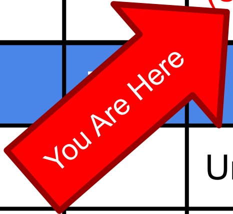
GCIS-124
Software Development & Problem Solving

6.1: Data Structures II

RIT | Golisano College of Computing and Information Sciences

SUN	MON (2/19)	TUE	WED (2/21)	THU	FRI (2/23)	SAT
Unit 5: Data Structures I					Unit 6: Data Structures II	
			Assignment 5.1 Due (start of class)		Unit 5 Mini-Practicum Assignment 5.2 Due (start of class)	

SUN	MON (2/26)	TUE	WED (2/28)	THU	FRI (3/1)	SAT
Unit 6: Data Structures II					Unit 7: Data Structures III	
	Midterm Exam 2 will be on ***Friday, March 8th*** (units 4-6).		Assignment 6.1 Due (start of class)		Unit 6 Mini-Practicum Assignment 6.2 Due (start of class)	



Data Structures II

You are already familiar with many data structures ***from the perspective of a user*** including arrays, queues, stacks, lists, sets, and dictionaries.

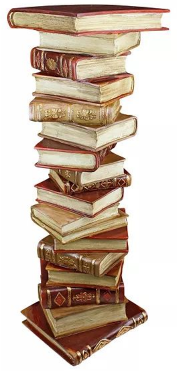
In this unit we will learn how ***different implementations*** of many of these data structures work.

We will also examine the situational ***strengths and weaknesses*** of each implementation in terms of ***complexity*** (both space and time).

- In this unit we will continue exploring ***abstract data types*** and ***data structures***.
  - An abstract data type describes the behavior of a data structure from the perspective of its user without providing any implementation details.
  - Every abstract data type (ADT) may be implemented in more than one way.
  - When solving a new problem, choosing the most efficient implementation is just as important as choosing the correct data structure.
- We will examine several different abstract data types and implement each of them in at least one way including:
  - Binary Trees
  - Binary Search Trees
  - Heaps
  - Comparable/Comparators
  - Tree Sets & Tree Maps
- We will begin by focusing on a new data structure: the ***binary tree***.

Review: Abstract Data Types

- A ***data structure*** is a grouping of related data.
  - We refer to the data as ***elements***.
- Most data structures provide some common operations including ***store***, ***retrieve***, ***remove***, and ***size***.
- An ***abstract data type*** (***ADT***) defines the behavior of a data structure from the point of view of its ***user***.
  - This includes the possible values and the operations available.
- Choosing the ADT that provides operations suitable for solving a problem usually isn't enough.
  - Each ADT may be implemented in ***multiple ways***.
  - Choosing the ***correct implementation*** is also important.
  - This requires understanding the ***complexity*** of each operation under different circumstances.



Choosing the correct ***abstract data type*** involves understanding the nature and constraints of the problem being solved.

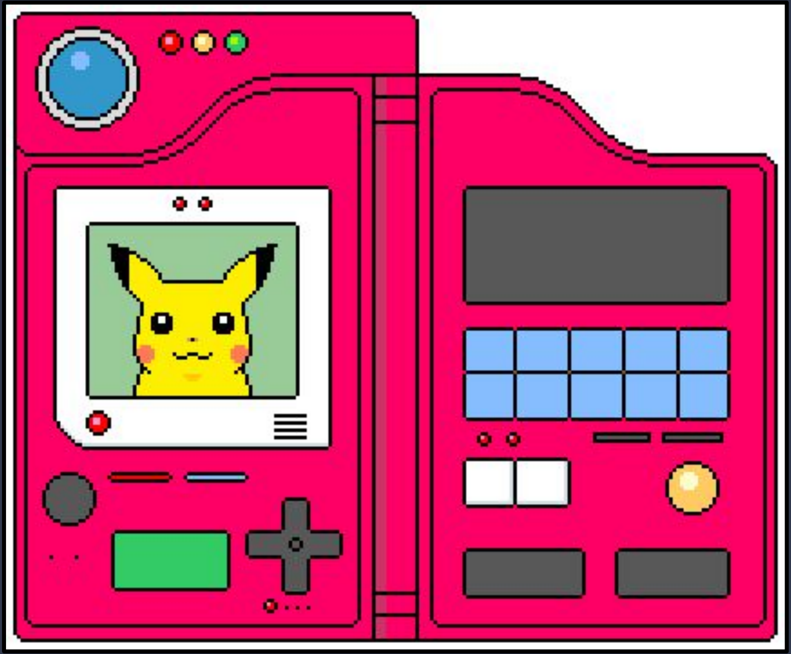
Choosing the correct ***implementation*** involves understanding which ***operations*** will be used the most and which of the available options provides the ***most efficient implementation*** of those operations.

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 that meets 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

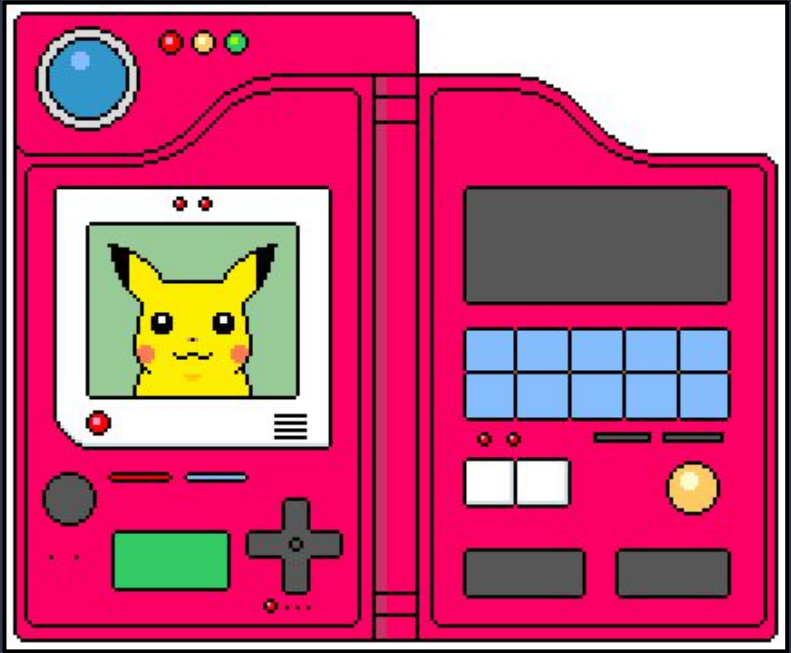


What if we try to keep the list in sorted order?

- 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.

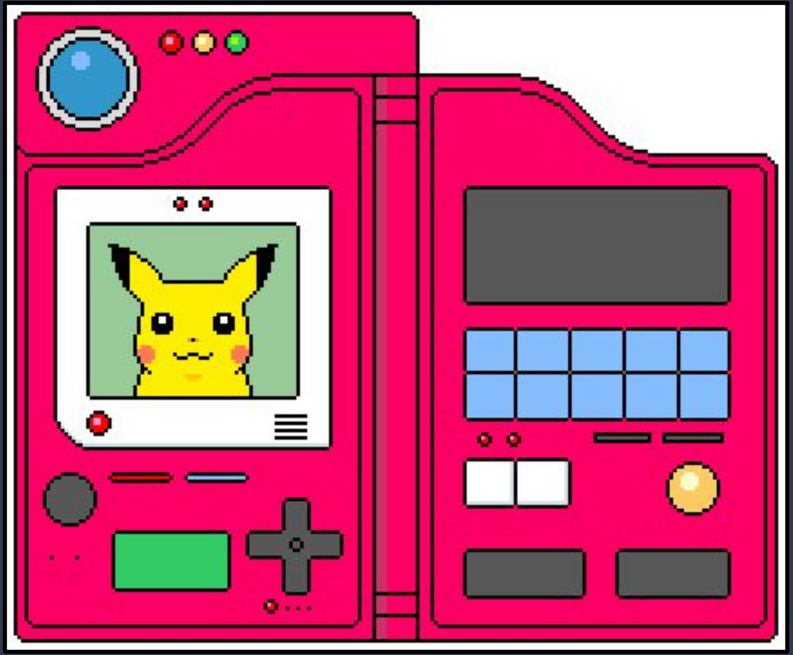
The Pokédex as a *Sorted* List

- ***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 Python dictionary?

The Pokédex as a Dictionary



If we want to avoid a lot of slow operations, it seems like we'll need a new data structure to solve this problem.

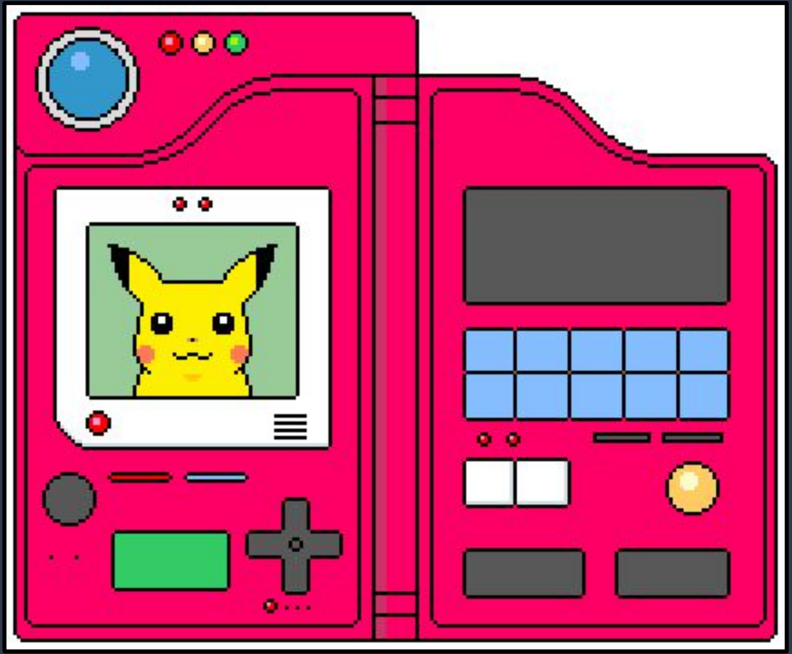
- Why not use a ***dictionary*** to keep track of your captured Pokémon?
  - Use the Pokémon's ***number*** as the ***key***.
- Adding a new Pokémon is a ***constant time*** (**O(C)**) operation.
- Searching for a Pokémon is also a ***constant time*** (**O(C)**) operation.
- But what about printing the 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.

Pokédex Options

Let's look at the data structures we already know and consider the complexity to:

- Add a Pokémon to the Pokédex.
- Determine if a Pokémon is already in the Pokédex.
- Add N Pokémon to the Pokédex.
- Print the Pokédex in order

	List	Sorted List	Dictionary
Add	O(C)	O(N)	O(C)
Find	O(N)	$O(\log_2 N)$	O(C)
Add N	O(N)	$O(N^2)$	O(N)
Print	$O(N^2)$	O(N)	$O(N+N\log_2 N)$



We have learned several options for storing data but none of them are great for this problem.

Binary Trees... What Are They Even?

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

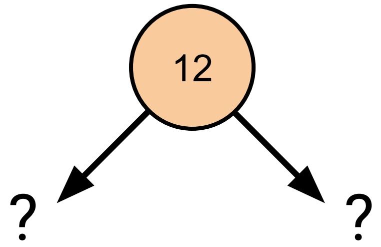
An ***empty tree***...

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`.

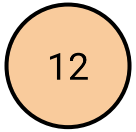
A ***non-empty tree...***



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***.

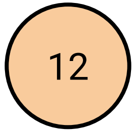


In diagrams, a node is represented as a circle.

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

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

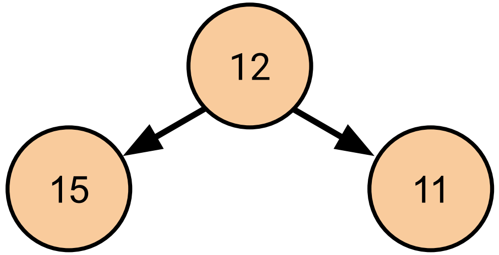
A ***leaf*** is a node whose left and right subtrees are both empty.



The arrows are not drawn for empty subtrees.

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

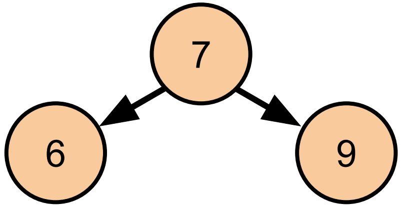
An ***internal node*** is a node that is not a leaf.



