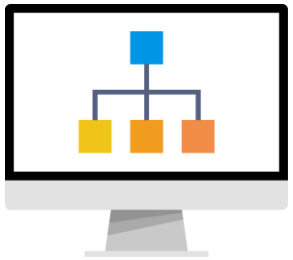


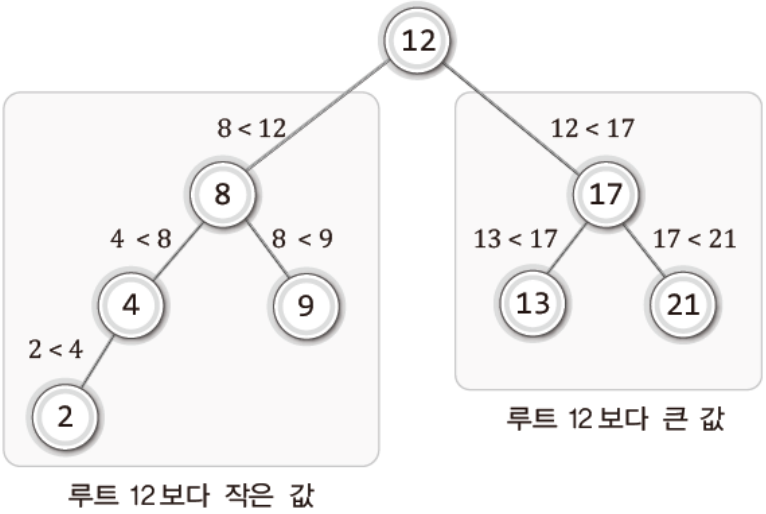
Data Structure & Algorithm
자료구조 및 알고리즘

22. 이진 탐색 트리 (Binary Search Tree)

이진 탐색 트리의 이해

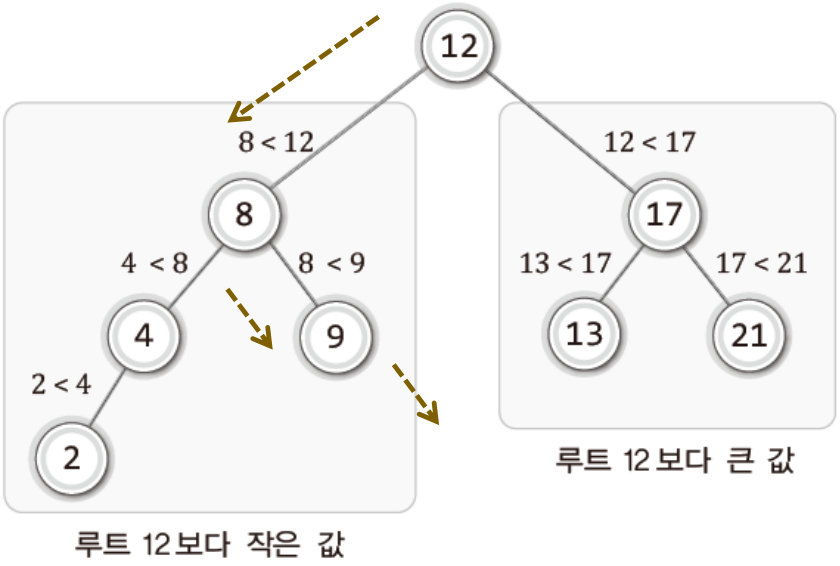
이진 탐색 트리 = 이진 트리 + 데이터의 저장 규칙

자료구조	최악의 경우 탐색 시간	데이터 추가 시간	비고
배열 + 선형탐색	O(N)	O(1)	
배열 + 이분탐색	O(log N)	O(N)	배열이 정렬되어 있음
(균형이 잘 맞는) 이진 탐색 트리	O(log N)	O(log N)	균형을 잘 맞추어야 함!

