**Project Description For Netflix**

Get a brief introduction to "Netflix" and learn which features we'll be building in this project.

**We'll cover the following**

* + [Introduction](https://www.educative.io/courses/decode-coding-interview-java/BnXQrnlAXyk#Introduction)
  + [Statement](https://www.educative.io/courses/decode-coding-interview-java/BnXQrnlAXyk#Statement)
  + [Features](https://www.educative.io/courses/decode-coding-interview-java/BnXQrnlAXyk#Features)

**Introduction**[#](https://www.educative.io/courses/decode-coding-interview-java/BnXQrnlAXyk#Introduction)

*Netflix* is the biggest video streaming platform in the world, offering movies, seasons, biographies, reality shows, and more. Their video repository is growing significantly. So the engineering team at *Netflix* keeps trying to find better ways to display content to their consumers.

The scenario and the problems discussed in this chapter also relate to any content displaying functionality and how we can improve it.

**Statement**[#](https://www.educative.io/courses/decode-coding-interview-java/BnXQrnlAXyk#Statement)

Let’s pretend you’re a developer on the *Netflix* engineering team. You are working on improving the user experience in finding content to watch. This involves the improvement of the search as well as recommendation functionality.

**Features**[#](https://www.educative.io/courses/decode-coding-interview-java/BnXQrnlAXyk#Features)

We will need to introduce the following features to implement the improvements discussed above:

* **Feature # 1:** We want to enable users to see relevant search results despite minor typos.
* **Feature # 2:** Enable the user to view the top-rated movies worldwide, given that we have movie rankings available separately for different geographic regions.
* **Feature # 3:** As part of a demographic study, we are interested in the median age of our viewers. We want to implement a functionality whereby the median age can be updated efficiently whenever a new user signs up for Netflix.
* **Feature # 4:** For efficiently distributing content to different geographic regions and for program recommendation to viewers, we want to determine titles that are gaining or losing popularity scores.
* **Feature # 5:** For the client application, we want to implement a cache with a replacement strategy that replaces the least recently watched title.
* **Feature # 6:** As an alternative to the above, we want to implement a strategy of replacing the least frequently watched titles in the cache.
* **Feature # 7:** During a user session, a user often “shops” around for a program to watch. During this session, we want to let them move back and forth in the history of programs they’ve just browsed. As a developer, you can smell a *stack*, right? But, we also want the user to be able to directly jump to the top-ranked program from the one’s they’ve browsed.
* **Feature # 8:** As you beta tested **feature #7**, a user complained that the *next* and *previous* functionality isn’t working correctly. Using their session history, we want to check if our implementation is correct or indeed buggy.
* **Feature # 9:** Given an initial set of categories/genres, we want to implement a functionality that returns all the possible movie combinations of a particular category/genre.
* **Feature # 10:** During a user session, there is often a possibility that issues like packet drops and buffering events might emerge. We want to record the median number of buffering events that might occur in a particular session, which could then be used to improve the user experience.
* **Feature # 11:** A set of movies needs to be presented in different orders to a different set of users. We want to generate all the possible permutations of movies in a given marathon.
* **Feature # 12:** For enhancing the Netflix user experience, we want to revamp the continue watching bar that will return the most frequently watched show.

Understanding these feature requests and designing their solutions will help us implement the requested functionality into Netflix’s system.

In the next few lessons, we’ll discuss the recommended implementations of these features. Before we start, we suggest that you think about how you would implement these features. As you do, you’ll realize some of the underlying problems that you’ll need to solve. The solutions to these basic problems are also applicable to other common coding interview questions.

**Feature #1: Group Similar Titles**

Implementing the "Group Similar Titles" feature for our "Netflix" project.

**We'll cover the following**

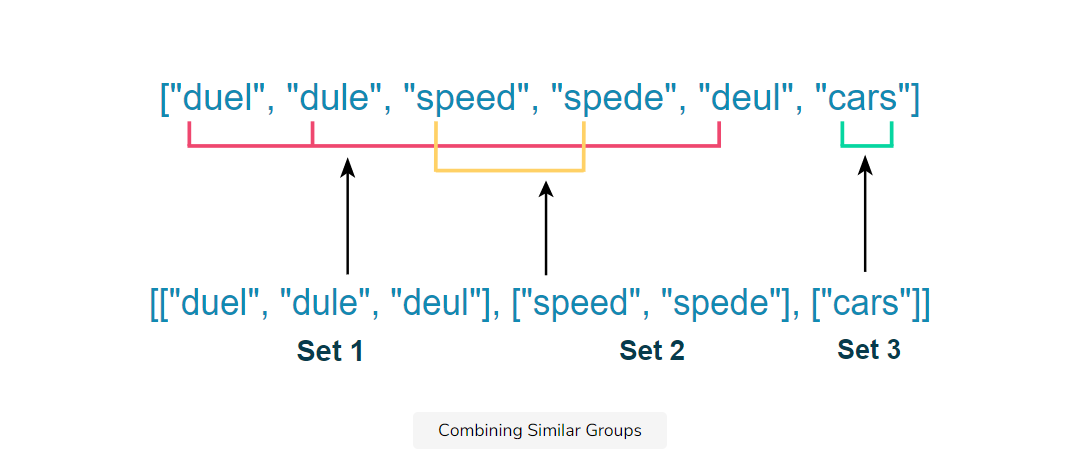
* + [Description](https://www.educative.io/courses/decode-coding-interview-java/3YZrlVv7789#Description)
  + [Solution](https://www.educative.io/courses/decode-coding-interview-java/3YZrlVv7789#Solution)
  + [Complexity measures](https://www.educative.io/courses/decode-coding-interview-java/3YZrlVv7789#Complexity-measures)
    - [Time Complexity](https://www.educative.io/courses/decode-coding-interview-java/3YZrlVv7789#Time-Complexity)
    - [Space complexity](https://www.educative.io/courses/decode-coding-interview-java/3YZrlVv7789#Space-complexity)

**Description**[#](https://www.educative.io/courses/decode-coding-interview-java/3YZrlVv7789#Description)

First, we need to figure out a way to individually group all the character combinations of each title. Suppose the content library contains the following titles: "duel", "dule", "speed", "spede", "deul", "cars". How would you efficiently implement a functionality so that if a user misspells speed as spede, they are shown the correct title?

