Code: 80503	DATA STRUCTURES (Common for EEE, ECE, CSE and IT)	L	T	P
Credits: 3		3	-	-

Pre requisite :Computer Programming

Course Objectives:

This course will deliver the knowledge in introducing the concepts of various data structures such as linked lists, stacks, queues, trees and graphs along with the applications.

MODULE-I: Performance Analysis and Introduction to data structures[10 Periods]

Performance Analysis: Algorithm definition and characteristics, time and space complexity, Asymptotic Notations – Big O, Omega and Theta notations.

Introduction to data structures: Types of data structures: Linear and Non-linear data structures. Recursion definition- Linear and Binary recursion, Design methodology and implementation of recursive algorithms, Recursive algorithms for Towers of Hanoi.

MODULE-II: Linked Lists [09 Periods]

Single Linked Lists: Definition, Operations-Insertion, Deletion and Searching, Concatenating single linked lists, Circular linked lists, Operations-Insertion, Deletion. **Double Linked Lists:** Definition, Operations- Insertion, Deletion. Applications of Linked list. Sparse matrices - Array and linked representations.

MODULE-III:Stacks and Queues [10 Periods]

A: Stacks: Basic stack operations, Representation of a stack using arrays and linked lists, Stack Applications - Reversing list, factorial calculation, postfix expression evaluation, infix-to-postfix conversion.

B:Queues: Basic queue operations, Representation of a queue using array and Linked list, Classification and implementation – Circular, Enqueue and Dequeue, Applications of Queues.

MODULE-IV: Trees and Graphs[10 Periods]

Trees: Basic concepts of Trees, Binary Tree: Properties, Representation of binary tree using array and linked lists, operations on a binary tree, binary tree traversals, creation of binary tree from in, pre and post-order traversals, Tree traversals using stack, Threaded binary tree.

Graphs: Basic concepts of Graphs, Representation of Graphs using Linked list and Adjacency matrix, Graph algorithms, Graph traversals- (BFS & DFS).

MODULE-V: Search Trees [09 Periods]

Binary Search Trees and AVL Trees: Binary Search Tree, Definition, Operations - Searching, Insertion and Deletion, AVL Trees (Elementary treatment-only Definitions and Examples).B-Trees and Red-Black Tree: B-Trees, Red-Black and Splay Trees (Elementary treatment-only Definitions and Examples), Comparison of Search Trees.

TEXT BOOKS:

- Jean Paul Tremblay, Paul G Sorenson, “An Introduction to Data Structures with Applications”, Tata McGraw Hills, 2nd Edition, 1984.
- Richard F. Gilberg, Behrouz A. Forouzan, “Data Structures: A Pseudo code approach with C ", Thomson (India), 2nd Edition, 2004.

REFERENCES:

- Horowitz, Ellis, Sahni, Sartaj, Anderson-Freed, Susan, “Fundamentals of Data Structure in C”, University Press (India), 2nd Edition, 2008.
- A. K. Sharma, “Data structures using C”, Pearson, 2nd Edition, June, 2013.
- R. Thareja, “Data Structures using C”, Oxford University Press, 2nd Edition, 2014.

E-RESOURCES:

- http://gvpcse.azurewebsites.net/pdf/data.pdf
- http://www.sncwgs.ac.in/wp-content/uploads/2015/11/Fundamental-Data- HYPERLINK "http://www.sncwgs.ac.in/wp-content/uploads/2015/11/Fundamental-Data-Structures.pdf" HYPERLINK "http://www.sncwgs.ac.in/wp-content/uploads/2015/11/Fundamental-Data-Structures.pdf"Structures.pdf
- http://www.learnerstv.com/Free-Computer-Science-Video-lectures-ltv247-Page1.htm
- http://ndl.iitkgp.ac.in/document/yVCWqd6u7wgye1qwH9xY7- HYPERLINK "http://ndl.iitkgp.ac.in/document/yVCWqd6u7wgye1qwH9xY7-3lcmoMApVUMmjlExpIb1zste4YXX1pSpX8a2mLgDzZ-E41CJ6PVmY4S0MqVbxsFQ" HYPERLINK "http://ndl.iitkgp.ac.in/document/yVCWqd6u7wgye1qwH9xY7-

3lcmoMApVUMmjlExpIb1zste4YXX1pSpX8a2mLgDzZ-E41CJ6PVmY4S0MqVbxsFQ"3lcmoMApVUMmjlExpIb1zste4YXX1pSpX8a2mLgDzZ-HYPERLINK "http://ndl.iitkgp.ac.in/document/yVCWqd6u7wgye1qwH9xY7-3lcmoMApVUMmjlExpIb1zste4YXX1pSpX8a2mLgDzZ-E41CJ6PVmY4S0MqVbxsFQ"E41CJ6PVmY4S0MqVbxsFQ

- http://nptel.ac.in/courses/106102064/1

Course Outcomes:

At the end of the course, students will be able to

- Identify the appropriate data structures and analyze the performance of algorithms.
- Understand and implement single, double, and circular linked-lists.
- Implement Stacks and Queues using array and linked-list representations.
- Develop programs by using non linear data structures such as trees and graphs.
- Design and Implement applications of advanced data structures.

Lecture Notes

MODULE-I: Performance Analysis and Introduction to data structures [10 Periods] Performance Analysis**: Algorithm definition and characteristics, time and space complexity, Asymptotic Notations – Big O, Omega and Theta notations.**

Introduction to data structures: Types of data structures: Linear and Non-linear data structures. Recursion definition- Linear and Binary recursion, Design methodology and implementation of recursive algorithms, Recursive algorithms for Towers of Hanoi

Performance analysis

Algorithm:

An algorithm may be defined as a finite sequence of instructions each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time. The algorithm word originated from the Arabic word “Algorism” which is linked to the name of the Arabic mathematician AI Khwarizmi. He is considered to be the first algorithm designer for adding numbers.

Structure and Properties of Algorithm:

An algorithm has the following structure

1. Input Step
2. Assignment Step
3. Decision Step
4. Repetitive Step
5. Output Step

Characteristics of algorithm:

Finiteness: An algorithm must terminate after a finite number of steps.

Definiteness: The steps of the algorithm must be precisely defined or unambiguously specified.

Generality: An algorithm must be generic enough to solve all problems of a particular class.

Effectiveness: the operations of the algorithm must be basic enough to be put down on pencil and paper. They should not be too complex to warrant writing another algorithm for the operation.

Input-Output: The algorithm must have certain initial and precise inputs, and outputs that may be generated both at its intermediate and final steps.

An algorithm does not enforce a language or mode for its expression but only demands adherence to its properties.

- **To save time (Time Complexity):** A program that runs faster is a better program.
- **To save space (Space Complexity):** A program that saves space over a competing program is considerable desirable.

Efficiency of Algorithms:

The performances of algorithms can be measured on the scales of **time** and **space**. The performance of a program is the amount of computer memory and time needed to run a program. We use two approaches to determine the performance of a program. One is analytical and the other is experimental. In performance analysis we use analytical methods, while in performance measurement we conduct experiments.

Time Complexity:

The time complexity of an algorithm or a program is a function of the running time of the algorithm or a program. In other words, it is the amount of computer time it needs to run to completion.

Space Complexity:

The space complexity of an algorithm or program is a function of the space needed by the algorithm or program to run to completion.

The time complexity of an algorithm can be computed either by an **empirical** or **theoretical** approach. The **empirical** or **posteriori testing** approach calls for implementing the complete algorithms and executing them on a computer for various instances of the problem. The time taken by the execution of the programs for various instances of the problem are noted and compared. The algorithm whose implementation yields the least time is considered as the best among the candidate algorithmic solutions.

Analyzing Algorithms:

Suppose M is an algorithm, and suppose n is the size of the input data. Clearly the complexity f(n) of M increases as n increases. It is usually the rate of increase of f(n) with some standard functions. The most common computing times are O(1), $O(\log_2 n)$, O(n), $O(n \log_2 n)$, $O(n^2)$, $O(n^3)$, $O(2^n)$

Example:

Program Segment A   **Program Segment B**   **Program Segment C**

```
-------------------------              -------------------------        -------------
-------------

x =x + 2;                  for k =1 to n do            for j =1 to n do

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

x =x + 2;                  for x = 1 to n do           end;

x =x + 2;          -------------------------        end

end;

-------------------------
```

Total Frequency Count of Program Segment A	
Program Statements	**Frequency Count**
-------------------------	
x =x + 2;	1

Total Frequency Count	1
Total Frequency Count of Program Segment B	
Program Statements	**Frequency Count**
-------------------------	
for k =1 to n do	(n+1)
x =x + 2;	n
end;	n

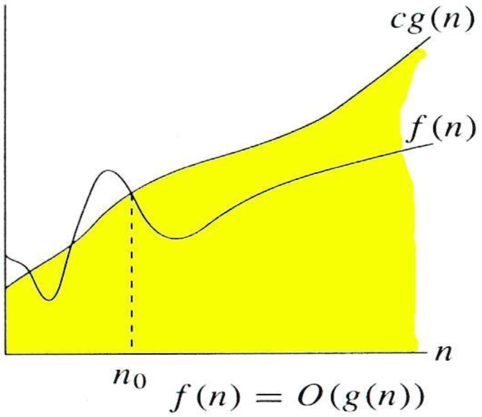
Total Frequency Count	3n+1

The total Frequency counts of the program segments A, B and C given by 1, (3n+1) and ($3n^2$+3n+1) respectively are expressed as O(1), O(n) and O(n^2). These are referred to as the time complexities of the program segments since they are indicative of the running times of the program segments. In a similar manner space complexities of a program can also be expressed in terms of mathematical notations, which is nothing but the amount of memory they require for their execution.

Asymptotic Notations:

It is often used to describe how the size of the input data affects an algorithm's usage of computational resources. Running time of an algorithm is described as a function of input size n for large n.

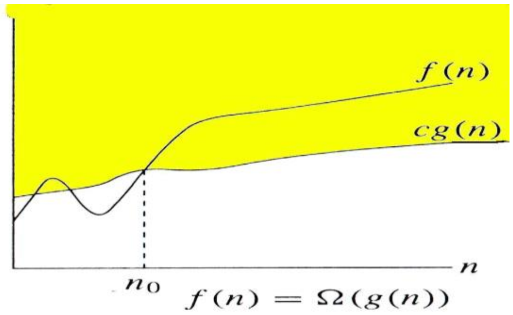
Big oh(O): Definition: f(n) = O(g(n)) (read as f of n is big oh of g of n) if there exist a positive integer n_0 and a positive number c such that $|f(n)| \leq c|g(n)|$ for all $n \geq n_0$. Here g(n) is the upper bound of the function f(n).



$f(n) = O(g(n))$

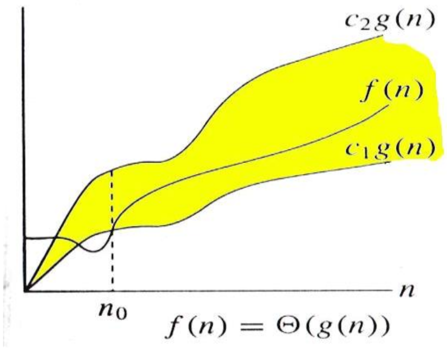
	f(n)			**g(n)**	
	3	2	3		3
16n	+ 45n	+	n		f(n) = O(n)
12n					
34n – 40			n		f(n) = O(n)
50			1		f(n) = O(1)

Omega(Ω): Definition: f(n) = Ω(g(n)) (read as f of n is omega of g of n), if there exists a positive integer n_0 and a positive number c such that $|f(n)| \geq c\ |g(n)|$ for all $n \geq n_0$. Here g(n) is the lower bound of the function f(n).



f(n)		**g(n)**	
3	2	3	3
16n +8n + 2		n	f(n) = Ω(n)
24n + 9		n	f(n) = Ω(n)

Theta(Θ): Definition: f(n) = Θ(g(n)) (read as f of n is theta of g of n), if there exists a positive integer n_0 and two positive constants c_1 and c_2 such that c_1 |g(n)| ≤ |f(n)| ≤ c_2 |g(n)| for all n ≥ n_0. The function g(n) is both an upper bound and a lower bound for the function f(n) for all values of n, n ≥ n_0 .



f(n)	**g(n)**	
$16n^3 + 30n^2 –$ 90	$_n2$	f(n) = Θ(n^2)
7. $2^n + 30n$	2^n	f(n) = Θ (2^n)

Little oh(o): Definition: f(n) = O(g(n)) (read as f of n is little oh of g of n), if f(n) = O(g(n)) and f(n) ≠ Ω(g(n)).

f(n)	**g(n)**	
18n + 9	n^2	f(n) = o(n^2) since f(n) = O(n^2) and f(n) ≠ Ω(n^2) however f(n) ≠ O(n).

Relations Between O, Ω, Θ:

Theorem : For any two functions g(n) and f(n),

f(n) = □(g(n)) iff

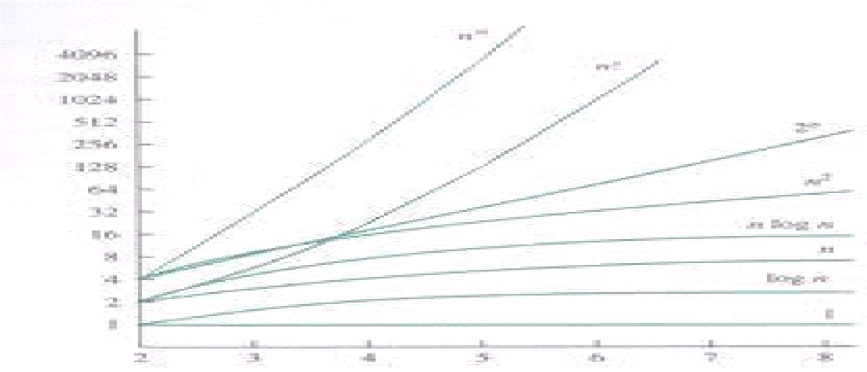
f(n) = O(g(n)) and f(n) = □(g(n)).

Time Complexity:

Complexity	**Notation**	**Description**

Constant	O(1)	Constant number of operations, not depending on the input data size.
Logarithmic	O(logn)	Number of operations proportional of log(n) where n is the size of the input data.
Linear	O(n)	Number of operations proportional to the input data size.
Quadratic	O(n^2)	Number of operations proportional to the square of the size of the input data.
Cubic	O(n^3)	Number of operations proportional to the cube of the size of the input data.
Exponential	O(2^n)	Exponential number of operations, fast growing.
	O(k^n)	
	O(n!)	

