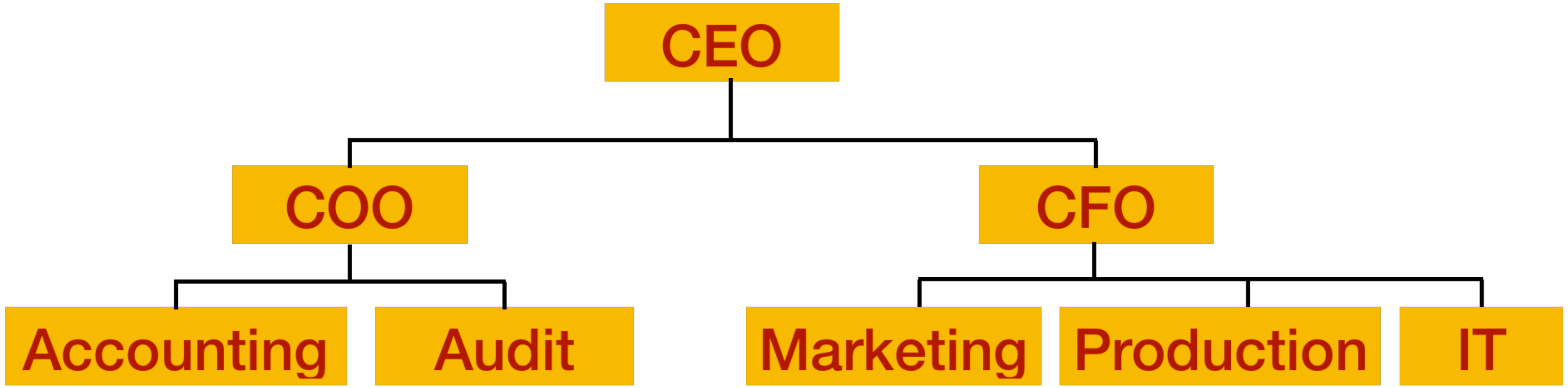
Tree

즐겁고 알찬 자료구조 튜터링

Tree

Tree

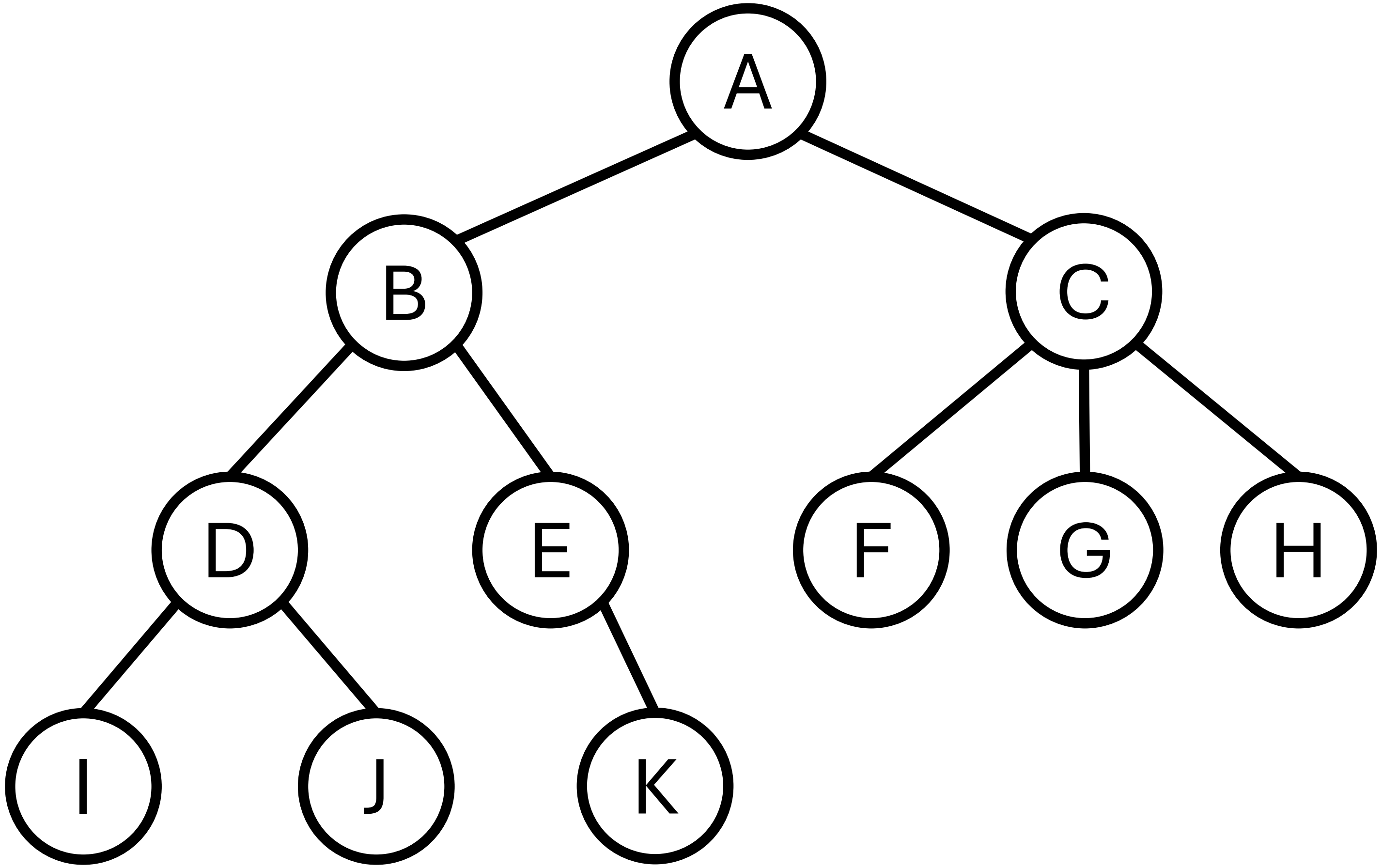
- 계층적 관계(Hierarchical Relationship)을 표현하는 비선형 자료구조



A Corporate Tree

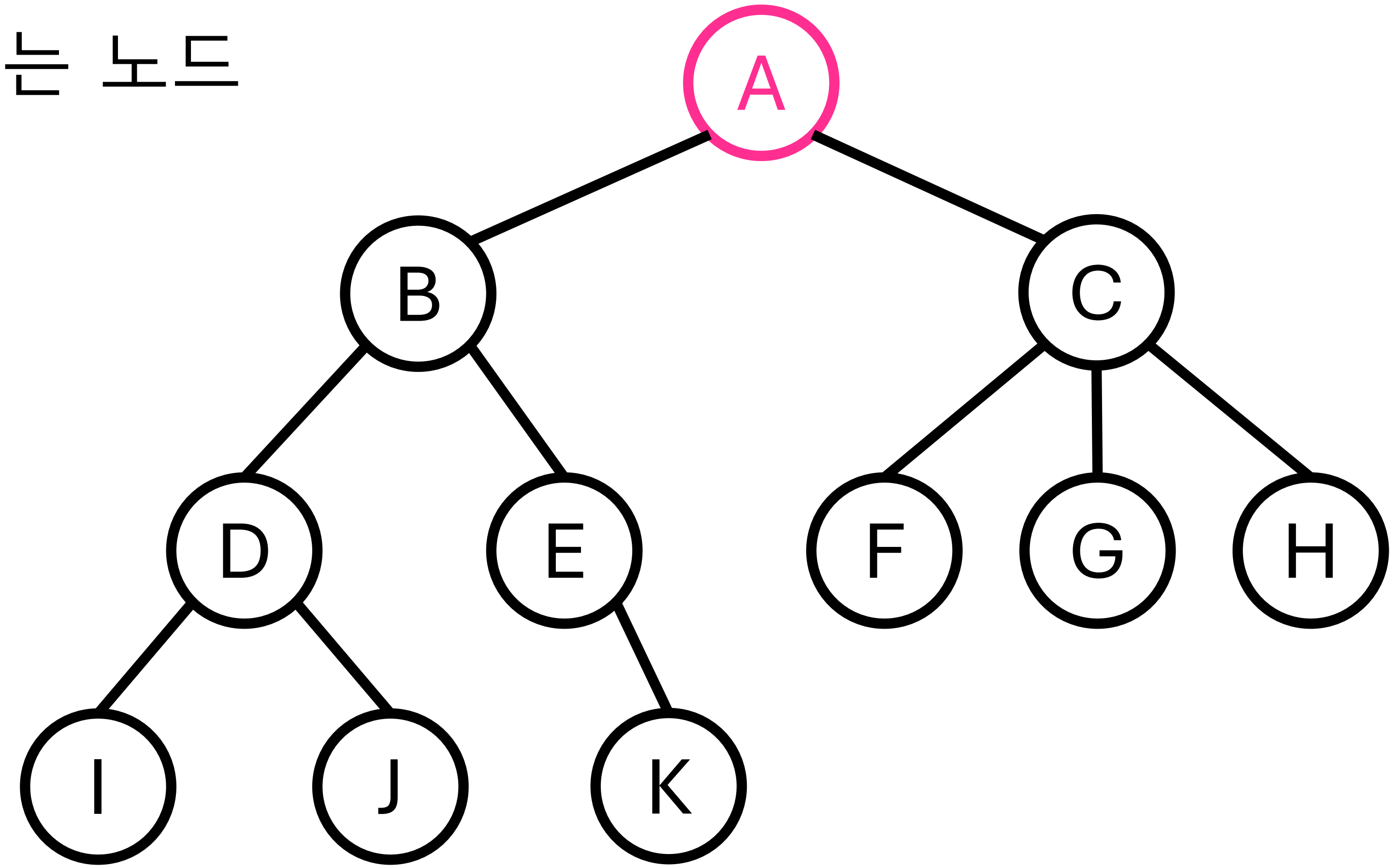
Tree의 용어

- root
- internal node
- external node
- ancestors
- depth
- height
- descendant
- subtree



