Data Structure

실습 5

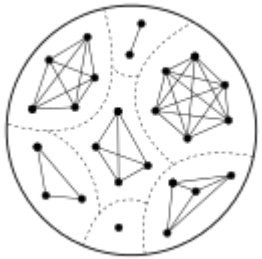
0. 이번 주 실습 내용

- (Equivalence Classes 복습)
- Binary Tree
  - Binary Tree 개념 & 구조
- Traversal of Binary Tree
- Binary Search Tree 실습

1. Equivalence classes

- **정의: 연관이 있는 원소들을 한 집합으로 묶은 것 (엄밀한 정의는 이론수업 참조)**

- 그래프에서 연결된 노드들을 군집화 할 때 사용



- 구현 형태: Linked List로 구현
  - 본 실습에서는 이론수업PPT와 같이 Linked List의 배열을 사용

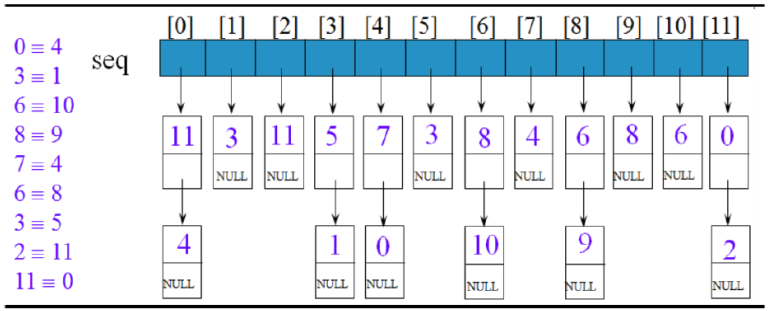
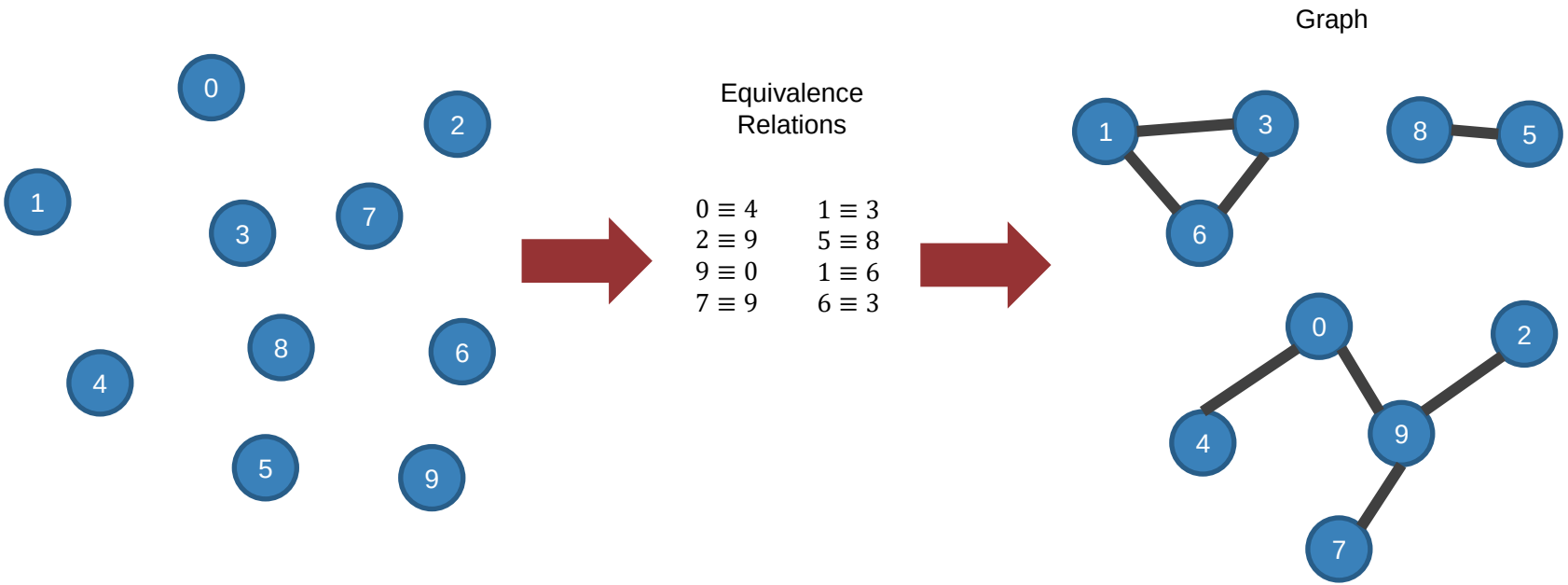


Figure 4.16 : Lists after pairs have been input

1. Equivalence Classes

- 동치관계는 그래프 상의 연결여부로 표현할 수 있다.



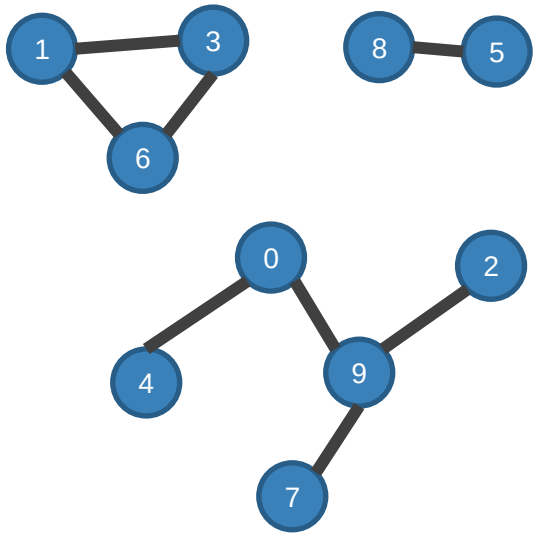
1. Equivalence Classes

- 동치관계는 그래프 상의 연결여부로 표현할 수 있다.

Equivalence Relations

$0 \equiv 4$ $1 \equiv 3$
$2 \equiv 9$ $5 \equiv 8$
$9 \equiv 0$ $1 \equiv 6$
$7 \equiv 9$ $6 \equiv 3$

Graph



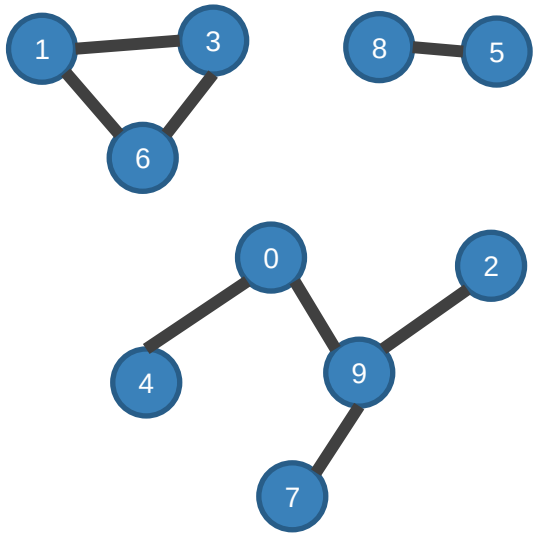
1. Equivalence Classes

- 동치류의 원소들은 동치류에 속하는 노드와 연결된 모든 노드를 방문하여 알 수 있다.

Equivalence Relations

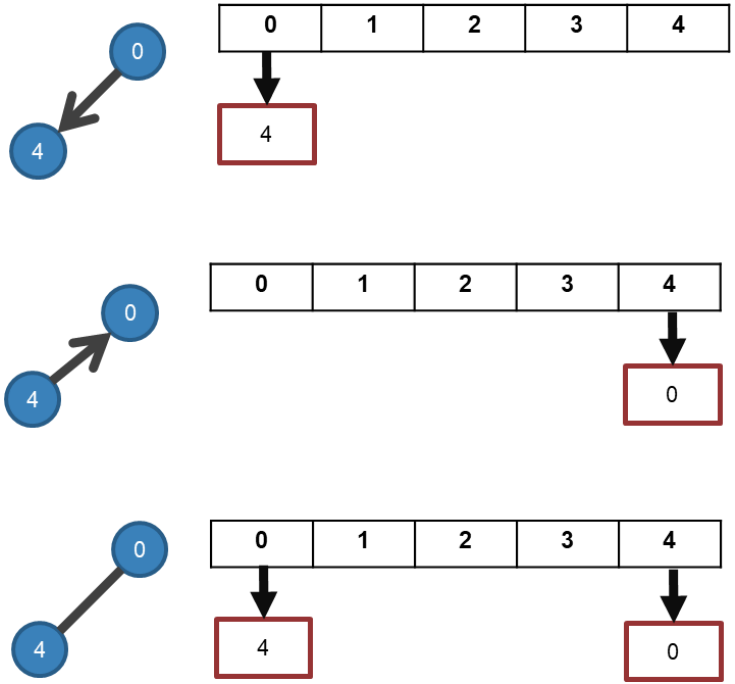
$0 \equiv 4$ $1 \equiv 3$
$2 \equiv 9$ $5 \equiv 8$
$9 \equiv 0$ $1 \equiv 6$
$7 \equiv 9$ $6 \equiv 3$

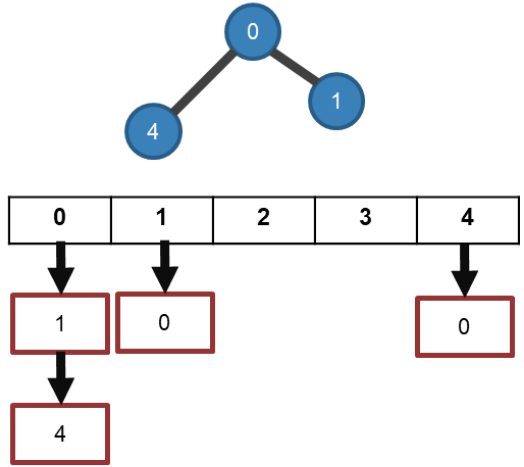
Graph



1. Equivalence Classes

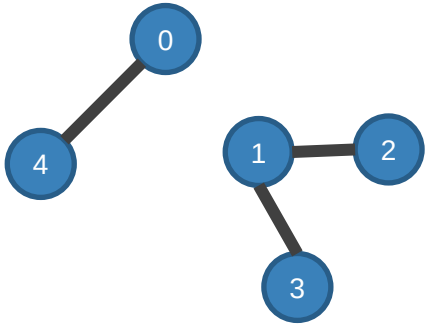
- 그래프는 여러 개의 리스트로 표현될 수 있다.

