

## 6. Data Structure and Algorithms

# 목차

- ▶ Tree as an Abstract Data Type and Structure
- ▶ Terminologies of Tree Structure
- ▶ Characteristics of Tree
- ▶ Binary Search Tree and Implementation
- ▶ Insert and Search Operation of Binary Search Tree
- ▶ Delete Operation and Minimum & Maximum of Binary Search Tree
- ▶ Tree Traversing

# 1. Tree as an Abstract Data Type and Tree as an abstract data type

## Tree structure

- ◆ Abstract data type
- ◆ Data stored
- ◆ As a tree structure
- ◆ Operations
  - ◆ Ordinary data structure operations just as linked lists
  - ◆ Insert
  - ◆ Delete
  - ◆ Search
  - ◆ Special searching approaches for trees and networks
  - ◆ Traverse

- Up - side - down 형태
- 컴퓨터에서는 아래의 그림과 같이 표현할 수 있다.
- 위에서부터 뺄어나와 다양한 일을 가지는 구조
- 데이터는 tree structure 형식으로 저장된다.
- Tree structure의 기능은 linked list로 insert, delete, search이다.
- Tree의 특별한 접근이 있는데 모든 데이터를 꺼내는것이 traversing이다.
- Linked list랑은 다르다.

# 1. Tree as an Abstract Data Type and

## Why do we use trees?

◆ Because the structure of trees is a good analogy to the various real world structures

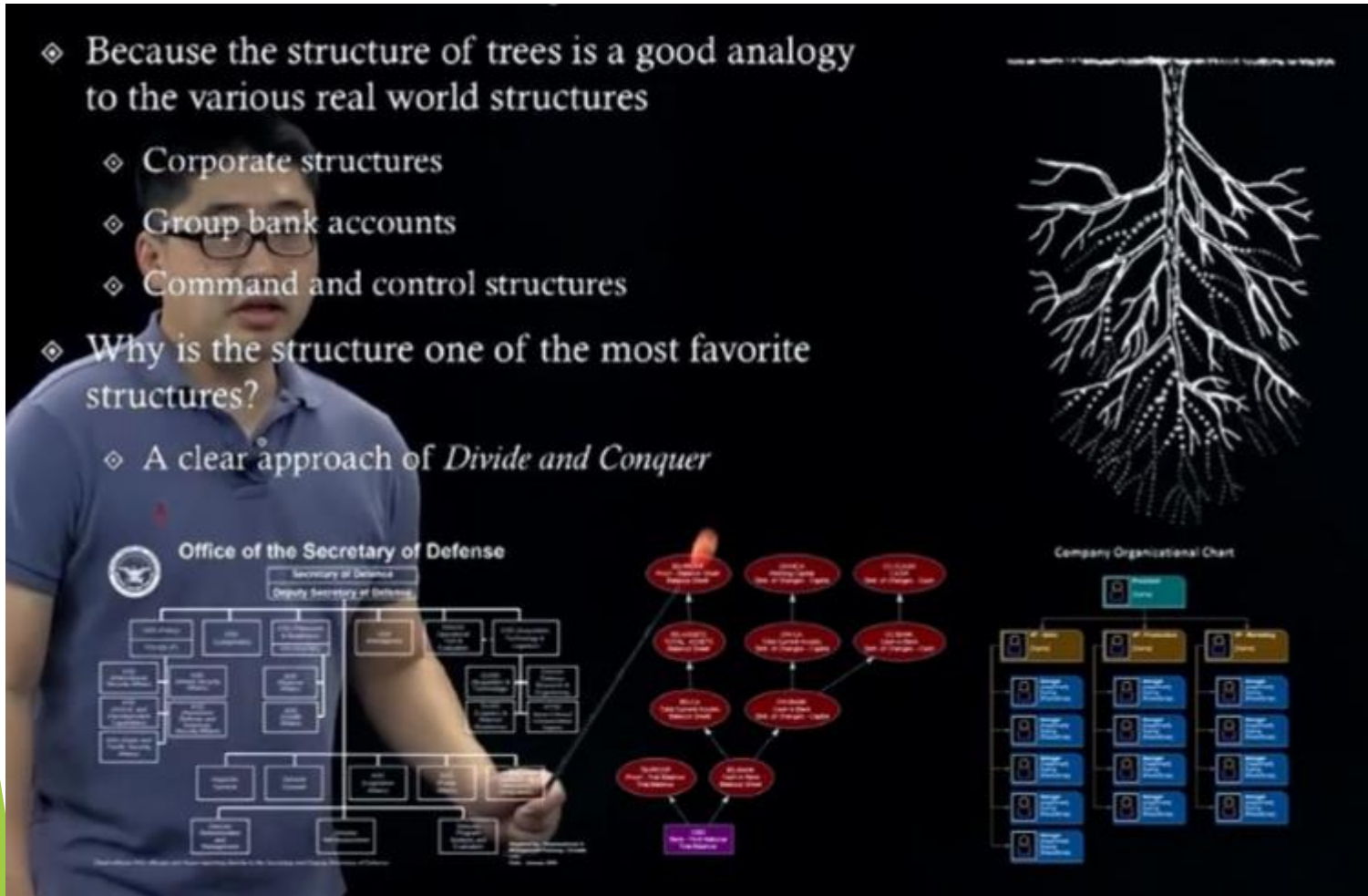
- ◆ Corporate structures
- ◆ Group bank accounts
- ◆ Command and control structures

◆ Why is the structure one of the most favorite structures?

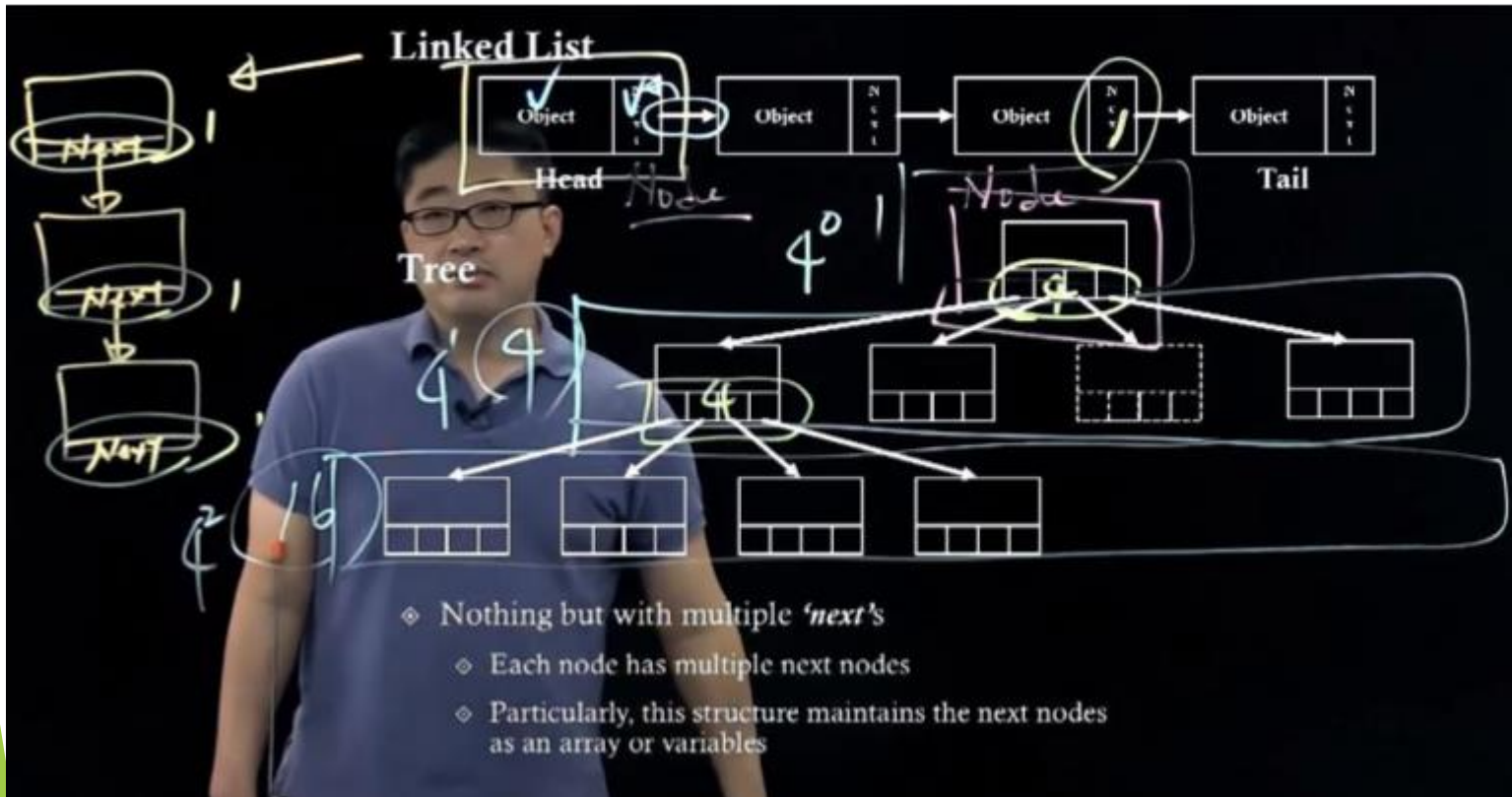
◆ A clear approach of *Divide and Conquer*



- Tree 구조는 현실에 많이 존재
- 어떤 조직의 조직도 - 어떤 회사의 그룹 bank account
- 회사는 개인과는 다르게 체계적인 bank account가 있을 수 있다.
- 조직도나 지휘체계도 있다.
- 이런 것들은 divide and conquer를 할 수 있다.
- 저장하는 데이터에 대해서 divide하고 저장한 문제를 풀어나가는 구조
- divide and conquer 적용이 되어있다.
- 또한 recursion 구조를 많이 적용한 다



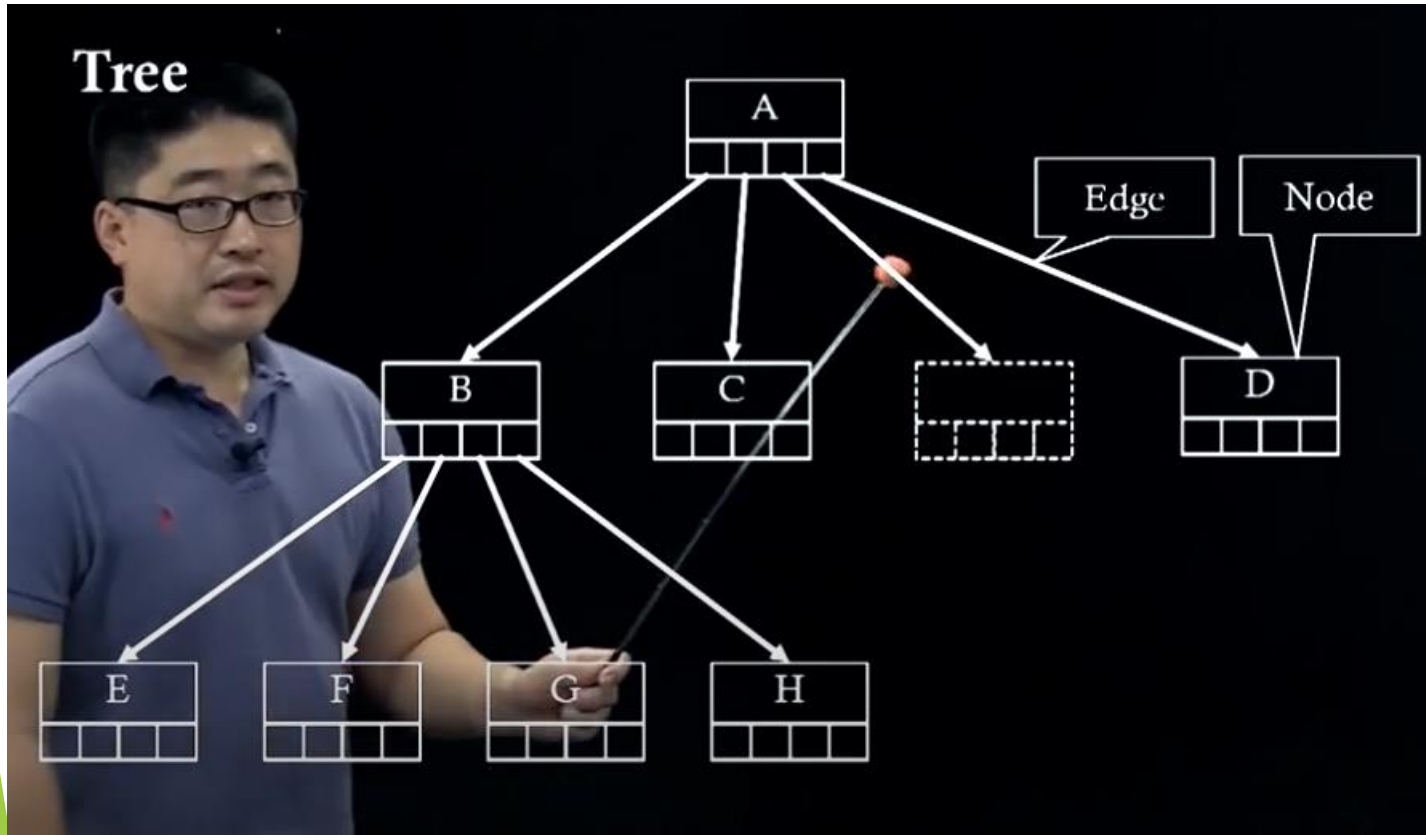
# 1. Tree as an Abstract Data Type and Structure of stored data



- Linked list는 Object를 저장하는 레퍼런스 하나
- 다음 node가 무엇인지 레퍼런스 하나로 이루어져 있다.
- tree구조와 유한데 tree 노드는 특징 점이 next가 여러 개로 나뉜다.
- Linked list는 next 노드가 하나만 되지만 tree는 여러 개다.
- Tree 구조는 그림에서 점점 커지게 된다.

