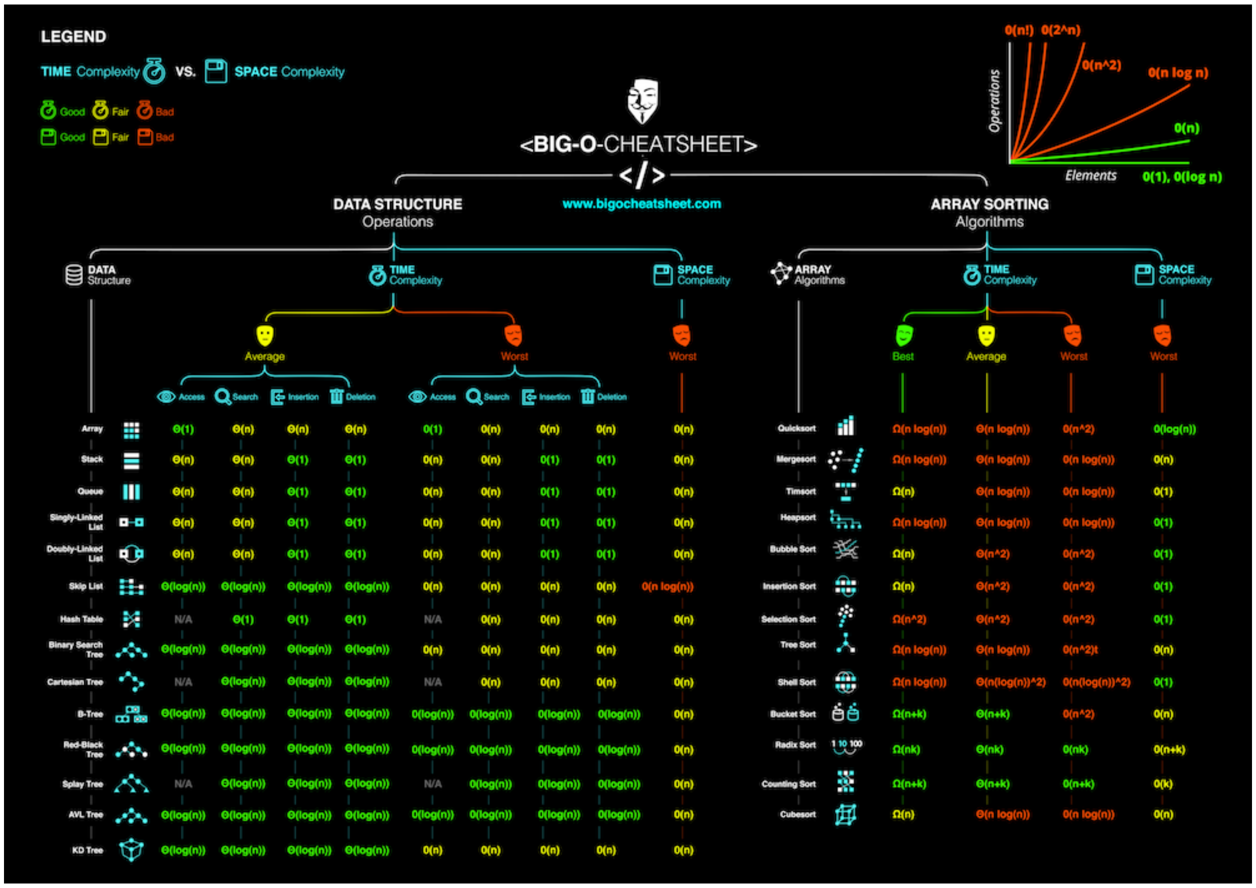
Module 13 Day 2

Lesson 1

Big O Notation

https://flexiple.com/algorithms/big-o-notation-cheat-sheet/


LEGEND
TIME Complexity vs. SPACE Complexity
Good Fair Bad
Good Fair Bad
<BIG-O-CHEATSHEET>
</>
www.bigocheatsheet.com
O(n!) O(2^n)
O(n^2)
O(n log n)
O(n)
O(1), O(log n)
Operations
Elements
DATA STRUCTURE Operations
ARRAY SORTING Algorithms

