21 SEPTEMBER 2024

ASSIGNMENT -1

Advanced Data Structures

Submitted to:

Ms.Akshara Sasidharan

Dept.of Computer Applications

Submitted by:

Anita Antony

S1 MCA 24-26

Rollno:18

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:

The efficient way to store the frequencies of scores above 50 is using an array of size 51. It avoids storing frequencies for scores 0 to 50, which are not required, thus saving space.

Steps:

1. **Array Initialization**: Create an array frequency of size 51 to store frequencies of scores from 51 to 100.
2. **Input and Counting**: For each score, check if it's greater than 50. If it is, calculate the index as score - 51 and increment the corresponding value in the array.
3. **Output**: After processing all scores, iterate through the frequency array and print the frequencies of scores from 51 to 100.

Breakdown:

1. **Array of size 51**: This array saves space by only storing frequencies for scores greater than 50. Each index in the array corresponds to scores 51 through 100.
2. **Efficient index calculation**: The index for a score is score - 51, ensuring that score 51 is stored at index 0, score 52 at index 1, and so on.
3. **Space Efficiency**: You avoid using space for scores below 51, making this a compact and efficient solution.

This method is both space and time efficient while meeting the requirement of only storing frequencies for scores greater than 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 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?

Solution:

Given that the queue has a size of 11 and both the front and rear pointers start at q[2], let's track the positions as elements are added.

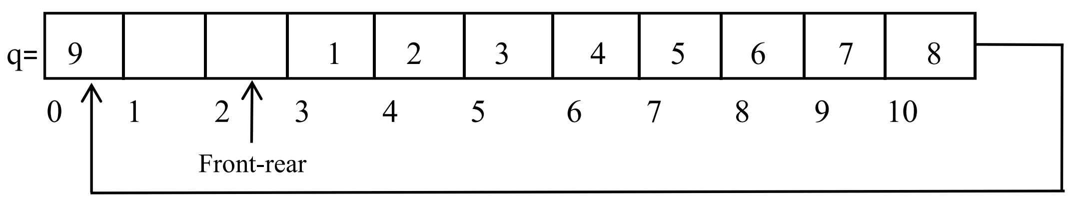
Initially,

Front = 2
Rear = 2

In a circular queue, when inserting an element, the rear pointer is updated as:

$$\textbf{rear=(rear+1)\%n}$$

* Initially, both the front and rear pointers are at q[2].
* After the first element is added, the rear pointer moves to q[3].
* For the second element rear pointer moves to q[4]
* For the third element, it moves to q[5].
* For the fourth element, it moves to q[6].
* For the fifth element, it moves to q[7].
* For the sixth element, it moves to q[8].
* For the seventh element, it moves to q[9].
* For the eighth element, it moves to q[10].
* For the ninth element, it will wrap around to q[0] since q[10] is the last position.
* Thus, the ninth element will be added at position q[0]



3.Write a C Program to implement Red Black Tree

Solution:

```
#include <stdio.h>
#include <stdlib.h>
#define RED 0
#define BLACK 1

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

Node* createNode(int data);
Node* rotateLeft(Node **root, Node *x);
Node* rotateRight(Node **root, Node *x);
void fixViolation(Node **root, Node *node);
void insertNode(Node **root, Node *dataNode);
void inorderTraversal(Node *root);
void printTree(Node *root, int space);

Node* createNode(int data) {
        Node *newNode = (Node *)malloc(sizeof(Node));
        newNode->data = data;
        newNode->left = newNode->right = newNode->parent = NULL;
        newNode->color = RED;
        return newNode;
    }

Node* rotateLeft(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)
                *root = y;
        else if (x == x->parent->left)
                x->parent->left = y;
        else
                x->parent->right = y;
        y->left = x;
        x->parent = y;
        return *root;
    }

Node* rotateRight(Node **root, Node *x) {
        Node *y = x->left;
        x->left = y->right;
        if (y->right != NULL)
                y->right->parent = x;
        y->parent = x->parent;
        if (x->parent == NULL)
                *root = y;
        else if (x == x->parent->right)
                x->parent->right = y;
        else
                x->parent->left = y;
        y->right = x;
        x->parent = y;
        return *root;
   }

void fixViolation(Node **root, Node *node) {
        Node *parent, *grandParent;
```

```
while ((node != *root) && (node->parent->color == RED)) {
        parent = node->parent;
        grandParent = parent->parent;
        if (parent == grandParent->left) {
                Node *uncle = grandParent->right;
                if (uncle && uncle->color == RED) {
                        parent->color = BLACK;
                        uncle->color = BLACK;
                        grandParent->color = RED;
                        node = grandParent;
                }
                else {
                        if (node == parent->right) {
                                node = parent;
                                *root = rotateLeft(root, node);
                            }
                        parent->color = BLACK;
                        grandParent->color = RED;
                        *root = rotateRight(root, grandParent);
                    }
        }
        else {
                Node *uncle = grandParent->left;
                if (uncle && uncle->color == RED) {
                        parent->color = BLACK;
                        uncle->color = BLACK;
                        grandParent->color = RED;
                        node = grandParent;
                    }
                else {
                        if (node == parent->left) {
                                node = parent;
                                *root = rotateRight(root, node);
                        }
```

```
                    parent->color = BLACK;
                    grandParent->color = RED;
                    *root = rotateLeft(root, grandParent);
                    }
            }
        }
        (*root)->color = BLACK;
}

void insertNode(Node **root, Node *dataNode) {
        Node *parent = NULL;
        Node *current = *root;
        while (current != NULL) {
                parent = current;
                if (dataNode->data < current->data)
                        current = current->left;
                else
                        current = current->right;
                }
        dataNode->parent = parent;
        if (parent == NULL) {
                *root = dataNode;
            }
         else if (dataNode->data < parent->data) {
                parent->left = dataNode;
           }
         else {
                parent->right = dataNode;
        }
        fixViolation(root, dataNode);
}

void inorderTraversal(Node *root) {
        if (root != NULL) {
```

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

void printTree(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, root->color == RED ? "RED" : "BLACK");
        printTree(root->left, space);
}

int main() {
        Node *root = NULL;
        insertNode(&root, createNode(10));
        insertNode(&root, createNode(20));
        insertNode(&root, createNode(30));
        insertNode(&root, createNode(15));
        insertNode(&root, createNode(25));

        printf("Inorder Traversal:\n");
        inorderTraversal(root);
        printf("\n");

        printf("Red-Black Tree Visualization:\n");
        printTree(root, 0);
        return 0;
}
```

Output:

Inorder Traversal:

10 15 20 25 30

Red-Black Tree Visualization:

```
        30(BLACK)

                        25(RED)

20(BLACK)

                15(RED)

        10(BLACK)
```