DATA STRUCTURE

Stack, Queues, Linked list, Array, Tree

Israt Jahan Jennat

October 10, 2018

Table of Contents

Stack

A stack is a useful data structure in programming. It is just like a pile of plates kept on top of each other.

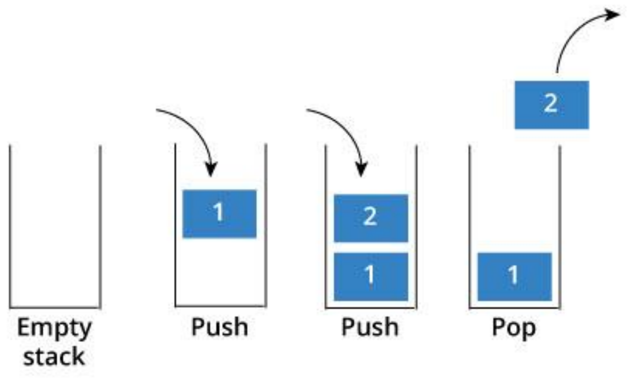
Think about the things you can do with such a pile of plates

- Put a new plate on top
- Remove the top plate

If you want the plate at the bottom, you have to first remove all the plates on top. Such kind of arrangement is called **Last In First Out** - the last item that was placed is the first item to go out.

Stack in Programming Terms

In programming terms, putting an item on top of the stack is called "push" and removing an item is called "pop".



In the above image, although item 2 was kept last, it was removed first - so it follows the Last In First Out(LIFO) principle.

Stack Specification

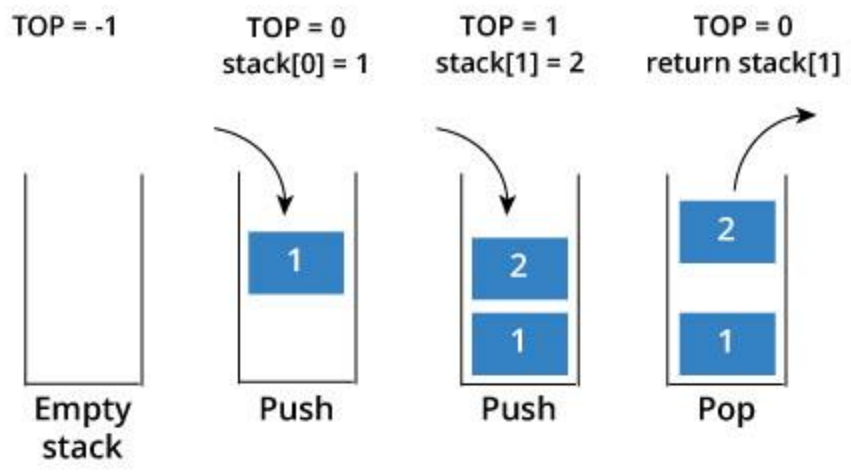
A stack is an object or more specifically an abstract data structure(ADT) that allows the following operations:

- `Push`: Add element to top of stack
- `Pop`: Remove element from top of stack
- `IsEmpty`: Check if stack is empty
- `IsFull`: Check if stack is full
- `Peek`: Get the value of the top element without removing it

How stack works

The operations work as follows:

1. A pointer called *TOP* is used to keep track of the top element in the stack.
2. When initializing the stack, we set its value to -1 so that we can check if the stack is empty by comparing `TOP == -1`.
3. On pushing an element, we increase the value of *TOP* and place the new element in the position pointed to by *TOP*.
4. On popping an element, we return the element pointed to by *TOP* and reduce its value.
5. Before pushing, we check if stack is already full
6. Before popping, we check if stack is already empty



Use of stack

Although stack is a simple data structure to implement, it is very powerful. The most common uses of a stack are:

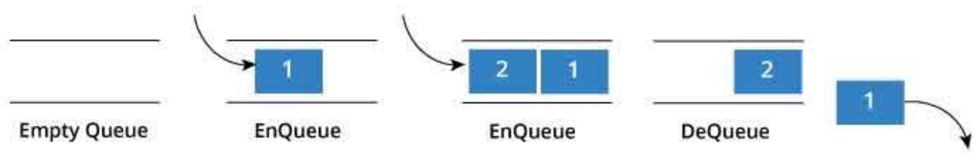
- **To reverse a word** - Put all the letters in a stack and pop them out. Because of LIFO order of stack, you will get the letters in reverse order.
- **In compilers** - Compilers use stack to calculate the value of expressions like 2+4/5*(7-9) by converting the expression to prefix or postfix form.
- **In browsers** - The back button in a browser saves all the urls you have visited previously in a stack. Each time you visit a new page, it is added on top of the stack. When you press the back button, the current URL is removed from the stack and the previous url is accessed.

(Studytonight, 2015)

Queue

A queue is a useful data structure in programming. It is similar to the ticket queue outside a cinema hall, where the first person entering the queue is the first person who gets the ticket.

Queue follows the **First In First Out(FIFO)** rule - the item that goes in first is the item that comes out first too.



In the above image, since 1 was kept in the queue before 2, it was the first to be removed from the queue as well. It follows the FIFO rule.

In programming terms, putting an item in the queue is called an "enqueue" and removing an item from the queue is called "dequeue".

We can implement queue in any programming language like C, C++, Java, Python or C#, but the specification is pretty much the same.

Queue Specifications

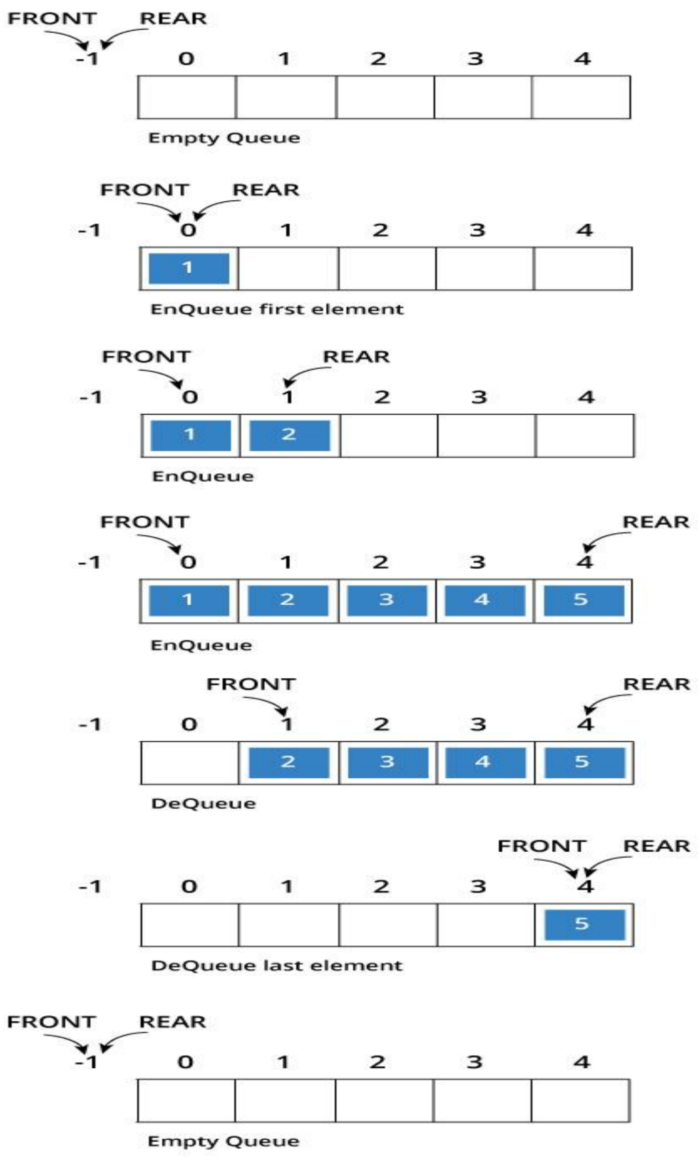
A queue is an object or more specifically an abstract data structure(ADT) that allows the following operations:

- Enqueue: Add element to end of queue
- Dequeue: Remove element from front of queue
- IsEmpty: Check if queue is empty
- IsFull: Check if queue is full
- Peek: Get the value of the front of queue without removing it

How Queue Works

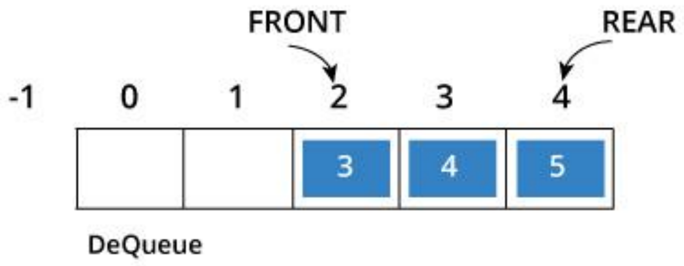
Queue operations work as follows:

1. Two pointers called *FRONT* and *REAR* are used to keep track of the first and last elements in the queue.
2. When initializing the queue, we set the value of *FRONT* and *REAR* to -1.
3. On enqueing an element, we increase the value of *REAR* index and place the new element in the position pointed to by *REAR*.
4. On dequeueing an element, we return the value pointed to by *FRONT* and increase the *FRONT* index.
5. Before enqueing, we check if queue is already full.
6. Before dequeuing, we check if queue is already empty.
7. When enqueing the first element, we set the value of *FRONT* to 0.
8. When dequeing the last element, we reset the values of *FRONT* and *REAR* to -1.



Limitation of this implementation

As you can see in the image below, after a bit of enqueueing and dequeueing, the size of the queue has been reduced.



The indexes 0 and 1 can only be used after the queue is reset when all the elements have been dequeued.

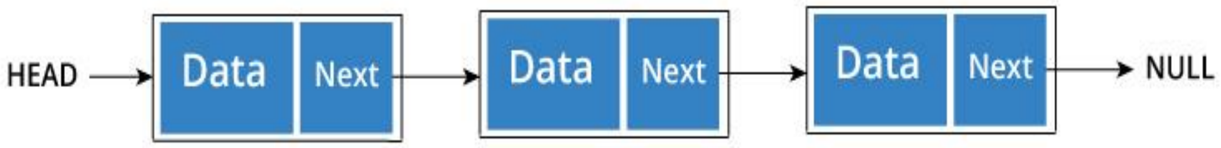
By tweaking the code for queue, we can use the space by implementing a modified queue called

(Studytonight, 2015)

Linked list

A linked list is similar. It is a series of connected "nodes" that contains the "address" of the next node. Each node can store a data point which may be a number, a string or any other type of data.

Linked List Representation



You have to start somewhere, so we give the address of the first node a special name called *HEAD*.

Also, the last node in the linkedlist can be identified because its next portion points to *NULL*.

How another node is referenced?

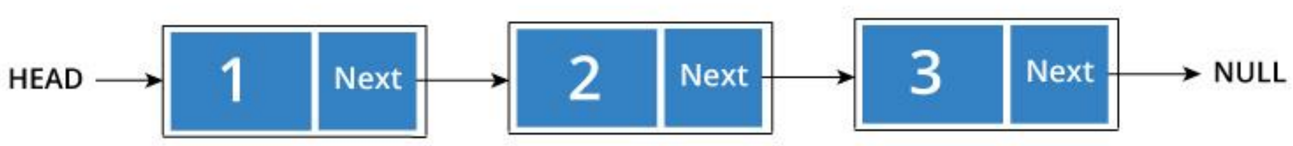
Some pointer magic is involved. Let's think about what each node contains:

- A data item
- An address of another node

We wrap both the data item and the next node reference in a struct as:

```
struct node
{
  int data;
  struct node *next;
};
```

n just a few steps, we have created a simple linkedlist with three nodes.



The power of linkedlist comes from the ability to break the chain and rejoin it. E.g. if you wanted to put an element 4 between 1 and 2, the steps would be:

- Create a new struct node and allocate memory to it.
- Add its data value as 4
- Point its next pointer to the struct node containing 2 as data value
- Change next pointer of "1" to the node we just created.

Doing something similar in an array would have required shifting the positions of all the subsequent elements.

Linked List Operations

Now that you have got an understanding of the basic concepts behind linked list and their types, its time to dive into the common operations that can be performed.

Two important points to remember:

- *head* points to the first node of the linked list
- *next* pointer of last node is *NULL*, so if next of current node is *NULL*, we have reached end of linked list.

In all of the examples, we will assume that the linked list has three nodes `1 --->2 --->3` with node structure as below:

```
struct node
{
  int data;
  struct node *next;
};
```

How to traverse a linked list

Displaying the contents of a linked list is very simple. We keep moving the temp node to the next one and display its contents.

When temp is *NULL*, we know that we have reached the end of linked list so we get out of the while loop

How to add elements to linked list

You can add elements to either beginning, middle or end of linked list.

Add to beginning

- Allocate memory for new node
- Store data
- Change next of new node to point to head
- Change head to point to recently created node

Add to end

- Allocate memory for new node
- Store data
- Traverse to last node
- Change next of last node to recently created node

Add to middle

- Allocate memory and store data for new node

- Traverse to node just before the required position of new node
- Change next pointers to include new node in between

How to delete from a linked list

You can delete either from beginning, end or from a particular position.

