ASSIGNMENT 2 FRONT SHEET

Qualification	**BTEC Level 5 HND Diploma in Computing**		
Unit number and title	Unit 19: Data Structures and Algorithms		
Submission date	24/10/2020	**Date Received 1st submission**	
Re-submission Date		**Date Received 2nd submission**	
Student Name	Le Nam Khanh	**Student ID**	GCH18713
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	Khanh

Grading grid

P4	P5	P6	P7	M4	M5	D3	D4

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

Table of Contents

Table of Figures

Introduction

In this report, I will design and implement two data structures: stack and queue. And provide ways to measure and evaluate algorithms.

I. Design

1. Stack

A stack is a linear data structure in which it is possible to access all data stored and retrieved only at one end. The data structure of the stack is Last In First Out (LIFO), all data added to the stack will go to the top, and it will also be taken from the top when that data is removed. If removed, all the data in the stack would be in reverse order when returned to the stack.

Example:

- Stack push:

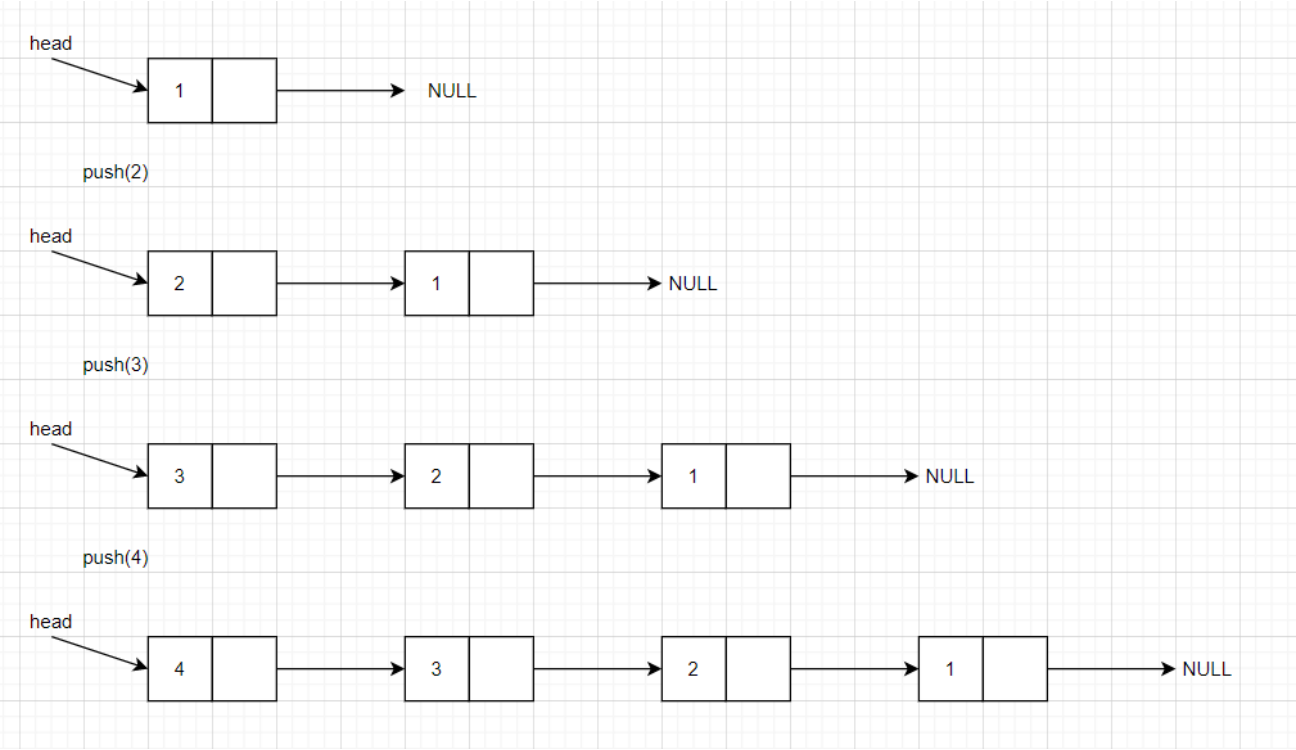

Figure 1: Stack push

- Stack pop:

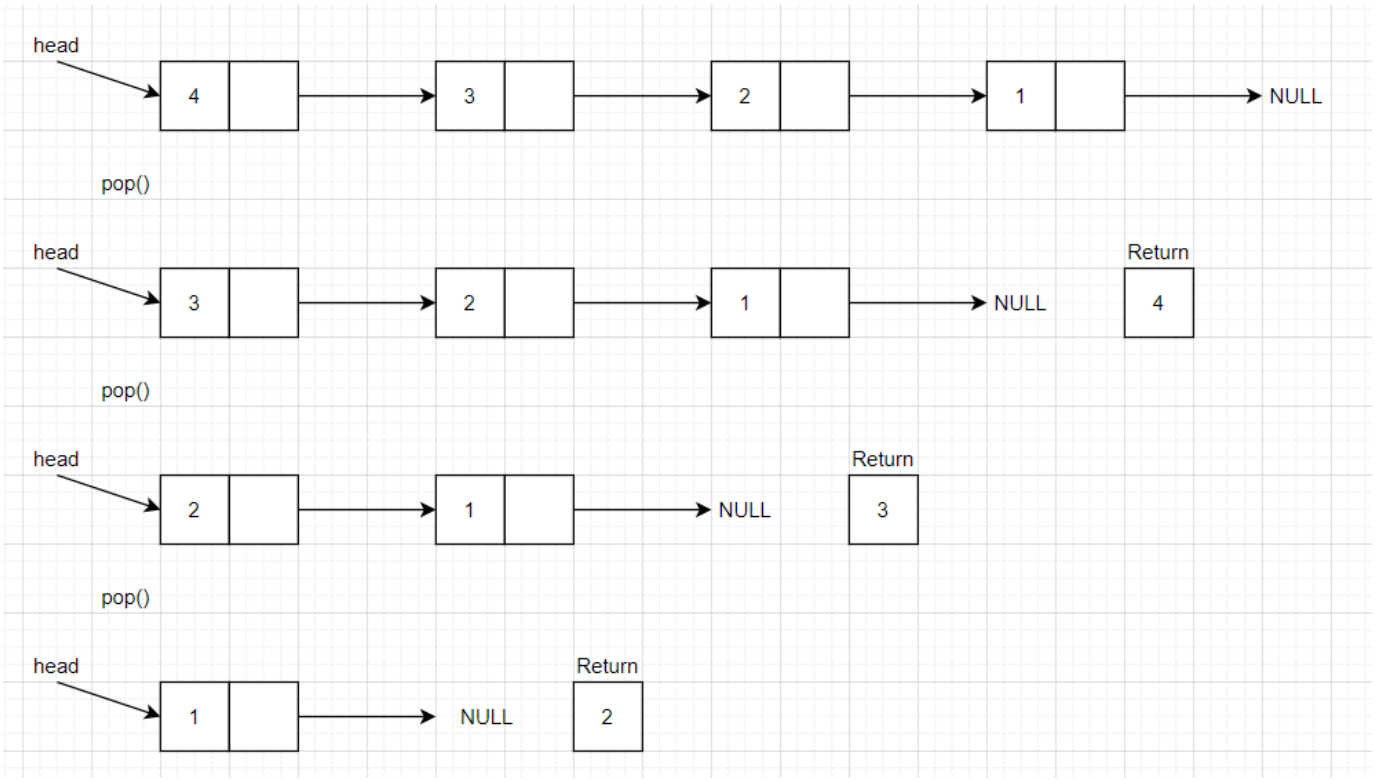

Figure 2: Stack pop

2. Queue

A queue is a linear structure following a fixed order in which the operations are carried out. First In First Out (FIFO) is the order here. In a stack, we delete the most recently added object; we delete the least recently added object in a queue.

