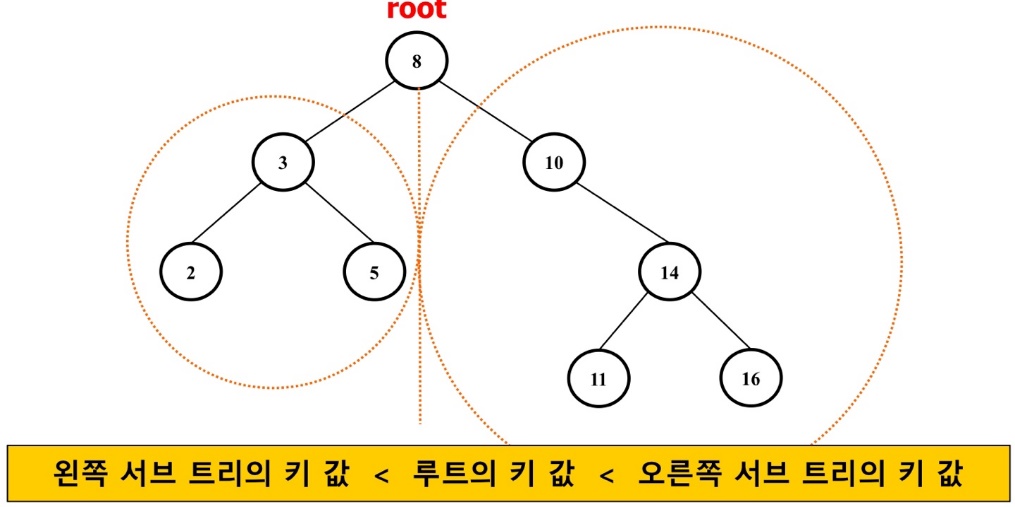
**08. 탐색트리( Search Tree)**

**08-1. 이진탐색 트리(Binary Search Tree)**

: 모든 노드는 서로 다른 키를 갖는다.(유일한 키 값)

**\* 각 노드는 최대 2개의 자식을 갖는다.**

****

ex) 같은 데이터와 다른 이진 탐색트리

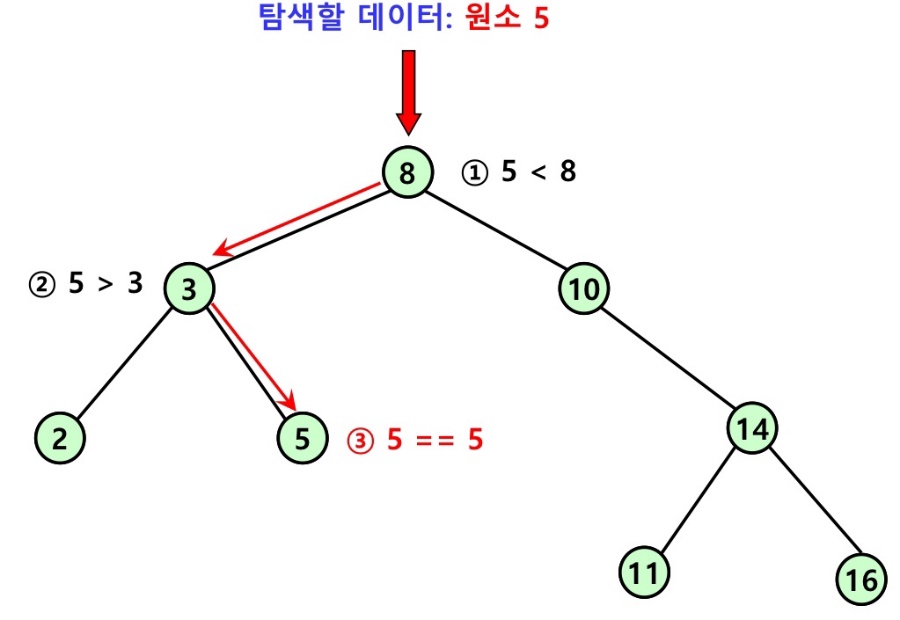
**스케치, 도표, 원, 그림이(가) 표시된 사진

자동 생성된 설명**

원, 도표, 스케치, 그림이(가) 표시된 사진

자동 생성된 설명

**# 탐색**

****

**# 삽입**

1. 삽입할 노드의 위치(부모 노드의 주소) 탐색

2. 노드삽입

텍스트, 도표, 라인, 스크린샷이(가) 표시된 사진

자동 생성된 설명

**# 삭제**

**1. 단말 노드**

도표, 라인이(가) 표시된 사진

자동 생성된 설명

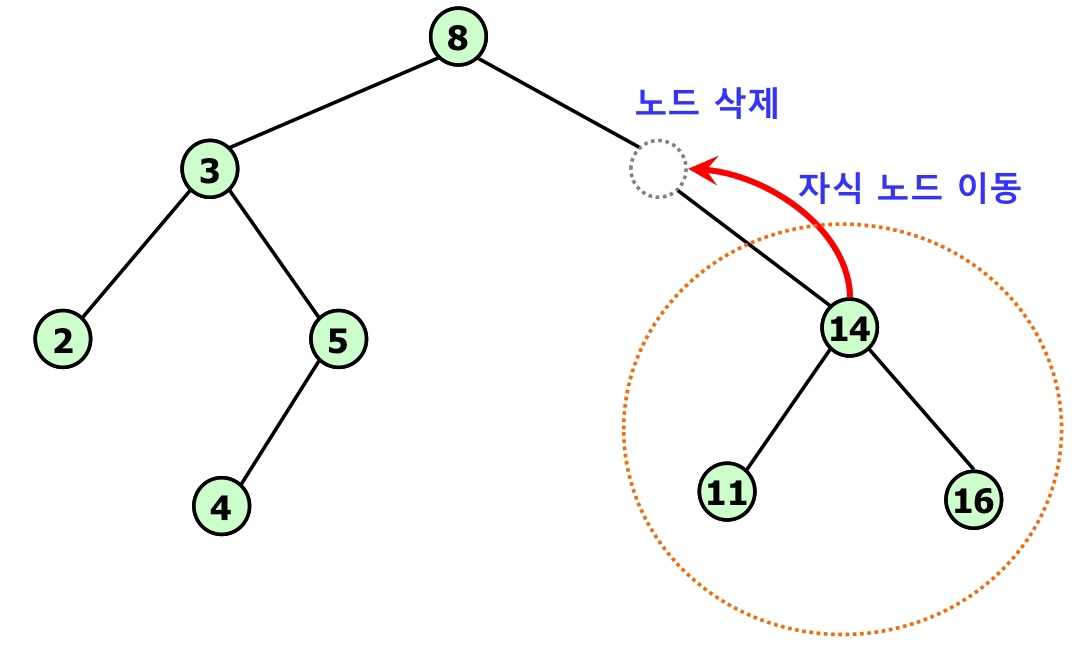
**2. 하나의 자식 노드만 존재**

**\* 삭제할 노드의 탐색**

**도표, 라인, 원, 스크린샷이(가) 표시된 사진

자동 생성된 설명**

**\* 삭제할 노드의 삭제 및 위치 조정**

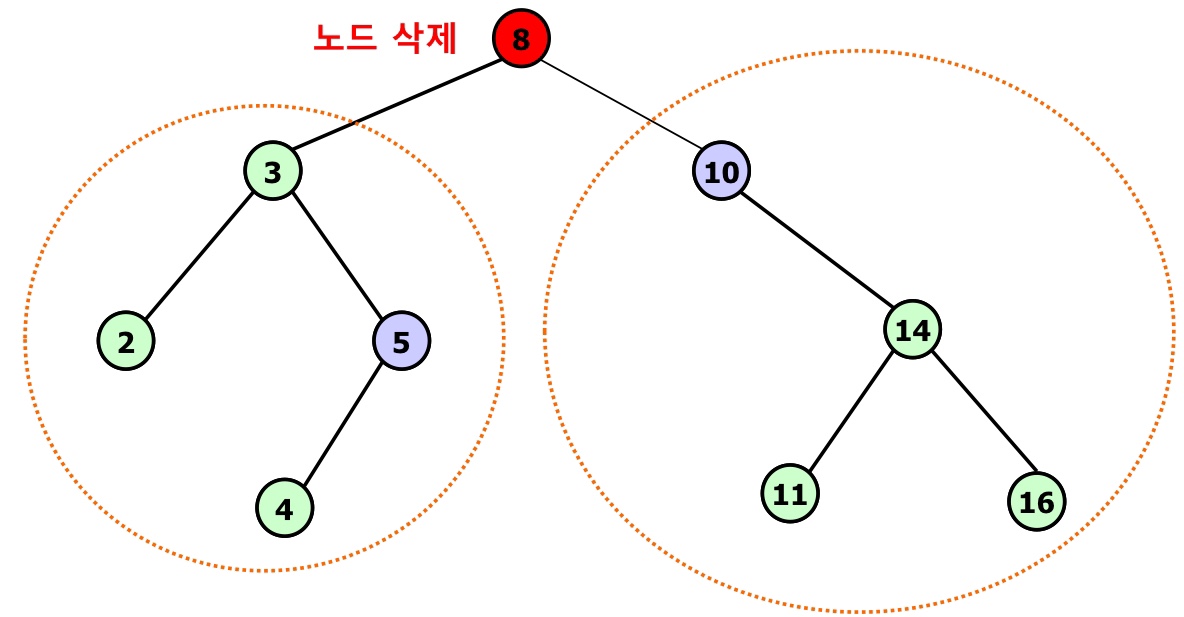
****

**3. 두개의 자식노드가 존재**

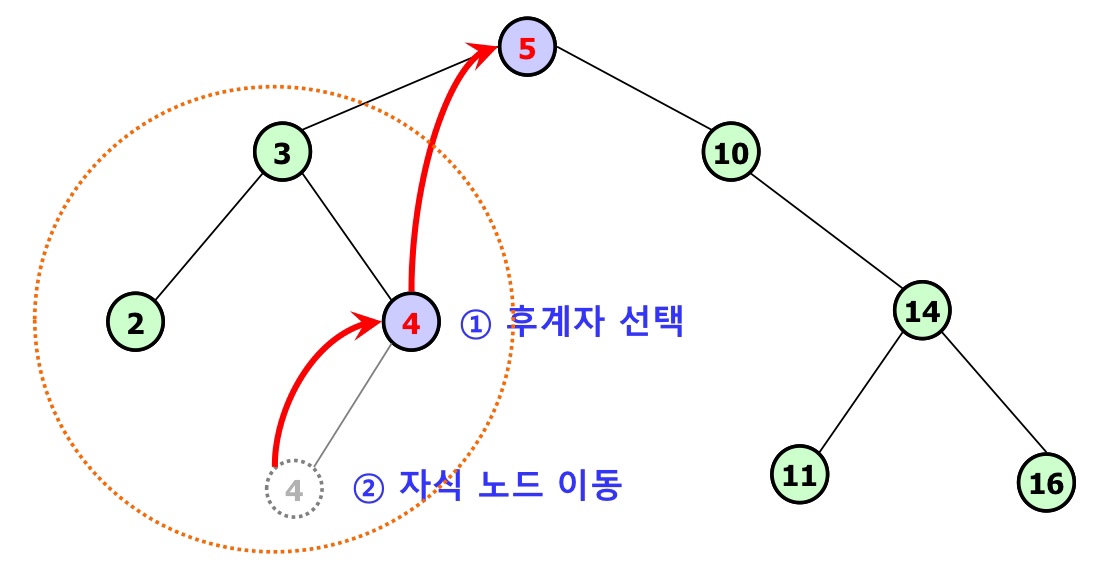
**\* 삭제할 노드의 탐색 및 후계자 노드 선정**

- 왼쪽 서브트리에서 가장 큰 키 값을 가진 노드

- 오른쪽 서브트리에서 가장 작은 키 값을 가진 노드



**\* 트리 재구성(데이터 5을 가진 노드를 후계자로 선택한 경우)**

****

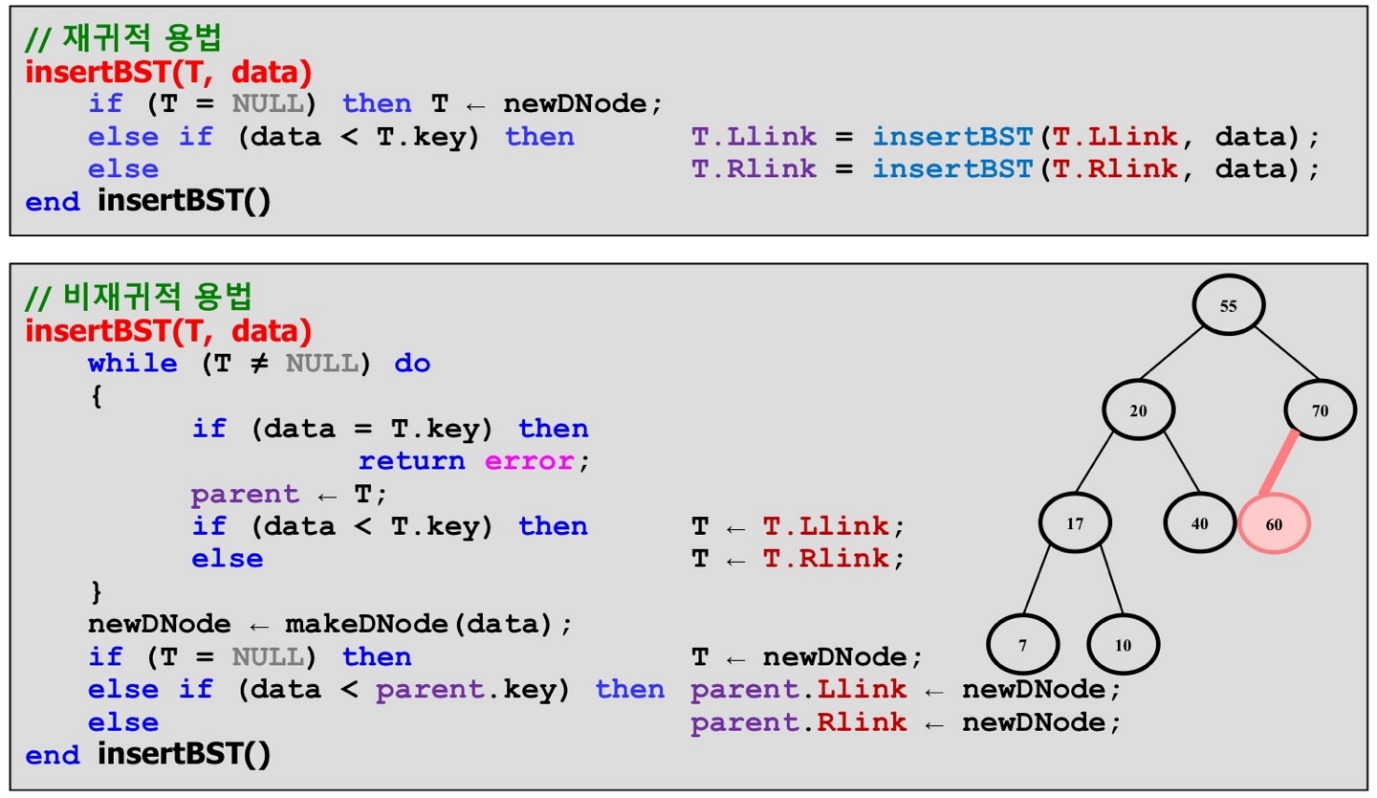
**# 알고리즘**

**탐색**

**텍스트, 스크린샷, 폰트, 도표이(가) 표시된 사진

자동 생성된 설명**

**삽입**

****

**삭제**

**텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명**

**# 알고리즘 구현**

**1. 재귀적용법(ver.C)**

#include <stdio.h>

#include <stdlib.h> // malloc

#include "BSTree.h" // DNode

// #include "LinkedDNode.h" // DNode

// 이진 검색 트리(BST): 데이터검색 -- 재귀적 용법

DNode\* searchBST(DNode\* root, element data) {

if (root == NULL || root->\_\_data == data)

return root;

