ASSIGNMENT 2 FRONT SHEET

Qualification	BTEC Level 5 HND Diploma in Computing		
Unit number and title	Unit 19: Data Structures and Algorithms		
Submission date	04/09/2021	**Date Received 1st submission**	04/09
Re-submission Date	15/9/2021	**Date Received 2nd submission**	15/09
Student Name	Trinh Minh Duc	**Student ID**	GCH190602
Class	GCH190602	**Assessor name**	Do Hong Quan

Student declaration

I certify that the assignment submission is entirely my own work and I fully understand the consequences of plagiarism. I understand that making a false declaration is a form of malpractice.

	Student's signature	Duc

Grading grid

P4	P5	P6	P7	M4	M5	D3	D4

☐ **Summative Feedback:** ☐ **Resubmission Feedback:**

Grade: | **Assessor Signature:** | **Date:**
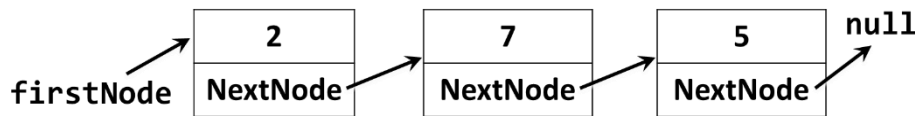
Internal Verifier's Comments:

IV Signature:

Assignment Brief and Guidance

Scenario

As assigned by the manager, in this work, you are asked to provide proper implementations and evaluations for the following tasks.

(a) Implement a stack by a "linked list" that follows this graph representation below as underlying data structure.

(b) Implement a queue by a "doubly-linked list" that follows this graph representation below as underlying data   structure.
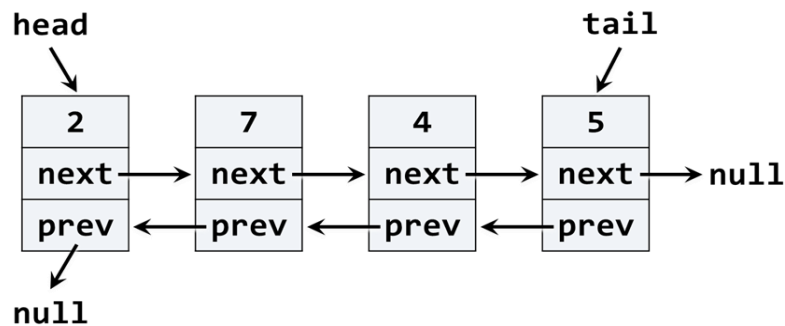
(c) Implement and compare two solutions to reverse a Stack, that are using Recursion and without Recursion. The implemented operation should take a Stack as the input. Consider using only those data structures that are already implemented in Q.a and Q.b.

Note that:

(1)   Most defined operations of Stack and Queue as stated in Assignment 1 should be implemented.

(2)   Errors should be handled carefully by exceptions and in the report some tests should be executed to prove the correctness of algorithms / operations.

(3)   Besides, in the report, the work should evaluate the use of ADT in design and development, including the complexity, the trade-off and the benefits.

Contents

I. Design and implementation of Stack ADT and Queue ADT (P4).

1. Stack

a. **Introduction**.

A stack of type T elements is a finite, ordered sequence of type T elements in which all insertions and deletions are limited to one end, the top.

The -Stack data structure is based on the Last In - First out (LIFO) principle. LIFO: The last item added to the stack is the first to be removed.

- We may use an array or a link-list to construct a stack. In this lesson, I'll use a link-list to implement a stack.

The author will create a Stack ADT based on the following fundamental structure of a singlylinkedlist with progressive form:
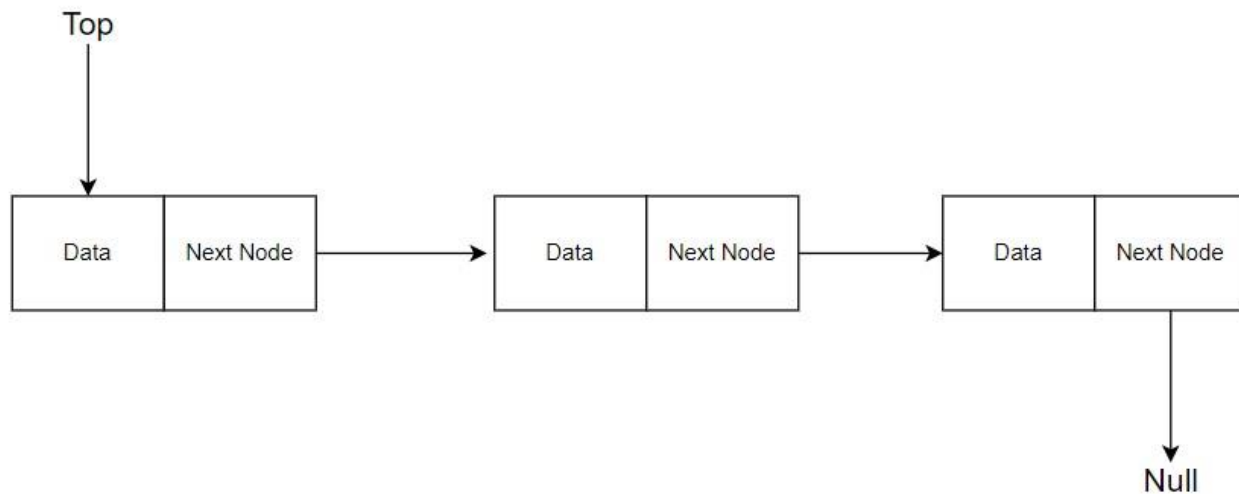

Figure 1: a singly-linked list Stack

- This link-list is made up of Nodes and has a single link-list structure; mark the first Node for the first Node. The Link-initial list's element is top.

Node's Structure:

```
 6⊖    static class Node {
 7           // Declare the attribute
 8           int Data;
 9           Node NextNode;
10           // Construct the Node
11⊖         public Node(int Data) {
12               this.Data = Data;
13               this.NextNode = null;
14           }
```
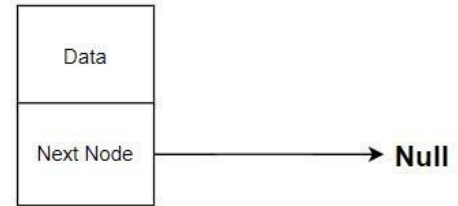
Figure 2: Description of Node

Line: 8 + 9 Declare **Data** and **NextNode** are the two components of a node.

