CSE 4303:DATA STRUCTURE

Topic(V): Queues and Deques

Namisa Najah Raisa **210042112**

December 2023

Islamic University of Technology
Software Engineering'21

Contents

1 Story/Realistic Problem Scenario
[story like statement of some CP problems]

1.1 Problem

There are two arrays each of size n,a and b consisting of the first n positive integers each exactly once, that is, they are permutations.

The task is to find the minimum time required to make both the arrays empty. The following two types of operations can be performed any number of times each taking 1 second:

- In the first operation, you are allowed to rotate the first array clockwise.
- In the second operation, when the first element of both the arrays is the same, they are removed from both the arrays and the process continues.

Input format:

- The first line contains an integer n, denoting the size of the array.
- The second line contains the elements of array a.
- The third line contains the elements of array b.

Output format:
Print the total time taken required to empty both the array.

Constraints:
$1 \leq n \leq 100$

Sample Input:	**Sample Output:**
3 1 3 2 2 3 1	6

2 Basic Understanding of Queues and Deques

2.1 Queues

A queue is a FIFO (First-In,First-Out)data structure in which the element that is inserted first is the first one to be taken out. The elements in a queue are added at one end called the REAR and removed from the other end called the FRONT.Queues can be implemented by using either arrays or linked lists.

2.1.1 Types of Queues

A queue data structure can be classified into the following types:

- Circular Queue
- Deque
- Priority Queue
- Multiple Queue

2.2 Deques

A deque is a list in which the elements can be inserted or deleted at either end.It is also known as a *head-tail linked list* because elements can be added to or removed from either the front(head) or the back(tail) end.But no element can be added and deleted from the middle.In a computer's memory a deque is implemented using either a

- circular array
  or
- circular doubly linked list

The elements in a deque extend from the LEFT end to the RIGHT end and since it is circular,Dequeue[N–1] is followed by Dequeue[0].

3 Operations Supported

3.1 Queues

Some of the basic operations for Queue in Data Structure are:

- **enqueue()** – Insertion of elements to the queue.
- **dequeue()** – Removal of elements from the queue.
- **peek() or front() -** Acquires the data element available at the front node of the queue without deleting it.
- **rear()** – This operation returns the element at the rear end without removing it.
- **isFull()** – Validates if the queue is full.
- **isEmpty()** – Checks if the queue is empty.
- **size() -** This operation returns the size of the queue i.e. the total number of elements it contains.

3.1.1 enqueue():

Inserts an element at the end of the queue i.e. at the rear end.

The following steps should be taken to enqueue (insert) data into a queue:

- Check if the queue is full.
- If the queue is full, return overflow error and exit.
- If the queue is not full, increment the rear pointer to point to the next empty space.
- Add the data element to the queue location, where the rear is pointing.
- return success.

```
    void queueEnqueue(int data)
{
    // Check queue is full or not
    if (capacity == rear) {
        printf("\nQueue is full\n");
        return;
    }

    // Insert element at the rear
    else {
        queue[rear] = data;
        rear++;
    }
    return;
}
```

3.1.2 dequeue():

This operation removes and returns an element that is at the front end of the queue.

The following steps are taken to perform the dequeue operation:

- Check if the queue is empty.
- If the queue is empty, return the underflow error and exit.
- If the queue is not empty, access the data where the front is pointing.
- Increment the front pointer to point to the next available data element.
- The Return success.

```c
    void queueDequeue()
{
    // If queue is empty
    if (front == rear) {
        printf("\nQueue is empty\n");
        return;
    }

    // Shift all the elements from index 2
    // till rear to the left by one
    else {
        for (int i = 0; i < rear - 1; i++) {
            queue[i] = queue[i + 1];
        }

        // decrement rear
        rear--;
    }
    return;
}
```

3.1.3 front():

This operation returns the element at the front end without removing it.

The following steps are taken to perform the front operation:

- If the queue is empty return the most minimum value.
- otherwise, return the front value.

```c
    int front(Queue* queue)
{
    if (isempty(queue))
        return INT_MIN;
    return queue->arr[queue->front];
}
```

3.1.4 rear():

This operation returns the element at the rear end without removing it.

