KAKINADA INSTITUE OF ENGINEERING AND TECHNOLOGY

MCA SEMESTER –I

DATA STRUCTURES.

INDEX

DATA STRUCTURES

UNIT I:

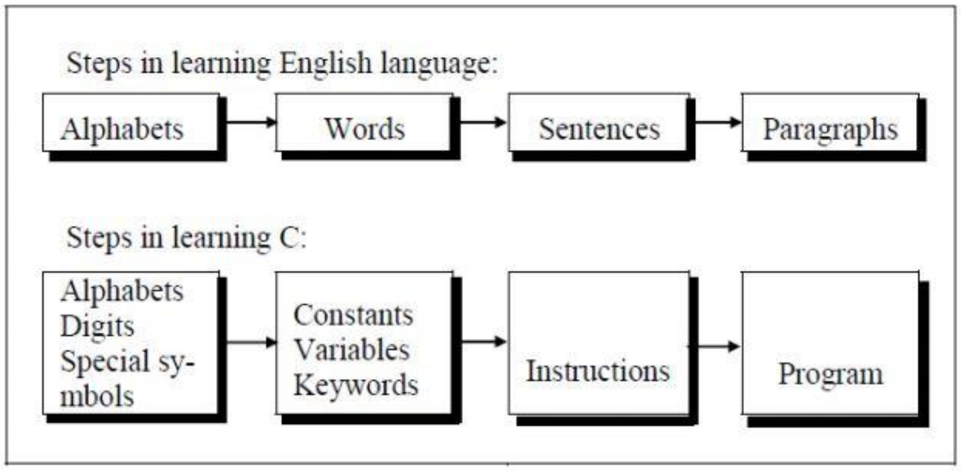
Introduction to C: Constants and variables, Operators and Expressions, Managing Input and Output operators, Decision making-branching and looping, Arrays.

Introduction to C:

C is a programming language developed at AT & T's Bell Laboratories of USA in 1972. It was designed and written by a man named Dennis Ritchie. In the late seventies C began to replace the more familiar languages of that time like PL/I, ALGOL, etc .

ANSI C standard emerged in the early 1980s, this book was split into two titles: The original was still called Programming in C, and the title that covered ANSI C was called Programming in ANSI C. This was done because it took several years for the compiler vendors to release their ANSI C compilers and for them to become ubiquitous. It was initially designed for programming UNIX operating system. Now the software tool as well as the C compiler is written in C. Major parts of popular operating systems like Windows, UNIX, Linux is still written in C. This is because even today when it comes to performance (speed of execution) nothing beats C. Moreover, if one is to extend the operating system to work with new devices one needs to write device driver programs. These programs are exclusively written in C. C seems so popular is because it is reliable, simple and easy to use. often heard today is – "C has been already superceded by languages like C++, C# and Java.

A computer program is just a collection of the instructions necessary to solve a specific problem. The basic operations of a computer system form what is known as the computer's instruction set. And the approach or method that is used to solve the problem is known as an algorithm.


Steps in learning English language:
Alphabets
Words
Sentences
Paragraphs
Steps in learning C:
Alphabets Digits Special sy-mbols
Constants Variables Keywords
Instructions
Program

So for as programming language concern these are of two types.

1) Low level language 2) High level language

Low level language:

Low level languages are machine level and assembly level language. In machine level language computer only understand digital numbers i.e. in the form of 0 and 1. So, instruction given to the computer is in the form binary digit, which is difficult to implement instruction in binary code. This type of program is not portable, difficult to maintain and also error prone. The assembly language is on other hand modified version of machine level language. Where instructions are given in English like word as ADD, SUM, MOV etc. It is easy to write and understand but not understand by the machine. So the translator used here is assembler to translate into machine level. Although language is bit easier, programmer has to know low level details related to low level language. In the assembly level language the data are stored in the computer register, which varies for different computer. Hence it is not portable.

High level language:

These languages are machine independent, means it is portable. The language in this category is Pascal, Cobol, Fortran etc. High level languages are understood by the machine. So it need to translate by the translator into machine level. A translator is software which is used to translate high level language as well as low level language in to machine level language.

Sample 'C' Program:

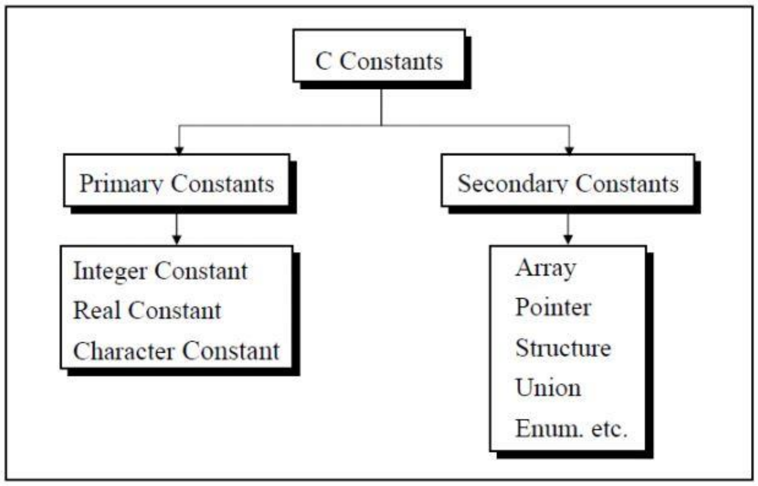
```
/*First c program with return statement*/

#include<stdio.h>

int main (void)

 {

printf ("welcome to c Programming language.\n");

return 0;

}
```

Output: welcome to c programming language.

Constants:

Constant is a any value that cannot be changed during program execution. In C, any number, single character, or character string is known as a constant. A constant is an entity that doesn't change whereas a variable is an entity that may change. For example, the number 50 represents a constant integer value. The character string "Programming in C is fun.\n" is an example of a constant character string.

C constants can be divided into two major categories: Primary Constants & Secondary Constants



Numeric constant:

Numeric constant consists of digits. It required minimum size of 2 bytes and max 4 bytes. It may be positive or negative but by default sign is always positive. No comma or space is allowed within the numeric constant and it must have at least 1 digit. The allowable range for integer constants is -32768 to 32767. Truly speaking the range of an Integer constant depends upon the compiler. For a 16-bit compiler like Turbo C or Turbo C++ the range is –32768 to 32767. For a 32-bit compiler the range would be even greater. Mean by a 16-bit or a 32-bit compiler, what range of an Integer constant has to do with the type of compiler.

It is categorized a **integer constant** and **real constant**. An integer constants are whole number which have no decimal point.

Types of integer constants are:
Decimal constant: 0-------9(base 10)
Octal constant: 0-------7(base 8)
Hexa decimal constant: 0----9, A------F(base 16)

Real constant is also called floating point constant. To construct real constant we must follow the rule of , -real constant must have at least one digit. -It must have a decimal point. -It could be either positive or negative. -Default sign is positive. -No commas or blanks are allowed within a real constant. Ex.: +325.34, 426.0, -32.76.

Character constant:
Character constant represented as a single character enclosed within a single quote. These can be single digit, single special symbol or white spaces such as ‘9’,’c’,’$’, ‘ ’ etc. Every character constant has a unique integer like value in machine’s character code as if machine using ASCII (American standard code for information interchange). Some numeric value associated with each upper and lower case alphabets and decimal integers are as:
A------------ Z ASCII value (65-90)
a-------------z ASCII value (97-122)
0-------------9 ASCII value (48-59)
; ASCII value (59)
String constant
Set of characters are called string and when sequence of characters are enclosed within a double quote (it may be combination of all kind of symbols) is a string constant. String constant has zero, one or more than one

character and at the end of the string null character(\0) is automatically placed by compiler. Some examples are ",sarathina" , "908", "3"," ", "A" etc. In C although same characters are enclosed within single and double quotes it represents different meaning such as "A" and 'A' are different because first one is string attached with null character at the end but second one is character constant with its corresponding ASCII value is 65.

<u>Variables in 'C' :</u>

Variable is a data name which is used to store some data value or symbolic names for storing program computations and results. The value of the variable can be change during the execution. The rule for naming the variables is same as the naming identifier. Before used in the program it must be declared. Declaration of variables specify its name, data types and range of the value that variables can store depends upon its data types.

<u>Rules for defining variables</u>

- A variable can have alphabets, digits, and underscore.
- A variable name can start with the alphabet, and underscore only. It can't start with a digit.
- No whitespace is allowed within the variable name.
- A variable name must not be any reserved word or keyword, e.g. int, float, etc.

There are many types of variables in c:

1. local variable
2. global variable
3. static variable
4. automatic variable
5. external variable

<u>Local Variable</u>

A variable that is declared inside the function or block is called a local variable.
It must be declared at the start of the block.

```
void function1(){
int x=10;//local variable
}
```

You must have to initialize the local variable before it is used.

<u>Global Variable</u>

A variable that is declared outside the function or block is called a global variable. Any function can change the value of the global variable. It is available to all the functions.
It must be declared at the start of the block.

1. **int** value=20;//global variable
2. **void** function1(){
3. **int** x=10;//local variable
4. }

Static Variable

A variable that is declared with the static keyword is called static variable.
It retains its value between multiple function calls.

1. **void** function1(){
2. **int** x=10;//local variable
3. **static int** y=10;//static variable
4. x=x+1;
5. y=y+1;
6. printf("%d,%d",x,y);
7. }

If you call this function many times, the **local variable will print the same value** for each function call, e.g, 11,11,11 and so on. But the **static variable will print the incremented value** in each function call, e.g. 11, 12, 13 and so on.

Automatic Variable

All variables in C that are declared inside the block, are automatic variables by default. We can explicitly declare an automatic variable using **auto keyword**.

1. **void** main(){
2. **int** x=10;//local variable (also automatic)
3. auto **int** y=20;//automatic variable
4. }

External Variable

We can share a variable in multiple C source files by using an external variable. To declare an external variable, you need to use **extern keyword**.

myfile.h

1. **extern int** x=10;//external variable (also global)

C Operators

An operator is simply a symbol that is used to perform operations. There can be many types of operations like arithmetic, logical, bitwise, etc.
There are following types of operators to perform different types of operations in C language.

- Arithmetic Operators
- Relational Operators
- Shift Operators
- Logical Operators

- Bitwise Operators
- Ternary or Conditional Operators
- Assignment Operator
- Misc Operator

Category	Operator	Associativity
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left
Comma	,	Left to right

<u>Expressions</u> :

An expression is a combination of variables, constants, operators and function call. It can be arithmetic, logical and relational .
for example:-
int z= x+y //
arithmatic expression a>b //
relational a==b // logical func(a, b) //
function call Expressions consisting entirely of constant values are called constant expressions.

So, the expression 121 + 17 - 110 is a constant expression because each of the terms of the expression is a constant value.
But if i were declared to be an integer variable, the expression 180 + 2 – j would not represent a constant expression.

The getchar() and putchar() Functions

The **int getchar(void)** function reads the next available character from the screen and returns it as an integer. This function reads only single character at a time. You can use this method in the loop in case you want to read more than one character from the screen.

The **int putchar(int c)** function puts the passed character on the screen and returns the same character. This function puts only single character at a time. You can use this method in the loop in case you want to display more than one character on the screen.

The scanf() and printf() Functions

The **int scanf(const char *format, ...)** function reads the input from the standard input stream **stdin** and scans that input according to the **format** provided.

The **int printf(const char *format, ...)** function writes the output to the standard output stream **stdout** and produces the output according to the format provided.

The **format** can be a simple constant string, but you can specify %s, %d, %c, %f, etc., to print or read strings, integer, character or float respectively. There are many other formatting options available which can be used based on requirements.

Decision making-branching and looping :

Decision making structures require that the programmer specifies one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

C programming language assumes any **non-zero** and **non-null** values as **true**, and if it is either **zero** or **null**, then it is assumed as **false** value.
C programming language provides the following types of decision making statements.

Sr.No.	Statement & Description
1	if statement An **if statement** consists of a boolean expression followed by one or more statements.

2	if...else statement An **if statement** can be followed by an optional **else statement**, which executes when the Boolean expression is false.
3	nested if statements You can use one **if** or **else if** statement inside another **if** or **else if** statement(s).
4	switch statement A **switch** statement allows a variable to be tested for equality against a list of values.
5	nested switch statements You can use one **switch** statement inside another **switch** statement(s).

You may encounter situations, when a block of code needs to be executed several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.
Programming languages provide various control structures that allow for more complicated execution paths.
A loop statement allows us to execute a statement or group of statements multiple times.
C programming language provides the following types of loops to handle looping requirements.

Sr.No.	Loop Type & Description
1	while loop Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
2	for loop Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
3	do...while loop It is more like a while statement, except that it tests the

	condition at the end of the loop body.
4	nested loops You can use one or more loops inside any other while, for, or do..while loop.

ARRAY:

Array is the collection of similar data types or collection of similar entity stored in contiguous memory location. Array of character is a string. Each data item of an array is called an element. And each element is unique and located in separated memory location. Each of elements of an array share a variable but each element having different index no. known as subscript.

An array can be a single dimensional or multi-dimensional and number of subscripts determines its dimension. And number of subscript is always starts with zero. One dimensional array is known as vector and two dimensional arrays are known as matrix.

ADVANTAGES: array variable can store more than one value at a time where other variable can store one value at a time.

Example: int arr[100];
int mark[100];

DECLARATION OF AN ARRAY :

Its syntax is :

Data type array name [size];
int arr[100];
int mark[100];
int a[5]={10,20,30,100,5}

The declaration of an array tells the compiler that, the data type, name of the array, size of the array and for each element it occupies memory space. Like for int data type, it occupies 2 bytes for each element and for float it occupies 4 byte for each element etc. The size of the array operates the number of elements that can be stored in an array and it may be a int constant or constant int expression.

INITIALIZATION OF AN ARRAY:

After declaration element of local array has garbage value. If it is global or static array then it will be automatically initialize with zero. An explicitly it can be initialize that Data type array name [size] = {value1, value2, value3…}

Example: in ar[5]={20,60,90, 100,120}

Arrays in Detail

Arrays are important to C and should need a lot more attention. The following important concepts related to array should be clear to a C programmer −

Sr.No.	Concept & Description
1	Multi-dimensional arrays C supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array.
2	Passing arrays to functions You can pass to the function a pointer to an array by specifying the array's name without an index.
3	Return array from a function C allows a function to return an array.
4	Pointer to an array You can generate a pointer to the first element of an array by simply specifying the array name, without any index.

UNIT II

Functions, Structures and Unions, Pointers, File handling in C.

Functions :

A function is a group of statements that together perform a task. Every C program has at least one function, which is **main()**, and all the most trivial programs can define additional functions.

A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function.

The C standard library provides numerous built-in functions that your program can call. For example, **strcat()** to concatenate two strings, **memcpy()** to copy one memory location to another location, and many more functions.

A function can also be referred as a method or a sub-routine or a procedure, etc.

Defining a Function:

The general form of a function definition in C programming language is as follows −

```
return_type function_name( parameter list ) {
body of the function
   }
```

A function definition in C programming consists of a *function header* and a *function body*. Here are all the parts of a function −

- **Return Type** − A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.
- **Function Name** − This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- **Parameters** − A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
- **Function Body** − The function body contains a collection of statements that define what the function does.

Function Declarations:

- A function **declaration** tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.
- A function declaration has the following parts −
- return_type function_name(parameter list);
- For the above defined function max(), the function declaration is as follows −
- int max(int num1, int num2);
- Parameter names are not important in function declaration only their type is required, so the following is also a valid declaration −
- int max(int, int);
- Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

Calling a Function:

While creating a C function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task.

When a program calls a function, the program control is transferred to the called function. A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the main program.

To call a function, you simply need to pass the required parameters along with the function name, and if the function returns a value, then you can store the returned value.

Function Arguments:

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function.

Formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways in which arguments can be passed to a function –

Sr.No.	Call Type & Description
1	Call by value This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
2	Call by reference This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

By default, C uses **call by value** to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function.

Structure:

Arrays allow to define type of variables that can hold several data items of the same kind. Similarly **structure** is another user defined data type available in C that allows to combine data items of different kinds.

Structures are used to represent a record. Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book −

- Title
- Author
- Subject
- Book ID

Defining a Structure:

To define a structure, you must use the struct statement. The struct statement defines a new data type, with more than one member. The format of the struct statement is as follows −

struct [structure tag] {

```
   member definition;
   member definition;
   ...
   member definition;
} [one or more structure variables];
```

The structure tag is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is optional. Here is the way you would declare the Book structure −

```
struct Books {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
} book;
```

Union:

A union is a special data type available in C that allows to store different data types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multiple-purpose.

Defining a Union

To define a union, you must use the union statement in the same way as you did while defining a structure. The union statement defines a new data type with more than one member for your program. The format of the union statement is as follows −

```
union [union tag] {
   member definition;
   member definition;
   ...
   member definition;
} [one or more union variables];
```

The union tag is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the union's definition, before the final semicolon, you can specify one or more union variables but it is optional. Here is the way you would define a union type named Data having three members i, f, and str −

```
union Data {
   int i;
   float f;
   char str[20];
} data;
```

Now, a variable of Data type can store an integer, a floating-point number, or a string of characters. It means a single variable, i.e., same memory location, can be used to store multiple types of data. You can use any built-in or user defined data types inside a union based on your requirement.

Let's see a simple example of union in C language.

```
#include <stdio.h>
#include <string.h>
union employee
{   int id;
    char name[50];
}e1;  //declaring e1 variable for union
int main( )
{
   //store first employee information
   e1.id=101;
   strcpy(e1.name, "Sonoo Jaiswal");//copying string into char array
   //printing first employee information
printf( "employee 1 id : %d\n", e1.id);
printf( "employee 1 name : %s\n", e1.name);
return 0;
}
```

Output:

```
employee 1 id : 1869508435
employee 1 name : Sonoo Jaiswal
```

Pointers:

Pointers in C are easy and fun to learn. Some C programming tasks are performed more easily with pointers, and other tasks, such as dynamic memory allocation, cannot be performed without using pointers. So it becomes necessary to learn pointers to become a perfect C programmer.

As you know, every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator, which denotes an address in memory.

Consider the following example, which prints the address of the variables defined −

```
#include <stdio.h>

int main () {

   int  var1;
   char var2[10];

   printf("Address of var1 variable: %x\n", &var1  );
   printf("Address of var2 variable: %x\n", &var2  );

   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Address of var1 variable: bff5a400
Address of var2 variable: bff5a3f6
```

What are Pointers?

A **pointer** is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before using it to store any variable address. The general form of a pointer variable declaration is −

```
type *var-name;
```

Here, **type** is the pointer's base type; it must be a valid C data type and **var-name** is the name of the pointer variable. The asterisk * used to declare a pointer is the same asterisk used for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Take a look at some of the valid pointer declarations −

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *ch     /* pointer to a character */
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

How to Use Pointers?

There are a few important operations, which we will do with the help of pointers very frequently. **(a)** We define a pointer variable, **(b)** assign the address of a variable to a pointer and **(c)** finally access the value at the address available in the pointer variable. This is done by using unary operator ***** that returns the value of the variable located at the address specified by its operand. The following example makes use of these operations −

```
#include <stdio.h>


int main () {
   int  var = 20;   /* actual variable declaration */
   int  *ip;        /* pointer variable declaration */

   ip = &var;  /* store address of var in pointer variable*/

   printf("Address of var variable: %x\n", &var  );

   /* address stored in pointer variable */
   printf("Address stored in ip variable: %x\n", ip );

   /* access the value using the pointer */
   printf("Value of *ip variable: %d\n", *ip );

   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Address of var variable: bffd8b3c
Address stored in ip variable: bffd8b3c
Value of *ip variable: 20
```

NULL Pointers

It is always a good practice to assign a NULL value to a pointer variable in case you do not have an exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a **null** pointer.

The NULL pointer is a constant with a value of zero defined in several standard libraries. Consider the following program −

```
#include <stdio.h>
```

```
int main () {

   int  *ptr = NULL;

   printf("The value of ptr is : %x\n", ptr  );

   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
The value of ptr is 0
```

In most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system. However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the null (zero) value, it is assumed to point to nothing.

To check for a null pointer, you can use an 'if' statement as follows −

```
if(ptr)     /* succeeds if p is not null */
if(!ptr)    /* succeeds if p is null */
```

Pointers in Detail

Pointers have many but easy concepts and they are very important to C programming. The following important pointer concepts should be clear to any C programmer −

Sr.No.	Concept & Description
1	Pointer arithmetic There are four arithmetic operators that can be used in pointers: ++, --, +, -

2	Array of pointers You can define arrays to hold a number of pointers.
3	Pointer to pointer C allows you to have pointer on a pointer and so on.
4	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.
5	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.

