

Search Algorithms II

Learning Objectives

- Identify whether a tree is a binary search tree
- Search for values in BSTs using binary search
- Analyze the efficiency of binary search on a balanced vs. unbalanced BST
- Search for paths in graphs using breadth-first search and depth-first search
- Analyze the efficiency of BFS and DFS on a graph

Binary Search Trees

Revisiting Search Algorithms

Recall the first lecture on Search Algorithms, when we discussed linear and binary search.

We've applied these algorithms to lists; can we apply them to other data structures too? Let's investigate how to search a tree.

Linear Search on a Tree

In linear search, we stepped through each element in a list until we either found the target item or ran out of items to look at. Trees aren't sequential, so how do we 'step through' a tree?

For every node in the tree, we need to check if that node is the target, then check whether the target is in one of the node's child subtrees. If we find the target in either subtree, we should return **True**.

We also have two base cases: one for when we reach an empty tree, and one for when we find the item. In both cases, we know what to return right away.

```
def search(t, target):  
    if t == None:  
        return False  
    elif t["contents"] == target:  
        return True  
    else:  
        return search(t["left"], target)  
    or \ search(t["right"], target)
```

Binary Search on a Tree

How would we apply Binary Search to a tree?

First, recall that for binary search to work, the input list must be sorted. We'll also need to find a way to "split" the tree similarly to how we split the list in binary search.

You do: how could you "sort" a tree?

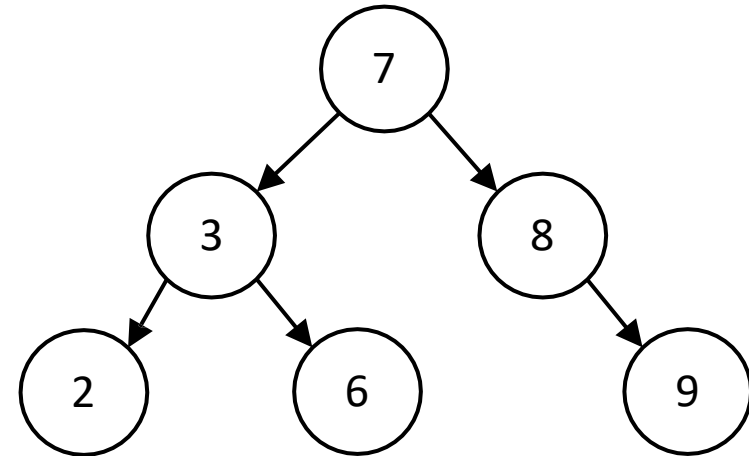
Binary Search Trees (BSTs) are "sorted"

We'll define a new kind of tree, a Binary Search Tree, as a binary tree that follows these constraints:

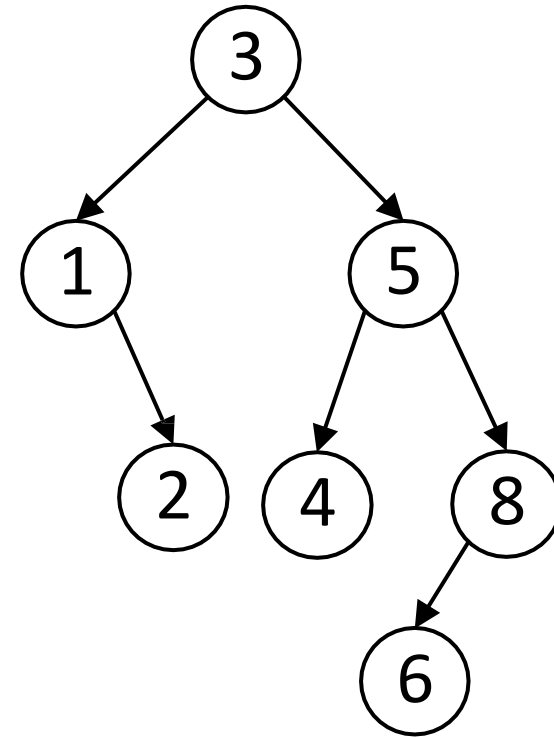
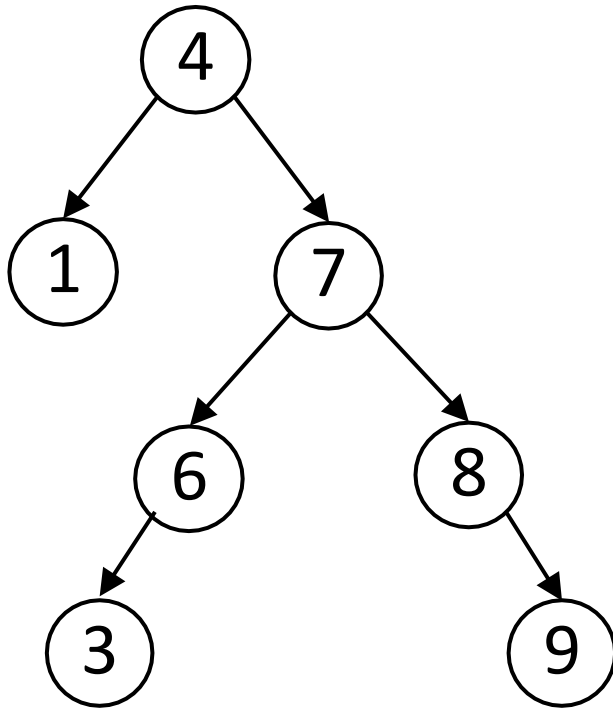
For every node n in a tree which has a value v :