Time Complexities of various Algorithms:



Numerical Comparisons of Different Algorithms:

S.No.	**$\log_2 n$**	**n**	**$n\log_2 n$**	**n^2**	**n^3**	**2^n**
1.	0	1	1	1	1	2
2.	1	2	2	4	8	4
3.	2	4	8	16	64	16
4.	3	8	24	64	512	256
5.	4	16	64	256	4096	65536

Reasons for analyzing algorithms:

- To predict the resources that the algorithm requires.
- Computational Time (CPU consumption).
    - Memory Space (RAM consumption).
    - Communication bandwidth consumption.
- To predict the running time of an algorithm
    - Total number of primitive operations executed.

Introduction to data structures:

Whenever we want to work with large amount of data, then organizing that data is very important. If that data is not organized effectively, it is very difficult to perform any task on that data. If it is organized effectively then any operation can be performed easily on that data.

A data structure can be defined as follows...

Data structure is a method of organizing large amount of data more efficiently so that any operation on that data becomes easy. Every data structure is used to organize the large amount of data very data structure follows a particular principle The operations in a data structure should not violate the basic principle of that data structure.

Data structures are divided into two types:

- Primitive data structures.

- Non-primitive data structures.

Primitive Data Structures are the basic data structures that directly operate upon the machine instructions. They have different representations on different computers. Integers, floating point numbers, character constants, string constants and pointers come under this category.

Non-primitive data structures are more complicated data structures and are derived from primitive data structures. They emphasize on grouping same or different data items with relationship between each data item. Arrays, lists and files come under this category.

A data structure is said to be ***linear*** if its elements form a sequence or a linear list. The linear data structures like an array, stacks, queues and linked lists organize data in linear order.

A data structure is said to be ***non linear*** if its elements form a hierarchical classification where, data items appear at various levels.

Trees and Graphs are widely used non-linear data structures. Tree and graph structures represents hierarchial relationship between individual data elements. Graphs are nothing but trees with certain restrictions removed.

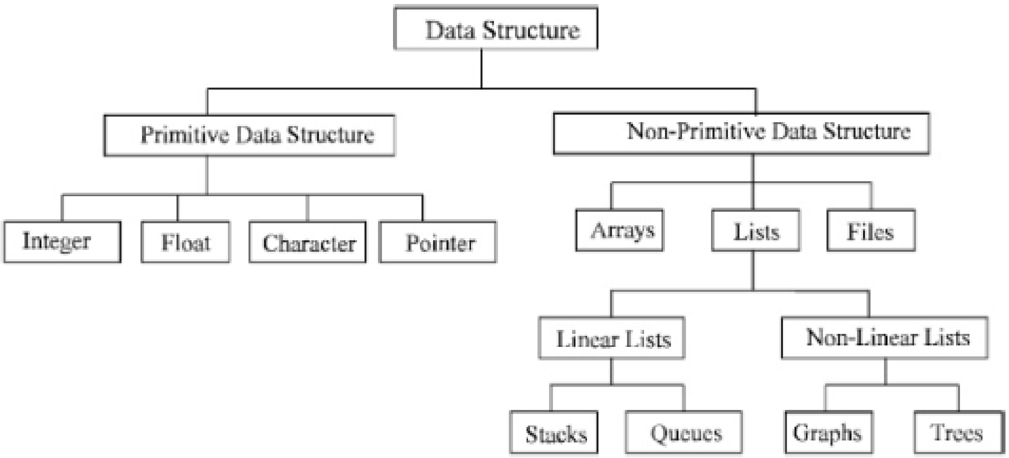


Figure: Classification of Data Structures

Recursion Definition:

- Recursion is a technique that solves a problem by solving a smaller problem of the same type.
- A recursive function is a function invoking itself, either directly or indirectly.
- Recursion can be used as an alternative to iteration.
- Recursion is an important and powerful tool in problem solving and programming.
- Recursion is programming techniques that naturally implements the divide and conquer problem solving methodology.

Four criteria of a Recursive Solution:

1. A recursive function calls itself.
2. Each recursive call solves an identical, but smaller problem.
3. A test for the **base case** enables the recursive calls to stop.
4. There must be a case of the problem(known as **base case** or **stopping case**) that is handled differently from the other cases.
5. In the **base case**, the recursive calls stop and the problem is solved directly.
6. Eventually, one of the smaller problems must be the base case.

Example: To define n! Recursively, n! Must be defined in terms of the factorial of a smaller number. n! = n * (n-1)!

Base case: 0! =1

n! = 1 if n=0

n! = n * (n-1)! if n >0

Designing Recursive Algorithms:

The Design Methodology:

- The statement that solves the problem is known as the **base case**.
- Every recursive algorithm must have a **base case**. The rest of the algorithm is known as the **general case**.
- The **general case** contains the logic needed to reduce the size of the problem.
  Example: in factorial example, the base case is fact(0) and the general case is n * fact(n-1).

Rules for designing a Recursive Algorithm:

1. First, determine the base case.
2. Then determine the general case.
3. Combine the base case and the general cases into an algorithm.

Limitations of Recursion:

1. Recursion works best when the algorithm uses a data structure that naturally supports recursion. E.g. Trees.
2. In other cases, certain algorithms are naturally suited to recursion. E.g. binary search, towers of hanoi.
3. On the other hand, not all looping algorithms can or should be implemented with reursion.
4. Recursive solutions may involve extensive overhead (both time and memory) because they use calls. Each call takes time to execute. A recursive algorithm therefore generally runs more slowly than its nonrecursive implementation.

Recusive Examples:

GCD Design: Given two integers a and b, the greatest common divisor is recursively found using the formula

gcd(a,b) = a	if b=0		
		Base case	
b	if a=0		
gcd(b, a mod b)		General case	

{
Fibonacci(n) = 0 if n=0
1 if n=1
}
Base case

Tracing a Recursive Function:

1. A stack is used to keep track of function calls.
2. Whenever a new function is called, all its parameters and local variables are pushed onto the stack along with the memory address of the calling statement.
3. For each function call, an activation call is created on the stack.

Difference between Recursion and Iteration:

1. A function is said to be recursive if it calls itself again and again within its body whereas iterative functions are loop based imperative functions.
2. Recursion uses stack whereas iteration does not use stack.

1. Recursion uses more memory than iteration as its concept is based on stacks.
2. Recursion is comparatively slower than iteration due to overhead condition of maintaining stacks.
3. Recursion makes code smaller and iteration makes code longer.
4. Iteration terminates when the loop-continuation condition fails whereas recursion terminates when a base case is recognized.
5. While using recursion multiple activation records are created on stack for each call where as in iteration everything is done in one activation record.
6. Infinite recursion can crash the system whereas infinite looping uses CPU cycles repeatedly.
7. Recursion uses selection structure whereas iteration uses repetetion structure.

Types of Recursion:

Recursion is of two types depending on whether a function calls itself from within itself or whether two functions call one another mutually. The former is called **direct recursion** and the later is called **indirect recursion**. Thus there are two types of recursion:

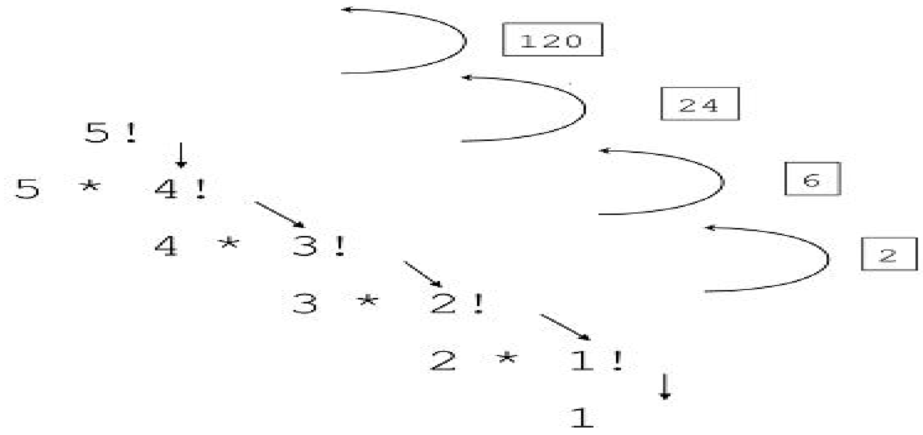
- Direct Recursion
- Indirect Recursion
- Linear Recursion
- Binary Recursion
- Multiple Recursion

Linear Recursion:

It is the most common type of Recursion in which function calls itself repeatedly until base condition [termination case] is reached. Once the base case is reached the results are return to the caller function. If a recursive function is called only once then it is called a linear recursion.

Example1: Finding the factorial of a number.

```
fact(int f)
{
if (f == 1) return 1;
return (f * fact(f - 1)); //called in function only once
}
int main()
{
int fact;
fact = fact(5);
cout<<"Factorial is %d", fact);
return 0;

}
```



Binary Recursion:

Some recursive functions don't just have one call to themselves; they have two (or more). Functions with two recursive calls are referred to as binary recursive functions.

Example1: The Fibonacci function fib provides a classic example of binary recursion. The Fibonacci numbers can be defined by the rule:

fib(n) = 0 if n is 0,

- 1 if n is 1,
- fib(n-1) + fib(n-2) otherwise

For example, the first seven Fibonacci numbers are

Fib(0) = 0

Fib(1) = 1

Fib(2) = Fib(1) + Fib(0) = 1

Fib(3) = Fib(2) + Fib(1) = 2

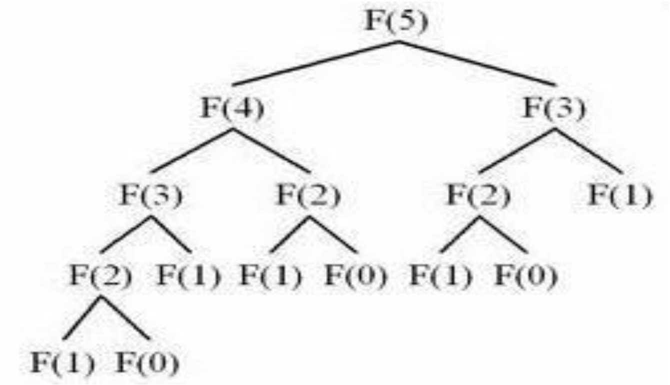
Fib(4) = Fib(3) + Fib(2) = 3

Fib(5) = Fib(4) + Fib(3) = 5

Fib(6) = Fib(5) + Fib(4) = 8

This leads to the following implementation in C:

```
int fib(int n)
{
if (n == 0)
return 0;
if (n == 1)
return 1;
return fib(n - 1) + fib(n - 2);



}
```



Tail Recursion:

Tail recursion is a form of linear recursion. In tail recursion, the recursive call is the last thing the function does. Often, the value of the recursive call is returned. As such, tail recursive functions can often be easily implemented in an iterative manner; by taking out the recursive call and replacing it with a loop, the same effect can generally be achieved. In fact, a good compiler can recognize tail recursion and convert it to iteration in order to optimize the performance of the code.

A good example of a tail recursive function is a function to compute the GCD, or Greatest Common Denominator, of two numbers:

```
int gcd(int m, int n)

{

int r;

if (m < n) return gcd(n,m);

r = m%n;

if (r == 0) return(n);

else return(gcd(n,r));

}
```

Example: Convert the following tail-recursive function into an iterative function:

```
int pow(int a, int b)

{

if (b==1) return a;

else return a * pow(a, b-1);

}

int pow(int a, int b)

{
```

```
int i, total=1;
for(i=0; i<b; i++) total *= a;
return total;
}
```

Recursive algorithms for Factorial, GCD, Fibonacci Series and Towers of Hanoi:

Factorial(n)

```
    Input: integer $n \geq 0$
    Output: $n!$
    If $n = 0$ then return (1)
    else return prod(n, factorial($n - 1$))
```

GCD(m, n)

```
    Input: integers $m > 0$, $n \geq 0$
    Output: gcd (m, n)
    If $n = 0$ then return (m)
    else return gcd(n,m mod n)
    Time-Complexity: $O(\ln n)$
```

Fibonacci(n)

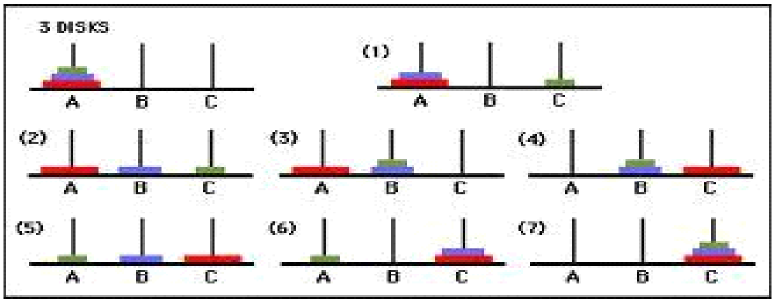
```
    Input: integer $n \geq 0$
    Output: Fibonacci Series: 1  1  2  3  5      8  13………………………………..
    if n=1 or n=2
    then Fibonacci(n)=1
    else Fibonacci(n) = Fibonacci(n-1) + Fibonacci(n-2)
```

Towers of Hanoi

Input: The aim of the tower of Hanoi problem is to move the initial n different sized disks from needle A to needle C using a temporary needle B. The rule is that no larger disk is to be placed above the smaller disk in any of the needle while moving or at any time, and only the top of the disk is to be moved at a time from any needle to any needle.

Output:

- If n=1, move the single disk from A to C and return,
- If n>1, move the top n-1 disks from A to B using C as temporary.
- Move the remaining disk from A to C.
- Move the n-1 disk disks from B to C, using A as temporary.



Basic concepts of Algorithm

Preliminaries of Algorithm:

An algorithm may be defined as a finite sequence of instructions each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time. The algorithm word originated from the Arabic word "Algorism" which is linked to the name of the Arabic mathematician AI Khwarizmi. He is considered to be the first algorithm designer for adding numbers.

Structure and Properties of Algorithm:

An algorithm has the following structure

1. Input Step

2. Assignment Step
3. Decision Step
4. Repetitive Step
5. Output Step

Finiteness: An algorithm must terminate after a finite number of steps.

Definiteness: The steps of the algorithm must be precisely defined or unambiguously specified.

Generality: An algorithm must be generic enough to solve all problems of a particular class.

Effectiveness: the operations of the algorithm must be basic enough to be put down on pencil and paper. They should not be too complex to warrant writing another algorithm for the operation.

Input-Output: The algorithm must have certain initial and precise inputs, and outputs that may be generated both at its intermediate and final steps.

An algorithm does not enforce a language or mode for its expression but only demands adherence to its properties.

- **To save time (Time Complexity):** A program that runs faster is a better program.
- **To save space (Space Complexity):** A program that saves space over a competing program is considerable desirable.

Efficiency of Algorithms:

The performances of algorithms can be measured on the scales of **time** and **space**. The performance of a program is the amount of computer memory and time needed to run a program. We use two approaches to determine the performance of a program. One is analytical and the other is experimental. In performance analysis we use analytical methods, while in performance measurement we conduct experiments.

