```

```
}
```

// 전위 순회

```
preorder( TreeNode *root ){
```

```
    if ( root ){
```

```
        printf("%d", root->data );
```

// 노드 방문

```
        preorder( root->left );
```

// 왼쪽서브트리 순회

```
        preorder( root->right );
```

// 오른쪽서브트리 순회

```
    }
```

```
}
```



// 후위 순회

```
postorder( TreeNode *root ){  
    if ( root ){  
        postorder( root->left );    // 왼쪽 서브 트리 순회  
        postorder( root->right );   // 오른쪽 서브 트리 순회  
        printf("%d", root->data );  // 노드 방문  
    }  
}
```





```
int main(void)
{
    printf("중위 순회=");
    inorder(root);
    printf("\n");

    printf("전위 순회=");
    preorder(root);
    printf("\n");

    printf("후위 순회=");
    postorder(root);
    printf("\n");
    return 0;
}
```





중위 순회=[1] [4] [15] [16] [20] [25]

전위 순회=[15] [4] [1] [20] [16] [25]

후위 순회=[1] [4] [16] [25] [20] [15]





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

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

#define SIZE 100
int top = -1;
TreeNode *stack[SIZE];

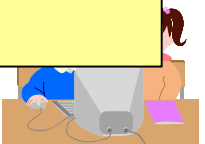
void push(TreeNode *p)
{
    if (top < SIZE - 1)
        stack[++top] = p;
}
```





```
TreeNode *pop()
{
    TreeNode *p = NULL;
    if (top >= 0)
        p = stack[top--];
    return p;
}

void inorder_iter(TreeNode *root)
{
    while (1) {
        for (; root; root = root->left)
            push(root);
        root = pop();
        if (!root) break;
        printf("[%d] ", root->data);
        root = root->right;
    }
}
```





바보적인 순회

```
//          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;

int main(void)
{
    printf("중위 순회=");
    inorder_iter(root);
    printf("\n");
    return 0;
}
```



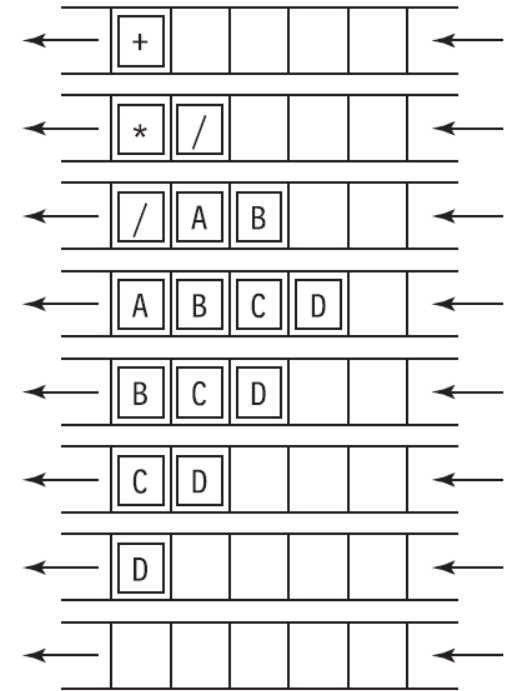
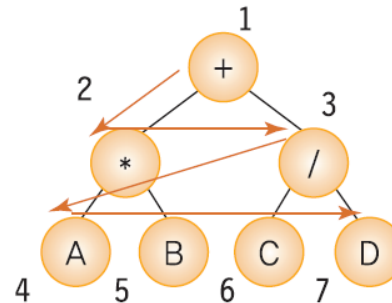


중위 순회=[1] [4] [15] [16] [20] [25]



- 레벨 순회(level order)는
각 노드를 레벨 순으로
검사하는 순회 방법

- 지금까지의 순회법이
스택을 사용했던 것에 비해
레벨 순회는 큐를 사용하는
순회 방법이다.





레벨 순회 알고리즘

□ level_order(root):

1. initialize queue;
2. enqueue(queue, root);
3. while is_empty(queue)≠TRUE do
4. $x \leftarrow \text{dequeue}(\text{queue});$
5. if($x \neq \text{NULL}$) then
6. print DATA(x);
7. enqueue(queue, LEFT(x));
8. enqueue(queue, RIGHT(x));



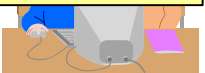


레벨 순회 프로그램

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

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

// ===== 원형큐 코드 시작 =====
#define MAX_QUEUE_SIZE 100
typedef TreeNode * element;
typedef struct { // 큐 타입
    element data[MAX_QUEUE_SIZE];
    int front, rear;
} QueueType;
```



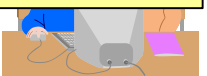


레벨 순회 프로그램

```
// 오류 함수
void error(char *message)
{
    fprintf(stderr, "%s\n", message);
    exit(1);
}

// 공백 상태 검출 함수
void init_queue(QueueType *q)
{
    q->front = q->rear = 0;
}

// 공백 상태 검출 함수
int is_empty(QueueType *q)
{
    return (q->front == q->rear);
}
```



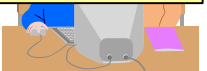


레벨 순회 프로그램

```
// 포화 상태 검출 함수
int is_full(QueueType *q)
{
    return ((q->rear + 1) % MAX_QUEUE_SIZE == q->front);
}

// 삽입 함수
void enqueue(QueueType *q, element item)
{
    if (is_full(q))
        error("큐가 포화상태입니다");
    q->rear = (q->rear + 1) % MAX_QUEUE_SIZE;
    q->data[q->rear] = item;
}

