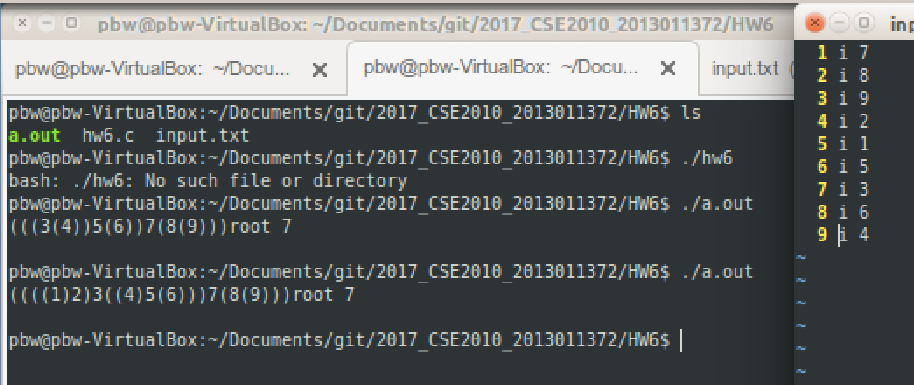
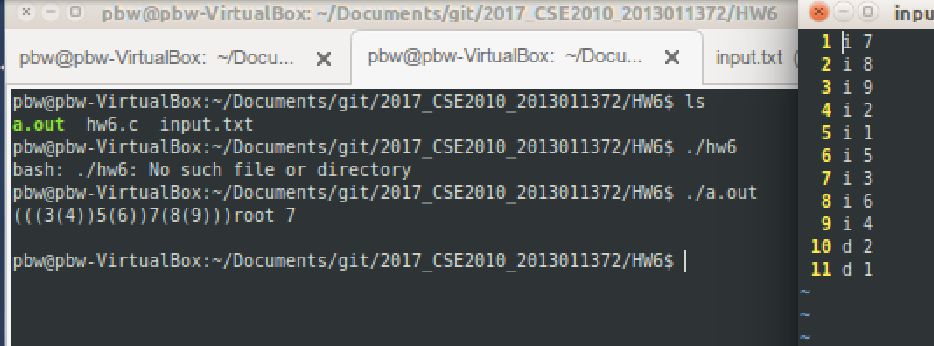
HW6: AVL tree

2013011372 박병욱

1. 과제 목표
   * 1. AVL tree ADT를 구현한다, single rotation, double rotation을 구현한다, avl\_delete함수를 구현한다.
2. 실행결과





1. 소스코드

#include <stdio.h>

#include <stdlib.h>

#include <memory.h>

#define min(a, b) (((a) < (b)) ? (a) : (b))

#define max(a, b) (((a) > (b)) ? (a) : (b))

struct avl\_node {

struct avl\_node \*left\_child, \*right\_child; /\* Subtrees. \*/

int data; /\* Pointer to data. \*/

};

struct avl\_node \*root;

struct avl\_node \*header; //루트를 가리키는 포인터 (루트 삭제시 쓰인다)

// 오른쪽 회전 함수

struct avl\_node\* rotate\_right(struct avl\_node \*parent)

{

struct avl\_node \*child = parent->left\_child;

parent->left\_child= child->right\_child;

child->right\_child= parent;

return child;

}

// 왼쪽 회전 함수

struct avl\_node\* rotate\_left(struct avl\_node \*parent)

{

struct avl\_node \*child = parent->right\_child;

parent->right\_child = child->left\_child;

child->left\_child = parent;

return child;

}

// 오른쪽-왼쪽 회전 함수

struct avl\_node\* rotate\_right\_left(struct avl\_node \*parent)

{

parent->right\_child = rotate\_right(parent->right\_child);

return rotate\_left(parent);

}

// 왼쪽-오른쪽 회전 함수

struct avl\_node\* rotate\_left\_right(struct avl\_node \*parent)

{

parent->left\_child= rotate\_left(parent->left\_child);

return rotate\_right(parent);

}

// 트리의 높이를 반환

int get\_height(struct avl\_node \*node)

{

int height=0;

if( node != NULL )

height = 1 + max(get\_height(node->left\_child), get\_height(node->right\_child));

return height;

}

// 노드의 균형인수를 반환

int get\_height\_diff(struct avl\_node \*node)

{

if( node == NULL ) return 0;

return get\_height(node->left\_child) - get\_height(node->right\_child);

