```
from collections import OrderedDict, defaultdict
import heapq
import threading

# Base class for eviction policy
class EvictionPolicy:
    def evict(self, cache):
        raise NotImplementedError

    def access(self, key):
        pass

    def insert(self, key):
        pass

    def remove(self, key):
        pass

# LRU Eviction Policy
class LRU(EvictionPolicy):
    def __init__(self):
        self.cache = OrderedDict()

    def access(self, key):
        self.cache.move_to_end(key)

    def insert(self, key):
        self.cache[key] = None

    def evict(self, cache):
        return self.cache.popitem(last=False)[0]
```

```
    def remove(self, key):
        if key in self.cache:
            del self.cache[key]

# LFU Eviction Policy
class LFU(EvictionPolicy):
    def __init__(self):
        self.freq = defaultdict(int)
        self.heap = []

    def access(self, key):
        self.freq[key] += 1
        heapq.heappush(self.heap, (self.freq[key], key))

    def insert(self, key):
        self.freq[key] += 1
        heapq.heappush(self.heap, (self.freq[key], key))

    def evict(self, cache):
        while self.heap:
            freq, key = heapq.heappop(self.heap)
            if self.freq[key] == freq:
                del self.freq[key]
                return key

    def remove(self, key):
        if key in self.freq:
            del self.freq[key]

# Cache Level Class
```

```
class CacheLevel:
    def __init__(self, size, eviction_policy):
        self.size = size
        self.cache = {}
        self.eviction_policy = eviction_policy

    def get(self, key):
        if key in self.cache:
            self.eviction_policy.access(key)
            return self.cache[key]
        return None

    def put(self, key, value):
        if key not in self.cache and len(self.cache) >= self.size:
            evict_key = self.eviction_policy.evict(self.cache)
            self.cache.pop(evict_key, None)
        self.cache[key] = value
        self.eviction_policy.insert(key)

    def remove(self, key):
        if key in self.cache:
            self.cache.pop(key)
            self.eviction_policy.remove(key)

    def display(self):
        return {k: v for k, v in self.cache.items()}

# Dynamic Multilevel Cache Class
class DynamicMultilevelCache:
    def __init__(self):
        self.levels = []
```

```
    self.lock = threading.Lock()

def add_cache_level(self, size, eviction_policy):
    if eviction_policy == 'LRU':
        policy = LRU()
    elif eviction_policy == 'LFU':
        policy = LFU()
    else:
        raise ValueError("Unsupported eviction policy")

    level = CacheLevel(size, policy)
    self.levels.append(level)

def get(self, key):
    with self.lock:
        for level in self.levels:
            value = level.get(key)
            if value is not None:
                self._promote_data_to_higher_levels(key, value, self.levels.index(level))
                return value

        value = self._fetch_from_memory(key)
        self.put(key, value)
        return value

def put(self, key, value):
    with self.lock:
        if self.levels:
            self.levels[0].put(key, value)

def remove_cache_level(self, index):
```

```
        with self.lock:
            if 0 <= index < len(self.levels):
                self.levels.pop(index)

    def _promote_data_to_higher_levels(self, key, value, start_level):
        for level in range(start_level, 0, -1):
            self.levels[level].remove(key)
            self.levels[level - 1].put(key, value)

    def _fetch_from_memory(self, key):
        return f"Value_for_{key}"

    def display_cache(self):
        with self.lock:
            for i, level in enumerate(self.levels):
                print(f"L{i + 1} Cache: {level.display()}")

# Main class with sample test cases
if __name__ == "__main__":
    cache_system = DynamicMultilevelCache()

    # Adding cache levels with different policies
    cache_system.add_cache_level(3, 'LRU')  # L1 Cache
    cache_system.add_cache_level(2, 'LFU')  # L2 Cache

    # Test case 1: Basic insertion and eviction in L1 cache (LRU)
    cache_system.put("A", "1")
    cache_system.put("B", "2")
    cache_system.put("C", "3")
    cache_system.display_cache()
```

```
    # Test case 2: Accessing "A" should keep it in the cache, "B" should be evicted on the next put
    cache_system.get("A")
    cache_system.put("D", "4")  # This should evict "B" due to LRU policy
    cache_system.display_cache()

    # Test case 3: Access "C" (from L2 to L1 promotion)
    cache_system.get("C")  # This should move "C" back to L1 from L2
    cache_system.display_cache()

    # Test case 4: Inserting more data into L1 and L2, and testing eviction in L2 (LFU)
    cache_system.put("E", "5")  # L1 Cache is full, "A" should be evicted as "D" and "C" are more recently used
    cache_system.get("D")  # Access "D" to increase its frequency
    cache_system.put("F", "6")  # L1 Cache is full, "C" should be evicted, "C" moved to L2, "B" in L2 should be evicted due to LFU
    cache_system.display_cache()

    # Test case 5: Removing a cache level
    cache_system.remove_cache_level(1)  # Removes L2 Cache
    cache_system.display_cache()
```