15-110 Quiz 4 Review

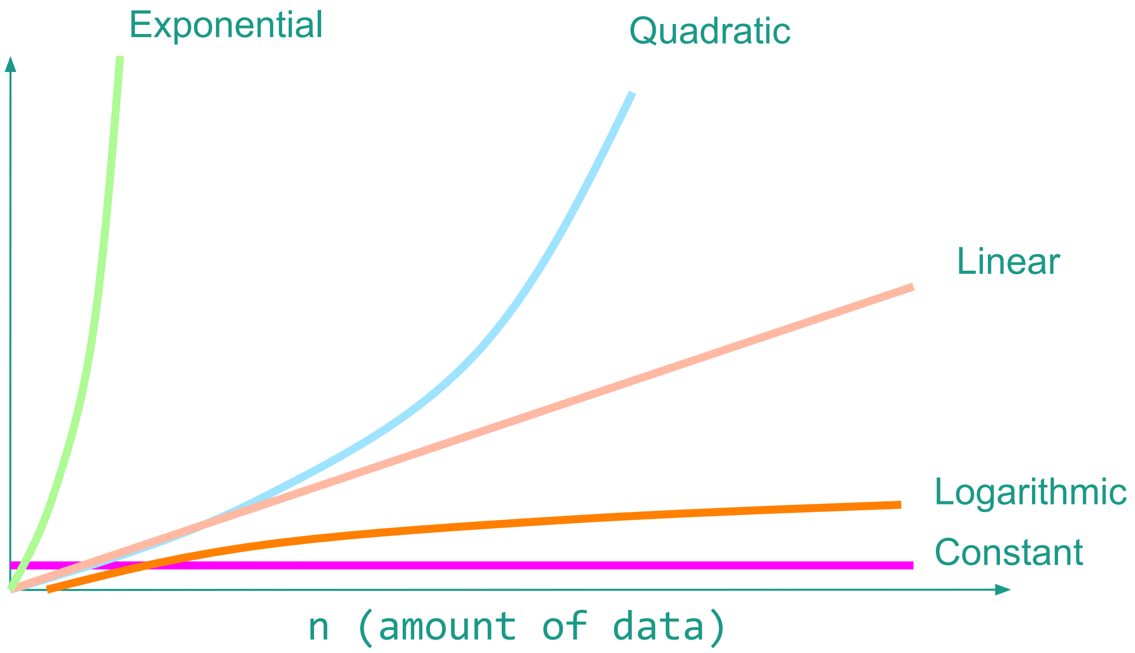
Runtime & Big-O Notation

- Whenever we compare the runtime of algorithms we want to know how they will perform with a **worst case** and **best case** input
    - **Best case:** an input of size n that results in the algorithm taking the least steps possible.
    - **Worst case:** an input of size n that results in the algorithm taking the most steps possible.

Function Families

We can see from the graph that exponential functions quickly skyrocket and quadratic functions grow rapidly compared to linear functions.

This is true even for very small inputs (n).

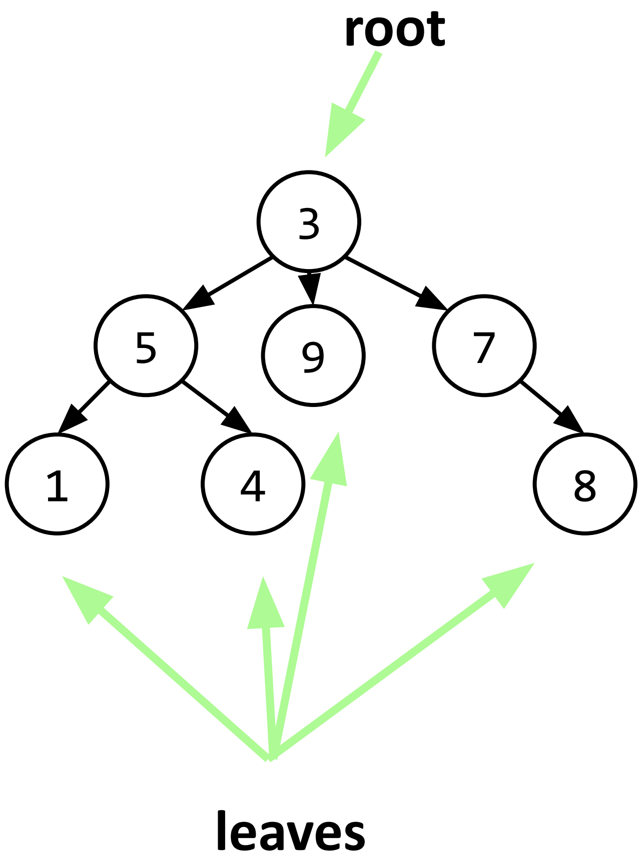


Trees

Root: the topmost node in the tree

Leaves: nodes that have no children

Nodes: each circle shown in the diagram (contains a value)

