# Graphs

15-110 – Monday 10/21

# Learning Goals

Use **data structures** to represent data in different formats for different purposes

- Map **different types of data** to different data structures
- Use **graphs** to represent networked or heavily-connected data

Analyze and improve the **efficiency** of different programs

- Discuss how **searching** can be implemented efficiently in a variety of data structures

# Graphs

# Graphs are Like More-Connected Trees

Last week we discussed trees, which let us store data by connecting nodes to each other to create a hierarchical structure.

Graphs are like trees- they use nodes, and connect those nodes together. However, they have fewer restrictions on how nodes can be connected. **Any node can be connected to any other node in the graph**.

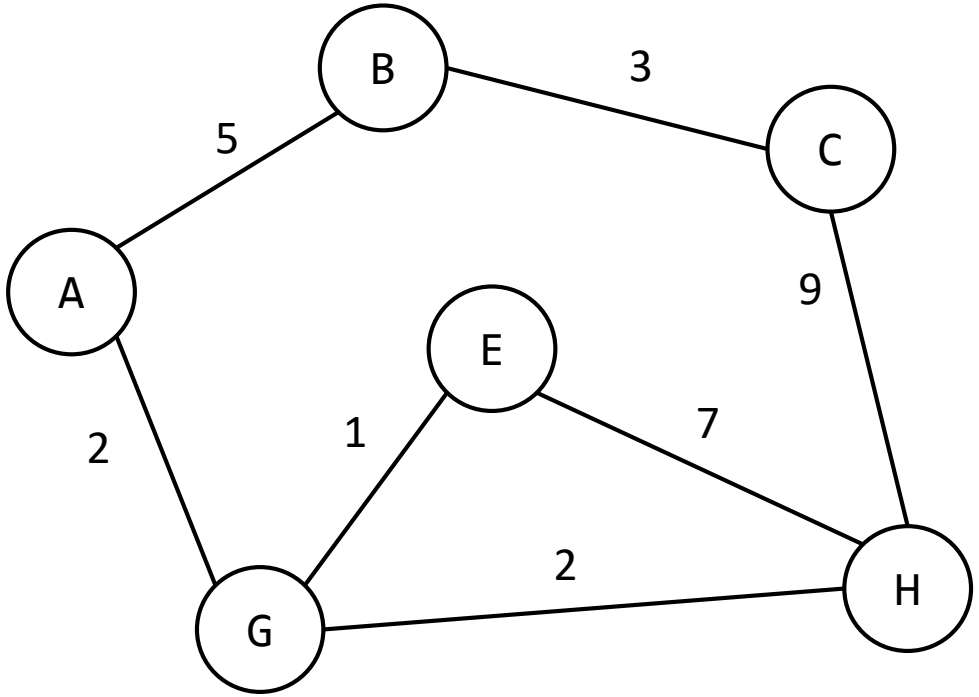
# Graphs are Made of Nodes and Edges

The **nodes** in a graph are the same as the nodes in a tree- they represent the values stored in the structure.

The **edges** of a graph are the connections between nodes. In a graph, these edges can sometimes have values too!

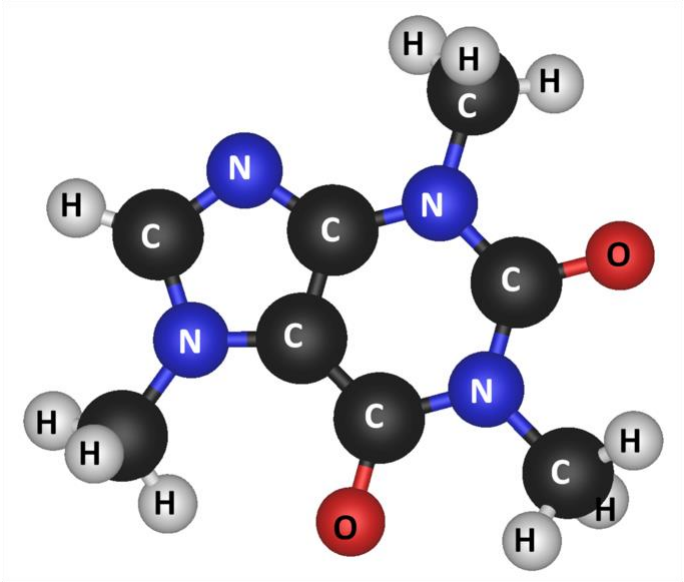
Edges can be **directed** (so that one can only go from A to B, and not vice versa), or **undirected** (so that one can go either way between two connected nodes).

