2016047883_정성훈

모든 소스코드에 대해서 일반적인 BST 의 구현은 이론수업에서 사용한 PPT 토대의 구현을 말함.

일반적인 BST 의 구조체는 다음을 의미함.

```
struct TreeNode;
typedef struct TreeNode *SearchTree;
typedef struct TreeNode *Node;

struct TreeNode
{
    int element;
    SearchTree left;
    SearchTree right;
};
```

또한 Find, Insert, Delete 는 다음과 같은 함수 원형을 공통적으로 가짐.

```
Node Find(int x, SearchTree T);
SearchTree Insert(int x, SearchTree T);
SearchTree Delete(int x, SearchTree T);
```

A

RangeSearch 함수를 구현하기 위해 다른 추가적인 함수는 구현하지 않았음.

단 count 를 전역변수로 선언하여 k1 ~ k2 의 범위에 해당하는 key value 를 count 하였음.

함수구현은 다음과 같음

```
void RangeSearch(SearchTree T, int k1, int k2)
{
    if(T!= NULL)
    {
        if(T->element > k1)
            RangeSearch(T->left, k1, k2);

        if(T->element >= k1 && T->element <= k2)
        {
            count++;
        }

        if(T->element < k2)
            RangeSearch(T->right, k1, k2);
    }
}
```

인자로 받은 노드의 element 가 k1 보다 클 경우 그 노드의 왼쪽 서브트리에 대해서 RangeSearch 함수를 호출하여 주었음.

인자로 받은 노드의 element 가 k1 보다 크거나 같고 k2 보다 작거나 같은 경우 조건을 만족하는 key value 이기에 count 를 증가시켜줌.

인자로 받은 노드의 element 가 k2 보다 작을 경우 그 노드의 오른쪽 서브트리에 대해서 RangeSearch 함수를 호출하여 주었음.

Inorder 방식으로 트리를 탐색하는 것과 유사한 방법이지만 k1과 k2 사이의 값에 대해서만 inorder가 일어나는 것과 유사함.

B

RangeSearch(T, k1, k2)의 수행에 필요한 시간이 O(h)가 되도록 한다는 것은, upper bound가 트리의 높이에 걸린다는 의미이고 $O(\log_2(n))$과 같음. 이를 위해서 tree node 구조체에 '자신 + 자식 노드의 개수'를 sub_count로 저장하였음.

구현한 구조체는 다음과 같음.

```
struct TreeNode
{
    int sub_count;
    int element;
    SearchTree left;
    SearchTree right;
};
```

int sub_count를 노드에 기록하여 RangeSearch가 $O(\log_2(n))$의 시간이 걸리게 하였음.

과정은 다음과 같음.

```
int RangeSearch(SearchTree T, int k1, int k2)
{
    int count = 0;
    SearchTree mid;
    mid = findMid(k1, k2, T);
    if ( mid == NULL ) return count;

    int midCount, leftCount, rightCount;
    midCount = 1;
    leftCount = findLeft(0, k1, mid->left);
    rightCount = findRight(0, k2, mid->right);

    count = midCount + leftCount + rightCount;


    return count;
}
```

(1) int RangeSearch(SearchTree T, int k1, int k2) 함수가 호출됨

(2) findMid함수를 호출하여 k1보다 크거나 같고 k2보다 작거나 같으며 가장 먼저 발견되는 노드의 포인터를 mid로 설정함 // O(lgn)

(3) mid의 값이 NULL인 경우 범위를 만족하는 노드가 없다는 의미이므로 0을 리턴함.

(4) 해당 노드 한 개 의미하기 위해 midCount 변수에 1을 저장함. // 상수시간.

(5) int findLeft(int x, int k1, SearchTree T)를 호출하여 mid보다 작으며, k1보다 크거나 같은 노드의 개수를 세어서 leftCount 변수에 리턴값을 저장함 // O(lgn). 재귀적으로 함수가 호출되나 트리의 높이보다 깊게 들어가지 않기 때문..

(6) int findRight(int x, int k2, SearchTree T)를 호출하여 mid보다 크면서, k2보다 작거나 같은 노드의 개수를 세어서 rightCount 변수에 리턴값을 저장함. // O(lgn). findLeft함수와 동일한 이유.

(7) 구해진 midCount, leftCount, rightCount를 모두 더하여 count변수에 저장함. // 상수시간.

(8) count 값을 리턴함

위의 과정에서 수행시간은 상수시간을 제외하고, O(lgn + lgn + lgn) = O(lgn)이기 때문에 RangeSearch함수의 수행에 필요한 시간이 O(h)가 된다.

나머지 구체적인 함수의 구현은 아래와 같다.

1) SearchTree Insert(int x, SearchTree T)

일반적인 BST 구현과 비슷하나, 노드가 삽입될 자리를 찾아가는 과정에서 함수를 재귀적으로 호출하기 전 해당 노드의 sub_count값을 1씩 증가시켜 주어야한다.

sub_count는 '자신 + 자신의 자식노드들의 개수'이기 때문이다.

