

MODULE 1

Introduction to programming methodologies

Programming methodology deals with the analysis, design and implementation of programs.

Algorithm

Algorithm is a step-by-step finite sequence of instruction, to solve a well-defined computational problem.

That is, in practice to solve any complex real life problems; first we have to define the problems. Second step is to design the algorithm to solve that problem.

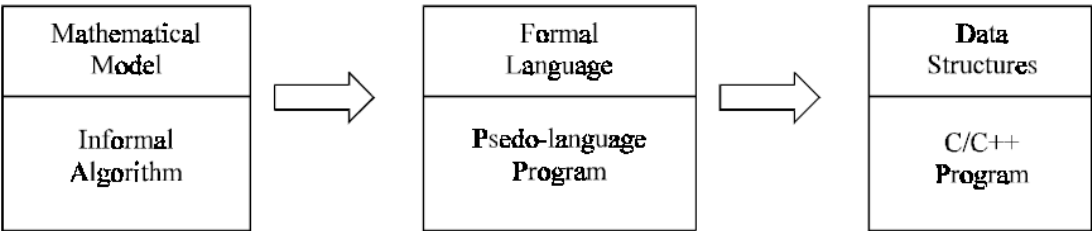
Writing and executing programs and then optimizing them may be effective for small programs. Optimization of a program is directly concerned with algorithm design. But for a large program, each part of the program must be well organized before writing the program. There are few steps of refinement involved when a problem is converted to program; this method is called **stepwise refinement method**. There are two approaches for algorithm design; they are **top-down** and **bottom-up** algorithm design.

Stepwise Refinement Techniques

We can write an informal algorithm, if we have an appropriate mathematical model for a problem. The initial version of the algorithm will contain general statements, *i.e.*, informal instructions. Then we convert this informal algorithm to formal algorithm, that is, more definite instructions by applying any programming language syntax and semantics partially. Finally a program can be developed by converting the formal algorithm by a programming language manual. From the above discussion we have understood that there are several steps to reach a program from a mathematical model. In every step there is a refinement (or conversion).

That is to convert an informal algorithm to a program, we must go through several stages of formalization until we arrive at a program — whose meaning is formally defined by a programming language manual — is called stepwise refinement techniques.

There are three steps in refinement process, which is illustrated in Figure



1. In the first stage, modeling, we try to represent the problem using an appropriate mathematical model such as a graph, tree etc. At this stage, the solution to the problem is an algorithm expressed very informally.
2. At the next stage, the algorithm is written in pseudo-language (or formal algorithm) that is, a mixture of any programming language constructs and less formal English statements. The operations to be performed on the various types of data become fixed.
3. In the final stage we choose an implementation for each abstract data type and write the procedures for the various operations on that type. The remaining informal statements in the pseudo-language algorithm are replaced by (or any programming language) C/C++ code.

Programming style

Following sections will discuss different programming methodologies to design a program.

1. Procedural
2. Modular
3. Structured
4. Object oriented

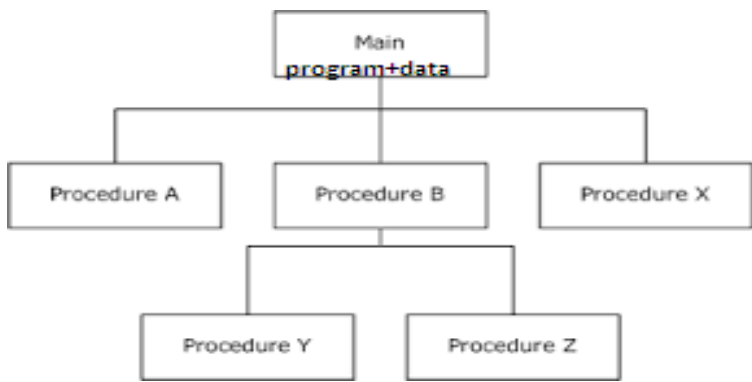
1.Procedural Programming

Procedural programming is a paradigm based on the concept of using procedures. Procedure (sometimes also called subprogram, routine or method) is a sequence of commands to be executed. Any procedure can be called from any point within the general program, including other procedures or even itself (resulting in a recursion).

Procedural programming is widely used in large-scale projects, when the following benefits are important:

- re-usability of pieces code designed as procedures
- ease of following the logic of program;
- Maintainability of code.
- Emphasis is on doing things (algorithms).
- Most of the functions share global data.
- Data move openly around the system from function to function.
- Functions transform data from one form to another.

Procedural programming is a sub-paradigm of imperative programming, since each step of computation is described explicitly, even if by the means of defining procedures.



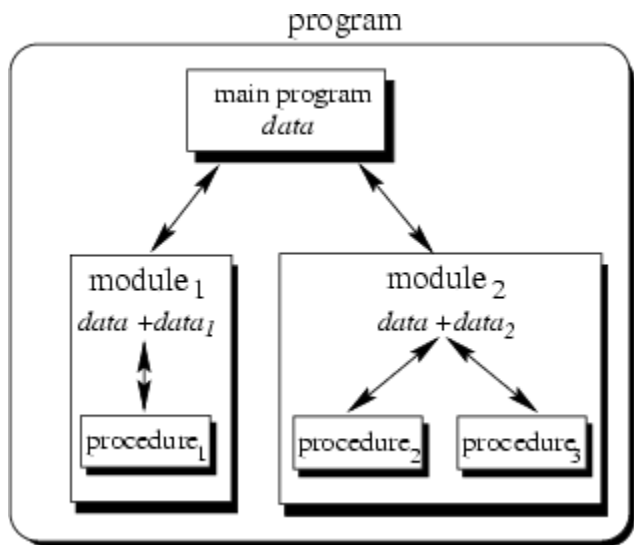
2.Modular Programming

The program is progressively decomposed into smaller partition called modules. The program can easily be written in modular form, thus allowing an overall problem to be decomposed into a sequence of individual sub programs. Thus we can consider, a module decomposed into successively subordinate module. Conversely, a number of modules can combined together to form a superior module.

A sub-module, are located elsewhere in the program and the superior module, whenever necessary make a reference to subordinate module and call for its execution. This activity on part of the superior module is known as a calling, and this module is referred as calling module, and sub module referred as called module. The sub module may be subprograms such as function or procedures.

The following are the steps needed to develop a modular program

- 1. Define the problem
- 2. Outline the steps needed to achieve the goal
- 3. Decompose the problem into subtasks
- 4. Prototype a subprogram for each sub task
- 5. Repeat step 3 and 4 for each subprogram until further decomposition seems counter productive



Modular Programming is heavily procedural. The focus is entirely on writing code (functions). Data is passive in Modular Programming. Modular Programming discourages the use of control variables and flags in parameters; their presence tends to indicate that the caller needs to know too much about how the function is implemented.

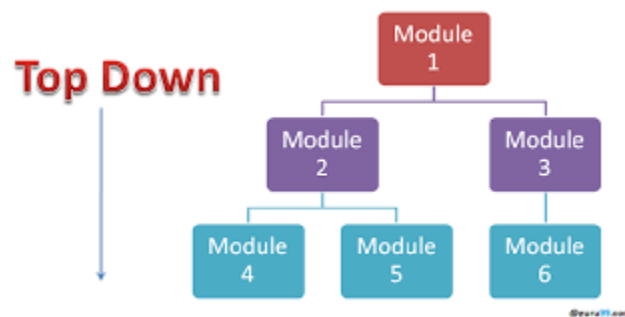
Two methods may be used for modular programming. They are known as **top-down and bottom-up**. Regardless of whether the top-down or bottom-up method is used, the end result is a modular program. This end result is important, because not all errors may be detected at the time of the initial testing. It is possible that there are still bugs in the program. If an error is discovered after the program supposedly has been fully tested, then the modules concerned can be isolated and retested by them. Regardless of the design method used, if a program has been written in modular form, it is easier to detect the source of the error and to test it in isolation, than if the program were written as one function.

Advantages of modular programming

1. Reduce the complexity of the entire problem
2. Avoid the duplication of code
3. debugging program is easier and reliable
4. Improves the performance
5. Modular program hides the use of data structure
6. Global data also hidden in module
7. Reusability- modules can be used in other program without rewriting and retesting
8. Modular program improves the portability of program
9. It reduces the development work

Top- down modular programming

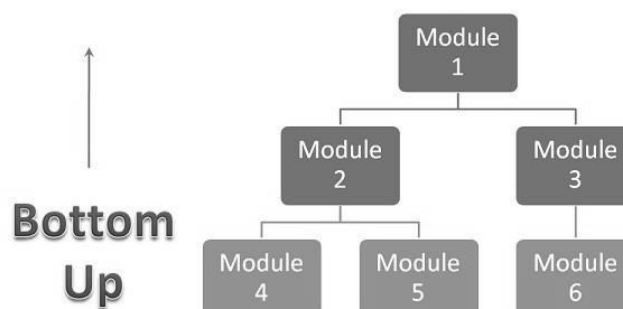
The principles of top-down design dictate that a program should be divided into a main module and its related modules. Each module should also be divided into sub modules according to software engineering and programming style. The division of modules processes until the module consists only of elementary process that are intrinsically understood and cannot be further subdivided.



Top-down algorithm design is a technique for organizing and coding programs in which a hierarchy of modules is used, and breaking the specification down into simpler and simpler pieces, each having a single entry and a single exit point, and in which control is passed downward through the structure without unconditional branches to higher levels of the structure. That is top-down programming tends to generate modules that are based on functionality, usually in the form of functions or procedures or methods.

Bottom-Up modular programming

Bottom-up algorithm design is the opposite of top-down design. It refers to a style of programming where an application is constructed starting with existing primitives of the programming language, and constructing gradually more and more complicated features, until the all of the application has been written. That is, starting the design with specific modules and build them into more complex structures, ending at the top. The bottom-up method is widely used for testing, because each of the lowest-level functions is written and tested first. This testing is done by special test functions that call the low-level functions, providing them with different parameters and examining the results for correctness. Once lowest-level functions have been tested and verified to be correct, the next level of functions may be tested. Since the lowest-level functions already have been tested, any detected errors are probably due to the higher-level functions. This process continues, moving up the levels, until finally the *main* function is tested.



3. Structured Programming

It is a programming style; and this style of programming is known by several names: Procedural decomposition, Structured programming, etc. Structured programming is not programming with structures but by using following types of code structures to write programs:

1. Sequence of sequentially executed statements
2. Conditional execution of statements (*i.e.*, “if” statements)
3. Looping or iteration (*i.e.*, “for, do...while, and while” statements)
4. Structured subroutine calls (*i.e.*, functions)

In particular, the following language usage is forbidden:

- “GoTo” statements
- “Break” or “continue” out of the middle of loops
- Multiple exit points to a function/procedure/subroutine (*i.e.*, multiple “return” statements)
- Multiple entry points to a function/procedure/subroutine/method

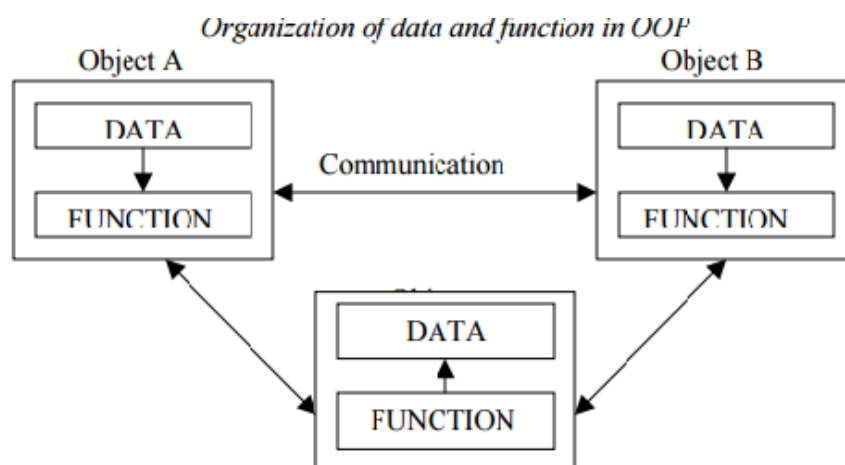
In this style of programming there is a great risk that implementation details of many data structures have to be shared between functions, and thus globally exposed. This in turn tempts other functions to use these implementation details; thereby creating unwanted dependencies between different parts of the program. The main disadvantage is that all decisions made from the start of the project depend directly or indirectly on the high-level specification of the application. It is a well known fact that this specification tends to change over a time. When that happens, there is a great risk that large parts of the application need to be rewritten.

Advantages of structured programming

1. clarity: structured programming has a clarity and logical pattern to their control structure and due to this tremendous increase in programming productivity
2. another key to structured programming is that each block of code has a single entry point and single exit point. so we can break up long sequence of code into modules
3. Maintenance: the clarity and modularity inherent in structured programming is of great help in finding an error and redesigning the required section of code.

4.Object oriented programming

The major motivating factor in the invention of object-oriented approach is to remove some of the flaws encountered in the procedural or modular approach. OOP treats data as a critical element in the program development and does not allow it to flow freely around the system. It ties data more closely to the function that operate on it, and protects it from accidental modification from outside function. OOP allows decomposition of a problem into a number of entities called objects and then builds data and function around these objects. The organization of data and function in object-oriented programs is shown in fig. The data of an object can be accessed only by the function associated with that object. However, function of one object can access the function of other objects.



Some of the features of object oriented programming are:

- Emphasis is on data rather than procedure.
- Programs are divided into what are known as objects.
- Data structures are designed such that they characterize the objects.
- Functions that operate on the data of an object are tied together in the data structure.
- Data is hidden and cannot be accessed by external function.
- Objects may communicate with each other through function.
- New data and functions can be easily added whenever necessary.
- Follows bottom up approach in program design.

Analysis of Algorithm

After designing an algorithm, it has to be checked and its correctness needs to be predicted; this is done by analyzing the algorithm. The algorithm can be analyzed by tracing all step-by-step instructions, reading the algorithm for logical correctness, and testing it on some data using mathematical techniques to prove it correct. Another type of analysis is to analyze the simplicity of the algorithm. That is, design the algorithm in a simple way so that it becomes easier to be implemented. However, the simplest and most Straight forward way of solving a problem may not be sometimes the best one. Moreover there may be more than one algorithm to solve a problem. The choice of a particular algorithm depends on following performance analysis and measurements:

1. Space complexity
2. Time complexity

Space Complexity

Analysis of space complexity of an algorithm or program is the amount of memory it needs to run to completion. Some of the reasons for studying space complexity are:

1. If the program is to run on multi user system, it may be required to specify the amount of memory to be allocated to the program.
2. We may be interested to know in advance that whether sufficient memory is available to run the program.
3. There may be several possible solutions with different space requirements.
4. Can be used to estimate the size of the largest problem that a program can solve.

The space needed by a program consists of following components.

- *Instruction space* : Space needed to store the executable version of the program and it is fixed.
- *Data space* : Space needed to store all constants, variable values and has further two components :
 - (a) Space needed by constants and simple variables. This space is fixed.
 - (b) Space needed by fixed sized structural variables, such as arrays and structures.
 - (c) Dynamically allocated space. This space usually varies.

• **Environment stack space:** This space is needed to store the information to resume the suspended (partially completed) functions. Each time a function is invoked the following data is saved on the environment stack :

- (a) Return address : *i.e.*, from where it has to resume after completion of the Called function.
- (b) Values of all local variables and the values of formal parameters in the function being invoked.

The amount of space needed by recursive function is called the recursion stack space. For each recursive function, this space depends on the space needed by the local variables and the formal parameter. In addition, this space depends on the maximum depth of the recursion *i.e.*, maximum number of nested recursive calls.

Time Complexity

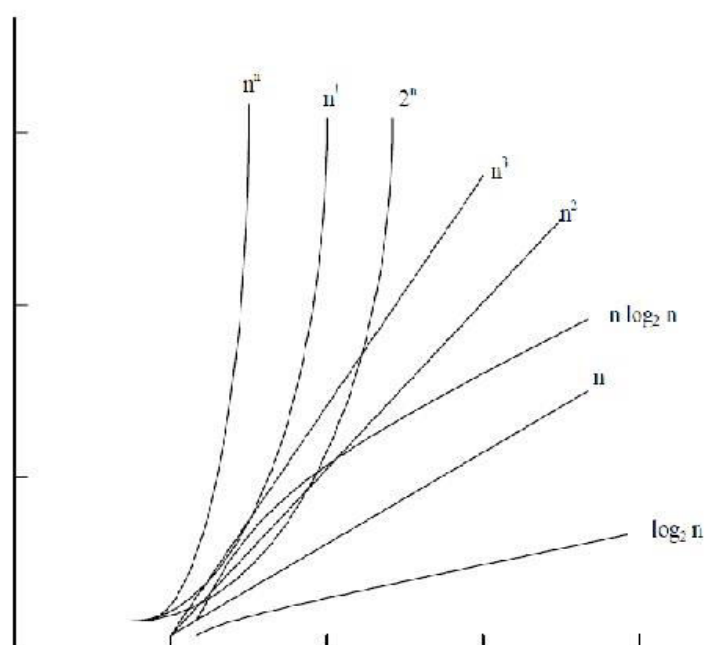
The time complexity of an algorithm or a program is the amount of time it needs to run to completion. The exact time will depend on the implementation of the algorithm, programming language, optimizing the capabilities of the compiler used, the CPU speed, other hardware characteristics/specifications and so on. To measure the time complexity accurately, we have to count all sorts of operations performed in an algorithm. If we know the time for each one of the primitive operations performed in a given computer, we can easily compute the time taken by an algorithm to complete its execution. This time will vary from machine to machine. By analyzing an algorithm, it is hard to come out with an exact time required. To find out exact time complexity, we need to know the exact instructions executed by the hardware and the time required for the instruction. The time complexity also depends on the amount of data inputted to an algorithm. But we can calculate the order of magnitude for the time required.

The time complexity also depends on the amount of data input to an algorithm, but we can calculate the order of magnitude for the time required. That is, our intention is to estimate the execution time of an algorithm irrespective of the computer machine on which it will be used.

Some of the reasons for studying time complexity are

- a) We may be interested to know in advance that whether an algorithm or program will provide a satisfactory real time response
- b) There may be several possible solutions with different time requirements

Here, the more sophisticated method is to identify the key operations and count such operations performed till the program completes its execution. A key operation in our algorithm is an operation that takes maximum time among all possible operations in the algorithm. Such an abstract, theoretical approach is not only useful for discussing and comparing algorithms, but also it is useful to improve solutions to practical problems. The time complexity can now be expressed as function of number of key operations performed. Before we go ahead with our discussions, it is important to understand the rate growth analysis of an algorithm, as shown in Figure.



The function that involves 'n' as an exponent, i.e., 2^n , n^n , $n!$ are called exponential functions, which is too slow except for small size input function where growth is less than or equal to n^c , (where 'c' is a constant) i.e.; n^3 , n^2 , $n \log_2 n$, n , $\log_2 n$ are said to be polynomial. Algorithms with polynomial time can solve reasonable sized problems if the constant in the exponent is small.

When we analyze an algorithm it depends on the input data, there are three cases :

1. Best case
2. Average case
3. Worst case

In the best case, the amount of time a program might be expected to take on best possible input data.

In the average case, the amount of time a program might be expected to take on typical (or average) input data.

In the worst case, the amount of time a program would take on the worst possible input configuration.

Frequency Count

Frequency count method can be used to analyze a program .Here we assume that every statement takes the same constant amount of time for its execution. Hence the determination of time complexity of a given program is is the just matter of summing the frequency counts of all the statements of that program Consider the following examples

.....	for(i=0;I,n;i++)	for(i=0;i<n;i++)
.....	X++;	for(j=0;j<n;j++)
X++;	x++;
(a)	(b)	(c)

In the example (a) the statement x++ is not contained within any loop either explicit or implicit. Then its frequency count is just one. In example (b) same element will be executed n times and in example (3) it is executed by n². From this frequency count we can analyze program

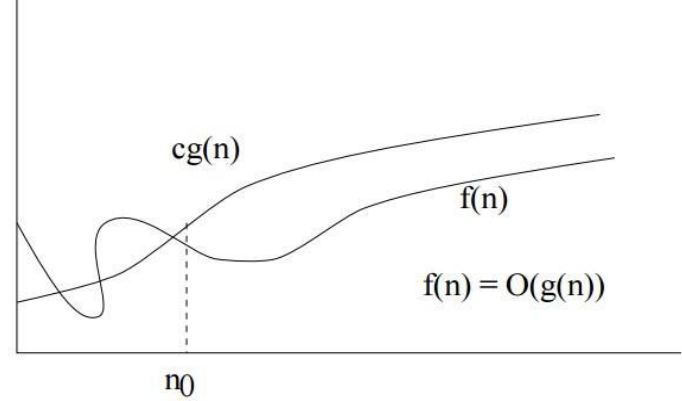
Growth of Functions and Asymptotic Notation

When we study algorithms, we are interested in characterizing them according to their efficiency. We are usually interesting in the order of growth of the running time of an algorithm, not in the exact running time. This is also referred to as the asymptotic running time. We need to develop a way to talk about rate of growth of functions so that we can compare algorithms. Asymptotic notation gives us a method for classifying functions according to their rate of growth.

Big-O Notation

Definition:

$f(n) = O(g(n))$ iff there are two positive constants c and n_0 such that $|f(n)| \leq c |g(n)|$ for all $n \geq n_0$. If $f(n)$ is nonnegative, we can simplify the last condition to $0 \leq f(n) \leq c g(n)$ for all $n \geq n_0$. then we say that “**f(n) is big-O of g(n).**” . As n increases, $f(n)$ grows n_0 faster than $g(n)$. In other words, $g(n)$ is an asymptotic upper bound on $f(n)$.



Example: $n^2 + n = O(n^3)$

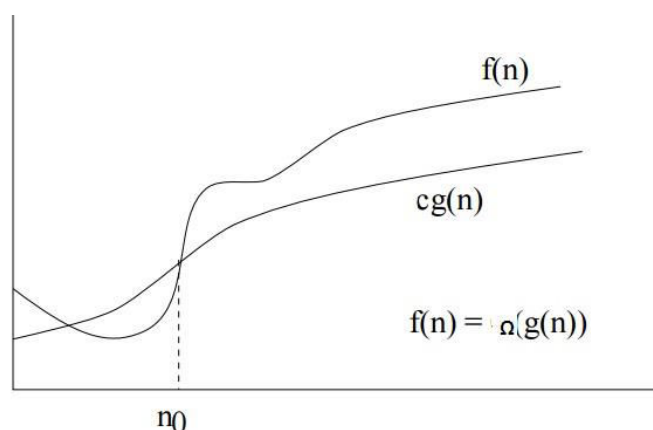
Proof: • Here, we have $f(n) = n^2 + n$, and $g(n) = n^3$

- Notice that if $n \geq 1$, $n \leq n^3$ is clear.
- Also, notice that if $n \geq 1$, $n^2 \leq n^3$ is clear.
- In general, if $a \leq b$, then $n^a \leq n^b$ whenever $n \geq 1$. This fact is used often in these types of proofs.
- Therefore, $n^2 + n \leq n^3 + n^3 = 2n^3$
- We have just shown that $n^2 + n \leq 2n^3$ for all $n \geq 1$
- Thus, we have shown that $n^2 + n = O(n^3)$ (by definition of Big-O, with $n_0 = 1$, and $c = 2$.)

Big-Ω notation

Definition:

$f(n) = \Omega(g(n))$ iff there are two positive constants c and n_0 such that $|f(n)| \geq c |g(n)|$ for all $n \geq n_0$. If $f(n)$ is nonnegative, we can simplify the last condition to $0 \leq c g(n) \leq f(n)$ for all $n \geq n_0$ • then we say that “ **$f(n)$ is omega of $g(n)$.**” • As n increases, $f(n)$ grows no slower than $g(n)$. In other words, $g(n)$ is an asymptotic lower bound on $f(n)$



Example: $n^3 + 4n^2 = \Omega(n^2)$

Proof: • Here, we have $f(n) = n^3 + 4n^2$, and $g(n) = n^2$

- It is not too hard to see that if $n \geq 0$, $n^3 \leq n^3 + 4n^2$
- We have already seen that if $n \geq 1$, $n^2 \leq n^3$
- Thus when $n \geq 1$, $n^2 \leq n^3 \leq n^3 + 4n^2$
- Therefore,

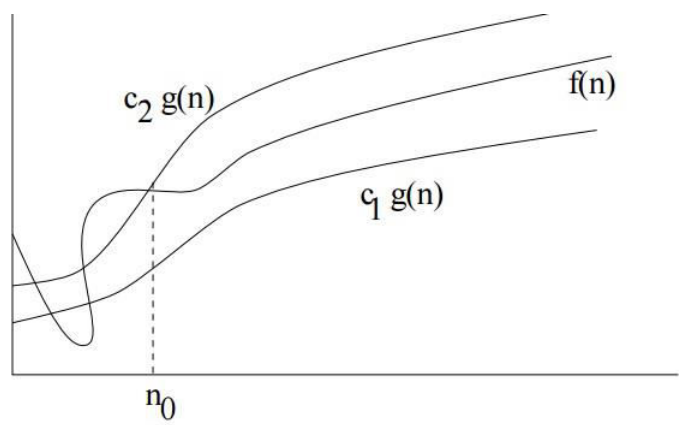
$$1n^2 \leq n^3 + 4n^2 \text{ for all } n \geq 1$$

- Thus, we have shown that $n^3 + 4n^2 = \Omega(n^2)$ (by definition of Big-Ω, with $n_0 = 1$, and $c = 1$.)

Big-Θ notation

Definition:

$f(n) = \Theta(g(n))$ iff there are three positive constants c_1 , c_2 and n_0 such that $c_1 |g(n)| \leq |f(n)| \leq c_2 |g(n)|$ for all $n \geq n_0$. If $f(n)$ is nonnegative, we can simplify the last condition to $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$. then we say that **“f(n) is theta of g(n).”** As n increases, $f(n)$ grows at the same rate as $g(n)$. In other words, $g(n)$ is an asymptotically tight bound on $f(n)$.



Example: $n^2 + 5n + 7 = \Theta(n^2)$

Proof: •

When $n \geq 1$, $n^2 + 5n + 7 \leq n^2 + 5n^2 + 7n^2 \leq 13n^2$

- When $n \geq 0$, $n^2 \leq n^2 + 5n + 7$
- Thus, when $n \geq 1$

$$1n^2 \leq n^2 + 5n + 7 \leq 13n^2$$

Thus, we have shown that $n^2 + 5n + 7 = \Theta(n^2)$ (by definition of Big-Θ, with $n_0 = 1$, $c_1 = 1$, and $c_2 = 13$.)

Comparison of different Algorithm

Algorithm	Best case	Average case	Worst case
Quick sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Heap sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$
Binary search	$O(1)$	$O(\log n)$	$O(\log n)$
Linear search	$O(1)$	$O(n)$	$O(n)$

Abstract and Concrete Data Structures- Basic data structures – vectors and arrays. Applications, Linked lists:- singly linked list, doubly linked list, Circular linked list, operations on linked list, linked list with header nodes, applications of linked list: polynomials,.

2.1. ABSTRACT AND CONCRETE DATA STRUCTURE:

- All data types which are absolutely defined are called as concrete data types
- Abstract data type can be constructed from known –or unknown data.

For example: Boolean, Integer, Floating point, String are examples of concrete data types as they are very strictly defined to contain the specified data type values

Array can be an example of an abstract data type as an array can consist of a number of Booleans, integers, Alphanumeric, Text, or even arrays (A program can (if properly programmed) seek out a particular cell in an array, and return the data there, no matter what data type.)

Concrete data type is reverse of abstract data type.

Differences are:

- The concrete data type is defined for certain inputs and outputs, whereas abstract is defined for all kind of inputs and outputs.
- A concrete data type is rarely reusable, whereas abstract data types are reusable repetitively
- Arrays, lists and trees are concrete data types whereas stacks, queues and heaps are abstract data types.

For implementing abstract data type, we need to choose a suitable concrete data type.

ABSTRACT AND CONCRETE DATA STRUCTURES

Abstract Data Structures	Concrete Data Structures
Concrete data types or structures (CDT's) are direct implementations of a relatively simple concept	Abstract Data Types (ADT's) offer a high level view (and use) of a concept independent of its implementation
Data types which are absolutely defined	constructed from known –or unknown data
Array, records, linked lists, trees, graphs	stacks, queues and heaps
	Usually there are many ways to implement the same ADT, using several different concrete data structures. An abstract stack can be implemented by a linked list or by an array

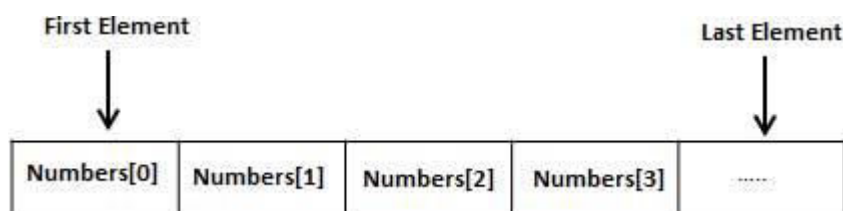
2.2 BASIC DATA STRUCTURE

2.2.1. Array

Arrays are a kind of data structure that can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



Declaring Arrays

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows –


```
type arrayName [ arraySize ];
```

This is called a *single-dimensional* array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C data type. For example, to declare a 10-element array called **balance** of type double, use this statement –

```
double balance[10];
```

Here *balance* is a variable array which is sufficient to hold up to 10 double numbers.

Initializing Arrays

You can initialize an array in C either one by one or using a single statement as follows –

```
double balance[5] = { 1000.0, 2.0, 3.4, 7.0, 50.0};
```

The number of values between braces { } cannot be larger than the number of elements that we declare for the array between square brackets [].

