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

CSC3100 Data Structures
Lecture 10: Queue

Yixiang Fang
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

Queue

- The queue on a set S of n elements supports two constrained update operations:
  - 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

- It can be found in various real-world scenarios…
- Used as buffers MP3 media 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
  - Handling interrupts
  - Data transformation
  - Scheduling shared resources like printer, disk, CPU, etc.

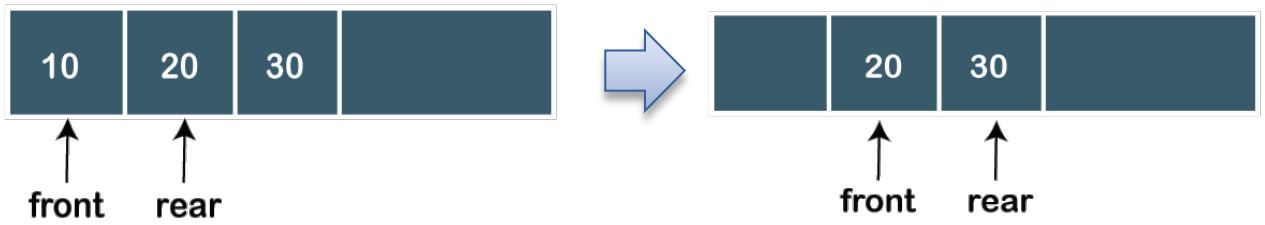
Queue applications

- Suppose we have a printer shared between various machines in a network, and it can serve a single request at a time
  - When any print request comes from the network, and if the printer is busy, the printer's program will put the print request in a queue
  - If the requests are available in the queue, the printer takes a request from the front of the queue, and serves it
- The processor in a computer is a shared resource
  - There are multiple requests that the processor must execute, but the processor can execute a single process at a time
  - The processes are kept in a queue for execution

Linear queue

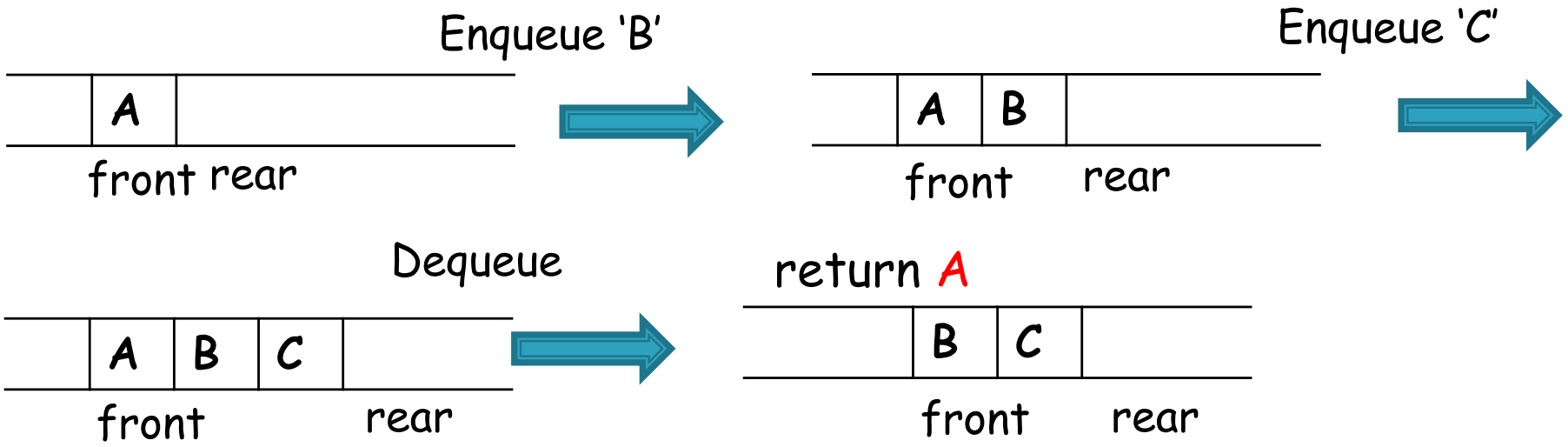
- **Linear queue:** an insertion takes place from the rear while the deletion occurs from the front
  - The elements are inserted from the rear; to insert more elements in a queue, the rear value gets incremented on every insertion

