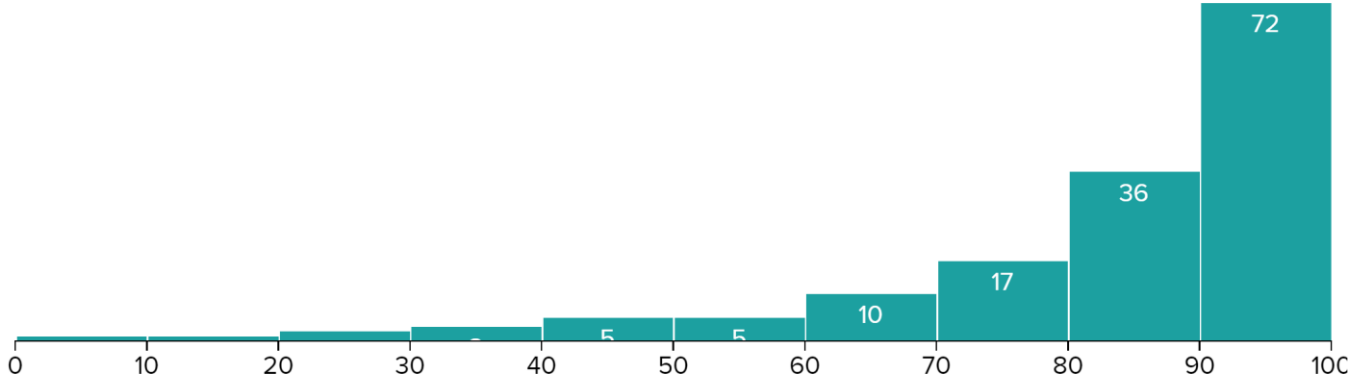
Unit 2 Review

15-110 – Friday 03/26

Announcements

- Hw4 due **Monday noon** (midsemester surveys due then as well!)
  - **Read the instructions carefully!** The programming questions are a little more advanced, but we provide a **lot** of advice for how to structure the algorithms.

- Quiz3 grades released
  - Median = 89 – well done!
  - There are an unusually high number of low scores. If you scored < 70, strongly consider talking to your TA or Prof. Kelly about course resources.



MINIMUM	MEDIAN	MAXIMUM	MEAN	STD DEV
8.0	89.0	100.0	82.34	18.53

Agenda

- Unit Overview
- Hashed Search
- Coding with Graphs
- Lecture 1:
  - Big-O
  - Coding with Recursion
- Lecture 2:
  - Breadth-First Search / Depth-First Search
  - List syntax

Unit 2 Overview

Unit 2 Goals

Our second unit had two major goals: use various **data structures** to organize data and calculate the **efficiency** of a variety of algorithms.

How did the topics we discussed fit into these themes?

Data Structures

We discussed several different ways to organize data in a data structure. All of these (except for strings) use **references** when copying data to let us change the values in the structure directly and to save space.

- **Strings** are a sequence of characters that we can access with indexing and slicing with "" syntax
- **Lists** store data sequentially and in multiple dimensions if needed with [] syntax
- **Dictionaries** store key-value pairs with {} syntax
- **Trees** store hierarchical data using **recursion**, implemented as a nested dictionary
- **Graphs** store connected data, implemented as a dictionary mapping nodes to lists of neighbors

Efficiency

We also discussed the **efficiency** of different algorithms. We abstracted away questions of computer power and specific implementation with the concept of **Big-O notation**, which represents an algorithm's function family.

We discussed a set of **search algorithms** over varying data structures. This included **linear search** for lists, **binary search** for lists and BSTs, **hashed search** for dictionaries, and **BFS/DFS** for graphs.

We also discussed how to improve efficiency with **hashed search** and with **merge sort**.

Finally, we discussed how the **tractability** of an algorithm influences its ability to run in 'reasonable' time on large sets of data.

Upcoming Topics

In the next unit, we'll talk about how to deal with inefficient algorithms by **scaling up** the amount of computing power used to run algorithms.

We'll then move on to a unit where we address how to apply computer science to **other domains**.

Hashed Search

Big Idea

Why do we care about hash functions?

