DATA STRUCTURES & ALGORITHMS

Assignment report

Tran Quang Huy

ID: GCD18457 | Class: GCD0605

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	Tran Quang Huy	**Student ID**	GCD18457
Class	GCD0605	**Assessor name**	Ho Van Phi
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

TABLE OF FIGURES

TABLE OF TABLES

INTRODUCTION

If the data is stored in well organized way on storage media and in computer's memory then it can be accessed quickly for processing that further reduces the latency and the user is provided fast response.

Data structure introduction refers to a scheme for organizing data, or in other words a data structure is an arrangement of data in computer's memory in such a way that it could make the data quickly available to the processor for required calculations. A data structure should be seen as a logical concept that must address two fundamental concerns. First, how the data will be stored, and second, what operations will be performed on it? As data structure is a scheme for data organization so the functional definition of a data structure should be independent of its implementation. The functional definition of a data structure is known as ADT (Abstract Data Type) which is independent of implementation. The implementation part is left on developers who decide which technology better suits to their project needs.

1. Data structure explaining the valid operation that can be carried out on the structure.

1.1. Queue data structure

A **queue** is linear list of elements in which deletion of an element can take place only at one end called the **front** and insertion can take place on the other end which is termed as the **rear**. The term front and rear are frequently used while describing queues in linked list. I will deal with the queue as arrays. (w3schools, n.d.)

Figure 1. Example Queue data structure

Queue is also known as an **abstract data structure**, somewhat similar like Stacks. Unlike stacks, 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, example: the data item stored first will be accessed first. (tutorialspoint, n.d.)
A real-life scenario in the form of example for queue will be the queue of people waiting to accomplish particular task where the first person in the queue is the first person to be served first.

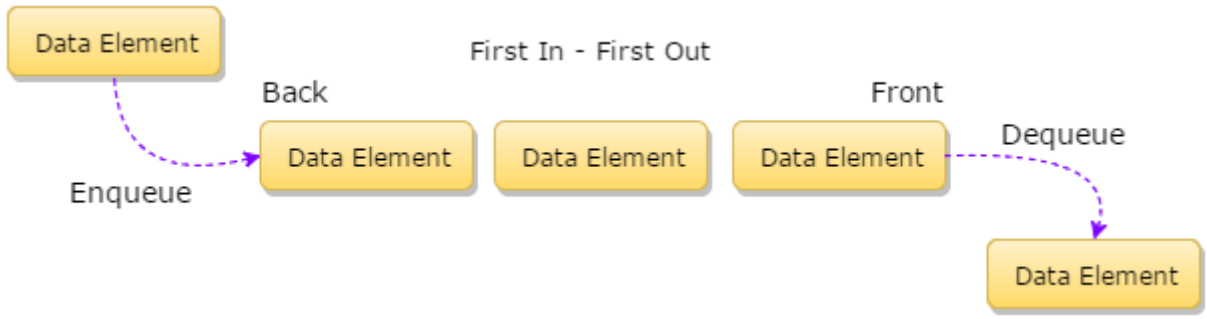


Figure 2. Example element in Queue

Other examples can also be noted within a computer system where the queue of tasks arranged in the list to perform for the line printer, for accessing the disk storage, or even in the time-sharing system for the use of CPU. So basically, queue is used within single program where there are multiple programs kept in the queue or one task may create other tasks which must have to be executed in turn by keeping them in the queue.

Basic Operations:
Queue operations may involve initializing or defining the queue, utilizing it and then completely erasing it from the memory:

- **Enqueue()** : add (stored) an item to the queue.
- **Dequeue()** : remove (access) an item from the queue.

Few more functions are required to make the above-mentioned queue operation efficient:

- **peek()** : Gets the element at the front from the queue without removing it.
- **isfull()** : Checks the queue is full or not.
- **isempty()** : Checks the queue is empty or not.
- **Size():** Show how many elements in queue.

Example for **queue** in **VDM**:

Name: Queue
Symbol: queue
Values: integer
Operators:

Operator	Name	Type
Enqueue()	Enqueue	Int → Array
Dequeue()	Dequeue	Array → Int
Peek()	Peek	Array → Int
IsFull()	IsFull	Array → Bool
IsEmpty()	IsEmpty	Array → Bool

Table 1. Example Queue in VDM

Queue operation:

- **Enqueue():**

Queues maintain two data pointers, **front** and **rear**. Its operations are comparatively difficult to implement than that of stacks.

The following steps should be taken to enqueue (insert) data into Queue:

- **Step 1**: Check if the queue is full.
- **Step 2:** If the queue is full, produce overflow error and exit.
- **Step 3:** If the queue is not full, increment rear pointer to point the next empty space.
- **Step 4**: Add data element to the queue location, where the rear is pointing.
- **Step 5:** return success.

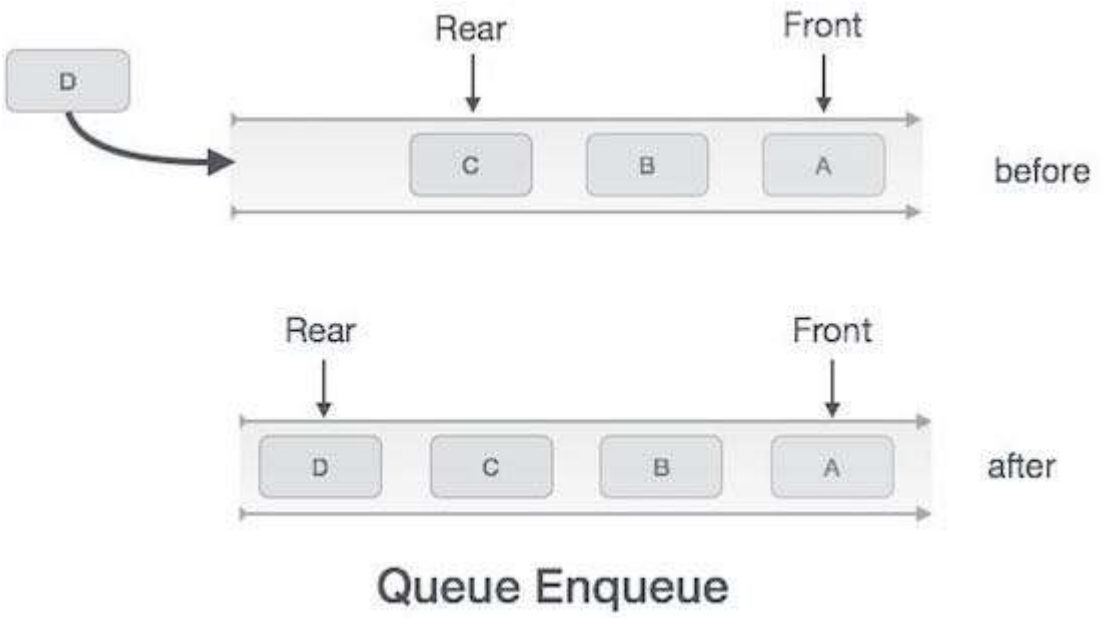


Figure 3. Enqueue operation in Queue

Implementation of **enqueue()** in **C#** programming language:

```
public void enQueue(char x)
        {

            Q[r] = x;
            if (Enum() != max)
            {
                r=(r+1)%max;
            }
        }
```

- **Dequeue():**

Accessing data from the queue is a process of two tasks: access the data where **front** is pointing and remove the data after access.
The following steps below are taken to perform **dequeue()** operation:

- **Step 1**: Check if the queue is empty.
- **Step 2:** If the queue is empty, produce underflow error and exit.
- **Step 3:** If the queue is not empty, access the data where front is pointing.
- **Step 4:** Increment front pointer to point to the next available data element.
- **Step 5**: Return success.

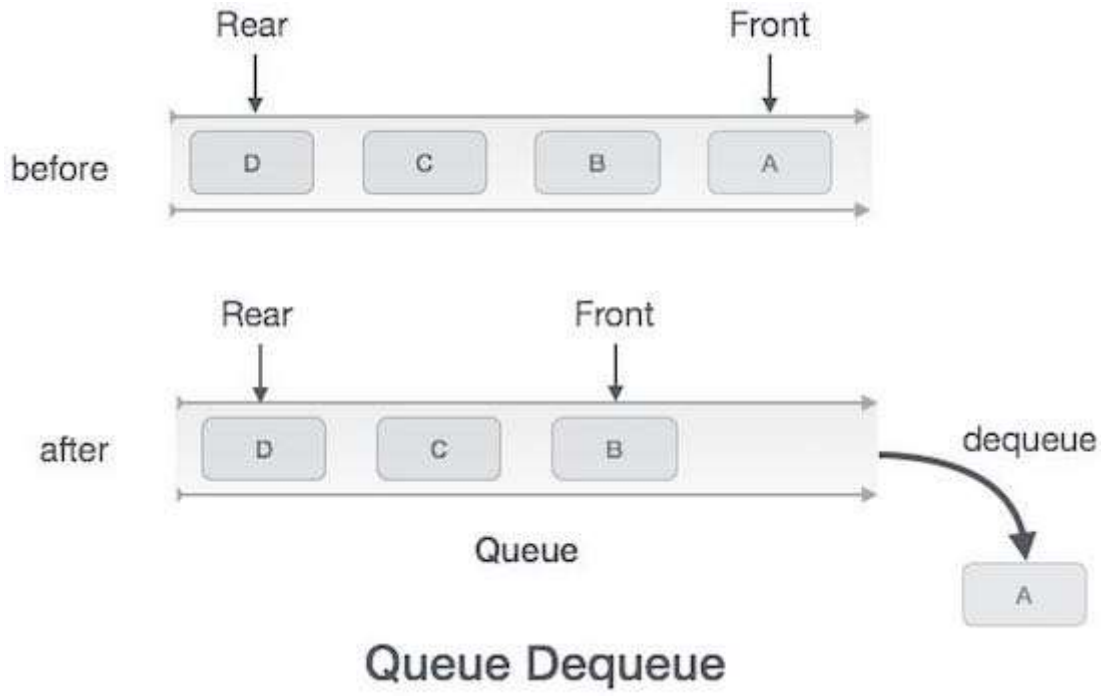


Figure 4. Dequeue operation in Queue

Implementation of **dequeue()** in **C#** programming language:

```
public char deQueue()
        {
            char dQ = Q[f];
            f = (f+1)%max;
            return dQ;
        }
```

- **Peek():**

This function to see the data at the **front** of the queue without delete data.

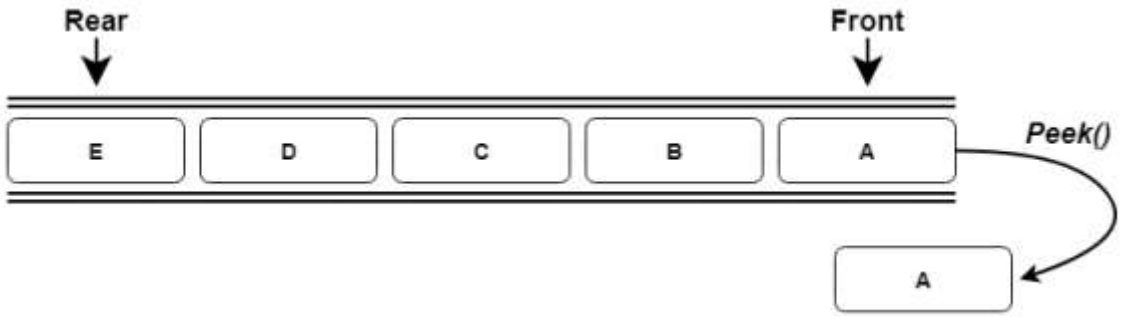


Figure 5. Peek operation in Queue

Implementation of **Peek()** in **C#** programming language:

```
public int Peek()
        {
            if (isEmpty())
                Console.WriteLine("Queue is Empty");
            else
            return Q[f];
        }
```

- **IsFull():**

As we are using single dimension array to implement queue, we just check for the **rear** pointer to reach at max-size to determine that the queue is full or not. In case we maintain the queue in circular linked-list, the algorithm will differ.

Implementation of **isfull()** function in **C#** programming language:

```
public bool isFull()
        {
            Console.WriteLine("Queue is Full");
            return Enum() == max;
        }
```

- **IsEmpty():**

The operation that check for the front pointer if it is less than 0 or larger than **rear** pointer, it tells that the queue is not yet initialized or empty.

Implementation of **isEmpty()** function in **C#** programming language:

```
public bool isEmpty()
      {
          Console.WriteLine("Queue is Empty");
          return f == r;
      }
```

1.2. Stack data structure

A stack is an **Abstract Data Type** (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example: a deck of cards or a pile of plates. (tutorialspoint, n.d.)

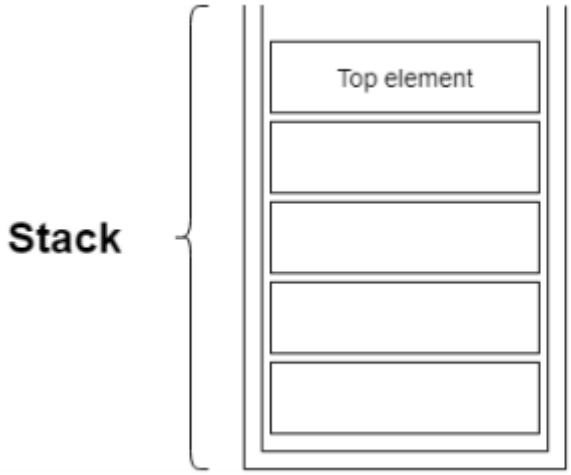


Figure 6. Stack data structure

Stack is linear data structure which follows a particular order in which the operations are performed. The order may be **LIFO** (Last In First Out) or **FILO** (First In Last Out). (geeksforgeeks, www.geeksforgeeks.org, n.d.)

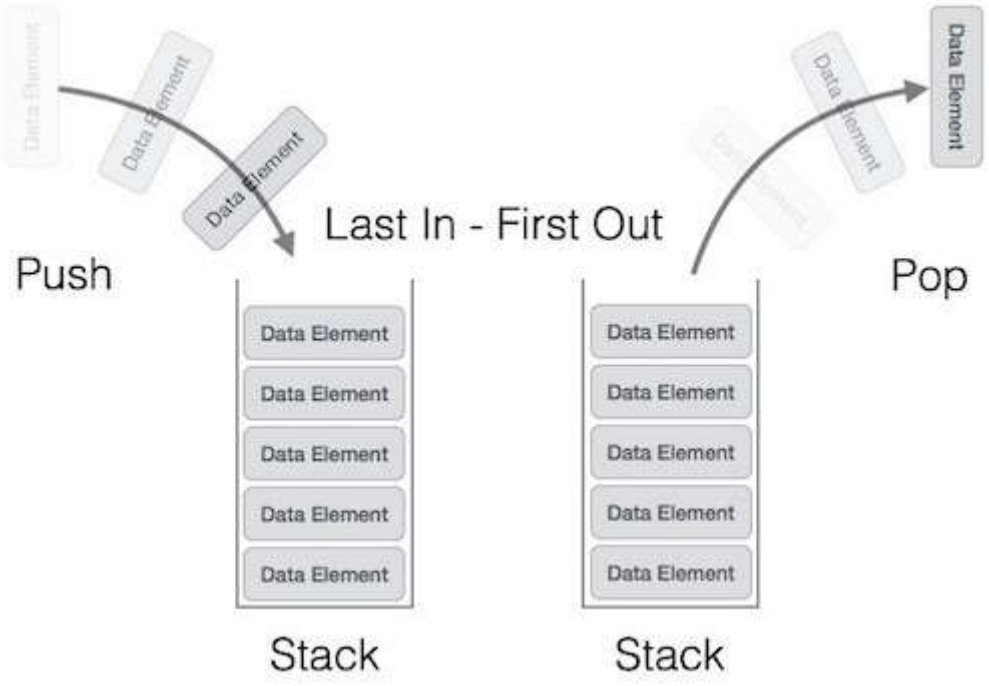


Figure 7. Example for element in Stack

Basic Operations:
Stack operations will involve initializing the stack, using it and then DE-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations:

- **push()** : Pushing (storing) an element on the stack.
- **pop()** : Removing (accessing) an element from the stack.

To use stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks:

- **peek() :** get the top data element of the stack, without removing it.
- **isFull() :** check if stack is full.
- **isEmpty() :** check if stack is empty.

Example for **stack** in **VDM**:

Name: Stack
Symbol: stack
Values: integer
Operators:

Operator	Name	Type
push()	push	Int → Array
pop()	pop	Array → Int
Peek()	Peek	Array → Int
IsFull()	IsFull	Array → bool
IsEmpty()	IsEmpty	Array → bool

Table 2. Example Stack in VDM

Stack operation:

- **Push():**

The process of putting a new data element into stack is known as Push Operation. Push operation involves a series of steps:

- **Step 1:** Checks if the stack is full.
- **Step 2:** If the stack is full, produces an error and exit.
- **Step 3:** If the stack is not full, increments top to point next empty space.
- **Step 4:** Adds data element to the stack location, where top is pointing.
- **Step 5**: Returns success.

