Stack with array

One class needed – Stack – Contains 3 variables

1. Size
2. Array of elements
3. Top

At first, size = 0 & top = -1

isEmpty method

top < 0

isFull method

size == array of elements.length

resize method

1. new variable to save the twice of the current length of the array.
2. new array is defined with the new length variable.
3. copy all the elements of the old array to new array using a for loop.
4. assign new array to old array.

Push method

1. Check if the array is full using isFull method. If true run resize method.
2. If not, save the new data to the next index of the array using top. (Eg : nums[++top]).
3. Size++.

Peek method

1. Check if the array is empty using isEmpty method. If true throw a new EmptyStackException().
2. If not, return the top element in the array (Eg : nums[top]).
3. Size++.

Pop method

1. Check if the array is empty using isEmpty method. If true throw a new EmptyStackException().
2. If not, save the removed data using peek method into a new variable.
3. Decrease the top and size by one (top--, size--).
4. Return the saved removed data.

🚀 Setup

- **Vars**:
  - `int[] data` – the storage array
  - `int top = -1` – index of the current top
  - `int size = 0` – count of elements
- **Law 0**: "Empty toes" → when no elements, top sits at –1.

🔍 Checks & Resize

1. **isEmpty()** → `top < 0`
   Law 1: If top's below zero, stack's flat.
2. **isFull()** → `size == data.length`
   Law 2: Count meets capacity → it's full.
3. **resize()** (when full):
   - Create `newData` of size `data.length × 2`
   - Copy old→new (`for i …`)
   - `data = newData`
     Law 3: Double & copy → keeps amortized O(1).

⭐ Core Ops

Method	Steps	Memory Hint
push(x)	1. if isFull() → resize() 2. `data[++top] = x` 3. `size++`	"Grow, slip on, count"
peek()	1. if isEmpty() → throw `EmptyStackException` 2. `return data[top]`	"Look at top toe"
pop()	1. if isEmpty() → throw 2. `val = peek()` 3. `top--, size–` 4. `return val`	"Un-toe, un-count, hand back"
deleteFirst	(same as pop)	–

insertFirst	(same as push at head)	–

📜 Quick "Stack Laws"

1. **L0**: *initial*: `top=-1, size=0`.
2. **L1**: *empty* ↔ `top<0`.
3. **L2**: *full* ↔ `size==capacity`.
4. **L3**: *push* → if full, resize; then `[++top]`, `size++`.
5. **L4**: *peek* → must not empty; return `data[top]`.
6. **L5**: *pop* → must not empty; peek; `top-`, `size-`.

Stack with Linked List

🚀 Setup

- **Vars**:
  - `Node top = null` – points to the first element (stack's "top")
  - `int size = 0` – number of elements
- **Node**: holds `data` and `next` link
- `private static class Node {`
- `    int data;`
- `    Node next;`
- `    Node(int data) { this.data = data; this.next = null } }`
- **Law 0**: "No head → no stack" (empty when `top == null`).

🔍 Checks

1. **isEmpty()** → `top == null`
   Law 1: If there's no head, the stack is flat.

⭐ Core Ops

Method	Steps	Memory Hint
push(x)	1. Create `newNode = new Node(x)` 2. `newNode.next = top` 3. `top = newNode` 4. `size++`	"Slip a new card on top"
peek()	1. if `isEmpty()` → throw `EmptyStackException` 2. return `top.data`	"Peek at the top card"
pop()	1. if `isEmpty()` → throw 2. `val = top.data` 3. `top = top.next` 4. `size--` 5. return `val`	"Remove the top card and hand it back"

📜 Quick "Stack Laws"

- **L0** *initial*: `top = null`, `size = 0`.
- **L1** *empty*: `top == null`.
- **L2** *push*: link new node at head, `size++`.
- **L3** *peek*: head must exist; read `top.data`.
- **L4** *pop*: head must exist; detach head, `size--`, return value.

Linear Queue with Array

🚀 Setup

- **Vars**
- `int[] data;` `// storage line`
- `int front = 0;` `// index of next element to dequeue`
- `int rear = -1;` `// index of last enqueued element`
- `int size = 0;` `// current number of elements`
- **Law 0**: "No size → no line" (when `size == 0`, queue is empty).

🔍 Checks & Resize

1. **isEmpty()** → `size == 0`
   Law 1: Zero size, zero queue.
2. **isFull()** → `size == data.length`
   Law 2: Size hits capacity → queue is full.
3. **resize()** (when full):

```
int newCap = data.length * 2;
int[] newData = new int[newCap];
for (int i = 0; i < size; i++) {
    newData[i] = data[front + i];
}
data  = newData;
 front = 0;
 rear  = size - 1;
```

   Law 3: Double capacity, line 'em up at front.

