QUEUE:-

Problem Statement

You are designing an event registration system for a conference.

As part of the system, you need to implement a Queue data structure using an array that stores only the even numbers representing the registration IDs of the participants.

The queue will be used to keep track of the order in which participants register for the event.
Input format :
The first line of input consists of an integer **N**, representing the number of participants to register.
The following **N** lines consist of integers, representing the registration IDs of the participants.
Output format :
The output prints only the even number registration IDs of the participants in the order they registered, separated by space.
For odd registration IDs, print "Invalid element <element> Only even numbers can be enqueued".
Code constraints :
1 <= N <= 100
Sample test cases :
Input 1 :

```
6
2
4
6
8
10
12
```

Output 1 :

```
2 4 6 8 10 12
```

Input 2 :

```
4
14
36
55
48
```

Output 2 :

```
Invalid element 55, only even numbers can be enqueued
14 36 48
```

```
#include <iostream>


const int MAX_SIZE = 100;


struct EvenNumberQueue {

   int queue[MAX_SIZE];

   int front;
```

```
    int rear;
};

void initializeQueue(EvenNumberQueue& q) {
    q.front = -1;
    q.rear = -1;
}

bool isEmpty(const EvenNumberQueue& q) {
    return q.front == -1;
}

bool isFull(const EvenNumberQueue& q) {
    return q.rear == MAX_SIZE - 1;
}

void enqueue(EvenNumberQueue& q, int data) {
    if (isFull(q)) {
        return;
    }

    if (data % 2 == 0) {
        if (isEmpty(q)) {
            q.front = 0;
        }

        q.rear++;
        q.queue[q.rear] = data;

        // std::cout << "Enqueued: " << data << std::endl;
    } else {
```

```
        std::cout << "Invalid element "<< data <<", only even numbers can be enqueued"<< std::endl;
    }
}

void display(const EvenNumberQueue& q) {
    if (isEmpty(q)) {
        return;
    }

    for (int i = q.front; i <= q.rear; i++) {
        std::cout << q.queue[i] << " ";
    }
    std::cout << std::endl;
}

int main() {
    EvenNumberQueue queue;
    initializeQueue(queue);

    int n;
    std::cin >> n;

    for (int i = 0; i < n; i++) {
        int element;
        std::cin >> element;
        enqueue(queue, element);
    }

    display(queue);

    return 0;
```

}

Problem Statement

You are assigned to develop a program that manages customer orders in a restaurant using a queue data structure. The restaurant can handle a limited number of orders at a time, and the orders will be stored in an array-based queue.

Your task is to implement the Queue data structure and the associated functions, which will provide the necessary operations to manage the queue of customer orders using an array.

The main functionalities of the queue include:

1. **Enqueue:** Adding a customer order to the end of the queue.
2. **Get Front:** Retrieve the details of the first customer order in the queue.
3. **Get Rear:** Retrieve the details of the last customer order in the queue.

Input format :
The first line of input consists of an integer **N**, which represents the capacity of the queue.
The second line consists of **N** space-separated integers, representing the customer orders.
Output format :
The output displays the front and rear elements in the queue in the following format:
"Front element: <<front_value>>"
"Rear element: <<rear_value>>"

Refer to the sample output for the exact text and format.
Code constraints :
The customer order values are positive integers.
Sample test cases :
Input 1 :

```
5
10 20 30 40 50
```

Output 1 :

```
Front element: 10
Rear element: 50
```

Input 2 :

```
3
5 8 2
```

Output 2 :

```
Front element: 5
Rear element: 2
```

```
#include <iostream>


struct Queue {

int* arr; // array to store queue elements

int front; // front points to the front element in the queue

int rear; // rear points to the last element in the queue

int capacity; // maximum capacity of the queue
```

```
int size; // current size of the queue
};

void initializeQueue(Queue& q, int capacity) {
q.capacity = capacity;
q.arr = new int[capacity];
q.front = 0;
q.rear = -1;
q.size = 0;
}

bool isEmpty(const Queue& q) {
return q.size == 0;
}

bool isFull(const Queue& q) {
return q.size == q.capacity;
}

void enqueue(Queue& q, int item) {
if (isFull(q)) {
return;
}
q.rear = (q.rear + 1) % q.capacity;
q.arr[q.rear] = item;
q.size++;
}

int getFront(const Queue& q) {
if (isEmpty(q)) {
std::cout << "Queue is empty. No front element." << std::endl;
```

```
return -1;
}
return q.arr[q.front];
}

int getRear(const Queue& q) {
if (isEmpty(q)) {
std::cout << "Queue is empty. No rear element." << std::endl;
return -1;
}
return q.arr[q.rear];
}

int main() {
int N;
std::cin >> N;
Queue queue;
initializeQueue(queue, N); // Creating a queue of capacity N

for (int i = 0; i < N; i++) {
   int num;
   std::cin >> num;
   enqueue(queue, num);
}

std::cout << "Front element: " << getFront(queue) << std::endl;
std::cout << "Rear element: " << getRear(queue) << std::endl;

return 0;
}
```

Problem Statement

In a grocery store, customers join a single queue to check out their items. The grocery store wants to implement a system to manage the checkout queue using a data structure.

Each customer's arrival is recorded and added to the queue. The grocery store staff wants to display the queue in reverse order to check the customers' positions.

Implement a program to manage a grocery store checkout queue using a linked list. The program should allow customers to join the queue, and the order of their arrival should be reversed, meaning the first customer to join becomes the last in line. The program should display the current state of the queue.

Input format :

The first line of input consists of an integer **N**, representing the number of customers in the queue.

The next line consists of **N** space-separated integers, representing the customer positions.

Output format :

If the queue is not empty: Print the current state of the checkout queue, and display the customer positions in reverse order.

If the queue is empty: A message stating that the queue is empty.

Refer to the sample output for the exact text and format.

