SWEN 601
Software Construction

Binary Trees, & BSTs

Activity: Getting Started

1. Begin by accepting the GitHub Classroom invitation for today's homework.
    a. *The project **<u>may</u>** already contain some code!*
2. Create a `session` package. This is where you will write your solutions to today's activities.
3. Create a `homework` package. This is where you will implement your solution to the homework.

When you submit your homework, you will include your activities. You may earn up to a 10% bonus on the homework if you have completed all of the activities.

<u>Do not</u> submit code that ***does not compile***. Comment it out if necessary.

Next Two Weeks

WEEK 05	SUN	MON	TUES	WEDS	THURS	FRI	SAT
QUIZ			Quiz #14		Quiz #15		
LECTURE			Binary Trees & Binary Search Trees		Heaps, N-ary Trees, & B-Trees		
HOMEWORK	Hwk 14 Due (**11:30pm**)		*Hwk 15 Assigned*		*Hwk 16 Assigned*	Hwk 15 Due (**11:30pm**)	

WEEK 04	SUN	MON	TUES	WEDS	THURS	FRI	SAT
QUIZ			Quiz #16		Quiz #17		
LECTURE			Graphs		BFS & DFS		
HOMEWORK	Hwk 16 Due (**11:30PM**)		*Hwk 17 Assigned*		*Hwk 18 Assigned*	Hwk 17 Due (**11:30pm**)	

Java ADT Implementations

	Interface	Implementation(s)	Comments
Stack	NA	`java.util.Stack`	There is no stack interface in Java. It is array-based.
Queue	`java.util.Queue`	`java.util.LinkedList`	The Queue interface defines add (enqueue) and poll (dequeue) methods. `LinkedList` is the provided implementation, meaning that random access is possible.
List	`java.util.List`	`java.util.LinkedList`	Node-based implementation.
		`java.util.ArrayList`	Array-based implementation.
Map	`java.util.Map`	`java.util.HashMap`	Hashing implementation of the Map interface provides constant time put and get implementations.
		`java.util.TreeMap`	Uses a tree to store keys in natural order. The performance of put and get operations is $O(N\log_2 N)$.
Set		`java.util.HashSet`	Set backed by a HashMap. Constant time add, remove, and contains operations.
		`java.util.TreeSet`	Set backed by a TreeMap; elements are ordered. Performance of add, remove, and contains operations is $O(N\log_2 N)$.

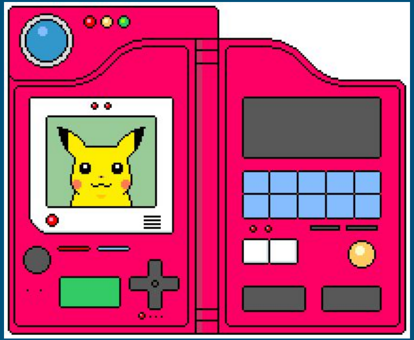
Gotta Catch 'em All!

- Pokémon trainers use a Pokédex to keep track of the Pokémon that they have captured.
- When encountering a Pokémon in the wild, it’s important for a trainer to know whether or not they have already captured that species.
- In order to implement the Pokédex, we will need a data structure that meets the following requirements:
    - ***Quickly search*** to determine whether or not a specific Pokémon is already in the collection.
    - ***Print*** all of the Pokémon in our Pokédex ***in order by number***.
- What is the best data structure to use to meet both of those requirements?

I need to know if it’s worth using my last Razz Berry and an Ultra Ball or not.

The Pokédex as a List

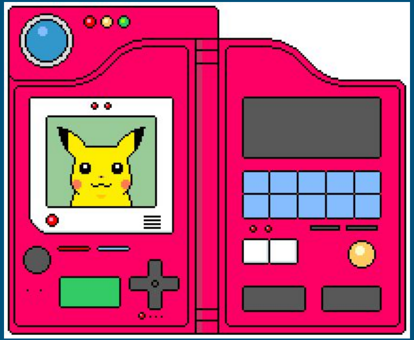
- Simply add each Pokémon to the end of a list as it is captured.
  - This is a ***constant time*** (**O(C)**) operation.
- How will you determine whether or not you have captured a specific Pokémon when you encounter it in the wild?
  - Perform a ***linear search***; an **O(N)** operation.
- What is the complexity of maintaining the Pokédex over time?
  - Each time you encounter a Pokémon, you will perform a linear search (**O(N)**).
  - The complexity over time will be **$O(N^2)$**.
- How will you print the Pokémon in order?
  - Search the list for the Pokémon with the next smallest number.
  - To print N Pokémon, you will need to perform N linear searches; again **$O(N^2)$**.
- There is probably a better way.