DNode\* pTemp = NULL;

if (data < root->\_\_data) pTemp = searchBST(root->\_\_Llink, data);

else if (data > root->\_\_data) pTemp = searchBST(root->\_\_Rlink, data);

return pTemp;

}

// 이진 검색 트리(BST): 데이터 삽입 -- 재귀적 용법

DNode\* insertBST(DNode\* root, element data) {

DNode\* newNode = NULL;

if (root == NULL) {

newNode = makeDNode(data);

return newNode;

}

else if (data < root->\_\_data) {

root->\_\_Llink = insertBST(root->\_\_Llink, data);

return root;

}

else if (data > root->\_\_data) {

root->\_\_Rlink = insertBST(root->\_\_Rlink, data);

return root;

}

else {

printf("\n 이미 같은 키가 있습니다!!! \n");

return NULL;

}

return root;

}

// 이진 검색 트리(BST): 데이터 삭제 -- 비재귀적 용법

void deleteBST(DNode\* root, element key) {

// 삭제할 노드가 없는 경우

if (root == NULL) {

printf("\n 삭제할 키가 이진 트리에 없습니다!!");

return;

}

DNode\* pTemp = root, \* pParent = NULL, \* pChild;

// 삭제할 노드의 위치 탐색

while (pTemp->\_\_data != key) {

pParent = pTemp;

if (key < pTemp->\_\_data) pTemp = pTemp->\_\_Llink;

else pTemp = pTemp->\_\_Rlink;

}

// 1) 삭제할 노드가 단말 노드인 경우

if ((pTemp->\_\_Llink == NULL) && (pTemp->\_\_Rlink == NULL)) {

if (pParent != NULL) {

if (pParent->\_\_Llink == pTemp) pParent->\_\_Llink = NULL;

else pParent->\_\_Rlink = NULL;

}

else root = NULL;

}

// 2) 삭제할 노드가 자식 노드를 한 개 가진 경우

else if ((pTemp->\_\_Llink == NULL) || (pTemp->\_\_Rlink == NULL)) {

if (pTemp->\_\_Llink != NULL) pChild = pTemp->\_\_Llink;

else pChild = pTemp->\_\_Rlink;

if (pParent == NULL) root = pChild;

else {

if (pParent->\_\_Llink == pTemp) pParent->\_\_Llink = pChild;

else pParent->\_\_Rlink = pChild;

}

}

// 3) 삭제할 노드가 자식 노드를 두 개 가진 경우

else {

DNode\* succParent = pTemp;

DNode\* succ = pTemp->\_\_Llink;

// 왼쪽 서브 트리에서 후계자 찾기

while (succ->\_\_Rlink != NULL) {

succParent = succ;

succ = succ->\_\_Rlink;

}

if (succParent->\_\_Llink == succ)

succParent->\_\_Llink = succ->\_\_Llink;

else succParent->\_\_Rlink = succ->\_\_Llink;

pTemp->\_\_data = succ->\_\_data;

pTemp = succ;

}

free(pTemp);

}

**2. 재귀적용법(ver.CPP)**

#include <iostream>

#include <queue>

#include <algorithm> // max

using namespace std;

template <typename E>

class LinkedBSTree;

// class DNode

template <typename E>

class DNode {

public:

DNode(const E& data);

E getData(void) const;

DNode<E>\* getLlink(void) const;

DNode<E>\* getRlink(void) const;

private:

E \_\_data;

DNode<E>\* \_\_Llink;

DNode<E>\* \_\_Rlink;

template <typename T> friend class LinkedBSTree;

};

// DNode: 멤버 함수 정의

template <typename E>

DNode<E>::DNode(const E& data) :

\_\_data(data), \_\_Llink(nullptr), \_\_Rlink(nullptr) {}

template <typename E>

E DNode<E>::getData(void) const { return \_\_data; }

template <typename E>

DNode<E>\* DNode<E>::getLlink(void) const { return \_\_Llink; }

template <typename E>

DNode<E>\* DNode<E>::getRlink(void) const { return \_\_Rlink; }

// class LinkedBSTree

template <typename E>

class LinkedBSTree {

public:

LinkedBSTree();

~LinkedBSTree();

DNode<E>\* getRoot(void) const;

void setRoot(DNode<E>\* root);

DNode<E>\* insertBST(DNode<E>\* root, const E& data) const;

DNode<E>\* deleteBST(DNode<E>\* root, const E& data) const;

DNode<E>\* searchBST(DNode<E>\* root, const E& data) const;

void Preorder(DNode<E>\* root) const;

private:

DNode<E>\* \_\_root;

};

// LinkedBSTree: 멤버 함수 정의

template <typename E>

LinkedBSTree<E>::LinkedBSTree() :

\_\_root(nullptr) {}

// 소멸자: 모든 노드 삭제

// 너비 우선 순회: 큐 구조 활용

template <typename E>

LinkedBSTree<E>::~LinkedBSTree() {

if (\_\_root == nullptr)

return;

DNode<E>\* tNode = nullptr;

queue<DNode<E>\*> Q;

Q.push(\_\_root);

while (!Q.empty()) {

tNode = Q.front();

Q.pop();

if (tNode->\_\_Llink) Q.push(tNode->\_\_Llink);

if (tNode->\_\_Rlink) Q.push(tNode->\_\_Rlink);

delete tNode;

}

}

template <typename E>

void LinkedBSTree<E>::setRoot(DNode<E>\* root) {

\_\_root = root;

}

template <typename E>

DNode<E>\* LinkedBSTree<E>::getRoot(void) const {

return \_\_root;

}

// 깊이 우선 순회: 전위 순회(재귀적 용법)

template <typename E>

void LinkedBSTree<E>::Preorder(DNode<E>\* root) const {

if (root) {

cout.width(3); cout << root->\_\_data;

Preorder(root->\_\_Llink);

Preorder(root->\_\_Rlink);

}

}

// 이진 검색 트리(BST): 데이터 검색 -- 재귀적 용법

template <typename E>

DNode<E>\* LinkedBSTree<E>::searchBST(DNode<E>\* root, const E& data) const {

if (root == nullptr || data == root->\_\_data)

return root;

DNode<E>\* tNode = nullptr;

if (data < root->\_\_data) tNode = searchBST(root->\_\_Llink, data);

else if (data > root->\_\_data) tNode = searchBST(root->\_\_Rlink, data);

return tNode;

}

// 이진 검색 트리(BST): 데이터 삽입 -- 재귀적 용법

template <typename E>

DNode<E>\* LinkedBSTree<E>::insertBST(DNode<E>\* root, const E& data) const {

// 탐색 실패: 새로운 노드 생성

DNode<E>\* newNode = nullptr;

if (root == nullptr) {

newNode = new DNode<E>(data);

return newNode;

}

// 중복된 데이터가 존재할 경우 데이터 삽입 불가능!!!

if (data == root->\_\_data) cout << "이미 같은 키가 있습니다!!!" << endl;

else if (data < root->\_\_data) root->\_\_Llink = insertBST(root->\_\_Llink, data);

else if (data > root->\_\_data) root->\_\_Rlink = insertBST(root->\_\_Rlink, data);

return root;

}

// 이진 검색 트리(BST): 데이터 삭제 -- 비재귀적 용법

template <typename E>

DNode<E>\* LinkedBSTree<E>::deleteBST(DNode<E>\* root, const E& data) const {

// 삭제할 노드가 없는 경우

if (root == nullptr) {

cout << "\n키를 찾지 못했습니다." << endl;

return root;

}

DNode<E>\* pParent = nullptr; // 삭제할 노드의 부모 노드

DNode<E>\* tNode = root; // 삭제할 노드

DNode<E>\* pChild; // 삭제할 노드의 자식 노드

// 삭제할 노드의 위치 탐색

while (data != tNode->\_\_data) {

pParent = tNode;

if (data < tNode->\_\_data) tNode = tNode->\_\_Llink;

else tNode = tNode->\_\_Rlink;

}

// 1) 삭제할 노드가 단말 노드인 경우

if ((tNode->\_\_Llink == nullptr) && (tNode->\_\_Rlink == nullptr)) {

if (pParent == nullptr) root = nullptr;

else if (pParent != nullptr) {

if (pParent->\_\_Llink == tNode) pParent->\_\_Llink = nullptr;

else pParent->\_\_Rlink = nullptr;

}

}

// 2) 삭제할 노드가 자식 노드를 한 개 가진 경우

else if ((tNode->\_\_Llink == nullptr) || (tNode->\_\_Rlink == nullptr)) {

// pChild : (후계자) 삭제할 노드의 자식 노드

if (tNode->\_\_Llink != nullptr) pChild = tNode->\_\_Llink;

else pChild = tNode->\_\_Rlink;

if (pParent == nullptr) root = pChild;

else {

if (pParent->\_\_Llink == tNode) pParent->\_\_Llink = pChild;

else pParent->\_\_Rlink = pChild;

}

}

// 3) 삭제할 노드가 자식 노드를 두 개 가진 경우

else {

// 왼쪽 서브 트리에서 후계자(최댓값) 찾기

DNode<E>\* sParent = tNode;

DNode<E>\* sNode = tNode->\_\_Llink;

while (sNode->\_\_Rlink != nullptr) {

sParent = sNode;

sNode = sNode->\_\_Rlink;

}

if (sParent->\_\_Llink == sNode)

sParent->\_\_Llink = sNode->\_\_Llink;

else

sParent->\_\_Rlink = sNode->\_\_Llink;

tNode->\_\_data = sNode->\_\_data;

tNode = sNode;

}

delete tNode;

return root;

}

// 전체 원소 출력, 데이터 검색, 삽입, 삭제

template <typename E> void printBSTAll(LinkedBSTree<E>& bst);

template <typename E> void SearchBST(LinkedBSTree<E>& bst);

template <typename E> void InsertBST(LinkedBSTree<E>& bst);

template <typename E> void DeleteBST(LinkedBSTree<E>& bst);

int main(void)

{

int num;

LinkedBSTree<int> bst;

while (true) {

system("cls");

cout << "\n ##### 이진 검색 트리 ##### \n" << endl;

cout << "1) 데이터 삽입" << endl;

cout << "2) 데이터 삭제" << endl;

cout << "3) 데이터 검색" << endl;

cout << "4) 전체 원소 출력" << endl;

cout << "5) 프로그램 종료" << endl;

cout << "\n메뉴 선택: ";

cin >> num;

switch (num) {

case 1: InsertBST(bst); break;

case 2: DeleteBST(bst); break;

case 3: SearchBST(bst); break;

case 4: printBSTAll(bst); break;

case 5: cout << "프로그램 종료!!!\n" << endl;

return 0;

default: cout << "메뉴를 잘못 선택하셨습니다." << endl;

}

system("pause");

}

return 0;

}

// 이진 검색 트리(BST): 전체 노드 출력(전위 순회)

template <typename E>

void printBSTAll(LinkedBSTree<E>& bst) {

cout << "\n이진 검색 트리: 전체 원소 출력" << endl;

bst.Preorder(bst.getRoot());

cout << "\n" << endl;

}

// 이진 검색 트리(BST): 데이터 검색

template <typename E>

void SearchBST(LinkedBSTree<E>& bst) {

int num;

DNode<E>\* tNode = nullptr;

cout << "\n이진 검색 트리: 데이터 검색" << endl;

while (true) {

cout << "임의의 정수 입력(종료: 0): ";

cin >> num;

if (num == 0)

break;

// 데이터 검색

tNode = bst.searchBST(bst.getRoot(), num);

if (tNode) cout << tNode->getData() << " 키를 찾았습니다!!!" << endl;

else cout << "키를 찾지 못했습니다." << endl;

}

}

// 이진 검색 트리(BST): 데이터 삽입

template <typename E>

void InsertBST(LinkedBSTree<E>& bst) {

int num;

DNode<E>\* root = nullptr;

cout << "\n이진 검색 트리: 데이터 삽입" << endl;

while (true) {

cout << "임의의 정수 입력(종료: 0): ";

cin >> num;

if (num == 0)

break;

// 데이터 삽입

root = bst.insertBST(bst.getRoot(), num);

if (root != bst.getRoot())

bst.setRoot(root);

}

}

// 이진 검색 트리(BST): 데이터 삭제

template <typename E>

void DeleteBST(LinkedBSTree<E>& bst) {

int num;

DNode<E>\* root = nullptr;

cout << "\n이진 검색 트리: 데이터 삭제" << endl;

while (true) {

cout << "임의의 정수 입력(종료: 0): ";

cin >> num;

if (num == 0)

break;

// 데이터 삭제

root = bst.deleteBST(bst.getRoot(), num);

if (root != bst.getRoot())

bst.setRoot(root);

}

}

**3. 비재귀적용법(ver.CPP)**

#include <iostream>

#include <queue>

#include <algorithm> // max

using namespace std;

template <typename E>

class LinkedBSTree;

// class DNode

template <typename E>

class DNode {

public:

DNode(const E& data);

E getData(void) const;

DNode<E>\* getLlink(void) const;

DNode<E>\* getRlink(void) const;

private:

E \_\_data;

DNode<E>\* \_\_Llink;

DNode<E>\* \_\_Rlink;

template <typename T> friend class LinkedBSTree;

};

// DNode: 멤버 함수 정의

template <typename E>

DNode<E>::DNode(const E& data) :

\_\_data(data), \_\_Llink(nullptr), \_\_Rlink(nullptr) {}

template <typename E>

E DNode<E>::getData(void) const { return \_\_data; }

template <typename E>

DNode<E>\* DNode<E>::getLlink(void) const { return \_\_Llink; }

template <typename E>

DNode<E>\* DNode<E>::getRlink(void) const { return \_\_Rlink; }

// class LinkedBSTree

template <typename E>

class LinkedBSTree {

public:

