DATA STRUCTURE ASSIGNMENT

MEENU K SURESH
S1 MCA
NO:45

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?

Answer:

To store the frequencies of scores above 50 for 500 integers in the range [0..100], the best approach would be to use an array. Here's how you can do it:

1. **Initialize an Array**: Create an array of size 51 (indices 0 to 50) to hold the frequencies of scores from 51 to 100. The index of the array will correspond to the score itself, and the value at each index will represent the frequency of that score.
2. **Count Frequencies**: As you read each score, if it's greater than 50, increment the value at the corresponding index in the array.
3. **Print Frequencies**: After processing all the scores, iterate through the array from index 51 to 100 to print the frequencies of the scores above 50.

Here's a simple pseudocode outline:

```
initialize frequency array freq[51] to 0
for i from 1 to 500:
    read score
    if score > 50:
        freq[score - 50] += 1
for score in range(51, 101):
    print "Score:", score, "Frequency:", freq[score - 50]
```

Benefits of this Approach:

- **Efficient Storage**: The array uses a fixed size, which is memory efficient given the limited range of scores.
- **Direct Access**: Accessing and updating the frequency is O(1), making it very fast.
- **Simplicity**: The logic is straightforward and easy to implement, making the code cleaner.

This method will efficiently track and display the frequency of scores above 50 for the given set of student scores.

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?

Answer:

In a circular queue of size 11, where both the front and rear pointers are initialized to point at q[2], we can track the addition of elements as follows:

Initial Setup

- **Queue Size**: 11
- **Front Pointer**: points to q[2]
- **Rear Pointer**: points to q[2]

Adding Elements

When you add elements to the queue, the rear pointer moves as you insert each element. Here's the sequence of positions for the first nine elements:

1. **1st Element**: Added at q[2] (rear moves to q[3])
2. **2nd Element**: Added at q[3] (rear moves to q[4])
3. **3rd Element**: Added at q[4] (rear moves to q[5])
4. **4th Element**: Added at q[5] (rear moves to q[6])
5. **5th Element**: Added at q[6] (rear moves to q[7])
6. **6th Element**: Added at q[7] (rear moves to q[8])
7. **7th Element**: Added at q[8] (rear moves to q[9])
8. **8th Element**: Added at q[9] (rear moves to q[10])
9. **9th Element**: Added at q[10] (rear moves to q[0])

Conclusion

The ninth element will be added at position q[10].

3) Write a C Program to implement Red Black Tree

Answer:

```
#include <stdio.h>
#include <stdlib.h>
// Node colors
#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 Red-Black Tree 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
    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->right)
        y->parent->right = x;
    else
        y->parent->left = x;
       x->right = y;
       y->parent = x;
}
// Fix the Red-Black Tree after insertion
```

```
void fixViolation(struct Node **root, struct Node *newNode) {
    struct Node *parent = NULL;
    struct Node *grandparent = NULL;
while ((newNode != *root) && (newNode->color == RED) &&
(newNode->parent->color == RED)) {
        parent = newNode->parent;
        grandparent = parent->parent;
if (parent == grandparent->left) {
            struct Node *uncle = grandparent->right;
// Case 1: Uncle is RED
            if (uncle != NULL && uncle->color == RED) {
                grandparent->color = RED;
                parent->color = BLACK;
                uncle->color = BLACK;
                newNode = grandparent;
            } else {
                // Case 2: newNode is right child
                if (newNode == parent->right) {
                    leftRotate(root, parent);
                    newNode = parent;
                    parent = newNode->parent;
                }
 // Case 3: newNode is left child
                rightRotate(root, grandparent);
                int temp = parent->color;
                parent->color = grandparent->color;
                grandparent->color = temp;
```

```
    newNode = parent;
        }
     } else
    {
        struct Node *uncle = grandparent->left;
        // Case 1: Uncle is RED
        if (uncle != NULL && uncle->color == RED) {
            grandparent->color = RED;
            parent->color = BLACK;
            uncle->color = BLACK;
            newNode = grandparent;
        } else {
            // Case 2: newNode is left child
            if (newNode == parent->left) {
                rightRotate(root, parent);
                newNode = parent;
                parent = newNode->parent;
            }
           // Case 3: newNode is right child
            leftRotate(root, grandparent);
            int temp = parent->color;
            parent->color = grandparent->color;
            grandparent->color = temp;
            newNode = parent;
        }
     }
}
```

```
    (*root)->color = BLACK; // Ensure the root is always black
}
// Insert a new node into the Red-Black Tree
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; // Tree was empty
    } else if (newNode->data < y->data) {
        y->left = newNode;
    } else {
        y->right = newNode;
    }
 // Fix violations
    fixViolation(root, newNode);
}
// Inorder traversal of the tree
void inorder(struct Node *root) {
```

```
if (root == NULL)
        return;
    inorder(root->left);
    printf("%d ", root->data);
    inorder(root->right);
}
// Main function to demonstrate the Red-Black Tree
int main() {
    struct Node *root = NULL;
   // Insert nodes into the Red-Black Tree
    insert(&root, 10);
    insert(&root, 20);
    insert(&root, 30);
    insert(&root, 15);
    insert(&root, 25);
    insert(&root, 5);
  // Display the inorder traversal of the tree
    printf("Inorder traversal of the Red-Black Tree:\n");
    inorder(root);
    printf("\n");
    return 0;
}
```