

# [자료구조]

이진탐색트리

안수빈 202312720

# 목차

- 1 균형 잡힌 트리
- 2 합 경로 세기
- 3 한영/영한 사전 검색

균형 잡힌 트리

01

## 균형 잡힌 트리 - 문제

삽입, 삭제, 검색 기능을 지원하는 이진탐색트리에서 트리가 균형 (balanced) 잡혀 있는지 확인할 수 있는 `isBalanced()` 함수를 구현하고자 한다.

이진 탐색 트리에서 균형 잡힘의 정의는 다음과 같다.

- \* 현재 노드를 기준으로 왼쪽 서브트리는 balanced여야 한다.
- \* 현재 노드를 기준으로 오른쪽 서브트리는 balanced여야 한다.
- \* 현재 노드를 기준으로 왼쪽 서브트리와 오른쪽 서브트리의 높이 (height)의 차이가 1이하인 상태를 말 한다.

# 균형 잡힌 트리 - 문제

## Input

입력의 첫 줄에는 하고자 하는 작업의 수  $n$  ( $1 \leq n \leq 20$ )이 들어온다.

다음의  $n$  줄에는 다음과 같은 작업 명령이 들어온다.

I  $x$  : key  $x$ 를 이진 탐색 트리에 삽입한다. (value는 key와 같다)

D  $x$  : key  $x$ 를 이진 탐색 트리에서 삭제한다. (value는 key와 같다)

## Output

트리가 균형 잡혀있는지를 출력한다.

균형이 잡혀있을 경우 Balanced를 출력하고, 그렇지 않은 경우 Unbalanced를 출력한다.

# 균형 잡힌 트리 - 알고리즘 (1)

[클래스 `BinaryTree`]

`getHeight(BinaryNode* node)`

설명: 주어진 노드의 높이를 반환합니다.

알고리즘:

- 노드가 `nullptr`이면 0을 반환합니다.
- 왼쪽 서브트리와 오른쪽 서브트리의 높이를 각각 계산합니다.
- 더 큰 높이에 1을 더하여 반환합니다.

`isBalanced(BinaryNode* node)`

설명: 주어진 노드를 루트로 하는 서브트리가 균형 잡혀 있는지 확인합니다.

알고리즘:

- 노드가 `nullptr`이면 `true`를 반환합니다.
- 왼쪽 및 오른쪽 서브트리의 높이를 계산합니다.
- 높이 차이가 1 이하이고, 왼쪽 및 오른쪽 서브트리가 모두 균형 잡혀 있으면 `true`를 반환합니다.
- 그렇지 않으면 `false`를 반환합니다.

## 균형 잡힌 트리 - 알고리즘 (2)

insertNode(BinaryNode\* node, int key)

설명: 주어진 키를 가진 새 노드를 트리에 삽입합니다.

알고리즘:

- 노드가 nullptr이면 새로운 노드를 생성하여 반환합니다.
- 키가 현재 노드의 데이터보다 작으면 왼쪽 서브트리에 삽입합니다.
- 키가 현재 노드의 데이터보다 크면 오른쪽 서브트리에 삽입합니다.
- 수정된 서브트리의 루트를 반환합니다.

## 균형 잡힌 트리 - 알고리즘 (3)

`deleteNode(BinaryNode* node, int key)`

설명: 주어진 키를 가진 노드를 트리에서 삭제합니다.

알고리즘:

- 노드가 `nullptr`이면 노드를 반환합니다.
- 키가 현재 노드의 데이터보다 작으면 왼쪽 서브트리에서 삭제합니다.
- 키가 현재 노드의 데이터보다 크면 오른쪽 서브트리에서 삭제합니다.
- 키가 현재 노드의 데이터와 같으면:
  - 자식이 없는 경우: 노드를 삭제하고 `nullptr`을 반환합니다.
  - 하나의 자식만 있는 경우: 자식을 현재 노드와 교체하고 현재 노드를 삭제합니다.
  - 두 자식이 있는 경우: 오른쪽 서브트리에서 최소값을 찾아 현재 노드와 교체하고, 최소값을 삭제합니다.

- 수정된 서브트리의 루트를 반환합니다.

## 균형 잡힌 트리 - 알고리즘 (4)

findMin(BinaryNode\* node)

설명: 주어진 노드의 서브트리에서 최소값을 가진 노드를 찾습니다.

알고리즘:

- 현재 노드의 왼쪽 자식이 nullptr이면 현재 노드를 반환합니다.
- 왼쪽 자식이 nullptr이 아닐 때까지 왼쪽 자식을 따라 내려갑니다.
- 최소값을 가진 노드를 반환합니다.

