# Search Algorithms II

15-110 – Wednesday 10/21

## Learning Objectives

- Identify whether or not a tree is a **binary search tree**
- Search for values in **binary search trees** using **binary search**
- Understand how and why **hashing** makes it possible to search for values in **O(1) time**
- Search for values in a **hashtable** using a specific **hash function**
- Search for values in **graphs** using **breadth-first search** and **depth-first search**

## 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 children. If we find the target in **either** node, 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.

```python
def search(t, target):
  if t == None:
    return False
  elif t["value"] == 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 then 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?

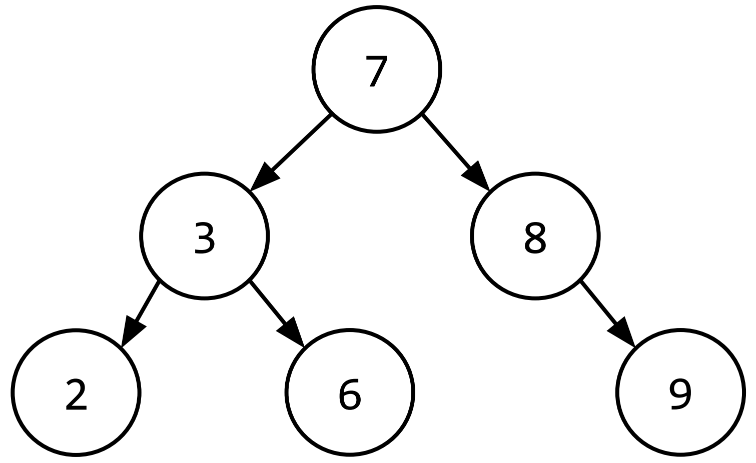
# Binary Search Trees (BSTs) are "sorted"

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

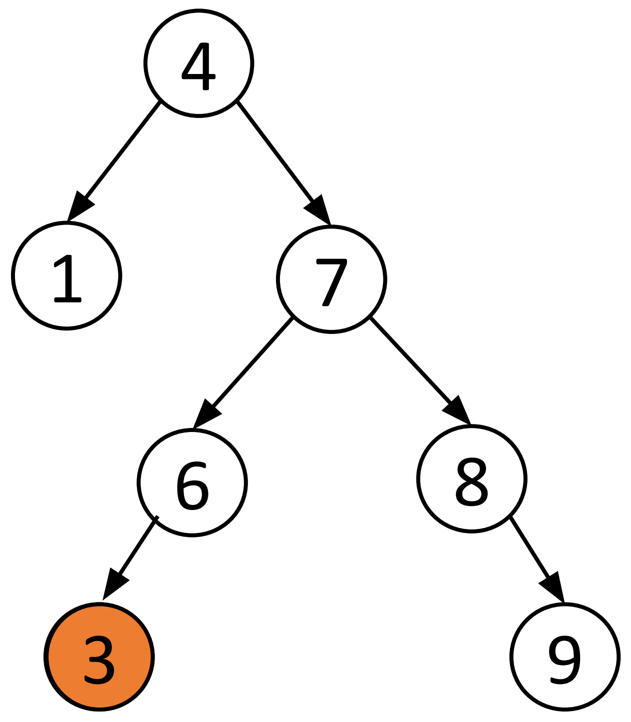
For every node n in a tree which has a value v:

- 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

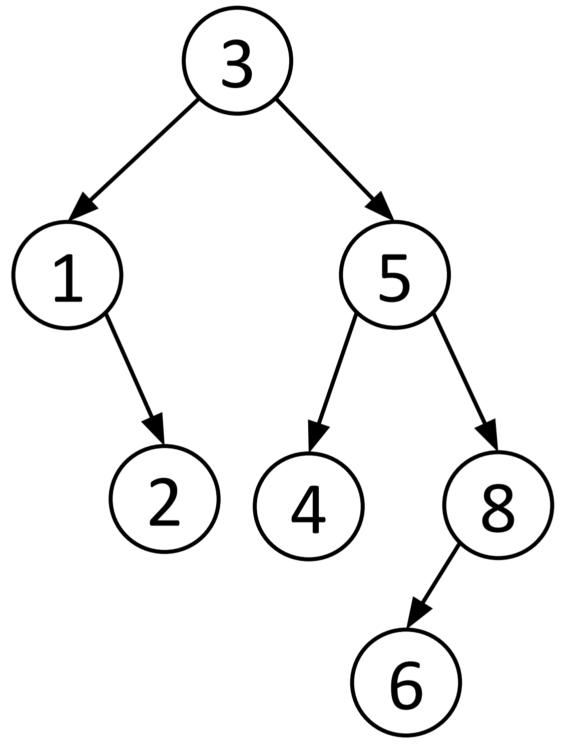
Note: the left and right subtrees are BSTs! BST constraints are **recursive**.



## Example: Is this a BST?



no



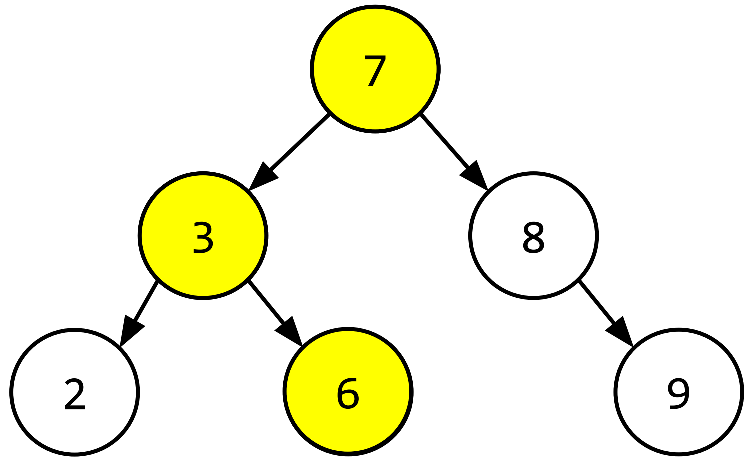
yes

## 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, and 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:

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

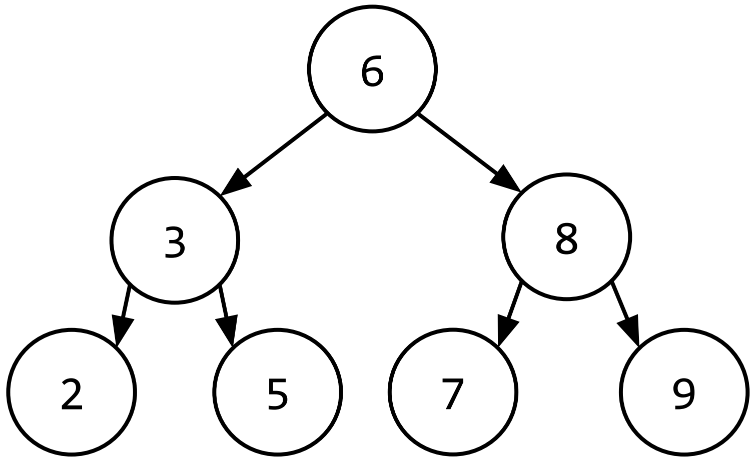
Note that we do just **one** recursive call, either on the left subtree or on the right subtree.

# 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.

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 you will indeed take O(log n) time.