// 삭제 함수
element dequeue(QueueType *q)
{
    if (is_empty(q))
        error("큐가 공백상태입니다");
    q->front = (q->front + 1) % MAX_QUEUE_SIZE;
    return q->data[q->front];
}
```



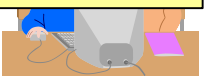


레벨 순회 프로그램

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

    init_queue(&q);      // 큐 초기화

    if (ptr == NULL) return;
    enqueue(&q, ptr);
    while (!is_empty(&q)) {
        ptr = dequeue(&q);
        printf(" [%d] ", ptr->data);
        if (ptr->left)
            enqueue(&q, ptr->left);
        if (ptr->right)
            enqueue(&q, ptr->right);
    }
}
```

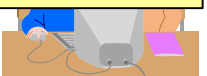




레벨 순회 프로그램

```
//          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;

int main(void)
{
    printf("레벨 순회=");
    level_order(root);
    printf("\n");
    return 0;
}
```

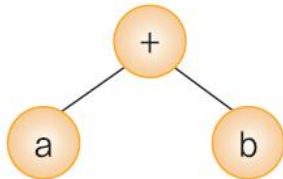




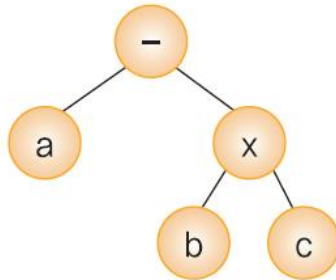
레벨 순회= [15] [4] [20] [1] [16] [25]



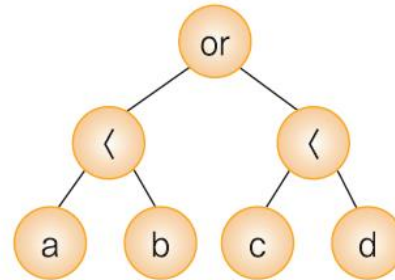
- 수식트리: 산술식을 트리형태로 표현한 것
 - ▣ 비단말노드: 연산자(operator)
 - ▣ 단말노드: 피연산자(operand)
- 예)



(a)



(b)



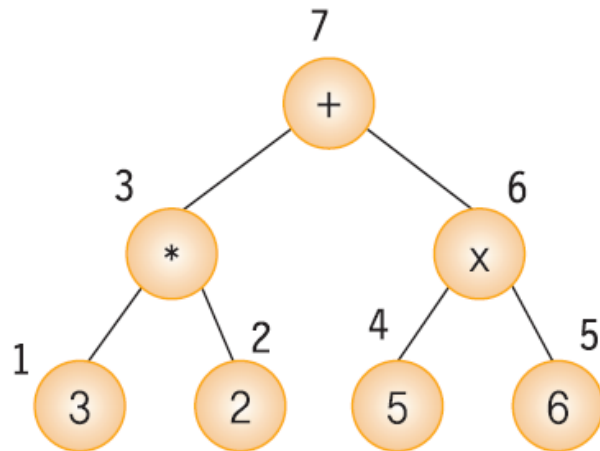
(c)

수식	$a + b$	$a - (b \times c)$	$(a < b) \text{ or } (c < d)$
전위순회	$+ a b$	$- a \times b c$	$\text{or} < a b < c d$
중위순회	$a + b$	$a - (b \times c)$	$(a < b) \text{ or } (c < d)$
후위순회	$a b +$	$a b c \times -$	$a b < c d < \text{or}$



수식 트리 계산

- 후위순회를 사용
- 서브트리의 값을 순환호출로 계산
- 비단말노드를 방문할 때 양쪽 서브트리의 값을 노드에 저장된 연산자를 이용하여 계산한다





수식 트리 알고리즘

evaluate(exp)


1. if exp = NULL
2. then return 0;
3. else $x \leftarrow \text{evaluate}(\text{exp} \rightarrow \text{left});$
4. $y \leftarrow \text{evaluate}(\text{exp} \rightarrow \text{right});$
5. $\text{op} \leftarrow \text{exp} \rightarrow \text{data};$
6. return (x op y);





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





```
// 수식 계산 함수
```

```
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);  
        printf("%d %c %d을 계산합니다.\n", op1, root->data, op2);  
        switch (root->data) {  
            case '+':    return op1 + op2;  
            case '-':    return op1 - op2;  
            case '*':    return op1 * op2;  
            case '/':    return op1 / op2;  
        }  
    }  
    return 0;  
}  
//  
int main(void)  
{  
    printf("수식의 값은 %d입니다. \n", evaluate(exp));  
    return 0;  
}
```



1 * 4을 계산합니다.

16 + 25을 계산합니다.

4 + 41을 계산합니다.

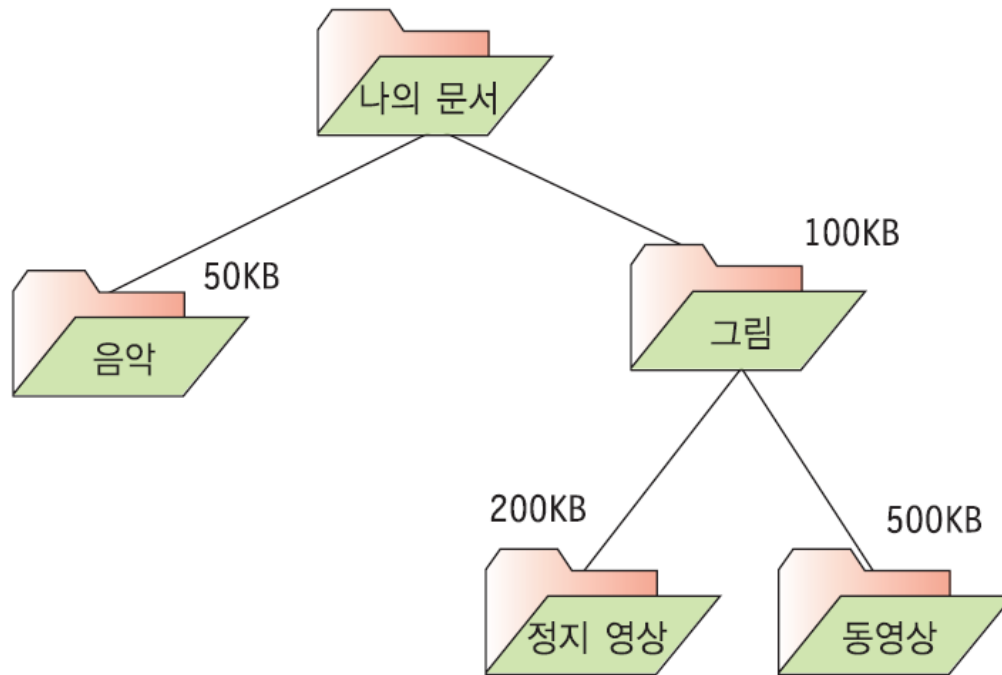
수식의 값은 45입니다.