The graph to the right is **undirected**; if it was directed, we'd add arrows to the lines to show directionality.



# Graphs in the Real World

Graphs show up all the time in real-world data! We can use them to represent **maps** (with locations as nodes and roads as edges) and **molecules** (with atoms as nodes and bonds as edges).




We also commonly use graphs in algorithms, to represent data like **social networks** (with people as nodes and friendships as edges), or **recommendation engines** (with items as nodes and edges between items that were bought together).

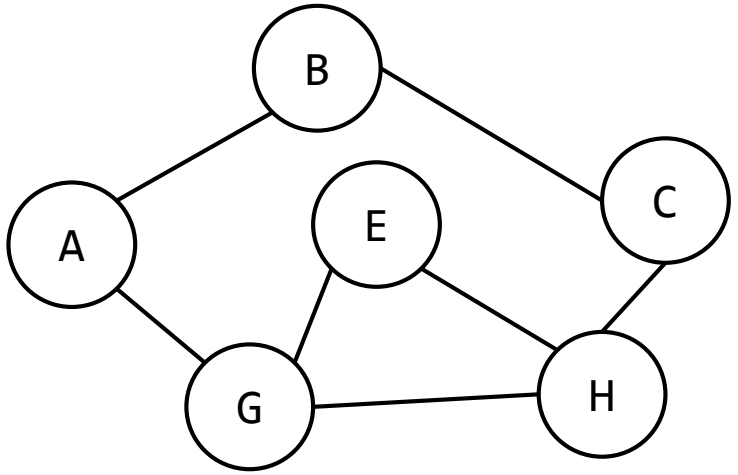
# Graphs in Python

Like trees, graphs are not implemented directly by Python. We need to use the built-in data structures to represent them.

We'll use two different implementation approaches: one for graphs where the edges don't have values, and one for graphs where edges do have values.

# Graphs in Python – Dictionary Approach

If the edges of a graph don't have values, we can use a similar **dictionary** implementation to the one we used for trees.




The keys of the dictionary will be the **values of the nodes**. They'll map to a **list of nodes that node is connected to**.

On the right, we show our example graph in its dictionary implementation.

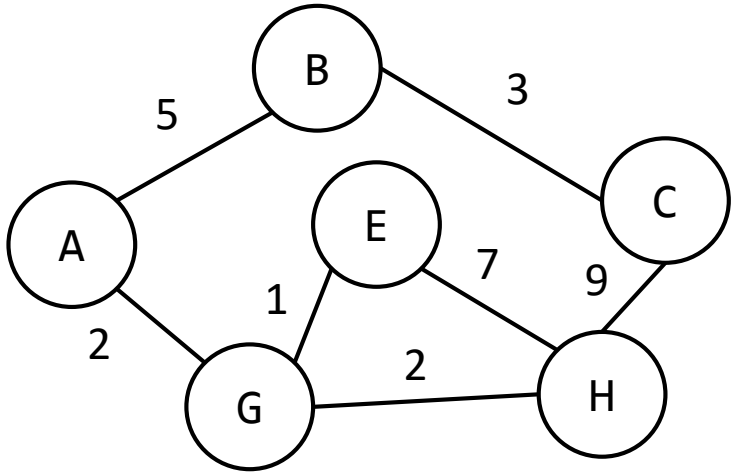
```
g = {
      "A" : [ "B", "G" ],
      "B" : [ "A", "C" ],
      "C" : [ "B", "H" ],
      "E" : [ "G", "H" ],
      "G" : [ "A", "E", "H" ],
      "H" : [ "C", "E", "G" ]
    }
```

# Graphs in Python – 2D List Approach

If the edges **do** have values, we can't use the dictionary approach; those values would be lost.

Instead, we'll create something called an **adjacency matrix**. This is a 2D list that is constructed so that matrix[i][j] holds the value of the edge between Node #i and Node #j, or None if those two nodes are not connected. We also use a 1D list to map the node numbers to their values.