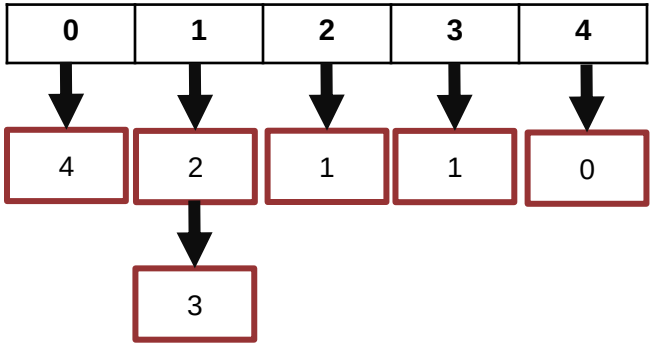




1. Equivalence Classes

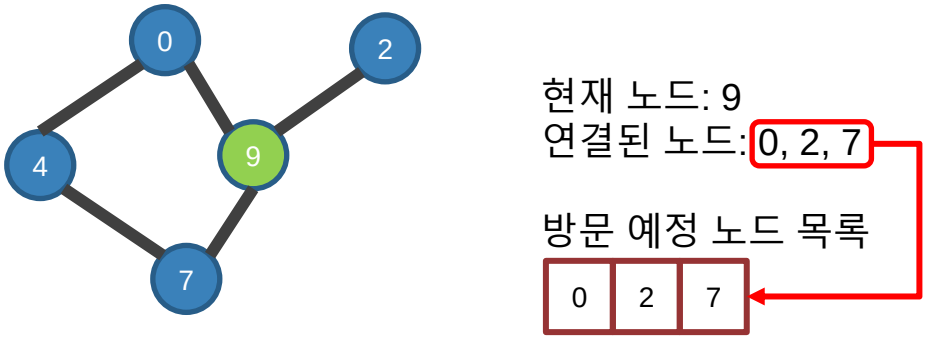
- 그래프는 여러 개의 리스트로 표현될 수 있다.





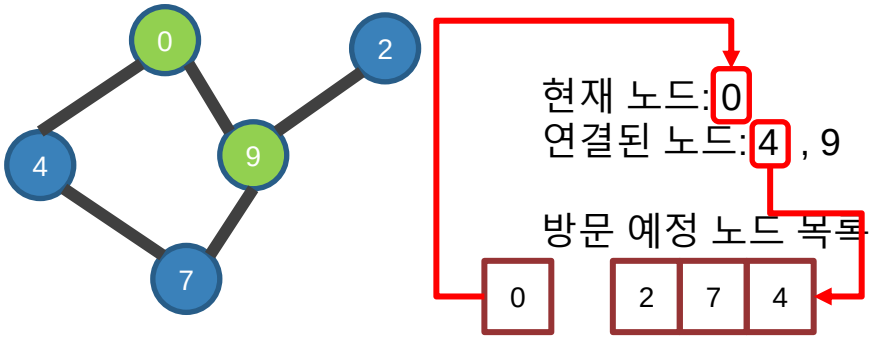
1. Equivalence Classes

- 그래프에서 연결된 노드들을 한번씩 방문하면 동치류의 모든 원소를 알 수 있다.
  - 하나의 노드에 방문하면 연결된 노드를 방문할 노드에 기록해 놓은 뒤 나중에 방문함.
  - 이미 방문예정목록에 있는 노드/방문했던 노드면 목록에 추가하지 않음.



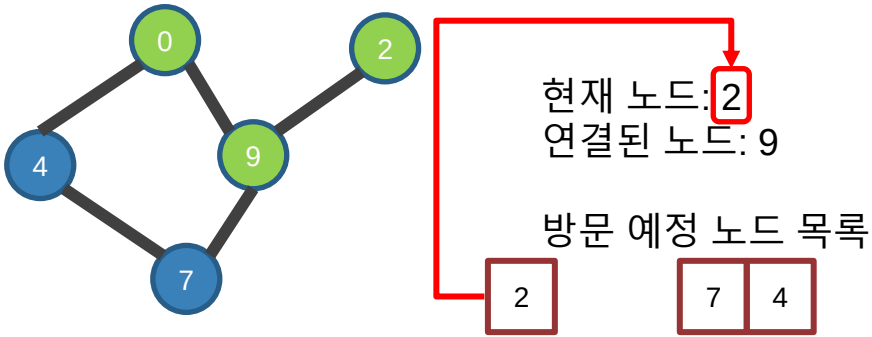
1. Equivalence Classes

- 그래프에서 연결된 노드들을 한번씩 방문하면 동치류의 모든 원소를 알 수 있다.
  - 하나의 노드에 방문하면 연결된 노드를 방문할 노드에 기록해 놓은 뒤 나중에 방문함.
  - 이미 방문예정목록에 있는 노드/방문했던 노드면 목록에 추가하지 않음.



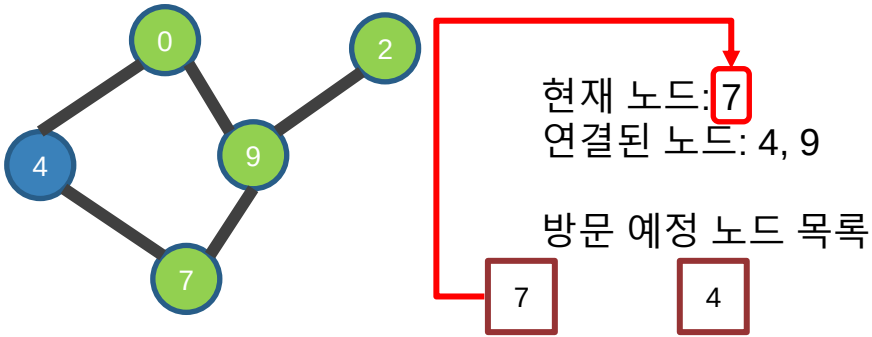
1. Equivalence Classes

- 그래프에서 연결된 노드들을 한번씩 방문하면 동치류의 모든 원소를 알 수 있다.
  - 하나의 노드에 방문하면 연결된 노드를 방문할 노드에 기록해 놓은 뒤 나중에 방문함.
  - 이미 방문예정목록에 있는 노드/방문했던 노드면 목록에 추가하지 않음.



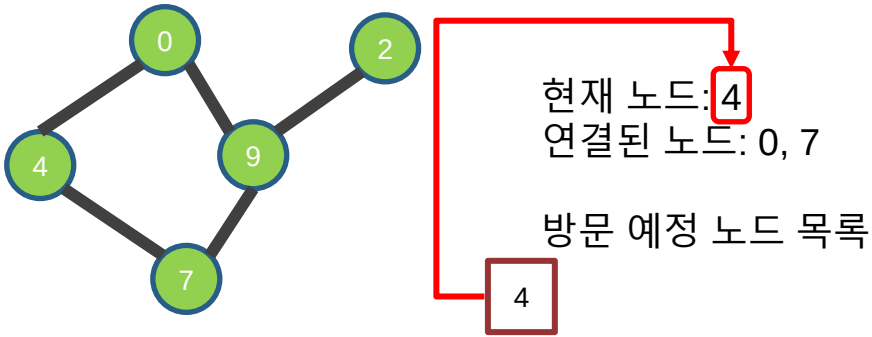
1. Equivalence Classes

- 그래프에서 연결된 노드들을 한번씩 방문하면 동치류의 모든 원소를 알 수 있다.
  - 하나의 노드에 방문하면 연결된 노드를 방문할 노드에 기록해 놓은 뒤 나중에 방문함.
  - 이미 방문예정목록에 있는 노드/방문했던 노드면 목록에 추가하지 않음.



1. Equivalence Classes

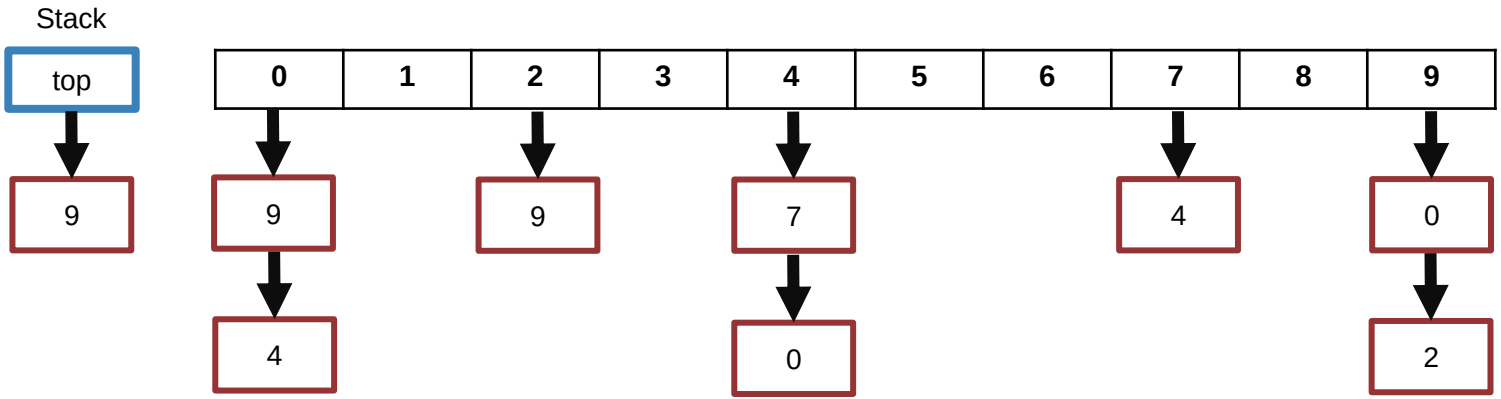
- 그래프에서 연결된 노드들을 한번씩 방문하면 동치류의 모든 원소를 알 수 있다.
  - 하나의 노드에 방문하면 연결된 노드를 방문할 노드에 기록해 놓은 뒤 나중에 방문함.
  - 이미 방문예정목록에 있는 노드/방문했던 노드면 목록에 추가하지 않음.



