
Embrace Opportunity

Software Engineering Day 14

Data Structures and Algorithms

Today's Overview

- Data structures and Algorithms
  - Array
  - Hash map
  - Linked list
  - Stack,
  - Queue
  - Tree.
- Class examples Palindrome, Anagram.
- Cheat Sheet:
  - iX Cheat Sheet
  - iX Cheat Sheet - Day 13

Data Structure and Algorithms

Array, Hash map, Linked List, Stack, Queue, Tree

Introduction to Data Structure and Algorithms

- Data Structures and Algorithms (DSA) is a fundamental part of software engineering, that focuses on the ability to solve complex problems.
- A data structure is a name location that can be used to store and organize data.
    - Collection of data values, the relationships between them, and the functions or operations that can be applied to them.
- An algorithm is a collection of steps to solve a particular problem.

Introduction to Data Structures

- Data structures is broken into two categories:
  - Linear - Elements are arranged in a sequence one after the other:
    - Array
    - Hash Maps
    - Stack
    - Queue
    - Linked List
  - Non-linear - Elements are in no set sequence, but rather arranged in a hierarchical manner (One element will be connect to another or more elements):
    - Tree

Algorithms - Big O

- Big ***O*** Notation
  - Used to describe the upper bound of a algorithm
    - Finds the worst case complexity for an algorithm
- Defines the runtime and memory required to execute an algorithm by identifying how the performance of the algorithm will change as the input size increases.
  - Simplified: Analysis of an algorithm's efficiency.
- Syntax:
  - O(n)
    - n - Expression for runtime
      - Number of values in the data set the algorithm is working on.
      - Input size.

Big O - Time and Space Complexity

- Time and Space Complexity:
  - Time (Runtime):
    - Specifies the time it takes to execute an algorithm as a function of its input size.
  - Space (Memory):
    - Specifies the total amount of memory required to execute an algorithm as a function of the size of the input.
- Types of Complexities:
  - Constant
  - Linear
  - Logarithmic
  - Quadratic
  - Exponential
  - Factorial

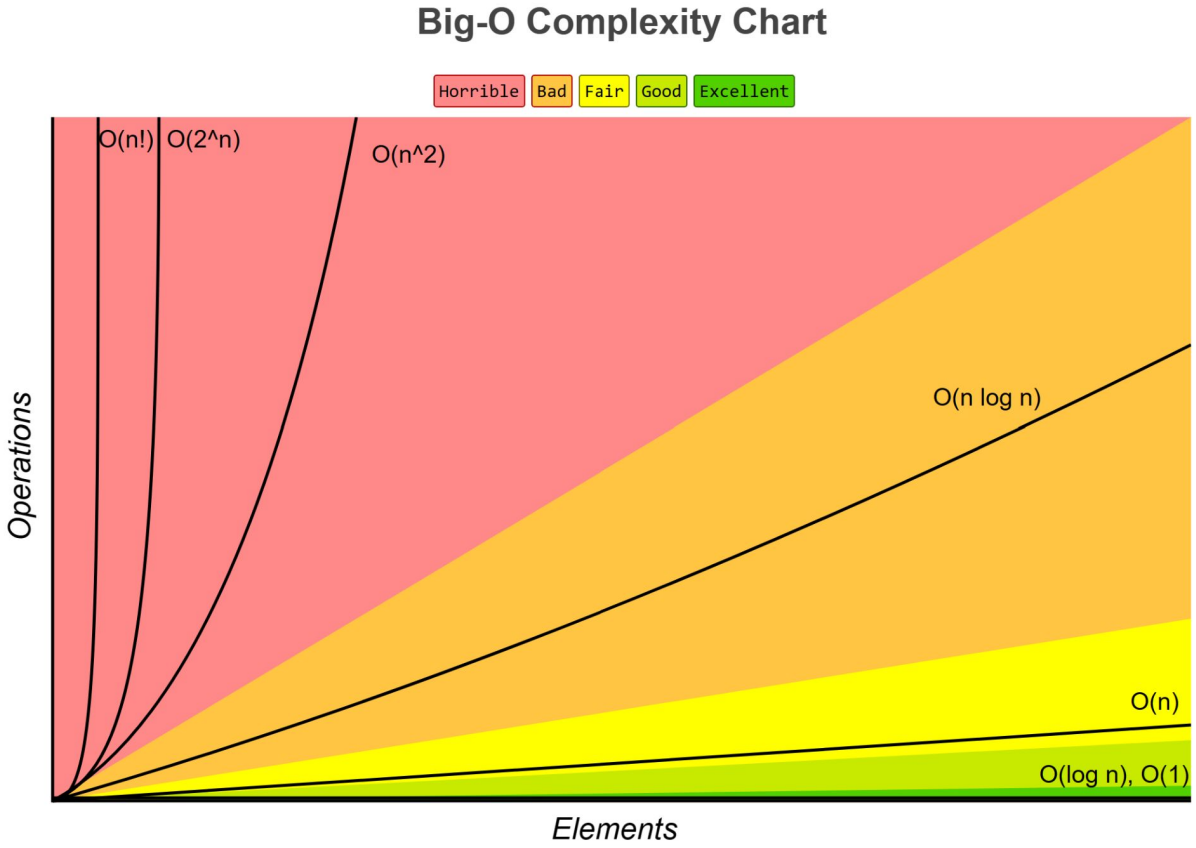
Big O - Time and Space Complexity

- Types of Complexities:
  - Constant
    - O(1)
  - Linear
    - O(n)
  - Logarithmic
    - O($n$log$n$)
  - Quadratic
    - O(n^2)
  - Exponential
    - O(2^n)
  - Factorial
    - O(n!)