LinkedBSTree();

~LinkedBSTree();

DNode<E>\* getRoot(void) const;

void setRoot(DNode<E>\* root);

DNode<E>\* insertBST(DNode<E>\* root, const E& data) const;

DNode<E>\* deleteBST(DNode<E>\* root, const E& data) const;

DNode<E>\* searchBST(DNode<E>\* root, const E& data) const;

void Preorder(DNode<E>\* root) const;

private:

DNode<E>\* \_\_root;

};

// LinkedBSTree: 멤버 함수 정의

template <typename E>

LinkedBSTree<E>::LinkedBSTree() :

\_\_root(nullptr) {}

// 소멸자: 모든 노드 삭제

// 너비 우선 순회: 큐 구조 활용

template <typename E>

LinkedBSTree<E>::~LinkedBSTree() {

if (\_\_root == nullptr)

return;

DNode<E>\* tNode = nullptr;

queue<DNode<E>\*> Q;

Q.push(\_\_root);

while (!Q.empty()) {

tNode = Q.front();

Q.pop();

if (tNode->\_\_Llink) Q.push(tNode->\_\_Llink);

if (tNode->\_\_Rlink) Q.push(tNode->\_\_Rlink);

delete tNode;

}

}

template <typename E>

void LinkedBSTree<E>::setRoot(DNode<E>\* root) {

\_\_root = root;

}

template <typename E>

DNode<E>\* LinkedBSTree<E>::getRoot(void) const {

return \_\_root;

}

// 깊이 우선 순회: 전위 순회(재귀적 용법)

template <typename E>

void LinkedBSTree<E>::Preorder(DNode<E>\* root) const {

if (root) {

cout.width(3); cout << root->\_\_data;

Preorder(root->\_\_Llink);

Preorder(root->\_\_Rlink);

}

}

// 이진 검색 트리(BST): 데이터 검색 -- 재귀적 용법

template <typename E>

DNode<E>\* LinkedBSTree<E>::searchBST(DNode<E>\* root, const E& data) const {

DNode<E>\* tNode = root;

while (tNode) {

if (data == tNode->\_\_data) return tNode;

else if (data < root->\_\_data) tNode = tNode->\_\_Llink;

else if (data > root->\_\_data) tNode = tNode->\_\_Llink;

}

return nullptr;

}

// 이진 검색 트리(BST): 데이터 삽입 -- 재귀적 용법

template <typename E>

DNode<E>\* LinkedBSTree<E>::insertBST(DNode<E>\* root, const E& data) const {

//삽입할 노드의 부모 노드 검색

DNode<E>\* pParent = nullptr;

DNode<E>\* tNode = root;

while (tNode) {

pParent = tNode;

if (data == tNode->\_\_data) {

cout << "이미 같은 키가 있습니다!!" << endl;

return root;

}

else if (data < tNode->\_\_data) tNode = tNode->\_\_Llink;

else if (data > root->\_\_data) tNode = tNode->\_\_Rlink;

}

//탐색 실패: 새로운 노드 생성

if (pParent == nullptr) root = new DNode<int>(data);

else if (data < pParent->\_\_data) pParent->\_\_Llink = new DNode<int>(data);

else if (data > pParent->\_\_data) pParent->\_\_Rlink = new DNode<int>(data);

return root;

}

// 이진 검색 트리(BST): 데이터 삭제 -- 비재귀적 용법

template <typename E>

DNode<E>\* LinkedBSTree<E>::deleteBST(DNode<E>\* root, const E& data) const {

// 삭제할 노드가 없는 경우

if (root == nullptr) {

cout << "\n키를 찾지 못했습니다." << endl;

return root;

}

DNode<E>\* pParent = nullptr; // 삭제할 노드의 부모 노드

DNode<E>\* tNode = root; // 삭제할 노드

DNode<E>\* pChild; // 삭제할 노드의 자식 노드

// 삭제할 노드의 위치 탐색

while (data != tNode->\_\_data) {

pParent = tNode;

if (data < tNode->\_\_data) tNode = tNode->\_\_Llink;

else tNode = tNode->\_\_Rlink;

}

// 1) 삭제할 노드가 단말 노드인 경우

if ((tNode->\_\_Llink == nullptr) && (tNode->\_\_Rlink == nullptr)) {

if (pParent == nullptr) root = nullptr;

else if (pParent != nullptr) {

if (pParent->\_\_Llink == tNode) pParent->\_\_Llink = nullptr;

else pParent->\_\_Rlink = nullptr;

}

}

// 2) 삭제할 노드가 자식 노드를 한 개 가진 경우

else if ((tNode->\_\_Llink == nullptr) || (tNode->\_\_Rlink == nullptr)) {

// pChild : (후계자) 삭제할 노드의 자식 노드

if (tNode->\_\_Llink != nullptr) pChild = tNode->\_\_Llink;

else pChild = tNode->\_\_Rlink;

if (pParent == nullptr) root = pChild;

else {

if (pParent->\_\_Llink == tNode) pParent->\_\_Llink = pChild;

else pParent->\_\_Rlink = pChild;

}

}

// 3) 삭제할 노드가 자식 노드를 두 개 가진 경우

else {

// 왼쪽 서브 트리에서 후계자(최댓값) 찾기

DNode<E>\* sParent = tNode;

DNode<E>\* sNode = tNode->\_\_Llink;

while (sNode->\_\_Rlink != nullptr) {

sParent = sNode;

sNode = sNode->\_\_Rlink;

}

if (sParent->\_\_Llink == sNode)

sParent->\_\_Llink = sNode->\_\_Llink;

else

sParent->\_\_Rlink = sNode->\_\_Llink;

tNode->\_\_data = sNode->\_\_data;

tNode = sNode;

}

delete tNode;

return root;

}

// 전체 원소 출력, 데이터 검색, 삽입, 삭제

template <typename E> void printBSTAll(LinkedBSTree<E>& bst);

template <typename E> void SearchBST(LinkedBSTree<E>& bst);

template <typename E> void InsertBST(LinkedBSTree<E>& bst);

template <typename E> void DeleteBST(LinkedBSTree<E>& bst);

int main(void)

{

int num;

LinkedBSTree<int> bst;

while (true) {

system("cls");

cout << "\n ##### 이진 검색 트리 ##### \n" << endl;

cout << "1) 데이터 삽입" << endl;

cout << "2) 데이터 삭제" << endl;

cout << "3) 데이터 검색" << endl;

cout << "4) 전체 원소 출력" << endl;

cout << "5) 프로그램 종료" << endl;

cout << "\n메뉴 선택: ";

cin >> num;

switch (num) {

case 1: InsertBST(bst); break;

case 2: DeleteBST(bst); break;

case 3: SearchBST(bst); break;

case 4: printBSTAll(bst); break;

case 5: cout << "프로그램 종료!!!\n" << endl;

return 0;

default: cout << "메뉴를 잘못 선택하셨습니다." << endl;

}

system("pause");

}

return 0;

}

// 이진 검색 트리(BST): 전체 노드 출력(전위 순회)

template <typename E>

void printBSTAll(LinkedBSTree<E>& bst) {

cout << "\n이진 검색 트리: 전체 원소 출력" << endl;

bst.Preorder(bst.getRoot());

cout << "\n" << endl;

}

// 이진 검색 트리(BST): 데이터 검색

template <typename E>

void SearchBST(LinkedBSTree<E>& bst) {

int num;

DNode<E>\* tNode = nullptr;

cout << "\n이진 검색 트리: 데이터 검색" << endl;

while (true) {

cout << "임의의 정수 입력(종료: 0): ";

cin >> num;

if (num == 0)

break;

// 데이터 검색

tNode = bst.searchBST(bst.getRoot(), num);

if (tNode) cout << tNode->getData() << " 키를 찾았습니다!!!" << endl;

else cout << "키를 찾지 못했습니다." << endl;

}

}

// 이진 검색 트리(BST): 데이터 삽입

template <typename E>

void InsertBST(LinkedBSTree<E>& bst) {

int num;

DNode<E>\* root = nullptr;

cout << "\n이진 검색 트리: 데이터 삽입" << endl;

while (true) {

cout << "임의의 정수 입력(종료: 0): ";

cin >> num;

if (num == 0)

break;

// 데이터 삽입

root = bst.insertBST(bst.getRoot(), num);

if (root != bst.getRoot())

bst.setRoot(root);

}

}

// 이진 검색 트리(BST): 데이터 삭제

template <typename E>

void DeleteBST(LinkedBSTree<E>& bst) {

int num;

DNode<E>\* root = nullptr;

cout << "\n이진 검색 트리: 데이터 삭제" << endl;

while (true) {

cout << "임의의 정수 입력(종료: 0): ";

cin >> num;

if (num == 0)

break;

// 데이터 삭제

root = bst.deleteBST(bst.getRoot(), num);

if (root != bst.getRoot())

bst.setRoot(root);

}

}

**08-2. 균형탐색 트리(Binary Search Tree)**

**텍스트, 스크린샷, 폰트, 도표이(가) 표시된 사진

자동 생성된 설명**

**08-2-1. AVL 트리**

: 전체 트리의 구조가 균형이 맞도록 하는 트리

* **모든 노드에 대해 좌 서브 트리의 높이(깊이)와 우 서브 트리의 높이의 차가 1을 넘지 않는다.(즉, 트리 구조가 한쪽으로 쏠리는 것을 막을 수 있다).**