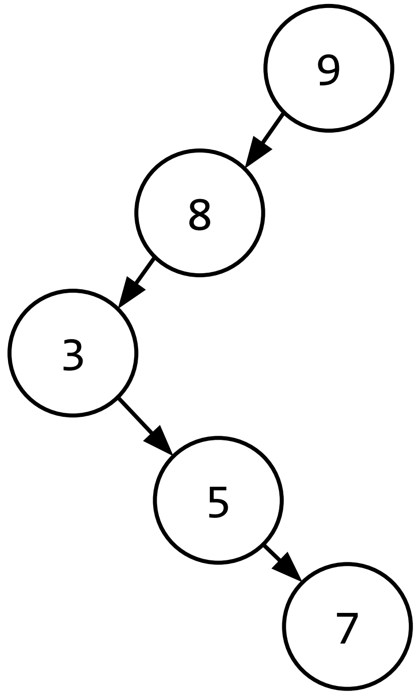


## BST Search Runtime – Unbalanced Trees

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! If you search it for a number like 6, it can still take **O(n)** time.

When we put data into BSTs, we usually strive to make them balanced to avoid these edge cases. You can assume the average runtime will be 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. But note: this will not keep the tree balanced. Rebalancing is beyond the scope of this course.

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

## Hashed Search

## Improving Search

We've discussed linear search (which runs in O(n)), and binary search (which runs in O(log n)).

We use search all the time, so we want to search as quickly as possible. **Can we search for an item in O(1) time?**

We can't *always* search for things in constant time, but there are certain circumstances where we can.

## Search in Real Life – Post Boxes

Consider how you receive mail. Your mail is sent to the post boxes at the lower level of the UC. Do you have to check every box to find your mail?

No- just check the one assigned to you.

This is possible because your mail has an **address** on the front that includes your mailbox number. Your mail will only be put into a box that has the same number as that address, not other random boxes.

Picking up your mail is a O(1) operation!

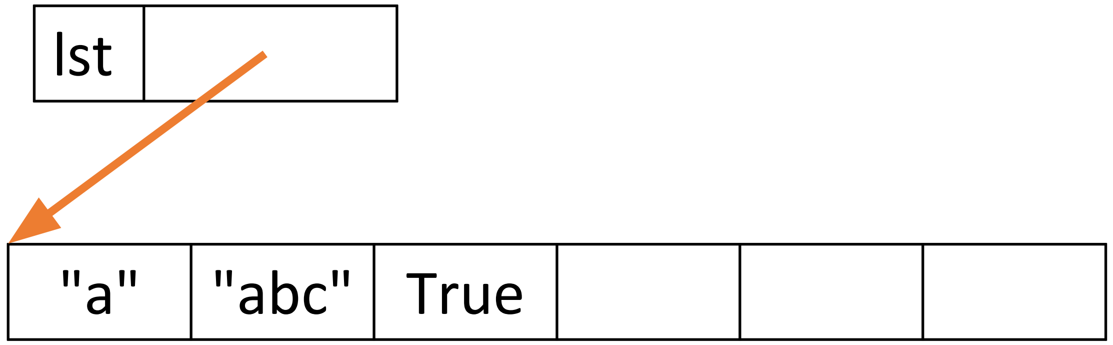
## Search in Programming – List Indexes

We can't search a list for an item in constant time, but we **can** look up an item based on an index in constant time.

Reminder: Python stores lists in memory as a series of **adjacent parts**. Each part holds a single value in the list, and all these parts use the same **amount of space**.

Example:

```
lst = ["a", "abc", True]
```



# Search in Programming – List Indexes

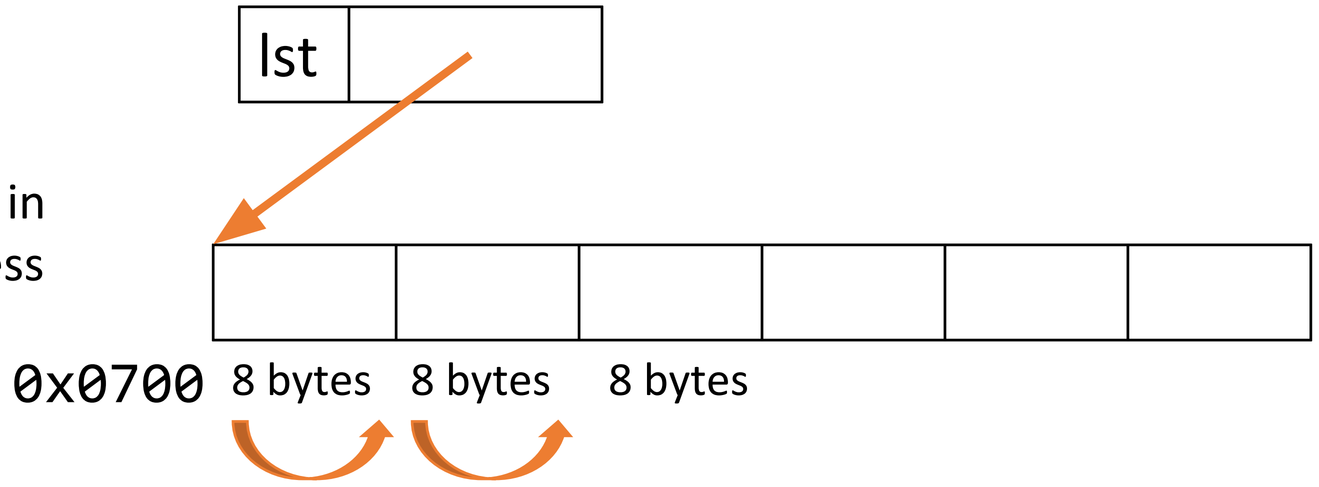
We can calculate the exact starting location of an index's memory address based on the first address where `lst` is stored. If the size of a part is N, we can find an index's address with the formula:

```
start + N * index
```

Example: in the list to the right, each part is 8 bytes in size and the memory values start at `x0700`. To access `lst[2]`, compute:

```
x0700 + 8 * 2 = x0716
```

**Given a memory address, we can get the value from that address in constant time.** Looking up an index in a list is O(1)!



## Combine the Concepts

To implement constant-time search, we want to combine the ideas of post boxes and list index lookup. Specifically, we want to be able to determine **which index a value is stored in based on the value itself**.

If we can calculate the index based on the value, and the number of possible indices increases with the number of values, we can retrieve the value in constant time.

## Hash Functions Map Values to Integers

In order to determine which list index should be used based on the value itself, we'll need to **map values to indexes**, i.e, integers.

We call a function that maps values to integers a **hash function**. This function must follow two rules:

- Given a specific value `x`, `hash(x)` must **always** return the same output `i`
- Given two different values `x` and `y`, `hash(x)` and `hash(y)` should **usually** return two different outputs, `i` and `j`

## Built-in Hash Function

We don't need to write our own hash function most of the time- Python already has one!

```
x = "abc"
hash(x)
```

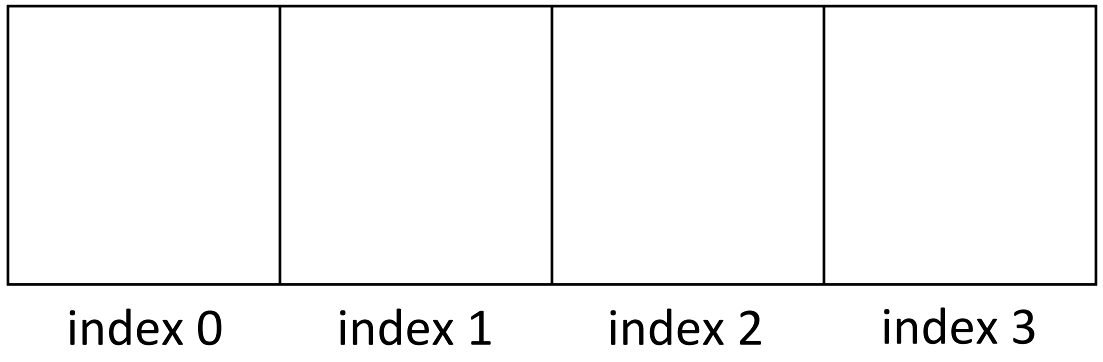
`hash()` works on integers, floats, Booleans, strings, and some other types as well.

# Hashtables Organize Values

Now that we have a hash function, we can use it to organize values in a special data structure.

A **hashtable** is a list with a fixed number of indexes. When we place a value in the list, we put it into an index **based on its hash value**, instead of placing it at the end of the list.

We often call these indexes 'buckets'. For example, the hashtable to the right has four buckets. Note that actual hashtables have far more buckets than this.



# Adding Values to a Hashtable

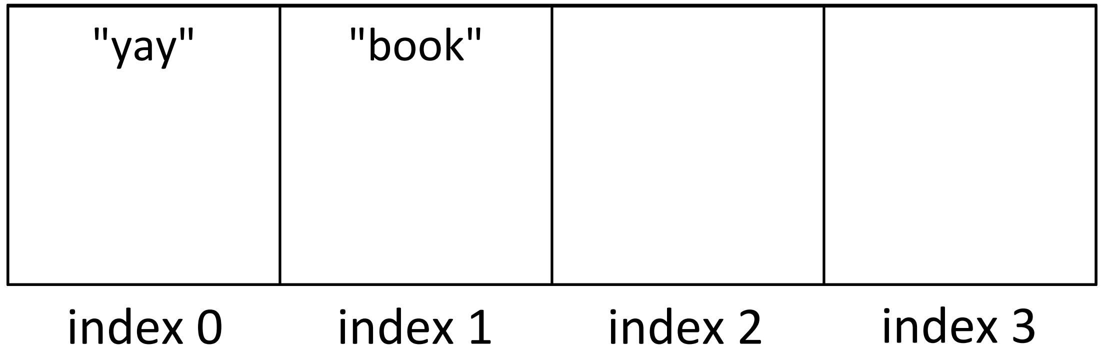
For simplicity, let's say this hashtable uses a hash function that maps strings to indexes using the first letter of the string, as shown to the right.

