1.RED BLACK TREE:

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

typedef enum { RED, BLACK } node_color;

typedef struct RBTreeNode {
    int data;
    node_color color;
    struct RBTreeNode* left;
    struct RBTreeNode* right;
    struct RBTreeNode* parent;
} RBTreeNode;

typedef struct RBTree {
    RBTreeNode* root;
    RBTreeNode* nil; // Sentinel node for leaves and root's parent
} RBTree;

RBTreeNode* create_node(int data, node_color color, RBTreeNode* nil) {
    RBTreeNode* node = (RBTreeNode*)malloc(sizeof(RBTreeNode));
    node->data = data;
    node->color = color;
    node->left = nil;
    node->right = nil;
    node->parent = nil;
    return node;
}

RBTree* create_rbtree() {
    RBTree* tree = (RBTree*)malloc(sizeof(RBTree));
    tree->nil = create_node(0, BLACK, NULL);
    tree->root = tree->nil;
    return tree;
}

void left_rotate(RBTree* tree, RBTreeNode* x) {
    RBTreeNode* y = x->right;
    x->right = y->left;
    if (y->left != tree->nil) {
        y->left->parent = x;
    }
    y->parent = x->parent;
    if (x->parent == tree->nil) {
        tree->root = y;
    } else if (x == x->parent->left) {
        x->parent->left = y;
    } else {
        x->parent->right = y;
    }
    y->left = x;
    x->parent = y;
}

void right_rotate(RBTree* tree, RBTreeNode* y) {
    RBTreeNode* x = y->left;
    y->left = x->right;
    if (x->right != tree->nil) {
        x->right->parent = y;
    }
    x->parent = y->parent;
    if (y->parent == tree->nil) {
        tree->root = x;
    } else if (y == y->parent->right) {
        y->parent->right = x;
    } else {
        y->parent->left = x;
    }
    x->right = y;
    y->parent = x;
}

void insert_fixup(RBTree* tree, RBTreeNode* z) {
    while (z->parent->color == RED) {
        if (z->parent == z->parent->parent->left) {
            RBTreeNode* y = z->parent->parent->right;
            if (y->color == RED) {
                z->parent->color = BLACK;
                y->color = BLACK;
                z->parent->parent->color = RED;
                z = z->parent->parent;
            } else {
                if (z == z->parent->right) {
                    z = z->parent;
                    left_rotate(tree, z);
                }
                z->parent->color = BLACK;
                z->parent->parent->color = RED;
                right_rotate(tree, z->parent->parent);
            }
        } else {
            RBTreeNode* y = z->parent->parent->left;
            if (y->color == RED) {
                z->parent->color = BLACK;
                y->color = BLACK;
                z->parent->parent->color = RED;
                z = z->parent->parent;
            } else {
                if (z == z->parent->left) {
                    z = z->parent;
                    right_rotate(tree, z);
                }
                z->parent->color = BLACK;
                z->parent->parent->color = RED;
                left_rotate(tree, z->parent->parent);
            }
        }
    }
    tree->root->color = BLACK;
}

void insert(RBTree* tree, int data) {
    RBTreeNode* z = create_node(data, RED, tree->nil);
    RBTreeNode* y = tree->nil;
    RBTreeNode* x = tree->root;

    while (x != tree->nil) {
        y = x;
        if (z->data < x->data) {
            x = x->left;
        } else {
            x = x->right;
        }
    }
    z->parent = y;
    if (y == tree->nil) {
        tree->root = z;
    } else if (z->data < y->data) {
        y->left = z;
    } else {
        y->right = z;
    }
    z->left = tree->nil;
    z->right = tree->nil;
    z->color = RED;
    insert_fixup(tree, z);
}

void inorder_traversal(RBTree* tree, RBTreeNode* node) {
    if (node != tree->nil) {
        inorder_traversal(tree, node->left);
        printf("%d ", node->data);
        inorder_traversal(tree, node->right);
    }
}

int main() {
    RBTree* tree = create_rbtree();
    int n, value;

    printf("Enter the number of elements to insert: ");
    scanf("%d", &n);

    for (int i = 0; i < n; i++) {
        printf("Enter value %d: ", i + 1);
        scanf("%d", &value);
        insert(tree, value);
    }

    printf("Inorder traversal: ");
    inorder_traversal(tree, tree->root);
    printf("\n");

    return 0;
}
```

INUPUT AND OUTPUT:

```
Enter the number of elements to insert: 5
Enter value 1: 10
Enter value 2: 20
Enter value 3: 30
Enter value 4: 15
Enter value 5: 25
Inorder traversal: 10 15 20 25 30
```

2.SPLAY TREE

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

typedef struct SplayTreeNode {
    int data;
    struct SplayTreeNode* left;
    struct SplayTreeNode* right;
} SplayTreeNode;

typedef struct SplayTree {
    SplayTreeNode* root;
} SplayTree;
```

```
SplayTreeNode* create_node(int data) {
    SplayTreeNode* node = (SplayTreeNode*)malloc(sizeof(SplayTreeNode));
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

SplayTree* create_splay_tree() {
    SplayTree* tree = (SplayTree*)malloc(sizeof(SplayTree));
    tree->root = NULL;
    return tree;
}

SplayTreeNode* right_rotate(SplayTreeNode* x) {
    SplayTreeNode* y = x->left;
    x->left = y->right;
    y->right = x;
    return y;
}

SplayTreeNode* left_rotate(SplayTreeNode* x) {
    SplayTreeNode* y = x->right;
    x->right = y->left;
    y->left = x;
    return y;
}

SplayTreeNode* splay(SplayTreeNode* root, int key) {
    if (root == NULL || root->data == key)
        return root;
    if (root->data > key) {
        if (root->left == NULL) return root;

        if (root->left->data > key) {
            root->left->left = splay(root->left->left, key);
            root = right_rotate(root);
        } else if (root->left->data < key) {
            root->left->right = splay(root->left->right, key);
            if (root->left->right != NULL)
                root->left = left_rotate(root->left);
        }

        return (root->left == NULL) ? root : right_rotate(root);
    } else {
        if (root->right == NULL) return root;

        if (root->right->data > key) {
            root->right->left = splay(root->right->left, key);
            if (root->right->left != NULL)
                root->right = right_rotate(root->right);
        } else if (root->right->data < key) {
            root->right->right = splay(root->right->right, key);
            root = left_rotate(root);
        }

        return (root->right == NULL) ? root : left_rotate(root);
    }
}

void insert(SplayTree* tree, int data) {
    if (tree->root == NULL) {
        tree->root = create_node(data);
        return;
    }

    tree->root = splay(tree->root, data);

    if (tree->root->data == data)
        return;

    SplayTreeNode* new_node = create_node(data);

    if (tree->root->data > data) {
        new_node->right = tree->root;
        new_node->left = tree->root->left;
        tree->root->left = NULL;
    } else {
        new_node->left = tree->root;
        new_node->right = tree->root->right;
        tree->root->right = NULL;
    }

    tree->root = new_node;
}

void inorder_traversal(SplayTreeNode* node) {
    if (node != NULL) {
        inorder_traversal(node->left);
        printf("%d ", node->data);
        inorder_traversal(node->right);
    }
```

```
}

int main() {
    SplayTree* tree = create_splay_tree();
    int n, value;

    printf("Enter the number of elements to insert: ");
    scanf("%d", &n);

    for (int i = 0; i < n; i++) {
        printf("Enter value %d: ", i + 1);
        scanf("%d", &value);
        insert(tree, value);
    }

    printf("Inorder traversal: ");
    inorder_traversal(tree->root);
    printf("\n");

    return 0;
}
```

INPUT AND OUTPUT:

```
Enter the number of elements to insert: 5
Enter value 1: 10
Enter value 2: 20
Enter value 3: 30
Enter value 4: 15
Enter value 5: 25
Inorder traversal: 10 15 20 25 30
```