Advanced Data Structure

Assignment -1

Submitted to,

AKSHARA SASIDHARAN

Submitted by,

CHRISTIN BENNY

S1 MCA

Q2) 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

1. **Array Initialization**: Create an array of size 51 (to hold frequencies for scores 51 to 100). You could also use an array of size 101 if you want to store all scores from 0 to 100 but only use indices 51 to 100 for the relevant frequencies.
2. **Reading Scores**: As you read each score, check if it is greater than 50. If it is, increment the corresponding index in the frequency array.
3. **Output Frequencies**: After processing all scores, iterate through the frequency array from index 51 to 100 and print the frequencies for those scores.

Q5) Consider a standard Circular Queue 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?

ANSWER

Front: 2

Rear: 2

Queue Size: 11

First Element: Added at q[2] (Rear moves to 2).

Second Element: Rear moves to 3, element added at q[3].

Third Element: Rear moves to 4, element added at q[4].

Fourth Element: Rear moves to 5, element added at q[5].

Fifth Element: Rear moves to 6, element added at q[6].

Sixth Element: Rear moves to 7, element added at q[7].

Seventh Element: Rear moves to 8, element added at q[8].

Eighth Element: Rear moves to 9, element added at q[9].

Ninth Element: Rear moves to 10, element added at q[10].

The ninth element will be added at position q[10]. After this, the rear pointer will wrap around to 0 for subsequent additions, provided there's space available.

Q6) Implementation of RB tree

ANSWER

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int data;
    struct Node *parent, *left, *right;
    int color; // 0 for red, 1 for black
};
struct Node *root = NULL;
struct Node *newNode(int data) {
    struct Node *node = (struct Node *)malloc(sizeof(struct Node));
    node->data = data;
    node->parent = node->left = node->right = NULL;
    node->color = 0; // Red by default
    return node;
}
void leftRotate(struct Node *node) {
    struct Node *rightChild = node->right;
    node->right = rightChild->left;
    if (rightChild->left != NULL) {
```

```
        rightChild->left->parent = node;
    }
    rightChild->parent = node->parent;
    if (node->parent == NULL) {
        root = rightChild;
    } else if (node == node->parent->left) {
        node->parent->left = rightChild;
    } else {
        node->parent->right = rightChild;
    }
    rightChild->left = node;
    node->parent = rightChild;
}
void rightRotate(struct Node *node) {
    struct Node *leftChild = node->left;
    node->left = leftChild->right;
    if (leftChild->right != NULL) {
        leftChild->right->parent = node;
    }
    leftChild->parent = node->parent;
    if (node->parent == NULL) {
        root = leftChild;
```

```
    } else if (node == node->parent->left) {
        node->parent->left = leftChild;
    } else {
        node->parent->right = rightChild;
    }
    leftChild->right = node;
    node->parent = leftChild;
}
void fixInsertion(struct Node *node) {
    while (node->parent != NULL && node->parent->color == 0) {
        if (node->parent == node->parent->parent->left) {
            struct Node *uncle = node->parent->parent->right;
            if (uncle->color == 0) {
                uncle->color = 1;
                node->parent->color = 1;
                node->parent->parent->color = 0;
                node = node->parent->parent;
            } else {
                if (node == node->parent->right) {
                    node = node->parent;
                    leftRotate(node);
                }
```

```
            node->parent->color = 1;
            node->parent->parent->color = 0;
            rightRotate(node->parent->parent);
        }
    } else {
        // Symmetric case, handle right uncle
    }
  }
  root->color = 1;
}
void insertNode(struct Node *node) {
  struct Node *y = NULL;
  struct Node *x = root;
  while (x != NULL) {
    y = x;
    if (node->data < x->data) {
      x = x->left;
    } else {
      x = x->right;
    }
  }
  node->parent = y;
```

```
    if (y == NULL) {
        root = node;
    } else if (node->data < y->data) {
        y->left = node;
    } else {
        y->right = node;
    }
    node->left = node->right = NULL;
    node->color = 0;
    fixInsertion(node);
}
void printInorder(struct Node *node) {
    if (node != NULL) {
        printInorder(node->left);
        printf("%d ", node->data);
        printInorder(node->right);
    }
}
int main() {
    int numNodes;
    printf("Enter the number of nodes: ");
    scanf("%d", &numNodes);
```

```
    for (int i = 0; i < numNodes; i++) {
        int data;
        printf("Enter data for node %d: ", i + 1);
        scanf("%d", &data);
        insertNode(newNode(data));
    }
    printf("Inorder traversal of the red-black tree:\n");
    printInorder(root);
    return 0;
}
```