IMPLEMENTING A PHONE DIRECTORY USING RED-BLACK TREES

SARTHAK JAIN (2K19/ME/216)

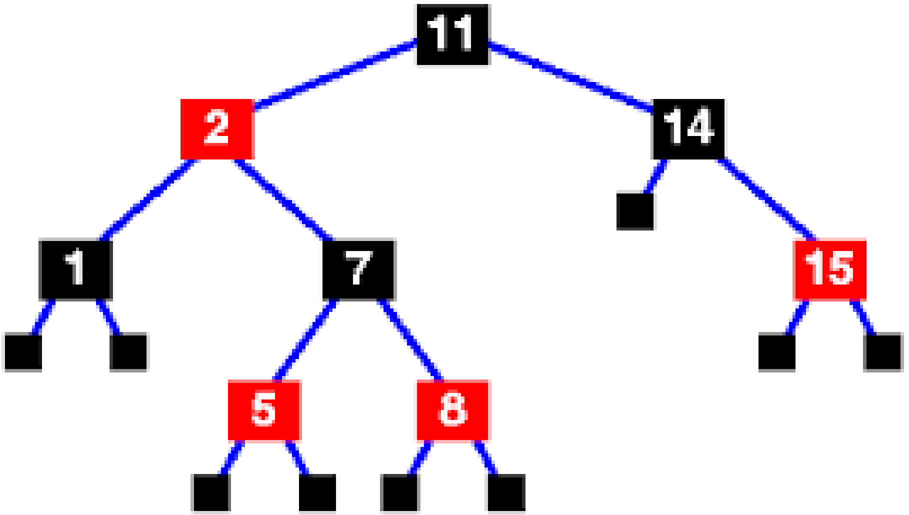
NITIN (2K19/AE/041)

INTRODUCTION TO RB TREES

1. Red black trees or RB trees are a type of self-balancing binary search trees (BST). Self-balancing BSTs rearrange their node in way that the left and right subtrees have near to equal height.
2. Red-black trees are not strictly balancing like AVL trees.
3. But unlike AVL trees, rearranging of nodes to achieve minimum balance requires consideralbly low number of rotations.
4. The nodes of a red-black tree (RBNode) has an additional property of color, which can be either red or black.

INTRODUCTION

1. In this project, we have implemented a phone directory using a red-black tree as the data structure.
2. This directory stores the name, phone number and email address of the contact.
3. The contacts get sorted in an alphabetical order.
4. A search functionality has been given to query any contact from the directory.
5. The deletion functionality can be used to delete any contact from the directory.



CHARACTERISTICS OF RB TREES

PROPERTIES

A Red-black tree ensures that its height is O(lg n) by following some properties, which are:

1. Every node is colored either red or black.
2. Root of the tree is black.
3. All leaves are black.
4. Both children of a red node are black i.e., there can't be consecutive red nodes.
5. All the simple paths from a node to descendant leaves contain the same number of black nodes.

ROTATIONS

There are two kinds of rotations in an RB tree.

1. Left-rotation
2. Right-rotation

Both rotations take constant time to complete

```c
struct RBNode *left_rotation(struct RBNode *tree, struct RBNode *node_to_rotate)
{
    struct RBNode *ptr;
    struct RBNode *parentNode = parentOf(node_to_rotate);
    ptr = node_to_rotate->right;
    node_to_rotate->right = ptr->left;
    if (ptr->left != NULL)
    {
        struct RBNode *leftChild = leftChildOf(ptr);
        leftChild->parent = node_to_rotate;
    }
    ptr->parent = parentNode;
    if (parentNode == NULL)
        tree = ptr;
    else if (node_to_rotate == parentNode->left)
        parentNode->left = ptr;
    else
        parentNode->right = ptr;
    ptr->left = node_to_rotate;
    node_to_rotate->parent = ptr;

    printf("\nLeft rotated\n");

    return tree;
}
```

LEFT ROTATION

```c
struct RBNode *right_rotation(struct RBNode *tree, struct RBNode *node_to_rotate)
{
    struct RBNode *ptr;
    struct RBNode *parentNode = parentOf(node_to_rotate);
    ptr = node_to_rotate->left;
    node_to_rotate->left = ptr->right;
    if (ptr->right != NULL)
    {
        struct RBNode *rightChild = rightChildOf(ptr);
        rightChild->parent = node_to_rotate;
    }
    ptr->parent = parentNode;
    if (parentNode == NULL)
        tree = ptr;
    else if (node_to_rotate == parentNode->right)
        parentNode->right = ptr;
    else
        parentNode->left = ptr;
    ptr->right = node_to_rotate;
    node_to_rotate->parent = ptr;

    printf("\nRight rotated\n");

    return tree;
}
```

RIGHT ROTATION

TIME COMPLEXITIES

RB tree, being a self-balancing BST, arranges its nodes in a manner that the height is nearly equal to $\log_2 n$. Thus, it significantly reduces the time complexity of various query operations, in comparison to a linked list or a standard BST.