If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write –

```
double balance[] = { 1000.0, 2.0, 3.4, 7.0, 50.0};
```

You will create exactly the same array as you did in the previous example. Following is an example to assign a single element of the array –

```
balance[4] = 50.0;
```

The above statement assigns the 5th element in the array with a value of 50.0. All arrays have 0 as the index of their first element which is also called the base index and the last index of an array will be total size of the array minus 1. Shown below is the pictorial representation of the array we discussed above –

	0	1	2	3	4
balance	1000.0	2.0	3.4	7.0	50.0

Input data into array

```

for (x=0; x<=n;x++)
{
    printf("enter the integer number %d\n", x);
    scanf("%d", &num[x]);
}

```

Reading out data from an array

```

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

```

Accessing Array Elements

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example –

```
double salary = balance[9];
```

salary variable. The above statement will take the 10

The following example Shows how to use all the three above mentioned concepts viz. declaration, assignment, and accessing arrays –

```

#include <stdio.h>

int main ()
{
    int n[ 10 ]; /* n is an array of 10 integers */
    int i,j;
    /* initialize elements of array n to 0 */
    for ( i = 0; i < 10; i++ )
    {
        n[ i ] = i + 100; /* set element at location i to i + 100 */
    }

    /* output each array element's value */
    for ( j = 0; j < 10; j++ )

```

```

        {
            printf("Element[%d] = %d\n", j, n[j] );
        }
    return 0;
}

```

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

```

Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109

```

Write a C Program to insert an element into the beginning of an array

```

#include<stdio.h>
#include<conio.h>
int main()
{
    int a[10],i,n,p;
    printf("Enter the number of elements\n");
    scanf("%d",&n);
    printf("Enter the Elements\n");
    for(i=0;i<n;i++)
    {
        scanf("%d",&a[i]);
    }
    if(n==10)
    {
        printf("Array is full");
    }
    else

```

```

{
    printf("Enter the element to be inserted")
    scanf("%d",&p);
    for(i=n-1;i>=0;i--)
    {
        a[i+1]=a[i];
    }
    n=n+1
    a[0]=p;
    printf("After insertion elements are:\n");
    for(i=0;i<n;i++)
    {
        printf("%d\t",a[i]);
    }
}
getch();
}

```

Write a C Program to insert an element into the end of an array

```

#include<stdio.h>
#include<conio.h>
int main()
{
    int a[10],i,n,p;
    printf("Enter the number of elements\n");
    scanf("%d",&n);
    printf("Enter the Elements\n");
    for(i=0;i<n;i++)
    {
        scanf("%d",&a[i]);
    }
    if(n==10)
    {
        printf("Array is full");
    }
    else

```

```

{
    printf("Enter the element to be inserted")
    scanf("%d",&p);
    a[n+1]=p;
    n=n+1;
    printf("After insertion elements are:\n");
    for(i=0;i<n;i++)
    {
        printf("%d\t",a[i]);
    }
}
getch();
}

```

Write a C Program to insert an element into the particular position of an array

```

#include <stdio.h>
int main()
{
    int array[100], position, c, n, value;
    printf("Enter number of elements in array\n");
    scanf("%d", &n);
    printf("Enter %d elements\n", n);
    for (c = 0; c < n; c++)
        scanf("%d", &array[c]);
    printf("Enter the location where you wish to insert an element\n");
    scanf("%d", &position);

    if(position>=100)
        printf("Array is full")
    else
    {
        printf("Enter the value to insert\n");
        scanf("%d", &value);

        for (c = n - 1; c >= position - 1; c--)

```

```
array[c+1] = array[c];
```

```
array[position-1] = value;
```

```
n=n+1;
```

```
printf("Resultant array is\n");
```

```
for (c = 0; c <=n; c++)
```

```
    printf("%d\n", array[c]);
```

```
}
```

```
getch();
```

```
}
```

Write a C Program to delete an element from the beginning of an array

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

```
#include<conio.h>
```

```
int main()
```

```
{
```

```
    int a[10],i,n,p;
```

```
    printf("Enter the number of elements\n");
```

```
    printf("Enter the Elements\n");
```

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

```
{
```

```
    scanf("%d",&a[i]);
```

```
}
```

```
    p=a[0];
```

```
    for(i=1;i<n;i++)
```

```
{
```

```
        a[i-1]=a[i];
```

```
}
```

```
    n=n-1;
```

```
    printf("Deleted elements are %d",p);
```

```
    if(n==0)
```

```
        printf("Array is empty");
```

```
else
```

```

{
    printf("After deletion elements are:\n");
    for(i=0;i<n;i++)
    {
        printf("%d\t",a[i]);
    }
}
getch();
}

```

Write a C Program to delete an element from the end of an array

```

#include<stdio.h>
#include<conio.h>
int main()
{
    int a[10],i,n;
    printf("Enter the number of elements\n");
    scanf("%d",&n);
    printf("Enter the Elements\n");
    for(i=0;i<n;i++)
    {
        scanf("%d",&a[i]);
    }
    p=a[n];
    printf("Deleted element is %d",p);
    n=n-1;
    if(n==0)
        printf("Array is empty");
    else
    {
        printf("After deletion elements are:\n");
        for(i=0;i<n;i++)
        {
            printf("%d\t",a[i]);
        }
    }
}

```



```

    getch();
}

```

Write a C Program to delete an element from the particular position of an array

```

#include <stdio.h>

int main()
{
    int array[100], position, c, n, value;
    printf("Enter number of elements in array\n");
    scanf("%d", &n);
    printf("Enter %d elements\n", n);
    for (c = 0; c < n; c++)
        scanf("%d", &array[c]);
    printf("Enter the location where you wish to delete an element\n");
    scanf("%d", &position);
    value=array[position-1];
    for (c = position; c >n; c--)
        array[c-1] = array[c];
    n=n-1;

    if(==0)
        printf("Array is empty");
    else
    {
        printf("Resultant array is\n");

        for (c = 0; c <=n; c++)
            printf("%d\n", array[c]);
    }
    getch();
}

```

Two dimensional array

Ordered in number of rows and columns .Array $m*n$ denotes m number of row and n number of columns

$a_{11}, a_{12}, a_{13} \dots a_{1n}$

$a_{21}, a_{22}, a_{23} \dots a_{2n}$

.....

.....

$a_{m1}, a_{m2}, a_{m3} \dots a_{mn}$

Subscript $a[i][j]$ represents the i th row and j th column.

Memory representation of matrix

It is also stored in contiguous memory location. There are two conventions of storing matrix in memory

1. Row-major representation
2. Column-major representation

1. Row major representation

Elements of the matrix are stored in row by row basis. Assume that base address is the first location of the memory, address of $a[i][j]$ is Address $a[i][j]$ = storing all elements in the first $(i-1)$ rows + the number of elements in the i -th row up to j -th column $= (i-1)*n + j$, where n is the number of columns (Assume base address is 1)

If the base address is M Address $a[i][j] = M + (i-1)*n + j - 1$

2. Column major representation

All elements in the first column is stored first, then second column and so on. Address of $a[i][j]$ = storing all elements in the first $(j-1)$ columns elements + The number of elements in j -th column up to i -th row $= (j-1)*m + i$, where m is the number of rows (Assume base address is 1) If the base address is M Address of $a[i][j] = M + (j-1)*m + i - 1$

2.2.2. VECTOR

- Vector is an array with a dynamic size.
- Instead of having a predefined size to the structure it increases, decreases its size as you add/remove elements from/ to it.
- Which together gives as some other advantages as adding elements at a specific index and removing from a specific index.
- Vector is not as fast as the array, but altogether efficient. Operations on a vector offer the same big O as their counterparts on an array.
- Like arrays, vector data is allocated in contiguous memory. This can be done either explicitly or by adding more data.
- In order to do this efficiently, the typical vector implementation grows by doubling its allocated space (rather than incrementing it) and often has more space allocated to it at any one

time than it needs. This is because reallocating memory can sometimes be an expensive operation.

APPLICATIONS

1. Stores Elements of Same Data Type
2. Used for Maintaining multiple variable names using single name
3. Can be Used for Sorting Elements
4. Can Perform Matrix Operation
5. Can be Used in CPU Scheduling
6. Can be Used in Recursive Function
7. Can be used to implement abstract data structures like stack, queue etc

2.3 LINKED LIST

If the memory is allocated before the execution of a program, it is fixed and cannot be changed. We have to adopt an alternative strategy to allocated memory only when it is required. There is a special data structure called linked list that provides a more flexible storage system and it does not require the use of array.

Linked lists are special list of some data elements linked to one another. The logical ordering is represented by having each element pointing to the next element. Each element is called a node, which has two parts, INFO part which stores the information and LINK which points to the next element.

Advantages

Linked list have many advantages. Some of the very important advantages are:

- ***Linked Lists are dynamic data structure:*** That is, they can grow or shrink during the execution of a program.
- ***Efficient memory utilization:*** Here, memory is not pre-allocated. Memory is allocated whenever it is required. And it is deallocated when it is no longer needed.
- ***Insertion and deletions are easier and efficient:*** Linked lists provide flexibility in inserting data item at a specified position and deletion of a data item from the given position.
- ***Many complex applications can be easily carried out with linked lists.***

Disadvantages

- ***More Memory:*** If the numbers of fields are more, then more memory space is needed.
- Access to an arbitrary data item is little bit cumbersome and also time consuming.

Types of Linked List

Following are the various flavours of linked list.

- Simple Linked List – Item Navigation is forward only.
- Doubly Linked List – Items can be navigated forward and backward way.
- Circular Linked List – Last item contains link of the first element as next and first element has link to last element as prev.

Basic Operations

- Insertion – add an element at the beginning of the list.
- Display – displaying complete list.
- Search – search an element using given key.
- Delete – delete an element using given key

2.3.1 Singly Linked List

A linked list is a non-sequential collection of data items called nodes. These nodes in principles are structures containing fields. Each node in a linked list has basically two fields.

1. DATA field

The DATA field contains an actual value to be stored and processed. And, the NEXT field contains the address of the next data item in the linked list. The address used to access a particular node is known as a pointer. Therefore, the elements in a linked list are ordered not by their physical placement in memory but their logical links stored as part of the data within the node itself.

Note that, the link field of the last node contains NULL rather than a valid address. It is a NULL pointer and indicates the end of the list. External pointer(HEAD) is a pointer to the very first node in the linked list, it enables us to access the entire linked list.



Basic Operations

Following are the basic operations supported by a list.

- **Insertion** – A new node may be inserted
 - At the beginning of the linked list
 - At the end of the linked list
 - At the specific position of the linked list
- **Deletion** – delete an element from the
 - Beginning of the linked list
 - End of the linked list
 - Specific position of the linked list
- **Display** – This operation is used to print each and every node's information.
- **Traversing** – It is a process of going through all the nodes of a linked list from one end to the other end. If we start traversing from the very first node towards the last node, it is called forward traversing. If the desired element is found, we signal operation "SUCCESSFUL". Otherwise, we signal it as "UNSUCCESSFUL".

Steps to create a linked list

Step 1 : Include alloc.h Header File

```
#include<alloc.h>
```

1. We don't know, how many nodes user is going to create once he execute the program.
2. In this case we are going to allocate memory using Dynamic Memory Allocation functions malloc.
3. Dynamic memory allocation functions are included in alloc.h

Step 2 : Define Node Structure

We are now defining the new global node which can be accessible through any of the function.

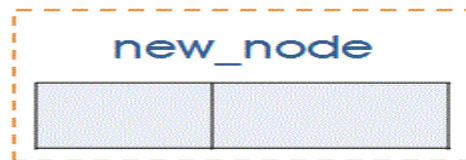
```
struct node
{
```

```
int data;
struct node *next;
}*start=NULL;
```

Step 3 : Create Node using Dynamic Memory Allocation

Now we are creating one node dynamically using malloc function. We don't have prior knowledge about number of nodes, so we are calling malloc function to create node at run time.

```
new_node=(struct node *)malloc(sizeof(struct node));
```

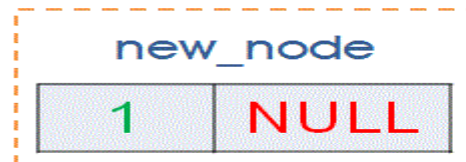



 Created New Node using Dynamic memory allocation

Step 4 : Fill Information in newly Created Node

Now we are accepting value from the user using scanf. Accepted Integer value is stored in the data field. Whenever we create new node, Make its Next Field as NULL.

```
printf("Enter the data : ");
scanf("%d",&new_node->data);
new_node->next=NULL;
```



 Fill Node with the data provided by user

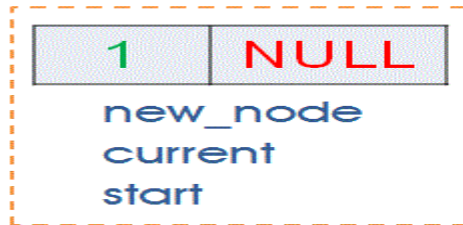
Step 5 : Creating Very First Node

If node created in the above step is very first node then we need to assign it as starting node. If start is equal to null then we can identify node as first node.

```
start = NULL
```

First node has 3 names : new_node, current, start

```
if(start == NULL) {
    start = new_node;
    curr = new_node;
}
```



Below are the some of the alternate names given to the Created Linked List node

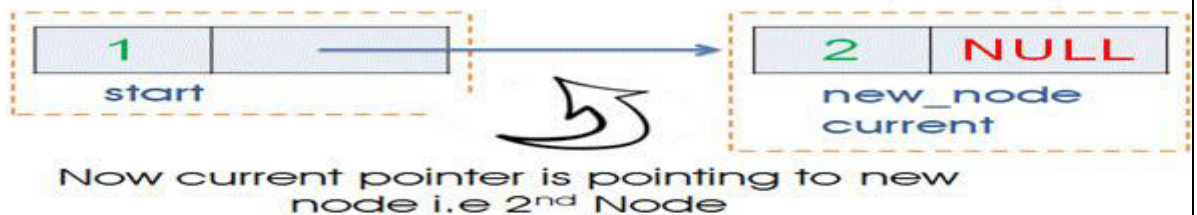
Step 6: Creating Second or nth node

1. Let's assume we have 1 node already created i.e we have first node. First node can be referred as "new_node", "curr", "start".
2. Now we have called create() function again

Now we already have starting node so control will be in the else block –

else

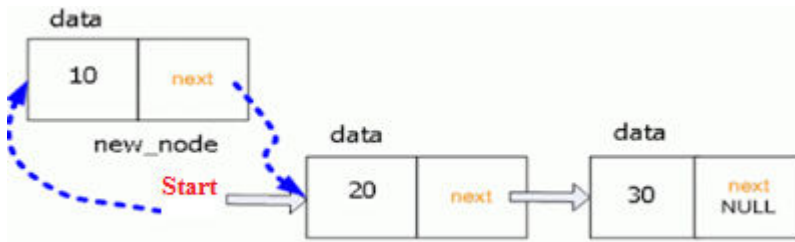
```
{
current->next = new_node; // link between new_node and current node
current = new_node; // move current pointer to next node
}
```



Insert node at Start/First Position in Singly Linked List

1. Allocate a memory for the new_node
2. Insert Data into the "Data Field" of new_node
3. If(start=NULL) list is empty then goto step 4 otherwise goto step 5
4. Start point to the new_node and set linked field of new_node to NULL

5. Linked field of the new_node pointed to start, then set start to new_node.



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

struct node
{
    int data;
    struct node *ptr;
};

int main()
{
    typedef struct node NODE;
    NODE *start=NULL, *temp;
    int item;
    printf("\n\n\tSingle Linked List ");
    temp=(NODE*)malloc(sizeof(NODE));
    printf("Enter the data to be inserted: ");
    scanf("%d", &temp->data);

    if(start==NULL)
    {
        temp->ptr=NULL;
        start=temp;
    }

    else
    {
        temp->ptr=start;
        start=temp;
    }

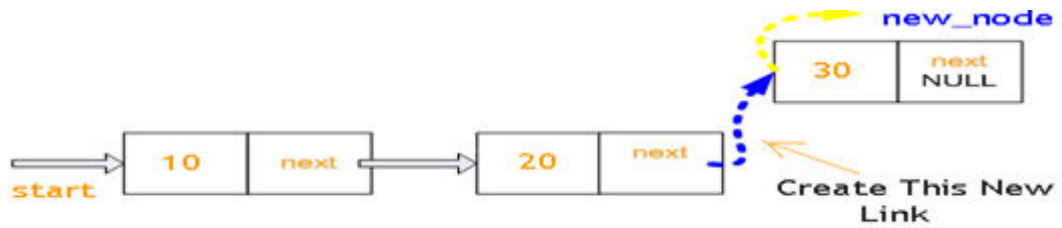
    getch();
}
```

Insert node at Start/First Position in Singly Linked List

Algorithm

1. Allocate a memory for the new_node
2. Insert Data into the “Data Field“ of new_node and set “Linked field of new_node to NULL.
3. If(start=NULL) list is empty then goto step 4 otherwise goto step 5

4. Start point to the new_node and set linked field of new_node to NULL
5. Node is to be inserted at Last Position so we need to traverse SLL upto Last Node.
6. Linked field of last node pointed to the new node.



Program

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

struct node
{
    int data;
    struct node *ptr;
};

int main()
{
    typedef struct node NODE;
    NODE *start=NULL, *temp, *p;
    int item;

    printf("\n\n\tSingle Linked List ");
    printf("\n Insert into end");
    temp=(NODE*)malloc(sizeof(NODE));
    printf("Enter the data to be inserted: ");
    scanf("%d", &item);
    item=temp->data;

    if(start==NULL)
    {
        temp->ptr=NULL;
        start=temp;
    }

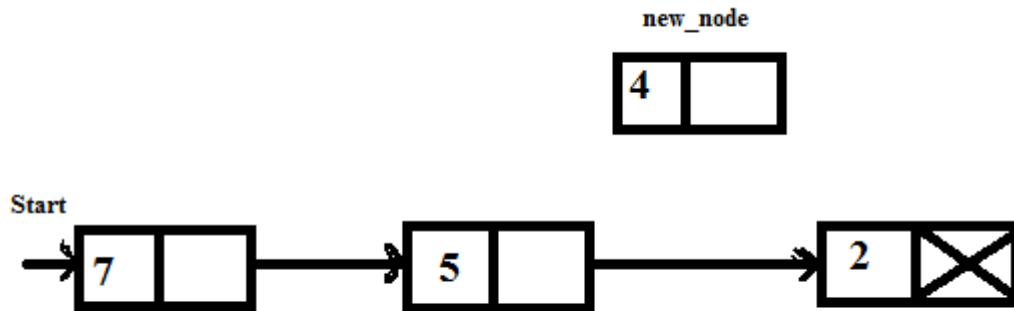
    else
    {
        p=start;
        while(p->ptr!=NULL)
        {
            p=p->ptr;
        }
        p->ptr=temp;
        temp->ptr=NULL;
    }
    getch();
}
```

}

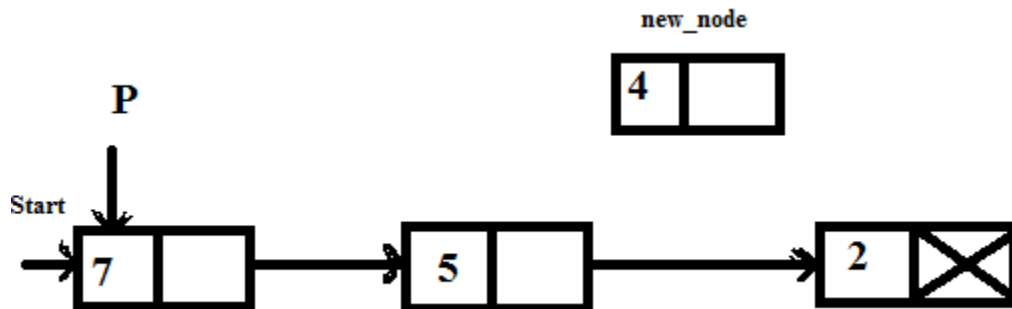
Insert node at Particular Position in Singly Linked List

Algorithm

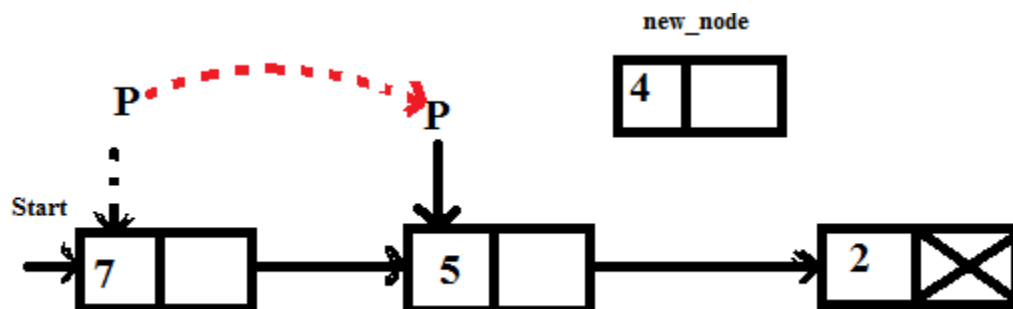
1. Allocate memory for the new node
2. Assign value to the data field of the new node
3. Find the position to insert
4. Suppose new node insert between node A and node B, make the link field of the new node point to the node B and make the link field of node A point to the new node

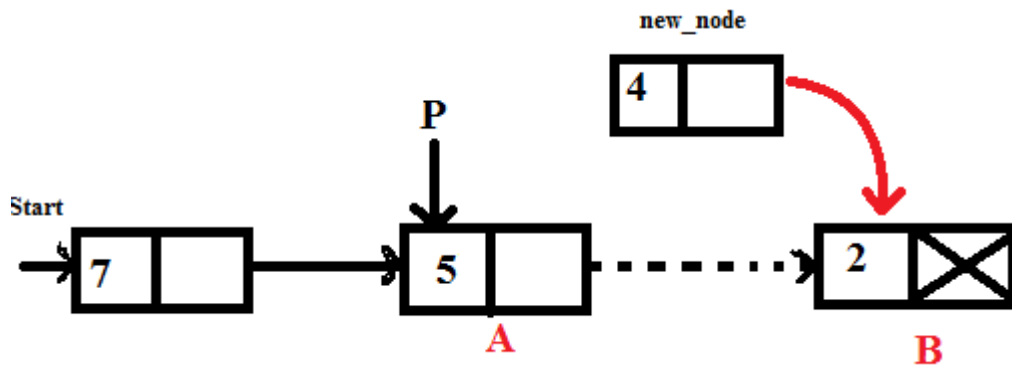
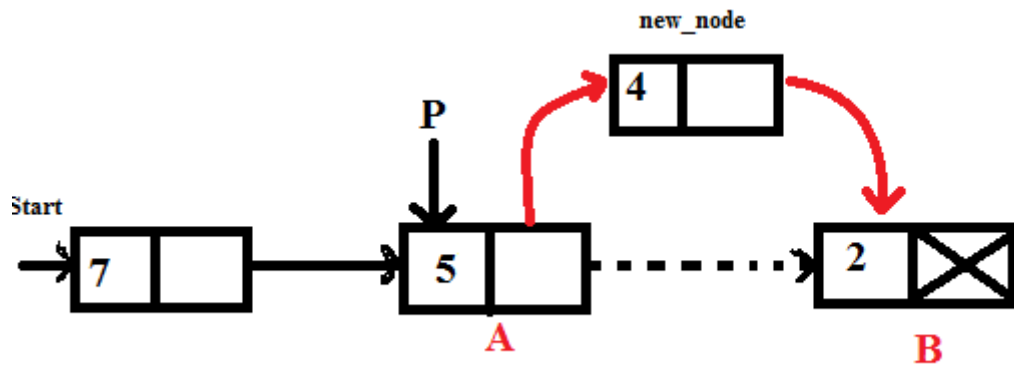


(a) Create a new node and insert the data into the data field



(b) p=start



(c) $p=p->ptr$ (d) $new_node->ptr=p->ptr$ Program

```

#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
struct node
{
    int data;
    struct node*ptr;
};

void main()
{
    typedef struct node NODE;
    NODE *start=NULL,*temp,*p,*t;

```

```

int ch,item,pos,i;
printf("\nEnter the number: ");
scanf("%d",&item);
temp=(NODE*)malloc(sizeof(NODE));
temp->data=item;
printf("\nEnter the position: ");
scanf("%d",&pos);
p=start;
for(i=1;i<pos-1;i++)
{
    p=p->ptr;
}
temp->ptr=p->ptr;
p->ptr=temp;
getch();
}

```

Display the elements in the linked list

2. Otherwise print elements from start to NULL

```

if(start==NULL)
    printf("\nList is empty");
else
{
    printf("\nElements are:");
    for(p=start;p!=NULL;p=p->ptr)
        printf(" %d",p->data);
}

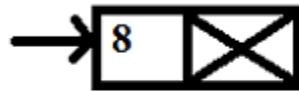
```

Delete node at Start/First Position in Singly Linked List

Algorithm

1. If list is empty, deletion is not possible.
2. If the list contain only one element, set external pointer to NULL. Otherwise move the external pointer point to the second node and delete the first node

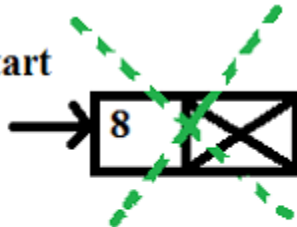
Start



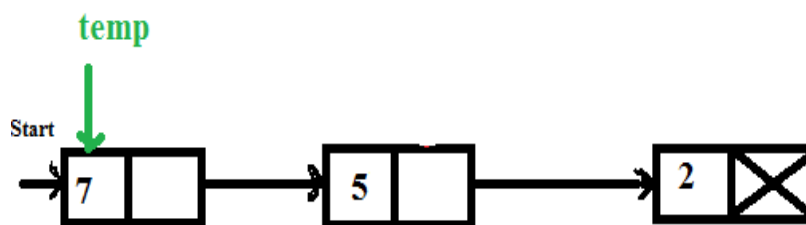
start->ptr=NULL

start->ptr=NULL
(Only one element)

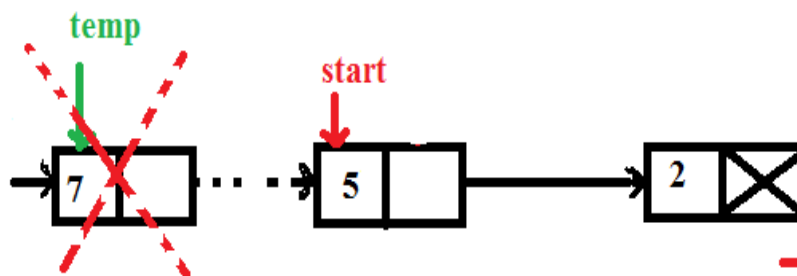
Start



start=NULL



More than one
elements



Program

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
struct node
{
    int data;
    struct node*ptr;
};
```

```

void main()
{
    typedef struct node NODE;
    NODE *start=NULL,*temp;
    int item;
    if(start==NULL)
        printf("\nDeletion is not possible");
    else if(start->ptr==NULL)
    {
        temp=start;
        start=NULL;
        printf("\nDeleted item is %d",temp->data);
        free(temp);
    }
    else
    {
        temp=start;
        start=start->ptr;
        printf("\nDeleted item is %d",temp->data);
        free(temp);
    }
    getch();
}

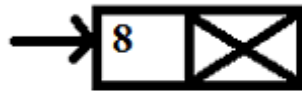
```

Delete node at End/Last Position in Singly Linked List

Algorithm

1. If the list is empty, deletion is not possible
2. If the list contain only one element, set external pointer to NULL
3. Otherwise go on traversing the second last node and set the link field point to NULL

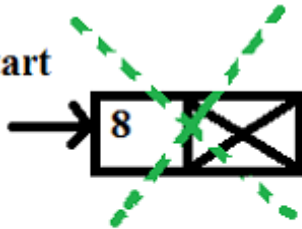
Start



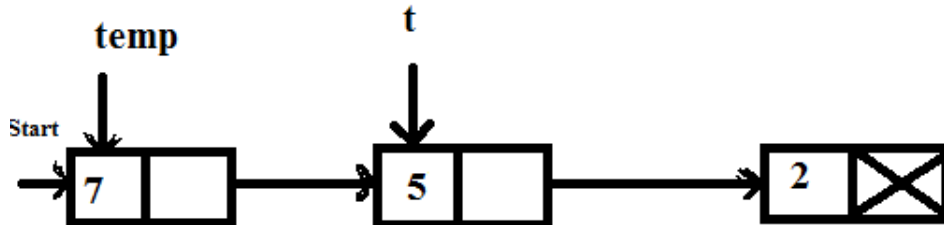
start->ptr=NULL

start->ptr=NULL
(Only one element)

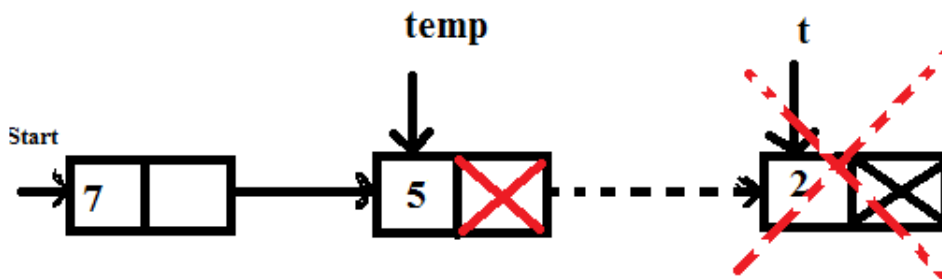
Start



start=NULL



more than one
node



Program

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
struct node
{
    int data;
    struct node*ptr;
};
```

```
void main()
```

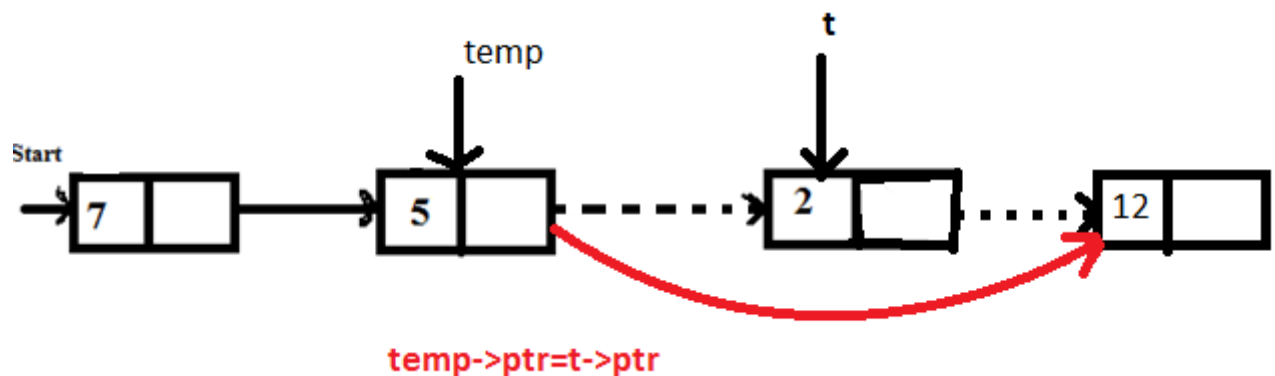
```

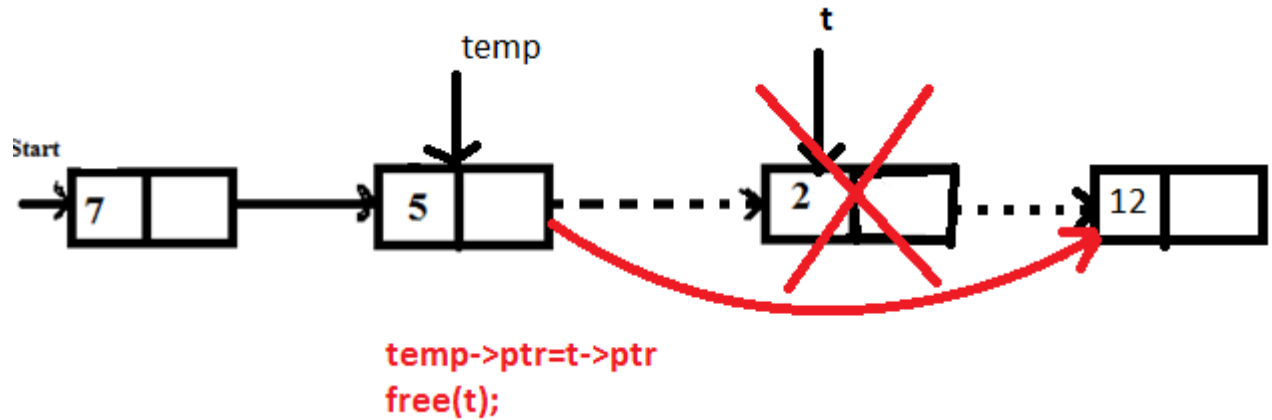
{
    typedef struct node NODE;
    NODE *start=NULL,*temp,*p,*t;
    int ch,item,pos,i;
    if(start==NULL)
        printf("\nDeletion is not possible");
    else if(start->ptr==NULL)
    {
        temp=start;
        start=NULL;
        printf("\nDeleted item is %d",temp->data);
        free(temp);
    }
    else
    {
        temp=start;
        t=start->ptr;
        while(t->ptr!=NULL)
        {
            t=t->ptr;
            temp=temp->ptr;
        }
        temp->ptr=NULL;
        printf("\nDeleted item is %d",t->data);
        free(t);
    }
    getch();
}

