ASSIGNMENT 2 FRONT SHEET

Qualification	**BTEC Level 5 HND Diploma in Computing**		
Unit number and title	Unit 19: Data Structures and Algorithms		
Submission date		**Date Received 1st submission**	
Re-submission Date		**Date Received 2nd submission**	
Student Name	Lê Minh Anh	**Student ID**	GCH190017
Class	GCH0801	**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	Anh

Grading grid

P4	P5	P6	P7	M4	M5	D3	D4

❐ Summative Feedback:		❐ Resubmission Feedback:
Grade:	Assessor Signature:	Date:
Internal Verifier's Comments:		
IV Signature:		

Contents

Introduction

In this report, Queue and Stack features in programming will be displayed and explained from coding. I will also give definition, measure the effectiveness of the algorithm.

I. Design

1. Stack

Stack is a linear data structure. Its data can be accessed only at one of its ends for storing and retrieving data. A stack is called a LIFO structure: Last In First Out.

Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called PUSH operation and removal operation is called POP operation.

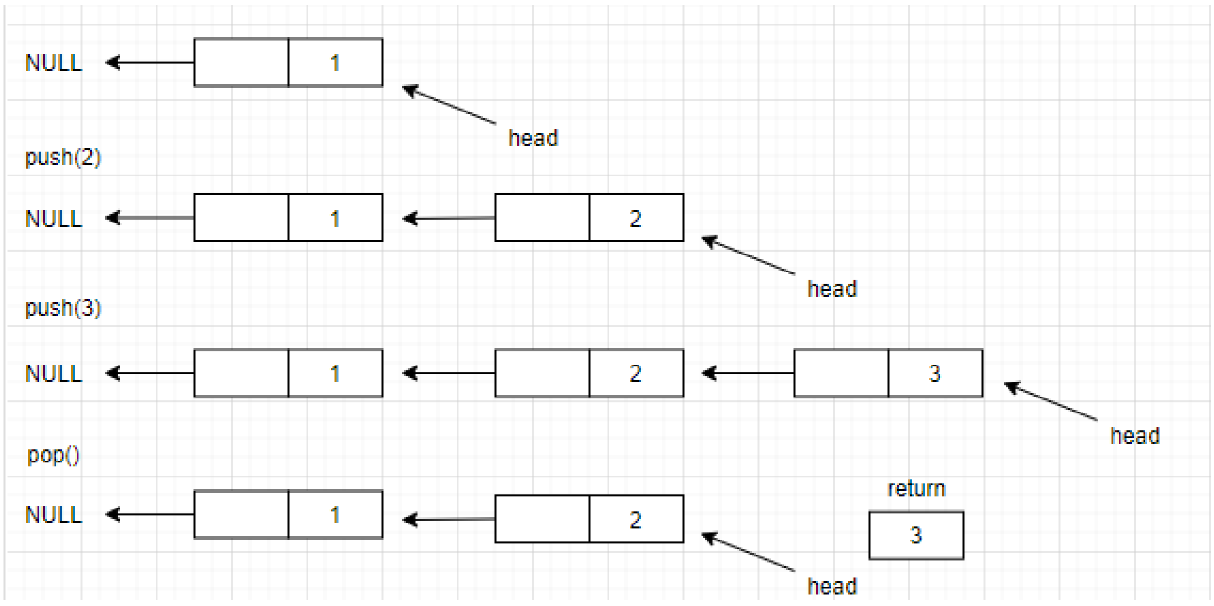


Figure 1: Stack Design

2.Queue

A queue is simple a list that grows by adding elements to its end and shrinks by taking elements from its front. Unlike a stack, queue is a structure in which both ends are used. The last element has to wait until all elements preceding it on the queue are removed. A queue is a FIFO structure: First In First Out.