What if we try to keep the list in sorted order?

The Pokédex as a Sorted List

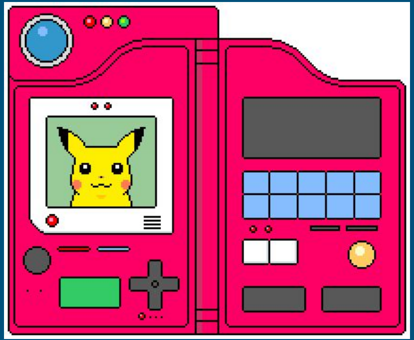
- *Sort* the list each time a new Pokémon is added.
- This will allow you to perform a ***binary search*** to determine whether or not you have already captured a Pokémon.
  - This is an **$O(\log_2 N)$** operation.
- What is the complexity of searching the Pokédex over time?
  - Each time you encounter a Pokémon, you will perform a ***binary search***.
  - The complexity over time will be **$O(N\log_2 N)$**. Much improved!
- Printing the Pokemon in order is as simple as ***iterating over the list***, an **$O(N)$** operation! Another big improvement!
- But what is the complexity of keeping the list ***sorted***?
  - Each ***insert*** is an **$O(N)$** operation.
  - You will perform an ***insert*** N times; **$O(N^2)$**. Ugh.
- Next!



What if we used a fast data structure like a Hash Map?

The Pokédex as a Hash Map

- Why not use a ***hash map*** to keep track of your captured Pokémon?
  - Use the Pokémon's ***number*** as the ***key***.
- Adding a new Pokémon is a ***constant time*** (**O(C)**) operation.
- Searching for a Pokémon is also a ***constant time*** (**O(C)**) operation.
- But what about printing the Pokémon in order by number?
  - Hash maps do not maintain the keys in a predictable order.
  - This means that you'd have to ***iterate*** over the keys looking for the Pokémon with the next smallest number; a **O(N)** operation.
    - And to print N Pokémon, you'd have to do the linear search ***N times***; **O(N^2)**. Gross.
  - Alternatively, you could ***copy*** all of the Pokémon into a list and ***sort*** them: **O($N+N\log_2 N$)**. Better, but not great.

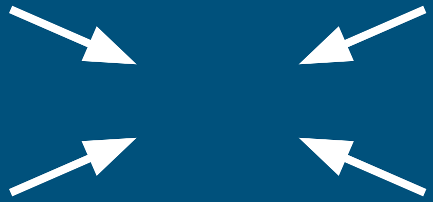


Seems like we'll need a new data structure to solve this problem.

Binary Trees... What Are They Even?

A ***binary tree*** is one of two things...

The ***empty tree***...

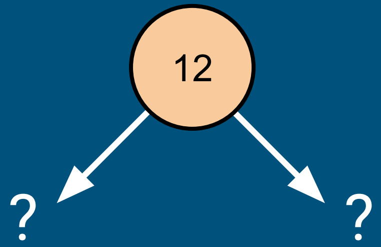


Can you see it? It's right there...

The empty tree is a tree with no data, and no sub-trees.

The empty tree is represented using `null`.

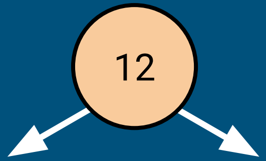
A ***non-empty tree...***



A non-empty tree includes ***a value*** and a ***left subtree*** and a ***right subtree***, either or both of which may be empty.

Parts of a Binary Tree

A non-empty tree comprises at least one ***node***.

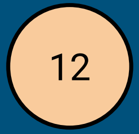


In diagrams, a node is represented as a circle.

The ***value*** is written inside the circle.

The ***subtrees*** are indicated using arrows.

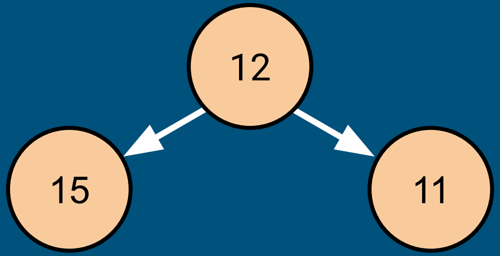
A ***leaf*** is a node whose left and right subtrees are both empty.



