Data Structure

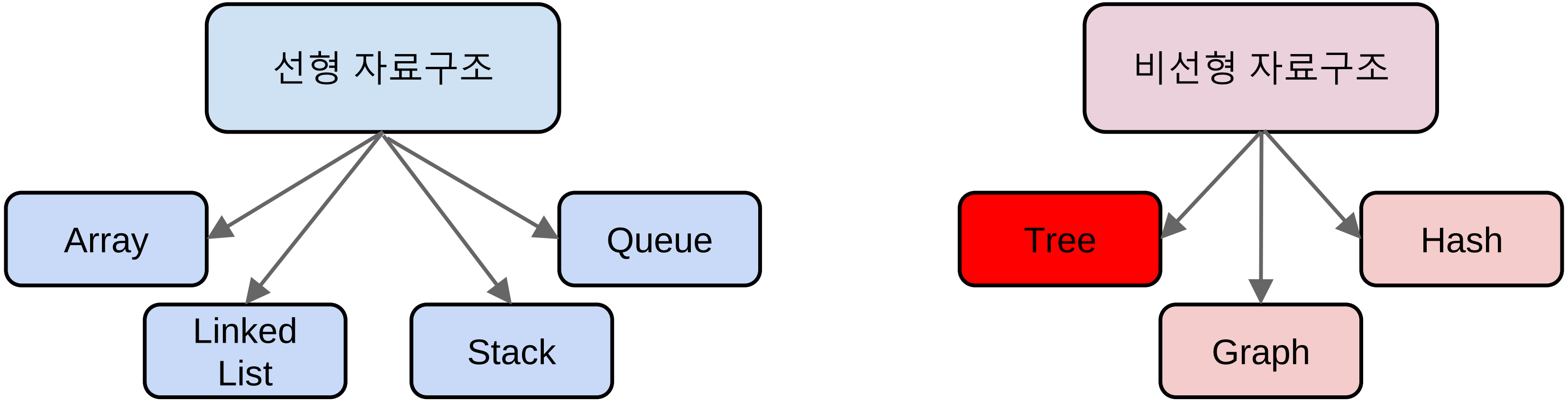
실습 6

0. 이번 주 실습 내용

- Binary Search Tree 복습
  - Traversal of BST
  - Operation of BST
- Binary Search Tree 실습

