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:

Table of Contents

I. INTRODUCTION: 4

II. LO1 – EXAMINE ABSTRACT DATA TYPES, CONCRETE DATA STRUCTURES AND ALGORITHMS: 5

1. P1 Create a design specification for data structures explaining the valid operations that can be carried out on the structures. 5

a) Definition of Abstract Data Type(ADT): 5

b) What is ADT Queue: 6

2. P2 Determine the operations of a memory stack and how it is used to implement function calls in a computer. 12

3. M1 Illustrate, with an example, a concrete data structure for a First In First out (FIFO) queue. 15

4. M2 Compare the performance of two sorting algorithms. 17

5. D1 Analyse the operation, using illustrations, of two network shortest path algorithms, providing an example of each. 22

a) Dijkstra’s algorithm: (Wikipedia, n.d.) (GeeksforGeeks, n.d.) 23

b) Bellman–Ford algorithm: 30

III. LO2 – SPECIFY ABSTRACT DATA TYPES AND ALGORITHMS IN A FORMAL NOTAION: 33

1. P3 Using an imperative definition, specify the abstract data type for a software stack. 33

2. M3 Examine the advantages of encapsulation and information hiding when using an ADT. 39

3. D2 Discuss the view that imperative ADTs are a basis for object orientation and, with justification, state whether you agree. 40

IV. REFERENCE: 41

I. INTRODUCTION:

I'm working as in-house software developer for Softnet Development Ltd, a software body-shop providing network provisioning solutions. My company is part of a collaborative service provisioning development project and my company has won the contract to design and develop a middleware solution that will interface at the front-end to multiple computer provisioning interfaces including SOAP, HTTP, JML and CLI, and the back-end telecom provisioning network via CLI .

My account manager has made I technical project leader and my role is to inform them about designing and implementing abstract data types. I have been asked to create a presentation for all collaborating partners on how ADTs can be utilised to improve software design, development and testing. Further, I have been asked to write an introductory report for distribution to all partners on how to specify abstract data types and algorithms in a formal notation. This report have 2 parts:

- Examine abstract data types, concrete data structures and algorithms.
- Specify abstract data types and algorithms in a formal notation.

II. 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.

a) Definition of Abstract Data Type(ADT):

❖ **Definition:** (GeeksforGeeks, n.d.) (Wikipedia, n.d.)

- 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.

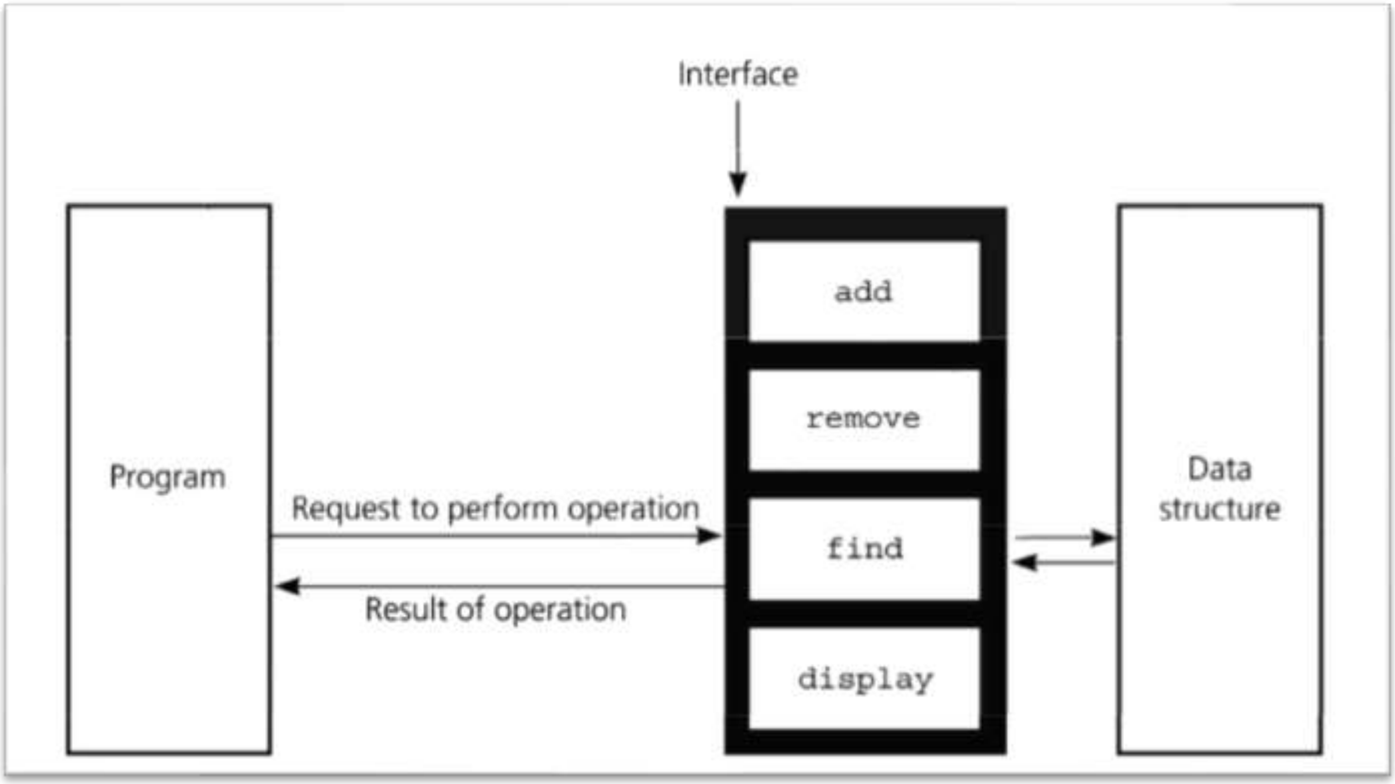


Figure 1 Overview of ADT

- 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.

b) What is ADT Queue:

- **Definition of Queue:** (Wikipedia, n.d.) (Vietjack, n.d.)
  - 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.

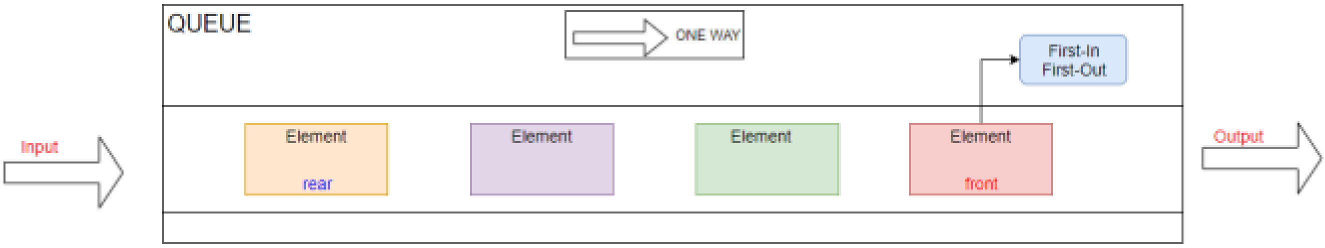


Figure 2 Overview of ADT Queue

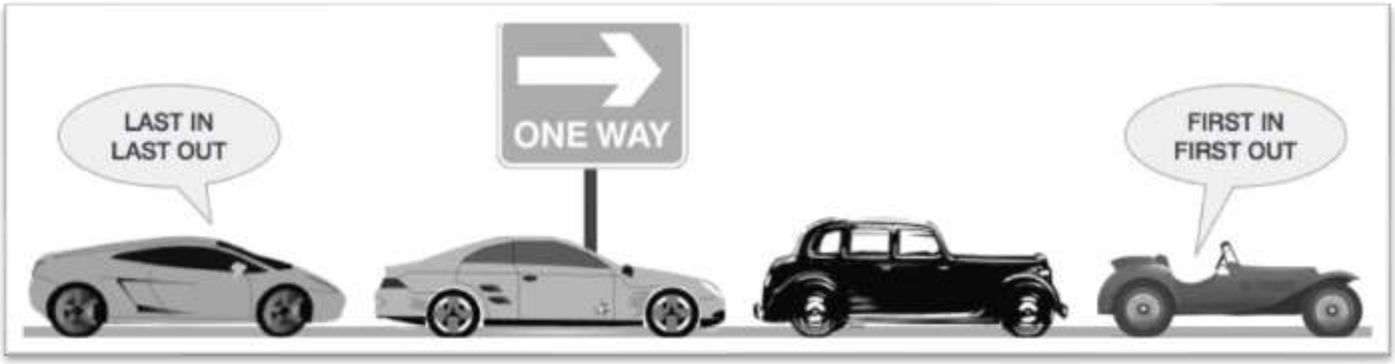


Figure 3 Example of ADT Queue

  - 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.**

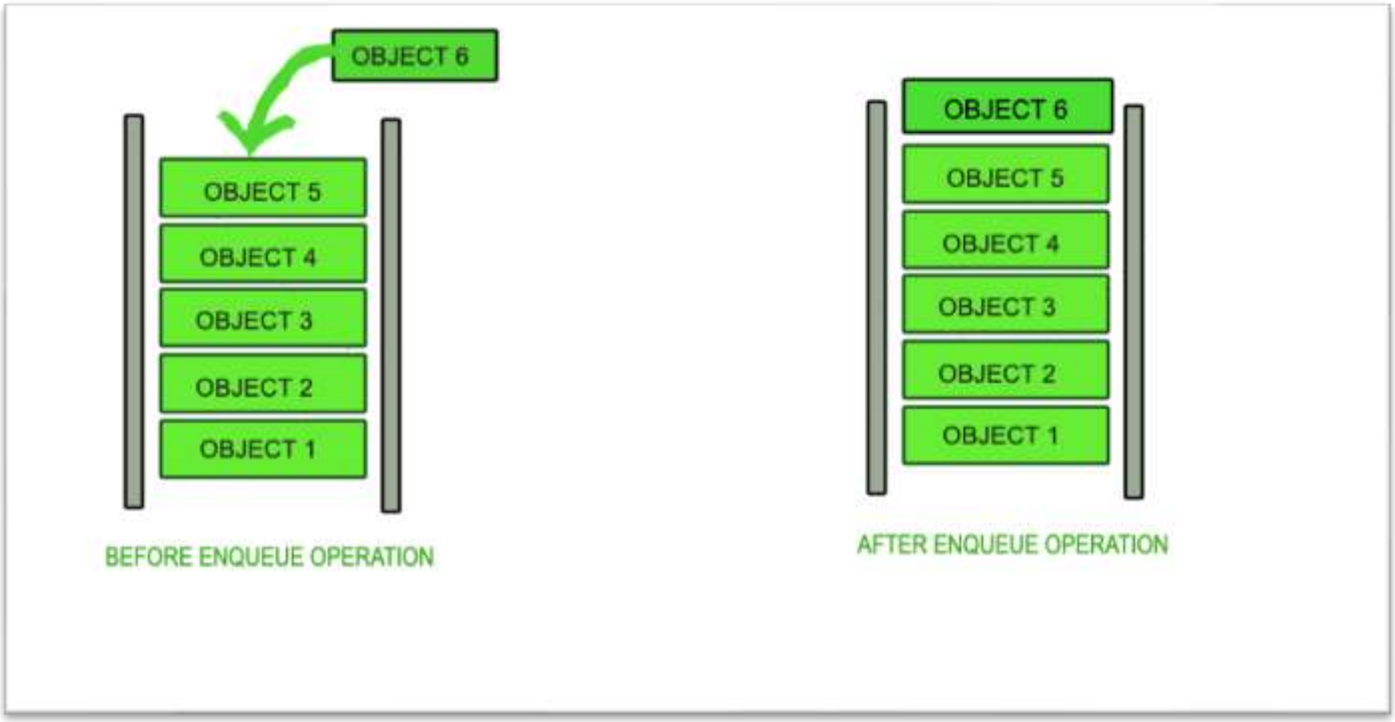


Figure 4 Operation - enqueue

```
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.**

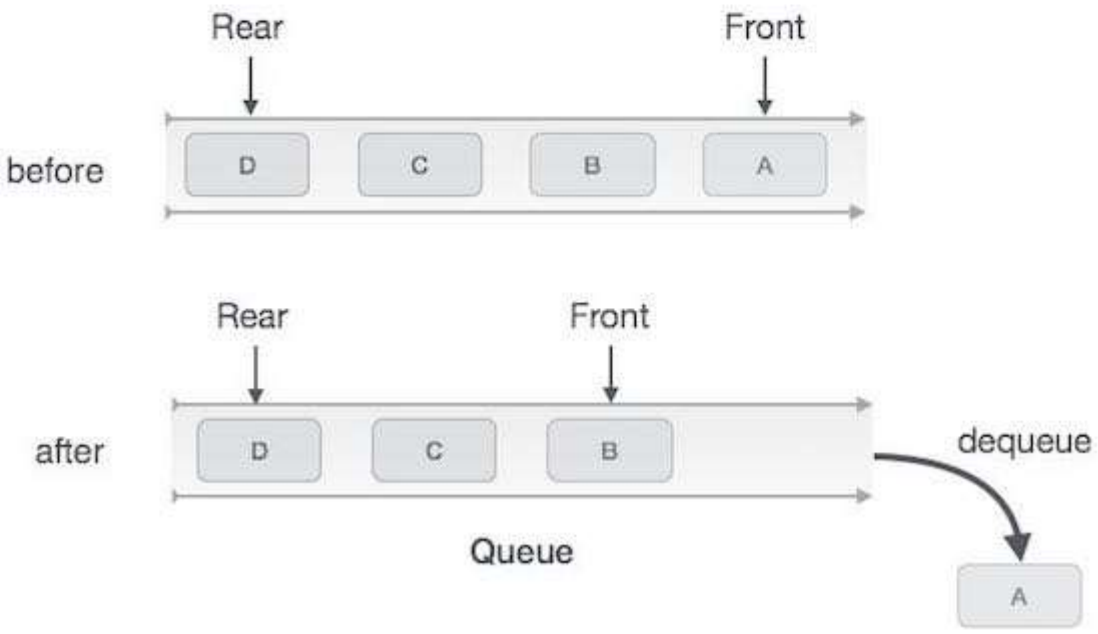


Figure 5 Operation - dequeue

