7 장. 트리

Jinseog Kim

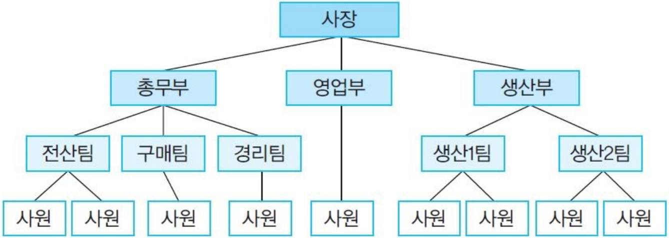
컴퓨터 공학과
동국대학교 WISE 캠퍼스

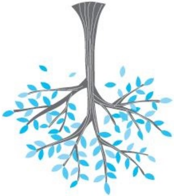
학습 내용

1. 트리 데이터 구조를 이해한다.
2. 이진 트리와 이진 트리의 종류를 이해
3. 이진 트리를 이용하여 순회 및 탐색을 할 수 있다.
4. 힙 트리를 이해하고 이용할 수 있다.

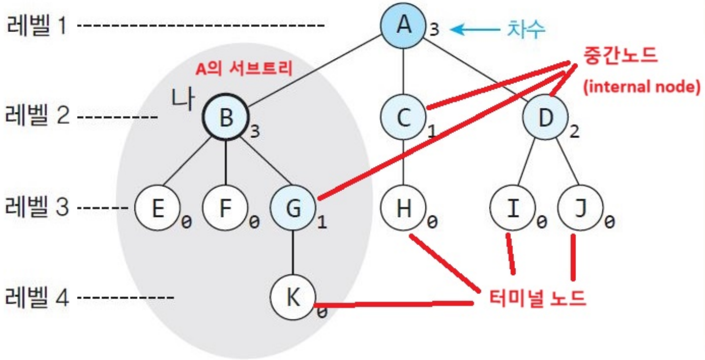
트리

- 나무를 닮은 자료구조
- 계층적인 관계 표현
- 가계도, 컴퓨터의 폴더 구조, 탐색 트리, 힙 트리, 결정 트리 등





트리의 용어



- 루트 노드: A
- B의 부모노드: A
- B의 자식 노드: E, F, G
- B의 자손 노드: E, F, G, K
- K의 조상 노드: G, B, A
- B의 형제 노드: C, D
- B의 차수: 3
- ~~단말 노드~~: E, F, K, H, I, J
- ~~비단말 노드~~: A, B, C, D, G
- 트리의 높이: 4
- 트리의 차수: 3

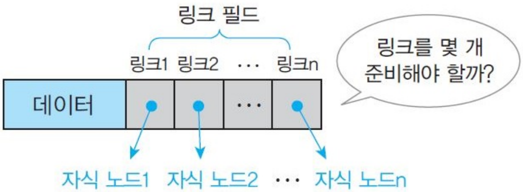
트리의 용어

1. 루트 (root) 노드: 주어진 트리의 시작 노드
2. 자식 (children) 노드: 어떤 노드와 직접 연결된 하위 노드
3. 부모 (parent) 노드: 어떤 노드와 직접 연결된 상위 노드
4. 터미널 (leaf, termial) 노드: 자손에 하나도 없는 노드
5. 중간 (internal) 노드: 루트도 아니고 잎 노드도 아닌 노드
6. 형제 (sibling) 노드: 동일한 부모 노드를 가지는 노드
7. 조상 (ancestor) 노드: 루트에서 그 노드사이의 경로에 있는 노드
8. 자손 (descendant) 노드: 그 노드에서 터미널 노드까지의 경로에 있는 모든 노드들

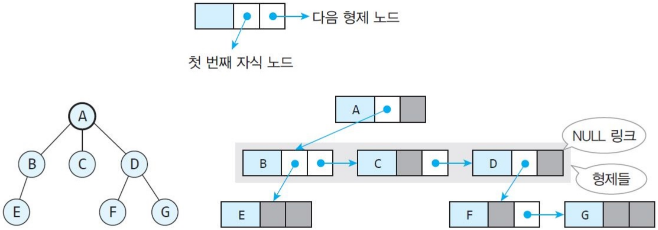
트리의 용어

1. 차수 (degree): 그 노드의 서브 트리의 개수 (즉 자식노드의 개수)
2. 레벨 (level): 루트의 레벨을 0 또는 1 로 놓고 자손 노드로 내려가면서 하나씩 증가
3. 깊이 (depth) 또는 높이 (height): 트리에서 노드가 가질 수 있는 최대 레벨
4. n-트리 (n-ary tree): 모든 중간노드가 최대 n 개의 자식노드를 가지는 트리
    - $n = 2$ 이면 이진트리 (binary tree)

