SQP8 Implementation LRU Cache

Tuesday, November 29, 2022 1:19 PM

(7) Implementation LRU Cache

LRU Cache is the one of the most Important Algorithm is Operating System that is Know as (Least Recent used)

→ We define the data Structure that follow the principle of LRU

→ In this problem we have tuple of function

1 get()
2 put()

Get()

→ Get function which have to return the value of particular key

↳ if the key is Not present then return (-1)

get(key)

Put()

→ put function which give the key and value and we put into the Cache data Structure

put(key, value)

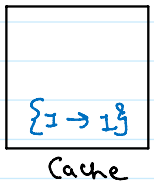
↳ if the capacity of the data Structure is fulled then Return (-1)

(Removed least Recent used guy)
↳ and then we will Insert the Given Pair

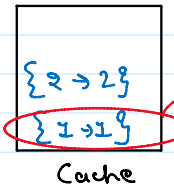
Let's Dry Run

Size = 2 (Size of LRU Cache)

put(1,1)
put this pair into the data Structure Initially



put(2,2)
that Means again put this pair into the Stack because the len of Cache is < 2



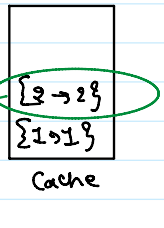
that is my Least Recent used value

get(1)
↳ Now check if the key present into the Cache data Structure

Cache used value

get(2)

→ Now check if the key present into the cache data structure then return its value otherwise return (-1)



ans = 1

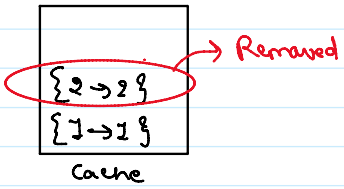
→ because key is present

→ Now after get the answer our Next recently used values changed by upper one pair ↑

put(3,3)

→ Now I can Not put that pair into the cache data structure because the my cache data structure is >= 2 So

→ So in this case we removed the least Recent used data from the Cache data structure

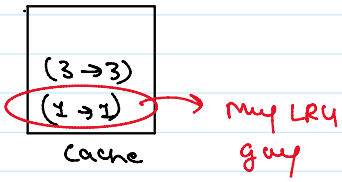


→ Now after Remove LRU Cache Now put the pair into the Cache data structure

put(3,3)

put that pair into the Cache data structure

→ and my LRU guy one changed to the lower one in this case ↓



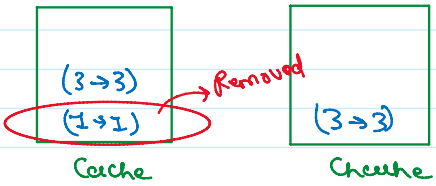
get(2)

→ Now key=2 is Not present into my data structure So Simply return (-1)

put(4,4)

→ In this case again my Cache is fulled So again follow Same process

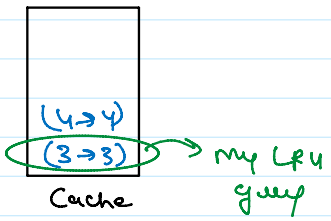
→ Removed the LRU Pair from the our data structure



Now after Removed the LRU Pair from the data Structure put the curr pair into the cache

put(4,4)
→ put the pair into the data structure

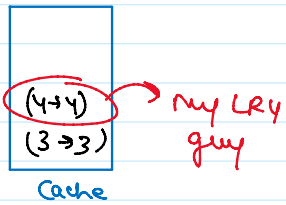
+ Again Same Case my LRU Changed with lower ↓



get(1)
that key=1 Not exist into the data Structure So Simply return -1

get(3)
3 is present into the data Structure So Simply return that value

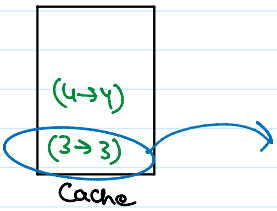
ans = 3
→ again in this case my LRU will be changed with upper one ↑



get(4)
Again follow Same Approach key=4 present into the data Structure So return that value

ans = 4
→ again in this case my LRU changed with upper one ↑

→ In this case Not exist any upper ↑ pair into the data Structure So In this case my LRU is lower ↓ one pair



