香港中文大學（深圳）
The Chinese University of Hong Kong

CSC3100 Data Structures
Lecture 10: Queue

Li Jiang
School of Data Science (SDS)
The Chinese University of Hong Kong, Shenzhen

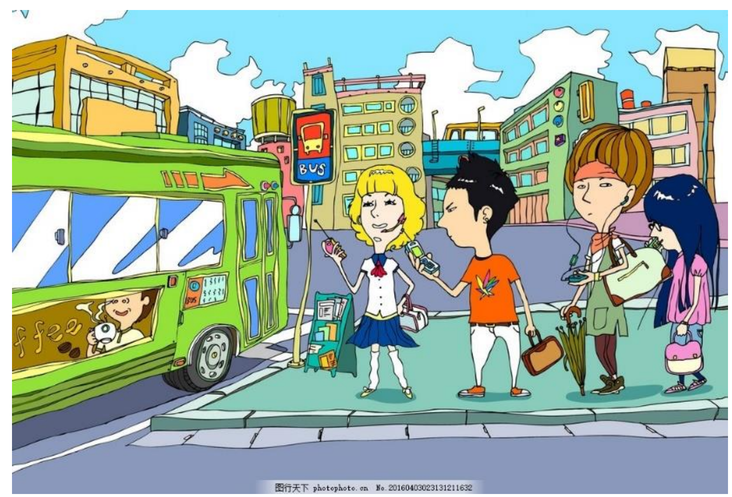
Outline

- Queue
  - Examples and definitions
  - First-In-First-Out (FIFO) property
- Typical type of queues
  - Linear queue and its implementations
  - Circular queue
  - Priority queue
  - Double ended queue

Real-world queues

- A queue of customers waiting to buy a product from a shop, where the customer that came first is served first

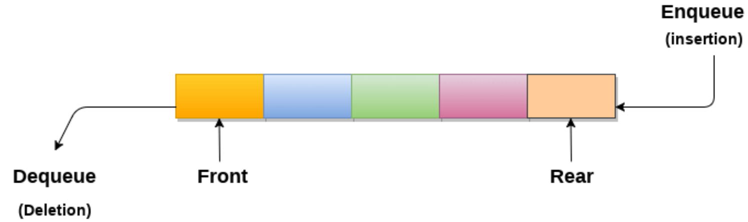
- A queue of students waiting to take a bus at a bus station, where the student that came first gets on first



Queue ADT

- A queue stores a set S of elements that have two constrained updates:
  - Enqueue(e): insert a new element e into S
  - Dequeue( ): remove the least recently inserted element from S, and return it

- Queue follows:
  - First-In-First-Out (FIFO): the first element being enqueued into a queue is the first element dequeued
  - We add from one end, called the rear, and delete from the other end, called the front



Queue applications in CS

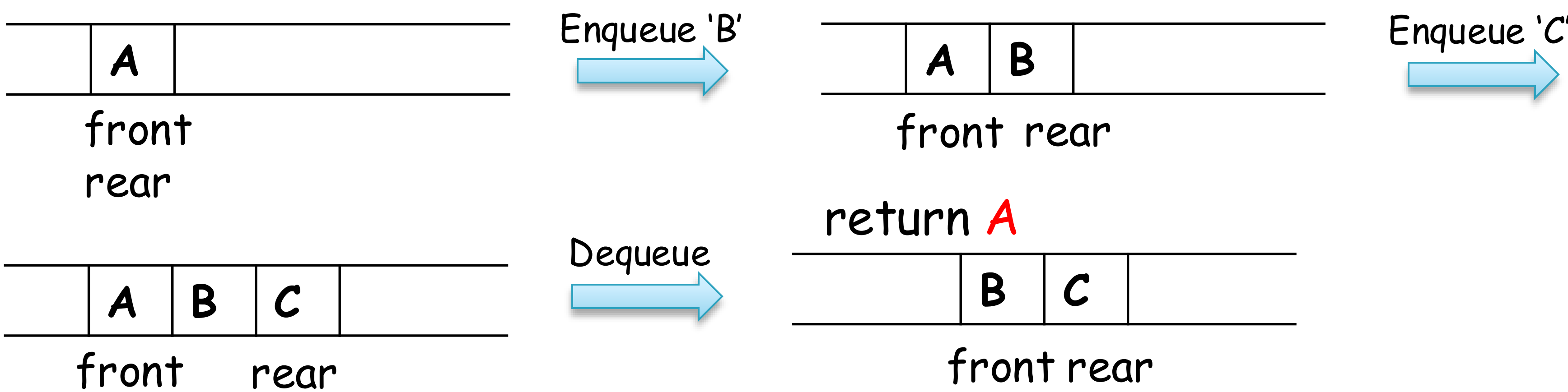
- Used as buffers MP3 player, CD player, etc.
  - Used to maintain the play list in media players in order to add and remove the songs from the play-list

