202135574 전시현 Jeon Sihyeon

// implementation of AVL tree

// source from https://www.thecrazyprogrammer.com/2014/03/c-program-for-avl-tree-implementation.html

/\*\*

 \* file phw3-1.c

 \*

 \* author : Jeon Sihyeon (wjstlgus13@gachon.ac.kr)

 \*

 \* Date : 03/30/2022

 \* Partnet : I worked alone

 \* Course : Data Structures(14461\_004)

 \*

 \* Summary of File :

 \*      This file contains code which generates random numbers and makes an AVL

 \*      height-balanced binary search tree with them. After it, it traverses and

 \*      prints keys in inorder, preorder, and postorder. And It prints the highest key

 \*      and all keys lower than 15, bigger than 50 and between 15 and 50.

 \*

\*\*/

#include <stdio.h>

#include <time.h>

#include <stdlib.h>

int cnt = 0;

typedef struct node

{

    int data;

    struct node \*left, \*right;

    int ht;

} node;

node \*insert(node \*, int);

void preorder(node \*);

void inorder(node \*);

void postorder(node \*);

int height(node \*);

node \*rotateright(node \*);

node \*rotateleft(node \*);

node \*RR(node \*);

node \*LL(node \*);

node \*LR(node \*);

node \*RL(node \*);

int BF(node \*);

node \*search(node \*Node, int x);

void GenerateRandomIntegers(int array[], int size, int num);

void FindHighestKey(node \*, int \*num);

void searchCertainRange(node \*Node, int x, int y);  //function declarations

int main()

{

    srand(time(NULL));  //random

    node \*root = NULL;

    int x, n, i;

    int list[500];

    GenerateRandomIntegers(list+2, 500, 198);   //calls GenerateRandomIntegers function

    root = NULL;

    list[0] = 15;

    list[1] = 50;

    for (i = 0; i < 200; i++)

    {

        x = list[i];

        root = insert(root, x); //calls insert function

        if(cnt==50)

            break;      //first 50 unique keys

    }

    printf("In inorder :\n");

    inorder(root);

    printf("\nIn preorder :\n");

    preorder(root);

    printf("\nIn postorder :\n");

    postorder(root);    //traverse and print keys in inorder, preorder, and postorder

    printf("\n\n");

    node \*temp1 = search(root, 15);

    temp1 = search(root, 20);

    temp1 = search(root, 50);

    temp1 = search(root, 70);

    temp1 = search(root, 90);   //search the tree for keys 15, 20, 50, 70 and 90

    printf("\n");

    int \*max;

    FindHighestKey(root, max);  //find the highest (max) key

    printf("highest(max) key : %d\n\n", \*max);

    printf("Keys smaller than 15 : ");

    searchCertainRange(root, 0, 15);    //search and print all keys < 15

    printf("\nKeys bigger than 50 : ");

    searchCertainRange(root, 50, 100);  //search and print all keys > 50

    printf("\nKeys between 15 and 50 : ");

    searchCertainRange(root, 15, 50);   //search and print all keys between 15 and 50

    printf("\n");

    return 0;

}

/\*\*

 \* node \*insert(node \*T, int x)

 \*

 \* Summary of the insert :

 \*      The insert function inserts x into binary tree. -- NOTE : it's a height-balanced

 \*                                                                  binary search tree

 \*

 \* Parameters   : struct pointer, integer number

 \*

 \* Return Value : node pointer

 \*

 \* Description :

 \*

 \*      This function utilizes recursive algorithm.

 \*

 \*/

node \*insert(node \*T, int x)

{

    if (T == NULL)

    {

        T = (node \*)malloc(sizeof(node));   //allocates memory on the heap

        T->data = x;

        T->left = NULL;

        T->right = NULL;

        cnt++;  //counts unique key

    }

    else

        if (x > T->data) // insert in right subtree

    {

        T->right = insert(T->right, x); //calls insert function

        if (BF(T) == -2)    //balance factor is -2

            if (x > T->right->data)

                T = RR(T);  //calls RR function

            else

                T = RL(T);  //calls RL function

    }

    else

        if (x < T->data)

    {

        T->left = insert(T->left, x);   //calls insert function

        if (BF(T) == 2) //balance factor is 2

            if (x < T->left->data)

                T = LL(T);  //calls LL function

            else

                T = LR(T);  //calls LR function

    }

    T->ht = height(T);  //calls height function

    return (T);

