Data Structure and Algorithms Tutorial

Top of Form

[PDF Version](https://www.tutorialspoint.com/data_structures_algorithms/data_structures_algorithms_pdf_version.htm)  [Quick Guide](https://www.tutorialspoint.com/data_structures_algorithms/dsa_quick_guide.htm)  [Resources](https://www.tutorialspoint.com/data_structures_algorithms/dsa_useful_resources.htm)  Job Search  [Discussion](https://www.tutorialspoint.com/data_structures_algorithms/dsa_discussion.htm)

Bottom of Form

Data Structures are the programmatic way of storing data so that data can be used efficiently. Almost every enterprise application uses various types of data structures in one or the other way. This tutorial will give you a great understanding on Data Structures needed to understand the complexity of enterprise level applications and need of algorithms, and data structures.

Why to Learn Data Structure and Algorithms?

As applications are getting complex and data rich, there are three common problems that applications face now-a-days.

* **Data Search** − Consider an inventory of 1 million(106) items of a store. If the application is to search an item, it has to search an item in 1 million(106) items every time slowing down the search. As data grows, search will become slower.
* **Processor speed** − Processor speed although being very high, falls limited if the data grows to billion records.
* **Multiple requests** − As thousands of users can search data simultaneously on a web server, even the fast server fails while searching the data.

To solve the above-mentioned problems, data structures come to rescue. Data can be organized in a data structure in such a way that all items may not be required to be searched, and the required data can be searched almost instantly.

Applications of Data Structure and Algorithms

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.

From the data structure point of view, following are some important categories of algorithms −

* **Search** − Algorithm to search an item in a data structure.
* **Sort** − Algorithm to sort items in a certain order.
* **Insert** − Algorithm to insert 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.

The following computer problems can be solved using Data Structures −

* Fibonacci number series
* Knapsack problem
* Tower of Hanoi
* All pair shortest path by Floyd-Warshall
* Shortest path by Dijkstra
* Project scheduling

Audience

This tutorial is designed for Computer Science graduates as well as Software Professionals who are willing to learn data structures and algorithm programming in simple and easy steps.

After completing this tutorial you will be at intermediate level of expertise from where you can take yourself to higher level of expertise.

Prerequisites

Before proceeding with this tutorial, you should have a basic understanding of C programming language, text editor, and execution of programs, etc.

[Previous Page](https://www.tutorialspoint.com/data_structures_algorithms/index.htm) [Print Page](javascript:printPage();)

[Next Page](https://www.tutorialspoint.com/data_structures_algorithms/data_structure_overview.htm)

# Data Structures & Algorithms - Overview

Data Structure is a systematic way to organize data in order to use it efficiently. Following terms are the foundation terms of a data structure.

* **Interface** − Each data structure has an interface. Interface represents the set of operations that a data structure supports. An interface only provides the list of supported operations, type of parameters they can accept and return type of these operations.
* **Implementation** − Implementation provides the internal representation of a data structure. Implementation also provides the definition of the algorithms used in the operations of the data structure.

Characteristics of a Data Structure

* **Correctness** − Data structure implementation should implement its interface correctly.
* **Time Complexity** − Running time or the execution time of operations of data structure must be as small as possible.
* **Space Complexity** − Memory usage of a data structure operation should be as little as possible.

Need for Data Structure

As applications are getting complex and data rich, there are three common problems that applications face now-a-days.

* **Data Search** − Consider an inventory of 1 million(106) items of a store. If the application is to search an item, it has to search an item in 1 million(106) items every time slowing down the search. As data grows, search will become slower.
* **Processor speed** − Processor speed although being very high, falls limited if the data grows to billion records.
* **Multiple requests** − As thousands of users can search data simultaneously on a web server, even the fast server fails while searching the data.

To solve the above-mentioned problems, data structures come to rescue. Data can be organized in a data structure in such a way that all items may not be required to be searched, and the required data can be searched almost instantly.

Execution Time Cases

There are three cases which are usually used to compare various data structure's execution time in a relative manner.

* **Worst Case** − This is the scenario where a particular data structure operation takes maximum time it can take. If an operation's worst case time is ƒ(n) then this operation will not take more than ƒ(n) time where ƒ(n) represents function of n.
* **Average Case** − This is the scenario depicting the average execution time of an operation of a data structure. If an operation takes ƒ(n) time in execution, then m operations will take mƒ(n) time.
* **Best Case** − This is the scenario depicting the least possible execution time of an operation of a data structure. If an operation takes ƒ(n) time in execution, then the actual operation may take time as the random number which would be maximum as ƒ(n).

Basic Terminology

* **Data** − Data are values or set of values.
* **Data Item** − Data item refers to single unit of values.
* **Group Items** − Data items that are divided into sub items are called as Group Items.
* **Elementary Items** − Data items that cannot be divided are called as Elementary Items.
* **Attribute and Entity** − An entity is that which contains certain attributes or properties, which may be assigned values.
* **Entity Set** − Entities of similar attributes form an entity set.
* **Field** − Field is a single elementary unit of information representing an attribute of an entity.
* **Record** − Record is a collection of field values of a given entity.
* **File** − File is a collection of records of the entities in a given entity set.

[Previous Page](https://www.tutorialspoint.com/data_structures_algorithms/index.htm) [Print Page](javascript:printPage();)

[Next Page](https://www.tutorialspoint.com/data_structures_algorithms/data_structure_environment.htm)

# Data Structures - Environment Setup

## Local Environment Setup

If you are still willing to set up your environment for C programming language, you need the following two tools available on your computer, (a) Text Editor and (b) The C Compiler.

### Text Editor

This will be used to type your program. Examples of few editors include Windows Notepad, OS Edit command, Brief, Epsilon, EMACS, and vim or vi.

The name and the version of the text editor can vary on different operating systems. For example, Notepad will be used on Windows, and vim or vi can be used on Windows as well as Linux or UNIX.

The files you create with your editor are called source files and contain program source code. The source files for C programs are typically named with the extension "**.c**".

Before starting your programming, make sure you have one text editor in place and you have enough experience to write a computer program, save it in a file, compile it, and finally execute it.

### The C Compiler

The source code written in the source file is the human readable source for your program. It needs to be "compiled", to turn into machine language so that your CPU can actually execute the program as per the given instructions.

This C programming language compiler will be used to compile your source code into a final executable program. We assume you have the basic knowledge about a programming language compiler.

Most frequently used and free available compiler is GNU C/C++ compiler. Otherwise, you can have compilers either from HP or Solaris if you have respective Operating Systems (OS).

The following section guides you on how to install GNU C/C++ compiler on various OS. We are mentioning C/C++ together because GNU GCC compiler works for both C and C++ programming languages.

## Installation on UNIX/Linux

If you are using **Linux or UNIX**, then check whether GCC is installed on your system by entering the following command from the command line −

$ gcc -v

If you have GNU compiler installed on your machine, then it should print a message such as the following −

Using built-in specs.

Target: i386-redhat-linux

Configured with: ../configure --prefix = /usr .......

Thread model: posix

gcc version 4.1.2 20080704 (Red Hat 4.1.2-46)

If GCC is not installed, then you will have to install it yourself using the detailed instructions available at <https://gcc.gnu.org/install/>

This tutorial has been written based on Linux and all the given examples have been compiled on Cent OS flavor of Linux system.

## Installation on Mac OS

If you use Mac OS X, the easiest way to obtain GCC is to download the Xcode development environment from Apple's website and follow the simple installation instructions. Once you have Xcode setup, you will be able to use GNU compiler for C/C++.

Xcode is currently available at [developer.apple.com/technologies/tools/](https://developer.apple.com/technologies/tools/)

## Installation on Windows

To install GCC on Windows, you need to install MinGW. To install MinGW, go to the MinGW homepage, [www.mingw.org](http://www.mingw.org/), and follow the link to the MinGW download page. Download the latest version of the MinGW installation program, which should be named MinGW-<version>.exe.

While installing MinWG, at a minimum, you must install gcc-core, gcc-g++, binutils, and the MinGW runtime, but you may wish to install more.

Add the bin subdirectory of your MinGW installation to your **PATH** environment variable, so that you can specify these tools on the command line by their simple names.

When the installation is complete, you will be able to run gcc, g++, ar, ranlib, dlltool, and several other GNU tools from the Windows command line.

[Previous Page](https://www.tutorialspoint.com/data_structures_algorithms/data_structure_overview.htm) [Print Page](javascript:printPage();)

[Next Page](https://www.tutorialspoint.com/data_structures_algorithms/algorithms_basics.htm)

# Data Structures - Algorithms Basics

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.

From the data structure point of view, following are some important categories of algorithms −

* **Search** − Algorithm to search an item in a data structure.
* **Sort** − Algorithm to sort items in a certain order.
* **Insert** − Algorithm to insert 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** − 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.

### Example

Let's try to learn algorithm-writing by using an example.

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

**Step 1** − START

**Step 2** − declare three integers **a**, **b** & **c**

**Step 3** − define values of **a** & **b**

**Step 4** − add values of **a** & **b**

**Step 5** − store output of step 4 to **c**

**Step 6** − print **c**

**Step 7** − STOP

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

**Step 1** − START ADD

**Step 2** − get values of **a** & **b**

**Step 3** − c ← a + b

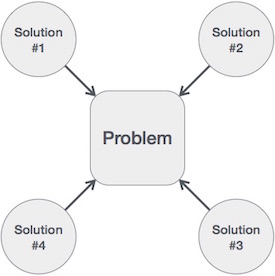
**Step 4** − display c

**Step 5** − STOP

In 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. He can observe what operations are being used and how the process is flowing.

