ASSIGNMENT 2 FRONT SHEET

Qualification	**BTEC Level 5 HND Diploma in Computing**		
Unit number and title	Unit 19: Data Structures and Algorithms		
Submission date	March 14, 2021	**Date Received 1st submission**	
Re-submission Date		**Date Received 2nd submission**	
Student Name	Nguyen Tien Hoc	**Student ID**	GCH190844
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	Hoc

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

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

1. Stack

a. Introduction

A stack is a data structure that uses the last in, first out (LIFO) concept. The item that was added last, that is, most recently, is in line to be the next one removed from the stack. Adding and removing objects from a stack are common of operations. Since a stack is known as a set of items of a particular base type, the base type is also included in the description. For each particular base type, users are given a different stack ADT (HWS, 2021).

b. Operations

A Stack contains elements of the same type arranged in sequential order. All operations take place at a single end that is top of the stack and following operations can be performed:

- push() – Insert an element at one end of the stack called top.
- pop() – Remove and return the element at the top of the stack, if it is not empty.
- peek() – Return the element at the top of the stack without removing it, if the stack is not empty.
- size() – Return the number of elements in the stack.
- isEmpty() – Return true if the stack is empty, otherwise return false

c. Implementation

Two ways to implement a Stack ADT are array and linked list.

Source code by geeksforgeeks.org (2021), use array:

```
public class Stack {

    static final int MAX = 1000;
    int top;
    int a[] = new int[MAX]; // Maximum size of Stack

    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("Empty Stack");
                return 0;
            }
            else {
                int x = a[top--];
                return x;
            }
        }

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


     class Main {
        public static void main(String args[])
        {
            Stack s = new Stack();
            s.push(10);
            s.push(20);
            s.push(30);
```

```
            System.out.println(s.pop() + " Popped from stack");
        }
}
```

Input:

Add 3 elements to Stack: 10, 20, 30.

Output:

"10 pushed into stack"
"20 pushed into stack"
"30 pushed into stack"
"30 Popped from stack"
Top element is: 20
Elements present in stack: 20 10

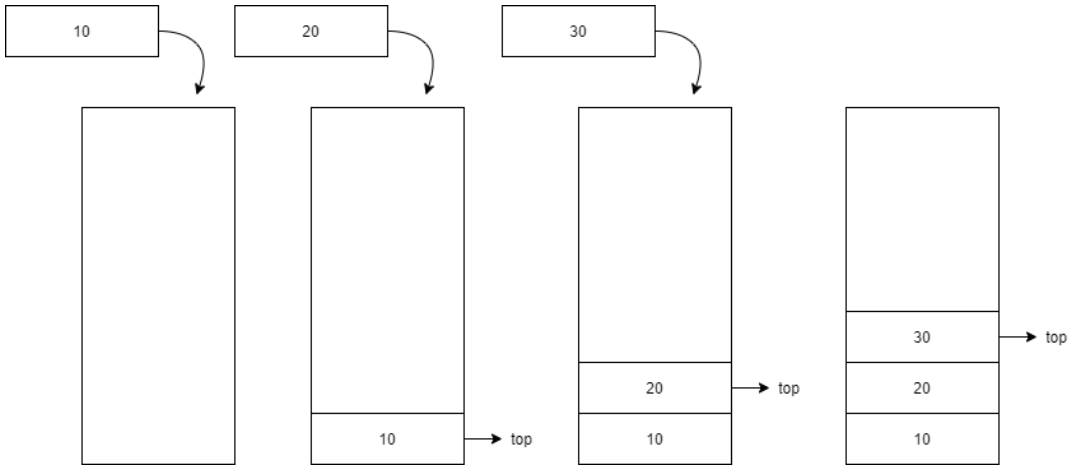


Figure 1: Add elements to Stack (push)

When adding element to Stack, the program first checks to see if the Stack is full. If full, the program announces and stops. If not yet, the program will increment Top by 1 unit and add a new element, then the new element will be in Top position and return success. This is push() method.