## 균형 잡힌 트리 - 알고리즘 (3)

main()

설명: 트리의 노드 삽입 및 삭제를 처리하고, 트리가 균형 잡혀 있는지 확인합니다.

알고리즘:

사용자로부터 노드 개수  $n$ 을 입력받습니다.

BinaryTree 객체를 생성합니다.

$n$ 개의 노드에 대해 삽입(I) 또는 삭제(D) 연산을 수행합니다.

트리가 균형 잡혀 있는지 확인하여 결과를 출력합니다.

# 균형 잡힌 트리 - 흐름도

시작

- └ 노드가 NULL인가?
  - ├ 예: 0 반환
  - └ 아니오: 다음 단계로 진행
    - ├ 현재 노드 값 currentSum에 추가
    - ├ currentSum - targetSum을 해시 맵에서 찾기
      - ├ 해당 합을 가진 경로의 수를 경로의 총 수에 더함
      - └ 현재 합이 targetSum과 같은가?
        - ├ 예: 경로의 총 수 증가
        - └ 아니오: 계속 진행
    - ├ 해시 맵 업데이트 (currentSum 증가)
    - ├ 왼쪽 자식에 대해 재귀 호출
    - ├ 오른쪽 자식에 대해 재귀 호출
    - └ 해시 맵 업데이트 (currentSum 감소)

종료

```
class BinaryNode //BinaryNode 클래스 정의
{
protected:
    int data; //노드의 데이터 값
    BinaryNode* left; //왼쪽 자식 노드
    BinaryNode* right; //오른쪽 자식 노드
public:
    //생성자: 데이터 값과 왼쪽, 오른쪽 자식 노드를 초기화
    BinaryNode(int val = 0, BinaryNode* l = nullptr, BinaryNode* r = nullptr)
        : data(val), left(l), right(r) {}
    ~BinaryNode() {} //소멸자: 현재는 특별한 작업을 하지 않음

    void setData(int val) { data = val; } //데이터 값을 설정하는 함수
    void setLeft(BinaryNode* l) { left = l; } //왼쪽 자식 노드를 설정하는 함수
    void setRight(BinaryNode* r) { right = r; } //오른쪽 자식 노드를 설정하는 함수
    int getData() { return data; } //데이터 값을 반환하는 함수
    BinaryNode* getLeft() { return left; } //왼쪽 자식 노드를 반환하는 함수
    BinaryNode* getRight() { return right; } //오른쪽 자식 노드를 반환하는 함수
};
```

//이진 트리를 표현하는 클래스

```
class BinaryTree {  
    BinaryNode* root; //트리의 루트 노드
```

```
public:
```

//생성자: 루트 노드를 nullptr로 초기화

```
BinaryTree() : root(nullptr) {}  
~BinaryTree() {} //소멸자
```

void setRoot(BinaryNode\* node) { root = node; } //루트 노드 설정 함수

BinaryNode\* getRoot() { return root; } //루트 노드 반환 함수

bool isEmpty() { return root == nullptr; } //트리가 비어 있는지 확인하는 함수

//주어진 노드의 높이를 반환하는 함수

```
int getHeight(BinaryNode* node) {  
    if (node == nullptr) return 0;  
    int hLeft = getHeight(node->getLeft());  
    int hRight = getHeight(node->getRight());  
    return (hLeft > hRight) ? hLeft + 1 : hRight + 1;  
}
```

//트리가 균형 잡혀 있는지 확인하는 함수

```
bool isBalanced() { return isBalanced(root); }
```

// 주어진 노드를 루트로 하는 서브트리가 균형 잡혀 있는지 확인하는 함수

```
bool isBalanced(BinaryNode* node) {  
    if (node == nullptr) return true;  
    int leftHeight = getHeight(node->getLeft());  
    int rightHeight = getHeight(node->getRight());  
    if (abs(leftHeight - rightHeight) <= 1 &&  
        isBalanced(node->getLeft()) && isBalanced(node->getRight()))  
        return true;  
    return false;  
}
```

// 주어진 키를 삽입하는 함수

```
BinaryNode* insertNode(BinaryNode* node, int key) {  
    if (node == nullptr) return new BinaryNode(key);  
    if (key < node->getData()) node->setLeft(insertNode(node->getLeft(), key));  
    else node->setRight(insertNode(node->getRight(), key));  
    return node;  
}
```

