DATASTRUCTURE & ALGORITHMS

**TOPICS TO BE COVERED**

***DATASTRUCTURES:***

1. Linked Lists
2. Doubly Linked Lists
3. Stacks & Queues
4. Binary Search Trees
5. Hash Tables
6. Graphs

***ALGORITHMS:***

1. Bubble Sort
2. Selection Sort
3. Insertion Sort
4. Merge Sort
5. Quick Sort
6. Tree Traversal

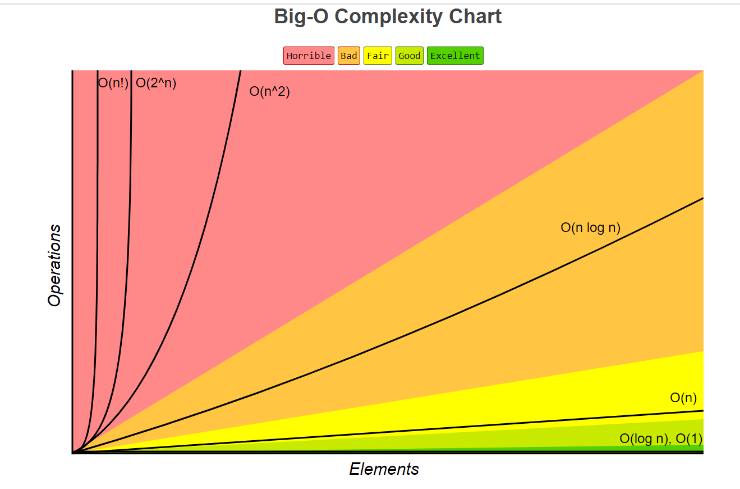
* Breadth First Search
* Depth First Search - Pre Order
* Depth First Search - Post Order
* Depth First Search - In Order

***COURSE TAKEN FROM UDEMY:***

[Java Data Structures & Algorithms + LEETCODE Exercises (udemy.com)](https://dxc.udemy.com/course/data-structures-and-algorithms-java/learn/lecture/27815396#overview)

**BIG O NOTATION**

* Time Complexity – Calculated Based on how long (time) it takes to execute the code
* Space Complexity – Calculated Based on how much space is consumed when code is executed

***CATERGORIES OF BIG – O:***

* *LINEAR TIME – O(n)*
* *LINEAR TIME- DROP CONSTANTS.*
* *QUADRATIC TIME – O(n^2)*
* *DROP NON-DOMINANTS*
* *CONSTANT TIME - O (1)*
* *LOGARITHMIC TIME -O (log n)*
* *QUASILINEAR TIME – O (n log n)*
* *DIFFERENT TERM OF INPUTS*

*For O(n) & O(n^2)*

***LINEAR TIME – O(n)***

*Examples:*

* *Looping through an element in an array*
* *Searching through a Linked List*

package com.big.o;  
  
public class LinearTime {  
//O(n) - LinearTime Example  
 public static void printItems(int n){  
 for(int i =0; i< n ;i++){  
 System.*out*.println(i);  
 }  
 }  
 public static void main(String[] args){  
 *printItems*(10);  
 }  
}

***LINEAR TIME- DROP CONSTANTS***

*O(Xn), where X may be 1,2,3…etc.*

*n + n = 2n , where we drop constants, and it becomes O(n)*

package com.big.o;  
  
public class LinearTime\_dropConstants {  
//O(2n) or O(3n)... O(Xn) = Drop Constants and its O(n) - Linear Time Example  
 public static void printItems(int n){  
 for(int i =0; i< n ;i++){  
 System.*out*.println(i);  
 }  
 for(int j =0; j< n ;j++){  
 System.*out*.println(j);  
 }  
 }  
 public static void main(String[] args){  
  
 *printItems*(10);  
 }  
}

***QUADRATIC TIME – O(n^2)***

***n \* n = n2 🡺 O(n2)***

*EXAMPLES :*

* *Insertion Sort*
* *Selection Sort*
* *Bubble Sort*
* package com.big.o;  
    
  public class QuadraticTime {  
    
   //O(n Exponent of 2) - n \* n = n exponent 2  
    
   public static void printItems(int n){  
   for(int i =0; i< n ;i++){  
   for(int j =0; j< n ;j++){  
   System.*out*.println(i + " " +j);  
   }  
   }  
    
