|  |
| --- |
| **SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES** |
| **COMPUTER SCIENCE AND ENGINEERING PROGRAMME** |

**SUB CODE: CSA0392 SUB NAME: Data Structures for Hashing Techniques**

**LIST OF PROGRAMS**

**DATE : 14.08.2024**

**Lab Questions to be practiced with test cases**

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

Answer:

#include <stdio.h>

#include <stdlib.h>

// Define the structure for a node in the BST

typedef struct Node {

int data;

struct Node\* left;

struct Node\* right;

} Node;

// Function to create a new node

Node\* createNode(int data) {

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

newNode->data = data;

newNode->left = NULL;

newNode->right = NULL;

return newNode;

}

// Function to insert a new node into the BST

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

if (root == NULL) {

return createNode(data);

}

if (data < root->data) {

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

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

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

}

return root;

}

// Function to search for a value in the BST

Node\* search(Node\* root, int value) {

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

return root;

}

if (value < root->data) {

return search(root->left, value);

} else {

return search(root->right, value);

}

}

// Function to find the minimum value in the BST

Node\* findMin(Node\* root) {

while (root && root->left != NULL) {

root = root->left;

}

return root;

}

// Function to find the maximum value in the BST

Node\* findMax(Node\* root) {

while (root && root->right != NULL) {

root = root->right;

}

return root;

}

// Function to display the BST in-order

void inorder(Node\* root) {

if (root != NULL) {

inorder(root->left);

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

inorder(root->right);

}

}

int main() {

Node\* root = NULL;

int choice, value;

Node\* result;

// Insert nodes into the BST

root = insert(root, 50);

insert(root, 30);

insert(root, 70);

insert(root, 20);

insert(root, 40);

insert(root, 60);

insert(root, 80);

while (1) {

printf("\nBST Operations Menu:\n");

printf("1. Search for a value\n");

printf("2. Find Minimum value\n");

printf("3. Find Maximum value\n");

printf("4. Display BST (In-order)\n");

printf("5. EXIT\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter the value to search: ");

scanf("%d", &value);

result = search(root, value);

if (result != NULL) {

printf("Value %d found in the BST.\n", value);

} else {

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

}

break;

case 2:

result = findMin(root);

if (result != NULL) {

printf("Minimum value in the BST is %d.\n", result->data);

} else {

printf("BST is empty.\n");

}

break;

case 3:

result = findMax(root);

if (result != NULL) {

printf("Maximum value in the BST is %d.\n", result->data);

} else {

printf("BST is empty.\n");

}

break;

case 4:

printf("BST elements (In-order): ");

inorder(root);

printf("\n");

break;

case 5:

printf("Exiting...\n");

exit(0);

default:

printf("Invalid choice. Please try again.\n");

}

}

return 0;

}

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.

Answer:

#include <stdio.h>

#include <stdlib.h>

// Define the structure for an AVL tree node

typedef struct Node {

int data;

struct Node\* left;

struct Node\* right;

int height;

} Node;

// Function to create a new AVL tree node

Node\* createNode(int data) {

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

newNode->data = data;

newNode->left = NULL;

newNode->right = NULL;

newNode->height = 1; // New node is initially at height 1

return newNode;

}

// Function to get the height of a node

int height(Node\* node) {

return (node == NULL) ? 0 : node->height;

}

// Function to get the balance factor of a node

int getBalance(Node\* node) {

return (node == NULL) ? 0 : height(node->left) - height(node->right);

}

// Function to right rotate a subtree rooted with y

Node\* rightRotate(Node\* y) {

Node\* x = y->left;

Node\* T2 = x->right;

// Perform rotation

x->right = y;

y->left = T2;

// Update heights

y->height = 1 + (height(y->left) > height(y->right) ? height(y->left) : height(y->right));

x->height = 1 + (height(x->left) > height(x->right) ? height(x->left) : height(x->right));

// Return new root

return x;

}

// Function to left rotate a subtree rooted with x

Node\* leftRotate(Node\* x) {

Node\* y = x->right;

Node\* T2 = y->left;

// Perform rotation

y->left = x;

x->right = T2;

// Update heights

x->height = 1 + (height(x->left) > height(x->right) ? height(x->left) : height(x->right));

y->height = 1 + (height(y->left) > height(y->right) ? height(y->left) : height(y->right));

// Return new root

return y;

}

// Function to insert a node into an AVL tree

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

// Perform the normal BST insert

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

if (data < node->data) {

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

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

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

} else {

return node; // Duplicate values are not allowed

}

// Update height of this ancestor node

node->height = 1 + (height(node->left) > height(node->right) ? height(node->left) : height(node->right));

// Get the balance factor of this ancestor node

int balance = getBalance(node);

// If this node becomes unbalanced, then there are 4 cases

// Left Left Case

if (balance > 1 && data < node->left->data) {

return rightRotate(node);

}

// Right Right Case

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

return leftRotate(node);

}

// Left Right Case

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

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

return rightRotate(node);

}

// Right Left Case

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

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

return leftRotate(node);

}

// Return the (unchanged) node pointer

return node;