- Used in OS
  - Multiple data transmission tasks
  - Scheduling shared resources like CPU and GPU

- Suppose we have a printer shared by several users, and it can serve a single request at a time
  - When any print request comes, and if the printer is busy, the request will be put into a queue
  - If the requests are available in the queue, the printer takes a request from the queue and serves it

Linear queue

- **A linear queue** is generally referred to as queue, or a linear data structure that follows the FIFO
- Example: a queue Q with only one element 'A'
  - 1. Enqueue 'B' to Q
  - 2. Enqueue 'C' to Q
  - 3. Dequeue from Q



Linear queue implementations

- How to implement queues?
  - Option 1: Linked list
  - Option 2: Arrays

Queue design with linked list

- createQueue: create a linked list L

Algorithm: **createQueue(capacity)**

```
Q <- allocate new memory for queue
Q.L <- allocate new memory for list
return Q
```

- enqueue: add e to the end of the linked list

Algorithm: enqueue(Q,e)

```
insert(Q.L,e)
```

- dequeue: retrieve the first element, i.e., $L.head.next.element$, and delete the first node, i.e., $L.head.next$.

Algorithm: dequeue(Q)

```
if isEmpty(Q.L) error "Queue empty"
frontNode = Q.L.head.next
frontElement = frontNode.element
delete(Q.L, frontNode)
return frontElement
```

Queue design with linked list

- isEmpty: return true if the linked list is empty, otherwise return false

Algorithm: **isEmpty(Q)**

```
return isEmpty(Q.L)
```

- isFull: always return NO

Algorithm: **isFull(Q)**

```
return false
```

Java implementation codes

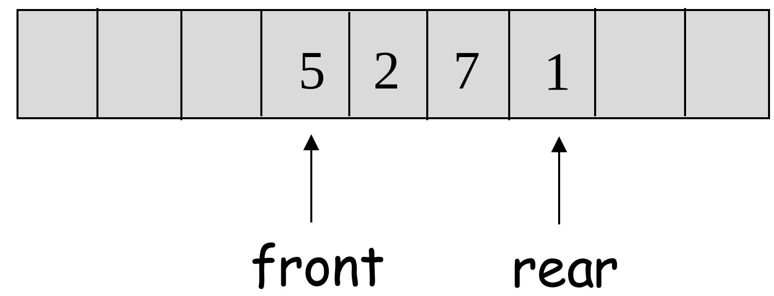
```java
// A Linked List Node
class Node {
    int data;       // integer data
    Node next;      // pointer to the next node
    public Node(int data) {
        // set data in the allocated node and return it
        this.data = data;
        this.next = null;
    }
}
class Queue {
    private static Node rear = null, front = null;
    // Utility function to dequeue the front element
    public static int dequeue() {    // delete at the beginning
        if (front == null) {
            System.out.print("\nQueue Underflow");
            System.exit(1);
        }
        Node temp = front;
        System.out.printf("Removing %d\n", temp.data);
        // advance front to the next node
        front = front.next;
        // if the list becomes empty
        if (front == null) {
            rear = null;
        }
        // deallocate the memory of the removed node and
        // optionally return the removed item
        int item = temp.data;
        return item;
    }
    // Utility function to add an item to the queue
    public static void enqueue(int item) {    // insertion at the end
        // allocate a new node in a heap
        Node node = new Node(item);
        System.out.printf("Inserting %d\n", item);
        // special case: queue was empty
        if (front == null) {
            // initialize both front and rear
            front = node;
            rear = node;
        } else {
            // update rear
            rear.next = node;
            rear = node;
        }
    }
    // Utility function to return the top element in a queue
    public static int peek() {
        // check for an empty queue
        if (front != null) {
            return front.data;
        } else {
            System.exit(1);
        }

        return -1;
    }
    // Utility function to check if the queue is empty or not
    public static boolean isEmpty() {
        return rear == null && front == null;
    }
}

class Main {
    public static void main(String[] args) {
        Queue q = new Queue();
        q.enqueue(1);
        q.enqueue(2);
        q.enqueue(3);
        q.enqueue(4);
        System.out.printf("The front element is %d\n", q.peek());
        q.dequeue();
        q.dequeue();
        q.dequeue();
        q.dequeue();
        if (q.isEmpty()) {
            System.out.print("The queue is empty");
        } else {
            System.out.print("The queue is not empty");
        }
    }
}
```

```
Inserting 1
Inserting 2
Inserting 3
Inserting 4
The front element is 1
Removing 1
Removing 2
Removing 3
Removing 4
The queue is empty
```

Queue design with array

- Array implementation

```
public class QueueArray {
    int capacity;
    int front, rear;
    ElementType[] array;
}
```



Operations:

- To enqueue an element X

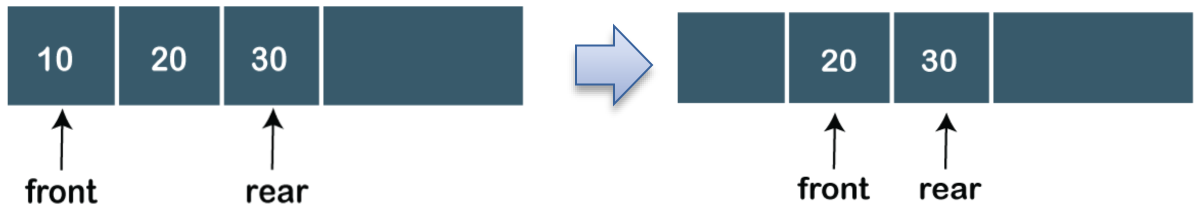
  Increase rear, then set
  array[rear] = X

- To dequeue an element

  Return the value of array[front],
  and then increase front

Queue design with array

- Problem: may run out of rooms
  - If the first several elements are deleted, we cannot insert more elements even though the space is available

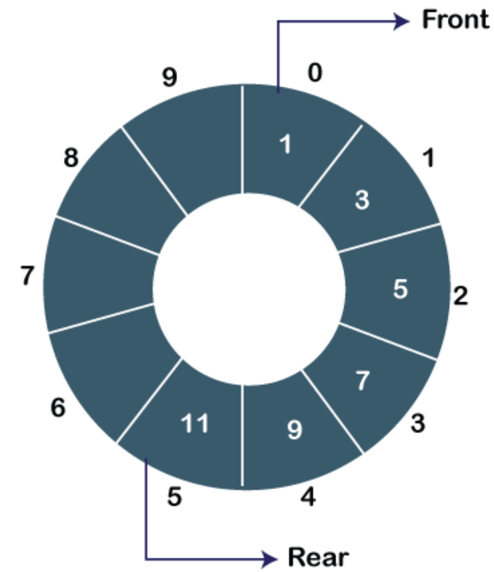


- Two solutions:
  - Keep front always at 0 by shifting the contents up the queue, but this solution is inefficient
  - Use a circular queue (wrap around & use a length variable to keep track of the queue length)

Circular queue

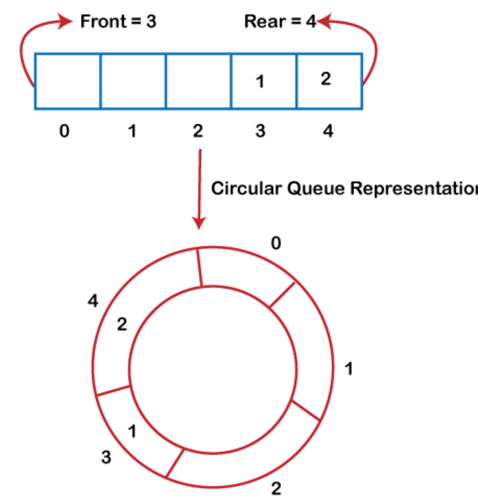
- All the nodes are represented as circular
  - Similar to the linear queue except that the last element is connected to the first one
  - A.k.a. **Ring Buffer** as its ends are connected, forming a continuous ring.

- Drawback of a linear queue is overcome
  - If the empty space is available in a circular queue, the new element can be added in an empty space by simply incrementing the value of rear



Circular queue

- Example:
  - In this array, there are only two elements and other three positions are empty
  - The rear is at the last position of the queue
  - If we try to insert the element, it will show that there are no empty spaces



- Solution:
  - We can insert the new element in the first available position, and update the rear

Applications of circular queue

- Memory management
  - The circular queue manages memory more efficiently than linear queue by placing the elements in a location which is unused

- Traffic system
  - Each traffic light gets ON one by one after an interval of time
  - Like green light gets ON for one minute, then yellow light for one minute, and then red light; after red light, the green light gets ON

Circular queue: enqueue

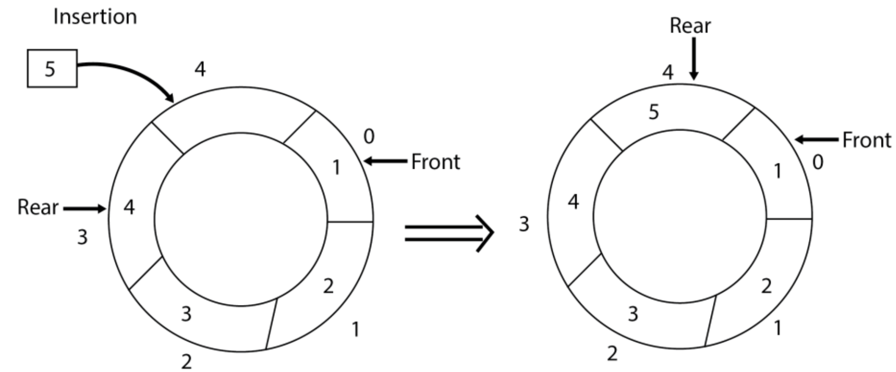
Step 1: IF (REAR+1)%MAX = FRONT
Write " OVERFLOW "
Goto step 4
[End OF IF]