The author assigns two values to the starting value. Both **Data** and **NextNode** are **Null**. To put it another way, the array of values contains nothing at first.

b. Operations
How the stack works as follows:
o **void push**(**item**): adds a new item to the stack's top. It requires the item but does not provide a response;
o **pop()**: pops the item at the top of the stack. It returns the item with no arguments. The stack has been changed. o **peek()** retrieves the stack's top item but does not delete it. There are no parameters required. The stack isn't changed.
o **boolean isEmpty**(): Determines whether or not the stack is empty. It has no inputs and outputs a boolean value. When the stack is empty, return "true."
o **int size**(): Returns the number of accessible stack items (Top+1). It has no inputs and outputs an integer.
c. Implementation -
My Stack interface:
The author created NextNode to mark a new Node added to MyStack.
The author initialize the variable count = the Nodes that MyStack has.MyStack.Max_size: is the maximum number of items that MyStack scan store that the user enters from the screen.

```
2 public class MyStack {
3     // Create class Node class
4     // sua so luong size
5     // sreach co tra ve vi tri ko tra vè -1;
6⊕    static class Node {□
16    // #1 Declare the attribute
17    // total number of items the stack can store
18    int Max_size;
19    // the first pointer points to the first position of the stack
20    Node firstNode = null;
21    // đếm số phần tử trongstack
22    int count = 0;
23⊕   public MyStack(int Max_size) {□
26    // #2 Operation
27    // create operation Push()
28⊕   public void Push(int items) {□
49
50    // create operation Pop()
51⊕   public int Pop() {□
74
75    // create operation size()
76⊕   public int Size() {□
81
82    // create operation isEmpty()
83⊕   public boolean isEmpty() {□
91
92    // create operation Print()
93⊕   public void Print() {□
108   // create operation peek()
109
110⊕  public int Peek() {□
117
118⊕  public boolean constain(int Data) {□
127
128 }
```

Figure 3: Stack interface

- **Push (item):** a void Push (int item) is a new method that adds a new item to the top of the stack. It requires the item but does not provide a response.
  Lines 30 to 34: The author created a conditional function to check if the elements are overvalued. At line 30 the author defaulted to **Max_size()** the maximum value of the array to be customized by the user. If in the case that is equal to or greater than the default element, the statement will return:

Stack is overflow. Otherwise, it will run and print as usual and without the statement of this function.

```
30        if (Size() >= Max_size) {
31            System.out.println("Stack is overflow");
32
33            return;
34        }
```

Lines 35 to 39, The author has created a hypothetical case in that array that does not have an element but wants to add an element, what will happen? Here the value of **FirstNode** changes from **null** to **aNode** and the variable **count** will increase.

```
35        Node aNode = new Node(items);
36        if (firstNode == null) {
37            firstNode = aNode;
38            count++;
39            return;
```
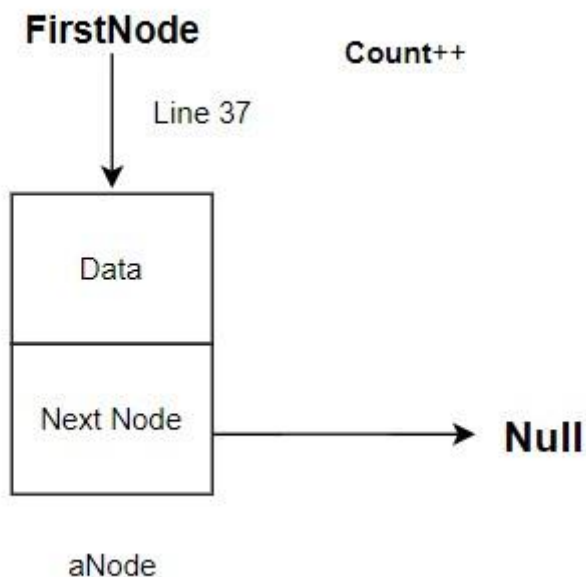

Figure 4: Description of line code 35 – 39

Lines 40 to 46, the line of code that executes when it already has an element in it. When adding a new Node in the array. Then it is necessary to assign **aNode.NextNode** to **FirstNode** and assign vertex to **aNode** and increment **Count** by 1.

```
40            } else {
41
42                    // Make aNode the first element
43                    aNode.NextNode = firstNode;
44                    // aNode becomes firsNode
45                    firstNode = aNode;
46                    count++;
47            }
```
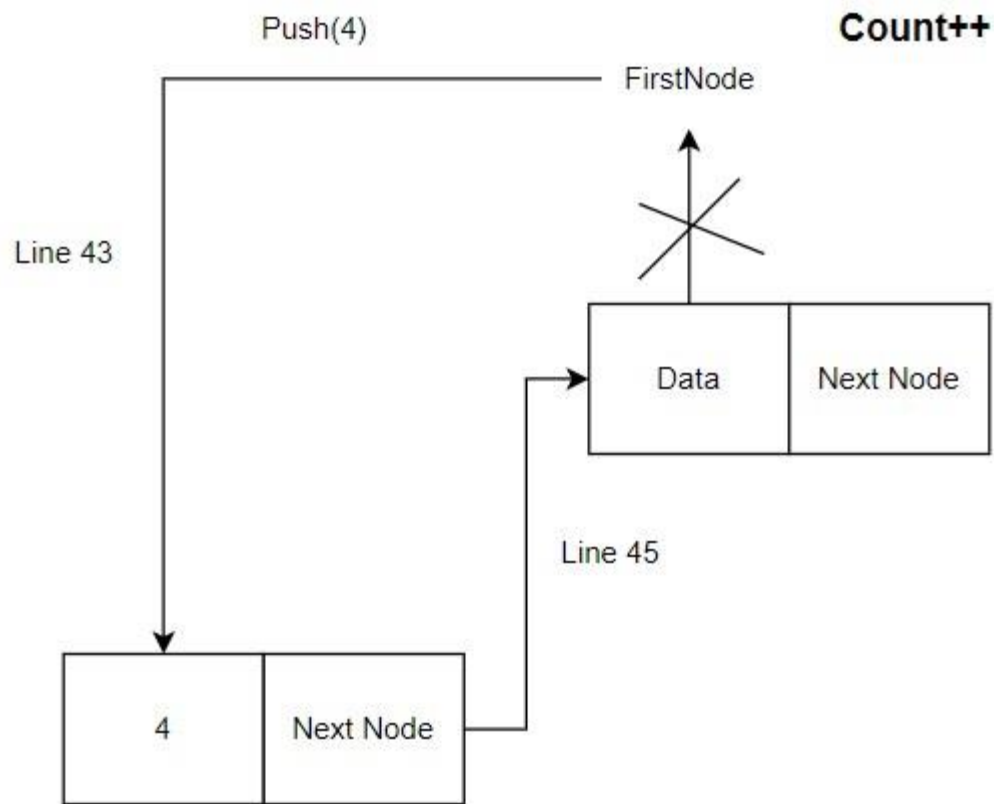

Figure 5: Description of line code 40 to 47.