```

Delete node at Particular Position in Singly Linked List

1. Find the position to delete.
2. Suppose we delete node between node A & node B, set the link field of node A point to the node B





Program

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
struct node
{
    int data;
    struct node*ptr;
};

void main()
{
    typedef struct node NODE;
    NODE *start=NULL,*temp,*p,*t;
    int ch,item,pos,i;

    printf("\nEnter the position: ");
    scanf("%d",&pos);
    temp=start;
    if(pos==1)
    {
        temp=start;
        start=start->ptr;
        printf("\nDeleted item is %d",temp->data);
        free(temp);
    }
    else
    {
        for(i=1;i<pos-1;i++)
            temp=temp->ptr;
        t=temp->ptr;
        temp->ptr=t->ptr;
        printf("\nDeleted item is %d",t->data);
        free(t);
    }
}
```

```

    }
    getch()
}

```

To write a program to implement singly linked list.

- 1. Insert at beginning**
- 2. Insert at particular position**
- 3. Insert at end**
- 4. Delete from beginning**
- 5. Delete from particular position**
- 6. Delete from end**
- 7. Display**

```

#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
struct node
{
    int data;
    struct node*ptr;
};

void main()
{
    typedef struct node NODE;
    NODE *start=NULL,*temp,*p,*t;
    int ch,item,pos,i;
    clrscr();
    while(1)
    {
        printf("\nMENU: \n1.Insert at beginning\n2.Insert at particular position\n3.Insert at end\n4.Delete
from beginning\n5.Delete from particular position\n6.Delete from end\n7.Display\n8.Exit\nEnter your
choice: ");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1:
                printf("\nEnter the number: ");
                scanf("%d",&item);
                temp=(NODE*)malloc(sizeof(NODE));
                temp->data=item;

```

```

if(start==NULL)
{
    temp->ptr=NULL;
    start=temp;
}
else
{
    temp->ptr=start;
    start=temp;
}
break;

```

case 2:

```

printf("\nEnter the number: ");
scanf("%d",&item);
temp=(NODE*)malloc(sizeof(NODE));
temp->data=item;
printf("\nEnter the position: ");
scanf("%d",&pos);
p=start;
for(i=1;i<pos-1;i++)
    p=p->ptr;
temp->ptr=p->ptr;
p->ptr=temp;

```

break;

case 3:

```

printf("\nEnter the number: ");
scanf("%d",&item);
temp=(NODE*)malloc(sizeof(NODE));
temp->data=item;
temp->ptr=NULL;
if(start==NULL)
    start=temp;
else
{
    p=start;
    while(p->ptr!=NULL)
        p=p->ptr;
    p->ptr=temp;
}
break;

```

case 4:

```

if(start==NULL)
    printf("\nDeletion is not possible");
else if(start->ptr==NULL)
{

```

```

        temp=start;
        start=NULL;
        printf("\nDeleted item is %d",temp->data);
        free(temp);
    }
    else
    {
        temp=start;
        start=start->ptr;
        printf("\nDeleted item is %d",temp->data);
        free(temp);
    }
    break;
case 5:

    printf("\nEnter the position: ");
    scanf("%d",&pos);
    temp=start;

    for(i=1;i<pos-1;i++)
        temp=temp->ptr;
    t=temp->ptr;
    temp->ptr=t->ptr;
    printf("\nDeleted item is %d",t->data);
    free(t);
break;
case 6:
    if(start==NULL)
        printf("\nDeletion is not possible");
    else if(start->ptr==NULL)
    {
        temp=start;
        start=NULL;
        printf("\nDeleted item is %d",temp->data);
        free(temp);
    }
    else
    {
        temp=start;
        t=start->ptr;
        while(t->ptr!=NULL)
        {
            t=t->ptr;
            temp=temp->ptr;
        }
        temp->ptr=NULL;

```

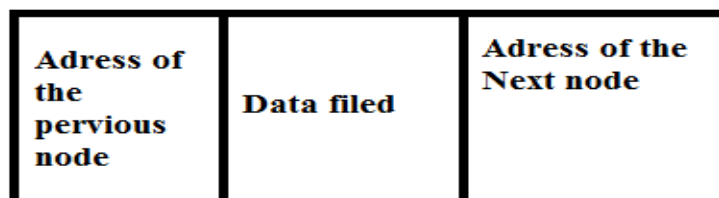
```

        printf("\nDeleted item is %d",t->data);
        free(t);
    }
    break;
case 7:
    if(start==NULL)
        printf("\nList is empty");
    else
    {
        printf("\nElements are:");
        for(p=start;p!=NULL;p=p->ptr)
            printf(" %d",p->data);
    }
    break;
case 8:
    exit(0);
default:
    printf("\nWrong Choice");
    break;
}
getch();
}
}

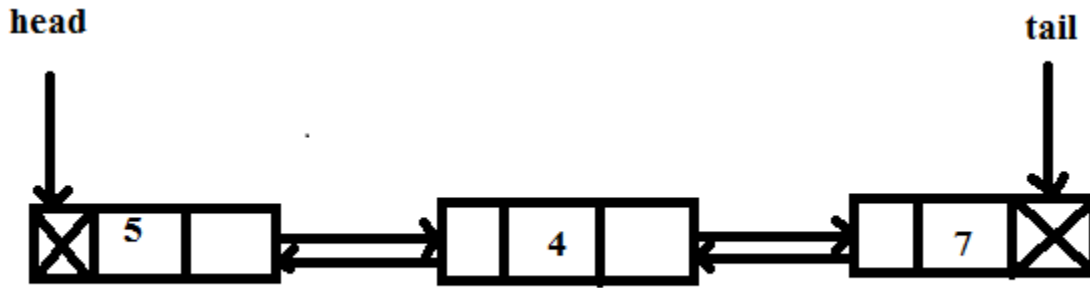
```

DOUBLY LINKED LIST

A more sophisticated kind of linked list is a doubly-linked list or a two-way linked list. In a doubly linked list, each node has two links: one pointing to the previous node and one pointing to the next node.



Example:



head is the external pointer ,used to point starting of the doubly linked list. Tail is used to denote end of the doubly linked list.

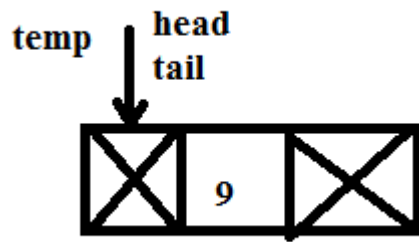
Representation of doubly linked list

struct node

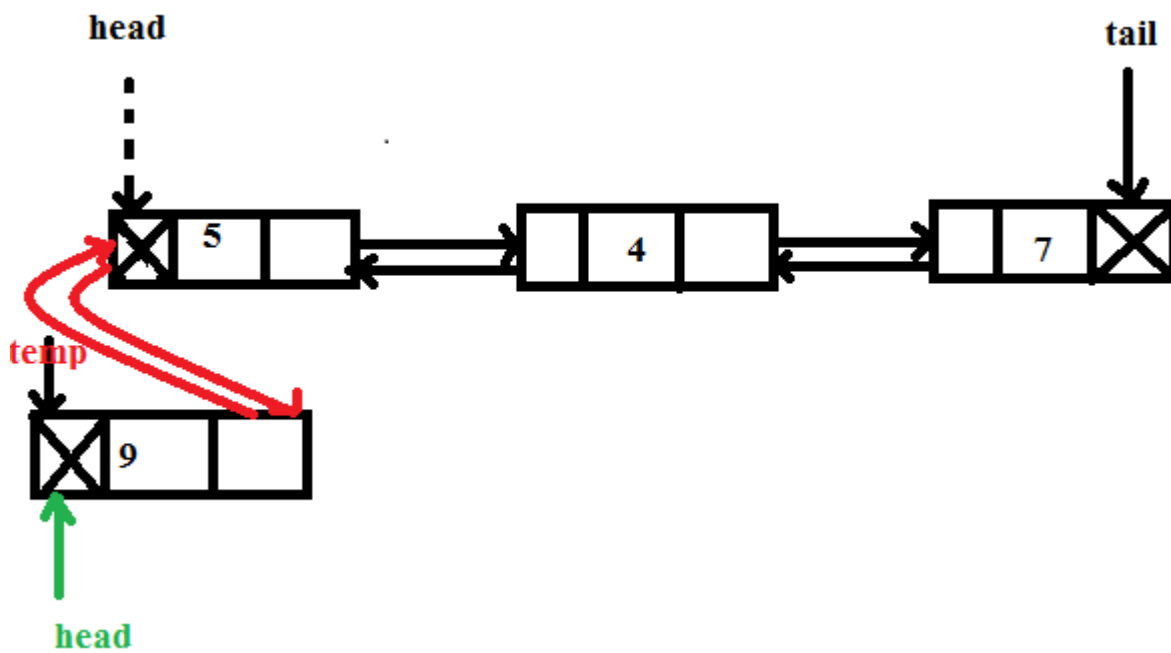
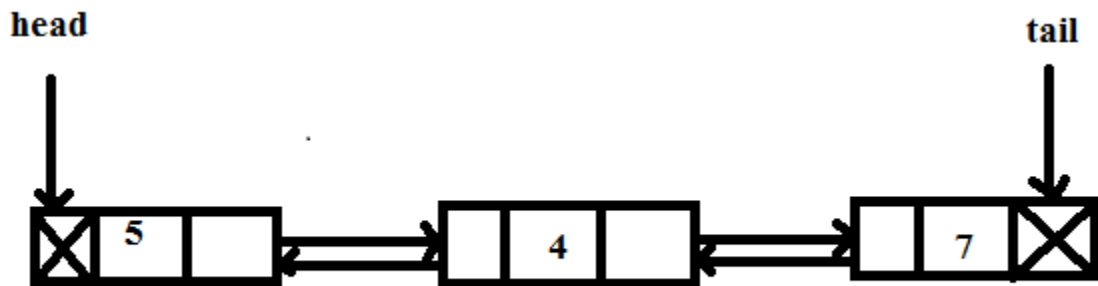
```
{
    int data;
    struct node *prev,*next;
};
```

Insert node at Start/First Position in Doubly Linked List

1. Allocate memory for new node
2. Assign value to the data field of the new node
3. If head=NULL then,
Set head and tail pointer point to the new node, set previous node is NULL and next node is NULL
4. Otherwise
Set temp→next=head and head→prev=temp and temp→prev=NULL
Set external pointer head point to the new node



head=NULL



head!=NULL

```
#include<stdio.h>
#include<conio.h>
#include<process.h>
struct node
{
    int data;
    struct node *prev,*next;
```

```

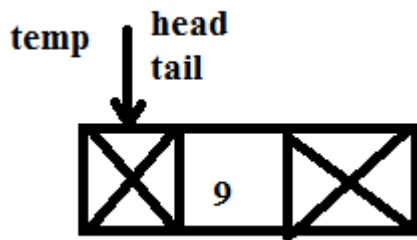
};
void main()
{
    typedef struct node NODE;
    NODE *head=NULL,tail=NULL,*temp;
    int no;
    temp=(NODE*)malloc(sizeof(NODE));
    printf("Enter the no: ");
    scanf("%d",&no);
    temp->data=no;
    if(start==NULL)
    {
        temp->prev=NULL;
        temp->next=NULL;
        head=tail=temp;
    }
    else
    {
        temp->next=head;
        head->prev=temp;
        temp->prev=NULL;
        head=temp;
    }
    if(head==NULL)
    {
        printf("No elements");
    }
    else
    {
        printf("\nElements are:");
        for(p= head;p!=NULL;p=p->next)
        {
            printf(" %d",p->data);
        }
    }
    getch();
}

```

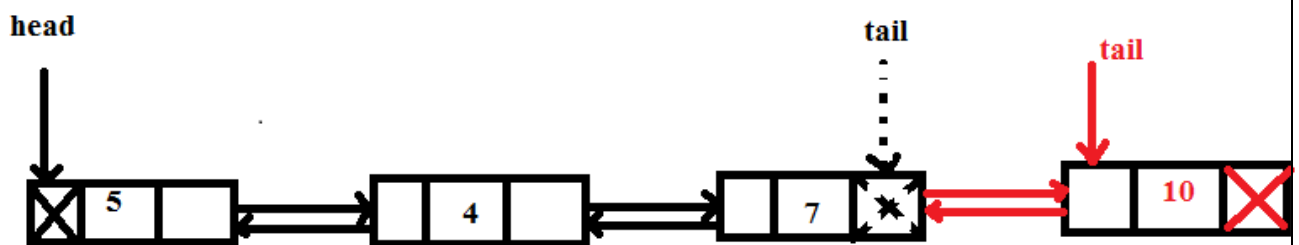
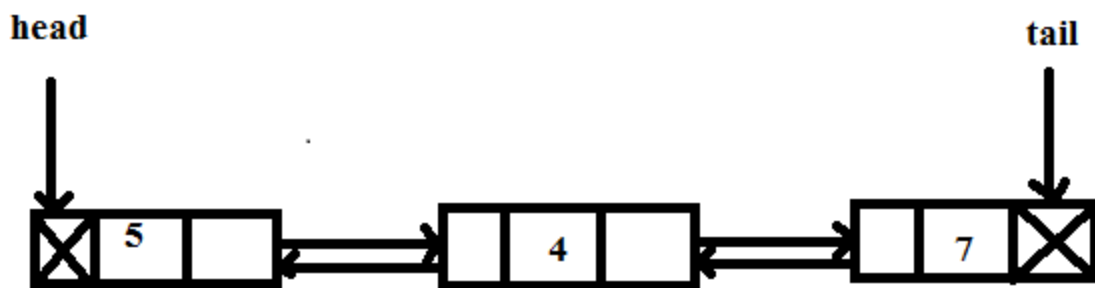
Insert node at End/Last Position in Doubly Linked List

1. Allocate memory for new node
2. Assign value to the new node
3. If head=NULL then,Set external pointer to the new node, set previous node is NULL and next node is NULL

4. Otherwise, $\text{tail} \rightarrow \text{next} = \text{temp}$, $\text{temp} \rightarrow \text{prev} = \text{tail}$ and next pointer of new node set to NULL. tail point to the new node



$\text{head} = \text{NULL}$



$\text{head} \neq \text{NULL}$

```
#include<stdio.h>
#include<conio.h>
#include<process.h>
struct node
{
    int data;
    struct node *prev,*next;
};
void main()
{
```

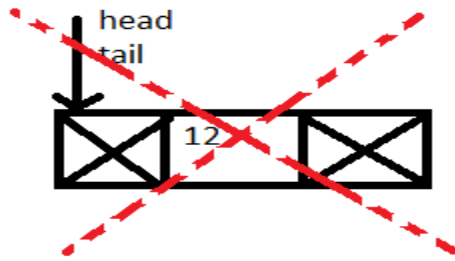
```

int ch,no;
typedef struct node NODE;
NODE *head=NULL,*tail=NULL,*temp;
temp=(NODE*)malloc(sizeof(NODE));
printf("Enter the no: ");
scanf("%d",&no);
temp->data=no;
    if(start==NULL)
    {
        temp->prev=NULL;
        temp->next=NULL;
        head=tail=temp;
    }
    else
    {
        tail->next=temp;
        temp->prev=tail;
        temp->next=NULL;
        tail=temp;
    }
    if(head==NULL)
    {
        printf("No elements");
    }
    else
    {
        printf("\nElements are:");
        for(p= head;p!=NULL;p=p->next)
        {
            printf(" %d",p->data);
        }
    }
getch();
}

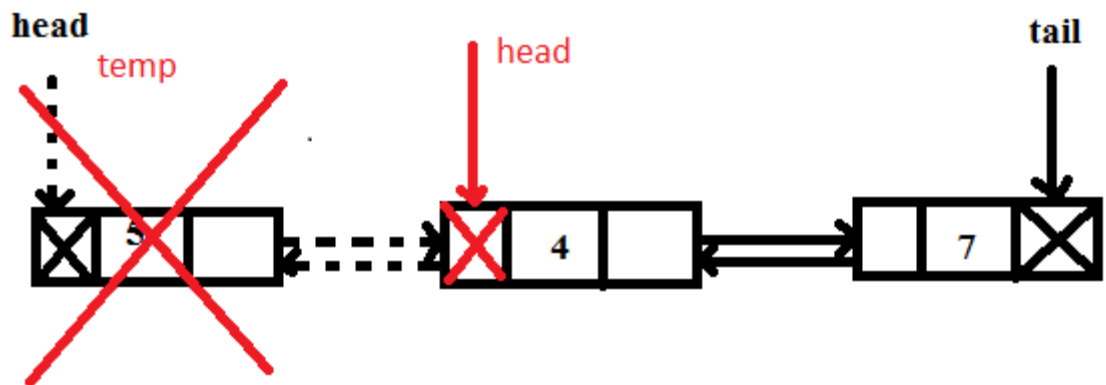
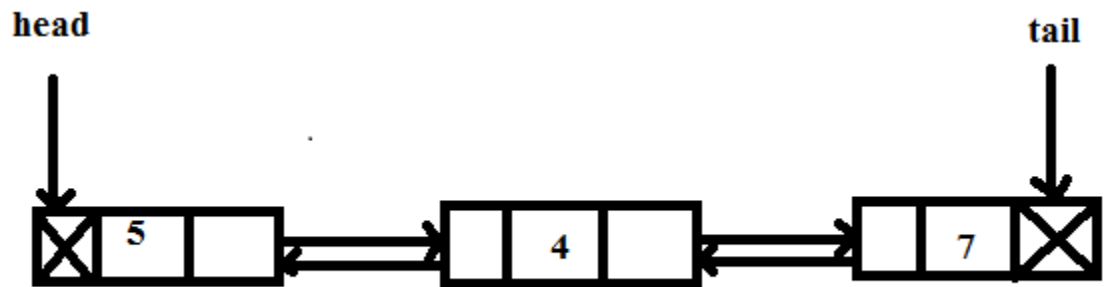
```

Delete a node at Start/First Position in Doubly Linked List

1. If start=NULL then,
Print deletion is not possible
2. If the list contain only one element, set external pointer to NULL
3. Otherwise move the external pointer point to the second node and delete first node



head=tail=NULL



```
#include<stdio.h>
#include<conio.h>
struct node
{
    int data;
    struct node *prev,*next;
};
void main()
{
    int no;
    typedef struct node NODE;
    NODE *head=NULL,*temp;
    if(head==NULL)
    {
        printf("Deletion is not possible");
    }
}
```

```

else if(head->next==NULL)
{
    temp=start;
    head=tail=NULL;
    printf("Deleted element is: %d",temp->data);
    free(temp);
}
else
{
    temp=head;
    head=temp->next;
    head->prev=NULL;
    printf("Deleted element is: %d",temp->data);
    free(temp);
}
if(head==NULL)
{
    printf("No elements");
}
else
{
    printf("\nElements are:");
    for(p=head;p!=NULL;p=p->next)
    {
        printf(" %d",p->data);
    }
}
}

```

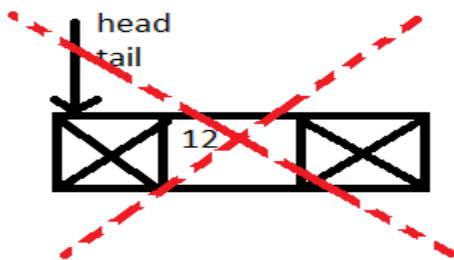
```

getch();
}

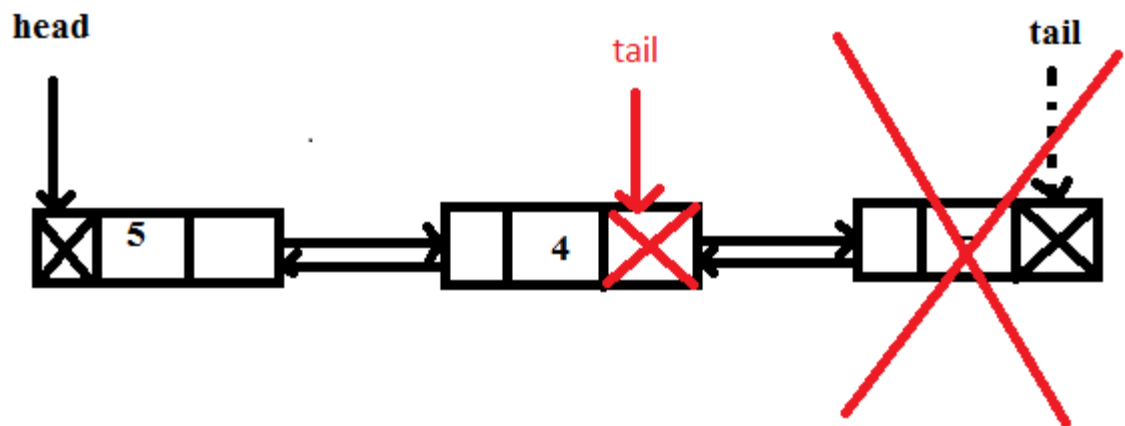
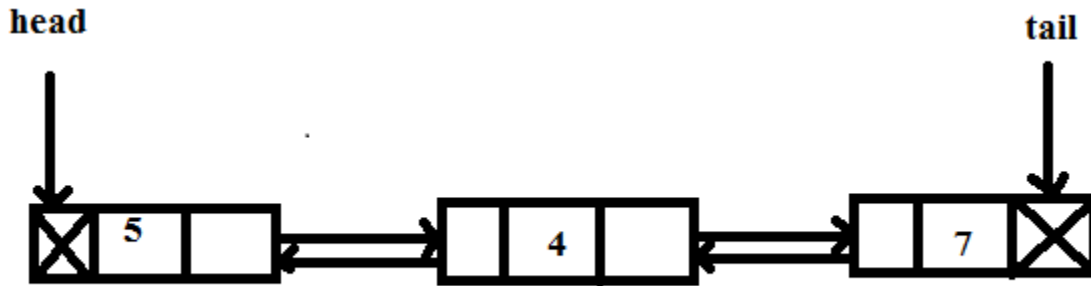
```

Delete a node at End/Last Position in Doubly Linked List

1. If the list is empty, deletion is not possible
2. If the list contain only one element, set external pointer to NULL
1. Otherwise go on traversing the last and set next field of second last node to NULL



head=tail=NULL



```
#include<stdio.h>
#include<conio.h>
struct node
{
    int data;
    struct node *prev,*next;
};
void main()
{
    int no;
    typedef struct node NODE;
    NODE *head=NULL,tail=NULL,*temp;
    if(head==NULL)
    {
        printf("Deletion is not possible");
    }
    else if(head->next==NULL)
    {
        temp=start;
        head=tail=NULL;
    }
}
```

```

        printf("Deleted element is: %d",temp->data);
        free(temp);
    }
    else
    {
        temp=tail;
        tail=tail->prev;
        free(temp);
    }
    if(head==NULL)
    {
        printf("No elements");
    }
    else
    {
        printf("\nElements are:");
        for(p=head;p!=NULL;p=p->next)
        {
            printf(" %d",p->data);
        }
    }

    getch();
}

```

To write a program to implement doubly linked list.

- 1. Insert at beginning**
- 2. Insert at end**
- 3. Delete from beginning**
- 4. Delete from end**
- 5. Display**

```

#include<stdio.h>
#include<conio.h>
struct node
{
    int data;
    struct node *prev,*next;
};
void main()
{
    int ch,no;
    typedef struct node NODE;
    NODE *head=NULL,tail=NULL,*temp,*p,*t;
    clrscr();
    while(1)
    {

```



```

printf("\nMENU\n1.Insert at beginning\n2.Insert at end"
      "\n3.Delete from beginning\n4.Delete from end\n5.Display\n6.Exit");
printf("\nEnter your choice: ");
scanf("%d",&ch);
switch(ch)
{
    case 1:
        temp=(NODE*)malloc(sizeof(NODE));
printf("Enter the no: ");
scanf("%d",&no);
temp->data=no;
        if(start==NULL)
        {
            temp->prev=NULL;
            temp->next=NULL;
            head=temp;
        }
        else
        {
            temp->next=head;
            head->prev=temp;
            temp->prev=NULL;
            head=temp;
        }
        break;
    case 2:
        temp=(NODE*)malloc(sizeof(NODE));
printf("Enter the no: ");
scanf("%d",&no);
temp->data=no;
        if(start==NULL)
        {
            temp->prev=NULL;
            temp->next=NULL;
            head=temp;
        }
        else
        {
            tail->next=temp;
            temp->prev=tail;
            temp->next=NULL;
            tail=temp;
        }
        break;
    case 3:
        if(head==NULL)

```

```

    {
        printf("Deletion is not possible");
    }
else if(head->next==NULL)
{
    temp=start;
    head=tail=NULL;
    printf("Deleted element is: %d",temp->data);
    free(temp);
}
else
{
    temp=head;
    head=temp->next;
    head->prev=NULL;
    printf("Deleted element is: %d",temp->data);
    free(temp);
}
break;
case 4:
if(head==NULL)
{
    printf("Deletion is not possible");
}
else if(head->next==NULL)
{
    temp=start;
    head=tail=NULL;
    printf("Deleted element is: %d",temp->data);
    free(temp);
}
else
{
    temp=tail;
    tail=tail->prev;
    free(temp);
}

break;
case 5:
if(start==NULL)
{
    printf("No elements");
}
else
{

```

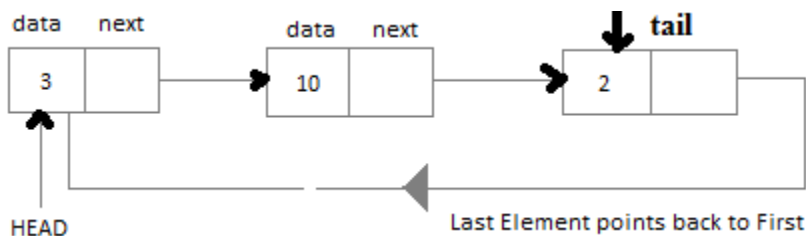
```

        printf("\nElements are:");
        for(p=head;p!=NULL;p=p->next)
        {
            printf(" %d",p->data);
        }
    }
    break;
case 6:
    exit(0);
}
getch();
}
}

```

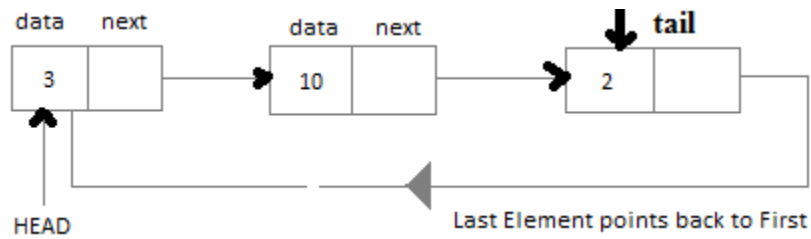
CIRCULAR LINKED LIST

Circular Linked List is little more complicated linked data structure. In the circular linked list we can insert elements anywhere in the list where as in the array we cannot insert element anywhere in the list because it is in the contiguous memory. In the circular linked list the previous element stores the address of the next element and the last element stores the address of the starting element. The elements points to each other in a circular way which forms a circular chain. The circular linked list has a dynamic size which means the memory can be allocated when it is required. A circular linked list has no end. Therefore, it is necessary to establish the head

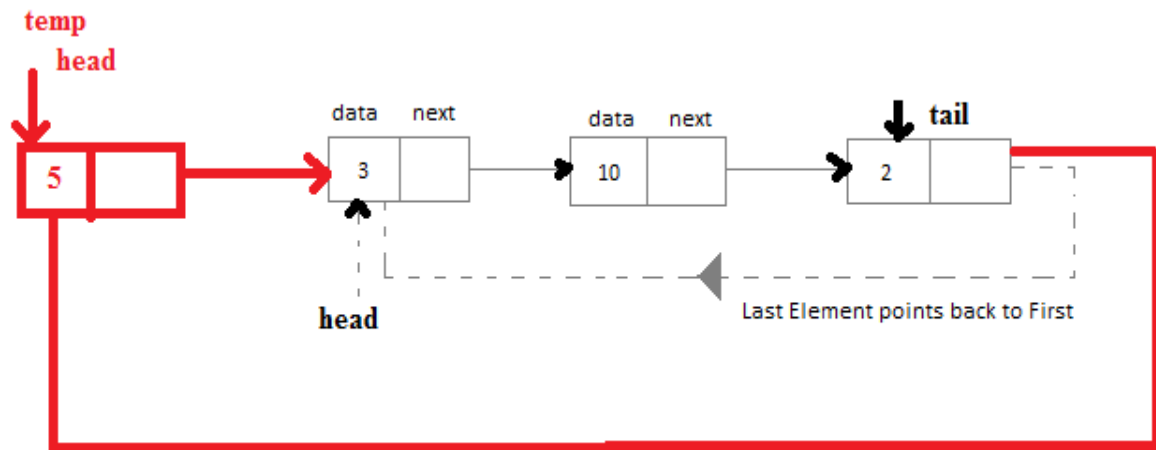


Inserting a node at the beginning

1. Allocate a memory for new node
2. If list is empty then head and tail point to the new node. And linked field of tail pointed to head.
3. If the list is not empty then
 - a) New node pointed to head
 - b) Head point to new node
 - c) Linked field of tail pointed to head



After insertion



We declare the structure for the circular linked list in the same way as we declare it for the linear linked lists