The arrows are not drawn for empty subtrees.

These subtrees may also be referred to as ***empty nodes*** and are represented using `null`.

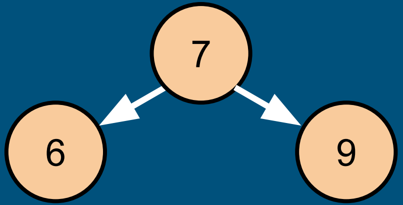
An ***internal node*** is a node that is not a leaf.



That is to say that **one** or **both** of its subtrees is not an empty tree.

Parts of a Binary Tree

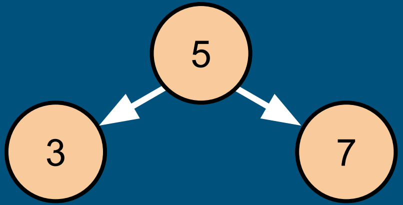
A node, **n_c**, is a ***child*** of another node, **n_p**, if **n_c** is the left or right subtree of **n_p**.



In this example, **n_6** is a child of **n_7** because **n_6** is the *left subtree*.

n_9 is also a child of **n_7** because **n_9** is the *right subtree* of **n_7**.

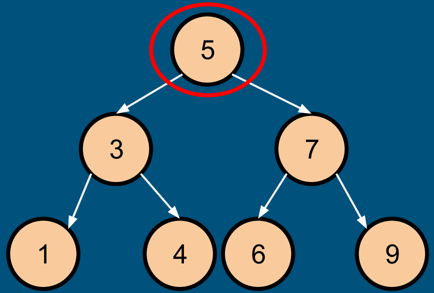
A node, **n_p**, is a ***parent*** of another node, **n_c**, if **n_c** is a subtree of **n_p**.



In this example, **n_5** is the parent of **n_7** because **n_7** is the *right subtree* of **n_5**.

n_5 is also the parent of **n_3** because **n_3** is its *left subtree*.

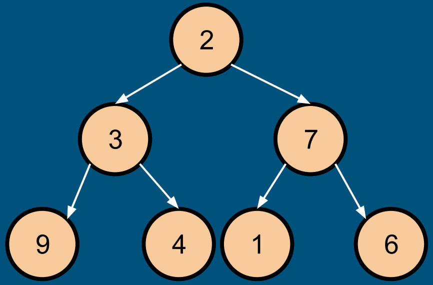
The ***root node*** is a node that has no parent.



In this example, **n_5** is the root node because it has no parent.

There is only *one* root node in a binary tree.

Activity: A BinaryNode



You should remember the `Node` class that we wrote and used to implement the `NodeStack` and `NodeQueue`. Now we're going to create a similar `class` to represent a node in a binary tree: `BinaryNode`.

- Create a new `class` called `BinaryNode`.
- The `class` should have the following methods:
  - A least one constructor that takes an `int` value.
  - A getter/setter for its `int` value.
  - A getter/setter for its left subtree (a `BinaryNode`).
  - A getter/setter for its right subtree (a `BinaryNode`).
- Add a `main` method and build the binary tree shown on the left.

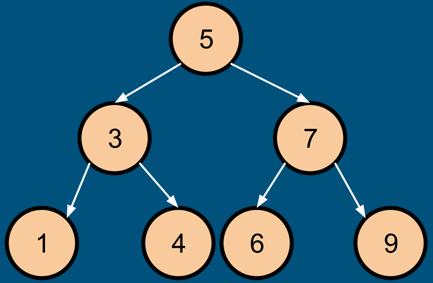
Printing the Pokédex

- The next part of the Pokédex problem is to print all of the Pokémon in the Pokédex by number.
- This will require us to ***stringify*** the binary tree that implements our Pokédex.
    - "Stringify" is a technical term that I totally did not just make up that refers to making a string from the values in the tree.
- Converting a binary tree into a string requires that we:
    - ***Visit*** each node in the tree.
    - Add its value to a string using ***concatenation***.
- The process of visiting each node in a tree is also referred to as ***traversing*** the tree.

I'd like to brag about all of the Pokémon that I've captured!

