1) 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 = NULL;
   newNode->right = NULL;
   return newNode;
}
Node* insert(Node* root, int data)
{
   if (root == NULL)
   {
      return createNode(data);
   }
   if (data < root->data)
   {
      root->left = insert(root->left, data);
   }
   else
   {
      root->right = insert(root->right, data);
   }
```

```
    return root;
}
int findMin(Node* root)
{
    Node* current = root;
    while (current && current->left != NULL)
    {
        current = current->left;
    }
    return current ? current->data : -1;
}
int findMax(Node* root)
{
    Node* current = root;
    while (current && current->right != NULL)
    {
        current = current->right;
    }
    return current ? current->data : -1;
}
void inorderTraversal(Node* root)
{
    if (root != NULL)
    {
        inorderTraversal(root->left);
        printf("%d ", root->data);
        inorderTraversal(root->right);
    }
}
int main()
{
```

```
    Node* root = NULL;
    root = insert(root, 15);
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 8);
    root = insert(root, 12);
    root = insert(root, 17);
    root = insert(root, 25);
    printf("Inorder traversal of the BST: ");
    inorderTraversal(root);
    printf("\n");
    printf("Minimum value in the BST: %d\n", findMin(root));
    printf("Maximum value in the BST: %d\n", findMax(root));
    return 0;
}
```

OUT PUT:

```
/tmp/eiIA1Sp2VH.o
Inorder traversal of the BST: 8 10 12 15 17 20 25
Minimum value in the BST: 8
Maximum value in the BST: 25


=== Code Execution Successful ===
```

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.

```
#include <stdio.h>
#include <stdlib.h>
```

```
typedef struct AVLNode
{
   int key;
   struct AVLNode *left, *right;
   int height;
} AVLNode;
AVLNode* createNode(int key)
{
   AVLNode *node = (AVLNode*)malloc(sizeof(AVLNode));
   node->key = key;
   node->left = node->right = NULL;
   node->height = 1;
   return node;
}
int height(AVLNode *node)
{
   if (node == NULL)
      return 0;
   return node->height;
}
int getBalance(AVLNode *node)
{
   if (node == NULL)
      return 0;
   return height(node->left) - height(node->right);
}
AVLNode* rightRotate(AVLNode *y)
{
   AVLNode *x = y->left;
   AVLNode *T2 = x->right;
   x->right = y;
```

```
    y->left = T2;
    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 x;
}
AVLNode* leftRotate(AVLNode *x)
{
    AVLNode *y = x->right;
    AVLNode *T2 = y->left;
    y->left = x;
    x->right = T2;
    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 y;
}
AVLNode* insert(AVLNode *node, int key)
{
    if (node == NULL)
        return createNode(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 + (height(node->left) > height(node->right) ? 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* minNode(AVLNode *node)
{
    AVLNode *current = node;
    while (current->left != NULL)
        current = current->left;
    return current;
}
AVLNode* delete(AVLNode *root, int key)
{
    if (root == NULL)
        return root;
    if (key < root->key)
        root->left = delete(root->left, key);
    else if (key > root->key)
        root->right = delete(root->right, key);
    else
    {
        if (root->left == NULL)
```

```
            return root->right;
        else if (root->right == NULL)
            return root->left;
        AVLNode *temp = minNode(root->right);
        root->key = temp->key;
        root->right = delete(root->right, temp->key);
    }
    if (root == NULL)
        return root;
    root->height = 1 + (height(root->left) > height(root->right) ? 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 (key < root->key)
        return search(root->left, key);
    return search(root->right, key);
}
void inOrder(AVLNode *root)
{
    if (root != NULL)
    {
        inOrder(root->left);
        printf("%d ", root->key);
        inOrder(root->right);
    }
}
int main()
{
    AVLNode *root = NULL;
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 30);
    root = insert(root, 15);
    printf("In-order traversal of the AVL tree is:\n");
    inOrder(root);
    printf("\n");
    int searchKey = 15;
    AVLNode *result = search(root, searchKey);
    if (result != NULL)
        printf("Element %d found in the AVL tree.\n", searchKey);
    else
        printf("Element %d not found in the AVL tree.\n", searchKey);
    root = delete(root, 20);
    printf("In-order traversal after deletion of 20:\n");
```

```
    inOrder(root);
    printf("\n");
    return 0;
}
```

OUT PUT:

```
/tmp/7MdfUekphP.o
In-order traversal of the AVL tree is:
10 15 20 30
Element 15 found in the AVL tree.
In-order traversal after deletion of 20:
10 15 30


=== Code Execution Successful ===
```

3) C program to implement Red black tree.