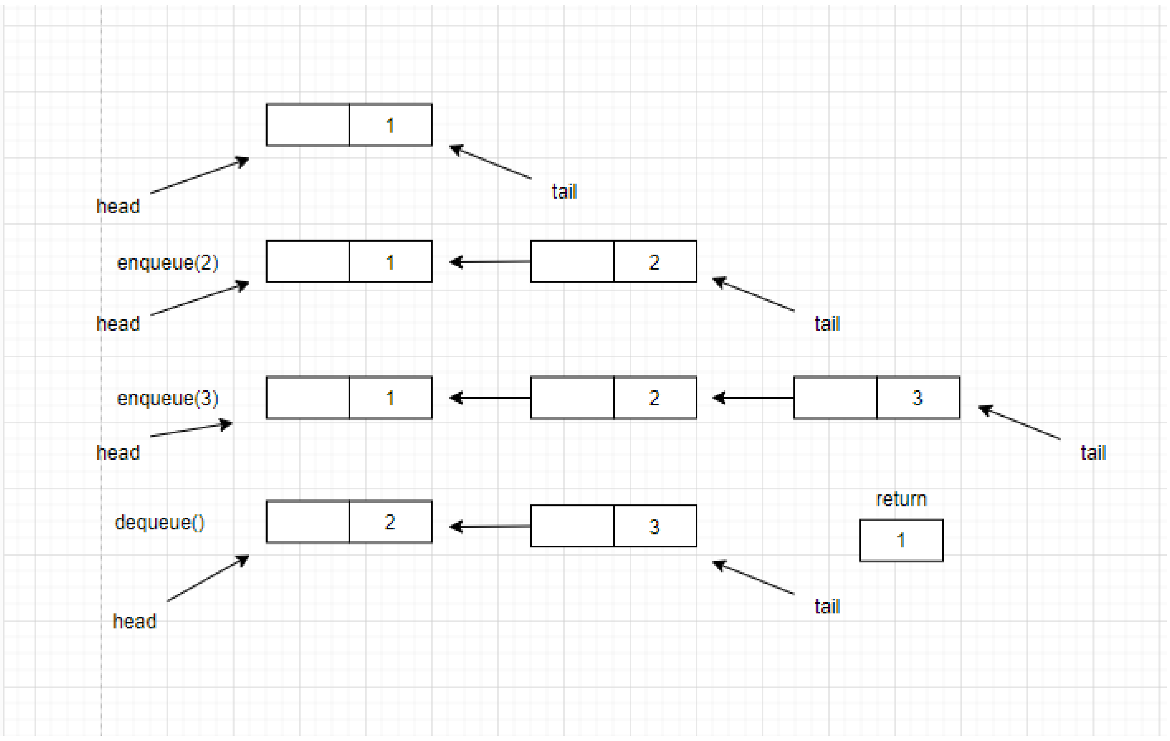
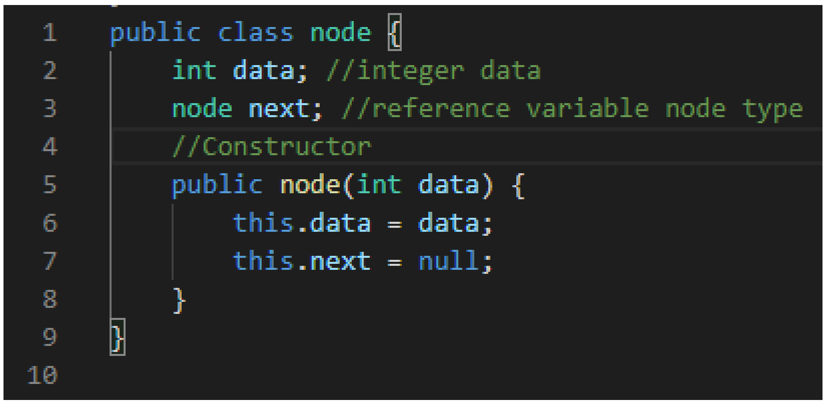