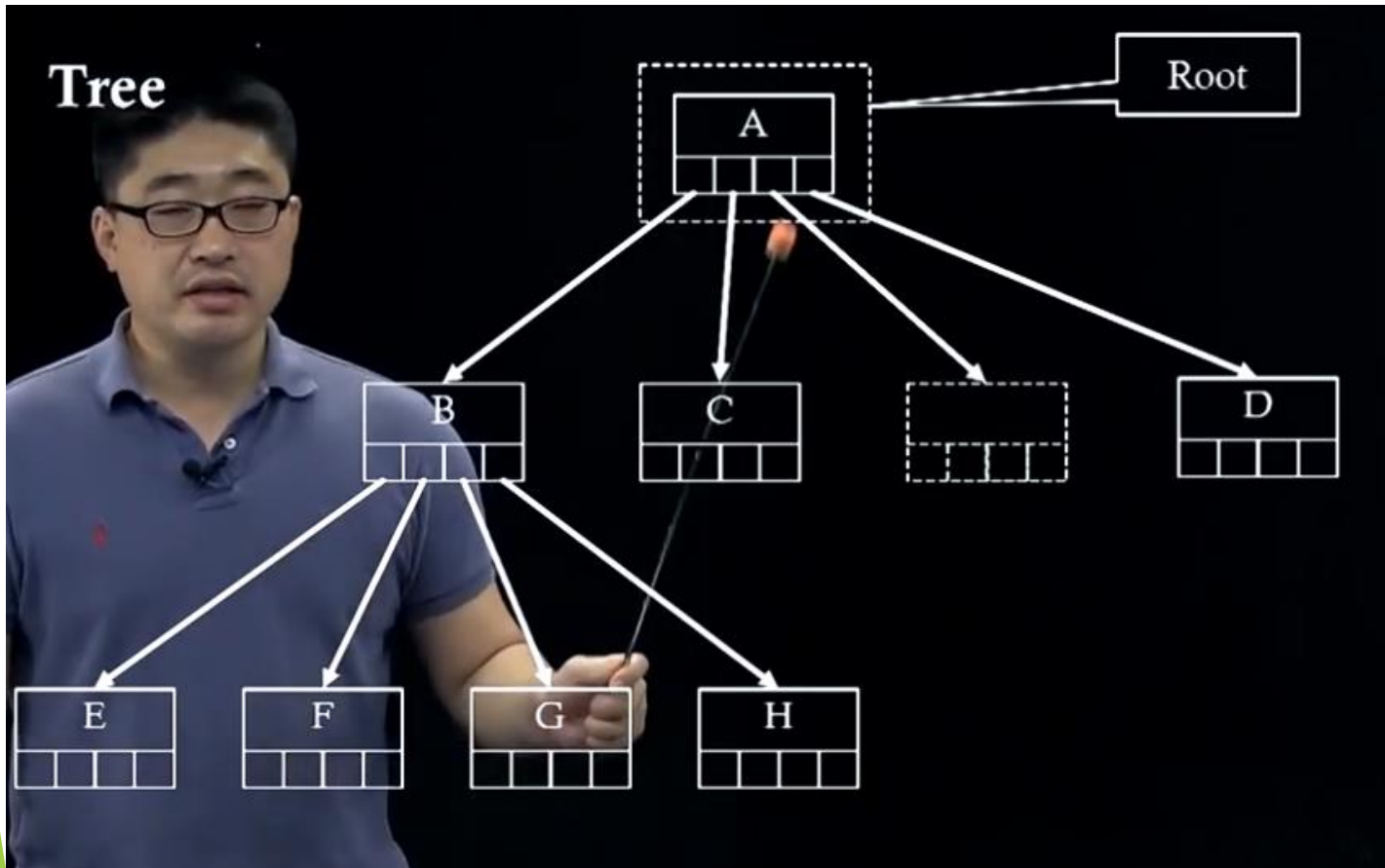


## 2. Terminologies of tree structure



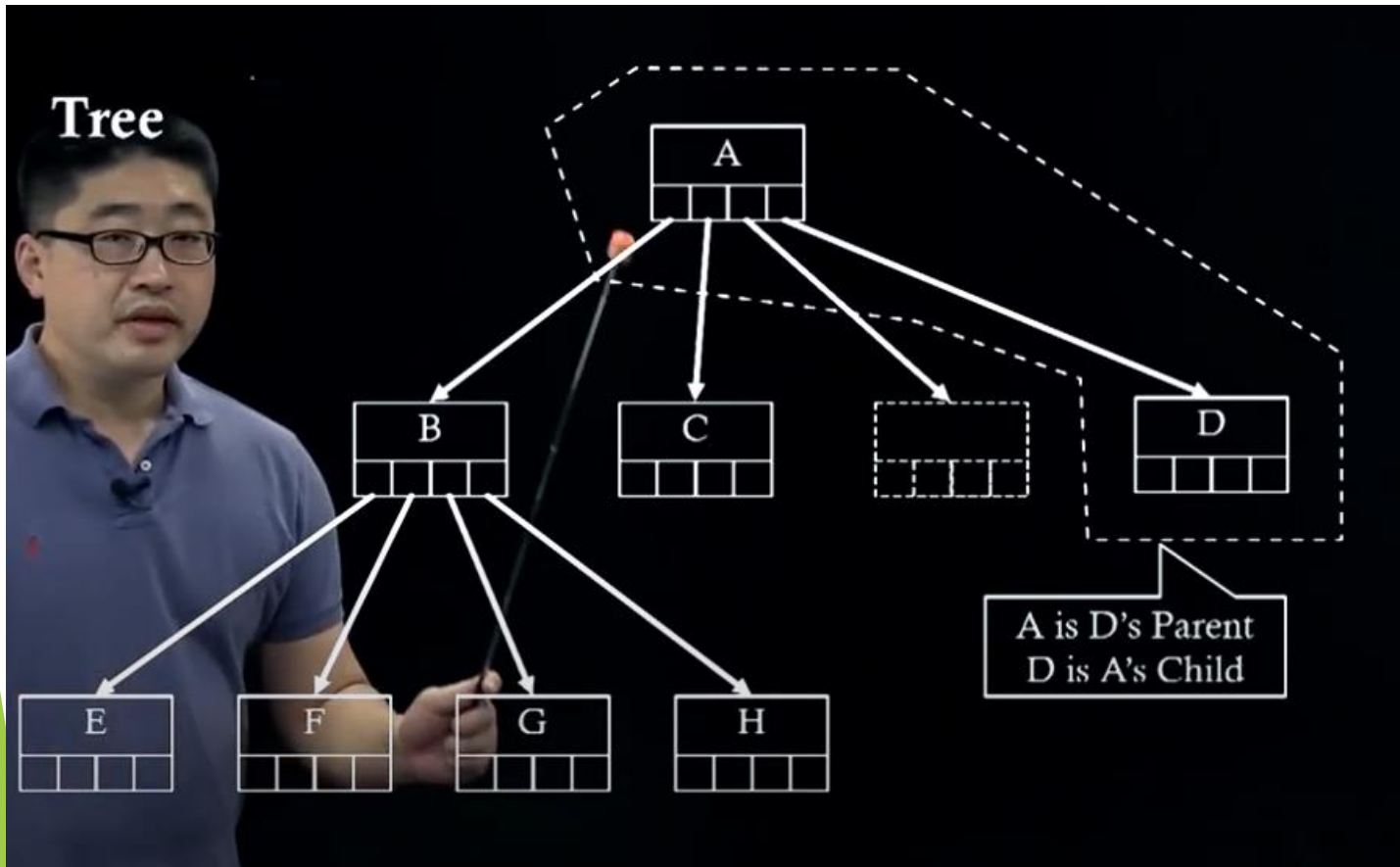
- **Nods** : 넥스트 레퍼런스가 다음 노드를 가르칠 때 이것을 **Node**
- **Edge** : 다음 노드를 가르치는 것
- **Root** : 맨 위에 있는 노드를 특정한 **Node** (Linked list에서 **head**와 **tail**을 부르는 것과 같다)
- **A is D's Parent** : A는 D의 **Parent**가 된다.
- **D is A's Child** : D는 A의 **Child**가 된다.
- **Siblings** : 동일한 레벨이 있는, 같은 **Parent**를 가진 노드 집합
- **Leaves or Terminal Nodes** : 노드들이 더 이상 넥스트 노드들 (garbage값을 가지고 있음)을 가지고 있지 않는 노드
- **Internal Nodes** : Leaves가 아닌 노드
- **Descendants** : A의 후손들, A node를 뺀 나머지
- **Ancestores of E** : E의 Ancestores들은 A와 B
- **Path to E** : root에서 edge를 통해 최단거리로 가는 길 (A-B-C)
- **Depth and level of B** : 특정 노드의 **depth**는 패스의 길이
- **High of Tree** : maximum path의 길이
- **Degree of B** : B의 노드가 가질 수 있는 **child**의 개수. 그래서 4개
- **Size of Tree** : tree에 있는 모든 노드의 개수

## 2. Terminologies of tree structure



- Nods : 넥스트 레퍼런스가 다음 노드를 가르칠 때 이것을 Node
- Edge : 다음 노드를 가르치는 것
- Root : 맨 위에 있는 노드를 특정한 Node (Linked list에서 head와 tail을 부르는 것과 같다)
- A is D's Parent : A는 D의 Parent가 된다.
- D is A's Child : D는 A의 Child가 된다.
- Siblings : 동일한 레벨이 있는, 같은 Parent를 가진 노드 집합
- Leaves or Terminal Nodes : 노드들이 더 이상 넥스트 노드들 (garbage값을 가지고 있음)을 가지고 있지 않는 노드
- Internal Nodes : Leaves가 아닌 노드
- Descendants : A의 후손들, A node를 뺀 나머지
- Ancestores of E : E의 Ancestores들은 A와 B
- Path to E : root에서 edge를 통해 최단거리로 가는 길 (A-B-C)
- Depth and level of B : 특정 노드의 depth는 패스의 길이
- High of Tree : maximum path의 길이
- Degree of B : B의 노드가 가질 수 있는 child의 개수. 그래서 4개
- Size of Tree : tree에 있는 모든 노드의 개수

