1)

```
BBQ::insert(int item)
{
     lock.acquire();
     while ( (nextEmpty – front) == MAX)
     {
          //Queue is full, wait for another thread to remove an item
          itemRemoved.wait(&lock);
     }
     items[nextEmpty % MAX] = item;
     nextEmpty++;
     itemAdded.signal();
     lock.release();
}

int BBQ::remove()
{
     int item;
     lock.acquire();
     while ( (front == nextEmpty)
     {
          //Nothing to remove from queue, wait for another thread to
          //add an item
          itemAdded.wait(&lock);
     }
     item = items[front % MAX];
     front++;
     itemRemoved.signal();
     lock.release();
     return item;
}
```

This does not guarantee free starvation because it can leave one of the threads to starve (to wait forever).
One example:
Let's say thread T1 is performing BBQ.insert and thread T2 is performing BBQ.remove. There can be a chance where T2 has already finished performing all removal processes and T2 has yet to finish insert process. But the BBQ list can be full and doesn't allow anymore insertion. Then T1 will forever wait for T2 to remove some items from the list but T2 has already finished and will not run again. Therefore T1 will starve forever. So to look at it sequentially:

```
T2.exit();  //Scheduler has removed T2
BBQ List [5, 1, 9, 8, 3];  //List is full
/*T1 tries to perform BBQ.insert*/
/*But list is full, so T1 is put on wait*/
```

```
itemRemove.wait(&T1.lock);
/*T1 will now wait for T2 to remove an item from the list*/
/*BUT T2 has already called exit*/
```

Vice-versa, T2 can also starve and wait for T1 to add an item into the list.

2)
For the Readers/Writers lock example, we use writeGo.signal() to release one thread from the condition variable. Technically, we can use writeGo.broadcast() to release all threads from the condition variable, BUT at most only 1 thread can be waiting to write according to the example. Thus, by calling writeGo.signal(), it'll be more efficient.

3)

It will create 3 queues. Then it will have some threads to insert some values into the queues. putSome has a loop to insert 50 items, so:

```
Q1 = {0, 1, 2, …, 49, 50}
Q2 = {0, 1, 2, …, 49, 50}
Q3 = {0, 1, 2, …, 49, 50}
```

Next, it runs testRemoval for all 3 queues. testRemoval has a loop to remove 20 items. Therefore, the precise output of the code will be:

```
Queue 0:
Removed 0
Removed 1
Removed 2
…
…
Removed 19
Queue 1:
Removed 0
Removed 1
Removed 2
…
…
Removed 18
Removed 19
Queue 2:
Removed 0
Removed 1
Removed 2
…
…
Removed 18
Removed 19
```