ASSIGNMENT 1 FRONT SHEET

Qualification	**BTEC Level 5 HND Diploma in Computing**		
Unit number and title	Unit 19: Data Structures and Algorithms		
Submission date	March 6, 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

P1	P2	P3	M1	M2	M3	D1	D2

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

Contents

A. Introduction

The report is about the design and implementation of abstract data types. Includes several sections such as how to use ADT to improve software design, development, and testing.

B. P1 Create a design specification for data structures explaining the valid operations that can be carried out on the structures.

I. ADT

Abstract data type (ADT) is a mathematical model for data types. An abstract data type is defined by its behavior (semantics) from the point of view of a user, of the data, specifically in terms of possible values, possible operations on data of this type, and the behavior of these operations. This mathematical model contrasts with data structures, which are concrete representations of data, and are the point of view of an implementer, not a user (Wikipedia, 2021).

An abstract data type is defined as a mathematical model of the data objects that make up a data type as well as the functions that operate on these objects. There are no standard conventions for defining them. A broad division may be drawn between "imperative" and "functional" definition styles (Wikipedia, 2021).

An abstract data type (ADT) consists of the following:

- A collection of data
- A set of operations on the data or subsets of the data
- A set of axioms, or rules of behavior governing the interaction of operations

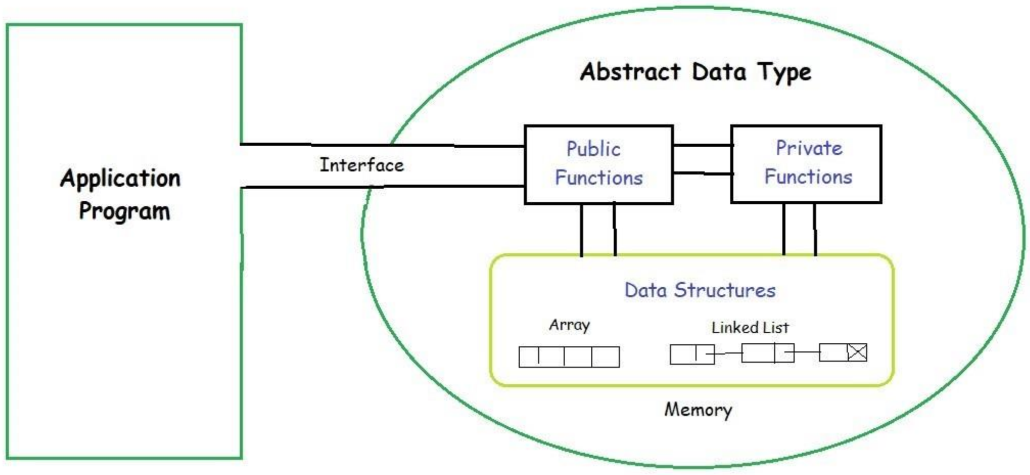


Figure 1: Abstract Data Types

II. Stack ADT

1. Definition

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).

Additionally, a peek operation may give access to the top without modifying the stack. The name "stack" for this type of structure comes from the analogy to a set of physical items stacked on top of each other. This structure makes it easy to take an item off the top of the stack, while getting to an item deeper in the stack may require taking off multiple other items first (Wikipedia, 2021).

2. Operations and working mechanism

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.