Enumerating equivalence class including 9.

Equivalence Relations

$0 \equiv 4$
$7 \equiv 4$
$2 \equiv 9$
$9 \equiv 0$



방문할 노드(9)를 스택에 push

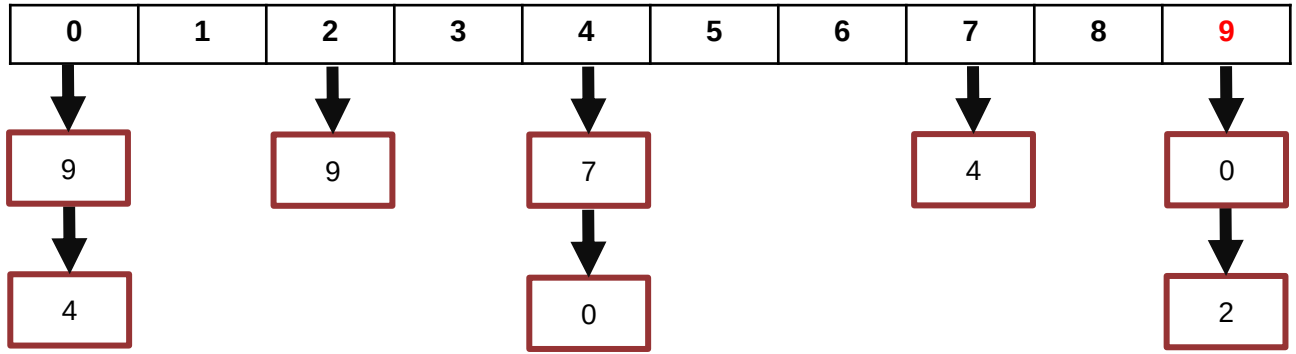
Enumerating equivalence class including 9.

Equivalence Relations

$0 \equiv 4$
$7 \equiv 4$
$2 \equiv 9$
$9 \equiv 0$

Stack

top



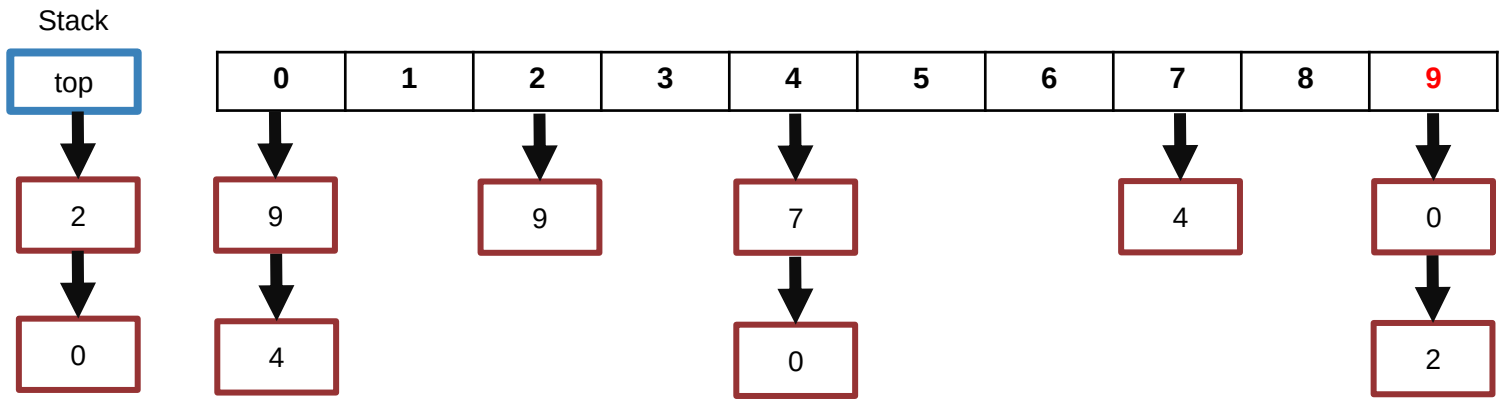
9

스택에서 노드를 pop하여 노드(9) 방문

Enumerating equivalence class including 9.

Equivalence Relations

$0 \equiv 4$
$7 \equiv 4$
$2 \equiv 9$
$9 \equiv 0$

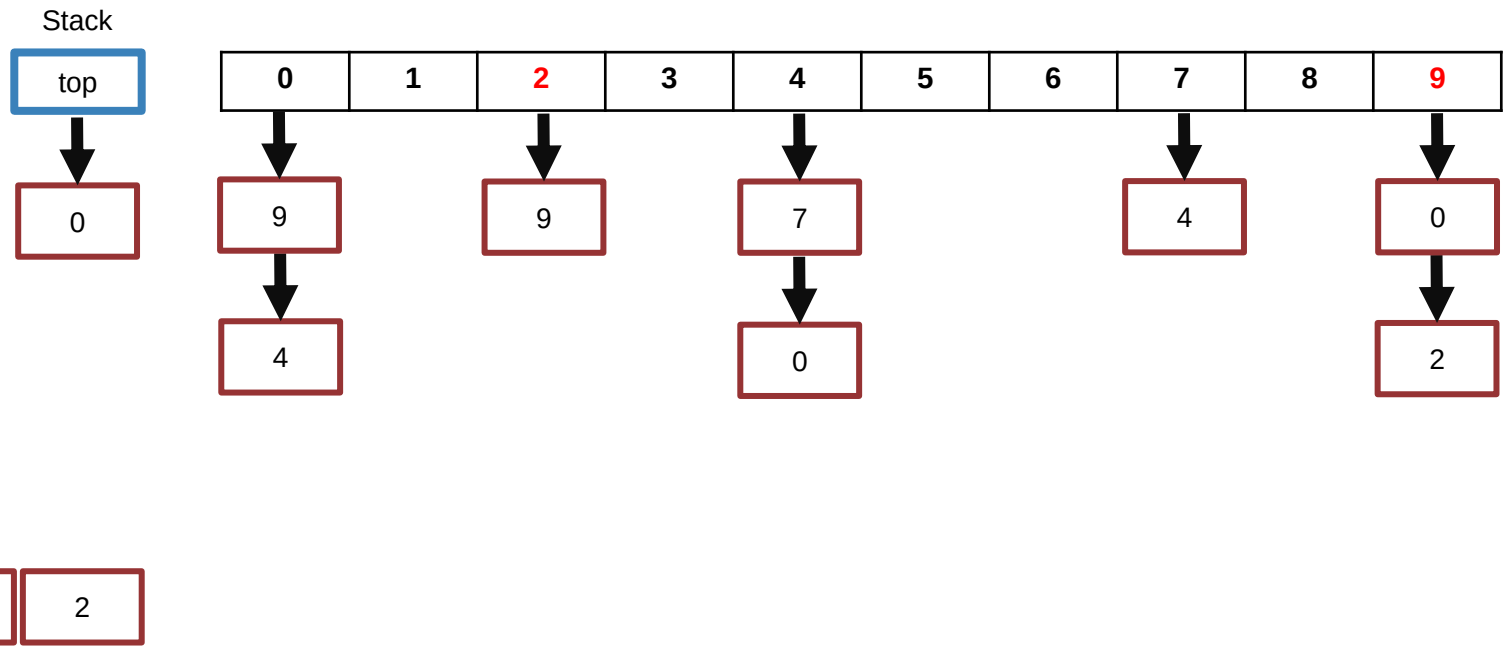


방문한 노드(9)에 연결된 노드(2, 0)를 스택에 push

Enumerating equivalence class including 9.

Equivalence Relations

$0 \equiv 4$
$7 \equiv 4$
$2 \equiv 9$
$9 \equiv 0$



스택에서 노드를 pop하여 노드(2) 방문 및 연결된 노드를 스택에 push

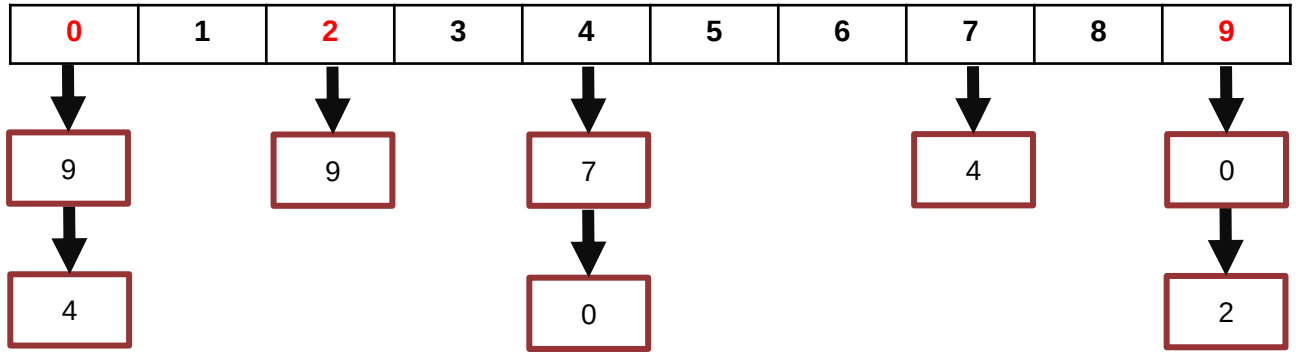
Enumerating equivalence class including 9.