The following steps are taken to perform the rear operation:

- If the queue is empty return the most minimum value.
- otherwise, return the rear value.

```
    int rear(Queue* queue)
{
    if (isEmpty(queue))
        return INT_MIN;
    return queue->arr[queue->rear];
}
```

3.1.5 isEmpty():

This operation returns a boolean value that indicates whether the queue is empty or not.

The following steps are taken to perform the Empty operation:

- check if front value is equal to -1 or not, if yes then return true means queue is empty.
- Otherwise return false, means queue is not empty

```
    // This function will check whether
// the queue is empty or not:
bool isEmpty()
{
    if (front == -1)
        return true;
    else
        return false;
}
```

3.1.6 isFull():

This operation returns a boolean value that indicates whether the queue is full or not.

The following steps are taken to perform the isFull() operation:

- Check if front value is equal to zero and rear is equal to the capacity of queue if yes then return true.
- otherwise return false

```
    // This function will check
// whether the queue is full or not.
bool isFull()
{
    if (front == 0 && rear == MAX_SIZE - 1) {
        return true;
```

```
    }
    return false;
}
```

3.1.7 size():

This operation returns the size of the queue i.e. the total number of elements it contains.

```
    // CPP program to illustrate
// Implementation of size() function
#include <iostream>
#include <queue>
using namespace std;

int main()
{
        int sum = 0;
        queue<int> myqueue;
        myqueue.push(1);
        myqueue.push(8);
        myqueue.push(3);
        myqueue.push(6);
        myqueue.push(2);

        // Queue becomes 1, 8, 3, 6, 2

        cout << myqueue.size();

        return 0;
}
```

3.2 Deques

There are two variants a deque.They include-

- **Input restricted deque:**Here insertions can only be done at one of the ends.While deletions can be done from both ends.
- **Output restricted deque:**Here deletions can be done only at one of the ends.While insertions can be done on both ends.

In a deque in data structure we can perform the following operations:

- Insertion at front
- Insertion at rear

- Deletion at front
- Deletion at rear

Before performing the following operations, we must follow the below steps :

- Take a deque(array) of size n
- Set two pointers variables front = -1 and rear = 0 at the first position.

3.2.1 Insertion at front

1. Initially, we will check the position of the front variable in our array
2. In case, the front variable is less than 1 (front $<$ 1), we will reinitialize the front as the last index of the array (front = n-1).
3. Otherwise, we will decrease the front by 1.
4. Add the new key for example 5 here into our array at the index front - array[front].
5. Every time we insert a new element inside the deque the size increases by 1.

3.2.2 Insertion at rear

1. At first, we need to check whether the deque in data structure is full or not.
2. If the deque data structure is full, we have to reinitialize the rear with 0 (rear = 0).
3. Else increase the rear by 1.
4. Add the new key for example 5 here into our array at the index rear - array[rear].
5. Every time we insert a new element inside the deque the size increases by 1.

3.2.3 Deletion at front

1. At first, we need to check whether the deque in data structure is empty or not.
2. If the deque data structure is empty i.e. front = -1, we cannot perform the deletion process and it will throw an error of underflow condition.

3. If the deque data structure contains only one element i.e. front = rear, set front = -1 and rear = -1.
4. Else if the front is at the last index i.e. front = n - 1, we point the front to the starting index of the deque data structure i.e. front = 0.
5. If none of the cases satisfy we simply increment our front by 1, front = front + 1.
6. Every time we delete any element from the deque data structure the size decreases by 1.

3.2.4 Deletion at rear

1. At first, we need to check whether the deque data structure is empty or not.
2. If the deque data structure is empty i.e. front = -1, we cannot perform the deletion process and it will throw an error of underflow condition.
3. If the deque data structure contains only one element i.e. front = rear, set front = -1 and rear = -1.
4. Else if the rear is at the starting index of the deque i.e. rear = 0, point the rear to the last index of the deque data structure i.e. rear = n-1.
5. If none of the cases satisfy we simply decrement our rear by 1, rear = rear - 1.
6. Every time we delete any element from the deque the size decreases by 1.

