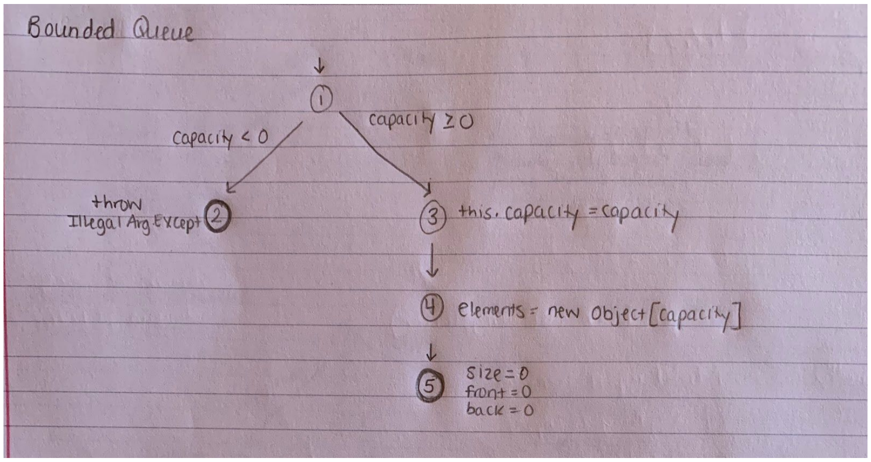
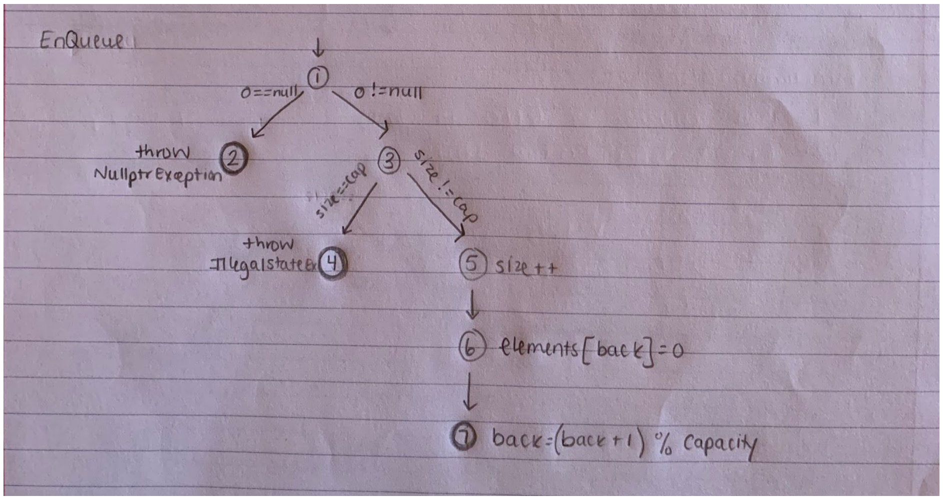
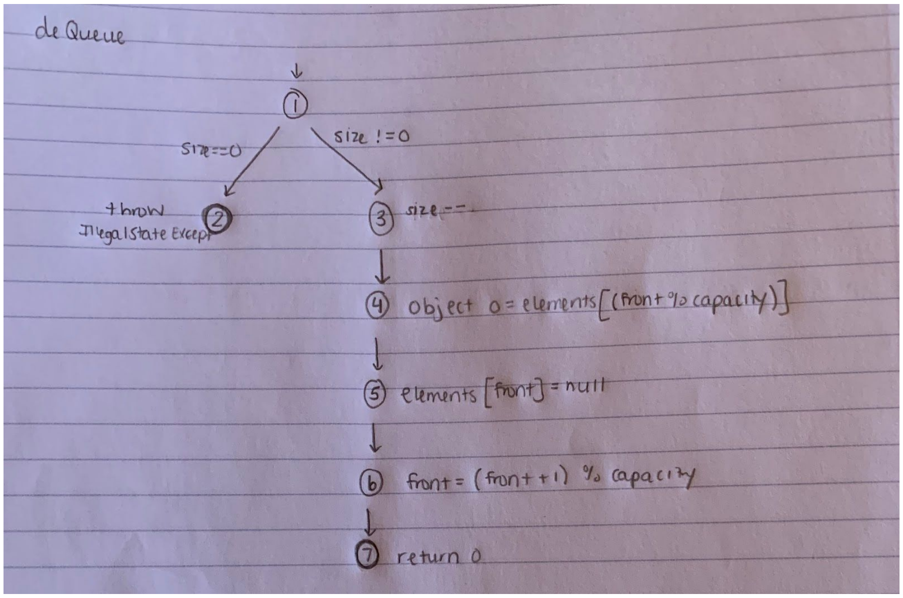