Equivalence Relations

$0 \equiv 4$
$7 \equiv 4$
$2 \equiv 9$
$9 \equiv 0$

Stack

top

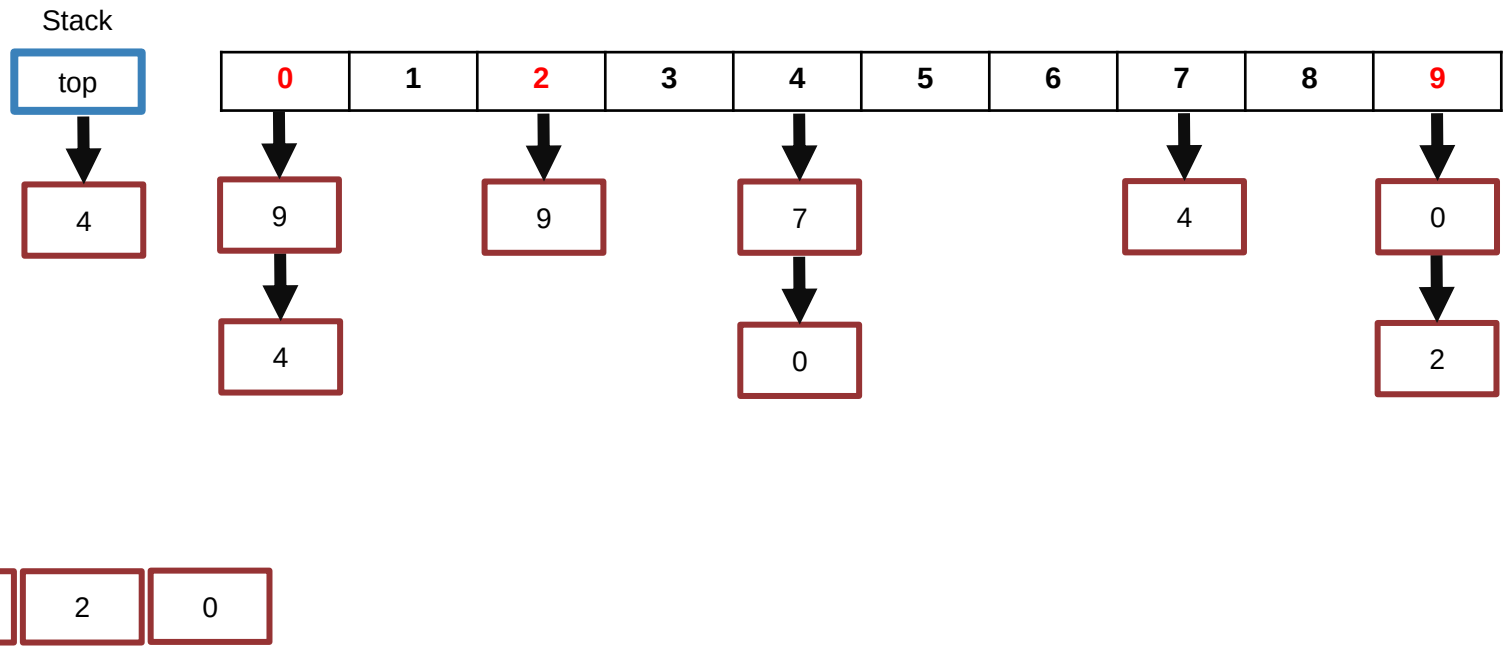


스택에서 노드를 pop하여 노드(0) 방문

Enumerating equivalence class including 9.

Equivalence Relations

$0 \equiv 4$
$7 \equiv 4$
$2 \equiv 9$
$9 \equiv 0$



방문한 노드(0)에 연결된 노드(4)를 스택에 push

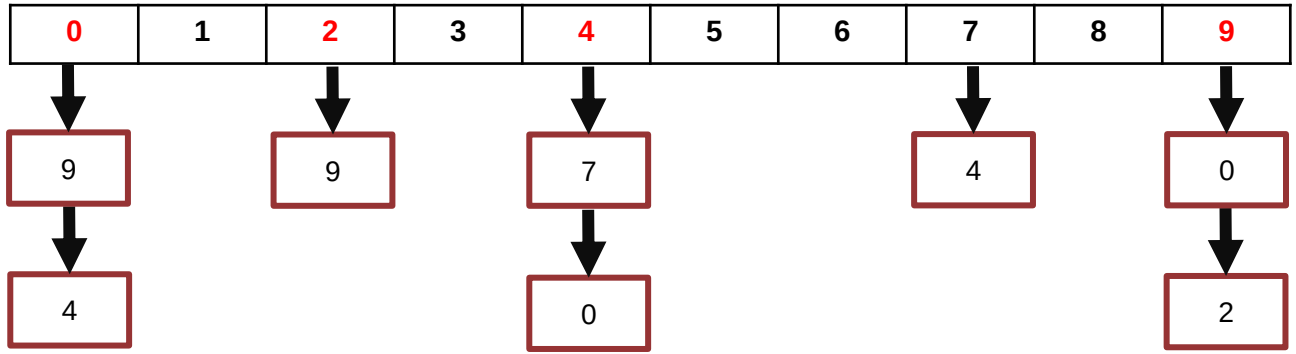
Enumerating equivalence class including 9.

Equivalence Relations

$0 \equiv 4$
$7 \equiv 4$
$2 \equiv 9$
$9 \equiv 0$

Stack

top

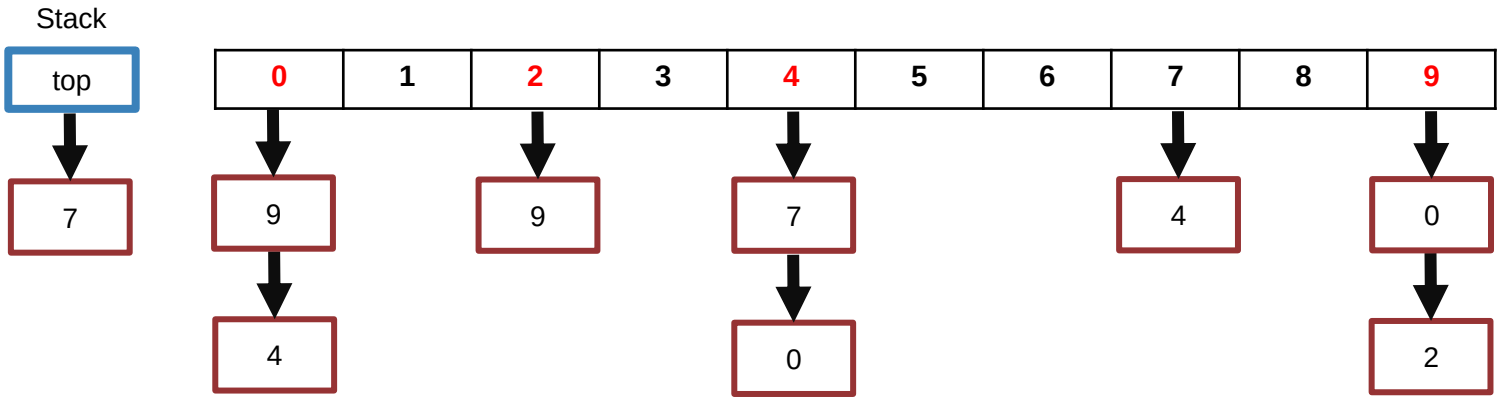


스택에서 노드를 pop하여 노드(4) 방문

Enumerating equivalence class including 9.

Equivalence Relations

$0 \equiv 4$
$7 \equiv 4$
$2 \equiv 9$
$9 \equiv 0$



방문한 노드(4)에 연결된 노드(7)를 스택에 push

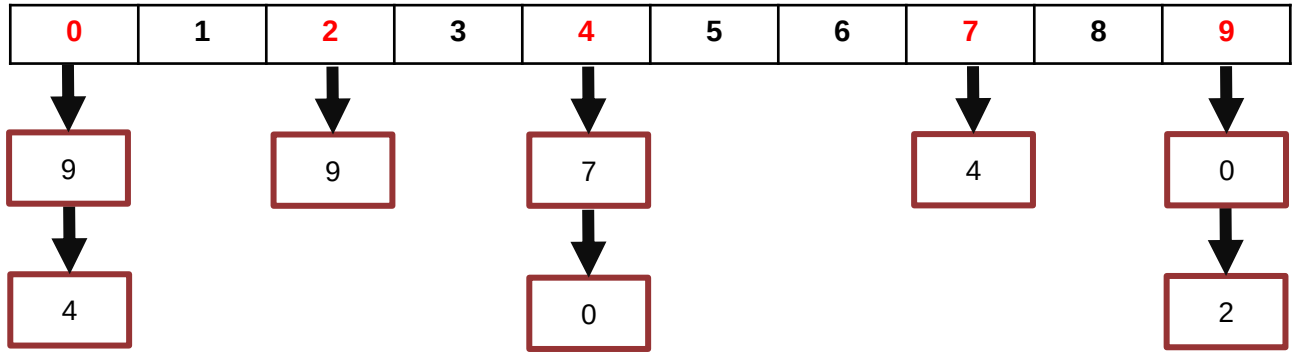
Enumerating equivalence class including 9.

