MINHEAP.PUSH(int value)
Method Under Test

```
public class MinHeap {
    private PriorityQueue<Integer> heap;

    public MinHeap() {
        heap = new PriorityQueue<>();
    }

    // Method Under Test
    public void push(int value) {       // line 1
        heap.offer(value);                  // line 2
    }
}
```

Conditions

ID	**Goal** (Notes)	**Condition**
(CC1)	Code Coverage	Ensure the method is called at least once with a valid integer.
(B1)	Branch Coverage (implicit)	Not strictly applicable here because push has no if-else branches.
(b1)	Boundary	If needed, test boundary integer values (e.g., Integer.MAX_VALUE).
(c1)	Compound boundary	If pushing multiple values, verify that the heap property is maintained.

Test

Test Condition	Conditions Satisfied	Assertion
1. push normal value	(CC1)	The heap is non-empty afterward.
2. push multiple values	(CC1, c1)	The smallest value should be the root upon pop.
3. push boundary value	(CC1, b1)	The method accepts extreme integer (e.g., MAX_VALUE) without error.

After pushing a single element, isEmpty() returns false.

After pushing multiple elements [8, 3, 7], the first pop is 3.

Bad Data

- *Null integer*: Java generally does not allow null for a primitive int. If your code used Integer objects, you could test pushing null—likely causing an exception.
- *Extreme integer values*: Negative, Integer.MIN_VALUE, Integer.MAX_VALUE.

Good Data

- *Typical usage*: Pushing a few normal integers like 5, 10, 1.

Error Guessing

- *Duplicate values*: Pushing the same value multiple times (5, 5, 5) to see if the heap still orders them properly.
- *Random input sequences*.

Stress Test

- Push a very large number of elements like 1 million integers and ensure the method completes in a reasonable time.
- Optionally measure memory usage or confirm no exceptions are thrown.

MINHEAP.POP()
Method Under Test

```
public class MinHeap {
    private PriorityQueue<Integer> heap;

    // Method Under Test
    public int pop() {                  // line 1
        if (heap.isEmpty()) {      // line 2
            throw new NoSuchElementException("Heap is empty");
        }
        return heap.poll();         // line 3
    }
}
```

Conditions

ID	**Goal** (Notes)	**Condition**
(CC1)	Code Coverage	Must call pop() when heap is non-empty.
(CC2)	Code Coverage / Branch Coverage	Must call pop() when heap is empty to trigger NoSuchElementException.
(b1)	Boundary	Heap has exactly one element—test that pop empties the heap.

(c1)	Compound boundary	If multiple elements exist, pop should always return the smallest.

Test

Test Condition	Conditions Satisfied	Assertion
1. pop from non-empty	(CC1, c1)	Returns smallest element.
2. pop from empty	CC2	Throws NoSuchElementException.
3. pop from single-element	(CC1, b1)	Returns the only element and leaves heap empty.

After pushing [5, 2, 9], consecutive pops return 2, 5, 9.
If the heap is empty, pop() throws NoSuchElementException.

Bad Data

- *Empty Heap*: Attempting pop on an empty heap.
- *Invalid State*: Not particularly applicable if PriorityQueue is well-defined, but could test extreme integer values.

Good Data

- *Normal Sequence*: Push [2, 5, 1], then pop 3 times.

Error Guessing

- *Repeatedly popping until empty*: If you pop exactly the number of pushed elements, the last pop empties the heap.
- *Off-by-one errors* in pop logic (caught by coverage).

Stress Test

- Push 1 million random integers, then pop them all. The output sequence should be in ascending order.
- Confirm no unexpected exceptions and acceptable performance.

MINHEAP.isEmpty()
Method Under Test

```
public class MinHeap {
    private PriorityQueue<Integer> heap;

    // Method Under Test
    public boolean isEmpty() {     // line 1
        return heap.isEmpty();    // line 2
    }
}
```

Condition ID	Goal (Notes)	Condition
(CC1)	Code Coverage	Call isEmpty() on a newly created MinHeap.
(CC2)	Code Coverage	Call isEmpty() after pushing at least one element.
(b1)	Boundary	Heap transitions from non-empty to empty after popping.