Here's an implementation of basic deque operations:

```
    // Deque implementation in C++

#include <iostream>
using namespace std;

#define MAX 10

class Deque {
  int arr[MAX];
  int front;
  int rear;
  int size;

   public:
  Deque(int size) {
    front = -1;
```

```
    rear = 0;
    this->size = size;
  }

  void insertfront(int key);
  void insertrear(int key);
  void deletefront();
  void deleterear();
  bool isFull();
  bool isEmpty();
  int getFront();
  int getRear();
};

bool Deque::isFull() {
  return ((front == 0 && rear == size - 1) ||
      front == rear + 1);
}

bool Deque::isEmpty() {
  return (front == -1);
}

void Deque::insertfront(int key) {
  if (isFull()) {
    cout << "Overflow\n"
       << endl;
    return;
  }

  if (front == -1) {
    front = 0;
    rear = 0;
  }

  else if (front == 0)
    front = size - 1;

  else
    front = front - 1;

  arr[front] = key;
}

void Deque ::insertrear(int key) {
  if (isFull()) {
```

```
      cout << " Overflow\n " << endl;
      return;
    }

    if (front == -1) {
      front = 0;
      rear = 0;
    }

    else if (rear == size - 1)
      rear = 0;

    else
      rear = rear + 1;

    arr[rear] = key;
}

void Deque ::deletefront() {
    if (isEmpty()) {
      cout << "Queue Underflow\n"
         << endl;
      return;
    }

    if (front == rear) {
      front = -1;
      rear = -1;
    } else if (front == size - 1)
      front = 0;

    else
      front = front + 1;
}

void Deque::deleterear() {
    if (isEmpty()) {
      cout << " Underflow\n"
         << endl;
      return;
    }

    if (front == rear) {
      front = -1;
      rear = -1;
    } else if (rear == 0)
```

```
    rear = size - 1;
  else
    rear = rear - 1;
}

int Deque::getFront() {
  if (isEmpty()) {
    cout << " Underflow\n"
       << endl;
    return -1;
  }
  return arr[front];
}

int Deque::getRear() {
  if (isEmpty() || rear < 0) {
    cout << " Underflow\n"
       << endl;
    return -1;
  }
  return arr[rear];
}

int main() {
  Deque dq(4);

  cout << "insert element at rear end \n";
  dq.insertrear(5);
  dq.insertrear(11);

  cout << "rear element: "
     << dq.getRear() << endl;

  dq.deleterear();
  cout << "after deletion of the rear element, the new
      rear element: " << dq.getRear() << endl;

  cout << "insert element at front end \n";

  dq.insertfront(8);

  cout << "front element: " << dq.getFront() << endl;

  dq.deletefront();

  cout << "after deletion of front element new front
```

```
        element: " << dq.getFront() << endl;
}
```

4 The Solution

4.1 The intuition behind the solution

This problem can be easily solved using well known data structure that is queue.Firstly insert all the elements of the first array in the push queue Q.Now,start iterating through the queue from the front,and check if the top element of the queue is similar to the second array or not,if they are same then pop the first element and continue the process until the queue get empty else remove the top element and push it at the back,repeat this process further.

- Perform operation 1 to make a = 3, 2, 1
- Perform operation 1 to make a = 2, 1, 3
- Now perform operation 2 to make a = 1, 3 and b = 3, 1
- Perform operation 1 to make a = 3, 1
- Now perform operation 2 to make a = 1 and b = 1
- Now perform operation 2 to make a = {} and b = {}

```
#include <iostream>
#include <queue>

using namespace std;

int main()
{
    int n, Num;//n is the size of the array,Num stores
        each input integer
    cin >> n;

    queue<int> Qa, Qb;//one for a array and one for b
        array
    for (int i = 0; i < n; i++)
    {
        cin >> Num;
        Qa.push(Num);//reads n Num integers for Qa and
            pushes them into Qa
    }