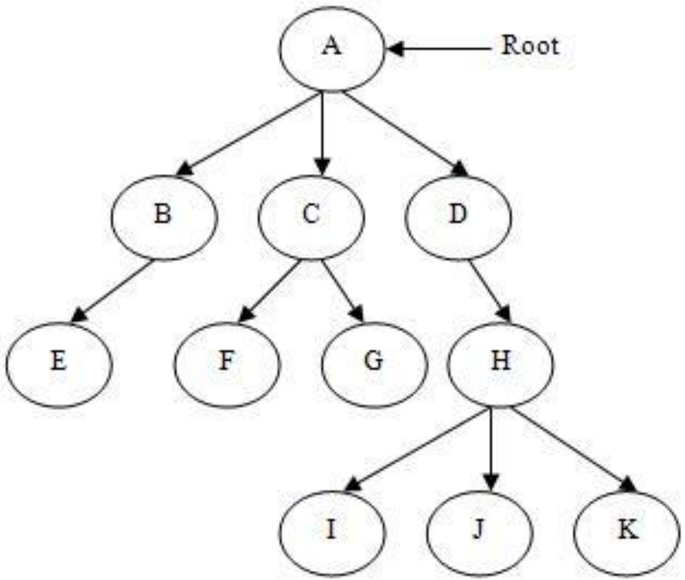
1. Binary Tree

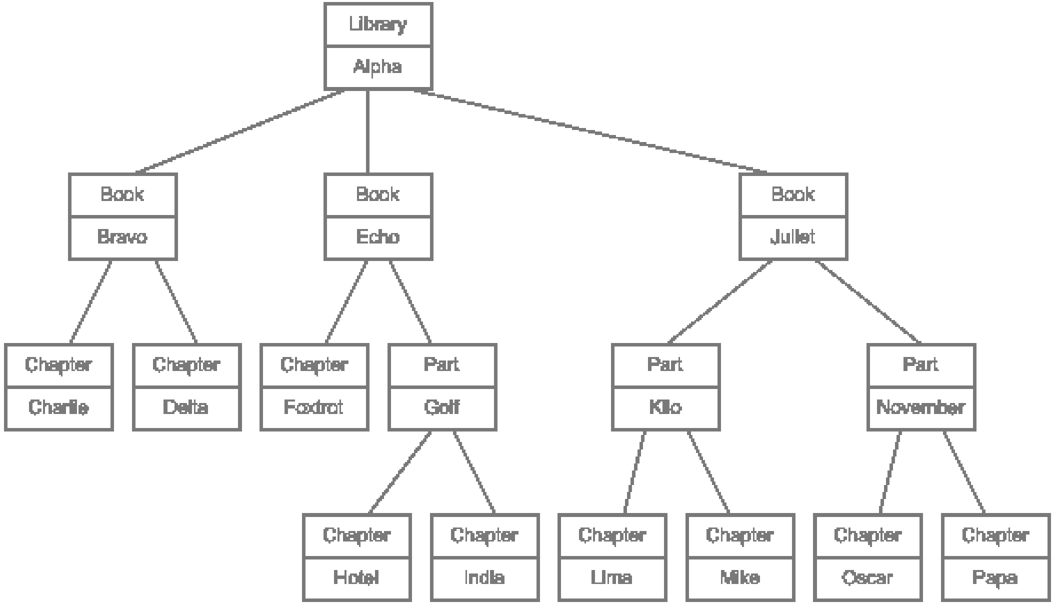
- **Data Structure**



1. Binary Tree

- **What is Tree?**

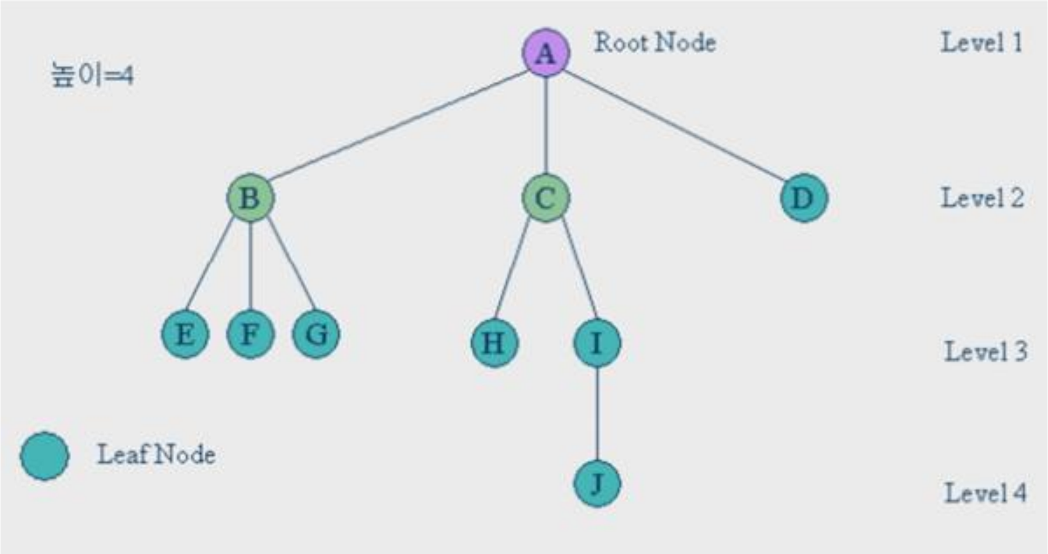




1. Binary Tree

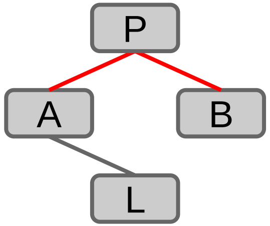
- **Tree (정의): 자료와 그 다음 자료의 위치 정보가 저장된 비선형의 자료구조**

- 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)

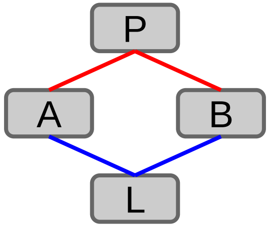


1. Binary Tree

- **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)

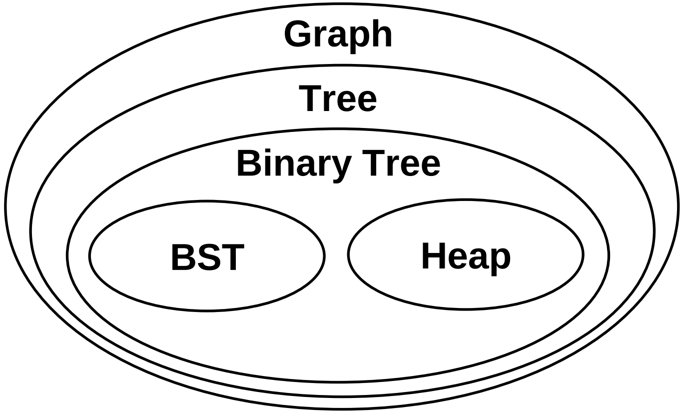


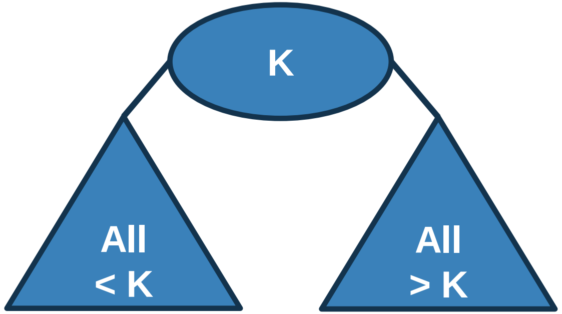
그래프(Graph)
나중에 배움

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

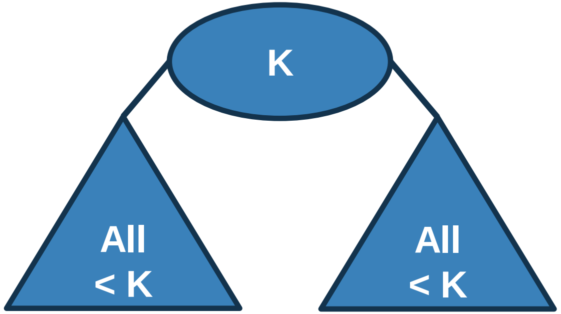
1. Binary Tree

- **비선형 자료구조의 포함관계**
  - Cycle이 존재하지 않는 Graph (=Tree)
  - 자식 노드가 2개만 존재하는 Tree (= Binary Tree)
  - 특수한 경우의 Binary Tree
    - Binary Search Tree
    - Heap (Max heap, Min Heap)





