**DAY 11 LAB:**

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

#include <stdio.h>

#include <stdlib.h>

typedef struct Node {

int data;

struct Node\* left;

struct Node\* right;

} Node;

Node\* createNode(int data) {

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

newNode->data = data;

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

return newNode;

}

Node\* insertNode(Node\* root, int data) {

if (root == NULL) {

root = createNode(data);

} else if (data < root->data) {

root->left = insertNode(root->left, data);

} else if (data > root->data) {

root->right = insertNode(root->right, data);

}

return root;

}

int findMin(Node\* root) {

while (root->left != NULL) {

root = root->left;

}

return root->data;

}

int findMax(Node\* root) {

while (root->right != NULL) {

root = root->right;

}

return root->data;

}

Node\* searchNode(Node\* root, int data) {

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

return root;

} else if (data < root->data) {

return searchNode(root->left, data);

} else {

return searchNode(root->right, data);

}

}

int main() {

Node\* root = NULL;

root = insertNode(root, 50);

root = insertNode(root, 30);

root = insertNode(root, 20);

root = insertNode(root, 40);

root = insertNode(root, 70);

root = insertNode(root, 60);

root = insertNode(root, 80);

int searchNum = 40;

Node\* searchResult = searchNode(root, searchNum);

if (searchResult != NULL) {

printf("Found %d in the BST.\n", searchNum);

} else {

printf("%d not found in the BST.\n", searchNum);

}

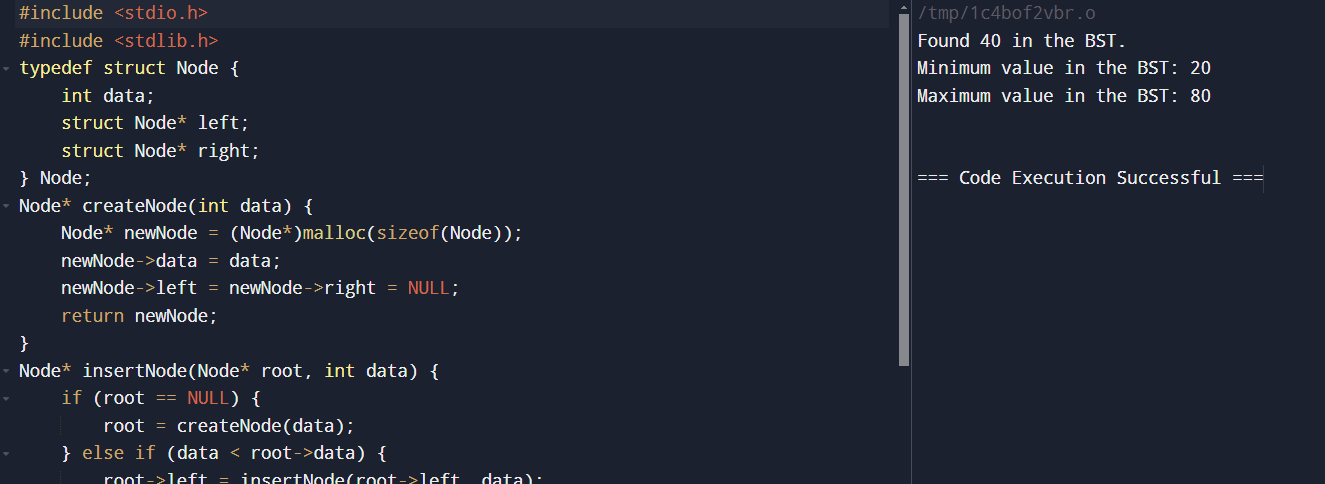
int minVal = findMin(root);

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

int maxVal = findMax(root);

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

}



**2.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.

Code:

#include <stdio.h>

#include <stdlib.h>

typedef struct AVLNode {

int key;

struct AVLNode \*left;

struct AVLNode \*right;

int height;

} AVLNode;

int height(AVLNode \*N) {

if (N == NULL)

return 0;

return N->height;

}

int max(int a, int b) {

return (a > b) ? a : b;

}

AVLNode\* newNode(int key) {

AVLNode\* node = (AVLNode\*)malloc(sizeof(AVLNode));

node->key = key;

node->left = NULL;

node->right = NULL;

node->height = 1;

return(node);

}

AVLNode\* rightRotate(AVLNode \*y) {

AVLNode \*x = y->left;

