Advanced Problems

Created	@August 21, 2022 8:02 PM
Class	
Type	
Materials	
Reviewed	☐

Cache

Cache is a type of memory storage. Just like RAM or HDD or SSD cache also stores data. The interesting part about cache is that, the cache is one of the fastest memory storage, it is even faster than RAM and SSD. Cache is very expensive when compared to ram and hdd.

Very small amount of cache memory also sometimes cost too much. That’s why we generally got very less cache memory in our systems.

Uses Of Cache:

In any situation when we have to serve a data which is very frequently asked, then in those cases cache can be used. For example: If we load a website again and again on our system, and that website or app doesn’t change often the browser tries to cache it.

If we are building a server and we know that there is certain amount of data that will be very frequently requested by huge number of users, its always wise to keep that in a cache.

Problem:

Because Cache is very small, let’s say the size of cache is 3 units. And let’s say we have to store one by one 4 unit of data,

1,2,3,4 → data to be inserted

→ [, ,] → cache of size 3

Insert - 1 → [1 , ,]

Insert - 2 → [1,2 ,]

Insert - 3 → [1, 2, 3]

Insert - 4 → Now we can’t directly insert 4 because cache is full. In order to store 4 we need to replace 4 with some old data.

Now what data we should replace ? We have 3 numbers, but which one to replace ?

The algorithms that are used to decide what data should be replaced for a cache are called as cache replacement policy algorithms.

Some of them are

- LRU (Least Recently Used)
- LFU (least frequently Used)

LRU →

In least recently used cache replacement policy we try to replace the data which was used way back and is now not recently used.

insert(1) , insert(2), get(1), insert(3), get(3) → so here 2 is LEAST RECENTLY USED

LFU →

In least frequently used cache replacement policy we try to replace the data which is used very less frequently, i.e. every time we use or insert the data, we increase the frequency of access.

insert(1), insert(2), get(1), get(1), insert(3), get(1), get(2) → 3 is LEAST FREQUENTLY USED

Implementing LRU Cache

We need a mechanism so that we can keep a track of which all elements are recently used and which are not recently used.

The one which is not recently used will be replaced.

- get(key) → value // O(1)
- put(key, value) → undefined //O(1)

Can we implement using arrays ?

We can implement the LRU cache using array of pairs, but the problem is if we will just store the data in array how can we find the the LRU ? Let's say whenever we use an element we will transfer it to the end index of the array. So when we have to evict (remove) an element we know the element on the 0th index is the LRU. But removing element at 0th index is O(n) task, and this will also not give us correct ans every time.

[2,3,4,1]

insert(1) insert(2) insert(3) insert(4) get(3) get(2) get(1) insert(5)

Can we implement using stacks or queues ?

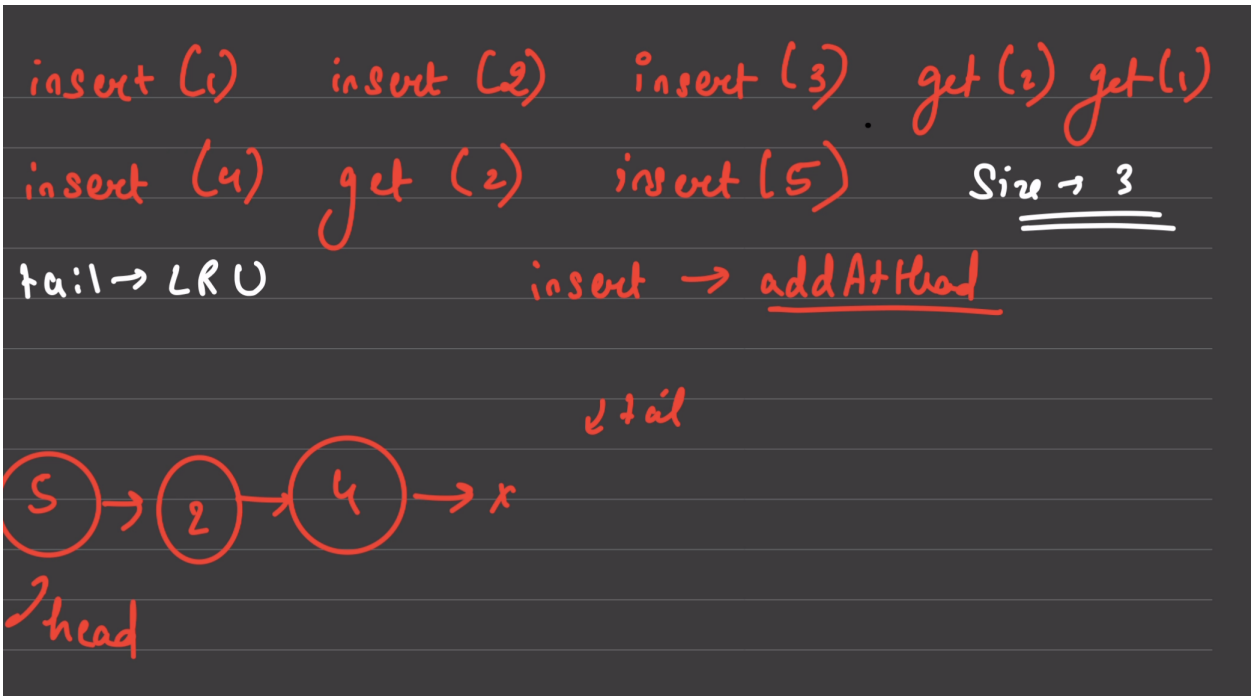
No, because they give us access to top and front only, if the used asks for getting any other element we can't do it.

