Advanced #3: Dictionaries, Trees, and Graphs

SAMS SENIOR CS TRACK

Learning Goals

Use **dictionaries** to represent mapping data with keys and values

Use **trees** to represent hierarchical data with parents and children

Use **graphs** to represent connected data with nodes and edges

Data Structures

There are many classic **data structures** in computer science. These are all different ways to describe how data can be stored and acted upon. We've already seen two examples of data structures: **strings** and **lists**.

The data structure you choose to store data changes how easily and efficiently you can perform certain actions. Identifying the correct data structure to use can make writing a program much easier.

We'll briefly go over three classic data structures: **dictionaries**, **trees**, and **graphs**.

Dictionaries

Abstract Dictionaries/Maps

A **dictionary** (or **map**) is a structure composed of two types of data: **keys** and **values**.

Dictionaries work in a similar way to lists, with one core difference: instead of mapping indexes to values, dictionaries map **specific keys** to values. This lets us associate the keys and values in a meaningful way.

Dictionaries exist in real life- think of an actual dictionary (which maps words to definitions), or a phone book (which maps names to phone numbers). We generally use them when we want to efficiently look up one kind of data based on a known value.

Dictionaries in Python

In Python, we use curly brackets to set up a dictionary:

```
d = { } # this is an empty dictionary, with no keys
```

To add a key/value pair to a dictionary, we use syntax similar to setting a value in a list index:

```
d[key] = value
```

Otherwise, dictionaries work very similarly to lists. We can access values with `d[key]`, check if a key is in a dictionary with `in`, and iterate over the keys of a dictionary with `for key in dict` .

Dictionary Example

With all three data types we learn in these slides, we'll try to solve the same problem. How can we find and return the largest item in the data structure?

```
# For dictionaries, we want to find and return the largest value (not key).
def findMax(d):
    best = None
    # Iterate through all the keys, then check the values.
    for key in d:
        # Set the first value we find to best automatically
        if best == None or d[key] > best:
            best = d[key]
    return best
```

Trees

Abstract Trees

A **tree** is a data structure that is composed of **nodes**. Nodes represent individual values (like the keys in a dictionary or values in a list).

In a tree, we organize the nodes **hierarchically**- nodes can have **parents** (which are connected directly above them in the tree) and **children** (which are connected directly below them). By organizing many nodes, we can create a many-layered tree.

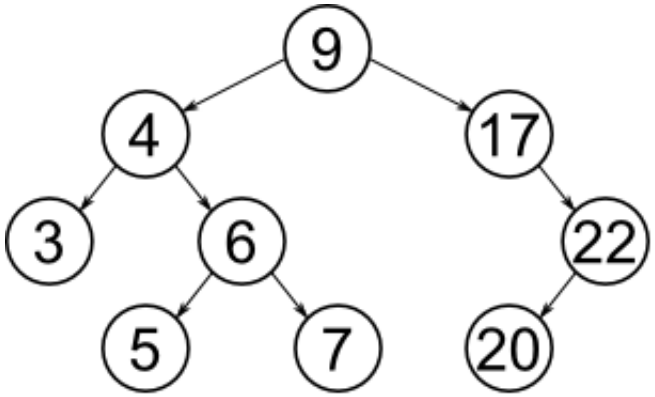
Trees (the data structure) exist in real life too. An obvious example is a family tree- each person has two parents, and 0+ children. Another example is the file system in your computer- folders can be contained in a folder, and can also hold their own folders.

Trees Don't have Cycles

We can't arrange nodes however we want in a tree- there are rules to follow!

First: the definition of a tree is that it **contains no cycles**. If a node A is the parent of a node B, then B cannot be a parent of A, nor can B have any children that are the parent of A.

This means that we'll always have at least one node that has no parents- the **root**. We'll also have at least one node that has no children- a **leaf**. But we draw our trees upside-down!