```
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.**

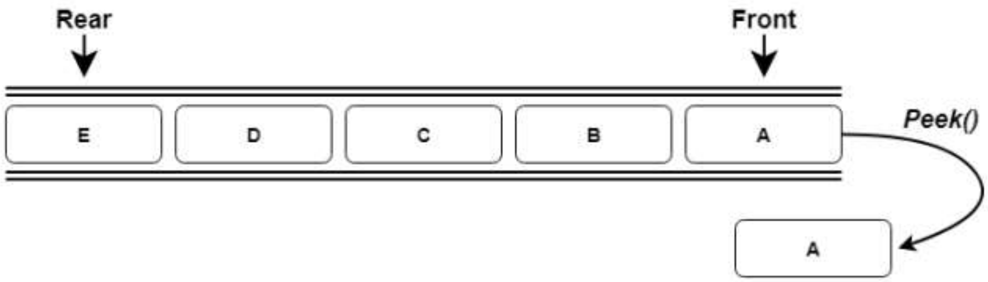


Figure 6 Queue Operation - Peek

```
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

Table 1 VDM table of ADT Queue

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)

Table 2f Operation in Queue

2. 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.

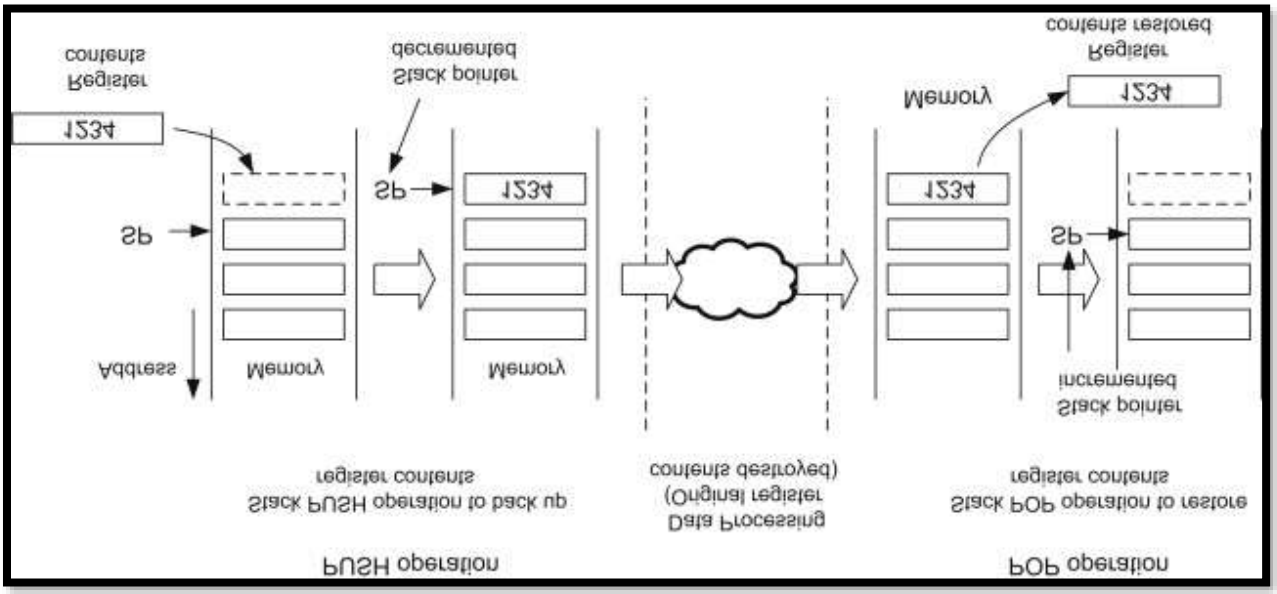


Figure 7 Overview of Memory stacck

- **Memory segments:** The generic operating system grants each program process a specific memory area. This memory consists of different segments: (Geeksforgeeks, n.d.) (Edux, n.d.)

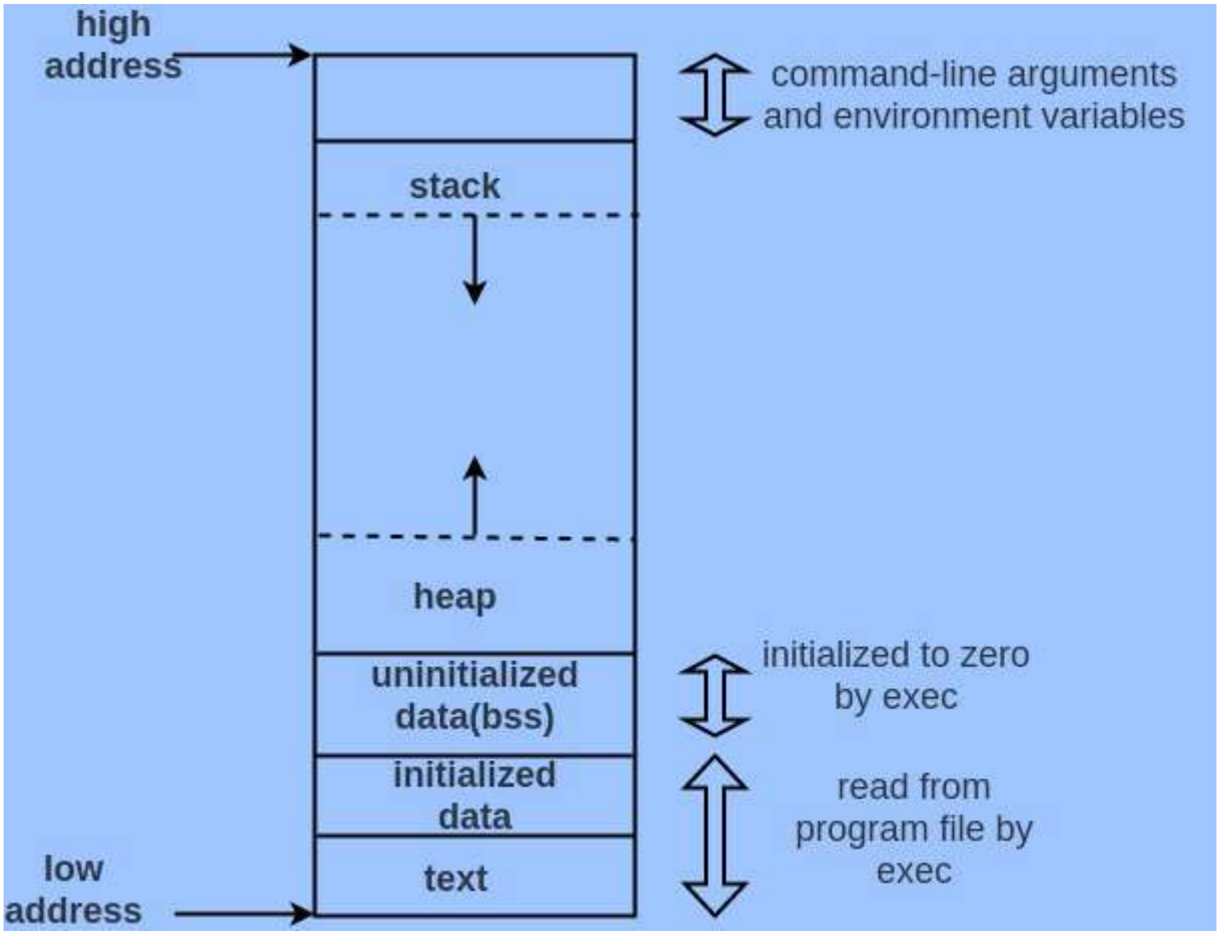

high address
command-line arguments and environment variables
stack
heap
uninitialized data(bss)
initialized to zero by exec
initialized data
text
read from program file by exec
low address

Figure 8 Memory 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.

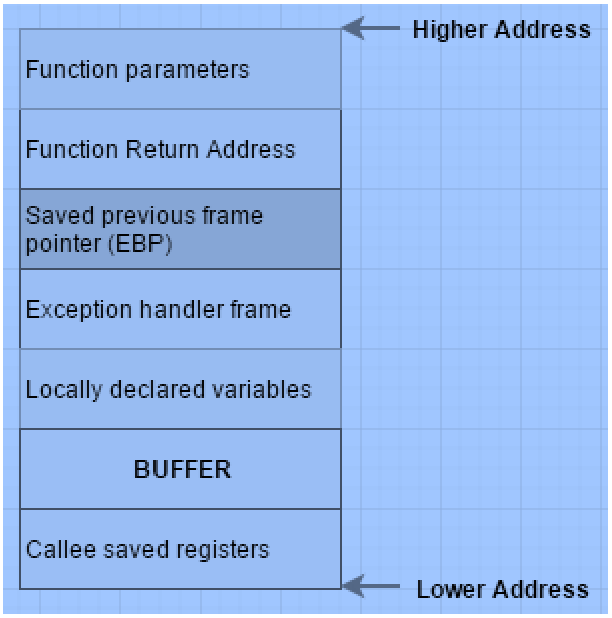


Figure 9 Function calls

- When a function returns:
  - The top stack frame is removed.
  - Pointer and return address are reseted.
  - Stack pointer is adjusted.

3. **M1 Illustrate, with an example, a concrete data structure for a First In First out (FIFO) queue.**

- **Overview of Queue:**

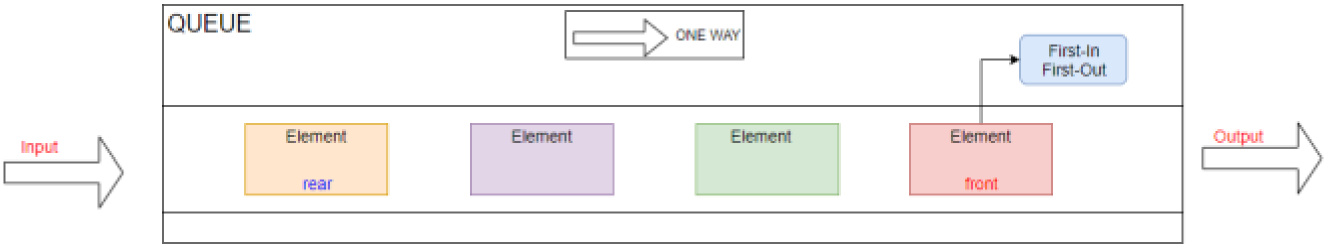


Figure 10 Overview of Queue for example

- As we discussed in section P1, Queue arranges and stores information in one way with the First-In First-Out principle. As shown in the image, the elements are stored and moved to one side if the Queue is affected. The first element is marked as the front element and the last element is marked as rear. When we want to manipulate Queue, we only need to manipulate two parameters: rear and front.

- **Dequeue():**

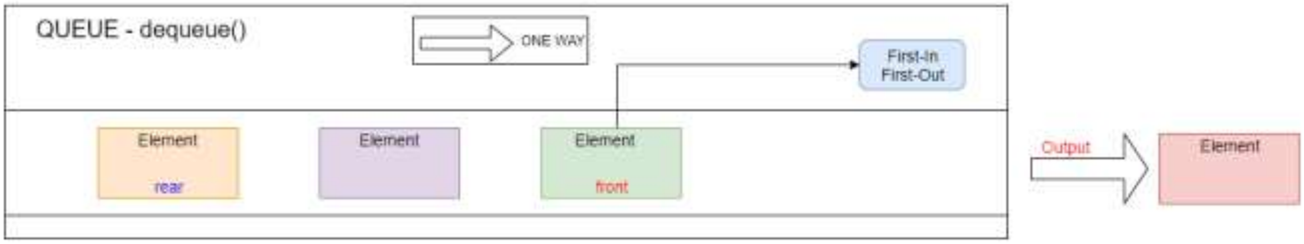


Figure 11 Dequeue for example

- **Dequeue()** is the function of taking an element marked as a front element out of Queue. When we want to get the value of the front parameter, we manipulate the front with the formula front = (front + 1)% max.

- **Enqueue():**

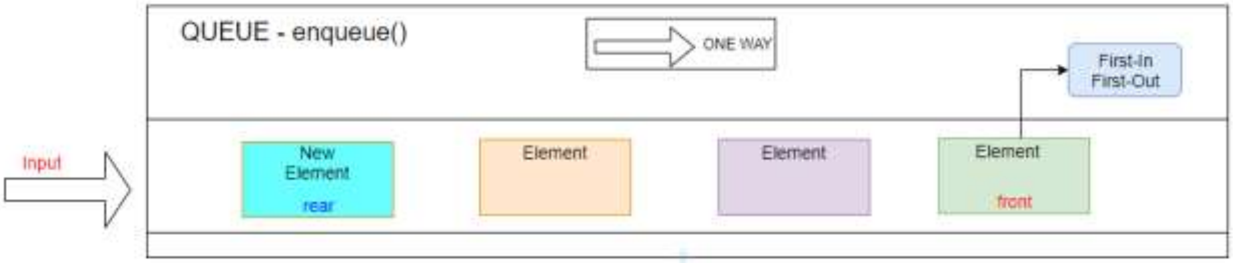


Figure 12 Enqueue for example

- **Enqueue()** is the function of adding elements to Queue. When we want to add we manipulate the parameter rear = (rear + 1)% max.

❖ **A real example in life as queue data structure (Traffic jam):**

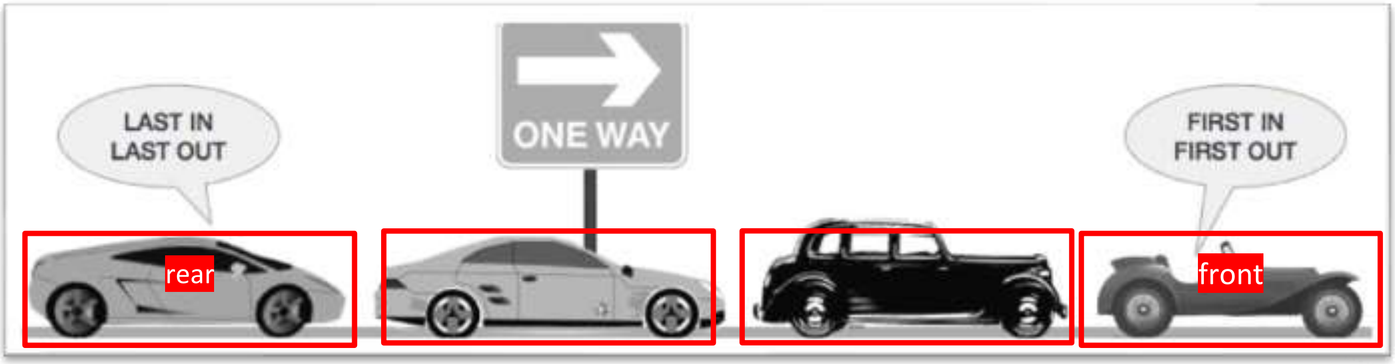


Figure 13 Real example for Queue

- As we can see in the picture above, the case of cars getting off the road is the same as the ADT Queue. All vehicles must move in a certain direction. The first vehicle to be removed from the congested convoy is the one marked front in the figure, whereas the last vehicle in the congested range is the one marked rear in the image. Vehicles will be exited from the convoy convoy in order from right to left. Like ADT Queue the elements are removed from a Queue in the same order.

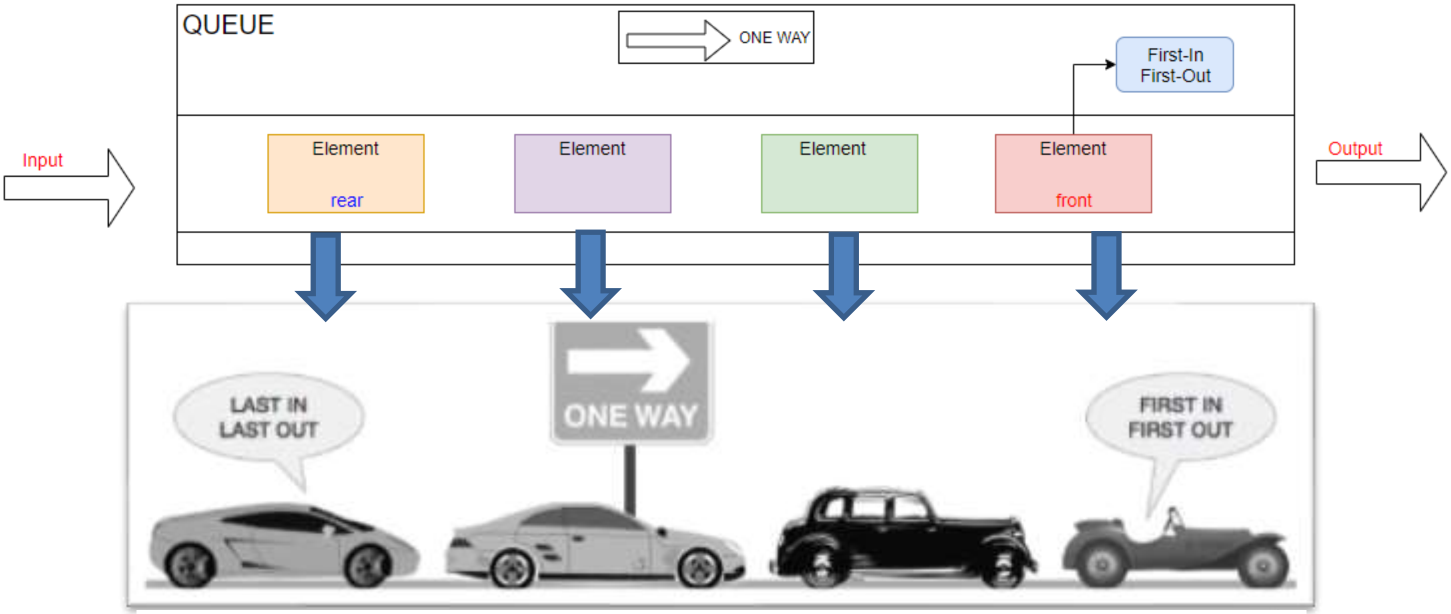


Figure 14 Comparee Real example & Definition

4. M2 Compare the performance of two sorting algorithms.

First, I use the C # language to implement two different sorting algorithms, and I use Visual Studio as the working environment to ensure fairness for the two algorithms. I use the `Diagnostics` library (using System.Diagnostics;) to calculate the execution time of the two algorithms.

