Advanced Programming #3 – Dictionaries, Trees, and Graphs

CS SCHOLARS – PROGRAMMING

Learning Goals

Use **dictionaries** when writing and reading code that uses pairs of data

Use **binary trees** implemented with dictionaries when reading and writing code

Use **graphs** implemented as dictionaries when reading and writing simple algorithms in code

Data Structures Organize Data

So far, we've learned about several different types of data – numbers, strings, Booleans, lists. Lists in particular are interesting because they let us store *other data types* inside of them.

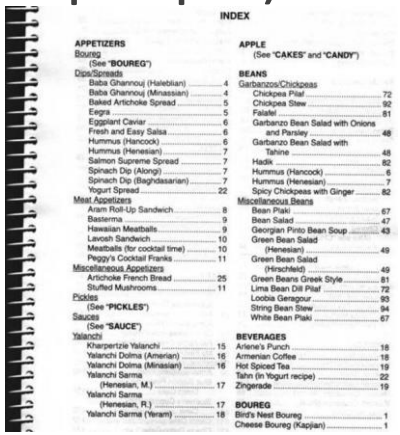
Lists aren't the only data type we have to work with! There are lots of ways to organize data. A list is a good for storing values in sequential (and indexed) order, but what other types of data might we work with?

Dictionaries

Python Dictionaries Map Keys to Values

The first data structure we'll discuss is the **dictionary**. Dictionaries store data in **pairs** by mapping **keys** to **values**.

We use dictionary-like data in the real world all the time! Examples include phonebooks (which map names to phone numbers), the index of a book (which maps terms to page numbers), or the CMU directory (which maps andrewIDs to information about people).



INDEX

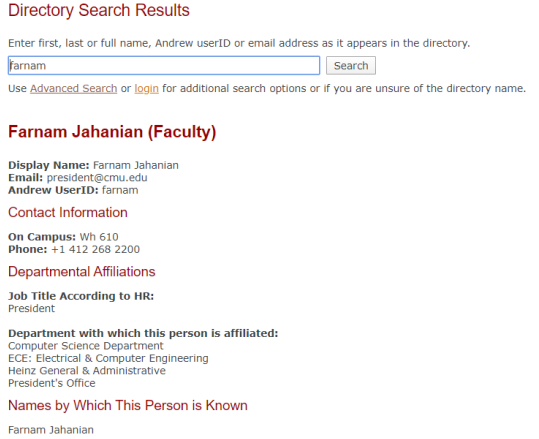
APPETIZERS
Boureg
(See "BOUREG")
Dips/Spreads
Baba Ghannouj (Halebian) 4
Baba Ghannouj (Minassian) 4
Baked Artichoke Spread 5
Eegra 5
Eggplant Caviar 6
Fresh and Easy Salsa 6
Hummus (Hancock) 6
Hummus (Henesian) 7
Salmon Supreme Spread 7
Spinach Dip (Alongi) 7
Spinach Dip (Baghdasarian) 7
Yogurt Spread 22
Meat Appetizers
Aram Roll-Up Sandwich 8
Basterma 9
Hawaiian Meatballs 9
Lavosh Sandwich 10
Meatballs (for cocktail time) 10
Peggy's Cocktail Franks 11
Miscellaneous Appetizers
Artichoke French Bread 25
Stuffed Mushrooms 11
Pickles
(See "PICKLES")
Sauces
(See "SAUCE")
Yalanchi
Kharpertzie Yalanchi 15
Yalanchi Dolma (Amerian) 16
Yalanchi Dolma (Minasian) 16
Yalanchi Sarma
(Henesian, M.) 17
Yalanchi Sarma
(Henesian, R.) 17
Yalanchi Sarma (Yeram) 18

APPLE
(See "CAKES" and "CANDY")

BEANS
Garbanzos/Chickpeas
Chickpea Pilaf 72
Chickpea Stew 92
Falafel 81
Garbanzo Bean Salad with Onions
and Parsley 48
Garbanzo Bean Salad with
Tahine 48
Hadik 82
Hummus (Hancock) 6
Hummus (Henesian) 7
Spicy Chickpeas with Ginger 82
Miscellaneous Beans
Bean Pilaki 67
Bean Salad 47
Georgian Pinto Bean Soup 43
Green Bean Salad
(Henesian) 49
Green Bean Salad
(Hirschfeld) 49
Green Beans Greek Style 81
Lima Bean Dill Pilaf 72
Loobia Geragour 93
String Bean Stew 94
White Bean Pilaki 67

BEVERAGES
Ariene's Punch 18
Armenian Coffee 18
Hot Spiced Tea 19
Tahn (In Yogurt recipe) 22
Zingerade 19

BOUREG
Bird's Nest Boureg 1
Cheese Boureg (Kapjian) 1



Directory Search Results

Enter first, last or full name, Andrew userID or email address as it appears in the directory.

farnam Search

Use Advanced Search or login for additional search options or if you are unsure of the directory name.

Farnam Jahanian (Faculty)

Display Name: Farnam Jahanian
Email: president@cmu.edu
Andrew UserID: farnam

Contact Information

On Campus: Wh 610
Phone: +1 412 268 2200

Departmental Affiliations

Job Title According to HR:
President

Department with which this person is affiliated:
Computer Science Department
ECE: Electrical & Computer Engineering
Heinz General & Administrative
President's Office

Names by Which This Person is Known

Farnam Jahanian

Key-Value Pairs

In a dictionary, a **key-value pair** is two values that have been paired together for organizational purposes. We'll be able to access the value by looking up the key, like how we can access a list value using its index.

For example, if we stored a phonebook in a dictionary, a **key** might be the string `"CMU"`, and its **value** would be the string `"412-268-2000"`. It wouldn't make sense to switch the roles because our default action is to look up a phone number based on a name, not vice versa.

Note: keys must be **immutable**, but values can be any type of data.

Python Dictionaries

Dictionaries have already been implemented for us in Python.

```
# make an empty dictionary
d = { }

# make a dictionary mapping strings to integers
d = { "apples" : 3, "pears" : 4 }
```

Python Dictionaries – Getting Values

Dictionaries are similar to lists. Instead of indexing by position, index by key:

```
d = { "apples" : 3, "pears" : 4 }
d["apples"] # the value paired with this key
len(d) # number of key-value pairs
```

If you try to access a key that doesn't exist, you'll get a runtime error.

```
d["ice cream"] # KeyError
```

We can also access all the keys or all the values separately:

```
d.keys()
d.values()
```

Python Dictionaries – Adding and Removing

How do we add a new key-value pair? Use **index assignment** with the key. This works whether or not that key has been assigned a value yet. If the key is already in the dictionary, the value for the key is updated; it does not add a new key-value pair.

```
d["bananas"] = 7 # adds a new key-value pair
d["apples"] = d["apples"] + 1 # updates the value
```

To remove a key-value pair, use **pop** with just the key as a parameter.

```
d.pop("pears") # destructively removes
```

Python Dictionaries – Search

We can **search** for a key in a dictionary using the built-in `in` operation.

```
d = { "apples" : 3, "pears" : 4 }
"apples" in d # True
"kiwis" in d # False
```

We can't use `in` to look up the dictionary's values; we need to loop over the keys and check each key's value instead. How do we loop over a dictionary?

