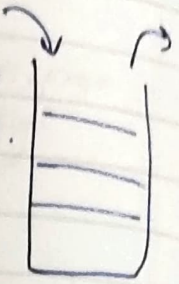
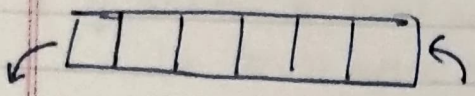
Stacks And Queues

1. Stack is LIFO (or) FILO (Last in first out).
   (first in last out).



2. Insertion in Stack → Push(). & constant time.
   Removing in Stack → Pop().

3. Queue is LILO (or) FIFO (first in first out)



→ Enqueue & Deque.

Stack & Queues in Collections

1. → Stack<Integer> stack1 = new Stack<>();

   → Imp func → push(), pop(), peek(), empty(), search(),
   some methods are inherited from vector* class.

2. Stack is a class, but Queue is an interface.

   Queue<Integer> queue1 = new LinkedList<>();

   Imp operations → peek(), add(), poll(), remove(),
   Methods of Collections Interface are inherited by queue while Implementing.

3. Deque (pronounced deck) (Also an interface) → A doubly ended queue, you can insert & remove from both sides.

   → Deque<Integer> deque = new ArrayDeque(x);

* Like Stack extends vector, deque extends queue.

Note: Read applications of DS you learn in real life & in other DS.

See poll() vs remove().
See add() vs offer().

ArrayDeque → Resizable Array Implementation of Deque.
faster than Linked List class.

Custom Stack Implementation (through Array).

(1)
```
public class CustomStack {
    protected int[] data;
    private static final int DEFAULT_SIZE = 10;

    public CustomStack() {
        this(DEFAULT_SIZE);
    }

    public CustomStack(int size) {
        this.data = new int[size];
    }

    int ptr = -1; // we maintain a pointer which points to the top of the stack.
```

Constructor calling constructor.

if size mention this constructor directly called.

(2) push(x)

Create a error func isFull, if array full.

```
public boolean push(int item) {
    if(isFull()) {
        return false;
    }
    ptr++;
    data[ptr] = item;
    return true;
}

private boolean isFull() {
    return ptr == data.length - 1; // true (or) false.
}

private boolean isEmpty() {
    return ptr == -1;
}
```

pop() fn.

(3)

```
public int pop() throws Exception {
    if (isEmpty()) {
        throw new Exception ("Cannot pop from empty");
    }
    int removed = data[ptr];
    ptr--;
    return removed;
}
```

peek fn.

(4)

```
public int peek() {
    return data[ptr];
}
```

// Exception can be added like in above method if req.

// Can also create custom stack exception class if needed.

---

(5) → In dynamic stack (A stack which never gets full) only thing differed is push() fn.

(6) → to use all func of CustomStack → extends used by Kunal.

to call it's constructor → super keyword is used.

to make the push method diff → @override is used.

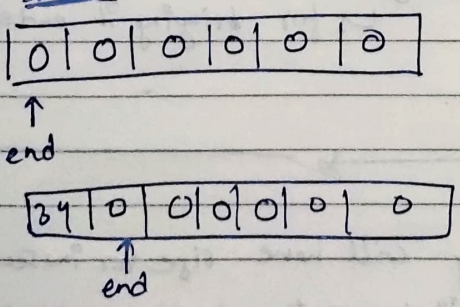
```
@Override
public boolean push(int item) {
    if (this.isFull()) {
        int[] temp = new int[data.length * 2];
        for (int i=0; i<data.length; i++) {
            temp[i] = data[i];
        }
        data = temp;
    }
```

Custom Queue Implementation (using Arrays).

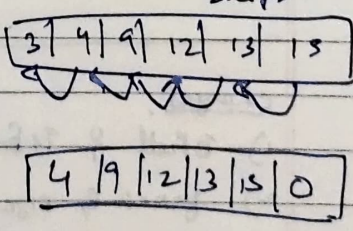
```
public class CustomQueue {
    private int[] data;
    private static final int DEFAULT_SIZE = 10;
    int end = 0;
    public CustomQueue() {
        this(DEFAULT_SIZE);
    }
    public CustomQueue(int size) {
        this.data = new int[size];
    }
    public boolean isFull() {
        return end == data.length;
    }
    public boolean isEmpty() {
        return end == 0;
    }
    public boolean insert(int item) {
        if(isFull()) {
            return false; }
        data[end++] = item;  // Same as (data[end] = item; end++;)
        return true;
    }
```

Exactly Same as Stack.

O(1) Insert



Removing. O(n)



This problem can be solved by circular queue.

```
public int remove() throws Exception {
    if (isEmpty()) {
        throw new Exception("Queue is Empty");
    }
    int removed = data[0];
    for(int i = 1; i < end; i++) {
        data[i-1] = data[i];
    }
    end--;
    return removed; // to show what was removed.
}
}
```