On the right, we show our example graph in the adjacency matrix format.




```
nodes = [ "A",  "B",  "C",  "E",  "G",  "H" ]
g = [ #    A,    B,    C,    E,    G,    H
      [ None,    5, None, None,    2, None ], # A
      [    5, None,    3, None, None, None ], # B
      [ None,    3, None, None, None,    9 ], # C
      [ None, None, None, None,    1,    7 ], # E
      [    2, None, None,    1, None,    2 ], # G
      [ None, None,    9,    7,    2, None ]  # H
    ]
```

# Example: Most Connected Node

Let's write a function that takes a graph and returns the node in the graph with the most connected edges.

First, we'll solve this problem using a graph in the dictionary format. We just need to loop over each node and check the number of nodes it's connected to.

We'll also keep track of the best node and best connection-count so far, so we can return the best version at the end.

```
def mostConnectedNode(g):
    bestCount = 0
    bestNode = None
    for node in g:
        if len(g[node]) > bestCount:
            bestCount = len(g[node])
            bestNode = node
    return bestNode
```

# Example: Most Connected Node

Now let's write the function again, but using a graph in the 2D list format.

We're no longer given a list of only the connected nodes- we need to go through the list of edge values and count the number of edges that are not None.

We also need to fetch each node's value from the nodeList, instead of using the index directly.

Otherwise, the approach is the same!

```
def mostConnectedNode(matrix, nodeList):
    bestCount = 0
    bestNode = None
    for i in range(len(matrix)):
        count = 0
        for j in range(len(matrix[i])):
            if matrix[i][j] != None:
                count = count + 1
        if count > bestCount:
            bestCount = count
            bestNode = nodeList[i]
    return bestNode
```

# Searching a graph

Now let's attempt a familiar but more complicated problem: search.

We want to search a graph starting from a specific node to see if it is connected to a node we're looking for. We'll return True if we find the sought node, and False if it isn't connected at all.

**Discuss:** how can we approach this algorithm?

# Two search approaches: BFS and DFS

We'll need to start at the start node and follow the edges to find all the other nodes it's connected to. There are two common approaches for determining which neighbor node to visit first.

In **Breadth-First Search (BFS)**, we slowly move outwards in the graph from the start point. We visit all the direct neighbors of start, then visit all the direct neighbors of the visited nodes, etc., until we've checked all the nodes which were connected in the graph.

In **Depth-First Search (DFS)**, we go all the way down one potential path, then backtrack and try other possible paths. So we choose one neighbor, then choose one of its neighbors, etc., until there are no unvisited neighbors left.

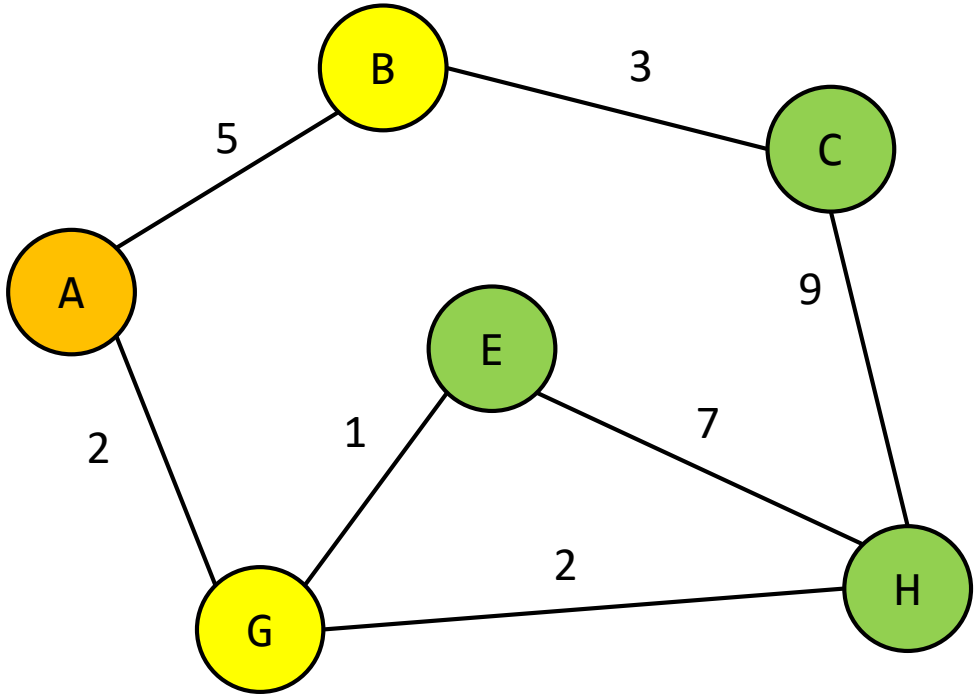
# Breadth-First Search Example

Let's say we want to run Breadth-First Search on our example graph, starting from A and searching for a value not in the graph, Q.