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

Parts of a Binary Tree

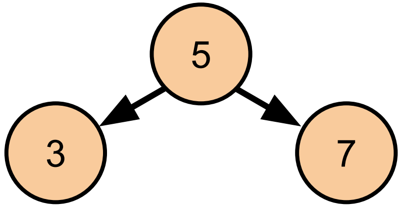
A node ***c*** is a ***child*** of another node ***p*** if it is the left or right subtree of ***p***.



In this example, **6** is a child of **7** because **6** is the *left subtree*.

9 is also a child of **7** because **9** is the *right subtree* of **7**.

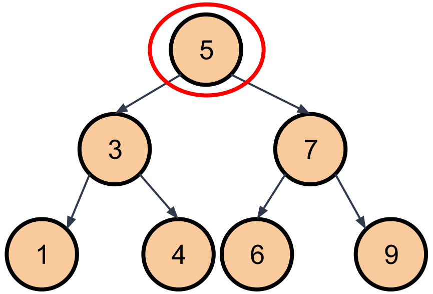
A node ***p*** is a ***parent*** of another node **c** if **c** is one of its subtrees.



In this example, **5** is the parent of **7** because **7** is the *right subtree* of **5**.

5 is also the parent of **3** because **3** is its *left subtree*.

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

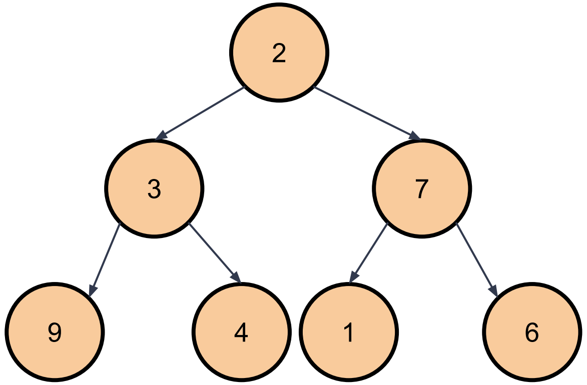


In this example, **5** is the root node because it has no parent.

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

6.1 A Binary Node

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`.



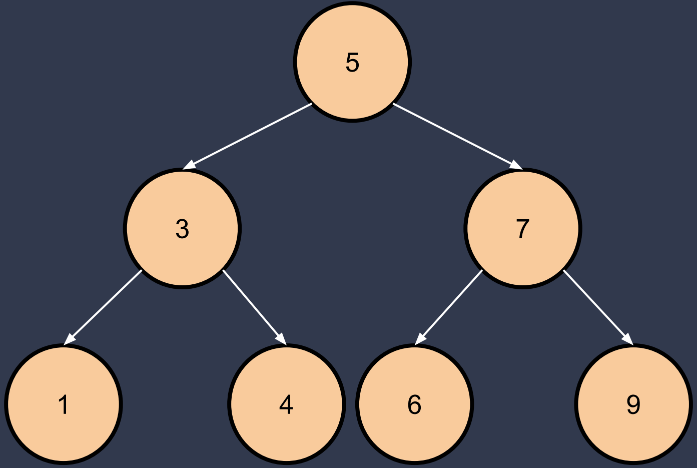
- Create a new `class` called `BinaryNode`.
- The `class` should have the following methods:
  - ***A least one*** constructor that takes an `int` value.
  - A getter/setter for its `int` value.
  - A getter/setter for its left subtree (a `BinaryNode`).
  - A getter/setter for its right subtree (a `BinaryNode`).
  - A `toString()` that returns a String in the format: `"BinaryNode{value=<value>, left=<left>, right=<right>}"`
- Add a `main` method and build the binary tree shown on the left. Print it to standard output.

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



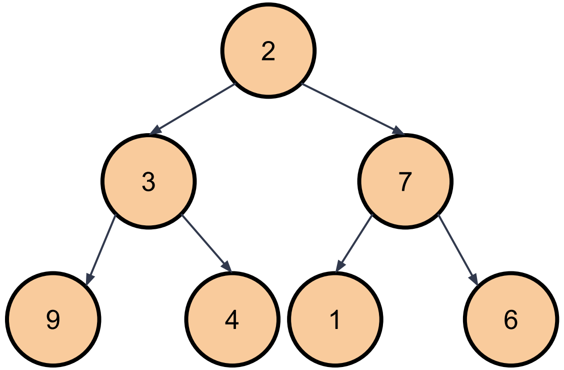
Consider how changing the order in which the nodes are visited would change the string we are building.

There are other kinds of traversal that change the order in which the middle node is visited.

- 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 left. 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***.

6.2 An Infix (in-order) Traversal

It's important that we be able to convert a Binary Tree into string that we can read and understand so that we can examine the contents of the tree as we manipulate it. Let's implement a method to perform an infix traversal of the tree and returns it as a string.



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

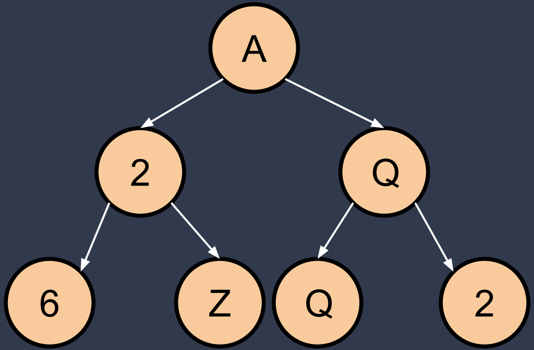
We will discuss that soon!

- Open the `BinaryNode` class and define a method named `"infixTraversal"` that returns a `String`.
  - Create an empty `String`.
  - If the ***left subtree*** is not `null`, call its `infixTraversal` method and add the return value to the string.
  - Then the node should visit ***itself*** - add the `value` to the string followed by a space, e.g. `"5 "`.
  - If the ***right subtree*** is not `null`, call its `infixTraversal` method and add the return value to the string.
  - Return the string!
- Update the `main` method in your `BinaryNode` to call the `infixTraversal` method on the root of 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***.

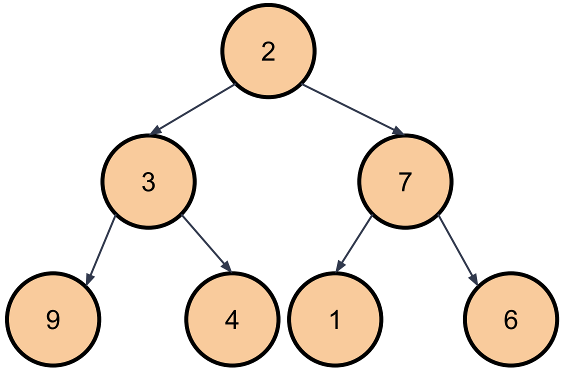


If the target is not in the root node, search one of the subtrees.

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

6.3 Searching a Binary Tree

Because the values in a Binary Tree are not in any particular order, searching means that, if the target doesn't match the value in a node, you may need to search ***both*** the left and right subtrees. Try implementing a search algorithm now. What is the time complexity when searching a binary tree with N nodes?

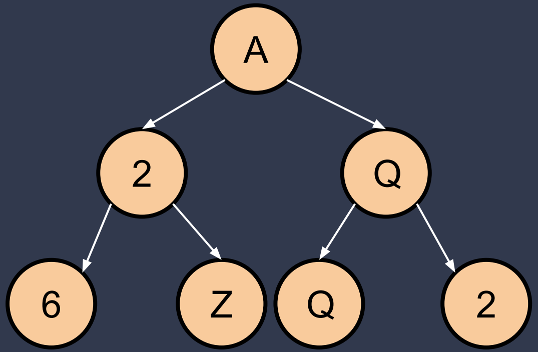


Make sure to search for values in both subtrees as well as values that are not present in the tree.

- Open the `BinaryNode` class and define a new method with the signature: `public boolean search(int target)`
  - If the node's `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 the `main` method to search for several values and print the results.

Search Complexity

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



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

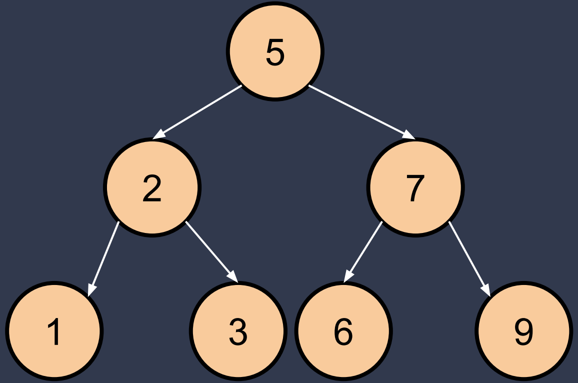
There is no relationship between the data in the nodes.

- 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?

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 that has some extra rules.

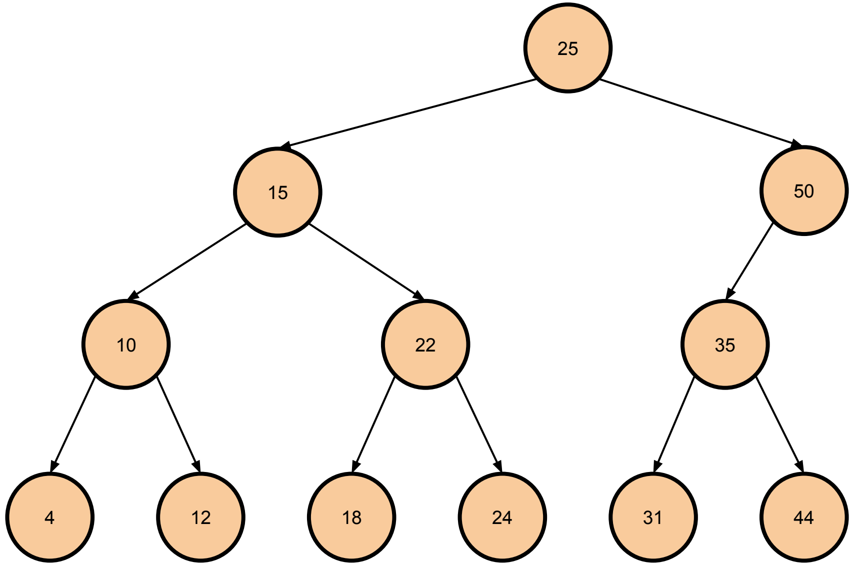


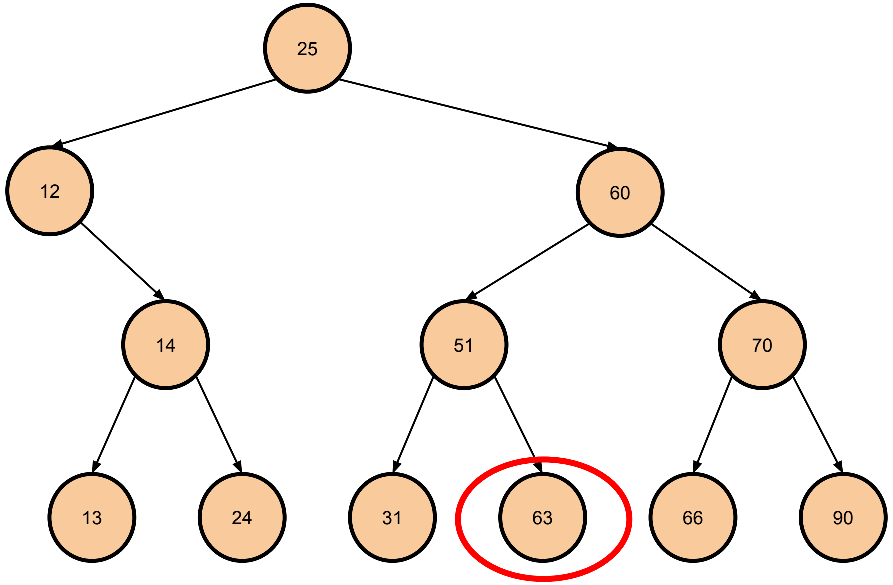
From any node in the tree, the left subtree contains only values that come ***before***...

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

Identifying Binary Search Trees

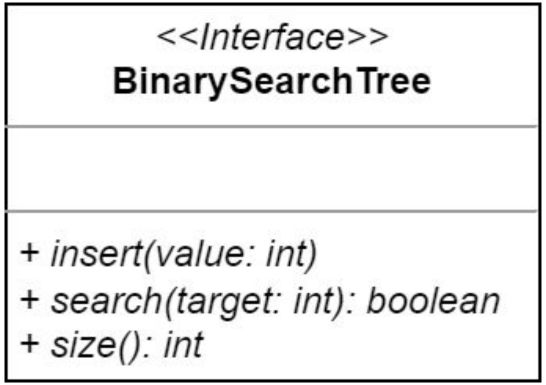
Is this a Binary Search Tree? If not, why not?





