

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



Real-world queues

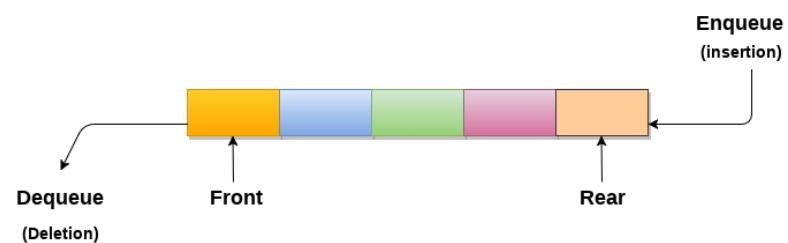
- ▶ A queue of customers waiting to buy a product from a shop, where the customer that came first is served first
- ▶ A queue of students waiting to take a bus at a bus station, where the student that came first gets on first





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

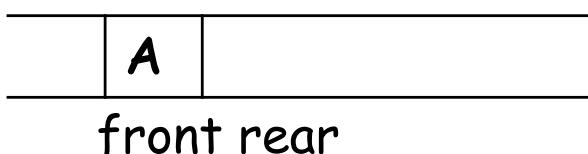
- ▶ Used as buffers MP3 player, CD player, etc.
 - Used to maintain the play list in media players in order to add and remove the songs from the play-list
- ▶ Used in OS
 - Multiple data transformation tasks
 - Scheduling shared resources like CPU and GPU
- ▶ Suppose we have a printer shared by several 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



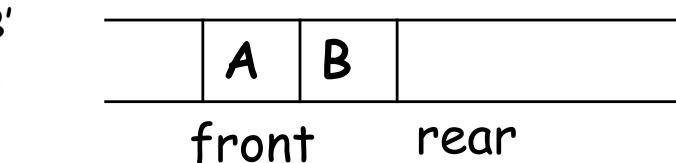


Linear queue

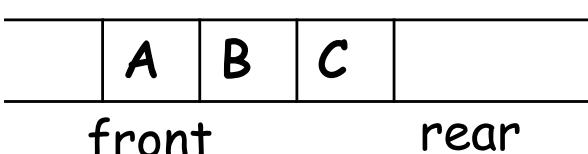
- ▶ A linear queue is generally referred to as queue, or a linear data structure that follows the FIFO
- ▶ Example: a queue Q with only one element 'A'
 - 1. Enqueue 'B' to Q
 - 2. Enqueue 'C' to Q
 - 3. Dequeue from Q



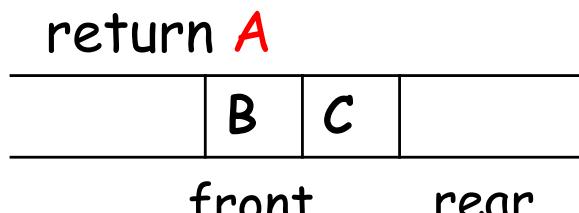
Enqueue 'B'



Enqueue 'C'



Dequeue



return A



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)`

```
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)
```

- ▶ `dequeue`: 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 implementation codes

```
1 // A Linked List Node
2 class Node {
3     int data;          // integer data
4     Node next;        // pointer to the next node
5     public Node(int data) {
6         // set data in the allocated node and return it
7         this.data = data;
8         this.next = null;
9     }
10 }
11 class Queue {
12     private static Node rear = null, front = null;
13     // Utility function to dequeue the front element
14     public static int dequeue() {    // delete at the beginning
15         if (front == null) {
16             System.out.print("\nQueue Underflow");
17             System.exit(1);
18         }
19         Node temp = front;
20         System.out.printf("Removing %d\n", temp.data);
21         // advance front to the next node
22         front = front.next;
23         // if the list becomes empty
24         if (front == null) {
25             rear = null;
26         }
27         // deallocate the memory of the removed node and
28         // optionally return the removed item
29         int item = temp.data;
30         return item;
31     }
32     // Utility function to add an item to the queue
33     public static void enqueue(int item) {    // insertion at the end
34         // allocate a new node in a heap
35         Node node = new Node(item);
36         System.out.printf("Inserting %d\n", item);
37         // special case: queue was empty
38         if (front == null) {
39             // initialize both front and rear
40             front = node;
41             rear = node;
42         } else {
43             // update rear
44             rear.next = node;
45             rear = node;
46         }
47     }
}
```

