Design a data structure for LRU Cache

Difficulty Level : Medium • Last Updated : 12 Jul, 2021

Design a data structure for 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.

Examples:

```
// Let's say we have a LRU cache of capacity 2.
LRUCache cache = new LRUCache(2);
cache.set(1, 10); // it will store a key (1) with value 10 in the cache.
cache.set(2, 20); // it will store a key (2) with value 20 in the cache.
cache.get(1); // returns 10
cache.set(3, 30); // evicts key 2 and store a key (3) with value 30 in the cache.
cache.get(2); // returns -1 (not found)
cache.set(4, 40); // evicts key 1 and store a key (4) with value 40 in the cache.
cache.get(1); // returns -1 (not found)
cache.get(3); // returns 30
cache.get(4); // returns 40
```

Asked In: Adobe, Hike and many more companies.

Solution:

1. Brute-force Approach:

We will keep an array of Nodes and each node will contain the following information:

C++

```
struct Node
{
    int key;
    int value;

    // It shows the time at which the key is stored.
    // We will use the timeStamp to find out the
    // least recently used (LRU) node.
    int timeStamp;

    Node(int _key, int _value)
    {
        key = _key;
        value = _value;

        // currentTimeStamp from system
        timeStamp = currentTimeStamp;
    }
};

// This code is contributed by subham348
```

Java

```
class Node {
    int key;
    int value;

    // it shows the time at which the key is stored.
    // We will use the timeStamp to find out the
    // least recently used (LRU) node.
    int timeStamp;

    public Node(int key, int value)
    {
        this.key = key;
        this.value = value;

        // currentTimeStamp from system
        this.timeStamp = currentTimeStamp;
    }
}
```

The size of the array will be equal to the given capacity of cache.

(a) For **get(int key):** We can simply iterate over the array and compare the key of each node with the given key and return the value stored in the node for that key. If we don't find any such node, return simply -1.
Time Complexity: **O(n)**

(b) For **set(int key, int value):** If the array if full, we have to delete one node from the array. To find the LRU node, we will iterate through the array and find the node with least timeStamp value. We will simply insert the new node (with new key and value) at the place of the LRU node.
If the array is not full, we can simply insert a new node in the array at the last current index of the array.
Time Complexity: **O(n)**

2. Optimized Approach:
The key to solve this problem is using a double linked list which enables us to quickly move nodes.
The LRU cache is a hash map of keys and double linked nodes. The hash map makes the time of get() to be O(1). The list of double linked nodes make the nodes adding/removal operations O(1).

Code using Doubly Linked List and HashMap:

C++

```
#include <bits/stdc++.h>
using namespace std;

class LRUCache{

    public:
    class node
    {
        public:
        int key;
        int value;
        node * prev;
        node * next;

        node(int _key,int _value)
        {
            key = _key;
            value = _value;
        }
    };

    node* head = new node(-1, -1);
    node* tail = new node(-1, -1);
    int cap;
```

```
    map<int, node *> m;

    // Constructor for initializing the
    // cache capacity with the given value.
    LRUCache(int capacity)
    {
        cap = capacity;
        head->next = tail;
        tail->prev = head;
    }

    void addnode(node * temp)
    {
        node * dummy = head->next;
        head->next = temp;
        temp->prev = head;
        temp->next = dummy;
        dummy->prev = temp;
    }

    void deletenode(node * temp)
    {
        node * delnext = temp->next;
        node * delprev = temp->prev;
        delnext->prev = delprev;
        delprev->next = delnext;
    }

    // This method works in O(1)
    int get(int key)
    {
        if (m.find(key) != m.end())
        {
            node * res =  m[key];
            m.erase(key);
            int ans = res->value;
            deletenode(res);
            addnode(res);
            m[key] = head->next;
           cout << "Got the value : " << ans
                << " for the key: " << key << "\n";
            return ans;
        }
        cout << "Did not get any value for the key: "
             << key << "\n";
        return -1;
    }

    // This method works in O(1)
    void set(int key, int value)
    {

        cout << "Going to set the (key, value) : ("
             << key << ", " << value << ")" << "\n";
        if (m.find(key) != m.end())
        {
            node * exist = m[key];
```

```
            m.erase(key);
            deletenode(exist);
        }

        if (m.size() == cap)
        {
            m.erase(tail->prev->key);
            deletenode(tail->prev);
        }
        addnode(new node(key, value));
        m[key] = head->next;
    }
};

// Driver code
int main()
{
    cout << "Going to test the LRU  "
         << "Cache Implementation\n";

    LRUCache * cache = new LRUCache(2);

    // It will store a key (1) with value
    // 10 in the cache.
    cache->set(1, 10);

    // It will store a key (1) with value 10 in the
    // cache.
    cache->set(2, 20);
    cout << "Value for the key: 1 is "
         << cache->get(1) << "\n"; // returns 10

    // Evicts key 2 and store a key (3) with
    // value 30 in the cache.
    cache->set(3, 30);

    cout << "Value for the key: 2 is "
         << cache->get(2) << "\n"; // returns -1 (not found)

    // Evicts key 1 and store a key (4) with
    // value 40 in the cache.
    cache->set(4, 40);
    cout << "Value for the key: 1 is "
         << cache->get(1) << "\n"; // returns -1 (not found)
    cout << "Value for the key: 3 is "
         << cache->get(3) << "\n"; // returns 30
    cout << "Value for the key: 4 is "
         << cache->get(4) << "\n"; // return 40

    return 0;
}

// This code is contributed by CoderSaty
```