Delete from beginning

- Point head to the second node

Delete from end

- Traverse to second last element
- Change its next pointer to null

Delete from middle

- Traverse to element before the element to be deleted
- Change next pointers to exclude the node from the chain

Array

An array is a collection of data that holds fixed number of values of same type. For example: if you want to store marks of 100 students, you can create an array for it.

The size and type of arrays cannot be changed after its declaration.

Arrays are of two types:

1. One-dimensional arrays
2. Multidimensional arrays

How to declare an array in C?

For example,

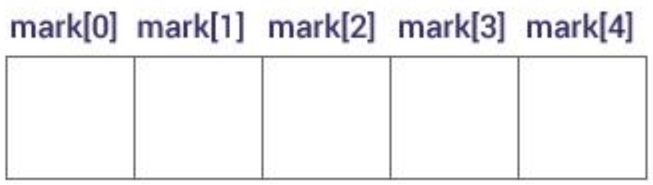
```
float mark[5];
```

Here, we declared an array, *mark*, of floating-point type and size 5. Meaning, it can hold 5 floating-point values.

Elements of an Array and How to access them?

You can access elements of an array by indices.

Suppose you declared an array *mark* as above. The first element is *mark[0]*, second element is *mark[1]* and so on.



Few key notes:

- Arrays have 0 as the first index not 1. In this example, *mark[0]*
- If the size of an array is *n*, to access the last element, `(n-1)` index is used. In this example, *mark[4]*
- Suppose the starting address of `mark[0]` is 2120d. Then, the next address, `a[1]`, will be 2124d, address of `a[2]` will be 2128d and so on. It's because the size of a float is 4 bytes.

How to initialize an array in C programming?

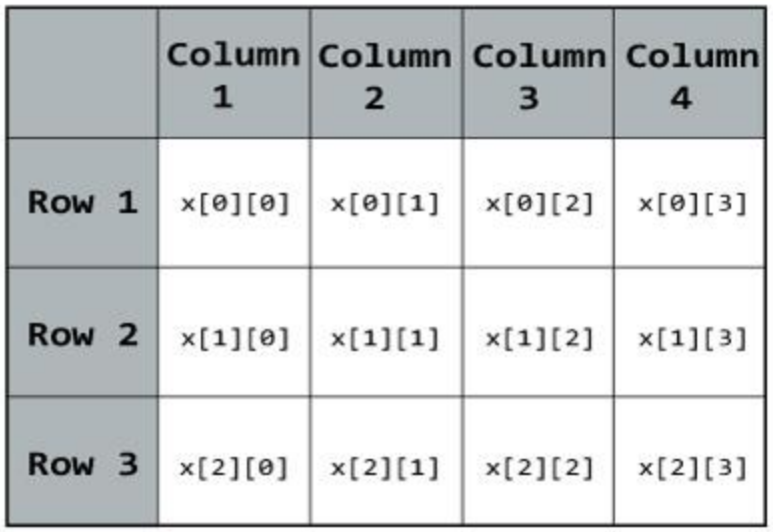
It's possible to initialize an array during declaration. For example,

```
int mark[5] = {19, 10, 8, 17, 9};
```

Multidimensional Array

In C programming, you can create an array of arrays known as multidimensional array. For example,

Here, *x* is a two-dimensional (2d) array. The array can hold 12 elements. You can think the array as table with 3 row and each row has 4 column.



	Column 1	Column 2	Column 3	Column 4
Row 1	x[0][0]	x[0][1]	x[0][2]	x[0][3]
Row 2	x[1][0]	x[1][1]	x[1][2]	x[1][3]
Row 3	x[2][0]	x[2][1]	x[2][2]	x[2][3]

Similarly, you can declare a three-dimensional (3d) array. For example,

```
float y[2][4][3];
```

Here, The array *y* can hold 24 elements.

You can think this example as: Each 2 elements have 4 elements, which makes 8 elements and each 8 elements can have 3 elements. Hence, the total number of elements is 24.

How to initialize a multidimensional array?

There is more than one way to initialize a multidimensional array.

Initialization of a two dimensional array

```
// Different ways to initialize two dimensional array

int c[2][3] = {{1, 3, 0}, {-1, 5, 9}};

int c[][3] = {{1, 3, 0}, {-1, 5, 9}};
```

int c[2][3] = {1, 3, 0, -1, 5, 9}; Above code are three different ways to initialize a two dimensional arrays.

(Studytonight, 2015)

Initialization of a three dimensional array.

You can initialize a three dimensional array in a similar way like a two dimensional array. Here's an example,

```
int test[2][3][4] = {
                     { {3, 4, 2, 3}, {0, -3, 9, 11}, {23, 12, 23, 2} },
                     { {13, 4, 56, 3}, {5, 9, 3, 5}, {3, 1, 4, 9} }
                 };
```

