해달 자료구조

4차시 - 트리

순서

1. 트리
2. 이진 트리
3. 이진 탐색 트리

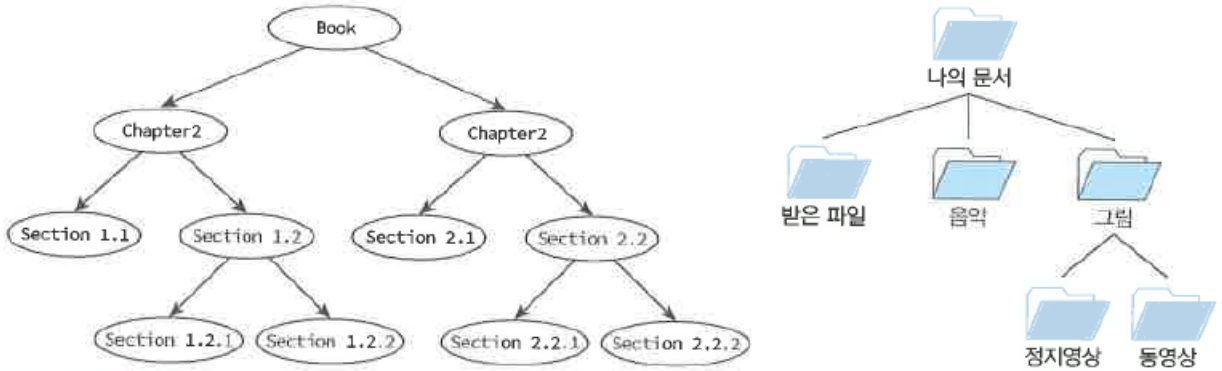
트리(Tree)

이때까지 다룬 자료구조들은 선형 자료구조(linear data structure)

계층적인 구조(hierarchical structure)를 가지는 자료를 표현하는 데에는 적합하지 않음

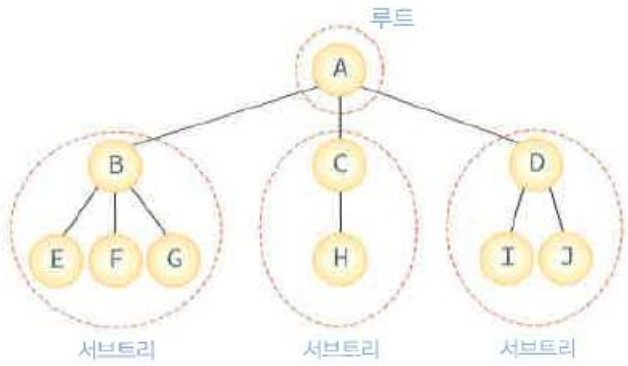
ex) 가계도, 회사 조직도, 컴퓨터의 디렉토리 구조 등

트리는 이런 계층적인 자료를 표현하는데 적합한 자료구조



[그림 8-2] 일상 생활에서 트리의 예

용어



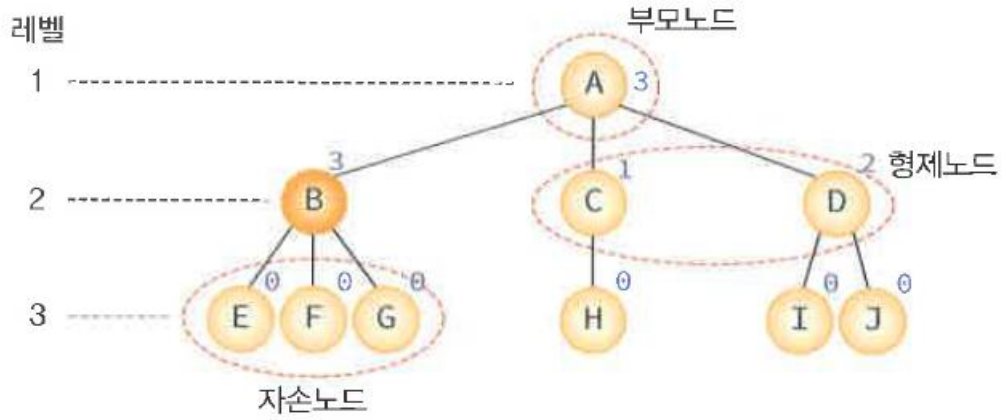
[그림 8-4] 트리의 용어: 루트, 서브트리

트리의 구성 요소에 해당하는 A~J를 노드(node)라고 함

트리는 한 개 이상의 노드로 이루어진 유한 집합
부모 노드가 없는 노드를 루트 노드(root node)

노드 간의 연결선을 간선(edge)

용어



노드의 차수(degree): 자식 노드의 개수

레벨(level): 트리의 각 층에 번호를 매기는 것

트리의 높이는 트리의 최대 레벨

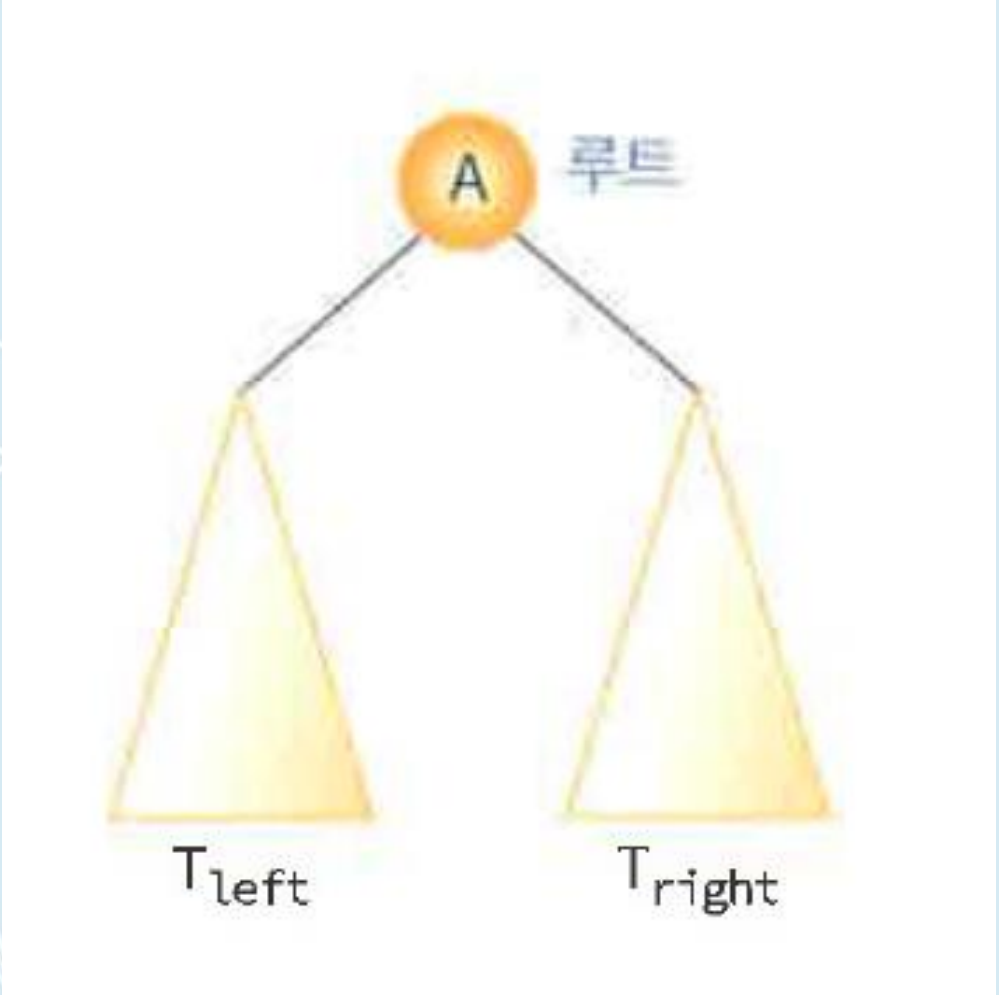
이진 트리

모든 노드가 2개의 서브 트리를 가지는 트리

서브 트리는 공집합일 수 있음

이진 트리는 다음과 같이 정의됨

(1) 공집합이거나
(2) 루트와 왼쪽 서브 트리, 오른쪽 서브 트리로 구성된 노드들의 유한 집합으로 구성된다. 이진 트리의 서브 트리들은 모두 이진 트리여야 한다.