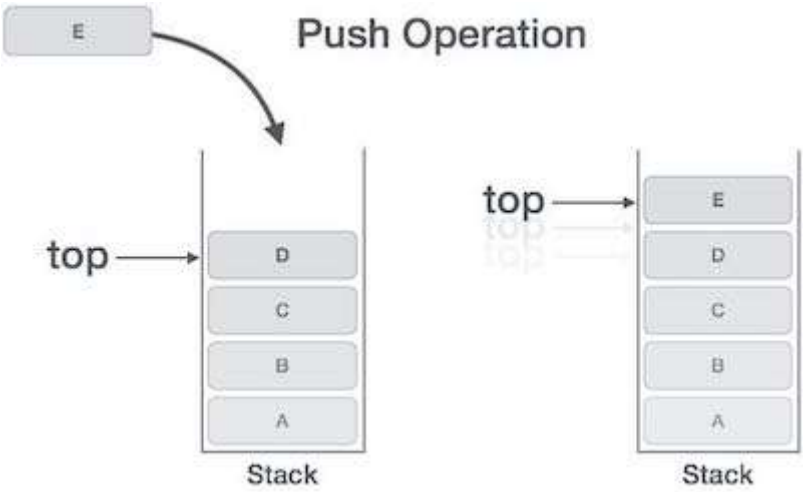


Figure 8. Push operation in Stack

Implementation of this algorithm in **C#**:

```
public static void push(int x)
        {
            if (top == 19)
            {
                Console.WriteLine("Array full");
            }
            else
            {
                top++;
                stackArray[top] = x;
            }
        }
```

- **Pop():**

Accessing the content while removing it from the stack, is known as **Pop** Operation. In an array implementation of **pop()** operation, the data element is not actually removed, instead top is decremented to lower position in the stack to point to the next value. But in linked-list implementation, **pop()** actually removes data element and deal-locates memory space.

A Pop operation may involve the following steps:

- **Step 1**: Checks if the stack is empty.
- **Step 2:** If the stack is empty, produces an error and exit.
- **Step 3:** If the stack is not empty, accesses the data element at which top is pointing.
- **Step 4:** Decreases the value of top by 1.
- **Step 5:** Returns success.

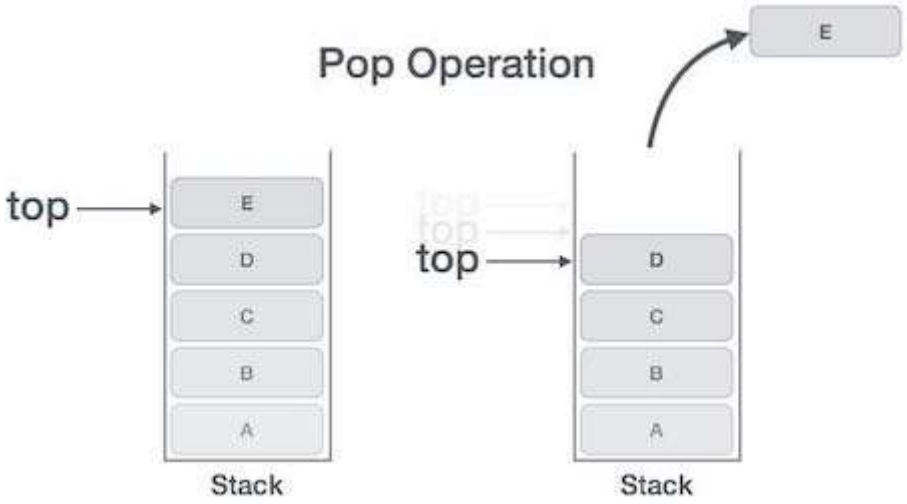


Figure 9. Pop operation in Stack

Implementation of this algorithm in C#:

```
public static int pop()
        {
            int x;
            if (top < 0)
            {
                x=-1;
            }
            else
            {
                x = stackArray[top];
                top--;
            }
            return x;
        }
```

- **Peek():**

The Peek() operation returns the top element value from the stack but not removing it.

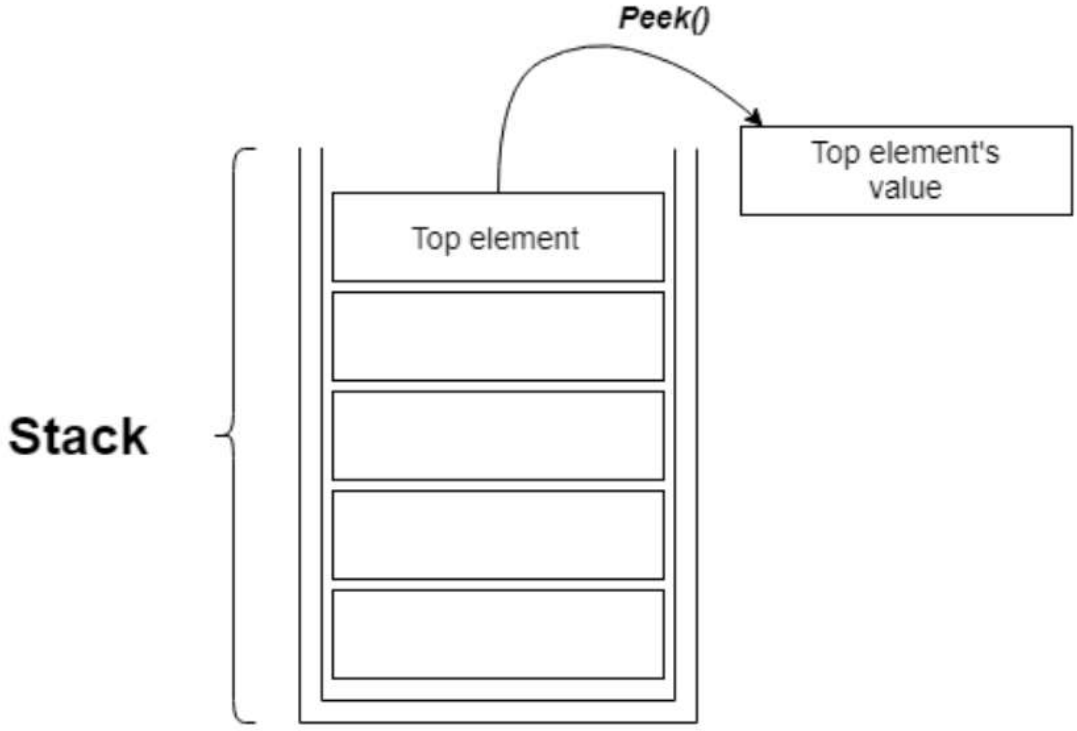


Figure 10. Peek operation in Stack

Implementation of this algorithm in C#:

```
public int Peek()
        {
            if (IsEmpty())
            {
                Console.WriteLine("Stack is empty");
            }
            return stackArray[top];
        }
```

- **isFull():**

IsFull() operation is the operation that check whether a stack is full or not, then returns a Boolean value.

Implementation of this algorithm in C#:

```
public static bool IsFull()
        {
            if (top == Stack.stackArray.Length - 1)
            {
                return true;
            }
            else
            {
                return false;
            }
        }
```

- **IsEmpty():**

Empty() operation is the operation that check whether stack is empty or not. This operation is slightly different from IsFull() operation, where the condition for it to return "True" value is only when the top value at "-1" since the index in array starts from 0.

Implementation of this algorithm in **C#:**

```
public static bool IsEmpty()
        {
            if (top == -1)
            {
                return true;
            }
            else
            {
                return false;
            }
        }
```

2. Operations of memory stack and how it is used to implement function calls in a computer.

2.1. Operation of memory stack

Memory stacks are linear data structures (locations) used to store data in a computer's memory. They may also be referred to as queues. Data within a stack must always of the same type. An example of a stack is illustrated.

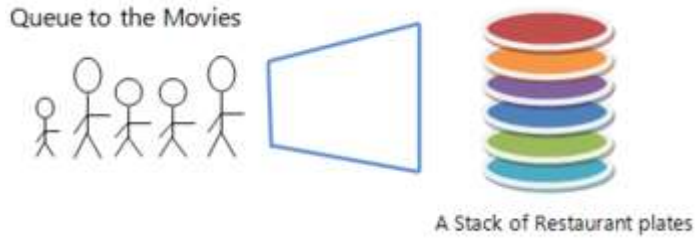


Figure 11. Example for memory stack

Items in a stack are inserted or removed in a linear order and not in any random sequence. As in any queue or collection that is assembled, the data items in a stack are stored and accessed in a specific way. In this case, a technique called LIFO (Last In First Out) is used. This involves a series of insertion and removal operations which we will discuss in the following section.

When a stack is accessed (has items inserted or removed), it is done in an orderly manner using LIFO. LIFO is an acronym which stands for Last In First Out. The computer uses this approach to service data requests received by the computer's memory. LIFO dictates that data inserted or stored last in any given stack, must be the first removed. If that was applied to our queue for the movies in Figure 1 above, there would be chaos! Operations on a stack are discussed in the following sections using Figure below:

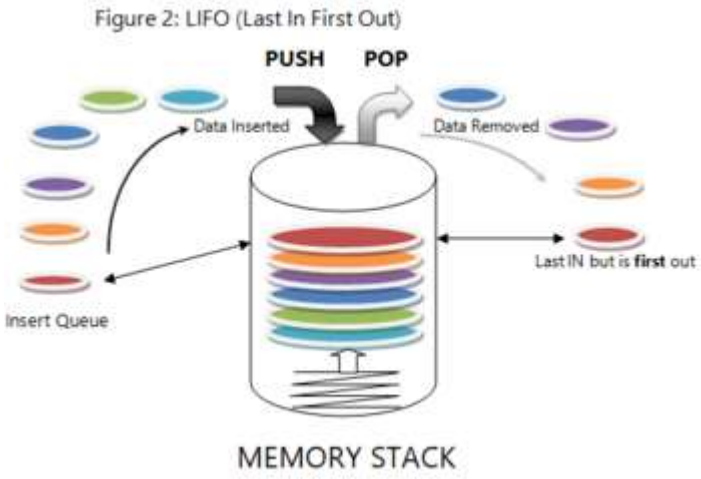


Figure 12. LIFO in memory stack

- **The Push Operation:**

The push operation involves inserting data items into a stack. Let us examine our restaurant plate dispenser. The push operation adds plates (data items) to the plate dispenser (stack). The first plate is pushed to the bottom of the stack with all subsequent plates following in order after it. The first data item inserted is the most inaccessible and positioned at the bottom of the stack.

- **The Pop Operation:**

The pop operation involves removing data items from a loaded stack. In our plate dispenser illustration, the last plate (data item) added is positioned at the top of the stack. This data item is popped out of the stack as the first item to be removed. Think of the spring loading system at the base of the plate dispenser. It pushes the stack of plates upwards each time a plate is removed. In memory, items continue to be popped out in that order.

- **The Peek Operation:**

In this operation, no data item is added to or removed from the stack. The peek operation simply requests the address location of the data item at the top of the stack (the last item to be pushed).

- **The Search Operation:**

In this operation, no data item is added to or removed from the stack either. The search operation requests the address location of any data item in the stack. It determines the item location in reference to the item at the top of the stack.

- **Duplicate operation:**

The duplicate operation pops the top item out of the stack and then pushed again (twice), so that an additional copy of the former top item is now on top, with the original below it.

- **Swap or exchange operation:**

The swap operation pops 2 topmost items out of the stack and then pushed back buy with exchanged place

- **Rotate operation:**

In rotate operation, the n topmost items are moved on the stack in a rotating fashion. For example, if n=3, items 1, 2, and 3 on the stack are moved to positions 2, 3, and 1 on the stack, respectively. Many variants of this operation are possible, with the most common being called left rotate and right rotate.

2.2. How stack used to implement function calls in a computer

Our computer systems function using many programs and applications. Each of these applications must use memory to execute its operations. Many memory operations can be abstract to the user, but stacking is one operation that is easily relocatable.

A call stack is composed of stack frames (also called activation records or activation frames). These are machine dependent and ABI-dependent data structures containing subroutine state information. Each stack frame corresponds to a call to a subroutine which has not yet terminated with a return. (wikipedia, wikipedia, n.d.)

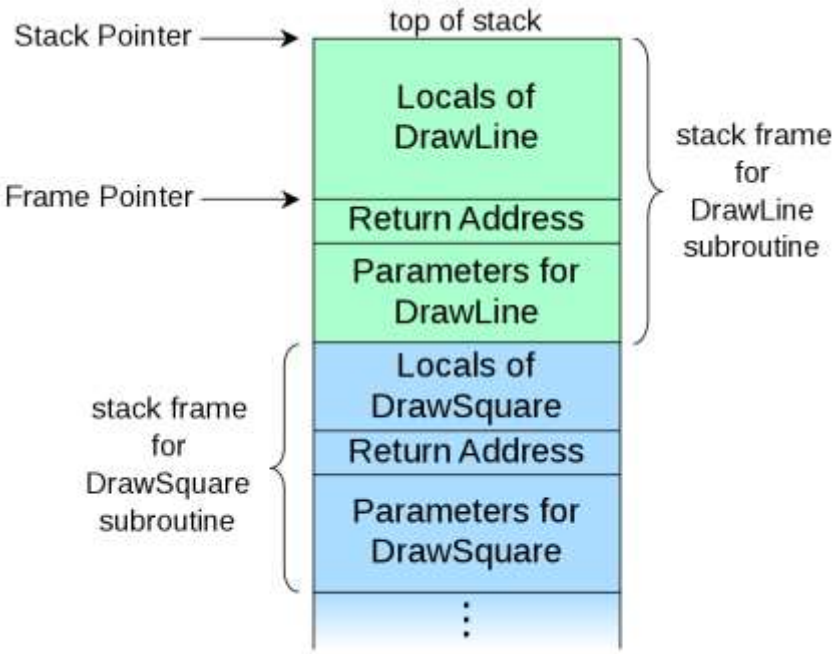


Figure 13. Example DrawLine in Stack

The stack frame at the top of the stack is for the currently executing routine. The stack frame usually includes at least the following items (in push order):

- The arguments (parameter values) passed to the routine (if any).
- The return address back to the routine's caller.
- Space for the local variables of the routine (if any).

When a function is called, a new stack frame is created:

- Arguments are stored on the stack.
- Current frame pointer and return address are recorded.
- Memory for local variables is allocated.
- Stack pointer is adjusted.

When a function returns, the top stack frame is removed:

- Old frame pointer and return address are restored.
- Stack pointer is adjusted.
- The caller can find the return value, if there is one, on top of the stack.

Example: The top values of the stack may be stored in CPU registers, or in the CPU cache, or the return value could be stored in a register instead of on the stack.

3. Data structure for a First In First out(FIFO) queue.

Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, 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.
Example: The data item stored first will be accessed first.

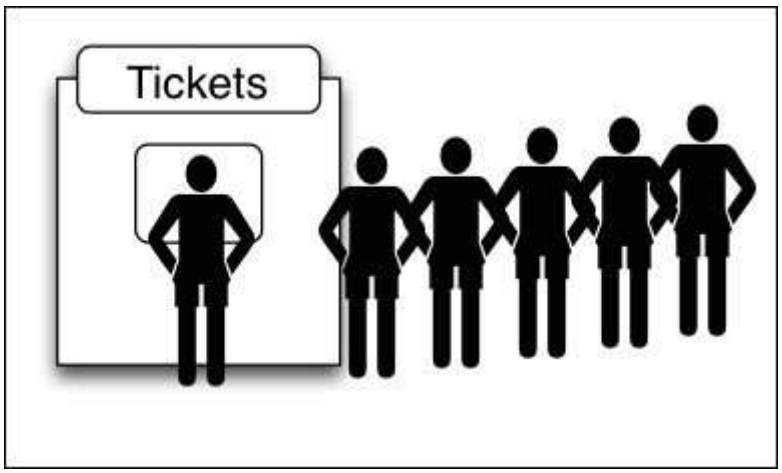


Figure 14. Example people buy tickets in Queue

A queue of people at ticket-window: The person who comes first gets the ticket first. The person who is coming last is getting the tickets in last. Therefore, it follows first-in-first-out (FIFO) strategy of queue.

As in stacks, a queue can also be implemented using arrays, linked-lists, pointers and structures.

Queues are implemented using arrays in a similar way to stacks, however, needs two integers indexes, one giving the position of the front element of the queue in the array, the other giving the position of the back element.