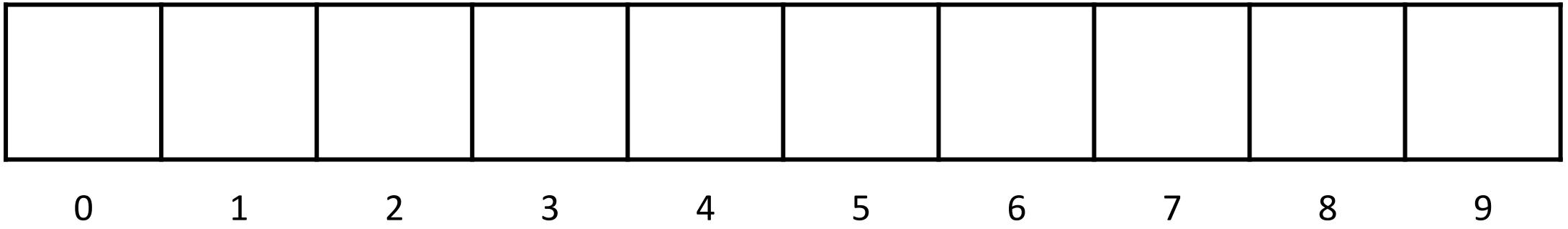
We search all the time, so we want the fastest possible search. Storing items in a hashtable lets us look up whether or not an item is in the table in **O(1)** time. You can't get faster than that!

How can we search in constant time? The algorithm needs to know **where** the value it's looking for will be stored **if** that value is actually in the table.

Hashtables

A **hashtable** is like a big, empty list of a designated size. Like in a list, each slot ('bucket') in the table is associated with an **integer index**, from 0 to len(table)-1.

When we want to put a value in the hashtable, we insert it at a specific index based on the result of a **hash function**.



Hash Functions

A **hash function** is a function that maps Python values to integers. Those integers can then be used to find an index in the hashtable to store the value.

We can use the built-in Python `hash()` function or write our own. Either way, the hash function must follow two rules:

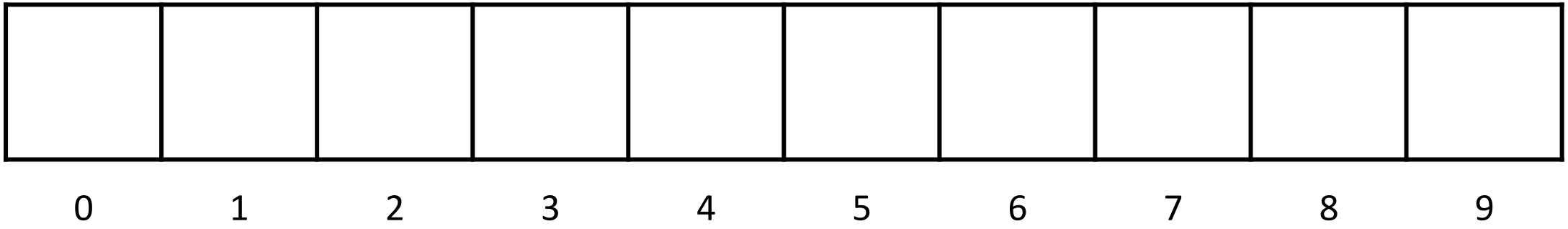
- The value returned when hash is called on x must not change across calls
- The function should usually return different numbers when called on different values

Storing/Finding Values in Hashtables

Both storing a value in a hashtable and checking whether a value is in a hashtable follow the same procedure to get the index to check.

1. Run the hash function on the value to get the hashed value.
2. If the hashed value is larger than the hashtable size, mod it by the hashtable size

Let's practice with some strings and the built-in hash function.



Why O(1)?

Why is looking up a value in a hashtable O(1) time?

We don't need to check every bucket in the hashtable. Only look in one bucket- the one with the index associated with the hashed value.

Important: this only works if the value we're searching for can't change (**immutable**) and if the hashtable is large enough for the stored values to spread out.

Coding With Graphs

Consider which nodes/edges you need to check

When coding with graphs, we often end up in one of two scenarios.