이진 트리와 일반 트리의 차이점

- 이진 트리의 모든 노드는 차수가 2 이하. 반면 일반 트리는 자식 노드의 개수에 제한이 없음
- 이진 트리는 노드를 하나도 갖지 않을 수도 있음
- 이진 트리에는 서브 트리 간에 순서가 존재

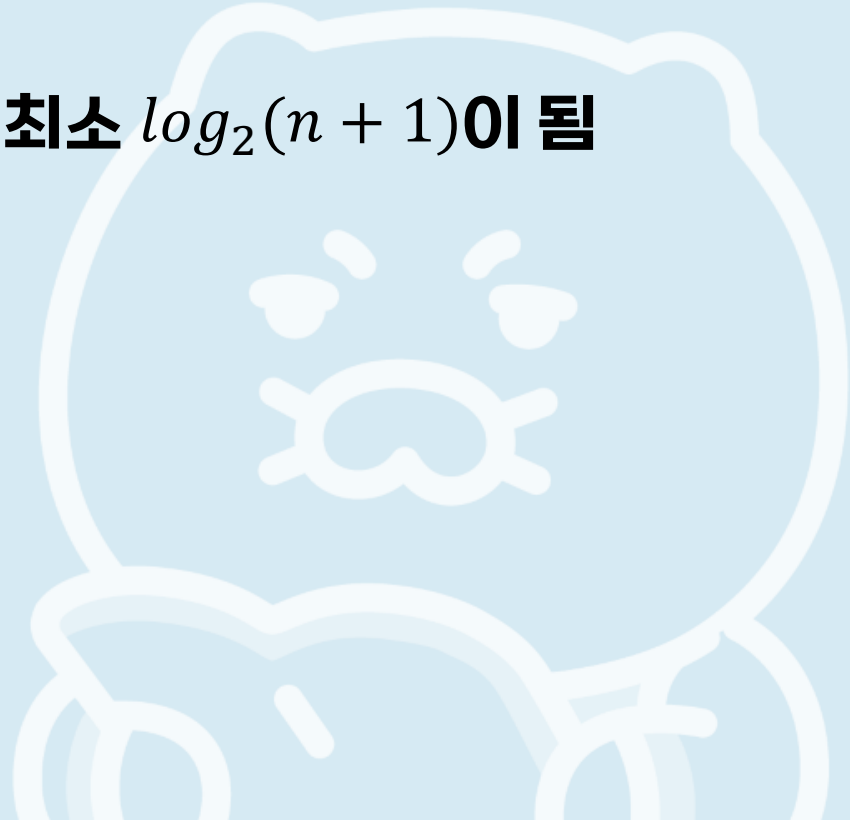
이진 트리의 성질

n개의 노드를 가지는 이진 트리는 n-1개의 간선을 가짐

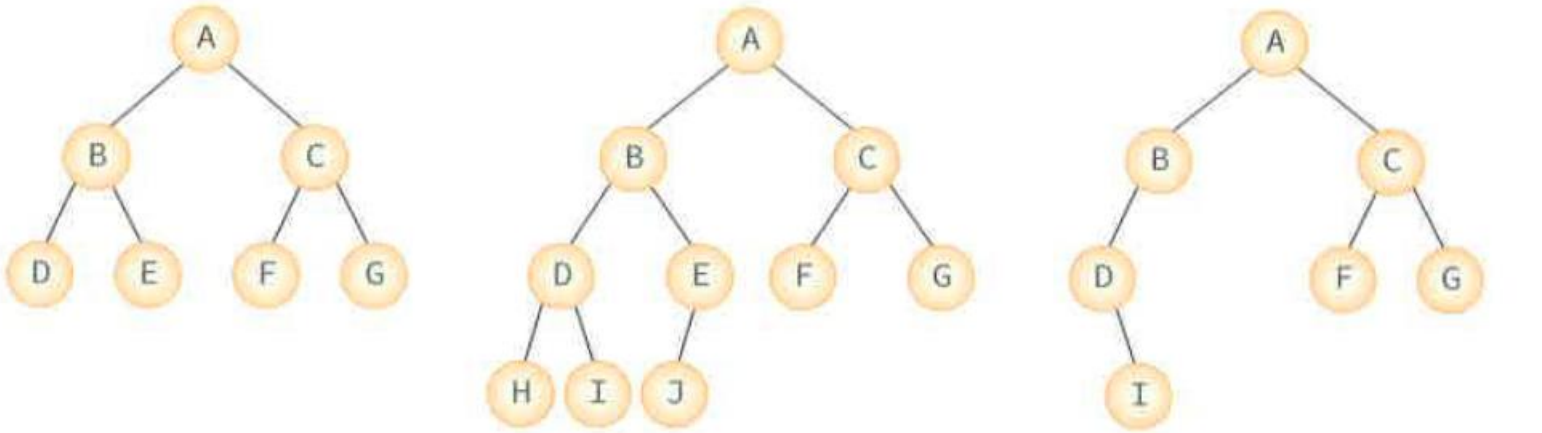
높이가 h인 이진 트리의 경우, 최소 h개의 노드를 가지며 최대 2^k-1개의 노드를 가짐

