LRU caching

RU Cache Implementation In Java

The term LRU Cache stands for **Least Recently Used** Cache. It means LRU cache is the one that was recently least used, and here the cache size or capacity is fixed and allows the user to use both get () and put () methods. When the cache becomes full, via put () operation, it removes the recently used cache.

In this section of Java, we will discuss LRU cache brief introduction, its implementation in Java, and what are the ways through which we can achieve LRU Cache.

What is LRU Cache

Everyone might be aware that cache is a part of computer memory that is used for storing the frequently used data temporarily. But the size of the cache memory is fixed, and there exists the management requirement where it can remove the unwanted data and store new data there. Here, LRU comes into the role. Thus, LRU is a cache replacement algorithm used for freeing the memory space for the new data by removing the least recently used data.

Implementing LRU Cache in Java

For implementing LRU Cache in Java, we have the following two data structures through which we can implement LRU Cache:

Queue: Using a doubly-linked list, one can implement a queue where the max size of the queue will be equal to the cache size (the total number of frames that exist). It is quite simple to find that the most recently used pages are present near the front end, and on the other hand, we can find the least recently used pages near the rear end of the doubly linked list.

Hash: A key here represents a hash with page number, and a value represents the address of the corresponding queue node.

Understanding LRU

Whenever a user references a page, there may be two cases. Either the page may exist within the memory, and if so, just detach the node of the list and bring that page to the front of the queue. Or, if the page is not available (does not exist) in the memory, then it is initially moved in the memory. For it, the user inserts a new node to the front of the queue and, after that, update the address of the corresponding node in the hash. In case if the user determines that the queue is already full (all frames are full), just remove a node from the rear end and, after that, add the new node to the front end of the queue.