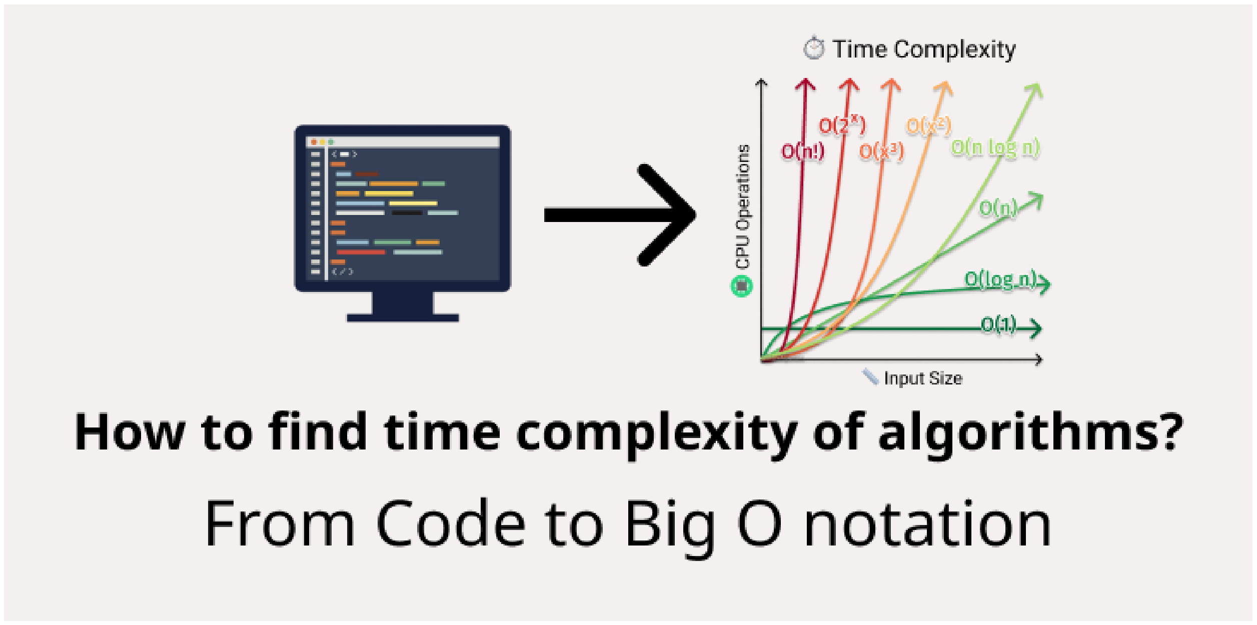
DATA Structure	TIME Complexity: Average — Access	Average — Search	Average — Insertion	Average — Deletion	Worst — Access	Worst — Search	Worst — Insertion	Worst — Deletion	SPACE Complexity: Worst
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

ARRAY Algorithms	TIME Complexity: Best	Average	Worst	SPACE Complexity: Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(1)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)t	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Source: https://www.bigocheatsheet.com/

What is it?

Big O Notation is a way to describe the efficiency of an algorithm in terms of how fast it grows as the size of the problem it solves increases. It is commonly used in computer science to compare the efficiency of different algorithms. It looks at efficiency in terms of two categories: time complexity and space complexity.



Source:
https://adrianmejia.com/how-to-find-time-complexity-of-an-algorithm-code-big-o-notation/

Why?

Big O notation can help you optimize your code and ensure that your applications are efficient and scalable.

- Time Complexity Analysis: Understanding big O notation can help you analyze the time complexity of algorithms and data structures used in your web applications. This knowledge can help you optimize the performance of your code and ensure that your applications can handle large amounts of data efficiently.
- Scaling Web Applications: As your web application grows and more users access it simultaneously, the performance of your application becomes a critical factor. Understanding the time complexity of different algorithms and data structures can help you make informed decisions about how to scale your application to handle more traffic.
- Code Optimization: Optimizing code is an essential part of web development. Knowing the time complexity of different algorithms can help you identify potential bottlenecks in your code and optimize it to improve its performance.

