DATASTRUCTURE & ALGORITHMS

STACK:

***STACK = LIFO data structure. Last In First Out***

* *Stores objects into a sort of vertical tower*
* *push () to add element to the top*
* *pop () to remove element from the top*
* *peek() to return the top element without remove/delete from stack*
* *other methods - empty(),search()*

***USES OF STACKS:***

1. *undo/redo features in text editors*
2. *moving back. Forward in browser history*
3. *backtracking algorithm (maze, file directories)*
4. *calling functions (call stack)*
5. Stack<String> stack = new Stack<String>();  
     
   System.out.println(stack.empty());  
   stack.push("Java");  
   stack.push("Spring");  
   stack.push("Hibernate");  
   stack.push("Oracle");  
   System.out.println(stack);  
   System.out.println(stack.peek());  
   stack.pop();  
   System.out.println(stack);  
   System.out.println(stack.empty());  
     
   /\*Exception in thread "main" java.lang.OutOfMemoryError:   
   Java heap space\*/  
   /\*for(int i=0;i<=1000000000; i++) {  
    stack.push("Microservice");  
   }\*/

***Exercise 🡪1 Stack: Constructor***

Implement a **Stack** class with a constructor that initializes a new stack with a given value.

The class should have the following attributes and constructor:

A **Node** class that represents the nodes of the stack:

* + An **int** attribute called **value** that stores the value of the node.
  + A **Node** attribute called **next** that points to the next node in the stack.
  + A constructor that accepts an integer value as an argument and initializes the **value** attribute with the given value.

1. A private **Node** attribute called **top** that points to the top node of the stack.
2. A private **int** attribute called **height** that keeps track of the number of elements in the stack.
3. A constructor for the **Stack** class that performs the following tasks:
   * Accepts an integer value as an argument, which will be the value of the first node in the stack.
   * Creates a new **Node** object called **newNode** with the given value.
   * Initializes the **top** attribute with the **newNode**.
   * Sets the **height** attribute to 1.

* *

***Exercise 🡪2 Stack: Push***

Implement a **push** method that adds a new node with a given value to the top of the stack.  
Return type: **void**

The method should perform the following tasks:

Accept an integer value as an argument, which will be the value of the new node.

1. Create a new **Node** object called **newNode** with the given value.
2. If the height of the stack is 0, set the **top** attribute to the **newNode**.
3. If the height of the stack is greater than 0, perform the following tasks:
   * Set the **next** attribute of the **newNode** to the current **top** node.
   * Update the **top** attribute to point to the **newNode**.
4. Increment the **height** attribute of the stack by 1.

*** ***

***Exercise 🡪3 Stack: Pop***

Implement a **pop** method that removes and returns the top node from the stack.  
Return type: **Node** (the node that was popped from the stack)

The method should perform the following tasks:

If the height of the stack is 0, return **null**.

1. Store the current **top** node in a temporary variable called **temp**.
2. Update the **top** attribute to point to the next node in the stack.
3. Set the **next** attribute of the **temp** node to **null** to disconnect it from the stack.
4. Decrement the **height** attribute of the stack by 1.
5. Return the **temp** node.

*** ***

QUEUE:

***QUEUE - FIFO Data structure. First In First Out***

***Example:***

1. *Line of People*
2. *A collection designed for holding elements prior to processing*
3. *Linear Data structure*

***Methods:***

1. *add element - enqueue ()/offer ()*
2. *remove element - dequeue ()/poll ()*
3. *returns head of the element - element ()/peek ()*

*/\*Queue<String> queue = new Queue<String>();\*/*

* *Error - Cannot instantiate the type Queue<String>*
* *Queue is an interface and not a class*
* *An interface cannot be instantiated*
* *Queue is implemented by 2 classes*

1. *LinkedList*
2. *Priority Queue*

* *Queue is inherited from Collection*
* *Methods inherited from Collection:*

1. *isEmpty()*
2. *size()*
3. *contains(obj)*

***USES OF QUEUE***

1. *Keyboard Buffer (letters should appear on the screen in the order it's typed)*
2. *Printer Queue (Print Jobs should be completed in order)*
3. *Used in LinkedList, Priority Queue, Breadth first search*
4. Queue<String> queue = new LinkedList<String>();  
    System.out.println(queue.isEmpty());  
    queue.offer("Print\_Job1");  
    queue.offer("Print\_Job2");  
    queue.offer("Print\_Job3");  
    queue.offer("Print\_Job4");  
    queue.offer("Print\_Job5");  
     