Equivalence Relations

$0 \equiv 4$
$7 \equiv 4$
$2 \equiv 9$
$9 \equiv 0$

Stack

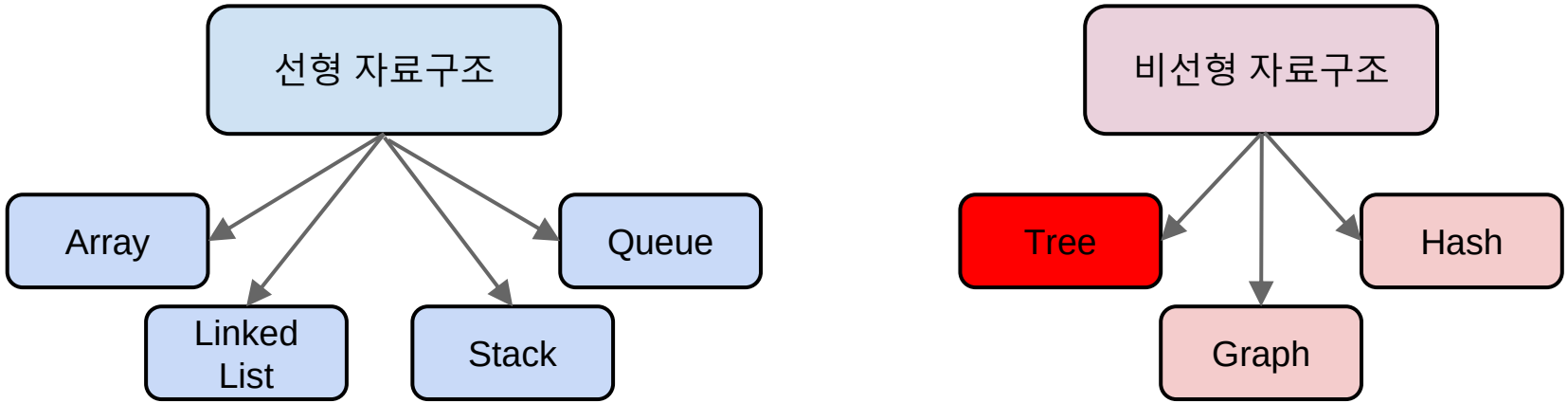
top



스택에서 노드를 pop하여 노드(7) 방문

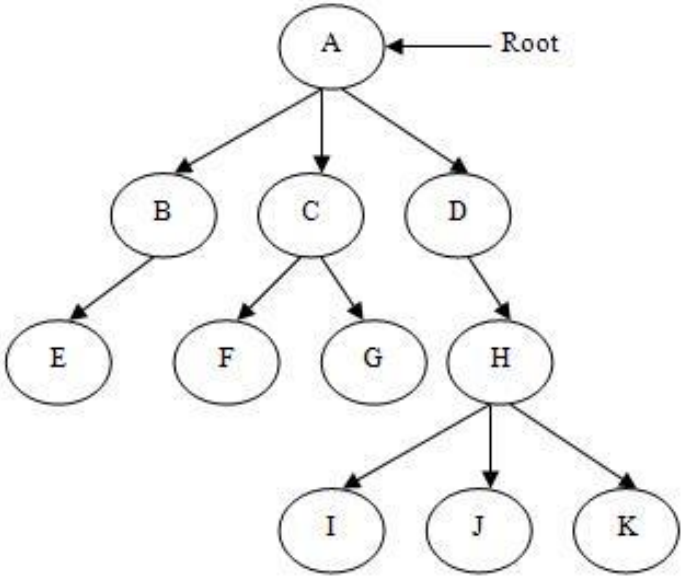
2. Binary Tree

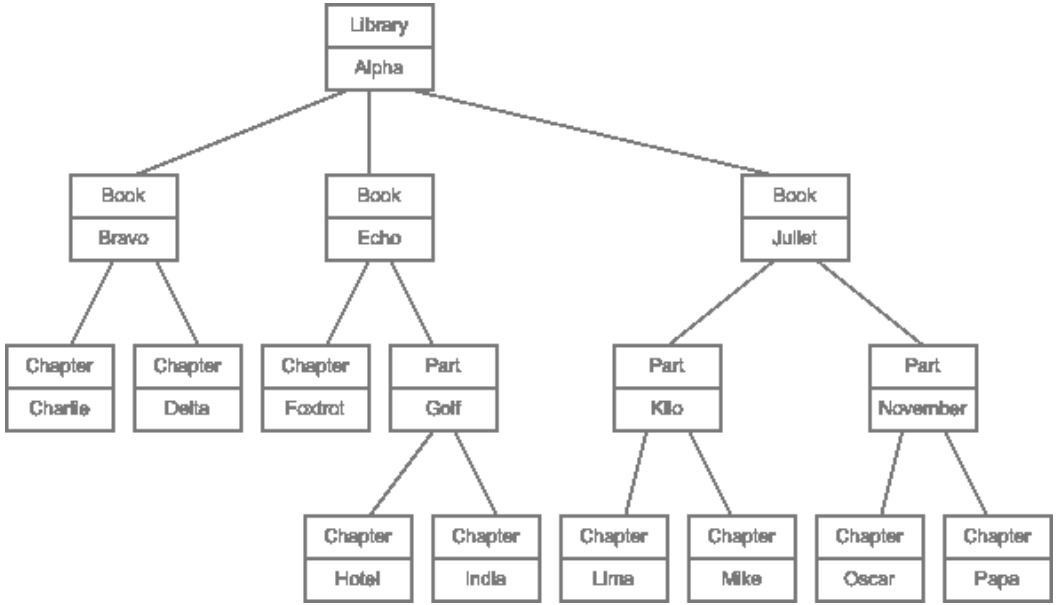
- **Data Structure**



2. Binary Tree

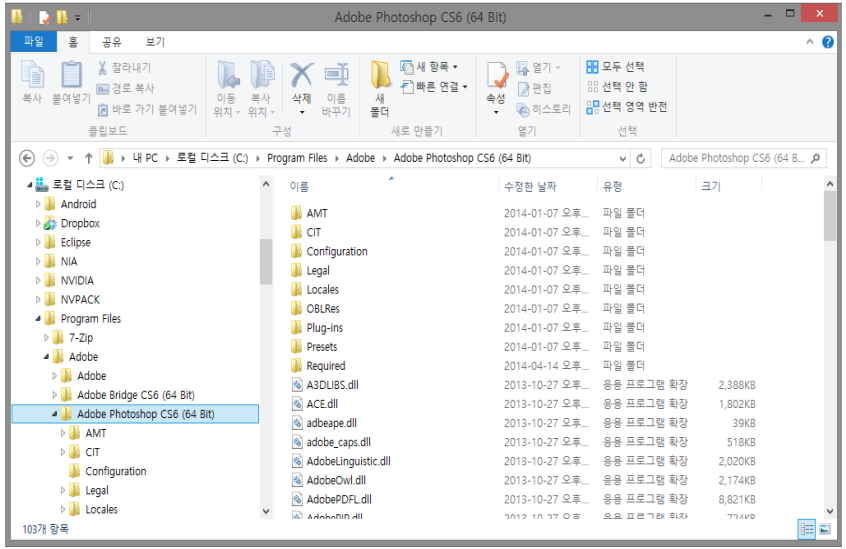
- **What is Tree?**



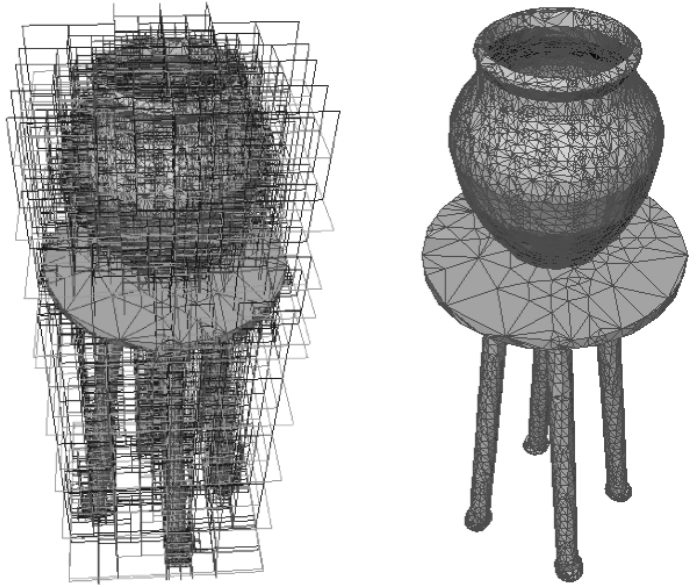


2. Binary Tree

- 언제 쓸까?



File System



Graphics

2. Binary Tree

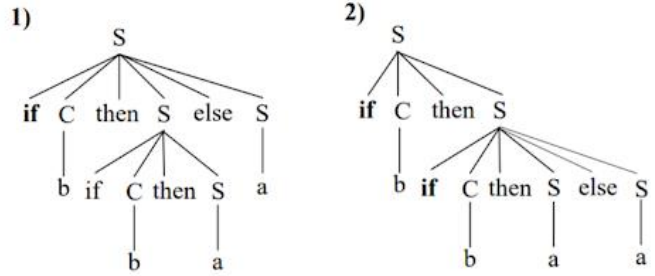
- 언제 쓸까?

ex) problem:

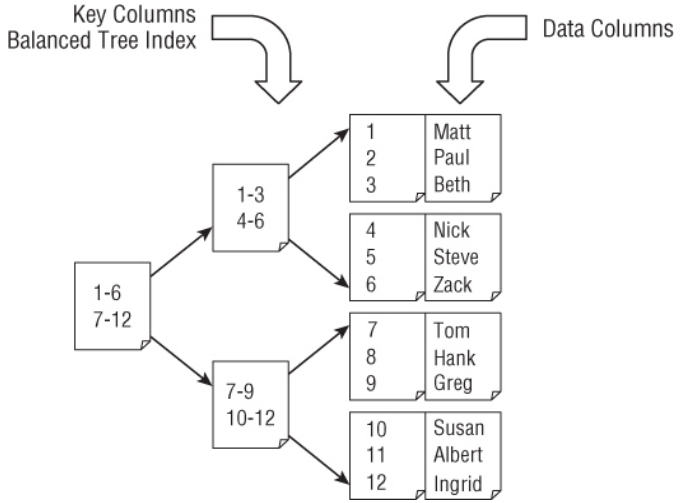
G: S → if C then S else S | if C then S | a

C → b

ω: if b then if b then a else a



Compiler

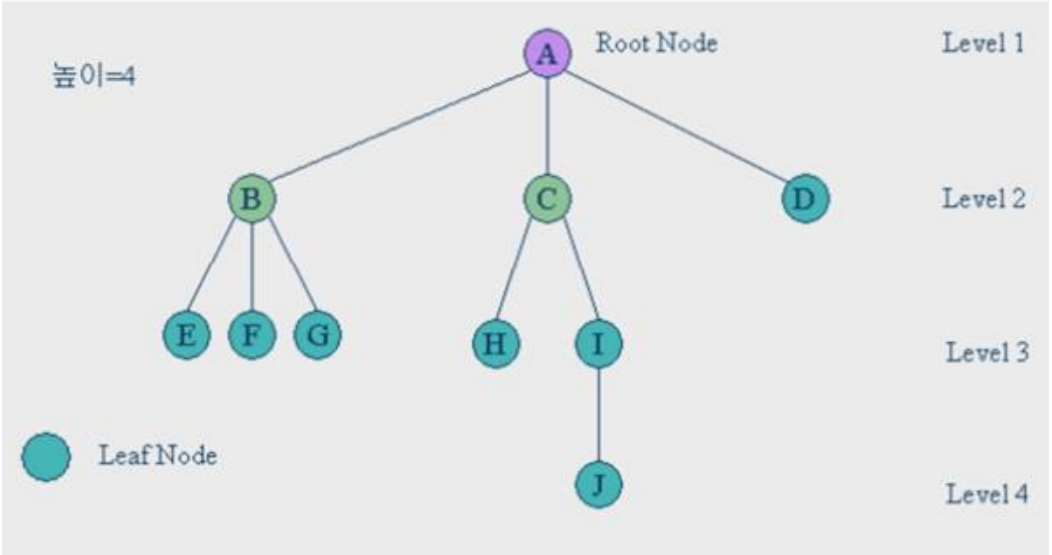


Database (Index)