Infix (in-order) Traversal

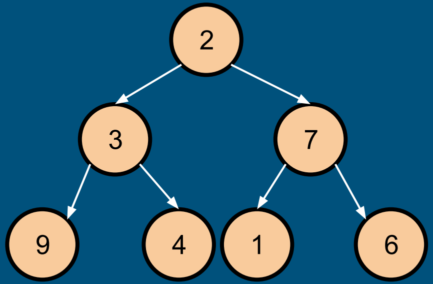
- There is more than one way in which to traverse a binary tree; the order in which the nodes are visited, and therefore the results, may change.
- Consider the binary tree to the right. If you wanted to print all of the node values in increasing numerical order, in which order would you need to visit each nodes?
  - First visit the ***left subtree***.
  - Then visit the ***middle node***.
  - Finally, visit the ***right subtree***.
- This is called an ***infix traversal*** or sometimes an ***in order traversal***.
  - The nodes are always visited in the order: ***left-middle-right***.



Consider how changing the order in which the nodes are visited would change the string we are building.

We will examine the other kinds of tree traversals in the homework.

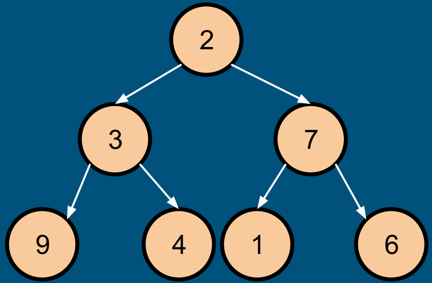
Activity: A Visitor Interface



There are lots of reasons that you may want to traverse a tree; converting it into a string is just one of them. Let's write an interface that can be used to visit each of the nodes in the tree.

- Name your new `interface Visitor`.
- It will define a single method to visit a node in the tree.
  - `void visit(BinaryNode node)`
- Now write a class called `StringifyVisitor` that implements your interface.
  - Each time it visits a node, it should append the node's value to a string (use a `StringBuilder`).
  - Add a `toString()` method that returns the string.

Activity: Implement Infix Traversal



The order is all wrong! How to we make sure that the nodes print in the correct order?

We will discuss that soon!

Now we will implement a method to perform an infix traversal of the binary tree.

- `public void infixTraversal(Visitor visitor)`
  - If the ***left subtree*** is not `null`, traverse it ***first***.
  - Then the node should visit ***itself***.
  - If the ***right subtree*** is not `null`, traverse it ***last***.

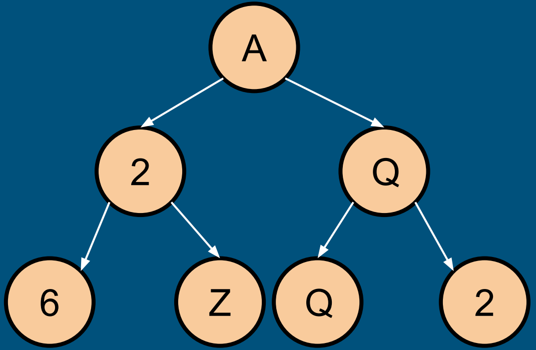
Update your `main` method to create a `StringifyVisitor` and call the `infixTraversal` method on the tree that you built previously. Print the resulting string.

- It should print the string: `9 3 4 2 1 7 6`

Searching a Binary Tree

- Once the values are added to a binary tree, how would we ***search*** the tree to determine whether or not it contains a specific value?
- Binary trees are a ***recursive*** data structure, so it should come as no surprise that a recursive algorithm can be used to search a binary tree ***BT***.
  - If `value` matches the target, return `true`.
  - Otherwise, if the ***left subtree*** is not `null`, search it. If the target is found, return `true`.
  - Otherwise, if the ***right subtree*** is not `null`, search it. If the target is found, return `true`.
  - Otherwise, return `false`.

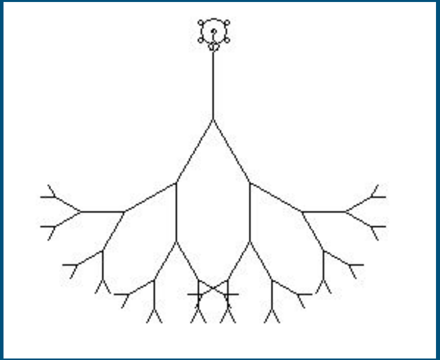
Determining whether or not a value is present in a binary tree requires a **search**.



If the target is not in the root node, search one of the subtrees.

If the target still isn't not found, search the other subtree.