Time Complexity:

The time complexity of an algorithm or a program is a function of the running time of the algorithm or a program. In other words, it is the amount of computer time it needs to run to completion.

Space Complexity:

The space complexity of an algorithm or program is a function of the space needed by the algorithm or program to run to completion.

The time complexity of an algorithm can be computed either by an **empirical** or **theoretical** approach. The **empirical** or **posteriori testing** approach calls for implementing the complete algorithms and executing them on a computer for various instances of the problem. The time taken by the execution of the programs for various instances of the problem are noted and compared. The algorithm whose implementation yields the least time is considered as the best among the candidate algorithmic solutions.

Analyzing Algorithms:

Suppose M is an algorithm, and suppose n is the size of the input data. Clearly the complexity f(n) of M increases as n increases. It is usually the rate of increase of f(n) with some standard functions. The most common computing times are O(1), O($\log_2$ n), O(n), O(n $\log_2$ n), O(n^2), O(n^3), O(2^n)

Example:

Program Segment A	**Program Segment B**	**Program Segment C**
-------------------------	-------------------------	-------------
-------------		
x =x + 2;	for k =1 to n do	for j =1 to n do
-------------------------		
x =x + 2;	for x = 1 to n do	end;
x =x + 2;	-------------------------	end
end;		

Total Frequency Count of Program Segment A	
Program Statements	**Frequency Count**
-------------------------	
x =x + 2;	1

Total Frequency Count	1
Total Frequency Count of Program Segment B	
Program Statements	**Frequency Count**
-------------------------	
for k =1 to n do	(n+1)
x =x + 2;	n
end;	n

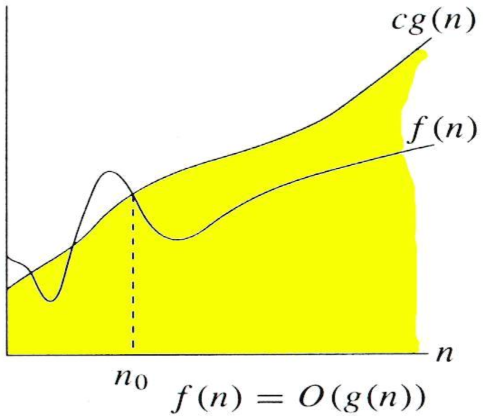
Total Frequency Count	3n+1

The total Frequency counts of the program segments A, B and C given by 1, (3n+1) and ($3n^2$+3n+1) respectively are expressed as O(1), O(n) and O(n^2). These are referred to as the time complexities of the program segments since they are indicative of the running times of the program segments. In a similar manner space complexities of a program can also be expressed in terms of mathematical notations, which is nothing but the amount of memory they require for their execution.

<u>Asymptotic Notations:</u>

It is often used to describe how the size of the input data affects an algorithm's usage of computational resources. Running time of an algorithm is described as a function of input size n for large n.

Big oh(O): Definition: f(n) = O(g(n)) (read as f of n is big oh of g of n) if there exist a positive integer n_0 and a positive number c such that $|f(n)| \leq c|g(n)|$ for all $n \geq n_0$. Here g(n) is the upper bound of the function f(n).

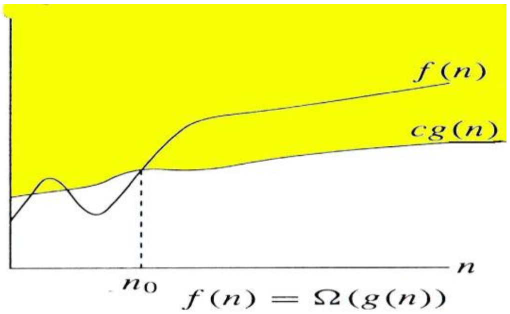


$f(n) = O(g(n))$

	f(n)			g(n)	
	3	2	3		3

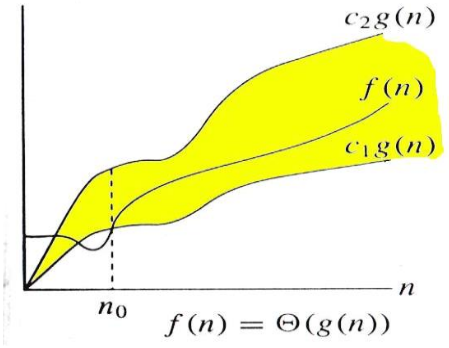
16n	+ 45n	+	n		f(n) = O(n)
12n					
34n – 40			n		f(n) = O(n)
50			1		f(n) = O(1)

Omega(Ω): Definition: f(n) = Ω(g(n)) (read as f of n is omega of g of n), if there exists a positive integer n_0 and a positive number c such that |f(n)| ≥ c |g(n)| for all n ≥ $n_{0.}$ Here g(n) is the lower bound of the function f(n).



f(n)		**g(n)**	
3	2	3	3
16n +8n + 2		n	f(n) = Ω(n)
24n + 9		n	f(n) = Ω(n)

Theta(Θ): Definition: f(n) = Θ(g(n)) (read as f of n is theta of g of n), if there exists a positive integer n_0 and two positive constants c_1 and c_2 such that c_1 |g(n)| ≤ |f(n)| ≤ c_2 |g(n)| for all n ≥ $n_{0.}$ The function g(n) is both an upper bound and a lower bound for the function f(n) for all values of n, n ≥ n_0 .



$f(n) = \Theta(g(n))$

f(n)	**g(n)**	
$16n^3 + 30n^2$ – 90	n2	f(n) = $\Theta(n^2\ )$
7. 2^n + 30n	2^n	f(n) = Θ (2^n)

Little oh(o): Definition: f(n) = O(g(n)) (read as f of n is little oh of g of n), if f(n) = O(g(n)) and f(n) ≠ Ω(g(n)).

f(n)	**g(n)**	
18n + 9	n^2	f(n) = o(n^2) since f(n) = O(n^2) and f(n) ≠ Ω(n^2) however f(n) ≠ O(n).

<u>Relations Between O, Ω, Θ</u>:

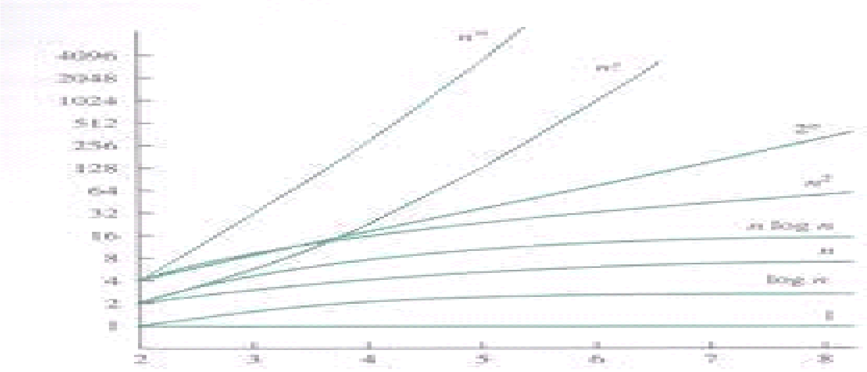
Theorem : For any two functions g(n) and f(n), f(n) = □(g(n)) iff

f(n) = O(g(n)) and f(n) = □(g(n)).

Time Complexity:

Complexity	**Notation**	**Description**
Constant	O(1)	Constant number of operations, not depending on the input data size.
Logarithmic	O(logn)	Number of operations proportional of log(n) where n is the size of the input data.
Linear	O(n)	Number of operations proportional to the input data size.
Quadratic	$O(n^2)$	Number of operations proportional to the square of the size of the input data.
Cubic	$O(n^3)$	Number of operations proportional to the cube of the size of the input data.
Exponential	$O(2^n)$ $O(k^n)$ O(n!)	Exponential number of operations, fast growing.

Time Complexities of various Algorithms:



Numerical Comparisons of Different Algorithms:

S.No.	$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n
1.	0	1	1	1	1	2
2.	1	2	2	4	8	4
3.	2	4	8	16	64	16
4.	3	8	24	64	512	256
5.	4	16	64	256	4096	65536

Reasons for analyzing algorithms:

- To predict the resources that the algorithm requires
  - Computational Time (CPU consumption).
  - Memory Space (RAM consumption).
  - Communication bandwidth consumption.
- To predict the running time of an algorithm
  - Total number of primitive operations executed.

Recursion Definition:

- Recursion is a technique that solves a problem by solving a smaller problem of the same type.
- A recursive function is a function invoking itself, either directly or indirectly.
- Recursion can be used as an alternative to iteration.
- Recursion is an important and powerful tool in problem solving and programming.
- Recursion is a programming technique that naturally implements the divide and conquer problem solving methodology.

Four criteria of a Recursive Solution:

1. A recursive function calls itself.
2. Each recursive call solves an identical, but smaller problem.
3. A test for the **base case** enables the recursive calls to stop.
4. There must be a case of the problem(known as **base case** or **stopping case**) that is handled differently from the other cases.
5. In the **base case**, the recursive calls stop and the problem is solved directly.
6. Eventually, one of the smaller problems must be the base case.

Example: To define n! Recursively, n! Must be defined in terms of the factorial of a smaller number. n! = n * (n-1)!

Base case: 0! =1

n! = 1 if n=0

n! = n * (n-1)! if n >0

Designing Recursive Algorithms:

The Design Methodology:

- The statement that solves the problem is known as the **base case**.
- Every recursive algorithm must have a **base case**. The rest of the algorithm is known as the **general case**.
- The **general case** contains the logic needed to reduce the size of the problem.

Example: in factorial example, the base case is fact(0) and the general case is n * fact(n-1).

Rules for designing a Recursive Algorithm:

1. First, determine the base case.
2. Then determine the general case.
3. Combine the base case and the general cases into an algorithm.

Limitations of Recursion:

1. Recursion works best when the algorithm uses a data structure that naturally supports recursion. E.g. Trees.
2. In other cases, certain algorithms are naturally suited to recursion. E.g. binary search, towers of hanoi.
3. On the other hand, not all looping algorithms can or should be implemented with reursion.
4. Recursive solutions may involve extensive overhead (both time and memory) because they use calls. Each call takes time to execute. A recursive algorithm therefore generally runs more slowly than its nonrecursive implementation.

Recusive Examples:

GCD Design: Given two integers a and b, the greatest common divisor is recursively found using the formula

gcd(a,b) = a	if b=0		

		Base case	
b	if a=0		
gcd(b, a mod b)		General case	

Fibonacci(n) = 0 if n=0

1 if n=1

Base case

Tracing a Recursive Function:

1. A stack is used to keep track of function calls.
2. Whenever a new function is called, all its parameters and local variables are pushed onto the stack along with the memory address of the calling statement.
3. For each function call, an activation call is created on the stack.

Difference between Recursion and Iteration:

1. A function is said to be recursive if it calls itself again and again within its body whereas iterative functions are loop based imperative functions.
2. Recursion uses stack whereas iteration does not use stack.

1. Recursion uses more memory than iteration as its concept is based on stacks.
2. Recursion is comparatively slower than iteration due to overhead condition of maintaining stacks.
3. Recursion makes code smaller and iteration makes code longer.
4. Iteration terminates when the loop-continuation condition fails whereas recursion terminates when a base case is recognized.
5. While using recursion multiple activation records are created on stack for each call where as in iteration everything is done in one activation record.
6. Infinite recursion can crash the system whereas infinite looping uses CPU cycles repeatedly.
7. Recursion uses selection structure whereas iteration uses repetetion structure.

Types of Recursion:

Recursion is of two types depending on whether a function calls itself from within itself or whether two functions call one another mutually. The former is called **direct recursion** and the later is called **indirect recursion**. Thus there are two types of recursion:

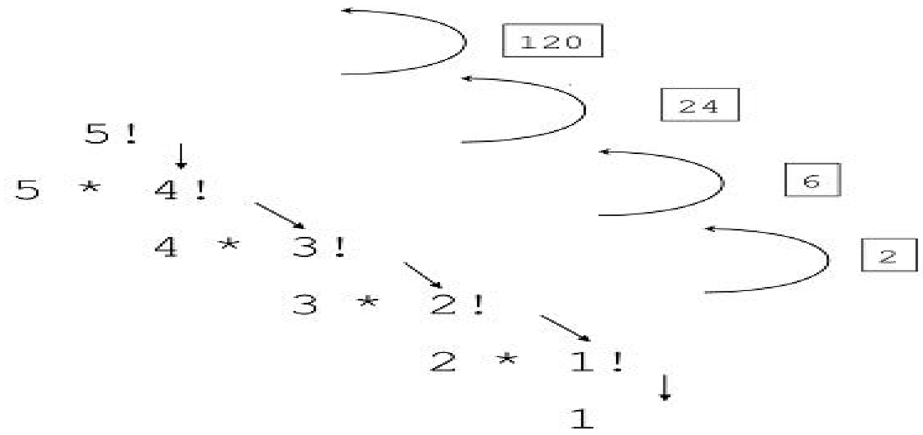
- Direct Recursion
- Indirect Recursion
- Linear Recursion
- Binary Recursion
- Multiple Recursion

Linear Recursion:

It is the most common type of Recursion in which function calls itself repeatedly until base condition [termination case] is reached. Once the base case is reached the results are return to the caller function. If a recursive function is called only once then it is called a linear recursion.

Example1: Finding the factorial of a number.

```
fact(int f)
{
if (f == 1) return 1;
return (f * fact(f - 1)); //called in function only once
}
int main()
{
int fact;
fact = fact(5);
cout<<"Factorial is %d", fact);
return 0;

}
```



Binary Recursion:

Some recursive functions don't just have one call to themselves; they have two (or more). Functions with two recursive calls are referred to as binary recursive functions.

Example1: The Fibonacci function fib provides a classic example of binary recursion. The Fibonacci numbers can be defined by the rule:

fib(n) = 0 if n is 0,

- 1 if n is 1,
- fib(n-1) + fib(n-2) otherwise

For example, the first seven Fibonacci numbers are

Fib(0) = 0

Fib(1) = 1

Fib(2) = Fib(1) + Fib(0) = 1

Fib(3) = Fib(2) + Fib(1) = 2

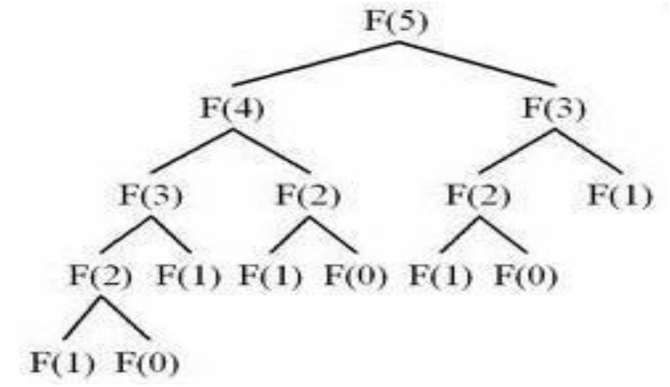
Fib(4) = Fib(3) + Fib(2) = 3

Fib(5) = Fib(4) + Fib(3) = 5

Fib(6) = Fib(5) + Fib(4) = 8

This leads to the following implementation in C:

```
int fib(int n)

{

if (n == 0)

return 0;

if (n == 1)

return 1;

return fib(n - 1) + fib(n - 2);





}
```



Tail Recursion:

Tail recursion is a form of linear recursion. In tail recursion, the recursive call is the last thing the function does. Often, the value of the recursive call is returned. As such, tail recursive functions can often be easily implemented in an iterative manner; by taking out the recursive call and replacing it with a loop, the same effect can generally be achieved. In fact, a good compiler can recognize tail recursion and convert it to iteration in order to optimize the performance of the code.

