ADVANCED DATA STRUCTURES
ASSIGNMENT- II

Submitted to:

Ms. Akshara Sasidharan

Department Of Computer Applications

Submitted by:

Noble Sibi

S1MCA

Roll no: 51

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?

Answer:

To store the frequencies of each score above 50, the best option would be to use an array of 50 numbers. Let's understand why this is the most optimal choice:

We have 500 students and each student has a score in the range of [0, 100]. We need to find the frequency of each score above 50. Why an Array of 50 Numbers? The range of scores is from 0 to 100, but we only need to consider scores above 50. Therefore, we are only interested in scores from 51 to 100. - Since the scores above 50 are in a range of 50 numbers, an array of 50 numbers would suffice to store the frequencies.

Benefits of Using an Array of 50 Numbers:

1. Memory Efficiency: - An array of 50 numbers is more memory-efficient than an array of 100 or 500 numbers. - Storing unnecessary numbers would result in wastage of memory.

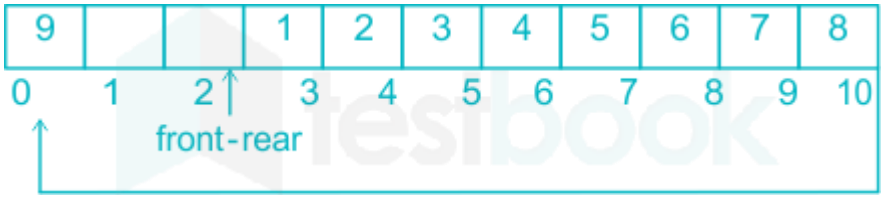
2. Time Complexity: - The time complexity of accessing an element in an array is O(1). - By using an array of 50 numbers, we can directly access the frequency of a score using its index, resulting in constant time complexity.

3. Simplicity: - An array of 50 numbers is easier to manage and understand compared to larger arrays. - It reduces the complexity of the code and makes it more readable.

4. Scalability: - If the range of scores changes in the future, for example, [0, 200], we can easily modify the array size to accommodate the new range without affecting the rest of the code. Conclusion: Using an array of 50 numbers to store the frequencies of scores above 50 is the best choice in terms of memory efficiency, time complexity, simplicity, and scalability. It optimizes both memory usage and code readability.

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?**

Answer:



The first element will get inserted in queue q[3].

Therefore, the **9th element** will get placed in **q[0].**

3) Write a C Program to implement Red Black Tree Answer:

```
#include <stdio.h>
#include <stdlib.h>
// Node colors
#define RED 0
#define BLACK 1
// Node structure typedef
struct Node {     int data;
   struct Node *left, *right, *parent;
int color;
} Node;
// Function prototypes
Node* createNode(int data);
Node* rotateLeft(Node *root, Node *x); Node*
rotateRight(Node *root, Node *x); void
fixViolation(Node *root, Node *node); Node*
insertNode(Node *root, Node *dataNode); void
inorderTraversal(Node *root); void
printTree(Node *root, int space);
// Utility function to create a new node
Node* createNode(int data) {
   Node *newNode = (Node *)malloc(sizeof(Node));     newNode-
>data = data;
   newNode->left = newNode->right = newNode->parent = NULL;
newNode->color = RED;  // New nodes are red by default     return
newNode;
}
// Utility function to perform a left rotation
```

```
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;
}
// Utility function to perform a right rotation
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;
}
// Function to fix violations of Red-Black Tree properties after insertion void
fixViolation(Node *root, Node *node) {
```

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

// Function to insert a node in the Red-Black Tree
Node* 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);
return root;
}
```

```
// Inorder Traversal of the Red-Black Tree
void inorderTraversal(Node *root) {     if
(root != NULL) {
inorderTraversal(root->left);
printf("%d ", root->data);
inorderTraversal(root->right);
   }
}
// Function to print the Red-Black Tree (for visualization)
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);
}
// Main function int
main() {
   Node *root = NULL;     // Inserting
nodes     root = insertNode(root,
createNode(10));     root = insertNode(root,
createNode(20));     root = insertNode(root,
createNode(30));     root = insertNode(root,
createNode(15));     root = insertNode(root,
createNode(25));     // Print the Red-Black
```

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