```
st.Start();
       long time = 600000000L;

st.Stop();

Console.WriteLine("\n{0} giay", st.Elapsed.ToString());
       if (Stopwatch.IsHighResolution)
           Console.WriteLine("Timed with Hi res");
            else Console.WriteLine("Not Timed with Hi res");
```

a) Sorting Array in C# - Selection method:

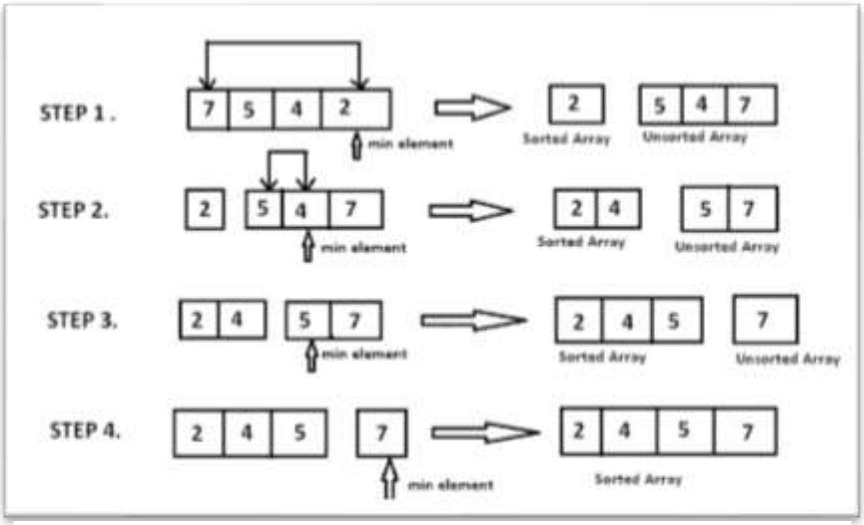


Figure 15 Overview of Sorting - Seletion method

In the selection sort algorithm, the steps are as follows: Take the top value of the list as the smallest value (or the largest depending on the arrangement) and browse through the list, when finding a smaller value, take that as the smallest value. After browsing the list, get the element with the smallest value, change the position with the position of the head of the list. The element at the top of the list is now in the correct position. repeat the list browsing process, with the list starting at the second element onwards. And repeat the process above until the list no longer has the element, the sorting process ends.

```
//Sorting Array - Selection method
               for (i = 0; i < (n - 1); i++)
               {
                   int min = i;
                   for (j = 0; j < n; j++)
                   {
                       if (arr[j] < arr[min])
                       {
                           min = j;
                       }

                   }
                   int tmp = arr[min];
                   arr[min] = arr[j];
                   arr[j] = tmp;
               }
```

b) Sorting Array in C# - Bubble method:

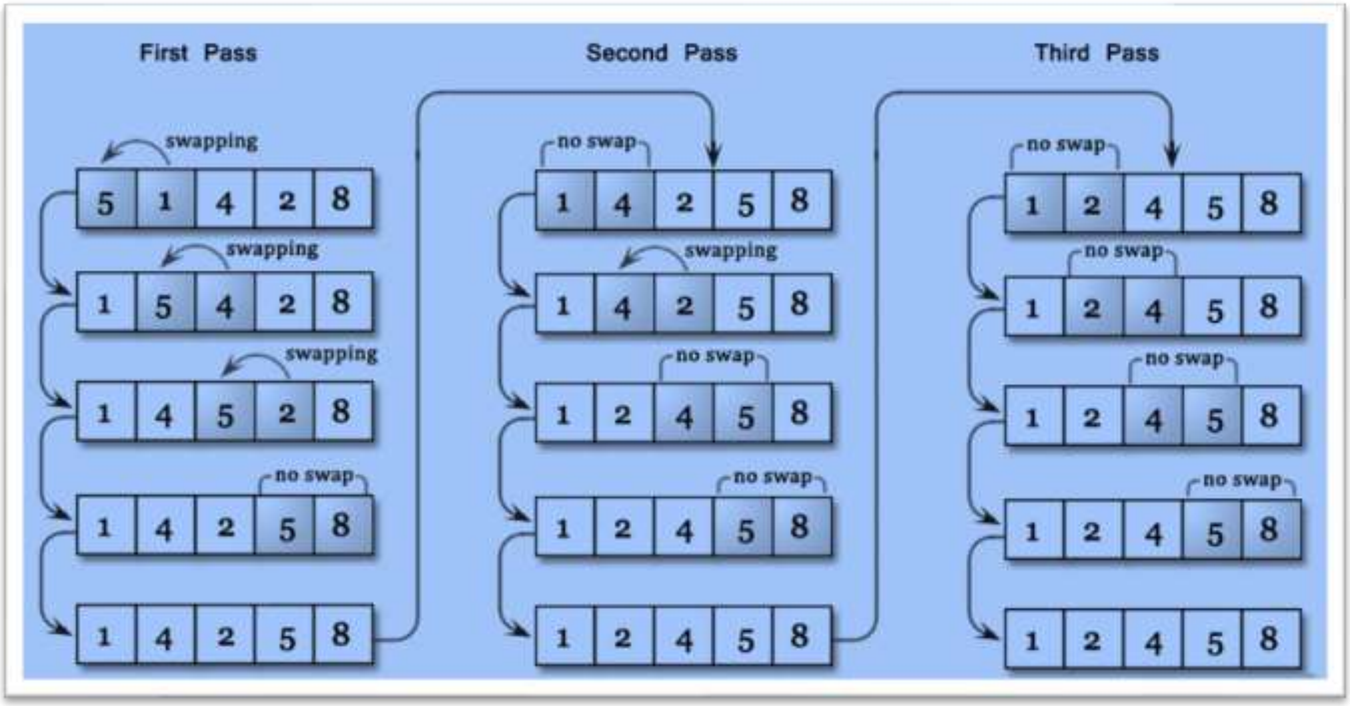
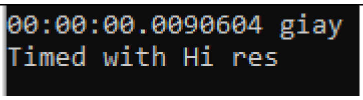
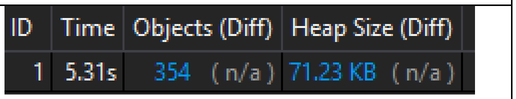
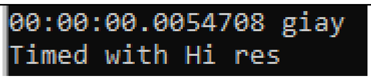
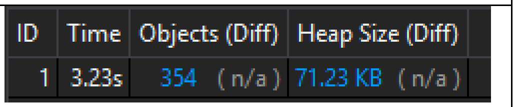
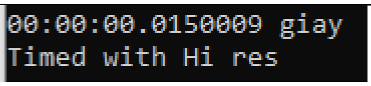
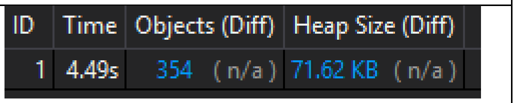
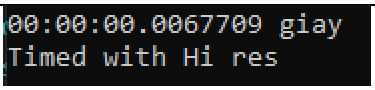
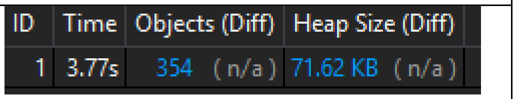
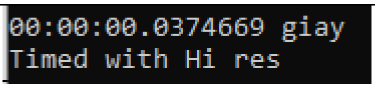
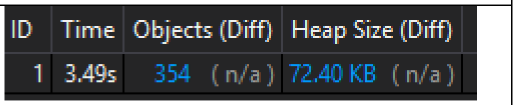
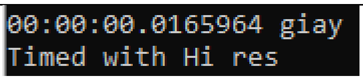
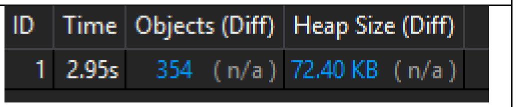
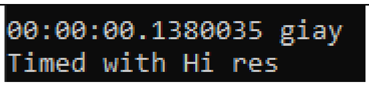
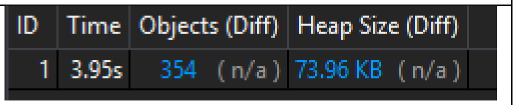
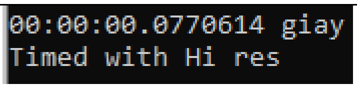
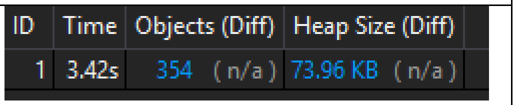
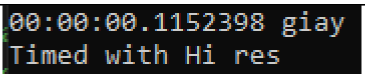
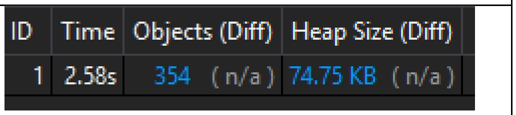
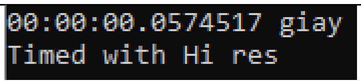
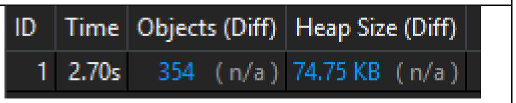
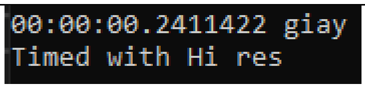
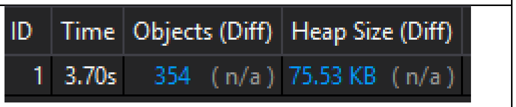
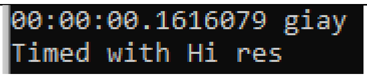
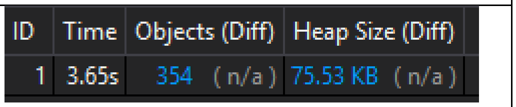
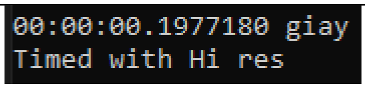
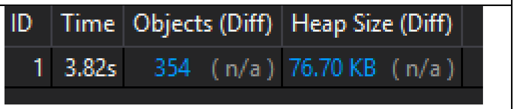
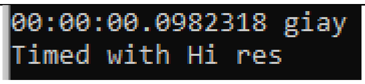
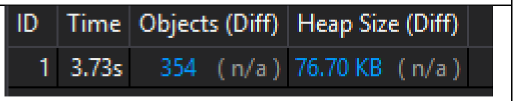
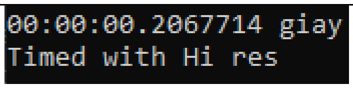
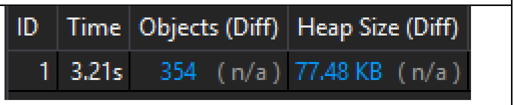


Figure 16 Overview Sorting - Bubble method

Bubble soft is a simple sorting algorithm, with a list of n elements, its algorithm starts to browse from the beginning to the end of the list, checking in adjacent pairs if the order of the 2 elements in pairs. If not, then change the position of the 2 elements to continue to the end of the list and exchange the wrong pairs. After browsing the entire list, the position at the end of the list is the exact position and repeat the browse with the new list of n-1 elements. Finally repeat the steps above until the list no longer has elements to browse.

