자료구조론

9장 트리(tree)

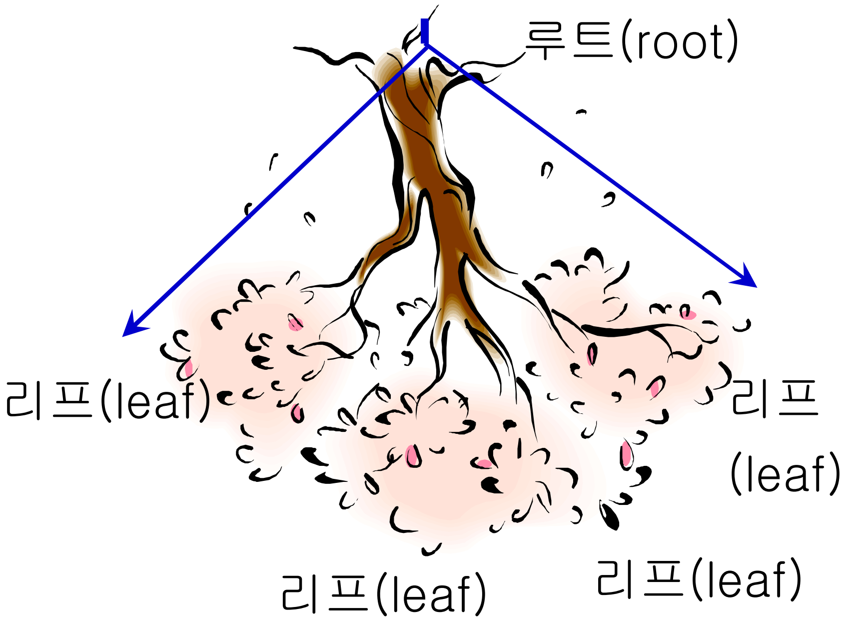
이 장에서 다룰 내용

- 트리(tree)
- 이진 트리(binary tree)
- 이진 트리의 구현
- 이진 트리의 순회
- 이진 탐색 트리
- 힙(heap)

❑ 트리

❖ 트리(tree)

- 원소들 간에 1:多 관계를 가지는 비선형 자료구조
- 원소들 간에 계층관계를 가지는 계층형 자료구조
- 상위 원소에서 하위 원소로 내려가면서 확장되는 트리(나무) 모양의 구조

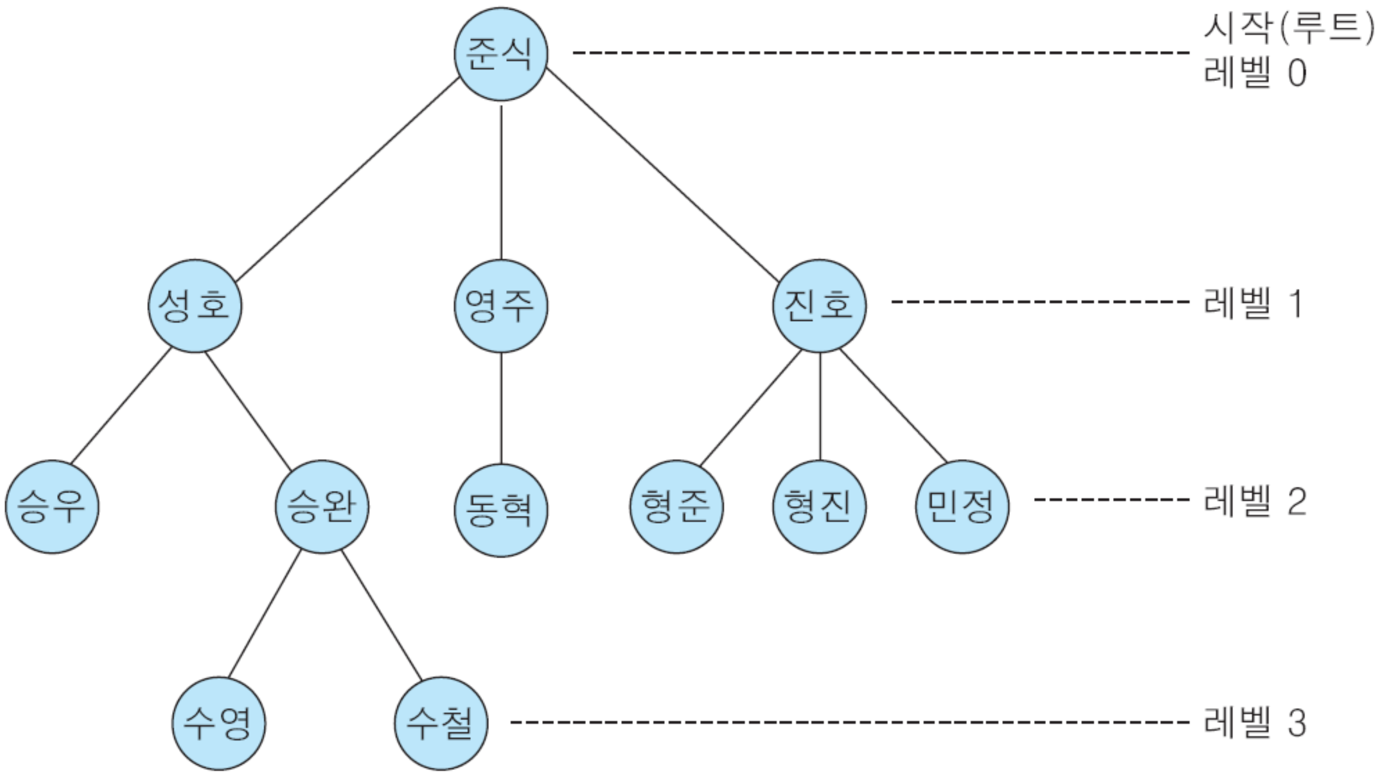


하나의 **뿌리(root)**에서 가지가 뻗어나가면서 확장되어 끝에 **잎(leaf)**이 달리는 구조

❑ 트리

❖ 트리 자료구조의 예 – 가계도

- 가계도의 자료 : 가족 구성원
- 자료를 연결하는 선 : 부모(parent)-자식(child) 관계 표현



❑ 트리

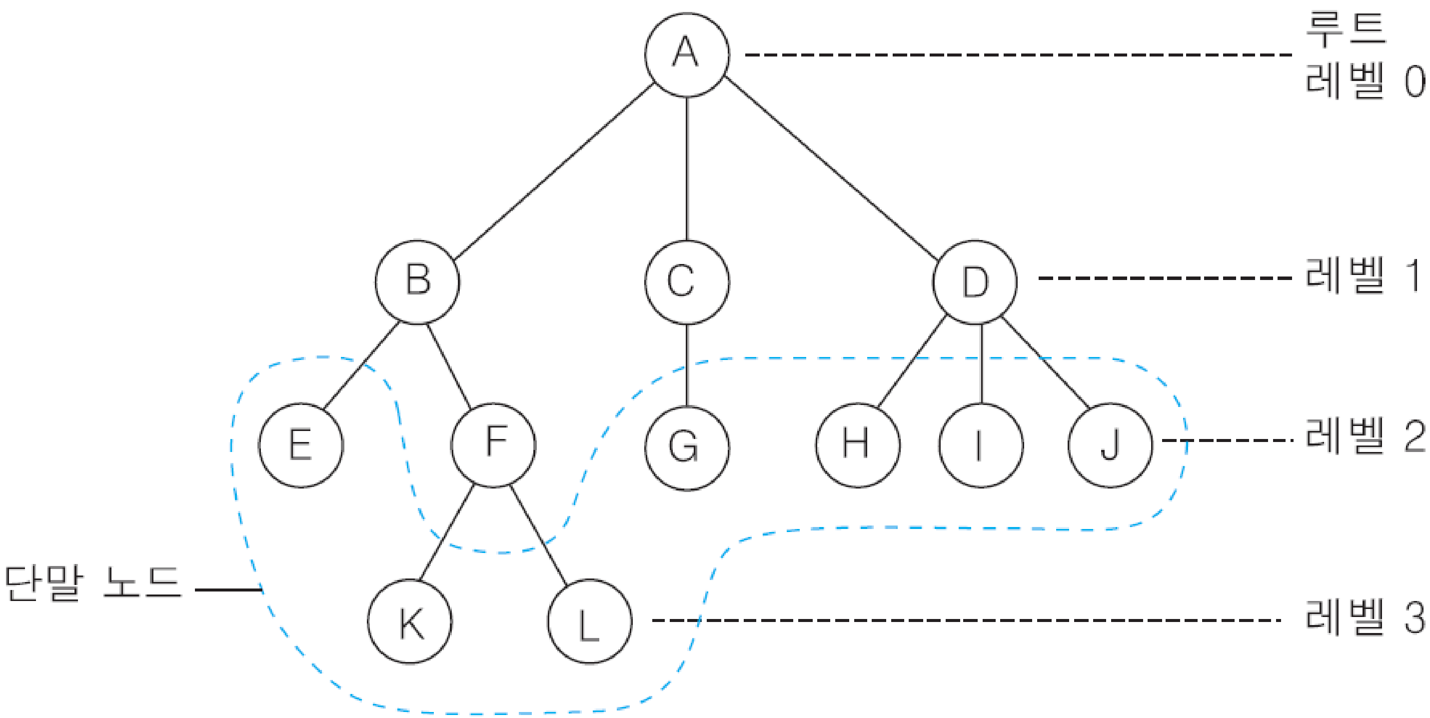
- 준식의 자식(child) – 성호, 영주, 진호
- 성호, 영주, 진호의 부모(parent) – 준식
- 성호, 영주, 진호는 형제관계(sibling)

- 수영의 조상 – 승완, 성호, 준식
- 성호의 자손 – 승우, 승완, 수영, 수철

- 선을 따라 내려가면서 다음 세대로 확장
- 가족 구성원 누구든지 자기의 가족을 데리고 분가하여 독립된 가계를 이룰 수 있다.

❑ 트리

❖ 트리 용어



❑ 트리

- 노드(node)
  - 트리의 원소
  - 트리 A의 노드는 총 12개 – A, B, C, D, E, F, G, H, I, J, K, L
- 루트(root) 노드
  - 트리의 시작 노드
  - 트리 A의 루트노드 – A
- 단말(terminal) 노드 – 차수가 0인 노드. 자식이 없는 노드.
  - 리프(leaf) 노드, 잎 노드라고도 함
  - 트리 A의 단말 노드는 총 7개 – E, K, L, G, H, I, J
- 간선(edge)
  - 부모(parent) 노드와 자식(child) 노드를 연결하는 선
  - 트리 A의 간선은 총 11개
- 서브 트리(subtree)
  - 부모 노드와 연결된 간선을 끊었을 때 생성되는 트리
  - 각 노드는 자식 노드의 개수 만큼 서브 트리를 가진다.

❑ 트리

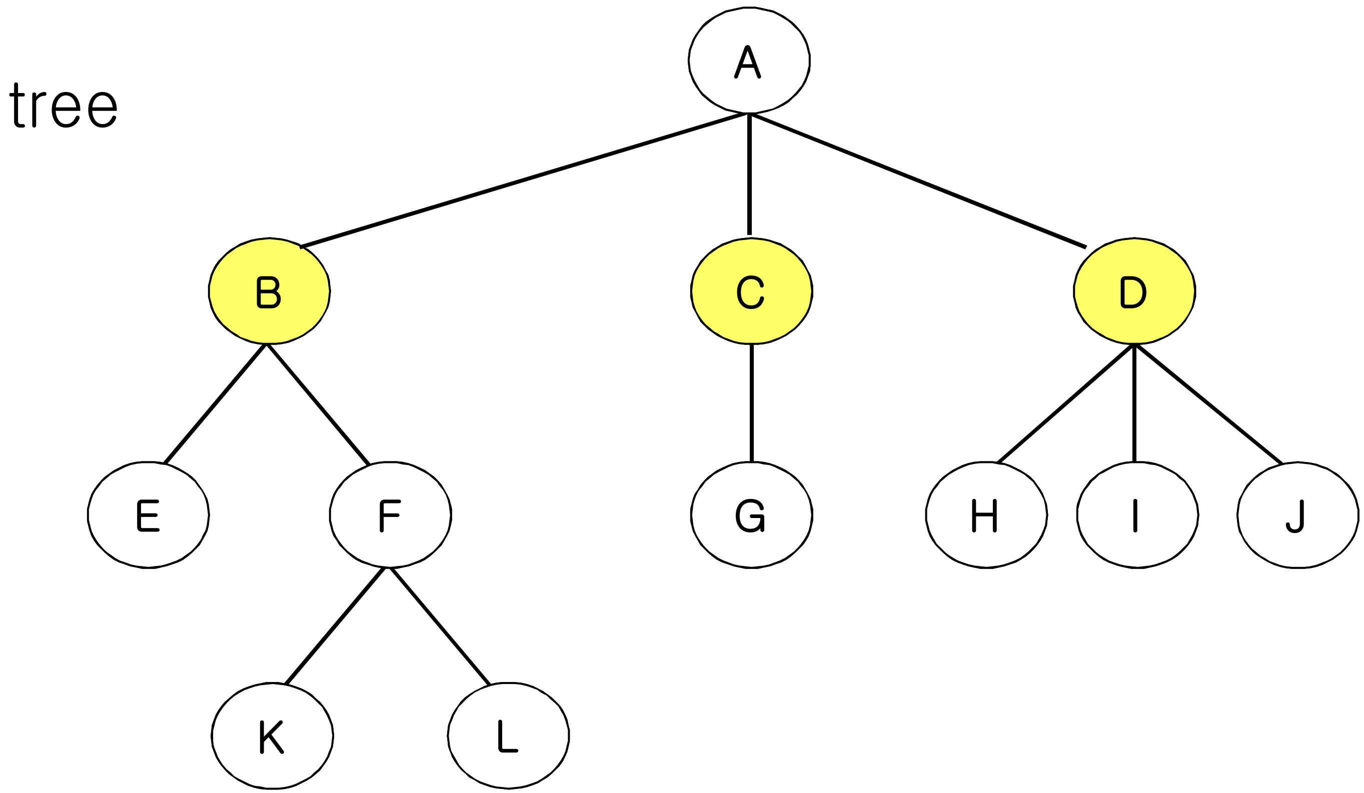
- 조상(ancestor) 노드
  - 간선을 따라 루트 노드까지 이르는 경로에 있는 모든 노드들
  - K의 조상 노드 : F, B, A
- 자손(descendant) 노드
  - 서브 트리에 있는 하위 레벨의 노드들
  - B의 자손 노드 – E, F, K, L
- 형제(sibling) 노드
  - 같은 부모 노드의 자식 노드들
  - H, I, J 는 형제 노드

❑ 트리

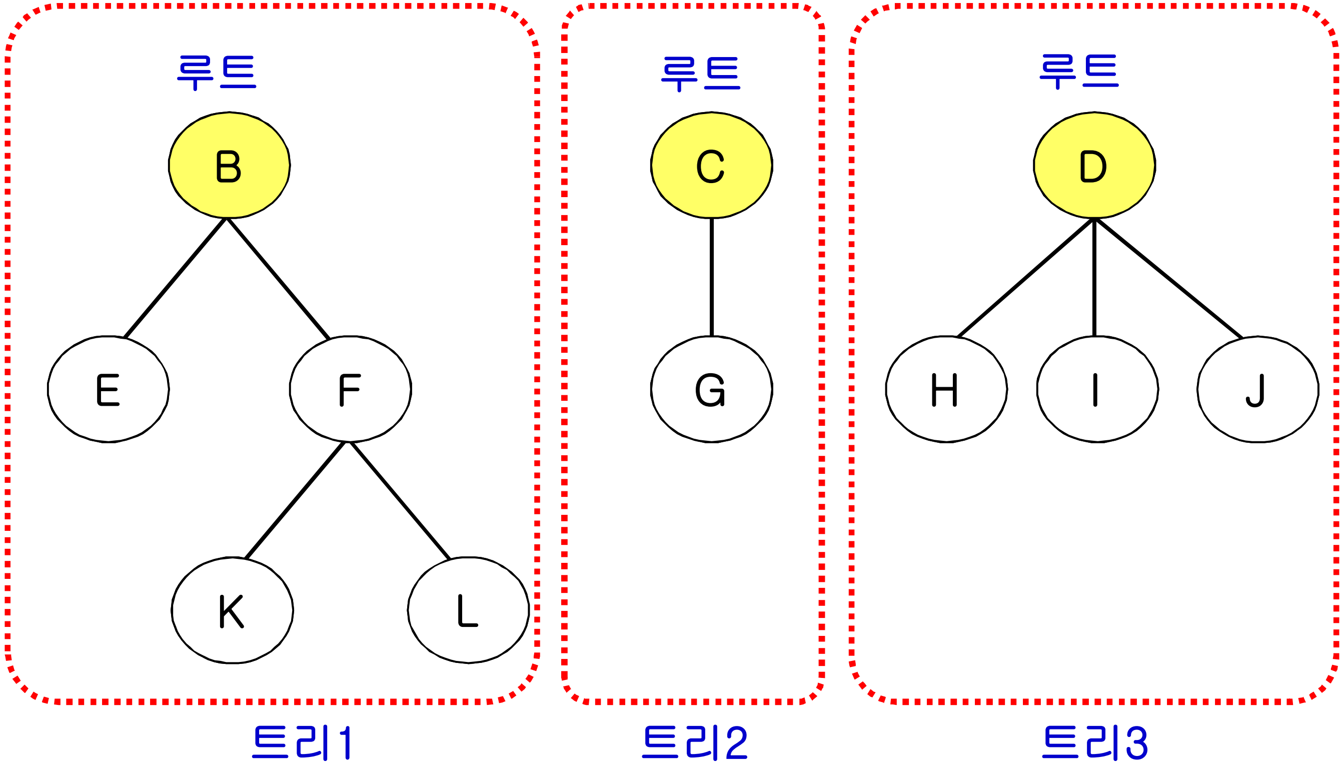
- **노드의 차수(degree)**
  - 노드에 연결된 자식 노드의 수
  - A의 차수=3, B의 차수=2, C의 차수=1
- **트리의 차수(degree)**
  - 트리에 있는 노드의 차수 중에서 가장 큰 값
  - 트리 A의 차수=3
- **노드의 높이(height)**
  - 루트에서 노드에 이르는 간선의 수
  - **노드의 레벨(level)**이라고도 함
  - B의 높이=1, F의 높이=2
- **트리의 높이(height)- 깊이(depth)**
  - 트리에 있는 노드의 높이 중에서 가장 큰 값. 최대 레벨
  - 트리 A의 높이=3

트리

- **포리스트(forest)** : 트리들의 집합
  - 트리 A에서 노드 A를 제거하면, A의 자식 노드 B, C, D를 루트로 하는 트리들이 생기고, 이들의 집합은 포리스트(숲)가 된다.



