


ASSIGNMENT 2 FRONT SHEET

Qualification	BTEC Level 5 HND Diploma in Computing		
Unit number and title	Unit 19: Data Structures and Algorithms		
Submission date	14/3/2021	Date Received 1st submission	
Re-submission Date		Date Received 2nd submission	
Student Name	Nguyen Gia Nam	Student ID	GCH190769
Class	GCH0805	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	

Grading grid

P1	P2	P3	M1	M2	M3	D1	D2

P4
✓

P5
✓

P6
✓

P7
✓

☐ Summative Feedback:

☐ Resubmission Feedback:

2.1

Grade:

Assessor Signature:

Date:

Internal Verifier's Comments:

IV Signature:

I. Design and implementation of Stack ADT and Queue ADT (P4).....	5
1. Stack	5
2. Queue.....	8
II. Application (P4)	12
1. Introduction	12
2. Implementation.....	12
III. Implement error handling and report test results (P5)	15
1. Testing Plan.....	15
2. Evaluation.....	16
IV. Discuss how asymptotic analysis can be used to assess the effectiveness of an algorithm (P6)	16
V. Determine two ways in which the efficiency of an algorithm can be measured, illustrating your answer with an example (P7)	17
VI. References	20

Figure 1. Stack operations (Source: Tutorialspoint, 2021).....	5
Figure 2. POP Operation.....	8
Figure 3. PUSH Operation.....	8
Figure 4. Enqueue Operation.....	11
Figure 5. Dequeue Operation.....	12
Figure 6. Requeue function.....	14
Figure 7. Reverse function.....	14
Figure 8. Big O Complexity	17
Figure 9. Bubble Sort (AlgorithmHub - Bubble Sort, 2021).....	19

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

1. Stack

A stack is a linear data structure in which operations are carried out in a specific order. The order may be LIFO (First In, First Out) or FILO (First In, Last Out) (First In Last Out).

Operations

The stack mainly performs the following three basic operations:

- **Push:** Inserts a new item into the stack. An overload condition occurs when the stack is full.
- **Pop:** Removes an item from the stack. Items are popped in the opposite order that they were pushed. It's considered an Underflow condition when the stack is empty.
- **Peek or Top:** Returns the stack's top element.
- **isEmpty:** If the stack is null, it returns true; otherwise, it returns false (geeksforgeeks.org, 2021).

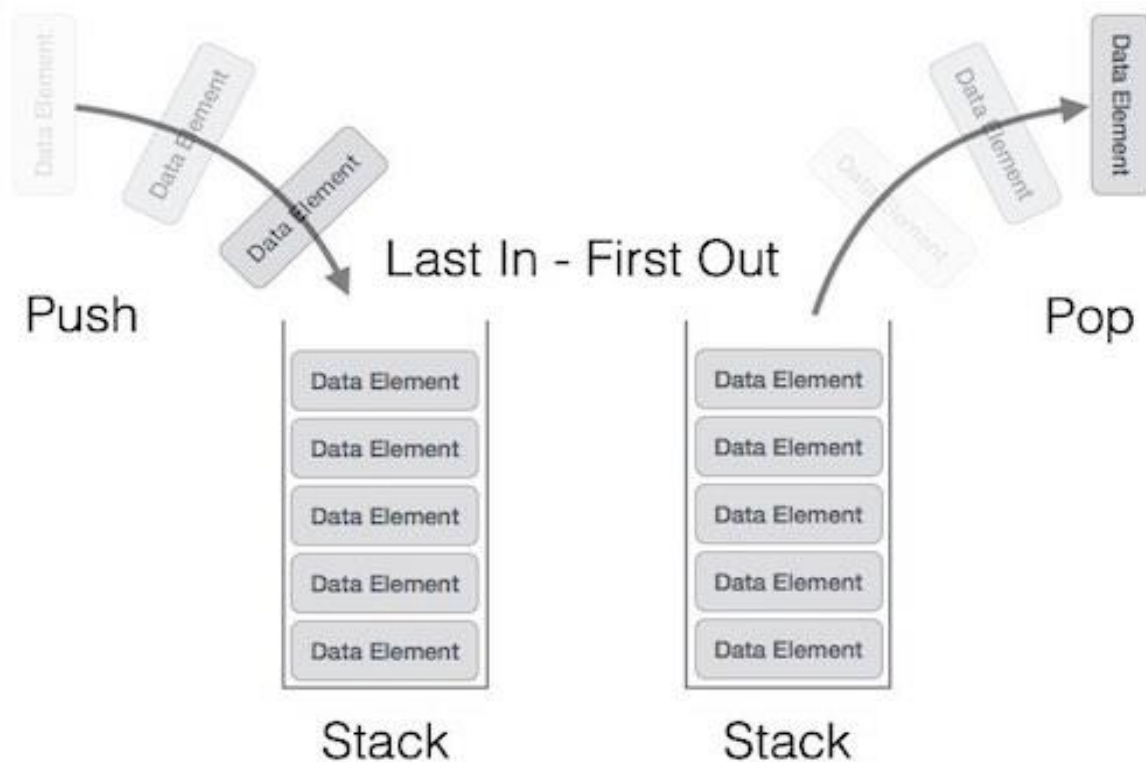


Figure 1. Stack operations (Source: Tutorialspoint, 2021)