2. Binary Tree

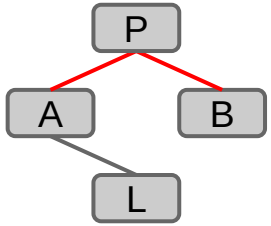
- **Tree (정의): 자료와 그 다음 자료의 위치 정보가 저장된 비선형의 자료구조**

- 구성
  - **Node (Vertex)**
    - 자료를 저장하는 공간
    - Root Node: 가장 위에 있는 Node
    - Leaf Node: 가장 아래 있는 Node
  - **Link (Edge)**
    - 다음 Node를 가리키는 링크(pointer)

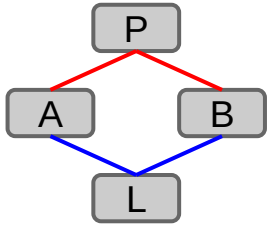


2. Binary Tree

- **Node 특징 (Tree)**
  - Root Node (P)
    - 가장 위에 있는 노드 (Head Node 같은 느낌)
  - Leaf Node (L)
    - 가장 아래에 있는 노드
  - Sibling Node (A,B)
    - 부모가 같은 노드



트리(Tree)



그래프(Graph)
나중에 배움

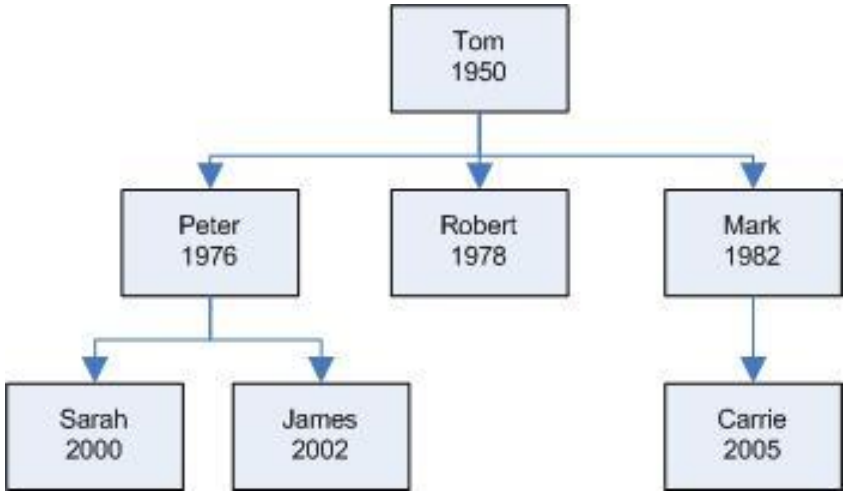
  - Node
    - 하나의 부모(parent) Node가 있어야 함 (예외 – Root Node)
    - 여러 개의 자식(child) Node를 가질 수 있음 (Link 개수에 따라 정해짐)
    - 한 Node에서 다른 Node로 가기 위한 경로가 유일해야 함 (Tree의 조건)

2. Binary Tree

- **General Tree**
  - 자식 Node의 개수가 여러 개
  - 규칙성이 존재하지 않음

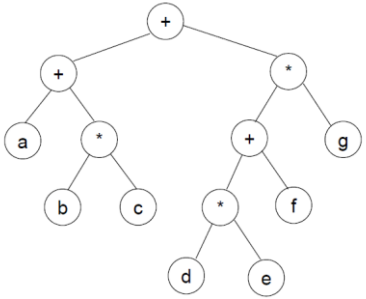
➔ **규칙성을 갖는 특정한 Tree 구조를 정의하자**

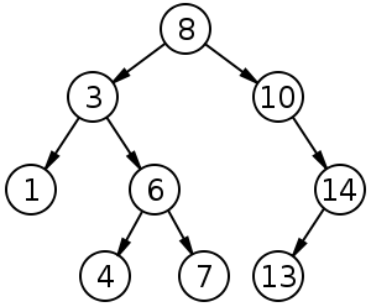
➔ **Binary Tree!**

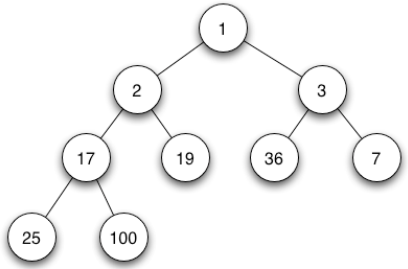


2. Binary Tree

- **Binary Tree(이진 트리)**
  - Node에 규칙을 추가
    - 한 Node는 2개 이하의 자식을 가지고 있음 (Left Child, Right Child)
    - 최대 2개의 자식 Node를 가지므로 아래로 내려갈 때 2가지 경로만 존재함
  - 쓰임이 정말 많은 자료구조
    - Parse Tree: 수식 계산
    - Heap : 여러 개의 값 중 가장 크거나 작은 값을 빠르게 찾기 위한 이진 트리. (정렬)
    - Binary Search Tree: 검색

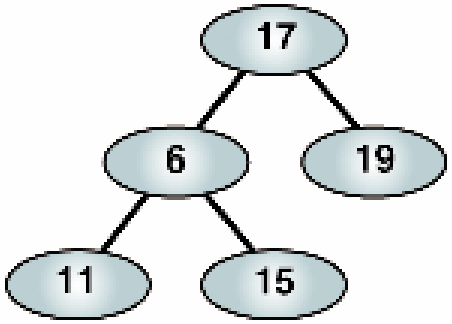




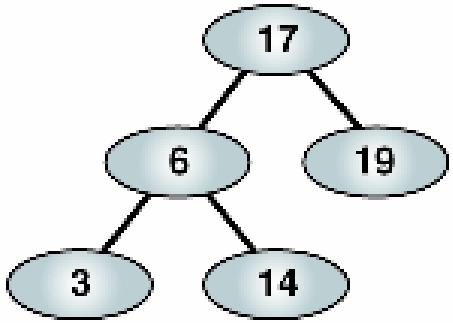


2. Binary Tree

- **Binary Search Tree(이진 탐색 트리)**
  - Binary Tree의 일종
  - Node의 왼쪽 자식 Node에는 자신보다 작은 값들만 존재
  - Node의 오른쪽 자식 Node에는 자신보다 큰 값들만 존재



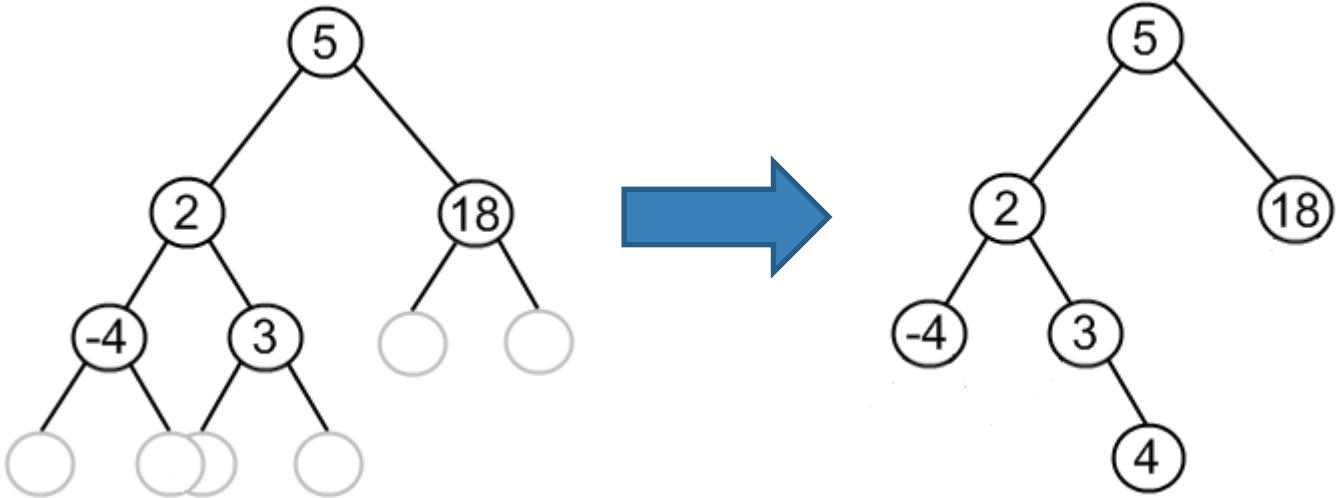
Binary tree



Binary search tree

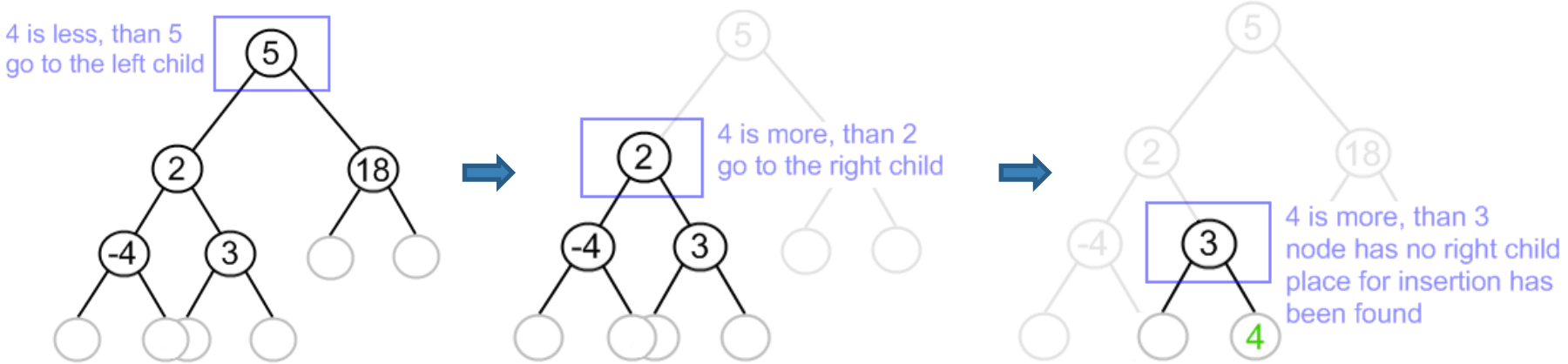
2. Binary Tree

- **Binary Search Tree(이진 탐색 트리)**
  - Add Node (add 4)