AVLNode \*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;

}

AVLNode\* leftRotate(AVLNode \*x) {

AVLNode \*y = x->right;

AVLNode \*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(AVLNode \*N) {

if (N == NULL)

return 0;

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

}

AVLNode\* insert(AVLNode\* 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;

}

AVLNode\* minValueNode(AVLNode\* node) {

AVLNode\* current = node;

while (current->left != NULL)

current = current->left;

return current;

}

AVLNode\* deleteNode(AVLNode\* 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)) {

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

if (temp == NULL) {

temp = root;

root = NULL;

} else

\*root = \*temp;

free(temp);

} else {

AVLNode\* 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;

}

AVLNode\* search(AVLNode\* root, int key) {

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

return root;

if (root->key < key)

return search(root->right, key);

return search(root->left, key);

}

void printInOrder(AVLNode \*root) {

if (root != NULL) {

printInOrder(root->left);

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

printInOrder(root->right);

}

}

int main() {

AVLNode \*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);

// Print the AVL Tree

printf("In-order traversal of the constructed AVL tree is:\n");

printInOrder(root);

printf("\n");

// Delete an element

printf("Deleting 10\n");

root = deleteNode(root, 10);

printf("In-order traversal after deletion of 10:\n");

printInOrder(root);

printf("\n");

// Search for a key

int key = 25;

AVLNode\* result = search(root, key);

if (result != NULL)

printf("Found %d in the AVL tree.\n", key);

else

printf("%d not found in the AVL tree.\n", key);

}



**3.Write a C program to implement Red black tree.**

#include <stdio.h>

#include <stdlib.h>

typedef enum { RED, BLACK } Color;

typedef struct Node {

int key;

Color color;

struct Node\* left;

struct Node\* right;

struct Node\* parent;

} Node;

Node\* createNode(int key) {

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

newNode->key = key;

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\* x) {

Node\* y = x->left;

x->left = y->right;

if (y->right != NULL) {

y->right->parent = x;

}

y->parent = x->parent;

if (x->parent == NULL) {

\*root = y;

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

x->parent->right = y;

} else {

x->parent->left = y;

}

y->right = x;

x->parent = y;

}

void fixInsert(Node\*\* root, Node\* z) {

while (z->parent != NULL && z->parent->color == RED) {

if (z->parent == z->parent->parent->left) {

Node\* y = z->parent->parent->right;

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

z->parent->color = BLACK;

y->color = BLACK;

z->parent->parent->color = RED;

z = z->parent->parent;

} else {

if (z == z->parent->right) {

z = z->parent;

leftRotate(root, z);

}

z->parent->color = BLACK;

z->parent->parent->color = RED;

rightRotate(root, z->parent->parent);

}

} else {

Node\* y = z->parent->parent->left;

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

z->parent->color = BLACK;

y->color = BLACK;

z->parent->parent->color = RED;

z = z->parent->parent;

} else {

if (z == z->parent->left) {

z = z->parent;

rightRotate(root, z);

}

z->parent->color = BLACK;

z->parent->parent->color = RED;

leftRotate(root, z->parent->parent);

}

}

}

(\*root)->color = BLACK;

}

void insertNode(Node\*\* root, int key) {

Node\* z = createNode(key);

Node\* y = NULL;

Node\* x = \*root;

while (x != NULL) {

y = x;

if (z->key < x->key) {

x = x->left;

} else {

x = x->right;

}

}

z->parent = y;

if (y == NULL) {

\*root = z;

} else if (z->key < y->key) {

y->left = z;

} else {

y->right = z;

}

fixInsert(root, z);

}

void printTree(Node\* root, int indent) {

if (root != NULL) {

printTree(root->right, indent + 4);

for (int i = 0; i < indent; i++) {

printf(" ");

}

printf("%d (%s)\n", root->key, root->color == RED ? "RED" : "BLACK");

printTree(root->left, indent + 4);

}

}

int main() {

Node\* root = NULL;

insertNode(&root, 10);

insertNode(&root, 20);

insertNode(&root, 30);

insertNode(&root, 40);