A good example of a tail recursive function is a function to compute the GCD, or Greatest Common Denominator, of two numbers:

```
int gcd(int m, int n)

{

int r;

if (m < n) return gcd(n,m);

r = m%n;

if (r == 0) return(n);

else return(gcd(n,r));

}
```

Example: Convert the following tail-recursive function into an iterative function:

```
int pow(int a, int b)

{

if (b==1) return a;

else return a * pow(a, b-1);

}

int pow(int a, int b)

{
```

```
int i, total=1;
for(i=0; i<b; i++) total *= a;
return total;
}
```

Recursive algorithms for Factorial, GCD, Fibonacci Series and Towers of Hanoi:

Factorial(n)

```
    Input: integer $n \geq 0$
    Output: n!
    If n = 0 then return (1)
    else return prod(n, factorial($n - 1$))
```

GCD(m, n)

```
    Input: integers $m > 0, n \geq 0$
    Output: gcd (m, n)
    If n = 0 then return (m)
    else return gcd(n,m mod n)
    Time-Complexity: O(ln n)
```

Fibonacci(n)

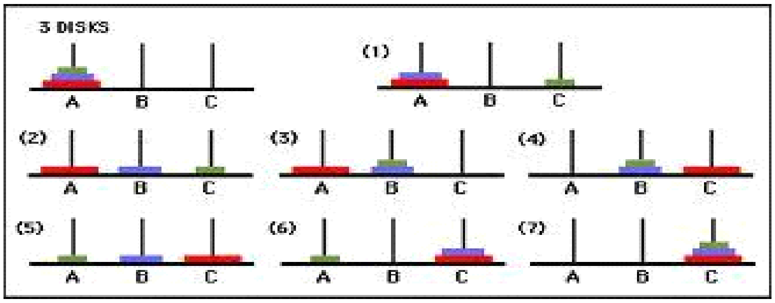
```
    Input: integer $n \geq 0$
    Output: Fibonacci Series: 1  1  2  3  5      8  13………………………………..
    if n=1 or n=2
    then Fibonacci(n)=1
    else Fibonacci(n) = Fibonacci(n-1) + Fibonacci(n-2)
```

Towers of Hanoi

Input: The aim of the tower of Hanoi problem is to move the initial n different sized disks from needle A to needle C using a temporary needle B. The rule is that no larger disk is to be placed above the smaller disk in any of the needle while moving or at any time, and only the top of the disk is to be moved at a time from any needle to any needle.

Output:

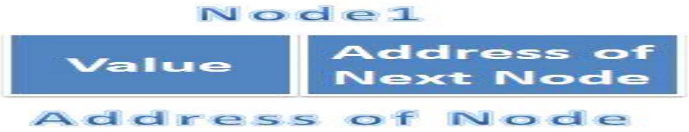
- If n=1, move the single disk from A to C and return,
- If n>1, move the top n-1 disks from A to B using C as temporary.
- Move the remaining disk from A to C.
- Move the n-1 disk disks from B to C, using A as temporary.



MODULE-II: Linked Lists [09 Periods] Single Linked Lists: **Definition, Operations-Insertion, Deletion and Searching, Concatenating single linked lists, Circular linked lists, Operations- Insertion, Deletion.** Double Linked Lists: **Definition, Operations- Insertion, Deletion. Applications of Linked list. Sparse matrices - Array and linked representations.**

Linked List Introduction:

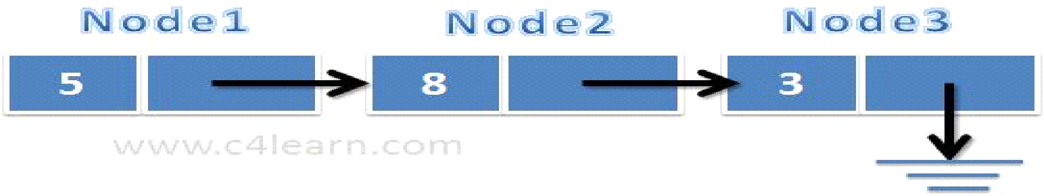
- When we want to work with unknown number of data values, we use a linked list data structure to organize that data.
- Linked list is a linear data structure that contains sequence of elements such that each element links to its next element in the sequence.
- Each element in a linked list is called as "Node".



<u>Linked List Definition:</u>

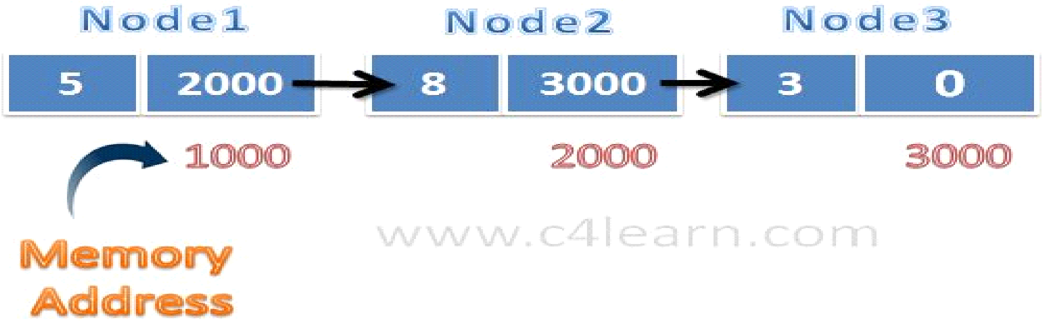
- Linked List is **<u>Series of Nodes</u>**
- Each node Consist of two Parts Data **<u>Part & Pointer Part</u>**

- Pointer Part stores the **address of the next node**



Linked List node:

- Each Linked List Consists of Series of Nodes
- In above diagram , **Linked List Consists of three nodes A,B,C**
- Node A has two part one data part which consists of the 5 as data and the second part which contain the address of the next node (**i.e it contain the address of the next node B**)



Advantages:

- Linked List is **Dynamic Data Structure**.
- Linked List **can grow and shrink during run time**.
- **Insertion and Deletion** Operations are Easier
- **Efficient Memory Utilization** , i.e no need to pre-allocate memory
- Faster Access time, can be expanded in **constant time without memory overhead**

- Linear Data Structures such as Stack, Queue can be **easily implemented** using Linked list.

Need of Linked List:

Suppose we are writing a program which will store marks of 10 students in maths. Then our logic would be like this during compile time –

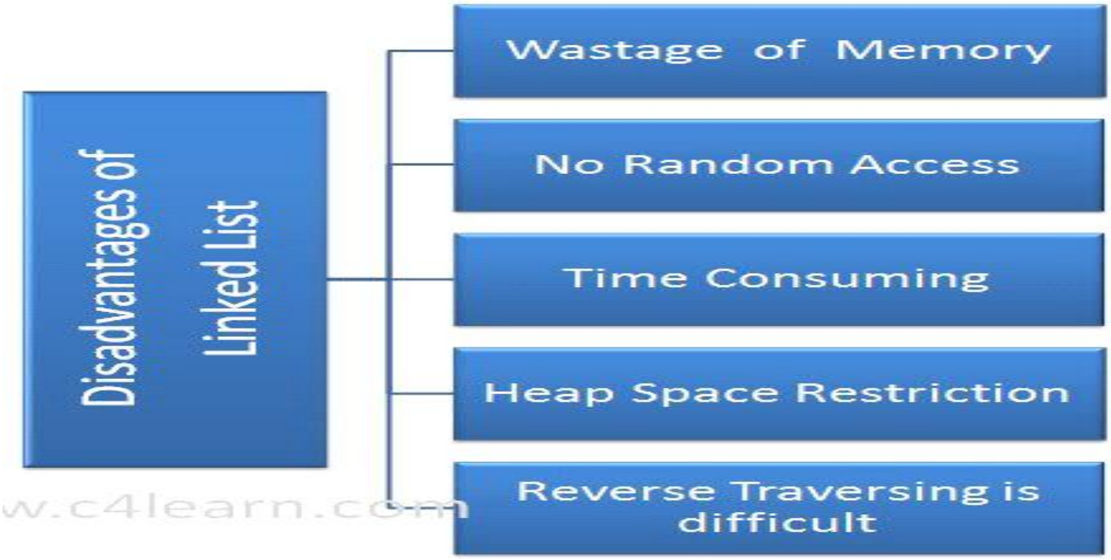
```
int marks[10];
```

Now at run time i.e after executing program if number of students are 11 then how we will store the address of 11th student ?

Or if we need to store only 5 students then again we are wasting memory unnecessarily.

Using linked list we can create memory at run time or free memory at run time so that we will be able to fulfil our need in efficient manner.

Drawbacks in linked lists:



1. Wastage of Memory –

- Pointer Requires **extra memory for storage**.

- Suppose we want to store 3 integer data items then we have to allocate memory –

in case of array –

Memory Required in Array = 3 Integer * Size

= 3 * 4 bytes

= 12 bytes

in case of array –

Memory Required in LL = 3 Integer * Size of Node

= 3 * Size of Node Structure

= 3 * Size (data + address pointer)

= 3 * (4 bytes + x bytes)

= 12 bytes + 3x bytes

*x is size of complete node structure it may vary

2. No Random Access

- In array we can access **nth element easily just by using a[n]**.
- In Linked list no random access is given to user, we have to **access each node sequentially**.
- Suppose we have to access nth node then **we have to traverse linked list n times**.

Suppose Element is present at the starting location then –

We can access element in first Attempt

Suppose Element is present at the Last location then –

We can access element in last Attempt

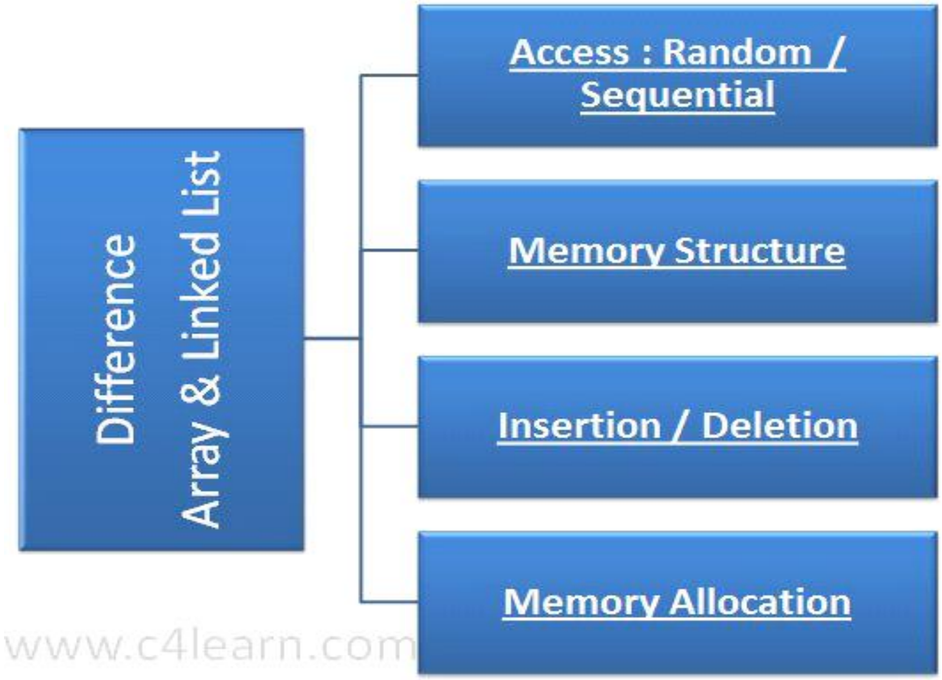
3 . Time Complexity

- Array can be randomly accessed , while the Linked list **cannot be accessed Randomly**
- Individual nodes are not stored in the **contiguous memory Locations**.
- Access time for Individual Element is **O(n)** whereas in Array it is O(1).

4. Reverse Traversing is difficult

- In case if we are using singly linked list then it is **very difficult to traverse linked list from end**.
- If using doubly linked list then though it becomes easier to traverse from end but **still it increases again storage space for back pointer**.

Difference Between array and Linked List : Array Vs Linked List:



1. Access :Random / Sequential

- Array elements can be randomly Accessed using **Subscript Variable**
- e.g a[0],a[1],a[3] can be **randomly** accessed
- While In Linked List We have to Traverse Through the Linked List for Accessing Element. So **O(n) Time** required for Accessing Element .
- Generally In linked List Elements are accessed **Sequentially**.

2 . Memory Structure :

- Array is stored in **contiguous Memory** Locations , i.e Suppose first element is Stored at 2000 then Second Integer element will be stored at 2002 .

- But It is not necessary to store **next element** at the Consecutive memory Location .
- Element is stored at any available Location , but the **Pointer** to that memory location is stored in Previous Node.

3 . Insertion / Deletion

- As the **Array** elements are stored in **Consecutive memory** Locations , so While Inserting elements ,we have to create space for Insertion.
- So More time required for **Creating space** and Inserting Element
- Similarly We have to Delete the Element from given Location and then **Shift All successive elements up by 1 position**
- In Linked List we have to Just Change the Pointer address field (Pointer), So Insertion and Deletion Operations are quite easy to implement

4 . Memory Allocation :

- Memory Should be allocated at **Compile-Time in Stack** . i.e at the time when Programmer is Writing Program
- In **Linked list** memory can be allocated at **Run-Time** , i.e After executing Program
- Array uses Static Memory Allocation and Linked List Uses Dynamic Memory Allocation

Singly Linked List

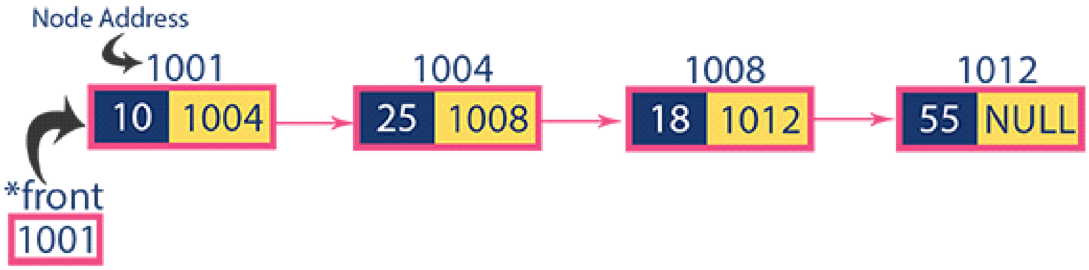
What is singly linked list? Explain its operations in detail.

