**Anthropic**

Examples:

<https://aonecode.com/iq/docs/antropic/online-assessment/cloud-storage-system>

<https://aonecode.com/iq/docs/antropic/online-assessment/in-memory-db>

Check out all examples here:

<https://nodeflair.com/companies/anthropic/interviews>

General Coding Assessment Framework

<https://github.com/Leader-board/OA-and-Interviews/blob/main/media/general-coding-assessment-framework.pdf>

General CodeSignal coding problems: <https://codesignal.com/blog/interview-prep/example-codesignal-questions/>

**CodeSignal**

Comments:

Previously:  email said it’s a time crunch + the cleanliness of the code didn't matter and the only thing that matters is completion, correctness, speed.

Code very very fast and not to worry about anything else other than passing test cases.

The problems were not difficult, but the later questions only unlocked after you completed the earlier questions, but some of them required changing the architecture of your code, in ways that if you knew about in advance, you would have done differently.

Offer accepted: a Zoom call, a take-home assessment, and a final manager interview, all completed within two weeks. Everyone was great and the take home assessment didn't take too long and was reasonable. Good interview process experience. Question asked: why do you want join Anthropics?

Task is to write a bunch of APIs for reading/writing from an in-memory "database". 4 progressive rounds in 90 mins - first two were short, third took the bulk of the time. Only left with a few minutes for the last round. I think if I had 3 more minutes, I could have finished the final round

Design a file system in 4 parts. (Jan 2025)

Setting up a transaction system. (Jan 2025)

OOP app that has each stage build on the last. so you need to do refactors and design carefully (Aug 2024).

CodeSignal test divided a few levels. The goal is pass all the unit test of a given level before going to the next level. The candidate is given 90 minutes (Sep 2024).

I was given a helper method that does URL crawling. The question is given a domain name and a main URL, parse through all the URLs and get a count of total num of urls that matches the given domain name. (Feb 2025).

Google Colab - convert nested stack traces into discrete start and end events: given a list of samples that each contain a list of function names, generate start and end events for each function across the list of samples (Feb 2025)

Multi-thread for phone interview. onsite questions are related to filesystem. (Dec 2025).

Coding interview can be in your local coding environment, and you can use the internet, but no AI assistants: Write a URL crawler, initially synchronous, later make it async, discuss tradeoffs, etc. (Nov 2024)

Hard-level leetcode problems (Oct, 2024)

Was told qkv attention in PyTorch, but was asked about Attention free transformers in a colab (matrix manipulations).

Multiple step coding question, reasonably simple but quite a lot of functionality to implement. Not a complex problem but you have to be able to code somewhat quickly. (Oct 2024).

4-level coding assessment where you're asked to implement a specific system and iteratively add functionalities that requires quick refactoring of the initial implementation (Sep 2024).

In Detail

<https://blog.goncharov.page/i-failed-my-anthropic-interview-and-came-to-tell-you-all-about-it-so-you-dont-have-to>

**1. Online Coding (1.5 hours).**

You had to hack together a class that exposed a public API exactly per the spec. It was a 4-level challenge. A new stage would unlock once you passed all the tests for the current one, which would require you to refactor your code.

I didn’t just say “hack together” for nothing. To pull it off in 90 minutes, you have to code at breakneck speed and completely forget about Big O. Forget about heaps, binary searches, and the like.

I barely managed to get everything running just 2 minutes before the timer finally went off.

There was no human interviewer—only an impersonal, automated system.

How to prepare?

Who the heck knows, honestly. Just code. You don’t need fancy algorithms—in fact, they might even trip you up.

**2. Face-to-Face Coding with a Human (1 hour).**

This round featured a LeetCode medium-level question with a twist. It felt easier than the typical FAANG coding session, where you usually get two different medium-level LeetCode problems.

Preparation? Just the usual grind on LeetCode. Nothing new under the sun.

Around the same time, they also reached out to the people you listed as references and asked them for some written feedback.