Can we use object or maps ?

No, because they don't maintain order of insertion always and we can maintain that the element recently used should be at a particular position.

Can we use LL ?

Yes, LL, can be used, because instead of swapping like arrays, whenever we use an element we can remove that element's node from it's original position and push it to the head. When we have to insert a new element and cache is full, tail will be pointing to the least recently used element, so we will do remove at tail.



But there is a problem in this approach,

- To remove the tail or any node in a singly linked list, we need access to the prev element also.
- To search an element in get operation, we are doing whole LL traversal so it is O(n)

How to resolve this ?

To resolve the first problem that in order to remove an element we need access to prev element, how about we keep a pointer to the prev element and instead of using singly linked list, we use doubly linked list ?

And, to resolve the second issue, that we have to search the whole LL, to find an element's node, what we can do is we can prepare a map of <element value → node's reference>

So when someone says get(2), then we will search the map for value 2 as key and in the value of the map we will get reference to the node whose value is 2, now we have access to the node, we can remove it then and there because now we are having DLL, and then insert the node at head.

So here get and insert will work in O(1) because removal in DLL if we have access to the node is O(1) and maintaining a map is also O(1)

```
class Node {
    constructor(data) {
        this.data = data;
        this.next = null;
        this.prev = null;
    }
}

class DoublyLL {
    constructor() {
        this.head = null;
        this.tail = null;
    }

    insertAtHead(data) {
        /**
         * Time: O(1)
         */
        let newNode = new Node(data);
        if(this.head == null) {
            this.head = newNode;
            this.tail = newNode;
            return newNode;
        }
        newNode.next = this.head;
        this.head.prev = newNode;
        this.head = newNode;
        return newNode;
    }

    display() {
        /**
         * Time O(n)
         */
        let temp = this.head;
        while(temp != null) {
            console.log(temp.data);
            temp = temp.next;
        }
    }
}

class LRU {
    constructor(sz) {
        this.dll = new DoublyLL();
        this.mp = {};
        this.sz = sz;
        this.total = 0; // the total number of elements we have
```

```
    }

    put(key, value) {
        if(!this.mp[key]) {
            if(this.total < this.sz) {
                let newNode = this.dll.insertAtHead([key, value]);
                this.total++;
                this.mp[key] = newNode;
            } else {
                let prev = this.dll.tail.prev;
                delete this.mp[this.dll.tail.data[0]]

                prev.next = null;
                this.dll.tail.prev = null;
                this.dll.tail = prev;
                let newNode = this.dll.insertAtHead([key, value]);
                this.mp[key] = newNode;
            }

        } else {
            let keyNode = this.mp[key]; // this is the node where our key value pair is pr
esent
            keyNode.data[1] = value; // on every node's data property we have an array 0th
-> key 1st -> value
            if(keyNode == this.dll.head) return;

            // now shift the node to head

            let prev = keyNode.prev;
            prev.next = keyNode.next; // 1->2->3 // prev -> 1
            if(keyNode.next != null) {
                keyNode.next.prev = prev;
            }
            keyNode.next = this.dll.head;
            this.dll.head.prev = keyNode;
            keyNode.prev = null;
            this.dll.head = keyNode;
            if(keyNode == this.dll.tail) this.dll.tail = prev;
        }
    }

    get(key) {
        if(!this.mp[key]) return undefined;
        let keyNode = this.mp[key];
        const ans = keyNode.data[1];
        if(keyNode == this.dll.head) return ans;
        let prev = keyNode.prev; // 4 -> 3 -> 2 , prev = 3, keynode = 2
        // console.log(prev);
        prev.next = keyNode.next; //
        if(keyNode.next != null) {
            keyNode.next.prev = prev;
        }
        keyNode.next = this.dll.head;
        keyNode.prev = null;
```

```
        this.dll.head = keyNode;
        if(keyNode == this.dll.tail) this.dll.tail = prev;
        return ans;
    }

    display() {
        this.dll.display();
    }

}


let cache = new LRU(3);
cache.put(1,"Sanket");
cache.put(2, "Shaik");
cache.put(3, "Sarthak");
cache.put(4, "JD");
// cache.display();
cache.put(2, "Shaik");
cache.display();
console.log(cache.get(2));
cache.put(5, "Pulkit");
console.log(cache.get(1));
cache.display();
console.log(cache.get(2))
cache.put(6, "Tanuj");
cache.display();
```