Other Topics in SE

Recursion:

1.What is the potential consequence of not including a base case in a recursive function?

Ans:Defining a **base case in recursive functions** is crucial because it determines when the recursion should **stop**. Without a base case, the function would continue calling itself indefinitely, leading to stack overflow errors and an **infinite loop**.

2.What does memoization allow you to avoid?

Ans:Memoization ensures that a method **doesn't run for the same inputs more than once** by keeping a record of the results for the given inputs (usually in a hash map).

#What are the benefits of memoization?

Memoization is a technique that can help you **speed up your algorithms** by storing and reusing the results of previous computations. It is especially useful for recursive or dynamic programming problems, where the **same subproblems are solved repeatedly**.

3.What are the steps to writing a recursive function?

Ans.Create a **regular function with a base case** that can be reached with its parameters.

- Pass arguments into the function that **immediately triggers** the base case.
- Pass the next arguments that trigger the recursive call just once.

```
function log(num){
    if(num > 5){
        return;
    }
    console.log(num);
    log(num + 1);
}

log(1);
```

A recursive function must have at **least one condition where it will stop calling itself**, or the function will call itself indefinitely until JavaScript throws an error.The condition that stops a recursive function from calling itself is known as **the base case**. In the log function above, the base case is when num is larger than 5.

Why don't you just use a loop?

Modern programming languages like JavaScript already have the **for and while statements** as **alternatives to recursive functions**. But some languages like Clojure do not have any looping statements, so you need to use recursion to repeatedly execute a piece of code.

Linked Lists:

1.Why would we use a Linked List instead of an array?

Ans**:Arrays** are best suited for situations where the **size of the collection is known in advance** and where elements need to be accessed or manipulated quickly. **Linked lists** are more **flexible and adaptable** and are best suited for situations where the size of the collection is not known.

What are two characteristics of linked lists?

Properties of Linked List: The linked list starts with a HEAD which denotes the starting point or the memory location of the first node.Linked list ends with the last node pointing to NULL value.Unlike array the elements are not stored in contiguous memory locations, but are stored in random location

What Are the Types of Linked Lists?

1.Singly linked lists.

2.Doubly linked lists.

3.Circular linked lists.

2.When we want to add a node to a linked list, do we have to scoot over the subsequent nodes (like we do for arrays)?

Ans:If there are one or more nodes in a linked list, it is a non-empty linked list. To append a node to a non-empty linked list, make the last node link to the new node. Unlike arrays, we cannot access any elements in a linked list directly. We must traverse from the head node to the last node.

How do you add a node to a linked list?

There is a four step process to insert a new node at the head of the linked list:

- Allocate memory for the new node.
- Put our data into the new node.
- Set Next of the new node to point to the previous Head.
- Reset Head to point to the new node.

3.Can we do index access like we can with arrays with linked lists?

Ans: No, you can't access elements in a linked list using index. Index based access can be implemented where memory allocation to elements is contiguous or on some fixed pattern. But the linked list works on random allocation.

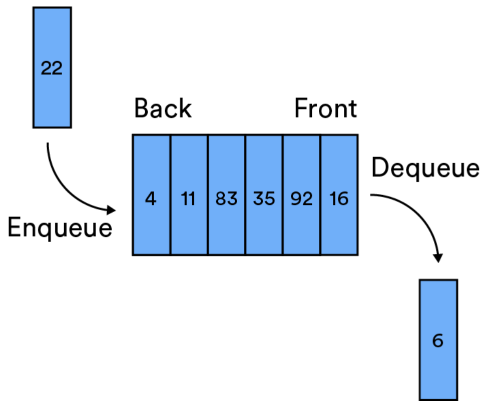
Stacks and Queues:

1.What are Stacks & Queues?

Stacks and queues in computer science are a lot like stacks and queues in real life. It helps if you think of stacks of pancakes and queues, or lines, of people.

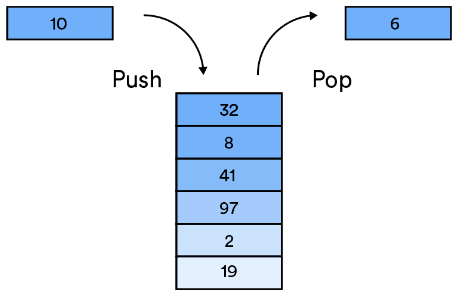
Queues:

Queues operate on "first-in, first-out" (aka, FIFO) behavior. Items are removed in the order they were added, from first to last.



Stacks:

A stack is defined by last-in, first-out behavior — the opposite of how a queue works. The last thing added to the stack — the freshest pancake — is the first thing to be removed.



The function call stack is a common example of stacks in programming. When you call a function to execute, it's pushed to the top of the stack and runs until we add another function to the stack, which then runs until it returns (or another function is pushed to the top), on a last in, first out basis. You can keep adding functions until you've run out of space in the stack, in which case you've reached stack overflow.

What data structure would be represented by the 'back' button in the browser?
This is referred to as last-in, first-out (LIFO). The back button in web browsers is a good example: each page you view is added to the stack, and when you click back, the current page (the last one added) is popped from the stack.
What data structure would be represented by sending documents to the printer? Queue data structure?

Printers: Queue data structure is used in printers to maintain the order of pages while printing. Interrupt handling in computers: The interrupts are operated in the same order as they arrive, i.e., interrupt which comes first, will be dealt with first.

What data structure would be represented by the 'undo' or Cmd-Z function?

Linear undo is implemented with a stack (last in first out (LIFO) data structure) that stores a history of all executed commands. When a new command is executed it is added to the top of the stack. Therefore, only the last executed command can be undone and removed from history.

Implementing Stacks & Queues:

The other thing that stacks and queues have in common is that they can both be implemented as an array or as a linked list.

The major difference between a linked list and an array is how they store data in a computer's memory?
In terms of memory allocation, **arrays require a contiguous block of memory**, which can be a limitation if large arrays are needed and memory is fragmented. **Linked lists**, however, do not require contiguous memory, as each node **can be stored anywhere in memory**.

Which of the following statements is true about how linked lists and arrays store data?

Ans:Linked lists can store data anywhere in a computer's memory using pointers.

Queue Variations: 1.Priority Queues; 2.Double-Ended Queues

These have three rules in addition to regular queues:

1. Every item has a priority associated with it.
2. An element with high priority is dequeued before an element with low priority.
3. If two elements have the same priority, they are served according to their order in the queue.

Examples: Airport priority boarding, CPU scheduling.

Double-Ended Queues

A double-ended queue, or "deque", performs insertions and deletions at both ends of the queue. They're usually implemented with doubly-linked lists or dynamic arrays.

Have you ever been last in a long line at the grocery store only to see a cashier one lane over to open up their register? Typically, that cashier will beckon to you to jump into their lane so you can be checked out right away. That’s basically what happens in a deque.

Example: spreading tasks between different servers as equally as possible

Q.You're working on building a message board and want to display only the 10 most recent messages a user posted, in the order they were posted. Which data structure should you use?

Ans:We use arrays and Stacks data structures

Q:You and your partner are going out for your anniversary dinner. When you arrive, you ask the host for a table for two and your name goes on the waiting list. While you’re waiting, a group of seven people walks right in and gets seated. What gives?!

Ans:Double-Ended Queues: Have you ever been last in a long line at the grocery store only to see a cashier one lane over to open up their register? Typically, that cashier will beckon to you to jump into their lane so you can be checked out right away. That’s basically what happens in a deque.

Testing:

1.Why do we create tests?

As programmers, we use code to solve problems. Most libraries and frameworks have testing libraries available that let us write code to evaluate the robustness, completeness and flexibility of our applications. In a production-level application, providing a high level of test coverage for an application is almost always required in order to guarantee that code is bug-free and functions as intended.

There are many types of tests that we can create for our applications:

- **Unit tests**: the smallest, **most microscopic level of testing**. Evaluates individual methods and functions within a codebase. The kind we'll be writing today!
- **Integration tests**: ensure that different services and modules work together.
- **End-to-end tests:** verify that application responds as expected to user interactions, such as evaluating how user input edge cases are handled.
- **Performance tests**: also known as load testing, and evaluate application's response to heavy traffic (number of requests, large amounts of data).
- **Acceptance tests**: ensure that the application meets its given business requirements.

TDD: Test-Driven Development:

JavaScript Testing with Mocha & Chai:

Mocha will be our testing framework, but we're mostly just using it as a test runner.
"Mocha is a **feature-rich JavaScript test framework** running on Node.js and the browser, making asynchronous testing simple and fun. Mocha tests run serially, allowing for flexible and accurate reporting, while mapping (associating) uncaught exceptions to the correct test cases."
For assertions, we will use Chai:
"Chai is a BDD / TDD assertion library for Node and the browser that can be delightfully paired with any JavaScript testing framework."
Q: What the heck is an assertion?
Ans: It's a way of writing a unit test that tests whether or not a test case is passing or failing by comparing the expected result with the actual result of a test.
2.What is an API? What is API Testing?

APIs are mechanisms that enable two software components to communicate with each other using a set of definitions and protocols. For example, the weather bureau’s software system contains daily weather data. The weather app on your phone “talks” to this system via APIs and shows you daily weather updates on your phone.
Websocket APIs
Websocket API is another modern web API development that uses JSON objects to pass data. A WebSocket API supports two-way communication between client apps and the server. The

server can send callback messages to connected clients, making it more efficient than REST API.

What are REST APIs?

REST stands for Representational State Transfer. REST defines a set of functions like GET, PUT, DELETE, etc. that clients can use to access server data. Clients and servers exchange data using HTTP.

What is API Testing?

Let’s distinguish between three kinds of test flows which comprise our test plan:

1. **Testing requests in isolation** – Executing a single API request and checking the response accordingly. Such basic tests are the minimal building blocks we should start with, and there’s no reason to continue testing if these tests fail.
2. **Multi-step workflow with several requests** – Testing a series of requests which are common user actions, since some requests can rely on other ones. For example, we execute a POST request that creates a resource and returns an auto-generated identifier in its response. We then use this identifier to check if this resource is present in the list of elements received by a GET request. Then we use a PATCH endpoint to update new data, and we again invoke a GET request to validate the new data. Finally, we DELETE that resource and use GET again to verify it no longer exists.
3. **Combined API and web UI tests –** This is mostly relevant to manual testing, where we want to ensure data integrity and consistency between the UI and API.

RESTful HTTP Methods:

Method	Crud functionality	DB Action
GET	**read**	**retrieve data**
POST	**create**	**add data**
PUT	**update**	**modify existing data**
PATCH	**update**	**modify existing data**
DELETE	**delete**	**delete existing data**