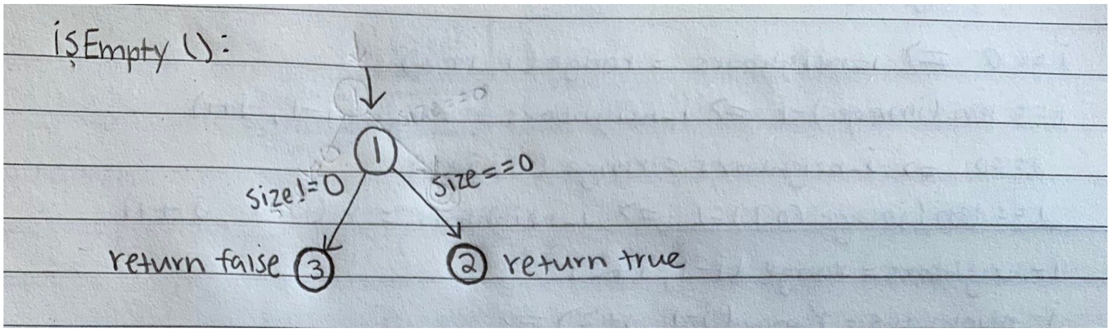
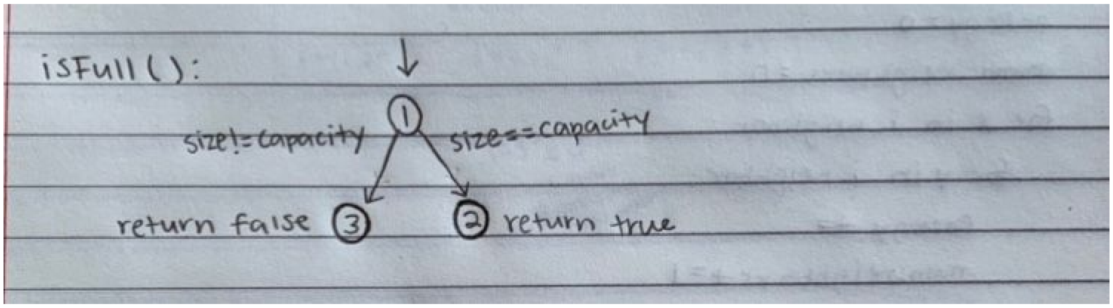
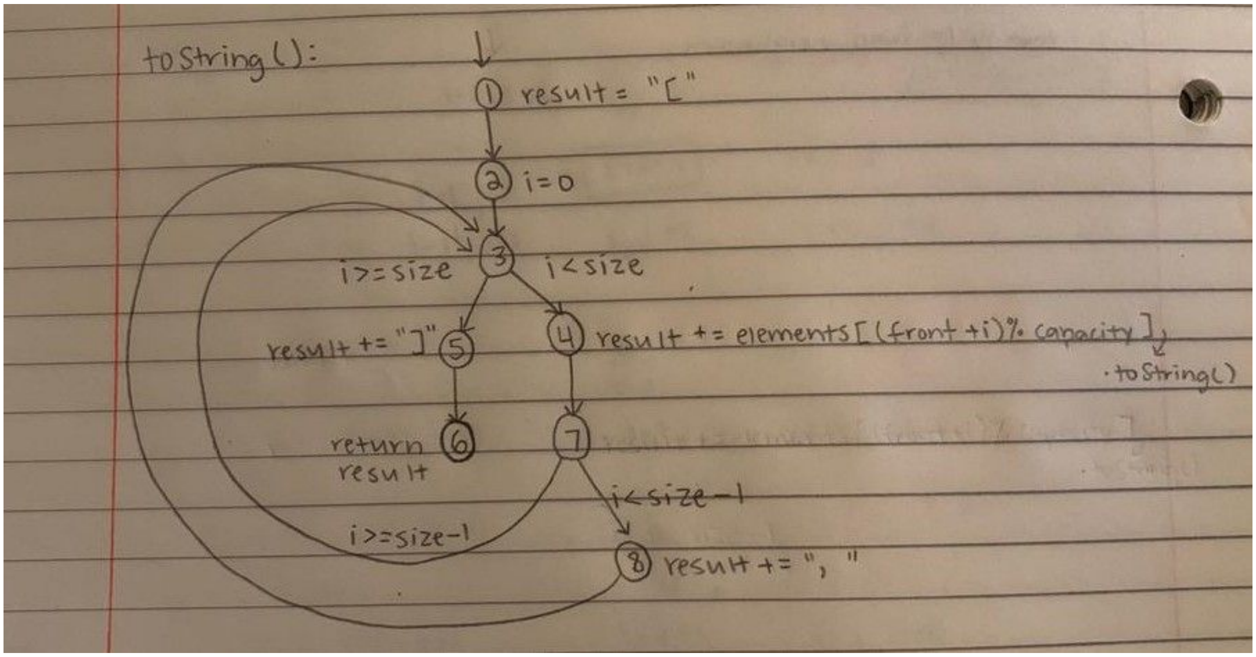