```
//Sorting Array - Bubble method
            for (i = 0; i < n; i++)
            {
                for (j = i + 1; j < n; j++)
                {
                    if (arr[j] < arr[i])
                    {
                        int tmp = arr[i];
                        arr[i] = arr[j];
                        arr[j] = tmp;
                    }
                }
            }
```

c) **Compare the performance of Selection Sorting algorithms & Bubble Sorting algorithms:**

Length of Array	Algorithms	Time perform	Memory usage
100	**Selection Sorting**	 00:00:00.0090604 giay Timed with Hi res	 ID Time Objects (Diff) Heap Size (Diff) 1 5.31s 354 (n/a) 71.23 KB (n/a)
	Bubble Sorting*	 00:00:00.0054708 giay Timed with Hi res	 ID Time Objects (Diff) Heap Size (Diff) 1 3.23s 354 (n/a) 71.23 KB (n/a)
200	**Selection Sorting**	 00:00:00.0150009 giay Timed with Hi res	 ID Time Objects (Diff) Heap Size (Diff) 1 4.49s 354 (n/a) 71.62 KB (n/a)
	Bubble Sorting*	 00:00:00.0067709 giay Timed with Hi res	 ID Time Objects (Diff) Heap Size (Diff) 1 3.77s 354 (n/a) 71.62 KB (n/a)
400	**Selection Sorting**	 00:00:00.0374669 giay Timed with Hi res	 ID Time Objects (Diff) Heap Size (Diff) 1 3.49s 354 (n/a) 72.40 KB (n/a)
	Bubble Sorting*	 00:00:00.0165964 giay Timed with Hi res	 ID Time Objects (Diff) Heap Size (Diff) 1 2.95s 354 (n/a) 72.40 KB (n/a)
800	**Selection Sorting**	 00:00:00.1380035 giay Timed with Hi res	 ID Time Objects (Diff) Heap Size (Diff) 1 3.95s 354 (n/a) 73.96 KB (n/a)
	Bubble Sorting*	 00:00:00.0770614 giay Timed with Hi res	 ID Time Objects (Diff) Heap Size (Diff) 1 3.42s 354 (n/a) 73.96 KB (n/a)
1000	**Selection Sorting**	 00:00:00.1152398 giay Timed with Hi res	 ID Time Objects (Diff) Heap Size (Diff) 1 2.58s 354 (n/a) 74.75 KB (n/a)
	Bubble Sorting*	 00:00:00.0574517 giay Timed with Hi res	 ID Time Objects (Diff) Heap Size (Diff) 1 2.70s 354 (n/a) 74.75 KB (n/a)
1200	**Selection Sorting**	 00:00:00.2411422 giay Timed with Hi res	 ID Time Objects (Diff) Heap Size (Diff) 1 3.70s 354 (n/a) 75.53 KB (n/a)
	Bubble Sorting*	 00:00:00.1616079 giay Timed with Hi res	 ID Time Objects (Diff) Heap Size (Diff) 1 3.65s 354 (n/a) 75.53 KB (n/a)
1500	**Selection Sorting**	 00:00:00.1977180 giay Timed with Hi res	 ID Time Objects (Diff) Heap Size (Diff) 1 3.82s 354 (n/a) 76.70 KB (n/a)
	Bubble Sorting*	 00:00:00.0982318 giay Timed with Hi res	 ID Time Objects (Diff) Heap Size (Diff) 1 3.73s 354 (n/a) 76.70 KB (n/a)
1700	**Selection Sorting**	 00:00:00.2067714 giay Timed with Hi res	 ID Time Objects (Diff) Heap Size (Diff) 1 3.21s 354 (n/a) 77.48 KB (n/a)

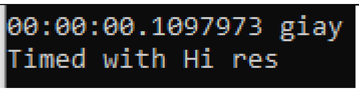
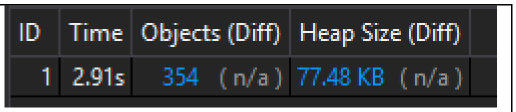
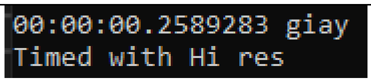
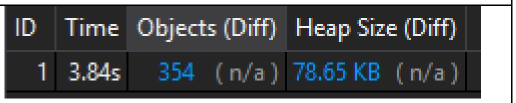
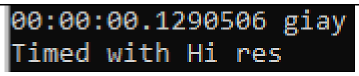
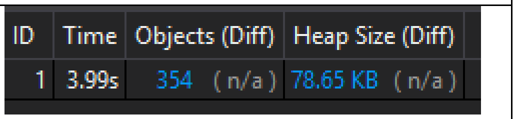
	Bubble Sorting*	 00:00:00.1097973 giay Timed with Hi res	 ID Time Objects (Diff) Heap Size (Diff) 1 2.91s 354 (n/a) 77.48 KB (n/a)
1999	**Selection Sorting**	 00:00:00.2589283 giay Timed with Hi res	 ID Time Objects (Diff) Heap Size (Diff) 1 3.84s 354 (n/a) 78.65 KB (n/a)
