2016047883_정성훈

모든 소스코드에 대해서 일반적인 BST, AVL tree 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;
};
```

단, 1-B-c 의 경우 다음과 같음.

```
typedef struct Node* Position;
typedef struct Node* NodePtr;
typedef struct Node
{
    int value;
    NodePtr left, right;
    int height;
}Node;
```

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.

일반적인 BST의 구조체 노드(element, left pointer, right pointer를 가지고 있음)에 **int dup_count** 변수를 추가하여 중복되는 키의 개수를 세어줌.

dup_count 는 자신을 포함하여 1이며 중복되는 키가 삽입될 때마다 1씩 증가.

1) Node Find(int x, SearchTree T);

   일반적인 BST와 똑같이 구현함.

2) SearchTree Insert(int x, SeartchTree T);

   일반적인 BST와 구현은 똑같으나, 같은 키값의 노드를 발견한 경우, 그 노드의 dup_count 값을 1 증가시킴.

3) SearchTree Delete(int x, SeartchTree T);

   일반적인 BST와 구현은 똑같음. 단, (1) 중복키가 있는 경우 와 (2) 중복키가 없는 경우로 나누어서 삭제를 진행함

   (1) 중복키가 있는 경우(dup_count > 1)

   단순히 그 노드의 dup_count를 1 감소시킴.

   (2) 중복키가 없는 경우(dup_count = 1)

   자식이 두 개인 경우 오른쪽 서브트리에서 가장 작은 노드와 자리를 교체한 후 삭제하는 일반적인 BST와 구현이 같음. 이 함수의 재귀적 호출을 통해서 만약 교체될 노드가 중복키를 가지고 있을 경우 노드의 dup_count를 1 감소시키고 가져오는 개념임. 노드를 삭제할 때는 노드의 value만 교환하면 되고 dup_count가 1일 때만 노드가 삭제되기 때문에 dup_count의 값을 재조정해줄 필요는 없음.

B.

a.

일반적으로 BST의 구조체 노드에 key값이 같은 노드를 배열에 저장하기 위해서, 그 배열을 가리키는 **SearchTree *list** 를 추가함. 또 그 배열이 노드를 가지고 있는 개수를 세어 주기 위해서 **int listIdx** 를 추가함.

Find 함수는 A와 같기 때문에 설명을 생략함.

1) SearchTree Insert(int x, SeartchTree T)

   A의 Insert와 똑같이 동작하나, 중복키를 발견하였을 때 LinkDupKey라는 함수를 호출함.

2) SearchTree LinkDupKey(int x, SearchTree T);

   인자로 중복키값과, 중복키값을 가진 노드의 포인터(T)를 받음.

   T->list == NULL 일 경우, 중복키의 노드를 저장하기 위한 배열을 동적할당 받음. 이때 동적할당하는 배열의 크기는 과제에 명시되어 있는 100임. 그리고 T->listIdx 값을 0으로 초기화 해줌.

   T->listIdx 가 100 보다 작다면, 즉 중복키를 100 개 미만으로 가지고 있다면 그 배열의 끝자리(T->listIdx 로 알 수 있음)에 노드를 하나 생성해서 저장한 뒤, 그 노드가 배열에 있다는 것을 알 수 있게 생성된 배열의 listIdx 값을 같게 만들어 줌.

   T->listIdx 가 100 보다 크거나 같다면 중복키를 100 개 가지고 있다는 의미이며 더 이상 중복키를 저장할 수 없음. 중복키를 저장할 수 없다는 출력문구를 출력해줌

3) SearchTree Delete(int x, SeartchTree T)

   일반적인 BST의 Delete와 구현은 같으나, 삭제할 키의 노드를 발견한 경우, 그 노드의 listIdx 값에 따라 나뉨.

   (1) listIdx 값이 0보다 큰 경우

   중복키를 가지고 있다는 의미이기에 listIdx를 통해서 맨끝 노드를 삭제해주고 listIdx를 1 감소시킴.

   (2) listIdx 값이 0인 경우

   중복키를 가지고 있지 않다는 의미임. 일반적인 BST delete 처럼 자식의 개수에 따라서 delete 가 이루어지고, 재귀적호출을 통해 중복노드가 있는 경우 값의 교환을 통해 끌어올림.

+ listIdx 가 배열의 끝을 가리키고 있게 하게 만들다보니, 실제로 출력시에는 개수를 나타내기 위해서 원래 값에 +1 을 해서 출력해야함.

b.

Insert, Delete 를 제외하고는 일반적인 BST 구현과 동일하기에 Insert, Delete 에 대해서만 설명을 하겠음.

1) SearchTree Insert(int x, SearchTree T);

   일반적인 BST 의 구현과 같으나, 중복키를 발견한 경우 과제에 명시된 대로 왼쪽 서브트리에 노드를 삽입함.

2) SearchTree Delete(int x, SearchTree T);

   일반적인 BST 의 구현과 같으나, 삭제할 노드를 발견한 경우 다음과 같이 나뉨.

   (1) 삭제할 노드의 왼쪽 자식이 중복키를 가진 노드인지 확인하고, 맞을 경우 그 노드에 대한 deletion 이 일어남.

   (2) 그렇지 않은 경우 일반적인 BST 의 노드 삭제와 동일함.

c.

위의 b 와 같이 중복키가 insert 될 때마다 left subtree 에 insert 할 경우 한쪽으로 skew 되어 height 가 커질 수 있음. 따라서 이를 보완하기 위해 AVL tree 를 이용하였음.

Find 함수는 동일하기에 설명을 생략하고 Insert, Delete 함수와 이를 위해 추가적으로 구현한 함수에 대해서만 설명을 하겠음.

1) NodePtr Insert(int X, NodePtr T)

   AVL tree 의 원리에 따라 삽입 후 해당 노드의 balance 가 맞는지 확인하고(양쪽 서브트리의 height 의 차이가 1 이하로 유지되어야함) 아닐 경우 어느 위치에 삽입되었는지에 따라 singleRotation, doubleRotation 이 나뉨. 중복키를 삽입하는 경우를 제외하고는 일반적인 AVL tree 와 같음.

   중복키가 삽입되는 경우, 삽입 후 밸런스를 체크하는데 과제에 명시된 대로 중복키는 왼쪽 서브트리에만 삽입이 일어남. 즉 원래 AVL tree 에서 왼쪽 서브트리에 삽입될 경우와 똑같이 balance 를 체크하고 rotation 을 진행하면 됨.

2) NodePtr Delete(int x, NodePtr T)

Delete 는 일반적인 BST 의 delete 와 동일하나, delete 를 한 뒤 매번 그 노드에 대해서 Balance 를 체크해주고 맞지 않는 경우 Rotation 을 통해 balance 를 맞춰주어야 함. 이를 위해 Delete 함수의 끝부분에 다음과 같은 부분을 구현하였음.