## // 주어진 키를 삭제하는 함수

```
BinaryNode* deleteNode(BinaryNode* node, int key) {
    if (node == nullptr) return node;
    if (key < node->getData()) node->setLeft(deleteNode(node->getLeft(), key));
    else if (key > node->getData()) node->setRight(deleteNode(node->getRight(), key));
    else {
        // 자식 노드가 하나만 있거나 없는 경우
        if (node->getLeft() == nullptr)
            BinaryNode* temp = node->getRight();
            delete node;
            return temp;

        else if (node->getRight() == nullptr) {
            BinaryNode* temp = node->getLeft();
            delete node;
            return temp;
        }
        // 두 자식 노드가 모두 있는 경우
        BinaryNode* temp = findMin(node->getRight());
        node->setData(temp->getData());
        node->setRight(deleteNode(node->getRight(), temp->getData()));
    }
    return node;
}
```

// 주어진 서브트리에서 가장 작은 값을 가진 노드를 찾는 함수

```
BinaryNode* findMin(BinaryNode* node) {  
    while (node->getLeft() != nullptr) node = node->getLeft();  
    return node;  
}
```

// 키를 삽입하는 함수

```
void insert(int key) {  
    root = insertNode(root, key);  
}
```

// 키를 삭제하는 함수

```
void remove(int key) {  
    root = deleteNode(root, key);  
}  
};
```

```
int main() {
    int n;
    scanf("%d", &n); // 연산의 수 입력

    BinaryTree tree; // 이진 트리 객체 생성
    char operation;
    int value;

    // 주어진 연산을 처리하는 루프
    for (int i = 0; i < n; ++i) {
        scanf(" %c %d", &operation, &value);
        if (operation == 'I')
            tree.insert(value); // 삽입 연산
        else if (operation == 'D')
            tree.remove(value); // 삭제 연산
    }

    // 트리가 균형 잡혀 있는지 확인하고 결과 출력
    if (tree.isBalanced())
        printf("Balanced\n");
    else
        printf("Unbalanced\n");
    return 0;
}
```

합경로세기

02

## 합 경로 세기 - 문제

이진 탐색 트리와 숫자 X가 주어졌을 때, 루트 노드부터 시작하는 경로 (path)에서 연속된 노드들이 가지고 있는 숫자들의 합이 X가 되는 모든 경로의 수를 찾는 프로그램을 작성하시오.

### Input

입력의 첫 줄에는 하고자 하는 작업의 수  $n$  ( $1 \leq n \leq 20$ )와 숫자 X가 주어진다.

다음의  $n$  줄에는 다음과 같은 작업 명령이 들어온다.

| x : key x를 이진 탐색 트리에 삽입한다. (value는 key와 같다)

D x : key x를 이진 탐색 트리에서 삭제한다. (value는 key와 같다)

### Output

루트 노드에서 시작하는 경로중 노드가 가지는 숫자의 합이 X가 되는 경로의 개수를 찾아 출력한다.

# 합 경로 세기 알고리즘 (1)

## [BinaryNode 클래스]

<목적>

이진 트리의 각 노드를 정의합니다.

<멤버 변수 및 함수>

int data: 노드의 데이터 값

BinaryNode\* left, right: 왼쪽 및 오른쪽 자식 노드

생성자: BinaryNode(int val = 0, BinaryNode\* l = nullptr, BinaryNode\* r = nullptr)

setData(int val), getData(): 데이터 값 설정 및 반환

setLeft(BinaryNode\* l), getLeft(): 왼쪽 자식 노드 설정 및 반환

setRight(BinaryNode\* r), getRight(): 오른쪽 자식 노드 설정 및 반환

# 합 경로 세기 - 알고리즘 (2)

## [BinaryTree 클래스]

### <목적>

이진 트리를 관리하며 삽입, 삭제 및 특정 합을 가지는 경로의 수를 계산합니다.

### <멤버 변수 및 함수>

BinaryNode\* root: 트리의 루트 노드

insert(int key): 주어진 키를 가진 노드를 트리에 삽입합니다.

remove(int key): 주어진 키를 가진 노드를 트리에서 삭제합니다.

countPathsWithSum(int targetSum): 루트 노드부터 시작하여 targetSum을 가지는 경로의 수를 계산합니다.

main(): 사용자로부터 입력을 받아 트리에 값을 삽입하거나 삭제하고, 특정 합을 가지는 경로의 수를 출력합니다.

# 합 경로 세기 - 흐름도

1. 시작

2. 노드가 NULL인가?

예: 0 반환

아니오: 다음 단계로

3. 현재 노드 값 currentSum에 추가

4. currentSum - targetSum을 해시 맵에서 찾기

경로의 수에 더함

5. currentSum이 targetSum과 같은가?