Big O - Time and Space Complexity

- Big ***O*** Notation is measured three ways:
  - Average case
  - Best case
  - Worst case
    - Typically work with worst case with algorithms.
- Rules:
  - Ignores constants
    - $5n$ = O(n)
  - Dominant Terms:
    - Higher-order terms, negates lower order terms:
    - O(1) < O(log$n$) < O(n) < O($n$log$n$) < O(n^2) < O(2^n) < O(n!)

Big O - Time and Space Complexity



Big O - Time and Space Complexity

- Complexity Ratings:
  - O(1)
    - Best
  - O(log$n$)
    - Good
  - O(n)
    - Fair
  - O($n$log$n$)
    - Bad
  - O(n^2) & O(2^n) & O(n!)
    - Worst
- Cheat Sheet

Data Structure

Arrays

Arrays

- Array is a linear data structure that is a collection of items stored at contiguous memory locations.
- Each element can be accessed through the index.
- Arrays start at index 0.
- Arrays are resizable and can contain a mix of different data types
- Operation:
    - *Insertion* - Inserting a new element in an array. O(n)
    - *Deletion* - Remove element from the array. O(n)
    - *Searching* - Check if an element exists in array. O(n)
    - *Sorting* - Maintaining the order of elements in the array.
    - *Traversal* - Traverse through the elements of the array.

Arrays - Visual Representation

- Array and Multidimensional Array:

```
const testArray = [0, 1, 1, 2, 3, 5, 8, 13]

// Multidimensional array (array of arrays).
const multiArray = [
    [0, 1, 1],
    [2, 3, 5],
    [8, 13, 21],
]
```

Data Structure

Hash Maps

Hash Maps

- Hash map is a data structure that maps a key to a value for highly efficient lookup, insertion and deletion.
- Time and space complexity:
  - O(n)
- Keys are hashed to determine the index and where the corresponding values are stored.
- In Javascript:
  - Implementing associate array (or object) is essentially hash map.
- Hash function (algorithm):
  - Takes a key and generates a unique index (hash code) corresponding to that value in a hash table.

Hash Maps

- Operations:
  - *Insertion* - Key-value pair is hashed and index is used to store the value in the corresponding slot.
  - *Deletion* - Key is hashed to find the index, item at the index is removed.
  - *Lookup* - Key is hashed, value at the index is returned.
- Time Complexity - worst case:
  - *Insertion*: O(n)
  - *Deletion*: O(n)
  - *Lookup*: O(n)

Hash Maps - Collisions

- Collisions:
  - Occurs when two different keys generate the exact same hash.
- Collision Resolution Techniques:
  - Chaining
    - Storing a data structure at each array index.
    - If collision occurs, append the new key-value pair at the index.

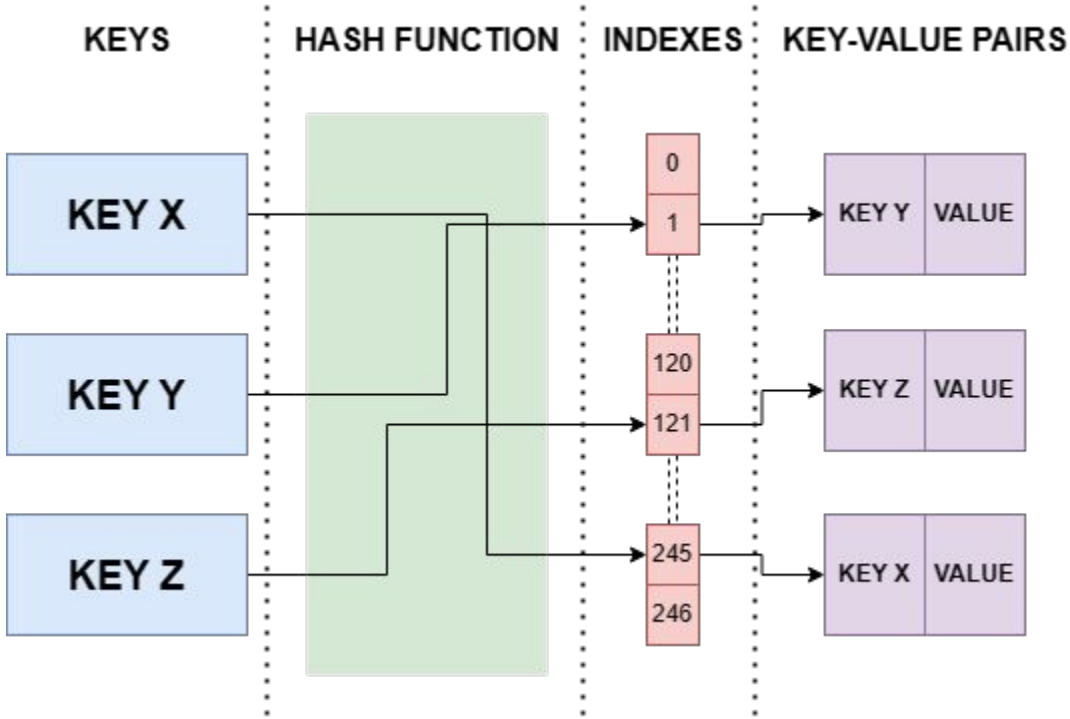
Hash Maps - Collisions

- Open addressing
  - Searches the array for the next empty slot for the key-value combination to be inserted.
    - Linear probing:
      - Algorithm moves linearly to the next available index.
    - Quadratic Probing:
      - Algorithm employs a quadratic function to find the next available index.
    - Double Hashing:
      - Algorithm calculates the step size between probes using a secondary hash function.