```
def hash(s):
    return ord(s[0]) - ord('a')
```

First, add `"book"` to the table.
`hash("book")` is `1`, so we'll put the value in bucket 1.

Next, add `"yay"`. The `hash("yay")` is `24`, which is outside the range of our table. How do we assign it?

Use `value % tableSize` to map integers larger than the size of the table to an index. `24 % 4 = 0`, so we put `"yay"` in bucket 0.



# Dealing with Collisions

When you add lots of values to a hashtable, two elements may **collide**. This happens if they are assigned to the same index. For example, if we try to add both `"cmu"` and `"college"` to our table, they will collide.

Hashtables are designed to handle collisions. They can put the collided values in a list and put that list in the bucket. If your table size is reasonably big and the indexes returned by the hash function are reasonably spread out, there will only be a **constant** number of values in each bucket.

Note: our example hash function is not good because it only looks at the first letter, so there are only 26 possible index values. A function that uses all the letters would be better.

```
def hash(s):
    return ord(s[0]) - ord('a')
```

| | | | |
|---|---|---|---|
| "yay" | "book" | "cmu"<br>"college" | |
| index 0 | index 1 | index 2 | index 3 |

# Searching a Hashtable is O(1)!

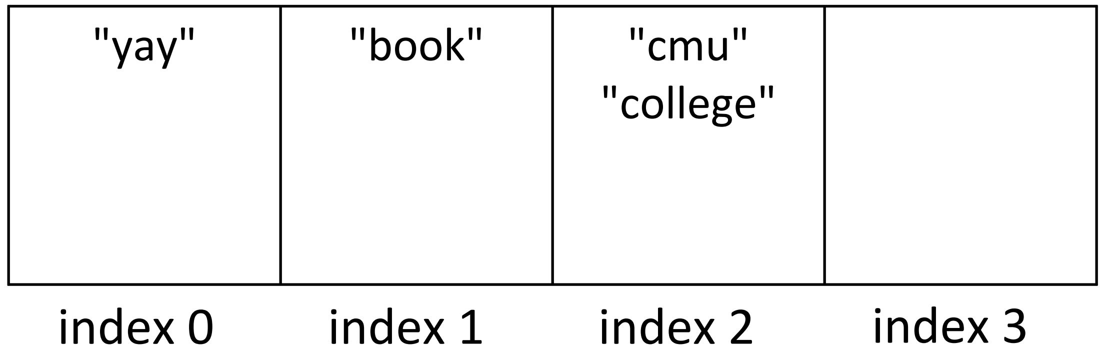
To search for a value, call the hash function on it, then mod the result by the table size. **The index produced is the only index you need to check!**

```
def hash(s):
    return ord(s[0]) - ord('a')
```

For example, we can check if `"book"` is in the table just by checking bucket 1.

If the value is in the table, it will be at that index. If it isn't, it won't be anywhere else either. To check for `"stella"` just look in in bucket 2.

Because we only need to check one index, and each index holds a constant number of items, finding a value is O(1).



## Activity: Compute the Hashed Index

Assume you're using a really simple hash function that maps floats to indexes by hashing them to the value in the ones place. For example, 42.5 would hash to 2.

We want to place the number 17.46 in a four-bucket hash table.

**Which bucket should it go into- 0, 1, 2, or 3?**

## Caveat: Don't Hash Mutable Values!

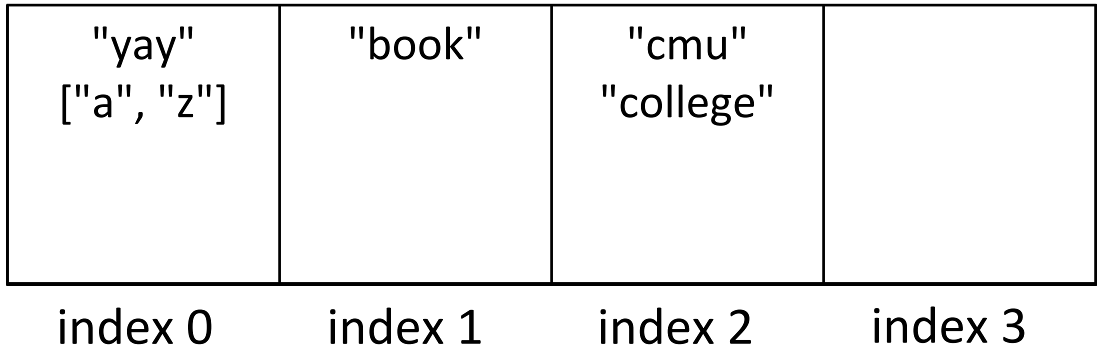
What happens if you try to put a list in a hashtable? Let's try using the given hash to add `lst`, where `lst = ["a", "z"]`.

This might seem fine at first, but it will become a problem if you change the list before searching. Let's say we set `lst[0] = "d"`.

Now, when we hash the list, the hashed value is `3`, not `0`. But the list isn't stored in bucket 3! We can't find it reliably.

For this reason, **we don't put mutable values into hashtables**. If you try to run the built-in `hash()` on a list, it will crash.