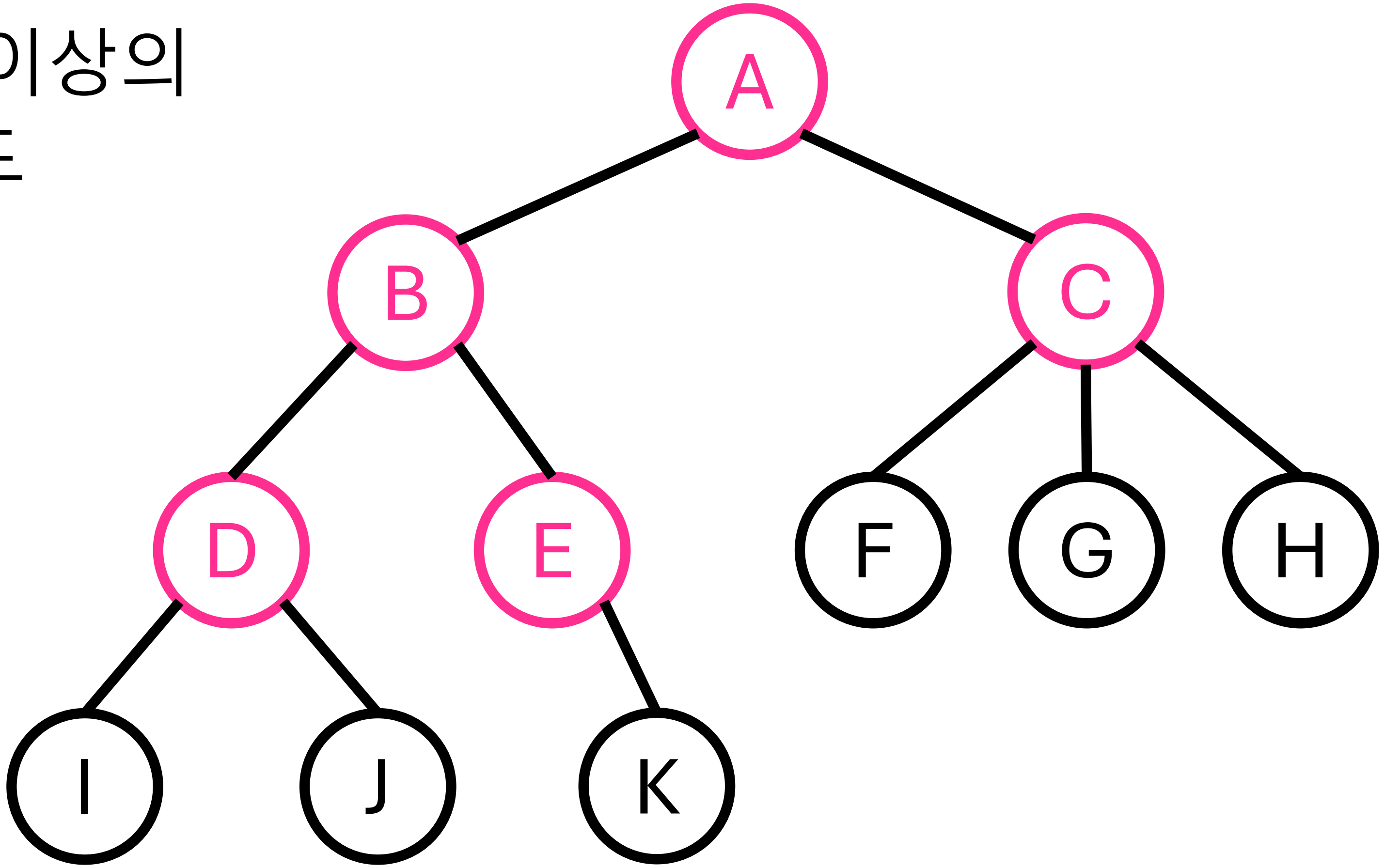
Tree의 용어

- root : 부모 노드가 없는 노드



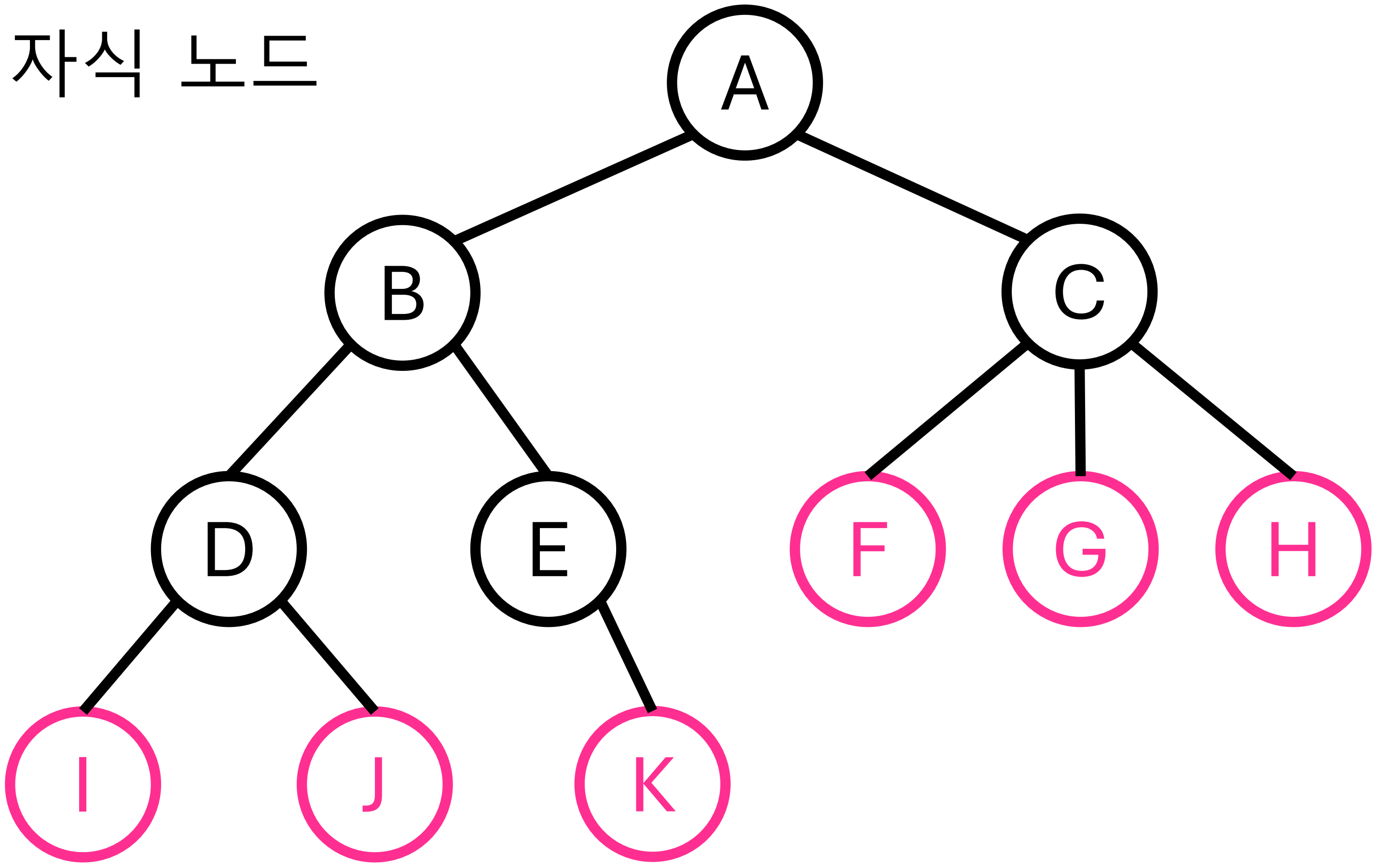
Tree의 용어

- internal node : 하나 이상의 자식 노드를 가진 노드



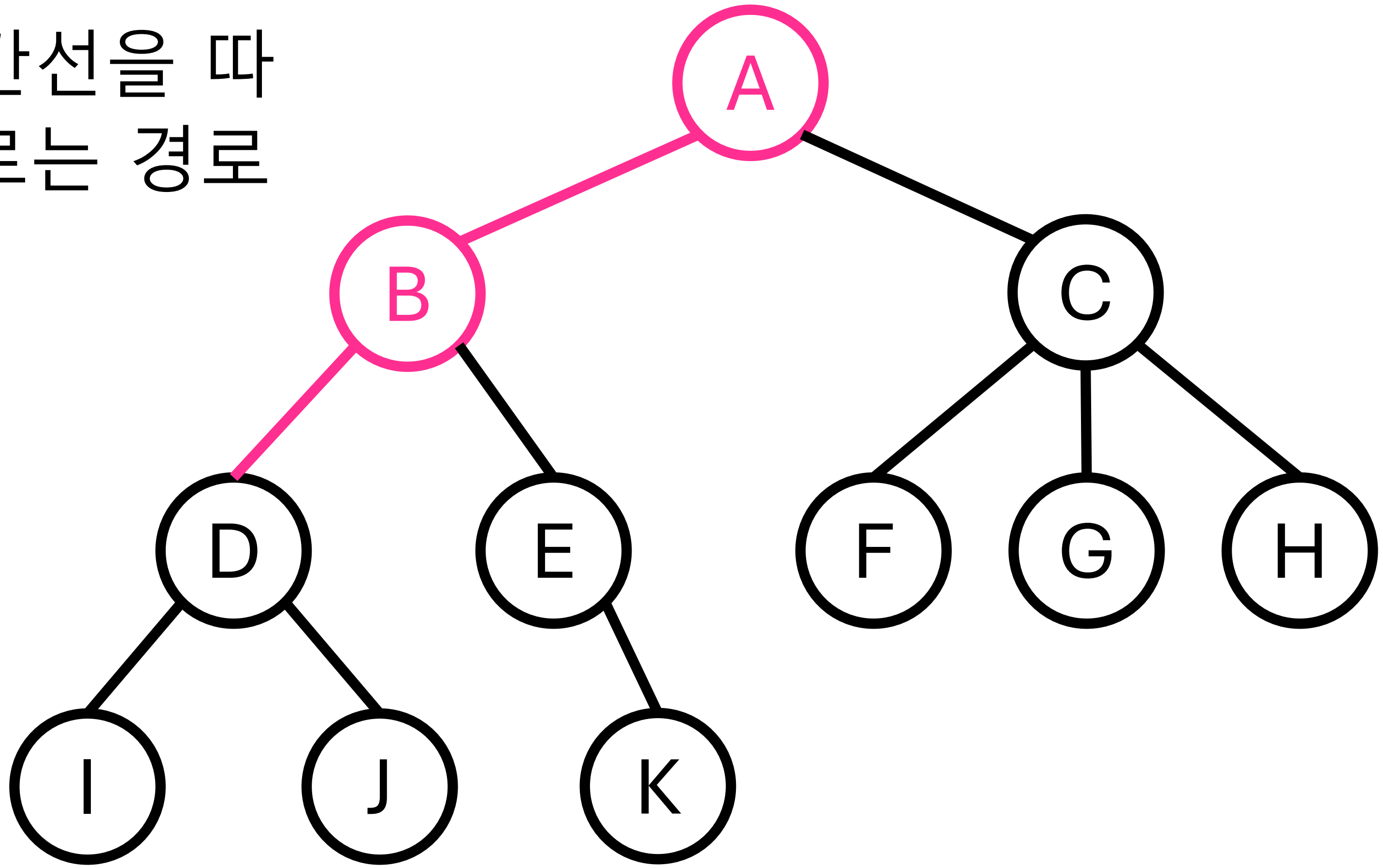
Tree의 용어

- external node (leaf) : 자식 노드가 없는 노드



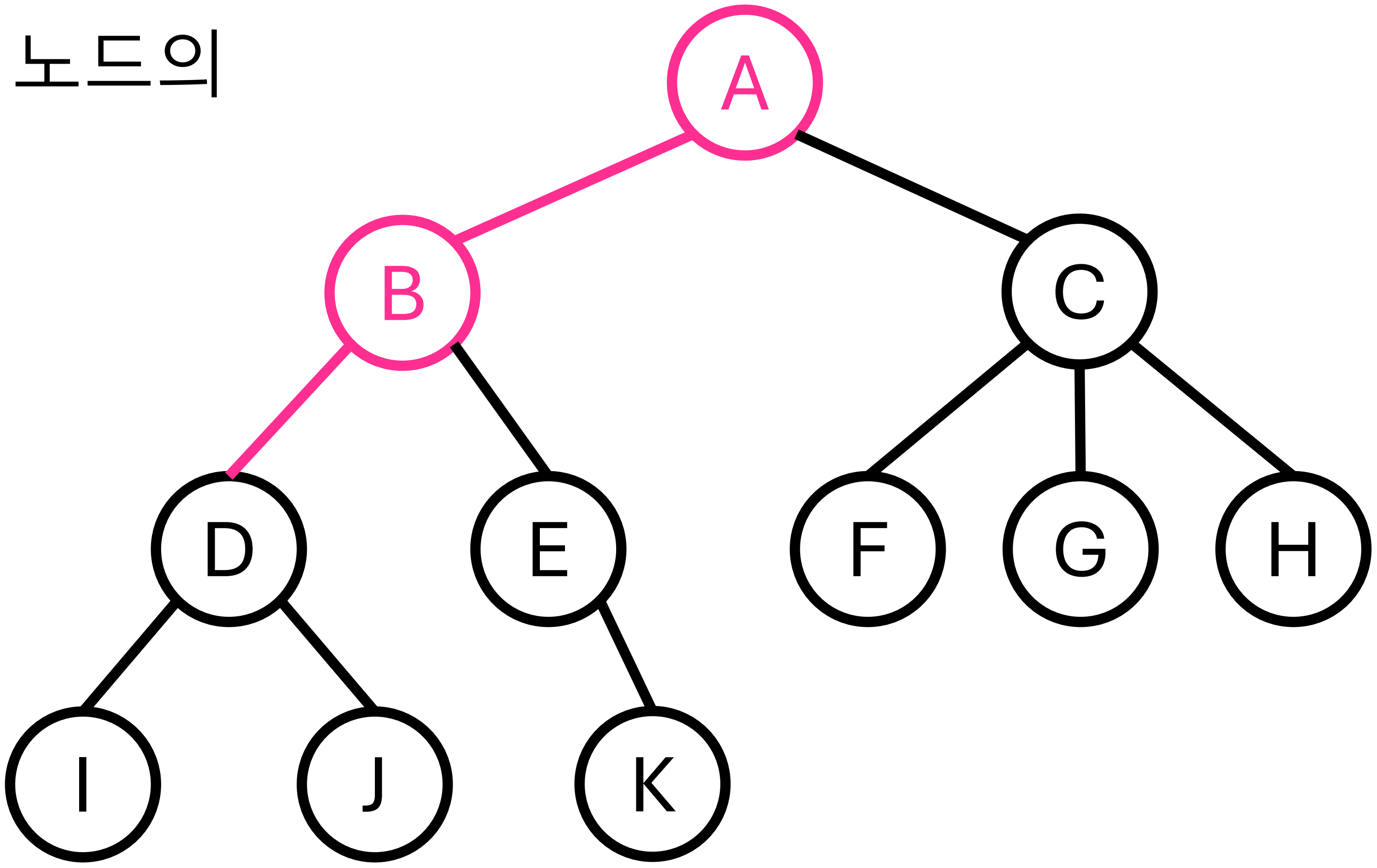
Tree의 용어

- ancestors of node : 간선을 따라 루트 노드까지 이르는 경로에 있는 모든 노드
- ancestors of D : A, B



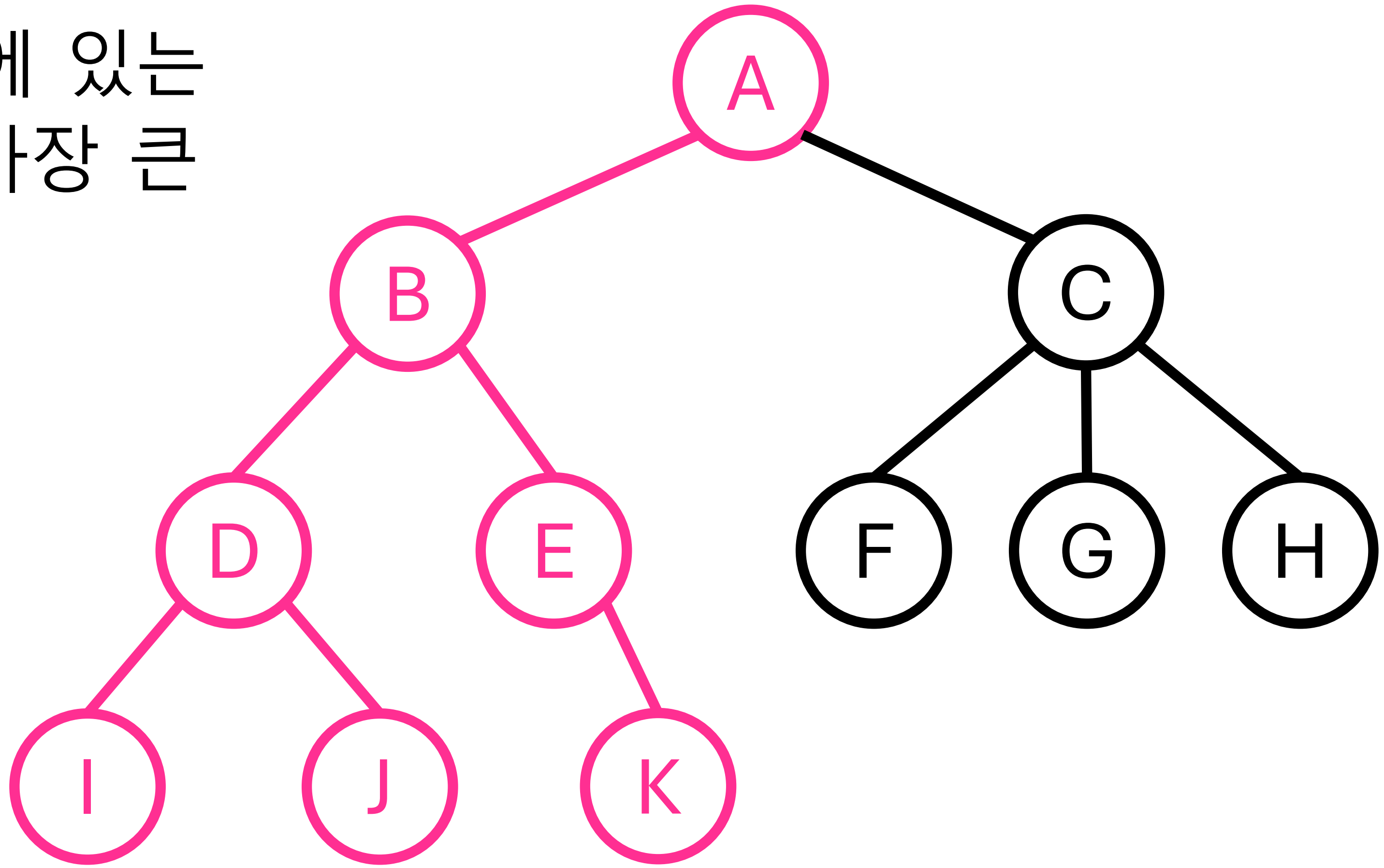
Tree의 용어

- depth of node : 조상 노드의 개수
- ex) depth of D : 2



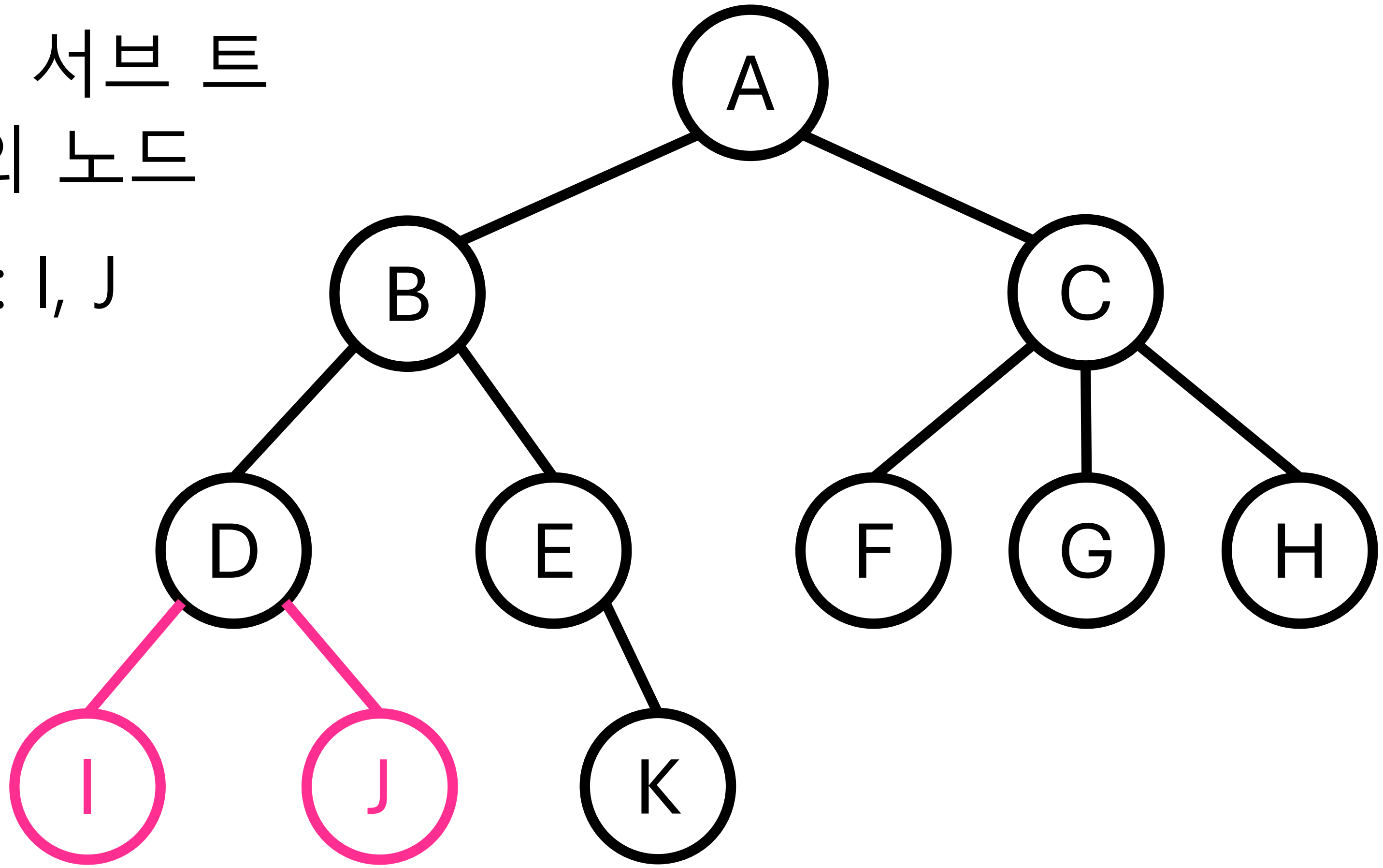
Tree의 용어

- height of tree : 트리에 있는 노드의 높이 중에서 가장 큰 값



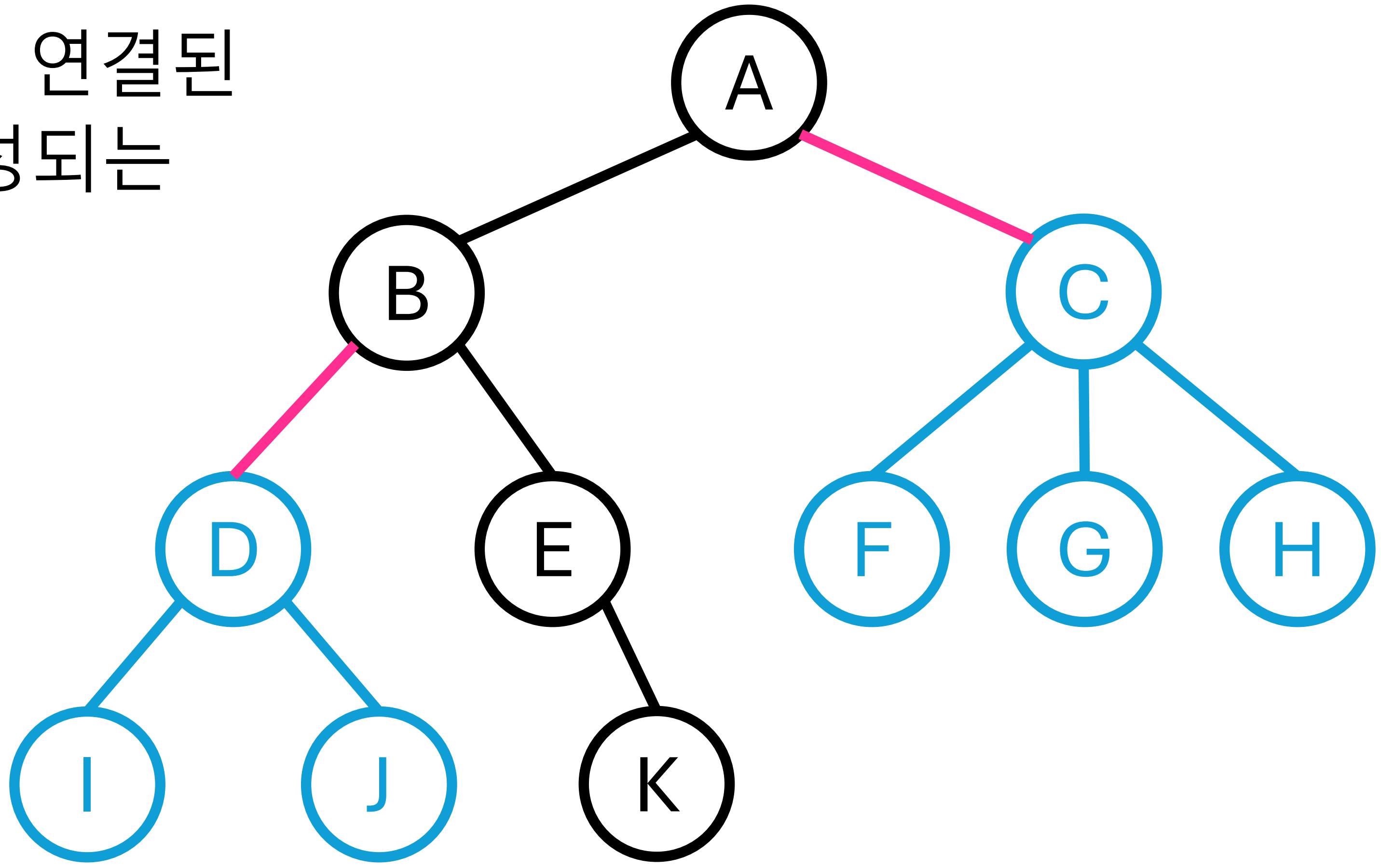
Tree의 용어

- descendant of node : 서브 트리에 있는 하위 레벨의 노드
- ex) descendant of D : I, J



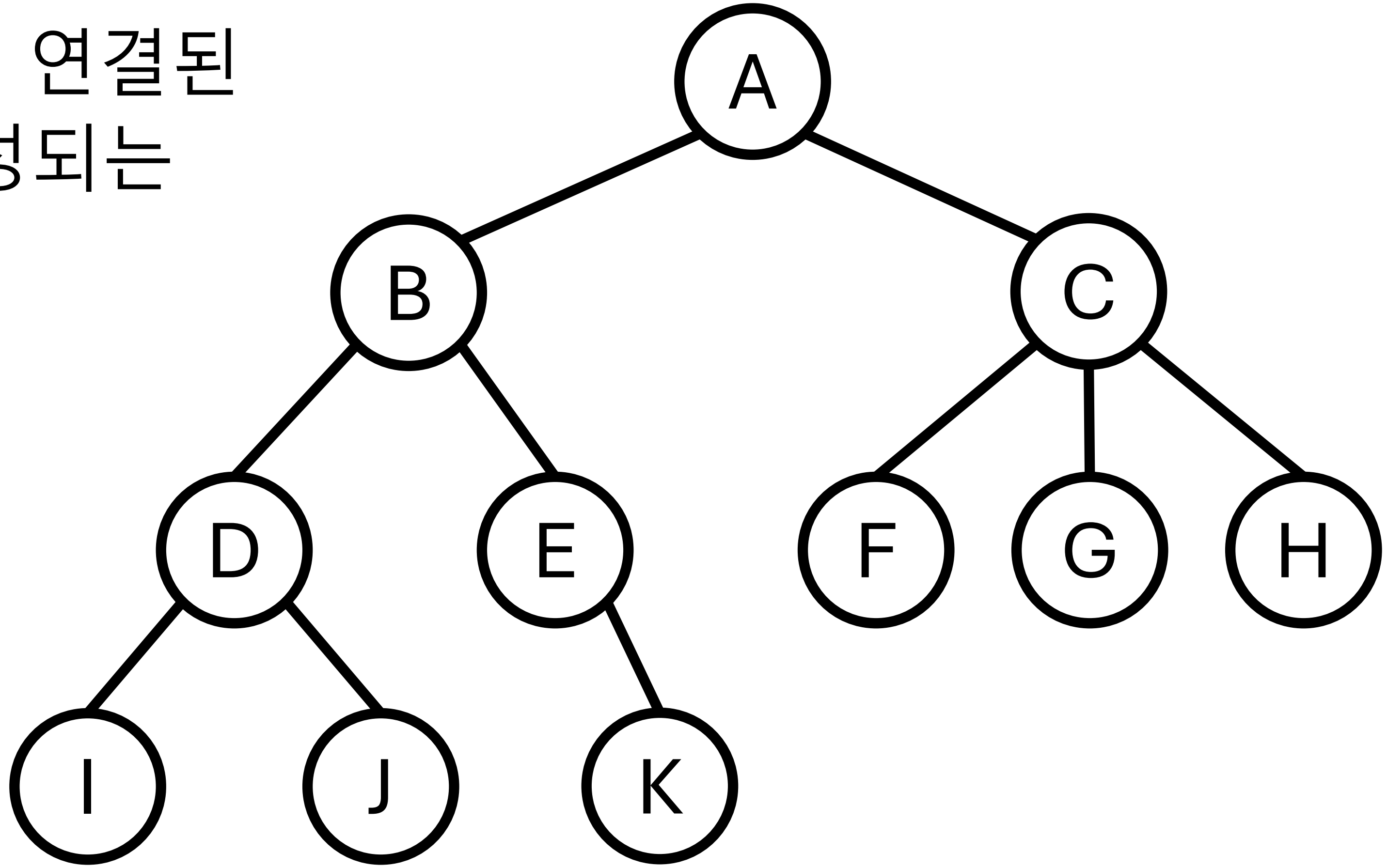
Tree의 용어

- subtree : 부모 노드와 연결된 간선을 끊었을 때 생성되는 트리



Tree의 구현

- subtree : 부모 노드와 연결된 간선을 끊었을 때 생성되는 트리

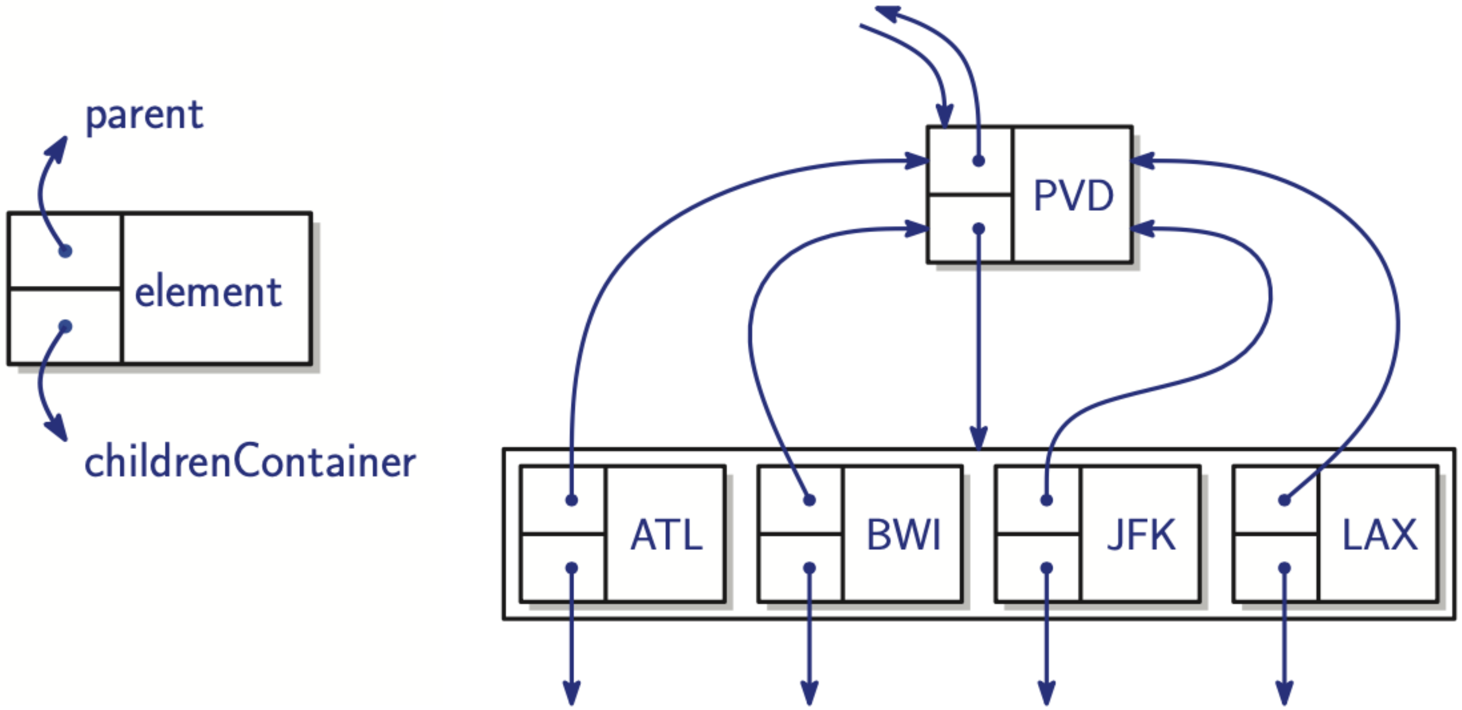


Tree의 ADT

- int size() : 노드의 개수를 반환
- bool empty() : 트리가 비었는지 반환
- Position root() : 루트 노드를 반환
- PositionList positions : 트리에 있는 모든 노드들을 반환
- Position p.parent() : p의 부모 노드를 반환
- PositionList p.children() : p의 자식 노드 리스트를 반환
- bool p.isRoot() : p가 루트 노드인지 반환
- bool p.isExternal() : p가 리프 노드인지 반환

Tree의 구현

- 노드가 가지는 정보 : 노드가 가지는 값, 부모 노드의 포인터, 자식 노드 리스트



Position Class

```
class PositionList;
template <typename T>
class Position {
private:
	T element;
	Position<T> *parent;
	PositionList children;
```

Position Class

```cpp
class PositionList;
template <typename T>
class Position {
private:
    T element;
    Position<T> *parent;
    PositionList children;
```

Position Class

```cpp
T& operator*() {
    return element;
}
Position<T> parent() const {
    return *parent;
}
PositionList children() const {
    return children;
}
```

```cpp
bool isRoot() const {
    return parent == nullptr;
}
bool isExternal() const {
    return children.empty();
}
```

Tree Class

```cpp
template <typename T>
class Tree {
private:
    PositionList _positions;
```

```cpp
int size() const {
    return _positions.size();
}

bool empty() const {
    return size() == 0;
}
```

Tree Class

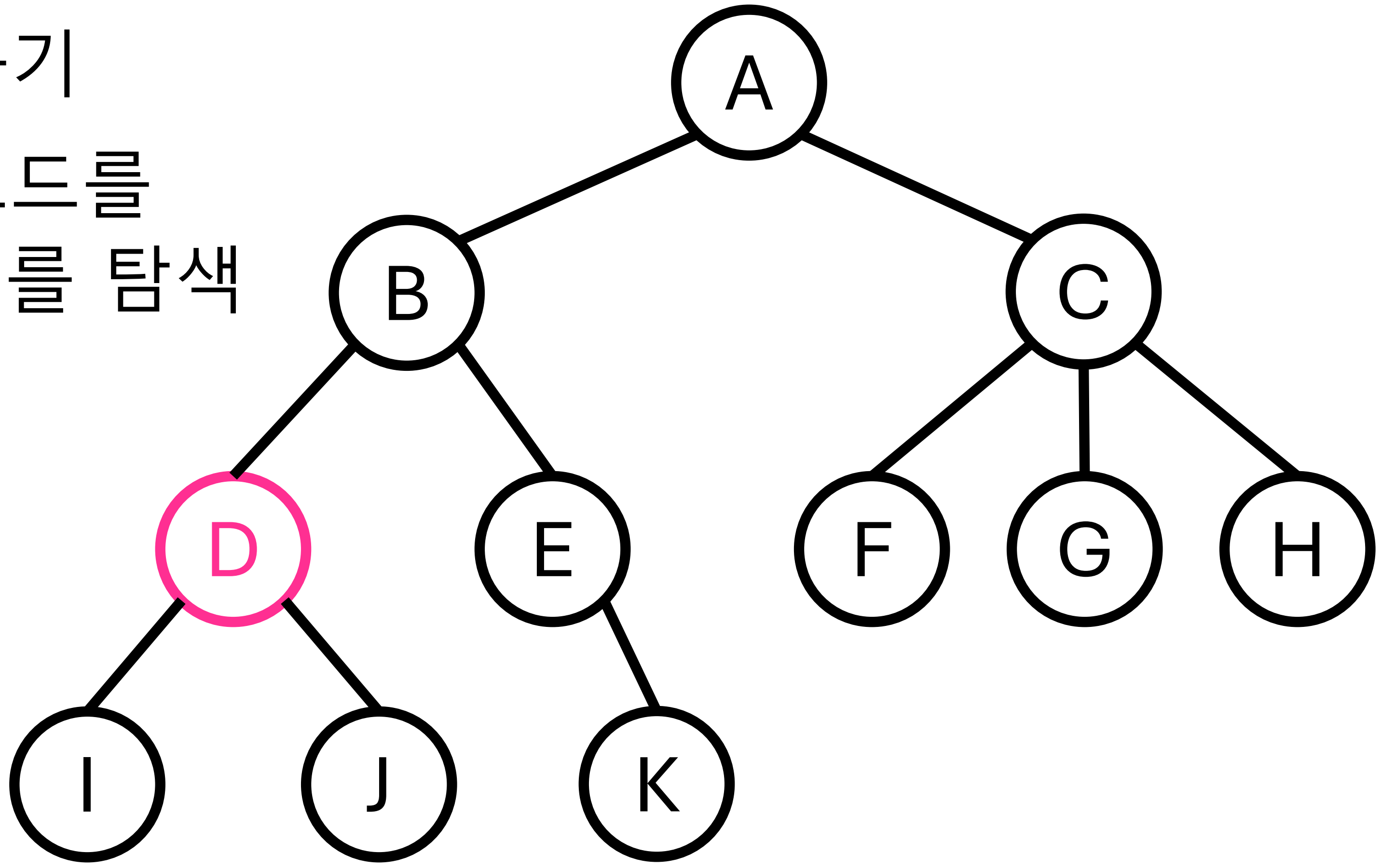
```cpp
Position<T> root() const {
    for (Position<T> p : _positions) {
        if (p.isRoot()) {
            return p;
        }
    }

    throw "Can not find root";
}
```

```cpp
PositionList positions() {
    return _positions;
}
```