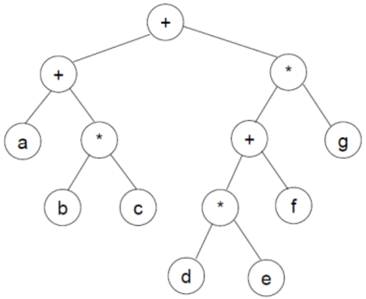
Binary Search Tree

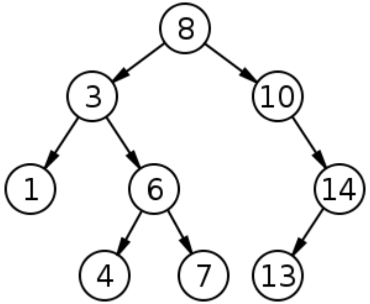


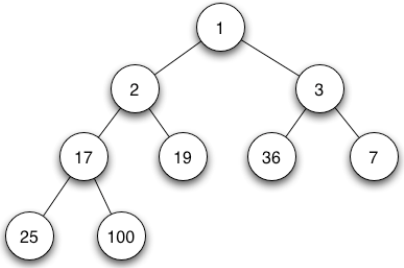
Heap (Max Heap)

1. 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

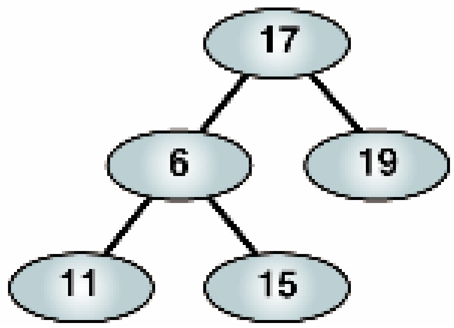




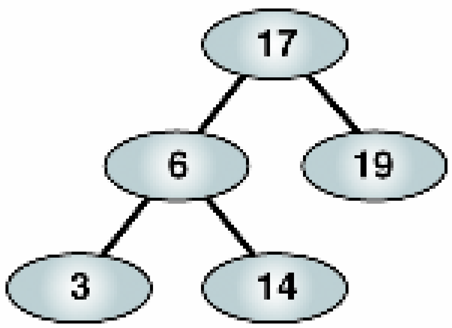


1. Binary Tree

- **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

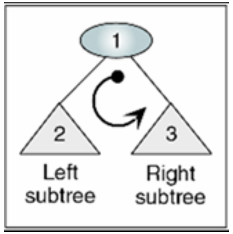


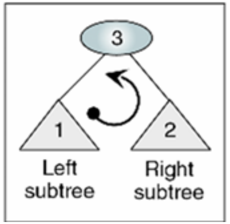
Binary search tree

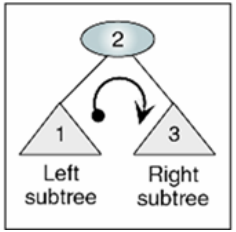
2. Traversal of BST

- **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 : <u>자기를 먼저</u>. 그 다음 왼쪽. 마지막에 오른쪽 탐색.
2. **In-order : 왼쪽 먼저 탐색. <u>그 다음 자기 자신</u>. 마지막에 오른쪽 탐색.**
3. Post-order : 왼쪽 먼저 탐색. 그 다음 오른쪽 탐색. <u>마지막에 자기 자신</u>

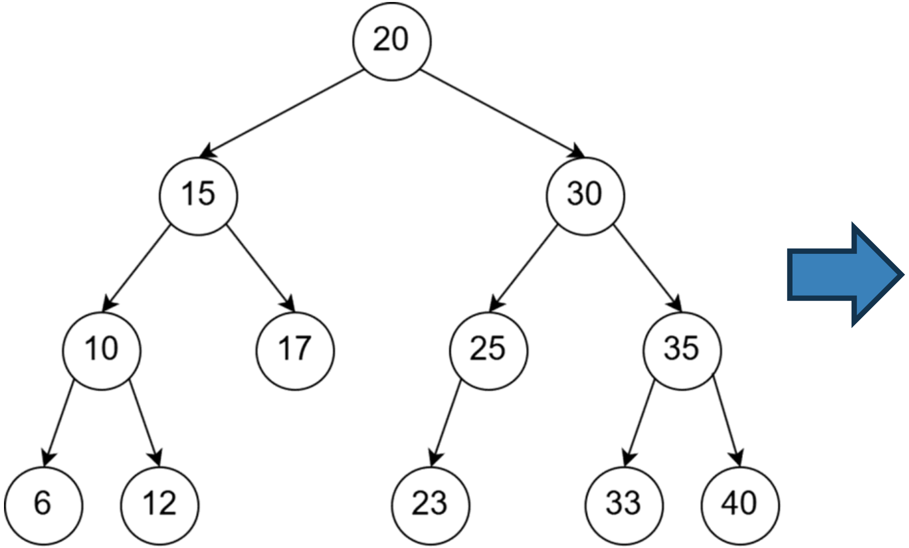






2. Traversal of BST

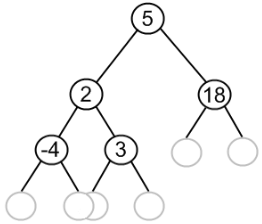
- **In-order traversal in BST**
  - Binary Search Tree에서 in-order traversal은 가장 일반적인 순회 방법