디렉토리 용량 계산

- 디렉토리의 용량을 계산하는데 후위 트리 순회 사용





디렉토리 용량 계산 프로그램

```
int calc_dir_size(TreeNode *root)
{
    int left_size, right_size;
    if (root == NULL) return 0;

    left_size = calc_dir_size(root->left);
    right_size = calc_dir_size(root->right);
    return (root->data + left_size + right_size);
}
//
int main(void)
{
    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("디렉토리의 크기=%d\n", calc_dir_size(&n1));
}
```





디렉토리의 크기=850

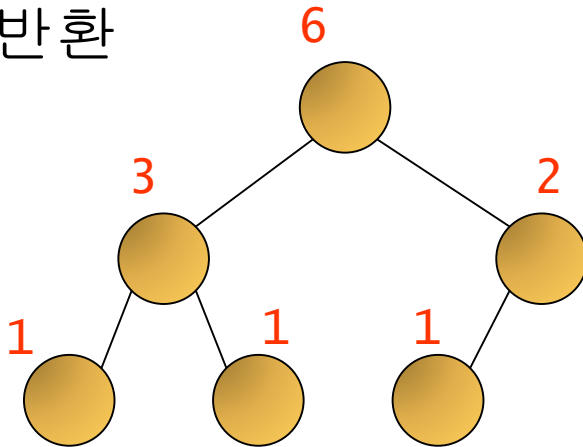




이진 트리 연산: 노드 개수

- 탐색 트리안의 노드의 개수를 계산
- 각각의 서브트리에 대하여 순환 호출한 다음, 반환되는 값에 1을 더하여 반환

```
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;
}
```

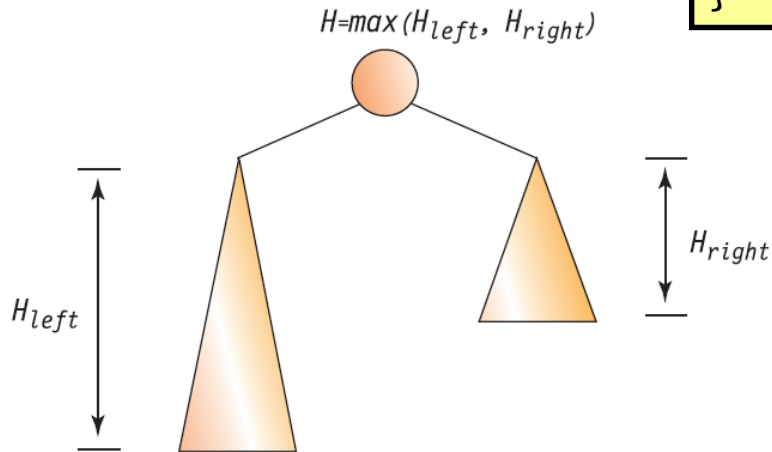




이진 트리 연산: 높이

- 서브트리에 대하여 순환호출하고 서브 트리들의 반환값 중에서 최대값을 구하여 반환

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





이진 트리 연산: 단말 노드 개수

```
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;
}
```

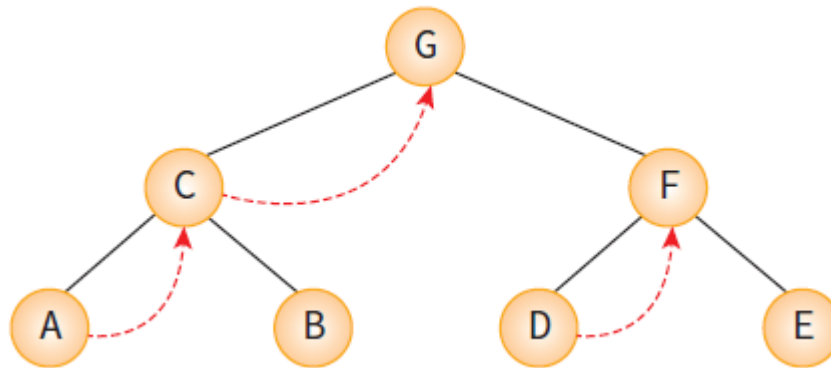




스레드 이진 트리

범위 x

- 이진트리의 NULL 링크를 이용하여 순환 호출 없이도 트리의 노드들을 순회
- NULL 링크에 중위 순회시에 후속 노드인 중위 후속자 (inorder successor)를 저장시켜 놓은 트리가 스레드 이진 트리(threaded binary tree)





스레드 이진 트리의 구현

- 단말노드와 비단말노드의 구별을 위하여 is_thread 필드 필요

```
typedef struct TreeNode {  
    int data;  
    struct TreeNode *left, *right;  
    int is_thread; //만약 오른쪽 링크가 스레드이면 TRUE  
} TreeNode;
```





