ASSIGNMENT 1 FRONT SHEET

Qualification	**BTEC Level 5 HND Diploma in Computing**		
Unit number and title	Unit **19: Data Structures & Algorithms**		
Submission date	12 August 2019	**Date Received 1st submission**	
Re-submission Date		**Date Received 2nd submission**	
Student Name	Huynh Thai Hieu	**Student ID**	GCD18314
Class	GCD0821	**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:		

DATA STRUCTURES & ALGORITHMS
Assignment 1

Huynh Thai Hieu
GCD18314
Class: GCD0821

1 August 2019

TABLE OF CONTENTS

TABLE OF FIGURES

1. INTRODUCTION TO DATA STRUCTURE

Data structure can be defined as the group of data elements which provides an efficient way of storing and organizing data in the computer so that it can enhance the performance of a software or a program as the main function of the software that to store and retrieve the user's data as fast as possible.

Data structures are widely used in almost every aspect of Computer Science such as Operating system, compiler design, artificial intelligence, graphics and many more.

Two of the most used Data structures are: **Stack** and **Queue**

1.1. STACK DATA STRUCTURE

Stack are dynamic data structures that follow the **Last In First Out (LIFO)** principle, which the last item to be inserted into a stack is the first one to be deleted from it. The *only* element of a stack that may be accessed is the one that was most recently inserted.

For example, a stack of plates on a table. The plate at the top of the stack is the first item to be moved if another plate is required from that stack.

Stack can be implemented using arrays, linked-lists, pointers and structures.

Stack have restrictions on the insertion and deletion of elements within it. Elements can be inserted or deleted only from one end of the stack, which is the top element of the stack, called as the *top* element.

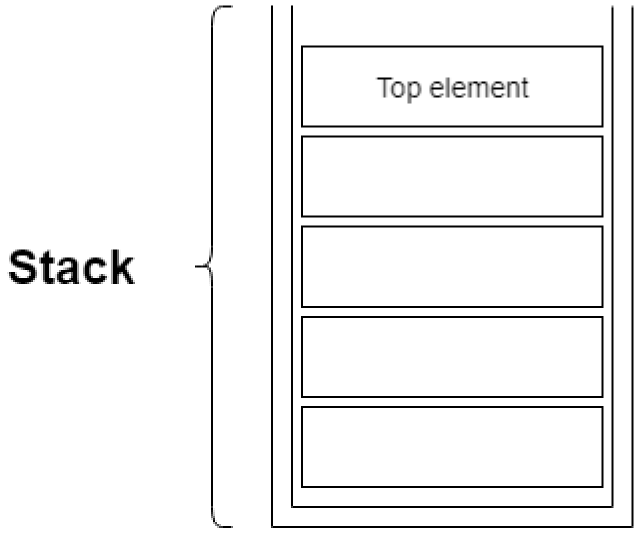


FIGURE 1. STACK VISUALIZATION

There are two basic operations on stack: *Push* (insert) and *Pop* (read and delete). Other operations on stack are: *Peek(), IsFull(), IsEmpty()*

Example for stack and its operations:

Name: S[]

Symbol: stack

Data type: integer array [5]

Data values: S=[, ,3,8,9,1]

Operators:

Operation	**Process**	**Results**
Push(7)	Add element ”7” to the top position of stack S[]	S=[**7**,3,8,9,1]
Pop()	Remove top element (“7”) from the stack S[]	S=[, ,3,8,9,1]

Peek()	Return the value of the top element (“3”) of the stack S[] (not deleting it)	S=[, ,3,8,9,1]
		Top= 3

IsFull()	Check if the stack S[] is full or not and return Boolean type value	False
IsEmpty()	Check if the stack S[] is empty or not and return Boolean type value	False

STACK OPERATION: *PUSH()*

Push() operation adds values into the top of the stack. When an element got added into the stack, it becomes the *top* element of the stack.

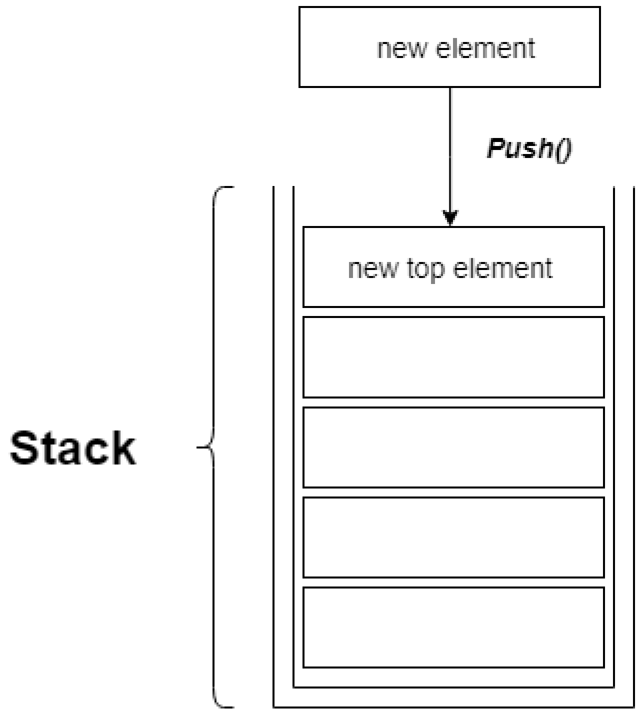


FIGURE 2. STACK'S PUSH() OPERATION

If the *Push()* operation call on a full stack will raise an Invalid Operation Exception (Stack overflow) where the stack elements array got fulfilled. Because of that, element numbers within the stack must be checked and be smaller than the maximum size of the stack array before using *Push()* operation.

Push() implementation in C#:

```
public static void Push(int[] Stack, int x)
        {
            if (top ==99)
            {
                Console.WriteLine("Stack overflow!");
            }
            else
            {
                top++;
                Stack[top]=x;
            }
         }
```

STACK OPERATION: *POP()*

Pop() removes an element from the top of the stack. When the *top* element of a stack is deleted, if the stack remains non-empty, the next element right below the previous *top* element becomes the new *top* element of a stack.

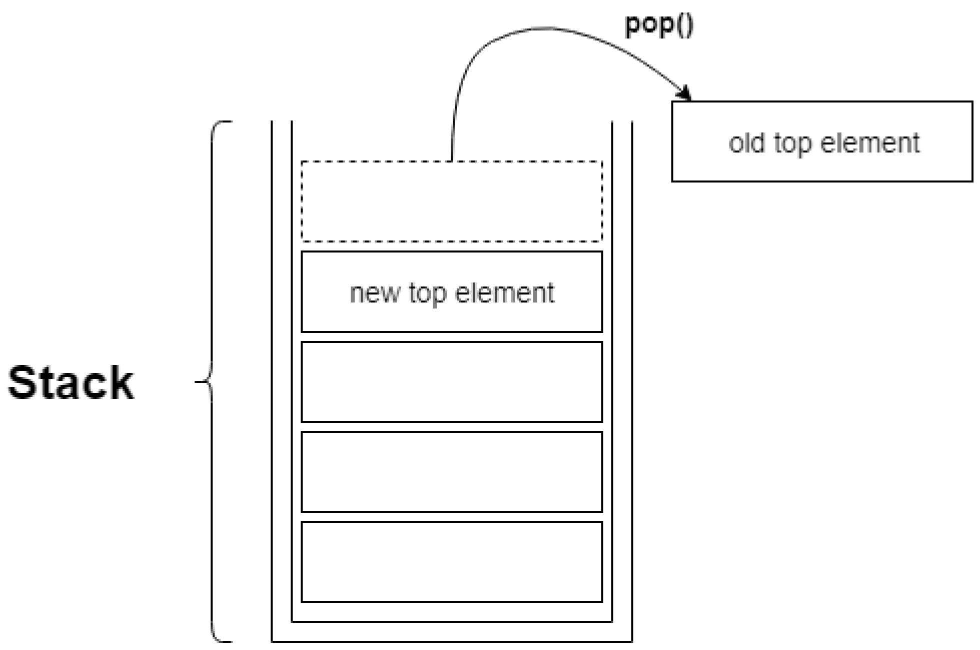


FIGURE 3. STACK'S POP() OPERATION

If the *Pop()* operation call on an empty stack will raise an Invalid Operation Exception (Stack underflow) where the stack having no element within it. Because of that, element numbers within the stack must be checked and be greater than 0 before using *Pop()* operation.

Pop() implementation in C#:

```
public static void Pop(int[] Stack, int n)
  {
      if (top < 0)
      {
          Console.WriteLine("Stack underflow!");
      }
      else
      {
          int x = Stack[top];
          top = top - 1;
          Console.WriteLine($"Poped element: {x}");
      }
   }
```

STACK OPERATION: *PEEK()*

Unlike *Pop(),* The *Peek()* operation returns the *top* element (top-most) value from the stack but not removing it.

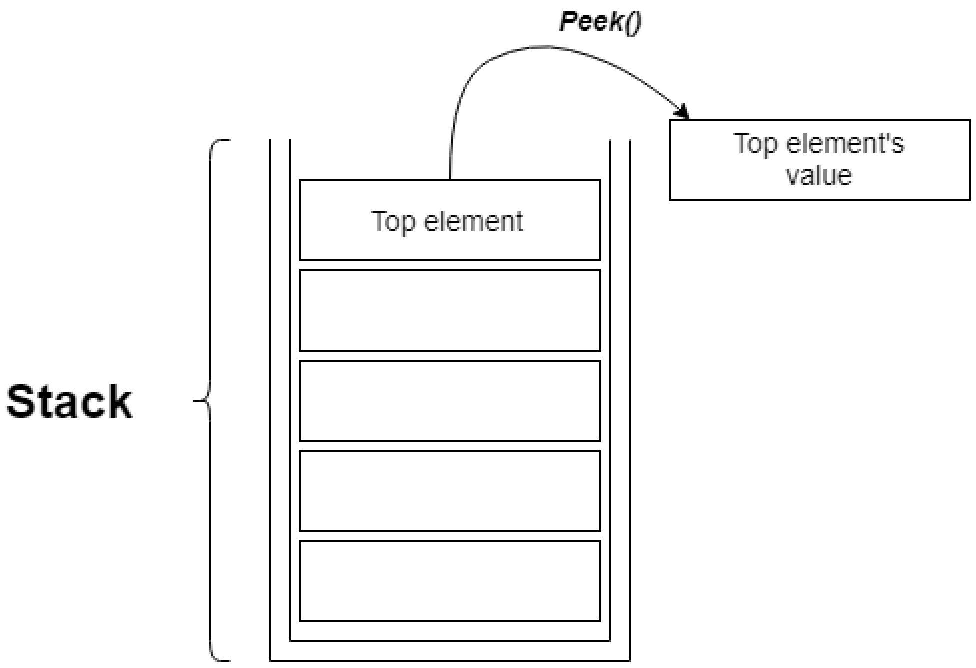


FIGURE 4. STACK'S PEEK() OPERATION

If the *Peek()* operation call on an empty stack will raise an Invalid Operation Exception (Stack underflow) where the stack having no element within it. Because of that, element numbers within the stack must be checked and be greater than 0 before using *Peek()* operation.

