



# CITY UNIVERSITY

Department of Computer Science & Engineering



# DATA STRUCTURE

## SUBMITTED BY

Md. Golam Rabbani  
Id: 171442511  
Batch: 44<sup>th</sup>  
CITY UNIVERSITY

## SUBMITTED TO

Richard Philip  
Lecturer  
CSE Department  
CITY UNIVERSITY



*Md. Golam Rabbani*



COMPUTER SCIENCE

# Contents

<b>ARRAY</b>	3
One Dimentional Array	4
Two Dimentional Array	5
<b>STACK</b>	6
Stack Representation	6
Basic Operations	7
peek()	7
isfull()	8
isempty()	9
Push Operation	10
Algorithm for PUSH Operation	11
Pop Operation	12
<b>QUEUE</b>	13
Queue Representation	13
Basic Operations	14
<b>LINKED LIST</b>	14
Linked List Representation	15
Types of Linked List	15
Basic Operations	16
<b>BREADTH FIRST SEARCH</b>	16
<b>DEPTH FIRST SEARCH</b>	19
<b>REFERENCES</b>	22

# ARRAY

An array is a collection of one or more values of the same type. Each value is called an element of the array. The elements of the array share the same variable name but each element has its own unique index number (also known as a subscript). An array can be of any type, For example: `int`, `float`, `char` etc. If an array is of type `int` then its elements must be of type `int` only.

To store roll no. of 100 students, we have to declare an array of size 100 i.e `roll_no[100]`. Here size of the array is 100, so it is capable of storing 100 values. In C, index or subscript starts from 0, so `roll_no[0]` is the first element, `roll_no[1]` is the second element and so on. Note that last element of the array will be at `roll_no[99]` not at `roll_no[100]` because index starts at 0.

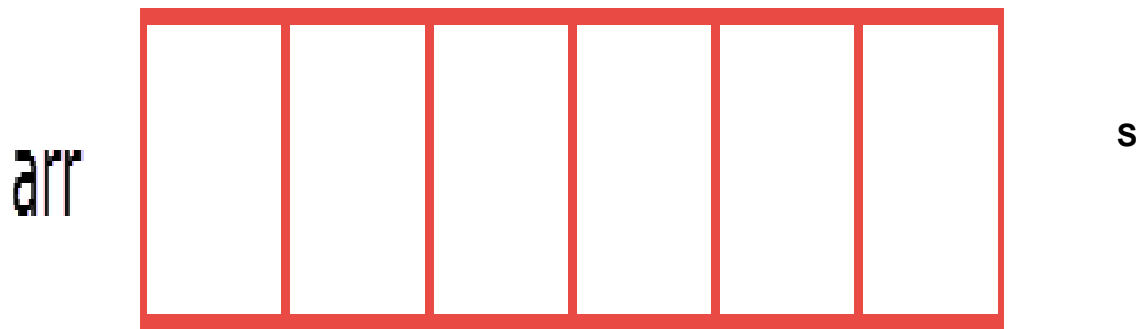


Arrays can be single or multidimensional. The number of subscript or index determines dimensions of the array. An array of one dimension is known as a one-dimensional array or 1-D array, while an array of two dimensions is known as a two-dimensional array or 2-D array.

(Overlq.com, 2017).

## One Dimensional Array

Conceptually you can think of a one-dimensional array as a row, where elements are stored one after another.



*an array of 6 elements*

---



**syntax:** `datatype array_name[size];`

**datatype:** It denotes the type of the elements in the array.

**array\_name:** it is the name given to an array. It must be any valid identifier.

**size:** It is the number of elements an array can hold.

(Overlq.com, 2017).

## Two Dimensional Array

The syntax declaration of 2-D array is not much different from 1-D array. In 2-D array, to declare and access elements of a 2-D array we use 2 subscripts instead of 1.

**Syntax:** `datatype array_name[ROW][COL];`

The total number of elements in a 2-D array is `ROW*COL`. Let's take an example.

```
int arr[2][3];
```

This array can store `2*3=6` elements. You can visualize this 2-D array as a matrix of 2 rows and 3 columns.

The individual elements of the above array can be accessed by using two subscript instead

		0	1	2	
	0	arr[0][0]	arr[0][1]	arr[0][2]	Row 0
	1	arr[1][0]	arr[1][1]	arr[1][2]	Row 1
		Col 0	Col 1	Col 2	

*A conceptual representation of 2-D array*

TheCguru.com

of one. The first subscript denotes row number and second denotes column number. As we can see in the above image both rows and columns are indexed from 0. So the first element of this array is at `arr[0][0]` and the last element is at `arr[1][2]`. Here are how you can access all the other elements:

`arr[0][0]` - refers to the first element `arr[0][1]` - refers to the second

element `arr[0][2]` - refers to the third element `arr[1][0]` - refers to the fourth

element `arr[1][1]` - refers to the fifth element `arr[1][2]` - refers to the sixth element

If you try to access an element beyond valid `ROW` and `COL`, C compiler will not display any kind of error message, instead, a garbage value will be printed. It is the responsibility of the programmer to handle the bounds.

`arr[1][3]` - a garbage value will be printed, because the last valid index

of `COL` is 2 `arr[2][3]` - a garbage value will be printed, because the last valid index

of `ROW` and `COL` is 1 and 2 respectively

Just like 1-D arrays, we can only also use constants and symbolic constants to specify the size of a 2-D array

(Overlq.com, 2017).

# STACK

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, etc.

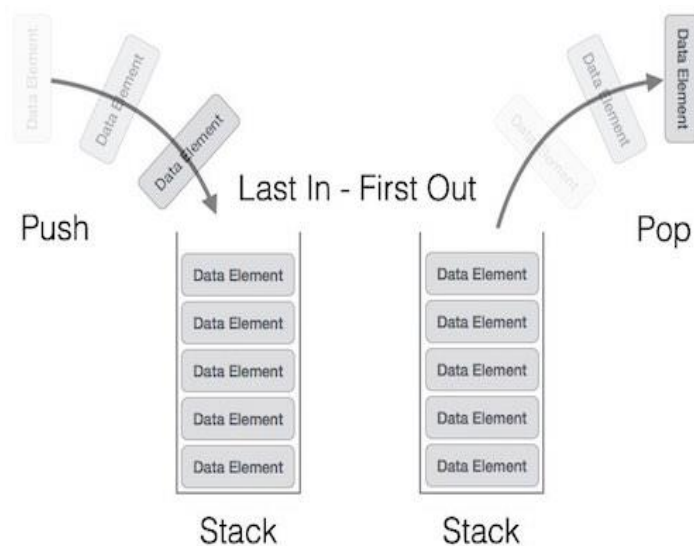


A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.

This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.

([tutorialsPoint, 2018](#)).

## Stack Representation



The following diagram depicts a stack and its operations – A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.

([tutorialsPoint, 2018](#)).

## Basic Operations

Stack operations may 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.

When data is PUSHed onto stack.

To use a 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.

At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer always represents the top of the stack, hence named **top**. The **top** pointer provides top value of the stack without actually removing it.

First we should learn about procedures to support stack functions –

### **peek()**

Algorithm of peek() function –

```
begin procedure peek
    return stack[top]
end procedure
```

Implementation of peek() function in C programming language –

## Example

```
int peek() {  
    return stack[top];  
}
```

## isfull()

Algorithm of isfull() function –

```
begin procedure isfull  
  
    if top equals to MAXSIZE  
        return true  
    else  
        return false  
    endif  
  
end procedure
```

Implementation of isfull() function in C programming language –

## Example

```
bool isfull() {  
    if(top == MAXSIZE)  
        return true;  
    else  
        return false;  
}
```



## isempty()

Algorithm of isempty() function –

```
begin procedure isempty

    if top less than 1
        return true
    else
        return false
    endif

end procedure
```

Implementation of isempty() function in C programming language is slightly different. We initialize top at -1, as the index in array starts from 0. So we check if the top is below zero or -1 to determine if the stack is empty. Here's the code –

### Example

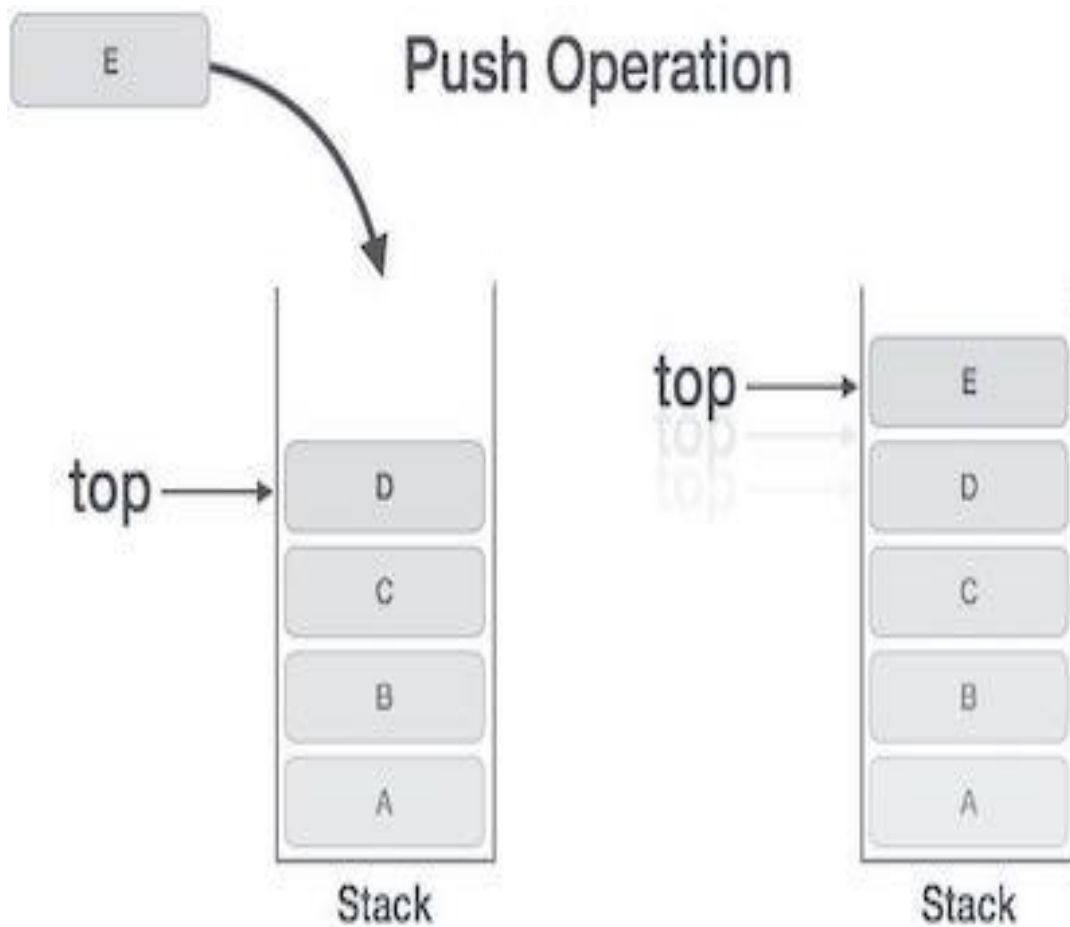
```
bool isempty() {
    if(top == -1)
        return true;
    else
        return false;
}
```

(tutorialsPoint, 2018).

## Push Operation

The process of putting a new data element onto stack is known as a 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.



If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.

## Algorithm for PUSH Operation

A simple algorithm for Push operation can be derived as follows –

```
begin procedure push: stack, data

    if stack is full
        return null
    endif

    top ← top + 1
    stack[top] ← data

end procedure
```

Implementation of this algorithm in C, is very easy. See the following code –

### Example

```
void push(int data) {
    if(!isFull()) {
        top = top + 1;
        stack[top] = data;
    } else {
        printf("Could not insert data, Stack is full.\n");
    }
}
```

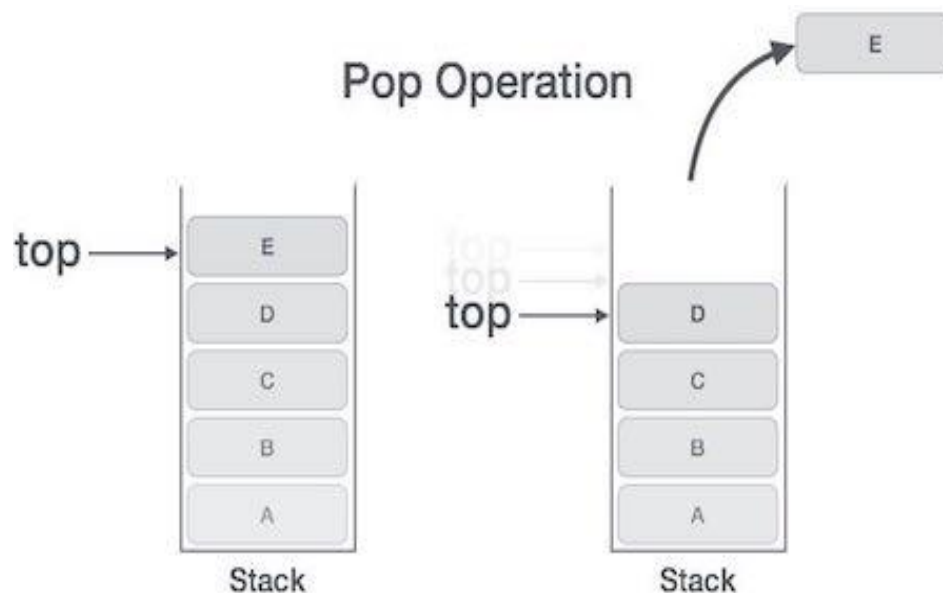
(tutorialsPoint, 2018).

## Pop Operation

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



(tutorialsPoint, 2018).

# 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.



A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops.

([tutorialsPoint, 2018](#)).

## Queue Representation

As we now understand that in queue, we access both ends for different reasons. The following diagram given below tries to explain queue representation as data structure –



As in stacks, a queue can also be implemented using Arrays, Linked-lists, Pointers and Structures. For the sake of simplicity, we shall implement queues using one-dimensional array.

([tutorialsPoint, 2018](#)).

## Basic Operations

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues –

- **enqueue()** – add (store) 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. These are –

- **peek()** – Gets the element at the front of the queue without removing it.
- **isfull()** – Checks if the queue is full.
- **isempty()** – Checks if the queue is empty.

In queue, we always dequeue (or access) data, pointed by **front** pointer and while enqueueing (or storing) data in the queue we take help of **rear** pointer.

(tutorialsPoint, 2018).

## LINKED LIST

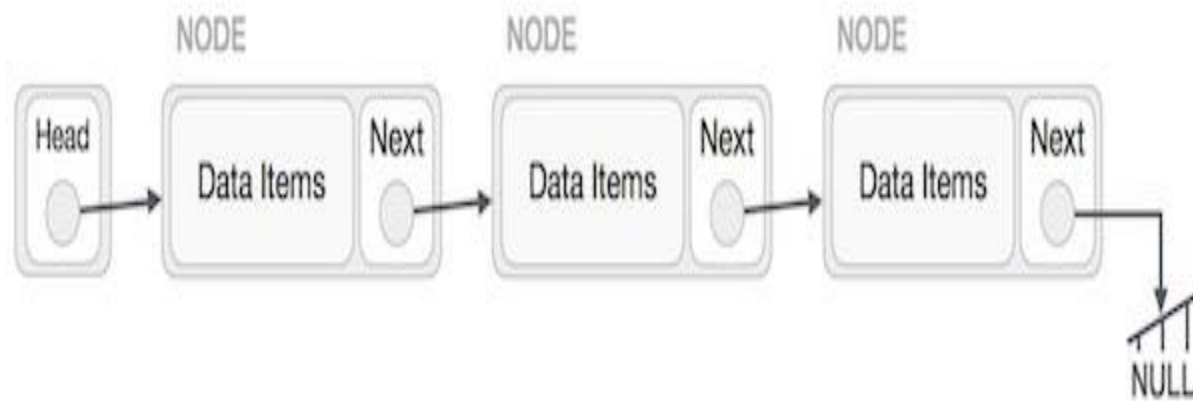
A linked list is a sequence of data structures, which are connected together via links. Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.

- **Link** – Each link of a linked list can store a data called an element.
- **Next** – Each link of a linked list contains a link to the next link called Next.
- **LinkedList** – A Linked List contains the connection link to the first link called First.

(tutorialsPoint, 2018).

## Linked List Representation

Linked list can be visualized as a chain of nodes, where every node points to the next node.



As per the above illustration, following are the important points to be considered.

- Linked List contains a link element called first.
- Each link carries a data field(s) and a link field called next.
- Each link is linked with its next link using its next link.
- Last link carries a link as null to mark the end of the list.

([tutorialsPoint, 2018](#)).

## Types of Linked List

Following are the various types of linked list.

- **Simple Linked List** – Item navigation is forward only.
- **Doubly Linked List** – Items can be navigated forward and backward.
- **Circular Linked List** – Last item contains link of the first element as next and the first element has a link to the last element as previous.

([tutorialsPoint, 2018](#)).

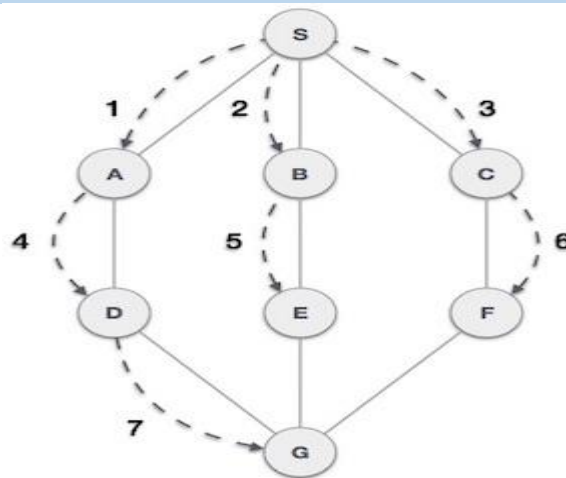
## Basic Operations

Following are the basic operations supported by a list.

- **Insertion** – Adds an element at the beginning of the list.
- **Deletion** – Deletes an element at the beginning of the list.
- **Display** – Displays the complete list.
- **Search** – Searches an element using the given key.
- **Delete** – Deletes an element using the given key.

(tutorialsPoint, 2018).

## BREADTH FIRST SEARCH

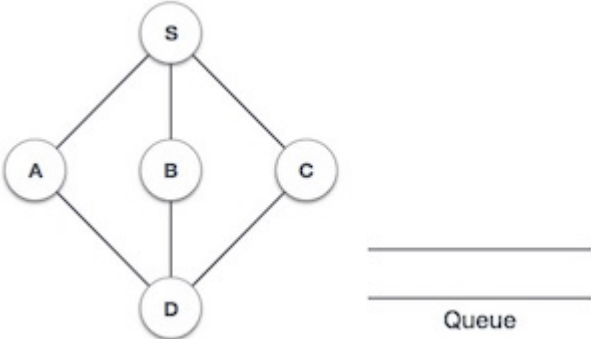
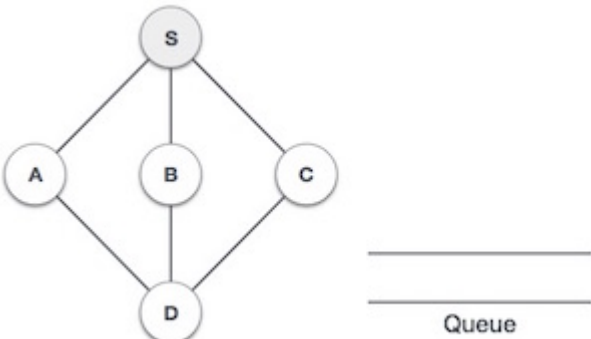
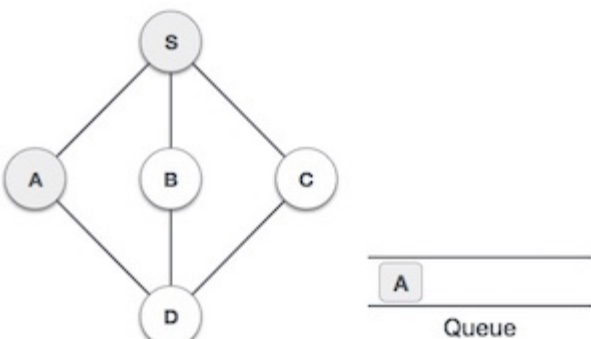


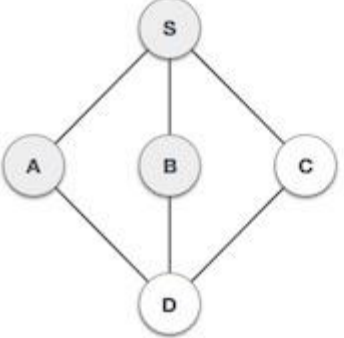
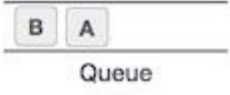
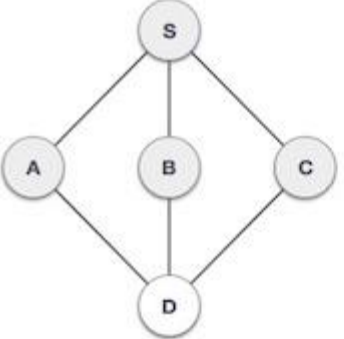
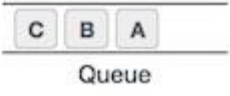
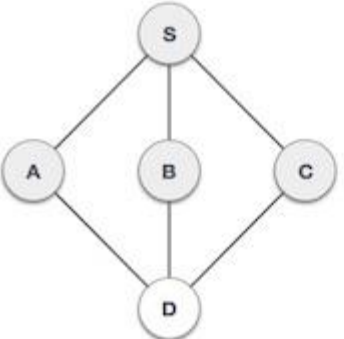
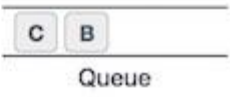
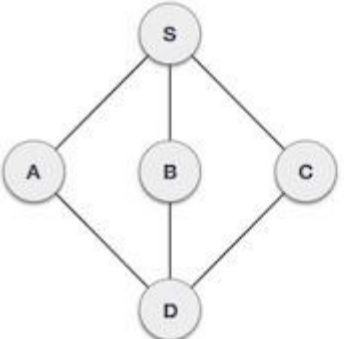
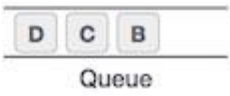
Breadth First Search (BFS) algorithm traverses a graph in a breadthward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

As in the example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs the following rules.

- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
- **Rule 2** – If no adjacent vertex is found, remove the first vertex from the queue.
- **Rule 3** – Repeat Rule 1 and Rule 2 until the queue is empty.



Step	Traversal	Description
1		Initialize the queue.
2		We start from visiting <b>S</b> (starting node), and mark it as visited.
3		We then see an unvisited adjacent node from <b>S</b> . In this example, we have three nodes but alphabetically we choose <b>A</b> , mark it as visited and enqueue it.

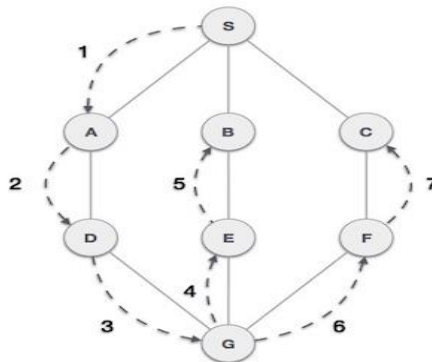
4	 	<p>Next, the unvisited adjacent node from <b>S</b> is <b>B</b>. We mark it as visited and enqueue it.</p>
5	 	<p>Next, the unvisited adjacent node from <b>S</b> is <b>C</b>. We mark it as visited and enqueue it.</p>
6	 	<p>Now, <b>S</b> is left with no unvisited adjacent nodes. So, we dequeue and find <b>A</b>.</p>
7	 	<p>From <b>A</b> we have <b>D</b> as unvisited adjacent node. We mark it as visited and enqueue it.</p>

At this stage, we are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied, the program is over.

([tutorialsPoint, 2018](#)).

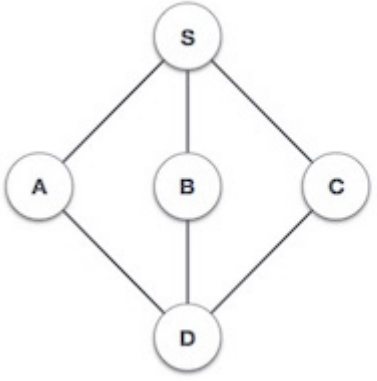

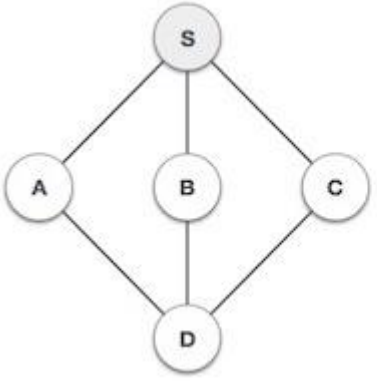

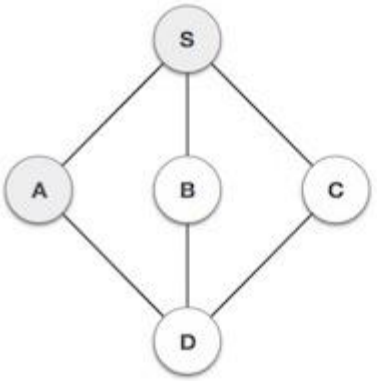
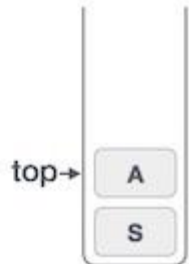
## DEPTH FIRST SEARCH

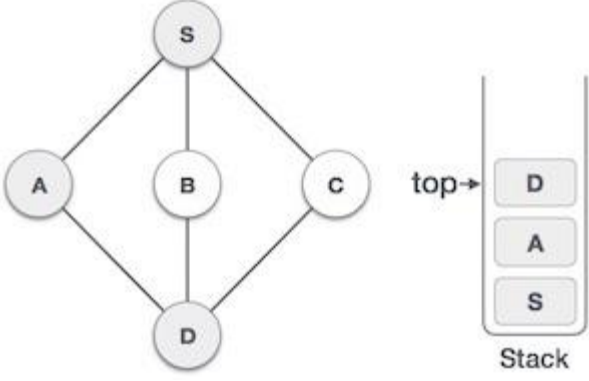
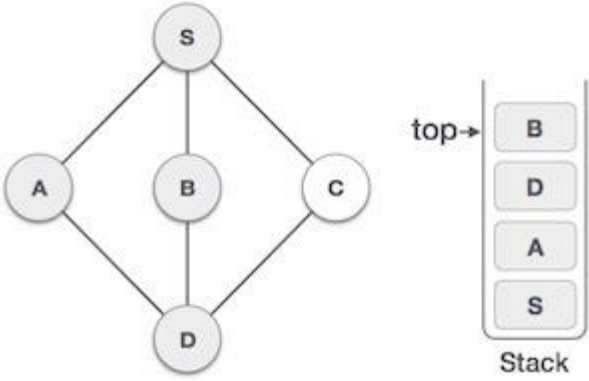
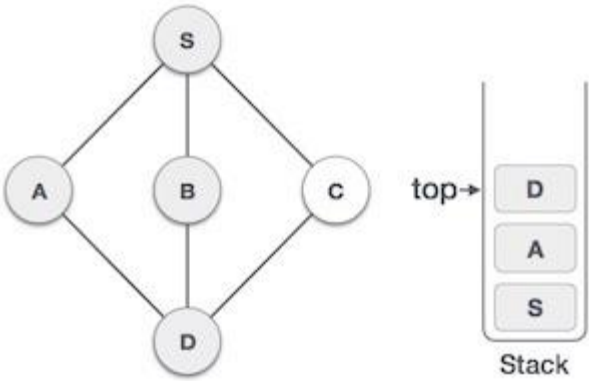
Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