That is the way how to Implement LRU Cache data Structure using dictionary or hashmap

TimeCompicity = O(N)

→ Because we iterate linearly time

SpaceCompicity = O(N)

→ Because we used the dic and stack data structures

Implementation

```
# lc: https://leetcode.com/problems/lru-cache/
# gfg: https://practice.geeksforgeeks.org/problems/lru-cache/1


class LRUCache:

    def __init__(self, capacity: int):

        self.CacheDS = {}
        self.capacity = capacity

    def get(self, key: int) -> int:

        if key in self.CacheDS:
            val = self.CacheDS.get(key)
            del self.CacheDS[key]
            self.CacheDS[key] = val

            return val

        return -1

    def put(self, key: int, value: int) -> None:

        if key in self.CacheDS:
            del self.CacheDS[key]

        if len(self.CacheDS) < self.capacity:
            self.CacheDS[key] = value

        else:

            last_key = next(iter(self.CacheDS.keys()))
            del self.CacheDS[last_key]

            self.CacheDS[key] = value

# Your LRUCache object will be instantiated and called as such:
# obj = LRUCache(capacity)
# param_1 = obj.get(key)
# obj.put(key,value)
```

→ That section we used O(N) time and space So Interviewer May be Not accepted this solution So Now time to Optimized this Approach

This Solution in Not Optimized Approach

→ Now the User recent task is defined that data

this solution so now time to Optimized this Approach

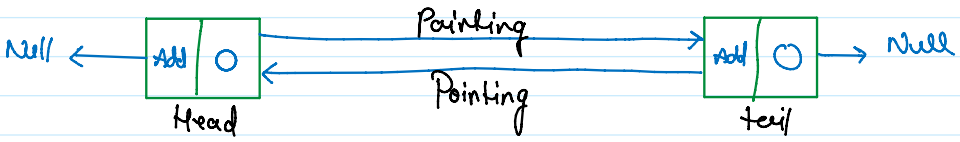
→ Now the User recent task is defined that data structure in the Constant time and Space

This Solution in Not Optimized Approach this code run on O(N) time
→ Might be Interviewer Not accept the Solution ☺

Optimized Approch

→ In Optimized Approch we required two things for Implement get and put function
- → hashmap/dicitioncary
- → Doubly Linked list

→ Now the Initial Configuration of the LinkedList is

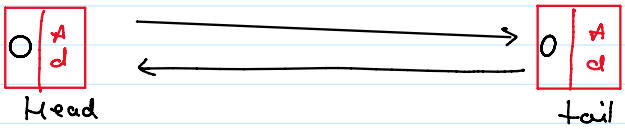


→ Initially our hashmap in Empty

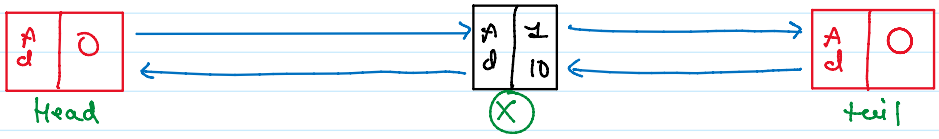
→ Now first we chek the if the key user present or not in Our hashmap and then put that pair into the front of the head

Let's Dary Run and Understand

Size = 3
put (1,10)
- → Now chek if (1) present into the hashmap (X)

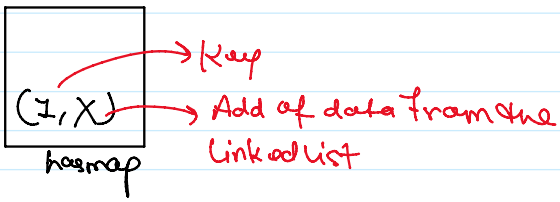


- → that means we visited first time So Simply take that pair and put into the Doubly linked list ☺