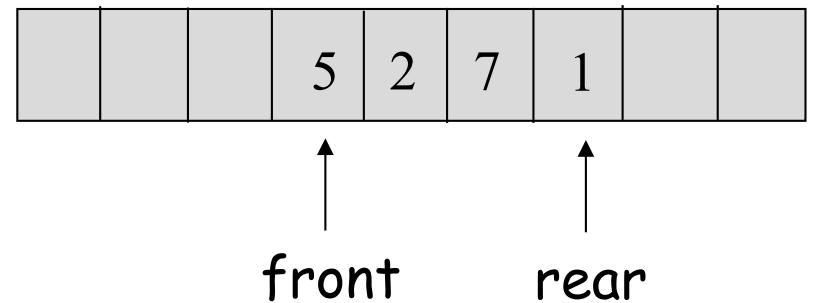
```
48     // Utility function to return the top element in a queue
49     public static int peek() {
50         // check for an empty queue
51         if (front != null) {
52             return front.data;
53         } else {
54             System.exit(1);
55         }
56     }
57     return -1;
58 }
59     // Utility function to check if the queue is empty or not
60     public static boolean isEmpty() {
61         return rear == null && front == null;
62     }
63 }
64
65 class Main {
66     public static void main(String[] args) {
67         Queue q = new Queue();
68         q.enqueue(1);
69         q.enqueue(2);
70         q.enqueue(3);
71         q.enqueue(4);
72         System.out.printf("The front element is %d\n", q.peek());
73         q.dequeue();
74         q.dequeue();
75         q.dequeue();
76         q.dequeue();
77         if (q.isEmpty()) {
78             System.out.print("The queue is empty");
79         } else {
80             System.out.print("The queue is not empty");
81         }
82     }
83 }
```



Queue design with array

▶ Array implementation

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



Operations:

- To **enqueue** an element X
Increase size and rear, then set
array[rear] = X
- To **dequeue** an element
Return the value of array[front],
decrease size, and then
increase front

```

1 // A class to represent a queue
2 class Queue {
3     private int[] arr;      // array to store queue elements
4     private int front;       // front points to the front element in the queue
5     private int rear;        // rear points to the last element in the queue
6     private int capacity;    // maximum capacity of the queue
7     private int count;       // current size of the queue
8
9     // Constructor to initialize a queue
10 Queue(int size) {
11     arr = new int[size];
12     capacity = size;
13     front = 0;
14     rear = -1;
15     count = 0;
16 }
17
18 // Utility function to dequeue the front element
19 public void dequeue() {
20     // check for queue underflow
21     if (isEmpty())
22     {
23         System.out.println("Underflow\nProgram Terminated");
24         System.exit(1);
25     }
26
27     System.out.println("Removing " + arr[front]);
28
29     front = (front + 1) % capacity;
30     count--;
31 }
32
33 // Utility function to add an item to the queue
34 public void enqueue(int item) {
35     // check for queue overflow
36     if (isFull())
37     {
38         System.out.println("Overflow\nProgram Terminated");
39         System.exit(1);
40     }
41
42     System.out.println("Inserting " + item);
43
44     rear = (rear + 1) % capacity;
45     arr[rear] = item;
46     count++;
47 }