2) SearchTree Delete(int x, SearchTree T)

Delete 역시 일반적인 BST와 구현이 비슷하나, 삭제될 노드를 찾아가는 과정에서 함수를 재귀적으로 호출하기 전 해당 노드의 sub_count값을 1씩 감소시켜야 한다.

단, 여기서 구현한 함수는 삭제하려는 노드가 없는 경우를 고려하지 않았다.

삭제하려는 노드가 없는 경우라도 sub_count의 값을 1씩 감소시키기 때문에 이 경우 RangeSearch함수가 정상적으로 동작하지 않을 것이다.

이를 방지하려면 delete를 하기전 find함수를 통해 해당 값이 있는지 없는지를 확인해야한다. 이 경우에도 O(lgn)이기 때문에 find를 사용하더라도 O(lgn)의 전체적인 시간에 어긋나지 않는다.

노드 삭제를 위해 값을 교체하면서 노드의 sub_tcount 값을 1씩 감소시키는 것을 제외하고는 일반적인 Delete 함수와 구현 방법이 같다.

3) SearchTree findMid(int k1, int k2, SearchTree T)

인자로 받은 k1과 k2에 대해서, k1보다 크거나 같고 k2보다 작거나 같은 가장 먼저 발견되는 노드의 주소값을 리턴하는 함수이다. 만약 이를 만족하는 노드가 없는 경우 NULL을 리턴함으로써, RangeSearch함수에서 count값을 0으로 설정할 수 있게 한다.

4) int findLeft(int x, int k1, SearchTree T)

```
int findLeft(int x, int k1, SearchTree T)
{
    if ( T == NULL )
        return x;
    else if ( T->element > k1 )
    {
        if ( T->left == NULL )
            return x + T->sub_count;
        return findLeft(x + (T->sub_count - T->left->sub_count), k1, T->left);
    }
    else if(T->element < k1 )
    {
        if ( T->right == NULL )
            return x;
        return findLeft(x, k1, T->right);
    }
    else
    {
        if ( T->left == NULL )
            return x + T->sub_count;
        return x + (T->sub_count - T->left->sub_count);
    }
}
```

인자로 받은 x 값은 범위를 만족하는 노드의 개수이며 k1은 범위를 나타내는 입력값이다. T는 노드의 포인터이며 RangeSearch함수에서 mid가 NULL이 아닌 경우 최초로 mid->left의 포인터값이 인자로 들어온다.

(1) 먼저 T가 NULL인 경우 x 값을 리턴한다. 이미 숫자를 다 세었다고 판단한 것이다.

(2) T의 키값이 k1보다 큰 경우 자신의 왼쪽 서브트리가 없다면 T->sub_count(자신 + 자식의 개수)가 곧 k1보다 큰 키값을 가진 노드의 개수를 의미하게 된다. 따라서 재귀적으로 구해온 x값에 해당 노드의 sub_count 값을 더하여 리턴한다.

(3) T의 키값이 k1보다 크면서 자신의 왼쪽 서브트리가 NULL이 아닌 경우, T->sub_count에서 T->left->sub_count의 값을 빼주면 k1보다 값이 큰 노드의 개수를 의미한다. 따라서 x의 값에 이 값을 더해주고 아직 k1보다 작은 값을 발견하지 않았으므로 다시 함수를 호출한다.

(4) T의 키값이 k1보다 작은 경우 자신의 오른쪽 서브트리가 NULL이면 곧 T->sub_count에 k1보다 큰 값이 없다는 의미이므로 그동안 구해온 x를 리턴해주면 된다.

(5) T의 키값이 k1보다 작으면서 오른쪽 서브트리가 NULL이 아니면 k1보다 큰 값이 오른쪽 서브트리에 존재할 가능성이 남아있기에 함수를 재귀적으로 호출하고 x값은 변경없이 인자로 전달한다(k1보다 크거나 같은 노드의 개수를 여기선 모르기 때문)

(6) T의 키값이 k1과 같으면서 자신의 왼쪽 서브트리가 NULL인 경우는 T->sub_count의 값이 k1보다 큰 노드의 개수를 의미하기 때문에 그동안 구해온 x에 T->sub_count를 더한 값을 리턴한다.

(7) T의 키값이 k1과 같으면서 자신의 왼쪽 서브트리가 NULL이 아니면 T->sub_count의 값에 k1보다 작은 노드의 개수들이 포함되어 있다는 의미이다. 따라서 x에 (T->sub_count – T->left->sub_count)값을 더하여 리턴한다.

5) int findRight(int x, int k2, SearchTree T)

```
int findRight(int x, int k2, SearchTree T)
{
    if ( T == NULL )
        return x;
    else if ( T->element < k2 )
    {
        if ( T->right == NULL )
            return x + T->sub_count;
        return findRight(x + (T->sub_count - T->right->sub_count), k2, T->right);
    }
    else if(T->element > k2 )
    {
        if ( T->left == NULL )
            return x;
        return findRight(x, k2, T->left);
    }
    else
    {
        if ( T->right == NULL )
            return x + T->sub_count;
        return x + (T->sub_count - T->right->sub_count);
    }
}
```

findLeft함수와 원리가 같고 방향을 반대로 설정하여 구현하였다.