Activity: Trace the code

After running the following code, what key-value pairs will the dictionary hold?

```
d = { "PA" : "Pittsburgh", "NY" : "New York City" }
d["WA"] = "Seattle"
d["NY"] = "Buffalo"
if "Pittsburgh" in d:
    d.pop("Pittsburgh")
```

Activity Answer

`"PA"` and `"Pittsburgh"`

`"NY"` and `"Buffalo"`

`"WA"` and `"Seattle"`

For Loops on Dictionaries

To loop over a dictionary, we must use a **for loop** directly over the dictionary. The loop visits all key-value pairs in some order. The loop control variable is set to the **key** of each key-value pair. To access the value, you must index into the dictionary with that key.

```
d = { "apples" : 5, "beets" : 2, "lemons" : 1 }
for k in d:
    print("Key:", k)
    print("Value:", d[k])
```

Activity: `countItems(foodCounts)`

You do: write the function `countItems(foodCounts)` that takes a dictionary mapping foods (strings) to counts (integers), loops over the key-value pairs, and returns the total amount of food stored in the dictionary. The function should also print the number of each individual food type as it counts up the total.

For example, if `d = { "apples" : 5, "beets" : 2, "lemons" : 1 }`, the function might print

```
5 apples
2 beets
1 lemons
```

then return `8`.

Activity Answer

```
def countItems(foodCounts):
    total = 0
    for food in foodCounts:
        print(foodCounts[food], food)
        total += foodCounts[food]
    return total
```

Coding with Dictionaries

Coding with Dictionaries – Track Information

We often use dictionaries when problem-solving. One common use of dictionaries is to **track information** about a list of values.

For example, given a list of students and their college (represented as "student,college"), how many students are in each college?

We will create a dictionary with college as the key and the student count as the value.

```
def countByCollege(studentLst):
    collegeDict = { }
    for student in studentLst:
        name = student.split(",")[0]
        college = student.split(",")[1]
        if college not in collegeDict:
            collegeDict[college] = 0
        collegeDict[college] += 1
    return collegeDict
```

Coding with Dictionaries – Find Most Common

We also use dictionaries to find the most common element of a list, by mapping elements to counts.

For example, given the dictionary returned by the previous function, which college is the most popular?

```
def mostPopularCollege(collegeDict):
    best = None
    bestScore = -1
    for college in collegeDict:
        if collegeDict[college] > bestScore:
            bestScore = collegeDict[college]
            best = college
    return best
```

Coding with Dictionaries – Nested Dictionaries

We can even use **nested** dictionaries in a similar way to how we use nested (2D) lists. Just map each key to another dictionary (which will map other keys to specific values).

For example, we can create a multiplication table in a nested dictionary (outer keys are x, inner keys are y, values are x*y).

```
def createMultDict(n):
    d = { }
    for x in range(1, n+1):
        innerD = { }
        for y in range(1, n+1):
            innerD[y] = x * y
        d[x] = innerD
    return d
m = createMultDict(4)
print(m[2][3]) # 6
```

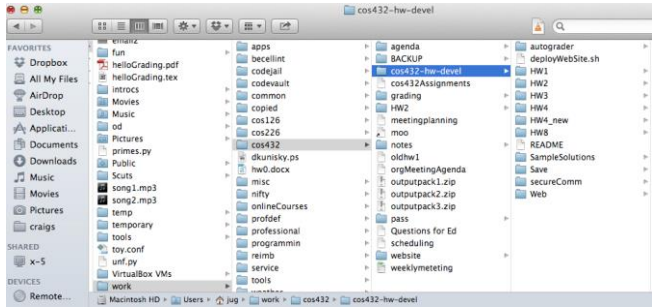
Trees

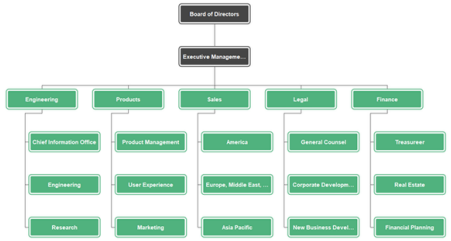
Trees Hold Hierarchical Data

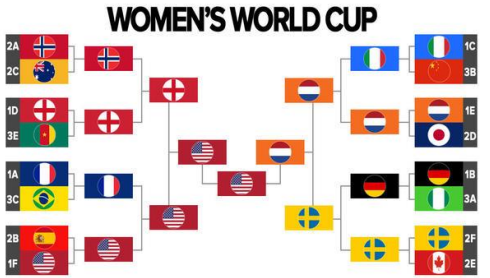
Sometimes we work with data that is **hierarchical** in nature. In this context, 'hierarchical' means that data occurs at different **levels** and is connected in some way.

Hierarchical data shows up in many different contexts.

- **File systems** in computers – each folder is a rank above the files it contains
- **Company organization schemas** – the CEO at the top, interns at the bottom
- **Sports tournament brackets** – the overall winner is ranked highest