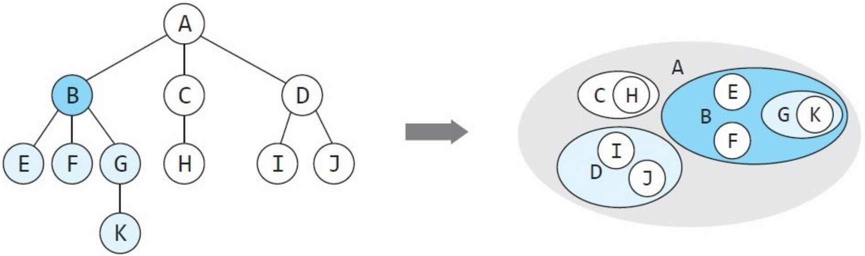
- N-링크 표현



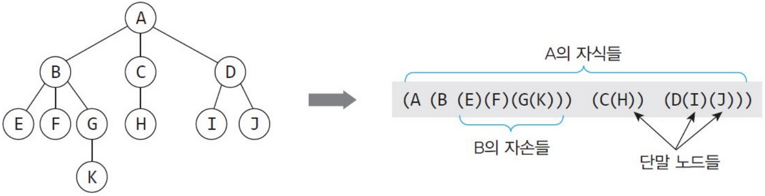
- 왼쪽 자식-오른쪽 형제 표현



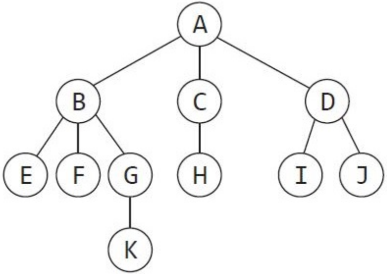
- 중첩된 집합

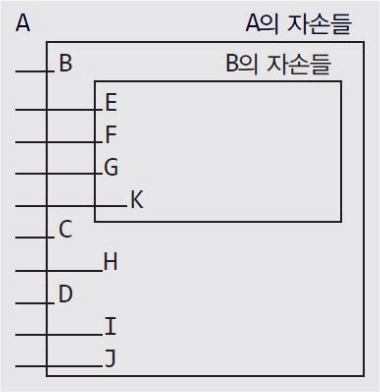


- 중첩된 괄호

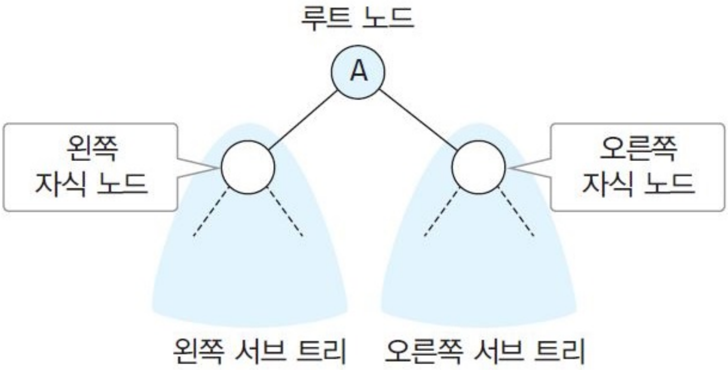


- 들여쓰기 (indentation)

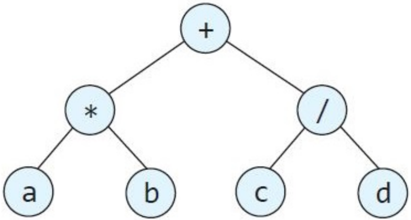




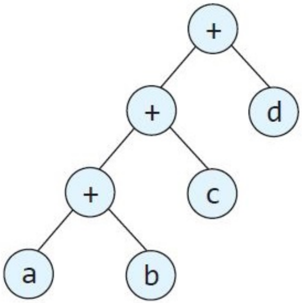
- **이진 트리 (binary tree):** 자식의 수를 최대 2 개로 제한한 트리
  - 왼쪽 자식과 오른쪽 자식은 반드시 구별



- 예: 수식 트리

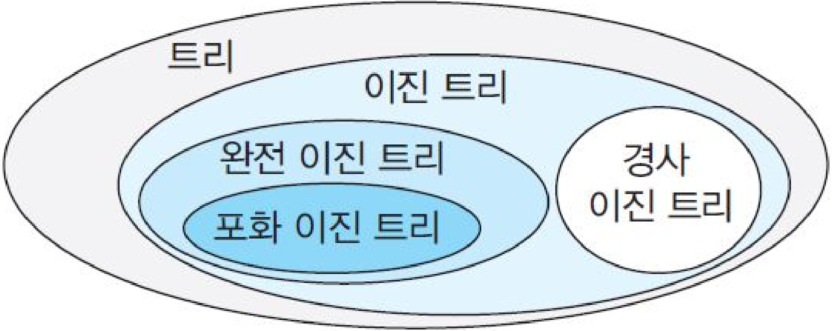


(a) (a*b)+(c/d)



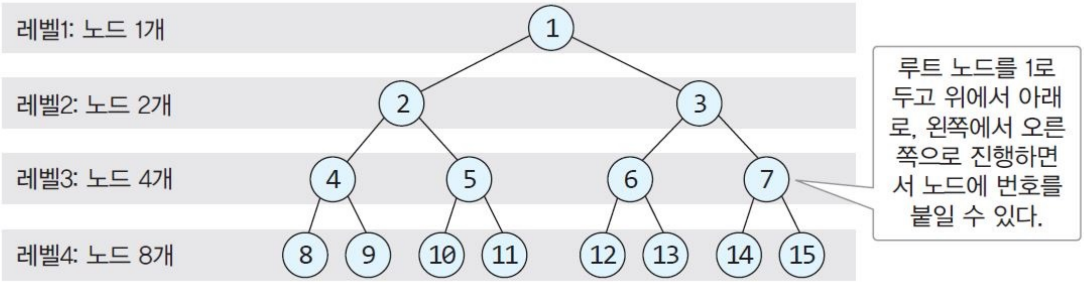
(b) ((a+b)+c)+d

이진 트리의 종류

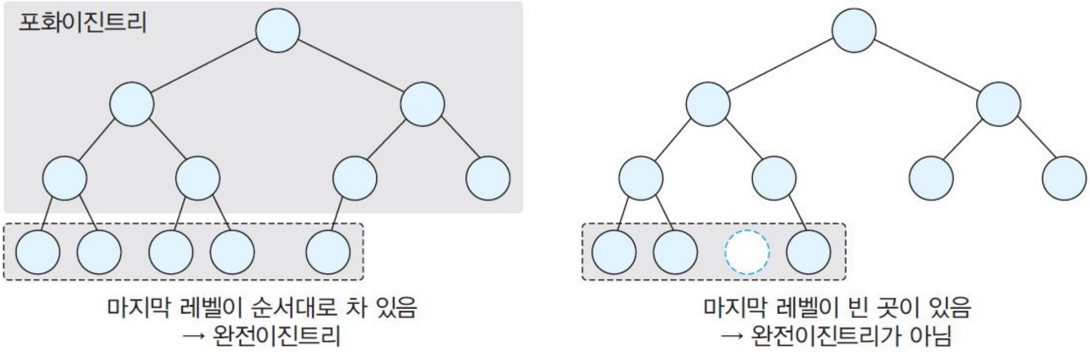


이진 트리의 종류 - 포화 이진 트리

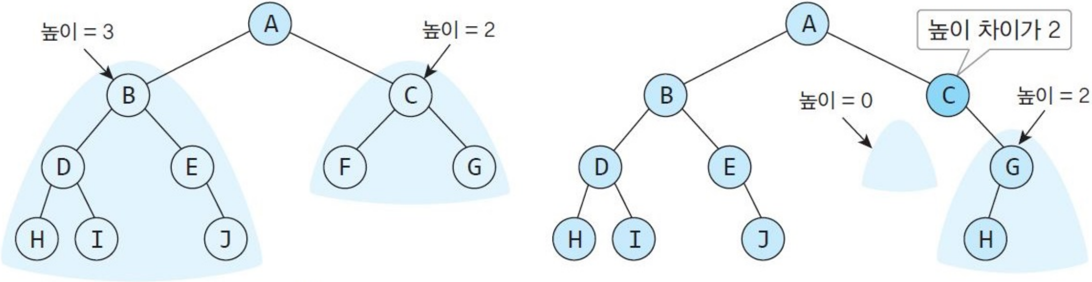
- 포화 이진 트리 (full binary tree): 터미널 노드를 제외한 모든 노드가 2 개의 자식노드를 가지는 트리



