LRU Cache

Design and implement a data structure for Least Recently Used (LRU) cache. It should support the following operations: `get` and `set`.

`get(key)` - Get the value (will always be positive) of the key if the key exists in the cache, otherwise return -1.
`set(key, value)` - Set or insert the value if the key is not already present. When the cache reached its capacity, it should invalidate the least recently used item before inserting a new item.

Solution 1

The problem can be solved with a hashtable that keeps track of the keys and its values in the double linked list. One interesting property about double linked list is that the node can remove itself without other reference. In addition, it takes constant time to add and remove nodes from the head or tail.

One particularity about the double linked list that I implemented is that I create a pseudo head and tail to mark the boundary, so that we don't need to check the NULL node during the update. This makes the code more concise and clean, and also it is good for the performance as well.

Voila, here is the code.

```
class DLinkedNode {
    int key;
    int value;
    DLinkedNode pre;
    DLinkedNode post;
}

/**
 * Always add the new node right after head;
 */
private void addNode(DLinkedNode node){
    node.pre = head;
    node.post = head.post;

    head.post.pre = node;
    head.post = node;
}

/**
 * Remove an existing node from the linked list.
 */
private void removeNode(DLinkedNode node){
    DLinkedNode pre = node.pre;
    DLinkedNode post = node.post;

    pre.post = post;
    post.pre = pre;
}

/**
 * Move certain node in between to the head.
 */
private void moveToHead(DLinkedNode node){
    this.removeNode(node);
    this.addNode(node);
}

// pop the current tail.
private DLinkedNode popTail(){
    DLinkedNode res = tail.pre;
    this.removeNode(res);
    return res;
    return res;
}

private Hashtable<Integer, DLinkedNode>
    cache = new Hashtable<Integer, DLinkedNode>();
private int count;
private int capacity;
private DLinkedNode head, tail;

public LRUCache(int capacity) {
    this.count = 0;
    this.capacity = capacity;

    head = new DLinkedNode();
    head.pre = null;

    tail = new DLinkedNode();
    tail.post = null;

    head.post = tail;
    tail.pre = head;
}

public int get(int key) {

    DLinkedNode node = cache.get(key);
    if(node == null){
        return -1; // should raise exception here.
    }

    // move the accessed node to the head;
    this.moveToHead(node);

    return node.value;
}


public void set(int key, int value) {
    DLinkedNode node = cache.get(key);

    if(node == null){

        DLinkedNode newNode = new DLinkedNode();
        newNode.key = key;
        newNode.value = value;

        this.cache.put(key, newNode);
        this.addNode(newNode);

        ++count;

        if(count > capacity){
            // pop the tail
            DLinkedNode tail = this.popTail();
            this.cache.remove(tail.key);
            --count;
        }
    } else {
    }else{
        // update the value.
        node.value = value;
        this.moveToHead(node);
    }

}
```

written by liaison original link here

Solution 2

There is a similar example in Java, but I wanted to share my solution using the new C++11 unordered_map and a list. The good thing about lists is that iterators are never invalidated by modifiers (unless erasing the element itself). This way, we can store the iterator to the corresponding LRU queue in the values of the hash map. Since using erase on a list with an iterator takes constant time, all operations of the LRU cache run in constant time.

```cpp
class LRUCache {
public:
    LRUCache(int capacity) : _capacity(capacity) {}

    int get(int key) {
        auto it = cache.find(key);
        if (it == cache.end()) return -1;
        touch(it);
        return it->second.first;
    }

    void set(int key, int value) {
        auto it = cache.find(key);
        if (it != cache.end()) touch(it);
        else {
            if (cache.size() == _capacity) {
                cache.erase(used.back());
                used.pop_back();
            }
            used.push_front(key);
        }
        cache[key] = { value, used.begin() };
    }

private:
    typedef list<int> LI;
    typedef pair<int, LI::iterator> PII;
    typedef unordered_map<int, PII> HIPII;

    void touch(HIPII::iterator it) {
        int key = it->first;
        used.erase(it->second.second);
        used.push_front(key);
        it->second.second = used.begin();
    }

    HIPII cache;
    LI used;
    int _capacity;
};
```

written by afernandez90 original link here

Solution 3

Solution is unusual - combination of 2 data structures - hash map and linked list. Algorithm:

1. hash map holds iterators to linked list
2. linked list holds key and value, key to access hash map items
3. when item is accessed, it's promoted - moved to the tail of the list - O(1) operation
4. when item should be removed, we remove head of the list - O(1) operation
5. when item is not promoted long time, it's moved to the head of the list automatically
6. get() - O(1) performance, set() - O(1) performance

   {

```
class LRUCache{
private:
    struct item_t{
        int key, val;
        item_t(int k, int v) :key(k), val(v){}
    };
    typedef list<item_t> list_t;
    typedef unordered_map<int, list_t::iterator> map_t;

    map_t   m_map;
    list_t  m_list;
    int     m_capacity;
public:
    LRUCache(int capacity) : m_capacity(capacity) {
    }
    int get(int key) {
        map_t::iterator i = m_map.find(key);
        if (i == m_map.end()) return -1;
        m_map[key] = promote(i->second);
        return m_map[key]->val;
    }
    void set(int key, int value) {
        map_t::iterator i = m_map.find(key);
        if (i != m_map.end()){
            m_map[key] = promote(i->second);
            m_map[key]->val = value;
        }
        else {
            if (m_map.size() < m_capacity){
                m_map[key] = m_list.insert(m_list.end(), item_t(key, value))
;
            }
            else {
                m_map.erase(m_list.front().key);
                m_list.pop_front();
                m_map[key] = m_list.insert(m_list.end(), item_t(key, value))
;
            }
        }
    }
    list_t::iterator promote(list_t::iterator i){
        list_t::iterator inew = m_list.insert(m_list.end(), *i);
        m_list.erase(i);
        return inew;
    }
};
```

}

btw LeetCode, it was really hard to insert this code, after pressing {} button, class was improperly formatted. I inserted additional braces around class.

written by yakov.sum original link here