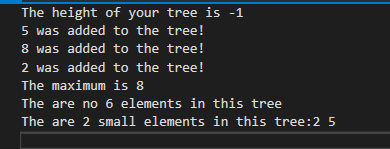
**תרגיל בית 4 – מת"מ -   
מגישים:  
אורי מלכא – 314862996  
אלן ציפין - 313206062  
  
מיין:  
מימוש:  
  
  
  
  
  
  
הכרזות:**

#include "HW4\_Header.h"

CMP Cmp\_Int(Pkey a, Pkey b) {

if (\*(int\*)a == \*(int\*)b)

return EQUAL;

return(\*(int\*)a > \*(int\*)b ? SMALLER : GREATER);

}

void Print\_Int(Pkey toprint,FILE\* out) {

fprintf(out,"%d ",\*(int\*)toprint);

}

void Free\_Int(Pkey tofree) {

free(tofree);

}

int main() {

//vars

PTree root = NULL;

FILE\* in, \* out;

int k,func\_in = 0;

void\* temp\_num = NULL;

//open reading and writing files

in = fopen("instructions.txt", "rt");

if (in == NULL) {

printf("We couldn't open the reading file! Error!");

exit(1);

}

out = fopen("output.txt", "wt");

if (out == NULL) {

printf("We couldn't open the writing file! Error!");

exit(1);

}

/\*

We are creating the "base" root from the main to have a better control on our tree from the main.

from this moment on if we exit the program from any reason we must free the tree using the delete tree function.

\*/

root = CreateNode(temp\_num);

if (root == NULL) {

printf("Memo Allocation failed!"); //here there is nothing to free if failed.

exit(1);

}

while (fscanf(in, "%d", &func\_in) != EOF) {

switch (func\_in) {

case 1:

temp\_num = (int\*)malloc(sizeof(int)); //we allocate the data type

if (temp\_num == NULL)

{

printf("Memo Allocation failed!");

DeleteTree(root, Free\_Int); //if failed we delete the current tree

exit(1); //to exit and close the text files.

}

fscanf(in, "%d", temp\_num);

if(AddNode(root, temp\_num, Cmp\_Int, Free\_Int)==FALSE) //function 1 + Case that we already has the same data in the tree

{

fprintf(out, "we already have %d key in the tree! %d wasn't added!\n", \*(int\*)temp\_num, \*(int\*)temp\_num);

Free\_Int(temp\_num); //we free the useless allocated key

}

else

fprintf(out, "%d was added to the tree!\n", \*(int\*)temp\_num);

break;

case 2:

PrintInOrder(root, Print\_Int, out);

fprintf(out, "\n");

break;

case 3:

if (temp\_num == NULL) {

fprintf(out, "The height of your tree is -1\n", TreeHeight(root));

}

else

fprintf(out, "The height of your tree is %d\n", TreeHeight(root));

break;

case 4:

FindMaxData(root, Print\_Int, out);

break;

case 5:

fscanf(in, "%d",&k);

PrintkMinData(root, k, Print\_Int, out);

break;

default:

fprintf(out, "there is no %d function to call", func\_in); //default case for a wrong function number to call

}

}

DeleteTree(root,Free\_Int); //destroy the tree an

fclose(in);

fclose(out);

return 0;

}

#include "HW4\_Header.h"

struct Tree

{

Pkey data;

struct Tree \*left;

struct Tree \*right;

};

BOOL AddNode(PTree root, Pkey toadd, Cmp\_Key cmpdata,Free\_Key freedata) {

CMP stat=NEUTRAL;

PTree new=NULL,q=root,prev=NULL;

/\*

Because we send the allocated memo of to add into this function

we want to return false when we already have the same pkey in the tree

if we have the same pkey in the tree we will free the memo of the "temp" allocated data in the main.

\*/

if (root->data == NULL) { //this is for the first NODE that we are making we wanna make sure that we are not allocating another node beacuse we have done it in the main for the base root.

root->data = toadd;

return TRUE;

}

while (q != NULL) {

stat=cmpdata(q->data, toadd); //we put the returned value of cmp into stat

if (stat == EQUAL) //case that a node with the same data already exists

return FALSE;

prev = q; //we save the prev node to indicate the previous node so we can link them.

if (stat == GREATER) { //if we need to go right

q = q->right;

}

else q = q->left; //else we need to go left

}

//once we got here we know that if stat=greater it indicates that we need to link the new node to prev->right else prev->left

new = CreateNode(toadd);

if (new == NULL) {

DeleteTree(root, freedata);

printf("Memo Allocation Failed!");

exit(1);

}

if (stat == GREATER)

prev->right = new;

else prev->left = new;

return TRUE;

}

void PrintInOrder(PTree root, Print\_Key printdata, FILE\* out) {

if (root == NULL)

return;

PrintInOrder(root->left, printdata, out);

printdata(root->data, out); //when we reach a node with left son null it means that we need to print it and keep go right.

PrintInOrder(root->right, printdata, out);

}

int TreeHeight(PTree root)// returns the Height of a given tree #3

{

if (root == NULL) return -1;

return Max(TreeHeight(root->left) + 1, TreeHeight(root->right) + 1); //we have a max function defined, because of the tree height definition that the tree height determined by the deepest node.

