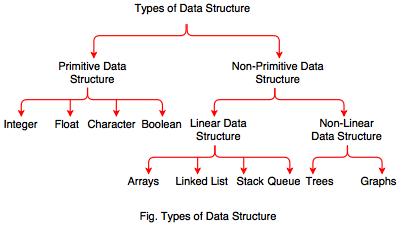
What is Data Structure?

* Data structure is an arrangement of data in computer's memory. It makes the data quickly available to the processor for required operations.
* It is a software artifact which allows data to be stored, organized and accessed.
* It is a structure program used to store ordered data, so that various operations can be performed on it easily.  
  **For example,** if we have an employee's data like name 'ABC' and salary 10000. Here, 'ABC' is of String data type and 10000 is of Float data type.  
  We can organize this data as a record like Employee record and collect & store employee's records in a file or database as a data structure like 'ABC' 10000, 'PQR' 15000, 'STU' 5000.
* Data structure is about providing data elements in terms of some relationship for better organization and storage.
* It is a specialized format for organizing and storing data that can be accessed within appropriate ways.

Why is Data Structure important?

* Data structure is important because it is used in almost every program or software system.
* It helps to write efficient code, structures the code and solve problems.
* Data can be maintained more easily by encouraging a better design or implementation.
* Data structure is just a container for the data that is used to store, manipulate and arrange. It can be processed by algorithms.  
  **For example,** while using a shopping website like Flipkart or Amazon, the users know their last orders and can track them. The orders are stored in a database as records.  
  However, when the program needs them so that it can pass the data somewhere else  (such as to a warehouse) or display it to the user, it loads the data in some form of data structure.

Types of Data Structure



**A. Primitive Data Type**

* Primitive data types are the data types available in most of the programming languages.
* These data types are used to represent single value.
* It is a basic data type available in most of the programming language.

|  |  |
| --- | --- |
| **Data type** | **Description** |
| Integer | Used to represent a number without decimal point. |
| Float | Used to represent a number with decimal point. |
| Character | Used to represent single character. |
| Boolean | Used to represent logical values either true or false. |

**B. Non-Primitive Data Type**

* Data type derived from primary data types are known as Non-Primitive data types.
* Non-Primitive data types are used to store group of values.

**It can be divided into two types:**  
  
1. Linear Data Structure  
2. Non-Linear Data Structure  
  
**1. Linear Data Structure**

* Linear data structure traverses the data elements sequentially.
* In linear data structure, only one data element can directly be reached.
* It includes array, linked list, stack and queues.

|  |  |
| --- | --- |
| **Types** | **Description** |
| Arrays | Array is a collection of elements. It is used in mathematical problems like matrix, algebra etc. each element of an array is referenced by a subscripted variable or value, called subscript or index enclosed in parenthesis. |
| Linked list | Linked list is a collection of data elements. It consists of two parts: Info and Link. Info gives information and Link is an address of next node. Linked list can be implemented by using pointers. |
| Stack | Stack is a list of elements. In stack, an element may be inserted or deleted at one end which is known as Top of the stack. It performs two operations: Push and Pop. Push means adding an element in stack and Pop means removing an element in stack. It is also called Last-in-First-out (LIFO). |
| Queue | Queue is a linear list of element. In queue, elements are added at one end called rear and the existing elements are deleted from other end called front. It is also called as First-in-First-out (FIFO). |

**2. Non-Linear Data Structure**

* Non-Linear data structure is opposite to linear data structure.
* In non-linear data structure, the data values are not arranged in order and a data item is connected to several other data items.
* It uses memory efficiently. Free contiguous memory is not required for allocating data items.
* It includes trees and graphs.

|  |  |
| --- | --- |
| **Type** | **Description** |
| Tree | Tree is a flexible, versatile and powerful non-linear data structure. It is used to represent data items processing  hierarchical relationship between the grandfather and his children & grandchildren. It is an ideal data structure for representing hierarchical data. |
| Graph | Graph is a non-linear data structure which consists of a finite set of ordered pairs called edges. Graph is a set of elements connected by edges. Each elements are called a vertex and node. |

## Types of Data Structure

Basically, data structures are divided into two categories:

* Linear data structure
* Non-linear data structure

Let's learn about each type in detail.

## Linear data structures

In linear data structures, the elements are arranged in sequence one after the other. Since elements are arranged in particular order, they are easy to implement.

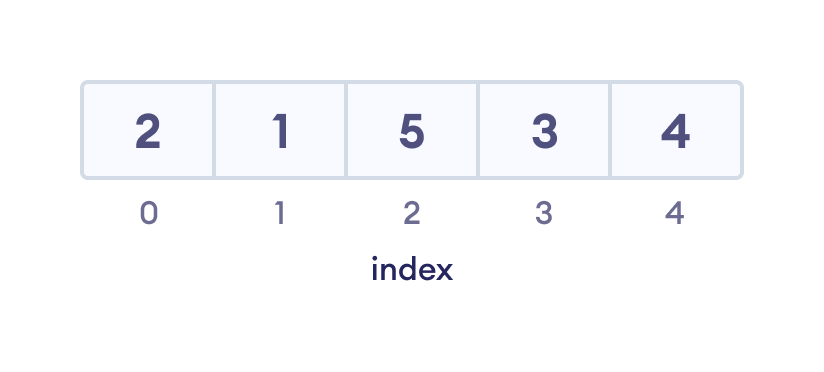
However, when the complexity of the program increases, the linear data structures might not be the best choice because of operational complexities.

**Popular linear data structures are:**

### 1. Array Data Structure

In an array, elements in memory are arranged in continuous memory. All the elements of an array are of the same type. And, the type of elements that can be stored in the form of arrays is determined by the programming language.