Implementation

We have two ways implementation a stack:

- Array
- Linked List

Source code:

```
class Stack {
    static final int MAX = 1000;
    int top;
    int a[] = new int[MAX];

    boolean isEmpty()
    {
        return (top < 0);
    }
    Stack()
    {
        top = -1;
    }

    boolean push(int x)
    {
        if (top >= (MAX - 1)) {
            System.out.println("Stack Overflow");
            return false;
        }
        else {
            a[++top] = x;
            System.out.println(x + " pushed into stack");
            return true;
        }
    }

    int pop()
    {
        if (top < 0) {
            System.out.println("Stack Underflow");
            return 0;
        }
        else {
            int x = a[top--];
            return x;
        }
    }
}
```

```
int peek()
{
    if (top < 0) {
        System.out.println("Stack Underflow");
        return 0;
    }
    else {
        int x = a[top];
        return x;
    }
}

class Main {
    public static void main(String args[])
    {
        Stack s = new Stack();
        s.push(1);
        s.push(2);
        s.push(3);
        System.out.println(s.pop() + " Popped from stack");
    }
}
```

Code is referenced by (Stack Data Structure (Introduction and Program) - GeeksforGeeks, 2021)

 Result

```
1 pushed into stack
2 pushed into stack
3 pushed into stack
3 popped from stack
```

PUSH

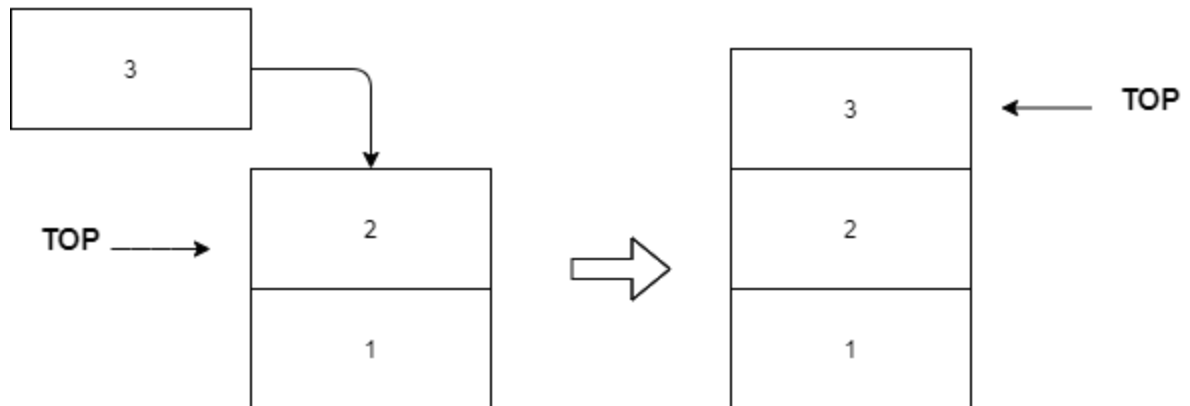


Figure 3. PUSH Operation

The Push function is added to a new item in the stack. The condition of the overflow is said to be if the stack is full. In the example code, 2 is the current top. After 3 is pushed to the stack, 3 is the new top of stack.

