Name: Vidyut Chakrabarti Roll no: 68/B4

Code:

* Binary Search tree Structure(bst.h):

#include <stdio.h>

#include <stdlib.h>

struct tnode

{

    int data;

    struct tnode \*left;

    struct tnode \*right;

};

struct tnode \*create\_tNode(int elem) {

    struct tnode \*temp;

    temp = (struct tnode \*)malloc(sizeof(struct tnode));

    if (temp != NULL) {

        temp->data = elem;

        temp->left = NULL;

        temp->right = NULL; }

    else {

        printf("Allocation of memory failed.");

        exit(0); }

    return temp; }

struct tnode \*insertnodebst(struct tnode \*root, int elem) {

    struct tnode \*temp;

    temp = root;

    if (temp == NULL) {

        struct tnode \*newnode = create\_tNode(elem);

        temp = newnode; }

    else {

        if (temp->data >= elem) {

            temp->left = insertnodebst(temp->left, elem); }

        else if (temp->data < elem) {

            temp->right = insertnodebst(temp->right, elem); }}

    return temp; }

void inorder(struct tnode \*root) {

    if (root != NULL) {

        inorder(root->left);

        printf("%d ", root->data);

        inorder(root->right); }}

void preorder(struct tnode \*root) {

    if (root != NULL)   {

        printf("%d ", root->data);

        preorder(root->left);

        preorder(root->right); }}

void postorder(struct tnode \*root){

    if (root != NULL) {

        postorder(root->left);

        postorder(root->right);

        printf("%d ", root->data); }}

int countnodes(struct tnode \*root) {

    if (root == NULL) {

        return 0; }

    else {

        return countnodes(root->left) + countnodes(root->right) + 1) }}

int leafnodes(struct tnode \*root) {

    if (root == NULL) {

        return 0; }

    else if (root->left == NULL && root->right == NULL) {

        return 1; }

    else {

        return leafnodes(root->left) + leafnodes(root->right); }}

int intern\_nodes(struct tnode \*root) {

    if (root == NULL) {

        return 0; }

    else if (root->left == NULL && root->right == NULL) {

        return 0; }

    else {

        return intern\_nodes(root->left) + intern\_nodes(root->right) + 1; }}

int max(int c, int b) {

    if (c > b) {

        return c; }

    else if (b > c) {

        return b; }

    else {

        return b;}}

int height(struct tnode \*root){

    if (root == NULL) {

        return 0; }

    else{

        return max(height(root->left), height(root->right)) + 1;}}

struct tnode\* deleteNode(struct tnode\* root, int k){

if (root == NULL)

return root;

if (root->data > k) {

root->left = deleteNode(root->left, k);

return root; }

else if (root->data < k) {

root->right = deleteNode(root->right, k);

return root; }

if (root->left == NULL) {

struct tnode\* temp = root->right;

free(root);

return temp; }

else if (root->right == NULL) {

struct tnode\* temp = root->left;

free(root);

return temp;

} else {

struct tnode\* succParent = root;

struct tnode\* succ = root->right;

while (succ->left != NULL) {

succParent = succ;

succ = succ->left; }

if (succParent != root)

succParent->left = succ->right;

else

succParent->right = succ->right;

root->data = succ->data;

free(succ);

return root; }}

struct tnode \*search(struct tnode \*root, int key){

// Base Cases: root is null or key is present at root

    if (root == NULL || root->data == key)

        return root;

    // Key is greater than root's key

    if (root->data < key)

        return search(root->right, key);