We want to split the list of titles into sets of words so that all words in a set are anagrams. In the above list, there are three sets: {"duel", "dule", "deul"}, {"speed", "spede"}, and {"cars"}. Search results should comprise all members of the set that the search string is found in. We should pre-compute these sets instead of forming them when the user searches a title.

Here is an illustration of this process:



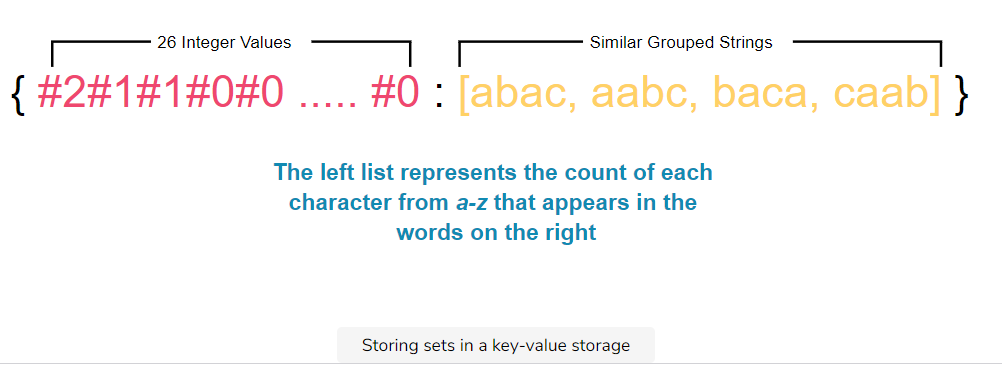
## Solution[#](https://www.educative.io/courses/decode-coding-interview-java/3YZrlVv7789#Solution)

From the above description, we see that all members of each set are characterized by the same frequency of each alphabet. This means that the frequency of each alphabet in words belonging to the same group is equal. In the set {{"speed", "spede"}}, the frequency of the characters s, p, e, and d are the same in each word.

Let’s see how we might implement this functionality:

1. For each title, compute a 26-element vector. Each element in this vector represents the frequency of an English letter in the corresponding title. This frequency count will be represented as a string delimited with # characters. For example, abbccc will be represented as #1#2#3#0#0#0...#0. This mapping will generate identical vectors for strings that are anagrams.
2. Use this vector as a key to insert the titles into a Hash Map. All anagrams will be mapped to the same entry in this Hash Map. When a user searches a word, compute the 26-element English letter frequency vector based on the word. Search in the Hash Map using this vector and return all the map entries.
3. Store the vector of the calculated character counts in the same Hash Map as a key and assign the respective set of anagrams as its value.
4. Return the values of the Hash Map, since each value will be an individual set.

Let’s look at the following illustration to clarify this process:



import java.util.\*;

class Solution {

public static List<List<String>> groupTitles(String[] strs){

if (strs.length == 0)

return new ArrayList<List<String>>();

Map<String, List<String>> res = new HashMap<String, List<String>>();

int[] count = new int[26];

for (String s : strs) {

Arrays.fill(count, 0);

for (char c : s.toCharArray()){

int index = c - 'a';

count[index]++;

}

StringBuilder delimStr = new StringBuilder("");

for (int i = 0; i < 26; i++) {

delimStr.append('#');

delimStr.append(count[i]);

}

String key = delimStr.toString();

if (!res.containsKey(key))

res.put(key, new ArrayList<String>());

res.get(key).add(s);

}

return new ArrayList<List<String>>(res.values());

}

public static void main(String[] args) {

// Driver code

String titles[] = {"duel","dule","speed","spede","deul","cars"};

List<List<String>> gt = groupTitles(titles);

String query = "spede";

// Searching for all titles

for (List<String> g : gt){

if (g.contains(query))

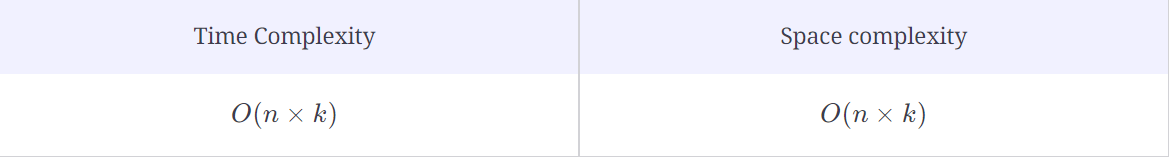
System.out.println(g);

}

}

}

## Complexity measures



Let **n** be the size of the list of strings, and **k** be the maximum length that a single string can have.

### Time Complexity[**#**](https://www.educative.io/courses/decode-coding-interview-java/3YZrlVv7789#Time-Complexity)

We are counting each letter for each string in a list, so the time complexity will be O(*n*×*k*).

### Space complexity[**#**](https://www.educative.io/courses/decode-coding-interview-java/3YZrlVv7789#Space-complexity)

Since every string is being stored as a value in the dictionary whose size can be **n**, and the size of the string can be **k**, so space complexity is O(*n*×*k*).

# Feature #2: Fetch Top Movies

Implementing the "Fetch Top Movies" feature for our "Netflix" project.

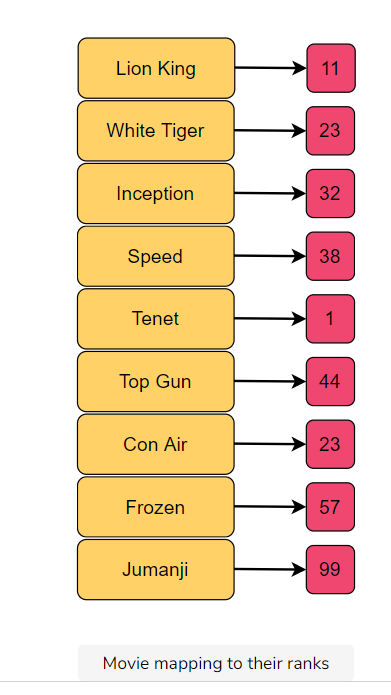
**We'll cover the following**

* + [Description](https://www.educative.io/courses/decode-coding-interview-java/NEZBZBgOo6K#Description)
  + [Solution](https://www.educative.io/courses/decode-coding-interview-java/NEZBZBgOo6K#Solution)
  + [Complexity measures](https://www.educative.io/courses/decode-coding-interview-java/NEZBZBgOo6K#Complexity-measures)
    - [Time Complexity](https://www.educative.io/courses/decode-coding-interview-java/NEZBZBgOo6K#Time-Complexity)
    - [Space complexity](https://www.educative.io/courses/decode-coding-interview-java/NEZBZBgOo6K#Space-complexity)

## Description[#](https://www.educative.io/courses/decode-coding-interview-java/NEZBZBgOo6K#Description)

Now, we need to build a criterion so the top movies from multiple countries will combine into a single list of top-rated movies. In order to scale, the content search is performed in a distributed fashion. Search results for each country are produced in separate lists. Each member of a given list is ranked by popularity, with 1 being most popular and popularity decreasing as the rank number increases.