Activity: Binary Tree Search



Using the algorithm below, add amethod to your `BinaryNode` that will search for a `target` value. Return `true` if the value is found, and `false` otherwise.

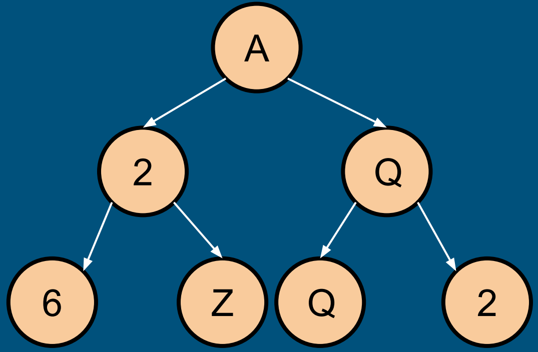
- `public boolean search(int target)`
  - If `value` matches the target, return `true`.
  - Otherwise, search the left subtree, and return `true` if the target is found.
  - Otherwise, search the right subtree, and return `true` if the target is found.
  - Otherwise, return `false`.

Update your `main` method to search your binary tree for several values and print the results of the search.

Search Complexity

- What is the average complexity of searching a binary tree with N nodes?
  - On average, about ***half*** of the nodes will need to be searched, so O(½N) or just O(N).
- This is no better than a ***linear search***! Is there a way that we can improve the efficiency of searching a binary tree?
- A ***binary search*** runs in ***logarithmic time*** ($O\log_2 N$) because it eliminates ***half*** of the values with each iteration; it chooses to search only the left half or the right half of an array.
- Is there a way to arrange values in a binary tree to accomplish the same thing?

A binary tree is ***any*** tree with the structure described previously.



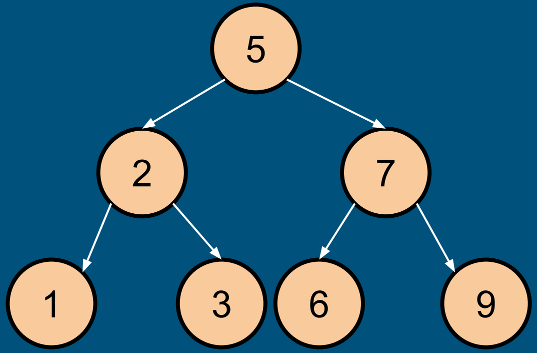
Each node has two children: a left subtree and a right subtree.

There is no relationship between the data in the nodes.

Binary Search Trees (BSTs)

- Assuming that the values in a binary tree can be arranged into some natural order, we can build the tree so that it is more efficiently searchable.
- Each time a new value is added to the tree:
  - If the new value comes ***before*** the root's value in natural order, then it is added to the ***left subtree***.
  - If the new value comes ***after*** the root's value in natural order, then it is added to the ***right subtree***.
- The process continues recursively through each node in the tree until an ***empty subtree*** is found.
  - The new value is added in place of the empty subtree.

A ***binary search tree*** (***BST***) is a special binary tree.