POP

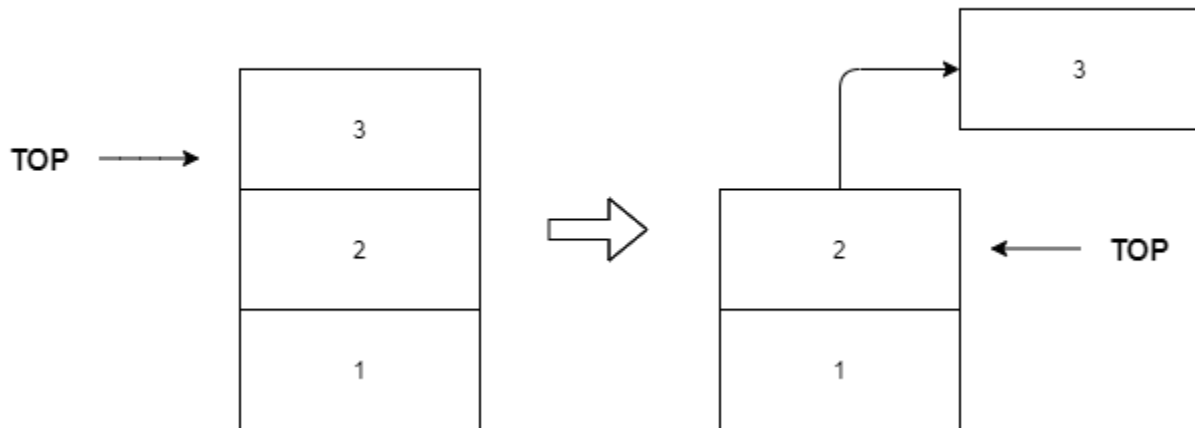


Figure 2. POP Operation

As opposed to Push, you can remove an item from the stack. The lower of the popped top element instances is the top element as it pops out from the stack. If the stack is empty, then an underflow state is said to be. In this example, 3 was the top of stack before pops out from stack. After that, 2 now is the top of stack.

2. Queue

Queue is a data structure that is similar to Stacks in that it is an abstract data structure. A queue, unlike stacks, is open on both ends. The one end is always used to insert data (enqueue), while the other is always used to delete data (dequeue) (dequeue). The First-In-

First-Out (FIFO) method is used in Queue, which means that the data item that was stored first will be accessed first.

Initializing or defining the queue, using it, and then completely erasing it from memory are examples of queue operations. There are basic operations associated with queues:

- **Enqueue:** add an item to the queue.
- **Dequeue:** remove an item from the queue.
- Few more functions are required to make the above-mentioned queue operation efficient.
- **Peek:** Gets the element at the front of the queue without removing it.
- **isfull:** Checks if the queue is full.
- **isempty:** Checks if the queue is empty.

In queue, we always dequeue data, pointed by front pointer and while enqueueing data in the queue we take help of rear pointer.

- **Front:** Get the front item from queue.
- **Rear:** Get the last item from queue. (Tutorialspoint, 2021)

 Source code

```
class Queue {
    int front, rear, size;
    int capacity;
    int array[];

    public Queue(int capacity)
    {
        this.capacity = capacity;
        front = this.size = 0;
        rear = capacity - 1;
        array = new int[this.capacity];
    }

    boolean isFull(Queue queue)
    {
        return (queue.size == queue.capacity);
    }

    boolean isEmpty(Queue queue)
    {
```

```
        return (queue.size == 0);
    }

    void enqueue(int item)
    {
        if (isFull(this))
            return;
        this.rear = (this.rear + 1) % this.capacity;
        this.array[this.rear] = item;
        this.size = this.size + 1;
        System.out.println(item + " enqueued to queue");
    }

    int dequeue()
    {
        if (isEmpty(this))
            return Integer.MIN_VALUE;

        int item = this.array[this.front];
        this.front = (this.front + 1) % this.capacity;
        this.size = this.size - 1;
        return item;
    }

    int front()
    {
        if (isEmpty(this))
            return Integer.MIN_VALUE;

        return this.array[this.front];
    }

    int rear()
    {
        if (isEmpty(this))
            return Integer.MIN_VALUE;

        return this.array[this.rear];
    }
}

public class Test {
    public static void main(String[] args)
    {
        Queue queue = new Queue(1000);
    }
}
```

```
queue.enqueue(1);  
queue.enqueue(2);  
queue.enqueue(3);
```

```
System.out.println(queue.dequeue() + " dequeued from queue\n");  
System.out.println("Front item is " + queue.front());  
System.out.println("Rear item is " + queue.rear());  
}  
}
```

Code is referenced by (Queue | Set 1 (Introduction and Array Implementation) - GeeksforGeeks, 2021)

Result

```
1 enqueued to queue  
2 enqueued to queue  
3 enqueued to queue  
1 dequeued from queue  
Front item is 2  
Rear item is 3
```

