**Ds day11**

1. Write a C program to search for a number, Min, Max from a BST:

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

struct Node\* newNode(int data) {

struct Node\* node = (struct Node\*)malloc(sizeof(struct Node));

node->data = data;

node->left = node->right = NULL;

return node;

}

struct Node\* insert(struct Node\* node, int data) {

if (node == NULL) return newNode(data);

if (data < node->data)

node->left = insert(node->left, data);

else

node->right = insert(node->right, data);

return node;

}

int findMin(struct Node\* node) {

struct Node\* current = node;

while (current && current->left != NULL)

current = current->left;

return current->data;

}

int findMax(struct Node\* node) {

struct Node\* current = node;

while (current && current->right != NULL)

current = current->right;

return current->data;

}

int main() {

struct Node\* root = NULL;

root = insert(root, 15);

insert(root, 10);

insert(root, 20);

insert(root, 8);

insert(root, 12);

insert(root, 17);

insert(root, 25);

printf("Minimum value in the BST: %d\n", findMin(root));

printf("Maximum value in the BST: %d\n", findMax(root));

return 0;

}

Output:



1. Write a C program to perform the following operations:

a) Insert an element into a AVL tree.

b) Delete an element from a AVL tree.

c) Search for a key element in a AVL tree.

#include <stdio.h>

#include <stdlib.h>

struct Node {

int key;

struct Node \*left;

struct Node \*right;

int height;

};

int max(int a, int b) {

return (a > b) ? a : b;

}

int height(struct Node \*N) {

if (N == NULL)

return 0;

return N->height;

}

struct Node\* newNode(int key) {

struct Node\* node = (struct Node\*)malloc(sizeof(struct Node));

node->key = key;

node->left = NULL;

node->right = NULL;

node->height = 1;

return node;

}

struct Node \*rightRotate(struct Node \*y) {

struct Node \*x = y->left;

struct Node \*T2 = x->right;

x->right = y;

y->left = T2;

y->height = max(height(y->left), height(y->right)) + 1;

x->height = max(height(x->left), height(x->right)) + 1;

return x;

}

struct Node \*leftRotate(struct Node \*x) {

struct Node \*y = x->right;

struct Node \*T2 = y->left;

y->left = x;

x->right = T2;

x->height = max(height(x->left), height(x->right)) + 1;

y->height = max(height(y->left), height(y->right)) + 1;

return y;

}

int getBalance(struct Node \*N) {

if (N == NULL)

return 0;

return height(N->left) - height(N->right);

}

struct Node\* insert(struct Node\* node, int key) {

if (node == NULL)

return newNode(key);

if (key < node->key)

node->left = insert(node->left, key);

else if (key > node->key)

node->right = insert(node->right, key);

else

return node;

node->height = 1 + max(height(node->left), height(node->right));

int balance = getBalance(node);

if (balance > 1 && key < node->left->key)

return rightRotate(node);

if (balance < -1 && key > node->right->key)

return leftRotate(node);

if (balance > 1 && key > node->left->key) {

node->left = leftRotate(node->left);

return rightRotate(node);

}

if (balance < -1 && key < node->right->key) {

node->right = rightRotate(node->right);

return leftRotate(node);

}

return node;

}

struct Node\* minValueNode(struct Node\* node) {

struct Node\* current = node;

while (current->left != NULL)

current = current->left;

return current;

}

struct Node\* deleteNode(struct Node\* root, int key) {

if (root == NULL)

return root;

if (key < root->key)

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

else if (key > root->key)

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

else {

if ((root->left == NULL) || (root->right == NULL)) {

struct Node \*temp = root->left ? root->left : root->right;

if (temp == NULL) {

temp = root;

root = NULL;

} else

\*root = \*temp;

free(temp);

} else {

struct Node\* temp = minValueNode(root->right);

root->key = temp->key;

root->right = deleteNode(root->right, temp->key);

}

}

if (root == NULL)

return root;

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

int balance = getBalance(root);

if (balance > 1 && getBalance(root->left) >= 0)

return rightRotate(root);

if (balance > 1 && getBalance(root->left) < 0) {

root->left = leftRotate(root->left);

return rightRotate(root);

}

if (balance < -1 && getBalance(root->right) <= 0)

return leftRotate(root);

if (balance < -1 && getBalance(root->right) > 0) {

root->right = rightRotate(root->right);

return leftRotate(root);

}

return root;

}

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

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

return root;

if (key < root->key)

return search(root->left, key);

return search(root->right, key);

}

void preOrder(struct Node \*root) {

if (root != NULL) {

printf("%d ", root->key);

preOrder(root->left);

preOrder(root->right);