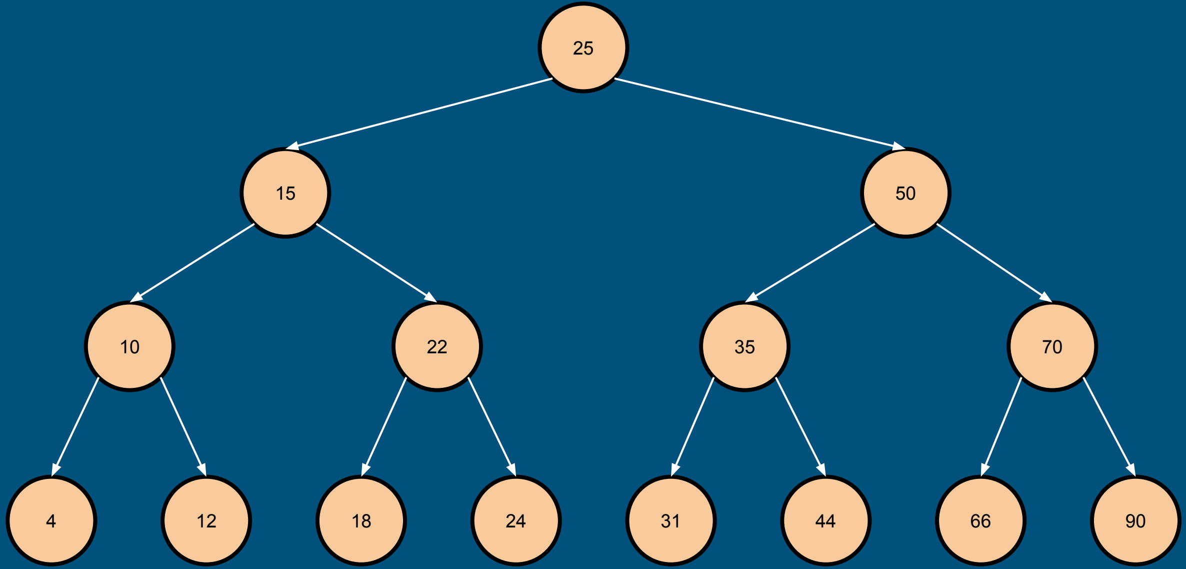


From any node in the tree, the left subtree contains only values that come ***before***...

...and the right subtree contains only values that come ***after***.

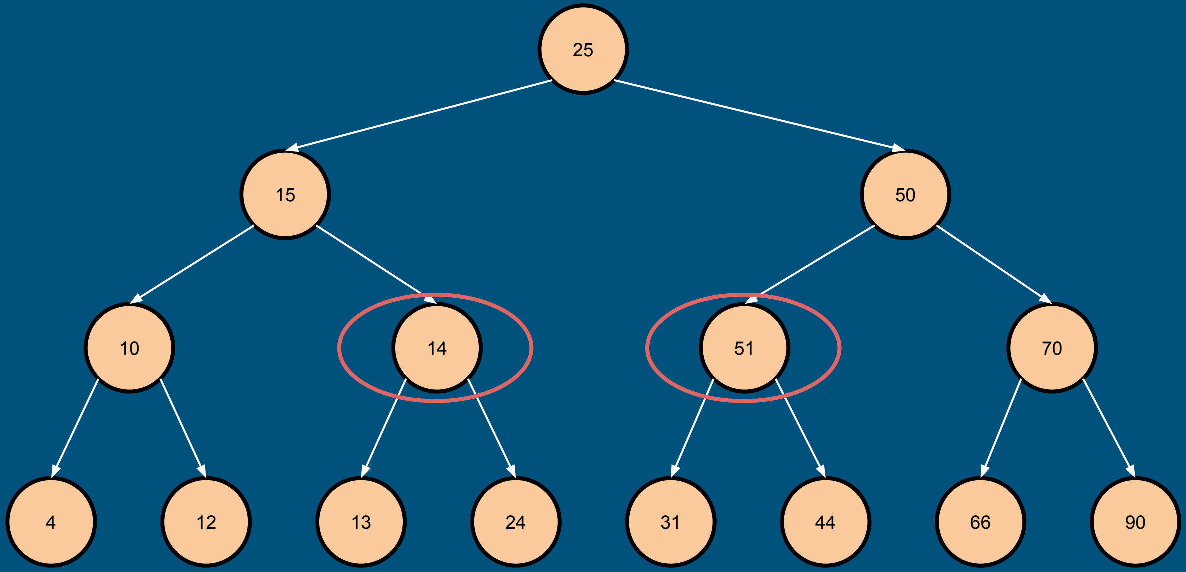
Identifying Binary Search Trees

- Is this a Binary Search Tree? If not, why not?



Identifying Binary Search Trees

- Is this a Binary Search Tree? If not, why not?

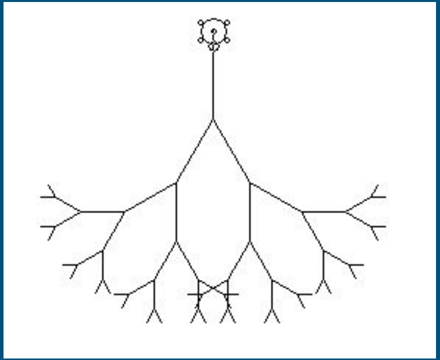


Inserting into the Pokédex

- So now that we know that we can implement the Pokédex using a ***binary search tree***, how do we go about adding a new Pokémon to the Pokédex?
- Each new Pokémon will need to be ***inserted*** into the correct position in the BST.
- Again, because a BST is a recursive data structure, a recursive algorithm can be used.
  - If the new value comes ***before*** `value`, insert the new value into the ***left subtree***.
  - Otherwise, insert the new value into the ***right subtree***.
  - Repeat this until an ***empty subtree*** is found. Create a new `BinaryNode` and add it to the tree.