    System.out.println(queue);  
    System.out.println(queue.isEmpty());  
    System.out.println(queue.size());  
    System.out.println(queue.peek());  
    queue.poll();  
    System.out.println(queue);  
    System.out.println(queue.size());  
    System.out.println(queue.contains("Print\_Job1"));

***Exercise*** 🡪***1 Queue: Constructor***

Create a **Queue** class that represents a queue data structure using nodes.

Implement the constructor and instance variables as follows:

1. Create a private instance variable **first** that will store a reference to the first node in the queue.
2. Create a private instance variable **last** that will store a reference to the last node in the queue.
3. Create a private instance variable **length** that will store the current length of the queue.
4. Create a nested **Node** class with an integer **value** and a reference to the next **Node** in the queue.
5. Implement a constructor for the **Queue** class that takes an integer value as an argument, creates a new **Node** with the given value, and initializes the **first**, **last**, and **length** instance variables.

***Exercise*** 🡪***2 Queue: Enqueue/Offer***

Implement the **enqueue** method for the **Queue** class, which adds a new node to the end of the queue.  
Return type: **void**

The method should perform the following tasks:

Accept an integer value as an argument, which will be the value of the new node.

1. Create a new **Node** object called **newNode** with the given value.
2. If the length of the queue is 0, set both the **first** and **last** pointers of the queue to **newNode**.
3. If the length of the queue is greater than 0, perform the following tasks: a. Set the **next** attribute of the current **last** node to **newNode**. b. Update the **last** pointer of the queue to point to **newNode**.
4. Increment the **length** attribute of the queue by 1.

*** ***

***Exercise*** 🡪3 ***Queue: Dequeue/Poll***

Implement the **dequeue** method for the **Queue** class, which removes and returns the first node from the queue.  
Return type: **Node** (the node that has been dequeued)

The method should perform the following tasks:

If the length of the queue is 0, return **null**.

1. Create a temporary **Node** variable called **temp** and set it to the current **first** node.
2. If the length of the queue is 1, set both the **first** and **last** pointers of the queue to **null**.
3. If the length of the queue is greater than 1, perform the following tasks: a. Update the **first** pointer of the queue to point to the next node. b. Set the **next** attribute of the **temp** node to **null**.
4. Decrement the **length** attribute of the queue by 1.
5. Return the **temp** node.

* *

TREE:

***Binary search tree is a data structure that quickly allows us to maintain a sorted list of numbers.***

* *It is called a binary tree because each tree node has a maximum of two children.*
* *It is called a search tree because it can be used to search for the presence of a number in O(log(n)) time.*

***The properties that separate a binary search tree from a regular binary tree is***

1. *All nodes of left subtree are less than the root node*
2. *All nodes of right subtree are more than the root node*
3. *Both subtrees of each node are also BSTs i.e. they have the above two properties\*

***Exercise*** 🡪**1 *Binary Search Tree: Constructor***

Create a simple binary search tree class called **BinarySearchTree** with a **Node** inner class and a **root** attribute to hold the reference to the root node of the tree.

The class should include the following:

A **Node** inner class with the following properties:

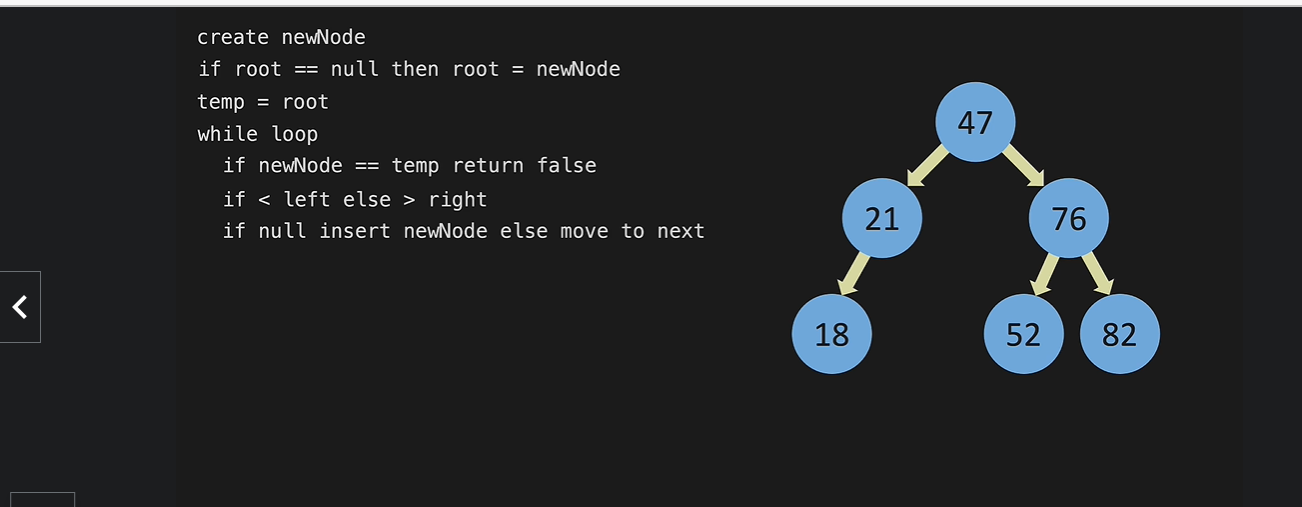
* + An integer value.
  + A reference to the left child node.
  + A reference to the right child node.
  + A constructor that takes an integer value and initializes the node with it.

1. A **root** attribute that will store a reference to the root node of the tree.

*** ***

***Exercise*** 🡪**2 *Binary Search Tree: Insert***

***Steps to Insert a New Node in Binary Search Tree***



Implement the **insert** method that adds a new node to the tree.  
  
Return type: **boolean**

The method should perform the following tasks:

1. Create a new **Node** object called **newNode** with the given integer value.
2. If the root node is null, set the root to the **newNode** and return true.
3. Create a temporary node called **temp** and set it to the root node.
4. Use a while loop to traverse the tree:
   * If the **newNode** value is equal to the **temp** node value, return false (no duplicates allowed).
   * If the **newNode** value is less than the **temp** node value:
     + If the left child of **temp** is null, set the left child to **newNode** and return true.
     + Otherwise, set **temp** to its left child.
   * If the **newNode** value is greater than the **temp** node value:
     + If the right child of **temp** is null, set the right child to **newNode** and return true.
     + Otherwise, set **temp** to its right child.

*** ***

***Exercise*** 🡪**3 *Binary Search Tree: Contains***

***Steps to Find whether a given Node is present in Binary Search Tree***

*If root == null return false*

*temp = root*

*while temp != nul*

*if < left*

*else if > right*

*else == return true*

*return false*

Implement the **contains** method that checks if a node with a given value exists in the tree.  
  
Return type: **boolean**

The method should perform the following tasks:

1. If the root node is null, return false.
2. Create a temporary node called **temp** and set it to the root node.
3. Use a while loop to traverse the tree:
   * If the given value is less than the **temp** node value, set **temp** to its left child.
   * If the given value is greater than the **temp** node value, set **temp** to its right child.
   * If the given value is equal to the **temp** node value, return true.
4. If the loop exits without finding a match, return false.

HASH TABLE:

***Exercise*** 🡪**1 *Hash Table: Constructor***

Create a **HashTable** class with the following attributes and an inner **Node** class:

1. An integer attribute named **size**, initialized with a default value of 7.
2. An array of **Node** objects named **dataMap** with the length equal to the **size** attribute.
3. An inner class called **Node** with the following attributes:
   * A String attribute named **key**.
   * An integer attribute named **value**.
   * A Node attribute named **next** (for handling collisions via chaining).
4. A constructor for the **Node** class that accepts a String **key** and an integer **value**, initializing the respective attributes.
5. A constructor for the **HashTable** class that initializes the **dataMap** attribute as an array of **Node** objects with a length equal to the **size** attribute.

***Exercise*** 🡪2 ***Hash Table: Hash Method***