Peek() implementation in C#:

```
public static void Peek(int[] Stack, int n)
  {
      if (top < 0)
      {
          Console.WriteLine("Stack underflow!");
      }
      else
      {
          int x = Stack[top];
          Console.WriteLine($"Top element value peeked: {x}");
      }
  }
```

STACK OPERATION: *ISFULL() & ISEMPTY()*

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

IsFull() implementation in C#:

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

IsEmpty() operation is the operation that check whether a 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. *IsEmpty()* implementation in C#:

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

1.2. QUEUE DATA STRUCTURE

Queue is an abstract data structure that is similar to stack. However, unlike stack, a queue is open at both its ends, one end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows **First In First Out (FIFO)** methodology, also means that the data item stored first will be accessed first.

For example, a single-lane one-way road, where the vehicles enters first, exit first is the real-world example of queue. (queues at the ticket box and bus-stops)

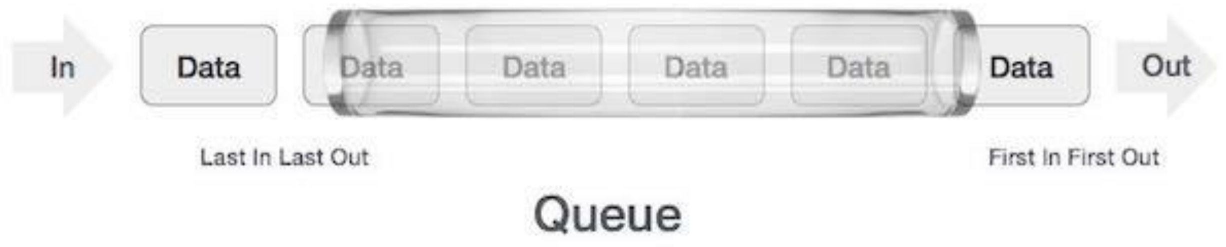


FIGURE 5. QUEUE VISUALIZATION (TUTORIALSPOINT, 2018)

There are two basic operations on queue: *Enqueue* (insert) and *Dequeue* (read and delete). Other operations on stack are: *Peek(), IsFull(), IsEmpty(), Size(),*

In queue, data got accessed by *dequeue()*, pointed by **front** pointer and use **rear** pointer while enqueuing (or inserting) data.

Example for queue and its operations:

Name: Q[]

Symbol: queue

Data type: integer array [5]

Data values: Q=[, ,3,8,9,1]

Operators:

Operation	**Process**	**Results**
Enqueue(7)	Add element "7" to the rear position of queue Q[]	Q=[**7**,3,8,9,1]
Dequeue()	Remove the front element ("1") from the queue Q[]	S=[,7,3,8,9,]

Peek()	Return the value of the front element ("9") of the queue Q[] (not deleting it)	S=[,7,3,8,9,]
		
		Front= 9

IsFull()	Check if the queue Q[] is full or not and return Boolean type value	False

IsEmpty()	Check if the queue Q[] is empty or not and return Boolean type value	False

QUEUE OPERATION: *ENQUEUE()*

Queue maintain two data pointers, **front** and **rear**. Therefore, its operations are comparatively difficult to implement than stacks.

The following steps should be taken to enqueue (insert) data into a 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.

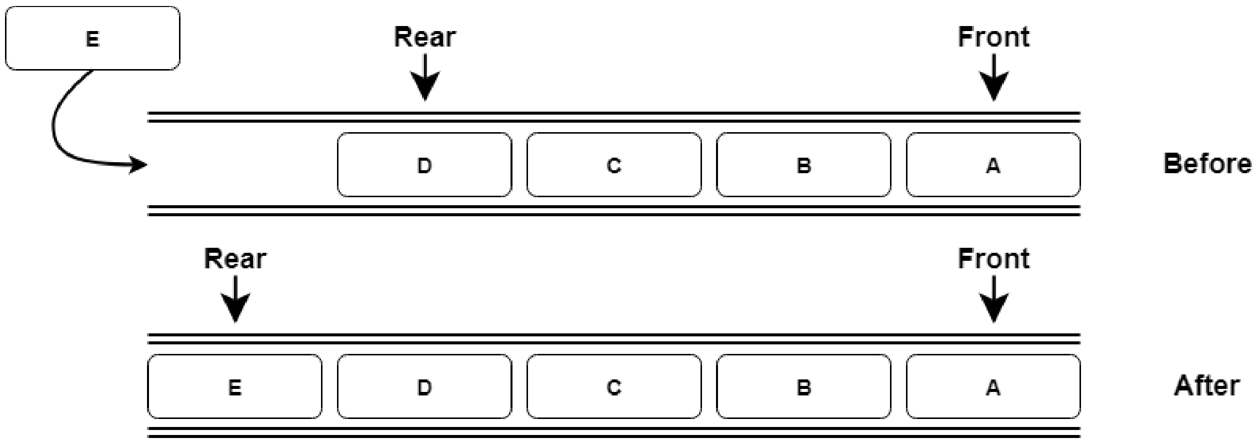


FIGURE 6. QUEUE ENQUEUE() OPERATION

Enqueue() implementation in C#:

```
    public static void Enqueue(char x)
                {
                    if (ENum() == 99)
                    {
                        Console.WriteLine("Queue is full");
                    }
                    {
                        Q[rear] = x;
                        rear = (rear + 1) % 100;
                    }
                 }
```

QUEUE OPERATION: D*EQUEUE()*

Accessing data from the queue is a process of two task: access the data where **front** is pointing and remove the data after access. The following steps 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.

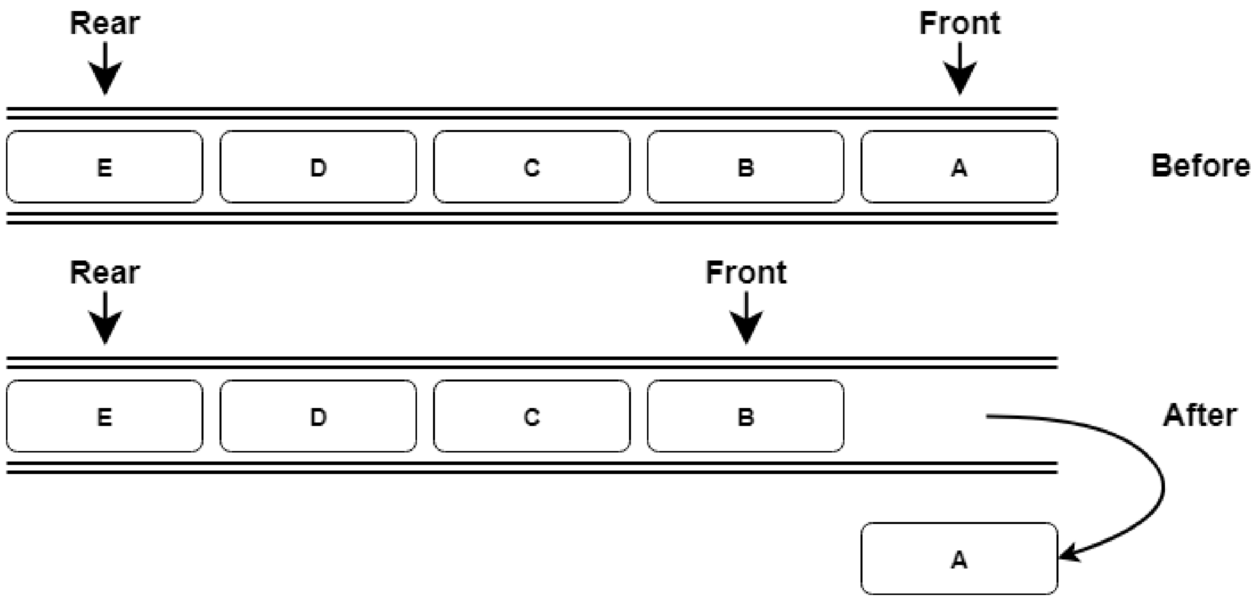


FIGURE 7. QUEUE DEQUEUE() OPERATION

Dequeue() implementation on C#:

```
public static char Dequeue()
            {
                char Temp;
                if (front == rear)
                {
                    Console.WriteLine("Queue is empty");
                }
                {
                    Temp = Q[front];
                    front = (front + 1) % 100;
                }
                return Temp;
            }
```

QUEUE OPERATION: PEEK*()*

Peek() is the operation that helps to see the data at the **front** of the queue but not deleting it from the queue.

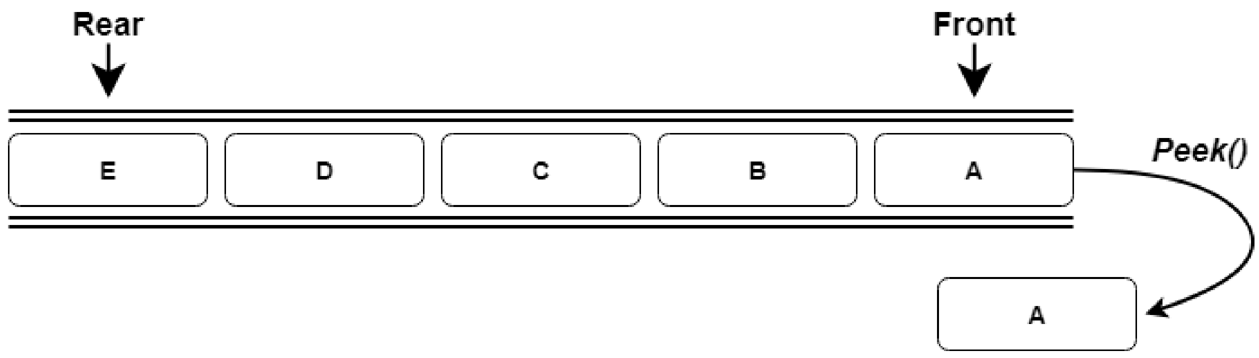


FIGURE 8. QUEUE PEEK() OPERATION

Peek() implementation on C#:

```
public static int Peek()
            {
                int Temp = 0;
                if (IsEmpty() == true)
                {
                    Console.WriteLine("Queue empty!");
                }
                else
                {
                    Temp = Q[front];
                }
                return Temp;
            }
```

QUEUE OPERATION: *ISFULL() & ISEMPTY()*

IsFull() is the operation that check for the **rear** pointer to reach at MAXSIZE to determine that the queue is full. The following *IsFull()* implementation code using single dimension array, if the queue got designed in a circular linked-list, the algorithm will differ.