- Each left child (and all its children, etc.) must be strictly less than v
- Each right child (and all its children, etc.) must be strictly greater than v

Note: the left and right subtrees are BSTs! BST constraints are recursive.



Example: Is this a BST?

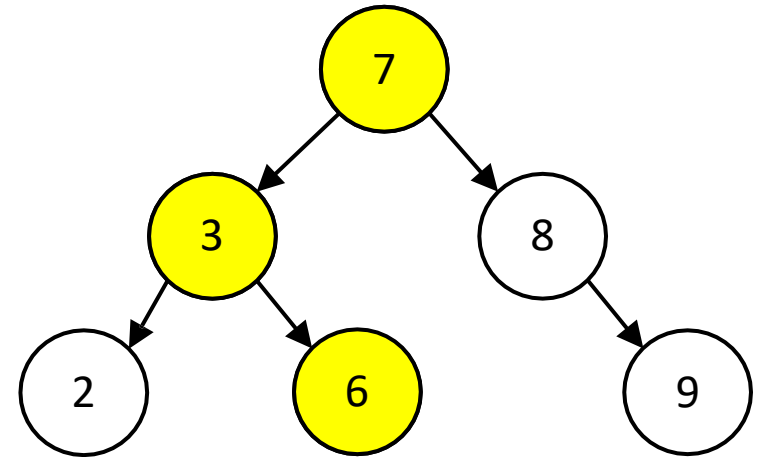


Binary Search Trees Can Use Binary Search

When we want to search for the value 5 in the tree to the left, we start at the root node, 7. Because all nodes less than 7 must be in the left child tree and 5 is less than 7, we only need to search the left child tree.

Then, when we compare 5 to 3, we know that all values greater than 3 (but less than 7) must be in the right child of 3. 5 is greater than 3, so we only need to search the right child.

We 'split' the tree by only looking at one of the node's two children. This is binary search!



BST Search in Python

We would write binary search for a BST as follows:

```
def search(t, target):  
    if t == None:  
        return False  
    elif t["contents"] == target:  
        return True  
    elif target < t["contents"]:  
        return search(t["left"], target)  
    else:  
        return search(t["right"], target)
```

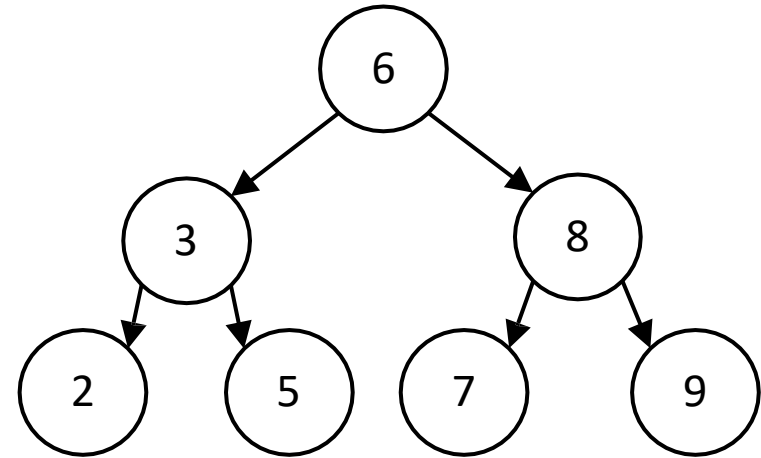
Note that we do just one recursive call, either on the left subtree or on the right subtree.

BST Search Runtime – Balanced Trees

Do we get the same $O(\log n)$ runtime for BST binary search that we did for list binary search? It depends. Let's first consider the runtime of search on a BST that is balanced. We'll analyze the runtime based on the number of nodes visited.

A tree is balanced if for every node in the tree, the node's left and right subtrees are approximately the same size. This results in a tree that minimizes the number of recursive levels.