6.4 A Binary Search Tree ADT

Remember, an ADT tells us ***what*** a data structure does, but not ***how*** it does it - it ***defines*** behavior without ***implementing*** it. We will need to define an ***abstract data type*** (***ADT***) to represent a ***binary search tree*** (***BST***) using a Java interface.



<<Interface>>
BinarySearchTree

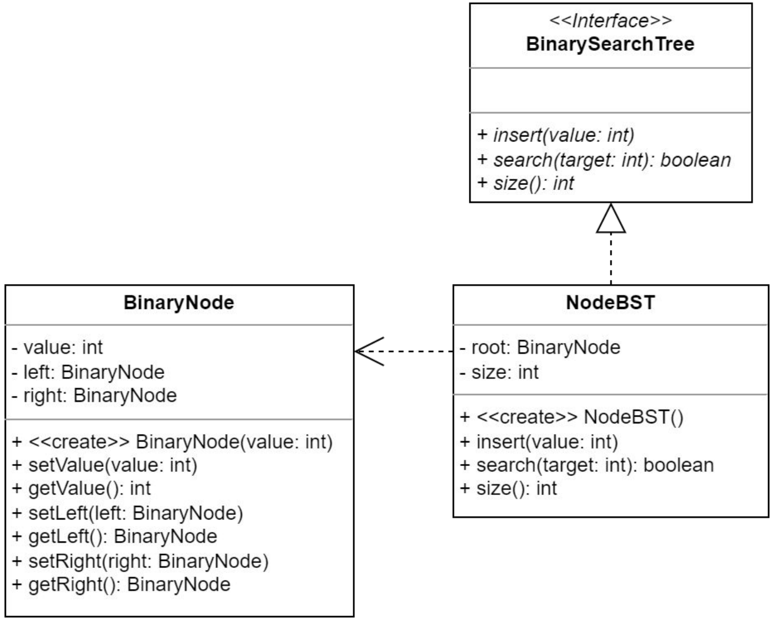
+ *insert(value: int)*
+ *search(target: int): boolean*
+ *size(): int*

For now, the BST will only work with integer values.

- Using the UML to the left, define a new Java interface to represent a binary search tree.

6.5 A Node-Based BST

As you know, most ADTs can be implemented in more than one way. Let's work on a node-based implementation of a BST using the `BinaryNode`!



- Using the UML to the left, begin implementing the `NodeBST` class.
  - Use the constructor to set the root node to `null`.
  - Implement the `size()` method just like we have for most of our data structures.
  - Stub out the other methods for now.
  - The string representation of the `NodeBST` should be an infix traversal of the tree, or `"<empty>"` if the tree is empty.
- Define a `main` method with the appropriate signature and use it to create an empty `NodeBST` and print it to standard output.

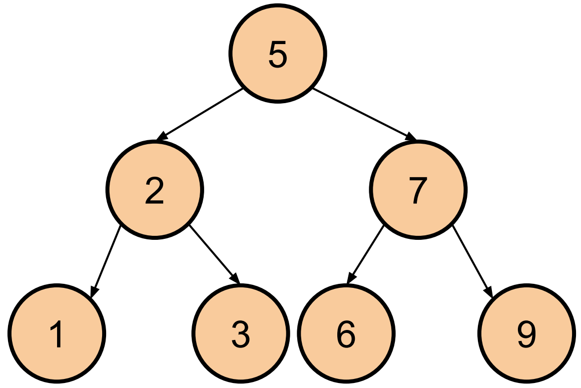
Inserting into the Pokédex

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

6.6 Practice Drawing BSTs

Drawing diagrams can help understand how something works. Let's practice inserting into binary search trees by drawing a few pictures!

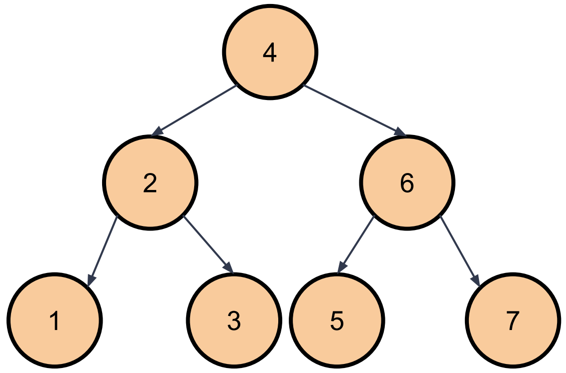


Get out a pen or pencil and some paper and draw a few trees!

- When inserting into a binary search tree, if the tree is empty, the value becomes the root. For each additional value:
  - If the value is less than the root, insert into the left subtree.
  - If the value is greater than the root, insert into the right subtree.
  - Recurse until an empty subtree is found; the new value becomes the subtree.
- Try it now with the following sets of values:
  - 5, 7, 6, 2, 9, 1, 3
  - 5, 4, 6, 3, 7, 2, 8, 1, 9
  - 10, 15, 20, 25, 30, 35, 40
  - 10, 9, 8, 7, 6, 5, 4, 3, 2, 1

6.7 Inserting into a Binary Search Tree

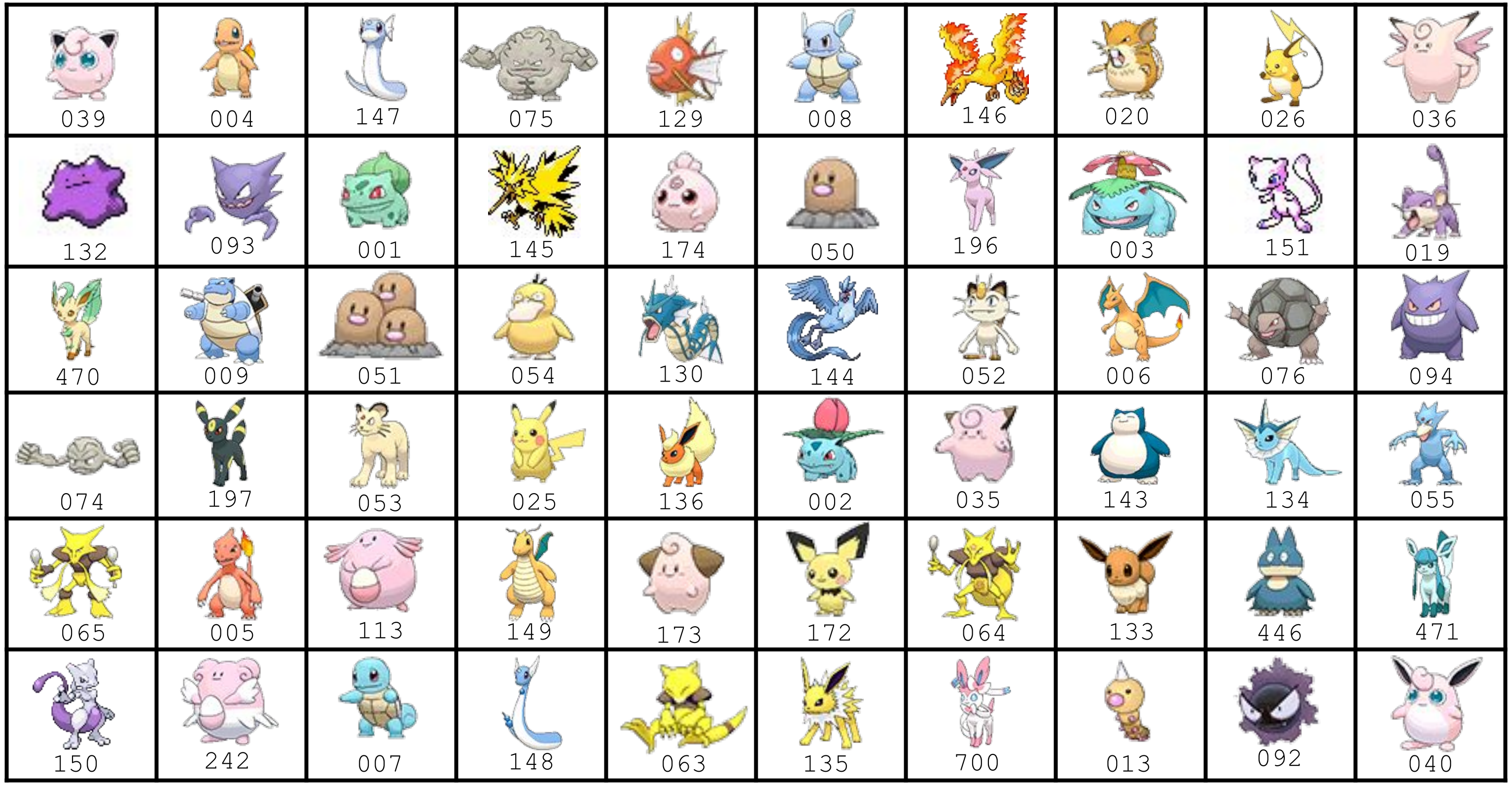
Building a BST requires inserting new values into a node's *left* subtree if they are less than the the node's value, and into the *right* subtree otherwise. This continues until an empty subtree is found. Try implementing binary insert now!



Use `main` to build the tree depicted above and print it to standard output.

- Open the `NodeBST` and define a private helper function named `binaryInsert` that declares a parameter for a `BinaryNode` `node` and an `int` `value` to be inserted.
    - If the `value` is ***less than*** the `node`'s value:
        - If the left subtree is empty, make a new `BinaryNode` with the `value` and make it the left subtree.
        - Otherwise, use recursion to insert the `value` into the left subtree.
    - If the `value` is ***greater than*** the `node`'s value:
        - If the right subtree is empty, make a new `BinaryNode` with the `value` and make it the right subtree.
        - Otherwise, use recursion to insert the `value` into the right subtree.
- Navigate to the insert method.
    - If the `root` node is `null`, create it using the `value`.
    - Otherwise, call `binaryInsert` with the `root` and the `value`.
    - Don't forget to increment `size`!

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

6.8 Building Your Pokédex

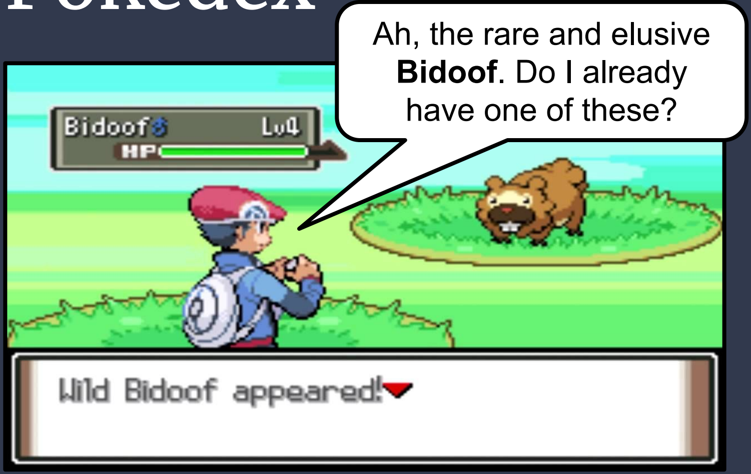
We'll be using a Binary Search Tree (BST) inside of a Pokédex that stores the number of any Pokémon that we encounter in the wild. As each Pokémon is encountered, its number will be inserted into the BST. When we are finished, we can print all of the stored Pokémon in order!

```
Pokédex
-------------------------------------
- bst: BinarySearchTree
-------------------------------------
+ <<create>> Pokédex()
+ addPokémon(number: int): void
+ toString(): String
```

- Use the UML to the left to implement a `Pokédex` class that uses a BST to provide the following behavior:
  - A Pokédex is constructed empty. How should you represent an empty BST?
  - `addPokémon` - inserts the Pokémon's number into the Pokédex.
  - `toString` - returns all of the Pokémon contained in the Pokédex in order.
- Define a `main` method.
  - Create an empty `Pokédex` and add 15 of the Pokémon from the previous table *in the order that they appear in the table.*
  - Print the `Pokédex` to standard output.

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`.

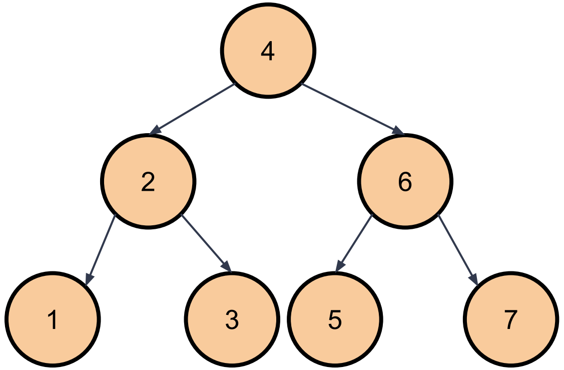

Ah, the rare and elusive **Bidoof**. Do I already have one of these?
Bidoof♂ Lv4
HP
Wild Bidoof appeared!