Scenario 1: we need to look at **all** the nodes and edges in the graph systematically

finding the most popular person, or the shortest edge

Scenario 2: we need to look at **some** of the nodes and edges of the graph, based on some node that is provided as a starting place

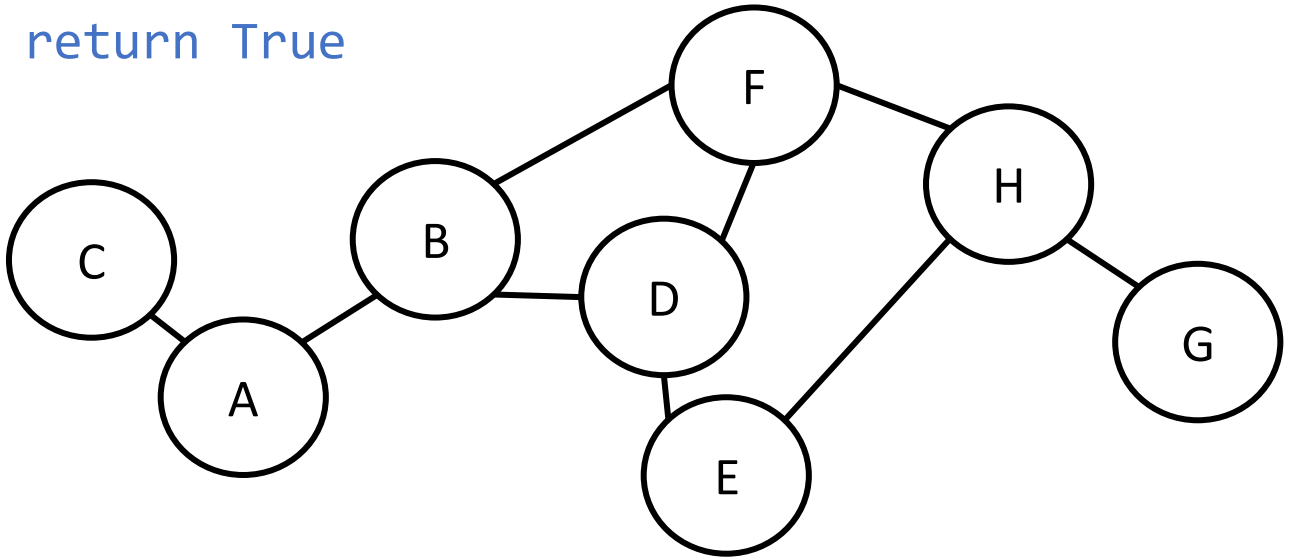
identifying nearby neighbors, or seeing if there's a path from A to B

Scenario 1: Fully-Connected Graphs

We've talked before about how the worst-case scenario for graphs is often a **fully-connected graph**, one where every pair of nodes is connected. How can we tell if a graph has this property?

This falls into Scenario 1. Loop over **every** node, then check if **every** other node is in that node's neighbors.

```
def fullyConnected(g):
    for node in g:
        neighbors = g[node]
        for anotherNode in g:
            if anotherNode != node and \
               anotherNode not in neighbors:
                return False
    return True
```