Every time you take a search step in a balanced tree, you cut the number of nodes to be searched in half. This means that the algorithm will indeed take $O(\log n)$ time.

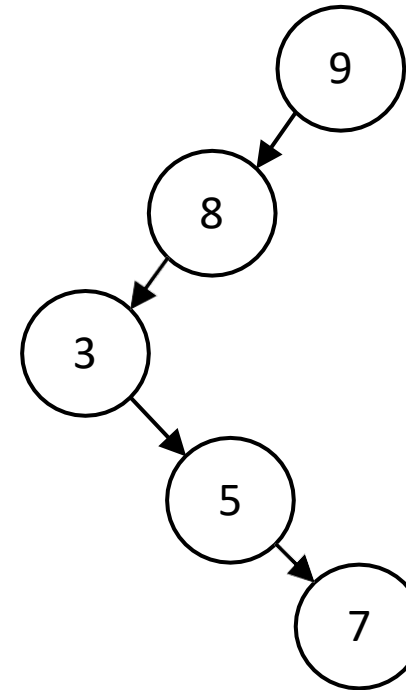


BST Search Runtime – Unbalanced Trees

A tree is considered unbalanced if at least one node has significantly different sizes in its left and right children. For example, consider the tree on the right.

This is a valid BST, but it is still difficult to search! You must visit every single node to determine a number like 6 isn't in the tree. In the worst case, this can still take $O(n)$ time.

When we put data into BSTs, we usually strive to make them balanced to avoid these edge cases. For efficiency purposes, assume our BSTs are balanced and the worst case is $O(\log n)$.



Benefits of BSTs

At first glance, BSTs may seem less useful than sorted lists. However, they have a few added perks!

BSTs make it much easier to add new data to a dataset. In a sorted list, you would need to slide a bunch of values over to make room for a new value; in a BST, you can just run a search for this new value. When you reach a leaf, add a node with the new value.

In general, try to choose a data structure that matches the task you need to solve.

Breadth-First Search and Depth-First Search

Searching a Graph for a Node

Determining whether a node exists in a graph is really easy given our dictionary structure: just check whether the node exists in the graph's keys.

```
def search(g, node):  
    return node in g
```

Finding a node is too easy. Let's ask a more interesting question.

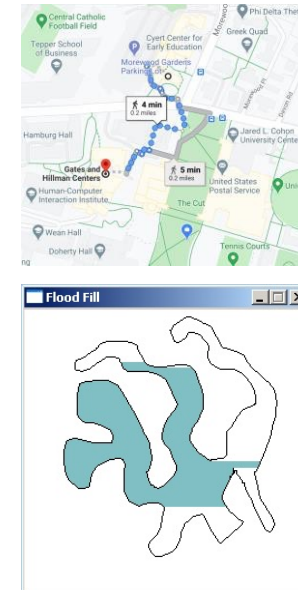
Searching a Graph for a Path

We now know about several data types that connect the data points in the structure (trees and graphs). We can search these structures to see if they contain a specific point of data, but it may be more interesting to see whether there is a connection between two nodes in the structure.

In other words: can we find a path that leads from a start node to a target node in the tree/graph?

This is useful in several contexts, including:

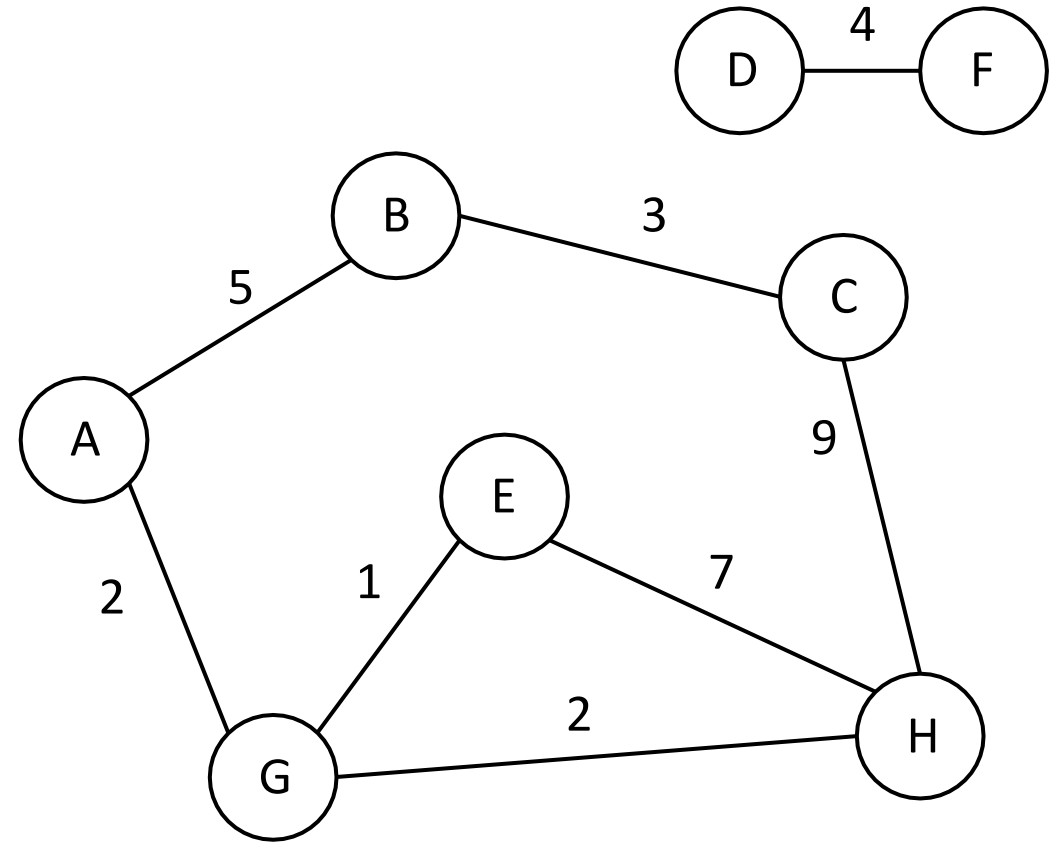
- finding walking/driving routes through a city
- contact tracing to identify people who are at risk
- flood fill in a paint application