Code constraints :

N > 0

Sample test cases :

Input 1 :

```
3
5 10 15
```

Output 1 :

```
Queue: 15 10 5
```

Input 2 :

```
6
10 15 20 25 30 35
```

Output 2 :

```
Queue: 35 30 25 20 15 10
```

Input 3 :

```
0
```

Output 3 :

```
Queue is empty.
```

```
#include <iostream>


using namespace std;


struct Node {

   int data;

   Node* next;
```

```
    Node(int value) : data(value), next(nullptr) {}
};

class Queue {
private:
    Node* front;
    Node* rear;

public:
    Queue() : front(nullptr), rear(nullptr) {}

    bool isEmpty() {
        return front == nullptr;
    }

    void enqueue(int value) {
        Node* newNode = new Node(value);

        if (isEmpty()) {
            front = newNode;
            rear = newNode;
        } else {
            newNode->next = front;
            front = newNode;
        }

        //cout << "Enqueued: " << value << endl;
    }

    void display() {
        if (isEmpty()) {
```

```
            cout << "Queue is empty." << endl;
            return;
        }

        cout << "Queue: ";
        Node* current = front;
        while (current != nullptr) {
            cout << current->data << " ";
            current = current->next;
        }
        cout << endl;
    }
};

int main() {
    Queue queue;

    int numElements;
    cin >> numElements;

    for (int i = 0; i < numElements; i++) {
        int element;
        cin >> element;
        queue.enqueue(element);
    }

    queue.display();

    return 0;
}
```

Problem Statement

You are assigned to design and implement a program that simulates a customer queue system for a business. The program should allow users to enqueue customer IDs, dequeue customer IDs, and display the current customer queue.

Implement the queue using linked lists to manage the customers in line at the counter.

Input format :

The input consists of integer values, each representing an action to be taken:

Option 1: Enqueue a new customer with a given customer ID.

If option 1 is chosen, then the next line of input is the customer ID, which is a positive integer.

Option 2: Dequeue the customer at the front of the line.

Option 3: Display the current customer IDs in the line.

Any option other than 1, 2, or 3 will be considered invalid.

Output format :

The program provides appropriate outputs based on the choice:

When enqueuing a customer (option 1), the program outputs the customer ID that is added to the line in the format:

"Customer ID <<value>> is enqueued."

When dequeuing a customer (option 2), the program outputs the customer ID that is dequeued in the format:

"Dequeued customer ID: <<value>>"

If a dequeue operation is attempted when the queue is empty, the program outputs "Queue is empty."

When displaying the customer IDs (option 3), the program shows the customer IDs currently in the line in the format:

"Customer IDs in the queue are: <<value 1>> <<value 2>> .... <<value n>>

If display operation is attempted when the queue is empty, the program outputs "Queue is empty."

If the user provides any option other than 1, 2, or 3 then the program outputs "Invalid option."

Refer to the sample output for the exact text and format.

Code constraints :

Each customer is represented by a unique customer ID, which is a positive integer.

Sample test cases :

Input 1 :

```
1
3
1
5
1
9
2
3
```

Output 1 :

```
Customer ID 3 is enqueued.
Customer ID 5 is enqueued.
Customer ID 9 is enqueued.
Dequeued customer ID: 3
Customer IDs in the queue are: 5 9
```

Input 2 :

```
1
10
1
20
```

```
3
2
3
```

Output 2 :

```
Customer ID 10 is enqueued.
Customer ID 20 is enqueued.
Customer IDs in the queue are: 10 20
Dequeued customer ID: 10
Customer IDs in the queue are: 20
```

Input 3 :

```
1
5
3
2
3
9
```

Output 3 :

```
Customer ID 5 is enqueued.
Customer IDs in the queue are: 5
Dequeued customer ID: 5
Queue is empty.
Invalid option.
```

Input 4 :

```
2
```

Output 4 :

```
Queue is empty.
```

```
#include <iostream>

struct Node {
    int customerID;
    Node* next;
};

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

void initializeQueue(Queue* q) {
    q->front = nullptr;
    q->rear = nullptr;
```

```
}

bool isEmpty(Queue* q) {
    return q->front == nullptr;
}

void enqueue(Queue* q, int customerID) { // Modified parameter name
    Node* newNode = new Node;
    newNode->customerID = customerID; // Modified assignment
    newNode->next = nullptr;

    if (isEmpty(q)) {
        q->front = q->rear = newNode;
    } else {
        q->rear->next = newNode;
        q->rear = newNode;
    }
}

bool dequeue(Queue* q, int& customerID) { // Modified parameter name
    if (isEmpty(q)) {
        return false;
    }

    Node* temp = q->front;
    customerID = temp->customerID; // Modified assignment
    q->front = q->front->next;

    if (q->front == nullptr) {
        q->rear = nullptr;
    }
```

```
    delete temp;
    return true;
}

void display(Queue* q) {
    if (isEmpty(q)) {
        std::cout << "Queue is empty." << std::endl;
    } else {
        Node* current = q->front;
        std::cout << "Customer IDs in the queue are: ";
        while (current != nullptr) {
            std::cout << current->customerID << " ";
            current = current->next;
        }
        std::cout << std::endl;
    }
}

int main() {
    Queue q;
    int customerID; // Modified variable name
    int option;

    initializeQueue(&q);

    while (true) {
        if (!(std::cin >> option)) {
            break;
        }
```

```
        switch (option) {
            case 1:
                if (!(std::cin >> customerID)) {
                    break;
                }

                enqueue(&q, customerID);
                std::cout << "Customer ID " << customerID << " is enqueued." << std::endl;
                break;

            case 2:
                if (dequeue(&q, customerID)) {
                    std::cout << "Dequeued customer ID: " << customerID << std::endl;
                } else {
                    std::cout << "Queue is empty." << std::endl;
                }
                break;

            case 3:
                display(&q);
                break;

            default:
                std::cout << "Invalid option." << std::endl;
                break;
        }
    }

    return 0;
}
```

Problem Statement

You are tasked with implementing a ticketing system for a concert event using a queue using a linked list data structure.

The program should store ticket numbers in a queue and display them in a zigzag pattern. The program should prompt the user to enter the capacity of the ticket queue and allow them to enqueue ticket numbers until they enter -1 to stop. After enqueueing, the program should display the ticket numbers in the zigzag pattern.