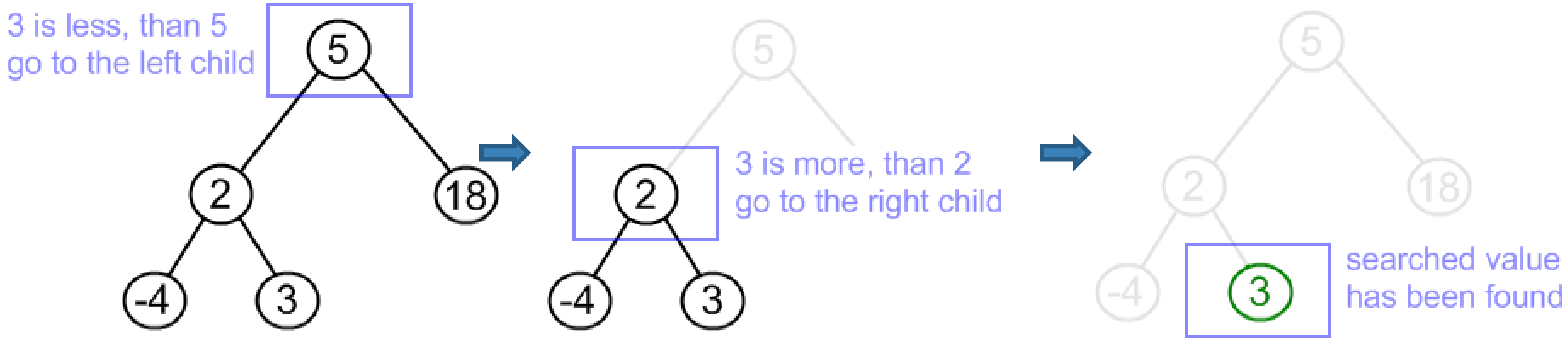
In-order traversal
: 6, 10, 12, 15, 17, 20, 23, 25, 30, 33, 35, 40

→ 자동으로 오름차순으로 정렬되어 있음

3. Operation of BST



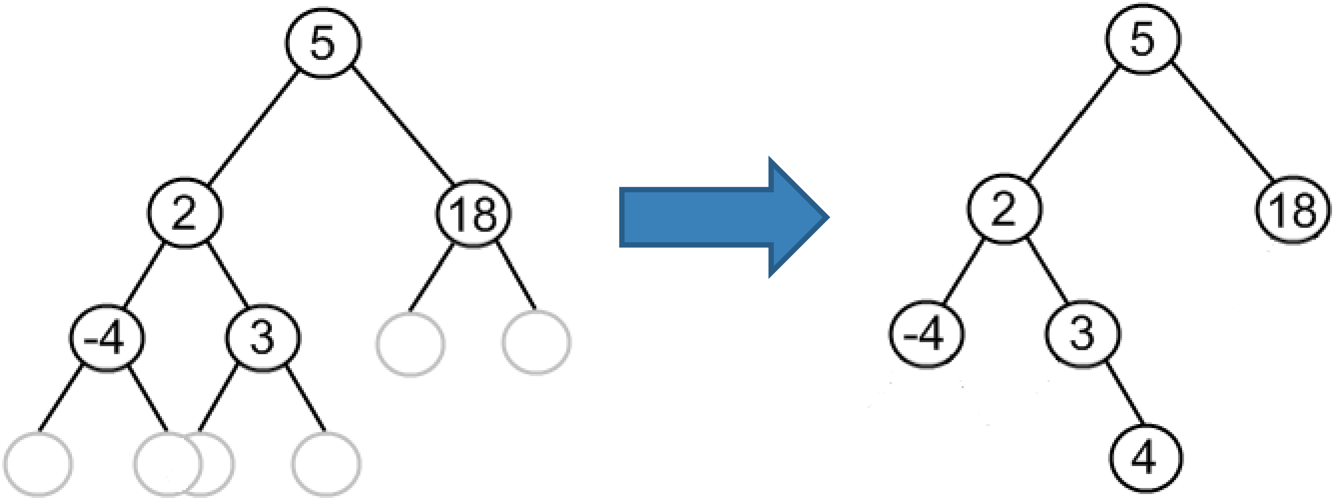
- **Binary Search Tree(이진 탐색 트리)**
  - Search Node (search 3)