예: 경로의 수를 증가

6. 해시 맵 업데이트 (currentSum 증가)

7. 왼쪽 자식에 대해 재귀 호출

8. 오른쪽 자식에 대해 재귀 호출

9. 해시 맵 업데이트 (currentSum 감소)

10. 총 경로 수 반환

11. 종료

```
class BinaryNode // BinaryNode 클래스 정의
{
protected:
    int data; // 노드의 데이터 값
    BinaryNode* left; // 왼쪽 자식 노드
    BinaryNode* right; // 오른쪽 자식 노드
public:
    // 생성자: 데이터 값과 왼쪽, 오른쪽 자식 노드를 초기화
    BinaryNode(int val = 0, BinaryNode* l = nullptr, BinaryNode* r = nullptr)
        : data(val), left(l), right(r) {}
    ~BinaryNode() {} // 소멸자: 현재는 특별한 작업을 하지 않음

    void setData(int val) { data = val; } // 데이터 값을 설정하는 함수
    void setLeft(BinaryNode* l) { left = l; } // 왼쪽 자식 노드를 설정하는 함수
    void setRight(BinaryNode* r) { right = r; } // 오른쪽 자식 노드를 설정하는 함수
    int getData() { return data; } // 데이터 값을 반환하는 함수
    BinaryNode* getLeft() { return left; } // 왼쪽 자식 노드를 반환하는 함수
    BinaryNode* getRight() { return right; } // 오른쪽 자식 노드를 반환하는 함수
};
```

```
class BinaryTree // BinaryTree 클래스 정의
```

```
{
```

```
    BinaryNode* root; // 트리의 루트 노드
```

```
public:
```

```
// 생성자: 루트 노드를 nullptr로 초기화
```

```
BinaryTree() : root(nullptr) {}
```

```
~BinaryTree() {} // 소멸자: 현재는 특별한 작업을 하지 않음
```

```
// 노드를 삽입하는 재귀 함수
```

```
BinaryNode* insertNode(BinaryNode* node, int key) {
```

```
    if (node == nullptr) return new BinaryNode(key); // 현재 노드가 nullptr이면 새로운 노드를 생성
```

```
// 삽입할 키가 현재 노드의 데이터보다 작으면 왼쪽 서브트리에 삽입
```

```
    if (key < node->getData()) node->setLeft(insertNode(node->getLeft(), key));
```

```
// 삽입할 키가 현재 노드의 데이터보다 크거나 같으면 오른쪽 서브트리에 삽입
```

```
    else node->setRight(insertNode(node->getRight(), key));
```

```
    return node; // 현재 노드를 반환
```

```
}
```

```
BinaryNode* deleteNode(BinaryNode* node, int key) // 노드를 삭제하는 재귀 함수
{
    if (node == nullptr) return node; // 현재 노드가 nullptr이면 그대로 반환

    // 삭제할 키가 현재 노드의 데이터보다 작으면 왼쪽 서브트리에서 삭제
    if (key < node->getData()) node->setLeft(deleteNode(node->getLeft(), key));

    // 삭제할 키가 현재 노드의 데이터보다 크면 오른쪽 서브트리에서 삭제
    else if (key > node->getData()) node->setRight(deleteNode(node->getRight(), key));

    else
    {
        // 현재 노드가 삭제할 노드인 경우
        // 왼쪽 자식이 없는 경우
        if (node->getLeft() == nullptr)
        {
            BinaryNode* temp = node->getRight(); // 오른쪽 자식을 임시 저장
            delete node; // 현재 노드를 삭제
            return temp; // 오른쪽 자식을 반환
        }
    }
}
```

```
else if (node->getRight() == nullptr) // 오른쪽 자식이 없는 경우
```

```
{
```

```
    BinaryNode* temp = node->getLeft(); // 왼쪽 자식을 임시 저장
```

```
    delete node; // 현재 노드를 삭제
```

```
    return temp; // 왼쪽 자식을 반환
```

```
}
```

```
// 양쪽 자식이 모두 있는 경우
```

```
BinaryNode* temp = findMin(node->getRight()); // 오른쪽 서브트리에서 최소값을 찾음
```

```
node->setData(temp->getData()); // 최소값을 현재 노드의 데이터로 설정
```

```
node->setRight(deleteNode(node->getRight(), temp->getData())); // 오른쪽 서브트리에서 최소값 노드를 삭제
```

```
} // else 끝
```

```
return node; // 현재 노드를 반환
```

```
} // deleteNode 끝
```

// 서브트리에서 최소값을 찾는 함수

```
BinaryNode* findMin(BinaryNode* node) {  
  
    while (node->getLeft() != nullptr) node = node->getLeft(); // 왼쪽 자식이 없을 때까지 반복  
    return node; // 최소값 노드를 반환  
}
```

// 삽입 연산을 위한 함수

```
void insert(int key) {  
    root = insertNode(root, key); // 루트 노드부터 시작하여 삽입  
}
```