Input format :

The first line of input consists of a single integer representing the capacity of the ticket queue.
The next line contains a sequence of integers representing the ticket numbers to enqueue.
The sequence should end with -1 to stop enqueueing.

Output format :

The output prints the ticket numbers stored in the queue, displayed in a zigzag pattern.

Code constraints :

The ticket numbers must be positive integers.
The capacity of the queue should be a positive integer.

Sample test cases :

Input 1 :

```
5
10 20 30 40 50 -1
```

Output 1 :

```
10 50 20 40 30
```

Input 2 :

```
8
11 22 33 44 55 66 77 88 -1
```

Output 2 :

```
11 88 22 77 33 66 44 55
```

```
#include <iostream>


// Node structure for the linked list

struct Node {

   int data;

   Node* next;

};


// Queue class

class Queue {

private:

   Node* front; // Pointer to the front of the queue

   Node* rear;  // Pointer to the rear of the queue
```

```
    int* arr;    // Array to store elements in a zigzag pattern
    int capacity; // Capacity of the array
    int size;    // Current size of the queue

public:
    Queue(int capacity) {
        this->capacity = capacity;
        this->arr = new int[capacity];
        this->front = nullptr;
        this->rear = nullptr;
        this->size = 0;
    }

    ~Queue() {
        delete[] arr;
    }

    void enqueue(int data) {
        Node* newNode = new Node;
        newNode->data = data;
        newNode->next = nullptr;

        if (rear == nullptr) {
            front = rear = newNode;
        } else {
            rear->next = newNode;
            rear = newNode;
        }

        arr[size] = data;
        size++;
```

```
}

int dequeue() {
    if (front == nullptr) {
        return -1; // Queue is empty
    }

    Node* temp = front;
    int data = temp->data;

    front = front->next;
    if (front == nullptr) {
        rear = nullptr;
    }

    delete temp;
    return data;
}

void displayZigzagPattern() {
    bool forward = true; // Flag to determine the direction of traversal
    int start = 0; // Starting index for traversal
    int end = size - 1; // Ending index for traversal

    while (start <= end) {
        if (forward) {
            std::cout << arr[start] << " ";
            start++;
        } else {
            std::cout << arr[end] << " ";
            end--;
```

```
            }
            forward = !forward; // Toggle the direction
        }

        std::cout << std::endl;
    }
};

int main() {
    int capacity;
    std::cin >> capacity;

    Queue queue(capacity);

    int element;
    std::cin >> element;
    while (element != -1) {
        queue.enqueue(element);
        std::cin >> element;
    }

    queue.displayZigzagPattern();

    return 0;
}
```

Problem Statement

You are developing a ticket management system for a popular theater. The system uses a Queue data structure to handle the ticketing process efficiently. Your task is to implement a program that simulates the ticket management system using an array-based queue.

The program should provide a user-friendly interface for theater staff to perform queue operations.

1. They should be able to insert a ticket into the queue, delete a ticket from the queue, and display the tickets in the queue.
2. The queue has a maximum capacity of 5 tickets. If the queue is full and an insertion is attempted, the program should display a message saying "Queue is full."
3. If the queue is empty and a deletion is attempted, the program should display a message saying "Queue is empty."

The program should handle these conditions appropriately and ensure that the queue operations are performed correctly.

Input format :

The input consists of an integer option representing the action to be performed:
Option 1: Insert an element into the queue. The next line contains an integer representing the element to be inserted.
Option 2: Delete an element from the queue.
Option 3: Display the elements in the queue.

Output format :

For each operation, the program should provide the appropriate output messages:
If option 1 is chosen, display a message indicating that the element is inserted in the queue.
If the maximum capacity of the queue is reached, and insertion is attempted, print "Queue is full."
If option 2 is chosen, display the deleted number.
If option 2 is chosen and if the queue is empty, and no elements can be deleted, print "Queue is empty."
If option 3 is chosen, display the elements in the queue.
If any other option other than 1, 2, or 3 is given, print "Invalid option."

Refer to the sample output for the exact text and format.

Code constraints :

The maximum size of the queue is defined as max = 5.
The queue can store integer values.

Sample test cases :

Input 1 :

```
1
10
3
5
```

Output 1 :

```
10 is inserted in the queue.
Elements in the queue are: 10
Invalid option.
```

Input 2 :

```
1
30
1
40
2
3
```

Output 2 :

```
30 is inserted in the queue.
40 is inserted in the queue.
Deleted number is: 30
Elements in the queue are: 40
```

Input 3 :

```
1
97
2
3
```

Output 3 :

```
97 is inserted in the queue.
```

```
Deleted number is: 97
Queue is empty.
```

Input 4 :

```
1
12
1
23
1
34
1
45
1
56
1
32
```

Output 4 :

```
12 is inserted in the queue.
23 is inserted in the queue.
34 is inserted in the queue.
45 is inserted in the queue.
56 is inserted in the queue.
Queue is full.
```

```
#include <iostream>

#define max 5


int insertq(int queue[max], int& rear, int data)

{

   if (rear == max - 1)

   return 0; // Return 0 to indicate queue is full

   else

   {

      rear++;

      queue[rear] = data;

      return 1;

   }

}


int delq(int queue[max], int& front, int& rear, int& data)

{
```

```
    if (front == rear)
    {
        data = -1; // Set data to a sentinel value
        return 0; // Return 0 to indicate an empty queue
    }
    else
    {
        front++;
        data = queue[front];
        if (front > rear)
        {
            front = -1;
            rear = -1;
        }
        return 1; // Return 1 to indicate successful deletion
    }
}

void display(int queue[max], int front, int rear)
{
    if (front == rear)
    {
        std::cout << "Queue is empty." << std::endl;
    }
    else
    {
        std::cout << "Elements in the queue are: ";
        for (int i = front + 1; i <= rear; i++)
        {
            std::cout << queue[i] << " ";
        }
```

```
        std::cout << std::endl;
    }
}

int main()
{
int queue[max], data;
int front, rear, reply, option;
front = rear = -1;

while (1)
{
    if (!(std::cin >> option))
        break;

    switch (option)
    {
        case 1:
            if (!(std::cin >> data))
                break;

            reply = insertq(queue, rear, data);
            if (reply == 0)
                std::cout << "Queue is full." << std::endl;
            else
                std::cout << data << " is inserted in the queue." << std::endl;
            break;

        case 2:
            reply = delq(queue, front, rear, data);
            if (reply == 0)
```

```
        {
            std::cout << "Queue is empty." << std::endl;
        }
        else
        {
            std::cout << "Deleted number is: " << data << std::endl;
        }
        break;

    case 3:
        display(queue, front, rear);
        break;

    default:
        std::cout << "Invalid option." << std::endl;
        break;
  }
}

return 0;
}
```

Problem Statement

You are tasked with developing a simple inventory management system for a bookstore. The system should allow the bookstore staff to manage the inventory of books, prioritize restocking, and efficiently handle restocking operations.