How to Search for Paths?

How would you systematically search the graph shown here to see if there's a path between A and C?

Would the same algorithm successfully determine if there's a path between A and D?



Two Search Algorithms: BFS and DFS

We'll need to start at the start node and repeatedly follow the edges to find all the other nodes it's connected to. Here are two common approaches for deciding in which order to visit the connected nodes.

In Breadth-First Search (BFS), we slowly move outwards in the graph/tree from the start node. We visit all the neighbors of start, then visit all the neighbors of the already visited nodes, etc., until we've checked all the nodes that were connected to the start node of the graph/tree. BFS is useful for finding targets that should be nearby (like directions to a location within a 5 mile radius).

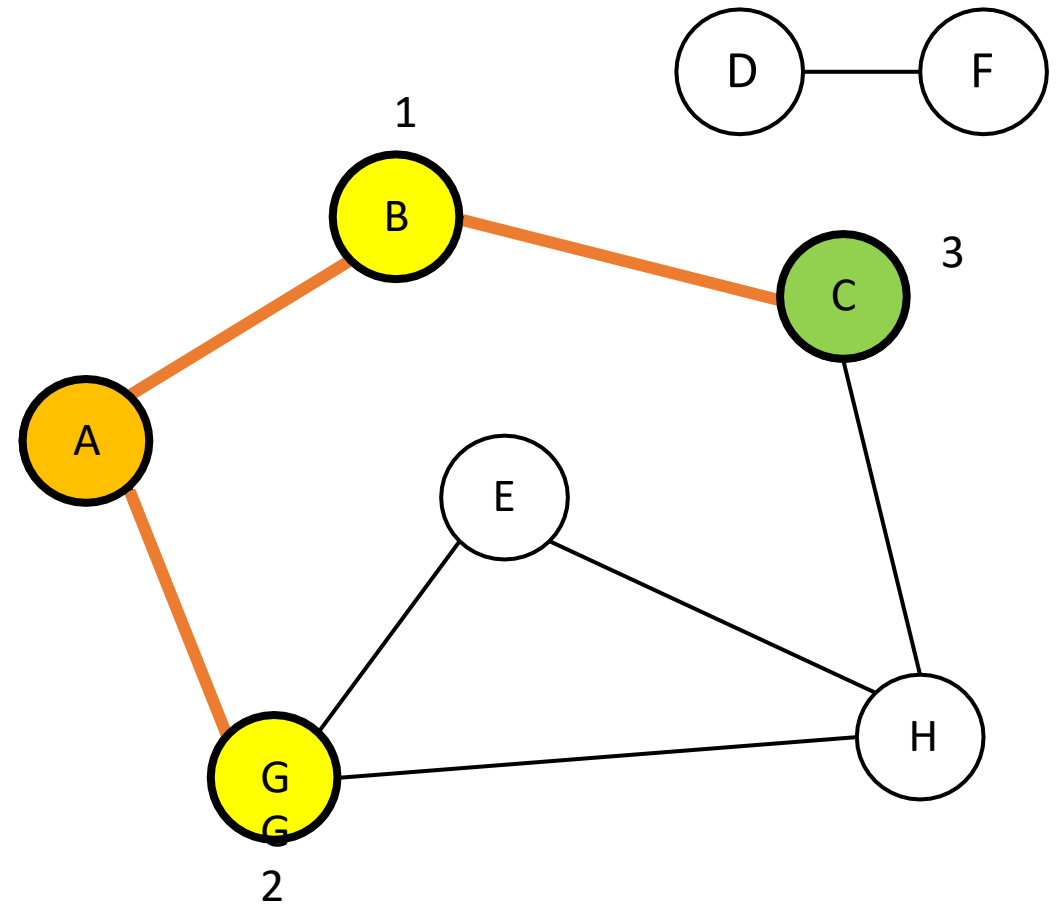
In Depth-First Search (DFS), we go all the way down one potential path, then backtrack and try other possible paths. So we choose one neighbor, then choose one of its neighbors, etc., until there are no unvisited neighbors left. DFS is useful for finding targets that are far away in a specific direction (like a route that goes straight across the United States east-to-west).