// 삭제 연산을 위한 함수

```
void remove(int key) {  
    root = deleteNode(root, key); // 루트 노드부터 시작하여 삭제  
}
```

## // 특정 합을 가지는 경로의 수를 세는 재귀 함수

```
int countPathsWithSum(BinaryNode* node, int targetSum, int currentSum, std::unordered_map<int, int>& pathCount)
{
    if (node == nullptr) return 0; // 현재 노드가 nullptr이면 0을 반환
    currentSum += node->getData(); // 현재 노드의 데이터를 현재 합에 더함
    int sum = currentSum - targetSum; // 현재 합에서 목표 합을 뺀 값
    int totalPaths = pathCount[sum]; // 목표 합을 이루는 경로의 수

    if (currentSum == targetSum) // 현재 합이 목표 합과 같으면 경로의 수를 1 증가
        totalPaths++;

    pathCount[currentSum]++; // 현재 합의 경로 수를 증가
    // 왼쪽 서브트리에서 경로 수를 세고 더함
    totalPaths += countPathsWithSum(node->getLeft(), targetSum, currentSum, pathCount);
    // 오른쪽 서브트리에서 경로 수를 세고 더함
    totalPaths += countPathsWithSum(node->getRight(), targetSum, currentSum, pathCount);
    pathCount[currentSum]--; // 현재 합의 경로 수를 감소

    return totalPaths; // 총 경로 수를 반환
}
```

// 특정 합을 가지는 경로의 수를 세는 함수

```
int countPathsWithSum(int targetSum) {  
    std::unordered_map<int, int> pathCount; // 경로의 합을 저장하는 해시맵  
    return countPathsWithSum(root, targetSum, 0, pathCount); // 루트 노드부터 시작하여 경로의 수를 셈  
}  
}; // BinaryTree 클래스 끝
```

```
int main() // 메인 함수
{
    int n, X;
    scanf("%d %d", &n, &X); // 노드의 개수 n과 목표 합 X를 입력받음
    BinaryTree tree; // 이진 트리 생성
    char operation; // 작업을 나타내는 문자 (I: 삽입, D: 삭제)
    int value; // 노드의 값

    for (int i = 0; i < n; ++i) // n개의 작업을 입력받아 처리
    {
        scanf(" %c %d", &operation, &value);
        if (operation == 'I') // 삽입 작업인 경우
            tree.insert(value);
        else if (operation == 'D') // 삭제 작업인 경우
            tree.remove(value);
    }
    int result = tree.countPathsWithSum(X); // 목표 합을 가지는 경로의 수를 계산
    printf("%d\n", result); // 결과 출력
    return 0;
}
```

한영/영한 사전 검색

03

## 한영/ 영한 사전 검색 - 문제

[한국어 단어 검색과 영어 단어 검색이 가능한 한-영 사전과 영-한 사전을 이진탐색트리로 구현하시오.]

- 한영(K-E) 사전: 한국어 단어가 키(key)가 되고 영어 단어가 값(value)이 됨
- 영한(E-K) 사전: 영어 단어가 키(key)가 되고 한국어 단어가 값(value)이 됨

(단, 키가 중복되는 경우는 없다. 한국어와 영어단어에 공백이 포함될 수 있다.)

한영사전과 영한사전의 각 기능의 설명은 다음과 같다.

입력(i): <한국어 영어> 순서로 입력 받아서 2개의 사전트리에 삽입함.  
한국어가 한 줄(line)로 입력되고, 영어가 한줄로 입력됨

한국어 단어 검색(k): 검색할 한국어 단어를 입력받아 K-E 사전에서 해당 영어 번역어를 출력함.  
만약 사전에 없는 단어를 입력받았을 시 <한국어단어 UNKNOWN ENTRY>를 출력함

## 한영/ 영한 사전 검색 - 문제

영어 단어 검색(e): 검색할 영어 단어를 입력받아 E-K 사전에서 해당 한국어 번역어를 출력함.  
만약 사전에 없는 단어를 입력받았을시 <영어단어 UNKNOWN ENTRY>를 출력함

출력(p): K-E사전을 먼저 출력하고, E-K사전을 출력함. 이때, K-E사전은 한국어 단어(key) 순서로 출력,  
E-K사전은 영어 단어(key) 알파벳 순서로 출력함.  
각 사전의 시작부분에 “K-E dictionary:”와 “E-K dictionary:”을 출력하고 내용을 출력한다.

종료(q): 프로그램을 종료함

## 한영/ 영한 사전 검색 - 알고리즘

Record 클래스:

단어와 의미를 저장하고 관리하는 역할을 수행합니다.

set, compare, display, copy 등의 메서드를 통해 단어와 의미를 설정, 비교, 출력, 복사할 수 있습니다.