1. Insertion : O(ln n)
2. Searching : O(ln n)
3. Deletion : O(ln n)

SPACE COMPLEXITY

A Red-black tree takes the amount of memory, given by the sum of the memory allocated to its nodes, which is:

$$\mathbf{O(n)}$$

OUR PROJECT

STRUCTURE OF A NODE IN OUR TREE

```
struct RBNode
{
    char *contactName;
    char *contactNo;
    char *email;

    enum COLOR node_color;
    struct RBNode *parent;
    struct RBNode *left;
    struct RBNode *right;
};
```

OPERATIONS PERFORMED

- Rb tree creation (New directory).
- Creating a new contact.
- Searching for a contact.
- Deleting a contact.
- Displaying the phone directory.

Creating a contact

This function creates a new node and inserts it into the RB tree.

```c
struct RBNode *create_node(char *name, char *no, char *mailid)
{
    struct RBNode *new_node;
    new_node = (struct RBNode *)malloc(sizeof(struct RBNode));

    new_node->contactName = malloc(strlen(name) + 1); // +1 to store null
    strcpy(new_node->contactName, name);
    new_node->contactNo = malloc(strlen(no) + 1);
    strcpy(new_node->contactNo, no);
    new_node->email = malloc(strlen(mailid) + 1);
    strcpy(new_node->email, mailid);

    new_node->node_color = RED;
    new_node->parent = NULL;
    new_node->left = NULL;
    new_node->right = NULL;

    printf("Node created");

    return new_node;
}
```

INSERT TO THE DIRECTORY

```c
struct RBNode *insertNode(struct RBNode *tree, struct RBNode *new_node)
{
    struct RBNode *ptr, *focus;
    int val;
    ptr = tree;
    focus = NULL;
    while (ptr != NULL)
    {
        focus = ptr;
        val = strcmp(new_node->contactName, ptr->contactName);
        if (val < 0)
            ptr = ptr->left;
        else
            ptr = ptr->right;
    }

    new_node->parent = focus;

    if (focus == NULL)
        tree = new_node;
    else if (strcmp(new_node->contactName, focus->contactName) < 0)
        focus->left = new_node;
    else
        focus->right = new_node;

    printf("\nNode inserted");

    return insertFixUp(tree, new_node); // for the reqd rotations
}
```

```c
struct RBNode *insertFixUp(struct RBNode *tree, struct RBNode *node_to_fix)
{
    struct RBNode *ptr;
    struct RBNode *parentNode = parentOf(node_to_fix);
    struct RBNode *grandParentNode, *uncle;
    grandParentNode = NULL;
    uncle = NULL;

    if (parentNode == NULL)···

    while (parentNode != NULL && parentNode->node_color == RED)
    {
        parentNode = parentOf(node_to_fix);
        grandParentNode = grandParentOf(node_to_fix);
        uncle = uncleOf(node_to_fix);

        if (parentNode == grandParentNode->left)
        {
            if (uncle != NULL && uncle->node_color == RED)···
            else if (node_to_fix == parentNode->right) // forms a triangle
            {
                node_to_fix = parentNode;
                tree = left_rotation(tree, node_to_fix);

                parentNode = parentOf(node_to_fix);
                grandParentNode = grandParentOf(node_to_fix);

                parentNode->node_color = BLACK;
                grandParentNode->node_color = RED;
                tree = right_rotation(tree, grandParentNode);
            }
            else if (node_to_fix == parentNode->left) // forms a line···
        }
        else // parentNode is in the right of grandParentNode···
    }
```

SEARCHING FOR THE CONTACT

```c
struct RBNode *nodeHavingValue(struct RBNode *tree, char *search)
{
    struct RBNode *curr;
    if (tree == NULL)
    {
        printf("\nThe tree is empty");
        return NULL;
    }
    curr = tree;
    if (strcmp(curr->contactName, search) == 0)
        return curr;
    while (curr != NULL && strcmp(search, curr->contactName) != 0)
        curr = (strcmp(search, curr->contactName) < 0) ? curr->left : curr->right;
    if (curr == NULL)
    {
        printf("Not found");
        return NULL;
    }
    else
        return curr;
}
```