Breadth-First Search Example

Let's consider Breadth-First Search on our example graph, starting from A and searching for C.

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. (A is a neighbor as well, but we don't visit it because it's been visited before.) As soon as we reach C, we've found the node, and we're done- a path exists!



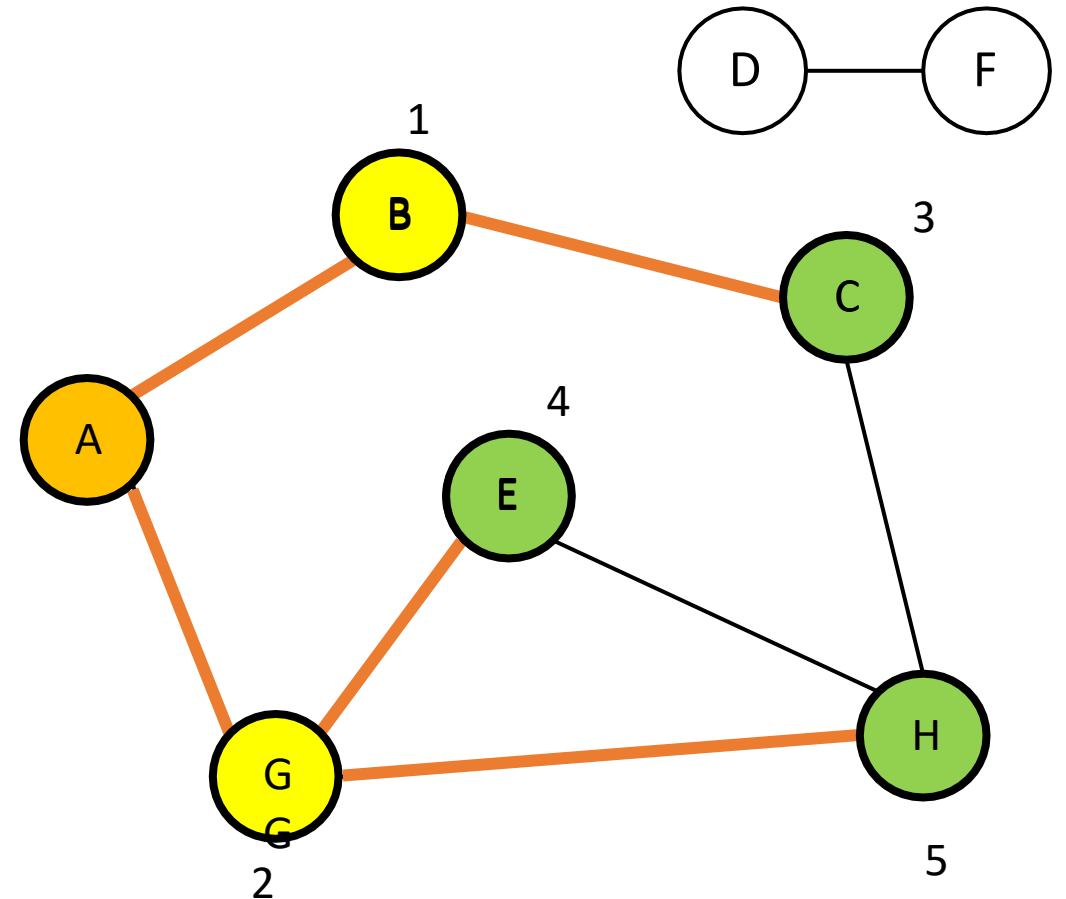
Breadth-First Search Example

Now let's run Breadth-First Search starting from A and searching for a value not connected to it, D.

A has two neighbors – B and G. As before, 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). We don't revisit A.

At this point, there are no nodes left that are neighbors of C, E, and H and have not been visited. We conclude there is no path from A to D.