- A list is a sequence of data, and linked list is a sequence of data linked with each other.

The formal definition of a single linked list is as follows...

- Single linked list is a sequence of elements in which every element has link to its next element in the sequence.
- In any single linked list, the individual element is called as "Node".
- Every "Node" contains two fields, data and next.
- The data field is used to store actual value of that node and next field is used to store the address of the next node in the sequence.
- In a single linked list, the address of the first node is always stored in a reference node known as "front" (also known as "head" or “start”).
- Always next part (reference part) of the last node must be NULL.

The graphical representation of a node in a single linked list is as follows…



In a single linked list we perform the following operations...

- Insertion
- Deletion
- Display

Before we implement actual operations, first we need to setup empty list. First perform the following steps before implementing actual operations.

Step 1: Include all the header files which are used in the program.

Step 2: Define a Node structure with two members data and next

Step 3: Declare all the user defined functions.

Step 4: Implement the main method by displaying operations menu and make suitable function calls in the main method to perform user selected operation.

Explanation:

Step 1: Include all the header files which are used in the program.

- We don't know, how many nodes user is going to create once he execute the program.
- In this case we are going to allocate memory using Dynamic Memory Allocation functions such as Alloc & Malloc.
- Dynamic memory allocation functions are included in alloc.h
- Finally we willl use stdio.h, alloc.h and stdlib.h header files here.

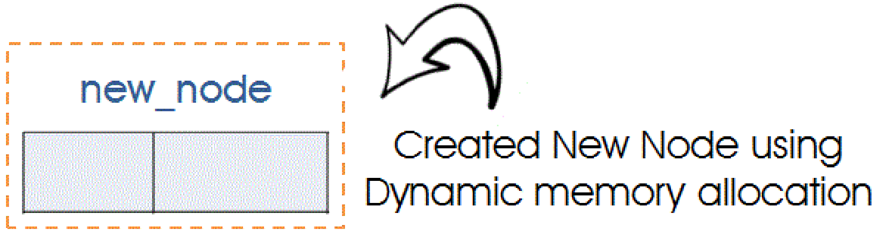
Step 2: Define a Node structure with two members data and next

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;
```

Create node using dynamic memory allocation

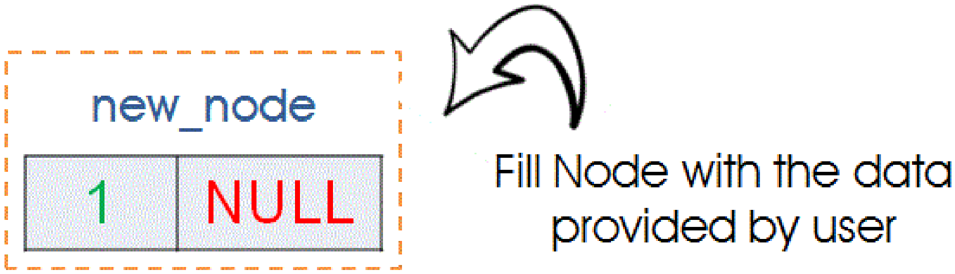
```
new_node=(struct node *)malloc(sizeof(struct node));
```



Created New Node using Dynamic memory allocation

Now fill information in newly created node

```
printf("\nEnter the data : ");
scanf("%d",&newnode->data);
newnode->next=NULL;
```



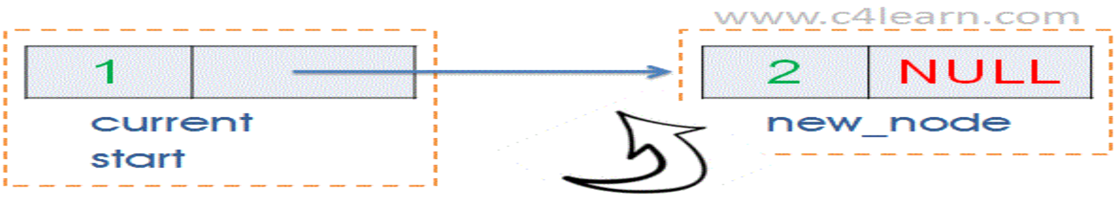
Fill Node with the data provided by user

If we want to create second node or nth node then

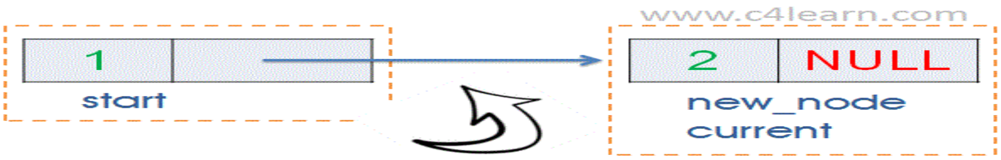
- Lets assume we have 1 node already created i.e we have first node. First node can be referred as “newnode”,”head”,”start”.
- Now we have called create() function again

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

```
else
  {
current->next = newnode;
current = newnode;
  }
```



Making Link between Current and new_node



Now current pointer is pointing to new node i.e 2nd Node

Display Singly Linked List from First to Last

- In order to write the program for display, We must create a linked list using create().

- Traversal Starts from **Very First node**. We cannot modify the address stored inside global variable “**start**” thus we have to declare one temporary variable -“**temp**” of type node.

- In order to traverse from start to end you should **assign Address of Starting node** in **Pointer variable** i.e temp

```
struct node *temp;  //Declare temp
temp = start;       //Assign Starting Address to temp
```

Now we are checking the value of pointer (i.e temp). If the temp is NULL then we can say that last node is reached.

```
while(temp!=NULL)
  {
    printf("|%d|%d|",temp->data,temp->next);
    temp=temp->next;
  }
```

Complete Display Function :

```
void display()
{
   struct node *temp;
   if(start==NULL)
   {
           printf(“empty List!\n”);
   }
  else {
       temp=start;
      while(temp!=NULL)
      {
       printf("|%d|%d|",temp->data,temp->next);
       temp=temp->next;
     }
} }
```

Insert node at Start/First Position in Singly Linked List:

Inserting node at start in the SLL (Steps):

We can use the following steps to insert a new node at beginning of the single linked list...

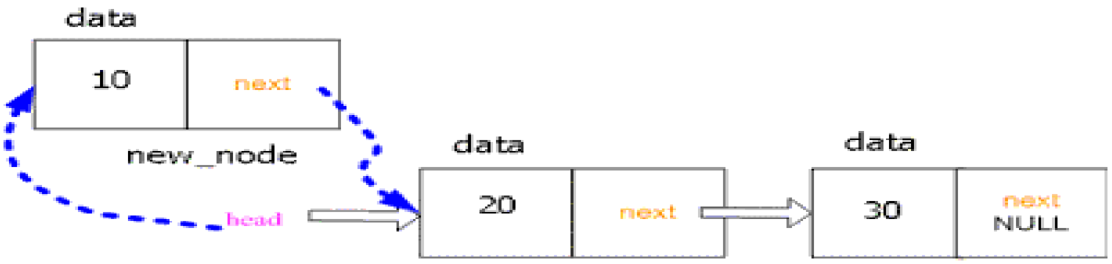
Step 1: Create a newnode with given value.

Step 2: Check whether list is Empty (start == NULL)

Step 3: If it is Empty then, set newnode→next = NULL and start = newnode.

Step 4: If it is Not Empty then,

set newnode→next = start and start = newnode.



- If starting node is not available, then **“Start = NULL”** then following part is executed

```
if(start==NULL)
    {
    start=newnode;
    current=newnode;
    }
```

- If we have previously created First or starting node then **<u>“else part”</u>** will be executed to insert node at start

```
    else
    {
     newnode->next=start;
     start=newnode;
    }
```

<u>Insert at begin snippet or code</u>

```
void insertAtBegin()
{
  newnode=(struct node*)malloc(sizeof(struct node));
  printf("\nEnter Element:");
  scanf("%d",&newnode->data);
```

```
    newnode->next=NULL;
    if(start==NULL)
    {
      start=newnode;
      current=newnode;
   }

else
   {

    newnode->next=start;
    start=newnode;
   }
}
```

Insert node at Last / End Position in Singly Linked List:

We can use the following steps to insert a new node at end of the single linked list...

Step 1: Create a newNode with given value and newNode → next as NULL.

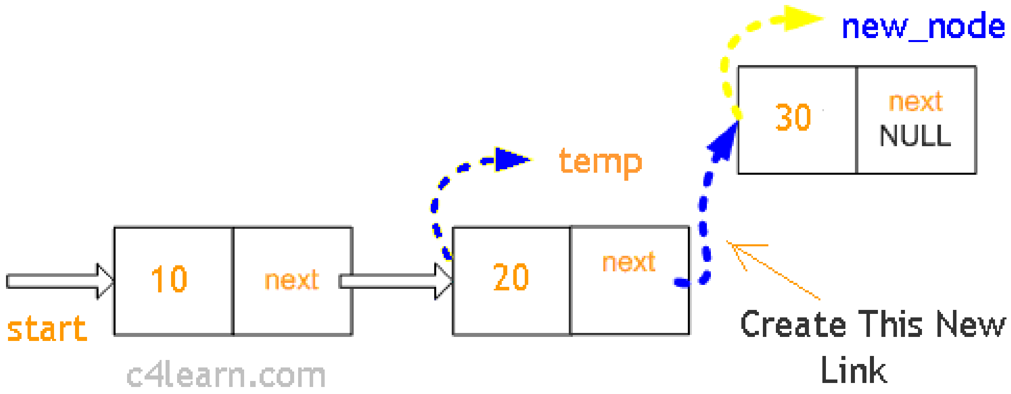
Step 2: Check whether list is Empty (start == NULL).

Step 3: If it is Empty then, set start = newNode.

Step 4: If it is Not Empty then, define a node pointer temp and initialize with head.

Step 5: Keep moving the temp to its next node until it reaches to the last node in the list (until temp → next is equal to NULL).

Step 6: Set temp → next = newNode.



- If starting node is not available then **“Start = NULL”** then following part is executed

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

- If we have previously created First or starting node then **<u>“else part”</u>** will be executed to insert node at start
- Traverse Upto Last Node., So that **<u>temp</u>** can keep track of Last node

```
else
{
  temp = start;
  while(temp->next!=NULL)
  {
    temp = temp->next;
  }
```

- Make **Link between Newly Created node and Last node** (temp)

```
temp->next = newnode;
```

<u>Insert at end snippet or code</u>

```
void insertAtEnd()
{
    newnode=(struct node*)malloc(sizeof(struct node));
     printf("nEnter the data : ");
     scanf("%d",&newnode->data);
     newnode->next=NULL;
     if(start==NULL)
     {
       start=newnode;
     }
     else

       {
       temp = start;
         while(temp->next!=NULL)
         {
          temp = temp->next;
         }
       temp->next = newnode;
    }
}
```

Inserting At Specific location in the list (After a Node):

We can use the following steps to insert a new node after a node in the single linked list...

Step 1: Create a newNode with given value.

Step 2: Check whether list is Empty (start == NULL)

Step 3: If it is Empty then, set newNode → next = NULL
and start = newNode.

Step 4: If it is Not Empty then, define a node pointer temp and initialize with start.

Step 5: Keep moving the temp to its next node until it reaches to the node after which we want to insert the newNode (until temp1 → data is equal to location, here location is the node value after which we want to insert the newNode).

Step 6: Finally, Set

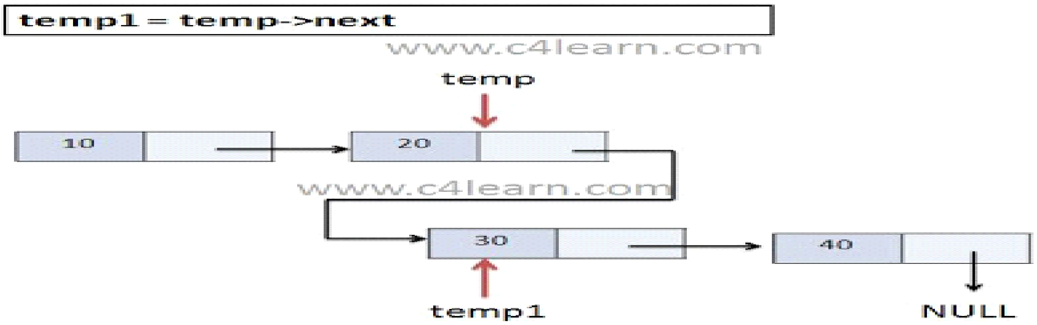
```
      temp1=temp->next;

     temp->next = newnode;

     newnode->next=temp1;
```

Insert at given position snippet or code

```
void insertAtPOS()
{
    int pos,i;
    newnode=(struct node *)malloc(sizeof(struct node));
    printf("nEnter the data : ");
    scanf("%d",&newnode->data);
    newnode->next=NULL;
    printf("nEnter the position : ");
    scanf("%d",&pos);
    if(start==NULL)
    {
       start=newnode;
     //  current=newnode;
    }
    else
    {
             temp = start;
             for(i=1;i< pos;i++)
             {
              temp = temp->next;
             }
       temp1=temp->next;
       temp->next = newnode;
    newnode->next=temp1;
  }
}
```

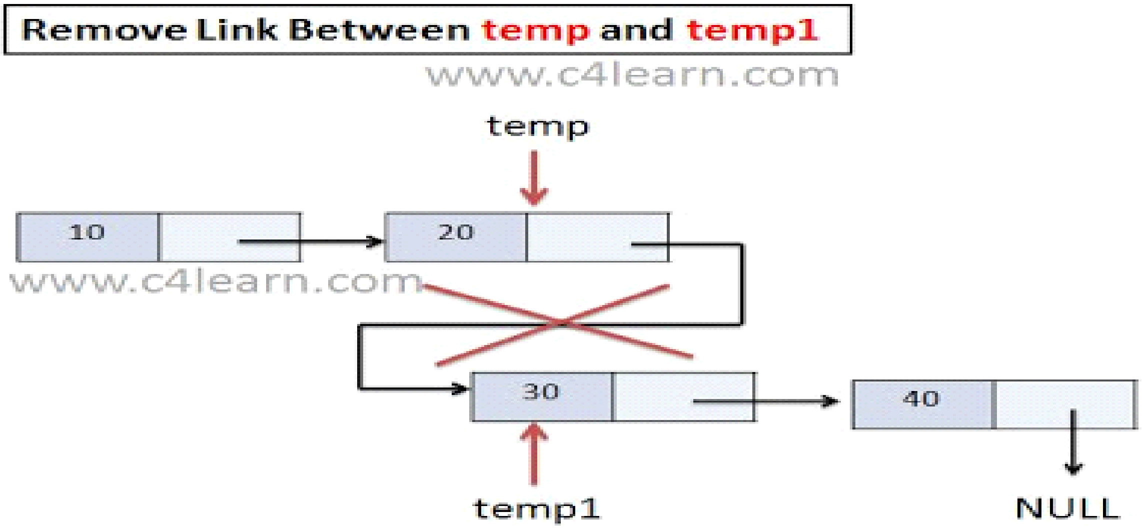
Explanation :
Step 1 : Get Current Position Of “temp” and “temp1” Pointer.