- 완전 이진 트리 (complete binary tree): 높이가 h 일 때 레벨 $h-1$ 까지는 모두 차 있고 레벨 h 에서는 왼쪽 노드부터 차례로 차 있는 이진 트리임 (예: 힙 트리)



- 균형 이진 트리 (balanced binary tree): 모든 노드에서 좌우 서브트리의 높이의 차이가 1 이하인 트리

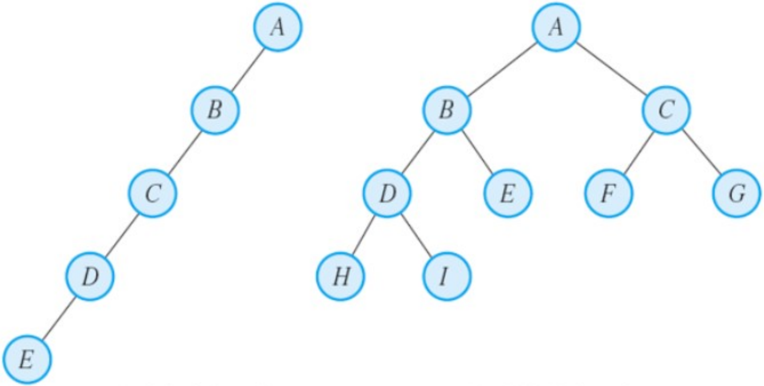


(a) 모든 노드의 좌우 서브트리 높이 차이가 1 이하임 → 균형이진트리

(b) 노드 C의 좌우 서브트리 높이 차이가 2임(1 초과) → 균형이진트리가 아님

이진 트리의 종류 - 사향 이진 트리

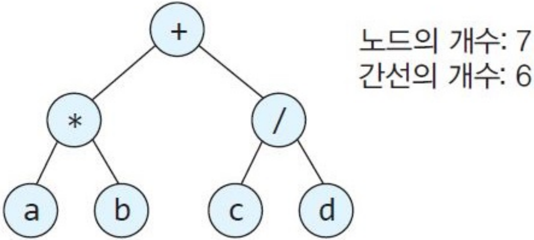
- 사향 이진 트리 (skewed binary tree): 왼쪽 또는 오른쪽으로 편향된 트리의 구조를 가짐



(1) 사향 이진 트리　　(2) 완전 이진 트리

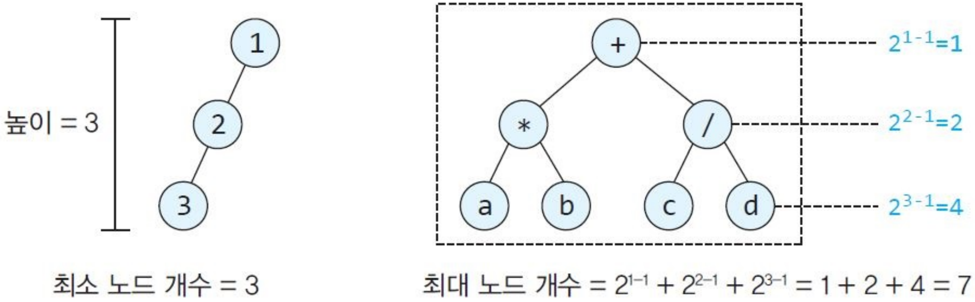
이진 트리의 성질

1. n 개의 노드를 가진 트리는 반드시 $n-1$ 개의 간선 (edge, link, relation)



이진 트리의 성질

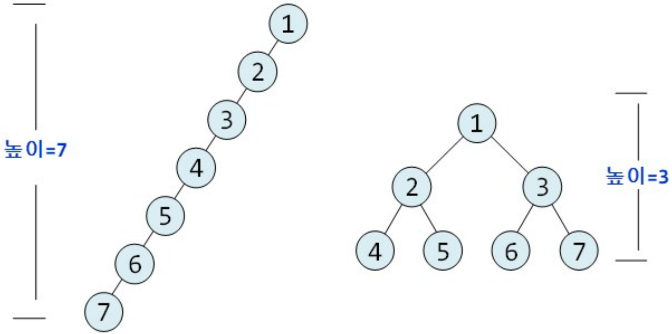
2. 높이가 h 인 이진 트리의 최대 노드 수: $n \leq 2^h - 1$



$$n \leq 2^h - 1 = 2^3 - 1 = 7$$

이진 트리의 성질

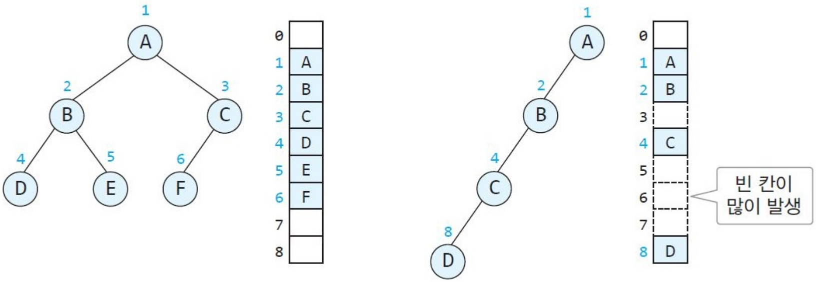
3. n 개 노드의 이진 트리의 최소 높이: $h \geq \log_2(n+1)$



$$h \geq \log_2(n+1) = \log_2(8) = 3$$

이진 트리의 표현 방법 (1. 배열 표현법)

- 이진 트리를 포화 이진 트리의 일부라고 생각하고 노드에 번호를 붙이는 것을 그대로 배열의 인덱스로 사용



(a) 완전 이진 트리
(배열 중간에 빈 칸이 없음)