Writing **step numbers**, is optional.

We design an algorithm to get a solution of a given problem. A problem can be solved in more than one ways.



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

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 algorithm. 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 programming language. This is then executed on target computer machine. In this analysis, actual statistics like running time and space required, are collected.

We shall learn about *a priori* algorithm analysis. 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.

## Algorithm Complexity

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

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

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 that 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, etc.
* A variable part is a space required by variables, whose size depends on the size of the problem. 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 −

Algorithm: SUM(A, B)

Step 1 - START

Step 2 - C ← A + B + 10

Step 3 - Stop

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

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, 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.

[Previous Page](https://www.tutorialspoint.com/data_structures_algorithms/data_structure_environment.htm) [Print Page](javascript:printPage();)

[Next Page](https://www.tutorialspoint.com/data_structures_algorithms/asymptotic_analysis.htm)

# Data Structures - Asymptotic Analysis

Asymptotic analysis of an algorithm refers to defining the mathematical boundation/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 a 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 may be 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.

Usually, the time required by an algorithm falls under three types −

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

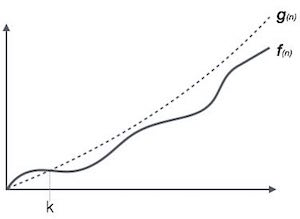
## Asymptotic Notations

Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

* Ο Notation
* Ω Notation
* θ Notation

### Big Oh Notation, Ο

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.



For example, for a function ***f*(n)**

Ο(*f*(n)) = { *g*(n) : there exists c > 0 and n0 such that *f*(n) ≤ c.*g*(n) for all n > n0. }

### Omega Notation, Ω

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.

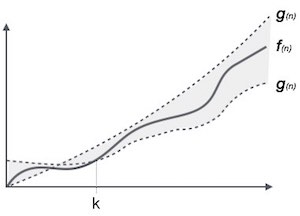


For example, for a function ***f*(n)**

Ω(*f*(n)) ≥ { *g*(n) : there exists c > 0 and n0 such that *g*(n) ≤ c.*f*(n) for all n > n0. }

### Theta Notation, θ

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 −



θ(*f*(n)) = { *g*(n) if and only if *g*(n) = Ο(*f*(n)) and *g*(n) = Ω(*f*(n)) for all n > n0. }

## Common Asymptotic Notations

Following is a list of some common asymptotic notations −

|  |  |  |
| --- | --- | --- |
| constant | − | Ο(1) |
| logarithmic | − | Ο(log n) |
| linear | − | Ο(n) |
| n log n | − | Ο(n log n) |
| quadratic | − | Ο(n2) |
| cubic | − | Ο(n3) |
| polynomial | − | nΟ(1) |
| exponential | − | 2Ο(n) |

[Previous Page](https://www.tutorialspoint.com/data_structures_algorithms/algorithms_basics.htm) [Print Page](javascript:printPage();)

[Next Page](https://www.tutorialspoint.com/data_structures_algorithms/greedy_algorithms.htm)

# Data Structures - Greedy Algorithms

An algorithm is designed to achieve optimum solution for a given problem. In greedy algorithm approach, decisions are made from the given solution domain. As being greedy, the closest solution that seems to provide an optimum solution is chosen.

Greedy algorithms try to find a localized optimum solution, which may eventually lead to globally optimized solutions. However, generally greedy algorithms do not provide globally optimized solutions.

## Counting Coins

This problem is to count to a desired value by choosing the least possible coins and the greedy approach forces the algorithm to pick the largest possible coin. If we are provided coins of ₹ 1, 2, 5 and 10 and we are asked to count ₹ 18 then the greedy procedure will be −

* **1** − Select one ₹ 10 coin, the remaining count is 8
* **2** − Then select one ₹ 5 coin, the remaining count is 3
* **3** − Then select one ₹ 2 coin, the remaining count is 1
* **4** − And finally, the selection of one ₹ 1 coins solves the problem

Though, it seems to be working fine, for this count we need to pick only 4 coins. But if we slightly change the problem then the same approach may not be able to produce the same optimum result.

For the currency system, where we have coins of 1, 7, 10 value, counting coins for value 18 will be absolutely optimum but for count like 15, it may use more coins than necessary. For example, the greedy approach will use 10 + 1 + 1 + 1 + 1 + 1, total 6 coins. Whereas the same problem could be solved by using only 3 coins (7 + 7 + 1)

Hence, we may conclude that the greedy approach picks an immediate optimized solution and may fail where global optimization is a major concern.

### Examples

Most networking algorithms use the greedy approach. Here is a list of few of them −

* Travelling Salesman Problem
* Prim's Minimal Spanning Tree Algorithm
* Kruskal's Minimal Spanning Tree Algorithm
* Dijkstra's Minimal Spanning Tree Algorithm
* Graph - Map Coloring
* Graph - Vertex Cover
* Knapsack Problem
* Job Scheduling Problem

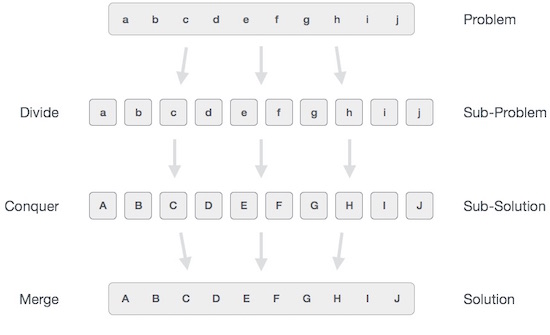
There are lots of similar problems that uses the greedy approach to find an optimum solution.

[Previous Page](https://www.tutorialspoint.com/data_structures_algorithms/asymptotic_analysis.htm) [Print Page](javascript:printPage();)

[Next Page](https://www.tutorialspoint.com/data_structures_algorithms/divide_and_conquer.htm)

# Data Structures - Divide and Conquer

In divide and conquer approach, the problem in hand, is divided into smaller sub-problems and then each problem is solved independently. When we keep on dividing the subproblems into even smaller sub-problems, we may eventually reach a stage where no more division is possible. Those "atomic" smallest possible sub-problem (fractions) are solved. The solution of all sub-problems is finally merged in order to obtain the solution of an original problem.



Broadly, we can understand **divide-and-conquer** approach in a three-step process.

## Divide/Break

This step involves breaking the problem into smaller sub-problems. Sub-problems should represent a part of the original problem. This step generally takes a recursive approach to divide the problem until no sub-problem is further divisible. At this stage, sub-problems become atomic in nature but still represent some part of the actual problem.

## Conquer/Solve

This step receives a lot of smaller sub-problems to be solved. Generally, at this level, the problems are considered 'solved' on their own.

## Merge/Combine

When the smaller sub-problems are solved, this stage recursively combines them until they formulate a solution of the original problem. This algorithmic approach works recursively and conquer & merge steps works so close that they appear as one.

### Examples

The following computer algorithms are based on **divide-and-conquer** programming approach −

* Merge Sort
* Quick Sort
* Binary Search
* Strassen's Matrix Multiplication
* Closest pair (points)

There are various ways available to solve any computer problem, but the mentioned are a good example of divide and conquer approach.

[Previous Page](https://www.tutorialspoint.com/data_structures_algorithms/greedy_algorithms.htm) [Print Page](javascript:printPage();)

[Next Page](https://www.tutorialspoint.com/data_structures_algorithms/dynamic_programming.htm)

# Data Structures - Dynamic Programming

Dynamic programming approach is similar to divide and conquer in breaking down the problem into smaller and yet smaller possible sub-problems. But unlike, divide and conquer, these sub-problems are not solved independently. Rather, results of these smaller sub-problems are remembered and used for similar or overlapping sub-problems.

Dynamic programming is used where we have problems, which can be divided into similar sub-problems, so that their results can be re-used. Mostly, these algorithms are used for optimization. Before solving the in-hand sub-problem, dynamic algorithm will try to examine the results of the previously solved sub-problems. The solutions of sub-problems are combined in order to achieve the best solution.

So we can say that −

* The problem should be able to be divided into smaller overlapping sub-problem.
* An optimum solution can be achieved by using an optimum solution of smaller sub-problems.
* Dynamic algorithms use Memoization.

## Comparison

In contrast to greedy algorithms, where local optimization is addressed, dynamic algorithms are motivated for an overall optimization of the problem.

In contrast to divide and conquer algorithms, where solutions are combined to achieve an overall solution, dynamic algorithms use the output of a smaller sub-problem and then try to optimize a bigger sub-problem. Dynamic algorithms use Memoization to remember the output of already solved sub-problems.

### Example

The following computer problems can be solved using dynamic programming approach −

* Fibonacci number series
* Knapsack problem
* Tower of Hanoi
* All pair shortest path by Floyd-Warshall
* Shortest path by Dijkstra
* Project scheduling

Dynamic programming can be used in both top-down and bottom-up manner. And of course, most of the times, referring to the previous solution output is cheaper than recomputing in terms of CPU cycles.

[Previous Page](https://www.tutorialspoint.com/data_structures_algorithms/divide_and_conquer.htm) [Print Page](javascript:printPage();)

[Next Page](https://www.tutorialspoint.com/data_structures_algorithms/data_structures_basics.htm)

# Data Structures & Algorithm Basic Concepts

This chapter explains the basic terms related to data structure.

## Data Definition

Data Definition defines a particular data with the following characteristics.

