**What is Data Structures?**

* In computer science, a **data structure** is a particular way of storing and organizing data in a computer so that it can be used efficiently.
* **Data Structures a**re generally **based** on the ability of a computer to fetch (retrieve) and store data at any place in its memory, specified by an address – a bit string that can be itself stored in memory and manipulated by the program.

#### ****How is it implemented?****

Data structure usually requires writing a set of procedures that create and manipulate instances of that structure.

**Classification of Data Structure According to Type**

* **Primitive** – these are basic data structures and are directly operated upon machine instructions, e.g., integer, character.
* **Non-primitive** – these are derived from primitive data structures, e.g., array.

**Classification of Data Structure According to Elements**

* **Homogeneous** – in this data structure, all elements are of the same type, e.g. array.
* **Heterogeneous** – in this data structure, elements are of different types, e.g., structure.

**Classification of Data Structure According to Size**

* **Static** – the size of this data structure cannot be changed after the initial allocation, like matrices.
* **Dynamic** – the size can change dynamically, like in lists.

**Classification of Data Structure According to Relationship**

* **Linear** – this data structure maintains a linear relationship between its elements, e.g., array.
* **Non-linear**– this data structure does not maintain any linear relationship between its elements, e.g., in a tree.

**Memory**

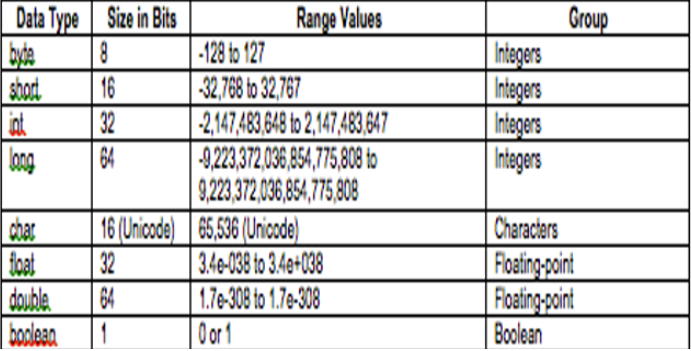
* **Main memory (RAM)** – where instructions (programs) and data are stored; volatile means may be lost once the computer is powered down.
* **Cache memory** in the central processing unit (CPU) – is used to store frequently used instructions and data that either is, will be, or has been used by the CPU. A segment of the CPU’s cache memory is called a register ( a small amount of memory that is used to temporarily store instructions and data.
* **Persistent storage** (external storage devices) – used to store instructions and data in external devices such as a hard disk; non-volatile; commonly used by the operating system as virtual memory (a technique an OS uses to increase the main memory capacity beyond the RAM).
  + Note: *as to access – CPU cache memory is the type of memory that has the fastest access speed, followed by the main memory, and lastly persistent storage because it usually involves a mechanical process that inhibits the quick transfer of instructions and data.*
* **Reserving Memory**
  + Data used by a program is stored in memory and manipulated by various data structure techniques, depending on the nature of the program.
  + Memory is organized into groups of eight bits called a byte, enabling 256 combinations of zeroes and ones that can store numbers from 0 through 255.
  + Data used in a program can be larger than a byte and requires 2, 4, or 8 bytes to be stored in memory. Before any data can be stored in memory, you must tell the computer how much space to reserve for data by using an abstract data type.

#### ****What is an Abstract Data Type(ADT)?****

It is a **keyword** of a programming language that **specifies the amount of memory needed to store data** and **the kind of data that will be stored in that memory location**.

The number of bytes reserved for an ADT varies, depending on the programming language used to write the program and the type of computer used to compile the program.

**Example of Abstract Data Type(ADT)?**

* Java ADT has a fixed size in order for programs to run in all Java runtime environments (JRE).
* C and C++ ADT have sizes based on the register size of the computer used to compile the program. The int and float data types are the size of the register.

**Abstract Data Type Groups**

* Integer
* Floating-point
* Character
* Boolean

**ADT - Integer**

**Integer** – stores whole numbers and signed numbers.

* **byte ADT**(Java reserves 8 bits of main memory)

–It is the smallest ADT in the integer group and is declared by using the keyword byte. It is used when sending data to and receiving data from a file or across a network. Choose a ‘byte’ whenever you need to move data to and from a file or across a network.

* **short ADT**(Java reserves 16 bits of main memory)

–It is ideal for use in programs that run on 16-bit computers and is the least used integer ADT. Choose a ‘short’ if you ever need to store an integer in a program that runs on a very old computer.