}

// Function to find the node with the minimum value

Node\* minNode(Node\* node) {

Node\* current = node;

while (current->left != NULL) {

current = current->left;

}

return current;

}

// Function to delete a node from an AVL tree

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

// STEP 1: PERFORM STANDARD BST DELETE

if (root == NULL) return root;

// If the data to be deleted is smaller than the root's data, then it lies in the left subtree

if (data < root->data) {

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

}

// If the data to be deleted is greater than the root's data, then it lies in the right subtree

else if (data > root->data) {

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

}

// If data is same as root's data, then this is the node to be deleted

else {

// Node with only one child or no child

if (root->left == NULL) {

Node\* temp = root->right;

free(root);

return temp;

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

Node\* temp = root->left;

free(root);

return temp;

}

// Node with two children: Get the inorder successor (smallest in the right subtree)

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

// Copy the inorder successor's data to this node

root->data = temp->data;

// Delete the inorder successor

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

}

// STEP 2: UPDATE HEIGHT OF CURRENT NODE

root->height = 1 + (height(root->left) > height(root->right) ? height(root->left) : height(root->right));

// STEP 3: GET THE BALANCE FACTOR OF THIS NODE (to check whether this node became unbalanced)

int balance = getBalance(root);

// If this node becomes unbalanced, then there are 4 cases

// Left Left Case

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

return rightRotate(root);

}

// Left Right Case

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

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

return rightRotate(root);

}

// Right Right Case

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

return leftRotate(root);

}

// Right Left Case

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

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

return leftRotate(root);

}

// Return the (unchanged) node pointer

return root;

}

// Function to search for a key in the AVL tree

Node\* search(Node\* 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 data

if (root->data < key) {

return search(root->right, key);

}

// Key is smaller than root's data

return search(root->left, key);

}

// Function to display the AVL tree in-order

void inorder(Node\* root) {

if (root != NULL) {

inorder(root->left);

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

inorder(root->right);

}

}

int main() {

Node\* root = NULL;

int choice, value;

while (1) {

printf("\nAVL Tree Operations Menu:\n");

printf("1. Insert an element\n");

printf("2. Delete an element\n");

printf("3. Search for a key\n");

printf("4. Display AVL Tree (In-order)\n");

printf("5. EXIT\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter the value to insert: ");

scanf("%d", &value);

root = insert(root, value);

printf("Inserted %d into the AVL tree.\n", value);

break;

case 2:

printf("Enter the value to delete: ");

scanf("%d", &value);

root = deleteNode(root, value);

printf("Deleted %d from the AVL tree.\n", value);

break;

case 3:

printf("Enter the key to search: ");

scanf("%d", &value);

Node\* result = search(root, value);

if (result != NULL) {

printf("Key %d found in the AVL tree.\n", value);

} else {

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

}

break;

case 4:

printf("AVL Tree elements (In-order): ");

inorder(root);

printf("\n");

break;

case 5:

printf("Exiting...\n");

exit(0);

default:

printf("Invalid choice. Please try again.\n");

}

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

Answer:

#include <stdio.h>

#include <stdlib.h>

// Define the color values

#define RED 0

#define BLACK 1

// Define the structure for a Red-Black Tree node

typedef struct Node {

int data;

int color; // RED or BLACK

struct Node\* left;

struct Node\* right;

struct Node\* parent;

} Node;

// Create a new node with the given data

Node\* createNode(int data) {

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

newNode->data = data;

newNode->color = RED; // New nodes are initially red

newNode->left = NULL;

newNode->right = NULL;

newNode->parent = NULL;

return newNode;

}

// Function prototypes

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

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

void fixInsert(Node\*\* root, Node\* node);

Node\* insert(Node\* root, int data);

void inorderTraversal(Node\* root);

Node\* search(Node\* root, int data);

// Perform left rotation on the subtree rooted with x

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;

}

// Perform right rotation on the subtree rooted with 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->right) {

y->parent->right = x;

} else {

y->parent->left = x;

}

x->right = y;

y->parent = x;

}

// Fix the Red-Black Tree after insertion of a node

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

Node\* parent = NULL;

Node\* grandparent = NULL;

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

parent = node->parent;

grandparent = parent->parent;

if (parent == grandparent->left) {

Node\* uncle = grandparent->right;

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

parent->color = BLACK;

uncle->color = BLACK;

grandparent->color = RED;

node = grandparent;

} else {

if (node == parent->right) {

node = parent;

leftRotate(root, node);

}

parent->color = BLACK;

grandparent->color = RED;

rightRotate(root, grandparent);

}

} else {

Node\* uncle = grandparent->left;

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

parent->color = BLACK;

uncle->color = BLACK;

grandparent->color = RED;

node = grandparent;

} else {

if (node == parent->left) {

node = parent;

rightRotate(root, node);

}

parent->color = BLACK;

grandparent->color = RED;

leftRotate(root, grandparent);

}

}

}

(\*root)->color = BLACK;

}

// Insert a new node into the Red-Black Tree

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

Node\* node = createNode(data);

Node\* y = NULL;

Node\* x = root;