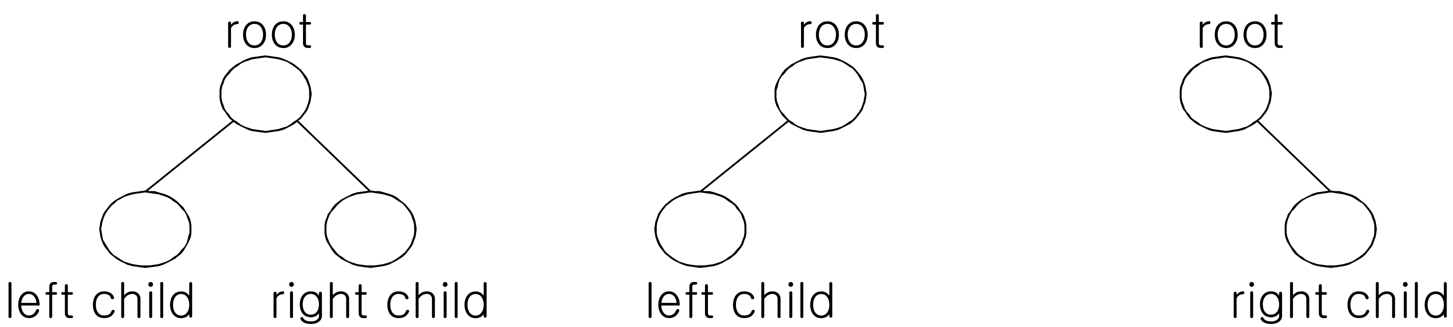
forest



❑ 이진 트리

❖ 이진 트리(binary tree)

- 트리의 모든 노드가 왼쪽 자식 노드와 오른쪽 자식 노드 만을 가지도록 함으로써 트리의 차수가 2 이하가 되도록 제한한 트리

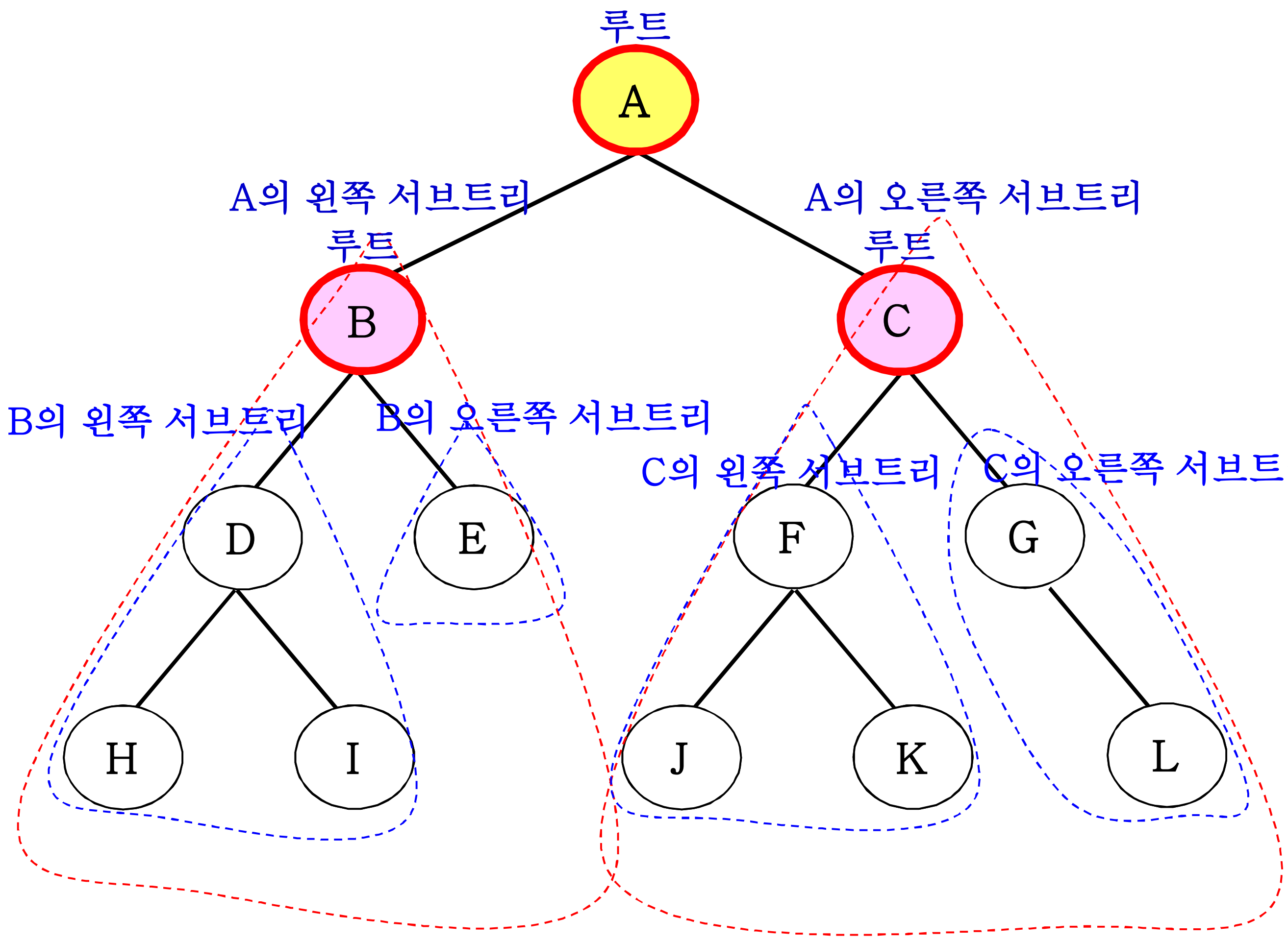


- 왼쪽 자식 노드, 오른쪽 자식 노드가 서로 구분된다.
- 이와 같이 트리 구조를 일정하게 정의하면 트리의 구현과 연산이 단순해진다.

❑ 이진 트리

❖ 이진 트리의 재귀적 구성

- 루트 노드의 왼쪽 자식노드를 루트로 하는 서브트리도 이진 트리
- 루트 노드의 오른쪽 자식노드를 루트로 하는 서브트리도 이진 트리



❑ 이진 트리

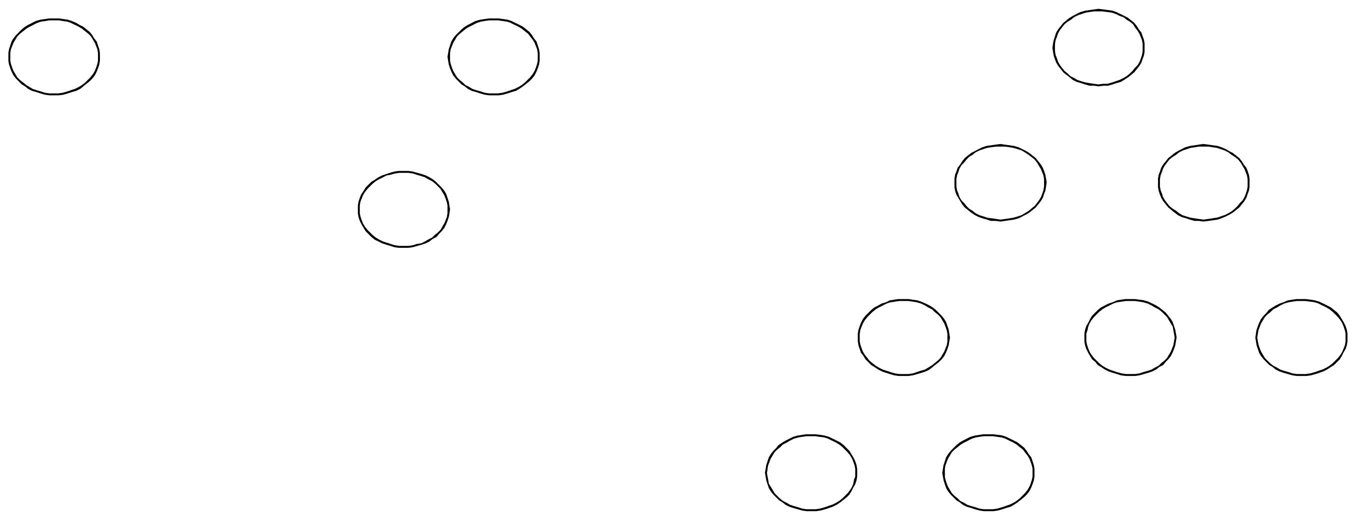
❖ 이진 트리에 대한 추상 자료형

```
ADT BinaryTree
데이터 : 공백이거나
         루트 노드, 왼쪽 서브 트리, 오른쪽 서브 트리로 구성된 노드들의 유한 집합
연산 : bt, bt1, bt2∈BinaryTree; item∈Element;
    createBT() ::= create an empty binary tree;
        // 공백 이진 트리를 생성하는 연산
    isEmpty(bt) ::= if (bt is empty) then return true else return false;
        // 이진 트리가 공백인지 아닌지를 확인하는 연산
    makeBT(bt1, item, bt2) ::= return {item을 루트로 하고 bt1을 왼쪽 서브 트리,
                               bt2를 오른쪽 서브 트리로 하는 이진 트리}
        // 두개의 이진 서브 트리를 연결하여 하나의 이진 트리를 만드는 연산
    leftSubtree(bt) ::= if (isEmpty(bt)) then return null
                        else return left subtree of bt;
        // 이진 트리의 왼쪽 서브 트리를 구하는 연산
    rightSubtree(bt) ::= if (isEmpty(bt)) then return null
                         else return right subtree of bt;
        // 이진 트리의 오른쪽 서브 트리를 구하는 연산
    data(bt) ::= if (isEmpty(bt)) then return null
                 else return the item in the root node of bt;
        // 이진 트리에서 루트 노드의 데이터(item)를 구하는 연산
End BinaryTree
```

이진 트리

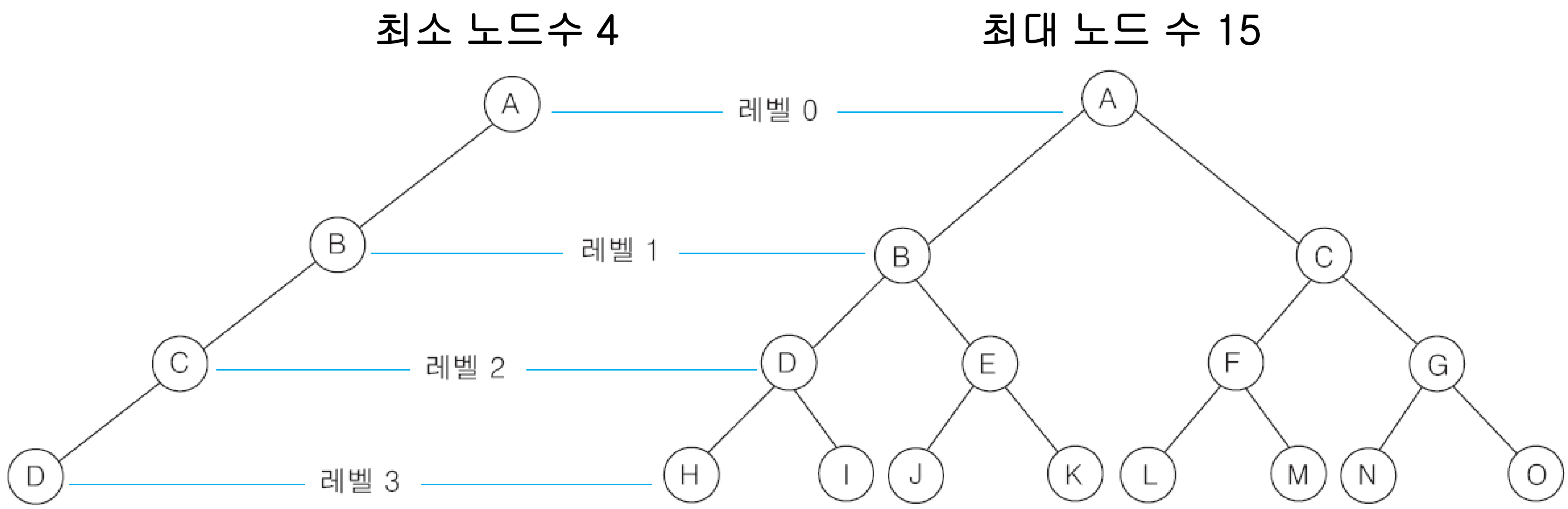
❖ 이진 트리의 특성

- 노드 수가 n 인 이진 트리의 간선 수는 (n-1)
  - 단, 공백 이진 트리(n = 0)는 제외
  - 루트를 제외한 (n-1)개의 노드가 부모 노드와 연결되는 한 개의 간선을 가지기 때문



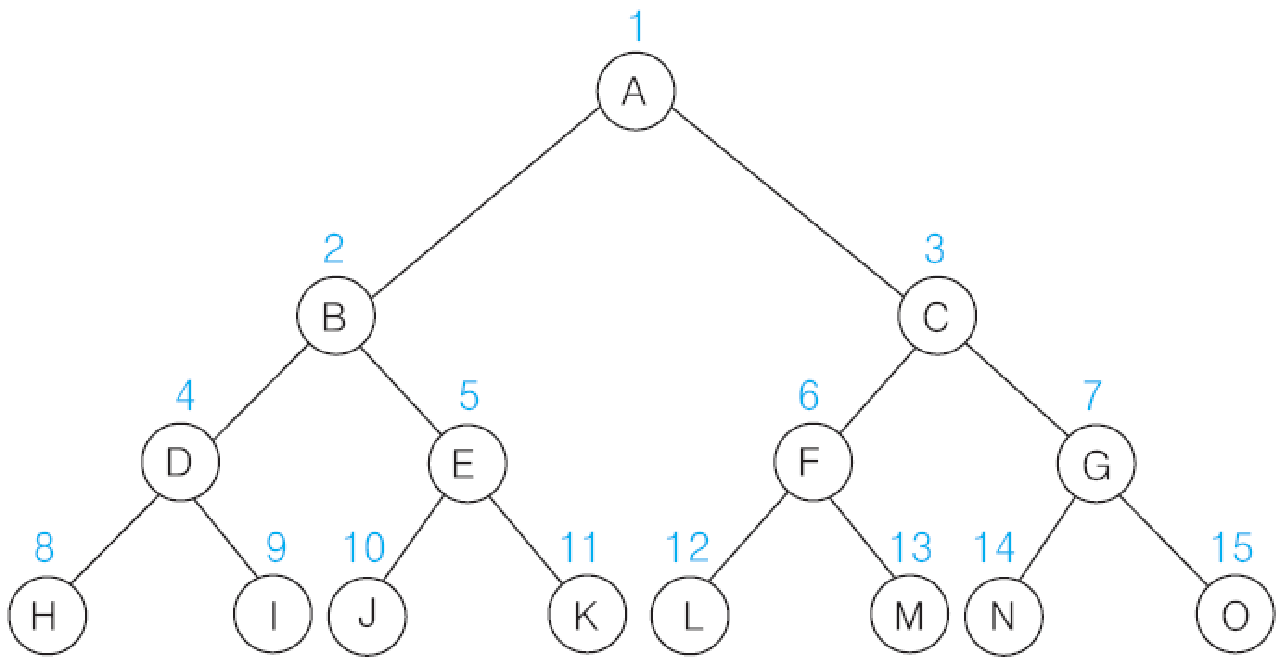
❖ 이진 트리의 특성

- 높이가 h인 이진 트리가 가질 수 있는 노드 수는
  최소 $(h+1)$ ~ 최대 $(2^{h+1}-1)$
  - 한 레벨에 최소한 한 개의 노드는 있어야 함. 따라서
    높이 h인 이진 트리의 최소 노드수 = $h+1$
  - 자식노드가 최대 2개이므로 레벨 i의 노드수는 최대 2^i. 따라서
    높이 h인 이진 트리의 최대 노드수 = $2^0+2^1+\ldots+2^h = 2^{h+1}-1$
  - 예) 높이 h = 3 인 두개의 트리를 비교해보자.