- Drawback: insertion is done only from the rear end
  - If the first several elements are deleted from the queue, we cannot insert more elements even though the space is available in a linear queue



Linear queue: example

- Example: a queue Q with only one element 'A'
  - 1. Enqueue 'B' to Q
  - 2. Enqueue 'C' to Q
  - 3. Deqeue from Q



Linear queue implementations

- Option 1: Linked list
- Option 2: Arrays

Queue design with linked list

- CreateQueue: create a linked list L

Algorithm: *CreateQueue(capacity)*

```
1 Q<-Allocate new memory for queue
2 Q.L <- allocate new memory for list
3 return Q
```

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

Algorithm: Enqueue(Q,e)

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

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

Algorithm: Dequeue(Q)

```
1 if isEmpty(Q.L) error “Queue empty”
2 frontNode = Q.L.head.next
3 frontElement = frontNode.element.
4 delete(Q.L, frontNode)
5 return frontElement
```

Queue design with linked list

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

Algorithm: *isEmpty(Q)*

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

- isFull: always return NO

Algorithm: *isFull(Q)*

```
1 return false
```

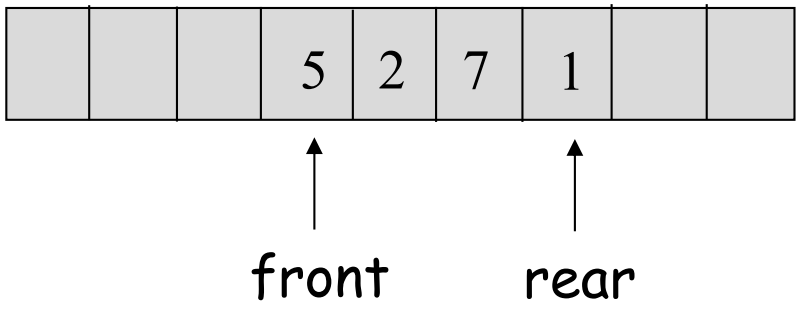
Java codes

```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");
        }
    }
}
```

Queue using array

- Array implementation

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



Operations:

- To enqueue an element X

  Increment size and rear, then set Queue[rear] = X

- To dequeue an element

  Return the value of Queue[front], decrease size, and then increment front

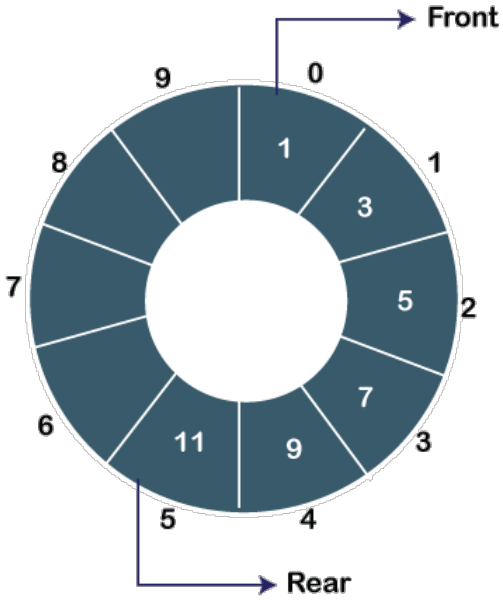
```java
// A class to represent a queue
class Queue {
    private int[] arr;      // array to store queue elements
    private int front;      // front points to the front element in the queue
    private int rear;       // rear points to the last element in the queue
    private int capacity;   // maximum capacity of the queue
    private int count;      // current size of the queue

    // Constructor to initialize a queue
    Queue(int size) {
        arr = new int[size];
        capacity = size;
        front = 0;
        rear = -1;
        count = 0;
    }

    // Utility function to dequeue the front element
    public void dequeue() {
        // check for queue underflow
        if (isEmpty())
        {
            System.out.println("Underflow\nProgram Terminated");
            System.exit(1);
        }

        System.out.println("Removing " + arr[front]);

        front = (front + 1) % capacity;
        count--;
    }

    // Utility function to add an item to the queue
    public void enqueue(int item) {
        // check for queue overflow
        if (isFull()) {
            System.out.println("Overflow\nProgram Terminated");
            System.exit(1);
        }

        System.out.println("Inserting " + item);

        rear = (rear + 1) % capacity;
        arr[rear] = item;
        count++;
    }

    // Utility function to return the front element of the queue
    public int peek() {
        if (isEmpty()) {
            System.out.println("Underflow\nProgram Terminated");
            System.exit(1);
        }
        return arr[front];
    }

    // Utility function to return the size of the queue
    public int size() {
        return count;
    }

    // Utility function to check if the queue is empty or not
    public Boolean isEmpty() {
        return (size() == 0);
    }

    // Utility function to check if the queue is full or not
    public Boolean isFull() {
        return (size() == capacity);
    }
}

class Main {
    public static void main (String[] args) {
        // create a queue of capacity 5
        Queue q = new Queue(5);
        q.enqueue(1);
        q.enqueue(2);
        q.enqueue(3);
        System.out.println("The front element is " + q.peek());
        q.dequeue();
        System.out.println("The front element is " + q.peek());
        System.out.println("The queue size is " + q.size());

        q.dequeue();
        q.dequeue();
        if (q.isEmpty()) {
            System.out.println("The queue is empty");
        } else {
            System.out.println("The queue is not empty");
        }
    }
}
```

Queue using array

- Problem: may run out of rooms
- 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

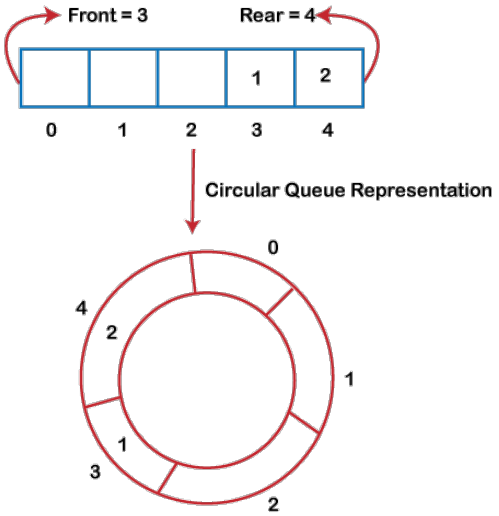
- All the nodes are represented as circular
  - Similar to the linear queue except that the last element of the queue is connected to the first one
  - A.k.a Ring Buffer as all the ends are connected to another end



- 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:
  - The rear is at the last position of the queue and front is pointing somewhere rather than the 0^{th} position

Applications of circular queue

- Memory management
  - The circular queue managed memory more efficiently (than linear queue) by placing the elements in a location which is unused
- CPU scheduling
  - The operating system also uses the circular queue to insert the processes and then execute them
- Traffic system
  - In a computer-control traffic system, traffic light is one of the best examples of the circular queue
  - Each light of traffic light gets ON one by one after every interval of time
  - Like red light gets ON for one minute then yellow light for one minute and then green light; After green light, the red light gets ON