n개의 노드를 가지는 이진 트리의 높이는 최대 n이거나 최소 $log_2(n+1)$이 됨

교재 p.260참고



이진 트리의 분류



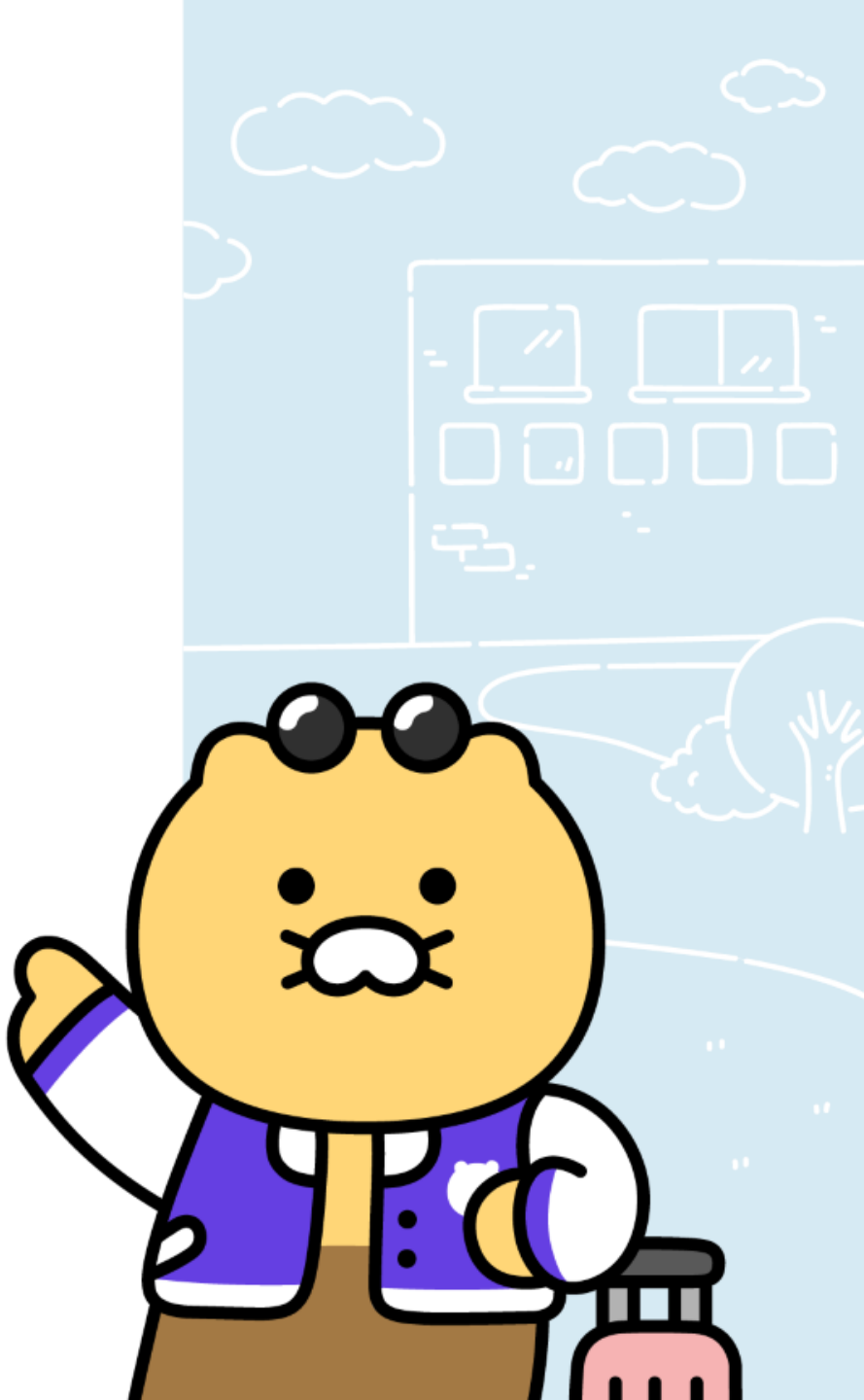
(a) 포화 이진트리 (b) 완전 이진트리 (c) 기타 이진트리

[그림 8-13] 이진트리의 종류

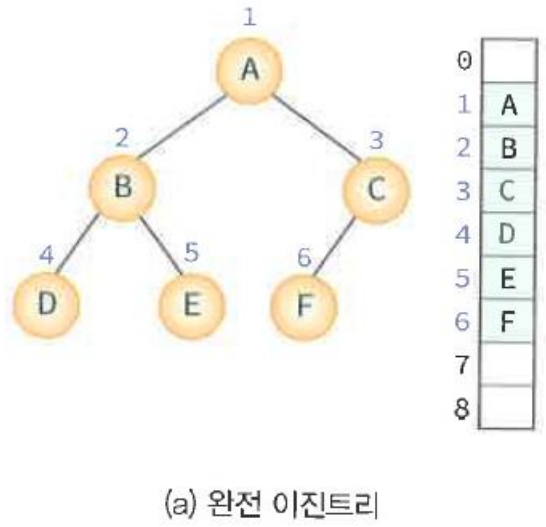
포화 이진트리 – 트리의 각 레벨에 노드가 꽉 차있는 이진트리
완전 이진트리 – 높이가 k일 때 레벨 1부터 k-1까지는 노드가 모두 채워져 있고 마지막 레벨 k에서는 왼쪽부터 오른쪽으로 노드가 순서대로 채워져 있는 이진트리

이진 트리의 표현

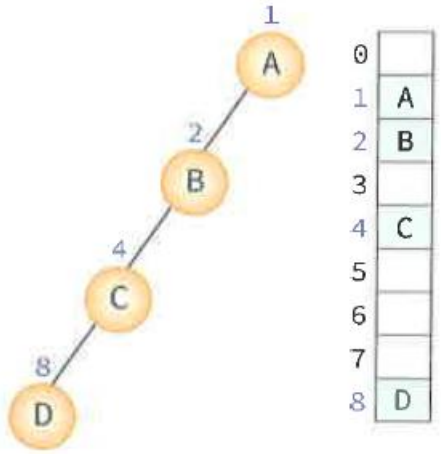
이진 트리를 표현하는 방법에는 2가지가 존재
- 배열을 이용하는 방법
- 포인터를 이용하는 방법



이진 트리의 표현 – 배열 이용



(a) 완전 이진트리



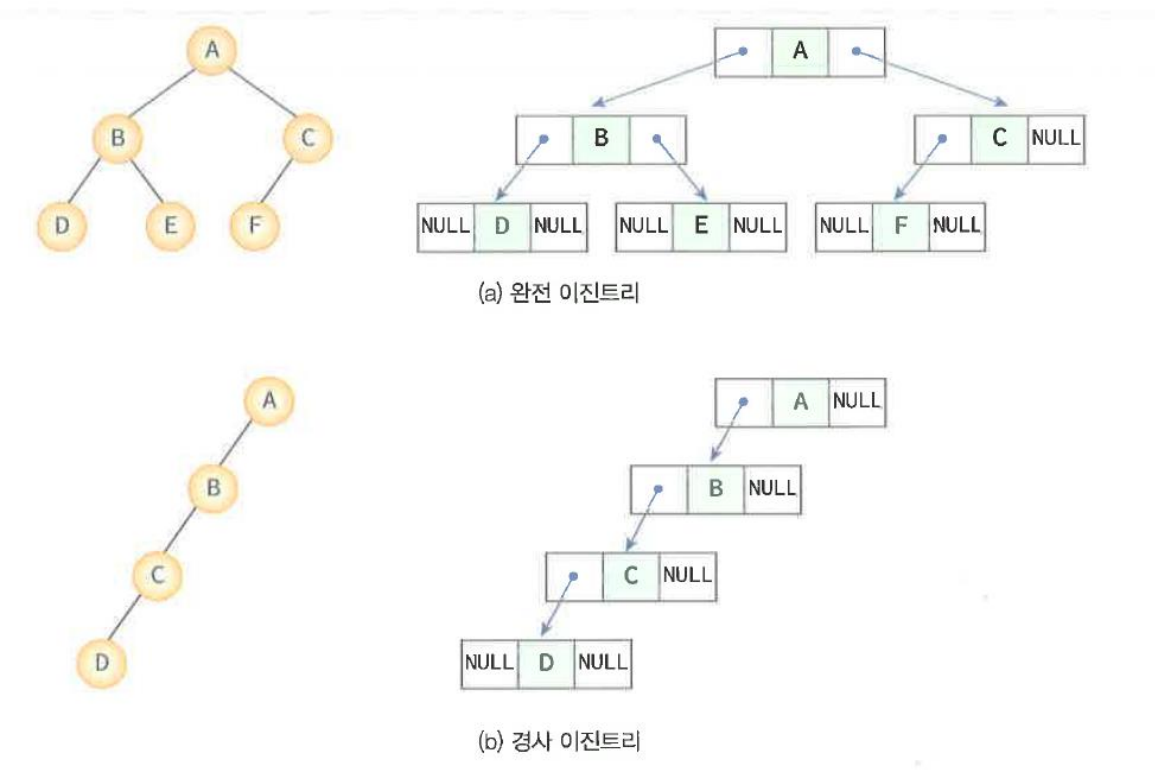
(b) 경사 이진트리

[그림 8-17] 완전 이진트리의 배열 표현법

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

주로 포화 이진 트리나 완전 이진 트리의 경우 많이 쓰이는 방법
완전 이진 트리로 가정한 후, 깊이가 k면 최대 $2^k - 1$개의 공간을 할당, 번호대로 노드들을 저장

이진 트리의 표현 – 포인터 이용(링크 표현법)