* **int ADT**(Java reserves 32 bits of main memory)

–It is the most frequently used ADT of the integer group for a number of reasons. Choose an ‘int’: for control variables in control loops in array indexes when performing integer math

* **long ADT**(Java reserves 64-bits of main memory)

–It is used whenever using whole numbers are beyond the range of an int data type.

**ADT – Floating Point**

**Floating-point** – stores real numbers (fractional values). A *real number* contains a decimal value. The precision of a number is the number of places after the decimal point that contains an accurate value.

* **float ADT**(Java reserves 32-bits of main memory)

–It used for real numbers that require single precision. Single precision means the value is precise up to 7 digits to the right of the decimal. Choose a ‘float’ whenever you need to store a decimal value where only 7 digits to the right of the decimal must be accurate.

* **double ADT**(Java reserves 64-bits of main memory)

–It is used to store real numbers that are very large or very small and require double the amount of memory that is reserved with a float ADT. Choose a ‘double’ whenever you need to store a decimal value where more than 7 digits to the right of the decimal must be accurate.

**ADT – Character**

* **char ADT**(Java reserves 16-bits of main memory)

–It is represented as an integer value that corresponds to a character set. A character set assigns an integer value to each character, punctuation, and symbol used in a language.

* + - Ex: the letter ‘A’ is stored in memory as the value 65, which corresponds to the letter ‘A’ in a character set.

–The keyword ‘char’ tells the computer that the integer stored in that memory location is treated as a character and not a number. There are two types of character sets:

* ASCII (American Standard Code for Information Interchange) uses a byte to represent a maximum of 256 characters of a language (English)
* Unicode

–used for languages (Russian, Arabic, Japanese, Chinese) that have more than 256 characters. It uses 2 bytes to represent each character.

NOTE: Choose a ‘char’ whenever you need to store a single character in memory.

**ADT – Boolean**

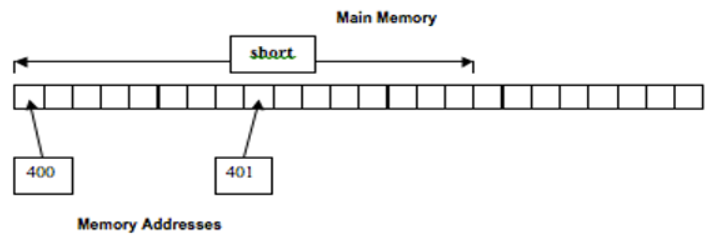
**Boolean**– stores a true or false value. The correct choice for storing a yes or no or true or false response to a question.

* **Boolean ADT**(Java reserves 1-bit of main memory)

–It reserves memory to store a ‘boolean’ value (true or false) represented as a zero or one. Choose a ‘boolean’ whenever you need to store one of two possibilities in memory.

#### ****Memory Addresses****

Imagine the main memory as a series of endless boxes organized into groups of eight. Each box holds a zero or one. Each group of eight boxes (1 byte) is assigned a unique number called a memory address.



#### ****How do Memory Address works?****

A memory address is indirectly or directly used within a program to access all eight boxes. The program tells the computer that it wants to copy data stored in memory location 401 – that is, the box whose address is 401. The computer goes to that memory location and copies the data (zero or one) from box 401 and copies data from the next seven boxes. Those next seven boxes do not have a memory address. They share the memory address of box 401.

#### ****Abstract Data Types and Memory Addresses****

Some ADT reserve memory in a size that is greater than 1 byte, for example, Java reserves 2 bytes of memory for a ‘short’ ADT (16-bits of main memory).

#### ****Algorithm****

An algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

**Basic Categories of Algorithm**

* **Search**− Algorithm to search an item in a data structure.
* **Sort**− Algorithm to sort items in a certain order.
* **Insert**− Algorithm to insert an item in a data structure.
* **Update**− Algorithm to update an existing item in a data structure.
* **Delete**− Algorithm to delete an existing item from a data structure.

**Characteristics of an Algorithm**

Not all procedures can be called an algorithm. An algorithm should have the following characteristics.

* **Unambiguous**− The algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.
* **Input**− An algorithm should have 0 or more well-defined inputs.
* **Output**− An algorithm should have 1 or more well-defined outputs and should match the desired output.
* **Finiteness**− Algorithms must terminate after a finite number of steps.
* **Feasibility**− Should be feasible with the available resources.
* **Independent**− An algorithm should have step-by-step directions, which should be independent of any programming code.