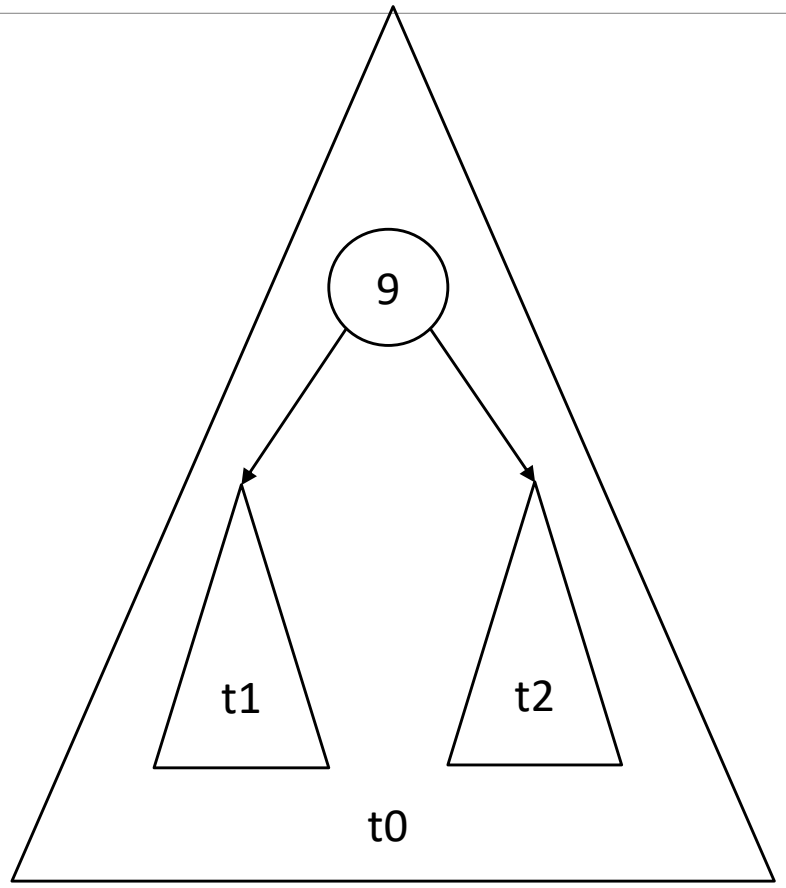
In the example above, 9 is the root, and it has 4 and 17 as children. 3, 5, 7, and 20 are all leaves- each has a parent, but no children.

Trees and Recursion

When we interact with trees, we need to consider them as a **recursive data structure**.

Instead of thinking of a tree as a bunch of connected nodes, think of it as a single node that has a value and a list of trees that are that node's children.

This is recursive because our definition of a tree contains a tree! But it will work due to our base case- the **base case** of a node with no children. In this case, no other trees are needed.



Trees in Python

Trees are not a built-in datatype in Python. However, we can create a tree easily using an **object**.

```
class Tree(object):
    def __init__(self, value, children):
        self.value = value
        self.children = children

t = Tree(9, [Tree(4, []), Tree(17, [])])
```

Alternatively, we could make a dictionary where the key is the root node and the value is a list of dictionaries (each a child tree). But we'll use the object approach here.

Tree Example

```
# Given a tree t, find and return the largest value.
def findMax(t):
    # Base case: no children. Return the current value instead.
    # We technically don't need to separate the base case, but we'll keep it for clarity.
    if len(t.children) == 0:
        return t.value
    else:
        best = t.value
        # Repeat the algorithm on each child to find the largest value among the children
        for child in t.children:
            tmp = findMax(child)
            if tmp > best:
                best = tmp
        return best
```

Graphs

Abstract Graphs

A **graph** is a data structure composed of **nodes** and **edges**. Nodes represent values; edges represent connections between nodes (and sometimes have values as well).

