**ASSIGNMENT 1 FRONT SHEET**

|  |  |  |  |
| --- | --- | --- | --- |
| **Qualification** | **BTEC Level 5 HND Diploma in Computing** | | |
| **Unit number and title** | Unit **19: Data Structures & Algorithms** | | |
| **Submission date** |  | **Date Received 1st submission** |  |
| **Re-submission Date** |  | **Date Received 2nd submission** |  |
| **Student Name** | Nguyen Cong Nhat | **Student ID** | GCDC17202 |
| **Class** |  | **Assessor name** |  |
| **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** |  |

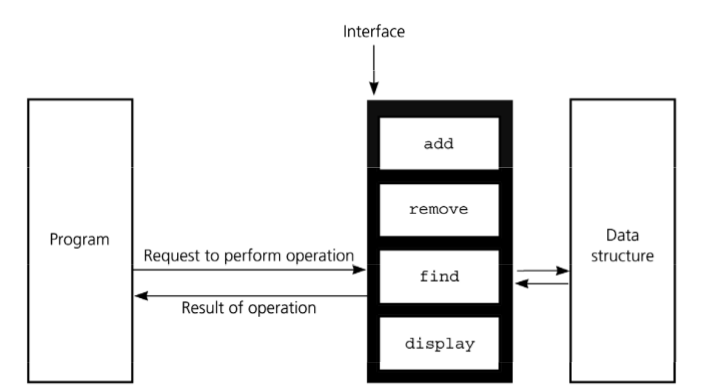
**Grading grid**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| P1 | P2 | P3 | M1 | M2 | M3 | D1 | D2 |
|  |  |  |  |  |  |  |  |

|  |  |  |
| --- | --- | --- |
| **❒ Summative Feedback: ❒ Resubmission Feedback:** | | |
| **Grade:** | **Assessor Signature:** | **Date:** |
| **Internal Verifier’s Comments:** | | |
| **Signature & Date:** | | |

1. **INTRODUCTION:**
2. **LO1 – EXAMINE ABSTRACT DATA TYPES, CONCRETE DATA STRUCTURES AND ALGORITHMS:**
   1. **P1 Create a design specification for data structures explaining the valid operations that can be carried out on the structures.**
3. **Definition of Abstract Data Type(ADT):**

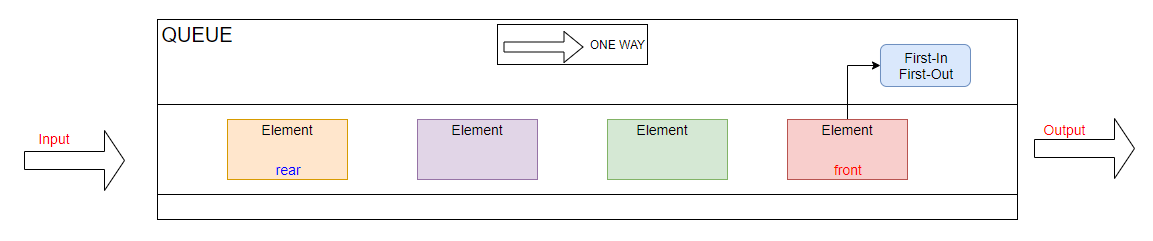
* **Definition:**
  + Abstract Data type (ADT) is a type (or class) for objects whose behaviour is defined by a set of value and a set of operations.
  + The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called “abstract” because it gives an implementation-independent view. The process of providing only the essentials and hiding the details is known as abstraction.



* + For example: When I use values like int, float, var. I just need to know what this type of data can work for, not the way that kind of data is done. Therefore, users only need to declare what kind of data can do, not need to explain the deployment process of that data.
  + Abstract Data Type have three ADTs namely List ADT, Stack ADT, Queue ADT.

1. **What is ADT Queue:**

* **Definition of Queue:**
  + Queue is an abstract data structure, is something similar to the queue in everyday life.
  + The queue structure is open at both ends. One end is always used to insert data (also known as line-up) and the other end is used to delete data (leave the row). The queue data structure follows the First-In-First-Out method, ie the data that is entered first will be accessed first.