The program uses a priority queue to manage the inventory. Each book in the inventory is represented by its title, the current quantity available, and a restock priority.

The book is restocked using restock priority. The restock priority is a value between 1 and 5, a low value indicating a higher priority. In the order of priority, 1 has high priority, the level gets reduced as the priority value increases, and 5 has low priority.

Include the following options:
1 - Add book to inventory
2 - Restock book

3 - View the next book to restock
4 - Exit

Input format :

The input consists of integers representing the choice from the menu.

For choice 1 (Add book to inventory), the following lines of input consists of:

1. Book title (string with spaces allowed)
2. Current quantity (integer)
3. Restock priority (integer between 1 and 5, inclusive)

For choices 2 (Restock book) and 3 (View next book to restock), no additional input is required.

For choice 4 (Exit), no additional input is required.

Output format :

For choice 1, the program prints: "Book added to the inventory."
For choice 2, the program prints: "Restocked book: [Book Title]"
For choice 3, the program prints after restocking the book: "Next book to restock: [Book Title]"
For choice 4, the program prints: "Exiting the application."

If the choices are not 1, 2, 3, or 4:

For invalid choices, the program prints: "Invalid choice. Please enter a valid option."

For choice 3:

When there are no books in the inventory, the program prints: "No books in the inventory."

Code constraints :

1 <= restock priority <= 5
The book title can include spaces and should be non-empty.
The choice entered by the user should be an integer corresponding to the available menu options.

Sample test cases :

Input 1 :

```
1
Wings of Fire
5
4
1
Harry Potter and the Sorcerer's Stone
7
2
2
3
4
```

Output 1 :

```
Book added to the inventory.
Book added to the inventory.
Restocked book: Harry Potter and the Sorcerer's Stone
Next book to restock: Wings of Fire
Exiting the application.
```

Input 2 :

```
1
The Great Mocking Bird
3
4
2
3
4
```

Output 2 :

```
Book added to the inventory.
```

```
Restocked book: The Great Mocking Bird
No books in the inventory.
Exiting the application.
```

Input 3 :

```
6
4
```

Output 3 :

```
Invalid choice. Please enter a valid option.
Exiting the application.
```

Input 4 :

```
2
4
```

Output 4 :

```
No books in the inventory.
Exiting the application.
```

```
#include <iostream>

#include <string>


const int MAX_SIZE = 100;


struct Book {

   std::string title;

   int quantity;

   int restockPriority;

};


struct PriorityQueue {

   Book arr[MAX_SIZE];

   int size;


   PriorityQueue() : size(0) {}


   bool isEmpty() {

      return size == 0;

   }
```

```
    void push(const Book& book) {
        if (size == MAX_SIZE) {
            std::cout << "Queue is full. Cannot add more books.\n";
            return;
        }

        int index = size;
        arr[size++] = book;

        while (index > 0) {
            int parent = (index - 1) / 2;
            if (arr[index].restockPriority < arr[parent].restockPriority) { // Lower values indicate higher priority
                std::swap(arr[index], arr[parent]);
                index = parent;
            } else {
                break;
            }
        }
    }

    void pop() {
        if (isEmpty()) {
            std::cout << "No books in the inventory.\n";
            return;
        }

        arr[0] = arr[--size];

        int index = 0;
        while (true) {
```

```
        int leftChild = 2 * index + 1;
        int rightChild = 2 * index + 2;
        int smallest = index;

        if (leftChild < size && arr[leftChild].restockPriority < arr[smallest].restockPriority) { // Lower values indicate higher priority
            smallest = leftChild;
        }

        if (rightChild < size && arr[rightChild].restockPriority < arr[smallest].restockPriority) { // Lower values indicate higher priority
            smallest = rightChild;
        }

        if (smallest != index) {
            std::swap(arr[index], arr[smallest]);
            index = smallest;
        } else {
            break;
        }
    }
  }

  Book top() {
    if (isEmpty()) {
        std::cout << "No books in the inventory.\n";
        return {"", 0, 0};
    }
    return arr[0];
  }
};
```

```
int main() {
    PriorityQueue inventory;

    int choice;
    do {

        std::cin >> choice;
        std::cin.ignore(); // Clear the newline character from the previous input

        switch (choice) {
            case 1: {
                std::string title;
                int quantity, priority;
                //std::cout << "Enter book title: ";
                std::getline(std::cin, title);
                //std::cout << "Enter quantity: ";
                std::cin >> quantity;
                //std::cout << "Enter restock priority (1-5): ";
                std::cin >> priority;
                if (priority < 1) priority = 1;
                if (priority > 5) priority = 5;
                inventory.push({title, quantity, priority});
                std::cout << "Book added to the inventory.\n";
                break;
            }
            case 2:
                if (!inventory.isEmpty()) {
                    std::cout << "Restocked book: " << inventory.top().title << "\n";
                    inventory.pop();
                } else {
                    std::cout << "No books in the inventory.\n";
```

```
            }
            break;
        case 3:
            if (!inventory.isEmpty()) {
                std::cout << "Next book to restock: " << inventory.top().title << "\n";
            } else {
                std::cout << "No books in the inventory.\n";
            }
            break;
        case 4:
            std::cout << "Exiting the application.";
            break;
        default:
            std::cout << "Invalid choice. Please enter a valid option.\n";
            break;
    }
  } while (choice != 4);


  return 0;
}
```

Problem Statement

You are tasked with implementing a program to calculate the average of elements in a queue. The queue represents a series of values, and you need to find the average of all the elements in the queue.