IsFull() implementation on C#:

```
public static bool IsFull()
            {
                if (rear == MAXSIZE)
                {
                    return true;
                }
                else
                {
                    return false;
                }
            }
```

IsEmpty() is 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.

IsEmpty() implementation on C#:

```
public static bool IsEmpty()
            {
                if (front < 0 || front > rear)
                {
                    return true;
                }
                else
                {
                    return false;
                }
            }
```

2. OPERATIONS AND USAGE OF A MEMORY STACK

2.1. OPERATIONS OF A MEMORY STACK

A typical stack is an area of computer memory with a fixed origin and a variable size, where data is added or removed in a Last In First Out (LIFO) manner.

Stack based memory is a natural match for the way that variables are allocated and created by a program constructed as a set of nested method calls. When a method or a function is called, all of its local variables are created on the top of the stack – creating its so called *stack frame*.

Initially, the size of the stack is zero. A *stack pointer*, usually in the form of a hardware register, holds the address of a memory location. When the stack has a size of zero, the stack pointer points to the origin of the stack.

Basic operations of a memory stacks are:

- *Push* operation:
  The *push* operation stores the contents of another register on the stack and then moves the *stack pointer* on by one.

- *Pop* operation:
  The *pop* operation, sometimes called "pull" operation, reverses the *push* operation, which moves the *stack pointer* back by one and then retrieves the value from 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

- *Peek* operation:
  The *peek* operation inspects (or return) the topmost item value, but the stack pointer and stack size does not change (meaning the item remains on the stack).

- *Swap* or *exchange* operation:
  The *swap* operation pops 2 topmost items out of the stack and then pushed back but 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

In most modern computer system, each thread of the processor has a reserved region of memory referred to as its stack. When a function executes, it may add some of its state data to the top of the stack. As function are called, the number of *stack* frames increases, and the stack grows. When the function exits or return to their caller, it is responsible for removing that data from the stack, the number of *stack frame* decreases, and the stack shrinks.

A program is made up of one or more functions which interact by calling each other. Every time a function is called, an area of memory is set aside, called a *stack frame*, for the new function call. This area of memory holds some crucial information, like:

1. Storage space for all the automatic variables for the newly called function
2. The line number of the calling function to return to when the called function returns
3. The arguments or parameters of the called function

Each function call gets its own *stack frame*. Collectively, all the stack frames make up the call stack. 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)

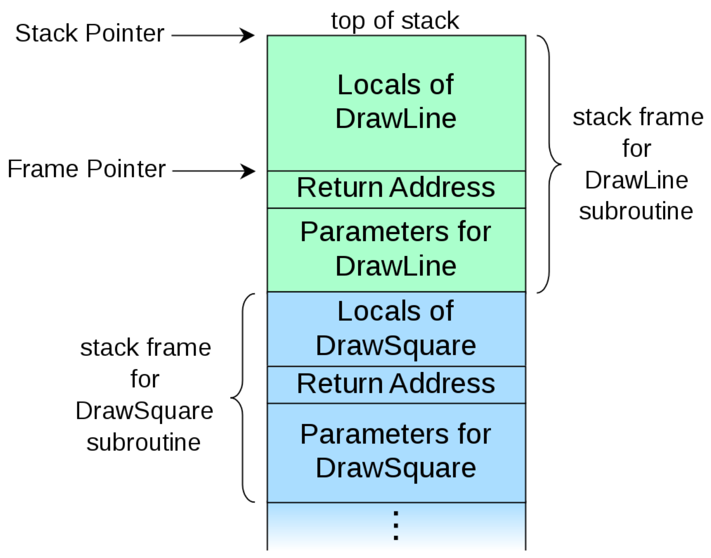


FIGURE 9. EXAMPLE OF STACK FRAME

(wikipedia.org, 2018)

Each *stack frame* corresponds to a call to a subroutine which has not yet terminated with a return. For example, if a subroutine function named *DrawLine* is currently running, having been called by a subroutine function *DrawSquare*, the top part of the call stack might be laid out like in **Figure 9.**

3. ADT SPECIFICATION FOR A SOFTWARE STACK

A stack can be implemented either through two basic type of **abstract data type**: array or linked list.

3.1. ARRAY STACK

An array is defined as a set of a definite, fixed number of homogeneous elements or data items. It means an array can contain one type data only, either all integers, all floating-point numbers or all characters, with specified size during the array declaration.

Element location is allocated and stored consecutively during the compile time.

As by the size of the array always needed to be specified during its declaration, arrays will require less memory, however, this makes resizing more problematic -essentially, needed to make a new array, and copy everything into it to resize it.

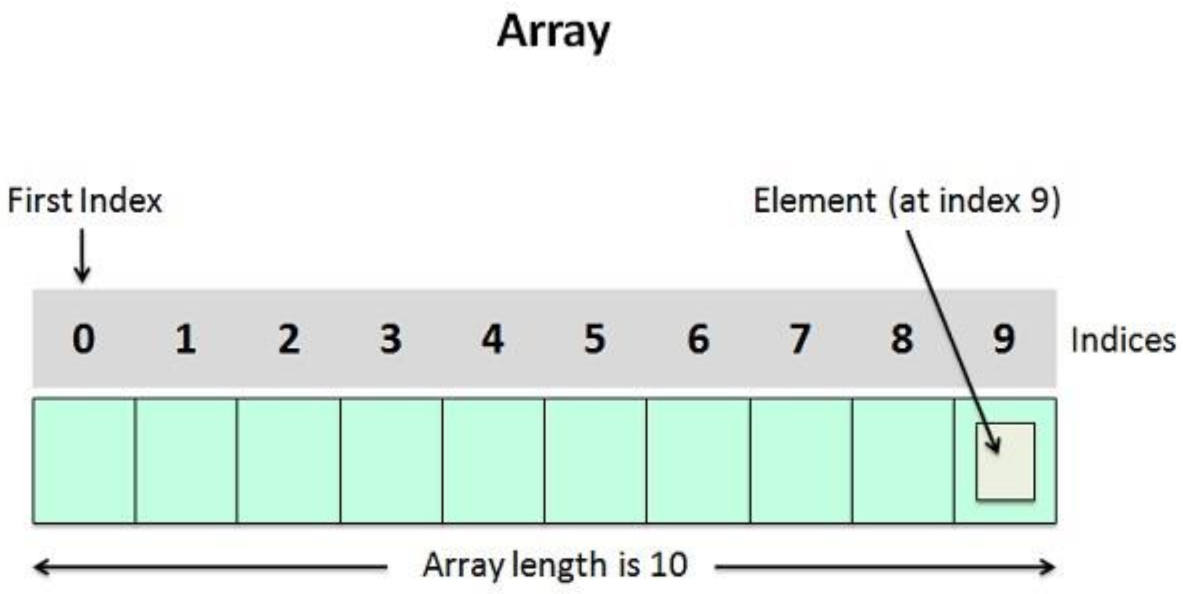


FIGURE 10.ARRAY CONFIGURATION (TECHDIFFERENCES, 2017)

It is common for stacks to be implemented using arrays rather than linked lists. Array stacks inherit the characteristics of array. The implementation of stacks using arrays is very simple, for example:

```
class ArrayStack
            {
                private Object[] array;
                private int count;
                public static final int MAX = 100;

                public ArrayStack()
                {
                    array = new Object[MAX];
                    count = 0;
                }

                public Object top()
                {
                    return array[count - 1];
                }

                public void pop()
                {
                    count--;
                }

                public void push(Object obj)
                {
                    array[count++] = obj;
                }

                public boolean isEmpty()
                {
                    return count == 0;
                }

            }
```

When a new stack is created (via the constructor) its array is set to the maximum size which it is considered the stack would ever grow to (in the example, set to 100), and its *cout* field is set to 0.

The advantage of using an array implementation for a stack is that none of the work associated with claiming new store as the size of the stack increases and garbage collecting it as it reduces.

As stacks are destructive and in general, a fixed amount of storage memory space set aside from the start for the stack (inherited from array characteristics) is not a big issue, even in fields of stack resizing or waste of empty declared store.

3.2. LINKED LIST STACK

Linked list is a particular list of some data elements linked to one other. In this, every element point to the next element which represents the logical ordering. Each element is called a *node*, which contains a pointer to its immediate successor node in the stack.

In linked list implementation of stack, the nodes are maintained non-contiguously in the memory. The top most node in the stack always contains null in its address field, while other node contains the address value, called as *pointer*, that leads to the upper node.

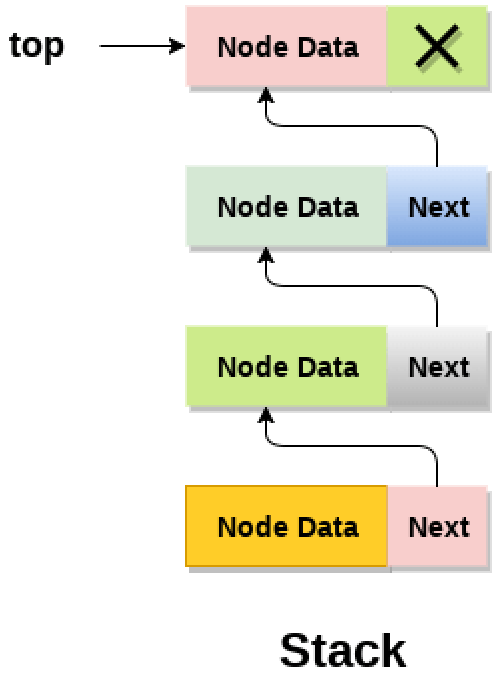


FIGURE 11. LINKED LIST STACK CONFIGURATION (JAVATPOINT.COM, 2018)

The implementation of stack using linked list can be describe in this following source code (in C#) as an example:

```
using System;

namespace linkedliststack
{
  public class Node
    {
     public int data;
     public Node next;
    }

  public static class GlobalMembers
    {
     public static Node top = null;
     public static void push(int val)
      {
        Node newnode = new Node();
        newnode.data = val;
        newnode.next = top;
        top = newnode;
       }
       public static void pop()
       {
        if (top == null)
         {
          Console.WriteLine("Stack Underflow");
         }
        else
         {
          Console.WriteLine("The popped element is ");
          Console.Write(top.data);
          Console.Write("\n");
          top = top.next;
         }
       }
       public static void display()
       {
        Node ptr;
        if (top == null)
         {
          Console.WriteLine("stack is empty");
         }
        else
         {
          ptr = top;
          Console.Write("Stack elements are: ");
          while (ptr != null)
            {
             Console.Write(ptr.data);
             Console.Write(" ");
             ptr = ptr.next;
            }
         }
        }

        static int Main()
        {
         int ch, val;
         Console.WriteLine("1) Push in stack");
         Console.WriteLine("2) Pop from stack");
         Console.WriteLine("3) Display stack");
         Console.WriteLine("4) Exit");
         do
          {
           Console.WriteLine("Enter choice: ");
           ch = Convert.ToInt32(Console.ReadLine());
           switch (ch)
           {
            case 1:
             {
              Console.WriteLine("Enter value to be pushed:");
              val = Convert.ToInt32(Console.ReadLine());
              push(val);
              break;
             }
            case 2:
             {
              pop();
              break;
             }
            case 3:
```

```
                {
                 display();
                 break;
                }
               case 4:
                {
                 Console.WriteLine("Exit");
                 break;
                }
               default:
                {
                 Console.WriteLine("Invalid Choice");
                }
                break;
             }
            } while (ch != 4);
            return 0;
           }
         }
}
```