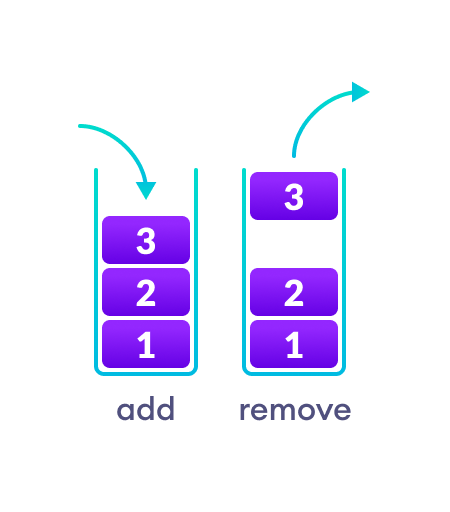
To learn more, visit [Java Array](https://www.programiz.com/java-programming/arrays).

An array with each element represented by an index

### 2. Stack Data Structure

In stack data structure, elements are stored in the LIFO principle. That is, the last element stored in a stack will be removed first.

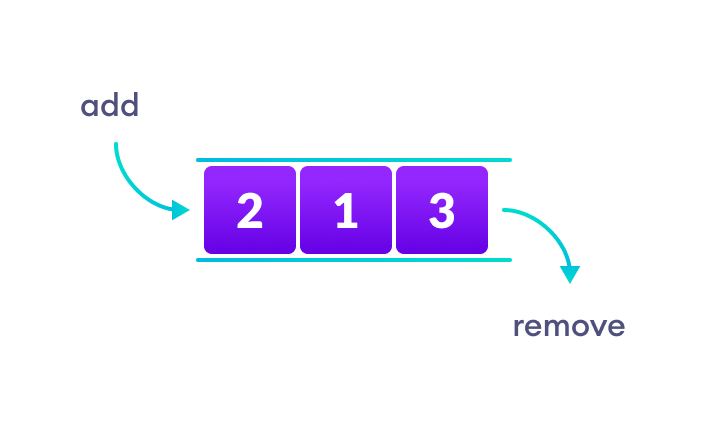
It works just like a pile of plates where the last plate kept on the pile will be removed first. To learn more, visit [Stack Data Structure](https://www.programiz.com/dsa/stack).

In a stack, operations can be perform only from one end (top here).

### 3. Queue Data Structure

Unlike stack, the queue data structure works in the FIFO principle where first element stored in the queue will be removed first.

It works just like a queue of people in the ticket counter where first person on the queue will get the ticket first. To learn more, visit [Queue Data Structure](https://www.programiz.com/dsa/queue).

In a queue, addition and removal are performed from separate ends.

### 4. Linked List Data Structure

In linked list data structure, data elements are connected through a series of nodes. And, each node contains the data items and address to the next node.

Abstract Data type (ADT)

**What is ADT?**

* ADT stands for **Abstract Data Type.**
* It is an abstraction of a data structure.
* Abstract data type is a mathematical model of a data structure.
* It describes a container which holds a finite number of objects where the objects may be associated through a given binary relationship.
* It is a logical description of how we view the data and the operations allowed without regard to how they will be implemented.
* ADT concerns only with what the data is representing and not with how it will eventually be constructed.
* It is a set of objects and operations. For example, List, Insert, Delete, Search, Sort.

**It consists of following three parts:**  
  
1. Data  
2. Operation  
3. Error  
  
**1. Data** describes the structure of the data used in the ADT.  
  
**2. Operation** describes valid operations for the ADT. It describes its interface.  
  
**3. Error** describes how to deal with the errors that can occur.

**Advantages of ADT**

* ADT is reusable and ensures robust data structure.
* It reduces coding efforts.
* Encapsulation ensures that data cannot be corrupted.
* ADT is based on principles of Object Oriented Programming (OOP) and Software Engineering (SE).
* It specifies error conditions associated with operations.

# What is an Algorithm?

In this tutorial, we will learn what algorithms are with the help of examples.

In computer programming terms, an algorithm is a set of well-defined instructions to solve a particular problem. It takes a set of input(s) and produces the desired output. For example,

An algorithm to add two numbers:

1. Take two number inputs
2. Add numbers using the + operator
3. Display the result

## Qualities of a Good Algorithm

* Input and output should be defined precisely.
* Each step in the algorithm should be clear and unambiguous.
* Algorithms should be most effective among many different ways to solve a problem.
* An algorithm shouldn't include computer code. Instead, the algorithm should be written in such a way that it can be used in different programming languages.

## Algorithm Examples

[Algorithm to add two numbers](https://www.programiz.com/dsa/algorithm#add-numbers)

[Algorithm to find the largest among three numbers](https://www.programiz.com/dsa/algorithm#largest)

[Algorithm to find all the roots of the quadratic equation](https://www.programiz.com/dsa/algorithm#quadratic)

[Algorithm to find the factorial](https://www.programiz.com/dsa/algorithm#factorial)

[Algorithm to check prime number](https://www.programiz.com/dsa/algorithm#prime)

[Algorithm of Fibonacci series](https://www.programiz.com/dsa/algorithm#fibonacci)

### Algorithm 1: Add two numbers entered by the user

Step 1: Start

Step 2: Declare variables num1, num2 and sum.

Step 3: Read values num1 and num2.

Step 4: Add num1 and num2 and assign the result to sum.

sum←num1+num2

Step 5: Display sum

Step 6: Stop

### Algorithm 2: Find the largest number among three numbers

Step 1: Start

Step 2: Declare variables a,b and c.