⭐ Core Ops

Method	Steps	Memory Hint
enqueue(x)	1. if `isFull()` → `resize()` 2. `data[++rear] = x` 3. `size++`	"New guest joins at tail of line."
peek()	1. if `isEmpty()` → throw `EmptyQueueException` 2. `return data[front]`	"Who's next to be served?"
dequeue()	1. if `isEmpty()` → throw 2. `val = data[front]` 3. `front++`, `size--` 4. **if** `size == 0` → `front = 0; rear = -1;` 5. `return val`	"First person leaves; clear markers if last."

📜 Quick “Queue Laws”

1. **L0** *initial*: `front=0, rear=-1, size=0`.
2. **L1** *empty*: `size == 0`.
3. **L2** *full*: `size == capacity`.
4. **L3** *enqueue*: if full → resize; `rear++`; place element; `size++`.
5. **L4** *peek*: must not empty; read at `front`.
6. **L5** *dequeue*: must not empty; remove at `front`; `front++`; `size--`.
7. **L6** *reset after last dequeue*: if `size == 0` then `front = 0, rear = -1`.

Circular Queue with Array

🚀 Setup

- **Vars**
- `int[] data;` `// storage ring`
- `int front = 0;` `// index of next element to dequeue`
- `int rear = -1;` `// index of last enqueued element`
- `int size = 0;` `// current element count`
- **Law 0**: “Zero size → empty circle” (when `size == 0`, queue is empty).

🔍 Checks & Resize

1. **isEmpty()** → `size == 0`
   Law 1: No riders, no queue.
2. **isFull()** → `size == data.length`
   Law 2: Count hits capacity → ring is full.
3. **resize()** (when full):

```
4. int oldCap = data.length;
5. int newCap = oldCap * 2;
6. int[] newData = new int[newCap];
7. // Unwrap the circle into a straight line
8. for (int i = 0; i < size; i++) {
9.     newData[i] = data[(front + i) % oldCap];
10.  }
11.  data  = newData;
12.  front = 0;
13.  rear  = size - 1;
```

Law 3: Unwind & extend the ring to keep the ride smooth

⭐ Core Ops

Method	Steps	Memory Hint
enqueue(x)	1. if `isFull()` → `resize()` 2. `rear = (rear + 1) % data.length` 3. `data[rear] = x` 4. `size++`	"Hop on the next seat of the carousel."
peek()	1. if `isEmpty()` → throw `EmptyQueueException` 2. `return data[front]`	"Who's at the front of the ride?"
dequeue()	1. if `isEmpty()` → throw 2. `val = data[front]` 3. `front = (front + 1) % data.length` 4. `size--` 5. **if** `size == 0` → `front = 0; rear = -1;` 6. `return val`	"First person leaves; clear markers if last."

📜 Quick "Circular Queue Laws"

1. **L0** *initial*: `front=0, rear=-1, size=0`.
2. **L1** *empty*: `size == 0`.
3. **L2** *full*: `size == capacity`.
4. **L3** *enqueue*: if full → resize; `rear=(rear+1)%cap`; place element; `size++`.
5. **L4** *peek*: must not empty; read at `front`.
6. **L5** *dequeue*: must not empty; remove at `front`; `front=(front+1)%cap`; `size--`.
7. **L6** *reset after last dequeue*: if `size == 0` then `front = 0, rear = -1`.

Linear Queue with Linked List

🚀 Setup

- **Vars**
- `Node front = null;    // index of next element to dequeue (head of list)`
- `Node rear  = null;    // index of last enqueued element (tail of list)`
- `int size   = 0;       // current number of elements`
- **Node** holds `data` and `next` link:
- `private static class Node {`
- `    int data;`
- `    Node next;`
- `    Node(int data) { this.data = data; this.next = null; } }`

- **Law 0**: "No head → no queue" (empty when `front == null`).

🔍 Checks

1. **isEmpty()** → `front == null`
   Law 1: If there's no head, the queue is empty.

⭐ Core Ops

Method	Steps	Memory Hint
enqueue(x)	1. `Node newNode = new Node(x);` 2. **if** `isEmpty()` → `front = newNode;` **else** → `rear.next = newNode;` 3. `rear = newNode;` 4. `size++;`	"New guest lines up at end of the line."
peek()	1. **if** `isEmpty()` → throw `EmptyQueueException` 2. `return front.data;`	"Who's at the front of the line?"

dequeue()	1. **if** `isEmpty()` → throw 2. `int val = front.data;` 3. `front = front.next;` 4. **if** `front == null` → `rear = null;` 5. `size--;` 6. `return val;`	"First person leaves; clear pointers if last."