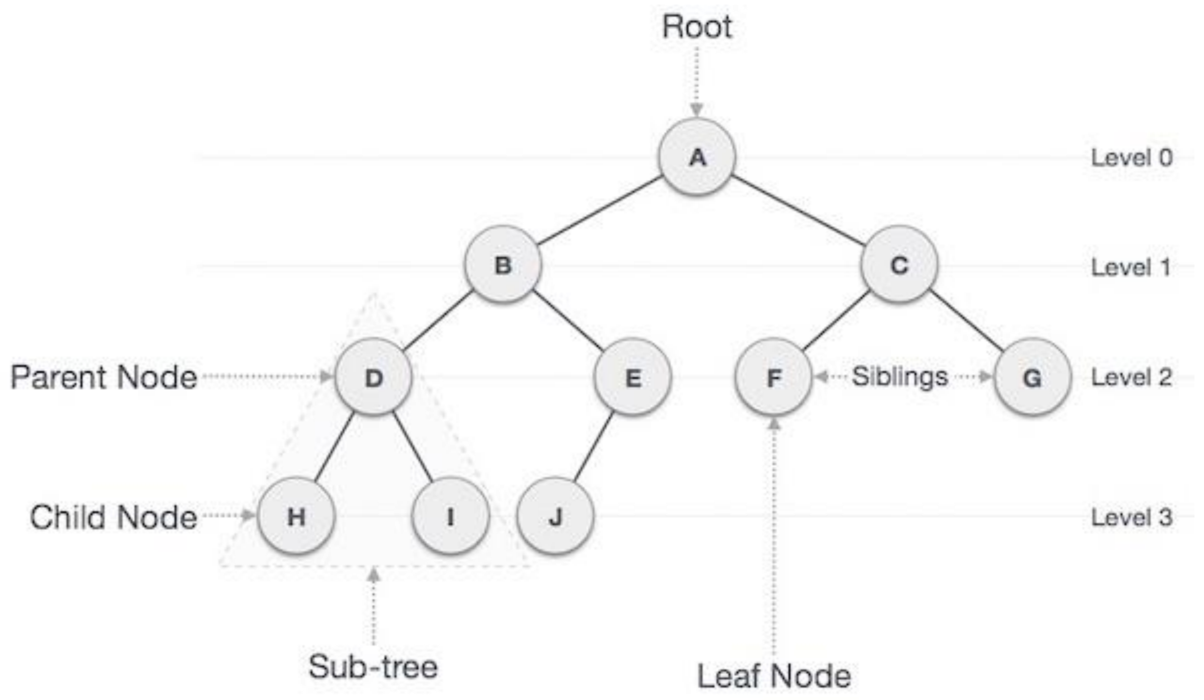
Multidimensional Arrays

- Multi-dimensional arrays are declared by providing more than one set of square [ ] brackets after the variable name in the declaration statement.
- One dimensional arrays do not require the dimension to be given if the array is to be completely initialized. By analogy, multi-dimensional arrays do not require **the first** dimension to be given if the array is to be completely initialized. All dimensions after the first must be given in any case.
- For two dimensional arrays, the first dimension is commonly considered to be the number of rows, and the second dimension the number of columns. We will use this convention when discussing two dimensional arrays.
- Two dimensional arrays are considered by C/C++ to be an array of (single dimensional arrays). For example, "int numbers[5][6]" would refer to a single dimensional array of 5 elements, wherein each element is a single dimensional array of 6 integers. By extension, "int numbers[12][5][6]" would refer to an array of twelve elements, each of which is a two dimensional array, and so on.
- Another way of looking at this is that C stores two dimensional arrays by rows, with all elements of a row being stored together as a single unit. Knowing this can sometimes lead to more efficient programs.
- Multidimensional arrays may be completely initialized by listing all data elements within a single pair of curly {} braces, as with single dimensional arrays.
- It is better programming practice to enclose each row within a separate subset of curly {} braces, to make the program more readable. This is required if any row other than the last is to be partially initialized. When subsets of braces are used, the last item within braces is not followed by a comma, but the subsets are themselves separated by commas.
- Multidimensional arrays may be partially initialized by not providing complete initialization data. Individual rows of a multidimensional array may be partially initialized, provided that subset braces are used.
- Individual data items in a multidimensional array are accessed by fully qualifying an array element. Alternatively, a smaller dimensional array may be accessed by partially qualifying the array name. For example, if "data" has been declared as a three dimensional array of floats, then data[1][2][5] would refer to a float, data[1][2] would refer to a one-dimensional array of floats, and data[1] would refer to a two-dimensional array of floats. The reasons for this and the incentive to do this relate to memory-management issues that are beyond the scope of these notes.

Tree

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

Binary Tree is a special datastructure 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.



Important Terms

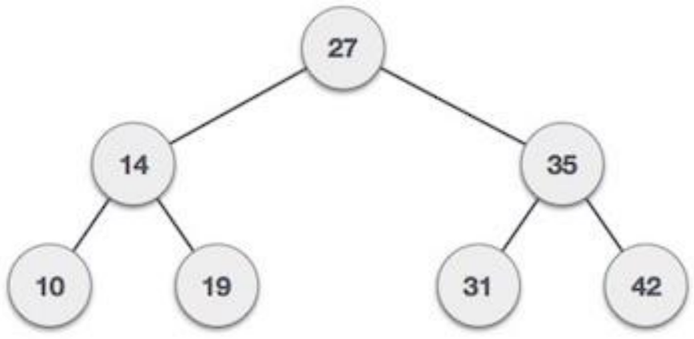
Following are the important terms with respect to tree.

- **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.

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.

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.

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.

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 −

```
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;
   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;

          }

        }

      }

  }

}
```

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

The implementation of this algorithm should look like this.

Traversal process

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree −

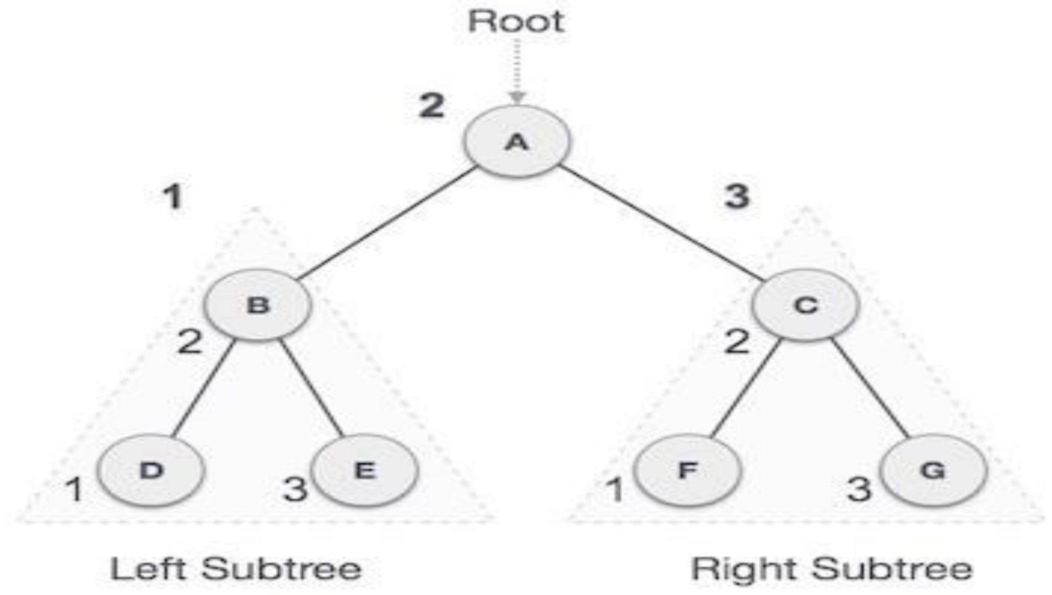
- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.

In-order Traversal

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.

If a binary tree is traversed **in-order**, the output will produce sorted key values in an ascending order.