}

void FindMaxData(PTree root, Print\_Key printdata, FILE\* out) {

PTree q = root;

while (q ->right!= NULL) { //when the q->right is null the q->data is our max (binary tree only)

q = q->right;

}

fprintf(out, "The maximum is ");

printdata(q->data, out);

fprintf(out, "\n");

}

void PrintkMinData(PTree root, int k, Print\_Key printdata, FILE\* out) {

PTree q = root, p = NULL;

int leftsize = 0, rightsize = 0, totalsize = 0, n = k;

totalsize = TreeSize(root);

if (n <= 0) //case for n that is not positive

{

fprintf(out, "We must have a positive number of elements to print!\n");

return;

}

if (totalsize < n) { //case that tree size is less then n

fprintf(out, "There are no %d elements in this tree!\n", k);

return;

}

fprintf(out, "The are %d small elements in this tree: ", k);

while (n > 0) { //logarithm similar to binary search

totalsize = TreeSize(root);

leftsize = TreeSize(q->left);

rightsize = TreeSize(q->right);

if (totalsize == n) { //print the whole tree/sub tree so far

PrintInOrder(q, printdata, out);

fprintf(out, "\n");

return;

}

if (n >= leftsize + 1) { //we want to find out if we need to print the whole left tree + the root

if(q->left!=NULL)

PrintInOrder(q->left, printdata, out); //make sure that the left size is not empty, if its not empty print the whole left tree

printdata(q->data, out); //print the base root data

q = q->right; //and our new base root is the right size.

n = n - leftsize - 1; //we reduce the n to save the amount of keys left to print.

}

else if (n == leftsize) { //if we don't need to print the root only the left size

PrintInOrder(q->left, printdata, out);

fprintf(out, "\n");

return;

}

else q = q->left; //it means that the left tree size is bigger then our n.

}

fprintf(out, "\n");

}

void DeleteTree(PTree root, Free\_Key freedata) {

if (root == NULL)

return;

DeleteTree(root->left,freedata);

DeleteTree(root->right, freedata);

freedata(root->data); //free the memo allocated data type outside

free(root);

root = NULL; //we wanna make sure that the tree will keep its structure so we can keep the delete processes

}

PTree CreateNode(Pkey toadd) {

PTree node;

node = (PTree)malloc(sizeof(struct Tree));

if (node == NULL)

return NULL; // we handle the case of memo failed outside this function.

//initialize

node->left = NULL;

node->right= NULL;

node->data = toadd;

return node;

}

int TreeSize(PTree root) {

if (root == NULL)

return 0;

else return(TreeSize(root->left) + 1 + TreeSize(root->right));

}

#define \_CRT\_SECURE\_NO\_WARNINGS

#ifndef \_HW4\_Imp

#define \_HW4\_Imp

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

#include <stdio.h>

#include <stdlib.h>

#include <conio.h>

typedef struct Tree \*PTree;

typedef enum {FALSE,TRUE} BOOL;

typedef enum { NEUTRAL,EQUAL, GREATER, SMALLER } CMP;

typedef void\* Pkey;

typedef CMP(\*Cmp\_Key)(Pkey,Pkey); // a ptr to a function that will compare the keys.

typedef void (\*Print\_Key)(Pkey,FILE\*); // a ptr to a function that will write the key.

typedef void (\*Free\_Key)(Pkey); // a ptr to a function that will free the key.

BOOL AddNode(PTree root, Pkey toadd, Cmp\_Key cmpdata, Free\_Key freedata); //#1

/\*

Function name: AddNode

Input: PTree,Pkey,Cmp\_Key,freedata

Output: BOOL (true or false) if its true it meants that we added a new node with the new key else, we already have the same key in the tree.

Function Algorithm: we create a new node if its not the first or call or the key is not in the tree, we call CreateNode function to help us allocate the memo

\*/

void PrintInOrder(PTree root,Print\_Key printdata, FILE\* out); //#2

/\*

Function name: PrintInOrder

Input: PTree,Print\_Key,FILE\*

Output: void

Function Algorithm: we print the whole tree recursively using the pointer to the print/write data so we can print the data type in ADT form.

\*/

int TreeHeight(PTree root); //#3

/\*

Function name: TreeHeight

Input: PTree the tree root

Output: int (the height of the tree) always an integer

Function Algorithm: we return the tree height recursively (defined to be the deepest node in the tree)

\*/

void FindMaxData(PTree root,Print\_Key printdata, FILE\* out); //#4

/\*

Function name: FindMaxData

Input: PTree,Print\_Key,FILE\*

Output: void

Function Algorithm: we find the max value with a while loop (because its binary tree we go until we have null in the right tree) and we write it to the output file.

\*/

void PrintkMinData(PTree root, int k, Print\_Key printdata, FILE\* out); //#5

/\*

Function name: PrintkMinData

Input:PTree,int,Print\_Key,out

Output: void

Function Algorithm: similar to binary search but we search a sub-tree and not a number that determined by k.

\*/

//General functions add ons

void DeleteTree(PTree root, Free\_Key freedata);

/\*

Function name: DeleteTree

Input:PTree,Free\_Key

Output:void

Function Algorithm:we delete the whole tree recursively.

\*/

PTree CreateNode(Pkey toadd);

/\*

Function name:CreateNode

Input:PKey

Output:PTree

Function Algorithm:We allocate memo for a new node, initliaze the node and put the data inside it,

(data is defined to be already allocated so if we return from this function NULL, its means that memo failed) and we will free the memo in the called function.

\*/

int TreeSize(PTree root);

/\*

Function name :TreeSize

Input : PTree

Output : int

Function Algorithm : finds the size of the tree (number of elements)

\*/

#endif