	Bubble Sorting*	 00:00:00.1290506 giay Timed with Hi res	 ID Time Objects (Diff) Heap Size (Diff) 1 3.99s 354 (n/a) 78.65 KB (n/a)

Table 3 Compare Selection Sort algorithm & Bubble algorithm

- On the table comparing the two algorithms Selections and Bubble, I use 9 lengths of Array 100,200,400,800,1000,1200,1500,1700 and 1999 to analyze the performance of the two algorithms more clearly.
- The first thing we notice is that the Performance RAM USAGE in the two algorithms is the same at all Array lengths.
- Next in terms of uptime, Bubble Sorting algorithm's uptime is faster than Selection Sorting algorithm at all levels I've tested.
- It is also possible that Selection Sorting algorithm works better in other languages, environments or at different Array lengths, so it is not possible to evaluate Selection Sort as an optimal algorithm lower than Bubble Sort.

	Selection Sorting	**Bubble Sorting**
Definition	Largest element is selected and swapped with the last element (in case of ascending order).	Adjacent element is compared and swapped
Complexity	$O(n^2)$	O(n)
Efficiency	Improved efficiency as compared to bubble sort	Inefficient
Stable	No	Yes
Method	Selection	Exchanging

Table 4 Compare characteristics Selection & Bubble

5. **D1 Analyse the operation, using illustrations, of two network shortest path algorithms, providing an example of each.**

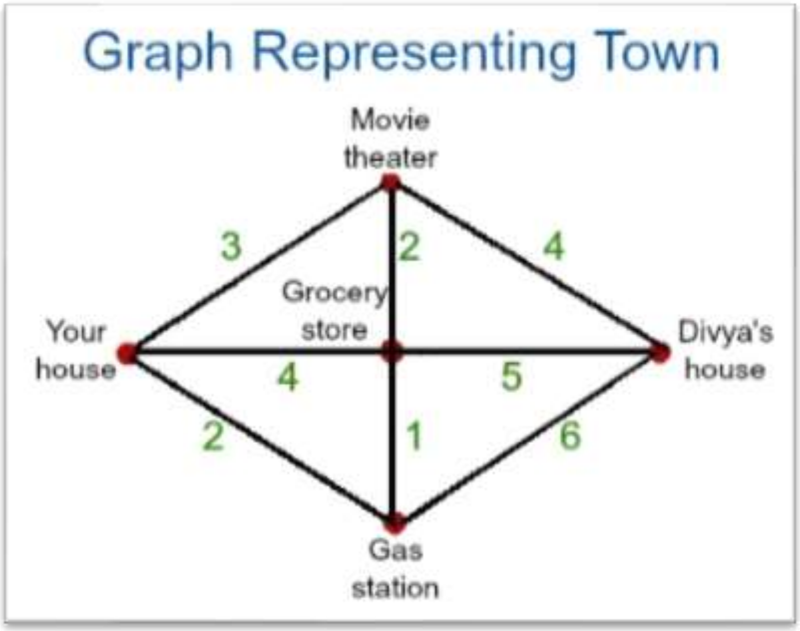


Figure 17 Shortest path graph algorithms

Have you ever used google maps to find your way? You have questions about how it can find the shortest path for you to move. As above, you want to go from your house to the house of Divya's house, which way would you choose? This is the problem of finding the shortest path. In the algorithm, it is called the shortest path algorithm. Now we will explore two algorithms to find the shortest path: **Dijkstra algorithm & Bellman-Ford algorithm.**

a) **Dijkstra’s algorithm:** (Wikipedia, n.d.) (GeeksforGeeks, n.d.)

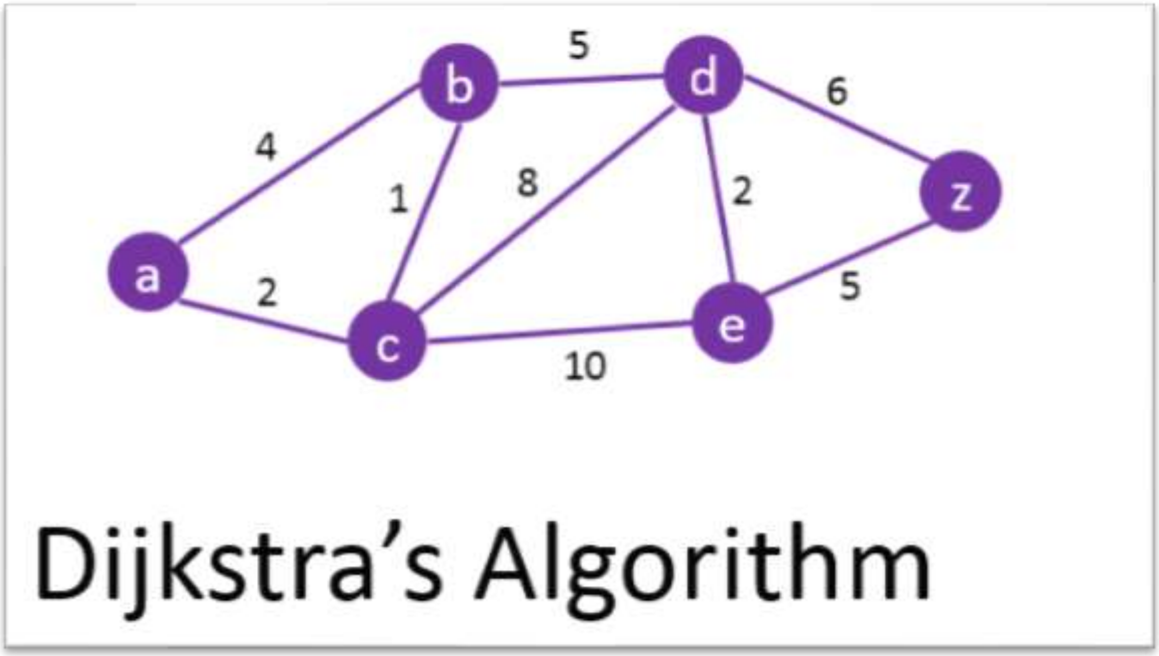


Figure 18 Dijkstra's Algorithm

- **Definition of Dijkstra’s algorithm:** Dijkstra's algorithm (or Shortest Path Dijkstra) is an algorithm for finding the shortest path between nodes in a graph, which can represent points and edges in the graph. It was conceived by computer scientist Edsger W. Dijkstra in 1956. Dijkstra's algorithm can be used to find the shortest distance or minimum cost depending on what is shown in the graph. Two typical features of each node in the algorithm are: Distance value, Status label.
- **Operation of Dijkstra’s:**
  - **Step 1**: Mark the points in the chart, this time the chart includes a set of possible point nodes.
  - **Step 2**: Identify the vertices (nodes) that can connect with each other and calculate the distance between the vertices that can be connected to each other.
  - **Step 3**: Label the current vertex as accessed by placing an X on it. When a vertex is accessed, we will not look back at it.
  - **Step 4**: Among the vertices you just marked, find a vertex with the smallest marker and move to that vertex. And repeat step 2.
  - **Step 5**: Once you've reached the end point The set of vertices marked with X is your shortest path.