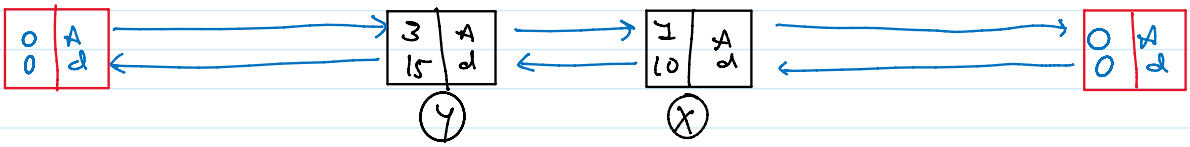
→ Now after put that pair into the doubly linked list put that (pairAdd, value) into the hashmap so we can direct access the pair from the

→ Now after put that pair into the doubly linked list put that (pairAdd, value) into the hashmap so we can direct access the pair from the linkedlist



put(3,15)

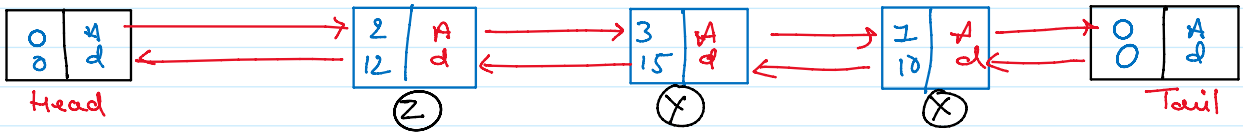
Again follow same Approach and check from the our hashmap
→ that key present or not into our hashmap and as well as put that pair into the linkedlist and hashmap



hashmap = (3, Y) (1, X)

put(2,12)

→ again follow same Approch
→ check key are present into the hashmap or Not
→ check the len(hashmap) < size ✓



hashmap = (2,Z) (3,Y) (1,X)

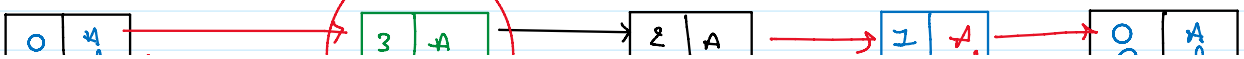
→ that the Approch How to Insert the Node into the doubly linked list

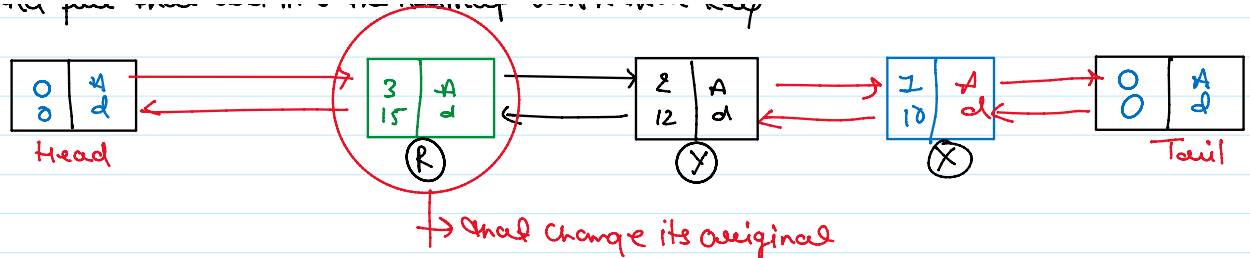
get(3)

Check that key present into the hashmap or Not and then return that Node Value from the linkedlist

ans = 15

→ after return the answer we moved that Node into the front of the head and put that add into the hashmap with that key





→ that change its original position to front of the head as well as change its Address also

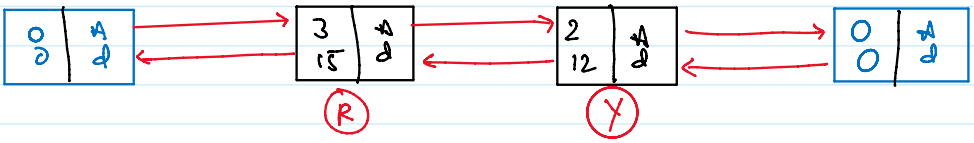
put(4,20)

