ASSIGNMENT-2

Advanced Data Structures

Submitted To

Akshara Ma'am

Submitted By

Amritha S

S1 MCA

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?

Generally the best data structure that we used to store the frequencies of the scores above 50 for the 500 integers is array. The size of array will be 51

Algorithm

Step 1 : Create an array name arr with size 51

```
int arr(51);
```

Step 2 : In this array the index will represent the scores from 51 to 100 like the below

arr[0] represents the count of the score 51,

arr[1] represents the count of the score 52,

....

…………………………………………………………………

arr[49] represents the count of the score 100

Step 3 : Read each score , if the score is greater than 50 , increment the corresponding index in the arr array .You can calculate the index as score -51.

Step 4 : After processing all scores, iterate through the frequency array and print the count for each score from 51 to 100.

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?

In the we are using the data structure name circular queue, that the last node of the queue is connected to the front of the queue. Once it reaches the last node, it continues from the beginning.

Here the Queue size is 11 that index from 0 to 10. In a circular queue we are using two pointers such as rear and front. The front pointer points to the front node of the queue and the rear pointer points to the last element in the queue.   Given that front and rear of the queue is at q [2].

When we add elements to the queue the rear moves forward and it will wrap around when it reaches the end of the array. So adding elements will be like this,

- The first element will be added at q [3], which is one position after the current rear (q [2]).
- The second element will be added at q [4].
- The third element will be added at q [5].
- The fourth element will be added at q [6].
- The fifth element will be added at q [7].
- The sixth element will be added at q [8].

- The seventh element will be added at q [9].
- The eighth element will be added at q [10].
- **The ninth element will be added at q [0]**, because the queue wraps around and came to front of the circular queue that is the front is now q [0].

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

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

enum Color { RED, BLACK };

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

// Utility function to create a new node
struct Node* createNode(int data) {
struct Node* newNode = (struct Node*)malloc(sizeof(struct
Node));
newNode->data = data;
newNode->color = RED; // New nodes are  inserted as RED
```

```
newNode->left = newNode->right = newNode->parent =
NULL;
return newNode;
}

// Perform an in-order traversal of the Red-Black Tree
void inorderTraversal(struct Node* root) {
if (root == NULL) return;

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

// Left rotation
struct Node* rotateLeft(struct Node* root, struct Node* pt) {
struct Node* pt_right = pt->right;

pt->right = pt_right->left;

if (pt->right != NULL)
pt->right->parent = pt;

pt_right->parent = pt->parent;

if (pt->parent == NULL)
root = pt_right;
```

```
else if (pt == pt->parent->left)
pt->parent->left = pt_right;
else
pt->parent->right = pt_right;

pt_right->left = pt;
pt->parent = pt_right;

return root;
}

// Right rotation
struct Node* rotateRight(struct Node* root, struct Node* pt)
{
struct Node* pt_left = pt->left;

pt->left = pt_left->right;

if (pt->left != NULL)
pt->left->parent = pt;

pt_left->parent = pt->parent;

if (pt->parent == NULL)
root = pt_left;
else if (pt == pt->parent->left)
pt->parent->left = pt_left;
```

```
else
pt->parent->right = pt_left;

pt_left->right = pt;
pt->parent = pt_left;

return root;
}

// Fix Red-Black Tree violations
struct Node* fixViolation(struct Node* root, struct Node* pt)
{
struct Node* parent_pt = NULL;
struct Node* grand_parent_pt = NULL;

while ((pt != root) && (pt->color != BLACK) && (pt->parent-
>color == RED)) {
parent_pt = pt->parent;
grand_parent_pt = pt->parent->parent;

// Case A: Parent is left child of grandparent
if (parent_pt == grand_parent_pt->left) {
struct Node* uncle_pt = grand_parent_pt->right;

// Case 1: The uncle is RED, recoloring
if (uncle_pt != NULL && uncle_pt->color == RED) {
grand_parent_pt->color = RED;
```

```
parent_pt->color = BLACK;
uncle_pt->color = BLACK;
pt = grand_parent_pt;
} else {
// Case 2: pt is right child of its parent, left-rotation needed
if (pt == parent_pt->right) {
root = rotateLeft(root, parent_pt);
pt = parent_pt;
parent_pt = pt->parent;
}

// Case 3: pt is left child, right-rotation needed
root = rotateRight(root, grand_parent_pt);
enum Color temp = parent_pt->color;
parent_pt->color = grand_parent_pt->color;
grand_parent_pt->color = temp;
pt = parent_pt;
}
}
// Case B: Parent is right child of grandparent
else {
struct Node* uncle_pt = grand_parent_pt->left;

// Case 1: The uncle is RED, recoloring
if (uncle_pt != NULL && uncle_pt->color == RED) {
grand_parent_pt->color = RED;
parent_pt->color = BLACK;
```

```
uncle_pt->color = BLACK;
pt = grand_parent_pt;
} else {
// Case 2: pt is left child of its parent, right-rotation needed
if (pt == parent_pt->left) {
root = rotateRight(root, parent_pt);
pt = parent_pt;
parent_pt = pt->parent;
}

// Case 3: pt is right child, left-rotation needed
root = rotateLeft(root, grand_parent_pt);
enum Color temp = parent_pt->color;
parent_pt->color = grand_parent_pt->color;
grand_parent_pt->color = temp;
pt = parent_pt;
}
}
}

root->color = BLACK;
return root;
}

// Insert a new node into the Red-Black Tree
struct Node* insert(struct Node* root, struct Node* pt) {
// Binary search tree insertion
```

```
if (root == NULL)
return pt;

if (pt->data < root->data) {
root->left = insert(root->left, pt);
root->left->parent = root;
} else if (pt->data > root->data) {
root->right = insert(root->right, pt);
root->right->parent = root;
}

return root;
}

// Main function to insert a new node and fix Red-Black Tree
violations
struct Node* insertRBTree(struct Node* root, int data) {
struct Node* pt = createNode(data);
root = insert(root, pt);
root = fixViolation(root, pt);
return root;
}

int main() {
struct Node* root = NULL;

// Insert nodes
```

```
root = insertRBTree(root, 10);
root = insertRBTree(root, 20);
root = insertRBTree(root, 30);
root = insertRBTree(root, 15);
root = insertRBTree(root, 25);

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

return 0;
}
```