- **Pop ():**

```
51    public int Pop() {
52        // check if the stack is empty
53        if (isEmpty()) {
54            System.out.println("Stack is underflow");
55            // returns extremely small number
56            return -1;
57        }
58        int temp;
59        // Save return value firstNode
60        temp = firstNode.Data;
61        if (firstNode.NextNode != null) {
62            Node curr = firstNode;
63            firstNode = firstNode.NextNode;
64            // curr.NextNode = null;
65            count--;
66        } else {
67            // Remove node at firstNode
68            firstNode = null;
69            count--;
70        }
71        // Return saved value
72        return temp;
73    }
```
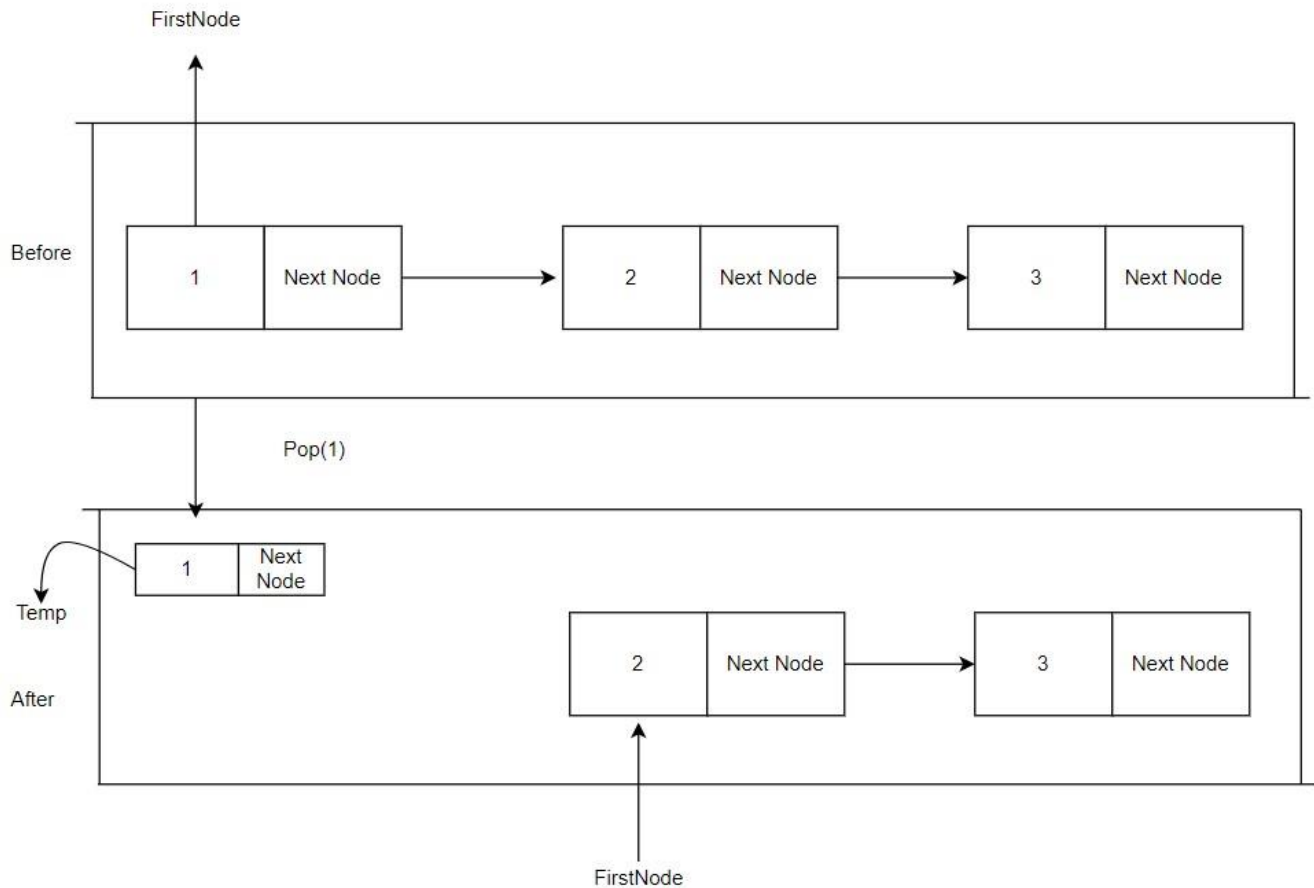
In the next action, the author has created a new **Pop**() function for the purpose of deleting the element he wants. **First in lines 53 to 57**, the author created a function that checks if there is an element in that function, if there is no element, it will return: Stack is underflow and will not execute the next statements.

From line 58 to 73, the author has created a variable temp that stores the first element to be deleted. If **FirstNode.NextNode** is non-null, the delete attempt will be assigned with **firstNode**, and then **firstNode = FirstNode.NextNode**. The **count** counter will decrement by 1 element.

In lines 67 to 69, which is the case with only 1 element, when deleted, **firstNode** will be equal to Null and count will decrease back to its original value of 0.

FirstNode

Before

1 | Next Node → 2 | Next Node → 3 | Next Node

Pop(1)

Temp

1 | Next Node

After

2 | Next Node → 3 | Next Node

FirstNode

- **Peek**():

```
109  public int Peek() {
110      if (isEmpty()) {
111          System.out.println("Stack is empty");
112          return -1;
113      }
114      return firstNode.Data;
115  }
```

The author created the Peek(): method to rerun the first element.

If there is no first element, Peek will return: Stack is empty.

- **isEmpty**():

```
82⊖    public boolean isEmpty() {
83         // if stack is empty return true not empty
84         if (firstNode == null) {
85             System.out.println("Stack is empty");
86             return true;
87         }
88         return false;
89     }
```

The author created a **boolean isEmpty**(): method to check if the function is empty or not. By first assigning the condition **firstNode = Null**, in this case the statements will return Stack is empty and **true**; and if not null, this value will return **false**.