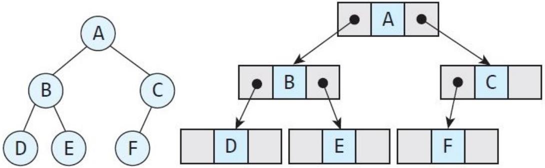
(b) 경사 이진 트리
(중간에 빈 칸이 많이 발생)

```
TElement t1[MAX_NSIZE] = {0, 'A', 'B', 'C', 'D', 'E', 'F', 0, 0}; //(a)
TElement t2[MAX_NSIZE] = {0, 'A', 'B', 0, 'C', 0, 0, 0, 'D', 'E', 'F', 0, 0}; //(b)
```

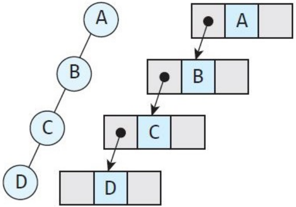
- 이중 연결 구조를 이용


```
typedef struct TNode {
    TElement      data;   // 노드에 저장할 자료
    struct TNode*  left;  // 왼쪽 서브 트리
    struct TNode*  right; // 오른쪽 서브 트리
} TNode;
```

이진 트리의 표현 방법 (2. 링크 표현법)



(a) 완전 이진 트리



(b) 경사 이진 트리

```
TNode f = {'F', NULL, NULL};
TNode e = {'E', NULL, NULL};
TNode d = {'D', NULL, NULL};
TNode c = {'C', &f, NULL};
TNode b = {'B', &d, &e};
TNode a = {'A', &b, &c};
TNode * t1 = &a;
```

```
TNode D = {'D', NULL, NULL};
TNode C = {'C', &D, NULL};
TNode B = {'B', &C, NULL};
TNode A = {'A', &B, NULL};
TNode * t2 = &A;
```

- 트리의 동적 생성 함수: create_tree(data, left, right)

```
TNode* create_tree(TElement data, TNode* left, TNode* right)
{
    TNode* n = (TNode*)malloc(sizeof(TNode)); // 루트노드 할당
    n->data = data;
    n->left = left;  // 왼쪽 서브트리 연결
    n->right = right; // 오른쪽 서브트리 연결
    return n; // 루트노드 반환
}
```

Lab: 동적 할당을 이용한 연결된 구조 표현

- 트리의 동적 해제 함수: delete_tree(n) 먼저 좌우 서브 트리를 삭제

```
void delete_tree(TNode* n)
{
    if (n != NULL) {
        delete_tree(n->left);   // 왼쪽 서브트리 삭제
        delete_tree(n->right);  // 오른쪽 서브트리 삭제
        free(n);                // 현재 노드 삭제
    }
}
```

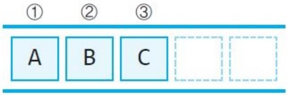
Lab: 동적 할당을 이용한 연결된 구조 표현

- 테스트

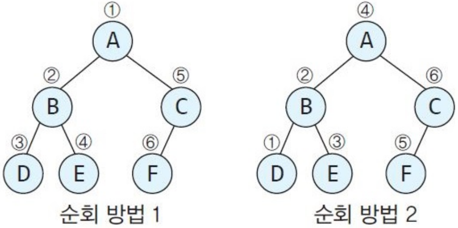
```c
void main()
{
    /* 그림 7.17(a) 트리 동적 생성 */
    TNode* d = create_tree('D', NULL, NULL);
    TNode* e = create_tree('E', NULL, NULL);
    TNode* b = create_tree('B', d, e);
    TNode* f = create_tree('F', NULL, NULL);
    TNode* c = create_tree('C', f, NULL);
    TNode* root1 = create_tree('A', b, c);

    /* 트리 동적 해제 */
    delete_tree(root1);
}
```

- **이진 트리의 순회 (binary tree traversal):** 각각의 노드를 정확히 한번만 체계적인 방법으로 방문하는 과정
  - 트리 순회의 결과로 각 노드에 들어 있는 데이터를 차례로 탐색하게 됨
  - 트리는 순회 방법이 다양함
  - 트리 순회는 각 노드와 그 노드의 서브 트리 (subtree) 를 같은 방법으로 순회할 수 있음 (재귀적 방법)

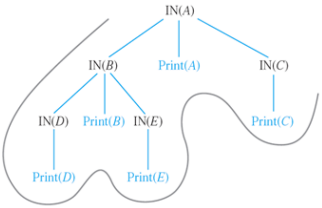


(a) 선형 자료 구조는
순회 방법이 단순함

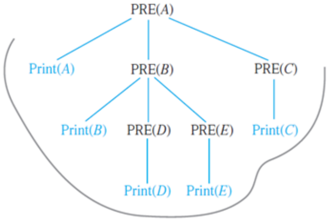


(b) 트리는 순회 방법이 다양함

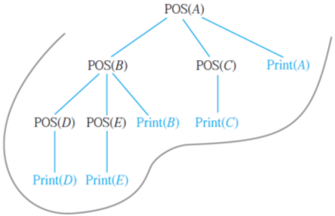
- 이진트리의 순회 방법은 다음과 같이 여러 방법이 있음
  1. 중순위 순회 (inorder traversal) = 대칭 순회 (symmetric)
  2. 전순위 순회 (preorder traversal) = 깊이 우선 순회 (depth-first traversal)
  3. 후순위 순회 (postorder traversal)
  4. 레벨 순서 순회 (level-order), 너비 우선 순회 (breadth-first traversal): 레벨이 낮은 순서로 순회



〈그림 8.10〉 중순위 탐방



〈그림 8.11〉 전순위 탐방

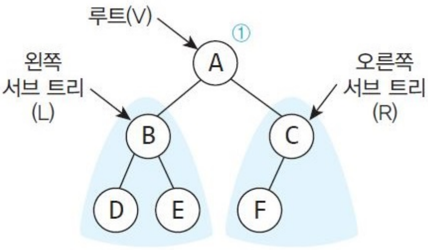


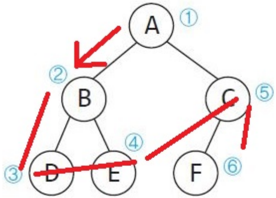
〈그림 8.12〉 후순위 탐방

- 방법에 따른 탐색 순서

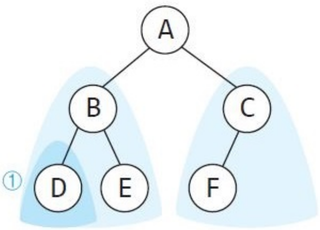
순회 방법	탐색순서
중순위 탐색	Left child $\longrightarrow$ **Data** $\longrightarrow$ Right child
전순위 탐색	**Data** $\longrightarrow$ Left child $\longrightarrow$ Right child
후순위 탐색	Left child $\longrightarrow$ Right child $\longrightarrow$ **Data**
레벨순서 탐색	레벨이 낮은 순서로 순회