```
struct node
{
int data;
struct node *next;
};
typedef struct node NODE;
NODE *head=NULL;
NODE *tail=NULL;
```

```
-----
NODE *temp;
temp=(struct NODE*)malloc(sizeof(NODE));
printf("Enter the element to be inserted")
scanf("%d",&item);
```

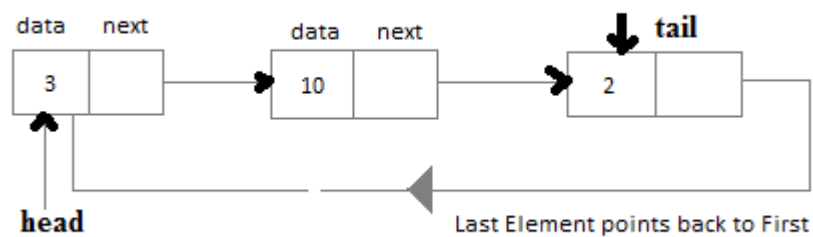
```

temp->data=item;
if(head==NULL)
{
    head=tail=temp;
    tail->next=head;
}
else
{
    temp->next=head;
    head=temp;
    tail->next=head;
}

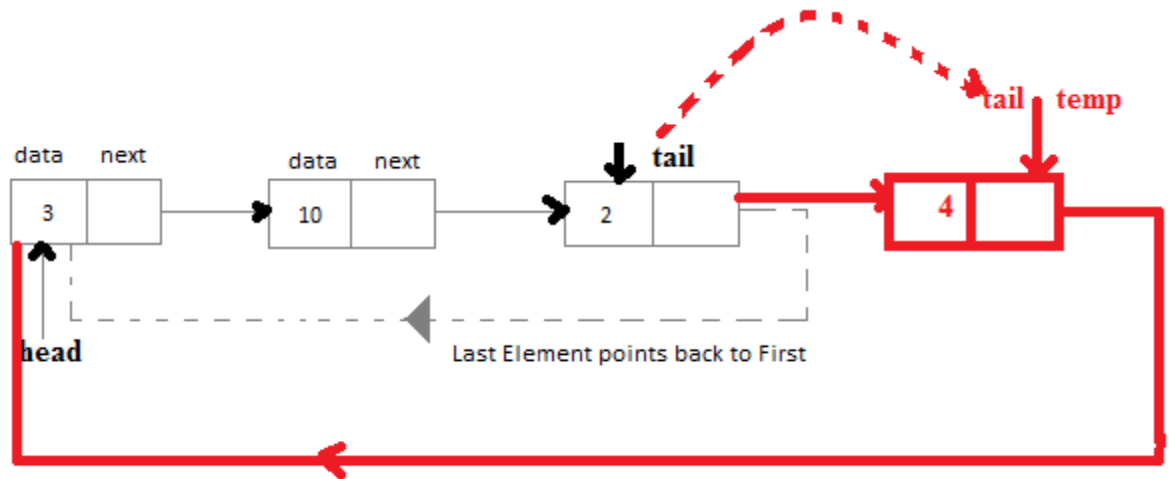
```

Inserting a node at the End

1. Allocate a memory for new node
2. If list is empty then head and tail point to the new node. And linked field of tail pointed to head.
3. If the list is not empty then
 - a) Linked field of tail point to the new node
 - b) tail point to new node



After insertion



```

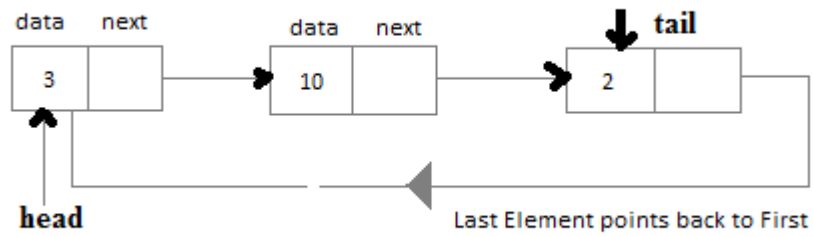
NODE *temp;
temp=(struct NODE*)malloc(sizeof(NODE));
printf("Enter the element to be inserted")
scanf("%d",&item);
temp->data=item;
if(head==NULL)
{
    head=tail=temp;
}

else
{
    tail->next=temp;
    tail=temp;
    tail->next=head;
}

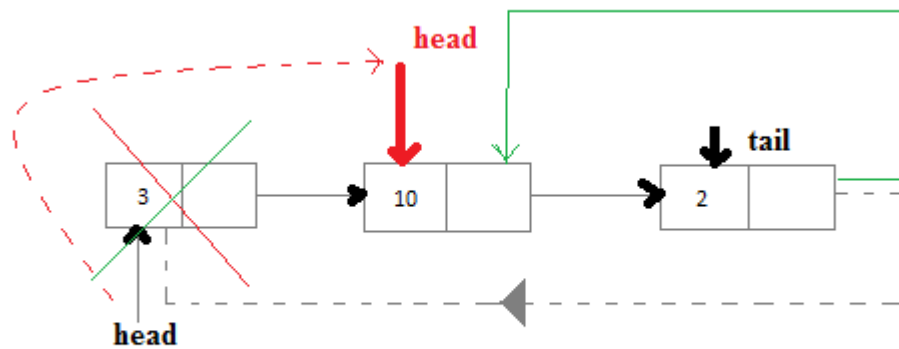
```

Delete a node from the beginning

1. If list is empty then print deletion is not possible
2. If the list contain only on element the set head and tail to NULL
3. If the list contain more than one element then
 - a) temp pointer point to the head node
 - b) head move to the next node
 - c) Linked field of tail pointed to head



After Deletion

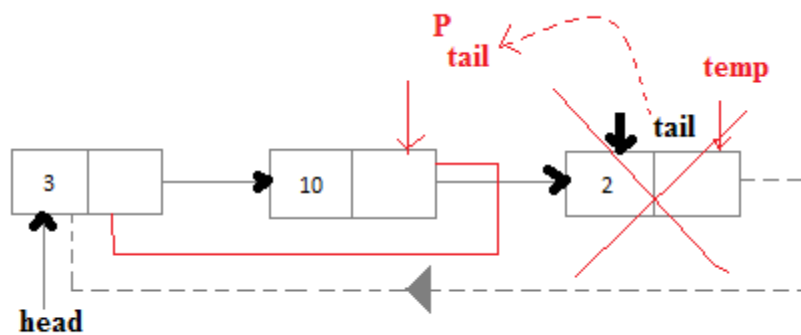
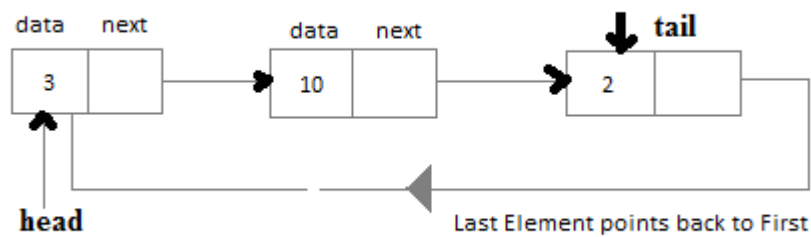


```
{
printf("Deletion is not possible\n");
}
else if(head==tail)    //List contains only one node
{
    free(head);
    head=tail=NULL;
}
else
{
    temp=head;
    head=head->next;
    tail->next=head;
    free(temp);
}
```

}

Delete a node from the beginning

1. If list is empty then print deletion is not possible
2. If the list contain only on element the set head and tail to NULL
3. If the list contain more than one element then
 - a) 'p' pointer point to the second last node and temp pointer points to the last node
 - b) tail move to the p
 - c) Linked field of tail pointed to head
 - d) Delete the last node



```

if(head==NULL)    //List is empty
{
printf("Deletion is not possible\n");
}
else if(head==tail)    //List contains only one node
{
free(head);

```



```

    head=tail=NULL;
}
else
{
    p=head;
    while(p->next!=tail)
    {
        p=p->next;
    }
    temp=p->next;
    tail=p;
    tail->next=head;
    free(temp);
}

```

Application of linked list-Polynomial

Polynomial Addition

Linked list are widely used to represent and manipulate polynomials. Polynomials are the expressions containing number of terms with nonzero coefficient and exponents. In the linked representation of polynomials, each term is considered as a node. And such a node contains three fields

- Coefficient field
- Exponent field
- Link field

The coefficient field holds the value of the coefficient of a term and the exponent field contains the exponent value of the term. And the link field contains the address of the next term in the polynomial. The polynomial node structure is

Coefficient(coeff)	Exponent(expo)	Address of the next node(next)

Algorithm

Two polynomials can be added. And the steps involved in adding two polynomials are given below

1. Read the number of terms in the first polynomial P

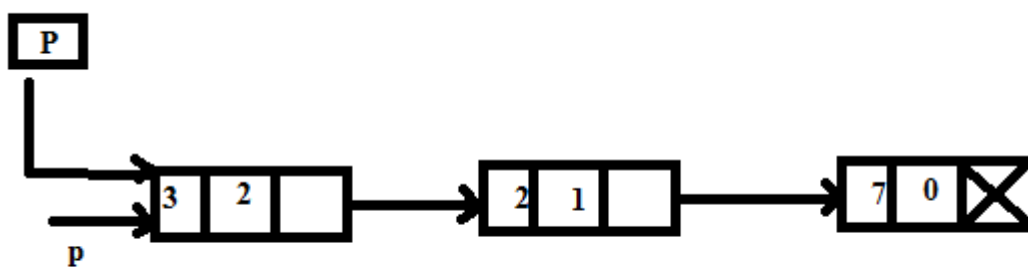
2. Read the coefficient and exponent of the first polynomial
3. Read the number of terms in the second polynomial Q
4. Read the coefficient and exponent of the second polynomial
5. Set the temporary pointers p and q to travers the two polynomials respectively
6. Compare the exponents of two polynomials starting from the first nodes
 - a) If both exponents are equal then add the coefficient and store it in the resultant linked list
 - b) If the exponent of the current term in the first polynomial P is less than the exponent of the current term of the second polynomial then added the second term to the resultant linked list. And, move the pointer q to point to the next node in the second polynomial Q.
 - c) If the exponent of the current term in the first polynomial P is greater than the exponent of the current term in the second polynomial Q, then the current term of the first polynomial is added to the resultant linked list. And move the pointer p to the next node.
 - d) Append the remaining nodes of either of the polynomials to the resultant linked list.

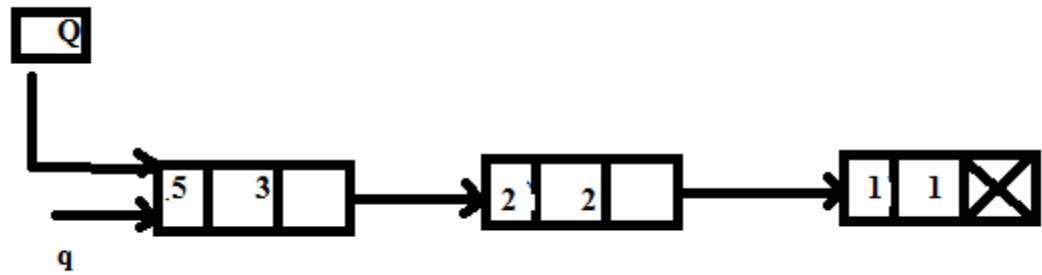
Let us illustrate the way the two polynomials are added. Let p and q be two polynomials having three terms each.

$$P=3x^2+2x+7$$

$$Q=5x^3+2x^2+x$$

These two polynomial can be represented as



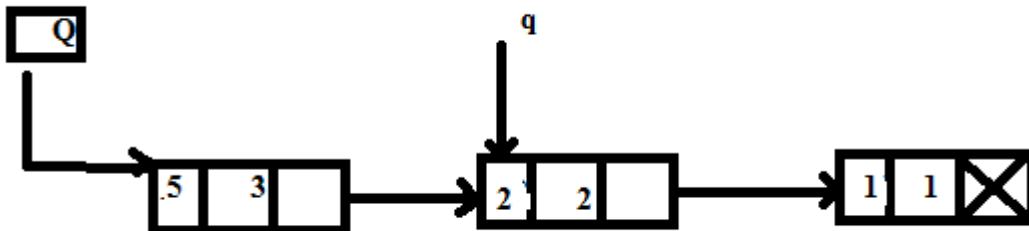
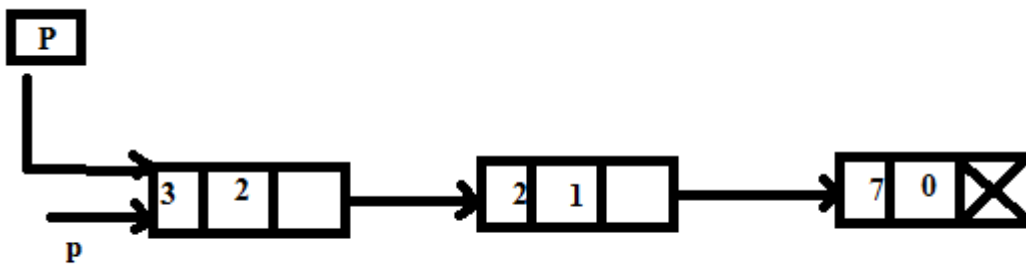


Step 1. Compare the exponent of p and the corresponding exponent of q. Here,

$$\text{expo}(p) < \text{expo}(q)$$

So, add the terms pointed to by q to the resultant list. And now advance the q pointer.

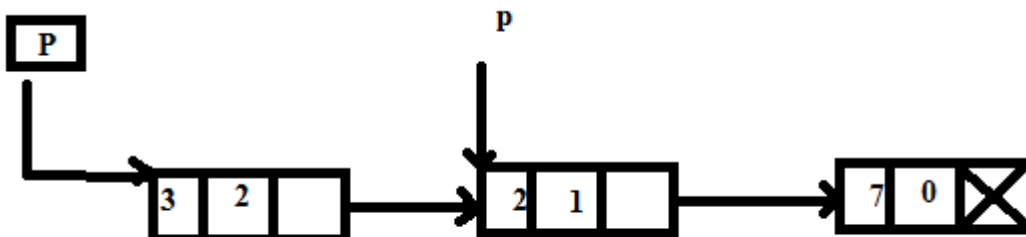
Step 2.

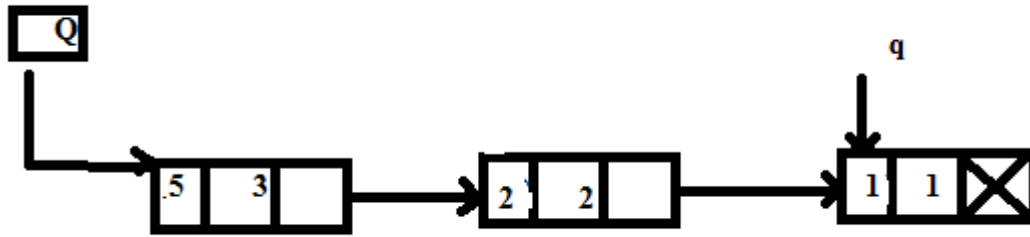


Compare the exponent of the current terms. Here,

$$\text{expo}(p) = \text{expo}(q)$$

So, add the coefficients of these two terms and link this to the resultant list. And, advance the pointers p and q to their next nodes.



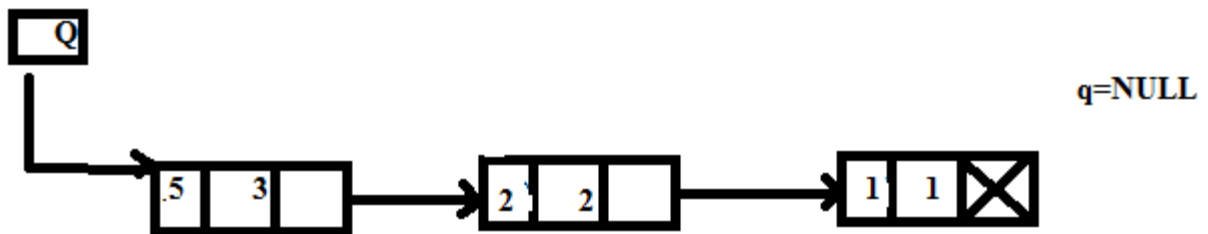
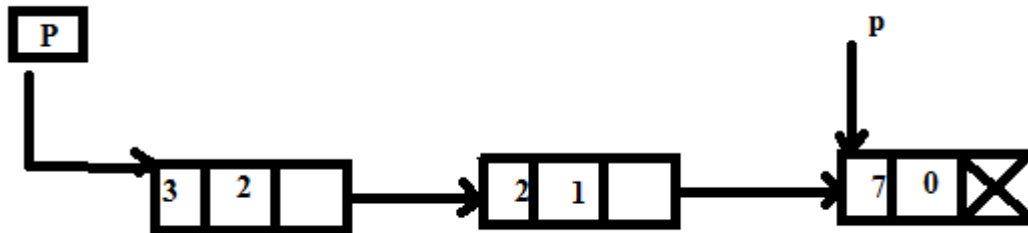


Compare the exponents of the current terms again

$$\text{expo}(p) = \text{expo}(q)$$

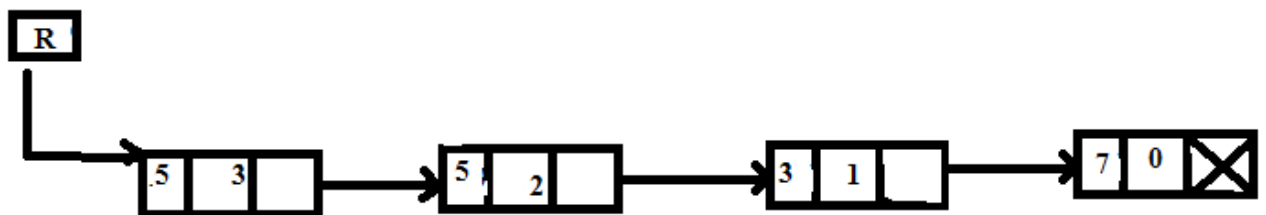
So, add the coefficients of these two terms and link this to the resultant linked list. And, advance the pointers to their next nodes. Q reaches the NULL and p points the last node.

Step 4.



There is no node in the second polynomial to compare with. So, the last node in the first polynomial is added to the end of the resultant linked list.

Step 5. Display the resultant linked list. The resultant linked list is pointed to by the pointer R



Algorithm for Polynomial Multiplication

1. Read the number of terms in the first polynomial
2. Read the coefficient and exponent of the first polynomial
3. Read the number of terms in the second polynomial
4. Read the coefficient and exponent of the second polynomial
5. if one of the list is empty then the nonempty linked list is added to the resultant linked list
Otherwise goto step 6.
6. for each term of the first list
 - a) multiply each term of the second linked list with a term of the first linked list
 - b) add the new term to the resultant polynomial
 - c) reposition the pointer to the starting of the second linked list
 - d) go to the next node
 - e) adds a term to the polynomial in the descending order of the exponent
7. Display the resultant linked list

Module 3 DS Note

Applications of linked list (continued): Memory management, memory allocation and de-allocation. First-fit, best-fit and worst-fit allocation schemes
Implementation of Stacks and Queues using arrays and linked list, DEQUEUE (double ended queue). Multiple Stacks and Queues, Applications.

Application of Linked List –Memory Management

Memory Management with Bitmaps:

When memory is assigned dynamically, the operating system must manage it. With a bitmap, memory is divided up into allocation units, perhaps as small as a few words and perhaps as large as several kilobytes. Corresponding to each allocation unit is a bit in the bitmap, which is 0 if the unit is free and 1 if it is occupied.

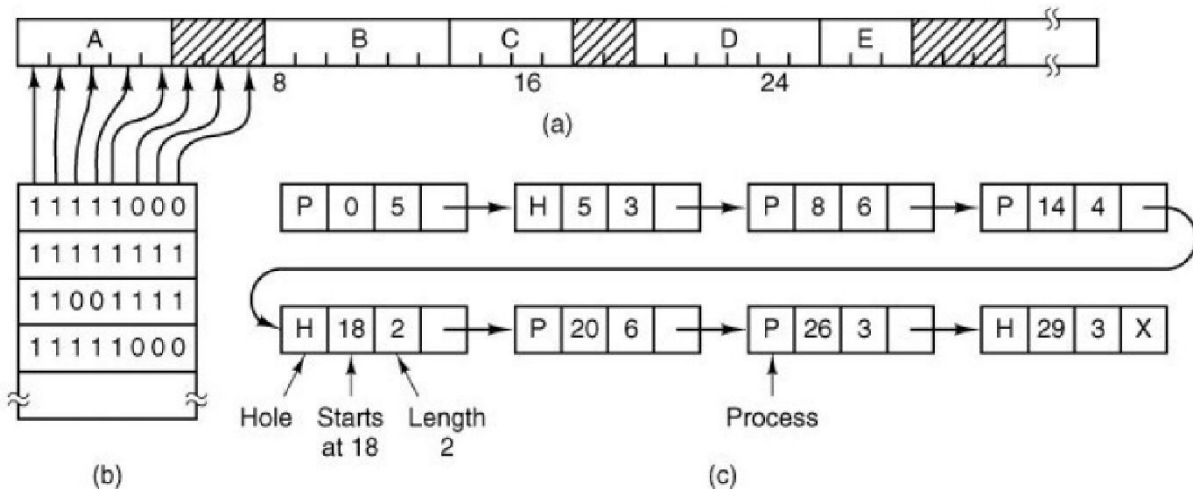
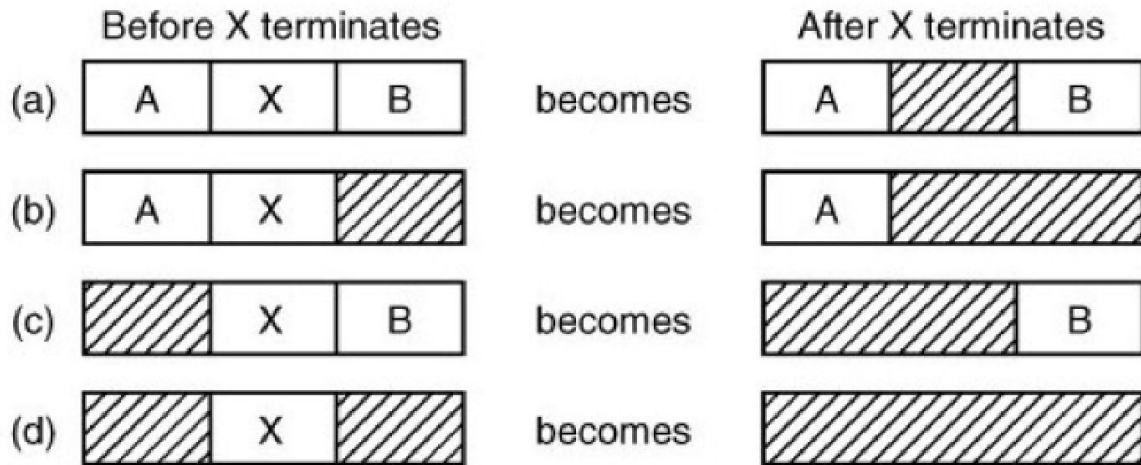


Fig:(a) A part of memory with five processes and three holes. The tick marks show the memory allocation units. The shaded regions (0 in the bitmap) are free. (b) The corresponding bitmap. (c) The same information as a list.

The size of the allocation unit is an important design issue. The smaller the allocation unit, the larger the bitmap. A bitmap provides a simple way to keep track of memory words in a fixed amount of memory because the size of the bitmap depends only on the size of memory and the size of the allocation unit. The main problem with it is that when

Module 3 DS Note

it has been decided to bring a k unit process into memory, the memory manager must search the bitmap to find a run of k consecutive 0 bits in the map. Searching a bitmap for a run of a given length is a slow operation. Memory Management with Linked Lists
Another way of keeping track of memory is to maintain a linked list of allocated and free memory segments, where a segment is either a process or a hole between two processes.



Each entry in the list specifies a hole (H) or process (P), the address at which it starts, the length, and a pointer to the next entry. In this example, the segment list is kept sorted by address. Sorting this way has the advantage that when a process terminates or is swapped out, updating the list is straightforward. A terminating process normally has two neighbors (except when it is at the very top or very bottom of memory). These may be either processes or holes, leading to the four combinations shown in fig above. When the processes and holes are kept on a list sorted by address, several algorithms can be used to allocate memory for a newly created process (or an existing process being swapped in from disk). We assume that the memory manager knows how much memory to allocate.

First Fit: The simplest algorithm is first fit. The process manager scans along the list of segments until it finds a hole that is big enough. The hole is then broken up into two pieces, one for the process and one for the unused memory, except in the statistically

Module 3 DS Note

unlikely case of an exact fit. First fit is a fast algorithm because it searches as little as possible.

Next Fit: It works the same way as first fit, except that it keeps track of where it is whenever it finds a suitable hole. The next time it is called to find a hole, it starts searching the list from the place where it left off last time, instead of always at the beginning, as first fit does.

Best Fit: Best fit searches the entire list and takes the smallest hole that is adequate. Rather than breaking up a big hole that might be needed later, best fit tries to find a hole that is close to the actual size needed.

Worst Fit: Always take the largest available hole, so that the hole broken off will be big enough to be useful. Simulation has shown that worst fit is not a very good idea either.

STACK

A stack is a list of elements in which an element may be inserted or deleted only at one end, called the top of the stack. Stacks are sometimes known as LIFO (last in, first out) lists.

As the items can be added or removed only from the top i.e. the last item to be added to a stack is the first item to be removed.

The two basic operations associated with stacks are:

- *Push:* is the term used to insert an element into a stack.
- *Pop:* is the term used to delete an element from a stack.

Module 3 DS Note

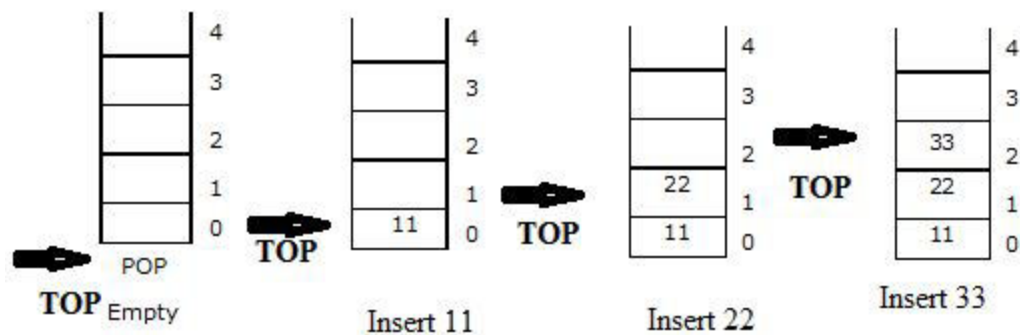
“Push” is the term used to insert an element into a stack. “Pop” is the term used to delete an element from the stack.

All insertions and deletions take place at the same end, so the last element added to the stack will be the first element removed from the stack. When a stack is created, the stack base remains fixed while the stack top changes as elements are added and removed. The most accessible element is the top and the least accessible element is the bottom of the stack.

Representation of Stack:

Let us consider a stack with 6 elements capacity. This is called as the size of the stack. The number of elements to be added should not exceed the maximum size of the stack. If we attempt to add new element beyond the maximum size, we will encounter a *stack overflow* condition. Similarly, you cannot remove elements beyond the base of the

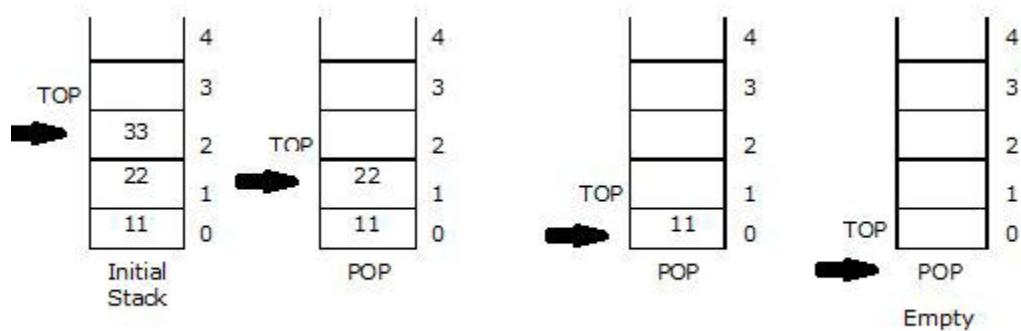
When an element is added to a stack, the operation is performed by push(). Figure shows the creation of a stack and addition of elements using push().



Module 3 DS Note

Push operations on stack

When an element is taken off from the stack, the operation is performed by pop(). Figure shows a stack initially with three elements and shows the deletion of elements using pop().



Pop operations on stack

Algorithm for push and pop

2. Set top=-1
3. Read choice ch
4. If ch=1(PUSH), then goto 4.a else 4.b
 - a. If top<4 then
Read the item, set the top=top+1
Set stack[top]=item
 - b. Else Print stack overflow
5. If ch=2(POP), then goto 5.a else 5.b
 - a. if top \geq 0 then
Set item=stack[top]
top=top-1
Print number deleted is item

Module 3 DS Note

- b. Else Print stack overflow
- 6. If ch=3, then
Display the elements in the stack
- 7. If ch=4, then
Exit
- 8. Stop

PROGRAM

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int stack[10],ch,i,item,top=-1;
    clrscr();
    while(1)
    {
        printf("\nMENU:\n1.Push\n2.Pop\n3.Traverse\n4.Exit\nEnter your choice: ");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1:
                if(top<=9)
                {
                    printf("\nStack overflow");
                }
            else
            {

```

Module 3 DS Note

```
printf("\nEnter the item: ");
scanf("%d",&item);
top=top+1;
stack[top]=item;
}

break;
case 2:
    if(top== -1)
        printf("\nStack underflow");

    else
    {
        item=stack[top];

        printf("\nDeleted item is %d\n",item);
    }

    break;
case 3:
    if(top>=0)
        printf("\nNo elements in stack");

    else
    {
        printf("\nElements are: ");
        for(i=top;i>=0;i--)
```

Module 3 DS Note

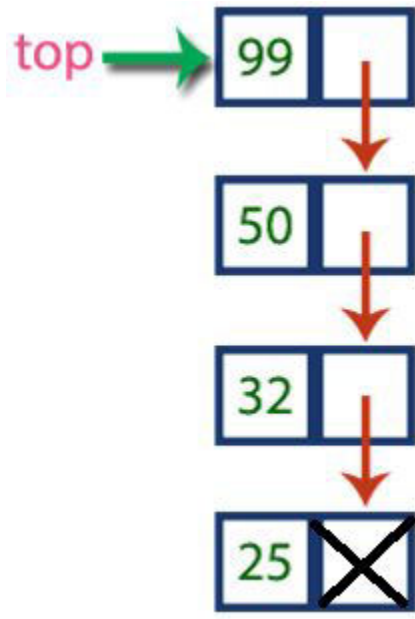
```
        printf(" %d",stack[i]);  
    }  
  
    break;  
case 4:  
    exit(0);  
}  
getch();  
}  
}
```

Stack Using Linked LIST

The major problem with the stack implemented using array is, it works only for fixed number of data values. That means the amount of data must be specified at the beginning of the implementation itself. Stack implemented using array is not suitable, when we don't know the size of data which we are going to use. A stack data structure can be implemented by using linked list data structure. The stack implemented using linked list can work for unlimited number of values. That means, stack implemented using linked list works for variable size of data. So, there is no need to fix the size at the beginning of the implementation. The Stack implemented using linked list can organize as many data values as we want.