Step 3: Read variables a,b and c.

Step 4: If a > b

If a > c

Display a is the largest number.

Else

Display c is the largest number.

Else

If b > c

Display b is the largest number.

Else

Display c is the greatest number.

Step 5: Stop

### Algorithm 3: Find Roots of a Quadratic Equation ax2 + bx + c = 0

Step 1: Start

Step 2: Declare variables a, b, c, D, x1, x2, rp and ip;

Step 3: Calculate discriminant

D ← b2-4ac

Step 4: If D ≥ 0

r1 ← (-b+√D)/2a

r2 ← (-b-√D)/2a

Display r1 and r2 as roots.

Else

Calculate real part and imaginary part

rp ← -b/2a

ip ← √(-D)/2a

Display rp+j(ip) and rp-j(ip) as roots

Step 5: Stop

### Algorithm 4: Find the factorial of a number

Step 1: Start

Step 2: Declare variables n, factorial and i.

Step 3: Initialize variables

factorial ← 1

i ← 1

Step 4: Read value of n

Step 5: Repeat the steps until i = n

5.1: factorial ← factorial\*i

5.2: i ← i+1

Step 6: Display factorial

Step 7: Stop

### Algorithm 5: Check whether a number is prime or not

Step 1: Start

Step 2: Declare variables n, i, flag.

Step 3: Initialize variables

flag ← 1

i ← 2

Step 4: Read n from the user.

Step 5: Repeat the steps until i=(n/2)

5.1 If remainder of n÷i equals 0

flag ← 0

Go to step 6

5.2 i ← i+1

Step 6: If flag = 0

Display n is not prime

else

Display n is prime

Step 7: Stop

### Algorithm 6: Find the Fibonacci series till the term less than 1000

Step 1: Start

Step 2: Declare variables first\_term,second\_term and temp.

Step 3: Initialize variables first\_term ← 0 second\_term ← 1

Step 4: Display first\_term and second\_term

Step 5: Repeat the steps until second\_term ≤ 1000

5.1: temp ← second\_term

5.2: second\_term ← second\_term + first\_term

5.3: first\_term ← temp

5.4: Display second\_term

Step 6: Stop

**Flowchart** is a diagrammatic representation of sequence of logical steps of a program. Flowcharts use simple geometric shapes to depict processes and arrows to show relationships and process/data flow.

## Flowchart Symbols

Here is a chart for some of the common symbols used in drawing flowcharts.

|  |  |  |
| --- | --- | --- |
| **Symbol** | **Symbol Name** | **Purpose** |
| Start Stop | Start/Stop | Used at the beginning and end of the algorithm to show start and end of the program. |
| Process | Process | Indicates processes like mathematical operations. |
| Input/ Output | Input/ Output | Used for denoting program inputs and outputs. |
| Decision | Decision | Stands for decision statements in a program, where answer is usually Yes or No. |
| Arrow | Arrow | Shows relationships between different shapes. |
| On-page Connector | On-page Connector | Connects two or more parts of a flowchart, which are on the same page. |
| Off-page Connector | Off-page Connector | Connects two parts of a flowchart which are spread over different pages. |

### Guidelines for Developing Flowcharts

These are some points to keep in mind while developing a flowchart −

* Flowchart can have only one start and one stop symbol
* On-page connectors are referenced using numbers
* Off-page connectors are referenced using alphabets
* General flow of processes is top to bottom or left to right
* Arrows should not cross each other

# Pointer

Pointer is used to points the address of the value stored anywhere in the computer memory. To obtain the value stored at the location is known as dereferencing the pointer. Pointer improves the performance for repetitive process such as:

* Traversing String
* Lookup Tables
* Control Tables
* Tree Structures

Pointer Details

* **Pointer arithmetic:** There are four arithmetic operators that can be used in pointers: ++, --, +, -
* **Array of pointers:** You can define arrays to hold a number of pointers.
* **Pointer to pointer:** C allows you to have pointer on a pointer and so on.
* **Passing pointers to functions in C:** Passing an argument by reference or by address enable the passed argument to be changed in the calling function by the called function.
* **Return pointer from functions in C:** C allows a function to return a pointer to the local variable, static variable and dynamically allocated memory as well.
* #include <stdio.h>
* **int** main( )
* {
* **int** a = 5;
* **int** \*b;
* **int** \*\*c;
* b = &a;
* c = &b;
* printf ("value of a = %d\n", a);
* printf ("value of a = %d\n", \*(&a));
* printf ("value of a = %d\n", \*b);
* printf ("value of a = %d\n", \*\*c);
* printf ("value of b = address of a = %u\n", b);
* printf ("value of c = address of b = %u\n", c);
* printf ("address of a = %u\n", &a);
* printf ("address of a = %u\n", b);
* printf ("address of a = %u\n", \*c);
* printf ("address of b = %u\n", &b);
* printf ("address of b = %u\n", c);
* printf ("address of c = %u\n", &c);
* **return** 0;
* }

# Structure

A structure is a composite data type that defines a grouped list of variables that are to be placed under one name in a block of memory. It allows different variables to be accessed by using a single pointer to the structure.

**Syntax**

1. struct structure\_name
2. {
3. data\_type member1;
4. data\_type member2;
5. .
6. .
7. data\_type memeber;
8. };

### Advantages

* It can hold variables of different data types.
* We can create objects containing different types of attributes.
* It allows us to re-use the data layout across programs.
* It is used to implement other data structures like linked lists, stacks, queues, trees, graphs etc.