- **Constrain(int Data):**

```
public boolean constain(int Data) {
    Node curr = firstNode;
    while (curr != null) {
        if (curr.Data == Data)
            return true;
        curr = curr.NextNode;
    }
    return false;
}
```

The author created a search function and named it Constrain(): To search for an element in the array, we must have an element in the array, the author gave a statement to check if that element is in the array if none will return false.

If there is a value, it will return that value and return true.

2. Queue
a. Introduction

A queue is a linear structure that executes tasks in a predetermined sequence. First in, first out (FIFO) refers to the order in which the pieces are placed in the Queue. A queue is a line of customers waiting for a resource, with the first client being served first. The stack and queue are dropped in the various procedures. The most recently added item is taken out of the stack, whereas the most recently added item is taken out of the queue. The author will discuss Queues' uses and construct a queue in the parts that follow. The ADT algorithm is built on a double topology.
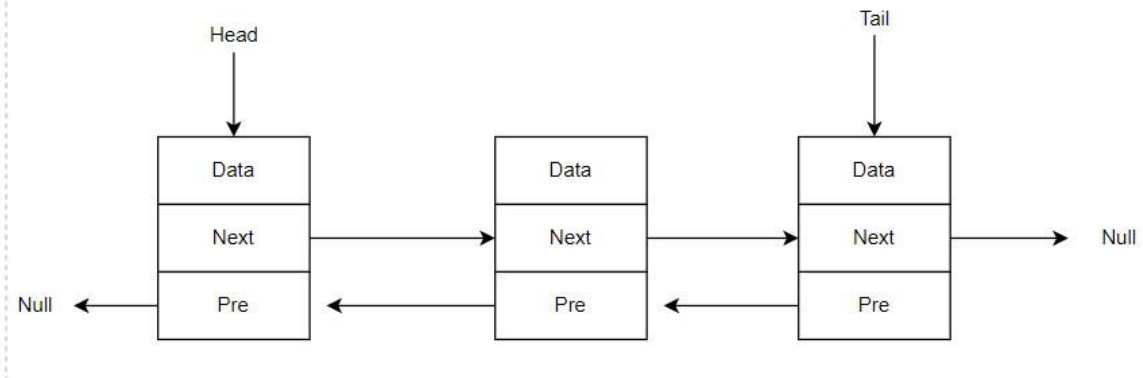
Figure 6: Double LinkedList

b. Operations

- void Enqueue(item): adds a new element to the queue's back end. It requires the item but does not provide a response.
- dequeue() removes the first item from the queue's front. It returns the item with no arguments. The queue has been changed.
- boolean isEmpty: determines whether or not the queue is empty. It has no parameters and produces a boolean value; if the queue is empty, it returns "true."
- int size(): returns the queue's size (rear-front+1). There are no parameters required, and it returns an integer.
- peek(): get the first element. It does not require any arguments. The item is returned. The queue is not changed.

c. Implementation

```
4    static class Node {
5        // Declare the attribute
6        int Data;
7        Node next, prev;
8        // Construct the Node
9        public Node(int Data) {
10            this.Data = Data;
11            Node next = null;
12            Node prev = null;
13        }
```

Similar to Stack, Queue also needs a corresponding Node. Queue's node is a bit different. If the Node of the Stack has Data and the NextNode. Then the Node of the Queue includes Data, Next and Prev. In the original Node class, the author assigned values for Next and prev which were both null. From there, we see that in the Queue function there are not any elements. **Enqueue(int item) :**

```
public void Enqueue(int item) {
    if (isFull()) {
        System.out.println("Queue is Overflow");
        return;
    } else {
        Node newNode = new Node(item);
        // if first insertion head should
        // also point to this node
        if (isEmpty()) {
            head = newNode;
            count++;
        } else {
            tail.next = newNode;
            newNode.prev = tail;
            count++;

        }
        tail = newNode;
    }
}
```

Because it is similar to Stack, Queue also has the same way of adding as that of Stack. The author on lines 27 to 30 has created a test condition in Methol isFull(). In isFull() will be responsible for checking if the number of elements in the array is greater than the allowed element. if greater than the element returns: Queue is Overflow and true. If it is not greater than the given element, it will just return false and continue running the following statements. In line 34, the author put another statement to check if this function is empty or not, if empty, head will point to that element. If the author only adds 1 element. Then Head = newNode.
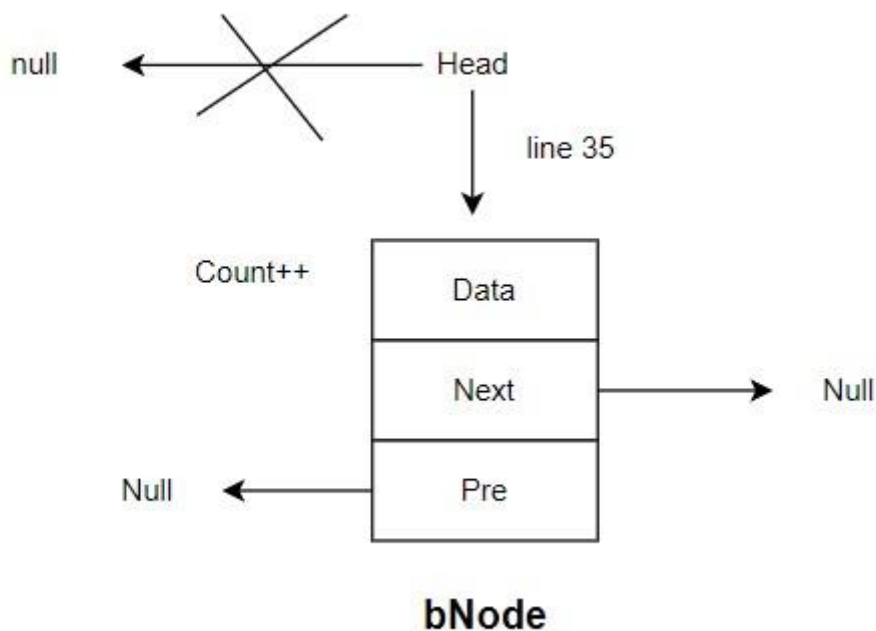

bNode

Figure 7: add 1 data in Queue

```
48⊖     public int Dequeue() {
49          if (head == null) // check if Queue is empty
50          {
51              System.out.println("Queue is Underflow");
52              return -1;// if queue is empty return -1
53          }
54          int temp = head.Data; // create a Current pointer point at head
55          if (head.next == null) // if queue has one element phầ§n
56          {
57              tail = null; // disconnect of the head . pointer
58              count--;
59          } else {
60              // previous of next node (new first) becomes null
61              head.next.prev = null;
62              count--;
63          }
64          head = head.next;
65          return temp;
66      }
```