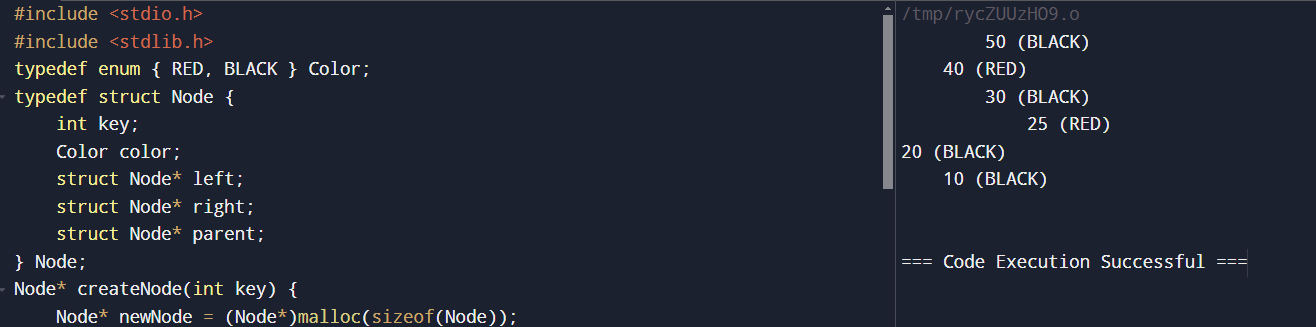
insertNode(&root, 50);

insertNode(&root, 25);

printTree(root, 0);

return 0;

}



**4.Write a C program to implement B Tree.**

#include <stdio.h>

#include <stdlib.h>

#define T 3

typedef struct BTreeNode {

int keys[2 \* T - 1];

struct BTreeNode \*children[2 \* T];

int numKeys;

int leaf;

} BTreeNode;

BTreeNode\* createNode(int leaf) {

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

node->numKeys = 0;

node->leaf = leaf;

for (int i = 0; i < 2 \* T; i++)

node->children[i] = NULL;

return node;

}

// Insert a non-full node

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

int i = node->numKeys - 1;

if (node->leaf) {

// Find location to insert the new key

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

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

i--;

}

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

node->numKeys++;

} else {

// Find child which is going to have the new key

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

i--;

i++;

if (node->children[i]->numKeys == 2 \* T - 1) {

// Split the child if it's full

BTreeNode \*child = node->children[i];

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

int median = T - 1;

// Copy the second half of keys to newChild

for (int j = 0; j < T - 1; j++)

newChild->keys[j] = child->keys[j + T];

if (!child->leaf)

for (int j = 0; j < T; j++)

newChild->children[j] = child->children[j + T];

child->numKeys = T - 1;

newChild->numKeys = T - 1;

// Insert median into parent

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

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

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

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

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

node->keys[i] = child->keys[median];

node->numKeys++;

// Recur for the child

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

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

} else {

// Recur for the child

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

}

}

}

// Insert key into B-Tree

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

BTreeNode \*r = \*root;

if (r->numKeys == 2 \* T - 1) {

// Tree root is full, need to split

BTreeNode \*s = createNode(0);

\*root = s;

s->children[0] = r;

s->leaf = 0;

splitChild(s, 0);

insertNonFull(s, key);

} else {

insertNonFull(r, key);

}

}

// Split a child of the node

void splitChild(BTreeNode \*node, int index) {

BTreeNode \*fullChild = node->children[index];

BTreeNode \*newChild = createNode(fullChild->leaf);

int median = T - 1;

// Copy the second half of the keys and children to newChild

for (int i = 0; i < T - 1; i++)

newChild->keys[i] = fullChild->keys[i + T];

if (!fullChild->leaf)

for (int i = 0; i < T; i++)

newChild->children[i] = fullChild->children[i + T];

fullChild->numKeys = T - 1;

newChild->numKeys = T - 1;

// Insert median into parent

for (int i = node->numKeys; i >= index; i--)

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

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

for (int i = node->numKeys - 1; i >= index; i--)

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

node->keys[index] = fullChild->keys[median];

node->numKeys++;

}

// Print the B-Tree (In-order traversal)

void printTree(BTreeNode \*root, int level) {

if (root == NULL)

return;

printf("Level %d: ", level);

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

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

printf("\n");

if (!root->leaf) {

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

printTree(root->children[i], level + 1);

}

}

int main() {

BTreeNode \*root = createNode(1);

insert(&root, 10);

insert(&root, 20);

insert(&root, 5);

insert(&root, 6);

insert(&root, 15);