Bounded Queue
1
capacity < 0
capacity ≥ 0
throw
Illegal Arg Except 2
3 this. capacity = capacity
4 elements = new Object[capacity]
5 size = 0
front = 0
back = 0


EnQueue
1
o==null
o != null
throw
Nullptr Exception 2
3
size==cap
size != cap
throw
Illegalstate Ex 4
5 size ++
6 elements[back]=o
7 back=(back+1) % capacity


de Queue
1
size==0
size != 0
throw
Illegal State Except 2
3 size --
4 object o = elements[(front % capacity)]
5 elements[front] = null
6 front = (front + 1) % capacity
7 return o


isEmpty ():
1
size!=0
size==0
return false 3
2 return true


isFull ():
1
size!=capacity
size==capacity
return false 3
2 return true


toString ():
1 result = "["
2 i=0
3
i>=size
i<size
result += "]" 5
4 result += elements[(front+i)% capacity]
.toString()
return 6
result
7
i<size-1
i>=size-1
8 result += ", "

1. **Apply structural-based coverage criteria and design tests → Edge-Pair Coverage**
   1.1 Derive test requirements that satisfy the chosen criterion. Be sure to identify and discuss any infeasible test requirements you may have.
   1.2 Identify test paths that achieve the test requirements.
   1.3 Design test cases for the test paths. Test cases include actual test values/inputs, expected outputs, and any applicable pre-state, post-state, and assumptions.

BoundedQueue
TR: {(1, 2), (1, 3, 4), (3, 4, 5)}
T1 = [1, 2]
T2 = [1, 3, 4, 5]

TC1 = (input: construct BoundedQueue with negative capacity; capacity = -2), Expected output = IllegalArgumentException
TC2 = (input: construct BoundedQueue with positive capacity; capacity = 4), Expected output = BoundedQueue of size=0, front=0, and back=0

enQueue
TR: {(1, 2), (1, 3, 4), (1, 3, 5), (3, 5, 6), (5, 6, 7)}
T1 = [1, 2]
T2 = [1, 3, 4]
T3 = [1, 3, 5, 6, 7]

TC1 = (input: o = null), Expected output = NullPointerException

Pre-state: Bounded queue with a size of 2 and contains values (1, 2)
TC2 = (input:boundedQueue with a size equal to capacity, boundedQueue.enqueue(3)), Expected output = IllegalStateException

Pre-state: Bounded queue with a size of 3 and contains values (1, 2)
TC3 = (input: boundedQueue that is not full; boundedQueue.enqueue(3))
Expected output = boundedQueue containing values: (1, 2, 3)
Post-state: boundedQueue containing values: (1, 2, 3)

deQueue
TR: {(1,2), (1,3,4), (3,4,5), (4,5,6), (5,6,7)}
T1 = [1,2]
T2 = [1,3,4,5,6,7]

TC1 = (input: call deQueue on empty BoundedQueue)
Expected output = IllegalArgumentException

Pre-state: Bounded queue with a size of 3 and contains values (1,2,3)
TC2 = (input: call deQueue on non-empty BoundedQueue and remove oldest element),
Expected output = 1
Post-state: BoundedQueue containing values (2,3) - 2 is at the front and 3 is at the back

isEmpty
TR: {(1, 2), (1, 3)
T1 = [1, 2]
T2 = [1, 3]

TC1 = (input: call isEmpty on an empty BoundedQueue), Expected output = true

Pre-state: BoundedQueue containing values (1, 2)
TC2 = (input: call isEmpty on a BoundedQueue that is not empty)
Expected output = false

isFull
TR: {(1,2), (1,3)}
T1 = [1,2]
T2 = [1,3]

Pre-state: Bounded queue with capacity 0
TC1 = (input: call isFull on a full BoundedQueue)
Expected output = true

Pre-state: Bounded queue with capacity 1
TC2 = (input: call isFull on a non-full BoundedQueue)
Expected output = false

toString
TR: {(1, 2, 3), (2, 3, 4), (2, 3, 5), (3, 4, 7), (3, 5, 6), (4, 7, 3), (4, 7, 8), (7, 3, 4), (7, 3, 5), (7, 8, 3), (8, 3, 5), (8, 3, 4)}

