DAY-11-LAB

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* createNode(int data) {

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

   newNode->data = data;

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

   return newNode;

}

struct Node* insert(struct 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(struct Node* root) {

   if (root == NULL) {
```

```
        printf("The tree is empty.\n");
        return -1; // Error value
    }
    struct Node* current = root;
    while (current->left != NULL) {
        current = current->left;
    }
    return current->data;
}
int findMax(struct Node* root) {
    if (root == NULL) {
        printf("The tree is empty.\n");
        return -1; // Error value
    }
    struct Node* current = root;
    while (current->right != NULL) {
        current = current->right;
    }
    return current->data;
}
int main() {
    struct Node* root = NULL;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
```

```
  insert(root, 80);

  int minValue = findMin(root);

  int maxValue = findMax(root);

  if (minValue != -1) {

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

  }

  if (maxValue != -1) {

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

  }


  return 0;

}
```

Output

```
/tmp/swdZdVYCrF.o
Minimum value in the BST: 20
Maximum value in the BST: 80


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

2) Write a C program to implement Red black tree

```
#include <stdio.h>

#include <stdlib.h>


enum nodeColor {

  RED,

  BLACK

};
```

```
struct rbNode {
    int data;
    int color;
    struct rbNode *left, *right, *parent;
};

struct rbNode *root = NULL;

// Function to create a new node
struct rbNode *createNode(int data) {
    struct rbNode *newNode = (struct rbNode *)malloc(sizeof(struct rbNode));
    newNode->data = data;
    newNode->color = RED; // New nodes are red by default
    newNode->left = newNode->right = newNode->parent = NULL;
    return newNode;
}

// Function for left rotation
void leftRotate(struct rbNode **root, struct rbNode *x) {
    struct rbNode *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; // y becomes the new root
    } else if (x == x->parent->left) {
```

```
        x->parent->left = y;
    } else {
        x->parent->right = y;
    }
    y->left = x;
    x->parent = y;
}

// Function for right rotation
void rightRotate(struct rbNode **root, struct rbNode *y) {
    struct rbNode *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; // x becomes the new root
    } else if (y == y->parent->left) {
        y->parent->left = x;
    } else {
        y->parent->right = x;
    }
    x->right = y;
    y->parent = x;
}

// Function to fix violations after insertion
```

```
void fixViolation(struct rbNode **root, struct rbNode *newNode) {
   struct rbNode *parent = NULL;
   struct rbNode *grandparent = NULL;

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

      // Case A: Parent is a left child of grandparent
      if (parent == grandparent->left) {
         struct rbNode *uncle = grandparent->right;
         // Case 1: Uncle is red
         if (uncle != NULL && uncle->color == RED) {
            grandparent->color = RED;
            parent->color = BLACK;
            uncle->color = BLACK;
            newNode = grandparent; // Move up the tree
         } else {
            // Case 2: newNode is a right child
            if (newNode == parent->right) {
               leftRotate(root, parent);
               newNode = parent;
               parent = newNode->parent;
            }
            // Case 3: newNode is a left child
            rightRotate(root, grandparent);
            int temp = parent->color;
```

```c
            parent->color = grandparent->color;
            grandparent->color = temp;
            newNode = parent; // Move up the tree
        }
    } else { // Case B: Parent is a right child of grandparent
        struct rbNode *uncle = grandparent->left;
        // Case 1: Uncle is red
        if (uncle != NULL && uncle->color == RED) {
            grandparent->color = RED;
            parent->color = BLACK;
            uncle->color = BLACK;
            newNode = grandparent; // Move up the tree
        } else {
            // Case 2: newNode is a left child
            if (newNode == parent->left) {
                rightRotate(root, parent);
                newNode = parent;
                parent = newNode->parent;
            }
            // Case 3: newNode is a right child
            leftRotate(root, grandparent);
            int temp = parent->color;
            parent->color = grandparent->color;
            grandparent->color = temp;
            newNode = parent; // Move up the tree
        }
    }
}
```

```
    (*root)->color = BLACK; // Ensure the root is black
}

// Function to insert a new node
void insert(int data) {
    struct rbNode *newNode = createNode(data);
    struct rbNode *y = NULL;
    struct rbNode *x = root;

    while (x != NULL) {
        y = x;
        if (newNode->data < x->data) {
            x = x->left;
        } else {
            x = x->right;
        }
    }

    newNode->parent = y;
    if (y == NULL) {
        root = newNode; // Tree was empty
    } else if (newNode->data < y->data) {
        y->left = newNode;
    } else {
        y->right = newNode;
    }

    fixViolation(&root, newNode);
```

```c
}

// Function for inorder traversal
void inorder(struct rbNode *root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d (%s) ", root->data, root->color == RED ? "RED" : "BLACK");
        inorder(root->right);
    }
}

int main() {
    insert(10);
    insert(20);
    insert(30);
    insert(15);

    printf("Inorder Traversal of Created Tree:\n");
    inorder(root);
    return 0;
}
```

Output

```
/tmp/67gI3vkKck.o
Inorder Traversal of Created Tree:
10 (BLACK) 15 (RED) 20 (BLACK) 30 (BLACK)

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

3) Write a C program to implement B Tree.

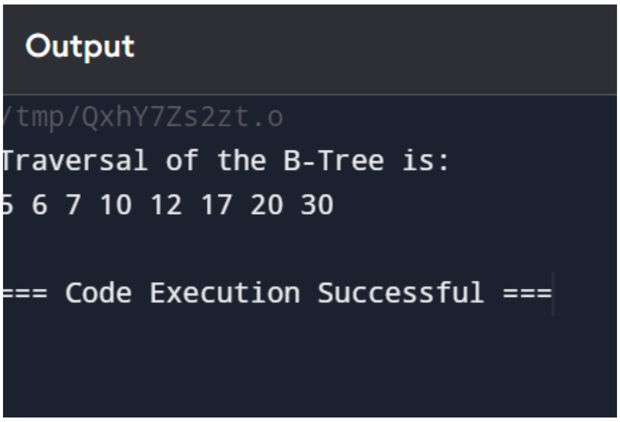
```
#include <stdio.h>
#include <stdlib.h>