→ key = 4 Not present into the hashmap but my len(hashmap) > Size

→ that is means remove the Least Recently used data that is Also and always prev Node of the tail Node

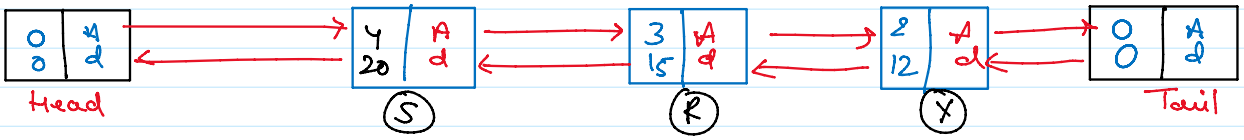
(In front of the tail Node is my Least Recently used Node)

→ So delete that Node from both LinkedList and also from the hashmap



(2,Y)
(3,R)
hashmap

Now after Removed that Node we insert the New Node in front of head and also put that (key, Add) into the hashmap



hashmap = (4,S) (3,Y) (3,R)

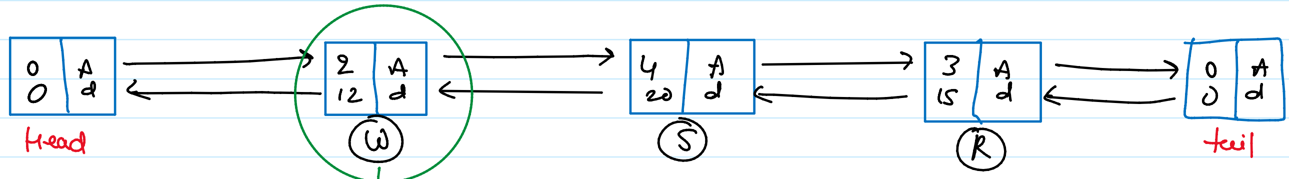
get(2)

Again follow same process and check if the key present into the hashmap so Return the data

→ After return data delete the Node and put Again that Node into the front of head

→ And Again update that address into the hashmap of that key

ans = 12



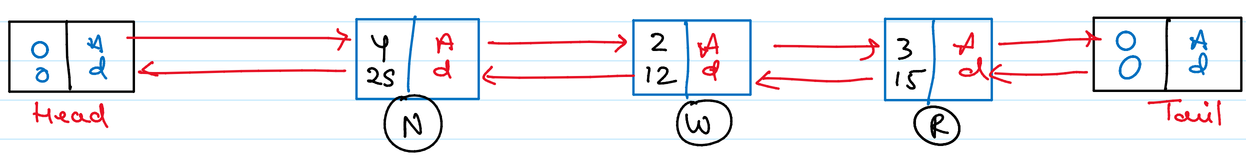
→ that is the Node that we Changed that position and update the Address

hashmap =
(4,S)
(2,W)
(3,R)

put(4,25)
Now In this case (4) already exist into the hashmap that means that Value is defeneitly different

→ first delet that Node from the Linkedlist because this is the Last Recent used Guy

→ and put that Node into the front of the head with New Value and Address



→ Now Update that address into the hashmap

hashmap =
(4,N) → Undated Address
(2,W)
(3,R)

☺

That is the Optimized Approch with the dubly linked List and hasmap

Time Compxity = O(N)
→ In the worst case may be we iterate Linearly

Space Complexity = O(1)
→ because we do not used any furture data Structure for our answer