스레드 이진 트리의 구현

□ 중위 후속자를 찾는 함수 작성

```
//
TreeNode *find_successor(TreeNode *p)
{
    // q는 p의 오른쪽 포인터
    TreeNode *q = p->right;
    // 만약 오른쪽 포인터가 NULL이거나 스레드이면 오른쪽 포인터를 반환
    if( q==NULL || p->is_thread == TRUE)
        return q;
    // 만약 오른쪽 자식이면 다시 가장 왼쪽 노드로 이동
    while( q->left != NULL ) q = q->left;
    return q;
}
```





스레드 이진 트리의 구현

```
typedef struct TreeNode {  
    int data;  
    struct TTreeNode *left, *right;  
    int is_thread; // 만약 오른쪽 링크가 스레드이면 TRUE  
} TTreeNode;
```





스레드 이진 트리의 구현

```
TreeNode * find_successor(TreeNode * p)
{
    // q는 p의 오른쪽 포인터
    TreeNode * q = p->right;
    // 만약 오른쪽 포인터가 NULL이거나 스레드이면 오른쪽 포인터를 반환
    if (q == NULL || p->is_thread == TRUE)
        return q;

    // 만약 오른쪽 자식이면 다시 가장 왼쪽 노드로 이동
    while (q->left != NULL) q = q->left;
    return q;
}
```





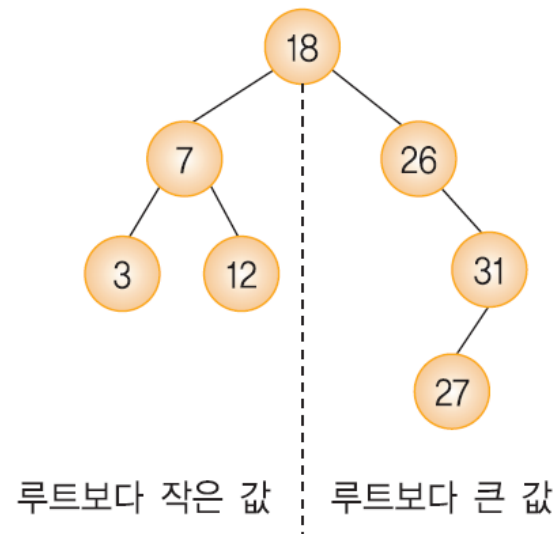
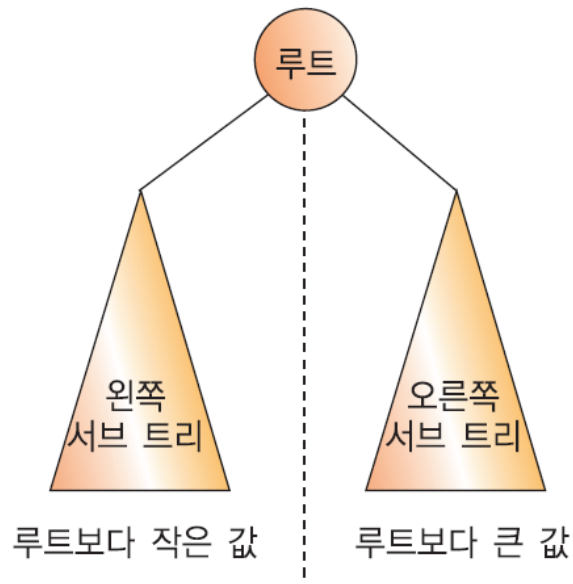
스레드 이진 트리의 구현

□ 스레드 버전 중위 순회 함수 작성

```
void thread_inorder(TreeNode *t)
{
    TreeNode *q;
    q=t;
    while (q->left) q = q->left; // 가장 왼쪽 노드로 간다.
    do
    {
        printf("%c ", q->data); // 데이터 출력
        q = find_successor(q); // 후속자 함수 호출
    } while(q);                // NULL이 아니면
}
```



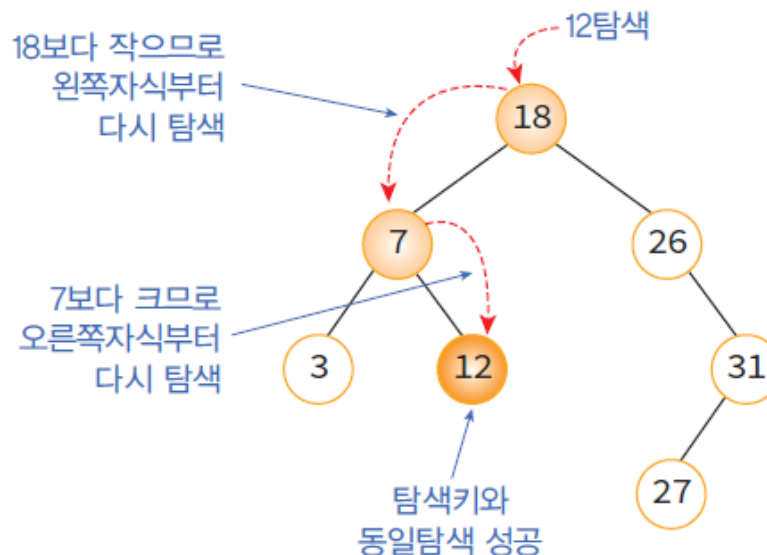
- 탐색작업을 효율적으로 하기 위한 자료구조
- $key(\text{왼쪽서브트리}) \leq key(\text{루트노드}) \leq key(\text{오른쪽서브트리})$
- 이진탐색를 중위순회하면 오름차순으로 정렬된 값을 얻을 수 있다.





이진탐색트리에서의 탐색연산

- 비교한 결과가 같으면 탐색이 성공적으로 끝난다.
- 비교한 결과가, 주어진 키 값이 루트 노드의 키값보다 작으면 탐색은 이 루트 노드의 왼쪽 자식을 기준으로 다시 시작한다.
- 비교한 결과가, 주어진 키 값이 루트 노드의 키값보다 크면 탐색은 이 루트 노드의 오른쪽 자식을 기준으로 다시 시작한다.