**Program**

1. #include<stdio.h>
2. #include<conio.h>
3. **void** main( )
4. {
5. struct employee
6. {
7. **int** id ;
8. **float** salary ;
9. **int** mobile ;
10. } ;
11. struct employee e1,e2,e3 ;
12. clrscr();
13. printf ("\nEnter ids, salary & mobile no. of 3 employee\n"
14. scanf ("%d %f %d", &e1.id, &e1.salary, &e1.mobile);
15. scanf ("%d%f %d", &e2.id, &e2.salary, &e2.mobile);
16. scanf ("%d %f %d", &e3.id, &e3.salary, &e3.mobile);
17. printf ("\n Entered Result ");
18. printf ("\n%d %f %d", e1.id, e1.salary, e1.mobile);
19. printf ("\n%d%f %d", e2.id, e2.salary, e2.mobile);
20. printf ("\n%d %f %d", e3.id, e3.salary, e3.mobile);
21. getch();
22. }
23. **C Pointers to struct**
24. Here's how you can create pointers to structs.
25. struct name {
26. member1;
27. member2;
28. .
29. .
30. };
31. int main()
32. {
33. struct name \*ptr, Harry;
34. }
35. Here, ptr is a pointer to struct.
36. **Example: Access members using Pointer**
37. To access members of a structure using pointers, we use the -> operator.
38. #include <stdio.h>
39. struct person
40. {
41. int age;
42. float weight;
43. };
44. int main()
45. {
46. struct person \*personPtr, person1;
47. personPtr = &person1;
48. printf("Enter age: ");
49. scanf("%d", &personPtr->age);
50. printf("Enter weight: ");
51. scanf("%f", &personPtr->weight);
52. printf("Displaying:\n");
53. printf("Age: %d\n", personPtr->age);
54. printf("weight: %f", personPtr->weight);
55. return 0;
56. }

## Dynamic memory allocation of structs

Before you proceed this section, we recommend you to check [C dynamic memory allocation](https://www.programiz.com/c-programming/c-dynamic-memory-allocation).

Sometimes, the number of struct variables you declared may be insufficient. You may need to allocate memory during run-time. Here's how you can achieve this in C programming.

### Example: Dynamic memory allocation of structs

#include <stdio.h>

#include <stdlib.h>

struct person {

int age;

float weight;

char name[30];

};

int main()

{

struct person \*ptr;

int i, n;

printf("Enter the number of persons: ");

scanf("%d", &n);

// allocating memory for n numbers of struct person

ptr = (struct person\*) malloc(n \* sizeof(struct person));

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

{

printf("Enter first name and age respectively: ");

// To access members of 1st struct person,

// ptr->name and ptr->age is used

// To access members of 2nd struct person,

// (ptr+1)->name and (ptr+1)->age is used

scanf("%s %d", (ptr+i)->name, &(ptr+i)->age);

}

printf("Displaying Information:\n");

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

printf("Name: %s\tAge: %d\n", (ptr+i)->name, (ptr+i)->age);

return 0;

}

Union in C

Like Structures, union is a user defined data type. In union, all members share the same memory location.

For example in the following C program, both x and y share the same location. If we change x, we can see the changes being reflected in y.

#include <stdio.h>

// Declaration of union is same as structures

union test {

int x, y;

};

int main()

{

// A union variable t

union test t;

t.x = 2; // t.y also gets value 2

printf("After making x = 2:\n x = %d, y = %d\n\n",

t.x, t.y);

t.y = 10; // t.x is also updated to 10

printf("After making y = 10:\n x = %d, y = %d\n\n",

t.x, t.y);

return 0;

}

Pointers to unions?

Like structures, we can have pointers to unions and can access members using the arrow operator (->). The following example demonstrates the same.

#include <stdio.h>

union test {

int x;

char y;

};

int main()

{

union test p1;

p1.x = 65;

// p2 is a pointer to union p1

union test\* p2 = &p1;

// Accessing union members using pointer

printf("%d %c", p2->x, p2->y);

return 0;

}

Output:

65 A

C Recursion

A function that calls itself is known as a recursive function. And, this technique is known as recursion.

How recursion works?

void recurse()

{

... .. ...

recurse();

... .. ...

}

int main()

{

... .. ...

recurse();

... .. ...

}

How recursion works in C programming?

Working of Recursion

The recursion continues until some condition is met to prevent it.

To prevent infinite recursion, if...else statement (or similar approach) can be used where one branch makes the recursive call, and other doesn't.

Example: Sum of Natural Numbers Using Recursion

#include <stdio.h>

int sum(int n);

int main() {

int number, result;

printf("Enter a positive integer: ");

scanf("%d", &number);

result = sum(number);

printf("sum = %d", result);

return 0;

}

int sum(int n) {

if (n != 0)

// sum() function calls itself

return n + sum(n-1);

else

return n;

}

Output

Enter a positive integer:3

sum = 6

Advantages and Disadvantages of Recursion

Recursion makes program elegant. However, if performance is vital, use loops instead as recursion is usually much slower.

That being said, recursion is an important concept. It is frequently used in data structure and algorithms. For example, it is common to use recursion in problems such as tree traversal.

typedef in C

The typedef is a keyword used in C programming to provide some meaningful names to the already existing variable in the C program. It behaves similarly as we define the alias for the commands. In short, we can say that this keyword is used to redefine the name of an already existing variable.

Syntax of typedef

typedef <existing\_name> <alias\_name>

In the above syntax, 'existing\_name' is the name of an already existing variable while 'alias name' is another name given to the existing variable.

