ALGORITHMS

A computer program is a collection of instructions to perform a specific task. For this, a computer program may need to store data, retrieve data, and perform computations on the data.

What Are Data Algorithms?

A data structure is a method of organizing data in a virtual system. Think of sequences of numbers, or tables of data: these are both well-defined data structures. An algorithm is a sequence of steps executed by a computer that takes an input and transforms it into a target output.

Together, data structures and algorithms combine and allow programmers to build whatever computer programs they'd like. Deep study into data structures and algorithms ensures well-optimized and efficient code.

How Do Data Structures and Algorithms Work Together?

There are many algorithms for different purposes. They interact with different data structures in the same computational complexity scale. Think of algorithms as dynamic underlying pieces that interact with static data structures.

The way data is expressed in code is flexible. Once you understand how algorithms are built, you can generalize across different programming languages. In a sense, it's a bit like knowing how a related family of languages work syntactically. Once you glimpse the fundamental rules behind programming languages and their organizing principles, you can more easily switch between the different languages and learn each faster.

Common Data Structures and Algorithms

Common data structures you'll see across different programming languages include:

- Linked lists
- Stacks
- Queues
- Sets
- Maps
- Hash tables
- Search trees

Each of these has its own computational complexity for associated functions like adding items and finding aggregate measures such as the mean for the underlying data structure.

Some common categories of algorithms are:

- Search
- Sorting
- Graph/tree traversing
- Dynamic programming
- Hashing and regex (string pattern matching)

Array

```
// instantiation
let empty = new Array();
let teams = new Array('One', 'Two', 'Three');

// literal notation
let otherTeams = ['Four', 'Five', 'Six'];

// size
console.log('Size:', otherTeams.length);

// access
console.log('Access:', teams[0]);

// sort
const sorted = teams.sort();
console.log('Sorted:', sorted);

// search
const filtered = teams.filter((team) => team === 'One');
console.log('Searched:', filtered);
```

Dynamic Array

```
/* Arrays are dynamic in Javascript :-) */

// instantiation
let empty = new Array();
let teams = new Array('One', 'Two', 'Three');

// literal notation
let otherTeams = ['Four', 'Five', 'Six'];

// size
console.log('Size:', otherTeams.length);

// access
console.log('Access:', teams[0]);

// sort
const sorted = teams.sort();
console.log('Sorted:', sorted);

// search
const filtered = teams.filter((team) => team === 'One');
console.log('Searched:', filtered);
```

Linked List

```
function LinkedListNode(val) {
    this.val = val;
    this.next = null;
  }

  class MyLinkedList {
    constructor() {
      this.head = null;
      this.tail = null;
    }

    prepend(newVal) {
      const currentHead = this.head;
      const newNode = new LinkedListNode(newVal);
      newNode.next = currentHead;
      this.head = newNode;

      if (!this.tail) {
        this.tail = newNode;
      }
    }

    append(newVal) {
      const newNode = new LinkedListNode(newVal);
      if (!this.head) {
        this.head = newNode;
        this.tail = newNode;
      } else {
        this.tail.next = newNode;
        this.tail = newNode;
      }
    }
  }

  var linkedList1 = new MyLinkedList();
  linkedList1.prepend(25);
  linkedList1.prepend(15);
  linkedList1.prepend(5);
  linkedList1.prepend(9);
```

Queue

```
class Queue {
  constructor() {
    this.queue = [];
  }

  enqueue(item) {
    return this.queue.unshift(item);
  }

  dequeue() {
    return this.queue.pop();
  }

  peek() {
    return this.queue[this.length - 1];
  }

  get length() {
    return this.queue.length;
  }

  isEmpty() {
    return this.queue.length === 0;
  }
}

const queue = new Queue();
queue.enqueue(1);
queue.enqueue(2);
console.log(queue);

queue.dequeue();
console.log(queue);
```

Stack

```
class Stack {
  constructor() {
    this.stack = [];
  }

  push(item) {
    return this.stack.push(item);
  }

  pop() {
    return this.stack.pop();
  }

  peek() {
    return this.stack[this.length - 1];
  }

  get length() {
    return this.stack.length;
  }

  isEmpty() {
    return this.length === 0;
  }
}

const newStack = new Stack();
newStack.push(1);
newStack.push(2);
console.log(newStack);

newStack.pop();
console.log(newStack);
```

Hash Table

```
class Hashmap {
    constructor() {
      this._storage = [];
    }

    hashStr(str) {
      let finalHash = 0;
      for (let i = 0; i < str.length; i++) {
        const charCode = str.charCodeAt(i);
        finalHash += charCode;
      }
      return finalHash;
    }

    set(key, val) {
      let idx = this.hashStr(key);

      if (!this._storage[idx]) {
        this._storage[idx] = [];
      }

      this._storage[idx].push([key, val]);
    }

    get(key) {
      let idx = this.hashStr(key);

      if (!this._storage[idx]) {
        return undefined;
      }

      for (let keyVal of this._storage[idx]) {
        if (keyVal[0] === key) {
          return keyVal[1];
        }
      }
    }
  }
}
```

