Rajalakshmi Engineering College

Name: SIVAGURU D
Email: 240701517@rajalakshmi.edu.in
Roll no: 240701517
Phone: 9345616842
Branch: REC
Department: I CSE FE
Batch: 2028
Degree: B.E - CSE

Scan to verify results

NeoColab_REC_CS23231_DATA STRUCTURES

REC_DS using C_Week 4_CY

Attempt : 1
Total Mark : 30
Marks Obtained : 30

Section 1 : Coding

1. Problem Statement

Saran is developing a simulation for a theme park where people wait in a queue for a popular ride.

Each person has a unique ticket number, and he needs to manage the queue using a linked list implementation.

Your task is to write a program for Saran that reads the number of people in the queue and their respective ticket numbers, enqueue them, and then calculate the sum of all ticket numbers to determine the total ticket value present in the queue.

Input Format

The first line of input consists of an integer N, representing the number of people

in the queue.

The second line consists of N space-separated integers, representing the ticket numbers.

Output Format

The output prints an integer representing the sum of all ticket numbers.

Refer to the sample output for formatting specifications.

Sample Test Case

Input: 5
2 4 6 7 5
Output: 24

Answer

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

typedef struct Node {
    int ticketNumber;
    struct Node* next;
} Node;

Node* enqueue(Node* rear, int ticketNumber) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    if (newNode == NULL) {
        fprintf(stderr, "Memory allocation failed\n");
        exit(EXIT_FAILURE);
    }
    newNode->ticketNumber = ticketNumber;
    newNode->next = NULL;

    if (rear == NULL) { // Queue is empty
        return newNode; // New node becomes both front and rear
    }

    // Find the last node in queue
    Node* current = rear;
```

```
    while(current->next != NULL){
        current = current->next;
    }
    current->next = newNode;
    return rear;
}

// Function to calculate the sum of ticket numbers
int calculateTicketSum(Node* front) {
    int sum = 0;
    Node* current = front;
    while (current != NULL) {
        sum += current->ticketNumber;
        current = current->next;
    }
    return sum;
}

// Function to free the queue
void freeQueue(Node* front) {
    Node* current = front;
    Node* nextNode;
    while (current != NULL) {
        nextNode = current->next;
        free(current);
        current = nextNode;
    }
}


int main() {
    int numPeople;
    scanf("%d", &numPeople);

    Node* front = NULL; // Front of the queue
    Node* rear = NULL;  // Rear of the queue

    for (int i = 0; i < numPeople; i++) {
        int ticketNumber;
        scanf("%d", &ticketNumber);
        rear = enqueue(rear, ticketNumber);
        if(front == NULL)
```

```
        front = rear;
    }

    int totalTicketValue = calculateTicketSum(front);
    printf("%d\n", totalTicketValue);

    freeQueue(front);

    return 0;
}
```

Status : Correct ***Marks : 10/10***

2. Problem Statement

A customer support system is designed to handle incoming requests using a queue. Implement a linked list-based queue where each request is represented by an integer. After processing the requests, remove any duplicate requests to ensure that each request is unique and print the remaining requests.

Input Format

The first line of input consists of an integer N, representing the number of requests to be enqueued.

The second line consists of N space-separated integers, each representing a request.

Output Format

The output prints space-separated integers after removing the duplicate requests.

Refer to the sample output for formatting specifications.

Sample Test Case

Input: 5

2 4 2 7 5

Output: 2 4 7 5

Answer

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

typedef struct Node {
    int data;
    struct Node* next;
} Node;

typedef struct Queue {
    Node* front;
    Node* rear;
} Queue;

Node* createNode(int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    if (newNode == NULL) {
        printf("Memory allocation failed.\n");
        exit(1);
    }
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

// Function to initialize a queue
Queue* createQueue() {
    Queue* queue = (Queue*)malloc(sizeof(Queue));
    if (queue == NULL) {
        printf("Memory allocation failed.\n");
        exit(1);
    }
    queue->front = queue->rear = NULL;
    return queue;
}

void enqueue(Queue* queue, int data) {
    Node* newNode = createNode(data);
    if (queue->rear == NULL) {
```

```
        queue->front = queue->rear = newNode;
        return;
    }
    queue->rear->next = newNode;
    queue->rear = newNode;
}

int dequeue(Queue* queue) {
    if (queue->front == NULL) {
        return -1;
    }
    int data = queue->front->data;
    Node* temp = queue->front;
    queue->front = queue->front->next;
    if (queue->front == NULL) {
        queue->rear = NULL;
    }
    free(temp);
    return data;
}

int contains(Queue* queue, int value) {
    Node* current = queue->front;
    while (current != NULL) {
        if (current->data == value) {
            return 1;
        }
        current = current->next;
    }
    return 0;
}

void removeDuplicates(Queue* queue) {
    if (queue->front == NULL) {
        return;
    }

    Node* current = queue->front;
    while (current != NULL) {
        Node* runner = current;
        while (runner->next != NULL) {
            if (runner->next->data == current->data) {
```

```
            Node* duplicate = runner->next;
            runner->next = runner->next->next;
            if (duplicate == queue->rear) {
                queue->rear = runner;
            }
            free(duplicate);
        } else {
            runner = runner->next;
        }
    }
    current = current->next;
  }
}

void printQueue(Queue* queue) {
  Node* current = queue->front;
  while (current != NULL) {
    printf("%d ", current->data);
    current = current->next;
  }
  printf("\n");
}


int main() {
  Queue* queue = createQueue();
  int request;
  int numRequests;

  scanf("%d", &numRequests);

  for (int i = 0; i < numRequests; i++) {
    scanf("%d", &request);
    enqueue(queue, request);
  }

  removeDuplicates(queue);

  printQueue(queue);
```

```
    while (queue->front != NULL) {
        dequeue(queue);
    }
    free(queue);

    return 0;
}
```