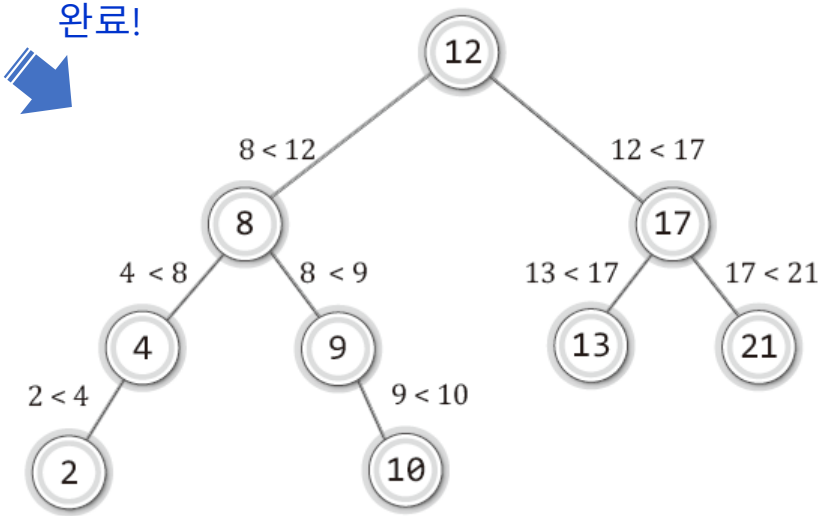


- 이진 탐색 트리의 노드에 저장된 키(key)는 유일!
- 루트 노드의 키 > 왼쪽 서브 트리를 구성하는 키
- 루트 노드의 키 < 오른쪽 서브 트리를 구성하는 키
- 왼쪽과 오른쪽 서브 트리도 이진 탐색 트리!

이진 탐색 트리의 노드 추가 과정



완료!



새 노드의 추가 과정은, 반대로 탐색의 과정이 된다.

이진 탐색 트리의 헤더파일

BinaryTree2.h (저번 수업 시간에 했음)

- `BTreeNode * MakeBTreeNode(void);`
  노드를 동적으로 할당해서 그 노드의 주소 값을 반환한다.
- `BTData GetData(BTreeNode * bt);`
  노드에 저장된 데이터를 반환한다.
- `void SetData(BTreeNode * bt, BTData data);`
  인자로 전달된 데이터를 노드에 저장한다.
- `BTreeNode * GetLeftSubTree(BTreeNode * bt);`
  인자로 전달된 노드의 왼쪽 자식 노드의 주소 값을 반환한다.
- `BTreeNode * GetRightSubTree(BTreeNode * bt);`
  인자로 전달된 노드의 오른쪽 자식 노드의 주소 값을 반환한다.
- `void MakeLeftSubTree(BTreeNode * main, BTreeNode * sub);` 기존 왼쪽 자식 노드 삭제!
  인자로 전달된 노드의 왼쪽 자식 노드를 교체한다.
- `void MakeRightSubTree(BTreeNode * main, BTreeNode * sub);` 기존 오른쪽 자식 노드 삭제!
  인자로 전달된 노드의 오른쪽 자식 노드를 교체한다.

BinarySearchTree.h 삭제 관련 함수는 후에 별도 추가!

```
// BST의 생성 및 초기화
void BSTMakeAndInit(BTreeNode ** pRoot);

// 노드에 저장된 데이터 반환
BSTData BSTGetNodeData(BTreeNode * bst);

// BST를 대상으로 데이터 저장(노드의 생성과정 포함)
void BSTInsert(BTreeNode ** pRoot, BSTData data);

// BST를 대상으로 데이터 탐색
BTreeNode * BSTSearch(BTreeNode * bst, BSTData target);
```

BinaryTree2.h에 선언된 함수들을 이용해서 위의 함수들을 정의한다!

이진 탐색 트리의 헤더파일

BinaryTree2.h (저번 수업 시간에 했음)

- `BTreeNode * MakeBTreeNode(void);`
  노드를 동적으로 할당해서 그 노드의 주소 값을 반환한다.
- `BTData GetData(BTreeNode * bt);`
  노드에 저장된 데이터를 반환한다.
- `void SetData(BTreeNode * bt, BTData data);`
  인자로 전달된 데이터를 노드에 저장한다.
- `BTreeNode * GetLeftSubTree(BTreeNode * bt);`
  인자로 전달된 노드의 왼쪽 자식 노드의 주소 값을 반환한다.
- `BTreeNode * GetRightSubTree(BTreeNode * bt);`
  인자로 전달된 노드의 오른쪽 자식 노드의 주소 값을 반환한다.
- `void MakeLeftSubTree(BTreeNode * main, BTreeNode * sub);` 기존 왼쪽 자식 노드 삭제!
  인자로 전달된 노드의 왼쪽 자식 노드를 교체한다.
- `void MakeRightSubTree(BTreeNode * main, BTreeNode * sub);` 기존 오른쪽 자식 노드 삭제!
  인자로 전달된 노드의 오른쪽 자식 노드를 교체한다.

BinarySearchTree.h 삭제 관련 함수는 후에 별도 추가!

```
// BST의 생성 및 초기화
void BSTMakeAndInit(BTreeNode ** pRoot);

// 노드에 저장된 데이터 반환
BSTData BSTGetNodeData(BTreeNode * bst);

// BST를 대상으로 데이터 저장(노드의 생성과정 포함)
void BSTInsert(BTreeNode ** pRoot, BSTData data);

// BST를 대상으로 데이터 탐색
BTreeNode * BSTSearch(BTreeNode * bst, BSTData target);
```

BinaryTree2.h에 선언된 함수들을 이용해서 위의 함수들을 정의한다!