**3. Virtual Onsite (Three Parts).**

A marathon stage divided into three segments: a research brainstorm (15 minutes), a take-home assignment with review (5 hours), and a culture fit session (1 hour). You could split it over several days if you felt like it.

At the same time, your references would get an email saying that Anthropic wanted to hop on a call with them! I’ve never seen anything like that before—but in this era of all-out scams, who knows what else might be expected to verify the candidate's background.

**3.1. Research Brainstorm (15 minutes).**

I hopped on a call with the head of alignment. After a quick intro, he posed two open-ended questions designed to elicit ideas. They didn’t require deep insider knowledge of LLMs, just some experience observing them as black boxes and a dash of creativity.

Alas, my creative juices were nowhere to be found that day. I got stuck on the first question, sitting in silence for about three minutes, mentally sifting through my modest math knowledge. Instead of impressing him with my preliminary understanding of linear algebra, I should have generated ideas! Easier said than done—especially when the clock is ticking. I produced a flurry of ideas… but only after I’d taken a stroll around the pond. I needed them right then and there.

By the end of the call, I noticed the interviewer’s bored expression, it was clear that the situation was deteriorating. And then, after about an hour, my references got an automatic message saying there was no need for further conversation, and my access to the candidate portal vanished. The next day, the recruiter wished me luck and canceled the rest of the virtual onsite.

How to prepare?

(Who the heck knows, honestly)^3

I have never been particularly adept at producing brilliant ideas on the spot. I’m much better at letting a problem marinate in my brain, taking a break, and then having a eureka moment—usually while frying an egg or, my favorite, while sitting on the toilet.

In the past, when preparing for FAANG interviews, I’d tackle this by lining up several interviews at different companies. If my in-the-moment creativity is just a roll of the dice, the more rolls the better. Alas, this time managing multiple interviews simultaneously just didn’t work out (I tried, pinky promise!).

**3.2. Take-Home Assignment (5 hours).**

On the day when the take-home assignment was officially canceled, I still received a link, by accident, to a Jupyter Notebook with the task.

They handed over the key to Anthropic’s API and asked me to poke around in the system like it was a mysterious black box, inspired by some research or blog post. At the end, you were supposed to jump on a call and present an overview of your solution.

**3.3. Culture Fit (1 hour).**

Totally terra incognita. They wouldn’t let me in— not even by accident.

**System Design**

As with the other interviews, expect something practical here. It will very likely be related to an issue that Anthropic has encountered before. You might be asked to:

Design a system that enables a GPT to handle multiple questions in a single thread

Design a Claude chat service

Design a banking app

Check out our guide to system design interviews to help you prepare.

**Anthropic Interview Experience**

(from my 2024 search)

**[Software Engineer Interview](https://www.glassdoor.com/Interview/Anthropic-Interview-RVW83919320.htm)**

Jan 31, 2024

**Anonymous Interview Candidate**

If Anthropic hiring is indicative of a world with artificial intelligence, it will be a sad world indeed. First, you receive an invitation to complete a codesignal screening problem. The e-mail is sent from an automated service. If you do less than perfect, you will receive an automated rejection letter (so it seems). If you pass, you move onto a hiring manager screen. This is the first time you get to meet a human being. If you pass, you move onto four 55 minute interviews with 5 minute breaks. Three are technical, one is a personality study. None of them seem to care much about your past experience or expertise, no one will ask. That's where it ends for me. I received a generic rejection letter from a person I had never met or talked to letting me know "it was great meeting you". I understand they must be inundated with applications. And I will have to accept that most of the working world is moving away from a human touch.

**Interview questions:**

Screener: **4 part question** Interview 1: 2 part data mutation question. Interview 2: 3 part sorting question. Interview 3: Many part architecture question. Interview 4: Open-ended questions.

**Anonymous Interview Candidate**

I did an online multi-stage code screen. Implementing a basic version of the spec is pretty easy IMO except for the time constraint. Speed (on a small toy project) seems to be essentially the only thing they're screening on. So much so that typing speed and facility with the online editor are not irrelevant. I also never communicated with a human at any point despite sending a follow-up email that merited some human response IMO.

Question 1

Asked to build core business logic for **a toy banking application**.

**Anonymous Interview Candidate**

Interviewed for a software engineering role. The technical questions were general coding and systems engineering problems. I was discomforted by the culture fit portion, in which the interviewers asked various questions about my **personal beliefs on effective altruism**.

**Anonymous Interview Candidate**

90 minutes to write a 4-tier software application, focusing on banking software & transactions. There was very little communication with a recruiter, as the first step is the coding screen.

**Interview questions**

Develop a banking application from scratch. 1st tier: record and hold transactions (deposits and transfers). 2nd tier: do data metrics, returning the top k accounts with outgoing money 3rd tier: add scheduled transactions and canceling them 4th tier: merge two accounts while maintaining separate account histories

**Anonymous Interview Candidate**

Did a prescreen on Code Signal. Implement some **business logic in 120 minutes**. It had multiple rounds, where the rounds depended on completing previous one. Did not have time to finish it, so I was automatically rejected soon after.

**Anonymous Interview Candidate**

**Design the Claude chat service**.

**Anonymous Interview Candidate**

Code a banking system with several parts of different criteria. Object oriented. One gotcha was **returning n accounts with most total transactions** (deposits and withdrawls), need an algo for **ranking system**. Part 3 was hard. Make sure transfer is accepted before doing funds transfer

**Anonymous Interview Candidate**

Question is NOT leetcode style. Make sure you read Codesignal's general coding framework or industrial coding framework doc. You can google the title and see how they score the candidate.

**Interview questions [1]**

Question 1

A project of 4 different levels with increasing difficulties. Expect to run out of time. Imaging finishing mini-project for a CS class in 90 minutes instead of one week

More: <https://interviewing.io/anthropic-interview-questions>

**Anthropic Online Assessment -- In Memory Database**

Requirements

Your task is to implement a simplified version of an in-memory database. Plan your design according to the level specifications below:

* Level 1: In-memory database should support basic operations to manipulate records, fields, and values within fields.
* Level 2: In-memory database should support displaying a specific record's fields based on a filter.
* Level 3: In-memory database should support TTL (Time-To-Live) configurations on database records.
* Level 4: In-memory database should support backup and restore functionality. To move to the next level, you need to pass all the tests at this level.

Note You will receive a list of queries to the system, and the final output should be an array of strings representing the returned values of all queries. Each query will only call one operation.

Level 1

The basic level of the in-memory database contains records. Each record can be accessed with a unique identifier key of string type. A record may contain several field-value pairs, both of which are of string type.

* SET <key> <field> <value> — should insert a field-value pair to the record associated with key. If the field in the record already exists, replace the existing value with the specified value. If the record does not exist, create a new one. This operation should return an empty string.
* GET <key> <field> — should return the value contained within field of the record associated with key. If the record or the field doesn't exist, should return an empty string.
* DELETE <key> <field> — should remove the field from the record associated with key. Returns if the field was successfully deleted, and "false" if the key or the field do not exist in the database.

Examples

The example below shows how these operations should work:

Queries

queries = [

["SET", "A", "B", "E"],

["SET", "A", "C", "F"],

["GET", "A", "B"],

["GET", "A", "D"],

["DELETE", "A", "B"],

["DELETE", "A", "D"]

]

Explanations

returns ""; database state: {"A": {"B": "E"}}

returns ""; database state: {"A": {"C": "F", "B":"E"}}

returns "E"

returns ""

returns "true"; database state: {"A": {"C": "F"}}

returns "false"; database state: {"A": {"C": "F"}}

Level 2

The database should support displaying data based on filters. Introduce an operation to support printing some fields of a record.

* SCAN <key> — should return a string representing the fields of a record associated with key. The returned string should be in the following format "<field1>(<value1>), <field2>(<value2>), ...", where fields are sorted lexicographically. If the specified record does not exist, returns an empty string.
* SCAN\_BY\_PREFIX <key> <prefix> — should return a string representing some fields of a record associated with key. Specifically, only fields that start with prefix should be included. The returned string should be in the same format as in the SCAN operation with fields sorted in lexicographical order.

Examples

The example below shows how these operations should work

Queries

queries = [

["SET", "A", "BC", "E"],

["SET", "A", "BD", "F"],

["SET", "A", "C", "G"],

["SCAN\_BY\_PREFIX", "A", "B"],

["SCAN", "A"],

["SCAN\_BY\_PREFIX", "B", "B"] ]

Explanations

returns ""; database state: {"A": {"BC": "E"}}

returns ""; database state: {"A": {"BC": "E", "BD": "F"}}

returns ""; database state: {"A": {"BC": "E", "BD": "F", "C": "G"}}

returns "BC(E), BD(F)"

returns "BC(E), BD(F), C(G)"

returns ""

the output should be ["", "", "", "BC(E), BD(F)", "BC(E), BD(F), C(G)", ""].

Level 3

Support the timeline of operations and TTL (Time-To-Live) settings for records and fields. Each operation from previous levels now has an alternative version with a timestamp parameter to represent when the operation was executed. For each field-value pair in the database, the TTL determines how long that value will persist before being removed. Notes:

* Time should always flow forward, so timestamps are guaranteed to strictly increase as operations are executed.
* Each test cannot contain both versions of operations (with and without timestamp). However, you should maintain backward compatibility, so all previously defined methods should work in the same way as before.
* SET\_AT <key> <field> <value> <timestamp> — should insert a field-value pair or updates the value of the field in the record associated with key. This operation should return an empty string.
* SET\_AT\_WITH\_TTL <key> <field> <value> <timestamp> <ttl> — should insert a field-value pair or update the value of the field in the record associated with key. Also sets its Time-To-Live starting at timestamp to be ttl. The ttl is the amount of time that this field-value pair should exist in the database, meaning it will be available during this interval: [timestamp, timestamp + ttl). This operation should return an empty string.
* DELETE\_AT <key> <field> <timestamp> — the same as DELETE, but with timestamp of the operation specified. Should return "true" if the field existed and was successfully deleted and "false" if the key didn't exist.
* GET\_AT <key> <field> <timestamp> — the same as GET, but with timestamp of the operation specified.
* SCAN\_AT <key> <timestamp> — the same as SCAN, but with timestamp of the operation specified.
* SCAN\_BY\_PREFIX\_AT <key> <prefix> <timestamp> — the same as SCAN\_BY\_PREFIX, but with timestamp of the operation specified.

Examples

The examples below show how these operations should work

Queries

queries = [

["SET\_AT\_WITH\_TTL", "A", "BC", "E", "1", "9"],

["SET\_AT\_WITH\_TTL", "A", "BC", "E", "5", "10"],

["SET\_AT", "A", "BD", "F", "5"],

["SCAN\_BY\_PREFIX\_AT", "A", "B", "14"],

["SCAN\_BY\_PREFIX\_AT", "A", "B", "15"]

]

Explanations

returns ""; database state: {"A": {"BC": "E"}}

where {"BC": "E"} expires at timestamp 10 returns ""; database state: {"A": {"BC": "E"}}

as field "BC" in record "A" already

exists, it was overwritten,

and {"BC": "E"} now expires at timestamp 15

returns ""; database state: {"A": {"BC": E", "BD": "F"}}

where {"BD": "F"} does not expire

returns "BC(E), BD(F)"

returns "BD(F)"

the output should be ["", "", "", "BC(E), BD(F)", "BD(F)"].

Example2

Queries

queries = [

["SET\_AT", "A", "B", "C",

"1"],

["SET\_AT\_WITH\_TTL", "X",

"Y", "Z", "2", "15"],

["GET\_AT", "X", "Y", "3"], ["SET\_AT\_WITH\_TTL", "A",

"D", "E", "4", "10"],

["SCAN\_AT", "A", "13"],

["SCAN\_AT", "X", "16"],

["SCAN\_AT", "X", "17"],

["DELETE\_AT", "X", "Y",

"20"]

]

Explanations

returns ""; database state: {"A": {"B": "C"}} returns ""; database state: {"X": {"Y": "Z"}, "A": {"B": "C"}}

where {"Y": "Z"} expires at timestamp 17 returns "Z"

returns ""; database state:

{"X": {"Y": "Z"}, "A": {"D": "E", "B": "C"}} where {"D": "E"} expires at timestamp 14 and {"Y": "Z"} expires at timestamp 17

returns "B(C), D(E)"

returns "Y(Z)"

returns ""; Note that all fields in record "X" have expired

returns "false"; the record "X" was expired at timestamp 17 and can't be deleted.

the output should be ["", "", "Z", "", "B(C), D(E)", "Y(Z)", "", "false"].

Level 4

The database should be backed up from time to time. Introduce operations to support backing up and restoring the database state based on timestamps. When restoring, ttl expiration times should be recalculated accordingly.

* BACKUP <timestamp> — should save the database state at the specified timestamp, including the remaining ttl for all records and fields. Remaining ttl is the difference between their initial ttl and their current lifespan (the duration between the timestamp of this operation and their initial timestamp). Returns a string representing the number of non-empty non-expired records in the database.
* RESTORE <timestamp> <timestampToRestore> — should restore the database from the latest backup before or at timestampToRestore. It's guaranteed that a backup before or at timestampToRestore will exist. Expiration times for restored records and fields should be recalculated according to the timestamp of this operation - since the database timeline always flows forward, restored records and fields should expire after the timestamp of this operation, depending on their remaining ttls at backup. This operation should return an empty string.

Examples

Queries

queries = [

["SET\_AT\_WITH\_TTL", "A", "B", "C", "1", "10"],

["BACKUP", "3"],

["SET\_AT", "A", "D", "E", "4"],

["BACKUP", "5"],

["DELETE\_AT", "A", "B",

"8"],

["BACKUP", "9"],

["RESTORE", "10", "7"],

["BACKUP", "11"],

["SCAN\_AT", "A", "15"],

["SCAN\_AT", "A", "16"]

]

Explanations

returns ""; database state: {"A": {"B": "C"}} with lifespan `[1, 11)`, meaning that the record should be deleted at timestamp = 11.

returns "1"; saves the database state

returns ""; database state: {"A": {"D": "E", "B": "C"}}

returns "1"; saves the database state

returns "true"; database state: {"A": {"D": "E"}} returns "1"; saves the database state

returns ""; restores the database to state of last backup at timestamp = 5:

{"A": {"D": "E", "B": "C"}}

with {"B": "C"} expiring at timestamp = 16: Since the initial ttl of the field is 10

and the database was restored to the state at timestamp = 5; this field has had

a lifespan of 4 and a remaining ttl of 6, so it will now expire at timestamp = 10 + 6 = 16. returns "1"; saves the database state

returns "B(C), D(E)"

returns "D(E)"

the output should be ["", "1", "", "1", "true", "1", "", "1", "B(C), D(E)", "D(E)"].

**Anthropic Online Assessment -- Cloud Storage System**

Instructions

Your task is to implement a simple cloud storage system. All operations that should be supported are listed below.

Solving this task consists of several levels. Subsequent levels are opened when the current level is correctly solved. You always have access to the data for the current and all previous levels.

Requirements

Your task is to implement a simple cloud storage system that maps objects (files) to their metainformation. Specifically, the storage system should maintain files along with some information about them (name, size, etc.). **Note that this system should be in-memory, you do not need to work with the real filesystem.**

Plan your design according to the level specifications below:

* **Level 1**: The cloud storage system should support adding new files, retrieving, and copying files.
* **Level 2**: The cloud storage system should support finding files by matching prefixes and suffixes.
* **Level 3**: The cloud storage system should support adding users with various capacity limits.
* **Level 4**: The cloud storage system should support compressing and decompressing files.

To move to the next level, you need to pass all the tests at this level.

**Note**

You will receive a list of queries to the system, and the final output should be an array of strings representing the returned values of all queries. Each query will only call one operation.

It is guaranteed that the given queries will never call operations that result in collisions between file and directory names.

Level 1

The cloud storage system should support operations to add files, copy files, and get files stored on the system.

ADD\_FILE **<name>** **<size>**

Should add a new file **name** to the storage. **size** is the amount of memory required in bytes.  
The current operation fails if a file with the same **name** already exists.  
Returns **"true"** if the file was added successfully or **"false"** otherwise.

COPY\_FILE **<nameFrom>** **<nameTo>**

Should copy the file at **nameFrom** to **nameTo**.  
The operation fails if **nameFrom** points to a file that does not exist or points to a directory.  
The operation fails if the specified file already exists at **nameTo**.  
Returns **"true"** if the file was copied successfully or **"false"** otherwise.

GET\_FILE\_SIZE **<name>**

Should return a string representing the size of the file **name** if it exists, or an empty string otherwise.

**Examples**

The example below shows how these operations should work (the section is scrollable to the right):

queries = [

["ADD\_FILE", "/dir1/dir2/file.txt", "10"],

["COPY\_FILE", "/not-existing.file", "/dir1/file.txt"],

["COPY\_FILE", "/dir1/dir2/file.txt", "/dir1/file.txt"],

["ADD\_FILE", "/dir1/file.txt", "15"],

["COPY\_FILE", "/dir1/file.txt", "/dir1/dir2/file.txt"],

["GET\_FILE\_SIZE", "/dir1/file.txt"],

["GET\_FILE\_SIZE", "/not-existing.file"] ]

Explanations:

- returns `"true"`; adds file `"/dir1/dir2/file.txt"` of 10 bytes

- returns `"false"`; the file `"/not-existing.file"` does not exist

- returns `"true"`; adds file `"/dir1/file.txt"` of 10 bytes

- returns `"false"`; the file `"/dir1/file.txt"` exists already

- returns `"false"`; the file `"/dir1/dir2/file.txt"` exists already

- returns `"10"`

- returns `""`; the file `"/not-existing.file"` does not exist

The output should be:

["true", "false", "true", "false", "false", "10", ""]

Level 2

Implement support for retrieving file names by searching directories via prefixes and suffixes.

FIND\_FILE **<prefix>** **<suffix>**

Should search for files with names starting with **prefix** and ending with **suffix**.  
Returns a string representing all matching files in this format:

"<name1>(<size1>), <name2>(<size2>), ..."

The output should be sorted in descending order of file sizes or, in the case of ties, **lexicographically**. If no files match the required properties, should return an empty string.

Examples

The example below shows how these operations should work (the section is scrollable to the right):

queries = [ ["ADD\_FILE", "/root/dir/another\_dir/file.mp3", "10"],

["ADD\_FILE", "/root/file.mp3", "5"],

["ADD\_FILE", "/root/music/file.mp3", "7"],

["COPY\_FILE", "/root/music/file.mp3", "/root/dir/file.mp3"],

["FIND\_FILE", "/root", ".mp3"],

["FIND\_FILE", "/root", "file.txt"],

["FIND\_FILE", "/dir", "file.mp3"] ]

Explanations:

- returns `"true"`

- returns `"true"`

- returns `"true"`

- returns `"true"`

- returns `"/root/dir/another\_dir/file.mp3(10), /root/dir/file.mp3(7), /root/music/file.mp3(7), /root/file.mp3(5)"`

- returns `""`; there is no file with the prefix `"/root"` and suffix `"file.txt"`

- returns `""`; there is no file with the prefix `"/dir"` and suffix `"file.mp3"`

The output should be:

["true", "true", "true", "true", "/root/dir/another\_dir/file.mp3(10), /root/dir/file.mp3(7), /root/music/file.mp3(7), /root/file.mp3(5)", "", ""]

Level 3

Implement support for different users sending queries to the system. All users share a common filesystem in the cloud storage, but each user is assigned an individual storage capacity limit.

ADD\_USER **<userId>** **<capacity>**

Should add a new user to the system, with **capacity** as their storage limit in bytes.  
The total size of all files owned by **userId** cannot exceed **capacity**.  
The operation fails if a user with **userId** already exists.  
Returns **"true"** if a user with **userId** is successfully created, or **"false"** otherwise.

ADD\_FILE\_BY **<userId>** **<name>** **<size>**

Should behave in the same way as the **ADD\_FILE** from Level 1, but the added file should be owned by the user with **userId**.  
A new file cannot be added to the storage if doing so will exceed the user’s **capacity** limit.  
Returns a string representing the remaining storage capacity for **userId** if the file is successfully added or an empty string otherwise.

***Note*** *All queries calling the****ADD\_FILE****operation implemented during Level 1 are run by the user with****userId = "admin"****, who has unlimited storage capacity. Also, assume that the****COPY\_FILE****operation preserves the ownership of the original file.*

UPDATE\_CAPACITY **<userId>** **<capacity>**

Should change the maximum storage capacity for the user with **userId**.  
If the total size of all user’s files exceeds the new **capacity**, the largest files (sorted **lexicographically** in case of a tie) should be removed from the storage until the total size of all remaining files will no longer exceed the new **capacity**.  
Returns a string representing the number of removed files, or an empty string if a user with **userId** does not exist.

Examples

The example below shows how these operations should work (the section is scrollable to the right):

queries = [

["ADD\_USER", "user1", "125"],

["ADD\_USER", "user1", "100"],

["ADD\_USER", "user2", "100"],

["ADD\_FILE\_BY", "user1", "/file.med", "30"],

["ADD\_FILE\_BY", "user2", "/file.med", "40"],

["COPY\_FILE", "/file.med", "/dir/another/file.med"],

["COPY\_FILE", "/file.med", "/file.small", "10"],

["ADD\_FILE\_BY", "admin", "/dir/file\_small", "5"],

["ADD\_FILE\_BY", "user1", "/my\_folder/file.huge", "100"],

["ADD\_FILE\_BY", "user3", "/my\_folder/file.huge", "100"],

["UPDATE\_CAPACITY", "user1", "300"],

["UPDATE\_CAPACITY", "user1", "50"],

["UPDATE\_CAPACITY", "user2", "1000"]

]

Explanations:

1. returns `"true"`; creates user `"user1"` with 125 bytes capacity

2. returns `"false"`; `"user1"` already exists

3. returns `"true"`; creates user `"user2"` with 100 bytes capacity

4. returns `"75"`

5. returns `""`; file named `"/file.med"` already exists and owned by `"user1"`

6. returns `"true"`; copying preserves the file owner. After copying, `"user1"` has 15 capacity left

7. returns `"false"`; `"user1"` does not have enough storage capacity left to perform copying operation

8. returns `"true"`; this operation is done by `"admin"` with unlimited capacity

9. returns `"false"`; `"user1"` does not have enough storage capacity left to add this file

10. returns `""`; `"user3"` doesn't exist

11. returns `"0"`; all files owned by `"user1"` can fit into the new capacity of 300 bytes

12. returns `"2"`; the files `"/dir/file.big"` and `"/dir/another/file.med"` should be deleted so the remaining files owned by `"user1"` can fit into the new capacity of 50 bytes

13. returns `""`; `"user2"` doesn't exist

The output should be:

["true", "false", "true", "75", "", "true", "false", "true", "false", "", "0", "2", ""]

Level 4

Implement support for file compression.

COMPRESS\_FILE **<userId>** **<name>**

Should compress the file **name** if it belongs to **userId**.  
The compressed file should be replaced with a new file named **<name>.COMPRESSED**.  
The size of the newly created file should be equal to half of the original file. The size of all files is guaranteed to be even, so there should be no fractional calculations.  
It is also guaranteed that **name** for this operation never points to a compressed file (i.e., it never ends with **.COMPRESSED**).  
Compressed files should be owned by **userId** — the owner of the original file.  
Returns a string representing the remaining storage capacity for **userId** if the file was compressed successfully or an empty string otherwise.

***Note*** *Because file names can only contain lowercase letters, compressed files cannot be added via****ADD\_FILE****.  
It is guaranteed that all****COPY\_FILE****operations will preserve the suffix****.COMPRESSED****.*

DECOMPRESS\_FILE **<userId>** **<name>**

Should revert the compression of the file **name** if it belongs to **userId**.  
It is guaranteed that **name** for this operation always ends with **.COMPRESSED**.  
If decompression results in the **userId** exceeding their storage capacity limit **or** a decompressed version of the file with the given name already exists, the operation fails.  
Returns a string representing the remaining capacity of **userId** if the file was decompressed successfully or an empty string otherwise.

Examples

The example below shows how these operations should work (the section is scrollable to the right):

queries = [

["ADD\_USER", "user1", "1000"],

["ADD\_USER", "user2", "5000"],

["ADD\_FILE\_BY", "user1", "/dir/file.mp4", "500"],

["ADD\_FILE\_BY", "user2", "/dir/file.mp4", "1"],

["COMPRESS\_FILE", "user3", "/dir/file.mp4"],

["COMPRESS\_FILE", "user1", "/folder/non\_existing\_file"],

["COMPRESS\_FILE", "user1", "/dir/file.mp4"],

["COMPRESS\_FILE", "user1", "/dir/file.mp4"],

["GET\_FILE\_SIZE", "/dir/file.mp4.COMPRESSED"],

["GET\_FILE\_SIZE", "/dir/file.mp4"],

["COPY\_FILE", "/dir/file.mp4.COMPRESSED", "/file.mp4.COMPRESSED"],

["ADD\_FILE\_BY", "user1", "/dir/file.mp4", "500"],

["DECOMPRESS\_FILE", "user1", "/dir/file.mp4.COMPRESSED"],

["UPDATE\_CAPACITY", "user1", "2000"],

["DECOMPRESS\_FILE", "user2", "/dir/file.mp4.COMPRESSED"],

["DECOMPRESS\_FILE", "user3", "/dir/file.mp4.COMPRESSED"],

["DECOMPRESS\_FILE", "user1", "/dir/file.mp4.COMPRESSED"]

]

Explanations:

1. returns `"true"`

2. returns `"true"`

3. returns `"500"`

4. returns `""`; the file `"/dir/file.mp4"` is owned by `"user1"`

5. returns `""`; `"user3"` doesn’t exist

6. returns `""`; the file `"/folder/non\_existing\_file"` doesn’t exist

7. returns `"750"`; the file `"/dir/file.mp4"` is compressed to size = 500 / 2 = 250 bytes

8. returns `""`

9. returns `"250"`

10. returns `""`

11. returns `"true"`

12. returns `"0"`; `"user1"` does not have enough storage capacity to decompress the file

13. returns `"0"`

14. returns `"true"`; the file `"/dir/file.mp4.COMPRESSED"` is owned by `"user1"`

15. returns `""`; `"user3"` doesn’t exist

16. returns `""`; the file `"/dir/file.mp4"` exists already

17. returns `"750"`

The output should be:

["true", "true", "500", "", "", "", "750", "", "250", "", "true", "0", "0", "true", "", "", "750"]