자료구조 Homework #10



충북대학교 컴퓨터공학과 2022040014 오재식

#include <stdio.h>

#include <stdlib.h>

typedef struct node {

int key;

struct node \*left;

struct node \*right;

} Node;

/\* for stack \*/

#define MAX\_STACK\_SIZE 20

Node\* stack[MAX\_STACK\_SIZE];

int top = -1;

Node\* pop();

void push(Node\* aNode);

/\* for queue \*/

#define MAX\_QUEUE\_SIZE 20

Node\* queue[MAX\_QUEUE\_SIZE];

int front = -1;

int rear = -1;

int initializeBST(Node\*\* h);

void recursiveInorder(Node\* ptr); /\* recursive inorder traversal \*/

int insert(Node\* head, int key); /\* insert a node to the tree \*/

int freeBST(Node\* head); /\* free all memories allocated to the tree \*/

/\* functions that you have to implement \*/

void iterativeInorder(Node\* ptr); /\* iterative inorder traversal \*/

void levelOrder(Node\* ptr); /\* level order traversal \*/

int deleteNode(Node\* head, int key); /\* delete the node for the key \*/

Node\* pop();

void push(Node\* aNode);

Node\* deQueue();

void enQueue(Node\* aNode);

/\* you may add your own defined functions if necessary \*/

void printStack();

int main()

{

char command;

int key;

Node\* head = NULL;

printf("------ 2022040014 ---- ohjaesik -------\n");

do{

printf("\n\n");

printf("----------------------------------------------------------------\n");

printf(" Binary Search Tree #2 \n");

printf("----------------------------------------------------------------\n");

printf(" Initialize BST = z \n");

printf(" Insert Node = i Delete Node = d \n");

printf(" Recursive Inorder = r Iterative Inorder (Stack) = t \n");

printf(" Level Order (Queue) = l Quit = q \n");

printf("----------------------------------------------------------------\n");

printf("Command = ");

scanf(" %c", &command);

switch(command) {

case 'z': case 'Z':

initializeBST(&head);

break;

case 'q': case 'Q':

freeBST(head);

break;

case 'i': case 'I':

printf("Your Key = ");

scanf("%d", &key);

insert(head, key);

break;

case 'd': case 'D':

printf("Your Key = ");

scanf("%d", &key);

deleteNode(head, key);

break;

case 'r': case 'R':

recursiveInorder(head->left);

break;

case 't': case 'T':

iterativeInorder(head->left);

break;

case 'l': case 'L':

levelOrder(head->left);

break;

case 'p': case 'P':

printStack();

break;

default:

printf("\n >>>>> Concentration!! <<<<< \n");

break;

}

}while(command != 'q' && command != 'Q');

return 1;

}

int initializeBST(Node\*\* h) {

/\* if the tree is not empty, then remove all allocated nodes from the tree\*/

if(\*h != NULL)

freeBST(\*h);

/\* create a head node \*/

\*h = (Node\*)malloc(sizeof(Node));

(\*h)->left = NULL; /\* root \*/

(\*h)->right = \*h;

(\*h)->key = -9999;

top = -1;

front = rear = -1;

return 1;

}

void recursiveInorder(Node\* ptr) //재귀 호출로 inorder 구현

{ //중위순회

if(ptr) {

recursiveInorder(ptr->left); //왼쪽으로 이동

printf(" [%d] ", ptr->key); //왼쪽 리프노드부터 출력딤

recursiveInorder(ptr->right); // 오른쪽으로 이동

}

}

/\*\*

\* textbook: p 224

\*/

void iterativeInorder(Node\* node) //반복문으로 inorder 구현

{

for(;;) // 무한 반복

{

for(; node; node = node->left) //왼쪽 리프노드에 도달할때까지 stack에 삽입

push(node);

node = pop(); //마지막 노드 pop

if(!node) break; //노드가 없다면 멈춤

printf(" [%d] ", node->key); // 현재 노드 값 출력

node = node->right; //노드의 오른쪽 자식으로 이동

}

}

/\*\*

\* textbook: p 225

\*/

void levelOrder(Node\* ptr) //BFS

{

// int front = rear = -1;

if(!ptr) return; /\* empty tree \*/

enQueue(ptr); //queue에 노드 삽입

for(;;)

{

ptr = deQueue(); //처음 삽입된 노드 추출

if(ptr) {

printf(" [%d] ", ptr->key); //노드값 출력

if(ptr->left) //존재한다면 노드의 왼쪽 자식 queue에 삽입

enQueue(ptr->left);

if(ptr->right)// 존재한다면 노드의 오른쪽 자식 queue에 삽입

enQueue(ptr->right);

}

else // queue가 비었다면 반복문 종료

break;

}

}

int insert(Node\* head, int key)

{

Node\* newNode = (Node\*)malloc(sizeof(Node));

newNode->key = key;

newNode->left = NULL;

newNode->right = NULL;

if (head->left == NULL) {

head->left = newNode;

return 1;

}

/\* head->left is the root \*/

Node\* ptr = head->left;

Node\* parentNode = NULL;

while(ptr != NULL) {

/\* if there is a node for the key, then just return \*/

if(ptr->key == key) return 1;

/\* we have to move onto children nodes,

\* keep tracking the parent using parentNode \*/

parentNode = ptr;

/\* key comparison, if current node's key is greater than input key

\* then the new node has to be inserted into the right subtree;

\* otherwise the left subtree.

