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* newNode(int data) {
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->data = data;
    node->left = node->right = NULL;
    return node;
}
struct Node* insert(struct Node* node, int data) {
    if (node == NULL) return newNode(data);
    if (data < node->data) node->left = insert(node->left, data);
    else node->right = insert(node->right, data);
    return node;
}
int findMin(struct Node* node) {
    while (node->left != NULL) node = node->left;
    return node->data;
}
```

```
int findMax(struct Node* node) {
    while (node->right != NULL) node = node->right;
    return node->data;
}
    int main() {
    struct Node* root = NULL;
    root = insert(root, 15);
    insert(root, 10);
    insert(root, 20);
    insert(root, 8);
    insert(root, 12);
    printf("Minimum: %d\n", findMin(root));
    printf("Maximum: %d\n", findMax(root));
    return 0;
}
```

output:

```
Minimum: 8
Maximum: 20
```

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;
        struct AVLNode *right;
        int height;
} AVLNode;
AVLNode* createNode(int key);
int height(AVLNode *node);
int max(int a, int b);
AVLNode* rightRotate(AVLNode *y);
AVLNode* leftRotate(AVLNode *x);
int getBalance(AVLNode *node);
AVLNode* insert(AVLNode *node, int key);
AVLNode* minValueNode(AVLNode *node);
AVLNode* deleteNode(AVLNode *root, int key);
AVLNode* search(AVLNode *root, int key);
void inorder(AVLNode *root);
void freeTree(AVLNode *root);
AVLNode* createNode(int key) {
        AVLNode *node = (AVLNode*)malloc(sizeof(AVLNode));
        node->key = key;
        node->left = NULL;
        node->right = NULL;
        node->height = 1;
        return node;
}
int height(AVLNode *node) {
        return (node == NULL) ? 0 : node->height;
```

```
}
int max(int a, int b) {
        return (a > b) ? a : b;
}
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 *node) {
        return (node == NULL) ? 0 : height(node->left) - height(node->right);
}
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 + 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 (key < root->key)
        return search(root->left, key);
    else
        return search(root->right, key);
}
void inorder(AVLNode *root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->key);
        inorder(root->right);
    }
}
void freeTree(AVLNode *root) {
    if (root != NULL) {
        freeTree(root->left);
        freeTree(root->right);
        free(root);
    }
}
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);printf("Inorder traversal of the AVL tree is: ");
    inorder(root);
    printf("\n");
    int key = 30;
    AVLNode *result = search(root, key);
    if (result != NULL)
        printf("Element %d found in the AVL tree.\n", key);
    else
        printf("Element %d not found in the AVL tree.\n", key);
    root = deleteNode(root, 10);
    printf("Inorder traversal after deleting 10: ");
    inorder(root);
    printf("\n");
    freeTree(root);

    return 0;
}
```

output:

```
Inorder traversal of the AVL tree is: 10 20 25 30 40 50
Element 30 found in the AVL tree.
```

```
Inorder traversal after deleting 10: 20 25 30 40 50
```

