ASSIGNMENT

1) Efficiently storing the frequencies of scores above 50 in a program:

A program `P` takes input of 500 integers, each between 0 and 100, which represent the scores of 500 students. The task is to output the frequency of each score greater than 50. To achieve this, the best method of storing the frequencies would be to use an array of size 51. Here's the detailed approach:

1. Array Setup: Create an array called frequency with 51 elements to store counts of scores from 51 to 100. For example:

   - frequency[0] will store the count of the score 51.
   - frequency[1] will store the count of the score 52.
   - ...
   - frequency[49] will store the count of the score 100.

2. Processing Input: As you process each score, check if it's greater than 50. If so, update the corresponding position in the frequency array. For a given score, the index in the array can be calculated as `score - 51`.

3. Output: After processing all the scores, loop through the frequency array and print the count of each score from 51 to 100.

2) Determining the position for the ninth element in a circular queue:

Consider a circular queue `q` of size 11, where elements are stored at positions `q[0]` through `q[10]`. Initially, both the front and rear pointers are set to `q[2]`. Let's determine where the ninth element will be inserted:

- Initially, both `front` and `rear` pointers are at position 2.
- For the first element, the rear pointer moves to `q[3]`.

- For the second element, the rear pointer moves to `q[4]`.
- This pattern continues until the eighth element is placed at `q[10]`.
- When adding the ninth element, the rear pointer wraps around to `q[0]`, as the queue is circular.

Thus, the ninth element will be added at position `q[0]`.

3) Implementing a Red-Black Tree in C:

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

typedef enum { RED, BLACK } Color;

typedef struct Node {
   int data;
   Color color;
   struct Node left, right, parent;
} Node;

Node root = NULL;

// Function prototypes
Node createNode(int data);
void rotateLeft(Node &root, Node &pt);
void rotateRight(Node &root, Node &pt);
void fixViolation(Node &root, Node &pt);
void insert(int data);
void inorder(Node root);
void printTree(Node root, int space);
```

```
int main() {
    insert(7); insert(3); insert(18); insert(10);
    insert(22); insert(8); insert(11); insert(26);

    printf("Inorder Traversal of the Tree:\n");
    inorder(root);

    printf("\nTree Structure:\n");
    printTree(root, 0);

    return 0;
}

Node createNode(int data) {
    Node newNode = (Node )malloc(sizeof(Node));
    newNode->data = data;
    newNode->color = RED;
    newNode->left = newNode->right = newNode->parent = NULL;
    return newNode;
}

void rotateLeft(Node &root, Node &pt) {
    Node pt_y = pt->right;
    pt->right = pt_y->left;

    if (pt->right != NULL)
        pt->right->parent = pt;

    pt_y->parent = pt->parent;

    if (pt->parent == NULL)
```

```
        root = pt_y;
    else if (pt == pt->parent->left)
        pt->parent->left = pt_y;
    else
        pt->parent->right = pt_y;

    pt_y->left = pt;
    pt->parent = pt_y;
}

void rotateRight(Node &root, Node &pt) {
    Node pt_y = pt->left;
    pt->left = pt_y->right;

    if (pt->left != NULL)
        pt->left->parent = pt;

    pt_y->parent = pt->parent;

    if (pt->parent == NULL)
        root = pt_y;
    else if (pt == pt->parent->left)
        pt->parent->left = pt_y;
    else
        pt->parent->right = pt_y;

    pt_y->right = pt;
    pt->parent = pt_y;
}

void fixViolation(Node &root, Node &pt) {
```

```
Node pt_parent = NULL;
Node pt_grandparent = NULL;

while ((pt != root) && (pt->color == RED) && (pt->parent->color == RED)) {
  pt_parent = pt->parent;
  pt_grandparent = pt_parent->parent;

  if (pt_parent == pt_grandparent->left) {
    Node pt_uncle = pt_grandparent->right;

    if (pt_uncle && pt_uncle->color == RED) {
      pt_grandparent->color = RED;
      pt_parent->color = BLACK;
      pt_uncle->color = BLACK;
      pt = pt_grandparent;
    } else {
      if (pt == pt_parent->right) {
        rotateLeft(root, pt_parent);
        pt = pt_parent;
        pt_parent = pt->parent;
      }
      rotateRight(root, pt_grandparent);
      Color temp = pt_parent->color;
      pt_parent->color = pt_grandparent->color;
      pt_grandparent->color = temp;
      pt = pt_parent;
    }
  } else {
    Node pt_uncle = pt_grandparent->left;

    if (pt_uncle && pt_uncle->color == RED) {
            pt_grandparent->color = RED;
            pt_parent->color = BLACK;
            pt_uncle->color = BLACK;
            pt = pt_grandparent;
        } else {
            if (pt == pt_parent->left) {
                rotateRight(root, pt_parent);
                pt = pt_parent;
                pt_parent = pt->parent;
            }
            rotateLeft(root, pt_grandparent);
            Color temp = pt_parent->color;
            pt_parent->color = pt_grandparent->color;
            pt_grandparent->color = temp;
            pt = pt_parent;
        }
      }
   }

   root->color = BLACK;
}

void insert(int data) {
   Node pt = createNode(data);
   root = bstInsert(root, pt);
   fixViolation(root, pt);
}

Node bstInsert(Node root, Node pt) {
   if (root == NULL)
      return pt;
```

```
    if (pt->data < root->data) {
        root->left = bstInsert(root->left, pt);
        root->left->parent = root;
    } else if (pt->data > root->data) {
        root->right = bstInsert(root->right, pt);
        root->right->parent = root;
    }

    return root;
}

void inorder(Node root) {
    if (root == NULL)
        return;
    inorder(root->left);
    printf("%d ", root->data);
    inorder(root->right);
}

void printTree(Node root, int space) {
    if (root == NULL)
        return;

    space += 10;

    printTree(root->right, space);
    printf("\n");
    for (int i = 10; i < space; i++)
        printf(" ");
    printf("%d(%s)\n", root->data, root->color == RED ? "RED" : "BLACK");
    printTree(root->left, space);
}
```

Explanation:

- Node Structure: Each node has a `data` value, a `color` (either RED or BLACK), and pointers to its `left`, `right`, and `parent` nodes.
- Insertion: The `insert` function inserts a node and calls `fixViolation` to maintain the Red-Black Tree properties.
- Rotations: The functions `rotateLeft` and `rotateRight` ensure tree balance by rotating nodes.
- Violation Fixing: The `fixViolation` function adjusts the tree to ensure the Red-Black properties are maintained.
- Traversal and Display: `inorder` performs an in-order traversal, and `printTree` visually prints the tree structure.

Submitted by:

Aswin Oommen Jacob

S1 MCA

Submitted to:

Ms. Akshara Sasidaran