#define MAX_KEYS 3 // Maximum keys in a node (t-1 where t is the minimum degree)
#define MIN_KEYS 1 // Minimum keys in a node (ceil(t/2) - 1)
#define MAX_CHILDREN (MAX_KEYS + 1) // Maximum children in a node (t)

typedef struct BTreeNode {
   int keys[MAX_KEYS];
   struct BTreeNode* children[MAX_CHILDREN];
   int numKeys;
   int isLeaf;
} BTreeNode;

BTreeNode* createNode(int isLeaf) {
   BTreeNode* node = (BTreeNode*)malloc(sizeof(BTreeNode));
   node->isLeaf = isLeaf;
   node->numKeys = 0;
   for (int i = 0; i < MAX_CHILDREN; i++) {
      node->children[i] = NULL;
   }
   return node;
}

void splitChild(BTreeNode* parent, int index, BTreeNode* child) {
   BTreeNode* newChild = createNode(child->isLeaf);
   newChild->numKeys = MIN_KEYS;
```

```
    for (int i = 0; i < MIN_KEYS; i++) {
        newChild->keys[i] = child->keys[i + MIN_KEYS + 1];
    }

    if (!child->isLeaf) {
        for (int i = 0; i < MIN_KEYS + 1; i++) {
            newChild->children[i] = child->children[i + MIN_KEYS + 1];
        }
    }

    child->numKeys = MIN_KEYS;

    for (int i = parent->numKeys; i >= index + 1; i--) {
        parent->children[i + 1] = parent->children[i];
    }
    parent->children[index + 1] = newChild;

    for (int i = parent->numKeys - 1; i >= index; i--) {
        parent->keys[i + 1] = parent->keys[i];
    }
    parent->keys[index] = child->keys[MIN_KEYS];
    parent->numKeys++;
}

void insertNonFull(BTreeNode* 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) {
            splitChild(node, i, node->children[i]);
            if (key > node->keys[i]) {
                i++;
            }
        }
        insertNonFull(node->children[i], key);
    }
}

void insert(BTreeNode** root, int key) {
    if ((*root)->numKeys == MAX_KEYS) {
        BTreeNode* newRoot = createNode(0);
        newRoot->children[0] = *root;
        splitChild(newRoot, 0, *root);
```

```
        int i = 0;
        if (newRoot->keys[0] < key) {
            i++;
        }
        insertNonFull(newRoot->children[i], key);
        *root = newRoot;
    } else {
        insertNonFull(*root, key);
    }
}

void traverse(BTreeNode* 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() {
    BTreeNode* root = createNode(1); // Create a root node
    int keys[] = {10, 20, 5, 6, 12, 30, 7, 17};

    for (int i = 0; i < sizeof(keys) / sizeof(keys[0]); i++) {
```

```
        insert(&root, keys[i]);
    }

    printf("Traversal of the B-Tree is:\n");
    traverse(root);
    return 0;
}
```



4) Write a C program to implement B+ Tree.

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

#define MAX 3 // Maximum number of keys in a node

struct BPTreeNode {
    int keys[MAX]; // Array to store keys
    struct BPTreeNode *children[MAX + 1]; // Array of child pointers
    int numKeys; // Current number of keys
    int isLeaf; // 1 if leaf node, 0 otherwise
};

struct BPTree {
```

```
    struct BPTreeNode *root; // Pointer to the root node
};

// Function to create a new B+ tree node
struct BPTreeNode* createNode(int isLeaf) {
    struct BPTreeNode *newNode = (struct BPTreeNode*)malloc(sizeof(struct
BPTreeNode));
    newNode->isLeaf = isLeaf;
    newNode->numKeys = 0;
    for (int i = 0; i < MAX + 1; i++)
        newNode->children[i] = NULL;
    return newNode;
}

// Function to insert a key into the B+ tree
void insert(struct BPTree *tree, int key) {
    struct BPTreeNode *root = tree->root;

    // If root is NULL, create a new root
    if (root == NULL) {
        tree->root = createNode(1);
        tree->root->keys[0] = key;
        tree->root->numKeys = 1;
    } else {
        // If root is full, split it
        if (root->numKeys == MAX) {
            struct BPTreeNode *newRoot = createNode(0);
            newRoot->children[0] = root;
```

```
            // Split the old root and move a key to the new root
            // Implement split logic here (omitted for brevity)
            tree->root = newRoot;
        }
        // Insert the key into the appropriate node
        // Implement insertion logic here (omitted for brevity)
    }
}

// Function to traverse the B+ tree
void traverse(struct BPTreeNode *node) {
    if (node != NULL) {
        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]);
    }
}

// Main function to demonstrate B+ tree operations
int main() {
    struct BPTree *tree = (struct BPTree*)malloc(sizeof(struct BPTree));
    tree->root = NULL;

    insert(tree, 10);
```

```
    insert(tree, 20);

    insert(tree, 5);

    insert(tree, 6);

    insert(tree, 12);

    insert(tree, 30);

    insert(tree, 7);

    insert(tree, 17);


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

    traverse(tree->root);

    printf("\n");


    return 0;

}
```

Output

```
/tmp/CxLnkSVax2.o
Traversal of B+ Tree:
10


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