DELETING THE CONTACT

This function first searches for the contact using nodeHavingValue() and then deletes it.

```c
struct RBNode *deleteNode(struct RBNode *tree, struct RBNode *node_to_delete)
{
    struct RBNode *inOrderSuccessor = BSTReplace(node_to_delete);
    struct RBNode *parentNode = parentOf(node_to_delete);

    boolean bothNodesAreBlack = ((inOrderSuccessor == NULL || inOrderSuccessor->node_color == BLACK)
                                  && node_to_delete->node_color == BLACK);

    if (inOrderSuccessor == NULL)
    {
        if (node_to_delete == tree)···
        else···
        free(node_to_delete);
        return tree;
    }

    if (node_to_delete->left == NULL || node_to_delete->right == NULL)
    {
        if (node_to_delete == tree)···
        else
        {
            if (node_to_delete == leftChildOf(parentNode))
                parentNode->left = inOrderSuccessor;
            else
                parentNode->right = inOrderSuccessor;
            free(node_to_delete);
            inOrderSuccessor->parent = parentNode;
            if (bothNodesAreBlack)
                tree = deleteFixUp(tree, inOrderSuccessor);
            else
                inOrderSuccessor->node_color = BLACK;
        }
        return tree;
    }
    swapValues(inOrderSuccessor, node_to_delete);
    tree = deleteNode(tree, inOrderSuccessor);
    return tree;
```

FIXING THE TREE AFTER DELETION

```c
struct RBNode *deleteFixUp(struct RBNode *tree, struct RBNode *node_to_fix)
{
    if (node_to_fix == tree)
        return tree;

    struct RBNode *sibling = siblingOf(node_to_fix);
    struct RBNode *parentNode = parentOf(node_to_fix);

    if (sibling == NULL)
        tree = deleteFixUp(tree, parentNode);
    else
    {
        if (sibling->node_color == RED)
        {
            parentNode->node_color = RED;
            sibling->node_color = BLACK;
            if (sibling == parentNode->left)
                tree = right_rotation(tree, parentNode);
            else
                tree = left_rotation(tree, parentNode);
            tree = deleteFixUp(tree, node_to_fix);
        }
        else // sibling is black
        {
            if (hasRedChild(sibling))
            {
                if (leftChildOf(sibling) != NULL && leftChildOf(sibling)->node_color == RED)
                {
                    if (sibling == leftChildOf(parentNode))
                    {
                        leftChildOf(sibling)->node_color = sibling->node_color;
                        sibling->node_color = parentNode->node_color;
                        tree = right_rotation(tree, parentNode);
                    }
```

```c
        else // sibling is black
        {
            if (hasRedChild(sibling))
            {
                if (leftChildOf(sibling) != NULL && leftChildOf(sibling)->node_color == RED)
                {
                    if (sibling == leftChildOf(parentNode))
                    {
                        leftChildOf(sibling)->node_color = sibling->node_color;
                        sibling->node_color = parentNode->node_color;
                        tree = right_rotation(tree, parentNode);
                    }
                    else
                    {
                        leftChildOf(sibling)->node_color = parentNode->node_color;
                        tree = right_rotation(tree, sibling);
                        tree = left_rotation(tree, parentNode);
                    }
                }
                else
                {
                    if (sibling == leftChildOf(parentNode))
                    {
                        rightChildOf(sibling)->node_color = parentNode->node_color;
                        tree = left_rotation(tree, sibling);
                        tree = right_rotation(tree, parentNode);
                    }
                    else
                    {
                        rightChildOf(sibling)->node_color = sibling->node_color;
                        sibling->node_color = parentNode->node_color;
                        tree = left_rotation(tree, parentNode);
                    }
                }
                parentNode->node_color = BLACK;
            }
```