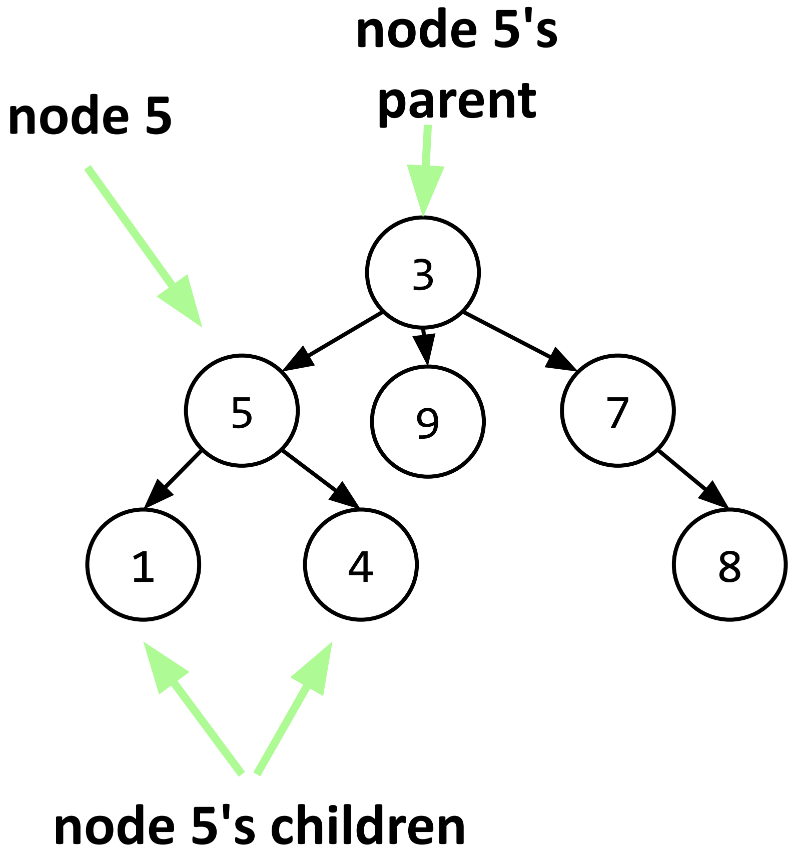


Trees - Parent/Child

(relative definitions)

Children: the node level below the parent node

Parent: the node level above a child node



Coding with Binary Trees

- Our trees will be implemented with a **NESTED dictionary**
- Each **node** of the tree will be a dictionary that has three keys
    - "contents" which maps to the value in the node
    - "left” which either maps to a node (dictionary) or None if there is no left child
    - "right" which either maps to a node (dictionary) or None if there is no right child.

```
t = { "contents" : 6,
      "left"     : { "contents" : 3,
                     "left"     : { "contents" : 8,
                                    "left"     : None,
                                    "right"    : None
},
                     "right"    : { "contents" : 7,
                                    "left"     : None,
                                    "right"    : None }
},
      "right"    : { "contents" : 2,
                     "left"     : None,
                     "right"    : { "contents" : 9,
                                    "left"     : None,
                                    "right"    : None }
} }
```

