CD-Rom Included

Second Revised & Updated Edition

DATA STRUCTURES THROUGH C IN DEPTH

S. K. Srivastava
Deepali Srivastava

BPB PUBLICATIONS

Basics

1. Overview Data Structures & Algorithms

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 ofa 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(10^6) items of a store. If the application is to search an item, it has to search an item in 1 million(10^6) 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.

ExecutionTimeCases

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 f(n) then this operation will not take more than f(n) time, where f(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 f(n) time in execution, then m operations will take mf(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 f(n) time in execution, then the actual operation may take time as the random number which would be maximum as f(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.

2. Environment Setup

Tryit Option Online

You really do not need to set up your own environment to start learning C programming language. Reason is very simple, we already have set up C Programming environment online, so that you can compile and execute all the available examples online at the same time when you are doing your theory work. This gives you confidence in what you are reading and to check the result with different options. Feel free to modify any example and execute it online.

Try the following example using the **Try it** option available at the top right corner of the sample code box −

```
#include <stdio.h>

int main(){
/* My first program in C */ printf(“Hello, World! \n”);

return 0; }
```

For most of the examples given in this tutorial, you will find Try it option, so just make use of it and enjoy your learning.

LocalEnvironment 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 http://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/

Installation onWindows

To install GCC on Windows, you need to install MinGW. To install MinGW, go to the MinGW homepage, 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.

Algorithm

3. 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 ofanAlgorithm

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 toWriteanAlgorithm?

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

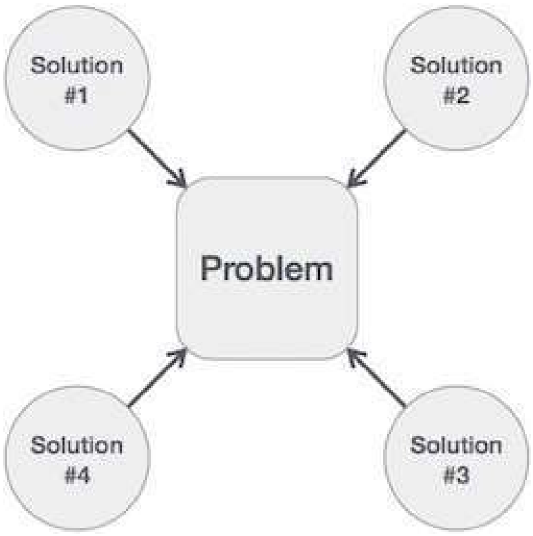
Algorithms tell the programmers how to code the program. Alternatively, the algorithm can be written as −

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.

AlgorithmAnalysis

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.

AlgorithmComplexity

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.

SpaceComplexity

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.

TimeComplexity

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.

4. 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 $g(n^2)$. 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.

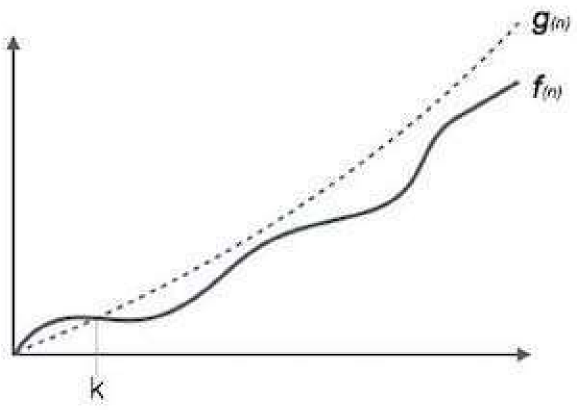
Usually, the time required by an algorithm falls under three types −
Best Case − Minimum time required for program execution.
Average Case − Average time required for program execution. **Worst Case** − Maximum time required for program execution.

AsymptoticNotations

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

Big Oh Notation, O

The notation O(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.

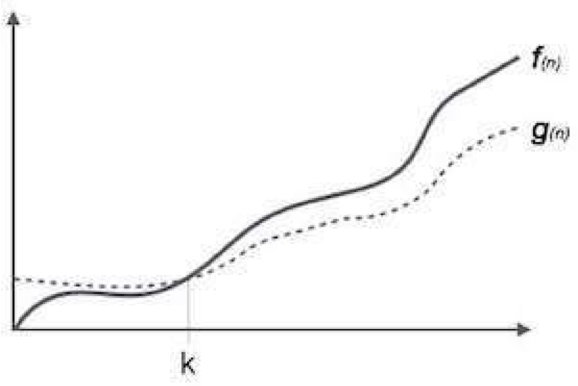


For example, for a function ***f*(n)**

O(f(n)) = { g(n) : there exists c > 0 and n0 such that g(n) ≤ c.f(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.