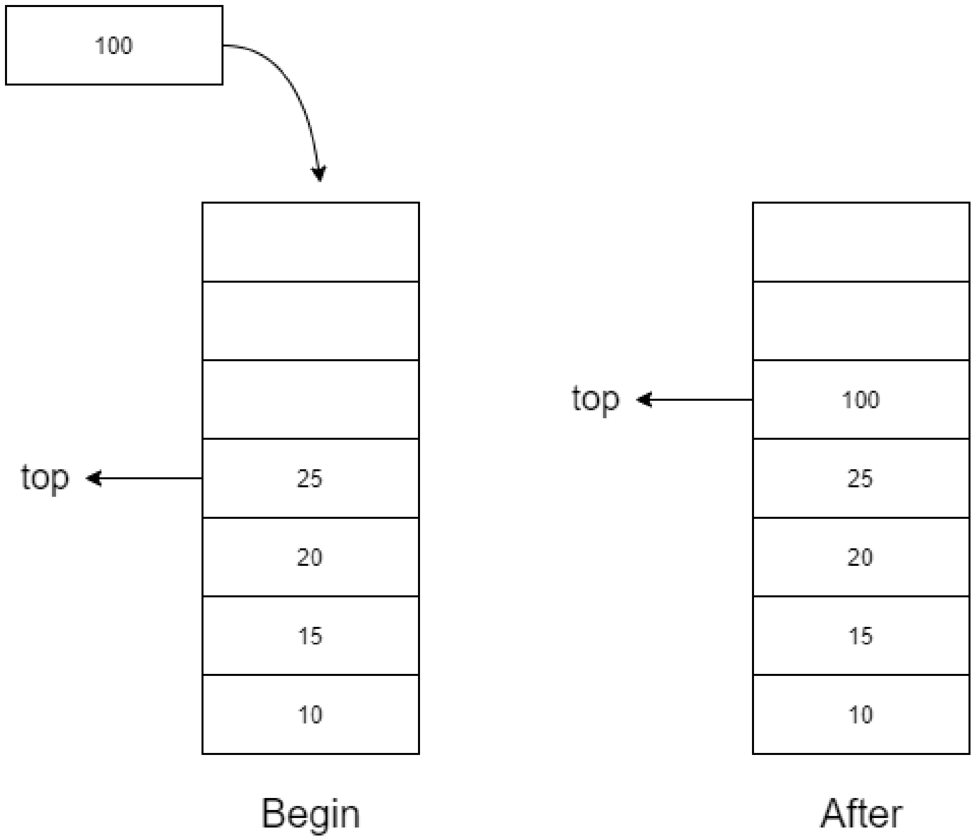


Figure 2: Push function in Stack ADT

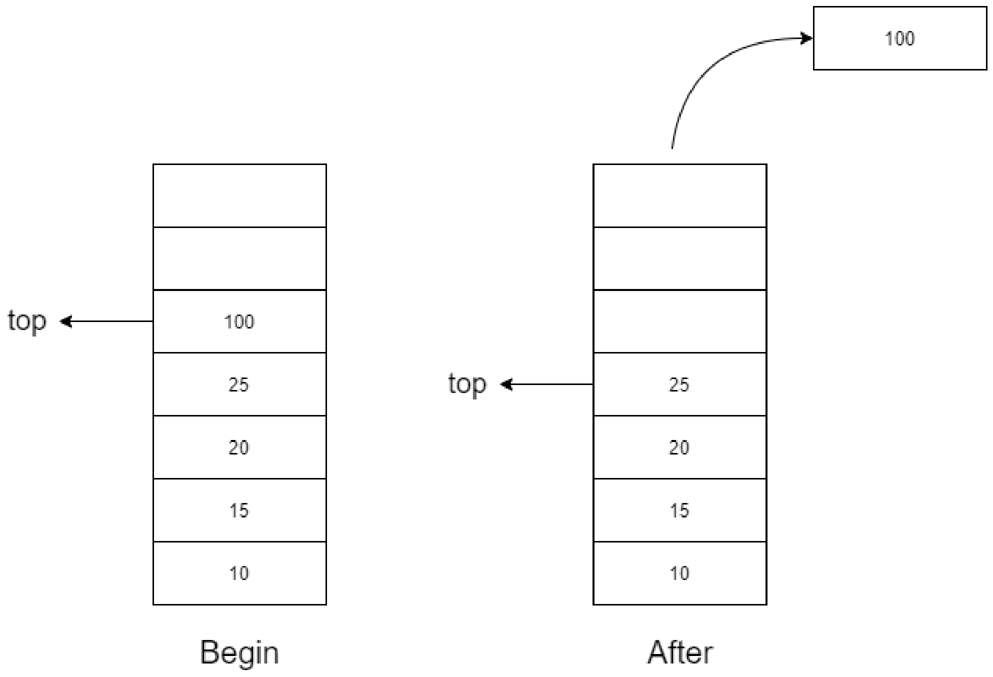


Figure 3: Pop function in Stack ADT

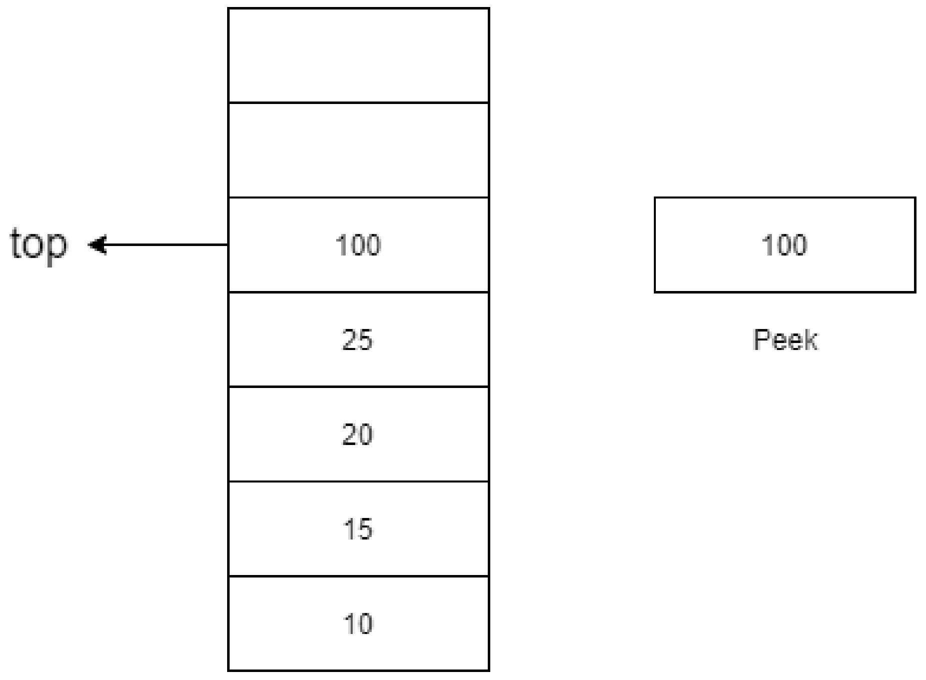


Figure 4: Peek function in Stack ADT

III. Queue ADT

1. Definition

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).

2. Operations and working mechanism

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.