However, there are a few scenarios to consider:

1. If the queue is empty, you should display the message "Queue is empty."
2. If the queue is not empty, you need to calculate the average of its elements and display the result.

Input format :
The first line of input consists of an integer **N**, representing the number of elements in the queue.
The second line consists of **N** space-separated integers representing the elements to be enqueued.
Output format :
If the queue is empty, print "Queue is empty." on a separate line.
If the queue is not empty, output the average of its elements, rounded off to two decimal places.
Code constraints :

N > 0
The maximum number of elements in the queue is defined as MAX_SIZE = 100.

Sample test cases :

Input 1 :

```
4
55 60 65 70
```

Output 1 :

```
62.50
```

Input 2 :

```
0
```

Output 2 :

```
Queue is empty.
```

Input 3 :

```
7
12 25 68 97 41 29 63
```

Output 3 :

```
47.86
```

```
#include <iostream>

#include <iomanip>


using namespace std;


const int MAX_SIZE = 100;


class Queue {

private:

   int arr[MAX_SIZE];

   int frontIdx;

   int rearIdx;


public:

   Queue() {

      frontIdx = -1;

      rearIdx = -1;

   }


   bool isEmpty() {
```

```
    return (frontIdx == -1 && rearIdx == -1);
}

bool isFull() {
    return (rearIdx == MAX_SIZE - 1);
}

void enqueue(int element) {
    if (isFull()) {
        return;
    }

    if (isEmpty()) {
        frontIdx = 0;
    }

    rearIdx++;
    arr[rearIdx] = element;
}

void dequeue() {
    if (isEmpty()) {
        return;
    }

    if (frontIdx == rearIdx) {
        frontIdx = -1;
        rearIdx = -1;
    } else {
        frontIdx++;
    }
```

```
    }

    int front() {
        if (isEmpty()) {
            return -1;
        }

        return arr[frontIdx];
    }
};

double findQueueAverage(Queue& q) {
    double sum = 0;
    int count = 0;

    while (!q.isEmpty()) {
        sum += q.front();
        q.dequeue();
        count++;
    }

    if (count != 0) {
        return sum / count;
    } else {
        return 0;
    }
}

int main() {
    Queue q;
    int n;
```

```
    cin >> n;

    if (n == 0) {
        cout << "Queue is empty." << endl;
        return 0;
    }

    for (int i = 0; i < n; i++) {
        int element;
        cin >> element;
        q.enqueue(element);
    }

    double average = findQueueAverage(q);

    // Set the output to two decimal places
    cout << fixed << setprecision(2) << average << endl;

    return 0;
}
```

Problem Statement

You are assigned the task of developing a package delivery system for a logistics company. The system should enable efficient management of packages for delivery, including prioritizing deliveries based on certain criteria.

The program utilizes a priority queue to manage packages for delivery. Each package is represented by its destination and delivery priority. The delivery priority is a value between 1 and 5, with lower values indicating higher priority for delivery. (1 has the highest priority, next 2, and so on)

The system offers the following options Add package, Deliver package, View next package for delivery, and Exit.

Input format :

The input consists of an integer representing the choice from the menu.

Choice 1: Add package

Choice 2: Deliver the package
Choice 3: View the next package for delivery
Choice 4: Exit

For Choice 1, the program expects the following inputs on separate lines:

1. Package destination (string with spaces allowed)
2. Delivery priority (integer between 1 and 5, inclusive)

For choice 2, choice 3, and for choice 4, no additional input is required.

Output format :

The program outputs messages based on the user's choices.
When adding a package to the delivery queue, the program outputs: "Package added to the delivery queue."
When delivering a package, the program outputs: "Delivered package to: [Package Destination]"
A lower delivery priority value indicates a higher priority.
When viewing the next package for delivery, the program outputs: "Next package for delivery: [Package Destination]"
When exiting the application, the program outputs: "Exiting the application."
For invalid choices, the program outputs: "Invalid choice."
When there are no packages in the delivery queue, the program outputs: "No packages in the delivery queue."

Refer to the sample output for the exact text and format.

Code constraints :

Queue size should be 5.

Sample test cases :

Input 1 :

```
1
Paris
3
1
San Francisco
2
2
4
```

Output 1 :

```
Package added to the delivery queue.
Package added to the delivery queue.
Delivered package to: San Francisco
Exiting the application.
```

Input 2 :

```
1
New York
2
2
4
```

Output 2 :

```
Package added to the delivery queue.
Delivered package to: New York
Exiting the application.
```

Input 3 :

```
1
London
4
3
4
```

Output 3 :

```
Package added to the delivery queue.
```

```
Next package for delivery: London
Exiting the application.
```

Input 4 :

```
5
4
```

Output 4 :

```
Invalid choice.
Exiting the application.
```

Input 5 :

```
2
4
```

Output 5 :

```
No packages in the delivery queue.
Exiting the application.
```

```
#include <iostream>

#include <string>

#include <queue>


struct Package {

    std::string destination;

    int priority;


    Package(const std::string& dest, int prio) : destination(dest), priority(prio) {}


    bool operator<(const Package& other) const {

        return priority > other.priority;

    }

};


int main() {

    std::priority_queue<Package> deliveryQueue;


    int choice;

    do {

        std::cin >> choice;
```

```
switch (choice) {
    case 1: {
        std::string destination;
        int priority;
        std::cin.ignore();
        std::getline(std::cin, destination);
        std::cin >> priority;
        if (priority < 1) priority = 1;
        if (priority > 5) priority = 5;
        deliveryQueue.push(Package(destination, priority));
        std::cout << "Package added to the delivery queue.\n";
        break;
    }
    case 2:
        if (!deliveryQueue.empty()) {
            std::cout << "Delivered package to: " << deliveryQueue.top().destination << "\n";
            deliveryQueue.pop();
        } else {
            std::cout << "No packages in the delivery queue.\n";
        }
        break;
    case 3:
        if (!deliveryQueue.empty()) {
            std::cout << "Next package for delivery: " << deliveryQueue.top().destination << "\n";
        } else {
            std::cout << "No packages in the delivery queue.\n";
        }
        break;
    case 4:
        std::cout << "Exiting the application.\n";
```

```
            break;
        default:
            std::cout << "Invalid choice.\n";
            break;
    }
  } while (choice != 4);

  return 0;
}
```

Problem Statement

A computing system executes multiple tasks simultaneously. Your task is to implement a priority queue that prioritizes these tasks based on their importance and urgency.

