Implementing Red-Black Tree in C:

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

enum nodeColor {
  RED,
  BLACK
};

struct rbNode {
  int data, color;
  struct rbNode *link[2];
};

struct rbNode *root = NULL;

// Create a red-black tree
struct rbNode *createNode(int data) {
  struct rbNode *newnode;
  newnode = (struct rbNode *)malloc(sizeof(struct rbNode));
  newnode->data = data;
  newnode->color = RED;
  newnode->link[0] = newnode->link[1] = NULL;
  return newnode;
}

// Insert an node
void insertion(int data) {
  struct rbNode *stack[98], *ptr, *newnode, *xPtr, *yPtr;
  int dir[98], ht = 0, index;
```

```
ptr = root;
if (!root) {
  root = createNode(data);
  return;
}

stack[ht] = root;
dir[ht++] = 0;
while (ptr != NULL) {
  if (ptr->data == data) {
    printf("Duplicates Not Allowed!!\n");
    return;
  }
  index = (data - ptr->data) > 0 ? 1 : 0;
  stack[ht] = ptr;
  ptr = ptr->link[index];
  dir[ht++] = index;
}
stack[ht - 1]->link[index] = newnode = createNode(data);
while ((ht >= 3) && (stack[ht - 1]->color == RED)) {
  if (dir[ht - 2] == 0) {
    yPtr = stack[ht - 2]->link[1];
    if (yPtr != NULL && yPtr->color == RED) {
      stack[ht - 2]->color = RED;
      stack[ht - 1]->color = yPtr->color = BLACK;
      ht = ht - 2;
    } else {
      if (dir[ht - 1] == 0) {
        yPtr = stack[ht - 1];
      } else {
        xPtr = stack[ht - 1];
```

```
      yPtr = xPtr->link[1];
      xPtr->link[1] = yPtr->link[0];
      yPtr->link[0] = xPtr;
      stack[ht - 2]->link[0] = yPtr;
    }
    xPtr = stack[ht - 2];
    xPtr->color = RED;
    yPtr->color = BLACK;
    xPtr->link[0] = yPtr->link[1];
    yPtr->link[1] = xPtr;
    if (xPtr == root) {
      root = yPtr;
    } else {
      stack[ht - 3]->link[dir[ht - 3]] = yPtr;
    }
    break;
  }
} else {
  yPtr = stack[ht - 2]->link[0];
  if ((yPtr != NULL) && (yPtr->color == RED)) {
    stack[ht - 2]->color = RED;
    stack[ht - 1]->color = yPtr->color = BLACK;
    ht = ht - 2;
  } else {
    if (dir[ht - 1] == 1) {
      yPtr = stack[ht - 1];
    } else {
      xPtr = stack[ht - 1];
      yPtr = xPtr->link[0];
      xPtr->link[0] = yPtr->link[1];
      yPtr->link[1] = xPtr;
```

```
        stack[ht - 2]->link[1] = yPtr;
      }
      xPtr = stack[ht - 2];
      yPtr->color = BLACK;
      xPtr->color = RED;
      xPtr->link[1] = yPtr->link[0];
      yPtr->link[0] = xPtr;
      if (xPtr == root) {
        root = yPtr;
      } else {
        stack[ht - 3]->link[dir[ht - 3]] = yPtr;
      }
      break;
    }
  }
}
root->color = BLACK;
}

// Delete a node
void deletion(int data) {
  struct rbNode *stack[98], *ptr, *xPtr, *yPtr;
  struct rbNode *pPtr, *qPtr, *rPtr;
  int dir[98], ht = 0, diff, i;
  enum nodeColor color;

  if (!root) {
    printf("Tree not available\n");
    return;
  }
```

```
ptr = root;
while (ptr != NULL) {
  if ((data - ptr->data) == 0)
    break;
  diff = (data - ptr->data) > 0 ? 1 : 0;
  stack[ht] = ptr;
  dir[ht++] = diff;
  ptr = ptr->link[diff];
}

if (ptr->link[1] == NULL) {
  if ((ptr == root) && (ptr->link[0] == NULL)) {
    free(ptr);
    root = NULL;
  } else if (ptr == root) {
    root = ptr->link[0];
    free(ptr);
  } else {
    stack[ht - 1]->link[dir[ht - 1]] = ptr->link[0];
  }
} else {
  xPtr = ptr->link[1];
  if (xPtr->link[0] == NULL) {
    xPtr->link[0] = ptr->link[0];
    color = xPtr->color;
    xPtr->color = ptr->color;
    ptr->color = color;

    if (ptr == root) {
      root = xPtr;
    } else {
```

```
    stack[ht - 1]->link[dir[ht - 1]] = xPtr;
  }

  dir[ht] = 1;
  stack[ht++] = xPtr;
} else {
  i = ht++;
  while (1) {
    dir[ht] = 0;
    stack[ht++] = xPtr;
    yPtr = xPtr->link[0];
    if (!yPtr->link[0])
      break;
    xPtr = yPtr;
  }