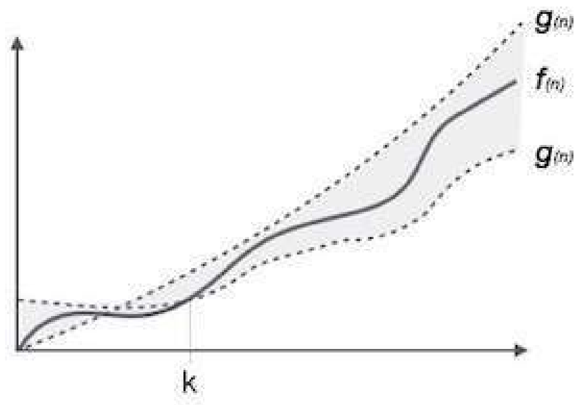


For example, for a function ***f*(n)**

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) = O(f(n)) and g(n) = Ω(f(n)) for all n > n0. }

CommonAsymptotic Notations

Following is a list of some common asymptotic notations:
constant − O(1)
logarithmic − O(log n)
linear − O(n)
n log n − O(n log n)
quadratic − $O(n^2)$
cubic − $O(n^3)$
polynomial − $n^{O(1)}$ exponential − $2^{O(n)}$

5. 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
3 − 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)

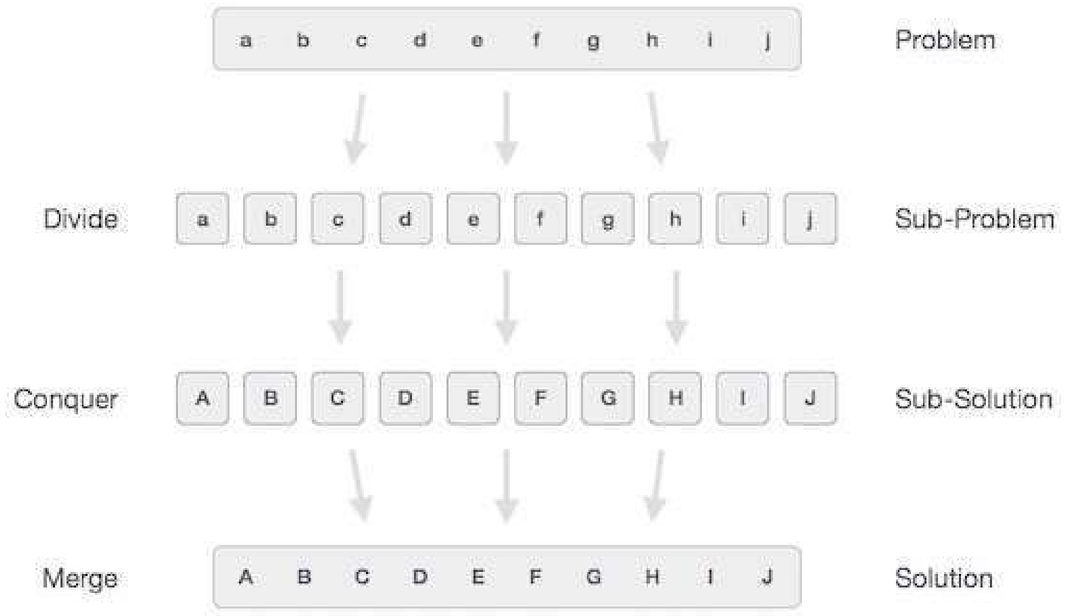
Hence, we may conclude that the greedy approach picks an immediate optimized solution and may fail where global optimization is a major concern.

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.

6. Divide & 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.

7. 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 −
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 subproblems.
Dynamic algorithms use memorization.

Comparison

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

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 memorization 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.