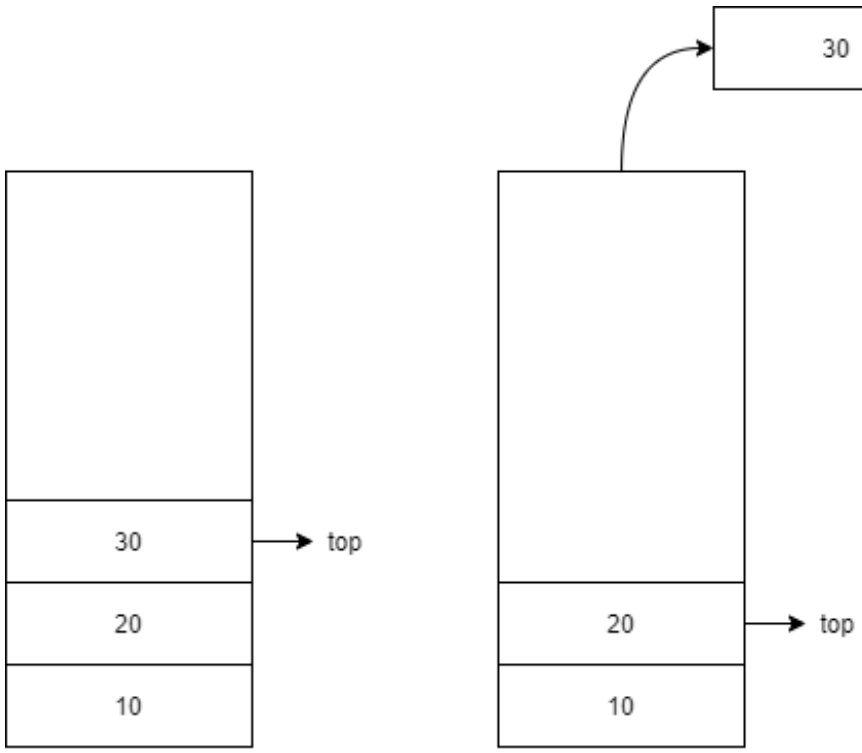


Figure 2: Remove the top element of Stack (pop)

The pop operation is to access the newest element and remove that element from the Stack. First, the program will check whether the Stack is empty or not. If it is empty, the program will announce and exit because there are no elements to delete. If Stack is not empty, access the element Top is pointing to and assign to x, then decrease Top by 1. This is pop() method.

2. Queue
   a. Introduction

      Queue is an abstract data structure. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first (Tutorialspoint, 2021).

   b. Operations

      A Queue contains elements of the same type arranged in sequential order. Operations take place at both ends, insertion is done at the end and deletion is done at the front. Following operations can be performed:

      - enqueue() – Insert an element at the end of the queue.
      - dequeue() – Remove and return the first element of the queue, if the queue is not
      - empty.
      - peek() – Return the element of the queue without removing it, if the queue is not
      - empty.

- size() – Return the number of elements in the queue.
- isEmpty() – Return true if the queue is empty, otherwise return false.

c. Implementation

Source code by geeksforgeeks.org (2021), use array:

```
public 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
                                          + " enqueue 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];
        }

        // Method to get rear of queue
        int rear()
        {
            if (isEmpty(this))
                return Integer.MIN_VALUE;

            return this.array[this.rear];
        }

        int peek() {
            return array[front];
        }
    }


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

            queue.enqueue(10);
            queue.enqueue(20);
            queue.enqueue(30);
            queue.enqueue(40);

            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());
        }
```

```
}
```

Input:
Add elements 10, 20, 30, 40 to Queue.
Remove front element (10).
Output:
"10 enqueued to queue"
"20 enqueued to queue"
"30 enqueued to queue"
"40 enqueued to queue"
"10 dequeued from queue"
"Front item is 20"
"Rear item is 40"