삼각형, 디자인이(가) 표시된 사진

자동 생성된 설명

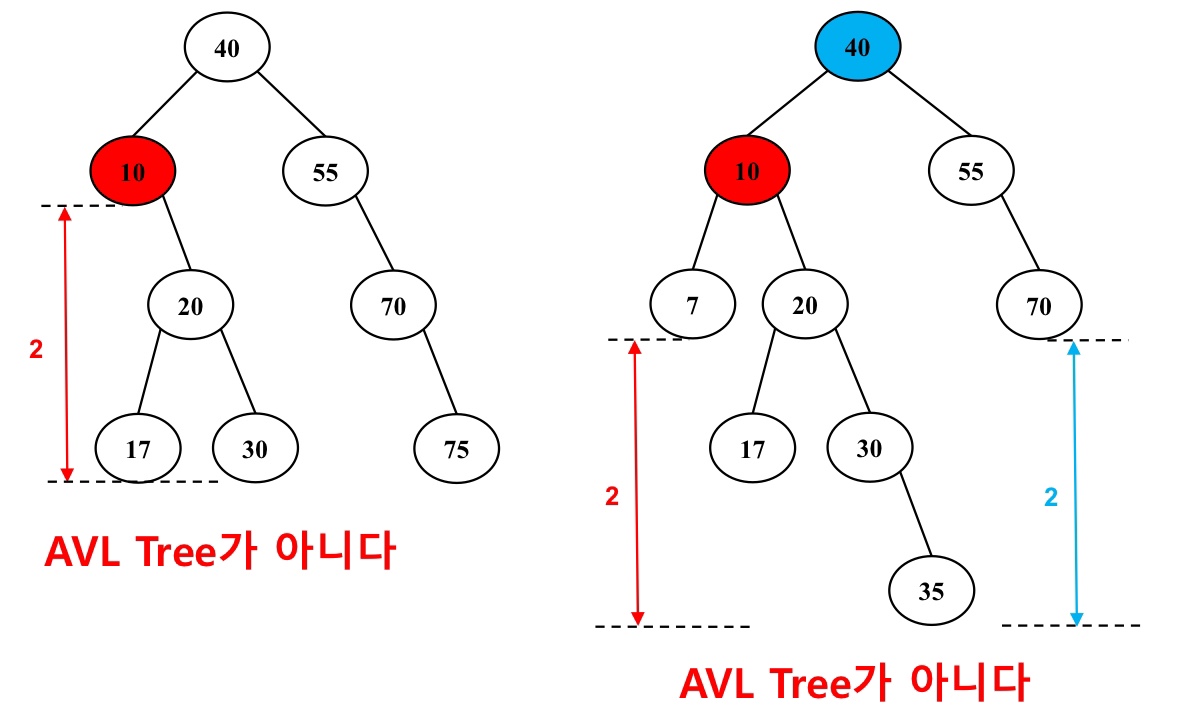
ex) AVL트리

스케치, 라인 아트, 도표, 화이트이(가) 표시된 사진

자동 생성된 설명

ex) AVL트리가 아닌 예

스케치, 그림, 도표, 라인 아트이(가) 표시된 사진

자동 생성된 설명

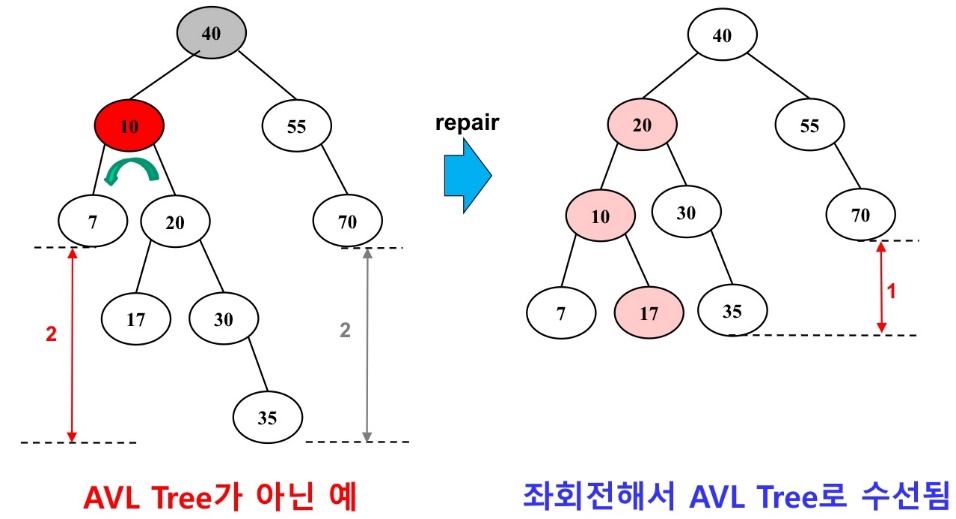
**# 균형 맞추기 : 좌회전**

1. **좌회전으로 불균형 해결**

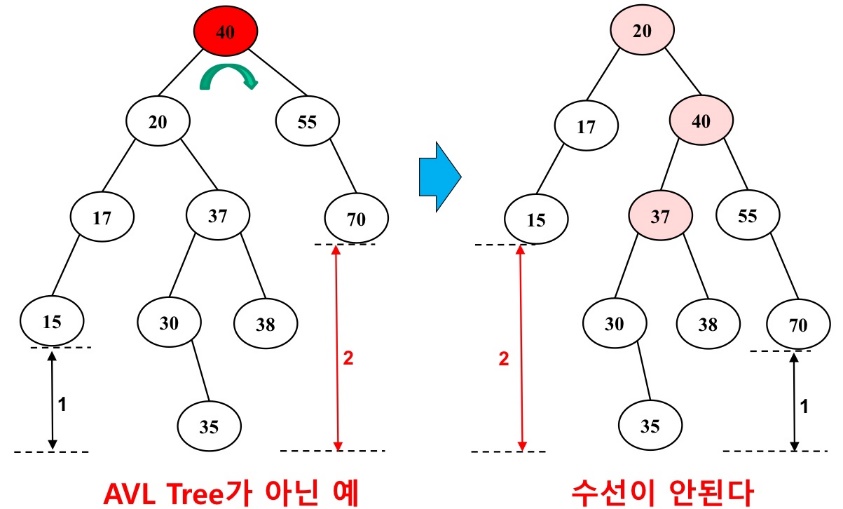
**도표, 원이(가) 표시된 사진

자동 생성된 설명**

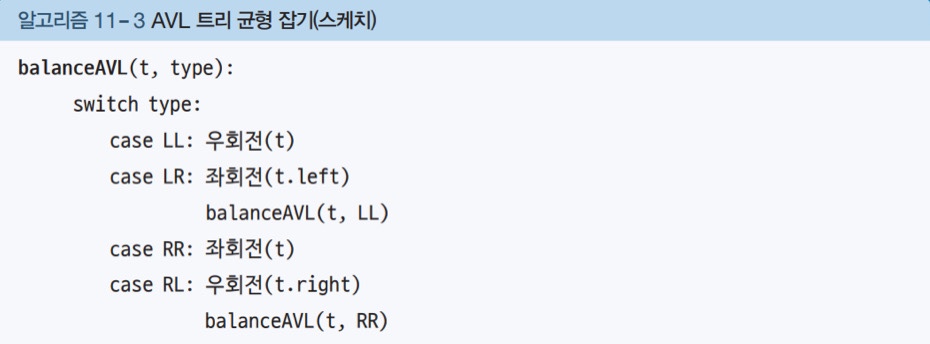
1. **좌회전으로 두 곳의 불균형 해결**

****

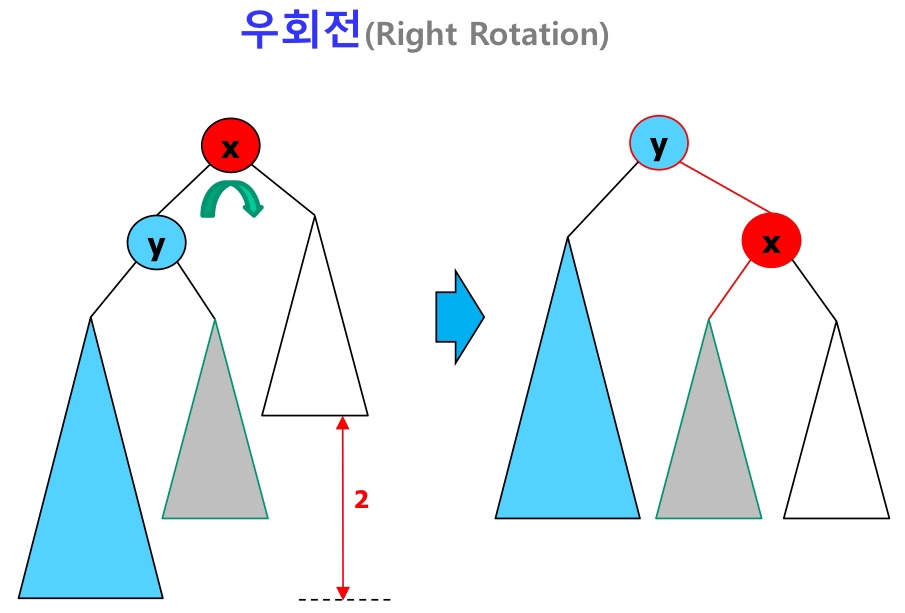
1. **단순한 회전으로 해결이 안되는 경우**

****

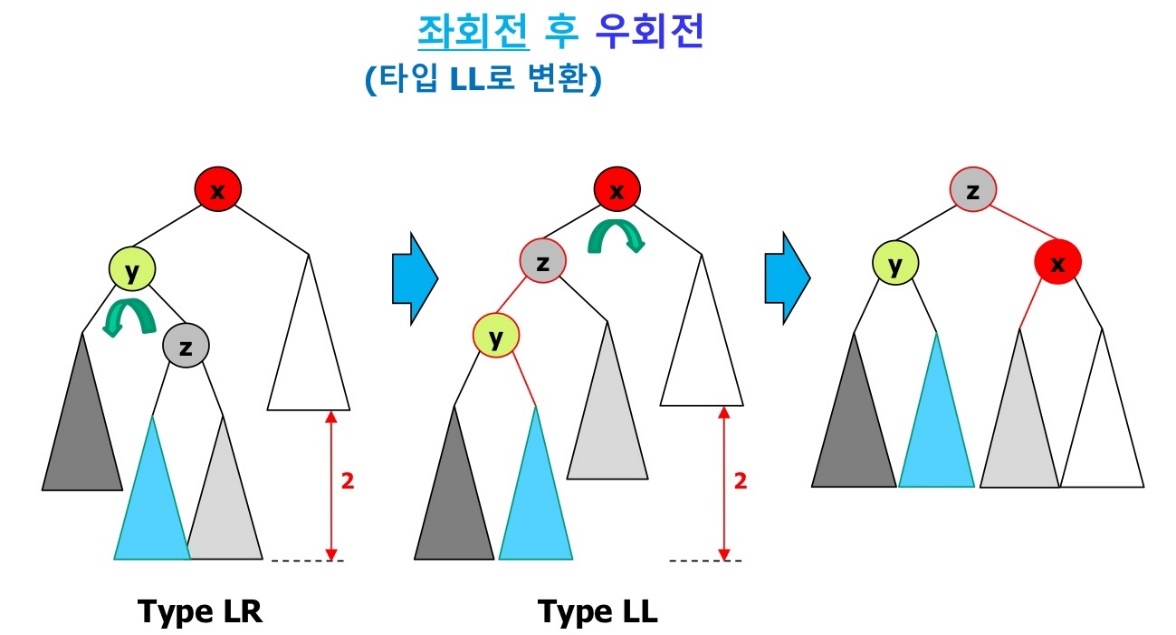
**# 4가지 유형(LL, LR, RR, RL)**

****

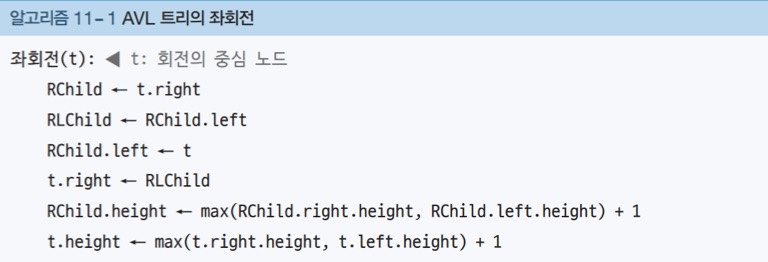
**1. LL : T.left.left가 가장 깊음** 🡪 **우회전(Right Rotation)**

****

**2. LR : T.left.right가 가장 깊음** 🡪 **좌회전(left Rotation)**🡪 **LL**🡪 **우회전**

****

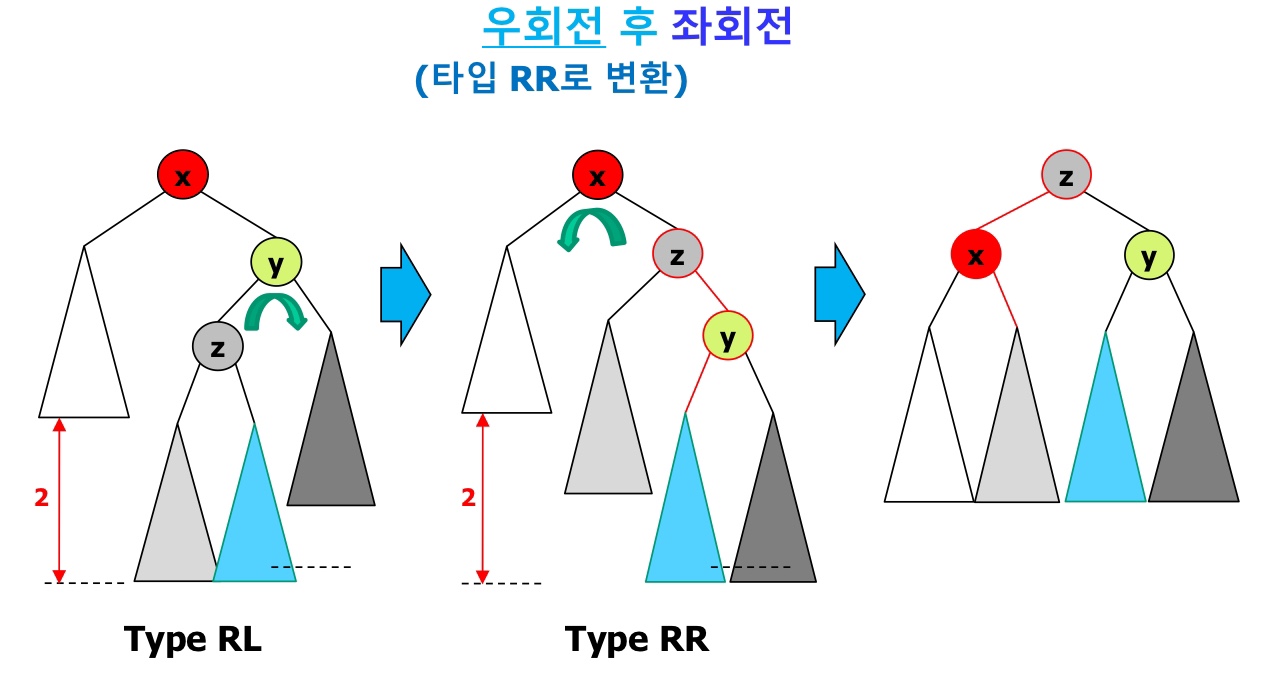
**3. RR : T.right.right가 가장 깊음** 🡪 **좌회전(left Rotation)**

****

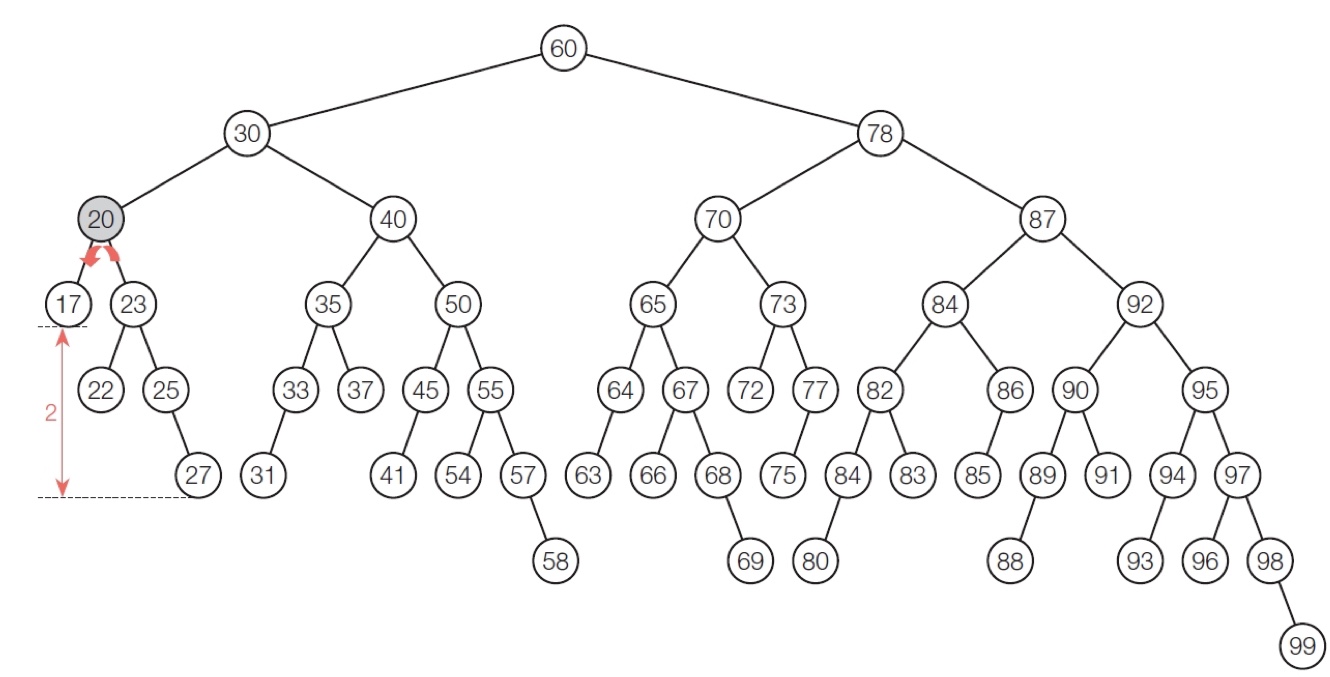
**도표, 디자인, 일러스트레이션이(가) 표시된 사진

자동 생성된 설명**

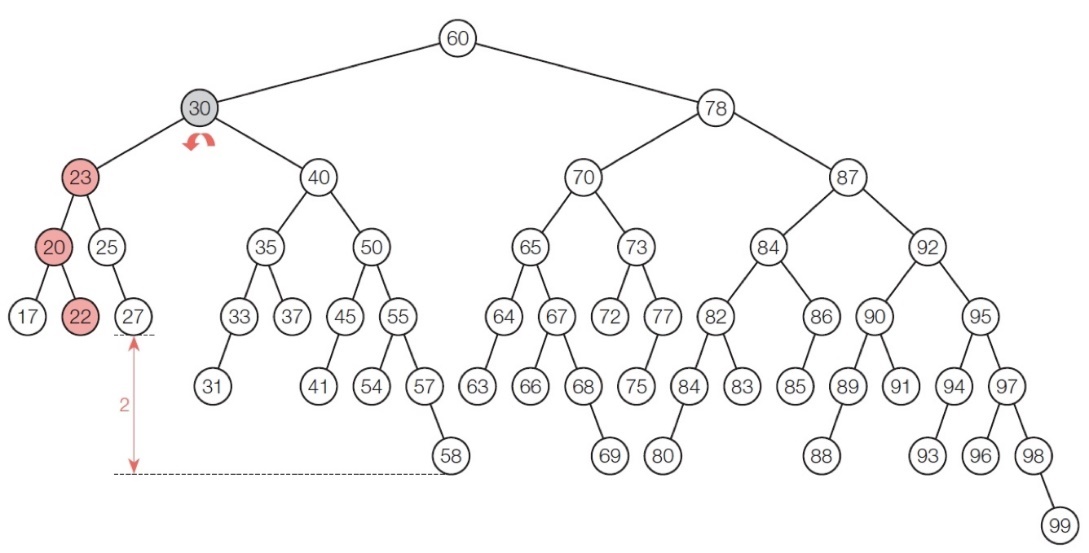
**4. RL : T.right.left가 가장 깊음** 🡪 **우회전(Right Rotation) 🡪 RR 🡪 좌회전**