8. 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

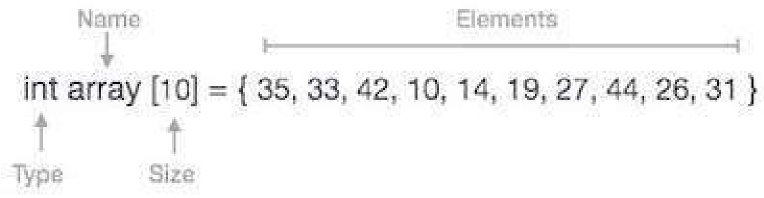
9. Arrays Data Structures & Algorithms

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.

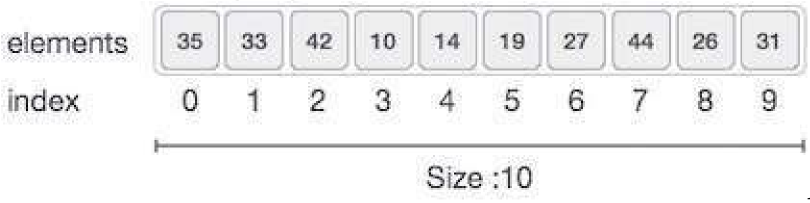
Element − Each item stored in an array is called an element.
Index − Each location of an element in an array has a numerical index, which is used to identify the element.

ArrayRepresentation

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 8 which means it can store 8 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 − Prints 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

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 −

Algorithm

Let **Array** be a linear unordered array of **MAX** elements.

Example

Result
Let **LA** be a Linear Array (unordered) with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm where ITEM is inserted into the K^{th}position of LA −

```
1. Start
2. Set J=N
3. Set N = N+1
4. Repeat steps 5 and 6 while J >= K
5. Set LA[J+1] = LA[J]
6. Set J = J-1
7. Set LA[K] = ITEM
8. Stop
```

Example

Following is the implementation of the above algorithm −

```
#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 −

```
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

ArrayInsertions

In the previous section, we have learnt how the insertion operation works. It is not always necessary that an element is inserted at the end of an array. Following can be a situation with array insertion −

Insertion at the beginning of an array
Insertion at the given index of an array
Insertion after the given index of an array Insertion before the given index of an array

Insertion at the BeginningofanArray

When the insertion happens at the beginning, it causes all the existing data items to shift one step downward. Here, we design and implement an algorithm to insert an element at the beginning of an array.

Algorithm

We assume **A** is an array with **N** elements. The maximum numbers of elements it can store is defined by **MAX**. We shall first check if an array has any empty space to store any element and then we proceed with the insertion process.

```
begin
IF N = MAX, return ELSE
N = N + 1
For All Elements in A
Move to next adjacent location
A[FIRST] = New_Element end
```

Implementation in C

```
#include <stdio.h> #define MAX 5

void main() {
int array[MAX] = {2, 3, 4, 5};
int N = 4; // number of elements in array int i = 0; // loop variable
int value = 1; // new data element to be stored in array

// print array before insertion
printf("Printing array before insertion −\n"); for(i = 0; i < N; i++) {
printf("array[%d] = %d \n", i, array[i]); }
// now shift rest of the elements downwards for(i = N; i >= 0; i—) {
array[i+1] = array[i];
}
// add new element at first position array[0] = value;
// increase N to reflect number of elements N++;
// print to confirm
printf("Printing array after insertion −\n");

for(i = 0; i < N; i++) {
printf("array[%d] = %d\n", i, array[i]);
}
}
```

This program should yield the following output −

```
Printing array before insertion − array[0] = 2
array[1] = 3
array[2] = 4
```

```
array[3] = 5
Printing array after insertion − array[0] = 1
array[1] = 2
array[2] = 3
array[3] = 4
array[4] = 5
```

Insertion attheGivenIndexof anArray

In this scenario, we are given the exact location (**index**) of an array where a new data element (**value**) needs to be inserted. First we shall check if the array is full, if it is not, then we shall move all data elements from that location one step downward. This will make room for a new data element.