- ❖ **Example and solution:**

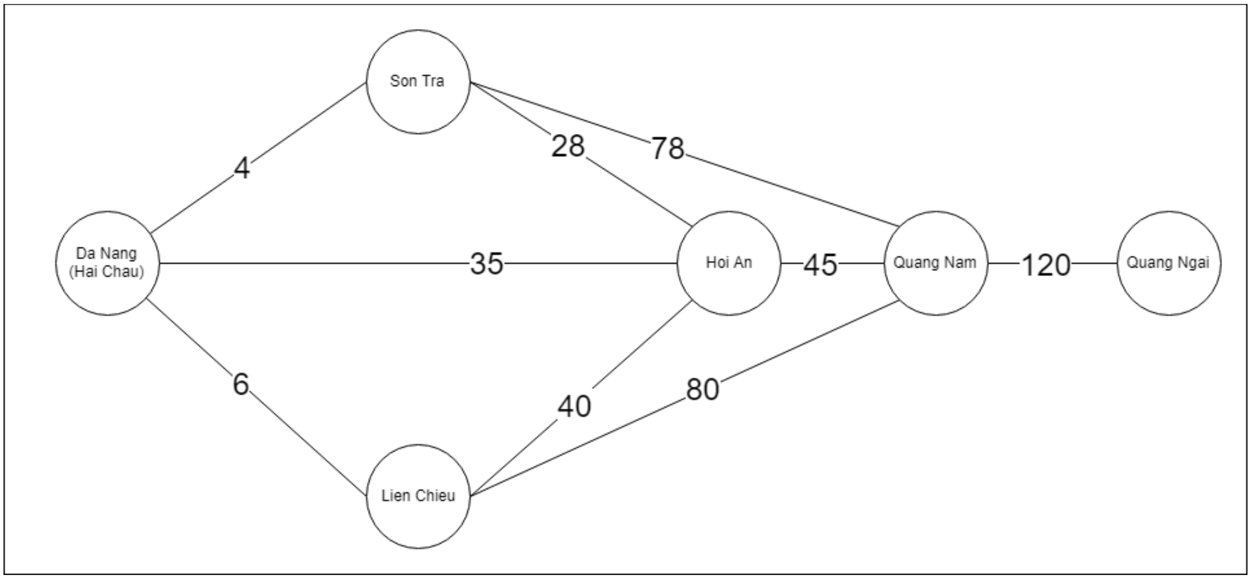


Figure 19 Example Dijkstra's algorithm

- ○ **Problem:** Above is a graph showing the ways from Da Nang (Hai Chau) to Quang Ngai. There are many ways to go from Da Nang to Quang Ngai, our problem now is to find the shortest distance between two points. There are peaks: Da Nang (Hai Chau), Son Tra, Lien Chieu, Hoi An, Quang Nam and Quang Ngai.
- ○ **Solution:**
  - ▪ **Step 1:** The adjacent matrix table shows the communal houses and shows the path between points. With the number 1 showing that it is possible to move between 2 points, the number 0 represents 1 thing that cannot be moved from one point to the other.

	Da Nang (Hai Chau)	Son Tra	Lien Chieu	Hoi An	Quang Nam	Quang Ngai
Da Nang (Hai Chau)	0	1	1	1	0	0
Son Tra	1	0	0	1	1	0
Lien Chieu	1	0	0	1	1	0
Hoi An	1	1	1	0	1	0
Quang Nam	0	1	1	1	0	1
Quang Ngai	0	0	0	0	1	0

Table 5 Adjacent matrix of Dijkstra's algorithm

- **Step 2:** The weight matrix table, showing the intervals between vertices with each other, with values of 0 means that two points are not connected to each other (cannot move back and forth).

	Da Nang (Hai Chau)	**Son Tra**	**Lien Chieu**	**Hoi An**	**Quang Nam**	**Quang Ngai**
Da Nang (Hai Chau)	0	4	6	35	0	0
Son Tra	3	0	0	28	78	0
Lien Chieu	6	0	0	40	80	0
Hoi An	35	28	40	0	45	0
Quang Nam	0	78	80	45	0	120
Quang Ngai	0	0	0	0	120	0

Table 6 Weight matrix Dijkstra's table

- **Step 3:** The first is from Da Nang (Hai Chau).

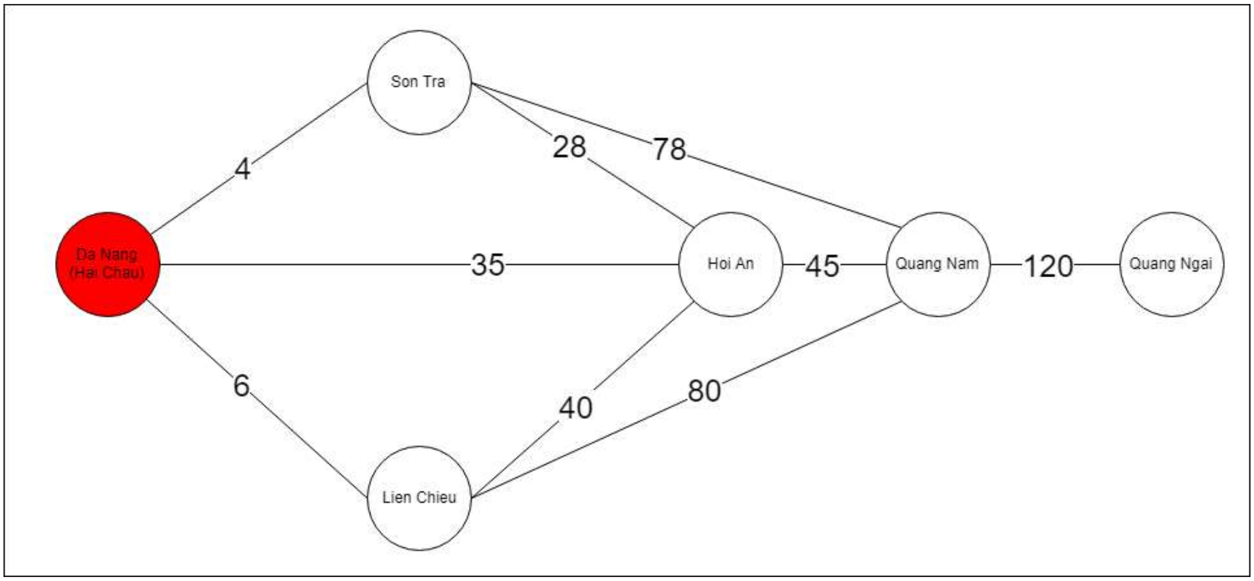


Figure 20 Resolving example Dijkstra's 1

- **Step 4:** Then find all vertices that it can find. As you can see in the picture, Da Nang (Hai Chau) can move to 3 points: Son Tra, Lien Chieu and Hoi An.

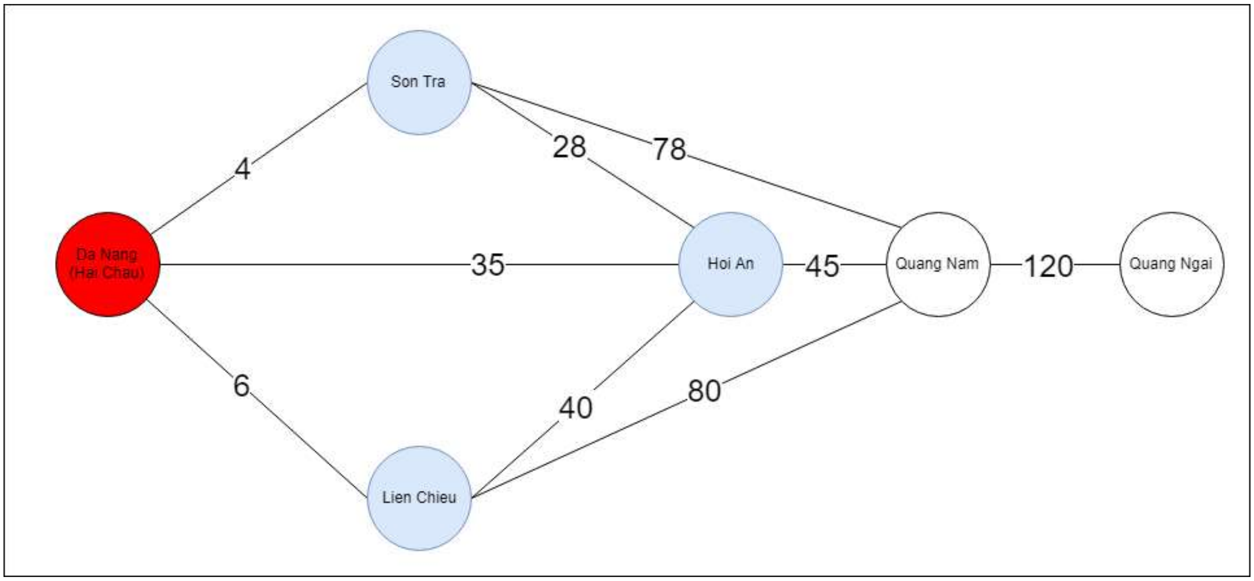


Figure 21 Resolving example Dijkstra's 2

- **Step 5:** After comparing the distance traveled to 3 points of Son Tra, Lien Chieu and Hoi An. The smallest Min () value is the Da Nang (Hai Chau) - Son Tra square. So the next stand is Son Tra.

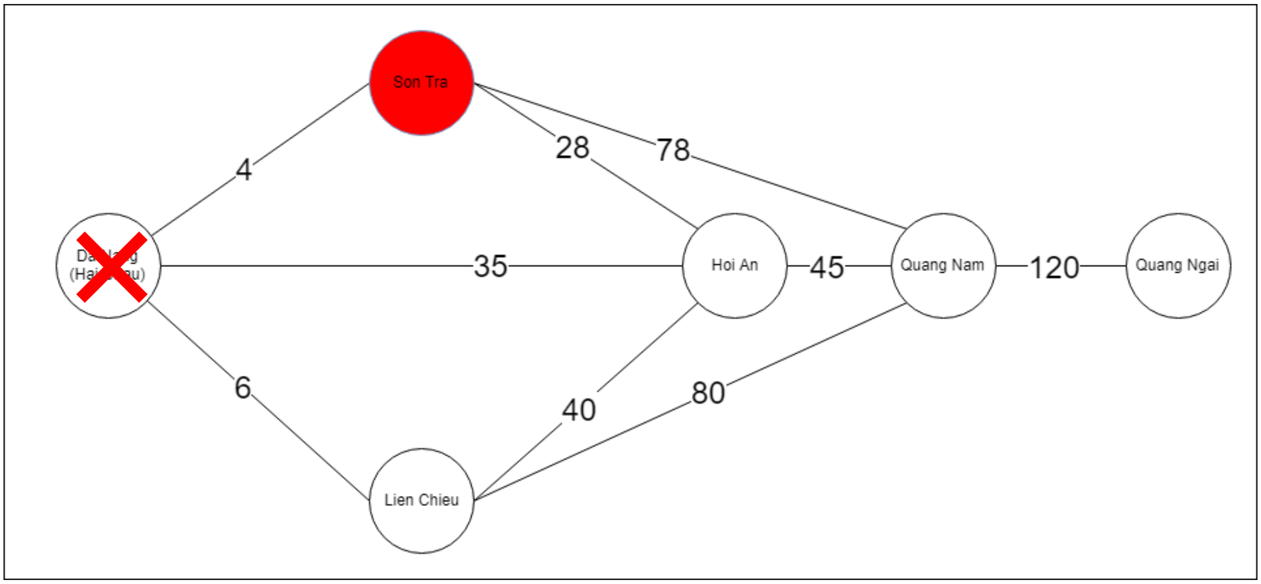


Figure 22 Resolving example Dijkstra's 3

- **Step 6:** After selecting vertical vertices, the algorithm will continue to search for the roads to other vertices (Except the points that have passed or missed in the previous step). As shown in the figure, the algorithm is selecting 2 streets Son Tra - Hoi An and Son Tra - Quang Nam.

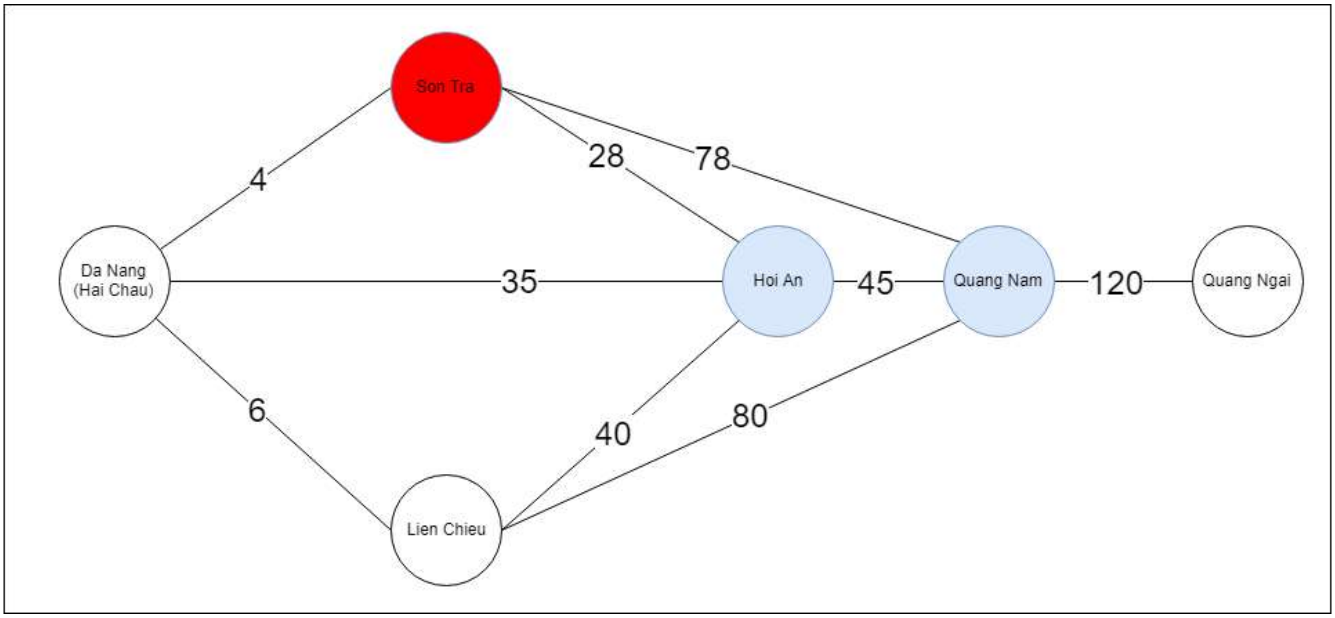


Figure 23 Resolving example Dijkstra's 4

- **Step 7:** After choosing the square of Son Tra - Hoi An, the algorithm chose the standing point as Hoi An. Keep searching for the shortest distance traveled, the algorithm will find the distance from Da Nang - Hai Chau.

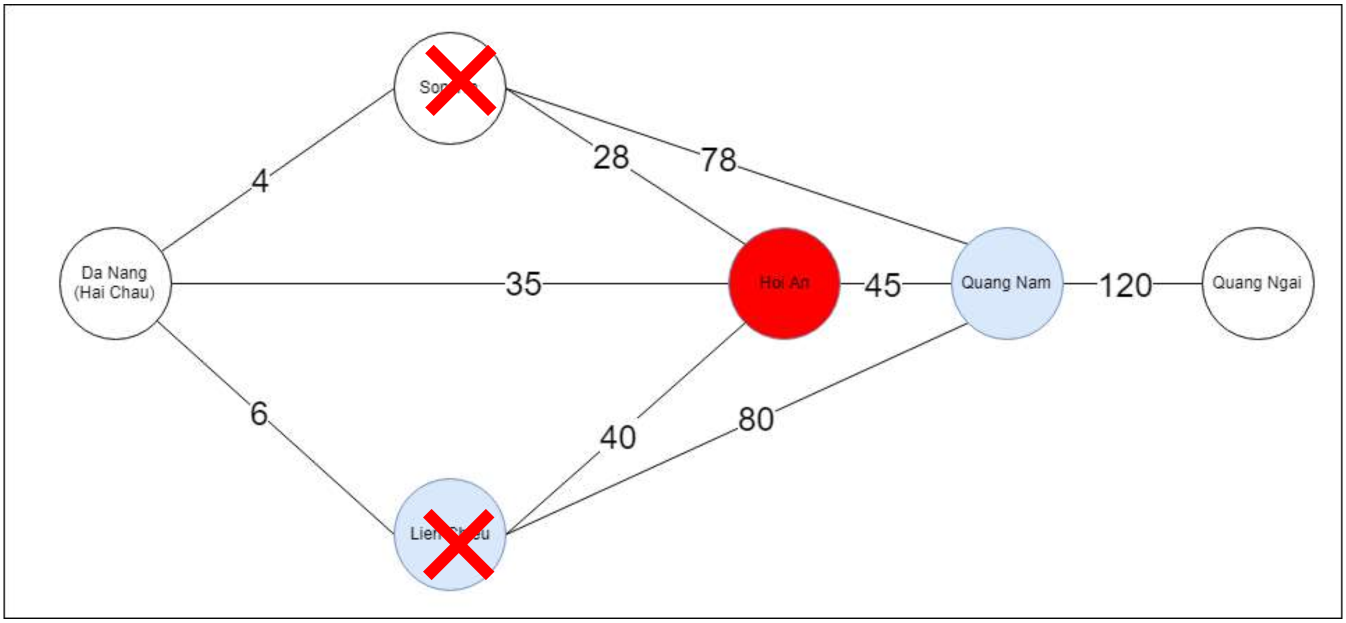
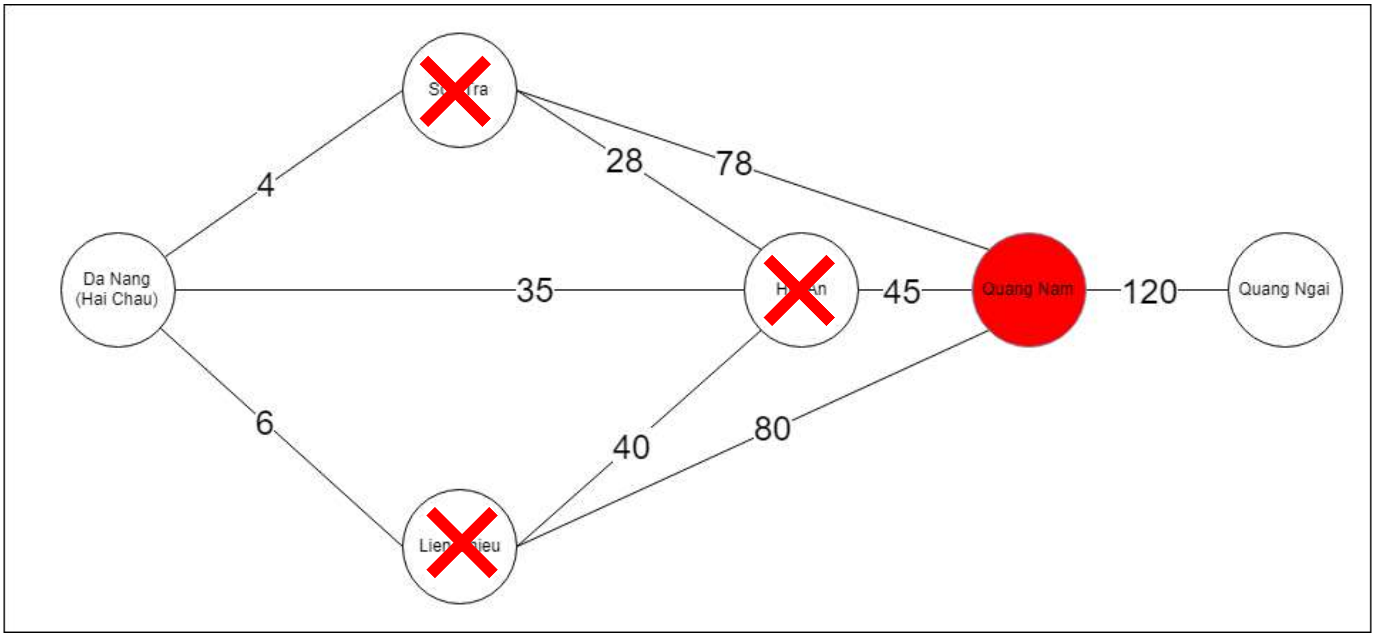
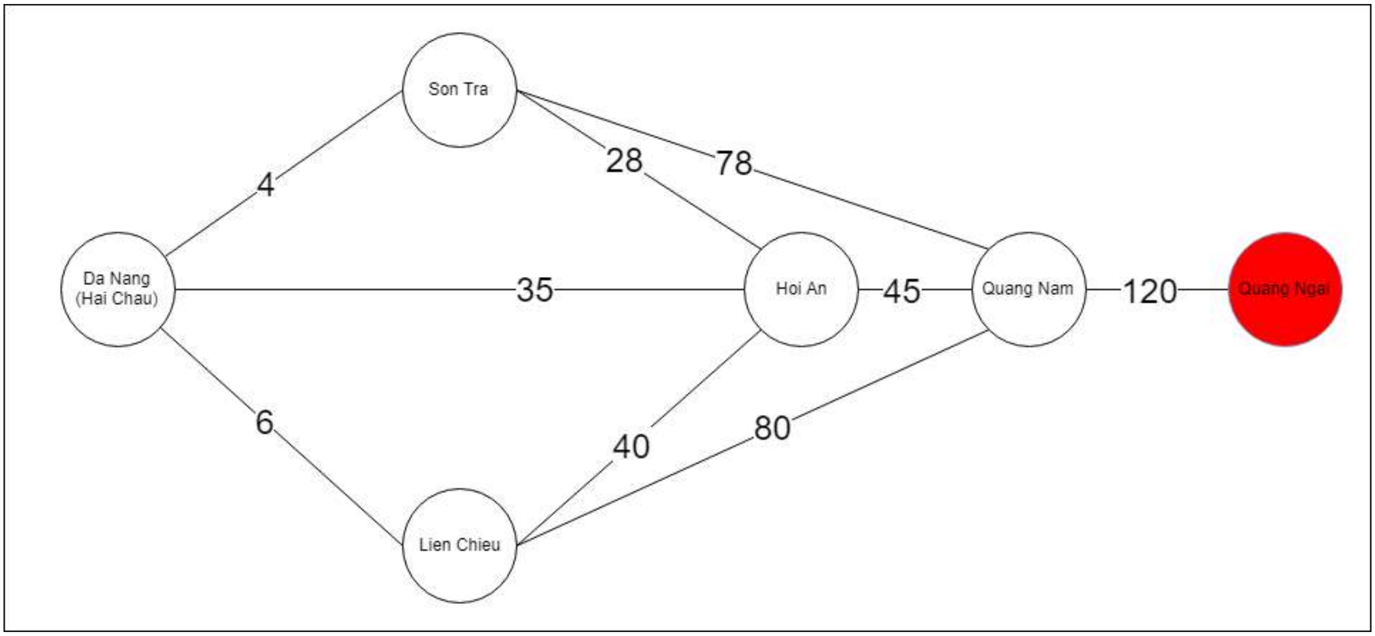


Figure 24 Resolving example Dijkstra's 5


S ra
4
28
78
Da Nang
(Hai Chau)
35
H An
45
Quang Nam
120
Quang Ngai
6
40
80
Lien hieu


Son Tra
4
28
78
Da Nang
(Hai Chau)
35
Hoi An
45
Quang Nam
120
Quang Ngai
6
40
80
Lien Chieu