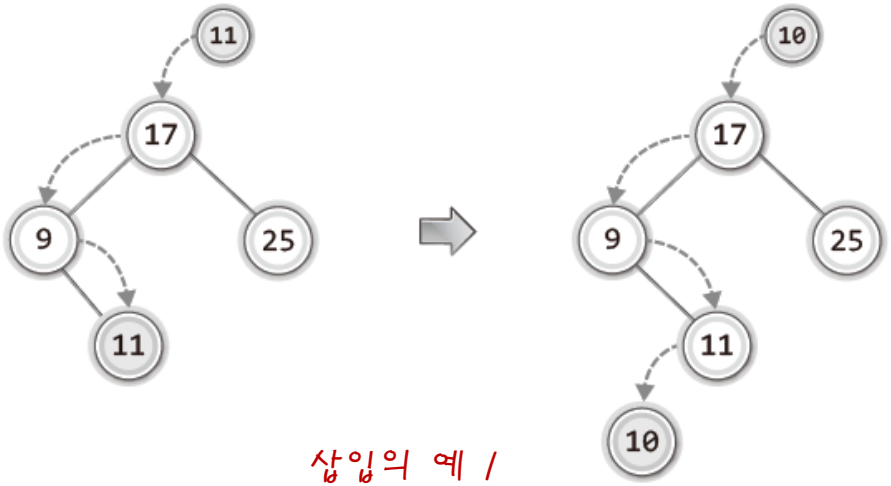
이진 탐색 트리 기반 main 함수

```
int main(void)
{
    BTreeNode * bstRoot;        // bstRoot는 BST의 루트 노드를 가리킨다.
    BTreeNode * sNode;

    BSTMakeAndInit(&bstRoot);   // Binary Search Tree의 생성 및 초기화

    BSTInsert(&bstRoot, 1);     // bstRoot에 1을 저장
    BSTInsert(&bstRoot, 2);     // bstRoot에 2를 저장
    BSTInsert(&bstRoot, 3);     // bstRoot에 3을 저장

    // 탐색! 1이 저장된 노드를 찾아서!
    sNode = BSTSearch(bstRoot, 1);

    if(sNode == NULL)
        printf("탐색 실패 \n");
    else
        printf("탐색에 성공한 키의 값: %d \n", BSTGetNodeData(sNode));
    return 0;
}
```

이진 탐색 트리의 사용자 입장에서는 단순히 데이터를 저장하고 탐색하면 된다!

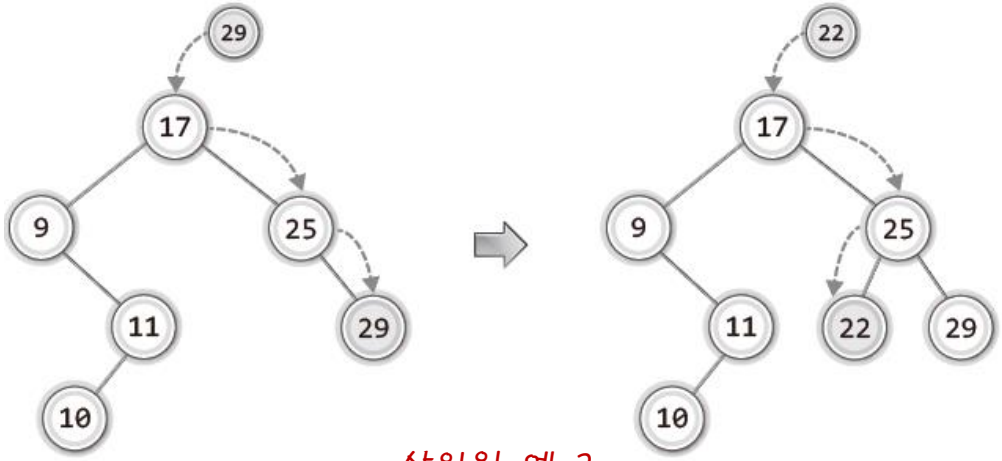
어떠한 형태로 트리가 구성이 되는지 알 필요가 없다!

이진 탐색 트리의 구현: 삽입과 탐색



삽입의 예 1

"비교대상이 없을 때까지 내려간다. 그리고 비교대상이 없는 그 위치가 새 데이터가 저장될 위치이다."