* **Atomic** − Definition should define a single concept.
* **Traceable** − Definition should be able to be mapped to some data element.
* **Accurate** − Definition should be unambiguous.
* **Clear and Concise** − Definition should be understandable.

## Data Object

Data Object represents an object having a data.

## Data Type

Data type is a way to classify various types of data such as integer, string, etc. which determines the values that can be used with the corresponding type of data, the type of operations that can be performed on the corresponding type of data. There are two data types −

* Built-in Data Type
* Derived Data Type

### Built-in Data Type

Those data types for which a language has built-in support are known as Built-in Data types. For example, most of the languages provide the following built-in data types.

* Integers
* Boolean (true, false)
* Floating (Decimal numbers)
* Character and Strings

### Derived Data Type

Those data types which are implementation independent as they can be implemented in one or the other way are known as derived data types. These data types are normally built by the combination of primary or built-in data types and associated operations on them. For example −

* List
* Array
* Stack
* Queue

## Basic Operations

The data in the data structures are processed by certain operations. The particular data structure chosen largely depends on the frequency of the operation that needs to be performed on the data structure.

* Traversing
* Searching
* Insertion
* Deletion
* Sorting
* Merging

[Previous Page](https://www.tutorialspoint.com/data_structures_algorithms/dynamic_programming.htm) [Print Page](javascript:printPage();)

[Next Page](https://www.tutorialspoint.com/data_structures_algorithms/array_data_structure.htm)

# Data Structures and Algorithms - Arrays

Array is a container which can hold a fix number of items and these items should be of the same type. Most of the data structures make use of arrays to implement their algorithms. Following are the important terms to understand the concept of Array.

* **Element** − Each item stored in an array is called an element.
* **Index** − Each location of an element in an array has a numerical index, which is used to identify the element.

## Array Representation

Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.



Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.



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

* Index starts with 0.
* Array length is 10 which means it can store 10 elements.
* Each element can be accessed via its index. For example, we can fetch an element at index 6 as 9.

## Basic Operations

Following are the basic operations supported by an array.

* **Traverse** − print all the array elements one by one.
* **Insertion** − Adds an element at the given index.
* **Deletion** − Deletes an element at the given index.
* **Search** − Searches an element using the given index or by the value.
* **Update** − Updates an element at the given index.

In C, when an array is initialized with size, then it assigns defaults values to its elements in following order.

|  |  |
| --- | --- |
| **Data Type** | **Default Value** |
| bool | false |
| char | 0 |
| int | 0 |
| float | 0.0 |
| double | 0.0f |
| void |  |
| wchar\_t | 0 |

## Traverse Operation

This operation is to traverse through the elements of an array.

### Example

Following program traverses and prints the elements of an array:

#include <stdio.h>

main() {

int LA[] = {1,3,5,7,8};

int item = 10, k = 3, n = 5;

int i = 0, j = n;

printf("The original array elements are :\n");

for(i = 0; i<n; i++) {

printf("LA[%d] = %d \n", i, LA[i]);

}

}

When we compile and execute the above program, it produces the following result −

### Output

The original array elements are :

LA[0] = 1

LA[1] = 3

LA[2] = 5

LA[3] = 7

LA[4] = 8

## Insertion Operation

Insert operation is to insert one or more data elements into an array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array.

Here, we see a practical implementation of insertion operation, where we add data at the end of the array −

### Example

Following is the implementation of the above algorithm −

[Live Demo](http://tpcg.io/YpAUzN)

#include <stdio.h>

main() {

int LA[] = {1,3,5,7,8};

int item = 10, k = 3, n = 5;

int i = 0, j = n;

printf("The original array elements are :\n");

for(i = 0; i<n; i++) {

printf("LA[%d] = %d \n", i, LA[i]);

}

n = n + 1;

while( j >= k) {

LA[j+1] = LA[j];

j = j - 1;

}

LA[k] = item;

printf("The array elements after insertion :\n");

for(i = 0; i<n; i++) {

printf("LA[%d] = %d \n", i, LA[i]);

}

}

When we compile and execute the above program, it produces the following result −

### Output

The original array elements are :

LA[0] = 1

LA[1] = 3

LA[2] = 5

LA[3] = 7

LA[4] = 8

The array elements after insertion :

LA[0] = 1

LA[1] = 3

LA[2] = 5

LA[3] = 10

LA[4] = 7

LA[5] = 8

For other variations of array insertion operation [click here](https://www.tutorialspoint.com/data_structures_algorithms/array_insertion_algorithm.htm)

## Deletion Operation

Deletion refers to removing an existing element from the array and re-organizing all elements of an array.

### Algorithm

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm to delete an element available at the Kth position of LA.

1. Start

2. Set J = K

3. Repeat steps 4 and 5 while J < N

4. Set LA[J] = LA[J + 1]

5. Set J = J+1

6. Set N = N-1

7. Stop

### Example

Following is the implementation of the above algorithm −

[Live Demo](http://tpcg.io/3FCSPD)

#include <stdio.h>

void main() {

int LA[] = {1,3,5,7,8};

int k = 3, n = 5;

int i, j;

printf("The original array elements are :\n");

for(i = 0; i<n; i++) {

printf("LA[%d] = %d \n", i, LA[i]);

}

j = k;

while( j < n) {

LA[j-1] = LA[j];

j = j + 1;

}

n = n -1;

printf("The array elements after deletion :\n");

for(i = 0; i<n; i++) {

printf("LA[%d] = %d \n", i, LA[i]);

}

}

When we compile and execute the above program, it produces the following result −

### Output

The original array elements are :

LA[0] = 1

LA[1] = 3

LA[2] = 5

LA[3] = 7

LA[4] = 8

The array elements after deletion :

LA[0] = 1

LA[1] = 3

LA[2] = 7

LA[3] = 8

## Search Operation

You can perform a search for an array element based on its value or its index.

### Algorithm

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm to find an element with a value of ITEM using sequential search.

1. Start

2. Set J = 0

3. Repeat steps 4 and 5 while J < N

4. IF LA[J] is equal ITEM THEN GOTO STEP 6

5. Set J = J +1

6. PRINT J, ITEM

7. Stop

### Example

Following is the implementation of the above algorithm −

[Live Demo](http://tpcg.io/613FSK)

#include <stdio.h>

void main() {

int LA[] = {1,3,5,7,8};

int item = 5, n = 5;

int i = 0, j = 0;

printf("The original array elements are :\n");

for(i = 0; i<n; i++) {

printf("LA[%d] = %d \n", i, LA[i]);

}

while( j < n){

if( LA[j] == item ) {

break;

}

j = j + 1;

}

printf("Found element %d at position %d\n", item, j+1);

}

When we compile and execute the above program, it produces the following result −

### Output

The original array elements are :

LA[0] = 1

LA[1] = 3

LA[2] = 5

LA[3] = 7

LA[4] = 8

Found element 5 at position 3

## Update Operation

Update operation refers to updating an existing element from the array at a given index.

### Algorithm

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm to update an element available at the Kth position of LA.

1. Start

2. Set LA[K-1] = ITEM

3. Stop

### Example

Following is the implementation of the above algorithm −

[Live Demo](http://tpcg.io/EQ4FEy)

#include <stdio.h>

void main() {

int LA[] = {1,3,5,7,8};

int k = 3, n = 5, item = 10;

int i, j;

printf("The original array elements are :\n");

for(i = 0; i<n; i++) {

printf("LA[%d] = %d \n", i, LA[i]);

}

LA[k-1] = item;

printf("The array elements after updation :\n");

for(i = 0; i<n; i++) {

printf("LA[%d] = %d \n", i, LA[i]);

}

}

When we compile and execute the above program, it produces the following result −

### Output

The original array elements are :

LA[0] = 1

LA[1] = 3

LA[2] = 5

LA[3] = 7

LA[4] = 8

The array elements after updation :

LA[0] = 1

LA[1] = 3

LA[2] = 10

LA[3] = 7

LA[4] = 8

[Previous Page](https://www.tutorialspoint.com/data_structures_algorithms/data_structures_basics.htm) [Print Page](javascript:printPage();)

[Next Page](https://www.tutorialspoint.com/data_structures_algorithms/linked_list_algorithms.htm)

# Data Structure and Algorithms - 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.

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.

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.

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.

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.

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;

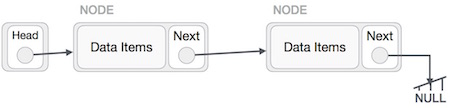


We need to use the deleted node. We can keep that in memory otherwise we can simply deallocate memory and wipe off the target node completely.

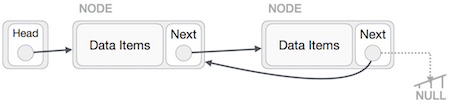


Reverse Operation

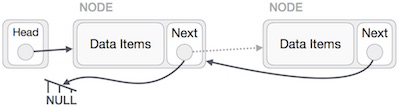
This operation is a thorough one. We need to make the last node to be pointed by the head node and reverse the whole linked list.



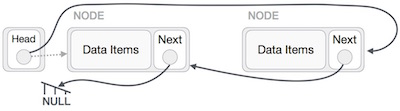
First, we traverse to the end of the list. It should be pointing to NULL. Now, we shall make it point to its previous node −



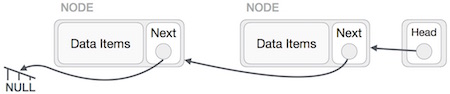
We have to make sure that the last node is not the last node. So we'll have some temp node, which looks like the head node pointing to the last node. Now, we shall make all left side nodes point to their previous nodes one by one.