Step 2: IF FRONT = -1 and REAR = -1
SET FRONT = REAR = 0
ELSE
SET REAR = (REAR + 1) % MAX
[END OF IF]

Step 3: SET QUEUE[REAR] = VAL

Step 4: EXIT

- First, check whether the queue is full or not
- Initially set both front and rear to -1
- To insert the first element, both front and rear are set to 0
- To insert a new element, the rear gets incremented



Circular queue: dequeue

Step 1: IF FRONT = -1
Write " UNDERFLOW "
Goto Step 4
[END of IF]

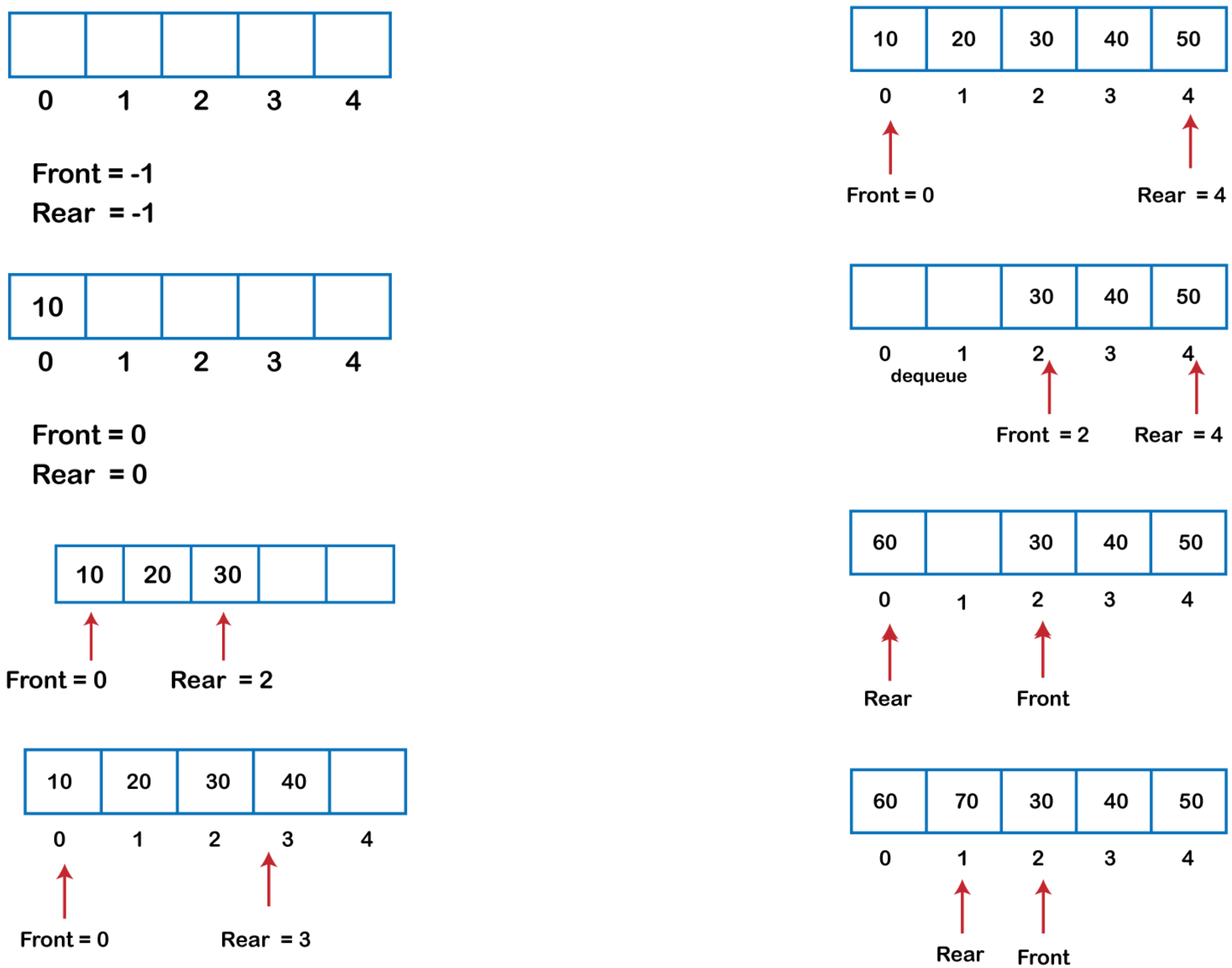
Step 2: SET VAL = QUEUE[FRONT]

Step 3: IF FRONT = REAR
SET FRONT = REAR = -1
ELSE
IF FRONT = MAX -1
SET FRONT = 0
ELSE
SET FRONT = FRONT + 1
[END of IF]
[END of IF]