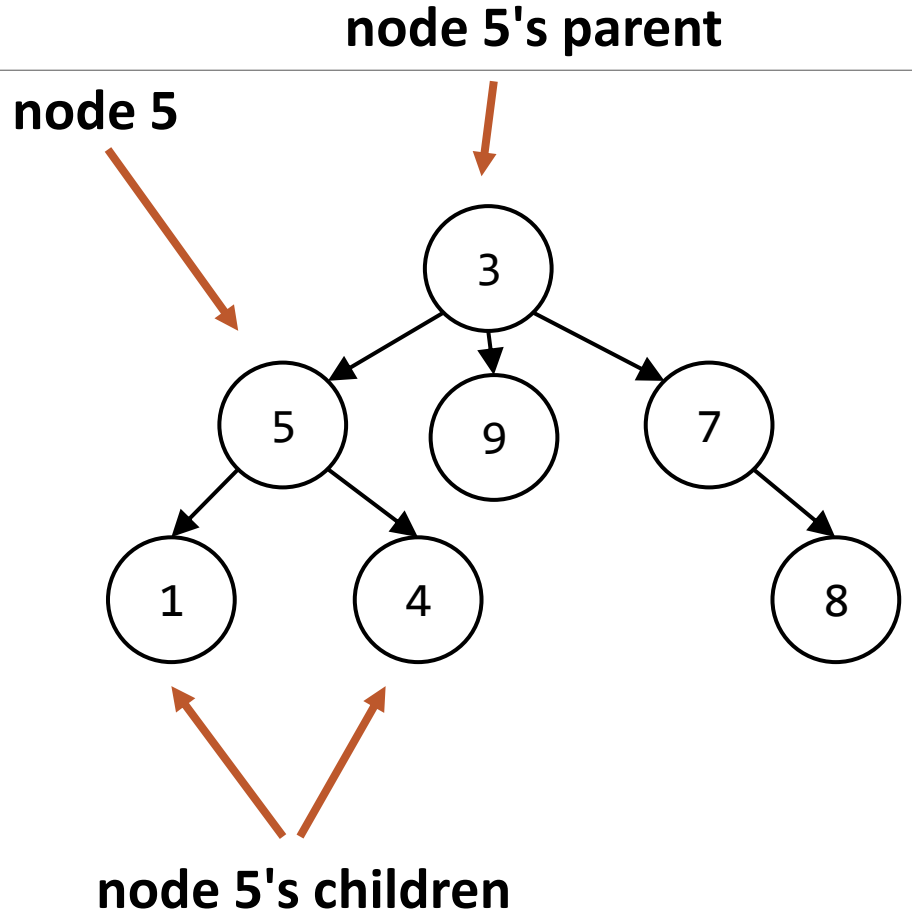


Trees are Hierarchical

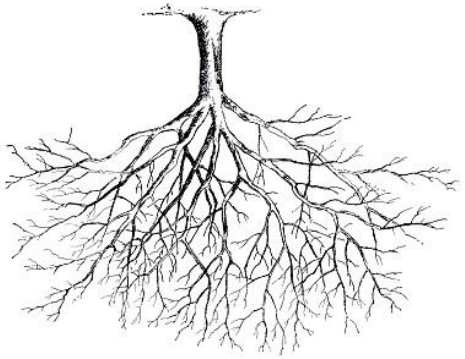
A **tree** is a hierarchical data structure composed of **nodes** (circles in the example shown to the right).

Each node can hold a **value** (its data).

The node the level above a node is called its **parent**, and nodes connected on the level below are called its **children**. In general, a node has exactly one parent and can have any number of children.



Trees are Upside-down

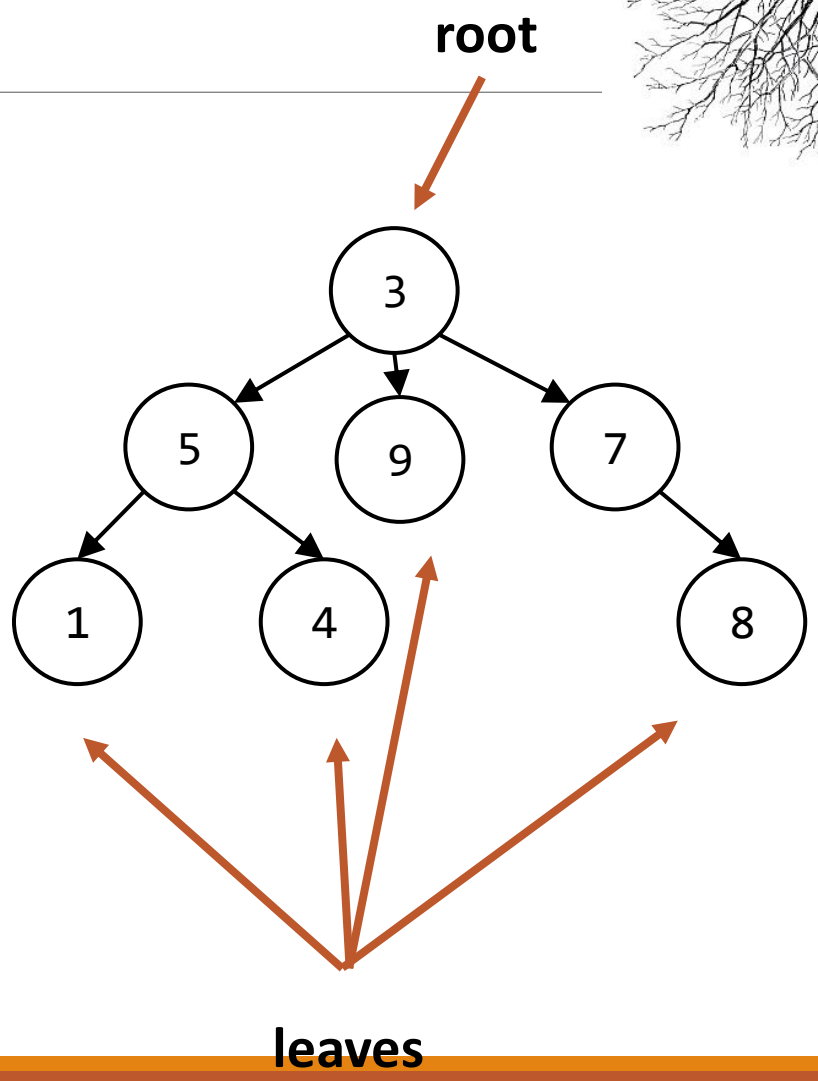


Unlike real trees, trees in computer science grow downward!

The top-most node is called the **root**. Every (non-empty) tree has a root. The root has no parent.

On the other hand, a node can have other nodes as children, and those nodes can have children as well. The number of levels a tree can have is unlimited.

Nodes that have no children are called **leaves**.



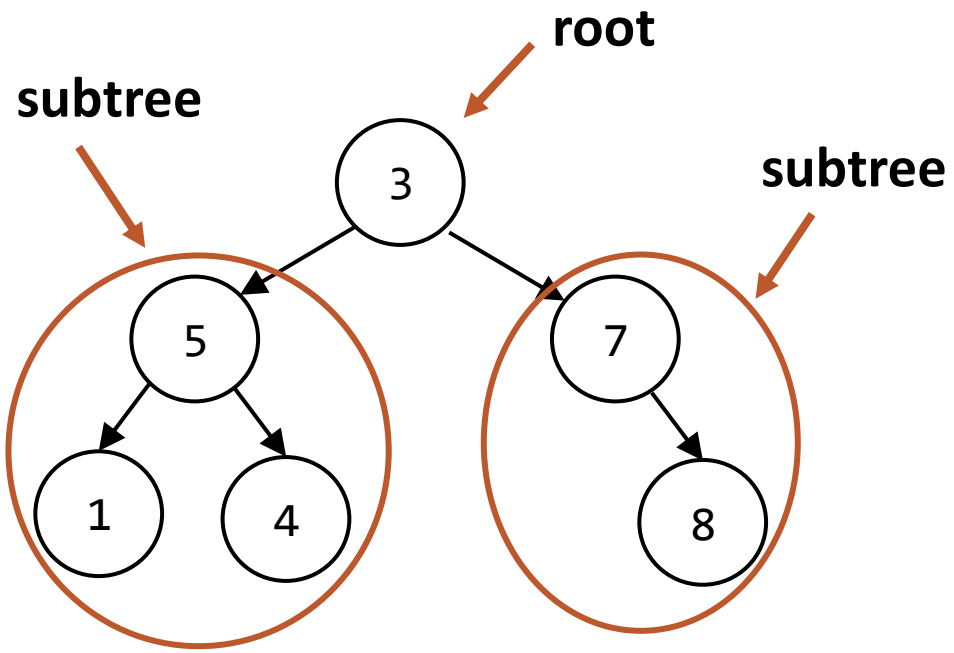
Trees are Recursive

A tree is a naturally recursive data structure. Each node's children are **subtrees**, which are just trees again.

For example, the root node 3 has two subtrees. The subtree on the left has a root node 5. The subtree on the right has a root node 7. Each of these root nodes have subtrees as children.

Our **base case** can be a leaf (or even an empty tree).