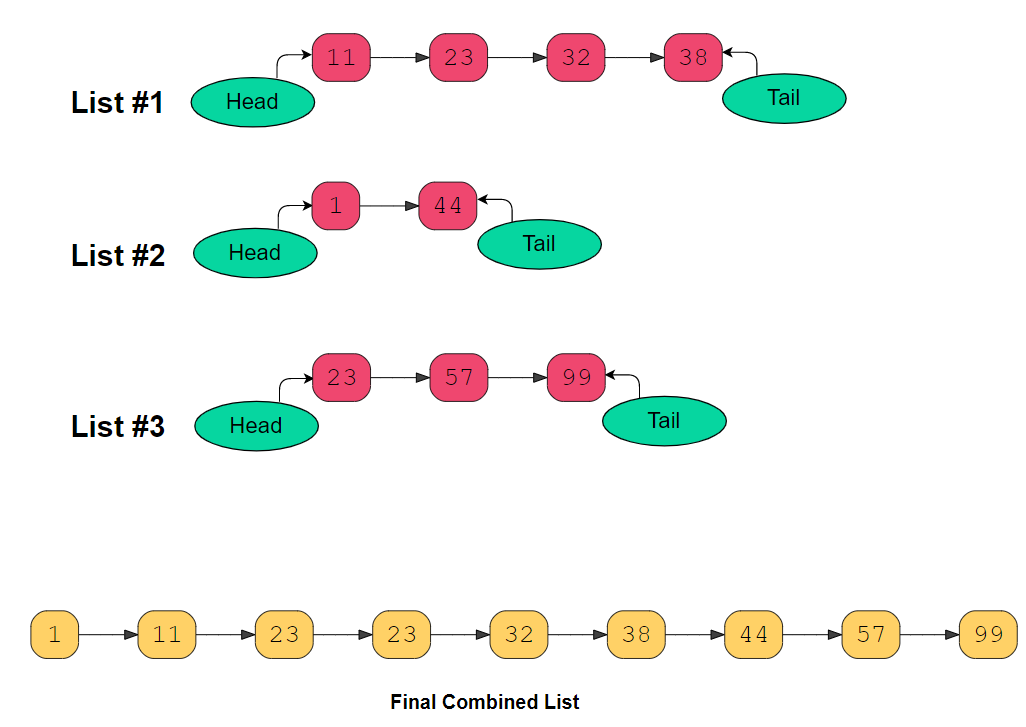
Let’s say that the following titles are represented by the provided IDs:



We’ll be given **n** arrays that are all sorted in ascending order of popularity rank. We have to combine these lists into a single list that will be sorted by rank in ascending order, meaning from best to worst.

Keep in mind that the ranks are unique to individual movies and a single rank can be in multiple lists.

Let’s understand this better with an illustration:



## Solution[#](https://www.educative.io/courses/decode-coding-interview-java/NEZBZBgOo6K#Solution)

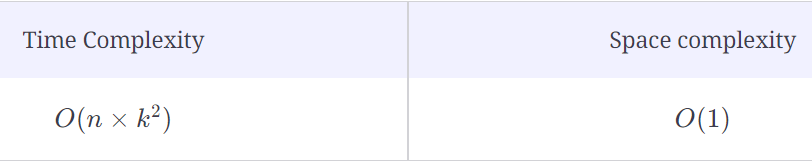
Since our task involves multiple lists, you should divide the problem into multiple tasks, starting with the problem of combining two lists at a time. Then, you should combine the result of those first two lists with the third list, and so on, until the very last one is reached.

Let’s discuss how we will implement this process:

1. Consider the first list as the result, and store it in a variable.
2. Traverse the list of lists, starting from the second list, and combine it with the list we stored as a result. The result should get stored in the same variable.
3. When combining the two lists, like l1 and l2, maintain a prev pointer that points to a dummy node.
4. If the value for list l1 is less than or equal to the value for list l2, connect the previous node to l1 and increment l1. Otherwise, do the same but for list l2.
5. Keep repeating the above step until one list points to a null value.
6. Connect the non-null list to the merged one and return.

Let’s look at the code for the solution below:

## Complexity measures



### Time Complexity[**#**](https://www.educative.io/courses/decode-coding-interview-java/NEZBZBgOo6K#Time-Complexity)

The time complexity will be O(n times k ^ 2), where **k** is the number of the lists and **n** is the maximum length of a single list.

### Space complexity[**#**](https://www.educative.io/courses/decode-coding-interview-java/NEZBZBgOo6K#Space-complexity)

O(1), as constant space was utilized.

# Feature #3: Find Median Age

Implementing the "Find Median Age" feature for our "Netflix" project.

**We'll cover the following**

* + [Description](https://www.educative.io/courses/decode-coding-interview-java/BnQv5mWkg1x#Description)
  + [Solution](https://www.educative.io/courses/decode-coding-interview-java/BnQv5mWkg1x#Solution)
  + [Complexity measures](https://www.educative.io/courses/decode-coding-interview-java/BnQv5mWkg1x#Complexity-measures)
    - [Time Complexity](https://www.educative.io/courses/decode-coding-interview-java/BnQv5mWkg1x#Time-Complexity)
    - [Memory complexity](https://www.educative.io/courses/decode-coding-interview-java/BnQv5mWkg1x#Memory-complexity)

## Description[#](https://www.educative.io/courses/decode-coding-interview-java/BnQv5mWkg1x#Description)

Our third task is building a filter that will fetch relevant content based on the age of the users for a specific country or region. For this, we use the median age of users and the preferred content for users that fall into that specified category.