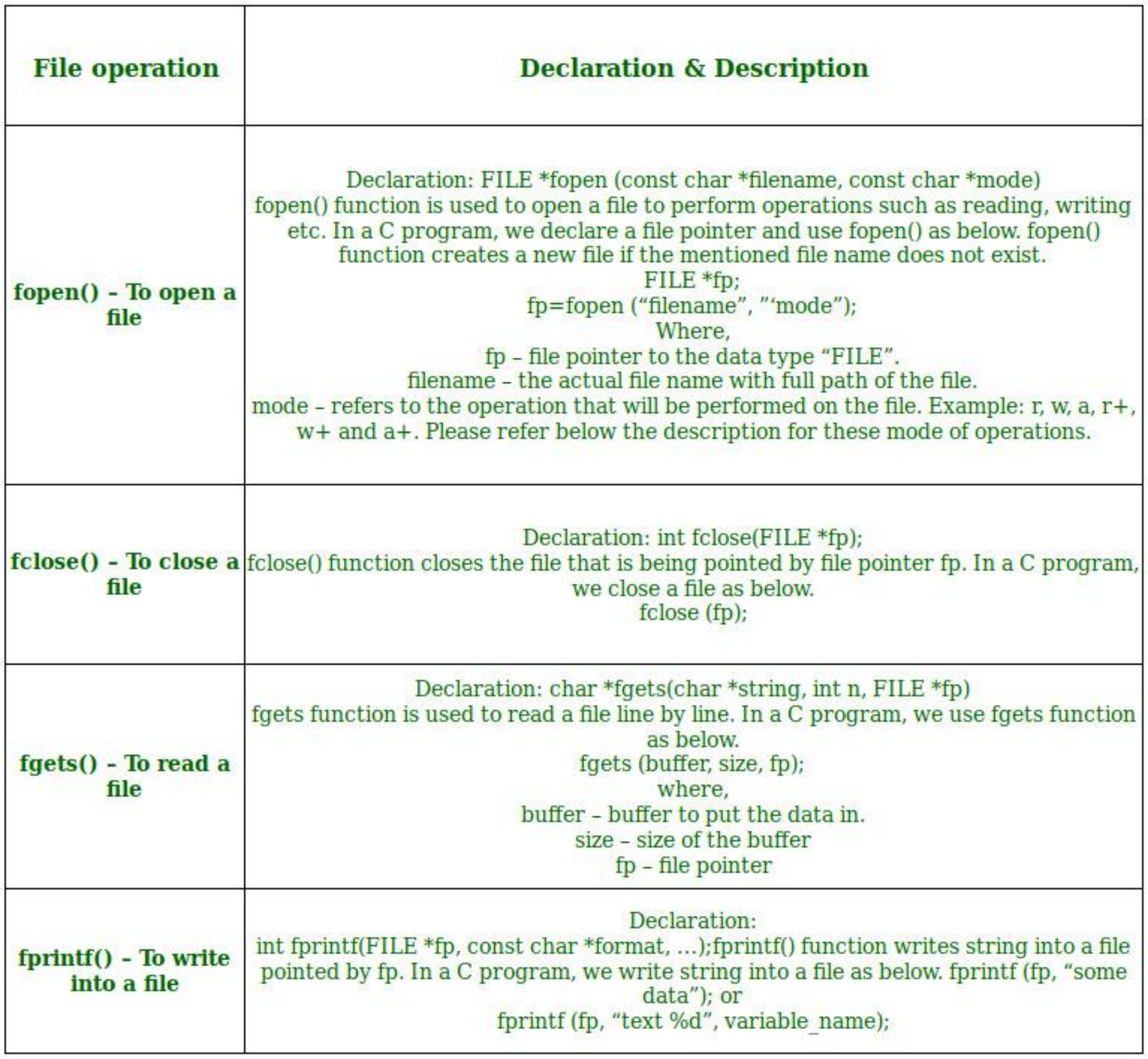
File Handling in C

In programming, we may require some specific input data to be generated several numbers of times. Sometimes, it is not enough to only display the data on the console. The data to be displayed may be very large, and only a limited amount of data can be displayed on the console, and since the memory is volatile, it is impossible to recover the programmatically generated data again and again. However, if we need to do so, we may store it onto the local file system which is volatile and can be accessed every time. Here, comes the need of file handling in C.

File handling in C enables us to create, update, read, and delete the files stored on the local file system through our C program. The following operations can be performed on a file.

- Creation of the new file
- Opening an existing file
- Reading from the file
- Writing to the file

- Deleting the file



File operation	Declaration & Description
fopen() – To open a file	Declaration: FILE *fopen (const char *filename, const char *mode) fopen() function is used to open a file to perform operations such as reading, writing etc. In a C program, we declare a file pointer and use fopen() as below. fopen() function creates a new file if the mentioned file name does not exist. FILE *fp; fp=fopen (“filename”, ”‘mode”); Where, fp – file pointer to the data type “FILE”. filename – the actual file name with full path of the file. mode – refers to the operation that will be performed on the file. Example: r, w, a, r+, w+ and a+. Please refer below the description for these mode of operations.

fclose() – To close a file	Declaration: int fclose(FILE *fp);
	fclose() function closes the file that is being pointed by file pointer fp. In a C program, we close a file as below.
	fclose (fp);

fgets() – To read a file	Declaration: char *fgets(char *string, int n, FILE *fp) fgets function is used to read a file line by line. In a C program, we use fgets function as below. fgets (buffer, size, fp); where, buffer – buffer to put the data in. size – size of the buffer fp – file pointer
fprintf() – To write into a file	Declaration: int fprintf(FILE *fp, const char *format, …);fprintf() function writes string into a file pointed by fp. In a C program, we write string into a file as below. fprintf (fp, “some data”); or fprintf (fp, “text %d”, variable_name);

UNIT – 3

Data structure: Definition, types of data structures Recursion Definition, Design Methodology and Implementation of recursive algorithms, Linear and binary recursion. Preliminaries of algorithms, analysis and complexity .
Linear list – singly linked list, Double linked list and circular linked list - implementation, insertion, deletion and searching operations on linear list.

Introduction to Data structures

In computer terms, a data structure is a Specific way to store and organize data in a computer's memory so that these data can be used efficiently later. Data may be arranged in many different ways such as the logical or mathematical model for a particular organization of data is termed as a data structure. The variety of a particular data model depends on the two factors -
 Firstly, it must be loaded enough in structure to reflect the actual relationships of the data with the real world object.
 Secondly, the formation should be simple enough so that anyone can efficiently process the data each time it is necessary.

Categories of Data Structure:

The data structure can be sub divided into major types:
 Linear Data Structure
 Non-linear Data Structure

Linear Data Structure:

A data structure is said to be linear if its elements combine to form any specific order. There are basically two techniques of representing such linear structure within memory.
 First way is to provide the linear relationships among all the elements represented by means of linear memory location. These linear structures are termed as arrays.
 The second technique is to provide the linear relationship among all the elements represented by using the concept of pointers or links. These linear structures are termed as linked lists.

The common examples of linear data structure are:
 Arrays

 Queues
 Stacks
 Linked lists

Non linear Data Structure:
This structure is mostly used for representing data that contains a hierarchical relationship among various elements.
Examples of Non Linear Data Structures are listed below:
 Graphs
 family of trees and
 table of contents

Tree: In this case, data often contain a hierarchical relationship among various elements. The data structure that reflects this relationship is termed as rooted tree graph or a tree.
Graph: In this case, data sometimes hold a relationship between the pairs of elements which is not necessarily following the hierarchical structure. Such data structure is termed as a Graph.
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

Recursion:
Recursion is the process of repeating items in a self-similar way. In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function.

```
void recursion() {
  recursion(); /* function calls itself */
}

int main() {
  recursion();
}
```

The C programming language supports recursion, i.e., a function to call itself. But while using recursion, programmers need to be careful to define an exit condition from the function, otherwise it will go into an infinite loop.

Recursive functions are very useful to solve many mathematical problems, such as calculating the factorial of a number, generating Fibonacci series, etc.

Properties

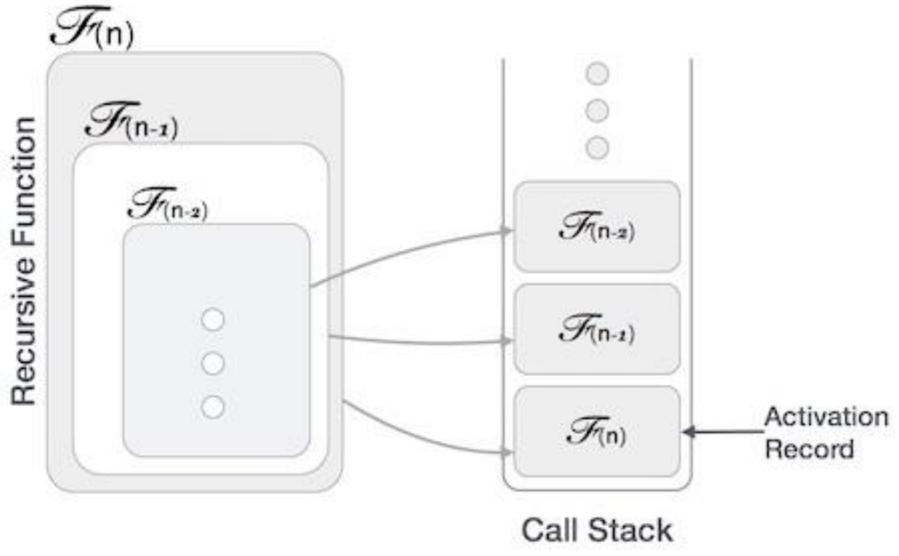
A recursive function can go infinite like a loop. To avoid infinite running of recursive function, there are two properties that a recursive function must have −

- **Base criteria** − There must be at least one base criteria or condition, such that, when this condition is met the function stops calling itself recursively.
- **Progressive approach** − The recursive calls should progress in such a way that each time a recursive call is made it comes closer to the base criteria.

Implementation

Many programming languages implement recursion by means of **stacks**. Generally, whenever a function (**caller**) calls another function (**callee**) or itself as callee, the caller function transfers execution control to the callee. This transfer process may also involve some data to be passed from the caller to the callee.

This implies, the caller function has to suspend its execution temporarily and resume later when the execution control returns from the callee function. Here, the caller function needs to start exactly from the point of execution where it puts itself on hold. It also needs the exact same data values it was working on. For this purpose, an activation record (or stack frame) is created for the caller function.



This activation record keeps the information about local variables, formal parameters, return address and all information passed to the caller function.

Analysis of Recursion

One may argue why to use recursion, as the same task can be done with iteration. The first reason is, recursion makes a program more readable and because of latest enhanced CPU systems, recursion is more efficient than iterations.

Time Complexity

In case of iterations, we take number of iterations to count the time complexity. Likewise, in case of recursion, assuming everything is constant, we try to figure out the number of times a recursive call is being made. A call made to a function is O(1), hence the (n) number of times a recursive call is made makes the recursive function O(n).

Space Complexity

Space complexity is counted as what amount of extra space is required for a module to execute. In case of iterations, the compiler hardly requires any extra space. The compiler keeps updating the values of variables used in the iterations. But in case of recursion, the system needs to store activation record each time a recursive call is made. Hence, it is considered that space complexity of recursive function may go higher than that of a function with iteration.

Recursion are mainly of two types depending on weather a function calls itself from within itself weather two function call one another mutually. The former is called direct recursion and t latter is called indirect recursion. **Thus, the two types of recursion are**:

1. **Direct recursion**
2. **Indirect recursion**

Both types of recursion are shown diagrammatically below:
Now before we proceed into the core programming with recursion, first of all we will see a brief idea of storage classes, after which we study some necessary conditions for the recursion to be implemented correctly.

Recursion may be further categorized as:

1. **Linear recursion**
2. **Binary recursion**
3. **Multiple recursion**

1. Linear recursion

In linear recursion the algorithm begins by testing set of base cases there should be at least one. Every possible chain of recursive calls must eventually reach base case, and the handling of each base case should not use recursion. **In linear recursion we follow:**

i. Perform a single recursive call. In this process the recursive step involves a test which decide out of all the several possible recursive calls which one is make, but it should ultimately choose to make just one of these calls each time we perform this step.

ii. Define each possible recursive call, so that it makes progress towards a base case.

A simple example of linear recursion.

Input

An integer array A and an integer n=1, such that A has at least n elements.

Output

The sum of first n integer in A

```
If n=1 then
return A[0]
else
return LinearSum (A, n-1) + A[n-1]
```

In creating recursive methods, it is important to define the methods in a way that facilitate recursion. This sometimes requires we define additional parameters that are passed to the method. For example, we define the array reversal method as **ReverseArray (A, i, j)**, not **ReverseArray (A)**.

Algorithm

ReverseArray (A, i, j);

Input

An array A and non-negative integer indices i and j.

Output

The revesal of the elements in A starting at index I and ending at index j.

```
If i<j then
Swap A[i] and A[j]
ReverseArray (A, i+1, j-1)
return
```

2. Binary recursion

Binary recursion occurs whenever there are two recursive calls for each non base case. Example is the problem to add all the numbers in an integer array **A**.

Algorithm

BinarySum (A, i, n)

Input

An array A and integers i and n.

Output

The sum of the integers in A starting from index i,

```
If n=1 then
return A[i]
```

```
else
return BinarySum [A, i, n/2] + BinarySum [A, i+n/2, n/2]
```

Example trace

Another example of binary recursion is computing Fibonacci numbers, Fibonacci numbers are defined recursively:

$F_0 = 0$
$F_1 = 0$
$F_i = Fi-1 + Fi-2$ for $i>1$

We represent this recursive algorithm as

Algorithm

BinaryFib (K)

Input

Non negative integer K

Output

The Kth Fibonacci number F$_k$

```
If K=1
Then return K
Else
return BinaryFib (K-1) + BinaryFib (K-2)
```

Analyzing the binary recursion Fibonacci algorithm:

Let n_k denote number of recursive calls made by BinaryFib (K), then

$n_0 = 1$
$n_1 = 1$
$n_2 = n1 + n0 + 1 = 1 + 1 + 1 = 3$
$n_3 = n1 + n2 + 1 = 3 + 1 + 1 = 5$
$n_4 = n3 + n2 + 1 = 5 + 3 + 1 = 9$
$n_5 = n4 + n3 + 1 = 9 + 5 + 1 = 15$
$n_6 = n5 + n4 + 1 = 15 + 9 + 1 = 25$
$n_7 = n6 + n5 + 1 = 25 + 15 + 1 = 41$
$n_8 = n7 + n6 + 1 = 41 + 25 + 1 = 67$

Note that the value at least doubles for every other value of n_k. That is $nk > 2^{k/2}$. It is exponential.
We can write the better Fibonacci algorithm which can run in O(k) time using linear recursion rather than binary recursion.

3. Multiple Recursion

In multiple recursion we make not just one or two calls but many recursive calls. One of the motivating examples is summation puzzles.

Pot + pan = bib
Dog + cat = pig
Boy + girl = baby

To solve such a puzzle, we need to assign a unique digit (that is 0, 1, 2,…..9) too each letter in the equation, in order to make the equation true. Typically, we solve such a puzzle by using out human observation of the particular we are trying to solve to eliminate configurations (that is, possible partial assignment of digit to letters) until we can work through the feasible configurations left, testing for the correctness of each one.

Algorithm

PuzzleSolve (K, S, U)

Input

An integer K, sequence S and set U (the universe of element to test)

Output

An enumeration of all K length extensions to S using elements in U without repetitions for all e in U.

```
Remove e from U {e is now being used}
Add e to the end of S
If k=1 then
Test weather S is a configuration that solve puzzle
If S solves puzzle then
Return “solution found;” S
Else
PuzzleSolve (K-1, S, U)
Add e back to U {e is now unused}
Remove e from the end of S.
```

Algorithm Analysis

Analysis of efficiency of an algorithm can be performed at two different stages, before implementation and after implementation, as

A priori analysis − This is defined as theoretical analysis of an algorithm.

Efficiency of algorithm is measured by assuming that all other factors e.g. speed of processor, are constant and have no effect on implementation.

A posterior analysis − This is defined as empirical analysis of an algorithm. The chosen algorithm is implemented using programming language. Next the chosen

algorithm is executed on target computer machine. In this analysis, actual statistics like running time and space needed are collected.
Algorithm analysis is dealt with the execution or running time of various operations involved. Running time of an operation can be defined as number of computer instructions executed per operation.