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](https://www.tutorialspoint.com/data_structures_algorithms/linked_list_program_in_c.htm).

# Linked List Program in C

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.

## Implementation in C

[Live Demo](http://tpcg.io/YirM1q)

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <stdbool.h>

struct node {

int data;

int key;

struct node \*next;

};

struct node \*head = NULL;

struct node \*current = NULL;

//display the list

void printList() {

struct node \*ptr = head;

printf("\n[ ");

//start from the beginning

while(ptr != NULL) {

printf("(%d,%d) ",ptr->key,ptr->data);

ptr = ptr->next;

}

printf(" ]");

}

//insert link at the first location

void insertFirst(int key, int data) {

//create a link

struct node \*link = (struct node\*) malloc(sizeof(struct node));

link->key = key;

link->data = data;

//point it to old first node

link->next = head;

//point first to new first node

head = link;

}

//delete first item

struct node\* deleteFirst() {

//save reference to first link

struct node \*tempLink = head;

//mark next to first link as first

head = head->next;

//return the deleted link

return tempLink;

}

//is list empty

bool isEmpty() {

return head == NULL;

}

int length() {

int length = 0;

struct node \*current;

for(current = head; current != NULL; current = current->next) {

length++;

}

return length;

}

//find a link with given key

struct node\* find(int key) {

//start from the first link

struct node\* current = head;

//if list is empty

if(head == NULL) {

return NULL;

}

//navigate through list

while(current->key != key) {

//if it is last node

if(current->next == NULL) {

return NULL;

} else {

//go to next link

current = current->next;

}

}

//if data found, return the current Link

return current;

}

//delete a link with given key

struct node\* delete(int key) {

//start from the first link

struct node\* current = head;

struct node\* previous = NULL;

//if list is empty

if(head == NULL) {

return NULL;

}

//navigate through list

while(current->key != key) {

//if it is last node

if(current->next == NULL) {

return NULL;

} else {

//store reference to current link

previous = current;

//move to next link

current = current->next;

}

}

//found a match, update the link

if(current == head) {

//change first to point to next link

head = head->next;

} else {

//bypass the current link

previous->next = current->next;

}

return current;

}

void sort() {

int i, j, k, tempKey, tempData;

struct node \*current;

struct node \*next;

int size = length();

k = size ;

for ( i = 0 ; i < size - 1 ; i++, k-- ) {

current = head;

next = head->next;

for ( j = 1 ; j < k ; j++ ) {

if ( current->data > next->data ) {

tempData = current->data;

current->data = next->data;

next->data = tempData;

tempKey = current->key;

current->key = next->key;

next->key = tempKey;

}

current = current->next;

next = next->next;

}

}

}

void reverse(struct node\*\* head\_ref) {

struct node\* prev = NULL;

struct node\* current = \*head\_ref;

struct node\* next;

while (current != NULL) {

next = current->next;

current->next = prev;

prev = current;

current = next;

}

\*head\_ref = prev;

}

void main() {

insertFirst(1,10);

insertFirst(2,20);

insertFirst(3,30);

insertFirst(4,1);

insertFirst(5,40);

insertFirst(6,56);

printf("Original List: ");

//print list

printList();

while(!isEmpty()) {

struct node \*temp = deleteFirst();

printf("\nDeleted value:");

printf("(%d,%d) ",temp->key,temp->data);

}

printf("\nList after deleting all items: ");

printList();

insertFirst(1,10);

insertFirst(2,20);

insertFirst(3,30);

insertFirst(4,1);

insertFirst(5,40);

insertFirst(6,56);

printf("\nRestored List: ");

printList();

printf("\n");

struct node \*foundLink = find(4);

if(foundLink != NULL) {

printf("Element found: ");

printf("(%d,%d) ",foundLink->key,foundLink->data);

printf("\n");

} else {

printf("Element not found.");

}

delete(4);

printf("List after deleting an item: ");

printList();

printf("\n");

foundLink = find(4);

if(foundLink != NULL) {

printf("Element found: ");

printf("(%d,%d) ",foundLink->key,foundLink->data);

printf("\n");

} else {

printf("Element not found.");

}

printf("\n");

sort();

printf("List after sorting the data: ");

printList();

reverse(&head);

printf("\nList after reversing the data: ");

printList();

}

If we compile and run the above program, it will produce the following result −

## Output

Original List:

[ (6,56) (5,40) (4,1) (3,30) (2,20) (1,10) ]

Deleted value:(6,56)

Deleted value:(5,40)

Deleted value:(4,1)

Deleted value:(3,30)

Deleted value:(2,20)

Deleted value:(1,10)

List after deleting all items:

[ ]

Restored List:

[ (6,56) (5,40) (4,1) (3,30) (2,20) (1,10) ]

Element found: (4,1)

List after deleting an item:

[ (6,56) (5,40) (3,30) (2,20) (1,10) ]

Element not found.

List after sorting the data:

[ (1,10) (2,20) (3,30) (5,40) (6,56) ]

List after reversing the data:

[ (6,56) (5,40) (3,30) (2,20) (1,10) ]

[Previous Page](https://www.tutorialspoint.com/data_structures_algorithms/linked_list_algorithms.htm) [Print Page](javascript:printPage();)

[Next Page](https://www.tutorialspoint.com/data_structures_algorithms/linked_list_algorithms.htm)

# Data Structure - Doubly Linked List

Doubly Linked List is a variation of Linked list in which navigation is possible in both ways, either forward and backward easily as compared to Single Linked List. Following are the important terms to understand the concept of doubly 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.
* **Prev** − Each link of a linked list contains a link to the previous link called Prev.
* **LinkedList** − A Linked List contains the connection link to the first link called First and to the last link called Last.

## Doubly Linked List Representation



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

* Doubly Linked List contains a link element called first and last.
* Each link carries a data field(s) and two link fields called next and prev.
* Each link is linked with its next link using its next link.
* Each link is linked with its previous link using its previous link.
* The last link carries a link as null to mark the end of the list.

## 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.
* **Insert Last** − Adds an element at the end of the list.
* **Delete Last** − Deletes an element from the end of the list.
* **Insert After** − Adds an element after an item of the list.
* **Delete** − Deletes an element from the list using the key.
* **Display forward** − Displays the complete list in a forward manner.
* **Display backward** − Displays the complete list in a backward manner.

## Insertion Operation

Following code demonstrates the insertion operation at the beginning of a doubly linked list.

### Example

//insert link at the first location

void insertFirst(int key, int data) {

//create a link

struct node \*link = (struct node\*) malloc(sizeof(struct node));

link->key = key;

link->data = data;

if(isEmpty()) {

//make it the last link

last = link;

} else {

//update first prev link

head->prev = link;

}

//point it to old first link

link->next = head;

//point first to new first link

head = link;

}

## Deletion Operation

Following code demonstrates the deletion operation at the beginning of a doubly linked list.

### Example

//delete first item

struct node\* deleteFirst() {

//save reference to first link

struct node \*tempLink = head;

//if only one link

if(head->next == NULL) {

last = NULL;

} else {

head->next->prev = NULL;

}

head = head->next;

//return the deleted link

return tempLink;

}

## Insertion at the End of an Operation

Following code demonstrates the insertion operation at the last position of a doubly linked list.

### Example

//insert link at the last location

void insertLast(int key, int data) {

//create a link

struct node \*link = (struct node\*) malloc(sizeof(struct node));

link->key = key;

link->data = data;

if(isEmpty()) {

//make it the last link

last = link;

} else {

//make link a new last link

last->next = link;

//mark old last node as prev of new link

link->prev = last;

}

//point last to new last node

last = link;

}

To see the implementation in C programming language, please [click here](https://www.tutorialspoint.com/data_structures_algorithms/doubly_linked_list_program_in_c.htm).

# Doubly Linked List Program in C

Doubly Linked List is a variation of Linked list in which navigation is possible in both ways, either forward and backward easily as compared to Single Linked List.

## Implementation in C

[Live Demo](http://tpcg.io/phdk1z)

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <stdbool.h>

struct node {

int data;

int key;

struct node \*next;

struct node \*prev;

};

//this link always point to first Link

struct node \*head = NULL;

//this link always point to last Link

struct node \*last = NULL;

struct node \*current = NULL;

//is list empty

bool isEmpty() {

return head == NULL;

}

int length() {

int length = 0;

struct node \*current;

for(current = head; current != NULL; current = current->next){

length++;

}

return length;

}

//display the list in from first to last

void displayForward() {

//start from the beginning

struct node \*ptr = head;

//navigate till the end of the list

printf("\n[ ");

while(ptr != NULL) {

printf("(%d,%d) ",ptr->key,ptr->data);

ptr = ptr->next;

}

printf(" ]");

}

//display the list from last to first

void displayBackward() {

//start from the last

struct node \*ptr = last;

//navigate till the start of the list

printf("\n[ ");

while(ptr != NULL) {

//print data

printf("(%d,%d) ",ptr->key,ptr->data);

//move to next item

ptr = ptr ->prev;

}

}

//insert link at the first location

void insertFirst(int key, int data) {

//create a link

struct node \*link = (struct node\*) malloc(sizeof(struct node));

link->key = key;

link->data = data;

if(isEmpty()) {

//make it the last link

last = link;

} else {

//update first prev link

head->prev = link;

}

//point it to old first link

link->next = head;

//point first to new first link

head = link;

}

//insert link at the last location

void insertLast(int key, int data) {

//create a link

struct node \*link = (struct node\*) malloc(sizeof(struct node));

link->key = key;

link->data = data;

if(isEmpty()) {

//make it the last link

last = link;