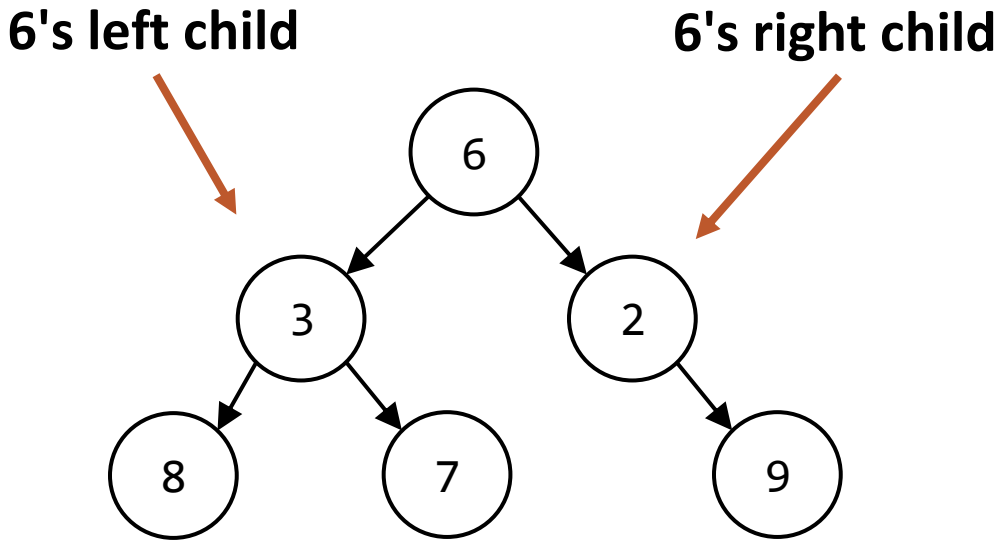
The **recursive case** makes the problem smaller by repeating on the children, which are also trees.



Binary Trees

It's possible to write algorithms for trees that have an arbitrary number of children, but in this class we'll focus on **binary trees**.

A binary tree is a tree that can have at most **2 children per node**. We assign these children names- **left** and **right**, based on their position.



Activity: Find the Tree Parts

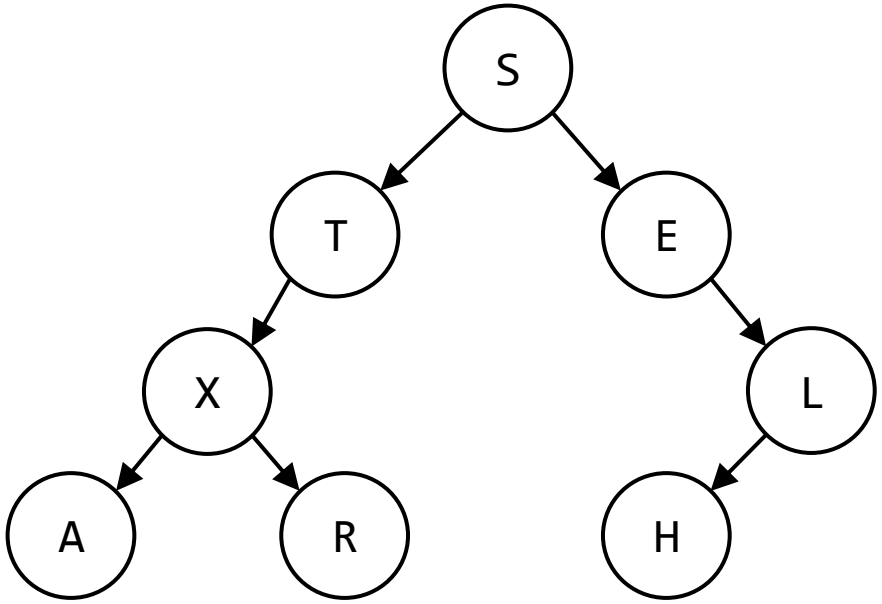
Given the tree shown to the right:

What is the **root**?

What are the **children** of node X?

What is the node X's **parent**?

What are the **leaves**?



Activity Answers

Root: S

Children of X: A, R

X's parent: T

Leaves: A, R, H

Coding with Trees

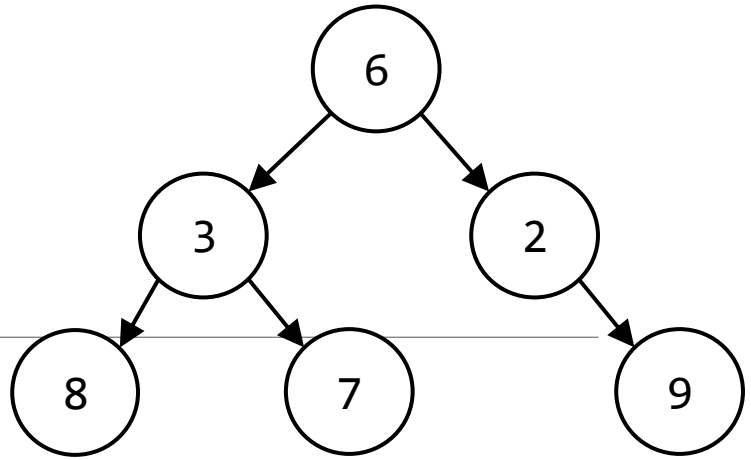
Implementing New Data Structures

Computer science uses a large number of classical data structures. Some of these (like lists and dictionaries) are implemented directly by Python. Others are not implemented directly; we need to design an implementation ourselves.

Python does not implement trees directly. We'll implement trees using **recursively nested dictionaries**.

Sidebar: these trees will be **mutable**; we can change the values in them and add/remove nodes. That's beyond the scope of this class, though.

Python Syntax – Trees as Dictionaries



Each **node** of the tree will be a dictionary that has three keys.

The first key is the string `"contents"`, which maps to the value in the node.

The second key is the string `"left"`, which either maps to a node (dictionary) if the node has a left child, or `None` if there is no left child.

The third key is the string `"right"`, which either maps to a node (dictionary) if the node has a right child, or `None` if there is no right child.

Our example tree is written as a dictionary to the right.

```
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 } } }
```

Simple Example: `getChildren(t)`

Given a tree, how can we get the children of the root node?

Access the `"left"` and `"right"` subtrees directly, then access their `"contents"`, *if they exist*.

Note that we use two separate `if`s, not an `if-elif`, because it's possible for both to be `True`.

```
def getChildren(t):
  result = []
  if t["left"] != None:
    leftT = t["left"]
    result.append(leftT["contents"])
  if t["right"] != None:
    rightT = t["right"]
    result.append(rightT["contents"])
  return result
```

Use Recursion When Coding with Trees

Because a tree is a recursive data structure, we'll usually need to use recursion to operate on trees.

The **base case** is when the current node is a leaf and we need to do something with its value.

In the **recursive case**, we'll call the function recursively on the left and then call again on the right child, if both exist. Usually we'll then combine those results in some way with the node's value.

Alternative approach: Make the base case when the tree is None (an empty tree) and always recurse on both left and right children in the recursive case. This can be more confusing to think about but is often simpler to program.

Example: countNodes

Let's write a program that takes a tree of values and counts the number of nodes in the tree.

The **base case**: return 1 (a single node).

The **recursive case**: add the counts of the left and right subtrees together if they exist, then add 1 more for the current node.

```
def countNodes(t):
  if t["left"] == None and \
     t["right"] == None:
    return 1
  else:
    count = 0
    if t["left"] != None:
      count += countNodes(t["left"])
    if t["right"] != None:
      count += countNodes(t["right"])
    return count + 1
```

Example: `countNodes` – Different Base Case

Alternatively, we could solve this by checking a different base case: whether the node is an empty tree (if the current node is `None`).