→ $O(height) = O(\log n)$의 시간 복잡도

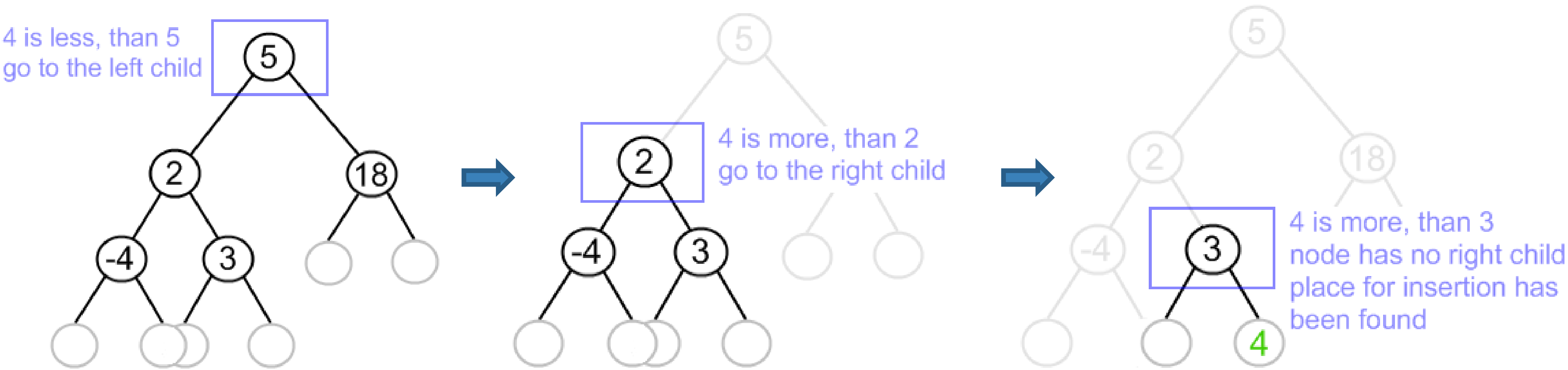
3. Operation of BST

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



3. Operation of BST

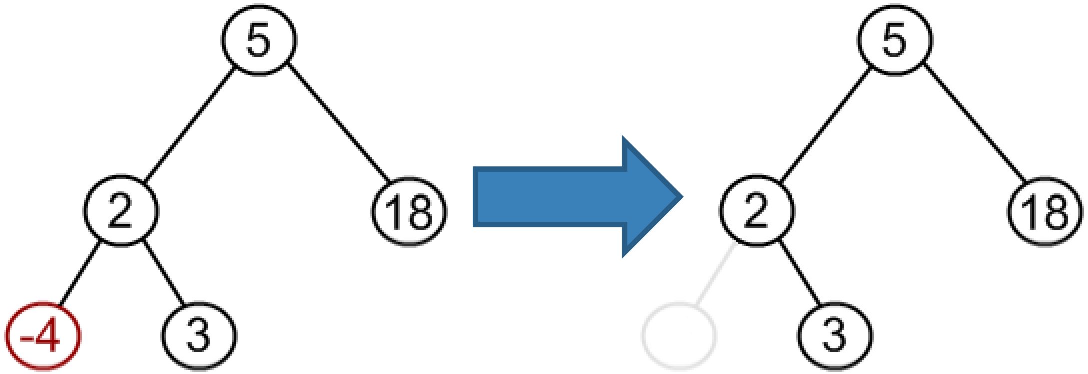
- **Binary Search Tree(이진 탐색 트리)**
  - Add Node (add 4)



→ $O(height)$ 의 시간 복잡도
(노드를 추가할 위치를 찾기 위한 Search 수행)

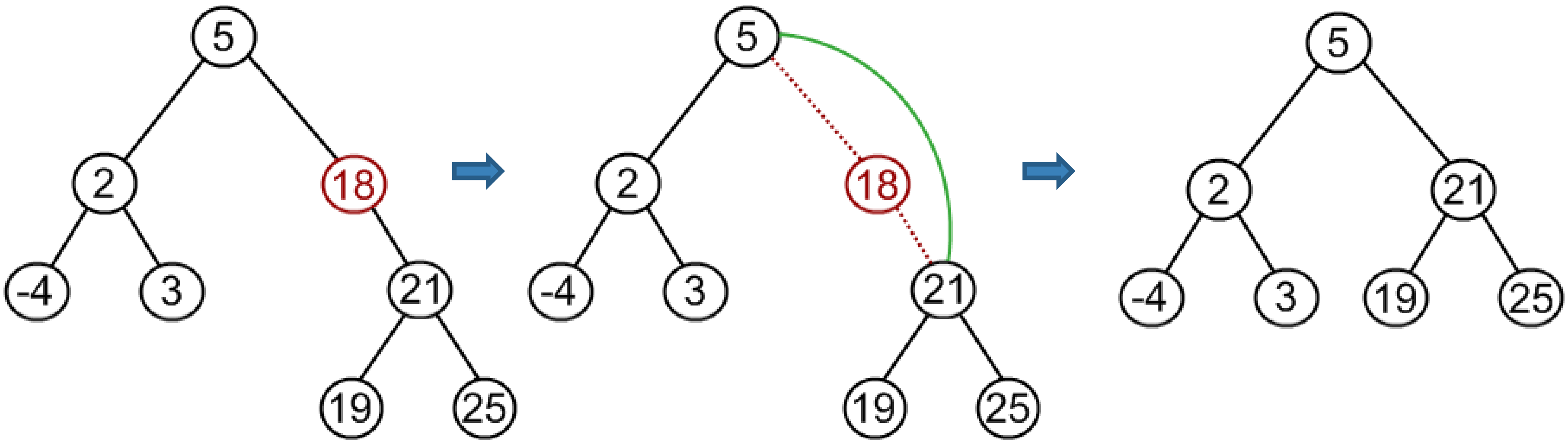
3. Operation of BST

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



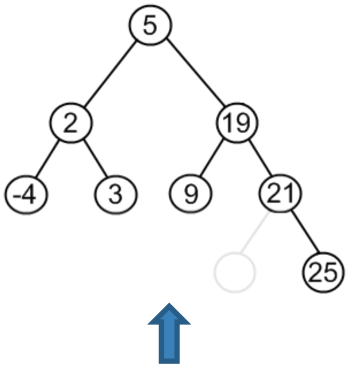
3. Operation of BST

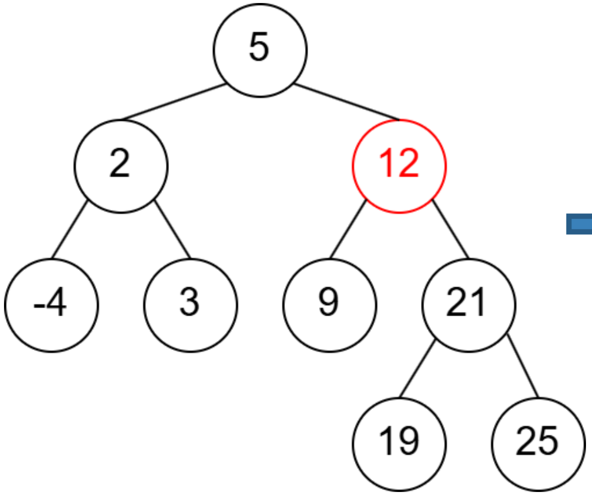
- **Binary Search Tree(이진 탐색 트리)**
  - Remove Node (remove 18) – **Case2.** child Node가 하나 존재하는 경우