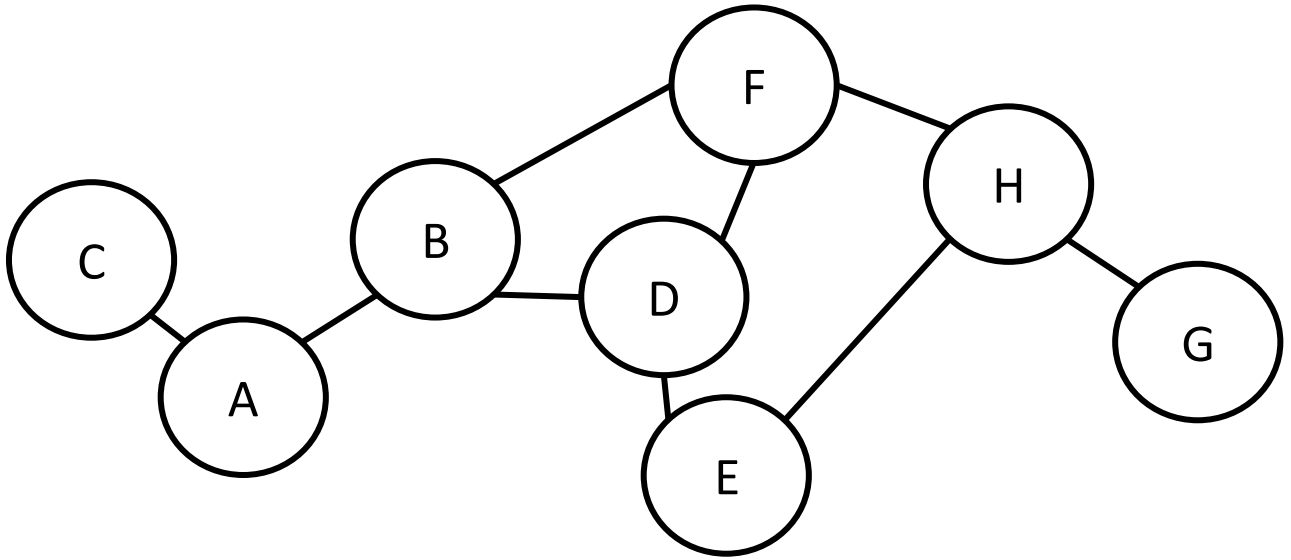


Scenario 2: Triads

We've shown how to tell whether two nodes are connected, but how can we tell if **three** nodes are all connected into each other? More specifically, if we're given a node, can we tell whether it's part of a connected triad?

This falls into Scenario 2 – we only need to look at a **subset** of the nodes and edges. Look at all pairs of the immediate neighbors of the given node and see if any are connected to each other.

```
def findTriad(g, node):
  for neighbor in g[node]:
    for otherNeighbor in g[node]:
      if neighbor != otherNeighbor and \
         otherNeighbor in g[neighbor]:
         return [node, neighbor, otherNeighbor]
  return None
```



Big-O

Lecture 1 only

Big-O Essentials: What to Count?

When measuring the Big-O complexity of an algorithm, we have to specify what it is we're counting. Some popular choices:

- comparisons: `target == lst[i]`
- assignments: `y[i+1] = x[i]`
- recursive calls: `recSearch(tree["left"], target)`
- all 'actions' in the program (all of the above, plus more)

Big-O Essentials: Find the Dominant Term

When calculating Big-O, we don't care about linear coefficients. An algorithms that makes 3n comparisons is considered just as fast as an algorithm that makes 2n comparisons: both are O(n).

Only the dominant term matters:

$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) < O(n!)$

Big-O Essentials: Mind the Exponent

$O(n^k)$ is polynomial in n and considered tractable, because k is **constant**

$O(k^n)$ is exponential in n and considered "slow" (intractable) because n is **variable**

Big-O Essentials: Look for a Pattern

Any algorithm that processes each element once is O(n).

- Add up the elements of a list
- Sum the numbers from 1 to N
- See if a list contains an odd number
- Find the index of the first even number

Big-O Essentials: When is an algorithm $O(n^2)$?

Doing an O(n) operation on every element of a list means the total operations is $O(n^2)$.

Common example: nested for loops that are both O(n):

```
for i in range(len(lst)):
    for j in range(i+1, len(lst)):
        if lst[i] == lst[j]:
            print(lst[i], "is duplicated")
```

When is an algorithm $O(n^2)$?

An algorithm can be $O(n^2)$ even if it has just one loop!

```
for i in range(len(lst)):
    if lst[i] in lst[i+1:]:
        print(lst[i], "is duplicated")
```

The `in` test is itself O(n) and its inside a for loop that does n iterations, so the algorithm is $O(n^2)$.

When is an algorithm O(log n)?

If we cut the problem size in half each time and only consider one of the halves, we can make $\log_2(n)$ such cuts, so the algorithm is O(log n).

For example, binary search cuts the list in half each time, so its O(log n).

Suppose we want the first digit of a long number:

```
while n > 9:
    n = n // 10
```

This code makes $\log_{10}(n)$ divisions, so it's O(log n).

When is an algorithm $O(2^n)$?

If we have a recursive algorithm operating on an input of size n and each call makes two recursive calls of size n-1, then the algorithm is $O(2^n)$. The number of calls **doubles** every time we increase the size by 1.

```
def abCombos(n, s):
    if n == 0:
        print(s)
    else:
        abCombos(n-1, s + "a") # first recursive call
        abCombos(n-1, s + "b") # second recursive call
```

When is an algorithm $O(2^n)$?

If we have a recursive algorithm and each call produces a result twice as long as the previous result, then the algorithm is $O(2^n)$.

```
def allSubsets(lst):
    if lst == [ ]:
        return [ lst ]
    else:
        result = [ ]
        subsets = allSubsets(lst[1:])
        for s in subsets:
            result.append(s)
            result.append([ lst[0] ] + s)
        return result
```

Coding with Recursion

Lecture 1 only

Big Idea

The core idea of recursion is that we can solve problems by **delegating** most of the work instead of solving it immediately.