EXERCISE: try drawing out the tree from the dictionary implementation above

Graphs

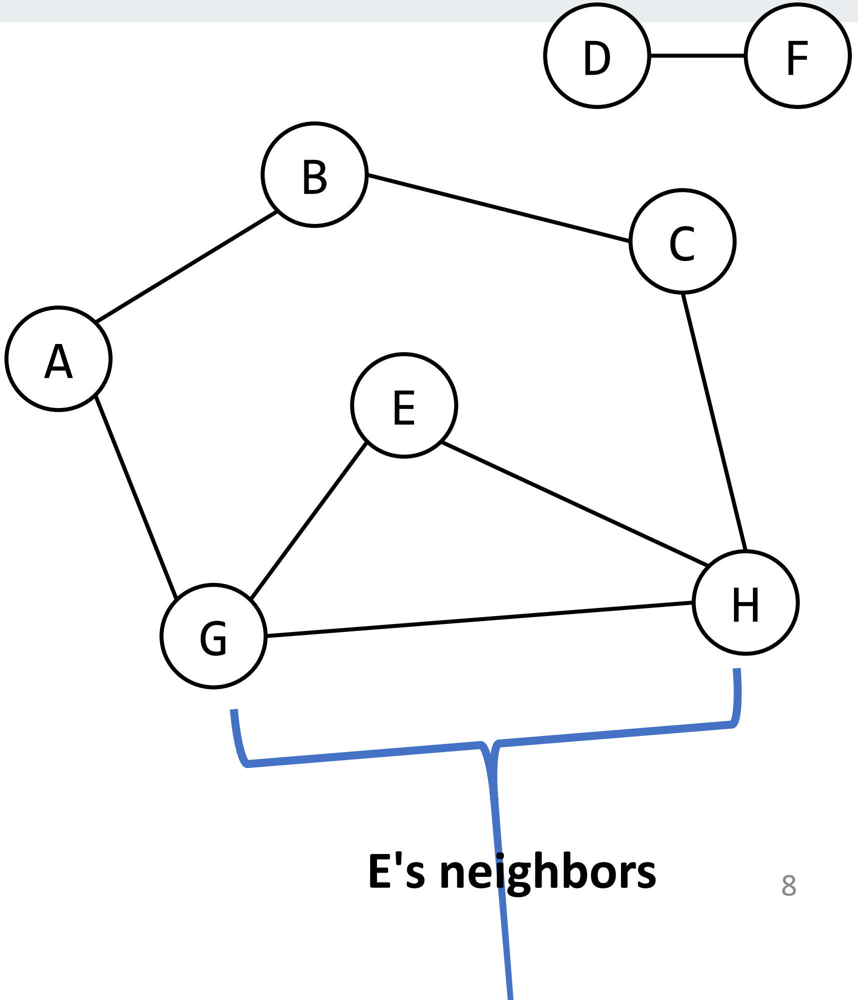
- Graphs are similar to trees with **fewer restrictions** for how nodes can be connected
  - Any node can be connected to any other node in the graph
- Graphs can be **directed** or **weighted** to represent more information
- We will be implementing graphs using adjacency lists (dictionary mapping to lists)

Graphs

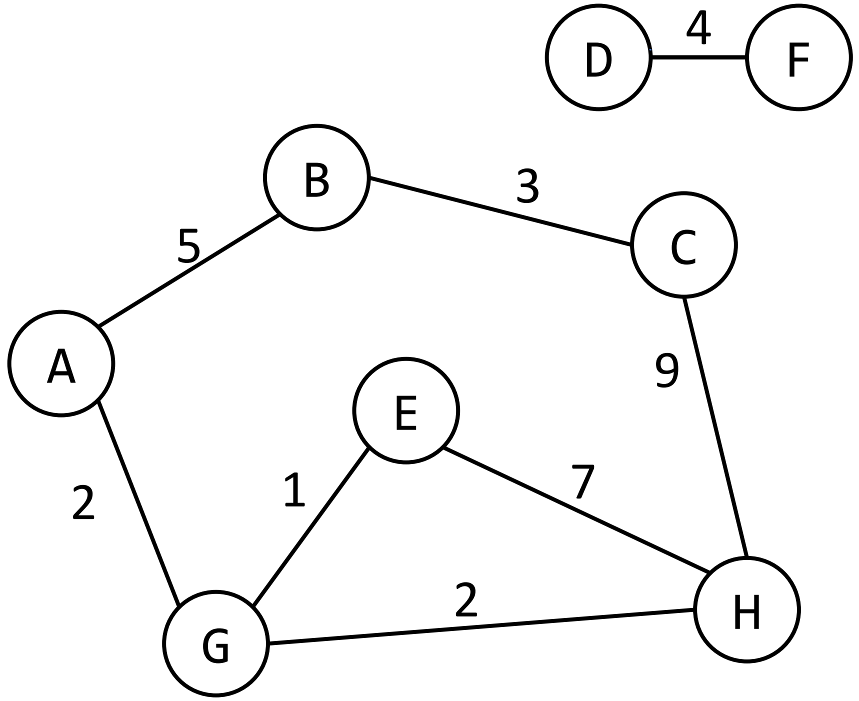
Nodes: hold the values stored in the structure

Edges: connections between nodes

Neighbours (for any node X): nodes that X connects to with an edge are X's **neighbors**.