Algorithm Complexity

Suppose X is treated as an algorithm and N is treated as the size of input data, the time and space implemented by the Algorithm X are the two main factors which determine the efficiency of X.
Time Factor − The time is calculated or measured by counting the number of key operations such as comparisons in sorting algorithm.
Space Factor − The space is calculated or measured by counting the maximum memory space required by the algorithm.
The complexity of an algorithm f(N) provides the running time and / or storage space needed by the algorithm with respect of N as the size of input data.
Space Complexity
Space complexity of an algorithm represents the amount of memory space needed the algorithm in its life cycle.
Space needed 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 (i.e. simple variables and constants, program size etc.), that are not dependent of the size of the problem.
A variable part is a space required by variables, whose size is totally dependent on the size of the problem. For example, recursion stack space, dynamic memory allocation etc.
Space complexity S(p) of any algorithm p is S(p) = A + Sp(I) Where A is treated as the fixed part and S(I) is treated as 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(P, Q)
Step 1 - START
Step 2 - R ← P + Q + 10
Step 3 - Stop
```

Here we have three variables P, Q and R and one constant. Hence S(p) = 1+3. Now space is dependent on data types of given constant types and variables and it will be multiplied accordingly.

Time Complexity

Time Complexity of an algorithm is the representation of the amount of time required by the algorithm to execute to completion. Time requirements can be denoted or defined as a numerical function t(N), where t(N) can be measured as the number of steps, provided each step takes constant time. For example, in case of addition of two n-bit integers, N steps are taken. Consequently, the total computational time is t(N) = c*n, where c is the time consumed for addition of two bits. Here, we observe that t(N) grows linearly as input size increases.

LINKED LIST :

A linked list is a sequence of data structures, which are connected together via links. Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array.

Following are the important terms to understand the concept of Linked List.

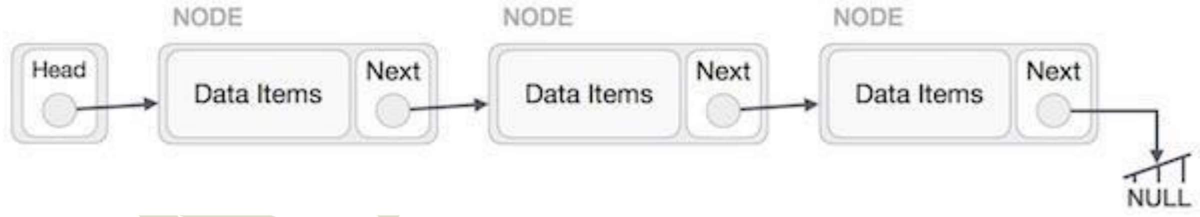
 Link − Each link of a linked list can store a data called an element.

 Next − Each link of a linked list contains a link to the next link called Next.

 LinkedList − A Linked List contains the connection link to the first link called First.

Linked List Representation

Linked list can be visualized as a chain of nodes, where every node points to the next node.



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

 Linked List contains a link element called first.

 Each link carries a data field(s) and a link field called next.

 Each link is linked with its next link using its next link.

 Last link carries a link as null to mark the end of the list.

Types of Linked List

Following are the various types of linked list.

 Simple Linked List − Item navigation is forward only.

 Doubly Linked List − Items can be navigated forward and backward.

 Circular Linked List − Last item contains link of the first element as next and the first element has a link to the last element as previous.

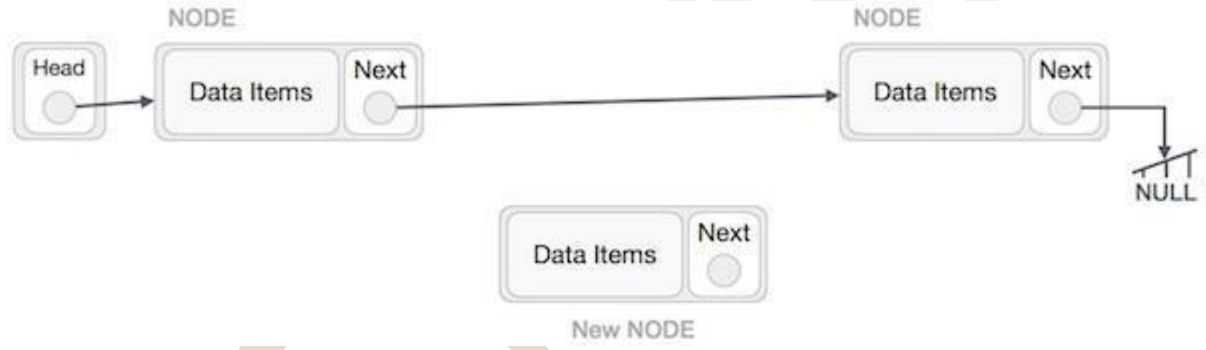
Basic Operations

Following are the basic operations supported by a list.

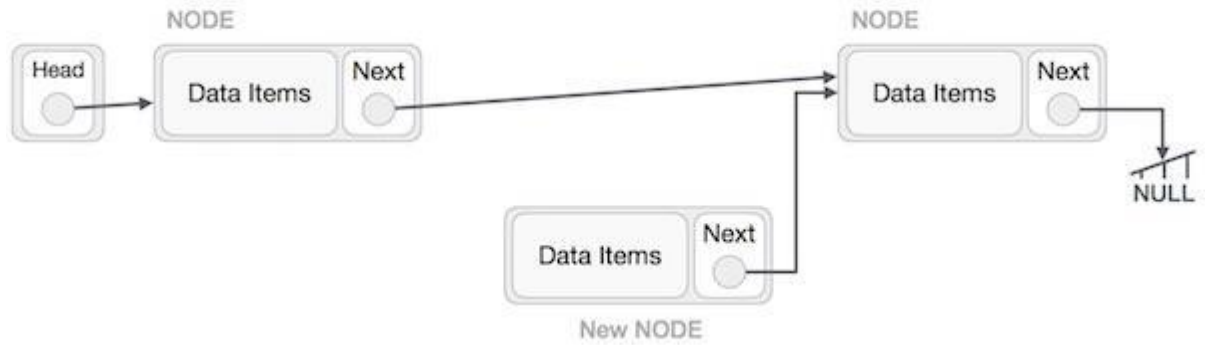
 Insertion − Adds an element at the beginning of the list.
 Deletion − Deletes an element at the beginning of the list.
 Display − Displays the complete list.
 Search − Searches an element using the given key.
 Delete − Deletes an element using the given key.

Insertion Operation

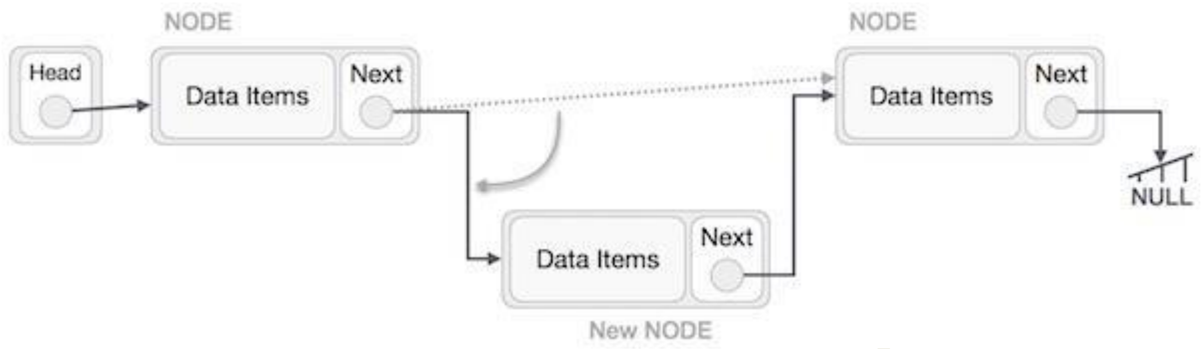
Adding a new node in linked list is a more than one step activity. We shall learn this with diagrams here. First, create a node using the same structure and find the location where it has to be inserted.



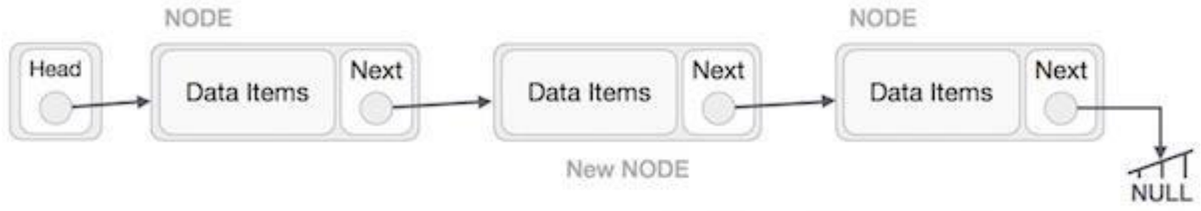
Imagine that we are inserting a node **B** (NewNode), between **A** (LeftNode) and **C** (RightNode). Then point B.next to C −
NewNode.next −> RightNode;
It should look like this –



Now, the next node at the left should point to the new node.
LeftNode.next −> NewNode;



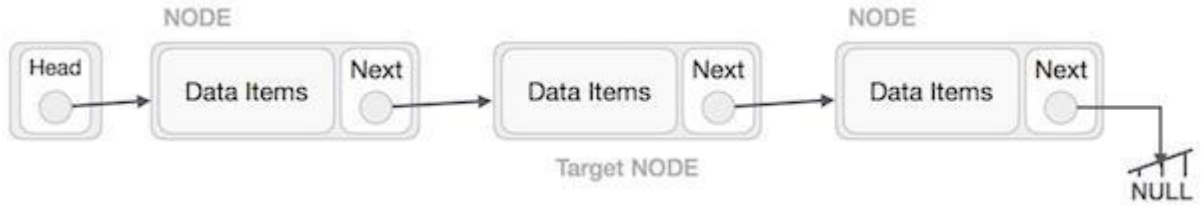
This will put the new node in the middle of the two. The new list should look like this –



Similar steps should be taken if the node is being inserted at the beginning of the list. While inserting it at the end, the second last node of the list should point to the new node and the new node will point to NULL

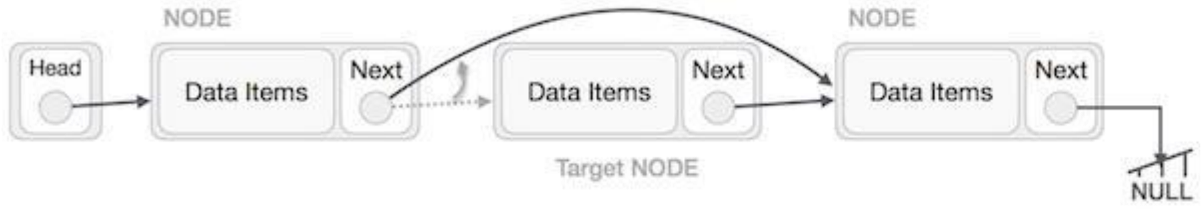
Deletion Operation

Deletion is also a more than one step process. We shall learn with pictorial representation. First, locate the target node to be removed, by using searching algorithms.



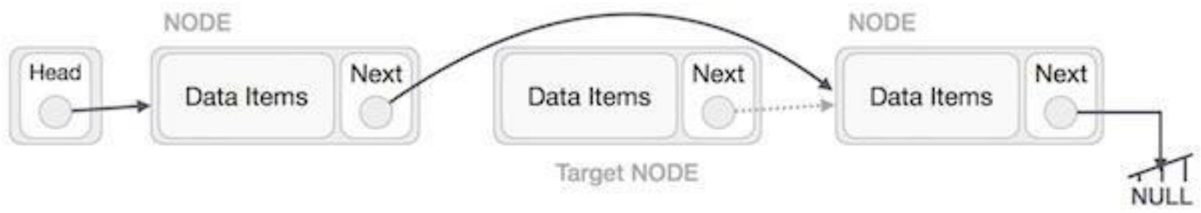
The left (previous) node of the target node now should point to the next node of the target node –

LeftNode.next −> TargetNode.next;

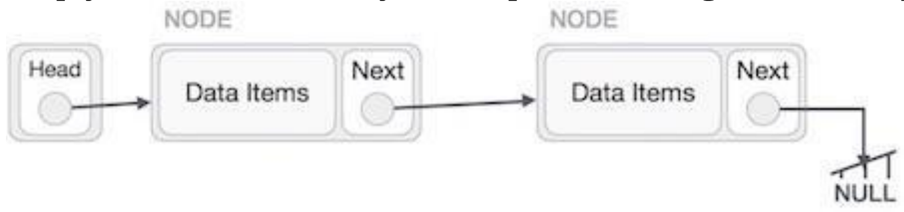


This will remove the link that was pointing to the target node. Now, using the following code, we will remove what the target node is pointing at.

TargetNode.next −> NULL;

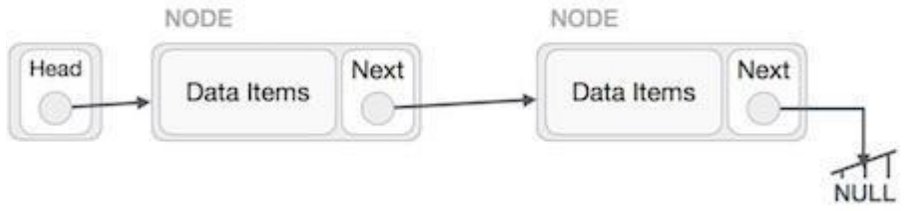


We need to use the deleted node. We can keep that in memory otherwise we can simply deallocate memory and wipe off the target node completely.

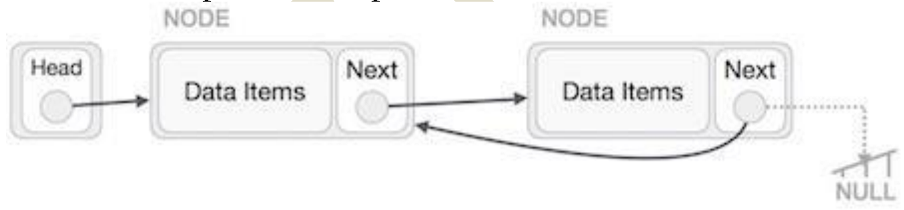


Reverse Operation

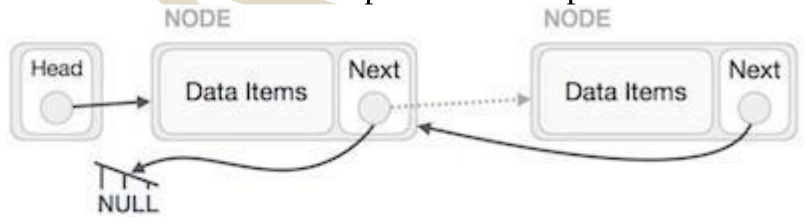
This operation is a thorough one. We need to make the last node to be pointed by the head node and reverse the whole linked list.



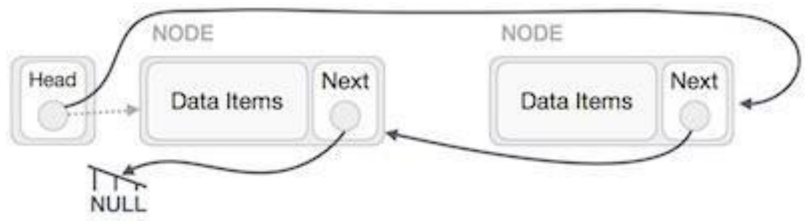
First, we traverse to the end of the list. It should be pointing to NULL. Now, we shall make it point to its previous node –



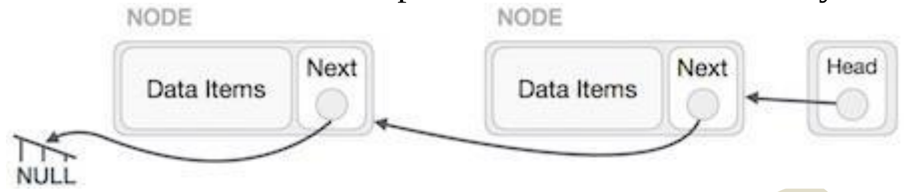
We have to make sure that the last node is not the lost node. So we'll have some temp node, which looks like the head node pointing to the last node. Now, we shall make all left side nodes point to their previous nodes one by one.



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



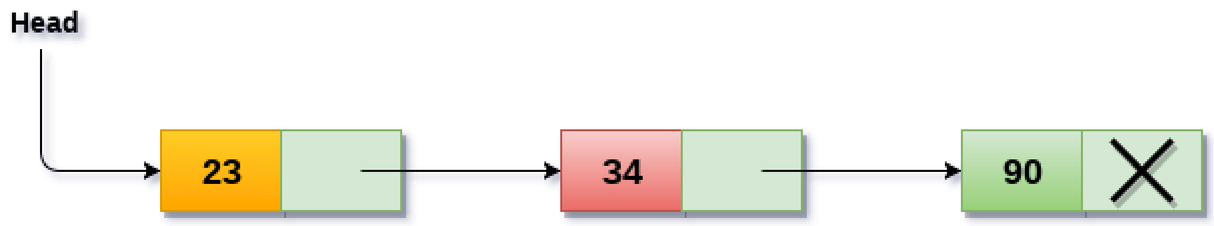
The linked list is now reversed.

Singly linked list or One way chain

Singly linked list can be defined as the collection of ordered set of elements. The number of elements may vary according to need of the program. A node in the singly linked list consist of two parts: data part and link part. Data part of the node stores actual information that is to be represented by the node while the link part of the node stores the address of its immediate successor.

One way chain or singly linked list can be traversed only in one direction. In other words, we can say that each node contains only next pointer, therefore we can not traverse the list in the reverse direction.

Consider an example where the marks obtained by the student in three subjects are stored in a linked list as shown in the figure.



In the above figure, the arrow represents the links. The data part of every node contains the marks obtained by the student in the different subject. The last node in the list is identified by the null pointer which is present in the address part of the last node. We can have as many elements we require, in the data part of the list.

Operations on Singly Linked List

There are various operations which can be performed on singly linked list. A list of all such operations is given below.

Node Creation

```
struct node
{
    int data;
    struct node *next;
};
struct node *head, *ptr;
ptr = (struct node *)malloc(sizeof(struct node *));
```

Insertion

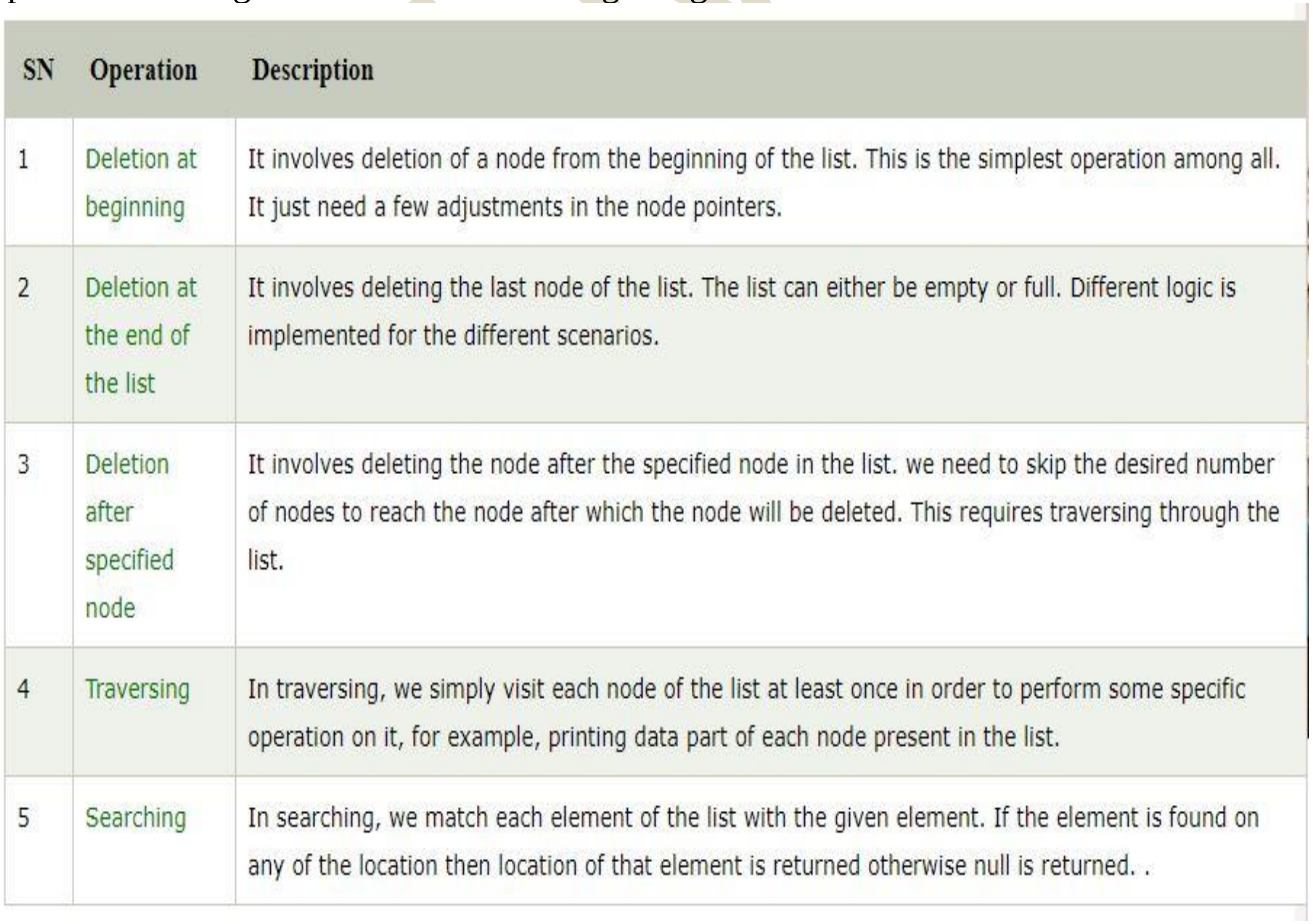
The insertion into a singly linked list can be performed at different positions. Based on the position of the new node being inserted, the insertion is categorized into the following categories.

SN	Operation	Description
1	Insertion at beginning	It involves inserting any element at the front of the list. We just need to a few link adjustments to make the new node as the head of the list.
2	Insertion at end of the list	It involves insertion at the last of the linked list. The new node can be inserted as the only node in the list or it can be inserted as the last one. Different logics are implemented in each scenario.
3	Insertion after specified node	It involves insertion after the specified node of the linked list. We need to skip the desired number of nodes in order to reach the node after which the new node will be inserted. .

SN	Operation	Description
1	Deletion at beginning	It involves deletion of a node from the beginning of the list. This is the simplest operation among all. It just need a few adjustments in the node pointers.
2	Deletion at the end of the list	It involves deleting the last node of the list. The list can either be empty or full. Different logic is implemented for the different scenarios.
3	Deletion after specified node	It involves deleting the node after the specified node in the list. we need to skip the desired number of nodes to reach the node after which the node will be deleted. This requires traversing through the list.
4	Traversing	In traversing, we simply visit each node of the list at least once in order to perform some specific operation on it, for example, printing data part of each node present in the list.
5	Searching	In searching, we match each element of the list with the given element. If the element is found on any of the location then location of that element is returned otherwise null is returned. .

Deletion and Traversing

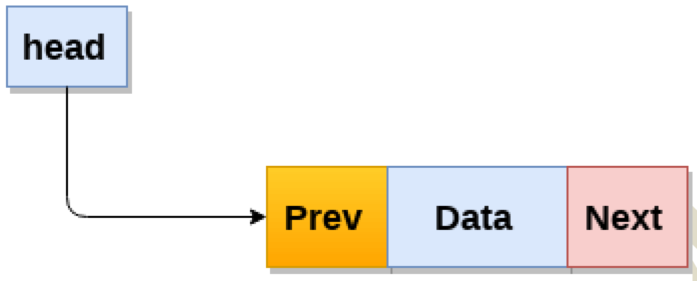
The Deletion of a node from a singly linked list can be performed at different positions. Based on the position of the node being deleted, the operation is categorized into the following categories.



SN	Operation	Description
1	Deletion at beginning	It involves deletion of a node from the beginning of the list. This is the simplest operation among all. It just need a few adjustments in the node pointers.
2	Deletion at the end of the list	It involves deleting the last node of the list. The list can either be empty or full. Different logic is implemented for the different scenarios.
3	Deletion after specified node	It involves deleting the node after the specified node in the list. we need to skip the desired number of nodes to reach the node after which the node will be deleted. This requires traversing through the list.
4	Traversing	In traversing, we simply visit each node of the list at least once in order to perform some specific operation on it, for example, printing data part of each node present in the list.
5	Searching	In searching, we match each element of the list with the given element. If the element is found on any of the location then location of that element is returned otherwise null is returned. .

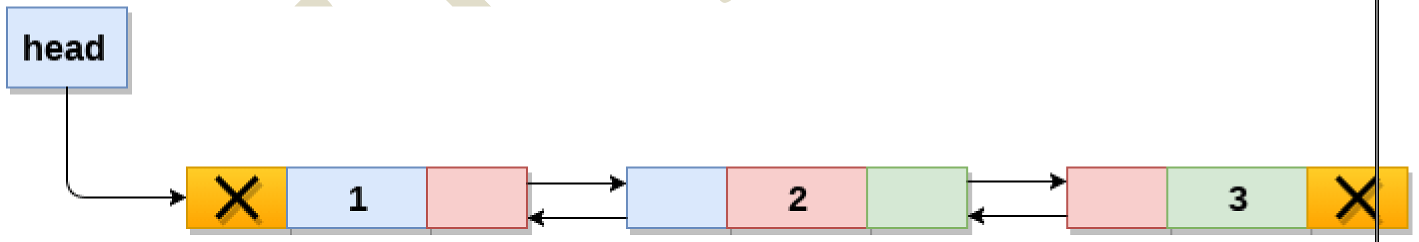
Doubly linked list

Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence. Therefore, in a doubly linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer) , pointer to the previous node (previous pointer). A sample node in a doubly linked list is shown in the figure.



Node

A doubly linked list containing three nodes having numbers from 1 to 3 in their data part, is shown in the following image.



Doubly Linked List

In C, structure of a node in doubly linked list can be given as :