Figure 2: Queue Design

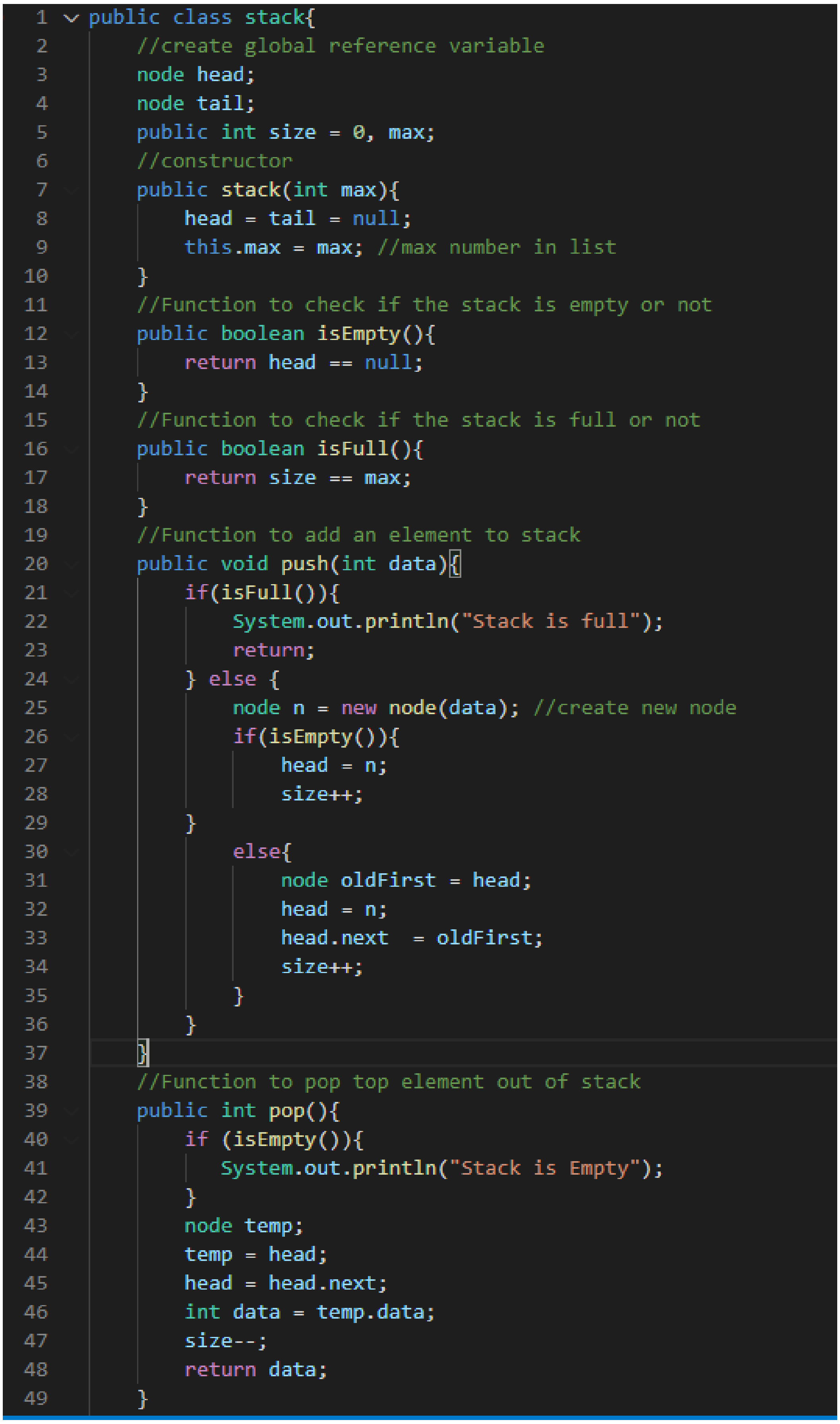
II. Application

Code & explanation



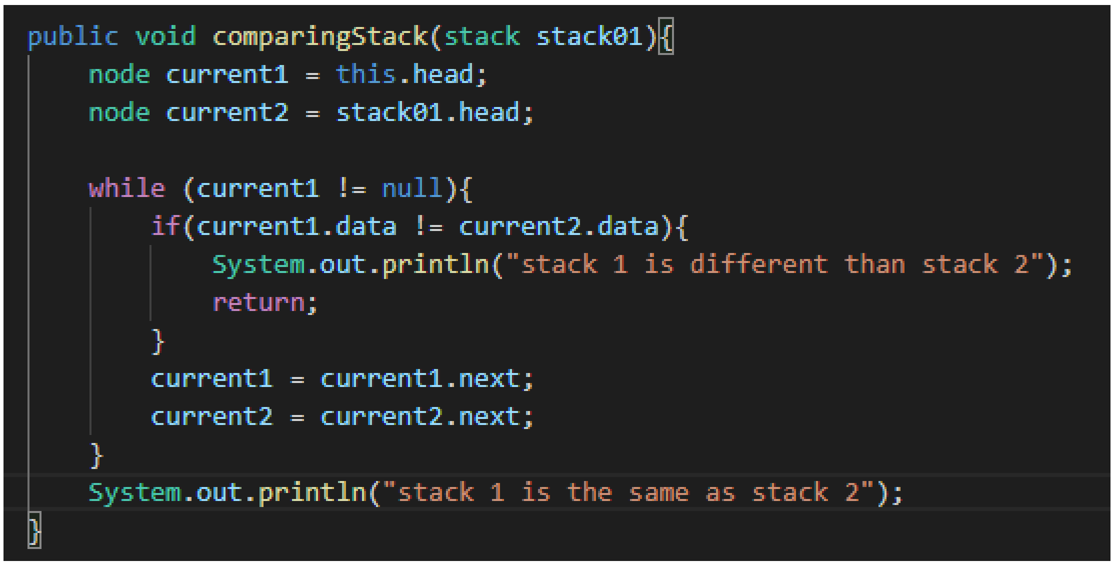
```java
public class node {
    int data; //integer data
    node next; //reference variable node type
    //Constructor
    public node(int data) {
        this.data = data;
        this.next = null;
    }
}

```

Figure 3: Node



```java
public class stack{
    //create global reference variable
    node head;
    node tail;
    public int size = 0, max;
    //constructor
    public stack(int max){
        head = tail = null;
        this.max = max; //max number in list
    }
    //Function to check if the stack is empty or not
    public boolean isEmpty(){
        return head == null;
    }
    //Function to check if the stack is full or not
    public boolean isFull(){
        return size == max;
    }
    //Function to add an element to stack
    public void push(int data){
        if(isFull()){
            System.out.println("Stack is full");
            return;
        } else {
            node n = new node(data); //create new node
            if(isEmpty()){
                head = n;
                size++;
        }
            else{
                node oldFirst = head;
                head = n;
                head.next  = oldFirst;
                size++;
            }
        }
    }
    //Function to pop top element out of stack
    public int pop(){
        if (isEmpty()){
          System.out.println("Stack is Empty");
        }
        node temp;
        temp = head;
        head = head.next;
        int data = temp.data;
        size--;
        return data;
    }
```