This works because we make the input to the problem **slightly smaller** every time the function is called. That means it will eventually hit a **base case**, where the answer is known right away.

Once the base case returns a value, all the recursive calls can start returning their own values up the **call stack** until they reach the initial call.

Writing Recursive Code: findNext

When working with recursive code, it often helps to think **abstractly** about how to solve the problem with delegation before jumping into coding.

For example: what if we wanted to modify our search approach to find the item that occurs **after** the item we're looking for?

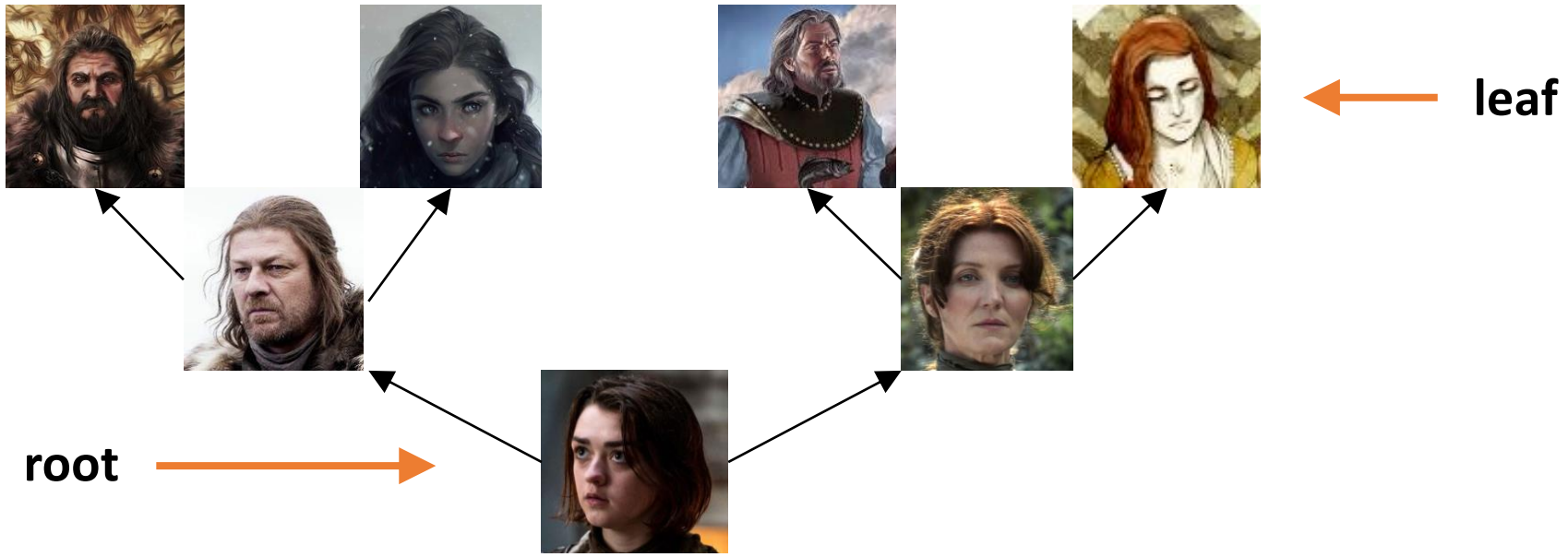
We want to start with an approach similar to our regular search – have base cases for when we find the item, and when we don't. And recurse by checking the rest of the list.

```
def findNext(lst, target):
    if len(lst) <= 1:
        return None
    elif lst[0] == target:
        return lst[1]
    else:
        return findNext(lst[1:], target)
```