Use Cases

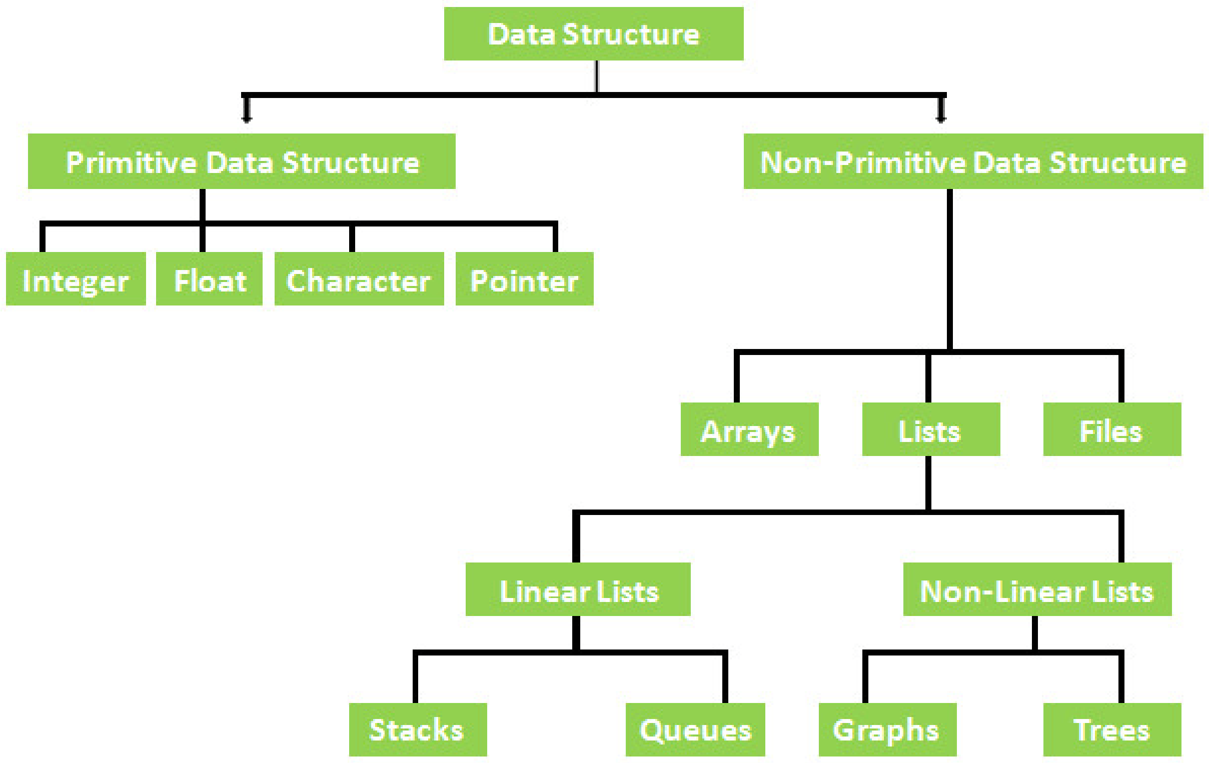
- Searching and Sorting Algorithms: When you search for a particular item in a database or sort a large collection of data, the time it takes to complete the operation depends on the size of the data set. The algorithms used for these tasks often have different time complexities, and understanding big O notation can help you choose the most efficient algorithm for your use case.
- Database Operations: Database operations can involve complex queries and data manipulation tasks. The time it takes to complete these operations depends on the size of the data set and the complexity of the queries. Understanding big O notation can help you optimize your queries and ensure that they complete in a reasonable amount of time, even for large data sets.
- Machine Learning Algorithms: Machine learning algorithms are used for a wide range of tasks, such as image recognition, natural language processing, and predictive analytics. These algorithms often involve processing large amounts of data, and their time complexity can vary depending on the algorithm used. Understanding big O notation can help you choose the most efficient algorithm for your machine learning task.
- Web Applications: Web applications involve a lot of data processing and manipulation, such as parsing user input, generating dynamic content, and querying databases. The time it takes to complete these tasks depends on the size of the data set and the complexity of the operations. Understanding big O notation can help you optimize your code and ensure that your web application can handle large amounts of traffic without slowing down.

Careers

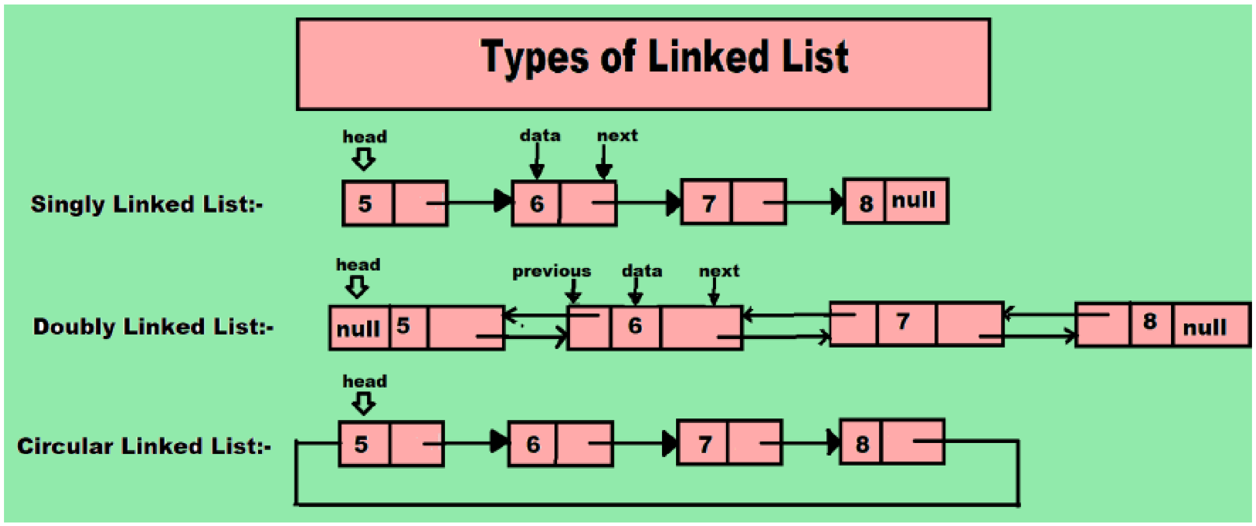
- Software Developer: As a software developer, you may need to optimize algorithms and data structures to improve the performance of your code. Understanding big O notation can help you analyze the time complexity of your code and choose the most efficient algorithms and data structures for your use case.
- Data Scientist: Data scientists use various algorithms and statistical models to analyze and process large datasets. Understanding big O notation can help you choose the most efficient algorithms for your data analysis tasks and optimize your code for faster processing.
- Database Administrator: As a database administrator, you may need to optimize queries and data manipulation tasks for faster processing. Understanding big O notation can help you choose the most efficient data structures and algorithms for your queries and ensure that your database can handle large amounts of data without slowing down.
- Machine Learning Engineer: Machine learning engineers develop and optimize algorithms and models for predictive analytics and data processing. Understanding big O notation can help you choose the most efficient algorithms for your machine learning tasks and optimize your code for faster processing.