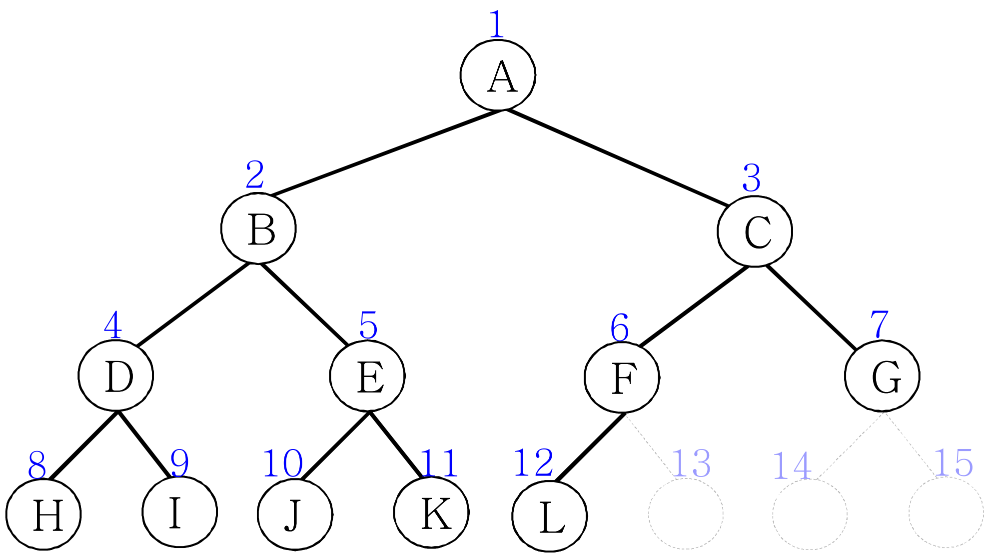
❖ 이진 트리의 종류

- **포화 이진 트리(full binary tree)**
  - 모든 레벨에 노드가 꽉 차서 해당 높이에서 가질 수 있는 최대 개수의 노드를 갖는 이진 트리
  - 즉, **높이가 h**일 때, **($2^{h+1}-1$)**개의 노드를 가진 이진 트리
  - 예) 높이가 3인 포화 이진 트리



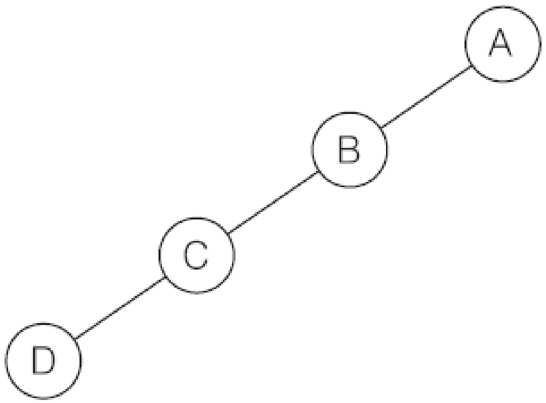
❑ 이진 트리

- 완전 이진 트리(complete binary tree)
  - 트리의 높이가 h일 때, 레벨 0부터 레벨 h-1까지는 포화 상태이고, 마지막 레벨 h는 왼쪽부터 차례로 노드가 채워진 이진 트리
  - 포화 이진 트리도 일종의 완전 이진 트리임
  - 예) 노드 수가 12인 완전 이진 트리

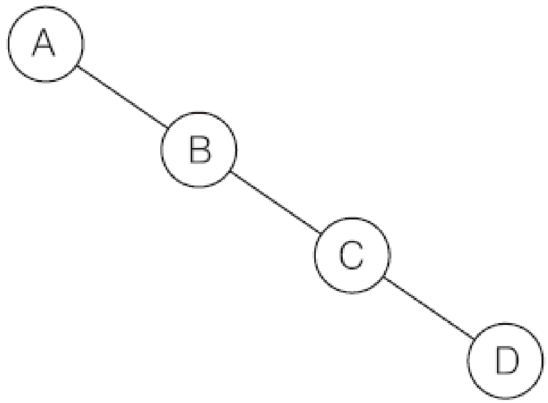


❑ 이진 트리

- **편향 이진 트리(skewed binary tree)**
  - 높이 h에 대한 최소 개수의 노드를 가지면서 한쪽 방향의 자식 노드만을 가진 이진 트리
  - 좌편향 이진 트리
    - ➢ 모든 노드가 왼쪽 자식 노드만을 가진 편향 이진 트리
  - 우편향 이진 트리
    - ➢ 모든 노드가 오른쪽 자식 노드만을 가진 편향 이진 트리



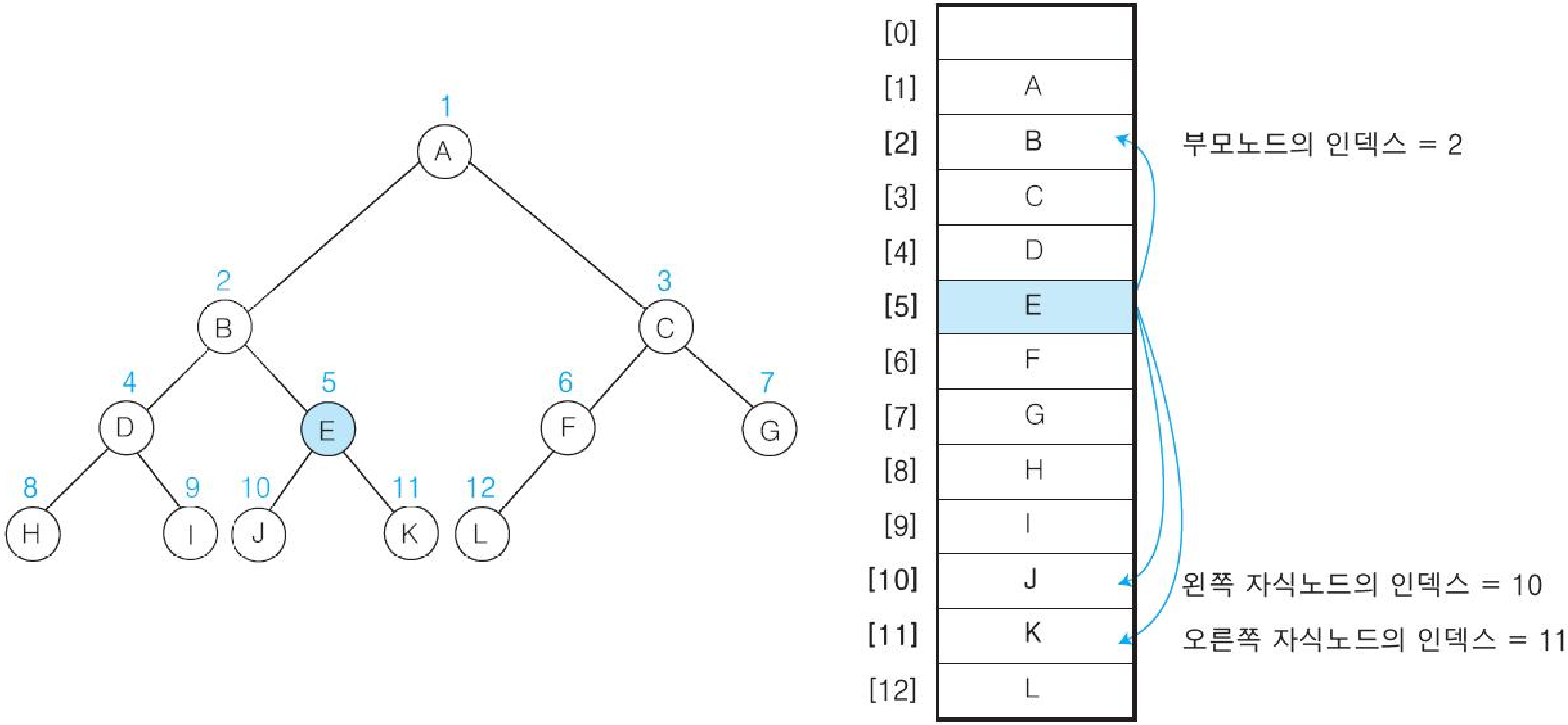
좌편향 이진 트리



우편향 이진 트리

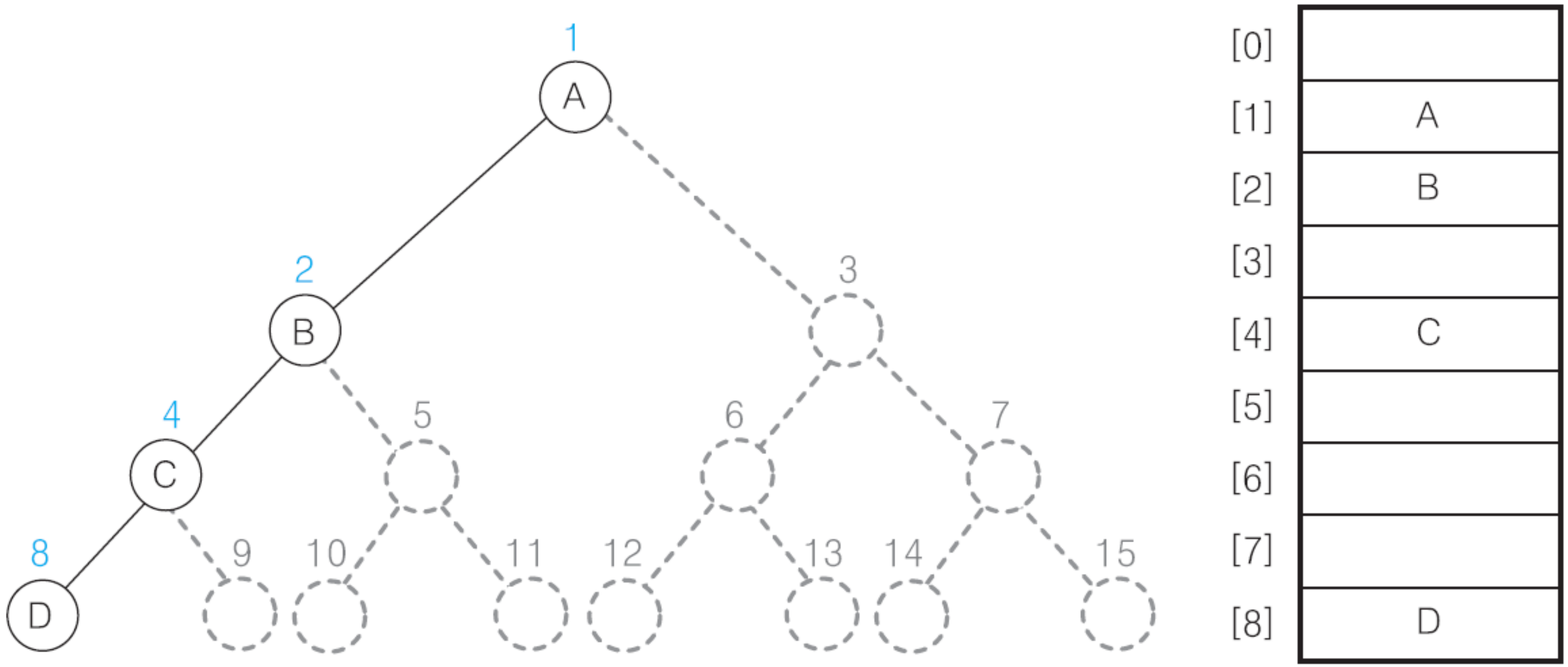
❑ 이진 트리의 구현 - 배열

- 1차원 배열을 이용한 이진 트리의 구현
  - 높이가 h인 포화 이진 트리의 노드번호를 배열의 인덱스로 사용
    - 교재에서는 인덱스 0은 비워두고, 인덱스 1에 루트를 저장
  - 예) 완전 이진 트리의 1차원 배열 표현



❑ 이진 트리의 구현 – 배열

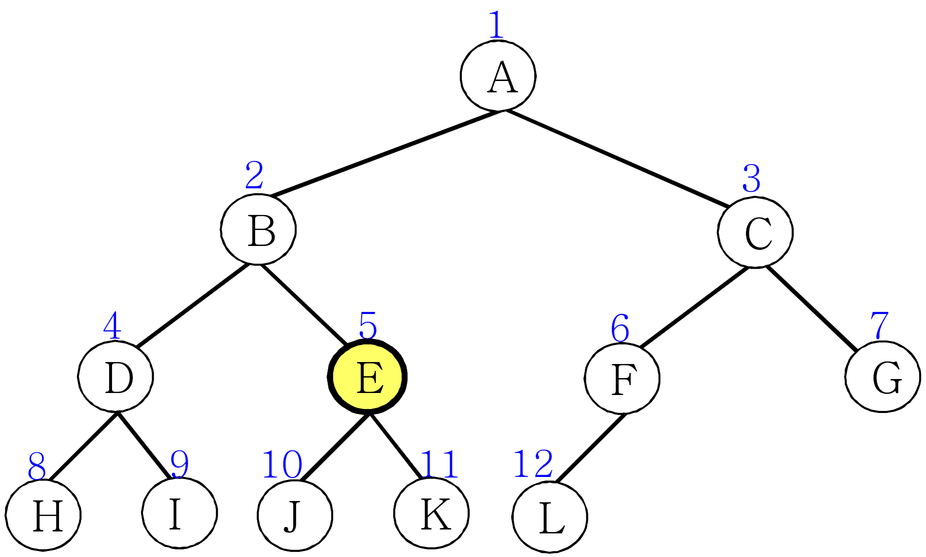
- 좌편향 이진 트리의 1차원 배열 표현

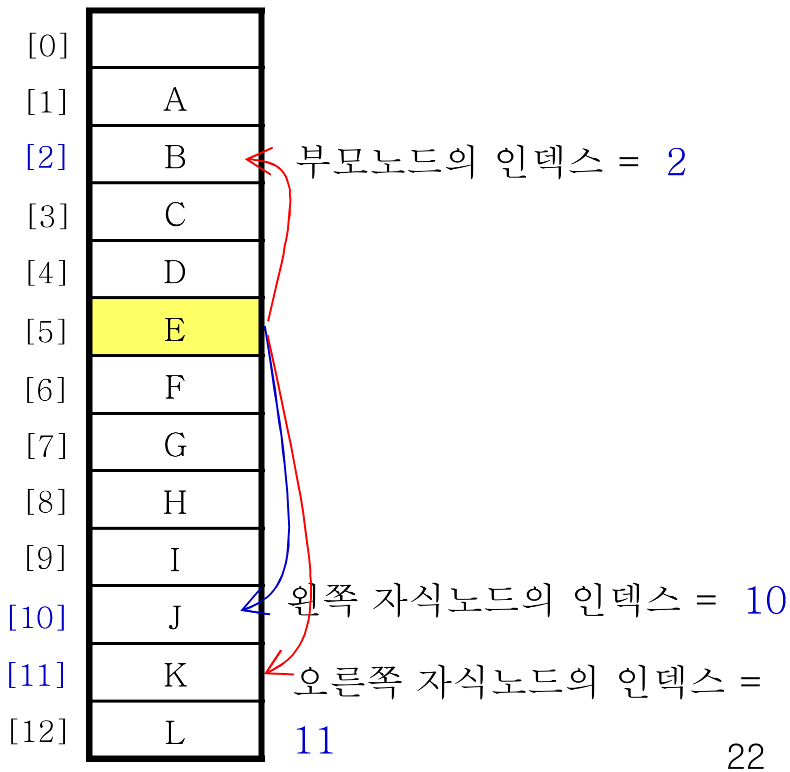


이진트리의 구현

이진 트리의 1차원 배열에서의 인덱스 관계

노드	인덱스	성립 조건
노드 i의 부모 노드	$\lfloor i/2 \rfloor$	$i > 1$
노드 i의 왼쪽 자식 노드	$2 \times i$	$(2 \times i) \leq n$
노드 i의 오른쪽 자식 노드	$(2 \times i) + 1$	$(2 \times i + 1) \leq n$
루트 노드	1	$0 < n$





❑ 이진 트리의 구현 – 배열

- 이진 트리를 1차원 배열로 표현하는 경우 단점
  - 포화 이진 트리 또는 완전 이진 트리가 아닌 경우, 배열 중간 중간에 사용하지 않는 원소들을 비워두어야 하므로 메모리 낭비
    - ➢ 편향 이진 트리인 경우 가장 낭비가 심하다.
  - 트리의 원소 삽입/삭제에 따라 트리 높이가 동적으로 변하는 경우 배열의 크기 변경이 어렵다.
- 장점은?