Algorithm

We assume **A** is an array with **N** elements. The maximum numbers of elements it can store is defined by **MAX**.

```
begin
IF N = MAX, return ELSE
N = N + 1
SEEK Location index
For All Elements from A[index] to A[N] Move to next adjacent location
A[index] = New_Element end
```

Implementation in C

```
#include <stdio.h> #define MAX 5
void main() {
int array[MAX] = {1, 2, 4, 5};

int N = 4;
int i = 0;
int index = 2; int value = 3; // number of elements in array // loop variable
// index location to insert new value // new data element to be inserted // print array before
insertion
printf("Printing array before insertion −\n");

for(i = 0; i < N; i++) {
printf("array[%d] = %d \n", i, array[i]);
}

// now shift rest of the elements downwards for(i = N; i >= index; i—) {
array[i+1] = array[i];
}
// add new element at first position array[index] = value;
// increase N to reflect number of elements N++;
// print to confirm
printf("Printing array after insertion −\n");

for(i = 0; i < N; i++) {
printf("array[%d] = %d\n", i, array[i]);
}
}
```

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

```
Printing array before insertion − array[0] = 1
array[1] = 2
array[2] = 4
array[3] = 5
Printing array after insertion − array[0] = 1
array[1] = 2
array[3] = 4
array[4] = 5
```

InsertionAftertheGivenIndexof an Array

In this scenario we are given a location (**index**) of an array after which a new data element (**value**) has to be inserted. Only the seek process varies, the rest of the activities are the same as in the previous example.

Algorithm

We assume **A** is an array with **N** elements. The maximum numbers of elements it can store is defined by **MAX**.

```
begin
IF N = MAX, return ELSE
N = N + 1
SEEK Location index
For All Elements from A[index + 1] to A[N] Move to next adjacent location
A[index + 1] = New_Element end
```

Implementation in C

```
#include <stdio.h> #define MAX 5
void main() {
int array[MAX] = {1, 2, 4, 5};

int N = 4; // number of elements in array
int i = 0; // loop variable
int index = 1; // index location after which value will be inserted int value = 3; // new data element to be inserted

// print array before insertion
printf("Printing array before insertion −\n");

for(i = 0; i < N; i++) {
printf("array[%d] = %d \n", i, array[i]);
}

// now shift rest of the elements downwards for(i = N; i >= index + 1; i—) {
array[i + 1] = array[i];
}
// add new element at first position array[index + 1] = value;
// increase N to reflect number of elements N++;
// print to confirm
printf("Printing array after insertion −\n");

for(i = 0; i < N; i++) {
printf("array[%d] = %d\n", i, array[i]);
}
}
```

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

```
Printing array before insertion − array[0] = 1
array[1] = 2
array[2] = 4
array[3] = 5
Printing array after insertion − array[0] = 1
array[2] = 3
array[3] = 4
array[4] = 5
```

InsertionBeforetheGivenIndexof an Array

In this scenario we are given a location **(index)** of an array before which a new data element (**value**) has to be inserted. This time we seek till **index-1**, i.e., one location ahead of the given index. Rest of the activities are the same as in the previous example.

Algorithm

We assume **A** is an array with **N** elements. The maximum numbers of elements it can store is defined by **MAX**.

```
begin
IF N = MAX, return ELSE
N = N + 1
SEEK Location index
For All Elements from A[index - 1] to A[N] Move to next adjacent location
A[index - 1] = New_Element end
```

Implementation in C

```
#include <stdio.h> #define MAX 5
void main() {
int array[MAX] = {1, 2, 4, 5};

int N = 4;
int i = 0;
int index = 3; int value = 3; // number of elements in array
// loop variable
// index location before which value will be inserted // new data element to be inserted

// print array before insertion
printf("Printing array before insertion −\n"); for(i = 0; i < N; i++) {

printf("array[%d] = %d \n", i, array[i]); }
// now shift rest of the elements downwards for(i = N; i >= index + 1; i—) {
array[i + 1] = array[i];
}
// add new element at first position array[index + 1] = value;
// increase N to reflect number of elements N++;
// print to confirm
printf("Printing array after insertion −\n");

for(i = 0; i < N; i++) {
printf("array[%d] = %d\n", i, array[i]);
}
}
```

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

```
Printing array before insertion − array[1] = 2
```

```
array[2] = 4
array[3] = 5
Printing array after insertion − array[0] = 1
array[1] = 2
array[2] = 3
array[3] = 4
array[4] = 5
```

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 Kthposition of LA.

```
1. Start
2. Set J=K
3. Repeat steps 4 and 5 while J < N
4. Set LA[J-1] = LA[J]
5. Set J = J+1
6. Set N = N-1
7. Stop
```

Example

Following is the implementation of the above algorithm −

```
#include <stdio.h>
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 −

```
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
```

SearchOperation

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 −

```
#include <stdio.h>
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 −

```
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 Kthposition of LA.