```
#include <stdio.h>
#include <stdlib.h>
typedef enum { RED, BLACK } Color;
typedef struct RBNode
{
    int key;
    Color color;
    struct RBNode *left, *right, *parent;
} RBNode;
RBNode* createNode(int key);
void rotateLeft(RBNode** root, RBNode* node);
void rotateRight(RBNode** root, RBNode* node);
void fixInsertion(RBNode** root, RBNode* node);
```

```
void insert(RBNode** root, int key);
RBNode* search(RBNode* root, int key);
void inOrder(RBNode* root);
void printTree(RBNode* root, int space);
RBNode* createNode(int key)
{
   RBNode* node = (RBNode*)malloc(sizeof(RBNode));
   node->key = key;
   node->left = node->right = node->parent = NULL;
   node->color = RED;
   return node;
}
void rotateLeft(RBNode** root, RBNode* node)
{
   RBNode* rightChild = node->right;
   node->right = rightChild->left;
   if (rightChild->left != NULL)
   {
      rightChild->left->parent = node;
   }
   rightChild->parent = node->parent;
   if (node->parent == NULL)
   {
      *root = rightChild;
   }
   else if (node == node->parent->left)
   {
      node->parent->left = rightChild;
   }
   else
   {
```

```
        node->parent->right = rightChild;
    }
    rightChild->left = node;
    node->parent = rightChild;
}
void rotateRight(RBNode** root, RBNode* node)
{
    RBNode* leftChild = node->left;
    node->left = leftChild->right;
    if (leftChild->right != NULL)
    {
        leftChild->right->parent = node;
    }
    leftChild->parent = node->parent;
    if (node->parent == NULL)
    {
        *root = leftChild;
    }
    else if (node == node->parent->right)
    {
        node->parent->right = leftChild;
    }
    else
    {
        node->parent->left = leftChild;
    }
    leftChild->right = node;
    node->parent = leftChild;
}
void fixInsertion(RBNode** root, RBNode* node)
{
```

```
RBNode* parent = NULL;
RBNode* grandparent = NULL;
while (node != *root && node->parent->color == RED) {
   parent = node->parent;
   grandparent = parent->parent;
   if (parent == grandparent->left)
   {
      RBNode* 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;
            rotateLeft(root, node);
         }
         parent->color = BLACK;
         grandparent->color = RED;
         rotateRight(root, grandparent);
      }
   }
   else
   {
      RBNode* 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;
                    rotateRight(root, node);
                }
                parent->color = BLACK;
                grandparent->color = RED;
                rotateLeft(root, grandparent);
            }
        }
    }
    (*root)->color = BLACK;
}
void insert(RBNode** root, int key)
{
    RBNode* node = createNode(key);
    RBNode* y = NULL;
    RBNode* x = *root;
    while (x != NULL)
    {
        y = x;
        if (node->key < x->key)
        {
```

```
            x = x->left;
        }
        else
        {
            x = x->right;
        }
    }
    node->parent = y;
    if (y == NULL)
    {
        *root = node;
    }
    else if (node->key < y->key)
    {
        y->left = node;
    }
    else
    {
        y->right = node;
    }
    fixInsertion(root, node);
}
RBNode* search(RBNode* root, int key)
{
    while (root != NULL && key != root->key)
    {
        if (key < root->key)
        {
            root = root->left;
        }
        else
```

```
        {
            root = root->right;
        }
    }
    return root;
}
void inOrder(RBNode* root)
{
    if (root != NULL)
    {
        inOrder(root->left);
        printf("%d (%s) ", root->key, root->color == RED ? "RED" : "BLACK");
        inOrder(root->right);
    }
}
void printTree(RBNode* root, int space)
{
    if (root == NULL) return;
    space += 10;
    printTree(root->right, space);
    printf("\n");
    for (int i = 10; i < space; i++) printf(" ");
    printf("%d (%s)\n", root->key, root->color == RED ? "RED" : "BLACK");
    printTree(root->left, space);
}
int main()
{
    RBNode* root = NULL;
    insert(&root, 20);
    insert(&root, 15);
    insert(&root, 25);
```

```
    insert(&root, 10);
    insert(&root, 5);
    insert(&root, 30);
    insert(&root, 35);
    printf("In-order traversal of the Red-Black Tree:\n");
    inOrder(root);
    printf("\n");
    printf("Tree structure:\n");
    printTree(root, 0);
    return 0;
}
```

OUT PUT:

```
/tmp/iBXlWjFOzm.o
In-order traversal of the Red-Black Tree:
5 (RED) 10 (BLACK) 15 (RED) 20 (BLACK) 25 (RED) 30 (BLACK) 35
    (RED)
Tree structure:

                    35 (RED)

          30 (BLACK)

                    25 (RED)

20 (BLACK)

                    15 (RED)

          10 (BLACK)

                    5 (RED)
```

4) C program to implement B Tree.