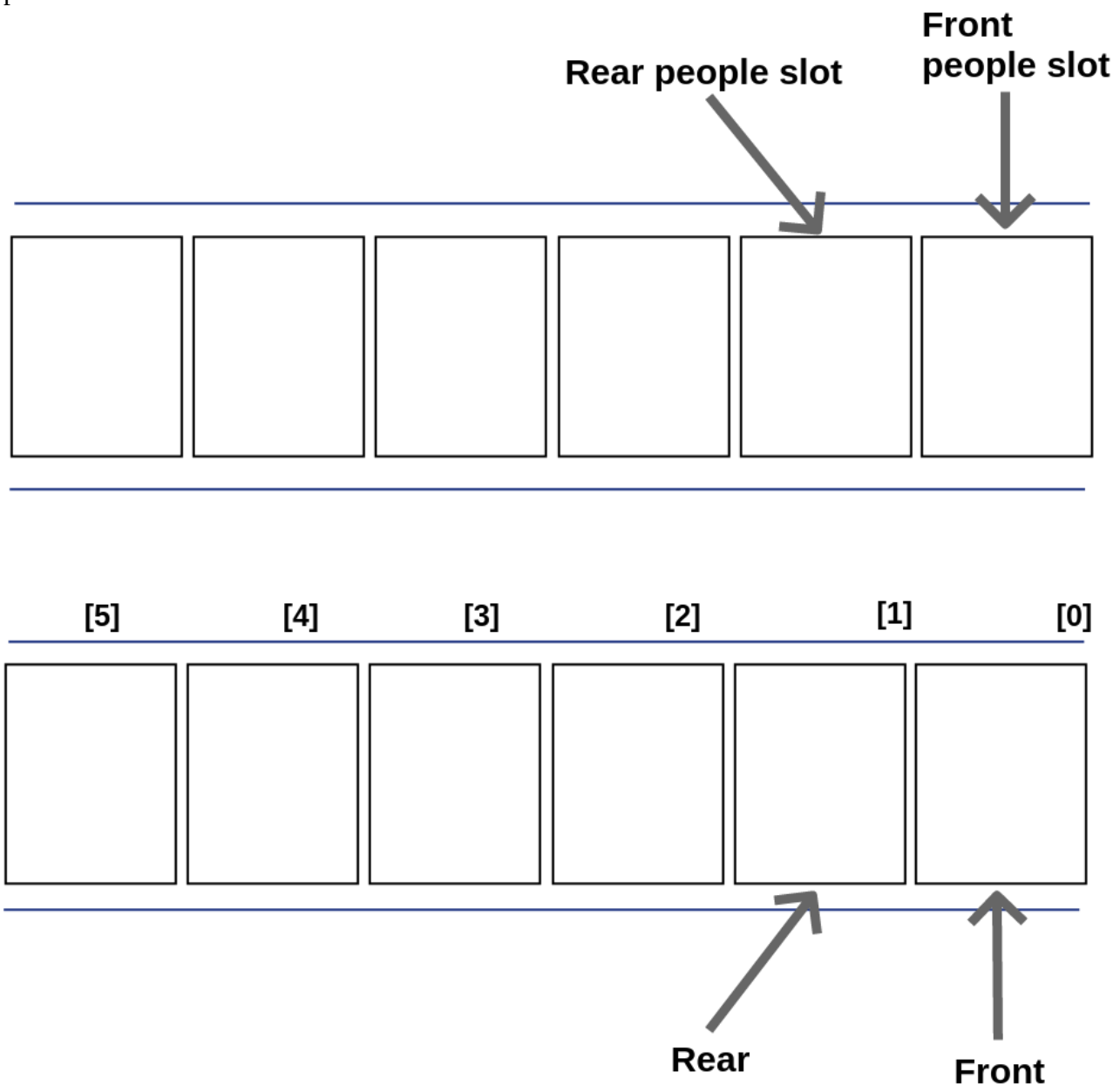


Figure 15. Example 1 people buy ticket in Queue

As more elements (people) add into the queue, the rear (rear people position) keeps on moving ahead, always pointing to the position where the next element (people) will be inserted, while the front (front people position) remains at the first index of the queue.

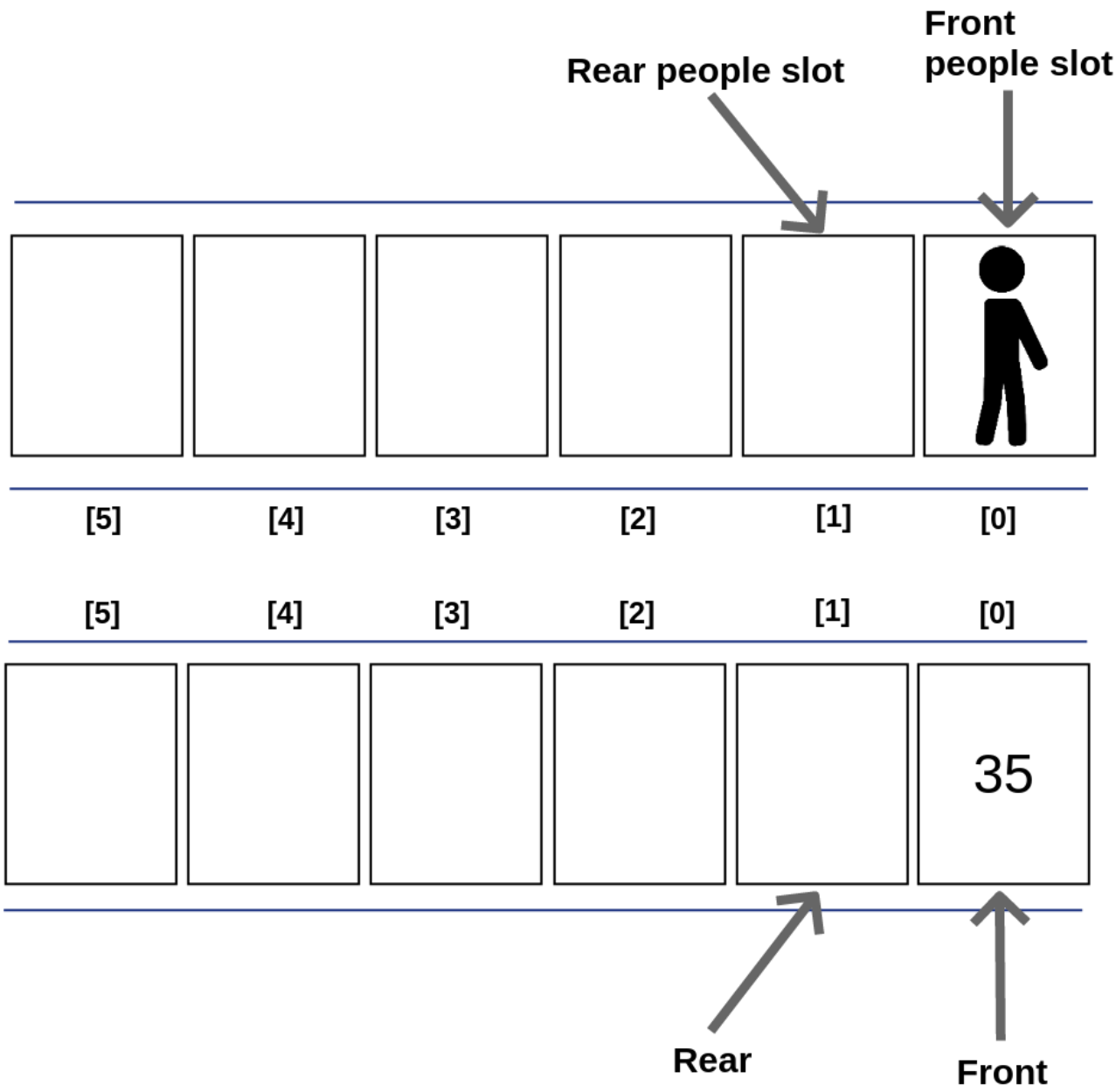


Figure 16. Example 2 people buy ticket in Queue

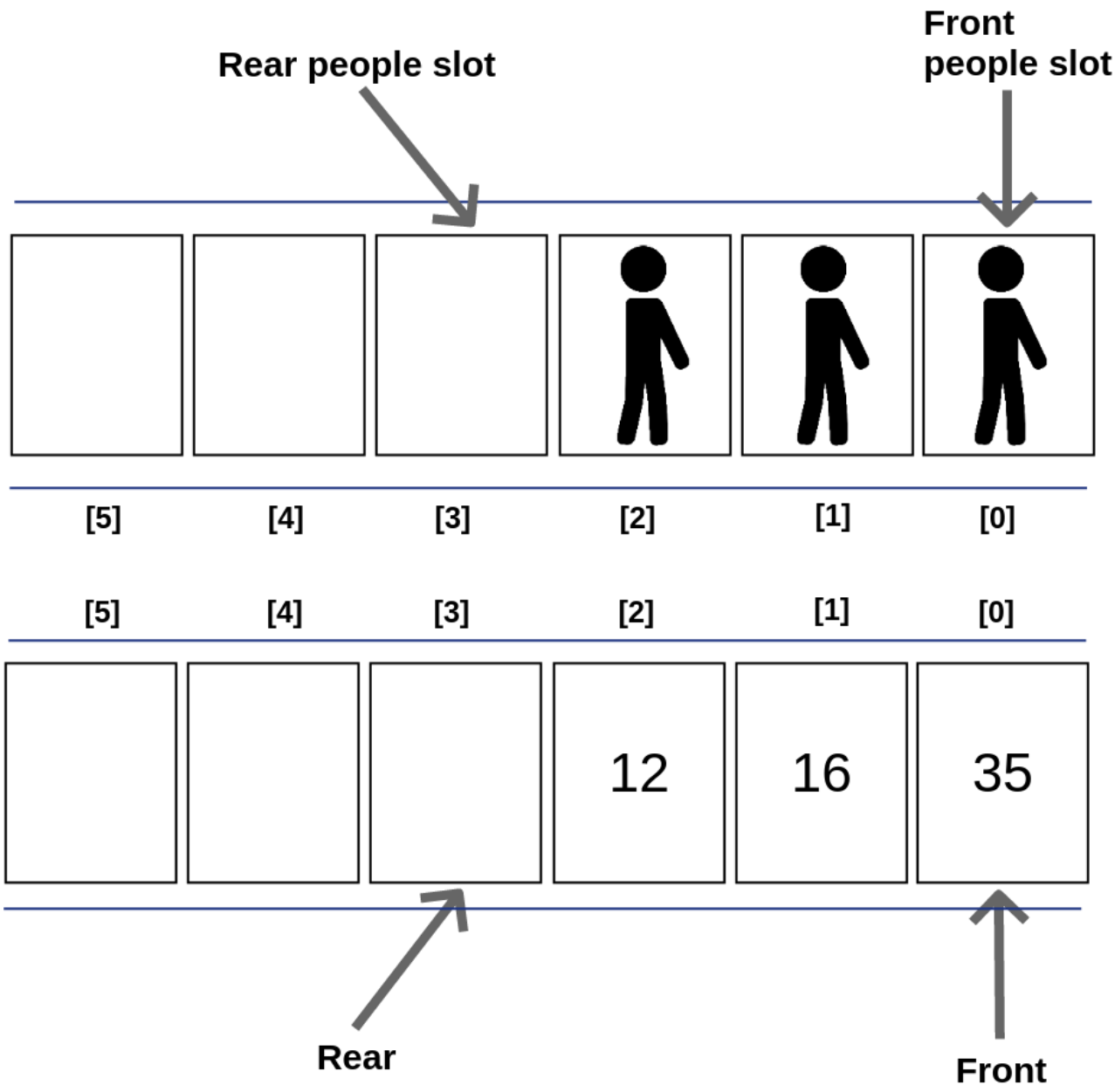


Figure 17. Example 3 people buy ticket in Queue

When an element got removed from the queue, there are two possible approaches:

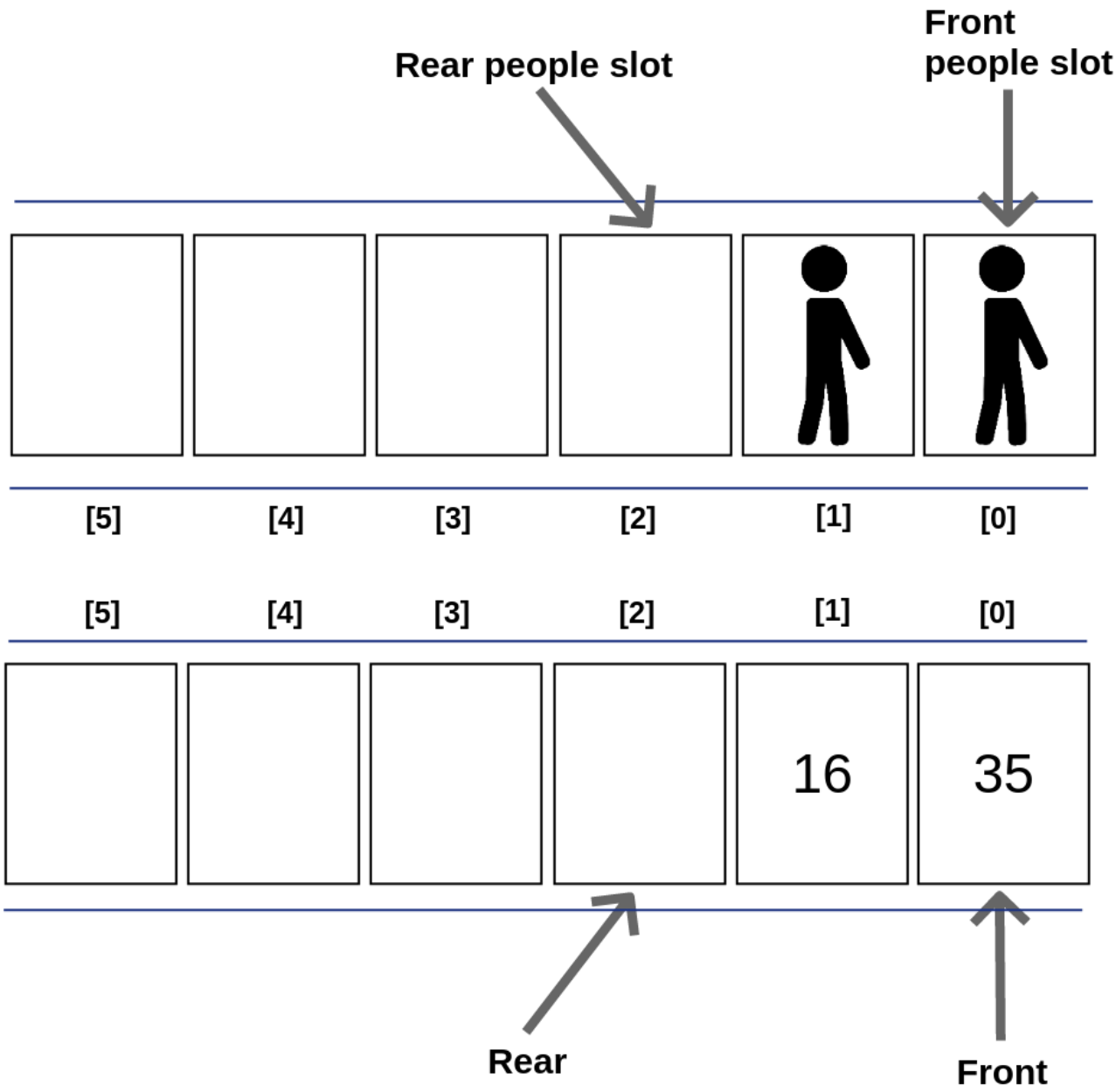


Figure 18. Example 4 people buy ticket in Queue

In the last figures, the element at front position got removed, and then one by one shift all the other elements in forward position. Or to be compared to people queue, first people in the queue go away, then every following up people moved forward to fill up the space the last people left.

In this approach, there is an overhead of shifting the elements one position forward every time the first element got removed.

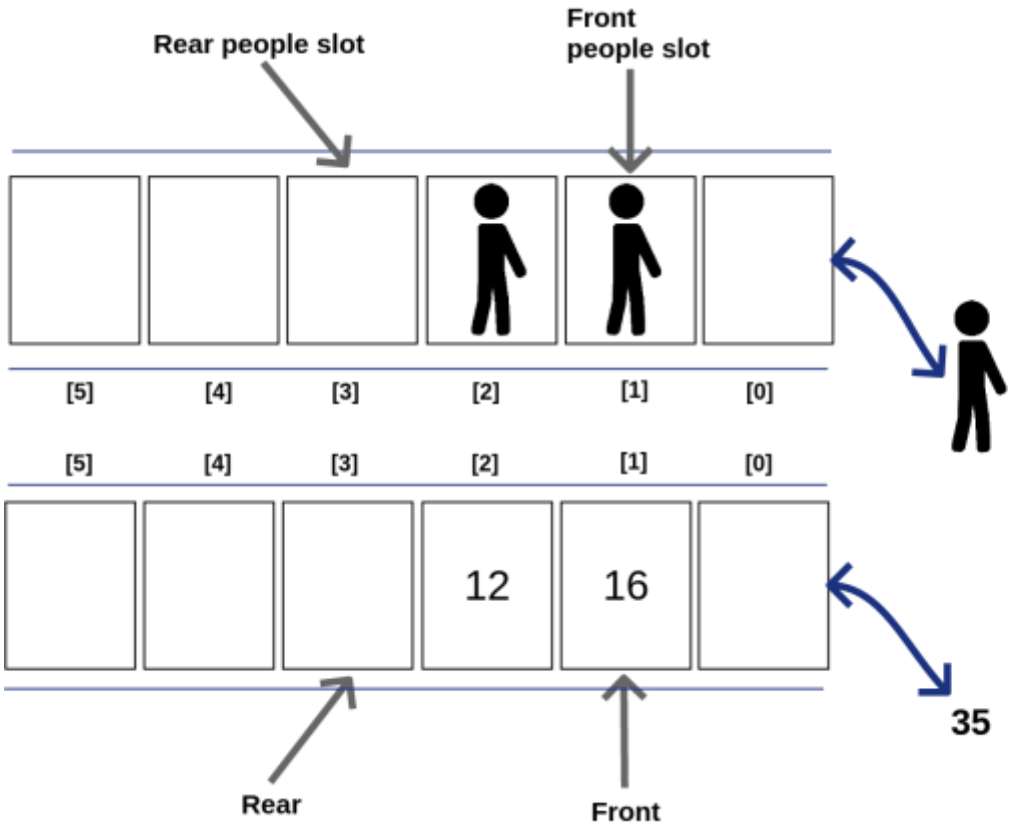


Figure 19. Example 5 people buy ticket in Queue

The element at front position got removed, and then the front move to the next position. Or to be compared to people queue, even if the first people in the queue left away, the following up next people will become the first-in-queue people, but not moving forward to fill up the space the last people left. In this approach, there is no such overhead, but whenever moving the front one position ahead, after removal of first element, the size on queue is reduced by one space each time.