Tree Traversal

Depth of a node

- 노드 D의 높이를 구하기
- D부터 시작해 루트 노드를
  만나기까지 부모 노드를 탐색

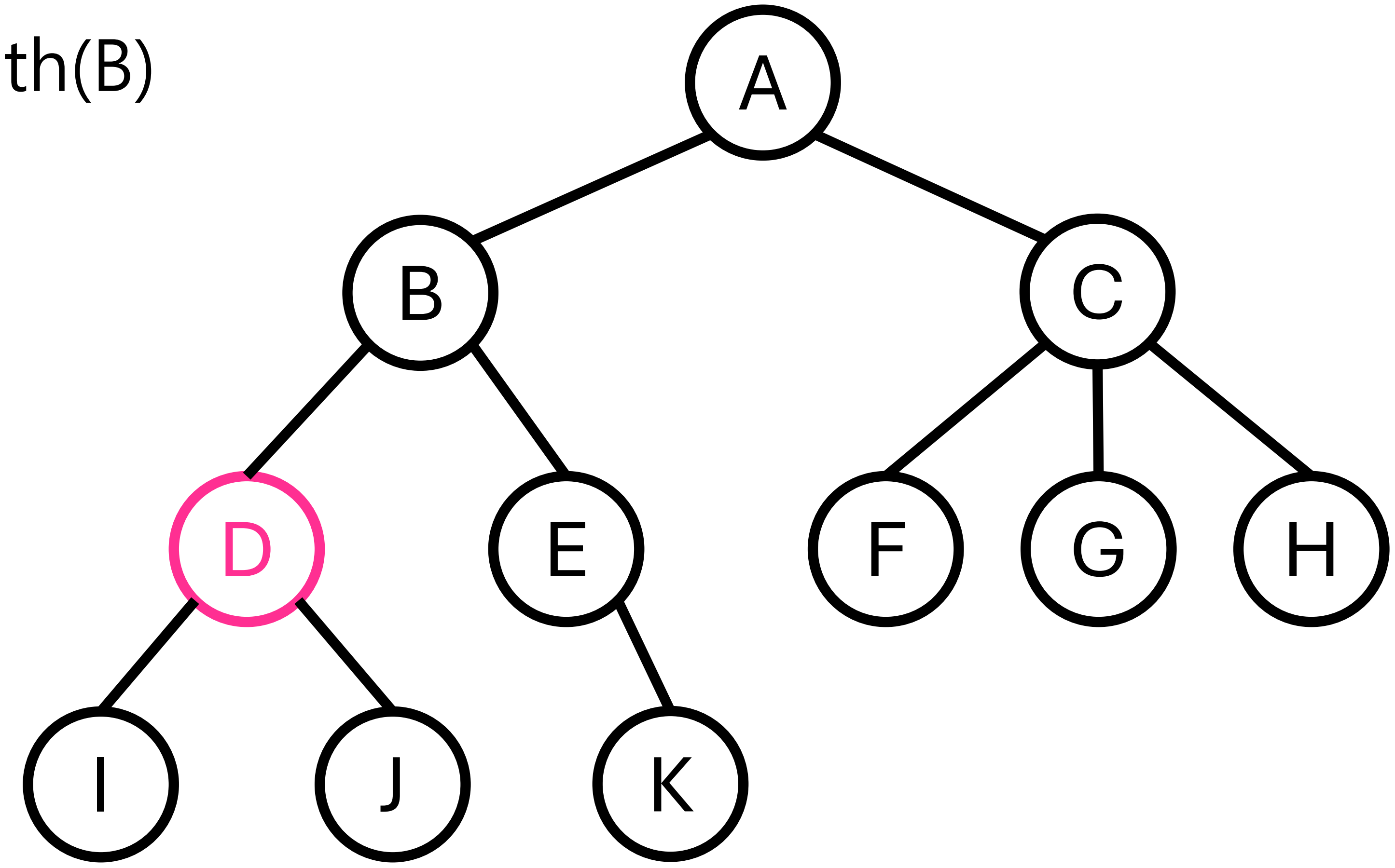


Depth of a node

```
int depth(const Tree<int> &t, const Position<int> &p) {
    if (p.isRoot())
        return 0;
    else
        return 1 + depth(t, p.parent());
}
```

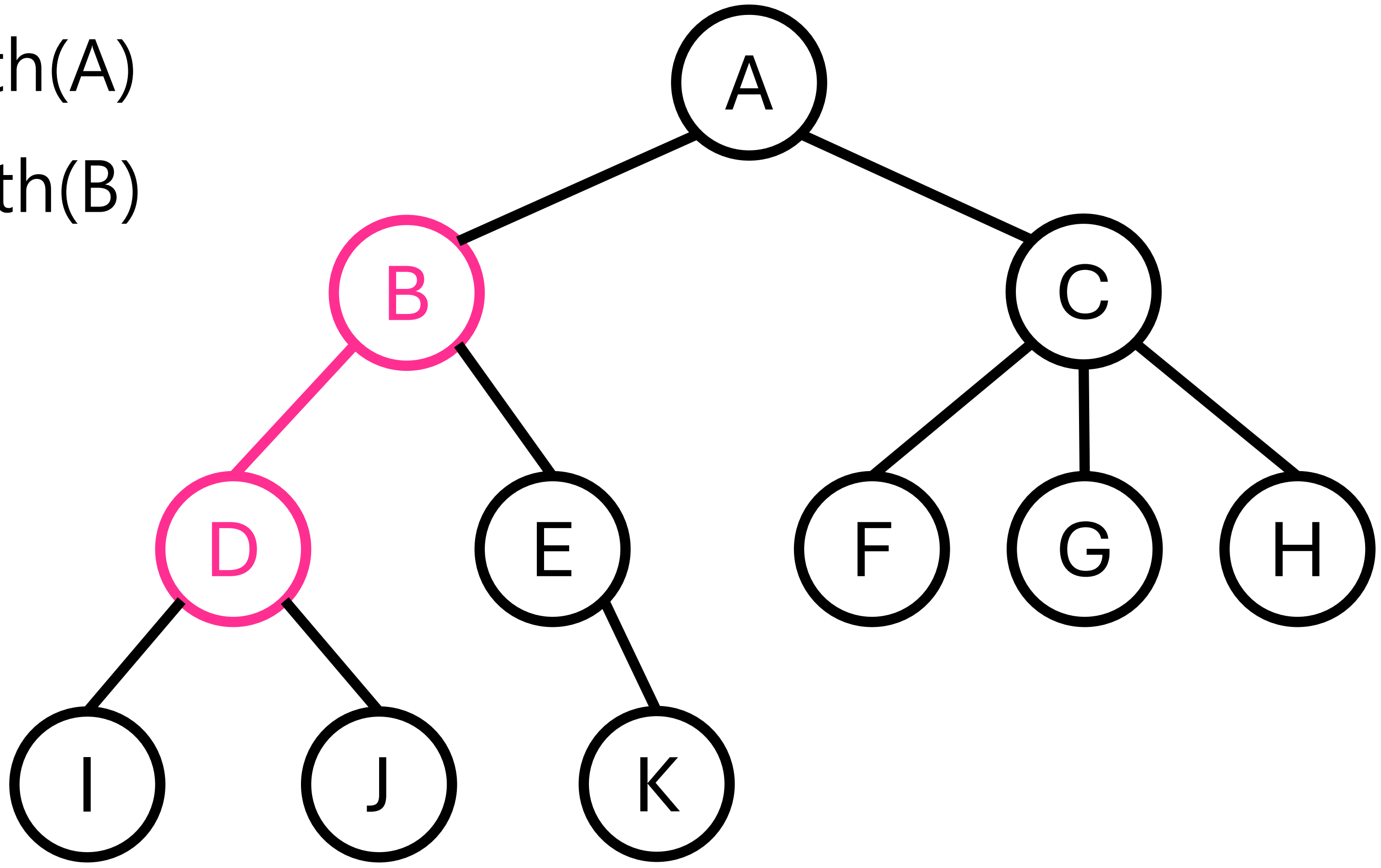
Depth of a node

- depth(D) -> 1 + depth(B)



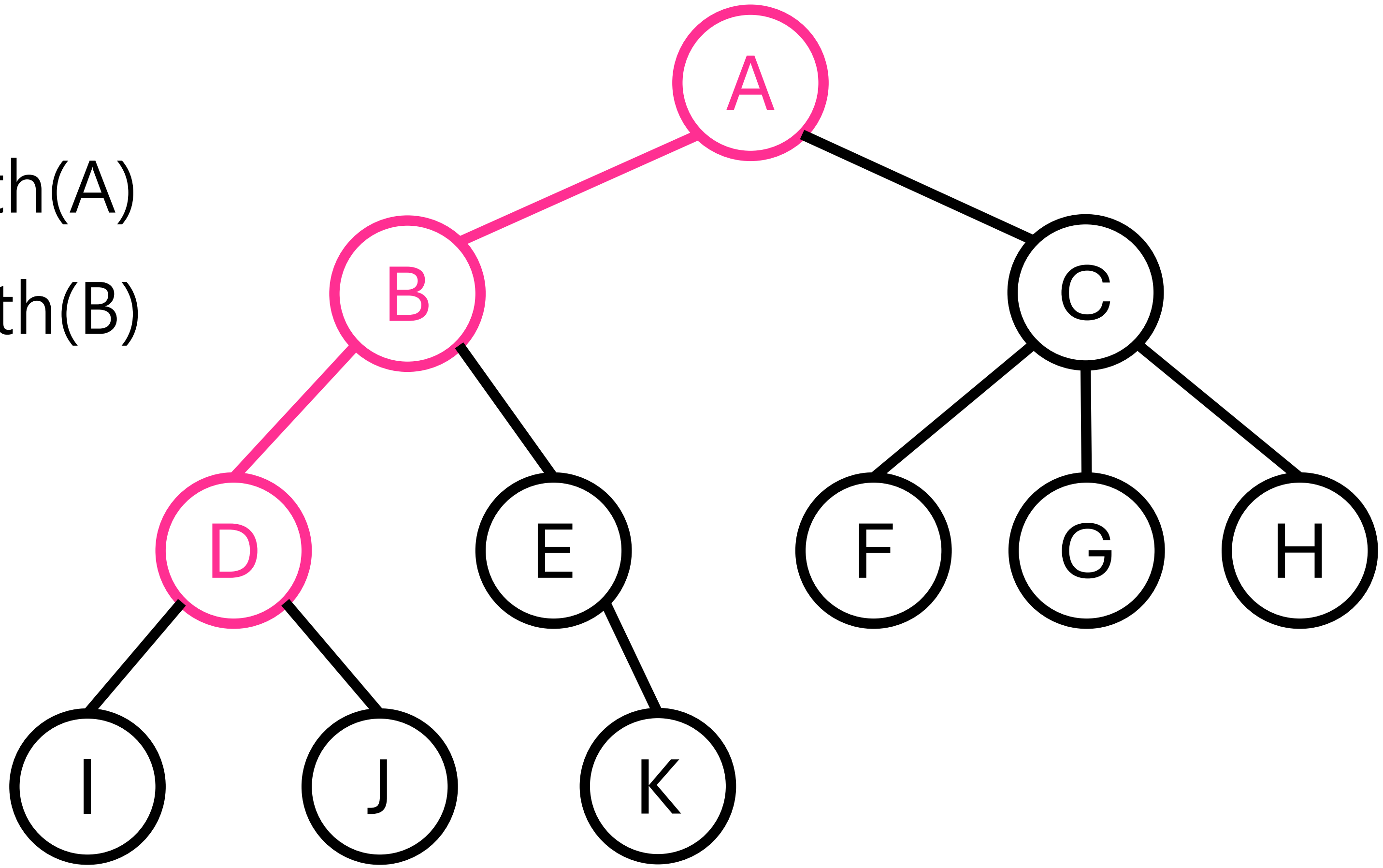
Depth of a node

- depth(B) -> 1 + depth(A)
- depth(D) -> 1 + depth(B)



Depth of a node

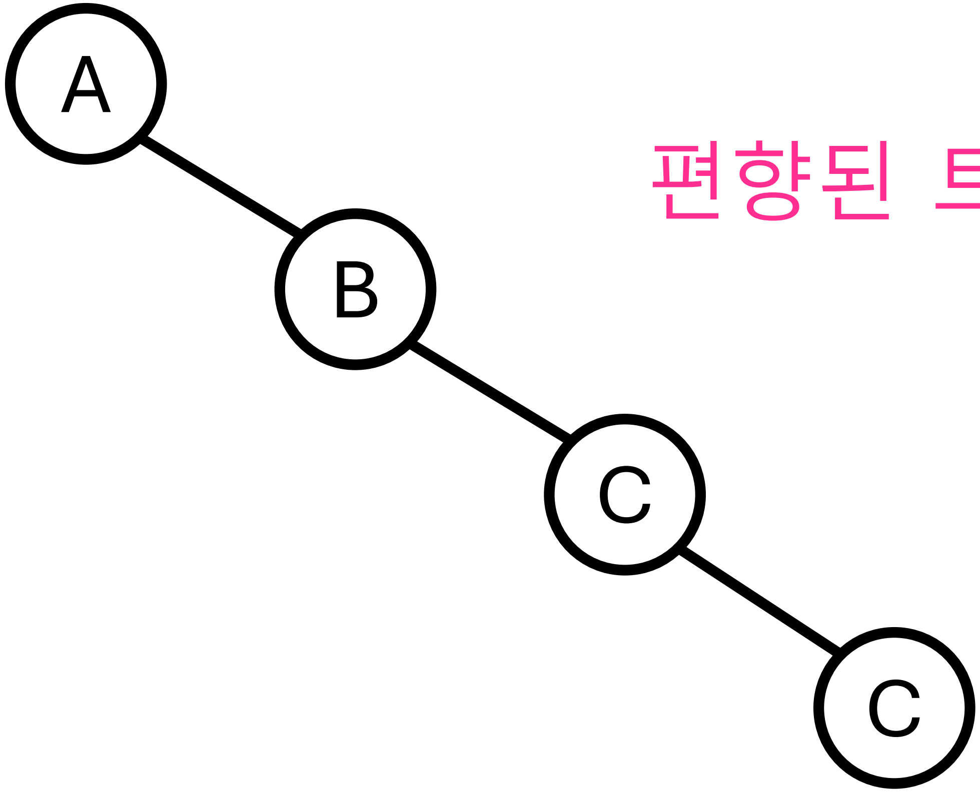
- depth(A) -> 0
- depth(B) -> 1 + depth(A)
- depth(D) -> 1 + depth(B)



depth 함수의 복잡도

- 최악의 경우 분석
  - 단위연산 : 노드가 루트 노드인지 확인
  - 입력크기 : 노드의 개수 n
  - 최악의 경우 트리가 편향된 경우(트리가 한쪽으로 치우쳐짐)에서 리프 노드의 depth를 계산하면 모든 노드에 대해서 함수가 호출
  - $W(n) = n$

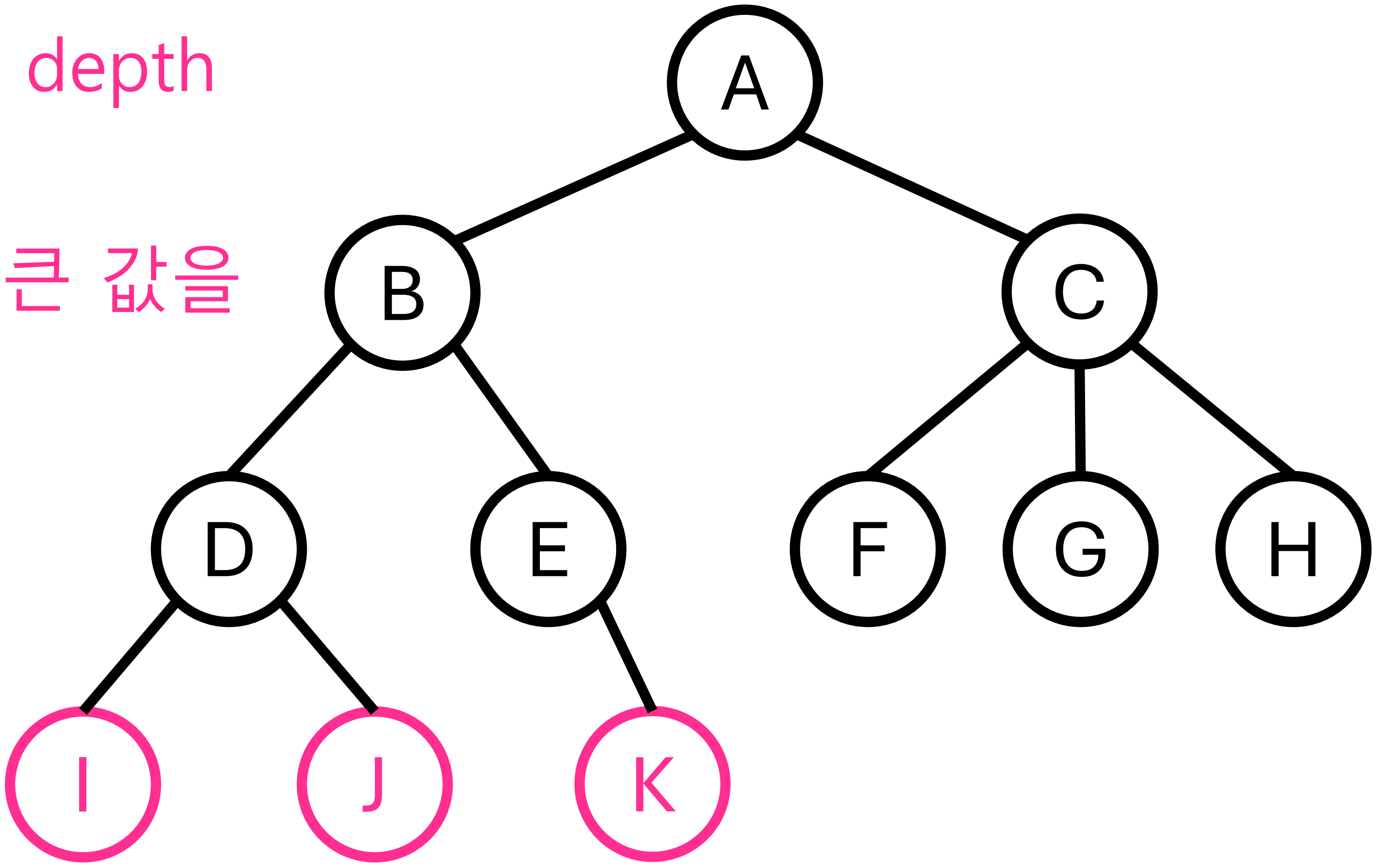
depth 함수의 복잡도



편향된 트리

Height of a tree

- 트리에서 모든 노드의 depth 를 계산
- 노드의 depth중 가장 큰 값을 height로 계산



Height of a tree

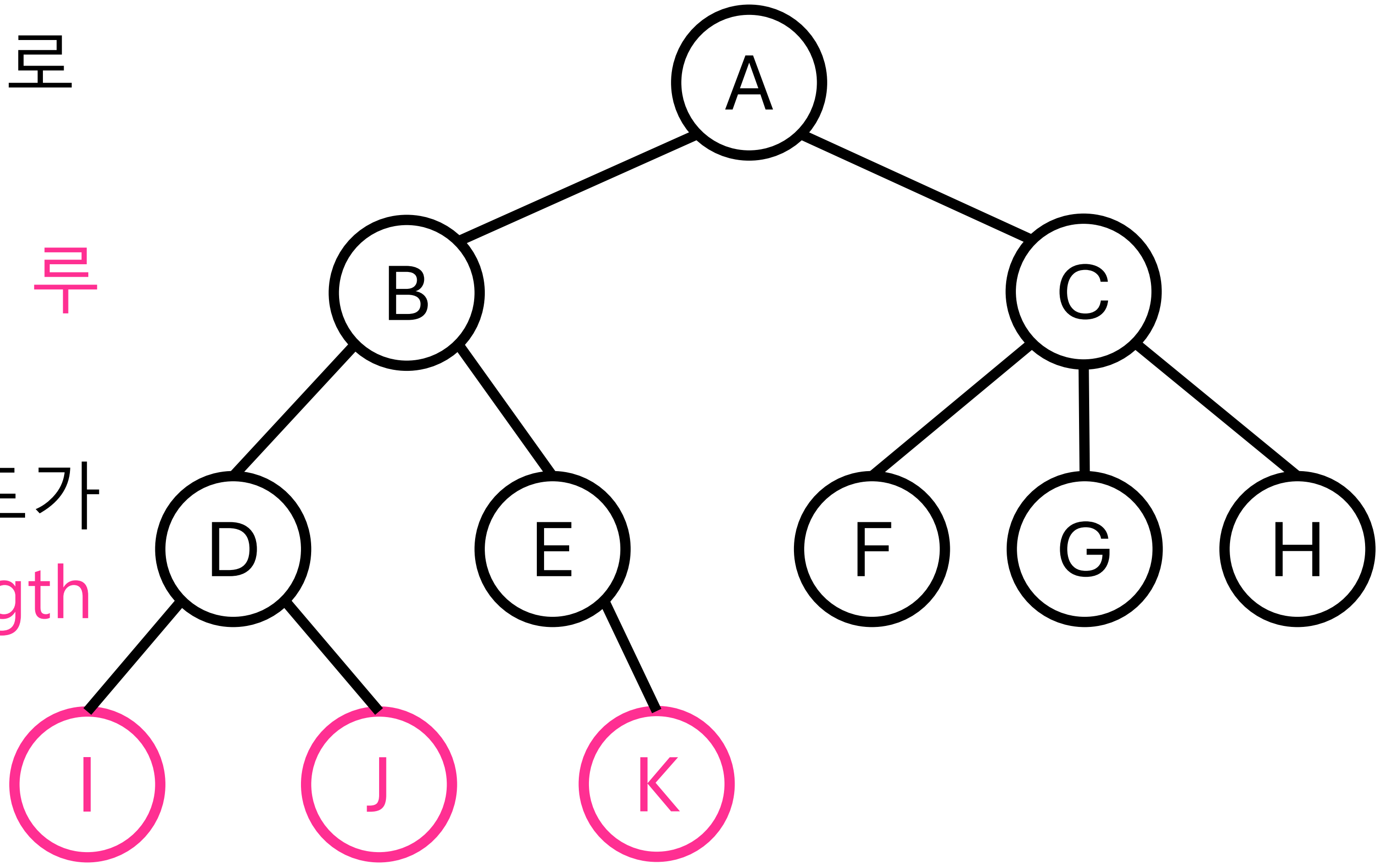
```cpp
int height1(const Tree<int> &t) {
    int h = 0;
    PositionList nodes = t.positions();
    for (auto q = nodes.begin(); q != nodes.end(); ++q) {
        if (q->isExternal()) {
            h = max(h, depth(t, q));
        }
    }
    return h;
}
```

height1 함수의 복잡도

- 최악의 경우 분석
  - 단위연산 : depth 함수 호출
  - 입력크기 : 노드의 개수 n
  - depth 함수의 최악의 경우 복잡도는 n이고, depth 함수는 모든 노드에 대해서 호출
  - W(n) = n^2