Because the number of users is constantly increasing on the Netflix platform, we need to figure out a structure or design that updates the median age of users in real-time. We will have an array that constantly receives age values, and we will output the median value after each new data point.

Let’s understand this better with an illustration:

## Solution[#](https://www.educative.io/courses/decode-coding-interview-java/BnQv5mWkg1x#Solution)

We will assume that **‘x’** is the median age of a user in a list. Half of the ages in the list will be smaller than (or equal to) **‘x’**, and the other half will be greater than (or equal to) **‘x’**. We can divide the list into two halves: one half to store the smaller numbers (say smallList), and one half to store the larger numbers (say largeList). The median of all ages will either be the largest number in the smallList or the smallest number in the largeList. If the total number of elements is even, we know that the median will be the average of these two numbers. The best data structure for finding the smallest or largest number among a list of numbers is a [Heap](https://www.educative.io/edpresso/what-is-a-heap).

Here is how we will implement this feature:

1. First, we will store the first half of the numbers (smallList) in a **Max Heap**. We use a **Max Heap** because we want to know the largest number in the first half of the list.
2. Then, we will store the second half of the numbers (largeList) in a **Min Heap**, because we want to know the smallest number in the second half of the list.
3. We can calculate the median of the current list of numbers using the top element of the two heaps.

Let’s look at the code for this solution below:

import java.util.\*;

class MedianOfAges {

PriorityQueue<Integer> smallList; //containing first half of numbers

PriorityQueue<Integer> largeList; //containing second half of numbers

public MedianOfAges() {

smallList = new PriorityQueue<>((a, b) -> b - a);

largeList = new PriorityQueue<>((a, b) -> a - b);

}

public void insertNum(int num) {

if (smallList.isEmpty() || smallList.peek() >= num)

smallList.add(num);

else

largeList.add(num);

// either both the heaps will have equal number of elements or max-heap will have one

// more element than the min-heap

if (smallList.size() > largeList.size() + 1)

largeList.add(smallList.poll());

else if (smallList.size() < largeList.size())

smallList.add(largeList.poll());

}

public double findMedian() {

if (smallList.size() == largeList.size()) {

// we have even number of elements, take the average of middle two elements

return smallList.peek() / 2.0 + largeList.peek() / 2.0;

}

// because max-heap will have one more element than the min-heap

return smallList.peek();

}

public static void main(String[] args) {

// Driver code

MedianOfAges MedianOfAges = new MedianOfAges();

MedianOfAges.insertNum(22);

MedianOfAges.insertNum(35);

System.out.println("The recommended content will be for ages under: " + MedianOfAges.findMedian());

MedianOfAges.insertNum(30);

System.out.println("The recommended content will be for ages under: " + MedianOfAges.findMedian());

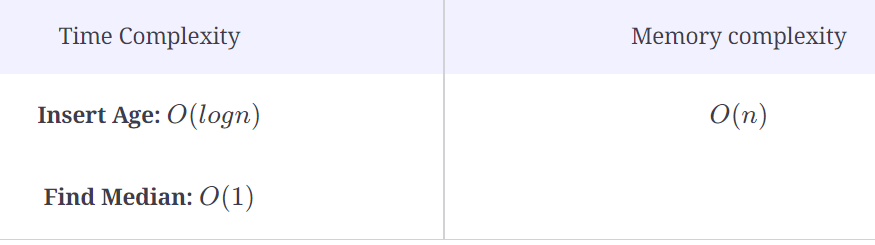
MedianOfAges.insertNum(25);

System.out.println("The recommended content will be for ages under: " + MedianOfAges.findMedian());

}

}

## Complexity measures[#](https://www.educative.io/courses/decode-coding-interview-java/BnQv5mWkg1x#Complexity-measures)



### Time Complexity[**#**](https://www.educative.io/courses/decode-coding-interview-java/BnQv5mWkg1x#Time-Complexity)

The time complexity of the **Insert Age** will be O(logn) because we inserted in the heap. The time complexity of the **Find Median** will be O(1) because we can find the median from the top elements of the heaps.

### Memory complexity[**#**](https://www.educative.io/courses/decode-coding-interview-java/BnQv5mWkg1x#Memory-complexity)

The memory complexity will be O(n) because we will be storing all the numbers at any time.

# Feature #4: Popularity Analysis

Implementing the "Popularity Analysis" feature for our "Netflix" project.

**We'll cover the following**

* + [Description](https://www.educative.io/courses/decode-coding-interview-java/B6A3DNxqD5k#Description)
  + [Solution](https://www.educative.io/courses/decode-coding-interview-java/B6A3DNxqD5k#Solution)
  + [Complexity measures](https://www.educative.io/courses/decode-coding-interview-java/B6A3DNxqD5k#Complexity-measures)
    - [Time complexity](https://www.educative.io/courses/decode-coding-interview-java/B6A3DNxqD5k#Time-complexity)
    - [Space complexity](https://www.educative.io/courses/decode-coding-interview-java/B6A3DNxqD5k#Space-complexity)

## Description[#](https://www.educative.io/courses/decode-coding-interview-java/B6A3DNxqD5k#Description)

Netflix maintains a popularity score for each of its titles. This popularity score is derived from customer feedback, likes, dislikes, etc. This score is updated weekly and added to the end of an array containing previous scores for the same title. This score array helps Netflix identify titles that may be increasing or decreasing in popularity over time. Some titles may be steady in popularity, increasing, decreasing, and fluctuating. We want to identify and separate a title if it is gaining or losing popularity.

We’ll be provided with an array of integers representing the popularity scores of a movie collected over a number of weeks. We need to identify only those titles that are either increasing or decreasing in popularity, so we can separate them from the fluctuating ones for better analysis.

## Solution[#](https://www.educative.io/courses/decode-coding-interview-java/B6A3DNxqD5k#Solution)

An array is increasing if the expression arr[i] <= arr[i + 1] evaluates to true for every element at index i. Similarly, an array is decreasing if the expression arr[i] >= arr[i + 1] evaluates to true for every element at index i.

