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?

To store the frequencies of each score above 50 efficiently, the best way for program PPP would be to use an **array** where the index represents the score and the value at each index represents the frequency of that score.

Approach:

1. **Declare an array of size 101** (since the score range is [0..100]), indexed from 0 to 100. This array will store the frequency of all scores, but the program will only print frequencies of scores greater than 50.
2. **Initialize all values of the array to 0**.
3. **For each input score**, increment the value at the corresponding index in the array. For example:
   - If the score is 55, increment frequency[55].
   - If the score is 78, increment frequency[78].
4. **After processing all scores**, iterate over the array from index 51 to 100 and print the frequencies for scores greater than 50.

Example:

- **Declare the frequency array:**

int frequency[101] = {0}; // Array to store frequencies of scores 0 to 100

- **Processing the input:** For each input score x:

frequency[x]++;

- **Printing frequencies of scores above 50:**

```
for (int i = 51; i <= 100; i++) {     if (frequency[i] > 0)

{        printf("Score %d: %d students\n", i,

frequency[i]);

    }

}
```

2) 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?

In a circular queue, the position of the next element to be added depends on the current position of the rear pointer. The formula used to determine the next position in a circular queue is:

new rear=(current rear+1)%queue size

Given that:

- The queue size is 11 (with indices 0 to 10).
- The front and rear pointers are both initially at q[2]q[2]q[2].
- You are asked where the **ninth** element will be added.

Here's the step-by-step approach:

1. After the first element is added, the rear will move to q[3]q[3]q[3].
2. After the second element is added, the rear will move to q[4]q[4]q[4].
3. After the third element is added, the rear will move to q[5]q[5]q[5].
4. After the fourth element is added, the rear will move to q[6]q[6]q[6].
5. After the fifth element is added, the rear will move to q[7]q[7]q[7].
6. After the sixth element is added, the rear will move to q[8]q[8]q[8].
7. After the seventh element is added, the rear will move to q[9]q[9]q[9].
8. After the eighth element is added, the rear will move to q[10]q[10]q[10].

9. Finally, after the ninth element is added, the rear will wrap around to q[0]q[0]q[0] (because of the circular nature of the 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>

// Define colors for Red-Black Tree

#define RED 1

#define BLACK 0

// Structure for Red-Black Tree node struct

Node {

    int data;     int 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 node

must be red     newNode->left = newNode->right = newNode-

>parent = NULL;     return newNode;

}

// Function to perform a left rotation 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 rotation 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->left) {
        y->parent->left = x;
```

```
    } else {        y->parent-
>right = x;
    }    x->right
= y;    y->parent
= x;
}
// Fix violations after insertion to maintain Red-Black Tree properties
void fixViolation(struct Node **root, struct Node *z) {    while (z !=
*root && z->parent->color == RED) {        if (z->parent == z-
>parent->parent->left) {            struct Node *y = z->parent->parent-
>right; // Uncle of z            if (y != NULL && y->color == RED) { //
Case 1: Uncle is red                z->parent->color = BLACK;
y->color = BLACK;                z->parent->parent->color = RED;
z = z->parent->parent;
```

} else { // Uncle is black if (z == z->parent-

>right) { // Case 2: z is a right child z = z->parent;

leftRotate(root, z);

}

// Case 3: z is a left child z-

>parent->color = BLACK; z->parent-

>parent->color = RED; rightRotate(root,

z->parent->parent);

}

} else { struct Node *y = z->parent->parent->left; //

Uncle of z if (y != NULL && y->color == RED) { // Case 1:

Uncle is red z->parent->color = BLACK; y-

>color = BLACK; z->parent->parent->color = RED;

z = z->parent->parent;

```
        } else { // Uncle is black                if (z == z->parent-
>left) { // Case 2: z is a left child                    z = z->parent;
rightRotate(root, z);
            }
            // Case 3: z is a right child                z-
>parent->color = BLACK;                z->parent-
>parent->color = RED;                leftRotate(root, z-
>parent->parent);
        }
      }
    }
    (*root)->color = BLACK; // Ensure the root is always 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;

    // Standard binary search tree insertion

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; // The tree was empty

} else if (z->data < y->data) {          y-

>left = z;

    } else {

        y->right = z;
```

```
    }

    // Fix the Red-Black Tree property violations

fixViolation(root, z);

}

// In-order traversal to print the tree

void inOrder(struct Node *root) {

if (root == NULL) {

        return;

    }

    inOrder(root->left);

printf("%d ", root->data);

inOrder(root->right);

}

// Utility to print the color of a node

void printColor(struct Node *node) { if

(node == NULL) {
```

```
        return;
    }
    printf("Node %d: %s\n", node->data, (node->color == RED ?
"Red" : "Black"));     printColor(node-
>left);     printColor(node->right);
} int main() {     struct Node *root = NULL;
insert(&root, 10);     insert(&root, 20);     insert(&root,
30);     insert(&root, 15);     insert(&root, 25);
insert(&root, 5);     printf("In-order traversal of the
Red-Black Tree:\n");     inOrder(root);

    printf("\n");     printf("Node colors of the Red-
Black Tree:\n");     printColor(root);     return 0;
}
```

Submitted By

Linta Maria Thomas

S1 MCA