Data Structure Assignment

Submitted to,

Akshara Miss

Submitted by,

Arjun Manoj

S1MCA 26

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?

An array of 50 is the best way for P to store the frequencies.

Array Size: Use an array of size 50 to store the frequency counts for scores from 51 to 100. Each index of the array will correspond to a specific score, with

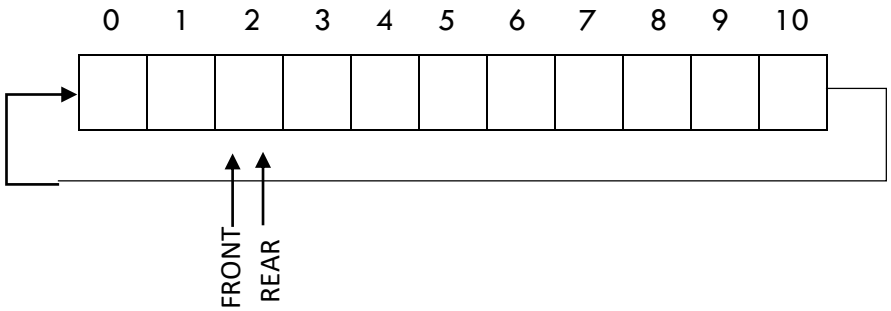
1. index 0 representing the score 51
2. index 1 representing the score 52
3. and so on, up to index 49 representing the score 100.

Initialization: Initialize the array with zeros.
Counting Frequencies: As you read each score, if it is greater than 50, increment the corresponding index in the array.
Output Frequencies: After reading all scores, iterate through the array and print the frequencies for each score from 51 to 100.

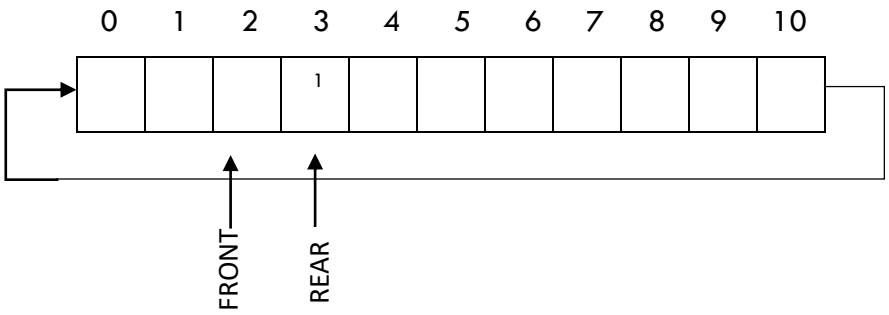
2. Consider a standard Circular Queue \q\ implementation (which has the same condition for Queue Full and Queue Empty) whose size is 11 and the elements of the queue are q[0], q[1], q[2].....,q[10]. The front and rear pointers are initialized to point at q[2] . In which position will the ninth element be added

The 9th element will be added at q[0]

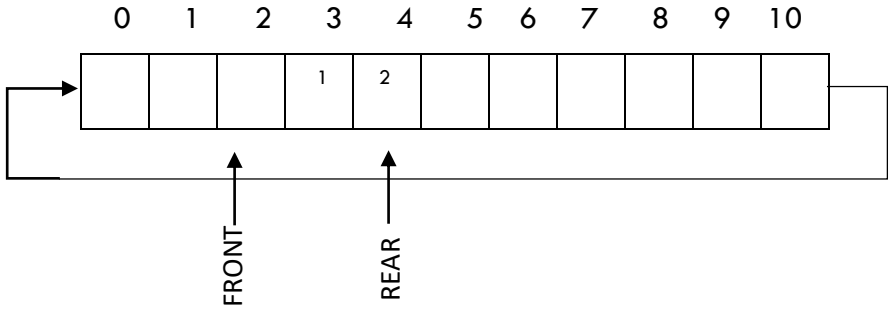
Given the front and rear points to q[2] ,Therefore the queue will look like this



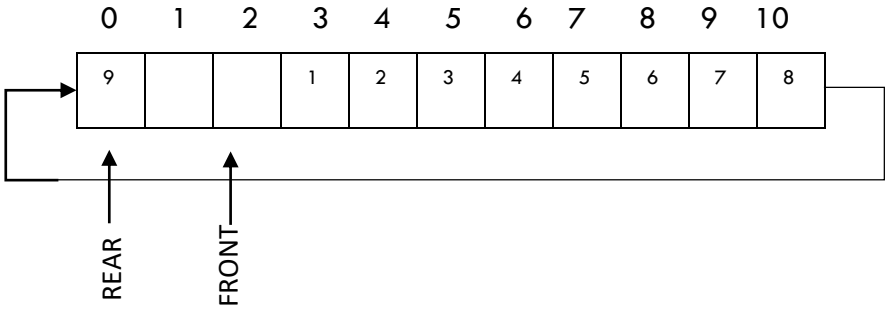
When adding the first element, rear increments and add element