   }  
   public static void main(String[] args){  
    
   *printItems*(10);  
   }  
  }

***DROP NON-DOMINANTS***

***O(n2+ n) = O(n2) ,where + n is dropped***

*If n =100, (n2) = 1000 and n = 100*

*Here* ***(n2) is Dominant*** *and* ***n is Non – Dominant***

package com.big.o;  
  
public class QuadraticTime\_NonDominant\_Drop {  
 /\*O(n2+ n ) = O(n2) ,where + n is dropped  
 If n =100, (n2) = 1000 and n = 100  
 Here (n2) is Dominant and n is Non – Dominant\*/  
 public static void printItems(int n){  
 for(int i =0; i< n ;i++){  
 for(int j =0; j< n ;j++){  
 System.*out*.println(i + " " +j);  
 }  
 }  
 for( int k = 0; k < n ; k++){  
 System.*out*.println(k);  
 }  
 }  
 public static void main(String[] args){  
 *printItems*(10);  
 }  
}

***CONSTANT TIME – O(1)***

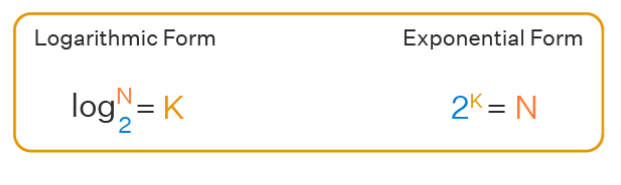
*n + n 🡺 No of Operations is only one, so O(1)*

*As n grows, No of Operations always stays constant*

*Examples:*

* *Random Access of an element in an array*
* *Insertion at the beginning of the LinkedList*
* package com.big.o;  
    
  public class ConstantTime {  
   /\*  
   n + n ⎝ No of Operations is only one, so O(1)  
   As n grows, No of Operations always stays constant\*/  
    
    
   public static void printItems(int n){  
   System.*out*.println(n+n);  
   }  
   public static void main(String[] args){  
    
   *printItems*(10);  
   }  
  }

***LOGARITHMIC TIME – O(log n)***



*If N = 8,*

*Log2N = K 🡺 Log28 = 3, i.e. 23=8*

*EXAMPLES:*

* *Binary Search*

*Here I want to find number 1 from an array.*

8

7

6

5

1

2

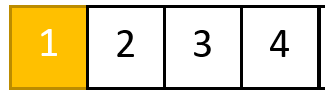
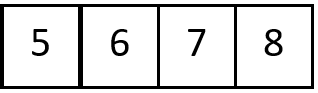
4

3

**OMIT IF TARGET NOT FOUND**

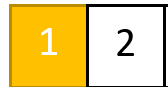
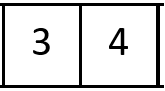
Split the Array into Two Halves equally

**Step = 1**

**OMIT IF TARGET NOT FOUND**

**Step = 2**

**Step = 3**

**OMIT IF TARGET NOT FOUND**

***QUASILINEAR TIME – O(n log n)***

*EXAMPLES*

* *Quick Sort*
* *Merge Sort*
* *Heap Sort*

***DIFFERENT TERM OF INPUTS***

* *Linear Time – O(n)*
* *Quadratic Time - O(n^2)*

*Linear Time – O(n) 🡺 if Two Inputs a, b then O (a +b)*

package com.big.o;  
  
public class DiffTermInputs1 {  
 public static void printItems(int a, int b){  
 for(int i =0; i< a ;i++){  
 System.*out*.println(i);  
 }  
 for(int j =0; j< b ;j++){  
 System.*out*.println(j);  
 }  
 }  
 public static void main(String[] args){  
  