삽입의 예 2

이진 탐색 트리의 구현: 삽입 함수의 구현

```
void BSTInsert(BTreeNode ** pRoot, BSTData data)
{
    BTreeNode * pNode = NULL;       // parent node
    BTreeNode * cNode = *pRoot;     // current node
    BTreeNode * nNode = NULL;       // new node

    // 새로운 노드가(새 데이터가 담긴 노드가) 추가될 위치를 찾는다.
    while(cNode != NULL)
    {
        if(data == GetData(cNode))
            return;    // 데이터의(키의) 중복을 허용하지 않음

        pNode = cNode;

        if(GetData(cNode) > data)
            cNode = GetLeftSubTree(cNode);
        else
            cNode = GetRightSubTree(cNode);
    }

    nNode = MakeBTreeNode();
    SetData(nNode, data);

    // pNode의 자식 노드로 새 노드를 추가
    if(pNode != NULL)     // 새 노드가 루트 노드가 아니라면,
    {
        if(data < GetData(pNode))
            MakeLeftSubTree(pNode, nNode);
        else
            MakeRightSubTree(pNode, nNode);
    }
    else       // 새 노드가 루트 노드라면,
    {
        *pRoot = nNode;
    }
}
```

이어서…

이진 탐색 트리의 구현: 탐색 함수의 구현

```
BTreeNode * BSTSearch(BTreeNode * bst, BSTData target)
{
    BTreeNode * cNode = bst;     // current node
    BSTData cd;                  // current data

    while(cNode != NULL)
    {
        cd = GetData(cNode);

        if(target == cd)
            return cNode;
        else if(target < cd)
            cNode = GetLeftSubTree(cNode);
        else
            cNode = GetRightSubTree(cNode);
    }

    return NULL;     // 탐색대상이 저장되어 있지 않음.
}
```

삽입의 과정을 근거로 탐색을 진행한다.

따라서 구현하기가 쉽다!

탐색에 성공하면 해당 노드의 주소 값을 반환!

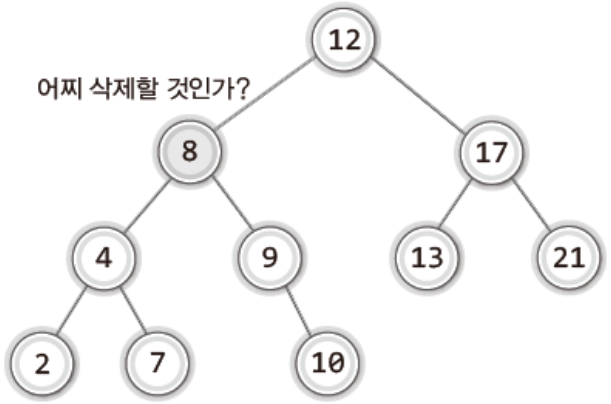
이진 탐색 트리의 구현: 소스파일 & 실행

```
#include "BinaryTree2.h"
#include "BinarySearchTree.h"

void BSTMakeAndInit(BTreeNode ** pRoot)
{
    *pRoot = NULL;
}

BSTData BSTGetNodeData(BTreeNode * bst)
{
    return GetData(bst);
}

void BSTInsert(BTreeNode ** pRoot, BSTData data)
{
    // 위에서 소개하였으니 생략합니다.
}

BTreeNode * BSTSearch(BTreeNode * bst, BSTData target)
{
    // 위에서 소개하였으니 생략합니다.
}
```

BinaryTree2.h
BinaryTree2.c
BinarySearchTree.h
BinarySearchTree.c
BinarySearchTreeMain.c

실행을 위한 파일의 구성!

이진 탐색 트리 삭제 구현: 상황 별 삭제



삭제로 인한 빈 자리를 어떻게 채워야 할까?

삭제와 관련해서 고려해야 할 세 가지 상황

- 상황 1 삭제할 노드가 단말 노드인 경우
- 상황 2 삭제할 노드가 하나의 자식 노드를(하나의 서브 트리를) 갖는 경우
- 상황 3 삭제할 노드가 두 개의 자식 노드를(두 개의 서브 트리를) 갖는 경우

각 상황 별로 추가로 삭제할 노드가 루트 노드인 경우를 구분해야 하지만 이를 피해가는 형태로 코드를 구성하기로 한다!

이진 탐색 트리 삭제 구현: 상황 1

삭제할 노드가 단말 노드인 경우!

가장 쉽게 삭제가 가능한 상황이다!

코드레벨 구현

```
// dNode와 pNode는 각각 삭제할 노드와 이의 부모 노드를 가리키는 포인터 변수
if(삭제할 노드가 단말 노드이다!)
{
    if(GetLeftSubTree(pNode) == dNode)     // 삭제할 노드가 왼쪽 자식 노드라면,
        RemoveLeftSubTree(pNode);          // 왼쪽 자식 노드 트리에서 제거
    else                                   // 삭제할 노드가 오른쪽 자식 노드라면,
        RemoveRightSubTree(pNode);         // 오른쪽 자식 노드 트리에서 제거
}
```