이진탐색트리에서의 탐색역사

```
search (root, key):
```

```
  if root == NULL
```

```
    then return NULL;
```

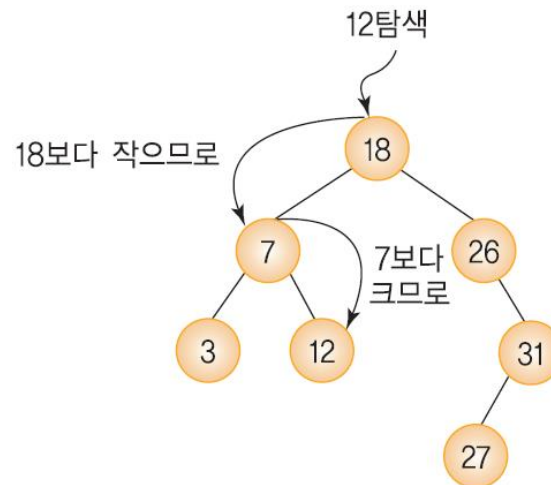
```
  if key == KEY(root)
```

```
    then return root;
```

```
    else if key < KEY(root)
```

```
      then return search(LEFT(root), k);
```

```
    else return search(RIGHT(root), k);
```





탐색을 구현하는 방법

- 반복적 방법
- 순환적 방법





순환적이 바버

//순환적인 탐색 함수

```
TreeNode *search(TreeNode *node, int key)
{
    if ( node == NULL ) return NULL;
    if ( key == node->key ) return node;    (1)
    else if ( key < node->key )
        return search(node->left, key);    (2)
    else
        return search(node->right, key); (3)
}
```





바보적이 바버

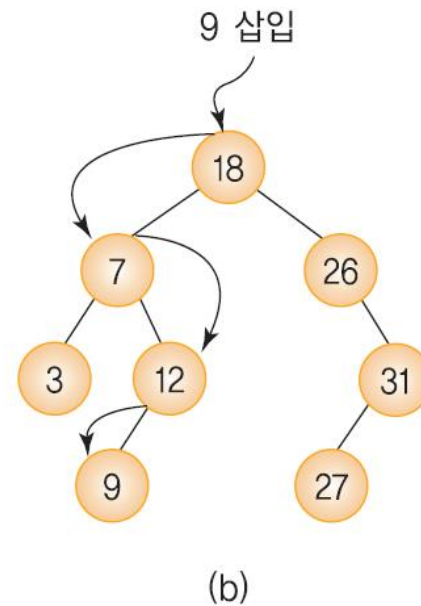
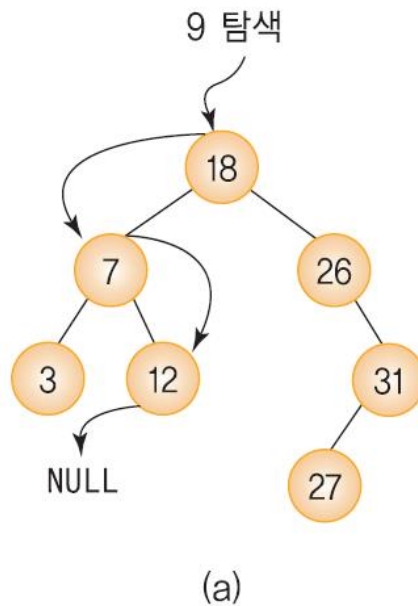
```
// 반복적인 탐색 함수
TreeNode *search(TreeNode *node, int key)
{
    while(node != NULL){
        if( key == node->key ) return node;
        else if( key < node->key )
            node = node->left;
        else
            node = node->right;
    }
    return NULL;          // 탐색에 실패했을 경우 NULL 반환
}
```





이진 탐색 트리에서의 삽입 연산

- 이진 탐색 트리에 원소를 삽입하기 위해서는 먼저 탐색을 수행하는 것이 필요
- 탐색에 실패한 위치가 바로 새로운 노드를 삽입하는 위치





이진 탐색트리에서의 삽입 연산

```
insert (root, n):
```

```
if KEY(n) == KEY(root)                // root와 키가 같으면
    then return;                       // return
else if KEY(n) < KEY(root) then        // root보다 키가 작으면
    if LEFT(root) == NULL              // root의 왼쪽 자식이
        then LEFT(root) ← n;          // 없으면 n이 왼쪽 자식
    else insert(LEFT(root),n);         // 있으면 순환 호출
else                                   // root보다 키가 크면
    if RIGHT(root) == NULL
        then RIGHT(root) ← n;
    else insert(RIGHT(root),n);
```





이진탐색트리에서의 삽입연산

```
TreeNode * insert_node(TreeNode * node, int key)
{
    // 트리가 공백이면 새로운 노드를 반환한다.
    if (node == NULL) return new_node(key);

    // 그렇지 않으면 순환적으로 트리를 내려간다.
    if (key < node->key)
        node->left = insert_node(node->left, key);
    else if (key > node->key)
        node->right = insert_node(node->right, key);

    // 변경된 루트 포인터를 반환한다.
    return node;
}
```





이진탐색트리에서의 삽입연산

```
TreeNode * new_node(int item)
{
    TreeNode * temp = (TreeNode *)malloc(sizeof(TreeNode));
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}
```



