ADVANCED DATA STRUCTURE

ASSIGNMENT

Submitted to:

Ms .AKSHARA SASIDHARAN

DEPARTMENT OF COMPUTER APPLICATION

Submitted from:

SOBIN THOMAS SONY

S1 MCA

ROLL NO:58

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:

- **Range of interest**: You only care about scores above 50, i.e., the scores between 51 and 100. This means you need to track 50 possible scores (from 51 to 100).
- **Efficiency**: Using an array is efficient in terms of both space and time. The array can be indexed directly using the score to store and update frequencies in constant time, which is faster than using other data structures like hashmaps.
- **Space**: An array of size 50 is a minimal and sufficient storage structure for this problem.

Steps to Implement:

1. **Create an Array for Frequencies**:
   Since there are only 50 scores of interest (from 51 to 100), create an array (list) of size 50. Initialize all values to 0 (because initially, no scores have been encountered).

   frequency = [0] * 50

2. **Read the Scores**:
   Loop through the 500 student scores and for each score greater than 50, update its corresponding position in the array.

3. **Map Scores to Indices**:
   - If a score is 51, it should correspond to index 0 in the array.

- If a score is 100, it should correspond to index 49. You can achieve this by subtracting 51 from the score to map it to the correct index in the frequency array.

4. **Print the Frequencies**:
   After updating the frequency array, print the frequencies of all the scores above 50.

```
frequency = [0] * 50
scores = [/* 500 integers between 0 and 100 */]
for score in scores:
    if score > 50:
        index = score - 51  # Map score 51 to index 0, score 52 to index 1, etc.
        frequency[index] += 1
for i in range(50):
    if frequency[i] > 0:  # Only print scores that occurred at least once
        print(f"Score {i + 51}: {frequency[i]} occurrences")
```

Explanation:

- **Step 1**: The frequency array is initialized to store the frequencies of scores from 51 to 100.
- **Step 2**: You loop through the 500 input scores (assumed to be stored in a list).
- **Step 3**: For each score, if it is greater than 50, you calculate the appropriate index in the frequency array and increment the count at that index.

- **Step 4**: Finally, you print the frequencies of each score that appeared at least once.

Benefits of Using an Array:

- **Fast access and updates**: The array allows constant-time updates to the frequency counts.
- **Memory-efficient**: Only 50 integers (the size of the array) are used to store the frequencies.

2) Consider a standard Circular Queue \'q\' implementation(which has the same condition for Queue Full and QueueEmpty) 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, the positions for adding and removing elements are handled using the **front** and **rear** pointers. The queue wraps around when it reaches the end of the array, hence the term "circular."

Problem Breakdown:

- The circular queue q has a size of 11, meaning it can store up to 11 elements in q[0] to q[10].
- The **front** and **rear** pointers are both initialized to point at q[2].
- We want to find the position where the **ninth element** will be added.

Key Points to Remember:

1. **Rear Pointer**: The rear pointer points to the position where the next element will be added.

2. **Modulo Operation**: Since it's a circular queue, once the rear pointer reaches the last position (q[10]), it wraps around back to q[0]. This can be achieved using the modulo operation with the size of the queue.
3. **First Element**: The first element will be added to the position pointed to by the **rear** (which is initially at q[2]).

Step-by-Step Calculation:

1. **Initial position**: The rear pointer starts at q[2].
2. The first element will be added at q[2] (current position of the rear).
3. After adding an element, the rear pointer moves to the next position. This can be calculated as (rear + 1) % 11.
4. Add subsequent elements and calculate the new rear pointer position for each:
    - 2nd element: (2 + 1) % 11 = 3, so added at q[3].
    - 3rd element: (3 + 1) % 11 = 4, so added at q[4].
    - 4th element: (4 + 1) % 11 = 5, so added at q[5].
    - 5th element: (5 + 1) % 11 = 6, so added at q[6].
    - 6th element: (6 + 1) % 11 = 7, so added at q[7].
    - 7th element: (7 + 1) % 11 = 8, so added at q[8].
    - 8th element: (8 + 1) % 11 = 9, so added at q[9].
    - 9th element: (9 + 1) % 11 = 10, so added at q[10].

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>
typedef enum { RED, BLACK } Color;
typedef struct RBNode {
   int data;
   Color color;
   struct RBNode *left;
   struct RBNode *right;
   struct RBNode *parent;
} RBNode;
RBNode* createNode(int data) {
   RBNode* node = (RBNode*)malloc(sizeof(RBNode));
   node->data = data;
   node->color = RED;
   node->left = NULL;
   node->right = NULL;
   node->parent = NULL;
   return node;
}
void leftRotate(RBNode **root, RBNode *x) {
   RBNode *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(RBNode **root, RBNode *y) {
    RBNode *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;
}
void fixViolation(RBNode **root, RBNode *z) {
    while (z != *root && z->parent->color == RED) {
        RBNode *grandparent = z->parent->parent;

        if (z->parent == grandparent->left) {
            RBNode *uncle = grandparent->right;
            if (uncle != NULL && uncle->color == RED) {
                z->parent->color = BLACK;
                uncle->color = BLACK;
                grandparent->color = RED;
                z = grandparent;
            }
            else {
```

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

```
            leftRotate(root, grandparent);
        }
    }
}

(*root)->color = BLACK;
}
void insert(RBNode **root, int data) {
RBNode *z = createNode(data);
RBNode *y = NULL;
RBNode *x = *root;

while (x != NULL) {
    y = x;
    if (z->data < x->data)
        x = x->left;
    else
        x = x->right;
}

z->parent = y;

if (y == NULL)
    *root = z;
```

```
    else if (z->data < y->data)
        y->left = z;
    else
        y->right = z;
    fixViolation(root, z);
}
void inorder(RBNode *root) {
    if (root == NULL)
        return;

    inorder(root->left);
    printf("%d (%s) ", root->data, (root->color == RED) ? "RED" :
"BLACK");
    inorder(root->right);
}

int main() {
    RBNode *root = NULL;
    int elements[] = {10, 20, 30, 15, 25, 40, 50};
    int n = sizeof(elements) / sizeof(elements[0]);

    for (int i = 0; i < n; i++) {
        insert(&root, elements[i]);
    }
```

```
    printf("Inorder Traversal of the Red-Black Tree:\n");
    inorder(root);
    printf("\n");

    return 0;
}
```