In linked list implementation of a stack, every new element is inserted as '**top**' element. That means every newly inserted element is pointed by '**top**'. Whenever we want to remove an element from the stack, simply remove the node which is pointed by '**top**' by moving '**top**' to its next node in the list. The **next** field of the first element must be always **NULL**.

Module 3 DS Note



In above example, the last inserted node is 99 and the first inserted node is 25. The order of elements inserted is 25, 32, 50 and 99.

push(value) - Inserting an element into the Stack

We can use the following steps to insert a new node into the stack...

- Step 1: Create a newNode with given value.
- Step 2: Check whether stack is Empty ($\text{top} == \text{NULL}$)
- Step 3: If it is Empty, then set $\text{newNode} \rightarrow \text{next} = \text{NULL}$.
- Step 4: If it is Not Empty, then set $\text{newNode} \rightarrow \text{next} = \text{top}$.
- Step 5: Finally, set $\text{top} = \text{newNode}$.

pop() - Deleting an Element from a Stack

We can use the following steps to delete a node from the stack...

Module 3 DS Note

- Step 1: Check whether stack is Empty (top == NULL).
- Step 2: If it is Empty, then display "Stack is Empty!!! Deletion is not possible!!!" and terminate the function
- Step 3: If it is Not Empty, then define a Node pointer 'temp' and set it to 'top'.
- Step 4: Then set 'top = top → next'.
- Step 7: Finally, delete 'temp' (free(temp)).

Program

```
#include<stdio.h>
#include<conio.h>
struct node
{
    int data;

void main()
{
    typedef struct node NODE;
    NODE *top=NULL,*temp,*t;
    int ch,item;
    clrscr();
    while(1)
    {
        printf("\nMENU:\n1.Push\n2.Pop\n3.Display\n4.Exit\nEnter your choice: ");
        scanf("%d",&ch);
        switch(ch)
        {
```

Module 3 DS Note

case 1:

```
temp=(NODE*)malloc(sizeof(NODE));
printf("\nEnter the item: ");
scanf("%d",&item);
temp->data=item;
if(top==NULL)
{
    temp->ptr=NULL;
    top=temp;
}
else
{
    temp->ptr=top;
    top=temp;
}
```

break;

case 2:

```
if(top==NULL)
    printf("\nDeletion is not possible");
else if(top->ptr==NULL)
{
    temp=top;
    top=NULL;
    printf("\nPopped item is %d",temp->data);
    free(temp);
}
else
```


Module 3 DS Note

```
{
    temp=top;
    top=top->ptr;
    printf("\nPoped item is %d",temp->data);
    free(temp);
}
break;
case 3:
    if(start==NULL)
        printf("\nStack is empty");
    else
    {
        printf("\nElements are:");
        for(t=top;t!=NULL;t=t->ptr)

    }
    break;
case 4:
    exit(0);
default:
    printf("\nWrong Choice");
    break;
}
getch();
}
}
```

Module 3 DS Note

Application of Stack

1.Stack Frames

Programs compiled from C make use of a stack frame for the working memory of each procedure or function invocation. When any procedure or function is called, a number of words – the stack frame –is push onto a program stack. When the procedure or function returns, this frame of data is popped off the stack. For example, when a program sends parameters to a function, the parameters are placed ON THE STACK. When the function completes its execution these parameters are popped off from the stack. When a function calls other function the current contents of the caller function are pushed onto the stack with the address of instruction just next to the call instruction, this is done so that after the execution of called function, the compiler can track back the path from where it is sent to the called function.

2. Reversing a String

exploited to reverse strings of line of characters. This can be simply thought of simply as pushing the individual characters, and when the complete line is pushed onto the stack, then individual characters of the line are popped off. Because the last inserted character pushed on stack would be the first character to be popped off, the line obviously be removed.

3. Converting INFIX to POSTFIX

The rules to be remembered during infix to postfix conversion are:

1. Parenthesize the expression starting from left to right.
2. During parenthesizing the expression, the operands associated with operator having higher precedence are first parenthesized.
3. The sub-expression which has been converted into postfix is to be treated as single operand
4. Once the expression is converted to postfix from remove the parenthesis.

Module 3 DS Note

Convert the $A*B+C/D$ to postfix form

$(A*B)+(C/D)$

$AB* + (C/D)$

$AB* + CD/$

$AB*CD/+$

Infix to postfix Conversion

Suppose Q is an arithmetic expression written in notation. This algorithm finds the equivalence postfix expression P

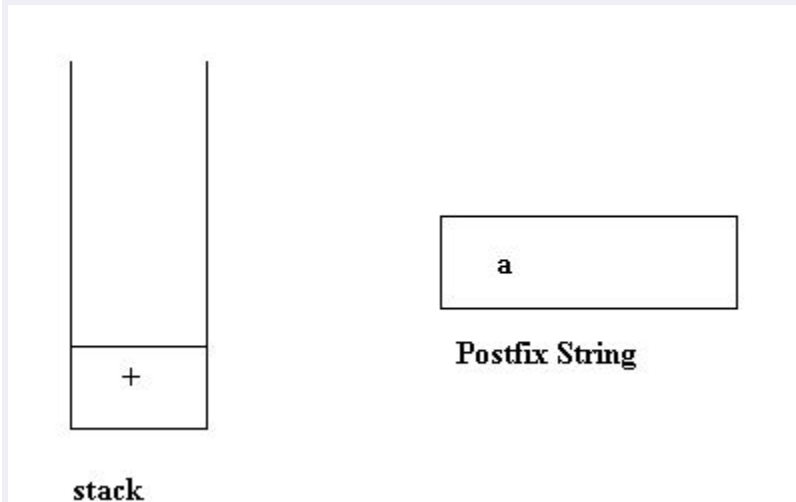
1. Scan Q from left to right and repeat step 2 to 5 for each element of Q until the STACK is empty
2. If an operand is encountered, add it to P.
3. If a left parenthesis is encountered, push it onto STACK.
 - a) Repeatedly pop from the STACK and add to P each operator (on the top of the STACK) which has the same precedence or higher precedence than \odot
 - b) Add \odot to STACK
5. If a right parenthesis is encountered, then
 - a) Repeatedly pop from the STACK and add to P each operator until a left parenthesis is encountered
 - b) Remove the left parenthesis [Do not add left parenthesis to P]
6. Exit

Let us see how the above algorithm will be implemented using an example.

Infix String : $a+b*c-d$

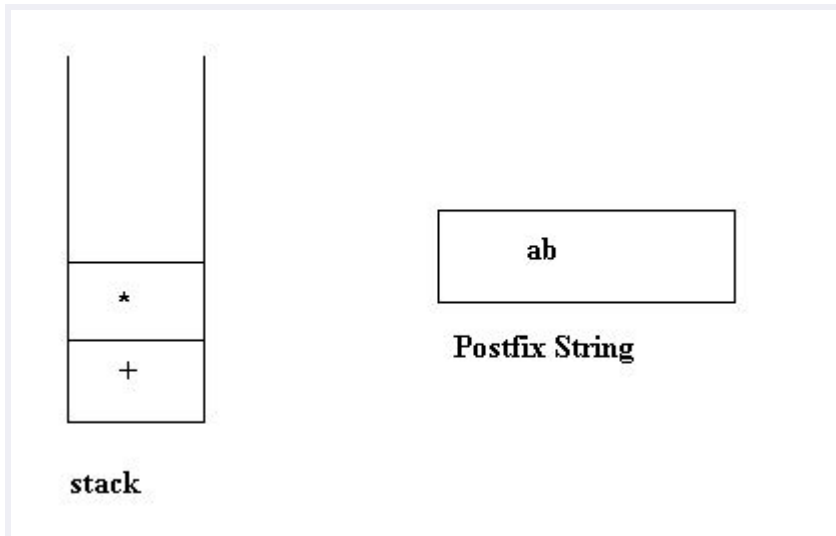
Module 3 DS Note

Initially the Stack is empty and our Postfix string has no characters. Now, the first character scanned is 'a'. 'a' is added to the Postfix string. The next character scanned is '+'. It being an operator, it is pushed to the stack.

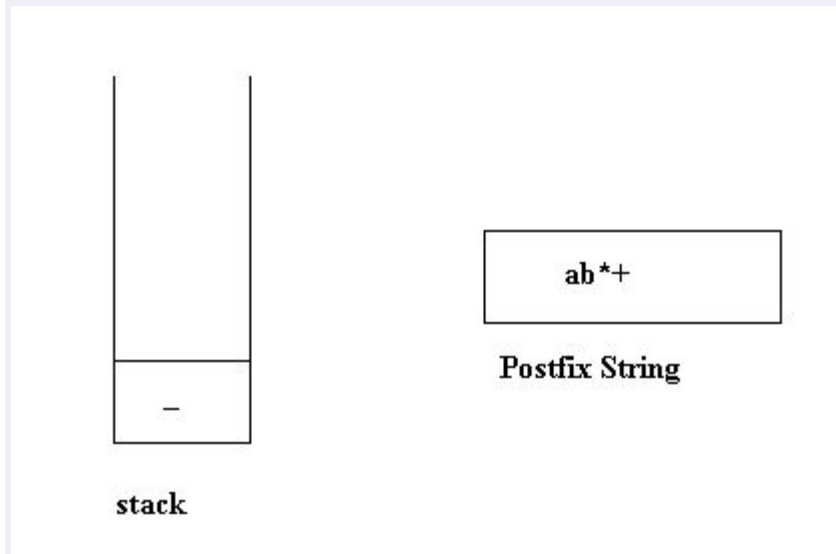


Next character scanned is 'b' which will be placed in the Postfix string. Next character is '*' which is an operator. Now, the top element of the stack is '+' which has lower precedence than '*', so '*' will be pushed to the stack.

Module 3 DS Note

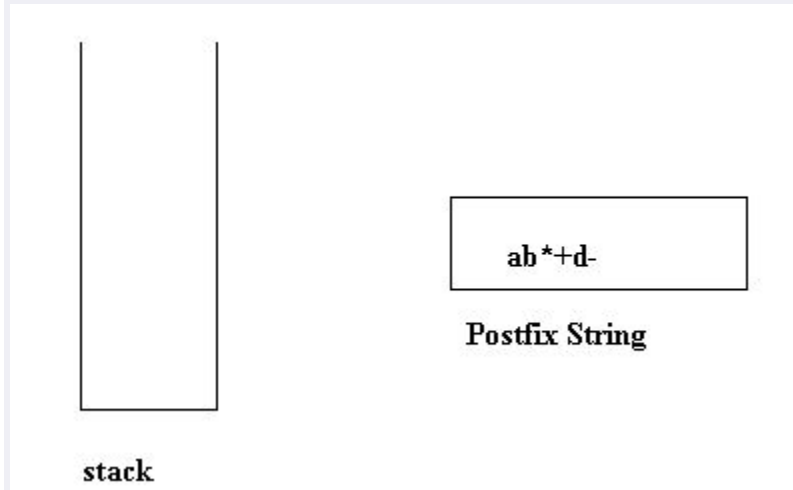


The next character is 'c' which is placed in the Postfix string. Next character scanned is '-'. The topmost character in the stack is '*' which has a higher precedence than '-'. Thus '*' will be popped out from the stack and added to the Postfix string. Even now the stack is not empty. Now the add it to the Postfix string. The '-' will be pushed to the stack.



Module 3 DS Note

Next character is 'd' which is added to Postfix string. Now all characters have been scanned so we must pop the remaining elements from the stack and add it to the Postfix string. At this stage we have only a '-' in the stack. It is popped out and added to the Postfix string. So, after all characters are scanned, this is how the stack and Postfix string will be :



End result :

- Infix String : $a+b*c-d$
- Postfix String : $abc*+d-$

Postfix evaluation

The reason to convert infix to postfix expression is that we can compute the answer of postfix expression easier by using a stack.

This algorithm finds the VALUE of the arithmetic expression P written in postfix notation. The following

Module 3 DS Note

algorithm, which uses a STACK to hold operands, evaluates P.

1. Scan P from left to right and repeat step 2 and 3 for each element of P until the end of the string.
2. If an operand is encountered, put it on stack.
3. If an operator \odot is encountered, then
 - a) Remove the two top elements of STACK, where A is the top element and B is the next top element
 - b) Evaluate $B \odot A$
 - c) Place the result of (b) back on \STACK[End of IF structure]
4. Set VALUE equal to the top element on stack

Example :

Let us see how the above algorithm will be implemented using an example.

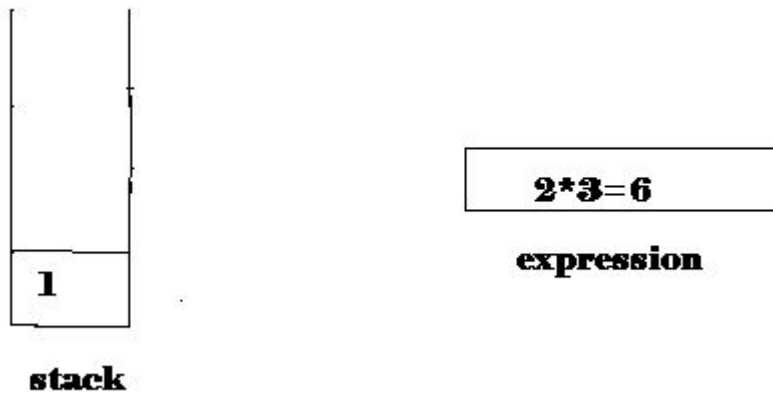
Postfix String : 123*+4-

Initially the Stack is empty. Now, the first three characters scanned are 1,2 and 3, which are operands. Thus they will be pushed into the stack in that order.

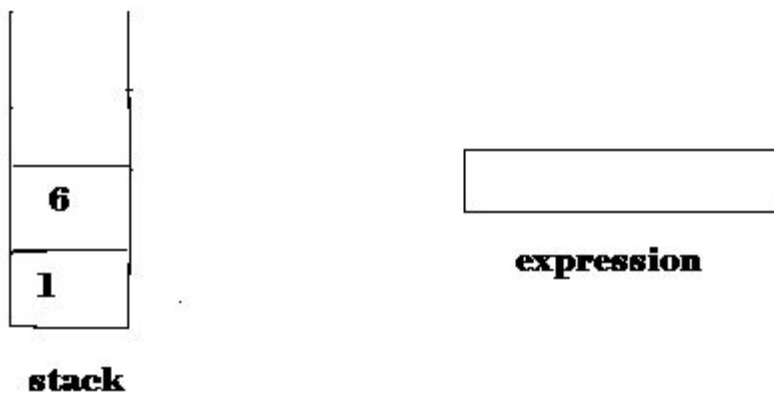


Module 3 DS Note

Next character scanned is "*", which is an operator. Thus, we pop the top two elements from the stack and perform the "*" operation with the two operands. The second operand will be the first element that is popped.

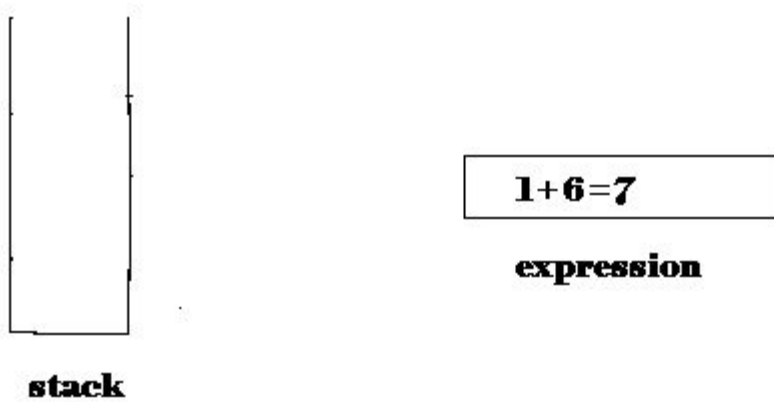


The value of the expression($2*3$) that has been evaluated(6) is pushed into the stack.

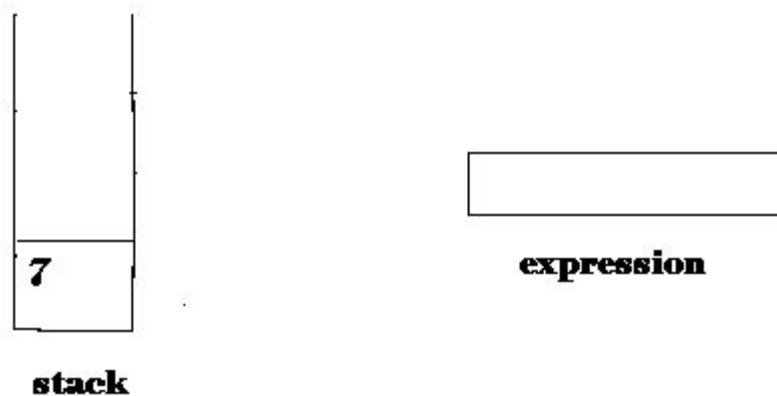


Module 3 DS Note

Next character scanned is "+", which is an operator. Thus, we pop the top two elements from the stack and perform the "+" operation with the two operands. The second operand will be the first element that is popped.

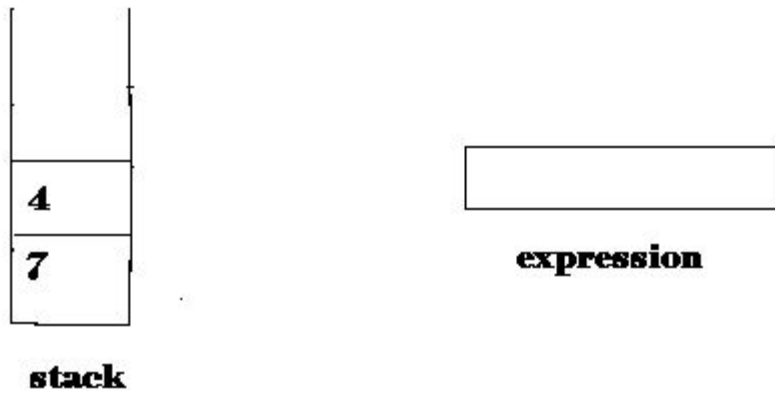


The value of the expression(1+6) that has been evaluated(7) is pushed into the stack.

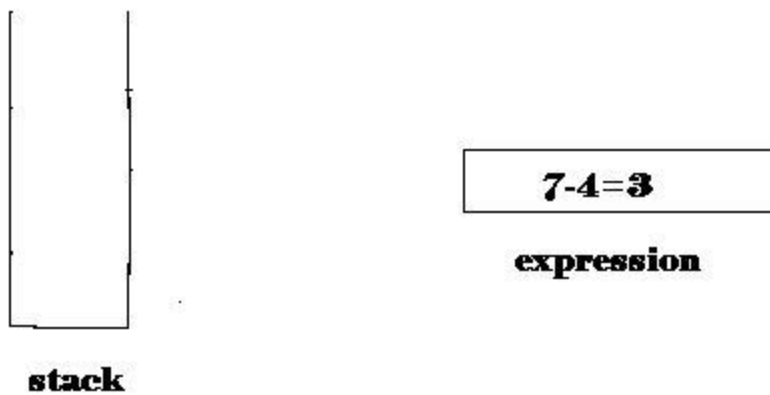


Next character scanned is "4", which is added to the stack.

Module 3 DS Note

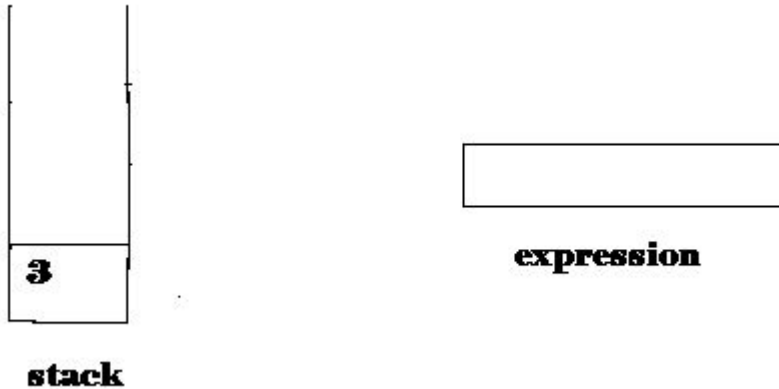


Next character scanned is "-", which is an operator. Thus, we pop the top two elements from the stack and perform the "-" operation with the two operands. The second operand will be the first element that is popped.



The value of the expression(7-4) that has been evaluated(3) is pushed into the stack.

Module 3 DS Note



Now, since all the characters are scanned, the remaining element in the stack (there will be only one element in the stack) will be returned.

End result :

- Postfix String : 123*+4-
- Result : 3

Queue

Basic features of Queue

1. Like Stack, Queue is also an ordered list of elements of similar data types.
2. Queue is a FIFO(First in First Out) structure.
3. Once a new element is inserted into the Queue, all the elements inserted before the new element in the queue must be removed, to remove the new element.
4. **peek()** function is often used to return the value of first element without deleting an element.

Queue using array

Module 3 DS Note

A queue data structure can be implemented using one dimensional array. But, queue implemented using array can store only fixed number of data values. The implementation of queue data structure using array is very simple, just define a one dimensional array of specific size and insert or delete the values into that array by using FIFO (First In First Out) principle with the help of variables 'front' and 'rear'. Initially both 'front' and 'rear' are set to -1. Whenever, we want to insert a new value into the queue, increment 'rear' value by one and then insert at that position. Whenever we want to delete a value from the queue, then increment 'front' value by one and then display the value at 'front' position as deleted element.

Queue Operations

Enqueue-Insert value into the queue

1. Start
2. If queue is full ($\text{rear} \geq \text{size} - 1$) then queue is full otherwise goto step 3
 - a) set $\text{front} = \text{rear} = 0$
 - b) insert element into the rear position
4. Increment rear and insert the element into the rear
5. Stop

Dequeue- delete an element from queue

1. Start
2. If queue is empty ($(\text{front} = -1 \text{ and } \text{rear} = -1)$ or $(\text{front} = \text{rear} + 1)$) then print queue is empty otherwise goto step 3
3. Increment the front value by one ($\text{front}++$). Then display $\text{queue}[\text{front}]$ as deleted element
4. Stop

Program

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

Module 3 DS Note

```
#include<conio.h>

#include<stdlib.h>

int main()

{

    int ch, front=-1, rear=-1, q[10], item, i;

    while(1)

    {

        printf("\n\t QUEUE\n\n 1. Insert \n 2. Delete \n 3. Display \n 4. Exit\n Enter ur choice: ");

        scanf("%d", &ch);

        switch(ch)

        {

            case 1:

                if(rear==9)

                {

                    printf("Queue is full ! ");

                }

                else

                {

                    if(front== -1 && rear== -1)

                    {
```

Module 3 DS Note

```
        front=0; rear=0;

    }

    else

        rear++;

    printf("Enter the inserted item : ");

    scanf("%d", &item);

    q[rear]=item;

}

break;

case 2:

    if((front==-1 && rear==-1) || front==rear+1)

    {

        printf("Queue is empty!");

    }

    else

    {

        item=q[front];

        front++;

        printf("Deleted item is : %d", item);

    }
```

Module 3 DS Note

```
    }

    break;

case 3:

    if((front==-1 && rear==-1) || front==rear+1)

    {

        printf("Queue is empty ! ");

    }

    else

    {

        printf("\nElements are :");

        for(i=front; i<=rear; i++)

            printf("%d \t", q[i]);

    }

    break;

case 4:

    exit(0);

    break;

}

}

getch();
```

Module 3 DS Note

}

Queue Using Linked List

The major problem with the queue implemented using array is, It will work for only fixed number of data. That means, the amount of data must be specified in the beginning itself. Queue using array is not suitable when we don't know the size of data which we are going to use. A queue data structure can be implemented using linked list data structure. The queue which is implemented using linked list can work for unlimited number of values. That means, queue using linked list can work for variable size of data (No need to fix the size at beginning of the implementation). The Queue implemented using linked list can organize as many data values as we want.

In linked list implementation of a queue, the last inserted node is always pointed by '**rear**' and the first node is always pointed by '**front**'.

To implement queue using linked list, we need to set the following things before implementing actual operations.

- Step 1: Include all the header files which are used in the program. And declare all the user defined functions.
- Step 2: Define a 'Node' structure with two members data and next.
- Step 3: Define two Node pointers 'front' and 'rear' and set both to NULL.
- Step 4: Implement the main method by displaying Menu of list of operations and make suitable function calls in the main method to perform user selected operation.

enQueue(value) - Inserting an element into the Queue

Module 3 DS Note

We can use the following steps to insert a new node into the queue...

- Step 1: Create a newNode with given value and set 'newNode → next' to NULL.
- Step 2: Check whether queue is Empty (rear == NULL)
- Step 3: If it is Empty then, set front = newNode and rear = newNode.
- Step 4: If it is Not Empty then, set rear → next = newNode and rear = newNode.

deQueue() - Deleting an Element from Queue

We can use the following steps to delete a node from the queue...

- Step 1: Check whether queue is Empty (front == NULL).
- Step 2: If it is Empty, then display "Queue is Empty!!! Deletion is not possible!!!" and terminate from the function
- Step 4: Then set 'front = front → next' and delete 'temp' (free(temp)).

display() - Displaying the elements of Queue

We can use the following steps to display the elements (nodes) of a queue...

- Step 1: Check whether queue is Empty (front == NULL).
- Step 2: If it is Empty then, display 'Queue is Empty!!!' and terminate the function.
- Step 3: If it is Not Empty then, define a Node pointer 'temp' and initialize with front.
- Step 4: Display 'temp → data --->' and move it to the next node. Repeat the same until 'temp' reaches to 'rear' (temp → next != NULL).

Module 3 DS Note

- Step 4: Finally! `_Display 'temp → data ---> NULL'`.

Program

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

```
struct node
```

$$\{$$

```
int data;
```

```
struct node *ptr;
```

$$\};$$

```
int main()
```

$$\{$$

```
typedef struct node NODE;
```

```
NODE *front=NULL,*rear=NULL, *temp, *p;
```

```
int i, ch, pos, item;
```

```
while(1)
```

$$\{$$

```
printf("\n\n\tQueue Using Linked List\n1.Insert\n2.Delete\n3.Display\n4.   Exit\n\nEnter your\nchoice: ");
```

```
scanf("%d", &ch);
```

switch(ch)

Module 3 DS Note

```
{  
  
    case 1:  
  
        temp=(NODE*)malloc(sizeof(NODE));  
  
        printf("Enter the data to be inserted: ");  
  
        scanf("%d", &temp->data);  
  
        if(rear==NULL)  
        {  
            temp->ptr=NULL;  
            front=rear=temp;  
        }  
        else  
        {  
            p=front;  
            while(p!=rear)  
            {  
                p=p->ptr;  
            }  
            p->ptr=temp;  
            temp->ptr=NULL;
```

Module 3 DS Note

```
    rear=temp;

}

break;

case 2:

    if(front==NULL)

    {

        printf("No elements to delete!");

    }

    else

    {

        if(front->ptr==NULL)

        {

            temp=front;

            item=temp->data;

            printf("Deleted element is %d", temp->data);

            free(temp);

            front=rear=NULL;

        }

        else

        {
```

Module 3 DS Note

```
temp=front;

item=temp->data;

front=front->ptr;

free(temp);

printf("Deleted element: %d", item);

}

}

break;

case 3:

if(front==NULL)

{

printf("No elements");

}

else

{

printf("\nElements are: ")

p=front;

while(p!=rear)

{

printf("%d ", p->data);
```

Module 3 DS Note

```
        p=p->ptr;

    }

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

}

break;

case 4:

    exit(0);

}

}

getch();

}
```

Applications of Queue

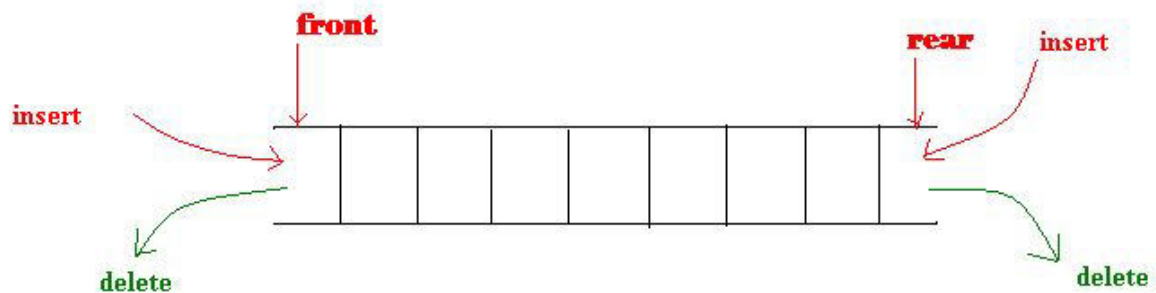
Queue, as the name suggests is used whenever we need to have any group of objects in an order in which the first one coming in, also gets out first while the others wait for their turn, like in the following scenarios :

1. Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
2. In real life, Call Center phone systems will use Queues, to hold people calling them in an order, until a service representative is free.
3. Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive, First come first served.

Module 3 DS Note

Double ended Queue (Deque)

Double Ended Queue is also a Queue data structure in which the insertion and deletion operations are performed at both the ends (front and rear). That means, we can insert at both front and rear positions and can delete from both front and rear positions.