A has two neighbors- B and G. We can visit B and then G, or G and then B.

Once both have been visited, we visit B and G's neighbors- C, E, and H. Again, these can be visited in any order (CEH, CHE, ECH, EHC, HEC, HCE). A is a neighbor as well, but we don't visit it, because it's been visited before.

Then there are no non-visited nodes left, so we're done!



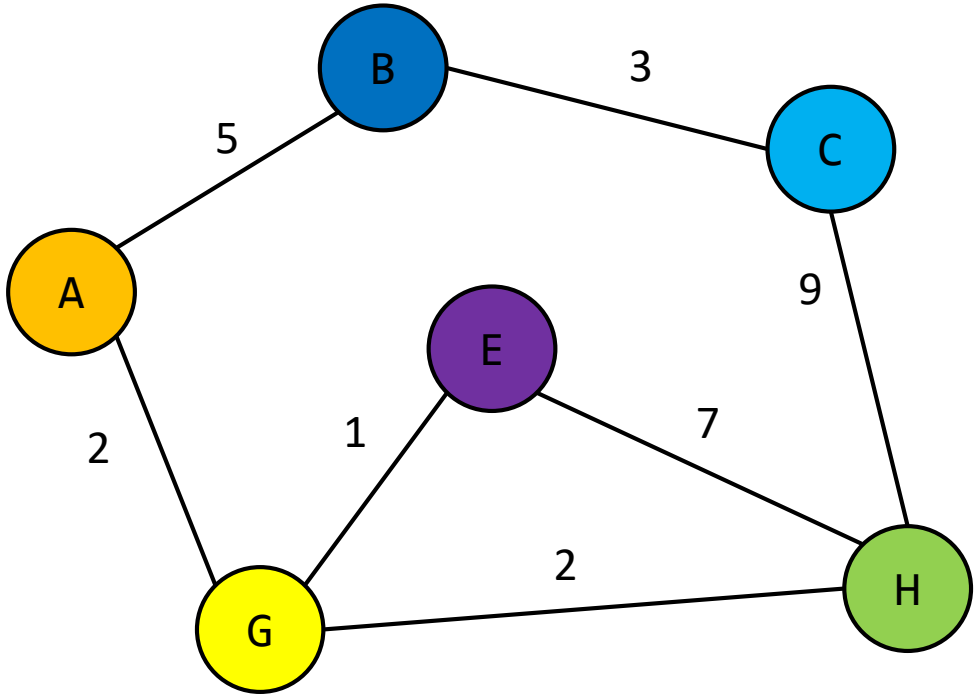
# Depth-First Search Example

Now let's search the example graph starting from A with depth-first search, still looking for Q.

There are two possible starting routes: B or G. Let's choose G. From G, we have two possible routes, E or H; we'll choose H.

From H, we have two more possible routes: E or C. We'll choose C. C's only remaining neighbor is B, so we must visit it.