Graph

```
class Graph {
  constructor() {
    this.adjacencyList = {};
  }

  addVertex(nodeVal) {
    this.adjacencyList[nodeVal] = [];
  }

  addEdge(src, dest) {
    this.adjacencyList[src].push(dest);
    this.adjacencyList[dest].push(src);
  }

  removeVertex(val) {
    if (this.adjacencyList[val]) {
      this.adjacencyList.delete(val);
    }

    this.adjacencyList.forEach((vertex) => {
      const neighborIdx = vertex.indexOf(val);
      if (neighborIdx >= 0) {
        vertex.splice(neighborIdx, 1);
      }
    });
  }

  removeEdge(src, dest) {
    const srcDestIdx = this.adjacencyList[src].indexOf(dest);
    this.adjacencyList[src].splice(srcDestIdx, 1);

    const destSrcIdx = this.adjacencyList[dest].indexOf(src);
    this.adjacencyList[dest].splice(destSrcIdx, 1);
  }
}

var graph = new Graph(7);
var vertices = ["A", "B", "C", "D", "E", "F", "G"];
for (var i = 0; i < vertices.length; i++) {
   graph.addVertex(vertices[i]);
 }
 graph.addEdge("A", "G");
 graph.addEdge("A", "E");
 graph.addEdge("A", "C");
 graph.addEdge("B", "C");
 graph.addEdge("C", "D");
 graph.addEdge("D", "E");
 graph.addEdge("E", "F");
 graph.addEdge("E", "C");
 graph.addEdge("G", "D");
 console.log(graph.adjacencyList);
```

Tree

```
function TreeNode(value) {
  this.value = value;
  this.children = [];
  this.parent = null;

  this.setParentNode = function (node) {
    this.parent = node;
  };

  this.getParentNode = function () {
    return this.parent;
  };

  this.addNode = function (node) {
    node.setParentNode(this);
    this.children[this.children.length] = node;
  };

  this.getChildren = function () {
    return this.children;
  };

  this.removeChildren = function () {
    this.children = [];
  };
}

const root = new TreeNode(1);
root.addNode(new TreeNode(2));
root.addNode(new TreeNode(3));

const children = root.getChildren();
for (let i = 0; i < children.length; i++) {
  for (let j = 0; j < 5; j++) {
    children[i].addNode(new TreeNode("second level child " + j));
  }
}

console.log(root);
children[0].removeChildren();
console.log(root);
console.log(root.getParentNode());
console.log(children[1].getParentNode());
```

Binary Search Tree

```
function Node(val) {
  this.val = val;
  this.left = null;
  this.right = null;
}

class BST {
  constructor(val) {
    this.root = new Node(val);
  }

  add(val) {
    let newNode = new Node(val);

    function findPosAndInsert(currNode, newNode) {
      if (newNode.val < currNode.val) {
        if (!currNode.left) {
          currNode.left = newNode;
        } else {
          findPosAndInsert(currNode.left, newNode);
        }
      } else {
        if (!currNode.right) {
          currNode.right = newNode;
        } else {
          findPosAndInsert(currNode.right, newNode);
        }
      }
    }

    if (!this.root) {
      this.root = newNode;
    } else {
      findPosAndInsert(this.root, newNode);
    }
  }

  remove(val) {
    let self = this;
     if (val === node.val) {
       if (!node.left && !node.right) {
         return null;
       }
```

```
    if (val === node.val) {
      if (!node.left && !node.right) {
        return null;
      }
      if (!node.left) {
        return node.right;
      }
      if (!node.right) {
        return node.left;
      }
      let temp = self.getMinimum(node.right);
      node.val = temp;
      node.right = removeNode(node.right, temp);
      return node;
    } else if (val < node.val) {
      node.left = removeNode(node.left, val);
      return node;
    } else {
      node.right = removeNode(node.right, val);
      return node;
    }
  };
  this.root = removeNode(this.root, val);
}

getMinimum(node) {
  if (!node) {
    node = this.root;
  }
  while (node.left) {
    node = node.left;
  }
  return node.val;
}
// helper method
  contains(value) {
    let doesContain = false;

    function traverse(bst) {
      if (this.root.value === value) {
        doesContain = true;
      } else if (this.root.left !== undefined && value < this.root.value) {
        traverse(this.root.left);
      } else if (this.root.right !== undefined && value >
                  this.root.value) {
        traverse(this.root.right);
      }
```

```
    }

    traverse(this);
    return doesContain;
  }
}

const bst = new BST(4);
bst.add(3);
bst.add(5);
bst.remove(3);
console.log(bst);
```