

안녕하세요, 오늘은 Tree 에 대해 다루어보도록 하겠습니다.

1. 기초 강의

강의 콘텐츠 확인 > 3. Tree

Link

https://swexpertacademy.com/main/learn/course/subjectDetail.do?courseId=CONTENTS_REVIEW&subjectId=AYVXaRK6QSQDFARs

※ 출석은 강의 수강 내역으로 확인합니다.

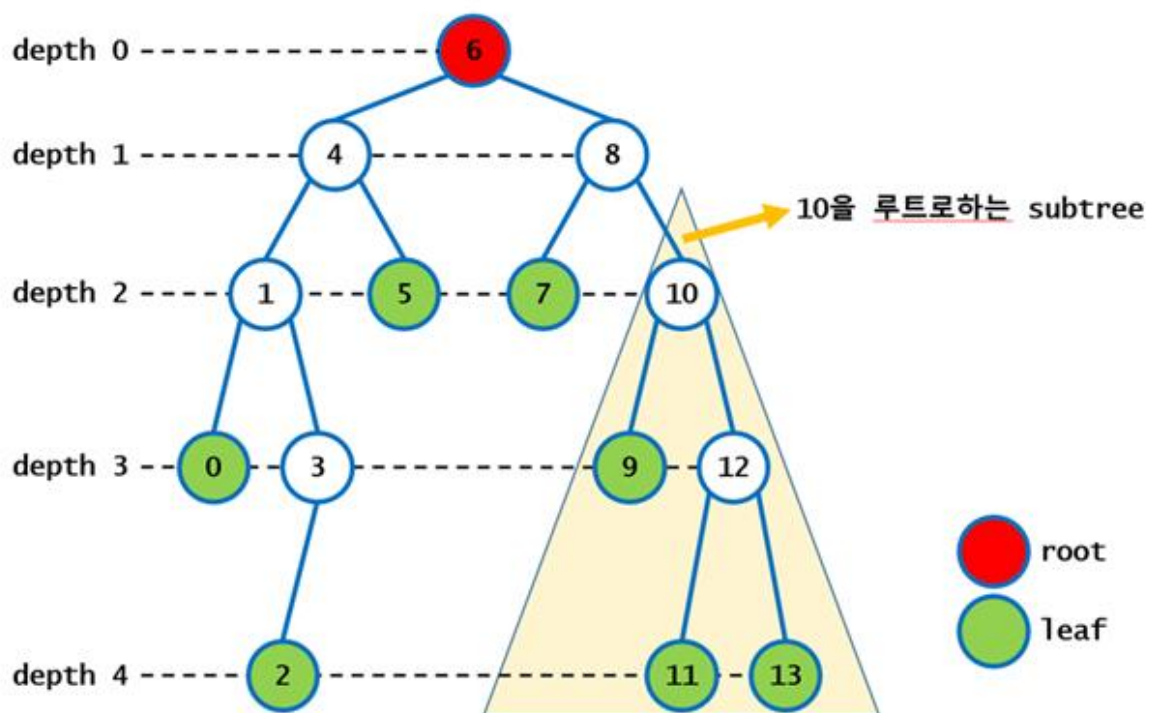
2. 심화 강의

트리는 사이클이 없는 연결 그래프입니다.

정의를 풀어서 설명하면, 각 노드가 하나의 부모 노드와 간선으로 연결되어있는 자료구조입니다. 예외적으로 가장 위의 노드는 부모를 가지지 않습니다.

트리를 활용한 자료구조는 정말 많습니다. 오늘은 그중에서 가장 기본적인 Binary Search Tree 를 구현해보겠습니다.

2.1 Tree 용어 정리



◆ **parent node / child node (부모 노드 / 자식 노드)**

연결된 두 노드 중 위에 있는 노드를 부모 노드, 아래에 있는 노드를 자식 노드라고 합니다.

예) 4 은 5 의 부모 노드이고, 5 는 4 의 자식 노드이다.

