자료구조 과제 11

* –

20151523

김동현

- 실행결과

그리기이(가) 표시된 사진

자동 생성된 설명

- 코드 및 알고리즘

#define \_CRT\_SECURE\_NO\_WARNINGS

#define MAX\_TERMS 100

#define IS\_FULL(ptr) (!(ptr))

#include <stdlib.h>

#include <stdio.h>

typedef struct treeNode\* treePtr;

typedef struct treeNode {

treePtr lchild;

int data;

treePtr rchild;

}treeNode;

void inorder(treePtr);

void preorder(treePtr);

void insert(treePtr\*, int);

treePtr modifiedSearch(treePtr, int);

treePtr modifiedSearch2(treePtr ptr, int key);

treePtr search\_sublmax(treePtr ptr);

treePtr search\_subrmin(treePtr ptr);

treePtr search\_parent(treePtr root, int key, int \*dir);

void delete(treePtr\*, int);

void rangeSearch(treePtr, int, int);

void main()

{

int i, j, n, m, A[MAX\_TERMS], D[MAX\_TERMS], k1, k2;

treePtr tree = NULL;

// 입력으로 들어오는 노드의 개수는 n에, 지울 값의 개수는 m에, 삽입하여야할

//정수들은 A에, 삭제할 정수들은 D에, 비교할 대상인 정수는 k1, k2 저장

// 순서대로 binary search tree에 삽입

FILE\* fp = fopen("input2.txt", "r");

fscanf(fp, "%d", &n);

for (i = 0; i < n; i++)

fscanf(fp, "%d", &A[i]);

fscanf(fp, "%d", &m);

for (i = 0; i < m; i++)

fscanf(fp, "%d", &D[i]);

fscanf(fp, "%d", &k1);

fscanf(fp, "%d", &k2);

for (i = 0; i < n; i++)

insert(&tree, A[i]);

inorder(tree);

printf("\n");

preorder(tree);

printf("\n");

// 구성된 binary search tree를 inorder와 preorder로 출력하여 올바르게 트리가

//생성되었는지 확인

// delete할 정수들을 구성된 binary search tree에서 삭제한 후, 올바르게 삭제되

// 었는지 확인하기 위해 inorder와 preorder로 출력

for (i = 0; i < m; i++)

delete(&tree, D[i]);

inorder(tree);

printf("\n");

preorder(tree);

printf("\n");

rangeSearch(tree, k1, k2);

}

**: insert, inorder, preorder, modifiedSearch의 코드는 생략**

treePtr modifiedSearch2(treePtr ptr, int key)

{

if (!ptr) return NULL;

if (key == ptr->data) return ptr;

if (key < ptr->data)

return modifiedSearch2(ptr->lchild, key);

return modifiedSearch2(ptr->rchild, key);

}

: 찾는 key를 data로 갖고있는 노드를 반환한다. 없으면 NULL을 반환.

void node\_count(treePtr ptr, int\* cnt) {

if (ptr) {

node\_count(ptr->lchild, cnt);

(\*cnt)++;

node\_count(ptr->rchild, cnt);

}

}

: 해당 노드를 root로 하는 트리의 노드 개수를 구하는 함수이다. Cnt를 포인터로 받아서 전달해준다. 나중에 2 child node를 삭제할 때 왼쪽 서브트리와 오른쪽 서브트리의 노드 개수를 비교하기위해 만들었다.

treePtr search\_sublmax(treePtr ptr)

{

treePtr temp;

if (!ptr) return NULL;

ptr = ptr->lchild;

while (ptr->rchild)

ptr = ptr->rchild;

return ptr;

}

: 왼쪽 서브트리에서 가장 큰 노드를 찾는 함수이다. 오른쪽 자식을 반복해서 null이 나올때까지 타고가면 가장 큰 노드를 얻을 수 있다.

treePtr search\_subrmin(treePtr ptr)

{

treePtr temp;

if (!ptr) return NULL;

ptr = ptr->rchild;

while (ptr->lchild)

ptr = ptr->lchild;

return ptr;

}

: 오른쪽 서브트리에서 가장 작 노드를 찾는 함수이다. 왼쪽 자식을 반복해서 null이 나올때까지 타고가면 가장 작은 노드를 얻을 수 있다.

treePtr search\_parent(treePtr root, int key, int \*dir)

{

treePtr temp;

\*dir = 0;

if (key == root->data) return NULL;

while (root) {

if (key < root->data) {

if (key == root->lchild->data) {

\*dir = -1;

return root;

}

root = root->lchild;