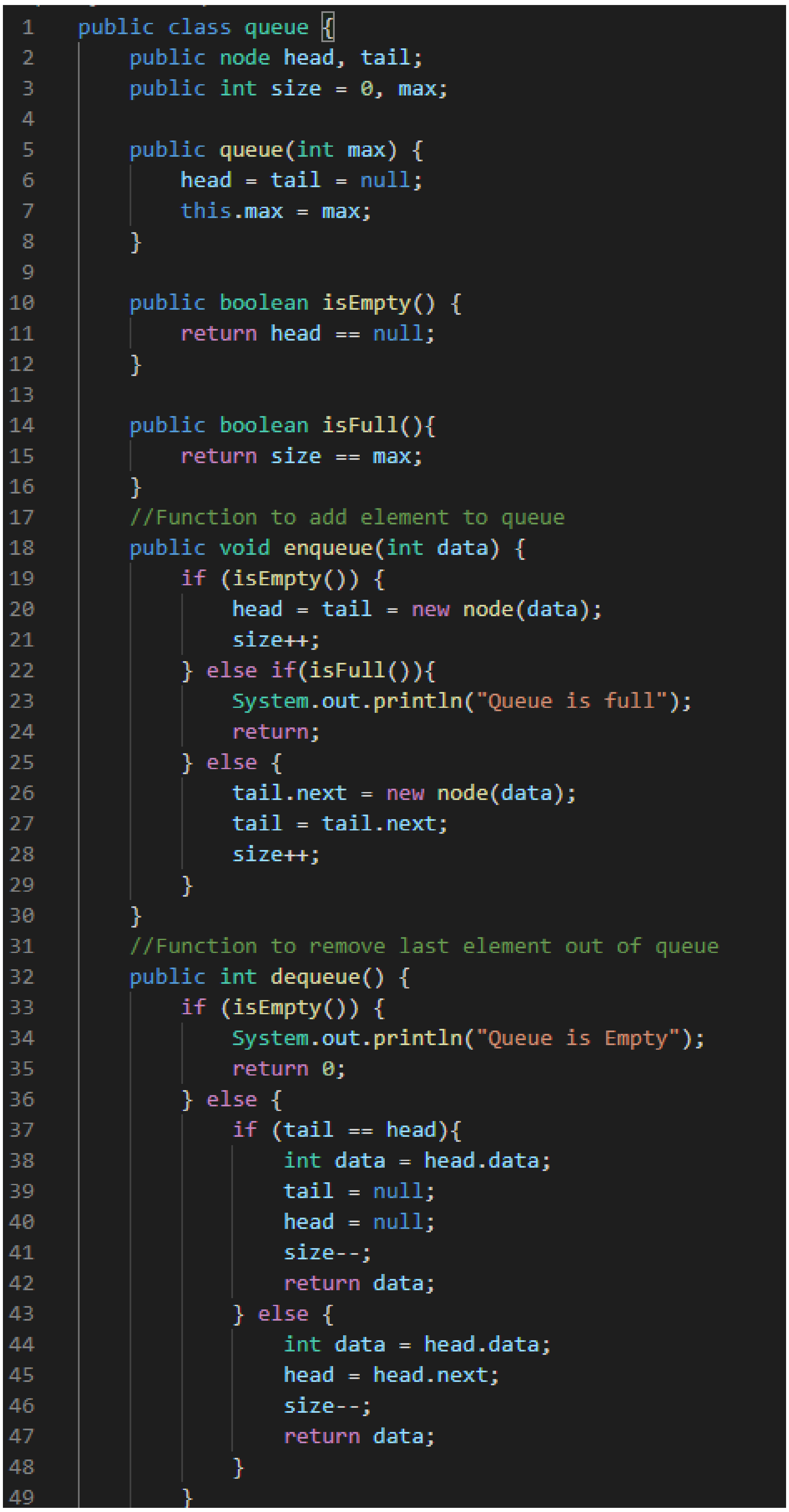
Figure 4: Stack



```java
public void comparingStack(stack stack01){
    node current1 = this.head;
    node current2 = stack01.head;

    while (current1 != null){
        if(current1.data != current2.data){
            System.out.println("stack 1 is different than stack 2");
            return;
        }
        current1 = current1.next;
        current2 = current2.next;
    }
    System.out.println("stack 1 is the same as stack 2");
}
```