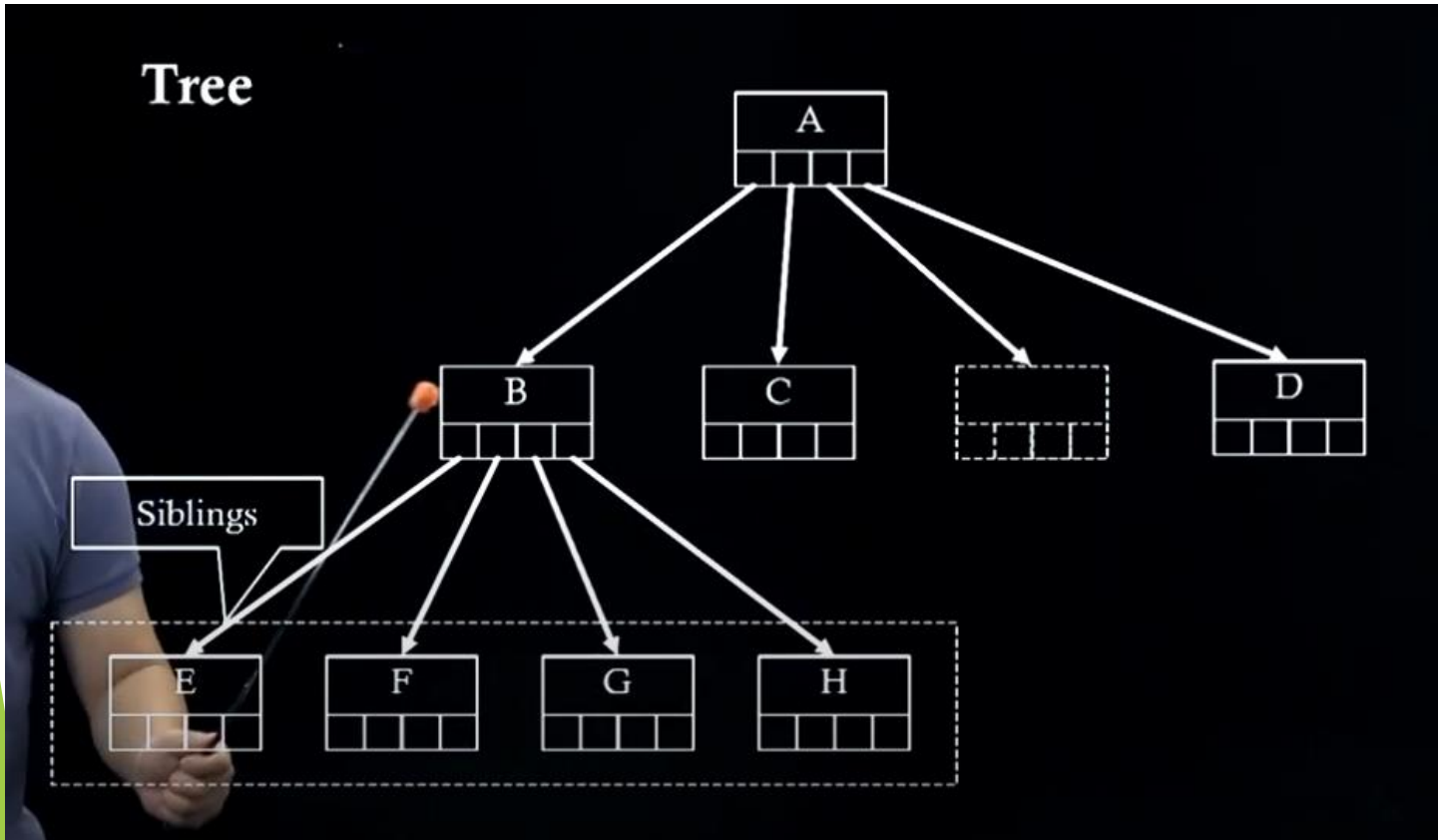
## 2. Terminologies of tree structure



- Nods : 넥스트 레퍼런스가 다음 노드를 가르칠 때 이것을 Node
- Edge : 다음 노드를 가르치는 것
- Root : 맨 위에 있는 노드를 특정한 Node (Linked list에서 head와 tail을 부르는 것과 같다)
- A is D's Parent : A는 D의 Parent가 된다.
- D is A's Child : D는 A의 Child가 된다.
- Siblings : 동일한 레벨이 있는, 같은 Parent를 가진 노드 집합
- Leaves or Terminal Nodes : 노드들이 더 이상 넥스트 노드들 (garbage값을 가지고 있음)을 가지고 있지 않는 노드
- Internal Nodes : Leaves가 아닌 노드
- Descendants : A의 후손들, A node를 뺀 나머지
- Ancestores of E : E의 Ancestores들은 A와 B
- Path to E : root에서 edge를 통해 최단거리로 가는 길 (A-B-C)
- Depth and level of B : 특정 노드의 depth는 패스의 길이
- High of Tree : maximum path의 길이
- Degree of B : B의 노드가 가질 수 있는 child의 개수. 그래서 4개
- Size of Tree : tree에 있는 모든 노드의 개수

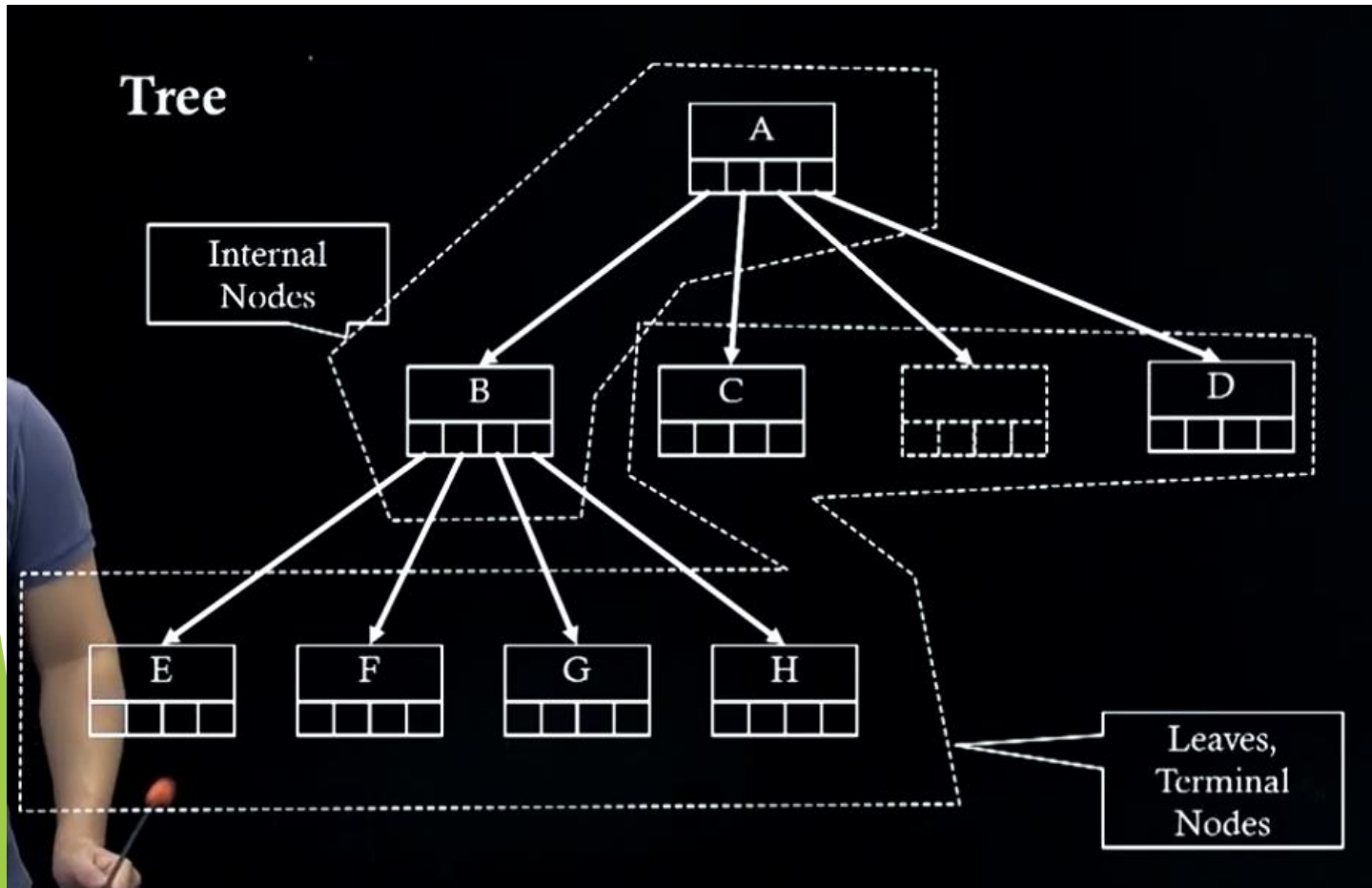


## 2. Terminologies of tree structure



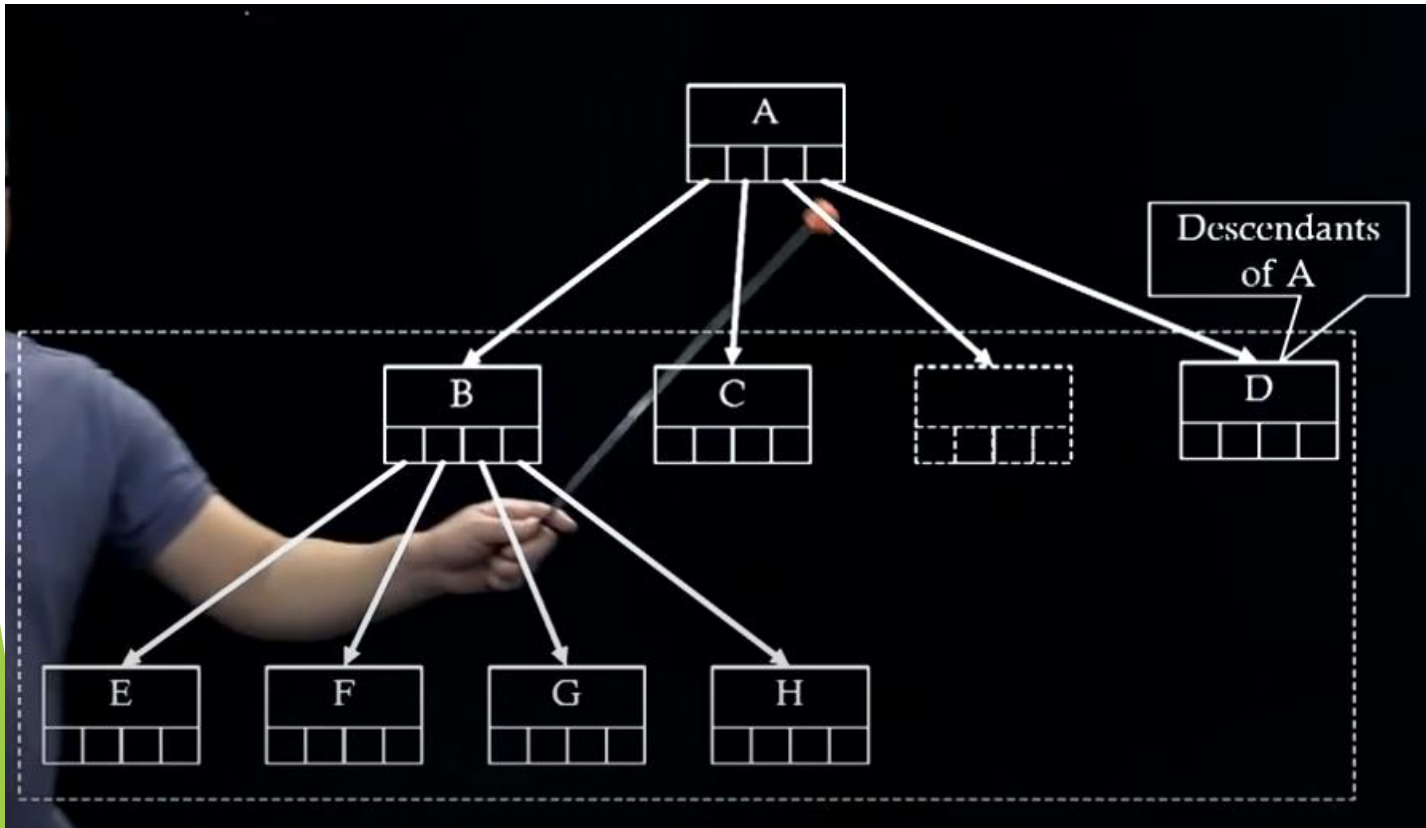
- **Nods :** 넥스트 레퍼런스가 다음 노드를 가르칠 때 이것을 **Node**
- **Edge :** 다음 노드를 가르치는 것
- **Root :** 맨 위에 있는 노드를 특정한 **Node** (Linked list에서 **head**와 **tail**을 부르는 것과 같다)
- **A is D's Parent :** A는 D의 **Parent**가 된다.
- **D is A's Child :** D는 A의 **Child**가 된다.
- **Siblings :** 동일한 레벨이 있는, 같은 **Parent**를 가진 노드 집합
- **Leaves or Terminal Nodes :** 노드들이 더 이상 넥스트 노드들 (garbage값을 가지고 있음)을 가지고 있지 않는 노드
- **Internal Nodes :** Leaves가 아닌 노드
- **Descendants :** A의 후손들, A node를 뺀 나머지
- **Ancestores of E :** E의 Ancestores들은 A와 B
- **Path to E :** root에서 edge를 통해 최단거리로 가는 길 (A-B-C)
- **Depth and level of B :** 특정 노드의 **depth**는 패스의 길이
- **High of Tree :** maximum path의 길이
- **Degree of B :** B의 노드가 가질 수 있는 **child**의 개수. 그래서 4개
- **Size of Tree :** tree에 있는 모든 노드의 개수