Binary Search Tree 는 각 노드가 최대 2 개의 자식 노드만 갖도록 제한한 트리입니다. 2 개의 자식 노드를 각각 left child, right child 라고 부릅니다.

◆ **ancestor / descendent (조상 / 자손)**

노드 V 에서 부모 노드로만 계속 이동해서 노드 U 로 갈 수 있다면 U 는 V 의 조상이고, V 는 U 의 자손입니다.

예) 1 의 조상은 1, 4, 6 이다. 10 의 자손은 10, 9, 12, 11, 13 이다.

◆ **root node (루트)**

부모 노드가 없는 노드입니다. 트리 가장 위에 위치합니다.

◆ **leaf node (리프)**

자식 노드가 없는 노드입니다.

◆ **depth of a node (깊이)**

루트 노드로부터 해당 노드까지 이동하기 위해 거쳐야 하는 간선의 수 입니다. 루트 노드의 깊이는 0 입니다. level 이라고 부르기도 하며 루트 노드의 깊이를 1 이라고 하는 경우도 있습니다.

◆ **height of a node (높이)**

해당 노드부터 가장 먼 리프 노드까지 이동하기 위해 거쳐야 하는 간선의 수입니다. 리프 노드의 높이는 0 입니다. 깊이와 마찬가지로 리프 노드의 높이를 1 이라고 하는 경우도 있습니다.

예) 1 의 높이는 1, 7 의 높이는 0 이다.

루트 노드의 높이를 트리의 높이라고 합니다.

◆ **subtree (서브트리)**

어떤 노드와 부모 노드간의 연결을 끊으면 해당 노드를 루트 노드로 하는 새로운 트리가 만들어집니다. 이것을 서브트리라고 합니다.

◆ **size of tree (트리의 크기)**

트리의 크기는 노드의 개수와 같습니다. 위의 트리는 크기가 14 인 트리입니다.

2.2 Binary Search Tree

Binary Search Tree 는 원소의 중복을 허용하지 않으며, 왼쪽 서브트리는 자기보다 작은 값을, 오른쪽 서브트리는 자기보다 큰 값을 저장하는 트리입니다.

C++ std::set 에는 Binary Search Tree 중 하나인 Red Black Tree 가 쓰입니다.

2.3 초기화

```
struct Node {  
    int key;  
    Node *left, *right;  
};  
  
// 정적 할당 방식  
constexpr size_t MAX_NODE = 1000;  
  
int node_count = 0;  
Node node_pool[MAX_NODE];  
  
Node* new_node(int x) {  
    node_pool[node_count].key = x;  
    node_pool[node_count].left = nullptr;  
    node_pool[node_count].right = nullptr;  
  
    return &node_pool[node_count++];  
}  
  
// 루트 노드를 가리키는 포인터
```

```
Node* root;
```

```
void init() {  
    root = nullptr;  
    node_count = 0;  
}
```

지난 연결 리스트 강의처럼 정적 할당을 사용하겠습니다.

init() 함수를 호출하면 루트 노드를 NULL 로 만듭니다. 즉, 트리에 아무 노드가 없는 상태가 됩니다.

2.4 삽입

```
// O(트리의 높이)
```

```
void insert(int x) {  
    root = insert_rec(root, x);  
}
```

```
Node* insert_rec(Node* node, int x) {  
    if (node == nullptr) {  
        return new_node(x);  
    }  
  
    if (x < node->key) {  
        node->left = insert_rec(node->left, x);  
    } else if (x > node->key) {  
        node->right = insert_rec(node->right, x);  
    }  
}
```

```

    }

    return node;
}

```

재귀로 구현한 노드 삽입입니다.

Node* insert_rec(Node* node, int x) 함수는 node 를 루트로 하는 서브트리에 x 를 삽입한 다음 서브트리의 루트를 리턴하는 함수입니다.

동작 과정 케이스를 3 개로 나눠서 이해해봅시다.