Recursion with 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
```

Breadth-First Search / Depth-First Search

Lecture 2 only

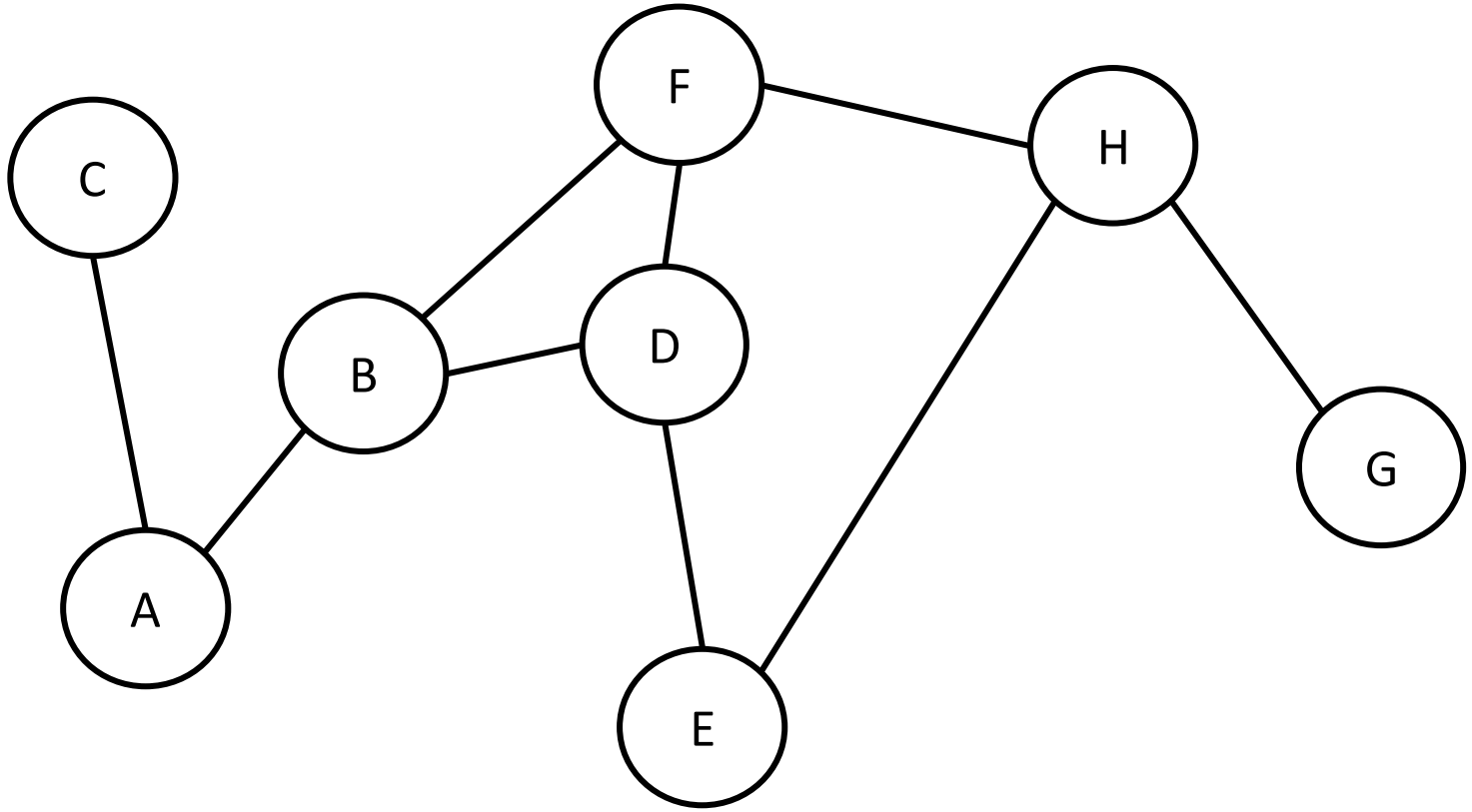
Breadth-First vs Depth-First Search

Both BFS (Breadth-First Search) and DFS (Depth-First Search) share a common goal: they need to find if there's a **path** between a given start node and a given target node in a graph.

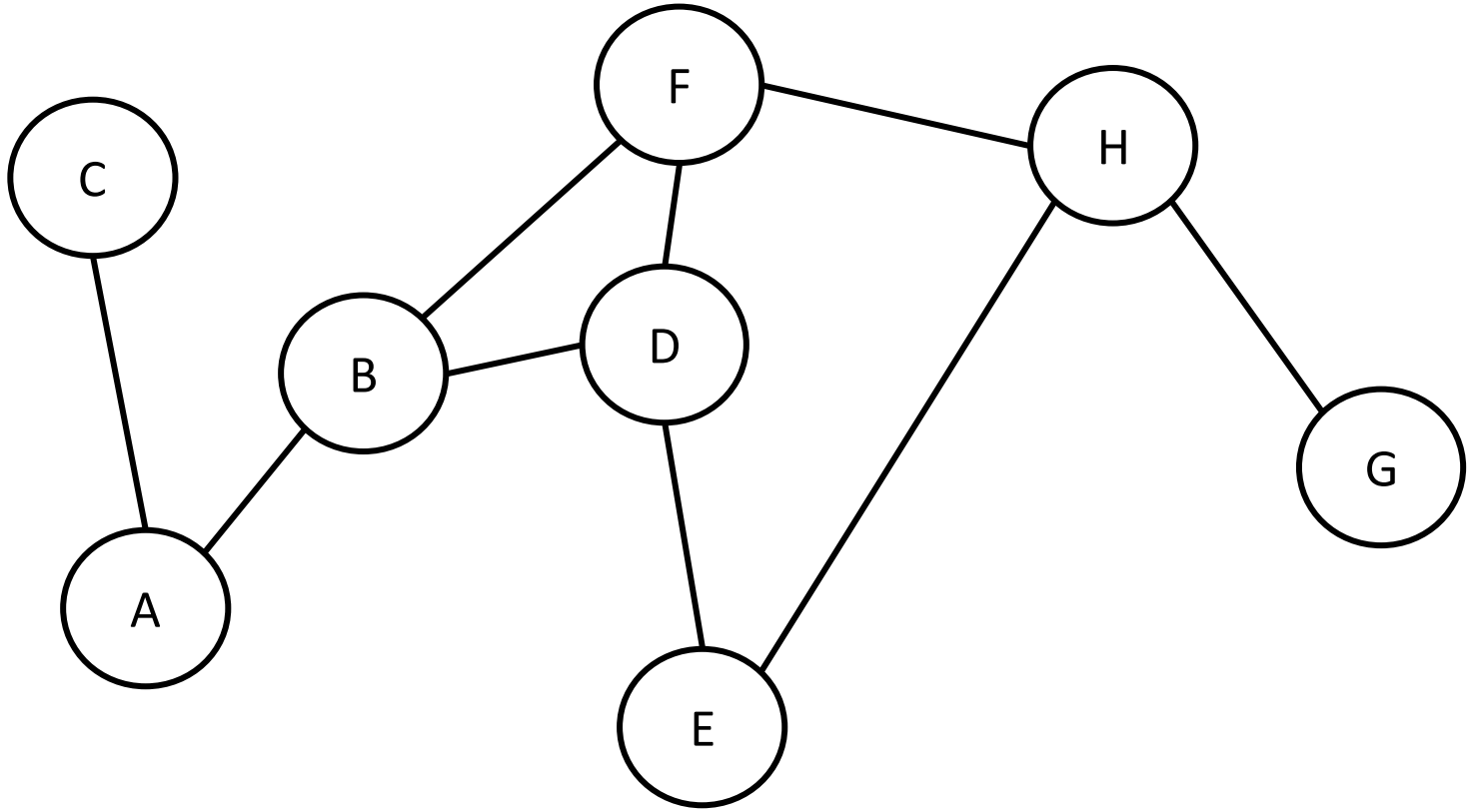