Step 4: EXIT

- First, check whether the queue is empty or not, and if the queue is empty, we cannot perform the dequeue operation
- When the element is deleted, the value of front gets incremented by 1
- If there is only one element left to be deleted, then the front and rear are reset to -1

Circular queue: an example



Java implementation codes

```java
import java.io.*;
public class CircularQ {
    int Q[] = new int[100];
    int n, front, rear;
    public CircularQ(int nn) {
        n = nn;
        front = rear = 0;
    }
    public void enqueue(int v) {
        if((rear + 1) % n != front) {
            rear = (rear + 1) % n;
            Q[rear] = v;
        }
        else
            System.out.println("Queue is full !");
    }
    public int dequeue() {
        int v;
        if(front != rear) {
            front = (front + 1) % n;
            v = Q[front];
            return v;
        }
        else
            return -9999;
    }
    public void disp() {
        int i;
        if(front != rear) {
            i = (front + 1) % n;
            while(i != rear) {
                System.out.println(Q[i]);
                i = (i + 1) % n;
            }
            System.out.println(Q[i]);
        }
        else
            System.out.println("Queue is empty !");
    }
```

```java
    public static void main(String args[]) throws IOException{
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        System.out.print("Enter the size of the queue : ");
        int size = Integer.parseInt(br.readLine());
        CircularQ call = new CircularQ(size + 1);
        int choice;
        boolean exit = false;
        while(!exit) {
            System.out.print("\n1 : Add\n2 : Delete\n" +
                    "3 : Display\n4 : Exit\n\nYour Choice : ");
            choice = Integer.parseInt(br.readLine());
            switch(choice) {
                case 1 :
                    System.out.print("\nEnter number to be added : ");
                    int num = Integer.parseInt(br.readLine());
                    call.enqueue(num);
                    break;
                case 2 :
                    int popped = call.dequeue();
                    if(popped != -9999)
                        System.out.println("\nDeleted : " + popped);
                    else
                        System.out.println("\nQueue is empty !");
                    break;
                case 3:
                    call.disp();
                    break;
                case 4:
                    exit = true;
                    break;
                default:
                    System.out.println("\nWrong Choice !");
                    break;
            }
        }
    }
}
```

```
Enter the size of the queue : 5

1 : Add
2 : Delete
3 : Display
4 : Exit

Your Choice : 1

Enter number to be added : 31

1 : Add
2 : Delete
3 : Display
4 : Exit

Your Choice : 1

Enter number to be added : 28

1 : Add
2 : Delete
3 : Display
4 : Exit

Your Choice : 1

Enter number to be added : 9

1 : Add
2 : Delete
3 : Display
4 : Exit

Your Choice : 1

Enter number to be added : 56

1 : Add
2 : Delete
3 : Display
4 : Exit

Your Choice : 3
31
28
9
56

1 : Add
2 : Delete
3 : Display
4 : Exit
```

Priority queue

- Behaves similarly to normal queue, except that each element has some priority
  - The priority of the elements in a priority queue will determine the order in which elements are removed from the queue
  - The element with the highest priority would dequeue first
  - Thus, it does not follow the FIFO principle
- Only supports comparable elements
  - Elements can be arranged either in an ascending or descending order

Characteristics of priority queue

- Consider a priority queue with the following values arranged in ascending order (smaller number has higher priority):

  1, 3, 4, 8, 14, 22

- Now, we will see how it will handle following operations:
  - poll( ): remove the highest priority element from the priority queue; Since the element '1' has the highest priority, it will be removed from the priority queue   3, 4, 8, 14, 22
  - add(2): insert '2' element in a priority queue; Since 2 is the smallest element among all the numbers so it will obtain the highest priority   2, 3, 4, 8, 14, 22
  - poll( ): remove '2' element from the priority queue as it has the highest priority queue   3, 4, 8, 14, 22
  - add(5): in the ascending order, 5 will be inserted after 4 since 5 is larger than 4 and lesser than 8, so it will obtain the third highest priority   3, 4, 5, 8, 14, 22

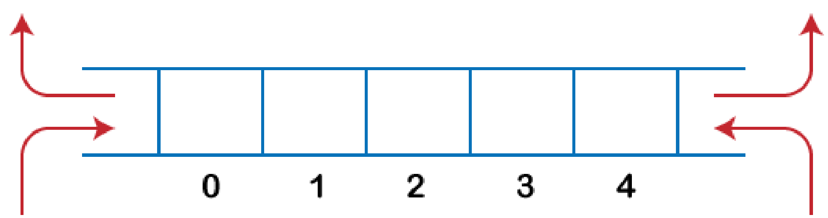
Priority queue: implementation

- Four different ways
  - Array
  - Linked list
  - Binary search tree
  - Heap data structure

Heap is the most efficient way
(we will learn it later)