1) node = NULL 인 경우

이는 서브트리가 비어있는 경우입니다. x 를 추가하면 노드가 x 하나뿐인 서브트리가 생기므로 x 노드를 만든 뒤 바로 리턴해주면 됩니다.

2) node 의 왼쪽 자식에 x 가 들어가야 하는 경우 ($x < \text{node} \rightarrow \text{key}$ 인 경우)

왼쪽 서브트리에 x 를 삽입한 뒤 왼쪽 서브트리의 루트를 왼쪽 자식으로 만들면 됩니다. 이는 insert_rec 함수를 그대로 사용하여 재귀적으로 풀 수 있습니다. 자식의 서브트리에 노드를 삽입해도 루트(node)는 변하지 않으므로 node 를 그대로 리턴하면 됩니다.

node 의 오른쪽 자식에 x 가 들어가야 하는 경우도 같은 논리로 해결됩니다.

3) node 의 key 와 x 가 같은 경우

이미 트리에 x 가 존재하는 경우입니다. Binary Search Tree 는 원소의 중복을 허용하지 않으므로 아무 것도 하지 않고 node 를 그대로 리턴합니다.

2.5 삭제

// O(트리의 높이)

```

void remove(int x) {
    root = remove_rec(root, x);
}

Node* remove_rec(Node* node, int x) {

```

```

    if (node == nullptr) {
        return node;
    }

    if (x < node->key) {
        node->left = remove_rec(node->left, x);
    } else if (x > node->key) {
        node->right = remove_rec(node->right, x);
    } else {
        if (node->left == nullptr) {
            return node->right;
        } else if (node->right == nullptr) {
            return node->left;
        }

        const int temp = find_min_key(node->right);
        node->key = temp;
        node->right = remove_rec(node->right, temp);
    }

    return node;
}

```

```

int find_min_key(Node* node) const {
    while (node->left != nullptr) {
        node = node->left;
    }

    return node->key;
}

```

}

삽입과 비슷합니다. Node* remove_rec(Node* node, int x) 함수는 node 를 루트로 하는 서브트리에 x 를 삭제한 다음 서브트리의 루트를 리턴하는 함수입니다. 역시 3 가지 경우로 나눠서 이해해봅시다.

1) node = NULL 인 경우

이는 x 가 원래 트리에 없는 경우입니다. 아무 것도 하지 않고 서브트리의 루트(NULL)을 리턴하면 됩니다.

2) node 의 왼쪽 자식에서 x 를 삭제해야 하는 경우 ($x < \text{node} \rightarrow \text{key}$ 인 경우)

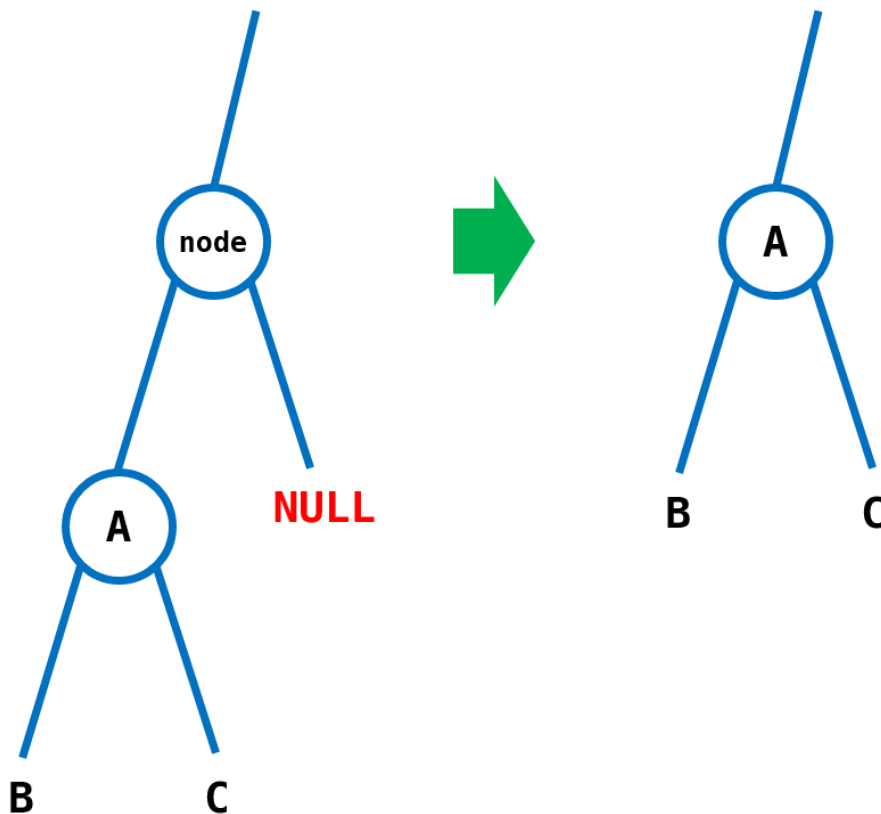
왼쪽 서브트리에 x 를 삭제한 다음 왼쪽 자식을 그 서브트리의 루트로 바꿔주면 됩니다. 삽입과 마찬가지로 재귀적으로 풀 수 있습니다.

