강의명: 자료구조

숙제 번호: 6

숙제 제목: Trees(트리)

학생 이름: 김성현

학번: 201910783

1. Tree traverse(트리 순회)

1.1

// ======================================================================

void init\_tree(TREE \* t)

{

int i;

for (i = 0; i < MAX\_TREE\_SIZE; i++) {

t->data[i] = -1;

}

}

// ======================================================================

void preorder\_tree(TREE \* t, int root)

{

int left, right; //root, left, right is index.

if ((t->data[root] == -1) || (root >= MAX\_TREE\_SIZE))

return;

left = root \* 2;

right = root \* 2 + 1;

printf("(%d)", t->data[root]);

preorder\_tree(t, left);

preorder\_tree(t, right);

}

// ======================================================================

void inorder\_tree(TREE \* t, int root)

{

int left, right;

if ((t->data[root] == -1) || (root >= MAX\_TREE\_SIZE))

return;

left = root \* 2;

right = root \* 2 + 1;

inorder\_tree(t, left);

printf("(%d)", t->data[root]);

inorder\_tree(t, right);

}

// ======================================================================

void postorder\_tree(TREE \* t, int root)

{

int left, right;

if ((t->data[root] == -1) || (root >= MAX\_TREE\_SIZE))

return;

left = root \* 2;

right = root \* 2 + 1;

postorder\_tree(t, left);

postorder\_tree(t, right);

printf("(%d)", t->data[root]);

}

// ======================================================================

\*이렇게 프로그램한 이유:

-먼저 void init\_tree(TREE \* t)함수는 tree t를 초기화(모든 node 위치에 -1 저장)하는 함수입니다. 그리고 tree t의 data는 array형태로 구현되어 있고, 초기화 상태는 node위치에 -1이 저장되는 것이기 때문에 아주 간단하게 for문을 tree크기의 최대까지 돌려 이걸 index로 하여 각 data위치에 접근, 각 위치에 -1을 저장하는 루프를 차근차근 반복하게 하여 하나씩 -1로 저장하여 결국 전부 다 -1로 저장되게 하였습니다.

-다음 void preorder\_tree(TREE \* t, int root)함수, void inorder\_tree(TREE \* t, int root)함수, void postorder\_tree(TREE \* t, int root)함수들은 preorder, inorder, postorder로 tree t를 traverse하여 출력하는 함수입니다. traversal함수이고 무엇보다 그냥 출력하면 되는 함수 이기에 preorder, inorder, postorder별로 root출력 + 왼쪽 트리 순회 + 오른쪽 트리 순회 이 3개 부분의 순서만 바꿔주면서 만들어주면 되는 것이기 때문에 일단 공통적인 부분을 먼저 각 함수마다 작성해주었습니다. 이 공통적인 부분은 먼저 왼쪽과 오른쪽으로 갈 수 있도록 그 index를 저장할 수 있는 변수 left, right를 선언하여 left에는 왼쪽 자식 트리를 지칭할 수 있게 root \* 2, right에는 오른쪽 자식 트리를 저장할 수 있게 root \* 2 + 1을 저장하였습니다. 그 후 만약 root인 부분이 비어있거나, TREE의 최대치 사이즈를 넘어선 root일 경우 그 트리는 이도저도 아닌 놈이기 때문에 그냥 바로 함수를 끝낼 수 있게 if문에 조건으로 넣어주면서 바로 return해버리도록 하였습니다. 딱 여기까지가 공통적인 부분이고 이 다음부터는 말했던 것처럼 각 함수별로 root-왼쪽순회-오른쪽순회 3부분의 순서를 다르게 해서 작성하였습니다. preorder함수는 전위순회임으로 root가 가장 먼저 출력되고 그 다음 왼쪽, 오른쪽 순서로 나아가야하기 때문에 root 출력->왼쪽순회 순환->오른쪽순회 순환의 형식으로 만들어 특히 다시 순환의 형식으로 만들어 다시 새로운 기준으로 또 순환해서 시작해 쭉 트리를 순회할 수 있도록 만들었습니다. 이와 같은 맥락으로 다른 두 함수도 각각 inorder, postorder순회의 형태로 각각 왼쪽순회 순환-root 출력-오른쪽순회 순환의 순서와 왼쪽순회 순환-오른쪽순회 순환-root 출력의 순서로 구성하여 코딩을 하였습니다.