For example, suppose we want to create a variable of type unsigned int, then it becomes a tedious task if we want to declare multiple variables of this type. To overcome the problem, we use a typedef keyword.

typedef unsigned int unit;

In the above statements, we have declared the unit variable of type unsigned int by using a typedef keyword.

Now, we can create the variables of type unsigned int by writing the following statement:

unit a, b;

instead of writing:

unsigned int a, b;

Till now, we have observed that the typedef keyword provides a nice shortcut by providing an alternative name for an already existing variable. This keyword is useful when we are dealing with the long data type especially, structure declarations.

Let's understand through a simple example.

#include <stdio.h>

int main()

{

typedef unsigned int unit;

unit i,j;

i=10;

j=20;

printf("Value of i is :%d",i);

printf("\nValue of j is :%d",j);

return 0;

}

Output

Value of i is :10

Value of j is :20

Using typedef with structures

Consider the below structure declaration:

struct student

{

char name[20];

int age;

};

struct student s1;

In the above structure declaration, we have created the variable of student type by writing the following statement:

struct student s1;

The above statement shows the creation of a variable, i.e., s1, but the statement is quite big. To avoid such a big statement, we use the typedef keyword to create the variable of type student.

struct student

{

char name[20];

int age;

};

typedef struct student stud;

stud s1, s2;

In the above statement, we have declared the variable stud of type struct student. Now, we can use the stud variable in a program to create the variables of type struct student.

The above typedef can be written as:

typedef struct student

{

char name[20];

int age;

} stud;

stud s1,s2;

From the above declarations, we conclude that typedef keyword reduces the length of the code and complexity of data types. It also helps in understanding the program.

Let's see another example where we typedef the structure declaration.

#include <stdio.h>

typedef struct student

{

char name[20];

int age;

}stud;

int main()

{

stud s1;

printf("Enter the details of student s1: ");

printf("\nEnter the name of the student:");

scanf("%s",&s1.name);

printf("\nEnter the age of student:");

scanf("%d",&s1.age);

printf("\n Name of the student is : %s", s1.name);

printf("\n Age of the student is : %d", s1.age);

return 0;

}

Output

Enter the details of student s1:

Enter the name of the student: Peter

Enter the age of student: 28

Name of the student is : Peter

Age of the student is : 28

Using typedef with pointers

We can also provide another name or alias name to the pointer variables with the help of the typedef.

For example, we normally declare a pointer, as shown below:

int\* ptr;

We can rename the above pointer variable as given below:

typedef int\* ptr;

In the above statement, we have declared the variable of type int\*. Now, we can create the variable of type int\* by simply using the 'ptr' variable as shown in the below statement:

ptr p1, p2 ;

In the above statement, p1 and p2 are the variables of type 'ptr'.

Linked List using Arrays

Array of linked list is an important data structure used in many applications. It is an interesting structure to form a useful data structure. It combines static and dynamic structure. Static means array and dynamic means linked list used to form a useful data structure. This array of linked list structure is appropriate for applications.

#include<stdio.h>

#define size 5

**int** **main**()

{

**int** arr[size] = {**1**, **20**, **5**, **78**, **30**};

**int** element, pos, i;

printf("Enter position and element**\n**");

scanf("%d%d",&pos,&element);

**if**(pos <= size && pos >= **0**)

{

//shift all the elements from the last index to pos by 1 position to right

**for**(i = size; i > pos; i--)

arr[i] = arr[i-**1**];

//insert element at the given position

arr[pos] = element;

/\*

\* print the new array

\* the new array size will be size+1(actual size+new element)

\* so, use i <= size in for loop

\*/

**for**(i = **0**; i <= size; i++)

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

}

**else**

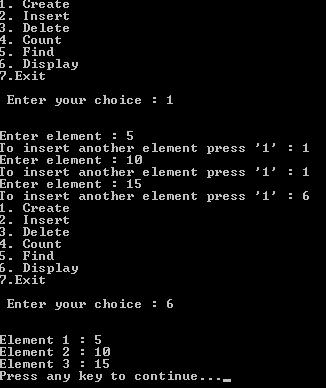
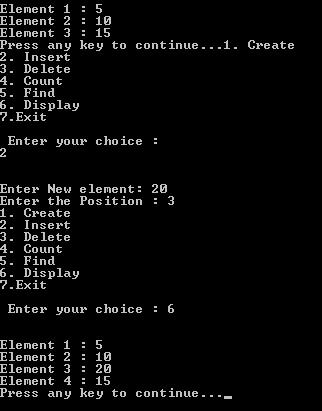
printf("Invalid Position**\n**");

**return** **0**;

}

Example: Demonstrating the Linked List Implementation using Array

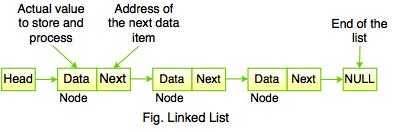
#include<stdio.h>  
#include<conio.h>  
#include<stdlib.h>  
#define MAX 10 // Macro defines maximum no. of elements in the list. It is a user defined data type  
  
struct  
{  
    int list[MAX];  
    int element;     //new element to be inserted  
    int pos;           //position of the element to be inserted or deleted  
    int length;     //total no of elements  
}l;  
enum boolean  
{  
    true, false  
};  
typedef enum boolean boolean;        //function prototypes  
  
int menu(void);           //This function displays the list of operations  
void create(void);       //This function creates initial set of elements  
void insert(int, int);    //This function inserts the element at specified position  
void delet(int);       //This function deletes the element at given position  
void find(int);        //This function finds the position of the given element, if exists  
void display(void);       //This function displays the elements in the list  
boolean islistfull(void);  //This function checks whether the list is full or not boolean  
boolean islistempty(void);        //This function checks whether the list is empty or not  
  