In **BFS**, you start with the nodes connected to the start node and slowly move outwards. It's like how might search for a tiny fallen item- start close to where you're standing, then move outwards.

In **DFS**, you go outwards rapidly, backtracking when necessary. It's like how you solve a hedge maze- go all the way down one path, then go back when it doesn't work.

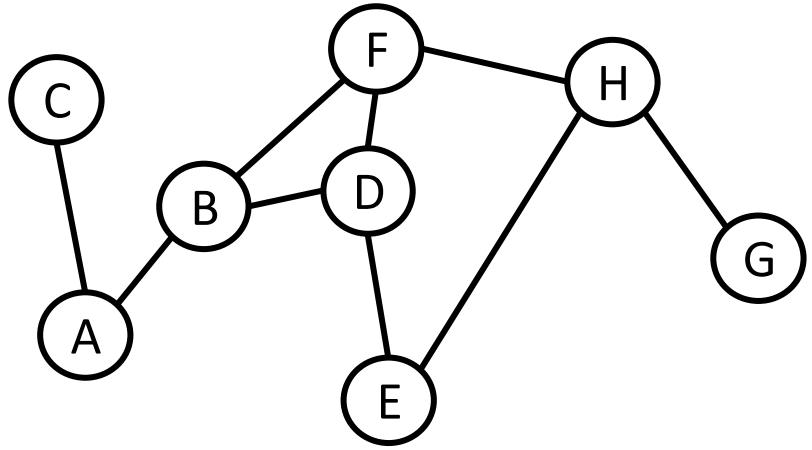
BFS Example – Start from A



DFS Example – Start from A



Tracing Graph Search Code

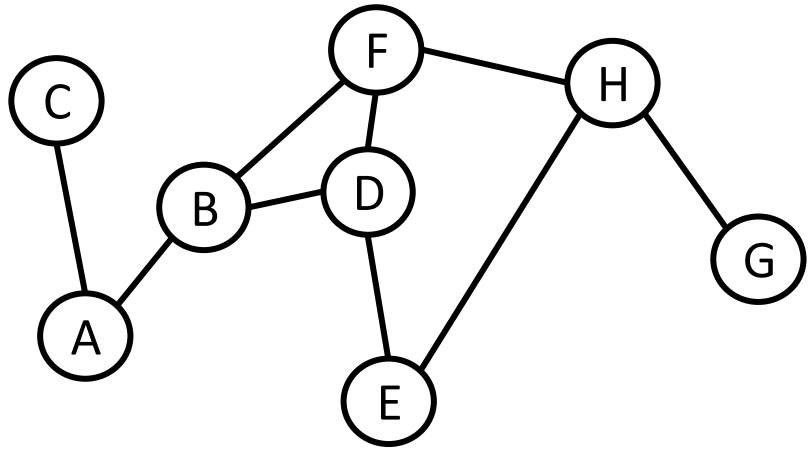