- **Step 8:** After the algorithm selects the vertical point that coincides with the end of the input. It will arrange and sum up the points that have passed. Then we get the Output with the shortest route is: **Da Nang (Hai Chau) - Son Tra - Hoi An - Quang Nam - Quang Ngai.**

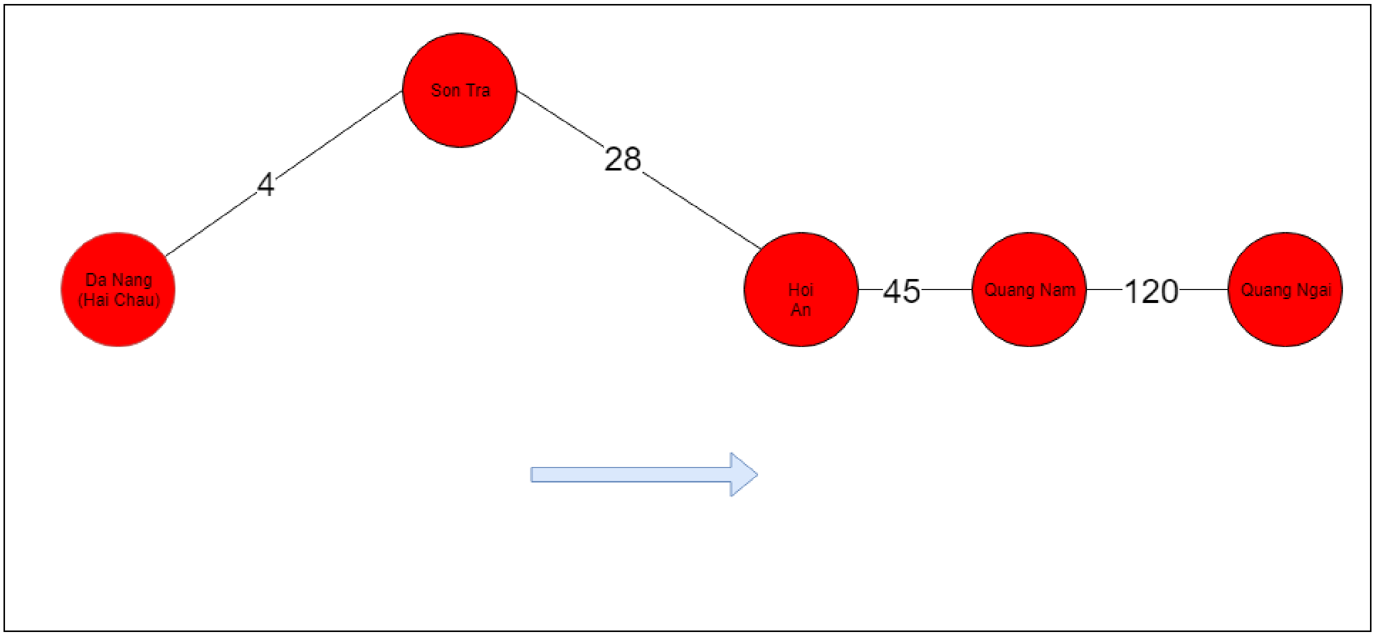


Figure 25 Resolving example Dijkstra's 6

b) Bellman–Ford algorithm:

- **Definition of Bellman–Ford algorithm:** The Bellman-Ford algorithm is also the shortest path calculation algorithm like Dijkstra but Bellman-Ford handles negative weight-weighted problems. The Bellman Ford algorithm operates on the formula O (V · E), where V is the number of vertices and E is the number of arcs of the graph. The advantage of Bellman-Ford is that when the algorithm runs it can start from one vertex to infer the shortest path from that vertex to the other vertices without having to start over. (Wikipedia, n.d.)
- **Operation of Bellman–Ford:**
  - **Step 1:** Each node on the graph computes the distance between it and all with the other nodes and stores it as a table.
  - **Step 2:** After the node is stored into a table, it sends the information table that it has just created to neighboring nodes in the graph.
  - **Step 3:** After the other nodes receive the information board, it will calculate the shortest route to the other nodes and update the information board until the algorithm finds the destination.
- **Example and analyze:** (Kundu, n.d.)

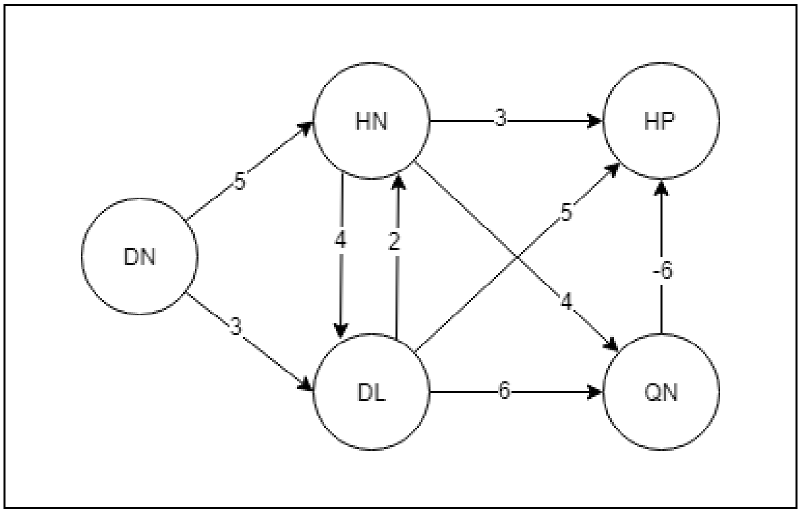


Figure 26 Example for Bellman-ford algorithm

- **Problem:** Find the shortest path from DN to HP.
- **Solution:**
  - Adjacent matrix table:

	DN	HN	DL	HP	QN
DN	**0**	**1**	**1**	**0**	**0**
HN	**0**	**0**	**1**	**1**	**1**
DL	**0**	**1**	**0**	**1**	**1**
HP	**0**	**0**	**0**	**0**	**0**
QN	**0**	**0**	**0**	**1**	**0**

Table 7 Bellman-Ford's Adjacent marix table

- Weight matrix table:

	DN	HN	DL	HP	QN
DN	**0**	**5**	**3**	**0**	**0**
HN	**0**	**0**	**4**	**3**	**4**
DL	**0**	**2**	**0**	**5**	**6**
HP	**0**	**0**	**0**	**0**	**0**
QN	**0**	**0**	**0**	**-6**	**0**

Table 8 Bellman-Ford's Weight matrix table

- **Step 1:** Choose a starting vertex.

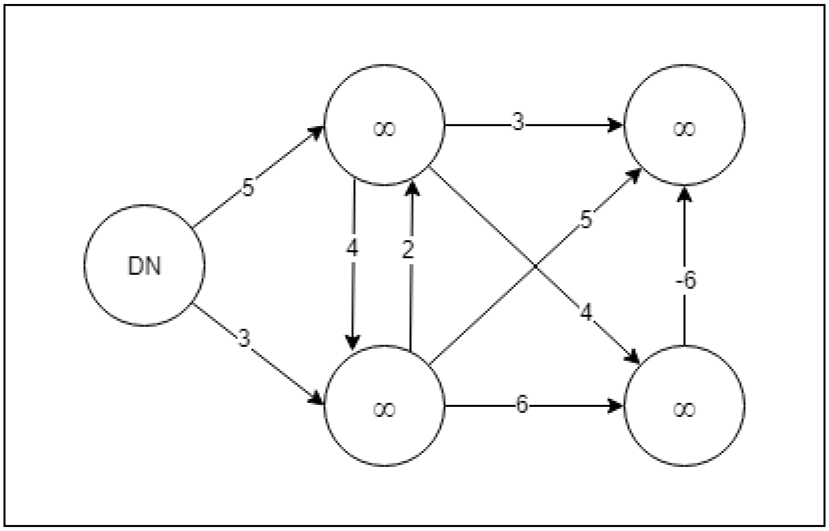


Figure 27 Resolving example of Bellman-Ford algorithm

- **Step 2:** Visit each node and relax the path distances.

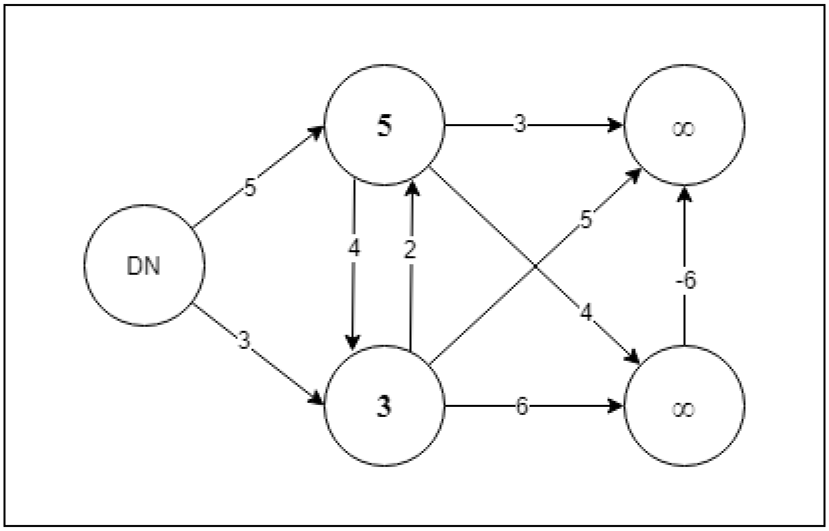


Figure 28 Resolving example of Bellman-Ford algorithm 2

- **Step 3:** Repeat.

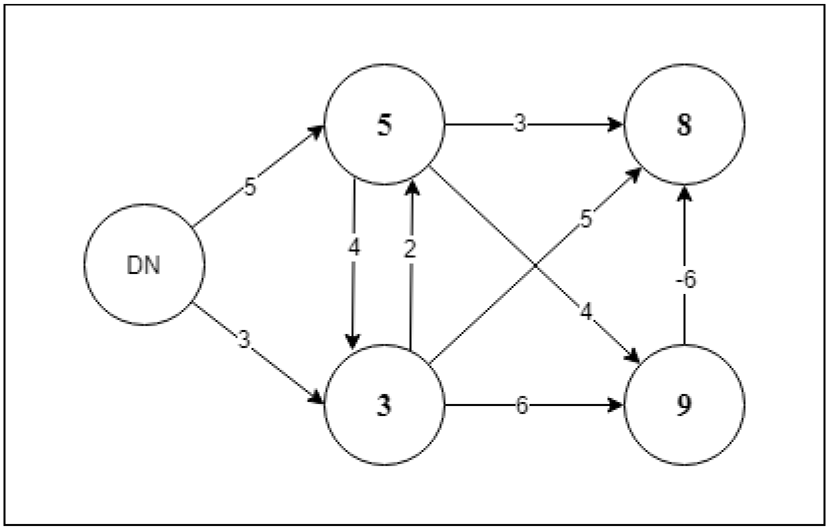


Figure 29 Resolving example of Bellman-Ford algorithm 3

- **Step 4:** End of the algorithm and the last vertex has a distance **value of 3**.

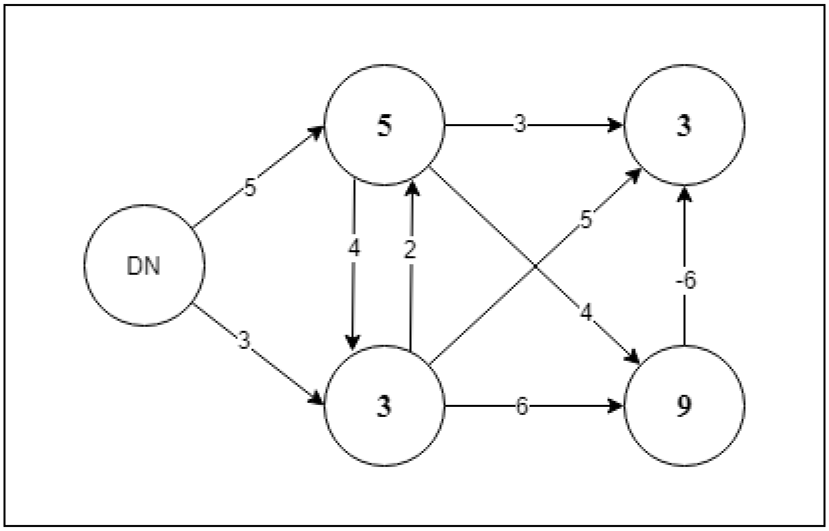


Figure 30 Resolving example of Bellman-Ford algorithm 4

- Information visit table:

DN	HN	DL	HP	QN
0	∞	∞	∞	∞
0	**5**	**3**	∞	∞
0	**5**	**3**	**8**	**9**
0	**5**	**3**	**3**	**9**

Table 9 Resolving example of Bellman-Ford algorithm

III. 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). (Wikipedia, n.d.)

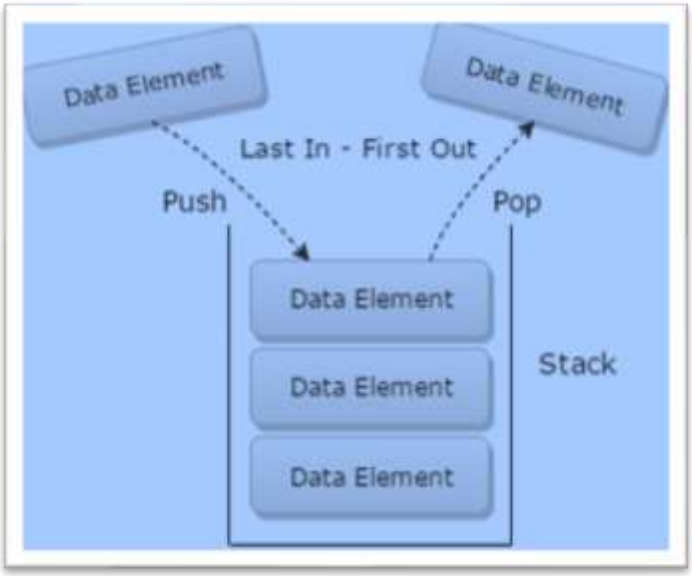


Figure 31 Overview of Stack

  - 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.

Figure 32 Example of Stack

- **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.

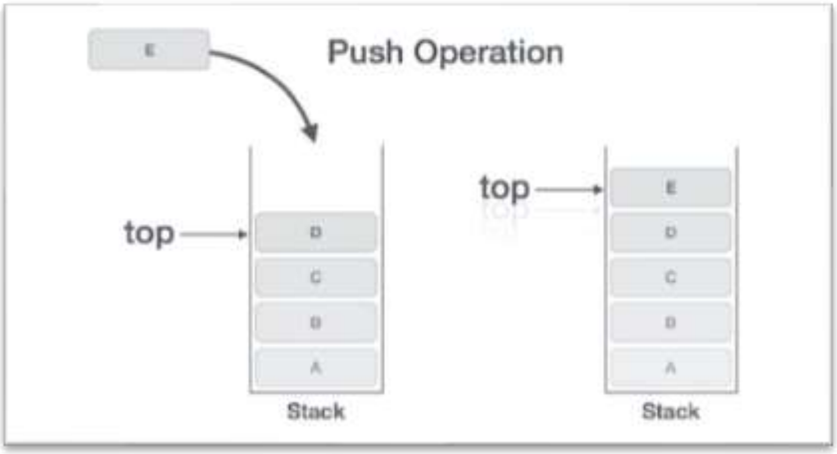


Figure 33 Push Operation of Stack

