ASSIGNMENT

Amruthalakshmi K S
Roll no:15
S1.MCA

1) 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?

Ans:In a circular queue, when an element is added, the rear pointer moves forward. If it reaches the end of the array, it wraps around to the beginning.

The queue has 11 positions (q[0] to q[10]).
Both front and rear pointers are initialized at q[2].
Now, let's find out where the ninth element will be added:

First element is added at q[2] (rear pointer moves to q[3]).
Second element is added at q[3] (rear moves to q[4]).
Third element is added at q[4] (rear moves to q[5]).
Fourth element is added at q[5] (rear moves to q[6]).
Fifth element is added at q[6] (rear moves to q[7]).
Sixth element is added at q[7] (rear moves to q[8]).
Seventh element is added at q[8] (rear moves to q[9]).
Eighth element is added at q[9] (rear moves to q[10]).
Ninth element is added at q[10] (rear moves to q[0] because of the circular nature).
So, the ninth element will be added at position q[0].

2)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?

Ans:The best way for program P to store the frequencies of scores above 50 is to use an array of size 51. Here's how:

1. **Create an array** called frequencies with 51 elements (index 0 to 50).
2. **Initialize all elements** to zero.
3. **Read the scores** of the 500 students.

4. For each score above 50, increment the value at frequencies[score - 50].
5. **Print the frequencies** from frequencies[1] to frequencies[50] (representing scores 51 to 100).

This method efficiently counts and stores the frequencies!

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

Ans:Here's a simple C program to implement a Red-Black Tree. Red-Black Trees are balanced binary search trees with additional properties to maintain balance during insertions and deletions. This program focuses on insertion into the tree.

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

enum Color { RED, BLACK };

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

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

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

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

// Function to fix Red-Black tree properties after insertion
void fixViolation(struct Node** root, struct Node* z) {
    while (z != *root && z->parent->color == RED) {
        struct Node* grandParent = z->parent->parent;
```

```
        if (z->parent == grandParent->left) {
            struct Node* uncle = grandParent->right;

            if (uncle != NULL && uncle->color == RED) {
                grandParent->color = RED;
                z->parent->color = BLACK;
                uncle->color = BLACK;
                z = grandParent;
            } else {
                if (z == z->parent->right) {
                    z = z->parent;
                    leftRotate(root, z);
                }
                z->parent->color = BLACK;
                grandParent->color = RED;
                rightRotate(root, grandParent);
            }
        } else {
            struct Node* uncle = grandParent->left;

            if (uncle != NULL && uncle->color == RED) {
                grandParent->color = RED;
                z->parent->color = BLACK;
                uncle->color = BLACK;
                z = grandParent;
            } else {
                if (z == z->parent->left) {
                    z = z->parent;
                    rightRotate(root, z);
                }
                z->parent->color = BLACK;
                grandParent->color = RED;
                leftRotate(root, grandParent);
            }
        }
    }
    (*root)->color = BLACK;
}
```

```
// Function to insert a new node into the Red-Black tree
void insert(struct Node** root, int data) {
    struct Node* z = createNode(data);
    struct Node* y = NULL;
    struct Node* x = *root;

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

    z->parent = y;

    if (y == NULL)
        *root = z;
    else if (z->data < y->data)
        y->left = z;
    else
        y->right = z;

    fixViolation(root, z);
}

// Function to print the tree (In-Order Traversal)
void inorder(struct Node* root) {
    if (root == NULL)
        return;
    inorder(root->left);
    printf("%d ", root->data);
    inorder(root->right);
}

int main() {
    struct Node* root = NULL;

    insert(&root, 10);
    insert(&root, 20);
```

```
    insert(&root, 30);
    insert(&root, 15);
    insert(&root, 25);

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

    return 0;
}
```