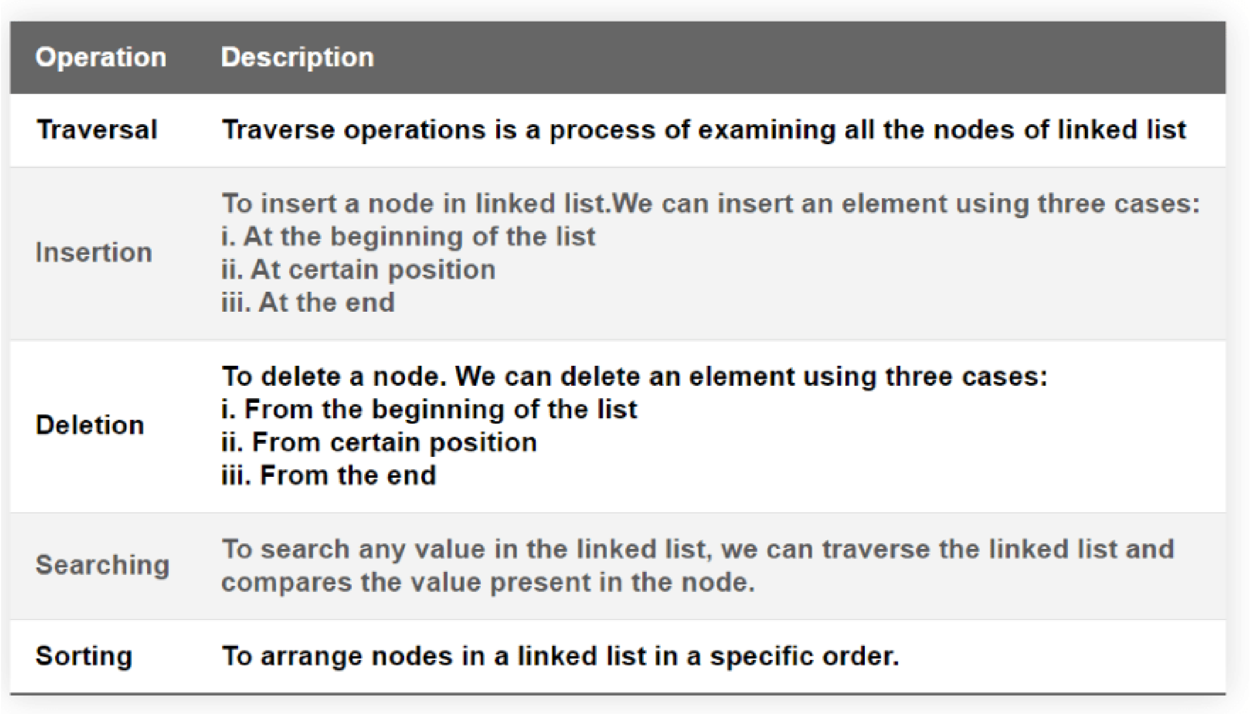
Lesson 2

Data Structures



Source: https://www.fiverr.com/alinawebdev/teach-you-data-structures-in-c-java-or-javascript





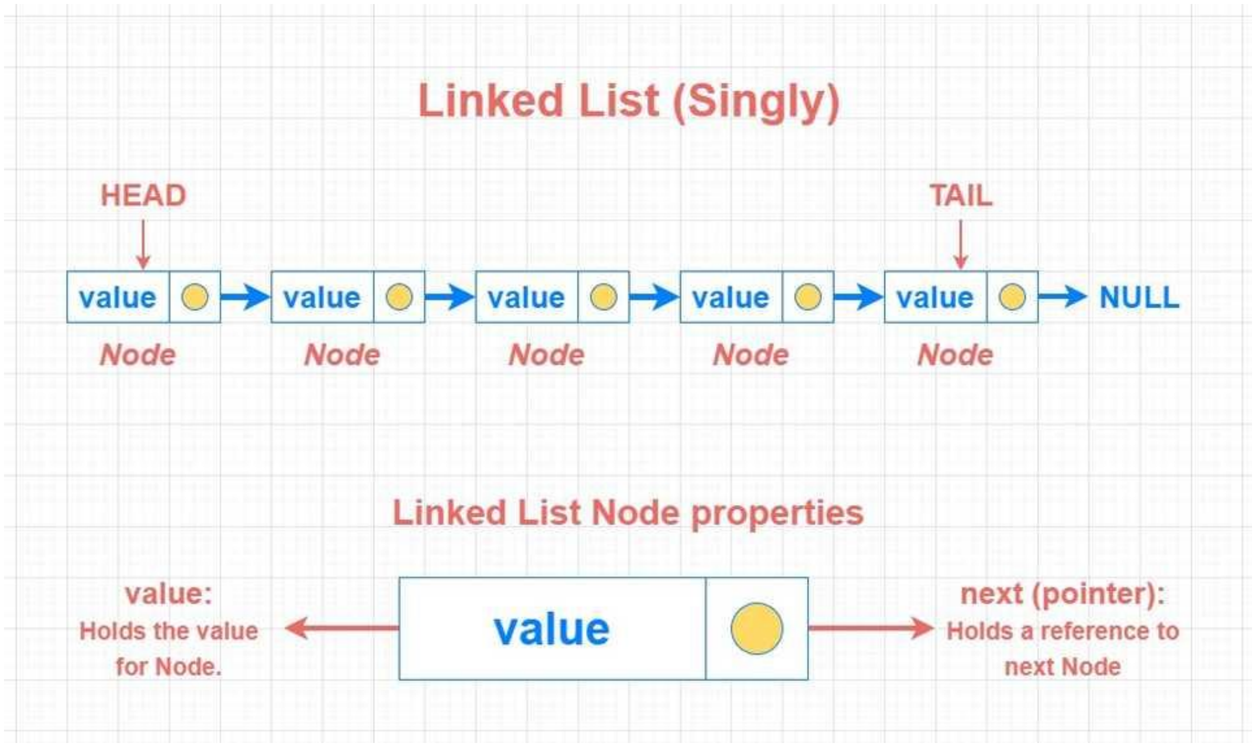
Operation	Description
Traversal	Traverse operations is a process of examining all the nodes of linked list
Insertion	To insert a node in linked list.We can insert an element using three cases: i. At the beginning of the list ii. At certain position iii. At the end
Deletion	To delete a node. We can delete an element using three cases: i. From the beginning of the list ii. From certain position iii. From the end
Searching	To search any value in the linked list, we can traverse the linked list and compares the value present in the node.
Sorting	To arrange nodes in a linked list in a specific order.