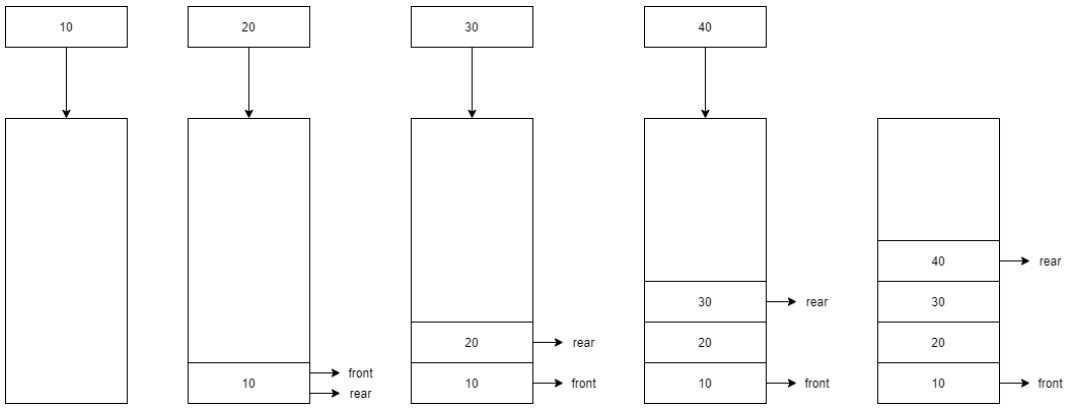


Figure 3: Add elements to Queue (enqueue)

The Queue data structure has two pointers, front (first element) and rear (last element). To add an element to the Queue, the program checks whether the queue is full. If full, it will announce and exit. If the Queue is not full, add that element to the Queue and increase the rear by 1 unit. At this point, the newest element will be in the rear pointer position. This is enqueue() method.

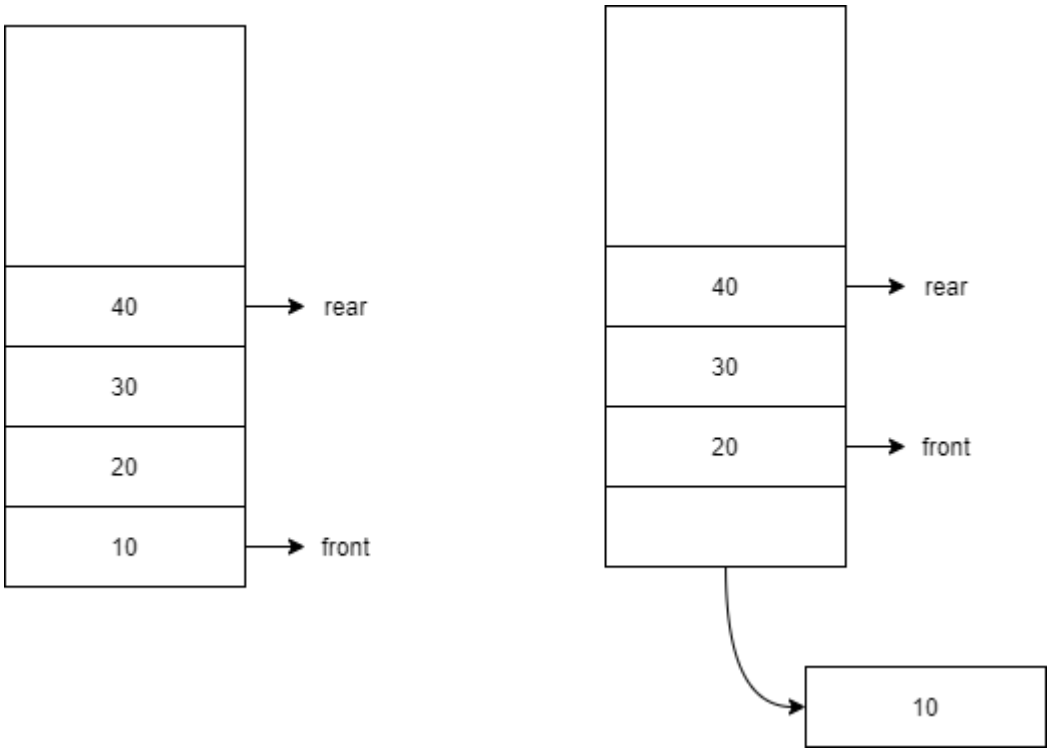


Figure 4: Remove element from Queue (dequeue)

To remove an element from the Queue, we need to use the front pointer because this is the first in, so it will first out. Check if the queue is empty or not, if empty there will be no elements to delete, the program announces and exits. If the queue is not empty, access the pointer front and assign the element at this position to x then increment front by 1 unit. At this point, front will be in the position that contains the next element. This is the dequeue() method.