개선된 height 함수

- 노드에 대해 재귀적으로 계산
- 최초 호출되는 노드는 루트 노드
- 해당 노드가 루트 노드가 되는 서브 트리의 heigth를 계산

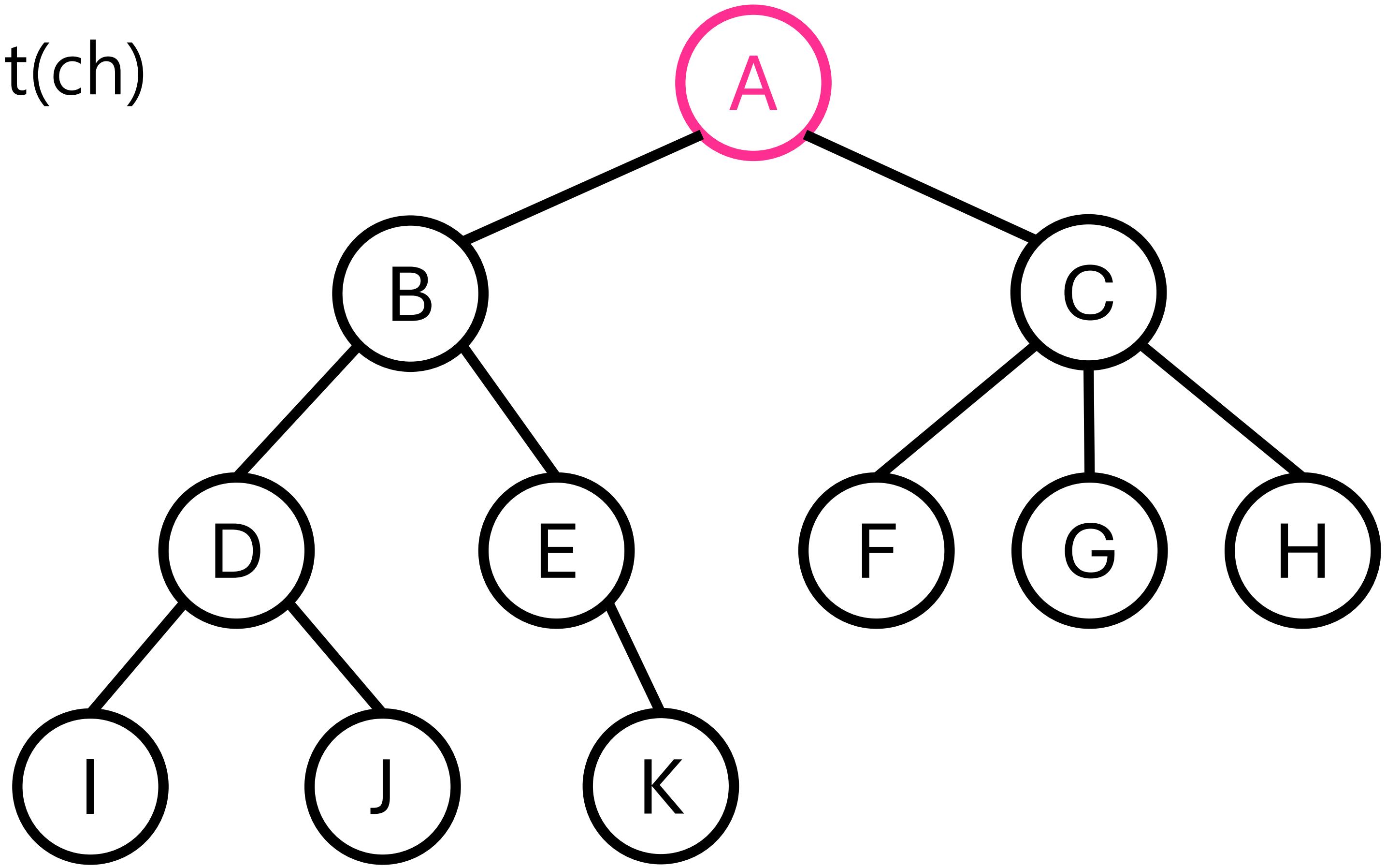


개선된 height 함수

```cpp
int height2(const Tree<int> &t, const Position<int> &p) {
    if (p.isExternal())
        return 0;

    int h = 0;
    PositionList ch = p.children();
    for (auto q = ch.begin(); q != ch.end(); ++q)
        h = max(h, height2(t, *q));
    return 1 + h;
}
```

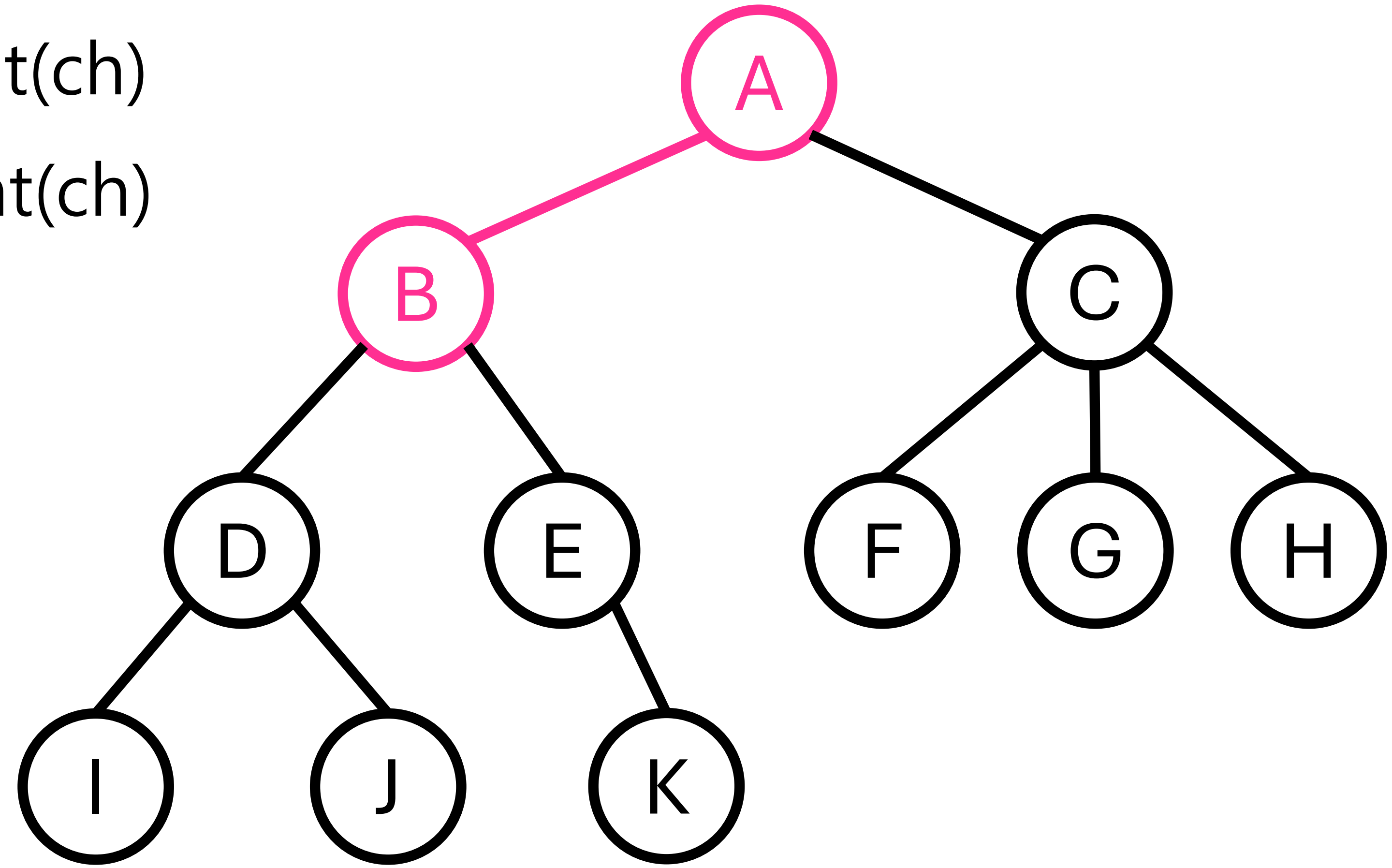
개선된 height 함수

- height(A) -> 1 + height(ch)



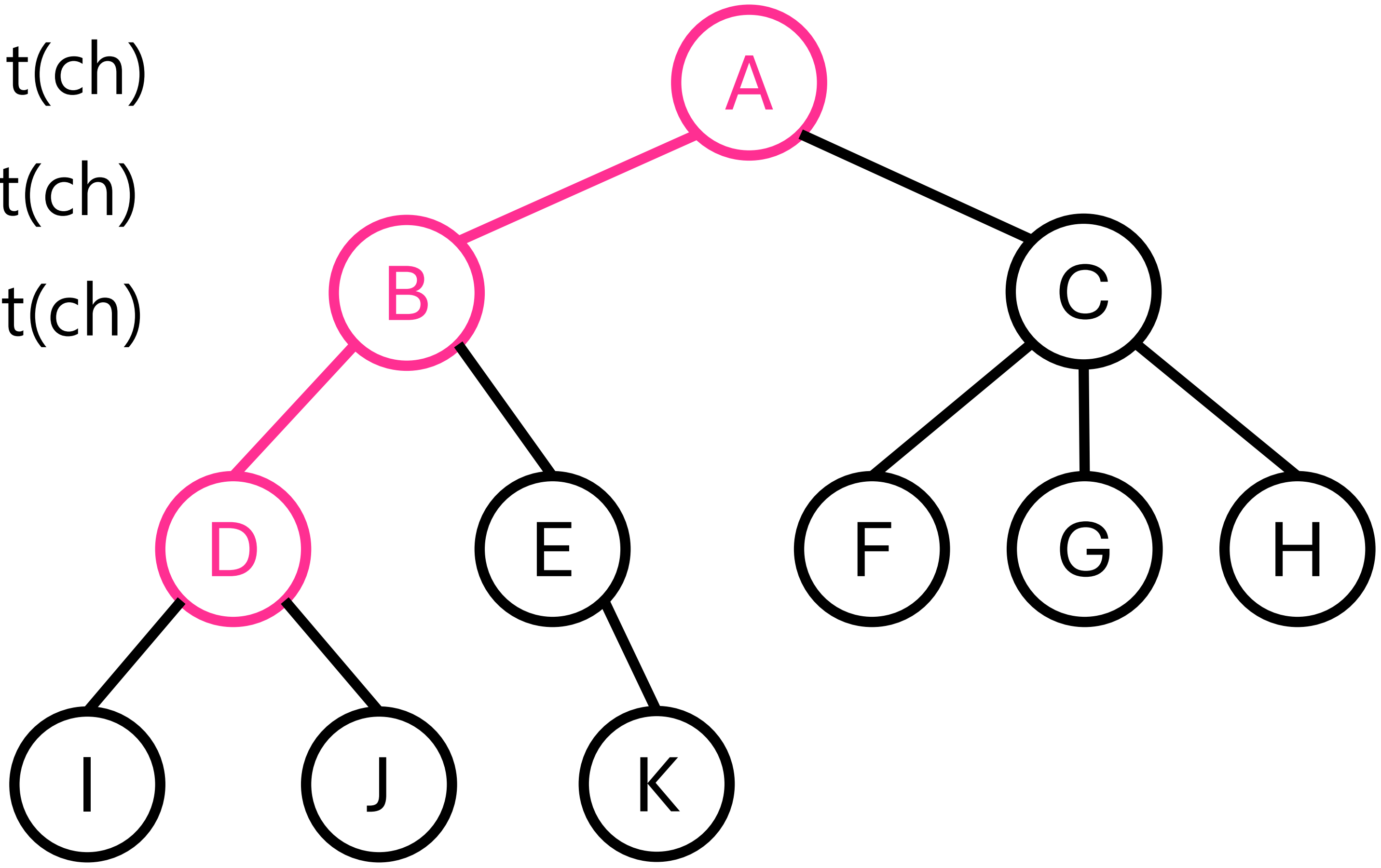
개선된 height 함수

- height(B) -> 1 + height(ch)
- height(A) -> 1 + height(ch)



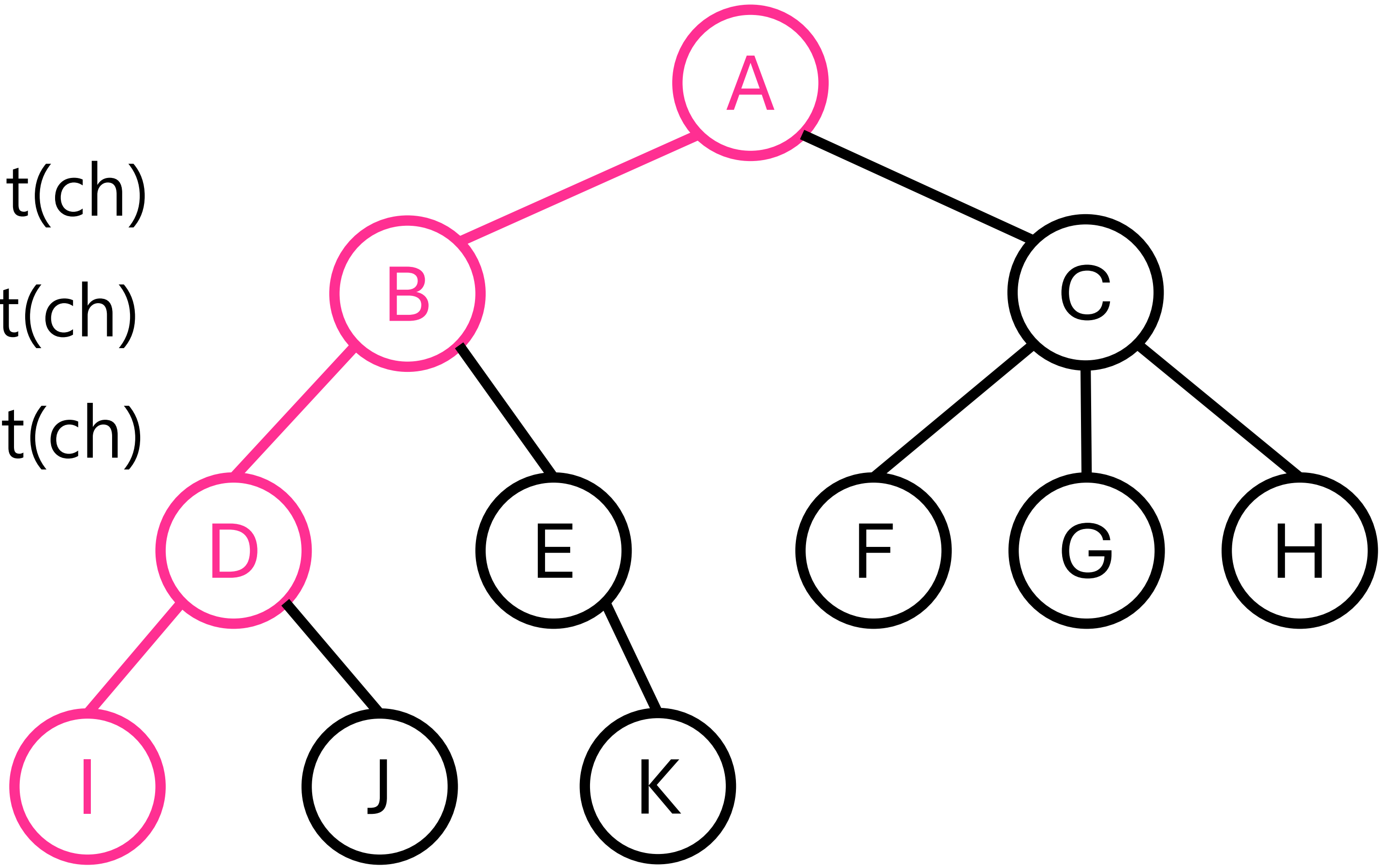
개선된 height 함수

- height(D) -> 1 + height(ch)
- height(B) -> 1 + height(ch)
- height(A) -> 1 + height(ch)



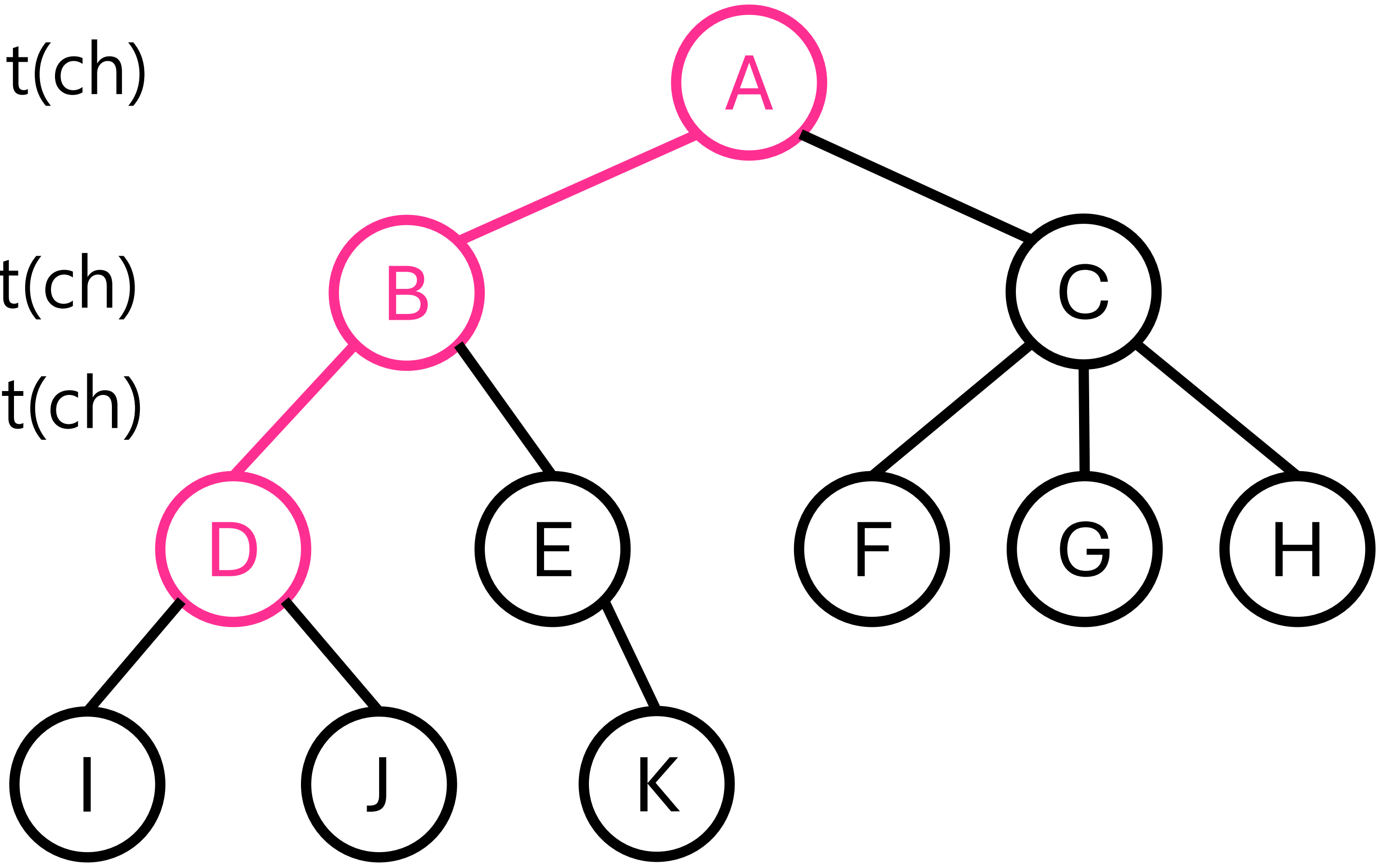
개선된 height 함수

- height(I) -> 0
- height(D) -> 1 + height(ch)
- height(B) -> 1 + height(ch)
- height(A) -> 1 + height(ch)



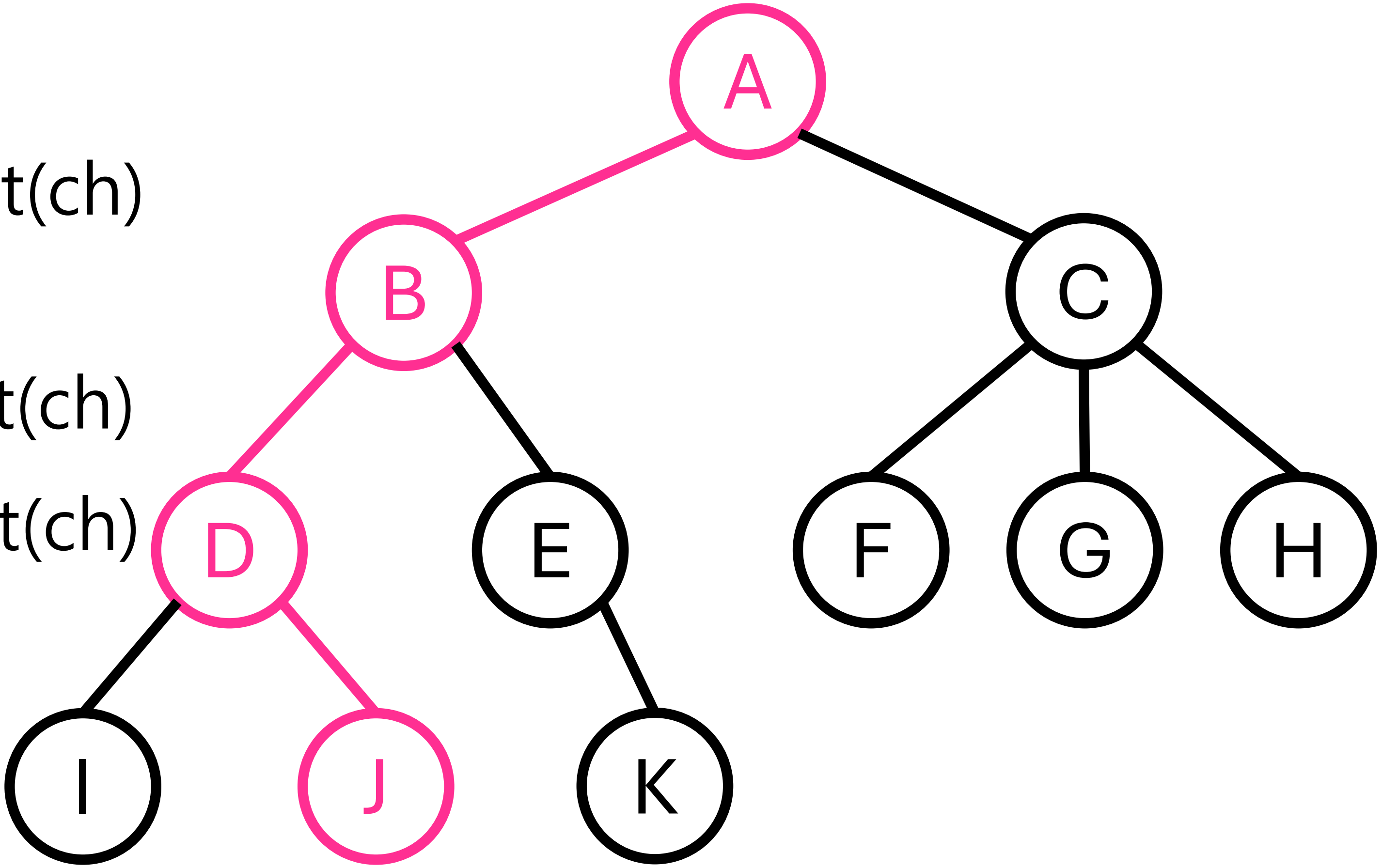
개선된 height 함수

- height(D) -> 1 + height(ch)
  - h = 1
- height(B) -> 1 + height(ch)
- height(A) -> 1 + height(ch)