Hash Maps

- Use cases:
  - Constant time lookup and insertion is required.
  - Cryptographic applications.
  - Indexing data is required.

Hash Maps - Visual Representation



Data Structure

Linked Lists

Linked Lists

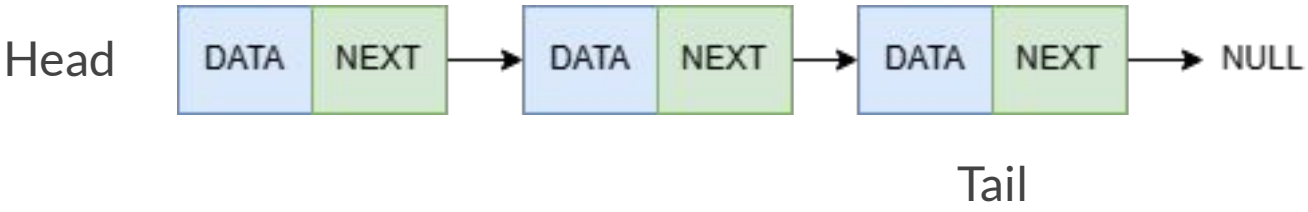
- Linked list is a linear data structure that includes a series of linked nodes, that stores values in the form of a list.
- Each node stores the data and the address of the next node in the link (Pointer).
  - First node (Head)
  - Last node (Identified by the pointer being null)(Tail)
- Three types of linked lists:
  - Singly linked:
    - Each node has a single pointer, that indicates the next node.
  - Doubly linked:
    - Each node has two pointers, that indicate the next node and the previous node.
  - Circular linked:
    - Last node has pointer to the first node, instead of null.

Linked Lists

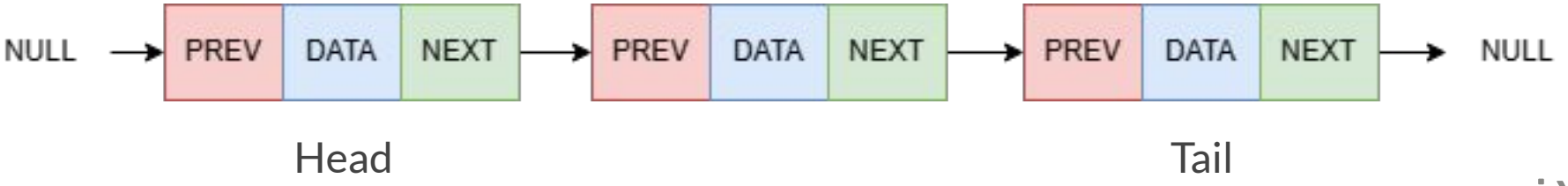
- Use cases:
  - Dynamic memory allocation.
  - Implement other data structures like: stack and queue.
  - Undo functionality.
- Operations:
  - *Search* - Find a note in the linked list. O(n)
  - *Insertion* - Create node and adjust pointers based on insertion position. O(1)
  - *Deletion* - Connecting the next and previous pointers than removing node. O(1)
  - *Traversal* - Go through a linked list following the links from one link to another.
  - *Sort* - Sort the nodes of the linked list.

Linked Lists - Visual Representation

- Singly Linked List:

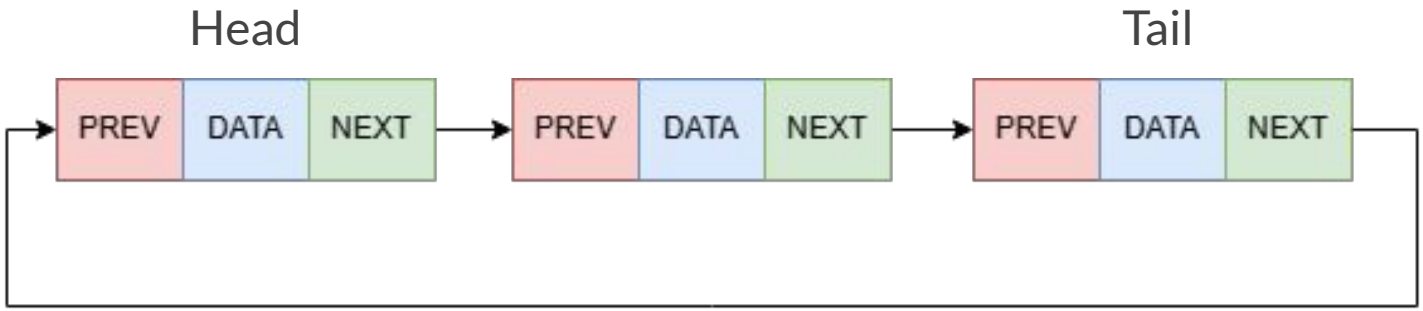


- Doubly Linked List:



Linked Lists - Visual Representation

- Circular Linked List:



Data Structure

Stacks and Queues

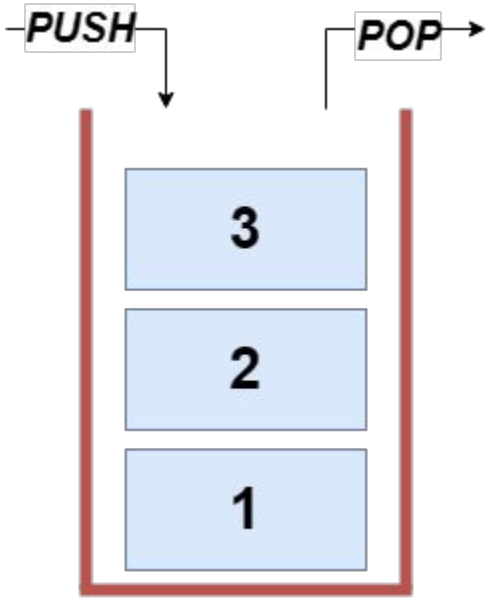
Stacks

- Stack is a linear data structure that follows the principle of last in first out (LIFO).
- Last elements that are inserted inside the stack are removed first.
  - Inserting element to the top of the stack (Push).
  - Removing element (Pop).
- Use cases:
  - JavaScript call stack.
  - Compile-time memory management.
  - Undo/redo functionality.
- Simplest implementation is an array with its push and pop method.

Stacks

- Operations:
  - *Push* - Add element to a stack. O(1)
  - *Pop* - Remove element from the stack. O(1)
  - *isEmpty* - Returns true if stack is empty.
  - *isFull* - Check if the stack is full.
  - *Peek* - Returns top most element in the stack.

Stacks - Visual Representation



Stacks - Example

- Problem Statement:
- Given a string *s* containing just the characters *'('*, *')'*, *'{'*, *'}'*, *'['* and *']'*, determine if the input string is valid.
- An input string is valid if:
  - Open brackets must be closed by the same type of brackets.
  - Open brackets must be closed in the correct order.
  - Every close bracket has a corresponding open bracket of the same type.

Stacks - Example - Solution

```js
var isValid = function(s) {
    let stack = [];
    for (let c of s) {
        if (c === '(' || c === '{' || c === '[') {
            stack.push(c);
        } else {
            if (!stack.length ||
                (c === ')' && stack[stack.length - 1] !== '(') ||
                (c === '}' && stack[stack.length - 1] !== '{') ||
                (c === ']' && stack[stack.length - 1] !== '[')) {
                return false;
            }
            stack.pop();
        }
    }
    return !stack.length;
};
```

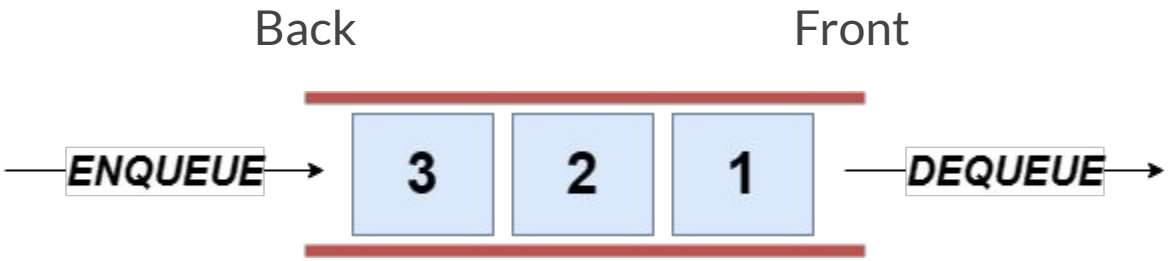
Queues

- Queue is a linear data structure that follows the principle of first in first out (FIFO).
- First element inserted into the queue is the first item that is removed.
  - Inserting element (Enqueue).
  - Removing element (Dequeue).
- Use cases:
  - Background tasks.
    - CPU and Disk scheduling
  - Handling of interrupt in real-time system.
- Simplest implementation is an array with its push and shift methods.

Queues

- Operations:
  - *Enqueue* - Add element at the end of the queue. O(1)
  - *Dequeue* - Remove an element from the front of the queue. O(1)
  - *isEmpty* - Check if there is an element in the queue.
  - *isFull* - Check if the queue is full.
  - *Peek* - Returns first element without removing it.

Queues - Visual Representation



Data Structure

Trees

Trees

- Tree is a nonlinear hierarchical data structure that consists of nodes connect by edges.
- Parent / child relationship - nodes depend on other nodes.
- Terminology:
  - Node - Element that contains a key or value and pointer to its child nodes.
  - Edge - Link between two nodes.
  - Root - First node of the tree.
  - Children - All nodes that come off the root node.
  - Leaf nodes - Children that are on the bottom of the tree.
  - Height - Number of parent/child connections.

Trees

- Requirement:
  - Only valid connections between nodes is from parent to child.
  - Only a single root node.
- Tree Types:
  - Binary trees:
    - Each node has a maximum of two children.
    - Useful in searching (Binary Search Trees (BSTs)).
    - Left node of its parent has a value less than its parent.
    - Right node of its parent has a value greater than its parent.