II. Application (P4)

1. Introduction

   In reversing a queue data structure using the stack, the program needs an empty stack to store data temporarily. The elements will be removed from the Queue and added to the Stack, then returned to the Queue. The stack has the 'LIFO' property so the reverse process causes the first element of the Queue to come out of the last Stack to go back to the Queue.

2. Implementation

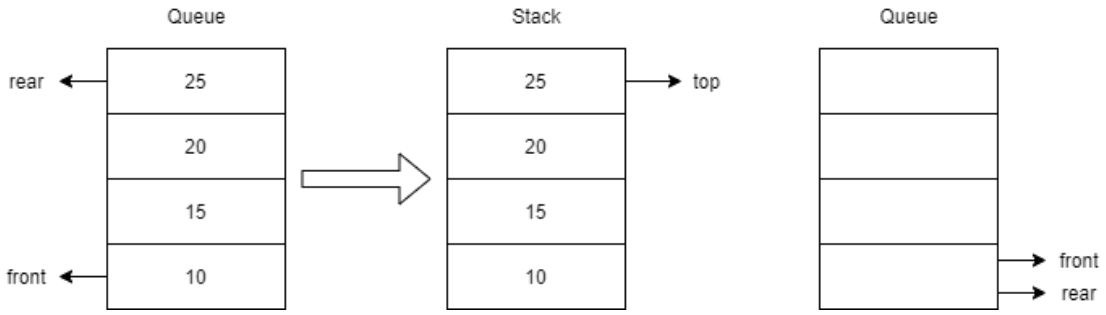
   Reverse() method uses the source code of the Queue and Stack above.

   Source code of Reverse():

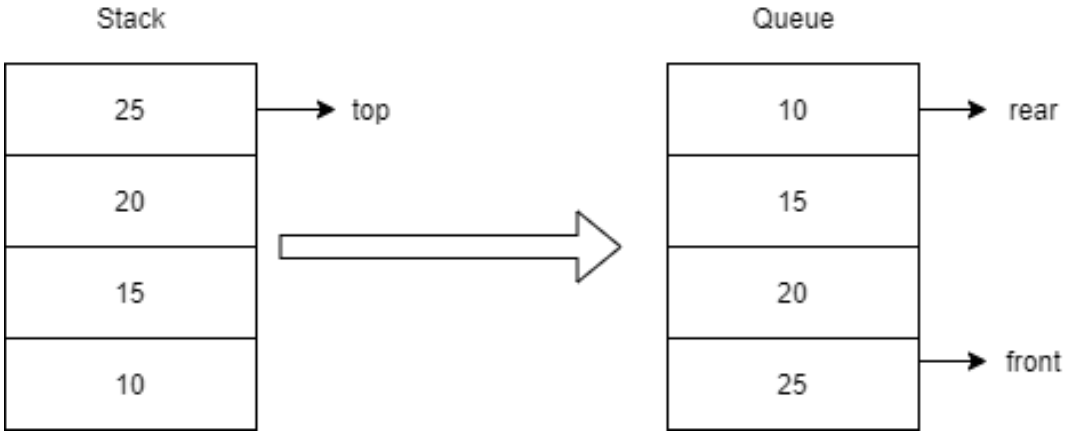
```
public class Reverse_Queue {

	public static void main(String[] args) {
		// TODO Auto-generated method stub
		Queue queue = new Queue(1000);

	    queue.enqueue(10);
	    queue.enqueue(15);
	    queue.enqueue(20);
	    queue.enqueue(25);

	    Reverse(queue);
	}

	public static void Reverse(Queue queue) {
		Stack stack = new Stack();
		while(!queue.isEmpty(queue)) {
			stack.push(queue.dequeue());
		}
		while(!stack.isEmpty()) {
			queue.enqueue(stack.pop());
		}
	}
}
```