4. Compare the performance of two sorting algorithms.

Sorting is a very classic problem of reordering items (that can be compared, **example:** integers, floating-point numbers, strings) of an array (a list) in a certain order (increasing, non-decreasing, decreasing, non-increasing, lexicographical). (visualgo, n.d.)
There are many different sorting algorithms, each has its own advantages and limitations.

Sorting is commonly used as the introductory problem in various Computer Science classes to showcase a range of algorithmic ideas.

When an (integer) array A is sorted, many problems involving A become easy (or easier):

- Searching for a specific value v in array A,
- Finding the min/max or the k-th smallest/largest value in (static) array A,
- Testing for uniqueness and deleting duplicates in array A,
- Counting how many times a specific value v appears in array A,
- Set intersection/union between array A and another sorted array B,
- Finding a target pair $x \in A$ and $y \in A$ such that x+y equals to a target z.

4.1. Selection sort

The Selection Sort algorithm is a simple algorithm. This sorting algorithm is an in-place comparison algorithm, in which the list is divided into two parts, the sorted list on the left and the unsorted list in the right. Initially, the sorted part is empty and the unsorted part is the whole list.

The smallest element is selected from the unsorted array and is swapped with the left most and that element becomes the element of the sorted array. This process continues until all of the elements in the unsorted array are moved to the sorted array.

This algorithm is not suitable for large data sets when the worst-case complexity and the average case is O (n^2) where n is the number of elements. (geeksforgeeks, geeksforgeeks, n.d.)

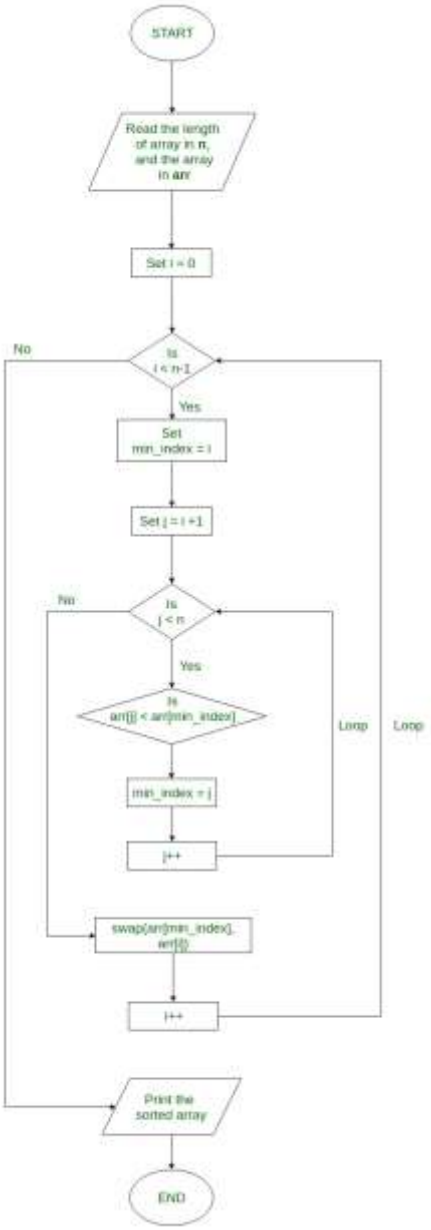


Figure 20. Flow Chart Selection Short

How Selection sort algorithm works:

The algorithm proceeds by finding the smallest (or largest, depends on sorting order) element in the unsorted sub-list, swapping it with the leftmost unsorted element. The process of searching the minimum element and placing it in the proper position is continued until all of the elements are placed at right position.

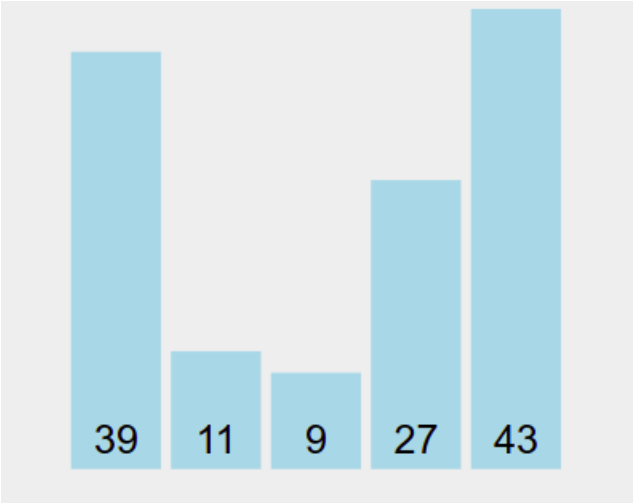
- **Step 1:** Set MIN to location 0
- **Step 2**: Search the minimum element in the list
- **Step 3**: Swap with value at location MIN
- **Step 4**: Increment MIN to point to next element
- **Step 5**: Repeat until list is sorted

The following example will visualize the selection sort algorithm:

Given an array of N items and L = 0, Selection Sort will:

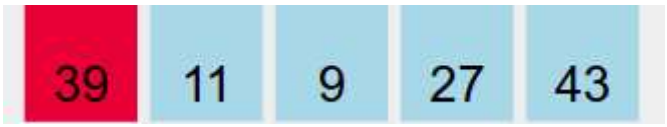
- Find the position X of the smallest item in the range of [L...N−1],
- Swap X-th item with the L-th item,
- Increase the lower-bound L by 1 and repeat Step 1 until L = N-2.

Without further ado, let's try Selection Sort on the same small example array [39, 11, 9, 27, 43].

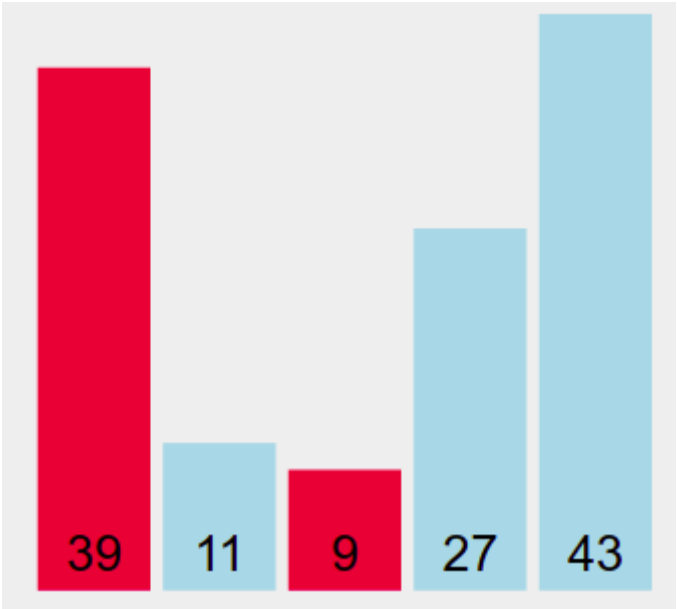


Without loss of generality, we can also implement Selection Sort in reverse: Find the position of the largest item Y and swap it with the last item.

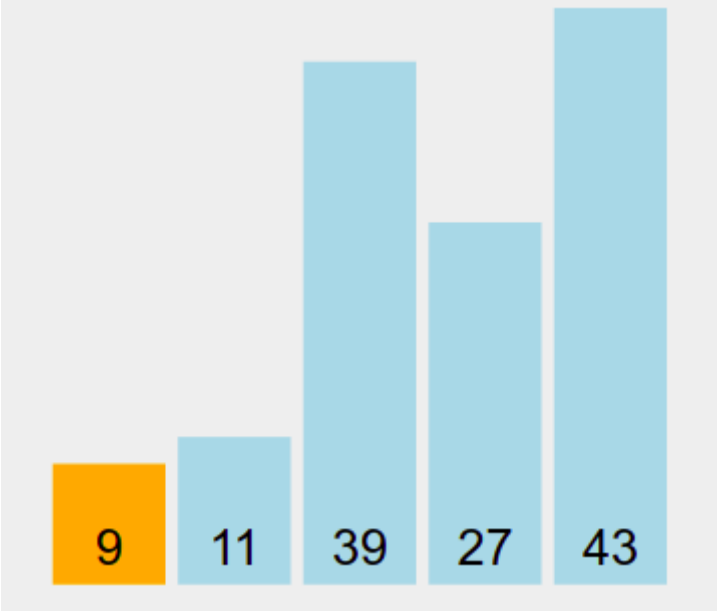
Iteration 1: Set 39 as the current minimum, then iterate through the remaining unsorted elements to find the true minimum.



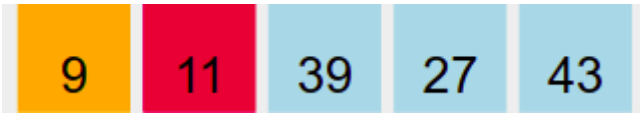
- Check if 11 is smaller than the current minimum (39).
- Set 11 as the new minimum.
- Check if 9 is smaller than the current minimum (11).
- Set 9 as the new minimum.
- Check if 27 is smaller than the current minimum (9).
- Check if 43 is smaller than the current minimum (9).
- Swap the minimum (9) with the first unsorted element (39).



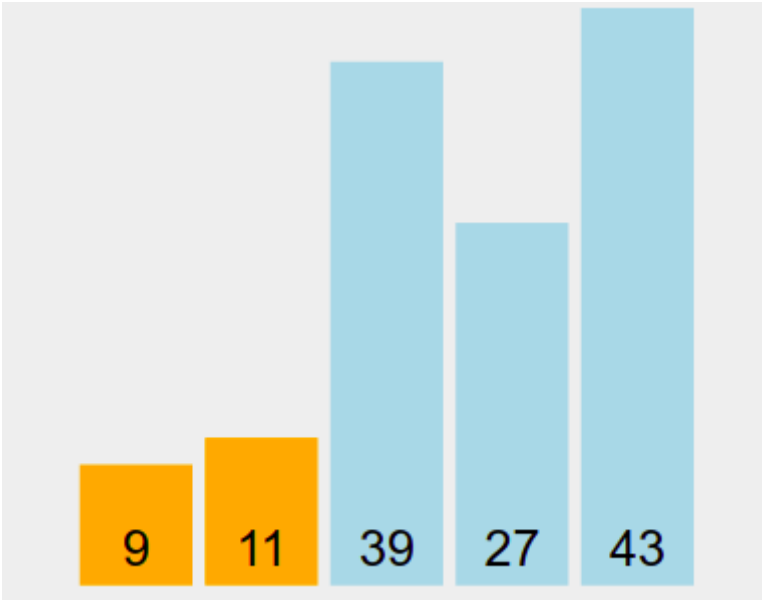
- 9 is now considered sorted.



Iteration 2: Set 11 as the current minimum, then iterate through the remaining unsorted elements to find the true minimum.



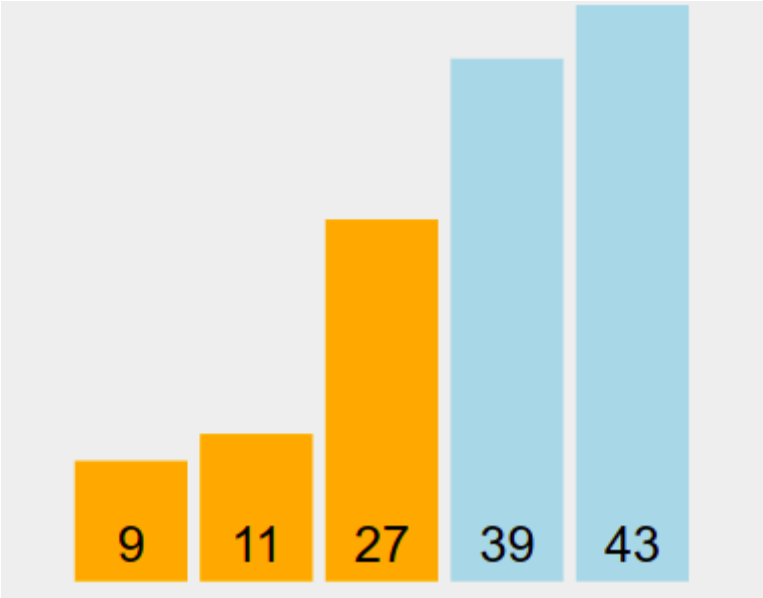
- Check if 39 is smaller than the current minimum (11).
- Check if 27 is smaller than the current minimum (11).
- Check if 43 is smaller than the current minimum (11).
- As the minimum is the first unsorted element, no swap is necessary.
- 11 is now considered sorted.



Iteration 3: Set 39 as the current minimum, then iterate through the remaining unsorted elements to find the true minimum.

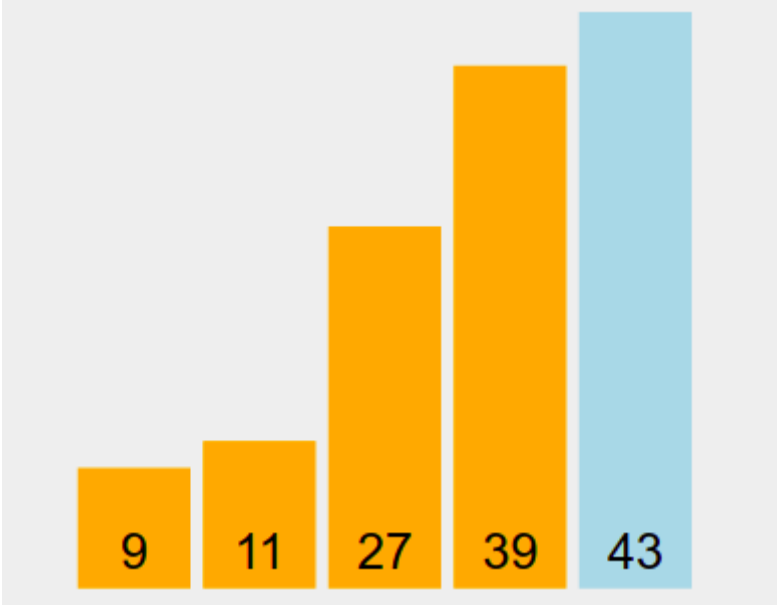
- Check if 27 is smaller than the current minimum (39).
- Set 27 as the new minimum.

- Check if 43 is smaller than the current minimum (27).
- Swap the minimum (27) with the first unsorted element (39).
- 27 is now considered sorted.



Iteration 4: Set 39 as the current minimum, then iterate through the remaining unsorted elements to find the true minimum.

- Check if 43 is smaller than the current minimum (39).
- As the minimum is the first unsorted element, no swap is necessary.
- 39 is now considered sorted.



List is sorted! (After all iterations, the last element will naturally be sorted.)

Implementation Code In C#:

```
public void SelectionShort(int[] arr,int n)
        {
            int temp, min;
            for (int i = 0; i < n - 1; i++)
            {
                min = i;
                for (int j = i + 1; j < n; j++)
                {
                    if (arr[j] < arr[min])
                    {
                    min = j;
                    }
                }
                if (min != i)
                {
                    temp = arr[min];
                    arr[min] = arr[i];
                    arr[i] = temp;
                }

            }
            Console.WriteLine();
            Console.Write("Sorted array is: ");
            for (int i = 0; i < n; i++)
            {
                Console.Write(arr[i] + " ");
            }
            Console.WriteLine("\n");
        }
```

4.2. Bubble sort

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst-case complexity are of $O(n^2)$ where n is the number of items.

Given an array of N elements, Bubble Sort will:

- Compare a pair of adjacent items (a, b),
- Swap that pair if the items are out of order (in this case, when a > b),
- Repeat Step 1 and 2 until we reach the end of array
- (the last pair is the (N-2)-th and (N-1)-th items as we use 0-based indexing),
- By now, the largest item will be at the last position.
- We then reduce N by 1 and repeat Step 1 until we have N = 1.

Without further ado, let's try Bubble Sort on the small example array [39, 11, 9, 27, 43].

You should see a 'bubble-like' animation if you imagine the larger items 'bubble up' (actually 'float to the right side of the array').

Set the swapped flag to false. Then iterate from index 1 to 4 inclusive.

- Checking if 39 > 11 and swap them if that is true. The current value of swapped = false.

- Swapping the positions of 39 and 11. Set swapped = true.

- Checking if 39 > 9 and swap them if that is true. The current value of swapped = true.