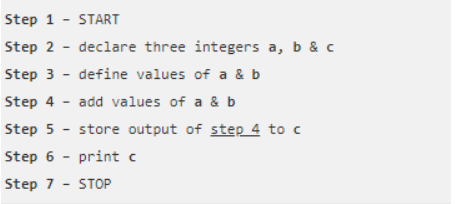
#### ****How to Write an Algorithm?****

There are no well-defined standards for writing algorithms. Rather, it is problem and resource-dependent. Algorithms are never written to support a particular programming code.

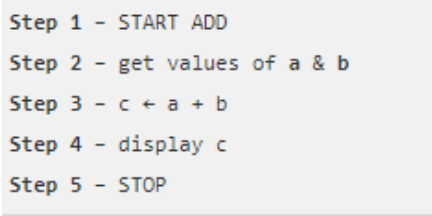
As we know that all programming languages share basic code constructs like loops (do, for, while), flow-control (if-else), etc. These common constructs can be used to write an algorithm.

We write algorithms in a step-by-step manner, but it is not always the case. Algorithm writing is a process and is executed after the problem domain is well-defined. That is, we should know the problem domain, for which we are designing a solution.

**Problem:**Design an algorithm to add two numbers and display the result.



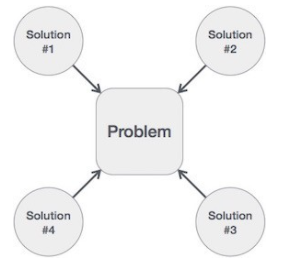
Algorithms tell the programmers how to code the program. Alternatively, the algorithm can be written as –



In the design and analysis of algorithms, usually, the second method is used to describe an algorithm. It makes it easy for the analyst to analyze the algorithm ignoring all unwanted definitions. Observe what operations are being used and how the process is flowing.

Writing **step numbers** is optional.

Design an algorithm to get a solution to a given problem. A problem can be solved in more than one way.



Hence, many solution algorithms can be derived for a given problem. The next step is to analyze those proposed solution algorithms and implement the best suitable solution.

**Algorithm Analysis**

The efficiency of an algorithm can be analyzed at two different stages, before implementation, and after implementation. They are the following

* + **A Priori Analysis**− This is a theoretical analysis of an Efficiency of an algorithm is measured by assuming that all other factors, for example, processor speed, are constant and have no effect on the implementation.
  + **A Posterior Analysis**− This is an empirical analysis of an algorithm. The selected algorithm is implemented using a programming language. This is then executed on the target computer In this analysis, actual statistics like running time and space required, are collected.

#### ****Algorithm Complexity****

Algorithm analysis deals with the execution or running time of various operations involved. The running time of an operation can be defined as the number of computer instructions executed per operation.

Suppose **X** is an algorithm and **n** is the size of input data, the time and space used by the algorithm X are the two main factors, which decide the efficiency of X.

**Time Factor** − Time is measured by counting the number of key operations such as comparisons in the sorting

**Space Factor** − Space is measured by counting the maximum memory space required by the

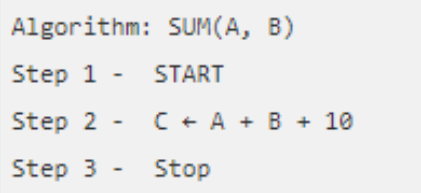
The complexity of an algorithm **f(n)** gives the running time and/or the storage space required by the algorithm in terms of **n** as the size of input data.

**Space Complexity**

The space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle. The space required by an algorithm is equal to the sum of the following two components −

* A fixed part is a space required to store certain data and variables, that are independent of the size of the problem. For example, simple variables and constants used, program size,
* A variable part is a space required by variables, whose size depends on the size of the, for example, dynamic memory allocation, recursion stack space, etc.

Space complexity S(P) of any algorithm P is S(P) = C + SP(I), where C is the fixed part and S(I) is the variable part of the algorithm, which depends on instance characteristic I. Following is a simple example that tries to explain the concept



Here we have three variables A, B, and C and one constant. Hence S(P) = 1 + 3. Now, space depends on data types of given variables and constant types and it will be multiplied accordingly.

#### ****Time Complexity****

The time complexity of an algorithm represents the amount of time required by the algorithm to run to completion. Time requirements can be defined as a numerical function T(n), where T(n) can be measured as the number of steps, provided each step consumes constant time.