```
def hash(s):
    return ord(s[0]) - ord('a')
```



## Dictionaries Use Hashed Search

Because hashed search requires immutable search values and a hashtable, it isn't used in lists or trees. However, it **is** used to implement dictionary search.

Recall that the keys of a dictionary must be immutable. This is because those keys are all stored in a hashtable. Each key points to its own value, so that the values can still be accessed.

This means that searching a dictionary takes O(1) time! They are **super efficient** for basic lookup tasks.

## Breadth-First Search and Depth-First Search

## 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 a node exists **and is connected to some other node** by some path through the structure.

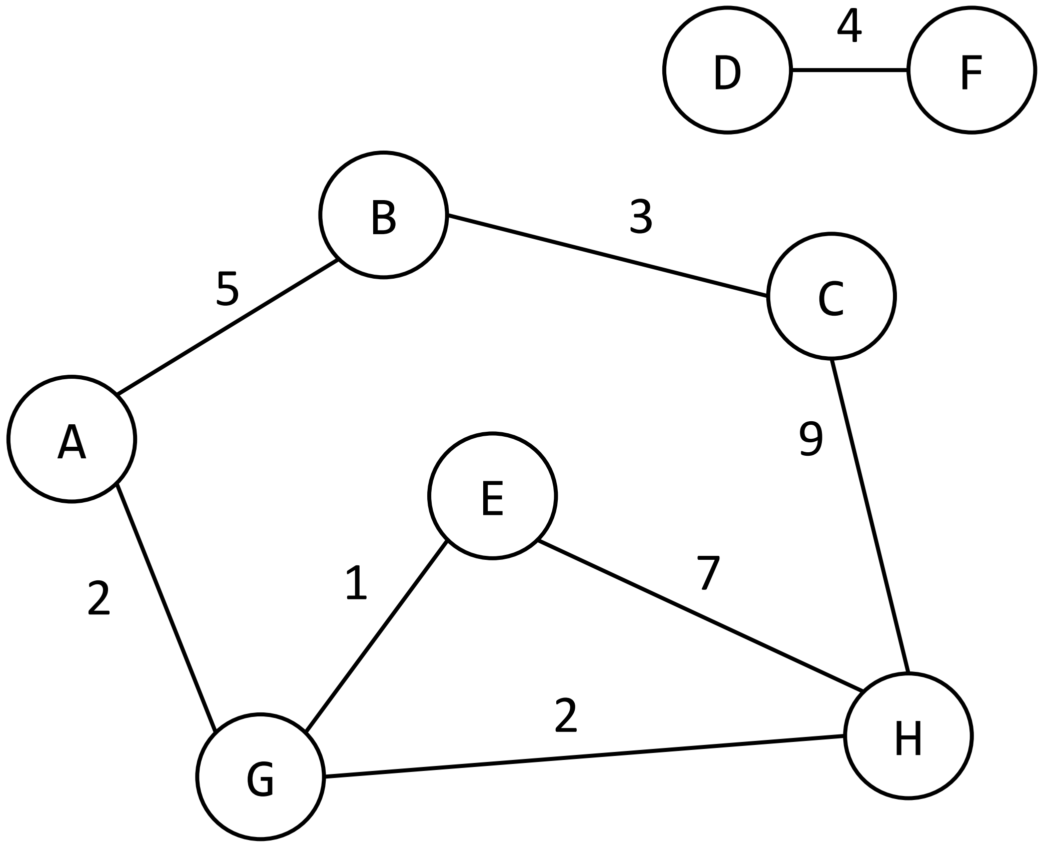
In other words: can we find a path that leads from a start node to a target node in the tree/graph?

This is useful in several contexts, including finding walking/driving routes in map applications.

## Discuss: How to Search?

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 follow the edges to find all the other nodes it's connected to. Here are two common approaches for determining in which order to visit the connected nodes.