When adding the second element, rear increments and add element



When adding the rest elements till 9th



3. Write a C Program to implement Red Black Tree

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

struct node {
    int d;
    int c;
    struct node* p
    struct node* r;
    struct node* l;
};
 struct node* root = NULL;
 struct node* bst(struct node* trav,
                  struct node* temp)
{
    if (trav == NULL)
        return temp;

       if (temp->d < trav->d)
    {
        trav->l = bst(trav->l, temp);
        trav->l->p = trav;
    }
    else if (temp->d > trav->d)
    {
        trav->r = bst(trav->r, temp);
        trav->r->p = trav;
    }

    return trav;
}
```

```
void rightrotate(struct node* temp)
{
    struct node* left = temp->l;
    temp->l = left->r;
    if (temp->l)
        temp->l->p = temp;
    left->p = temp->p;
    if (!temp->p)
        root = left;
    else if (temp == temp->p->l)
        temp->p->l = left;
    else
        temp->p->r = left;
    left->r = temp;
    temp->p = left;
}

// Function performing left rotation
// of the passed node
void leftrotate(struct node* temp)
{
    struct node* right = temp->r;
    temp->r = right->l;
    if (temp->r)
        temp->r->p = temp;
    right->p = temp->p;
    if (!temp->p)
        root = right;
    else if (temp == temp->p->l)
        temp->p->l = right;
    else
```

```
        temp->p->r = right;
    right->l = temp;
    temp->p = right;
}

void fixup(struct node* root, struct node* pt)
{
    struct node* parent_pt = NULL;
    struct node* grand_parent_pt = NULL;

    while ((pt != root) && (pt->c != 0)
           && (pt->p->c == 1))
    {
        parent_pt = pt->p;
        grand_parent_pt = pt->p->p;

        /*  Case : A
            Parent of pt is left child
            of Grand-parent of
           pt */
        if (parent_pt == grand_parent_pt->l)
        {

            struct node* uncle_pt = grand_parent_pt->r;

            /* Case : 1
               The uncle of pt is also red
               Only Recoloring required */
            if (uncle_pt != NULL && uncle_pt->c == 1)
            {
               grand_parent_pt->c = 1;
```

```
        parent_pt->c = 0;
        uncle_pt->c = 0;
        pt = grand_parent_pt;
    }

    else {

        /* Case : 2
             pt is right child of its parent
             Left-rotation required */
        if (pt == parent_pt->r) {
            leftrotate(parent_pt);
            pt = parent_pt;
            parent_pt = pt->p;
        }

        /* Case : 3
             pt is left child of its parent
             Right-rotation required */
        rightrotate(grand_parent_pt);
        int t = parent_pt->c;
        parent_pt->c = grand_parent_pt->c;
        grand_parent_pt->c = t;
        pt = parent_pt;
    }
}

/* Case : B
     Parent of pt is right
     child of Grand-parent of
   pt */
```

```
else {
    struct node* uncle_pt = grand_parent_pt->l;

    /*  Case : 1
        The uncle of pt is also red
        Only Recoloring required */
    if ((uncle_pt != NULL) && (uncle_pt->c == 1))
    {
        grand_parent_pt->c = 1;
        parent_pt->c = 0;
        uncle_pt->c = 0;
        pt = grand_parent_pt;
    }
    else {
        /* Case : 2
           pt is left child of its parent
           Right-rotation required */
        if (pt == parent_pt->l) {
            rightrotate(parent_pt);
            pt = parent_pt;
            parent_pt = pt->p;
        }

        /* Case : 3
             pt is right child of its parent
             Left-rotation required */
        leftrotate(grand_parent_pt);
        int t = parent_pt->c;
        parent_pt->c = grand_parent_pt->c;
        grand_parent_pt->c = t;
        pt = parent_pt;
```

```
            }
        }
    }
}
void inorder(struct node* trav)
{
    if (trav == NULL)
        return;
    inorder(trav->l);
    printf("%d ", trav->d);
    inorder(trav->r);
}
 int main()
{
    int n = 7;
    int a[7] = { 7, 6, 5, 4, 3, 2, 1 };

    for (int i = 0; i < n; i++) {

        // allocating memory to the node and initializing:
        // 1. color as red
        // 2. parent, left and right pointers as NULL
        // 3. data as i-th value in the array
        struct node* temp
            = (struct node*)malloc(sizeof(struct node));
        temp->r = NULL;
        temp->l = NULL;
        temp->p = NULL;
        temp->d = a[i];
        temp->c = 1;
         root = bst(root, temp);
```

```
        fixup(root, temp);
          root->c = 0;
    }

    printf("Inorder Traversal of Created Tree\n");
    inorder(root);

    return 0;
}
```