❑ 이진 트리의 구현 – 연결 자료구조

❖ 연결 자료구조를 이용한 이진 트리의 구현

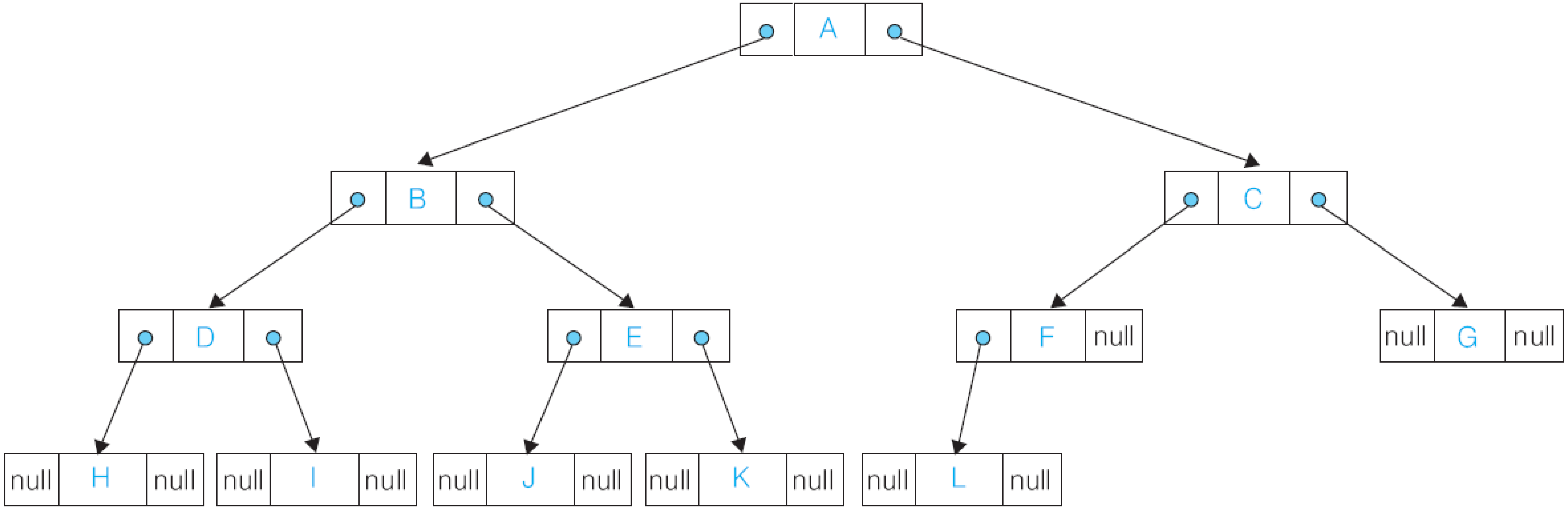
- 이진 트리의 모든 노드는 2개의 자식 노드를 가지므로 다음과 같은 구조의 노드를 사용

- 이진 트리의 노드 구조를 자바 클래스로 정의 (노드에 문자 데이터를 저장하는 경우)

```
class TreeNode{
    char data;
    TreeNode leftChild;
    TreeNode rightChild;
}
```

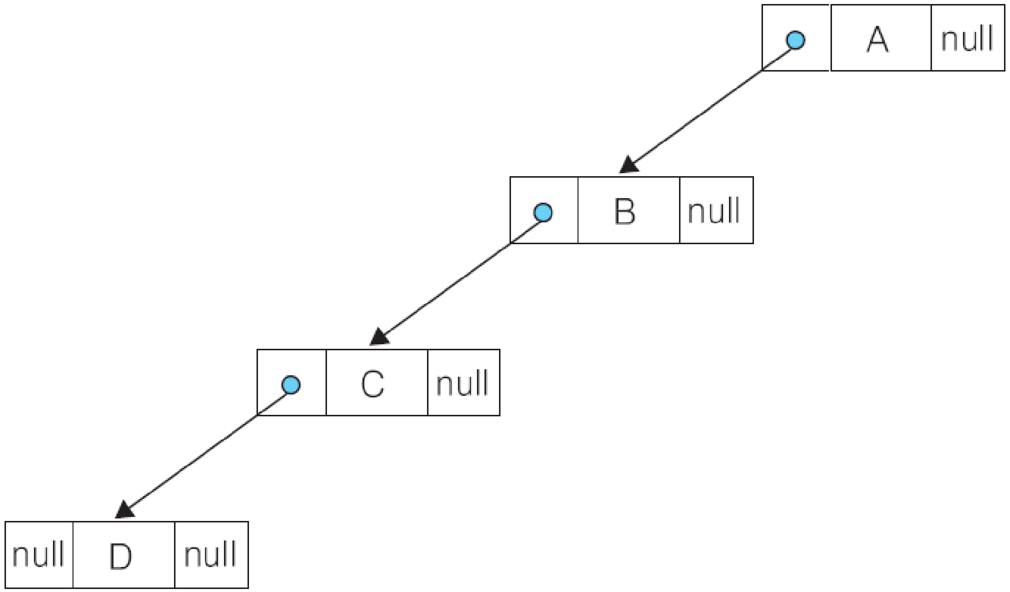
❑ 이진 트리의 구현 - 연결 자료구조

- 완전 이진 트리의 연결 자료구조 표현



❑ 이진 트리의 구현 – 연결 자료구조

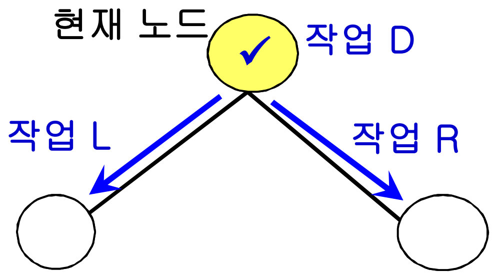
- 좌편향 이진 트리의 연결 자료구조 표현



❑ 이진 트리의 순회

❖ 이진 트리의 순회(traversal)

- 이진 트리의 모든 노드를 한번씩 방문하여 데이터를 처리하는 연산
- 순회를 위해 수행할 수 있는 작업을 다음과 같이 정의하자.
  (1) 현재 노드를 방문하여 데이터를 처리하는 작업 D
  (2) 현재 노드의 왼쪽 서브트리를 순회하는 작업 L
  (3) 현재 노드의 오른쪽 서브트리를 순회하는 작업 R



- 이진 트리가 재귀적으로 정의되므로, 순회도 재귀적으로 이루어짐
- 순회의 종류
  - 전위 순회(preorder) : DLR
  - 중위 순회(inorder) : LDR
  - 후위 순회(postorder) : LRD

❑ 이진 트리의 순회

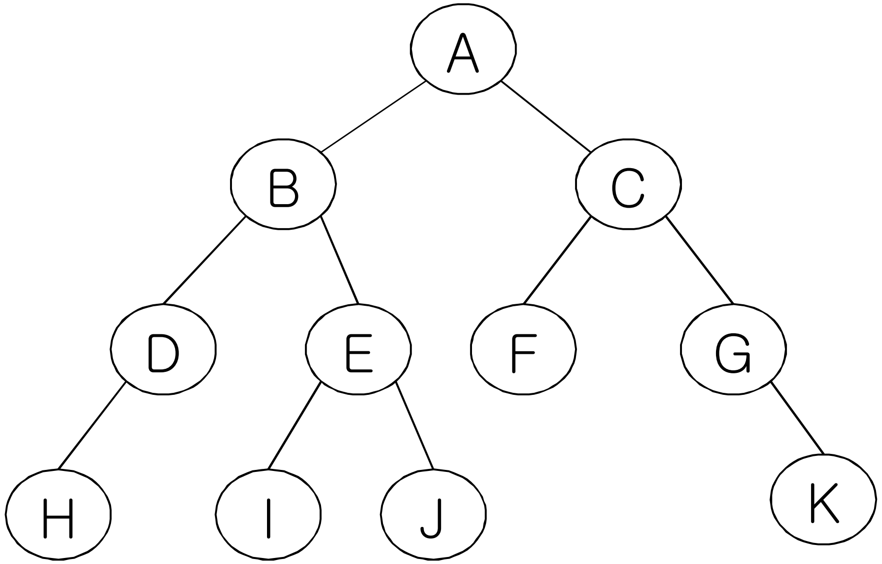
- 전위 순회(preorder traversal)
  - 수행 방법
    ① 현재 노드 n을 방문하여 처리: D
    ② 현재 노드 n의 왼쪽 서브트리를 전위 순회 : L
    ③ 현재 노드 n의 오른쪽 서브트리를 전위 순회 : R

  - 전위 순회 알고리즘

```
preorder(T)
    if (T≠null) then {
        visit T.data;
        preorder(T.leftChild);
        preorder(T.rightChild);
    }
end preorder()
```

❑ 이진 트리의 순회

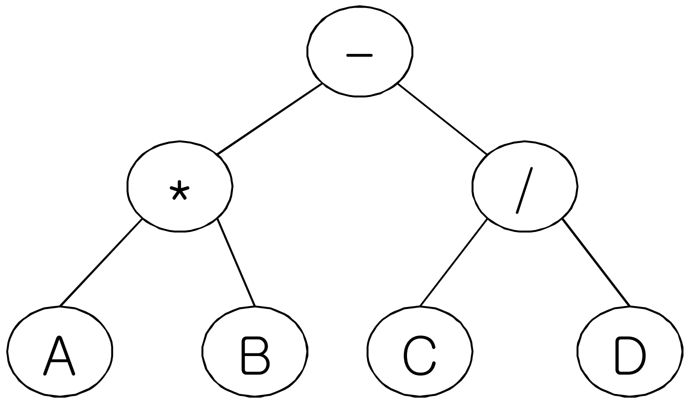
- 전위 순회의 예
  - A-B-D-H-E-I-J-C-F-G-K



❑ 이진 트리의 순회

- 수식 이진 트리의 전위 순회
  - 수식을 이진 트리로 구성한 수식 이진 트리를 전위 순회하면, 수식에 대한 전위 표기식을 구할 수 있다.
  - 예

    A * B – C / D ➔ – * A B / C D



❑ 이진 트리의 순회

❖ 중위 순회(inorder traversal)

- 수행 방법

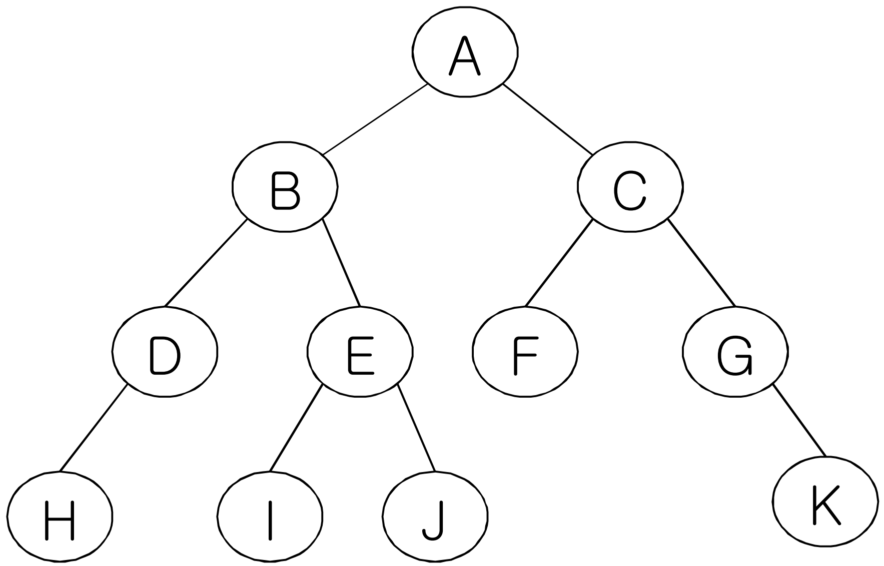
  ① 현재 노드 n의 왼쪽 서브트리를 중위 순회 : L
  ② 현재 노드 n을 방문하여 처리 : D
  ③ 현재 노드 n의 오른쪽 서브트리를 중위 순회 : R

- 중위 순회 알고리즘

```
inorder(T)
    if (T≠null) then {
        inorder(T.leftChild);
        visit T.data;
        inorder(T.rightChild);
    }
end inorder()
```

이진 트리의 순회

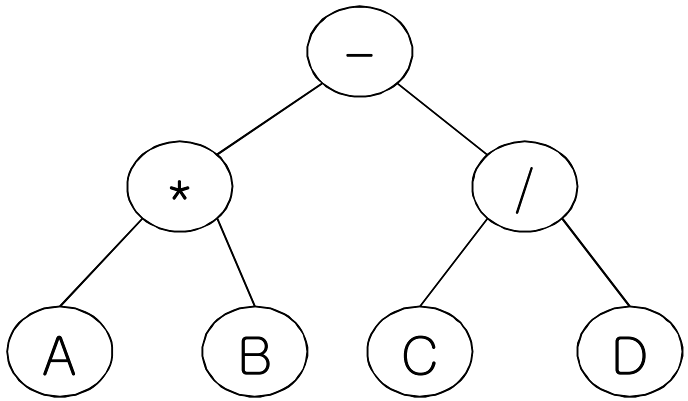
- 중위 순회의 예
  - H-D-B-I-E-J-A-F-C-G-K



❑ 이진 트리의 순회

- 수식 이진 트리의 중위 순회
  - 수식 이진 트리를 중위 순회하면, 수식에 대한 중위 표기식을 구할 수 있다.
  - 예

    A * B – C / D ➔ A * B – C / D



이진 트리의 순회

❖ 후위 순회(postorder traversal)

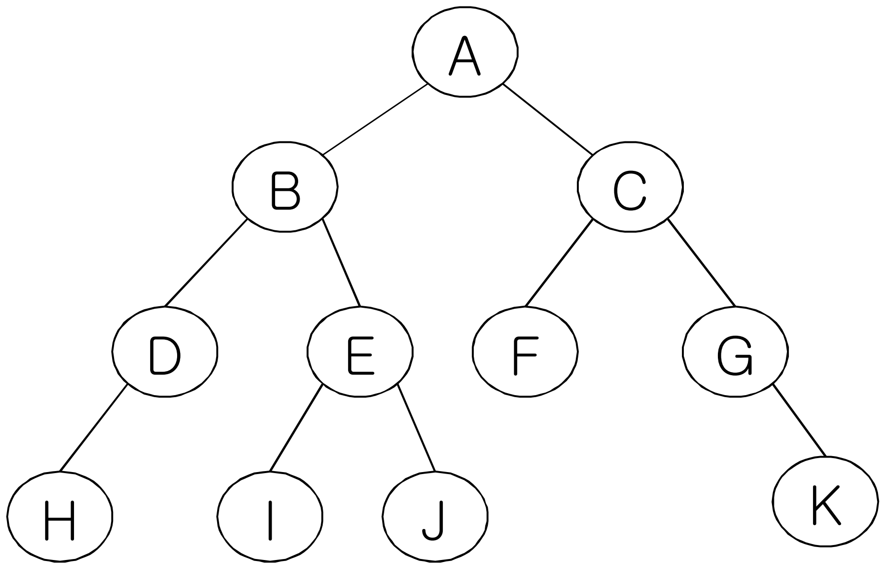
- 수행 방법
  - ① 현재 노드 n의 왼쪽 서브트리를 후위 순회 : L
  - ② 현재 노드 n의 오른쪽 서브트리를 후위 순회 : R
  - ③ 현재 노드 n을 방문하여 처리 : D

- 후위 순회 알고리즘

```
postorder(T)
    if (T≠null) then {
        postorder(T.leftChild);
        postorder(T.rightChild);
        visit T.data;
    }
end postorder()
```