```
temp = start;
 for(i=1;i< pos-1;i++)
 {
 temp = temp->next;
 }
```



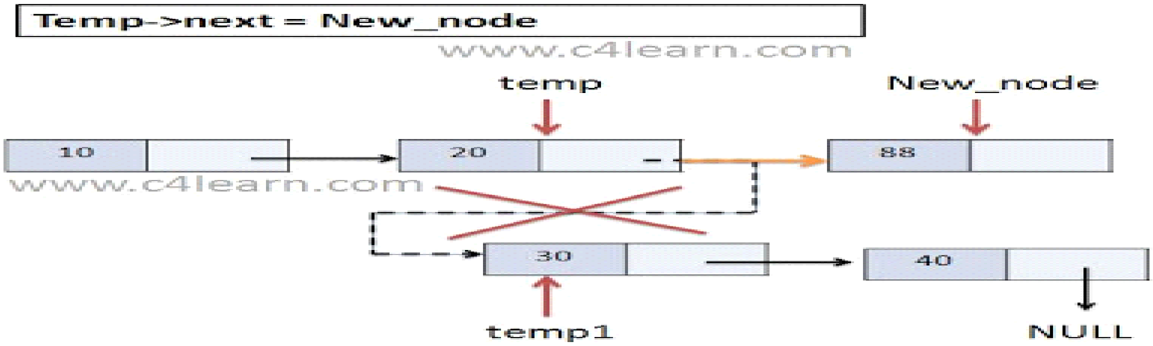
Step 2 :

```
temp1=temp->next;
```



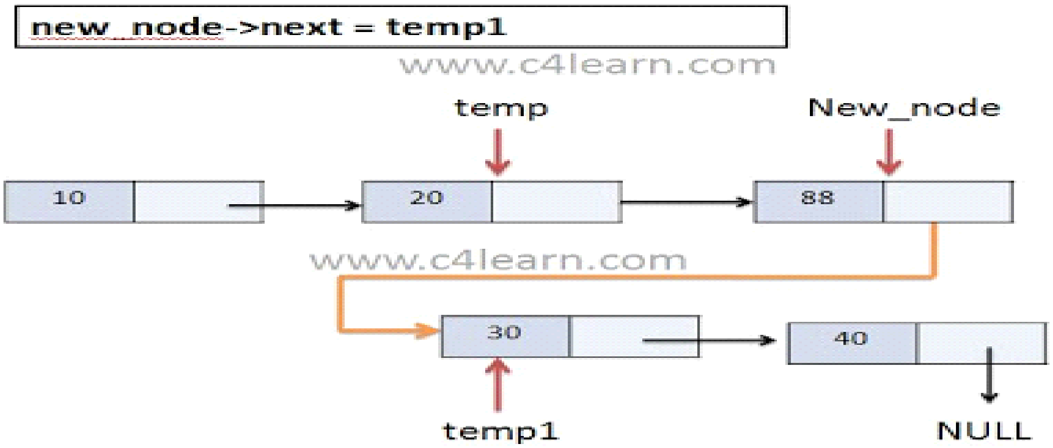
Step 3 :

temp->next = new_node;



Step 4 :

new_node->next = temp1



Delete First Node from Singly Linked List:

We can use the following steps to delete a node from beginning of the single linked list...

Step 1: Check whether list is Empty (start == NULL)

Step 2: If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.

Step 3: If it is Not Empty then, define a Node pointer 'temp' and initialize with start.

Step 4: Check whether list is having only one node (temp → next == NULL)

Step 5: If it is TRUE then set start = NULL and
delete temp (Setting Empty list conditions)

Step 6: If it is FALSE then set start = temp → next, and delete temp.

Delete at begin snippet or code

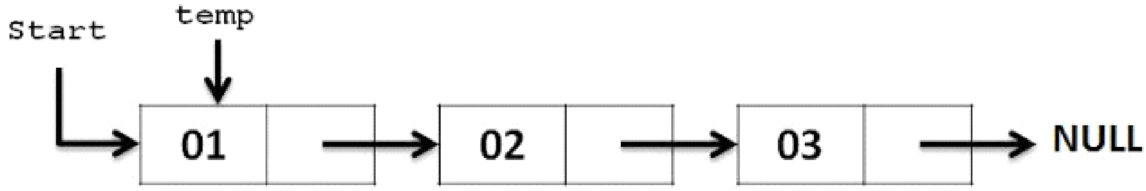
```
void deleteBegin()
```

```
{
   if(start==NULL)
       printf("empty list!delete not possible\n");
   else
   {
    temp = start;
    start = start->next;
    if(temp->next==NULL)
    {
        start=NULL;
    }
    free(temp);
    printf("\nThe Element deleted Successfully ");

   }
}
```

Explanation:

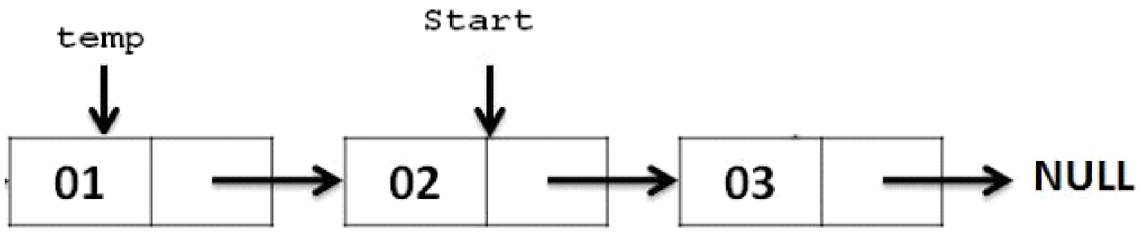
Step 1 : Store Current Start in Another Temporary Pointer

```
temp = start;
```



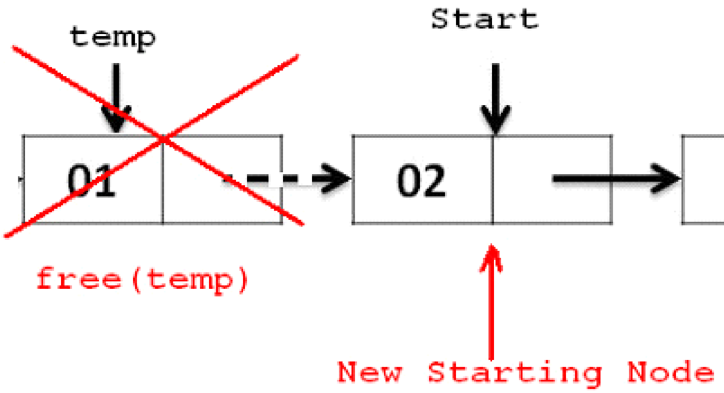
Step 2 : Move Start Pointer **One position Ahead**

```
start = start->next;
```



Step 3 : Delete temp i.e Previous Starting Node as we have Updated Version of Start Pointer

free(temp);



Deleting from End of the list:

We can use the following steps to delete a node from end of the single linked list...

Step 1: Check whether list is Empty (start == NULL)

Step 2: If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.

Step 3: If it is Not Empty then, define two Node pointers 'temp' and 'temp1' and initialize 'temp' with start.

Step 4: Check whether list has only one Node (temp → next == NULL)

Step 5: If it is TRUE. Then, set start = NULL and delete temp. And terminate the function. (Setting Empty list condition)

Step 6: If it is FALSE. Then, set 'temp1 = temp ' and move temp to its next node. Repeat the same until it reaches to the last node in the list. (until temp → next == NULL)

Step 7: Finally, Set temp1 → next = NULL and delete temp.

Delete at end snippet or code

```
void deleteAtEnd()
{
 if(start==NULL)
      printf("empty list!delete not possible\n");
 else
 {
   temp=start;
   while(temp->next != NULL)
   {
    temp1=temp;
    temp=temp->next;
   }
```

```
    free(temp1->next);
    temp1->next=NULL;
  }
}
```

Deleting a Specific Node from the list:

We can use the following steps to delete a specific node from the single linked list...

Step 1: Check whether list is Empty (start == NULL)

Step 2: If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.

Step 3: If it is Not Empty then, define two Node pointers 'temp' and 'temp1' and initialize 'temp' with start.

Step 4: Keep moving the temp until it reaches to the exact node to be deleted or to the last node. And every time set 'temp1 = temp' before moving the 'temp' to its next node.

Step 5: If it is reached to the last node then display 'Given node not found in the list! Deletion not possible!!!'. And terminate the function.

Step 6: If it is reached to the exact node which we want to delete, then check whether list is having only one node or not

Step 7: If list has only one node and that is the node to be deleted, then set start = NULL and delete temp (free(temp)).

Step 8: If list contains multiple nodes, then check whether temp is the first node in the list (temp == start).

Step 9: If temp is the first node then move the start to the next node (start= start → next) and delete temp.

Step 10: If temp is not first node then check whether it is last node in the list (temp → next == NULL).

Step 11: If temp is last node then set temp1 → next = NULL and delete temp (free(temp)).

Step 12: If temp is not first node and not last node then set temp1 → next = temp1 → next and delete temp (free(temp)).

<u>Delete at position snippet or code</u>

```
void deleteatPos()
{

    //struct node *temp,*temp1;
    int pos, count=1;
    printf("\nEnter the Position of the element you would like to delete:");
    scanf("%d",&pos);
    temp=temp1=start;
    while(temp!=NULL)
    {
        temp1=temp;
        temp=temp->next;
        count++;
        if(count>=pos)
           break;
    }
    if(pos==1)
    {
        start=temp1->next;
        printf("\nExceuted-->First Node is deleted!!");
        free(temp1);
    }
    else if(temp==current)
    {
    temp1->next=NULL;
    free(temp);
        printf("\nExecuted-->Last Node is deleted!!");
    }
    else
    {
        temp1->next=temp->next;
        free(temp);
        printf("\nExecuted-->Node has been deleted!!");
    }
}
```

//C PROGRAM TO IMPLEMENT SINGLY LINKED LIST & OPERATIONS

```
#include<stdio.h>
#include<stdlib.h>
void create();
void display();
int menu();
void insertAtBegin();
void insertAtEnd();
void insertAtPOS();
void deleteBegin();
void deleteAtEnd();
void deleteatPos();
struct node
{
    int data;
    struct node *next;
}*start=NULL,*current=NULL,*newnode=NULL,*temp=NULL,*temp1=NULL;
int main()
{
  int ch;
  while(1)
  {
      ch=menu();
      switch(ch)
      {
          case 1: create();break;

          case 2: display();break;

          case 3: exit(0);

          case 4: insertAtBegin();break;

          case 5: insertAtEnd();break;

          case 6: insertAtPOS();break;

          case 7: deleteBegin();break;

          case 8: deleteAtEnd();break;

          case 9: deleteatPos();break;

          default : printf("enter valid choice");

      }

  }

}

int menu()

{


    int ch;

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

    printf("\nSingle Linked List");

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


    printf("\n1.Create\n2display\n3.Exit\n 4. insertAtBegin\n 5. insert at end\n6:insert at pos\n");

    printf("\n7:deleteFromFirst Position\n8:delete at end\n9:delete at pos\n");

    printf("\n\n-->Enter Your Choice:");

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


}
void create()
{
    newnode=(struct node*)malloc(sizeof(struct node));
    printf("\nEnter the Data in the Node:");
    scanf("%d",&newnode->data);
    newnode->next=NULL;
    if(start==NULL)
    {
        start=newnode;
        current=newnode;
    }
    else
    {
        current->next=newnode;
        current=newnode;
    }



}
void display()
{
    temp=start;
    while(temp!=NULL)
    {
      printf("|%d|%d| --> ",temp->data,temp->next);
        temp=temp->next;
    }
}
void insertAtBegin()
{

    newnode=(struct node*)malloc(sizeof(struct node));
    printf("\nEnter Element:");
    scanf("%d",&newnode->data);
    newnode->next=NULL;
    if(start==NULL)
    {
      start=newnode;
      current=newnode;
   }
   else
   {

    newnode->next=start;
    start=newnode;
   }
```

```
}



void insertAtEnd()
{
   newnode=(struct node*)malloc(sizeof(struct node));

    printf("nEnter the data : ");
    scanf("%d",&newnode->data);
    newnode->next=NULL;

    if(start==NULL)
    {
      start=newnode;
      //current=newnode;
    }
    else
    {
      temp = start;
        while(temp->next!=NULL)
        {
         temp = temp->next;
        }
```

```
       temp->next = newnode;
      }
  }
  void insertAtPOS()
  {
     int pos,i;
     newnode=(struct node *)malloc(sizeof(struct node));
     printf("nEnter the data : ");
     scanf("%d",&newnode->data);
     newnode->next=NULL;
     printf("nEnter the position : ");
     scanf("%d",&pos);
     if(start==NULL)
     {
        start=newnode;
        current=newnode;
     }
     else
     {
             temp = start;
             for(i=1;i< pos;i++)
             {
              temp = temp->next;
             }
        temp1=temp->next;
```

```
}

       temp->next = newnode;

       newnode->next=temp1;

   }

}





void deleteBegin()

{

    if(start==NULL)

        printf("empty list!delete not possible\n");

    else

    {

     temp = start;

     start = start->next;

     if(temp->next==NULL)

     {

         start=NULL;

     }

     free(temp);

     printf("nThe Element deleted Successfully ");


    }

}
```

```
void deleteAtEnd()
{
  if(start==NULL)
        printf("empty list!delete not possible\n");
  else
  {
    temp=start;
    while(temp->next != NULL)
    {
     temp1=temp;
     temp=temp->next;
    }
    free(temp1->next);
    temp1->next=NULL;
  }
}
void deleteatPos()
{

    //struct node *temp,*temp1;
    int pos, count=1;
    printf("\nEnter the Position of the element you would like to delete:");
    scanf("%d",&pos);
    temp=temp1=start;
    while(temp!=NULL)
```