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 search much more efficiently.

6.9 Searching a BST

Values in a Binary Search Tree (BST) are organized such that the left subtree only contains values that are less than a node's value, and the right subtree only contains values that are greater than the node's value. Because of this, a search of the BST is able to make a decision about which subtree to search (just like *Binary Search*). Let's implement a binary search now!

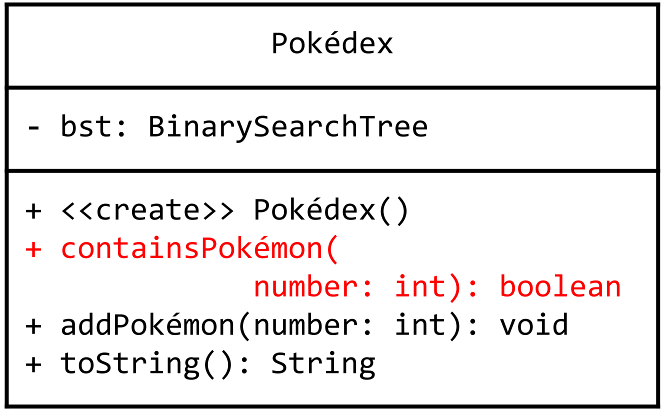


A search through a BST will be able to make decisions about which subtrees to search.

- Begin by creating a private helper function named `binarySearch` that declares parameters for a `BinaryNode node` and an `int target`.
  - If the `target` is equal to the `node`'s value, return `true`.
  - If it is less than the `node`'s value, recursively search the left subtree.
  - Otherwise, recursively search the right subtree.
  - If you find an empty subtree, return `false`.
- Implement the `search` method by calling the `binarySearch` method with the `root` and the `target` and return the result.
- Use the `main` method to search for at least 4 values, 2 of which are not in the tree.

6.10 Searching Your Pokédex

It's not enough to simply add Pokémon to the Pokédex as they are encountered - the Pokémon Trainer also wants to be able to quickly determine whether or not a Pokémon is already in the Pokédex so that they can decide whether or not it is worth taking the time to try and catch it!

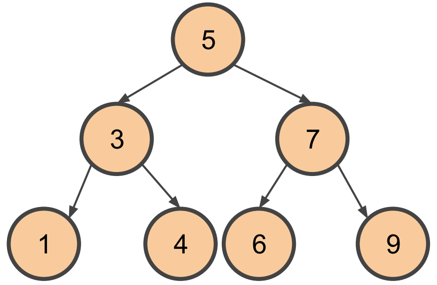


```
Pokédex
- bst: BinarySearchTree
+ <<create>> Pokédex()
+ containsPokémon(
            number: int): boolean
+ addPokémon(number: int): void
+ toString(): String
```

- Enhance the `Pokédex` class so that it also provides the following behavior:
  - `containsPokémon` - returns `true` if the Pokémon's number is already in the Pokédex, and `false` otherwise.
  - `addPokémon` - update the method so that it should only add a Pokémon to the Pokédex if it is not already present.
- Use the `main` method to search for several Pokémon and print the results to standard output.
  - Make sure to search for Pokémon that are in the Pokédex as well as those that are not.
  - What happens if you try to add the same Pokémon twice?

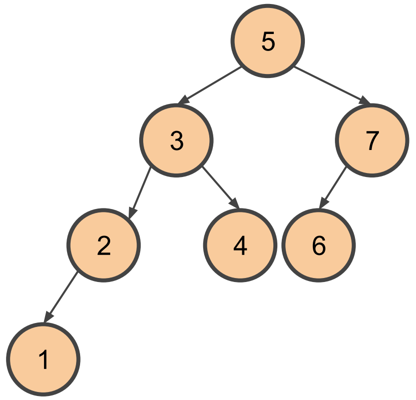
Balance

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



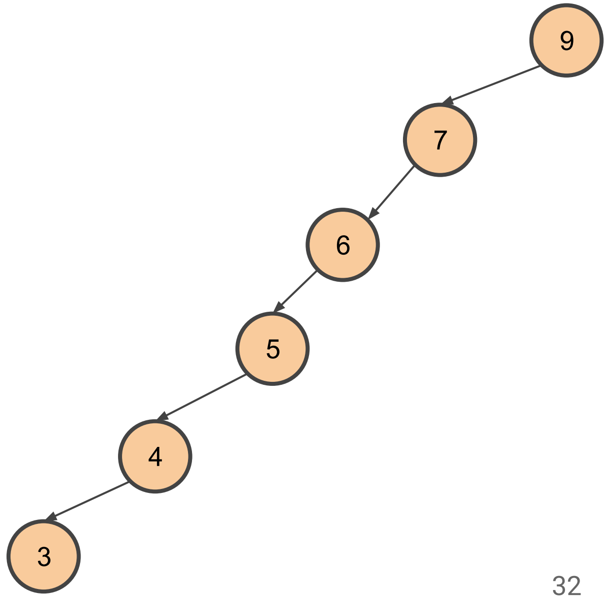
This means that there are approximately the same number of nodes in both halves of the tree ***at every level***.

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



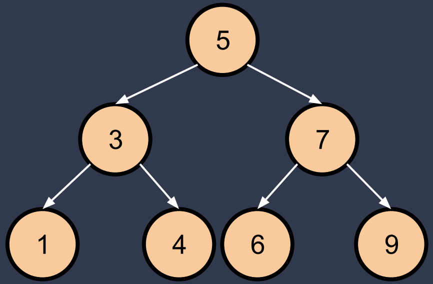
Unbalanced trees are more ***branchy*** than ***bushy***.

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

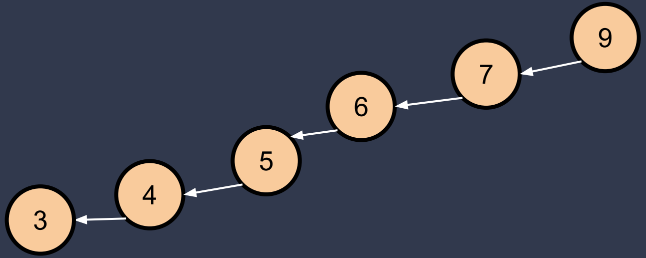


Complexity of Searching

- 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 complexity is therefore $O(\log_2 N)$ (just like *binary search*).



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

Determining Order

SORT THESE ALIEN SYMBOLS, HUMAN, OR IT'S BACK TO THE PROBE CHAMBER!
NOT THE PROBE CHAMBER?!

It's easy enough to see that two symbols ***aren't equal***, but how to tell ***which comes first***?

- Sorting ***numbers*** is easy.
  - You either sort them in ***ascending*** order from smallest to largest, or ***descending*** order from largest to smallest.
- Sorting strings is also easy. Sort of.
  - Most humans like to sort strings ***alphabetically***., e.g. "apple", "Cat", "monkey", "Zoo".
  - Most computers like to sort them ***lexicographically***, which is in ***ascending*** order by ***ASCII*** value, e.g. "Cat", "Zoo", "apple", "monkey".
- But what about sorting other types of things?
  - Like fruits, or dogs, or goats, or trolls?
  - Or the ***symbols in an alien language***?
- We already know how to compare arbitrary objects for ***equality***.
  - We use the `equals(Object)` method that is inherited from the `Object` class.
  - The default implementation compares the ***identity*** of the two objects, but we can override that.
- That's great, but knowing whether or not two objects are ***equal*** to each other tells us nothing about which one comes ***first*** in sorted order.

Comparable

- The easiest way to sort values of some type is if they are ***naturally comparable to each other***.
  - The type defines its own ***natural order***.
  - That is to say, given two values of the same type, you can compare ***one*** to the ***other*** to determine which comes first.
- In Java, a type is made ***naturally comparable*** by implementing the `java.lang.Comparable` interface.
  - `Comparable` is a ***generic interface*** that defines a type parameter `T`. This is used to specify the type to which the class can be compared.
  - A class usually specifies ***its own type*** in place of the type parameter.
- The `Comparable` interface defines a single method: `public int compareTo(T other)`
- Given two instances of a comparable class `A` and `B`, `A.compareTo(B)` returns:
  - `0` if `A` and `B` are ***equal*** to each other.
  - `< 0` if `A` ***comes before*** `B` in sorted order.
  - `> 0` if `A` ***comes after*** `B` in sorted order.

CAN YOU TELL ME WHICH OF YOU GUYS COMES FIRST IN SORTED ORDER?
HELP
HE DOES!
I DO!

If a class implements `Comparable`, then you can ***directly compare*** two instances of the class using the `compareTo(T)` method to see which comes first in ***sorted order***.

A Closer Look at Comparable

```java
public class AlienAlphabet implements Comparable<AlienAlphabet> {
    private final int sequenceNumber;
    private final String symbol;

    public AlienAlphabet(int sequenceNumber, String symbol) {
        this.sequenceNumber = sequenceNumber;
        this.symbol = symbol;
    }

    @Override
    public int compareTo(AlienAlphabet other) {
        // sort by sequence number
        return this.sequenceNumber - other.sequenceNumber;
    }
}
```

A class implements `Comparable` to define its own ***natural order***, which determines how instances of the class will be arranged in ***sorted order***.

`Comparable` is a ***generic type***. Its ***type parameter*** determines the type of value that can be passed into the `compareTo(T o)` method.

Most of the time, a class wants to be compared with ***its own type***, and so it will specify its own type in place of the type parameter.

When `A.compareTo(B)` is called on an instance it should return `<0` if `a` ***comes before*** `b` in sorted order, `0` if `A` ***and*** `B` ***are equal***, and `>0` if `A` ***comes after*** `B`.

6.11 Comparable Pokémon

Right now we are representing a Pokémon using its number. But there is a lot more to Pokémon than a mere number - each Pokémon should also at least have a name! Let's create a new class to represent Pokémon and implement it so that Pokémon are naturally comparable by number.

```
public class Alien implements Comparable<Alien> {
    private final int seqNumber;
    private final String symbol;

    public Alien(int seqNumber, String symbol) {
        this.seqNumber = seqNumber;
        this.symbol = symbol;
    }

    @Override
    public int compareTo(Alien other) {
        // sort by sequence number
        return this.seqNumber - other.seqNumber;
    }
}
```

Up until now we have been representing Pokémon using integers.

Let's create a class that defines its own ***natural order*** instead.

- Create a new Java class named "`Pokémon`".
  - Add fields and a constructor for the Pokémon's `number` and `name`.
  - ***Override*** `toString()` to return a `String` in the format `"#: <name>"`, e.g. `"7: Squirtle"`.
  - ***Implement*** the `Comparable` interface.
    - The class should specify ***its own type*** for the generic type parameter.
    - Compare two Pokémon using their numbers in ***ascending order***.
    - Remember, `poke1.compareTo(poke2)` should return an integer that is `<0` if `poke1` is first, `0` if they are equal, and `>0` if `poke1` is second.
- Add a `main` method with the appropriate signature.
  - Create a Java `List` that contains at least 3 different Pokémon. Make sure to add them out of order.
  - Call the `Collections.sort(List)` method to sort the list and then print it to standard output.

Comparators

WE HAVE NO IDEA.
CAN YOU TELL ME WHICH OF YOU GUYS COMES FIRST IN SORTED ORDER?
HELP
I CAN HELP!

A `Comparator` is a ***separate class*** that can compare instances of ***some other class***. It can help sort things that ***do not*** have a natural order ***or*** sort them in a different way if they do.

- If a class implements `Comparable`, it defines its own ***natural order*** and can be sorted.
  - By using `Collections.sort(List)` for example.
- But what do you do if a class ***doesn't*** implement `Comparable` and you would still like to arrange objects of the class into some order?
- Or what if the class ***does*** implement `Comparable` but you want to arrange objects into ***some other order***?
- Java provides another interface that allows you to create a ***helper class*** that can be used to arrange objects of ***some other class***: `java.util.Comparator`
  - It is a ***generic type*** that defines a single method, `compare(T a, T b)`, that compares two instances of some ***other class***.
  - It follows the same rules as `Comparator` and returns an integer `<0` if `a` comes ***before*** `b` in sorted order, `0` if `a` and `b` are ***equal***, and `>0` if `a` comes ***after*** `b`.
- A class that implements `Comparator` acts as a helper to sort objects of another type.