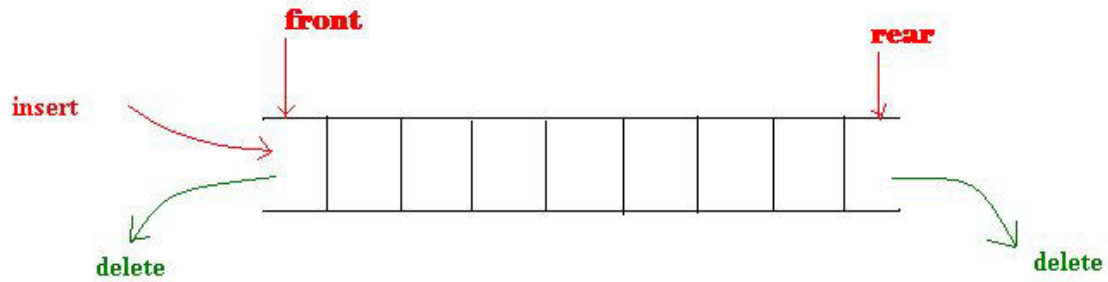
Double Ended Queue can be represented in TWO ways, those are as follows...

1. Input Restricted Double Ended Queue
2. Output Restricted Double Ended Queue

Input Restricted Double Ended Queue

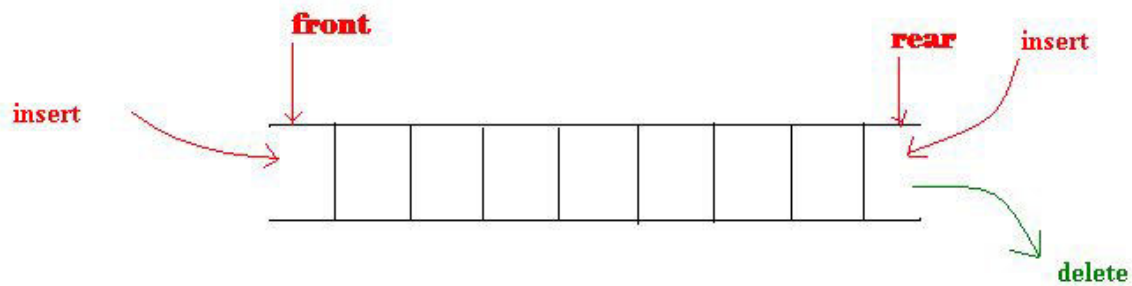
In input restricted double ended queue, the insertion operation is performed at only one end and deletion operation is performed at both the ends.

Module 3 DS Note



Output Restricted Double Ended Queue

In output restricted double ended queue, the deletion operation is performed at only one end and insertion operation is performed at both the ends.



Algorithm for Insertion at rear end

Step -1: [Check for overflow]

```
if(rear==MAX)
```

```
Print("Queue is Overflow");
```


Module 3 DS Note

return;

Step-2: [Insert element]

else

rear=rear+1;

q[rear]=no;

[Set rear and front pointer]

if rear=-1

rear=front=0;

Step-1 : [Check for the front position]

if(front==0)

•

Print (“Cannot add item at front end”);

return;

Step-2 : [Insert at front]

else

Module 3 DS Note

```
front=front-1;
```

```
q[front]=no;
```

Step-3 : Return

Algorithm for Deletion from front end

Step-1 [Check for front pointer]

```
if front=-1
```

```
    print(" Queue is Underflow");
```

```
    return;
```

Step-2 [Perform deletion]

```
    print("Deleted element is",no);
```

```
    [Set front and rear pointer]
```

```
    if front=rear
```

```
        front=rear=-1;
```

```
    else
```

```
        front=front+1;
```

Module 3 DS Note

Step-3 : Return

Algorithm for Deletion from rear end

Step-1 : [Check for the rear pointer]

```
if rear=0  
  
    print("Cannot delete value at rear end");  
  
    return;
```

Step-2: [perform deletion]

```
else  
  
  
  
  
  
  
  
  
  
if front= rear  
  
    front=rear=-1;  
  
else  
  
    rear=rear-1;  
  
    print("Deleted element is",no);
```

Step-3 : Return

Multiple Stacks

Module 3 DS Note

This C Program Implements two Stacks using a Single Array & Check for Overflow & Underflow. A Stack is a linear data structure in which a data item is inserted and deleted at one record. A stack is called a Last In First Out (LIFO) structure. Because the data item inserted last is the data item deleted first from the stack.

To implement two stacks in one array, there can be two methods.

First is to divide the array in to two equal parts and then give one half to each stack. But this method wastes space.

So a better way is to let the two stacks to push elements by comparing tops of each other, and not up to one half of the array.

```
#include<stdio.h>
#include<stdlib.h>
void main()
{
int n,top1,top2,ch=1,a,i,arr[100];
printf("Enter size of array you want to use\n");
scanf("%d",&n);
top1=-1;
top2=n/2;
while(ch!=0)
{
printf("What do u want to do?\n");
printf("1.Push element in stack 1\n");
printf("2.Push element in stack 2\n");
printf("3.Pop element from stack 1\n");
printf("4.Pop element from stack 2\n");
```

Module 3 DS Note

```
printf("5.Display stack 1\n");
printf("6.Display stack 2\n");
printf("0.EXIT\n");
scanf("%d",&ch);
switch(ch)
{
case 1:
{
printf("Enter the element\n");
scanf("%d",&a);
if(top1==n/2-1)
printf("Stack1 is full");
else
{
top1++;
arr[top1]=a;
}
break;
}
case 2:
{
printf("Enter the element\n");
scanf("%d",&a);
if(top2==n)
printf("Stack 2 is full\n");
else
{
```

Module 3 DS Note

```
top2++;
arr[top2]=a;
}
break;
}
case 3:
{
if(top1==-1)
printf("Stack1 is empty\n");
else
{
a=arr[top1];
top1--;
printf("%d\n",a);
}
break;
}
case 4:
{
if(top2==n/2)
printf("Stack2 is empty\n");
else
{
a=arr[top2];
top2--;
printf("%d\n",a);
}
```

Module 3 DS Note

```
break;
}
case 5:
{
if(top1==-1)
printf("Stack1 is empty\n");
else
{
printf("Stack1 is-->\n");
for(i=0;i<=top1;i++)
printf("%d ",arr[i]);
printf("\n");
}
break;
}
case 6:
{
if(top2==n/2)
printf("Stack2 is empty\n");
else
{
printf("Stack2 is-->\n");
for(i=(n/2+1);i<=top2;i++)
printf("%d ",arr[i]);
printf("\n");
}
break;
```

Module 3 DS Note

```
}  
case 0:  
break;  
}  
}  
}
```

Multiple Queues

We can maintain two queues in the same array which is possible. If one grows from position 0 of the array and the other grows from the last position, queue refers to the data structure where the elements are stored such that a new value is inserted at the rear end of the queue and deleted at the front end of the queue.

In order to preserve two queues, there should be two front and two rear of the two queues. Both the queues can grow upto any extent from 0th position to maximum. Hence there should

```
front=-1 rear=-1
```

```
front2=n/2-1 rear2=n/2-1
```

Insert into First Queue

```
if(rear1==n/2-1)  
{  
printf("First Queue is full\n");  
}  
else
```


Module 3 DS Note

```
{  
  
printf("Enter the element to be inserted");  
  
scanf("%d",&item);  
  
If((front1==-1)&&(rear1==-1))  
  
{  
  
front1=rear1=0;  
  
}  
  
else  
  
{  
  
rear1++;  
  
}  
  
Q[rear1]=item;  
  
}
```

Insert into Second Queue

```
if(rear2==n/-1)  
  
{  
  
printf("Second Queue is full\n");  
  
}  
  
else
```

Module 3 DS Note

```
{  
  
printf("Enter the element to be inserted");  
  
scanf("%d",&item);  
  
If((front2==n/2-1)&&(rear1==n/2-1))  
  
{  
  
front2=rear2=n/2;  
  
}  
  
else  
  
{  
  
rear2++;  
  
}  
  
Q[rear2]=item;  
  
}
```

Delete from First Queue

```
if(front1== -1)  
  
{  
  
printf("First Queue is empty\n");  
  
}  
  
else
```

Module 3 DS Note

```
{  
  
item=Q[front1];  
  
printf("The deleted element in the first queue is %d",item);  
  
If((front1==rear1)  
  
{  
  
front1=rear1=-1;  
  
}  
  
else  
  
{  
  
front1++;  
  
}  
  
}
```

Delete from Second Queue

```
if(front2==n/2-1)  
  
{  
  
printf("Second Queue is empty\n");  
  
}  
  
else  
  
{
```

Module 3 DS Note

```
item=Q[front2];  
  
printf("The deleted element in the first queue is %d",item);  
  
If((front2==rear2)  
  
{  
  
front2=rear2=n/2-1;  
  
}  
  
else  
  
{  
  
front2++;  
  
}  
  
}
```

Module 3 DS Note

Module 3 DS Note

String: - representation of strings, concatenation, substring searching and deletion.

Trees: - m-ary Tree, Binary Trees – level and height of the tree, complete-binary tree representation using array, tree traversals (Recursive and non-recursive), applications. Binary search tree – creation, insertion and deletion and search operations, applications.

String Representation

In C programming, array of characters is called a string. A string is terminated by a null character /0.

For example:

Here, "c string tutorial" is a string. When, compiler encounter strings, it appends a null character /0 at the end of string.

c		s	t	r	i	n	g		t	u	t	o	r	i	a	l	\0
---	--	---	---	---	---	---	---	--	---	---	---	---	---	---	---	---	----

Declaration of strings

Before we actually work with strings, we need to declare them first.

Strings are declared in a similar manner as arrays. Only difference is that, strings are of char type.

Using arrays

char s[5];

s[0]	s[1]	s[2]	s[3]	s[4]

Initialization of strings

In C, string can be initialized in a number of different ways.

For convenience and ease, both initialization and declaration are done in the same step.

Using arrays

```
char c[] = "abcd";
```

OR,

```
char c[50] = "abcd";
```

OR,

```
char c[] = {'a', 'b', 'c', 'd', '\0'};
```

OR,

```
char c[5] = {'a', 'b', 'c', 'd', '\0'};
```

c[0]	c[1]	c[2]	c[3]	c[4]
a	b	c	d	\0

String concatenation

String is nothing but an array of characters (OR char data types). While doing programming in C language, you may have faced challenges wherein you might want to compare two strings, concatenate strings, copy one string to another & perform various string manipulation operations.

ng.h” header All of such kind of operations plus many more oth

file. In order to use these string functions you must include string.h file in your C program.

In simple words, String is an one dimensional array of characters. This library function concatenates a string onto the end of the other string.i.e, The strcat() function appends s2 to the end of s1. String s2 is unchanged.

WAP to concatenate two string with using string functions

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

```
#include <string.h>
```

```
int main()
```

```
{
```

```
    char a[100], b[100];
```

```
    printf("Enter the first string\n");
```



```

gets(a);

printf("Enter the second string\n");
gets(b);

strcat(a,b);

printf("String obtained on concatenation is %s\n",a);

}

```

WAP to concatenate two string without using string functions

```

#include<stdio.h>
#include<string.h>

int main()
{
    char s1[50], s2[30];
    int i=0,j=0;
    printf("\nEnter String 1 :");
    gets(s1);
    printf("\nEnter String 2 :");
    gets(s2);
    while(s1[i]!='\0')
    {
        i++;
    }
    while(s2[j]!='\0')
    {

```

```

    s1[i]=s2[j];
    i++;
    j++;
}
    s1[i] = '\0';
    printf("\nConcatenated string is :%s", s1);
}

```

Substring searching and deletion

C programming code to check if a given string is present in another string, For example the string "programming" is present in "c programming". If the string is present then it's location (i.e. at which position it is present) is printed.

Substring searching

```

#include<stdio.h>

#include<string.h>

int main()

{

char s1[20],s2[20];

int l,j,m,l1,l;

printf("Enter the first string");

gets(s1);

printf("Enter the second string");

gets(s2);

l=strlen(s1);

l1=strlen(s2);

```

```

for(i=0; i<1; i++)

{

    j=0;

    if(s1[i]==s2[j])

    {

        m=i;

        sign=0;

        while((s2[j]!='\0')&&(sign!=1))

        {

            if(s1[m]==s2[j])

            {

                m++;

                j++;

            }

            else

                sign=1;

        }

    }

    if(sign==0)

    {

        printf("The given substring is present in the location %d", i+1);

    }

```

```
else
```

```
{
```

```
printf("Substring not present");
```

```
}
```

```
}
```

Substring Deletion

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

```
#include<string.h>
```

```
int main()
```

```
{
```

```
char s1[20],s2[20];
```

```
int l,j,m,l1;
```

```
printf("Enter the first string");
```

```
gets(s1);
```

```
printf("Enter the second string");
```

```
gets(s2);
```

```
l=strlen(s1);
```

```
l1=strlen(s2);
```

```
for(i=0;l<l1;i++)
```

```
{
```

```
    j=0;
```

```
    if(s1[i]==s2[j])
```

```
    {
```

```

    m=i;

    sign=0;

while((s2[j]!='\0')&&(sign!=1))

{

    if(s1[m]==s2[j])

    {

        m++;

        j++;

    }

else

    sign=1;

}

}

}

if(sign==0)

{

    printf("The given substring is present in the location %d",i+1);

    m=i+1;

    while(s1[m]!='\0')

    {

        s1[i]=s1[m];

        i++;

```

```

        m++;

    }

    s1[i]='\0';

    printf("Substring after deletion is");

    puts(s1);

}

else

{

    printf("Substring is not present");

}

}

```

Tree

Definition

Tree data structure is a collection of data (Node) which is organized in hierarchical structure and this is a recursive definition

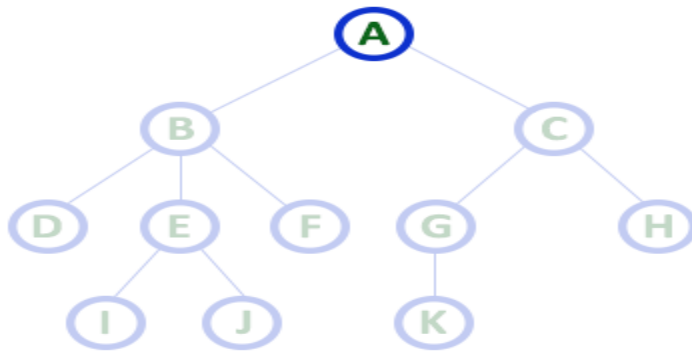
In tree data structure, every individual element is called as Node. Node in a tree data structure, stores the actual data of that particular element and link to next element in hierarchical structure.

Tree Terminologies

In a tree data structure, we use the following terminology...

1. Root

In a tree data structure, the first node is called as **Root Node**. Every tree must have root node. We can say that root node is the origin of tree data structure. In any tree, there must be only one root node. We never have multiple root nodes in a tree.

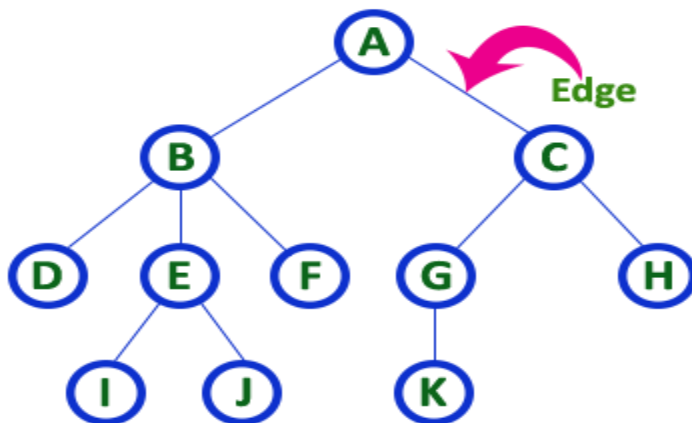


Here 'A' is the 'root' node

- In any tree the first node is called as **ROOT** node

2. Edge

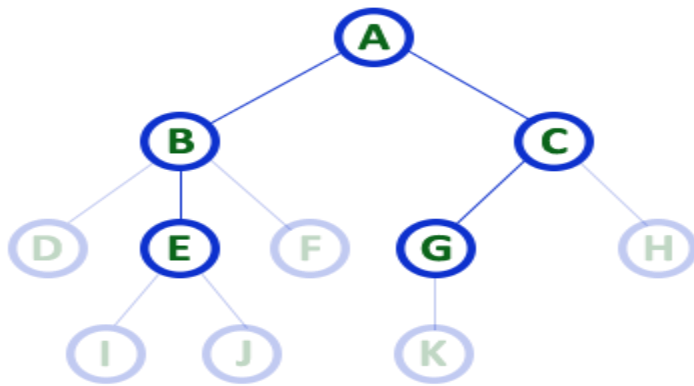
In a tree data structure, the connecting link between any two nodes is called as **EDGE**. In a tree with 'N' number of nodes there will be a maximum of '**N-1**' number of edges.



- In any tree, 'Edge' is a connecting link between two nodes.

3. Parent

In a tree data structure, the node which is predecessor of any node is called as **PARENT NODE**. In simple words, the node which has branch from it to any other node is called as parent node. Parent node can also be defined as "**The node which has child / children**".

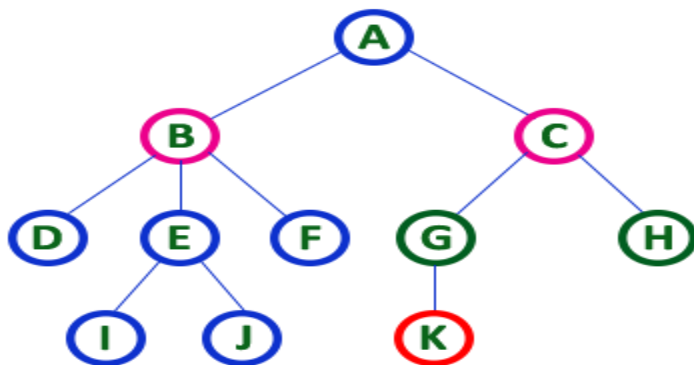


Here A, B, C, E & G are **Parent** nodes

- In any tree the node which has child / children is called '**Parent**'
- A node which is predecessor of any other node is called '**Parent**'

4. Child

In a tree data structure, the node which is descendant of any node is called as **CHILD Node**. In simple words, the node which has a link from its parent node is called as child node. In a tree, any parent node can have any number of child nodes. In a tree, all the nodes except root are child nodes.



Here B & C are **Children of A**

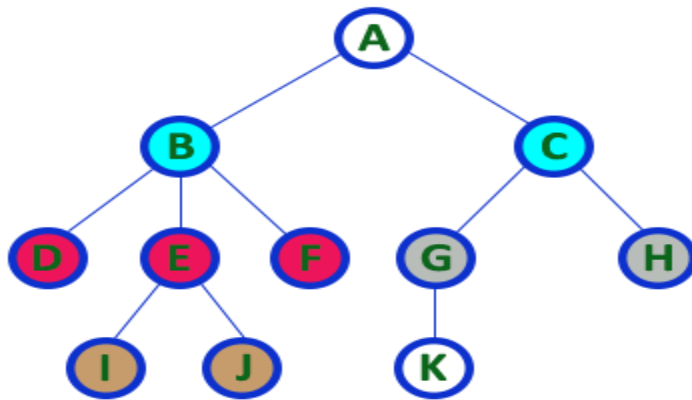
Here G & H are **Children of C**

Here K is **Child of G**

- descendant of any node is called as **CHILD Node**

5. Siblings

In a tree data structure, nodes which belong to same Parent are called as **SIBLINGS**. In simple words, the nodes with same parent are called as Sibling nodes.



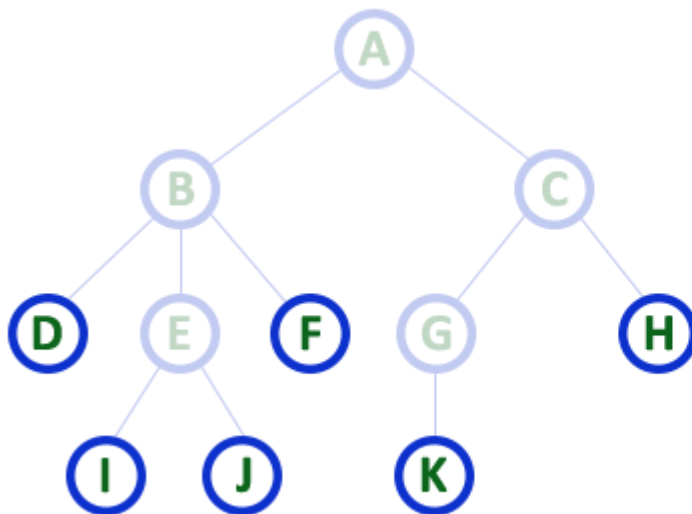
Here **B & C** are **Siblings**
 Here **D E & F** are **Siblings**
 Here **G & H** are **Siblings**
 Here **I & J** are **Siblings**

- In any tree the nodes which has same Parent are called '**Siblings**'
- The children of a Parent are called '**Siblings**'

6. Leaf/Terminal Node

In a tree data structure, the node which does not have a child is called as **LEAF Node**. In simple words, a leaf is a node with no child. The degree of a node is zero then it is a leaf

In a tree data structure, the leaf nodes are also called as **External Nodes**. External node is also a node with no child. In a tree, leaf node is also called as '**Terminal**' node.



Here **D, I, J, F, K & H** are **Leaf** nodes

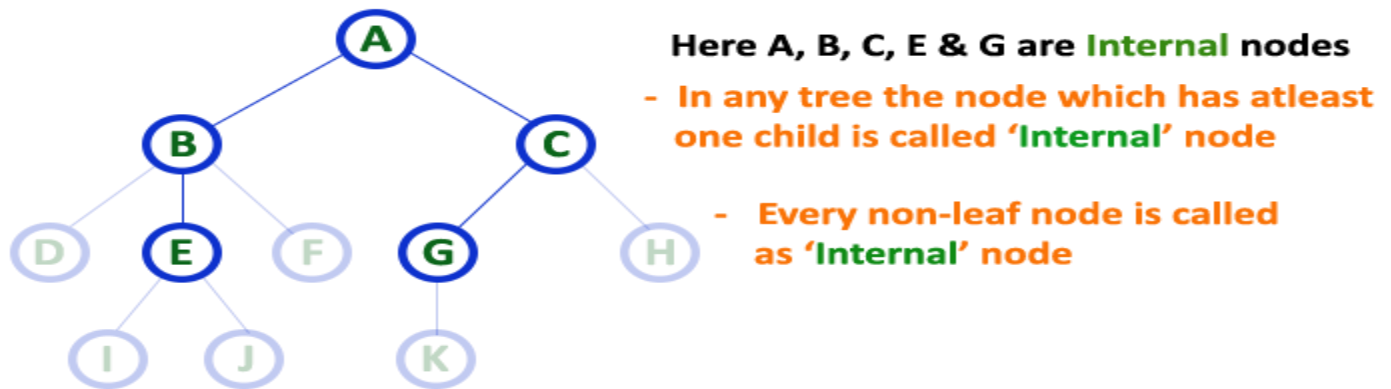
- In any tree the node which does not have children is called '**Leaf**'
- A node without successors is called a '**leaf**' node

7. Internal Nodes/Non Terminal Nodes

In a tree data structure, the node which has atleast one child is called as **INTERNAL Node**. In simple words, an internal node is a node with atleast one child.

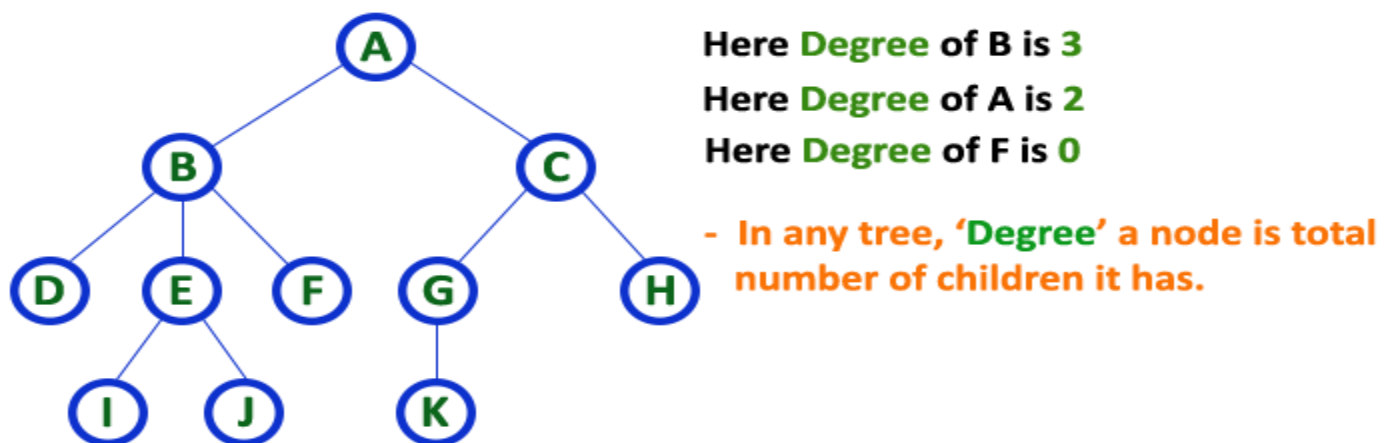
In a tree data structure, nodes other than leaf nodes are called as **Internal Nodes**. The root node is also

said to be **Internal Node** if the tree has more than one node. Internal nodes are also called as 'Non-Terminal' nodes.



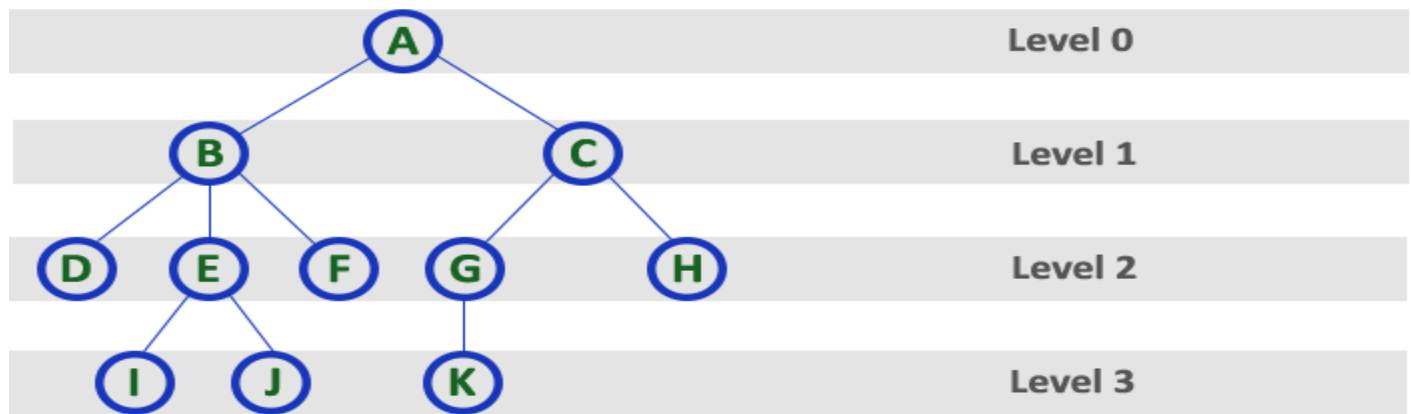
8. Degree

In a tree data structure, the total number of children of a node is called as **DEGREE** of that Node. In simple words, the Degree of a node is total number of children it has. The highest degree of a node among all the nodes in a tree is called as '



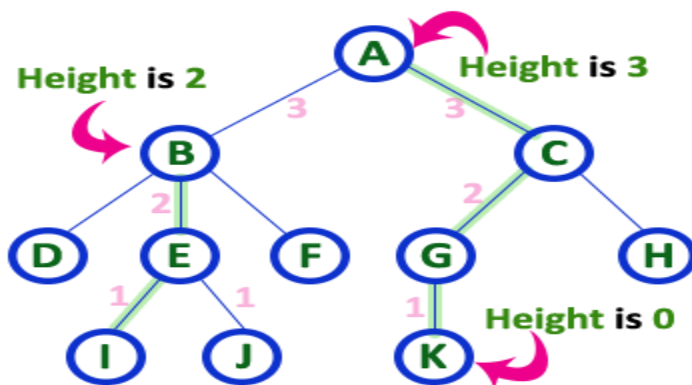
9. Level

In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on... In simple words, in a tree each step from top to bottom is called as a Level and the Level count starts with '0' and incremented by one at each level (Step).



10. Height

In a tree data structure, the total number of edges from leaf node to a particular node in the longest path is called as **HEIGHT** of that Node. In a tree, height of the root node is said to be **height of the tree**. In a tree, **height of all leaf nodes is '0'**.

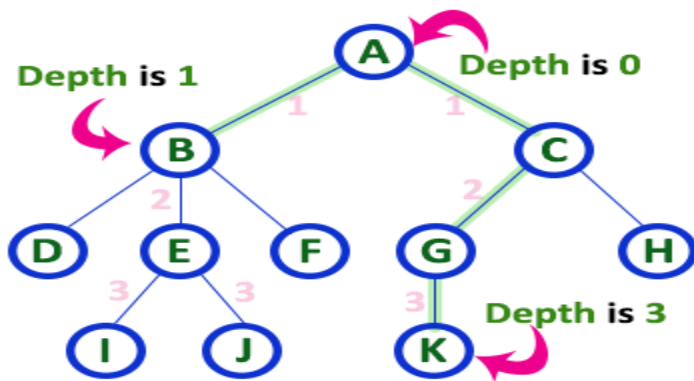


Here Height of tree is 3

- In any tree, 'Height of Node' is total number of Edges from leaf to that node in longest path.
- In any tree, 'Height of Tree' is the height of the root node.

11. Depth

In a tree data structure, the total number of edges from root node to a particular node is called as **DEPTH** of that Node. In a tree, the total number of edges from root node to a leaf node in the longest path is said to be **Depth of the tree**. In simple words, the highest depth of any leaf node in a tree is said to be depth of that tree. In a tree, **depth of the root node is '0'**.

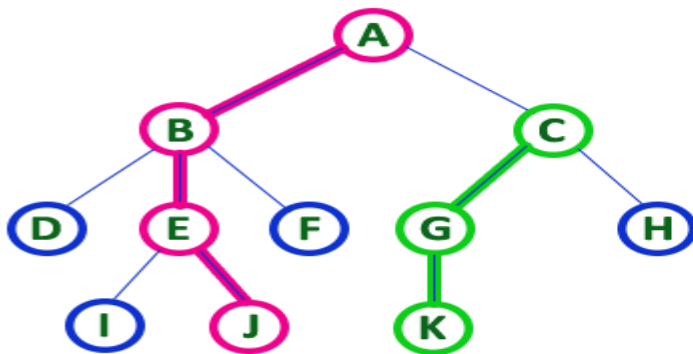


Here Depth of tree is 3

- In any tree, 'Depth of Node' is total number of Edges from root to that node.
- In any tree, 'Depth of Tree' is total number of edges from root to leaf in the longest path.