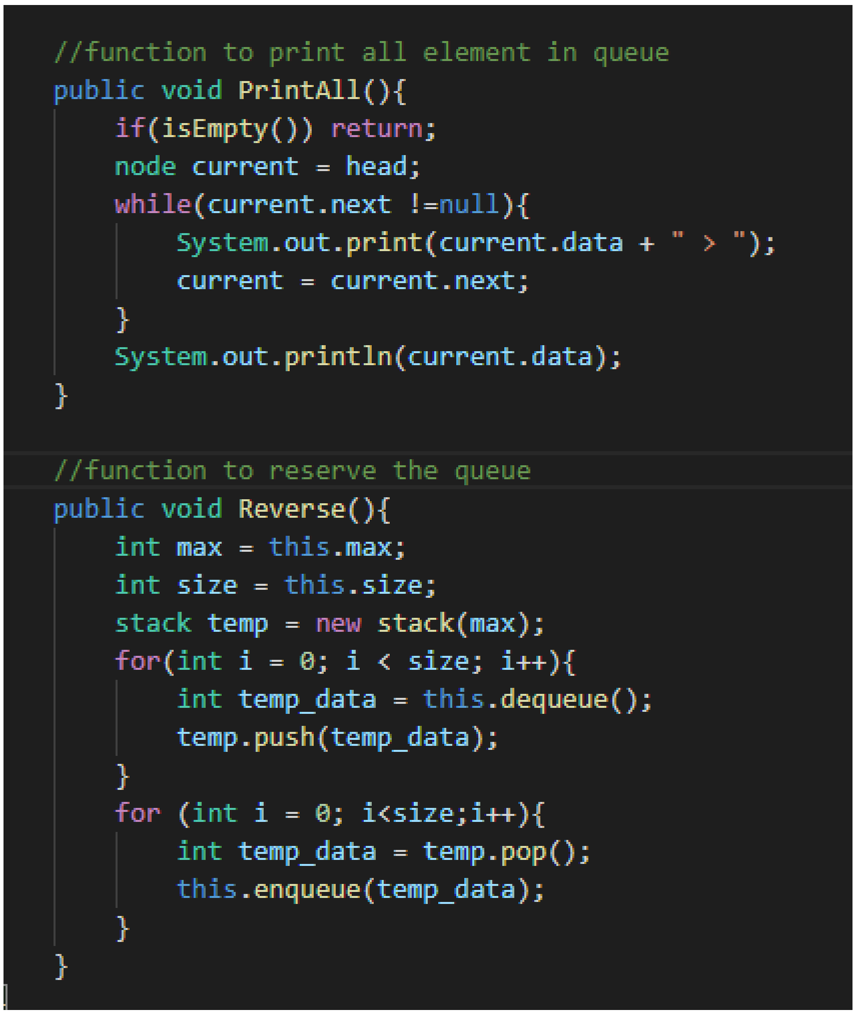
Figure 5: Another function in stack

In the comparingStack() function pass a stack, then create two nodes current1 and current2 and assign them by stack01 and stack01.top respectively. Then compare each element to each other. If the same, then print "stack 1 is the same as stack 2", if there are different elements, print "stack 1 is different from stack 2".



```java
public class queue {
    public node head, tail;
    public int size = 0, max;

    public queue(int max) {
        head = tail = null;
        this.max = max;
    }

    public boolean isEmpty() {
        return head == null;
    }

    public boolean isFull(){
        return size == max;
    }
    //Function to add element to queue
    public void enqueue(int data) {
        if (isEmpty()) {
            head = tail = new node(data);
            size++;
        } else if(isFull()){
            System.out.println("Queue is full");
            return;
        } else {
            tail.next = new node(data);
            tail = tail.next;
            size++;
        }
    }
    //Function to remove last element out of queue
    public int dequeue() {
        if (isEmpty()) {
            System.out.println("Queue is Empty");
            return 0;
        } else {
            if (tail == head){
                int data = head.data;
                tail = null;
                head = null;
                size--;
                return data;
            } else {
                int data = head.data;
                head = head.next;
                size--;
                return data;
            }
        }
```

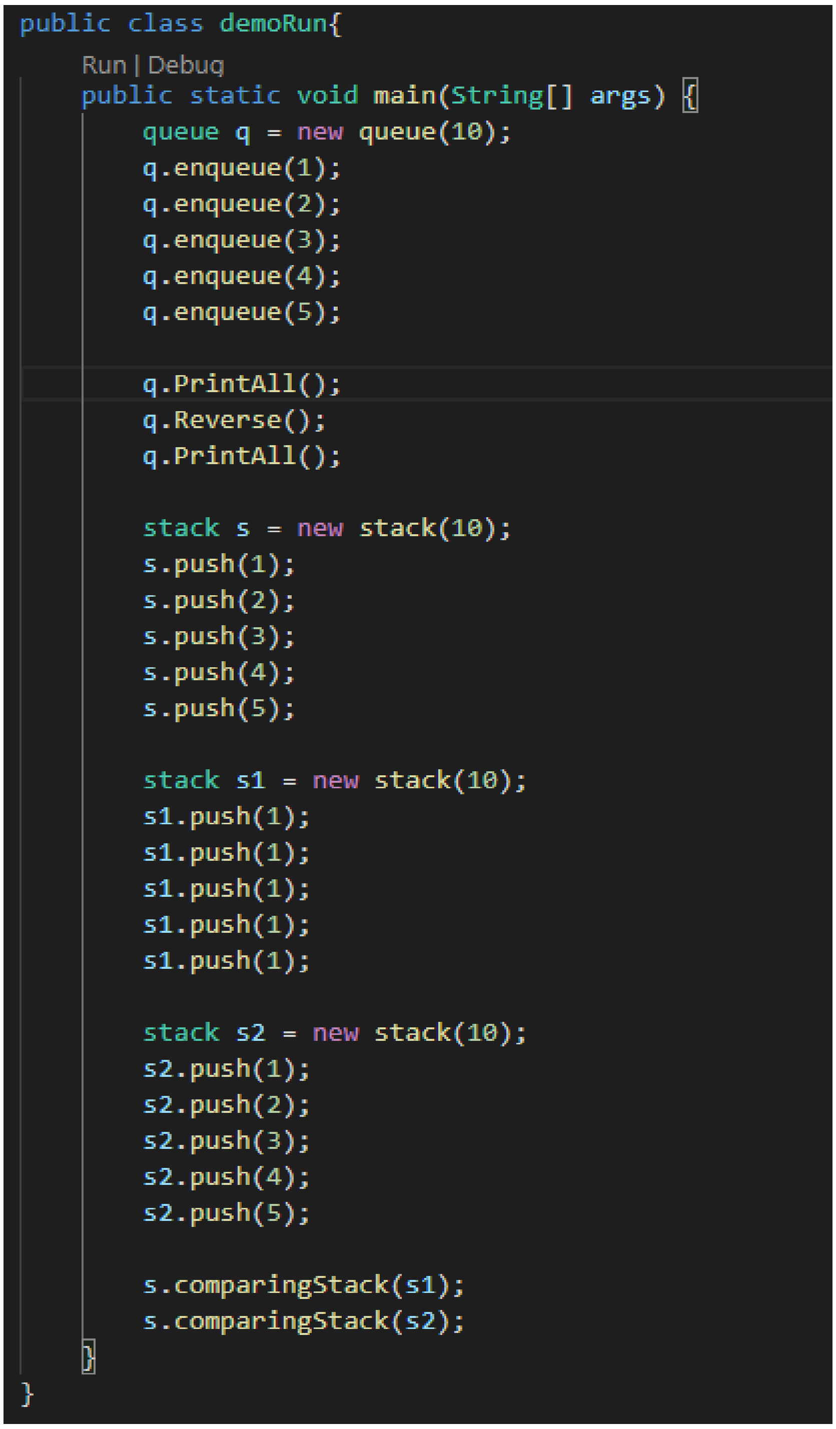
Figure 6: Queue



```java
//function to print all element in queue
public void PrintAll(){
    if(isEmpty()) return;
    node current = head;
    while(current.next !=null){
        System.out.print(current.data + " > ");
        current = current.next;
    }
    System.out.println(current.data);
}

//function to reserve the queue
public void Reverse(){
    int max = this.max;
    int size = this.size;
    stack temp = new stack(max);
    for(int i = 0; i < size; i++){
        int temp_data = this.dequeue();
        temp.push(temp_data);
    }
    for (int i = 0; i<size;i++){
        int temp_data = temp.pop();
        this.enqueue(temp_data);
    }
}
```