RemoveLeftSubTree, RemoveRightSubTree 함수는 BinaryTree2.h와 BinaryTree2.c에 추가 할 함수

이진 탐색 트리 삭제 구현: 상황 2

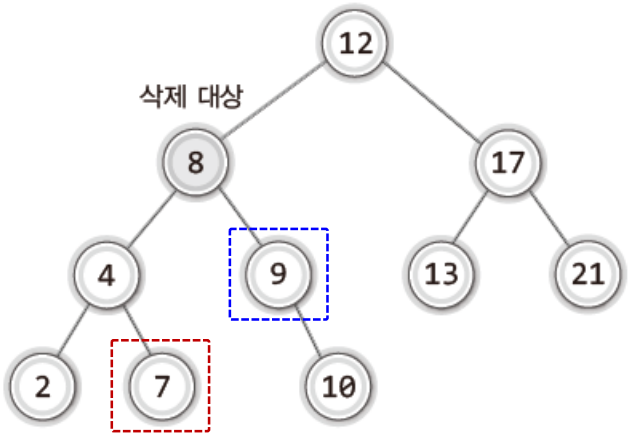
삭제할 노드가 하나의 자식 노드 갖는 경우

```
// dNode와 pNode는 각각 삭제할 노드와 이의 부모 노드를 가리키는 포인터 변수
if(삭제할 노드가 하나의 자식 노드를 지닌다!)
{
    BTreeNode * dcNode;      // 삭제 대상의 자식 노드를 가리키는 포인터 변수

    // 삭제 대상의 자식 노드를 찾는다.
    if(GetLeftSubTree(dNode) != NULL)       // 자식 노드가 왼쪽에 있다면,
        dcNode = GetLeftSubTree(dNode);
    else                                    // 자식 노드가 오른쪽에 있다면,
        dcNode = GetRightSubTree(dNode);

    // 삭제 대상의 부모 노드와 자식 노드를 연결한다.
    if(GetLeftSubTree(pNode) == dNode)      // 삭제 대상이 왼쪽 자식 노드이면,
        ChangeLeftSubTree(pNode, dcNode);       // 왼쪽으로 연결
    else                                    // 삭제 대상이 오른쪽 자식 노드이면,
        ChangeRightSubTree(pNode, dcNode);      // 오른쪽으로 연결
}
```

