**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, one Popular Key, multiple Popular Keys

Test cases:

(1). Empty history.

(2). One popular key = “test”

(3). Two popular key: “test1”, “test2”

(4). First last key the same, others different

Bugs:

1. No bugs are found

**LRUCache.replace**

Input: *entries*

Equivalence classes:

*entries*: empty, 1 entry, some entries, fullSize entries(1. No tie in most recent, 2.ties in most recent)

Test cases:

(1). Empty entry.

(2). one entry.

(3). Some entries.

(4). Full size entries(no tie).

(5). Full size entries(tie).

Bugs:

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

**LFUCache.replace**

Input: *entries*

Equivalence classes:

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

Test cases:

(1). Empty entry.

(2). one entry.

(3). Some entries.

(4). Full size entries(no tie).

1. least frequent entry is the first entry

2. least frequent entry is the last entry

3. least frequent entry is in other slots

(5). Full size entries(tie).

Bugs:

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

**RandomCache.replace**

Input: *entries*

Equivalence classes:

*entries*: empty, one entry, some entries, fullSize entries

Test cases:

(1). Empty entry.

(2). one entry.

(3). Some entries.

(4). Full size entries

Bugs:

1. No bugs are found.

**Cache.get**

Input: *key*, *hits*, *misses*, *entries*, *history*

Equivalence classes:

*entries*: empty, one entry, fullSize-1 entries, fullSize entries

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

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

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

*history*: empty, one entry, some entries

Test cases:

(1).null key, hits = -8, misses = -8, empty history entry, empty cache entries.

(2). Existing key, hits = 0, misses = 0, one history entry, one cache entry.

(3). Existing key, hits = 1, misses = 1, one history entry, full size cache entry.

(4). Existing key, hits = INT\_MAX, misses = INT\_MAX, some history entries, some cache entries.

(5). Non-existing key, hits = 1, misses = 1, some history entries, fullSize-1 cache entries.

Three test method: 1.LRUCache, 2.LFUCache, 3RandomCache

(6). Non-existing key, hits = 1, misses = 1, some history entries, full size cache entry.

Three test method: 1.LRUCache, 2.LFUCache, 3RandomCache

Bugs:

1. When key is null, the return value is not null as specified, also, a NullPointerException is thrown.
2. When hits/misses is INT\_MAX, an overflow will happen after a hit/miss.
3. In test case 4, when cacheEntries is not full and a cache miss happens, the call of get() will cause an NullPointerException. For LRU and LFU cache, there will always be an exception, for Random cache, it depends.
4. In test case 4, when Random cache entries is not full and a cache miss happens, if no exception is thrown, the return result is still wrong: the key is not null as expected.
5. When cache entry is full and a miss happens, the Eviction bugs are the same as recorded above in each cache’s replace method.
6. When cache entry is full and a miss happens, the access time of new added cache entry is not correct for all three caches.