And in case you want to add elements to the existing elements. Then head will project on the newly added element, tail.next = newNode. newNode.prev = tail and increment count by 1.
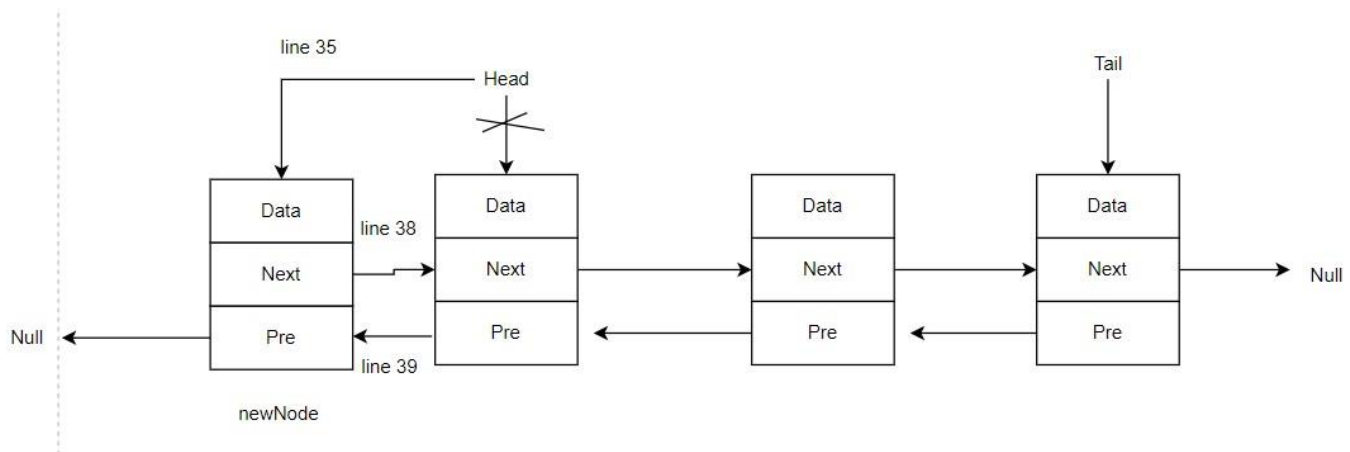

Figure 8: Add 1 data to many data in Queue

- **Dequeuer():**

    In the next action, the Author also creates a function named Dequeue() for the purpose of deleting elements in the array. To delete, the author has put a conditional sentence in line 49, to check if the Head element is empty, if it is empty, the Queue will have no elements to delete. If it is not empty, it will be run to the next statement. In line 54, if there is only 1 element, then the delete statement will be executed as follows: assign the value head.next = null and will also assign the value tail to null. Then the count counter will decrease back to the original value of 0. Removed that element.
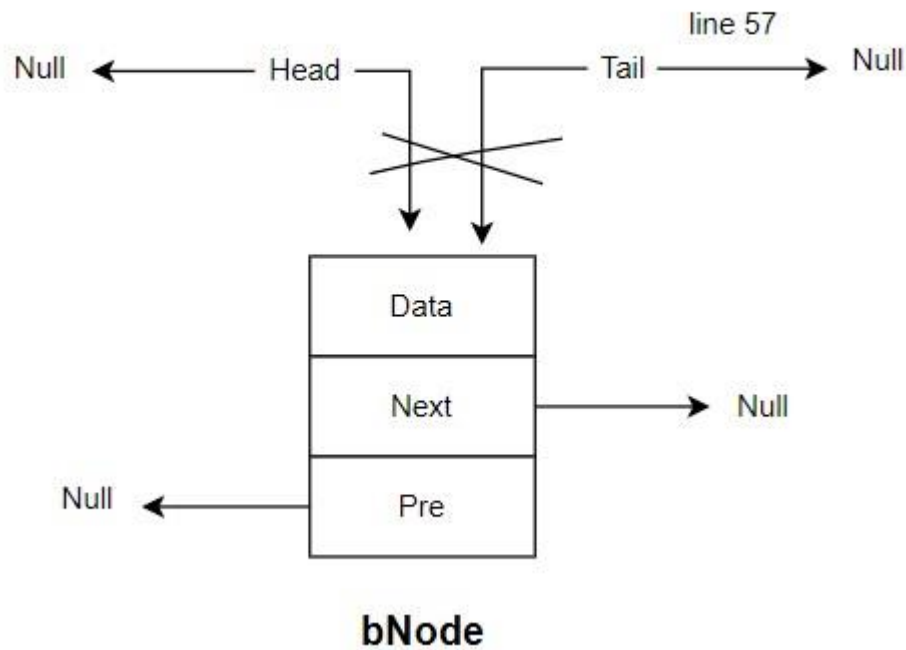
Figure 9: Remove 1 data when have 1 element

If there are 2 or more elements in the array, the Dequeue function will do the following: Head.next.prew will be null.
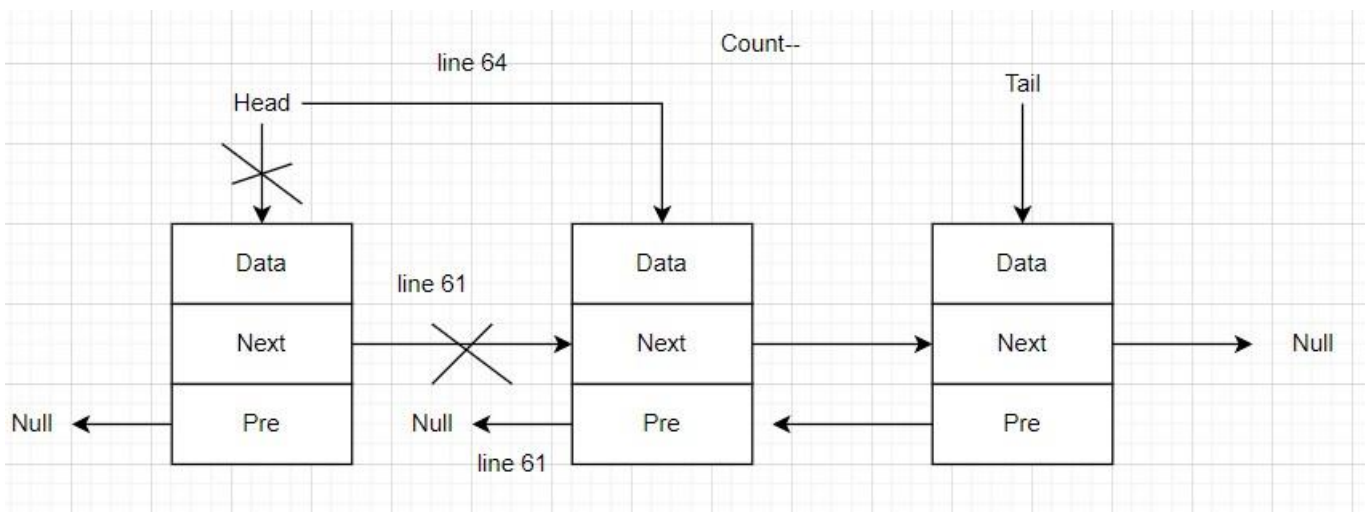