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

노드가 구조체로 표현되고, 포인터를 이용해 노드와 노드를 연결하는 방식

이진 트리의 순회

앞에서 다룬 자료구조들은 대개 데이터를 선형적으로 저장하고 있었기 때문에 자료를 순차적으로 순회하는 방법은 하나 뿐

그러나 트리는 여러가지 순서로 노드가 가지고 있는 자료에 접근할 수 있음

- 전위 순회
- 중위 순회
- 후위 순회

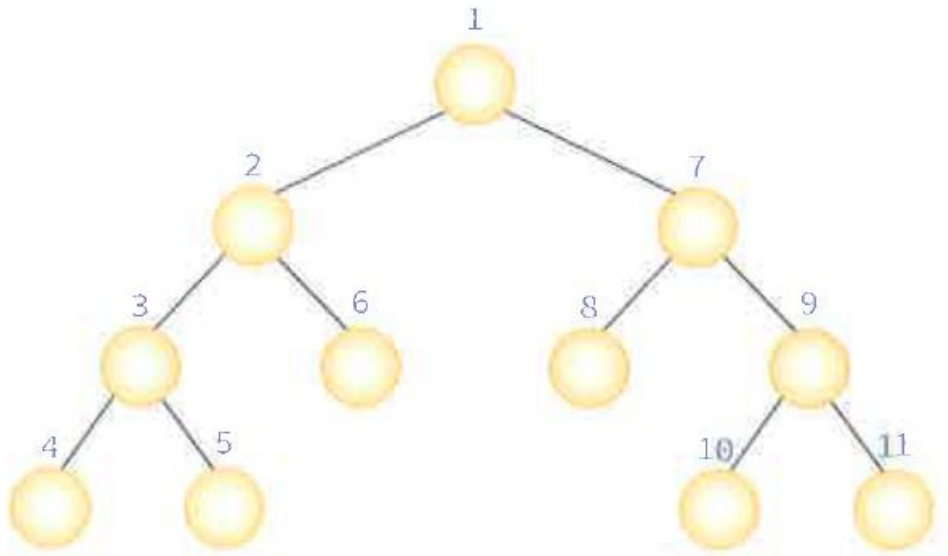
이진 트리의 순회 – 전위 순회

알고리즘 8.1 트리 전위 순회 알고리즘

```
preorder(x):

1.  if x≠NULL
2.      then   print DATA(x);
3.             preorder(LEFT(x));
4.             preorder(RIGHT(x));
```

알고리즘 설명

1. 노드 x가 NULL이면 더 이상 순환호출을 하지 않는다.
2. x의 데이터를 출력한다.
3. x의 왼쪽 서브트리를 순환호출하여 방문한다.
4. x의 오른쪽 서브트리를 순환호출하여 방문한다.



[그림 8-25] 전위순회에서의 노드 방문 순서

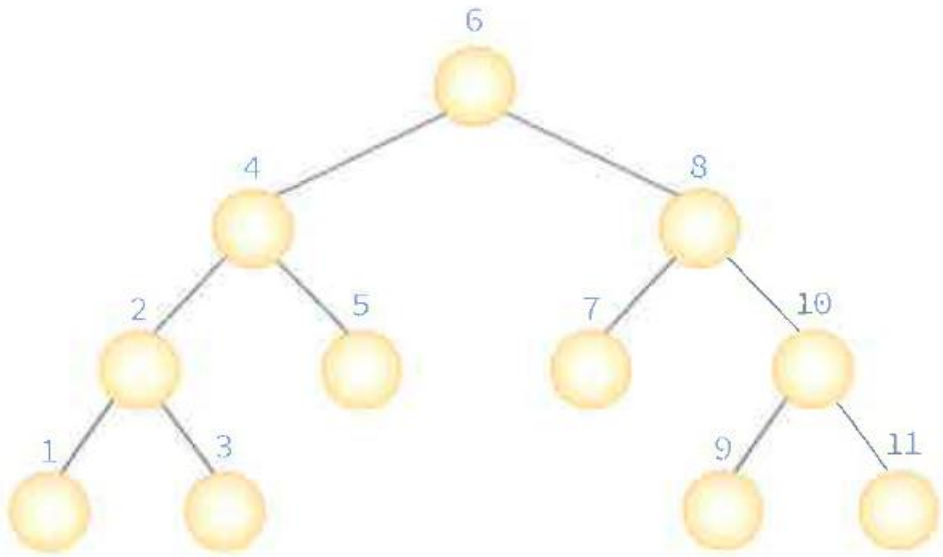
이진 트리의 순회 – 중위 순회

알고리즘 8.2 트리 중위 순회 알고리즘

```
inorder(x):

1.  if x≠NULL
2.      then   inorder(LEFT(x));
3.             print DATA(x);
4.             inorder(RIGHT(x));
```

알고리즘 설명

1. 노드 x가 NULL이면 더 이상 순환 호출을 하지 않는다.
2. x의 왼쪽 서브트리를 순환 호출하여 방문한다.
3. x의 데이터를 출력한다.
4. x의 오른쪽 서브트리를 순환 호출하여 방문한다.



[그림 8-27] 중위 순회의 예

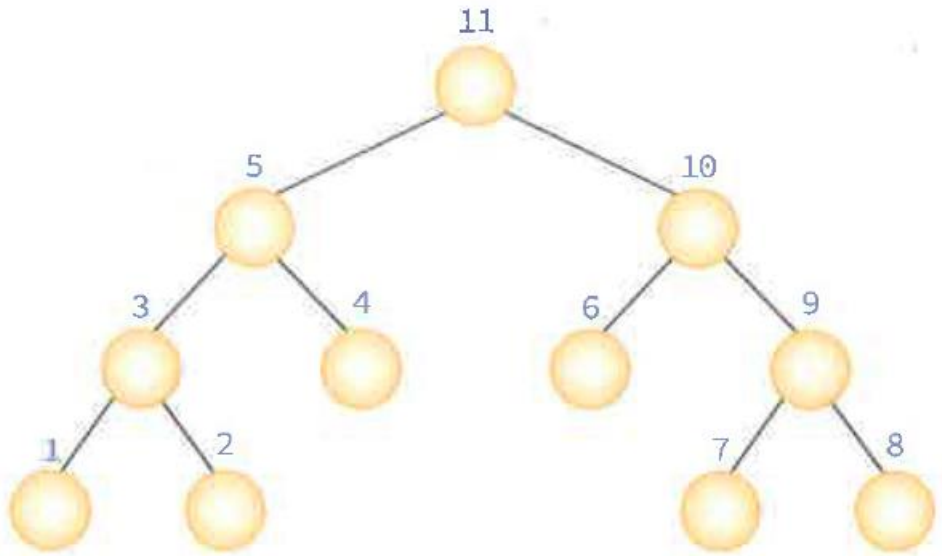
이진 트리의 순회 – 후위 순회

알고리즘 8.3 트리 후위 순회 알고리즘

```
postorder(x):

1.  if x≠NULL
2.      then   postorder(LEFT(x));
3.             postorder(RIGHT(x));
4.             print DATA(x);
```

알고리즘 설명

1. 노드 x가 NULL이면 더 이상 순환호출을 하지 않는다.
2. x의 왼쪽 서브트리를 순환호출하여 방문한다.
3. x의 오른쪽 서브트리를 순환호출하여 방문한다.
4. x의 데이터를 출력한다.