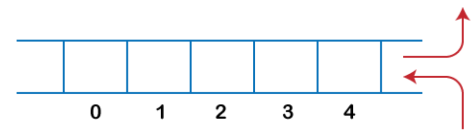
Double ended queue

- Double ended queue (Deque) is a linear data structure, which is a generalized queue
  - It does not follow the FIFO principle
  - Insertion and deletion can occur from both ends

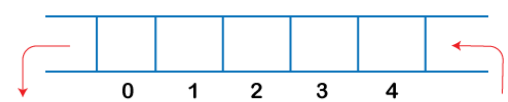


Double ended queue: examples

- Used as stack
  - The insertion and deletion operation can be performed from one side
  - The stack follows the LIFO rule in which both the insertion and deletion can be performed only from one end



- Used as queue
  - The insertion can be performed on one end, and the deletion can be done on another end
  - The queue follows the FIFO rule in which the element is inserted on one end and deleted from another end



Double ended queue: operations

- **enqueue_front():** it is used to insert the element from the front end
- **enqueue_rear():** it is used to insert the element from the rear end

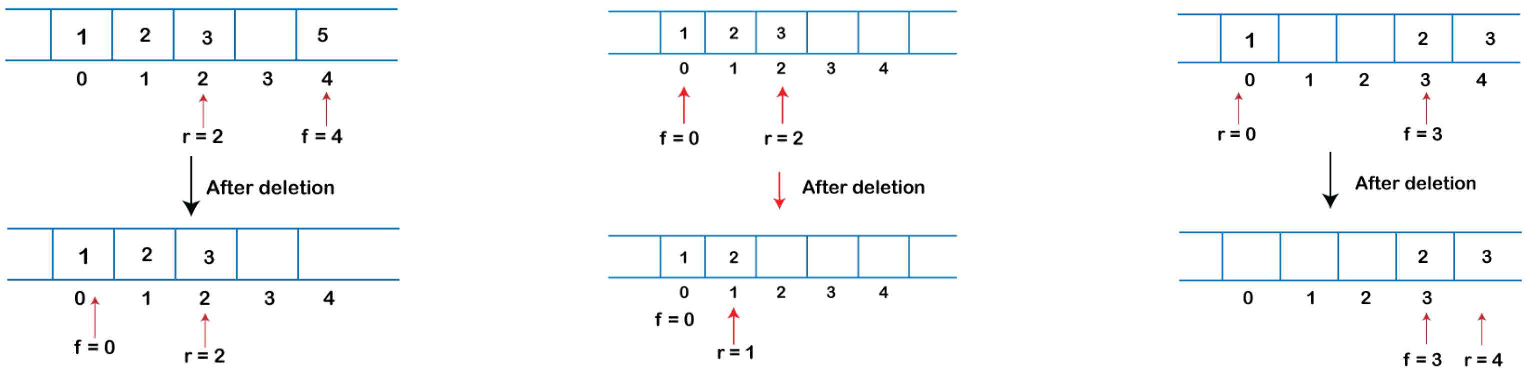
- **dequeue_front():** it is used to delete the element from the front end
- **dequeue_rear():** it is used to delete the element from the rear end

- **getfront():** it is used to return the front element of the deque
- **getrear():** it is used to return the rear element of the deque

Double ended queue: enqueue

- Initially, the double ended queue is empty; both front and rear are -1, i.e., f = r = -1.
- Suppose we have inserted element 1, then f = 0 and r = 0.
- To insert the element from the rear end, increment the rear, i.e., rear=rear+1. Now, the rear is pointing to the second element, and the front is pointing to the first element.
- To again insert the element from the rear, first increment the rear, and now rear points to the third element.
- To insert the element from the front end, and insert an element from the front, decrement the value of front by 1. Then the front points to -1 location, which is not any valid location in an array. So, set the front as (n-1), which is equal to 4 as n is 5.

Double ended queue: dequeue



- Suppose the front is pointing to the last element. To delete an element from the front, set front=front+1. Currently, the front is 4, and it becomes 5 which is not valid. Therefore, if front points to the last element, then front is set to 0 for delete operation.