The priority queue should prioritize tasks based on the following criteria:

1. Tasks with higher priority should be executed first.
2. In the case of tasks with equal priority, tasks with earlier deadlines should be executed first.

Your goal is to implement a custom priority queue using an array-based data structure.

The priority queue should support the following operations:

1. push(task): Inserts a new task into the priority queue according to the priority and deadline criteria.
2. pop(): Removes and returns the highest-priority task from the priority queue.
3. top(): Returns the highest-priority task from the priority queue without removing it.
4. empty(): Returns true if the priority queue is empty; otherwise, returns false.

Note: A higher integer specifies high priority.
Input format :
The first line of the input consists of an integer **N**, the number of tasks to be executed by the computing system.
The following **N** lines of the input contain task details in the following format:
priority deadline taskName
where the **priority** is an integer representing the priority of the task, the **deadline** is an integer representing the deadline of the task, and **taskName** is a string representing the name of the task.
Output format :
The output prints a list of tasks executed by the computing system, in the order they were executed.

Refer to the sample output for the exact text and format.
Code constraints :
1 <= N <= 10^5
1 <= priority <= 10^9
1 <= deadline <= 10^9
Sample test cases :
Input 1 :

```
4
5 1 Task1
1 2 Task2
4 2 Task3
6 1 Task4
```

Output 1 :

```
Executed Tasks:
Task4
Task1
Task3
Task2
```

Input 2 :

```
4
100 4 Task1
100 2 Task2
100 3 Task3
100 1 Task4
```

Output 2 :

```
Executed Tasks:
Task4
Task2
Task3
Task1
```

```
#include <iostream>

#include <string>

#include <algorithm>


const int MAX_TASKS = 100005;


struct Task {

int priority;

int deadline;

std::string name;

};


// Custom comparison function for sorting tasks

bool compareTasks(const Task& a, const Task& b) {

if (a.deadline != b.deadline) {

return a.deadline > b.deadline;

}
```

```
return a.priority < b.priority;
}

// Custom priority queue implementation using an array
struct PriorityQueue {
Task tasks[MAX_TASKS];
int size;
PriorityQueue() : size(0) {}

void push(const Task& task) {
   tasks[size++] = task;
   std::push_heap(tasks, tasks + size, compareTasks);
}

void pop() {
   std::pop_heap(tasks, tasks + size, compareTasks);
   --size;
}

Task top() const {
   return tasks[0];
}

bool empty() const {
   return size == 0;
}
};

int main() {
PriorityQueue pq;
int n;
```

```
std::cin >> n;


int priority, deadline;
std::string name;
for (int i = 0; i < n; i++) {
   std::cin >> priority >> deadline >> name;
   Task task;
   task.priority = priority;
   task.deadline = deadline;
   task.name = name;
   pq.push(task);
}


std::cout << "Executed Tasks:" << std::endl;
while (!pq.empty()) {
   Task task = pq.top();
   pq.pop();
   std::cout << task.name << std::endl;
}


return 0;
}
```

PRIORITY QUEUE:-

Problem Statement

You are given a Queue of **N** integers. Write a program to implement the **dequeue** operation using the linked list.

Input format :

The first line of input consists of an integer **N**, denoting the size of the queue.

The second line consists of **N** space-separated integers, denoting the elements of the queue.

Output format :

The output prints the queue after performing the dequeue operation.

If an underflow occurs, print "Underflow", and print "Queue is empty".

Code constraints :

Queue MAXIMUM SIZE is predefined (MAX = 25).

Sample test cases :

Input 1 :

```
5
1 2 3 4 5
```

Output 1 :

```
2 3 4 5
```

Input 2 :

```
0
0
```

Output 2 :

```
Underflow
Queue is empty
```

```
#include <iostream>

#define MAX 25


int queue[MAX];

int rear = -1;

int front = -1;


void Enqueue(int data) {

   if (rear == MAX - 1)

      std::cout << "Overflow" << std::endl;

   else {

      if (front == -1)

         front = 0;

      rear = rear + 1;

      queue[rear] = data;

   }

}


void Dequeue() {

   if (front == -1 || front > rear) {

      std::cout << "Underflow" << std::endl;

      return;

   } else {
```

```
        front = front + 1;
    }
}

void display() {
    if (front == -1)
        std::cout << "Queue is empty" << std::endl;
    else {
        for (int i = front; i <= rear; i++)
            std::cout << queue[i] << " ";
        std::cout << std::endl;
    }
}

int main() {
    int n, i, e;
    std::cin >> n;
    for (i = 0; i < n; i++) {
        std::cin >> e;
        Enqueue(e);
    }
    Dequeue();
    display();

    return 0;
}
```

Problem Statement

You are working on a text processing system for a search engine. As part of the system, you need to implement a pattern-matching algorithm using a sliding window approach.

Given a large text document and a pattern, you are required to find all occurrences of the pattern in the text document efficiently.

Your task is to write a program that performs the following operations:

1. Read the large text document and store it in memory.
2. Read the pattern from the user.
3. Implement a sliding window algorithm using a **linked list-based deque** to find all occurrences of the pattern in the text document.
4. Display the positions (starting indices) of all occurrences of the pattern in the text document.

Write a program to implement the above operations and display the positions of all occurrences of the pattern in the text document.

Your program should prompt the user to enter the required inputs and then output the positions of all occurrences of the pattern.

Note: This is a sample question asked in TCS recruitment.
Input format :
The first line of input consists of a sequence of strings.
The second line consists of the pattern to be found in the given string.
Output format :
If the pattern is found in the string, print the index or indexes where the pattern occurs. (index starts from 0)
If the pattern is not found, print "Pattern not found".
If the pattern is longer than the string, print "Pattern is longer than the string".