```

```
    for (int i = 0; i < n; i++)
    {
        cin >> Num;
        Qb.push(Num);//same as Qa
    }

    int TimeNeeded = 0;//the required time
    while (!Qa.empty() && !Qb.empty())//loop continues
        until both queues are empty
    {
        if (Qa.front() == Qb.front())
        {
            Qa.pop();//if both of the front elements
               are equal then they are popped as per
               the operation 2.
            Qb.pop();
            TimeNeeded++;
        }
        else
        {
            int FrontElement_Qa = Qa.front();//stores
               the front element of Qa in a temporary
               variable
            Qa.pop();//pops the first element
            Qa.push(FrontElement_Qa);//and then pushes
                it in the back of Qa.That way it
               rotates clockwise.
            TimeNeeded++;
        }
    }

    cout << TimeNeeded << endl;
    return 0;
}
```

4.2 Complexity Analysis

4.2.1 Time Complexity

- Reading input involves iterating through both arrays a and b,which has a time complexity of O(n) due to two separate loops.
- Enqueuing the elements into the queues (push()): O(n) for each queue (it's 2*O(n) for two queues which is still O(n)).

- Dequeuing and checking elements while both queues are not empty: In the worst case, the while loop will run for a maximum of 2*n times (once per each element in both arrays).
  - Checking front elements (front()): O(1)
  - Removing elements from the queues (pop()): O(1)
  - Pushing elements to the back of queue Qa (push()): O(1)

  So, the total complexity of operations inside the loop is 2n*O(1),which is O(n).

4.2.2 Space Complexity

- **Queue Space:**Two queues Qa and Qb are created to store the elements of arrays a and b respectively.Each queue stores 'n' elements.Therefore,the space complexity for the queues is O(n+n),which is O(n).
- **Additional Space:**Apart from the queues,the solution uses a few integer variables (n,Num,TimeNeeded,i)which are not dependent on the input size and are constant. Hence, they don't significantly contribute to the space complexity.

  Therefore, the overall space complexity of this solution is O(n) due to the queues storing the elements from both arrays.

5 3 Other Problem Links

- **Find next right node of a given key.**
- **Valeriy and Deque**
- **Disk Tower**

6 References

1. Data Structures using C 2nd Edition by Reema Thareja
2. Empty Arrays-The problem
3. Overleaf learning
4. geeksforgeeks
5. Queue uses and operations
6. Deque Operations

7 Conclusion Comments

Basically queues and deques are fundamental data structures used in computer science and programming to manage elements in an ordered manner.They both have distinct characteristics and optimal use cases.

Both structures have their strengths: queues excel in maintaining order and managing processes, while dequeues offer versatility and efficiency in handling elements from both ends. Understanding their characteristics and choosing the right one for a specific task or problem is crucial for efficient algorithm design and implementation.

Queues can be used in-

- Managing requests on a single shared resource such as CPU scheduling and disk scheduling.
- Handling hardware or real-time systems interrupts.
- Handling website traffic.
- Routers and switches in networking.
- Maintaining the playlist in media players.

Deques can be used in-

- Implementing Stack
- Double-ended priority queue
- Undo operations in Editors

Queues and deques are versatile data structures,but there are scenarios where they might not be the optimal choice or might not fit the requirements efficiently.

- **Random Access:** Queues and deques are designed for sequential access and efficient insertion/deletion at specific ends. If frequent random access to elements by index is a primary requirement,arrays might be more suitable.
- **Large-Scale Data Sorting:** Queues are not typically used for large-scale sorting due to inefficiencies in comparison-based sorting algorithms.
- **Real-Time Data Processing:** In scenarios requiring real-time data processing,the overhead involved in managing queues or deques might not be suitable.
- **Complex-Graph Algorithms:** While queues are fundamental for algorithms like BFS more complex graph algorithms might require additional data structures or specialized implementations to optimize memory usage or improve runtime efficiency.

—X—