ASSIGNMENT

1) A program P reads in 500 integers in the range [0..100] representing the scores of 500 students. It then prints the frequency of each score above 50. What would be the best way for P to store the frequencies?

The best way for program P to store the frequencies of scores above 50 is to use an array with 51 elements. This array will be indexed such that:

- Index 0 corresponds to the score 51
- Index 1 corresponds to the score 52
- ...
- Index 49 corresponds to the score 100

Implementation Steps

1. **Initialization**: Create an array frequency of size 51 and initialize all elements to zero. This will hold the counts for scores from 51 to 100.
2. **Reading Scores**: As the program reads each of the 500 scores, check if the score is greater than 50.
3. **Updating Frequencies**: If the score is greater than 50, increment the corresponding index in the frequency array:
   - For a score s, use frequency[s - 51] += 1.
4. **Printing Frequencies**: After processing all scores, iterate through the frequency array and print the frequencies for scores from 51 to 100.

2) In Consider a standard Circular Queue \'q\' implementation (which has the same condition for Queue Full and Queue Empty) whose size is 11 and the elements of the queue are q[0], q[1], q[2].....,q[10]. The front and rear pointers are initialized to point at q[2] . In which position will the ninth element be added?

a circular queue, when both the front and rear pointers are initialized to the same position, it indicates that the queue is empty. In this case, both pointers point to `q[2]`.

Here's how the enqueue operation works in a circular queue:

1. **First Element**: When the first element is added, the rear pointer moves to the next position. So after adding the first element, the rear will point to `q[3]`.
2. **Second Element**: The rear moves to `q[4]`.
3. **Third Element**: The rear moves to `q[5]`.
4. **Fourth Element**: The rear moves to `q[6]`.
5. **Fifth Element**: The rear moves to `q[7]`.
6. **Sixth Element**: The rear moves to `q[8]`.
7. **Seventh Element**: The rear moves to `q[9]`.
8. **Eighth Element**: The rear moves to `q[10]`.
9. **Ninth Element**: Finally, the rear moves to the next position, which wraps around to `q[0]` (since it's a circular queue).

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

3) Write a C Program to implement Red Black Tree?

```
#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 to create a new node
Node* createNode(int data) {
    Node *newNode = (Node *)malloc(sizeof(Node));
    newNode->data = data;
    newNode->color = RED; // New nodes are red by default
    newNode->left = newNode->right = newNode->parent = NULL;
    return newNode;
}

// Function to perform a left rotation
void leftRotate(Node *x) {
    Node *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; // Update root
    else if (x == x->parent->left)
        x->parent->left = y;
    else
        x->parent->right = y;

    y->left = x;
    x->parent = y;
}

// Function to perform a right rotation
void rightRotate(Node *y) {
    Node *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; // Update root
    else if (y == y->parent->left)
        y->parent->left = x;
    else
        y->parent->right = x;

    x->right = y;
    y->parent = x;
}

// Fix the tree after insertion
void fixViolation(Node *z) {
    while (z->parent != NULL && z->parent->color == RED) {
        if (z->parent == z->parent->parent->left) {
            Node *y = z->parent->parent->right;
            if (y != NULL && 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;
                    leftRotate(z);
                }
                z->parent->color = BLACK;
                z->parent->parent->color = RED;
                rightRotate(z->parent->parent);
            }
        } else {
            Node *y = z->parent->parent->left;
            if (y != NULL && 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;
                    rightRotate(z);
                }
                z->parent->color = BLACK;
                z->parent->parent->color = RED;
                leftRotate(z->parent->parent);
            }
        }
    }
    root->color = BLACK;
}
```

```
// Function to insert a new node
void insert(int data) {
    Node *newNode = createNode(data);
    Node *y = NULL;
    Node *x = root;

    while (x != NULL) {
        y = x;
        if (newNode->data < x->data)
            x = x->left;
        else
            x = x->right;
    }

    newNode->parent = y;

    if (y == NULL) {
        root = newNode; // Tree was empty
    } else if (newNode->data < y->data) {
        y->left = newNode;
    } else {
        y->right = newNode;
    }

    fixViolation(newNode);
}

// In-order traversal of the tree
void inorder(Node *root) {
    if (root == NULL)
        return;
    inorder(root->left);
    printf("%d ", root->data);
    inorder(root->right);
}

// Main function
int main() {
    insert(7);
    insert(3);
    insert(18);
    insert(10);
    insert(22);
    insert(8);
    insert(11);
    insert(26);

    printf("In-order traversal of the Red-Black Tree:\n");
    inorder(root);
    printf("\n");

    return 0;}
```

Submitted by,
Ammu Ajimon
S1 MCA

Submitted to,
Akshara Sasidharan