Trees

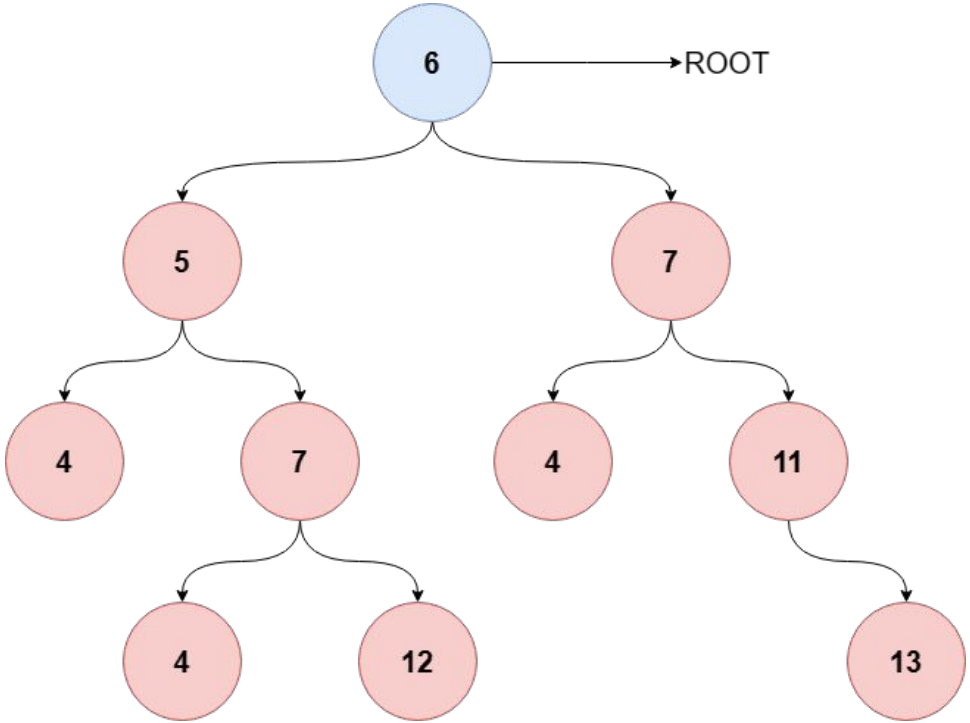
- Heaps:
  - Two types of heaps:
    - Max Heaps
      - Parent nodes are always greater than children.
    - MinHeaps
      - Parent nodes are always smaller than children.
  - Used frequently to implement priority queues.
    - Dijkstra's path-finding algorithm.

Trees

- Use cases:
  - Binary Search Trees(BSTs) - check whether an element is present in a set or not.
  - Compilers use a syntax tree to validate the syntax.
  - DOM model.
  - File folders in operating systems.