```

```

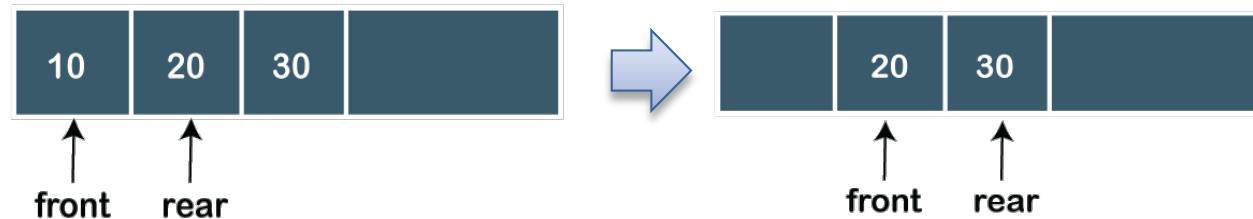
48 // Utility function to return the front element of the queue
49 public int peek() {
50     if (isEmpty())
51     {
52         System.out.println("Underflow\nProgram Terminated");
53         System.exit(1);
54     }
55     return arr[front];
56 }
57
58 // Utility function to return the size of the queue
59 public int size() {
60     return count;
61 }
62
63 // Utility function to check if the queue is empty or not
64 public Boolean isEmpty() {
65     return (size() == 0);
66 }
67
68 // Utility function to check if the queue is full or not
69 public Boolean isFull() {
70     return (size() == capacity);
71 }
72
73 class Main {
74     public static void main (String[] args) {
75         // create a queue of capacity 5
76         Queue q = new Queue(5);
77         q.enqueue(1);
78         q.enqueue(2);
79         q.enqueue(3);
80         System.out.println("The front element is " + q.peek());
81         q.dequeue();
82         System.out.println("The front element is " + q.peek());
83         System.out.println("The queue size is " + q.size());
84
85         q.dequeue();
86         q.dequeue();
87         if (q.isEmpty())
88         {
89             System.out.println("The queue is empty");
90         } else
91         {
92             System.out.println("The queue is not empty");
93         }
94     }
95 }

```



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

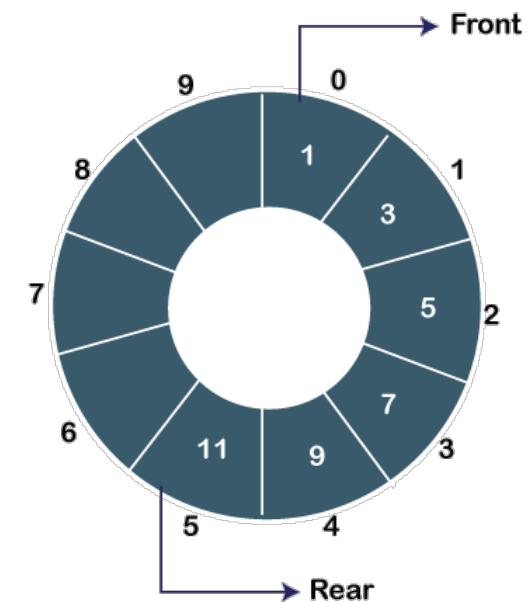


- ▶ Two solutions:
 - Keep front always at 0 by shifting the contents up the queue, but this solution is inefficient
 - Use a circular queue (wrap around & use a length variable to keep track of the queue length)



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 since 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

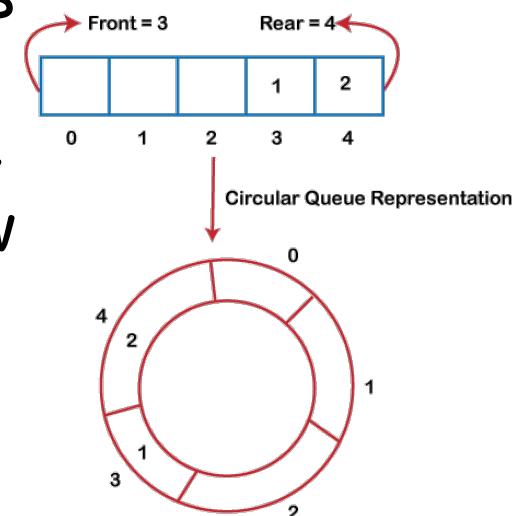




Circular queue

▶ 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:

- 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 **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