[그림 8-29] 후위 순회

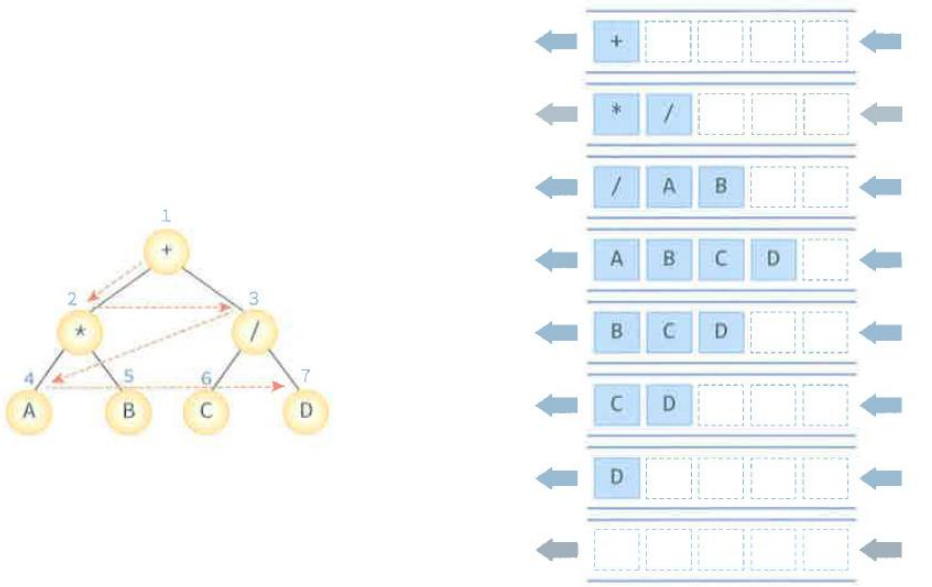
이진 트리의 순회 – 코드

프로그램 8.2 이진트리의 3가지 순회방법

```
// 중위 순회
void inorder(TreeNode *root) {
    if (root != NULL) {
        inorder(root->left);          // 왼쪽서브트리 순회
        printf("[%d] ", root->data); // 노드 방문
        inorder(root->right);         // 오른쪽서브트리 순회
    }
}
// 전위 순회
void preorder(TreeNode *root) {
    if (root != NULL) {
        printf("[%d] ", root->data); // 노드 방문
        preorder(root->left);         // 왼쪽서브트리 순회
        preorder(root->right);        // 오른쪽서브트리 순회
    }
}
// 후위 순회
void postorder(TreeNode *root) {
    if (root != NULL) {
        postorder(root->left);        // 왼쪽서브트리 순회
        postorder(root->right);       // 오른쪽서브트리순회
        printf("[%d] ", root->data); // 노드 방문
    }
}
```

이진 트리의 순회 – 레벨 순회

각 노드를 레벨 순으로 검사하는 순회 방법
동일한 레벨의 경우 좌에서 우로 방문



[그림 8-31] 레벨 순회

레벨 순회

큐에 하나라도 노드가 있으면 계속 반복

큐에 있는 노드를 꺼내어 방문한 다음,
그 노드의 자식 노드를 큐에 삽입

알고리즘 8.4 트리 레벨 순회 알고리즘

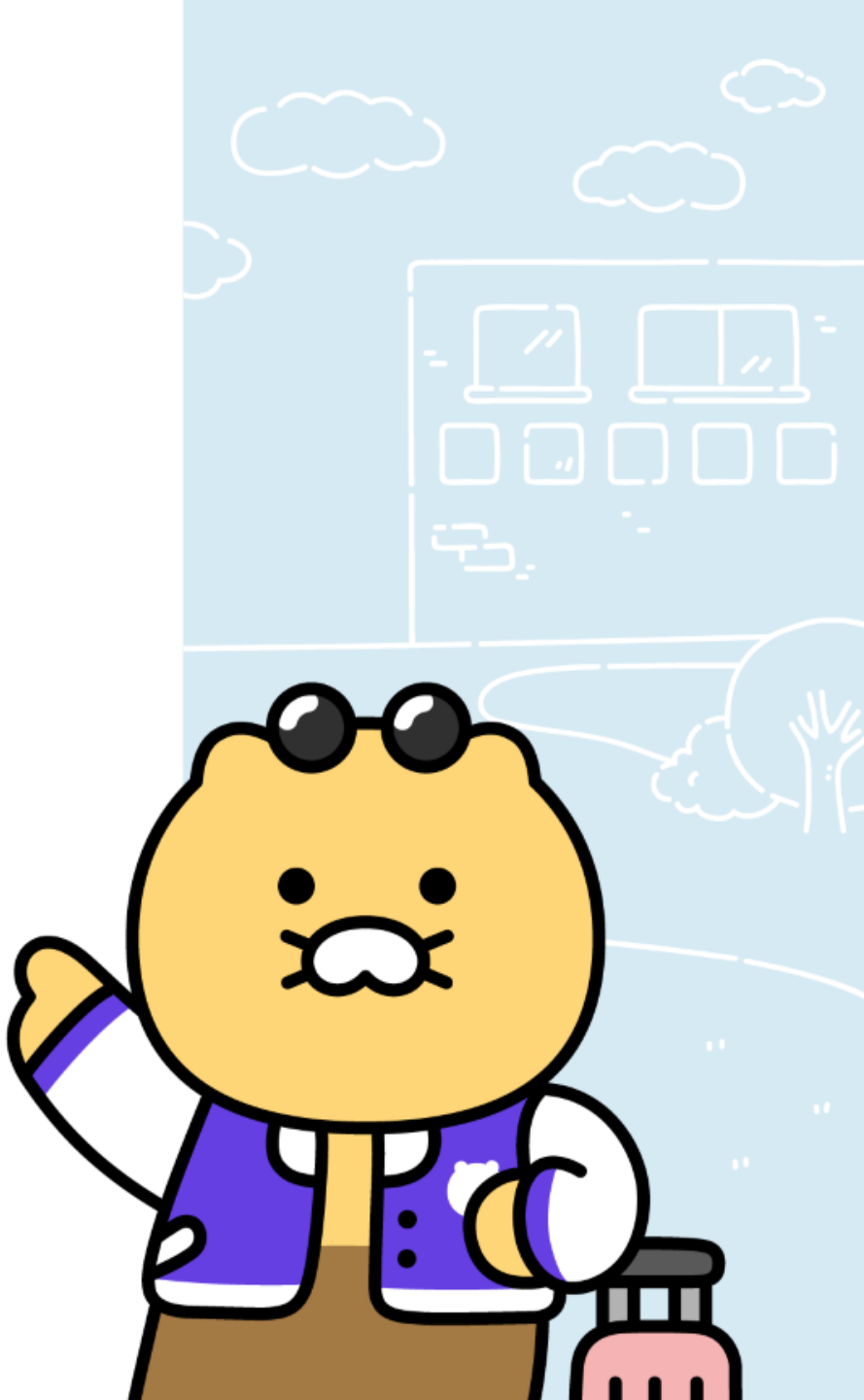
```
level_order(root):

1.  initialize queue;
2.  if(root == null) then return;
3.  enqueue(queue, root);
4.  while is_empty(queue)≠TRUE do
5.      x ← dequeue(queue);
6.      print x->data;
7.      if(x->left != NULL) then
8.                  enqueue(queue, x->left);
9.      if(x->right != NULL) then
10.                 enqueue(queue, x->left);
```

스레드 이진 트리

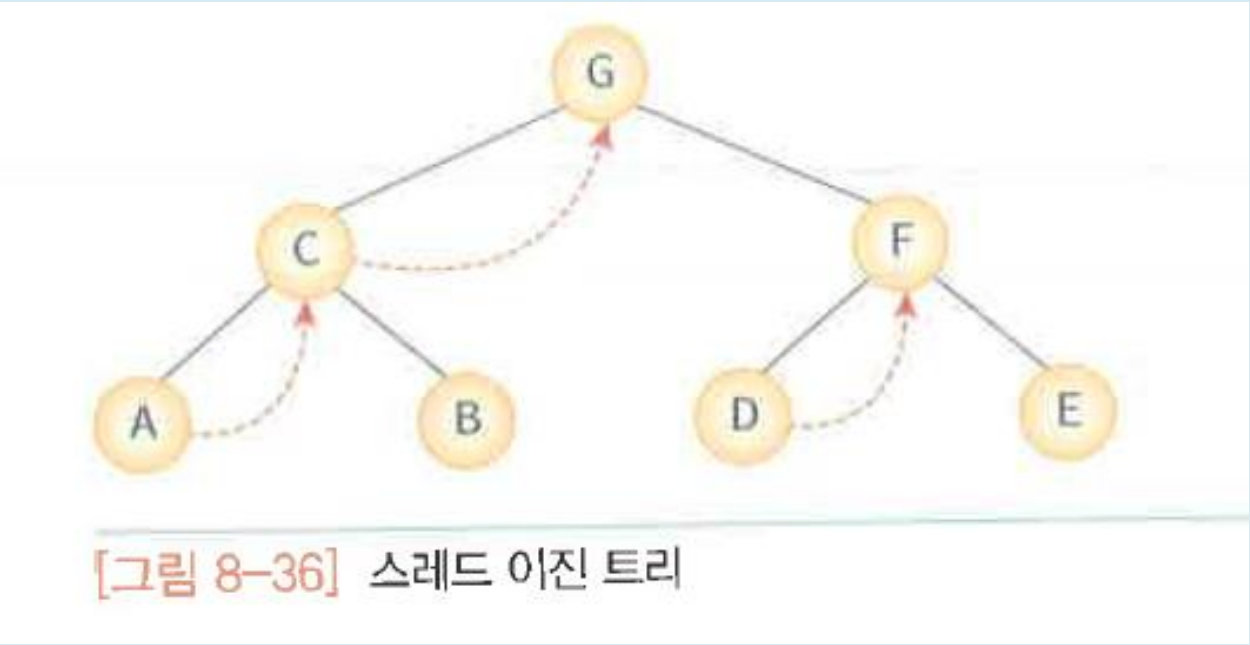
이진 트리 순회는 순환 호출을 사용함.
순환 호출은 함수를 호출해야 하므로 비효율적일 수 있음

