ASSIGNMENT-II

ADVANCED DATA STRUCTURE

Submitted to,

Ms. Akshara Sashi Dharan

Dept. Of computer Application

submitted by,

Meenu M.R

S1 MCA 24-26

Roll no: 46.

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?

Solution:

To store the frequencies of scores above 50, a simple and efficient approach is to use an array. Since the

scores are in the range [0...100], you can create an array of size 51 (to cover scores from 51 to 100) where

each index represents the score, and the value at that index represents the frequency of that score.

1. Initialize an Array: Create an array called frequency of size 51 (indices 0 to 50 will be unused,

corresponding to scores 0 to 50).

2. Read Scores: As you read each of the 500 scores, check if the score is greater than 50. If it is, increment the corresponding index in the frequency array.

3. Print Frequencies: After processing all scores, print the frequencies of scores from 51 to 100.

Summary

- Efficiency: Using an array allows for O (1)O(1)O(1) access time for counting frequencies.
- Simplicity: The logic is straightforward and easy to implement.
- Low Memory Usage: You only need to store counts for 50 possible scores, making it memory efficient.

Using an array of size 51 to track frequencies is efficient in both space and time, making it easy to access.
and manipulate the data. This method allows you to quickly count and display the frequencies of scores.
above 50.

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 area [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 of size 11, if both the front and rear pointers start at position q [2], the queue can be visualized as follows:

front points to q [2]

rear points to q [2]

When elements are added, the `rear` pointer moves to the next position in a circular manner.

Adding Elements

1. **1st element**: rear moves to q [3]
2. **2nd element**: rear moves to q [4]
3. **3rd element:** rear moves to q [5]

4**. 4th element**: rear moves to q [6]
5. **5th element**: rear moves to q [7]
6. **6th element**: rear moves to q [8]
7. **7th element**: rear moves to q [9]
8. **8th element**: rear moves to q [10]
9. **9th element**: `rear` moves to q [0] (wraps around)

Conclusion
Thus, the ninth element will be added at position q [0].

3) Write a C Program to implement Red Black Tree?

Answer:

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

#define RED 0
#define BLACK 1

// Structure for a Red-Black Tree node
struct Node {
    int data;
    int color ;
    struct Node *left, *right, *parent;
};

// Utility function to create a new Red-Black Tree node
struct Node* create Node(int data) {
    struct Node* new Node = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->color = RED; // New node is always RED initially
    newNode->left = new Node->right = newNode->parent = NULL;
    return new Node;
}

// Function to perform 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 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;
}

// Function to fix Red-Black Tree property violations after insertion
void fixViolation(struct Node** root, struct Node* z) {
    while (z != *root && z->parent->color == RED) {
        struct Node* grandparent = z->parent->parent;

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

            if (uncle != NULL && uncle->color == RED) {
                // Case 1: Uncle is red
                grandparent->color = RED;
                z->parent->color = BLACK;
                uncle->color = BLACK;
                z = grandparent;
            } else {
                // Case 2: Uncle is black, zig-zag case
                if (z == z->parent->right) {
                    z = z->parent;
                    leftRotate(root, z);
                }
```

```
            // Case 3: Uncle is black, zig-zig case
            z->parent->color = BLACK;
            grandparent->color = RED;
            rightRotate(root, grandparent);
        }
    } else {
        struct Node* uncle = grandparent->left;

        if (uncle != NULL && uncle->color == RED) {
            // Case 1: Uncle is red
            grandparent->color = RED;
            z->parent->color = BLACK;
            uncle->color = BLACK;
            z = grandparent;
        } else {
            // Case 2: Uncle is black, zig-zag case
            if (z == z->parent->left) {
                z = z->parent;
                rightRotate(root, z);
            }

            // Case 3: Uncle is black, zig-zig case
            z->parent->color = BLACK;
            grandparent->color = RED;
            leftRotate(root, grandparent);
        }
    }
}

(*root)->color = BLACK;
}

// Helper function to insert a new node in the Red-Black Tree (as in a Binary Search
Tree)
struct Node* bstInsert(struct Node* root, struct Node* parent, int data) {
    if (root == NULL) {
        struct Node* newNode = createNode(data);
        newNode->parent = parent;
        return newNode;
    }

    if (data < root->data)
        root->left = bstInsert(root->left, root, data);
    else if (data > root->data)
        root->right = bstInsert(root->right, root, data);

    return root;
}

// Function to insert a node in the Red-Black Tree
```

```
void insert(struct Node** root, int data) {
    *root = bstInsert(*root, NULL, data);
    struct Node* newNode = *root;

    while (newNode->data != data) {
        if (data < newNode->data)
            newNode = newNode->left;
        else
            newNode = newNode->right;
    }

    fixViolation(root, newNode);
}

// In-order traversal of the Red-Black Tree
void inorder(struct Node* root) {
    if (root == NULL)
        return;

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

// Utility function to print the color of a node
const char* getColor(struct Node* node) {
    return node->color == RED ? "Red" : "Black";
}

// Function to print the Red-Black Tree structure
void printTree(struct Node* root, int space) {
    if (root == NULL)
        return;

    space += 10;

    printTree(root->right, space);

    printf("\n");
    for (int i = 10; i < space; i++)
        printf(" ");
    printf("%d(%s)\n", root->data, getColor(root));

    printTree(root->left, space);
}

// Main function
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:\n");
    inorder(root);
    printf("\n\nRed-Black Tree Structure:\n");
    printTree(root, 0);

    return 0;
}
```

OUTPUT

```
In-order Traversal of the Red-Black Tree:
10 20 25 30 40 50

Red-Black Tree Structure:

        50(Black)

    40(Red)

        30(Black)

            25(Red)

10(Black)

    20(Red)
```