```
1. Start
2. Set LA[K-1] = ITEM
3. Stop
```

Example

Following is the implementation of the above algorithm −

```
#include <stdio.h>
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 −

```
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
```

Linked List

10. Linked List─ Basics

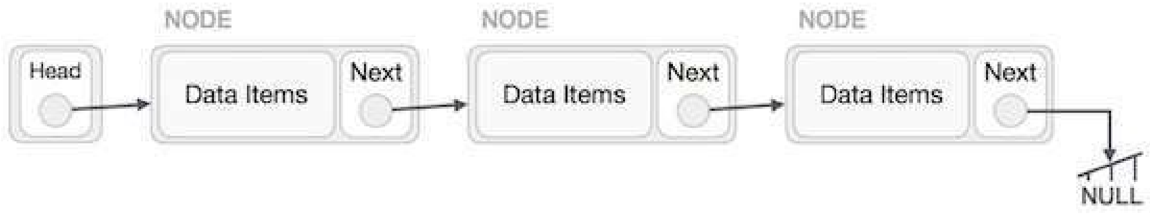
A linked list is a sequence of data structures, which are connected together via links.

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.
Linked List − 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

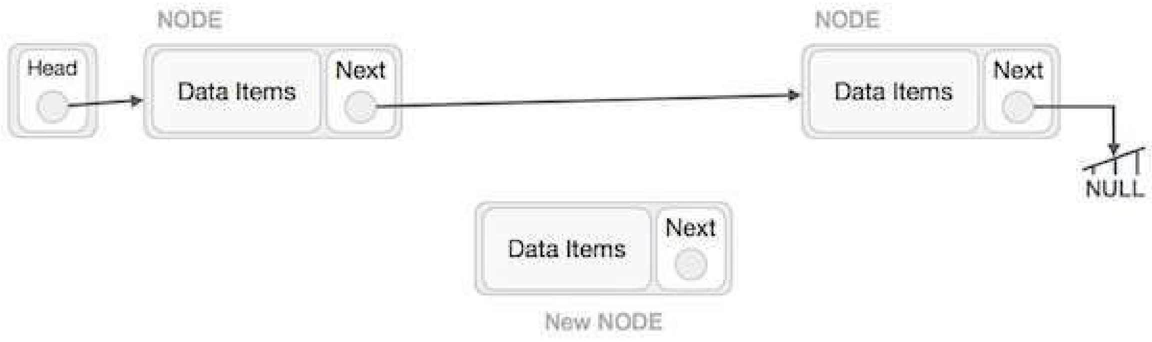
Following are the various types of linked list.
Simple Linked List − Item navigation is forward only.
Doubly Linked List − Items can be navigated forward and backward.
Circular Linked List − Last item contains link of the first element as next and the first element has a link to the last element as previous.

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

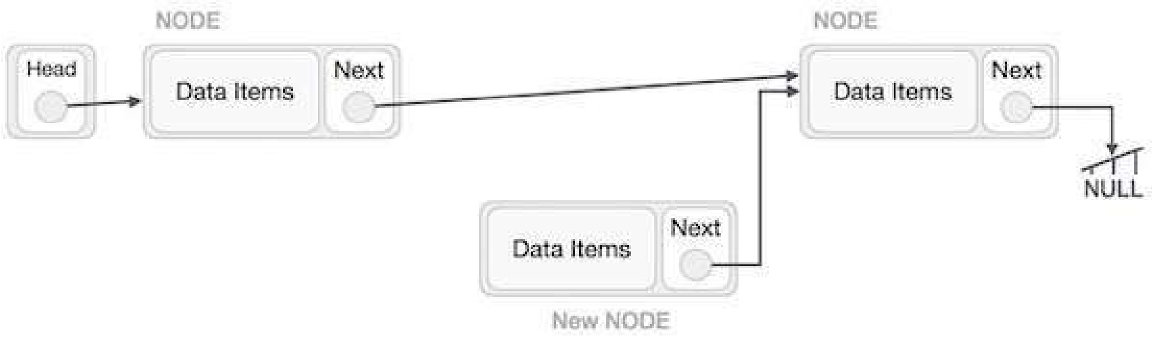
Adding a new node in linked list is a more than one step activity. We shall learn this with diagrams here. First, create a node using the same structure and find the location where it has to be inserted.