For example, the addition of two n-bit integers takes **n**steps. Consequently, the total computational time is T(n) = c ∗ n, where c is the time taken for the addition of two bits. Here, we observe that T(n) grows linearly as the input size increases.

**Asymptotic analysis**

Asymptotic analysis of an algorithm refers to defining the mathematical foundation/framing of its run-time performance. Using asymptotic analysis, we can very well conclude the best case, average case, and worst-case scenario of an algorithm.

 Asymptotic analysis is input bound i.e. if there's no input to the algorithm, it is concluded to work in constant time. Other than the "input" all other factors are considered constant.

Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation. For example, the running time of one operation is computed as *f*(n), and maybe for another operation, it is computed as *g*(n2). This means the first operation running time will increase linearly with the increase in **n**and the running time of the second operation will increase exponentially when **n**increases. Similarly, the running time of both operations will be nearly the same if **n**is significantly small.

The time required by an algorithm falls under three types −

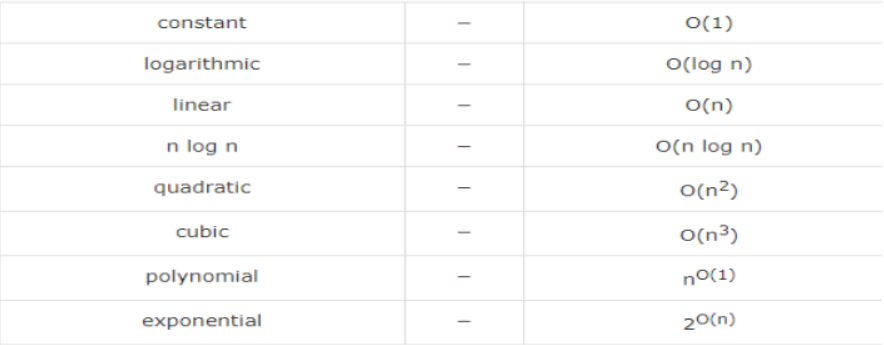
* **Best Case**− Minimum time required for program
* **Average Case**− Average time required for program
* **Worst Case**− Maximum time required for program

#### The notation Ο(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst-case time complexity or the longest amount of time an algorithm can possibly take to complete.

The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.

The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time. It is represented as follows –

#### ****Common Asymptotic Notation****



#### ****Asymptotic Notation****

The running time of an algorithm depends on how long it takes a computer to run the lines of code of the algorithm—and that depends on the speed of the computer, the programming language, and the compiler that translates the program from the programming language into code that runs directly on the computer, among other factors.  
  
Let's think about the running time of an algorithm more carefully. We can use a combination of two ideas. First, we need to determine how long the algorithm takes, in terms of the size of its input. This idea makes intuitive sense, doesn't it? We've already seen that the maximum number of guesses in linear search and binary search increases as the length of the array increases. Or think about a GPS. If it knew about only the interstate highway system, and not about every little road, it should be able to find routes more quickly, right? So, we think about the running time of the algorithm as a function of the size of its input.

ArrayList

#### ****ArrayList****

ArrayList is a part of collection framework and is present in java.util package.  It provides a dynamic way of manipulating data.  Though, it may be slower than standard arrays but can be helpful in programs where lots of manipulation in the array is needed.  Some of the notable characteristics of ArrayList are the following:

* ArrayList inherits AbstractList class and implements List interface.
* ArrayList is initialized by size, however, the size can increase if collection grows or shrunk if objects are removed from the collection.
* Java ArrayList allows us to randomly access the list.
* Use a wrapper class if an ArrayList can not use primitive types, like int, char, etc.
* ArrayList in Java can be seen as similar to a vector in C++.

