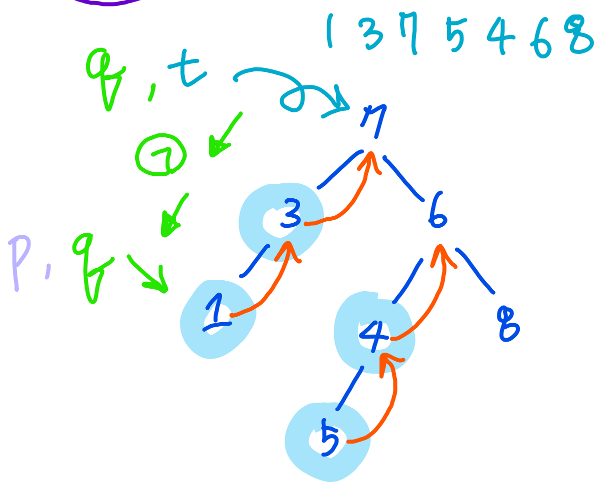
```
#include <stdio.h>
#define TRUE 1
#define FALSE 0

typedef struct TreeNode {
        int data;
        struct TreeNode *left, *right;
        int is_thread; // 단말인지 비단말인지 구별  쓰레드 노드인지 아닌지
} TreeNode;


//              7
//      3               6
//  1       2       4       5
//
TreeNode n1 = { '1',  NULL, NULL, 1 }; // 단말   n1.right = &n3;
TreeNode n2 = { '2',  NULL,  NULL, 1 }; // 단말
TreeNode n3 = { '3',  &n1,  &n2, 0 };
TreeNode n4 = { '4', NULL, NULL, 1 }; // 단말
TreeNode n5 = { '5', NULL, NULL, 0 }; // 단말이지만 중위 순회의 마지막 노드이므로 0으로
TreeNode n6 = { '6', &n4,  &n5, 0 };
TreeNode n7 = { '7', &n3,  &n6, 0 };
TreeNode *exp = &n7;

//p의 중위 후속 노드를 리턴한다.
TreeNode *find_thread_successor(TreeNode *p)
{
  TreeNode *q = p -> right;
  if ( p -> if_thread == 1) return p -> right;

  return q;
}
void thread_inorder(TreeNode *t)
{
        TreeNode *q;
        q = t;

        // 일단 중위 순회의 첫 노드로 이동
        while (q -> left != NULL)
            q = q -> left;
        do
        {
            q->data 출력
            q = find_thread_successor(q); -> 모듈화됨!

        } while (q);
}
main()
{
        // 단말 노드의 오른쪽 링크에 중위 후속 노드를 연결한다
        n1.right = &n3;


        thread_inorder(exp);
}
```



재귀적(개정판 교재)

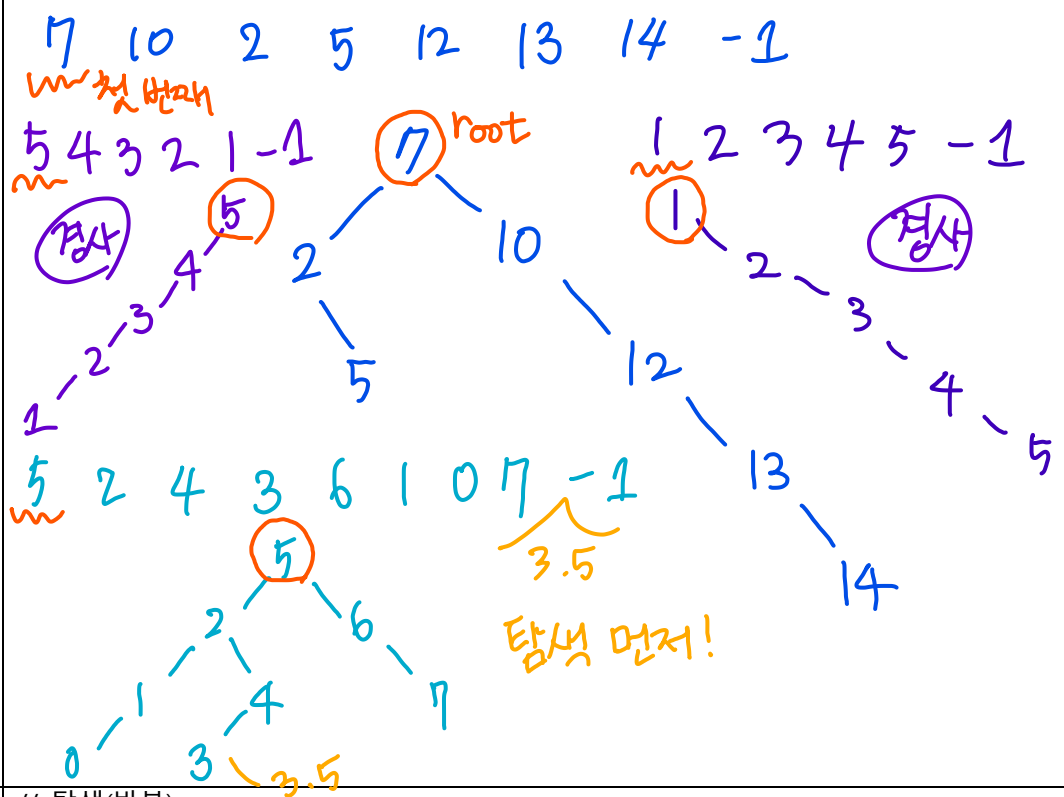
```
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#define TRUE 1
#define FALSE 0

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

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

void inorder(TreeNode * root) {   // 중위 순회
	if (root) {
		inorder(root->left);// 왼쪽서브트리 순회
		printf("[%d] ", root->key);   // 노드 방문
		inorder(root->right);// 오른쪽서브트리 순회
	}
}
```

```
//탐색(순환)
TreeNode *search(TreeNode *node, int key)
{
	if( node == NULL ) return NULL;









}
```