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

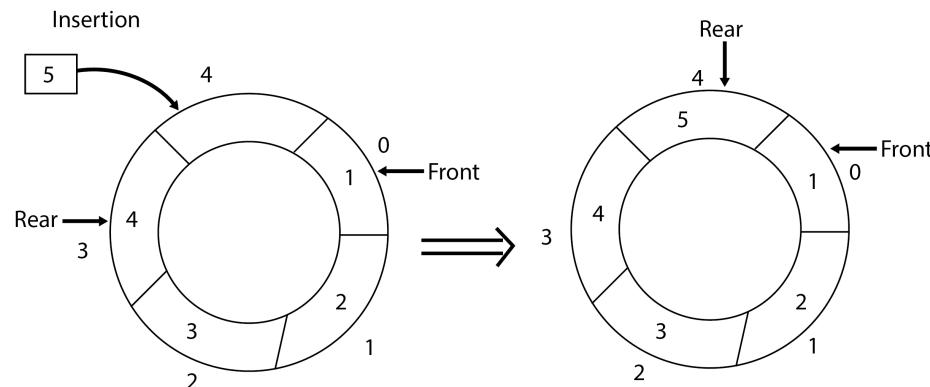
SET REAR = (REAR + 1) % MAX

[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

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]

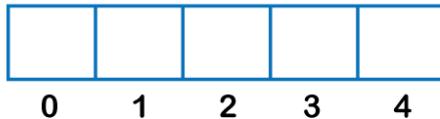
[END OF IF]

Step 4: EXIT

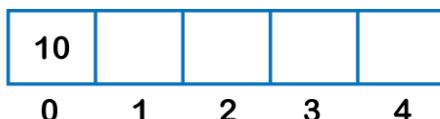
- ▶ First, check whether the queue is empty or not, and if the queue is empty, we cannot perform the dequeue operation
- ▶ When the element is deleted, the value of front gets incremented by 1
- ▶ If there is only one element left to be deleted, then the front and rear are reset to -1



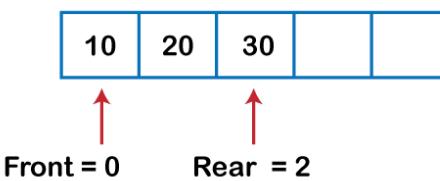
Circular queue: an example



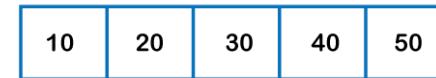
Front = -1
Rear = -1



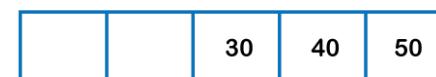
Front = 0
Rear = 0



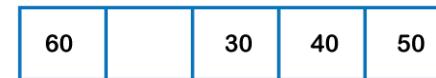
Front = 0 Rear = 3



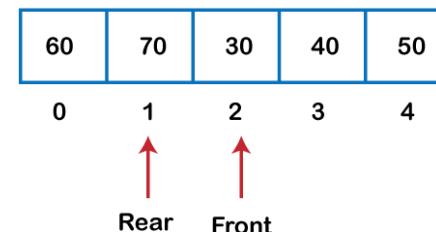
A horizontal array of 5 slots. The first slot contains the value 0, and the last slot contains the value 4. An arrow points from the label "Front = 0" to the slot containing 0. Another arrow points from the label "Rear = 4" to the slot containing 4.



Front = 2 Rear = 4



A horizontal scale with numerical labels 0, 1, 2, 3, and 4. Red arrows point upwards from the labels 'Rear' and 'Front' to the numbers 0 and 2 respectively.



```

1 import java.io.*;
2 class CircularQ {
3     int Q[] = new int[100];
4     int n, front, rear;
5     static BufferedReader br = new BufferedReader(new
6         InputStreamReader(System.in));
7     public CircularQ(int nn) {
8         n=nn;
9         front = rear = 0;
10    }
11    public void enqueue(int v) {
12        if((rear+1) % n != front) {
13            rear = (rear+1)%n;
14            Q[rear] = v;
15        }
16        else
17            System.out.println("Queue is full !");
18    }
19    public int dequeue() {
20        int v;
21        if(front!=rear) {
22            front = (front+1)%n;
23            v = Q[front];
24            return v;
25        }
26        else
27            return -9999;
28    }
29    public void disp() {
30        int i;
31        if(front != rear) {
32            i = (front +1) %n;
33            while(i!=rear) {
34                System.out.println(Q[i]);
35                i = (i+1) % n;
36            }
37        }
38        else
39            System.out.println("Queue is empty !");
40    }