Circular queue: enqueue algorithm

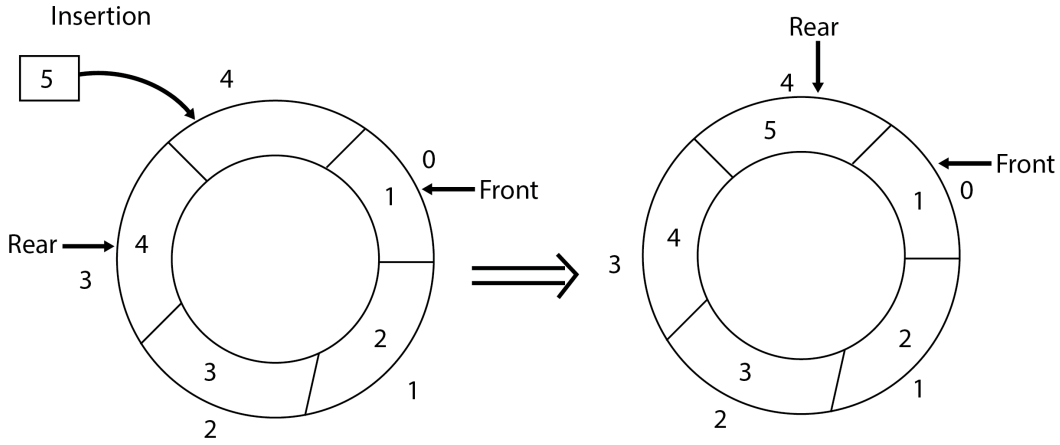
Step 1: IF (REAR+1)%MAX = FRONT
Write " OVERFLOW "
Goto step 4
[End OF IF]

Step 2: IF FRONT = -1 and REAR = -1
SET FRONT = REAR = 0
ELSE IF REAR = MAX - 1 and FRONT ! = 0
SET 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 algorithm

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

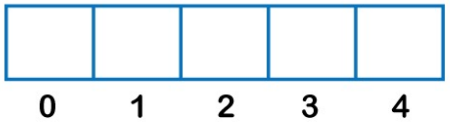
Step 2: SET VAL = QUEUE[FRONT]

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]

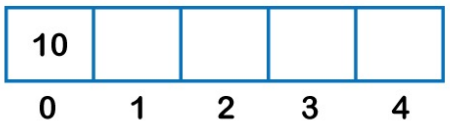
Step 4: EXIT

- 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 decremented by 1
- If there is only one element left to be deleted, then the front and rear are reset to -1