## 2. Terminologies of tree structure



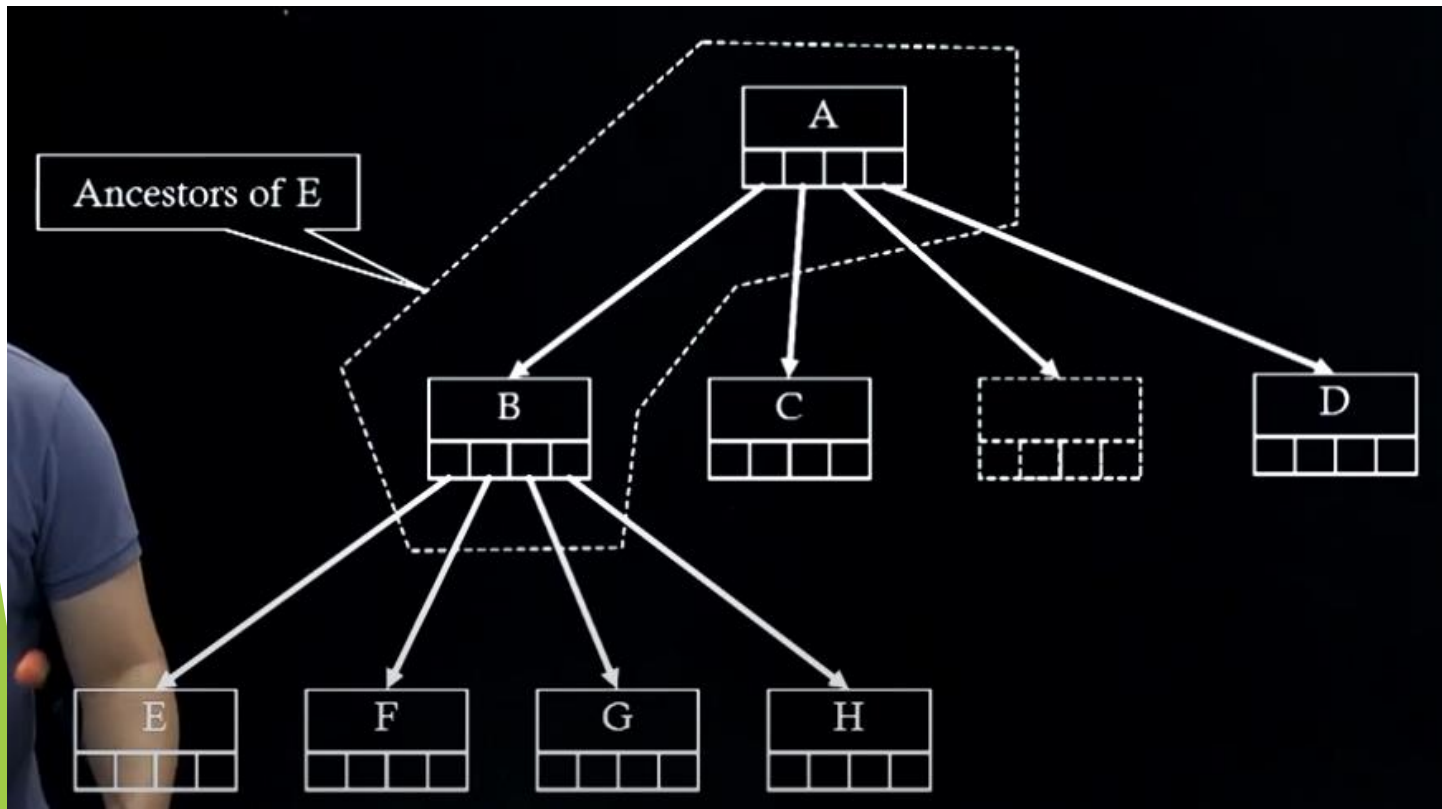
- Nods : 넥스트 레퍼런스가 다음 노드를 가르칠 때 이것을 Node
- Edge : 다음 노드를 가르치는 것
- Root : 맨 위에 있는 노드를 특정한 Node (Linked list에서 head와 tail을 부르는 것과 같다)
- A is D's Parent : A는 D의 Parent가 된다.
- D is A's Child : D는 A의 Child가 된다.
- Siblings : 동일한 레벨이 있는, 같은 Parent를 가진 노드 집합
- Leaves or Terminal Nodes : 노드들이 더 이상 넥스트 노드들 (garbage값을 가지고 있음)을 가지고 있지 않는 노드
- Internal Nodes : Leaves가 아닌 노드
- Descendants : A의 후손들, A node를 뺀 나머지
- Ancestores of E : E의 Ancestores들은 A와 B
- Path to E : root에서 edge를 통해 최단거리로 가는 길 (A-B-C)
- Depth and level of B : 특정 노드의 depth는 패스의 길이
- High of Tree : maximum path의 길이
- Degree of B : B의 노드가 가질 수 있는 child의 개수. 그래서 4개
- Size of Tree : tree에 있는 모든 노드의 개수

## 2. Terminologies of tree structure



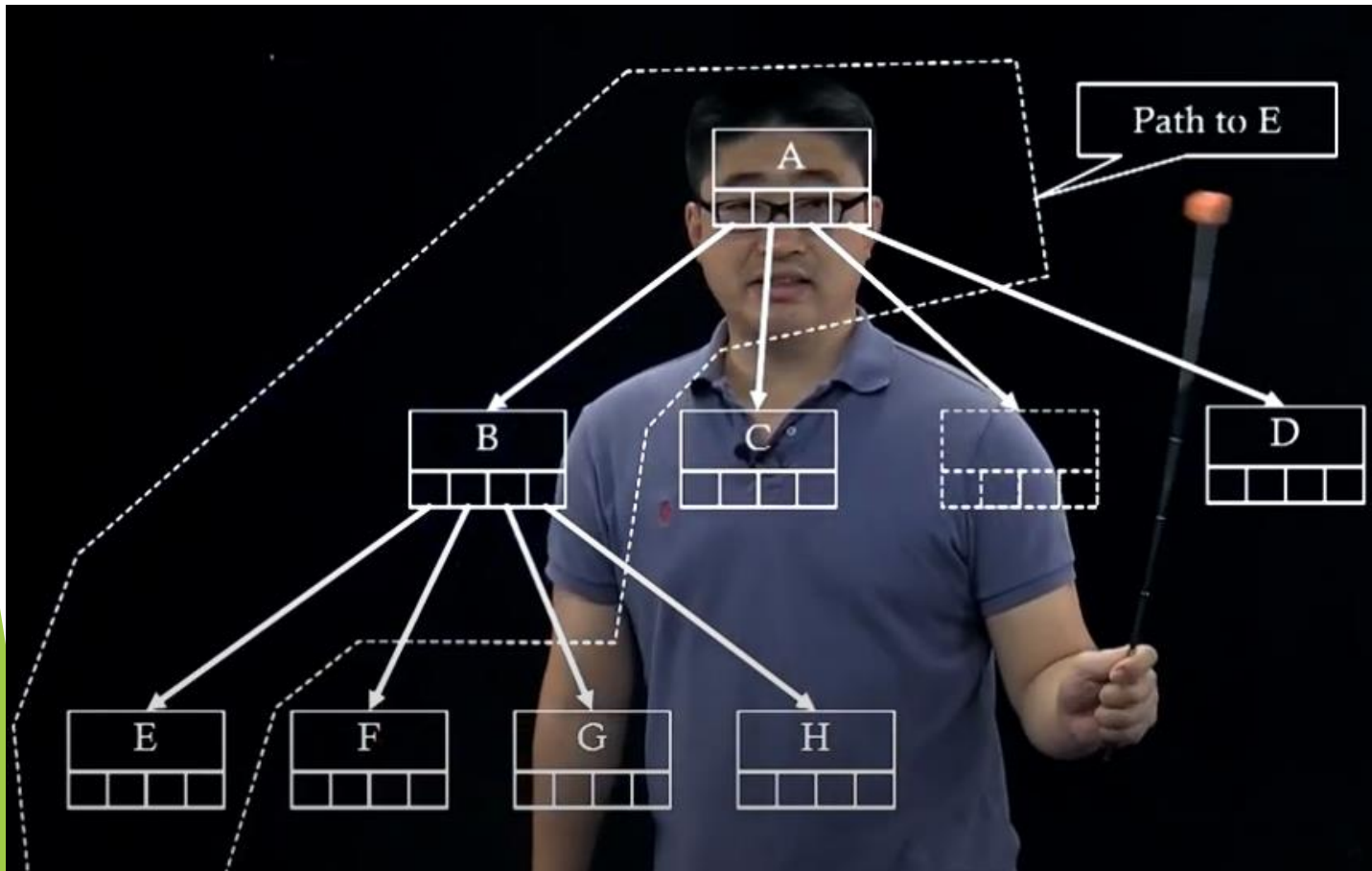
- Nods : 넥스트 레퍼런스가 다음 노드를 가르칠 때 이것을 Node
- Edge : 다음 노드를 가르치는 것
- Root : 맨 위에 있는 노드를 특정한 Node (Linked list에서 head와 tail을 부르는 것과 같다)
- A is D's Parent : A는 D의 Parent가 된다.
- D is A's Child : D는 A의 Child가 된다.
- Siblings : 동일한 레벨이 있는, 같은 Parent를 가진 노드 집합
- Leaves or Terminal Nodes : 노드들이 더 이상 넥스트 노드들 (garbage값을 가지고 있음)을 가지고 있지 않는 노드
- Internal Nodes : Leaves가 아닌 노드
- Descendants : A의 후손들, A node를 뺀 나머지
- Ancestores of E : E의 Ancestores들은A와 B
- Path to E : root에서 edge를 통해 최단거리로 가는 길(A-B-C)
- Depth and level of B : 특정 노드의 depth는 패스의 길이
- High of Tree : maximum path의 길이
- Degree of B : B의 노드가 가질 수 있는 child의 개수. 그래서 4개
- Size of Tree : tree에 있는 모든 노드의 개수

## 2. Terminologies of tree structure



- Nods : 넥스트 레퍼런스가 다음 노드를 가르칠 때 이것을 Node
- Edge : 다음 노드를 가르치는 것
- Root : 맨 위에 있는 노드를 특정한 Node (Linked list에서 head와 tail을 부르는 것과 같다)
- A is D's Parent : A는 D의 Parent가 된다.
- D is A's Child : D는 A의 Child가 된다.
- Siblings : 동일한 레벨이 있는, 같은 Parent를 가진 노드 집합
- Leaves or Terminal Nodes : 노드들이 더 이상 넥스트 노드들 (garbage값을 가지고 있음)을 가지고 있지 않는 노드
- Internal Nodes : Leaves가 아닌 노드
- Descendants : A의 후손들, A node를 뺀 나머지
- Ancestores of E : E의 Ancestores들은A와 B
- Path to E : root에서 edge를 통해 최단거리로 가는 길(A-B-C)
- Depth and level of B : 특정 노드의 depth는 패스의 길이
- High of Tree : maximum path의 길이
- Degree of B : B의 노드가 가질 수 있는 child의 개수. 그래서 4개
- Size of Tree : tree에 있는 모든 노드의 개수

