Queue Data Structure

Introduction to Queue

- Recall: there are two commonly used data structures in computer science:
  - Stack (LIFO)
  - Queue (FIFO)

> **Queue data structure**: A queue is a data structure that organizes the stored data in a First In First Out (FIFO) manner.

- To achieve the FIFO behavior, the queue only provide the following two methods to access the data stored in a queue:
  - `enqueue(x)` : add `x` to the tail/back of the queue
  - `dequeue()` : remove the item at the head/front of the queue and return it.
- Some computer algorithms/processes with a natural LIFO behavior:
  - Scheduling for fairness:
    - FIFO is a service ordering that is fair.
    - Scheduling algorithms that serve requests from different clients often implement a FIFO service policy using a queue.
  - The Breath First Search (BFS) algorithm in graph applications:
    - The BFS algorithm will probe nodes that are nearest to the source nodes first.
    - To implement the "search the nearest nodes first" behavior, the BFS algorithm use a queue to store nodes to visit next.

The Que Interface

```java
public interface MyQueueInterface<E> {
    boolean isEmpty(); // returns true if the queue is empty
    boolean isFull(); // returns true if the queue is full

    void enqueue(E e); // insert the element e at the back of the queue

    E dequeue(); // remove the element at the front of the queue and return it

    E peek(); // return the element at the front without removing it.
}
```

Implementing a Queue using a (Circular) Array

- We can implement a queue with a fixed size array and 2 indices `head` and `tail`, where
  - `head` = location of the front (first position) of the queue
  - `tail` = location of the open spot at the tail (last position) of the queue
- The `enqueue(x)` method will append the element `x` at the tail of the queue.
- However, elements dequeued from the front of the array will cause some spaces unused. To solve it, we need to copy the elements to the front of the array, which requires many copy operations.
- Then, to avoid frequently copying arrays to save spaces, we will use a circular array instead.

> **Circular Buffer**: also known as circular array or ring buffer, is a data structure that uses an array as if it were connected end-to-end.

- The circular buffer data structure consists of an array with 2 pointers or indices:
  - `read` pointer = the index of the read position in the array
  - `write` pointer = the index of the write position in the array
- When the `write` (or `read`) pointer reaches the end of the array/buffer, the `write` (or `read`) will wrap around and reset to 0. To achieve the wrap around effect, the `read` (or `write`) pointer variables is updated using modulo arithmetic.

```
// Normal increment operation:
write = wrtie + 1;
read = read + 1;

// Increatment operation with modulo arithmetic:
write = (write + 1) % buf.length;

// Suppose writhe = 15 and buf.length = 16, then
// write = (write+1)%buf.length = (15+1)%16 = 16%16 = 0
```

- The `write()` operation on a circular buffer will store the data at the `write` pointer and advance it. The basic implementation of the `write()` operation (without checking if buffer is full):

```
void write(T e) {
    buf[write] = e;
    write = (write + 1) % buf.length;
}
```

- The `read()` operation will return the data at the `read` pointer and advance it. The basic implementation of the `read()` operation (without checking if buffer is empty):

```
T read() {
    T retVal = but[read];
    read = (read + 1) % buf.length;
    return retVal;
}
```

- How to tell if a circular buffer if empty or full:
    - The circular buffer is not empty when `read != write`
    - The circular buffer is empty when `read == write`
    - The circular buffer is not full when `read != write`
    - The circular buffer is full when `read == write` . --> This causes problem: this also means empty.
    - To avoid ambiguity, we define the circular buffer to be full when there is 1 empty slot left.
- The implementation of the queue data structure using a circular buffer:

```java
public class IntegerQueue implements MyQueueInterface<Integer> {
    private Integer[] buf; // Array of the circular buffer
    private int read; // read pointer (= head index of queue)
    private int write; // write pointer (= tail index of queue)

    // Constructor
    public IntegerQueue(int N) {
        // Variable of the circular buffer
        buf = new Integer[N]; // Create new array
        read = 0; // initialize read pointer
        write = 0; // initialize write pointer
    }

    /**
     * The queue is empty when read pointer == write pointer
     * @return true if the queue is empty
     */
    public boolean isEmpty() {
        return read == write;
    }

    /**
     * The queue is full when there is one open spot left
     * @return true if the queue is full
     */
    public boolean isFull() {
        // buffer has 1 open spot
        // <==> write 1 item into the buffer and it's full
        return (write + 1) % buf.length == read;
    }

    public void enqueue(Integer e) {
        if (isFull()) {
            System.out.println("Full"); // or throw exception
            return;
        }
        buf[write] = e;
        write = (write + 1) % buf.length;
    }

    public Integer dequeue() {
        if (isEmpty()) {
            System.out.println("Empty"); // or throw exception
            return null;
        }
        Integer retVal = buf[read];
        read = (read + 1) % buf.length;
        return retVal;
    }

    public Integer peek() {
        if (isEmpty()) {
            System.out.println("Empty"); // or throw exception
            return null;
        }
        return buf[read];
    }
}
```

Other Kinds of Queues - Double Ended (DE) Queue

> **Double Ended Queue**: A double ended queue is a data structure where we can insert or delete elements from either end

- The operations on a `Deque` are:
    - `addFirst(x)` : insert `x` at the front of the `Deque`
    - `addLast(x)` : insert `x` at the tail of the `Deque`
    - `removeFirst(x)` : remove the element at the front of the `Deque` and return it
    - `removeLast(x)` : remove the element at the tail of the `Deque` and return it
- A Double Ended Queue can work/operate like a stack. It can also work/operate like a queue.

Other Kinds of Queues - Priority Queue

> **Priority Queue**: A priority queue is a data structure where
>
> - The stored data items can be ranked by some field `rank` in the data
> - The `dequeue()` operation will always remove the item in the priority queue that has the highest value in the `rank` field.

- The priority queue is important in time cricical applications:
    - Select the next job to run (pick the most urgent one)
    - Select the next patient to treat, etc.
- A native implementation of the priority queue:

    - Use a fixed size array
    - `enqueue(x)` must make sure the array is sorted all the time (from large to small)
    - `dequeue(x)` removes the first element in the array
- Another naive implementation of the priority queue:
    - Use a fixed size array
    - `enqueue(x)` insert the element at the end
    - `dequeue(x)` search the array for the largest element and remove it
- An efficient implementation of the priority queue:
    - The heat data structure
    - A heap is a complete binary tree data structure where the largest value is always stored in the root of the tree.

Use a `Queue` in Java's Library

- Unlike the `Stack` (which is a class), the `Queue` is an interface in Java's library.
- The `Queue` interface is implemented by several classes, including
    - `ArrayDeque`
    - `PriorityQueue`
- Some basic methods in the `Queue` interface:
    - `add(E e)` : Inserts the specified element into this queue
    - `remove()` : retrieves and removes the head of this queue
- Example:

```java
public static void main(String[] args) {
    Queue<Integer> s = new ArrayDeque<>();

    s.add(1);
    System.out.println(s); // [1]
    s.add(2);
    System.out.println(s); // [1, 2]

    System.out.println(s.remove()); // 1
    System.out.println(s); // [2]
}
```