```
def breadthFirstSearch(g, start, target):
    visited = [ ]
    nextNodes = [ start ]

    while len(nextNodes) > 0:
        next = nextNodes[0]

        if next == target:
            return True
        else:
            for node in g[next]:
                if node not in visited and \
                   node not in nextNodes:
                    nextNodes = nextNodes + [ node ]

        nextNodes.remove(next)
        visited.append(next)

    return False
```

visited:

nextNodes:

Tracing Graph Search Code




```
def depthFirstSearch(g, start, target):
    visited = [ ]
    nextNodes = [ start ]

    while len(nextNodes) > 0:
        next = nextNodes[0]

        if next == target:
            return True
        else:
            for node in g[next]:
                if node in nextNodes:
                    nextNodes.remove(node)
                if node not in visited and \
                   node not in nextNodes:
                    nextNodes = [ node ] + nextNodes

        nextNodes.remove(next)
        visited.append(next)

    return False
```

visited:

nextNodes:

List Syntax

Lecture 2 only

Constructing lists

We can construct a **1D list** using square brackets with commas separating values. For example:

```
[ 2, 4, 8 ]
```

To construct a **2D** (two-dimensional) list, just put lists inside of lists!

```
[ [ 1 ], [ 2, 3 ], [ 4, 5, 6 ] ]
```

Adding elements to lists

We usually add elements to lists using either a **method** or **concatenation**.

When we use `lst.append(x)`, `x` is added as an **element inside the list**.

```
lst = [ 1, 2, 3 ]
lst.append(4) # lst = [ 1, 2, 3, 4 ]
```

When we use `lst = lst + x`, the **elements inside `x`** are added to the list. `x` must be a list itself.

```
lst = [ 1, 2, 3 ]
lst = lst + [ 4, 5 ] # lst = [ 1, 2, 3, 4, 5 ]
```

What happens if we append a list? **We get a 2D list!**

```
lst = [ 1, 2, 3 ]
lst.append([ 4, 5 ]) # lst = [ 1, 2, 3, [ 4, 5 ] ]
```

Loop by Index vs Loop by Element

When we want to **loop** over everything in the list, we can use one of two approaches.

Loop by index generates every index in the list. We can then use indexing to get the value at that location.

```
for i in range(len(lst)):
    item = lst[i]
    print(item)
```

Loop by element generates every element in the list directly. We do less work, but have no information about the element's location or what is around it.

```
for item in lst:
    print(item)
```

Looping over 2D lists

We can also use either loop approach to loop over a 2D list. Again, choose your loop based on whether or not you need information about the item's location.

```
for i in range(len(lst)):
    innerList = lst[i]
    for j in range(len(innerList)):
        item = innerList[j] # can also do lst[i][j] directly
        print(item)

for innerList in lst:
    for item in innerList:
        print(item)
```

Agenda

- Unit Overview
- Hashed Search
- Coding with Graphs
- Lecture 1:
  - Big-O
  - Coding with Recursion
- Lecture 2:
  - Breadth-First Search / Depth-First Search
  - List syntax

- Feedback: http://bit.ly/110-s21-feedback