}

/\*\*

 \* int height(node \*T)

 \*

 \* Summary of the height :

 \*      The height function returns T's height

 \*

 \* Parameters   : struct pointer

 \*

 \* Return Value : integer number

 \*

 \*/

int height(node \*T)

{

    int lh, rh;

    if (T == NULL)

        return (0);

    if (T->left == NULL)    //There's nothing on T->left

        lh = 0;

    else                    //There's something on T->left

        lh = 1 + T->left->ht;

    if (T->right == NULL)   //There's nothing on T->right

        rh = 0;

    else                    //There's something on T->right

        rh = 1 + T->right->ht;

    if (lh > rh)

        return (lh);

    return (rh);    //returns bigger number between rh and lh

}

/\*\*

 \* node \*rotateright(node \*x)

 \*

 \* Summary of the rotateright :

 \*      The rotateright function rotates the nodes in RR : single rotation(left)

 \*

 \* Parameters   : struct pointer

 \*

 \* Return Value : struct pointer

 \*

 \*/

node \*rotateright(node \*x)

{

    node \*y;

    y = x->left;        //y is pointing x->left

    x->left = y->right; //x->left is y->right now

    y->right = x;       //y->right is x now

    x->ht = height(x);

    y->ht = height(y);  //calls height function

    return (y);

}

/\*\*

 \* node \*rotateleft(node \*x)

 \*

 \* Summary of the rotateleft :

 \*      The rotateleft function rotates the nodes in LL : single rotation(right)

 \*

 \* Parameters   : struct pointer

 \*

 \* Return Value : struct pointer

 \*

 \*/

node \*rotateleft(node \*x)

{

    node \*y;

    y = x->right;

    x->right = y->left;

    y->left = x;

    x->ht = height(x);

    y->ht = height(y);  //calls height function

    return (y);

}

/\*\*

 \* node \*RR(node \*T)

 \*

 \* Summary of the RR :

 \*      The RR function rotates the nodes in RR : single rotation(left)

 \*

 \* Parameters   : struct pointer

 \*

 \* Return Value : struct pointer

 \*

 \*/

node \*RR(node \*T)

{

    T = rotateleft(T);  //calls rotateleft function

    return (T);

}

/\*\*

 \* node \*LL(node \*T)

 \*

 \* Summary of the LL :

 \*      The LL function rotates the nodes in LL : single rotation(right)

 \*

 \* Parameters   : struct pointer

 \*

 \* Return Value : struct pointer

 \*

 \*/

node \*LL(node \*T)

{

    T = rotateright(T); //calls rotateright function

    return (T);

}

/\*\*

 \* node \*LR(node \*T)

 \*

 \* Summary of the LR :

 \*      The LR function rotates the nodes in LR : double rotation(left and right)

 \*

 \* Parameters   : struct pointer

 \*

 \* Return Value : struct pointer

 \*

 \*/

node \*LR(node \*T)

{

    T->left = rotateleft(T->left);  //calls rotateleft function

    T = rotateright(T);             //calls rotateright function

    return (T);

}

/\*\*

 \* node \*RL(node \*T)

 \*

 \* Summary of the RL :

 \*      The RL function rotates the nodes in RL : double rotation rotation(right and left)

 \*

 \* Parameters   : struct pointer

 \*

 \* Return Value : struct pointer

 \*

 \*/

node \*RL(node \*T)

{

    T->right = rotateright(T->right);   //calls rotateright function

    T = rotateleft(T);                  //calls rotateleft function

    return (T);

}

/\*\*

 \* int BF(node \*T)

 \*

 \* Summary of the BF :

 \*      This function returns the balance factor of T

 \*

 \* Parameters   : struct pointer

 \*

 \* Return Value : integer number

 \*

 \*/

int BF(node \*T)

{

    int lh, rh;

    if (T == NULL)

        return (0);

    if (T->left == NULL)    //if there's nothing on T->left

        lh = 0;

    else                    //if there's something on T->left

        lh = 1 + T->left->ht;

    if (T->right == NULL)   //if there's nothing on T->right

        rh = 0;

    else                    //if there's something on T->right

        rh = 1 + T->right->ht;

    return (lh - rh);

}

/\*\*

 \* void preorder(node \*T)

 \*

 \* Summary of the preorder :

 \*      The preorder function prints the value of the nodes in preorder

 \*

 \* Parameters   : struct pointer

 \*

 \* Return Value : nothing

 \*

 \* Description :

 \*

 \*      This function utilizes recursive algorithm.