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

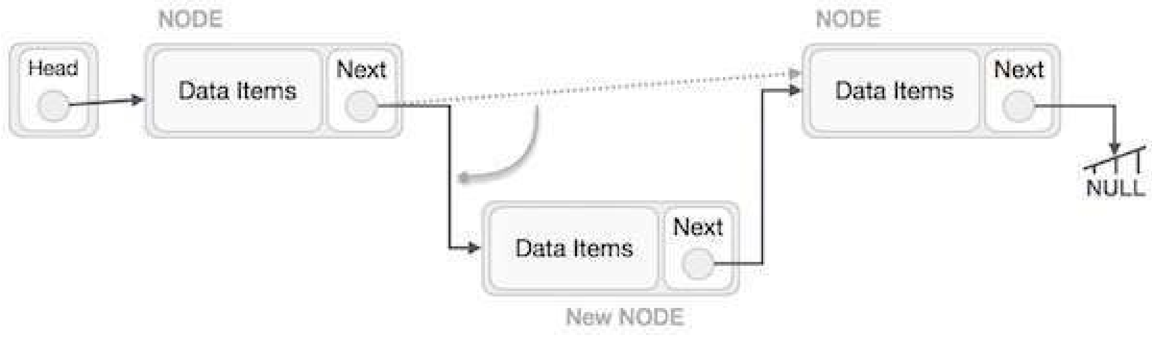
NewNode.next −> RightNode;

It should look like this −

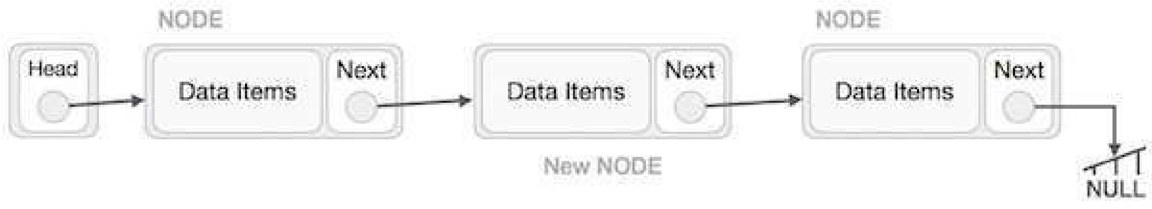


Now, the next node at the left should point to the new node.

LeftNode.next −> NewNode;



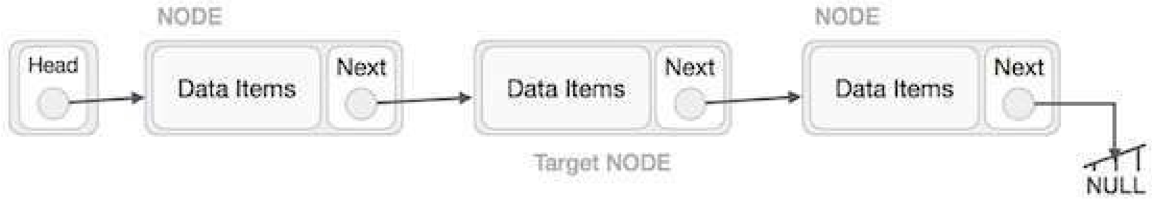
This will put the new node in the middle of the two. The new list should look like this −



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.

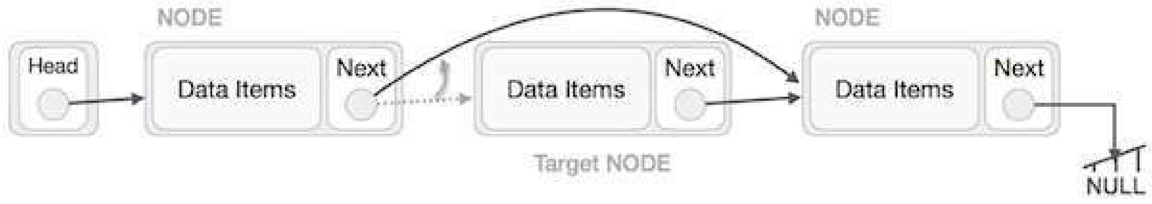
Deletion Operation

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.