📜 Quick "Linked-List Queue Laws"

1. **L0** *initial*: `front = null, rear = null, size = 0`.
2. **L1** *empty*: `front == null`.
3. **L2** *enqueue*: new node; if empty → both `front` & `rear` = node; else link at `rear.next`, then `rear = node`; `size++`.
4. **L3** *peek*: must not empty; read `front.data`.
5. **L4** *dequeue*: must not empty; remove head → `front = front.next`; if now empty → `rear = null`; `size--`; return value.

Circular Queue with Linked List

🚀 Setup

- **Vars**
- `Node rear = null;    // tail of the ring; rear.next → head`
- `int size = 0;        // current element count`
- **Node** holds `data` + `next` link:
- `private static class Node {`
- `    int data;`
- `    Node next;`
- `    Node(int data) { this.data = data; }`
- `}`
- `public class CircularQueue {`
- `    Node rear;`
- `    CircularQueue (Node rear) {`
- `        this.rear = rear;`
- `        If (this.rear != null) {`
- `            this.rear.next = this.rear;`
- `}`
-
- **Law 0**: "No rear → empty circle" (when `rear == null`, queue is empty).

🔍 Checks

1. **isEmpty()** → `rear == null`
   Law 1: No tail, no ring.

⭐ Core Ops

Method	Steps	Memory Hint
enqueue(x)	1. `Node newNode = new Node(x);` 2. **if** `isEmpty()`: • `rear = newNode;` • `rear.next = rear;` **else**: • `newNode.next = rear.next;` • `rear.next = newNode;` • `rear = newNode;` 3. `size++;`	"Insert a new link into the ring."
peek()	1. **if** `isEmpty()` → throw `EmptyQueueException` 2. `return rear.next.data;`	"Look at the link right after rear (the head)."
dequeue()	1. **if** `isEmpty()` → throw 2. `Node head = rear.next;` 3. **if** `head == rear` → `rear = null;` **else** → `rear.next = head.next;` 4. `size--;` 5. `return head.data;`	"Break off the head link; if it was the only one, clear rear."

📜 Quick "Circular Linked-List Queue Laws"

1. **L0** *initial*: `rear = null, size = 0`.
2. **L1** *empty*: `rear == null`.
3. **L2** *enqueue*: new node; if empty → node.next→itself, rear=node; else → insert after rear, rear=node; `size++`.
4. **L3** *peek*: must not empty; return `rear.next.data`.
5. **L4** *dequeue*: must not empty; head=rear.next; if head==rear → rear=null; else → rear.next=head.next; `size--`; return value.

Singly Linked List

🚀 Setup

```
Node head = null;     // first element
int size = 0;         // number of nodes
```

Node holds `data` + `next` link:

```
private static class Node {
    int data;
    Node next;
    Node(int data) { this.data = data; }
}
```

Law 0: "No head → empty list" (`head == null`).

🔍 Checks

- **isEmpty()** → `head == null`
  Law 1: If there's no head, the list is empty.

⭐ Core Ops

Method	Steps	Memory Hint

insert(x)	1. `Node n = new Node(x);`
	2. **if** `isEmpty()` → `head = n;`
	else → `Node cur = head;`
	`while(cur.next != null)`
	`cur = cur.next;`
	`cur.next = n;`
	3. `size++;`
		"Walk to the last car and hook on a new one."

insertFirst(x)	1. `Node n = new Node(x);` 2. `n.next = head;` 3. `head = n;` 4. `size++;`	"Slip a new person to the front of the line."

find(x)	1. `Node cur = head;` 2. **while** `cur != null && cur.data != x` → `cur = cur.next;` 3. `return cur;`	"Follow the chain of clues until you spot the treasure."
deleteFirst()	1. **if** `isEmpty()` → throw `NoSuchElementException();` 2. `int v = head.data;` 3. `head = head.next;` 4. `size--;` 5. `return v;`	"Snap off the first sushi roll from the belt."

delete(x)	1. **if** `isEmpty()` → return `false;`
	2. **if** `head.data == x` → `head = head.next; size--; return true;`
	3. `Node prev = head;`
	`while(prev.next != null && prev.next.data != x)`
	`prev = prev.next;`
	4. **if** `prev.next == null` → return `false;`
	5. `prev.next = prev.next.next; size--; return true;`
		"Remove the bad link and reconnect the chain."

📜 Quick "Singly Linked-List Laws"

1. **L0** *initial*: `head = null, size = 0`.
2. **L1** *empty*: `head == null`.
3. **L2** *insert(x)*: append to tail; if empty → `head = n`; else → walk to end; `size++`.
4. **L3** *insertFirst(x)*: new node at front; link it; `size++`.
5. **L4** *find(x)*: traverse `next` pointers from head until match or end.
6. **L5** *deleteFirst()*: remove head; `size--`.
7. **L6** *delete(x)*: bypass target node; `size--`.