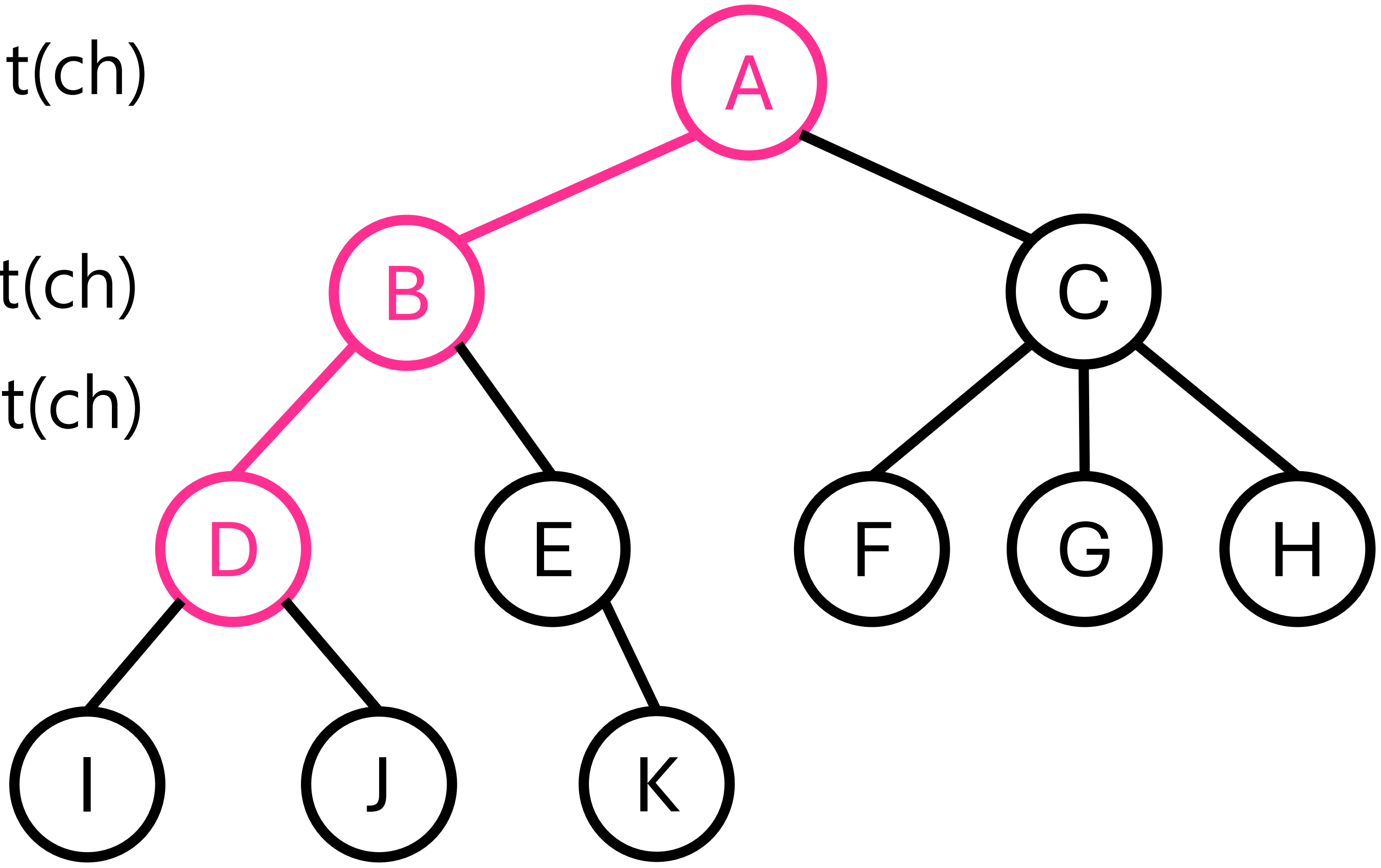
개선된 height 함수

- height(J) -> 0
- height(D) -> 1 + height(ch)
  - h = 1
- height(B) -> 1 + height(ch)
- height(A) -> 1 + height(ch)



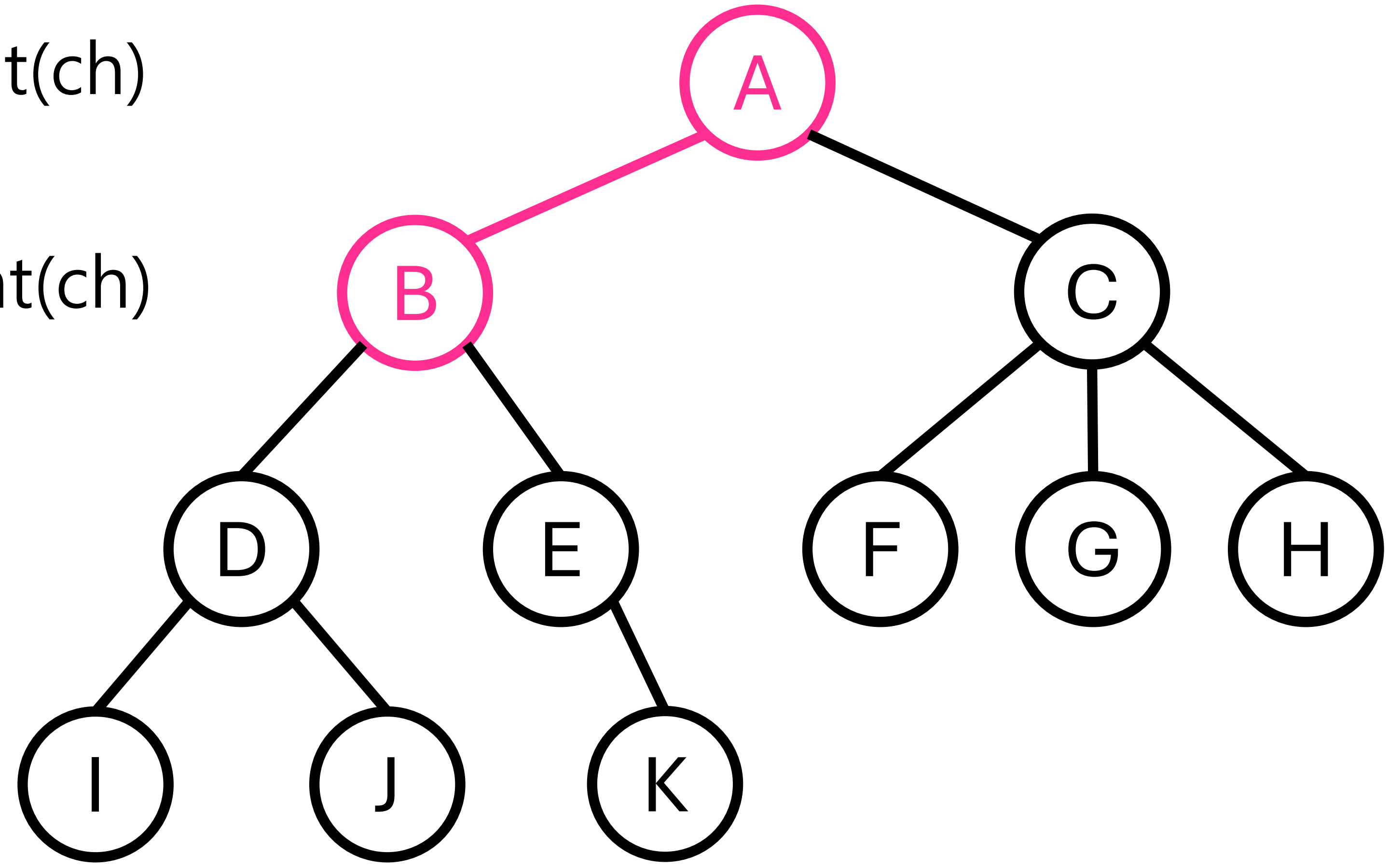
개선된 height 함수

- height(D) -> 1 + height(ch)
  - h = 1
- height(B) -> 1 + height(ch)
- height(A) -> 1 + height(ch)



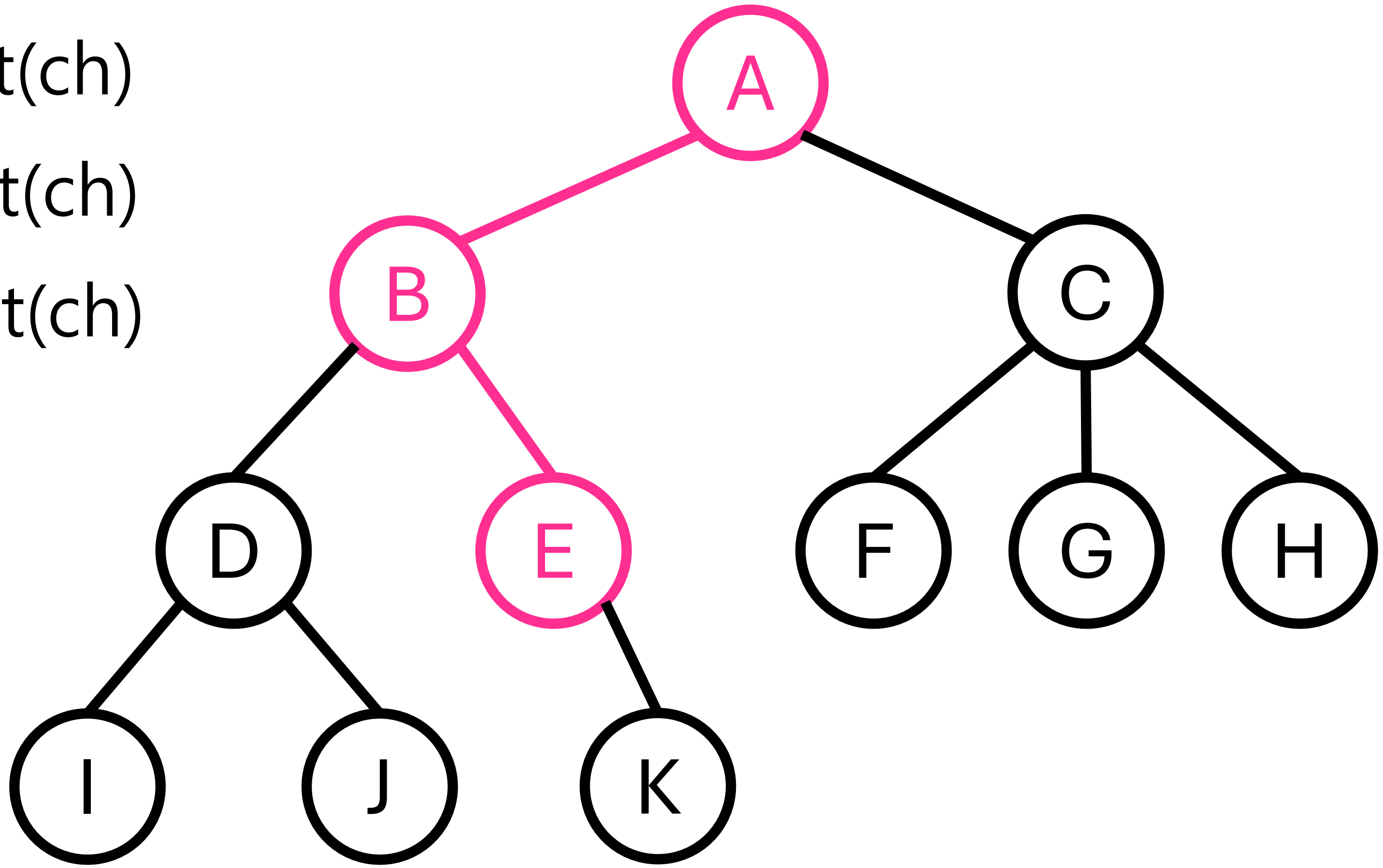
개선된 height 함수

- height(B) -> 1 + height(ch)
  - h = 2
- height(A) -> 1 + height(ch)



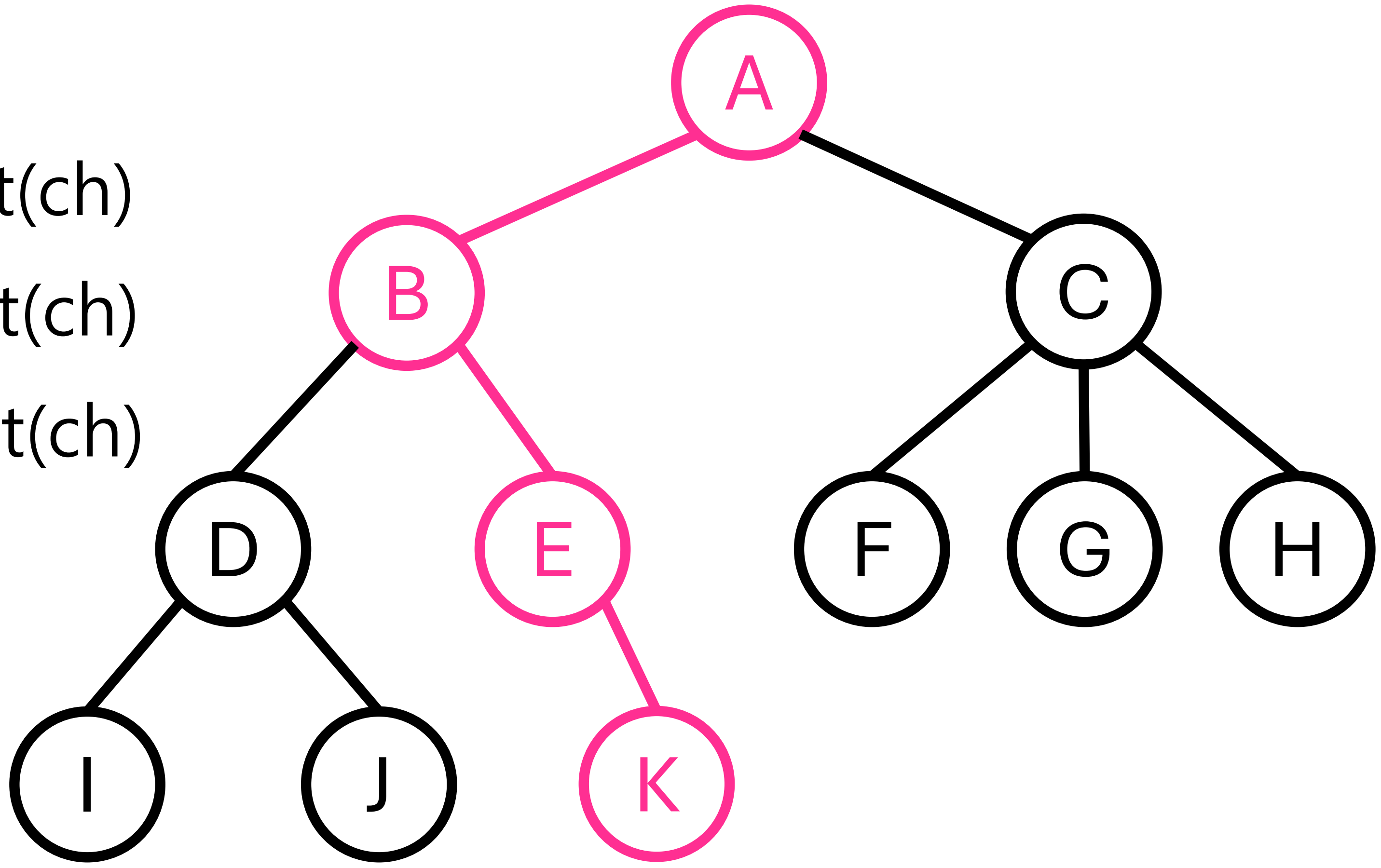
개선된 height 함수

- height(E) -> 1 + height(ch)
- height(B) -> 1 + height(ch)
- height(A) -> 1 + height(ch)



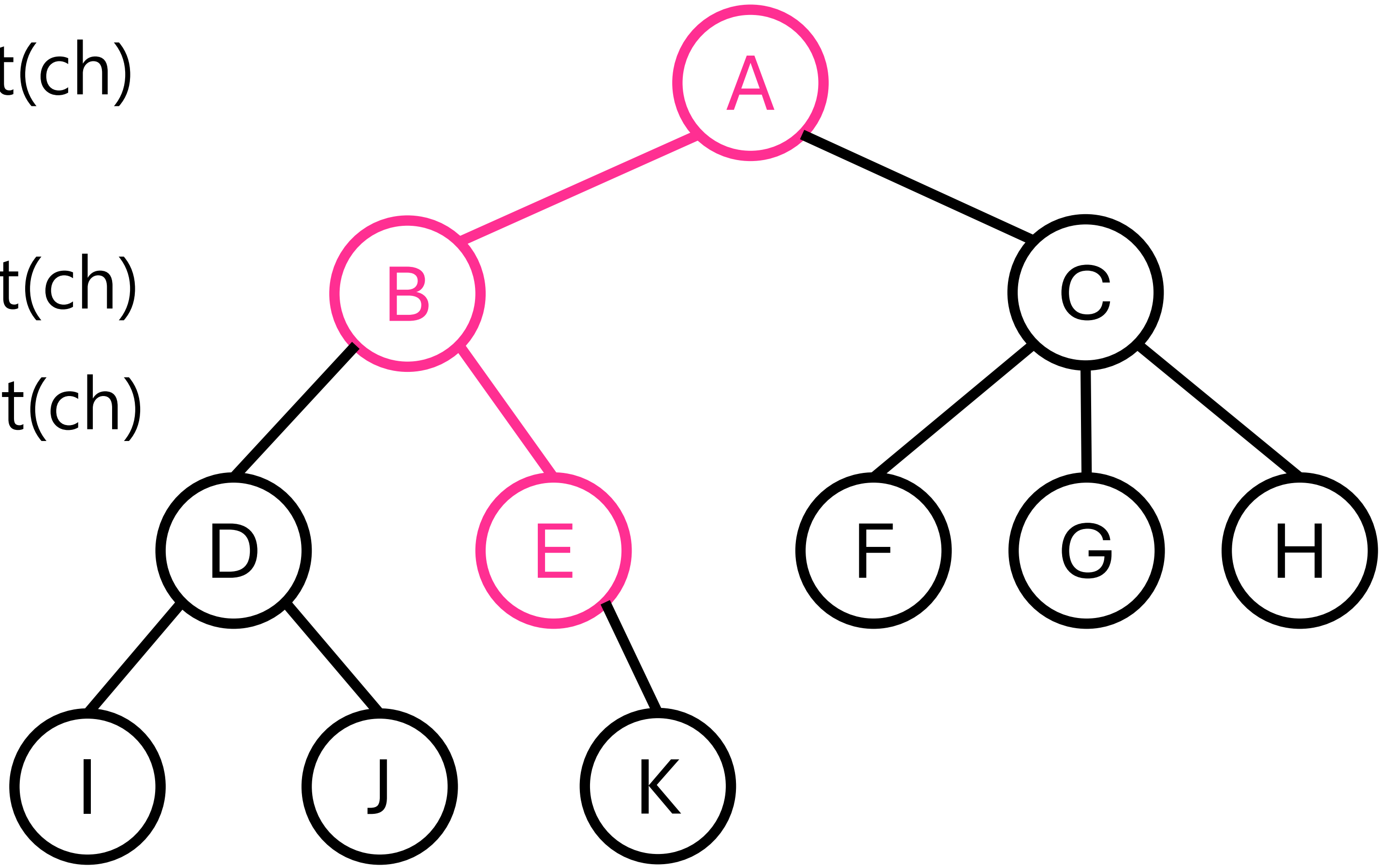
개선된 height 함수

- height(K) -> 0
- height(E) -> 1 + height(ch)
- height(B) -> 1 + height(ch)
- height(A) -> 1 + height(ch)



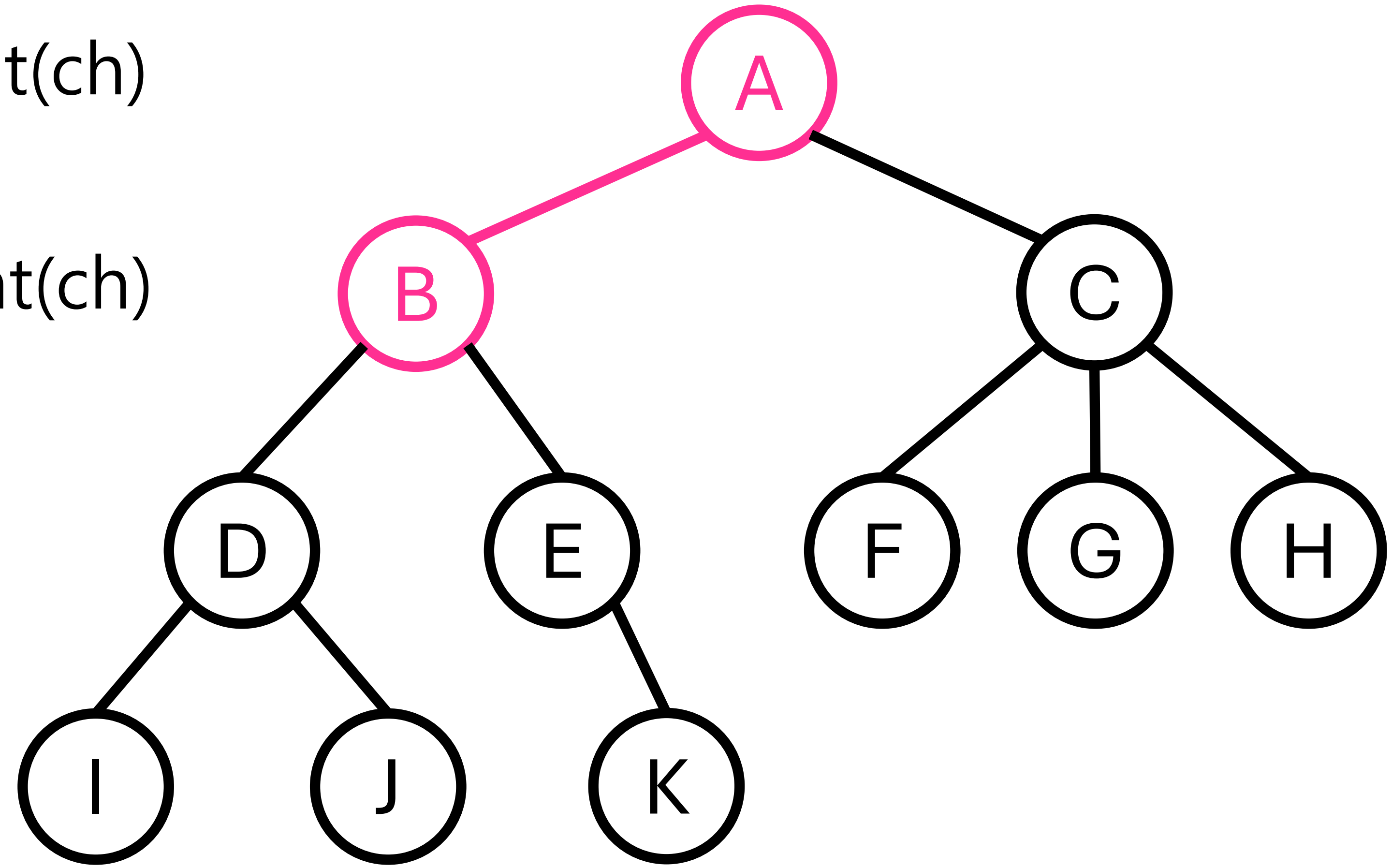
개선된 height 함수

- height(E) -> 1 + height(ch)
  - h = 1
- height(B) -> 1 + height(ch)
- height(A) -> 1 + height(ch)