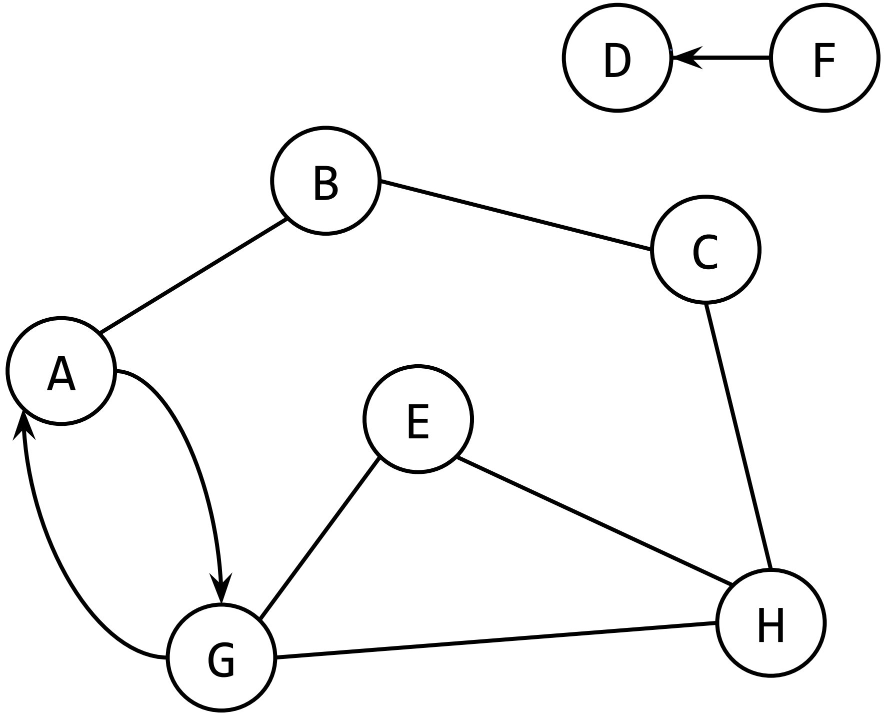
Weighted Graphs



- Graph edges can contain **weights** to indicate information about:
  - The length of a road
  - Cost of a flight
  - Distance between cities
  - Time to travel between 2 places
- These are indicated by numbers next to each edge

Directed Graphs

- Graph edges can also be **directed**
  - Each direction needs to be specified
  - (F to D) doesn’t imply (D to F)
- Undirected nodes go in either direction



Coding with Graphs

UNWEIGHTED

- The graph is a **dictionary** with keys representing the **value** of the nodes and mapping to a list of adjacent nodes (it’s neighbours).

```
unweightedGraph = {
          "A" : [ "B", "G" ],
          "B" : [ "A", "C" ],
          "C" : [ "B", "H" ],
          "D" : [ "F" ],
          "E" : [ "G", "H" ],
          "F" : [ "D" ],
          "G" : [ "A", "E", "H" ],
          "H" : [ "C", "E", "G" ]
        }
```

EXERCISE: try drawing out the graph from the dictionary implementation above

Coding with Graphs

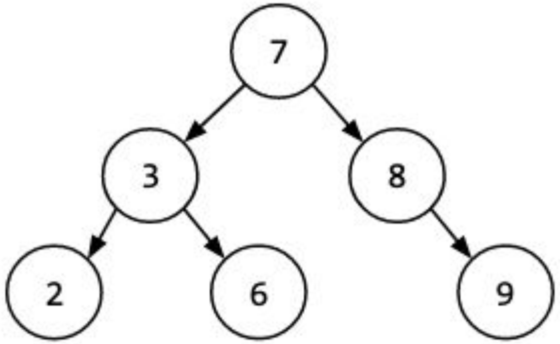
WEIGHTED

- Each of the inner lists becomes a node/edge pair
    - **First value:** represents the neighbouring node
    - **Second value:** represents the weight on the edge between the 2 nodes

```
weightedGraph = {
        "A" : [ ["B", 5], ["G", 2] ],
        "B" : [ ["A", 5], ["C", 3] ],
        "C" : [ ["B", 3], ["H", 9] ],
        "D" : [ ["F", 4] ],
        "E" : [ ["G", 1], ["H", 7] ],
        "F" : [ ["D", 4] ],
        "G" : [ ["A", 2], ["E", 1], ["H", 2]
],
        "H" : [ ["C", 9], ["E", 7], ["G", 2]
]
       }
```