Output:

10 enqueue to queue
15 enqueue to queue
20 enqueue to queue
25 enqueue to queue
10 pushed into stack
15 pushed into stack
20 pushed into stack
25 pushed into stack
25 enqueue to queue
20 enqueue to queue
15 enqueue to queue
10 enqueue to queue



The program transfers data from Queue to Stack. The front element is added to the Stack then dequeue from the Queue, continuing until the element ends. Thus, front will come out of the last Stack, and the original rear will be the top of the Stack



Finally, reverse data transfer from Stack to Queue. The top of the stack (initially rear) will first become the front of the Queue.

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

1. Testing plan

No	Scope	Operation	Testing type	Input	Expected output	Actually output	Status
1	Stack ADT: Stack	Pop()	Data validation	Stack:[]; pop()	Stack:[] Print "Empty Stack"; return 0	The same as expected output	Passed
2		Push()	Normal	Stack:[5,10,15]; push(20)	Stack:[20,5,10,15]; Size of Stack = 4; Top returns 20	The same as expected output	Passed

3		Peek()	Normal	Stack:[5,10,15]; Peek()	Stack:[5,10,15]; Return 5; Print "top = 5"	The same as expected output	Passed
4		Pop()	Normal	Stack:[5,10,15]; Pop()	Stack:[10,15]; Return 5; Print "5 popped from stack"	The same as expected output	Passed
5		Peek()	Data validation	Stack:[]; peek()	Stack:[]; return 0; Print "Empty stack"	The same as expected output	Passed
6	Queue ADT: Queue	Dequeue()	Data validation	Queue:[]; dequeue()	Queue:[]; Return -2147483648	The same as expected output	Passed
7		Enqueue()	Normal	Queue:[5,10,15]; enqueue(20)	Queue:[5,10,15,20]; Size of Queue = 4; Queue.rear() = 20;	The same as expected output	Passed
8		Dequeue()	Normal	Queue:[5,10,15]; dequeuer()	Queue:[10,15]; Size of queue = 2; Queue.front() = 10	The same as expected output	Passed
9		Front()	Normal	Queue:[5,10,15]; front()	Queue:[5,10,15]; Return 5	The same as expected output	Passed
10		Rear()	Normal	Queue:[5,10,15]; rear();	Queue:[5,10,15]; return 15	The same as expected output	Passed
11	Queue ADT: Reverse_ Queue	Reverse()	Normal	Queue:[5,10,15, 20]; reverse()	Queue:[20,15,10,5]	The same as expected output	Passed

2. Evaluation

   I have 11 test cases, none of which failed. These cases are not complicated and the numbers are small, so I will check back in the future for changes to improve the program.

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

The main goal of asymptotic analysis is to develop a measure of algorithm efficiency that is independent of machine-specific constants and does not necessitate the implementation of

algorithms or the comparison of program execution times. For asymptotic analysis, asymptotic notations are mathematical instruments for representing the time complexity of algorithms. The time complexity of algorithms is generally expressed using the following three asymptotic notations.

Θ Notation: The theta notation bounds a function from above and below, so it defines exact asymptotic behavior. A simple way to get Theta notation of an expression is to drop low order terms and ignore leading constants.
3n3 + 6n2 + 6000 = Θ(n3)
Dropping lower order terms is always fine because there will always be a number(n) after which Θ(n3) has higher values than Θ(n2) irrespective of the constants involved.
For a given function g(n), we denote Θ(g(n)) is following set of functions.

```
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}
```

If f(n) is theta of g(n), then the value f(n) is always between c1*g(n) and c2*g(n) for large values of n (n >= n0). The definition of theta also requires that f(n) must be non-negative for values of n greater than n0 (GeeksforGeeks, 2021).

Big O Notation: The Big O notation defines an upper bound of an algorithm, it bounds a function only from above. It takes linear time in best case and quadratic time in worst case.
If use Θ notation to represent time complexity of Insertion sort, have to use two statements for best and worst cases (GeeksforGeeks, 2021):

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

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

Ω Notation: can be useful when have lower bound on time complexity of an algorithm. The best case performance of an algorithm is generally not useful; the Omega notation is the least used notation among all three.

