**AIM:**

To implement a Binary Search Tree and perform the following operations.

1. Insert
2. Delete
3. Search
4. Display

**ALGORITHM:**

1. **INSERTION**

Step 1: Start at the root node:

* If the tree is empty, the new node becomes the root.

Step 2: Traverse the tree:

* Compare the value of the new node with the value of the current node.
* If the new node's value is less than the current node's value, go to the left child.
* If the new node's value is greater than the current node's value, go to the right child.

Step 3: Insert the node:

* If you reach a null reference (i.e., there is no left or right child), insert the new node at that position.
* If there is already a node at that position, repeat step 2.

1. **DELETION**

Step 1: Search for the Node to Delete:

* Start from the root of the BST.
* Compare the key you want to delete with the key of the current node.
* If the key is smaller, move to the left child.
* If the key is larger, move to the right child.
* Repeat this process until you find the node with the matching key.

Step 2: Once the Node is Found:

* You have three main cases to consider: the node has no children, the node has one child, or the node has two children.

 Case 1**:** Node with No Children (Leaf Node):

* Simply remove the node from the tree.
* If the node is a left child, set its parent's left child to None.
* If the node is a right child, set its parent's right child to None.

 Case 2**:** Node with One Child:

* Remove the node and link its parent directly to its child.
* If the node is a left child, set its parent's left child to the node's only child.
* If the node is a right child, set its parent's right child to the node's only child.

 Case 3**:** Node with Two Children:

* Find the node's in-order successor. This is the smallest node in the right subtree.
  + To find the in-order successor, go to the right child of the node and then keep moving to the leftmost child until you reach a leaf node.
* Replace the value of the node to be deleted with the value of the in-order successor.
* Delete the in-order successor, which will have at most one child (it cannot have two children).

1. **SEARCHING**

Step 1: Start at the root of the tree.

Step 2: Compare the target value with the value of the current node:

* If the target value equals the current node's value, you have found the node.
* If the target value is less than the current node's value, move to the left child.
* If the target value is greater than the current node's value, move to the right child.

Step 3: Repeat step 2 until you either find the node or reach a null pointer (indicating that the value is not in the tree).

1. **DISPLAY**

Step 1: Start at the root node of the BST.

Step 2: Check if the current node is NULL. If it is, then return because there are no nodes to display.

Step 3: Recursively traverse the left subtree:

* Call the display function with the left child of the current node.
* This step ensures that all nodes in the left subtree are displayed first, following the left-to-right order.

Step 4: Display the data of the current node:

* Print the value stored in the current node.
* This step ensures that the data of the current node is displayed.

Step 5: Recursively traverse the right subtree:

* Call the display function with the right child of the current node.
* This step ensures that all nodes in the right subtree are displayed after the current node, following the left-to-right order.

**CODE:**

#include <stdio.h>

#include <stdlib.h>

struct node {

int data;

struct node \*left;

struct node \*right;

};

struct node \*createNode(int value) {

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

newNode->data = value;

newNode->left = NULL;

newNode->right = NULL;

return newNode;

}

struct node \*insert(struct node \*root, int value) {

if (root == NULL)

return createNode(value);

if (value < root->data)

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

else if (value > root->data)

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

return root;

}

struct node \*search(struct 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);

}

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

struct node \*current = node;

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

current = current->left;

return current;

}

struct node \*deleteNode(struct node \*root, int value) {

if (root == NULL)

return root;

if (value < root->data)

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

else if (value > root->data)

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

else {

if (root->left == NULL) {

struct node \*temp = root->right;

free(root);

return temp;

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

struct node \*temp = root->left;

free(root);

return temp;

}

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

root->data = temp->data;

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

}

return root;

}

void inorderTraversal(struct node \*root) {

if (root != NULL) {

inorderTraversal(root->left);

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

inorderTraversal(root->right);

}

}

int main() {

struct node \*root = NULL;

int choice, value;

do {

printf("\nBinary Search Tree Operations:\n");

printf("1. Insert\n");

printf("2. Delete\n");

printf("3. Search\n");

printf("4. Display\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);

break;

case 2:

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

scanf("%d", &value);

root = deleteNode(root, value);

break;

case 3:

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

scanf("%d", &value);

if (search(root, value) != NULL)

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

else

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

break;

case 4:

printf("BST elements in inorder traversal: ");

inorderTraversal(root);

printf("\n");

break;

case 5:

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

break;

default:

printf("Invalid choice! Please enter a valid option.\n");

}

} while (choice != 5);

return 0;

}

**OUTPUT:**

Binary Search Tree Operations:

1. Insert

2. Delete

3. Search

4. Display

5. Exit

Enter your choice: 1

Enter the value to insert: 50

Binary Search Tree Operations:

1. Insert

2. Delete

3. Search

4. Display

5. Exit

Enter your choice: 1

Enter the value to insert: 30

Binary Search Tree Operations:

1. Insert

2. Delete

3. Search

4. Display

5. Exit

Enter your choice: 1

Enter the value to insert: 70

Binary Search Tree Operations:

1. Insert

2. Delete

3. Search

4. Display

5. Exit

Enter your choice: 1

Enter the value to insert: 20

Binary Search Tree Operations:

1. Insert

2. Delete

3. Search

4. Display

5. Exit

Enter your choice: 1

Enter the value to insert: 40

Binary Search Tree Operations:

1. Insert

2. Delete

3. Search

4. Display

5. Exit

Enter your choice: 4

BST elements in inorder traversal: 20 30 40 50 70

Binary Search Tree Operations:

1. Insert

2. Delete

3. Search

4. Display

5. Exit

Enter your choice: 3

Enter the value to search: 30

Value 30 found in the tree.

Binary Search Tree Operations:

1. Insert

2. Delete

3. Search

4. Display

5. Exit

Enter your choice: 2

Enter the value to delete: 30

Binary Search Tree Operations:

1. Insert

2. Delete

3. Search

4. Display

5. Exit

Enter your choice: 4

BST elements in inorder traversal: 20 40 50 70

Binary Search Tree Operations:

1. Insert

2. Delete

3. Search

4. Display

5. Exit

Enter your choice: 5

Exiting...

**RESULT:**

Hence the program for Insert, Delete, Search, Display in a binary search tree is implemented and performed successfully.