1.2

// ======================================================================

int main(void)

{

int i;

TREE t1;

printf("===== test-tree1 =====\n");

init\_tree(&t1);

for (i = 1; i < MAX\_TREE\_SIZE - 3; i++)

t1.data[i] = 10 + i;

printf("preorder=");

preorder\_tree(&t1, 1);

printf("\n");

printf("inorder=");

inorder\_tree(&t1, 1);

printf("\n");

printf("postorder=");

postorder\_tree(&t1, 1);

printf("\n");

return EXIT\_SUCCESS;

}

// ======================================================================

\*함수의 수행 작업 설명 with tree:

-tree1.c를 test하는 파일인 test-tree1.c는 이를 위한 main함수가 코딩되어 있습니다. 먼저 테스트를 위해 새로운 트리인 t1을 선언한 것을 볼 수 있습니다. 그리고 그 이후 init\_tree를 통해 tree함수를 초기화하여 -1을 싹 다 넣으셨습니다. 그 다음 for문 루프를 통해 11부터 22까지 총 12개의 숫자를 트리에 넣어 구성하는 것을 볼 수 있습니다. 그렇게 구성된 트리는 아래 그림과 같습니다. (초기화된 트리 노드는 제외)

텍스트이(가) 표시된 사진

자동 생성된 설명

이렇게 구성된 트리를 가지고 main함수에서는 그 이후 preorder, inorder, postorder순회를 하여 그 함수들이 제대로 작성되었는지를 확인하고 있습니다. 따라서 각 함수가 수행되면 traversal, 순회가 일어나게 되며 전위, 중위, 후위에 따라 위 그림을 가지고 설명하자면 다음 그림들과 같게 행동이 실행되어 그 순서대로 출력되게 됩니다.

텍스트이(가) 표시된 사진

자동 생성된 설명<preoder, 전위순회>

11

10

11

6

9

1

2

3

8

7

5

4

텍스트이(가) 표시된 사진

자동 생성된 설명<inorder, 중위순회>

11

3

5

7

6

12

10

8

4

2

9

1

텍스트이(가) 표시된 사진

자동 생성된 설명<postorder, 후위순회>

12

11

2

4

5

6

10

9

7

3

8

1

이렇게 3가지의 순회를 체크한 뒤 main함수는 EXIT\_SUCCESS를 return하고 함수를 종료하여 마무리 짓습니다.

1.3

s1910783@oak:hw06$ gcc tree1.c test-tree1.c -o test-tree1

s1910783@oak:hw06$ ./test-tree1

===== test-tree1 =====

preorder=(11)(12)(14)(18)(19)(15)(20)(21)(13)(16)(22)(17)

inorder=(18)(14)(19)(12)(20)(15)(21)(11)(22)(16)(13)(17)

postorder=(18)(19)(14)(20)(21)(15)(12)(22)(16)(17)(13)(11)

2. Tree add child(트리 자식 추가)

2.1

// ======================================================================

NODE \*create\_node(int item)

{

NODE \* node;

node = (NODE \*)malloc(sizeof(NODE));

if (node == NULL) return NULL;

node->data = item;

node->left = NULL;

node->right = NULL;

return node;

}

// ======================================================================

void add\_child(NODE \* root, NODE \* left, NODE \* right)

{

root->left = left;

root->right = right;

}

// ======================================================================

int sum\_tree(NODE \* root)

{

int total = 0; //at the first, it should be 0. ha ha ha.

if (root != NULL) {

total = root->data + sum\_tree(root->left) + sum\_tree(root->right);

}

return total;

}

// ======================================================================

\*이렇게 프로그램한 이유:

-먼저 NODE \*create\_node(int item)함수는 하나의 tree node를 create하는 함수로서 일단 가장 기초적인 메모리 공간을 확보해 node를 실질적으로 만드는 malloc생성을 위해 node를 NODE의 포인터 타입으로 선언하여 NODE type의 크기만큼 메모리 공간을 node에 부여하도록 malloc을 사용하였습니다. 그 다음 혹시 모를 사태로 메모리만 띡 줘놓고 아무런 item을 넣지 않고 그냥 끝내버리면 그 node는 이제 쓸 수 없는 아이이니 if문으로 node가 NULL일 경우 NULL을 return하도록 하였습니다. 그 다음은 이제 정상적으로 하나의 노드를 만들어야 하니 data부분에 item을 넣고, 양 옆 left, right 포인터에는 NULL값을 넣어 아직 연결이 안된 독립적인 상태의 노드를 하나 만들게 하는 걸로 함수를 끝내고 node 를 return하였습니다.

-그 다음 void add\_child(NODE \* root, NODE \* left, NODE \* right)함수는 root에 left를 left subtree로, right를 right subtree로 연결시키는, 이른바 트리로 만들기 위해 각 노드들을 연결시키는 동작을 하는 함수입니다. 이를 위해 아주 간단하지만 중요한 것이 바로 노드의 left, right의 포인터에 각각 left와 right를 넣는 것입니다. 이렇게 하면 그 노드의 왼쪽에는 left가 연결되고, 오른쪽에는 right가 연결되니 간단하지만 아주 중요한 것입니다. 구현은 간단하게 root의 left에 left를, root의 right에 right를 넣어서 제가 말한 내용을 작동하도록 하였습니다.

-마지막으로 int sum\_tree(NODE \* root)함수는 root가 형성하는 tree의 모든 node에 저장된 자연수를 합한 값을 return해야하는 함수입니다. 이를 위해서 제가 택한 방법은 바로 좌우 서브트리에 대한 순환호출(recursive call)을 하여 각 data의 값들을 더하는 방식입니다. 이를 위해 우선 더한 값들을 저장할 수 있는 변수 total를 선언하여 꼭! 0으로 초기화하였습니다. 그 다음 if문을 통해 root가 NULL이 아닌, 즉 그 node가 존재할 경우만 다뤄서 total에 root의 data값을 더하고 그 다음 left와 right를 sum\_tree로 순환 호출하여 쭉 또 새로운 각각의 기준으로 쭉 더해 나가서 나온 총 합계를 return해 주었습니다.

2.2

// ======================================================================

int main(void)

{

int sum;

NODE \*t0, \*t1, \*t2, \*t3, \*t4, \*t5, \*t6, \*t7, \*t8, \*t9, \*t10, \*t11;

printf("===== test-tree2 =====\n");

t0 = create\_node(0);

t1 = create\_node(10);

t2 = create\_node(20);

t3 = create\_node(30);

t4 = create\_node(40);

t5 = create\_node(50);

t6 = create\_node(60);

t7 = create\_node(70);

t8 = create\_node(80);

t9 = create\_node(90);

t10 = create\_node(100);

t11 = create\_node(110);

add\_child(t3, t7, t8);

add\_child(t4, t9, t10);

add\_child(t5, t11, NULL);

add\_child(t1, t3, t4);

add\_child(t2, t5, t6);

add\_child(t0, t1, t2);

sum = sum\_tree(t0);

printf("sum\_tree(t0)=%d\n", sum);

sum = sum\_tree(t1);

printf("sum\_tree(t1)=%d\n", sum);

sum = sum\_tree(t2);

printf("sum\_tree(t2)=%d\n", sum);

sum = sum\_tree(t4);

printf("sum\_tree(t4)=%d\n", sum);

sum = sum\_tree(t5);

printf("sum\_tree(t5)=%d\n", sum);

return EXIT\_SUCCESS;

}

// ======================================================================

\*어떻게 이게 작동되는가 with tree:

-먼저 트리를 구성하는 노드 12개를 NODE pointer타입으로 일단 독립적으로 선언하는 것을 처음에 볼 수 있다. 그 다음 t0부터 시작해서 t11까지 0~110까지 10단위로 노드에 값을 넣어 node에 대한 생성을 완료하는 것을 목격할 수 있다. 다음 그림은 이렇게 생성된 t0부터 t11까지의 노드를 표현한 것이다.