이진 트리의 순회

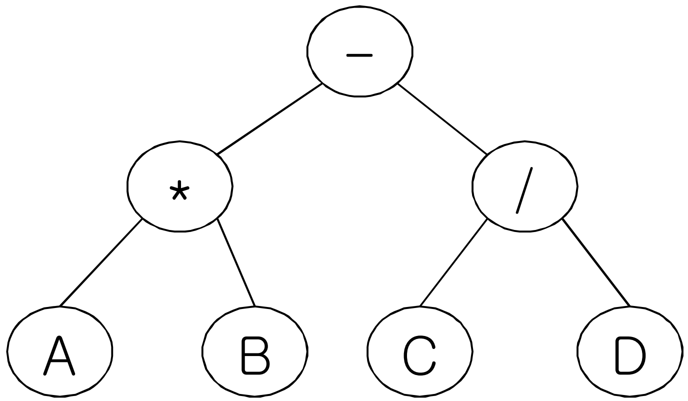
- 후위 순회의 예
  - H-D-I-J-E-B-F-K-G-C-A



❑ 이진 트리의 순회

- 수식 이진 트리의 후위 순회
  - 수식 이진 트리를 후위 순회하면, 수식에 대한 후위 표기식을 구할 수 있다.
  - 예

    A * B – C / D ➔ A B * C D / –



이진 트리의 순회

- 이진 트리의 순회 프로그램 : 이진 트리 순회를 보여주는 간단한 예제

```
public class Ex9_1{
    public static void main(String[] args) {
        LinkedTree tree7 = new LinkedTree('D', null, null);
        LinkedTree tree6 = new LinkedTree('C', null, null);
        LinkedTree tree5 = new LinkedTree('B', null, null);
        LinkedTree tree4 = new LinkedTree('A', null, null);
        LinkedTree tree3 = new LinkedTree('/', tree6, tree7);
        LinkedTree tree2 = new LinkedTree('*', tree4, tree5);
        LinkedTree tree1 = new LinkedTree('-', tree2, tree3);

        System.out.print("preorder : ");
        tree1.preorder();

        System.out.print("\ninorder : ");
        tree1.inorder();

        System.out.print("\npostorder : ");
        tree1.postorder();
}
```

❑ 이진 트리의 순회

```
public class LinkedTree {
    private TreeNode root = null;

    public LinkedTree(char data, LinkedTree leftSubtree,
                                  LinkedTree rightSubtree) {
        root = new TreeNode();
        root.data = data;

        if (leftSubtree == null)
            root.leftChild = null;
        else
            root.leftChild = leftSubtree.root;

        if (rightSubtree == null)
            root.rightChild = null;
        else
            root.rightChild = rightSubtree.root;
    }

    // ... 다음 슬라이드에 계속
```

```
private class TreeNode {
    char data;
    TreeNode leftChild;
    TreeNode rightChild;
}
```

❑ 이진 트리의 순회

```
public void preorder() {   // 전위 순회를 위한 서비스 메소드
    preorder(root);
}

private void preorder(TreeNode root) {  // 전위 순회 메소드
    if (root != null) {
        System.out.print(root.data + " ");
        preorder(root.leftChild);
        preorder(root.rightChild);
    }
}
// ... 다음 슬라이드에 계속
```

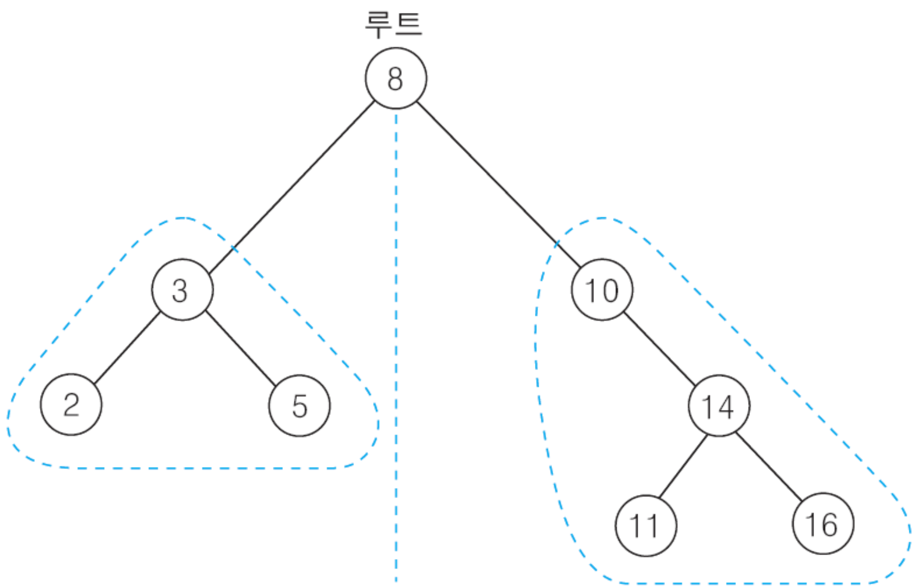
이진 트리의 순회

```
    public void inorder() {    // 중위 순회를 위한 서비스 메소드
        inorder(root);
    }

    private void inorder(TreeNode root)  {  // 중위 순회 메소드
        ...
    }

    public void postorder() {    // 후위 순회를 위한 서비스 메소드
        postorder(root);
    }

    private void postorder(TreeNode root)  { // 후위 순회 메소드
        ...
    }
}
```

❑ 이진 탐색 트리

❖ 이진 탐색 트리(binary search tree)

- "이진 검색 트리"
- 이진 트리에 탐색을 위한 조건을 추가한 자료구조로서 다음과 같이 정의된다.
  - (1) 모든 원소는 서로 다른 유일한 키(key)를 갖는다.
  - (2) **왼쪽** 서브트리 원소의 키들은 그 루트의 키보다 작다.
  - (3) **오른쪽** 서브트리 원소의 키들은 그 루트의 키보다 크다.
  - (4) 왼쪽 서브트리와 오른쪽 서브트리도 이진 탐색 트리이다.



이진 탐색 트리

❖ 이진 탐색 트리의 탐색 연산

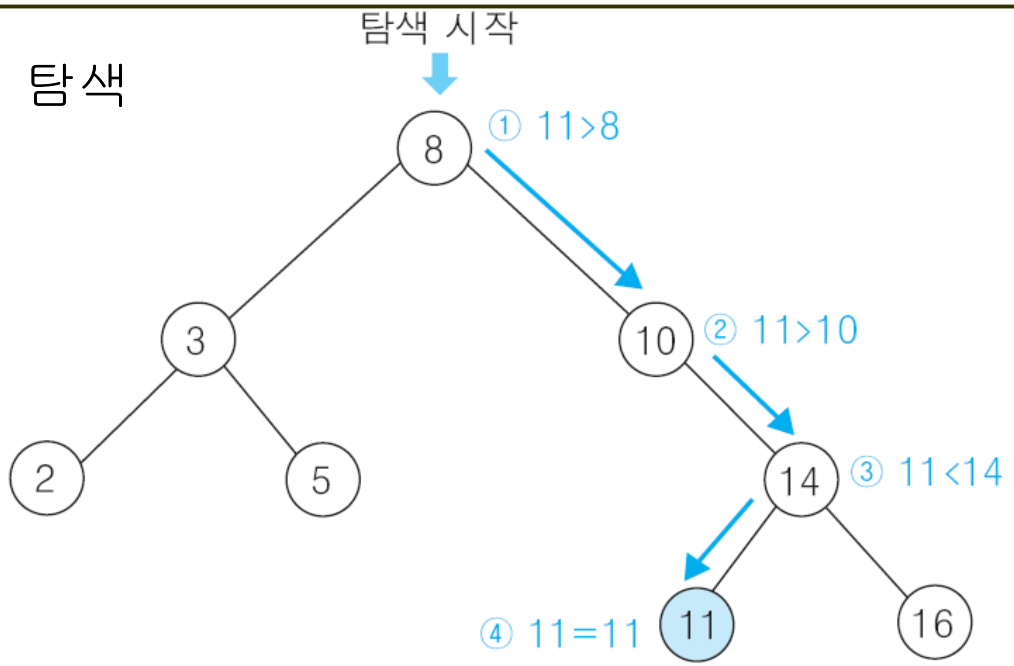
- 루트에서 시작한다.
- 탐색할 키값 x를 루트 노드의 키값과 비교한다.
  - (x = 루트노드의 키 값)인 경우 :
    원하는 원소를 찾았으므로 탐색연산 성공
  - (x < 루트노드의 키 값)인 경우 :
    루트노드의 왼쪽 서브트리에 대해서 탐색연산 수행
  - (x > 루트노드의 키 값)인 경우 :
    루트노드의 오른쪽 서브트리에 대해서 탐색연산 수행
- 서브트리에 대해서 재귀적으로 탐색 연산을 반복한다.

❑ 이진 탐색 트리

- 탐색 알고리즘

```
searchBST(bsT, x)   // bsT를 루트로 하는 이진탐색트리에서 키값이 x인
                    //     노드를 찾아 리턴; 탐색에 실패하면 null 리턴
    if (bsT = null) then  return null;
    if (x = bsT.key) then  return bsT;
    else if (x < bsT.key) then  return searchBST(bsT.leftChild, x);
    else return searchBST(bsT.rightChild, x);
end searchBST()
```

- 예) 원소 11 탐색



❑ 이진 탐색 트리

❖ 이진 탐색 트리의 삽입 연산

1) 먼저 탐색 연산을 수행한다.
   - 삽입할 원소와 동일한 원소가 트리에 있으면 삽입할 수 없으므로 동일한 원소가 트리에 있는지 탐색하여 확인한다.
   - 탐색 성공하면 삽입할 수 없다.
   - 탐색 실패하면 삽입할 수 있다.

2) 탐색 실패한 위치에 원소를 삽입한다.

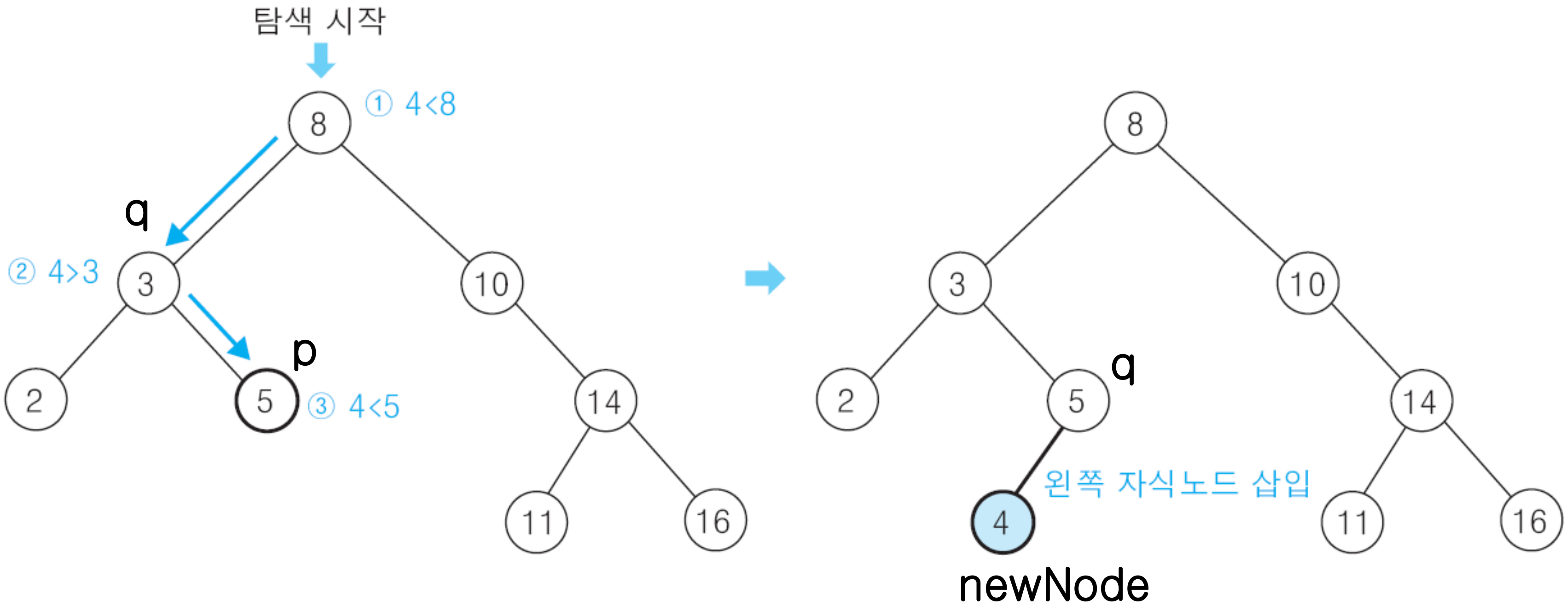
❑ 이진 탐색 트리

- 삽입 알고리즘

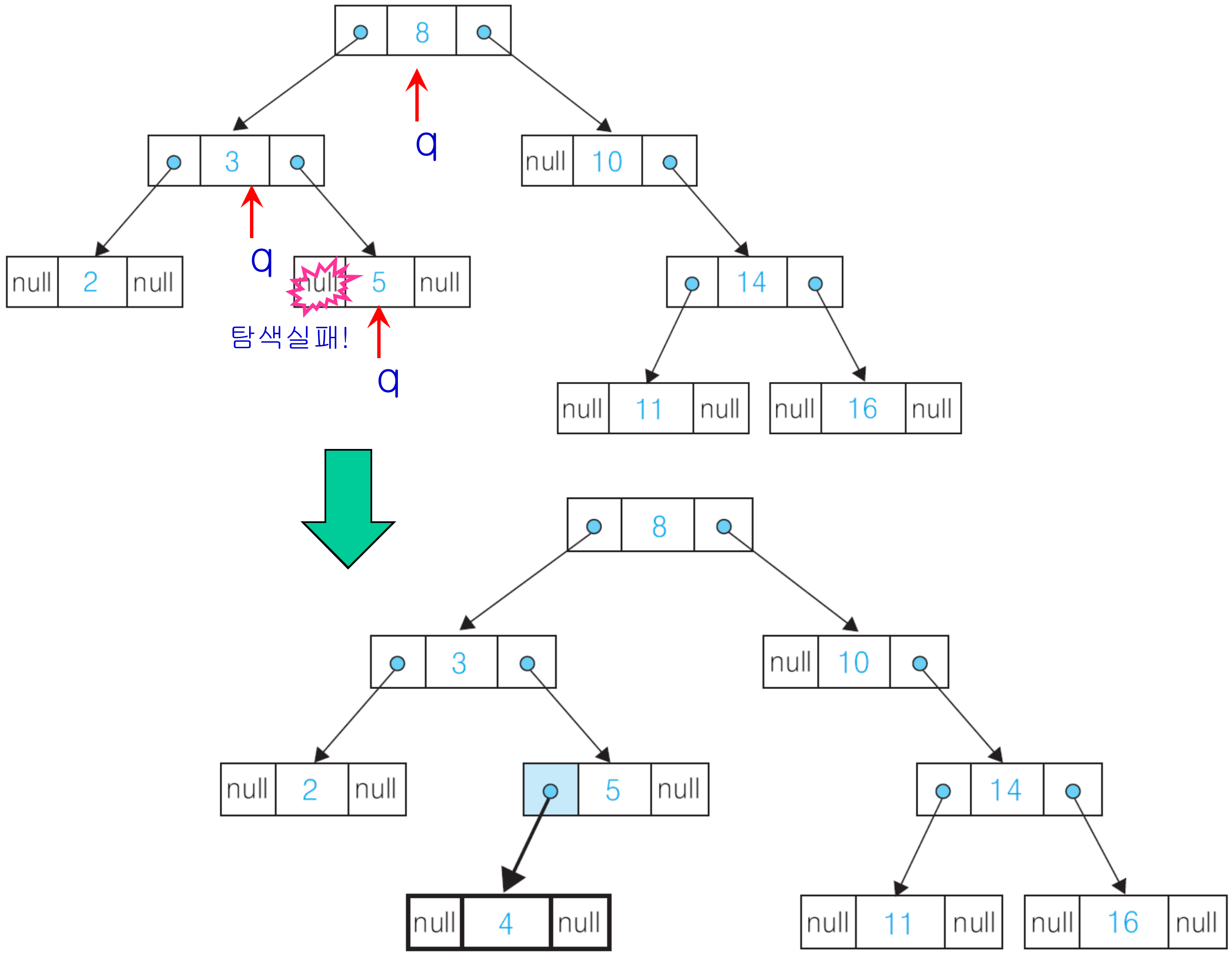
```
insertBST(bsT, x) // bsT를 루트로 하는 이진탐색트리에 키값 x 인 노드를
                  // 삽입하고, 삽입된 트리의 루트 노드를 리턴
    p ← bsT;
    while (p ≠ null) do {                                    ┐
        if (x = p.key) then 삽입 실패;                          │
        q ← p;                                               │ 삽입할 자리 탐색
        if (x < p.key) then p ← p.leftChild;                 │
        else p ← p.rightChild;                               │
    }                                                        ┘
    newNode ← getNode();                                     ┐
    newNode.key ← x;                                         │ 삽입할 노드 만들기
    newNode.leftChild ← null;                                │
    newNode.rightChild ← null;                               ┘
    if (bsT = null) then return newNode                      ┐
    else if (x < q.key) then q.leftChild ← newNode;          │ 탐색한 자리에 노드 연결
    else q.rightChild ← newNode;                             ┘
    return bsT;
end insertBST()
```

❑ 이진 탐색 트리

- 예) 원소 4 삽입



- 연결 자료구조로 구현한 이진트리에서의 원소 4 삽입



❖ 이진 탐색 트리의 삭제 연산

1) 먼저 탐색 연산을 수행하여 삭제할 노드를 찾는다.