In the above program, the structure Node is used to create the linked list that is implemented as a stack:

```
    public class Node
   {
    public int data;
    public Node next;
   }
```

The *push()* function takes argument *val* (value to be pushed into the stack). Then a new node is created and *val* is inserted into the data part. This node is added to the front of the lined list and top points to it:

```
    public static void push(int val)
    {
      Node newnode = new Node();
      newnode.data = val;
      newnode.next = top;
      top = newnode;
     }
```

The *pop()* function pops the topmost value of the stack, if there is any value. In case the stack is empty, underflow notification is printed:

```
public static void pop()
 {
  if (top == null)
   {
    Console.WriteLine("Stack Underflow");
   }
  else
   {
    Console.WriteLine("The popped element is ");
    Console.Write(top.data);
    Console.Write("\n");
    top = top.next;
   }
 }
```

The *display()* function displays all the elements in the stack. This is done by using ptr that initially points to top but goes till the end of the stack. All the data values corresponding ti ptr are printed:

```
public static void display()
 {
  Node ptr;
  if (top == null)
   {
    Console.WriteLine("stack is empty");
   }
  else
   {
    ptr = top;
    Console.Write("Stack elements are: ");
    while (ptr != null)
      {
       Console.Write(ptr.data);
       Console.Write(" ");
       ptr = ptr.next;
      }
   }
  }
```

The main advantage of using linked list over an array is that it is possible to implements a stack that can shrink or grow as much as needed. In using array will put a restriction to the maximum capacity of the array which can lead to stack overflow. Here each new node will be dynamically allocated so overflow is not possible. However, due to its unrestricted to size and easily expand, linked list stack require more memory base than array stack.

4. FIRST IN FIRST OUT (FIFO) QUEUE

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

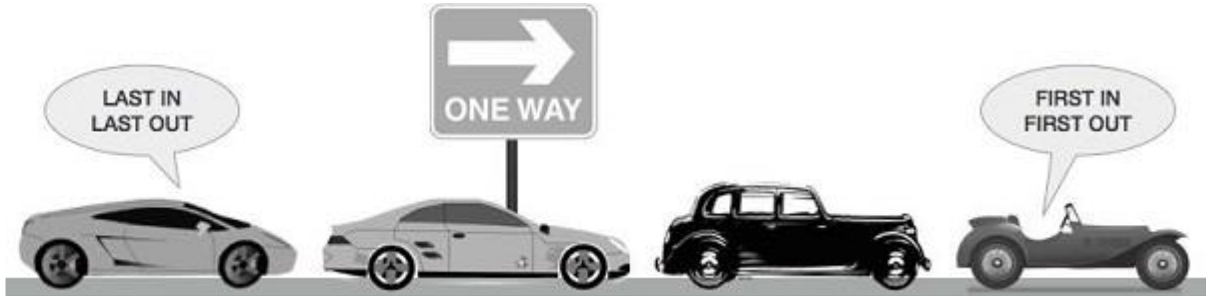


FIGURE 12. FIFO QUEUE EXAMPLE (TUTORIALSPOINT.COM, 2018)

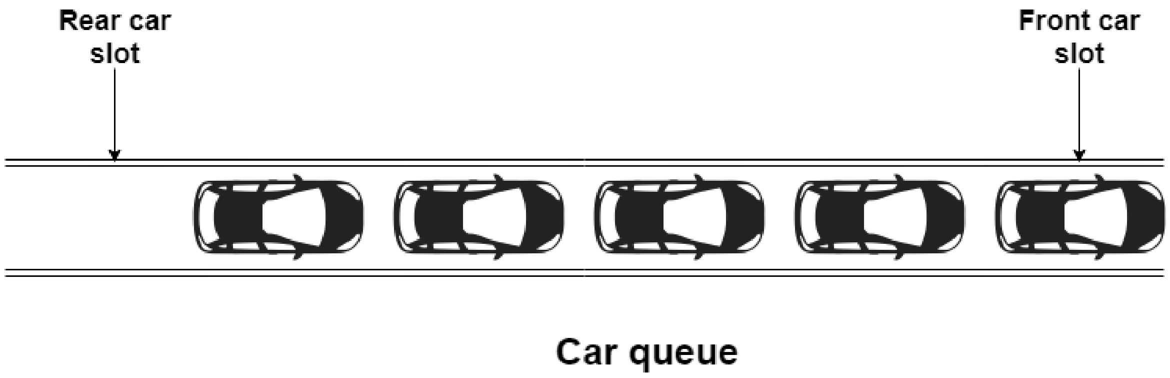


FIGURE 13. FIFO CAR QUEUE EXAMPLE

A real-world example of queue can be a single-lane one-way road, where the vehicle enters first then exits first.

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 integer 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 14**).

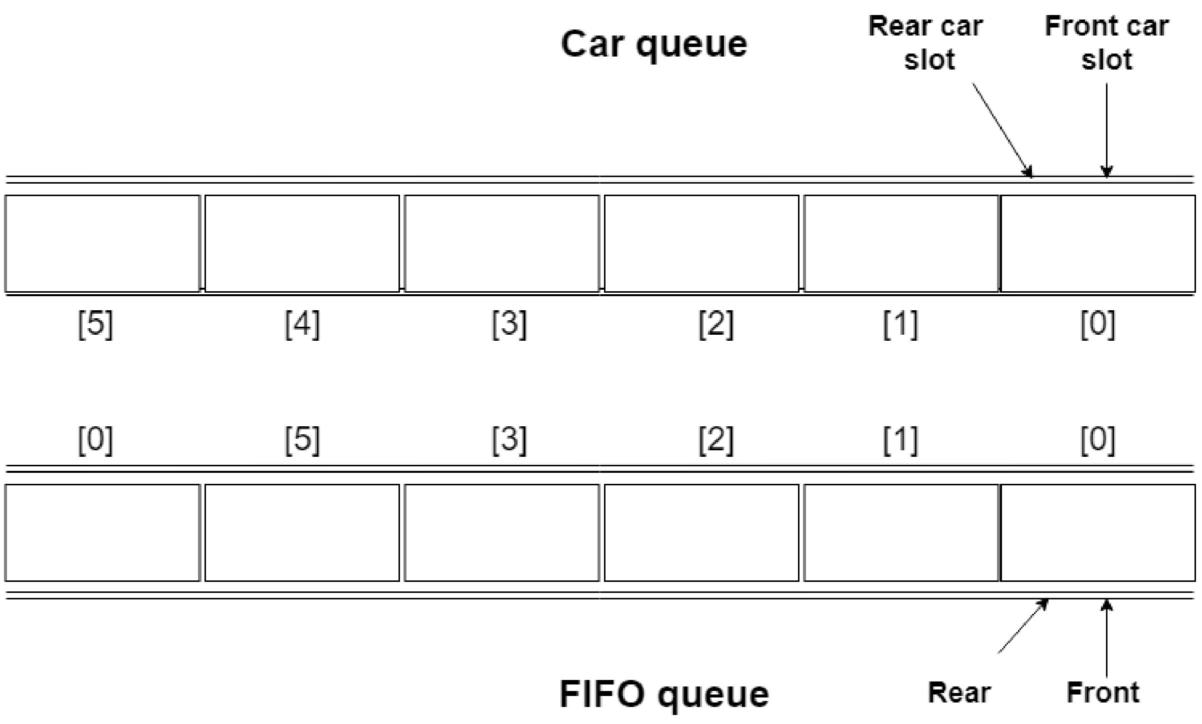


FIGURE 14.QUEUE ILLUSTRATION

As more elements (cars) add into the queue, the rear (rear car position) keeps on moving ahead, always pointing to the position where the next element (car) will be inserted, while the front (front car position) remains at the first index of the queue (**Figure 15 & Figure 16**).

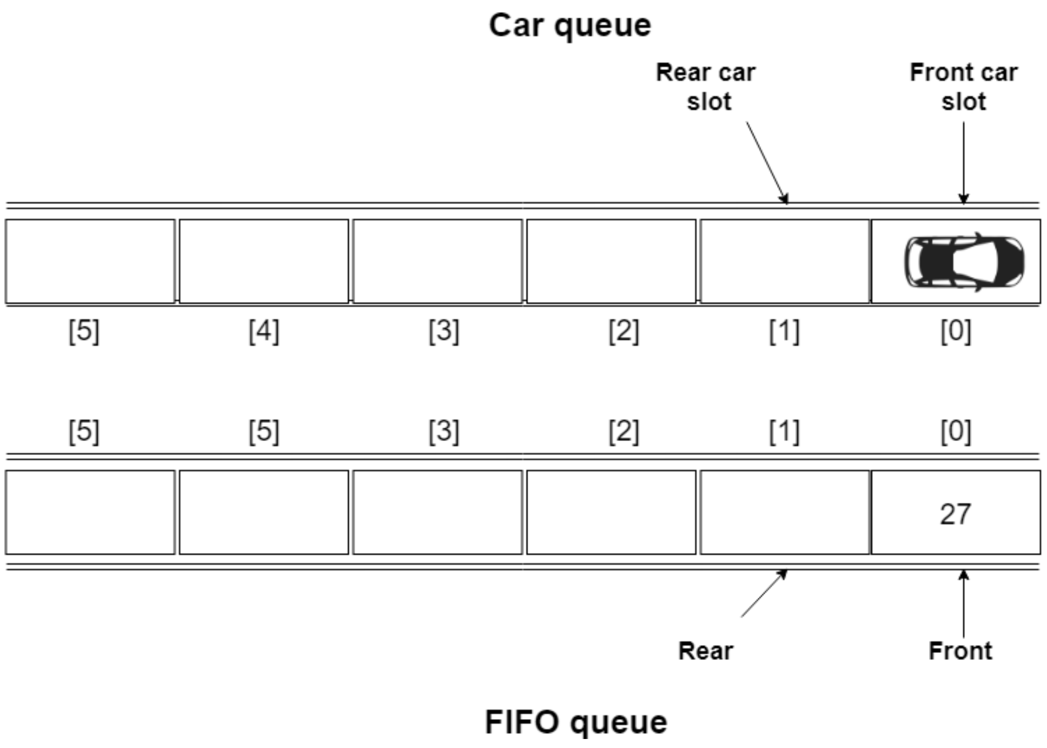


FIGURE 15.QUEUE ENQUEUE OPERATION ILLUSTRATION(PART 1)