 *printItems*(5,4);  
 }  
}

*Quadratic Time - O(n^2)🡺 if Two Inputs a, b then O (a \* b)*

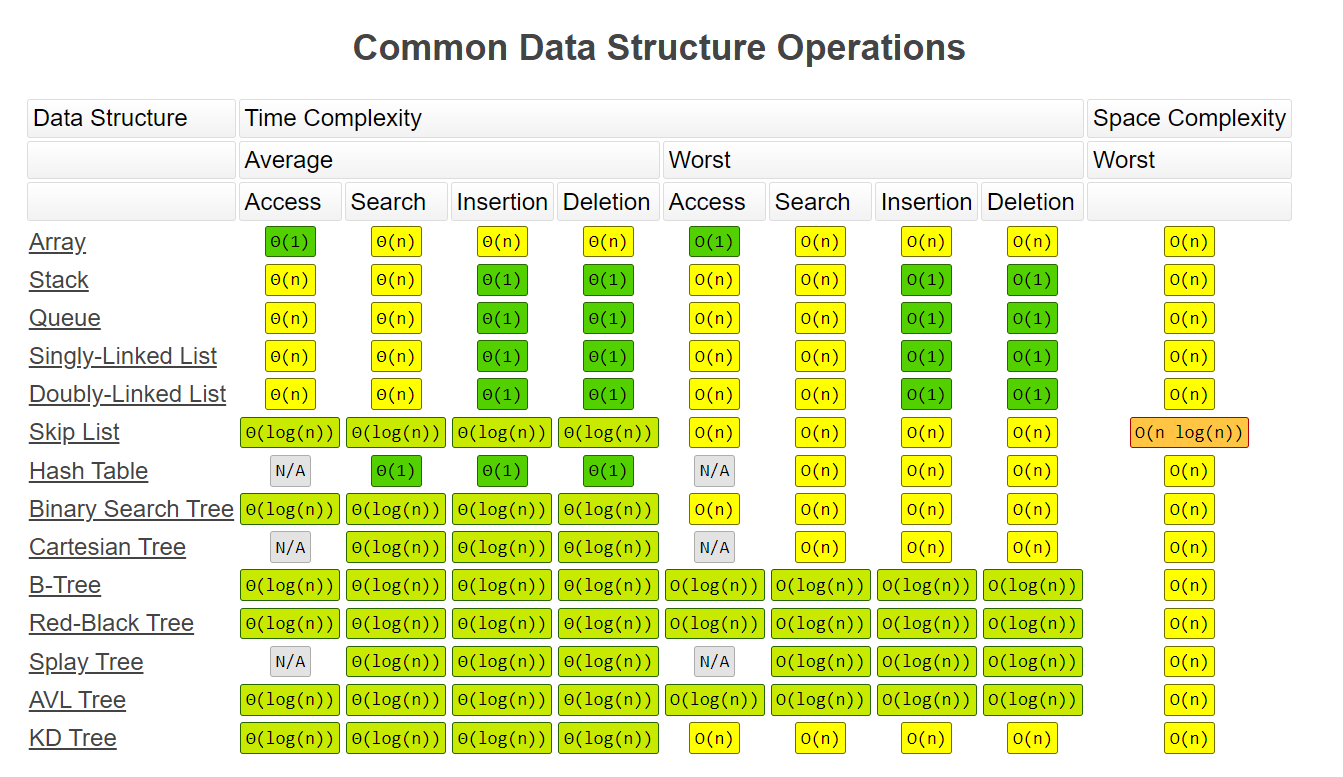
package com.big.o;  
  
public class DiffTermInputs2 {  
 public static void printItems(int a, int b){  
 for(int i =0; i< a ;i++){  
 for(int j =0; j< b ;j++){  
 System.*out*.println(i + " " +j);  
 }  
 }  
 }  
 public static void main(String[] args){  
  