12. Path

In a tree data structure, the sequence of Nodes and Edges from one node to another node is called as **PATH** between that two Nodes. **Length of a Path** is total number of nodes in that path. In below example the path A - B - E - J has length 4.



- In any tree, 'Path' is a sequence of nodes and edges between two nodes.

Here, 'Path' between A & J is

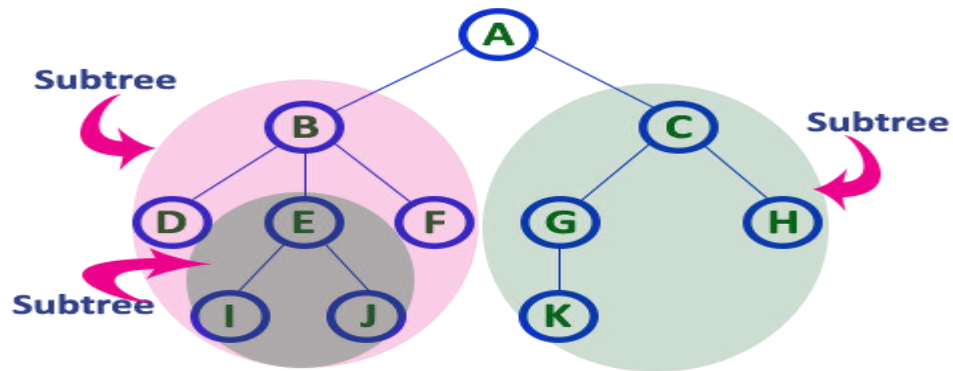
A - B - E - J

Here, 'Path' between C & K is

C - G - K

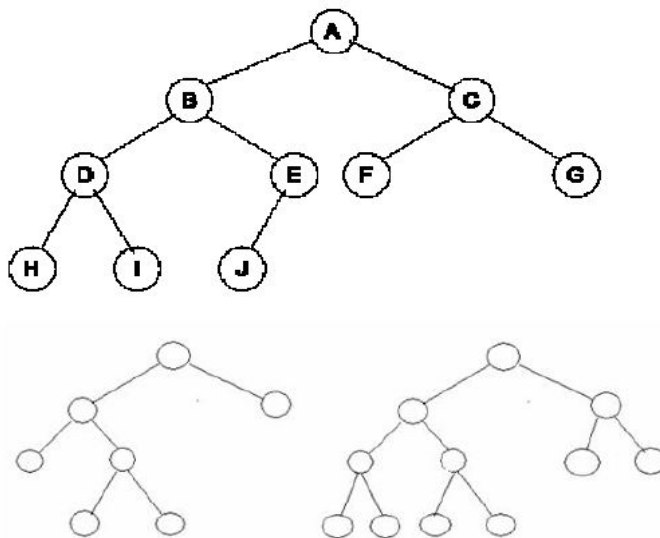
13. Sub Tree

In a tree data structure, each child from a node forms a subtree recursively. Every child node will form a subtree on its parent node.



Binary Tree:

A Tree in which each node has a degree of atmost 2. i.e. it can have either 0,1 or 2 children.

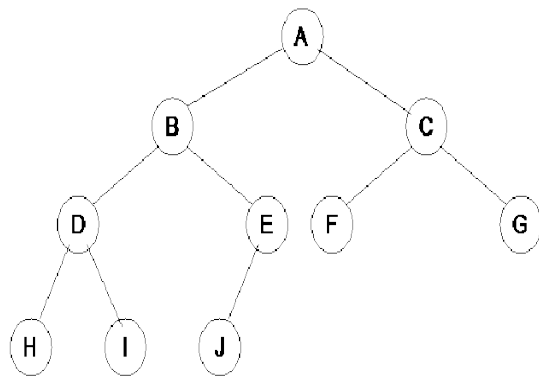


Here, leaves are H, I, J. Except these, remaining internal nodes has atmost 2 nodes as their child.

Complete Binary Tree

- A complete binary tree is a tree that is completely filled, with the possible exception of the bottom level.
- The bottom level is filled from left to right.

A binary tree T with n levels is complete if all levels except possibly the last are completely full, and the last level has all its nodes to the left side. You may find the definition of complete binary tree in the books little bit different from this. A perfectly complete binary tree has all the leaf nodes. In the complete binary tree, all the nodes have left and right child nodes except the bottom level. At the bottom level, you will find the nodes from left to right. The bottom level may not be completely filled, depicting that the tree is not a perfectly complete one. Let's see a complete binary tree in the figure below:

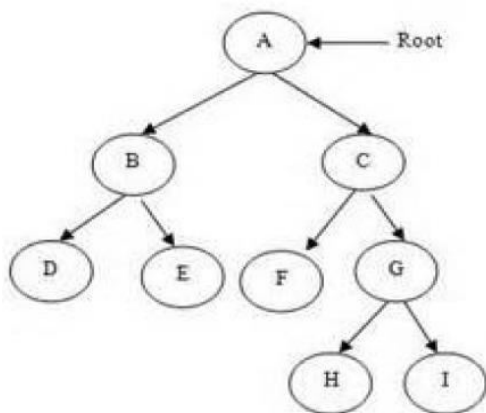


In the above tree, we have nodes as A, B, C, D, E, F, G, H, I, J. The node D has two children at the lowest level whereas node E has only left child at the lowest level that is J. The right child of the node E is missing. Similarly node F and G also lack child nodes. This is a complete binary tree according to the definition given above. At the lowest level, leaf nodes are present from left to right but all the inner nodes have both children. Let's recap some of the properties of complete binary tree.

- A complete binary tree of height h has between 2^h to $2^{h+1} - 1$ nodes.

Full Binary tree:

- Strictly BT either has 0 or 2 subtrees



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 random access a node in a tree. There are three ways which we use to traverse a tree –

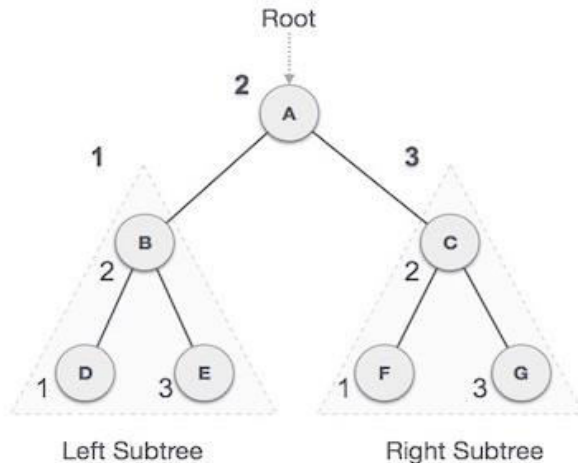
- In-order Traversal
- Pre-order Traversal

- Post-order Traversal

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

Algorithm

Until all nodes are traversed –

Step 1 Traverse the left subtree in Inorder

Step 2 – Visit root node.

Step 3 – Traverse the right subtree in Inorder

Non-Recursive Algorithm

This algorithm traverses the tree non-recursively in Inorder using STACK. Initially push NULL onto stack and then set P=root and the repeat the following steps until NULL is popped from stack

Step 1 – Proceed down the leftmost path rooted at P, pushing each node N onto stack and stopping when a node N with no left child is pushed onto stack

Step 2 – Pop and process the nodes on stack. If NULL is popped then exit. If a node N with a right child P->right is processed, set P=P->right and return to step 1.

```
void intrav(nodeptr tree)
{
    if(tree!=NULL)
    {
        intrav(tree->left);
```

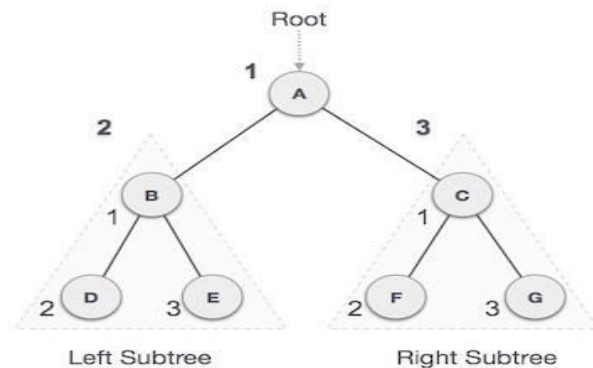
```

    printf("->%d",tree->data);
    intrav(tree->right);
}
}

```

Preorder Traversal

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 –

Algorithm

Until all nodes are traversed –

Step 1 – Visit root node.

Step 2 – Traverse the left subtree in Preorder.

Step 3 – Traverse the right subtree in Preorder.

Non-Recursive Algorithm

This algorithm traverses the tree non-recursively in preorder using STACK. Initially push NULL onto stack and then set P=root and the repeat the following steps input p!=NULL

Step 1 – Proceed down the leftmost path rooted at P, processing each node N on the path and pushing each right child, P->right. If any, onto stack. The traversing ends after a node N with no left child L(N) is processed. Therefore the pointer P is updated using the assignment P=P->left and the traversing stops when P->left=NULL.

Step 2 – Pop the element from stack and assign to P. If P≠NULL, then return step 1 else exit

```
void pretrav(nodeptr tree)
```

```

{
    if(tree!=NULL)
    {
        printf("->%d",tree->data);
        pretrav(tree->left);
        pretrav(tree->right);
    }
}

```



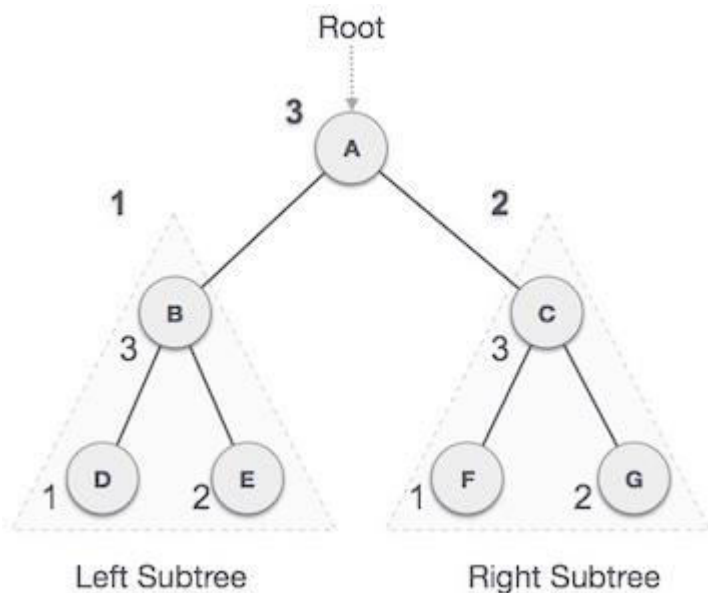
```

}
}

```

Post-order Traversal

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 pre-order traversal, we first visit the left subtree **B**. **B** is also traversed

versal of this

$D \rightarrow E \rightarrow B \rightarrow F \rightarrow G \rightarrow C \rightarrow A$

Algorithm

Until all nodes are traversed –

Step 1 – Traverse the left subtree in Postorder.

Step 2 – Traverse the right subtree in Postorder.

Step 3 – Visit root node.

Non-Recursive Algorithm

This algorithm traverses the tree non-recursively in Inorder using STACK. Initially push NULL onto stack and then set $P = \text{root}$ and the repeat the following steps until NULL is popped from stack

Step 1 – Proceed down the leftmost path rooted at P. At each node N of the path and push N of the path, push N onto stack and if N has a right child $P \rightarrow \text{right}$, push $P \rightarrow \text{right}$ onto

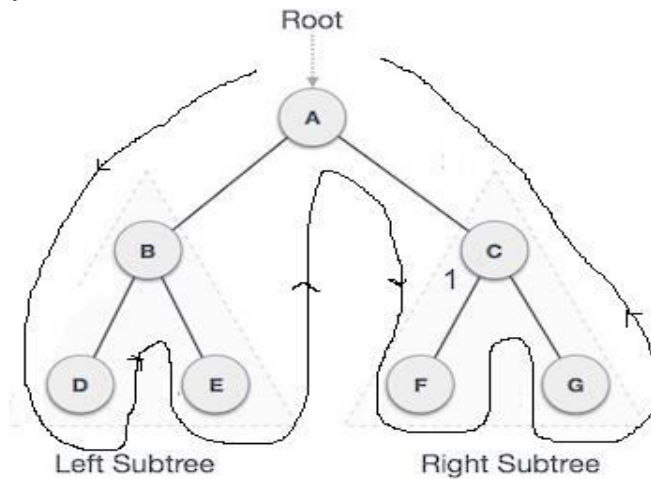
Step 2 – Pop and process nodes on stack. If Null is popped, then exit. Return to step 1

void posttrav(nodeptr tree)

```

{
  if(tree!=NULL)
  {
    posttrav(tree->left);
    posttrav(tree->right);
    printf("->%d",tree->data);
  }
}

```



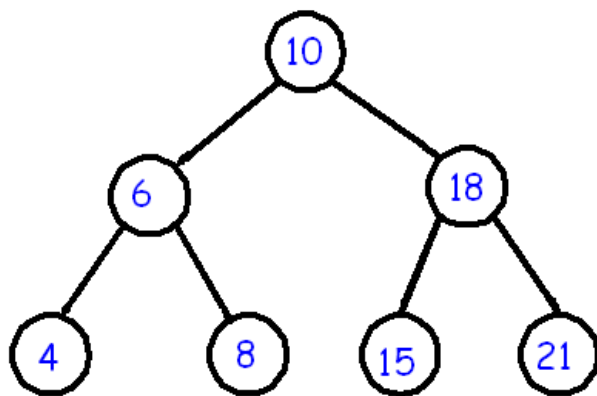
Preorder- A->B->D->E->C->F->G

Inorder- C-

Postorder-D->E->B->F->G->C->A

Binary Search Trees

We consider a particular kind of a binary tree called a Binary Search Tree (BST). The basic idea behind this data structure is to have such a storing repository that provides the efficient way of data sorting, searching and retrieving.



A BST is a binary tree where nodes are ordered in the following way:

- each node contains one key (also known as data)
- the keys in the left subtree are less than the key in its parent node, in short $L < P$;
- the keys in the right subtree are greater than the key in its parent node, in short $P < R$;

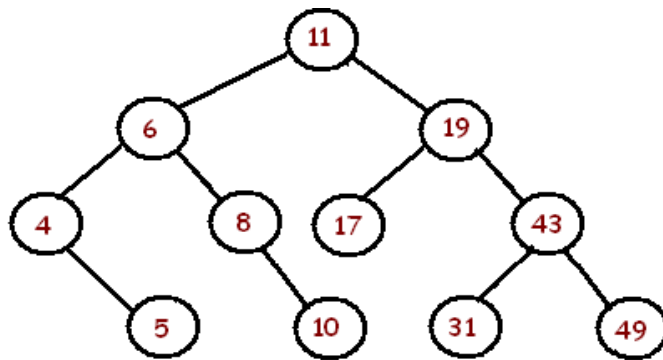
- duplicate keys are not allowed.

In the following tree all nodes in the left subtree of 10 have keys < 10 while all nodes in the right subtree > 10 . Because both the left and right subtrees of a BST are again search trees; the above definition is recursively applied to all internal nodes:

Exercise. Given a sequence of numbers:

11, 6, 8, 19, 4, 10, 5, 17, 43, 49, 31

Draw a binary search tree by inserting the above numbers



Searching

To search a given key in Binary Search Tree, we first compare it with root, if the key is present at root, we return root. If key is greater than root's key, we recur for right subtree of root node. Otherwise we recur for left subtree.

Now, let's see more detailed description of the search algorithm. Like an add operation, and almost every operation on BST, search algorithm utilizes recursion. Starting from the root,

Steps

1. check, whether value in current node and searched value are equal. If so, **value is found**. Otherwise,
2. if searched value is less, than the node's value:
 - if current node has no left child, **searched value doesn't exist in the BST**;
 - otherwise, handle the left child with the same algorithm.
3. if a new value is greater, than the node's value:

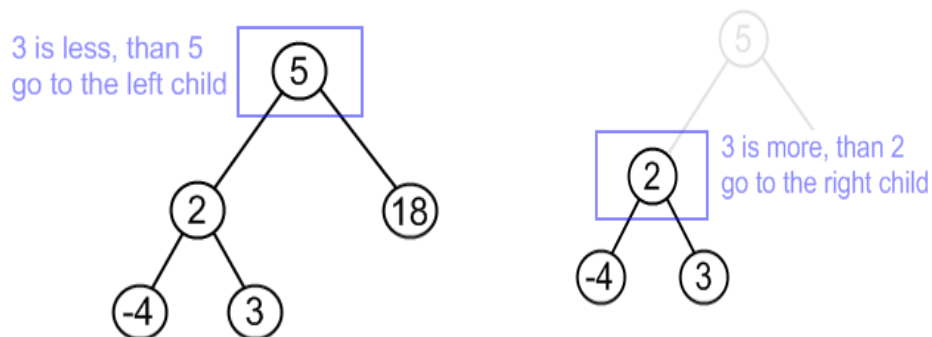
- if current node has no right child, **searched value doesn't exist in the BST**;
- otherwise, handle the right child with the same algorithm.

Algorithm

1. void search(node *tree, int digit)
2. if(tree==NULL) then step 3 otherwise goto step 4
3. printf("The Number does not exists");
4. else if(digit==tree->num) then goto step 5 otherwise go to step 6
5. printf("%d is found",digit);
6. else if(digit<tree->num) then goto step 7 otherwise go to step 8
7. search(tree->left,digit)
8. else search(tree->right,digit)

Example

Search for 3 in the tree, shown above.

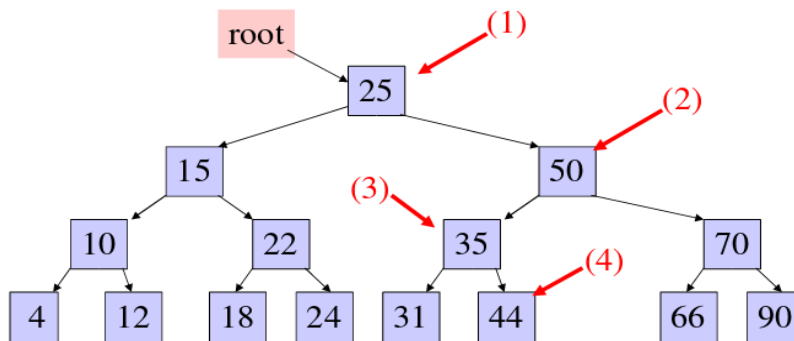




Example: search for 45 in the tree

(key fields are show in node rather than in separate obj ref to by data field):

1. start at the root, 45 is greater than 25, search in right subtree
2. 45 is less than 50, search in 50's left subtree
3. 45 is greater than 35, search in 35's right subtree
4. 45 is greater than 44, but 44 has no right subtree so 45 is not in the BST



Insertion

Steps for insertion

- Check whether root node is present or not(tree available or not). If root is NULL, create root node.
- If the element to be inserted is less than the element present in the root node, traverse the left sub-tree recursively until we reach T->left/T->right is NULL and place the new node at T->left(key in new node < key in T)/T->right (key in new node > key in T).
- If the element to be inserted is greater than the element present in root node, traverse the right sub-tree recursively until we reach T->left/T->right is NULL and place the new node at T->left/T->right.

Algorithm

1. void insert(node *tree,int digit)
2. if(tree==NULL) then step from 3 to 6 then goto step 6
3. tree=(node*)malloc(sizeof(node));
4. tree->left=tree->right=NULL;
5. tree->num=digit;
6. if(digit<tree->num) then goto step 7 otherwise step 8
7. tree->left=insert(tree->left,digit)
8. if(digit>tree->num) then goto step 9 otherwise goto step 10
9. tree->right=insert(tree->right,digit)
10. if(digit==tree->num)then goto step 11
11. print "Duplicate Nodes"
12. return(tree)
13. End

A new key is always inserted at leaf. We start searching a key from root till we hit a leaf node. Once a leaf node is found, the new node is added as a child of the leaf node.

Example:

Insert 20 into the Binary Search Tree.

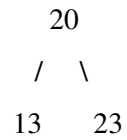
Tree is not available. So, create root node and place 10 into it.

20

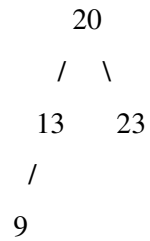
Insert 23 into the given Binary Search Tree. $23 > 20$ (data in root). So, 23 needs to be inserted in the right sub-tree of 20.

20
 \
 23

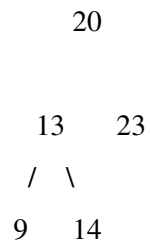
Insert 13 into the given Binary Search Tree. $13 < 20$ (data in root). So, 13 needs to be inserted in left sub-tree of 20.



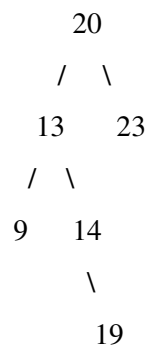
Insert 9 into the given Binary Search Tree.



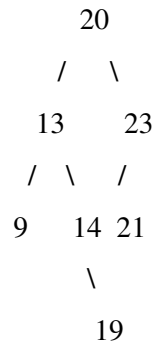
Inserting 14.



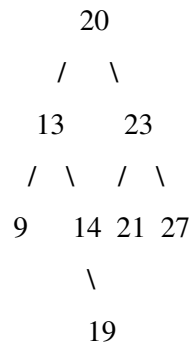
Inserting 19.



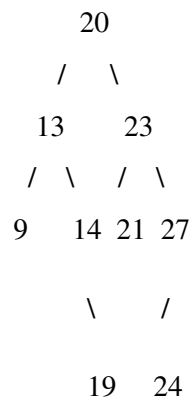
Inserting 21.



Inserting 27.

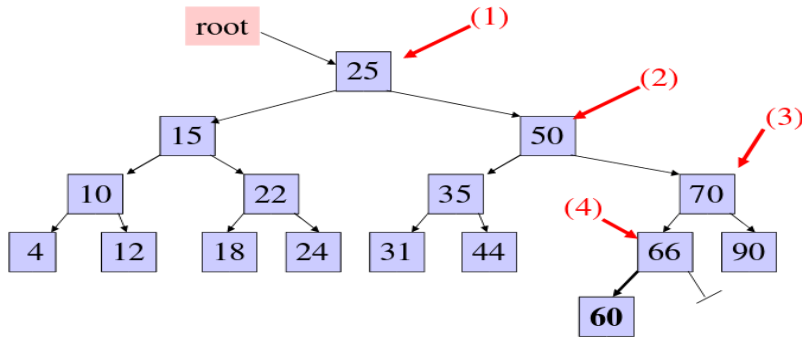


Inserting 24.



Example: insert 60 in the tree:

1. start at the root, 60 is greater than 25, search in right subtree
2. 60 is greater than 50, search in 50's right subtree
3. 60 is less than 70, search in 70's left subtree
4. 60 is less than 66, add 60 as 66's left child



Deletion

Remove operation on binary search tree is more complicated, than add and search. Basically, it can be divided into two stages:

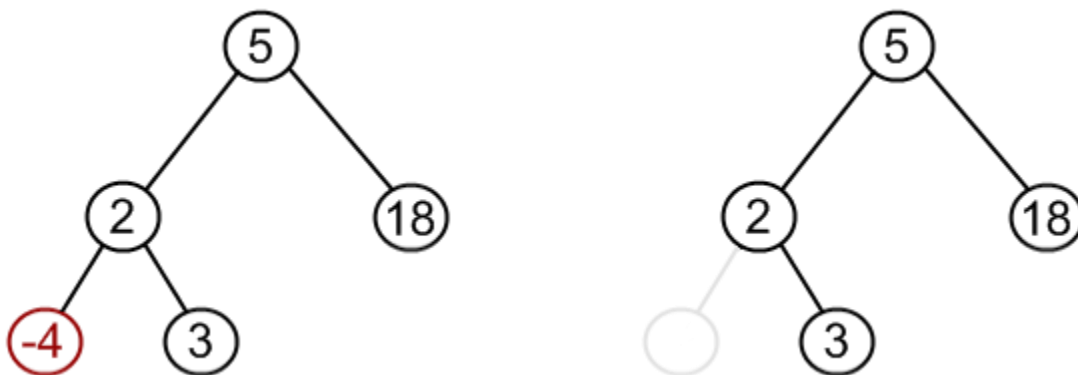
- search for a node to remove;

Now, let's see more detailed description of a remove algorithm. First stage is identical to algorithm for lookup, except we should track the parent of the current node. Second part is more tricky. There are three cases, which are described below.

1. Node to be removed has no children.

This case is quite simple. Algorithm sets corresponding link of the parent to NULL and disposes the node.

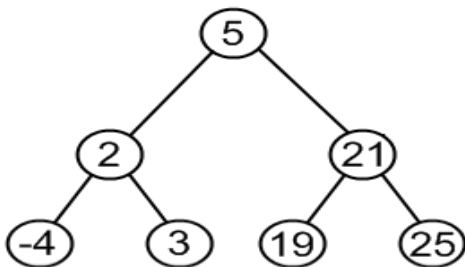
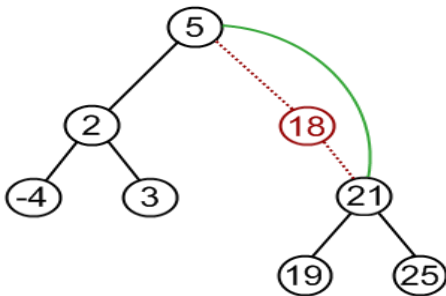
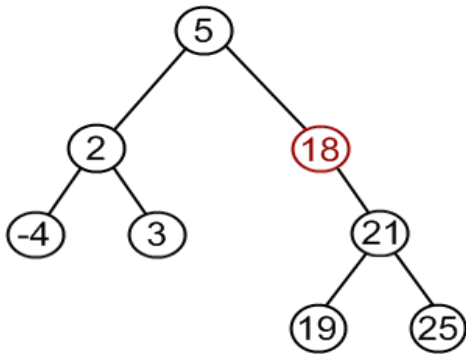
Example. Remove -4 from a BST.



2. Node to be removed has one child.

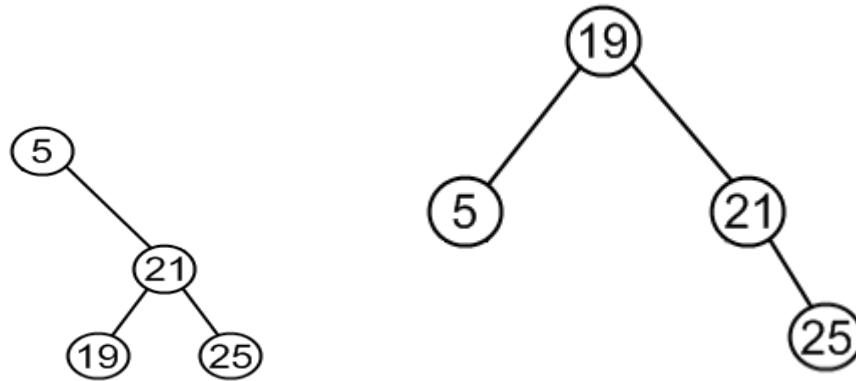
It this case, node is cut from the tree and algorithm links single child (with it's subtree) directly to the parent of the removed node.

Example. Remove 18 from a BST.



3. Node to be removed has two children.

This is the most complex case. To solve it, let us see one useful BST property first. We are going to use the idea, that the same set of values may be represented as different BST



contains the same values {5, 19, 21, 25}. To transform first tree into second one, we can do following:

- choose minimum element from the right subtree (19 in the example);
- replace 5 by 19;
- hang 5 as a left child.

The same approach can be utilized to remove a node, which has two children:

- find a minimum value in the right subtree;
- replace value of the node to be removed with found minimum. Now, right subtree
- apply remove to the right subtree to remove a duplicate.

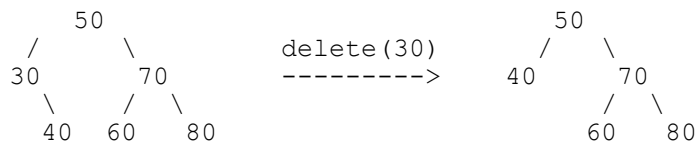
Notice, that the node with minimum value has no left child and, therefore, it's removal may result in first or second cases only.

Example. Remove 12 from a BST.

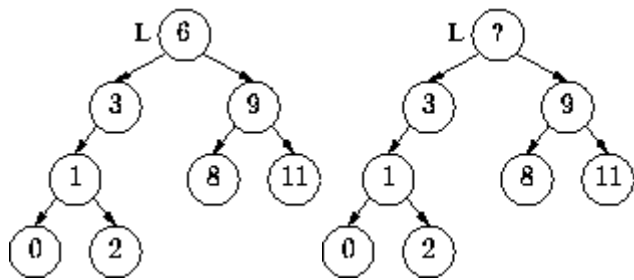
1) Node to be deleted is leaf: Simply remove from the tree.



2) Node to be deleted has only one child: Copy the child to the node and delete the child



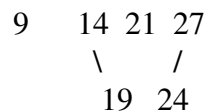
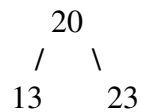
3) Node to be deleted has two children



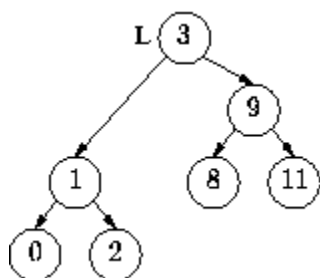
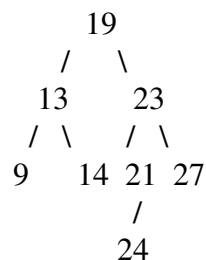
Now, what value can we move into the vacated node and have a binary search tree? Well, here's how to figure it out. If we choose value X, then:

1. everything in the left subtree must be smaller than X.
2. everything in the right subtree must be bigger than X.

Let's suppose we're going to get X from the left subtree. (2) is guaranteed because everything in the left subtree is smaller than everything in the right subtree. What about (1)? If X is coming from the left subtree, (1) says that there is a unique choice for X - we must choose X to be the largest value in the left subtree. In our example, 3 is the largest value in the left subtree. So if we put 3 in the vacated node and delete it from its current position we will have a BST with 6 deleted.



Delete 20 from the above binary tree. Find the smallest in the left subtree of 20. So, replace 19 with 20.