 \*

 \*/

void preorder(node \*T)

{

    if (T != NULL)

    {

        printf("%d(Bf=%d)", T->data, BF(T));

        preorder(T->left);  //calls preorder function

        preorder(T->right); //calls preroder function

    }

}

/\*\*

 \* void inorder(node \*T)

 \*

 \* Summary of the inorder :

 \*      The inorder function prints the value of the nodes in inorder

 \*

 \* Parameters   : struct pointer

 \*

 \* Return Value : nothing

 \*

 \* Description :

 \*

 \*      This function utilizes recursive algorithm.

 \*

 \*/

void inorder(node \*T)

{

    if (T != NULL)

    {

        inorder(T->left);   //calls inorder function

        printf("%d(Bf=%d)", T->data, BF(T));

        inorder(T->right);  //calls inorder function

    }

}

/\*\*

 \* void postorder(node \*T)

 \*

 \* Summary of the postorder :

 \*      The postorder function prints the value of the nodes in postorder

 \*

 \* Parameters   : struct pointer

 \*

 \* Return Value : nothing

 \*

 \* Description :

 \*

 \*      This function utilizes recursive algorithm.

 \*

 \*/

void postorder(node \*T){

    if (T != NULL){

        postorder(T->left);     //calls postorder function

        postorder(T->right);    //calls postorder function

        printf("%d(Bf=%d)", T->data, BF(T));

    }

}

// serach for a node in a tree

/\*\*

 \* node \*search(node \*Node, int x)

 \*

 \* Summary of the search :

 \*      The search function searches the x and prints if there's x in the binary tree

 \*

 \* Parameters   : struct pointer, integer number

 \*

 \* Return Value : struct pointer

 \*

 \* Description :

 \*

 \*      This function utilizes recursive algorithm.

 \*

 \*/

node \*search(node \*Node, int x)

{

    if (Node == NULL)

    {

        printf("Search failed: %d does not exist.\n", x);

    }

    else if (x == Node->data)   //x is found

    {

        printf("Found %d.\n", Node->data);

    }

    else if (x < Node->data)

    {

        Node = search(Node->left, x); //calls search function with Node->left

    }

    else if (x > Node->data)

    {

        Node = search(Node->right, x); //calls search function with Node->right

    }

    return Node;

}

/\*\*

 \* void GenerateRandomIntegers(int array[], int size, int num)

 \*

 \* Summary of the GenerateRandomIntegers :

 \*      The GenerateRandomIntegers function generates random integers and save in the array

 \*

 \* Parameters   : integer array, integer number

 \*

 \* Return Value : nothing

 \*

 \*/

void GenerateRandomIntegers(int array[], int size, int num){

    for (int i = 0; i<num; i++){

        array[i] = rand()%100+1;    //1~100

    }

}

/\*\*

 \* void FindHighestKey(node \*T, int \*num)

 \*

 \* Summary of the FindHighestKey :

 \*      The FindHighestKey function finds the highest key in the binary tree

 \*

 \* Parameters   : struct pointer, integer pointer

 \*

 \* Return Value : nothing

 \*

 \* Description :

 \*

 \*      This function utilizes recursive algorithm.

 \*

 \*/

void FindHighestKey(node \*T, int \*num){

    if(T->right!=NULL){

        FindHighestKey(T->right, num);  //calls FindHighestKey function

    }

    else{   //T->right is NULL

        \*num = T->data;

    }

}

/\*\*

 \* void searchCertainRange(node \*Node, int x, int y)

 \*

 \* Summary of the searchCertainRange :

 \*      The searchCertainRange function prints the value of the nodes in certain range

 \*

 \* Parameters   : struct pointer, integer number

 \*

 \* Return Value : nothing

 \*

 \* Description :

 \*

 \*      This function utilizes recursive algorithm.

 \*

 \*/

void searchCertainRange(node \*Node, int x, int y){      //x<y

    if (Node == NULL){

        return;

    }

    if (x <= Node->data)

    {

        searchCertainRange(Node->left, x, y);   //calls searchCertainRange function

    }

    if (x < Node->data && y>Node->data)

    {

        printf("%d ", Node->data);

    }

    searchCertainRange(Node->right, x, y);      //calls searchCertainRange function

}

텍스트이(가) 표시된 사진

자동 생성된 설명