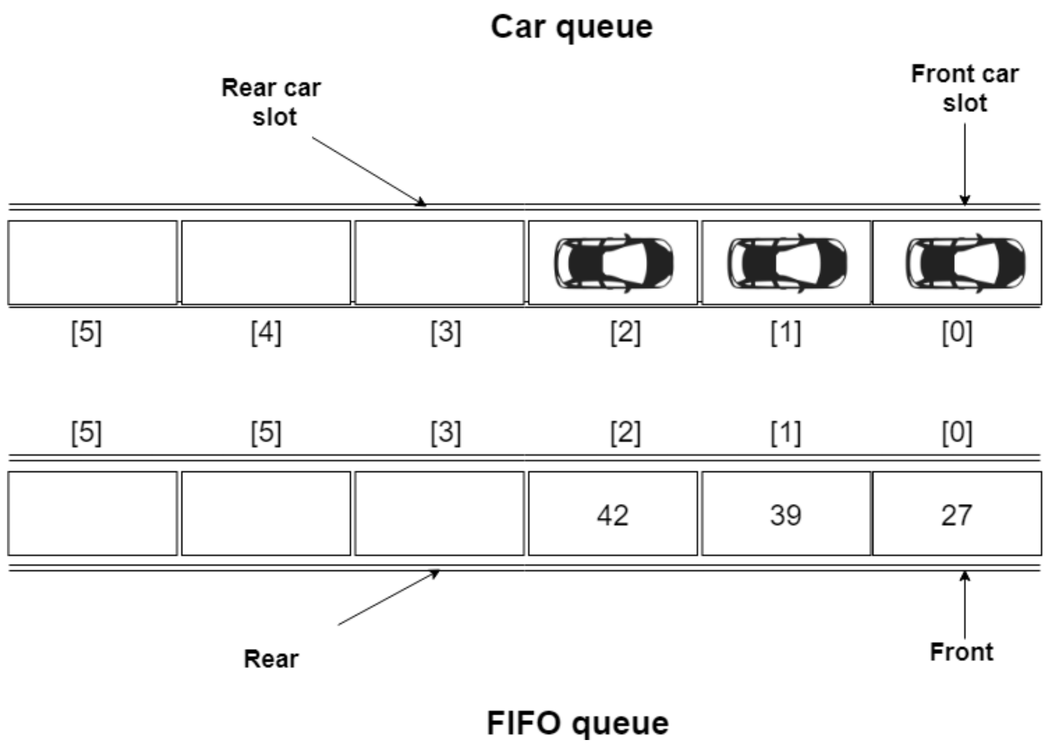


FIGURE 16. QUEUE ENQUEUE OPERATION ILLUSTRATION(PART 2)

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

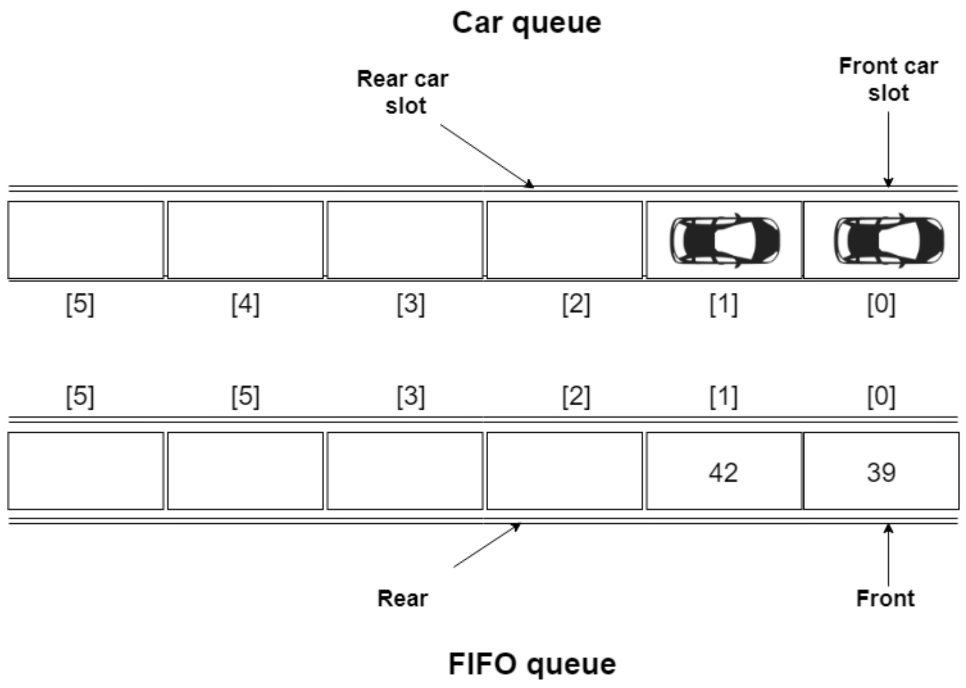


FIGURE 17. QUEUE DEQUEUE OPERATION ILLUSTRATION (1ST APPROACH)

In the 1st approach (**Figure 17**), 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 car queue, the first car in the queue go away, then every following up car moved forward to fill up the space the last car left.

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

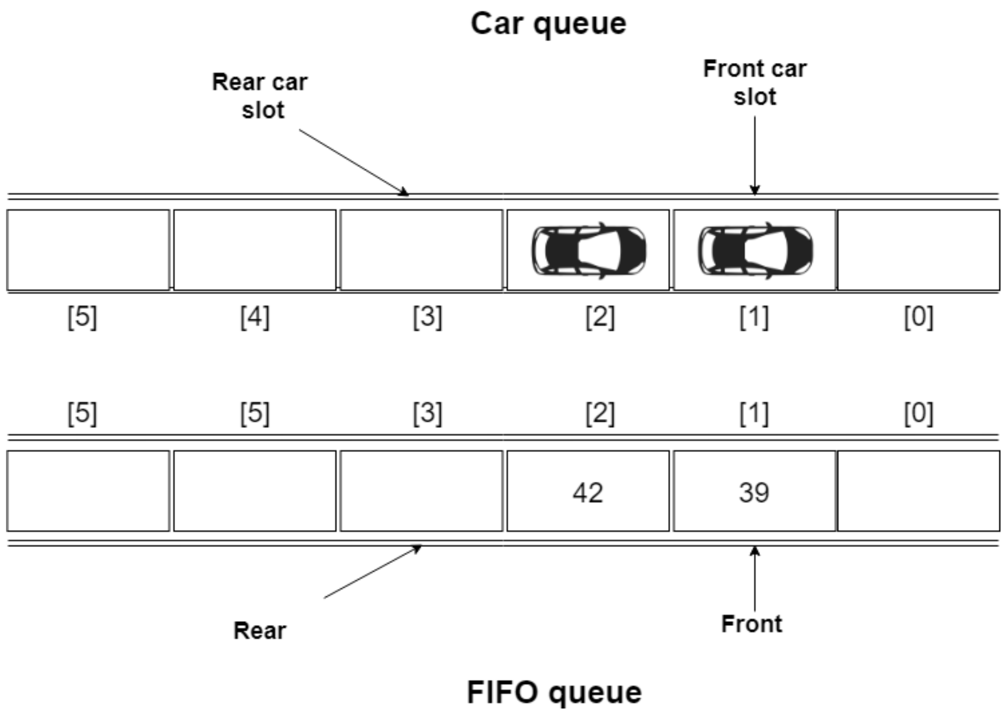


FIGURE 18. QUEUE DEQUEUE OPERATION ILLUSTRATION (2ND APPROACH)

In the 2nd approach, the element at **front** position got removed, and then the **front** move to the next position. Or to be compared to car queue, even if the first car in the queue left away, the following up next car will become the first-in-queue car, but not moving forward to fill up the space the last car 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.

5. PERFORMANCE COMPARISON OF TWO SORTING ALGORITHMS

A Sorting Algorithm is used to rearrange a given array or list elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of element in the respective data structure.

This report will compare two of the simplest sorting algorithms, which are: **insertion sort** and **selection sort**

5.1. INSERTION SORTING ALGORITHM

Insertion sort is a simple **sorting algorithm** that builds the final sorted array (or list) one item at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort. However, insertion sort provides several advantages:

- Simple implementation
- Efficient for (quite) small data sets, much like other quadratic sorting algorithms
- More efficient in practice than most other simple quadratic (i.e., O(n^2)) algorithms such as selection sort or bubble sort
- Adaptive, i.e., efficient for data sets that are already substantially sorted: the time complexity is O(*kn*) when each element in the input is no more than *k* places away from its sorted position
- Stable; i.e., does not change the relative order of elements with equal keys
- In-place; i.e., only requires a constant amount O(1) of additional memory space
- Online; i.e., can sort a list as it receives it

When people manually sort cards in a bridge hand, most use a method that is similar to insertion sort.

IMPLEMENTATION CODE IN C#

```
public static void InsertionSort(int[] k, int n)
        {
            for (int i = 0; i < n-1; i++)
            {
                for(int j = i + 1; j > 0; j--)
                {
                    if (k[j - 1] > k[j])
                    {
                        Swap(ref k[j - 1], ref k[j]);
                    }
                }
            }
        }
```

HOW IT WORKS

Insertion sort works by inserting the set of values in the existing sorted file. It constructs the sorted array by inserting a single element at a time. This process continues until whole array is sorted in some order. The primary concept behind insertion sort is to insert each item into its appropriate place in the final list. The insertion sort method saves an effective amount of memory.

Insertion sort works as following steps**:**

1. The first step involves the comparison of the element in question with its adjacent element.
2. And if at every comparison reveals that the element in question can be inserted at a particular position, then space is created for it by shifting the other elements one position to the right and inserting the element at the suitable position.
3. The above procedure is repeated until all the element in the array is at their apt position.

Let us now understand working with the following example:

Consider the following array: 25, 17, 31, 13, 2

First Iteration: Compare 25 with 17. The comparison shows 17< 25. Hence swap 17 and 25. The array now looks like:

17, 25, 31, 13, 2

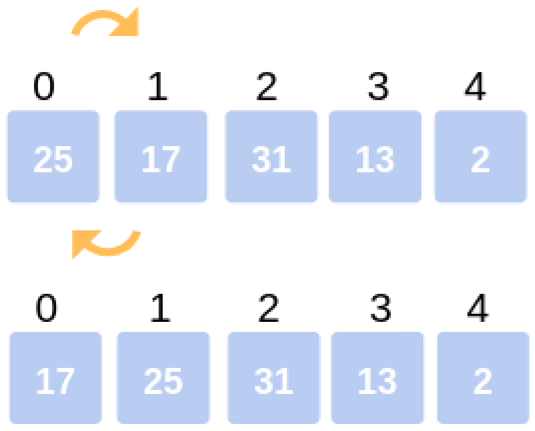


FIGURE 19. FIRST ITERATION OF INSERTION SORT

Second Iteration: Begin with the second element (25), but it was already swapped on for the correct position, so we move ahead to the next element.
Now hold on to the third element (31) and compare with the ones preceding it.