} else {

//make link a new last link

last->next = link;

//mark old last node as prev of new link

link->prev = last;

}

//point last to new last node

last = link;

}

//delete first item

struct node\* deleteFirst() {

//save reference to first link

struct node \*tempLink = head;

//if only one link

if(head->next == NULL){

last = NULL;

} else {

head->next->prev = NULL;

}

head = head->next;

//return the deleted link

return tempLink;

}

//delete link at the last location

struct node\* deleteLast() {

//save reference to last link

struct node \*tempLink = last;

//if only one link

if(head->next == NULL) {

head = NULL;

} else {

last->prev->next = NULL;

}

last = last->prev;

//return the deleted link

return tempLink;

}

//delete a link with given key

struct node\* delete(int key) {

//start from the first link

struct node\* current = head;

struct node\* previous = NULL;

//if list is empty

if(head == NULL) {

return NULL;

}

//navigate through list

while(current->key != key) {

//if it is last node

if(current->next == NULL) {

return NULL;

} else {

//store reference to current link

previous = current;

//move to next link

current = current->next;

}

}

//found a match, update the link

if(current == head) {

//change first to point to next link

head = head->next;

} else {

//bypass the current link

current->prev->next = current->next;

}

if(current == last) {

//change last to point to prev link

last = current->prev;

} else {

current->next->prev = current->prev;

}

return current;

}

bool insertAfter(int key, int newKey, int data) {

//start from the first link

struct node \*current = head;

//if list is empty

if(head == NULL) {

return false;

}

//navigate through list

while(current->key != key) {

//if it is last node

if(current->next == NULL) {

return false;

} else {

//move to next link

current = current->next;

}

}

//create a link

struct node \*newLink = (struct node\*) malloc(sizeof(struct node));

newLink->key = newKey;

newLink->data = data;

if(current == last) {

newLink->next = NULL;

last = newLink;

} else {

newLink->next = current->next;

current->next->prev = newLink;

}

newLink->prev = current;

current->next = newLink;

return true;

}

void main() {

insertFirst(1,10);

insertFirst(2,20);

insertFirst(3,30);

insertFirst(4,1);

insertFirst(5,40);

insertFirst(6,56);

printf("\nList (First to Last): ");

displayForward();

printf("\n");

printf("\nList (Last to first): ");

displayBackward();

printf("\nList , after deleting first record: ");

deleteFirst();

displayForward();

printf("\nList , after deleting last record: ");

deleteLast();

displayForward();

printf("\nList , insert after key(4) : ");

insertAfter(4,7, 13);

displayForward();

printf("\nList , after delete key(4) : ");

delete(4);

displayForward();

}

If we compile and run the above program, it will produce the following result −

## Output

List (First to Last):

[ (6,56) (5,40) (4,1) (3,30) (2,20) (1,10) ]

List (Last to first):

[ (1,10) (2,20) (3,30) (4,1) (5,40) (6,56) ]

List , after deleting first record:

[ (5,40) (4,1) (3,30) (2,20) (1,10) ]

List , after deleting last record:

[ (5,40) (4,1) (3,30) (2,20) ]

List , insert after key(4) :

[ (5,40) (4,1) (7,13) (3,30) (2,20) ]

List , after delete key(4) :

[ (5,40) (4,13) (3,30) (2,20) ]

[Previous Page](https://www.tutorialspoint.com/data_structures_algorithms/doubly_linked_list_algorithm.htm) [Print Page](javascript:printPage();)

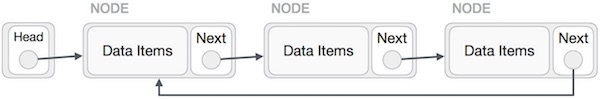
[Next Page](https://www.tutorialspoint.com/data_structures_algorithms/doubly_linked_list_algorithm.htm)

# Data Structure - Circular Linked List

Circular Linked List is a variation of Linked list in which the first element points to the last element and the last element points to the first element. Both Singly Linked List and Doubly Linked List can be made into a circular linked list.

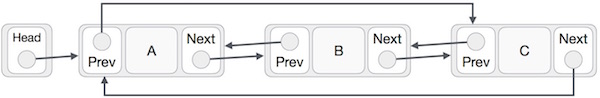
## Singly Linked List as Circular

In singly linked list, the next pointer of the last node points to the first node.



## Doubly Linked List as Circular

In doubly linked list, the next pointer of the last node points to the first node and the previous pointer of the first node points to the last node making the circular in both directions.



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

* The last link's next points to the first link of the list in both cases of singly as well as doubly linked list.
* The first link's previous points to the last of the list in case of doubly linked list.

## Basic Operations

Following are the important operations supported by a circular list.

* **insert** − Inserts an element at the start of the list.
* **delete** − Deletes an element from the start of the list.
* **display** − Displays the list.

## Insertion Operation

Following code demonstrates the insertion operation in a circular linked list based on single linked list.

### Example

insertFirst(data):

Begin

create a new node

node -> data := data

if the list is empty, then

head := node

next of node = head

else

temp := head

while next of temp is not head, do

temp := next of temp

done

next of node := head

next of temp := node

head := node

end if

End

## Deletion Operation

Following code demonstrates the deletion operation in a circular linked list based on single linked list.

deleteFirst():

Begin

if head is null, then

it is Underflow and return

else if next of head = head, then

head := null

deallocate head

else

ptr := head

while next of ptr is not head, do

ptr := next of ptr

next of ptr = next of head

deallocate head

head := next of ptr

end if

End

## Display List Operation

Following code demonstrates the display list operation in a circular linked list.

display():

Begin

if head is null, then

Nothing to print and return

else

ptr := head

while next of ptr is not head, do

display data of ptr

ptr := next of ptr

display data of ptr

end if

End

To know about its implementation in C programming language, please [click here](https://www.tutorialspoint.com/data_structures_algorithms/circular_linked_list_program_in_c.htm).

# Circular Linked List Program in C

Circular Linked List is a variation of Linked list in which the first element points to the last element and the last element points to the first element. Both Singly Linked List and Doubly Linked List can be made into a circular linked list.

## Implementation in C

[Live Demo](http://tpcg.io/jBzSzW)

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <stdbool.h>

struct node {

int data;

int key;

struct node \*next;

};

struct node \*head = NULL;

struct node \*current = NULL;

bool isEmpty() {

return head == NULL;

}

int length() {

int length = 0;

//if list is empty

if(head == NULL) {

return 0;

}

current = head->next;

while(current != head) {

length++;

current = current->next;

}

return length;

}

//insert link at the first location

void insertFirst(int key, int data) {

//create a link

struct node \*link = (struct node\*) malloc(sizeof(struct node));

link->key = key;

link->data = data;

if (isEmpty()) {

head = link;

head->next = head;

} else {

//point it to old first node

link->next = head;

//point first to new first node

head = link;

}

}

//delete first item

struct node \* deleteFirst() {

//save reference to first link

struct node \*tempLink = head;

if(head->next == head) {

head = NULL;

return tempLink;

}

//mark next to first link as first

head = head->next;

//return the deleted link

return tempLink;

}

//display the list

void printList() {

struct node \*ptr = head;

printf("\n[ ");

//start from the beginning

if(head != NULL) {

while(ptr->next != ptr) {

printf("(%d,%d) ",ptr->key,ptr->data);

ptr = ptr->next;

}

}

printf(" ]");

}

void main() {

insertFirst(1,10);

insertFirst(2,20);

insertFirst(3,30);

insertFirst(4,1);

insertFirst(5,40);

insertFirst(6,56);

printf("Original List: ");

//print list

printList();

while(!isEmpty()) {

struct node \*temp = deleteFirst();

printf("\nDeleted value:");

printf("(%d,%d) ",temp->key,temp->data);

}

printf("\nList after deleting all items: ");

printList();

}

If we compile and run the above program, it will produce the following result −

## Output

Original List:

[ (6,56) (5,40) (4,1) (3,30) (2,20) ]

Deleted value:(6,56)

Deleted value:(5,40)

Deleted value:(4,1)

Deleted value:(3,30)

Deleted value:(2,20)

Deleted value:(1,10)

List after deleting all items:

[ ]

[Previous Page](https://www.tutorialspoint.com/data_structures_algorithms/circular_linked_list_algorithm.htm) [Print Page](javascript:printPage();)

[Next Page](https://www.tutorialspoint.com/data_structures_algorithms/circular_linked_list_algorithm.htm)

# Data Structure and Algorithms - Stack

A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.



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

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

## Stack Representation

The following diagram depicts a stack and its operations −



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

## Basic Operations

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations −

* **push()** − Pushing (storing) an element on the stack.
* **pop()** − Removing (accessing) an element from the stack.

When data is PUSHed onto stack.

To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks −

* **peek()** − get the top data element of the stack, without removing it.
* **isFull()** − check if stack is full.
* **isEmpty()** − check if stack is empty.

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

First we should learn about procedures to support stack functions −

### peek()

Algorithm of peek() function −

begin procedure peek

return stack[top]

end procedure

Implementation of peek() function in C programming language −

**Example**

int peek() {

return stack[top];

}

### isfull()

Algorithm of isfull() function −

begin procedure isfull

if top equals to MAXSIZE

return true

else

return false

endif

end procedure

Implementation of isfull() function in C programming language −

**Example**

bool isfull() {

if(top == MAXSIZE)

return true;

else

return false;

}

### isempty()

Algorithm of isempty() function −

begin procedure isempty

if top less than 1

return true

else

return false

endif

end procedure

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

**Example**

bool isempty() {

if(top == -1)

return true;

else

return false;

}

## Push Operation

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps −

* **Step 1** − Checks if the stack is full.
* **Step 2** − If the stack is full, produces an error and exit.
* **Step 3** − If the stack is not full, increments **top** to point next empty space.
* **Step 4** − Adds data element to the stack location, where top is pointing.
* **Step 5** − Returns success.