  dir[i] = 1;
  stack[i] = yPtr;
  if (i > 0)
    stack[i - 1]->link[dir[i - 1]] = yPtr;

  yPtr->link[0] = ptr->link[0];

  xPtr->link[0] = yPtr->link[1];
  yPtr->link[1] = ptr->link[1];

  if (ptr == root) {
    root = yPtr;
  }

  color = yPtr->color;
```

```
    yPtr->color = ptr->color;
    ptr->color = color;
  }
}

if (ht < 1)
  return;

if (ptr->color == BLACK) {
  while (1) {
    pPtr = stack[ht - 1]->link[dir[ht - 1]];
    if (pPtr && pPtr->color == RED) {
      pPtr->color = BLACK;
      break;
    }

    if (ht < 2)
      break;

    if (dir[ht - 2] == 0) {
      rPtr = stack[ht - 1]->link[1];

      if (!rPtr)
        break;

      if (rPtr->color == RED) {
        stack[ht - 1]->color = RED;
        rPtr->color = BLACK;
        stack[ht - 1]->link[1] = rPtr->link[0];
        rPtr->link[0] = stack[ht - 1];
```

```
  if (stack[ht - 1] == root) {
    root = rPtr;
  } else {
    stack[ht - 2]->link[dir[ht - 2]] = rPtr;
  }
  dir[ht] = 0;
  stack[ht] = stack[ht - 1];
  stack[ht - 1] = rPtr;
  ht++;

  rPtr = stack[ht - 1]->link[1];
}

if ((!rPtr->link[0] || rPtr->link[0]->color == BLACK) &&
  (!rPtr->link[1] || rPtr->link[1]->color == BLACK)) {
  rPtr->color = RED;
} else {
  if (!rPtr->link[1] || rPtr->link[1]->color == BLACK) {
    qPtr = rPtr->link[0];
    rPtr->color = RED;
    qPtr->color = BLACK;
    rPtr->link[0] = qPtr->link[1];
    qPtr->link[1] = rPtr;
    rPtr = stack[ht - 1]->link[1] = qPtr;
  }
  rPtr->color = stack[ht - 1]->color;
  stack[ht - 1]->color = BLACK;
  rPtr->link[1]->color = BLACK;
  stack[ht - 1]->link[1] = rPtr->link[0];
  rPtr->link[0] = stack[ht - 1];
  if (stack[ht - 1] == root) {
```

```
      root = rPtr;
    } else {
      stack[ht - 2]->link[dir[ht - 2]] = rPtr;
    }
    break;
  }
} else {
  rPtr = stack[ht - 1]->link[0];
  if (!rPtr)
    break;

  if (rPtr->color == RED) {
    stack[ht - 1]->color = RED;
    rPtr->color = BLACK;
    stack[ht - 1]->link[0] = rPtr->link[1];
    rPtr->link[1] = stack[ht - 1];

    if (stack[ht - 1] == root) {
      root = rPtr;
    } else {
      stack[ht - 2]->link[dir[ht - 2]] = rPtr;
    }
    dir[ht] = 1;
    stack[ht] = stack[ht - 1];
    stack[ht - 1] = rPtr;
    ht++;

    rPtr = stack[ht - 1]->link[0];
  }
  if ((!rPtr->link[0] || rPtr->link[0]->color == BLACK) &&
    (!rPtr->link[1] || rPtr->link[1]->color == BLACK)) {
```

```
        rPtr->color = RED;
      } else {
        if (!rPtr->link[0] || rPtr->link[0]->color == BLACK) {
          qPtr = rPtr->link[1];
          rPtr->color = RED;
          qPtr->color = BLACK;
          rPtr->link[1] = qPtr->link[0];
          qPtr->link[0] = rPtr;
          rPtr = stack[ht - 1]->link[0] = qPtr;
        }
        rPtr->color = stack[ht - 1]->color;
        stack[ht - 1]->color = BLACK;
        rPtr->link[0]->color = BLACK;
        stack[ht - 1]->link[0] = rPtr->link[1];
        rPtr->link[1] = stack[ht - 1];
        if (stack[ht - 1] == root) {
          root = rPtr;
        } else {
          stack[ht - 2]->link[dir[ht - 2]] = rPtr;
        }
        break;
      }
    }
    ht--;
  }
 }
}

// Print the inorder traversal of the tree
void inorderTraversal(struct rbNode *node) {
 if (node) {
```

```
    inorderTraversal(node->link[0]);
    printf("%d  ", node->data);
    inorderTraversal(node->link[1]);
  }
  return;
}