****



* For example: As shown in the picture above, Queue is similar to traffic jams, cars will be first to exit and the cars will be arranged sequentially from front to back.
* **The following operations are needed to properly manage a queue:**
  + **First, we need to declare the Queue:**

public class Queue

{

public int max;

public int[] Q;

public int r = 0;

public int f = 0;

public Queue(int max, int[] Q)

{

this.max = max;

this.Q = Q;

}

}

* + - * *In the queue data structure, we always: (1)* ***dequeue*** *(delete) the data pointed by the* ***front pointer*** *and (2)* ***enqueue*** *(enter) the data into the queue by the help of the* ***rear cursor****.*
  + **isFull(): check to see if queue if full.**

public bool Isfull()

{

if ((((max - f) + r) % max) == (max-1))

{

Console.WriteLine("Queue is full");

return true;

}

else

return false;

}

* + - * ***max*** *is the number of elements in the queue.*
  + **isEmpty(): check to see if queue is empty.**

public bool Isempty()

{

if (f == r)

{

Console.WriteLine("Queue is empty");

return true;

}

else

return false;

}

* + - * *When* ***front = rear*** *then the Queue is empty.*
  + **enqueue(): put the element at the end of the queue.**

public void Enqueue(int x)

{

Q[r] = x;

r = (r + 1) % max;

}

* + - * *When we wants to add the new element to Queue, we will increase* ***rear***  *and assign* ***Q[r] = x****.*
  + **dequeue(): take the first element and remove it from the queue.**

public int Dequeue()

{

int De;

De = Q[f];

f = (f + 1) % max;

return De;

}

* + - * *When we wants to put out element in Queue, we will assign the element by a number* ***De = Q[f]*** *and increase* ***Front***
  + **Peek(): Print the front element that not remove it.**

public int Peek()

{

return Q[f];

}

* + - * *When we wants to print the front element in Queue, we just take element* ***Q[f]*** *in Queue.*
  + **size() – Return the number of elements in the queue.**

public int Enum()

{

if (r == f)

{

return 0;

}

else

return (((max - f) + r) % max);

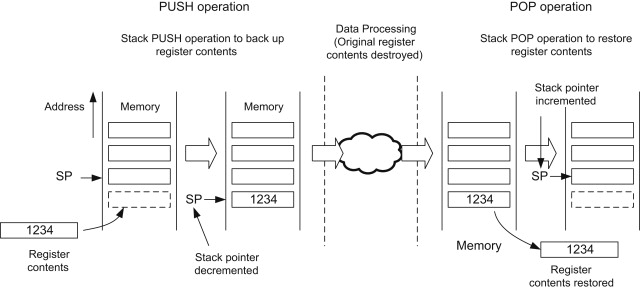
}

* + - * *When* ***rear = front****, the Queue is empty, else* ***Size of Queue*** *is calculated using the formula above.*

|  |  |  |
| --- | --- | --- |
| ADT Queue in VDM example: Interger type  Name: Example Queue  Symbol: Interger  Values: Array | | |
| Operator | Name | Type |
| Enqueue(x) | Insert element into Queue | Int 🡪 Array |
| Enqueue(x) | Insert element into Queue | Int 🡪 Array |
| Dequeue() | Remove element in Queue | Int |
| Peek() | Print front element in Queue | Int |
| isEmpty() | Check Queue’s empty | Boolean |
| isFull() | Check Queue full | Boolean |
| Size() | Check Queue’s Size | Int |

|  |  |  |
| --- | --- | --- |
| **Operation** | **Output** | **Queue Status** |
| enqueue(9) | \_\_\_ | (9) |
| enqueue(2) | \_\_\_ | (9, 2) |
| dequeue() | 9 | (2) |
| enqueue(7) | \_\_\_ | (2, 7) |
| dequeue() | 2 | (7) |
| dequeue() | 7 | () |
| dequeue() | “error” | () |
| isEmpty() | true | () |
| enqueue(2) | \_\_\_ | (2) |
| enqueue(1) | \_\_\_ | (2, 1) |
| size() | 2 | (2,1) |
| Peek() | 2 | (2,1) |

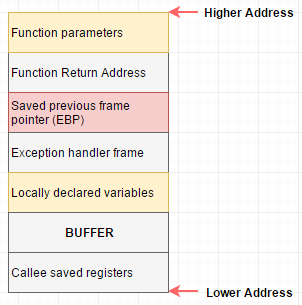
* 1. **P2 Determine the operations of a memory stack and how it is used to implement function calls in a computer.**
* **Definition of Memory stack:**
  + Stack memory is the memory to store information temporarily, when we want to read information in the direction of memory back to the memory written in it, regardless of the address of the data organized.
  + The stack memory acts as a queue, so that data can be written to the top of the stack, at the same time changing the data information stored in the next location in the direction of the queue's storage. When we read the information at the top of the stack, it is discarded and a new information is inserted in its place and moves all the stored information to one side of the stack.
* **Operation on memory stack:**
  + Push: insert data into stack.
  + Read: read the top of the stack.
  + Pop: delete data in the top and shifted up data in stack