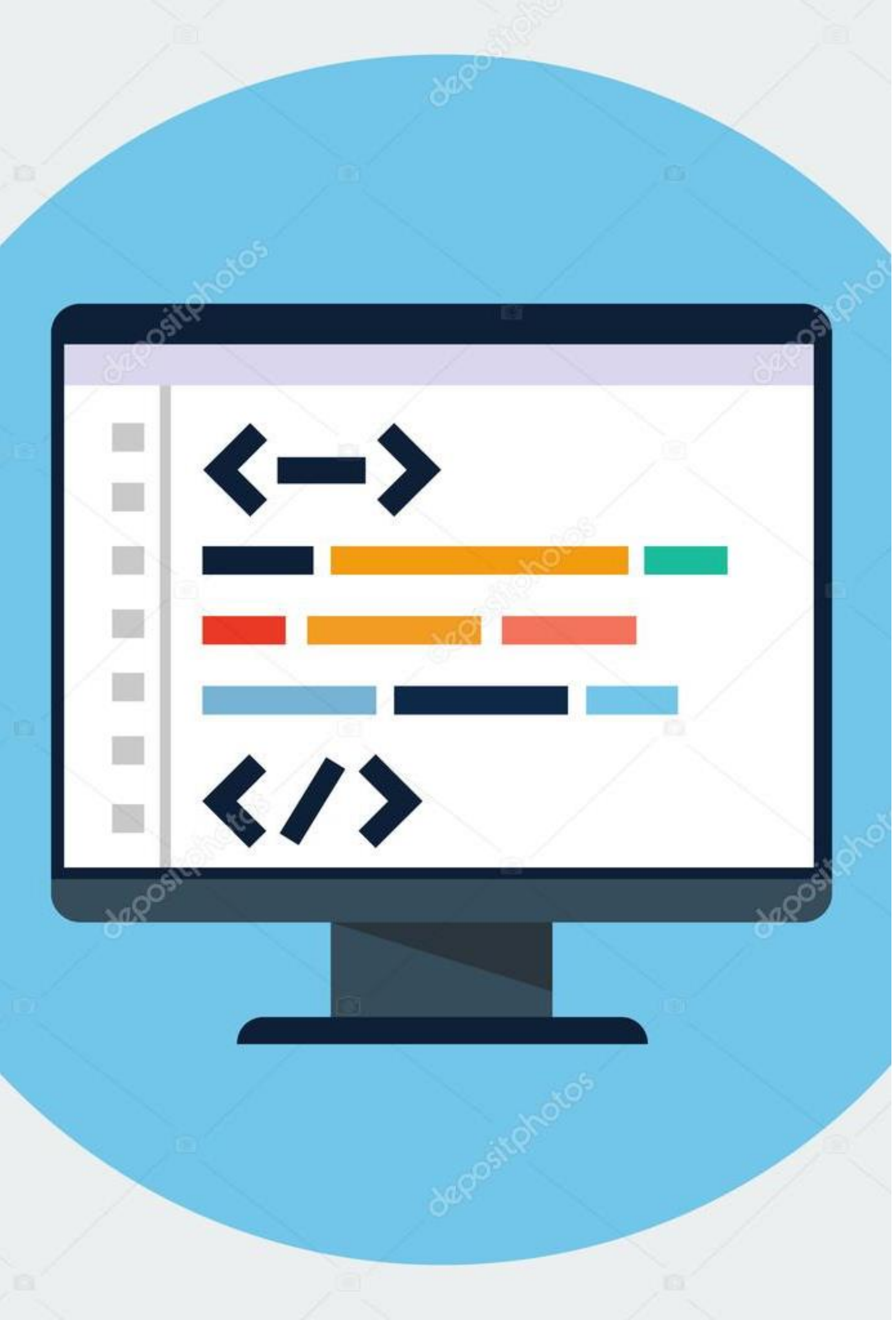
```c
            else
            {
                sibling->node_color = RED;
                if (parentNode->node_color == BLACK)
                    tree = deleteFixUp(tree, parentNode);
                else
                    parentNode->node_color = BLACK;
            }
        }
    }
    return tree;
}
```

DISPLAYING THE DIRECTORY

Here, we have used the inorder traversal of a BST to print the complete directory in a sorted (ascending) order, as we know that the inorder traversal of a BST gives the elements of the tree as (Left Root Right).

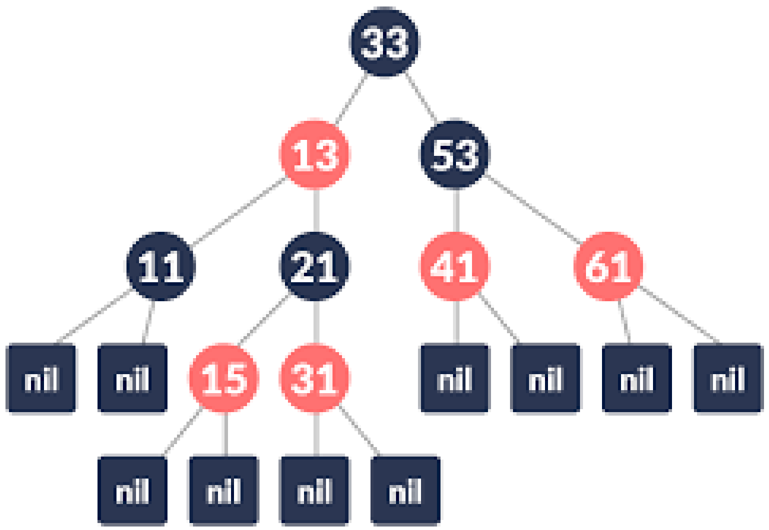
```
void inOrderTraverseTree(struct RBNode *root)
{
    if (root != NULL)
    {
        inOrderTraverseTree(root->left);
        printf("\n %s ", root->contactName);
        printf("%s ", root->contactNo);
        printf("%s ", root->email);
        inOrderTraverseTree(root->right);
    }
}
```

DEMO



LINK TO GIT REPOSITORY

PHONE DIRECTORY IMPLEMENTATION



THANK YOU!