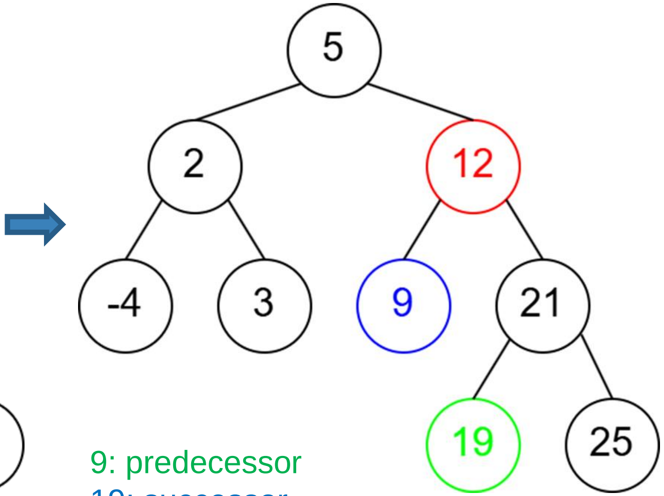


3. Operation of BST

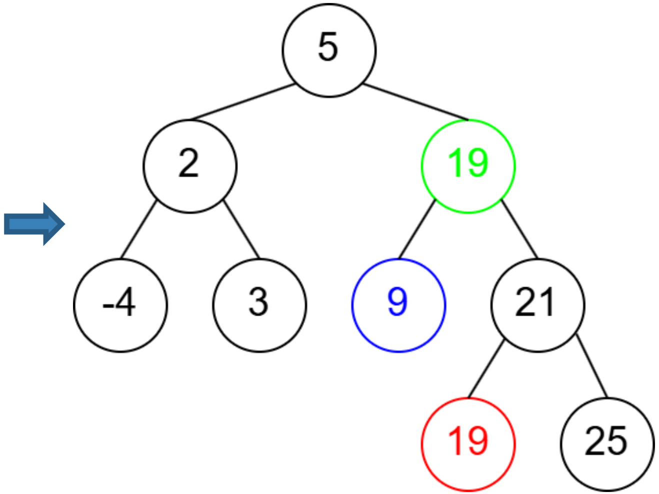
- **Binary Search Tree(이진 탐색 트리)**
  - Remove Node (remove 12) – **Case3.** <u>child Node가 둘 다 있는 경우</u>







9: predecessor
19: successor
→ 일반적으로 successor 선택

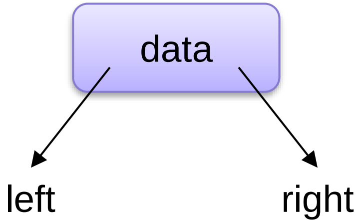


→ $O(height)$ 의 시간 복잡도
(삭제할 노드를 찾기 위한 Search 수행)

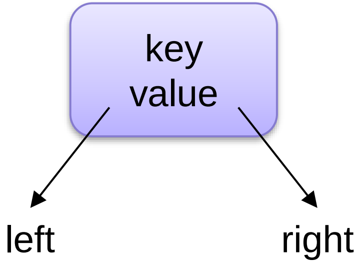
3. Binary Search Tree

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

```c
typedef struct Node
{
    int data;
    struct Node* left;
    struct Node* right;
} Node;
```




```c
typedef struct Node
{
    int key;
    int value;
    struct Node* left;
    struct Node* right;
} Node;
```



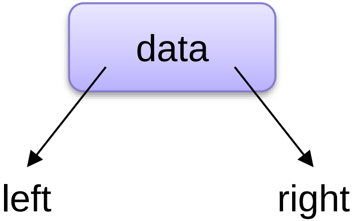
단일값 기반 BST 노드

Dictionary 기반 BST 노드

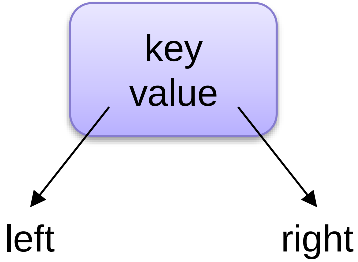
3. Binary Search Tree

- **Node Structure and C++ Correspondence**

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




```
typedef struct Node
{
    int key;
    int value;
    struct Node* left;
    struct Node* right;
} Node;
```



std::set

Defined in header `<set>`

```
template<
    class Key,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<Key>
> class set;
```

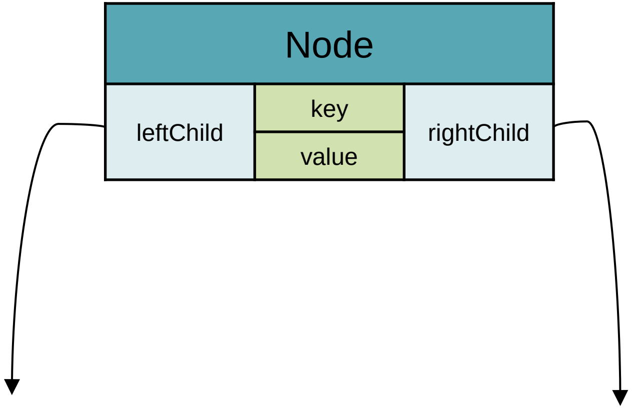
std::map

Defined in header `<map>`

```
template<
    class Key,
    class T,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<std::pair<const Key, T>>
> class map;
```