이진탐색(1/4)

반복적

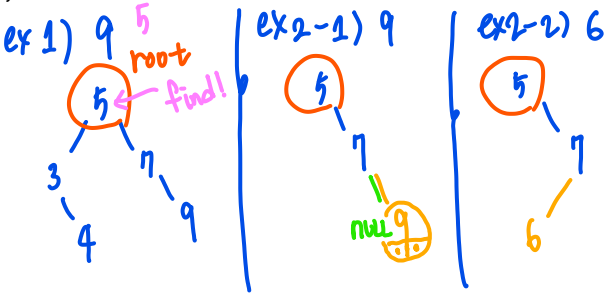
(왼쪽과 같음)




```
// 탐색(반복)
TreeNode *search(TreeNode *node, int key)
{
	while(node != NULL){









}
	return NULL; //탐색에 실패했을 경우 NULL 반환

}
```

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

    // 그렇지 않으면 순환적으로 트리를 내려간다.
    if (key < root->key)
        root->left = insert_node(root->left, key);
    else if (key > root->key)
        root->right = insert_node(root->right, key);
    // 변경된 루트 포인터를 반환한다.
    return root;
}
```

재귀

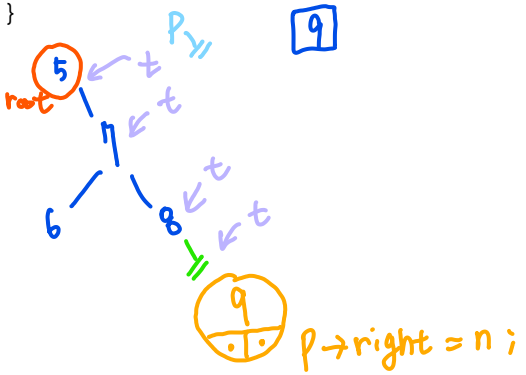


ex0) 9

9

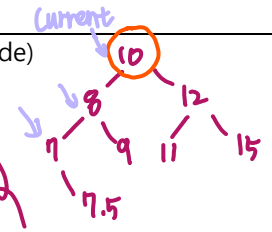
```
TreeNode *insert_node(TreeNode *root, int key)
{
    TreeNode *p, *t; // p는 부모노드, t는 현재노드
    TreeNode *n;     // n은 새로운 노드

    t = root;
    p = NULL;

    // 탐색을 먼저 수행, 반복을 이용해서 search(위의 search함수 참조)
    while (t != NULL){ // 현재노드가 NULL이 될때까지
        // if( key == t->key ) return root;
        p = t; // 현재노드를 부모노드로 하고
        // 현재노드를 전진
        if( key < t->key )
            t = t->left;
        else
            t = t->right;
    }
    // key가 트리 안에 없으므로 삽입 가능
    n = new_node(key);