BinaryNode 클래스:

Record 클래스를 상속받아 이진 트리의 노드 역할을 합니다.

왼쪽 및 오른쪽 자식 노드를 관리하는 메서드 (setLeft, setRight, getLeft, getRight)와 노드가 리프 노드인지 확인하는 메서드 (isLeaf)를 포함합니다.

BinaryTree 클래스:

이진 트리를 나타내며 노드들을 관리합니다.

루트 노드 설정 및 반환, 트리가 비어 있는지 확인, 내림차순으로 중위 순회를 수행하는 메서드 (reverseInorder)를 포함합니다.

# 한영/ 영한 사전 검색 - 알고리즘

BinSrchTree 클래스:

BinaryTree 클래스를 상속받아 이진 검색 트리를 구현합니다.

노드 삽입 (insert), 삭제 (remove), 검색 (search) 등의 메서드를 포함합니다.

Dictionary 클래스:

한국어-영어 사전과 영어-한국어 사전을 관리합니다.

단어 및 의미 검색 (searchWord, searchMeaning), 삽입 (insert), 모든 단어 출력 (printAllWords) 등의 메서드를 포함합니다.

메인 함수:

사용자로부터 명령어를 입력받아 사전 관리 기능을 수행합니다.

# 한영/영한 사전 검색 - 흐름도

1. 시작

2. 명령어 입력

command에 명령어 저장

3. 명령어 검사 및 처리

삽입 명령어 (i)

word 입력

meaning 입력

dict.insert(word, meaning) 호출

4. 단어 검색 명령어 (k)

word 입력

dict.searchWord(word) 호출

5. 의미 검색 명령어 (e)

meaning 입력

dict.searchMeaning(meaning) 호출

6. 모든 단어 출력 명령어 (p)

dict.printAllWords() 호출

7. 종료 명령어 (q)

프로그램 종료

8. 루프 종료 (명령어가 q가 아니면 2로 돌아감)

9. 종료

// Record 클래스는 단어와 그 의미를 저장합니다.

class Record

{

protected:

char word[MAX\_WORD\_SIZE]; // 단어를 저장할 배열

char meaning[MAX\_MEANING\_SIZE]; // 의미를 저장할 배열

public:

// 생성자: 단어와 의미를 초기화합니다.

Record(const char\* word = "", const char\* meaning = "") { set(word, meaning); }

~Record() {} // 소멸자

// 단어와 의미를 설정하는 함수

void set(const char\* w, const char\* m)

{

strncpy(this->word, w, sizeof(this->word) - 1); // 단어를 배열에 복사

this->word[sizeof(this->word) - 1] = '\0'; // 널 문자를 넣어 문자열의 끝을 표시

strncpy(this->meaning, m, sizeof(this->meaning) - 1); // 의미를 배열에 복사

this->meaning[sizeof(this->meaning) - 1] = '\0'; // 널 문자를 넣어 문자열의 끝을 표시

}

int compare(Record\* n) { return compare(n->word); } // 다른 Record 객체와 단어를 비교하는 함수

int compare(const char\* w) { return strcmp(w, word); } // 단어와 문자열을 비교하는 함수

void display() { printf("%s %s\n", word, meaning); } // 단어와 의미를 출력하는 함수

void copy(Record\* n) { set(n->word, n->meaning); } // 다른 Record 객체의 단어와 의미를 복사하는 함수

};

// BinaryNode 클래스는 Record를 상속받고, 이진 트리의 노드 역할을 합니다.

```
class BinaryNode : public Record
```

```
{
```

```
protected:
```

```
    BinaryNode* left; // 왼쪽 자식 노드
```

```
    BinaryNode* right; // 오른쪽 자식 노드
```

```
public:
```

// 생성자: 단어와 의미, 자식 노드를 초기화합니다.

```
BinaryNode(const char* w = "", const char* m = "")
```

```
    : Record(w, m), left(nullptr), right(nullptr) {}
```

```
~BinaryNode() {} // 소멸자
```

void setLeft(BinaryNode\* l) { left = l; } // 왼쪽 자식 노드를 설정하는 함수

void setRight(BinaryNode\* r) { right = r; } // 오른쪽 자식 노드를 설정하는 함수

BinaryNode\* getLeft() { return left; } // 왼쪽 자식 노드를 반환하는 함수

BinaryNode\* getRight() { return right; } // 오른쪽 자식 노드를 반환하는 함수

// 노드가 리프 노드인지 확인하는 함수 (왼쪽과 오른쪽 자식 노드가 없는 경우)

```
bool isLeaf() { return left == nullptr && right == nullptr; }
```

```
};
```