We start from **A**, and following in-order traversal, we move to its left subtree **B**. **B** is also traversed in-order. The process goes on until all the nodes are visited. The output of inorder traversal of this tree will be −

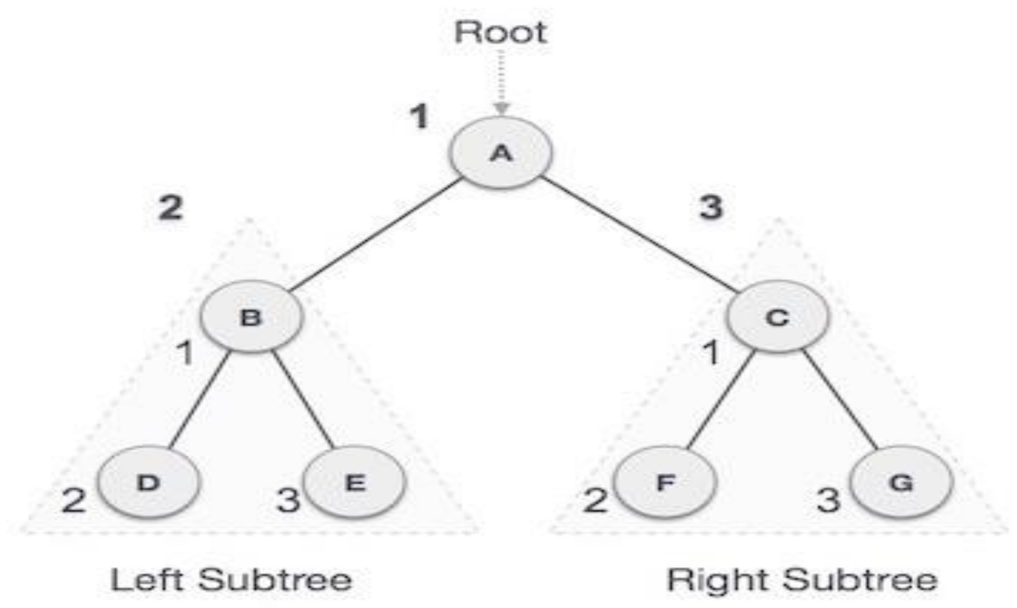
D → B → E → A → F → C → G

Algorithm

```
Until all nodes are traversed −
Step 1 − Recursively traverse left subtree.
Step 2 − Visit root node.
Step 3 − Recursively traverse right subtree.
```

Pre-order Traversal

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.



We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left subtree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be −

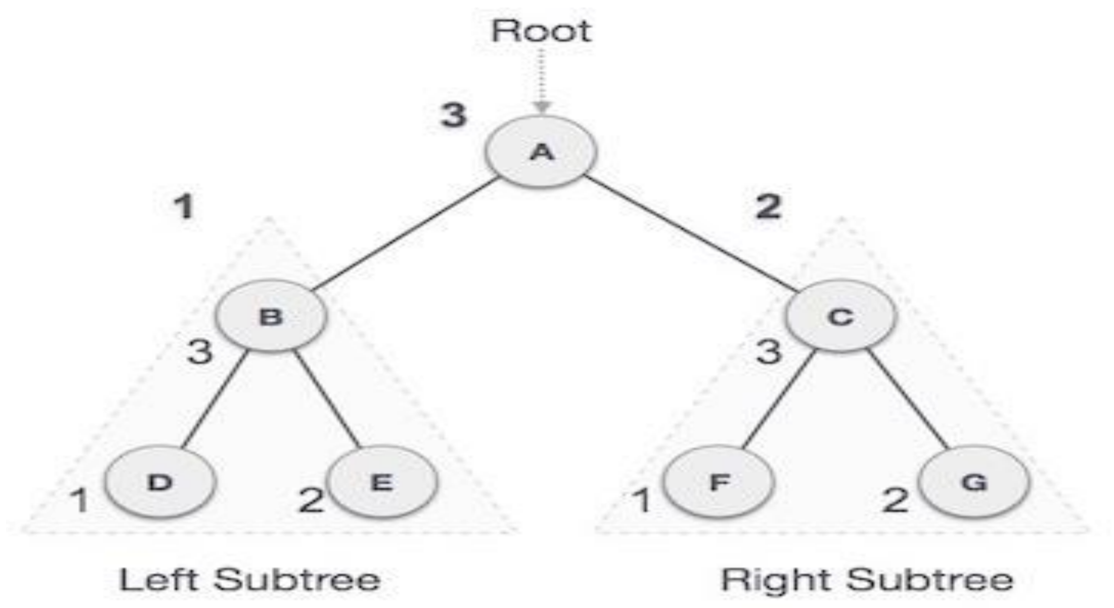
A → B → D → E → C → F → G

Algorithm

```
Until all nodes are traversed −
Step 1 − Visit root node.
Step 2 − Recursively traverse left subtree.
Step 3 − Recursively traverse right subtree.
```

Post-order Traversal

In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.



We start from **A**, and following Post-order traversal, we first visit the left subtree **B**. **B** is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be −

D → E → B → F → G → C → A

Algorithm

```
Until all nodes are traversed −
Step 1 − Recursively traverse left subtree.
Step 2 − Recursively traverse right subtree.
Step 3 − Visit root node.
```

(tutorialspoint, 2015)

Breadth First Search (BFS)

Graph traversals

Graph traversal means visiting every vertex and edge exactly once in a well-defined order. While using certain graph algorithms, you must ensure that each vertex of the graph is visited exactly once. The order in which the vertices are visited are important and may depend upon the algorithm or question that you are solving.

During a traversal, it is important that you track which vertices have been visited. The most common way of tracking vertices is to mark them.

Breadth First Search (BFS)

There are many ways to traverse graphs. BFS is the most commonly used approach.

BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layer wise thus exploring the neighbor nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbor nodes.

As the name BFS suggests, you are required to traverse the graph breadthwise as follows:

1. First move horizontally and visit all the nodes of the current layer
2. Move to the next layer

Consider the following diagram.

The distance between the nodes in layer 1 is comparatively lesser than the distance between the nodes in layer 2. Therefore, in BFS, you must traverse all the nodes in layer 1 before you move to the nodes in layer 2.