```
#include <stdio.h>
#include <stdlib.h>
#define T 3
typedef struct BTreeNode
{
   int *keys;
   struct BTreeNode **children;
   int numKeys;
   int leaf;
} BTreeNode;
BTreeNode* createNode(int t, int leaf)
{
   BTreeNode *newNode = (BTreeNode *)malloc(sizeof(BTreeNode));
   newNode->keys = (int *)malloc((2*t - 1) * sizeof(int));
   newNode->children = (BTreeNode **)malloc(2*t * sizeof(BTreeNode *));
   newNode->numKeys = 0;
   newNode->leaf = leaf;
   return newNode;
}
void splitChild(BTreeNode *parent, int i, int t)
{
   BTreeNode *fullChild = parent->children[i];
   BTreeNode *newChild = createNode(t, fullChild->leaf);
   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->leaf) {
        for (int j = 0; j < t; j++)
            newChild->children[j] = fullChild->children[j + t];
    }
}
void traverse(BTreeNode *root)
{
    if (root == NULL)
        return;
    int i;
    for (i = 0; i < root->numKeys; i++)
    {
        if (!root->leaf)
            traverse(root->children[i]);
        printf("%d ", root->keys[i]);
    }
    if (!root->leaf)
        traverse(root->children[i]);
}
void insertNonFull(BTreeNode *root, int key, int t)
{
    int i = root->numKeys - 1;
    if (root->leaf)
    {
        while (i >= 0 && key < root->keys[i])
        {
            root->keys[i + 1] = root->keys[i];
            i--;
        }
        root->keys[i + 1] = key;
```

```
        root->numKeys++;
    }
    else
    {
        while (i >= 0 && key < root->keys[i])
        {
            i--;
        }
        i++;
        if (root->children[i]->numKeys == 2 * t - 1)
        {
            splitChild(root, i, t);
            if (key > root->keys[i])
            {
                i++;
            }
        }
        insertNonFull(root->children[i], key, t);
    }
}
void insert(BTreeNode **root, int key, int t)
{
    BTreeNode *r = *root;
    if (r->numKeys == 2 * t - 1)
    {
        BTreeNode *s = createNode(t, 0);
        *root = s;
        s->children[0] = r;
        splitChild(s, 0, t);
        insertNonFull(s, key, t);
    }
```

```
    else
    {
        insertNonFull(r, key, t);
    }
}
int main()
{
    BTreeNode *root = createNode(T, 1);
    insert(&root, 10, T);
    insert(&root, 20, T);
    insert(&root, 5, T);
    insert(&root, 6, T);
    insert(&root, 15, T);
    insert(&root, 30, T);
    printf("Traversal of B-Tree:\n");
    traverse(root);
    printf("\n");
    return 0;
}
```

OUT PUT:

```
/tmp/MuyYqVfysj.o
Traversal of B-Tree:
5 6 10 15 20 30


=== Code Execution Successful ===
```

5) C program to implement B+ Tree

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_KEYS 3
#define MIN_KEYS (MAX_KEYS / 2)
typedef struct Node
{
   int keys[MAX_KEYS];
   struct Node* children[MAX_KEYS + 1];
   int numKeys;
   int isLeaf;
} Node;
Node* createNode(int isLeaf)
{
   Node* newNode = (Node*)malloc(sizeof(Node));
   newNode->numKeys = 0;
   newNode->isLeaf = isLeaf;
   for (int i = 0; i <= MAX_KEYS; i++)
   {
      newNode->children[i] = NULL;
   }
   return newNode;
}
void insertNonFull(Node* 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_KEYS) {
        }
        insertNonFull(node->children[i], key);
    }
}
void splitChild(Node* parent, int index)
{
}
void insert(Node** root, int key)
{
    Node* r = *root;
    if (r->numKeys == MAX_KEYS)
    {
        Node* s = createNode(0);
        *root = s;
        s->children[0] = r;
        splitChild(s, 0);
        insertNonFull(s, key);
    }
    else
```

```
    {
        insertNonFull(r, key);
    }
}
void inorder(Node* node)
{
    if (node != NULL)
    {
        int i;
        for (i = 0; i < node->numKeys; i++)
        {
            if (!node->isLeaf)
            {
                inorder(node->children[i]);
            }
            printf("%d ", node->keys[i]);
        }
        if (!node->isLeaf)
        {
            inorder(node->children[i]);
        }
    }
}
int main()
{
    Node* root = createNode(1);
    insert(&root, 10);
    insert(&root, 20);
    insert(&root, 5);
    insert(&root, 6);
    insert(&root, 15);
```

```
    printf("In-order traversal of B+ Tree:\n");
    inorder(root);
    printf("\n");
    return 0;
}
```

OUT PUT:

```
/tmp/Iw6fFKCoAn.o
In-order traversal of B+ Tree:
5 6 10 15 20


=== Code Execution Successful ===
```