```
{
    temp1=temp;
    temp=temp->next;
    count++;
    if(count>=pos)
       break;
}

if(pos==1)
{
    start=temp1->next;
    printf("\nExceuted-->First Node is deleted!!");
    free(temp1);
}
else if(temp==current)
{
temp1->next=NULL;
free(temp);
    printf("\nExecuted-->Last Node is deleted!!");
}
else
{
        temp1->next=temp->next;

        free(temp);

        printf("\nExecuted-->Node has been deleted!!");

    }

}
```

Output:

```
------------

Single Linked List

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

1.Create

2display

3.Exit

 4. insertAtBegin

 5. insert at end

6:insert at pos


7:deleteFromFirst Position

8:delete at end

9:delete at pos



-->Enter Your Choice:4
```

```
Enter Element:4


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

Single Linked List

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

1.Create

2display

3.Exit

 4. insertAtBegin

 5. insert at end

6:insert at pos


7:deleteFromFirst Position

8:delete at end

9:delete at pos



-->Enter Your Choice:4


Enter Element:3


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

Single Linked List

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

1.Create
```

```
2display

3.Exit

 4. insertAtBegin

 5. insert at end

6:insert at pos


7:deleteFromFirst Position

8:delete at end

9:delete at pos




-->Enter Your Choice:4


Enter Element:2


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

Single Linked List

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

1.Create

2display

3.Exit

 4. insertAtBegin

 5. insert at end

6:insert at pos
```

```
7:deleteFromFirst Position

8:delete at end

9:delete at pos



-->Enter Your Choice:2

|2|6425952| --> |3|6425920| --> |4|0| -->

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

Single Linked List

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

1.Create

2display


3.Exit

 4. insertAtBegin

 5. insert at end

6:insert at pos


7:deleteFromFirst Position

8:delete at end

9:delete at pos



-->Enter Your Choice:5

nEnter the data : 10
```

```
------------
Single Linked List
------------
1.Create
2display
3.Exit
 4. insertAtBegin
 5. insert at end
6:insert at pos

7:deleteFromFirst Position
8:delete at end
9:delete at pos



-->Enter Your Choice:2
|2|6425952| --> |3|6425920| --> |4|6425984| --> |10|0| -->
------------
Single Linked List
------------
1.Create
2display
3.Exit
 4. insertAtBegin
 5. insert at end
6:insert at pos

7:deleteFromFirst Position
8:delete at end
9:delete at pos


-->Enter Your Choice:6
nEnter the data : 222
nEnter the position : 2

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

Single Linked List
------------
1.Create
2display
3.Exit
 4. insertAtBegin
 5. insert at end
6:insert at pos

7:deleteFromFirst Position
8:delete at end
```

```
9:delete at pos



-->Enter Your Choice:2
|2|6425952| --> |3|6426000| --> |222|6425920| --> |4|6425984| --> |10|0| -->
------------
Single Linked List
------------
1.Create
2display
3.Exit
 4. insertAtBegin
 5. insert at end
6:insert at pos

7:deleteFromFirst Position
8:delete at end
9:delete at pos



-->Enter Your Choice:7
nThe Element deleted Successfully
------------
Single Linked List
------------
```

```
1.Create
2display
3.Exit
 4. insertAtBegin
 5. insert at end
6:insert at pos

7:deleteFromFirst Position
8:delete at end
9:delete at pos


-->Enter Your Choice:2

|3|6426000| --> |222|6425920| --> |4|6425984| --> |10|0| -->
------------
Single Linked List
------------
1.Create
2display
3.Exit
 4. insertAtBegin
 5. insert at end
6:insert at pos
```

```
7:deleteFromFirst Position

8:delete at end

9:delete at pos



-->Enter Your Choice:8


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

Single Linked List

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

1.Create

2display

3.Exit

 4. insertAtBegin

 5. insert at end

6:insert at pos


7:deleteFromFirst Position

8:delete at end

9:delete at pos



-->Enter Your Choice:2

|3|6426000| --> |222|6425920| --> |4|0| -->

------------
```

```
Single Linked List

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

1.Create

2display

3.Exit

 4. insertAtBegin

 5. insert at end

6:insert at pos


7:deleteFromFirst Position

8:delete at end

9:delete at pos




-->Enter Your Choice:9


Enter the Position of the element you would like to delete:2


Executed-->Node has been deleted!!

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

Single Linked List

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

1.Create

2display
```

```
3.Exit
 4. insertAtBegin
 5. insert at end
6:insert at pos

7:deleteFromFirst Position
8:delete at end
9:delete at pos



-->Enter Your Choice:2
|3|6425920| --> |4|0| -->
------------
Single Linked List
------------
1.Create
2display
3.Exit
 4. insertAtBegin
 5. insert at end
6:insert at pos

7:deleteFromFirst Position
8:delete at end
9:delete at pos
```

```
-->Enter Your Choice:3


Process returned 0 (0x0)   execution time : 73.199 s

Press any key to continue.
```

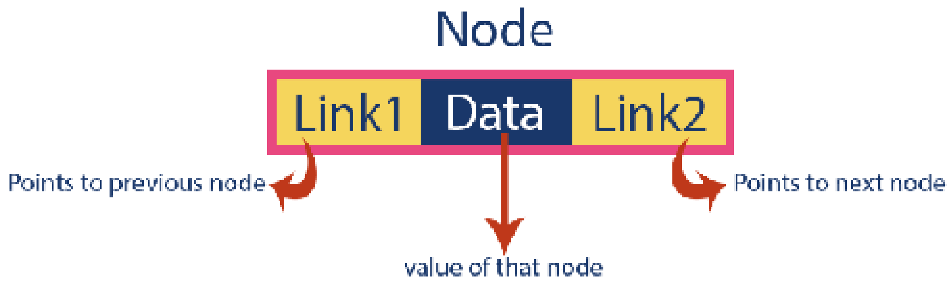
DOUBLY LINKED LIST

What is doubly linked list. Explain its operations.

- In a single linked list, every node has link to its next node in the sequence. So, we can traverse from one node to other node only in one direction and we can not traverse back.
- We can solve this kind of problem by using double linked list.

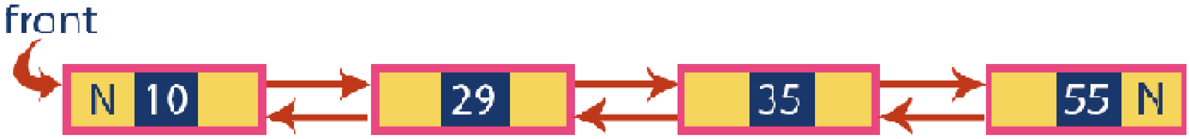
Double linked list can be defined as follows...

- Double linked list is a sequence of elements in which every element has links to its previous element and next element in the sequence.
- In double linked list, every node has link to its previous node and next node.
- So, we can traverse forward by using next field and can traverse backward by using previous field.
- Every node in a double linked list contains three fields and they are shown in the following figure...



-
- Here, 'link1' field is used to store the address of the previous node in the sequence, 'link2' field is used to store the address of the next node in the sequence and 'data' field is used to store the actual value of that node.

Example



☀ In double linked list, the first node must be always pointed by head.
☀ Always the previous field of the first node must be NULL.
☀ Always the next field of the last node must be NULL.

Operations on DLL:

Insertion

Deletion

Display

Insertion:

In a double linked list, the insertion operation can be performed in three ways as follows...

- Inserting At Beginning of the list
- Inserting At End of the list
- Inserting At Specific location in the list

1.Inserting At Beginning of the list:

We can use the following steps to insert a new node at beginning of the double linked list...

Step 1: Create a newNode with given value and
newNode → previous as NULL.

Step 2: Check whether list is Empty (start == NULL)

Step 3: If it is Empty then, assign NULL to newNode → next and newNode to start.

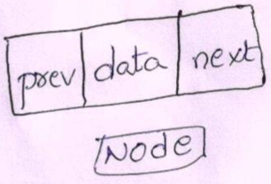
Step 4: If it is not Empty then, assign start to newNode → next and newNode to start.

```
void insertatbegin()

{

   newnode=(struct node*)malloc(sizeof(struct node));

   printf("\nEnter the Data in the Node:");

   scanf("%d",&newnode->data);

   newnode->next=NULL;

   newnode->prev=NULL;


   if(start==NULL)

   {

      newnode->next==NULL;

      start=newnode;

   }

   else

   {

      newnode->next=start;

      start=newnode;

   }

}
```

Doubly Linked Lists

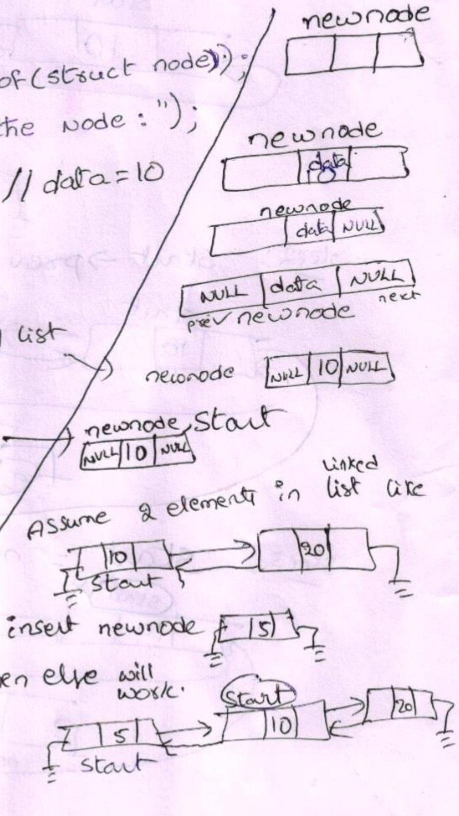
Creating node:-

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

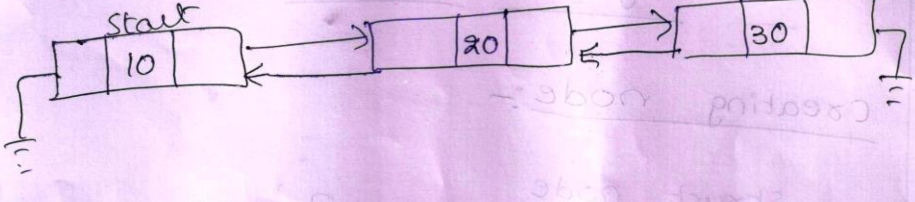


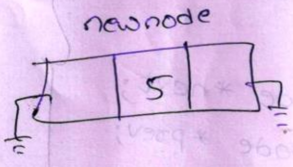
Insert at begin:-

```
insertat begin(){
newnode = (Struct node *) malloc ( sizeof(struct node));
printf (" \n Enter the data in the node : ");
scanf(" %d", & newnode -> data);   // data=10
newnode -> next = NULL;
newnode -> prev = NULL;
if (Start = = NULL) //empty linked list
{
    //newnode -> next = NULL;
    start = newnode;
}
else
{
    newnode -> next = start;
    start -> prev = newnode;
    Start = newnode;
}
}
```

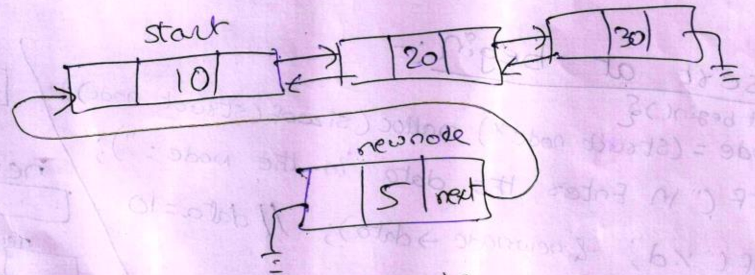


More detail:

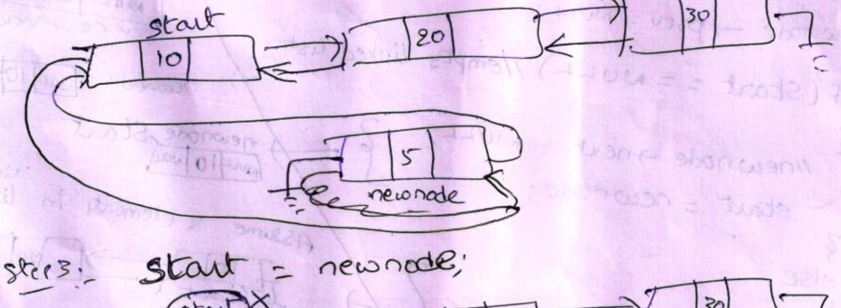




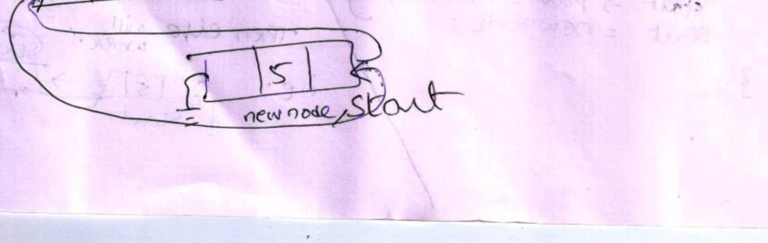
step1:- newnode -> next = start.



step2:- start -> prev = newnode.



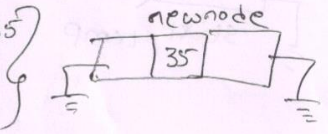
step3:- start = newnode;

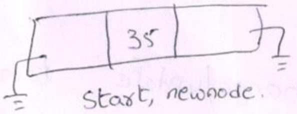


DLL

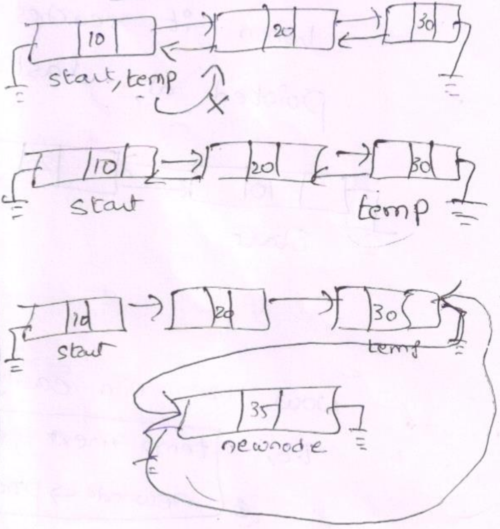
Insert at end :-