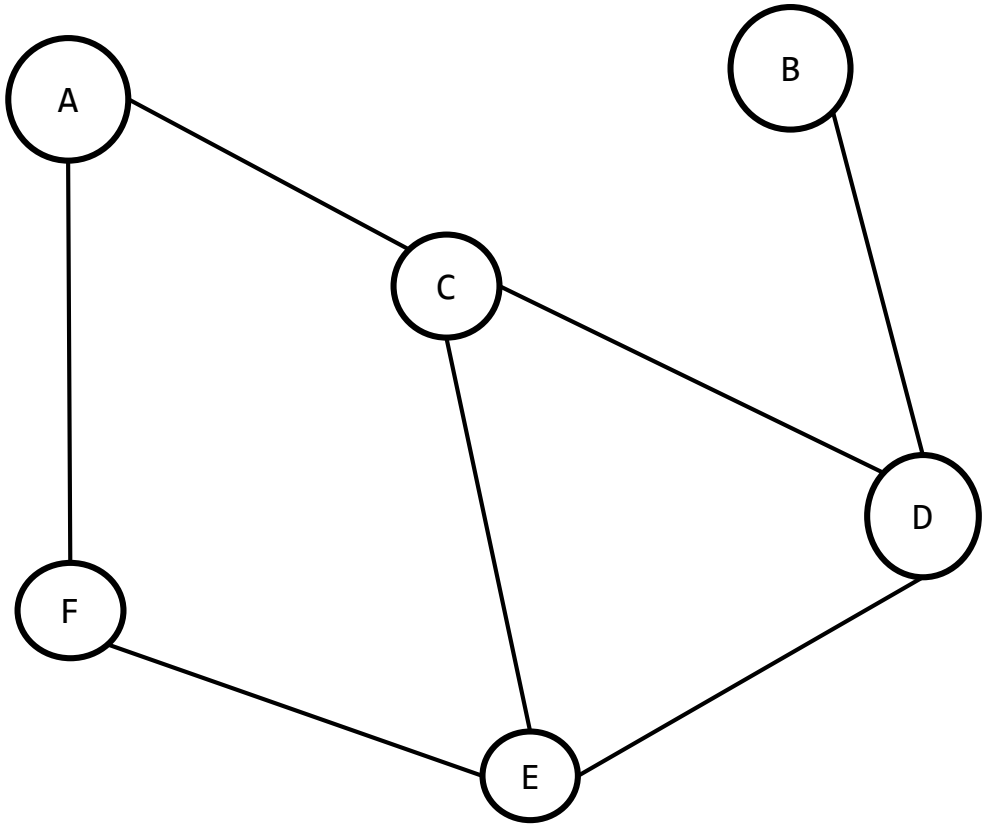
Now B has no neighbors remaining, so we must **backtrack** to the last node that had an unvisited neighbor. That's H with neighbor E; we visit E, and we're done.



# Activity: BFS and DFS Tracing

Given the graph to the right and starting from A, which potential trace through the graph is a valid trace for **Breadth-First Search,** and then for **Depth-First Search**?

Choose your answer on Piazza. (Note that not all possible answers are included).



# Coding BFS and DFS

To code these search algorithms, we'll need to keep track of two pieces of data. One is the **nodes we need to search next**. The other is **the nodes we've already visited**. It's important to keep track of what we've visited so far, to avoid cycling back to nodes we've seen before and looping forever!

We'll use a **while loop** to iterate over the nodes we need to search, since we'll update the list as we go. Each iteration will check the next node that hasn't been visited yet on the to-search list, to see if it's the one we're looking for.

If we find the node, we'll return True right away! If we don't, we'll add all the node's neighbors to the to-visit list. **How we add the nodes changes based on whether we implement BFS or DFS**.

# Breadth-First Search Code

Note that in the BFS code, we add neighbors of each node we visit to the **end** of the to-visit list. This prioritizes neighbors that are connected earlier in the graph.

```
def breadthFirstSearch(g, start, item):
    # Set up two lists for visited nodes and to-visit nodes
    visited = [ ]
    nextNodes = [ start ]

    # Repeat while there are nodes to visit
    while len(nextNodes) > 0:
        next = nextNodes[0]
        nextNodes = nextNodes[1:]

        # Only check this node if we haven't visited it before, to avoid repeats
        if next not in visited:
            visited.append(next)

            if next == item: # If it's what we're looking for- we're done!
                return True
            else: # Otherwise, add the neighbors to the back of the to-visit list
                nextNodes = nextNodes + g[next]
    return False
```

# Depth-First Search Code

In the DFS code, we add neighbors of each node we visit to the **start** of the to-visit list. This prioritizes neighbors that are connected deeper inside the graph. Otherwise, the algorithm is the same!

```
def depthFirstSearch(g, start, item):
    # Set up two lists for visited nodes and to-visit nodes
    visited = [ ]
    nextNodes = [ start ]

    # Repeat while there are nodes to visit
    while len(nextNodes) > 0:
        next = nextNodes[0]
        nextNodes = nextNodes[1:]

        # Only check this node if we haven't visited it before, to avoid repeats
        if next not in visited:
            visited.append(next)

            if next == item: # If it's what we're looking for- we're done!
                return True
            else: # Otherwise, add the neighbors to the front of the to-visit list
                nextNodes = g[next] + nextNodes
    return False
```

# Testing BFS and DFS

We only change one line of code between BFS and DFS, but it makes a big difference in the way the algorithms work. The test code to the right demonstrates how the algorithm moves through the nodes of an example graph for two different nodes.