#### ****ArrayList Constructors****

1. **ArrayList()**: This constructor is used to build an empty array list
2. **ArrayList(Collection c)**: This constructor is used to build an array list initialized with the elements from collection c
3. **ArrayList(int capacity)**: This constructor is used to build an array list with initial capacity being specified

#### ****Basic Structure of an ArrayList****

ArrayList<Integer> arrayList1 = new ArrayList<Integer>();  
  
ArrayList<String> listarrayList2=new ArrayList<String>();  
  
ArrayList<Boolean> arrayList3 = new ArrayList<Boolean>();

#### ****ArrayList Methods****

|  |  |
| --- | --- |
| add(int index, Object element) | This method is used to insert a specific element at a specific position index in a list. |
| add(Object o) | This method is used to append a specific element to the end of a list. |
| addAll(Collection C) | This method is used to append all the elements from a specific collection to the end of the mentioned list, in such an order that the values are returned by the specified collection’s iterator. |
| addAll(int index, Collection C) | Used to insert all of the elements starting at the specified position from a specific collection into the mentioned list. |
| clear() | This method is used to remove all the elements from any list. |
| clone() | This method is used to return a shallow copy of an ArrayList. |
| contains? (Object o) | Returns true if this list contains the specified element. |
| ensureCapacity?(int minCapacity) | Increases the capacity of this ArrayList instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument. |
| forEach?(Consumer<? super E> action) | Performs the given action for each element of the Iterable until all elements have been processed or the action throws an exception. |
| get?(int index) | Returns the element at the specified position in this list. |
| indexOf(Object O) | The index the first occurrence of a specific element is either returned or -1 in case the element is not in the list. |
| isEmpty?() | Returns true if this list contains no elements. |
| lastIndexOf(Object O) | The index of the last occurrence of a specific element is either returned or -1 in case the element is not in the list. |
| listIterator?() | Returns a list iterator over the elements in this list (in proper sequence). |
| listIterator?(int index) | Returns a list iterator over the elements in this list (in proper sequence), starting at the specified position in the list. |
| remove?(int index) | Removes the element at the specified position in this list. |
| remove?(Object o) | Removes the first occurrence of the specified element from this list, if it is present. |
| removeAll?(Collection c) | Removes from this list all of its elements that are contained in the specified collection. |
| removeIf?(Predicate filter) | Removes all of the elements of this collection that satisfy the given predicate. |
| removeRange?(int fromIndex, int toIndex) | Removes from this list all of the elements whose index is between fromIndex, inclusive, and toIndex, exclusive. |
| retainAll?(Collection<?> c) | Retains only the elements in this list that are contained in the specified collection. |
| set?(int index, E element) | Replaces the element at the specified position in this list with the specified element. |
| size?() | Returns the number of elements in this list. |
| spliterator?() | Creates a late-binding and fail-fast Spliterator over the elements in this list. |
| subList?(int fromIndex, int toIndex) | Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive. |
| toArray() | This method is used to return an array containing all of the elements in the list in the correct order. |
| toArray(Object[] O) | It is also used to return an array containing all of the elements in this list in the correct order same as the previous method. |
| trimToSize() | This method is used to trim the capacity of the instance of the ArrayList to the list’s current size. |

# Stack

**A stack** is a way to group things together by placing one thing on top of another and then removing things one at a time from the top of the stack.

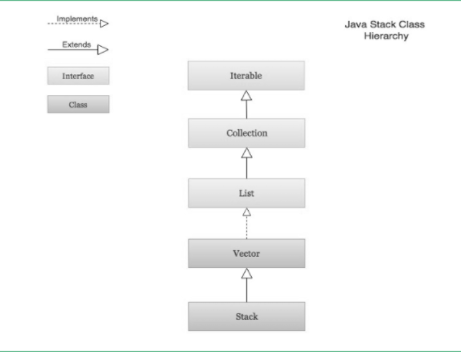
In computer science, a **stack** is **a Last-In, First-Out (L I F O)** abstract data type and data structure. A stack can have any abstract data type as an element but is characterized by only two fundamental operations: push and pop.

The **push**operation adds to the top of the list, hiding any items already on the stack, or initializing the stack if it is empty. The **pop**operation removes an item from the top of the list and returns this value to the caller. A pop either reveals previously concealed items or results in an empty list.

#### ****Stack as Data Structure****

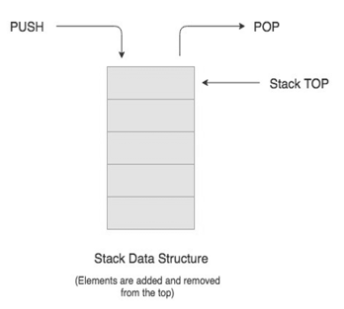
A **stack**is a restricted data structure because only a small number of operations are performed on it. The nature of the pop and push operations also means that stack elements have a natural order. Elements are removed from the stack in the reverse order to the order of their addition: therefore, the lower elements are typically those that have been in the list the longest.