Once I've captured a new Pokémon, I need to insert it into my Pokédex!

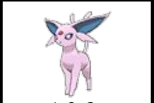
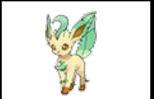
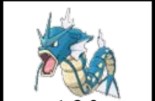
Activity: Implementing Insert



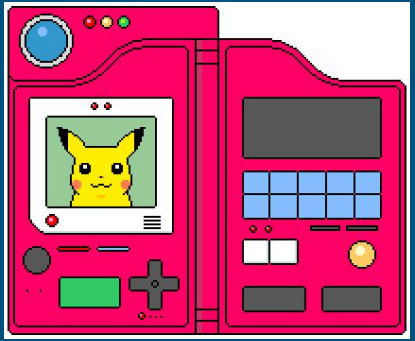
Add a new method to your `BinaryNode` class that will insert a new value into the correct place in the tree using the algorithm below.

- `public void binaryInsert(int value)`
    - If the new value comes ***before*** `value`, insert the new value into the ***left subtree***.
    - Otherwise, insert the new value into the ***right subtree***.
    - Repeat this until an ***empty subtree*** is found. Create a new `BinaryNode` and add it to the tree.

Available Pokémon

039	004	147	075	129	008	146	020	026	036
132	093	001	145	174	050	 196	003	151	019
 470	009	051	054	 130	144	052	006	076	094
074	197	053	025	136	002	035	143	134	055
065	005	113	149	173	172	064	133	446	471
150	242	007	148	063	135	700	013	092	040

Activity: Building Your Pokédex



Modify your `main` so that it uses your `insert` function to build a Pokédex with 10-15 of your favorite Pokémon chosen from the table on the previous slide.

- Create the root node with the first Pokémon.
- Insert each Pokémon using its number. Do not worry about inserting them in order; if you've implemented your `insert` method correctly, they will be inserted in order.
- What does your function print now?

Searching the Pokédex

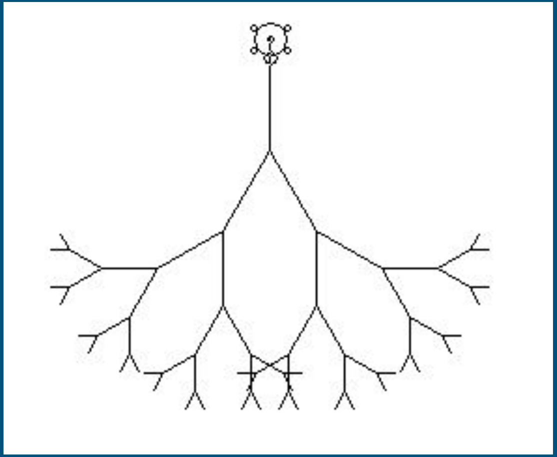
- Each time a new Pokémon is encountered, it is essential that we can ***quickly search*** the Pokédex and determine whether or not we've already captured a Pokémon of that species.
- It should come as no surprise that there is a recursive algorithm for searching a BST that is only a little different from the algorithm to search a plain binary tree.
  - If `value` matches the target, return `true`.
  - If the target is ***less than*** `value`, search the ***left subtree***.
    - Return `false` if the left subtree is `null`.
  - Otherwise, search the ***right subtree***.
    - Return `false` if the right subtree is `null`.

Ah, the rare and elusive **Bidoof**. Do I already have one of these?
Bidoof♂ Lv4
HP
Wild Bidoof appeared!

The old search method blindly searches the left subtree and then the right. It doesn't pay attention to the relationship between the `value` and the `target`.

In a plain ***binary tree***, this is the only option. But in a ***BST***, we can such much more efficiently.

Activity: Implementing Search

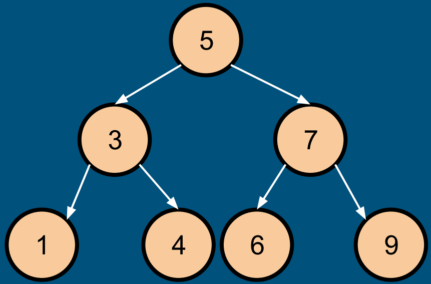


Add a new method to your BinaryNode class that will search for a target value in the tree.