```
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.

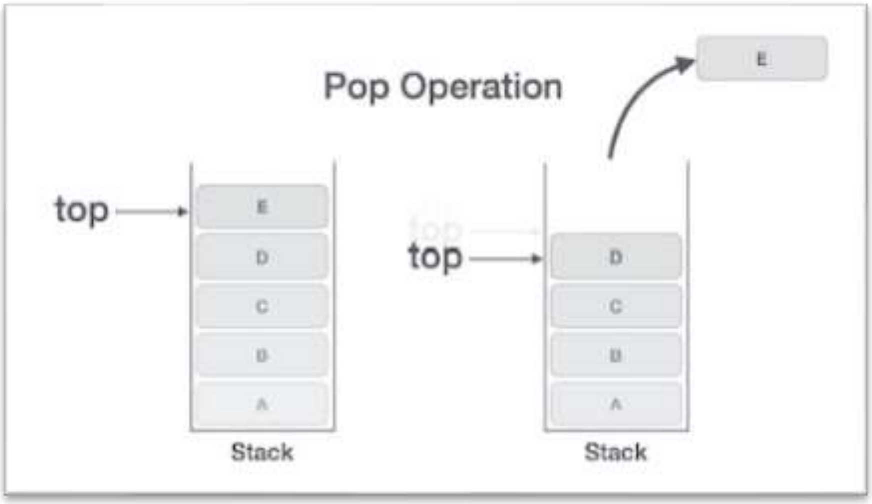


Figure 34 Pop Operation of Stack

```
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.

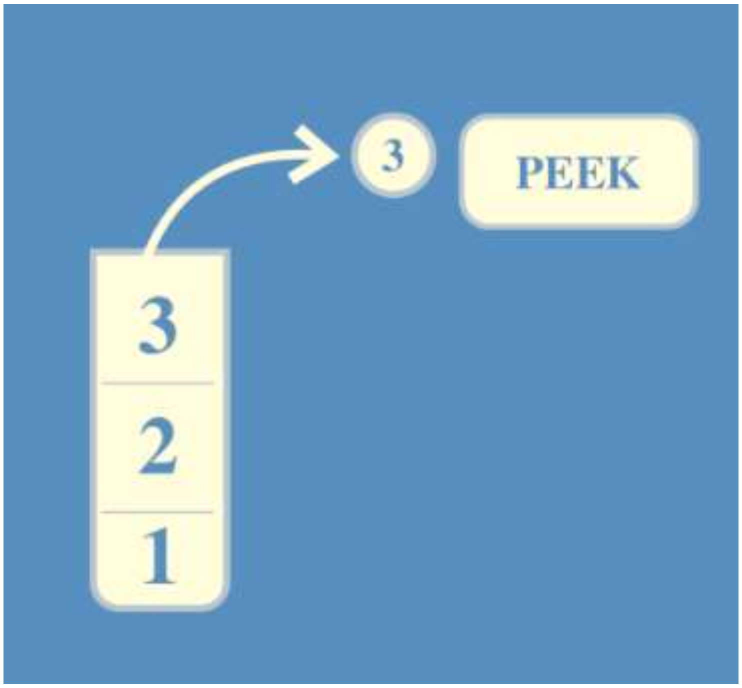


Figure 35 Peek Operation of Stack

```
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

Table 10 VDM table of ADT Stack

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)

Table 11 Operation in Stack

2. M3 Examine the advantages of encapsulation and information hiding when using an ADT.

In computer science, an abstract data type (ADT) is a mathematical model for a data type in which the data is determined by its behavior (semantics) from the point of view of that data user, specifically. Especially about the possible values and possible activities of this data type, not about the behavior of these activities. (Wikipedia, n.d.)

An ADT includes not only activities, but also basic data values and constraints on operations. An "interface" usually refers only to activities and perhaps some constraints to operations, notably pre- and post-condition conditions, not other constraints, for example. as the relationship between activities. (Wikipedia, n.d.)

Abstraction provides a promise that any ADT implementation has certain attributes and capabilities; Knowing this is all it takes to use an ADT object. Users do not need any technical knowledge on how to deploy to use ADT. In this way, the implementation can be complex but will be encapsulated in a simple interface when it is actually used.

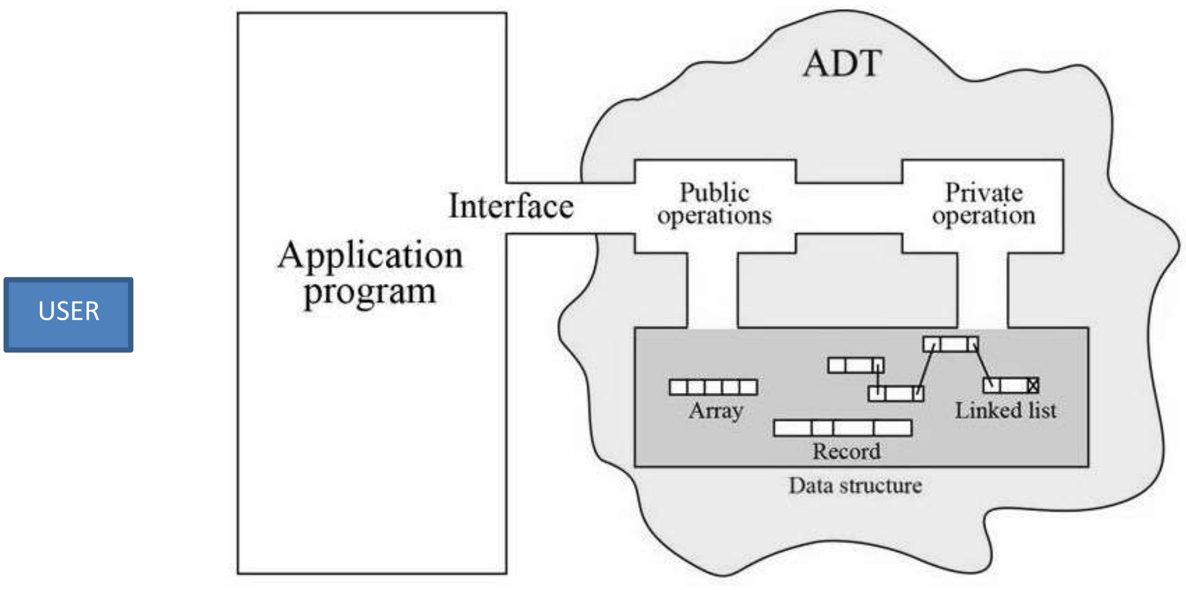


Table 12 User interface and activity of ADT

In fact, ADT can be made up of different types of data and data structures and is written in many different programming languages. When developing an ADT, developers often implement it as modules, so the module's interface will interact and respond to ADT operations. ADT will be encapsulated as a hidden data type in one or more modules, the interface only contains the signature of the operation. The implementation of modules, data structures and processes is hidden by the programmer from most clients, keeping the program unaffected without affecting the user. It is called a hidden data type. If the implementation fails, ATD and the process structure are given, it is called a transparent data type. So below is the advantage of packaging when using ADT:

- It improves maintain of application.
- Making coding process easier beacause it only care about class.
- Improve the ability to read code of Application.
- Increase security of Application. (edu, n.d.)

3. **D2 Discuss the view that imperative ADTs are a basis for object orientation and, with justification, state whether you agree.**

In computer science, an abstract data type is a mathematical model for the data type and defines that data by data user behavior (values, data performance) rather than Galleries of activities. (Cook, 09 June 2005)

Object-oriented programming is a programming model based on "Object Technology", the data is contained by objects often called attributes and the source code will be organized into operating methods. (Cook, 09 June 2005)

The basic difference is that **object-oriented programming** uses **procedural abstraction** to abstract data, while **abstract data types** depend on **abstract types**. So, What is procedural abstraction and adstract types?

> **Procedural abstraction** is the mechanism to separate usage from implementation. It is associated with each specific method performing a specified function. Each method is accompanied by a brief and visual description of what it does. Each unit of behavior is encapsulated in a procedure, also known as procedural abstraction.
>
> An **ADT** consists of data values, operations, and activity constraints. Interfaces usually refer to operations, so there's no way to differentiate data types unless mathematical limits are set. It does not correspond to specific features of the computer language, but is compatible with data operations.

So, my opinion is that **ADT is not the basic ideads behind of OOP** because OOP's characteristics are (encapsulation, inheritance, polymorphism, abstraction), the program is divided into entities called objects, Data structures are designed so that they are specific to objects and functions that operate on the object's data are tied together in the structure data. (jacob, 11-13-2015)

IV. REFERENCE:

Cook, W. R., 09 June 2005. *Object-oriented programming versus abstract data types.* s.l.:s.n.

edu, s., n.d. *Abstraction, Information Hiding and Encapsulation,* s.l.: s.n.

Edux, n.d. *Stack memmory in Computers,* s.l.: s.n.

GeeksforGeeks, n.d. *Abstract Data Types.* s.l.:s.n.

GeeksforGeeks, n.d. *Dijkstra's shortest path algorithm | Greedy Algo-7,* s.l.: s.n.

Geeksforgeeks, n.d. *Memory Layout of C Programs.* s.l.:s.n.

jacob, J., 11-13-2015. *Characteristics of Object Oriented programming language - oops.* s.l.:s.n.

Kundu, A., n.d. *BELLMAN FORD ALGORITHM ,* s.l.: s.n.

Magnum, L., Sep 7, 2018 . *#SideNotes — Stack — Abstract Data Type and Data Structure.* s.l.:s.n.

Stein, L. A., 2003. *Interactive Programming in Java.* s.l.:s.n.

Vietjack, n.d. *Cấu trúc dữ liệu hàng đợi (Queue).* s.l.:s.n.

Wikipedia, n.d. *Abstract data type.* s.l.:s.n.

Wikipedia, n.d. *Bellman-Ford algorithm,* s.l.: s.n.

Wikipedia, n.d. *Bellman-Ford algorithm,* s.l.: s.n.

Wikipedia, n.d. *Dijkstra's algorithm,* s.l.: s.n.

Wikipedia, n.d. *Queue (abstract data type).* s.l.:s.n.

Wikipedia, n.d. *Stack.* s.l.:s.n.

Wikipedia, n.d. *Stack,* s.l.: s.n.