void main()  
{  
    int ch;  
    int element;  
    int pos;  
    l.length = 0;  
    while(1)  
    {  
        ch = menu();  
        switch (ch)  
        {  
             case 1:   l.length = 0;  
             create();  
             break;  
             case 2:  
             if (islistfull() != true)  
             {  
                  printf("Enter New element: ");  
                  scanf("%d", &element);  
                  printf("Enter the Position : ");  
                  scanf("%d", &pos);  
                  insert(element, pos);  
             }  
             else  
             {  
                  printf("List is Full. Cannot insert the element");  
                  printf("\n Press any key to continue...");  
                  getch();  
             }  
             break;  
             case 3:  
             if (islistempty() != true)  
             {  
                  printf("Enter the position of element to be deleted : ");  
                  scanf("%d", &pos);  
                  delet(pos);  
             }  
             else  
             {  
                  printf("List is Empty.");  
                  printf("\n Press any key to continue...");  
                  getch();  
             }  
             break;  
             case 4:  
             printf("No of elements in the list is %d", l.length);  
             printf("\n Press any key to continue...");  
             getch();  
             break;  
             case 5:  
             printf("Enter the element to be searched : ");  
             scanf("%d", &element);  
             find(element);  
             break;  
             case 6:  
             display();  
             break;  
             case 7:  
             printf("Exit");  
             exit(0);  
             break;  
             default:  printf("Invalid Choice");  
             printf("\n Press any key to continue...");  
             getch();  
        }  
    }  
}        //function to display the list of elements  
int menu()  
{  
    int ch;  
    //clrscr();  
    printf("1. Create\n2. Insert\n3. Delete\n4. Count\n5. Find\n6. Display\n7.Exit\n\n Enter your choice : ");  
    scanf("%d", &ch);  
    printf("\n\n");  
    return ch;  
}        
void create(void)  
{  
    int element;  
    int flag=1;  
    while(flag==1)  
    {  
        printf("Enter element : ");  
        scanf("%d", &element);  
        l.list[l.length] = element;  
        l.length++;  
        printf("To insert another element press '1' : ");  
        scanf("%d", &flag);  
    }  
}       
void display(void)  
{  
    int i;  
    for (i=0; i<l.length; i++)  
        printf("Element %d : %d \n", i+1, l.list[i]);  
    printf("Press any key to continue...");  
    getch();  
}    
void insert(int element, int pos)  
{  
    int i;  
    if (pos == 0)  
    {  
        printf("\nCannot insert an element at 0th position")  
        getch();  
        return;  
    }  
  
    if (pos-1 > l.length)  
    {  
        printf("\nOnly %d elements exit. Cannot insert at %d position", l.length, pos);  
        printf("\n Press any key to continue...");  
        getch();  
    }  
    else  
    {  
        for (i=l.length; i>=pos-1; i--)  
        {  
            l.list[i+1] = l.list[i];  
        }  
        l.list[pos-1] = element;  
        l.length++;  
    }  
}       
void delet(int pos)  
{  
    int i;  
    if(pos == 0)  
    {  
        printf("\nCannot delete at an element 0th position");  
        getch();  
        return;  
    }  
    if (pos > l.length)  
    {  
        printf("\n\n Only %d elements exit. Cannot delete", l.length, pos);  
        printf("\n Press any key to continue...");  
        getch();  
        return;  
    }  
    for (i=pos-1; i<l.length; i++)  
    {  
        l.list[i] = l.list[i+1];  
    }  
    l.length--;  
}        
  
void find(int element)  
{  
    int i;  
    int flag = 1;  
  
    for (i=0; i<l.length; i++)  
    {  
        if(l.list[i] == element)  
        {  
            printf ("%d exists at %d position",element, i+1);  
            flag = 0;  
            printf("\n Press any key to continue...");  
            getch();  
            break;  
        }  
    }  
    if(flag == 1)  
    {  
        printf("Element not found.\n Press any key to continue...");  
        getch();  
    }  
}    
boolean islistfull(void)  
{  
    if (l.length == MAX)  
        return true;  
    else  
        return false;  
}     
boolean islistempty(void)  
{  
    if (l.length == 0)  
        return true;  
    else  
        return false;  
}

**Output:**  
  
**1. Create**  
  
  
  
**2. Insert**  
  
  
  
**3. Delete**  
  


Difference between Array and Linked List

|  |  |
| --- | --- |
| **Array** | **Linked List** |
| Array is a collection of elements having same data type with common name. | Linked list is an ordered collection of elements which are connected by links. |
| Elements can be accessed randomly. | Elements cannot be accessed randomly. It can be accessed only sequentially. |
| Array elements can be stored in consecutive manner in memory. | Linked list elements can be stored at any available place as address of node is stored in previous node. |
| Insert and delete operation takes more time in array. | Insert and delete operation cannot take more time. It performs operation in fast and in easy way. |
| Memory is allocated at compile time. | Memory is allocated at run time. |
| It can be single dimensional, two dimensional or multidimensional. | It can be singly, doubly or circular linked list. |
| Each array element is independent and does not have a connection with previous element or with its location. | Location or address of element is stored in the link part of previous element or node. |
| Array elements cannot be added, deleted once it is declared. | The nodes in the linked list can be added and deleted from the list. |
| In array, elements can be modified easily by identifying the index value. | In linked list, modifying the node is a complex process. |
| Pointer cannot be used in array. So, it does not require extra space in memory for pointer. | Pointers are used in linked list. Elements are maintained using pointers or links. So, it requires extra memory space for pointers. |

What is Linked List?