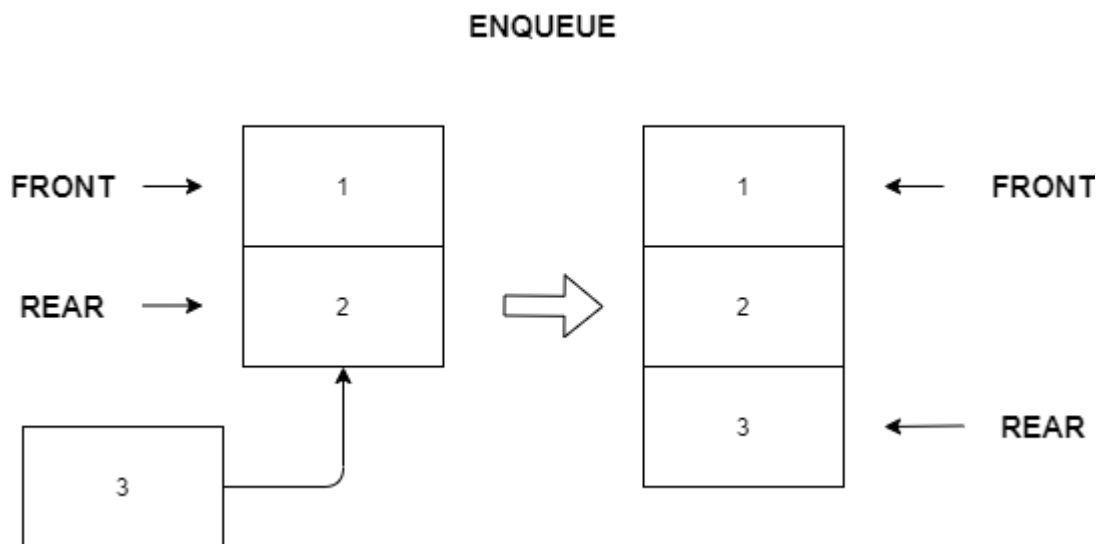


Figure 4. Enqueue Operation

Add an item from the rear of the queue with enqueue function. If the queue is full, the overflow statement will be show up. In the example, number 2 is the current rear of queue. Add number 1 to the queue, so now the current rear is number 1.

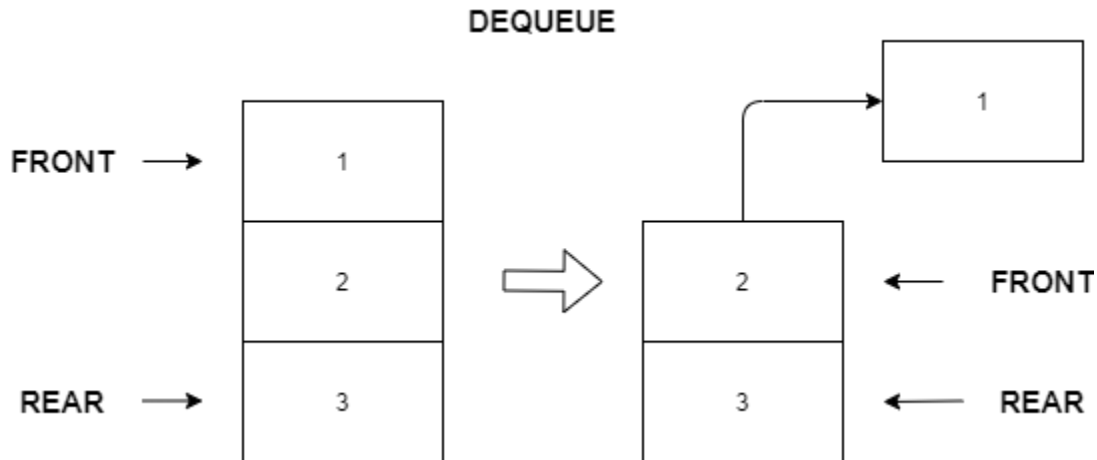


Figure 5. Dequeue Operation

Remove an item from the front of the queue is the function of dequeue. If the queue is empty, the underflow statement will be show up. In the example, number 3 was the front of queue before dequeues form the queue. Now, number 2 is the current front of queue.

II. Application (P4)

1. Introduction

To self-reverse the queue, we could store the queue elements in a temporary data structure such that they are reinserted in the same order as the queue elements are reinserted. We must now choose to use a data structure to view the data. According to the system, since the first element of the reverse queue will be the last element added, the temporary data structure should have the 'LIFO' property.

2. Implementation

Stack and queue support the reverse function

```
public class Requeue {
    public static void main(String[] args) {
        Queue queue = new Queue(1000);
        queue.enqueue(1);
        queue.enqueue(2);
    }
}
```

```
        queue.enqueue(3);  
        reverse(queue);  
    }  
  
    public static void reverseQueue(Queue q) {  
        Stack<Integer> s = new Stack<>();  
        while(!q.isEmpty()) {  
            s.push(q.dequeue());  
        }  
        while(!s.isEmpty()) {  
            q.enqueue(s.pop());  
        }  
    }  
}
```

Code is referenced by (queue, Rodriguez and Melnychuk, 2021)

Result

```
1 enqueue to queue  
2 enqueue to queue  
3 enqueue to queue  
3 enqueue to queue  
2 enqueue to queue  
1 enqueue to queue
```