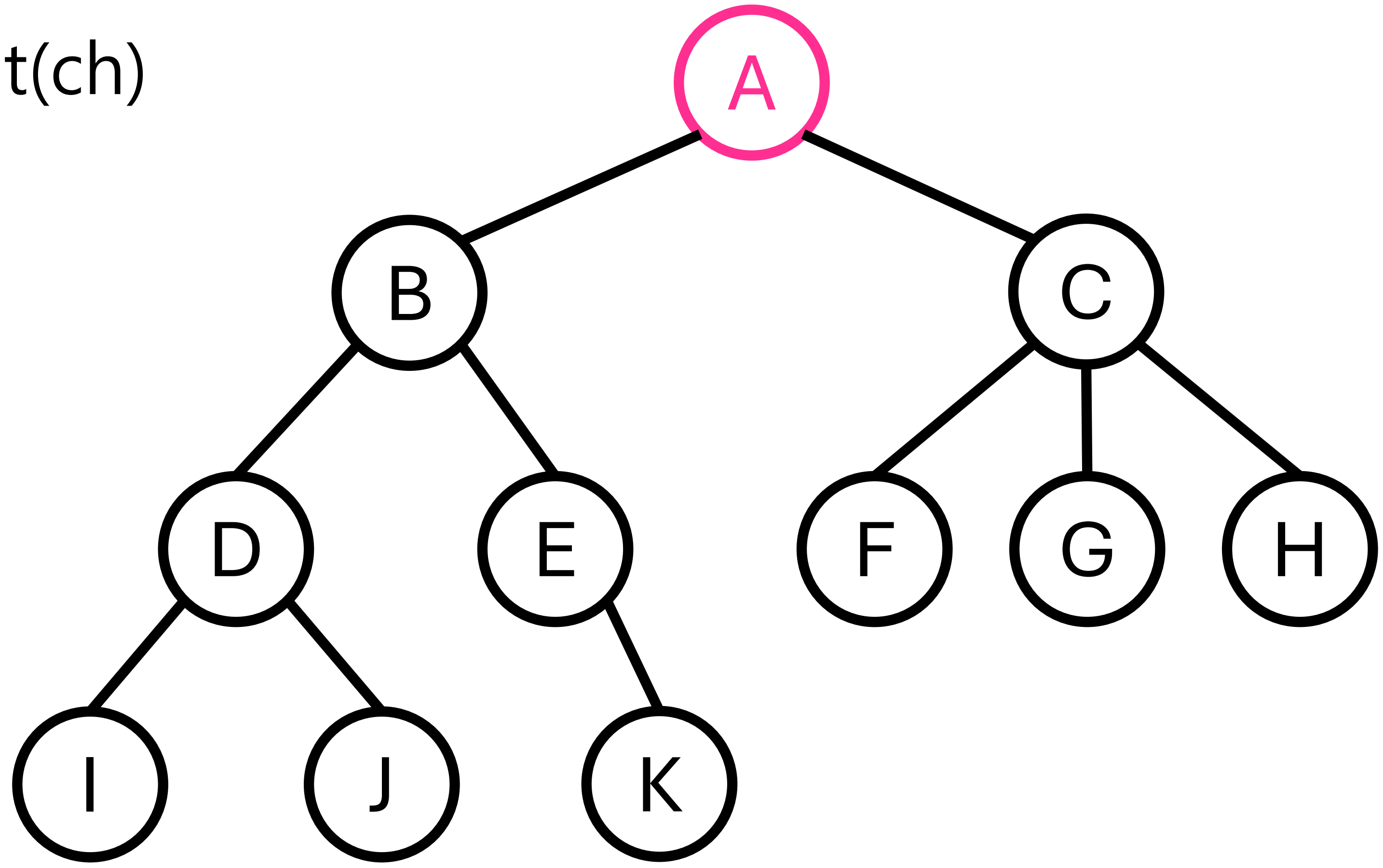
개선된 height 함수

- height(B) -> 1 + height(ch)
  - h = 2
- height(A) -> 1 + height(ch)



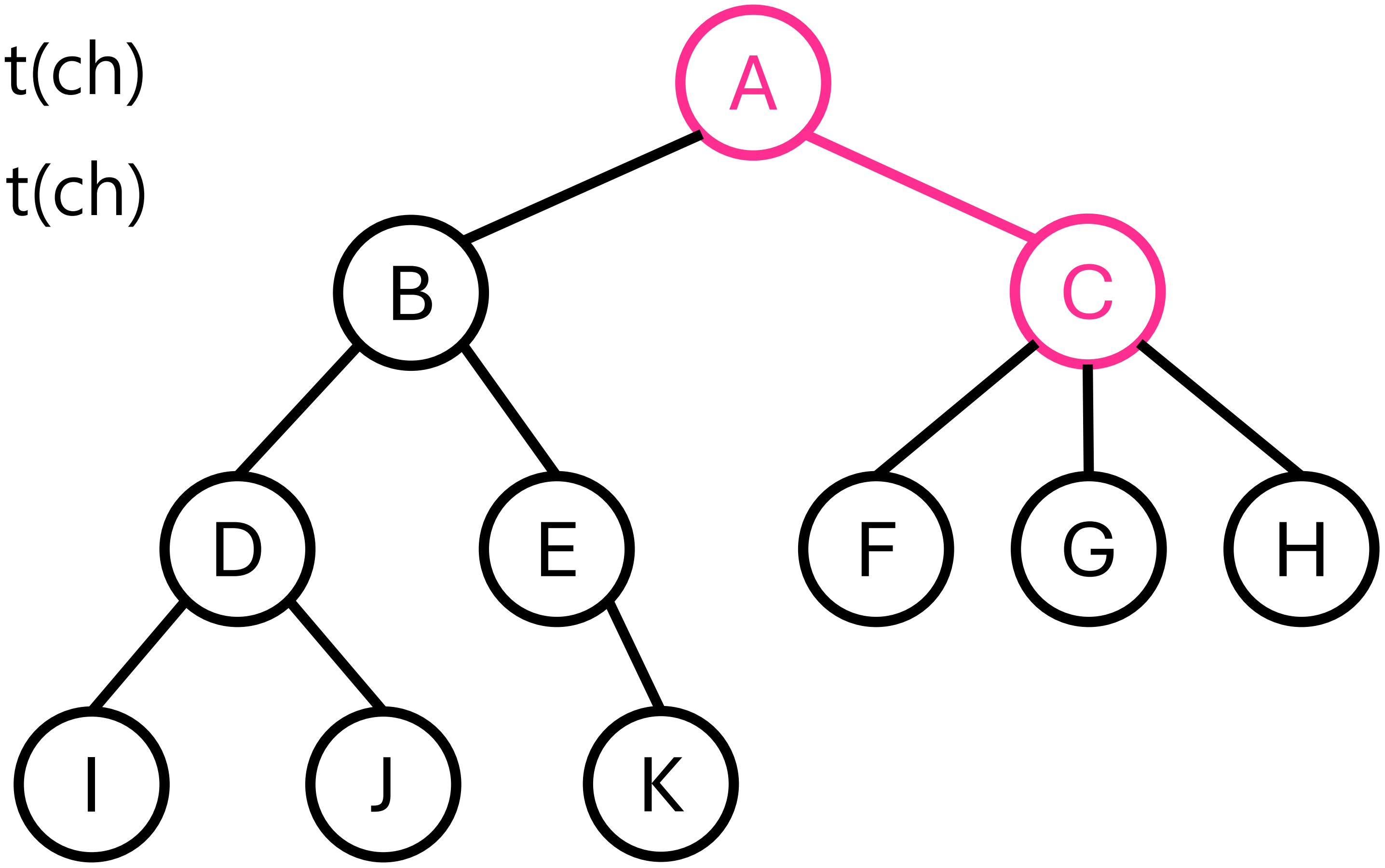
개선된 height 함수

- height(A) -> 1 + height(ch)
  - h = 3



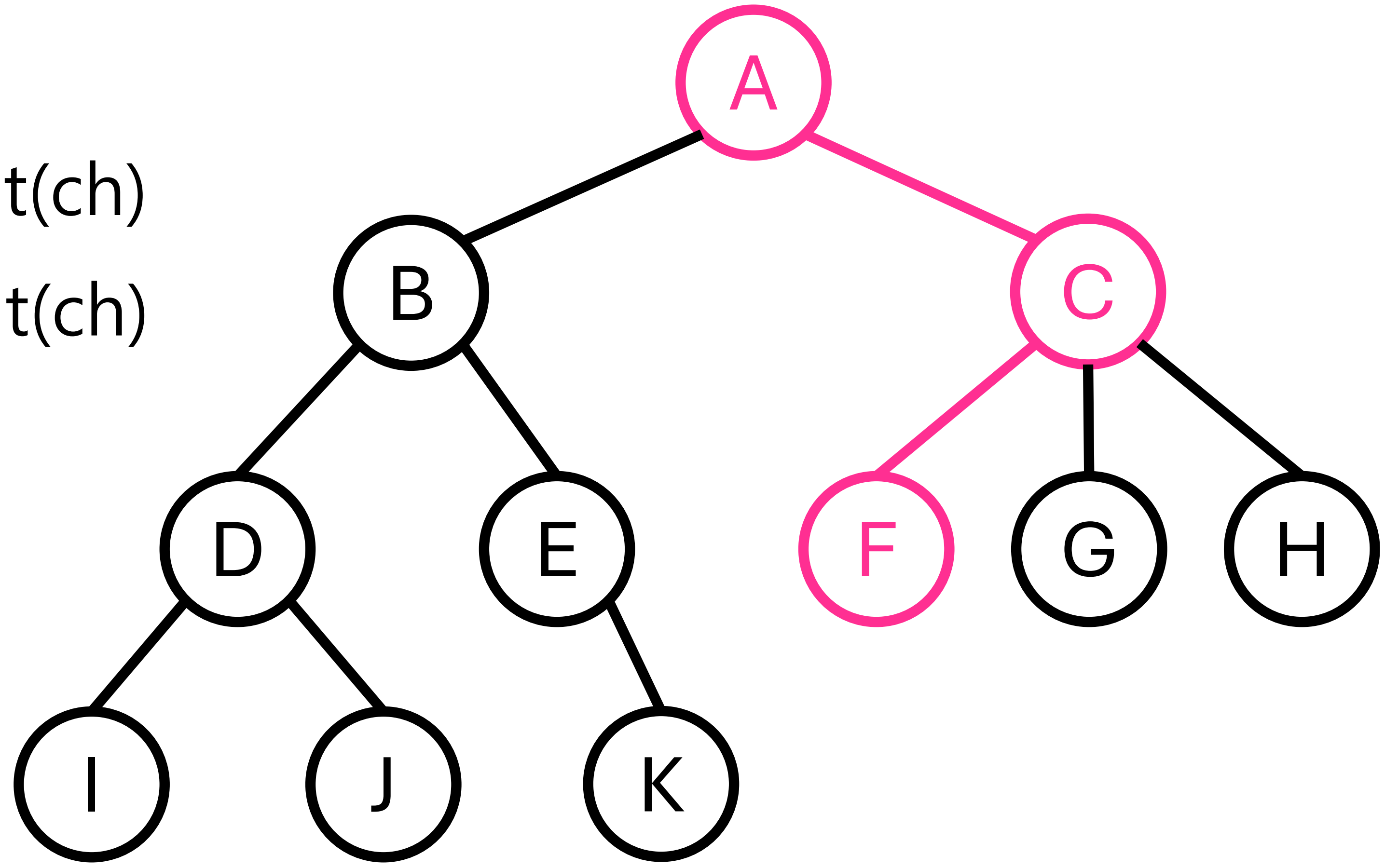
개선된 height 함수

- height(C) -> 1 + height(ch)
- height(A) -> 1 + height(ch)
  - h = 3



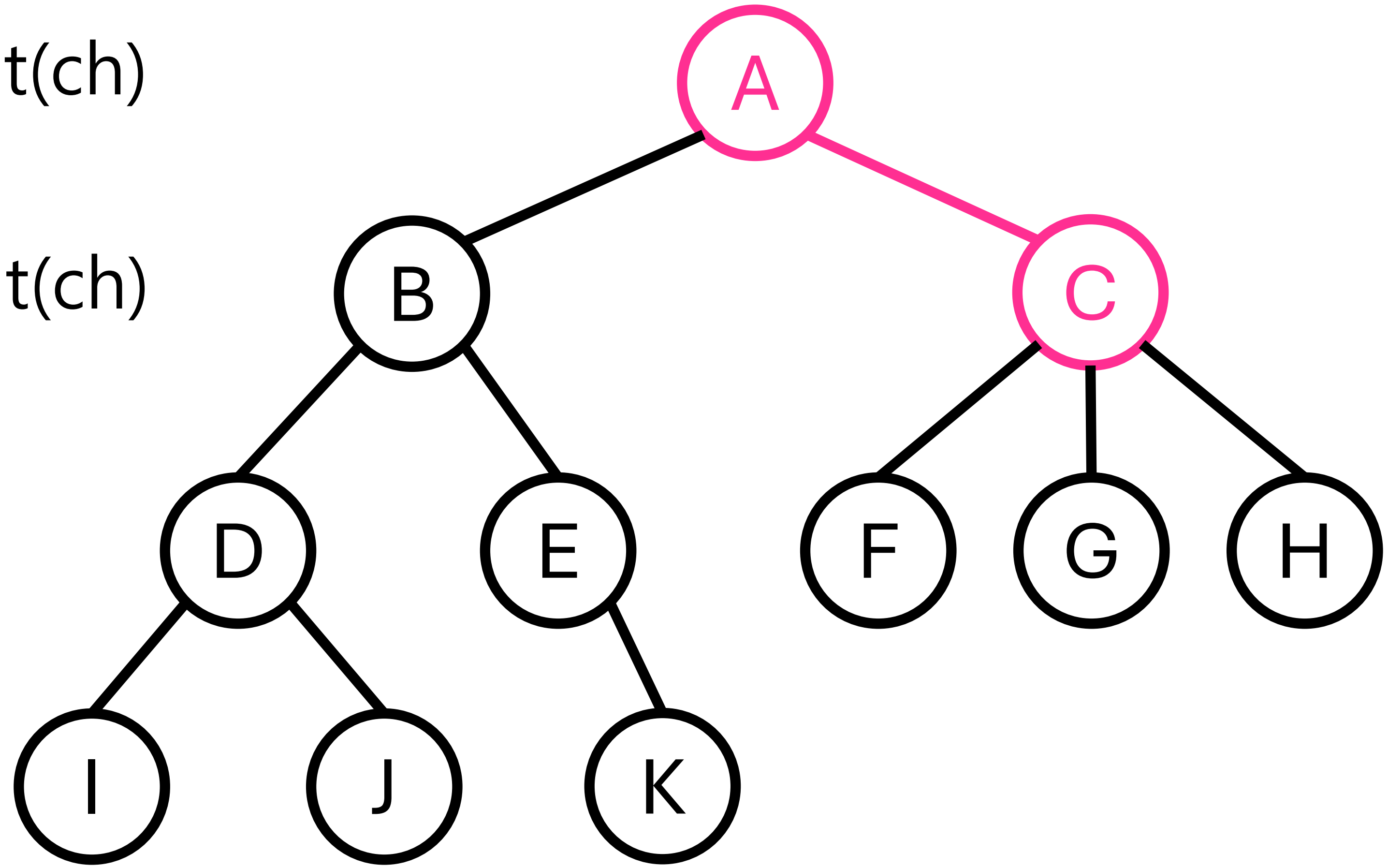
개선된 height 함수

- height(F) -> 0
- height(C) -> 1 + height(ch)
- height(A) -> 1 + height(ch)
  - h = 3



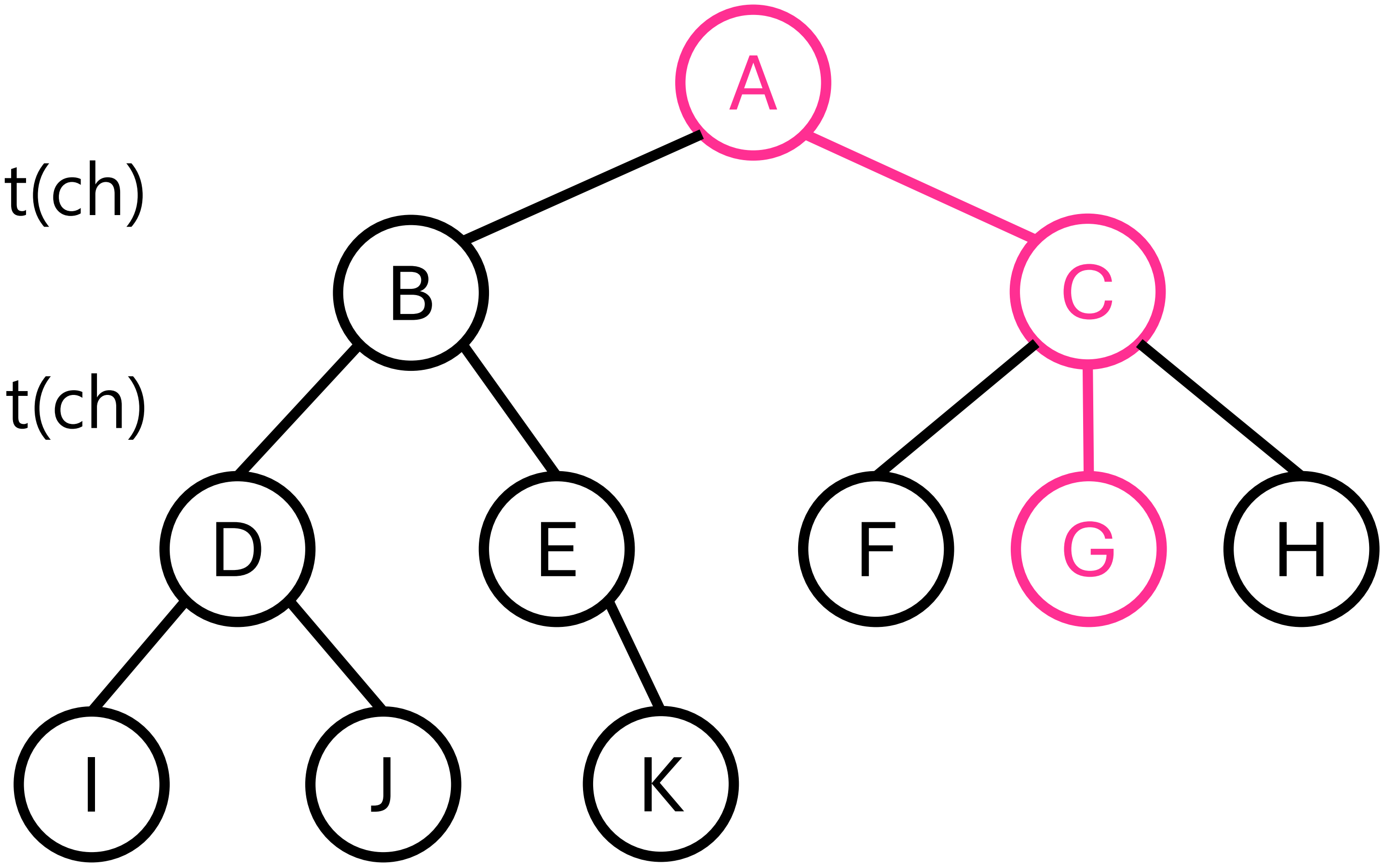
개선된 height 함수

- height(C) -> 1 + height(ch)
  - h = 1
- height(A) -> 1 + height(ch)
  - h = 3



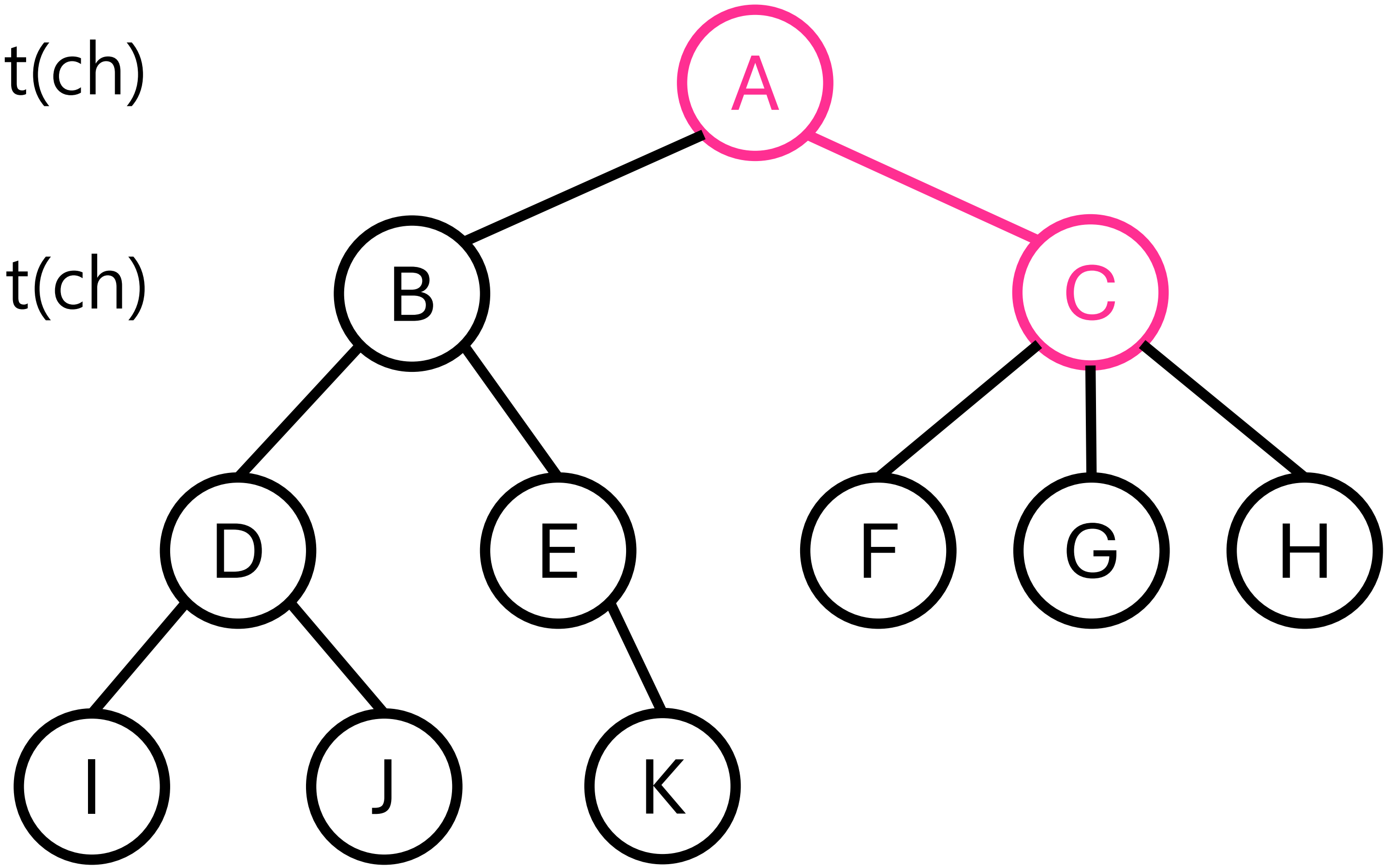
개선된 height 함수

- height(G) -> 0
- height(C) -> 1 + height(ch)
  - h = 1
- height(A) -> 1 + height(ch)
  - h = 3