2) 찾은 노드를 삭제한다.

- 다음과 같은 3가지 경우로 나눌 수 있다.

  [경우1] 삭제할 노드가 단말노드인 경우

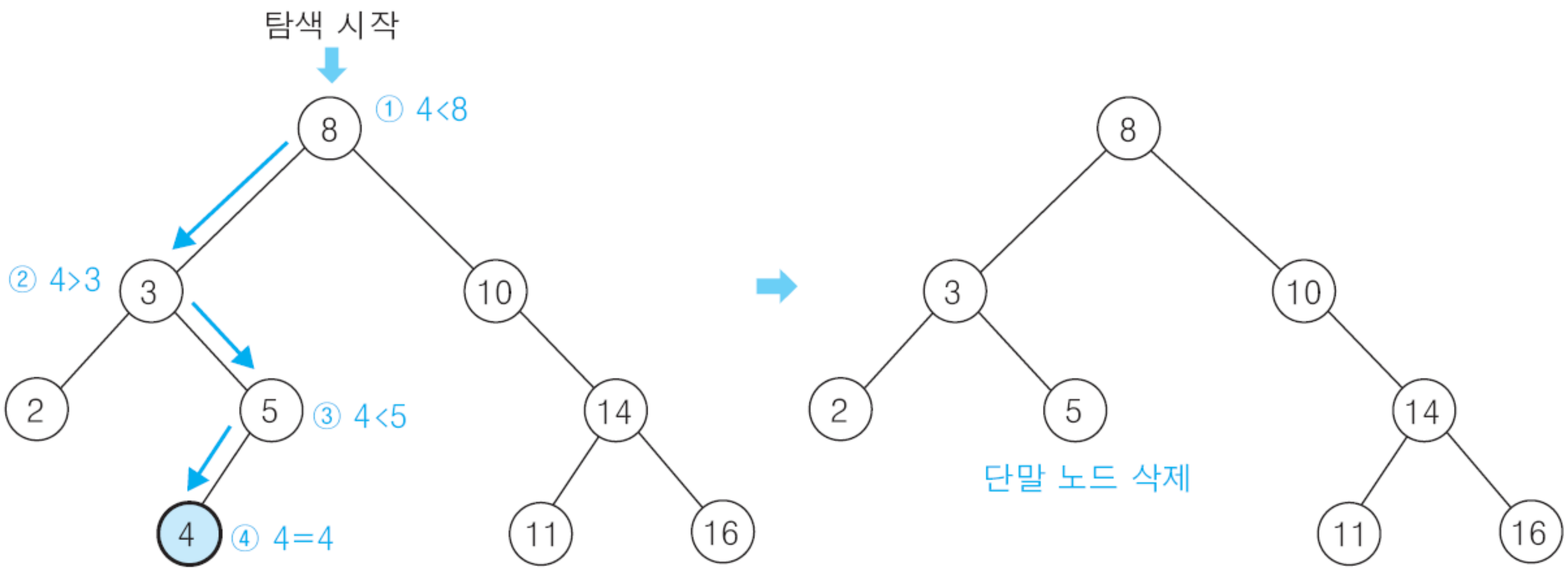
  [경우2] 삭제할 노드가 하나의 자식노드를 가진 경우

  [경우3] 삭제할 노드가 두개의 자식노드를 가진 경우

- 노드의 삭제 후에도 이진 탐색 트리를 유지해야 하므로 이진 탐색 트리의 재구성 작업(후속 처리)이 필요하다.

이진 탐색 트리

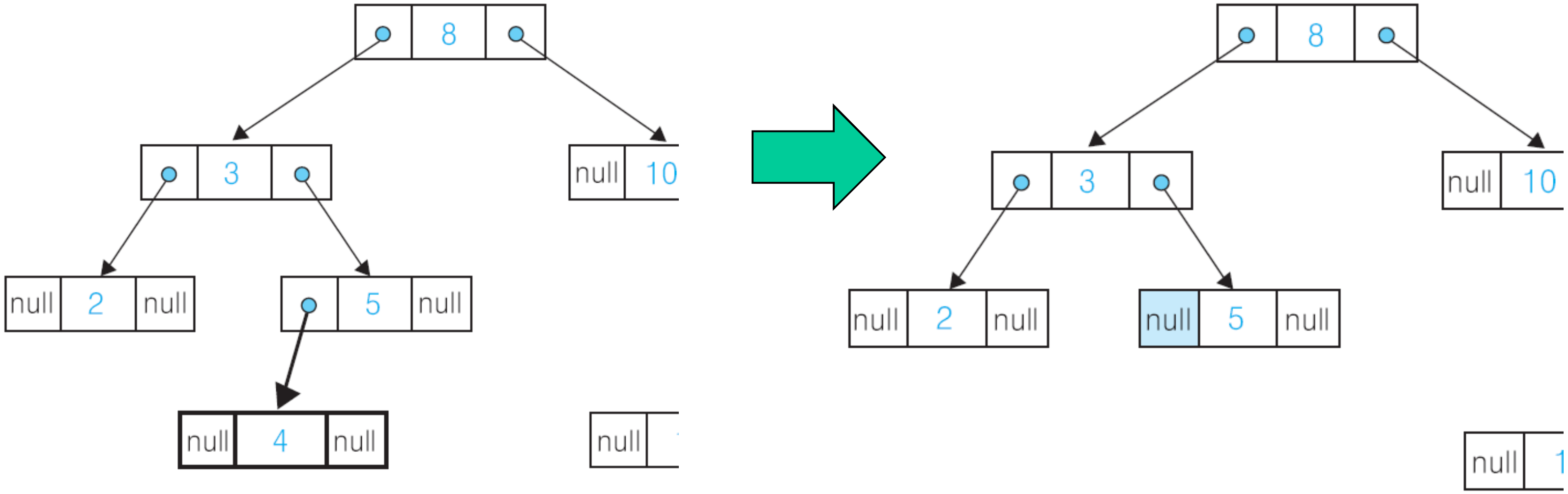
- [경우1] 단말 노드의 삭제 – 간단
  - 예) 노드 4 삭제



(1) 삭제할 노드 탐색　　　(2) 노드 삭제

❑ 이진 탐색 트리

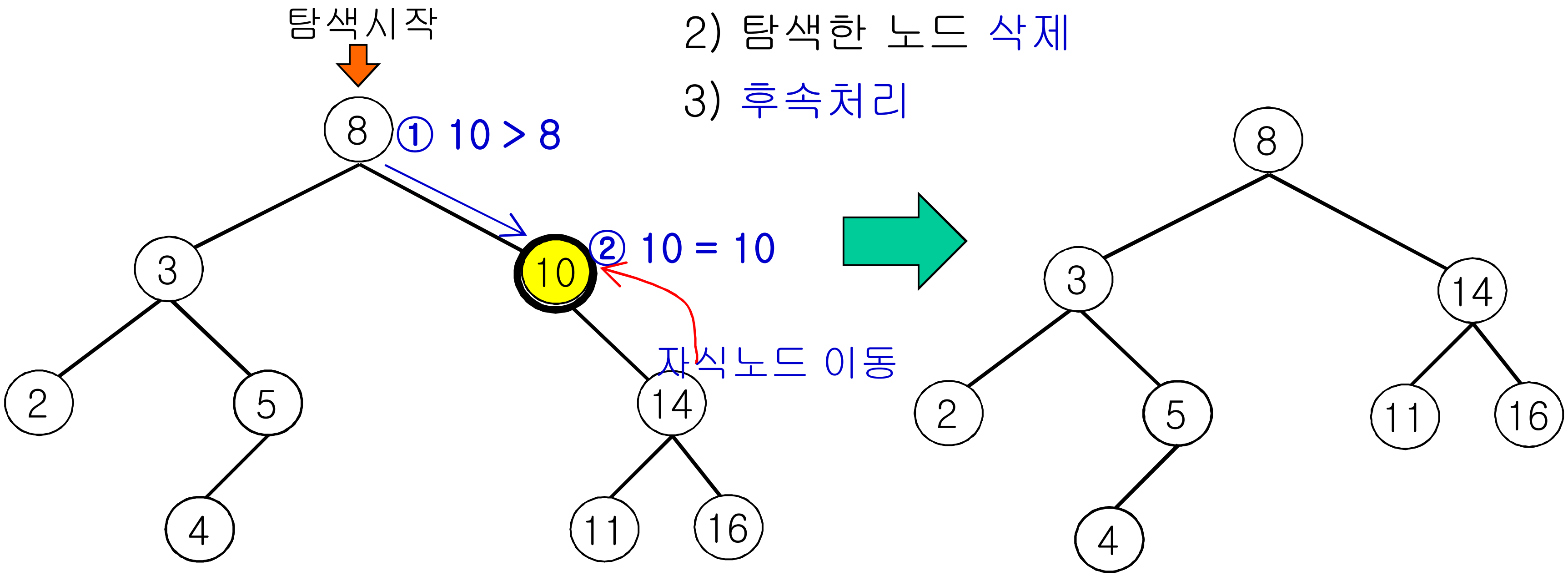
- 연결 자료구조로 구현한 이진트리에서의 원소 4 삭제
  - ➢ 노드를 삭제하고, 삭제한 노드의 부모 노드의 링크 필드를 null로 설정



❑ 이진 탐색 트리

- [경우2] 자식 노드가 하나인 노드의 삭제
  - 노드 p를 삭제하면, p의 자식 노드는 트리에서 연결이 끊어져서 고아가 된다.
  - 후속 처리 : **<u>삭제한 노드의 자리를 자식 노드에게 물려준다.</u>**
  - 예) 노드 10 삭제

1) 삭제할 노드 탐색
2) 탐색한 노드 삭제
3) 후속처리



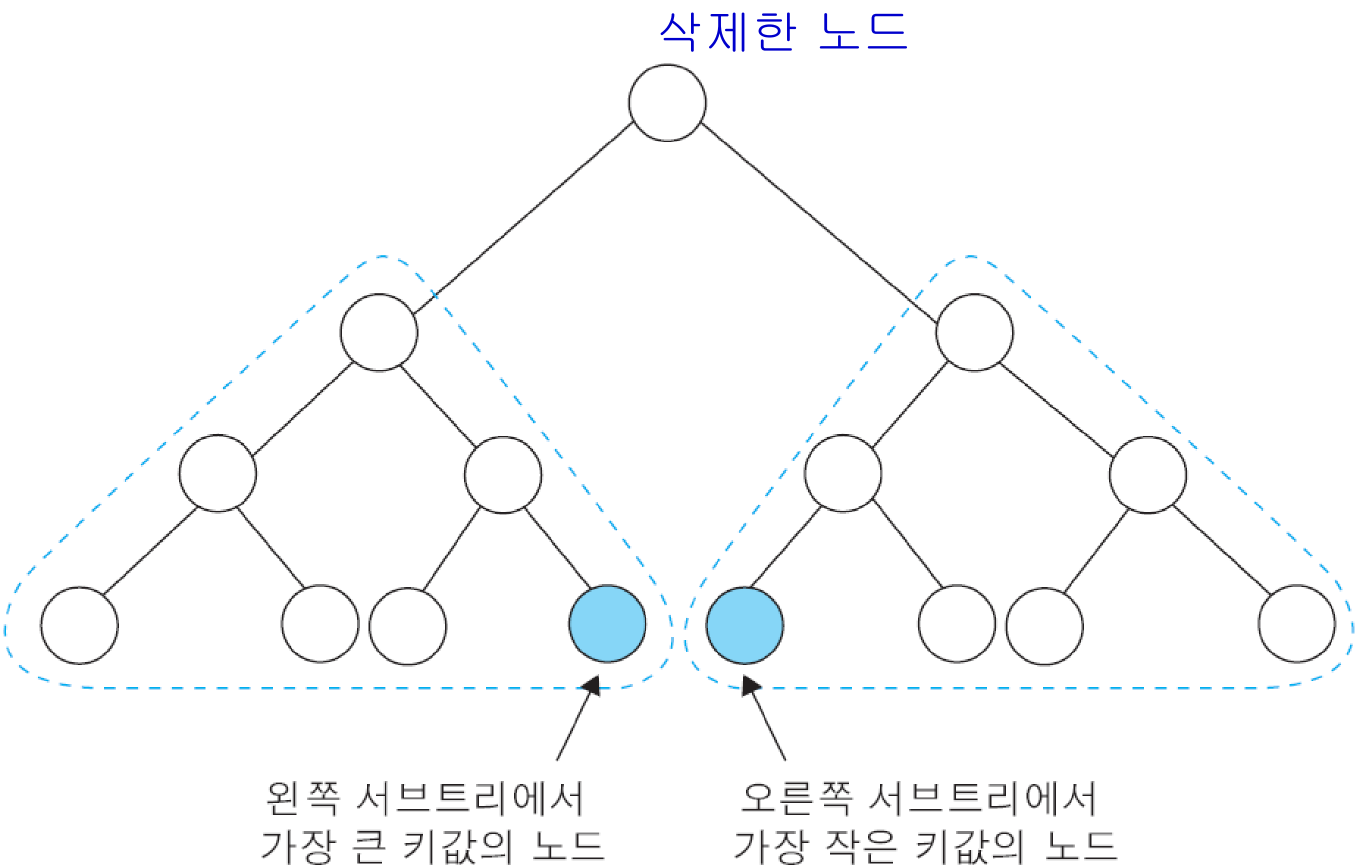
❑ 이진 탐색 트리

- [경우3] 자식 노드가 둘인 노드의 삭제
  - 노드 p를 삭제하면, p의 자식 노드들은 트리에서 연결이 끊어져 고아가 된다.
  - 후속 처리 : 삭제한 노드의 자리를 **후계자에게 물려준다. 후계자는 자손노드들 중에서 선택한다.**

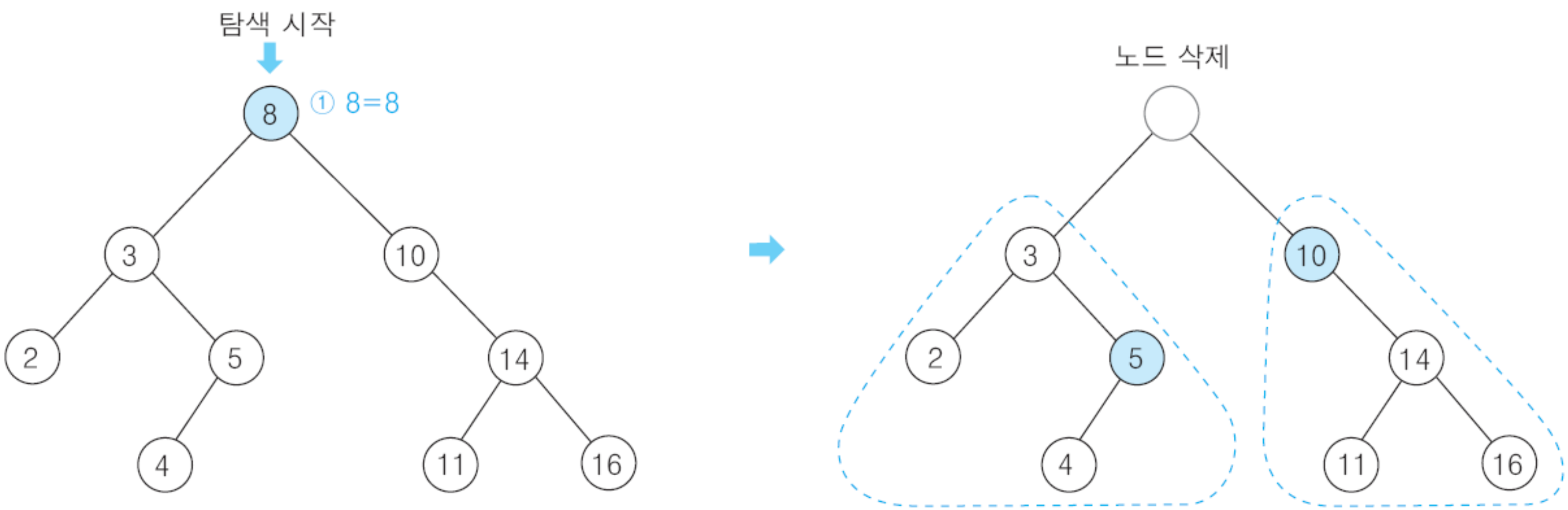
  - 후계자 선택 방법 두가지
    1) **왼쪽 서브트리에서 가장 큰 자손노드 선택** : 왼쪽 서브트리의 오른쪽 링크를 따라 계속 이동하여 오른쪽 링크 필드가 null인 노드. 즉, 가장 오른쪽에 있는 노드가 후계자가 된다.
    2) **오른쪽 서브트리에서 가장 작은 자손노드 선택** : 오른쪽 서브트리에서 왼쪽 링크를 따라 계속 이동하여 왼쪽 링크 필드가 null인 노드. 즉, 가장 왼쪽에 있는 노드가 후계자가 된다.