EXERCISE: try drawing out the graph from the dictionary implementation above

Binary Search Tree



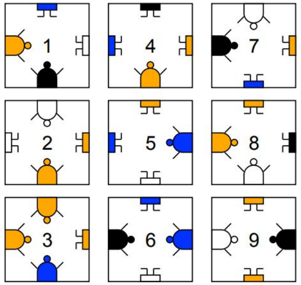
- at most two children per node (left & right)
- must be sorted
- binary search: ex. search for 6, search for 12
  - compare node to target value
  - if target value > node: search node’s right children
  - if target value < node: search node’s left children
  - recursive: child becomes new node and repeat until target value found or searched whole tree
- if balanced tree: for every node in the tree, the node's left and right subtrees are approximately the same size → runtime is O(log n)
- if unbalanced tree: if at least one node has significantly different sizes in its left and right children → runtime is O(n)

Breadth- vs. Depth-First Search

- Breadth-First Search: we search all immediate neighbors first, then search the neighbors of each of these previously visited nodes
- Depth-First Search: we go as far down a single path as we can, and backtrack to try all other possible paths
- *don’t revisit nodes
- worst case runtime for both: O(n)

Brute Force Approaches

- check every single possible solution
  - pros: easy to understand, implement, and test, and versatile
  - con: efficiency!
- Brute force approach: O(n!)
  - Travelling Salesperson Problem: find the shortest possible route that visits every city, then returns home.
  - Puzzle Solving:

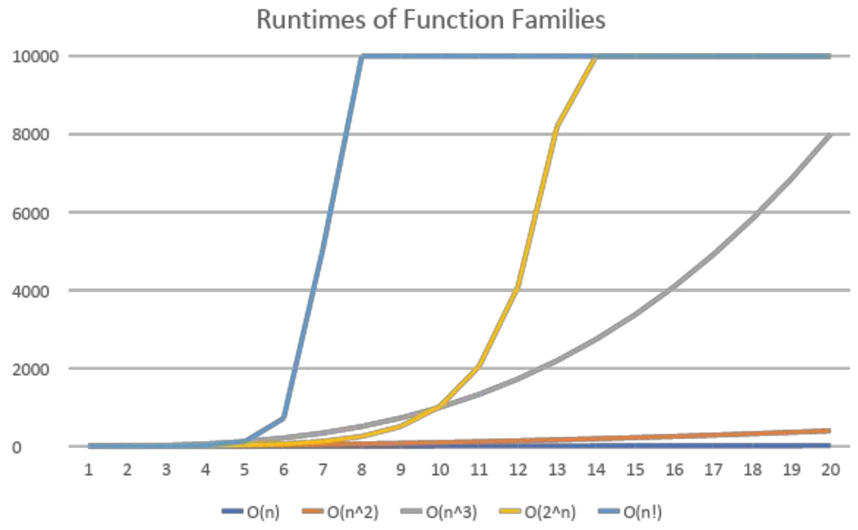


9 choices	8 choices	7 choices
6 choices	5 choices	4 choices
3 choices	2 choices	1 choice

- Brute force approach: O(2^n)
  - Subset Sum: generate all possible subsets, see if any of them sum to x.
- Brute force approach: O(k^n)
  - Exam Scheduling: generate a schedule that fits within the period and results in no student having two exams in the same slot

Tractability

- A problem is said to be **tractable** if it has a reasonably efficient runtime so that we can use it for practical input sizes
  - runtime is reasonable if it can be expressed as polynomial equation
  - ex. O(1), O(log n), O(n), O(n log n), O(n^2), and O(n^k)
- **intractable** ex. O(2^n), O(k^n), and O(n!)



Complexity Classes P & NP

- complexity class **P** to be the set of problems that we know can be **solved in polynomial time**
- complexity class **NP** to be the set of problems that can be **verified in polynomial time**
  - every algorithm in P is included in NP
- some problems are in neither classes, i.e. Travelling Salesperson Problem
- have not proven P = NP
  - pros: complex problems can be solved more quickly
  - cons: modern security and cryptography can be easily broken

Heuristics

- a search technique used by an algorithm to find a **good-enough** solution to a problem
- Travelling Salesperson Heuristic: ranks next-possible paths based on their length, choose the closest city next → generated in polynomial time
- Subset Sum Heuristic: order all the values in the list from largest to smallest, always try adding the largest available value to the subset first.