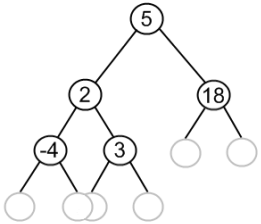


2. Binary Tree

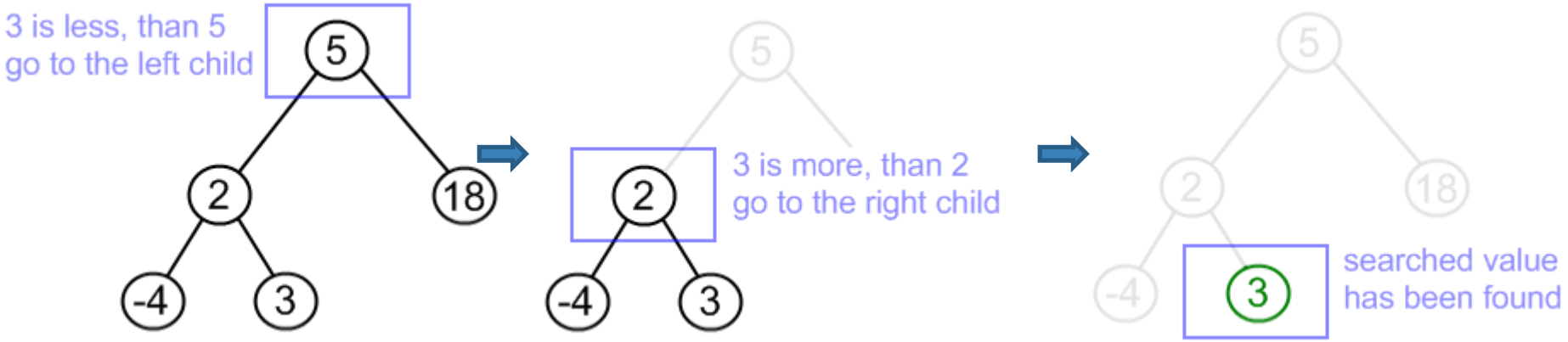
- **Binary Search Tree(이진 탐색 트리)**
  - Add Node (add 4)



2. Binary Tree

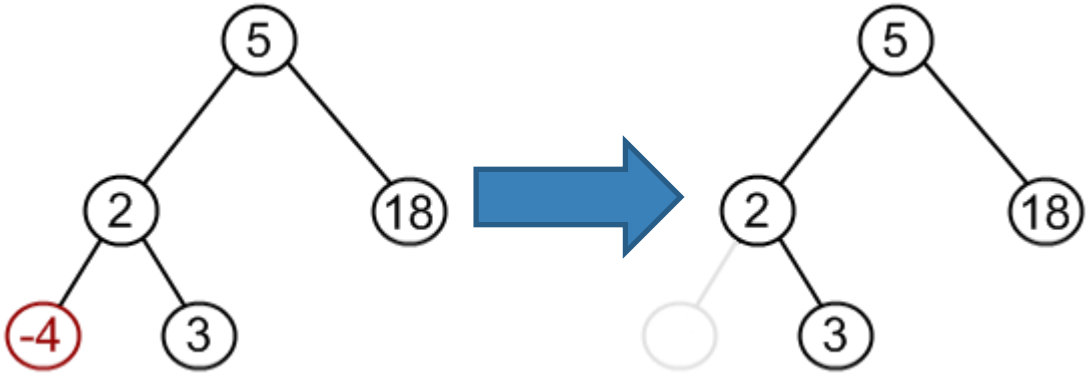


- **Binary Search Tree(이진 탐색 트리)**
  - Search Node (search 3)



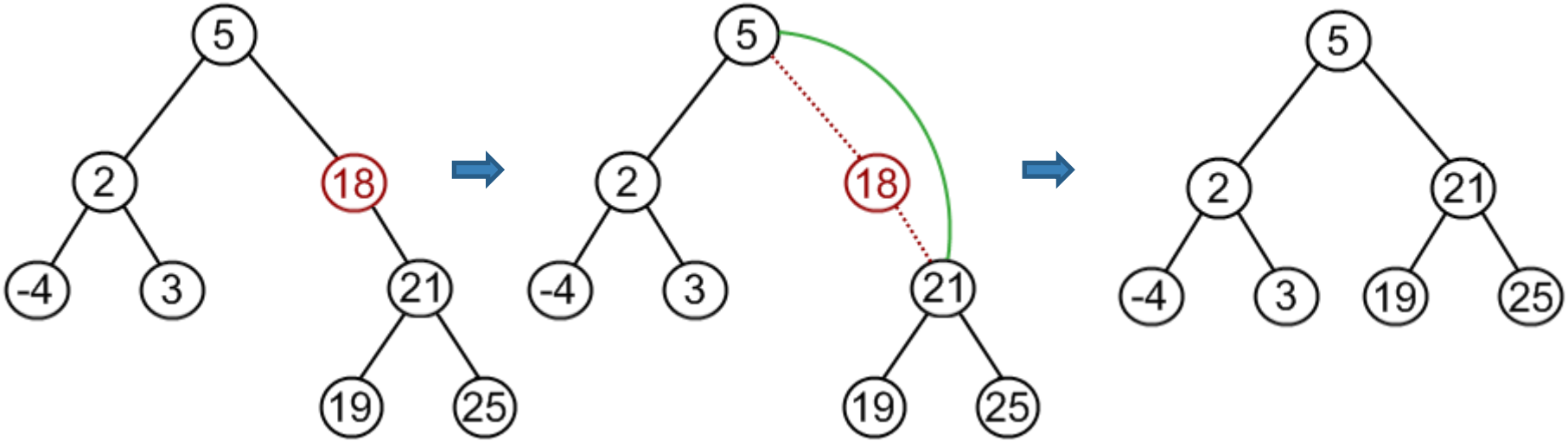
2. Binary Tree

- **Binary Search Tree(이진 탐색 트리)**
  - Remove Node (remove -4) – **Case1.** child Node가 하나도 없는 경우



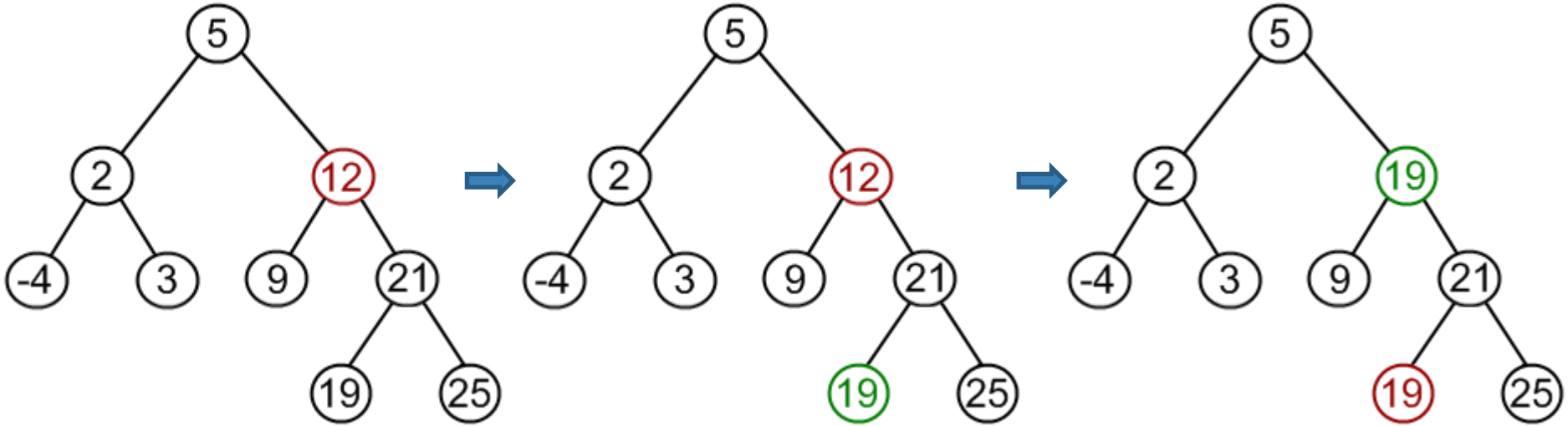
2. Binary Tree

- **Binary Search Tree(이진 탐색 트리)**
  - Remove Node (remove 18) – **Case2.** child Node가 하나 존재하는 경우



2. Binary Tree

- **Binary Search Tree(이진 탐색 트리)**
  - Remove Node (remove 12) – **Case3.** child Node가 둘 다 있는 경우



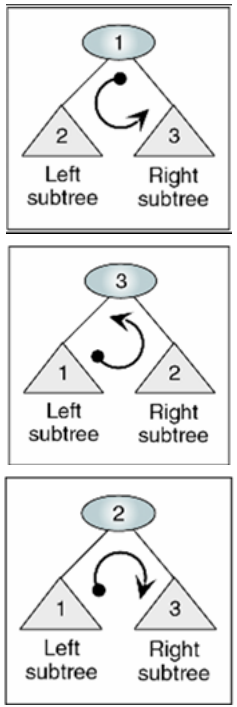
3. Traversal of Binary Tree

- **How to show all data in binary tree?**
  - Tree는 비선형 자료구조
  - 단순 선형 자료구조들과 달리 각 Node 들을 방문하기 위한 규칙이 필요

1. Pre-order : 자기를 먼저. 그 다음 왼쪽. 마지막에 오른쪽 탐색.

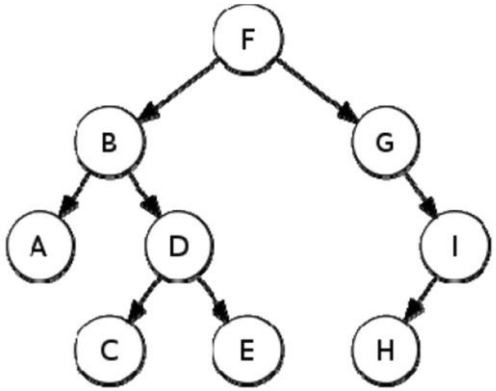
2. In-order : 왼쪽 먼저 탐색. 그 다음 자기 자신. 마지막에 오른쪽 탐색.