Refer to the sample output for the exact text and format.
Code constraints :
The strings are case-sensitive.
Sample test cases :
Input 1 :

```
abcdabcdeabcdabcdeabcdabcde
abcd
```

Output 1 :

```
Pattern found at index 0
Pattern found at index 4
Pattern found at index 9
Pattern found at index 13
Pattern found at index 18
Pattern found at index 22
```

Input 2 :

```
abcdefghij
abcdefghijkl
```

Output 2 :

```
Pattern is longer than the string
```

Input 3 :

```
Harry Potter
harry
```

Output 3 :

```
Pattern not found
```

```
#include <iostream>

#include <string>

#include <deque>
```

```
using namespace std;

struct Node {
    char data;
    Node* next;
};

void findPattern(string str, string pattern) {
    Node* head = nullptr;
    Node* tail = nullptr;
    int n = str.size(), m = pattern.size();
    bool found = false;

    // Check if the pattern is longer than the string
    if (m > n) {
        cout << "Pattern is longer than the string" << endl;
        return;
    }

    // Initialize the deque with the first m characters of the string
    deque<char> dq(str.begin(), str.begin() + m);

    // Initialize the linked list with the first m characters of the string
    for (int i = 0; i < m; i++) {
        Node* newNode = new Node();
        newNode->data = dq[i];
        newNode->next = nullptr;
        if (head == nullptr) {
            head = newNode;
            tail = newNode;
```

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

// Check if the linked list matches the pattern and print the first match
Node* current = head;
string temp = "";
while (current != nullptr) {
    temp += current->data;
    current = current->next;
}
if (temp == pattern) {
    cout << "Pattern found at index 0" << endl;
    found = true;
}

// Slide the window through the string and check for matches
for (int i = m; i < n; i++) {
    dq.pop_front();
    dq.push_back(str[i]);
    Node* newNode = new Node();
    newNode->data = str[i];
    newNode->next = nullptr;
    tail->next = newNode;
    tail = newNode;
    head = head->next;
    temp = "";
    current = head;
```

```
        while (current != nullptr) {
            temp += current->data;
            current = current->next;
        }
        if (temp == pattern) {
            cout << "Pattern found at index " << i - m + 1 << endl;
            found = true;
        }
    }

    // If pattern is not found, print a message
    if (!found) {
        cout << "Pattern not found" << endl;
    }
}

int main() {
    string str, pattern;
    getline(cin, str);
    getline(cin, pattern);
    findPattern(str, pattern);
    return 0;
}
```

Problem Statement

Your task is to implement a **double-ended queue** data structure. A deque is a linear data structure that allows elements to be inserted or removed from both the front and the rear.

Your program should provide the following functionality:

1. Initialize an empty deque.
2. Push an element to the back of the deque.
3. Check if the deque is empty.
4. Find and print the maximum element in the deque.

Input format :

The input consists of the elements that should be inserted into the deque.
The input is terminated by entering -1.

Output format :
The output prints the maximum element in the given deque.
Sample test cases :
Input 1 :

```
1
2
3
4
-1
```

Output 1 :

```
4
```

Input 2 :

```
41
36
-1
```

Output 2 :

```
41
```

```
#include <iostream>

#include <stdlib.h>


struct Node {

   int data;

   Node* next;

};


struct Deque {

   Node* front;

   Node* rear;

};


void initDeque(Deque* deque) {

   deque->front = NULL;

   deque->rear = NULL;

}


bool isEmpty(Deque* deque) {

   return deque->front == NULL;
```

```
}

void pushBack(Deque* deque, int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    newNode->next = NULL;
    if (isEmpty(deque)) {
        deque->front = newNode;
        deque->rear = newNode;
    } else {
        deque->rear->next = newNode;
        deque->rear = newNode;
    }
}

int popFront(Deque* deque) {
    if (isEmpty(deque)) {
        std::cout << "The deque is empty." << std::endl;
        exit(1);
    }
    int data = deque->front->data;
    Node* temp = deque->front;
    deque->front = deque->front->next;
    free(temp);
    if (deque->front == NULL) {
        deque->rear = NULL;
    }
    return data;
}

int main() {
```

```
    Deque myDeque;
    initDeque(&myDeque);
    int num;
    while (1) {
        std::cin >> num;
        if (num == -1)
            break;
        pushBack(&myDeque, num);
    }
    if (isEmpty(&myDeque)) {
        std::cout << "The deque is empty." << std::endl;
    } else {
        int maxElement = myDeque.front->data;
        Node* current = myDeque.front->next;
        while (current != NULL) {
            if (current->data > maxElement) {
                maxElement = current->data;
            }
            current = current->next;
        }
        std::cout<<maxElement;
    }
    return 0;
}
```

Problem Statement

You are working on a program that manages a **deque** implemented using an array. The deque initially contains a set of integer elements.

Write a program to implement the reversal operation on the deque. Your program should prompt the user to enter the elements of the deque and then output the reversed elements.

Note: This kind of question will be helpful in clearing Infosys recruitment.
Input format :
The first line of input consists of the number of elements **N** in the deque.

The second line consists of the **N** deque elements, separated by space.

Output format :

The output displays the reversed deque elements, separated by space.

Code constraints :

N > 0

Sample test cases :

Input 1 :

```
5
12 59 -91 62 48
```

Output 1 :

```
48 62 -91 59 12
```

Input 2 :

```
10
7 8 5 3 6 2 5 9 4 1
```

Output 2 :

```
1 4 9 5 2 6 3 5 8 7
```

```
#include <iostream>

using namespace std;


void reverseDeque(int* dq, int n) {

   int left = 0, right = n - 1;

   while (left < right) {

      int temp = dq[left];

      dq[left] = dq[right];

      dq[right] = temp;

      left++;

      right--;

   }

}


int main() {

   int n;

   cin >> n;


   int* myDeque = new int[n];

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

```
        cin >> myDeque[i];
    }

    reverseDeque(myDeque, n);

    for (int i = 0; i < n; i++) {
        cout << myDeque[i] << " ";
    }
    // cout << endl;

    delete[] myDeque;

    return 0;
}
```

Problem Statement

You have been assigned to develop a waitlist management system for a popular restaurant. The restaurant experiences high customer demand and often has a waitlist for customers to secure a table. Your task is to develop a program that simulates this waitlist using a double-ended queue.