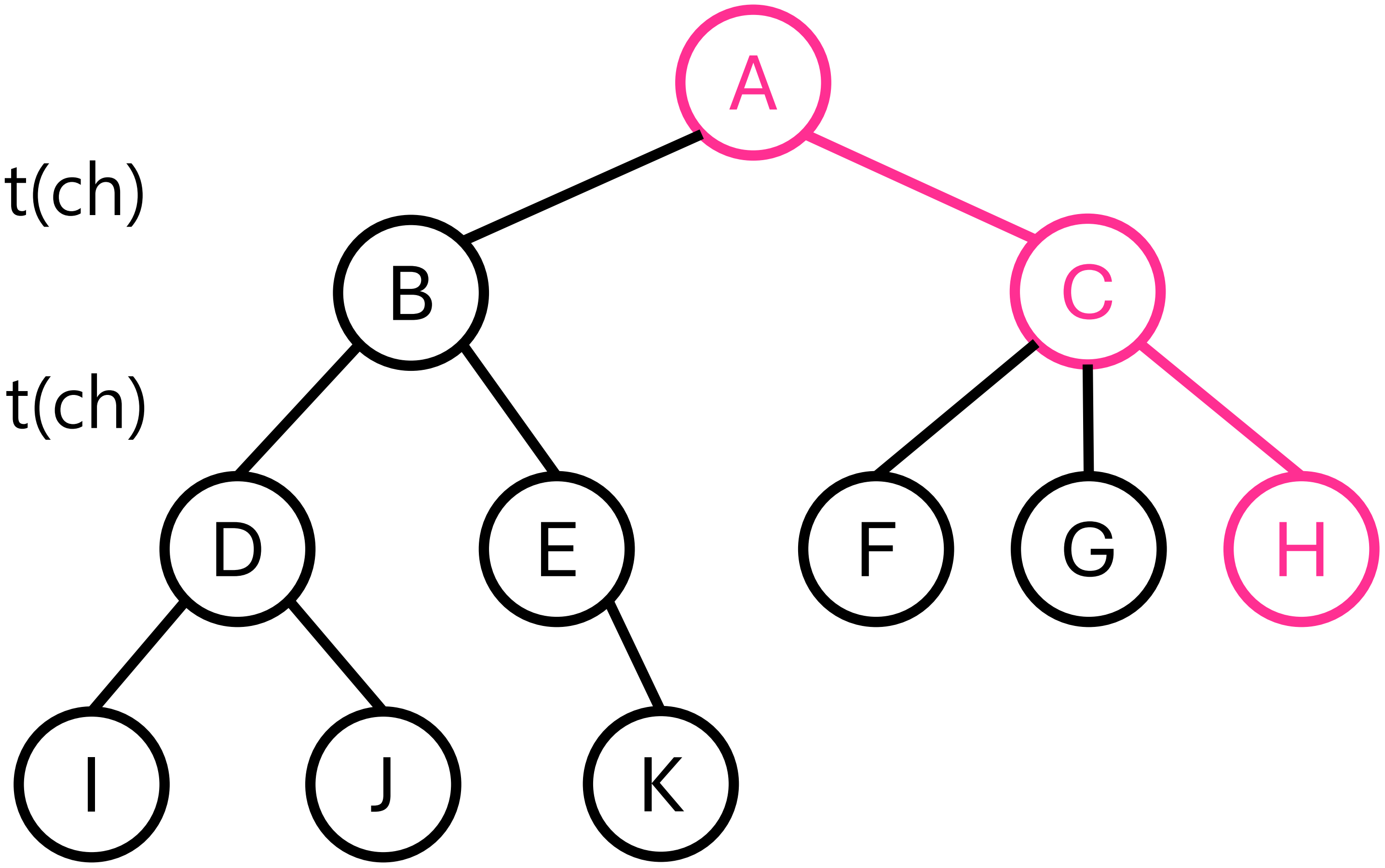
개선된 height 함수

- height(C) -> 1 + height(ch)
  - h = 1
- height(A) -> 1 + height(ch)
  - h = 3



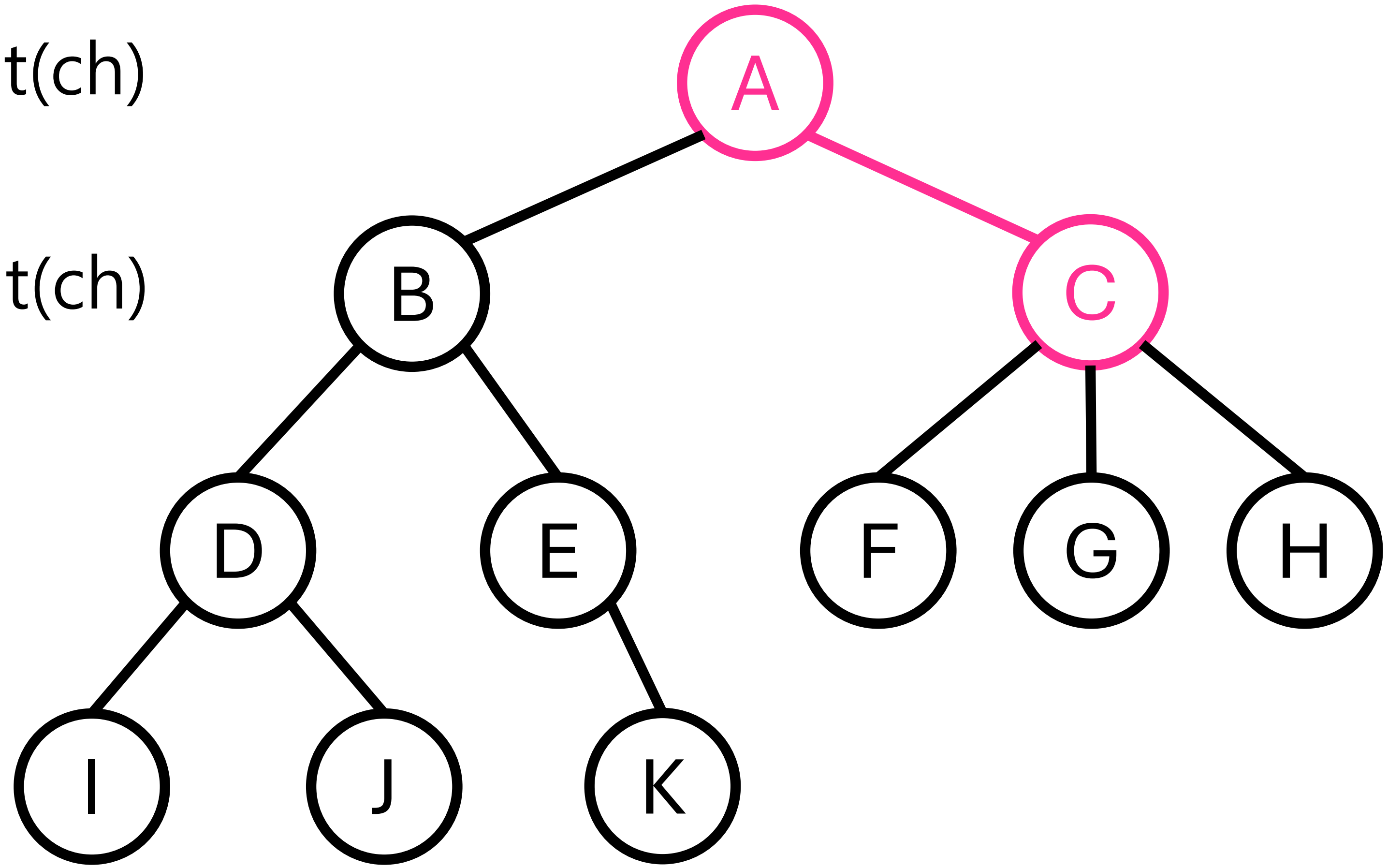
개선된 height 함수

- height(H) -> 0
- height(C) -> 1 + height(ch)
  - h = 1
- height(A) -> 1 + height(ch)
  - h = 3



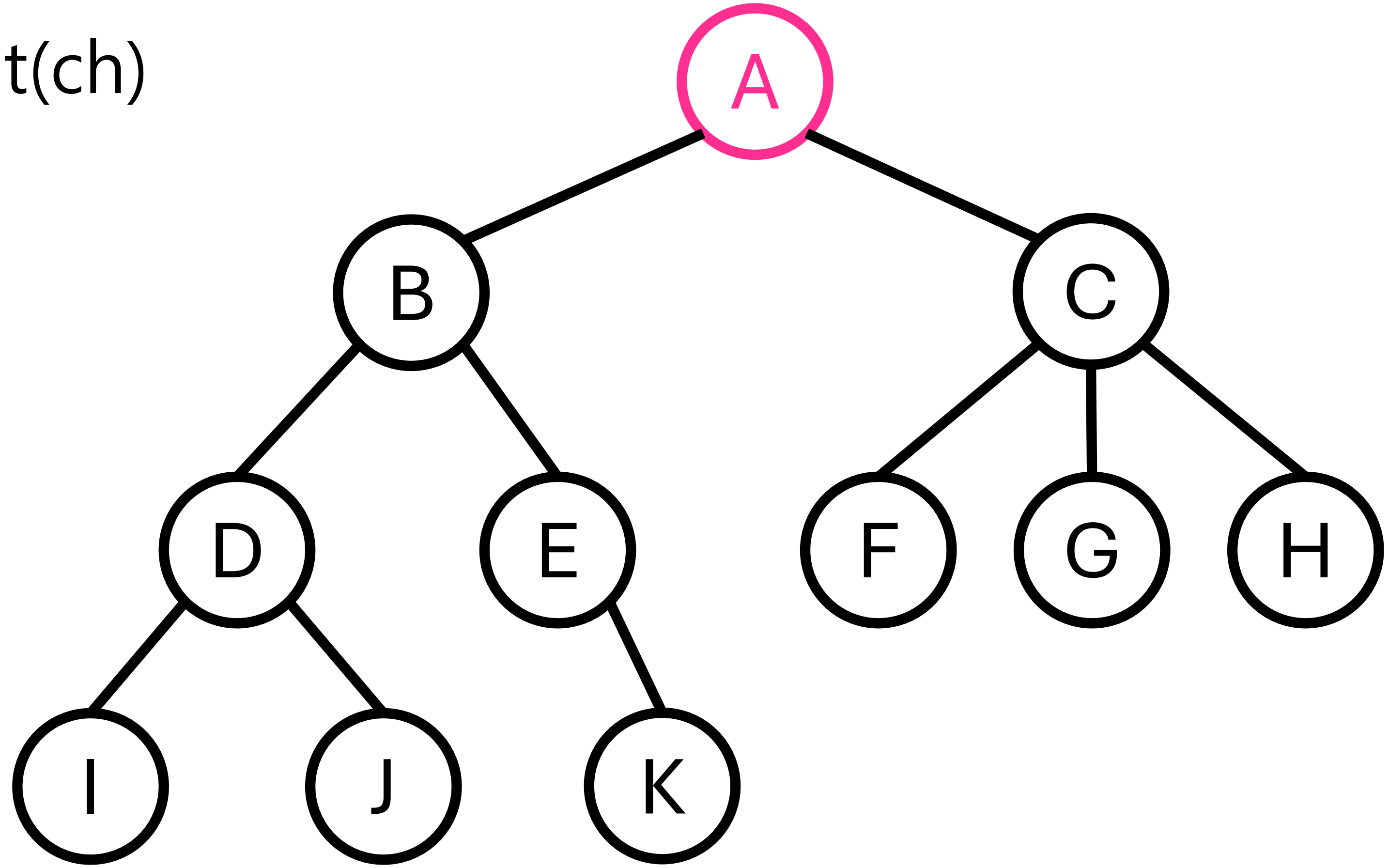
개선된 height 함수

- height(C) -> 1 + height(ch)
  - h = 1
- height(A) -> 1 + height(ch)
  - h = 3



개선된 height 함수

- height(A) -> 1 + height(ch)
  - h = 3



height2 함수의 복잡도

- 모든 경우에 대한 분석
  - 단위연산 : height2 함수 호출
  - 입력크기 : 자식 노드의 개수 c_p
  - 재귀 함수에서 height2 함수 호출은 자식 노드의 개수 만큼
  - 루트 노드에서 시작해 각 노드의 자식 노드에 대해 함수를 호출하고, 한번 호출된 노드에 대해서는 다시 계산이 이루어지지 않음
  - 즉 모든 노드에 대해 한 번씩만 함수가 호출이 됨
  - T(n) = n

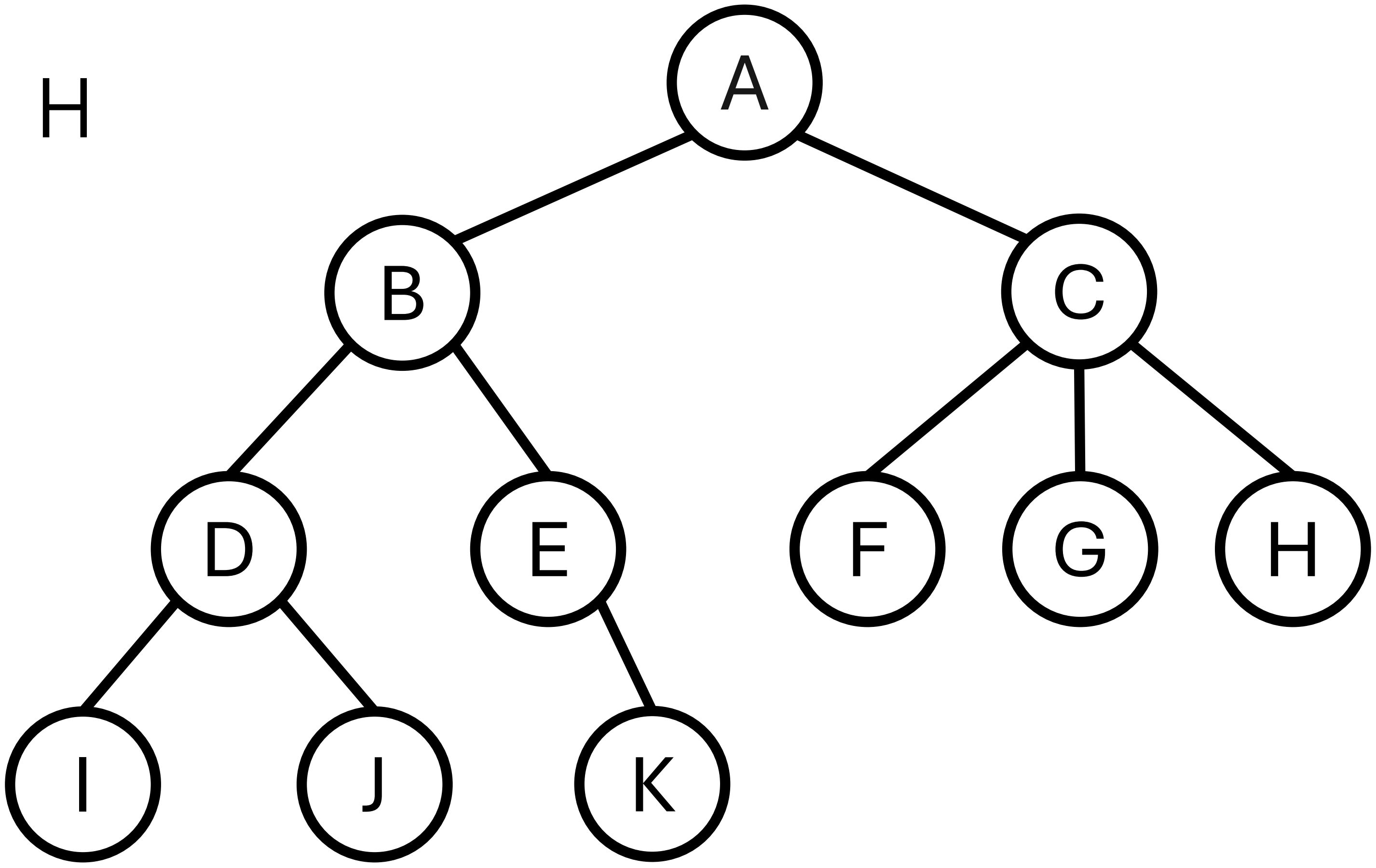
트리의 순회 방법

- pre order (전위순회)
  - 자식 노드들을 탐색하기 이전에 방문 처리
- post order (후위순회)
  - 자식 노드들을 탐색한 이후에 방문 처리
- in order (전위순회)
  - 자식 노드들을 탐색하는 중에 방문 처리

Pre order

- A B D I J E K C F G H

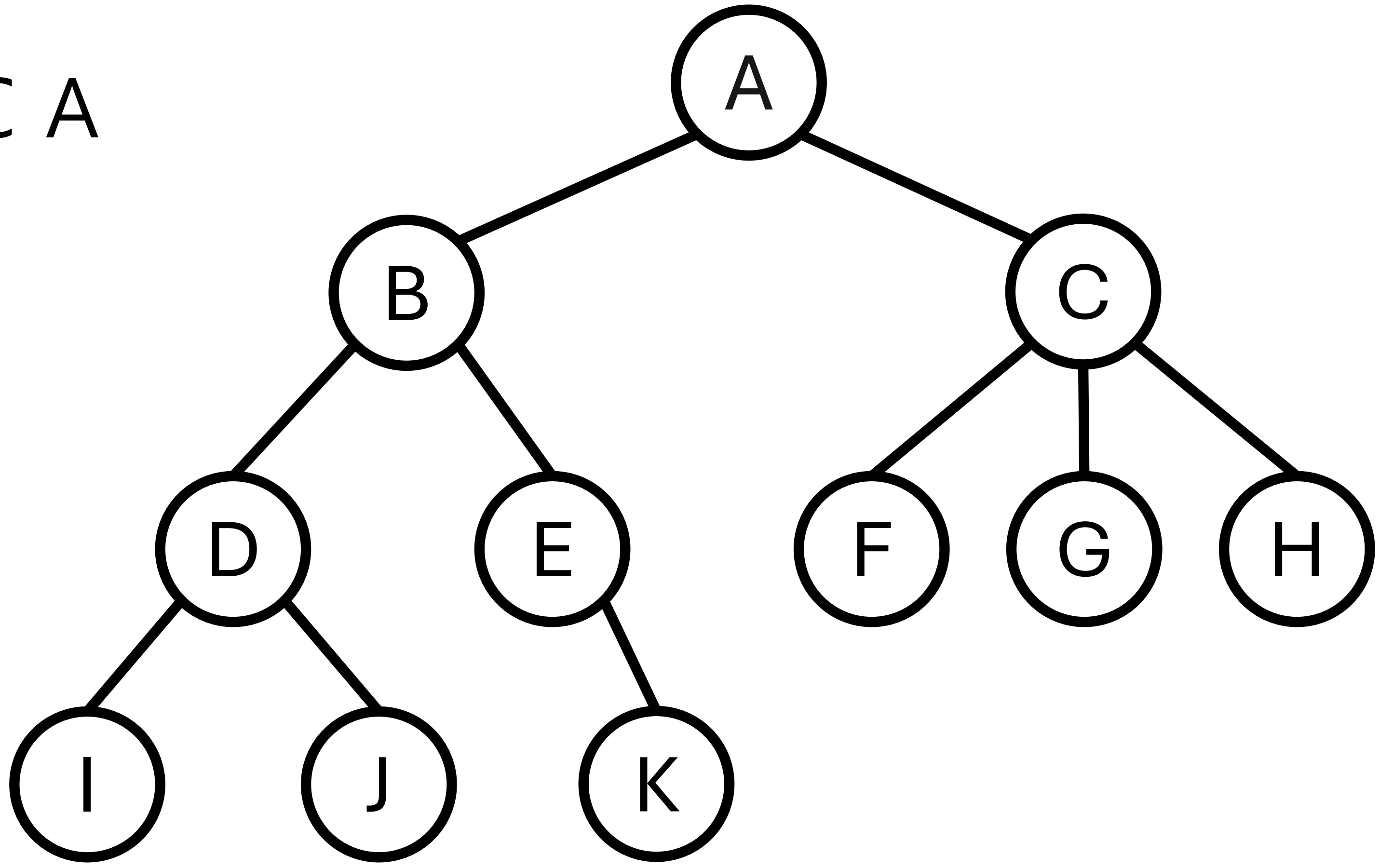
```
preorder(x) {
  visit(x)
  preorder(x.childs)
}
```



Post order

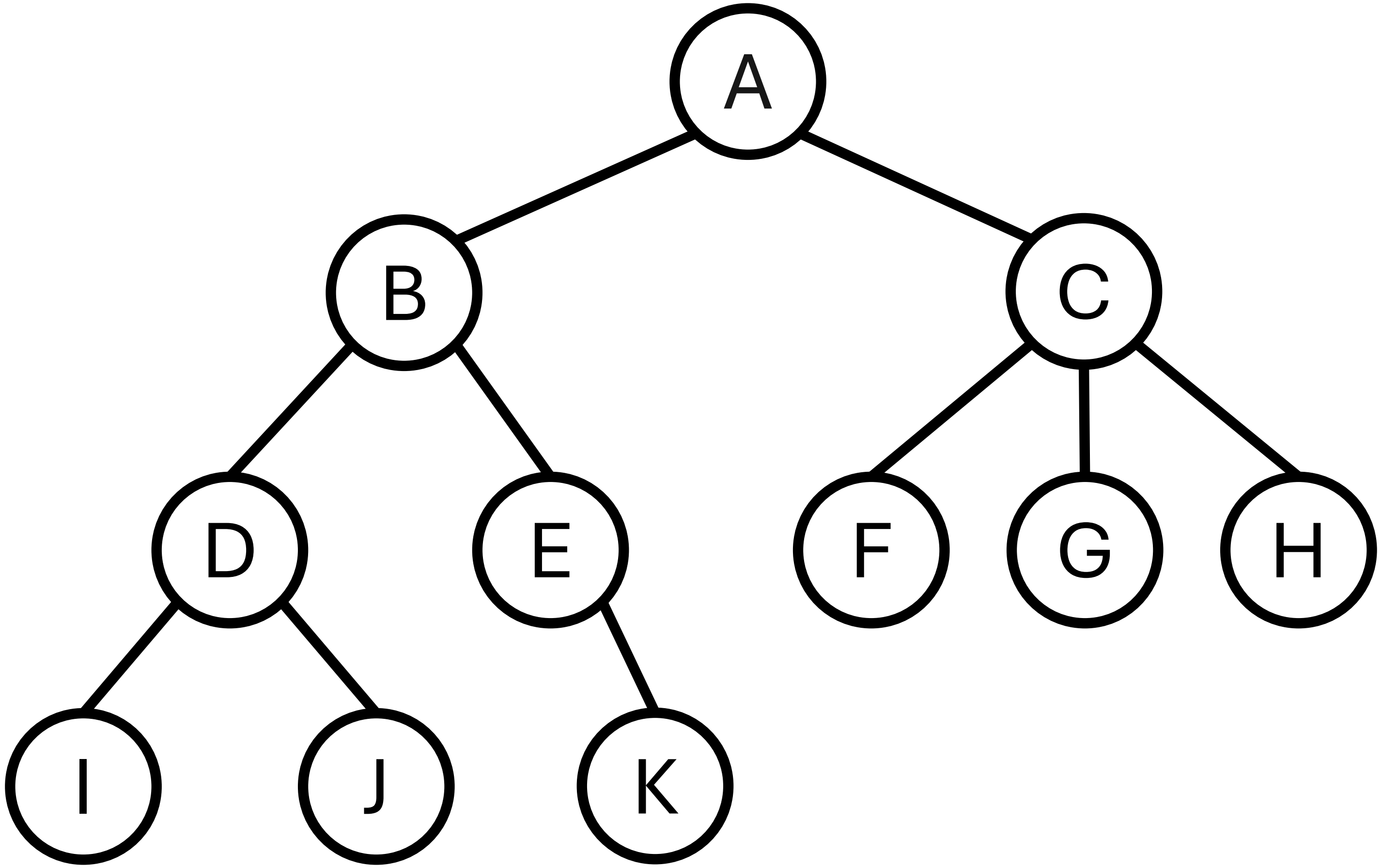
- I J D K E B F G H C A

```
postorder(x) {
  postorder(x.childs)
  visit(x)
}
```