// Driver code
int main() {
  int ch, data;
  while (1) {
    printf("1. Insertion\t2. Deletion\n");
    printf("3. Traverse\t4. Exit");
    printf("\nEnter your choice:");
    scanf("%d", &ch);
    switch (ch) {
      case 1:
        printf("Enter the element to insert:");
        scanf("%d", &data);
        insertion(data);
        break;
      case 2:
        printf("Enter the element to delete:");
        scanf("%d", &data);
        deletion(data);
        break;
      case 3:
        inorderTraversal(root);
        printf("\n");
        break;
      case 4:
```

```
        exit(0);
      default:
        printf("Not available\n");
        break;
    }
    printf("\n");
  }
  return 0;
}
```

SLAY TREE:

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

// An AVL tree node
struct node
{
    int key;
    struct node *left, *right;
};

/* Helper function that allocates a new node with the given key and
    NULL left and right pointers. */
struct node* newNode(int key)
{
    struct node* node = (struct node*)malloc(sizeof(struct node));
    node->key   = key;
    node->left  = node->right  = NULL;
    return (node);
}
```

```
// A utility function to right rotate subtree rooted with y
// See the diagram given above.
struct node *rightRotate(struct node *x)
{
    struct node *y = x->left;
    x->left = y->right;
    y->right = x;
    return y;
}

// A utility function to left rotate subtree rooted with x
// See the diagram given above.
struct node *leftRotate(struct node *x)
{
    struct node *y = x->right;
    x->right = y->left;
    y->left = x;
    return y;
}

// This function brings the key at root if key is present in tree.
// If key is not present, then it brings the last accessed item at
// root.  This function modifies the tree and returns the new root
struct node *splay(struct node *root, int key)
{
    // Base cases: root is NULL or key is present at root
    if (root == NULL || root->key == key)
        return root;

    // Key lies in left subtree
```

```
if (root->key > key)
{
    // Key is not in tree, we are done
    if (root->left == NULL) return root;

    // Zig-Zig (Left Left)
    if (root->left->key > key)
    {
        // First recursively bring the key as root of left-left
        root->left->left = splay(root->left->left, key);

        // Do first rotation for root, second rotation is done after else
        root = rightRotate(root);
    }
    else if (root->left->key < key) // Zig-Zag (Left Right)
    {
        // First recursively bring the key as root of left-right
        root->left->right = splay(root->left->right, key);

        // Do first rotation for root->left
        if (root->left->right != NULL)
            root->left = leftRotate(root->left);
    }

    // Do second rotation for root
    return (root->left == NULL)? root: rightRotate(root);
}
else // Key lies in right subtree
{
    // Key is not in tree, we are done
    if (root->right == NULL) return root;
```

```
        // Zig-Zag (Right Left)
        if (root->right->key > key)
        {
            // Bring the key as root of right-left
            root->right->left = splay(root->right->left, key);

            // Do first rotation for root->right
            if (root->right->left != NULL)
                root->right = rightRotate(root->right);
        }
        else if (root->right->key < key)// Zag-Zag (Right Right)
        {
            // Bring the key as root of right-right and do first rotation
            root->right->right = splay(root->right->right, key);
            root = leftRotate(root);
        }

        // Do second rotation for root
        return (root->right == NULL)? root: leftRotate(root);
    }
}

// Function to insert a new key k in splay tree with given root
struct node *insert(struct node *root, int k)
{
    // Simple Case: If tree is empty
    if (root == NULL) return newNode(k);

    // Bring the closest leaf node to root
    root = splay(root, k);
```

```
    // If key is already present, then return
    if (root->key == k) return root;

    // Otherwise allocate memory for new node
    struct node *newnode  = newNode(k);

    // If root's key is greater, make root as right child
    // of newnode and copy the left child of root to newnode
    if (root->key > k)
    {
        newnode->right = root;
        newnode->left = root->left;
        root->left = NULL;
    }

    // If root's key is smaller, make root as left child
    // of newnode and copy the right child of root to newnode
    else
    {
        newnode->left = root;
        newnode->right = root->right;
        root->right = NULL;
    }

    return newnode; // newnode becomes new root
}

// A utility function to print preorder traversal of the tree.
// The function also prints height of every node
void preOrder(struct node *root)
```

```
{
   if (root != NULL)
   {
      printf("%d ", root->key);
      preOrder(root->left);
      preOrder(root->right);
   }
}

/* Driver program to test above function*/
int main()
{
   struct node *root = newNode(100);
   root->left = newNode(50);
   root->right = newNode(200);
   root->left->left = newNode(40);
   root->left->left->left = newNode(30);
   root->left->left->left->left = newNode(20);
   root = insert(root, 25);
   printf("Preorder traversal of the modified Splay tree is \n");
   preOrder(root);
   return 0;
}
```