How to prepare

Try to solve problem, write the code on paper, test your code on paper, type your code as is into pc.

**Required knowledge**

**Data Structures**

**Linked list**

Each node in the list links to the next element. Goes in one direction/links in one direction. Can contain most data types. To get to the fourth node in a linked list you must start at one and go up to four. This is one way it differs from an array. Insertion and deletions to the begging of the list is very fast as it is done in constant time. Insertion and deletion to anywhere else in the list is slow and down in linear time.( o(n) ). The first node in the list is called the head node.

**Doubly linked list**

Like a linked list but instead of nodes only linking to the next element they also link to the previous element. This can improve the performance and allow you to start at the end instead of the beginning.

**Trees**

Trees start with a root node. The root node has a coupe child nodes that then can have more child nodes. A node with no children is called a leaf. Tree are most often associated with binary trees.

**Binary Tree**

A tree that has two nodes or less for every node. A left or right node. A Binary search tree has a specific ordering property. The far bottom left nodes are the lowest number or first in alphabet. The root node is the middle/median number or middle of the alphabet. The far right nodes are the highest number or latest in the alphabet. This structure allows us to eliminate half the data when searching every time. ( o( log n ) ).

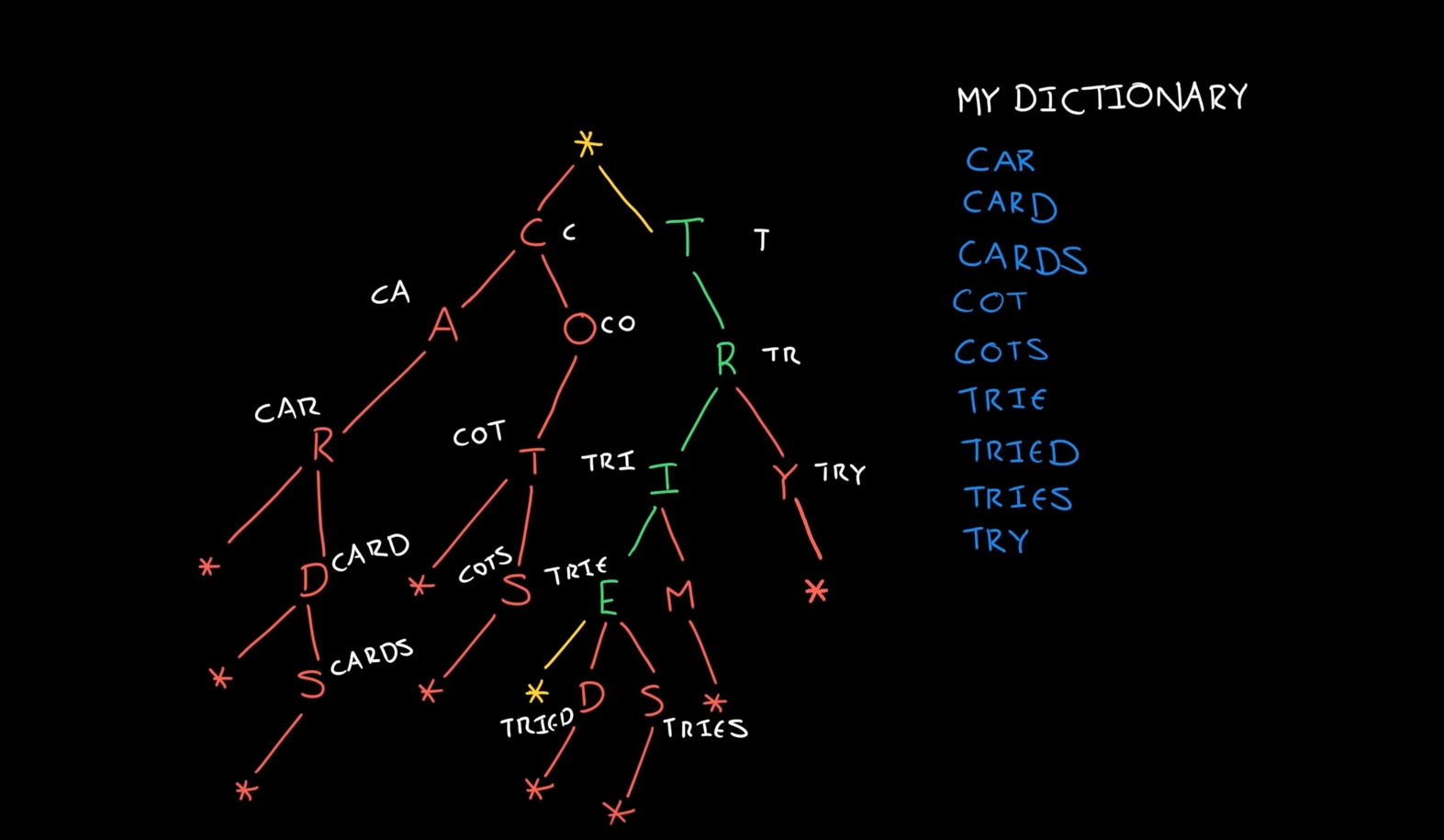
**Tree Traversing orders**

**Inorder:** left then root the right.most common for binary search trees. Prints the nodes in order.

**Preorder:**  Root then left then right.

**Postorder:** Left the right then root.

**Tries**



A Trie is a type of tree. It typicaly stores characters as values. Each character is actually representing a word or a part of a word. If you follow a path down they will make up one word. It allows for very fast lookups of a particular kind.

**Graphs**

Graphs are a collection of nodes/vertices and the relationship and connections between the nodes/vertices. The connection between to nodes/vertices are called edges. Example is a social network where nodes are people and edges are them being friends or not.

**Directed Graphs**

Edges have a direction. Like a web page link. It goes one way.

**Undirected Graphs**

Edges have no direction. Like a friendship. It could go either way

**Adjacency list**

**Adjacency matrix**

**Incidence matrix**

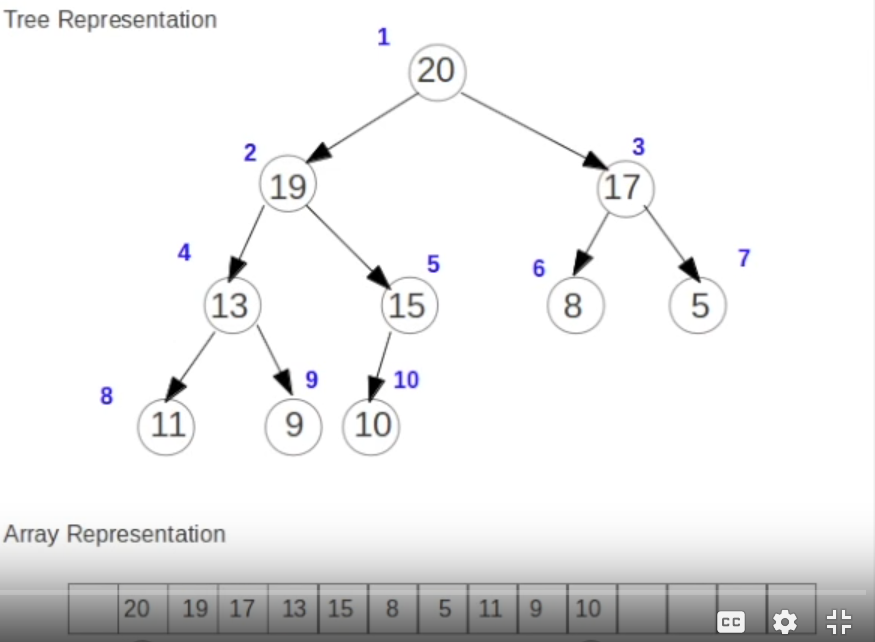
**Stacks**

First added last out. Last added first out. Just like a stack of books. The first book is at the bottom and is the last book to come out. Example URL stack the page on top of the root domain.

**Queues**

First added First out. Last added last out. Like a line of people waiting. The person who was first in line is the person first let in or out of line.