3. Binary Search Tree – 구현 실습

- **Tree Node Data Type**



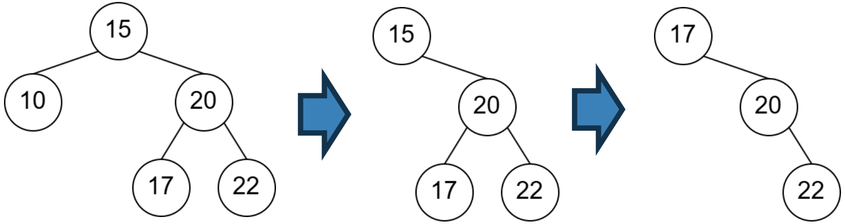

```c
#include<stdio.h>
#include<stdlib.h>

// Dictionary 구조
typedef struct Node{
    int key;
    int value;
    struct Node* leftChild;
    struct Node* rightChild;
} Node;


void insertTreeNode(Node** p, int key, int value);
void printTreeInorder(Node* p);
void deleteTreeNode(Node** p, int key);

void threeWayJoin(Node* mid, Node* small, Node* big);
void splitTree(Node* root, int key, Node** small, Node** mid, Node** big);
```

3. Binary Search Tree – 구현 실습

- **main**

Microsoft Visual Studio 디버그

```
Key: 10, Value: 200
Key: 15, Value: 100
Key: 17, Value: 300
Key: 20, Value: 400
Key: 22, Value: 500
-------------------
Key: 15, Value: 100
Key: 17, Value: 300
Key: 20, Value: 400
Key: 22, Value: 500
-------------------
Key: 17, Value: 300
Key: 20, Value: 400
Key: 22, Value: 500
-------------------
```




```c
int main() {
    Node* pParentNode = NULL;

    insertTreeNode(&pParentNode, 15, 100);
    insertTreeNode(&pParentNode, 10, 200);
    insertTreeNode(&pParentNode, 20, 300);
    insertTreeNode(&pParentNode, 17, 400);
    insertTreeNode(&pParentNode, 22, 500);

    printTreeInorder(pParentNode);
    printf("-------------------\n");

    // Degree가 1인 노드를 삭제하는 경우
    deleteTreeNode(&pParentNode, 10);
    printTreeInorder(pParentNode);
    printf("-------------------\n");

    //// root 노드를 삭제하는 경우
    deleteTreeNode(&pParentNode, 15);
    printTreeInorder(pParentNode);
    printf("-------------------\n");

    return 0;
}
```

3. Binary Search Tree – 구현 실습

- **Insert & In-order (재귀함수로 구현)**
  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

```c
void insertTreeNode(Node** p, int key, int value){
    // 노드가 NULL인 경우
    if ((*p) == NULL) {
        (*p) = (Node*)malloc(sizeof(Node));
        (*p)->key = key;
        (*p)->value = value;
        (*p)->leftChild = NULL;
        (*p)->rightChild = NULL;
    }
    // leftChild로 들어가는 경우
    else if ((*p)->key > key) {
        insertTreeNode(&((*p)->leftChild), key, value);
    }
    // rightChild로 들어가는 경우
    else {
        insertTreeNode(&((*p)->rightChild), key, value);
    }
}

void printTreeInorder(Node* p){
    if (p == NULL) return;

    printTreeInorder(p->leftChild);
    printf("Key: %d, Value: %d\n", p->key, p->value);
    printTreeInorder(p->rightChild);

}
```

3. Binary Search Tree – 구현

- **Delete**
  1. **삭제하려는 key의 노드의 존재 확인**
  2. **삭제하려는 key 노드의 위치에 따라 삭제 수행**
     I. 삭제하는 노드가 leaf node인 경우
     II. 삭제하는 노드의 degree가 1인 경우
     III. 삭제하는 노드의 degree가 2인 경우
  3. **삭제하려는 노드가 leaf node인 경우**
     I. 부모 노드의 자식 포인터를 NULL로 변경하고 삭제

```c
void deleteTreeNode(Node** p, int key)
{
    Node* pNode = *p;
    Node* pParent = NULL;
    Node* child = NULL;
    Node* successor = NULL;
    Node* successorParent = NULL;

    // key를 찾을 때까지 반복
    while (pNode != NULL && pNode->key != key) {
        pParent = pNode;
        if (key < pNode->key) {
            pNode = pNode->leftChild;
        }
        else {
            pNode = pNode->rightChild;
        }
    }

    // key를 찾지 못한 경우
    if (pNode == NULL) return;

    // Degree가 0인 경우 (리프 노드)
    if (pNode->leftChild == NULL && pNode->rightChild == NULL) {
        if (pParent == NULL) {
            free(*p);
            *p = NULL;
        }
        else if (pParent->leftChild == pNode) {
            free(pParent->leftChild);
            pParent->leftChild = NULL;
        }
        else {
            free(pParent->rightChild);
            pParent->rightChild = NULL;
        }
    }
```

3. Binary Search Tree – 구현

- **Delete**
  4. **삭제하는 노드의 degree가 1인 경우**
     I. 부모 노드가 자식 노드를 직접 가리키도록 연결
     II. 기존 노드를 삭제

  5. **삭제하는 노드의 degree가 2인 경우**
     I. 삭제할 노드의 successor (오른쪽 서브트리의 가장 작은 노드)에 해당하는 노드 탐색
     II. Successor의 데이터를 복사
     III. 부모 노드와 자식 노드를 연결
     IV. Successor 노드 삭제