Status : Correct ***Marks : 10/10***

3. Problem Statement

John is working on a project to manage and analyze the data from various sensors in a manufacturing plant. Each sensor provides a sequence of integer readings, and John needs to process this data to get some insights. He wants to implement a queue to handle these sensor readings efficiently. The requirements are as follows:

Enqueue Operations: Each sensor reading needs to be added to the circular queue.Average Calculation: Calculate and print the average of every pair of consecutive sensor readings.Sum Calculation: Compute the sum of all sensor readings.Even and Odd Count: Count and print the number of even and odd sensor readings.

Assist John in implementing the program.

Input Format

The first input line contains an integer n, which represents the number of sensor readings.

The second line contains n space-separated integers, each representing a sensor reading.

Output Format

The first line should print "Averages of pairs:" followed by the averages of every pair of consecutive sensor readings, separated by spaces.

The second line should print "Sum of all elements: " followed by the sum of all sensor readings.

The third line should print "Number of even elements: " followed by the count of even sensor readings.

The fourth line should print "Number of odd elements: " followed by the count of odd sensor readings.

Refer to the sample output for the formatting specifications.

Sample Test Case

Input: 5
1 2 3 4 5
Output: Averages of pairs:
1.5 2.5 3.5 4.5 3.0
Sum of all elements: 15
Number of even elements: 2
Number of odd elements: 3

Answer

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

typedef struct Node {
    int data;
    struct Node* next;
} Node;

Node* rear = NULL;

Node* createNode(int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

void enqueue(int data) {
    Node* newNode = createNode(data);
    if (rear == NULL) {
        rear = newNode;
```

```
        newNode->next = newNode;
    } else {
        newNode->next = rear->next;
        rear->next = newNode;
        rear = newNode;
    }
}

int dequeue() {
    if (rear == NULL) {
        printf("Queue is empty\n");
        return -1;
    }

    Node* front = rear->next;
    int data = front->data;

    if (front == rear) {
        free(front);
        rear = NULL;
    } else {
        rear->next = front->next;
        free(front);
    }

    return data;
}

void printAverages(int size) {
    if (rear == NULL || size < 2) return;

    Node* temp = rear->next;
    for (int i = 0; i < size; i++) {
        int first = temp->data;
        temp = temp->next;
        int second = temp->data;
        printf("%.1f ", (first + second) / 2.0);
    }
    printf("\n");
}

void printSum() {
```

```
    if (rear == NULL) return;
    Node* temp = rear->next;
    int sum = 0;
    do {
        sum += temp->data;
        temp = temp->next;
    } while (temp != rear->next);
    printf("Sum of all elements: %d\n", sum);
}

void printEvenOddCount() {
    if (rear == NULL) return;
    Node* temp = rear->next;
    int evenCount = 0, oddCount = 0;
    do {
        if (temp->data % 2 == 0) evenCount++;
        else oddCount++;
        temp = temp->next;
    } while (temp != rear->next);
    printf("Number of even elements: %d\n", evenCount);
    printf("Number of odd elements: %d\n", oddCount);
}

int main() {
    int n;
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        int value;
        scanf("%d", &value);
        enqueue(value);
    }

    printf("Averages of pairs:\n");
    printAverages(n);
    printSum();
    printEvenOddCount();

    return 0;
}
```

Status : Correct *Marks : 10/10*