**CacheEntry.access**

Inputs: *count*, *accesses, time*

Equivalence classes:

*count*: <0, 0, 1, 9, 10, INT\_MAX

*accesses(time)*: 0 access, 1 accesses, 9 accesses, 10 accesses, > 10accesses

*time*: <0, >= 0

Test cases:

(1). count < 0, 0 access, time < 0.

(2). count = INT\_MAX, 0 access, time >= 0.

(2). count = 0, 0 access, time >= 0.

(3). count = 1, 1 access, time >= 0.

(4). count = 9, 9 accesses, time >= 0.

(5). count = 10, 10 accesses, time >= 0.

(6). count = 12, 12 accesses, time >= 0.

Bugs:

1. Allows variable *count* to be negative, no IllegalStateException thrown.
2. Allows variable *time* to be negative, no IllegalArgumentException thrown.
3. count may overflow.

**Cache.remove**:

Input: *key*, *entries*

Equivalence classes:

*key*: null, existing key, non-existing key

*entries*: 0 entry, 1 entry, someSize entries, fullSize -1 entries, fullSize entries.

Test cases:

(1). Key = null, 0 entry.

(2). Key = existing key, 1 entry.

(3). Key = existing key, fullSize-1 entries

(4). Key = non-existing key, fullSize entries.

(5). Key = non-existing key, 3 entries.

Bugs:

1. When cacheEntry is not full size, call remove() with a non-existing key will result in getting an NullPointerException.

**Cache.hitRate**

Input: *hits*, *misses*

Equivalence classes:

*Hits*: < 0, 0, > 0 INT\_MAX

*Misses*: <0, 0, >0, INT\_MAX

Test cases:

(1). hits = INT\_MAX, misses = INT\_MAX

(2). hits = 0, misses = 0

(3). hits = -1, misses = 1

(4). hits = 1, misses = -1

(5). hits = 3, misses = 5

(6). hits = 1, misses = 0

Bugs:

1. When cache is just initialized, that is, *hits* is 0 and *misses* is 0, the if we call hitRate() and NaN result is returned.
2. When *hits* is negative and *misses* is positive, no IllegalStateException is thrown.
3. When *misses* is negative and *hits* is positive, no IllegalStateException is thrown.
4. When hits or misses is INT\_MAX, or when hits + misses causes overflow, no IllegalStateException is thrown.
5. When misses and hits are positive, the result of hitRate() is not correct.

**Cache.growCache**

Input: *entries*, *initialSize*

Equivalence classes:

*initialSize*: <0, 0, >0, INT\_MAX

*entries*: 0 entry, some entries, fullSize-1 entries, fullSize entries.

Test cases:

(1). initialSize = -10, 0 entry.

(2). initialSize = 0, 0 entry.

(3). initialSize = 8, 1 entry.

(4). initialSize = INT\_MAX, 7 entry.

(5). initialSize = 8, 8 entry.

(6). initialSize = 8, 7 entry.

Bugs:

1. When initialSize is negative, no appropriate exception message is thrown.
2. When initialSize of cacheEntry is positive, the call of growCache() does not increase the size of cacheEntry array.
3. growCache() does not handle the case of overflow(cacheEntry.length + initialSize).

**Cache.mostPopularKey**:

Input: history

Equivalence classes:

*history*: empty, oneKey, multiKey

Bugs:

**LRUCache.replace**

Input: history

Equivalence classes:

*history*: empty, some entries, fullSize entries (tie or not tie)

Bugs:

1. When cache entries is full, the call of replace() does not return the least resent visited entry correctly

**LFUCache.replace**

Input: history

Equivalence classes:

*history*: empty, some entries, fullSize entries (tie or not tie)

Bugs:

1. When cache entries is full, the call of replace() does not return the least resent visited entry correctly

**RandomCache.replace**

Input: history

Equivalence classes:

*history*: empty, some entries, fullSize entries (tie or not tie)

Get()

Input: key, hits, misses, entries, history

Part2:

Bugs:

1. When call get() with null key, the expected result is false, but a NullPointerException is thrown.
2. When call get() with an existing key, the “key” field in the returned CacheResult is not set to the existing key but is set to null.