Example:

- Queue enqueue:

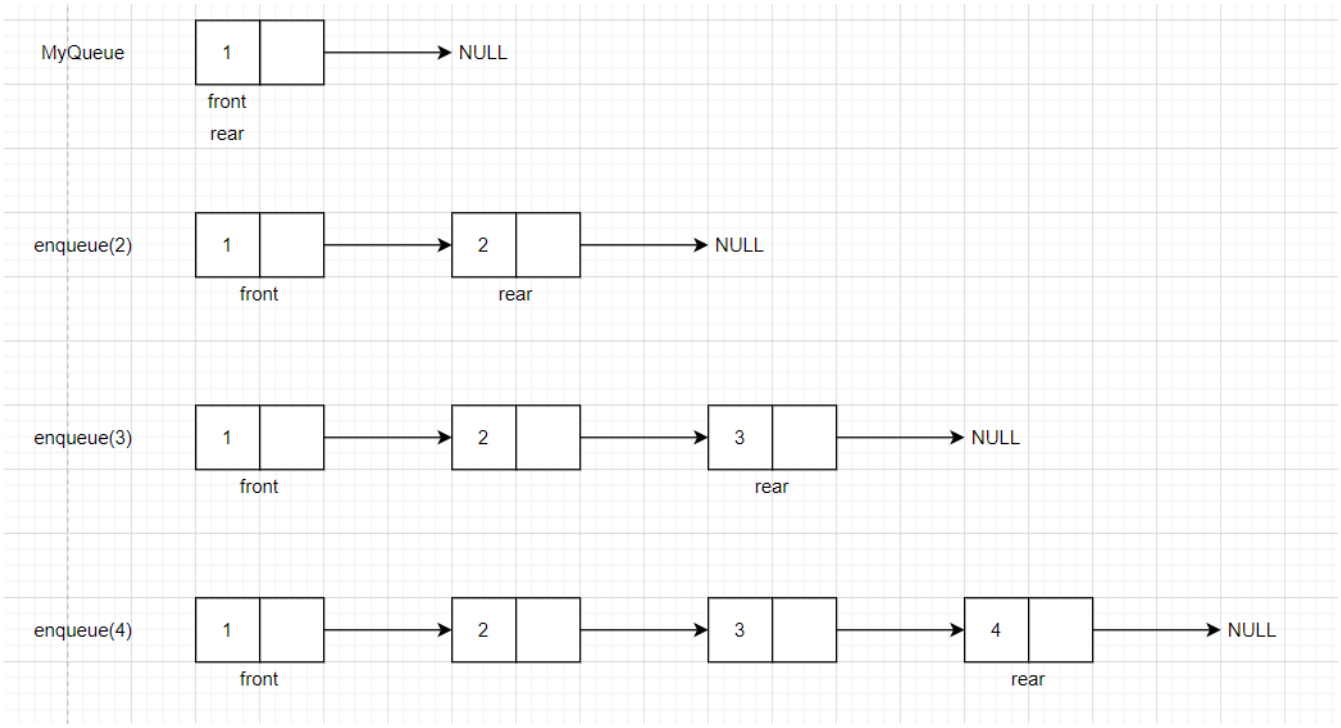

Figure 3: Queue enqueue

- Queue dequeue:

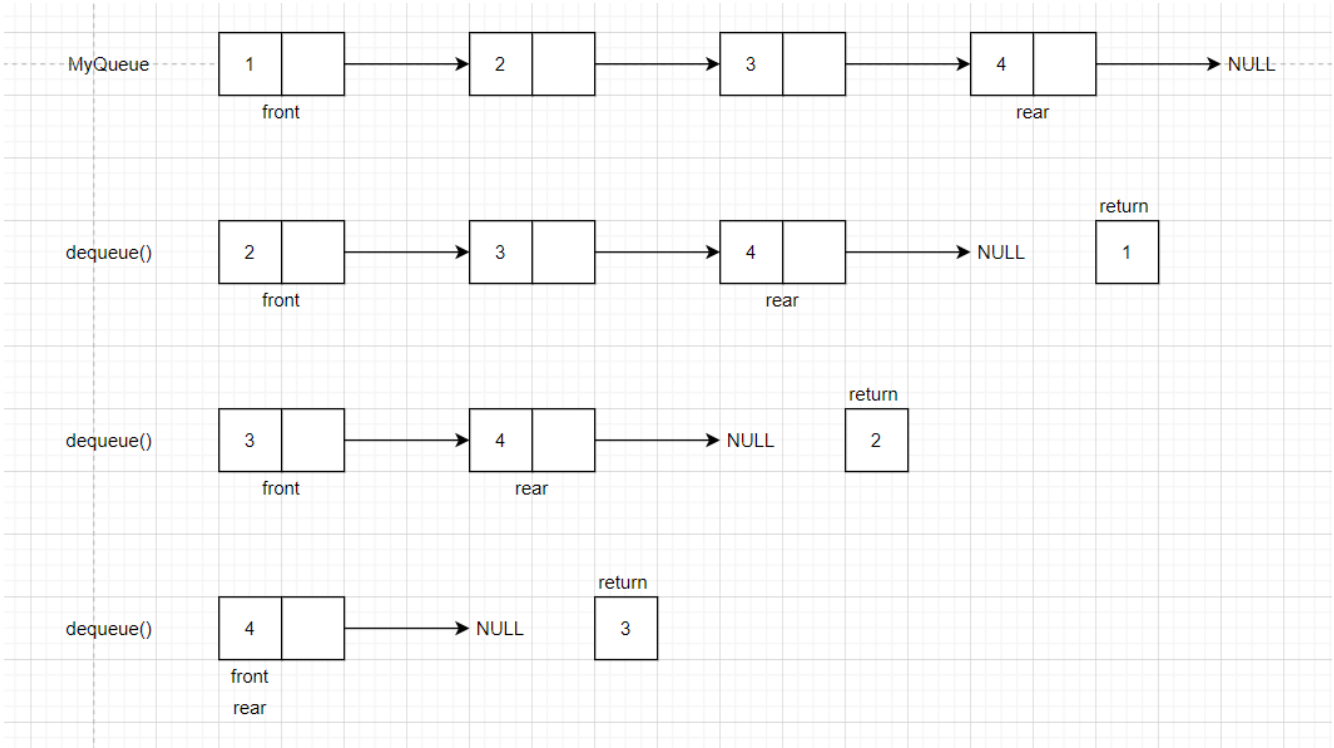

Figure 4: Queue dequeue

II. Application

- Class Node:

```
1  package asm2;
2
3  public class Node {
4      int data;
5      Node next;
6      public Node (int x) {
7      this.data = x;
8      this.next = null;
9      }
10 }
```

Figure 5: Class Node

- Creates a compile that stores data and a reference and assigns default values to them.
- Create a constructor Node () and pass the value.

- Class Stack:

```java
package asm2;

public class Stack {
    public Node top = null;
    public int size = 0;
    public int max;

    public Stack(int max) {
        this.max = max;
    }

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

    public boolean isFull() {
        return (size == max);
    }

    public void push(int data)
    {
        if (isFull()) {
            System.out.println("Stack is full");
            return;
        }
        else {
            Node node = new Node(data);
            if(top == null)
            {
                top = node;
                size++;
            }
            else {
                Node temp = top;
                top = node;
                top.next = temp;
                size++;
            }
        }
    }
    public int pop()
    {
        if (isEmpty()) {
            System.out.println("Cannot pop because stack is empty");
            return 0;
        }
        else {
            Node temp = top;
            top = top.next;
            int data = temp.data;
            size--;
            return data;
        }
    }
    public void comparingStack(Stack stack01) {

        Node current1 = this.top;
        Node current2 = stack01.top;
        while (current1 != null) {
            if (current1.data != current2.data) {
                System.out.println("stack 1 is different from stack 2");
                return ;
            }
        current1 = current1.next;
        current2 = current2.next;
        }
        System.out.println("stack 1 is the same as stack 2");
    }
    }

```

Figure 6: Class Stack

- Creates a constructor that passes a max value
- Create a push() function pass the data value then execute isFull (), if the stack is full, print "Stack is full". If the stack is not full, then create a new node, assign top by a temporary variable (temp), then top will equal the new node, and top.next = temporary variable (temp).
- Function pop() to remove a value from the stack. If the stack doesn't have any value, print "Cannot pop because the stack is empty". If the stack gets to the top by the temporary variable (temp), then top it with the next value in the stack and get the value out.
- 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".