텍스트, 시계이(가) 표시된 사진

자동 생성된 설명<첫 부분: t0~t11까지의 node 생성, create\_node(0~110)>

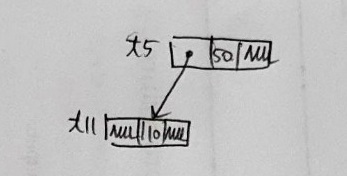
-그 다음 main함수는 이렇게 생성된 node들과 add\_child함수를 이용해 각 노드들에 child관계를 주어 트리를 만드는 것을 볼 수 있다. 먼저 t3에 t7과 t8을, t4에는 t9와 t10을, t5에는 t11만 주고(right에 NULL을 주는 것은 안주겠다는 것과 마찬가지), t1에는 이렇게 만든 t3와 t4를, t2에는 만든 t5와 t6를, 마무리로 가장 위 트리의 root부분인 t0에 t1과 t2를 만들어 놓은 걸 줌으로써 쫙 다 연결되게 하여 트리를 완성시킨다. 다음은 이와 같은 과정을 그림과 함께 설명한 것이다.

시계이(가) 표시된 사진

자동 생성된 설명<add\_child(t3, t7, t8)>

텍스트이(가) 표시된 사진

자동 생성된 설명<add\_child(t4, t9, t10)>

<add\_child(t5, t11, NULL)>

텍스트이(가) 표시된 사진

자동 생성된 설명<add\_child(t1, t3, t4)>

텍스트, 거리, 그래피티이(가) 표시된 사진

자동 생성된 설명<add\_child(t2, t5, t6)>

텍스트이(가) 표시된 사진

자동 생성된 설명<add\_child(t0, t1, t2)>

텍스트, 화이트보드이(가) 표시된 사진

자동 생성된 설명<마무리: 전체 완성도>

그 다음 마지막 부분으로 sum\_tree, 즉 node 하나를 주면 그 부분부터 시작해서 형성되어 있는 tree의 모든 노드들의 값을 더해보는 부분이다. 먼저 sum\_tee(t0)는 t0부터 시작해서 형성되는 tree의 모든 data값들을 더하는 함수이다. 이를 그림에서 보자면 아래와 같다.

텍스트이(가) 표시된 사진

자동 생성된 설명<sum\_tree(t0)>

위 그림처럼 되므로 t0~t11까지 싹 다 더해지는 행동이 실행된다.

나머지 것들도 동일하게 실행된다.

텍스트이(가) 표시된 사진

자동 생성된 설명<sum\_tree(t1)>

T1부터 아래 연결된 모든 트리의 값을 더하는 것이 sum\_tree(t1)이고,

텍스트, 화이트보드이(가) 표시된 사진

자동 생성된 설명<sum\_tree(t2)>

T2부터 아래 연결된 모든 트리의 값을 더하는 것이 sum\_tree(t2),

텍스트, 화이트보드이(가) 표시된 사진

자동 생성된 설명<sum\_tree(t4)>

T4부터 아래 연결된 모든 트리의 값을 더하는 것이 sum\_tree(t4),

텍스트, 화이트보드이(가) 표시된 사진

자동 생성된 설명<sum\_tree(t5)>

T5부터 아래 연결된 모든 트리의 값을 더하는 것이 sum\_tree(t5)이다.

마무리로 main함수는 EXIT\_SUCCESS를 return하고 함수를 종료하는 것을 볼 수 있다.