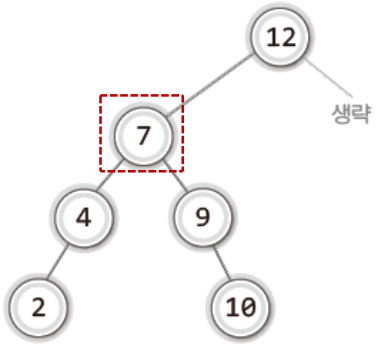
코드레벨 구현

ChangeLeftSubTree, ChangeRightSubTree 함수는 BinaryTree2.h와 BinaryTree2.c에 추가 할 함수

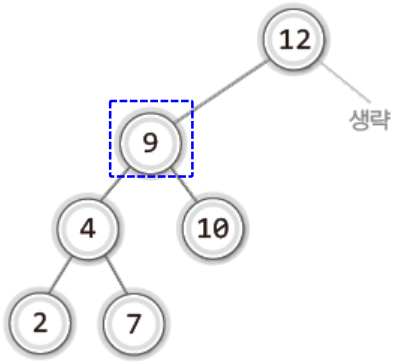
이진 탐색 트리 삭제 구현: 상황 3



삭제 완료



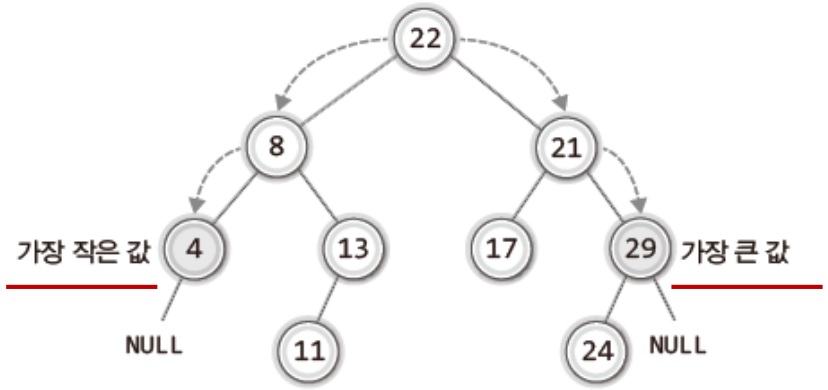
왼쪽 가장 큰 값 대체 결과



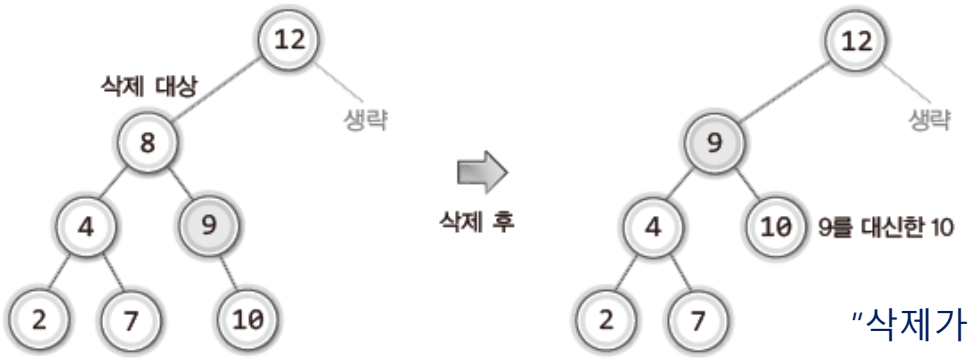
오른쪽 가장 작은 값 대체 결과

왼쪽 서브 트리에서 가장 큰 값,

또는 오른쪽 서브 트리에서 가장 작은 값으로 대체

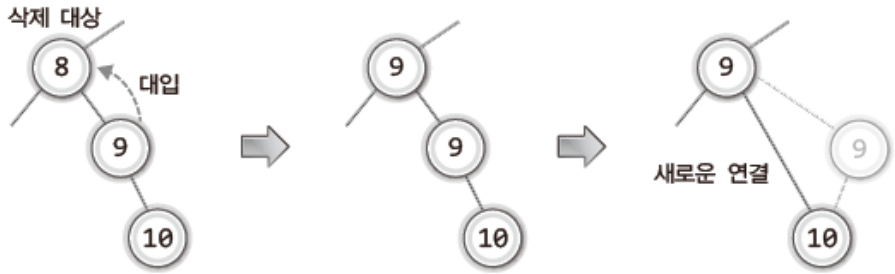


이진 탐색 트리 삭제 구현: 적용 모델



진행 단계

"삭제가 되는 8이 저장된 노드를 9가 저장된 노드로 대체한다. 그리고 이로 인해서 생기는 빈 자리는 9가 저장된 노드의 자식 노드로 대체한다."



- 단계 1 삭제할 노드를 대체할 노드를 찾는다.
- 단계 2 대체할 노드에 저장된 값을 삭제할 노드에 대입한다.
- 단계 3 대체할 노드의 부모 노드와 자식 노드를 연결한다.

이진 탐색 트리 삭제 구현: 상황 3의 구현

```
// dNode와 pNode는 각각 삭제할 노드와 이의 부모 노드를 가리키는 포인터 변수
if(삭제할 노드가 두 개의 자식 노드를 지닌다)
{
    BTreeNode * mNode = GetRightSubTree(dNode);    // mNode는 대체 노드 가리킴
    BTreeNode * mpNode = dNode;      // mpNode는 대체 노드의 부모 노드 가리킴
    . . . .

    // 단계 1. 삭제 대상의 대체 노드를 찾는다.
    while(GetLeftSubTree(mNode) != NULL)
    {
        mpNode = mNode;
        mNode = GetLeftSubTree(mNode);
    }

    // 단계 2. 대체할 노드에 저장된 값을 삭제할 노드에 대입한다.
    SetData(dNode, GetData(mNode));

    // 단계 3. 대체할 노드의 부모 노드와 자식 노드를 연결한다.
    if(GetLeftSubTree(mpNode) == mNode)        // 대체할 노드가 왼쪽 자식 노드라면
    {
        // 대체할 노드의 자식 노드를 부모 노드의 왼쪽에 연결
        ChangeLeftSubTree(mpNode, GetRightSubTree(mNode));
    }
    else       // 대체할 노드가 오른쪽 자식 노드라면
    {
        // 대체할 노드의 자식 노드를 부모 노드의 오른쪽에 연결
        ChangeRightSubTree(mpNode, GetRightSubTree(mNode));
    }
    . . . .
}
```

이어서…

자식 노드가 있다면 오른쪽 자식 노드이다!

자식 노드가 있다면 오른쪽 자식 노드이다!

삭제 구현을 위한 이진 트리의 확장

BinaryTree2.h와 BinaryTree2.c에 추가되는 함수들!

BinaryTree3.h의 일부

```
// 왼쪽 자식 노드를 트리에서 제거, 제거된 노드의 주소 값이 반환된다.
• BTreeNode * RemoveLeftSubTree(BTreeNode * bt);

// 오른쪽 자식 노드를 트리에서 제거, 제거된 노드의 주소 값이 반환된다.
• BTreeNode * RemoveRightSubTree(BTreeNode * bt);

// 메모리 소멸을 수반하지 않고 main의 왼쪽 자식 노드를 변경한다.
• void ChangeLeftSubTree(BTreeNode * main, BTreeNode * sub);

// 메모리 소멸을 수반하지 않고 main의 오른쪽 자식 노드를 변경한다.
• void ChangeRightSubTree(BTreeNode * main, BTreeNode * sub);
```

필요에 의해서 앞서 만들어 놓은 도구의 기능을 확장 및 추가!