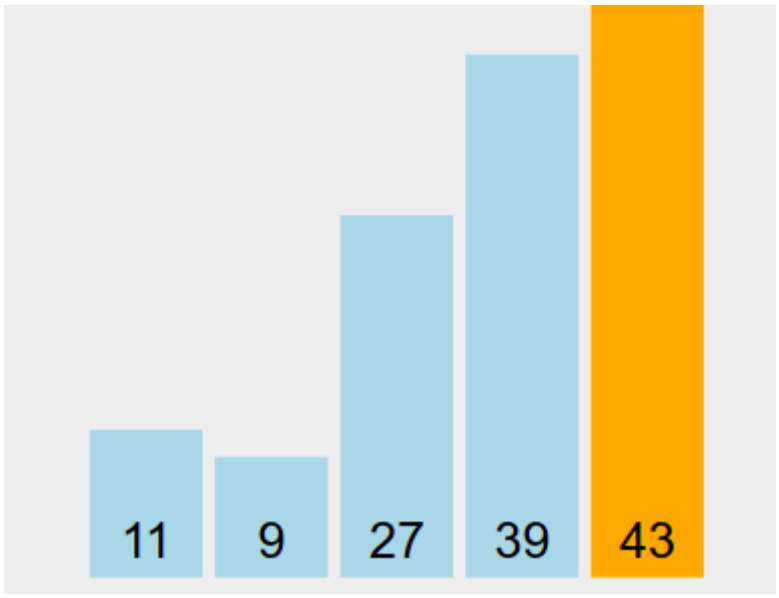
- Swapping the positions of 39 and 9. Set swapped = true.

- Checking if 39 > 27 and swap them if that is true. The current value of swapped = true.

- Swapping the positions of 39 and 27. Set swapped = true.

- Checking if 39 > 43 and swap them if that is true. The current value of swapped = true.

- Mark this element as sorted now. As at least one swap is done in this pass, we continue.



Set the swapped flag to false. Then iterate from index 1 to 3 inclusive.

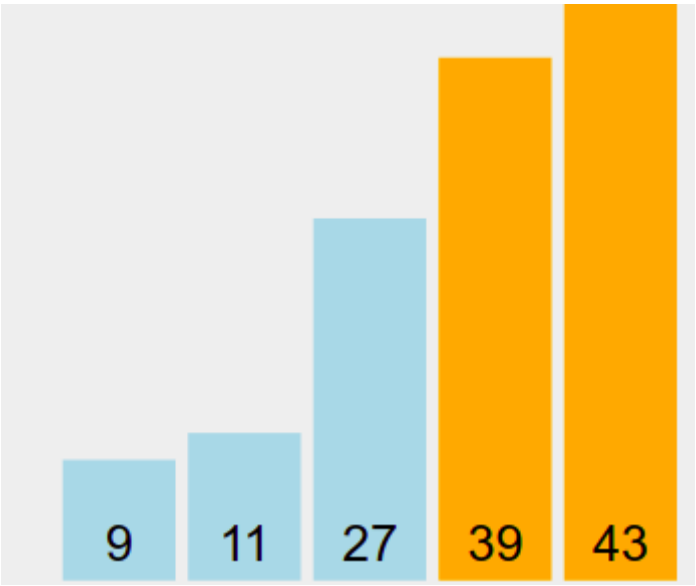
- Checking if 11 > 9 and swap them if that is true. The current value of swapped = false.

- Swapping the positions of 11 and 9. Set swapped = true.

- Checking if 11 > 27 and swap them if that is true. The current value of swapped = true.

- Checking if 27 > 39 and swap them if that is true. The current value of swapped = true.

- Mark this element as sorted now. As at least one swap is done in this pass, we continue.

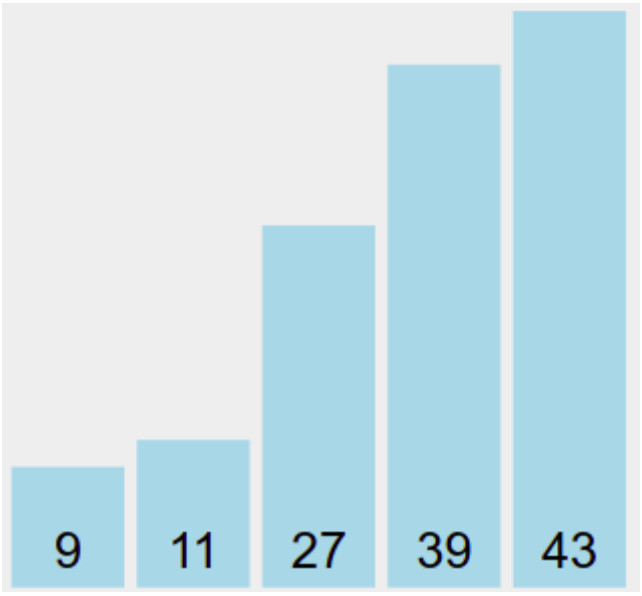


Set the swapped flag to false. Then iterate from index 1 to 2 inclusive.

Checking if 9 > 11 and swap them if that is true. The current value of swapped = false.

Checking if 11 > 27 and swap them if that is true. The current value of swapped = false.

- No swap is done in this pass. We can terminate Bubble Sort now.



List is sorted!

Implementation Code In C#:

```
public void BubbleShort(int[] arr,int n)
        {
            int temp;
            for (int i =0;i<n-1;i++)
            {
                for (int j=i+1;j<n;j++)
                {
                    if (arr[i]>arr[j])
                        {
                            temp=arr[i];
                            arr[i] = arr[j];
                            arr[j] = temp;
                        }
                }
            }
            Console.Write("Sorted array is: ");
            for (int i = 0; i < n; i++)
            {
                Console.Write(arr[i] + " ");
            }
            Console.WriteLine("\n");
        }
```

4.3. Compare performance

In a same array have 500 elements, Selection Short is faster than Bubble short because bubble sort uses more swap times, while selection sort avoids this.

Evaluation criteria	Bubble Short	Selection Short
Definition	A simple sorting algorithm that continuously steps through the list and compares the adjacent pairs to sort the elements	A sorting algorithm that takes the smallest value (considering ascending order) in the list and moves it to the proper position in the array.
Functionality	Compares the adjacent element and swap accordingly	Selecting the minimum element from the unsort sub-array and places it at the next position of the sorted sub-array
Efficiency	Less efficient	More efficient
Speed	Slower	Faster
Method	Uses item exchanging	Use item selection

Table 3. Compare 2 short algorithm

In summary, the main difference between bubble sort and selection sort is that the bubble sort operates by repeatedly swapping the adjacent elements if they are in the wrong order. In contrast, selection sort sorts an array by repeatedly finding the minimum element from the unsorted part and placing that at the beginning of the array.

5. The Advantages of encapsulation and information hiding when using an ADT.

Syntactically, abstract data type (ADT) is an enclosure that includes only:

- The data representation of one specific data type
- The subprograms that provide the operations for that type.

Through access controls, unnecessary details of the type can be hidden from units outside the enclosure that use the type. Program units that use an abstract data type can declare variables of that type, even though the actual representation is hidden from them. (kontmedikal, n.d.)

Although all the items in a class are private by default, programmers can change the access levels when needed. Three levels of access are available in both C++ and C# and an additional two in C# only.

- **Public:** All objects can access the data.
- **Protected:** Access is limited to members of the same class or descendants.
- **Private:** Access is limited to members of the same class.
- **Internal:** Access is limited to the current assembly. (C# only)
- **Protected Internal:** Access is limited to the current assembly or types derived from the containing class. (C# only). (thoughtco, n.d.)

An ADT encapsulates the data representation and makes data access possible at a higher level of abstraction and data can only be accessed via methods.

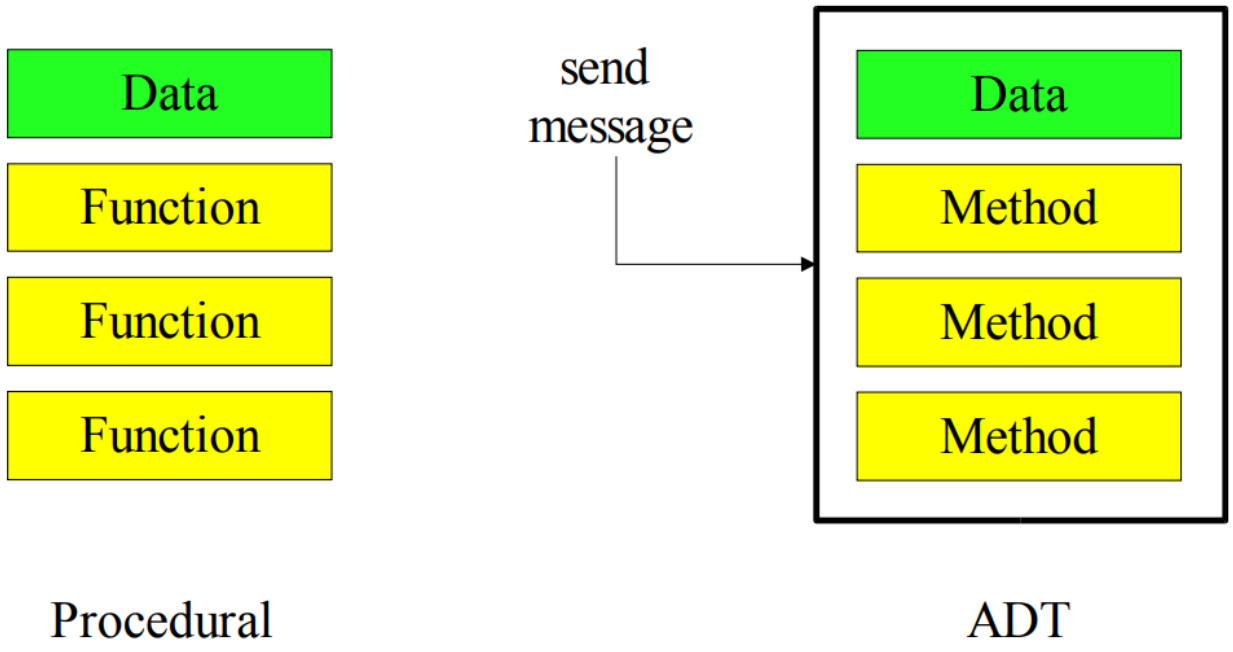
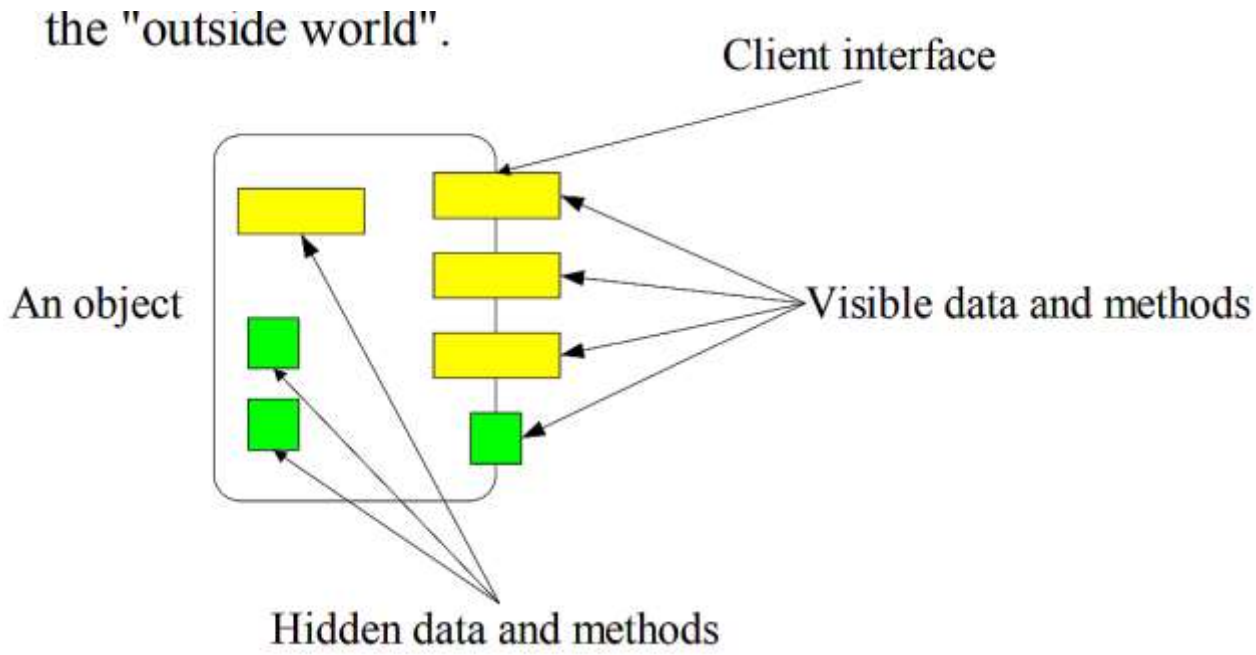


Figure 21. Example Procedural and ADT

The object is a "fire-wall" between the object and the "outside world" and the hidden data and methods can be changed without affecting the "outside world".



Sum up the advantages of encapsulation and information hiding when using an ADT:

- Encapsulation protects an object from unwanted access by clients.
- Encapsulation allows access to a level without revealing the complex details below that level.
- It reduces human errors.
- Simplifies the maintenance of the application
- Makes the application easier to understand.

Since there will usually be many different ways to implement an ADT, this implementation independence allows the programmer to switch the details of the implementation without changing the way the user of the data interacts with it. The user can remain focused on the problem-solving process.

6. The Operation, using illustrations of two network shortest path algorithm and example of each.

6.1. Dijkstra algorithm

Dijkstra's algorithm (or Dijkstra's Shortest Path First algorithm, SPF algorithm) is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks. It was conceived by computer scientist **Edsger W. Dijkstra** in 1956 and published three years later.

The algorithm exists in many variants. Dijkstra's original algorithm found the shortest path between two given nodes, but a more common variant fixes a single node as the "source" node and finds shortest paths from the source to all other nodes in the graph, producing a shortest-path tree.

For a given source node in the graph, the algorithm finds the shortest path between that node and every other. It can also be used for finding the shortest paths from a single node to a single destination node by stopping the algorithm once the shortest path to the destination node has been determined. (wikipedia, wikipedia, n.d.)
Dijkstra algorithm operates in steps, where at each step, the algorithm improves the distance values by determining the shortest distance from node s to another node. The state of each node consists of two features that got characterized by the algorithm:

- **Distance value:** distance value of a node is a scalar representing an estimate of the distance from node **s**.
- **Status label:** is an attribute specifying whether the distance value of a node is equal to the shortest distance to node s or not. The status label of a node is permanent if its distance value is equal to the shortest distance from node s, otherwise, it is temporary.

The **O((V+E) log V)** Dijkstra's algorithm is the most frequently used SSSP algorithm for typical input: Directed weighted graph that has no negative weight edge at all, formally: **∀ edge(u, v) ∈ E, w(u, v) ≥ 0**. Such weighted graph is very common in real life as traveling from one place to another always use positive time unit (s).

Dijkstra's algorithm can also be implemented differently. The **O((V+E) log V)** Modified Dijkstra's algorithm can be used for directed weight graphs that may have negative weight edges but no negative weight cycle.

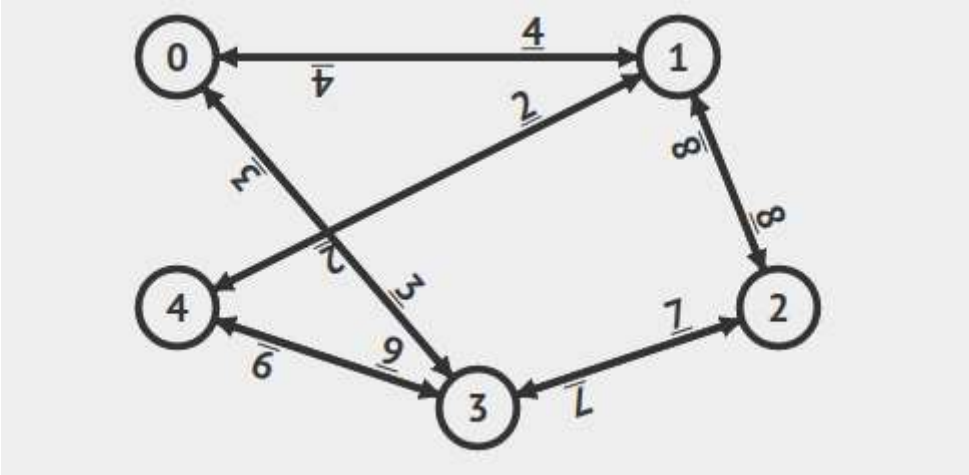
Such input graph appears in some practical cases, (For **example:** traveling using an electric car that has battery and our objective is to find a path from source vertex s to another vertex that minimizes overall battery usage. As usual, during acceleration (or driving on flat/uphill road), the electric car uses (positive) energy from the battery). However, during braking (or driving on downhill road), the electric car recharges (or use negative) energy to the battery. There is no negative weight cycle due to kinetic energy loss.

Dijkstra operation:
Dijkstra algorithm will assign some initial distance values to each node and will try to improve them step by step as follows:

- Mark all nodes un-visited. Create a set of all the un-visited nodes called the un-visited set.
- Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes. Set the initial node as current.
- For the current node, consider all of its un-visited neighbors and calculate their tentative distances through the current node. Compare the newly calculated tentative distance to the current assigned value and assign the smaller one.
- When we are done considering all of the un-visited neighbors of the current node, mark the current node as visited and remove it from the unvisited set. A visited node will never be checked again.
- If the destination node has been marked visited (when planning a route between two specific nodes) or if the smallest tentative distance among the nodes in the un-visited set is infinity (when planning a complete traversal; occurs when there is no connection between the initial node and remaining un-visited nodes), then stop. The algorithm has finished.
- Otherwise, select the un-visited node that is marked with the smallest tentative distance, set it as the new current node, and go back to step 3.