Algorithm

```
void deletenode(node *tree,int digit)
{
    struct node *r,*q;
    if(tree==NULL)
    {
        print "Tree is empty"
        exit(0);
    }
    if(digit<tree->num)
        deletenode(tree->left,digit);
    else if(digit>tree->num)
        deletenode(tree->right,digit);
    q=tree;
    if((q->right==NULL)&&(q->left==NULL))
        q=NULL;
    else if(q->right==NULL)
        tree=q->left;
    else if (q->left==NULL)
        tree=q->right;
    else
        delnum(q->left,digit);
    free(q)
}

delnum(int struct node *r,int digit)
{
    struct node * q;
    if(r->right!=NULL)
        delnum(r->right,digit);
```

Else

```
q->num=r->num;
```

```
q=r;
```

```
r=r->left;
```

```
}
```

Construct binary Tree from inorder and preorder traversal

The following procedure demonstrates on how to rebuild tree from given inorder and preorder traversals of a binary tree:

- Preorder traversal visits Node, left subtree, right subtree recursively
- Inorder traversal visits left subtree, node, right subtree recursively
- Since we know that the first node in Preorder is its root, we can easily locate the root node in the inorder traversal and hence we can obtain left subtree and right subtree from the inorder traversal recursively

Example 1: in-order: 4 2 5 (1) 6 7 3 8

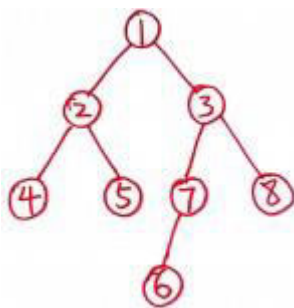
pre-order: (1) 2 4 5 3 7 6 8

Construct binary tree

Soln: From the pre-order array, we know that first element is the root. We can find the root in in-order array. Then we can identify the left and right sub-trees of the root from in-order array.

Using the length of left sub-tree, we can identify left and right sub-trees in pre-order array. Recursively, we can build up the tree.

For this example, the constructed tree is:



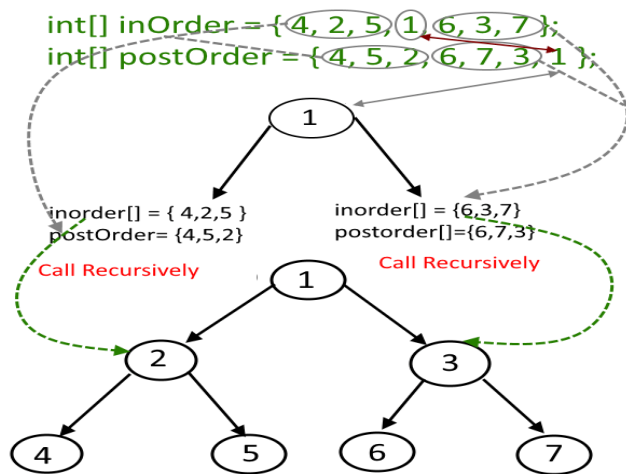
Example 2: inOrder = { 4, 2, 5, 1, 6, 3, 7 };

postOrder = { 4, 5, 2, 6, 7, 3, 1 };.Construct binary tree

Soln

- Last element in the *postorder* [] will be the *root* of the tree, here it is 1.

- Now the search element 1 in *inorder[]*, say you find it at position *i*, once you find it, make note of elements which are left to *i* (this will construct the leftsubtree) and elements which are right to *i* (this will construct the rightSubtree).
- Suppose in previous step, there are X number of elements which are left of '*i*' (which will construct the leftsubtree), take first X elements from the postorder[] traversal, this will be the post order traversal for elements which are left to *i*. similarly if there are Y number of elements which are right of '*i*' (which will construct the rightsubtree), take next Y elements, after X elements from the postorder[] traversal, this will be the post order traversal for elements which are right to *i*
- From previous two steps construct the left and right subtree and link it to root.left and root.right respectively.
- See the picture for better explanation.

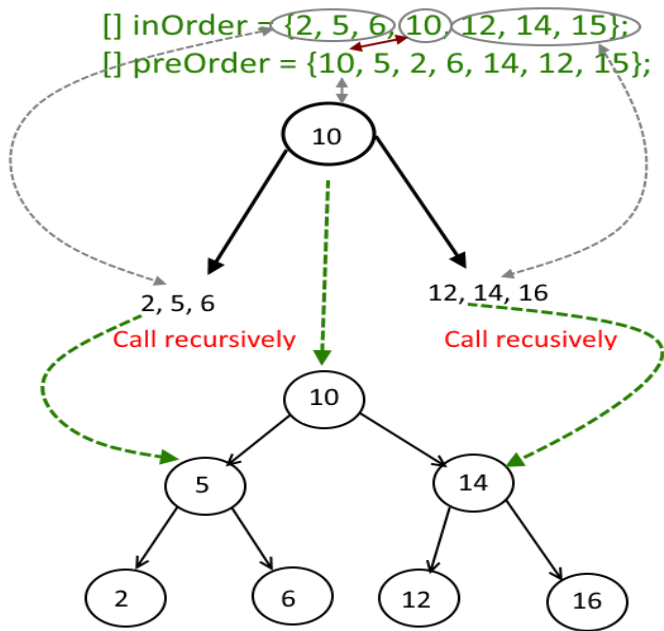


Example 3: inOrder = { 2, 5, 6, 10, 12, 14, 15 };

preOrder = { 10, 5, 2, 6, 14, 12, 15 }; Construct binary tree

Soln

- First element in *preorder[]* will be the *root* of the tree, here its 10.
- Now the search element 10 in *inorder[]*, say you find it at position *i*, once you find it, make note of elements which are left to *i* (this will construct the leftsubtree) and elements which are right to *i* (this will construct the rightSubtree).
- See this step above and recursively construct left subtree and link it root.left and recursively construct right subtree and link it root.right.
- See the picture and code.



Example 4:Preorder Traversal: 1 2 4 8 9 10 11 5 3 6 7

Inorder Traversal: 8 4 10 9 11 2 5 1 6 3 7

Construct binary tree

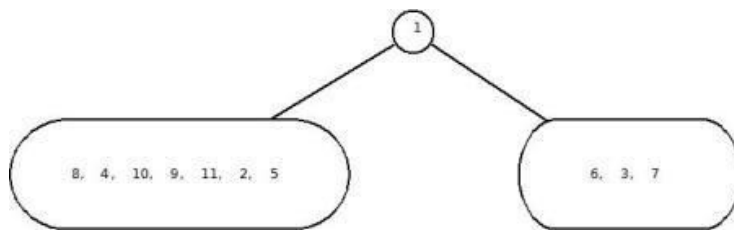
Soln

Iteration 1:

Root – {1}

Left Subtree – {8,4,10,9,11,2,5}

Right Subtree – {6,3,7}



Iteration 2:

Root – {2}

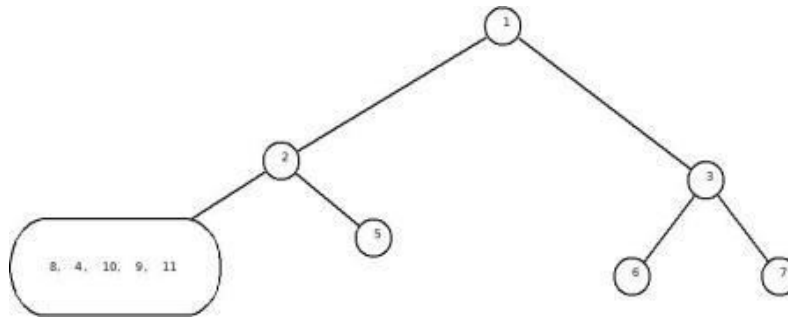
Left Subtree – {8,4,10,9,11}

Right Subtree – {5}

Root – {3}

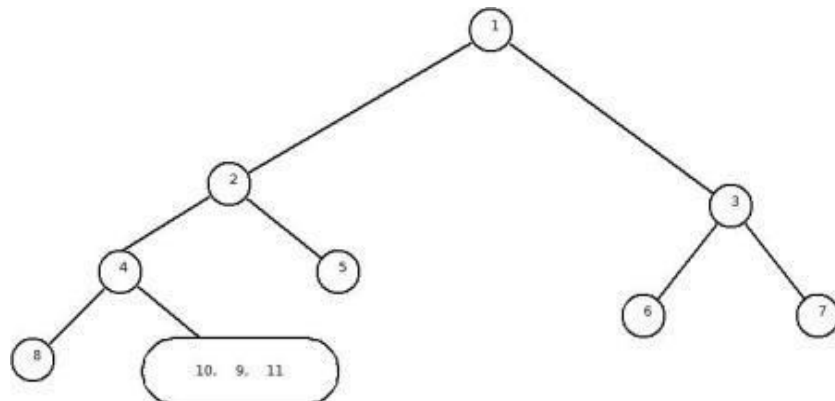
Left Subtree – {6}

Right Subtree – {7}



Iteration 3:

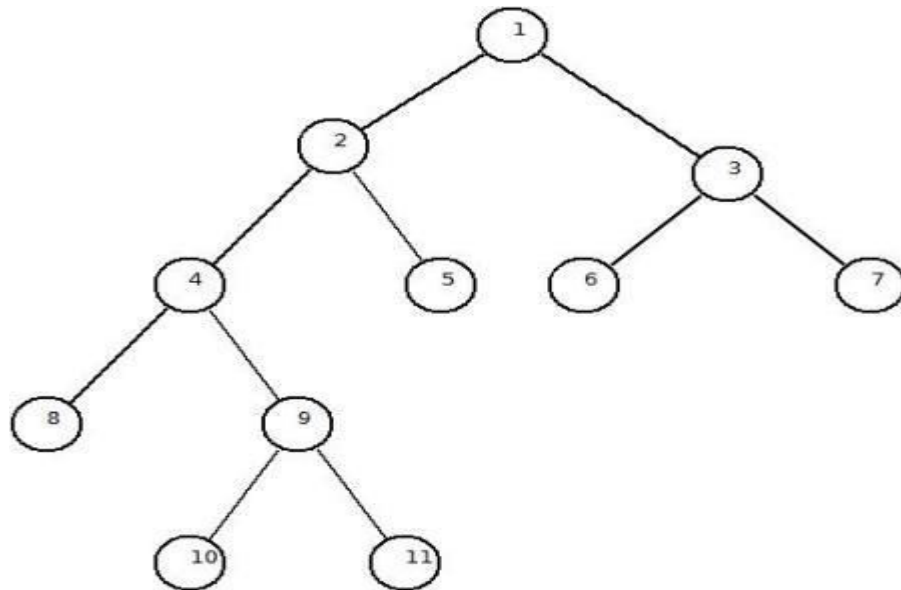
Root – {2}		Root – {3}
Left Subtree – {8,4,10,9,11}		Left Subtree – {6}
Right Subtree – {5}		Right Subtree – {7}
Root – {4}	Done	Done
Left Subtree – {8}		
Right Subtree – {10,9,11}		



Iteration 4:

Root – {2}		Root – {3}
Left Subtree – {8,4,10,9,11}		Left Subtree – {6}
Right Subtree – {5}		Right Subtree – {7}
Root – {4}	Done	Done
Left Subtree – {8}		
Right Subtree – {10,9,11}		

Done	R – {9} Left ST – {10} Right ST- {11}	Done	Done
------	---	------	------



The following are the two versions of programming solutions even though both are based on above mentioned algorithm:

- Creating left preorder, left inorder, right preorder, right inorder lists at every iteration to construct tree
- Passing index of preorder and inorder traversals and using the same input list to construct tree

Advantages of trees

Trees are so useful and frequently used, because they have some very serious advantages:

- Trees reflect structural relationships in the data
- Trees are used to represent hierarchies
- Trees provide an efficient insertion and searching
- Trees are very flexible data, allowing to move subtrees around with minimum effort

Application of tree

1. Manipulate hierarchical data.

2. Make information easy to search (see tree traversal).
3. Manipulate sorted lists of data.
4. As a workflow for compositing digital images for visual effects.
5. Router algorithms
6. Windows file system

MODULE 5

SORTING TECHNIQUES

Sorting method can be classified into two.

- 1) Internal Sorting
- 2) External Sorting

In internal sorting the data to be sorted is placed in main memory. In external sorting the data is placed in external memory such as hard disk, floppy disk etc. The internal sorting can be classified into

- 1) n^2 sorting
- 2) $n \log n$ sorting

n^2 sorting - it can be classified into

- 1) Bubble sort
- 2) Selection sort
- 3) Insertion sort

$n \log n$ sorting - It can be classified into

- 1) Merge sort
- 2) Quick sort
- 3) Heap sort

Bubble sort

Bubblesort(a[],n)

Input- an array a[size], n is the no. of element currently present in array

Output- Sorted array

DS- Array

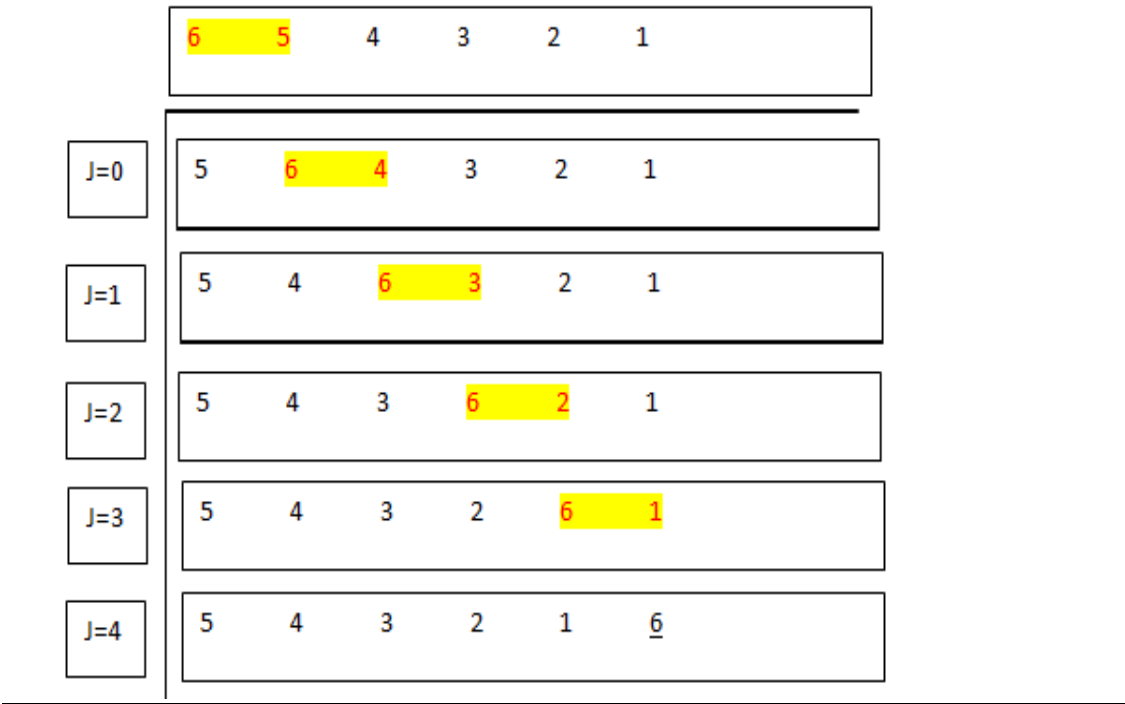
Algorithm

1. Start
2. $i=0$
3. While $i < n-1$
 1. $j=0$
 2. While $j < n-1-i$
 1. If $a[j] > a[j+1]$
 1. $temp = a[j]$
 2. $a[j] = a[j+1]$
 3. $a[j+1] = temp$
 2. end if
 3. $j = j+1$
3. end while
4. $i = i+1$

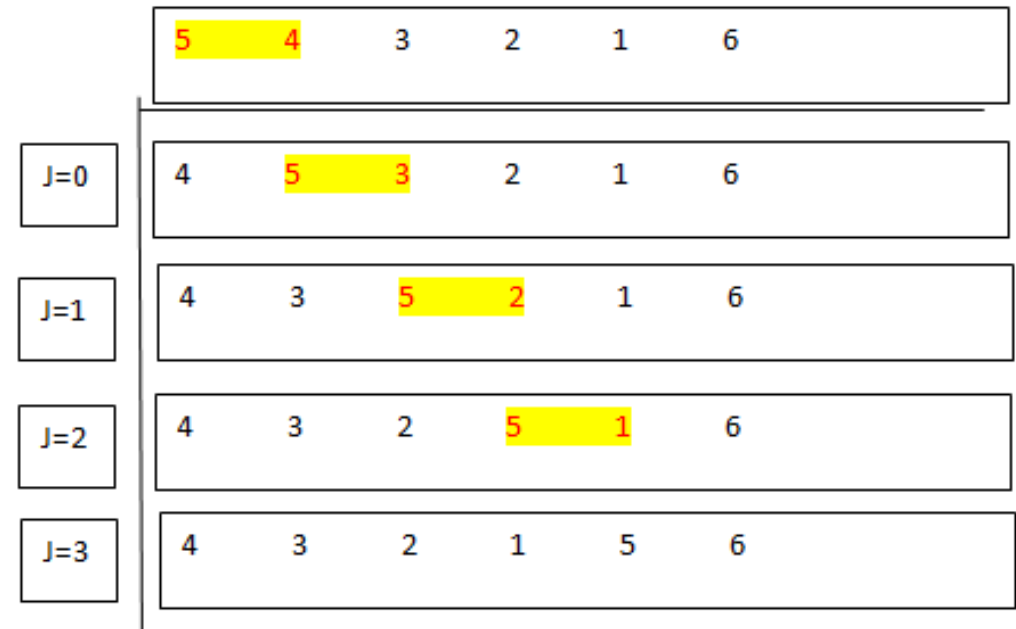
4. end while

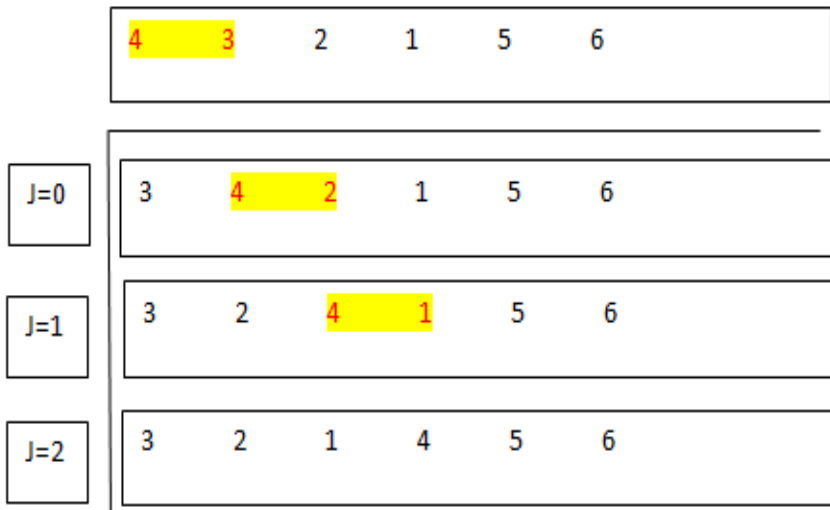
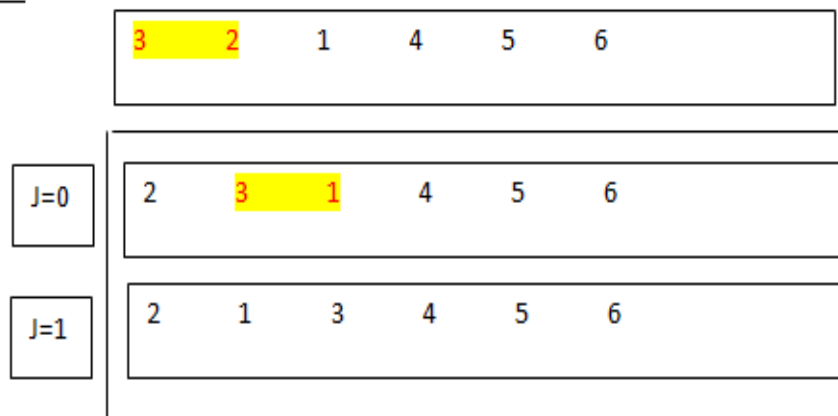
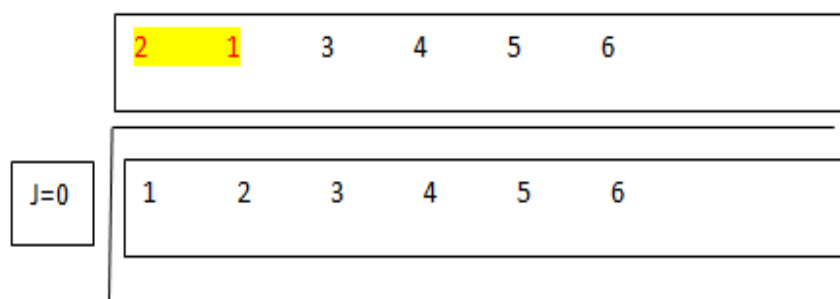
Here first element is compared with second element. If first one is greater than second element then swap each other. Then second element is compared with third element . If second element is greater than third element then perform swapping. This process is continued until the comparison of (n-1)th element with nth element. These process continues (n-1) times. Consider the example-

i=0



i=1



i=2**i=3****i=4**

Thus i have values from 0 to n-1 and j have values from 0 to n-1-i.

Analysis

Here during the first iteration n-1 comparisons are required. During the second iteration n-2 comparisons are required etc..during last iteration, 1 comparison is required. Therefore total comparisons

$$= (n-1) + (n-2) + (n-3) + \dots + 1$$

$$= \frac{n(n-1)}{2}$$

$$=O(n^2)$$

SELECTION SORT

Selectionsort(a[],n)

Input: An unsorted array a[], n is the no.of elements

Output: a sorted array

DS: Array

Algorithm

- 1: Start
2. i=0
3. while i<n-1 do
 1. j=i+1
 2. small=i
 3. while j<n do
 - 1.if a[small]>a[j]
 1. small=j
 2. end if
 3. j=j+1
 4. end while
 5. if i!=small
 1. temp=a[i]
 2. a[i]=a[small]
 3. a[small]=temp
 6. end if
 7. i=i+1
4. end while
5. stop

Here the first element in the array is selected as the smallest element. Then check for any element smaller than this from the second position to the last. If found any, then they are interchanged. Similarly next we check for the smallest element from third position to last. And the process continues upto (n-1)th position.

Analysis

Here first iteration when i=0 takes n-1 comparisons, during second iteration takes n-2 comparison and so on.

Total no. of comparisons=(n-1)+(n-2)+(n-3)+.....+2+1

$$= \frac{n(n-1)}{2}$$

$$= O(n^2)$$

Example

	0	1	2	3	4	5	6
J=0	25	6	1	5	100	3	7
J=1	1	6	25	5	100	3	7
J=2	1	3	25	5	100	6	7
J=3	1	3	5	25	100	6	7
J=4	1	3	5	6	100	25	7
J=5	1	3	5	6	7	25	100
	1	3	5	6	7	25	100

INSERTION SORT

insertionsort(a[],n)

Input: An unsorted array a[], n is the no.of elements

Output: a sorted array

DS: Array

Algorithm

- 1. Start
- 2. i=1
- 3. While i<n
 - 1. j=i
 - 2. While a[j]<a[j-1] and j>0
 - 1. t=a[j]
 - 2. a[j]=a[j-1]
 - 3. a[j-1]=t
 - 4. j=j-1
 - 3. end while
 - 4. i=i+1
- 4.end while
- 5. stop

Here first iteration takes 1 comparison, 2nd iteration takes 2 comparison, and last iteration takes (n-1) comparison

Total comparison=1+2+...+n-1

$$\begin{aligned} &= \frac{n(n-1)}{2} \\ &= O(n^2) \end{aligned}$$

Example

	0	1	2	3	4	5	6
J=0	25	6	1	5	100	3	7
J=1	6	25	1	5	100	3	7
J=2	1	6	25	5	100	3	7
J=3	1	5	6	25	100	3	7
J=4	1	5	6	25	100	3	7
J=5	1	3	5	6	25	100	7
	1	3	5	6	7	25	100

MERGE SORT

mergesort(start,end)

Input: An unsorted array a[], n is the no.of elements, start indicates lower bound of the array, end indicates the upper bound of the array

Output: a sorted array

DS: Array

Algorithm

- 1.start
- 2. if(start!= end)
 - 1. mid=(start+end)/2
 - 2. mergesort(start,mid)
 - 3.mergesort(mid+1,end)
 - 4. merge(start,mid,end)
- 3. end if
- 4.stop

Algorithm for merge

Merge(start,mid,end)

```

1. i=start
2. j=mid+1
3. k=start
4. while i<=mid and j<= end do
    1. if a[i]<=a[j]
        1. temp[k]=a[i]
        2. i=i+1
        3. k=k+1
    2. Else
        1. Temp[k] =a[j]
        2. J=j+1
        3. k=k+1
    3. end if
5.end while
6. while i<=mid do
    1. temp[k]=a[i]
    2. i=i+1
    3. k=k+1
7. end while
8. While j<=end
    1. Temp[k]=a[j]
    2. j=j+1
    3. k=k+1
9. end while
10. k=start
11. while k<=end
    1. a[k]=temp[k]
    2. k=k+1
12. end while
13.stop

```

Merge sort follows the strategy divide and conquer method. Here the given base array is divided into two sub lists. These 2 sub lists is again divided into 4 sub lists. The process is continued until subsists contain single element. Then repeatedly merge these two sub lists to a single sub list. So that a sorted array is created from sorted sub lists. The process continues until a sub list contains all the elements that are sorted.

Analysis of merge sort

Suppose the merge sort follows the recurrence relation.

$$T(n)=2T(n/2)+n$$

Comparing with standard recurrence equation $T(n)=aT(n/b)+f(n)$

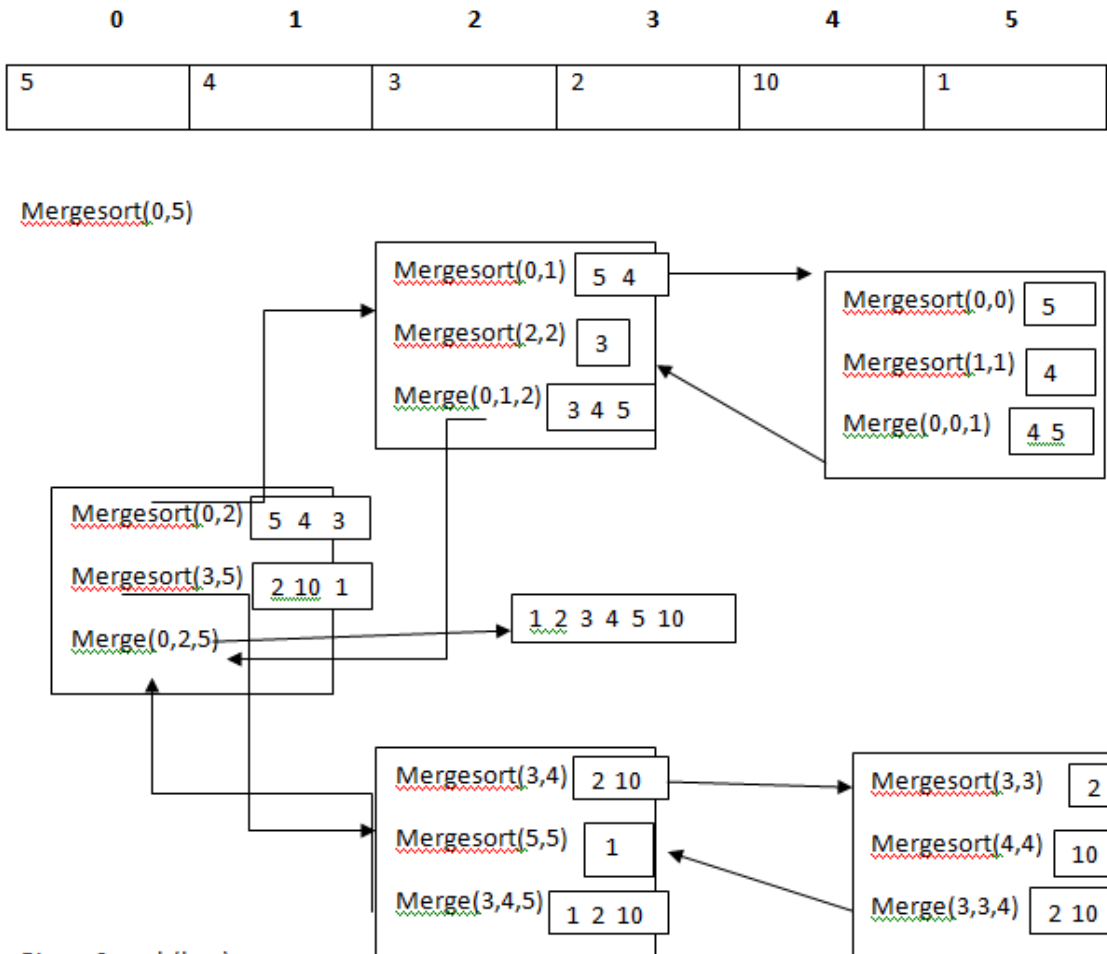
MEA Engineering College, Perinthalmanna

$a=2, b=2, f(n)=n$

then $n^{\log_b a} = n$

therefore time complexity is $\Theta(n \log n)$

Consider the following example



QUICK SORT

Quicksort(lb,ub)

Input: An unsorted array a[], n is the no.of elements, lb indicates lower bound of the array, ub indicates the upper bound of the array

Output: a sorted array

DS: Array

Algorithm

- 1.start
- 2.if lb<ub
 - 1. loc=partition(lb,ub)
 - 2.quicksort(lb,loc-1)
 - 3.quicksort(loc+1,ub)
- 3. end if
- 4. stop

Algorithm for partition

Partition(lb,ub)

1. pivot=a[lb]
2. up=ub
3. down=lb
4. while down< up
 1. while pivot>=a[down] and down<=up
 - 1.down=down+1
 2. end while
 3. while a[up]>pivot
 - 1.up=up-1
 4. end while
 5. if down< = up
 1. swap(a[down], a[up])
 6. end if
5. end while
6. swap(a[lb],a[up])
7. return up
8. stop

Quick sort algorithm basically takes the following steps

1. Choose a pivot element(the pivot element may be in any position).

Normally first element is chosen as pivot

2. Perform partition function in such a way that all the elements which are lesser than pivot goes to the left part of the array and all the elements greater than pivot go to the right part of the array. The partition function also places pivot in exact position.

3. Recursively perform quick sort algorithm in these two sub arrays

Analysis

The recurrence relation of quick sort in worst case is

$$T(n)=T(n/2)+T(n/2)+(n^2)$$

$$T(n)=2T(n/2)+(n^2)$$

Comparing with the standard recurrence equation

$$T(n)=aT(n/b)+f(n)$$

Substitute the values of a and b in $n^{\log_b a}$ and compare it with f(n).

In this case a=2, b=2 then $n^{\log_b a}=n$

Whereas f(n)= n^2 . Therefore time complexity of quick sort is $\Theta(n^2)$