```

```

41    public static void main(String args[]) throws IOException {
42        System.out.print("Enter the size of the queue : ");
43        int size = Integer.parseInt(br.readLine());
44        CircularQ call = new CircularQ(size);
45        int choice;
46        boolean exit = false;
47        while(!exit) {
48            System.out.print("\n1 : Add\n2 : Delete\n" +
49                "3 : Display\n4 : Exit\n\nYour Choice : ");
50            choice = Integer.parseInt(br.readLine());
51            switch(choice) {
52                case 1 :
53                    System.out.print("\nEnter number to be added :");
54                    int num = Integer.parseInt(br.readLine());
55                    call.enqueue(num);
56                    break;
57                case 2 :
58                    int popped = call.dequeue();
59                    if(popped != -9999)
60                        System.out.println("\nDeleted : " +popped);
61                    else
62                        System.out.println("\nQueue is empty !");
63                    break;
64                case 3 :
65                    call.disp();
66                    break;
67                case 4 :
68                    exit = true;
69                    break;
70                default :
71                    System.out.println("\nWrong Choice !");
72                    break;
73            }
74        }
75    }
76 }

```

```

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 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 deque first
 - Thus, it does not follow the FIFO principle
- ▶ Only supports comparable elements
 - Elements are either in an ascending or descending order



Characteristics of priority queue

- ▶ Consider a priority queue with the following values arranged in ascending order :
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
 - **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
 - **poll()**: remove '2' element from the priority queue as it has the highest priority queue
 - **add(5)**: insert 5 element after 4 since 5 is larger than 4 and lesser than 8, so it will obtain the third highest priority



Priority queue: implementation

- ▶ Four different ways

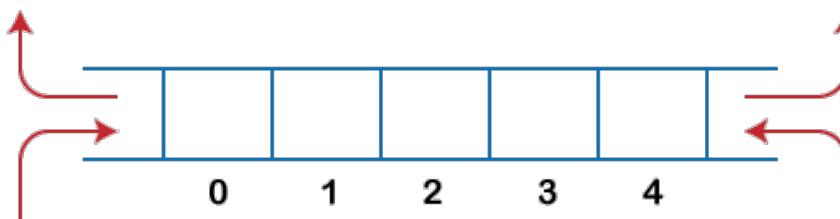
- Array
- Linked list
- Heap data structure
- Binary search tree

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



Double ended queue

- ▶ Double ended queue 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

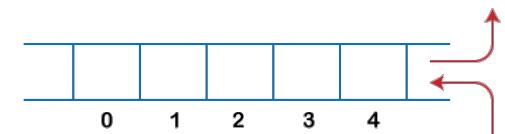




Double ended queue: examples

► 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



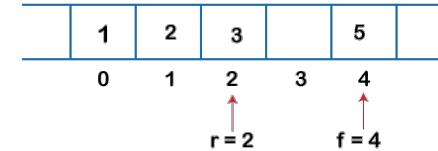
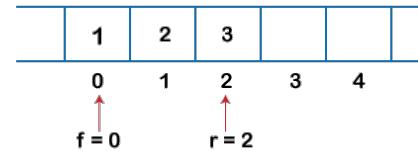
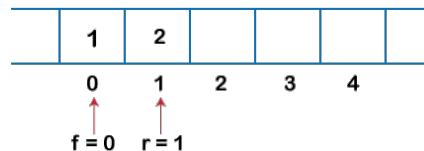
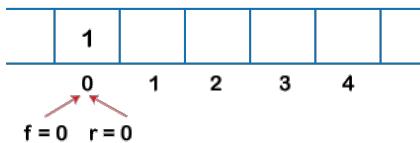


Double ended queue: operations

- `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



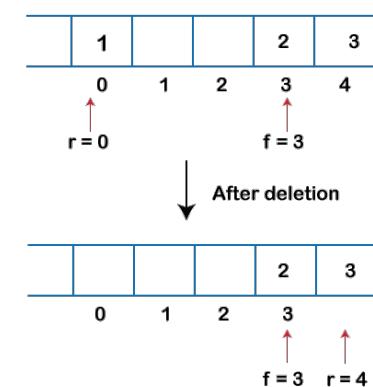
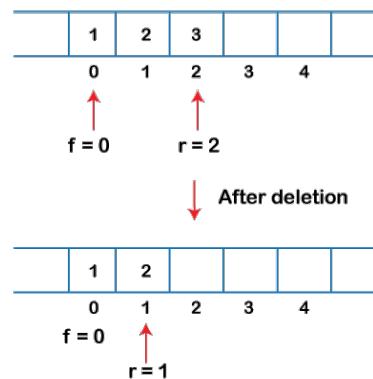
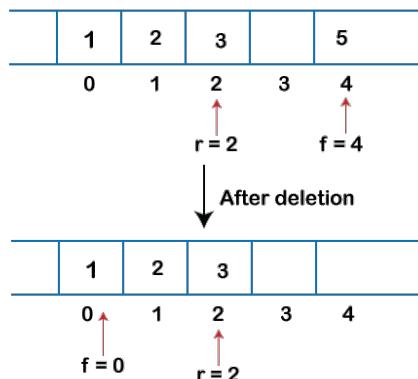
Double ended queue: enqueue



- Initially, the deque is empty; both front and rear are -1, i.e., $f = -1$ and $r = -1$.
- 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., $\text{rear}=\text{rear}+1$. 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 $\text{front}=\text{front}+1$. 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., $\text{rear}=\text{rear}-1$.
- Suppose the rear is pointing to the first element. To delete the element from the rear end, set $\text{rear}=n-1$ where n is the size of the array.



Exercise: implementing stack using queue