3. Post-order : 왼쪽 먼저 탐색. 그 다음 오른쪽 탐색. 마지막에 자기 자신



3. Traversal of Binary Tree

- **How to show all data in binary tree?**
  - Tree는 비선형 자료구조
  - 단순 선형 자료구조들과 달리 각 Node 들을 방문하기 위한 규칙이 필요

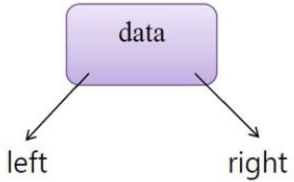


Pre-order: F B A D C E G I H
In-order: A B C D E F G H I
Post-order: A C E D B H I G F

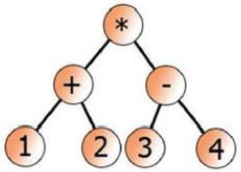
3. Binary Search Tree

- **Binary Search Tree Implementation (Linked List)**
  - Node Structure

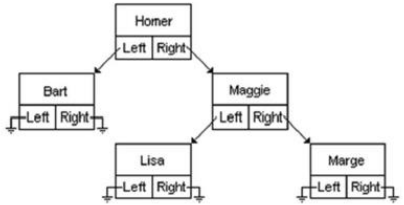
```
typedef struct TreeNode{
    int data;
    struct TreeNode* left;
    struct TreeNode* right;
}TreeNode;
```



  - Binary Tree Example

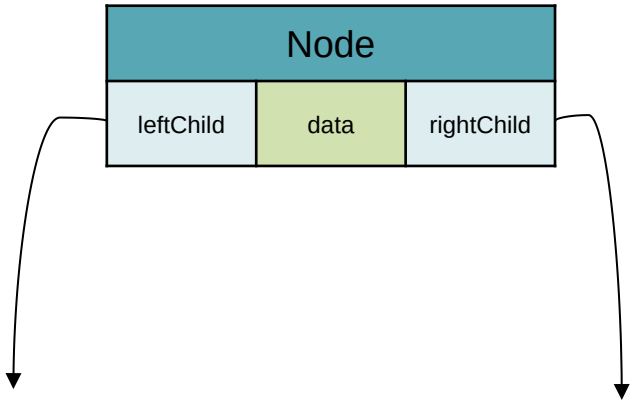


((1+2)*(3-4))



3. Binary Search Tree – 구현 실습

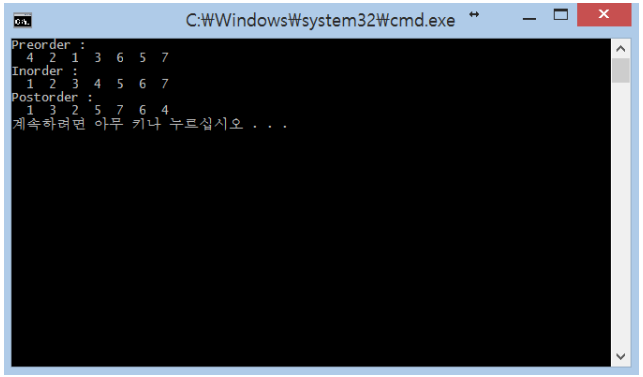
- **Tree Node Data Type**

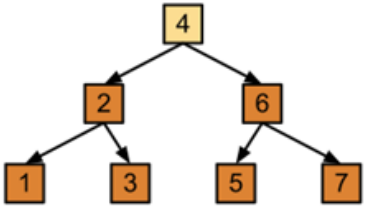



```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   typedef struct Node
5   {
6       int data;
7       struct Node* leftChild;
8       struct Node* rightChild;
9   }Node;
10
11
12  void insertTreeNode(Node** p, int value);
13  void printTreePreorder(Node* pNode);
14  void printTreeInorder(Node* pNode);
15  void printTreePostorder(Node* pNode);
16
17
18
19
```

3. Binary Search Tree – 구현 실습

- **main**






```
20  int main()
21  {
22
23      Node* pParentNode = NULL;
24
25      insertTreeNode(&pParentNode, 4);
26      insertTreeNode(&pParentNode, 2);
27      insertTreeNode(&pParentNode, 6);
28      insertTreeNode(&pParentNode, 1);
29      insertTreeNode(&pParentNode, 3);
30      insertTreeNode(&pParentNode, 5);
31      insertTreeNode(&pParentNode, 7);
32
33      printf("Preorder : ₩n");
34      printTreePreorder(pParentNode);
35      printf("₩nInorder :₩n");
36      printTreeInorder(pParentNode);
37      printf("₩nPostorder: ₩n");
38      printTreePostorder(pParentNode);
39
40      printf("₩n");
41
42
43      return 0;
44  }
45
```

3. Binary Search Tree – 구현 실습

- **Insert (재귀함수를 이용하면 간단!)**
  1. **Node가 존재하는가**
     I. 존재하지 않으면 여기에다 집어넣자
     II. 존재하면 넣을 곳을 탐색하자 (2번이나 3번)
  2. **해당 Node의 data값보다 작으면 왼쪽 자식 Node로 내려가자**
     I. 다시 1번을 수행하자
  3. **해당 Node의 data값보다 크면 오른쪽 자식 Node로 내려가자**
     I. 다시 1번을 수행하자

```
47  void insertTreeNode(Node** p, int value)
48  {
49      if ((*p) == NULL)
50      {
51          // Create TreeNode with value
52
53
54
55
56      }
57      else if ((*p)->data > value)
58      {
59          // Recursive call to leftChild
60
61      }
62      else
63      {
64          // Recursive call to rightChild
65
66      }
67  }
68
```

3. Binary Search Tree – 구현 실습

- **Insert (재귀함수를 이용하면 간단!)**
  1. **Node가 존재하는가**
     I. 존재하지 않으면 여기에다 집어넣자
     II. 존재하면 넣을 곳을 탐색하자 (2번이나 3번)
  2. **해당 Node의 data값보다 작으면 왼쪽 자식 Node로 내려가자**
     I. 다시 1번을 수행하자
  3. **해당 Node의 data값보다 크면 오른쪽 자식 Node로 내려가자**
     I. 다시 1번을 수행하자

```
47  void insertTreeNode(Node** p, int value)
48  {
49      if ((*p) == NULL)
50      {
51          // Create TreeNode with value
52          (*p) = (Node*)malloc(sizeof(Node));
53          (*p)->data = value;
54          (*p)->leftChild = NULL;
55          (*p)->rightChild = NULL;
56      }
57      else if ((*p)->data > value)
58      {
59          // Recursive call to leftChild
60          insertTreeNode(&((*p)->leftChild), value);
61      }
62      else
63      {
64          // Recursive call to rightChild
65          insertTreeNode(&((*p)->rightChild), value);
66      }
67  }
68
```

3. Binary Search Tree – 구현 실습

- **show (재귀함수를 이용하면 간단!)**
  1. **Preorder**
     I. 자기 자신 Node의 데이터를 출력
     II. 왼쪽 child Node show
     III. 오른쪽 child Node show
  2. **Inorder**
     I. 왼쪽 child Node show
     II. 자기 자신 Node의 데이터를 출력
     III. 오른쪽 child Node show
  3. **Postorder**
     I. 왼쪽 child Node show
     II. 오른쪽 child Node show
     III. 자기 자신 Node의 데이터를 출력

```
69
70  void printTreePreorder(Node* pNode)
71  {
72      if (pNode == NULL)
73          return;
74
75
76
77
78  }
79
80  void printTreeInorder(Node* pNode)
81  {
82      if (pNode == NULL)
83          return;
84
85
86
87
88  }
89
90  void printTreePostorder(Node* pNode)
91  {
92      if (pNode == NULL)
93          return;
94
95
96
97
98  }
99
```

3. Binary Search Tree – 구현 실습

- **show (재귀함수를 이용하면 간단!)**
  1. **Preorder**
     I. 자기 자신 Node의 데이터를 출력
     II. 왼쪽 child Node show
     III. 오른쪽 child Node show
  2. **Inorder**
     I. 왼쪽 child Node show
     II. 자기 자신 Node의 데이터를 출력
     III. 오른쪽 child Node show
  3. **Postorder**
     I. 왼쪽 child Node show
     II. 오른쪽 child Node show
     III. 자기 자신 Node의 데이터를 출력

```
69
70  void printTreePreorder(Node* pNode)
71  {
72      if (pNode == NULL)
73          return;
74
75      printf("%3d", pNode->data);
76      printTreePreorder(pNode->leftChild);
77      printTreePreorder(pNode->rightChild);
78  }
79
80  void printTreeInorder(Node* pNode)
81  {
82      if (pNode == NULL)
83          return;
84
85      printTreeInorder(pNode->leftChild);
86      printf("%3d", pNode->data);
87      printTreeInorder(pNode->rightChild);
88  }
89
90  void printTreePostorder(Node* pNode)
91  {
92      if (pNode == NULL)
93          return;
94
95      printTreePostorder(pNode->leftChild);
96      printTreePostorder(pNode->rightChild);
97      printf("%3d", pNode->data);
98  }
99
```