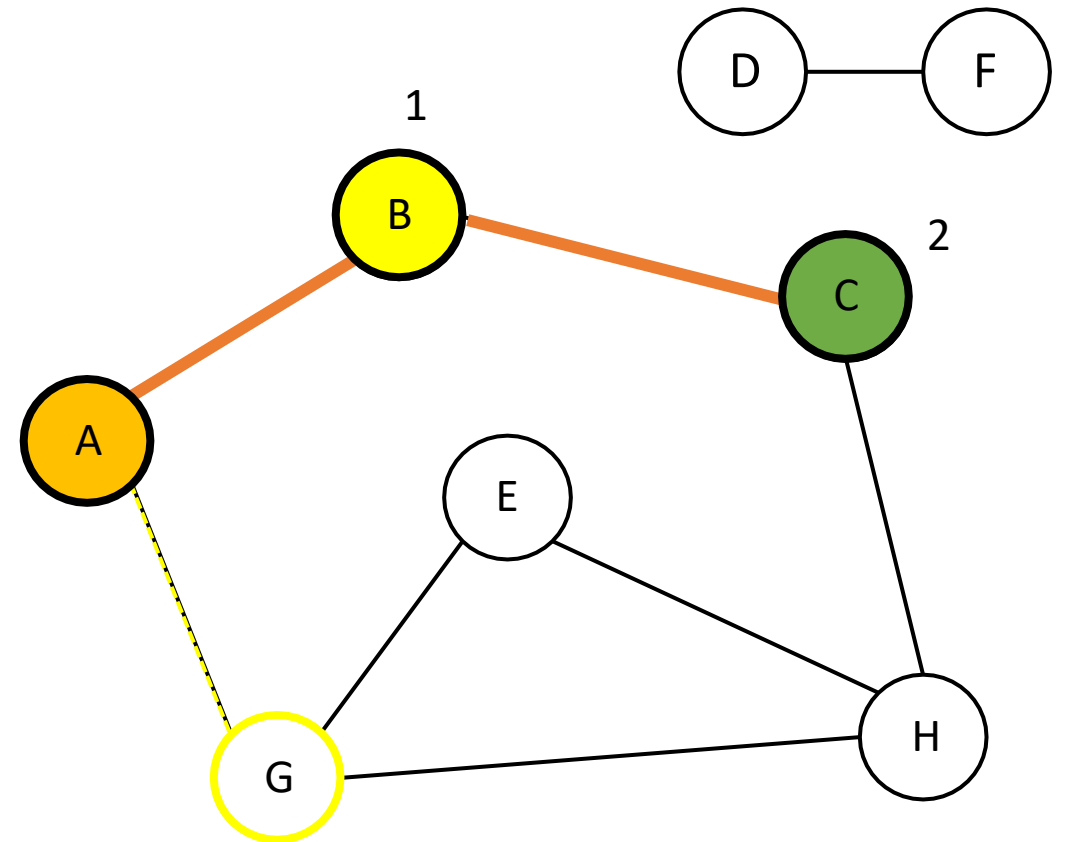


Depth-First Search Example

Now let's search the example graph starting from A with depth-first search, searching for C.

There are two possible starting routes: B or G. Let's choose B. We'll store G as a backup option in case we run into a dead end.

From B, we only have one unvisited neighbor: C. We've found the node we're looking for, so we're done!



