Key Difference between Manual Testing and Model Checking

- Manual testing (unit testing)
  - A user should test one concrete execution scenario by checking a pair of concrete input values and the expected concrete output values
- Model checking (concolic testing)
  - A user should imagine all possible execution scenarios and model a general environment that can enable all possible executions
  - A user should describe general invariants on input values and output values

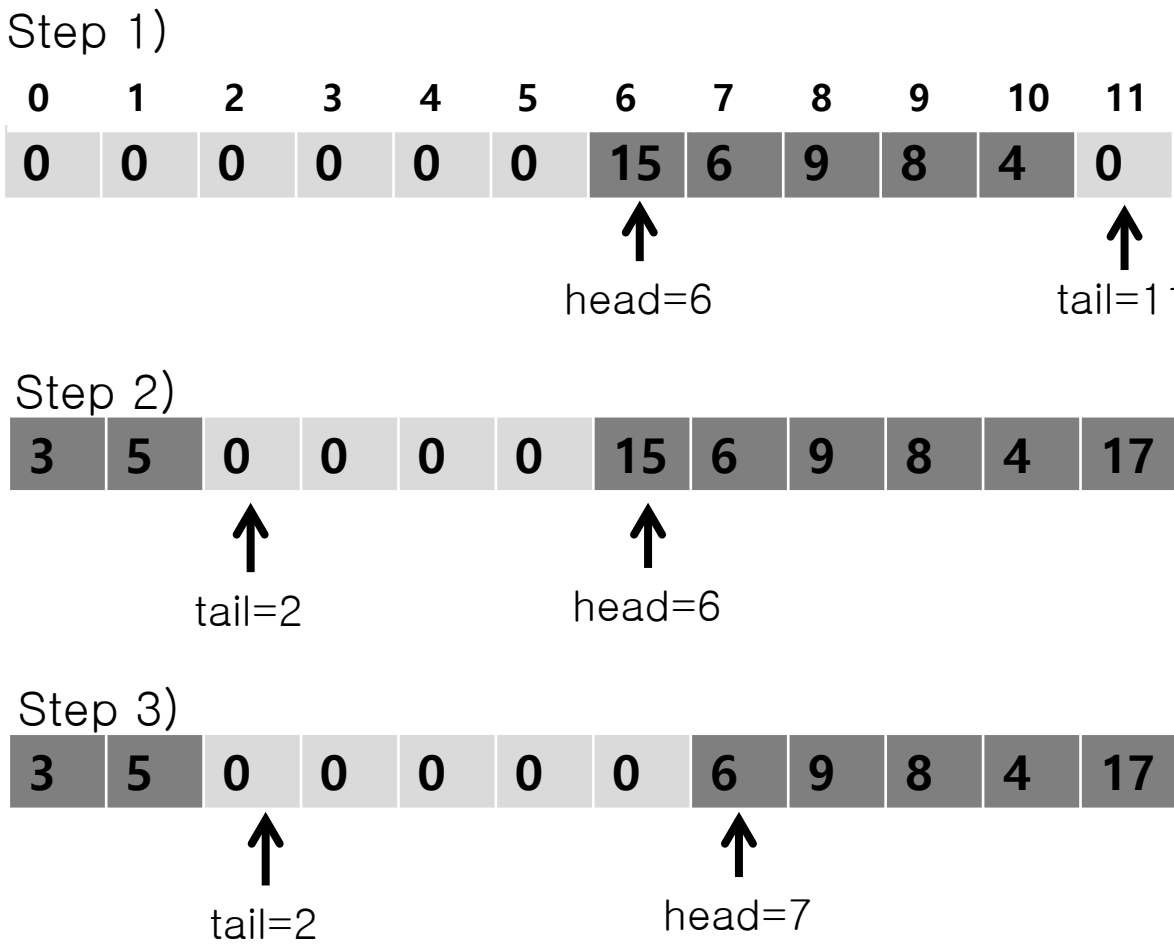
Ex1. Circular Queue of Positive Integers

```
#include<stdio.h>
#define SIZE 12
#define EMPTY 0

// We assume that q[] is
// empty if head==tail
unsigned int q[SIZE],head,tail;

void enqueue(unsigned int x)
{
    q[tail]=x;
    tail=(++tail)%SIZE;
}

unsigned int dequeue() {
    unsigned int ret;
    ret = q[head];
    q[head]=0;
    head= (++head)%SIZE;
    return ret;}
```



```
void enqueue_verify() {
    unsigned int x, old_head, old_tail;
    unsigned int old_q[SIZE], i;
    __CPROVER_assume(x>0);

    for(i=0; i < SIZE; i++) old_q[i]=q[i];
    old_head=head;
    old_tail=tail;

    enqueue(x);

    assert(q[old_tail]==x);
    assert(tail== ((old_tail +1) % SIZE));
    assert(head==old_head);
    for(i=0; i < old_tail; i++)
            assert(old_q[i]==q[i]);
    for(i=old_tail+1; i < SIZE; i++)
            assert(old_q[i]==q[i]);
}
```

```
int main() {// cbmc q.c –unwind
SIZE+2
    environment_setup();
    enqueue_verify();}
```

```
void dequeue_verify() {
    unsigned int ret, old_head, old_tail;
    unsigned int old_q[SIZE], i;

    for(i=0; i < SIZE; i++) old_q[i]=q[i];
    old_head=head;
    old_tail=tail;
    __CPROVER_assume(head!=tail);

    ret=dequeue();

    assert(ret==old_q[old_head]);
    assert(q[old_head]== EMPTY);
    assert(head==(old_head+1)%SIZE);
    assert(tail==old_tail);
    for(i=0; i < old_head; i++)
            assert(old_q[i]==q[i]);
    for(i=old_head+1; i < SIZE; i++)
            assert(old_q[i]==q[i]);}
```

```
int main() {// cbmc q.c –unwind
SIZE+2
    environment_setup();
    dequeue_verify();}
```

```
#include<stdio.h>
#define SIZE 12
#define EMPTY 0

unsigned int q[SIZE],head,tail;

void enqueue(unsigned int x)
{
    q[tail]=x;
    tail=(++tail)%SIZE;
}

unsigned int dequeue() {
    unsigned int ret;
    ret = q[head];
    q[head]=0;
    head= (++head)%SIZE;
    return ret;
}
```

```
// Initial random queue setting following the script
void environment_setup() {
    int i;
    for(i=0;i<SIZE;i++) { q[i]=EMPTY;}

    head=non_det();
    __CPROVER_assume(0<= head && head < SIZE);

    tail=non_det();
    __CPROVER_assume(0<= tail && tail < SIZE);

    if( head < tail)
        for(i=head; i < tail; i++) {
            q[i]=non_det();
            __CPROVER_assume(0< q[i]);
        }
    else if(head > tail) {
        for(i=0; i < tail; i++) {
            q[i]=non_det();
            __CPROVER_assume(0< q[i]);
        }
        for(i=head; i < SIZE; i++) {
            q[i]=non_det();
            __CPROVER_assume(0< q[i]);
        }
    } // We assume that q[] is empty if head==tail
}
```