```
1.  struct node
2.  {
3.      struct node *prev;
4.      int data;
5.      struct node *next;
6.  }
```

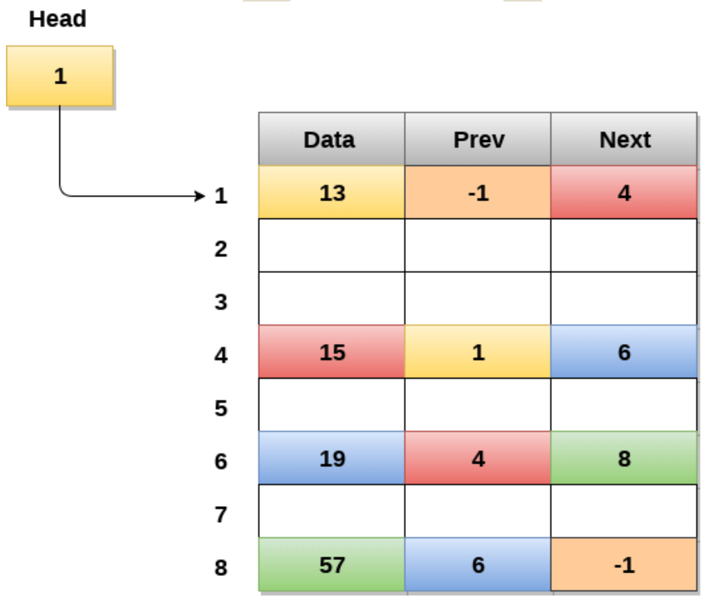
The **prev** part of the first node and the **next** part of the last node will always contain null indicating end in each direction.

In a singly linked list, we could traverse only in one direction, because each node contains address of the next node and it doesn't have any record of its previous nodes. However, doubly linked list overcome this limitation of singly linked list. Due to the fact that, each node of the list contains the address of its previous node, we can find all the details about the previous node as well by using the previous address stored inside the previous part of each node.

<u>Memory Representation of a doubly linked list</u>

Memory Representation of a doubly linked list is shown in the following image. Generally, doubly linked list consumes more space for every node and therefore, causes more expansive basic operations such as insertion and deletion. However, we can easily manipulate the elements of the list since the list maintains pointers in both the directions (forward and backward).

In the following image, the first element of the list that is i.e. 13 stored at address
1. The head pointer points to the starting address 1. Since this is the first element being added to the list therefore the **prev** of the list **contains** null. The next node of the list resides at address 4 therefore the first node contains 4 in its next pointer. We can traverse the list in this way until we find any node containing null or -1 in its next part.



	Data	Prev	Next
1	13	-1	4
2			
3			
4	15	1	6
5			
6	19	4	8
7			
8	57	6	-1

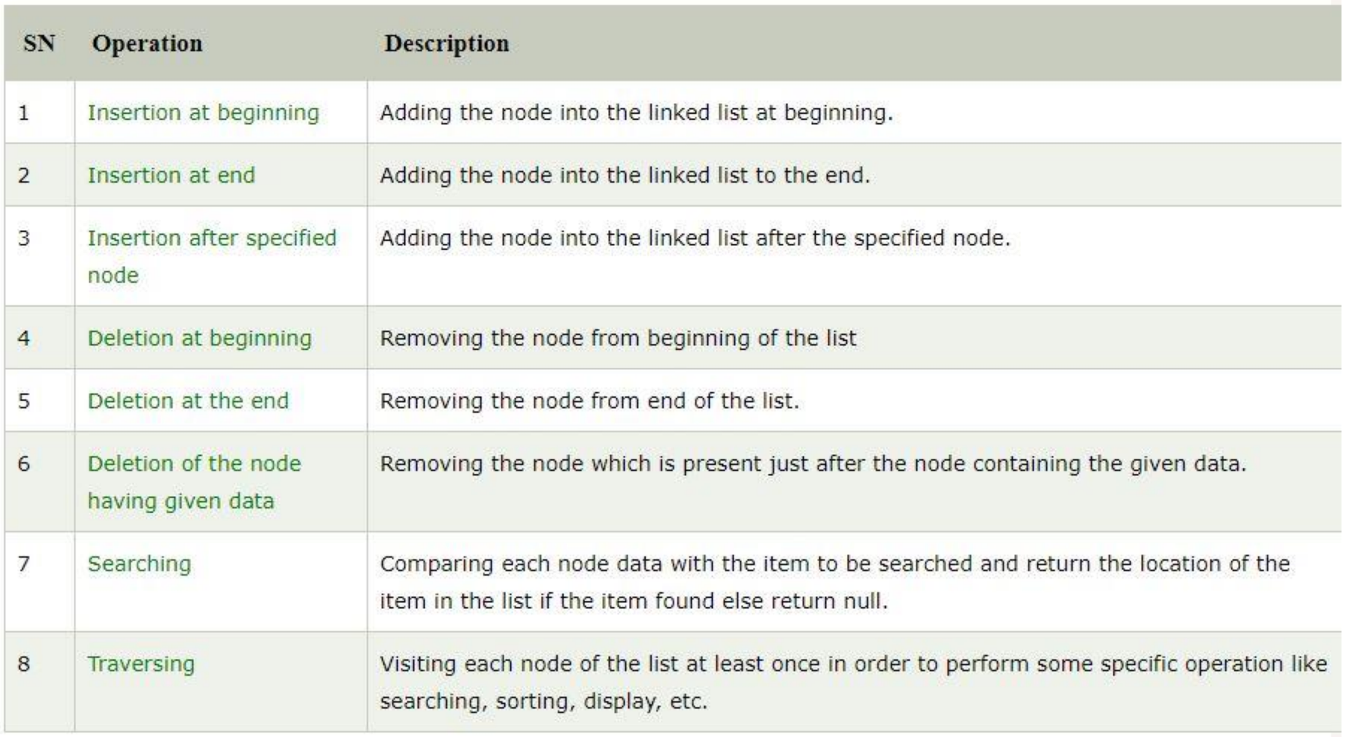
Memory Representation of a Doubly linked list

Operations on doubly linked list

Node Creation

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

All the remaining operations regarding doubly linked list are described in the following table.



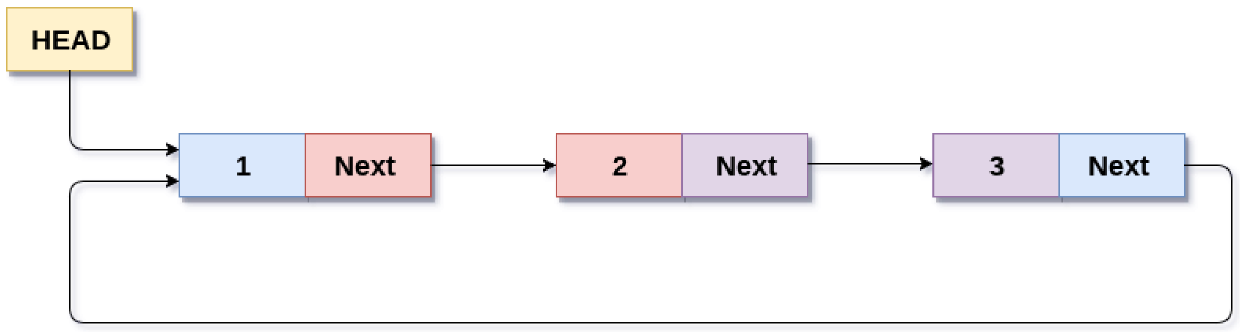
SN	Operation	Description
1	Insertion at beginning	Adding the node into the linked list at beginning.
2	Insertion at end	Adding the node into the linked list to the end.
3	Insertion after specified node	Adding the node into the linked list after the specified node.
4	Deletion at beginning	Removing the node from beginning of the list
5	Deletion at the end	Removing the node from end of the list.
6	Deletion of the node having given data	Removing the node which is present just after the node containing the given data.
7	Searching	Comparing each node data with the item to be searched and return the location of the item in the list if the item found else return null.
8	Traversing	Visiting each node of the list at least once in order to perform some specific operation like searching, sorting, display, etc.

Circular Singly Linked List

In a circular Singly linked list, the last node of the list contains a pointer to the first node of the list. We can have circular singly linked list as well as circular doubly linked list.

We traverse a circular singly linked list until we reach the same node where we started. The circular singly liked list has no beginning and no ending. There is no null value present in the next part of any of the nodes.

The following image shows a circular singly linked list.



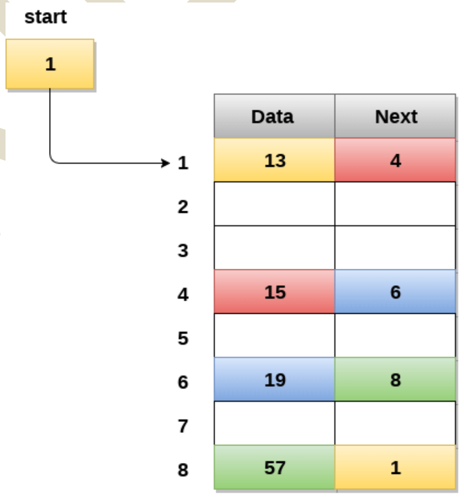
Circular Singly Linked List

Circular linked list are mostly used in task maintenance in operating systems. There are many examples where circular linked list are being used in computer science including browser surfing where a record of pages visited in the past by the user, is maintained in the form of circular linked lists and can be accessed again on clicking the previous button.

Memory Representation of circular linked list:

In the following image, memory representation of a circular linked list containing marks of a student in 4 subjects. However, the image shows a glimpse of how the circular list is being stored in the memory. The start or head of the list is pointing to the element with the index 1 and containing 13 marks in the data part and 4 in the next part. Which means that it is linked with the node that is being stored at 4th index of the list.

However, due to the fact that we are considering circular linked list in the memory therefore the last node of the list contains the address of the first node of the list.

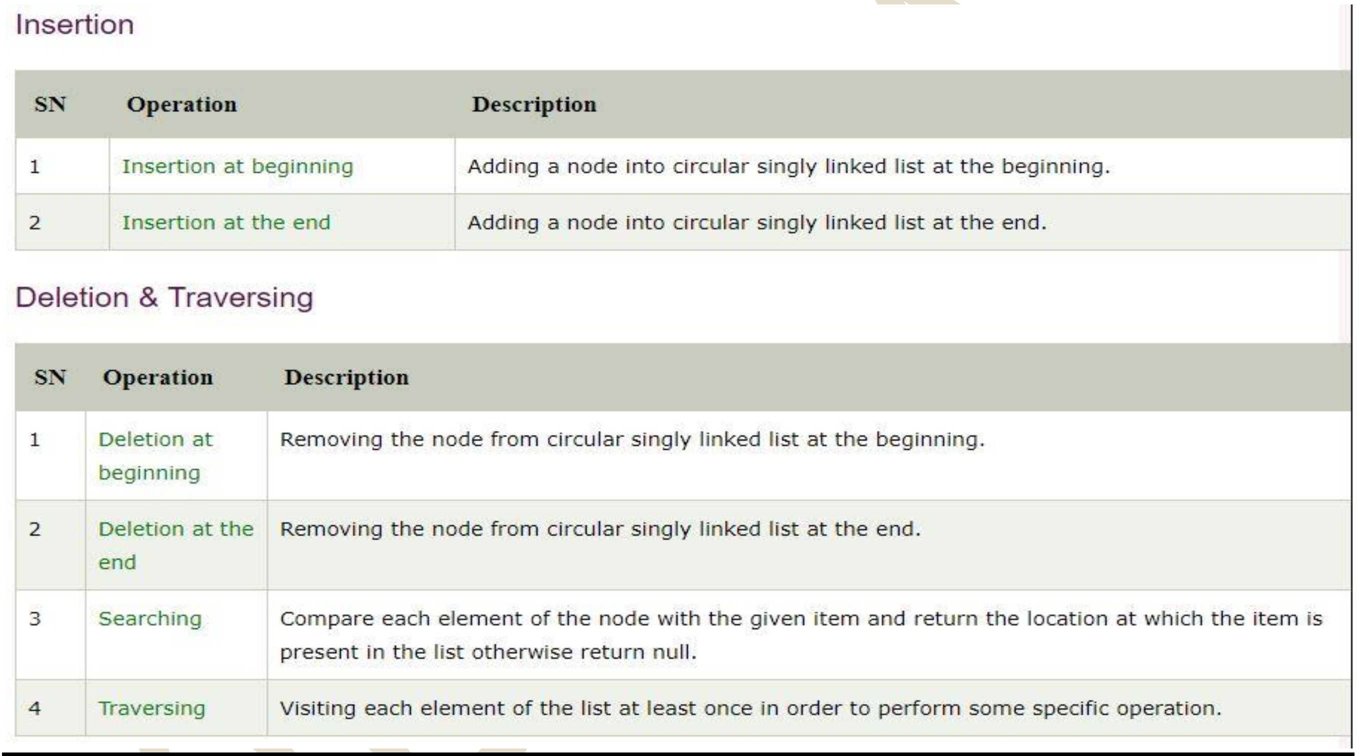


	Data	Next
1	13	4
2		
3		
4	15	6
5		
6	19	8
7		
8	57	1

Memory Representation of a circular linked list

We can also have more than one number of linked list in the memory with the different start pointers pointing to the different start nodes in the list. The last node is identified by its next part which contains the address of the start node of the list. We must be able to identify the last node of any linked list so that we can find out the number of iterations which need to be performed while traversing the list.

Operations on Circular Singly linked list:



Insertion

SN	Operation	Description
1	Insertion at beginning	Adding a node into circular singly linked list at the beginning.
2	Insertion at the end	Adding a node into circular singly linked list at the end.

Deletion & Traversing

SN	Operation	Description
1	Deletion at beginning	Removing the node from circular singly linked list at the beginning.
2	Deletion at the end	Removing the node from circular singly linked list at the end.
3	Searching	Compare each element of the node with the given item and return the location at which the item is present in the list otherwise return null.
4	Traversing	Visiting each element of the list at least once in order to perform some specific operation.

UNIT-4

Stacks-Operations, array and linked representations of stacks, stack applications,
Queues-operations, array and linked representations.
Hash Table Representation: hash functions, collision resolution-separate chaining, open addressing-linear probing, quadratic probing, double hashing and rehashing, extendible hashing.

What is a Stack?

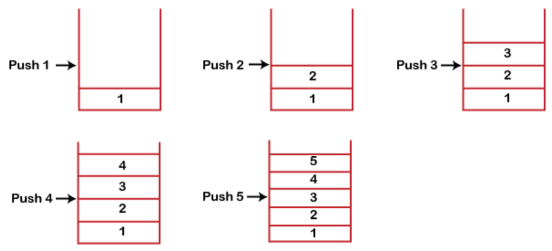
A Stack is a linear data structure that follows the **LIFO (Last-In-First-Out)** principle. Stack has one end, whereas the Queue has two ends (**front and rear**). It contains only one pointer **top pointer** pointing to the topmost element of the stack. Whenever an element is added in the stack, it is added on the top of the stack, and the element can be deleted only from the stack. In other words, a ***stack can be defined as a container in which insertion and deletion can be done from the one end known as the top of the stack.***

Some key points related to stack

- It is called as stack because it behaves like a real-world stack, piles of books, etc.
- A Stack is an abstract data type with a pre-defined capacity, which means that it can store the elements of a limited size.
- It is a data structure that follows some order to insert and delete the elements, and that order can be LIFO or FILO.
-

Working of Stack

Stack works on the LIFO pattern. As we can observe in the below figure there are five memory blocks in the stack; therefore, the size of the stack is 5. Suppose we want to store the elements in a stack and let's assume that stack is empty. We have taken the stack of size 5 as shown below in which we are pushing the elements one by one until the stack becomes full.



Since our stack is full as the size of the stack is 5. In the above cases, we can observe that it goes from the top to the bottom when we were entering the new element in the stack. The stack gets filled up from the bottom to the top.

When we perform the delete operation on the stack, there is only one way for entry and exit as the other end is closed. It follows the LIFO pattern, which means that the value entered first will be removed last. In the above case, the value 5 is entered first, so it will be removed only after the deletion of all the other elements.

Standard Stack Operations

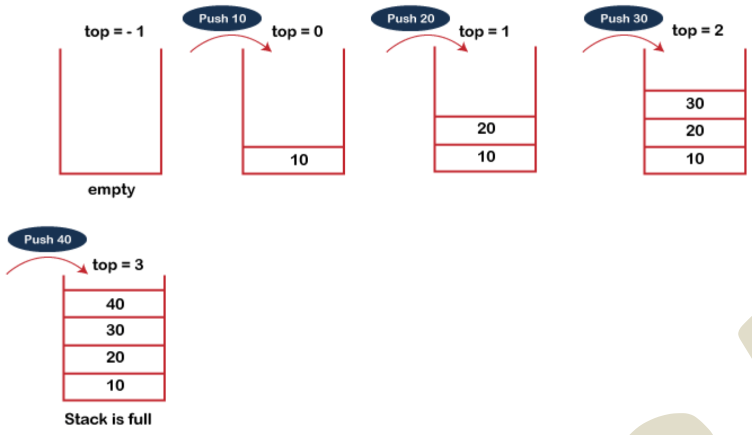
The following are some common operations implemented on the stack:

- **push():** When we insert an element in a stack then the operation is known as a push. If the stack is full then the overflow condition occurs.
- **pop():** When we delete an element from the stack, the operation is known as a pop. If the stack is empty means that no element exists in the stack, this state is known as an underflow state.
- **isEmpty():** It determines whether the stack is empty or not.
- **isFull():** It determines whether the stack is full or not.'
- **peek():** It returns the element at the given position.
- **count():** It returns the total number of elements available in a stack.
- **change():** It changes the element at the given position.
- **display():** It prints all the elements available in the stack.
-

PUSH operation

The steps involved in the PUSH operation is given below:

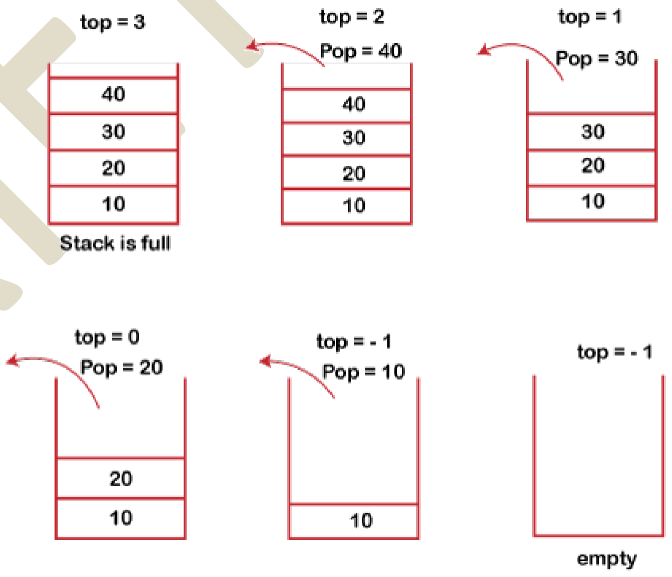
- Before inserting an element in a stack, we check whether the stack is full.
- If we try to insert the element in a stack, and the stack is full, then the ***overflow*** condition occurs.
- When we initialize a stack, we set the value of top as -1 to check that the stack is empty.
- When the new element is pushed in a stack, first, the value of the top gets incremented, i.e., **top=top+1,** and the element will be placed at the new position of the **top**.
- The elements will be inserted until we reach the ***max*** size of the stack.



POP operation

The steps involved in the POP operation is given below:

- Before deleting the element from the stack, we check whether the stack is empty.
- If we try to delete the element from the empty stack, then the ***underflow*** condition occurs.
- If the stack is not empty, we first access the element which is pointed by the ***top***
- Once the pop operation is performed, the top is decremented by 1, i.e., **top=top-1**.



Applications of Stack

The following are the applications of the stack:

- **<u>Balancing of symbols</u>:** Stack is used for balancing a symbol. For example, we have the following program:

1. **int** main()
2. {
3.   cout<<"Hello";
4.   cout<<"javaTpoint";
5. }

As we know, each program has *an opening* and *closing* braces; when the opening braces come, we push the braces in a stack, and when the closing braces appear, we pop the opening braces from the stack. Therefore, the net value comes out to be zero. If any symbol is left in the stack, it means that some syntax occurs in a program.

- **<u>String reversal</u>:** Stack is also used for reversing a string. For example, we want to reverse a "**javaTpoint**" string, so we can achieve this with the help of a stack.
  First, we push all the characters of the string in a stack until we reach the null character.
  After pushing all the characters, we start taking out the character one by one until we reach the bottom of the stack.
- **<u>UNDO/REDO</u>:** It can also be used for performing UNDO/REDO operations. For example, we have an editor in which we write 'a', then 'b', and then 'c'; therefore, the text written in an editor is abc. So, there are three states, a, ab, and abc, which are stored in a stack. There would be two stacks in which one stack shows UNDO state, and the other shows REDO state.
  If we want to perform UNDO operation, and want to achieve 'ab' state, then we implement pop operation.
- **<u>Recursion</u>:** The recursion means that the function is calling itself again. To maintain the previous states, the compiler creates a system stack in which all the previous records of the function are maintained.
- **<u>DFS(Depth First Search):</u>** This search is implemented on a Graph, and Graph uses the stack data structure.
- **<u>Backtracking:</u>** Suppose we have to create a path to solve a maze problem. If we are moving in a particular path, and we realize that we come on the wrong way. In order to come at the beginning of the path to create a new path, we have to use the stack data structure.
- **<u>Expression conversion:</u>** Stack can also be used for expression conversion. This is one of the most important applications of stack. The list of the expression conversion is given below:
- Infix to prefix
- Infix to postfix

- Prefix to infix
- Prefix to postfix

Postfix to infix

- **Memory management:** The stack manages the memory. The memory is assigned in the contiguous memory blocks. The memory is known as stack memory as all the variables are assigned in a function call stack memory. The memory size assigned to the program is known to the compiler. When the function is created, all its variables are assigned in the stack memory. When the function completed its execution, all the variables assigned in the stack are released.

Array implementation of Stack

In array implementation, the stack is formed by using the array. All the operations regarding the stack are performed using arrays. Lets see how each operation can be implemented on the stack using array data structure.

Adding an element onto the stack (push operation)

Adding an element into the top of the stack is referred to as push operation. Push operation involves following two steps.

1. Increment the variable Top so that it can now refere to the next memory location.
2. Add element at the position of incremented top. This is referred to as adding new element at the top of the stack.

Stack is overflown when we try to insert an element into a completely filled stack therefore, our main function must always avoid stack overflow condition.

Algorithm:

```
1. begin
2.     if top = n then stack full
3.     top = top + 1
4.     stack (top) : = item;
5. end
```

Time Complexity : o(1)

implementation of push algorithm in C language

```
void push (int val,int n) //n is size of the stack
{
    if (top == n )
    printf("\n Overflow");
    else
    {
    top = top +1;
    stack[top] = val;
    }
}
```

Deletion of an element from a stack (Pop operation)

Deletion of an element from the top of the stack is called pop operation. The value of the variable top will be incremented by 1 whenever an item is deleted from the stack. The top most element of the stack is stored in an another variable and then the top is decremented by 1. the operation returns the deleted value that was stored in another variable as the result.

The underflow condition occurs when we try to delete an element from an already empty stack.

Algorithm :

```
1. begin
2.     if top = 0 then stack empty;
3.     item := stack(top);
4.     top = top - 1;
5. end;
```

Time Complexity : o(1)

Implementation of POP algorithm using C language

```
int pop ()
{
    if(top == -1)
    {
        printf("Underflow");
        return 0;
    }
    else
    {
        return stack[top - - ];
    }
}
```

Visiting each element of the stack (Peek operation)

Peek operation involves returning the element which is present at the top of the stack without deleting it. Underflow condition can occur if we try to return the top element in an already empty stack.

Algorithm :

PEEK (STACK, TOP)

```
1. Begin
2.     if top = -1 then stack empty
3.     item = stack[top]
4.     return item
```

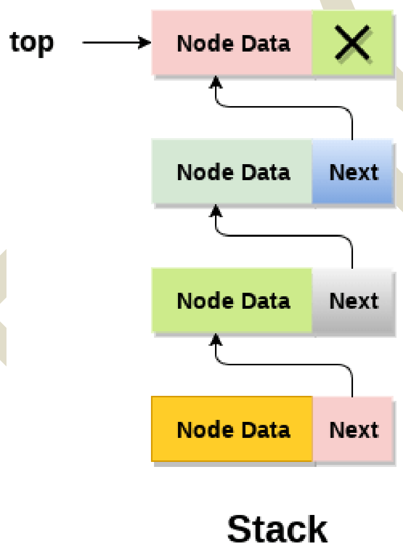
5. End

<u>Time complexity</u>: o(n)

<u>Linked list implementation of stack</u>

Instead of using array, we can also use linked list to implement stack. Linked list allocates the memory dynamically. However, time complexity in both the scenario is same for all the operations i.e. push, pop and peek.

In linked list implementation of stack, the nodes are maintained non-contiguously in the memory. Each node contains a pointer to its immediate successor node in the stack. Stack is said to be overflown if the space left in the memory heap is not enough to create a node.



The top most node in the stack always contains null in its address field. Lets discuss the way in which, each operation is performed in linked list implementation of stack.

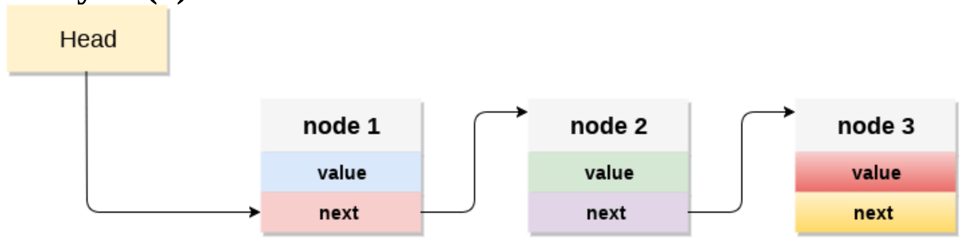
Adding a node to the stack (Push operation)

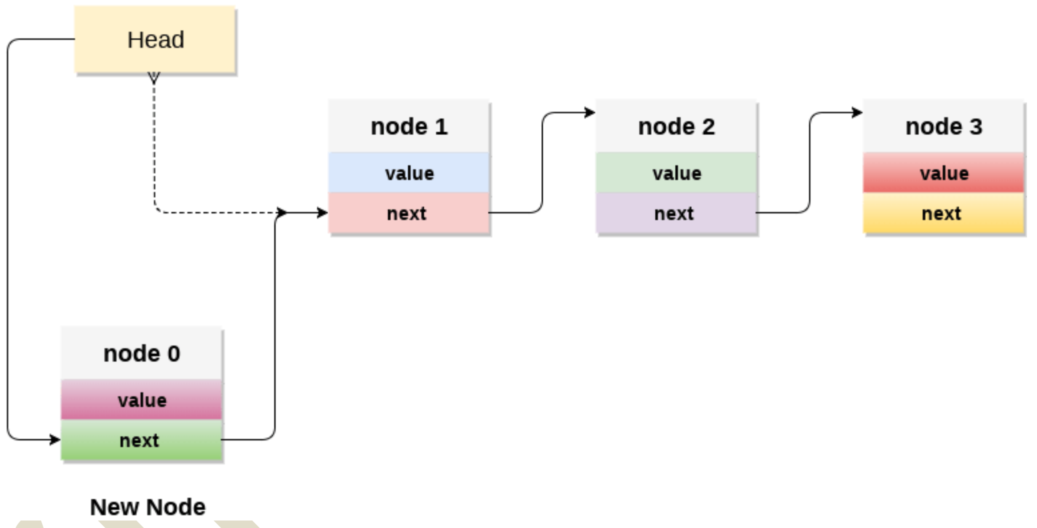
Adding a node to the stack is referred to as **push** operation. Pushing an element to a stack in linked list implementation is different from that of an array implementation. In order to push an element onto the stack, the following steps are involved.

1. Create a node first and allocate memory to it.
2. If the list is empty then the item is to be pushed as the start node of the list. This includes assigning value to the data part of the node and assign null to the address part of the node.

3. If there are some nodes in the list already, then we have to add the new element in the beginning of the list (to not violate the property of the stack). For this purpose, assign the address of the starting element to the address field of the new node and make the new node, the starting node of the list.

Time Complexity : o(1)





<u>Deleting a node from the stack (POP operation)</u>

Deleting a node from the top of stack is referred to as **pop** operation. Deleting a node from the linked list implementation of stack is different from that in the array implementation. In order to pop an element from the stack, we need to follow the following steps :

1. **Check for the underflow condition:** The underflow condition occurs when we try to pop from an already empty stack. The stack will be empty if the head pointer of the list points to null.
2. **Adjust the head pointer accordingly:** In stack, the elements are popped only from one end, therefore, the value stored in the head pointer must be deleted and the node must be freed. The next node of the head node now becomes the head node.

Time Complexity : o(n)

<u>Display the nodes (Traversing</u>)

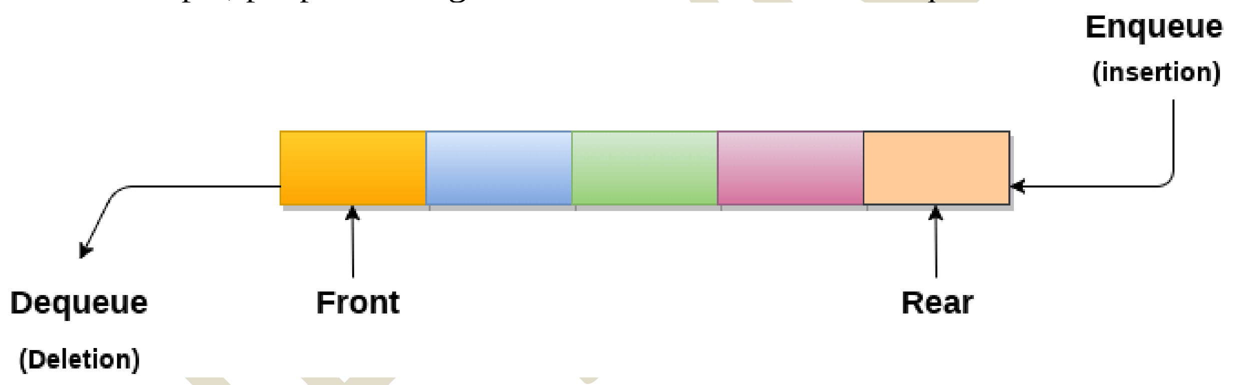
Displaying all the nodes of a stack needs traversing all the nodes of the linked list organized in the form of stack. For this purpose, we need to follow the following steps.

1. Copy the head pointer into a temporary pointer.
2. Move the temporary pointer through all the nodes of the list and print the value field attached to every node.

Time Complexity : o(n)

<u>Queue:</u>

1. A queue can be defined as an ordered list which enables insert operations to be performed at one end called **REAR** and delete operations to be performed at another end called **FRONT**.
2. Queue is referred to be as First In First Out list.
3. For example, people waiting in line for a rail ticket form a queue.

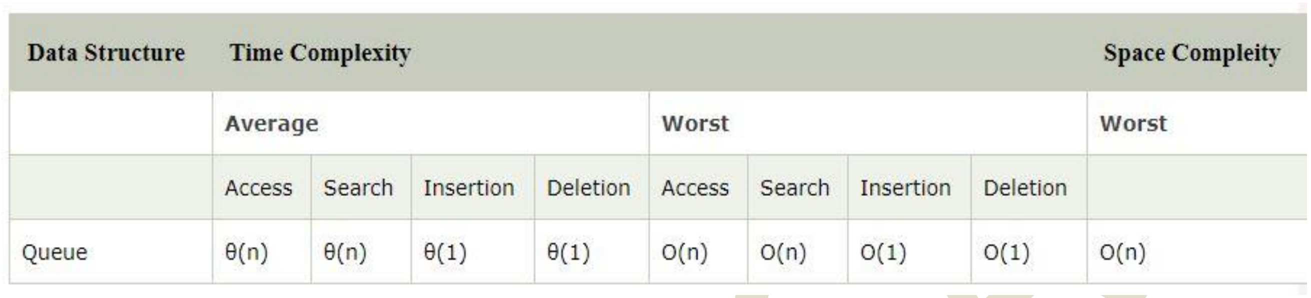


<u>Applications of Queue</u>

Due to the fact that queue performs actions on first in first out basis which is quite fair for the ordering of actions. There are various applications of queues discussed as below.

1. Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
2. Queues are used in asynchronous transfer of data (where data is not being transferred at the same rate between two processes) for eg. pipes, file IO, sockets.
3. Queues are used as buffers in most of the applications like MP3 media player, CD player, etc.

4. Queue are used to maintain the play list in media players in order to add and remove the songs from the play-list.
5. Queues are used in operating systems for handling interrupts.



Data Structure	Time Complexity								Space Compleity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Queue	θ(n)	θ(n)	θ(1)	θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)

<u>Operations on Queue</u>

There are two fundamental operations performed on a Queue:

- **Enqueue:** The enqueue operation is used to insert the element at the rear end of the queue. It returns void.
- **Dequeue:** The dequeue operation performs the deletion from the front-end of the queue. It also returns the element which has been removed from the front-end. It returns an integer value. The dequeue operation can also be designed to void.
- **Peek:** This is the third operation that returns the element, which is pointed by the front pointer in the queue but does not delete it.
- **Queue overflow (isfull):** When the Queue is completely full, then it shows the overflow condition.
- **Queue underflow (isempty):** When the Queue is empty, i.e., no elements are in the Queue then it throws the underflow condition.

A Queue can be represented as a container opened from both the sides in which the element can be enqueued from one side and dequeued from another side as shown in the below figure:

Implementation of Queue

There are two ways of implementing the Queue**:**

- **Sequential allocation:** The sequential allocation in a Queue can be implemented using an array.

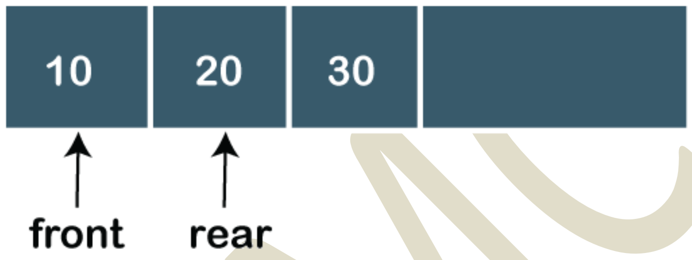
- **Linked list allocation:** The linked list allocation in a Queue can be implemented using a linked list.

Types of Queue

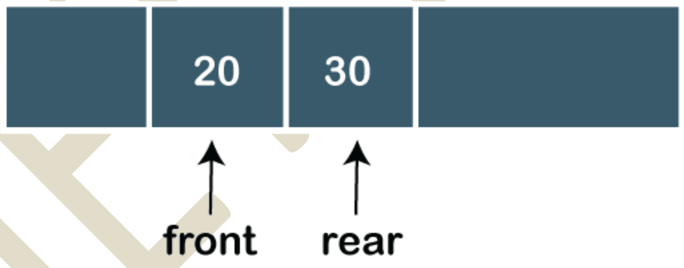
There are four types of Queues:

- **Linear Queue**

In Linear Queue, an insertion takes place from one end while the deletion occurs from another end. The end at which the insertion takes place is known as the rear end, and the end at which the deletion takes place is known as front end. It strictly follows the FIFO rule. The linear Queue can be represented, as shown in the below figure:



The above figure shows that the elements are inserted from the rear end, and if we insert more elements in a Queue, then the rear value gets incremented on every insertion. If we want to show the deletion, then it can be represented as:

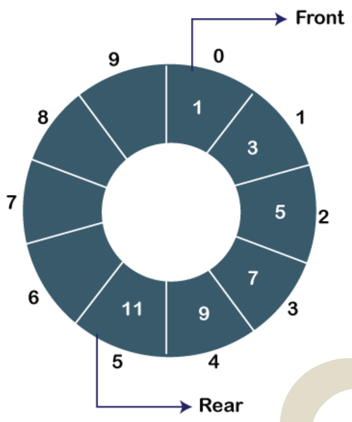


In the above figure, we can observe that the front pointer points to the next element, and the element which was previously pointed by the front pointer was deleted.

The major drawback of using a **linear Queue** is that insertion is done only from the rear end. If the first three elements are deleted from the Queue, we cannot insert more elements even though the space is available in a Linear Queue. In this case, the linear Queue shows the **overflow** condition as the rear is pointing to the last element of the Queue.

- **Circular Queue**

In Circular Queue, all the nodes are represented as circular. It is similar to the linear Queue except that the last element of the queue is connected to the first element. It is also known as **Ring Buffer** as all the ends are connected to another end. The circular queue can be represented as:



The drawback that occurs in a linear queue is overcome by using the circular queue. If the empty space is available in a circular queue, the new element can be added in an empty space by simply incrementing the value of rear.

- **Priority Queue**

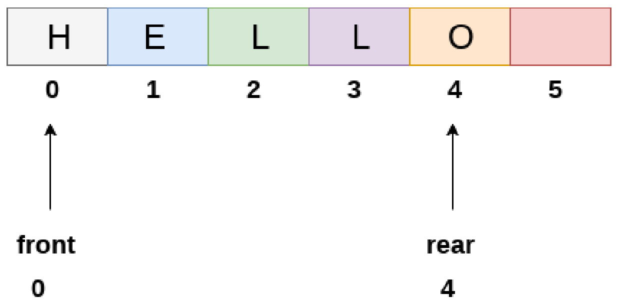
A priority queue is another special type of Queue data structure in which each element has some priority associated with it. Based on the priority of the element, the elements are arranged in a priority queue. If the elements occur with the same priority, then they are served according to the FIFO principle.
In priority Queue, the insertion takes place based on the arrival while the deletion occurs based on the priority. The priority Queue can be shown as:
The above figure shows that the highest priority element comes first and the elements of the same priority are arranged based on FIFO structure.

- **Deque**

Both the Linear Queue and Deque are different as the linear queue follows the FIFO principle whereas, deque does not follow the FIFO principle. In Deque, the insertion and deletion can occur from both ends.

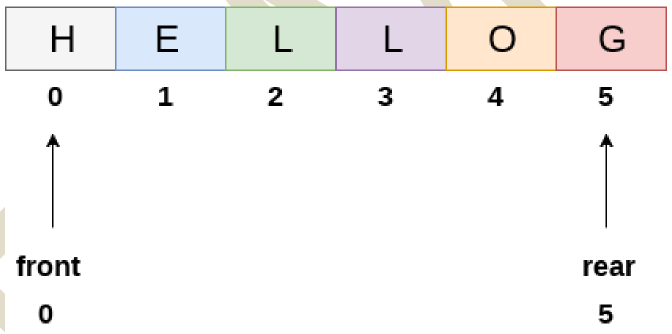
Array representation of Queue

We can easily represent queue by using linear arrays. There are two variables i.e. front and rear,that are implemented in the case of every queue. Front and rear variables point to the position from where insertions and deletions are performed in a queue. Initially, the value of front and queue is -1 which represents an empty queue. Array representation of a queue containing 5 elements along with the respective values of front and rear, is shown in the following figure.



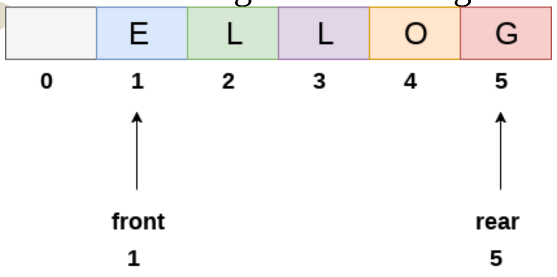
Queue

The above figure shows the queue of characters forming the English word **"HELLO"**. Since, No deletion is performed in the queue till now, therefore the value of front remains -1 . However, the value of rear increases by one every time an insertion is performed in the queue. After inserting an element into the queue shown in the above figure, the queue will look something like following. The value of rear will become 5 while the value of front remains same.



Queue after inserting an element

After deleting an element, the value of front will increase from -1 to 0. however, the queue will look something like following.



Queue after deleting an element

Algorithm to insert any element in a queue

Check if the queue is already full by comparing rear to max - 1. if so, then return an overflow error.

If the item is to be inserted as the first element in the list, in that case set the value of front and rear to 0 and insert the element at the rear end.

Otherwise keep increasing the value of rear and insert each element one by one having rear as the index.

Algorithm