****

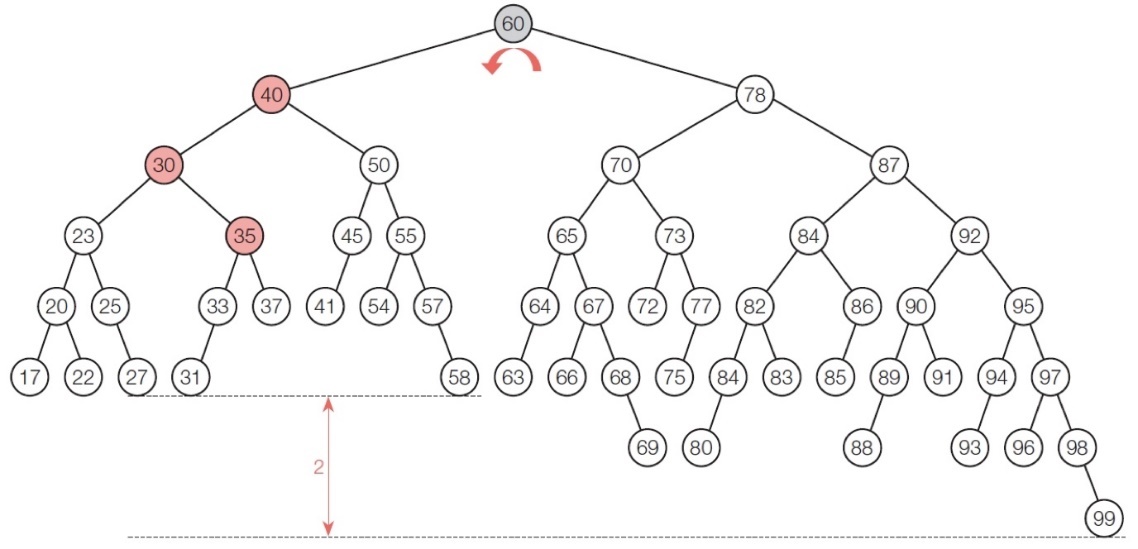
**ex1) RR 타입으로 균형이 깨진 예**

****

**ex2) 한 번의 좌회전 후 상위 서브 트리에서 새롭게 균형이 깨진 상태**

****

**ex3) 두 번의 좌회전 후 상위 서브 트리에서 새롭게 균형이 깨진 상태**

****

**ex4) 세 번의 좌회전 후 균형이 해결된 상태**

**화이트, 패턴, 예술이(가) 표시된 사진

자동 생성된 설명**

**# 알고리즘 구현**

/\*

균형 탐색 트리(AVL 트리): 알고리즘 구현

파일명: AVLTree(demo).cpp

- main : 삽입(Insert), 삭제(Remove), 검색(Search), 전체 출력

\*/

#include <iostream>

#include <algorithm> // max

#include "AVLTree.h" // AVLTree

using namespace std;

void Insert(AVLTree\* avl);

void Remove(AVLTree\* avl);

void Search(AVLTree\* avl);

int main(void)

{

int num;

AVLTree\* avl = new AVLTree();

while (true) {

system("cls");

cout << "\n ### 균형 이진 탐색 트리: AVL ### \n" << endl;

cout << "1) 데이터 삽입" << endl;

cout << "2) 데이터 삭제" << endl;

cout << "3) 데이터 검색" << endl;

cout << "4) 전체 출력" << endl;

cout << "5) 프로그램 종료" << endl;

cout << "\n메뉴 선택: ";

cin >> num;

switch (num) {

case 1: Insert(avl); break;

case 2: Remove(avl); break;

case 3: Search(avl); break;

case 4: avl->Inorder(avl->getRoot()); break;

case 5: cout << "프로그램 종료!!!\n" << endl; return 0;

default: cout << "메뉴를 잘못 선택하셨습니다." << endl;

}

system("pause");

}

return 0;

}

// 데이터 입력

void Insert(AVLTree\* avl) {

int num;

while (true) {

cout << "임의의 정수 입력(종료: 0): ";

cin >> num;

if (num == 0) break;

avl->insert(num);

}

}

// 데이터 삭제

void Remove(AVLTree\* avl) {

int num;

while (true) {

cout << "\n삭제할 임의의 정수 값을 입력하세요(종료: 0): ";

cin >> num;

if (num == 0) break;

avl->remove(num);

}

}

// 데이터 검색

void Search(AVLTree\* avl) {

int num;

AVLNode\* temp = nullptr;

while (true) {

cout << "\n찾을 임의의 정수 값을 입력하세요(종료: 0): ";

cin >> num;

if (num == 0) break;

temp = avl->search(num);

if (temp) cout << temp->getData() << " 키를 찾았습니다!!!" << endl;

else cout << "키를 찾지 못했습니다." << endl;

}

}

/\*

균형 탐색 트리(AVL 트리): 알고리즘 구현

파일명: AVLTree.cpp

클래스: AVLNode

클래스: AVLTree

- 생성자.소멸자 : AVLTree, ~AVLTree

- 데이터 삽입.삭제.검색 : insert, remove, search

- 균형 맞추기 : balance, leftRotate, rightRotate

- 깊이 우선 순회 : Preorder, Inorder, Postorder

\*/

#pragma once

#include <iostream>

#include <algorithm> // max

using namespace std;

// class AVLNode

class AVLNode {

int \_\_data;

AVLNode\* \_\_Llink;

AVLNode\* \_\_Rlink;

int \_\_height;

friend class AVLTree;

public:

AVLNode(const int& data);

int getData(void) const;

int getHeight(void) const;

};

// AVLNode: 생성자와 메소드

AVLNode::AVLNode(const int& data) :

\_\_data(data), \_\_height(1), \_\_Llink(nullptr), \_\_Rlink(nullptr) {}

int AVLNode::getData(void) const { return \_\_data; }

int AVLNode::getHeight(void) const { return \_\_height; }

// class AVLTree

class AVLTree {

private:

AVLNode\* \_\_root;

public:

AVLTree();

~AVLTree();

AVLNode\* getRoot(void) const;

int getHeight(AVLNode\* tNode) const;

AVLNode\* search(const int& data) const;

void insert(const int& data);

void remove(const int& data);

AVLNode\* transPlant(AVLNode\*& root);

AVLNode\* balance(AVLNode\* root);

AVLNode\* leftRotate(AVLNode\* root);

AVLNode\* rightRotate(AVLNode\* root);

void Preorder(AVLNode\* root) const;

void Inorder(AVLNode\* root) const;

void Postorder(AVLNode\* root) const;

};

// AVLTree: 생성자와 메소드

AVLTree::AVLTree() : \_\_root(nullptr) {}

AVLTree::~AVLTree() {}

AVLNode\* AVLTree::getRoot(void) const { return \_\_root; }

int AVLTree::getHeight(AVLNode\* tNode) const {

if (tNode == nullptr)

return 0;

return tNode->\_\_height;

}

// 데이터 검색

AVLNode\* AVLTree::search(const int& data) const {

AVLNode\* tNode = \_\_root;

while (tNode) {

if (tNode->\_\_data == data) return tNode;

else if (tNode->\_\_data > data) tNode = tNode->\_\_Llink;

else tNode = tNode->\_\_Rlink;

}

return nullptr;

}

// 데이터 삽입

void AVLTree::insert(const int& data) {

if (\_\_root == nullptr) {

\_\_root = new AVLNode(data);

return;

}

else if (\_\_root->\_\_data == data)

return;

// 삽입할 노드의 부모 노드 검색

AVLNode\* parent = nullptr;

AVLNode\* tNode = \_\_root;

while (tNode) {

parent = tNode;

if (data < parent->\_\_data) tNode = tNode->\_\_Llink;

else tNode = tNode->\_\_Rlink;

}

// 새로운 노드 생성 및 삽입

if (data < parent->\_\_data) parent->\_\_Llink = new AVLNode(data);

else if (data > parent->\_\_data) parent->\_\_Rlink = new AVLNode(data);

// AVLTree: 높이(\_\_height)

parent->\_\_height = 1 + max(getHeight(parent->\_\_Llink), getHeight(parent->\_\_Rlink));

parent = balance(parent); // 균형 맞추기

}

// 데이터 삭제

void AVLTree::remove(const int& data) {

if (\_\_root == nullptr) {

return;

}

// 삭제할 노드의 위치 탐색

AVLNode\* parent = nullptr, \* child = nullptr;

AVLNode\* tNode = \_\_root;

while (tNode->\_\_data != data) {

parent = tNode;

if (data < tNode->\_\_data) tNode = tNode->\_\_Llink;

else tNode = tNode->\_\_Rlink;

}

// 1) 삭제할 노드가 단말 노드인 경우

if ((tNode->\_\_Llink == nullptr) && (tNode->\_\_Rlink == nullptr)) {

if (parent == nullptr) {

\_\_root->\_\_height = 0;

\_\_root = nullptr;

}

else {

if (parent->\_\_Llink == tNode) parent->\_\_Llink = nullptr;

else parent->\_\_Rlink = nullptr;

// AVLTree: 높이(\_\_height)

parent->\_\_height = 1 + max(getHeight(parent->\_\_Llink), getHeight(parent->\_\_Rlink));

parent = balance(parent); // 균형 맞추기

}

}

// 2) 삭제할 노드가 자식 노드를 한 개 가진 경우

else if ((tNode->\_\_Llink == nullptr) || (tNode->\_\_Rlink == nullptr)) {

if (tNode->\_\_Llink) child = tNode->\_\_Llink;

else child = tNode->\_\_Rlink;

if (parent == nullptr) \_\_root = child;

else {

if (parent->\_\_Llink == tNode) parent->\_\_Llink = child;

else parent->\_\_Rlink = child;

// AVLTree: 높이(\_\_height)

parent->\_\_height = 1 + max(getHeight(parent->\_\_Llink), getHeight(parent->\_\_Rlink));

parent = balance(parent); // 균형 맞추기

}

}

// 3) 삭제할 노드가 자식 노드를 두 개 가진 경우

else {

// 왼쪽 서브 트리에서 후계자 찾기

AVLNode\* succParent = tNode;

AVLNode\* succ = tNode->\_\_Llink;

while (succ->\_\_Rlink != nullptr) {

succParent = succ;

succ = succ->\_\_Rlink;

}

if (succParent->\_\_Llink == succ)

succParent->\_\_Llink = succ->\_\_Llink;

else

succParent->\_\_Rlink = succ->\_\_Llink;

tNode->\_\_data = succ->\_\_data;

tNode = succ;

// AVLTree: 높이(\_\_height)

succParent->\_\_height = 1 + max(getHeight(succParent->\_\_Llink), getHeight(succParent->\_\_Rlink));

succParent = balance(succParent); // 균형 맞추기

}

delete tNode;

}

AVLNode\* AVLTree::transPlant(AVLNode\*& root) {

AVLNode\* parent = root;

AVLNode\* RChild = root->\_\_Rlink;

if (root->\_\_Llink == nullptr) root = root->\_\_Rlink;

else if (root->\_\_Rlink == nullptr) root = root->\_\_Llink;

else {

// 오른쪽 서브 트리에서 가장 작은 값

while (RChild->\_\_Llink != nullptr) {

parent = RChild;

RChild = RChild->\_\_Llink;

}

// root <-- RChild

root->\_\_data = RChild->\_\_data;

if (parent == root) root->\_\_Rlink = RChild->\_\_Rlink;

else parent->\_\_Llink = RChild->\_\_Rlink;

root = RChild;

}

delete RChild;

return root;

}

// 균형 맞추기: LL, LR, RR, RL 유형

AVLNode\* AVLTree::balance(AVLNode\* root) {

if (getHeight(root->\_\_Llink) >= getHeight(root->\_\_Rlink) + 2) { // L 유형

if (getHeight(root->\_\_Llink->\_\_Rlink) > getHeight(root->\_\_Llink->\_\_Llink) + 2)

root->\_\_Llink = leftRotate(root->\_\_Llink); // LR 유형

root = rightRotate(root); // L 유형: LL, LR

}

else if (getHeight(root->\_\_Rlink) >= getHeight(root->\_\_Llink) + 2) { // R 유형

if (getHeight(root->\_\_Rlink->\_\_Llink) > getHeight(root->\_\_Rlink->\_\_Rlink) + 2)

root->\_\_Rlink = rightRotate(root->\_\_Rlink); // RL 유형

root = leftRotate(root); // R 유형: RR, RL

}

return root;

}

// 왼쪽(left) 회전

AVLNode\* AVLTree::leftRotate(AVLNode\* root) {

AVLNode\* RChild = root->\_\_Rlink;

root->\_\_Rlink = RChild->\_\_Llink;

RChild->\_\_Llink = root;

// 높이 재조정: root, RChild

root->\_\_height = max(getHeight(root->\_\_Llink), getHeight(root->\_\_Rlink)) + 1;

RChild->\_\_height = max(getHeight(RChild->\_\_Llink), getHeight(RChild->\_\_Rlink)) + 1;

return RChild; // root --> RChild

}

// 오른쪽(right) 회전

AVLNode\* AVLTree::rightRotate(AVLNode\* root) {

AVLNode\* LChild = root->\_\_Llink;

root->\_\_Llink = LChild->\_\_Rlink;

LChild->\_\_Rlink = root;

// 높이 재조정: root, LChild

root->\_\_height = max(root->\_\_Llink->\_\_height, root->\_\_Rlink->\_\_height) + 1;

LChild->\_\_height = max(LChild->\_\_Llink->\_\_height, LChild->\_\_Rlink->\_\_height) + 1;

return LChild; // root --> LChild

}

// 깊이 우선 순회: 전위 순회

void AVLTree::Preorder(AVLNode\* root) const {

if (root == nullptr)

return;

cout << root->\_\_data << " ";

Preorder(root->\_\_Llink);

Preorder(root->\_\_Rlink);

}

// 깊이 우선 순회: 중위 순회

void AVLTree::Inorder(AVLNode\* root) const {

if (root == nullptr)

return;

Inorder(root->\_\_Llink);

cout << root->\_\_data << " ";

Inorder(root->\_\_Rlink);

}

// 깊이 우선 순회: 후위 순회

void AVLTree::Postorder(AVLNode\* root) const {

if (root == nullptr)

return;

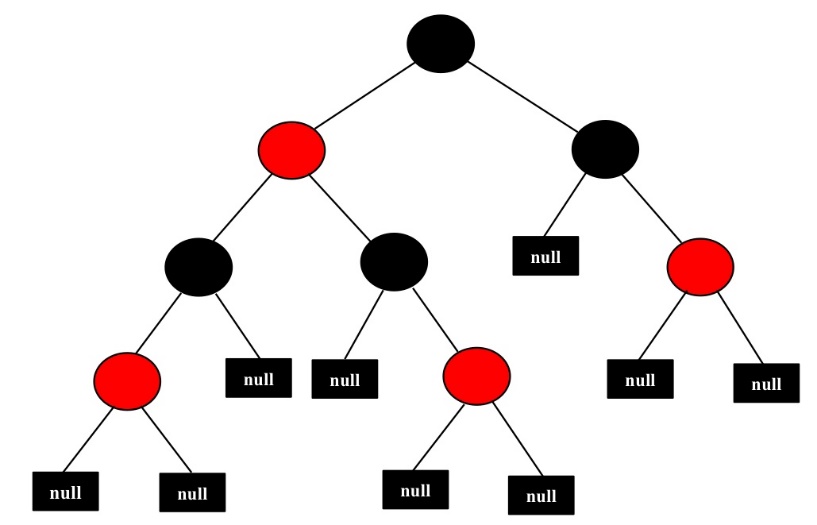
Postorder(root->\_\_Llink);

Postorder(root->\_\_Rlink);

cout << root->\_\_data << " ";

}

**08-2-2. 레드-블랙 트리(Red-Black Tree, RB Tree)**

****

**# 특성**

**1. 모든 null 자리에 단말 노드(leaf node)를 둔다.**

- RB-Tree에서 단말 노드는 이 null를 말한다.