}

// 트리를 균형트리로 만든다

struct avl\_node\* rebalance(struct avl\_node \*\*node)

{

//FOR ADDED

header->right\_child = root;

header->left\_child = root;

int height\_diff = get\_height\_diff(\*node);

if( height\_diff > 1 ){

if( get\_height\_diff((\*node)->left\_child) >= 0 )

\*node = rotate\_right(\*node);

else

\*node = rotate\_left\_right(\*node);

}

else if ( height\_diff < -1 ){

if( get\_height\_diff((\*node)->right\_child) <= 0 )

\*node = rotate\_left(\*node);

else

\*node = rotate\_right\_left(\*node);

}

return \*node;

}

// AVL 트리의 삽입 연산

struct avl\_node \* avl\_add(struct avl\_node \*\*root, int new\_key)

{

if( \*root == NULL ){

\*root = (struct avl\_node \*)malloc(sizeof(struct avl\_node));

if( \*root == NULL ){

exit(1);

}

(\*root)->data = new\_key;

(\*root)->left\_child = (\*root)->right\_child = NULL;

}

else if( new\_key < (\*root)->data ){

(\*root)->left\_child = avl\_add(&((\*root)->left\_child), new\_key);

\*root = rebalance(root);

}

else if( new\_key > (\*root)->data ){

(\*root)->right\_child=avl\_add(&((\*root)->right\_child), new\_key);

\*root = rebalance(root);

}

else{

printf("중복된 키\n");

exit(1);

}

return \*root;

}

//출력함수 구현되어있음

void display(struct avl\_node \*node)

{

if (node != NULL) {

printf("(");

display(node->left\_child);

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

display(node->right\_child);

printf(")");

}

}

// AVL 트리의 탐색함수

struct avl\_node \*avl\_search(struct avl\_node \*node, int key)

{

if( node == NULL ) return NULL;

if( key == node->data ){

return node;

}

else if( key < node->data )

return avl\_search(node->left\_child, key);

else

return avl\_search(node->right\_child, key);

}

// AVL 트리의 부모탐색함수

struct avl\_node \*avl\_parent\_search(struct avl\_node \*node, int key)

{

if( node == NULL ){

printf("there is no %d\n",key);

return NULL;

}

if(node == header) return avl\_parent\_search(root,key);

if(key == root->data){

header->left\_child = root;

header->right\_child = root;

return header;

}

if( node->left\_child != NULL ){

if(key == node->left\_child->data) {

return node;

}

}

if( node->right\_child != NULL ){

if(key == node->right\_child->data){

return node;

}

}

if( key < node->data ) {

return avl\_parent\_search(node->left\_child, key);

}

else {

return avl\_parent\_search(node->right\_child, key);

}

}

//루트로 부터 key를 가진 tmp를 찾고,

//tmp의 부모를 pTmp에 넣는다.

//1. tmp가 left,right 둘다 없을 경우 바로 free하고, 부모에서 tmp쪽 방향에 NULL을 넣는다.

//2. tmp에 right 가 있을경우(right만 있거나 left, right둘다 있을경우)

// right에서 제일 왼쪽에 있는 노드를 찾아 tmp의 key값과 교환하고,

// free하고, 부모에서 left쪽 방향에 NULL을 넣는다.

//3. tmp에 right가 없고, left만 있을경우 tmp의 key값과 교환하고,

// left에서 제일 오른쪽에 있는 노드를 찾아 free하고, 부모에서 right쪽 방향에 NULL을 넣는다.

// root는 free를 해도 NULL이 되지 않으므로, 삭제할 수 없다.

struct avl\_node \* avl\_delete(struct avl\_node \*\*root, int new\_key)

{

struct avl\_node \*temp;

struct avl\_node \*pTemp,\*ppTemp;

struct avl\_node \*iter, \*pIter;

int isleft, data;

pTemp = avl\_parent\_search(\*root,new\_key);

//ppTemp is used for connecting rebalanced node;

if(pTemp !=header)

ppTemp = avl\_parent\_search(\*root, pTemp->data);

else

ppTemp = pTemp;

if(pTemp == NULL){

printf("There is no %d\n",new\_key);

return NULL;

}

//Determine whether temp is positioned to left or right of it's parent

if(pTemp->left\_child != NULL && pTemp->left\_child->data == new\_key){

isleft = 1;

temp = pTemp->left\_child;