From the start, we’ll assume that the array is either increasing or decreasing. We have established that there won’t be any adjacent values of the form arr[i] > arr[i + 1] for the increasing array, and there won’t be any adjacent values of the form arr[i] < arr[i + 1] for decreasing array. For the array to be either increasing or decreasing, all pairs of consecutive elements must obey one of these conditions, but not both.

If both increasing and decreasing variables are still true by the end, it means that the popularity is constant and not fluctuating.

Let’s look at the code for the solution:

class Solution {

public static boolean identifyTitles(int[] scores) {

boolean increasing = true;

boolean decreasing = true;

for (int i = 0; i < scores.length - 1; i++) {

if (scores[i] > scores[i+1])

increasing = false;

if (scores[i] < scores[i+1])

decreasing = false;

}

return (increasing || decreasing);

}

public static void main( String args[] ) {

// Driver code

int[][] movie\_ratings = {

{1,2,2,3},

{4,5,6,3,4},

{8,8,7,6,5,4,4,1}

};

for (int[] movie\_rating : movie\_ratings){

if (identifyTitles(movie\_rating))

System.out.println("Title Identified and Separated");

else

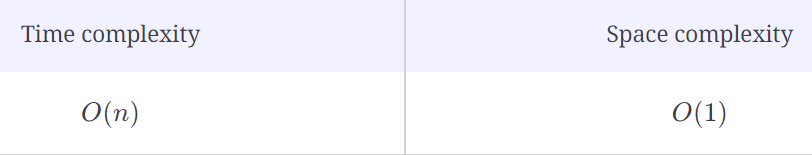
System.out.println("Title Score Fluctuating");

}

}

}

## Complexity measures



Let **n** be the size of the list.

### Time complexity[**#**](https://www.educative.io/courses/decode-coding-interview-java/B6A3DNxqD5k#Time-complexity)

The time complexity will be O(n) because we only did a single pass of the list.

### Space complexity[**#**](https://www.educative.io/courses/decode-coding-interview-java/B6A3DNxqD5k#Space-complexity)

The space complexity will be O(1) because constant space will be utilized.

# Feature #5: Fetch Most Recently Watched Titles

Implementing the "Fetch Most Recently Watched Titles" feature for our "Netflix" project.

**We'll cover the following**

* [Description](https://www.educative.io/courses/decode-coding-interview-java/m77pNlAMJ19#Description)
* [Solution](https://www.educative.io/courses/decode-coding-interview-java/m77pNlAMJ19#Solution)
* [Complexity measures](https://www.educative.io/courses/decode-coding-interview-java/m77pNlAMJ19#Complexity-measures)
  + [Time Complexity](https://www.educative.io/courses/decode-coding-interview-java/m77pNlAMJ19#Time-Complexity)
  + [Space complexity](https://www.educative.io/courses/decode-coding-interview-java/m77pNlAMJ19#Space-complexity)

## Description[#](https://www.educative.io/courses/decode-coding-interview-java/m77pNlAMJ19#Description)

We want to implement a feature so that the user can view the **n** titles that they’ve watched or accessed most recently. You need to come up with a data structure for this feature. Let’s break it down. If we think it through, we realize the following: **i)** This data structure should maintain titles in order of time since last access; **ii)** If the data structure is at its capacity, an insertion should replace the least recently accessed item.

We need to design a structure that takes the ID of our movies, adds them to our data structure, and fetches movie titles when accessed through their ID. We also need to consider the size of our data structure and maintain the least recently watched property.

## Solution[#](https://www.educative.io/courses/decode-coding-interview-java/m77pNlAMJ19#Solution)

A doubly linked list provides an ideal way of maintaining ordered elements. We can keep the most recently accessed item at the tail. But when a recently watched item is accessed again, we’ll move it to the tail of the linked list. This will require searching for an element in the linked list, which is expensive. To fix this, we can use a hash map to store the pointers of the linked list elements.

Here’s how we implement the feature:

1. If the element exists in the hashmap, our first step is to move the accessed element to the tail of the linked list.
2. If the data structure is at its capacity, remove the element at the head of the linked list and the Hash Map. Then, we add the new element at the tail of the linked list and in the Hash Map.
3. Finally, we retrieve the element and return.

Let’s look at the code for the solution below:

class LRUCache {

int capacity;

HashMap<Integer, LinkedListNode> cache;

MyLinkedList cacheVals;

public LRUCache(int capacity) {

this.capacity = capacity;

cache = new HashMap<Integer, LinkedListNode>(capacity);

cacheVals = new MyLinkedList();

}

LinkedListNode Get(int key) {

if(!cache.containsKey(key)){

return null;

}

else {

int value = cache.get(key).data;

cacheVals.removeNode(cache.get(key));

cacheVals.insertAtTail(key, value);

return cacheVals.getTail();

}

}

void Set(int key, int value) {

if (!cache.containsKey(key)){

evictIfNeeded();

cacheVals.insertAtTail(key, value);

cache.put(key, cacheVals.getTail());

}

else {

cacheVals.removeNode(cache.get(key));

cacheVals.insertAtTail(key, value);

cache.put(key, cacheVals.getTail());

}

}

void evictIfNeeded(){

if(cacheVals.size >= capacity) {

LinkedListNode node = cacheVals.removeHead();

cache.remove(node.key);

}

}

void print() {

LinkedListNode curr = cacheVals.head;

while(curr != null){

System.out.print("(" + curr.key + "," + curr.data + ")");

curr = curr.next;

}

System.out.println("");

}

public static void main(String[] args){

LRUCache cache = new LRUCache(3);

System.out.print("The most recently watched titles are: (key, value)");

cache.Set(10,20);

cache.print();

cache.Set(15,25);

cache.print();

cache.Set(20,30);

cache.print();

cache.Set(25,35);

cache.print();

cache.Set(5,40);

cache.print();

cache.Get(25);

cache.print();

}

}

public class MyLinkedList {

public LinkedListNode head;

public LinkedListNode tail;

public int size;

public MyLinkedList(){

this.head = null;

this.tail = null;

this.size = 0;

}

public void insertAtHead(int key, int data) {

LinkedListNode newNode = new LinkedListNode(key, data);

if(this.head == null){

this.head = newNode;

this.tail = newNode;

}

else{

newNode.next = this.head;

this.head.prev = newNode;

this.head = newNode;

}

this.size++;