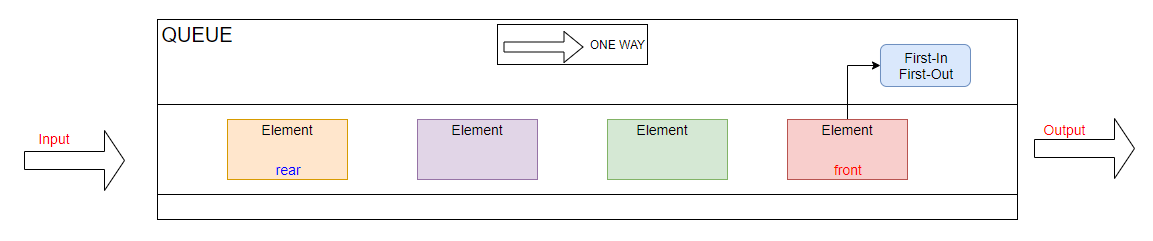
* **Memory segments:** The generic operating system grants each program process a specific memory area. This memory consists of different segments:



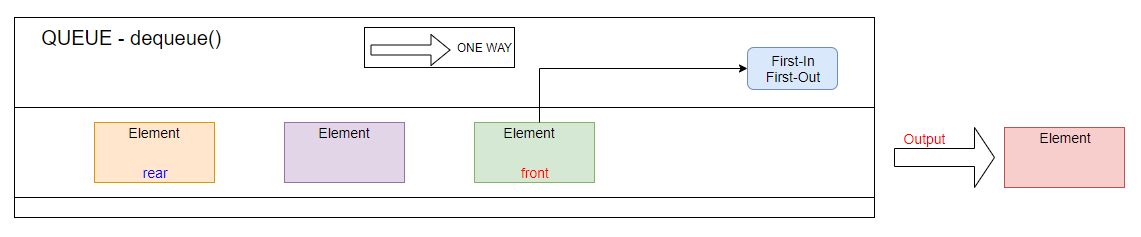
* + Stack: The stack is a place to store automatic variables along with information obtained after each function is called. When the function is called, the address where the information is returned is the environment of the caller of that function. And when the Stack pointer met the heap, the free memory ran out
  + Heap: Heap is the segment that usually allocates dynamic memory.
  + Uninitialized Data Segment: Uninitialized data begins at the end of the data segment and contains all global variables and static variables initialized to 0 or with no explicit initialization in the source code.
  + Initialized: The data segment is a part of the program's virtual address space, containing global and static variables that are programmed by the programmer.
  + Text: A device, a piece of text containing execution instructions.
* **A Memory stack is used to implement function calls in a computer:**
  + When a function is called, each stack frame in the call stack is used to hold:
    - Parameter passed to the function.
    - Pointer to current context.
    - The return address.
    - Local variables of a function under execution.
    - Stack pointer is adjusted.



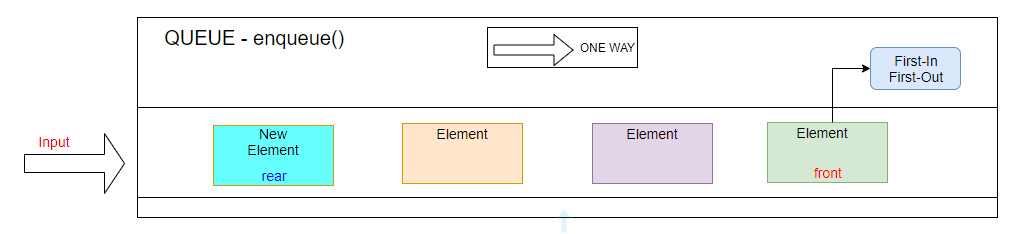
* When a function returns:
  + The top stack frame is removed.
  + Pointer and return address are reseted.
  + Stack pointer is adjusted.
  1. **M1 Illustrate, with an example, a concrete data structure for a First In First out (FIFO) queue.**
* **Overview of Queue:**

****

* **Dequeue():**

****

* **Enqueue():**

****

* **A real example in life as queue data structure:**



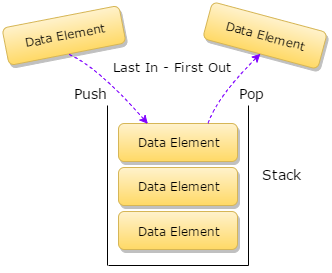
front

rear

* 1. **M2 Compare the performance of two sorting algorithms.**
  2. **D1 Analyse the operation, using illustrations, of two network shortest path algorithms, providing an example of each.**

1. **LO2 – SPECIFY ABSTRACT DATA TYPES AND ALGORITHMS IN A FORMAL NOTAION:**
   1. **P3 Using an imperative definition, specify the abstract data type for a software stack.**

* **Definition of ADT Stack:**
  + Stacks are a collection of elements like Arrays and Lists, but they are sorted from the beginning.
  + The organizing principle of Stack is Last-In First-Out (LIFO).



* + Stack's LIFO principle is like a vial containing the pills inside as the figure below. The first-placed pills will be on the bottom and the last-placed pills will be on the top, when we want to get the medicine, we will get the top-first pill because the bottle has only one end to take medicine and the tip is also the way for the pills to enter the vial.