순환 호출 없는 이진 트리 순회
→ NULL 링크를 잘 사용하면 가능

(중위순회 시)
NULL 링크에 선행 노드인 중위 선행자나 후속 노드인 중위 후속자를 저장시켜 놓은 트리가 스레드 이진 트리



스레드 이진 트리



[그림 8-36] 스레드 이진 트리

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

NULL 링크에 스레드가 저장되면 자식 노드를 가리키는 포인터가 저장되어 있는지,
스레드가 저장되어 있는지 구별해야 함
따라서 노드 구조가 오른쪽 사진 같이 변경됨

스레드 이진 트리

```c
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;
}
```

```c
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
}
```

NULL 링크에 스레드가 저장되면 자식 노드를 가리키는 포인터가 저장되어 있는지, 스레드가 저장되어 있는지 구별해야 함
따라서 노드 구조가 오른쪽 사진 같이 변경됨

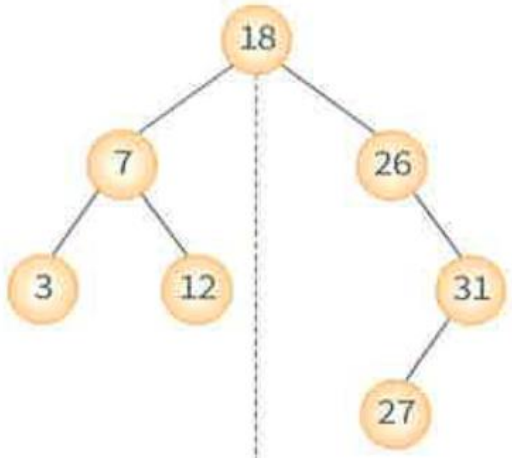
이진 탐색 트리

이진 트리 기반의 탐색을 위한 자료구조

이진 탐색 트리의 정의는 다음과 같음

정의 8.2 이진 탐색 트리

- 모든 원소의 키는 유일한 키를 가진다.
- 왼쪽 서브 트리 키들은 루트 키보다 작다.
- 오른쪽 서브 트리의 키들은 루트의 키보다 크다.
- 왼쪽과 오른쪽 서브 트리도 이진 탐색 트리이다.

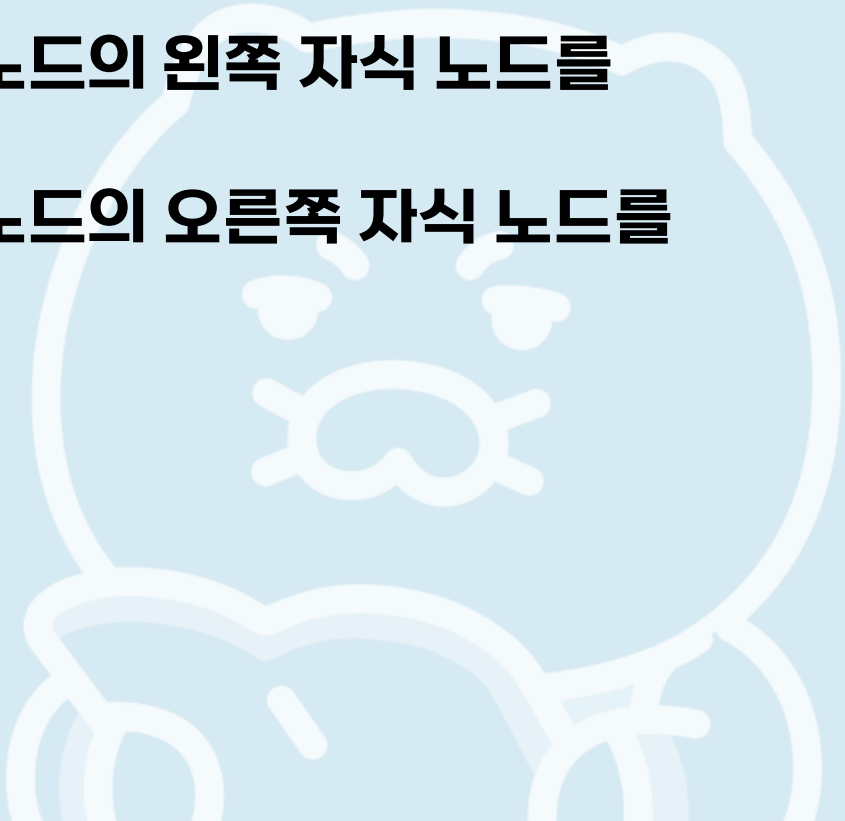


[그림 8-40] 이진탐색트리의 예

이진 탐색 트리 - 탐색

우선 찾고자 하는 값과 루트 노드의 값을 비교함. 비교 결과에 따라서 다음 3가지로 나누어짐

- 같다면 탐색 종료
- 찾고자 하는 값이 루트 노드의 값보다 작다면, 루트 노드의 왼쪽 자식 노드를 기준으로 탐색 다시 시작
- 찾고자 하는 값이 루트 노드의 값보다 크다면, 루트 노드의 오른쪽 자식 노드를 기준으로 탐색 다시 시작