- `public boolean binarySearch(int target)`
    - If `value` matches the target, return `true`.
    - If the target is ***less than*** `value`, search the ***left subtree***.
        - Return `false` if the left subtree is `null`.
    - Otherwise, search the ***right subtree***.
        - Return `false` if the right subtree is `null`.

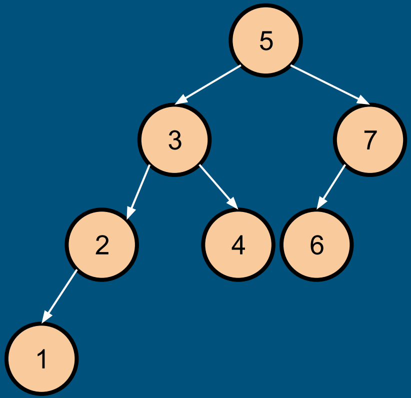
Balance

A binary tree is ***balanced*** if the left and right subtrees are both ***bushy*** and about equally so.



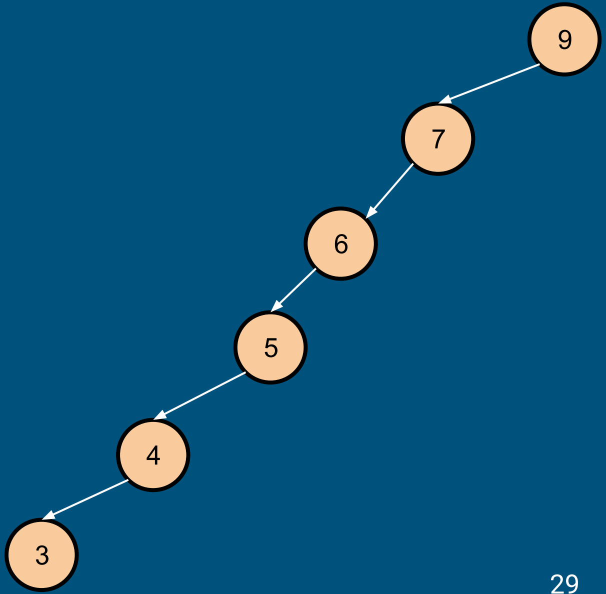
This means that there are approximately the same number of nodes in both halves of the tree.

A binary tree is ***unbalanced*** if there are more nodes in one half of the tree than the other.



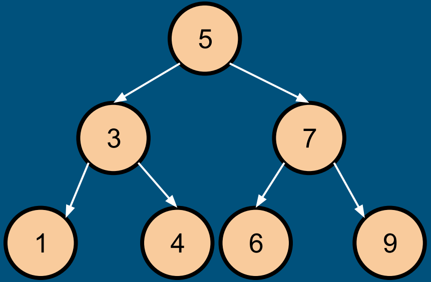
Unbalanced trees are more ***branchy*** than ***bushy***.

In the extreme case, an unbalanced tree becomes a ***list***.

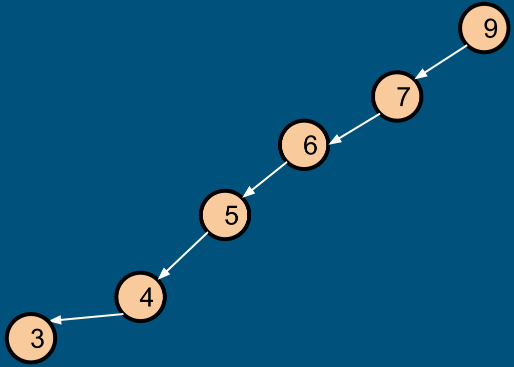


Complexity of Searching

- The complexity of searching a binary search tree depends on how ***balanced*** the tree is.
  - A balanced tree has the same number of nodes in the left subtree as it has in its right subtree.
  - These trees are referred to as ***bushy***.
- An ***unbalanced*** tree has more nodes in one subtree than the other.
  - These trees are referred to as ***branchy***.
  - The extreme case is a ***list***.
- If a BST is balanced, a search eliminates ***half*** of the nodes in the tree with each iteration, just like a binary search; the complexity is **$O(\log_2 N)$**.
- If the BST is unbalanced, the performance approaches linear time (**$O(N)$**).



Each iteration eliminates half of the nodes in a balanced tree (the half that is larger or smaller than the target). The complexity is therefore $O(\log_2 N)$.



In the extreme case in an unbalanced tree, every node is searched: $O(N)$.