 *printItems*(5,4);  
 }  
}

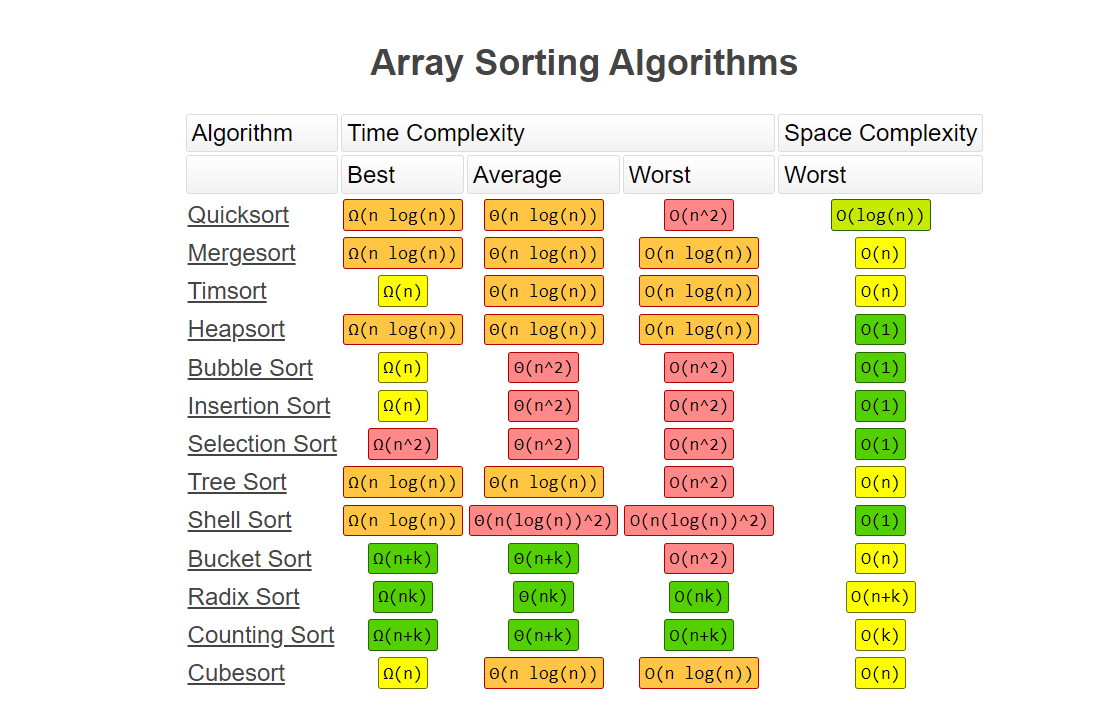
***BIG O ARRAYLIST***

* *If Adding or Removing an Element at Last Index of An array, then Time Complexity is O(1)*
* *If Adding or Removing an Element at First or Middle Index of an array, then Time Complexity is O(n)*
* *If fetching the value from an array using that specific value, then Time Complexity is O(n)*
* *If fetching the value from an array using that specific index, then Time Complexity is O(1)*

package com.big.o;  
  
import java.util.ArrayList;  
import java.util.Arrays;  
import java.util.List;  
  
public class BigOarrayList {  
 public static void main(String[] args){  
 List<Integer> myList = new ArrayList<Integer>(Arrays.*asList*(3,5,7,9,11));  
 myList.add(13);  
 System.*out*.println("Element Added " +myList);  
 myList.remove(5);  
 System.*out*.println("Element Removed " +myList);  
 myList.add(1,0);  
 System.*out*.println("Element Removed " +myList);  
 myList.remove(1);  
 System.*out*.println("Element Removed " +myList);  
 }  
}

***BIG O CHEAT SHEET***





***CLASSES :***

package com.classes;  
  
public class Cookie {  
 private String color;  
  
 public Cookie(String color) {  
 this.color = color;  
 }  
 public String getColor() {  
 return color;  
 }  
 public void setColor(String color) {  
 this.color = color;  
 }  
  
}

package com.classes;  
  
// Press Shift twice to open the Search Everywhere dialog and type `show whitespaces`,  
// then press Enter. You can now see whitespace characters in your code.  
public class Main {  
 public static void main(String[] args) {  
  
 Cookie cookieOne = new Cookie("Yellow");  
 Cookie cookieTwo = new Cookie("Red");  
 cookieOne.setColor("Pink");  
 System.*out*.println(cookieOne.getColor());  
 System.*out*.println(cookieTwo.getColor());  
 }  
}

***POINTERS:***

*Without Pointers Example:*

package com.pointers;  
import java.sql.SQLOutput;  
public class WithoutPointers {  
 public static void main(String[] args){  
 int num1 = 11;  
 int num2 = num1;  
 num1 = 22;  
 System.*out*.println(num1);  
 System.*out*.println(num2);  
 }  
}