A Closer Look at Comparator

```java
public class AlienComparator implements Comparator<AlienAlphabet> {

    @Override
    public int compare(AlienAlphabet o1, AlienAlphabet o2) {
        String sym1 = o1.getSymbol();
        String sym2 = o2.getSymbol();
        return sym1.toLowerCase().compareTo(sym2.toLowerCase());
    }
}
```

A class that implements `Comparator` defines a sort order for ***some other class***, which is specified for the `Comparator`'s type parameter.

A `Comparator` is needed when the other class does not define a natural order at all ***or*** to sort instances of the class in some other way.

The other class is specified in place of the type parameter, and determines the parameter types for the `compare(T o1, T o2)` method.

When `C.compare(A, B)` is called on the `Comparator` it should return `<0` if `A` ***comes before*** `B` in sorted order, `0` if `A` ***and*** `B` ***are equal***, and `>0` if `A` ***comes after*** `B`.

It's worth noting that `String`, like many other Java types, implements `Comparable` to define ***its own natural order***. This means that you can ***use*** its `compareTo(String)` method in your own classes.

6.12 Pokémon Comparator

Pokémon define a *natural order* that sorts them in ascending order by number. That's great! But what if we wanted to sort them them in some other way, e.g. alphabetically by name? We'd need a separate `Comparator` class that will help Java's sorting algorithms to arrange the Pokémon in a different way, *other than* the natural sorting that we defined in the previous activity.

```
public class AlienComp
        implements Comparator<Alien> {

    @Override
    public int compare(Alien o1, Alien o2) {
        String sym1 = o1.getSymbol();
        String sym2 = o2.getSymbol();
        return sym1.toLowerCase().
                compareTo(sym2.toLowerCase());
    }
}
```

It's great that Pokémon are naturally sortable by number, but what if we wanted to sort them in another way? A `Comparator` can help with that!

- Create a new Java class named "`PokémonComparator`".
  - ***Implement*** the `Comparator` interface.
    - Specify `Pokémon` as the comparable type. Both parameters to the `compare(T o1, T o2)` should therefore also be `Pokémon`.
    - Compare two Pokémon ***alphabetically by name***.
    - Remember, `comp.compare(poke1, poke2)` should return an integer that is `<0` if `poke1` is first, `0` if they are equal, and `>0` if `poke1` is second.
- Define a `main` method with the appropriate signature.
  - Create a Java `List` that contains at least 3 different `Pokémon`. Make sure to add them out of order.
  - Call the static `Collections.sort(List, Comparator)` method to use the `PokémonComparator` to sort the list and then print it to standard output.

Comparable vs. Comparator

I am an interface implemented by a class that wants to **define its own natural order**.

I am an interface implemented by a **separate class** that wants to define an order for some **other** class.

I am called on one object of the class. Another object is passed in as a parameter, e.g. `a.compareTo(b)`.

This is either because the other class's natural order **doesn't work** for the problem that I am trying to solve or the class **doesn't have a natural order** at all.

We are both used for sorting.

The first object compares **itself** to the second and returns <0, 0, or >0 if the first is before, equal to, or after the second.

Two instances of the class I can compare are passed into my compare method, e.g. `me.compare(a, b)`.

6.13 Comparable Fruit

It can take a long time to understand the uses of Comparable and Comparator and how they are different from each other. The best way to get a handle on it is to practice implementing both a few times to solve different problems. Let's try another example now!

```java
public class Alien implements Comparable<Alien> {
    private final int seqNumber;
    private final String symbol;

    public Alien(int seqNumber, String symbol) {
        this.seqNumber = seqNumber;
        this.symbol = symbol;
    }

    @Override
    public int compareTo(Alien other) {
        // sort by sequence number
        return this.seqNumber - other.seqNumber;
    }
}
```

- Create a new Java class named "`Fruit`".
    - Every fruit has a `name` and a `price`.
    - ***Override*** `toString()` to return a `String` in the format `"<name> $<price>"`, e.g. `"Apple $1.50"`.
    - ***Implement*** the `Comparable` interface.
        - Implement the `compareTo` method so that two fruits are compared by name. If two fruits have the same name, compare them by price.
        - Remember, `fruit1.compareTo(fruit2)` should return an integer that is <0 if `fruit1` is first, 0 if they are equal, and >0 if `fruit1` is second.
- Add a `main` method with the appropriate signature.
    - Create a `List` with at least 3 fruits. Print it, sort it, and then print it again.

6.14 A Fruit Comparator

If a class does not implement `Comparator` to define its own natural sorting order *or* if you need to sort objects of the class in some other way, you will need to create a `Comparator` to help sort them. Practice now by defining a `Comparator` that sorts fruits by price.

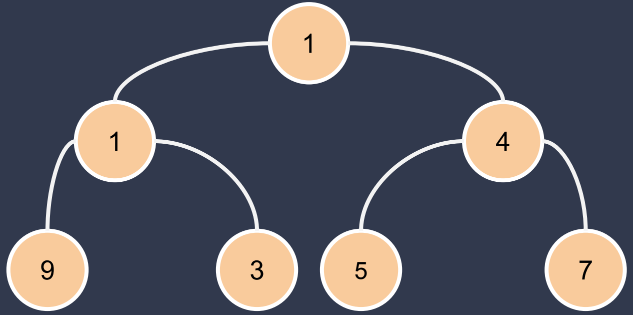
```
public class AlienComp
        implements Comparator<Alien> {

    @Override
    public int compare(Alien o1, Alien o2) {
        String sym1 = o1.getSymbol();
        String sym2 = o2.getSymbol();
        return sym1.toLowerCase().
                compareTo(sym2.toLowerCase());
    }
}
```

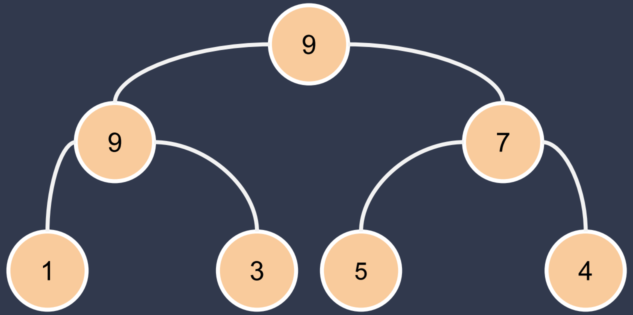
- Create a new Java class named "`FruitComparator`".
    - ***Implement*** the `Comparator` interface.
        - Specify `Fruit` as the comparable type. Both parameters to the `compare(T o1, T o2)` should therefore also be `Fruit`.
        - Compare two Fruit by ***price*** and then ***name***.
        - Remember, `comp.compare(fruit1, fruit2)` should return an integer that is `<0` if `fruit1` is first, `0` if they are equal, and `>0` if `fruit1` is second.
- Add a `main` method with the appropriate signature.
    - Create a Java `List` that contains at least 3 different fruit. Print if, use the `FruitComparator` to sort it, and then print it again.

Heaps

- A ***heap*** is a ***another kind*** of binary tree that is similar to a binary search tree.
- Each node in the heap has a value that is an ***equal or lower priority*** than the value of its ***parent***.
  - The exception of course is the root, which has no parent and always contains the ***highest priority*** value.
- The ***horizontal*** order of the nodes (i.e. from left to right) is ***arbitrary***.
  - In other words, there is no horizontal relationship between nodes; the left ***or*** the right child may be higher priority.
- ***Priority*** is determined by the specific problem statement.
  - It is up to the programmer to decide what "highest priority" means, especially when dealing with non-numerical data.
- A heap can be organized in one of two ways.
  - A ***min-heap*** organizes values from ***smallest to largest***, and so the root will always have the ***smallest value*** in the heap.
  - A ***max-heap*** organizes values from ***largest to smallest***, and so the root will always have the ***largest*** value in the heap.



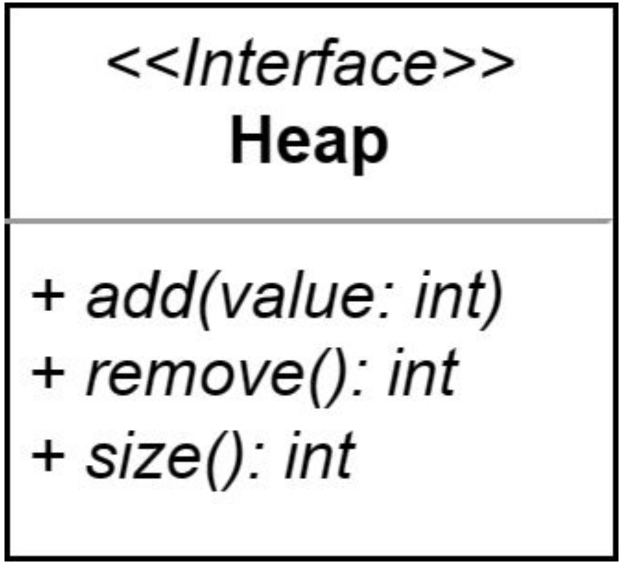
An example of a ***min-heap***. Notice that there is no ***horizontal*** relationship between nodes.



An example of a ***max-heap***.

6.15 The Heap Abstract Data Type (ADT)

There is more than one possible way to implement a heap! Like all Abstract Data Types (ADTs), the definition of a *heap* tells us *what* a heap can do, but not *how* it does it. Let's define a Java interface that defines the essential behavior that a heap provides.

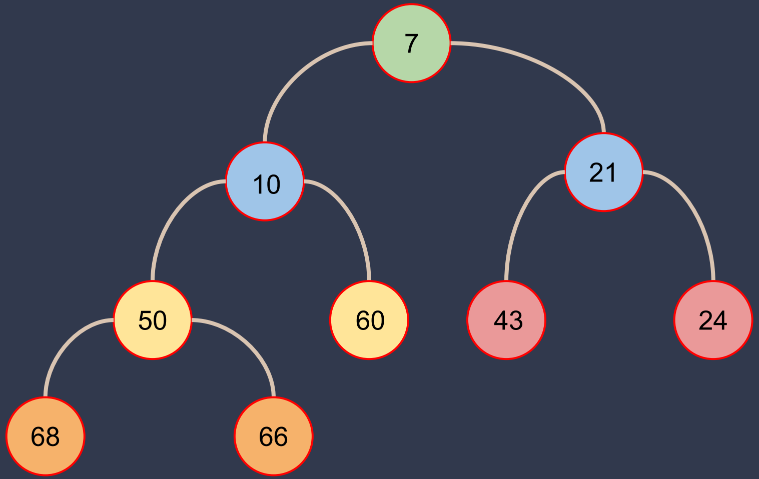


A heap only provides basic operations to `add` and `remove` values.

- Create a "`heaps`" package, and then use the UML diagram to the left to implement a Heap abstract data type.
- When you are finished, verify that you do not have any problems in your project and push your code.

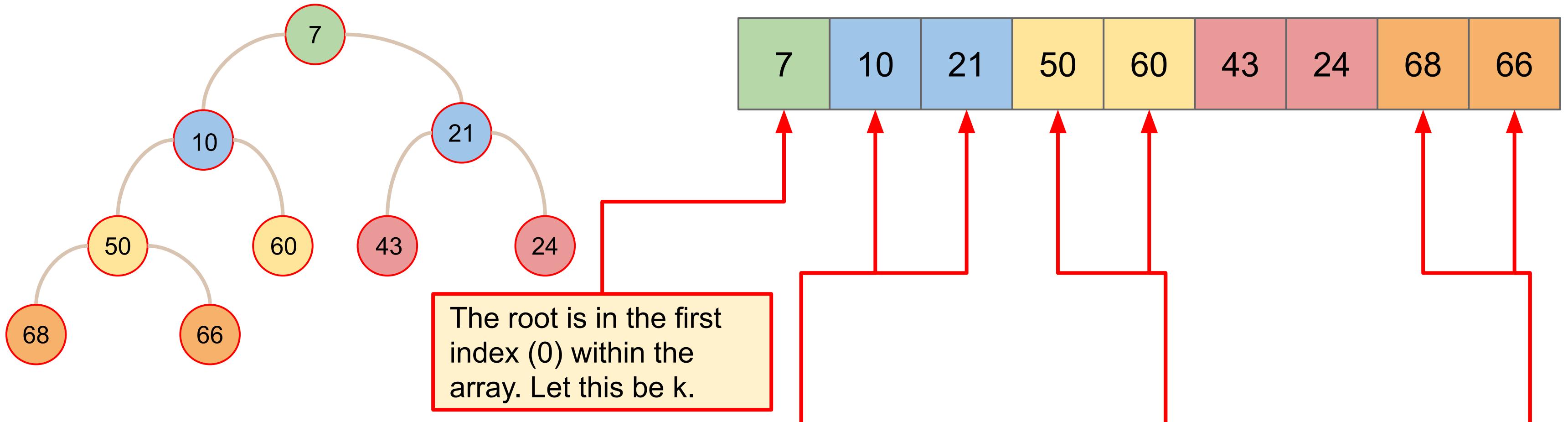
A Heap Overlay

- Like many data structures, a heap can be implemented in more than one way.
  - One obvious way would be to try to implement a heap using ***binary nodes***.
  - But perhaps there is a way to implement a heap using an ***array***?
- The heap can be created using an ***overlay*** on an array by adhering to the following rules assuming an array of size ***N*** with indexes ranging from ***0 to N-1***.
  - The root of the heap is at ***index 0***.
  - Let ***k*** be the index of a node in the tree where ***0 ≤ k < N***.
  - The ***left*** child of node *k* is located at index ***2k+1***.
  - The ***right*** child of node *k* is located at index ***2k+2***.
- For example:
  - If ***k*** is `0`, its left child is at ***2*0+1 = 1***, and its right child is at ***2*0+2 = 2***.
  - If ***k*** is `11`, it's left child is at ***2*11+1 = 23***, and its right child is at ***2*11+2 = 24***.
- Like other array-based data structures, one challenge is ***growing the array*** then it has filled up with values.
  - This is easier than some other data structures, because the values can be copied into the same indexes in a larger array.