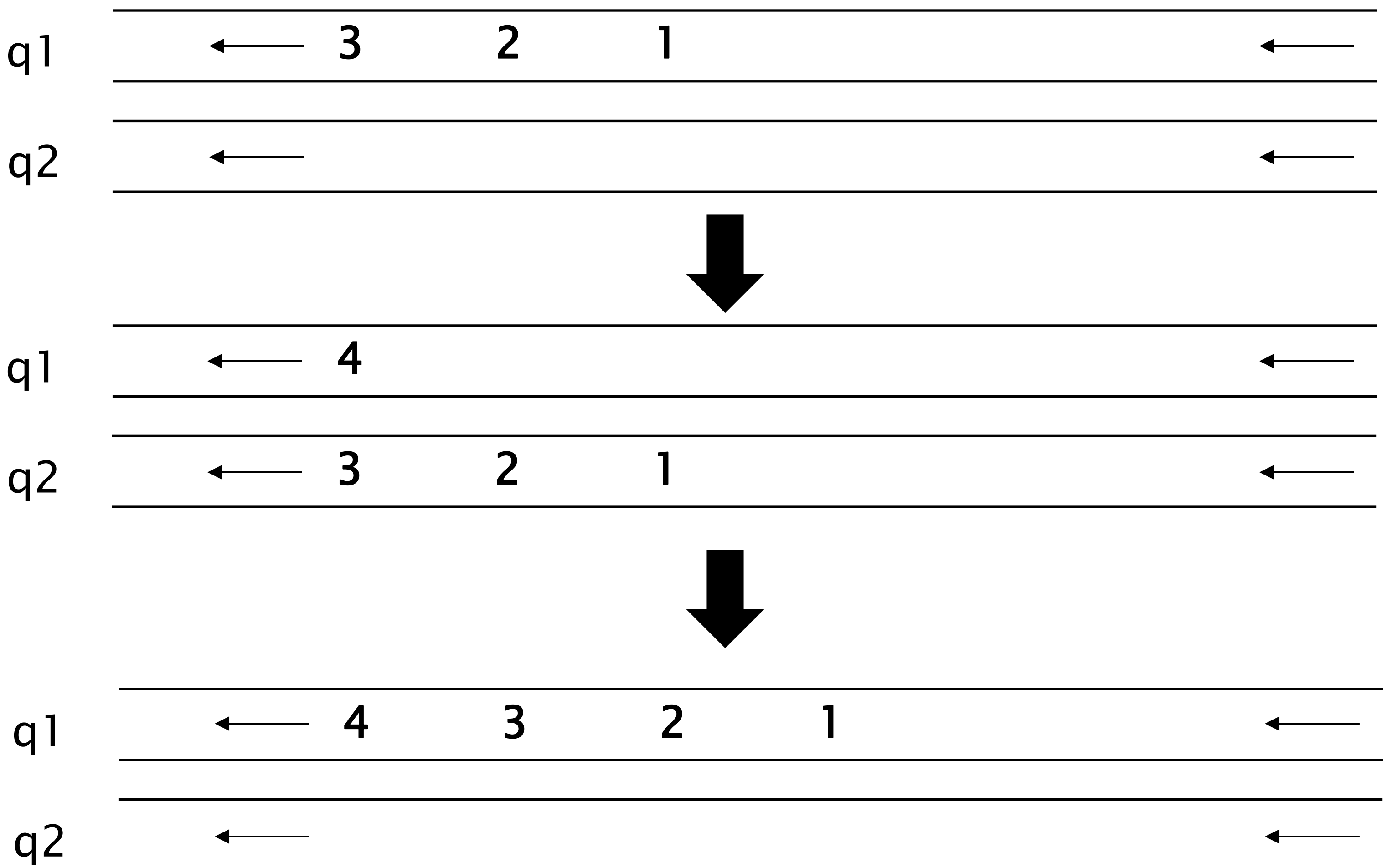
- To delete the element from rear end, then decrement the rear value by 1, i.e., rear=rear-1.

- Suppose the rear is pointing to the first element. To delete the element from the rear end, set rear=n-1 where n is the size of the array.

Exercise: implementing stack using queue

```java
import java.util.ArrayDeque;
import java.util.Queue;
// Implement stack using two queues
class Stack<T> {
    Queue<T> q1, q2;
    // Constructor
    public Stack() {
        q1 = new ArrayDeque<>();
        q2 = new ArrayDeque<>();
    }
    // Insert an item into the stack
    void add(T data) {
        // Move all elements from the first queue to the second queue
        while (!q1.isEmpty()) {
            q2.add(q1.peek());
            q1.poll();
        }
        // Push the given item into the first queue
        q1.add(data);
        // Move all elements back to the first queue from the second queue
        while (!q2.isEmpty()) {
            q1.add(q2.peek());
            q2.poll();
        }
    }
    // Remove the top item from the stack
    public T poll() {
        // if the first queue is empty
        if (q1.isEmpty()) {
            System.out.println("Underflow!!");
            System.exit(0);
        }
        // return the front item from the first queue
        T front = q1.peek();
        q1.poll();
        return front;
    }
}
class Main {
    public static void main(String[] args) {
        int[] keys = { 1, 2, 3, 4, 5 };
        // insert the above keys into the stack
        Stack<Integer> s = new Stack<Integer>();
        for (int key: keys) {
            s.add(key);
        }
        for (int i = 0; i <= keys.length; i++) {
            System.out.println(s.poll());
        }
    }
}
```

