ADVANCED DATA STRUCTURE ASSIGNMENT

Submitted to:

Ms .AKSHARA SASIDHARAN

DEPARTMENT OF COMPUTER APPLICATION

Submitted from: ARJUN M

S1 MCA

ROLL NO:25

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:

- Target Scores: Only scores between 51 and 100 are of interest, meaning we need to track a total of 50 scores.

- Efficiency: Using an array is optimal here as it allows quick access and updates, which is faster than other data structures like linked lists or hashmaps.

- Memory Use: A simple array of size 50 is sufficient to store the frequency of scores from 51 to 100.

Steps to Implement:

- Create a Frequency Array: An array of size 50 will be used, initialized with zero to count occurrences of scores between 51 and 100.

```
int frequency[50] = {0};
```

- Read and Process the Scores: Iterate over the 500 student scores and for each score above 50, update the corresponding index in the frequency array.

- Map Scores to Array Indices: For example, a score of 51 will be mapped to index 0 in the array, and a score of 100 will correspond to index 49.

- Print the Frequencies: After updating the array, print the frequencies for the scores that occurred at least once.

```
#include <stdio.h>

int main() {
   int frequency[50] = {0};
   int scores[500];
   // Example: Initialize scores array (replace with input logic)
   for (int i = 0; i < 500; i++) {
      scores[i] = rand() % 101;
   }

   // Process each score
   for (int i = 0; i < 500; i++) {
      if (scores[i] > 50) {
```

```
            int index = scores[i] - 51;
            frequency[index]++;
        }
    }

    // Print frequencies of scores above 50
    for (int i = 0; i < 50; i++) {
        if (frequency[i] > 0) {
            printf("Score %d: %d occurrences\n", i + 51, frequency[i]);
        }
    }

    return 0;
}
```

Explanation:
Step 1: The frequency array stores counts for scores between 51 and 100.
Step 2: Each score is processed, and if it is above 50, its corresponding index in the array is updated.
Step 3: The mapping is done by subtracting 51 from the score to determine its position in the array.
Step 4: The final frequencies are printed for scores that appeared at least once.

Advantages of Using an Array:

- Fast Access: Updates and accesses are done in constant time.
- Memory Efficiency: Only 50 integers are needed to store the frequency counts.

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 this problem, a circular queue of size 11 is given, with the elements stored in positions q[0] to q[10]. Both the front and rear pointers are initialized at q[2]. The goal is to determine the position of the ninth element to be added.

In a circular queue, both addition and removal of elements wrap around the array when the end is reached. The positions are handled using the front and rear pointers.

Modulo Operation: The rear pointer wraps around when it reaches the last position (q[10]). This can be calculated using the modulo operation (rear + 1) % size.

Step-by-Step Calculation:

Initial Position: Both the front and rear pointers are at q[2].

Element Addition: The first element is added at q[2] (the initial rear position), and after each addition, the rear pointer is updated as (rear + 1) % 11.

Adding Nine Elements: The positions are calculated as follows:

1st element at q[2]

2nd element at q[3]

3rd element at q[4]

4th element at q[5]

5th element at q[6]

6th element at q[7]

7th element at q[8]

8th element at q[9]

9th element at q[10]

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

3) Write a C Program to implement Red Black Tree

Answer:

A Red-Black Tree is a balanced binary search tree with additional properties that ensure the tree remains balanced, allowing for efficient insertion, deletion, and search operations. Each node in a Red-Black Tree contains an extra bit that signifies the color of the node, either red or black.

Key Properties of Red-Black Tree:

Every node is either red or black.

The root is always black.

All leaves (NIL nodes) are black.

If a node is red, its children must be black (no two consecutive red nodes).

Every path from a node to its descendant leaves must have the same number of black nodes.

C Code Implementation:

```
#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;
}
```