}

}

int main() {

struct Node \*root = NULL;

root = insert(root, 10);

root = insert(root, 20);

root = insert(root, 30);

root = insert(root, 40);

root = insert(root, 50);

root = insert(root, 25);

printf("Preorder traversal of the AVL tree is \n");

preOrder(root);

root = deleteNode(root, 10);

printf("\nPreorder traversal after deletion of 10 \n");

preOrder(root);

struct Node\* foundNode = search(root, 30);

if (foundNode != NULL)

printf("\nElement 30 found in the AVL tree.\n");

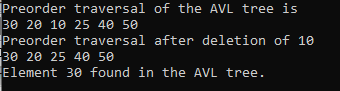
else

printf("\nElement 30 not found in the AVL tree.\n");

return 0;

}

Output:



1. Write a C program to implement Red black tree.

#include <stdio.h>

#include <stdlib.h>

#define RED 1

#define BLACK 0

typedef struct Node {

int data;

int color;

struct Node \*left, \*right, \*parent;

} Node;

Node\* createNode(int data) {

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

if (newNode == NULL) {

fprintf(stderr, "Memory allocation failed for new node.\n");

exit(EXIT\_FAILURE);

}

newNode->data = data;

newNode->color = RED;

newNode->left = newNode->right = newNode->parent = NULL;

return newNode;

}

void leftRotate(Node \*\*root, Node \*x) {

Node \*y = x->right;

x->right = y->left;

if (y->left != NULL) {

y->left->parent = x;

}

y->parent = x->parent;

if (x->parent == NULL) {

\*root = y;

} else if (x == x->parent->left) {

x->parent->left = y;

} else {

x->parent->right = y;

}

y->left = x;

x->parent = y;

}

void rightRotate(Node \*\*root, Node \*y) {

Node \*x = y->left;

y->left = x->right;

if (x->right != NULL) {

x->right->parent = y;

}

x->parent = y->parent;

if (y->parent == NULL) {

\*root = x;

} else if (y == y->parent->left) {

y->parent->left = x;

} else {

y->parent->right = x;

}

x->right = y;

y->parent = x;

}

void fixViolation(Node \*\*root, Node \*newNode) {

Node \*parent = NULL;

Node \*grandparent = NULL;

while ((newNode != \*root) && (newNode->color == RED) && (newNode->parent->color == RED)) {

parent = newNode->parent;

grandparent = parent->parent;

if (parent == grandparent->left) {

Node \*uncle = grandparent->right;

if (uncle != NULL && uncle->color == RED) {

grandparent->color = RED;

parent->color = BLACK;

uncle->color = BLACK;

newNode = grandparent;

} else {

if (newNode == parent->right) {

leftRotate(root, parent);

newNode = parent;

parent = newNode->parent;

}

rightRotate(root, grandparent);

int temp = parent->color;

parent->color = grandparent->color;

grandparent->color = temp;

newNode = parent;

}

} else {

Node \*uncle = grandparent->left;

if ((uncle != NULL) && (uncle->color == RED)) {

grandparent->color = RED;

parent->color = BLACK;

uncle->color = BLACK;

newNode = grandparent;

} else {

if (newNode == parent->left) {

rightRotate(root, parent);

newNode = parent;

parent = newNode->parent;

}

leftRotate(root, grandparent);

int temp = parent->color;

parent->color = grandparent->color;

grandparent->color = temp;

newNode = parent;

}

}

}

(\*root)->color = BLACK;

}

void insert(Node \*\*root, int data) {

Node \*newNode = createNode(data);

if (\*root == NULL) {

newNode->color = BLACK;

\*root = newNode;

return;

}

Node \*parent = NULL;

Node \*current = \*root;

while (current != NULL) {

parent = current;

if (newNode->data < current->data) {

current = current->left;

} else {

current = current->right;

}

}

newNode->parent = parent;

if (newNode->data < parent->data) {

parent->left = newNode;

} else {

parent->right = newNode;

}

fixViolation(root, newNode);

}

void inOrder(Node \*root) {

if (root == NULL) return;

inOrder(root->left);

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

inOrder(root->right);

}

int main() {

Node \*root = NULL;

insert(&root, 10);

insert(&root, 20);

insert(&root, 30);

insert(&root, 15);

printf("In-order traversal of the Red-Black Tree:\n");

inOrder(root);

printf("\n");

return 0;

}

Output:



1. Write a C program to implement B Tree:

#include <stdio.h>

#include <stdlib.h>

#define MIN\_DEGREE 2

typedef struct BTreeNode {

int \*keys;

int t;

struct BTreeNode \*\*C;

int n;

int leaf;

} BTreeNode;

BTreeNode\* createNode(int t, int leaf) {

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

newNode->t = t;

newNode->leaf = leaf;

newNode->keys = (int\*)malloc((2 \* t - 1) \* sizeof(int));

newNode->C = (BTreeNode\*\*)malloc(2 \* t \* sizeof(BTreeNode\*));

newNode->n = 0;

return newNode;