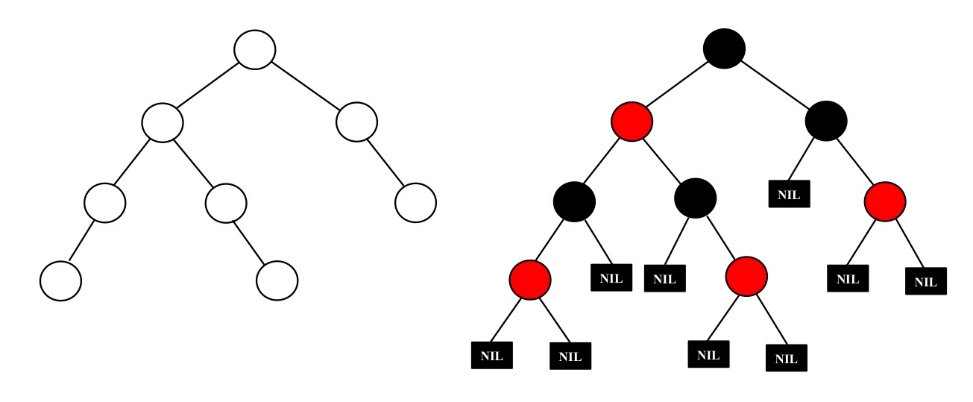
**2. 모든 노드는 Red 또는 Black의 색을 갖는다.**

**- 루트와 모든 단말 노드는 블랙이다.**

- 임의의 단말 노드에 이르는 경로 상에 레드 노드 두 개가 연속으로 출현하지 못한다.

- 임의의 단말 노드에 이르는 경로에서 만나는 블랙 노드의 수(black height)는 모두 같다.

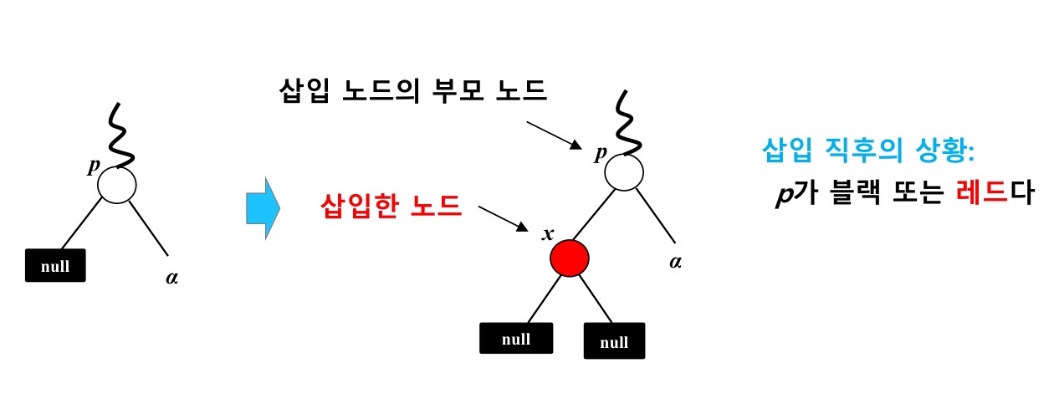
**ex) 이진 검색 트리를 RB-Tree로 만든 예**

****

**# 구성 및 구현**

**\* 삽입ver.1**

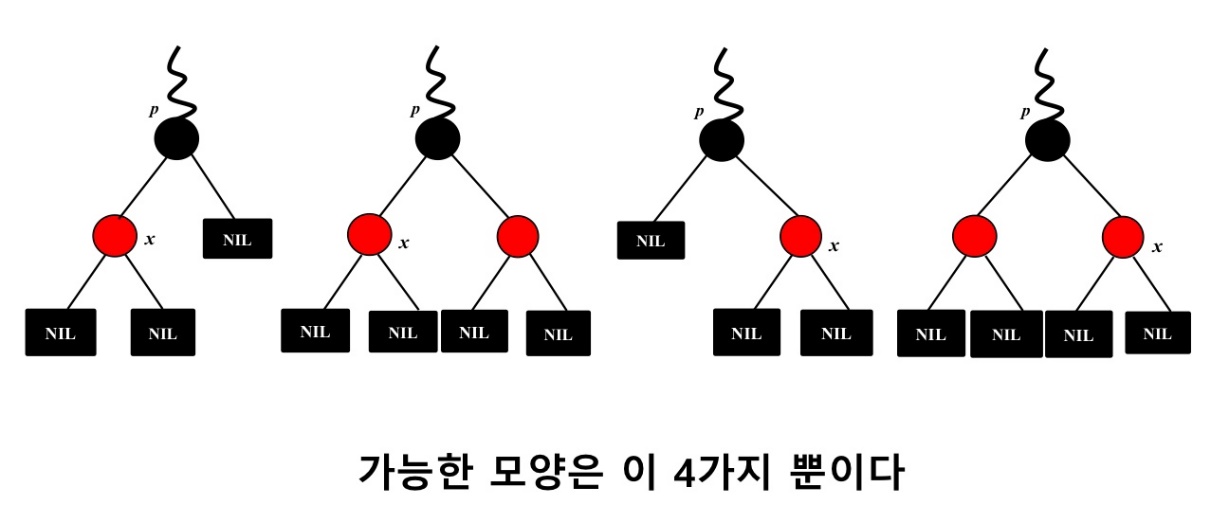
🡪 일반적인 BST의 삽입 작업 후 삽입 노드에 레드를 칠하고, 삽입한 노드의 좌우에 null 리프를 달아준다.

****

**\* 삽입ver.2**

삽입 직후의 상황 : x의 부모 p가 블랙 또는 레드다.

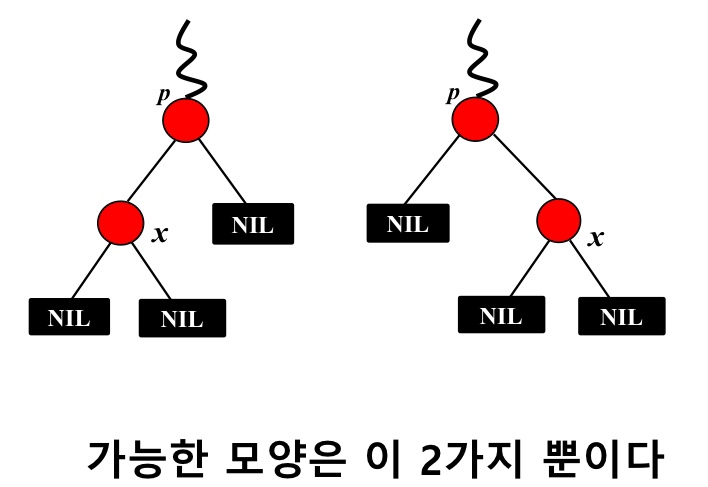
만약 p가 블랙 : RB-Tree 특성을 모두 만족. 완료!!

****

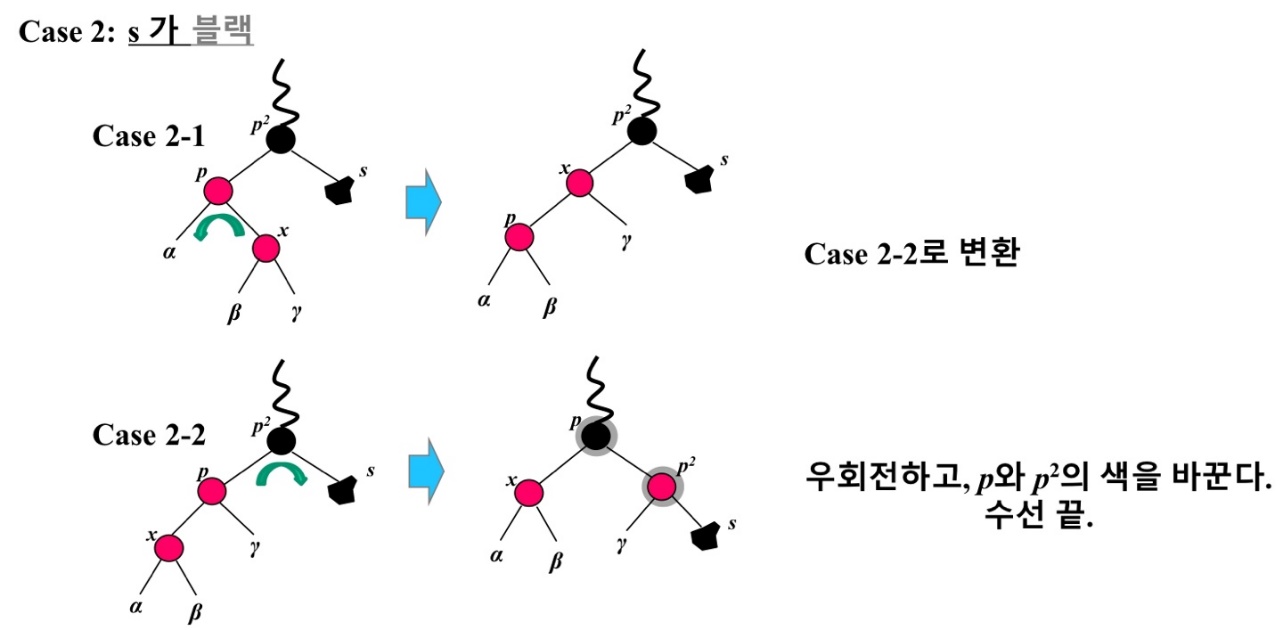
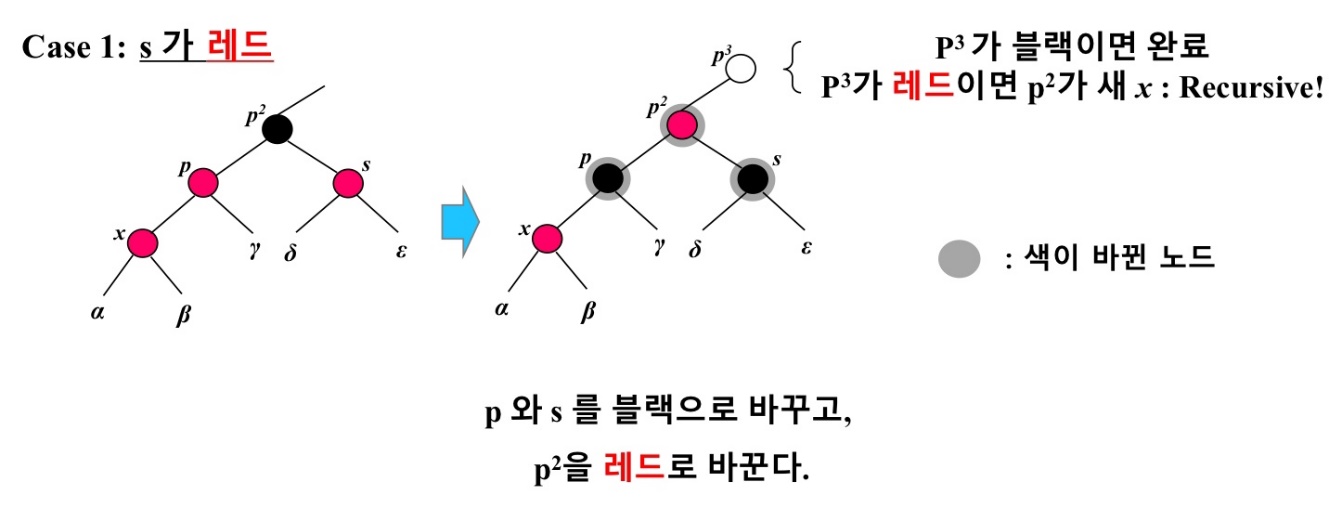
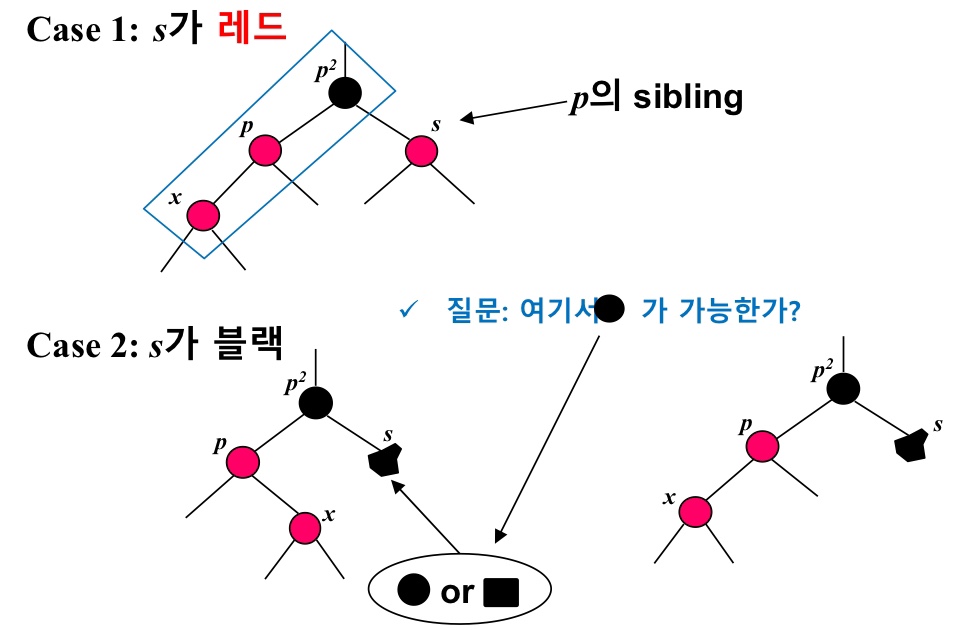
**\* 삽입ver.3**

**만약 p가 레드**

* **RB 특성이 깨졌다. 🡪 수선**
* **임의의 단말 노드에 이르는 경로 상에 레드 노드 두개가 연속으로 출현하지 못한다.**

****

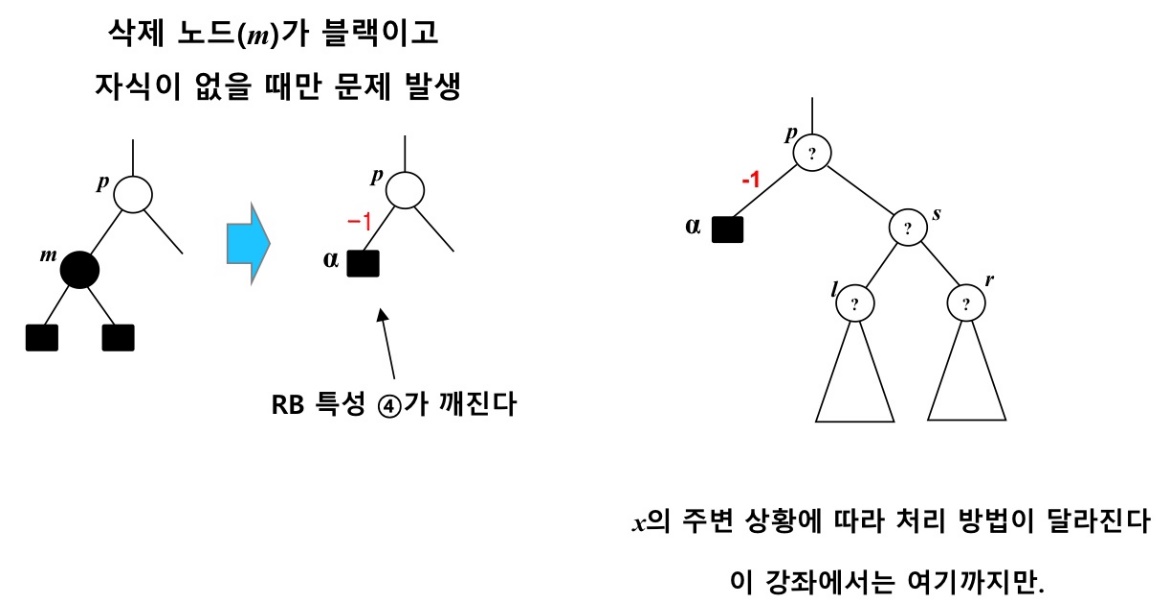
* **p의 형제(sibling) 노드 s에 따라 두가지로 나뉨.**

****

**\* 삭제**

: BST의 삭제 작업 중 Case 1과 2만 고려하면 됨.

**텍스트, 스크린샷, 도표이(가) 표시된 사진

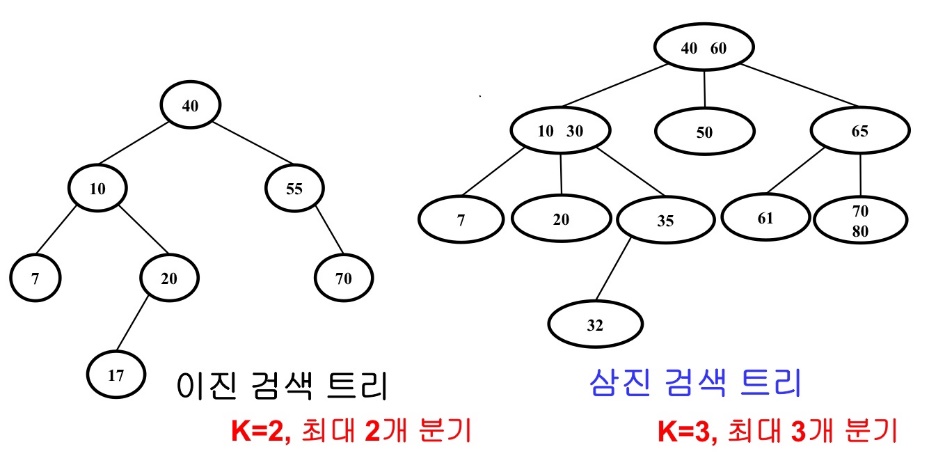
자동 생성된 설명**

**# 시간 복잡도**

**: AVL트리와 RB트리 모두 검색, 삽입, 삭제에 O(logn)시간이 보장된다.**

**08-2-3. B 트리**

ex) K-진 검색 트리

****

**텍스트, 도표, 라인이(가) 표시된 사진

자동 생성된 설명**

**# B-Tree의 환경**

**1. 디스크의 접근 단위는 블록(페이지)**

**2. 디스크에 한 번 접근하는 시간은 수십만 명령어의 처리 시간과 맞먹는다.**

**3. 검색 트리가 디스크에 저장되어 있다면, 트리의 높이를 최소화하는 것이 유리하다.**

**4. B-트리는 K-진 검색 트리가 균형을 유지하도록 하여 최악의 경우 디스크 접근 횟수를 줄인 것이다.**

**# B-Tree의 성질**

**1. 루트를 제외한 모든 노드는 [K/2]~K 개의 키를 갖는다.**

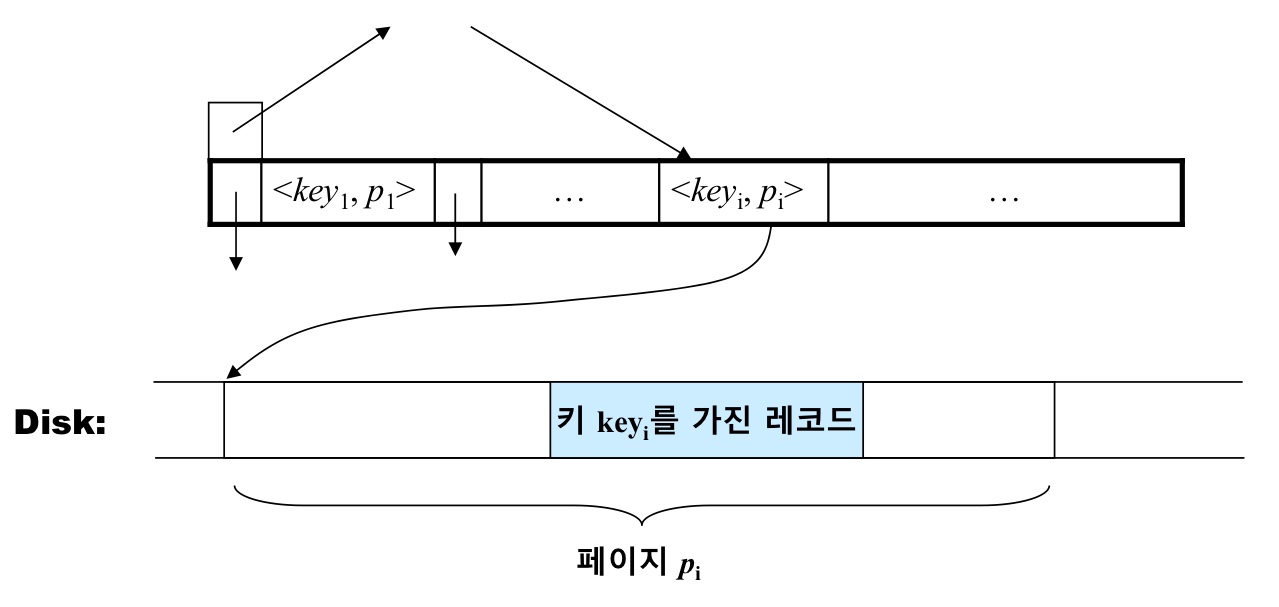
**2. 모든 리프 노드는 같은 깊이를 가진다.**

**# B-Tree의 노드 구조**

**텍스트, 라인, 도표, 폰트이(가) 표시된 사진

자동 생성된 설명**

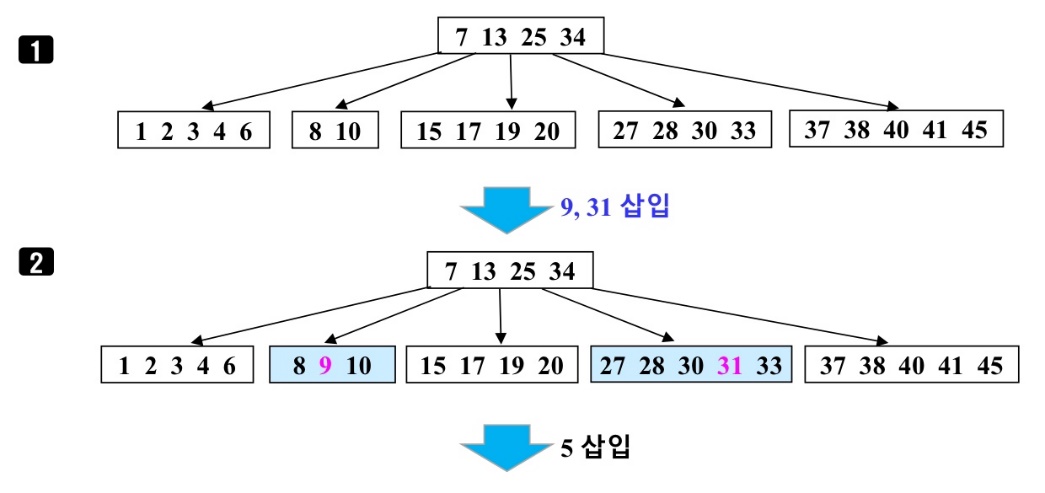
**# B-Tree를 통해 레코드에 접근하는 과정**

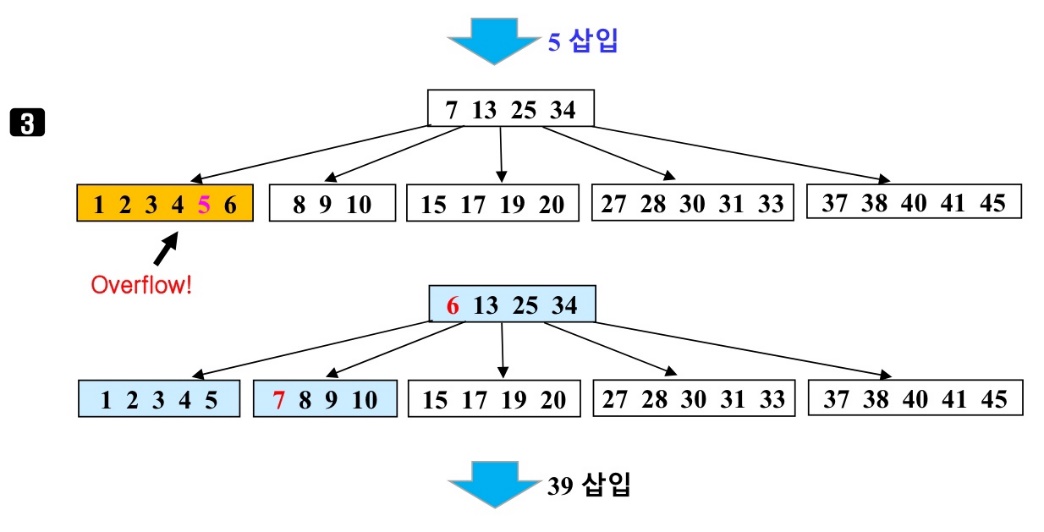
****

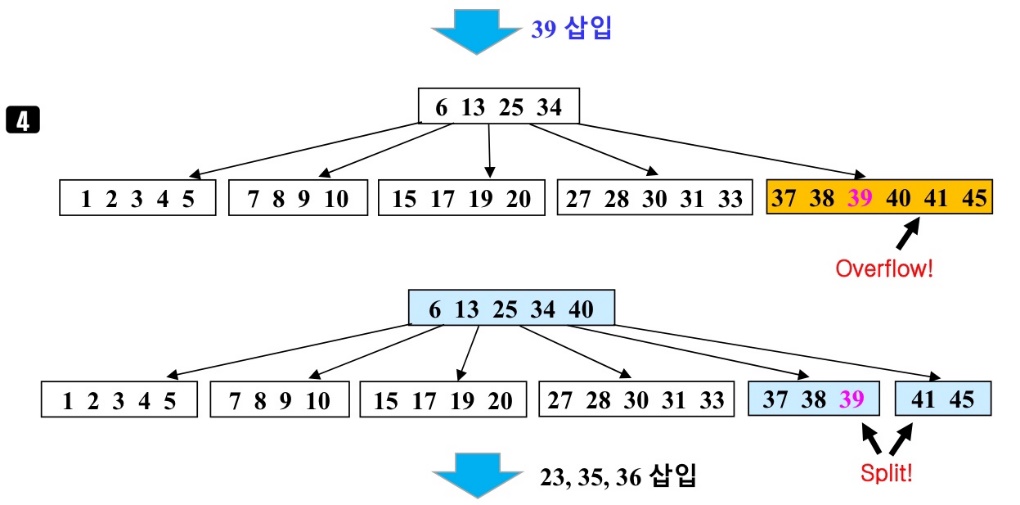
**# B-Tree의 삽입(k=5인 예)**

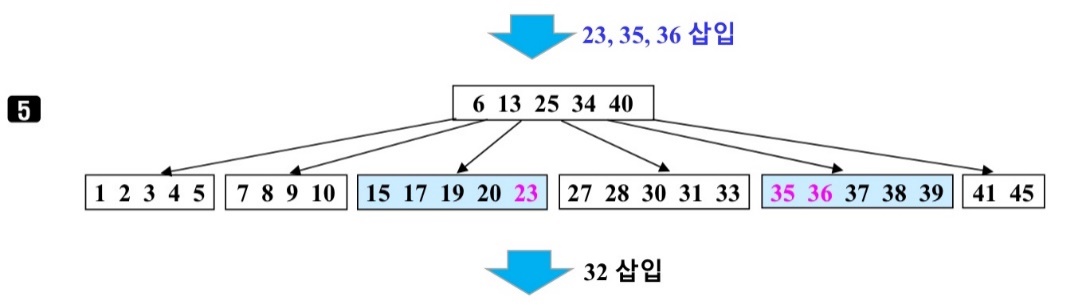
**텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