In **Breadth-First Search (BFS)**, we slowly move outwards in the graph 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 the graph.

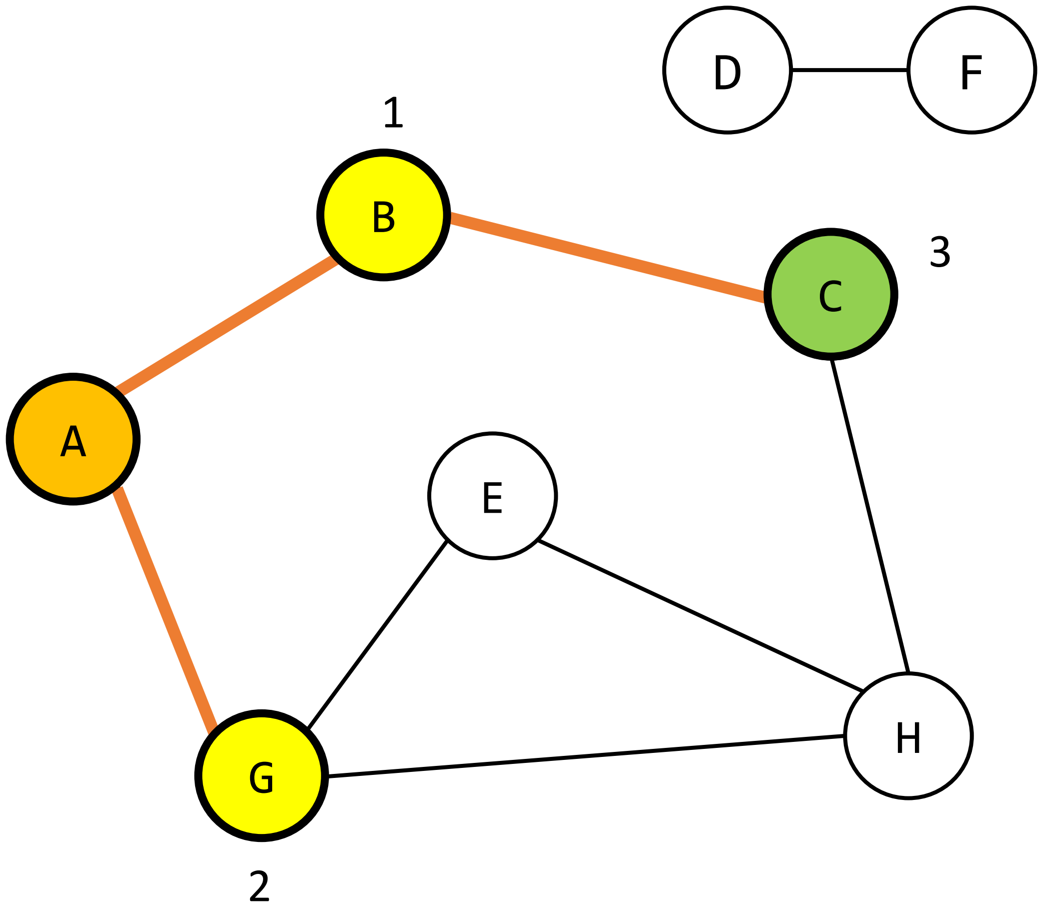
In **Depth-First Search (DFS)**, we go all the way down one potential path, then backtrack and try other possible paths. So we choose one neighbor, then choose one of its neighbors, etc., until there are no unvisited neighbors left.

# Breadth-First Search Example

Let's 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!



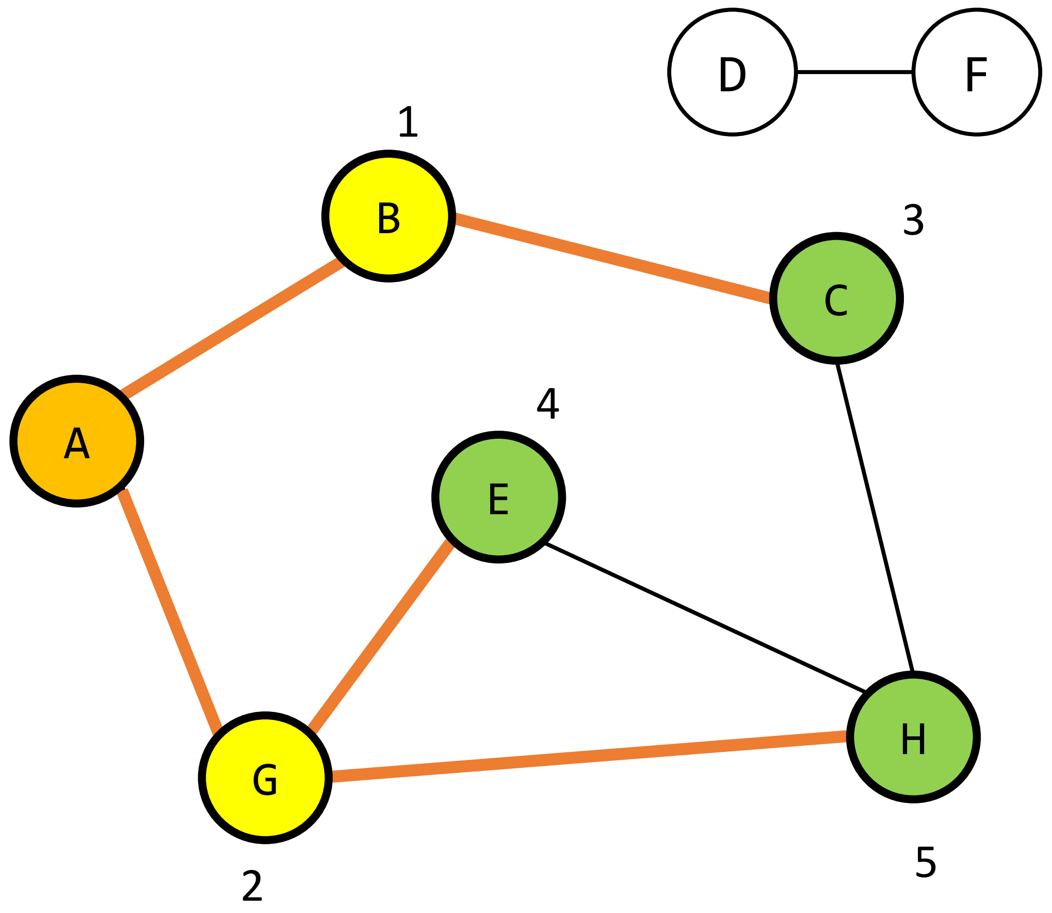
# Breadth-First Search Example

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.

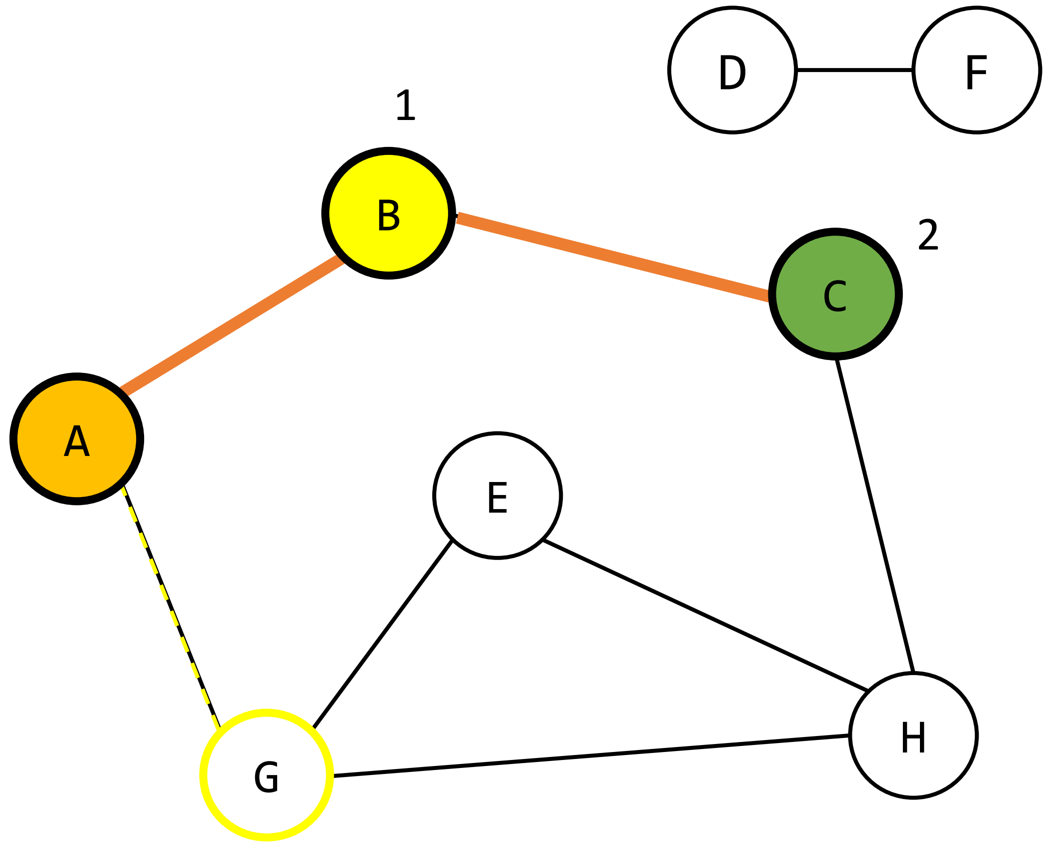


## Depth-First Search Example

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

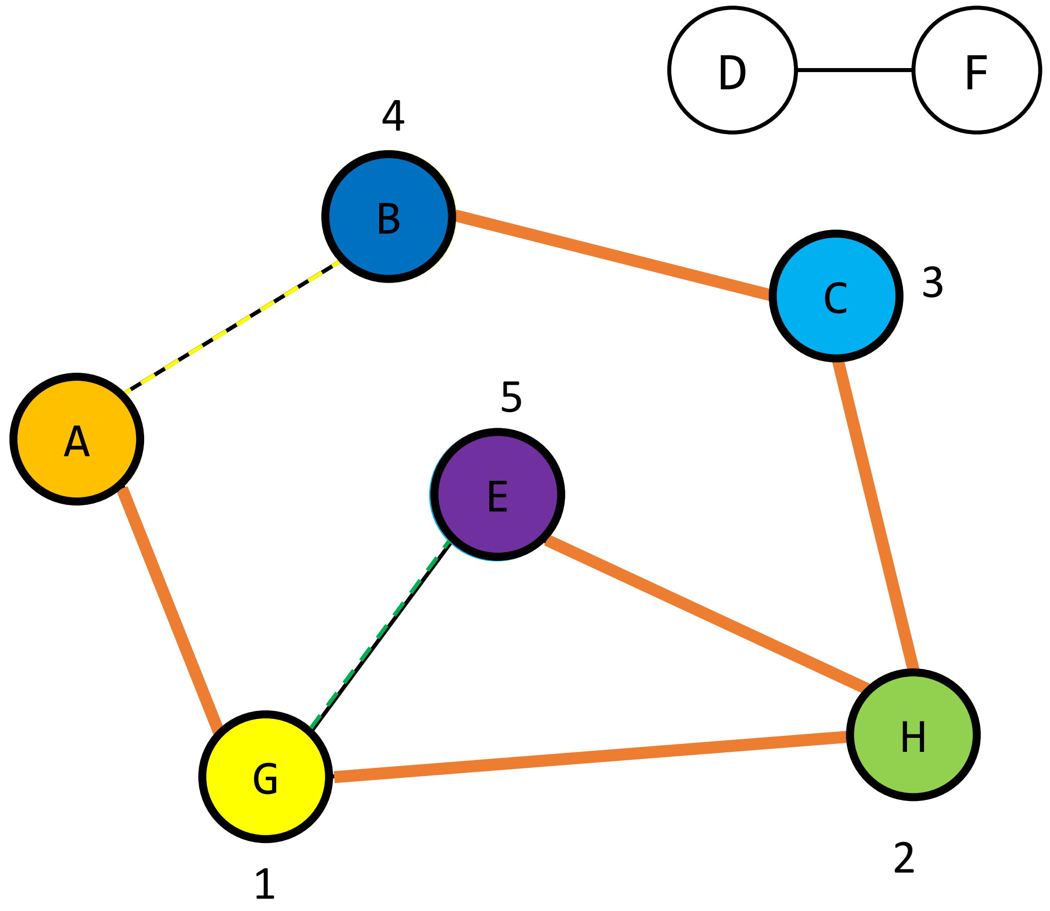
What if we search the example graph starting from A with depth-first search, now looking for D?

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. C's only remaining neighbor is B (H is not valid), so we must visit it.

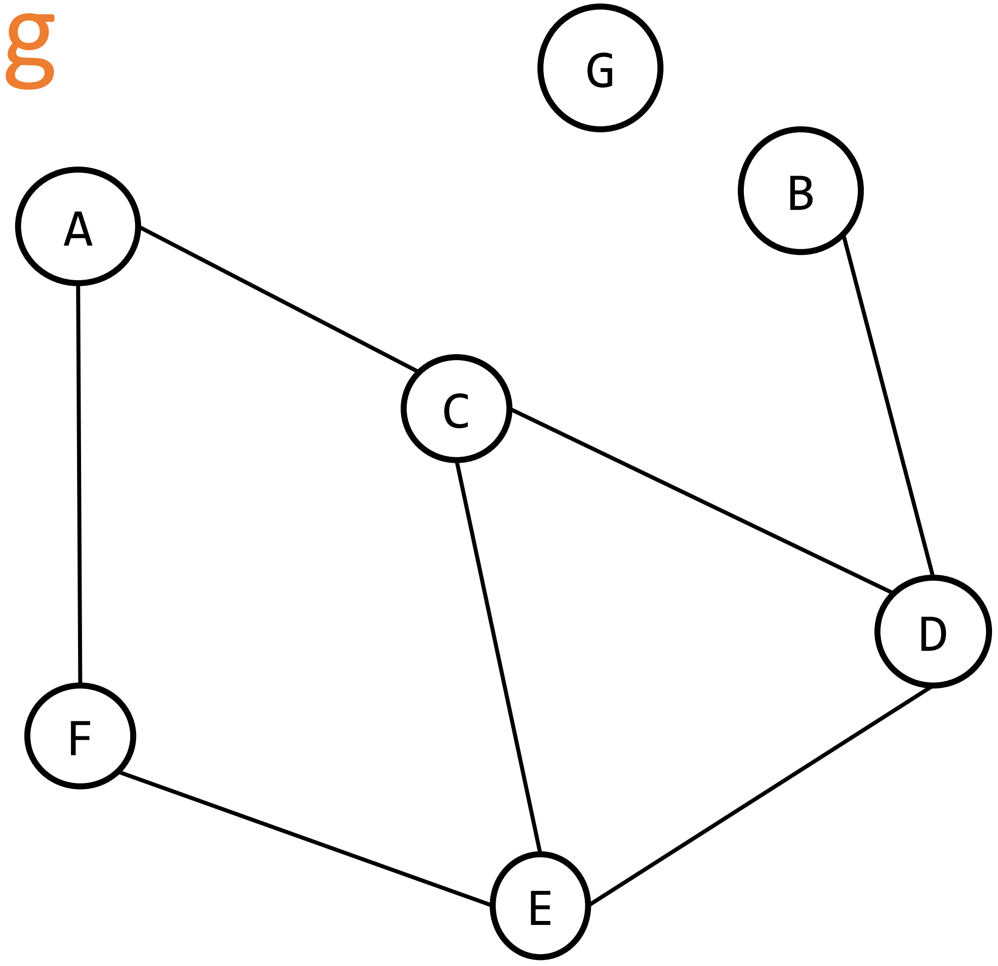