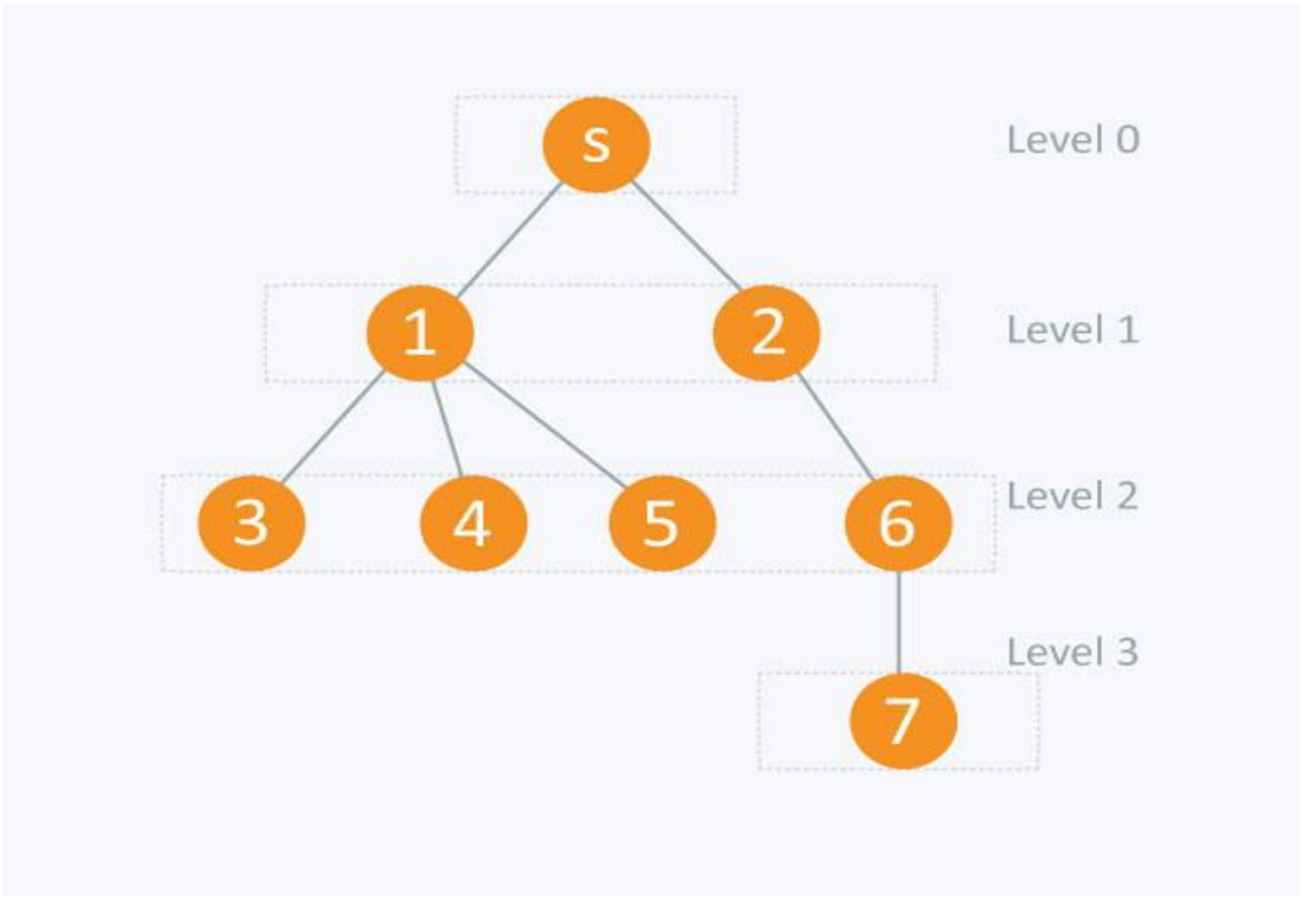
Traversing child nodes

A graph can contain cycles, which may bring you to the same node again while traversing the graph. To avoid processing of same node again, use a Boolean array which marks the node after it is processed. While visiting the nodes in the layer of a graph, store them in a manner such that you can traverse the corresponding child nodes in a similar order.

In the earlier diagram, start traversing from 0 and visit its child nodes 1, 2, and 3. Store them in the order in which they are visited. This will allow you to visit the child nodes of 1 first (i.e. 4 and 5), then of 2 (i.e. 6 and 7), and then of 3 (i.e. 7) etc.

To make this process easy, use a queue to store the node and mark it as 'visited' until all its neighbors (vertices that are directly connected to it) are marked. The queue follows the First In First Out (FIFO) queuing method, and therefore, the neighbors of the node will be visited in the order in which they were inserted in the node i.e. the node that was inserted first will be visited first, and so on.

In this code, while you visit each node, the level of that node is set with an increment in the level of its parent node. This is how the level of each node is determined.



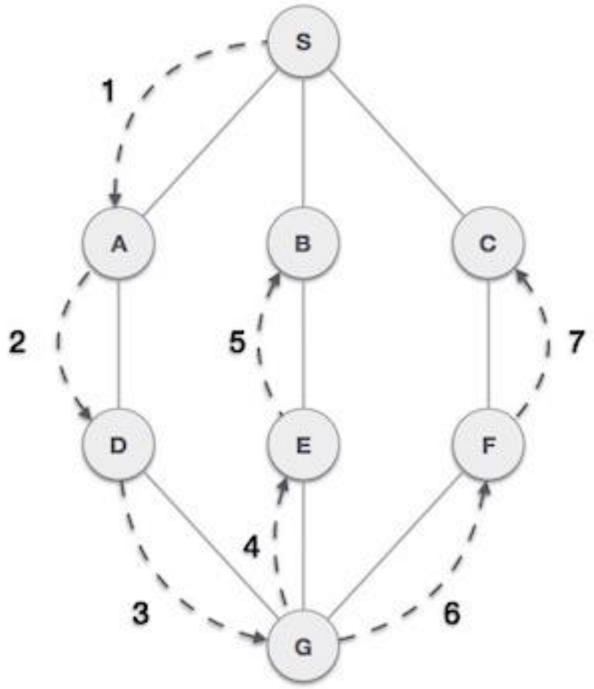
```
node                level [node]

    s (source node)        0
    1               1
    2               1
    3               2
    4               2
    5               2
    6               2
    7               3
```

(hackerearth, 2016)

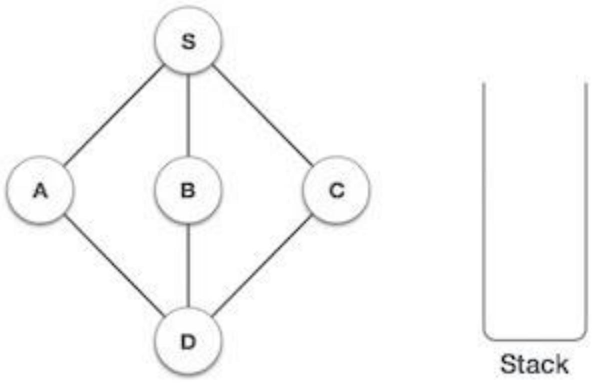
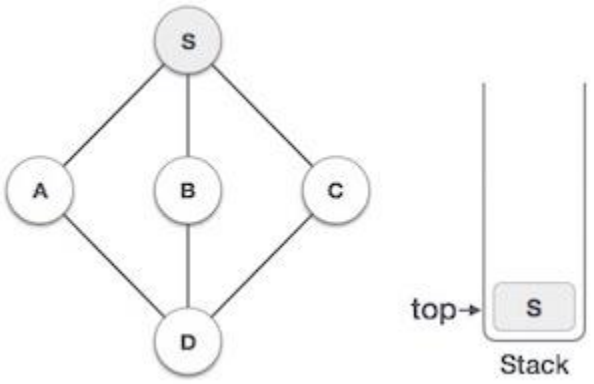
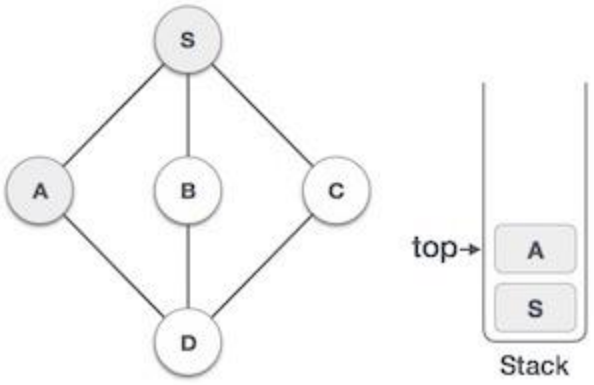
Depth First Search (DFS)