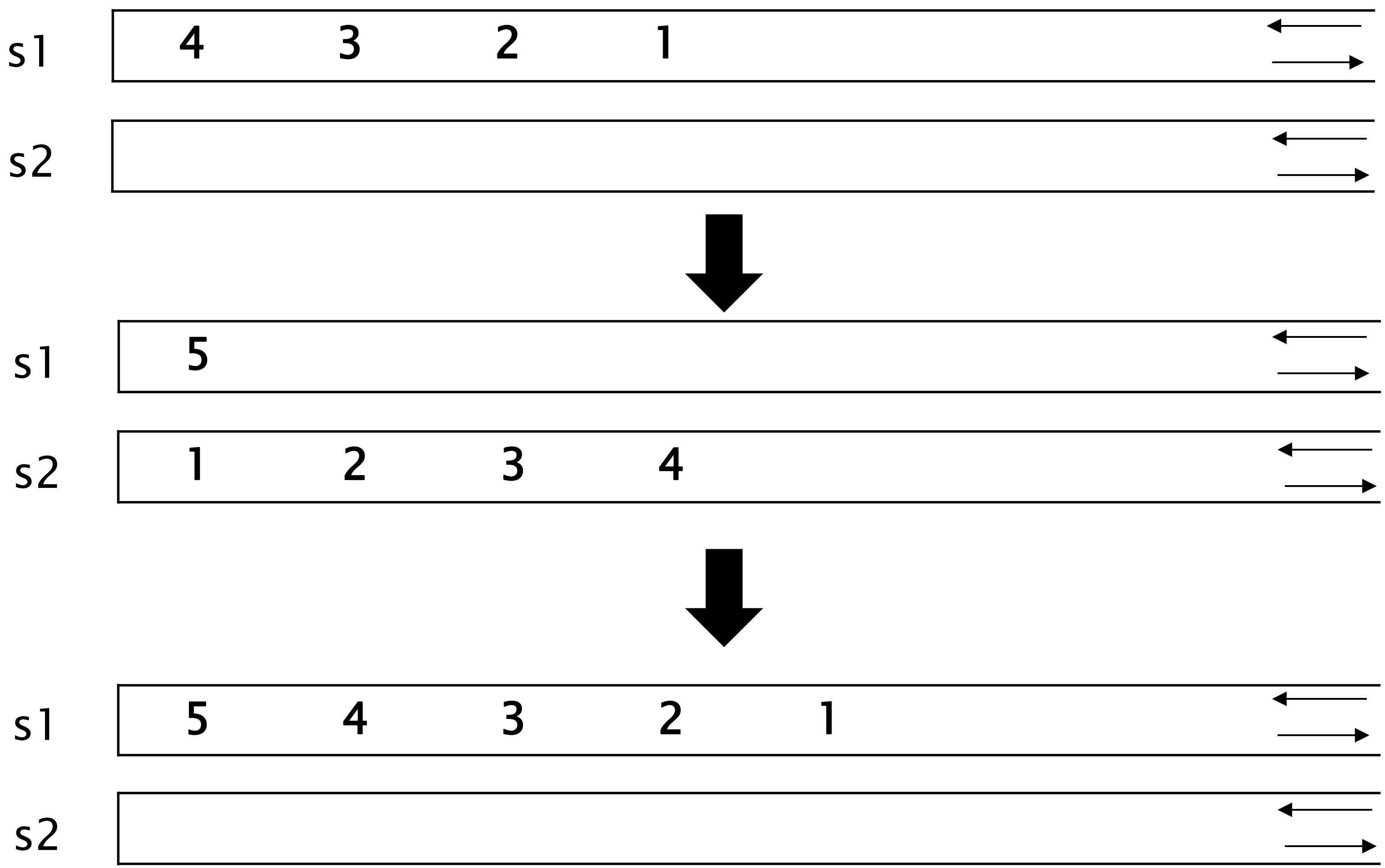
Exercise: implementing stack using queue



Exercise: implementing queue using stack

```java
import java.util.Stack;
// Implement a queue using two stacks
class Queue<T> {
    private Stack<T> s1, s2;
    // Constructor
    Queue() {
        s1 = new Stack<>();
        s2 = new Stack<>();
    }
    // Add an item to the queue
    public void enqueue(T data) {
        // Move all elements from the first stack to the second stack
        while (!s1.isEmpty()) {
            s2.push(s1.pop());
        }

        // push item into the first stack
        s1.push(data);

        // Move all elements back to the first stack from the second stack
        while (!s2.isEmpty()) {
            s1.push(s2.pop());
        }
    }
    // Remove an item from the queue
    public T dequeue() {
        // if the first stack is empty
        if (s1.isEmpty())
        {
            System.out.println("Underflow!!");
            System.exit(0);
        }

        // return the top item from the first stack
        return s1.pop();
    }
}
class Main {
    public static void main(String[] args) {
        int[] keys = { 1, 2, 3, 4, 5 };
        Queue<Integer> q = new Queue<Integer>();
         // insert above keys
        for (int key: keys) {
            q.enqueue(key);
        }
        System.out.println(q.dequeue());    // print 1
        System.out.println(q.dequeue());    // print 2
    }
}
```

Exercise: implementing queue using stack



Recommended reading

- Reading
  - Chapter 10, textbook

- Next lectures
  - Sorting algorithms