node 의 오른쪽 자식에서 x 를 삭제해야 하는 경우도 같은 논리로 해결됩니다.

3) node 의 key 와 x 가 같은 경우

node 를 삭제해야 합니다. node 를 삭제하고 node 의 부모 노드와 node 의 자식 노드를 트리 구조에 맞게 이어줘야 합니다. 이는 아래 2 가지 경우로 또 나뉩니다.

3-1) node 의 한 쪽 자식이 없는 경우



노드를 삭제해도 됩니다. 트리를 1. 루트 2. 왼쪽 서브트리 3. 오른쪽 서브트리 이렇게 세 부분으로 나눌 수 있습니다. 루트 노드는 지금 삭제할 것이고, 한쪽 서브트리는 이미 없으니 나머지 하나의 서브트리만 남습니다. 그 서브트리가 원래의 트리를 대체하게 됩니다.

3-2) node 에게 왼쪽 자식, 오른쪽 자식 모두 있는 경우

node 를 삭제하고 그 자리에 node 의 왼쪽 자식이 와야 할까요 오른쪽 자식이 와야 할까요? 둘 다 안됩니다. Binary Search Tree 구조를 유지하기 위해 node 자리에는 node 의 왼쪽 서브트리의 모든 값보다 크고, node 의 오른쪽 서브트리의 모든 값보다 작은 값이 와야 합니다.

이를 위해서 node 의 key 를 오른쪽 서브트리에서 가장 작은 key 값으로 바꾸고, 오른쪽 서브트리에서 그 key 를 대신 삭제하는 대안을 사용합니다. (왼쪽 서브트리에서 가장 큰 key 값을 찾아도 됩니다)

2.6 탐색

// O(트리의 높이)

```
bool find(int x) {  
    Node* node = root;  
    while (node != nullptr) {  
        if (node->key == x) {  
            return true;  
        }  
        node = x < node->key ? node->left : node->right;  
    }  
    return false;  
}
```

2.7 트리 순회

Binary Tree 를 순회하는 방법에는 크게 3 가지가 있습니다.

1) pre-order (전위 순회)

자신 -> 왼쪽 서브트리 -> 오른쪽 서브트리 순서로 방문합니다.

pre-order 방문 순서는 위상 정렬 결과와 같습니다.

2.) in-order (중위 순회)

왼쪽 서브트리 -> 자신 -> 오른쪽 서브트리 순서로 방문합니다.

Binary Search Tree 에서 in-order 방문 순서는 key 를 정렬한 결과와 같습니다.

3) post-order (후위 순회)

왼쪽 서브트리 -> 오른쪽 서브트리 -> 자신 순서로 방문합니다.

post-order 순회는 자식 서브트리를 모두 방문한 후에 자신을 방문하므로 자식 노드에서 계산된 결과를 자신이 활용할 수 있습니다. 이를 이용한 대표적인 예로 계산기 구현, 세그먼트 트리가 있습니다.