Dijksatra example:

Let understand with the following example:



Adjacency matrix:

	0	*1*	*2*	*3*	*4*
0	0	1	0	1	0
1	1	0	1	0	1
2	0	1	0	1	0
3	1	0	1	0	1
4	0	1	0	1	0

Table 4. Adjacency matrix Dijkstra

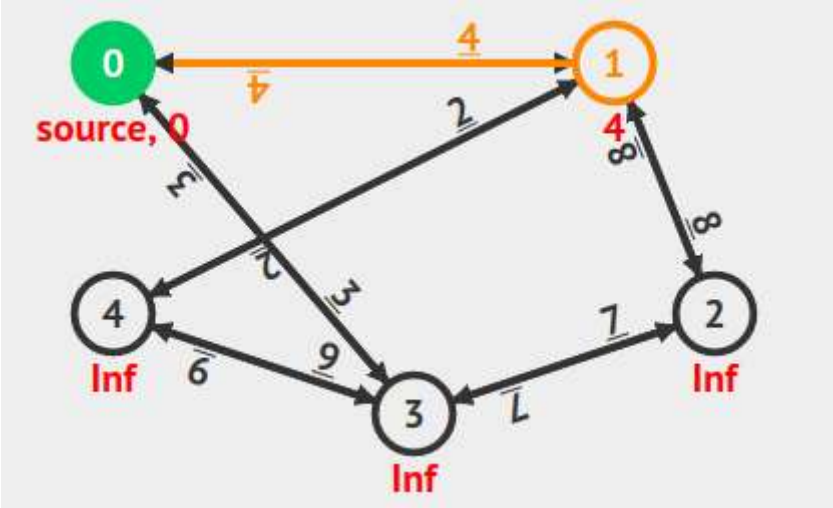
Weight matrix:

	0	1	2	3	4
0	0	4	0	3	0
1	4	0	8	0	2
2	0	8	0	7	0
3	3	0	7	0	9
4	0	2	0	9	0

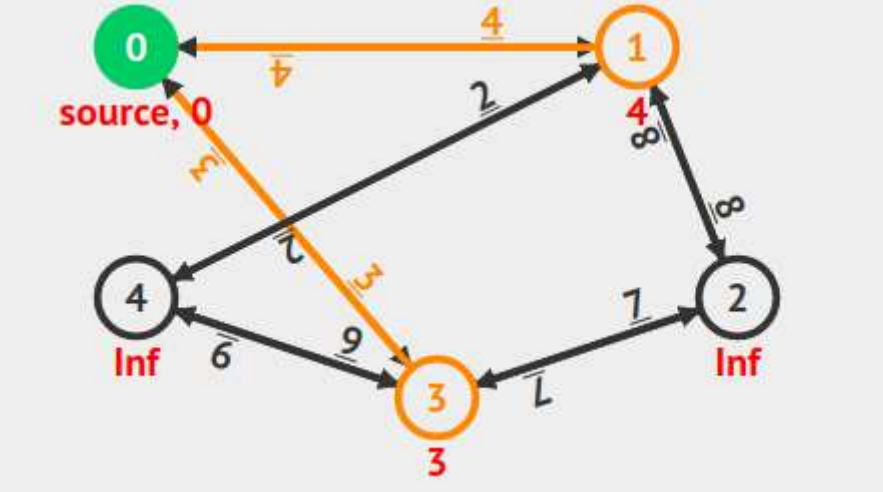
Table 5. Weight matrix Dijkstra

Find the shortest way:

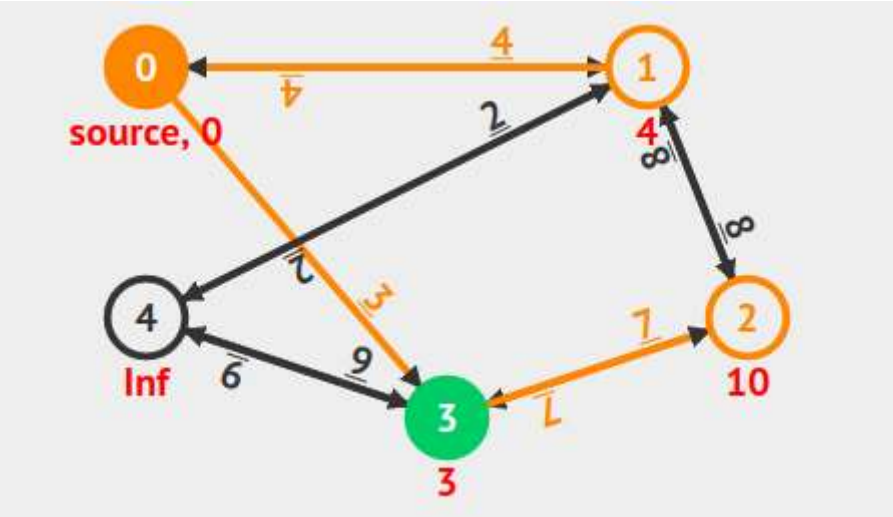
- 0 is the source vertex. Set p[v] = -1, d[v] = Inf, but d[0] = 0, PQ = {(0,0)}.
- The current priority queue {(0,0)}. Exploring neighbors of vertex u = 0, d[u] = 0.
- relax(0,1,4), #edge_processed = 1. d[1] = d[0]+w(0,1) = 0+4 = 4, p[1] = 0, PQ = {(4,1)}.



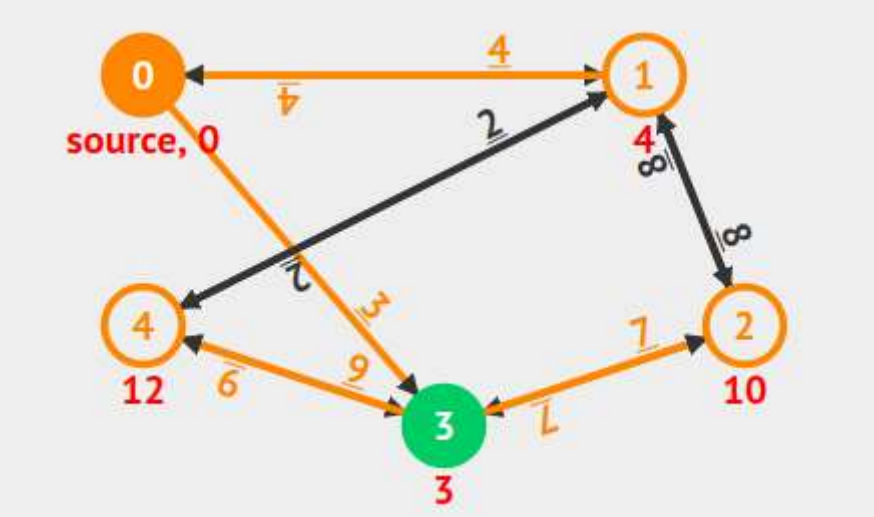
- relax(0,3,3), #edge_processed = 2. d[3] = d[0]+w(0,3) = 0+3 = 3, p[3] = 0, PQ = {(3,3), (4,1)}.



- d[0] = 0 can still be re-updated in the future as necessary as this vertex is only 'temporarily' completed.
- The current priority queue {(3,3), (4,1)}. Exploring neighbors of vertex u = 3, d[u] = 3.
- relax(3,0,3), #edge_processed = 3. d[3]+w(3,0) > d[0], i.e. 3+3 > 0, so there is no change.
- relax(3,2,7), #edge_processed = 4. d[2] = d[3]+w(3,2) = 3+7 = 10, p[2] = 3, PQ = {(4,1), (10,2)}.



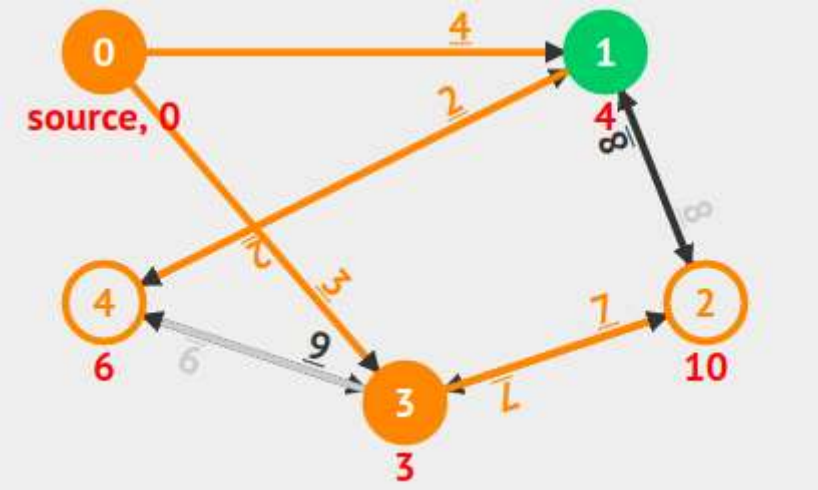
- relax(3,4,9), #edge_processed = 5. d[4] = d[3]+w(3,4) = 3+9 = 12, p[4] = 3, PQ = {(4,1), (10,2), (12,4)}.



- d[3] = 3 can still be re-updated in the future as necessary as this vertex is only 'temporarily' completed.

- The current priority queue {(4,1), (10,2), (12,4)}. Exploring neighbors of vertex u = 1, d[u] = 4.

- relax(1,0,4), #edge_processed = 6. d[1]+w(1,0) > d[0], i.e. 4+4 > 0, so there is no change.
- relax(1,2,8), #edge_processed = 7. d[1]+w(1,2) > d[2], i.e. 4+8 > 10, so there is no change.
- relax(1,4,2), #edge_processed = 8. d[4] = d[1]+w(1,4) = 4+2 = 6, p[4] = 1, PQ = {(6,4), (10,2), (12,4)}.



- d[1] = 4 can still be re-updated in the future as necessary as this vertex is only 'temporarily' completed.

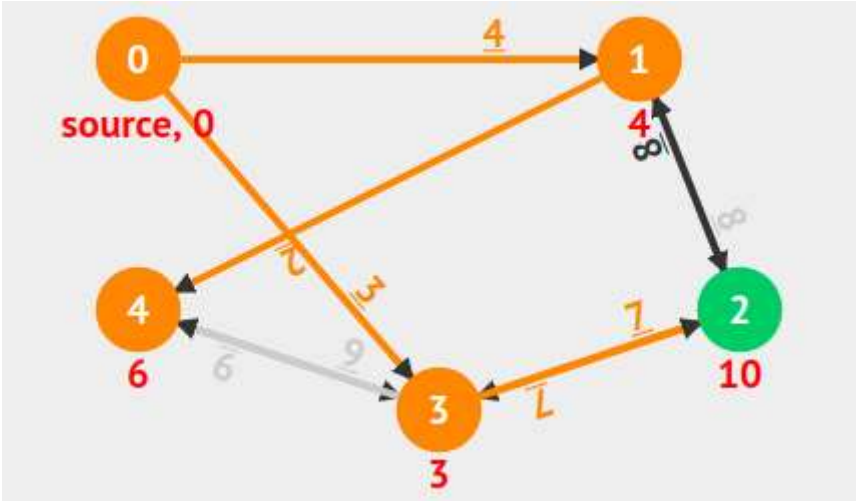
- The current priority queue {(6,4), (10,2), (12,4)}. Exploring neighbors of vertex u = 4, d[u] = 6.

- relax(4,1,2), #edge_processed = 9. d[4]+w(4,1) > d[1], i.e. 6+2 > 4, so there is no change.

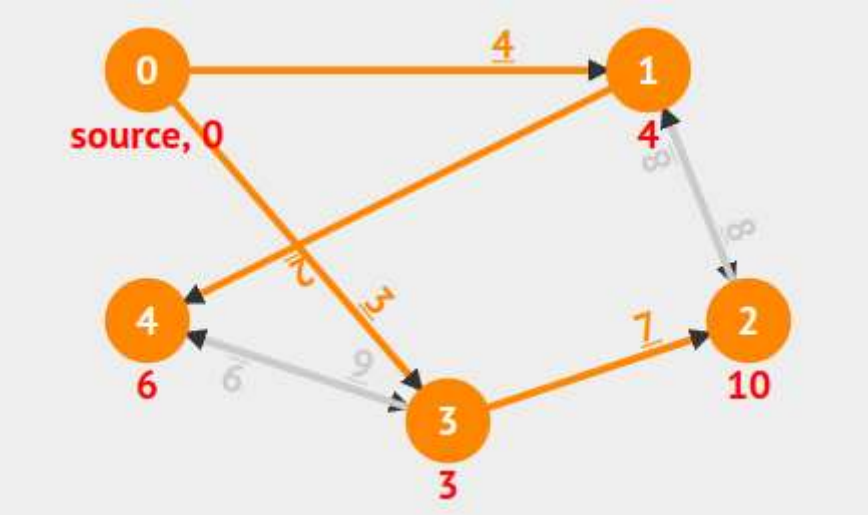
- relax(4,3,9), #edge_processed = 10. d[4]+w(4,3) > d[3], i.e. 6+9 > 3, so there is no change.

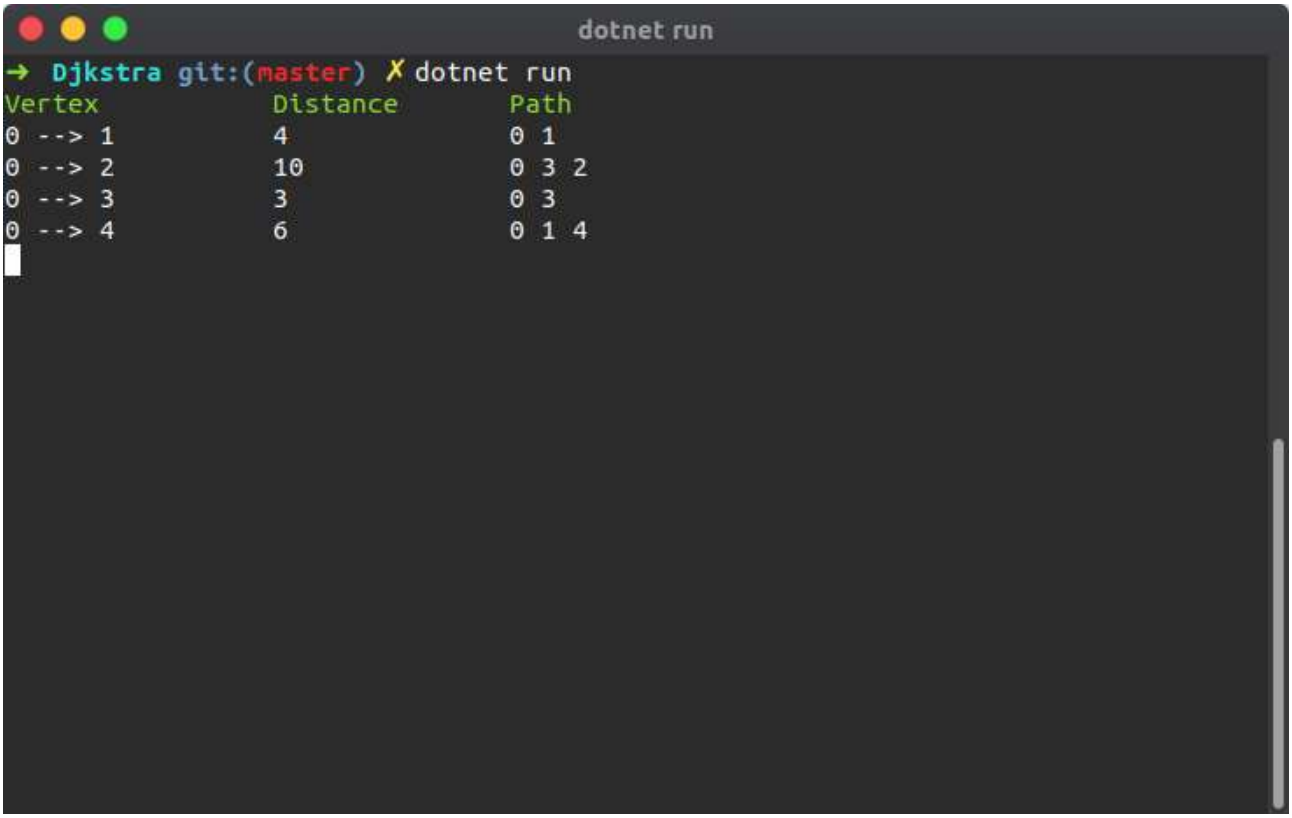
- d[4] = 6 can still be re-updated in the future as necessary as this vertex is only 'temporarily' completed.

- The current priority queue {(10,2), (12,4)}. Exploring neighbors of vertex u = 2, d[u] = 10.