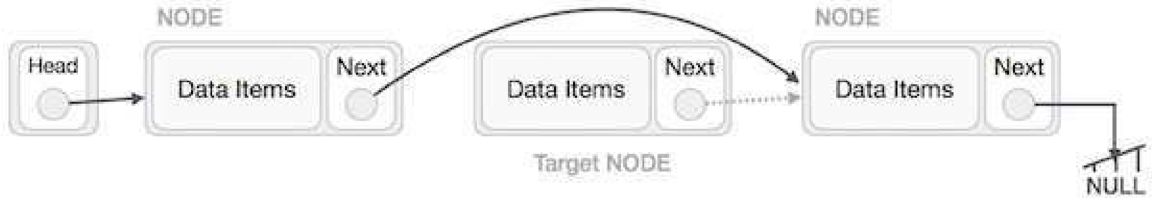
The left (previous) node of the target node now should point to the next node of the target node −

LeftNode.next −> TargetNode.next;

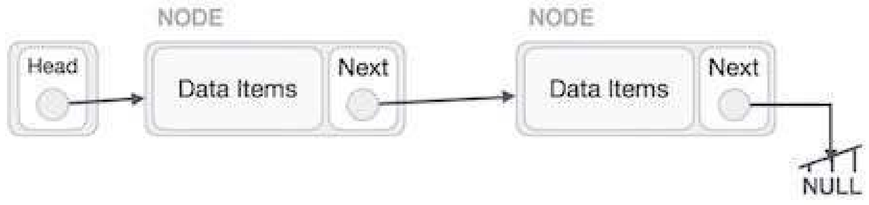


This will remove the link that was pointing to the target node. Now, using the following code, we will remove what the target node is pointing at.

TargetNode.next −> NULL;

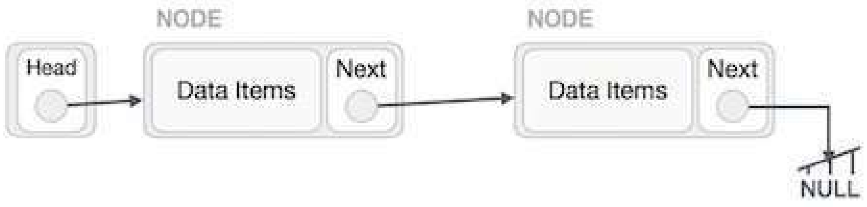


We need to use the deleted node. We can keep that in memory otherwise we can simply deallocate memory and wipe off the target node completely.

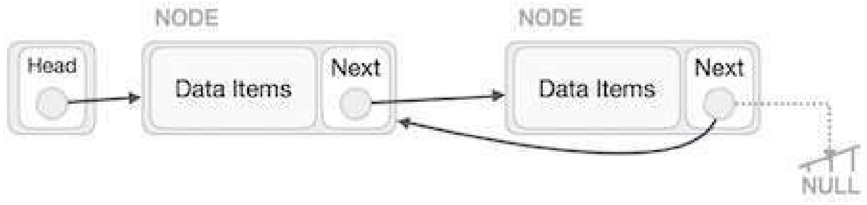


Reverse Operation

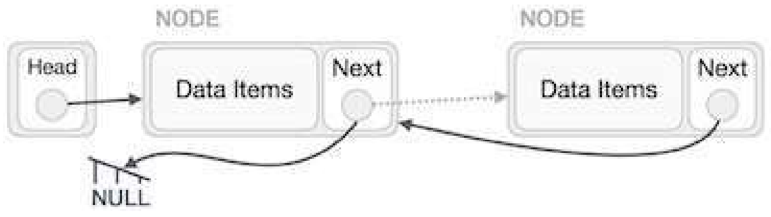
This operation is a thorough one. We need to make the last node to be pointed by the head node and reverse the whole linked list.



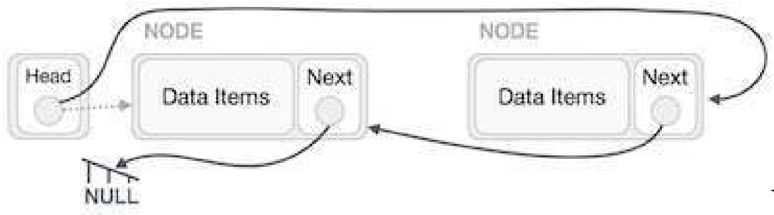
First, we traverse to the end of the list. It should be pointing to NULL. Now, we shall make it point to its previous node −