A stack is a linear data structure that follows a particular order in which the operations are performed.  It is a part of Java’s collections framework. The following figure shows the class hierarchy of the Stack.



The Stack class extends Vector which implements the List interface.  A Vector is a re-sizable collection.  It grows its size to accommodate new elements and shrinks the size when the elements are removed.  Since the Stack class extends Vector; it also grows and shrinks its size as needed when new elements are added or removed.

#### ****Visualization of a Stack****



#### ****Real-life Example of a Stack****

Consider an example of plates stacked over one another in the canteen. The plate which is at the top is the first one to be removed, i.e. the plate which has been placed at the bottommost position remains in the stack for the longest period of time. So, it can be simply seen to follow LIFO(Last In First Out)/FILO(First In Last Out) order.

#### ****Creation of Stack****

Stack class is used by importing the **java.util library**.  A Stack object is created using the following structure.

Stack<BaseType> s = new Stack<BaseType>();

Just like ArrayList, Stack requires a base type to be an object

#### ****Stack Operations****

The **push( )** operation is used both to initialize the stack and to store values to it. It is responsible for inserting (copying) the value into the array and for incrementing the element counter (size). In a responsible implementation, it is also necessary to check whether the array is already full to prevent an overrun.

It is an operation used to insert or add a data item or element to the stack. The push ( ) method requires an item when called.  The push operation is used for insertion of new elements in the Stack.

The **pop( )** operation is responsible for removing a value from the stack, and decrementing the value of size. A responsible implementation will also need to check that the array is not already empty.

It is an operation used to delete or remove a data item or element at the top of the stack. The pop ( ) method returns the item being removed in the Stack.  The pop operation is used for the deletion of the top element of the Stack

The **size( )**operation is an operation to determine the size (number of items) of the Stack. It is used mainly to control loops. The size operation is used for checking the size of a Stack.

The **peek( )** operation is a method that looks at the item at the top of a stack. The peek ( ) method returns the item at the top without removing it.  The peek operation is used to determine what item is at the top of the Stack

The **search ( )** member method is a method that returns the position (in number) of an item from the top of a stack. The search ( ) method requires the desired item when called.  The search operation is used to determine the position of the item.

The **empty ( )** member method is a method that tests if a stack object is empty or not. The empty ( ) method returns either a boolean value of true or false. The method returns true if the stack is empty and false if it still contains an item or element.  The empty operation is used for checking if the Stack is empty.

#### ****Infix to Postfix Conversion Using Stack****

One of the applications of Stack is in the conversion of arithmetic expressions in high-level programming languages into machine-readable form.  As our computer system can only understand and work on a binary language, it assumes that an arithmetic operation can take place in two operands only e.g., **A+B, C\*D, D/A,** etc.

But in our usual form, an arithmetic expression may consist of more than one operator and two operands e.g. **(A+B)\*C(D/(J+D))**.

These complex arithmetic operations can be converted into polish notation using stacks which then can be executed in two operands and an operator form.

#### ****Infix Expression****

Infix expression follows the scheme of **<operand><operator><operand>** i.e. an <operator> is preceded and succeeded by an <operand>.  The expression of the form **a op b**.  When an operator is in-between every pair of operands.  Such an expression is termed infix expression, e.g., **A+B**.

#### ****Postfix Expression****

Postfix expression follows the scheme of **<operand><operand><operator>** i.e. an <operator> is succeeded by both the <operand>.  The expression of the form **a b op**. When an operator is followed for every pair of an operand, e.g., **AB+**.

#### ****Why Postfix Representation of the Expression is Important?****

The compiler scans the expression either from left to right or from right to left.  Consider the below expression:

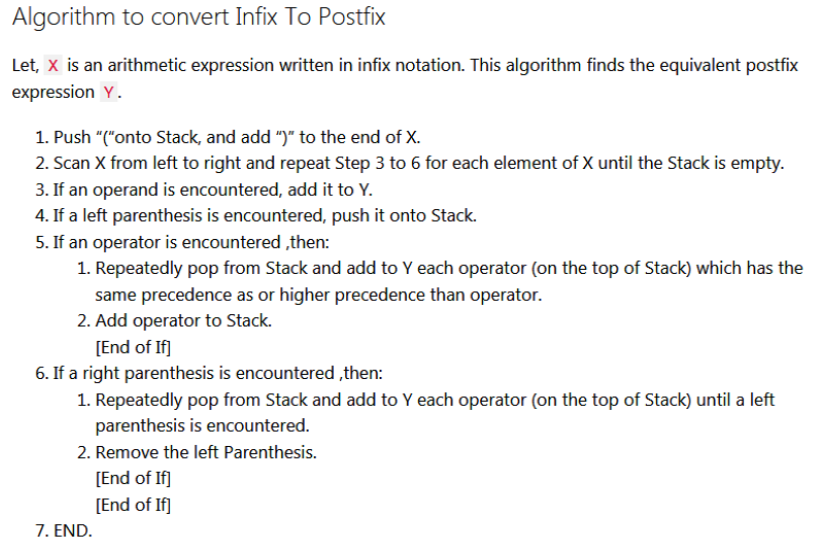
a op1 b op2 c op3 d  
  
If op1 = +, op2 = \*, op3 = +

The compiler first scans the expression to evaluate the expression**b \* c**, then again scan the expression to add a to it. The result is then added to d after another scan.  The repeated scanning makes it very in-efficient. It is better to convert the expression to postfix(or prefix) form before evaluation.  The corresponding expression in the postfix form is **abc\*+d+**. The postfix expressions can be evaluated easily using a stack.

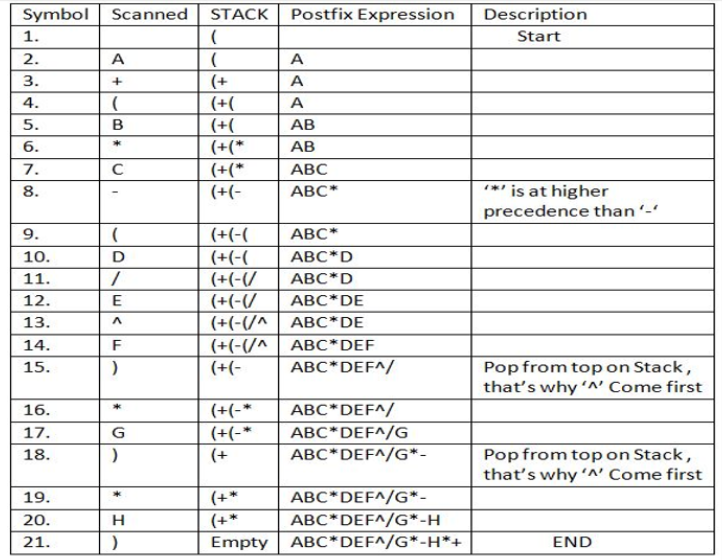
**Algorithm to convert Infix To Postfix**

1. Scan the infix expression from left to right.
2. If the scanned character is an operand, output it.
3. Else,
   * If the precedence of the scanned operator is greater than the precedence of the operator in the stack(or the stack is empty or the stack contains a ‘(‘ ), push it.
   * Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)
4. If the scanned character is an ‘(‘, push it to the stack.
5. If the scanned character is an ‘)’, pop the stack and output it until a ‘(‘ is encountered, and discard both the parenthesis.
6. Repeat steps 2-6 until infix expression is scanned.
7. Print the output
8. Pop and output from the stack until it is not empty.

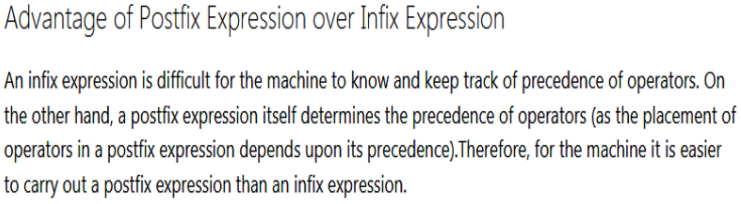
#### ****Algorithm to convert Infix To Postfix****



#### ****Conversion of Infix to Postfix****



Advantage of postfix expression over infix expression



**Prefix to Infix Conversion**

* **Infix**: An expression is called the Infix expression if the operator appears in between the operands in the expression. Simply of the form (operand1 operator operand2).

Example : (A+B) \* (C-D)

* **Prefix**: An expression is called the prefix expression if the operator appears in the expression before the operands. Simply of the form (operator operand1 operand2).

Example : \*+AB-CD (Infix : (A+B) \* (C-D) )

Given a Prefix expression, convert it into an Infix expression.  Computers usually do the computation in either prefix or postfix (usually postfix).  But for humans, its easier to understand an Infix expression rather than a prefix.  Hence conversion is needed for human understanding.