}

public void insertAtTail(int key, int data) {

LinkedListNode newNode = new LinkedListNode(key, data);

if (this.tail == null){

this.tail = newNode;

this.head = newNode;

newNode.next = null;

newNode.prev = null;

}

else{

this.tail.next = newNode;

newNode.prev = this.tail;

this.tail = newNode;

newNode.next = null;

}

this.size++;

}

public LinkedListNode getHead(){

return this.head;

}

public LinkedListNode getTail(){

return this.tail;

}

public LinkedListNode removeNode(LinkedListNode node){

if(node == null){

return null;

}

if(node.prev != null){

node.prev.next = node.next;

}

if(node.next != null){

node.next.prev = node.prev;

}

if(node == this.head){

this.head = this.head.next;

}

if(node == this.tail){

this.tail = this.tail.prev;

}

this.size--;

return node;

}

public void remove(int data){

LinkedListNode i = this.getHead();

while(i != null){

if(i.data == data){

this.removeNode(i);

}

i = i.next;

}

}

public LinkedListNode removeHead(){

return this.removeNode(this.head);

}

public LinkedListNode removeTail(){

return this.removeNode(this.tail);

}

}

class LinkedListNode {

public int key;

public int data;

public LinkedListNode next;

public LinkedListNode prev;

public LinkedListNode(int data) {

this.data = data;

this.next = null;

this.prev = null;

}

public LinkedListNode(int key, int data) {

this.key = key;

this.data = data;

this.next = null;

this.prev = null;

}

public LinkedListNode(int data, LinkedListNode next) {

this.data = data;

this.next = next;

this.prev = null;

}

public LinkedListNode(int data, LinkedListNode next, LinkedListNode prev) {

this.data = data;

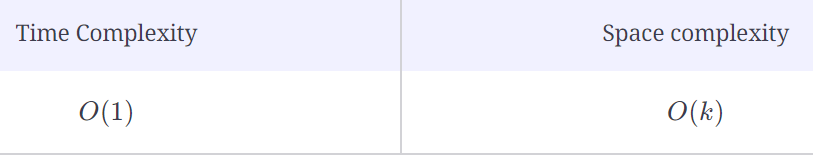
this.next = next;

this.prev = prev;

}

}

## Complexity measures[#](https://www.educative.io/courses/decode-coding-interview-java/m77pNlAMJ19#Complexity-measures)



### Time Complexity[**#**](https://www.educative.io/courses/decode-coding-interview-java/m77pNlAMJ19#Time-Complexity)

We know that the time complexity of this solution is constant, O(1) because of these complexities:

Get (HashMap): Constant, O(1)

Set (HashMap): Constant, O(1)

Addition, deletion from the head and tail of LinkedList: Constant, O(1)

### Space complexity[**#**](https://www.educative.io/courses/decode-coding-interview-java/m77pNlAMJ19#Space-complexity)

The memory complexity of this solution is, O(k), where **k** is the size of the cache.

# Feature #6: Fetch Most Frequently Watched Titles

Implementing the "Fetch Most Frequently Watched Titles" feature for our "Netflix" project.

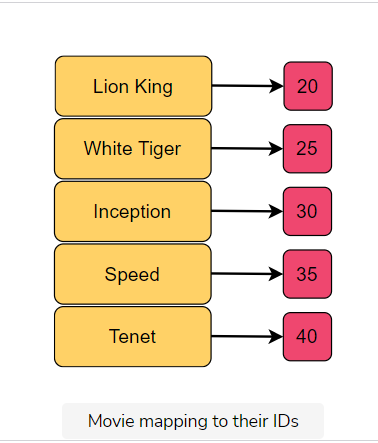
**We'll cover the following**

* [Description](https://www.educative.io/courses/decode-coding-interview-java/gxZvZOQq9YY#Description)
* [Solution](https://www.educative.io/courses/decode-coding-interview-java/gxZvZOQq9YY#Solution)
* [Complexity measures](https://www.educative.io/courses/decode-coding-interview-java/gxZvZOQq9YY#Complexity-measures)
  + [Time Complexity](https://www.educative.io/courses/decode-coding-interview-java/gxZvZOQq9YY#Time-Complexity)
  + [Space complexity](https://www.educative.io/courses/decode-coding-interview-java/gxZvZOQq9YY#Space-complexity)

## Description

This feature and the one we discussed before are almost similar, but now the titles are maintained in order of viewing/access frequency. When the data structure is at capacity, a newly inserted item will replace the least frequently accessed item. In the case of a tie, the least recently accessed item should be replaced.

Let’s say that a user has gone through the following titles:



We need to design a structure that takes the ID of the movies and adds them to our structure. It also needs to fetch a movie title when accessed through its ID. We also have to consider the size of the data structure and maintain the least frequently watched property. This means that when our structure is at its capacity then a newly inserted item will replace the item whose frequency count is the lowest.

Here is an illustration of this process. When we insert the value 5, then the element with the least frequency is 1. Hence, it gets replaced by 5.

## Solution

We’ll build this structure on top of the one from the previous lesson. We will also use a Hash Map and a doubly linked list. In this case, we’ll also store data on how frequently titles are accessed along with their respective keys and values. We’ll append each new entry at the tail of the doubly linked list. A Hash Map will be used to keep track of the keys and their values in the linked list.

You could create a Hash Map that will store the key, value, and frequency count for that value. Since multiple titles can have the same access frequency, the value stored in the Hash Map for each key will be a doubly linked list. The most recently accessed title is inserted at the tail of this linked list. We can easily remove the least recently accessed title by removing it from the head of the linked list. However, the Hash Map entries are linked lists, so lookups are slow. To fix this problem, we can use a second Hash Map with a unique key for every item in the first Hash Map. The value corresponding to a key in this second Hash Map will point to the corresponding node in the linked list.

Here’s how we will implement this feature:

1. If an element is accessed and is present in our data structure, we will:
   * Increase its frequency count
   * Move it to the end of its respective list
2. If an element is added and there is space for it in our structure, we create a node with the specified key and value, assign the node a frequency count of **1**, and increase the size of the structure.
3. If an element is added and eviction is needed, we will delete the keys and references of the least frequent node from both the Hash Maps. Then, we simply repeat step 2.
4. If a key already exists for a certain key-value pair, we update the value of the node for the respective key.