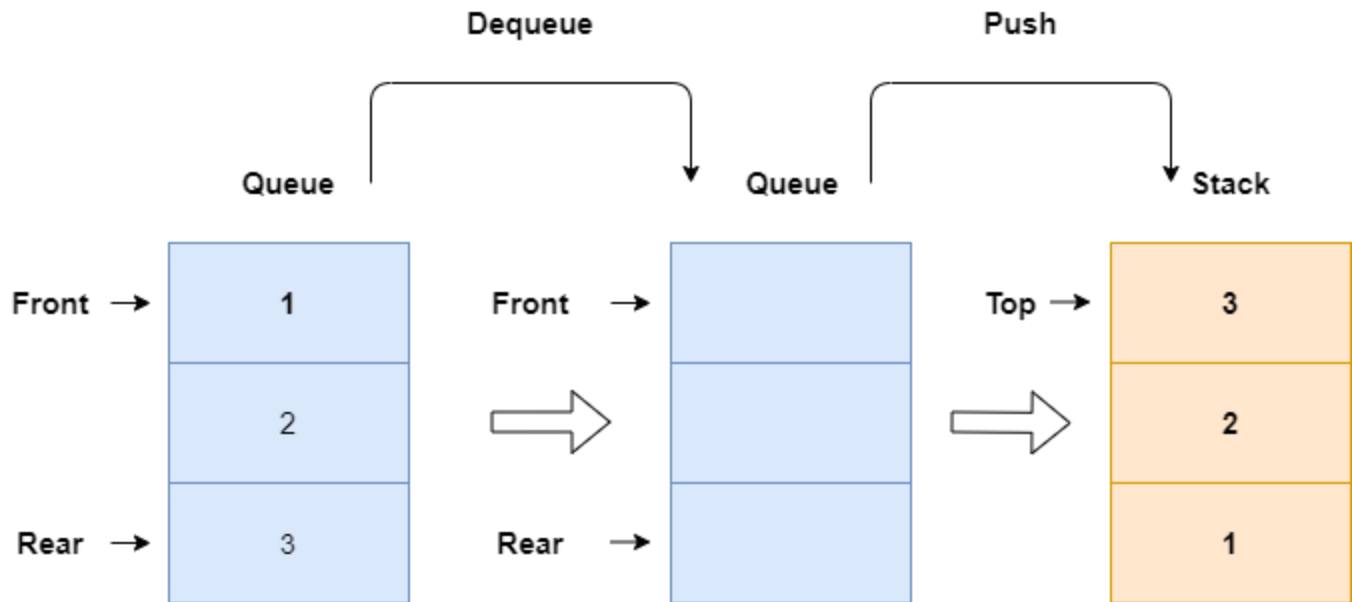


Figure 6. Requeue function

First, I dequeue all the element of queue and push them to stack. Before that, number 3 is the rear, number 1 is front of queue. Now, number 3 is the top of stack.

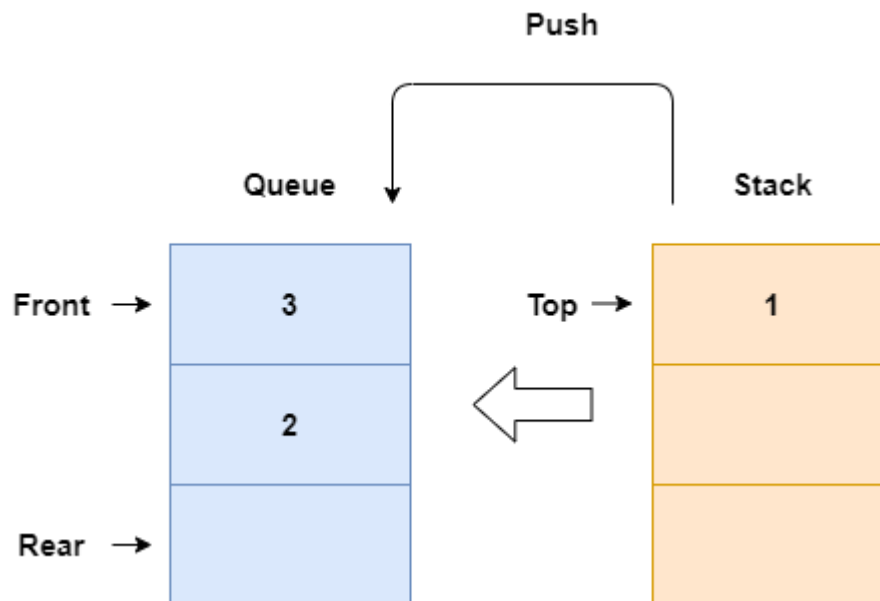


Figure 7. Reverse function

Second, I pop all the element of stack back to queue. The top of stack will become the front of queue and the last element become the rear of queue. This function called “Reverse”.

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

1. Testing Plan

No	Scope	Operation	Testing type	Input	Expected output	Actual output	Status
1	Stack ADT: Stack	pop()	Normal	Stack:[] pop()	Stack: [] Print an error message	Stack is null popped from stack	Passed
		pop()	Normal	Stack:[1] pop()	Stack: [] peek() returns 0	The same as expected output	Passed
		push()	Data validatio n	Stack:[2,1]push(3)	Stack: [3,2,1] peek() returns 3	The same as expected output	Passed
2	Queue ADT: Queue	enqueue()	Normal	Queue:[1,2] enqueue(3)	Queue:[1,2,3] q.rear.key = 3	The same as expected output	Passed
		dequeue()	Data validatio n	Queue:[]dequeue()	Queue:[] Print an error message()	Queue is null dequeue d from queue	Passed
		dequeue()	Normal	Queue:[1,2,3] dequeue()	Queue:[2,3] q.front.key = 2	The same as expected output	Passed
3	Queue reverse: Requeu e	reverseQue ue()	Normal	Queue:[1,2,3]rever se()	Queue:[3,2,1]	The same as expected output	Passed
					Queue:[1,2,3]	The same as	Passed

						expected output	
--	--	--	--	--	--	--------------------	--

2. Evaluation

I completed 8 basic case tests, that were successful. However, I just add small quantities here; if I add huge amounts, an error will occur. In the future, I'll provide more storage space to improve this.

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

Asymptotic analysis' main principle is to calculate the efficacy of algorithms that do not depend on machine-specific constants and do not necessitate algorithm execution or software comparison. Asymptotic notations are mathematical expressions for the time complexity of asymptotic analysis algorithms. The following 3 asymptotic notations are often used to describe the time complexity of algorithms.

The theta notation (Θ) defines exact asymptotic conduct by bounding a function from above and below. Dropping low order terms and avoiding leading constants is an easy way to get Theta notation for an expression. Take the following scenario: $3n^3 + 6n^2 + 6000 = \Theta(n^3)$

The Big O notation is being used to define an algorithm's upper bound; it only bounds a function from above. Consider the assistance of Insertion Sort, by example. In the best case, it takes linear time, and in the worst case, it takes quadratic time. We can confidently assume that the Insertion sort has an $O(n^2)$ time complexity. It's important to note that $O(n^2)$ also has linear time.

Just as Big O notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound. Ω Notation can be useful when we have lower bound on time complexity of an algorithm. However, the Omega notation is the least used notation among all three.

We need to use two statements for the best and worst cases if we use notation to represent time complexity of Insertion sort:

- The worst-case time complexity of Insertion Sort is $\Theta(n^2)$.
- The best-case time complexity of Insertion Sort is $\Theta(n)$.

When we only have an upper bound on an algorithm's time complexity, the Big O notation comes in useful. We can often find an upper bound simply by looking at the algorithm (geeksforgeeks.org, 2021).