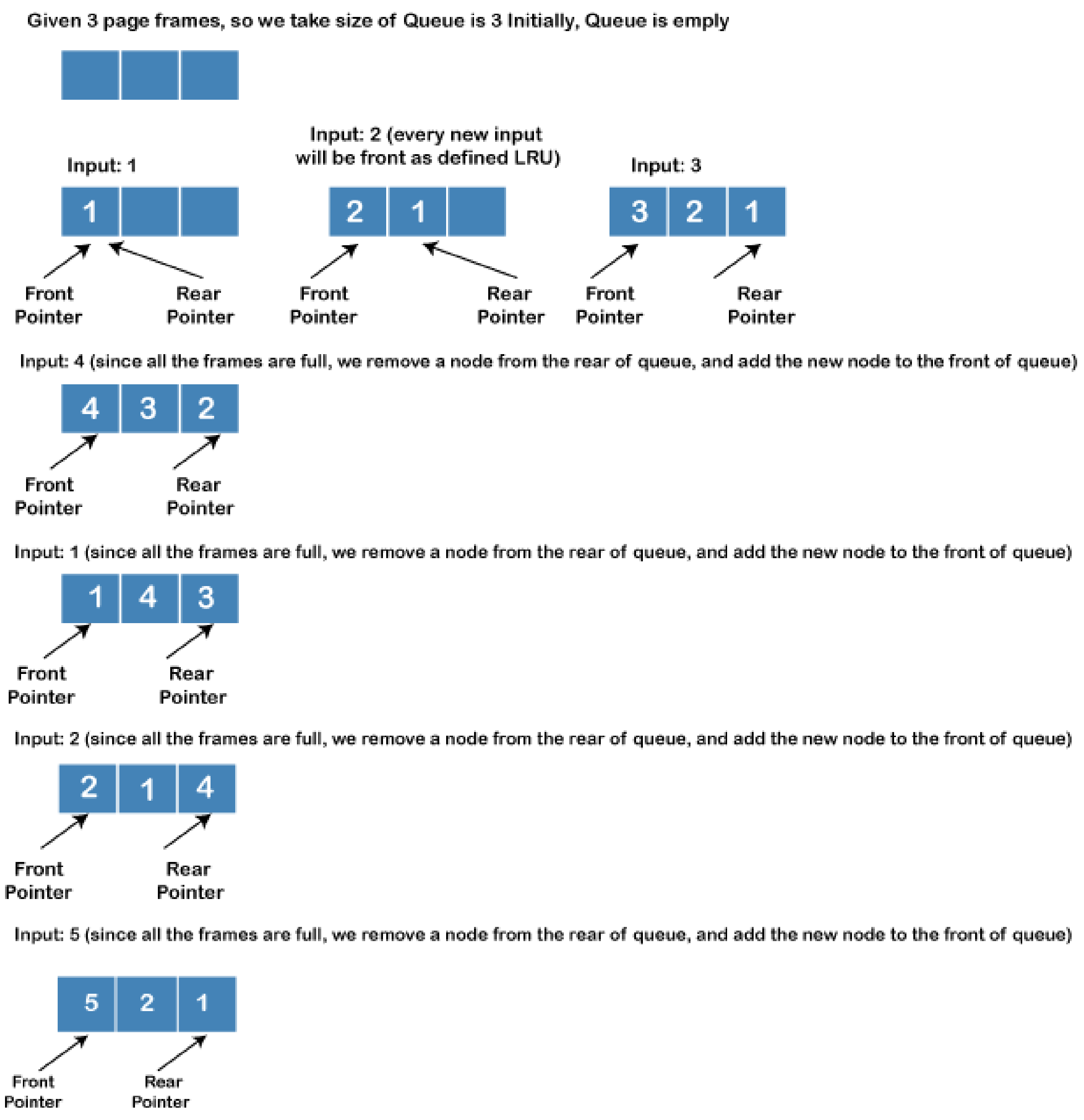
Example of LRU

There is the following given reference string:

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Thus using the LRU page replacement algorithm, one can find the number of page faults where page frames are 3.

In the below-shown diagram, you can see how we have performed the LRU algorithm to find the number of page faults:


Given 3 page frames, so we take size of Queue is 3 Initially, Queue is emply
Input: 1
1
Front Pointer
Rear Pointer
Input: 2 (every new input will be front as defined LRU)
2 1
Front Pointer
Rear Pointer
Input: 3
3 2 1
Front Pointer
Rear Pointer
Input: 4 (since all the frames are full, we remove a node from the rear of queue, and add the new node to the front of queue)
4 3 2
Front Pointer
Rear Pointer
Input: 1 (since all the frames are full, we remove a node from the rear of queue, and add the new node to the front of queue)
1 4 3
Front Pointer
Rear Pointer
Input: 2 (since all the frames are full, we remove a node from the rear of queue, and add the new node to the front of queue)
2 1 4
Front Pointer
Rear Pointer
Input: 5 (since all the frames are full, we remove a node from the rear of queue, and add the new node to the front of queue)
5 2 1
Front Pointer
Rear Pointer

Implementing LRU Cache via Queue

To implement the LRU cache via a queue, we need to make use of the Doubly linked list. Although the code is lengthy enough, it is the basic implementation version for the LRU Cache.

Below is the following code:

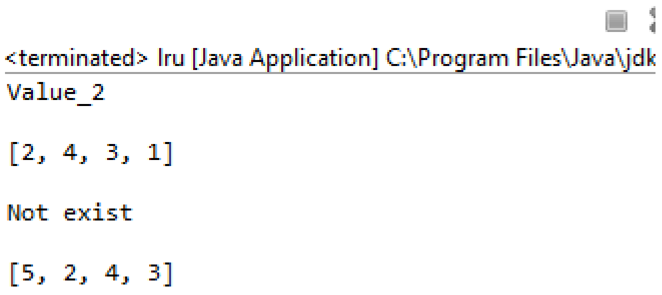
```
import java.util.Deque;
import java.util.HashMap;
import java.util.LinkedList;
import java.util.Map;
 class Cache
{
  int key;
  String value;
 Cache(int key, String value) {
    this.key = key;
    this.value = value;
  }
}
 public class lru {
 static Deque<Integer> q = new LinkedList<>();
  static Map<Integer, Cache> map = new HashMap<>();
  int CACHE_CAPACITY = 4;
  public String getElementFromCache(int key)
  {
   if(map.containsKey(key))
    {
      Cache current = map.get(key);
      q.remove(current.key);
      q.addFirst(current.key);
      return current.value;
    }
  return "Not exist";
  }
 public void putElementInCache(int key, String value)
  {
    if(map.containsKey(key))
    {
      Cache curr = map.get(key);
      q.remove(curr.key);
    }
    else
  {
   if(q.size() == CACHE_CAPACITY)
   {
    int temp = q.removeLast();
    map.remove(temp);
   }
  }
  Cache newItem = new Cache(key, value);
  q.addFirst(newItem.key);
  map.put(key, newItem);
 }
public static void main(String[] args)
 {
  lru cache = new lru();
  cache.putElementInCache(1, "Value_1");
  cache.putElementInCache(2, "Value_2");
  cache.putElementInCache(3, "Value_3");
  cache.putElementInCache(4, "Value_4");
  System.out.println(cache.getElementFromCache(2));
  System.out.println();
  System.out.println(q);
  System.out.println();
  System.out.println(cache.getElementFromCache(5));
  cache.putElementInCache(5,"Value_5");
  System.out.println();
  System.out.println(q);
  System.out.println();
 }
}
```

Code Explanation:

- In the above code, we have imported the inbuilt Deque package and all other collection classes and created a class lru that has implemented the main() method.