Model checking v.s. random sequence of method calls

- You may try to test the circular queue code by calling enqueue and dequeuer randomly
  - Ex.

```
void test1(){e(10);r=d();assert(r==10);)
void test2(){e(10);e(20);d();e(30);r=d();
   assert(r =20);)
void test3(){...} …
```

- Note that model checking covers all test scenarios of the above random method sequence calls
  - Note that a random sequence of method calls just provide ONE input instance/state to the circular queue
  - MC provides ALL input instances/states through environment/input space modeling

Ex2. Tower of Hanio

Write down a C program to solve the Tower of Hanoi game (3 poles and 4 disks) by using CBMC

- Hint: you may non-deterministically select the disk to move
- Find the shortest solution by analyzing counter examples. Also explain why your solution is the shortest one.
  - Use non-determinism and `__CPROVER_assume()` properly for the moving choice
  - Use `assert` statement to detect when all the disks are moved to the destination

Top[3]

2	-1	-1

	0	1	2
2	1	0	0
1	2	0	0
0	3	0	0

```
// cbmc hanoi3.c –unwind 7 –no-unwinding-assertions
// Increase n from 1 to 10 in –unwind [n] to find the shortest solution
1:signed char disk[3][3] = {{3,2,1},{0,0,0},{0,0,0}};
2:char top[3]={2,-1,-1};// The position where the top disk is located at.
3:                          // If the pole does not have any disk, top is -1
4:int main() {
5:  unsigned char dest, src;
14:     while(1) {
15:     src = non_det();
16:     __CPROVER_assume(src==0 || src==1 || src==2);
17:     __CPROVER_assume(top[src] != -1);
18:
19:     dest= non_det();
20:     __CPROVER_assume((dest==0 || dest==1 || dest==2) && (dest != src));
21:     __CPROVER_assume(top[dest]==-1 || (disk[src][top[src]] < disk[dest][top[dest]]) );
22:
25:     top[dest]++;
26:     disk[dest][top[dest]]=disk[src][top[src]];
27:
28:     disk[src][top[src]]=0;
29:     top[src]--;
30:
31:     assert( !(disk[2][0]==3 &&  disk[2][1]==2 &&  disk[2][2]==1 ));
} }
```

	0	1	2
2	0	0	1
1	0	0	2
0	0	0	3