위의 순회 방법은 재귀 호출 단계에서 자신을 언제 호출하냐에 따라 간단히 구현할 수 있습니다.

```
void traversal_rec(Node* node) const {  
    if (node == nullptr) return;  
    // if (type == 0) std::cout << node->key << ' '; // 이 주석을 풀면 pre-order  
    traversal_rec(node->left, type);  
    // if (type == 1) std::cout << node->key << ' '; // 이 주석을 풀면 in-order  
    traversal_rec(node->right, type);  
    // if (type == 2) std::cout << node->key << ' '; // 이 주석을 풀면 post-order  
}
```

비재귀로 트리 순회를 구현할 수 있습니다. 스택 자료구조를 사용하면 되죠. pre-order 순회를 스택으로 구현한 함수입니다.

// 스택을 사용한 트리 순회

```
void pre_order() {  
    std::stack<Node*> stk;  
    stk.emplace(root);
```

```

std::cout << "pre-order  ";

while (!stk.empty()) {

    const Node* node = stk.top();

    stk.pop();

    std::cout << node->key << ' ';

    if (node->right != nullptr) stk.emplace(node->right);
    if (node->left != nullptr) stk.emplace(node->left);
}

std::cout << '\n';
}

```

pre-order 뿐만 아니라 in-order, post-order 모두 스택을 사용해 구현할 수 있습니다!

2.8 Binary Search Tree 구현

위의 함수를 종합한 결과입니다! 자유롭게 테스트해보세요.

```

#include <array>

#include <iostream>

#include <string>

struct Node {

    int key;

    Node *left, *right;

};

```

```
constexpr size_t MAX_NODE = 1000;
```

```
int node_count = 0;
```

```
Node node_pool[MAX_NODE];
```

```
Node* new_node(int x) {  
    node_pool[node_count].key = x;  
    node_pool[node_count].left = nullptr;  
    node_pool[node_count].right = nullptr;  
  
    return &node_pool[node_count++];  
}
```

```
class BinarySearchTree {  
    Node* root;  
  
public:  
    BinarySearchTree() = default;  
  
    void init() {  
        root = nullptr;  
        node_count = 0;  
    }  
  
    void insert(int x) {  
        root = insert_rec(root, x);  
    }  
};
```

```
}
```

```
void remove(int x) {
```

```
    root = remove_rec(root, x);
```

```
}
```

```
bool find(int x) const {
```

```
    Node* node = root;
```

```
    while (node != nullptr) {
```

```
        if (node->key == x) {
```

```
            return true;
```

```
        }
```

```
        node = x < node->key ? node->left : node->right;
```

```
    }
```

```
    return false;
```

```
}
```

```
void traversal(int type) const {
```

```
    static constexpr std::array<const char*, 3> traversal_types = {"pre", "in", "post"};
```

```
    std::cout << traversal_types[type] << "-order  ";
```

```
    traversal_rec(root, type);
```

```
    std::cout << '\n';
```

```
}
```

private:

```
Node* insert_rec(Node* node, int x) {
```

```
    if (node == nullptr) {
```

```

        return new_node(x);
    }

    if (x < node->key) {
        node->left = insert_rec(node->left, x);
    } else if (x > node->key) {
        node->right = insert_rec(node->right, x);
    }

    return node;
}

```

```

Node* remove_rec(Node* node, int x) {
    if (node == nullptr) {
        return node;
    }

    if (x < node->key)
        node->left = remove_rec(node->left, x);
    else if (x > node->key)
        node->right = remove_rec(node->right, x);
    else {
        if (node->left == nullptr) {
            return node->right;
        } else if (node->right == nullptr) {
            return node->left;
        }
    }
}