- A class Cache is implemented where we have declared two variables (key and value) where the key is for accessing the actual data and value is for cache to access the data.
- Next, by creating a constructor for the Cache class, we have set the values for both variables.
- Moving to the lru class, we declared a queue that acts as a Cache for storing data and a Map to store key-value pair of the data items. A variable CACHE_CAPACITY value is set as 4.
- Next, implemented a method getElementFromCache(int key) where if the key already exists, fetch data from the cache. Inside it, if the item is present in the cache, remove it and add it on the front of the cache. If not, return that no such element exists.
- Now, using the map, we checked if the element already exists in the cache, and for it, another method putELementCache(nt key, String value), has been implemented. Within it, if the element already exists in the cache, remove it. Else, if the size of the cache is full, then remove an element from the last of the queue. After that, just add the already existing element or the new element with the given key and value in the queue.
- In the main() method, we have invoked the methods we have created in the main() method.

Thus, on executing the above code, we got the below-shown output:



```
<terminated> lru [Java Application] C:\Program Files\Java\jdk
Value_2

[2, 4, 3, 1]

Not exist

[5, 2, 4, 3]
```

Implementing LRU Cache using LinkedHashMap

A LinkedHashMap is similar to a HashMap, but in LinkedHashMap, there is a feature that allows the user to maintain the order of elements that are inserted into the

LinkedHashMap. As a result, such a feature of LinkedHashMap makes the LRU cache implementation simple and short. If we implement using HashMap, we will get a different sequence of the elements, and thus, we may further code to arrange those elements but using LinkedHashMap, we do not need to increase more code lines.

Below is the given code that will let you implement LRU cache using LinkedHashMap

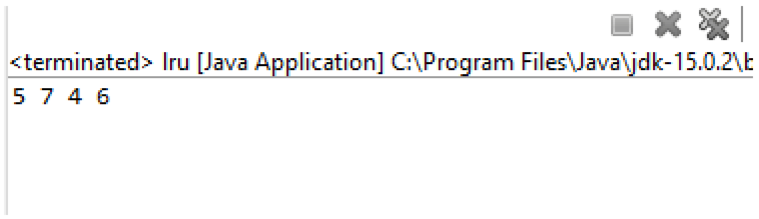
```java
import java.util.*;

class lru {

    Set<Integer> cache;
    int capacity;

    public lru(int capacity)
    {
        this.cache = new LinkedHashSet<Integer>(capacity);
        this.capacity = capacity;
    }
public boolean get(int key)
    {
        if (!cache.contains(key))
            return false;
        cache.remove(key);
        cache.add(key);
        return true;
    }
    public void get_Value(int key)
    {
        if (get(key) == false)
           put(key);
    }

    public void display()
    {
      LinkedList<Integer> list = new LinkedList<>(cache);
      Iterator<Integer> itr = list.descendingIterator();

      while (itr.hasNext())
            System.out.print(itr.next() + " ");
    }
public void put(int key)
    {

      if (cache.size() == capacity) {
            int firstKey = cache.iterator().next();
            cache.remove(firstKey);
        }

        cache.add(key);
    }
     public static void main(String[] args)
    {
        lru obj = new lru(4);
        obj.get_Value(4);
        obj.get_Value(5);
        obj.get_Value(6);
        obj.get_Value(4);
        obj.get_Value(7);
        obj.get_Value(5);
        obj.display();
    }
}
```

Code Explanation:

- In the above code, we have created a class lru and implemented the program using LinkedHashSet.
- We declared a cache and a variable capacity and set their values using the constructor.

- Next, we have created a Boolean return type function get () that returns false if the key is not present in the cache. Otherwise, it moves the key to the front by removing it first. Then adding it and finally returns Boolean true.
- After that created a function get_Value in which if get(key)==false, then put(key).
- Then we created a display () function to display the elements of cache in reverse order.
- Finally, executed the main () method.

When we executed the above code, we got the below-shown output:



```
<terminated> lru [Java Application] C:\Program Files\Java\jdk-15.0.2\b
5 7 4 6
```

It is clear from the output that we got the appropriate result with fewer lines of code in comparison to the above one.

Therefore, in this way, we can implement LRU Cache in Java and represent it.

How did you find this lesson?