확장된 함수의 구현

```
// 메모리의 소멸을 수반하지 않고 main의 왼쪽 자식 노드를 변경한다.
void ChangeLeftSubTree(BTreeNode * main, BTreeNode * sub)
{
    main->left = sub;
}

// 메모리의 소멸을 수반하지 않고 main의 오른쪽 자식 노드를 변경한다.
void ChangeRightSubTree(BTreeNode * main, BTreeNode * sub)
{
    main->right = sub;
}
```

BinaryTree3.c에 정의된 네개의 함수

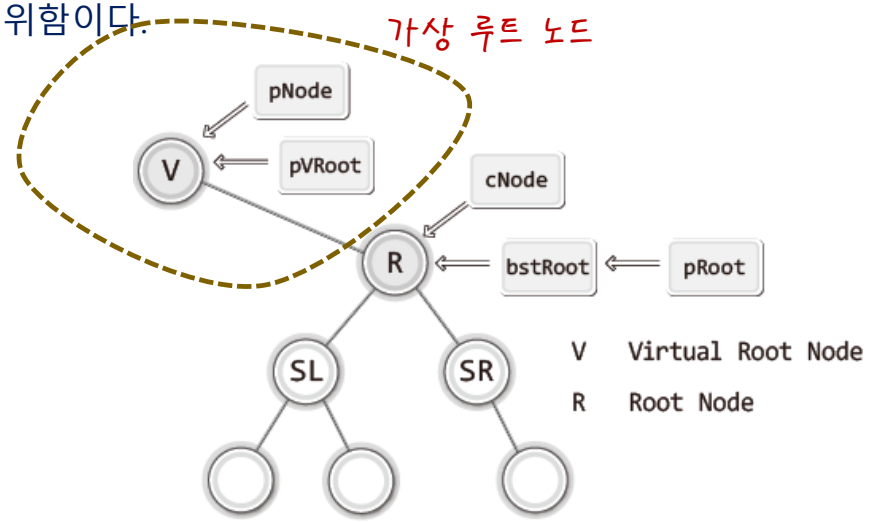
```
// 왼쪽 자식 노드 제거, 제거된 노드의 주소 값이 반환된다.
BTreeNode * RemoveLeftSubTree(BTreeNode * bt)
{
    BTreeNode * delNode;

    if(bt != NULL) {
        delNode = bt->left;
        bt->left = NULL;
    }
    return delNode;
}

// 오른쪽 자식 노드 제거, 제거된 노드의 주소 값이 반환된다.
BTreeNode * RemoveRightSubTree(BTreeNode * bt)
{
    BTreeNode * delNode;

    if(bt != NULL) {
        delNode = bt->right;
        bt->right = NULL;
    }
    return delNode;
}
```

이진 탐색 트리 삭제의 완전한 구현: 초기화

```
BTreeNode * BSTRemove(BTreeNode ** pRoot, BSTData target)
{
    BTreeNode * pVRoot = MakeBTreeNode();
    BTreeNode * pNode = pVRoot;
    BTreeNode * cNode = *pRoot;
    BTreeNode * dNode;

    ChangeRightSubTree(pVRoot, *pRoot);
    . . . .
}
```

가상 루트 노드를 형성한 이유는 삭제할 노드가 루트 노드인 경우의 예외적인 삭제흐름을 일반화하기 위함이다.

```
// bstRoot에 루트 노드의 주소값 저장
BSTRemove(&bstRoot, 7);
```



이진 탐색 트리 삭제의 완전한 구현: 삭제 대상 찾기

```
// 삭제 대상인 노드를 탐색
while(cNode != NULL && GetData(cNode) != target)
{
    pNode = cNode;

    if(target < GetData(cNode))
        cNode = GetLeftSubTree(cNode);
    else
        cNode = GetRightSubTree(cNode);
}

if(cNode == NULL)     // 삭제 대상이 존재하지 않는다면,
    return NULL;

dNode = cNode;        // 삭제 대상을 dNode가 가리키게 한다.
```

cNode의 부모 노드를 pNode가 가리키게 해야 하기 때문에 이 부분을 BSTSearch 함수의 호출로 대신할 수 없다!

여기까지가 실제 삭제의 상황 1 or 상황 2 or 상황 3 의 진행을 위한 준비과정이다!

이진 탐색 트리 삭제의 완전한 구현: 상황 1

앞서 작성한 코드

```
// dNode와 pNode는 각각 삭제할 노드와 이의 부모 노드를 가리키는 포인터 변수
if(삭제할 노드가 단말 노드이다!)
{
    if(GetLeftSubTree(pNode) == dNode)    // 삭제할 노드가 왼쪽 자식 노드라면,
        RemoveLeftSubTree(pNode);         // 왼쪽 자식 노드 트리에서 제거
    else                                  // 삭제할 노드가 오른쪽 자식 노드라면,
        RemoveRightSubTree(pNode);        // 오른쪽 자식 노드 트리에서 제거
}
```