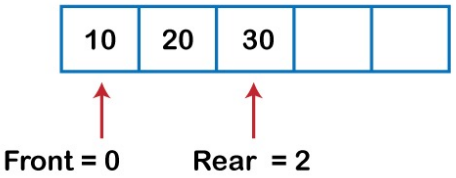
Circular queue: an example

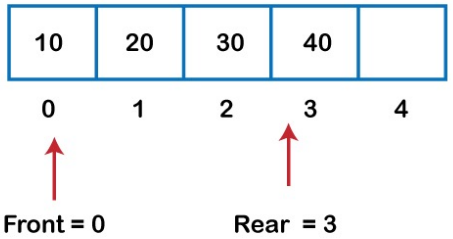


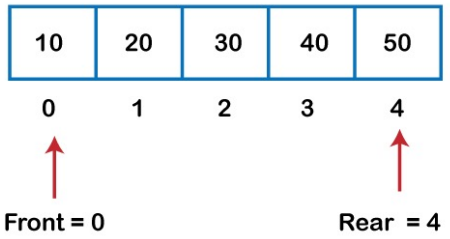
Front = -1
Rear = -1

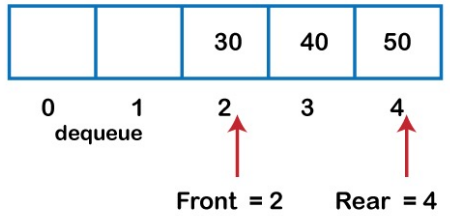


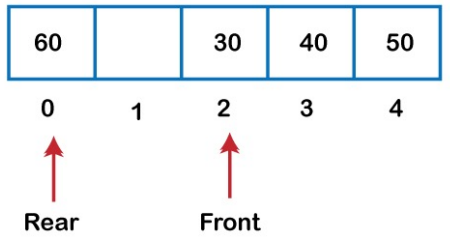
Front = 0
Rear = 0

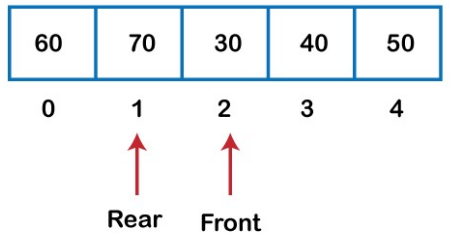












```java
import java.io.*;
class CircularQ {
    int Q[] = new int[100];
    int n, front, rear;
    static BufferedReader br = new BufferedReader(new
            InputStreamReader(System.in));
    public CircularQ(int nn) {
        n=nn;
        front = rear = 0;
    }
    public void enqueue(int v) {
        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;
            }
        }
        else
            System.out.println("Queue is empty !");
    }
    public static void main(String args[]) throws IOException {
        System.out.print("Enter the size of the queue : ");
        int size = Integer.parseInt(br.readLine());
        CircularQ call = new CircularQ(size);
        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

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

Priority queue

- Behaves similarly to normal queue, except that each element has some priority
  - The element with the highest priority would come first in a priority queue
  - The priority of the elements in a priority queue will determine the order in which elements are removed from the queue
- Only supports comparable elements
  - Elements are either in an ascending or descending order
- Example
  - Suppose we have values 1, 3, 4, 8, 14, 22 inserted in a priority queue with an ordering imposed on the values is from least to the greatest, so 1 has the highest priority while 22 has the lowest priority

Characteristics of priority queue

- Every element in a priority queue has some priority associated with it
- An element with the higher priority will be deleted before the deletion of the lesser priority
- If two elements in a priority queue have the same priority, they will be arranged using the FIFO principle

Characteristics of priority queue

- Consider a priority queue with the following values:
  1, 3, 4, 8, 14, 22

- All values are arranged in ascending order. Now, we will see how it will look after the following operations:
  - poll(): remove the highest priority element from the priority queue. The '1' element has the highest priority, so it will be removed from the priority queue.
  - add(2): Insert '2' element in a priority queue. As 2 is the smallest element among all the numbers so it will obtain the highest priority.
  - poll(): Remove '2' element from the priority queue as it has the highest priority queue.
  - add(5): Insert 5 element after 4 as 5 is larger than 4 and lesser than 8, so it will obtain the third highest priority in a priority queue.