**Heaps**



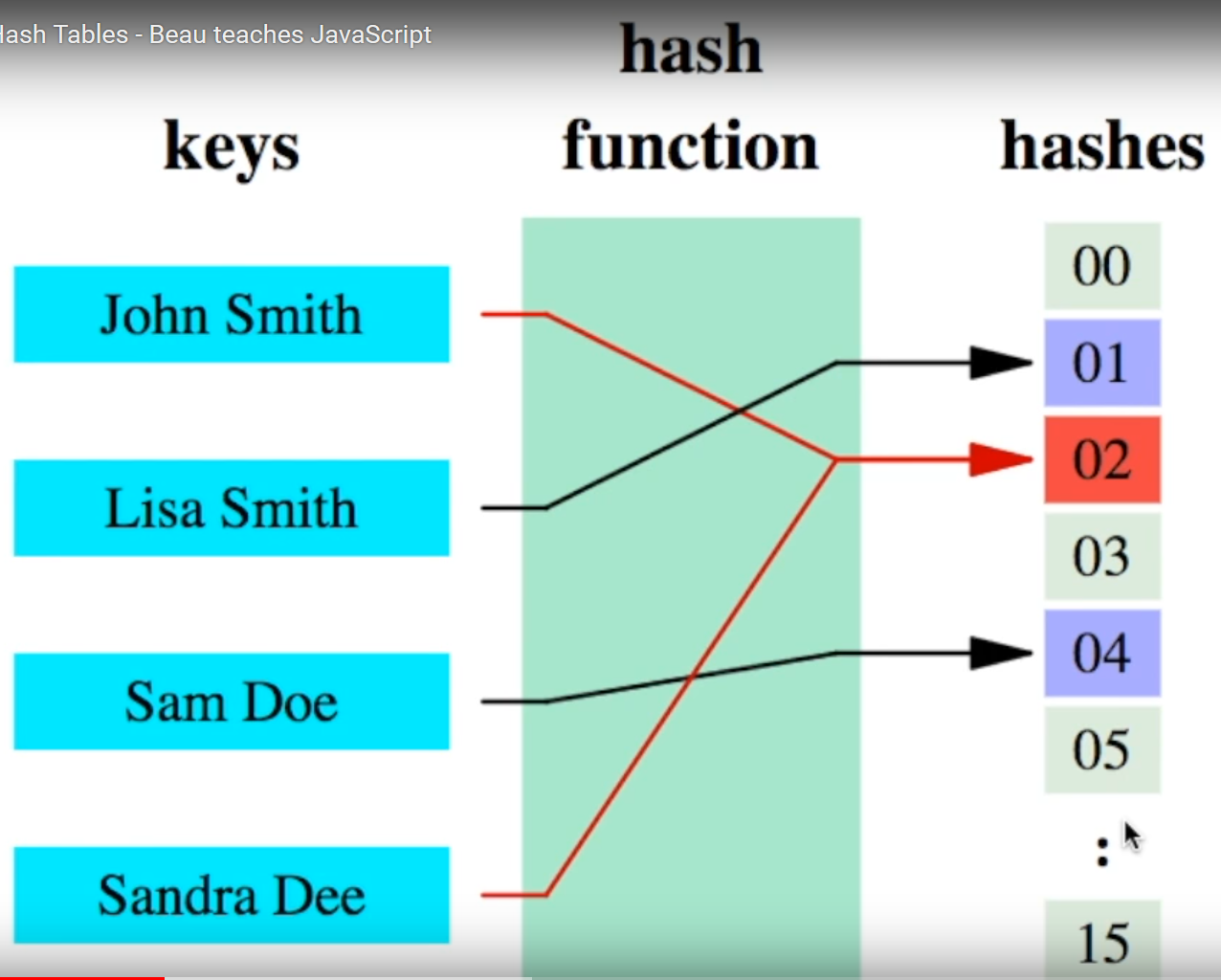
Heaps are a type of tree. Very similar to a binary tree as each node can only have two children but, Heaps are ordered differently than binary search trees. Heaps can be order for min or max. Min is where the root node is the smallest number. Children must be smaller than the root or parent node. Min and max Heaps can be expresses with arrays. Node on the same level in a heap don’t have to be ordered. The heap nodes are filled from left to right in the tree and only order by the parent node having to be bigger than the child node. To find corresponding values in an array for a heap do this. Left child index \* 2. Right child index \*2 + 1. To find parent node of a child do index / 2. (index is the index of the node in the array).

**Ordering of Heaps**

A heap maintains order by a sorting process. It first adds the new node value to the end of the Heap tree or array. It then compares the parent node to the current node to see if the parent node is bigger or smaller. In a min heap if the parent node is smaller then it doses nothing. If the parent node is bigger then it swaps the values of the parent node and current node. The it repeats this process until the parent node is smaller than the current node.

**Vectors / array List**

***Hash Tables (*** *very important )*



Hash tables are key value lookups. Key are often strings but can be anything. The value is stored into a array. The hash function takes the string key gets a integer based on the string and then puts the value in the array under the integer. The average time of hash tables is o(1) so it is very fast.

**Collisions**

Collisions happen in a hash tables when two string key have the same hash code integer. When this happens, we make that index a linked list that contains both values. This method is called chaining collisions for hash tables.

**Js Hash table**

Js has hash tables built in. If you use a named index in an array it should convert it to a hash table. But check to be sure.

**Maps**

**Algorithms**

**Breadth-First Search**

Is a traversal algorithms to find nodes in a graph. BFS first visits all the nodes children/connected nodes first. Then moves to the neighbor nodes children and further out until it finds the node.

**Depth First Search**

Is a traversal algorithm to find nodes in a graph. DFS go to one child node and the that child child nodes until it reaches a dead end or return to the original node. Once reaching an end or original node it then goes backwards and checks each node to see if there is any unvisited child nodes. It does this until all nodes are visited.

**Binary Search**

**Merge Sort**

**Quick Sort**

**Concepts**

**Bit manipulation**

**Memory (stack vs Heap)**

**Recursion**

**Dynamic Programming**

**Big O time and Space**