Linked list is a linear data structure. It is a collection of data elements, called nodes pointing to the next node by means of a pointer.  
  
Linked list is used to create trees and graphs.  
  
In linked list, each node consists of its own data and the address of the next node and forms a chain.  
  
  
  
The above figure shows the sequence of linked list which contains data items connected together via links. It can be visualized as a chain of nodes, where every node points to the next node.  
  
Linked list contains a link element called **first** and each link carries a **data item**. Entry point into the linked list is called the**head of the list.**  
  
Link field is called next and each link is linked with its next link. Last link carries a link to null to mark the end of the list.  
  
**Note:** Head is not a separate node but it is a reference to the first node. If the list is empty, the head is a null reference.  
  
Linked list is a dynamic data structure. While accessing a particular item, start at the head and follow the references until you get that data item.  
  
**Linked list is used while dealing with an unknown number of objects:**  
  
linked list example  
  
In the above diagram, Linked list contains two fields - First field contains value and second field contains a link to the next node. The last node signifies the end of the list that means NULL.  
  
The real life **example of Linked List** is that of Railway Carriage. It starts from engine and then the coaches follow. Coaches can traverse from one coach to other, if they connected to each other.

**Advantages of Linked List**

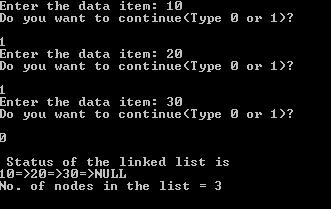
* Linked list is dynamic in nature which allocates the memory when required.
* In linked list, stack and queue can be easily executed.
* It reduces the access time.
* Insert and delete operation can be easily implemented in linked list.

**Disadvantages of Linked List**

* Reverse traversing is difficult in linked list.
* Linked list has to access each node sequentially; no element can be accessed randomly.
* In linked list, the memory is wasted as pointer requires extra memory for storage.

Example: Program to create a simple linked list.

#include<stdio.h>  
#include <stdlib.h>  
int main()  
{  
   struct node  
   {  
      int num;  
      struct node \*ptr;  
   };  
   typedef struct node NODE;  
   NODE \*head, \*first, \*temp=0;  
   int count = 0;  
   int choice = 1;  
   first = 0;  
   while(choice)  
   {  
      head =(NODE\*) malloc(sizeof(NODE));  
      printf("Enter the data item: ");  
      scanf("%d", &head-> num);  
      if(first != 0)  
      {  
         temp->ptr = head;temp = head;  
      }  
      else  
      {  
         first = temp = head;  
      }  
      fflush(stdin);  
      printf("Do you want to continue(Type 0 or 1)?\n\n");  
      scanf("%d", &choice);  
   }  
   temp->ptr = 0;  
   temp = first; /\* reset temp to the beginning\*/  
   printf("\n Status of the linked list is\n");  
   while(temp!=0)  
   {  
      printf("%d=>", temp->num);  
      count++;  
      temp = temp -> ptr;  
   }  
   printf("NULL\n");  
   printf("No. of nodes in the list = %d\n", count);  
   return 0;  
}

**Output:**  
  


Linked List Operations

**A node is declare as below:**

Typedef struct node  
{  
   int data;  
   struct node \*next;  
}node;  
node\* head = NULL;

The above code identifies the basic structure of declaring a node. Struct node contains an int data field and a pointer to another node can be defined as struct node\* next.  
  
**Following are the operations that can be performed on a Linked List:**  
  
1. Create  
2. Insert  
3. Delete  
4. Traverse  
5. Search  
6. Concatenation  
7. Display

**1. Create**

* Create operation is used to create constituent node when required.
* In create operation, memory must be allocated for one node and assigned to head as follows.

**Creating first node**

head = (node\*) malloc (sizeof(node));  
head -> data = 20;  
head -> next = NULL;

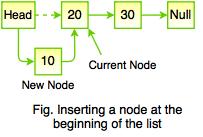
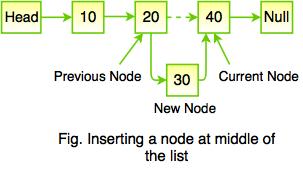
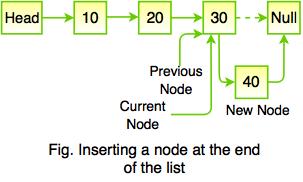
create linked list

**2. Insert**

* Insert operation is used to insert a new node in the linked list.
* Suppose, we insert a node B(New Node), between A(Left Node) and C(Right Node), it is represented as:

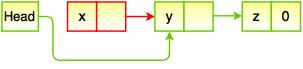
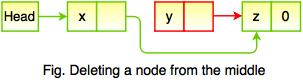
point B.next to B  
  
NewNode.next -> RightNode;  
  
**We can insert an element using three cases:**  
  
i. At the beginning of the list  
ii. At a certain position (Middle)  
iii. At the end  
  
**Inserting an element**

node\* nextnode = malloc(sizeof(node));  
nextnode -> data = 22;  
nextnode -> next = NULL;  
head -> next = nextnode;

insert linked list  
  
The above figure represents the example of create operation, where the next element (i.e 22) is added to the next node by using insert operation.  
  
**i. At the beginning of the list**  
  
  
  
New node becomes the new head of the linked list because it is always added before the head of the given linked list.  
  
**ii. At certain position (Middle)**  
  
  
  
While inserting a node in middle of a linked list, it requires to find the current node. The dashed line represents the old node which points to new node.  
  
**iii. At the end**  
  
  
  
While inserting a node at the end of the list, it is achieved by comparing the element values.