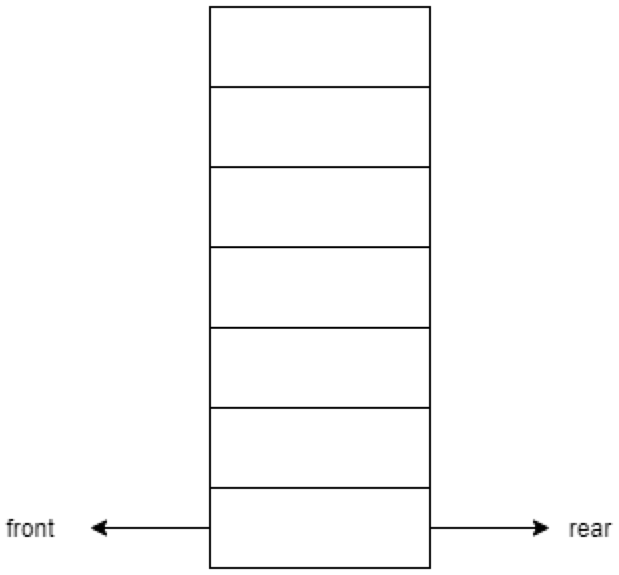


Figure 5: Empty Queue

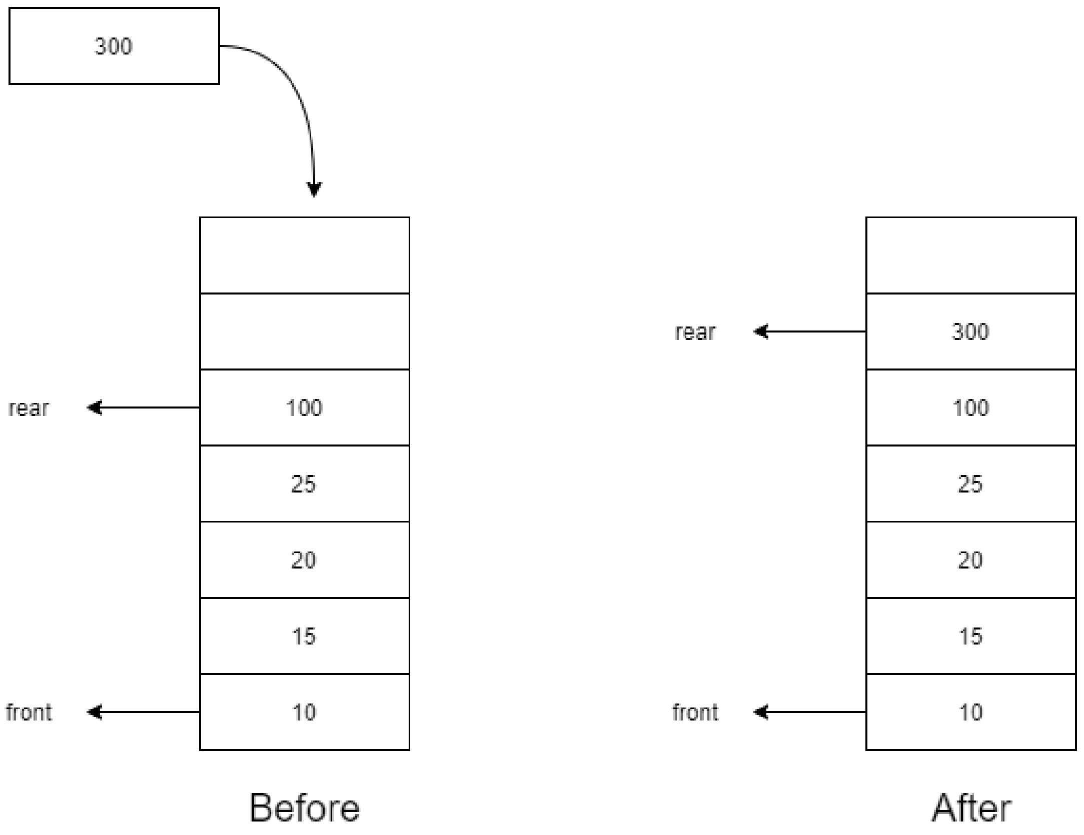


Figure 6: Enqueue function in Queue ADT

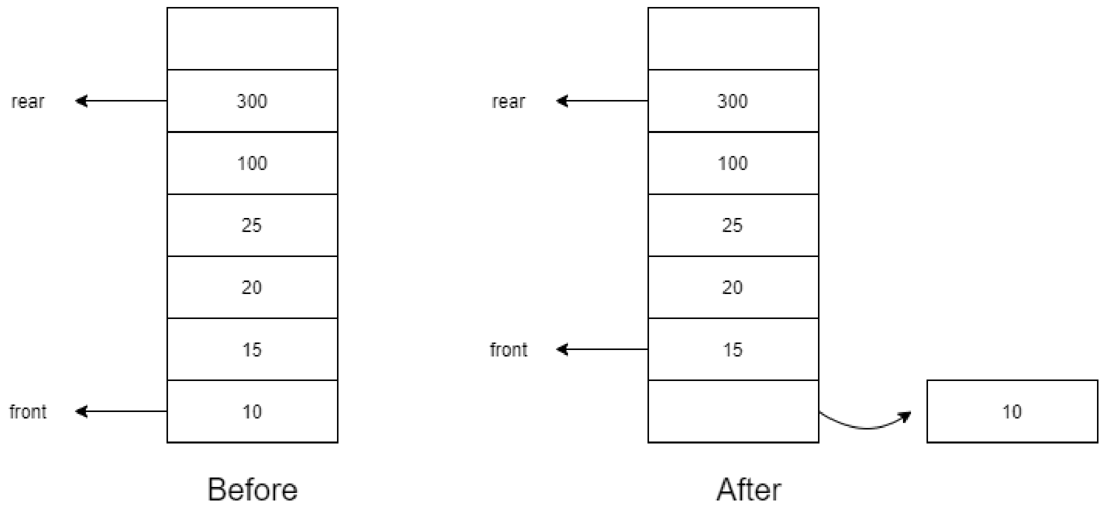


Figure 7: Dequeue function in Queue ADT

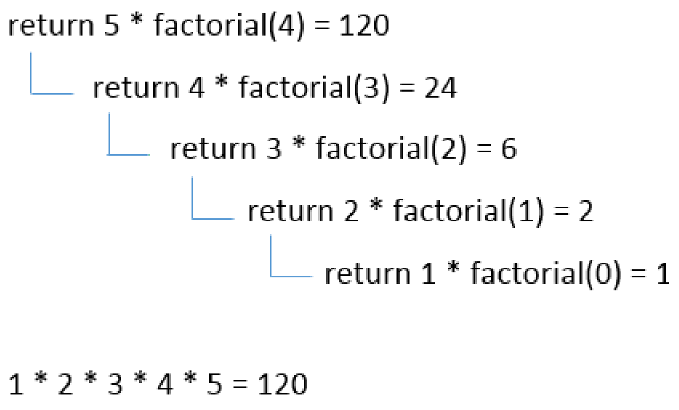
C. P2 Determine the operations of a memory stack and how it is used to implement function calls in a computer.

1. Memory stack

   Stack memory is a memory usage mechanism that enables machine memory to be used as a first-in-last-out buffer for temporary data storage. A register called the Stack Pointer is one of the most important components of stack memory operation. The stack pointer shows the current stack memory position and is automatically changed each time a stack operation is performed (Sciencedirect, 2021).

2. Function calls

   One common way to implement function calls using memory stack is Recursion. And a recursive function is a function that calls itself during its execution. The process may repeat several times, outputting the result and the end of each iteration.



return 5 * factorial(4) = 120
return 4 * factorial(3) = 24
return 3 * factorial(2) = 6
return 2 * factorial(1) = 2
return 1 * factorial(0) = 1
1 * 2 * 3 * 4 * 5 = 120

Figure 8: Recursion chart

Implement function calls:

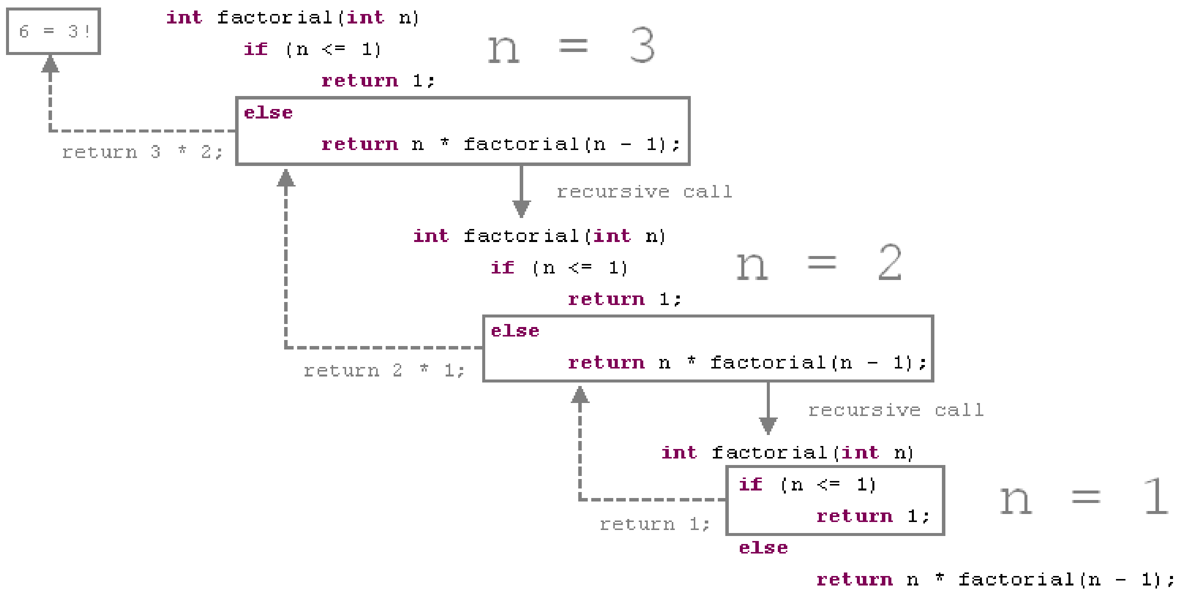


Figure 9: Recursion function call (algolist, 2021)

Use the recursive function to calculate the factorial of 3. The first step, n = 3 so the condition n <= 1 is false, the program goes down else and returns (n * factorial (n - 1)). Repeat until the condition n <= 1 is true, and returns 1. After that, we get a stack memory of {3, 2, 1} and multiply them together, resulting in 6.

D. P3 Using an imperative definition, specify the abstract data type for a software stack.

In an implementation, functions hide their inner workings. With the same concept in mind, data types can be developed. Client code, that is, other functions that use the structure, may use the interface provided by functions that implement operations on the structure. The structure's variables, as well as any helper functions, have their own implementation. For a structure produced without the use of OOP techniques.

This approach has two key advantages: by using the data form, design specifics can be overlooked, and various implementations of a structure can be substituted without affecting the operations seen by client code in the interface. In certain compiled languages, a unit or package containing the implementation can be changed and compiled separately without requiring the client code to be recompiled. A rebuild is only needed when the interface has changed.

Pre and post-conditions of Stack ADT

- Declare Stack

Stack(max_size: integer): s: Stack

pre true

post s = a new Stack instance

s.max_size = max_size

size(s) = 0

is_empty(s) = True

is_full(s) = False

invariant: size(s) >= 0

- Size of Stack

size(S: Stack): s: (0.. S.max_size)

pre true

post s = j - k **where** {S = Stack(), j * push(S, i), k * pop(S), j >= k}

{the sum of valid pushes - pops since S was created}

- Check if the Stack is empty

is_empty(S: Stack): s: bool

pre true

post s = (size(S) = 0)

- Check if the Stack is full

is_full(S: Stack): s: bool

pre true

post s = (size(S) = S.max_size)

- Get the value of the top element

top(S: Stack): s: Item

pre not is_empty(s)

post s = i **where** {i = pop(S), S = push(S, i)}

{defines i, S after i has been popped off and pushed back on again}

- Push new element to the Stack

push(S: Stack, i: Item): None

pre not is_full(S) **and** n = size(S)

post top(S) = i **and** size(S) = n + 1

- Remove the top element of Stack

pop(S: Stack): s : Item

pre not is_empty(S) **and** i = top(S) and n = size(S)

post s = i **and** size(S) = n - 1

E. References

En.wikipedia.org, 2021, Abstract data type, Wikipedia, viewed March 1, 2021, <https://en.wikipedia.org/wiki/Abstract_data_type>.

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>.

En.wikipedia.org, 2021, Stack (abstract data type), Wikipedia, viewed March 3, 2021, <https://en.wikipedia.org/wiki/Stack_(abstract_data_type)>.

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

Sciencedirect, 2021, Stack Memory - an overview | ScienceDirect Topics, Sciencedirect, viewed March 6, 2021, <https://www.sciencedirect.com/topics/engineering/stack-memory>.

Algolist.net, 2021, RECURSION (Java, C++) | Algorithms and Data Structures, viewed March 6, 2021, <http://www.algolist.net/Programming_concepts/Recursion>.