2.3

s1910783@oak:hw06$ gcc tree2.c test-tree2.c -o test-tree2

s1910783@oak:hw06$ ./test-tree2

===== test-tree2 =====

sum\_tree(t0)=660

sum\_tree(t1)=420

sum\_tree(t2)=240

sum\_tree(t4)=230

sum\_tree(t5)=160

3. Binary search tree(이진 탐색 트리)

3.1

// ======================================================================

NODE \*insert\_node(NODE \* root, int key)

{

NODE \* node;

node = (NODE \*)malloc(sizeof(NODE));

node->key = key;

node->left = NULL;

node->right = NULL;

if (root == NULL) return node;

if (key < root->key) {

root->left = insert\_node(root->left, key);

}

else if (key > root->key) {

root->right = insert\_node(root->right, key);

}

return root;

}

// ======================================================================

void print\_inorder(NODE \* root)

{

if (root) {

print\_inorder(root->left); //left-first

printf("(%d)", root->key); //root-second

print\_inorder(root->right); //right-last

}

}

// ======================================================================

\*이렇게 프로그램한 이유:

-먼저 NODE \*insert\_node(NODE \* root, int key)함수는 node root가 구성하는 binary search tree에 key를 규칙에 맞게 넣어야 되기 때문에 먼저 key가 저장될 node를 malloc으로 생성해야 한다. 이를 위해 NODE pointer 타입으로 새 노드인 node를 선언하고, 그 node에 malloc을 통해 NODE 타입의 크키만큼 메모리 공간을 부여한다. NODE pointer타입으로의 강제 형변환은 덤으로 혹시 몰라 하였다. 그 다음 이제 독립적인 새 노드임과 key를 집어넣기 위한 마무리로써 key 필드에 key값을, 나머지 left, right link필드에는 NULL값을 주어 독립적은 key node를 만들어 넣을 준비를 완료하였다. 그 다음부터 본격적으로 이진 탐색 트리의 insert를 시작하는데, 먼저 이진탐색트리 root가 아예 비어서 아무것도 없는 상태일 경우를 생각한다. 그럴 경우 새로 insert할 key를 넣은 node가 새 이진 탐색 트리의 시작이 되기 때문에 그냥 if문에 조건으로 root가 NULL일 경우를 주고 이 경우 바로 node를 return시켜버려 해결했다. 이 경우가 아닌 경우에는 정상적으로 존재하는 이진 탐색 트리에 key값을 넣은 node를 insert시켜야 하기 때문에 이진 탐색 트리의 조건에 맞게 if-else if문으로 주어 한쪽은 key가 root의 key값보다 작을 경우 root의 left에 넣어 다시 insert\_node함수를 부르는 순환으로, 다른 한쪽은 key가 root의 key값보다 클 경우 root의 right에 넣어 다시 insert\_node함수를 부르는 순환으로 순환을 거듭해가며 그 자리에 NULL이 나오면 그냥 if문이고 else-if문이고 수행이 안되게 되니 그 자리에 바로 넣어 지게 되고, 그러면 그냥 root를 return시키면 그 상태의 이진 탐색 트리가 return되게 된다.

-그 다음 void print\_inorder(NODE \* root)함수는 앞서 1번에서 했던 중위순회 함수와 정말 말그대로 똑같고 pointer인 것만 차이가 있기에 동일한 맥락으로 왼쪽 순회 순환-중간 출력-오른쪽 순회 순환의 형식으로 간단히 해주면 된다. 다만 여기서 if문의 조건으로 root가 NULL일 경우에는 출력할 께 없으므로 조건으로 주어 이를 체크한다.

3.2

// ======================================================================

#include <stdio.h>

#include <stdlib.h>

#include "tree3.h"

// ======================================================================

int main(void)

{

int key;