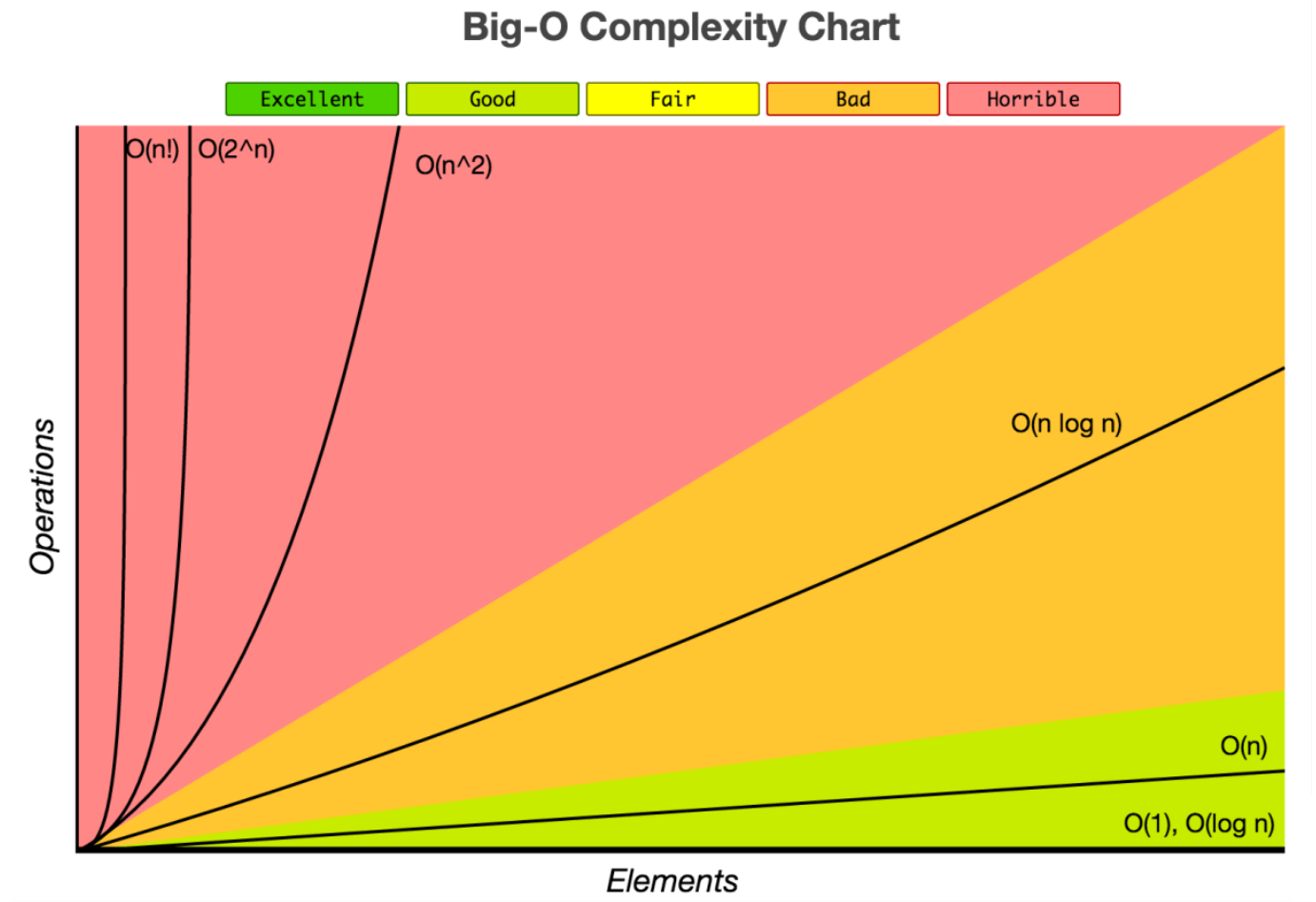


Figure 8. Big O Complexity

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

In computer science, algorithmic efficiency refers to how much computational resources an algorithm uses. An algorithm must be analyzed in order to evaluate its resource use, and an algorithm's performance can be calculated based on the utilization of different resources. Algorithmic efficiency can be compared to engineering productivity as applied to a repeating or continuous operation.

We need to use as few energies as possible to maximize efficiency. However, since different resources, such as time and space complexity, cannot be clearly compared, one of

two algorithms is perceived to be more efficient also depends on which efficiency measure is considered more relevant.

The two most common measures of an algorithm's resource use are speed and memory use; other measures may include transfer speed, temporary disk use, long-term disk usage, power use, overall cost of ownership, response time to external stimuli, and so on. Many of these metrics are based on the size of the algorithm's input, or the volume of data to be processed. They may also be affected by how the data is structured; for example, certain sorting algorithms deal with data that has already been sorted or is sorted in reverse order. (Algorithmic efficiency, 2021).

Time Complexity: time (computation time or response time) consumed in performing a given task.

Space Complexity: the data storage (RAM, HDD etc.) consumed in performing a given task.

Bubble Sort is a basic algorithm for sorting a set of n elements that is given in the form of an array of n elements. Bubble Sort compares each variable individually and sorts them according to their values.

If an array must be sorted in ascending order, bubble sort will begin by comparing the first and second elements of the array, swapping all elements if the first element is greater than the second, and then moving on to compare the second and third elements, and so on.

If an array must be sorted in ascending order, bubble sort will begin by comparing the first and second elements of the array, swapping all elements if the first element is greater than the second, and then moving on to compare the second and third elements, and so on.

If there are n elements in all, we must replicate the procedure $n-1$ times.

It's called bubble sort because the largest variable in the given sequence bubbles up to the last position or highest index for each full iteration, similar to how a water bubble rises to the water surface.

Sorting is achieved by sorting through all of the elements one by one, comparing them to the next closest element, and switching them if necessary.