}

void splitChild(BTreeNode \*parent, int i, BTreeNode \*child) {

BTreeNode \*newChild = createNode(child->t, child->leaf);

newChild->n = MIN\_DEGREE - 1;

for (int j = 0; j < MIN\_DEGREE - 1; j++)

newChild->keys[j] = child->keys[j + MIN\_DEGREE];

if (!child->leaf) {

for (int j = 0; j < MIN\_DEGREE; j++)

newChild->C[j] = child->C[j + MIN\_DEGREE];

}

child->n = MIN\_DEGREE - 1;

for (int j = parent->n; j >= i + 1; j--)

parent->C[j + 1] = parent->C[j];

parent->C[i + 1] = newChild;

for (int j = parent->n - 1; j >= i; j--)

parent->keys[j + 1] = parent->keys[j];

parent->keys[i] = child->keys[MIN\_DEGREE - 1];

parent->n++;

}

void insertNonFull(BTreeNode \*node, int key) {

int i = node->n - 1;

if (node->leaf) {

while (i >= 0 && key < node->keys[i]) {

node->keys[i + 1] = node->keys[i];

i--;

}

node->keys[i + 1] = key;

node->n++;

} else {

while (i >= 0 && key < node->keys[i])

i--;

if (node->C[i + 1]->n == 2 \* MIN\_DEGREE - 1) {

splitChild(node, i + 1, node->C[i + 1]);

if (key > node->keys[i + 1])

i++;

}

insertNonFull(node->C[i + 1], key);

}

}

void insert(BTreeNode \*\*root, int key) {

if ((\*root)->n == 2 \* MIN\_DEGREE - 1) {

BTreeNode \*newRoot = createNode((\*root)->t, 0);

newRoot->C[0] = \*root;

splitChild(newRoot, 0, \*root);

insertNonFull(newRoot, key);

\*root = newRoot;

} else {

insertNonFull(\*root, key);

}

}

void traverse(BTreeNode \*root) {

int i;

for (i = 0; i < root->n; i++) {

if (!root->leaf)

traverse(root->C[i]);

printf("%d ", root->keys[i]);

}

if (!root->leaf)

traverse(root->C[i]);

}

int main() {

BTreeNode \*root = createNode(MIN\_DEGREE, 1);

insert(&root, 10);

insert(&root, 20);

insert(&root, 5);

insert(&root, 6);

insert(&root, 12);

insert(&root, 30);

insert(&root, 7);

insert(&root, 17);

printf("Traversal of the B-tree is: ");

traverse(root);

return 0;

}

Output:  


1. Write a C program to implement B+ Tree:

#include <stdio.h>

#include <stdlib.h>

#define MAX 3 // Maximum degree of B+ Tree

typedef struct BPlusNode {

int keys[MAX - 1];

struct BPlusNode \*children[MAX];

int numKeys;

int isLeaf;

} BPlusNode;

BPlusNode\* createNode(int isLeaf) {

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

newNode->isLeaf = isLeaf;

newNode->numKeys = 0;

for (int i = 0; i < MAX; i++) newNode->children[i] = NULL;

return newNode;

}

void insertNonFull(BPlusNode\* node, int key) {

int i = node->numKeys - 1;

if (node->isLeaf) {

while (i >= 0 && key < node->keys[i]) {

node->keys[i + 1] = node->keys[i];

i--;

}

node->keys[i + 1] = key;

node->numKeys++;

} else {

while (i >= 0 && key < node->keys[i]) i--;

i++;

if (node->children[i]->numKeys == MAX - 1) {

// Split child

BPlusNode\* newChild = createNode(node->children[i]->isLeaf);

for (int j = 0; j < MAX / 2; j++) {

newChild->keys[j] = node->children[i]->keys[j + MAX / 2];

}

newChild->numKeys = MAX / 2;

node->children[i]->numKeys = MAX / 2;

for (int j = node->numKeys; j >= i + 1; j--) {

node->children[j + 1] = node->children[j];

}

node->children[i + 1] = newChild;

node->keys[i] = node->children[i]->keys[MAX / 2 - 1];

node->numKeys++;

insertNonFull(node->children[i], key);

} else {

insertNonFull(node->children[i], key);

}

}

}

void insert(BPlusNode\*\* root, int key) {

if ((\*root)->numKeys == MAX - 1) {

BPlusNode\* newRoot = createNode(0);

newRoot->children[0] = \*root;

\*root = newRoot;

insertNonFull(newRoot, key);

} else {

insertNonFull(\*root, key);

}

}

void traverse(BPlusNode\* node) {

for (int i = 0; i < node->numKeys; i++) {

if (!node->isLeaf) traverse(node->children[i]);

printf("%d ", node->keys[i]);

}

if (!node->isLeaf) traverse(node->children[node->numKeys]);

}

int main() {

BPlusNode\* root = createNode(1);

insert(&root, 10);

insert(&root, 20);

insert(&root, 5);

insert(&root, 6);

insert(&root, 12);

printf("Traversal of B+ Tree: ");

traverse(root);

return 0;

}

Output:

