Data Structures:
Trees and Graphs
A tree is a hierarchical data structure composed of nodes.

- **Root:** the top-most node (unlike real trees, trees in computer science grow downward!). Every (non-empty) tree has one.
- **Parent:** the node connected directly above the current one. Every node (except for the root) has one.
- **Child:** a node connected below the current one. Each node can have 0 or more.
- **Leaf:** a node that has no children.
- **Depth/Level:** the length of the path (edges) from the root to a node (depth/level of the root is 0).
- **Tree Height:** the maximum depth from of any node in the tree.

A tree commonly used in computing is a **binary tree**.

- A binary tree consists of nodes that have at most 2 children.
- Commonly used in: data compression, file storage, game trees
Binary Tree Example

Which node is the root?
Which nodes are the leaves?
Which nodes are internal nodes?
What is the height of this tree?
Binary Tree Example

The root contains the data value 84.
There are 4 leaves in this binary tree: nodes containing 48, 37, 50, 53.
There are 4 internal nodes in this binary tree: containing 84, 65, 96, 24.
This binary tree has height 3 – considering root is at level 0,
the length of the longest path from root to a leaf is 3.
Binary Trees: A recursive structure!

- The yellow node with the key 65 can be viewed as the **root** of the left subtree, which in turn has
  - a left subtree of blue nodes
  - a right subtree of orange nodes

- In general, Binary Trees can be:
  - Empty
  - A root node with
    - a left binary tree
    - a right binary tree
A common implementation of binary trees uses nodes:
Each node has a “left” node and a “right” node.

How to represent these nodes and pointers? With a Class (like a Struct…)
We could also use a list to implement binary trees. For example:

```
0  1  2  3  4  5  6  7
-  65 24 50 48 37 - 14
```

Level 0

```
  65

  24

  48  37
```

Level 1

```
  50

  14
```

Level 2
A binary search tree (BST) is a binary tree with no duplicate nodes that imposes an ordering on its nodes.

BST ordering invariant: At any node $n$ with value $k$,
- all values of nodes in the left subtree of $n$ are strictly less than $k$
- all values of nodes in the right subtree of $n$ are strictly greater than $k$
**Example: Binary Search Tree**

**BST ordering invariant:** At any node with value $k$,
all values of elements in the left subtree are strictly less than $k$ and
all values of elements in the right subtree are strictly greater than $k$
(assuming that there are no duplicates in the tree)
Example: Is this a BST?

no

yes
For each data value that you wish to insert into the binary search tree:

- If you reach an empty tree (must test this first, why?), create a new leaf node with your value at that location
- Recursively search the BST for your value until you either find it or reach an empty tree
- If you find the value, throw an exception since duplicates are not allowed
Insertion Example

- Insert: 84, 41, 96, 24, 37, 50, 13, 98
Binary Search Tree Complexity

Tree 1

1
5
9

4
8
6

O(log n)

Tree 2

1
4
6
7
8
9

O(n)
Relationship Data

From this...
Relationship Data

To this...
A graph is a data structure that contains a set of vertices and a set of edges which connect pairs of the vertices.

- A vertex (or node) can be connected to any number of other vertices using edges.
- An edge may be bidirectional or directed (one-way).
- An edge may have a weight on it that indicates a cost for traveling over that edge in the graph.

Unlike trees, graphs can contain cycles

- In fact, a tree is an acyclic graph

Applications: computer networks, transportation systems, social networks
Example Graphs

Undirected

Directed
We usually represent graphs using a table (2d list) where each column and row is associated with a specific vertex. This is called an adjacency matrix.

A separate list of vertices shows which vertex name (city, person, etc.) is associated with each index.

The values of the 2d list are the weights of the edges between the row vertices and column vertices.

If there is not an edge between the two vertices, we use infinity, or None, to represent that.
Graph Representation

from to | A | B | C | D |
--------|---|---|---|---|
| A     | 0 | 6 | 7 | 5 |
| B     | 6 | 0 | 4 | ∞ |
| C     | 7 | 4 | 0 | 3 |
| D     | 5 | ∞ | 3 | 0 |

from to | A | B | C | D |
--------|---|---|---|---|
| A     | 0 | 6 | 7 | 5 |
| B     | ∞ | 0 | 4 | ∞ |
| C     | 2 | ∞ | 0 | 3 |
| D     | ∞ | ∞ | 9 | 0 |

weight

Diagram 1: Graph with weights

Diagram 2: Graph with weights
Graphs in Python

vertices = ['A', 'B', 'C', 'D']

graph =
[[0, 6, 7, 5],
 [6, 0, 4, None],
 [7, 4, 0, 3],
 [5, None, 3, 0]]
Graph Algorithms

- Lots! Here are some examples.

- There are algorithms to search graphs efficiently for a value
  - Breadth-first search and Depth-first search

- There are algorithms to compute the shortest path between a start vertex and all the others
  - Dijkstra’s algorithm

- There are algorithms for operations research, which can be used to solve network flow problems
  - For example, how to efficiently distribute water through a system of pipes
Shortest Path (Dijkstra's algorithm)

- Assign every node an initial distance (0 to the source, ∞ for all others); mark all nodes as unvisited.

- While there are unvisited nodes:
  - select unvisited node with smallest distance (current).
  - consider all unvisited neighbors of current node:
    - compute distance to each neighbor from current node.
    - if less than current distance, replace with new distance.
  - mark current node as visited (and never evaluate again).
Dijkstra example
What does this assume?