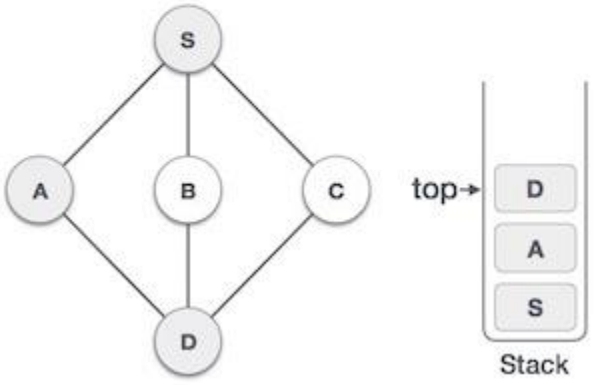
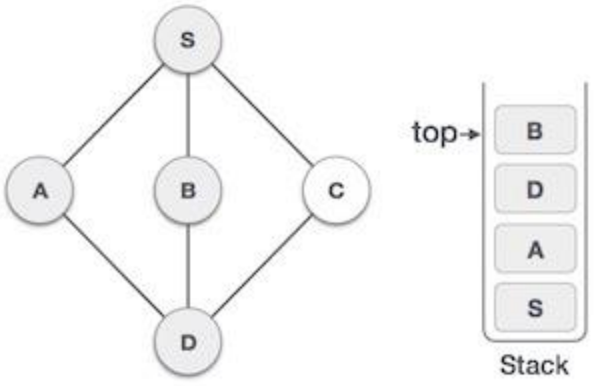
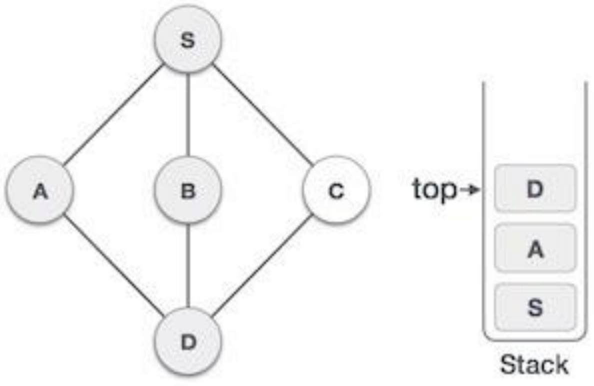
Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

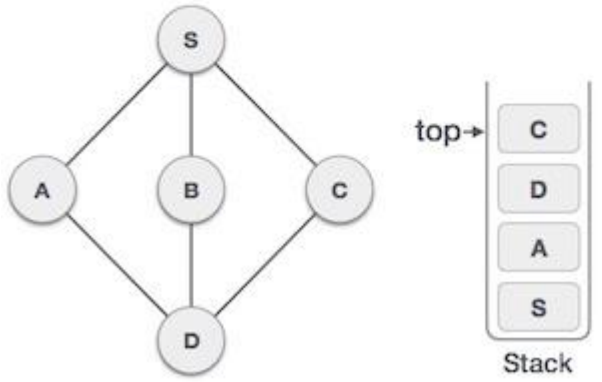


As in the example given above, DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to C. It employs the following rules.

- **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.

(tutorialspoint, 2015)

Sorting

Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.

The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats. Following are some of the examples of sorting in real-life scenarios −

- **Telephone Directory** − The telephone directory stores the telephone numbers of people sorted by their names, so that the names can be searched easily.
- **Dictionary** − The dictionary stores words in an alphabetical order so that searching of any word becomes easy.
-

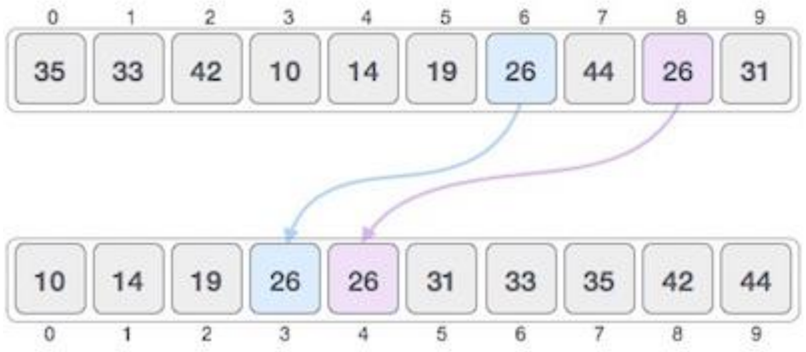
In-place Sorting and Not-in-place Sorting

Sorting algorithms may require some extra space for comparison and temporary storage of few data elements. These algorithms do not require any extra space and sorting is said to happen in-place, or for example, within the array itself. This is called **in-place sorting**. Bubble sort is an example of in-place sorting.

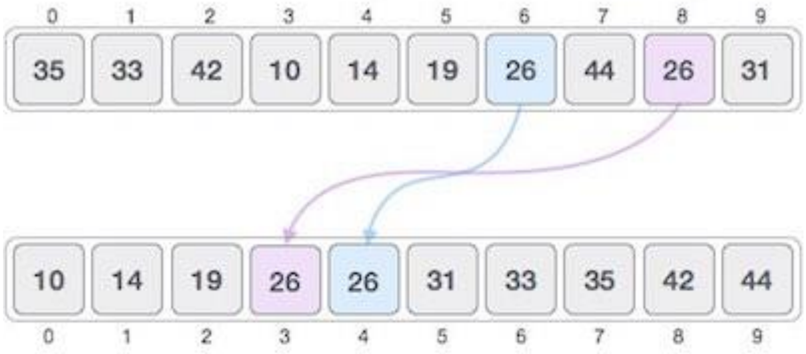
However, in some sorting algorithms, the program requires space which is more than or equal to the elements being sorted. Sorting which uses equal or more space is called **not-in-place sorting**. Merge-sort is an example of not-in-place sorting.