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

### Algorithm for PUSH Operation

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

begin procedure push: stack, data

if stack is full

return null

endif

top ← top + 1

stack[top] ← data

end procedure

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

**Example**

void push(int data) {

if(!isFull()) {

top = top + 1;

stack[top] = data;

} else {

printf("Could not insert data, Stack is full.\n");

}

}

## 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.



### Algorithm for Pop Operation

A simple algorithm for Pop operation can be derived as follows −

begin procedure pop: stack

if stack is empty

return null

endif

data ← stack[top]

top ← top - 1

return data

end procedure

Implementation of this algorithm in C, is as follows −

**Example**

int pop(int data) {

if(!isempty()) {

data = stack[top];

top = top - 1;

return data;

} else {

printf("Could not retrieve data, Stack is empty.\n");

}

}

For a complete stack program in C programming language, please [click here](https://www.tutorialspoint.com/data_structures_algorithms/stack_program_in_c.htm)

# Stack Program in C

We shall see the stack implementation in C programming language here. You can try the program by clicking on the Try-it button. To learn the theory aspect of stacks, click on visit previous page.

## Implementation in C

[Live Demo](http://tpcg.io/5mrKa9)

#include <stdio.h>

int MAXSIZE = 8;

int stack[8];

int top = -1;

int isempty() {

if(top == -1)

return 1;

else

return 0;

}

int isfull() {

if(top == MAXSIZE)

return 1;

else

return 0;

}

int peek() {

return stack[top];

}

int pop() {

int data;

if(!isempty()) {

data = stack[top];

top = top - 1;

return data;

} else {

printf("Could not retrieve data, Stack is empty.\n");

}

}

int push(int data) {

if(!isfull()) {

top = top + 1;

stack[top] = data;

} else {

printf("Could not insert data, Stack is full.\n");

}

}

int main() {

// push items on to the stack

push(3);

push(5);

push(9);

push(1);

push(12);

push(15);

printf("Element at top of the stack: %d\n" ,peek());

printf("Elements: \n");

// print stack data

while(!isempty()) {

int data = pop();

printf("%d\n",data);

}

printf("Stack full: %s\n" , isfull()?"true":"false");

printf("Stack empty: %s\n" , isempty()?"true":"false");

return 0;

}

If we compile and run the above program, it will produce the following result −

## Output

Element at top of the stack: 15

Elements:

15

12

1

9

5

3

Stack full: false

Stack empty: true

[Previous Page](https://www.tutorialspoint.com/data_structures_algorithms/stack_algorithm.htm) [Print Page](javascript:printPage();)

[Next Page](https://www.tutorialspoint.com/data_structures_algorithms/stack_algorithm.htm)

# Data Structure - Expression Parsing

The way to write arithmetic expression is known as a **notation**. An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression. These notations are −

* Infix Notation
* Prefix (Polish) Notation
* Postfix (Reverse-Polish) Notation

These notations are named as how they use operator in expression. We shall learn the same here in this chapter.

## Infix Notation

We write expression in **infix** notation, e.g. a - b + c, where operators are used **in**-between operands. It is easy for us humans to read, write, and speak in infix notation but the same does not go well with computing devices. An algorithm to process infix notation could be difficult and costly in terms of time and space consumption.

## Prefix Notation

In this notation, operator is **prefix**ed to operands, i.e. operator is written ahead of operands. For example, **+ab**. This is equivalent to its infix notation **a + b**. Prefix notation is also known as **Polish Notation**.

## Postfix Notation

This notation style is known as **Reversed Polish Notation**. In this notation style, the operator is **postfix**ed to the operands i.e., the operator is written after the operands. For example, **ab+**. This is equivalent to its infix notation **a + b**.

The following table briefly tries to show the difference in all three notations −

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr.No.** | **Infix Notation** | **Prefix Notation** | **Postfix Notation** |
| 1 | a + b | + a b | a b + |
| 2 | (a + b) ∗ c | ∗ + a b c | a b + c ∗ |
| 3 | a ∗ (b + c) | ∗ a + b c | a b c + ∗ |
| 4 | a / b + c / d | + / a b / c d | a b / c d / + |
| 5 | (a + b) ∗ (c + d) | ∗ + a b + c d | a b + c d + ∗ |
| 6 | ((a + b) ∗ c) - d | - ∗ + a b c d | a b + c ∗ d - |

## Parsing Expressions

As we have discussed, it is not a very efficient way to design an algorithm or program to parse infix notations. Instead, these infix notations are first converted into either postfix or prefix notations and then computed.

To parse any arithmetic expression, we need to take care of operator precedence and associativity also.

### Precedence

When an operand is in between two different operators, which operator will take the operand first, is decided by the precedence of an operator over others. For example −

Operator Precendence

As multiplication operation has precedence over addition, b \* c will be evaluated first. A table of operator precedence is provided later.

### Associativity

Associativity describes the rule where operators with the same precedence appear in an expression. For example, in expression a + b − c, both + and – have the same precedence, then which part of the expression will be evaluated first, is determined by associativity of those operators. Here, both + and − are left associative, so the expression will be evaluated as **(a + b) − c**.

Precedence and associativity determines the order of evaluation of an expression. Following is an operator precedence and associativity table (highest to lowest) −

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr.No.** | **Operator** | **Precedence** | **Associativity** |
| 1 | Exponentiation ^ | Highest | Right Associative |
| 2 | Multiplication ( ∗ ) & Division ( / ) | Second Highest | Left Associative |
| 3 | Addition ( + ) & Subtraction ( − ) | Lowest | Left Associative |

The above table shows the default behavior of operators. At any point of time in expression evaluation, the order can be altered by using parenthesis. For example −

In **a + b\*c**, the expression part **b**\***c** will be evaluated first, with multiplication as precedence over addition. We here use parenthesis for **a + b** to be evaluated first, like **(a + b)\*c**.

## Postfix Evaluation Algorithm

We shall now look at the algorithm on how to evaluate postfix notation −

Step 1 − scan the expression from left to right

Step 2 − if it is an operand push it to stack

Step 3 − if it is an operator pull operand from stack and perform operation

Step 4 − store the output of step 3, back to stack

Step 5 − scan the expression until all operands are consumed

Step 6 − pop the stack and perform operation

To see the implementation in C programming language, please [click here](https://www.tutorialspoint.com/data_structures_algorithms/expression_parsing_using_statck.htm).

# Expression Parsing Using Stack

Infix notation is easier for humans to read and understand whereas for electronic machines like computers, postfix is the best form of expression to parse. We shall see here a program to convert and evaluate **infix** notation to **postfix** notation −

## Example

[Live Demo](http://tpcg.io/dRrsuh)

#include<stdio.h>

#include<string.h>

//char stack

char stack[25];

int top = -1;

void push(char item) {

stack[++top] = item;

}

char pop() {

return stack[top--];

}

//returns precedence of operators

int precedence(char symbol) {

switch(symbol) {

case '+':

case '-':

return 2;

break;

case '\*':

case '/':

return 3;

break;

case '^':

return 4;

break;

case '(':

case ')':

case '#':

return 1;

break;

}

}

//check whether the symbol is operator?

int isOperator(char symbol) {

switch(symbol) {

case '+':

case '-':

case '\*':

case '/':

case '^':

case '(':

case ')':

return 1;

break;

default:

return 0;

}

}

//converts infix expression to postfix

void convert(char infix[],char postfix[]) {

int i,symbol,j = 0;

stack[++top] = '#';

for(i = 0;i<strlen(infix);i++) {

symbol = infix[i];

if(isOperator(symbol) == 0) {

postfix[j] = symbol;

j++;

} else {

if(symbol == '(') {

push(symbol);

} else {

if(symbol == ')') {

while(stack[top] != '(') {

postfix[j] = pop();

j++;

}

pop(); //pop out (.

} else {

if(precedence(symbol)>precedence(stack[top])) {

push(symbol);

} else {

while(precedence(symbol)<=precedence(stack[top])) {

postfix[j] = pop();

j++;

}

push(symbol);

}

}

}

}

}

while(stack[top] != '#') {

postfix[j] = pop();

j++;

}

postfix[j]='\0'; //null terminate string.