T1 = [1, 2, 3, 5, 6]
T2 = [1, 2, 3, 4, 7, 3, 5, 6]
T3 = [1, 2, 3, 4, 7, 8, 3, 5, 6]

TC1 = (input: call toString on an empty BoundedQueue), Expected output = “[]”

Pre-state: BoundedQueue containing one value: (2)
TC2 = (input: call toString on BoundedQueue with one value), Expected output = “[2]”

Pre-state: Bounded queue containing two values (1,2)

TC3 = (input: call toString on BoundQueue with more than one value), Expected output = “[1, 2]”

2. **Apply data-flow coverage criteria and design tests → All-defs Coverage**
   2.1 Derive test requirements that satisfy the chosen criterion. Be sure to identify and discuss any infeasible test requirements you may have.
   2.2 Identify test paths that achieve the test requirements.
   2.3 Design test cases for the test paths. Test cases include actual test values/inputs, expected outputs, and any applicable pre-state, post-state, and assumptions.

BoundedQueue

Assumption: “capacity” is defined at start

TR capacity: {(1, 3)}

TR size, front, back: {(5)}

T1 = [1, 3, 4, 5] → capacity

T2 = [1, 3, 4, 5] → front

T3 = [1, 3, 4, 5] → back

T4 = [1, 3, 4, 5] → size

Since they are all the same paths, only one test case is needed

TC1 = (input: construct BoundedQueue with input = 1), Expected output = BoundedQueue of size 0

enQueue

Assumptions: “o”, “size”, “elements”, and “back” are defined at the start

TR o: {(1, 2)}

TR size: {(1, 3)}

TR elements: {(1, 3, 5, 6)}

TR back: {(1, 3, 5, 6)}

T1 = [1, 2] → o

T2 = [1, 3, 4] → size

T3 = [1, 3, 5, 6, 7] → elements

T4 = [1, 3, 5, 6, 7] → back

T4 is the same as T3

TC1 = (input: call enQueue on null), Expected output = NullPointerException

TC2 = (input: boundedQueue with a size equal to capacity, boundedQueue.enqueue(3)), Expected output = IllegalStateException

Pre-state: Bounded queue with a size of 3 and contains values (1, 2)
TC3 = (input: boundedQueue that is not full; boundedQueue.enqueue(3))
Expected output = boundedQueue containing values: (1, 2, 3)
Post-state: boundedQueue containing values: (1, 2, 3)

deQueue
Assumptions: “size”, “elements”, “front”, “capacity”, are defined at the beginning
TR size: {(1, 2)}
TR elements: {(1, 3, 4)}
TR front: {(1, 3, 4)}
TR capacity: {(1, 3, 4)}
TR o: {(4,5,6,7)}

T1 = [1, 2] → size
T2 = [1, 3, 4, 5, 6, 7] → elements
T3 = [1, 3, 4, 5, 6, 7] → front
T4 = [1, 3, 4, 5, 6, 7] → capacity
T5 = [1, 3, 4, 5, 6, 7] → o

T2-T5 are repeated
TC1 = (input: call deQueue on empty BoundedQueue)
Expected output = IllegalStateException

Pre-state: Bounded queue with a size of 3 and contains values (1,2,3)
TC2 = (input: call deQueue on non-empty BoundedQueue and remove oldest element),
Expected output = 1
Post-state: BoundedQueue containing values (2,3) - 2 is at the front and 3 is at the back

isEmpty
Assumption: “size” is defined at start
TR size: {(1)}
T1 = [1, 2] → size
TC1 = (input: call isEmpty on an empty BoundedQueue (no elements), Expected output = true

isFull
Assumption: “size” is defined at start
TR size: {(1)}
T1 = [1,2] → size
Pre-state: Bounded queue with capacity 0

TC1 = (input: call isFull on a full BoundedQueue), Expected output = true

toString

Assumption: “size” is defined at the start

TR i: {(2,3)}

TR result: {(1,2,3,5), (5,6)}

TR size: {(1,2,3)}

T1 = [1,2,3,4,7,8,3,5,6] → i

T2 = [1,2,3,5,6] → result

T3 = [1,2,3,5,6] = → size

TC1 = (input: call toString on an empty BoundedQueue), Expected output = “[]”

Pre-state: Bounded queue containing two values (1,2)

TC2 = (input: call toString on BoundQueue with more than one value), Expected output = “[1, 2]”

3. Apply data-flow coverage criteria and design tests
   2.1 Derive test requirements that satisfy the chosen criterion. Be sure to identify and discuss any infeasible test requirements you may have.

   2.2 Identify test paths that achieve the test requirements.
   2.3 Design test cases for the test paths. Test cases include actual test values/inputs, expected outputs, and any applicable pre-state, post-state, and assumptions.
4.