Since 31> 25, no swapping takes place.

Also, 31> 17, no swapping takes place and 31 remains at its position.

The array after the Second iteration looks like:

17, 25, 31, 13, 2

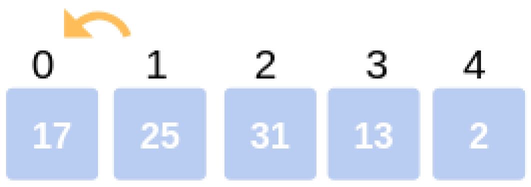


FIGURE 20.SECOND ITERATION OF INSERTION SORT

Third Iteration: Start the following Iteration with the fourth element (13), and compare it with its preceding elements.
Since 13< 31, we swap the two.

Array now becomes: 17, 25, 13, 31, 2.

But there still exist elements that we haven't yet compared with 13. Now the comparison takes place between 25 and 13. Since, 13 < 25, we swap the two.

The array becomes **17, 13, 25, 31, 2**.
The last comparison for the iteration is now between 17 and 13. Since 13 < 17, we swap the two.

The array now becomes **13, 17, 25, 31, 2**.

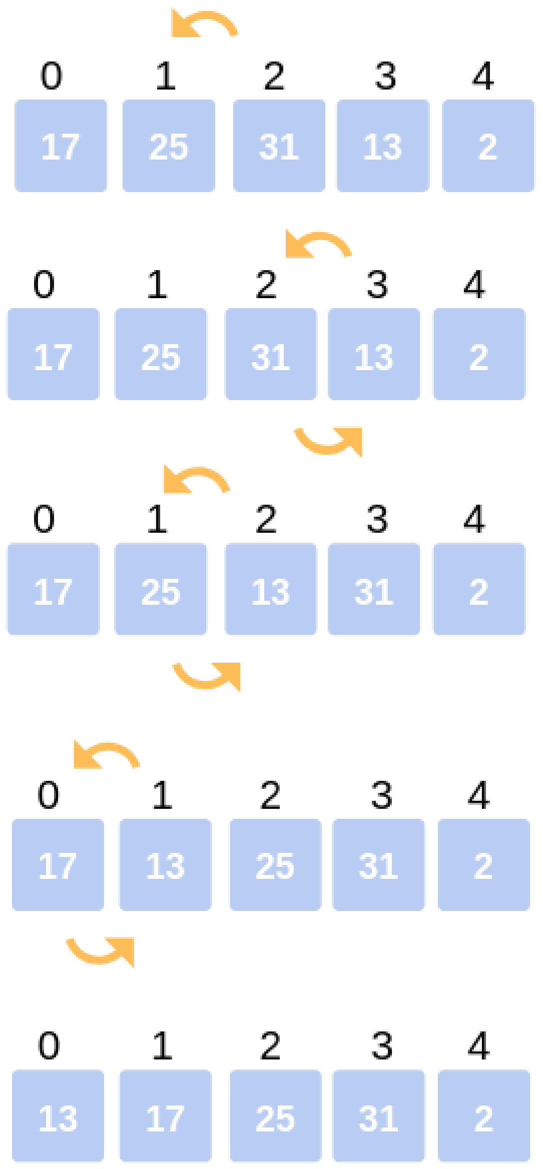


FIGURE 21. THIRD ITERATION OF INSERTION SORT

Fourth Iteration: The last iteration calls for the comparison of the last element (2), with all the preceding elements and make the appropriate swapping between elements.
Since, 2< 31. Swap 2 and 31.

Array now becomes: 13, 17, 25, 2, 31.

Compare 2 with 25, 17, 13.

Since, 2< 25. Swap 25 and 2.

13, 17, 2, 25, 31.
Compare 2 with 17 and 13.

Since, 2<17. Swap 2 and 17.

Array now becomes:

13, 2, 17, 25, 31.
The last comparison for the Iteration is to compare 2 with 13.

Since 2< 13. Swap 2 and 13.

The array now becomes:

2, 13, 17, 25, 31.
This is the final array after all the corresponding iterations and swapping of elements.

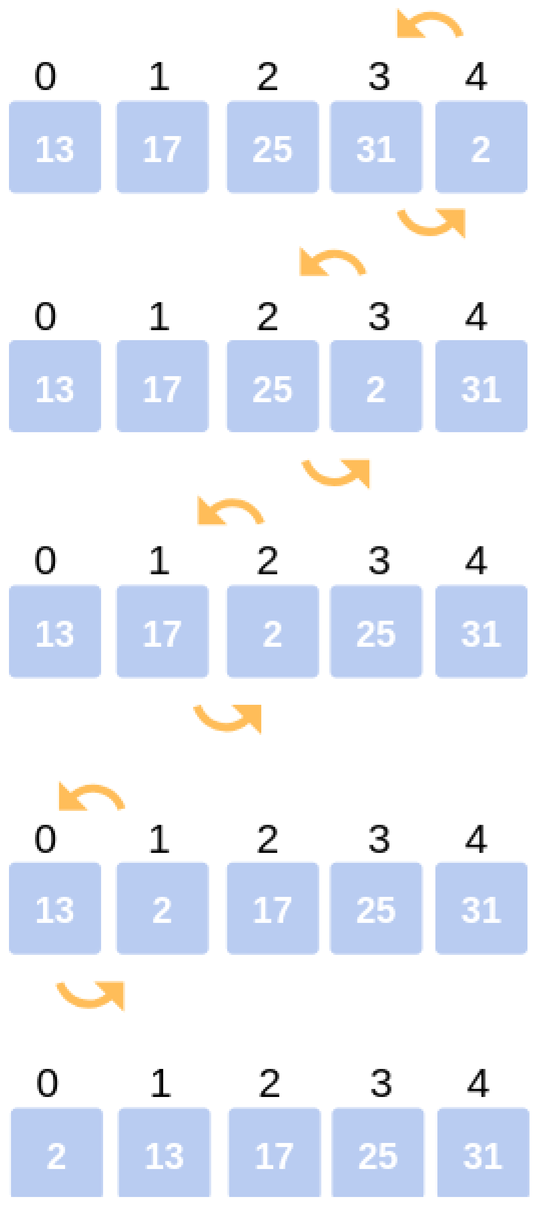


FIGURE 22. FOURTH ITERATION OF INSERTION SORT

COMPLEXITY

The best case input is an array that is already sorted. In this case insertion sort has a linear running time (i.e., O(*n*)). During each iteration, the first remaining element of the input is only compared with the right-most element of the sorted subsection of the array.

The simplest worst case input is an array sorted in reverse order. The set of all worst case inputs consists of all arrays where each element is the smallest or second-smallest of the elements before it. In these cases, every iteration of the inner loop will scan and shift the entire sorted subsection of the array before inserting the next element. This gives insertion sort a quadratic running time (i.e., O(n^2)).

The average case is also quadratic, which makes insertion sort impractical for sorting large arrays.

However, insertion sort is one of the fastest algorithms for sorting very small arrays, even faster than quicksort; indeed, good quicksort implementations use insertion sort for arrays smaller than a certain threshold, also when arising as subproblems; the exact threshold must be determined experimentally and depends on the machine, but is commonly around ten.

5.2. SELECTION SORT

Selection sort is a **sorting algorithm**, specifically an in-place **comparison sort**.

Selection sort is easy to implement and has performance advantages over more complicated algorithms in certain situations, however, requires *O(N^2)* comparisons and so it should only be used on small files.

IMPLEMENTATION CODE IN C#

```
public static void SelectionShort(int[] k, int n)
        {
            int i;
            for (i = 0; i < n - 1; i++)
            {
                int m = i;
                for (int j = i + 1; j < n; j++)
                {
                    if (k[j] < k[m])
                    {
                        m = j;
                    }
                }
                if (m != i)
                {
                    Swap(ref k[i], ref k[m]);
                }
            }
        }
```

HOW IT WORKS

The algorithm divides the input list into two parts:

- the sub-list of items already sorted (initially empty), which is built up from left to right at the front (or left) of the list
- the sub-list of items remaining to be sorted that occupy the rest of the list (initially the entire input list)

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.

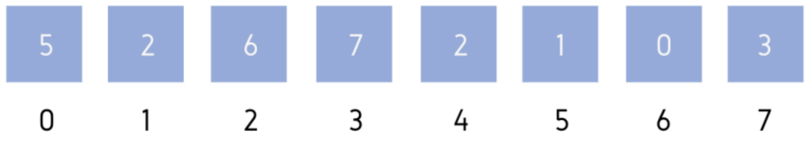
The following example will visualize this selection sort algorithm:

🟦 Part of unsorted array

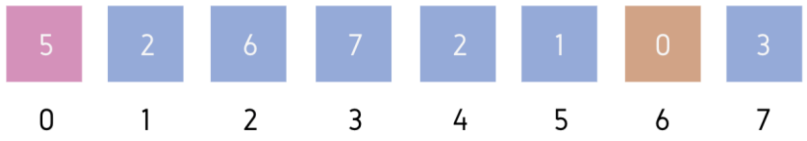
🟩 Part of sorted array

🟪 Leftmost element in unsorted array

🟫 Minimum element in unsorted array



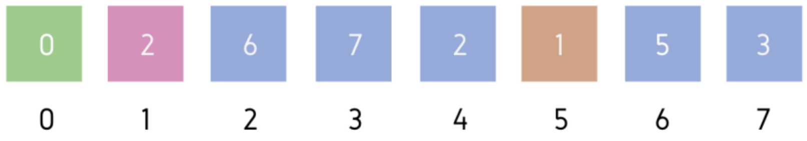
This is the example initial array A = [5, 2, 6, 7, 2, 1, 0, 3]



Leftmost element of unsorted part = A[0]

Minimum element of unsorted part = A[6]

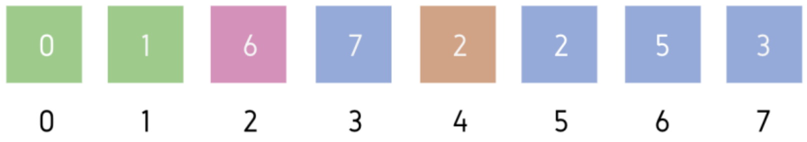
A[0] and A[6] will be swapped. Then, A[0] became a part of sorted subarray.