}

//int stack

int stack\_int[25];

int top\_int = -1;

void push\_int(int item) {

stack\_int[++top\_int] = item;

}

char pop\_int() {

return stack\_int[top\_int--];

}

//evaluates postfix expression

int evaluate(char \*postfix){

char ch;

int i = 0,operand1,operand2;

while( (ch = postfix[i++]) != '\0') {

if(isdigit(ch)) {

push\_int(ch-'0'); // Push the operand

} else {

//Operator,pop two operands

operand2 = pop\_int();

operand1 = pop\_int();

switch(ch) {

case '+':

push\_int(operand1+operand2);

break;

case '-':

push\_int(operand1-operand2);

break;

case '\*':

push\_int(operand1\*operand2);

break;

case '/':

push\_int(operand1/operand2);

break;

}

}

}

return stack\_int[top\_int];

}

void main() {

char infix[25] = "1\*(2+3)",postfix[25];

convert(infix,postfix);

printf("Infix expression is: %s\n" , infix);

printf("Postfix expression is: %s\n" , postfix);

printf("Evaluated expression is: %d\n" , evaluate(postfix));

}

If we compile and run the above program, it will produce the following result −

## Output

Infix expression is: 1\*(2+3)

Postfix expression is: 123+\*

Result is: 5

[Previous Page](https://www.tutorialspoint.com/data_structures_algorithms/expression_parsing.htm) [Print Page](javascript:printPage();)

[Next Page](https://www.tutorialspoint.com/data_structures_algorithms/expression_parsing.htm)

# Data Structure and Algorithms - 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.

## 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.

## 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.

Let's first learn about supportive functions of a queue −

### peek()

This function helps to see the data at the **front** of the queue. The algorithm of peek() function is as follows −

**Algorithm**

begin procedure peek

return queue[front]

end procedure

Implementation of peek() function in C programming language −

**Example**

int peek() {

return queue[front];

}

### isfull()

As we are using single dimension array to implement queue, we just check for the rear pointer to reach at MAXSIZE to determine that the queue is full. In case we maintain the queue in a circular linked-list, the algorithm will differ. Algorithm of isfull() function −

**Algorithm**

begin procedure isfull

if rear equals to MAXSIZE

return true

else

return false

endif

end procedure

Implementation of isfull() function in C programming language −

**Example**

bool isfull() {

if(rear == MAXSIZE - 1)

return true;

else

return false;

}

### isempty()

Algorithm of isempty() function −

**Algorithm**

begin procedure isempty

if front is less than MIN OR front is greater than rear

return true

else

return false

endif

end procedure

If the value of **front** is less than MIN or 0, it tells that the queue is not yet initialized, hence empty.

Here's the C programming code −

**Example**

bool isempty() {

if(front < 0 || front > rear)

return true;

else

return false;

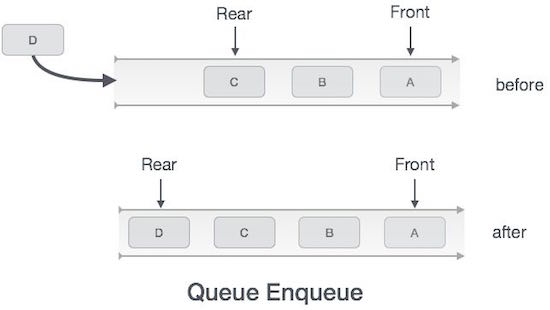
}

## 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.



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

### Algorithm for enqueue operation

procedure enqueue(data)

if queue is full

return overflow

endif

rear ← rear + 1

queue[rear] ← data

return true

end procedure

Implementation of enqueue() in C programming language −

**Example**

int enqueue(int data)

if(isfull())

return 0;

rear = rear + 1;

queue[rear] = data;

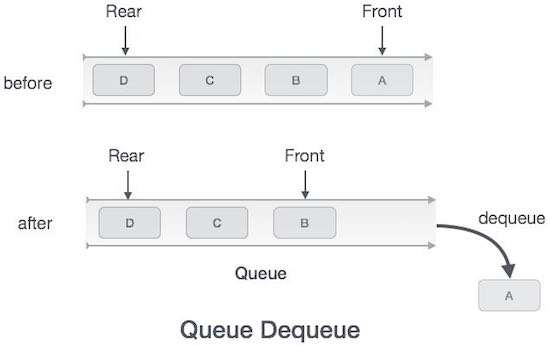
return 1;

end procedure

## 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.



### Algorithm for dequeue operation

procedure dequeue

if queue is empty

return underflow

end if

data = queue[front]

front ← front + 1

return true

end procedure

Implementation of dequeue() in C programming language −

**Example**

int dequeue() {

if(isempty())

return 0;

int data = queue[front];

front = front + 1;

return data;

}

For a complete Queue program in C programming language, please [click here](https://www.tutorialspoint.com/data_structures_algorithms/queue_program_in_c.htm).

# Queue Program In C

We shall see the stack implementation in C programming language here. You can try the program by clicking on the Try-it button. To learn the theory aspect of stacks, click on visit previous page.

## Implementation in C

[Live Demo](http://tpcg.io/F9sxGQ)

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <stdbool.h>

#define MAX 6

int intArray[MAX];

int front = 0;

int rear = -1;

int itemCount = 0;

int peek() {

return intArray[front];

}

bool isEmpty() {

return itemCount == 0;

}

bool isFull() {

return itemCount == MAX;

}

int size() {

return itemCount;

}

void insert(int data) {

if(!isFull()) {

if(rear == MAX-1) {

rear = -1;

}

intArray[++rear] = data;

itemCount++;

}

}

int removeData() {

int data = intArray[front++];

if(front == MAX) {

front = 0;

}

itemCount--;

return data;

}

int main() {

/\* insert 5 items \*/

insert(3);

insert(5);

insert(9);

insert(1);

insert(12);

// front : 0

// rear : 4

// ------------------

// index : 0 1 2 3 4

// ------------------

// queue : 3 5 9 1 12

insert(15);

// front : 0

// rear : 5

// ---------------------

// index : 0 1 2 3 4 5

// ---------------------

// queue : 3 5 9 1 12 15

if(isFull()) {

printf("Queue is full!\n");

}

// remove one item

int num = removeData();

printf("Element removed: %d\n",num);

// front : 1

// rear : 5

// -------------------

// index : 1 2 3 4 5

// -------------------

// queue : 5 9 1 12 15

// insert more items

insert(16);

// front : 1

// rear : -1

// ----------------------

// index : 0 1 2 3 4 5

// ----------------------

// queue : 16 5 9 1 12 15

// As queue is full, elements will not be inserted.

insert(17);

insert(18);

// ----------------------

// index : 0 1 2 3 4 5

// ----------------------

// queue : 16 5 9 1 12 15

printf("Element at front: %d\n",peek());

printf("----------------------\n");

printf("index : 5 4 3 2 1 0\n");

printf("----------------------\n");

printf("Queue: ");

while(!isEmpty()) {

int n = removeData();

printf("%d ",n);

}

}

If we compile and run the above program, it will produce the following result −

## Output

Queue is full!

Element removed: 3

Element at front: 5

----------------------

index : 5 4 3 2 1 0

----------------------

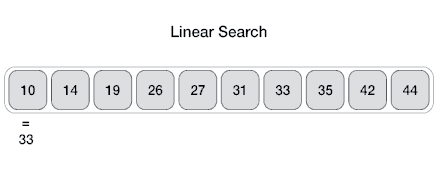
Queue: 5 9 1 12 15 16

[Previous Page](https://www.tutorialspoint.com/data_structures_algorithms/dsa_queue.htm) [Print Page](javascript:printPage();)

[Next Page](https://www.tutorialspoint.com/data_structures_algorithms/dsa_queue.htm)

# Data Structure and Algorithms Linear Search

Linear search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.



## Algorithm

Linear Search ( Array A, Value x)

Step 1: Set i to 1

Step 2: if i > n then go to step 7

Step 3: if A[i] = x then go to step 6

Step 4: Set i to i + 1

Step 5: Go to Step 2

Step 6: Print Element x Found at index i and go to step 8

Step 7: Print element not found

Step 8: Exit

## Pseudocode

procedure linear\_search (list, value)

for each item in the list

if match item == value

return the item's location

end if

end for

end procedure

To know about linear search implementation in C programming language, please [click-here](https://www.tutorialspoint.com/data_structures_algorithms/linear_search_program_in_c.htm).

# Linear Search Program in C

Here we present the implementation of linear search in C programming language. The output of the program is given after the code.

## Linear Search Program

[Live Demo](http://tpcg.io/j27oLx)

#include <stdio.h>

#define MAX 20

// array of items on which linear search will be conducted.

int intArray[MAX] = {1,2,3,4,6,7,9,11,12,14,15,16,17,19,33,34,43,45,55,66};

void printline(int count) {

int i;

for(i = 0;i <count-1;i++) {

printf("=");

}

printf("=\n");

}

// this method makes a linear search.

int find(int data) {

int comparisons = 0;

int index = -1;

int i;

// navigate through all items

for(i = 0;i<MAX;i++) {

// count the comparisons made

comparisons++;

// if data found, break the loop

if(data == intArray[i]) {

index = i;

break;

}

}

printf("Total comparisons made: %d", comparisons);

return index;

}

void display() {

int i;

printf("[");

// navigate through all items

for(i = 0;i<MAX;i++) {

printf("%d ",intArray[i]);

}

printf("]\n");

}

void main() {

printf("Input Array: ");

display();

printline(50);

//find location of 1

int location = find(55);

// if element was found

if(location != -1)

printf("\nElement found at location: %d" ,(location+1));

else

printf("Element not found.");

}

If we compile and run the above program, it will produce the following result −

## Output

Input Array: [1 2 3 4 6 7 9 11 12 14 15 16 17 19 33 34 43 45 55 66 ]

==================================================

Total comparisons made: 19

Element found at location: 19

[Previous Page](https://www.tutorialspoint.com/data_structures_algorithms/linear_search_algorithm.htm) [Print Page](javascript:printPage();)

[Next Page](https://www.tutorialspoint.com/data_structures_algorithms/linear_search_algorithm.htm)

# Data Structure and Algorithms Binary Search

Binary search is a fast search algorithm with run-time complexity of Ο(log n). This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in the sorted form.

Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the subarray reduces to zero.