```
o Step 1: IF REAR = MAX - 1
  Write OVERFLOW
  Go to step
  [END OF IF]
o Step 2: IF FRONT = -1 and REAR = -1
  SET FRONT = REAR = 0
  ELSE
  SET REAR = REAR + 1
  [END OF IF]
o Step 3: Set QUEUE[REAR] = NUM
o Step 4: EXIT
```

Algorithm to delete an element from the queue

If, the value of front is -1 or value of front is greater than rear , write an underflow message and exit.

Otherwise, keep increasing the value of front and return the item stored at the front end of the queue at each time.

Algorithm

```
o Step 1: IF FRONT = -1 or FRONT > REAR
  Write UNDERFLOW
  ELSE
  SET VAL = QUEUE[FRONT]
  SET FRONT = FRONT + 1
  [END OF IF]
o Step 2: EXIT
```

Linked List implementation of Queue

Due to the drawbacks discussed in the previous section of this tutorial, the array implementation can not be used for the large scale applications where the queues are implemented. One of the alternative of array implementation is linked list implementation of queue.

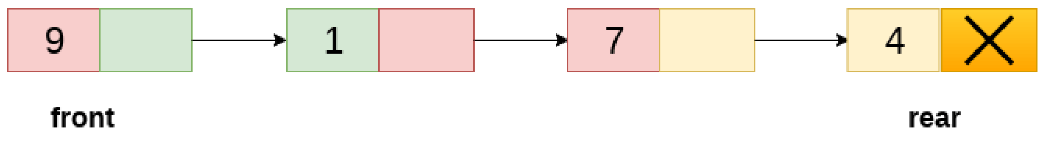
The storage requirement of linked representation of a queue with n elements is o(n) while the time requirement for operations is o(1).

In a linked queue, each node of the queue consists of two parts i.e. data part and the link part. Each element of the queue points to its immediate next element in the memory.

In the linked queue, there are two pointers maintained in the memory i.e. front pointer and rear pointer. The front pointer contains the address of the starting element of the queue while the rear pointer contains the address of the last element of the queue.

Insertion and deletions are performed at rear and front end respectively. If front and rear both are NULL, it indicates that the queue is empty.

The linked representation of queue is shown in the following figure.



Linked Queue

Operation on Linked Queue

There are two basic operations which can be implemented on the linked queues. The operations are Insertion and Deletion

Insert operation

The insert operation append the queue by adding an element to the end of the queue. The new element will be the last element of the queue. Firstly, allocate the memory for the new node ptr by using the following statement.

1. Ptr = (struct node *) malloc (sizeof(struct node));

There can be the two scenario of inserting this new node ptr into the linked queue.

In the first scenario, we insert element into an empty queue. In this case, the condition front = NULL becomes true. Now, the new element will be added as the only element of the queue and the next pointer of front and rear pointer both, will point to NULL.

```
1. ptr -> data = item;
2.         if(front == NULL)
3.         {
4.             front = ptr;
5.             rear = ptr;
6.             front -> next = NULL;
7.             rear -> next = NULL;
8.         }
```

In the second case, the queue contains more than one element. The condition front = NULL becomes false. In this scenario, we need to update the end pointer rear so that the next pointer of rear will point to the new node ptr. Since, this is a linked queue, hence we also need to make the rear pointer point to the newly added node **ptr**. We also need to make the next pointer of rear point to NULL.

```
1. rear -> next = ptr;
2.             rear = ptr;
3.             rear->next = NULL;
```

In this way, the element is inserted into the queue. The algorithm and the C implementation is given as follows.

Algorithm

```
○ Step 1: Allocate the space for the new node PTR
○ Step 2: SET PTR -> DATA = VAL
○ Step 3: IF FRONT = NULL
      SET FRONT = REAR = PTR
      SET FRONT -> NEXT = REAR -> NEXT = NULL
      ELSE
      SET REAR -> NEXT = PTR
      SET REAR = PTR
      SET REAR -> NEXT = NULL
      [END OF IF]
○ Step 4: END
```

Deletion:

Deletion operation removes the element that is first inserted among all the queue elements. Firstly, we need to check either the list is empty or not. The condition front == NULL becomes true if the list is empty, in this case , we simply write underflow on the console and make exit.

Otherwise, we will delete the element that is pointed by the pointer front. For this purpose, copy the node pointed by the front pointer into the pointer ptr. Now, shift the front pointer, point to its next node and free the node pointed by the node ptr. This is done by using the following statements.

1. ptr = front;
2. front = front -> next;
3. free(ptr);

The algorithm and C function is given as follows.

Algorithm

```
○ Step 1: IF FRONT = NULL
  Write " Underflow "
  Go to Step 5
  [END OF IF]
○ Step 2: SET PTR = FRONT
○ Step 3: SET FRONT = FRONT -> NEXT
○ Step 4: FREE PTR
○ Step 5: END
```

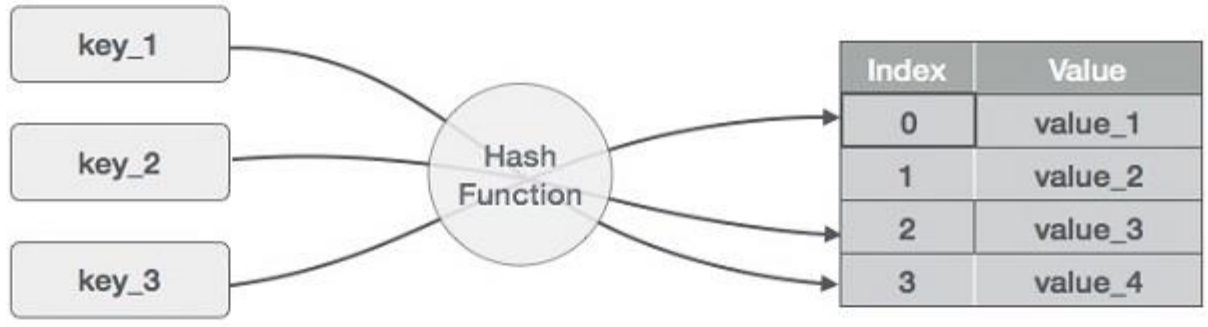
Hash Table Representation

Hash Table is a data structure which stores data in an associative manner. In a hash table, data is stored in an array format, where each data value has its own unique index value. Access of data becomes very fast if we know the index of the desired data.

Thus, it becomes a data structure in which insertion and search operations are very fast irrespective of the size of the data. Hash Table uses an array as a storage medium and uses hash technique to generate an index where an element is to be inserted or is to be located from.

Hashing

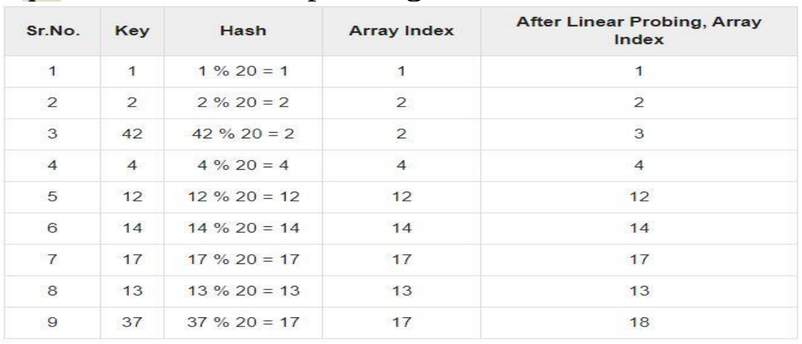
Hashing is a technique to convert a range of key values into a range of indexes of an array. We're going to use modulo operator to get a range of key values. Consider an example of hash table of size 20, and the following items are to be stored. Item are in the (key,value) format



- (1,20)
- (2,70)
- (42,80)
- (4,25)
- (12,44)
- (14,32)
- (17,11)
- (13,78)
- (37,98)

Linear Probing

As we can see, it may happen that the hashing technique is used to create an already used index of the array. In such a case, we can search the next empty location in the array by looking into the next cell until we find an empty cell. This technique is called linear probing.



Sr.No.	Key	Hash	Array Index	After Linear Probing, Array Index
1	1	1 % 20 = 1	1	1
2	2	2 % 20 = 2	2	2
3	42	42 % 20 = 2	2	3
4	4	4 % 20 = 4	4	4
5	12	12 % 20 = 12	12	12
6	14	14 % 20 = 14	14	14
7	17	17 % 20 = 17	17	17
8	13	13 % 20 = 13	13	13
9	37	37 % 20 = 17	17	18

Basic Operations

Following are the basic primary operations of a hash table.

- **Search** − Searches an element in a hash table.
- **Insert** − inserts an element in a hash table.
- **delete** − Deletes an element from a hash table.

DataItem

Define a data item having some data and key, based on which the search is to be conducted in a hash table.

```
struct DataItem {
int data;
int key;
};
```

Hash Method

Define a hashing method to compute the hash code of the key of the data item.

```
int hashCode(int key){
return key % SIZE;
}
```

Search Operation

Whenever an element is to be searched, compute the hash code of the key passed and locate the element using that hash code as index in the array. Use linear probing to get the element ahead if the element is not found at the computed hash code.

Insert Operation

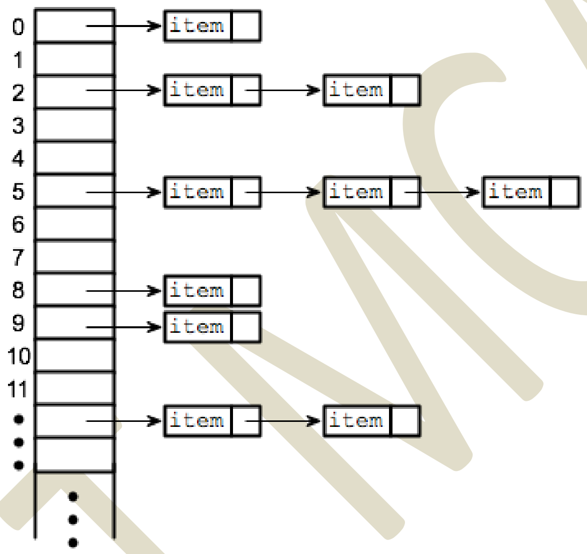
Whenever an element is to be inserted, compute the hash code of the key passed and locate the index using that hash code as an index in the array. Use linear probing for empty location, if an element is found at the computed hash code.

Delete Operation

Whenever an element is to be deleted, compute the hash code of the key passed and locate the index using that hash code as an index in the array. Use linear probing to get the element ahead if an element is not found at the computed hash code. When found, store a dummy item there to keep the performance of the hash table intact.

Separate chaining (open hashing):

Separate chaining is one of the most commonly used collision resolution techniques. It is usually implemented using linked lists. In separate chaining, each element of the hash table is a linked list. To store an element in the hash table you must insert it into a specific linked list. If there is any collision (i.e. two different elements have same hash value) then store both the elements in the same linked list.



The cost of a lookup is that of scanning the entries of the selected linked list for the required key. If the distribution of the keys is sufficiently uniform, then the average cost of a lookup depends only on the average number of keys per linked list. For this reason, chained hash tables remain effective even when the number of table entries (N) is much higher than the number of slots.

For separate chaining, the worst-case scenario is when all the entries are inserted into the same linked list. The lookup procedure may have to scan all its entries, so the worst-case cost is proportional to the number (N) of entries in the table.

Linear probing (open addressing or closed hashing)

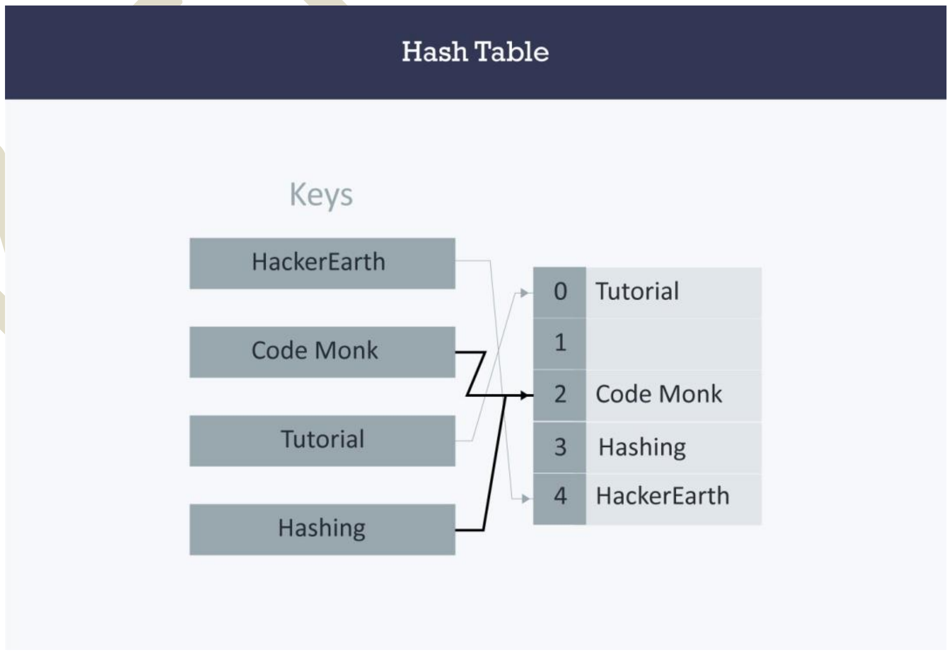
In open addressing, instead of in linked lists, all entry records are stored in the array itself. When a new entry has to be inserted, the hash index of the hashed value is computed and then the array is examined (starting with the hashed index). If the slot at the hashed index is unoccupied, then the entry record is inserted in slot at the hashed index else it proceeds in some probe sequence until it finds an unoccupied slot.

The probe sequence is the sequence that is followed while traversing through entries. In different probe sequences, you can have different intervals between successive entry slots or probes.

When searching for an entry, the array is scanned in the same sequence until either the target element is found or an unused slot is found. This indicates that there is no such key in the table. The name "open addressing" refers to the fact that the location or address of the item is not determined by its hash value.

Linear probing is when the interval between successive probes is fixed (usually to 1). Let's assume that the hashed index for a particular entry is **index**. The probing sequence for linear probing will be:

index = index % hashTableSize
index = (index + 1) % hashTableSize
index = (index + 2) % hashTableSize
index = (index + 3) % hashTableSize
and so on…



Quadratic Probing

Quadratic probing is similar to linear probing and the only difference is the interval between successive probes or entry slots. Here, when the slot at a hashed index for an entry record is already occupied, you must start traversing until you find an unoccupied slot. The interval between slots is computed by adding the successive value of an arbitrary polynomial in the original hashed index.

Let us assume that the hashed index for an entry is **index** and at **index** there is an occupied slot. The probe sequence will be as follows:

index = index % hashTableSize
index = (index + 1^2) % hashTableSize
index = (index + 2^2) % hashTableSize
index = (index + 3^2) % hashTableSize
and so on…

Double hashing

Double hashing is similar to linear probing and the only difference is the interval between successive probes. Here, the interval between probes is computed by using two hash functions.

Let us say that the hashed index for an entry record is an index that is computed by one hashing function and the slot at that index is already occupied. You must start traversing in a specific probing sequence to look for an unoccupied slot. The probing sequence will be:

index = (index + 1 * indexH) % hashTableSize;
index = (index + 2 * indexH) % hashTableSize;
and so on…

Here, **indexH** is the hash value that is computed by another hash function.

Rehashing:

As the name suggests, **rehashing means hashing again**. Basically, when the load factor increases to more than its pre-defined value (default value of load factor is 0.75), the complexity increases. So to overcome this, the size of the array is increased (doubled) and all the values are hashed again and stored in the new double sized array to maintain a low load factor and low complexity.

Why rehashing?

Rehashing is done because whenever key value pairs are inserted into the map, the load factor increases, which implies that the time complexity also increases as explained above. This might not give the required time complexity of O(1).

Hence, rehash must be done, increasing the size of the bucketArray so as to reduce the load factor and the time complexity.

How Rehashing is done?

Rehashing can be done as follows:

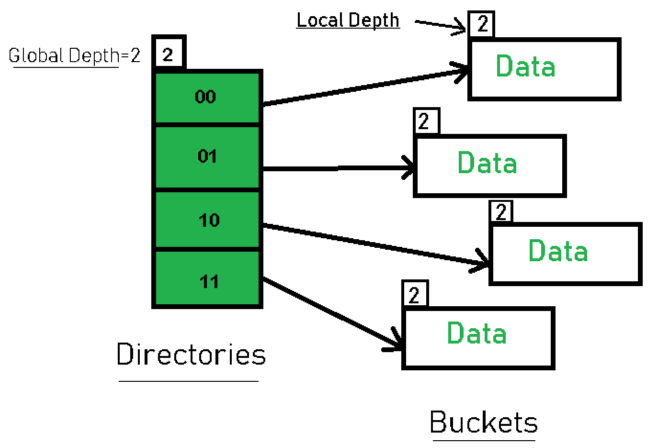
- For each addition of a new entry to the map, check the load factor.
- If it’s greater than its pre-defined value (or default value of 0.75 if not given), then Rehash.
- For Rehash, make a new array of double the previous size and make it the new bucketarray.
- Then traverse to each element in the old bucketArray and call the insert() for each so as to insert it into the new larger bucket array.

Extendible Hashing

It is a dynamic hashing method wherein directories, and buckets are used to hash data. It is an aggressively flexible method in which the hash function also experiences dynamic changes.

Main features of Extendible Hashing: The main features in this hashing technique are:

- **Directories:** The directories store addresses of the buckets in pointers. An id is assigned to each directory which may change each time when Directory Expansion takes place.
- **Buckets:** The buckets are used to hash the actual data.



Extendible Hashing

Frequently used terms in Extendible Hashing:

- **Directories:** These containers store pointers to buckets. Each directory is given a unique id which may change each time when expansion takes place. The hash function returns this directory id which is used to navigate to the appropriate bucket. Number of Directories = 2^Global Depth.
- **Buckets:** They store the hashed keys. Directories point to buckets. A bucket may contain more than one pointers to it if its local depth is less than the global depth.
- **Global Depth:** It is associated with the Directories. They denote the number of bits which are used by the hash function to categorize the keys. Global Depth = Number of bits in directory id.
- **Local Depth:** It is the same as that of Global Depth except for the fact that Local Depth is associated with the buckets and not the directories. Local depth in accordance with the global depth is used to decide the action that to be performed in case an overflow occurs. Local Depth is always less than or equal to the Global Depth.
- **Bucket Splitting:** When the number of elements in a bucket exceeds a particular size, then the bucket is split into two parts.
- **Directory Expansion:** Directory Expansion Takes place when a bucket overflows. Directory Expansion is performed when the local depth of the overflowing bucket is equal to the global depth.

Advantages:

1. Data retrieval is less expensive (in terms of computing).
2. No problem of Data-loss since the storage capacity increases dynamically.
3. With dynamic changes in hashing function, associated old values are rehashed w.r.t the new hash function.

4.

Limitations Of Extendible Hashing:

1. The directory size may increase significantly if several records are hashed on the same directory while keeping the record distribution non-uniform.
2. Size of every bucket is fixed.
3. Memory is wasted in pointers when the global depth and local depth difference becomes drastic.

Applications

- **Associative arrays:** Hash tables are commonly used to implement many types of in-memory tables. They are used to implement associative arrays (arrays whose indices are arbitrary strings or other complicated objects).
- **Database indexing:** Hash tables may also be used as disk-based data structures and database indices (such as in dbm).
- **Caches:** Hash tables can be used to implement caches i.e. auxiliary data tables that are used to speed up the access to data, which is primarily stored in slower media.
- **Object representation:** Several dynamic languages, such as Perl, Python, JavaScript, and Ruby use hash tables to implement objects.
- Hash Functions are used in various algorithms to make their computing faster

UNIT – 5

Sorting Techniques: Insertion sort, selection sort, exchange-bubble sort, quick sort and merge sort Algorithms. **Trees:** Binary Trees, terminology, representation and traversals- pre, post & in order traversals. **Search Trees:** Binary Search Trees, Definition, Implementation, Operations- Searching, Insertion and Deletion
----

Sorting:

Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.
The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats. Following are some of the examples of sorting in real-life scenarios −

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

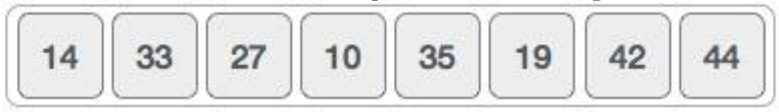
Insertion sort:

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be 'insert'ed in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, insertion sort.

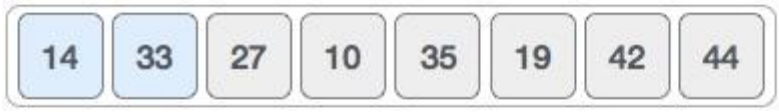
The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$, where n is the number of items.

How Insertion Sort Works?

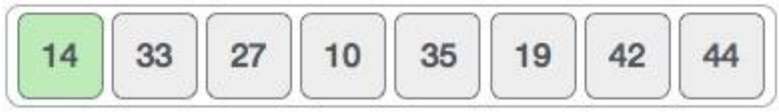
We take an unsorted array for our example.



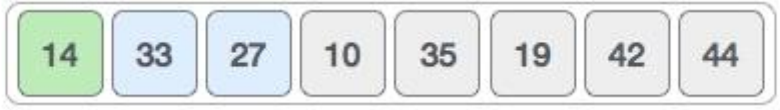
Insertion sort compares the first two elements.



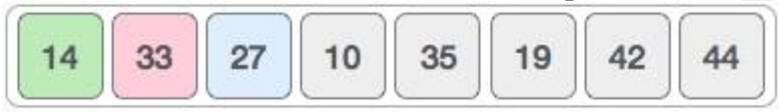
It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.



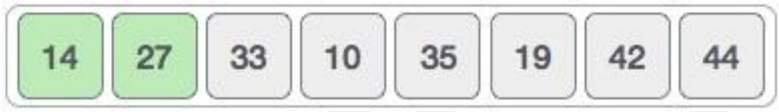
Insertion sort moves ahead and compares 33 with 27.



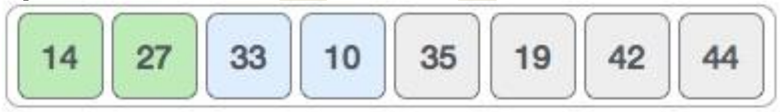
And finds that 33 is not in the correct position.



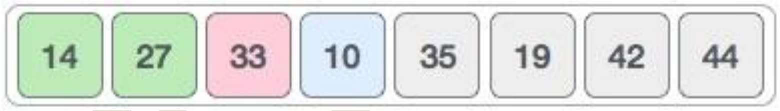
It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.



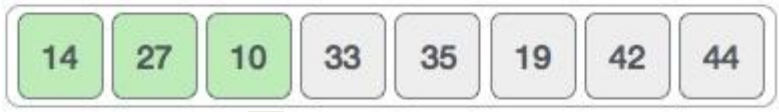
By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.



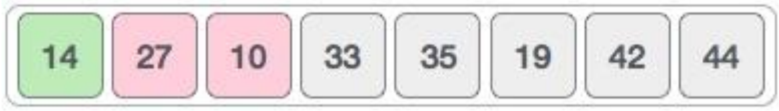
These values are not in a sorted order.



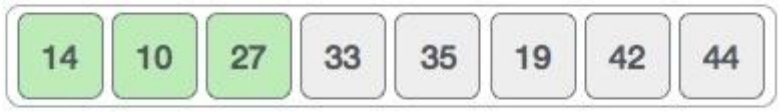
So we swap them.



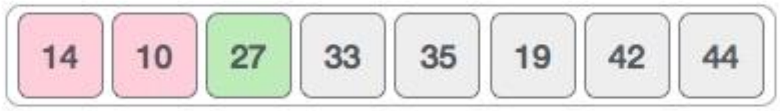
However, swapping makes 27 and 10 unsorted.



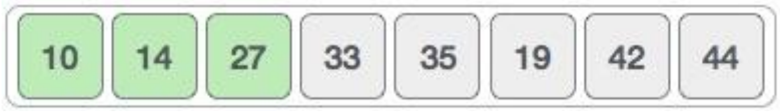
Hence, we swap them too.



Again we find 14 and 10 in an unsorted order.



We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.



This process goes on until all the unsorted values are covered in a sorted sub-list. Now we shall see some programming aspects of insertion sort.

Algorithm

Now we have a bigger picture of how this sorting technique works, so we can derive simple steps by which we can achieve insertion sort.