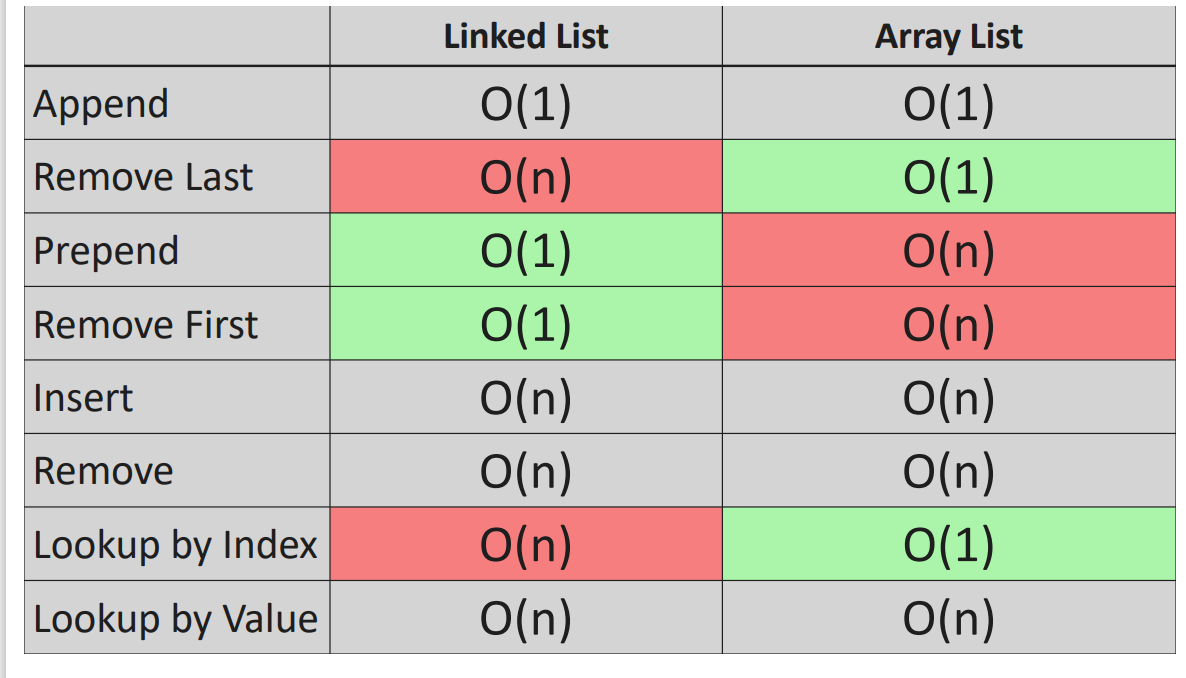
*With Pointers Example:*

package com.pointers;  
import java.util.HashMap;  
public class WithPointers {  
 public static void main(String[] args){  
 HashMap<String,Integer> map1 = new HashMap<>();  
 HashMap<String,Integer> map2 = new HashMap<>();  
 map1.put("value",11);  
 map2 = map1;  
 map1.put("value",22);  
 System.*out*.println(map1);  
 System.*out*.println(map2);  
 }  
}

***DIFFERENEC BETWEEN ARRAYLIST AND LINKEDLIST***

|  |  |  |  |
| --- | --- | --- | --- |
|  | **ArrayList** | **LinkedList** | |
| Internal Implementation | Uses an array to store elements | Uses a doubly linked list to store elements |
| Access time | O(1) for random access, O(n) for insertion and deletion | O(n) for random access, O(1) for insertion and deletion |
| Memory usage | More memory is used for maintaining the size of the array | Less memory is used since only the elements and pointers are stored |
| Iteration performance | Fast, since elements are stored in contiguous memory locations | Slower, since elements are not stored in contiguous memory locations |
| Adding elements | Can be slow if the size of the array needs to be increased to accommodate new elements | Fast, since only pointers need to be updated |
| Removing elements | Can be slow if elements need to be shifted to fill the gap left by the removed element | Fast, since only pointers need to be updated |
| Thread safety | Not inherently thread-safe, but can be made thread-safe using synchronization. | Not inherently thread-safe, but can be made thread-safe using synchronization. |
| Use cases | Best suited for scenarios where random access is required and the list will not be modified frequently | Best suited for scenarios where insertion and deletion are frequent, and random access is not required. |

***BIG O -LINKEDLIST vs ARRAYLIST***



***CODING EXERCISE FOR LINKED LIST***

**EXERCISE 1 🡺LL: Constructor**

Write a Java class called **LinkedList** that represents a singly linked list.  
The class should have the following properties and methods:

1. A private **Node** object called **head** that points to the first node in the linked list.
2. A private **Node** object called **tail** that points to the last node in the linked list.
3. A private integer attribute called **length** that represents the number of nodes in the linked list.
4. A nested **Node** class with the following attributes:
   * An integer attribute called **value** that stores the value of the node.
   * A **Node** object called **next** that points to the next node in the linked list.
   * A constructor that accepts an integer **value** as an argument and initializes the **value** attribute.
5. A constructor for the **LinkedList** class that accepts an integer **value** as an argument and performs the following tasks:
   * Create a new **Node** object called **newNode** with the given **value**.
   * Set both the **head** and **tail** pointers of the list to the **newNode**.
   * Set the **length** attribute of the list to 1.