Depth-First Search Example

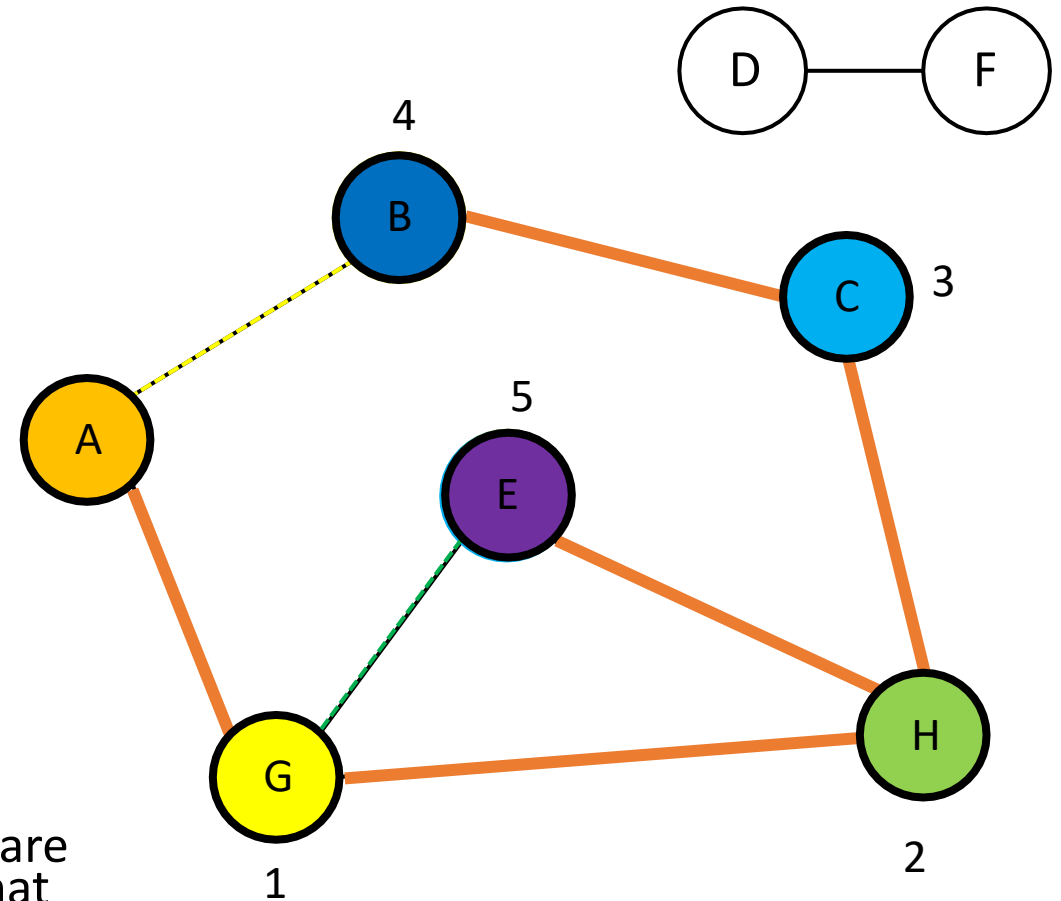
What if we search the example graph starting from A with depth-first search, now looking for D?

There are two possible starting routes: B or G. Choose G, and place B in the backup list.

From G, we have two possible routes, E or H; choose H and mark E as backup. Note that A is not a valid choice, as it's already been visited.

From H, we have two more possible routes: E or C (G is not valid). We'll choose C and put E on the backup list again. C's only remaining neighbor is B (H is not valid), so we must visit it.

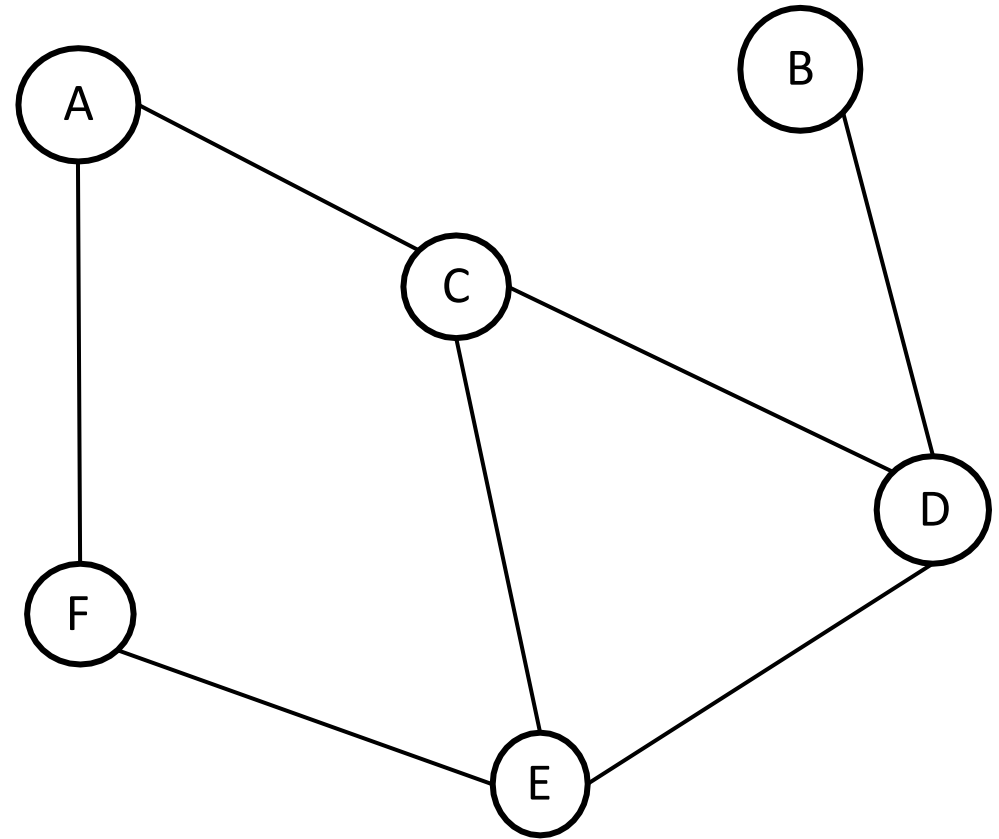
Now B has no unvisited neighbors remaining (A and C are both visited), so we must backtrack to the last node that had an unvisited neighbor. If we check our backup list, the only unvisited node remaining is E (which was G and H's neighbor). We visit E, and we're done. There is no path from A to D.