- relax(2,1,8), #edge_processed = 11. d[2]+w(2,1) > d[1], i.e. 10+8 > 4, so there is no change.
- relax(2,3,7), #edge_processed = 12. d[2]+w(2,3) > d[3], i.e. 10+7 > 3, so there is no change.
- d[2] = 10 can still be re-updated in the future as necessary as this vertex is only 'temporarily' completed.
- The current priority queue {(12,4)}. (12,4) is an old information and skipped.
- #edge_processed = 12, O((V+E) log V) = 40. This is the SSSP spanning tree from source vertex 0





Implement Dijkstra in C#:

```
private static readonly int NO_PARENT = -1;

    private static void dijkstra(int[,] adjacencyMatrix, int startVertex)
    {
        int nVertices = adjacencyMatrix.GetLength(0);

        int[] shortestDistances = new int[nVertices];

        bool[] added = new bool[nVertices];


        for (int vertexIndex = 0; vertexIndex < nVertices; vertexIndex++)
        {
            shortestDistances[vertexIndex] = int.MaxValue;
            added[vertexIndex] = false;
        }

        shortestDistances[startVertex] = 0;

        int[] parents = new int[nVertices];

        parents[startVertex] = NO_PARENT;

        for (int i = 1; i < nVertices; i++)
        {

            int nearestVertex = -1;
            int shortestDistance = int.MaxValue;
            for (int vertexIndex = 0; vertexIndex < nVertices; vertexIndex++)
            {
                if (!added[vertexIndex] && shortestDistances[vertexIndex] <
shortestDistance)
                {
                    nearestVertex = vertexIndex;
                    shortestDistance = shortestDistances[vertexIndex];
                }
            }

            added[nearestVertex] = true;

            for (int vertexIndex = 0; vertexIndex < nVertices; vertexIndex++)
            {
                int edgeDistance = adjacencyMatrix[nearestVertex,vertexIndex];

                if (edgeDistance > 0 && ((shortestDistance + edgeDistance) <
shortestDistances[vertexIndex]))
                {
                    parents[vertexIndex] = nearestVertex;
                    shortestDistances[vertexIndex] = shortestDistance + edgeDistance;
                }
            }
        }

        printSolution(startVertex, shortestDistances, parents);
    }
```

6.2. Bellman Ford algorithm

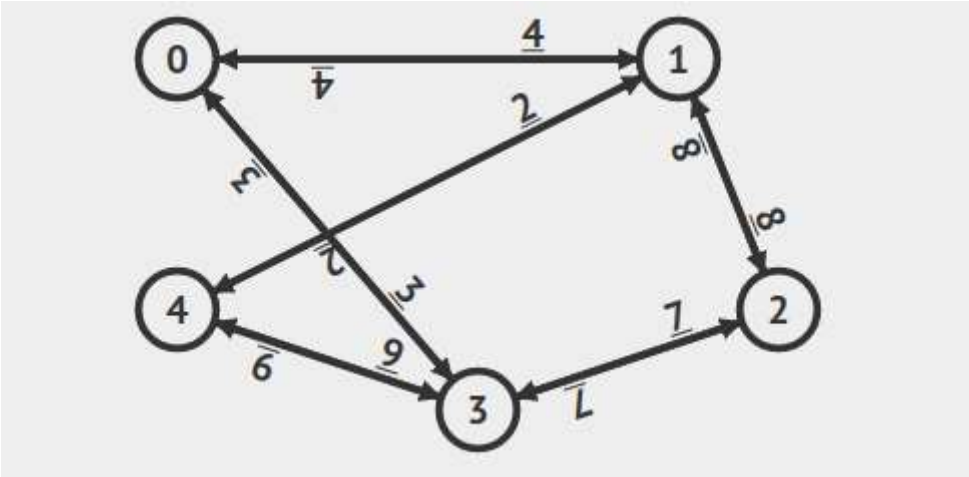
The **Bellman–Ford** algorithm is an algorithm that computes shortest paths from a single source vertex to all of the other vertices in a weighted digraph. It is slower than Dijkstra's algorithm for the same problem, but more versatile, as it is capable of handling graphs in which some of the edge weights are negative numbers. The algorithm was first proposed by **Alfonso Shimbel** (1955), but is instead named after Richard Bellman and Lester Ford, Jr., who published it in 1958 and 1956, respectively. Edward F. Moore also published the same algorithm in 1957, and for this reason it is also sometimes called the Bellman–Ford–Moore algorithm.

Negative edge weights are found in various applications of graphs, hence the usefulness of this algorithm. If a graph contains a "negative cycle" (**For example:** A cycle whose edges sum to a negative value) that is reachable from the source, then there is no cheapest path: any path that has a point on the negative cycle can be made cheaper by one more walk around the negative cycle. In such a case, the Bellman–Ford algorithm can detect and report the negative cycle.

Bellman-Ford Algorithm Operation:
The Bellman-Ford algorithm is based on the relaxation operation. This relaxation procedure takes two nodes as arguments and an edge connecting these nodes. If the distance from the source to the **first node (A)** plus the edge length is less than distance to the second node, than the first node is denoted as the predecessor of the second node and the distance to the second node is recalculated with this formula: (**distance(A) + edge.length**), otherwise, no changes are applied.
The path from the source node to any other node can be at maximum **|V |-1 edges long**, provided there is no cycle of negative length. If all nodes got performed the **|V |-1 relaxation** operation, the algorithm will find all shortest paths. The output should be verified by running the relaxation once more- if some edge will be relaxed, than the algorithm contains a cycle of negative length and the output is invalid, otherwise, the output is valid and the algorithm can return shortest path tree.

Describing this algorithm in the simple way, Bellman Ford algorithm works by overestimating the length of the path from the starting vertex to all other vertices. Then it iteratively relaxes those estimates by finding new paths that are shorter than the previously overestimated paths. By doing this repeatedly for all vertices, the end results are optimized-guaranteed.

Bellman Ford example:



Adjacency matrix:

	0	*1*	*2*	*3*	*4*
0	0	1	0	1	0
1	1	0	1	0	1
2	0	1	0	1	0
3	1	0	1	0	1
4	0	1	0	1	0

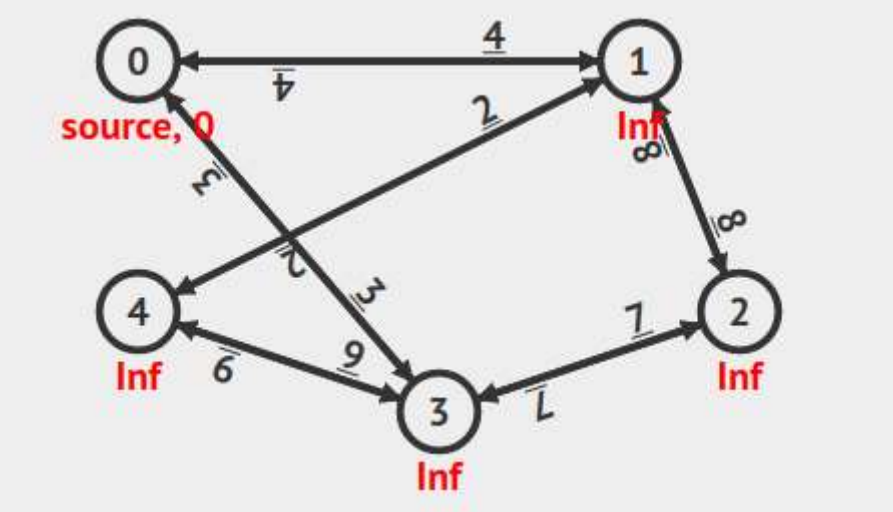
Table 6. Adjacency matrix Bellman Ford

Weight matrix:

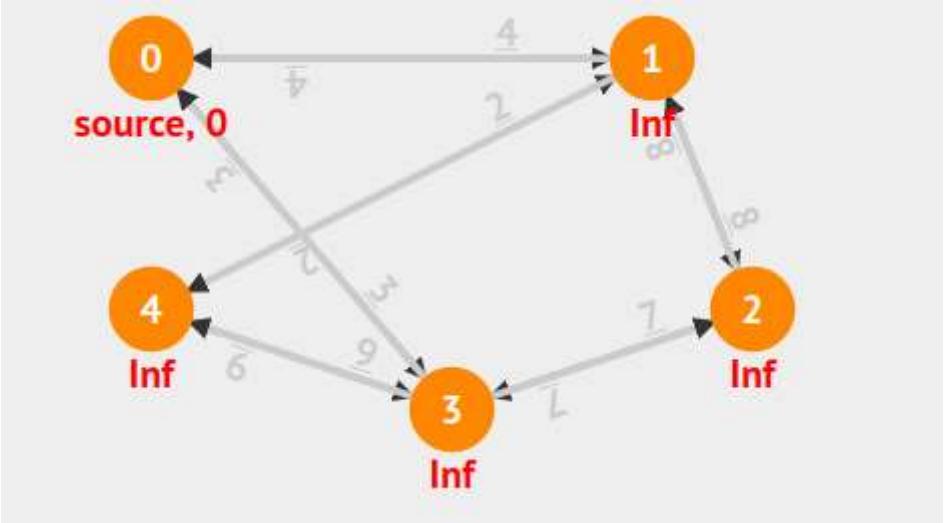
	0	*1*	*2*	*3*	*4*
0	0	4	0	3	0
1	4	0	8	0	2
2	0	8	0	7	0
3	3	0	7	0	9
4	0	2	0	9	0

Table 7. Weigth matrix Bellman Ford

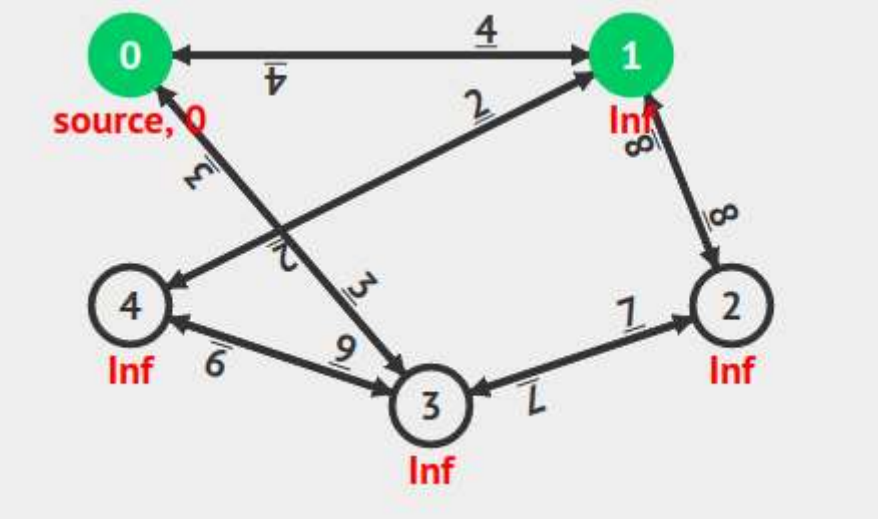
Find the shortest way:



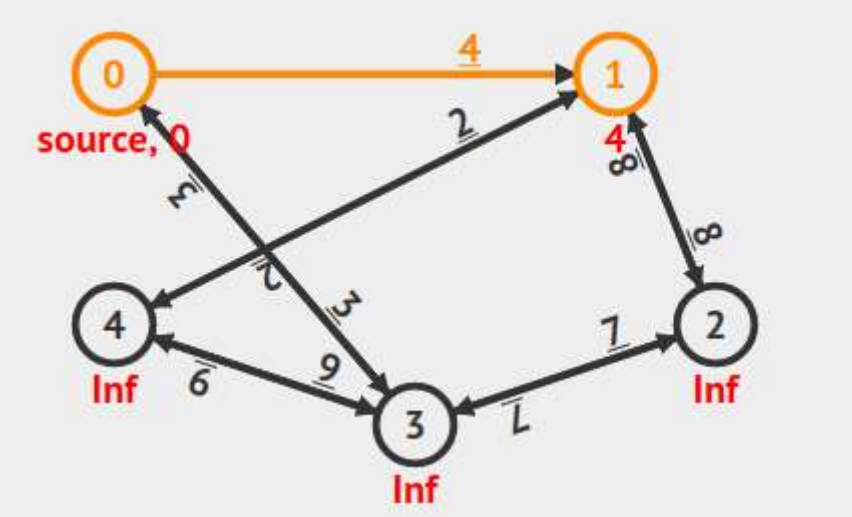
0 is the source vertex. Set p[v] = -1, d[v] = Inf, but d[" + sourceVertex + "] = 0.
This is the first pass. The highlighted edges are the current SSSP spanning tree so far.



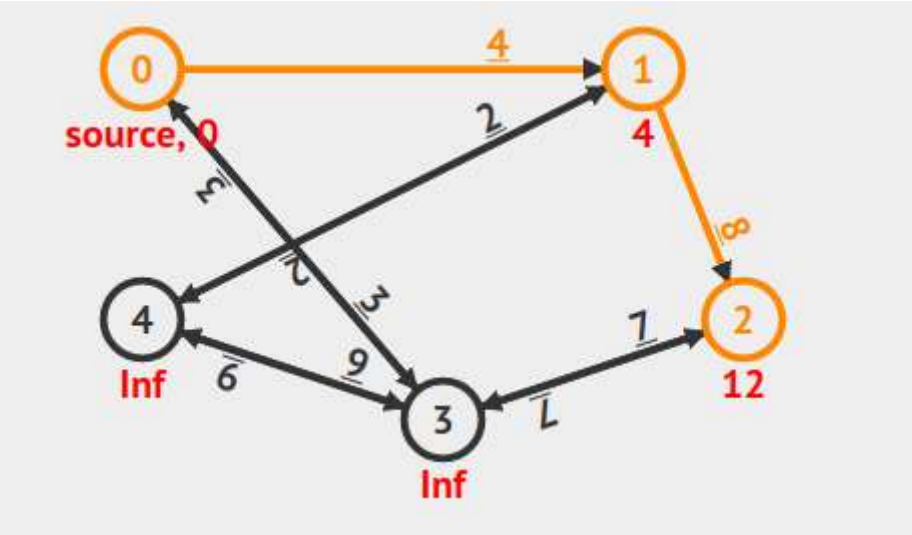
Prepare all edges for this #pass: 1.
#pass: 1, relax(0,1,4), #edge_processed = 1.
#pass: 1, relax(0,1,4), #edge_processed = 1. d[1] = 4, p[1] = 0.



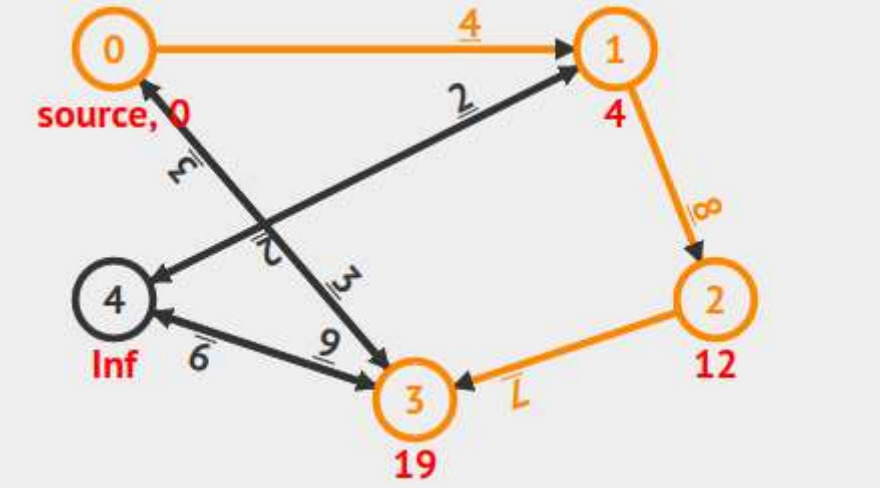
#pass: 1, relax(1,0,4), #edge_processed = 2.
#pass: 1, relax(1,0,4), #edge_processed = 2. No change.



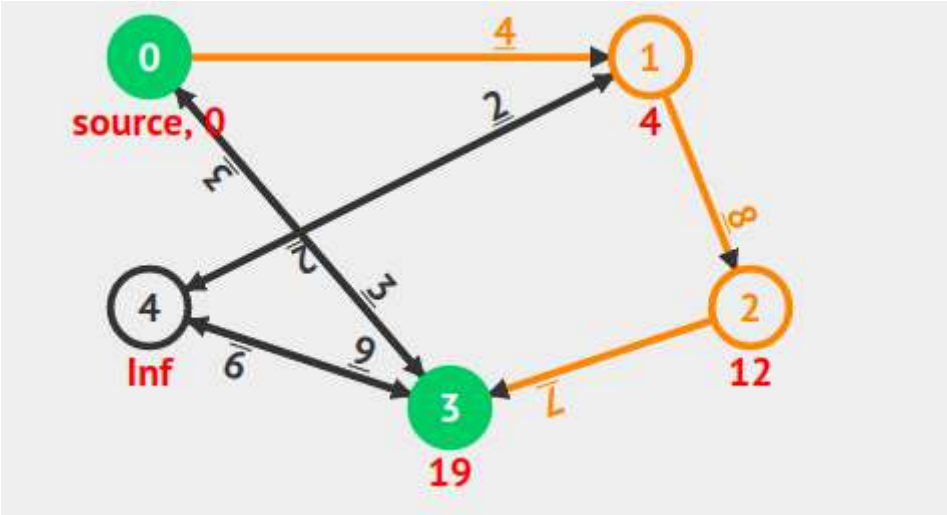
#pass: 1, relax(1,2,8), #edge_processed = 3.
#pass: 1, relax(1,2,8), #edge_processed = 3. d[2] = 12, p[2] = 1.
#pass: 1, relax(2,1,8), #edge_processed = 4.
#pass: 1, relax(2,1,8), #edge_processed = 4. No change.