```
Step 1 − If it is the first element, it is already sorted. return 1;
Step 2 − Pick next element
Step 3 − Compare with all elements in the sorted sub-list
Step 4 − Shift all the elements in the sorted sub-list that is greater than the
         value to be sorted
Step 5 − Insert the value
Step 6 − Repeat until list is sorted
```

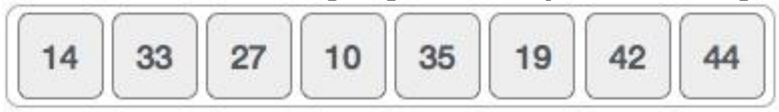
Selection sort:

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

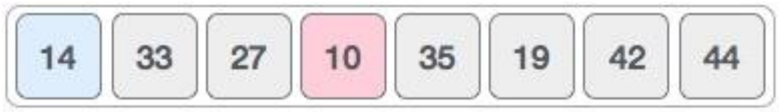
The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right. This algorithm is not suitable for large data sets as its average and worst case complexities are of $O(n^2)$, where **n** is the number of items.

How Selection Sort Works?

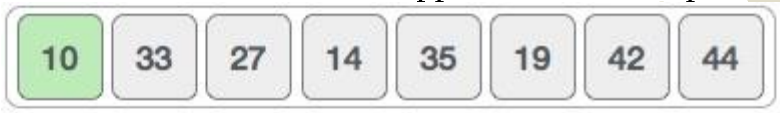
Consider the following depicted array as an example.



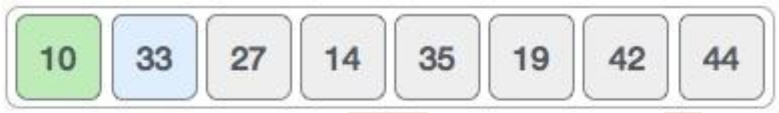
For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.



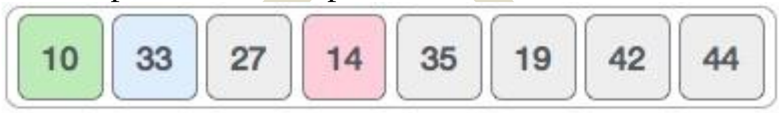
So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.



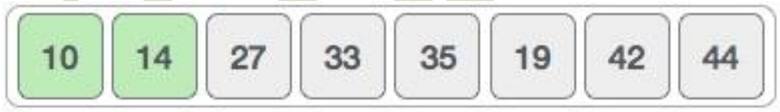
For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.



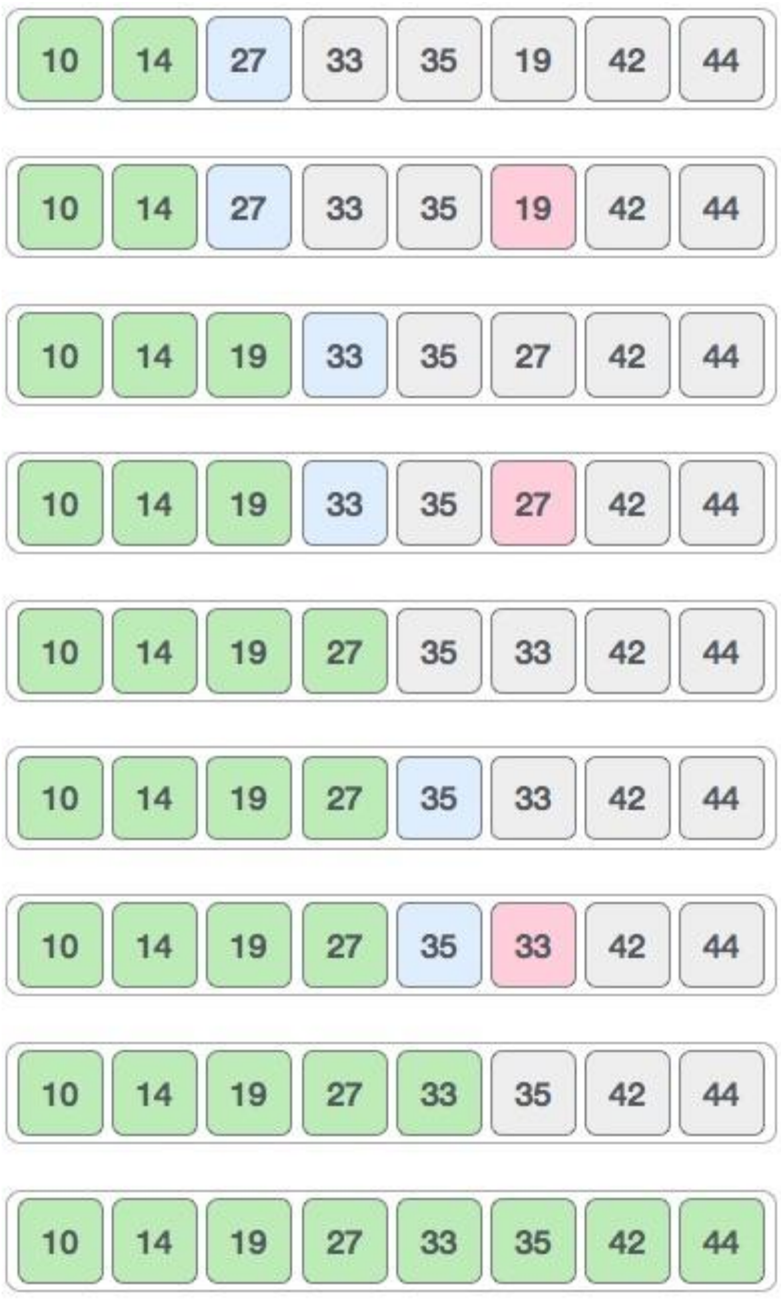
We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.



After two iterations, two least values are positioned at the beginning in a sorted manner.



The same process is applied to the rest of the items in the array.
Following is a pictorial depiction of the entire sorting process −



Now, let us learn some programming aspects of selection sort.

<u>Algorithm</u>

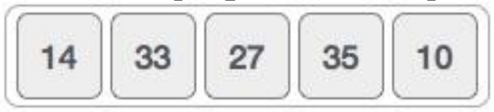
Step 1 − Set MIN to location 0
Step 2 − Search the minimum element in the list
Step 3 − Swap with value at location MIN
Step 4 − Increment MIN to point to next element
Step 5 − Repeat until list is sorted

Bubble sort:

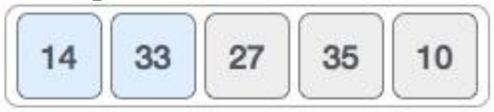
Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of O(n^2) where **n** is the number of items.

How Bubble Sort Works?

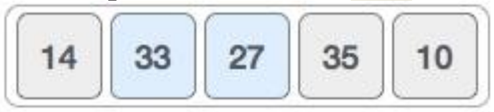
We take an unsorted array for our example. Bubble sort takes O(n^2) time so we're keeping it short and precise.



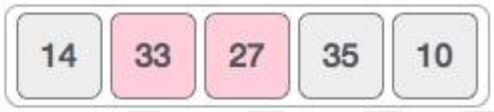
Bubble sort starts with very first two elements, comparing them to check which one is greater.



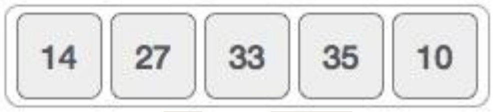
In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.



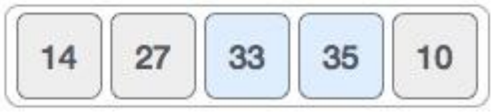
We find that 27 is smaller than 33 and these two values must be swapped.



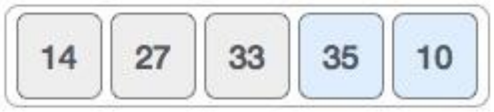
The new array should look like this −



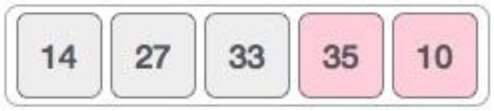
Next we compare 33 and 35. We find that both are in already sorted positions.



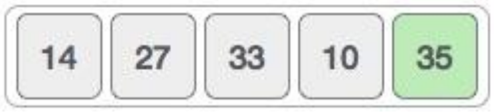
Then we move to the next two values, 35 and 10.



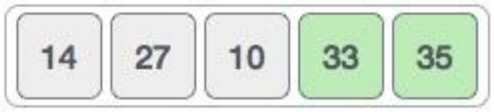
We know then that 10 is smaller 35. Hence they are not sorted.



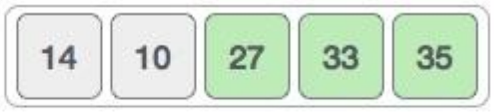
We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −



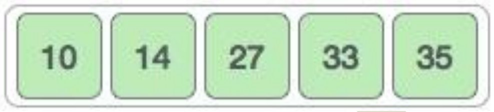
To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this −



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



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



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

Algorithm

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

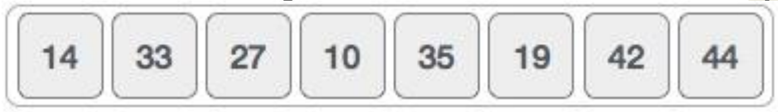
```
begin BubbleSort(list)

   for all elements of list
      if list[i] > list[i+1]
         swap(list[i], list[i+1])
      end if
   end for

