ASSIGNMENT

Q1)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:

To effectively store the frequencies of scores above 50 from a list of 500 integers (ranging from 0 to 100), the best approach for program P is to utilize an array specifically designed for this purpose.

Steps for Implementation

1. Initialization: Initialize an array `frequency` to zero.
2. Counting Frequencies:
   - Iterate through the input scores.
   - For each score greater than 50, increment the corresponding index in the frequency array.
3. Output: After processing all scores, iterate through the frequency array from index 51 to 100 and print out any non-zero counts.

Q2.)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?

Answer:

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

Positions of Insertion:

- 1st element: R = 2 (adds to q[2])
- 2nd element: R = 3 (adds to q[3])
- 3rd element: R = 4 (adds to q[4])
- 4th element: R = 5 (adds to q[5])

- 5th element: R = 6 (adds to `q[6]`)
- 6th element: R = 7 (adds to `q[7]`)
- 7th element: R = 8 (adds to `q[8]`)
- 8th element: R = 9 (adds to `q[9]`)
- 9th element: R = 10 (adds 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].

Q3) Write a C Program to implement Red Black Tree ?

Answer:

```
#include <stdio.h>

#include <stdlib.h>

 struct node {

   int data;     char color;

   struct node *left, *right, *parent;

};

 void leftRotate(struct node **root, struct node *x);

void rightRotate(struct node **root, struct node *y);

void insertFixUp(struct node **root, struct node *z);

void insert(struct node **root, int data);

void inorder(struct node *root);

 void 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; // y becomes the new root
    else if (x == x->parent->left)
        x->parent->left = y;
else
 x->parent->right = y;
    y->left = x;
    x->parent = y;
}void 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->right)
  y->parent->right = x;
  Else{
  y->parent->left = x;
  x->right = y;
  y->parent = x;
}
void insertFixUp(struct node **root, struct node *z) {
    while (z != *root && z->parent != NULL && z->parent->color == 'R') {
        struct node *y;
```

```
if (z->parent == z->parent->parent->left) {
    y = z->parent->parent->right;
    if (y != NULL && y->color == 'R') { // Case 1: Uncle is red
        z->parent->color = 'B';
        y->color = 'B';
        z->parent->parent->color = 'R';
        z = z->parent->parent;
    }
   else {
        if (z == z->parent->right) {
            z = z->parent;
            leftRotate(root, z);
        }
       z->parent->color = 'B';
        z->parent->parent->color = 'R';
        rightRotate(root, z->parent->parent);
    }
} else {
    y = z->parent->parent->left; // uncle
    if (y != NULL && y->color == 'R') {
     z->parent->color = 'B';
        y->color = 'B';
        z->parent->parent->color = 'R';
        z = z->parent->parent;
    } else { // Case 2: Uncle is black
```

```
        if (z == z->parent->left) {
          z = z->parent;
          rightRotate(root, z);
        }
        z->parent->color = 'B';
        z->parent->parent->color = 'R';
        leftRotate(root, z->parent->parent);
      }
    }
  }
  (*root)->color = 'B';
}
 void insert(struct node **root, int data) {
  struct node *z = (struct node *)malloc(sizeof(struct node));
  z -> data = data;
  z -> left = NULL;
  z -> right = NULL;
  z -> color = 'R';
  struct node *y = NULL;
  struct node *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; // Tree was empty
    } else if (z -> data < y -> data) {
        y -> left = z;
    } else {
        y -> right = z;
    }

    insertFixUp(root, z);
}

void inorder(struct node *root) {
    if (root != NULL) {
        inorder(root -> left);
        printf("%d (%c) ", root -> data, root -> color);
        inorder(root -> right);
    }
}

int main() {
    struct node *root = NULL;
    insert(&root, 10);
    insert(&root, 20);
    insert(&root, 30);
    insert(&root, 15);

    printf("Inorder traversal of the Red-Black Tree:\n");
    inorder(root);

    return 0;
}
```