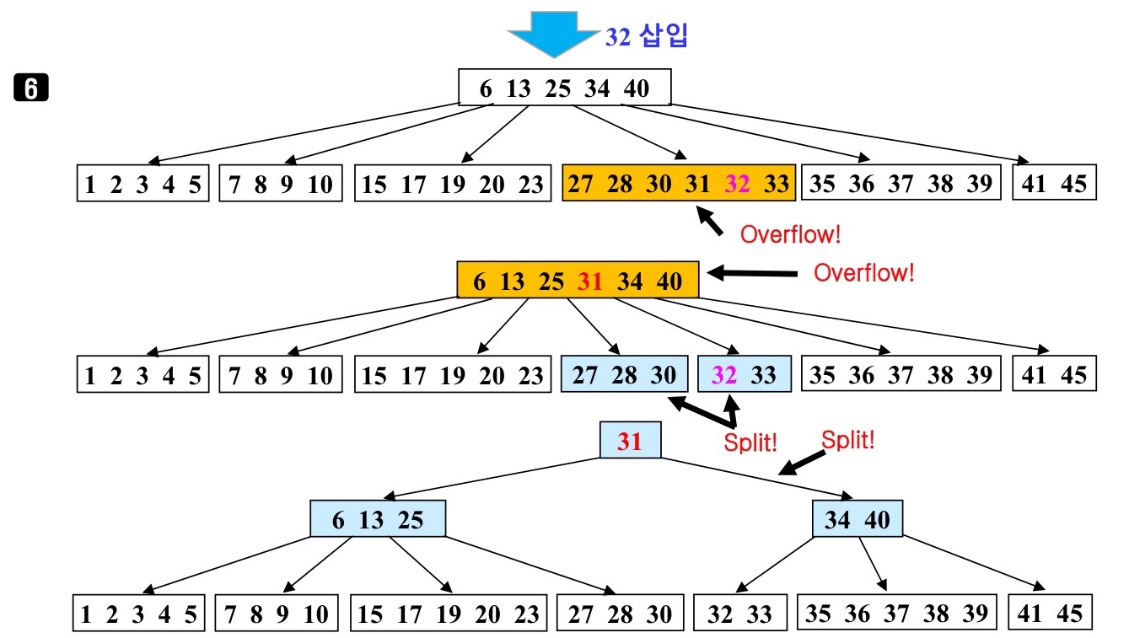
자동 생성된 설명**

****

****

****

****

****

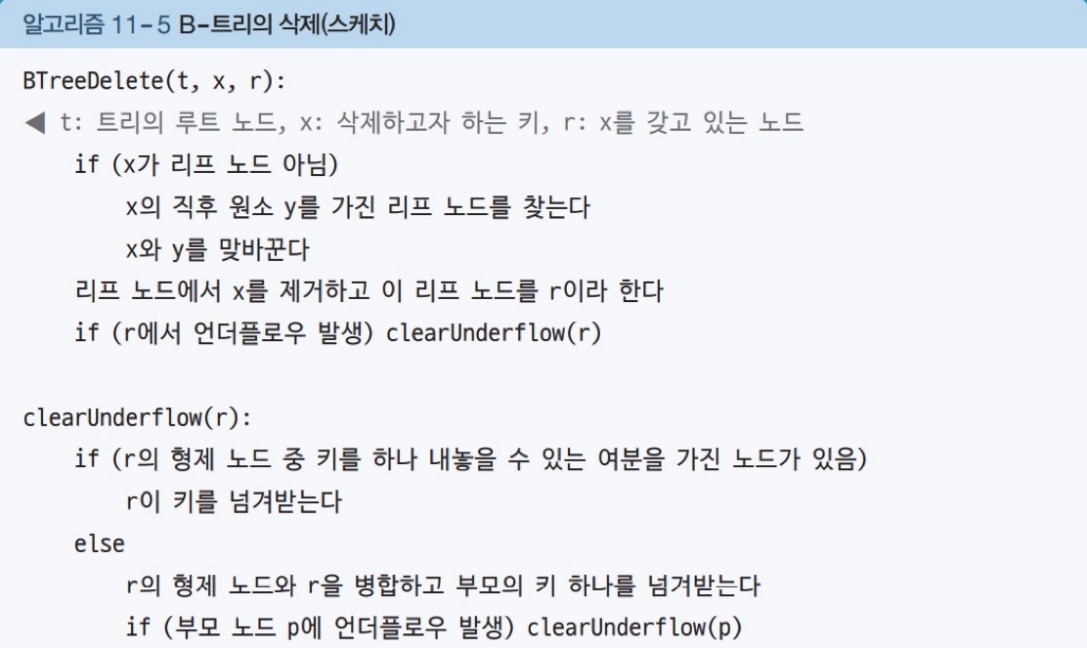
**# B-Tree의 삭제**

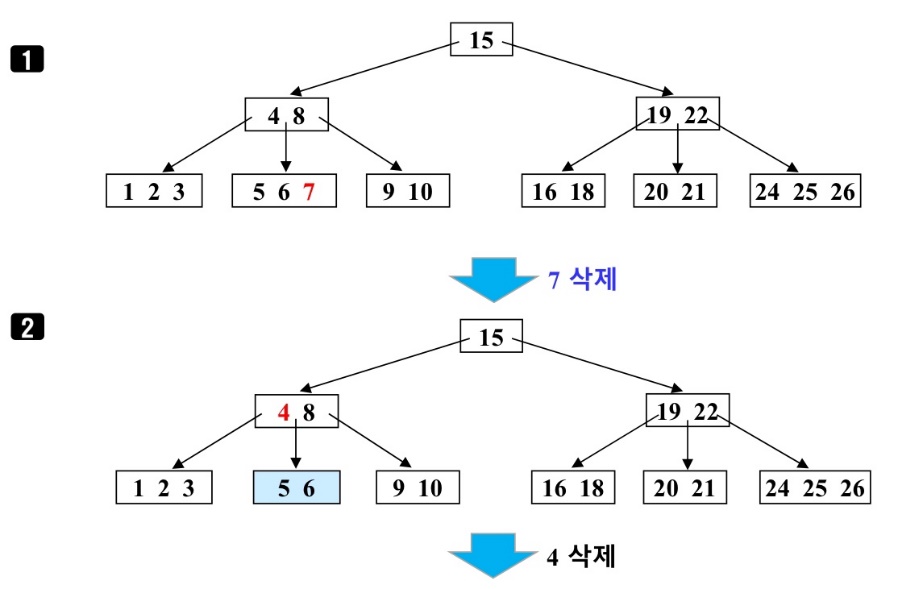
**1. x를 키로 갖고 있는 노드를 찾는다.**

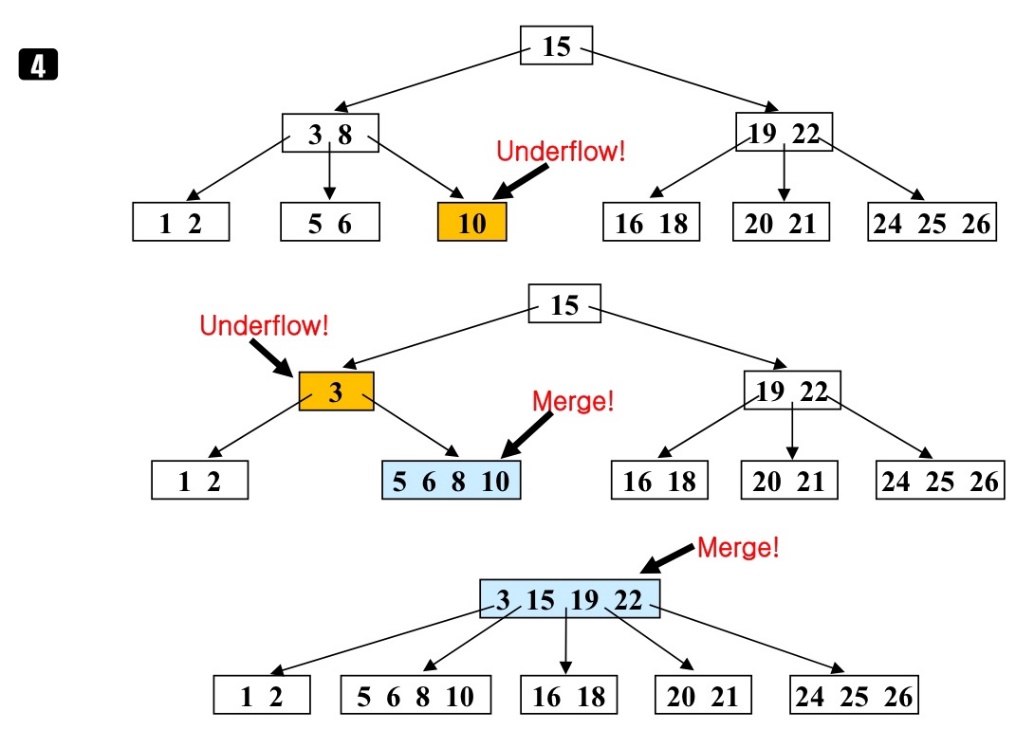
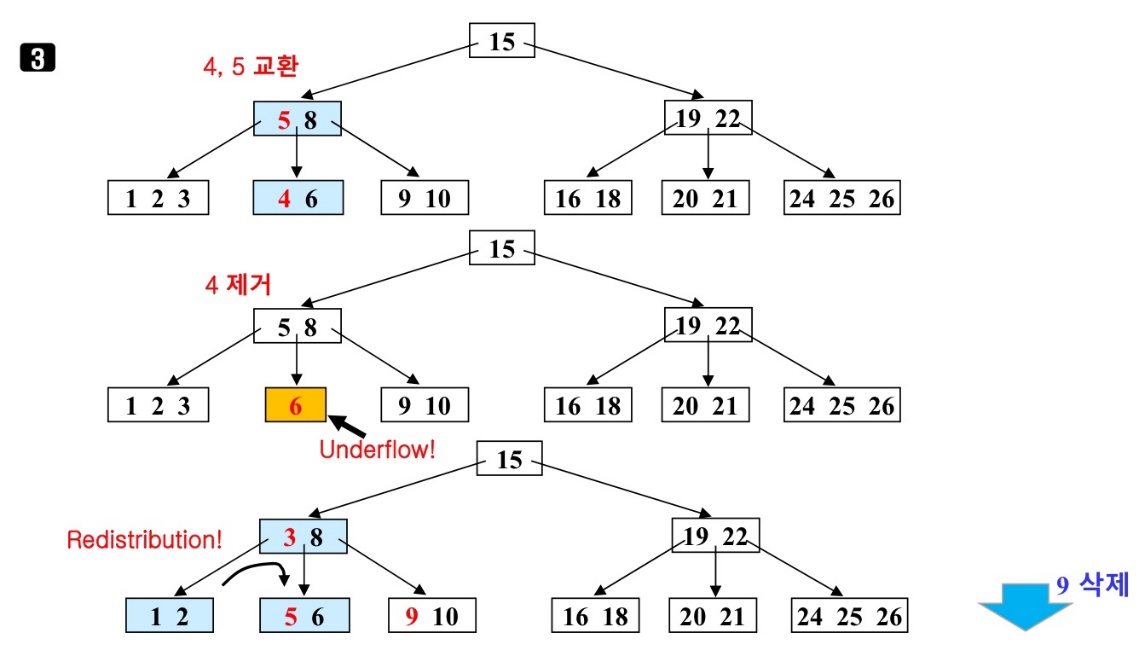
**2. 이 노드가 리프 노드가 아니면 x의 직후 원소 y를 가진 리프 노드 r을 찾아 x와 y을 맞바꾼다.(직후 원소 y는 반드시 리프 노드에 있다.)**

**3. 리프 노드 r에서 x를 제거한다.**

**4. x를 제거한 후 노드에 언더플로우가 발생하면 적절히 해소한다.**

****

****

****

**08-3. 이진탐색트리\_송: 완성해보기**

**# BinarySearchTree.cpp**

#include <iostream>

#include "BinarySearchTree.h"

using namespace std;

void searchBST(BinarySearchTree\* bst);

void insertBST(BinarySearchTree\* bst);

void deleteBST(BinarySearchTree\* bst);

int main(void)

{

int num;

BinarySearchTree\* bst = new BinarySearchTree();

while (true) {

system("cls");

cout << "\n ### 이진 탐색 트리 ### \n" << endl;

cout << "1) 데이터 삽입" << endl;

cout << "2) 데이터 검색" << endl;

cout << "3) 데이터 삭제" << endl;

cout << "4) 전체 출력" << endl;

cout << "5) 프로그램 종료" << endl;

cout << "\n메뉴 선택: ";

cin >> num;

switch (num) {

case 1: insertBST(bst); break;

case 2: searchBST(bst); break;

case 3: deleteBST(bst); break;

case 4: bst->printBSTAll(bst->getRoot()); break;

case 5: cout << "프로그램 종료!!!\n" << endl; return 0;

default: cout << "메뉴를 잘못 선택하셨습니다." << endl;

}

system("pause");

}

return 0;

}

// 이진 탐색 트리(BST): 데이터 검색

void searchBST(BinarySearchTree\* bst) {

int num;

DNode\* temp = nullptr;

while (true) {

cout << "\n찾을 임의의 정수 값을 입력하세요(종료: 0): ";

cin >> num;

if (num == 0)

break;

// 데이터검색

temp = bst->search(num);

if (temp) cout << temp->\_\_data << " 키를 찾았습니다!!!" << endl;

else cout << "키를 찾지 못했습니다." << endl;

}

}

// 이진 탐색 트리(BST): 데이터 입력

void insertBST(BinarySearchTree\* bst) {

int num;

while (true) {

cout << "임의의 정수 입력(종료: 0): ";

cin >> num;

if (num == 0)

break;

// 데이터 삽입

bst->insert(num);

}

}

// 이진 탐색 트리(BST): 데이터 삭제

void deleteBST(BinarySearchTree\* bst) {

int num;

DNode\* temp = nullptr;

while (true) {

cout << "\n삭제할 임의의 정수 값을 입력하세요(종료: 0): ";

cin >> num;

if (num == 0)

break;

// 데이터 삭제

bst->remove(num);

}

}

**# BinarySearchTree.h**

#pragma once

#include <iostream>

#include <queue>

using namespace std;

class DNode {

public:

int \_\_data;

DNode\* \_\_Llink;

DNode\* \_\_Rlink;

DNode(const int& data);

};

// 새로운 노드(data, link) 생성

DNode::DNode(const int& data) :

\_\_data(data), \_\_Llink(nullptr), \_\_Rlink(nullptr) {}

class BinarySearchTree {

private:

DNode\* \_\_root;

DNode\* \_\_search(DNode\* root, const int& data) {

}

void \_\_insert(DNode\* root, const int& data) {

}

public:

DNode\* getRoot(void) const { return \_\_root; }

DNode\* search(const int& data);

void insert(const int& data);

void remove(const int& data);

void printBSTAll(DNode\* root) const;

};

// 전체 노드 출력 -- 중위 순회

void BinarySearchTree::printBSTAll(DNode\* root) const {

if (root) {

printBSTAll(root->\_\_Llink);

cout << root->\_\_data << " ";

printBSTAll(root->\_\_Rlink);

}

}

// 데이터 검색

DNode\* BinarySearchTree::search(const int& data) {

}

// 데이터 삽입

void BinarySearchTree::insert(const int& data) {

}

// 데이터 삭제

void BinarySearchTree::remove(const int& data) {

}