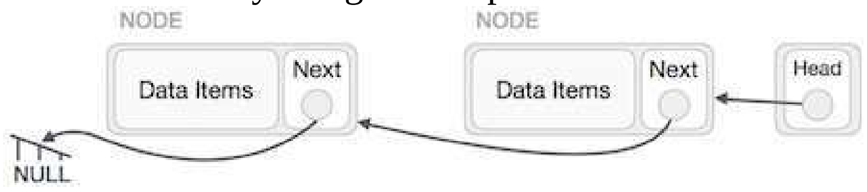
We have to make sure that the last node is not the lost node. So we'll have some temp node, which looks like the head node pointing to the last node. Now, we shall make all left side nodes point to their previous nodes one by one.



Except the node (first node) pointed by the head node, all nodes should point to their predecessor, making them their new successor. The first node will point to NULL.



We'll make the head node point to the new first node by using the temp node.



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

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

```
#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;
}
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 −

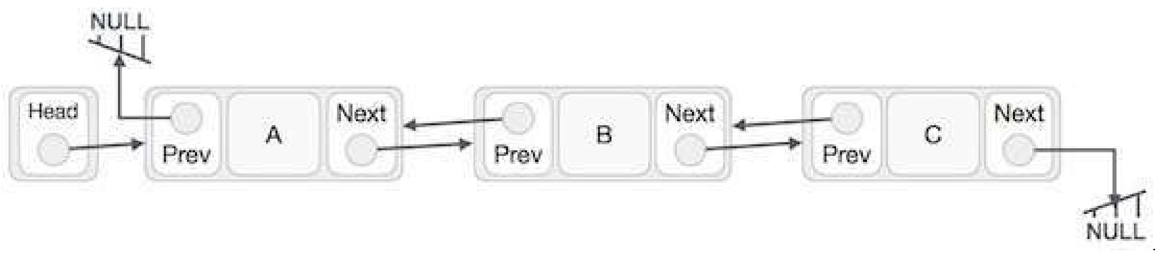
```
Original List:
[ (6,56) (5,40) (4,1) (3,30) (2,20) (1,10) ] Deleted value:(6,56)
Deleted value:(5,40)
Deleted value:(4,1)
Deleted value:(3,30)
Deleted value:(2,20)
Deleted value:(1,10)
List after deleting all items:
[ ]
Restored List:
[ (6,56) (5,40) (4,1) (3,30) (2,20) (1,10) ] Element found: (4,1)
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) ]
```

11. 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.
Linked List − 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 a link field called next.
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.

```
//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.

```
//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 attheEndof an Operation

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

```
//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.

Doubly Linked List ProgramінC

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

```
#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; printf(" ");

}
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;

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 = key;

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;

}

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 −

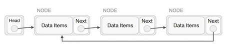
```
List (First to Last):
[ (6,56) (5,40) (4,1) (3,30) (2,20) (1,10) ]

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) (4,13) (3,30) (2,20) ] List , after delete key(4) :
[ (5,40) (4,13) (3,30) (2,20) ]
```

12. Circular Linked List

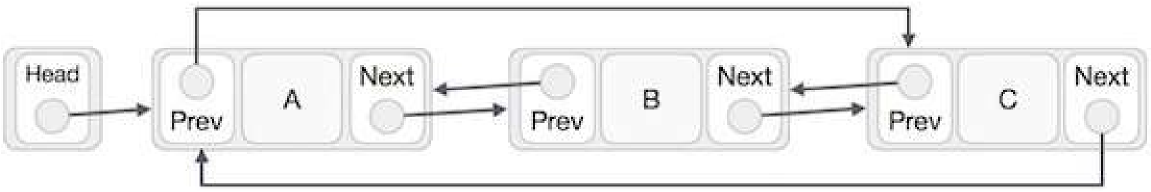
Circular Linked List is a variation of Linked list in which the first element points to the last element and the last element points to the first element. Both Singly Linked List and Doubly Linked List can be made into a circular linked list.

SinglyLinkedListasCircular

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

Doubly LinkedListasCircular

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.

```
//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;
}
}
```

Deletion Operation

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

```
//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 List Operation

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

```
//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(” ]”); }
```

To know about its implementation in C programming language, please click here.

Circular Linked List Programin 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

```
#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(" ]"); }
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 −

```
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: [ ]
```

Stack & Queue

13. Stack Data Structures & Algorithms

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.