Leftmost element of unsorted part = A[1]

Minimum element of unsorted part = A[5]

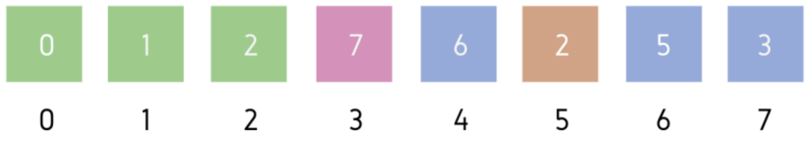
A[1] and A[5] will be swapped. Then, A[1] became a part of sorted subarray.



Leftmost element of unsorted part = A[2]

Minimum element of unsorted part = A[4]

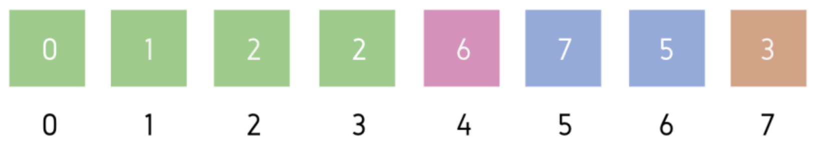
A[2] and A[4] will be swapped. Then, A[2] became a part of sorted subarray.



Leftmost element of unsorted part = A[3]

Minimum element of unsorted part = A[5]

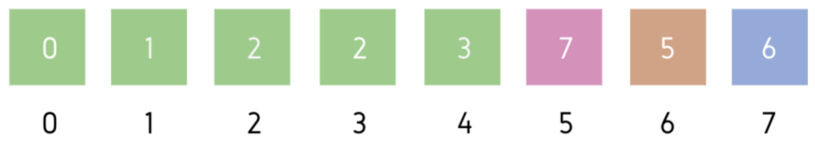
A[3] and A[5] will be swapped. Then, A[3] became a part of sorted subarray.



Leftmost element of unsorted part = A[4]

Minimum element of unsorted part = A[7]

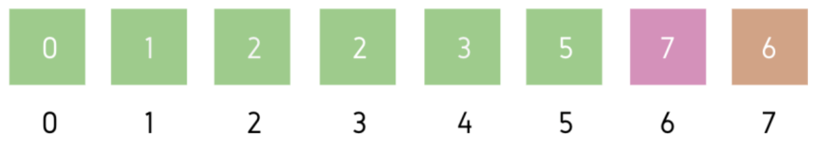
A[4] and A[7] will be swapped. Then, A[4] became a part of sorted subarray.



Leftmost element of unsorted part = A[5]

Minimum element of unsorted part = A[6]

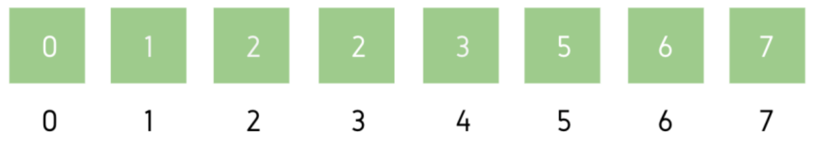
A[5] and A[6] will be swapped. Then, A[5] became a part of sorted subarray.



Leftmost element of unsorted part = A[6]

Minimum element of unsorted part = A[7]

A[6] and A[7] will be swapped. Then, A[6] became a part of sorted subarray.



This is the final sorted array.

COMPLEXITY

As the working of selection, sort does not depend on the original order of the elements in the array, so there is not much difference between best case and worst case complexity of selection sort.

The selection sort selects the minimum value element, in the selection process all the 'n' number of elements are scanned; therefore n-1 comparisons are made in the first pass. Then, the elements are interchanged. Similarly, in the second pass also to find the second smallest element we require scanning of rest n-1 elements and the process is continued till the whole array sorted.

Thus, running time complexity of selection sort is **$O(n^2)$**.
=(n-1)+(n-2)+………..+2+1
= n(n-1)/2 = $O(n^2)$

5.3. COMPARISION TABLE

BASIS FOR COMPARISON	INSERTION SORT	SELECTION SORT
Basic	The data is sorted by inserting the data into an existing sorted file.	The data is sorted by selecting and placing the consecutive elements in sorted location.
Nature	Stable	Unstable
Processes to be followed	Elements are known beforehand while location to place them is searched.	Location is previously known while elements are searched.
Immediate data	Insertion sort is live sorting technique which can deal with immediate data.	Unable to deal with immediate data, it needs to be present at the beginning.
Average case complexity	$O(n^2)$	$O(n^2)$
Method	Insertion	Selection

6. ADVANTAGES OF ENCAPSULATION AND INFORMATION HIDING WHEN USING AN ADT

In computer science, an **Abstract Data Type** (**ADT**) is a mathematical model for data types, where a data type is defined by its behavior (semantics) from the point of view of a *user* of the data, specifically in terms of possible values, possible operations on data of this type, and the behavior of these operations.

An **ADT** defines a class of abstract object which is completely characterized by the operations available on those objects. This means that an **ADT** can be defined by defining the characterizing operations for that type, therefore, captures the fundamental properties of abstract objects.

When a programmer makes use of an **ADT** object, he is concerned only with the behavior which that object exhibits but not with any details of how that behavior is achieved by means of an implementation. Implementation information, such as how the object is represented in storage, is only needed when defining how the characterizing operations are to be implemented. The user of the object is not required to know or supply this information. By providing this level of abstraction, implement **ADT**s creates an **encapsulation** around the data. The idea is that by encapsulating the details of the implementation, everything would be hidden from the user's view. This is called **information hiding**.

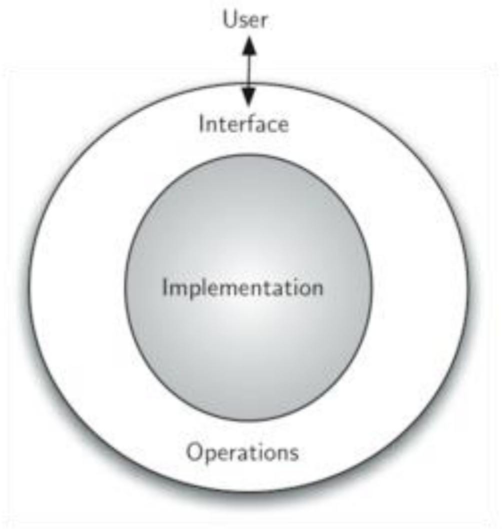


FIGURE 23. ABSTRACT DATA TYPE VISUALIZATION (RUNESTONE.ACADEMY, 2018)

Figure 23. shows a picture of what an **ADT** is and how it operates. The user only interacts with the interface, using the operations that have been specified by the **ADT**. The **ADT** is the shell that the user interacts with. The implementation is hidden one level deeper and the user is not concerned with the details of the implementation.

The implementation of an **ADT** that often referred to as a **data structure**, will require a provision of a physical view of the data using some collection of programming constructs and primitive data types. As mentioned earlier, the separation of these two perspectives will allows to define the complex data models for different problems without giving any indication as to the details of how the model will actually be built.

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.

7. NETWORK SHORTEST PATH ALGORITHMS

7.1. DIJKSTRA ALGORITHM

Dijkstra algorithm is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks, conceived by computer scientist Edsger W. Dijkstra in 1956 and published three years later.

In terms of networking and shortest path finding, **Dijkstra algorithm** finds the shortest path from a given node *s*, known as a **starting node** or an **initial node**, to all other nodes in the network, by assigning some initial values for the distances from node *s* and to every other node in the network.

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**.

DIJKSTRA ALGORITHM OPERATION

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

1. Mark all nodes unvisited. Create a set of all the unvisited nodes called the *unvisited set*.
2. 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
3. For the current node, consider all of its unvisited neighbours 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.
4. When we are done considering all of the unvisited neighbours 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.
5. 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 *unvisited set* is infinity (when planning a complete traversal; occurs when there is no connection between the initial node and remaining unvisited nodes), then stop. The algorithm has finished.
6. Otherwise, select the unvisited node that is marked with the smallest tentative distance, set it as the new "current node", and go back to step 3.

When planning a route, it is actually not necessary to wait until the destination node is "visited" as above: the algorithm can stop once the destination node has the smallest tentative distance among all "unvisited" nodes (and thus could be selected as the next "current").

DIJKSTRA ALGORITHM EXAMPLE

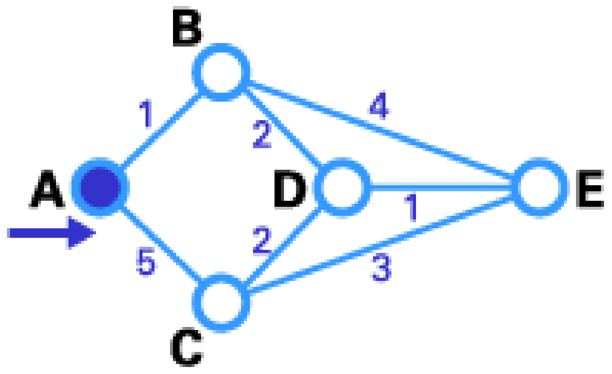


FIGURE 24. DIJKSTRA ALGORITHM EXAMPLE (1)

At this example, the best route between A and E will be determined by using **Dijkstra algorithm**.

	A	**B**	**C**	**D**	**E**
A	0	1	1	0	0
B	1	0	0	1	1
C	1	0	0	1	1
D	0	1	1	0	1
E	0	1	1	1	0

Figure 24's adjacent matrix

	A	**B**	**C**	**D**	**E**
A	0	1	5	0	0
B	1	0	0	2	4
C	5	0	0	2	3
D	0	2	2	0	1
E	0	4	3	1	0

Figure 24's weight matrix

Step 1: As in **Figure 24**, the *s* node (A) has been chosen as T-node (**temporary** node), and so its label is **permanent** (**permanent** nodes are the nodes with filled circles and T-nodes with the → symbol)

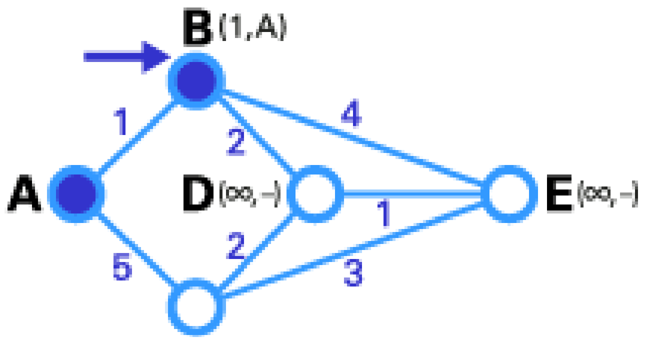


Figure 25.DIJKSTRA ALGORITHM EXAMPLE (2)

Step 2: In this step, the status record set of tentative nodes directly linked to T-node (B-C) has been changed. Since B has less weight (1 compared to 5 of C), it has been chosen as T-node and its label has changed to permanent.