When we search for "D", BFS is more efficient- it only needs to check three nodes, compared to DFS's four. But when we search for "E", DFS only takes three moves, compared to BFS's five!

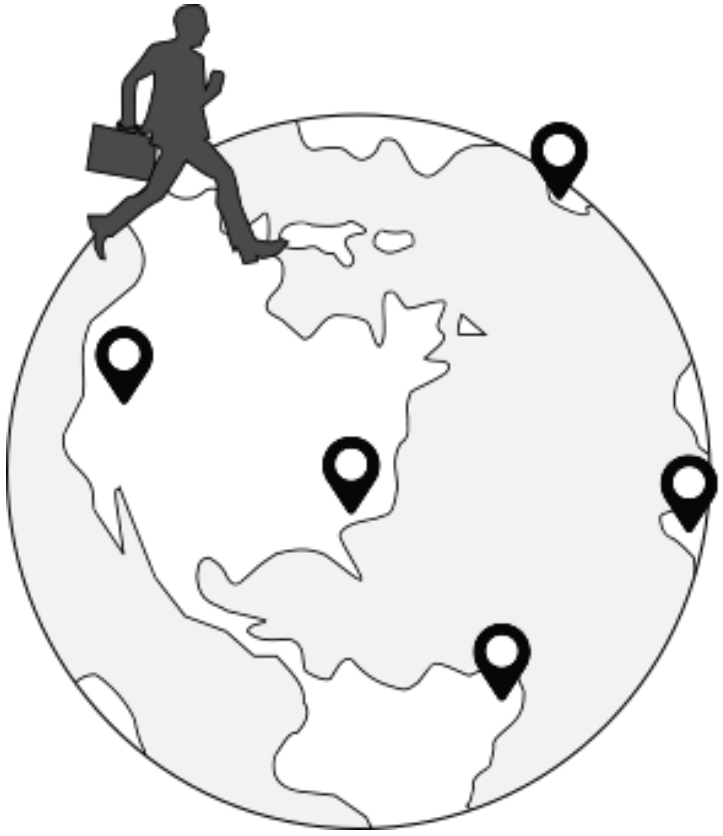
Both algorithms are **O(n)**, where n is the number of nodes in the graph, because both must check every node in the graph in the case that the sought value doesn't exist.

```
g = { "A" : [ "B", "D", "F" ],
      "B" : [ "A", "E" ],
      "C" : [ ],
      "D" : [ "A", "E" ],
      "E" : [ "A", "B", "D" ],
      "F" : [ "A" ]
    }

breadthFirstSearch(g, "A", "D")
breadthFirstSearch(g, "A", "E")

depthFirstSearch(g, "A", "D")
depthFirstSearch(g, "A", "E")
```

# Graph Example: Travelling Salesperson Problem

There are many classic algorithmic problems that involve graphs, because graphs can be difficult to interact with.

One especially well-known problem is called the **Travelling Salesperson problem**. You're given a list of cities to visit, and a graph of distances between cities. You want to find the shortest possible route that lets you visit every city, then gets you back home.



# One Solution: Brute Force

One intuitive way to solve the Travelling Salesperson problem is to plan out **every possible route** from the starting city across all the others, then choose the shortest route of them all.

This type of approach- generating all possibilities and then comparing them- is called a **brute force approach**. It's a simple and intuitive way to solve problems, but it does have drawbacks.

# Brute Force Efficiency

Consider the efficiency of a brute-force approach. Let's say that generating a path of n stops counts as one action. How many possible paths are there?

We have n possible first stops on the route. For each of those routes, there are n-1 possible second stops. Then there are n-2 third stops per route, etc... until there is only one city left for the last stop.

This means that the number of possible routes is n * (n-1) * (n-2) * ... * 1. **It's O(n!).**

Here's our big question: **can we find a faster way to solve the Travelling Salesperson problem?** We'll address this in Wednesday's lecture.

# Learning Goals

Use **data structures** to represent data in different formats for different purposes

- Map **different types of data** to different data structures
- Use **graphs** to represent networked or heavily-connected data

Analyze and improve the **efficiency** of different programs

- Discuss how **searching** can be implemented efficiently in a variety of data structures