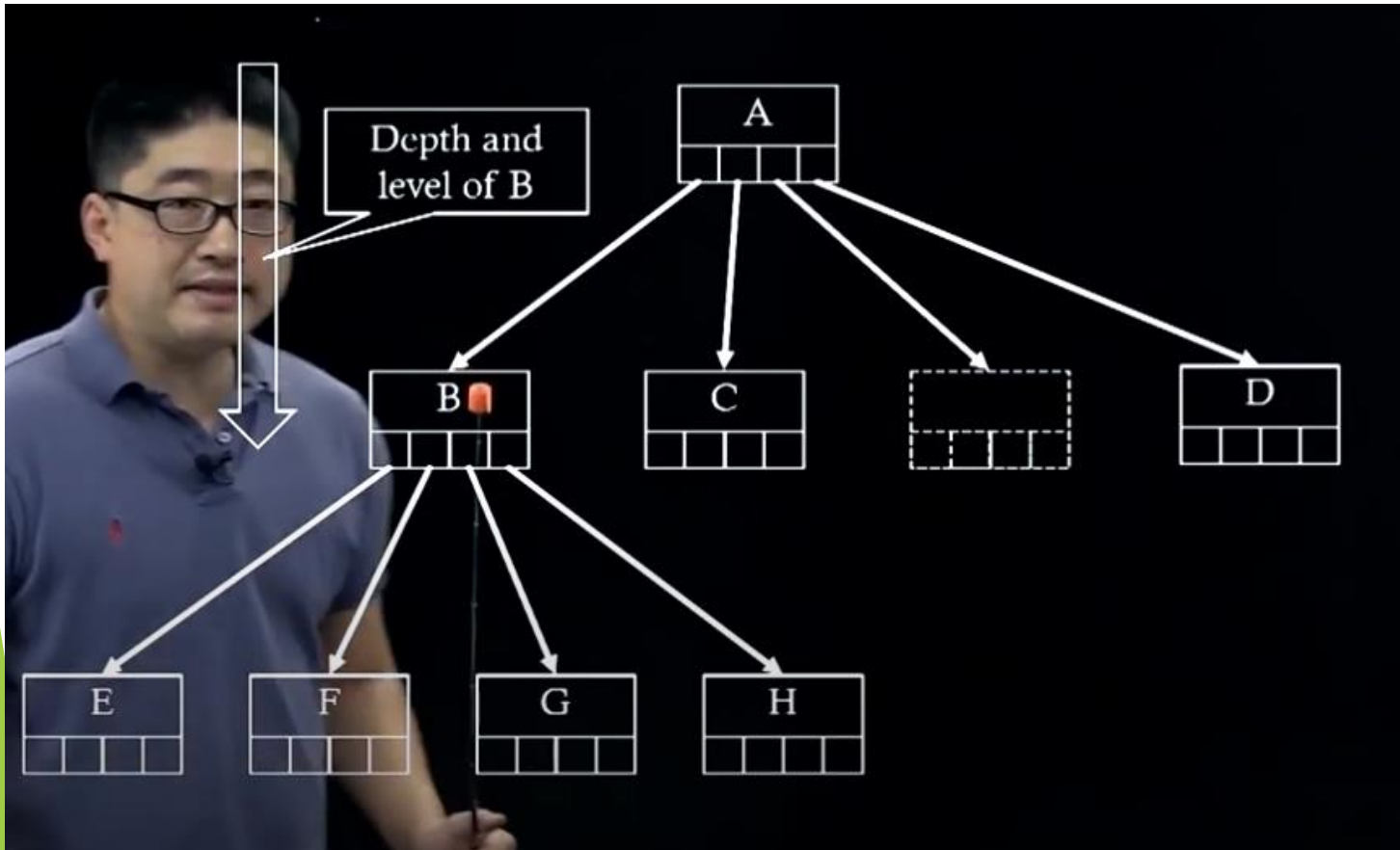
## 2. Terminologies of tree structure



- Nods : 넥스트 레퍼런스가 다음 노드를 가르칠 때 이것을 Node
- Edge : 다음 노드를 가르치는 것
- Root : 맨 위에 있는 노드를 특정한 Node (Linked list에서 head와 tail을 부르는 것과 같다)
- A is D's Parent : A는 D의 Parent가 된다.
- D is A's Child : D는 A의 Child가 된다.
- Siblings : 동일한 레벨이 있는, 같은 Parent를 가진 노드 집합
- Leaves or Terminal Nodes : 노드들이 더 이상 넥스트 노드들 (garbage값을 가지고 있음)을 가지고 있지 않는 노드
- Internal Nodes : Leaves가 아닌 노드
- Descendants : A의 후손들, A node를 뺀 나머지
- Ancestores of E : E의 Ancestores들은A와 B
- Path to E : root에서 edge를 통해 최단거리로 가는 길(A-B-C)
- Depth and level of B : 특정 노드의 depth는 패스의 길이
- High of Tree : maximum path의 길이
- Degree of B : B의 노드가 가질 수 있는 child의 개수. 그래서 4개
- Size of Tree : tree에 있는 모든 노드의 개수

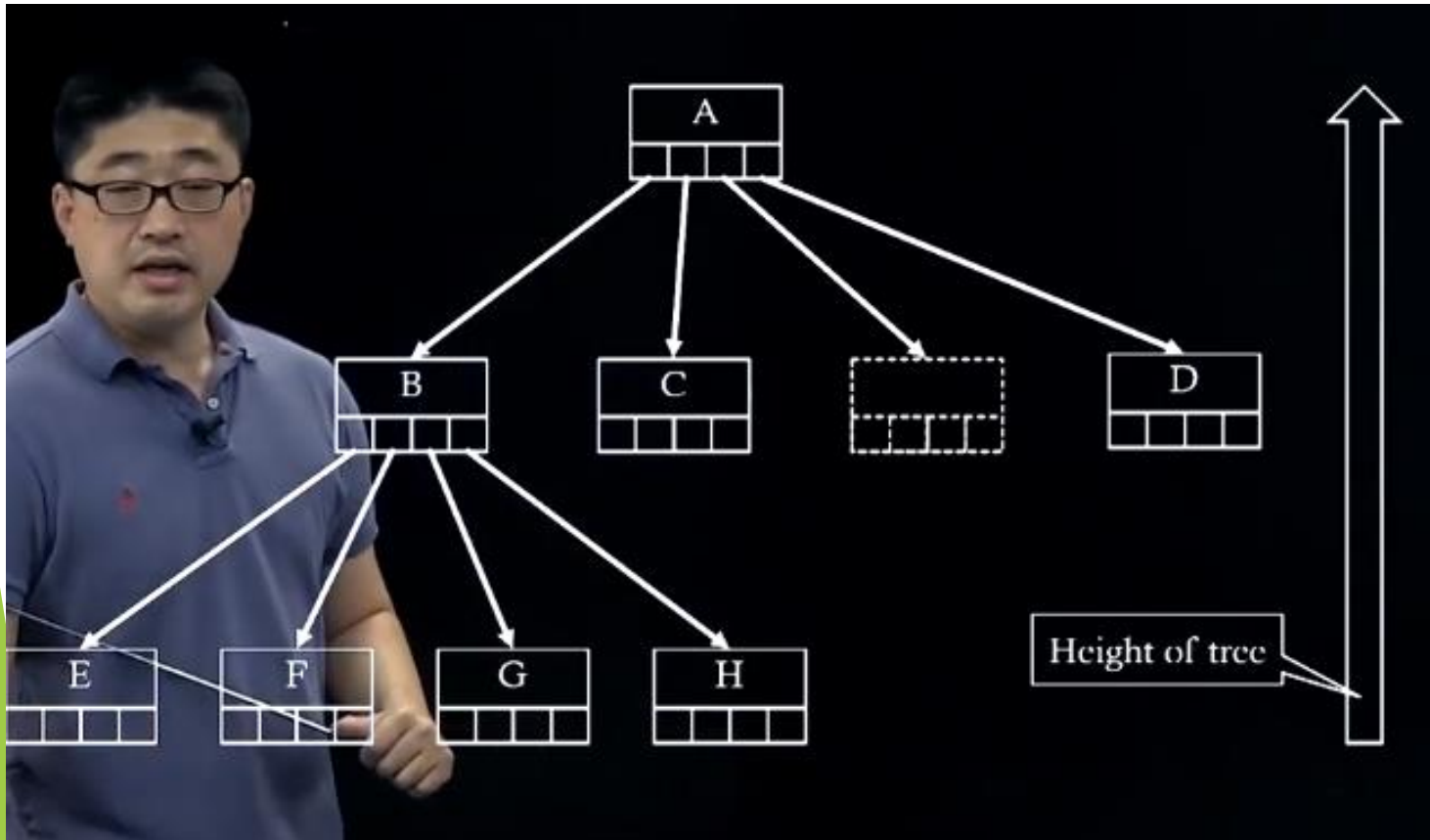


## 2. Terminologies of tree structure



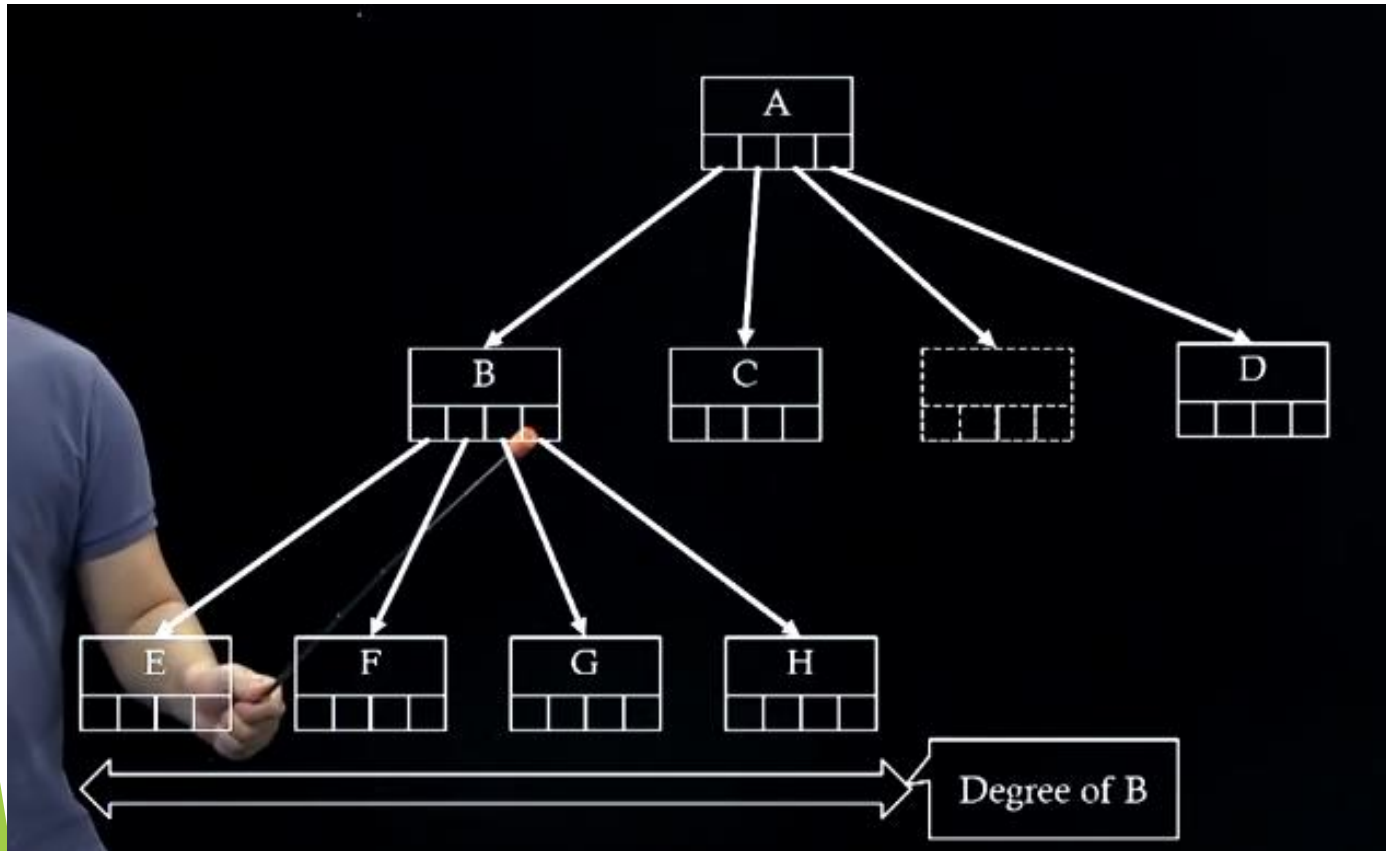
- Nods : 넥스트 레퍼런스가 다음 노드를 가르칠 때 이것을 Node
- Edge : 다음 노드를 가르치는 것
- Root : 맨 위에 있는 노드를 특정한 Node (Linked list에서 head와 tail을 부르는 것과 같다)
- A is D's Parent : A는 D의 Parent가 된다.
- D is A's Child : D는 A의 Child가 된다.
- Siblings : 동일한 레벨이 있는, 같은 Parent를 가진 노드 집합
- Leaves or Terminal Nodes : 노드들이 더 이상 넥스트 노드들 (garbage값을 가지고 있음)을 가지고 있지 않는 노드
- Internal Nodes : Leaves가 아닌 노드
- Descendants : A의 후손들, A node를 뺀 나머지
- Ancestores of E : E의 Ancestores들은 A와 B
- Path to E : root에서 edge를 통해 최단거리로 가는 길 (A-B-C)
- Depth and level of B : 특정 노드의 depth는 패스의 길이
- High of Tree : maximum path의 길이
- Degree of B : B의 노드가 가질 수 있는 child의 개수. 그래서 4개
- Size of Tree : tree에 있는 모든 노드의 개수

## 2. Terminologies of tree structure



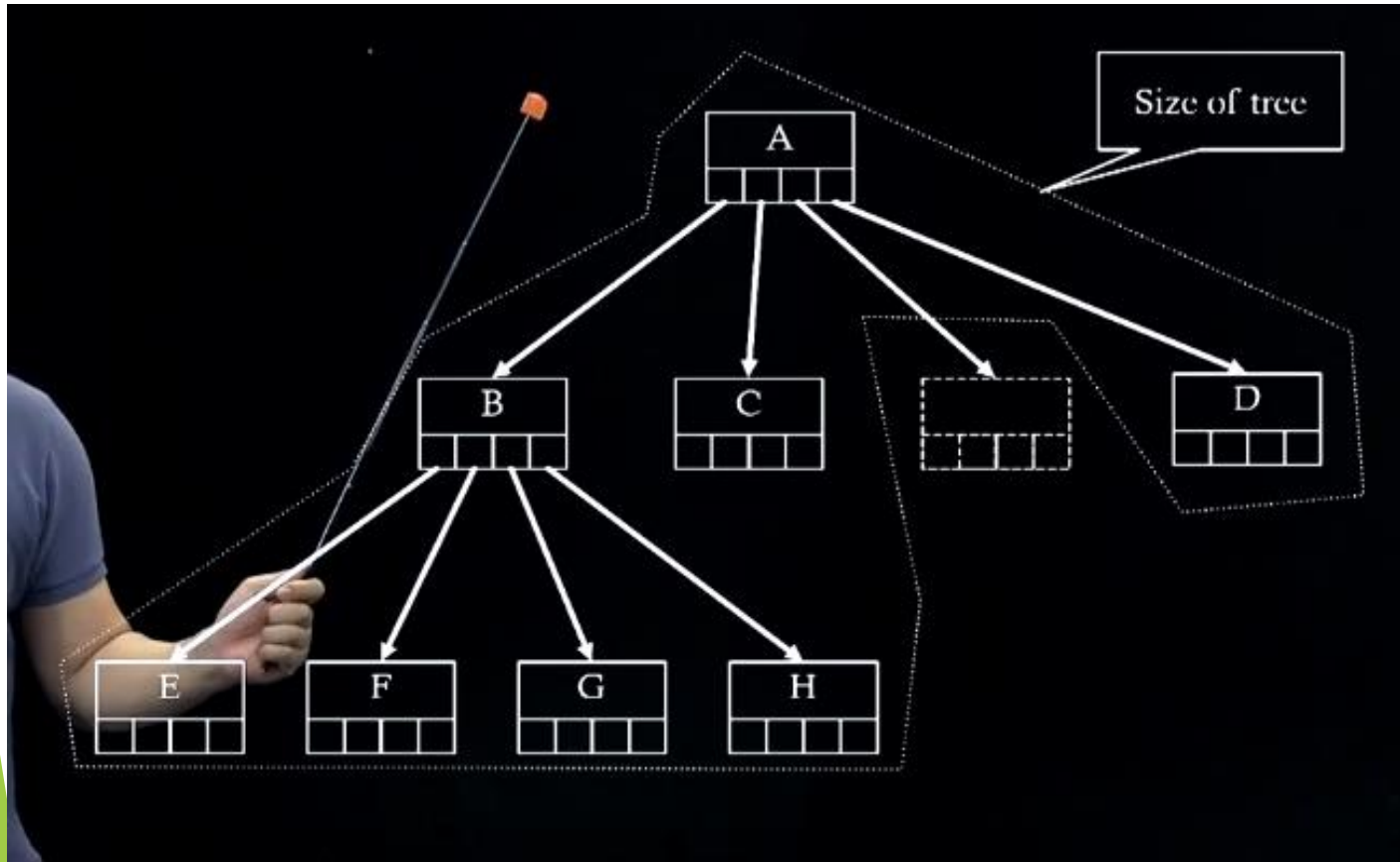
- **Nods** : 넥스트 레퍼런스가 다음 노드를 가르칠 때 이것을 **Node**
- **Edge** : 다음 노드를 가르치는 것
- **Root** : 맨 위에 있는 노드를 특정한 **Node** (Linked list에서 **head**와 **tail**을 부르는 것과 같다)
- **A is D's Parent** : A는 D의 **Parent**가 된다.
- **D is A's Child** : D는 A의 **Child**가 된다.
- **Siblings** : 동일한 레벨이 있는, 같은 **Parent**를 가진 노드 집합
- **Leaves or Terminal Nodes** : 노드들이 더 이상 넥스트 노드들 (garbage값을 가지고 있음)을 가지고 있지 않는 노드
- **Internal Nodes** : Leaves가 아닌 노드
- **Descendants** : A의 후손들, A node를 뺀 나머지
- **Ancestores of E** : E의 Ancestores들은 A와 B
- **Path to E** : root에서 edge를 통해 최단거리로 가는 길 (A-B-C)
- **Depth and level of B** : 특정 노드의 **depth**는 패스의 길이
- **High of Tree** : maximum path의 길이
- **Degree of B** : B의 노드가 가질 수 있는 **child**의 개수. 그래서 4개
- **Size of Tree** : tree에 있는 모든 노드의 개수

## 2. Terminologies of tree structure



- Nods : 넥스트 레퍼런스가 다음 노드를 가르칠 때 이것을 Node
- Edge : 다음 노드를 가르치는 것
- Root : 맨 위에 있는 노드를 특정한 Node (Linked list에서 head와 tail을 부르는 것과 같다)
- A is D's Parent : A는 D의 Parent가 된다.
- D is A's Child : D는 A의 Child가 된다.
- Siblings : 동일한 레벨이 있는, 같은 Parent를 가진 노드 집합
- Leaves or Terminal Nodes : 노드들이 더 이상 넥스트 노드들 (garbage값을 가지고 있음)을 가지고 있지 않는 노드
- Internal Nodes : Leaves가 아닌 노드
- Descendants : A의 후손들, A node를 뺀 나머지
- Ancestores of E : E의 Ancestores들은 A와 B
- Path to E : root에서 edge를 통해 최단거리로 가는 길 (A-B-C)
- Depth and level of B : 특정 노드의 depth는 패스의 길이
- High of Tree : maximum path의 길이
- Degree of B : B의 노드가 가질 수 있는 child의 개수. 그래서 4개
- Size of Tree : tree에 있는 모든 노드의 개수

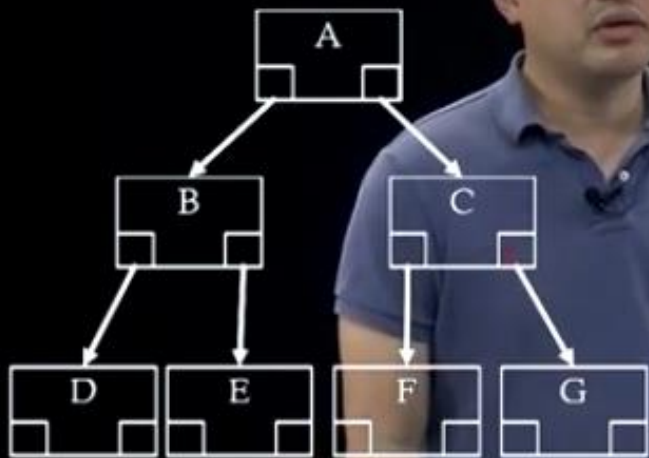
## 2. Terminologies of tree structure



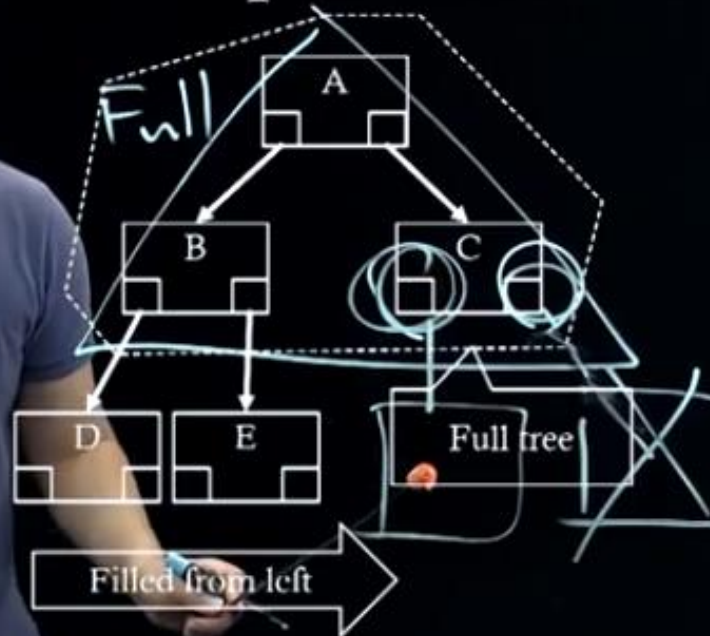
- Nods : 넥스트 레퍼런스가 다음 노드를 가르칠 때 이것을 Node
- Edge : 다음 노드를 가르치는 것
- Root : 맨 위에 있는 노드를 특정한 Node (Linked list에서 head와 tail을 부르는 것과 같다)
- A is D's Parent : A는 D의 Parent가 된다.
- D is A's Child : D는 A의 Child가 된다.
- Siblings : 동일한 레벨이 있는, 같은 Parent를 가진 노드 집합
- Leaves or Terminal Nodes : 노드들이 더 이상 넥스트 노드들 (garbage값을 가지고 있음)을 가지고 있지 않는 노드
- Internal Nodes : Leaves가 아닌 노드
- Descendants : A의 후손들, A node를 뺀 나머지
- Ancestores of E : E의 Ancestores들은 A와 B
- Path to E : root에서 edge를 통해 최단거리로 가는 길(A-B-C)
- Depth and level of B : 특정 노드의 depth는 패스의 길이
- High of Tree : maximum path의 길이
- Degree of B : B의 노드가 가질 수 있는 child의 개수. 그래서 4개
- Size of Tree : tree에 있는 모든 노드의 개수

## 2. Terminologies of tree structure

**Full Tree**



**Complete Tree**



- Full Tree :

- 1) depth가 동일함, internal node은 완전히 모든 다음 노드들을 가르키고 있다.
- 2) Internal node 들은 완전히(fully) 노드들은 가지고 있다.
- 3) Terminal node들은 완전히(fully) 넥스트 노드들을 가지고 있지 않다.
- 4) 이렇게 삼각형이 된 트리를 full tree라고 한다.

- Complete tree :

- 1) 바로 직전 덤스까지는 full tree structure이다.
- 2) 맨 왼쪽에서 다음 노드들을 하나하나씩 채워 나갈때 complete tree라고 한다.
- 3) 오른쪽에 먼저 노드가 생기면 complete tree가 아니다.



### 3. Characteristics of Tree

◇ (Num. of edges) =  
(Num. of nodes) - 1

◇ Depth of root  
◇ 0

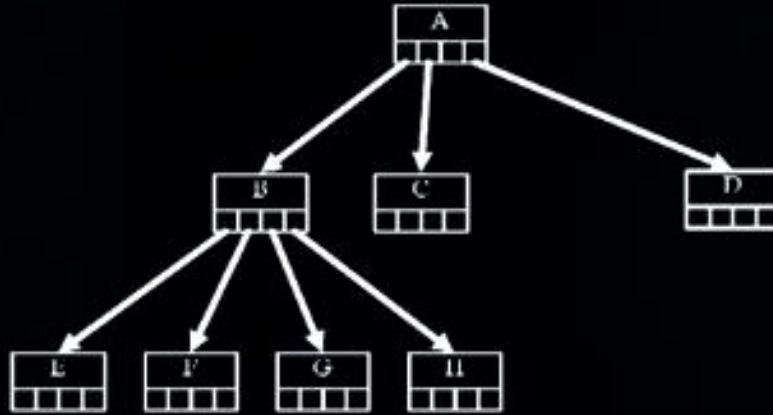
◇ Height of root  
= height of tree

◇ (Maximum num. of nodes at level  $i$  with degree  $d$ )  
=  $d^i$

◇ (Maximum num. of leaves with height  $h$  and degree  $d$ )  
=  $d^h$

◇ (Maximum size of a tree with height  $h$  and degree  $d$ )  
=  $1 + d + d^2 + \dots + d^h = \frac{d^{h+1} - 1}{d - 1}$

◇ (Height of a **complete** tree with size  $s$  and degree  $d$ )  
=  $\lceil \log_d(s(d - 1) + 1) \rceil - 1$



1. 전체 노드에서 root를 빼면 전체 edge가 된다.
2. Path의 길이인데 root는 path가 없다.
3. Root의 height는 tree의 height가 된다.
4. Level 1이니 d니깐 4의 1승이 되다. ??
5. Height  $h$ 는 2고, degree  $d$ 의 maximum node는 4의 2승
6. 공식에 따라서 구하면 된다.
7. 공식에 따라서 구하면 된다.

# 4. Binary Search Tree and Implementation

Binary search tree: a simple structure

## ◆ Binary tree

◆ Tree with degree 2

## ◆ Binary search tree

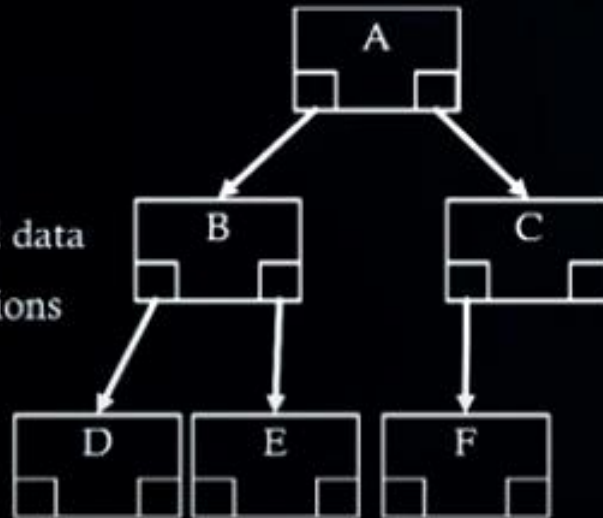
◆ Tree with degree 2

◆ Tree designed for a fast search of stored data

◆ So far, what we have studied the definitions and the characteristics of stored data

◆ Now, this is related to the operations

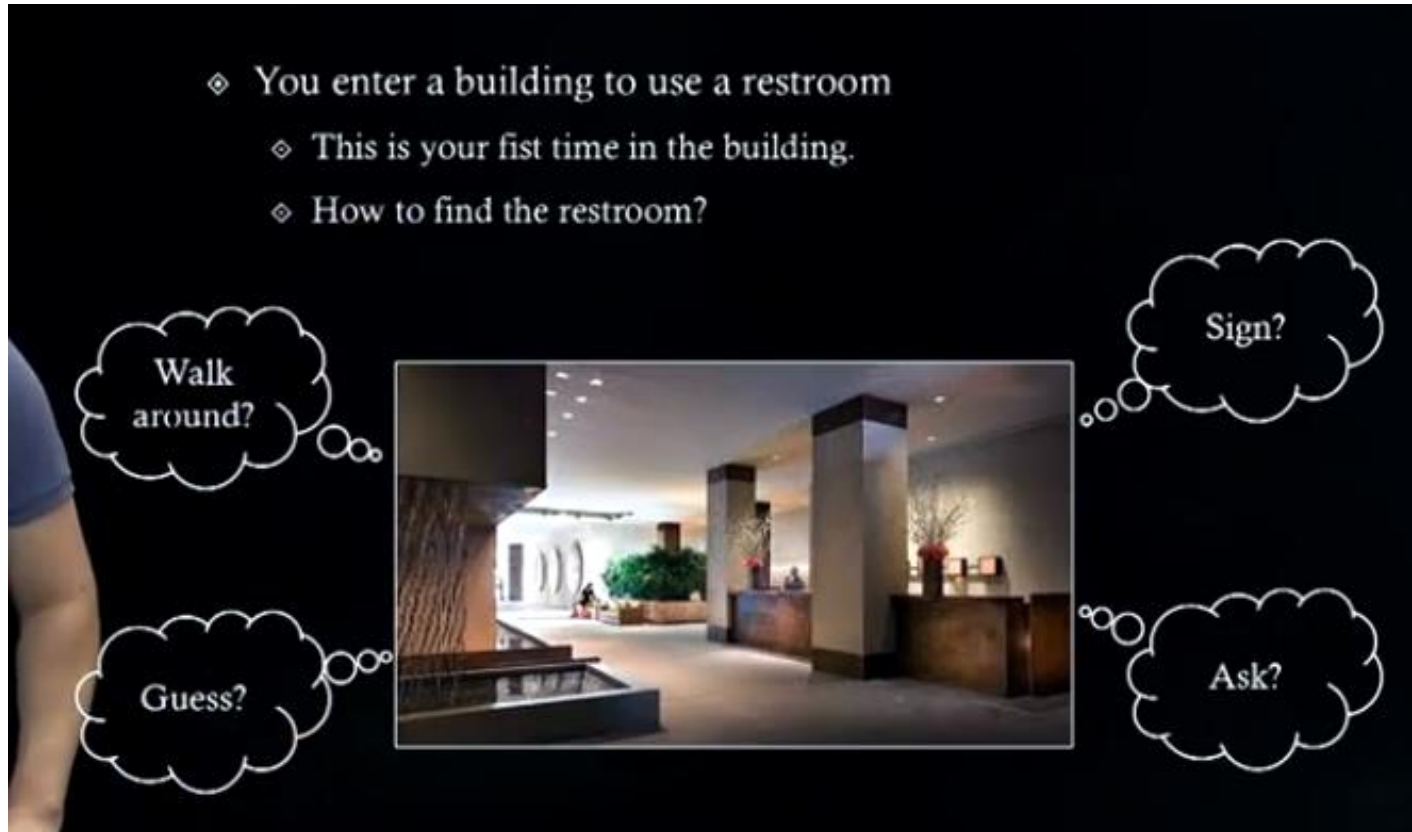
◆ How to perform a faster search?



1. BST라고도 부른다.
2. Tree중에 가장 간단한 tree이다.
3. Degree가 2인 tree
4. Stored data에 fast search를 위해 설계
5. Search를 빠르게 하기 위해 data의 어떤 특성을 활용해서 구조를 만드는 것

# 4. Binary Search Tree and Implementation

## Detour: Intuitive Analogy - Finding Restroom in Building



1. 화장실을 가기 위해 둘러보거나 표지판을 보거나 물어보거나 추측해 본다.
2. **Guess**를 많이 한다. 규칙을 알 수 있기 때문.
3. 그래서 데이터의 규칙을 넣으면 **Guess**를 통해서 데이터를 **search**하는데 많은 시간을 줄일 수 있다.

# 4. Binary Search Tree and Implementation

A scenario of using binary search tree

**Bank Account Management System**

Account #	Name	Amount	Type
1001	Smith	100,000	Simple Interest
1002	Koh	50,000	Compound Interest
1003	Moon	10,000	Simple Interest
1004	Kim	30,000	No Interest

*Finding 1004 in the two abstract data types*

**Linked List**

1001 → 1002 → 1003 → 1004

4 retrievals

**Binary Search Tree**

1. 1004를 찾는 bank account
2. Linked list는 4번의 retrievals를 해야함
3. 규칙 - 오른쪽 넥스트는 큰값을 저장, 왼쪽은 작은 값
4. 1003이라는 루트를 만나면 찾으려고하는 1004는 오른쪽에 있음을 안다.

# 4. Binary Search Tree and Implementation

## Implementation of tree node

### Implementation of tree node



#### ◆ Has three references

- ◆ Left hand side (LHS)
- ◆ Right hand side (RHS)
- ◆ Its own value
- ◆ Its parent node
- ◆ Not implemented here, but
  - ◆ LHS stores
    - ◆ Values have lower than its own value
  - ◆ RHS stores
    - ◆ Values have higher than its own value
  - ◆ Just as we all know that the department stores do not have a restroom on the first floor

#### ◆ Other than four references,

- ◆ Simple get/set methods
  - ◆ What are the get/set methods?
  - ◆ Coming from encapsulation

```
@class TreeNode:
    nodeLHS = ''
    nodeRHS = ''
    nodeParent = ''
    value = ''

    def __init__(self, value, nodeParent):
        self.value = value
        self.nodeParent = nodeParent

    def getLHS(self):
        return self.nodeLHS

    def getRHS(self):
        return self.nodeRHS

    def getValue(self):
        return self.value

    def getParent(self):
        return self.nodeParent

    def setLHS(self, LHS):
        self.nodeLHS = LHS

    def setRHS(self, RHS):
        self.nodeRHS = RHS

    def setValue(self, value):
        self.value = value

    def setParent(self, nodeParent):
        self.nodeParent = nodeParent
```

} Four references

1. Parent로 가는 node(previous같은 경우)
2. Left hand side : 항상 작은 값
3. Right hand side : 항상 큰 값
4. Value : 자기 자신의 value
5. 이런 레퍼런스들을 다양하게 get과 set할 수 있는 클래스에서 가져올 수 있고 정할 수 있는 메소드를 정리



# 4. Binary Search Tree and Implementation

## Implementation of BST

Implementation of BST

```
class BinarySearchTree:
    root = ''

    def __init__(self):
        pass

    def insert(self, value, node = ''):
        pass

    def search(self, value, node = ''):
        pass

    def delete(self, value, node = ''):
        pass

    def findMax(self, node = ''):
        pass

    def findMin(self, node = ''):
        pass

    def traverseLevelOrder(self):
        pass

    def traverseInOrder(self, node = ''):
        pass

    def traversePreOrder(self, node = ''):
        pass

    def traversePostOrder(self, node = ''):
        pass
```

- ◆ BST handles the data stored through its root
  - ◆ Root has its own value
  - ◆ Tree instance access to the root
  - ◆ Only through the root, the tree instances access to the descendant nodes of the root

1. 바이너리 서치 트리는 기본적으로 전체 노드에 대한 레퍼런스를 저장할 수 없다
2. Root에 대해서만 레퍼런스를 저장하게 된다.
3. Root에서 search해 나가면서 값들을 insert, delete, search를 한다.
4. 다양한 operations를 정의를 해놓음
5. 이런것들을 하나하나씩 채워나가면서 구현

# 5. Insert and Search Operation of Binary Search Tree

## Insert operation of binary search tree

### ◆ Insertion operation

- ◆ Retrieve the current node value
  - ◆ If the value is equal to the value to insert
    - ◆ Return already there!
  - ◆ If the value is smaller than the value to insert
    - ◆ If there is a node in the right hand-side (RHS), then move to the RHS node (*Recursion*)
    - ◆ If there is no node in RHS, create a RHS node with the value to insert
  - ◆ If the value is larger than the value to insert
    - ◆ If there is a node in the left hand-side (LHS), then move to the LHS node (*Recursion*)
    - ◆ If there is no node in LHS, create a LHS node with the value to insert

```
def insert(self, value, node = ''):
    if node == '':
        node = self.root
    if self.root == '':
        self.root = TreeNode(value, '')
        return
    if value == node.getValue():
        return
    if value > node.getValue():
        if node.getRHS() == '':
            node.setRHS(TreeNode(value, node))
        else:
            self.insert(value, node.getRHS())
    if value < node.getValue():
        if node.getLHS() == '':
            node.setLHS(TreeNode(value, node))
        else:
            self.insert(value, node.getLHS())
    return
```

Insert numbers: 3, 2, 0, 5, 7, 4.....

Windows 정품 인증  
[설정]으로 이동하여 Wind

1. Insert를 3, 2, 0, 5, 7, 4.. 로 입력
2. 두번의 recursion 평선 콜이 있다.
3. Return하는 부분이 escape문
4. Node가 정의되어 있지 않으면 root를 받아옴
5. 노드의 value보다 저장하려는 value가 크면 RHS, 작으면 LHS

# 5. Insert and Search Operation of Binary Search Tree

## Search operation of binary search tree

### Search operation

#### Retrieve the current node value

◇ If the value is equal to the value to search

◇ Return **TRUE**

◇ If the value is smaller than the value to search

◇ If there is a node in the right hand-side (RHS), then move to the RHS node (*Recursion*)

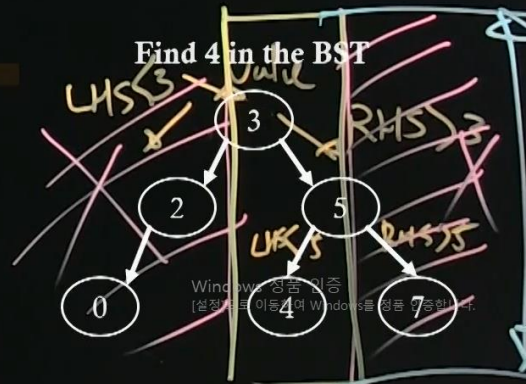
◇ If there is no node in RHS, return **FALSE**

◇ If the value is larger than the value to search

◇ If there is a node in the left hand-side (LHS), then move to the LHS node (*Recursion*)

◇ If there is no node in LHS, return **FALSE**

```
def search(self, value, node = ''):
    if node == '':
        node = self.root
    if value == node.getValue():
        return True
    if value > node.getValue():
        if node.getRHS() == '':
            return False
        else:
            return self.search(value, node.getRHS())
    if value < node.getValue():
        if node.getLHS() == '':
            return False
        else:
            return self.search(value, node.getLHS())
```



1. Insert에 이미 규칙이 있기 때문에 3에서 LHS는 무시하게 된다.
2. 5에서도 위의 과정을 반복한다.
3. Likend list보다 binary tree가 search 성능이 더 뛰어나다
4. 초기화- 값이 없으면 root값으로
5. 그다음 escape 문
6. 현재 노드의 값보다 value가 크면 RHS
7. 현재 노드의 값보다 value가 작으면 LHS
8. 다음 노드가 없으면 False: 찾는 값 없음



# 6. Delete Operation and Minimum & Maximum of Binary Search Tree

## Delete operation of binary search tree (1)

- First, you need to find the node to delete through recursions

### Three deletion cases

#### Case 1: deleting a node with no children

- Just remove the node by modifying its parent

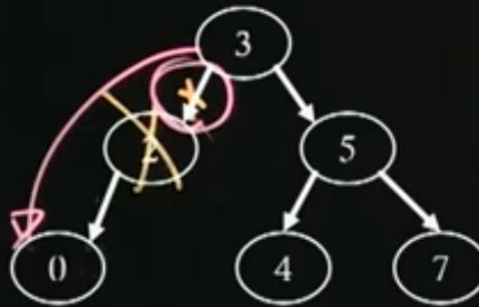
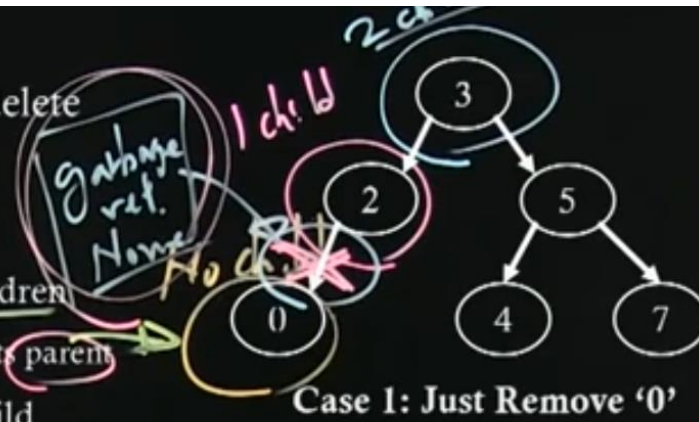
#### Case 2: deleting a node with one child

- Replace the node with the child

#### Case 3: deleting a node with two children

- Find either

- A maximum in the LHS or A minimum in the RHS
- Substitute the node to delete with the found value
- Delete the found node in the LHS or the RHS



Case 2: Replace '2' with '0'

Windows 정품 인증  
[설정]으로 이동하여 Windows를 정품 인증합니다.

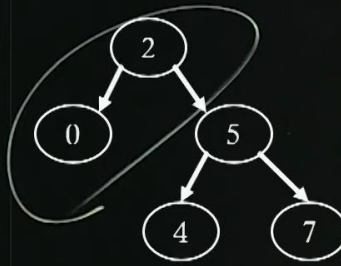
- 0은 nochild라 delet하기 쉽다.
- 2는 1child라 child를 tree 어딘가에 남겨줘야한다.
- 3은 2child라 2와5를 어떻게 남겨야할지 고민해야 함
- Tree deletion cases
- Case1: 지우고자 하는 parents 노드에 가서 막아 버리면 된다.(linked list랑 비슷함)
- Case2: 2로가는 reference를 차단하고 0으로 바로 연결한다.

# 6. Delete Operation and Minimum & Maximum of Binary Search Tree

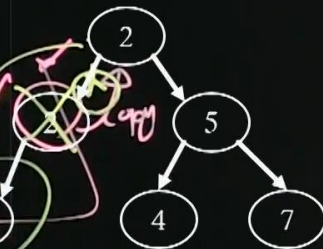
## Delete operation of binary search tree (2)

Delete operation of binary search tree (2)

```
def delete(self, value, node = ''):
    if node == '':
        node = self.root
    if node.getValue() < value:
        return self.delete(value, node.getRHS())
    if node.getValue() > value:
        return self.delete(value, node.getLHS())
    if node.getValue() == value:
        if node.getLHS() != '' and node.getRHS() != '':
            nodeMin = self.findMin( node.getRHS() )
            node.setValue(nodeMin.getValue())
            self.delete(nodeMin.getValue(), node.getRHS())
            return
        parent = node.getParent()
        if node.getLHS() != '':
            if node == self.root:
                self.root = node.getLHS()
            elif parent.getLHS() == node:
                parent.setLHS(node.getLHS())
            else:
                parent.setRHS(node.getLHS())
            node.getLHS().setParent(parent)
        else:
            parent.setRHS(node.getRHS())
            node.getRHS().setParent(parent)
        return
    if node.getRHS() != '':
        if node == self.root:
            self.root = node.getRHS()
        elif parent.getLHS() == node:
            parent.setLHS(node.getRHS())
        else:
            parent.setRHS(node.getRHS())
        node.getRHS().setParent(parent)
    return
```



Delete '2' in LHS



Windows 정품 인증  
[설정]으로 이동하여 Windows를 정품 인증합니다.

Case 3: Replace '3' with 'X'

1. Case3: 바로 밑에 있는 노드를 올리는것은 적용되지 않는다.
2. RHS의 minimum이 중간에 가까움
3. 3의 오른쪽에서 계속 왼쪽으로 내려가면 미니멈
4. LHS의 maximum이 중간에 가까움
5. 3의 왼쪽에서 계속 오른쪽으로 내려가면 맥시멈
6. Minimum을 3으로 copy함
7. 4는 nochild거나 1child임을 보장되어 있다.
8. Nochild나 1child는 case1,2처럼 지운다.
9. 반대쪽도 같은 과정
10. Case3는 두가지의 경우의 값이 나올수 있다.



# 7. Tree traversing

## Tree traversing

### ◆ Tree

◆ Complicated than a list

◆ Multiple ways to show the entire dataset

◆ If it were a list

◆ Just show the values from the beginning to the end

◆ Since this is a BST

◆ You have to choose what to show at a time

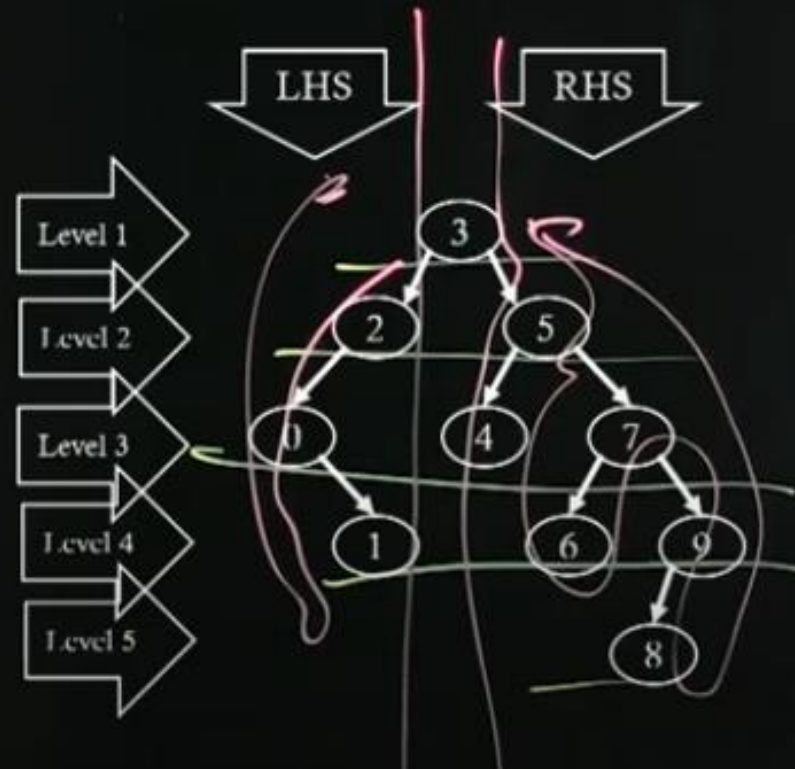
◆ The value in LHS

◆ The value in RHS

◆ The value that you have

◆ Hence there are multiple traversing approaches

Inserting 3, 2, 0, 5, 7, 4, 6, 1, 9, 8

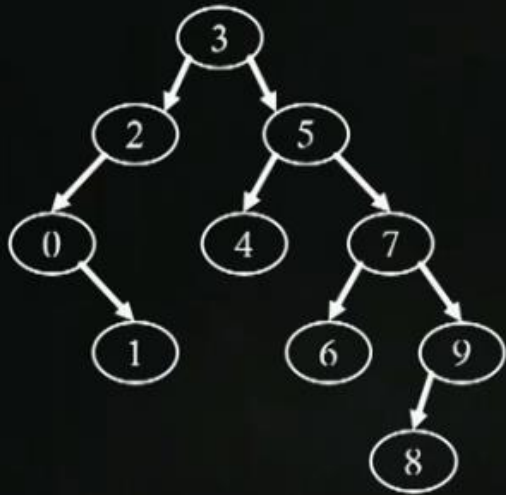


1. 링크드 리스트의 경우 출력을 순서대로 프리트함
2. 트리의 경우 출력을하는데에 있어 문제를 어떻게 풀지 정의를 해줘야 한다.
3. 트리는 출력하는데 링크드리스트보다 복잡하다.

# 7. Tree traversing

## Depth first traverse

- ◆ Pre-order traverse
  - ◆ Order: Current, LHS, RHS in Recursion
- ◆ In-order traverse
  - ◆ Order: LHS, Current, RHS in Recursion
- ◆ Post-order traverse
  - ◆ Order: LHS, RHS, Current in Recursion



```
def traverseInOrder(self, node = ''):
    if node == '':
        node = self.root
    ret = []
    if node.getLHS() != '':
        ret = ret + self.traverseInOrder(node.getLHS())
    ret.append( node.getValue() )
    if node.getRHS() != '':
        ret = ret + self.traverseInOrder(node.getRHS())
    return ret

def traversePreOrder(self, node = ''):
    if node == '':
        node = self.root
    ret = []
    ret.append( node.getValue() )
    if node.getLHS() != '':
        ret = ret + self.traversePreOrder(node.getLHS())
    if node.getRHS() != '':
        ret = ret + self.traversePreOrder(node.getRHS())
    return ret

def traversePostOrder(self, node = ''):
    if node == '':
        node = self.root
    ret = []
    if node.getLHS() != '':
        ret = ret + self.traversePostOrder(node.getLHS())
    if node.getRHS() != '':
        ret = ret + self.traversePostOrder(node.getRHS())
    ret.append( node.getValue() )
    return ret
```

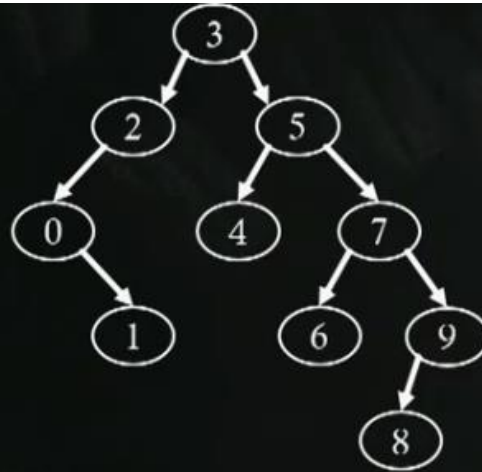
1. LHS를 먼저 출력하고 RHS를 그 다음으로 출력
2. Parents의 value의 출력순서에 따라 Pre-order, In-order, Post-order traverse가 있다.
3. Pre-order : Current, LHS, RHS(3,2,0,1,5,4,7,6,9,8)
4. In-order : LHS, Current, RHS(0,1,2,3,4,5,6,7,8,9)
5. Post-order : LHS, RHS, Current(1,0,2,4,6,8,9,7,5,3)
6. Append의 순서가 달라짐

# 7. Tree traversing

## Breadth first traverse

### Queue-based level-order traverse

- 3, 2, 5, 0, 4, 7, 1, 6, 9, 8
- Enqueue the root
  - While until queue is empty
    - Current = Dequeue one element
    - Print current
    - If Current's LHS exist
      - Enqueue current.LHS
    - If Current's RHS exist
      - Enqueue current.RHS



Current	Queue
	3
3	2, 5
2	5, 0
5	0, 4, 7
0	4, 7, 1
4	7, 1
7	1, 6, 9
1	6, 9
6	9
9	8
8	

```
def traverseLevelOrder(self):
    ret = []
    Q = Queue()
    Q.enqueue(self.root)
    while Q.isEmpty() == False:
        node = Q.dequeue()
        if node == '':
            continue
        ret.append(node.getValue())
        if node.getLHS() != '':
            Q.enqueue(node.getLHS())
        if node.getRHS() != '':
            Q.enqueue(node.getRHS())
    return ret
```

1. Root를 enqueue 먼저 한다.
2. Level-order traverse를 하기 위해 구현
3. Queue를 하나 만들고
4. Root를 enqueue 함
5. Q가 empty 될때까지 while문 반복

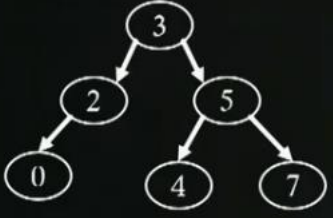
# 7. Tree traversing

## Performance of binary search tree

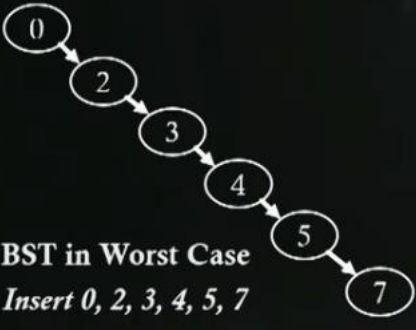
Coming from divide and conquer

	Linked List	BST in Average	BST in Worst Case
Search	$O(n)$	$O(\log n)$	$O(n)$
Insert after search	$O(1)$	$O(1)$	$O(1)$
Delete after search	$O(1)$	$O(1)$	$O(1)$
Traverse	$O(n)$	$O(n)$	$O(n)$

**BST in Average**  
Insert 3, 2, 0, 5, 4, 7



**BST in Worst Case**  
Insert 0, 2, 3, 4, 5, 7



1. Binary search tree는 depth가 높고 Height가 높으면 search하는데 매우 좋지 않다
2. Binary search tree는 height에 따라서 성능이 달라짐
3. 두번째의 경우는 Linked list와 같다.
4. O 라는 알고리즘은 성능을 재는 척도