알고리즘 실습(6주차)

201000287 일어일문학과 유다훈

1. 과제 설명 및 해결 방법

1. Bineary Search Tree 구현

① Tree Insert를 이용하여 주어진 배열에서 하나 씩 원소 삽입

② 주어진 배열을 일단 정렬한 후 중간 값을 이용하여 Tree Insert를 수행한다. 이 것을 재귀적으로 반복한다.

2. Tree Delete

주어진 트리에서 어떠한 값을 가진 노드를 찾아 삭제한다.

2. 주요 부분 코드 설명(알고리즘 부분)

1. Tree Insert

- 정렬되지 않은 데이터를 그냥 삽입하기.

```
void tree_insert(node* head, int z){ //tree insert

    struct node *insertNode = (struct node *) malloc(sizeof(struct node));
    insertNode->value = z;
    insertNode->parent = NULL;
    insertNode->left = NULL;
    insertNode->right = NULL;

    if (head->value < z) {
        if(head->right != NULL) {
            tree_insert(head->right, z);
        } else {
            insertNode->parent = head;
            head->right = insertNode;
        }
    } else if (head->value > z) {
        if (head->left != NULL) {
            tree_insert(head->left, z);
        } else {
            insertNode->parent = head;
            head->left = insertNode;
        }
    }
}
```

- 구조체 Node를 이용하여 연결리스트를 생성하여 트리에 삽입해준다.
- 삽입하는 값의 부모, 왼쪽, 오른쪽의 값을 null로 지정해준다.
- head의 값이 삽입될 값보다 크다면, head의 오른쪽에 재귀적으로 삽입을 실행한다.
- 오른쪽의 노드가 null값이라면 삽입되는 노드의 부모 값을 현재 head 값으로 지정해주고 head값의 오른쪽에 삽입을 해준다.
- 왼쪽도 오른쪽과 동일하게 해주며 부등호만 다르게 해준다.

- 데이터를 한 번 정렬한 후 정렬 된 값의 중간값을 이용하여 재귀적으로 트리 생성하기.

```
/* 배열 정렬을 위한 퀵소트*/
void swap(int array[], int i, int j) {
    int temp = array[i];
    array[i] = array[j];
    array[j] = temp;
}

int partition(int array[], int pivot, int right) {
    int x = array[right];
    int i = pivot-1;

    for (int j = pivot; j <= right-1 ; j++) {
        if (array[j] < x ) {
            i++;
            swap(array, i, j);
        }
    }
    swap(array, i+1, right);
    return i+1;
}

void quicksort(int array[], int pivot, int right){
    if (pivot < right) {
        int q = partition(array, pivot, right);
        quicksort(array, pivot, q-1);
        quicksort(array, q+1, right);

    }
}
/* 배열 정렬을 위한 퀵소트*/
```

  - 데이터값을 정렬하여 중간값을 이용하여 트리에 삽입하는 작업은 우선 정렬되지 않은 데이터를 퀵정렬을 이용하여 오름차순으로 정렬해 주었다.

```
void recursively_tree_insert (int array[], int start ,int arrayLength, node *head) {

    if (start < arrayLength) {
        int last = arrayLength;

        int mid = (last+start) /2 ;
         tree_insert(head, array[mid]);

        recursively_tree_insert(array, start, mid-1, head);
        recursively_tree_insert(array, mid+1, arrayLength, head);
    }
}
```

- 이후 데이터의 중간 인덱스값을 찾은 후 해당 인덱스 값을 트리에 삽입해가며 재귀적으로 트리를 생성한다.

2. Tree Delete

- Transplant

```
void transplant(node *head, node *deleteNode, node *deleteNodeChildren) { //트리의 노드 바꿔치는 작업
    if (deleteNode->parent == NULL) {
        head = deleteNodeChildren;
    } else if ( deleteNode == deleteNode->parent->left) {
        deleteNode->parent->left = deleteNodeChildren;
    } else {
        deleteNode->parent->right = deleteNodeChildren;
    }
    if (deleteNodeChildren != NULL) {
        deleteNodeChildren->parent = deleteNode->parent;
    }
//    free(deleteNode);
}
```

  - 트리에서 어떤 노드를 삭제하기 위한 작업을 하는 함수.
  - 삭제하고자 하는 노드의 부모와 삭제하고자 하는 노드의 자식을 연결하고 삭제하고자 하는 노드를 빼버리는 작업.

- Tree Delete

```
void tree_delete(node *head, int deleteValue) { //트리에서 노드 삭제작업

    node* delNode = tree_search(head, deleteValue); //삭제할 값을 가진 노드 찾기

    if (delNode->left == NULL) { //오른쪽노드만 가진 노드
        transplant(head, delNode, delNode->right);

    } else if (delNode->right == NULL) { //왼쪽 노드를 가진 노드
        transplant(head, delNode, delNode->left);
    } else { //둘다 가진 노드 삭제?
        node* y = tree_minimum(delNode->right);
        if (y->parent != delNode) {
            transplant(head, y, y->right);
            y->right = delNode->right;
            y->right->parent = y;
        }
        transplant(head, delNode, y);
        y->left = delNode->left;
        y->left->parent = y;
    }
    delete delNode;
}
```

- 먼저 삭제하고자 하는 값이 들어있는 노드를 찾는다.
- 삭제하고자 하는 노드의 자식이 한쪽만 있는 자식이라면, transplant 함수를 이용하여 Recursive하게 찾는다.
- 자식을 양 쪽 다 가지고 있는 노드라면, 일단 삭제할 노드의 오른쪽 자식에서 최소값을 찾는다(Successor)
- 그 후 successor의 부모가 삭제할 노드가 아니라면, transplant를 이용하여 자리를 바꿔주는 작업을 한다.
- successor의 부모가 삭제할 노드라면, successor와 삭제할 노드를 transplant 작업을 진행해준다.
- successor의 왼쪽 자식을 지우고자하는 노드의 왼쪽으로 지정하고, successor의 왼쪽 자식의 부모를 successor로 지정해준다.
- 마지막으로 삭제하고자 하는 노드의 메모리를 삭제해준다.

3. 결과

- Tree insert
  - 정렬되지 않은 데이터를 이용하여 정렬을 할 경우 트리 모양은 뒤틀린 형태로 생성되게 되며 운이 나쁠 경우 O(n)의 시간이 걸리게 된다.
  - 재귀적인 방법으로 중간 값부터 삽입해가며 트리를 생성할 경우 트리 생성 시간은 O(log n)이 시간이 걸린다.
  - 삽입 시간의 경우 평균 O(log n)의 시간이 걸린다.
- Tree delete
  - C++의 경우 메모리 관리가 힘들어 구현하면서 디버깅하는 도중 메모리 꼬임 현상이 많이 발생하여 구현하기가 힘들었다.
  - 자식노드가 아예 없는 경우, 한 쪽만 있는 경우, 둘 다 있는 경우를 제각각 고려해야하므로 번거롭고 구현이 까다롭다.
- Inorder-Tree-walk
  - 이번 과제에서 결과물을 확인하기 위해 inorder traverse를 재귀적으로 실행하는 함수를 작성하여 출력하여 확인하였다.

```
void inorder_tree_walk(node* head) { //inorder traverse
    if (head != NULL) {
        inorder_tree_walk(head->left);
        printf("%d\n", head->value);
        inorder_tree_walk(head->right);
    }
}
```