Doubly Linked List

🚀 Setup

- **Vars**
- `Node head = null;  // first element`
- `Node tail = null;  // last element`
- `int size = 0;  // number of nodes`

- **Node** holds data + two links:
- `private static class Node {`
- `    int data;`
- `    Node prev, next;`
- `    Node(int data) { this.data = data; }`
- `}`
- **Law 0**: "No head & no tail → empty list" (`head==null`).

🔍 Checks

1. **isEmpty()** → `head == null`
   Law 1: If there's no head, the list is empty.

⭐ Core Ops

Method	Steps	Memory Hint
insertFirst(x)	1. `Node n = new Node(x);` 2. `n.next = head;` 3. **if** empty → `tail = n;` **else** → `head.prev = n;` 4. `head = n;` 5. `size++;`	"Slip a car onto the front coupler."
insertLast(x)	1. `Node n = new Node(x);` 2. `n.prev = tail;` 3. **if** empty → `head = n;` **else** → `tail.next = n;` 4. `tail = n;` 5. `size++;`	"Attach a car onto the back coupler."
find(x)	1. `Node cur = head;` 2. **while** `cur != null && cur.data != x`: `cur = cur.next;` 3. `return cur;` (`null` if not found)	"Walk cars until you spot the right label."
deleteFirst()	1. **if** empty → throw `NoSuchElementException`; 2. `int v = head.data;` 3. `head = head.next;` 4. **if** `head != null` → `head.prev = null;` **else** → `tail = null;` 5. `size--;` 6. `return v;`	"Unhitch the first car."

deleteLast()	1. **if** empty → throw; 2. `int v = tail.data;` 3. `tail = tail.prev;` 4. **if** `tail != null` → `tail.next = null;` **else** → `head = null;` 5. `size--;` 6. `return v;`	"Unhitch the last car."

📜 Quick "Doubly Linked-List Laws"

1. **L0** *initial*: `head = null, tail = null, size = 0`.
2. **L1** *empty*: `head == null`.
3. **L2** *insertFirst*: new node at front; update `head`, link both ways; if first → `tail = head`; `size++`.
4. **L3** *insertLast*: new node at end; update `tail`, link both ways; if first → `head = tail`; `size++`.
5. **L4** *find*: traverse `next` pointers from `head` until match or `null`.
6. **L5** *deleteFirst*/_deleteLast_: detach ends; update `head`/`tail` and their opposite pointers; if becomes empty → clear both; `size--`.

Binary Search Tree

🚀 Setup

```
Node root = null;     // entry point
int size = 0;         // number of nodes
```

Node holds `data` + two links:

```
private static class Node {
    int data;
    Node left, right;
    Node(int data) { this.data = data; }
}
```

Law 0: "No root → empty tree" (`root == null`).

🔍 Checks

- **isEmpty()** → `root == null`
  Law 1: If there's no root, the BST is empty.

⭐ Core Ops

Method	Steps	Memory Hint
insert(x)	1. **if** `isEmpty()` → `root = new Node(x); size++; return;` 2. `Node cur = root, parent = null;` 3. **while** `cur != null`: • `parent = cur;` • **if** `x < cur.data` → `cur = cur.left` **else** → `cur = cur.right` 4. **if** `x < parent.data` → `parent.left = new Node(x)` **else** → `parent.right = new Node(x)` 5. `size++;`	"Plant a seed in the proper branch of the orchard."
find(x)	1. `Node cur = root;` 2. **while** `cur != null` and `cur.data != x`: • **if** `x < cur.data` → `cur = cur.left` **else** → `cur = cur.right` 3. **return** `cur;` (null if not found)	"Binary-search through a sorted library."
delete(x)	1. Locate node `cur` and its `parent` via the same loop as in `find`. 2. **if** `cur == null` → return `false`. 3. **Case A** (leaf): detach from parent. 4. **Case B** (one child): replace `cur` with its single child. 5. **Case C** (two children): find **successor** (`min` in `cur.right`), copy its value to `cur`, then delete the successor node (which is now leaf or single-child).6. `size--; return true;`	"Prune a branch: clip leaf, splice single, or graft successor."

📜 Quick "BST Laws"

1. **L0** *initial*: `root = null, size = 0`.
2. **L1** *empty*: `root == null`.
3. **L2** *insert*: walk from `root`, go left/right, attach at null, `size++`.
4. **L3** *find*: walk from `root`, binary decisions until match or null.
5. **L4** *delete*: handle 3 cases—leaf, single-child, two-child (use successor); `size--`.