The program should provide the following functionalities:

Add Customers: The program should allow the host/hostess to add customers to the waitlist. Each customer is identified by their name. The host/hostess will enter the customer's name, and the program should add the customer to the end of the waitlist.

Remove Customers: Once a table becomes available, the host/hostess can remove customers from the waitlist. The program should display the customer at the front of the waitlist and remove them from the list. The host/hostess can then seat the customer at the available table.

Move Customers: If a customer at the front of the waitlist is not ready to be seated yet, the host/hostess can move them to the end of the waitlist. This is typically done if the customer requests more time or needs to wait for additional members of their party to arrive. The program should move the customer from the front to the back of the waitlist.

Display Waitlist: At any time, the host/hostess can view the current waitlist. The program should display the names and party sizes of all customers on the waitlist, starting from the front to the back.

Your solution should utilize a **double-ended queue** to implement the waitlist functionality. The program should prompt the user for the required actions and perform the corresponding operations on the waitlist. It should also display appropriate messages to notify the host/hostess about the status of the waitlist and any changes made.

Note: This is a sample question asked in a HCL interview.
Input format :
The input consists of choices.
If the choice is 1, enter the customer's name.
If the option is 2, the customer is removed from the waitlist.
If the choice is 3, the customer is moved to the back of the waitlist.
If the choice is 4, the current waitlist is displayed.
If the choice is 5, exit.

Refer to the sample input for a better understanding.
Output format :
The output prints the results based on the given choices.
If the choice is 1, display a message indicating that the customer has been added to the waitlist.
If the choice is 2, display a message indicating that the customer has been removed from the waitlist.
If the choice is 3, display a message indicating that the customer has been moved to the back of the waitlist. If there are no customers, print "Error: Waitlist is empty".
If the choice is 4, display the current waitlist. if there are no customers, print "Empty".
If the choice is 5, display the exit message.
If the choice is greater than 5, print "Invalid option!".

Refer to the sample output for the exact text and format.
Sample test cases :
Input 1 :

```
1
Alice
1
Bob
3
2
4
5
```

Output 1 :

```
Alice has been added to the waitlist.
Bob has been added to the waitlist.
Alice has been moved to the back of the waitlist.
Bob has been removed from the waitlist.
Current waitlist:
Alice
Exiting
```

Input 2 :

```
1
John
4
5
```

Output 2 :

```
John has been added to the waitlist.
Current waitlist:
John
Exiting
```

Input 3 :

```
1
Alice
2
3
4
5
```

Output 3 :

```
Alice has been added to the waitlist.
Alice has been removed from the waitlist.
Error: Waitlist is empty
Current waitlist:
Empty
Exiting
```

Input 4 :

```
10
5
```

Output 4 :

```
Invalid option!
Exiting
```

Input 5 :

```
1
Damon
1
Stefan
1
Elena
1
Bonnie
1
Enzo
1
Klaus
3
3
4
5
```

Output 5 :

```
Damon has been added to the waitlist.
Stefan has been added to the waitlist.
Elena has been added to the waitlist.
Bonnie has been added to the waitlist.
Enzo has been added to the waitlist.
Klaus has been added to the waitlist.
Damon has been moved to the back of the waitlist.
Stefan has been moved to the back of the waitlist.
Current waitlist:
Elena
Bonnie
Enzo
Klaus
Damon
Stefan
Exiting
```

```
#include <iostream>

using namespace std;

#define MAX_NAME_LENGTH 20
```

```
typedef struct node {
   char name[MAX_NAME_LENGTH];
   struct node* prev;
   struct node* next;
} Node;

Node* front = NULL;
Node* rear = NULL;

void add_customer(char name[]) {
   Node* new_node = (Node*) malloc(sizeof(Node));
   if (new_node == NULL) {
      cout << "Error: Out of memory\n";
      return;
   }
   for (int i = 0; i < MAX_NAME_LENGTH; i++) {
      new_node->name[i] = name[i];
      if (name[i] == '\0') {
         break;
      }
   }
   new_node->prev = rear;
   new_node->next = NULL;
   if (rear != NULL) {
      rear->next = new_node;
   }
   rear = new_node;
   if (front == NULL) {
      front = new_node;
   }
   cout << new_node->name << " has been added to the waitlist.\n";
```

```
}

void remove_customer() {
    if (front == NULL) {
        cout << "Error: Waitlist is empty\n";
        return;
    }
    Node* temp = front;
    front = front->next;
    if (front == NULL) {
        rear = NULL;
    } else {
        front->prev = NULL;
    }
    cout << temp->name << " has been removed from the waitlist.\n";
    free(temp);
}

void move_customer() {
    if (front == NULL) {
        cout << "Error: Waitlist is empty\n";
        return;
    }
    Node* temp = front;
    front = front->next;
    if (front == NULL) {
        rear = NULL;
    } else {
        front->prev = NULL;
    }
    temp->prev = rear;
```

```
    temp->next = NULL;
    if (rear != NULL) {
        rear->next = temp;
    }
    rear = temp;
    cout << rear->name << " has been moved to the back of the waitlist.\n";
}

void display_waitlist() {
    cout << "Current waitlist:\n";
    if (front == NULL) {
        cout << "Empty\n";
        return;
    }
    Node* current = front;
    while (current != NULL) {
        cout << current->name << endl;
        current = current->next;
    }
}

int main() {
    char name[MAX_NAME_LENGTH];
    int option;

    do {
        scanf("%d", &option);

        switch (option) {
            case 1:
                scanf("%s", name);
```

```
            add_customer(name);
            break;
        case 2:
            remove_customer();
            break;
        case 3:
            move_customer();
            break;
        case 4:
            display_waitlist();
            break;
        case 5:
            cout << "Exiting\n";
            break;
        default:
            cout << "Invalid option!\n";
            break;
      }
   } while (option != 5);

   return 0;
}
```