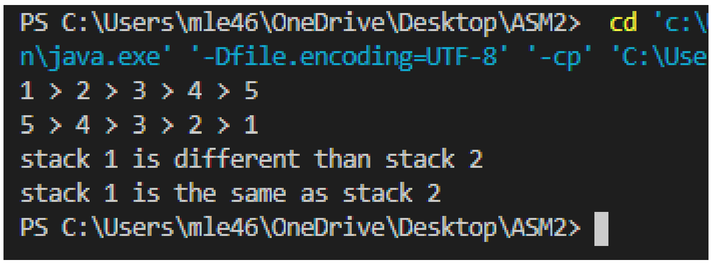
Figure 7: Other functions in queue

In the Reverse() function creates a new stack with the same max value, the size = size of the queue. For the first loop, the queue value will be dequeue and then pushed onto the temporary stack. The second loop will pop all the values in the stack and then enqueue into the queue.



```
public class demoRun{
    Run | Debug
    public static void main(String[] args) {
        queue q = new queue(10);
        q.enqueue(1);
        q.enqueue(2);
        q.enqueue(3);
        q.enqueue(4);
        q.enqueue(5);

        q.PrintAll();
        q.Reverse();
        q.PrintAll();

        stack s = new stack(10);
        s.push(1);
        s.push(2);
        s.push(3);
        s.push(4);
        s.push(5);

        stack s1 = new stack(10);
        s1.push(1);
        s1.push(1);
        s1.push(1);
        s1.push(1);
        s1.push(1);

        stack s2 = new stack(10);
        s2.push(1);
        s2.push(2);
        s2.push(3);
        s2.push(4);
        s2.push(5);

        s.comparingStack(s1);
        s.comparingStack(s2);
    }
}
```

Figure 8: Main



```
PS C:\Users\mle46\OneDrive\Desktop\ASM2>  cd 'c:\
n\java.exe' '-Dfile.encoding=UTF-8' '-cp' 'C:\Use
1 > 2 > 3 > 4 > 5
5 > 4 > 3 > 2 > 1
stack 1 is different than stack 2
stack 1 is the same as stack 2
PS C:\Users\mle46\OneDrive\Desktop\ASM2>
```

Figure 9: Result

III. Implement error handling and report test results

1. Testing plan

No	Scope	Operation	Testing type	Input	Expected Outcome	Actual Outcome	Status
1	Queue	Enqueue()	Normal	Queue: [] Enqueue(1)	Queue: [1] Size = 1	Queue: [1] Size = 1	passed
2		Dequeue()	Normal	Queue: [5,4,3,2,1] Dequeue()	Queue: [5,4,3,2] Size = 4	Queue: [5,4,3,2] Size = 4	passed
3		Enqueue()	Data validation	Queue: [5,4,3,2,1] Size = 5 Enqueue(6)	Queue is full	Queue is full	passed
4		Dequeue()	Data validation	Queue: [] Size = 0 Dequeue	Queue is Empty	Queue is Empty	passed
5	Stack	Push()	Normal	Stack: [] Push(5)	Stack: [5] Size = 1	Stack: [5] Size = 1	passed
6		Push()	Data validation	Stack: [1,2,3,4,5] Size = 5 Push(6)	Stack is full	Stack is full	passed
7		Pop()	Normal	Stack:[1,2,3,4,5] Pop()	Stack: [2,3,4,5] Size = 4	Stack: [2,3,4,5] Size = 4	passed
8		Pop()	Data validation	Stack: [] Pop()	“Stack is Empty”	“Stack is Empty”	passed
9	Question a	Reverse()	Normal	Queue: [1,2,3,4] Reverse()	Queue:[4,3,2,1]	Queue:[4,3,2,1]	passed
10	Question a	Reverse()	Data validation	Queue: [] Reverse()	Queue: []	Queue: []	Passed
11	Question b	Comparing Stack()	Normal	S1: [1,2,3,4,5] S2: [1,2,3,4,5] comparingStack()	Stack 1 is the same as stack 2	Stack 1 is the same as stack 2	passed
12	Question b	Comparing Stack()	Normal	S1: [1,2,3,4,5] S2: [1,1,1,1,1]	Stack 1 is different than stack 2	Stack 1 is different than stack 2	passed

2. Evaluation

After few test cases, the system seems to be ready to use. No error is found and everything is working as expected.