Figure 10: Remove 1 data in many

elements in Queue -

```
public int pickfirst() {
    if (isEmpty())// check if the queue is empty
    {
        System.out.println("Stack is empty");
        return -1;// if queue is empty return -1
    }
    return head.Data;// else return head value
}
```

Method Pickfirst created with the purpose of saving the initial value for the array.

Start method Pickfirst will check if function is empty or not if empty function will return Stack is empty and value -
1.

If the function is not empty, it will return the head value of the array (Head).

- **Size():**

```
80    public int Size() {
81
82        return count; // returns the value of count
83    }
```

The size() method returns a counter function: that counts the elements in the array. This function is quite special in case if the function has no value then it will return zero by itself and don't need to give anything like other methods.

- isFull():

```
111    public boolean isFull() {
112        if (Size() >= MAX_CAPACITY) // if stack is empty re
113                                    // return false
114        {
115            System.out.println("Queue is Overflow");
116            return true;
117
118        }
119        return false;
120    }
121
```

Boolean isFull() method as mentioned above this function helps to check whether the newly added elements in the array exceed the specified Max_Capacity? If it passes will return Queue is overflow and true. Otherwise, it will return False.

II. Implement error handling and report test results (P5)

1. Testing plan

No	Scope	Operation	Testing type	Input	Expected Output	Actual Output	Status
1	Singly Linked List	Push(int data)	Normal	MyStack[2,3,5] push(8);	MyStack[8,2,3,5] size()=4 peek():return 8	The same as expected output	Pass
2		Push(int data)	Normal	MyStack[] push(8);	MyStack[8,] size()=1 peek():return 8	The same as expected output	Pass
3		Push(int data)	Data validation	MyStack[2,3,5] push(@@@);	Printf("Incorrect input")	program stop	Fail
4		Push(int data)	Normal	MyStack[8,2,3,5] Max_element=4 push(10);	MyStack[8,2,3,5] size()=4 peek():return 8 "Overflow"	The same as expected output	Pass
5		Pop()	Normal	MyStack[2,3,5] pop();	MyStack[3,5] pop():return 2 size()=2 peek():return 3	The same as expected output	Pass

6			Normal	MyStack[] pop();	MyStack[] "Underflow"	MyStack[] "Underflow"	Pass

7		peek()	Normal	MyStack[2,3,4] peek();	MyStack[2,3,4] size()=3 peek():return 2	MyStack[2,3,4] size()=3 peek():return 2	Pass
8			Normal	MyStack[] peek();	MyStack[] "Underflow"	MyStack[] "Underflow"	Pass
9		size()	Normal	MyStack[2,3,5] size();	MyStack[2,3,5] size()=3	MyStack[2,3,5] size()=3	Pass
10		isEmpty	Normal	MyStack[2,3,5] isEmpty();	MyStack[2,3,5] return: false	MyStack[2,3,5] return: false	Pass
11		isEmpty()	Normal	MyStack[] isEmpty();	MyStack[] return: true	MyStack[] return: true	Pass
12		Constrain	Normal	Mystack[2,3,5] Constrain(2)	Mystack[2,3,5] Return true	Mystack[2,3,5] Return true	Pass
13		Constrain	Normal	Mystack[2,3,5] Constrain (6)	Mystack[2,3,5] Return false	Mystack[2,3,5] Return false	Pass
14	Double LinkedList	Enqueuer()	Normal	Myqueue[2,3,5] enqueue(1);	Myqueue[1,2,3,5] size():4 peek(): return 5	Myqueue[1,2,3,5] size():4 peek(): return 5	Pass
15		Enqueuer()	Normal	Myqueue[8,2,3,5] Max_element=4 enqueue(10);	MyStack[8,2,3,5] size()=4 peek():return 8 "Overflow"	MyStack[8,2,3,5] size()=4 peek():return 8 "Overflow"	Pass
16		Enqueuer()	Data validation	Myqueue[2,3,5] enqueue(@);	Printf("Incorrect input")	program stop	Fail

17		Dequeue	Normal	Myqueue[2,3,5] dequeue();	Myqueue[2,3] size():2 peek(): return 3 dequeue():return5	Myqueue[2,3] size():2 peek(): return 3 dequeue():return5	Pass
18		Dequeue	Normal	Myqueue[] dequeue();	Myqueue[] "Underflow"	Myqueue[] "Underflow"	Pass
19		peek()	Normal	Myqueue[] peek();	Myqueue[] "Underflow"	Myqueue[] "Underflow"	Pass
20		peek()	Normal	Myqueue[4,5,7] peek()	Myqueue[4,5,7] peek(): return 7	Myqueue[4,5,7] peek(): return 7	Pass
21		size()	Normal	Myqueue[4,5,7] size();	Myqueue[4,5,7] size():3	Myqueue[4,5,7] size():3	Pass
22		isEmpty()	Normal	Myqueue[4,5,7] isEmpty()	Myqueue[4,5,7] Return: false	Myqueue[4,5,7] Return: false	Pass
23		isEmpty()	Normal	Myqueue[] isEmpty();	Myqueue[] Return: True	Myqueue[] Return: True	Pass

2. Evaluation

In the push() and enqueue() sections of the Stack ADT and Queue ADT, there are 22 tests implemented and two failed tests.