```
    ...
    ...
    if(T == NULL)
        return T;

    T->height = MAX(height(T->left), height(T->right)) + 1;
    int balance = FindBalance(T);
    if(balance > 1)
    {
        if(FindBalance(T->right) >= 0)
            return singleRotateWithRight(T);
        else
            return doubleRotateWithRight(T);
    }
    if(balance < -1)
    {
        if(FindBalance(T->left) <= 0)
            return singleRotateWithLeft(T);
        else
            return doubleRotateWithLeft(T);
    }

    return T;
}
```

(함수의 윗부분에서 T 가 NULL 인 경우 따로 return 을 안 해주었기 때문에, NULL 인 경우 return 을 해줌)

먼저 삭제가 일어난 노드의 높이를 재조정해줘야함. 그래서 해당 노드의 왼쪽 서브트리의 높이와 오른쪽 서브트리의 높이 중 큰 값을 대입해줌.

그 뒤에, FindBalance 함수를 통해 그 노드의 balance 값을 구해줌. 이 함수의 리턴 값은 '인자로 들어온 노드의 오른쪽 서브트리의 높이 – 왼쪽 서브트리의 높이'임. 이 값이 1 보다 큰 경우 delete 이후 오른쪽 서브트리의 높이가 커서 밸런스가 맞지 않는다는 의미이며 값이 -1 보다 작은 경우 왼쪽 서브트리의 높이가 커서 밸런스가 맞지 않는다는 의미임. 따라서 두 경우로 나누어서 밸런스를 조정해줘야 함.

(1) 오른쪽 서브트리의 높이가 긴 경우

사실 삽입 시와 동일하게 두 경우로 나누어서 rotate 를 해주면 됨. 이때 오른쪽 서브트리의 balance 값을 FindBalance 함수로 구해봄으로써, 이 값이 0 보다 크거나 같은 경우 삽입시에 '오른쪽 자식의 오른쪽 서브트리에 삽입이 일어남'과 같기에 singleRoatateWithRight 함수를 사용함.

반대로 오른쪽 서브트리의 balance 값이 0 보다 작은 경우 삽입시에 '오른쪽 자식의 왼쪽 서브트리에 삽입이 일어남'과 같기에 doubleRoatateWithRight 함수를 사용함.

(2) 왼쪽 서브트리의 높이가 긴 경우

오른쪽 서브트리의 높이가 긴 경우와 같음. 단, 여기선 왼쪽 서브트리의 balance 값이 0 보다 작거나 같은 경우 삽입시에 '왼쪽 자식의 왼쪽 서브트리에 대해서 삽입이 일어남'과 같기에 singleRotateWithLeft 함수를 사용하였고, 0 보다 큰 경우 삽입시에 '왼쪽 자식의 오른쪽 서브트리에 삽입이 일어남'과 같은 경우기에 doubleRoatateWithLeft 함수를 사용함.

Delete 함수의 재귀적 호출을 통해, 삭제가 일어난 노드부터 위로 올라가며 트리의 Balance 조정이 일어남.

따라서 왼쪽에만 중복키를 삽입하더라도 트리가 한쪽으로 skew 되어 height 가 커지는 것을 방지할 수 있음.

3) int FindBalance(NodePtr T)

인자로 받은 노드에 대해서, '오른쪽 서브트리의 높이 – 왼쪽 서브트리의 높이'값을 리턴하는 함수임. 이때 서브트리를 가리키는 포인터 값이 NULL 인 경우(즉 없다는 의미) 높이를 -1 로 설정하여 계산함.