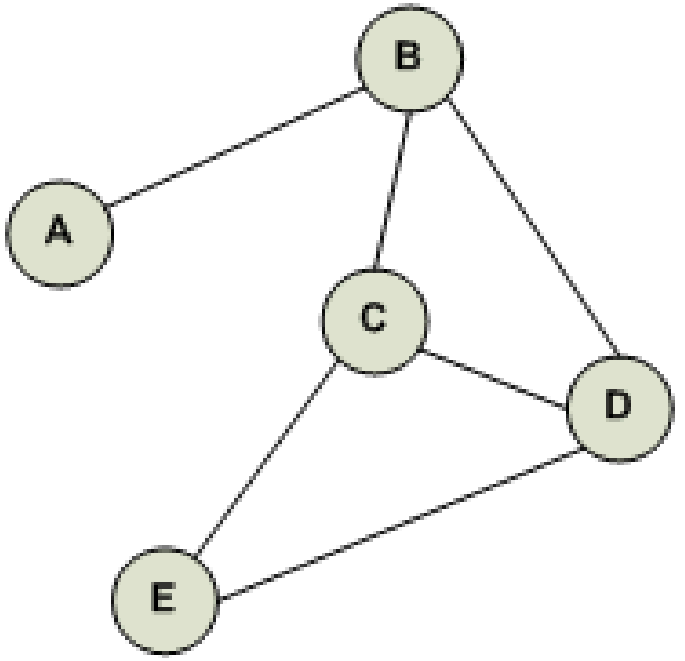
Unlike trees, graphs can have edges between any pair of nodes- in other words, **cycles** are allowed. Trees are technically a sub-category of graphs, though they are often treated as a separate data type entirely.

Graphs exist in many types of data in real life. Social networks are stored as graphs (where people are nodes and friendships are edges); scheduling travel by flight is also stored as a graph (cities/airports are nodes, scheduled flights between cities are edges). We use graphs when we have data points with multiple connections.

Interacting with Graphs

When investigating the nodes of a graph, we'll need to take repeated actions (as we did when investigating the nodes of a tree). However, **graphs have no endpoint**- if you're not careful, you can keep looping forever!

In general, we avoid this by keeping track of **visited nodes**, so we can avoid visiting the same node multiple times.



Graphs in Python

As with trees, there is no built-in data structure for graphs in Python. There are several ways we can try to set up a graph.

One approach is to use objects. Instead of representing nodes as having values and children, each node can have a value and a list of edges.

Another approach is to use a dictionary. Each node's value can be a key, and each key's value can be a list of nodes that node is connected to. This is useful when we don't need to associate edges with values, and when each node's value is unique. We'll use this format for now.

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

Graphs with Edges

The dictionary approach works well when only nodes have values. If your graph has edges with values as well, then additional information needs to be stored.

We can use an **adjacency matrix** to store this information. The N nodes are given indexes from 0 to N-1; we can then make a 2D list where each i,j position holds the value of the edge between nodes i and j, if it exists, or None if there is no edge.

In the example to the right, we use 0 to indicate same-node, 1 to indicate an edge, and None to indicate no edge.

```
nodes = ["A", "B", "C", "D", "E"]


matrix = [ #   A,    B,    C,    D,    E
           [    0,    1, None, None, None ], # A
           [    1,    0,    1,    1, None ], # B
           [ None,    1,    0,    1,    1 ], # C
           [ None,    1,    1,    0,    1 ], # D
           [ None, None,    1,    1,    0 ]  # E
         ]
```

Graph Example

```
# Using the dictionary format, find the largest value that is connected to the starting value 0.
def findMax(g):
    # There are two classic algorithms for searching a graph: depth-first and breadth-first.
    # Here, we demonstrate breadth-first.
    visited = [ 0 ]
    i = 0
    best = 0
    while i < len(visited):
        # Check if the current value is the best
        value = visited[i]
        if value > best:
            best = value
        # Go through the connected nodes, add any we haven't visited
        for node in g[value]:
            if node not in visited:
                visited.append(node)
        # Then check the next node until we visit them all
        i += 1
    return best
```

Learning Goals

Use **dictionaries** to represent mapping data with keys and values

Use **trees** to represent hierarchical data with parents and children

Use **graphs** to represent connected data with nodes and edges

Want to learn more? Read more on dictionaries here, and read more on trees and graphs here.

Beyond the basics of implementation, there are tons of algorithms for advanced data structures, especially graphs. If you're interested, read more here!

Bonus Task

Bonus Task

For this bonus task, you'll need to find a dataset, download it, and write a simple script that loads and stores that dataset into either a **dictionary**, **tree**, or **graph**, depending on which data type best fits the data you want to analyze. We'll go over how to load files on Thursday.

You should then write a simple function that takes the data structure you created and prints it out in some way, such that all of the data is visible.

You can find lots of interesting datasets at https://catalog.data.gov/dataset . Be careful- some of these datasets are huge!

Submit your answer to the bonus3 assignment on Autolab by **noon on Friday 7/19**. You don't need to upload the dataset itself- just put a comment with a link to it at the top of your file.