```

```

        const int temp = find_min_key(node->right);

        node->key = temp;

        node->right = remove_rec(node->right, temp);

    }

    return node;

}

int find_min_key(Node* node) const {

    while (node->left != nullptr) {

        node = node->left;

    }

    return node->key;

}

void traversal_rec(Node* node, int type) const {

    if (node == nullptr) return;

    if (type == 0) std::cout << node->key << ' ';

    traversal_rec(node->left, type);

    if (type == 1) std::cout << node->key << ' ';

    traversal_rec(node->right, type);

    if (type == 2) std::cout << node->key << ' ';

}

};

int main() {

    BinarySearchTree tree;

```

```

// 0 : 초기화

// 1 x : x 삽입

// 2 x : x 삭제

// 3 x : x 탐색

// 4 t : 순회 (t: 0 전위, 1 중위, 2 후위)

int cmd, x;

for (;;) {

    std::cin >> cmd;

    switch (cmd) {

        case 0:

            tree.init();

            break;

        case 1:

            std::cin >> x;

            tree.insert(x);

            break;

        case 2:

            std::cin >> x;

            tree.remove(x);

            break;

        case 3:

            std::cin >> x;

            std::cout << (tree.find(x) ? "found" : "not found") << '\n';

            break;

        case 4:

            std::cin >> x;

            if (x < 0 || x > 2) return std::cout << "invalid traversal type\n", 0;

```

```

        tree.traversal(x);

        break;

    default:

        return std::cout << "invalid command\n", 0;

    }

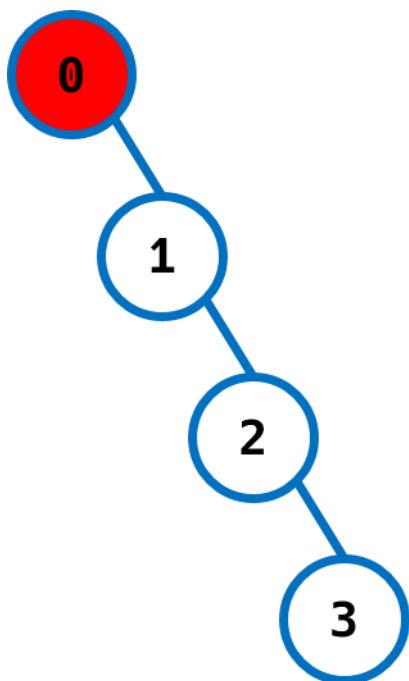
}

```

2.9 Self-Balanced Binary Search Tree

Binary Search Tree 에서 원소의 삽입/삭제/탐색의 시간복잡도가 모두 $O(\text{트리의 높이})$ 입니다. 트리의 높이는 얼마일까요?

위에서 만든 Binary Search Tree 에 0, 1, 2, 3, ... 을 순서대로 삽입해봅시다. 그러면 아래처럼 오른쪽 자식으로 계속 이어지는 트리가 만들어집니다 이는 마치 연결 리스트와 같죠. 시간복잡도가 $O(N)$ 이 됩니다.



그렇다면 최소 높이는 얼마일까요. 높이가 H 인 트리에 노드를 최대한 넣으면 크기가 $2^{(H + 1)} - 1$ 인 트리가 됩니다.

$$N \leq 2^{H+1} - 1$$

$$H \geq \lceil \log_2(N + 1) - 1 \rceil \geq \lceil \log_2 N \rceil$$

높이의 하한은 $\log N$ 입니다. 이렇게 높이가 $O(\log N)$ 인 트리를 height-balanced 하다고 합니다. 이런 트리에서는 삽입/삭제/탐색이 모두 $O(\log N)$ 이 보장되고, 원소가 삽입/삭제될 때마다 height-balanced 하게 높이를 조절하는 Binary Search Tree 를 Self-Balanced Binary Search Tree 라고 합니다. 이런 트리의 예시로는 Red-Black Tree, Splay Tree, B-Tree 등이 있습니다.

3. 기본 문제

- 중위순회
- 사칙연산 유효성 검사
- 사칙연산
- 공통 조상

4. 응용 문제

- Directory