// Find the appropriate position to insert the new node

while (x != NULL) {

y = x;

if (node->data < x->data) {

x = x->left;

} else {

x = x->right;

}

}

node->parent = y;

if (y == NULL) {

root = node;

} else if (node->data < y->data) {

y->left = node;

} else {

y->right = node;

}

// Fix the Red-Black Tree

fixInsert(&root, node);

return root;

}

// In-order traversal to display the Red-Black Tree

void inorderTraversal(Node\* root) {

if (root != NULL) {

inorderTraversal(root->left);

printf("%d(%s) ", root->data, root->color == RED ? "R" : "B");

inorderTraversal(root->right);

}

}

// Search for a value in the Red-Black Tree

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

while (root != NULL && data != root->data) {

if (data < root->data) {

root = root->left;

} else {

root = root->right;

}

}

return root;

}

int main() {

Node\* root = NULL;

int choice, value;

while (1) {

printf("\nRed-Black Tree Operations Menu:\n");

printf("1. Insert an element\n");

printf("2. Display Red-Black Tree (In-order)\n");

printf("3. Search for a key\n");

printf("4. EXIT\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter the value to insert: ");

scanf("%d", &value);

root = insert(root, value);

printf("Inserted %d into the Red-Black Tree.\n", value);

break;

case 2:

printf("Red-Black Tree elements (In-order): ");

inorderTraversal(root);

printf("\n");

break;

case 3:

printf("Enter the key to search: ");

scanf("%d", &value);

Node\* result = search(root, value);

if (result != NULL) {

printf("Key %d found in the Red-Black Tree.\n", value);

} else {

printf("Key %d not found in the Red-Black Tree.\n", value);

}

break;

case 4:

printf("Exiting...\n");

exit(0);

default:

printf("Invalid choice. Please try again.\n");

}

}

return 0;

}

1. Write a C program to implement B Tree.

Answer:

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

Answer:

#include <stdio.h>

#include <stdlib.h>

#define T 3 // Minimum degree (defines the range for number of keys)

// Define the structure for a B+ Tree node

typedef struct BPlusNode {

int \*keys;

struct BPlusNode \*\*children;

struct BPlusNode \*next; // Pointer to the next leaf node

int numKeys;

int isLeaf;

} BPlusNode;

// Function prototypes

BPlusNode\* createNode(int isLeaf);

void traverse(BPlusNode\* root);

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

void splitChild(BPlusNode\* parent, int i, BPlusNode\* fullChild);

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

void display(BPlusNode\* root, int level);

void rangeSearch(BPlusNode\* root, int low, int high);

// Create a new B+ Tree node

BPlusNode\* createNode(int isLeaf) {

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

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

newNode->children = (BPlusNode\*\*)malloc(2 \* T \* sizeof(BPlusNode\*));

newNode->next = NULL;

newNode->numKeys = 0;

newNode->isLeaf = isLeaf;

return newNode;

}

// Traverse and print the B+ Tree (in-order traversal)

void traverse(BPlusNode\* root) {

BPlusNode\* current = root;

while (current) {

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

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

}

current = current->next;

}

printf("\n");

}

// Insert a new key into a non-full node

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 == 2 \* T - 1) {

splitChild(node, i, node->children[i]);

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

i++;

}

}

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

}

}

// Split a full child node into two nodes

void splitChild(BPlusNode\* parent, int i, BPlusNode\* fullChild) {

BPlusNode\* newChild = createNode(fullChild->isLeaf);

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

parent->keys[i] = fullChild->keys[T - 1];

parent->numKeys++;

newChild->numKeys = T - 1;

fullChild->numKeys = T - 1;

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

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

}

if (!fullChild->isLeaf) {

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

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

}

}

if (fullChild->isLeaf) {

newChild->next = fullChild->next;

fullChild->next = newChild;

}

}

// Insert a key into the B+ Tree

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

BPlusNode\* r = \*root;

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

BPlusNode\* s = createNode(0);

\*root = s;

s->children[0] = r;

splitChild(s, 0, r);

insertNonFull(s, key);

} else {

insertNonFull(r, key);

}

}

// Display the B+ Tree structure

void display(BPlusNode\* root, int level) {

if (root) {

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

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

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

}

printf("\n");

if (!root->isLeaf) {

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

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

}

}

}

}

// Range search in the B+ Tree

void rangeSearch(BPlusNode\* root, int low, int high) {

BPlusNode\* current = root;

while (current) {

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

if (current->keys[i] >= low && current->keys[i] <= high) {

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

}

}

current = current->next;

}

printf("\n");

}

// Main function to demonstrate B+ Tree operations

int main() {

BPlusNode\* root = createNode(1);

// Inserting elements into the B+ Tree

insert(&root, 10);

insert(&root, 20);

insert(&root, 5);

insert(&root, 6);

insert(&root, 15);

insert(&root, 30);

insert(&root, 25);

insert(&root, 35);

printf("B+ Tree elements in sorted order: ");

traverse(root);

printf("Displaying B+ Tree structure:\n");

display(root, 0);

printf("Range search between 10 and 30: ");

rangeSearch(root, 10, 30);

return 0;

}