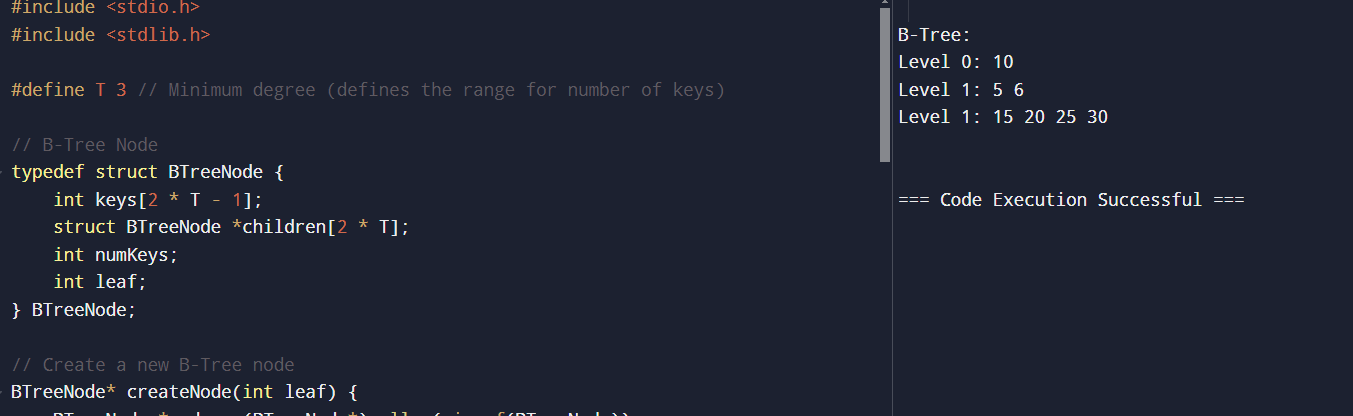
insert(&root, 30);

insert(&root, 25);

printf("B-Tree:\n");

printTree(root, 0);

}



**5.Write a C program to implement B+ Tree.**

#include <stdio.h>

#include <stdlib.h>

#define T 3

typedef struct BPlusTreeNode {

int keys[2 \* T - 1];

struct BPlusTreeNode \*children[2 \* T];

struct BPlusTreeNode \*next;

int numKeys;

int leaf;

} BPlusTreeNode;

BPlusTreeNode\* createNode(int leaf) {

BPlusTreeNode \*node = (BPlusTreeNode\*)malloc(sizeof(BPlusTreeNode));

node->numKeys = 0;

node->leaf = leaf;

node->next = NULL;

for (int i = 0; i < 2 \* T; i++)

node->children[i] = NULL;

return node;

}

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

int i = node->numKeys - 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->numKeys++;

} else {

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

i--;

i++;

if (node->children[i]->numKeys == 2 \* T - 1) {

splitChild(node, i);

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

i++;

}

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

}

}

void splitChild(BPlusTreeNode \*node, int index) {

BPlusTreeNode \*fullChild = node->children[index];

BPlusTreeNode \*newChild = createNode(fullChild->leaf);

int median = T - 1;

for (int i = 0; i < T - 1; i++)

newChild->keys[i] = fullChild->keys[i + T];

if (!fullChild->leaf)

for (int i = 0; i < T; i++)

newChild->children[i] = fullChild->children[i + T];

fullChild->numKeys = T - 1;

newChild->numKeys = T - 1;

for (int i = node->numKeys; i >= index; i--)

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

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

for (int i = node->numKeys - 1; i >= index; i--)

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

node->keys[index] = fullChild->keys[median];

node->numKeys++;

}

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

BPlusTreeNode \*r = \*root;

if (r->numKeys == 2 \* T - 1) {

BPlusTreeNode \*s = createNode(0);

\*root = s;

s->children[0] = r;

splitChild(s, 0);

insertNonFull(s, key);

} else {

insertNonFull(r, key);

}

}

void printLeaves(BPlusTreeNode \*root) {

BPlusTreeNode \*current = root;

while (current && !current->leaf)

current = current->children[0]; while (current) {

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

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

current = current->next;

}

printf("\n");

}

int main() {

BPlusTreeNode \*root = createNode(1);

insert(&root, 10);

insert(&root, 20);

insert(&root, 5);

insert(&root, 6);

insert(&root, 15);

insert(&root, 30);

insert(&root, 25);

printf("B+ Tree Leaves:\n");

printLeaves(root);

}