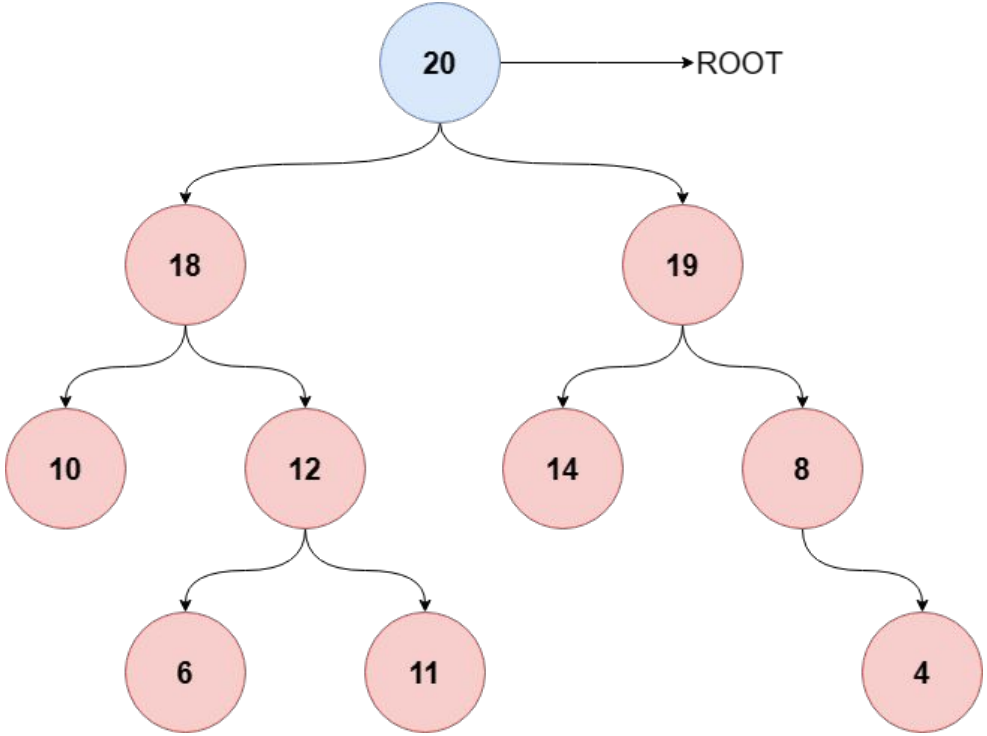
Trees - Visual Representation

- Binary Trees



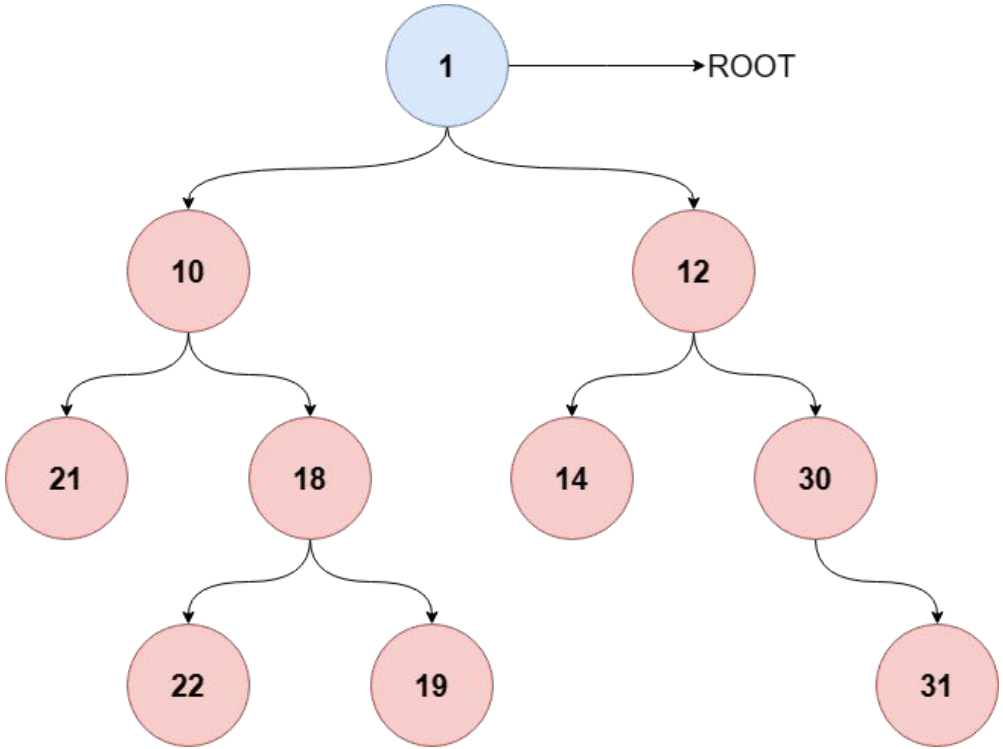
Trees - Visual Representation

- Max Heaps



Trees - Visual Representation

- MinHeaps



Algorithms

Recursion, Dynamic Programming,
Sorting and Searching

Algorithms - Recursion

- Recursion is an algorithm used to solve problems by creating a function that calls itself until the desired end result is achieved (recursive function).
- Three components of a recursive function:
  - Function definition.
  - Base condition:
    - Condition to end the recursion.
  - Recursive call:
    - Handles the function calling itself again.

Recursion - Example

```
function getFactorial(num) {
    if (num === 0) {
        return 1;
    }
    let factorial = num * getFactorial(num - 1)
    return factorial;
}
```

- Explanation:
  - The function will call itself till the base condition (num === 0) is reached, and will give the result, for example num == 3:
  - 3 * getFactorial(3 - 1) = 3 * 2
  - 6 * getFactorial(2 - 1) = 6 * 1
  - 6 * getFactorial(1 - 1) = 6 * 1 = 6
    - End reached, result 6.

Algorithms - Dynamic Programming

- Dynamic Programming (DP) is a technique that breaks a down problem into subproblems and solving each subproblem to solve to whole problem.
  - Avoid redundant computing.
- Dynamic programming can optimize recursive solutions that make repeat calls to the same input.
- Two methods of Dynamic Programming:
  - Memoization:
    - Top-down approach.
  - Tabulation:
    - Bottom-up approach.

Algorithms - Dynamic Programming

- Memoization:
  - Caching results of function calls, and return the result if the function is called again with the same inputs.
  - Top-down approach.
    - This approach checks whether it has previously solved the subproblem, if yes it returns the cached result and caches further calculations. Else it calculates the sub-problems like usual.
      - It remembers the results previously computed.
  - Suited for small sets of inputs.

Algorithms - Dynamic Programming

- Tabulation:
  - Storing results of subproblems in a table and use the results to solve larger subproblems until the desired result.
  - Bottom-up approach.
    - Reverse of top-down approach, but an iterative version of it.
    - Sorts the subproblems by their input size and solves them iteratively in the order of smallest to largest.
    - When solving a subproblem it will first solve all of the smaller subproblems its solution depends on and stores their values in memory.
  - Suited for large set of inputs.