Let’s look at the code for the solution below:

class LFUCache {

int capacity;

int size;

int minFreq;

//LinkedListNode holds key and value pairs

HashMap<Integer, LinkedListNode> keyDict;

HashMap<Integer, MyLinkedList> freqDict;

public LFUCache(int capacity) {

this.capacity = capacity;

this.size = 0;

this.minFreq = 0;

keyDict = new HashMap<Integer, LinkedListNode>(capacity);

freqDict = new HashMap<Integer, MyLinkedList>(capacity);

}

LinkedListNode Get(int key) {

if(!keyDict.containsKey(key)){

return null;

}

LinkedListNode temp = this.keyDict.get(key);

this.freqDict.get(temp.freq).deleteNode(temp);

if(this.freqDict.get(this.keyDict.get(key).freq) == null){

this.freqDict.remove(this.keyDict.get(key).freq);

if(this.minFreq == this.keyDict.get(key).freq){

this.minFreq += 1;

}

}

this.keyDict.get(key).freq += 1;

if(!this.freqDict.containsKey(this.keyDict.get(key).freq)){

this.freqDict.put(this.keyDict.get(key).freq, new MyLinkedList());

}

this.freqDict.get(this.keyDict.get(key).freq).append(this.keyDict.get(key));

return this.keyDict.get(key);

}

void Set(int key, int value) {

if(this.Get(key) != null){

this.keyDict.get(key).val = value;

return;

}

if(this.size == this.capacity){

this.keyDict.remove(this.freqDict.get(this.minFreq).head.key);

this.freqDict.get(this.minFreq).deleteNode(this.freqDict.get(this.minFreq).head);

if(this.freqDict.get(this.minFreq) == null){

this.freqDict.remove(this.minFreq);

}

this.size -= 1;

}

this.minFreq = 1;

this.keyDict.put(key, new LinkedListNode(key, value, this.minFreq));

if(!this.freqDict.containsKey(this.keyDict.get(key).freq)){

this.freqDict.put(this.keyDict.get(key).freq, new MyLinkedList());

}

this.freqDict.get(this.keyDict.get(key).freq).append(this.keyDict.get(key));

this.size++;

}

void print() {

for (Map.Entry<Integer, LinkedListNode> entry : keyDict.entrySet()) {

System.out.print("(" + entry.getKey() + ", " + entry.getValue().val + ")");

}

System.out.println("");

}

public static void main(String[] args){

LFUCache cache = new LFUCache(3);

System.out.print("The most frequently watched titles are: (key, value)\n");

cache.Set(1, 2);

cache.Set(2, 2);

cache.print();

cache.Get(1);

cache.Get(1);

cache.Set(3, 3);

cache.print();

}

}

public class MyLinkedList{

public LinkedListNode head;

public LinkedListNode tail;

public MyLinkedList(){

this.head = null;

this.tail = null;

}

public void append(LinkedListNode node){

node.next = null;

node.prev = null;

if(this.head == null){

this.head = node;

}

else{

this.tail.next = node;

node.prev = this.tail;

}

this.tail = node;

}

public void deleteNode(LinkedListNode node){

if(node.prev != null){

node.prev.next = node.next;

}

else{

this.head = node.next;

}

if(node.next != null){

node.next.prev = node.prev;

}

else{

this.tail = node.prev;

}

node.next = null;

node.prev = null;

}

}

public class LinkedListNode {

public int key;

public int val;

public int freq;

public LinkedListNode next;

public LinkedListNode prev;

public LinkedListNode(int key, int val, int freq) {

this.key = key;

this.val = val;

this.freq = freq;

this.next = null;

this.prev = null;

}

}

## Complexity measures

### Time Complexity

We know that the time complexity of this solution is constant, O(1), because of these complexities:

Get (HashSet): Constant, O(1)

Set (HashSet): Constant, O(1)

Deletion at the head when adding a new element (linked list): Constant, O(1)

Adding a new element to tail (linked list): O(1)

### Space complexity

The memory complexity of this solution is linear, O(n), where **n** is the capacity of the cache.

# Feature #7: Browse Ratings

Implementing the "Browse Ratings" feature for our "Netflix" project.

**We'll cover the following**

* [Description](https://www.educative.io/courses/decode-coding-interview-java/qZ6DmLDZAJr#Description)
* [Solution](https://www.educative.io/courses/decode-coding-interview-java/qZ6DmLDZAJr#Solution)
* [Complexity measures](https://www.educative.io/courses/decode-coding-interview-java/qZ6DmLDZAJr#Complexity-measures)
  + [Time complexity](https://www.educative.io/courses/decode-coding-interview-java/qZ6DmLDZAJr#Time-complexity)
  + [Space complexity](https://www.educative.io/courses/decode-coding-interview-java/qZ6DmLDZAJr#Space-complexity)

## Description[#](https://www.educative.io/courses/decode-coding-interview-java/qZ6DmLDZAJr#Description)

In this feature, the user will be able to randomly browse through movie titles and read their summaries and reviews. We want to enable a **Back** button so the user can return to the previous title in the viewing history. We also want the user to immediately get the title with the highest viewer rating from their viewing history. We want to implement both of these operations in constant time to provide a good user experience.

We’ll be provided with a sequential input of ratings to simulate the user viewing them one by one. For simplicity, we’ll assume that the movie ratings are all unique.

Let’s say that a user has browsed through the movies with the following ratings:



## Solution[#](https://www.educative.io/courses/decode-coding-interview-java/qZ6DmLDZAJr#Solution)

To enable a Back button, we need to store the user’s viewing history in some data structure. Pressing the **Back** button fetches the last viewed item. This indicates a **last in first out (LIFO)** structure, which is characteristic of a stack. In a stack, push and pop operations can be easily implemented in O(1)*O*(1). However, the stack doesn’t allow random access to its elements, let alone access to the element with the maximum rating. We will need to create a stack-like data structure that offers a getMax operation, in addition to push and pop, that all run in O(1)*O*(1).