For a given function g(n), we denote by Ω(g(n)) the set of functions.

```
Ω (g(n)) = {f(n): there exist positive constants c and
```

```
n0 such that 0 <= c*g(n) <= f(n) for
all n >= n0}.
```

The time complexity of Insertion Sort can be written as Ω(n), but it is not a very useful information about insertion sort, as user are generally interested in worst case and sometimes in average case (GeeksforGeeks, 2021).

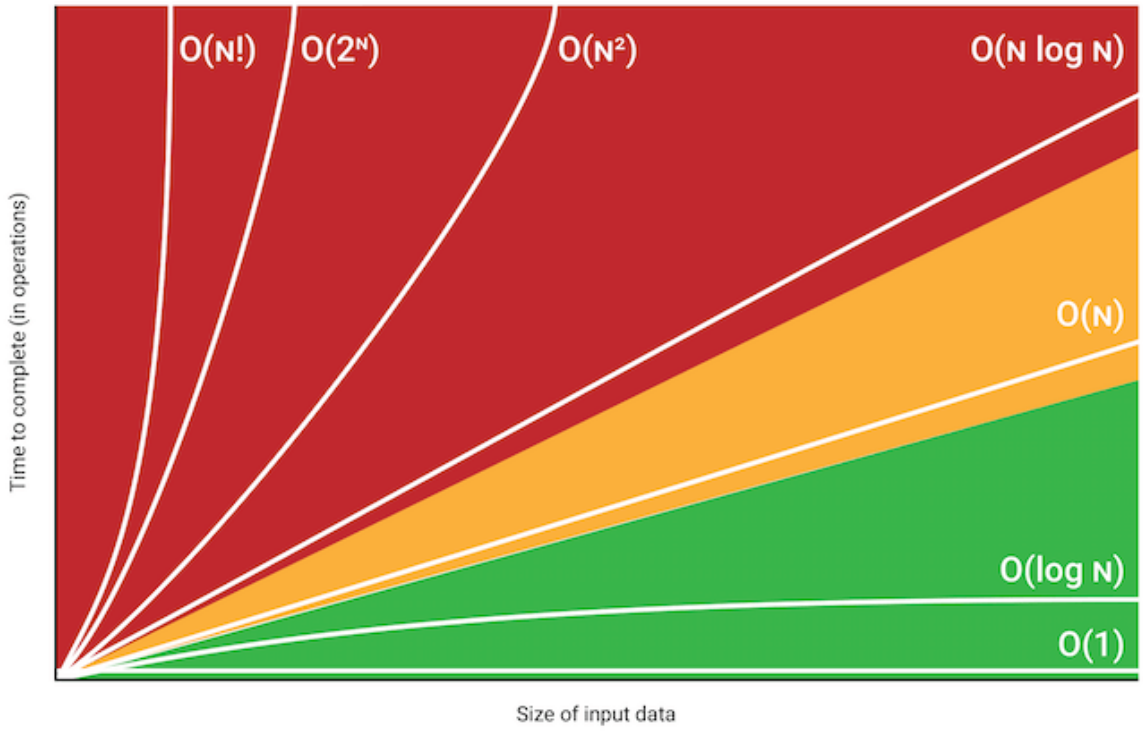


Figure 5: (Big-O Notation Explained | Daniel Miessler, 2021)

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

The time complexity in computer science is the computational complexity that explains how long it takes a computer to run an algorithm. The number of elementary operations performed by the algorithm is widely used to estimate time complexity, assuming that each elementary operation takes a fixed amount of time to complete. As a result, the algorithm's time and the number of elementary operations it performs are expected to vary by at most a constant factor (Wikipedia, 2021).

Space Complexity of an algorithm is total space taken by the algorithm with respect to the input size. Space complexity includes both Auxiliary space and space used by input.

Space complexity is a parallel concept to time complexity. If we need to create an array of size n, this will require O(n) space. If we create a two dimensional array of size n*n, this will require O(n2) space (geeksforgeeks, 2021).