\*/

if(ptr->key < key)

ptr = ptr->right;

else

ptr = ptr->left;

}

/\* linking the new node to the parent \*/

if(parentNode->key > key)

parentNode->left = newNode;

else

parentNode->right = newNode;

return 1;

}

int deleteNode(Node\* head, int key)

{

if (head == NULL) {

printf("\n Nothing to delete!!\n");

return -1;

}

if (head->left == NULL) {

printf("\n Nothing to delete!!\n");

return -1;

}

/\* head->left is the root \*/

Node\* root = head->left;

Node\* parent = NULL;

Node\* ptr = root;

while((ptr != NULL)&&(ptr->key != key)) {

if(ptr->key != key) {

parent = ptr; /\* save the parent \*/

if(ptr->key > key)

ptr = ptr->left;

else

ptr = ptr->right;

}

}

/\* there is no node for the key \*/

if(ptr == NULL)

{

printf("No node for key [%d]\n ", key);

return -1;

}

/\*

\* case 1: the node which has to be removed is a leaf node

\*/

if(ptr->left == NULL && ptr->right == NULL)

{

if(parent != NULL) { /\* parent exists, parent's left and right links are adjusted \*/

if(parent->left == ptr)

parent->left = NULL;

else

parent->right = NULL;

} else {

/\* parent is null, which means the node to be deleted is the root \*/

head->left = NULL;

}

free(ptr);

return 1;

}

/\*\*

\* case 2: if the node to be deleted has one child

\*/

if ((ptr->left == NULL || ptr->right == NULL))

{

Node\* child;

if (ptr->left != NULL)

child = ptr->left;

else

child = ptr->right;

if(parent != NULL)

{

if(parent->left == ptr)

parent->left = child;

else

parent->right = child;

} else {

/\* parent is null, which means the node to be deleted is the root

\* and the root has one child. Therefore, the child should be the root

\*/

root = child;

}

free(ptr);

return 1;

}

/\*\*

\* case 3: the node (ptr) has two children

\*

\* we have to find either the biggest descendant node in the left subtree of the ptr

\* or the smallest descendant in the right subtree of the ptr.

\*

\* we will find the smallest descendant from the right subtree of the ptr.

\*

\*/

Node\* candidate;

parent = ptr;

candidate = ptr->right;

/\* the smallest node is left deepest node in the right subtree of the ptr \*/

while(candidate->left != NULL)

{

parent = candidate;

candidate = candidate->left;

}

/\* the candidate node is the right node which has to be deleted.

\* note that candidate's left is null

\*/

if (parent->right == candidate)

parent->right = candidate->right;

else

parent->left = candidate->right;

/\* instead of removing ptr, we just change the key of ptr

\* with the key of candidate node and remove the candidate node

\*/

ptr->key = candidate->key;

free(candidate);

return 1;

}

void freeNode(Node\* ptr)

{

if(ptr) {

freeNode(ptr->left);

freeNode(ptr->right);

free(ptr);

}

}

int freeBST(Node\* head)

{

if(head->left == head)

{

free(head);

return 1;

}

Node\* p = head->left;

freeNode(p);

free(head);

return 1;

}

Node\* pop() //stack의 top값 추출.

{

if (top < 0) return NULL; //top이 0보다 작을때

return stack[top--];

}

void push(Node\* aNode) // stack의 top에 삽입

{

stack[++top] = aNode;

}

void printStack()

{

int i = 0;

printf("--- stack ---\n");

while(i <= top)

{

printf("stack[%d] = %d\n", i, stack[i]->key);

}

}

Node\* deQueue() //stack의 front값 추출

{

if (front == rear) {

// printf("\n....Now Queue is empty!!\n" );

return NULL;

}

front = (front + 1) % MAX\_QUEUE\_SIZE;

return queue[front];

}

void enQueue(Node\* aNode) // queue에 rear에 값 삽입

{

rear = (rear + 1) % MAX\_QUEUE\_SIZE;

if (front == rear) {

// printf("\n....Now Queue is full!!\n");

return;

}

queue[rear] = aNode;

}

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진

자동 생성된 설명텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명

<https://github.com/ohjaesik/homework10>