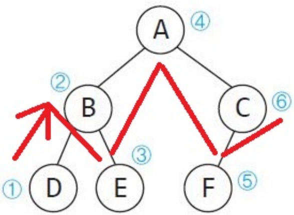
트리의 순회 방법- 전순위 순회





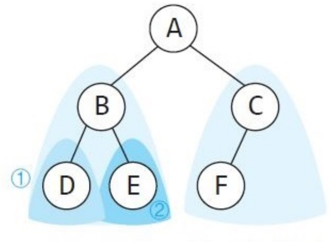

```c
// 전순위
void preorder(TREE * current){
  if(current != NULL){
    printf("%c", current->data)
    preorder(current->left);
    preorder(current->right);
  }
}
```

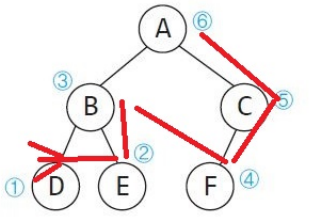





```
// 중순위
void inorder(TREE * current){
  if(current != NULL){
    inorder(current->left);
    printf("%c", current->data)
    inorder(current->right);
  }
}
```

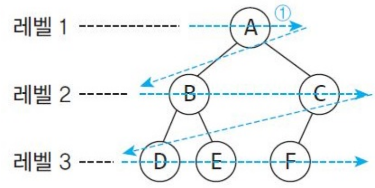
트리의 순회 방법- 후순위 순회

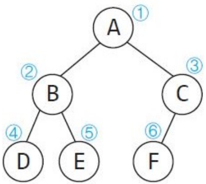


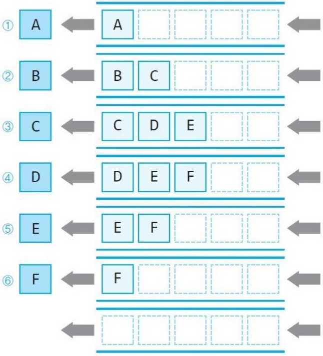



```
// 후순위
void postorder(TREE * current){
  if(current != NULL){
    postorder(current->left);
    postorder(current->right);
    printf("%c", current->data)
  }
}
```

트리의 순회 방법- 레벨 순회 (level order)





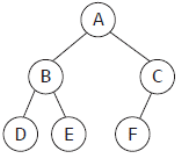



```
levelorder(root)
    if current != NULL :
        init_queue()
        enqueue(root)
        while !is_empty(queue) :
            n <- dequeue()
            if n is not NULL :
                VisitNode(n)
                enqueue(n->left)
                enqueue(n->right)
```

트리순회의 구현

- sample code: ch07/TreeTraversal.c

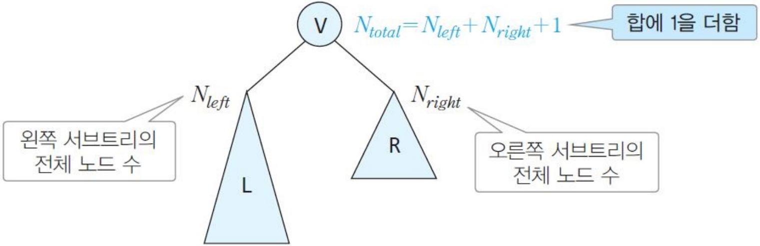
Lab: 이진 트리의 중첩된 괄호 표현



(A (B (D) (E)) (C (F)))

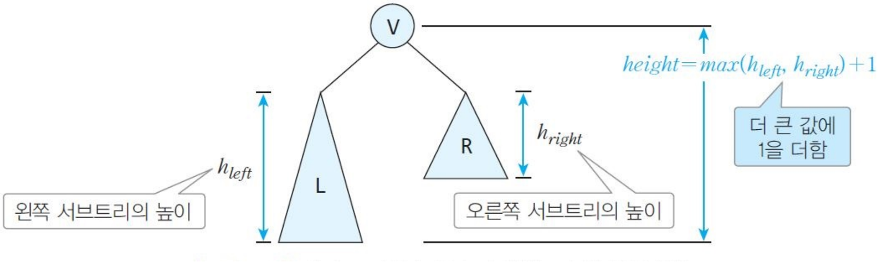
```c
// 중첩된 괄호 표현을 위한 전위순회 함수
void preorder2(TNode* n){
    if (n != NULL) {
        printf("( ");
        printf("%c ", n->data);
        preorder2(n->left);
        preorder2(n->right);
        printf(" )");
    }
}
```

이진 트리의 노드 개수




```
count_node(n)
    if (n == NULL):
        return 0
    n_L <- count_node(n.left)
    n_R <- count_node(n.right)
    return(n_L + n_R)
```

이진 트리의 높이

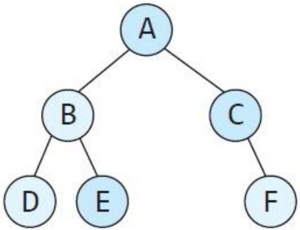



```
calc_height(n)
    if n == NULL:
        return 0
    h_L <- calc_height(n.left)
    h_R <- calc_height(n.right)
    return 1 + max(h_L, h_R)
```

이진 트리의 대칭 트리


```
reverse(n)
    if n != NULL:
      n.left <-> n.right // 좌우 서브트리의 루트를 교환
      reverse(n.left)
      reverse(n.right)
```