이진 탐색 트리 – 탐색

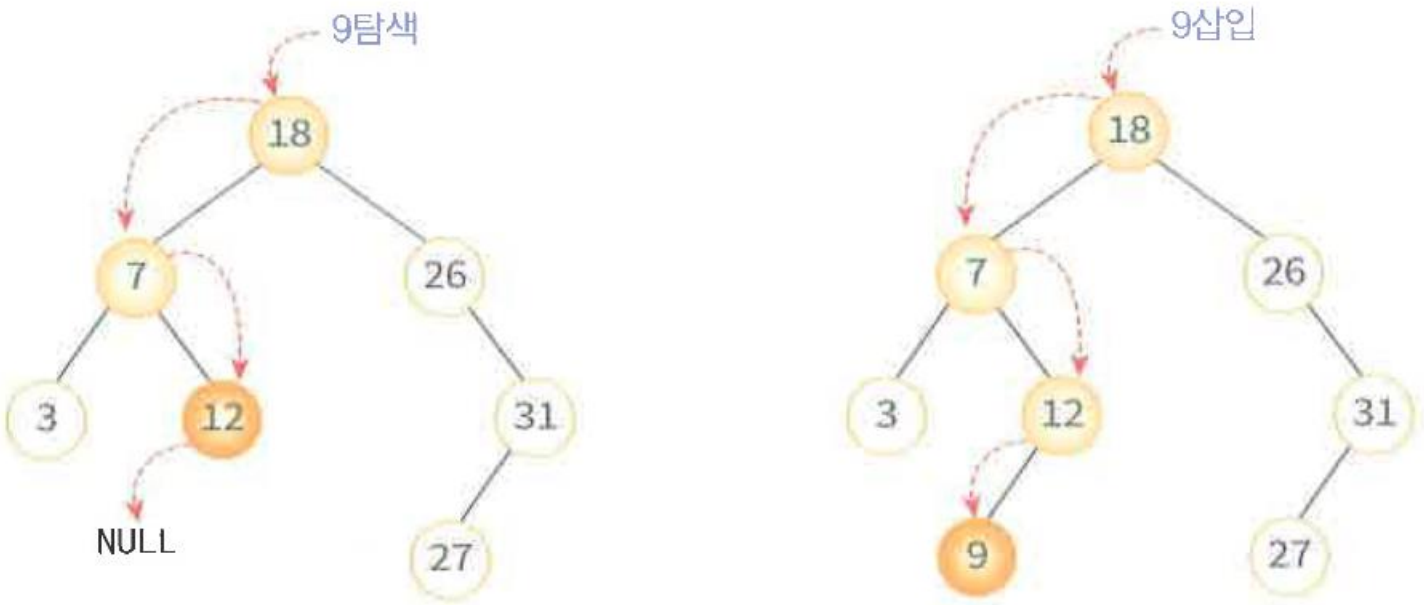
프로그램 8.9 순환적인 탐색함수

```
// 순환적인 탐색 함수
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);
}
```

프로그램 8.10 반복적인 탐색함수

```
// 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 트리 – 삽입



(a) 탐색을 먼저 수행

(b) 탐색이 실패한 위치에 9를 삽입

[그림 8-42] 이진탐색트리에서의 삽입연산

이진 탐색 트리 – 삽입