Java

```
import java.util.HashMap;

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

    public Node(int key, int value)
    {
        this.key = key;
        this.value = value;
    }
}

class LRUCache {
    private HashMap<Integer, Node> map;
    private int capacity, count;
    private Node head, tail;

    public LRUCache(int capacity)
    {
        this.capacity = capacity;
        map = new HashMap<>();
        head = new Node(0, 0);
        tail = new Node(0, 0);
        head.next = tail;
        tail.pre = head;
        head.pre = null;
        tail.next = null;
        count = 0;
    }

    public void deleteNode(Node node)
    {
        node.pre.next = node.next;
        node.next.pre = node.pre;
    }

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

    // This method works in O(1)
    public int get(int key)
    {
        if (map.get(key) != null) {
            Node node = map.get(key);
            int result = node.value;
            deleteNode(node);
            addToHead(node);
            System.out.println("Got the value : " + result
```

```
                                + " for the key: " + key);
            return result;
        }
        System.out.println("Did not get any value"
                           + " for the key: " + key);
        return -1;
    }

    // This method works in O(1)
    public void set(int key, int value)
    {
        System.out.println("Going to set the (key, "
                           + "value) : (" + key + ", "
                           + value + ")");
        if (map.get(key) != null) {
            Node node = map.get(key);
            node.value = value;
            deleteNode(node);
            addToHead(node);
        }
        else {
            Node node = new Node(key, value);
            map.put(key, node);
            if (count < capacity) {
                count++;
                addToHead(node);
            }
            else {
                map.remove(tail.pre.key);
                deleteNode(tail.pre);
                addToHead(node);
            }
        }
    }
}

public class TestLRUCache {
    public static void main(String[] args)
    {
        System.out.println("Going to test the LRU "
                           + " Cache Implementation");
        LRUCache cache = new LRUCache(2);

        // it will store a key (1) with value
        // 10 in the cache.
        cache.set(1, 10);

        // it will store a key (1) with value 10 in the
        // cache.
        cache.set(2, 20);
        System.out.println("Value for the key: 1 is "
                           + cache.get(1)); // returns 10

        // evicts key 2 and store a key (3) with
        // value 30 in the cache.
        cache.set(3, 30);
```

```
        System.out.println(
            "Value for the key: 2 is "
            + cache.get(2)); // returns -1 (not found)

        // evicts key 1 and store a key (4) with
        // value 40 in the cache.
        cache.set(4, 40);
        System.out.println(
            "Value for the key: 1 is "
            + cache.get(1)); // returns -1 (not found)
        System.out.println("Value for the key: 3 is "
                           + cache.get(3)); // returns 30
        System.out.println("Value for the key: 4 is "
                           + cache.get(4)); // return 40
    }
}
```