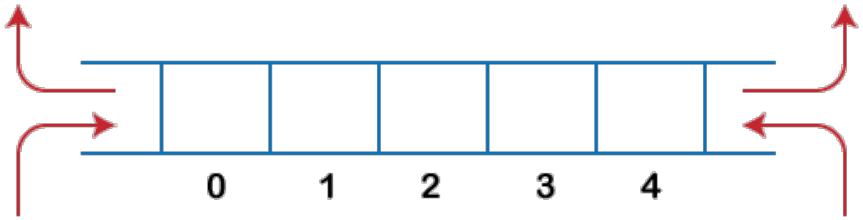
Priority queue: implementation

- Four ways
  - Arrays
  - Linked list
  - Heap data structure
  - Binary search tree

- The heap data structure is the most efficient way
  - We will learn it later

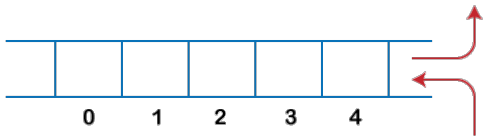
Double ended queue

- Double ended queue is a linear data structure, a generalized queue
  - Does not follow the FIFO principle
  - Insertion and deletion can occur from both ends

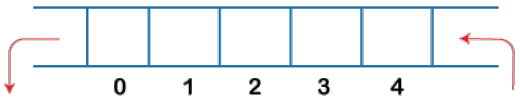


Double ended queue: examples

- Used both as stack and queue
  - 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

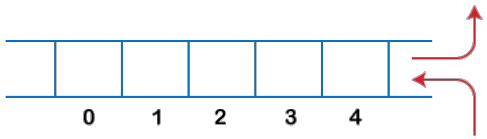


  - The insertion can be performed on one end, and the deletion can be done on another end
  - The queue follows the FIFO rule in which the element is inserted on one end and deleted from another end

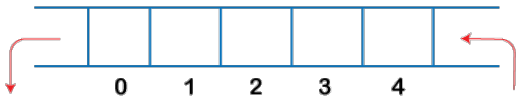


Double ended queue: examples

- Input-restricted queue:
  - Some restrictions are applied to the insertion: the insertion is applied to one end while the deletion is applied from both the ends



- Output-restricted queue:
  - some restrictions are applied to the deletion operation: the deletion can be applied only from one end, whereas the insertion is possible from both ends



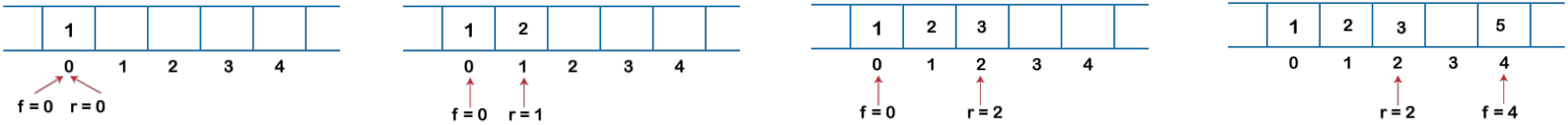
Double ended queue: operations

- **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

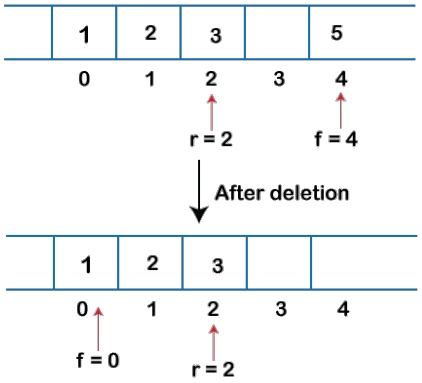
- **getfront()**: It is used to return the front element of the deque
- **getrear()**: It is used to return the rear element of the deque

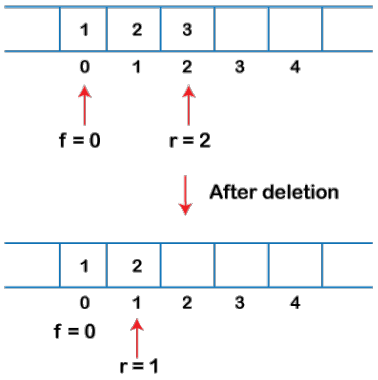
Double ended queue: enqueue

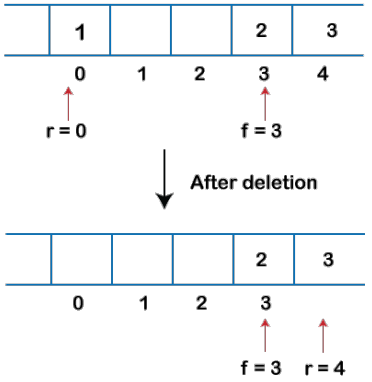


- Initially, the deque is empty. Both front and rear are -1, i.e., f = -1 and r = -1.
- As the deque is empty, so inserting an element either from the front or rear end would be the same thing. Suppose we have inserted element 1, then front is equal to 0, and the rear is also equal to 0.
- To insert 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 stacks using queues

```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 queues using stacks

```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: The Josephus Problem

- The Josephus Problem
  - There are n people standing in a circle waiting to be executed. After the first man is executed, k - 1 people are skipped and the k-th man is executed. Then again, k-1 people are skipped and the k-th man is executed. The elimination proceeds around the circle (which is becoming smaller and smaller as the executed people are removed), until only the last man remains, who is given freedom.

  - The task is to choose the place in the initial circle so that you survive, given n and k.

Recommended reading

- Reading
  - Chapter 10, textbook
- Next lectures
  - Sorting algorithms