## How Binary Search Works?

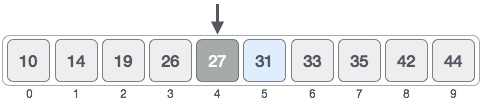
For a binary search to work, it is mandatory for the target array to be sorted. We shall learn the process of binary search with a pictorial example. The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.



First, we shall determine half of the array by using this formula −

mid = low + (high - low) / 2

Here it is, 0 + (9 - 0 ) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.



Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.



We change our low to mid + 1 and find the new mid value again.

low = mid + 1

mid = low + (high - low) / 2

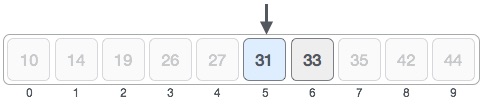
Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.



The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.



Hence, we calculate the mid again. This time it is 5.



We compare the value stored at location 5 with our target value. We find that it is a match.



We conclude that the target value 31 is stored at location 5.

Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

## Pseudocode

The pseudocode of binary search algorithms should look like this −

Procedure binary\_search

A ← sorted array

n ← size of array

x ← value to be searched

Set lowerBound = 1

Set upperBound = n

while x not found

if upperBound < lowerBound

EXIT: x does not exists.

set midPoint = lowerBound + ( upperBound - lowerBound ) / 2

if A[midPoint] < x

set lowerBound = midPoint + 1

if A[midPoint] > x

set upperBound = midPoint - 1

if A[midPoint] = x

EXIT: x found at location midPoint

end while

end procedure

To know about binary search implementation using array in C programming language, please [click here](https://www.tutorialspoint.com/data_structures_algorithms/binary_search_program_in_c.htm).

# Binary Search Program in C

Binary search is a fast search algorithm with run-time complexity of Ο(log n). This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in a sorted form.

## Implementation in C

[Live Demo](http://tpcg.io/lUVmC7)

#include <stdio.h>

#define MAX 20

// array of items on which linear search will be conducted.

int intArray[MAX] = {1,2,3,4,6,7,9,11,12,14,15,16,17,19,33,34,43,45,55,66};

void printline(int count) {

int i;

for(i = 0;i <count-1;i++) {

printf("=");

}

printf("=\n");

}

int find(int data) {

int lowerBound = 0;

int upperBound = MAX -1;

int midPoint = -1;

int comparisons = 0;

int index = -1;

while(lowerBound <= upperBound) {

printf("Comparison %d\n" , (comparisons +1) );

printf("lowerBound : %d, intArray[%d] = %d\n",lowerBound,lowerBound,

intArray[lowerBound]);

printf("upperBound : %d, intArray[%d] = %d\n",upperBound,upperBound,

intArray[upperBound]);

comparisons++;

// compute the mid point

// midPoint = (lowerBound + upperBound) / 2;

midPoint = lowerBound + (upperBound - lowerBound) / 2;

// data found

if(intArray[midPoint] == data) {

index = midPoint;

break;

} else {

// if data is larger

if(intArray[midPoint] < data) {

// data is in upper half

lowerBound = midPoint + 1;

}

// data is smaller

else {

// data is in lower half

upperBound = midPoint -1;

}

}

}

printf("Total comparisons made: %d" , comparisons);

return index;

}

void display() {

int i;

printf("[");

// navigate through all items

for(i = 0;i<MAX;i++) {

printf("%d ",intArray[i]);

}

printf("]\n");

}

void main() {

printf("Input Array: ");

display();

printline(50);

//find location of 1

int location = find(55);

// if element was found

if(location != -1)

printf("\nElement found at location: %d" ,(location+1));

else

printf("\nElement not found.");

}

If we compile and run the above program then it would produce following result −

## Output

Input Array: [1 2 3 4 6 7 9 11 12 14 15 16 17 19 33 34 43 45 55 66 ]

==================================================

Comparison 1

lowerBound : 0, intArray[0] = 1

upperBound : 19, intArray[19] = 66

Comparison 2

lowerBound : 10, intArray[10] = 15

upperBound : 19, intArray[19] = 66

Comparison 3

lowerBound : 15, intArray[15] = 34

upperBound : 19, intArray[19] = 66

Comparison 4

lowerBound : 18, intArray[18] = 55

upperBound : 19, intArray[19] = 66

Total comparisons made: 4

Element found at location: 19

[Previous Page](https://www.tutorialspoint.com/data_structures_algorithms/binary_search_algorithm.htm) [Print Page](javascript:printPage();)

[Next Page](https://www.tutorialspoint.com/data_structures_algorithms/binary_search_algorithm.htm)

# Data Structure - Interpolation Search

Interpolation search is an improved variant of binary search. This search algorithm works on the probing position of the required value. For this algorithm to work properly, the data collection should be in a sorted form and equally distributed.

Binary search has a huge advantage of time complexity over linear search. Linear search has worst-case complexity of Ο(n) whereas binary search has Ο(log n).

There are cases where the location of target data may be known in advance. For example, in case of a telephone directory, if we want to search the telephone number of Morphius. Here, linear search and even binary search will seem slow as we can directly jump to memory space where the names start from 'M' are stored.

## Positioning in Binary Search

In binary search, if the desired data is not found then the rest of the list is divided in two parts, lower and higher. The search is carried out in either of them.

BST Step OneBST Step TwoBST Step ThreeBST Step Four

Even when the data is sorted, binary search does not take advantage to probe the position of the desired data.

## Position Probing in Interpolation Search

Interpolation search finds a particular item by computing the probe position. Initially, the probe position is the position of the middle most item of the collection.

Interpolation Step OneInterpolation Step Two

If a match occurs, then the index of the item is returned. To split the list into two parts, we use the following method −

mid = Lo + ((Hi - Lo) / (A[Hi] - A[Lo])) \* (X - A[Lo])

where −

A = list

Lo = Lowest index of the list

Hi = Highest index of the list

A[n] = Value stored at index n in the list

If the middle item is greater than the item, then the probe position is again calculated in the sub-array to the right of the middle item. Otherwise, the item is searched in the subarray to the left of the middle item. This process continues on the sub-array as well until the size of subarray reduces to zero.

Runtime complexity of interpolation search algorithm is **Ο(log (log n))** as compared to **Ο(log n)** of BST in favorable situations.

### Algorithm

As it is an improvisation of the existing BST algorithm, we are mentioning the steps to search the 'target' data value index, using position probing −

Step 1 − Start searching **data** from middle of the list.

Step 2 − If it is a match, return the index of the item, and exit.

Step 3 − If it is not a match, probe position.

Step 4 − Divide the list using probing formula and find the new midle.

Step 5 − If data is greater than middle, search in higher sub-list.

Step 6 − If data is smaller than middle, search in lower sub-list.

Step 7 − Repeat until match.

## Pseudocode

A → Array list

N → Size of A

X → Target Value

Procedure Interpolation\_Search()

Set Lo → 0

Set Mid → -1

Set Hi → N-1

While X does not match

if Lo equals to Hi OR A[Lo] equals to A[Hi]

EXIT: Failure, Target not found

end if

Set Mid = Lo + ((Hi - Lo) / (A[Hi] - A[Lo])) \* (X - A[Lo])

if A[Mid] = X

EXIT: Success, Target found at Mid

else

if A[Mid] < X

Set Lo to Mid+1

else if A[Mid] > X

Set Hi to Mid-1

end if

end if

End While

End Procedure

To know about the implementation of interpolation search in C programming language, [click here](https://www.tutorialspoint.com/data_structures_algorithms/interpolation_search_in_c.htm).

# Interpolation Search Program in C

Interpolation search is an improved variant of binary search. This search algorithm works on the probing position of the required value. For this algorithm to work properly, the data collection should be in sorted and equally distributed form.

It's runtime complexity is **log2(log2 n)**.

## Implementation in C

[Live Demo](http://tpcg.io/GcjF3B)

#include<stdio.h>

#define MAX 10

// array of items on which linear search will be conducted.

int list[MAX] = { 10, 14, 19, 26, 27, 31, 33, 35, 42, 44 };

int find(int data) {

int lo = 0;

int hi = MAX - 1;

int mid = -1;

int comparisons = 1;

int index = -1;

while(lo <= hi) {

printf("\nComparison %d \n" , comparisons ) ;

printf("lo : %d, list[%d] = %d\n", lo, lo, list[lo]);

printf("hi : %d, list[%d] = %d\n", hi, hi, list[hi]);

comparisons++;

// probe the mid point

mid = lo + (((double)(hi - lo) / (list[hi] - list[lo])) \* (data - list[lo]));

printf("mid = %d\n",mid);

// data found

if(list[mid] == data) {

index = mid;

break;

} else {

if(list[mid] < data) {

// if data is larger, data is in upper half

lo = mid + 1;

} else {

// if data is smaller, data is in lower half

hi = mid - 1;

}

}

}

printf("\nTotal comparisons made: %d", --comparisons);

return index;

}

int main() {

//find location of 33

int location = find(33);

// if element was found

if(location != -1)

printf("\nElement found at location: %d" ,(location+1));

else

printf("Element not found.");

return 0;

}

If we compile and run the above program, it will produce the following result −

## Output

Comparison 1

lo : 0, list[0] = 10

hi : 9, list[9] = 44

mid = 6

Total comparisons made: 1

Element found at location: 7

You can change the search value and execute the program to test it.

[Previous Page](https://www.tutorialspoint.com/data_structures_algorithms/interpolation_search_algorithm.htm) [Print Page](javascript:printPage();)

[Next Page](https://www.tutorialspoint.com/data_structures_algorithms/interpolation_search_algorithm.htm)