Activity: BFS and DFS Tracing

Given the graph to the right and starting from A while searching for G, what is a valid trace for Breadth-First Search, and what is a valid trace for Depth-First Search?

Visit neighbors alphabetically
(while following the search rules)
to make things simpler.



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 and update the list as we go. Each iteration will check the next node on the to-visit list to see if it's the end node we're looking for.

If we find the end node, we'll return `True` right away (a path is possible; making that path is out of scope for this class). If we're on a different node, we'll add all of its unvisited 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/tree.

```
def breadthFirstSearch(g, start, target):  
    # 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]  
  
        if next == target: # If it's what we're looking for- we're done!  
            return True  
        else: # Otherwise, add unvisited neighbors  
            for node in g[next]:  
                if node not in visited and node not in nextNodes: # Not seen before  
                    nextNodes = nextNodes + [ node ] # Add to the BACK of the list  
  
            nextNodes.remove(next)  
            visited.append(next) # We've visited the node now  
    return False
```

Depth-First Search Code

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

```
def depthFirstSearch(g, start, target):  
    # 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]  
  
        if next == target: # If it's what we're looking for- we're done!  
            return True  
        else: # Otherwise, add unvisited neighbors  
            for node in g[next]:  
                if node in nextNodes: # seen before, but higher-priority now  
                    nextNodes.remove(node)  
                if node not in visited and node not in nextNodes: # Not seen before  
                    nextNodes = [ node ] + nextNodes # Add to the FRONT of the list  
            nextNodes.remove(next)  
            visited.append(next) # We've visited the node now  
    return False
```

BFS / DFS Efficiency

If we only consider the worst case scenario and measure efficiency based on the number of nodes visited, BFS and DFS look the same.

The worst case is when there is no path between the start and target node, but every other node in the graph is connected to the start. In either algorithm we would need to visit every single node to determine that there is no path. The overall efficiency is $O(n)$.

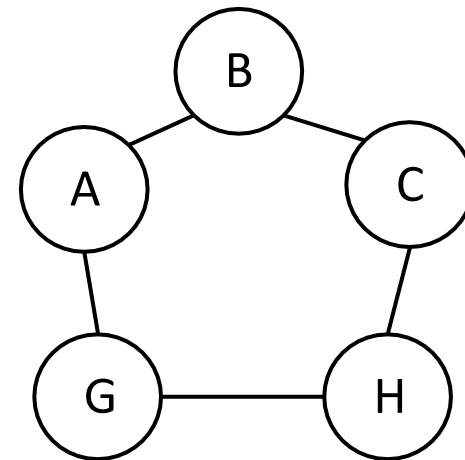
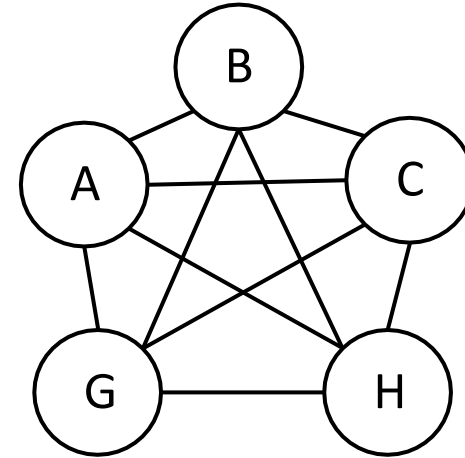
... or is it?

Same Nodes, Different Edges

It's possible for us to visit a node more than once in BFS or DFS if it appears multiple times in neighbor lists. For example, consider the two graphs to the right. Each has the same number of nodes.

The upper graph is heavily connected – every node is connected to every other node. For each node we visit we must verify that every other node in the graph is already in the visited or to-visit list.

In the lower graph we only need to deal with two neighbors per node. There are fewer repeats.



Edges Affect Efficiency

The efficiency of BFS/DFS is affected by the number of edges in the graph, just like how BST search was affected by how balanced the tree's edges were! When it comes to determining efficiency, edges matter.

As with BSTs, we'll simplify things for this class. We'll assume that the number of edges each node has is constant. Then the runtime for BFS / DFS can still be $O(n)$.