DATA STRUCTURED ASSIGNMENT

Submitted to :Akshara Miss
Submitted by:Sona Anna Koshy
S1MCA
Roll No:59

QUESTION : 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 :

The best way for program P to store the frequencies of scores above 50 would be to use an array. Since the scores are in the range [0..100], and we are only interested in scores above 50, we can use an array of size 51 (for scores 51 to 100).

Here's how you can implement it:

1. **Declare an array of size 51 (index 0 will correspond to score 51, index 1 to score 52, and so on, up to index 49 for score 100).**
2. **Initialize the array to zero to start with no counts.**
3. **Read the 500 integers, and for each score that is greater than 50, increment the corresponding index in the frequency array.**
4. **Print the frequencies for each score from 51 to 100.**

Here's a simple outline in pseudocode:

plaintext
Copy code

```
initialize frequency[51] to 0

for each score in input:
    if score > 50:
        frequency[score - 51] += 1

for i from 0 to 49:
    print "Score", i + 51, ":", frequency[i]
```

QUESTION : 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 implementation, the front and rear pointers help track where elements are added and removed. Given that both pointers are initialized to point at q[2], we can derive the position where the ninth element will be added.

1. **Initial Setup:**
   - **front = 2**
   - **rear = 2**
2. **Calculating Positions:**
   - **The queue is initially not full, as it has 11 slots (0 through 10).**
   - **To keep track of the number of elements in the queue, we will need to update the rear pointer with each addition.**
3. **Adding Elements:**
   - **The first element will be added at rear = 2.**
   - **The second element will be added at rear = (2 + 1) % 11 = 3.**
   - **The third element will be added at rear = (3 + 1) % 11 = 4.**
   - **The fourth element will be added at rear = (4 + 1) % 11 = 5.**
   - **The fifth element will be added at rear = (5 + 1) % 11 = 6.**
   - **The sixth element will be added at rear = (6 + 1) % 11 = 7.**
   - **The seventh element will be added at rear = (7 + 1) % 11 = 8.**
   - **The eighth element will be added at rear = (8 + 1) % 11 = 9.**
   - **The ninth element will be added at rear = (9 + 1) % 11 = 10.**

Thus, the ninth element will be added at position q[10].

QUESTION : 3

Write a C Program to implement Red Black Tree

ANSWER :

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

```c
typedef enum { RED, BLACK } Color;

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

Node *root = NULL;

Node* createNode(int data) {
    Node *newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    newNode->color = RED; // New nodes are always red
    newNode->left = newNode->right = newNode->parent = NULL;
    return newNode;
}

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

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

// Function to fix violations after insertion
void fixViolation(Node **root, Node *newNode) {
    Node *parent = NULL;
    Node *grandparent = NULL;

    while ((newNode != *root) && (newNode->color == RED) &&
(newNode->parent->color == RED)) {
        parent = newNode->parent;
        grandparent = parent->parent;

        // Case A: Parent is the left child of the grandparent
        if (parent == grandparent->left) {
            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: New node is the right child
                if (newNode == parent->right) {
                    leftRotate(root, parent);
                    newNode = parent;
```

```
            parent = newNode->parent;
        }
        // Case 3: New node is the left child
        rightRotate(root, grandparent);
        Color temp = parent->color;
        parent->color = grandparent->color;
        grandparent->color = temp;
        newNode = parent;
      }
    } else { // Case B: Parent is the right child of the grandparent
      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: New node is the left child
        if (newNode == parent->left) {
            rightRotate(root, parent);
            newNode = parent;
            parent = newNode->parent;
        }
        // Case 3: New node is the right child
        leftRotate(root, grandparent);
        Color temp = parent->color;
        parent->color = grandparent->color;
        grandparent->color = temp;
        newNode = 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;
    }

    // Fix violations
    fixViolation(&root, newNode);
}

// Function to do inorder traversal
void inorderHelper(Node *node) {
    if (node == NULL) {
        return;
    }
    inorderHelper(node->left);
    printf("%d (%s) ", node->data, node->color == RED ? "R" : "B");
    inorderHelper(node->right);
}

// Function to print inorder traversal
void inorder() {
    inorderHelper(root);
    printf("\n");
}
```

```c
int main() {
    insert(10);
    insert(20);
    insert(30);
    insert(15);
    insert(25);

    printf("Inorder Traversal of Created Tree:\n");
    inorder();

    return 0;
}
```