Selection sort example, select smallest element returns to the first position of current array and doesn't care about it anymore. Then the array has only n - 1 element of the original array, continuing to consider from the 2 element of the array. And repeat until the current sequence has only 1 element left.

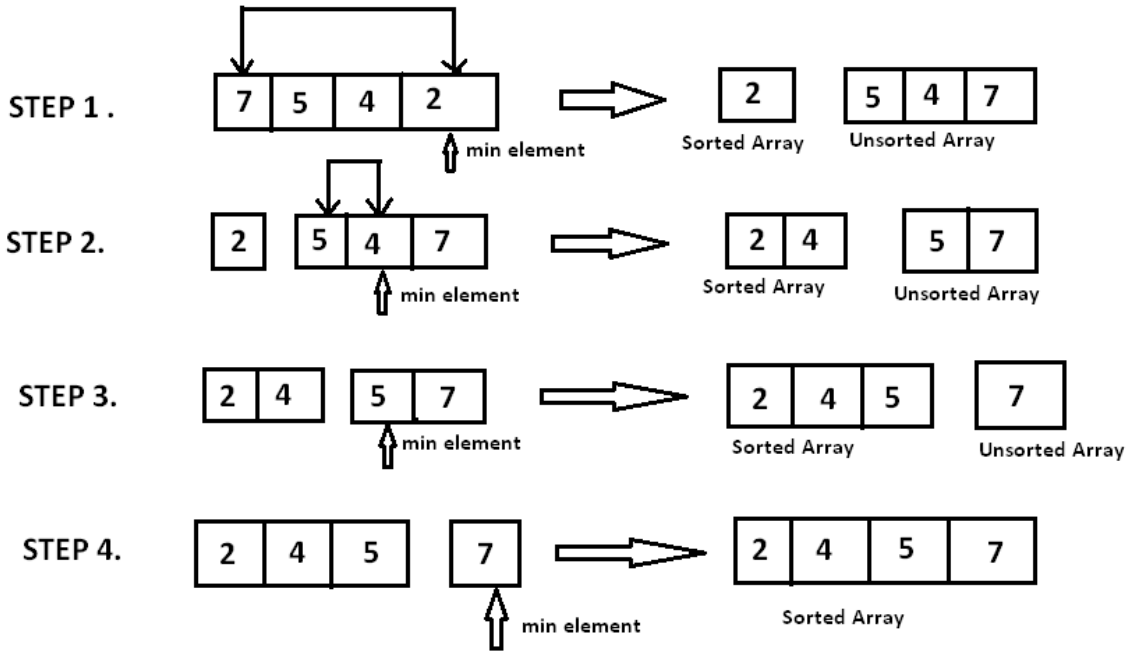


Figure 6: Selection sort algorithm

- Time complexity: O(n2)
- Space complexity: 0(1)
- Best case time complexity: Ω(N)
- Average case time complexity: Θ(N2)
- Worst case time complexity: O(N2)

VI. References

HWS Math and CS, 2021, Abstract data types, HWS Math and CS, viewed March 3, 2021, <http://math.hws.edu/eck/cs327_s04/chapter2.pdf>.

GeeksforGeeks, 2021, Stack Data Structure (Introduction and Program), GeeksforGeeks, viewed March 11, 2021, <https://www.geeksforgeeks.org/stack-data-structure-introduction-program/>

Tutorialspoint, 2021, Data Structure and Algorithms – Queue, Tutorialspoint, viewed March 3, 2021, <https://www.tutorialspoint.com/data_structures_algorithms/dsa_queue.htm>.

Daniel Miessler. 2021. Big-O Notation Explained | Daniel Miessler. [online] Available at: <https://danielmiessler.com/study/big-o-notation/> [Accessed 14 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 14 March 2021].

En.wikipedia.org. 2021. Space complexity. [online] Available at: <https://en.wikipedia.org/wiki/Space_complexity> [Accessed 14 March 2021].

GeeksforGeeks. 2021. What does 'Space Complexity' mean? - GeeksforGeeks. [online] Available at: <https://www.geeksforgeeks.org/g-fact-86/> [Accessed 14 March 2021].