    // Key is smaller than root's key

    return search(root->left, key);}

void levelOrderTraversal(struct tnode \*root) {

    if (root == NULL)

        return;

    struct tnode \*\*queue =

        (struct tnode \*\*)malloc(sizeof(struct tnode \*) \* 50);

    int front = 0, rear = 0;

    queue[rear++] = root;

    while (front < rear) {

        struct tnode \*current = queue[front++];

        printf("%d ", current->data);

        if (current->left != NULL) {

            queue[rear++] = current->left; }

        if (current->right != NULL) {

            queue[rear++] = current->right; }}}

int diameter(struct tnode \*root) {

    if (root == NULL)

        return 0;

    int lh = height(root->left);

    int rh = height(root->right);

    int ldir = diameter(root->left);

    int rdir = diameter(root->right);

    return max(lh + rh + 1,

               max(ldir, rdir));

}

void inorder\_iter(struct tnode \*root) {

    if (root == NULL){

        return; }

    struct tnode \*stack[50];

    int top = -1;

    struct tnode \*current = root;

    while (current != NULL || top != -1){

        // Traverse to the leftmost node

        while (current != NULL) {

            stack[++top] = current;

            current = current->left; }

        // Visit the top of the stack

        current = stack[top--];

        printf("%d ", current->data);

        // Move to the right subtree

        current = current->right; }}

* Driver\_main.c:

#include <stdio.h>

#include "bstb\_68.h"

int main() {

struct tnode \*root = create\_tNode(30);

insertnodebst(root, 5);

insertnodebst(root, 3);

insertnodebst(root, 40);

insertnodebst(root, 35);

insertnodebst(root, 32);

insertnodebst(root, 50);

inorder(root);

printf("\n");

preorder(root);

printf("\n");

postorder(root);

int c = countnodes(root);

printf("\ntotal nodes: %d", c);

int d = leafnodes(root);

printf("\ntotal leaf nodes: %d", d);

int e = intern\_nodes(root);

printf("\ntotal internal nodes: %d", e - 1);

int h = height(root);

printf("\nheight of tree: %d", h);

deleteNode(root, 3);

deleteNode(root, 35);

deleteNode(root, 30);

printf("\nAfter deletion of key 30\ninorder traversal:\n");

inorder(root);

struct tnode \*searched = search(root, 3);

if (searched != NULL)

printf("\nkey=3 found at: %d\n", searched);

else

printf("\nkey not found.\n");

printf("level order traversal:\n");

levelOrderTraversal(root);

int dir = diameter(root);

printf("\nThe diameter of\nthe tree is: %d \n", dir-1);

printf("inorder traversal\nthrough iteration:\n");

inorder\_iter(root); }

**TEST CASES:**

|  |  |  |  |
| --- | --- | --- | --- |
| **SN** | **Input** | **Expected Output** | **Actual Output** |
| 1 | Insert 1,2,3,4,5,6,7,8 | Inorder: 1 2 3 4 5 6 7 8  Preorder: 1 2 3 4 5 6 7 8  Postorder: 8 7 6 5 4 3 2 1 |  |
| 2 | Insert  30,5,3,40,35,32,50 | Inorder:   3   5  30   32   35   40   50  Similarly print preorder and  postorder |  |
| 3 | Delete 3 | Inorder after deletion:  5 30 32 35 40 50 |  |
| 4 | Delete 35 | Inorder after deletion:  5 30 32 40 50 |  |
| 5 | Delete 30 | Inorder after deletion:  5 32 40 50 |  |
| 6 | Height of tree after  step 1 | 8 |  |
| 7 | Height of tree after  step 2 | 4 |  |
| 8 | Height of tree after  step 5 | 3 |  |
| 9 | Count Nodes after 2 | 7 |  |
| 10 | Level order traversal  after step 2 | 30 5 40 3 35 50 32 |  |
| 11 | Count leaf nodes  after step 2 | 3 |  |
| 12 | Count non-leaf  nodes after step 2 | 4 | Total non-leaf nodes: 4 |
| 13 | Count internal after  step 2 | 3 |  |
| 14 | Search key after step  2 | Key = 3 Print the address of 3 |  |
| 15 | Diameter of the tree  created in step 2 | Diameter if the tree = 5 |  |
| 16 | Iterative inorder | Same as step 2 |  |