```c
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;
}
```

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

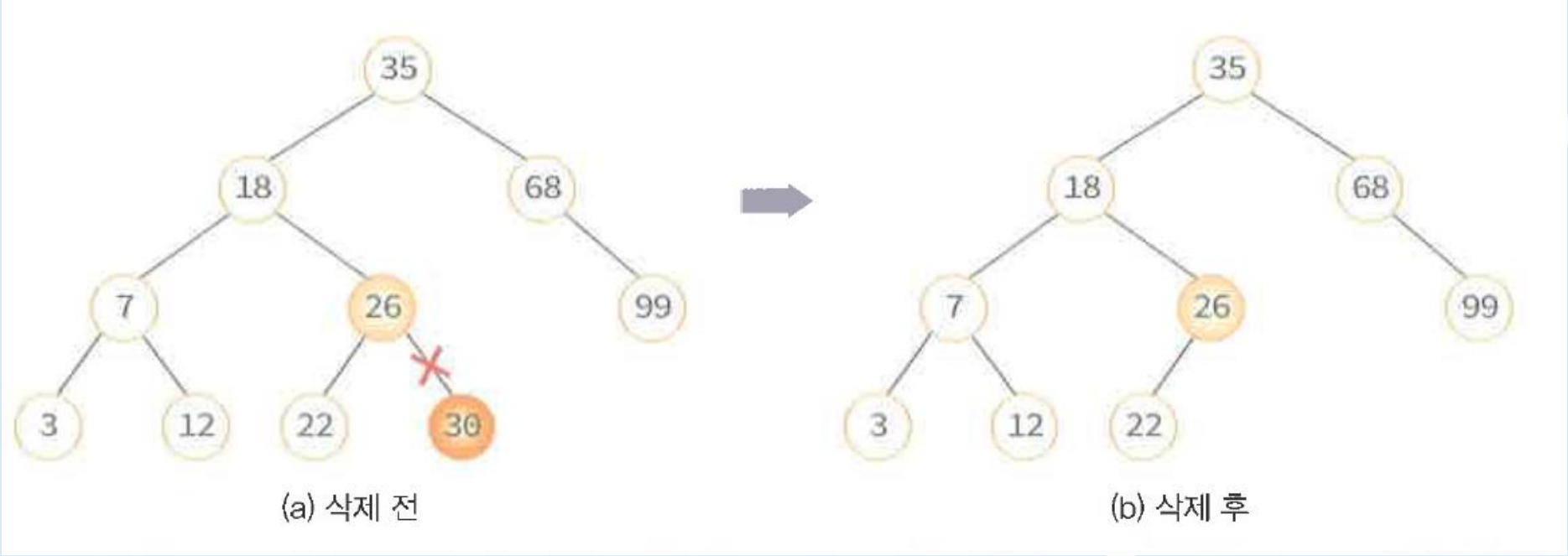
이진 탐색 트리 – 삭제

삽입과 마찬가지로 탐색이 선행되어야 하고, 그 후 다음 3가지 경우를 고려해야함

삭제하려는 노드가

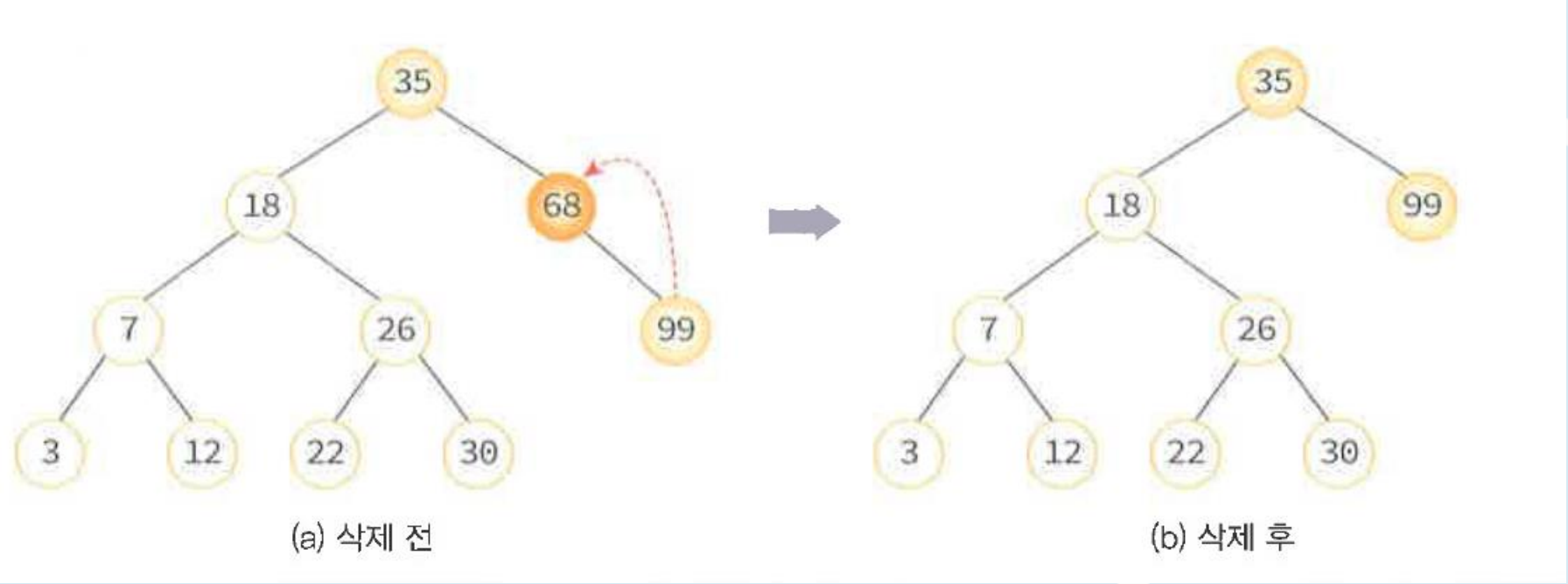
1. 단말 노드일 경우
2. 하나의 왼쪽이나 오른쪽 서브 트리 중 하나만 가지고 있는 경우
3. 두 개의 서브 트리 모두 가지고 있는 경우

이진 탐색 트리 삭제 – 첫 번째 경우



(a) 삭제 전 (b) 삭제 후

이진 탐색 트리 삭제 – 두 번째 경우



(a) 삭제 전 (b) 삭제 후

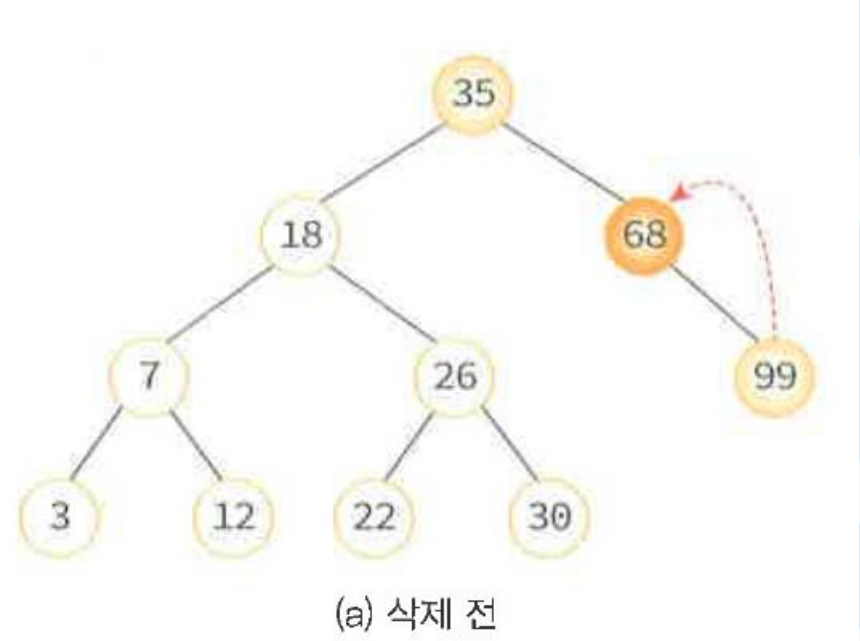
이진 탐색 트리 삭제 – 세 번째 경우

문제는 삭제하려는 노드의 위치로 어떤 노드를 가져올 것이냐는 것

자식 노드를 그대로 가져오는 것은 안됨

오른쪽 그림에서 18을 제거하고
70이나 26을 가져오게 되면 이진 탐색 트리의
조건을 만족하지 않음

따라서, 삭제하려는 노드와 **가장 가까운 값**을
가져와야 함



(a) 삭제 전

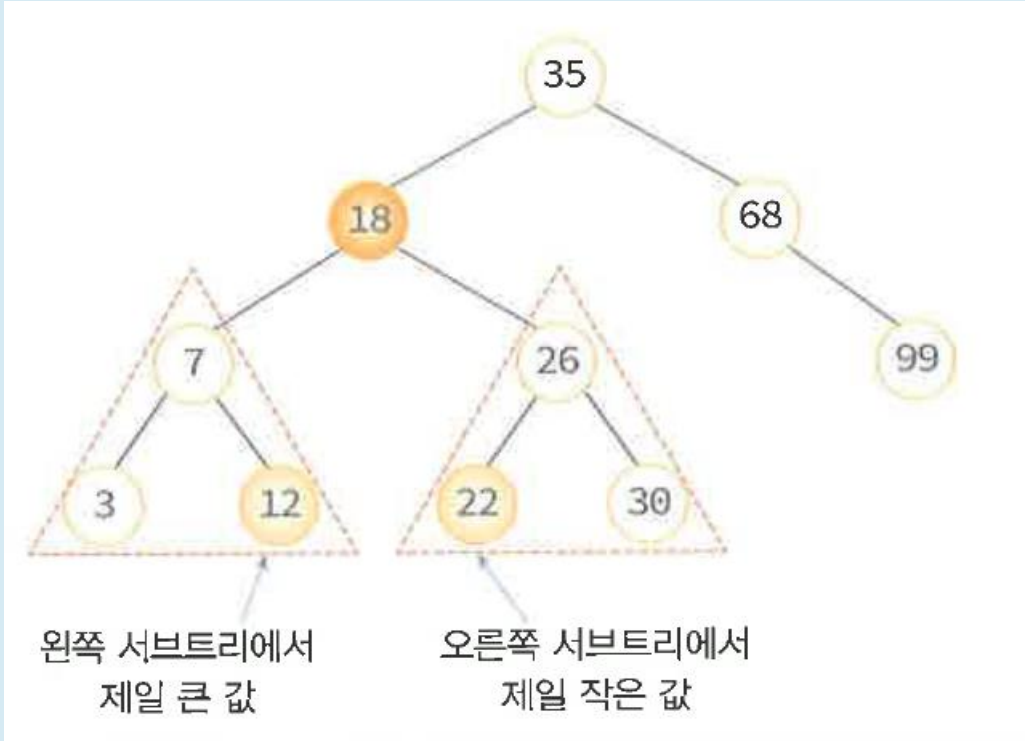
이진 탐색 트리 삭제 – 세 번째 경우

삭제하려는 노드와 가장 가까운 값은
왼쪽 서브트리에서 제일 큰 값 혹은
오른쪽 서브트리에서 제일 작은 값

또한 이 노드들은
중위순회할 때 선행노드와 후속노드에 해당

둘 중 어느 노드를 선택하든 상관없음

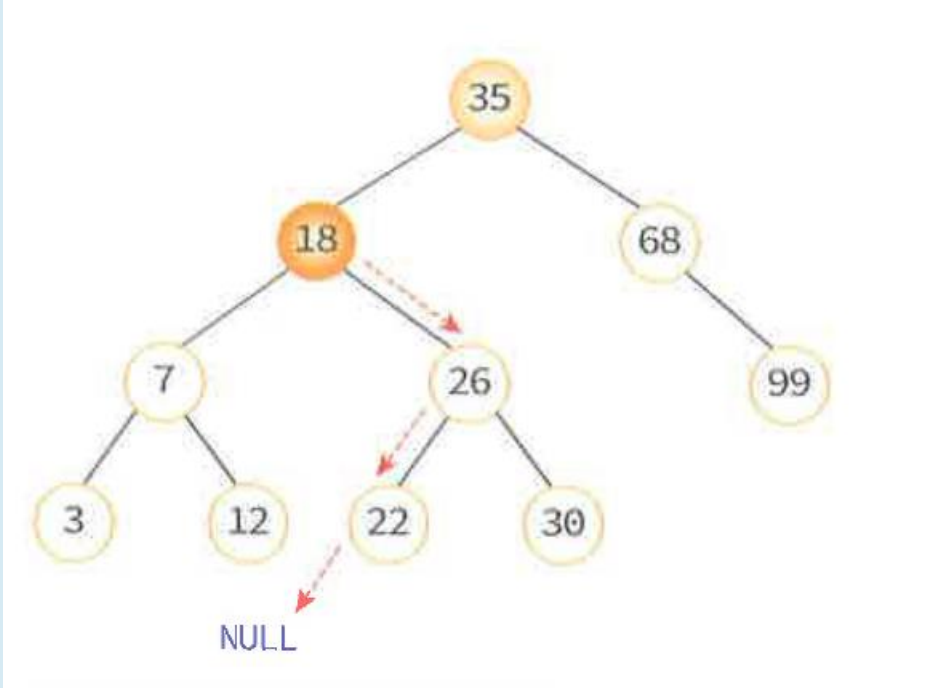
여기서는 교재를 따라서 후자를 선택함



이진 탐색 트리 삭제 – 세 번째 경우

삭제되는 노드의 오른쪽 서브트리에서
가장 작은 값을 갖는 노드는

오른쪽 서브트리에서 왼쪽 자식 링크를 타고
NULL을 만날 때까지 진행하면 됨



이진 탐색 트리 삭제 코드

```
    // 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);

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

감사합니다 :)