Source: https://dev.to/nehasoni__/7-javascript-data-structures-you-must-know-57ah

Singly and doubly linked lists:

A linked list is a sequence of elements where each element is linked to its next (and/or previous) element via a pointer. Linked lists are commonly used to implement dynamic data structures such as stacks, queues, and hash tables.

Singly linked list



A Linked List is consisted by a series of connected Nodes. Each Node contains 2 properties:

Value: Holds the value / data for the Node.

Next (pointer): Holds a reference (pointer) to the next Node.

We also have specific names for the first and the last node in the list. We call the first node "HEAD" and the last node "TAIL". As you see above, tail node points to a null value - which means Linked Lists are *"null terminated"*. In simpler words, this is how we know we are at the end of a Linked List.

Source:
https://www.sahinarslan.tech/posts/deep-dive-into-data-structures-using-javascript-linked-list

Real World Example- To-do list: A to-do list can be implemented as a singly linked list, with each task being represented by a node and the next node pointing to the next task in the list. This allows for easy traversal of the list and adding/removing tasks.

Codepen Example: https://codepen.io/shafferma08/pen/KKGvVgz

https://www.freecodecamp.org/news/implementing-a-linked-list-in-javascript/

Stacks:

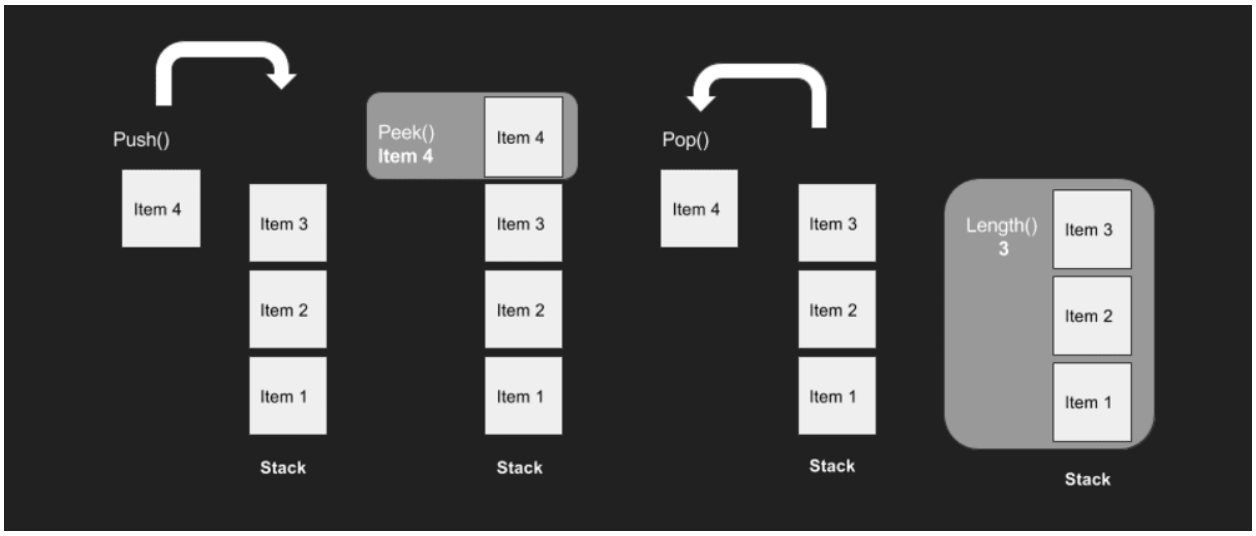
Examples of Stacks in Real Life

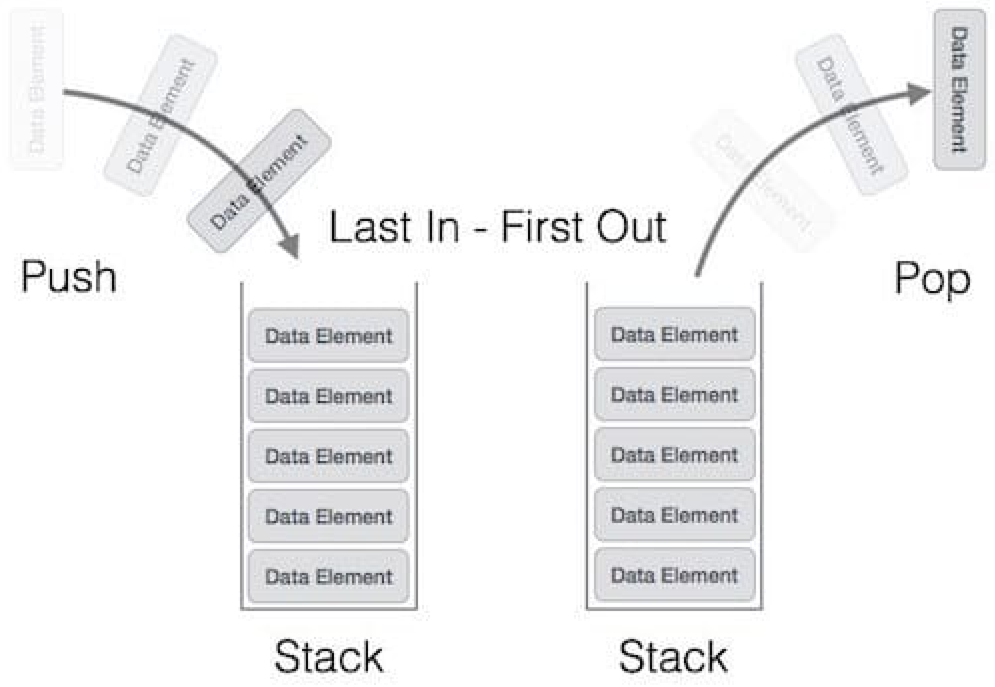
Data Structures and Programming Techniques

3

Operation	Description
push()	The process of inserting a new element in stack is known as push operation. If the stack is full then the overflow condition occurs.
pop()	The process of removing an element from the stack is known as pop operation. If the stack is empty then the underflow condition occurs.
isEmpty()	This method is used to check whether the stack is empty or not.
isFull()	This method is used to check whether the stack is full or not.
peek()	It is used to retrieve the first element of the Stack or the element present at the top of the Stack

Definition: A stack is a collection of elements that supports two main operations: push (adds an element to the top of the stack) and pop (removes the top element from the stack). Stacks are commonly used in programming languages to implement function calls, and in web browsers to implement the back button.





Source: https://dev.to/erickarugu/implementing-the-stack-data-structure-in-javascript-59n5

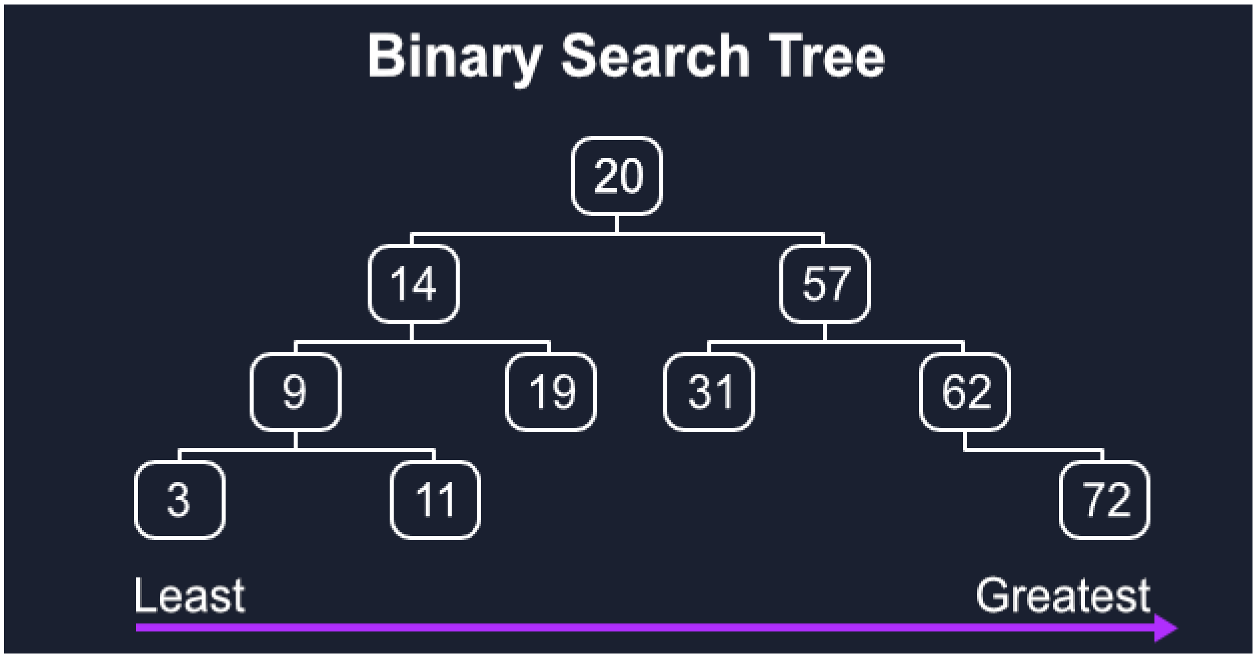
Real World Example- Undo/redo functionality: A text editor may use a stack to store the state of the document at different points in time, allowing users to undo and redo changes.

