Problem_1_LRU_Cache

March 11, 2020

Analyze: Problem 1: LRU Cache

Analyze: I need to design the stored elements in some sort of structure because I need to find the least recently used entry. The LinkedList has an order data structure. I need to keep track of what's the next item. so I need to move the used node to the list head. and I'd better used a double linked list to easily move the node. All operations must take O(1) time, So I can't use a while loop to find the node. where the inefficiency is, how can I do something to address that inefficiency and make it more efficient. so I need a cache map to get data immediately.

Subtask: 1. DLinkedList to track the order 2. Cache map to get the data immediately

All operations take O(1) time complexity and there is a Double linked list. so the space complexity is linear O(n)

[107]:
```
class DLinkedNode(object):
    def __init__(self, key=None, value=None):
        self.key = key
        self.value = value
        self.next = None
        self.previous = None
```

[116]:
```
class LRU_Cache(object):

    def __init__(self, capacity):
        # Initialize class variables
        self.capacity = capacity
        self.size = 0
        self.cache = {}

        self.head, self.tail = DLinkedNode(), DLinkedNode()
        self.head.next = self.tail
        self.tail.previous = self.head

    def _add_node(self, node):
        # add new node into DLinkedNode
        node.next = self.head.next
        self.head.next.previous = node

        node.previous = self.head
```

```
        self.head.next = node

    def _remove_node(self, node):
        # remove node from DLinkedNode
        node.next.previous = node.previous
        node.previous.next = node.next

    def _move_to_head(self, node):
        # move node to DLinkedNode's head
        self._remove_node(node)
        self._add_node(node)

    def _pop_tail(self):
        # pop the tial node that removes the least recently used entry
        node = self.tail.previous
        self._remove_node(node)
        return node

    def get(self, key):
        # Retrieve item from provided key. Return -1 if nonexistent.
        node = self.cache.get(key)
        if not node :
            return -1

        self._move_to_head(node)
        return node.value

    def set(self, key, value):
        # Set the value if the key is not present in the cache. If the cache is␣
↪at capacity remove the oldest item.
        node = self.cache.get(key)

        if not node:
            new_node = DLinkedNode(key, value)
            if self.size >= self.capacity:
                # delete the last node of DLinkedList
                del_node = self._pop_tail()
                del self.cache[del_node.key]
                self.size -= 1

            # add the new node
            self._add_node(new_node)
            self.cache[key] = new_node
            self.size += 1

        else:
            # update value by key
```

```
            node.value = value
            self._move_to_head(node)
```

[117]:
```
our_cache = LRU_Cache(5)

our_cache.set(1, 1);
our_cache.set(2, 2);
our_cache.set(3, 3);
our_cache.set(4, 4);
```

[118]:
```
our_cache.get(1)       # returns 1
```

[118]: 1

[119]:
```
our_cache.get(2)       # returns 2
```

[119]: 2

[120]:
```
our_cache.get(9)      # returns -1 because 9 is not present in the cache
```

[120]: -1

[121]:
```
our_cache.set(5, 5)
our_cache.set(6, 6)
```

[122]:
```
our_cache.get(3)      # returns -1 because the cache reached it's capacity and␣
↪3 was the least recently used entry
```

[122]: -1