Let's look at an example of how this heap would be overlaid onto an array.

An Array-Based Heap



The root is in the first index (0) within the array. Let this be k.

The children of the root are in index *2k+1* (1) and *2k+2* (2).

The children of the left child are ar *2k+1* (3) and *2k+2* (4).

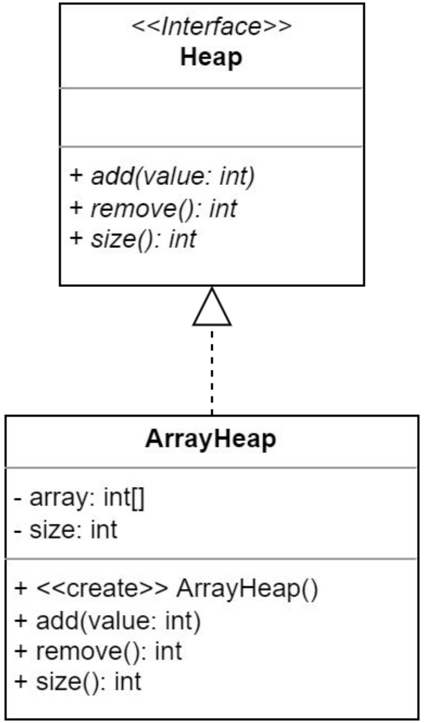
The children of the left child are ar *2k+1* (7) and *2k+2* (8).

Let *k* now be the index of the left child: 1.

Now let k be the index of *this* left child: 3.

6.16 An Array-Based Heap

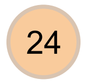
Unlike the other trees that we have implemented in this unit, we will not implement the heap using nodes. Instead, we will implement our first heap as an ***overlay*** on top of an array.



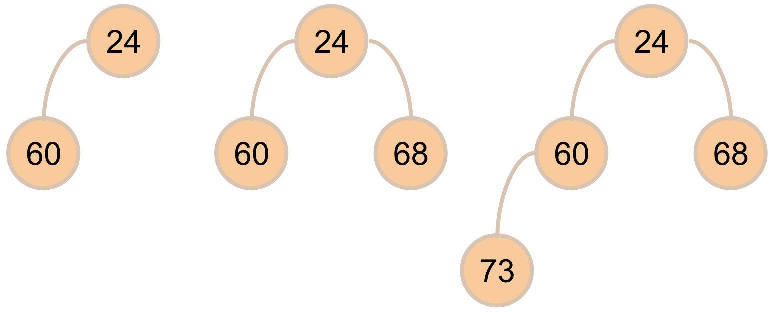
- Using the UML to the left as a guide, begin your array-based implementation of a heap by creating a new Java class named "`ArrayHeap`" inside the `heaps` package.
    - In the ***constructor***:
        - Initialize the `array` to a capacity of `3`.
        - Set the `size` to `0`.
    - Add a `toString()` in the format `"<size>, <array>"`
        - For example: `"0, [0, 0, 0]"`
        - ***Hint:*** use `Arrays.toString()`.
    - Implement the `size()` method.
    - Stub out the remaining methods.
- Add a `main` method to the class.
    - Create an instance of the `ArrayHeap` and print its `size` to standard output.

Adding to a Heap

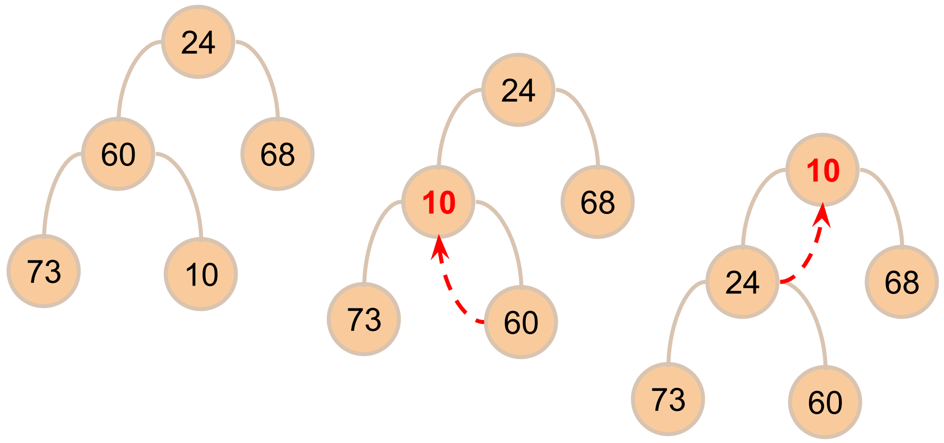
The first node added to the heap becomes the ***root***.



Each subsequent node is added to the ***leftmost*** open position in the ***bottom*** level of the tree.



If a newly added node has a higher priority than it's parent, the values are ***swapped***.

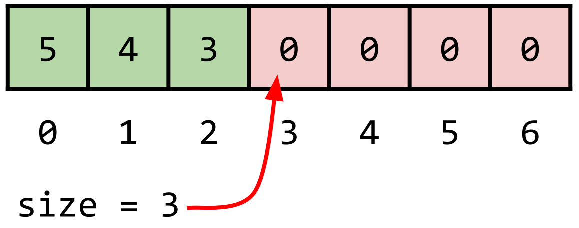


The swaps continue until the value is in the correct priority position. This is called ***sifting up***.

6.17 Adding to the Heap

The first few steps of adding a new value to a heap is exactly the same as appending a value to an `ArrayList`. Let's focus on that part first, before we worry about *sifting up*.

Remember that Java fills `int` arrays with `0`s by default.

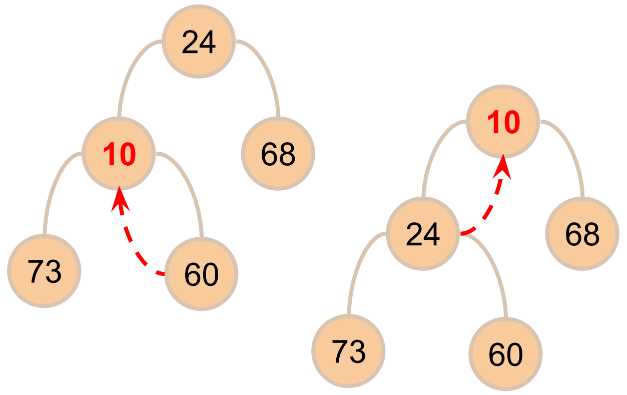


The next value added to this heap will go into the index to which `size` refers (**3**).

- Open the `ArrayHeap` class and navigate to the `add` method.
  - Values are always initially added to ***leftmost-open position*** on the bottom of the heap. The good news is that this is the ***index to which `size` refers***!
  - First, check to see if `size` is equal to the length of the `array`; if so, use `Arrays.copyOf(original, newSize)` to create a bigger copy of the array.
  - Next, put the new value in the correct location in the `array`.
  - Don't forget to ***increment*** `size`!
- Update your `main` method.
  - Verify that you are able to add ***4 or more*** values to the heap.
  - Print the the heap. You should notice that the values are in the array, but not in the correct order.

6.18 Sifting Up

Unless we are very lucky, newly added values in a heap are not likely to be in the correct position. The new values will need to be *sifted up* into the heap until they are in the correct position, i.e. a higher priority than *both* of the children. Let's take a crack at implementing the algorithm now!

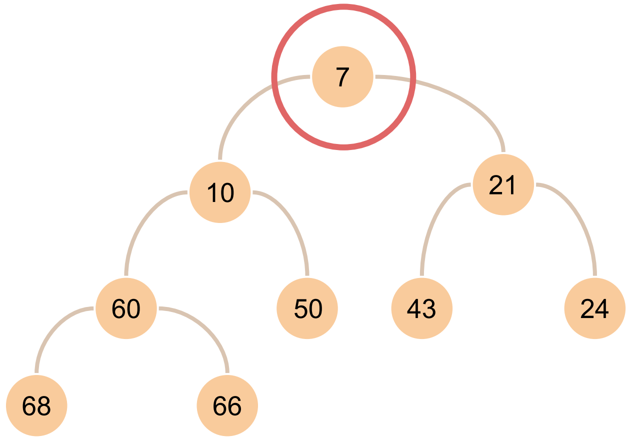


A value added to the bottom of the heap may not be in the correct place and needs to ***sift up*** to the correct location.

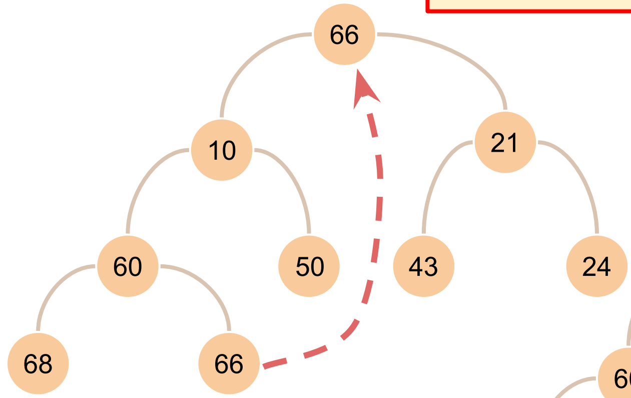
- Sifting requires that we ***swap*** values at two different indexes in an array. Open the `ArrayHeap` class and define a new `private` helper method named "`swap`" that declares parameters for two indexes `a` and `b`.
    - If `a` ***and*** `b` ***are not equal***, ***swap*** the values at the specified indexes.
- Next, implement the algorithm to sift up in your add method ***before*** you increment `size`:
    - Set the `child` index equal to `size`.
    - Set the `parent` index equal to `(child - 1) / 2`.
    - As long as `array[parent] > array[child]`:
        - ***Swap*** the values.
        - Set `child` equal to `parent`.
        - Set `parent` equal to `(child - 1) / 2`.
- ***Run*** `main` to verify that the values are being sifted up.

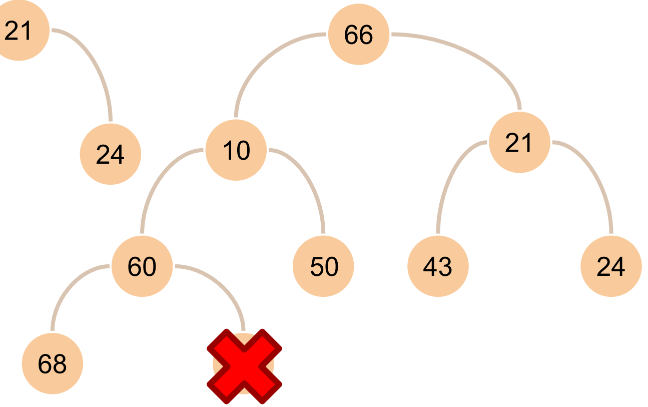
Removing From the Heap

The root always contains the ***highest priority*** value, and so the root is always the value ***removed*** from the heap.



Then, ***copy*** the value from the ***rightmost*** node in the ***bottom*** level to the root...