   return list

end BubbleSort
```

Merge sort

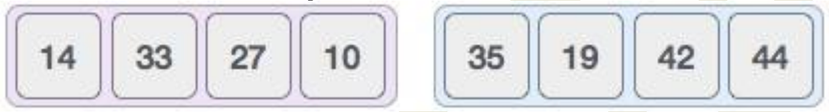
Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being O(n log n), it is one of the most respected algorithms.
Merge sort first divides the array into equal halves and then combines them in a sorted manner.

How Merge Sort Works?

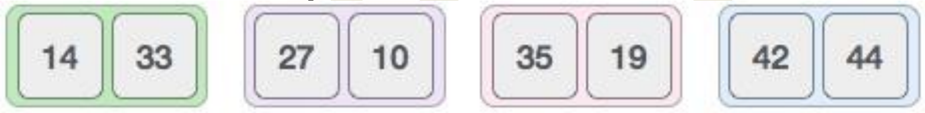
To understand merge sort, we take an unsorted array as the following −



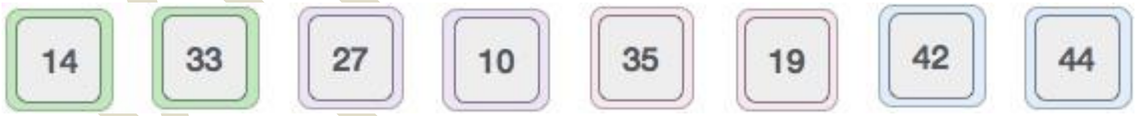
We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.



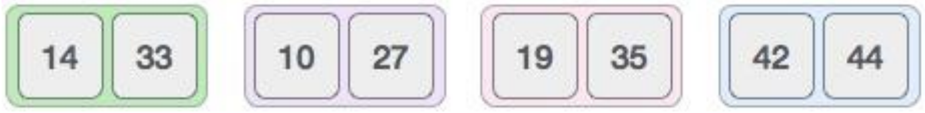
This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.



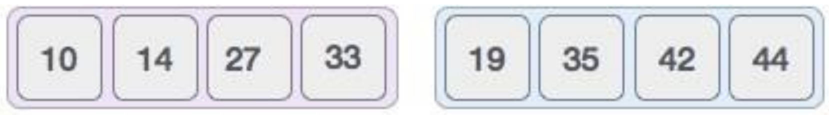
We further divide these arrays and we achieve atomic value which can no more be divided.



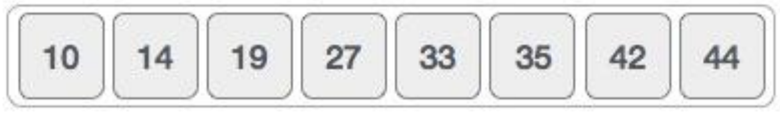
Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.
We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed sequentially.



In the next iteration of the combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in a sorted order.



After the final merging, the list should look like this −



Now we should learn some programming aspects of merge sorting.

Algorithm

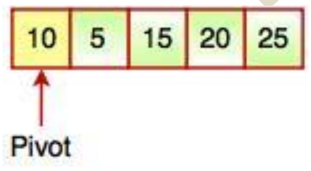
Merge sort keeps on dividing the list into equal halves until it can no more be divided. By definition, if it is only one element in the list, it is sorted. Then, merge sort combines the smaller sorted lists keeping the new list sorted too.
Step 1 − if it is only one element in the list it is already sorted, return.
Step 2 − divide the list recursively into two halves until it can no more be divided.
Step 3 − merge the smaller lists into new list in sorted order

Quick Sort

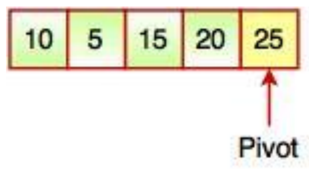
- Quick sort is also known as Partition-exchange sort based on the rule of Divide and Conquer.
- It is a highly efficient sorting algorithm.
- Quick sort is the quickest comparison-based sorting algorithm.
- It is very fast and requires less additional space, only O(n log n) space is required.
- Quick sort picks an element as pivot and partitions the array around the picked pivot.

There are different versions of quick sort which choose the pivot in different ways:

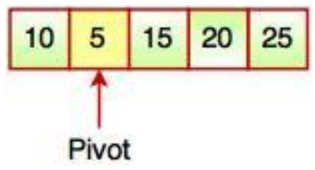
1. First element as pivot



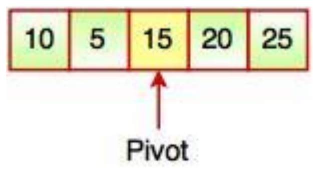
2. Last element as pivot



3. Random element as pivot



4. Median as pivot



<u>Algorithm for Quick Sort:</u>

Step 1: Choose the highest index value as pivot.

Step 2: Take two variables to point left and right of the list excluding pivot.

Step 3: Left points to the low index.

Step 4: Right points to the high index.

Step 5: While value at left < (Less than) pivot move right.

Step 6: While value at right > (Greater than) pivot move left.

Step 7: If both Step 5 and Step 6 does not match, swap left and right.

Step 8: If left = (Less than or Equal to) right, the point where they met is new pivot.

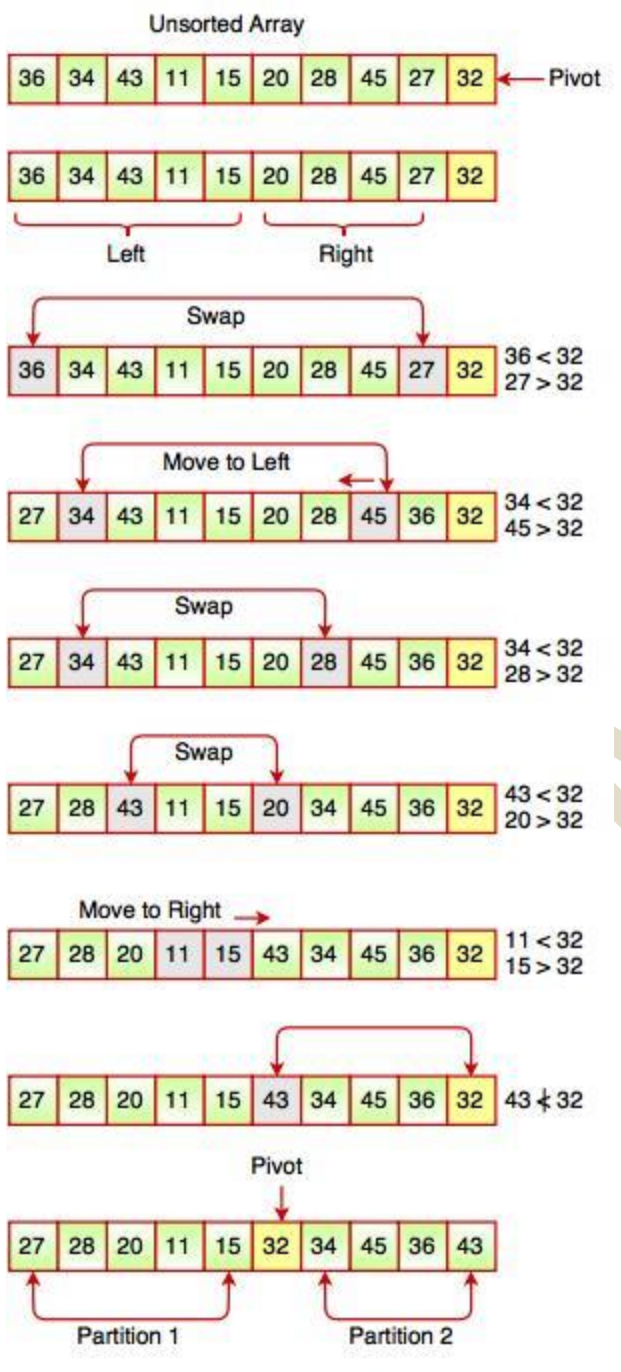
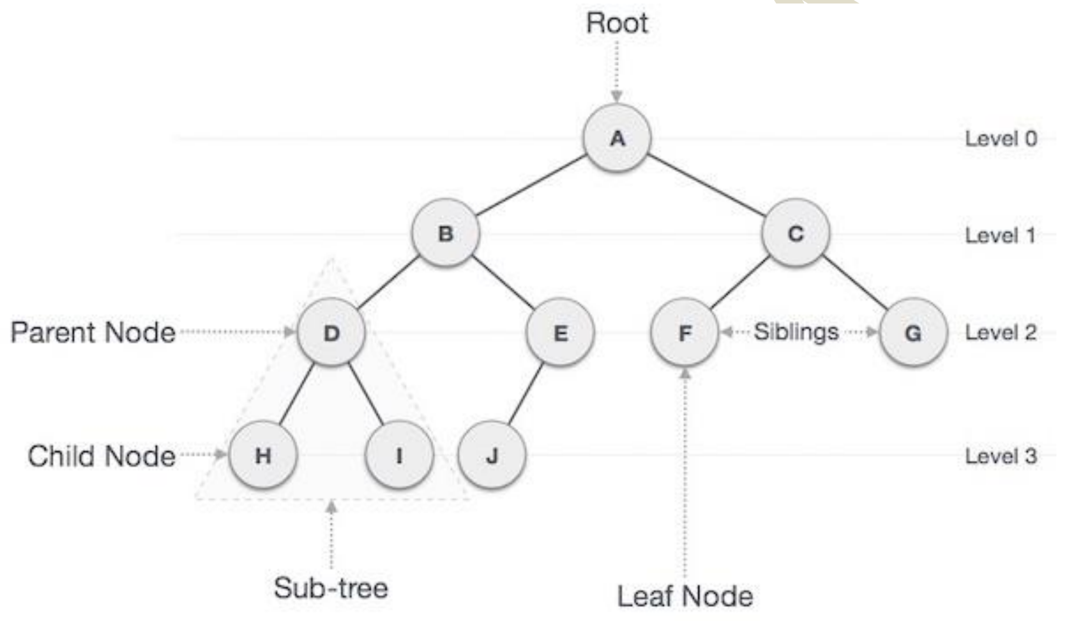


Fig. Finding Pivot Value in an Array

The above diagram represents how to find the pivot value in an array. As we see, pivot value divides the list into two parts (partitions) and then each part is processed for quick sort. Quick sort is a recursive function. We can call the partition function again.

Trees:

Tree represents the nodes connected by edges. We will discuss binary tree or binary search tree specifically.
Binary Tree is a special datastructure used for data storage purposes. A binary tree has a special condition that each node can have a maximum of two children. A binary tree has the benefits of both an ordered array and a linked list as search is as quick as in a sorted array and insertion or deletion operation are as fast as in linked list.



Important Terms

Following are the important terms with respect to tree.

- **Path** − Path refers to the sequence of nodes along the edges of a tree.
- **Root** − The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
- **Parent** − Any node except the root node has one edge upward to a node called parent.
- **Child** − The node below a given node connected by its edge downward is called its child node.
- **Leaf** − The node which does not have any child node is called the leaf node.
- **Subtree** − Subtree represents the descendants of a node.
- **Visiting** − Visiting refers to checking the value of a node when control is on the node.
- **Traversing** − Traversing means passing through nodes in a specific order.

- **Levels** − Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.
- **keys** − Key represents a value of a node based on which a search operation is to be carried out for a node.

Tree Traversal:

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

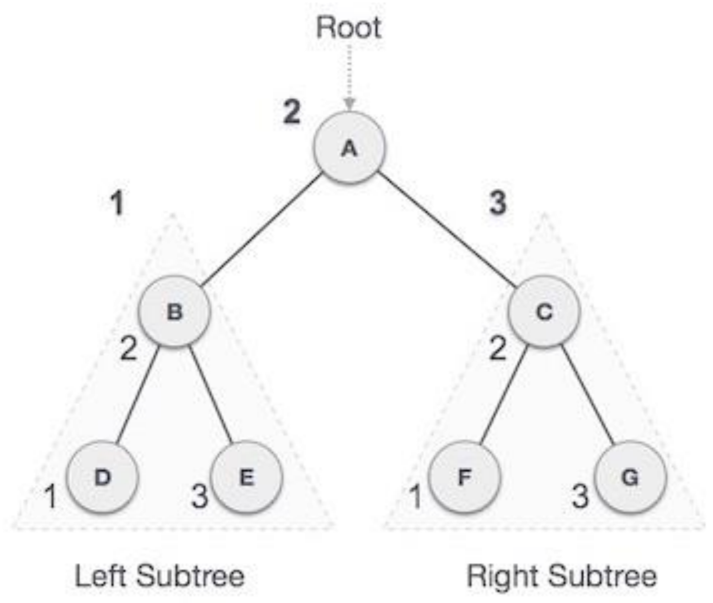
- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

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

In-order Traversal

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

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



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

D → B → E → A → F → C → G

<u>Algorithm</u>

Until all nodes are traversed −

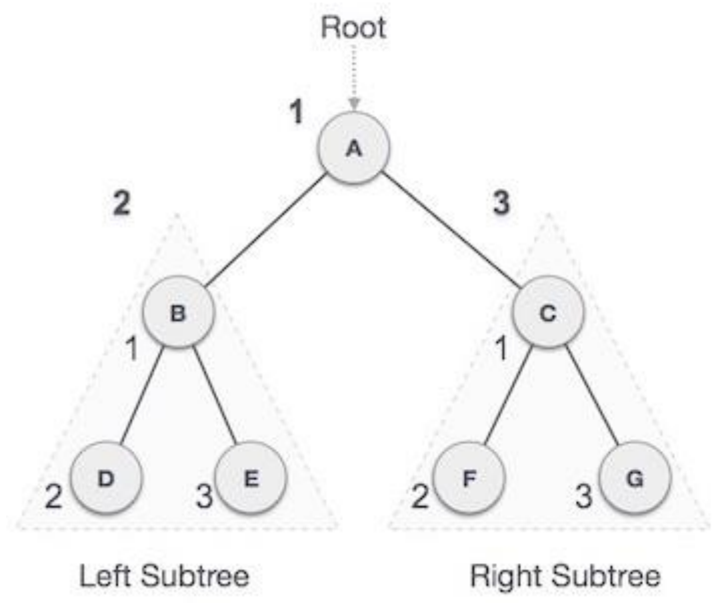
Step 1 − Recursively traverse left subtree.

Step 2 − Visit root node.

Step 3 − Recursively traverse right subtree.

<u>Pre-order Traversal</u>

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



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

A → B → D → E → C → F → G

<u>Algorithm</u>

Until all nodes are traversed −

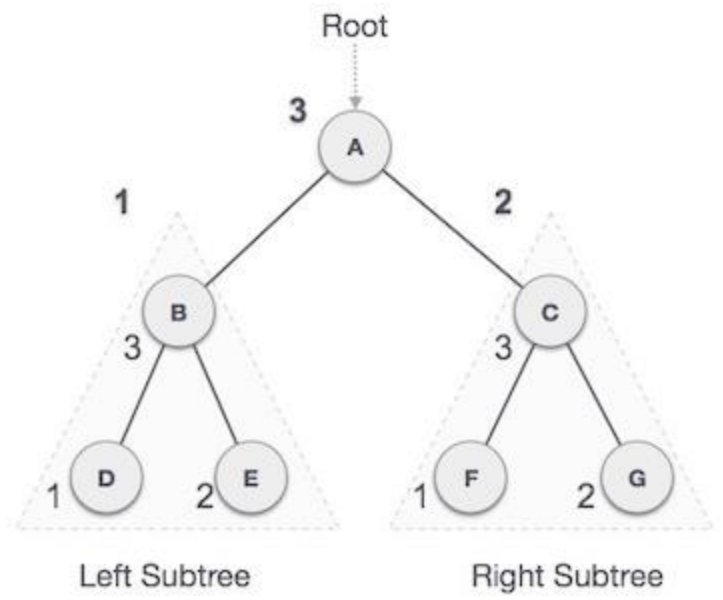
Step 1 − Visit root node.

Step 2 − Recursively traverse left subtree.

Step 3 − Recursively traverse right subtree.

<u>Post-order Traversal</u>

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



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

D → E → B → F → G → C → A

Algorithm

Until all nodes are traversed −

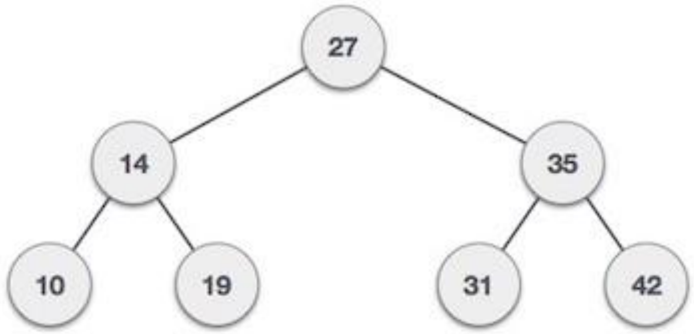
Step 1 − Recursively traverse left subtree.

Step 2 − Recursively traverse right subtree.

Step 3 − Visit root node.

Binary Search Tree Representation

Binary Search tree exhibits a special behavior. A node's left child must have a value less than its parent's value and the node's right child must have a value greater than its parent value.



We're going to implement tree using node object and connecting them through references.

Tree Node

The code to write a tree node would be similar to what is given below. It has a data part and references to its left and right child nodes.

```
struct node {
   int data;
   struct node *leftChild;
   struct node *rightChild;
};
```

In a tree, all nodes share common construct.

BST Basic Operations

The basic operations that can be performed on a binary search tree data structure, are the following −

- **Insert** − Inserts an element in a tree/create a tree.
- **Search** − Searches an element in a tree.
- **Preorder Traversal** − Traverses a tree in a pre-order manner.
- **Inorder Traversal** − Traverses a tree in an in-order manner.
- **Postorder Traversal** − Traverses a tree in a post-order manner.

Insert Operation

The very first insertion creates the tree. Afterwards, whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.

Algorithm

```
If root is NULL
  then create root node
return

If root exists then
  compare the data with node.data

  while until insertion position is located

    If data is greater than node.data
      goto right subtree
    else
      goto left subtree

  endwhile

  insert data

end If
```

Search Operation

Whenever an element is to be searched, start searching from the root node, then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree. Follow the same algorithm for each node.

Algorithm

```
If root.data is equal to search.data
  return root
else
  while data not found

    If data is greater than node.data
      goto right subtree
    else
```

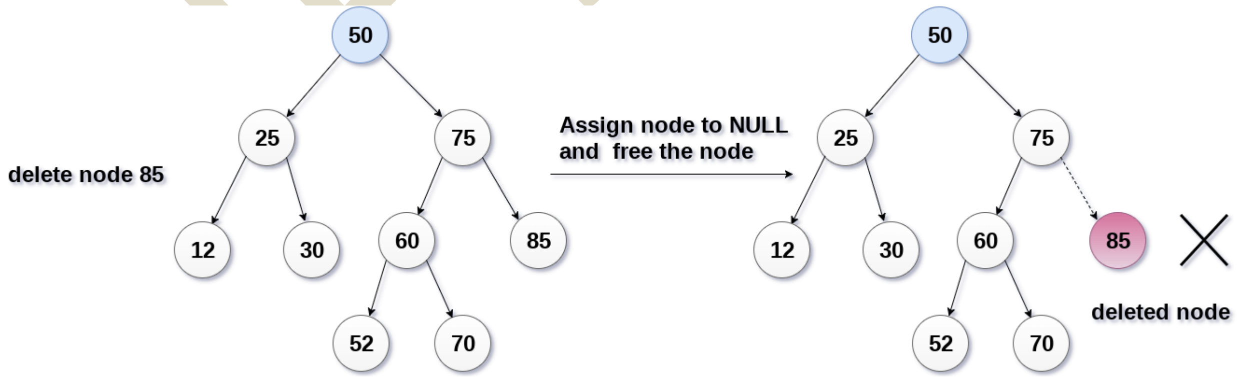
```
        goto left subtree

     If data found
        return node
  endwhile

  return data not found

end if
```

Deletion

Delete function is used to delete the specified node from a binary search tree. However, we must delete a node from a binary search tree in such a way, that the property of binary search tree doesn't violate. There are three situations of deleting a node from binary search tree.

The node to be deleted is a leaf node.It is the simplest case, in this case, replace the leaf node with the NULL and simple free the allocated space.

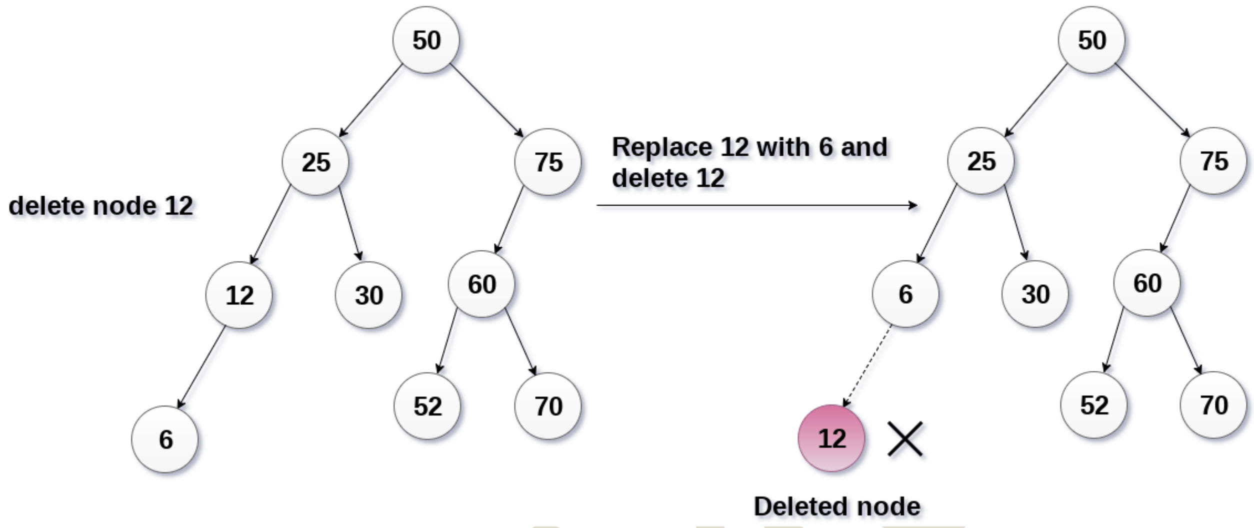
In the following image, we are deleting the node 85, since the node is a leaf node, therefore the node will be replaced with NULL and allocated space will be freed.



The node to be deleted has only one child.

In this case, replace the node with its child and delete the child node, which now contains the value which is to be deleted. Simply replace it with the NULL and free the allocated space.

In the following image, the node 12 is to be deleted. It has only one child. The node will be replaced with its child node and the replaced node 12 (which is now leaf node) will simply be deleted.



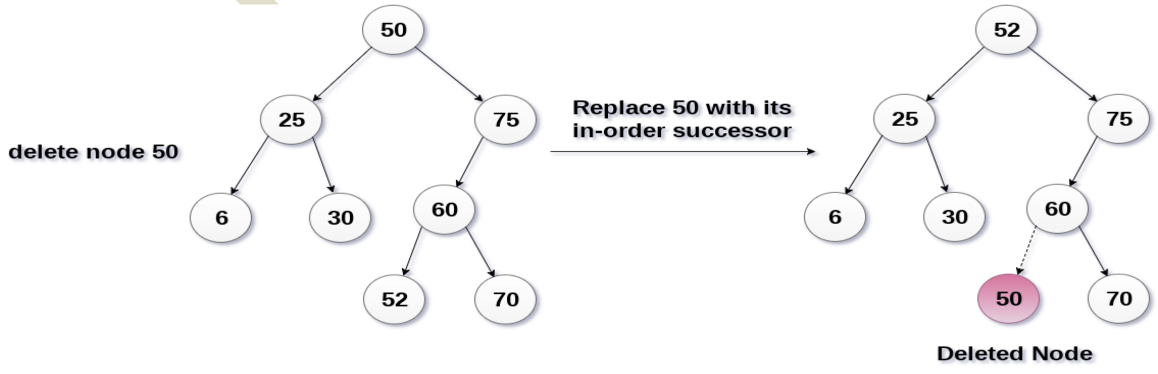
The node to be deleted has two children.

It is a bit complexed case compare to other two cases. However, the node which is to be deleted, is replaced with its in-order successor or predecessor recursively until the node value (to be deleted) is placed on the leaf of the tree. After the procedure, replace the node with NULL and free the allocated space.

In the following image, the node 50 is to be deleted which is the root node of the tree. The in-order traversal of the tree given below.

6, 25, 30, 50, 52, 60, 70, 75.

replace 50 with its in-order successor 52. Now, 50 will be moved to the leaf of the tree, which will simply be deleted.



<u>Algorithm</u>

Delete (TREE, ITEM)

```
○ Step 1: IF TREE = NULL
      Write "item not found in the tree" ELSE IF ITEM < TREE -> DATA
     Delete(TREE->LEFT, ITEM)
     ELSE IF ITEM > TREE -> DATA
      Delete(TREE -> RIGHT, ITEM)
     ELSE IF TREE -> LEFT AND TREE -> RIGHT
     SET TEMP = findLargestNode(TREE -> LEFT)
     SET TREE -> DATA = TEMP -> DATA
      Delete(TREE -> LEFT, TEMP -> DATA)
     ELSE
      SET TEMP = TREE
      IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL
      SET TREE = NULL
     ELSE IF TREE -> LEFT != NULL
     SET TREE = TREE -> LEFT
     ELSE
       SET TREE = TREE -> RIGHT
     [END OF IF]
     FREE TEMP
   [END OF IF]
○ Step 2: END
```