- Class Queue:

```java
package asm2;

public class Queue {
        public Node head = null;
        public Node rear = null;
        public int size = 0;
        public int max;
        public Queue(int max) {
            this.max = max;
        }
        public boolean isEmpty() {
            return (head == null);
        }

        public boolean isFull() {
            return (size == max);
        }

        public void enQueue (int x) {
            if (isFull()) {
                System.out.println("Queue is full");
                return;
            }
            else {
                Node node = new Node(x);
                if(head == null) {
                    head = node;
                    rear = node;
                    size++;
                }
                else {
                    Node temp = rear;
                    rear = node;
                    temp.next = rear;
                    size++;
                }
            }
        }
        public int deQueue(){
            if (isEmpty())
            {
                System.out.println("Queue is empty");
                return 0;
            }
            else {
                if (rear == head) {
                    int data = head.data;
                    rear = null;
                    head = null;
                    size--;
                    return data;
                }
                else {
                    int data = head.data;
                    head = head.next;
                    size--;
                    return data;
                }
            }
        }
        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);
        }

        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: Class Queue

- Create the queue's start and end points head and rear.
- Creates a constructor function that passes the max value.

- Creates enQueue() function that passes a value, first checks to see if the queue is empty. Then create a new node. If there are no elements in the queue, then assign head and rear with that node. If there is an element, then assign rear by temporary variable (temp), then assign rear by new node, rear = temp.next and increase the size.
- In dequeue() function if the queue has only 1 element then head = rear = null. If the queue has more than 1 element, then remove the element at the head position, reduce the size and get the value out.
- The printAll() function to print out each value on the stack.
- In the Reverse() function creates a new stack that passes the max value, the size = size of the queue. For the first loop, the queue value will be dequeue and then pushed onto the stack. The second loop will pop all the values in the stack and then enqueue into the queue.

- Class Main:

```
1  package asm2;
2
3  public class Main {
4          public static void main(String[] args) {
5          //create new queue
6          Queue queue = new Queue(8);
7          queue.enQueue(1);
8          queue.enQueue(2);
9          queue.enQueue(3);
10         queue.enQueue(4);
11         queue.enQueue(5);
12         //creat new stack01
13         Stack stack01 = new Stack(8);
14         stack01.push(1);
15         stack01.push(2);
16         stack01.push(3);
17         stack01.push(4);
18         stack01.push(5);
19         //create new stack02
20         Stack stack02 = new Stack();
21         stack02.push(1);
22         stack02.push(2);
23         stack02.push(3);
24         stack02.push(4);
25         stack02.push(5);
26         //Reverse queue
27         queue.printAll();
28         queue.Reverse();
29         queue.printAll();
30         //Compare 2 stacks
31         stack01.comparingStack(stack02);
32         }
33 }
34
```

Figure 8: Class Main

- The Main class creates a new queue and the 2 new stacks also have an element limit of 8.
- Then pass the values 1, 2, 3, 4, 5 into the queue and stack.
- Then execute the Reverse () function and utilizeStack () function.

- Result:

Here are the results after running the program:

```
1 => 2 => 3 => 4 => 5
5 => 4 => 3 => 2 => 1
stack 1 is the same as stack 2
```

Figure 9: Result

III. Test case

No	Scope	Operation	Testing type	Input	Expected Output	Actual Output	Status
1	Stack ADT: Stack	push()	Normal	Stack: [1,2,3] push(4)	Stack: [1,2,3,4] Size of Stack = 4 Top() return 4	The same as expected output	passed
2		push()	Normal	Stack: [1,2,3,4] push(5)	Stack: [1,2,3,4,5] Size of Stack = 5 Top() return 5	The same as expected output	passed
3		pop()	Normal	Stack: [1,2,3] pop()	Stack: [1,2] Size of Stack = 2 Top() return 2	The same as expected output	passed
4		pop()	Normal	Stack: [1,2] pop()	Stack: [1] Size of Stack = 1 Top() return 1	The same as expected output	passed
5		pop()	Data validation	Stack: [] pop()	Stack: [] Print an error message	The same as expected output	passed

6	Queue ADT: Queue	enQueue()	Normal	Queue: [1,2,3] enQueue(4)	Queue: [1,2,3,4] Size of Queue = 4	The same as expected output	passed
7		enQueue()	Normal	Queue: [1,2,3,4] enQueue(5)	Queue: [1,2,3,4,5] Size of Queue = 5	The same as expected output	passed
8		deQueue()	Normal	Queue: [1,2,3] deQueue()	Queue: [2,3] Size of Queue = 2	The same as expected output	passed
9		deQueue()	Normal	Queue: [2,3] deQueue()	Queue: [3] Size of Queue = 1	The same as expected output	passed
10		deQueue()	Data validation	Queue: [] deQueue()	Queue: [] Print an error message	The same as expected output	passed
11	Question a	Reverse()	Normal	Queue: [1,2,3,4] Reverse ()	Queue :[4,3,2,1]	The same as expected output	passed
12	Question a	Reverse()	Data validation	Queue: [] Reverse ()	Queue :[]	The same as expected output	passed
13	Question b	comparing Stack()	Normal	Stack01:[1,2,3,4,5] Stack02:[1,2,3,4,5] comparingStack()	True	The same as expected output	passed
14	Question b	comparing Stack()	Normal	Stack01:[1,2,3,4,5] Stack02:[1,6,7,8,9] comparingStack()	False	The same as expected output	passed

- Evaluation:

In all, 14 cases were made and all were successful. When trying some validation data when the stack is empty, it cannot pop and when the stack is full, it cannot be pushed

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

Asymptotic analysis of an algorithm is about defining the mathematical boundary / framework of their run-time results. The best case, average case, and worst-case scenario of an algorithm can very well be inferred using asymptotic analysis.

Asymptotic analysis is bound by input, i.e. As such it would conclude operating for a constant period of time if the algorithm had no input. All other variables are considered unchanged, except for input.

Asymptotic analysis refers to the runtime calculation of all operations using arithmetic units. For example, one operation has a run time calculated in f (n) and can be computed for another with g (n2). This shows that with n increments the first activity increases linearly and every time n increases, the second activity exponentially increases time. Likewise, if n is small, tthe running time of both operations would be almost the same (Anon., 2020).

The execution time by an algorithm usually falls into the following three categories:

- Best case: Minimum time needed for implementation of the program.
- Average case: Average time needed for implementation of the program.
- Bad case: Maximum time needed for implementation of the program.

The widely used asymptotic notations to measure an algorithm's running time complexity are below:

- Big-O notation (O):

Big O notation is a mathematical notation to describe the limited behavior of a function when it tends to a certain or infinity value. Big O notation is a relative representation of an algorithm's complexity, often used to divide algorithms in order of execution time or memory space as input sizes increase (Anon., 2020).

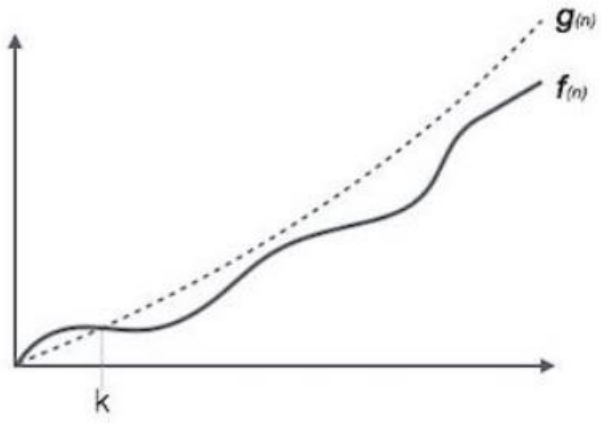

Figure 10: Big O notation (Anon., 2020)

- Omega Notation (Ω):

Omega notation represents the lower limit of algorithm run time. So it represents the best case of the algorithms. But the best-case performance of an algorithm usually doesn't make much sense, so Omega notation is the least used notation (Anon., 2020).

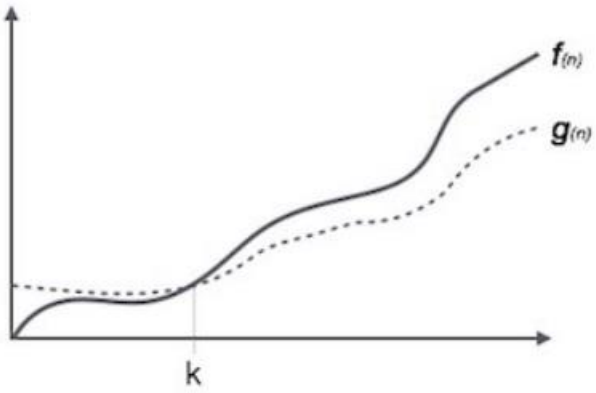

Figure 11: Omega notation (Anon., 2020)

- Theta Notation (θ):

Theta notation is a formal way to denote both the lower and upper bounds of an algorithm's run time. It is therefore often used to evaluate the average case complexity of an algorithm (Anon., 2020).

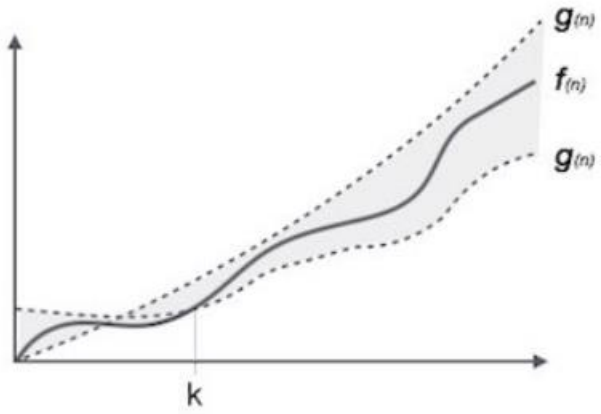

Figure 12: Theta notation (Anon., 2020)

V. Two ways in which the efficiency of an algorithm can be measured

Algorithm efficiency in computer science is an algorithm property related to the amount of resources used for the calculation of the algorithm. In order to evaluate and make judgments about its resource use, an algorithm must be evaluated carefully and an algorithm that can know its effectiveness through heavy usage. different resources. For a repetitive or continuous operation, algorithmic efficiency can be considered the same as technical productivity (Anon., 2020).

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:

Bubble sort and timsort are two algorithms used to sort lists in order from small to large. Bubble sort sorts the list by time proportional to the number of squared elements (O (n^2)) but only requires a small amount of memory (O (1)). Timsort sorts the list chronologically (proportional to the quantity multiplied by its logarithmic) in the list length (O (nlogn)) and requires a linear space by the length of the list (O (n)). If you want to sort quickly, timsort is a more suitable choice, but if you want to minimize memory capacity, bubble is the better choice.

Conclusion

My report has covered how stacks and queues work and how to implement a program that applies them.

Also give the concepts and examples of asymptotic notations.

References

Anon., 2020. *geeksforgeeks.* [Online]
Available at: https://www.geeksforgeeks.org/analysis-of-algorithms-set-3asymptotic-notations/?ref=lbp

Anon., 2020. *tutorialspoint.* [Online]
Available at: https://www.tutorialspoint.com/data_structures_algorithms/asymptotic_analysis.htm