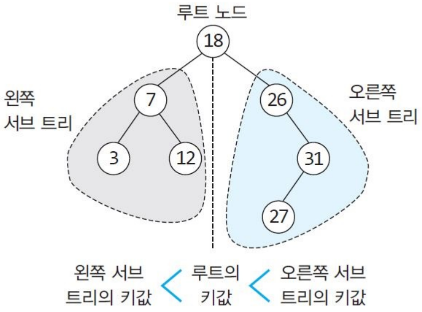
이진 트리에서 노드의 레벨 구하기



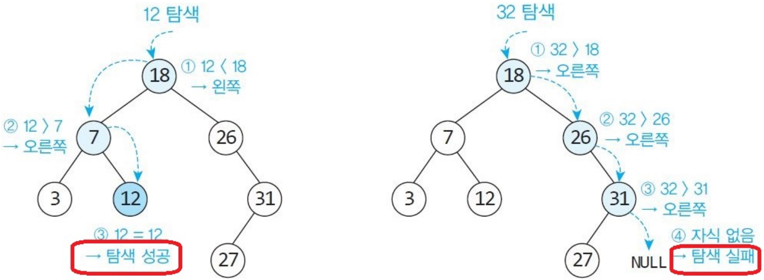
노드 A의 레벨 → 1
노드 C의 레벨 → 2
노드 E의 레벨 → 3

```
calc_level(n, key, level)
    if n == NULL: return 0
    if n == key: return level
    lev <- calc_level(n.left, key, level+1)
    if lev > 0: return lev
    else:
      return calc_level(n.right, key, level+1)
```

- 이진 탐색 트리 (BST, Binary Search Tree): 효율적인 탐색을 위한 이진 트리 기반의 자료구조
- 이진 탐색 트리 정의
  1. 모든 노드는 유일한 키 (key) 를 가짐
  2. 왼쪽 서브트리의 모든 키 (keys) < 루트의 키 (key)
  3. 오른쪽 서브트리의 모든 키 (keys) > 루트의 키 (key)
  4. 왼쪽, 오른쪽 서브 트리는 각각 이진 탐색 트리임



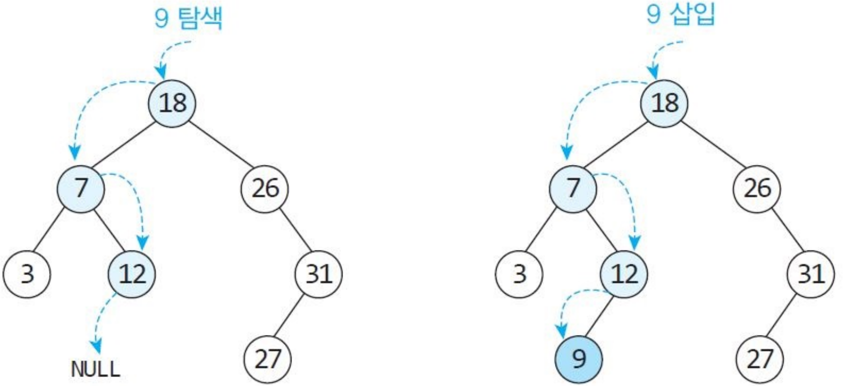
이진 탐색 트리: 탐색 연산



- 재귀호출 방식

```
search(root, key)
    if root == NULL: return NULL
    if KEY(root) == key: return root
    else if KEY(root) > key: //왼쪽 서브트리 탐색
      return search(root.left, key)
    else: //오른쪽 서브트리 탐색
      return search(root.right, key)
```

이진 탐색 트리: 삽입 연산



(a) 탐색을 먼저 수행　　(b) 탐색이 실패한 위치에 노드 삽입

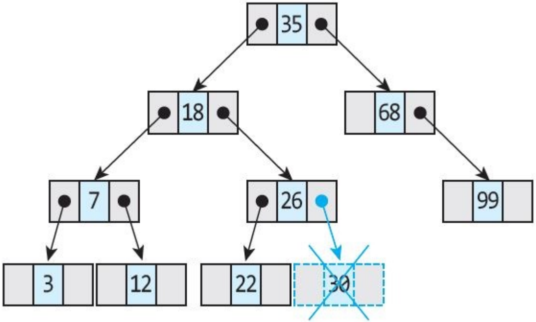
이진 탐색 트리: 삽입 연산 (알고리즘)

- 삽입 연산 역시 재귀호출 방식을 따름

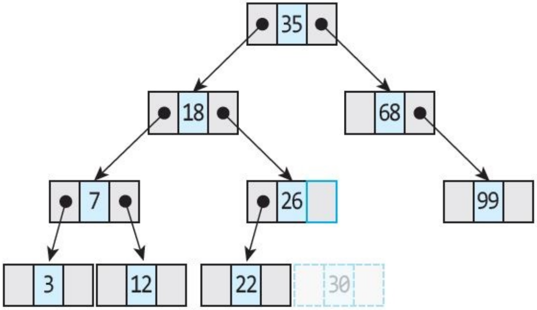
```
insert(root, n)
    if KEY(n) < KEY(root) : //왼쪽 서브트리에 삽입
      if root.left == NULL: root.left <- n
      else : insert(root.left, n)

    else if KEY(n) > KEY(root) : //오른쪽 서브트리에 삽입
      if root.right == NULL: root.left <- n
      else : insert(root.right, n)
    else:
      return delete_node(n)
```