☺

used any future data structure for our answer ☺

#Implementation Python

```
# lc: https://leetcode.com/problems/lru-cache/
# gfg: https://practice.geeksforgeeks.org/problems/lru-cache/1

# todo optmized code using Dubly LinkedList and hasmap

class Node:
    def __init__(self, key, value):
        self.key = key
        self.val = value
        self.next = None
        self.prev = None


class DoublyList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0

    def insert(self, key, val):
        node = Node(key, val)
        if self.tail != None:
            node.prev = self.tail
            self.tail.next = node
            self.tail = self.tail.next
        else:
            self.tail = self.head = node
        self.size += 1
        return node

    def remove(self, mp):
        if self.head:

            del mp[self.head.key]
            if self.head == self.tail:
                self.tail = None
                self.head = None
            else:
                self.head = self.head.next
            self.size -= 1


    def move_to_front(self, node):
        if not self.head or not node or node == self.tail:
            return
        if self.size == 1:
            return
        if node == self.head:
            self.head = self.head.next
            self.head.prev = None
            self.tail.next = node
            node.prev = self.tail
            self.tail = self.tail.next
            self.tail.next = None
        else:
            node.prev.next = node.next
```

#Implementation Java

```
import java.util.*;
public class L7_LRU_Cache {
    public static void main(String[] args) {
        System.out.println("L6_RU Cache");
    }
}

class Node {
    int key, value;
    Node next, pre;

    Node(int key, int value) {
        this.key = key;
        this.value = value;
        next = pre = null;
    }u
}

class LRUCache
{
    private static Map<Integer, Node> hsMap;
    private static int capacity, count;
    private static Node head, tail;

    //Constructor for initializing the cache capacity with
the given value.
    LRUCache(int cap)
    {
        hsMap = new HashMap<>();
        this.capacity = cap;
        head = new Node(0, 0);
        tail = new Node(0, 0);
        head.next = tail;
        head.pre = null;
        tail.next = null;
        tail.pre = head;
        count = 0;
    }

    public static void addToHead(Node node)
    {
        node.next = head.next;
        node.next.pre = node;
        node.pre = head;
        head.next = node;
    }

    //Function to delete a node.
    public static void deleteNode(Node node)
    {
        node.pre.next = node.next;
        node.next.pre = node.pre;
    }

    //Function to return value corresponding to the
key.
```

```
            self.tail.next = None
        else:
            node.prev.next = node.next
            node.next.prev = node.prev
            node.next = None
            self.tail.next = node
            node.prev = self.tail
            self.tail = self.tail.next


class LRUCache:
    def __init__(self, size):
        self.size = size
        self.list = DoublyList()
        self.hasmap = {}

    def get(self, key):
        if not self.hasmap.get(key):
            return -1
        node = self.hasmap[key]
        self.list.move_to_front(node)
        return node.val

    def put(self, key, val):
        if self.hasmap.get(key):
            self.hasmap[key].val = val
            self.list.move_to_front(self.hasmap[key])
            return
        self.hasmap[key] = self.list.insert(key, val)
        if self.list.size > self.size:
            self.list.remove(self.hasmap)

# Your LRUCache object will be instantiated and called as such:
# obj = LRUCache(capacity)
# param_1 = obj.get(key)
# obj.put(key,value)
```

```
    //Function to return value corresponding to the
key.
    public static int get(int key)
    {
        //if element is present in map,
        if (hsMap.get(key) != null)
        {
            Node node = hsMap.get(key);
            int result = node.value;
            deleteNode(node);
            addToHead(node);

            //returning the value.
            return result;
        }
        //else we return -1.
        return -1;
    }

    //Function for storing key-value pair.
    public static void put(int key, int value)
    {
        if (hsMap.get(key) != null)
        {
            Node node = hsMap.get(key);
            node.value = value;
            deleteNode(node);
            addToHead(node);
        }
        else
        {
            Node node = new Node(key, value);
            hsMap.put(key, node);
            if (count < capacity)
            {
                count++;
                addToHead(node);
            }
            else
            {
                hsMap.remove(tail.pre.key);
                deleteNode(tail.pre);
                addToHead(node);
            }
        }
    }
}

/**
 * Your LRUCache object will be instantiated and
called as such:
 * LRUCache obj = new LRUCache(capacity);
 * int param_1 = obj.get(key);
 * obj.put(key,value);
 */
```