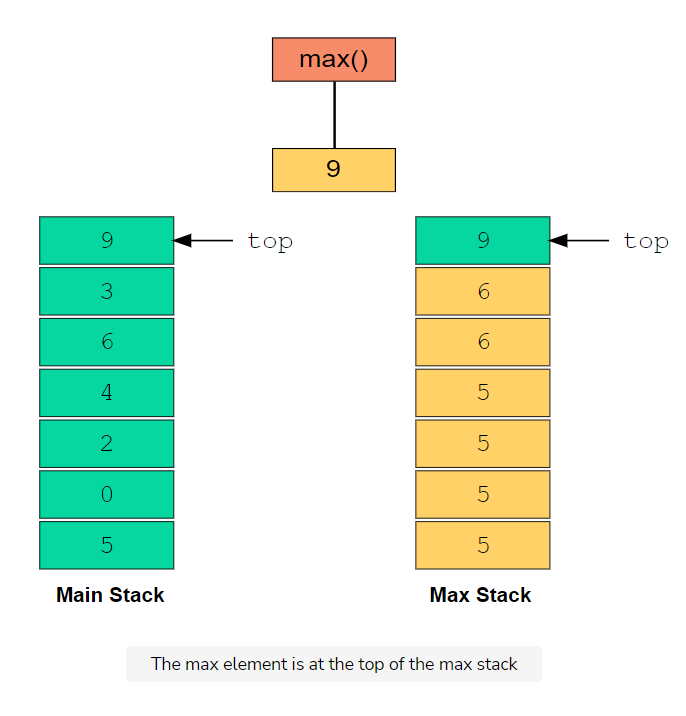
The implementation of such a data structure can be realized with the help of two stacks: maxStack and mainStack. The mainStack holds the actual stack with all the elements, and maxStack is a stack whose top always contains the current maximum value in the stack.

How does it do this? The answer is in the implementation of the push function. Whenever push is called, mainStack simply inserts it at the top. However, maxStack checks the value being pushed. If maxStack is empty, this value is pushed into it and becomes the current maximum. If maxStack already has elements in it, the value is compared with the top element in this stack. The element is pushed into maxStack if it is greater than the top element else, the top element is pushed again.

The pop() function simply pops off the top element from mainStack and maxStack.

Due to all these safeguards, the maxRating function only needs to return the value at the top of maxStack.

Below is an illustration of this process:



Let’s look at the code for the solution:

class Solution {

public static void main(String args[]) {

maxStack stack = new maxStack(7);

stack.push(5);

stack.push(0);

stack.push(2);

stack.push(4);

stack.push(6);

stack.push(3);

stack.push(10);

System.out.println("Maximum Rating: " + stack.maxRating());

stack.pop();

System.out.println("\nAfter clicking back button\n");

System.out.println("Maximum Rating: " + stack.maxRating());

}

}

public class maxStack {

int maxSize;

Stack<Integer> mainStack;

Stack<Integer> maxStack;

//constructor

public maxStack(int maxSize) {

//We will use two stacks mainStack to hold original values

//and maxStack to hold maximum values. Top of maxStack will always

//be the maximum value from mainStack

this.maxSize = maxSize;

mainStack = new Stack<>(maxSize);

maxStack = new Stack<>(maxSize);

}

//removes and returns value from stack

public int pop(){

//1. Pop element from maxStack to make it sync with mainStack,

//2. Pop element from mainStack and return that value

maxStack.pop();

return mainStack.pop();

}

//pushes value into the stack

public void push(Integer value){

//1. Push value in mainStack and check value with the top value of maxStack

//2. If value is greater than top, then push top in maxStack

//else push value in maxStack

mainStack.push(value);

if(!maxStack.isEmpty() && maxStack.top() > value)

maxStack.push(maxStack.top());

else

maxStack.push(value);

}

//returns maximum value in O(1)

public int maxRating(){

return maxStack.top();

}

}

public class Stack<V> {

private int maxSize;

private int top;

private V[] array;

private int currentSize;

/\*

Java does not allow generic type arrays. So we have used an

array of Object type and type-casted it to the generic type V.

This type-casting is unsafe and produces a warning.

Comment out the line below and execute again to see the warning.

\*/

@SuppressWarnings("unchecked")

public Stack(int maxSize) {

this.maxSize = maxSize;

this.top = -1; //initially when stack is empty

array = (V[]) new Object[maxSize];//type casting Object[] to V[]

this.currentSize = 0;

}

public int getCurrentSize() {

return currentSize;

}

//returns the maximum size capacity

public int getMaxSize() {

return maxSize;

}

//returns true if Stack is empty

public boolean isEmpty() {

return top == -1;

}

//returns true if Stack is full

public boolean isFull() {

return top == maxSize - 1;

}

//returns the value at top of Stack

public V top() {

if (isEmpty())

return null;

return array[top];

}

//inserts a value to the top of Stack

public void push(V value) {

if (isFull()) {

System.err.println("Stack is Full!");

return;

}

array[++top] = value; //increments the top and adds value to updated top

currentSize++;

}

//removes a value from top of Stack and returns

public V pop() {

if (isEmpty())

return null;

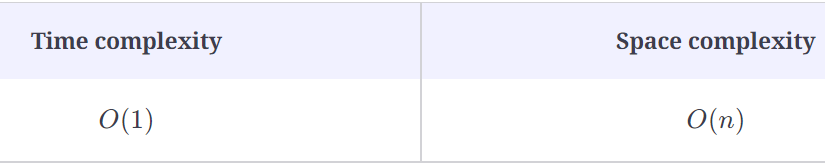
currentSize--;

return array[top--]; //returns value and top and decrements the top

}

}

## Complexity measures[#](https://www.educative.io/courses/decode-coding-interview-java/qZ6DmLDZAJr#Complexity-measures)



### Time complexity[**#**](https://www.educative.io/courses/decode-coding-interview-java/qZ6DmLDZAJr#Time-complexity)

In the push() operation, we are setting one element in the stack, so the time complexity will be O(1)*O*(1). In the pop() operation, we are removing one element, so the time complexity will also be O(1)*O*(1). Additionally, the maxRating function only fetches the top element of the max stack, it will be O(1)*O*(1) as well.

### Space complexity[**#**](https://www.educative.io/courses/decode-coding-interview-java/qZ6DmLDZAJr#Space-complexity)

In the worst case, all operations would be push on the two stacks, and O(n)*O*(*n*) space will be utilized.

What if you had to implement an efficient min stack, meaning a stack from which you can access the smallest element in constant time? Think about how your implementation would change?