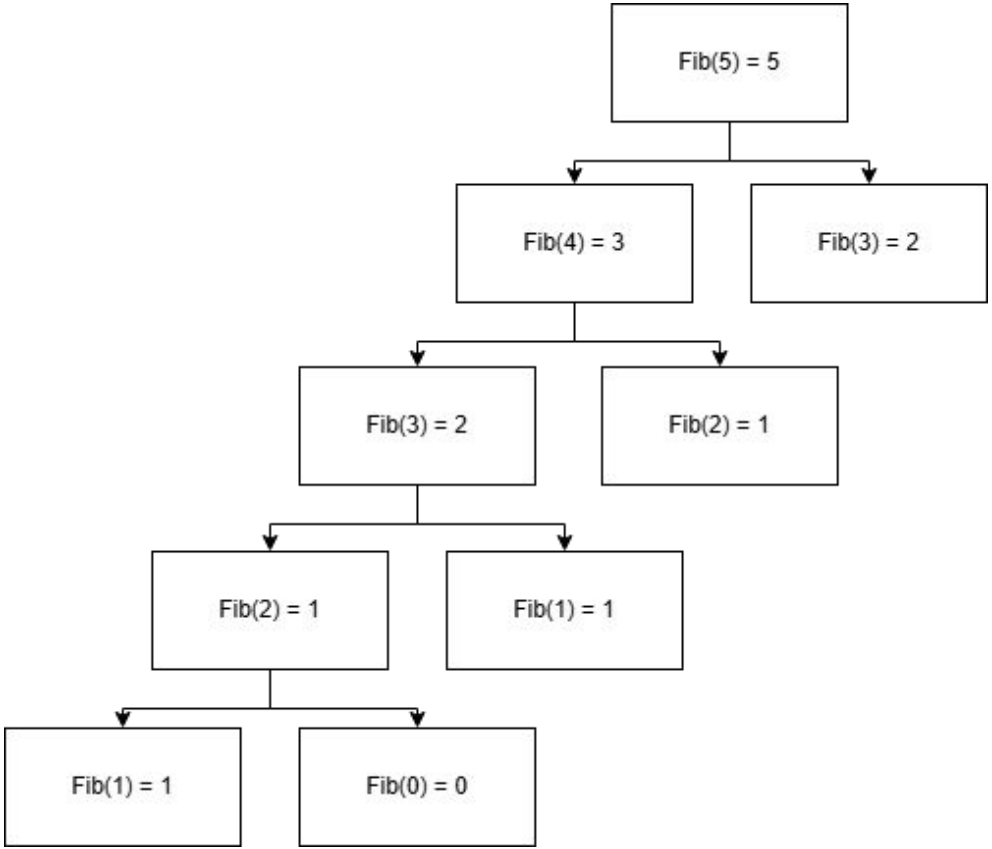
Dynamic Programming - Example

- Memoization:

```
const cache = new Map();
function fib(n) {
    if (cache.has(n)) {
        return cache.get(n);
    }

    if (n === 0) {
        return 0;
    } else if (n === 1) {
        return 1;
    } else {
      let result = fib(n - 1) + fib(n - 2);
      cache.set(n, result);
      return result;
    }
}
```

Dynamic Programming - Example Visualised



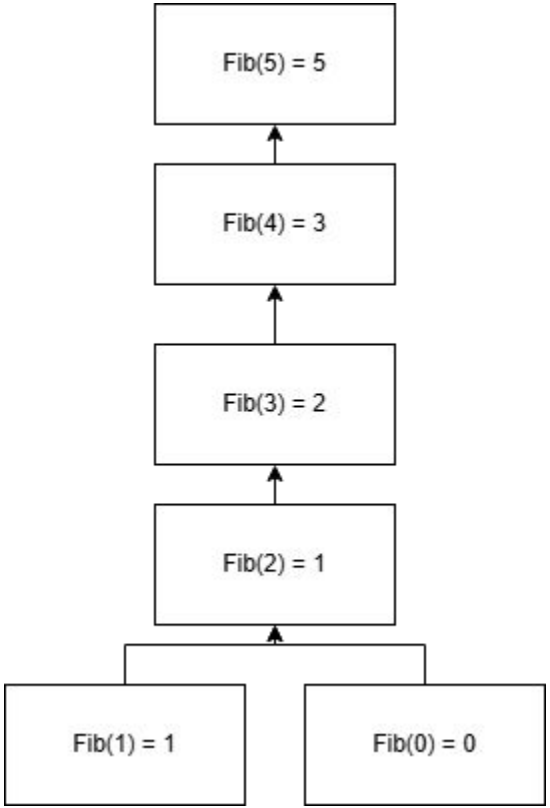
Dynamic Programming - Example

- Tabulation:

```
function  fib(n) {
    if (n === 0 ) {
        return 0;
    } else if (n === 1) {
        return 1;
    } else {
        let table = new Array(n+2);
        table[0] = 0;
        table[1] = 1;

        for (let i = 2; i <= n; i++)
        {
            table[i] = table[i-1] + table[i-2];
        }
        return table[n];
    }
}
```

Dynamic Programming - Example Visualised



Dynamic Programming

- Differences:
  - Top-down approach
    - Is a recursive problem solving approach.
    - Starts from the large input size, reaches the smallest version of the problem (base case), and stores subproblem solutions from the base case to the larger problem.
  - Bottom-up approach
    - Is an iterative problem solving approach.
    - First store the solution for the base case and store subproblem solutions in a particular order from the base case to the larger sub-problem.