// BinaryTree 클래스는 이진 트리를 나타내고, 노드들을 관리합니다.

```
class BinaryTree
```

```
{
```

```
protected:
```

```
    BinaryNode* root; // 루트 노드
```

```
public:
```

// 생성자: 루트 노드를 nullptr로 초기화합니다.

```
BinaryTree() : root(nullptr) {}
```

```
~BinaryTree() {} // 소멸자
```

void setRoot(BinaryNode\* node) { root = node; } // 루트 노드를 설정하는 함수

BinaryNode\* getRoot() { return root; } // 루트 노드를 반환하는 함수

bool isEmpty() { return root == nullptr; } // 트리가 비어 있는지 확인하는 함수

void reverseInorder() { reverseInorder(root); } // 내림차순으로 중위 순회를 수행하는 함수

```
void reverselnorder() { reverselnorder(root); } // 내림차순으로 중위 순회를 수행하는 함수
```

// 주어진 노드에서 내림차순 중위 순회를 수행하는 함수

```
void reverselnorder(BinaryNode* node)
{
    if (node != nullptr)
    {
        reverselnorder(node->getRight()); // 오른쪽 자식 노드로 이동
        node->display(); // 현재 노드를 출력
        reverselnorder(node->getLeft()); // 왼쪽 자식 노드로 이동
    }
}
```

};

// BinSrchTree 클래스는 이진 검색 트리로, BinaryTree를 상속받습니다.

class BinSrchTree : public BinaryTree

{

public:

BinSrchTree() {} // 생성자

~BinSrchTree() {} // 소멸자

// 주어진 키로 노드를 검색하는 함수

BinaryNode\* search(const char\* key)

{

    return search(root, key);

}

// 주어진 키로 노드를 검색하는 재귀 함수

BinaryNode\* search(BinaryNode\* node, const char\* key)

{

    if (node == nullptr) return nullptr; // 노드가 없으면 nullptr 반환

    int cmp = node->compare(key); // 노드의 단어와 키를 비교

    if (cmp == 0)

        return node; // 키와 일치하는 노드를 찾음

    else if (cmp > 0)

        return search(node->getLeft(), key); // 왼쪽 자식 노드에서 검색 계속

    else

        return search(node->getRight(), key); // 오른쪽 자식 노드에서 검색 계속

}

// 노드를 삽입하는 함수

```
bool insert(BinaryNode* node)
{
    if (node == nullptr) return false;
    if (isEmpty())
    {
        root = node; // 트리가 비어 있으면 루트 노드로 설정
        return true;
    }
    return insert(root, node);
}
```

// 주어진 루트 노드 아래에 노드를 삽입하는 재귀 함수

```
bool insert(BinaryNode* root, BinaryNode* node)
{
    int cmp = node->compare(root); // 노드의 단어와 루트 노드의 단어를 비교
    if (cmp == 0)
    {
        return false; // 중복된 키
    }
```

```
else if (cmp < 0)
{
    if (root->getLeft() == nullptr)
        root->setLeft(node); // 왼쪽 자식 노드가 없으면 설정
    return true;
else
    return insert(root->getLeft(), node); // 왼쪽 자식 노드로 이동하여 삽입 시도
}
else
{
    if (root->getRight() == nullptr)
        root->setRight(node); // 오른쪽 자식 노드가 없으면 설정
    return true;
else
    return insert(root->getRight(), node); // 오른쪽 자식 노드로 이동하여 삽입 시도
}
}
```

// 주어진 키로 노드를 삭제하는 함수

```
void remove(const char* key)
{
    if (isEmpty()) return;
    BinaryNode* parent = nullptr; // 부모 노드
    BinaryNode* node = root; // 현재 노드
    while (node != nullptr && node->compare(key) != 0)
    {
        parent = node; // 부모 노드를 현재 노드로 설정
        if (node->compare(key) > 0)
            node = node->getLeft(); // 왼쪽 자식 노드로 이동
        else
            node = node->getRight(); // 오른쪽 자식 노드로 이동
    }
    if (node == nullptr)
    {
        printf("Error: key not found in the tree!\n");
        return;
    }
    remove(parent, node); // 노드를 찾으면 삭제
}
```

```
// 주어진 부모 노드와 자식 노드를 이용해 노드를 삭제하는 함수
void remove(BinaryNode* parent, BinaryNode* node)
{
    if (node->getLeft() == nullptr || node->getRight() == nullptr)
    {
        BinaryNode* child = (node->getLeft() != nullptr) ? node->getLeft() : node->getRight(); // 하나의 자식 노드만 있는 경우
        if (node == root)
            root = child; // 루트 노드를 삭제할 경우
        else if (parent->getLeft() == node)
            parent->setLeft(child); // 부모 노드의 왼쪽 자식 노드로 설정
        else
            parent->setRight(child); // 부모 노드의 오른쪽 자식 노드로 설정
        delete node; // 노드를 삭제
    }
    else
    {
        BinaryNode* succParent = node; // 후계자 부모 노드
        BinaryNode* succ = node->getRight(); // 후계자 노드
        while (succ->getLeft() != nullptr)
        {
            succParent = succ; // 후계자 부모 노드를 설정
            succ = succ->getLeft(); // 후계자 노드의 왼쪽 자식 노드로 이동
        }
        node->copy(succ); // 후계자의 데이터를 현재 노드로 복사
        remove(succParent, succ); // 후계자를 삭제
    }
};
```