    // 부모 노드와 링크 연결
    if (p != NULL)
        if (key < p->key)
            p->left = n;
        else
            p->right = n;
    else root = n; // 애초에 트리가 비어있었으면
    return root;
}
```

반복



5 6 2 5 -1

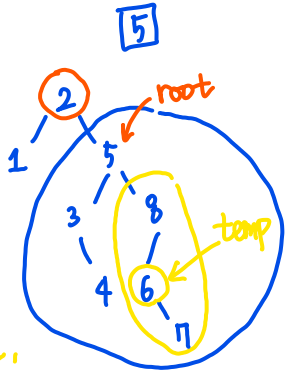
```
TreeNode * min_value_node(TreeNode * node)
{
    TreeNode * current = node;
    while (current->left != NULL) // 맨 왼쪽 단말 노드를 찾으러 내려감
        current = current->left;
    return current;
}

// 삭제 함수
TreeNode * delete_node(TreeNode * root, int key)
{
    TreeNode *temp;
    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
        temp = min_value_node(root->right);

        // 중외 순회시 후계 노드를 복사한다.
        root->key = temp->key;

        // 중외 순회시 후계 노드를 삭제한다.
        root->right = delete_node(root->right,
                                  temp->key);
    }
    return root;
}
```



2개

Case 1 단말
2 one leaf
3 two leaf

one leaf



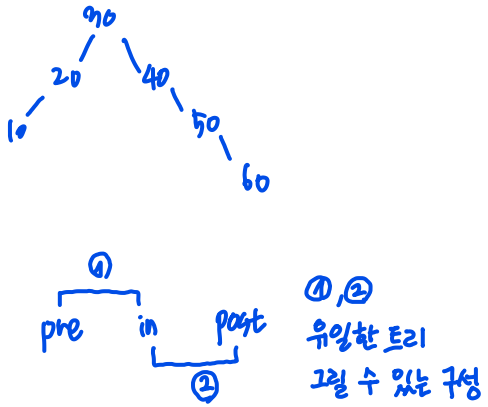
사형X

반복

```
// 삭제 함수
TreeNode *delete_node(TreeNode *root, int key) // 나중에 ROOT <-> *root로 바꿈
{
    TreeNode *p, *child, *succ, *succ_p, *t;

    // key를 갖는 노드 t를 탐색, p는 t의 부모노드
    p = NULL;
    t = root;

    // key를 갖는 노드 t를 탐색한다.
    while( t != NULL && t->key != key      ){
        p = t; // 자식노드를 부모가
        t = ( key < t->key ) ? t->left : t->right; // 자식노드 전진
    }
    // 탐색이 종료된 시점에 t가 NULL이면 트리안에 key가 없음
    if( t == NULL ) {  // 탐색트리에 없는 키
        printf("key is not in the tree");
        return root;
    }

    // 첫번째 경우: 단말노드인 경우
    if( (t->left==NULL) && (t->right==NULL) ){
        if( p != NULL ){
            // 부모노드의 자식필드를 NULL로 만든다.
            if( p->left == t )    // 삭제하려는 것이 왼쪽 노드이면
                p->left = NULL;
            else
                p->right = NULL;
        }
        else//p가 NULL이면(즉 단 하나의 단말 노드이고 이걸 삭제하려면) 루트 삭제
            root = NULL;
    }
    // 두번째 경우: 하나의 자식만 가지는 경우
    else if((t->left==NULL)||(t->right==NULL)){
        child = (t->left != NULL) ? t->left : t->right;
        if( p != NULL ){
            if( p->left == t )  // 부모를 자식과 연결
                p->left = child;
            else p->right = child;
        }
        else    // 만약 부모노드가 NULL이면 삭제되는 노드가 루트
            root = child;
```

```
    // 세번째 경우: 두개의 자식을 가지는 경우
    else{
        // 오른쪽 서브트리에서 후계자를 찾는다.
        succ_p = t;
        succ = t->right;
        // 후계자를 찾아서 계속 왼쪽으로 이동한다.
        while(succ->left != NULL){
            succ_p = succ;
            succ = succ->left;
        }
        // 후속자의 부모와 자식을 연결
        if( succ_p->left == succ )
            succ_p->left = succ->right;
        else
            succ_p->right = succ->right;
        // 후속자가 가진 키값을 현재 노드에 복사
        t->key = succ->key;
        // 원래의 후속자 삭제
        t = succ;
    }
    free(t);
    return root;
}
```

```
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");

    root = delete_node(root, 40);
    inorder(root);
    return 0;
}
```



(왼쪽과 같음)

이진탐색(4/4)