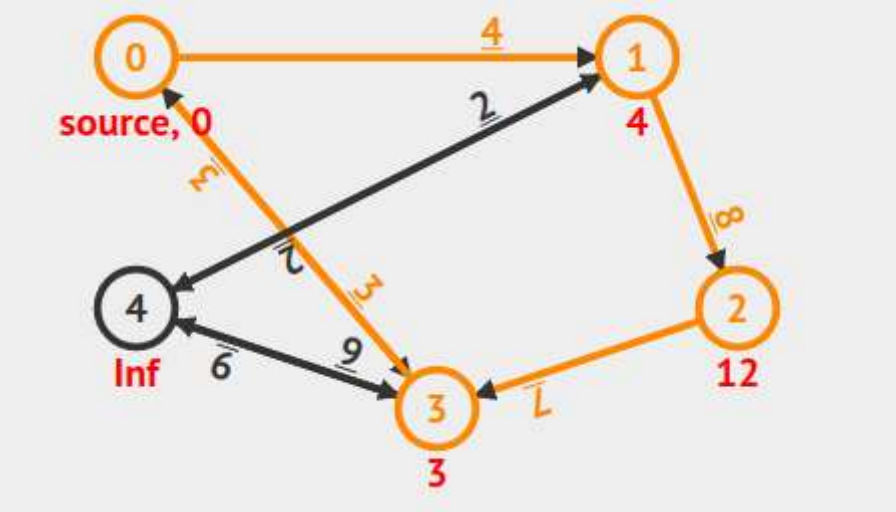
#pass: 1, relax(2,3,7), #edge_processed = 5.
#pass: 1, relax(2,3,7), #edge_processed = 5. d[3] = 19, p[3] = 2.
#pass: 1, relax(3,2,7), #edge_processed = 6.
#pass: 1, relax(3,2,7), #edge_processed = 6. No change.



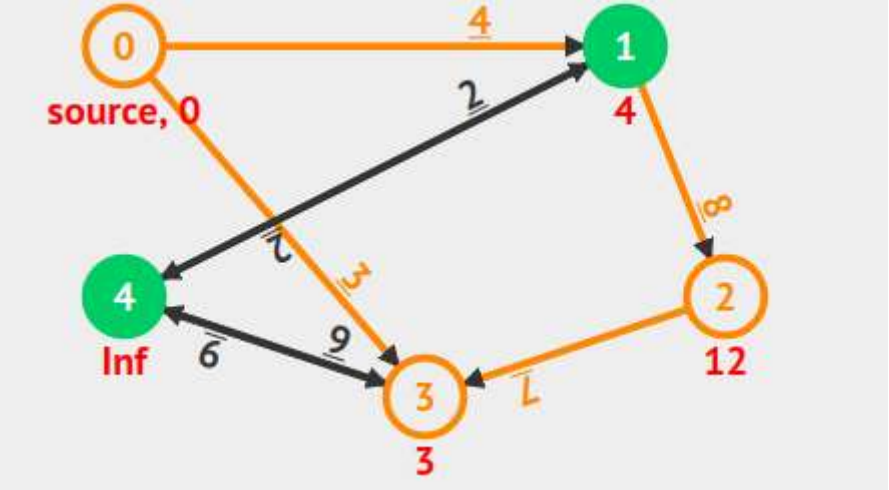
#pass: 1, relax(0,3,3), #edge_processed = 7.



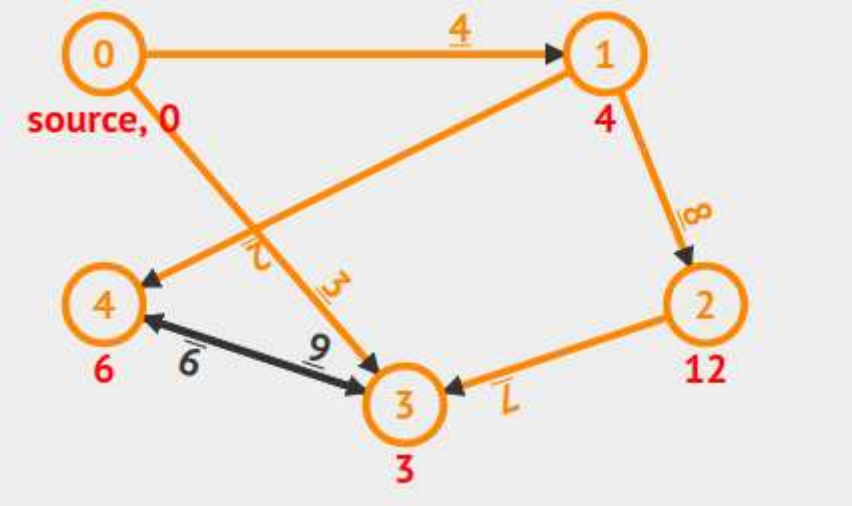
#pass: 1, relax(0,3,3), #edge_processed = 7. d[3] = 3, p[3] = 0.



#pass: 1, relax(3,0,3), #edge_processed = 8.
#pass: 1, relax(3,0,3), #edge_processed = 8. No change.
#pass: 1, relax(4,1,2), #edge_processed = 9.

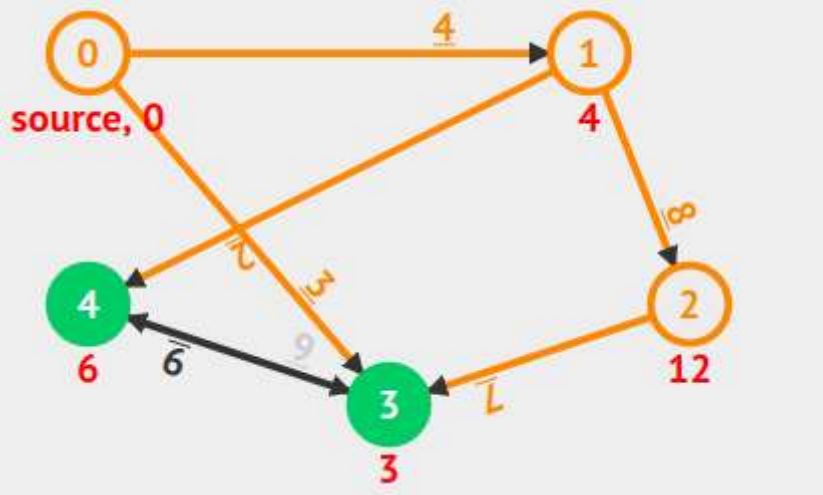


#pass: 1, relax(4,1,2), #edge_processed = 9. No change.
#pass: 1, relax(1,4,2), #edge_processed = 10.
#pass: 1, relax(1,4,2), #edge_processed = 10. d[4] = 6, p[4] = 1.



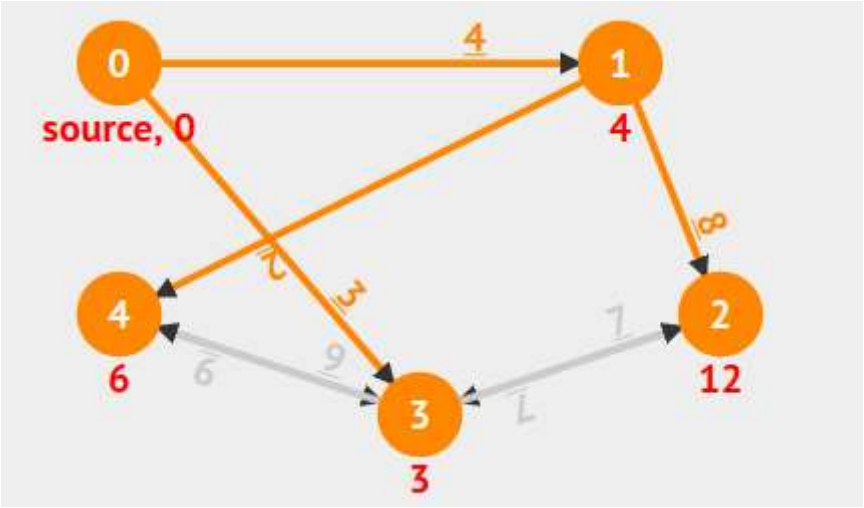
#pass: 1, relax(4,3,9), #edge_processed = 11.
#pass: 1, relax(4,3,9), #edge_processed = 11. No change.

#pass: 1, relax(3,4,9), #edge_processed = 12.



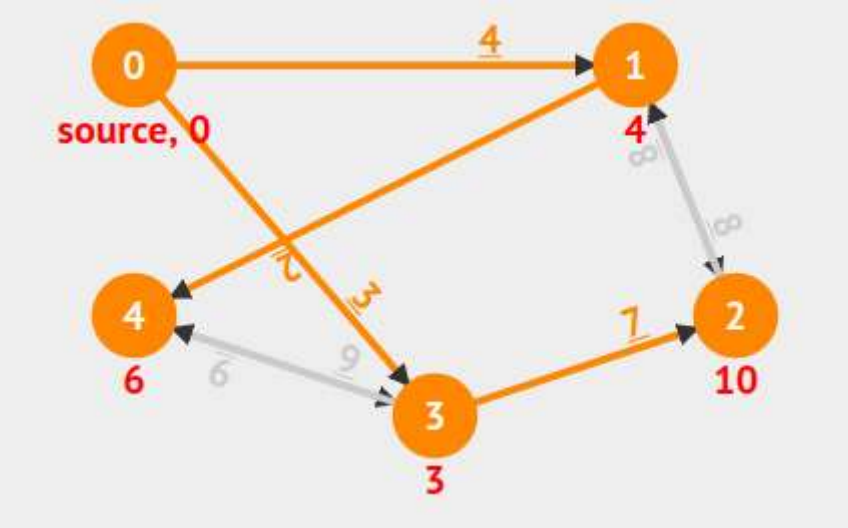
#pass: 1, relax(3,4,9), #edge_processed = 12. No change.

5 orange edge relaxation(s) in the last pass, we will continue. The highlighted edges are the current SSSP spanning tree so far.



Like #pass1 Prepare all edges for this #pass: 2, #pass: 3, #pass: 4. After that we will have:

#edge_processed = 48, V*E = 5*12 = 60. This is the SSSP spanning tree from source vertex 0.




```
dotnet run
→ BellManFord git:(master) ✗ dotnet run
Vertex Distance from Source
0               0
1               4
2               10
3               3
4               6
```

Implement Bellman-Ford in C#:

```
static void BellmanFord(int [,]graph, int V,
                        int E, int src)
{
    int []dis = new int[V];
    for (int i = 0; i < V; i++)
        dis[i] = int.MaxValue;
    dis[src] = 0;
    for (int i = 0; i < V - 1; i++)
    {
        for (int j = 0; j < E; j++)
        {
            if (dis[graph[j, 0]] + graph[j, 2] <
                dis[graph[j, 1]])
                dis[graph[j, 1]] =
                dis[graph[j, 0]] + graph[j, 2];
        }
    }
    for (int i = 0; i < E; i++)
    {
        int x = graph[i, 0];
        int y = graph[i, 1];
        int weight = graph[i, 2];
        if (dis[x] != int.MaxValue &&
                dis[x] + weight < dis[y])
            Console.WriteLine("Graph contains negative" +
                                        " weight cycle");
    }

    Console.WriteLine("Vertex Distance from Source");
    for (int i = 0; i < V; i++)
        Console.WriteLine(i + "\t\t" + dis[i]);
}
```

7. The view that imperative ADTs are basis for object orientation.

Abstract data types(ADT) are often called user-defined data types, because they allow programmers to define new types that resemble primitive data types. Just like a primitive type INTEGER with operations "+,−, ∗," an abstract data type has a type domain, whose representation is unknown to clients, and asset of operations defined on the domain. Abstract data types were first formulated in their pure form in CLU. The theory of abstract data types is given by existential types. They are also closely related to algebraic specification. In this context the phrase "abstract type" can be taken to mean that there is a type that is "conceived apart from concrete realities".

Object-oriented programming(OOP) involves the construction of objects which have a collection of methods, or procedures, that share access to private local state. Objects resemble machines or other things in the real world more than any well-known mathematical concept. In this tutorial, Smalltalk is taken as the paradigmatic object-oriented language. A useful theory of objects associates them with some form of closure, although other models are possible. The term "object" is not very descriptive of the use of collections of procedures to implement a data abstraction. Thus, we adopt the term procedural data abstraction as a more precise name for a technique that uses procedures as abstract data.

It is argued that **abstract data types** and procedural data abstraction are two distinct techniques for implementing abstract data. The basic difference is in the mechanism used to achieve the abstraction barrier between a client and the data. In **abstract data types**, the primary mechanism is type abstraction, while in procedural data abstraction it is procedural abstraction. This means, roughly, that in an ADT the data is abstract by virtue of an opaque type: one that can be used by a client to declare variables but whose representation cannot be inspected directly. In PDA, the data is abstract because it is accessed through a procedural interface – although all of the types involved may be known to the user. This characterization is not completely strict, in that the type of a procedural data value can be viewed as being partially abstract, because not all of the interface may be known; in addition, abstract data types rely upon procedural abstraction for the definition of their operations. (utexas, n.d.)

An abstract data type (ADT) is a mathematical model for a certain class of data structures that have similar behavior; or for certain data types of one or more programming languages that have similar semantics. An abstract data type is defined indirectly, only by the operations that may be performed on it and by mathematical constraints on the effects. Abstract datatype is not necessarily an OOP concept. It is an older term to describe the concepts of for example Stack and Queue in terms of their functionality, without describing the implementation. (stackoverflow, n.d.)

For Example: Suppose you have to make a program to deal with cars and motorbikes. You can define the classes (entities) of Car and Bike and you will see they have much (but not all) functionality in common. It would be a mistake to derive Car from Bike or the other way around. What you need to do is to define a common abstract base-class MotorVehicle and derive both Car and Bike from that class.

Data abstraction is an encapsulation of a data type and the subprograms that provide the operations for that type.

Programs can declare instances of that ADT, called an object. Object-oriented programming is based on this concept.

In object-orientation, ADTs may be referred to as classes. Therefore, a class defines properties of objects which are the instances in an object-oriented environment.

ADT and Object-oriented Programming are different concepts. OOPs use the concept of ADT. (toolbox., n.d.)

So, **ADTs are not one of the basic ideas behind Object orientation**. The basic ideas of Object-oriented programming are encapsulation (local retention, protection, and hiding of state-process), late-binding in all things and messaging.

CONCLUSION

Data Structure is a way of collecting and organising data in such a way that we can perform operations on these data in an effective way. Data Structures is about rendering data elements in terms of some relationship, for better organization and storage. For example, we have some data which has, player's name "Virat" and age 26. Here "Virat" is of String data type and 26 is of integer data type.

We can organize this data as a record like Player record, which will have both player's name and age in it. Now we can collect and store player's records in a file or database as a data structure.

If you are aware of Object Oriented programming concepts, then a class also does the same thing, it collects different type of data under one single entity. The only difference being, data structures provides for techniques to access and manipulate data efficiently.

In simple language, Data Structures are structures programmed to store ordered data, so that various operations can be performed on it easily. It represents the knowledge of data to be organized in memory. It should be designed and implemented in such a way that it reduces the complexity and increases the efficiency.

Bibliography

geeksforgeeks. (n.d.). *geeksforgeeks*. Retrieved from https://www.geeksforgeeks.org/selection-sort/

geeksforgeeks. (n.d.). *www.geeksforgeeks.org*. Retrieved from https://www.geeksforgeeks.org/stack-data-structure/

kontmedikal. (n.d.). *kontmedikal*. Retrieved from http://www.kontmedikal.com.tr/public/media/pdf/1518682272introduction.pdf

stackoverflow. (n.d.). *stackoverflow.com*. Retrieved from https://stackoverflow.com/questions/12982257/difference-between-abstract-data-type-and-object

thoughtco. (n.d.). *thoughtco*. Retrieved from https://www.thoughtco.com/definition-of-encapsulation-958068

toolbox. (n.d.). *it.toolbox.com*. Retrieved from https://it.toolbox.com/question/oop-vs-adt-051210

tutorialspoint. (n.d.). Retrieved from https://www.tutorialspoint.com/data_structures_algorithms/dsa_queue.htm

tutorialspoint. (n.d.). Retrieved from https://www.tutorialspoint.com/data_structures_algorithms/dsa_queue.htm

tutorialspoint. (n.d.). Retrieved from https://www.tutorialspoint.com/data_structures_algorithms/stack_algorithm.htm

utexas. (n.d.). *www.cs.utexas.edu*. Retrieved from https://www.cs.utexas.edu/users/wcook/papers/OOPvsADT/CookOOPvsADT90.pdf

visualgo. (n.d.). *visualgo.net*. Retrieved from https://visualgo.net/en/sorting?slide=1

w3schools. (n.d.). Retrieved from https://www.w3schools.in/data-structures-tutorial/queue/

wikipedia. (n.d.). *wikipedia*. Retrieved from https://en.wikipedia.org/wiki/Call_stack#/media/File:Call_stack_layout.svg

wikipedia. (n.d.). *wikipedia*. Retrieved from https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm