DATA STRUCTURE AND APPLICATIONS

Prepared for

Sir Richard Philip

Lecturer

Department of CSE

City University

Prepared by

Billal Hossen

Students of

City University

October 24, 2018

Contents

Array

Arrays a kind of data structure that can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.
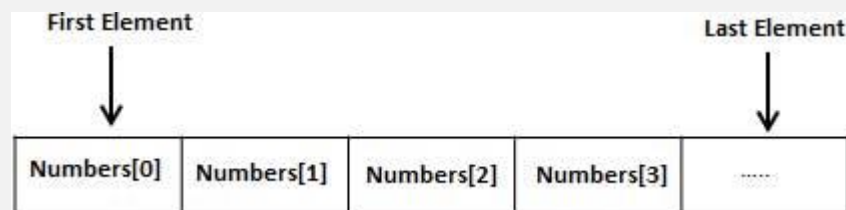
1D ARRAY:

An array is a data structure which can store multiple homogenous data elements under one name.

Suppose you want to store age of 50 students using a programming language. This can be done by taking 50 different variables for each student. But it is not easy to handle 50 different variables for this.

An alternative to this is arrays where we need not declare 50 variables for 50 students. When a list of data item is specified under one name using a single subscript, then such a variable is called a one-dimensional (1-D) array. An array is declared in following way:

Specifies type of element such as int, float, char etc .and size indicates maximum number of elements that can be stored inside the array.

1.1   2D ARRAY:

Implementing a database of information as a collection of arrays can be inconvenient when we have to pass many arrays to utility functions to process the database. It would be nice to have a single data structure which can hold all the information, and pass it all at once.

2-dimensional arrays provide most of this capability. Like a 1D array, a 2D array is a collection of data cells, all of the same type, which can be given a single name. However, a 2D array is organized as a matrix with a number of rows and columns.

(C - Arrays, 2018)

Stack

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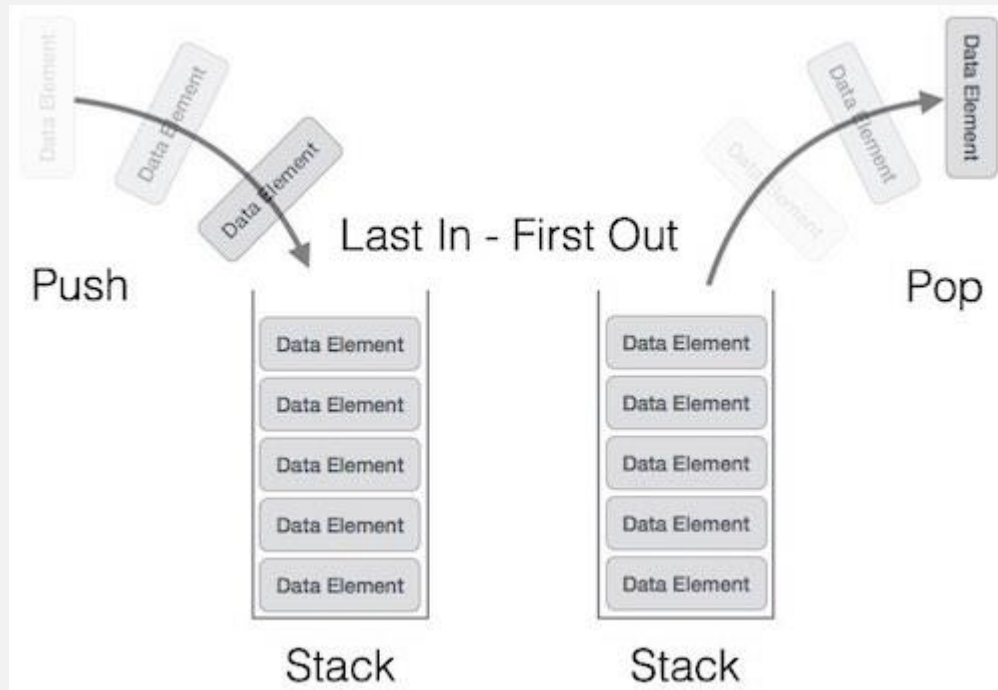
A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.

1.2   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.

1.3   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.
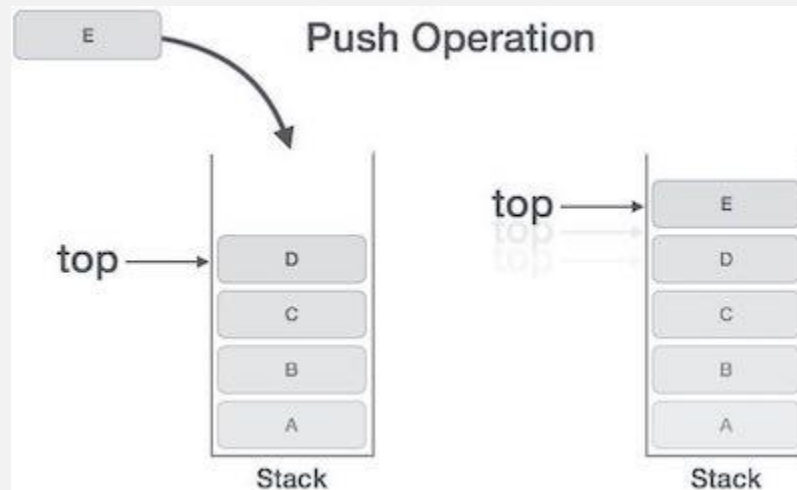
If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.

1.4   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.
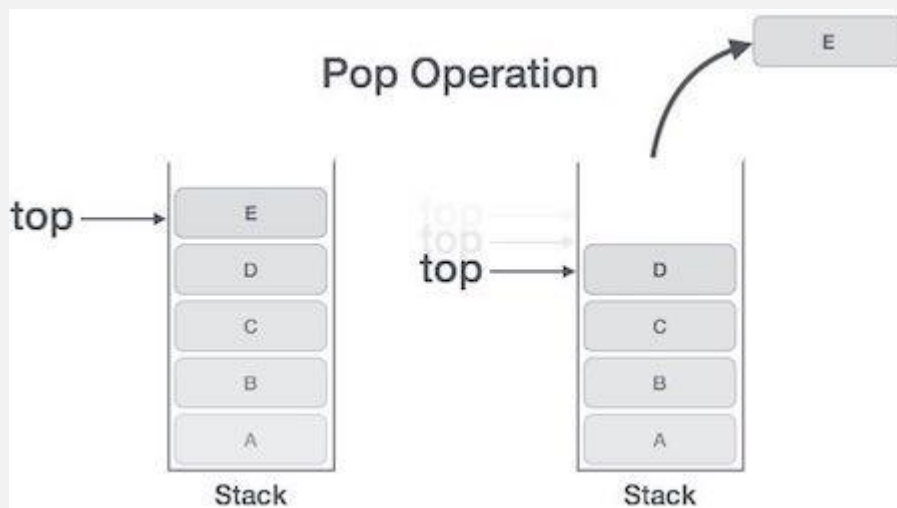
(Data Structure and Algorithms – Stack, 2018)

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.

1.5   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.

1.6   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 enqueing (or storing) data in the queue we take help of **rear** pointer.

1.7   Enqueue Operation

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

Queue Enqueue
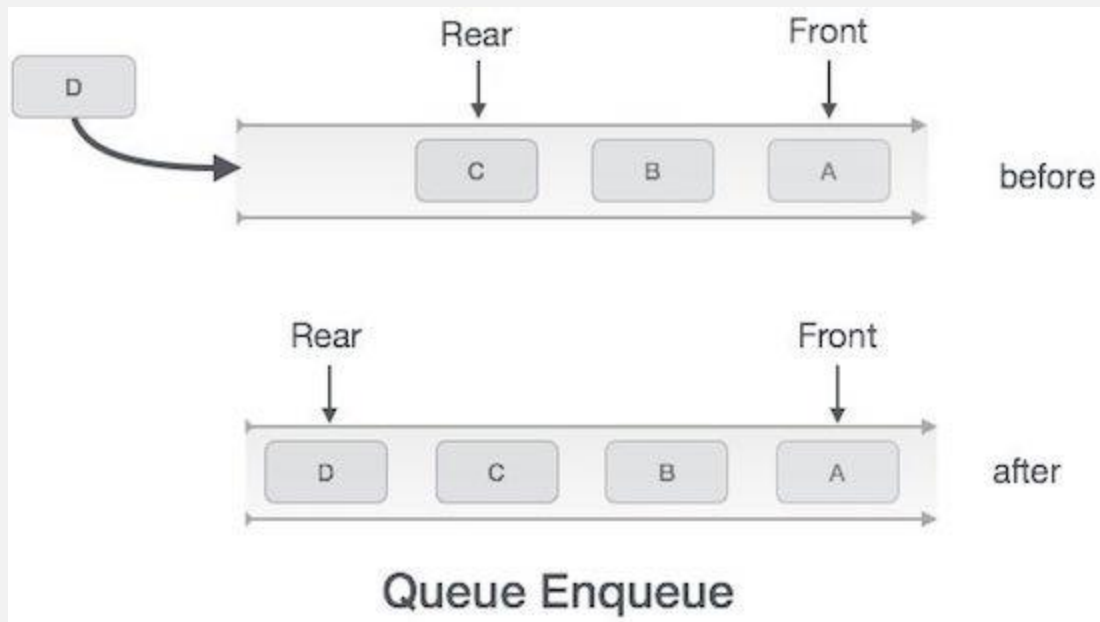
Sometimes, we also check to see if a queue is initialized or not, to handle any unforeseen situations.

1.8   Dequeue Operation

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 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.

Queue Dequeue
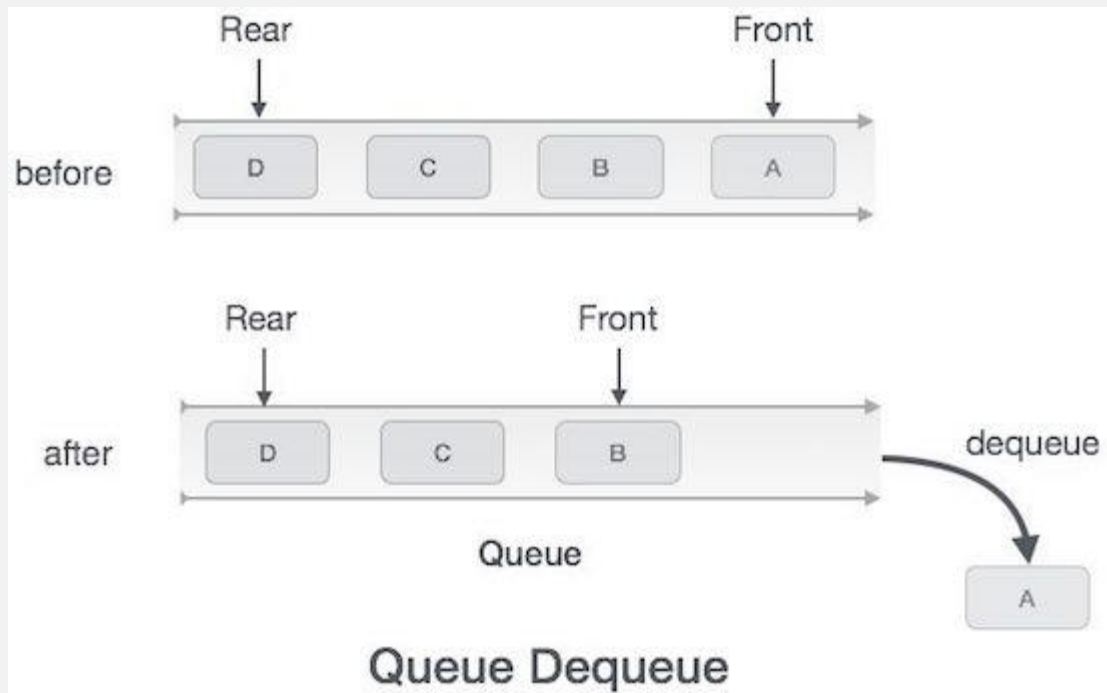
)Data Structure and Algorithms - Queue, 2018(

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.

1.9 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.

1.10 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.

1.11 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.

1.12 Insertion Operation

Adding a new node in linked list is a more than one step activity. We shall learn this with diagrams here. First, create a node using the same structure and find the location where it has to be inserted.
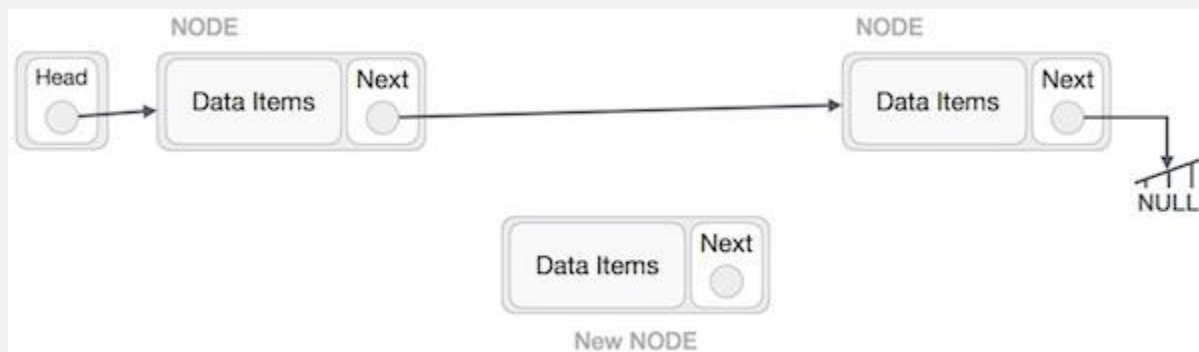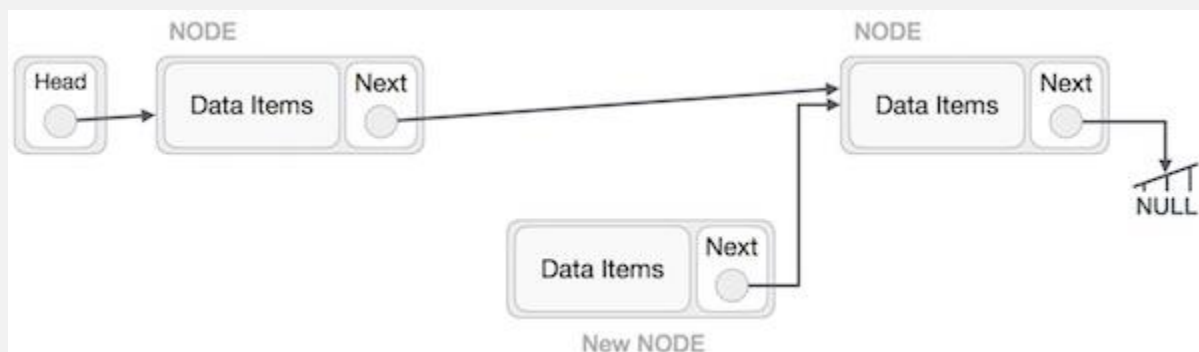

Imagine that we are inserting a node **B** (NewNode), between **A** (LeftNode) and **C** (RightNode). Then point B.next to C −

NewNode.next −> RightNode;

It should look like this −

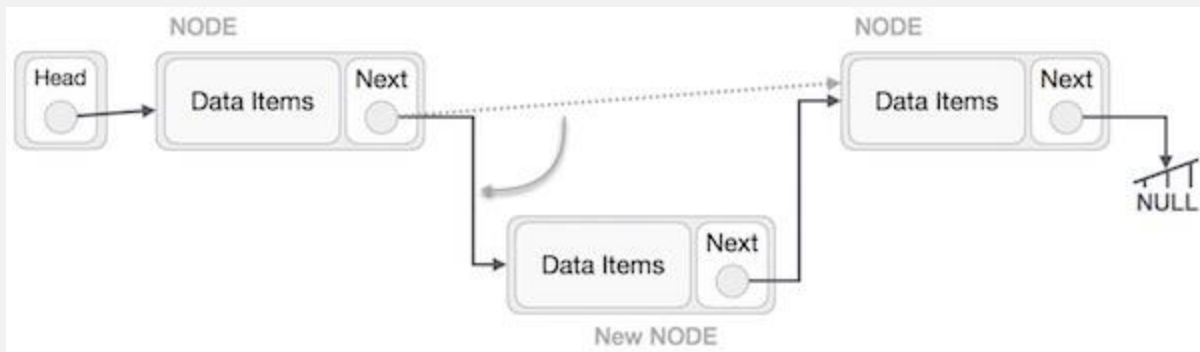

Now, the next node at the left should point to the new node.

LeftNode.next −> NewNode;

This will put the new node in the middle of the two. The new list should look like this −

Similar steps should be taken if the node is being inserted at the beginning of the list. While inserting it at the end, the second last node of the list should point to the new node and the new node will point to NULL.
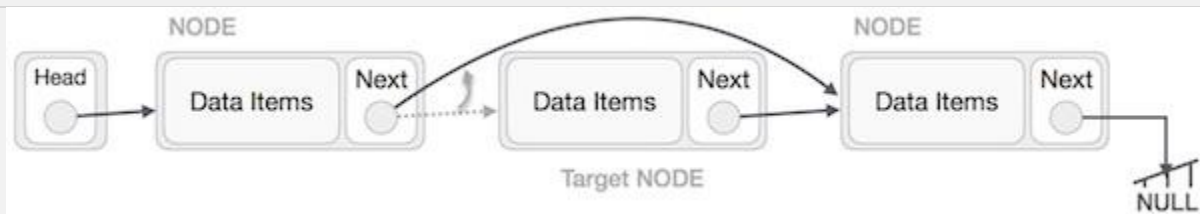
1.13 Deletion Operation

Deletion is also a more than one step process. We shall learn with pictorial representation. First, locate the target node to be removed, by using searching algorithms.

The left (previous) node of the target node now should point to the next node of the target node −

```
LeftNode.next −> TargetNode.next;
```
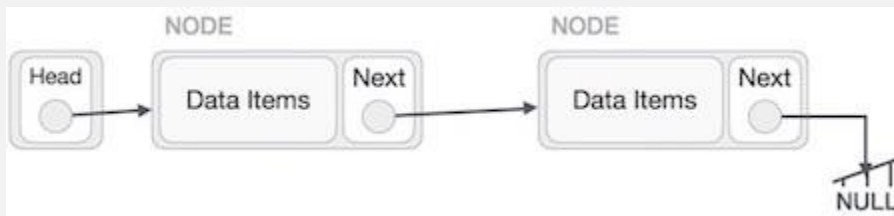

This will remove the link that was pointing to the target node. Now, using the following code, we will remove what the target node is pointing at.
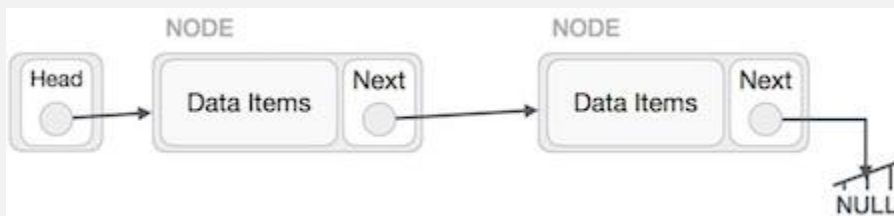
```
TargetNode.next -> NULL;
```


We need to use the deleted node. We can keep that in memory otherwise we can simply deallocate memory and wipe off the target node completely.
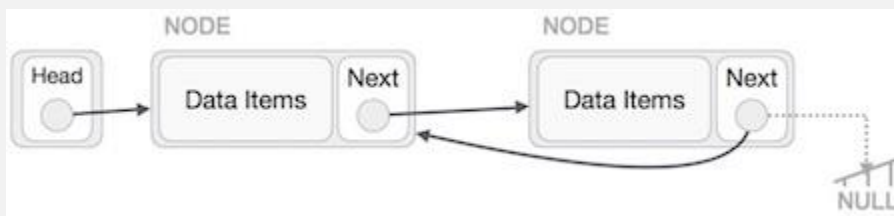

1.14   Reverse Operation

This operation is a thorough one. We need to make the last node to be pointed by the head node and reverse the whole linked list.

First, we traverse to the end of the list. It should be pointing to NULL. Now, we shall make it point to its previous node −


We have to make sure that the last node is not the lost node. So we'll have some temp node, which looks like the head node pointing to the last node. Now, we shall make all left side nodes point to their previous nodes one by one.
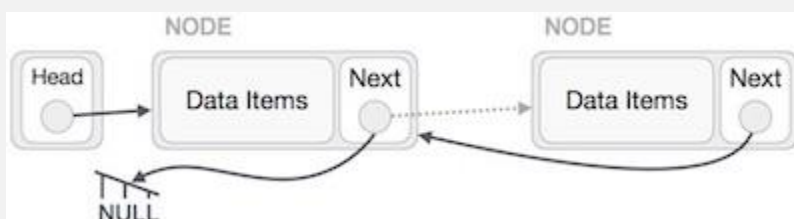
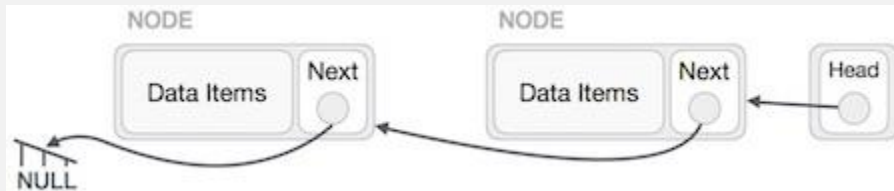Except the node (first node) pointed by the head node, all nodes should point to their predecessor, making them their new successor. The first node will point to NULL.
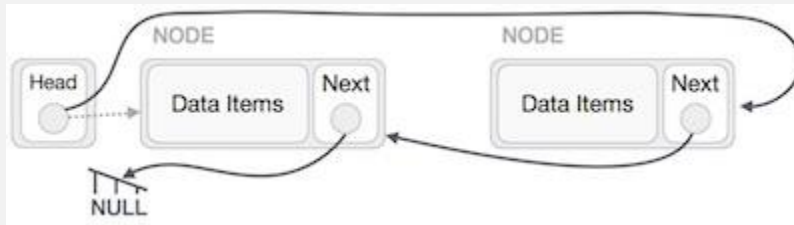
We'll make the head node point to the new first node by using the temp node.

The linked list is now reversed. To see linked list implementation in C programming language, please click here.

)Data Structure and Algorithms - Linked List, 2018(

Tree

Tree represents the nodes connected by edges. We will discuss binary tree or binary search tree specifically.

Binary Tree is a special data structure used for data storage purposes. A binary tree has a special condition that each node can have a maximum of two children. A binary tree has the benefits of both an ordered array and a linked list as search is as quick as in a sorted array and insertion or deletion operation are as fast as in linked list.
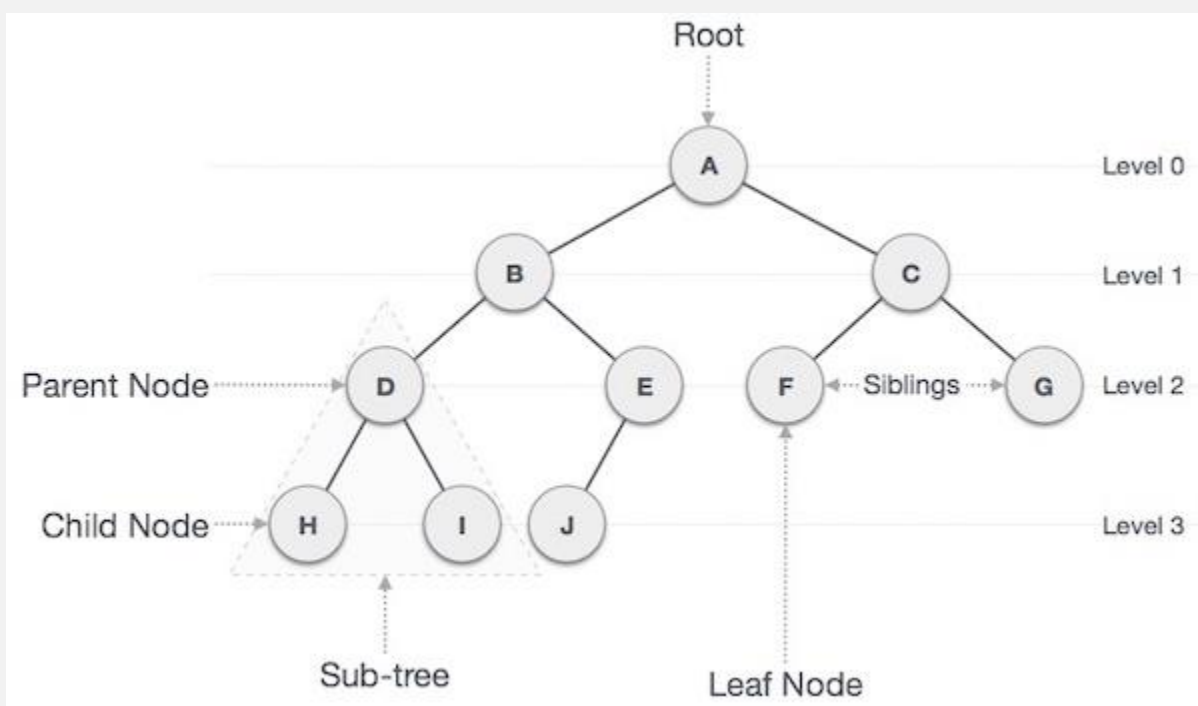
1.15 Important Terms

Following are the important terms with respect to tree.

- **Path** − Path refers to the sequence of nodes along the edges of a tree.

- **Root** − The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.

- **Parent** − Any node except the root node has one edge upward to a node called parent.

- **Child** − The node below a given node connected by its edge downward is called its child node.

- **Leaf** − The node which does not have any child node is called the leaf node.

- **Subtree** − Subtree represents the descendants of a node.

- **Visiting** − Visiting refers to checking the value of a node when control is on the node.

- **Traversing** − Traversing means passing through nodes in a specific order.

- **Levels** − Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.

- **keys** − Key represents a value of a node based on which a search operation is to be carried out for a node.

1.16 Binary Search Tree Representation

Binary Search tree exhibits a special behavior. A node's left child must have a value less than its parent's value and the node's right child must have a value greater than its parent value.
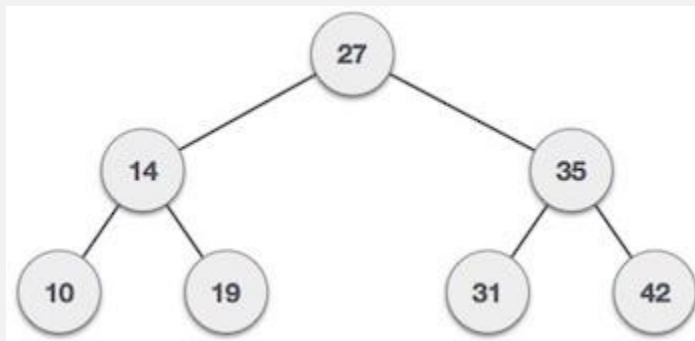
We're going to implement tree using node object and connecting them through references.

1.17 Tree Node

The code to write a tree node would be similar to what is given below. It has a data part and references to its left and right child nodes.

```
struct node {

   int data;

   struct node *leftChild;

   struct node *rightChild;
```

```
};
```

In a tree, all nodes share common construct.

1.18   BST Basic Operations

The basic operations that can be performed on a binary search tree data structure, are the following −

- **Insert** − Inserts an element in a tree/create a tree.

- **Search** − Searches an element in a tree.

- **Preorder Traversal** − Traverses a tree in a pre-order manner.

- **Inorder Traversal** − Traverses a tree in an in-order manner.

- **Postorder Traversal** − Traverses a tree in a post-order manner.

We shall learn creating (inserting into) a tree structure and searching a data item in a tree in this chapter. We shall learn about tree traversing methods in the coming chapter.

1.19   Insert Operation

The very first insertion creates the tree. Afterwards, whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.

Algorithm

```
If root is NULL

   then create root node

return


If root exists then

   compare the data with node.data


   while until insertion position is located
```

If data is greater than node.data

            goto right subtree

        else

            goto left subtree

    endwhile

    insert data

end If

Implementation

The implementation of insert function should look like this −

```c
void insert(int data) {
    struct node *tempNode = (struct node*) malloc(sizeof(struct node));
    struct node *current;
    struct node *parent;

    tempNode->data = data;
    tempNode->leftChild = NULL;
    tempNode->rightChild = NULL;

    //if tree is empty, create root node
    if(root == NULL) {
        root = tempNode;
    } else {
        current = root;
```

```c
parent  = NULL;


while(1) {
  parent = current;


  //go to left of the tree
  if(data < parent->data) {
    current = current->leftChild;


    //insert to the left
    if(current == NULL) {
      parent->leftChild = tempNode;

      return;

    }

  }


  //go to right of the tree
  else {
    current = current->rightChild;


    //insert to the right
    if(current == NULL) {
      parent->rightChild = tempNode;

      return;

    }

  }
```

```
    }

  }

}
```

1.20   Search Operation

Whenever an element is to be searched, start searching from the root node, then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree. Follow the same algorithm for each node.

Algorithm

```
If root.data is equal to search.data

   return root

else

   while data not found


     If data is greater than node.data

        goto right subtree

     else

        goto left subtree


     If data found

        return node

   endwhile


   return data not found


end if
```

The implementation of this algorithm should look like this.

```c
struct node* search(int data) {

  struct node *current = root;

  printf("Visiting elements: ");


  while(current->data != data) {

    if(current != NULL)

    printf("%d ",current->data);


    //go to left tree


    if(current->data > data) {

      current = current->leftChild;

    }
    //else go to right tree
    else {

      current = current->rightChild;

    }


    //not found

    if(current == NULL) {

      return NULL;

    }


    return current;

  }
```

```
}
```

Binary Search Tree

A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties −

- The left sub-tree of a node has a key less than or equal to its parent node's key.

- The right sub-tree of a node has a key greater than to its parent node's key.

Thus, BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree and can be defined as −

```
left_subtree (keys)  ≤  node (key)  ≤  right_subtree (keys)
```

1.21   Representation

BST is a collection of nodes arranged in a way where they maintain BST properties. Each node has a key and an associated value. While searching, the desired key is compared to the keys in BST and if found, the associated value is retrieved.

Following is a pictorial representation of BST −
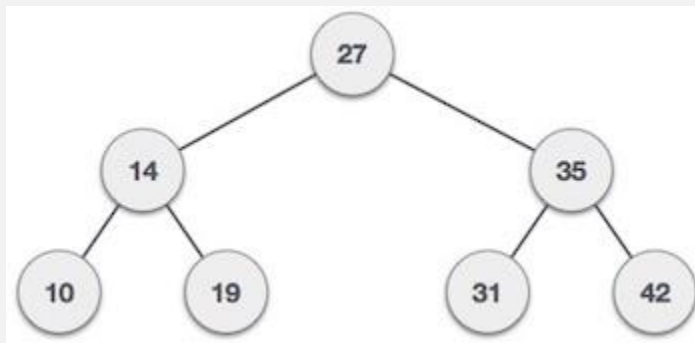
We observe that the root node key (27) has all less-valued keys on the left sub-tree and the higher valued keys on the right sub-tree.

1.22   Basic Operations

Following are the basic operations of a tree −

- **Search** − Searches an element in a tree.

- **Insert** − Inserts an element in a tree.

- **Pre-order Traversal** − Traverses a tree in a pre-order manner.

- **In-order Traversal** − Traverses a tree in an in-order manner.

- **Post-order Traversal** − Traverses a tree in a post-order manner.

1.23 Node

Define a node having some data, references to its left and right child nodes.

```c
struct node {

  int data;

  struct node *leftChild;

  struct node *rightChild;

};
```

1.24 Search Operation

Whenever an element is to be searched, start searching from the root node. Then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree. Follow the same algorithm for each node.

Algorithm

```c
struct node* search(int data){

  struct node *current = root;

  printf("Visiting elements: ");


  while(current->data != data){


    if(current != NULL) {

      printf("%d ",current->data);


      //go to left tree

      if(current->data > data){

        current = current->leftChild;
```

```
      } //else go to right tree

      else {

         current = current->rightChild;

      }


      //not found

      if(current == NULL){

         return NULL;

      }

   }

 }


   return current;

}
```

1.25   Insert Operation

Whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.

Algorithm

```
void insert(int data) {

   struct node *tempNode = (struct node*) malloc(sizeof(struct node));

   struct node *current;

   struct node *parent;


   tempNode->data = data;
```

```c
tempNode->leftChild = NULL;

tempNode->rightChild = NULL;


//if tree is empty

if(root == NULL) {

  root = tempNode;

} else {

  current = root;

  parent = NULL;


  while(1) {

    parent = current;


    //go to left of the tree

    if(data < parent->data) {

      current = current->leftChild;

      //insert to the left


      if(current == NULL) {

        parent->leftChild = tempNode;

        return;

      }

    } //go to right of the tree

    else {

      current = current->rightChild;
```

```
        //insert to the right

        if(current == NULL) {

            parent->rightChild = tempNode;

            return;

        }

      }

    }

  }

}
```

)Tree Traversal in C, 2018(

Graph Data Structure

)Data Structure - Graph Data Structure, 2018(

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as **vertices**, and the links that connect the vertices are called **edges**.

Formally, a graph is a pair of sets **(V, E)**, where **V** is the set of vertices and **E**is the set of edges, connecting the pairs of vertices. Take a look at the following graph −
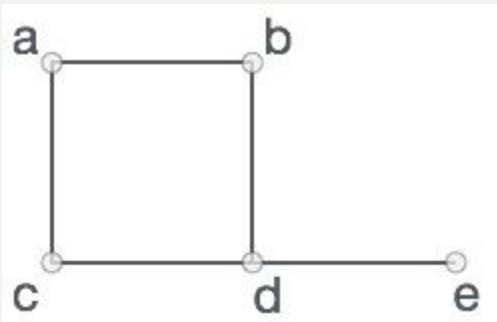
In the above graph,

V = {a, b, c, d, e}

E = {ab, ac, bd, cd, de}

Graph Data Structure

Mathematical graphs can be represented in data structure. We can represent a graph using an array of vertices and a two-dimensional array of edges. Before we proceed further, let's familiarize ourselves with some important terms −

- **Vertex** − Each node of the graph is represented as a vertex. In the following example, the labeled circle represents vertices. Thus, A to G are vertices. We can represent them using an array as shown in the following image. Here A can be identified by index 0. B can be identified using index 1 and so on.

- **Edge** − Edge represents a path between two vertices or a line between two vertices. In the following example, the lines from A to B, B to C, and so on represents edges. We can use a two-dimensional array to represent an array as

shown in the following image. Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column 2 and so on, keeping other combinations as 0.

- **Adjacency** − Two node or vertices are adjacent if they are connected to each other through an edge. In the following example, B is adjacent to A, C is adjacent to B, and so on.

- **Path** − Path represents a sequence of edges between the two vertices. In the following example, ABCD represents a path from A to D.
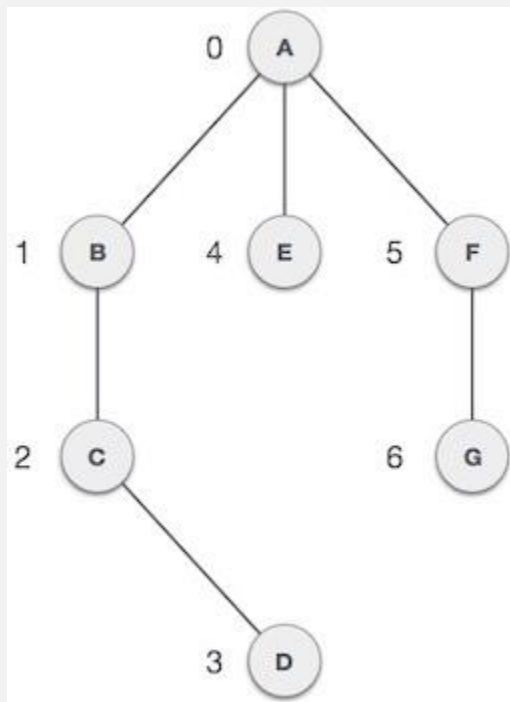

Basic Operations

Following are basic primary operations of a Graph −

- **Add Vertex** − Adds a vertex to the graph.

- **Add Edge** − Adds an edge between the two vertices of the graph.

- **Display Vertex** − Displays a vertex of the graph.