In order

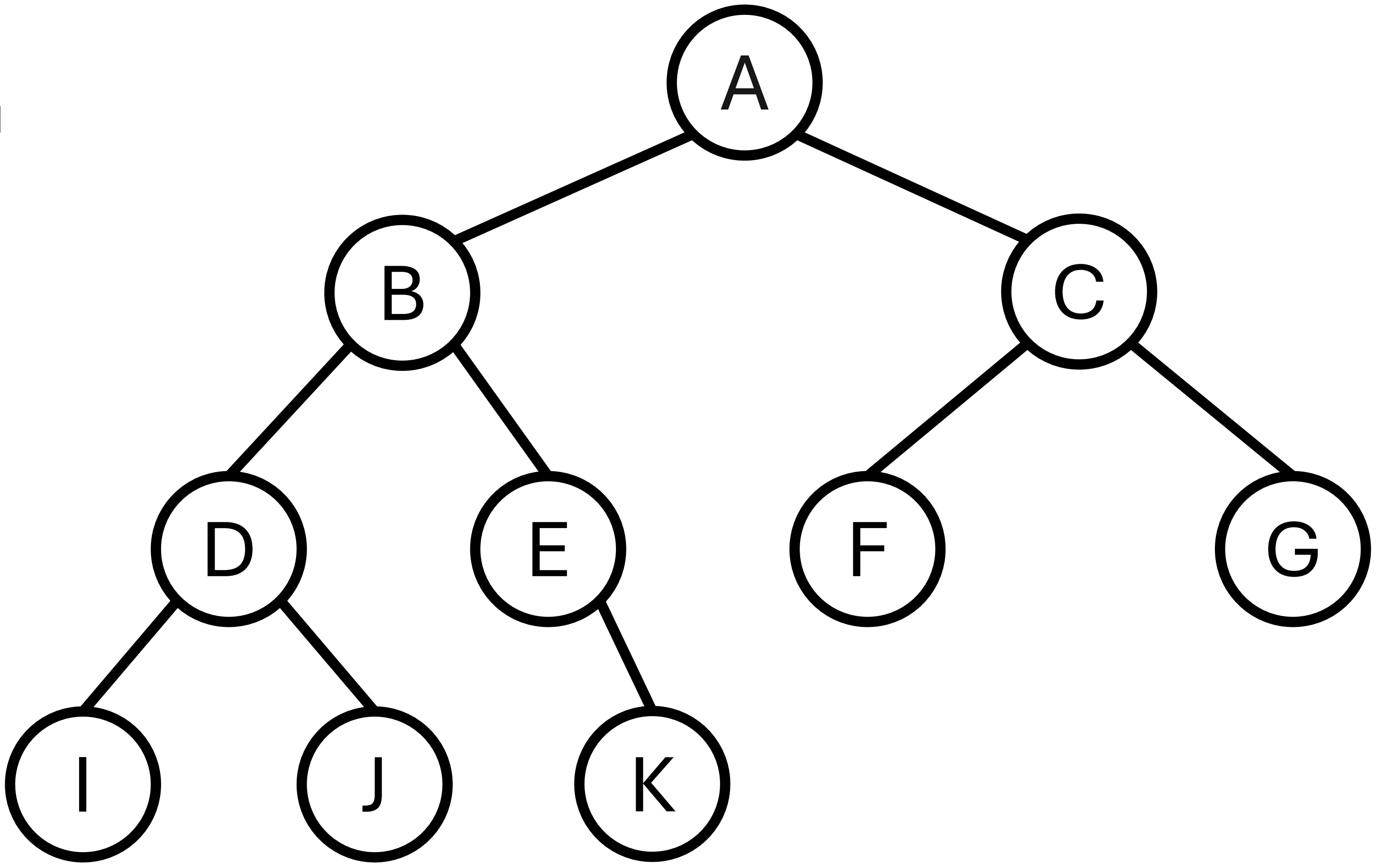
- ???



In order

- I D J B E K A F C G

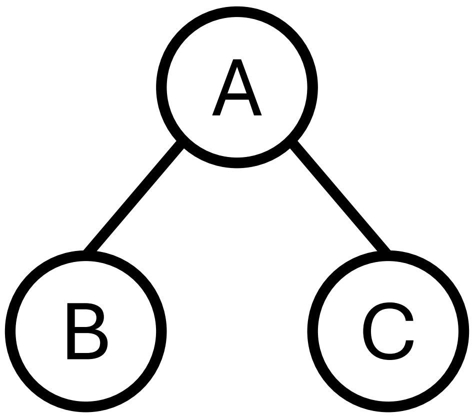
```
inorder(x) {
  inorder(x.left)
  visit(x)
  inorder(x.right)
}
```



Binary Tree

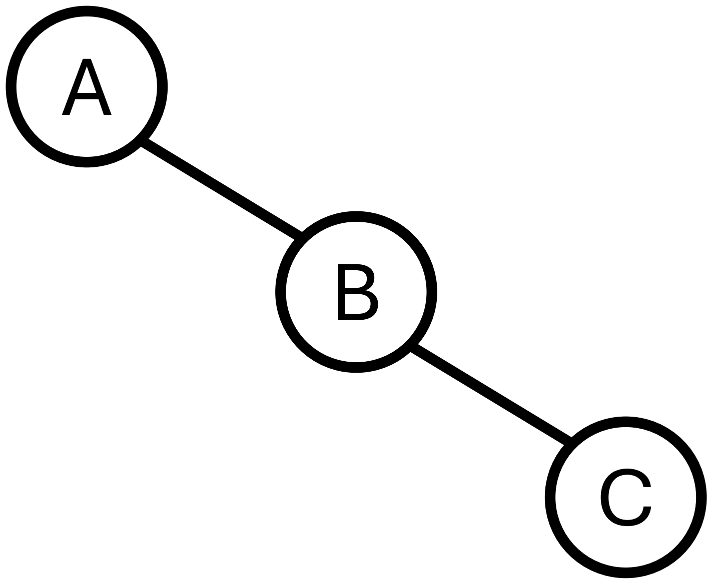
Binary Tree

- 루트 노드를 중심으로 두 개의 서브 트리로 나뉘어짐
- 나뉘어진 두 서브 트리도 모두 이진 트리여야 함
- 트리가 공백인 경우도 가능



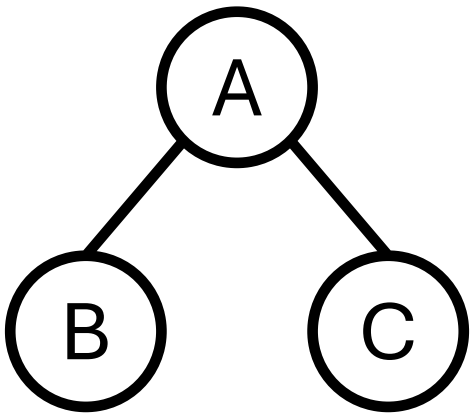
Binary Tree의 종류

- 편향 이진 트리 (Skewed Binary Tree)
  - 높이가 k일 때 k개의 노드를 가지면서 모든 노드가 한쪽 방향으로만 서브 트리를 가지고 있는 트리



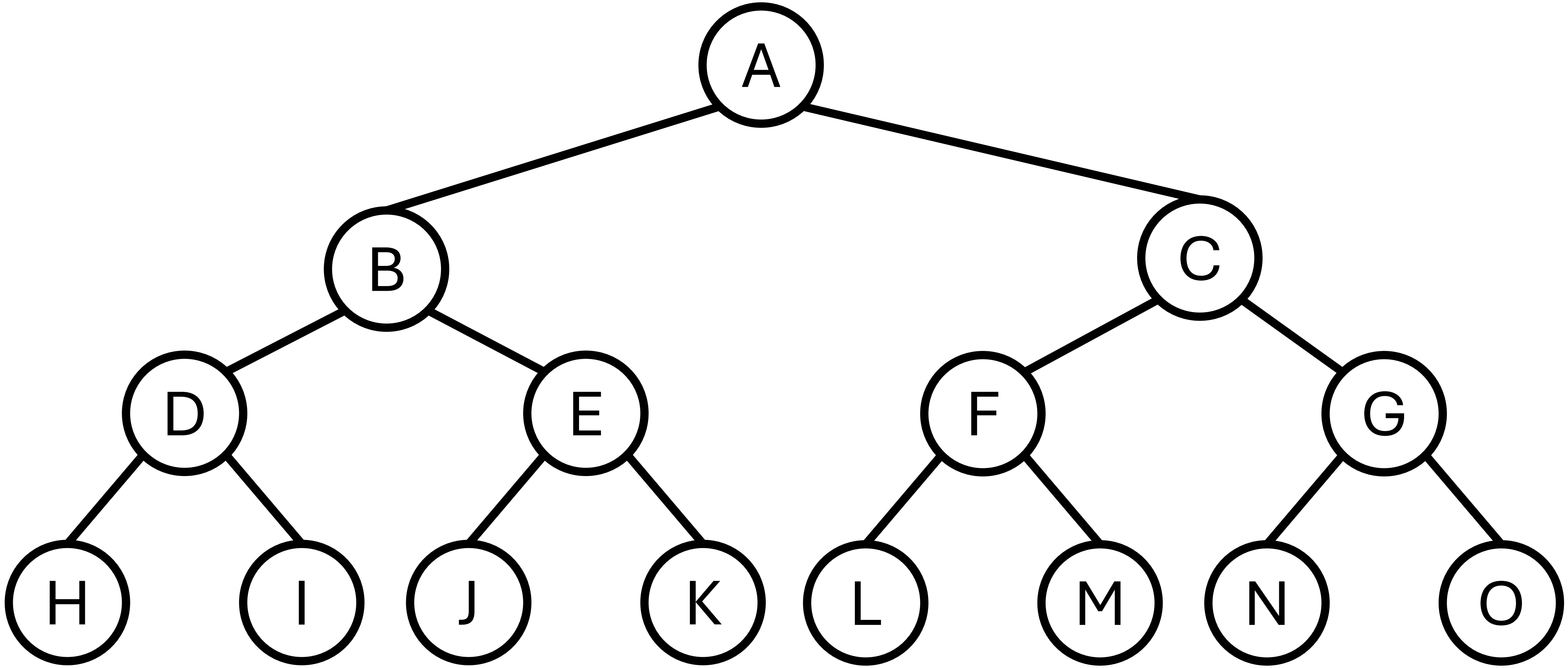
Binary Tree의 종류

- 포화 이진 트리(Full Binary Tree)
  - 모든 레벨에 노드가 포화상태로 차 있는 이진 트리
  - 깊이가 k일 때, 최대의 노드 개수인 $2^k - 1$의 노드를 갖는 이진 트리



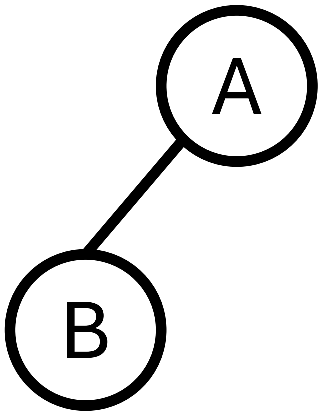
Binary Tree의 종류

- 포화 이진 트리(Full Binary Tree)



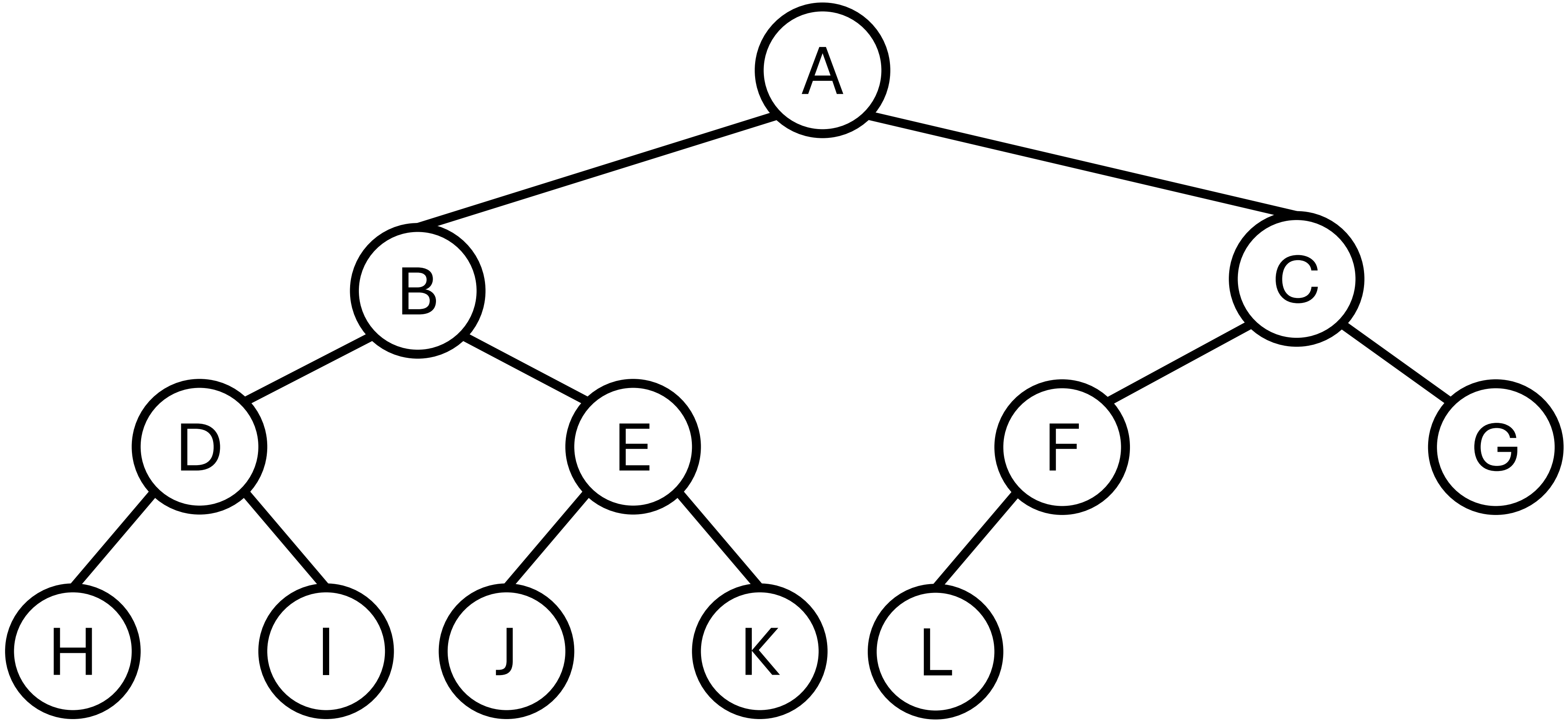
Binary Tree의 종류

- 완전 이진 트리(Complete Binary Tree)
  - 깊이가 k이고 노드 수가 n인 이진 트리
  - 단, 각 노드들이 깊이가 k인 포화 이진 트리에서 1부터 n까지 번호를 붙인 노드와 1대 1로 일치
  - 완전 이진 트리의 높이 : ceil(log(n+1))



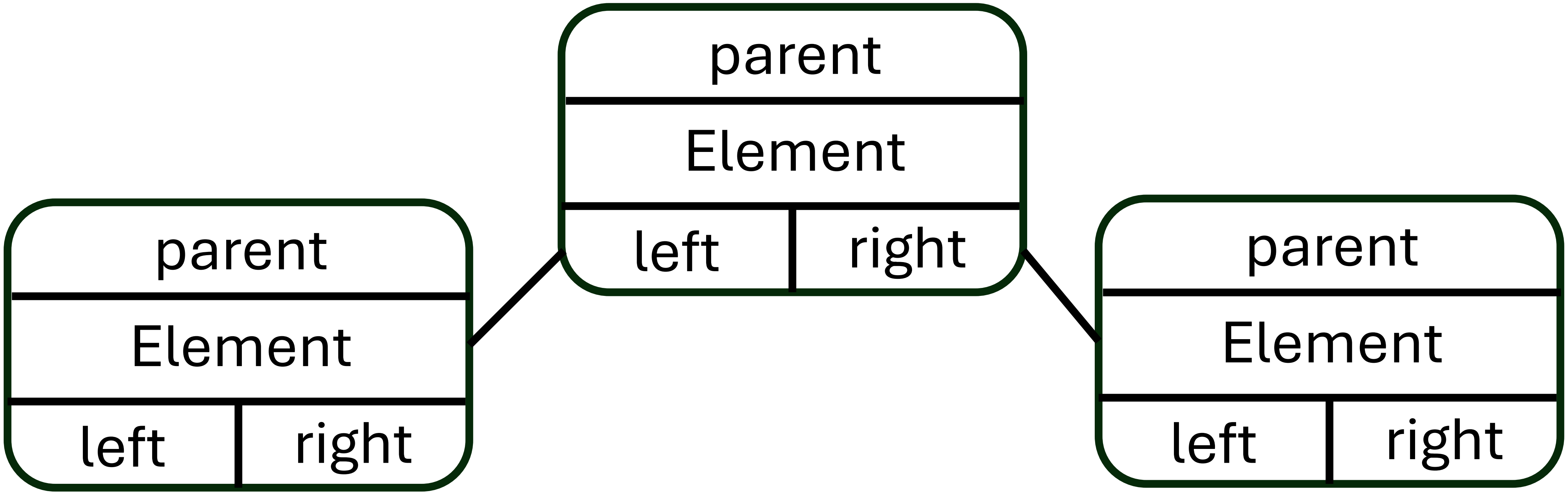
Binary Tree의 종류

- 포화 이진 트리(Full Binary Tree)



Linked Binary Tree

- 노드를 동적으로 생성하고, 노드들을 포인터로 연결하여 포인터로 연결



Linked Binary Tree

```cpp
template <typename T>
class Node {
private:
      T element;
      Node<T> *parent;
      Node<T> *left;
      Node<T> *right;
```

```cpp
Node(T e) {
      element = e;
      parent = nullptr;
      left = nullptr;
      right = nullptr;
}
```

Linked Binary Tree

```
T get_element() const {
    return element;
}
Node<T> *get_parent() const {
    return parent;
}
```

```
Node<T> *get_left() const {
    return left;
}
Node<T> *get_right() const {
    return right;
}
```

Linked Binary Tree

```
template <typename T>
class LinkedBinaryTree {
private:
    Node<T> *root;
    int n;
```

```
LinkedBinaryTree() {
    root = nullptr;
    n = 0;
}
```

Linked Binary Tree

```cpp
int size() const {
	return n;
}
bool empty() const {
	return size() == 0;
}
```

```cpp
Node<T> *get_root() const {
	return root;
}
void add_root(T e) {
	root = new Node<T>(e);
	n = 1;
}
```

Linked Binary Tree

```cpp
void add_left(Node<T> *node, T e) {
    Node<T> *new_node = new Node<T>(e);
    new_node->parent = node;
    node->left = new_node;
    n++;
}
```

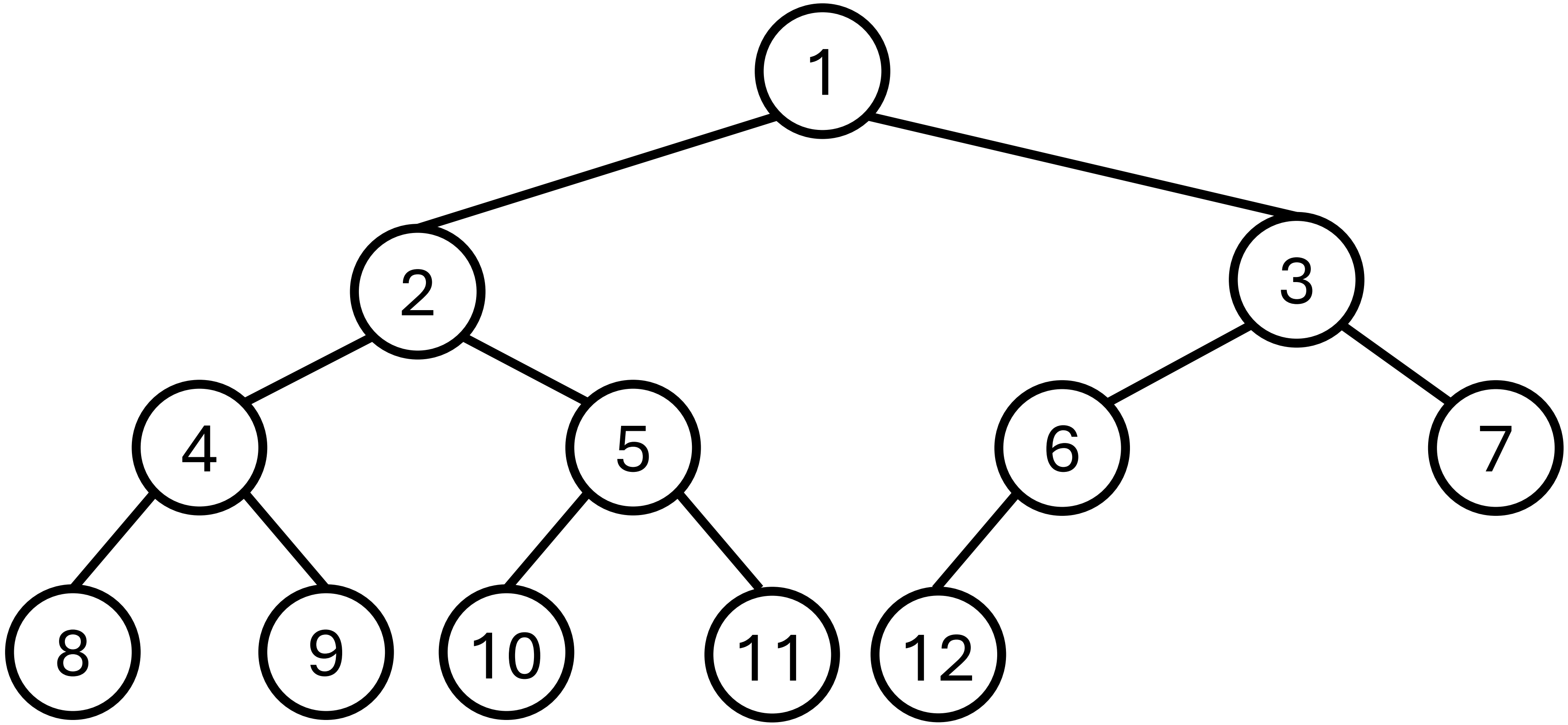
Linked Binary Tree

```cpp
void remove(Node<T> *node) {
    if (node->get_parent() == nullptr) {
        root = nullptr;
    } else {
        Node<T> *parent = node->get_parent();
        if (node == parent->get_left()) {
            parent->left = nullptr;
        } else {
            parent->right = nullptr;
        }
    }
    delete node;
    n--;
}
```

Array Binary Tree

- 노드에 번호를 부여하고 그 번호에 해당하는 값을 배열의 인덱스 값으로 활용
- 0번 인덱스는 활용하지 않고, 1번 인덱스를 루트 노드로 사용
  - 부모 노드의 번호 : x / 2
  - 왼쪽 자식 노드의 번호 : x * 2
  - 오른쪽 사직 노드의 번호 : x * 2 + 1

Array Binary Tree



Array Binary Tree

```cpp
template <typename T>
class ArrayBinaryTree {
private:
      T *arr;
      int capacity;
      int n;
```

```cpp
ArrayBinaryTree(int cap) {
      capacity = cap;
      arr = new T[capacity];
      n = 0;
}
```

Array Binary Tree

```
int size() const {
    return n;
}
bool empty() const {
    return size() == 0;
}
```

```
int get_root() const {
    return 1;
}
void add_root(const T e) {
    arr[1] = e;
    n = 1;
}
```

Array Binary Tree

```
void add_left(int node, T e) {
    arr[2 * node] = e;
}
void add_right(int node, T e) {
    arr[2 * node + 1] = e;
}
```

```
void remove(int node) {
    arr[node] = NULL;
    n--;
}
```