❑ 이진 탐색 트리

- 삭제한 노드의 자리를 물려받을 수 있는 후계자 노드

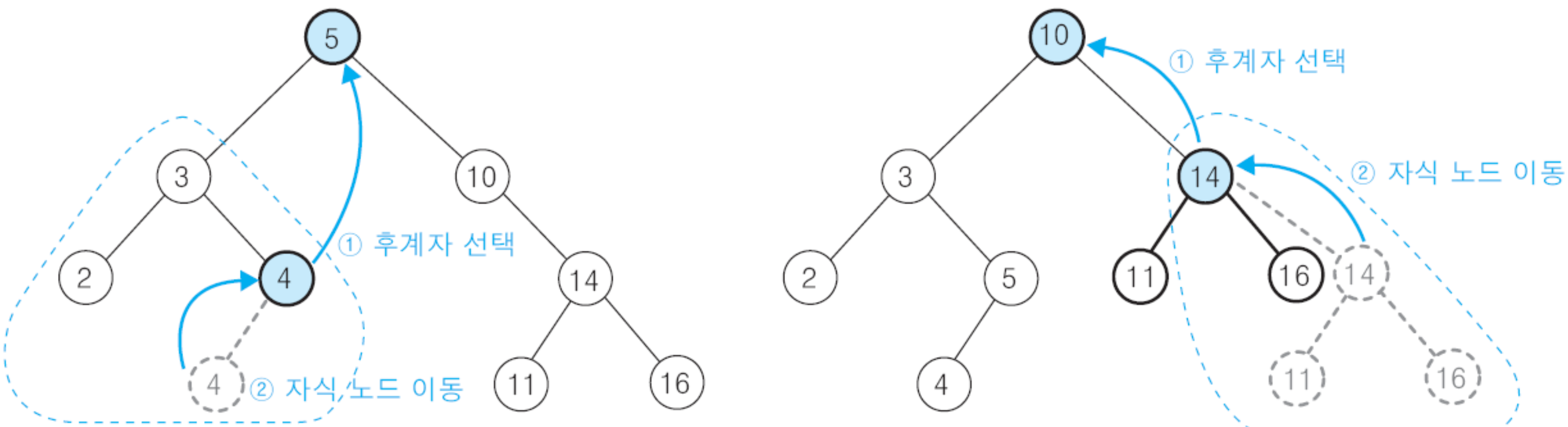


- 예) 노드 8 삭제



(1) 삭제할 노드 탐색

(2) 노드 삭제



(a) 트리 재구성_노드 5를 후계자로 선택한 경우

(b) 트리 재구성_노드 10을 후계자로 선택한 경우

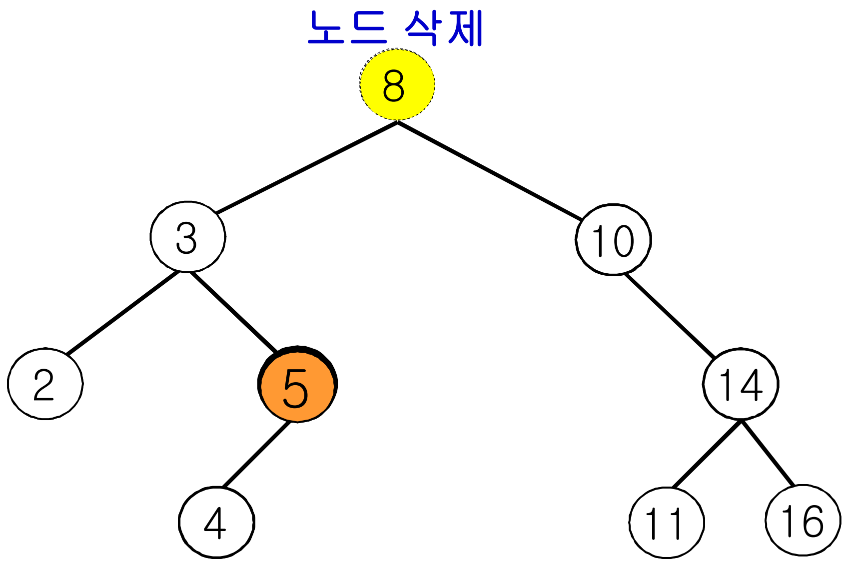
❑ 이진 탐색 트리

- 노드 5를 후계자로 선택한 경우

  ① 후계자 노드 5를 삭제하여 삭제노드 8의 자리로 옮긴다.

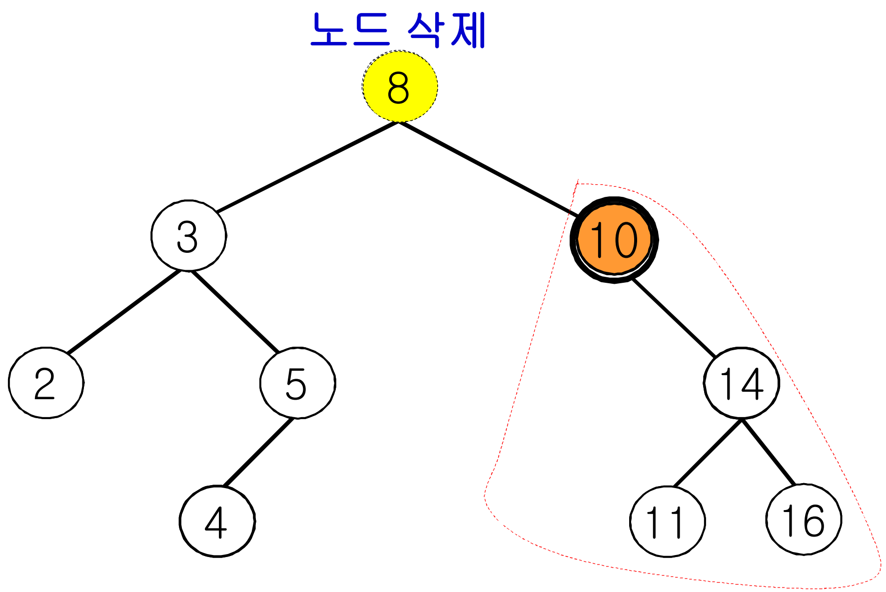
  ② 이 때 후계자 노드는 하나의 자식 노드만을 가지므로 노드 삭제 [경우2]에 해당하는 후속 처리를 해 준다.
  즉, 5의 원래자리는 자식노드 4에게 물려준다.

  (후계자 노드가 단말 노드이면 별도의 후속 처리 필요 없다)



❑ 이진 탐색 트리

- 노드 10을 후계자로 선택한 경우 (앞의 예와 대칭임)
  ① 후계자 노드 10을 삭제하여 삭제노드 8의 자리로 옮긴다.
  ② 이 때 후계자 노드는 하나의 자식 노드만을 가지므로 노드 삭제 [경우2]에 해당하는 후속 처리를 해 준다.
  즉, 10의 원래자리는 자식노드 14에게 물려준다.
  (후계자 노드가 단말 노드이면 별도의 후속 처리 필요 없다)



❑ 이진 탐색 트리

- 삭제 알고리즘

```
deleteBST(bsT, x)   // bsT를 루트로 하는 이진탐색트리에서 키값 x 인
                    // 노드를 삭제하고, 삭제된 트리의 루트 노드를 리턴
    p ← 삭제할 노드;
    parent ← 삭제할 노드의 부모 노드;

    if (p = null) then 삭제 실패;

    if (p.leftChild = null and p.rightChild = null) then {   // [경우1]
        if (parent = null) then return null;
        else if (parent.leftChild = p) then parent.leftChild ← null;
        else parent.rightChild ← null;
    }
    // ... 다음 슬라이드에 계속
```

❑ 이진 탐색 트리

```
    else if (p.leftChild = null or p.rightChild = null) then {  // [경우2]
        if (p.leftChild ≠ null) then {
            if (parent = null) return p.leftChild;
            else if (parent.leftChild = p) then parent.leftChild ← p.leftChild;
            else parent.rightChild ← p.leftChild;
        }
        else {
            if (parent = null) return p.rightChild;
            else if (parent.leftChild = p) then parent.leftChild ← p.rightChild;
            else parent.rightChild ← p.rightChild;
        }
    }
    else {   // [경우3] 방법 1
        q ← maxNode(p.leftChild);
        p.key ← q.key;
        p.leftChild ← deleteBST(p.leftChild, q.key);   // 노드 q를 삭제
    }
    return bsT;
end deleteBST()
```

이진 탐색 트리

- 이진 탐색 트리의 연산 프로그램

```
public class Ex9_2 {
    public static void main(String[] args){
        BinarySearchTree bsT = new BinarySearchTree();

        bsT.insert('G');  bsT.insert('I');  bsT.insert('H');
        bsT.insert('D');  bsT.insert('B');  bsT.insert('M');
        bsT.insert('N');  bsT.insert('A');  bsT.insert('J');
        bsT.insert('E');  bsT.insert('Q');

        System.out.print("\nBinary Tree >>> ");
        bsT.print();

        System.out.print("Is There \"A\" ? >>> ");
        bsT.search('A');

        System.out.print("Is There \"Z\" ? >>> ");
        bsT.search('Z');
    }
}
```

❑ 이진 탐색 트리

```
public class BinarySearchTree {
    private TreeNode root = null;

    public void insert(char key) {
        root = insertKey(root, key);
    }

    public void search(char key) {
        TreeNode p = searchBST(key);
        if(p != null)
            System.out.println("Searching Success! Searched key : "+ key);
        else
            System.out.println("Searching fail! There is no " + key);
    }

    public void print() {
        inorder(root);
        System.out.println();
    }
    // ... 다음 슬라이드에 계속
```

```
private class TreeNode{
    char data;
    TreeNode leftChild;
    TreeNode rightChild;
}
```

이진 탐색 트리

```
private TreeNode insertKey(TreeNode p, char key) {
    if(p == null) {
        TreeNode newNode = new TreeNode();
        newNode.data = key;
        newNode.leftChild = null;
        newNode.rightChild = null;
        return newNode;
    }
    else if(key < p.data) {
        p.leftChild = insertKey(p.leftChild, key);
        return p;
    }
    else if(key > p.data) {
        p.rightChild = insertKey(p.rightChild, key);
        return p;
    }
    else {
        System.out.println("Insertion fail! key duplication : " + key);
        return p;  // no insertion
    }
}  // ... 다음 슬라이드에 계속
```

❑ 이진 탐색 트리

```
    private TreeNode searchBST(char key) {
        TreeNode p = root;

        while(p != null) {
            if(key < p.data)  p = p.leftChild;
            else if (key > p.data) p = p.rightChild;
            else return p;    // 탐색 성공
        }
        return null;    // 탐색 실패
    }

    private void inorder(TreeNode p) {
        if(p != null) {
            inorder(p.leftChild);
            System.out.print(p.data + " " );
            inorder(p.rightChild);
        }
    }
}
```

이진 탐색 트리

- 이진 탐색 트리의 연산 수행 시간
  - 노드 수가 n이고, 높이가 h인 경우, 삽입/삭제/검색 시간은 O(h)
    - 트리 좌우 균형이 맞아 높이가 O(log n)인 경우

      O(h) = O(log n)
    - 편향 이진 트리인 경우

      O(h) = O(n)
  - 즉, 트리의 높이를 낮게 유지할수록 수행시간 복잡도가 낮아짐

- 이진 탐색 트리의 삽입/삭제/검색 시간이 O(log n)이 되려면
  - 트리의 좌우 균형을 맞추어 높이를 O(log n)으로 유지해야 한다.

  ➔ 균형이 맞는 트리(balanced tree) :

  AVL tree, Red-Black tree, …

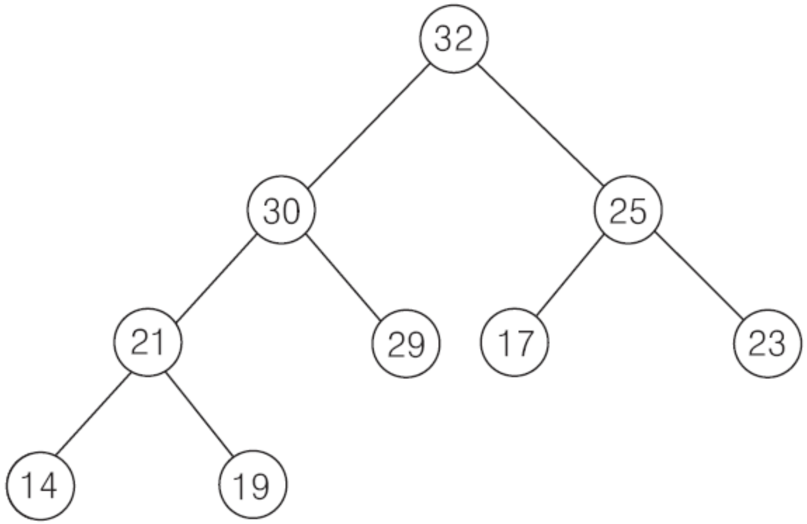
힙

힙(heap)

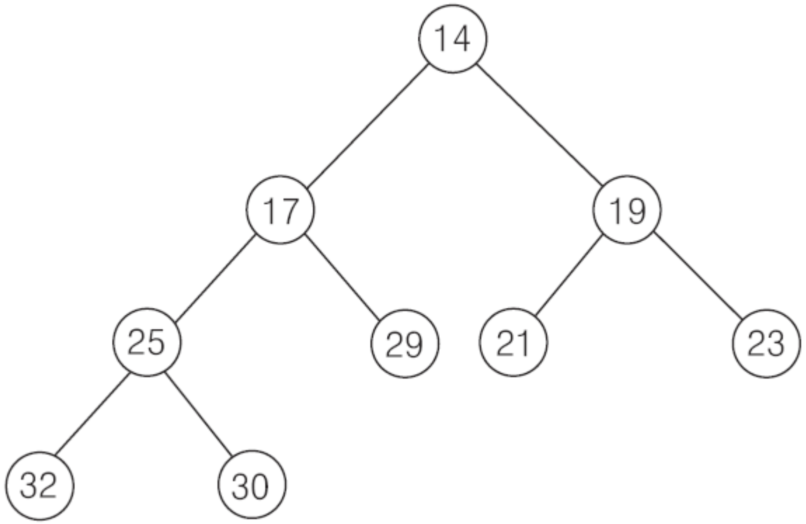
- 키 값이 가장 큰(또는 작은) 노드를 쉽게 찾기 위해 만든 완전 이진 트리로서, 다음 두가지로 나뉜다.
- **최대 힙(max heap)**
  - 키 값이 가장 큰 노드를 빨리 찾기 위한 자료구조
  - 완전 이진 트리이며, 다음 특성을 만족한다.
    부모노드의 키 값 ≥ 자식노드의 키 값
  - 따라서 키 값이 가장 큰 노드는 루트 노드
- **최소 힙(min heap)**
  - 키 값이 가장 작은 노드를 빨리 찾기 위한 자료구조
  - 완전 이진 트리이며, 다음 특성을 만족한다.
    부모노드의 키 값 ≤ 자식노드의 키 값
  - 따라서 키 값이 가장 작은 노드는 루트 노드