}

if(pTemp->right\_child != NULL && pTemp->right\_child->data == new\_key){

isleft = 0;

temp = pTemp->right\_child;

}

//찰드가 아무도 없을 경우

if(temp->left\_child == NULL && temp->right\_child == NULL){

free(temp);

if(isleft) pTemp->left\_child = NULL;

if(!isleft) pTemp->right\_child = NULL;

//rebalance based on pTemp, and connect it to parent of pTemp

if(pTemp == ppTemp->right\_child){

ppTemp->right\_child = rebalance(&pTemp);

}else if(pTemp == ppTemp->left\_child){

ppTemp->left\_child = rebalance(&pTemp);

}

return;

}else{

//삭제될 노드가 오른쪽 찰드가 있을 경우

if(temp->right\_child != NULL){

iter = temp->right\_child;

while(iter->left\_child != NULL){

iter = iter->left\_child;

}

pIter = avl\_parent\_search(\*root,iter->data);

data = iter->data;

avl\_delete(root,iter->data);

temp->data = data;

if(iter == temp->right\_child) {

temp->right\_child = iter->right\_child;

//rebalance based on pTemp, and connect it to parent of pTemp

if(pTemp == ppTemp->right\_child){

ppTemp->right\_child = rebalance(&pTemp);

}else if(pTemp == ppTemp->left\_child){

ppTemp->left\_child = rebalance(&pTemp);

}

}

\*root = rebalance(root);

return;

//삭제 될 노드가 왼쪽 찰드가 있을 경우

}else {

iter = temp->left\_child;

while(iter->right\_child != NULL){

iter = iter->right\_child;

}

pIter = avl\_parent\_search(\*root,iter->data);

data = iter->data;

avl\_delete(root,iter->data);

temp->data = data;

//rebalance based on pTemp, and connect it to parent of pTemp

if(iter == temp->left\_child) {

temp->left\_child = iter->left\_child;

if(pTemp == ppTemp->right\_child){

ppTemp->right\_child = rebalance(&pTemp);

}else if(pTemp == ppTemp->right\_child){

ppTemp->left\_child= rebalance(&pTemp);

}

}

}

\*root = rebalance(root);

return;

}

}

void main()

{

char command;

int key;

FILE \*input;

root = NULL;

input = fopen("input.txt", "r");

header = (struct avl\_node \*)malloc(sizeof(struct avl\_node));

header->left\_child = root;

header->right\_child = root;

header->data = -1;

while (1) {

command = fgetc(input);

if (feof(input)) break;

switch (command) {

case 'i':

fscanf(input, "%d", &key);

avl\_add(&root, key);

break;

case 'd':

fscanf(input, "%d", &key);

avl\_delete(&root, key);

break;

default:

break;

}

}

display(header->left\_child);

printf("root %d\n",root->data);

printf("\n");

}

1. 코드 설명

오른쪽 회전 함수는 왼쪽함수와 마찬가지 방법으로 구현하였다.

찰드를 패런트의 레프트로 두고,

패런트의 레프트에는 찰드의 라이트를 넣어 회전시 붕뜨게되는 노드를 넣었다.

그 후 찰드의 오른쪽에 패런트를 넣고 찰드를 리턴하였다.

오른쪽-왼쪽 회전 함수는

Parent->right\_child 에 right\_child를 로테이트 라이트한 값을 넣은후,

Parent를 다시 left회전하여 반환하였다.

왼쪽-오른쪽은 이와 반대로 구현하였다.

그 후 , 딜리트는

루트로 부터 key를 가진 tmp를 찾고,

tmp의 부모를 pTmp에 넣는다.

1. tmp가 left,right 둘다 없을 경우 바로 free하고, 부모에서 tmp쪽 방향에 NULL을 넣는다.

2. tmp에 right 가 있을경우(right만 있거나 left, right둘다 있을경우)

right에서 제일 왼쪽에 있는 노드를 찾아 tmp의 key값과 교환하고,

free하고, 부모에서 left쪽 방향에 NULL을 넣는다.

3. tmp에 right가 없고, left만 있을경우 tmp의 key값과 교환하고,

left에서 제일 오른쪽에 있는 노드를 찾아 free하고, 부모에서 right쪽 방향에 NULL을 넣는다.

root는 free를 해도 NULL이 되지 않으므로, 삭제할 수 없다.

4.tmp를 삭제한 후 부모를 기준으로 rebalance()한 후, 이 값을 부모의 부모의 찰드로 다시 넣어 지정해주었다.