ASSIGNMENT 1

2)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?

Ans:

By creating an array of 50 elements.

Explanation:

Create an array (frequency) of 50 slots where each slots represents each score ranging from 51 – 100. That is, for slot 0 will count the number of times 51 appears, slot 1 represents the number of times 52 appears and so on up to slot 49 representing the number of times 100 appears.

Each time you read a score in program P, subtract 51 from the score and the result will be stored in the corresponding index number of the array frequency[50].
Eg:

If score is 53
index = 53 – 51 = 2
therefore in slot 2, the value will be incremented by 1.

Once you have processed all the scores, print the array.

5) 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?

Ans:

Element 9 will be at q[0].

Explaination:

Front and rear pointers are pointed at q[2]. When we add an element, the rear pointer is moved to the next position (here q[3]) where the new element will be added. Therefore,

Element 1 will be in q[3]
Element 2 in q[4]
Element 3 in q[5]
Element 4 in q[6]
Element 5 in q[7]
Element 6 in q[8]
Element 7 in q[9]
Element 8 in q[10]
Element 9 in q[0]

Therefore element 9 is in q[0].

6) Write a C Program to implement Red Black Tree

Ans:

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

// Enum to define node color
enum Color { RED, BLACK };

// Structure to represent a node in the Red-Black Tree
struct Node {
  int data;
  enum Color color;
  struct Node *left, *right, *parent;
};

// Function to create a new Red-Black Tree node
struct Node* createNode(int data, struct Node* parent) {
  struct Node* node = (struct Node*)malloc(sizeof(struct Node));
  node->data = data;
  node->color = RED; // New nodes are always red
  node->left = node->right = NULL;
  node->parent = parent;
  return node;
}

// Left rotation for balancing the Red-Black Tree
struct Node* leftRotate(struct Node* root, struct Node* x) {
  struct 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;
}

// Right rotation for balancing the Red-Black Tree
struct Node* rightRotate(struct Node* root, struct Node* y) {
    struct Node* 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->left)
        y->parent->left = x;
    else
```

```
        y->parent->right = x;

    x->right = y;
    y->parent = x;
    return root;
}

// Function to fix Red-Black Tree violations
struct Node* fixViolation(struct Node* root, struct Node* z) {
    while (z != root && z->parent->color == RED) {
        if (z->parent == z->parent->parent->left) {
            struct Node* y = z->parent->parent->right; // Uncle
            if (y != NULL && y->color == RED) {
                z->parent->color = BLACK;
                y->color = BLACK;
                z->parent->parent->color = RED;
                z = z->parent->parent;
            } else {
                if (z == z->parent->right) {
                    z = z->parent;
                    root = leftRotate(root, z);
                }
                z->parent->color = BLACK;
                z->parent->parent->color = RED;
                root = rightRotate(root, z->parent->parent);
            }
        } else {
            struct Node* y = z->parent->parent->left; // Uncle
            if (y != NULL && y->color == RED) {
                z->parent->color = BLACK;
                y->color = BLACK;
```

```
            z->parent->parent->color = RED;
            z = z->parent->parent;
        } else {
            if (z == z->parent->left) {
                z = z->parent;
                root = rightRotate(root, z);
            }
            z->parent->color = BLACK;
            z->parent->parent->color = RED;
            root = leftRotate(root, z->parent->parent);
        }
    }
  }
  root->color = BLACK;
  return root;
}

// Insert function to add a new node in Red-Black Tree
struct Node* insert(struct Node* root, int data) {
  struct Node* parent = NULL;
  struct Node* x = root;

  // Find the correct position for the new node
  while (x != NULL) {
    parent = x;
    if (data < x->data)
      x = x->left;
    else
      x = x->right;
  }
```

```
    // Create the new node
    struct Node* newNode = createNode(data, parent);

    // Attach it to the parent
    if (parent == NULL)
        root = newNode;
    else if (data < parent->data)
        parent->left = newNode;
    else
        parent->right = newNode;

    // Fix any violations of Red-Black Tree properties
    root = fixViolation(root, newNode);
    return root;
}

// Inorder traversal of the tree to display the elements
void inorder(struct Node* root) {
    if (root == NULL)
        return;
    inorder(root->left);
    printf("%d ", root->data);
    inorder(root->right);
}

// Main function to demonstrate the Red-Black Tree operations
int main() {
    struct Node* root = NULL;

    // Insert nodes into the Red-Black Tree
    root = insert(root, 7);
```

```
    root = insert(root, 3);
    root = insert(root, 18);
    root = insert(root, 10);
    root = insert(root, 22);
    root = insert(root, 8);
    root = insert(root, 11);
    root = insert(root, 26);

    // Print the inorder traversal of the tree
    printf("Inorder Traversal of the tree:\n");
    inorder(root);
    printf("\n");

    return 0;
}
```