ADVANCED DATA STRUCTURE

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?

```
#include <stdio.h>
int main() {
int scores[500];
int frequency[101] = {0};
int i, score;
for (i = 0; i < 500; i++) {
printf("Enter score %d (0 to 100): ", i + 1);
scanf("%d", &score);
while (score < 0 || score > 100) {
printf("Invalid score. Please enter a score between 0 and 100: ");
scanf("%d", &score);
}
scores[i] = score;
}
for (i = 0; i < 500; i++) {
if (scores[i] > 50) {
frequency[scores[i]]++;
}
}
for (i = 51; i <= 100; i++) {
if (frequency[i] > 0) {
printf("Score %d: %d students\n", i, frequency[i]);
```

```
}
}
return 0;
}
```

Explanation:

- Step 1: The program asks for scores of 500 students.
- Step 2: It makes sure that every score entered is valid (between 0 and 100).
- Step 3: It counts how many students scored above 50.
- Step 4: It displays how many students scored each value from 51 to 100.

By the end of the program, you'll know how many students scored, for example, 55, 60, or 90—provided their scores were greater than 50.

2) This C program reads 500 student scores (between 0 and 100), ensuring valid input. It stores each score in an array and then counts how many times each score above 50 appears using a frequency array. Finally, the program prints the count (frequency) of students who scored more than 50, showing how many students received each score between 51 and 100.

In a circular queue, elements are added and removed in a circular manner

A circular queue has a fixed size, and when the end is reached, it wraps around to the beginning.

The front pointer points to the location where elements are removed from. The rear pointer points to the last added element.

Given:

The size of the circular queue is 11 (i.e., q[0] to q[10]).The front and rear pointers are both initialized at q[2].

1. First element: The first element will be added at q[3] (as the rear pointer will move to the next position).

2. Second element: The second element will be added at q[4].

3. Third element: The third element will be added at q[5].

4. Fourth element: The fourth element will be added at q[6].

5. Fifth element: The fifth element will be added at q[7].

6. Sixth element: The sixth element will be added at q[8].

7. Seventh element: The seventh element will be added at q[9].

8. Eighth element: The eighth element will be added at q[10].

9. Ninth element: Now, since the queue is circular, when the rear pointer moves past q[10], it wraps around to the beginning of the queue. Therefore, the ninth element will be added at q[0].

The ninth element will be added at position q[0] because the circular queue wraps around when the end is reached.

3) Write a C Program to implement Red Black Tree

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int data;
    struct Node *left, *right, *parent;
    int color;
};
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*) malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->left = newNode->right = newNode->parent = NULL;
    newNode->color = 1;
    return newNode;
}
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;
}

void rightRotate(struct Node **root, struct Node *x) {
    struct Node *y = x->left;
    x->left = y->right;
    if (y->right != NULL)
        y->right->parent = x;
    y->parent = x->parent;
    if (x->parent == NULL)
        *root = y;
    else if (x == x->parent->right)
        x->parent->right = y;
    else
        x->parent->left = y;
    y->right = x;
    x->parent = y;
}
void fixViolation(struct Node **root, struct Node *z) {
    while (z != *root && z->parent->color == 1) {
```

```
if (z->parent == z->parent->parent->left) {
    struct Node *y = z->parent->parent->right;
    if (y != NULL && y->color == 1) {
        z->parent->color = 0;
        y->color = 0;
        z->parent->parent->color = 1;
        z = z->parent->parent;
    } else {
        if (z == z->parent->right) {
            z = z->parent;
            leftRotate(root, z);
        }
        z->parent->color = 0;
        z->parent->parent->color = 1;
        rightRotate(root, z->parent->parent);
    }
} else {
    struct Node *y = z->parent->parent->left;
    if (y != NULL && y->color == 1) {
        z->parent->color = 0;
        y->color = 0;
        z->parent->parent->color = 1;
        z = z->parent->parent;
    } else {
        if (z == z->parent->left) {
            z = z->parent;
            rightRotate(root, z);
        }
```

```
            z->parent->color = 0;
            z->parent->parent->color = 1;
            leftRotate(root, z->parent->parent);
        }
    }
  }
  (*root)->color = 0;
}
void insert(struct Node **root, int data) {
  struct Node *newNode = createNode(data);
  struct Node *y = NULL;
  struct 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;
  else if (newNode->data < y->data)
    y->left = newNode;
  else
    y->right = newNode;


  fixViolation(root, newNode);
```

```
}

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, 40);
    insert(&root, 50);
    insert(&root, 25);

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

    return 0;
```

}

Explanation :

1. **Node Structure**: Each node contains:
    - data: The value stored in the node.
    - left, right: Pointers to the left and right child nodes.
    - parent: Pointer to the parent node.
    - color: Whether the node is red (1) or black (0).
2. **Rotations**:
    - **Left Rotation**: The node is rotated to the left to balance the tree.
    - **Right Rotation**: The node is rotated to the right to balance the tree.
3. **Fixing Violations**:
    - When a red node is inserted, the fixViolation function ensures the Red-Black Tree properties are maintained (e.g., no two consecutive red nodes). It may involve recoloring nodes or performing rotations.
4. **Insertion**: The insertion is done like in a regular binary search tree, followed by calling fixViolation to restore the Red-Black Tree properties.
5. **In-order Traversal**: This function prints the elements of the tree in sorted order.