힙

- 힙의 예



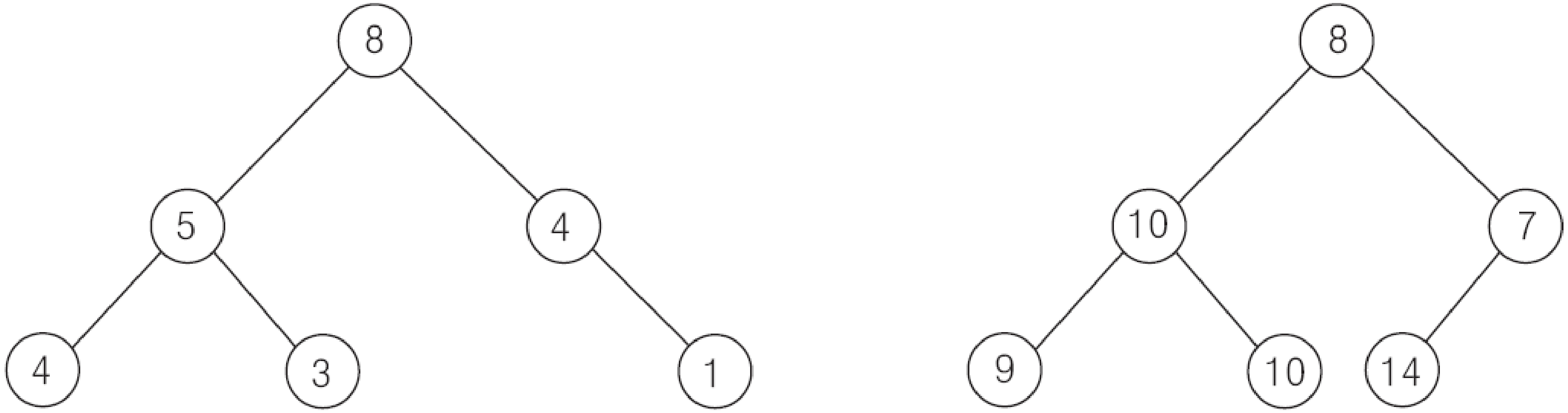
(a) 최대 힙



(b) 최소 힙

힙

- 힙이 아닌 이진 트리의 예



(a) (b)

힙

❖ 힙의 추상 자료형

```
ADT MaxHeap
데이터 : n개의 원소로 구성된 완전 이진 트리로서 각 노드의 키값은 그의 자식
        노드의 키값보다 크거나 같다.(부모 노드의 키값 ≥ 자식 노드의 키값)
연산 :
    heap∈Heap; item∈Element;
    createHeap() ::= create an empty heap;
        // 공백 힙의 생성 연산
    isEmpty(heap) ::= if (heap is empty) then return true; else return false;
        // 힙이 공백인지를 검사하는 연산
    insertHeap(heap, item) ::= insert item into heap;
        // 힙의 적당한 위치에 원소(item)를 삽입하는 연산
    deleteHeap(heap) ::= if (isEmpty(heap)) then return error;
                        else {  item ← 힙에서 가장 큰 원소;
                                remove {힙에서 가장 큰 원소};
                                return item;  }
        // 힙에서 키값이 가장 큰 원소를 삭제하고 반환하는 연산
End Heap()
```

❖ 히프에서의 삽입 연산

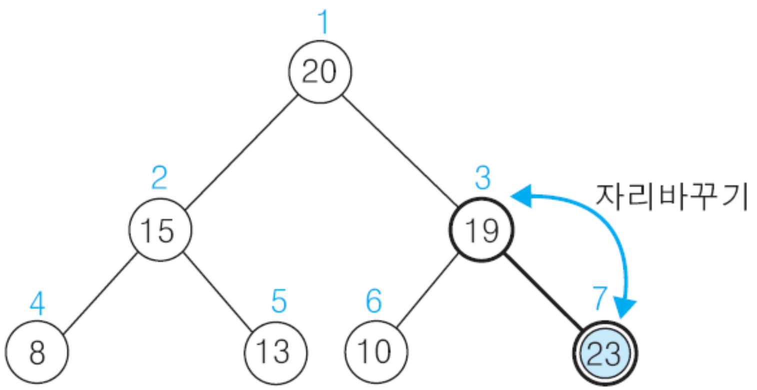
1단계 : 완전 이진 트리를 유지하면서 노드를 확장하여, 삽입할 원소를 임시 저장

- 노드가 n개인 완전 이진 트리에서 다음 노드의 확장 자리는 n+1번의 노드가 된다.
- n+1번 자리에 노드를 확장하고, 그 자리에 삽입할 원소를 임시 저장한다.

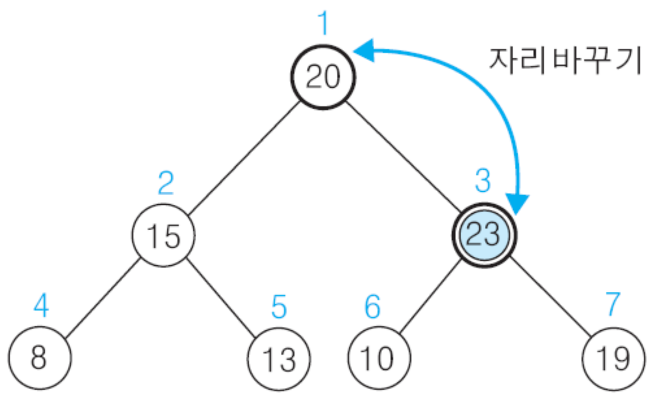
2단계 : 만들어진 완전 이진 트리 내에서 삽입 원소의 제자리를 찾는다.

- 현재 위치에서 부모노드와 비교하여 크기 관계를 확인한다.
- {현재부모노드의 키 값 ≥ 삽입 원소의 키 값}의 관계가 성립하지 않으면,
  현재부모노드의 원소와 삽입 원소의 자리를 서로 바꾼다.

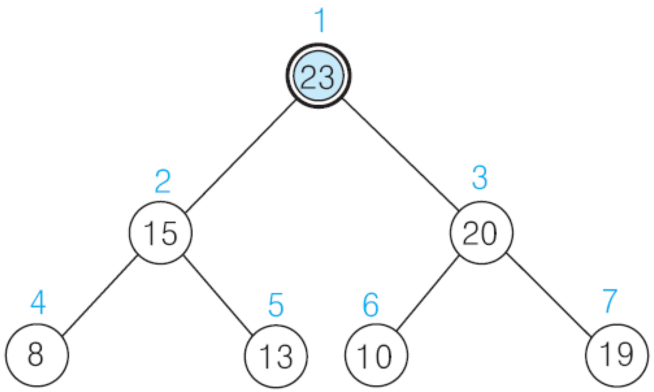
- 히프에서의 삽입 연산 예2) **– 23을 삽입하는 경우**



(1) (삽입 노드 23 > 부모 노드 19) : 자리바꾸기



(2) (삽입 노드 23 > 부모 노드 20) : 자리바꾸기



(3) 비교할 부모 노드가 없으므로 자리 확정

히프

- 히프에서의 삽입 연산 알고리즘

➢[알고리즘 9-7]

```
insertHeap(heap, item)
    if (n = heapSize) then heapFull();
    n ← n+1; // ❶
    for (i ← n; ;) do {
        if (i = 1) then exit;
        if (item ≤ heap[ └i/2┘ ]) then exit; // ❷
            heap[i] ← heap[ └i/2┘ ]; // ❸
        i ← └i/2┘  // ❹
    }
    heap[i] ← item; // ❺
end insertHeap()
```

❖ 히프에서의 삭제 연산

- 히프에서는 루트 노드의 원소만을 삭제 할 수 있다.

1단계 : 루트 노드의 원소를 삭제하여 반환한다.

2단계 : 원소의 개수가 n-1개로 줄었으므로, 노드의 수가 n-1인 완전 이진 트리로 조정한다.

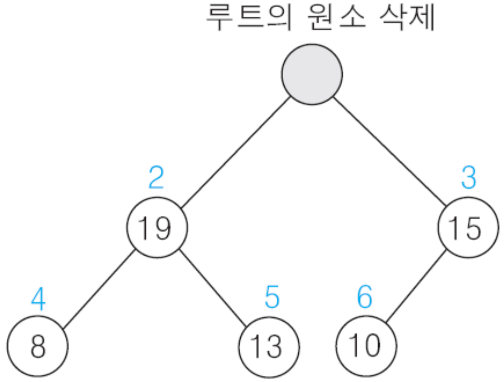
- 노드가 n개인 완전 이진 트리에서 노드 수 n-1개의 완전 이진 트리가 되기 위해서 마지막 노드, 즉 n번 노드를 삭제한다.
- 삭제된 n번 노드에 있던 원소는 비어있는 루트노드에 임시 저장한다.

3단계 : 완전 이진 트리 내에서 임시 저장된 원소의 제자리를 찾는다.

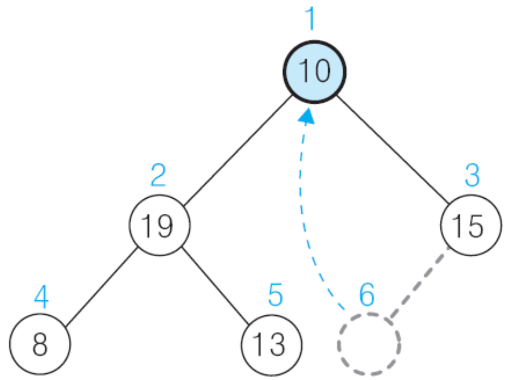
- 현재 위치에서 자식노드와 비교하여 크기 관계를 확인한다.
- {임시저장 원소의 키 값 ≥ 현재자식노드의 키 값 }의 관계가 성립하지 않으면, 현재자식노드의 원소와 임시저장 원소의 자리를 서로 바꾼다.

히프

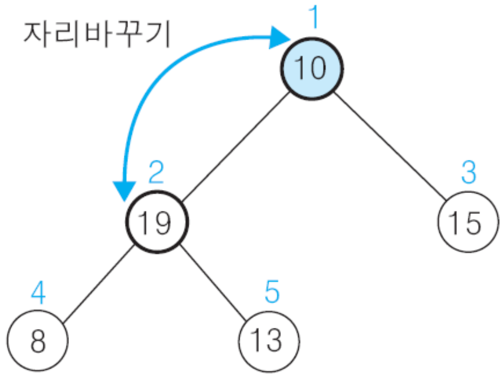
- 히프에서의 삭제 연산 예)



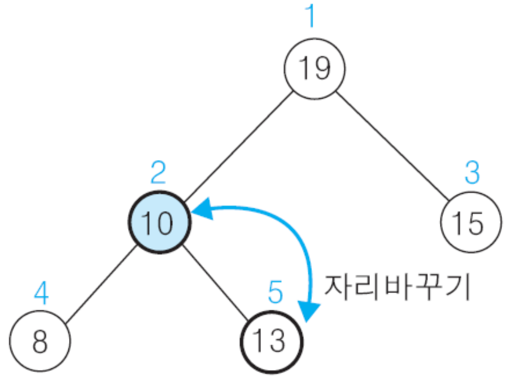
(1) 루트 노드의 원소 삭제



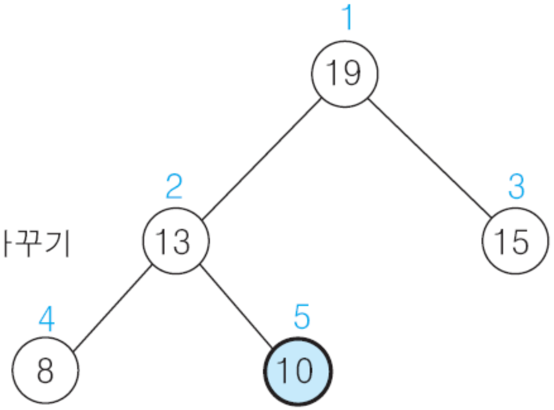
(2) 마지막 노드 삭제



(3) (삽입 노드 10 < 자식노드 19) : 자리바꾸기



(4) (삽입 노드 10 < 자식 노드 13) : 자리바꾸기



(5) 자리 확정

❑ 히프

- 히프에서의 삭제 연산 알고리즘

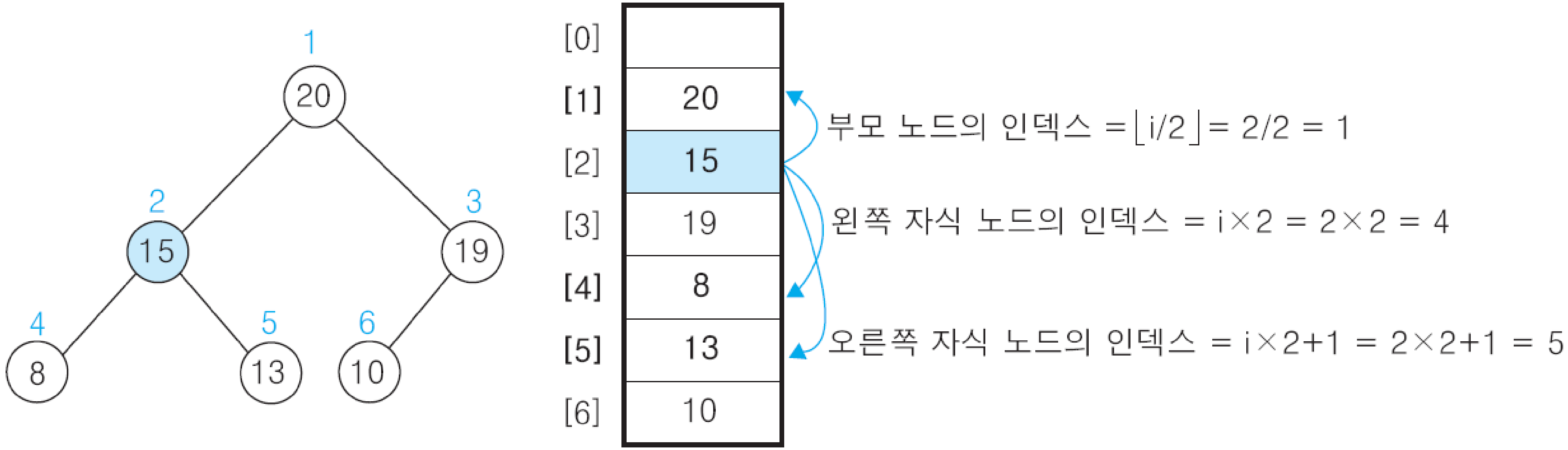
➢[알고리즘 9-8]

```
deleteHeap(heap)
    if (n = 0) then return error;
    item ← heap[1]; // ❶
    temp ← heap[n]; // ❷
    n ← n-1; // ❸
    i ← 1; // ❹
    j ← 2;
    while (j ≤ n) do {
        if (j < n) then
            if (heap[j] < heap[j+1]) then j ← j+1; // ❺
        if (temp ≥ heap[j]) then exit;
        heap[i] ← heap[j];
        i ← j; // ❻
        j ← j*2;
    }
    heap[i] ← temp; // ❼
    return item; // ❽
end deleteHeap()
```

❑ 히프의 구현

❖ 순차 자료구조를 이용한 히프의 구현

- 부모노드와 자식노드를 찾기 쉬운 1차원 배열의 순차 자료구조 이용
- 1차원 배열을 이용한 히프의 표현 예



❑ 히프의 구현

- 순차 자료구조를 이용한 최대 힙 프로그래밍

➢[예제 9-3]

```
class Heap{
	private int heapSize;
	private int itemHeap[];

	public Heap(){
		heapSize = 0;
		itemHeap = new int [50];
	}

	public void insertHeap(int item){
		int i = ++heapSize;
		while((i != 1) && (item > itemHeap[i/2])){
			itemHeap[i] = itemHeap[i/2];
			i/=2;
		}
		itemHeap[i] = item;
	}
```

➢>> 계속

히프의 구현

➤[예제 9-3]

```

    public int getHeapSize(){
        return this.heapSize;
    }

    public int deleteHeap(){
        int parent, child;
        int item, temp;
        item = itemHeap[1];
        temp = itemHeap[heapSize--];
        parent = 1; child = 2;

        while(child <= heapSize){
            if((child < heapSize) && (itemHeap[child] < itemHeap[child+1]))
                child++;
            if(temp >= itemHeap[child]) break;

            itemHeap[parent] = itemHeap[child];
```

➤>> 계속

❑ 히프의 구현

➢[예제 9-3]

```
                parent = child;
                child *= 2;
            }
            itemHeap[parent] = temp;
            return item;
        }

        public void printHeap(){
            System.out.printf("\nHeap >>> ");
            for(int i=1; i<=heapSize; i++)
                System.out.printf("[%d] ", itemHeap[i]);
        }
    }

    class Ex9_3{
        public static void main(String args[]){
            int n, item;
            Heap h = new Heap();
```

➢>> 계속

히프의 구현

[예제 9-3]

```
        h.insertHeap(13);
        h.insertHeap(8);
        h.insertHeap(10);
        h.insertHeap(15);
        h.insertHeap(20);
        h.insertHeap(19);

        h.printHeap();

        n = h.getHeapSize();
        for(int i=1; i<=n; i++){
            item = h.deleteHeap();
            System.out.printf("\n deleted Item : [%d]", item);
        }
    }
}
```

>> 계속