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

```
#include <stdio.h>
#include <stdlib.h>
#define RED 0
#define BLACK 1
typedef struct RBTreeNode {
        int key;
        struct RBTreeNode *left;
        struct RBTreeNode *right;
        struct RBTreeNode *parent;
        int color; // RED or BLACK
} RBTreeNode;
RBTreeNode* createNode(int key);
void rotateLeft(RBTreeNode** root, RBTreeNode* x);
void rotateRight(RBTreeNode** root, RBTreeNode* y);
void insertFixup(RBTreeNode** root, RBTreeNode* node);
void insertNode(RBTreeNode** root, int key);
void inorderTraversal(RBTreeNode* root);
void freeTree(RBTreeNode* root);
RBTreeNode* createNode(int key) {
        RBTreeNode*                              node                              =
(RBTreeNode*)malloc(sizeof(RBTreeNode));
        node->key = key;
        node->left = NULL;
        node->right = NULL;
        node->parent = NULL;
        node->color = RED; // New nodes are always RED
        return node;
}
void rotateLeft(RBTreeNode** root, RBTreeNode* x) {
        RBTreeNode* 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 rotateRight(RBTreeNode** root, RBTreeNode* y) {
        RBTreeNode* 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->left)
        y->parent->left = x;
    else
        y->parent->right = x;
    x->right = y;
    y->parent = x;
}
void insertFixup(RBTreeNode** root, RBTreeNode* node) {
    RBTreeNode* uncle;
    while (node != *root && node->parent->color == RED) {
        if (node->parent == node->parent->parent->left) {
            uncle = node->parent->parent->right;
            if (uncle != NULL && uncle->color == RED) {
                node->parent->color = BLACK;
                uncle->color = BLACK;
                node->parent->parent->color = RED;
                node = node->parent->parent;
            } else {
                if (node == node->parent->right) {
                    node = node->parent;
                    rotateLeft(root, node);
                }
                node->parent->color = BLACK;
                node->parent->parent->color = RED;
                rotateRight(root, node->parent->parent);
            }
        } else {
            uncle = node->parent->parent->left;
            if (uncle != NULL && uncle->color == RED) {
                node->parent->color = BLACK;
                uncle->color = BLACK;
                node->parent->parent->color = RED;
                node = node->parent->parent;
            } else {
                if (node == node->parent->left) {
                    node = node->parent;
                    rotateRight(root, node);
                }
                node->parent->color = BLACK;
                node->parent->parent->color = RED;
                rotateLeft(root, node->parent->parent);
            }
        }
    }
    (*root)->color = BLACK;
}
void insertNode(RBTreeNode** root, int key) {
    RBTreeNode* node = createNode(key);
    RBTreeNode* y = NULL;
    RBTreeNode* 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;

    insertFixup(root, node);
}
void inorderTraversal(RBTreeNode* root) {
    if (root != NULL) {
        inorderTraversal(root->left);
        printf("%d ", root->key);
        inorderTraversal(root->right);
    }
}
void freeTree(RBTreeNode* root) {
    if (root != NULL) {
        freeTree(root->left);
        freeTree(root->right);
        free(root);
    }
}
int main() {
    RBTreeNode* root = NULL;
    insertNode(&root, 10);
    insertNode(&root, 20);
    insertNode(&root, 30);
    insertNode(&root, 15);
    insertNode(&root, 25);
    printf("Inorder traversal of the Red-Black Tree: ");
    inorderTraversal(root);
    printf("\n");
    freeTree(root);

    return 0;
}
```

output:

```
Inorder traversal of the Red-Black Tree: 10 15 20 25 30
```

4.Write a 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 leaf);
void traverse(BTreeNode* root);
void insertNonFull(BTreeNode* node, int key);
void splitChild(BTreeNode* parent, int i);
void insert(BTreeNode** root, int key);
BTreeNode* search(BTreeNode* root, int key);
void freeTree(BTreeNode* root);
BTreeNode* createNode(int leaf) {
    BTreeNode* node = (BTreeNode*)malloc(sizeof(BTreeNode));
    node->keys = (int*)malloc((2 * T - 1) * sizeof(int));
    node->children = (BTreeNode**)malloc(2 * T * sizeof(BTreeNode*));
    node->numKeys = 0;
    node->leaf = leaf;
    return node;
}
void traverse(BTreeNode* root) {
    int i;
    if (root != NULL) {
        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* 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(BTreeNode* parent, int i) {
    BTreeNode* fullChild = parent->children[i];
    BTreeNode* newChild = createNode(fullChild->leaf);
    int medianIndex = T - 1;

    newChild->numKeys = medianIndex;
    for (int j = 0; j < medianIndex; 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];
        }
    }
    fullChild->numKeys = medianIndex;
    for (int j = parent->numKeys; j >= i + 1; j--) {
        parent->children[j + 1] = parent->children[j];
    }
    parent->children[i + 1] = newChild;

    for (int j = parent->numKeys - 1; j >= i; j--) {
        parent->keys[j + 1] = parent->keys[j];
    }
    parent->keys[i] = fullChild->keys[medianIndex];
    parent->numKeys++;
}
void insert(BTreeNode** root, int key) {
    BTreeNode* r = *root;
    if (r->numKeys == 2 * T - 1) {
        BTreeNode* s = createNode(0);
        *root = s;
        s->children[0] = r;
          splitChild(s, 0);
        insertNonFull(s, key);
    } else {
        insertNonFull(r, key);
    }
}
BTreeNode* search(BTreeNode* root, int key) {
    int i = 0;
    while (i < root->numKeys && key > root->keys[i]) {
        i++;
    }
    if (i < root->numKeys && key == root->keys[i]) {
        return root;
    }
    if (root->leaf) {
        return NULL;
    }
    return search(root->children[i], key);
}
void freeTree(BTreeNode* root) {
    if (root != NULL) {
        if (!root->leaf) {
            for (int i = 0; i <= root->numKeys; i++) {
                freeTree(root->children[i]);
            }
        }
        free(root->keys);
        free(root->children);
        free(root);
    }
}
int main() {
    BTreeNode* root = createNode(1);
    insert(&root, 10);
    insert(&root, 20);
    insert(&root, 5);
    insert(&root, 6);
    insert(&root, 15);printf("Inorder traversal of the B-Tree: ");
    traverse(root);
    printf("\n");
    int key = 15;
    BTreeNode* result = search(root, key);
    if (result != NULL)
        printf("Element %d found in the B-Tree.\n", key);
```