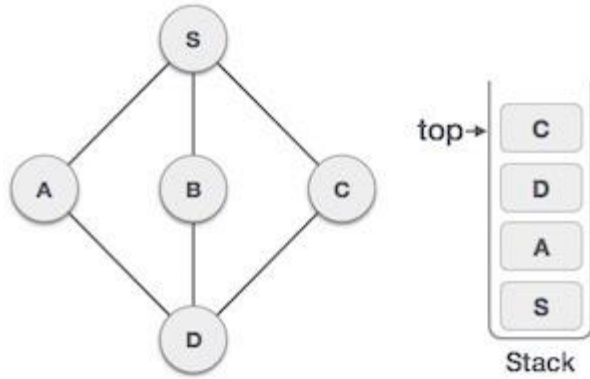
As in the example given above, DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to C. It employs the following rules.

- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
- **Rule 2** – If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)
- **Rule 3** – Repeat Rule 1 and Rule 2 until the stack is empty.

Step	Traversal	Description
1	  <p>Stack</p>	Initialize the stack.
2	  <p>Stack</p>	Mark <b>S</b> as visited and put it onto the stack. Explore any unvisited adjacent node from <b>S</b> . We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.
3	  <p>Stack</p>	Mark <b>A</b> as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both <b>S</b> and <b>D</b> are adjacent to <b>A</b> but we are concerned for unvisited nodes only.

4		<p>Visit <b>D</b> and mark it as visited and put onto the stack. Here, we have <b>B</b> and <b>C</b> nodes, which are adjacent to <b>D</b> and both are unvisited. However, we shall again choose in an alphabetical order.</p>
5		<p>We choose <b>B</b>, mark it as visited and put onto the stack. Here <b>B</b> does not have any unvisited adjacent node. So, we pop <b>B</b> from the stack.</p>
6		<p>We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find <b>D</b> to be on the top of the stack.</p>

7



Only unvisited adjacent node is from **D** is **C** now. So we visit **C**, mark it as visited and put it onto the stack.

As **C** does not have any unvisited adjacent node so we keep popping the stack until we find a node that has an unvisited adjacent node. In this case, there's none and we keep popping until the stack is empty.

([tutorialsPoint, 2018](#)).

## References

- Overlq.com. (2017). *One dimensional Array in C*. Retrieved from Overlq.com:  
<https://overlq.com/c-programming/101/one-dimensional-array-in-c/>
- tutorialsPoint. (2018). *Data Structure - Breadth First Traversal*. Retrieved from tutorialsPoint:  
[https://www.tutorialspoint.com/data\\_structures\\_algorithms/breadth\\_first\\_traversal.htm](https://www.tutorialspoint.com/data_structures_algorithms/breadth_first_traversal.htm)
- tutorialsPoint. (2018). *Data Structure - Depth First Traversal*. Retrieved from tutorialsPoint:  
[https://www.tutorialspoint.com/data\\_structures\\_algorithms/depth\\_first\\_traversal.htm](https://www.tutorialspoint.com/data_structures_algorithms/depth_first_traversal.htm)
- tutorialsPoint. (2018). *Data Structure and Algorithms - Linked List*. Retrieved from tutorialsPoint:  
[https://www.tutorialspoint.com/data\\_structures\\_algorithms/linked\\_list\\_algorithms.htm](https://www.tutorialspoint.com/data_structures_algorithms/linked_list_algorithms.htm)
- tutorialsPoint. (2018). *Data Structure and Algorithms - Queue*. Retrieved from tutorialsPoint:  
[https://www.tutorialspoint.com/data\\_structures\\_algorithms/dsa\\_queue.htm](https://www.tutorialspoint.com/data_structures_algorithms/dsa_queue.htm)
- tutorialsPoint. (2018). *Data Structure and Algorithms - Stack*. Retrieved from tutorialsPoint:  
[https://www.tutorialspoint.com/data\\_structures\\_algorithms/stack\\_algorithm.htm](https://www.tutorialspoint.com/data_structures_algorithms/stack_algorithm.htm)