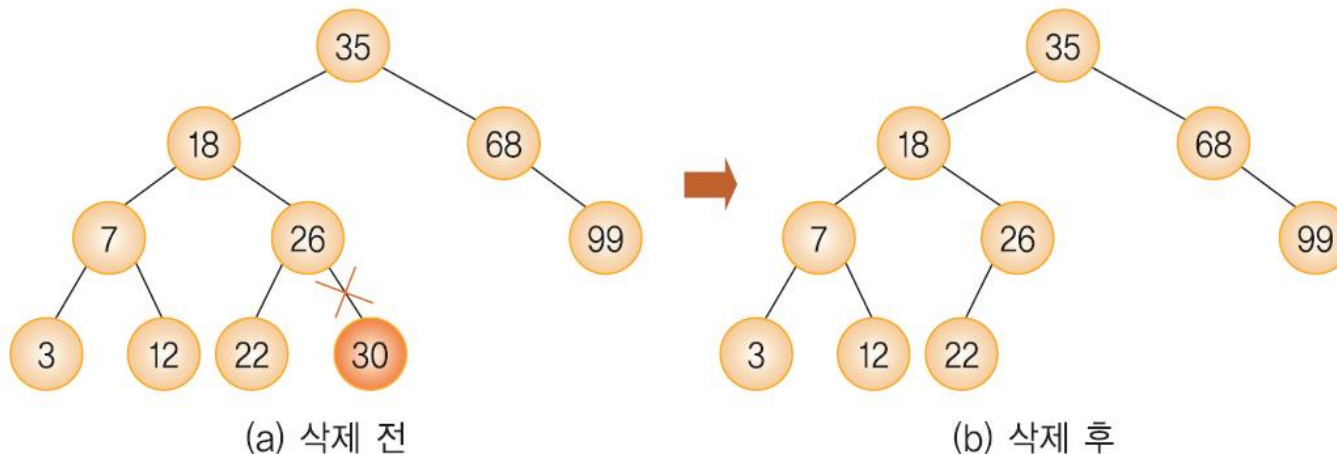


이진 탐색 트리에서의 삭제 연산

□ 3가지의 경우

1. 삭제하려는 노드가 단말 노드 일 경우
2. 삭제하려는 노드가 하나의 왼쪽이나 오른쪽 서브 트리 중 하나만 가지고 있는 경우
3. 삭제하려는 노드가 두개의 서브 트리 모두 가지고 있는 경우

□ CASE 1: 삭제하려는 노드가 단말 노드일 경우: 단말노드 으

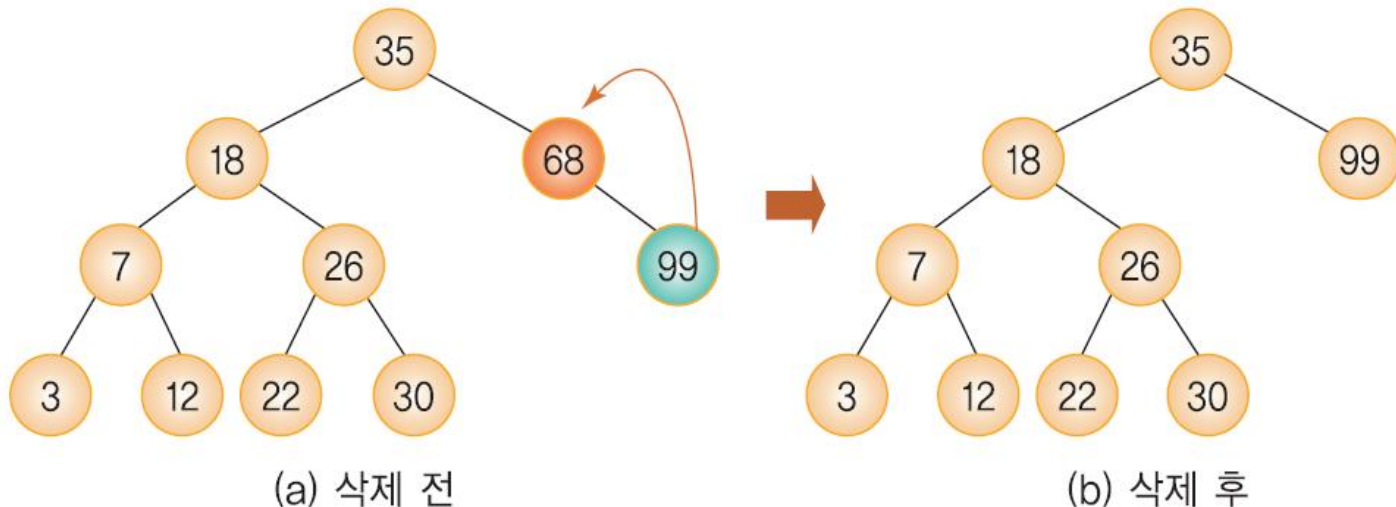




이진 탐색 트리에서의 삭제 연산

- **CASE 2:** 삭제하려는 노드가 하나의 서브트리만 갖고 있는 경우 :

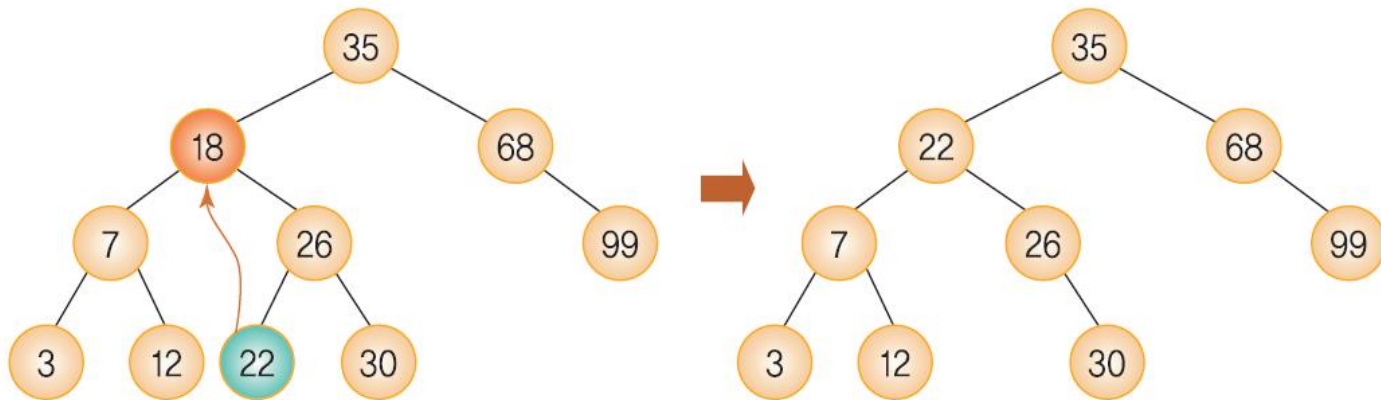
삭제되는 노드가 왼쪽이나 오른쪽 서브 트리중 하나만 갖고 있을 때, 그 노드는 삭제하고 서브 트리는 부모 노드에 붙여준다.