Tests (Separate Tests)

Test Condition	**Conditions Satisfied**	Assertion
1. brand new heap	(CC1)	isEmpty() returns true.
2. after push	(CC2)	isEmpty() returns false once an element is pushed.
3. after push & pop	(b1)	isEmpty() returns true after last element is popped.

Immediately after new MinHeap(), isEmpty() == true.
After push(10), isEmpty() == false.
After popping that single element, isEmpty() == true.

Bad Data

- This method doesn't take parameters. "Bad data" is not directly applicable.

Good Data

- *Normal usage*: push a few elements, check isEmpty(), pop them all, check isEmpty() again.

Error Guessing

- *Heap corruption* is mostly outside the scope of isEmpty().
- Possibly check after repeated push/pop cycles.

Stress Test

- Repeatedly push/pop large numbers of elements, calling isEmpty() between operations to ensure consistent results.
-

SCHEDULER.scheduleExecution(int[] permutation, int[] indices)

Method Under Test

```
public class Scheduler {
    private MinHeap minHeap;
    private int curr;

    // Method Under Test
    public List<Integer> scheduleExecution(int[] permutation, int[] indices) {     // line 1
```

```
        List<Integer> result = new ArrayList<>();
        curr = 0;
        for (int queryIndex : indices) {        // line 2
            try {
                while (curr < queryIndex) { // line 3
                    minHeap.push(permutation[curr]);
                    curr++;
                }
                int chosen = minHeap.pop(); // line 4
                result.add(chosen);
            } catch (NoSuchElementException e) {
                throw new IllegalArgumentException(
                    "Not enough elements available for index " + queryIndex);
            }
        }
        return result;
    }
}
```

Conditions

Condition ID	Goal (Notes)	Condition
(CC1)	Code Coverage	Normal case: each query index is within `permutation` bounds, pop returns a valid element.
(CC2)	Branch Coverage	If pop fails (heap empty), catch `NoSuchElementException` and throw `IllegalArgumentException`.
(B1)	Boundary	A query index equals `permutation.length` (fully pushing all elements).
(B2)	Boundary	Repeated indices (e.g., `[2, 2]`), ensuring no new elements get pushed the second time.
(b1)	Out-of-bounds (Bad Data)	`indices` contains a value greater than `permutation.length`, possibly causing

		ArrayIndexOutOfBoundsException.
(c1)	Compound boundary	Mixed queries that cause some pushes and some immediate pops.

Test

Test Condition	Conditions Satisfied	Assertion
1. Normal queries	(CC1, B1, c1)	If indices = [3, 3, 5] and permutation = [4, 2, 5, 1, 3], result = [2, 4, 1].
2. Heap empty triggers exception	(CC2)	If indices requests more pops than elements pushed, throw IllegalArgumentException.
3. Repeated indices	(B2, c1)	If indices = [2, 2] with permutation = [3, 1, 4], second pop might fail or produce next smallest, verifying repeated index boundary.
4. Out-of-bounds index	(b1)	If indices has a value > permutation.length, an ArrayIndexOutOfBoundsException is expected (or a custom error if your code handles differently).

Bad Data

- *indices with negative or out-of-range values* (b1).
- *permutation or indices = null* (if not handled, might throw NullPointerException).
- *Empty permutation* but non-empty indices.

Good Data

- *Typical case*: permutation of moderate size, indices all within range.
- *Minimum normal size*: permutation length 1, indices length 1.

Error Guessing

- *Repeated queries*: Indices with duplicates [2,2,2].
- *Non-sorted indices* (like [2, 5, 3]) if your code depends on them being sorted.

Stress Test

- *Large permutation (e.g., 100,000 or 1 million elements)*, large or random indices.

- Ensure performance is acceptable, no memory issues or timeouts.
- Possibly measure how quickly scheduleExecution completes.