Python3

```
# Class for a Doubly LinkedList Node
class DLLNode:
    def __init__(self, key, val):
        self.val = val
        self.key = key
        self.prev = None
        self.next = None

# LRU cache class
class LRUCache:

    def __init__(self, capacity):
        # capacity:  capacity of cache
        # Initialize all variable
        self.capacity = capacity
        self.map = {}
        self.head = DLLNode(0, 0)
        self.tail = DLLNode(0, 0)
        self.head.next = self.tail
        self.tail.prev = self.head
        self.count = 0

    def deleteNode(self, node):
        node.prev.next = node.next
        node.next.prev = node.prev

    def addToHead(self, node):
        node.next = self.head.next
        node.next.prev = node
        node.prev = self.head
        self.head.next = node

    # This method works in O(1)
    def get(self, key):
        if key in self.map:
```

```
            node = self.map[key]
            result = node.val
            self.deleteNode(node)
            self.addToHead(node)
            print('Got the value : {} for the key: {}'.format(result, key))
            return result
        print('Did not get any value for the key: {}'.format(key))
        return -1

    # This method works in O(1)
    def set(self, key, value):
        print('going to set the (key, value) : ( {}, {})'.format(key, value))
        if key in self.map:
            node = self.map[key]
            node.val = value
            self.deleteNode(node)
            self.addToHead(node)
        else:
            node = DLLNode(key, value)
            self.map[key] = node
            if self.count < self.capacity:
                self.count += 1
                self.addToHead(node)
            else:
                del self.map[self.tail.prev.key]
                self.deleteNode(self.tail.prev)
                self.addToHead(node)


if __name__ == '__main__':
    print('Going to test the LRU Cache Implementation')
    cache = LRUCache(2)

    # it will store a key (1) with value
    # 10 in the cache.
    cache.set(1, 10)

    # it will store a key (1) with value 10 in the cache.
    cache.set(2, 20)
    print('Value for the key: 1 is {}'.format(cache.get(1)))  # returns 10

    # evicts key 2 and store a key (3) with
    # value 30 in the cache.
    cache.set(3, 30)

    print('Value for the key: 2 is {}'.format(
        cache.get(2)))  # returns -1 (not found)

    # evicts key 1 and store a key (4) with
    # value 40 in the cache.
    cache.set(4, 40)
    print('Value for the key: 1 is {}'.format(
        cache.get(1)))  # returns -1 (not found)
    print('Value for the key: 3 is {}'.format(cache.get(3)))  # returns 30
    print('Value for the key: 4 is {}'.format(cache.get(4)))  # returns 40
```

Output:

```
Going to test the LRU  Cache Implementation
Going to set the (key, value) : (1, 10)
Going to set the (key, value) : (2, 20)
Got the value : 10 for the key: 1
Value for the key: 1 is 10
Going to set the (key, value) : (3, 30)
Did not get any value for the key: 2
Value for the key: 2 is -1
Going to set the (key, value) : (4, 40)
Did not get any value for the key: 1
Value for the key: 1 is -1
Got the value : 30 for the key: 3
Value for the key: 3 is 30
Got the value : 40 for the key: 4
Value for the key: 4 is 40
```

Another implementation in Java using LinkedHashMap:

removeEldestEntry() is overridden to impose a policy for removing old mappings when size goes beyond capacity.

Java

```java
import java.util.LinkedHashMap;
import java.util.Map;

class LRUCache {
    private LinkedHashMap<Integer, Integer> map;
    private final int CAPACITY;
    public LRUCache(int capacity)
    {
        CAPACITY = capacity;
        map = new LinkedHashMap<Integer, Integer>(capacity, 0.75f, true) {
            protected boolean removeEldestEntry(Map.Entry eldest)
            {
                return size() > CAPACITY;
            }
        };
    }

    // This method works in O(1)
    public int get(int key)
    {
        System.out.println("Going to get the value " +
                           "for the key : " + key);
```

```
        return map.getOrDefault(key, -1);
    }

    // This method works in O(1)
    public void set(int key, int value)
    {
        System.out.println("Going to set the (key, " +
             "value) : (" + key + ", " + value + ")");
        map.put(key, value);
    }
}

public class TestLRUCacheWithLinkedHashMap {

    public static void main(String[] args)
    {
        System.out.println("Going to test the LRU "+
                           " Cache Implementation");
        LRUCache cache = new LRUCache(2);

        // it will store a key (1) with value
        // 10 in the cache.
        cache.set(1, 10);

        // it will store a key (1) with value 10 in the cache.
        cache.set(2, 20);
        System.out.println("Value for the key: 1 is " +
                           cache.get(1)); // returns 10

        // evicts key 2 and store a key (3) with
        // value 30 in the cache.
        cache.set(3, 30);

        System.out.println("Value for the key: 2 is " +
                cache.get(2)); // returns -1 (not found)

        // evicts key 1 and store a key (4) with
        // value 40 in the cache.
        cache.set(4, 40);
        System.out.println("Value for the key: 1 is " +
               cache.get(1)); // returns -1 (not found)
        System.out.println("Value for the key: 3 is " +
                           cache.get(3)); // returns 30
        System.out.println("Value for the key: 4 is " +
                           cache.get(4)); // return 40

    }
}
```

Output:

```
Going to test the LRU  Cache Implementation
Going to set the (key, value) : (1, 10)
```

```
Going to set the (key, value) : (2, 20)
Going to get the value for the key : 1
Value for the key: 1 is 10
Going to set the (key, value) : (3, 30)
Going to get the value for the key : 2
Value for the key: 2 is -1
Going to set the (key, value) : (4, 40)
Going to get the value for the key : 1
Value for the key: 1 is -1
Going to get the value for the key : 3
Value for the key: 3 is 30
Going to get the value for the key : 4
Value for the key: 4 is 40
```

You can find the C++ code here

Previous Next