IV. Discuss how asymptotic analysis can be used to assess the effectiveness of an algorithm

Usually, when evaluating the running time or space usage of programs, we try to measure time or space as a function of the input size. For example, when estimating the worst-case running time of a function that sorts a list of numbers, we'll be concerned with how long it takes, based on the length of the input list. (Cornell, 2004).

Usually, the time required by an algorithm falls under three types:

- Best Case − Minimum time required for program execution.
- Average Case − Average time required for program execution.
- Worst Case − Maximum time required for program execution.

When we say that an algorithm runs in time O(n), we mean that O(n) is an upper bound on the running time that holds for all inputs of size n. This is called worst-case analysis.

Average-case analysis is a common alternative to worst-case analysis. Here we do not relate the worst-case running time, but try to measure the estimated time spent on a randomly selected input. This form of analysis is usually more complicated, as it includes probabilistic arguments and often requires assumptions about the distribution of inputs that may be difficult to explain. On the other hand, it can be more useful, since often the worst-case behavior of an algorithm is misleadingly negative. A good example of this is the famous quicksort algorithm, whose worst-case running time on an input sequence of length n is proportional to n^2 but whose anticipated run time is proportional to n log n.

1. Big-O notation

In computer science, Big O notation is used to describe the performance or complexity of an algorithm. Big O explicitly defines the worst-case scenario and can be used to represent the time of execution needed or the space used (e.g. in memory or on disk) by an algorithm (Bell, 2009).

Big O notation characterizes functions according to their growth rates: different functions with the same growth rate may be represented using the same O notation.

The letter O is used because the growth rate of a function is also referred to as the order of the function. A description of a function in terms of big O notation usually only provides an upper bound on the growth rate of the function.

Below are some common orders of growth:

- O(1) describes an algorithm that will always execute in the same time (or space) regardless of the size of the input data set.
- O(N) describes an algorithm whose performance will grow linearly and in direct proportion to the size of the input data set.
- O(N^2) represents an algorithm whose performance is directly proportional to the square of the size of the input data set. This is common with algorithms that involve nested iterations over the data set. Deeper nested iterations will result in O(N^3), O(N^4) etc.

- O(2^N) denotes an algorithm whose growth doubles with each addition to the input data set. The growth curve of an O(2^N) function is exponential - starting off very shallow, then rising meteorically.

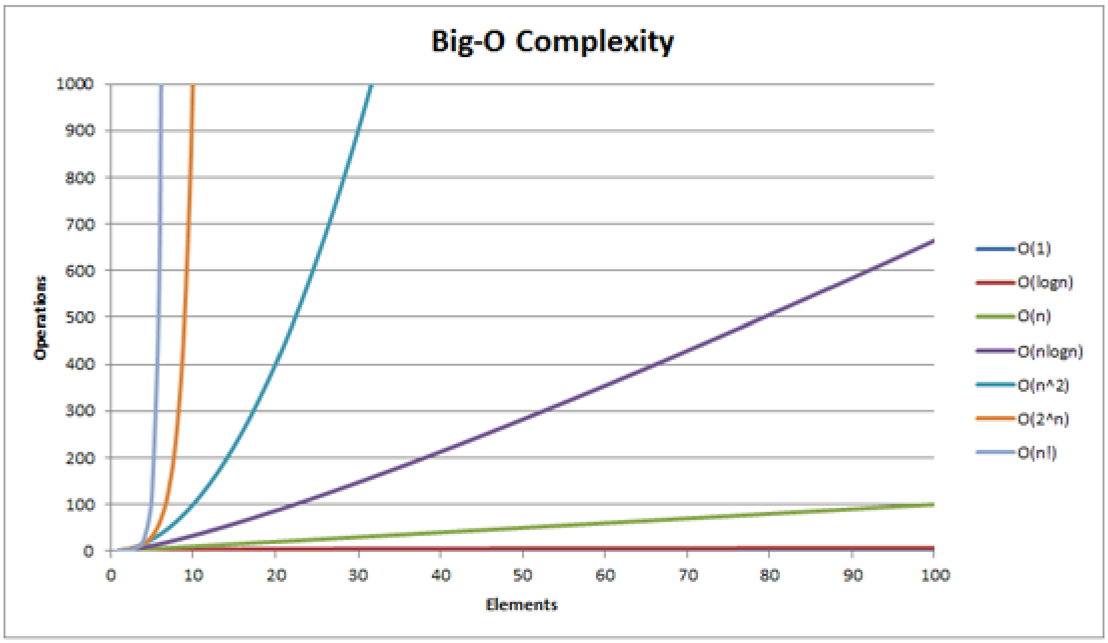


Figure 10: Big-O Complexity

For example:

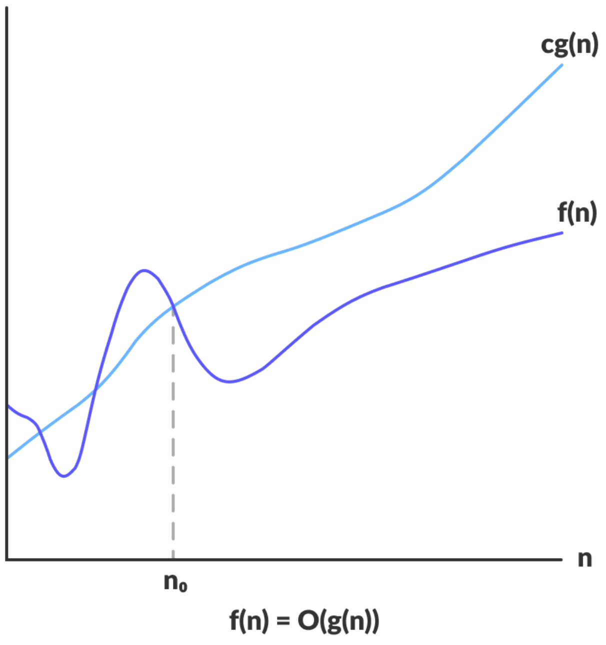


Figure 11: Big-O gives the upper bound of a function (Programiz, 2020)

The above expression can be described as a function f(n) belongs to the set O(g(n)) if there exists a positive constant c such that it lies between 0 and cg(n), for sufficiently large n.

For any value of n, the running time of an algorithm does not cross time provided by O(g(n)).

Since it gives the worst-case running time of an algorithm, it is widely used to analyze an algorithm as we are always interested in the worst-case scenario.

2. Theta notation

Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average case complexity of an algorithm.

For example:

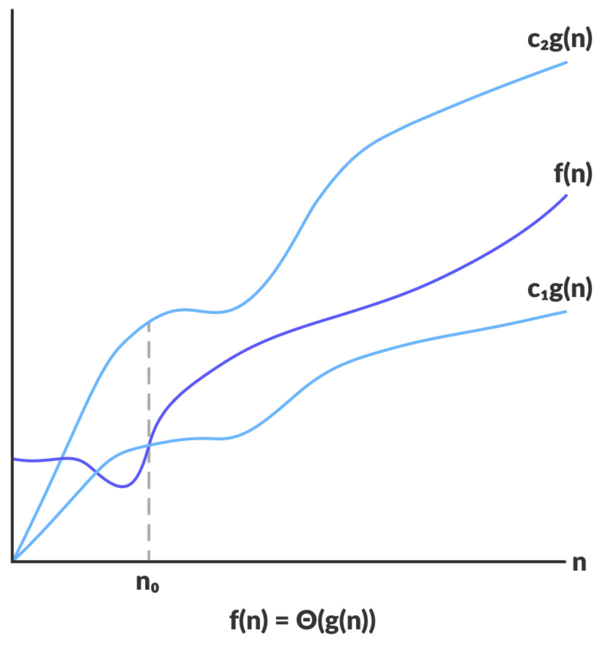


Figure 12: Theta bounds the function within constants factors (Programiz, 2020)

The above expression can be described as a function f(n) belongs to the set Θ(g(n)) if there exist positive constants c_1 and c_2 such that it can be sandwiched between $c_1g(n)$ and $c_2g(n)$, for sufficiently large n.

If a function f(n) lies anywhere in between $c_1g(n)$ and $c_2g(n)$ for all n ≥ n0, then f(n) is said to be asymptotically tight bound.

V. Ways to measure the efficiency of an algorithm

In computer science, algorithmic efficiency is a attribute of an algorithm that refers to the amount of computational resources used by the algorithm. To assess the use of resources, an algorithm must be evaluated and the efficiency of an algorithm can be calculated on the basis of the use of different resources. Algorithmic efficiency can be thought of as equivalent to repeat or continuous process engineering productivity.

For big O, time and space are usually measured to show how the time or run space requirement of an algorithm increases as the input size increases.

- Time complexity: time to perform a task (calculation time or response time)
- Space complexity: data storage (RAM, HDD, etc.) is consumed while performing a certain task.

Example:

For the example, we will compare space and time complexity of Quick Sort and Selection Sort. While Quick Sort in average case takes O(nlogn) time to finish and also take O(logn) in space, Selection sort is much worse in time complexity with O(n^2) but better in space complexity with O(1). So, when you choose which sort algorithm you will use in the program, if time is priority, quicksort is better than selection sort but when space is priority, selection sort is better than quicksort.

Algorithms	Average	Best	Worst	Space
Quick Sort	O(nlogn)	O(nlogn)	O(n^2)	O(logn)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)

Figure 13: Comparison between Quick Sort and Selection Sort (Y.Yang, 2011)

VI. Conclusion

My report showed the Queue and Stack system as well as explained and examined. I also include Big O descriptions and examples and evaluate performance by time and space.

References

Bell, R. (2009, June 23). *A beginner's guide to Big O notation*. Retrieved from rob-bell: https://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/

Cornell, U. (2004). *Lecture 16: Introduction to Asymptotic Analysis.* New York: 2004.

Programiz. (2020). *Asymptotic Analysis*. Retrieved from Programiz: https://www.programiz.com/dsa/asymptotic-notations#:~:text=Theta%20Notation%20(%CE%98%2Dnotation),the%20function%20within%20constants%20factors

Y.Yang, P. a. (2011). Experimental study on the five sort algorithms. *2011 Second International Conference on Mechanic Automation and Control Engineering*, 1314-1317.