*  *

**EXERCISE 2 🡺LL: Append**

Implement the **append** method that appends a new node to the end of the linked list.  
Return type: **void**

The method should perform the following tasks:

1. Accept an integer **value** as an argument, which will be the value of the new node.
2. Create a new **Node** object called **newNode** with the given **value**.
3. If the length of the linked list is 0, set both the **head** and **tail** pointers of the list to the **newNode**.
4. If the length of the linked list is greater than 0, perform the following tasks:
   * Set the **next** attribute of the current **tail** node to the **newNode**.
   * Update the **tail** pointer of the list to point to the **newNode**.
5. Increment the **length** attribute of the list by 1.

* *

**EXERCISE 3 🡺LL: Remove Last**

Implement a method called **removeLast** that removes the last node from the linked list.  
Return type: **Node** (we are returning the Node that is being removed)

The method should perform the following tasks:

1. If the length of the linked list is 0, return null.
2. Create two variables that can point to a **Node** , **temp** and **pre**, both initially pointing to the head of the linked list.
3. Traverse the linked list until **temp.next** is null, performing the following tasks during traversal:
   * Update **pre** to point to the current **temp** node.
   * Update **temp** to point to the next node in the linked list.
4. After traversal, set the **tail** pointer to **pre**.
5. Set the **next** attribute of the **tail** node to null.
6. Decrement the **length** attribute of the list by 1.
7. If the length of the linked list becomes 0 after removing the last node, set both the **head** and **tail** pointers to null.
8. Return the removed node (pointed to by **temp**).

* *

**EXERCISE 4 🡺LL: Prepend**

Implement a method called **prepend** that adds a new node at the beginning of the linked list.  
Return type: **void**

The method should perform the following tasks:

1. Accept an integer value as an argument, which will be the value of the new node.
2. Create a new Node object called **newNode** with the given value.
3. If the length of the linked list is 0, set both the head and tail pointers of the list to the **newNode**.
4. If the length of the linked list is greater than 0, perform the following tasks:
   * Set the **next** attribute of **newNode** to the current head node.
   * Update the head pointer of the list to point to the **newNode**.
5. Increment the length attribute of the list by 1.

* *

**EXERCISE 5 🡺LL: Remove First**

Implement a method called **removeFirst** that removes the first node from the linked list and returns it.  
Return type: **Node** (the node that is removed)

The method should perform the following tasks:

1. If the length of the linked list is 0, return null.
2. Create a temporary Node object called **temp** and set it to the current head node.
3. Update the head pointer of the list to point to the next node in the list.
4. Set the **next** attribute of **temp** to null.
5. Decrement the length attribute of the list by 1.
6. If the length of the list becomes 0 after removing the first node, set the tail pointer to null.
7. Return the removed **temp** node.

* *

**EXERCISE 6 🡺LL: Get**

Implement a method called **get** that retrieves a node from the linked list by its index.  
Return type: **Node**

The method should perform the following tasks:

1. Accept an integer **index** as an argument, which will be the index of the node to be retrieved.
2. If the index is less than 0 or greater than or equal to the length of the linked list, return null.
3. Create a temporary Node object called **temp** and set it to the current head node.
4. Iterate through the linked list by updating **temp** to its next node until the desired index is reached.
5. Return the node located at the specified index.

**EXERCISE 7 🡺LL: Set**

Implement a method called **set** that updates the value of a node in the linked list by its index.  
Return type: **Boolean**

The method should perform the following tasks:

Accept an integer **index** as an argument, representing the index of the node to be updated.

1. Accept an integer **value** as an argument, representing the new value to set for the node at the specified index.
2. Call the **get** method with the provided index to obtain a pointer to the node at the specified index.
3. If the node exists (i.e., the **get** method returns a non-null value), update the node's value with the provided value and return true.
4. If the node does not exist (i.e., the **get** method returns null), return false.

* *

**EXERCISE 8 🡺LL: Insert**

Implement a method called **insert** that inserts a new node at a specified index in the linked list.  
Return type: **Boolean**

The method should perform the following tasks:

Accept an integer **index** as an argument, representing the index at which the new node should be inserted.

1. Accept an integer **value** as an argument, representing the value of the new node.
2. If the index is less than 0 or greater than the length of the list, return false.
3. If the index is 0, call the **prepend** method with the provided value and return true.
4. If the index is equal to the length of the list, call the **append** method with the provided value and return true.
5. Create a new Node object called **newNode** with the provided value.
6. Call the **get** method with the index minus 1 to obtain the node before the specified index.
7. Update the **next** attribute of the **newNode** to point to the **next** attribute of the node before the specified index.
8. Update the **next** attribute of the node before the specified index to point to the **newNode**.
9. Increment the length attribute of the list by 1.