- case 1: 터미널 노드의 삭제



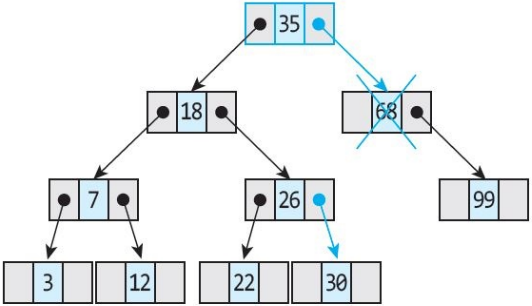
(a) 단말 노드 30의 삭제



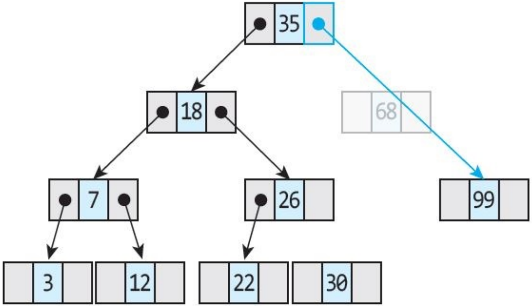
(b) 부모 노드(26)의 링크가 변경됨

이진 탐색 트리: 삭제 연산

- case 2: 자식이 하나인 경우

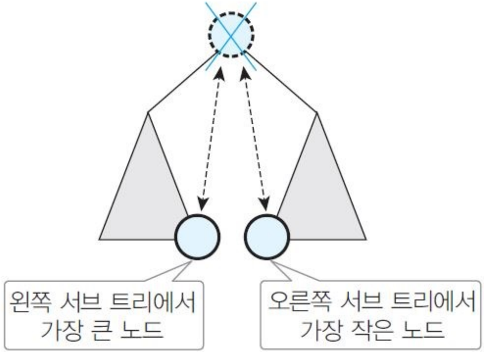


(a) 유일한 자식을 갖는 노드 68 삭제

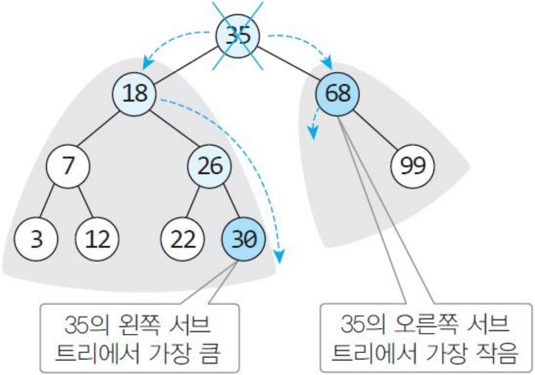


(b) 부모 노드(35)의 링크가 변경됨

- case 3: 자식이 두 개인 경우

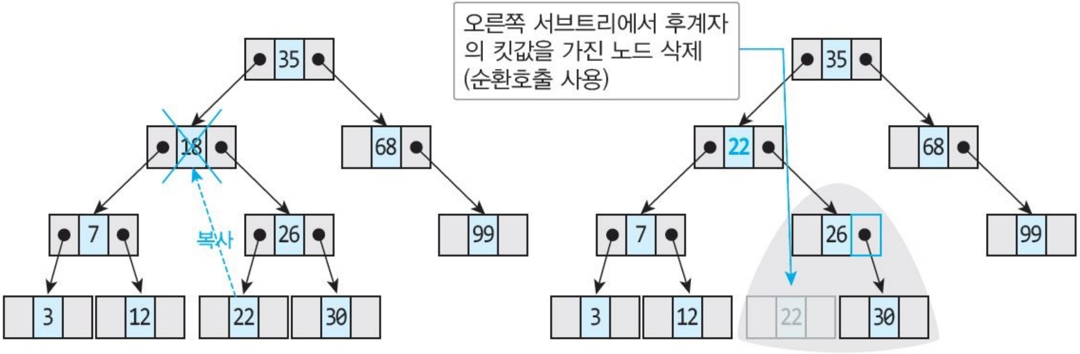


(a) 가능한 후계자 노드



(b) 35의 가능한 후계자 노드

- case 3: 자식이 두 개인 경우



(a) 양쪽 자식을 모두 갖는 노드 18의 삭제

(b) 후계자의 데이터를 복사하고, 실제로는 후계자를 삭제

이진 탐색 트리: 삭제 연산 (알고리즘)

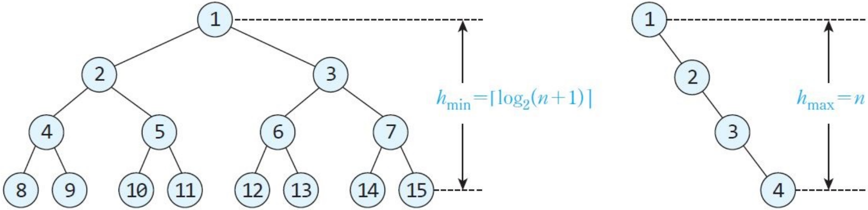
```
delete(root, key)
  n <- search(root, key) //삭제할 노드
  if n = NULL: return root
  p <- n.parent  //n 의 부모노드
  // case 1: 자식이 없음 (터미널 노드)
  if n.left = NULL and n.right = NULL:
    if p = NULL: root <- NULL
    else if p.left = n:
      p.left <- NULL
    else: p.right <- NULL
```

```
// case 2: 자식이 하나
else if n.left = NULL or n.right = NULL:
  ch <- n.left or n.right
  if p = NULL : root <- ch
  else if p.left = n:
    p.left <- ch
  else: p.right <- ch
// case 3: 자식이 둘
else :
  ch <- n.right
  while ch.left != NULL:
    ch <- ch.left
  n.data <- ch.data
  n.right <- delete(n.right, KEY(ch))

return root
```

이진 탐색 트리의 성능

- 시간 복잡도: 트리의 높이 h 에 비례, O(h)

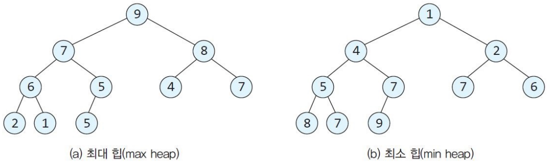


(a) 포화 이진 트리이면 연산의 효율이 좋음

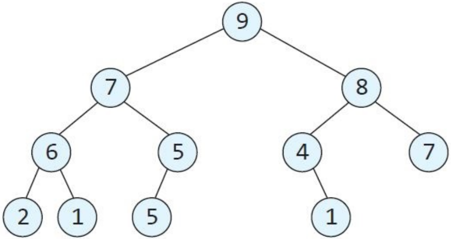
(b) 완전 경사 트리이면 연산의 효율이 좋지 않음

- 균형화 (balancing) 가 필요
  1. AVL 트리 (Adelson-Velskii and Landis, 10 장)
  2. B-tree: More efficient than AVL-tree

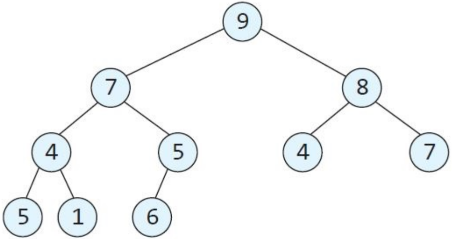
- **완전 이진 트리 (complete binary tree):** 높이가 h 일 때 레벨 $h-1$ 까지는 모두 차 있고 레벨 h 에서는 왼쪽 노드부터 차례로 차 있는 이진 트리
- 힙 (heap) 트리: 키 (key) 값이 가장 큰 (작은) 노드를 빨리 찾을 수 있도록 설계
  1. 완전 이진 트리
  2. 루트의 키 (key) 가 자식의 키 (key) 보다 크거나 같음 (또는 작거나 같음)
  3. 왼쪽, 오른쪽 서브트리는 각각 heap 트리
- 힙 (heap) 의 종류
  - 최대 힙 (max heap): 부모의 키 (key) 값 $\geq$ 자식의 키 (key) 값
  - 최소 힙 (min heap): 부모의 키 (key) 값 $\leq$ 자식의 키 (key) 값



