Chapter 6: Input Space Partitioning (ISP)

1. **Refer to Chapter 6.1, problem #3**

Answer the following questions for the method `search()` below:

```
public static int search (List list, Object element)
// Effects: if list or element is null throw NullPointerException
//   else if element is in the list, return an index
//   of element in the list; else return -1
//   for example, search ([3,3,1], 3) = either 0 or 1
//       search ([1,7,5], 2) = -1
```

Base your answer on the following characteristic partitioning:

```
Characteristic: Location of element in list
       Block 1: element is first entry in list
       Block 2: element is last entry in list
       Block 3: element is in some position other than first or last
```

a. "Location of element in list" fails the disjointness property. Give an example that illustrates this.
b. "Location of element in list" fails the completeness property. Give an example that illustrates this.
c. Supply one or more new partitions that capture the intent of "Location of element in list" but do not suffer from completeness or disjointness problems.

Answer:

a. If list has one element i.e. list = [1], then it falls in both Block 1 and 2 which fails the disjointness property.

b. If list passed is null, it will throw NullPointerException in this method. list=null will not fall into any block thus it fails completeness property. NullPointerException is considered as a valid outcome in this method so it should be considered during partition.
Another problem is that element may not be in the list: list = [5; 3]; e = 4

c. Block 1: Whether e is first entry in list: true, false
Block 2: Whether e is last entry in list: true, false
Block 3: Whether e is in list: true, false

2. **Refer to Chapter 6.1, problem #4**

Derive input space partitioning test inputs for the `GenericStack` class with the following method signatures:

- `public GenericStack ();`
- `public void push (Object X);`
- `public Object pop ();`
- `public boolean isEmpty ();`

Assume the usual semantics for the `GenericStack`. Try to keep your partitioning simple and choose a small number of partitions and blocks.

a. List all of the input variables, including the state variables.
b. Define characteristics of the input variables. Make sure you cover all input variables.

c. Partition the characteristics into blocks.
d. Define values for each block.

Answer:

a. Object X, stack(state)

b. **Whether the stack is empty**.

-true (Value stack = [])
-false (Values stack = [“cat"], [“cat", “hat"])

The size of the stack.
-0 (Value stack = [])
-1 (Possible values stack = [“cat"], [null])
-More than 1 (Possible values stack = [“cat", “hat"], [“cat", null], [“cat", “hat",”ox"])

Whether the stack contains null entries
-true (Possible values stack = [null], [null, “cat", null])
-false (Possible values stack = [“cat", “hat"], [“cat", “hat", “ox"])

Whether x is null.
-true (Value x = null)
-false (Possible values x = “cat", “hat", “”)

It can also be interesting to consider a characteristic that involves the combination of Object x and the stack state. For instance,
Does Object x appear in the stack?
-true (Possible values: x = null, stack = ([null, “cat", null]), (x = “cat", stack = [“cat", “hat"]))
-false (Possible values: (x = null, stack = [“cat"]), (x = “cat", stack = [“hat", “ox"]))

3. **Refer to Chapter 6.2, problem #6**

Derive input space partitioning test inputs for the BoundedQueue class with the following signature:

```
public BoundedQueue (int capacity); // The maximum number of elements
public void enQueue (Object X);
public Object deQueue ();
public boolean isEmpty ();
public boolean isFull ();
```

Assume the usual semantics for a queue with a fixed, maximal capacity. Try to keep your partitioning simple - choose a small number of partitions and blocks. (Remember queues work last in last out).

a. List all of the input variables, including the state variables.
b. Define characteristics of the input variables. Make sure you cover all input variables.
c. Partition the characteristics into blocks. Designate one block in each partition as the “Base" block.
d. Define values for each block.
e. Define a test set that satisfies Base Choice Coverage (BCC). Write your tests with the values from the previous step. Be sure to include the test oracles.

Answer:

a. The abstract state variable is “queue” which signifies the queue of objects.
There is another abstract state variable called “cap” to signify queue’s capacity.
We have “capacity” in constructor and a variable “X” in the method Enqueue().
b. C1: Queue is empty
C2: Queue is full

C3: Size of the queue
C4: Value of cap
C5: Value of capacity
C6: Whether variable X is null

c. C1: Whether the queue is empty

a1: True (queue = [])
a2: False (queue = [“a”, “b”]) Base Block

C2: Whether the queue is full

b1: True (queue = [“a”], capacity=1)
b2: False (queue = [“a”, “b”, “c”], capacity=5) Base Block

C3: Size of the queue

c1: 0 (queue = [])
c2: 1 (queue = [“a”])
c3: More than 1 (queue = [“a”, “b”, “c”]) Base Block

C4: Value of cap

d1: Negative (“cap”=-1) -> (may not be possible)
d2: 0 (“cap”=0) -> (may not be possible)
d3: 1 (“cap”=1)
d4: More than 1 (“cap”=2) Base Block

C5: Value of capacity

e1: Negative (“capacity”=-1) -> (may not be possible)
e2: 0 (“capacity”=0) -> (may not be possible)
e3: 1 (“capacity”=1)
e4: More than 1 (“capacity”=2) Base Block

C6: Whether variable X is null

f1: True (X = null)
f2: False (X = ”a”) Base Block

d. Refer solution (c)

e. For this problem we assume we are testing “Enqueue()” as it has input parameter X.

Note that capacity in the constructor is not relevant to this test sets, but the constructor will be called in the setup of the tests.

This means we will skip blocks e1, e2, e3, e4 while defining the base tests for Enqueue()

Base Test = (a2, b2, c3, d4, f2)

Varying every partition provide 8 more tests:

(a1, b2, c3, d4, f2)
(a2, b1, c3, d4, f2)
(a2, b2, c1, d4, f2)
(a2, b2, c2, d4, f2)
(a2, b2, c3, d1, f2)
(a2, b2, c3, d2, f2)

(a2, b2, c3, d3, f2)
(a2, b2, c3, d4, f1)

To check what one of the tests above would look like, take (a2, b2, c3, d3, f2), where we focus on varying d, the cap value.

The queue value specified by C3 is not compatible with this selection.

Either b2 or a2 must change as if cap=1, the queue will always be either full or queue, such as “One”.

For running this test, tester must construct a queue
q = new BoundedQueue(1);

Insert an element
q.Enqueue(“One”);

To get started with the test carry out another call
q.Enqueue(“One”);

Basically, we check that result queue have two “One”.

At this stage, we may decide that having two “One” wasn’t as informative as having, a One and a Two, thereby motivating expansion of possible values for the block f2 to include “Two”.