**3. Delete**

* Delete operation is used to delete node from the list.
* This operation is more than one step process.

**We can delete an element using three cases:**  
  
i. From the beginning of the list  
ii. From the middle  
iii. From the end  
  
**Deleting the element**  
  
int delete (node\*\* head, node\* n);    // Delete the node n if exists.  
  
**i. From the beginning of the list**  
  
delete beginning linked list  
  
When deleting the node from the beginning of the list then there is no relinking of nodes to be performed; it means that the first node has no preceding node. The above figure shows the removing node with x. However, it requires to fix the pointer to the beginning of the list which is shown in the figure below:  
  
  
  
**ii. From the middle**  
  
  
Deleting a node from the middle requires the preceding node to skip over the node being removed.  
  
The above figure shows the removal of node with x. It means that there is a need refer to the node before we can remove it.  
  
**iii. From the end**  
  
delete end linked list  
  
Deleting a node from the end requires that the preceding node becomes the new end of the list that points to nothing after it. The above figure shows removing the node with z.

**4. Traverse**

* Traverse operations is a process of examining all the nodes of linked list from the end to the other end.
* In traverse operation, recursive function is used to traverse a linked list in a reverse order.

**The following code snippet represents traversing a node in a linked list:**

void traverse(node \*head)  
{  
  if(head != NULL)  
  {  
    traverse (head -> next);  
    printf(“%d”, head -> data);  
  }  
}

**5. Search**

* Search operation is used for finding a particular element in a linked list.
* Sequential search is the most common search used on linked list structure.
* Search operation ends with a success if the element is found.
* If the element is not found, search ends in a failure.

**6. Concatenation**

Concatenation is the process of appending a second list to the end of the first list.

**7. Display**

* Display operation is used to print each and every node's information.
* This operation displays the complete list.

# Linked list Data Structure

In this tutorial, you will learn about linked list data structure and it's implementation in Python, Java, C, and C++.

A linked list is a linear data structure that includes a series of connected nodes. Here, each node stores the **data** and the **address** of the next node. For example,

Linked list Data Structure

You have to start somewhere, so we give the address of the first node a special name called HEAD. Also, the last node in the linked list can be identified because its next portion points to NULL.

Linked lists can be of multiple types: **singly**, **doubly**, and **circular linked list**. In this article, we will focus on the **singly linked list**. To learn about other types, visit [Types of Linked List](https://www.programiz.com/dsa/linked-list-types).

**Note:** You might have played the game Treasure Hunt, where each clue includes the information about the next clue. That is how the linked list operates.

## Representation of Linked List

Let's see how each node of the linked list is represented. Each node consists:

* A data item
* An address of another node

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

struct node

{

int data;

struct node \*next;

};

Understanding the structure of a linked list node is the key to having a grasp on it.

Each struct node has a data item and a pointer to another struct node. Let us create a simple Linked List with three items to understand how this works.

/\* Initialize nodes \*/

struct node \*head;

struct node \*one = NULL;

struct node \*two = NULL;

struct node \*three = NULL;

/\* Allocate memory \*/

one = malloc(sizeof(struct node));

two = malloc(sizeof(struct node));

three = malloc(sizeof(struct node));

/\* Assign data values \*/

one->data = 1;

two->data = 2;

three->data=3;

/\* Connect nodes \*/

one->next = two;

two->next = three;

three->next = NULL;

/\* Save address of first node in head \*/

head = one;

If you didn't understand any of the lines above, all you need is a refresher on [pointers and structs](https://www.programiz.com/c-programming/c-structures-pointers).

In just a few steps, we have created a simple linked list with three nodes.

Linked list Representation

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

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

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

In python and Java, the linked list can be implemented using classes as shown in [the codes below](https://www.programiz.com/dsa/linked-list#code).

## Linked List Utility

Lists are one of the most popular and efficient data structures, with implementation in every programming language like C, C++, Python, Java, and C#.

Apart from that, linked lists are a great way to learn how pointers work. By practicing how to manipulate linked lists, you can prepare yourself to learn more advanced data structures like graphs and trees.

## Linked List Implementations in Python, Java, C, and C++ Examples

[Python](https://www.programiz.com/dsa/linked-list#python-code)

[Java](https://www.programiz.com/dsa/linked-list#java-code)

[C](https://www.programiz.com/dsa/linked-list#c-code)

[C++](https://www.programiz.com/dsa/linked-list#cpp-code)

// Linked list implementation in C

#include <stdio.h>

#include <stdlib.h>

// Creating a node

struct node {

int value;

struct node \*next;

};

// print the linked list value

void printLinkedlist(struct node \*p) {

while (p != NULL) {

printf("%d ", p->value);

p = p->next;

}

}

int main() {

// Initialize nodes

struct node \*head;

struct node \*one = NULL;

struct node \*two = NULL;

struct node \*three = NULL;

// Allocate memory

one = malloc(sizeof(struct node));

two = malloc(sizeof(struct node));

three = malloc(sizeof(struct node));

// Assign value values

one->value = 1;

two->value = 2;

three->value = 3;

// Connect nodes

one->next = two;

two->next = three;

three->next = NULL;

// printing node-value

head = one;

printLinkedlist(head);

}

## Linked List Complexity

Time Complexity

|  |  |  |
| --- | --- | --- |
|  | Worst case | Average Case |
| **Search** | O(n) | O(n) |
| **Insert** | O(1) | O(1) |
| **Deletion** | O(1) | O(1) |

Space Complexity: O(n)

## Linked List Applications

* Dynamic memory allocation
* Implemented in stack and queue
* In **undo** functionality of softwares
* Hash tables, Graphs