DATA STRUCTURE ASSIGNMENT

SUBMITTED BY,

SNEHA S VENU

ROLLNO:57

2)A program P reads 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 is to use a simple array. Here's how to implement this:

Steps:

1. **Create an Array**: Use an array of size 51 to store frequencies for scores from 51 to 100. The index of the array will correspond to the score minus 51.
   - Index 0 → Score 51
   - Index 1 → Score 52
   - ...
   - Index 49 → Score 100
2. **Initialize the Array**: Set all values in the array to 0.
3. **Read Scores**: Loop through the input scores:
   - If the score is greater than 50, increment the corresponding index in the array.
4. **Print Frequencies**: After processing all scores, iterate through the array and print the frequency of each score from 51 to 100.

5)Consider a standard Circular Queue \&#39;q\&#39; 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 queue is typically managed using a fixed-size array, and the front and rear pointers indicate where the next elements will be dequeued and enqueued, respectively.

In this case, the size of the circular queue is 11, and both the front and rear pointers are initialized to point at `q[2]`. Here's how we can determine where the ninth element will be added:

1. **Initialization**: Both `front` and `rear` are at index 2.
2. **Adding Elements**: The first element is added at the position pointed by `rear` (which is `q[2]`), and after each addition, `rear` is incremented. In a circular queue, this increment wraps around when it reaches the end of the array (i.e., it becomes `(rear + 1) % size`).
3. **Sequence of Enqueues**:
    - **1st element**: Added at `q[2]`, `rear` moves to `(2 + 1) % 11 = 3`.
    - **2nd element**: Added at `q[3]`, `rear` moves to `(3 + 1) % 11 = 4`.
    - **3rd element**: Added at `q[4]`, `rear` moves to `(4 + 1) % 11 = 5`.
    - **4th element**: Added at `q[5]`, `rear` moves to `(5 + 1) % 11 = 6`.
    - **5th element**: Added at `q[6]`, `rear` moves to `(6 + 1) % 11 = 7`.
    - **6th element**: Added at `q[7]`, `rear` moves to `(7 + 1) % 11 = 8`.
    - **7th element**: Added at `q[8]`, `rear` moves to `(8 + 1) % 11 = 9`.
    - **8th element**: Added at `q[9]`, `rear` moves to `(9 + 1) % 11 = 10`.
    - **9th element**: Added at `q[10]`, `rear` moves to `(10 + 1) % 11 = 0`.

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

6)Write a C Program to implement Red Black Tree.

ANSWER:

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

typedef enum { RED, BLACK } NodeColor;

typedef struct Node {
    int data;
    NodeColor 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 perform a left rotation
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) // x is root
        *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(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) // y is root
        *root = x;
    else if (y == y->parent->left)
        y->parent->left = x;
    else
```

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

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

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;

        if (parent == grandparent->left) {
            Node *uncle = grandparent->right;

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

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

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) // Tree was empty
        root = newNode;
    else if (newNode->data < y->data)
        y->left = newNode;
    else
        y->right = newNode;

    fixViolation(&root, newNode);
}

void inOrder(Node *root) {
    if (root == NULL)
        return;

    inOrder(root->left);
    printf("%d ", root->data);
    inOrder(root->right);
}
```

```
// Function to search a node
Node* search(Node *root, int data) {
    if (root == NULL || root->data == data)
        return root;

    if (data < root->data)
        return search(root->left, data);
    else
        return search(root->right, data);
}

int main() {
    int values[] = {20, 15, 25, 10, 5, 30};
    int n = sizeof(values) / sizeof(values[0]);

    for (int i = 0; i < n; i++)
        insert(values[i]);

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

    int searchValue = 15;
    Node *result = search(root, searchValue);
    if (result != NULL)
        printf("Node %d found in the tree.\n", searchValue);
    else
        printf("Node %d not found in the tree.\n", searchValue);
    return 0;
}
```