An empty tree has a `0` nodes; a non-empty tree has a number of nodes based on its two subtrees, plus the current node.

The difference here is that there are always recursive calls to both children, even if they might be `None`.

```
def countNodes(t):
    if t == None:
        return 0
    count = 0
    count += countNodes(t["left"])
    count += countNodes(t["right"])
    return count + 1
```

Example: `sumNodes(t)`

What if we instead wanted to add all the nodes in the tree? (Let's assume it's a tree of integers). Now we'll need to use the nodes' **values**.

Base case: directly return the value of the only node (the leaf).

Recursive case: combine the sums of the two subtrees (if they exist) with the current node's value.

Our code structure is very similar to `countNodes`, but now we're using `t["contents"]`.

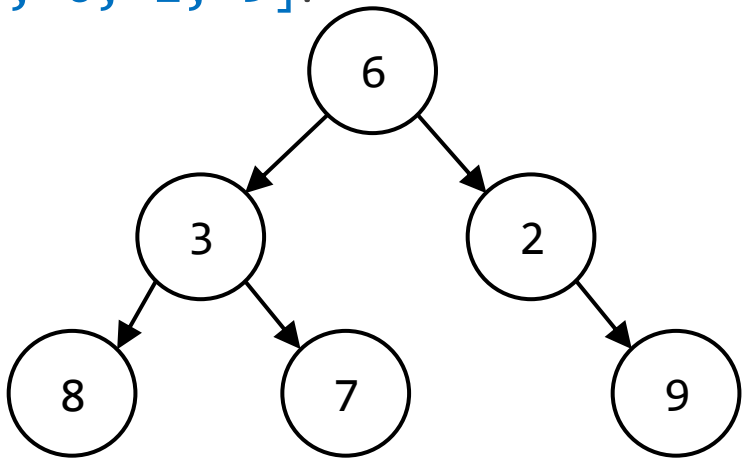
```
def sumNodes(t):
  if t["left"] == None and \
     t["right"] == None:
    return t["contents"]
  else:
    result = 0
    if t["left"] != None:
      result += sumNodes(t["left"])
    if t["right"] != None:
      result += sumNodes(t["right"])
    return result + t["contents"]
```

Activity: `listValues`

You do: write the function `listValues(t)`, which takes a tree and returns a list of all the values in the tree. The values can be in any order, but try to put them in left-to-right order if possible.

Hint: this is *almost* the same structure as `sumNodes`, but you need to consider the **type** of the values you'll return.

Given our example tree (shown below), the function returns: `[8, 3, 7, 6, 2, 9]`.



You can test your code by copying the example tree's implementation from the earlier slide.

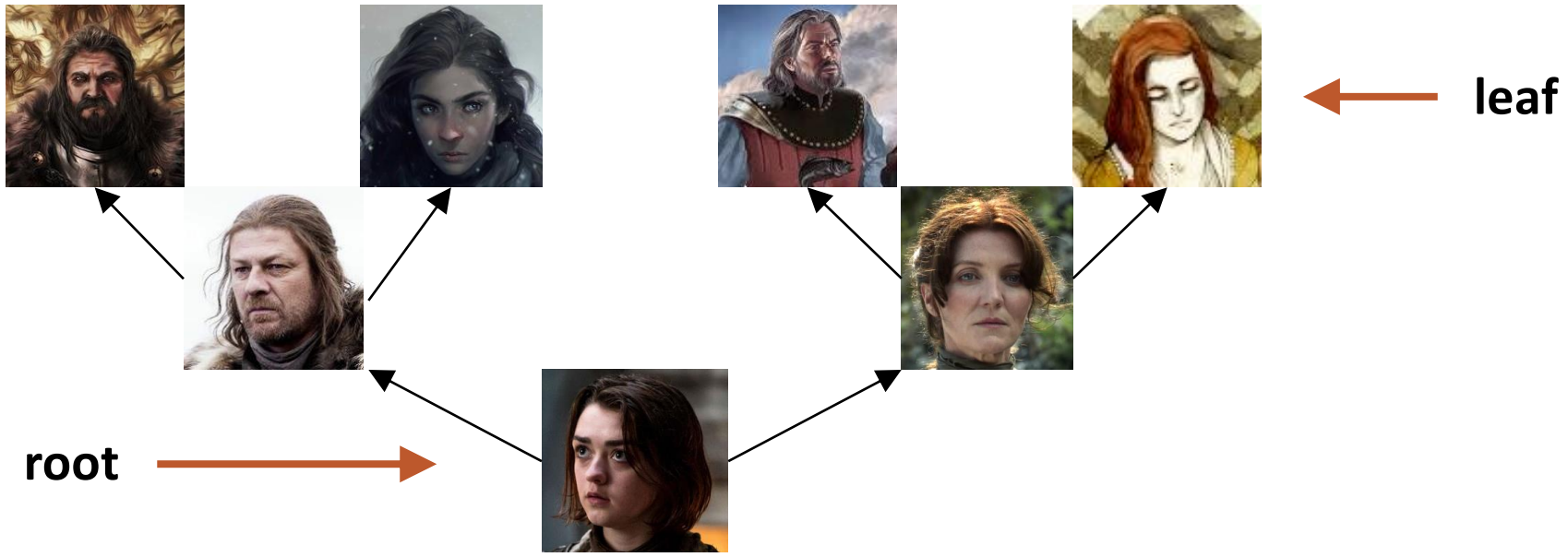
Activity Answer

```
def listValues(t):
    if t["left"] == None and t["right"] == None:
        return [t["value"]]
    else:
        values = []
        if t["left"] != None:
            values += listValues(t["left"])
        values.append(t["value"])
        if t["right"] != None:
            values += listValues(t["right"])
        return values
```

Advanced Example: Family Trees

Now let's write a function that takes a genealogical family tree as data.

We have to flip the tree – the child is at the root, their parents are the node's children, etc.



Advanced Example: `getPastGen`

Let's write a function that finds all the child's ancestors from N generations ago. N=1 would be their parents; N=2 would be grandparents; etc.

Note that for this problem, our base case is not a leaf- it's when we reach the generation we're looking for.

```
def getPastGen(t, n):
  if n == 0:
    return [ t["contents"] ]
  else:
    gen = [ ]
    if t["left"] != None:
      gen += getPastGen(t["left"], n-1)
    if t["right"] != None:
      gen += getPastGen(t["right"], n-1)
    return gen
```

Graphs

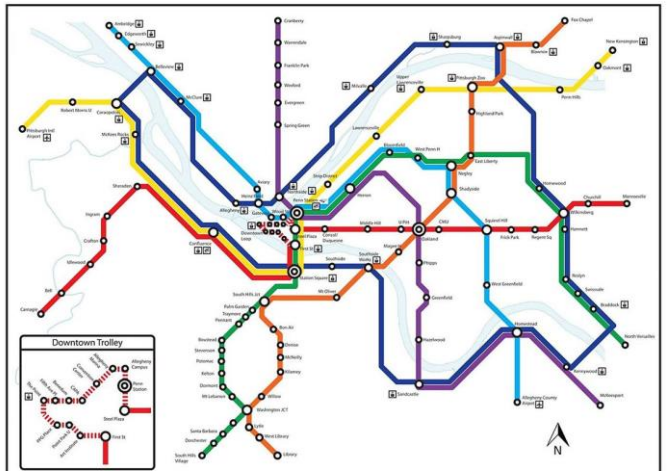
Graphs are Like More-Connected Trees

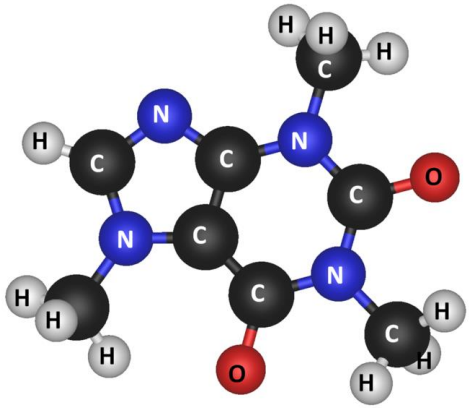
Last time we discussed trees, which let us store data by connecting nodes to each other to create a hierarchical structure.

Graphs are like trees – they are composed of 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 in the Real World

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





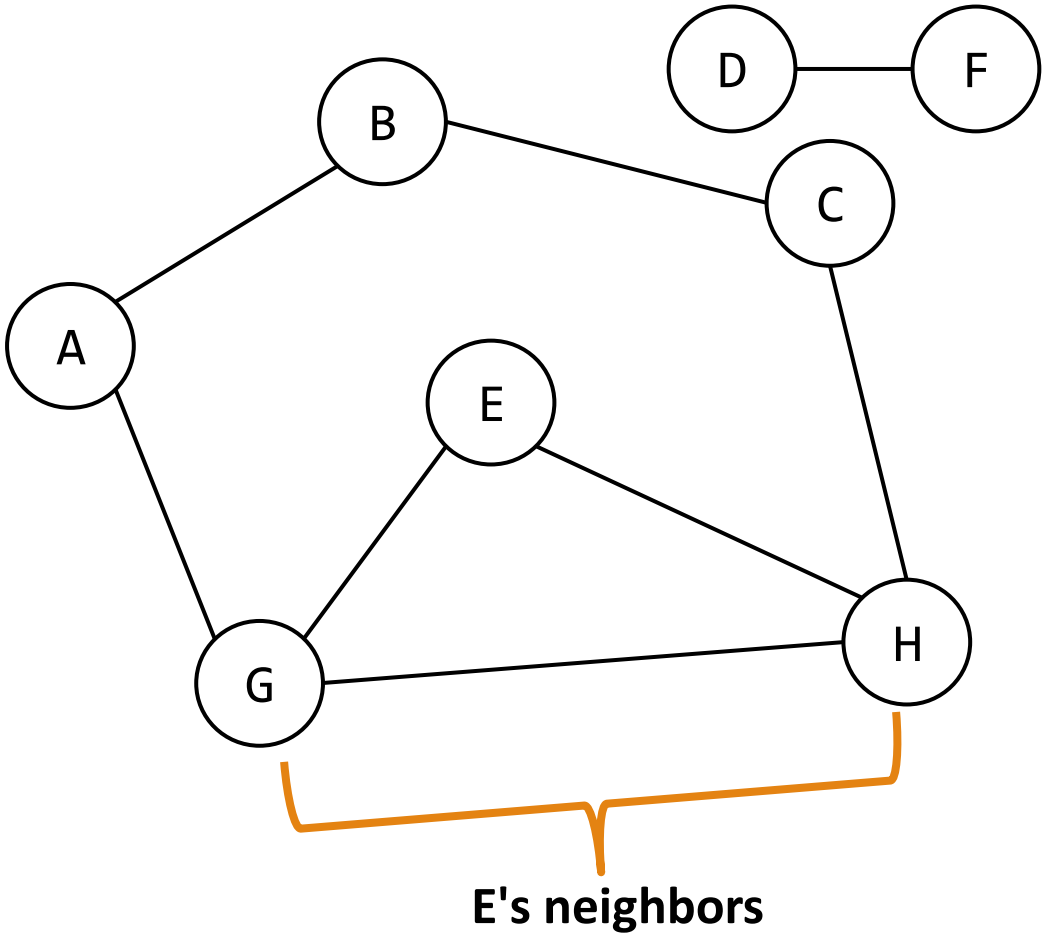
We also commonly use graphs in algorithms, to represent data like **social networks** (with people connected by friendships), or **recommendation engines** (with items connected if they were purchased together).

Graphs are Made of Nodes and Edges

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

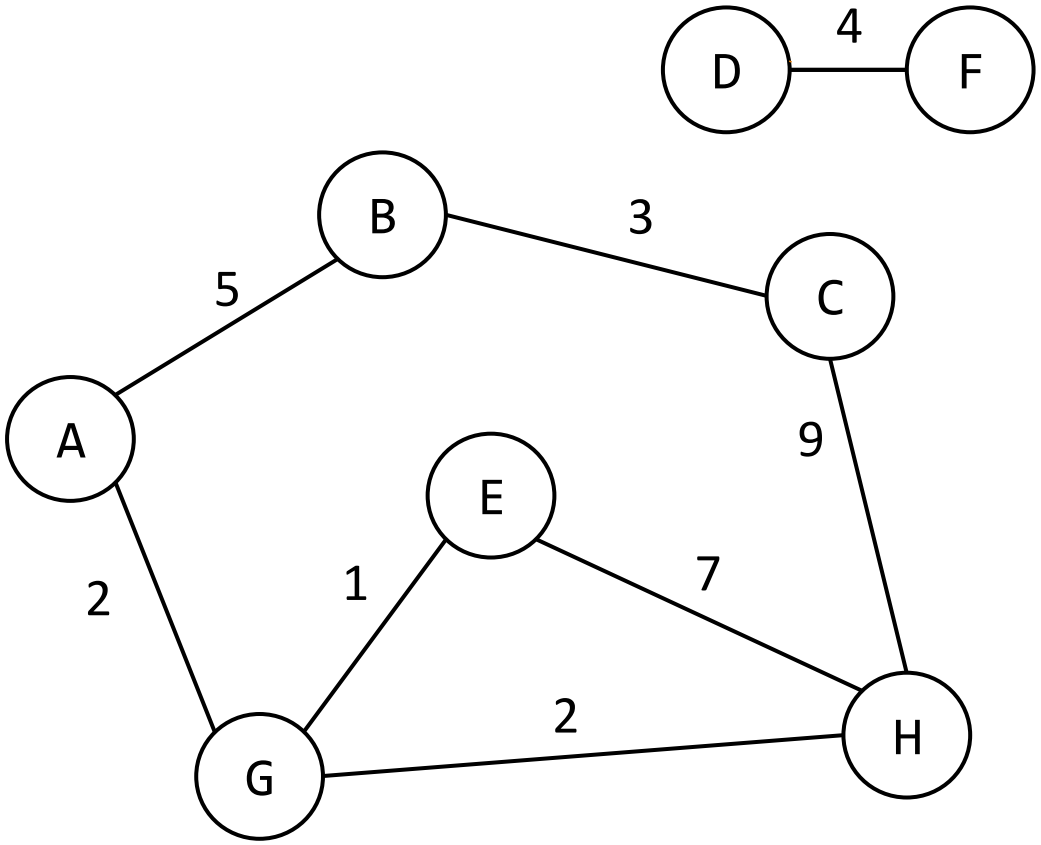
The **edges** of a graph are the connections between nodes.

We say that for a node X, any nodes that X connects to with an edge are X's **neighbors**.



Edges Can Have Weights

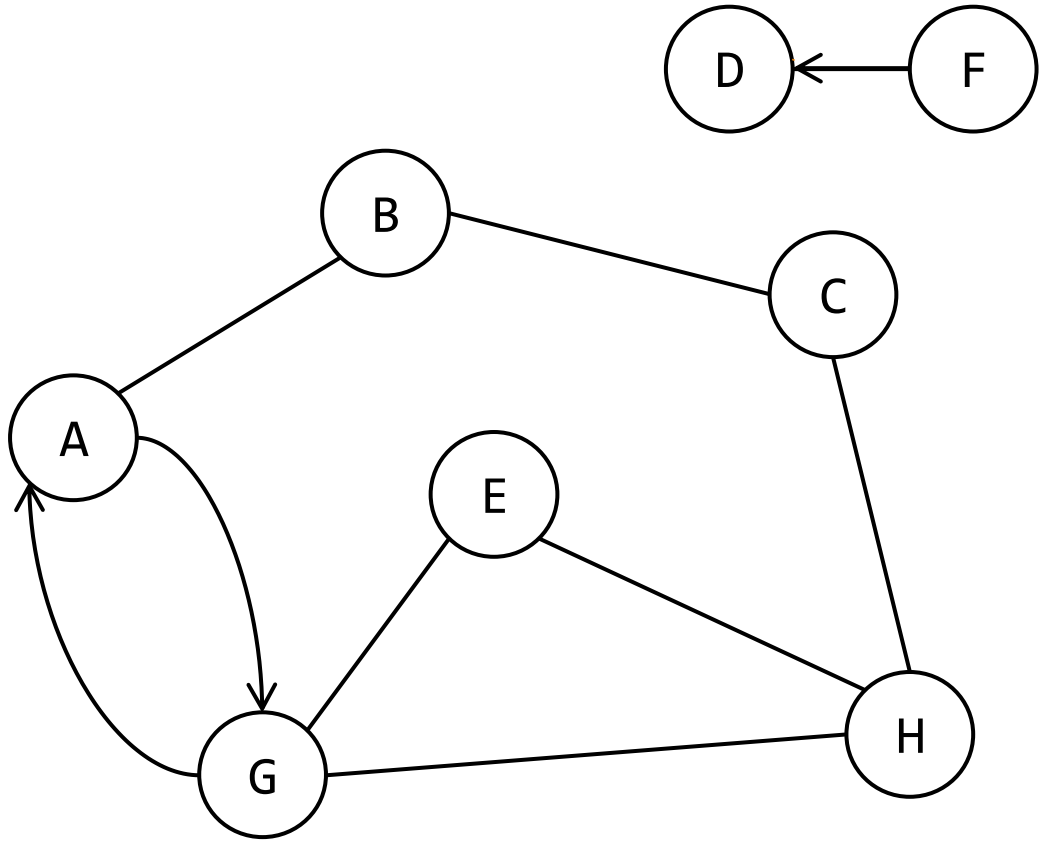
Sometimes edges can have **weights**, such as the length of a road or the cost of a flight. Our example graph here has weights- the numbers next to lines.



Edges Can Have Directions

Edges can also be **directed** (from A to B but not from B to A unless there is another directed edge from B to A), or **undirected** (go in either direction on an edge between nodes).

The main graph to the right is mostly undirected, except for the edge between nodes D and F and the edges between A and G, which are directed (notice the arrows). Usually directionality is not mixed like this in a graph.



Activity: Recognize the Parts

Consider the graph to the right.

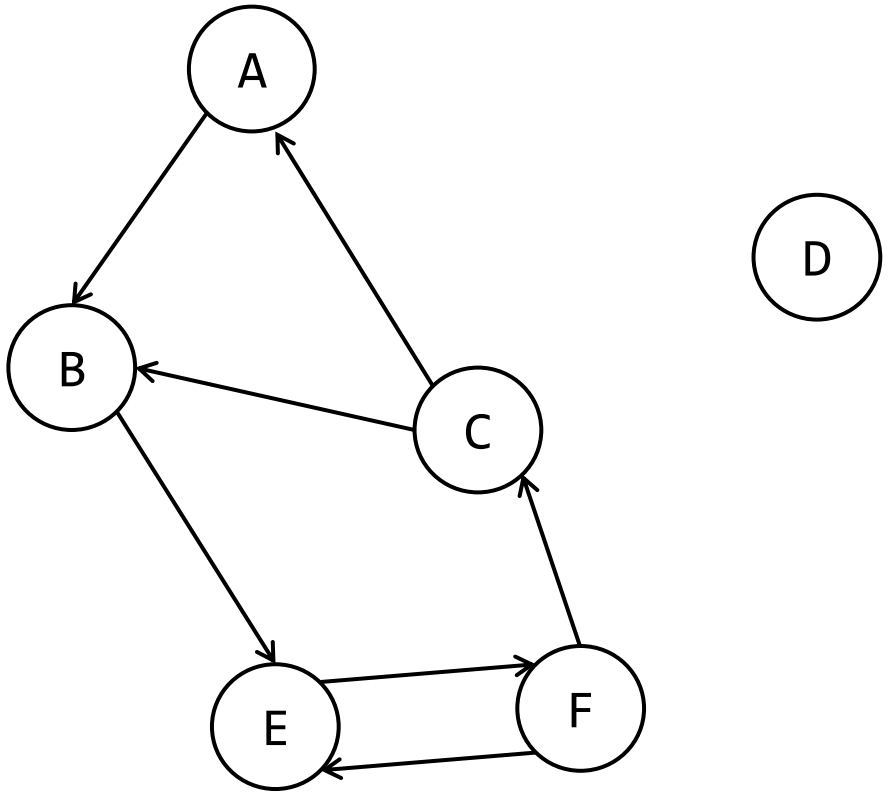
How many **nodes** does the graph have?

How many **edges?**

What are the **neighbors** of node F?

Do the edges have **weights?**

Are the edges **directed?**



Activity Answer

Nodes: 6

Edges: 7

Neighbors of F: C, E

Unweighted

Directed

Coding with Graphs

Represent Graphs in Python with Dictionaries

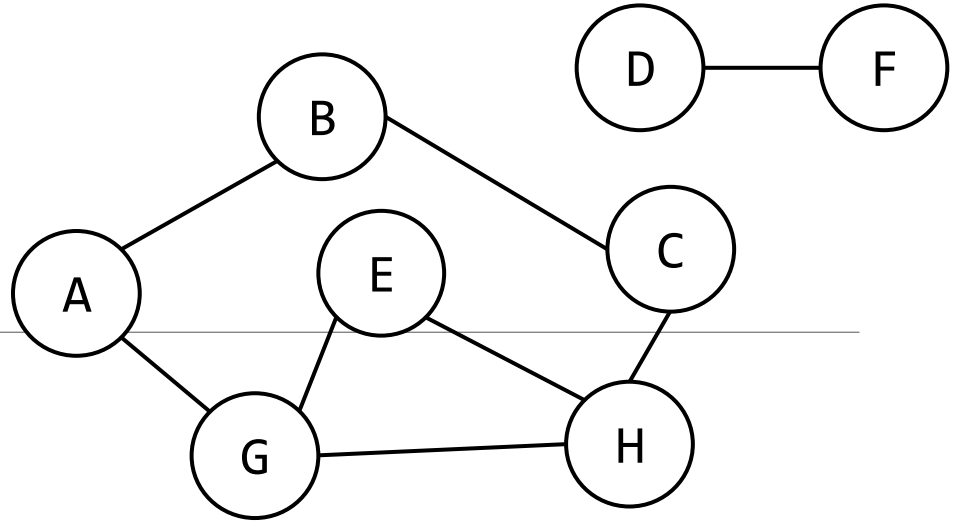
Like trees, graphs are not implemented directly by Python. We need to use the built-in data structures to represent them.

Our implementation for this class will use a **dictionary** that maps node values to lists. This is commonly called an **adjacency list**.

Unlike the tree representation, graphs will not be nested dictionaries; we'll be able to access all the node values directly. That's because graphs aren't inherently recursive.

We'll need to slightly alter this representation based on whether or not the edges of the graph have weights.

Graphs in Python – Unweighted Graphs



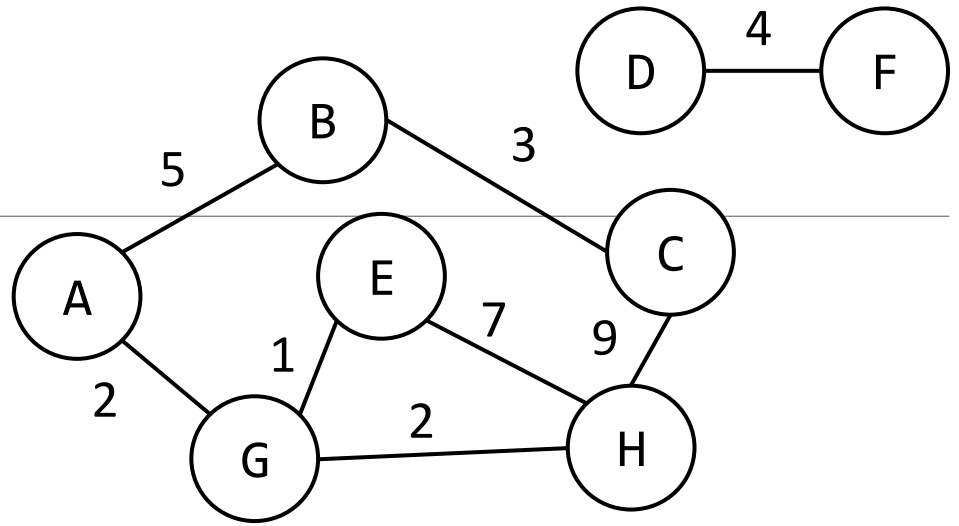
Graphs with no values on the edges are called **unweighted graphs**.

The keys of the dictionary will be the **values of the nodes**. Each node maps to a **list of its adjacent nodes (neighbors)**, the nodes it has a direct connection with.

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

```
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" ]
}
```

Graphs in Python – Weighted Graphs



Weighted graphs have values associated with the edges. We need to store these values in the dictionary also.

We'll do this by changing the list of adjacent nodes to be a 2D list. Each of the inner lists represents a node/edge pair, so it has two values – the adjacent node's value and the weight of the edge.

On the right, we show our updated example graph in this format.

```
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] ]
    }
```

Finding a Graph's Nodes

Let's look at some basic examples of programming with graphs.

To print all the nodes in a graph, just look at every key in the dictionary.

```
def printNodes(g):
    for node in g:
        print(node)
```

Finding a Node's Neighbors

If we want to get the neighbors of a particular node, index into that node in the dictionary.

```
def getNeighbors(g, node):
    return g[node]
```

If the graph has weights, we'll need to reconstruct the neighbor list:

```
def getNeighbors(g, node):
    neighbors = [ ]
    for pair in g[node]:
        neighbors.append(pair[0])
    return neighbors
```

Finding a Graph's Edges

To print all the edges, you'll need to loop over each **value** in the dictionary too (a list of nodes).

```
def printEdges(g):
    for node in g:
        for neighbor in g[node]:
            print(node + "-" + neighbor)
```

Note that this example is for an unweighted graph. To get neighbor values in a weighted graph, index into `neighbor[0]`.

Finding an Edge's Weight

Finally, to find an edge's weight, index and loop to find the appropriate pair.

```
def getEdgeWeight(g, node1, node2):
    for pair in g[node1]:
        if pair[0] == node2:
            return pair[1]
```

Example: Most Popular Person

Now that we have the basics, we can start problem solving.

Let's write a function that takes a social network as a graph and returns the person in the network who has the most friends.

This is just our typical find-largest-property algorithm applied to a graph.

```
def findMostPopular(g):
    biggestCount = 0
    mostPopular = None
    for person in g:
        if len(g[person]) > biggestCount:
            biggestCount = len(g[person])
            mostPopular = person
    return mostPopular
```

Example: Make Invite List

Now let's say a person wants to make more friends, so they're holding a party. They want to invite their own friends, but also anyone who is a friend of one of their friends.

Now we have to loop over each of the person's friends, to access that node's own list of friends.

```
def makeInviteList(g, person):
    # start with immediate friends
    invite = g[person] + [ ] # break alias
    for friend in g[person]:
        # find friends-of-friends
        for theirFriend in g[friend]:
            if theirFriend not in invite and \
               theirFriend != person:
                invite.append(theirFriend)
    return invite
```

Activity: `friendsInCommon(g, p1, p2)`

You do: Given an unweighted graph of a social network (like in the previous two examples) and two nodes (people) in the graph, return a list of the friends that those two people have in common.

For example, in the graph shown to the right, calling `friendsInCommon` on `"Jon"` and `"Jaime"` would return the list `[ "Tyrion" ]`.

Hint: start by looping over all the friends of the first person. Check whether any of them are also friends of the second person and add them to a result list if they are.

```
g = { "Jon" : [ "Arya", "Tyrion" ],
      "Tyrion" : [ "Jaime", "Pod", "Jon" ],
      "Arya" : [ "Jon" ],
      "Jaime" : [ "Tyrion", "Brienne" ],
      "Brienne" : [ "Jaime", "Pod" ],
      "Pod" : [ "Tyrion", "Brienne", "Jaime" ],
      "Ramsay" : [ ]
    }
```

Activity Answer

```
def friendsInCommon(g, p1, p2):
    common = []
    for friend in g[p1]:
        if friend in g[p2]:
            common.append(friend)
    return common
```

Learning Goals

Use **dictionaries** when writing and reading code that uses pairs of data

Use **binary trees** implemented with dictionaries when reading and writing code

Use **graphs** implemented as dictionaries when reading and writing simple algorithms in code