NODE \*t1;

printf("===== test-tree3 =====\n");

key = 30;

printf("insert=%d\n", key);

t1 = insert\_node(NULL, key);

printf("tree=");

print\_inorder(t1);

printf("\n");

key = 80;

printf("insert=%d\n", key);

t1 = insert\_node(t1, key);

printf("tree=");

print\_inorder(t1);

printf("\n");

key = 90;

printf("insert=%d\n", key);

t1 = insert\_node(t1, key);

printf("tree=");

print\_inorder(t1);

printf("\n");

key = 10;

printf("insert=%d\n", key);

t1 = insert\_node(t1, key);

printf("tree=");

print\_inorder(t1);

printf("\n");

key = 70;

printf("insert=%d\n", key);

t1 = insert\_node(t1, key);

printf("tree=");

print\_inorder(t1);

printf("\n");

key = 50;

printf("insert=%d\n", key);

t1 = insert\_node(t1, key);

printf("tree=");

print\_inorder(t1);

printf("\n");

key = 60;

printf("insert=%d\n", key);

t1 = insert\_node(t1, key);

printf("tree=");

print\_inorder(t1);

printf("\n");

key = 20;

printf("insert=%d\n", key);

t1 = insert\_node(t1, key);

printf("tree=");

print\_inorder(t1);

printf("\n");

key = 40;

printf("insert=%d\n", key);

t1 = insert\_node(t1, key);

printf("tree=");

print\_inorder(t1);

printf("\n");

return EXIT\_SUCCESS;

}

// ======================================================================

\*어떻게 작동하는 지 설명 with tree:

-먼저 main함수에서 이진 탐색 트리인 NODE pointer 타입의 t1을 선언하고 그에 넣을 key변수를 선언하여 테스트를 할 준비를 한다. 그 다음 본격적으로 테스트를 시작하는데, key에 일정한 값들을 넣고 넣을 때 마다 중위 순회로 그 트리를 순회시켜 제대로 넣어졌는지를 체크하는 형식으로 대체적으로 짜여져 있다. 시작은 key값을 30으로 하여 t1에 insert하고 그 결과를 중위 순회로 출력하고 있다. 첫 시작이기에 트리는 그냥 key값을 30으로 가지는 node 하나만을 가지는 형태로 insert된다.

시계, 측정기이(가) 표시된 사진

자동 생성된 설명<key=30; insert\_node(NULL, key)>

다음 insert부터는 이제 만들어진 이진 탐색 트리에 규칙을 지키면서 insert함수에 의해 차례차례 삽입이 된다. 아래는 나머지 쭉 이어진 코드를 시각화한 것이다.

시계이(가) 표시된 사진

자동 생성된 설명<key=80; insert\_node(t1, key)>

측정기, 시계이(가) 표시된 사진

자동 생성된 설명<key=90; insert\_node(t1, key)>

텍스트이(가) 표시된 사진

자동 생성된 설명<key=10; insert\_node(t1, key)>

텍스트이(가) 표시된 사진

자동 생성된 설명<key=70; insert\_node(t1, key)>

텍스트이(가) 표시된 사진

자동 생성된 설명<key=50; insert\_node(t1, key)>

텍스트이(가) 표시된 사진

자동 생성된 설명<key=60; insert\_node(t1, key)>

텍스트이(가) 표시된 사진

자동 생성된 설명<key=20; insert\_node(t1, key)>

텍스트, 화이트보드이(가) 표시된 사진

자동 생성된 설명<key=40; insert\_node(t1, key)>

이렇게 해서 마무리 되고 EXIT\_SUCCESS로 main함수가 종료된다. 결론적으로 위 key40을 insert한 것 까지가 이진 탐색 트리의 완성이라고 할 수 있다.

3.3

s1910783@oak:hw06$ gcc tree3.c test-tree3.c -o test-tree3

s1910783@oak:hw06$ ./test-tree3

===== test-tree3 =====

insert=30

tree=(30)

insert=80

tree=(30)(80)

insert=90

tree=(30)(80)(90)

insert=10

tree=(10)(30)(80)(90)

insert=70

tree=(10)(30)(70)(80)(90)

insert=50

tree=(10)(30)(50)(70)(80)(90)

insert=60

tree=(10)(30)(50)(60)(70)(80)(90)

insert=20

tree=(10)(20)(30)(50)(60)(70)(80)(90)

insert=40

tree=(10)(20)(30)(40)(50)(60)(70)(80)(90)

끝.