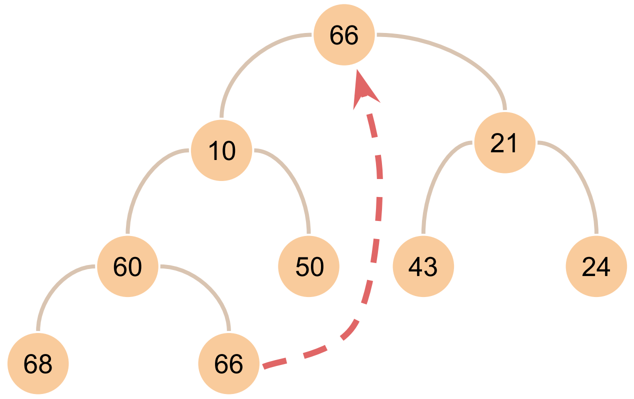




...and then ***remove*** that rightmost node from the bottom level (set it to 0).

6.19 Removing From a Heap

The highest priority value is always the next to be removed from the heap. Let's begin implementing the algorithm now by focusing on the first few steps, which are fairly straightforward.



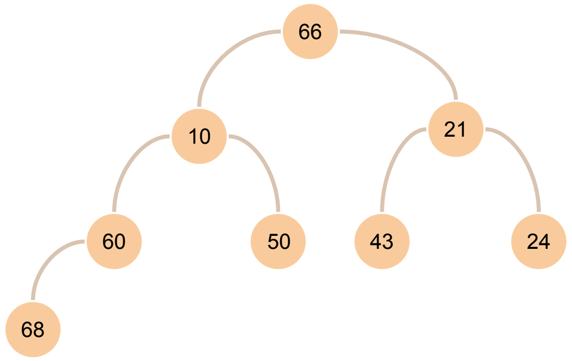
Once finished, you will be doing ***most*** of the work of removing.

The next step is to ***sift*** the value swapped into index 0 ***down*** into the correct position in the heap.

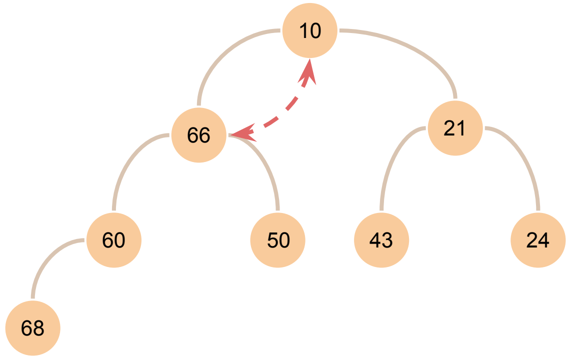
- Open the `ArrayHeap` class and navigate to the `remove()` method. The first few steps are fairly straightforward.
  - ***Save*** the value at index `0` in a temporary variable.
  - ***Swap*** the value in rightmost position in the bottom of the tree into index 0.
    - ***Hint***: it will be at the last index that a value was added to the heap.
    - Be sure to set the value at that index to `0` after swapping.
  - Don't forget to ***decrement*** `size`!
  - ***Return*** the temporary value.
- Update the `main` to ***remove and print*** a few values (and the heap).
  - The ***first*** value should be the highest priority (i.e. lowest value), but the next value(s) that you remove probably will not be in the correct priority order.
  - This is because the value swapped into index `0` is not yet ***sifted down***.

Removing From a Heap: Part 2

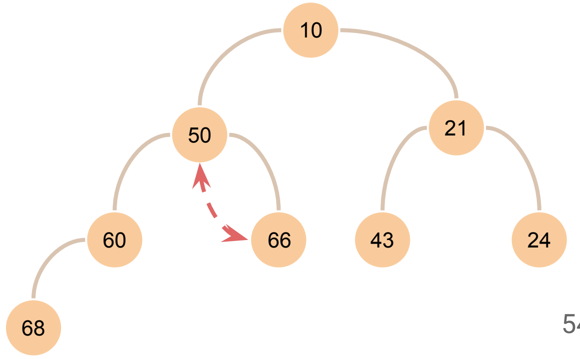
Now the value in the root needs to be ***sifted down*** to its proper position in the heap.



Compare its value to both children and, if it is not ***higher priority*** than both, ***swap*** it's value with the ***higher priority*** of the two.

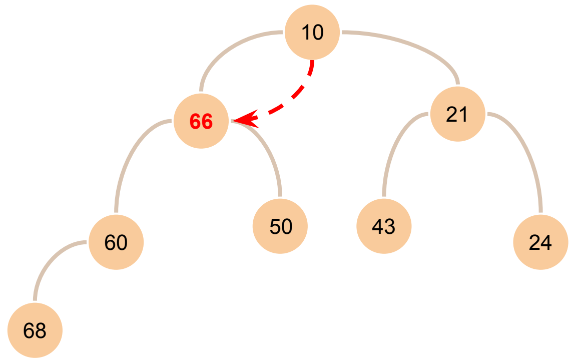


Repeat this until the value is lower priority than both of its children.



6.20 Sifting Down

At this point, the value that *was* in the rightmost bottom node in the tree is now at the root, meaning that it will need to be *sifted down* until it is in the correct priority position. This is more complicated than sifting up because it will need to be compared to *both* of its children and swapped with the higher priority of the two!



Once sift down is working, your heap implementation will be complete.

- The sifting algorithm is a little more complex than sifting up.
  - The parent value needs to be compared with ***both*** of its children to decide which one to swap with (if either).
  - We also need to make sure not to swap into an index that is ***outside*** of the heap (e.g. into an unused part of the array).
- Open `ArrayHeap` and navigate to the remove method. Implement the ***sift down*** algorithm before the method returns.
  - Start the `parent` at index at `0` (the value you swapped).
  - As long as `parent` is less than `size`:
    - Compare the parent to ***both*** of its children (i.e. ***2k+1***, ***2k+2***).
      - Make sure that they exist!
    - If the `parent` is not higher priority than ***both***:
      - Swap it with the higher priority of the two.
      - Set parent to the swap index.
    - Otherwise, break.

Heap Complexity

Q: What is the worst case time complexity for ***adding*** a new value to a heap?

A: We are using the `size` of the heap to find the correct location, which means that adding the value to the array is **O(C)**.

In the worst case a value added to the bottom of the tree is ***sifted up*** all the way to the root of the tree. What is the complexity of that operation?

The maximum number of swaps is determined by the height of the tree, which is $\log_2 N$. Therefore the worst case time complexity is also **$O(\log_2 N)$**.

Q: What is the worst case time complexity for ***removing*** a value from a heap?

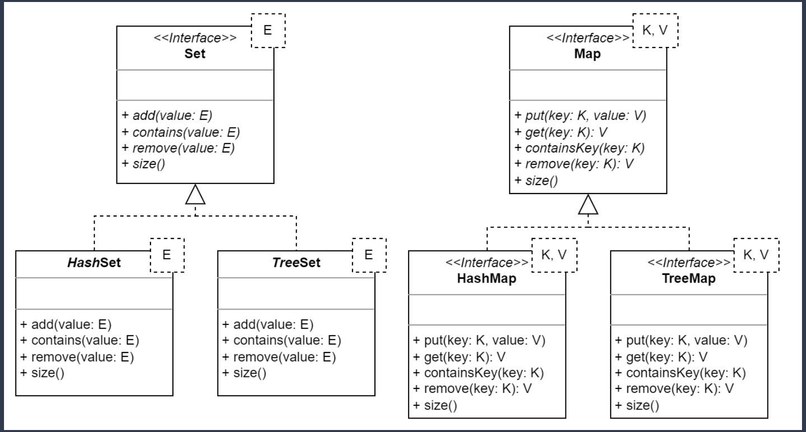
A: We use the `size` of the heap to find the rightmost bottom value in the tree, which is an **O(C)** operation.

In the worst case the value copied into the root must be ***sifted down*** all the way back to the bottom of the tree, which is again the height of the tree: $\log_2 N$.

The worst case complexity of removing is therefore **$O(\log_2 N)$**.

Tree-Based Data Structures: `TreeSet` and `TreeMap`

- So far in this unit, we have examined several different kinds of ***binary trees***.
  - A ***binary tree*** is one in which each node has a ***left subtree*** and a ***right subtree***.
  - A ***binary search tree*** is a binary tree that guarantees that all of the values in a node's left subtree are ***less than*** the node's value, and all of the values in the right subtree are ***greater than***.
  - A ***heap*** is a binary tree that guarantees that the values in each node's left and right subtrees are a ***lower priority*** than the node's value.
- We have also discussed different ways of implementing each kind of tree, e.g. using ***nodes*** or an ***overlay on an array***.
- But binary trees can also be used to implement other data structures!
- The ***TreeSet*** in Java is an implementation of the ***Set*** abstract data type that maintains its elements in sorted order.
- The ***TreeMap*** in Java is an implementation of the ***Map*** abstract data type that maintains its ***keys*** in sorted order.
- ***Both*** data structures require that the elements implement `Comparable` or that a `Comparator` is provided to help with the sorting.



The `TreeSet` and `TreeMap` are alternate implementations of abstract data types with which you are already familiar. They provide all of the same operations.

The fundamental difference is that both keep the values that they store ***in sorted order***. This makes them perform slightly ***slower***.

A Closer Look at the `TreeSet`

```java
Set<String> strings = new TreeSet<>();
strings.add("z");
strings.add("y");
strings.add("x");

for(String s : strings) {
    System.out.println(s);
}
```

A Closer Look at the TreeSet

TreeSet is an implementation of the same abstract data type as HashSet.

A TreeSet provides all of the same operations as a HashSet, and also guarantees that the elements are unique…

The major difference that you will notice is that, when iterating over the elements, they are in ***sorted order***.

```java
Set<String> strings = new TreeSet<>();
strings.add("z");
strings.add("y");
strings.add("x");

for(String s : strings) {
    System.out.println(s);
}
```

```
x
y
z
```

The type used with a TreeSet must be naturally Comparable…

…***or*** a Comparator can be passed into the constructor, e.g.:

```java
new TreeSet<>(new DogComparator());
```

6.21 TreeSet Practice

There is nothing like conducting an experiment to see how the different Set implementations work. Practice using a HashSet and a TreeSet to store strings and see how they behave differently.

```
Set<String> strings = new TreeSet<>();
strings.add("z");
strings.add("y");
strings.add("x");

for(String s : strings) {
    System.out.println(s);
}
```

We are using ***polymorphism*** to write code that will work with any kind of Set.

- Create a new Java class named "SetsAndMaps" and define a method named "addAndPrint" that declares a parameter for a Set<String>.
  - Add at least 5 strings to the set in no particular order. Make sure to vary the case of the first letters of the strings, e.g. "aardvark", "Zoo", "Monkey", "zebra", "shark".
  - Use a for-each loop to iterate over the values in the set and print them to standard output.
- Define a main method with the appropriate signature.
  - Call addAndPrint with an empty HashSet<String>.
  - Call it a second time with an empty TreeSet<String>.
  - Run your code.

6.22 More TreeSet Practice

You may have noticed that the natural ordering of strings arranges capital letters before lowercase letters, e.g. "Zoo" comes before "aardvark". Let's fix that by writing a custom String Comparator and providing it to a TreeSet to use in sorting.

```
public class AlienComp
        implements Comparator<Alien> {

    @Override
    public int compare(Alien o1, Alien o2) {
        String sym1 = o1.getSymbol();
        String sym2 = o2.getSymbol();
        return
            sym1.compareToIgnoreCase(sym2);
    }
}
```

- Create a new Java class named AlphabeticComparatorthat implements Comparator<String>.
    - Override the compare method to compare two strings.
    - Convert the strings to ***lowercase***, then compare the first to the second, e.g. a.compareTo(b) .
    - Return the result.
- In the main method in the SetsAndMaps class:
    - Create an instance of the AlphabeticComparator and pass it into the constructor of a new TreeSet<String> .
    - Call the addAndPrint method with the new TreeSet.
    - Run your code!

A Closer Look at the `TreeMap`

```java
Map<Dog, Breed> dogs = new TreeMap<>(new DogComparator());
dogs.put(new Dog("Buttercup", 10), Breed.BASSET);
dogs.put(new Dog("Flash", 8), Breed.BLOODHOUND);
dogs.put(new Dog("Hermione", 2), Breed.BASSET);
dogs.put(new Dog("Princess Fluffypants", 1), Breed.CORGIE);

for(Dog dog : dogs.keySet()) {
    System.out.println(dog + " is a " + dogs.get(dog));
}
```

A Closer Look at the `TreeMap`

`TreeMap` is an implementation of the same abstract data type as `HashMap`.