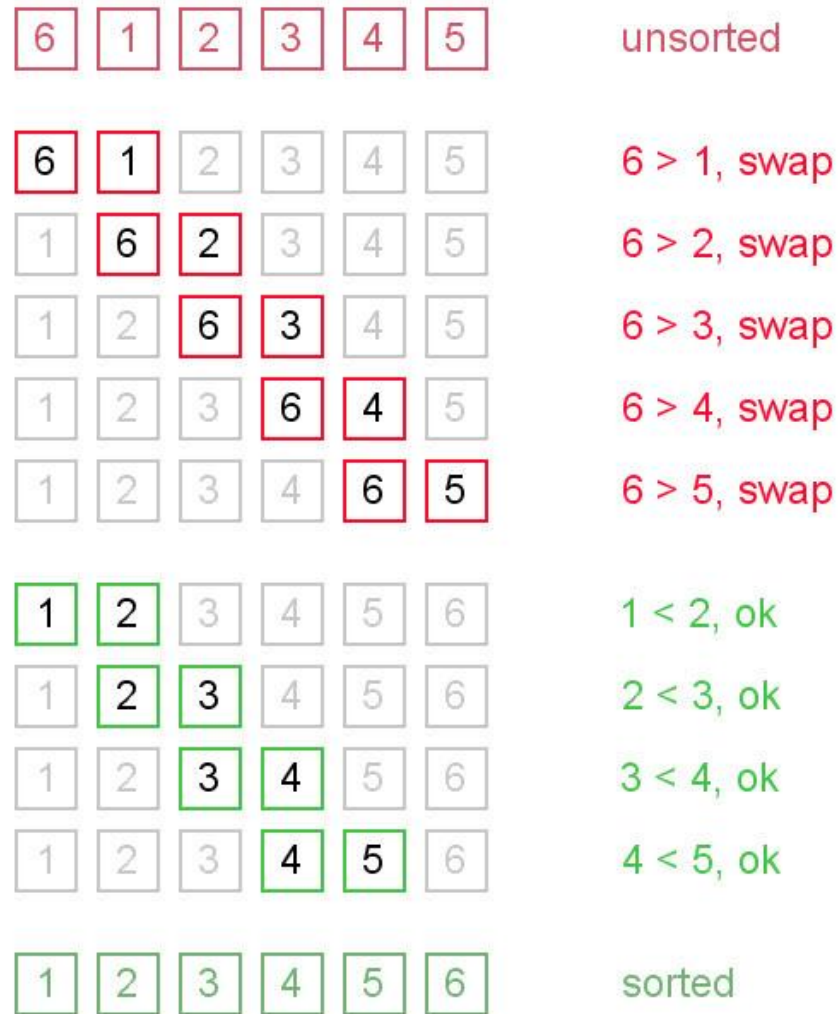


Figure 9. Bubble Sort (AlgorithmHub - Bubble Sort, 2021)

Time complexity of Bubble Sort is $O(n^2)$.

- The space complexity for Bubble Sort is $O(1)$, because only a single additional memory space is required i.e. for temp variable.
- Also, the best case time complexity will be $O(n)$, it is when the list is already sorted.
- Following are the Time and Space complexity for the Bubble Sort algorithm.
- Worst Case Time Complexity [Big-O]: $O(n^2)$
- Best Case Time Complexity [Big-omega]: $O(n)$
- Average Time Complexity [Big-theta]: $O(n^2)$
- Space Complexity: $O(1)$ (Bubble Sort Algorithm | Studytonight, 2021)

VI. References

Tutorialspoint.com. 2021. Data Structure And Algorithms - Queue - Tutorialspoint.

[online] Available at:

https://www.tutorialspoint.com/data_structures_algorithms/dsa_queue.htm [Accessed 10 March 2021].

GeeksforGeeks. 2021. Analysis Of Algorithms | Set 3 (Asymptotic Notations) -

Geeksforgeeks. [online] Available at: <https://www.geeksforgeeks.org/analysis-of-algorithms-set-3asymptotic-notations/> [Accessed 10 March 2021].

Studytonight.com. 2021. Bubble Sort Algorithm | Studytonight. [online] Available at:

<https://www.studytonight.com/data-structures/bubble-sort> [Accessed 11 March 2021].

AlgoHub.me. 2021. Algorithmhub - Bubble Sort. [online] Available at:

<http://www.algoHub.me/algo/bubble-sort.html> [Accessed 14 March 2021].

GeeksforGeeks. 2021. Queue | Set 1 (Introduction And Array Implementation) -

Geeksforgeeks. [online] Available at: <https://www.geeksforgeeks.org/queue-set-1introduction-and-array-implementation/> [Accessed 10 March 2021].

queue, R., Rodriguez, A. and Melnychuk, D., 2021. Reverse Method Reverses Elements Of A Queue. [online] Stack Overflow. Available at:

<https://stackoverflow.com/questions/16857276/reverse-methodreverses-elements-of-a-queue> [Accessed 10 March 2021].

GeeksforGeeks. 2021. Stack Data Structure (Introduction And Program) - Geeksforgeeks.

[online] Available at: <https://www.geeksforgeeks.org/stack-data-structure-introduction-program/> [Accessed 10 March 2021].

En.wikipedia.org. 2021. Algorithmic Efficiency. [online] Available at:

https://en.wikipedia.org/wiki/Algorithmic_efficiency#:~:text=There%20are%20many%20ways%20in,time%20to%20external%20stimuli%2C%20etc. [Accessed 11 March 2021].

Rodriguez, A. and Melnychuk, D., 2021. Reverse method reverses elements of a queue.

[online] Stack Overflow. Available at:

<https://stackoverflow.com/questions/16857276/reverse-method-reverses-elements-of-a-queue> [Accessed 13 March 2021].

Index of comments

2.1 The correct front-sheet for ASM 2 should be used.

- P4: Some figures have been used, however they are not good and detailed enough to illustrate how to implement the operations of Stack and Queue.

Q.a is solved, but it requires to use your own Stack instead of one from the Java library.

- P5: Some test cases are presented. The expected outputs for testing Queue should be revised. There, the way access the elements at position front and rear is not precise.

- P6 + P7: Discussion about different ways analyzing an algorithm is presented but shortly. It requires more examples in P6.