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.
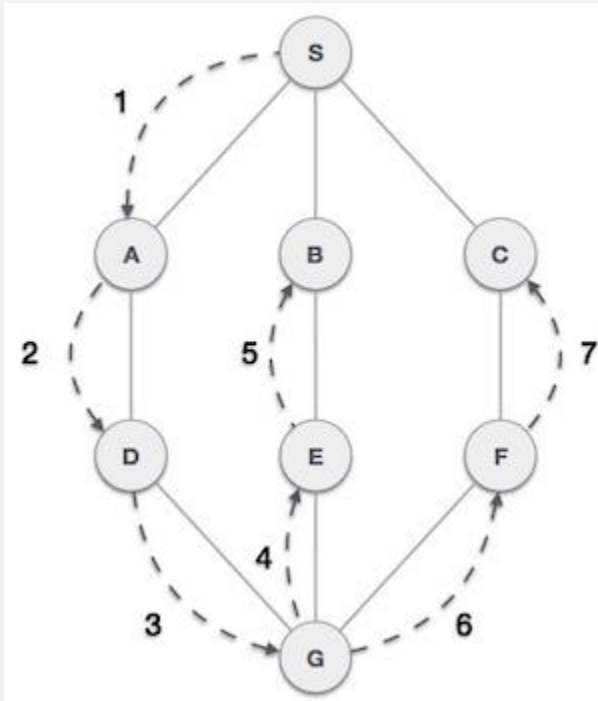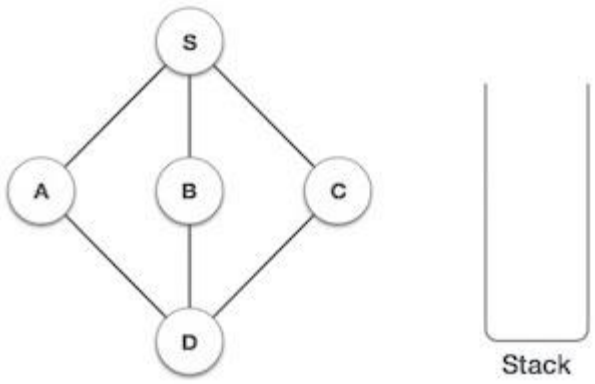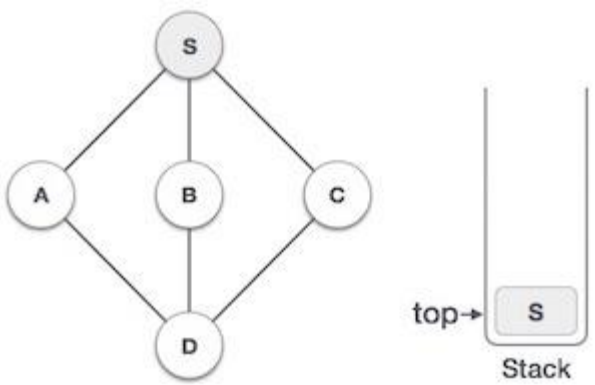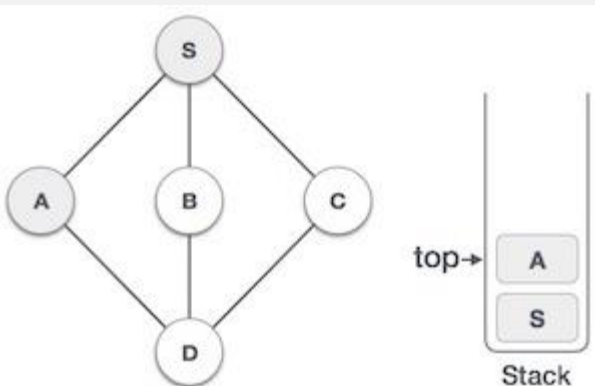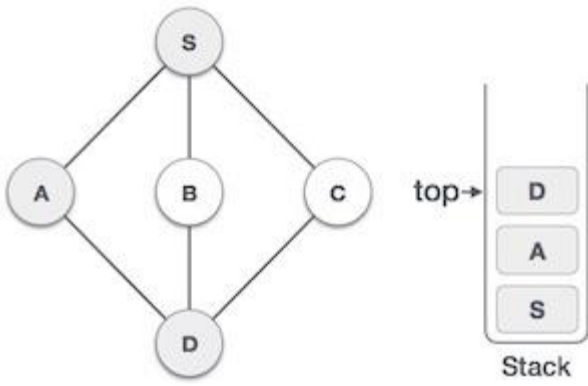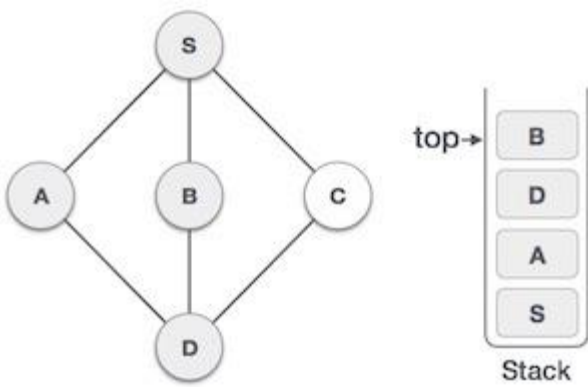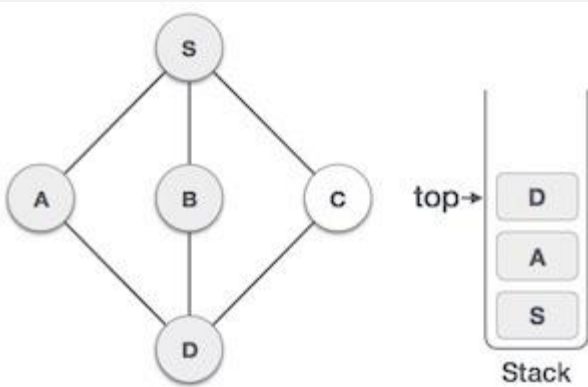

As in the example given above, DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to C. It employs the following rules.

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

4		Visit **D** and mark it as visited and put onto the stack. Here, we have **B** and **C** nodes, which are adjacent to **D** and both are unvisited. However, we shall again choose in an alphabetical order.
5		We choose **B**, mark it as visited and put onto the stack. Here **B**does not have any unvisited adjacent node. So, we pop **B**from the stack.
6		We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find **D** to be on the top of the stack.
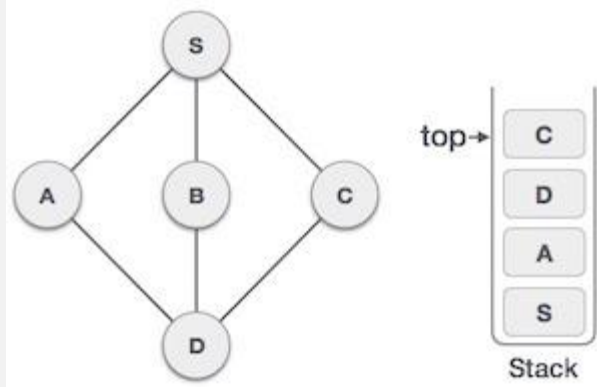
| 7 | | Only unvisited adjacent node is from **D** is **C** now. So we visit **C**, mark it as visited and put it onto the stack. |

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.

2 Bibliography

C - Arrays. (2018, October 31). Retrieved from Tutorials Point: https://www.tutorialspoint.com/cprogramming/c_arrays.htm

Data Structure - Graph Data Structure. (2018, October 31). Retrieved from Tutorials Point: https://www.tutorialspoint.com/data_structures_algorithms/graph_data_structure.htm

Data Structure and Algorithms - Linked List. (2018, October 31). Retrieved from Tutorials Point: https://www.tutorialspoint.com/data_structures_algorithms/linked_list_algorithms.htm

Data Structure and Algorithms - Queue. (2018, October 31). Retrieved from Tutorials Point: https://www.tutorialspoint.com/data_structures_algorithms/dsa_queue.htm

Data Structure and Algorithms - Stack. (2018, October 31). Retrieved from Tutorials Point: https://www.tutorialspoint.com/data_structures_algorithms/stack_algorithm.htm

Tree Traversal in C. (2018, October 31). Retrieved from Tutorials Point: https://www.tutorialspoint.com/data_structures_algorithms/tree_traversal_in_c.htm