The push() and enqueue() actions went successfully, however both methods failed to fulfill expectations when it came to input data validation. To fix the problem, I'll write a function that checks if the input data is valid or not; if it is, it will print "Input Invalid," and if it isn't, it will print "Incorrect input."

Also The author will develop method constraint for next batch. To be able to search for the element and display that element and also how much the position of the word is.

III. Discuss how asymptotic analysis can be used to assess the effectiveness of an algorithm (P6)

1. Definition

Asymptotic notations, according to Iqbal (2019), can be used to describe the effectiveness of a function. They're frequently used in math and computer science, particularly in the areas of analytical number theory, combinatory, and computational complexity while studying algorithms.

Bachmann (1984), a number theorist, suggested the Big-O notation, or simply O-notation, as the first asymptotic notation. Its characteristics and physical interpretation grew in popularity over time, and it was widely used in algorithm analysis. In 1909, Landau introduced Little-O notation, and in 1976, Knuth introduced Big-, Big-, and Little-notations.

Each notation has a specific application that can be summarized as follows:
• The symbol Big O is used to denote the worst case asymptotic upper bound.
• The term "little O" refers to an upper-bound that cannot be tightened.
• For the best scenario, big is used to express the asymptotic lower bound and represented.
• A lower-bound that cannot be tight is referred to as a little.
• The term "big-" is used to define both the upper and lower bounds, and it is used to denote the typical scenario.

The author will explain the Big O notation and why it can be used to evaluate the success of an algorithm in this assignment.

2. Asymptotic notations

Asymptotic notation is a mathematical technique for expressing the temporal complexity of algorithms in asymptotic analysis. Some of the most fundamental asymptotic notations for determining an algorithm's running time complexity are included below.

Big On Notation (O): It is the formal term for an algorithm's maximum running time. It calculates the worst-case time complexity, or how long an algorithm will take to finish in the worst-case scenario.
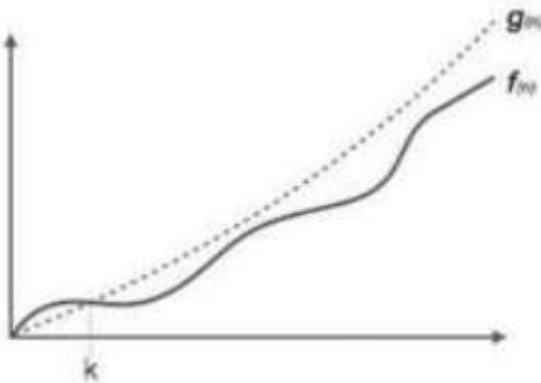

Figure 11: Example for Big On Notation (Tutorialspoint.com. 2021).

For example, for a function f(n)

$$O(f(n)) = \{g(n): \text{there exists } c > 0 \text{ and } n_0 \text{ such that } f(n) \le c.g(n) \text{ for all } n > n_0.\}$$

Θ Notation: A notation is a method of using symbols or signs as a means of communication or a quick comment. A chemist may employ the symbol AuBr to indicate gold bromide......(uncountable) The operation, process, form, or instance by which symbols or signs, figures, or characters are represented in a system or collection.
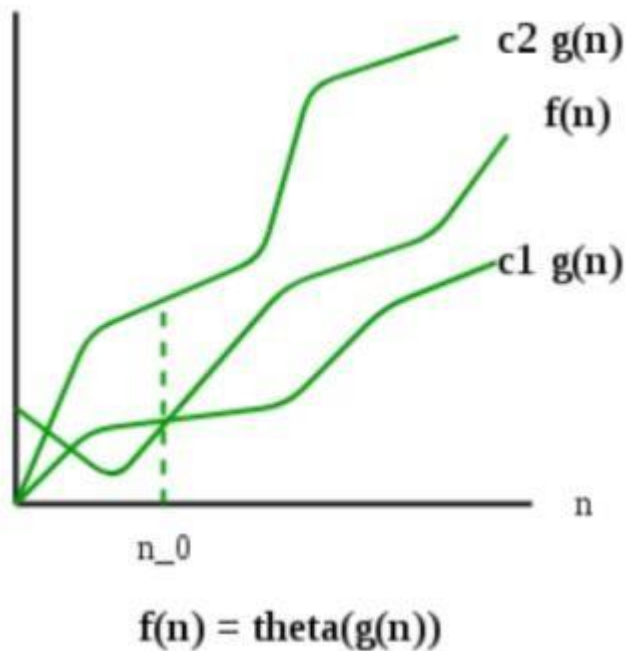
Figure 12: : Θ Notation (geeksforgeeks, 2021)

For example: As an example, consider the following phrase.

$3n3 + 6n2 + 3n3 = 6000$ (n3)
Dropping down lower order terms is always beneficial because, regardless of the constants involved, there will always be a number(n) with larger values than (n2) (n2).

For a given function g, we refer to a number of features (g(n)) (n)

```
Θ(g(n)) = {f(n): there exist positive constants c1, c2 and n0 such
              that 0 <= c1*g(n) <= f(n) <= c2*g(n) for all n >= n0}
```

Ω **Notation**: The bottom bound of an algorithm's execution time is represented by the omega notation. As a result, it offers the algorithm's best-case complexity.

If there is a positive constant c that lies above cg(n) for sufficiently big n, the preceding equation can be represented as a function f(n) belonging to the set (g(n)). Omega(g(n)) gives the shortest time required by the method for any value of n.
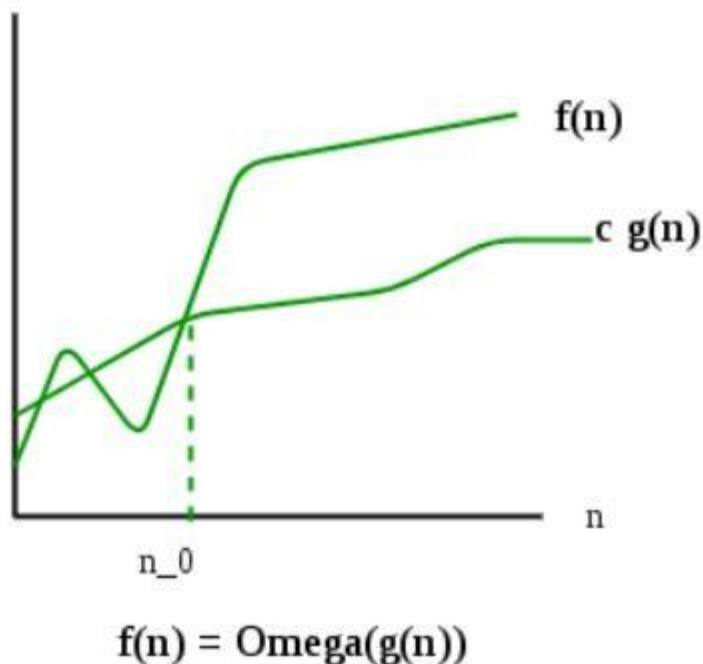
$$f(n) = Omega(g(n))$$

Figure 13: Omega gives the lower bound of a function (GeeksforGeeks, 2021)

3. Examples about big O notation.

Notation	Name	Example

O(1)	Constraint	Specifies an algorithm that, regardless of the size of the input data set, will always execute in the same amount of time (or space).
		