}

else {

if (key == root->rchild->data) {

\*dir = 1;

return root;

}

root = root->rchild;

}

}

return NULL;

}

: 부모 노드를 찾는 함수이다. 삭제 후, 부모와 삭제노드의 자식노드를 이어줘야하기 때문에 필요해서 작성하였다. Dir로 자식노드가 부모노드의 왼쪽인지 오른쪽인지 구별하게끔 하였다.

void delete(treePtr\* node, int num) {

treePtr root, cnode, pnode = NULL, ptr, tlnode, ptlnode, temp;

int dir, lcount=0, rcount=0;

: cnode : 삭제할 자식노드, pnode : 부모노드 , tlnode : 삭제할 노드의 후손중 가장 큰(작은) 노드, ptlnode : tlnode의 부모노드. 해당 함수에 대한설명은 주석으로 설명하겠다.

root = \*node;

cnode = modifiedSearch2(root, num);

if (cnode) {

if (cnode->lchild == NULL && cnode->rchild == NULL) { // case 1 : leaf node

pnode = search\_parent(root, cnode->data, &dir);

if (dir == -1) // 1: right , -1 : left

pnode->lchild = NULL;

else if (dir == 1)

pnode->rchild = NULL;

free(cnode);

if (pnode == NULL) \*node = cnode;

}

else if (cnode->lchild == NULL || cnode->rchild == NULL) { // case 2 : non-leaf node with single child

pnode = search\_parent(root, cnode->data, &dir);

if (pnode == NULL) { // cnode가 root node인 경우

if (cnode->lchild != NULL)

\*node = cnode->lchild;

else

\*node = cnode->rchild;

free(cnode);

}

else {

if (dir == -1) {

if (cnode->lchild != NULL)//부모 노드의 왼쪽 자식 = 자식 노드의 왼쪽 자식

pnode->lchild = cnode->lchild;

else

pnode->lchild = cnode->rchild;//부모 노드의 왼쪽 자식 = 자식 노드의 오른쪽 자식

}

else {

if (cnode->lchild != NULL) //부모 노드의 오른쪽 자식 = 자식 노드의 왼쪽 자식

pnode->rchild = cnode->lchild;

else

pnode->rchild = cnode->rchild;//부모 노드의 오른쪽 자식 = 자식 노드의 오른쪽 자식

}

free(cnode);

}

}

else { // case 3 : non-leaf node with double child

node\_count(cnode->lchild, &lcount);

node\_count(cnode->rchild, &rcount);

if (lcount >= rcount) { //왼쪽 서브트리의 노드 개수가 더 많거나 같은 경우

tlnode = search\_sublmax(cnode); //왼쪽 서브트리에서 가장 큰 노드

ptlnode = search\_parent(root, tlnode->data, &dir);

pnode = search\_parent(root, cnode->data, &dir);

if (tlnode != cnode->lchild) {

// 바로 부모자식 관계가 아닐 경우 이어주는 기능

ptlnode->rchild = tlnode->lchild;

tlnode->lchild = cnode->lchild;

}

tlnode->rchild = cnode->rchild;

if (pnode == NULL)//삭제할 노드가 root인 경우

\*node = tlnode;

else {//root가 아니라면 부모노드에 연결

if (dir == -1)

pnode->lchild = tlnode;

else

pnode->rchild = tlnode;

}

free(cnode);

}

else {{ //오른쪽 서브트리의 노드 개수가 더 많은 경우

tlnode = search\_subrmin(cnode);

ptlnode = search\_parent(root, tlnode->data, &dir);

pnode = search\_parent(root, cnode->data, &dir);

if (tlnode != cnode->rchild) {

ptlnode->lchild = tlnode->lchild;

tlnode->lchild = cnode->lchild;

}

tlnode->rchild = cnode->rchild;

if (pnode == NULL)

\*node = tlnode;

else {

if (dir == -1)

pnode->lchild = tlnode;

else

pnode->rchild = tlnode;

}

free(cnode);

}

}

}

else //삭제할 num이 트리에 없는 경우

printf("%d 가 트리에 존재하지 않습니다.\n", num);

}

void rangeSearch(treePtr ptr, int k1, int k2) {

if (ptr) {

rangeSearch(ptr->lchild, k1, k2);

if ((ptr->data >= k1) && (ptr->data <= k2))

printf("%d ", ptr->data);

rangeSearch(ptr->rchild, k1, k2);

}

}

: 범위를 토대로 해당 범위의 data값만 출력하는 함수이다. 단순히 해당 범위에 포함 되어야지만 printf를 하도록 하였다.