```
          else
                printf("Element %d not found in the B-Tree.\n", key);
          freeTree(root);
      return 0;
     }
```

output

```
Inorder traversal of the B-Tree: 5 6 10 15 20
Element 15 found in the B-Tree.
```

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

```
#include <stdio.h>
#include <stdlib.h>
#define T 3
typedef struct BPlusTreeNode {
    int *keys;
    struct BPlusTreeNode **children;
    struct BPlusTreeNode *next; // Pointer to the next leaf node
    int numKeys;
    int leaf;
} BPlusTreeNode;
BPlusTreeNode* createNode(int leaf);
void traverse(BPlusTreeNode* root);
void insertNonFull(BPlusTreeNode* node, int key);
void splitChild(BPlusTreeNode* parent, int i);
void insert(BPlusTreeNode** root, int key);
```

```
BPlusTreeNode* search(BPlusTreeNode* root, int key);
void freeTree(BPlusTreeNode* root);
BPlusTreeNode* createNode(int leaf) {
    BPlusTreeNode* node = (BPlusTreeNode*)malloc(sizeof(BPlusTreeNode));
    node->keys = (int*)malloc((2 * T - 1) * sizeof(int));
    node->children = (BPlusTreeNode**)malloc(2 * T *
sizeof(BPlusTreeNode*));
    node->next = NULL;
    node->numKeys = 0;
    node->leaf = leaf;
    return node;
}
void traverse(BPlusTreeNode* root) {
    BPlusTreeNode* current = root;
    while (current != NULL) {
        for (int i = 0; i < current->numKeys; i++) {
            printf("%d ", current->keys[i]);
        }
        current = current->next;
    }
}
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* parent, int i) {
    BPlusTreeNode* fullChild = parent->children[i];
    BPlusTreeNode* newChild = createNode(fullChild->leaf);
    int medianIndex = T - 1;
    newChild->numKeys = medianIndex;
    for (int j = 0; j < medianIndex; 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];
        }
    }
    fullChild->numKeys = medianIndex;

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

    for (int j = parent->numKeys - 1; j >= i; j--) {
        parent->keys[j + 1] = parent->keys[j];
    }
    parent->keys[i] = fullChild->keys[medianIndex];
    parent->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);
    }
}
BPlusTreeNode* search(BPlusTreeNode* root, int key) {
    int i = 0;
    while (i < root->numKeys && key > root->keys[i]) {
        i++;
    }
    if (i < root->numKeys && key == root->keys[i]) {
        return root;
    }
    if (root->leaf) {
        return NULL;
    }
    return search(root->children[i], key);
}
void freeTree(BPlusTreeNode* root) {
    if (root != NULL) {
        if (!root->leaf) {
            for (int i = 0; i <= root->numKeys; i++) {
                freeTree(root->children[i]);
            }
        }
        free(root->keys);
        free(root->children);
        free(root);
    }
}
int main() {
    BPlusTreeNode* root = createNode(1);
    insert(&root, 10);
    insert(&root, 20);
    insert(&root, 5);
    insert(&root, 6);
    insert(&root, 15);
    printf("Inorder traversal of the B+ Tree: ");
    traverse(root);
    printf("\n");
    int key = 15;
    BPlusTreeNode* result = search(root, key);
    if (result != NULL)
```

```
          printf("Element %d found in the B+ Tree.\n", key);
     else
          printf("Element %d not found in the B+ Tree.\n", key);
     freeTree(root);
return 0;
}
```

output:

```
Inorder traversal of the B+ Tree: 5 6 10 15 20
Element 15 found in the B+ Tree.
```