* **Operations of ADT Stack:**
  + **First,** We have to create a Stack:

internal class Stack

{

static readonly int MAX = 1000;

int top;

int[] stack = new int[MAX];

public Stack()

{

top = -1;

}

}

First, I've declared the **top** element of the Stack and limited the Stack to 1000 elements (**MAX**).

* + **Push():** Add an element to the stack.

internal bool Push(int data)

{

if (top >= MAX)

{

Console.WriteLine("Stack Overflow");

return false;

}

else

{

stack[++top] = data;

return true;

}

}

When I want to **add** an element to the Stack, do I need to **check** if the Stack has been overflowed? If the Stack does not overflow, we increase the stack's **top value** to an element and assign the **stack[top] = new element**.

* + **Pop():** Removes an element into the stack.

internal int Pop()

{

if (top < 0)

{

Console.WriteLine("Stack Underflow");

return 0;

}

else

{

int value = stack[top--];

return value;

}

}

When we want to **delete** an element of Stack, we need to check whether the Stack exists? If the stack exists we only need to **reduce** the **top value** to one unit.

* + **Peek():** Print a top element and that element is not removed.

internal void Peek()

{

if (top < 0)

{

Console.WriteLine("Stack Underflow");

return;

}

else

Console.WriteLine("The topmost element of Stack is : {0}", stack[top]);

}

When we want to print the top stack element, we need to check if the Stack exists? Then print out the element **stack[top].**

* + **isEmpty():** Check for the existence of the stack.

bool IsEmpty()

{

if (top == -1)

{

return true;

}

else

{

return false;

}

}

When we want to check for the existence of Stack, we check the Stack's **top value** if **top = -1**, then Stack is empty.

* + **isFull():** Check the stack's fulling..

bool IsFull()

{

if (top == MAX - 1)

{

return true;

}

else

{

return false;

}

}

If the top value of **Stack = with the value of max-1**. The Stack is **fulling**, called **Stack overflow**.

* + **PrintStack():** Print all elements of Stack.

internal void PrintStack()

{

if (top < 0)

{

Console.WriteLine("Stack Underflow");

return;

}

else

{

Console.WriteLine("Items in the Stack are :");

for (int i = top; i >= 0; i--)

{

Console.WriteLine(stack[i]);

}

}

}

|  |  |  |
| --- | --- | --- |
| ADT Stack in VDM example: Interger type  Name: Example Stack  Symbol: Interger  Values: Array | | |
| Operator | Name | Type |
| isEmpty() | Check for the existence of the stack. | Boolean |
| Push(x) | Insert element into Stack | Int 🡪 Array |
| Push(x) | Insert element into Stack | Int 🡪 Array |
| Push(x) | Insert element into Stack | Int 🡪 Array |
| Pop() | Remove element in Stack | Interger |
| Peek() | Print element that not remove | Interger |
| isFull() | Check Stack full | Boolean |
| PrintStack() | Print all elenment | Array |

|  |  |  |
| --- | --- | --- |
| **Operation** | **Output** | **Stack Status** |
| isEmpty() | true | () |
| Push(32) | Successfully | (32) |
| Push(99) | Successfully | (99, 32) |
| Push(56) | Successfully | (56, 99, 32) |
| Pop() | 56 | (99, 32) |
| Peek() | 99 | (99, 32) |
| isFull() | “false” | (99, 32) |
| PrintStack() | (99, 32) | (99, 32) |

* + **M3 Examine the advantages of encapsulation and information hiding when using an ADT.**
  + **D2 Discuss the view that imperative ADTs are a basis for object orientation and, with justification, state whether you agree.**

1. **SUMMARY**
2. **REFERENCE:**

<https://en.wikipedia.org/wiki/Queue_(abstract_data_type)?fbclid=IwAR1yMV9DB3lXp-RXuV3GzN7L9tsS5YspGdCufnxSMmYda-jws_HNuukiUjQ>

<https://en.wikipedia.org/wiki/Abstract_data_type?fbclid=IwAR0Np0TZ1Z4V4IASLwtwc6gaVZW86jaipQJ6AJ6m1rThPQO_co52EW1_B2E>

<https://www.geeksforgeeks.org/abstract-data-types/>

<https://vietjack.com/cau-truc-du-lieu-va-giai-thuat/cau-truc-du-lieu-hang-doi.jsp>

<https://medium.com/@lucasmagnum/sidenotes-stack-abstract-data-type-and-data-structure-30e74bb5303a>

<https://www.geeksforgeeks.org/memory-layout-of-c-program/>

<https://medium.com/@lucasmagnum/sidenotes-stack-abstract-data-type-and-data-structure-30e74bb5303a>