Codepen Example: https://codepen.io/shafferma08/pen/PoyKZbv

https://www.javascripttutorial.net/javascript-stack/

Lesson 3

Binary search trees:



Source: https://www.digitalocean.com/community/tutorials/js-binary-search-trees

Definition: A binary search tree is a tree-like data structure where each node has at most two child nodes, with the left child node having a smaller value and the right child node having a larger value. Binary search trees are commonly used for fast searching and sorting of elements, and are often used in database indexing.

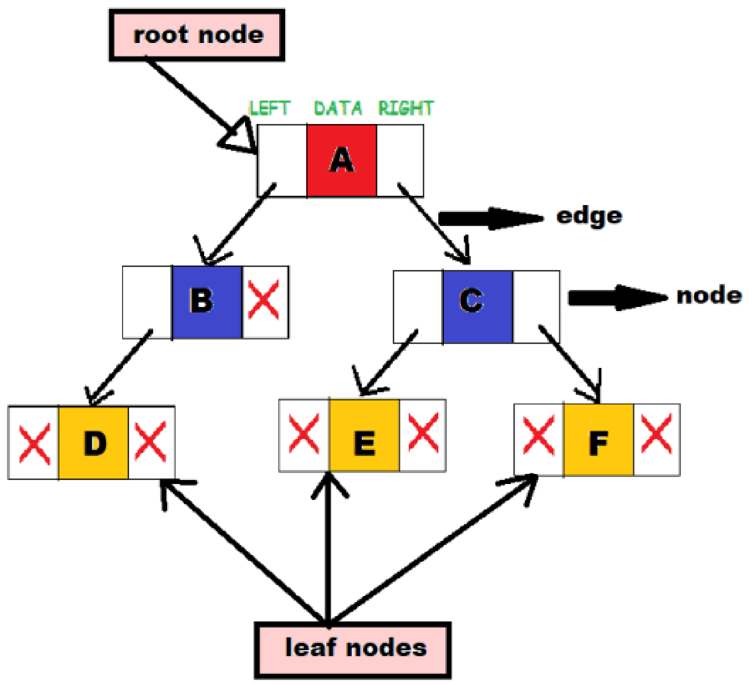
Real World Example- Database indexing: In a database, an index can be implemented as a binary search tree, with keys representing the indexed values and values representing the records. This allows for efficient searching and sorting of records.

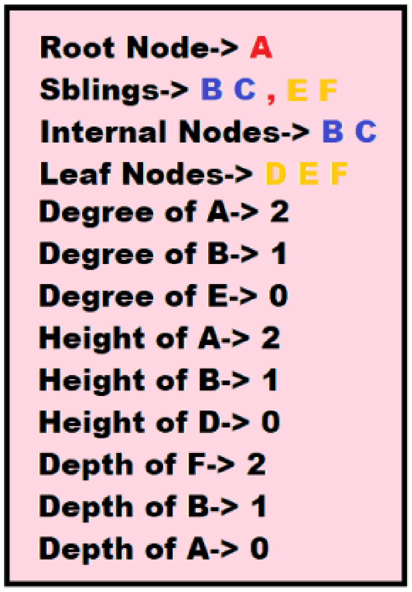
Codepen example: https://codepen.io/shafferma08/pen/ExdvPZb

https://www.freecodecamp.org/news/binary-tree-algorithms-for-javascript-beginners/

Binary Tree and Binary Search Tree are typically the most often used.

Tree Representation:-




Root Node-> A
Sblings-> B C , E F
Internal Nodes-> B C
Leaf Nodes-> D E F
Degree of A-> 2
Degree of B-> 1
Degree of E-> 0
Height of A-> 2
Height of B-> 1
Height of D-> 0
Depth of F-> 2
Depth of B-> 1
Depth of A-> 0

Basic Terminology

- Node => A node is an entity that contains a data and pointer to its child nodes.
- Edge => The connecting link between any two nodes is called as Edge.
- Root => It is the topmost node of a tree.
- Parent => The node which is a predecessor of any node is called as Parent Node.
- Child => The node which is a descendant of any node is called as Child Node.
- Siblings => Nodes that belong to the same parent are called Siblings.
- Leaf => The node which does not have a child is called Leaf Node.
- Internal Nodes => The node which has atleast one child is called Internal Node.
- Degree => The total number of children of a node is called as DEGREE of that Node.
- Level => Each step from top to bottom is called the Level of a tree. The root node is said to be at Level 0 and the children of the root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on...
- Height of a Node => The Height of a Node is the number of edges from the node to the deepest leaf (ie. the longest path from the node to a leaf node).