A `TreeMap` provides all of the same operations as a `HashMap`, and also guarantees that the keys are unique…

The major difference that you will notice is that, when iterating over the keys, they are in ***sorted order***.

```java
Map<Dog, Breed> dogs = new TreeMap<>(new DogAgeComparator());
dogs.put(new Dog("Buttercup", 10), Breed.BASSET);
dogs.put(new Dog("Flash", 8), Breed.BLOODHOUND);
dogs.put(new Dog("Hermione", 2), Breed.BASSET);
dogs.put(new Dog("Princess Fluffypants", 1), Breed.CORGIE);

for(Dog dog : dogs.keySet()) {
    System.out.println(dog + " is a " + dogs.get(dog));
}
```

```
Princess Fluffypants(age 1) is a CORGIE
Hermione(age 2) is a BASSET
Flash(age 8) is a BLOODHOUND
Buttercup(age 10) is a BASSET
```

The ***key*** type used must be naturally `Comparable` or a `Comparator` must be given to the `TreeMap` to use when sorting the keys.

It does not matter whether or not the ***values*** in the map are `Comparable`. ***Only*** the keys are sorted.

6.23 TreeMap Practice

There is nothing like conducting an experiment to see how the different `Map` implementations work. Practice using a `HashMap` and a `TreeMap` to store string keys and see how they behave differently.

```
Map<Dog, Breed> dogs =
    new TreeMap<>(new DogComparator());
dogs.put(new Dog("Buttercup", 10), Breed.BASSET);
dogs.put(new Dog("Flash", 8), Breed.BLOODHOUND);
dogs.put(new Dog("Hermione", 2), Breed.BASSET);
dogs.put(new Dog("Princess Fluffypants", 1),
    Breed.CORGIE);

for(Dog dog : dogs.keySet()) {
    System.out.println(dog + " is a " +
        dogs.get(dog));
}
```

We are using ***polymorphism*** to write code that will work with any kind of `Map`.

- Open the `SetsAndMaps` class and define a method named "`putAndPrint`" that declares a parameter for a `Map<String, Integer>`.
  - Add at least 5 key/value pairs to the map in no particular order. Make sure to vary the case of the first letters of the keys, e.g. `"aardvark"`, `"Zoo"`, `"Monkey"`, `"zebra"`, `"shark"`.
  - Use a `for`-each loop to iterate over the keys in the map and print them to standard output.
- Define a `main` method with the appropriate signature.
  - Call `putAndPrint` with an empty `HashMap<String>`.
  - Call it a second time with an empty `TreeMap<String>`.
  - Call it a third time with an empty `TreeMap<String>` that uses an `AlphabeticComparator`.
  - Run your code.

Tree vs. Hash

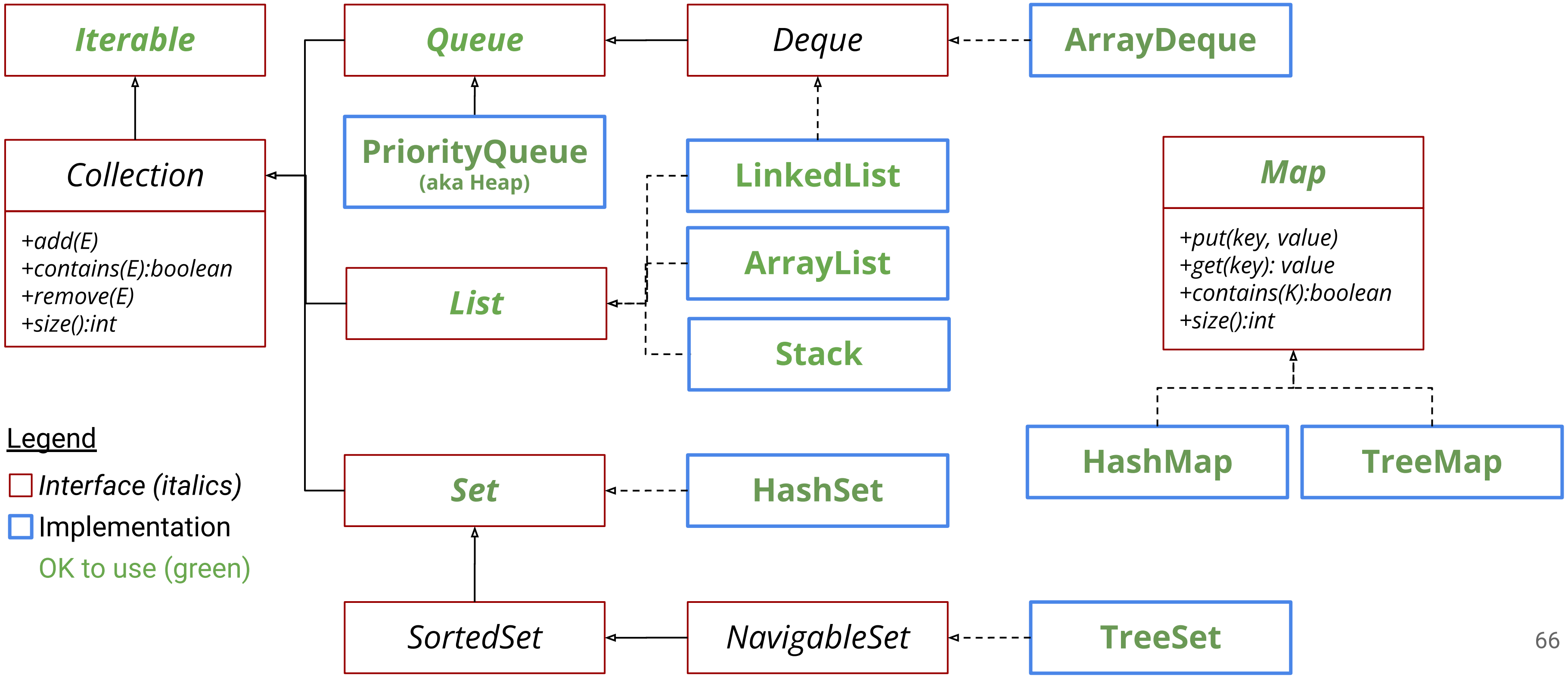
Not only are `TreeSet` and `TreeMap` harder to use because of the additional requirements that elements be `Comparable` and/or a `Comparator` be provided…

…but they are also ***slower*** than their hashing counterparts.

Unless the elements absolutely, positively ***must*** be kept in sorted order, choose the hashing version for its speed.

- A ***Red-Black Tree*** is a self-balancing binary search tree.
  - "***Self-balancing***" means that, as values are added, the tree makes sure that they are moved around to maintain balance without breaking the rules of a BST.
  - Remember that a balanced BST guarantees ***$O(\log_2 n)$*** search time.
  - The complexity of inserting into or removing from a red-black tree is ***$O(\log_2 n)$***.
- Java's `TreeSet` and `TreeMap` both use red-black trees to keep elements in sorted order.
  - This means that inserting a new value or removing a value has a complexity of ***$O(\log_2 n)$***.
  - It also means that determining whether or not a value is contained in the data structure has a complexity of ***$O(\log_2 n)$***.
- Recall that `HashSet` and `HashMap` provide ***$O(C)$*** (***constant time***) performance for all basic operations.
  - This means that adding, removing, and searching all run much faster than in the tree-based versions.
- Therefore, you should ***only*** use a `TreeSet` or `TreeMap` if you ***must*** maintain the elements in sorted order.
  - Otherwise, you should choose the ***faster*** versions.

An *Incomplete* JCF Class Hierarchy



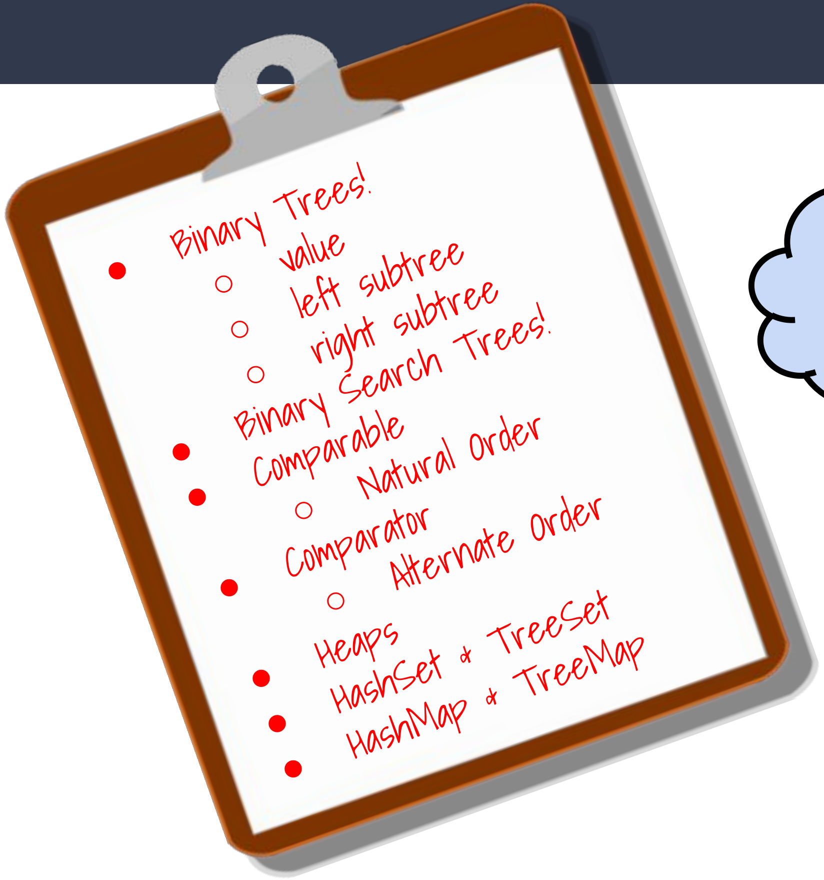
<u>Legend</u>

🟥 *Interface (italics)*

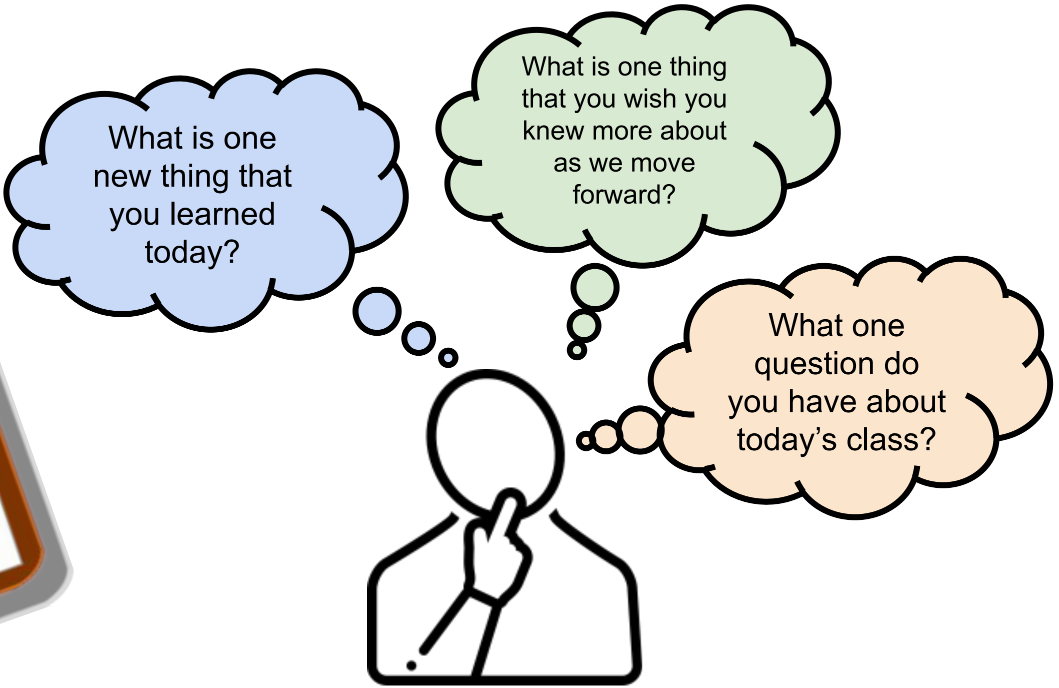
🟦 Implementation

OK to use (green)

Summary & Reflection



- Binary Trees!
  - value
  - left subtree
  - right subtree
- Binary Search Trees!
- Comparable
  - Natural Order
- Comparator
  - Alternate Order
- Heaps
- HashSet & TreeSet
- HashMap & TreeMap



What is one new thing that you learned today?

What is one thing that you wish you knew more about as we move forward?

What one question do you have about today's class?

Please answer the questions above in your notes for today.