상황 1의 완성된 코드

```
// 첫 번째 경우: 삭제 대상이 단말 노드인 경우
if(GetLeftSubTree(dNode) == NULL && GetRightSubTree(dNode) == NULL)
{
    if(GetLeftSubTree(pNode) == dNode)
        RemoveLeftSubTree(pNode);
    else
        RemoveRightSubTree(pNode);
}
```

이진 탐색 트리 삭제의 완전한 구현: 상황 2

앞서 작성한 코드

```
// dNode와 pNode는 각각 삭제할 노드와 이의 부모 노드를 가리키는 포인터 변수
if(삭제할 노드가 하나의 자식 노드를 지닌다!)
{
    BTreeNode * dcNode;      // 삭제 대상의 자식 노드를 가리키는 포인터 변수

    // 삭제 대상의 자식 노드를 찾는다.
    if(GetLeftSubTree(dNode) != NULL)       // 자식 노드가 왼쪽에 있다면,
        dcNode = GetLeftSubTree(dNode);
    else                                    // 자식
        dcNode = GetRightSubTree(dNode);

    // 삭제 대상의 부모 노드와 자식 노드를 연결한다.
    if(GetLeftSubTree(pNode) == dNode)      // 삭제
        ChangeLeftSubTree(pNode, dcNode);       //
    else                                    // 삭제
        ChangeRightSubTree(pNode, dcNode);      //
}
```

상황 2의 완성된 코드

```
// 두 번째 경우: 삭제 대상이 하나의 자식 노드를 갖는 경우
else if(GetLeftSubTree(dNode) == NULL || GetRightSubTree(dNode) == NULL)
{
    BTreeNode * dcNode;      // 삭제 대상의 자식 노드 가리킴

    if(GetLeftSubTree(dNode) != NULL)
        dcNode = GetLeftSubTree(dNode);
    else
        dcNode = GetRightSubTree(dNode);

    if(GetLeftSubTree(pNode) == dNode)
        ChangeLeftSubTree(pNode, dcNode);
    else
        ChangeRightSubTree(pNode, dcNode);
}
```

이진 탐색 트리 삭제의 완전한 구현: 상황 3

```
// 세 번째 경우: 두 개의 자식 노드를 모두 갖는 경우
else
{
    BTreeNode * mNode = GetRightSubTree(dNode);    // 대체 노드 가리킴
    BTreeNode * mpNode = dNode;      // 대체 노드의 부모 노드 가리킴
    int delData;

    // 삭제 대상의 대체 노드를 찾는다.
    while(GetLeftSubTree(mNode) != NULL) {
        mpNode = mNode;
        mNode = GetLeftSubTree(mNode);
    }

    // 대체 노드에 저장된 값을 삭제할 노드에 대입한다.
    delData = GetData(dNode);
    SetData(dNode, GetData(mNode));

        // 대체 노드의 부모 노드와 자식 노드를 연결한다.
        if(GetLeftSubTree(mpNode) == mNode)
            ChangeLeftSubTree(mpNode, GetRightSubTree(mNode));
        else
            ChangeRightSubTree(mpNode, GetRightSubTree(mNode));

        dNode = mNode;
        SetData(dNode, delData);     // 백업 데이터 복원
    }
```

이어서…

이진 탐색 트리 삭제의 완전한 구현: 마무리

```
    // 삭제된 노드가 루트 노드인 경우에 대한 추가적인 처리
    if(GetRightSubTree(pVRoot) != *pRoot)
        *pRoot = GetRightSubTree(pVRoot);    // 루트 노드의 변경을 반영

    free(pVRoot);      // 가상 루트 노드의 소멸
    return dNode;      // 삭제 대상의 반환
}
```

BinaryTree3.h
BinaryTree3.c
BinarySearchTree2.h
BinarySearchTree2.c
BinarySearchTreeMain.c

실행을 위한 파일의 구성!

요약

- 이진 탐색 트리는 부모의 키 값이 왼쪽 서브트리의 키 값들보다는 크고 오른쪽 서브트리의 키 값들보다는 작다는 조건을 만족한다.
- 이 조건을 이용하면 (균형이 잘 맞을 때) 데이터 삽입, 탐색, 삭제를 O(log N) 시간에 수행할 수 있다.
- 삽입, 탐색의 경우 루트 노드에서부터 현재 값과 트리의 키 값을 비교하여 작으면 왼쪽 크면 오른쪽으로 탐색한다.
- 자식이 두개인 노드를 삭제할 경우 왼쪽 서브트리의 최댓값 또는 오른쪽 서브트리의 최솟값을 가지는 키로 대체한다.

출석 인정을 위한 보고서 작성

- 아래 질문에 대해 답한 후 포털에 제출

- 이진 탐색 트리에서 데이터가 어떤 순서로 삽입이 되는 경우 최악의 시간복잡도를 지닐까? 또, 이 때 삽입/삭제/탐색의 시간복잡도는 얼마일까?