// Dictionary 클래스는 한국어-영어, 영어-한국어 사전을 관리합니다.

```
class Dictionary
{
protected:
    BinSrchTree KEDict; // 한국어-영어 사전
    BinSrchTree EKDict; // 영어-한국어 사전
public:
    Dictionary() {} // 생성자
    ~Dictionary() {} // 소멸자
```

// 모든 단어를 출력하는 함수

```
void printAllWords()
{
    std::cout << "K-E dictionary:\n";
    KEDict.reverseInorder(); // 한국어-영어 사전을 내림차순으로 출력
    std::cout << "E-K dictionary:\n";
    EKDict.reverseInorder(); // 영어-한국어 사전을 내림차순으로 출력
}
```

// 주어진 단어를 검색하는 함수

```
void searchWord(const char* word)
{
    if (strlen(word) == 0)
    {
        printf("Error: Empty word\n");
        return;
    }
    BinaryNode* node = KEDict.search(word); // 한국어-영어 사전에서 검색
    if (node != nullptr)
    {
        node->display();
    }
    else
    {
        printf("%s UNKNOWN ENTRY\n", word);
    }
}
```

// 주어진 의미를 검색하는 함수

```
void searchMeaning(const char* meaning)
{
    if (strlen(meaning) == 0)
    {
        printf("Error: Empty meaning\n");
        return;
    }
    BinaryNode* node = EKDict.search(meaning); // 영어-한국어 사전에서 검색
    if (node != nullptr)
    {
        node->display();
    }
    else
    {
        printf("%s UNKNOWN ENTRY\n", meaning);
    }
}
```

// 주어진 단어와 의미를 사전에 삽입하는 함수

```
void insert(const char* word, const char* meaning)
{
    if (strlen(word) == 0 || strlen(meaning) == 0)
    {
        printf("Error: Empty word or meaning\n");
        return;
    }
    if (strlen(word) >= MAX_WORD_SIZE || strlen(meaning) >= MAX_MEANING_SIZE)
    {
        printf("Error: Word or meaning exceeds maximum length\n");
        return;
    }
    BinaryNode* newKNode = new BinaryNode(word, meaning); // 한국어-영어 노드 생성
    if (!KEDict.insert(newKNode))
    {
        printf("Error: Duplicate word entry\n");
        delete newKNode;
        return;
    }
```

```
BinaryNode* newENode = new BinaryNode(meaning, word); //영어-한국어 노드 생성
if (!EKDict.insert(newENode))
{
    printf("Error: Duplicate meaning entry\n");
    delete newENode;
    return;
}
};

};
```

// 메인 함수는 사용자로부터 명령어를 입력받고, 해당 명령어를 처리합니다.

```
int main()
```

```
{
```

```
    char command; // 명령어를 저장할 변수
```

```
    std::string word, meaning; // 단어와 의미를 저장할 문자열
```

```
    Dictionary dict; // 사전 객체 생성
```

// 사용자가 'q' 명령어를 입력할 때까지 반복

```
do
```

```
{
```

```
    std::cin >> command; // 명령어 입력
```

```
    std::cin.ignore(); // Enter 키 무시
```

```
    switch (command)
```

```
{
```

```
    case 'i': // 삽입 명령어
```

```
        std::getline(std::cin, word); // 단어 입력
```

```
        std::getline(std::cin, meaning); // 의미 입력
```

```
        dict.insert(word.c_str(), meaning.c_str());
```

```
    break;
```

```
case 'k': // 단어 검색 명령어
    std::getline(std::cin, word);
    dict.searchWord(word.c_str());
    break;

case 'e': // 의미 검색 명령어
    std::getline(std::cin, meaning);
    dict.searchMeaning(meaning.c_str());
    break;

case 'p': // 모든 단어 출력 명령어
    dict.printAllWords();
    break;
}

} while (command != 'q'); // 종료 명령어 'q'가 입력될 때까지 반복

return 0;
}
```

**발표 들어주셔서 감사합니다.**