Now B has no unvisited neighbors remaining (A and C are both visited), so we must **backtrack** to the last node that had an unvisited neighbor. If we check our backup list, the only unvisited node remaining is E (which was G and H's neighbor). We visit E, and we're done.



## 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, 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, 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]
        nextNodes.pop(0)

        if next == target: # If it's what we're looking for- we're done!
            return True

        # Only expand this node if we haven't visited it before, to avoid repeats
        if next not in visited:
            visited.append(next)
            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, 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]
        nextNodes.pop(0)

        if next == target: # If it's what we're looking for- we're done!
            return True

        # Only check this node if we haven't visited it before, to avoid repeats
        if next not in visited:
            visited.append(next)
            nextNodes = g[next] + nextNodes
    return False
```

# BFS and DFS Runtime

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 node doesn't exist and the graph is fully connected.

```
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")
```

## Applying BFS and DFS

Breadth-First Search and Depth-First Search can be applied to trees as well. The only change to the algorithm is that the left and right children of each node are added to the list of nodes to visit, instead of adding larger numbers of children.

BFS and DFS are usually applied in circumstances where it is expected that a target will be nearby, or far away in a specific direction. BFS is useful for finding directions to a location within a 5 mile radius; DFS is useful for tracing a route that goes straight across the United States, east to west.

## Learning Objectives

- Identify whether or not a tree is a **binary search tree**
- Search for values in **binary search trees** using **binary search**
- Understand how and why **hashing** makes it possible to search for values in **O(1) time**
- Search for values in a **hashtable** using a specific **hash function**
- Search for values in **graphs** using **breadth-first search** and **depth-first search**

- **Feedback:** https://bit.ly/110-feedback