Stable and Not Stable Sorting

If a sorting algorithm, after sorting the contents, does not change the sequence of similar content in which they appear, it is called **stable sorting**.



If a sorting algorithm, after sorting the contents, changes the sequence of similar content in which they appear, it is called **unstable sorting**.



Stability of an algorithm matters when we wish to maintain the sequence of original elements, like in a tuple for example.

Adaptive and Non-Adaptive Sorting Algorithm

A sorting algorithm is said to be adaptive, if it takes advantage of already 'sorted' elements in the list that is to be sorted. That is, while sorting if the source list has some element already sorted, adaptive algorithms will take this into account and will try not to re-order them.

A non-adaptive algorithm is one which does not take into account the elements which are already sorted. They try to force every single element to be re-ordered to confirm their sortedness.

Important Terms

Some terms are generally coined while discussing sorting techniques, here is a brief introduction to them −

Increasing Order

A sequence of values is said to be in **increasing order**, if the successive element is greater than the previous one. For example, 1, 3, 4, 6, 8, 9 are in increasing order, as every next element is greater than the previous element.

Decreasing Order

A sequence of values is said to be in **decreasing order**, if the successive element is less than the current one. For example, 9, 8, 6, 4, 3, 1 are in decreasing order, as every next element is less than the previous element.

Non-Increasing Order

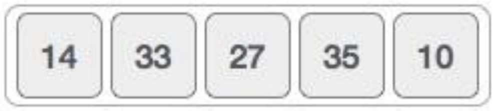
A sequence of values is said to be in **non-increasing order**, if the successive element is less than or equal to its previous element in the sequence. This order occurs when the sequence contains duplicate values. For example, 9, 8, 6, 3, 3, 1 are in non-increasing order, as every next element is less than or equal to (in case of 3) but not greater than any previous element.

Non-Decreasing Order

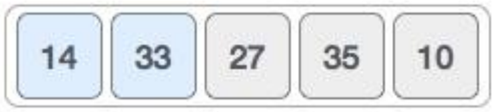
A sequence of values is said to be in **non-decreasing order**, if the successive element is greater than or equal to its previous element in the sequence. This order occurs when the sequence contains duplicate values. For example, 1, 3, 3, 6, 8, 9 are in non-decreasing order, as every next element is greater than or equal to (in case of 3) but not less than the previous one.

How Bubble Sort Works?

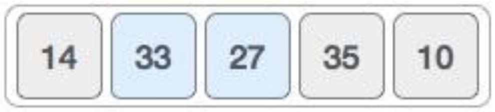
We take an unsorted array for our example. Bubble sort takes $O(n^2)$ time so we're keeping it short and precise.



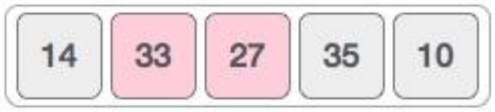
Bubble sort starts with very first two elements, comparing them to check which one is greater.



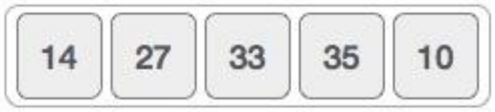
In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.



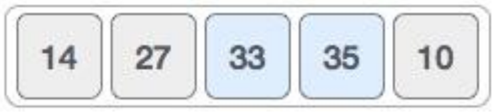
We find that 27 is smaller than 33 and these two values must be swapped.



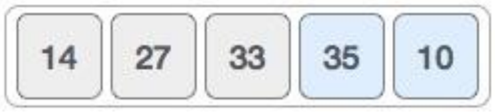
The new array should look like this −



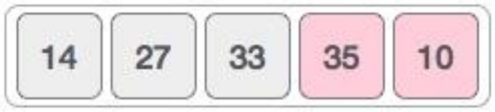
Next we compare 33 and 35. We find that both are in already sorted positions.



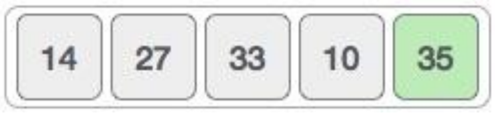
Then we move to the next two values, 35 and 10.



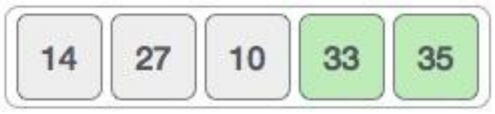
We know then that 10 is smaller 35. Hence they are not sorted.



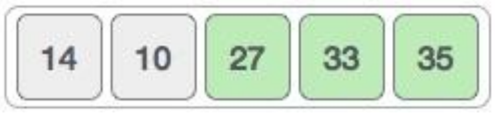
We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −



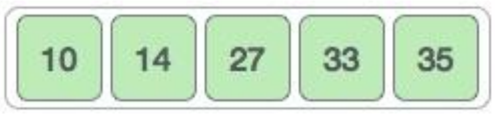
To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this −



Notice that after each iteration, at least one value moves at the end.



And when there's no swap required, bubble sorts learns that an array is completely sorted.



Now we should look into some practical aspects of bubble sort.

Algorithm

We assume **list** is an array of **n** elements. We further assume that **swap**function swaps the values of the given array elements.

We observe in algorithm that Bubble Sort compares each pair of array element unless the whole array is completely sorted in an ascending order. This may cause a few complexity issues like what if the array needs no more swapping as all the elements are already ascending.

To ease-out the issue, we use one flag variable **swapped** which will help us see if any swap has happened or not. If no swap has occurred, i.e. the array requires no more processing to be sorted, it will come out of the loop.

Pseudocode of Bubble Sort algorithm can be written as follows −

(tutorialspoint, 2015)

References:

c4learn. (2013, 1 20). C Array Types. Retrieved 10 15, 2018, from c4learn: http://www.c4learn.com/c-programming/c-array-types/

hackerearth. (2016, 10 1). Breadth First Search. Retrieved 10 28, 2018, from hackerearth: https://www.hackerearth.com/practice/algorithms/graphs/breadth-first-search/tutorial/

Studytonight. (2015, 1 10). C LANGUAGE. Retrieved 10 15, 2018, from Studytonight: https://www.studytonight.com/c/arrays-in-c.php

thecrazyprogrammer. (2015, 6 15). Retrieved 10 28, 2018, from thecrazyprogrammer: https://www.thecrazyprogrammer.com/2014/03/depth-first-search-dfs-traversal-of-a-graph.html

tutorialspoint. (2015, 6 20). Data Structure and Algorithms - Linked List. Retrieved 10 15, 2018, from tutorialspoint: https://www.tutorialspoint.com/data_structures_algorithms/linked_list_algorithms.htm