- Height of a Tree => The Height of a Tree is the height of the root node or the depth of the deepest node.
- Depth of a Node => The Depth of a Node is the number of edges from the root to the node.
- Depth of a Tree => The Depth of a Tree is the total number of edges from the root node to a leaf node in the longest path.

Check the implementation of tree in JS here.

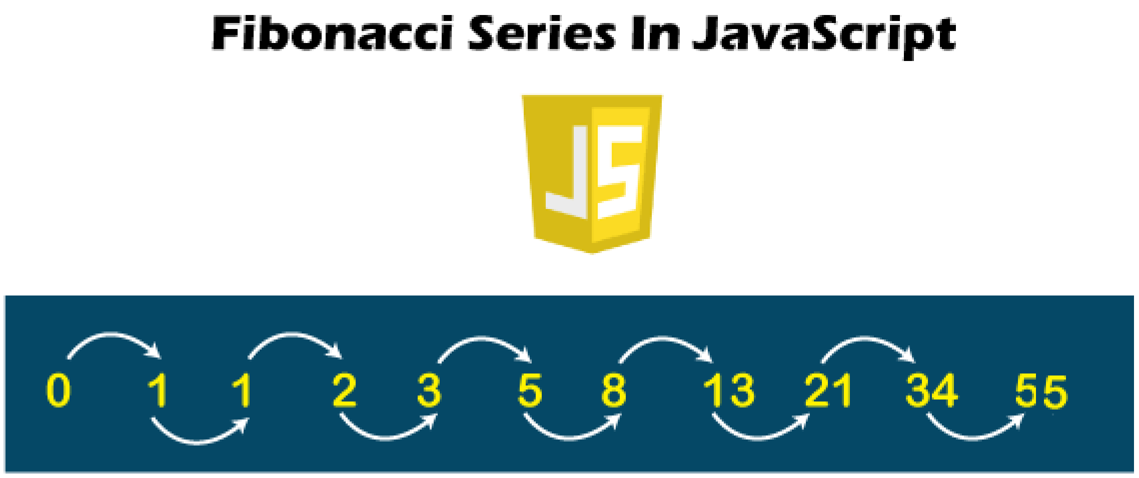
Source: https://dev.to/nehasoni__/7-javascript-data-structures-you-must-know-57ah

Recursion

https://codepen.io/shafferma08/pen/mdzMRoj

- Recursion is a programming technique where a function calls itself to solve a problem or perform a task
- Recursive functions have two parts: the base case and the recursive case
- The base case stops the function from calling itself indefinitely
- The recursive case calls the function with a modified input or state
- Each time the function calls itself, it creates a new instance of the function on the call stack
- When the base case is met, the function starts returning values and popping instances off the call stack until the original call is reached and the final result is returned
- Recursion is often used for traversing and manipulating tree-like data structures, and implementing recursive algorithms like the Fibonacci sequence or a recursive binary search
- Recursive algorithms can have exponential time complexity, which can make them impractical for large inputs
- Recursion can be optimized to reduce time complexity and make code easier to read and understand

Fibonacci



https://codepen.io/shafferma08/pen/JjmyEgy

The Fibonacci sequence is a famous example of a recursive algorithm. The sequence is defined as follows:

- The first two numbers in the sequence are 0 and 1.
- Each subsequent number is the sum of the two preceding numbers.

So the sequence goes: 0, 1, 1, 2, 3, 5, 8, 13, 21, and so on.

Table of Data Structures

Data Structure	Use Cases	Advantages	Disadvantages
Array	- Sequential data storage - Quick access by index	- Constant-time access - Memory efficient for static data	- Fixed size (for static arrays) - Expensive insertions and deletions
Linked List	- Dynamic data storage - Implementing stacks and queues	- Dynamic size - Efficient insertions and deletions	- (O(n)) access time - More memory per element
Stack	- Last in, first out (LIFO) operations - Backtracking, like browser back button	- Constant-time push and pop - Simple and memory efficient	- No random access to elements
Queue	- First in, first out (FIFO) operations - Order processing, task scheduling	- Constant-time enqueue and dequeue - Dynamic size	- No random access to elements
Hash Table / Map	- Key-value store - Quick lookups,	- Average constant-time operations - Flexible keys	- Space overhead - Collision handling can be complex

	insertions, deletions		
Binary Search Tree (BST)	- Sorted data storage - Quick lookups, insertions, deletions	- Logarithmic time for balanced trees - In-order traversal gives sorted data	- Can become unbalanced - Complex to implement
Graph	- Representing networks, relations - Pathfinding algorithms	- Can represent many real-world systems - Very flexible	- More memory intensive - Algorithms can be complex