```
1 import java.util.ArrayDeque;
2 import java.util.Queue;
3 // Implement stack using two queues
4 ▼ class Stack<T> {
5     Queue<T> q1, q2;
6     // Constructor
7 ▼ public Stack() {
8     q1 = new ArrayDeque<>();
9     q2 = new ArrayDeque<>();
10 }
11 // Insert an item into the stack
12 ▼ void add(T data) {
13     // Move all elements from the first queue to the second queue
14     while (!q1.isEmpty()) {
15         q2.add(q1.peek());
16         q1.poll();
17     }
18     // Push the given item into the first queue
19     q1.add(data);
20     // Move all elements back to the first queue from the second queue
21     while (!q2.isEmpty()) {
22         q1.add(q2.peek());
23         q2.poll();
24     }
25 }
26 // Remove the top item from the stack
27 ▼ public T poll() {
28     // if the first queue is empty
29     if (q1.isEmpty()) {
30         System.out.println("Underflow!!!");
31         System.exit(0);
32     }
33     // return the front item from the first queue
34     T front = q1.peek();
35     q1.poll();
36     return front;
37 }
38 }
39 ▼ class Main {
40     public static void main(String[] args) {
41         int[] keys = { 1, 2, 3, 4, 5 };
42         // insert the above keys into the stack
43         Stack<Integer> s = new Stack<Integer>();
44         for (int key: keys) {
45             s.add(key);
46         }
47         for (int i = 0; i <= keys.length; i++) {
48             System.out.println(s.poll());
49         }
50     }
51 }
```



Exercise: implementing queue using stack

```
1 import java.util.Stack;
2 // Implement a queue using two stacks
3 class Queue<T> {
4     private Stack<T> s1, s2;
5     // Constructor
6     Queue() {
7         s1 = new Stack<T>();
8         s2 = new Stack<T>();
9     }
10    // Add an item to the queue
11    public void enqueue(T data) {
12        // Move all elements from the first stack to the second stack
13        while (!s1.isEmpty()) {
14            s2.push(s1.pop());
15        }
16        // push item into the first stack
17        s1.push(data);
18
19        // Move all elements back to the first stack from the second stack
20        while (!s2.isEmpty()) {
21            s1.push(s2.pop());
22        }
23    }
24    // Remove an item from the queue
25    public T dequeue() {
26        // if the first stack is empty
27        if (s1.isEmpty())
28        {
29            System.out.println("Underflow!!!");
30            System.exit(0);
31        }
32
33        // return the top item from the first stack
34        return s1.pop();
35    }
36}
37
38 class Main {
39     public static void main(String[] args) {
40         int[] keys = { 1, 2, 3, 4, 5 };
41         Queue<Integer> q = new Queue<Integer>();
42         // insert above keys
43         for (int key: keys) {
44             q.enqueue(key);
45         }
46         System.out.println(q.dequeue());    // print 1
47         System.out.println(q.dequeue());    // print 2
48     }
49 }
```



Recommended reading

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