		 ``java
		public void getHead() {
		    if (isEmpty())// check if the queue is empty
		    {
		        System.out.println("Queue is empty");
		        return;// if queue is empty return myInf
		    }
		    System.out.println("the first element is:" + head.Data);// else return head va
		}
		``

O(n)	Linear	print All Operation in Queue and Stack implementation

| | | ```
 96     // create operation Print()
 97⊖    public void Print() {
 98         // create a Current pointer point at firstNode
 99         Node current = firstNode;
100         System.out.print("[");
101         // stop when running to second last of the Stack
102         while (current != null) {
103             // print out satisfactory values
104             System.out.print(current.Data + ",");
105             // move current to the next Node
106
107             current = current.NextNode;
108         }
109         // print the value of the last element
110         System.out.print("]");
111     }
112     // create operation peek()
113
114⊖    public int Peek() {
115         if (isEmpty()) {
116             System.out.println("Stack is empty");
117             return myInf;
118         }
119         return firstNode.Data;
120     }
``` |
| O(logn) | logarithmic time | Essentially, time advances linearly while n advances exponentially. So, if computing 10 elements takes 1 second, computing 100 elements takes 2 seconds, computing 1000 elements takes 3 seconds, and so on.<br><br>```
cout = 0;
for (var i = 1; i< n; i = i *2){
    count++;
}
return count;
``` |
| O(n!) | Factorial | ```
public static void factorial(int n){
    for (int i= 0; i<n; i++){
        factorial(n-1);
    }
}
``` |
| O(n²) | quadratic time | selection sort algorithm<br><br>```
int i, j, min_idx;

// One by one move boundary of unsorted subarray
for (i = 0; i < n-1; i++)
{
    // Find the minimum element in unsorted array
    min_idx = i;
    for (j = i+1; j < n; j++)
    if (arr[j] < arr[min_idx])
        min_idx = j;

    // Swap the found minimum element with the first element
    int temp=arr[min_idx];
    arr[min_idx]=arr[i];
    arr[i]=temp;
```<br>- The author will run from 0 to n - 2. For each value of i the inner for loop will run n - i - 1 times (j from i + 1 to n - 1), so the number of comparisons to make is Σ(n - i - 1) |

		(where i runs from 0 to n - 2) = (n - 1) + (n - 2)+ (n - 3) +...+ 2 + 1 = (n - 1)*n/2 = n²/2 - n/2. My our team say the complexity of the above algorithm is O(n²)

The author will demonstrate the difference in the complexity of the notations in the following figure.

Big-O Complexity Chart

Algorithms that run in O(2n) and O(n!) can only handle very small datasets, as we can see. O(2n), or exponential time, doubles in length with each addition to the input. O(n!), or factorial time, is even worse. The running time increases by a factor of n when n is increased by one.

As a result, it's obvious that every programmer's ultimate goal is to create and develop the most optimal algorithm possible, because the number of operations and complexity can quickly spiral out of control.

IV. **Determine two ways in which the efficiency of an algorithm can be measured, illustrating your answer with an example(P7)**

The complexity of an algorithm is a property that indicates its efficiency in terms of the quantity of data it must handle. This function's domain and range are generally stated in natural units. There are two main complexity tests for determining the efficiency of an algorithm.

Space complexity: is a function that describes how much memory (space) an algorithm uses in terms of the quantity of data it receives. We talk about "additional" memory needs when we don't include the RAM needed to store the input. To do so, we'll utilize regular (but fixed-length) units once more. We can use bytes, but metrics like the number of integers utilized, the number of fixedsize structures, and so on are handier. Finally, the amount of bytes required to represent the unit will have no bearing

on the feature we develop. Space complexity is sometimes ignored since the space utilized is little and/or evident, yet it may be just as important as time. (educative, 2021)

For example:

```java
public int sumArray(int[] array)
{
    int size = 0;
    int sum = 0;
    for (int i = 0 ; i <size ; i++) {
        sum += array[i];
    }
    return sum;
}
```

As stated above, three integer variables are used. Depending on the program's compilation, the integer size is 2 or 4 bytes. Let's suppose its two bytes. As a result, the total amount of space used in the given software is $2 * 3 = 6$ bytes. Because there are no other variables, no spaces are required. Without getting into the temporal complexity, the preceding code requires array storage and n items. As a result, this code has a Space Complexity of O. (n)

Time complexity: is a function that defines how long it takes an algorithm to process a given quantity of data. The term "time" can refer to the number of memory accesses, integer comparisons, the number of times an inner loop is run, or any other natural measure that represents how long the algorithm would take in real time. We attempt to distinguish this idea of time from "wall clock" time, because actual time can be influenced by a number of variables unrelated to the algorithm (like the language used, type of computing hardware, proficiency of the programmer, optimization in the compiler, etc.). All of the other elements, it turns out, will fall into place if we pick our units properly. It turns out that if we pick the right units, the rest doesn't matter, and we can derive an independent measure of the algorithm's efficiency. (educative, 2021)

```java
for (int i = 1 ; i <= n; i++)
{
    for ( int j = 1 ; j <= n ; j++) {
        System.out.println("Hello World" + i + "and" + j);
    }
}
```

There are two loops visible. For the first loop, perform n steps; for loop 2, run n steps as well. As a result, the temporal complexity will be O. (n2)

V.   Reference

Bibliography

educative, 2021. *Time complexity vs. space complexity.* [Online]
Available at: https://www.educative.io/edpresso/time-complexity-vs-space-complexity [Accessed 9 4 2021].

GeeksforGeeks, 2021. *Time-Space Trade-Off in Algorithms.* [Online]
Available at: https://www.geeksforgeeks.org/time-space-trade-off-in-algorithms/ [Accessed 9 4 2021].