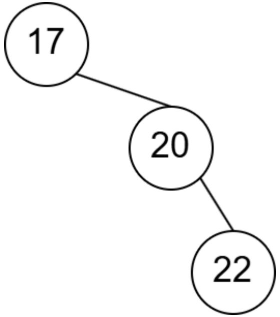
```c
    // Degree가 1인 경우 (자식이 하나인 경우)
    else if (pNode->leftChild == NULL || pNode->rightChild == NULL) {
        if (pNode->leftChild != NULL) child = pNode->leftChild;
        else child = pNode->rightChild;

        if (pParent == NULL) {
            free(*p);
            *p = child;
        }
        else if (pParent->leftChild == pNode) {
            free(pParent->leftChild);
            pParent->leftChild = child;
        }
        else {
            free(pParent->rightChild);
            pParent->rightChild = child;
        }
    }
    // Degree가 2인 경우 (자식이 두 개인 경우)
    else {
        successor = pNode->rightChild;
        successorParent = pNode;

        while (successor->leftChild != NULL) {
            successorParent = successor;
            successor = successor->leftChild;
        }

        // successor의 데이터를 복사
        pNode->key = successor->key;
        pNode->value = successor->value;

        // successor를 제거 (successor는 리프 또는 자식 하나로 가정)
        if (successorParent->leftChild == successor) {
            successorParent->leftChild = successor->rightChild;
        }
        else {
            successorParent->rightChild = successor->rightChild;
        }
        free(successor);
    }
}
```

3. Binary Search Tree

- **Height of Binary Search Tree**
  - n개의 노드를 갖는 BST는 평균적으로 $O(\log n)$의 시간 복잡도
  - 어느 한쪽 방향의 자식 노드만 존재하는 경우 $O(height) = O(n)$ (Worst Case)

- **Solution**
  - 트리의 balance가 무너져 탐색이 저하되는 경우 트리를 재구성하여 해결 가능
  - threeWayJoin: 분리된 트리를 다시 병합
  - splitTree: 특정 key를 기준으로 트리를 분할

→ 시간 복잡도를 다시 $O(\log n)$으로 조정



3. Binary Search Tree – 구현 실습

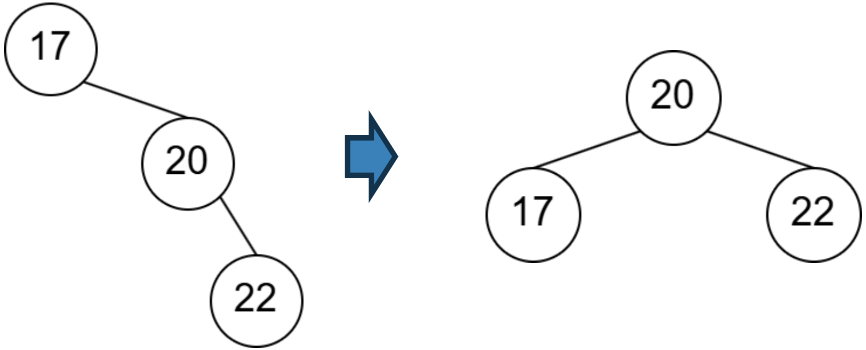
- **threeWayJoin**
  1. 중간 노드(mid)를 기준으로 small과 big트리를 연결하여 하나의 BST로 통합

- **splitTree**
  1. 재귀적으로 탐색하여 주어진 key를 기준으로 트리를 세 부분으로 분리
     I. key 보다 작은 노드들 (small)
     II. key와 일치하는 노드 (mid)
     III. key 보다 큰 노드들 (big)

```
void threeWayJoin(Node* mid, Node* small, Node* big) {
    if (mid == NULL) return;
    mid->leftChild = small;
    mid->rightChild = big;
}

void splitTree(Node* root, int key, Node** small, Node** mid, Node** big) {
    if (root == NULL) {
        *small = *mid = *big = NULL;
        return;
    }

    if (key < root->key) {
        splitTree(root->leftChild, key, small, mid, &(root->leftChild));
        *big = root;
    }
    else if (key > root->key) {
        splitTree(root->rightChild, key, &(root->rightChild), mid, big);
        *small = root;
    }
    else {
        *mid = root;
        *small = root->leftChild;
        *big = root->rightChild;
    }
}
```

3. Binary Search Tree – 구

- main (with Split & Join)




```c
int main() {
    Node* pParentNode = NULL;
    Node* left = NULL, * mid = NULL, * right = NULL;
    Node* joined = NULL;

    insertTreeNode(&pParentNode, 15, 100);
    insertTreeNode(&pParentNode, 10, 200);
    insertTreeNode(&pParentNode, 20, 400);
    insertTreeNode(&pParentNode, 17, 300);
    insertTreeNode(&pParentNode, 22, 500);

    printTreeInorder(pParentNode);
    printf("------------------\n");

    // Degree가 1인 노드를 삭제하는 경우
    deleteTreeNode(&pParentNode, 10);
    printTreeInorder(pParentNode);
    printf("------------------\n");

    //// root 노드를 삭제하는 경우
    deleteTreeNode(&pParentNode, 15);
    printTreeInorder(pParentNode);
    printf("------------------\n");

    // Tree를 split하고 Join
    splitTree(pParentNode, 20, &left, &mid, &right);
    joined = mid;
    threeWayJoin(joined, left, right);
    printTreeInorder(joined);

    return 0;
}
```