- 힙 트리가 아닌 예
  - 힙의 구조 (완전 이진 트리) 와 크기 조건 (최대 또는 최소 힙) 을 만족해야 함

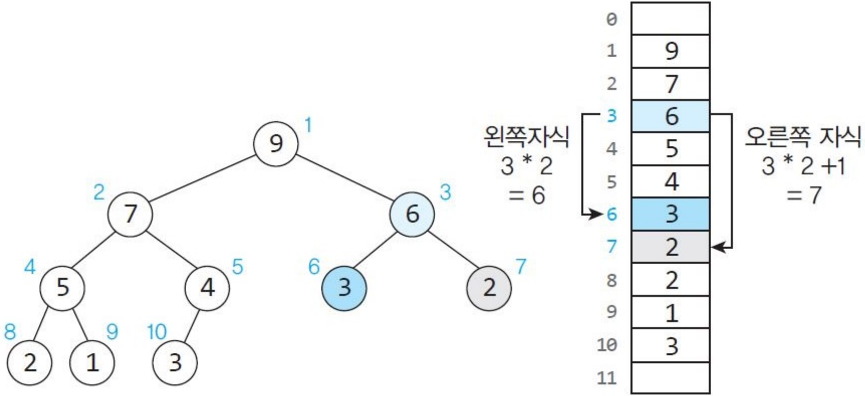


(a) 완전 이진 트리가 아니므로 힙이 아님



(b) 크기 조건을 만족하지 않는 노드가 있으므로 힙이 아님

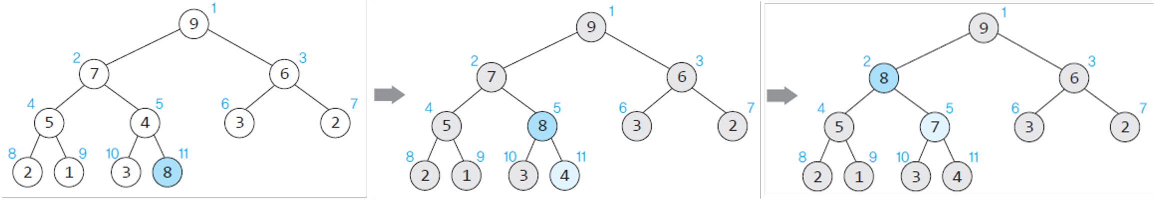
- 힙 트리의 배열 표현



- k 의 부모 인덱스: PARENT(k) = k/2
- k 의 왼쪽 자식 인덱스: LEFT(k) = k/2
- k 의 오른쪽 자식 인덱스: PARENT(k) = k*2 + 1

힙의 삽입 연산

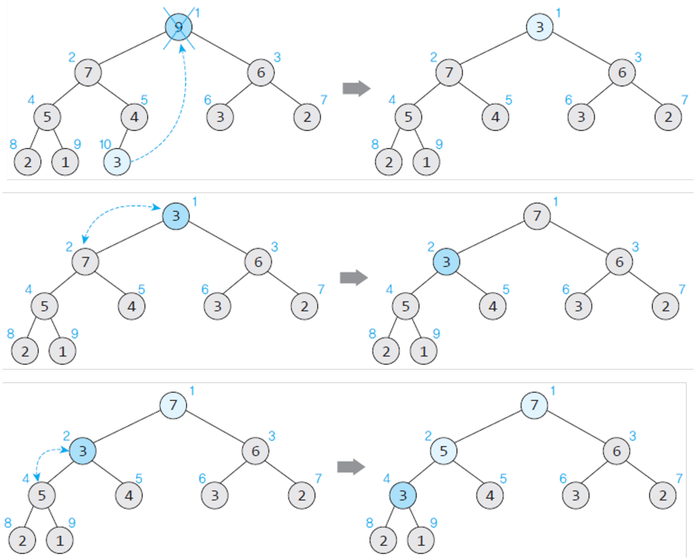
- 마지막 노드로 삽입: Up-heap



```
heap_push(n)
    heap_size <- heap_size + 1
    i <- heap_size
    A[i] <- node
    while i != 1 :
      if KEY(i) > KEY (PARENT(i)) :
        A[i] <-> A[PARENT(i)]
        i <- PARENT(i)
      else : break
```

힙의 삭제 연산

- 루트 삽입 –> 마지막 노드를 루트로 –> down-heap



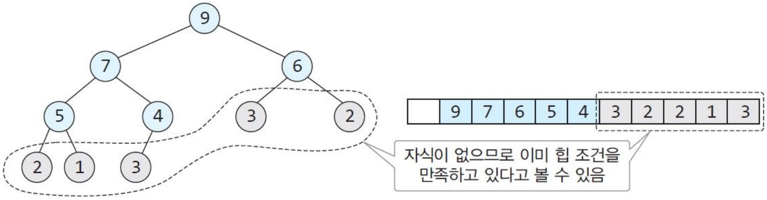
힙의 삭제 연산 - 알고리즘

```
heap_pop()
    root <- A[1]
    A[1] <- A[heap_size]
    heap_size <- heap_size - 1
    i <- 1
    while LEFT(i) <= heap_size :
      if LEFT(i) < heap_size & KEY(LEFT(i)) > KEY(RIGHT(i)) :
        child <- LEFT(i)
      else : child <- RIGHT(i)
      if KEY(i) > KEY(child) : break
      else :
        A[i] <-> A[child]
        i <- child
    return root
```

7 장/MaxHeap.c

Lab: 배열이 최대 힙인지 검사하기

```c
int a[] = { 0, 9, 7, 6, 5, 4, 3, 2, 2, 1, 3 }; // 최대힙 맞음
int b[] = { 0, 9, 7, 6, 5, 3, 4, 2, 2, 3, 1 }; // 최대힙 아님
```



(a) 최대 힙　　(b) 최대 힙의 배열 표현

```c
int is_max_heap(HNode arr[], int len){
    for (int i = 1; i <= len / 2; i++)
        if (arr[i] < arr[LEFT(i)] || arr[i] < arr[RIGHT(i)])
            return 0;   // 크기 조건이 맞지 않음 -> 최대 힙이 아님
    return 1;           // 모든 노드에서 크기 조건 만족 -> 최대힙 맞음
}
```

연습문제