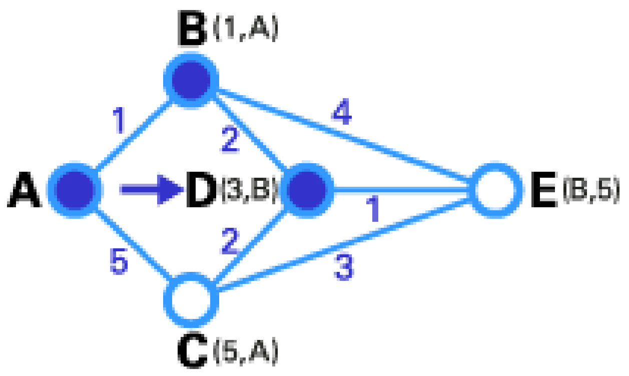


FIGURE 26.DIJKSTRA ALGORITHM EXAMPLE (3)

Step 3: In this step, just like the step 2, the status record set of tentative nodes that have a direct link to T-node (D-E) has been changed. Also, since D has less weight, it has been chosen as T-node and its label has changed to permanent.

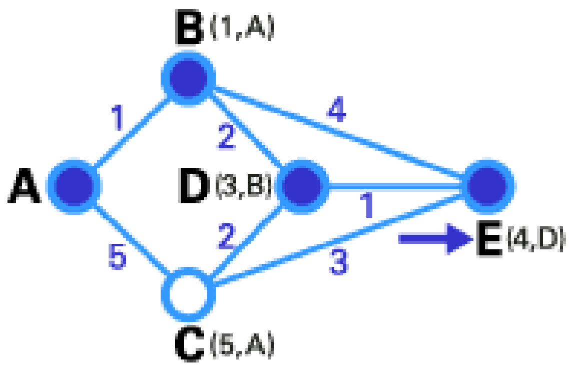


FIGURE 27.DIJKSTRA ALGORITHM EXAMPLE (4)

Step 4: In this last step, there is none further tentative nodes, so E, which having the least weight, has been chosen as T-node. Lastly, E is the destination, so the algorithm stop here.

Results: after using **Dijkstra** algorithm, the shortest path from A to E is A → B → D → E with the total of weight is 4

7.2. BELLMAN FORD ALGORITHM

Bellman-Ford algorithm, also known as **DV algorithm** and **Ford-Fulkerson 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 Dijstra's algorithm in solving the same problem, but more versatile, as it is capable of handling graphs in which some of the edge weights are negative numbers.

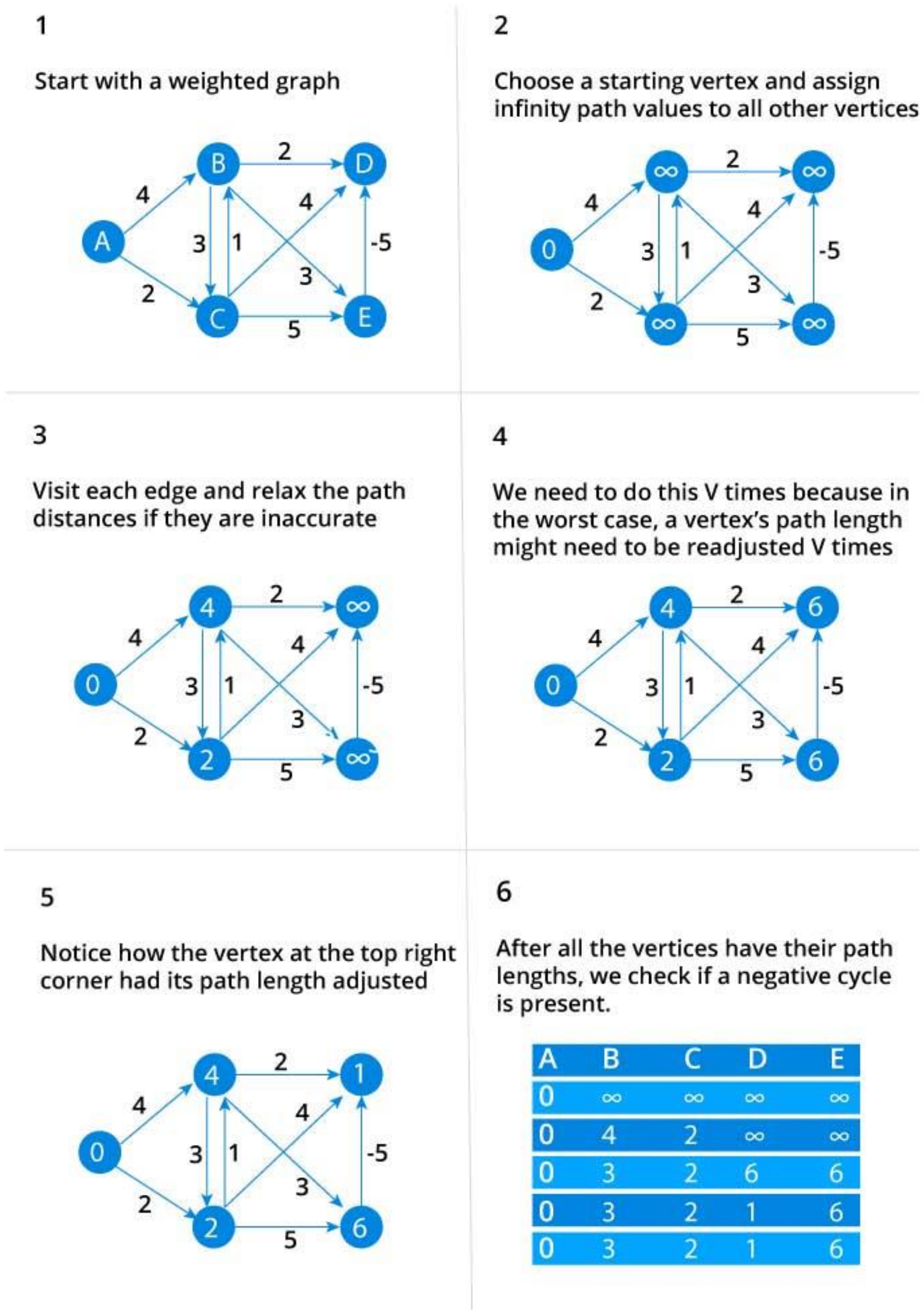
BELLMAN-FORD ALGORITHM OPERATION

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 ALGORITHM EXAMPLE


1

Start with a weighted graph

2

Choose a starting vertex and assign infinity path values to all other vertices

3

Visit each edge and relax the path distances if they are inaccurate

4

We need to do this V times because in the worst case, a vertex's path length might need to be readjusted V times

5

Notice how the vertex at the top right corner had its path length adjusted

6

After all the vertices have their path lengths, we check if a negative cycle is present.

A	B	C	D	E
0	∞	∞	∞	∞
0	4	2	∞	∞
0	3	2	6	6
0	3	2	1	6
0	3	2	1	6

FIGURE 28. BELLMAN FORD ALGORITHM EXAMPLE (PROGRAMIZ.COM, 2018)

	A	B	C	D	E
A	0	1	1	0	0
B	0	0	1	1	1
C	0	1	0	1	1
D	0	0	0	0	0
E	0	0	0	1	0

Figure 28's adjacent matrix

	A	B	C	D	E
A	0	4	2	0	0
B	0	0	3	2	3
C	0	1	0	4	5
D	0	0	0	0	0
E	0	0	0	-5	0

Figure 28's weight matrix

In order to find the shortest path from A to D, after implementing **Bellman-Ford** algorithm, the results is: A → C → E → D with weight values is 2.

8. DISCUSSION ON THE VIEW THAT IMPERATIVE ADT ARE A BASIS FOR OBJECT ORIENTATION

According to the research by (Cook, 1990) , **Abstract data types** (**ADTs**) are often called *user-defined data types*, because they allow programmers to define new types that resemble primitive data types, for example, just like a primitive type *Integer* with operations such as +,-,*, etc., an **ADT** has a type domain, whose representation is unknown to clients, and a set of operations defined on the domain. Inheritance is not part of the concept. Parametric polymorphism and encapsulation are required, but there is no concept of and *object* containing its own operations. The semantics of the operations are an essential part of the definition of an **ADT** and not alterable.

Object-oriented programming involves the construction of *objects* which have a collection of methods or procedures that share access to private local *state*. The term ”*object* ” is not very descriptive of the use of collections of procedures to implement a data abstraction, it is essentially defined by the names of the messages it receives and the values it encapsulates. **Object-orientation** should be considered as a technique that uses procedures as abstract data.

It is argued that **ADTs** and **Object-orientation** 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.

The essence of **Object-oriented programming** is *procedural data abstraction*, in which procedures are used to represent data and procedural interfaces provide information hiding and abstraction. This technique is complementary to ADTs, in which concrete algebras are used to represent data, and type abstraction provides information hiding.

In conclusion, as in my opinion, **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.

REFERENCES

.

Cook, W. R. (1990). Object-Oriented Programming Versus Abstract Data Types. *REX Workshop/School on the Foundations of Object-Oriented*, 151-178.

javatpoint.com. (2018). *Linked list implementation of stack*. Retrieved from javatpoint.com: https://www.javatpoint.com/ds-linked-list-implementation-of-stack

Programiz.com. (2018). *Bell-man Ford Algorithm*. Retrieved from Programmiz: https://www.programiz.com/dsa/bellman-ford-algorithm

Runestone.Academy. (2018). *Why study data structures and abstract data types?* Retrieved from Runestone Academy: https://runestone.academy/runestone/books/published/pythonds/Introduction/WhyStudyDataStructuresandAbstractDataTypes.html

TechDifferences. (2017, august 1). *Difference between Array and Linked List*. Retrieved from TechDifferences: https://techdifferences.com/difference-between-array-and-linked-list.html

Tutorialspoint. (2018). *Data structure and Algorithms*. Retrieved from Tutorials Point: https://www.tutorialspoint.com/data_structures_algorithms/dsa_queue

Tutorialspoint.com. (2018). *Data structure and algorithms- Queue*. Retrieved from tutorialspoint.com: https://www.tutorialspoint.com/data_structures_algorithms/dsa_queue

wikipedia.org. (2018). *Call stack*. Retrieved from Wikipedia.org: https://en.wikipedia.org/wiki/Call_stack

wikipedia.org. (2018). *Dijkstra algorithm*. Retrieved from Wikipedia.org: https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

HowStuffWorks. 2019. Example: Dijkstra Algorithm - How Routing Algorithms Work | HowStuffWorks. [ONLINE] Available at: https://computer.howstuffworks.com/routing-algorithm3.htm. [Accessed 08 August 2019].