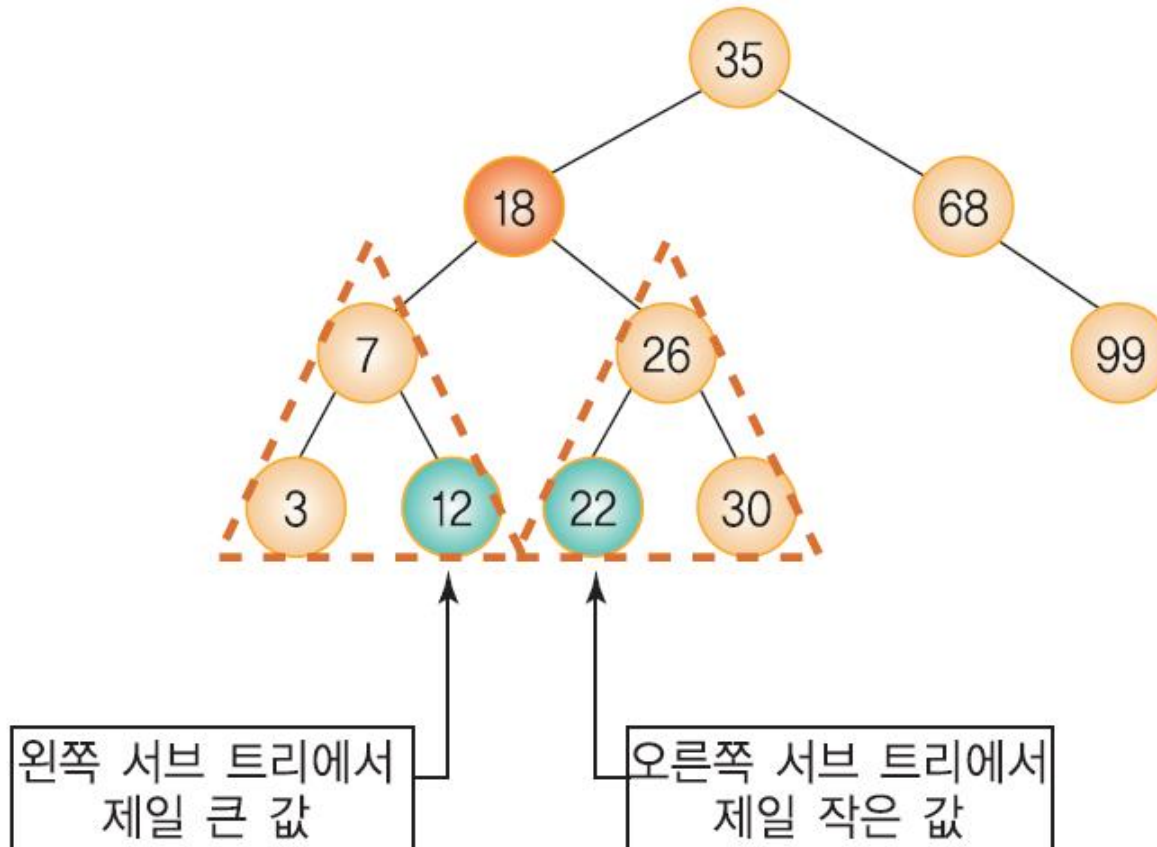
이진 탐색트리에서의 삭제연산

- **CASE 3:** 삭제하려는 노드가 두개의 서브트리를 갖고 있는 경우: 삭제노드와 가장 비슷한 값을 가진 노드를 삭제노드 위치로 가져온다.





가장 비슷한 값은 어디에 있을까?





```
// 이진 탐색 트리와 키가 주어지면 키가 저장된 노드를 삭제하고
// 새로운 루트 노드를 반환한다.
TreeNode * delete_node(TreeNode * root, int key)
{
    if (root == NULL) return root;

    // 만약 키가 루트보다 작으면 왼쪽 서브 트리에 있는 것임
    if (key < root->key)
        root->left = delete_node(root->left, key);
    // 만약 키가 루트보다 크면 오른쪽 서브 트리에 있는 것임
    else if (key > root->key)
        root->right = delete_node(root->right, key);
    // 키가 루트와 같으면 이 노드를 삭제하면 됨
```





```
else {  
    // 첫 번째나 두 번째 경우  
    if (root->left == NULL) {  
        TreeNode * temp = root->right;  
        free(root);  
        return temp;  
    }  
    else if (root->right == NULL) {  
        TreeNode * temp = root->left;  
        free(root);  
        return temp;  
    }  
    // 세 번째 경우  
    TreeNode * temp = min_value_node(root->right);  
  
    // 중위 순회시 후계 노드를 복사한다.  
    root->key = temp->key;  
    // 중위 순회시 후계 노드를 삭제한다.  
    root->right = delete_node(root->right, temp->key);  
}  
return root;  
}
```





```
TreeNode * min_value_node(TreeNode * node)
{
    TreeNode * current = node;

    // 맨 왼쪽 단말 노드를 찾으러 내려감
    while (current->left != NULL)
        current = current->left;

    return current;
}
```





```
int main(void)
{
    TreeNode * root = NULL;
    TreeNode * tmp = NULL;

    root = insert_node(root, 30);
    root = insert_node(root, 20);
    root = insert_node(root, 10);
    root = insert_node(root, 40);
    root = insert_node(root, 50);
    root = insert_node(root, 60);

    printf("이진 탐색 트리 중위 순회 결과 \n");
    inorder(root);
    printf("\n\n");
    if (search(root, 30) != NULL)
        printf("이진 탐색 트리에서 30을 발견함 \n");
    else
        printf("이진 탐색 트리에서 30을 발견못함 \n");
    return 0;
}
```





이진 탐색 트리 중위 순회 결과

[10] [20] [30] [40] [50] [60]

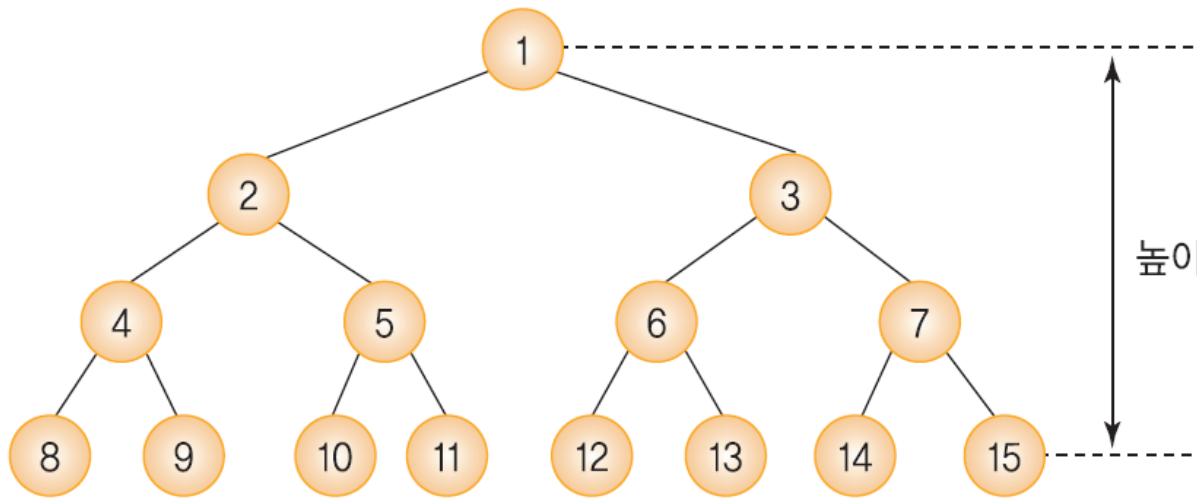
이진 탐색 트리에서 30을 발견함





이진 탐색 트리의 성능 분석

- 이진 탐색 트리에서의 탐색, 삽입, 삭제 연산의 시간 복잡도는 트리의 높이를 h 라고 했을 때 h 에 비례한다



$$\text{높이} = \lceil \log_2 n \rceil = \lceil \log_2 15 \rceil = 4$$

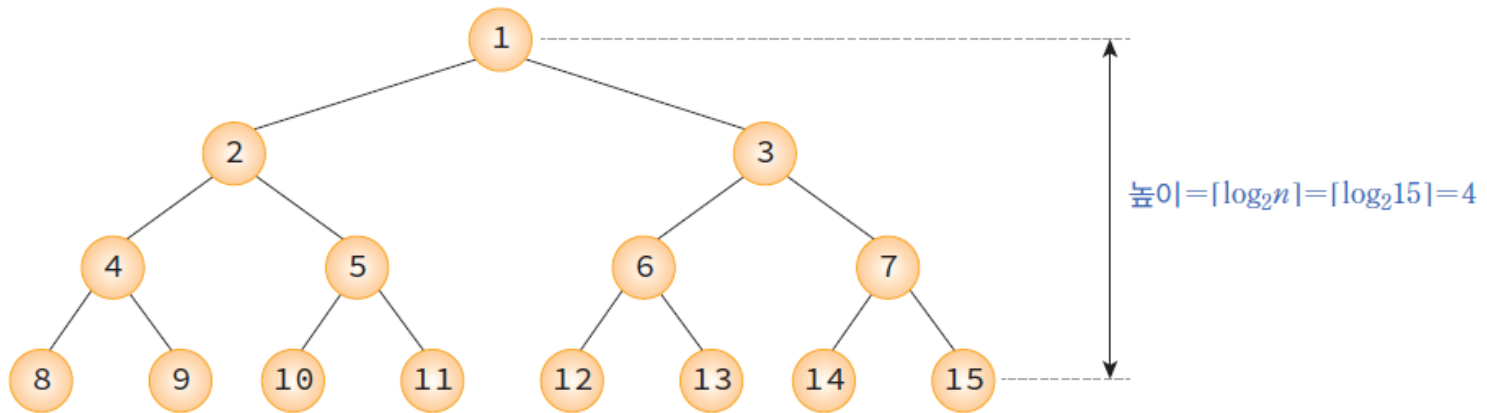




이진 탐색 트리의 성능 분석

□ 최선의 경우

- 이진 트리가 균형적으로 생성되어 있는 경우
- $h = \log_2 n$





이진탐색트리의 성능분석

- 최악의 경우
 - ▣ 한쪽으로 치우친 경사이진트리의 경우
 - ▣ $h=n$
 - ▣ 순차탐색과 시간복잡도가 같다.