Algorithms - Sorting

- Sorting Algorithm is used to arrange elements of an array (or list) in a specific order.
- Types:
  - Bubble sort:
    - Algorithm that sorts an array from the lowest value to the highest value by looping through every value of an array, comparing it to the next value
    - Time complexity:
      - O(n^2)

Algorithms - Sorting

- Selection sort:
  - Algorithm that finds the lowest value in the unsorted portion of the array and moves it to the sorted portion of the array.
  - Time complexity:
    - O(n^2)
- Insertion Sort:
  - Algorithm that uses one part of the array to hold sorted values, and another part to hold values that are not sorted yet.
    - Takes one value at a time from the unsorted part and inserts it in the right position in the sorted part of the array.
  - Time complexity:
    - O(n^2)

Algorithms - Sorting

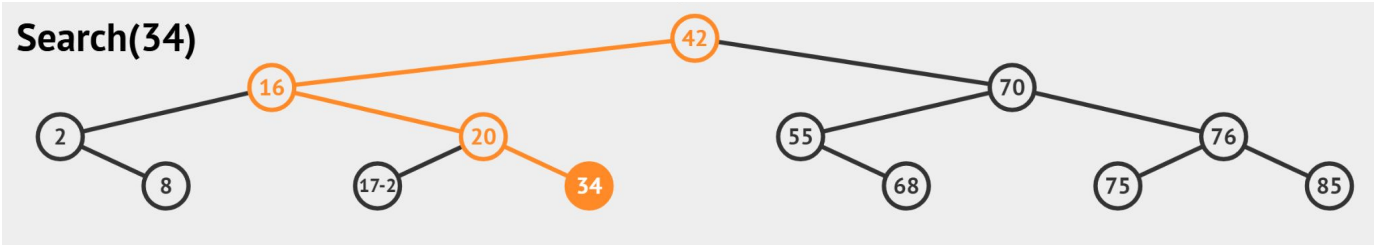
- Quick Sort:
  - Algorithm that takes an array of values, and uses a value as a pivot element, and moves lower values left of the pivot and higher values right of the pivot.
    - After which it swaps the pivot element for the first element of the higher values.
  - Time complexity:
    - O($n$log$n$)

Algorithms - Searching

- Searching algorithms are used to locate specific items within a collection of data.
  - Efficiently navigating through data structures to find the desired information.
- Types:
  - Linear Search:
    - Algorithm searches each value of an array to find the searched for value and returns the index of that value.
    - Time complexity:
      - $O(n)$

Algorithms - Searching

- Binary Search:
  - Algorithm searches a sorted array by comparing the center value to the desired value, if it is lower it searches the left half of the array, if higher it searches the right half of the array.
    - It reduces the array to the half it found from the center value, narrowing the search at each iteration.
  - Time complexity:
    - $O(\log_2 n)$



Class Examples

Palindrome and Anagram

Class Example - Palindrome

- Problem Statement:
  - Given a string as an input, validate whether it is a palindrome or not.
- Palindrome:
  - A word that is written forwards and backwards the same.
- Example:
  - Kayak -> [True]
  - Lesson -> Nossel [False]

Class Example - Palindrome

- Time complexity:
  - O(n)
- Space complexity:
  - O(n)

```
function isPalindrome(s) {
  s = s.toLowerCase().replace(/[_\W]/g,'');
  for (let i = 0, j = s.length - 1; i <= j; i++, j--) {
    if (s[i] !== s[j]) {
        return false;
  }
  return true;
}
```

Class Example - Palindrome

- Time complexity:
  - O(n)
- Space complexity:
  - O(1)
  - Constant

```
function isAlphanumeric(char) {
    const code = char.charCodeAt(0);
    return (code >= 97 && code <= 122) || (code >= 48 && code <= 57);
}

function isPalindrome(s) {
    s = s.toLowerCase();
    let i = 0, j = s.length - 1;

    while (i < j) {
        while (i < j && !isAlphanumeric(s[i])) {
          i++;
        }

        while (i < j && !isAlphanumeric(s[j])) {
          j--;
        }

        if (s[i] !== s[j]) {
          return false;
        }

        i++;
        j--;
    }

    return true;
}
```

Class Example - Anagram

- Problem Statement:
  - Given two strings as the input, validate whether they are anagrams of each other.
- Anagram:
  - A word made by transposing the letters of another word.
- Example:
  - Below, Elbow [True]
  - Cat, Act [True]

Class Example - Anagram

- Time complexity:
  - O(n)
- Space complexity:
  - O(n)

```
function isAnagram(s, t)
{
    let counter1 = new Array(256).fill(0);
    let counter2 = new Array(256).fill(0);

    for (let i = 0; i < s.length &&
         i < t.length; i++)
    {
        counter1[s[i].charCodeAt(0)]++;
        counter2[t[i].charCodeAt(0)]++;
    }

    if (s.length != t.length)
        return false;

    for (i = 0; i < 256; i++)
    if (counter1[i] != counter2[i])
        return false;

    return true;
}
```

Class Example - Anagram

- Time complexity:
  - O(n)
- Space complexity:
  - O(1)
  - Constant

```
function isAnagram(s, t) {
    if(s.length != t.length) {
        return false;
    }

    let alphabetDif = new Array(26).fill(0);

    for(let i = 0; i < s.length; i++) {
        alphabetDif[s.charAt(i).charCodeAt(0) - 'a'.charCodeAt(0)]++;
        alphabetDif[t.charAt(i).charCodeAt(0) - 'a'.charCodeAt(0)]--;
    }

    return alphabetDif.every(index => index === 0);
};
```

Homework

Apply what we have learned

Homework

- Solve the TwoSum problem.
  - Given an array of integers and a target, return the two numbers in the array that add up to target.
- Bonus:
  - Given a string, find the longest substring without repeating characters.
- Create new solution directory in GitHub and push solution.

- Make sure you are up to date until day 12 with the Blog app, so you are up to date before we start file upload tomorrow.

Next Class

Class presentations