```
void insertatend(){
newnode = (struct node *) malloc (sizeof (struct node));

printf (" Enter data");

scanf (" %d", &newnode->data); //35
newnode ->next = NULL;
newnode -> prev = NULL;
if (start = = NULL)
{
     start = newnode;
}
else
{
   temp = start;
   while(temp->next != NULL)
   {
       temp = temp->next;
   } //After this temp will
    //reach last node.
   temp ->next = newnode
   newnode ->prev = temp;
}
}
```

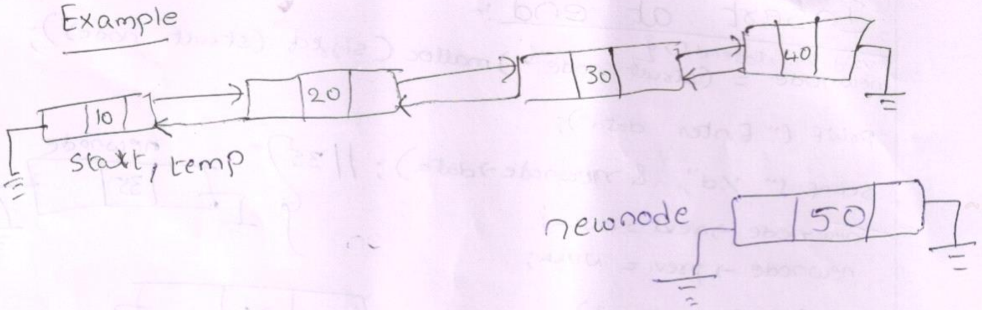




Example



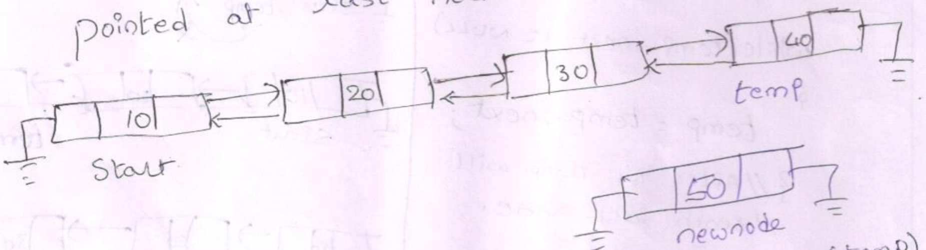
Example



Change/update temp = temp → next

until temp → next reaches NULL.

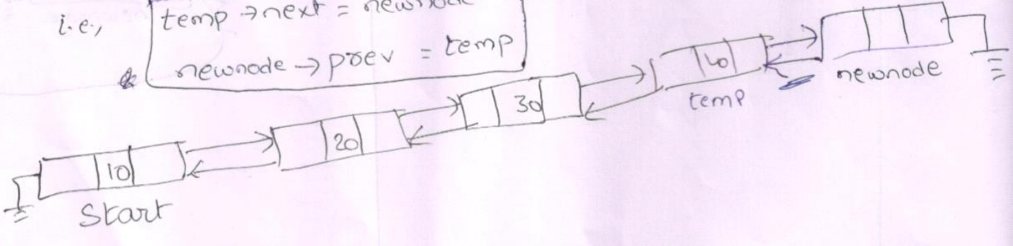
When it reaches NULL that means, temp is pointed at last node of the list.



Now we can easily give links to last node (temp) to newnode.

i.e., temp → next = newnode.

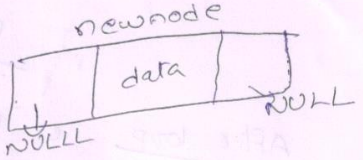
newnode → prev = temp



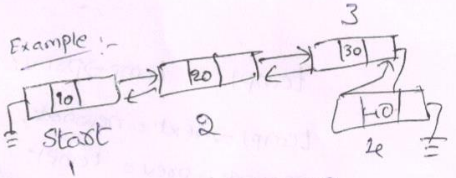
Insert at pos:-

```
void insertatPos()
{
  int i, pos;
  newnode = (struct node*)malloc(sizeof
                    (struct node));
  printf("Enter the data");
  scanf("%d", &newnode->data);
  newnode->prev = NULL;
  newnode->next = NULL;
  printf("Enter the position");
  scanf("%d", &pos);
  if(start == NULL)
  {
     start = newnode;
  }
  else
  {
     temp = start;
     for(i=1; i<pos; i++)
     {
        temp = temp->next;
     }
     templ = temp->prev;
     templ->next = newnode;
     newnode->prev = templ;
     newnode->next = temp;
     temp->prev = newnode;
  }
}
```

DLL



Example :-

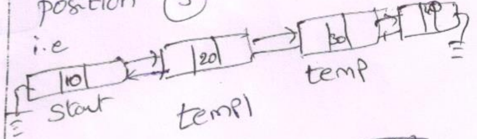


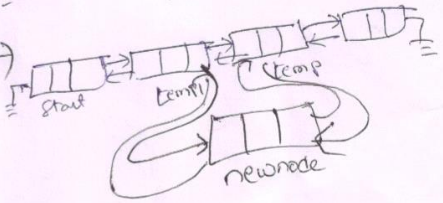
Assume ; position = 3

temp = start.

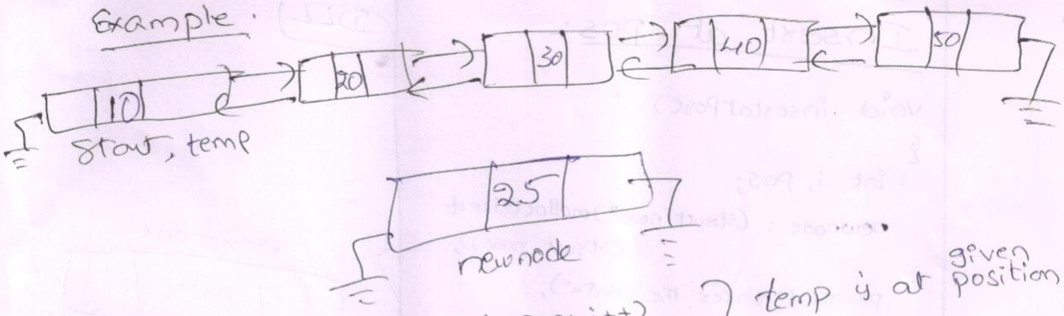
After this loop temp will be located at position 3

i.e





Example.

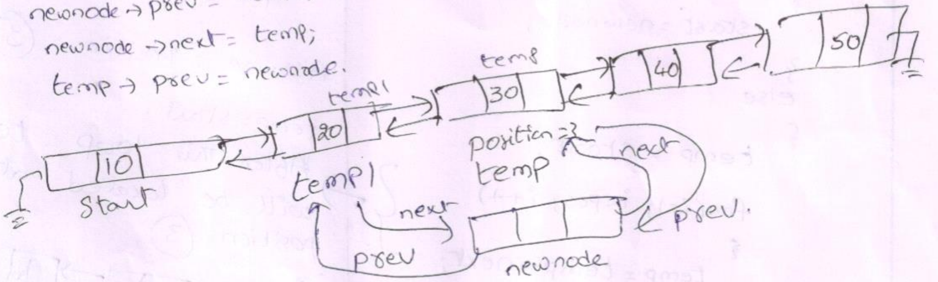


After loop i.e

```
for(i=1; i<pos; i++)
{  temp = temp->next;
}
```

} temp is at given position.

```
temp1 = temp->prev;
temp1->next = newnode;
newnode->prev = temp1;
newnode->next= temp;
temp->prev= newnode.
```



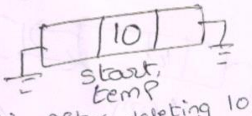
Delete at begin:-

```
void deleteatbegin()
{
 if (start == NULL)
 {
  printf("empty list");
 }
 else
 {
   temp = start;
   if(temp -> prev == temp -> next)
   {
     start = NULL;
     free(temp);
   }
   else
   {
     start = temp -> next;
     start -> prev = NULL;
     free(temp);
   }
 }
}
```

if empty linked list
i.e start == NULL
then empty.

only one element in the list.

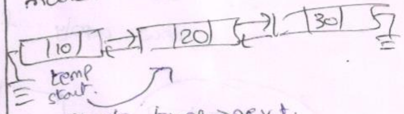
i.e



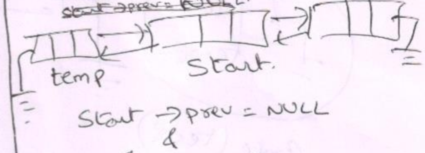
for this, After deleting 10, start is NULL.

i.e $\boxed{temp \rightarrow prev == temp \rightarrow next}$
NULL NULL.

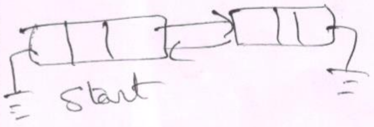
more than one element



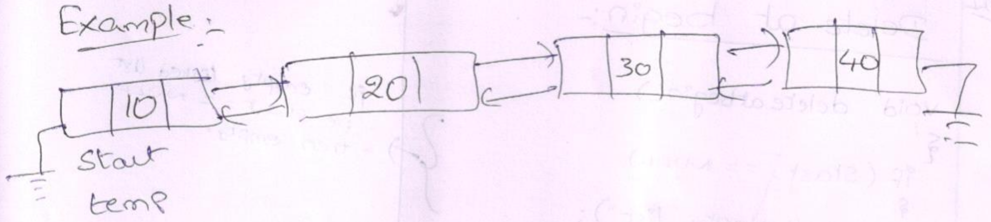
Start = temp -> next.
~~start -> prev = NULL.~~



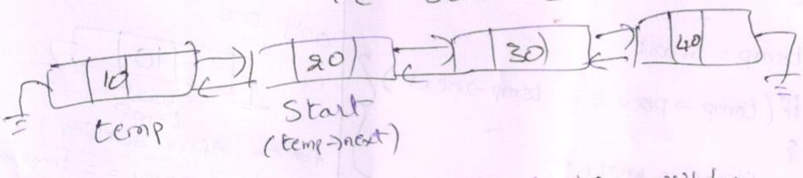
Start -> prev = NULL
&
free(temp)



Example :-

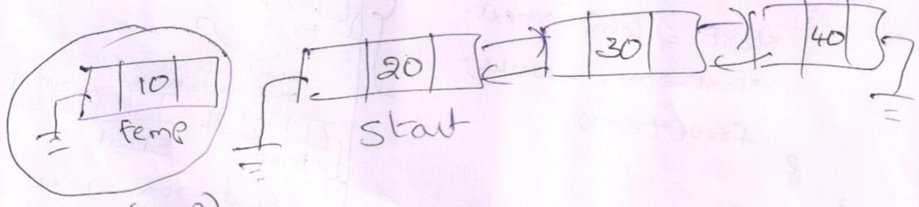


change the start position to second.
i.e start = temp→next



now a Starting list prev must be NULL.

∴ start→prev = NULL;
now apply free(temp) to delete first node.



free(temp)

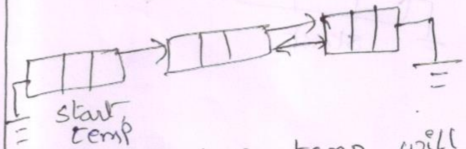
DLL

Delete at end

```
deletatend()
{
  if (start == NULL)
  {
    printf("empty list");
  }
  else
  {
     temp = start;
     if ( temp->prev == temp->next)
     {
         start = NULL;
         free (temp);
     }
     else
     {
      while(temp->next != NULL)
      {
        temp = temp->next;
      }
      temp->prev->next = NULL;
      free(temp);
     }
  }
}
```

print empty list i.e no elements to delete

only one element in the list.



→ After this loop, temp will be at end of the list

by using this statement, we are assigning NULL to last before node. and then free the temp.

(or)

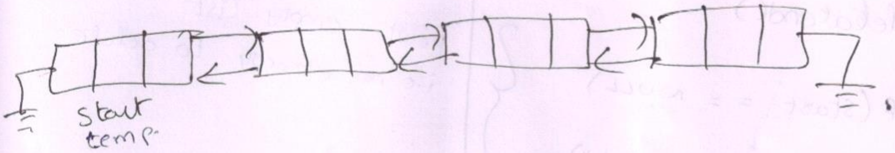
temp1 = temp->prev;
temp1->next = NULL;
free(temp)

P.T.O.

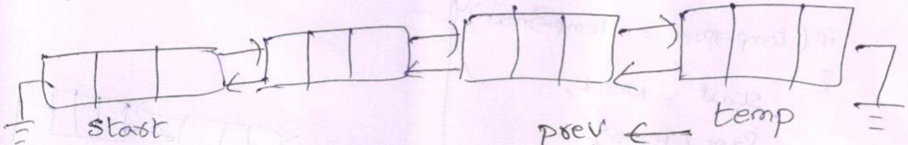
is same as

temp->prev->next = NULL;
free(temp);

Example :-



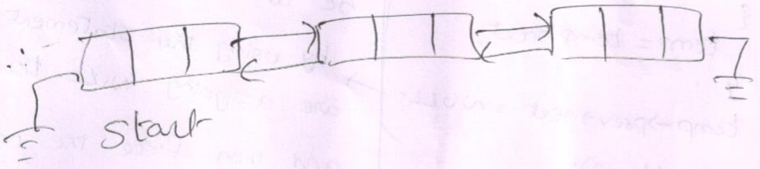
```
while (temp->next != NULL)
{
    temp = temp->next;
}
```



prev ⟵
⟶next = NULL

temp → prev → next = NULL

free(temp) // last node is deleted



DLL

Delete at Position

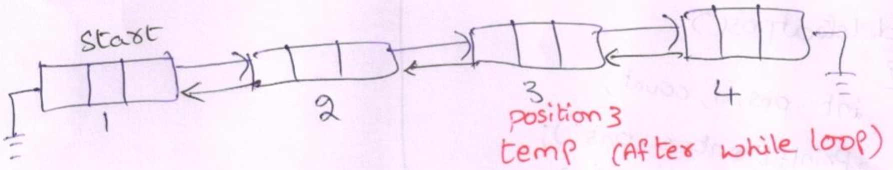
```
deleteatpos()
{
  int pos, i, count;
  printf("enter pos");
  scanf("%d", &pos);
  temp = start;
  if(start == NULL)
  {
   printf(" List is empty");
  }
  else
  {
   if(temp->prev == temp->next)
   {
     start = NULL;
     free(temp);
   }
   else
   {
    for(i=1,count=1; i<pos; i++)
    {
      temp = temp->next;
      count++;
    }
    if(count == pos)
    { deleteatend();
    }
    else
    { temp1 = temp->prev;
      temp1->next = temp->next;
      temp->next->prev = temp1;
      free(temp);
    }
   }
}
```

if list is empty

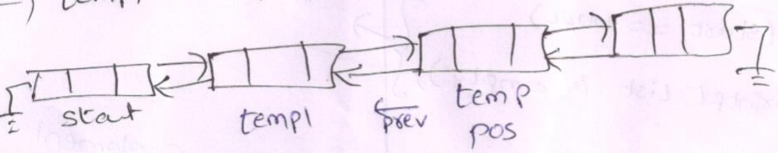
only one element in the list, then after deleting start is assigned to NULL.

if the given position is last element, then we will call delete at end function (deleteatend())

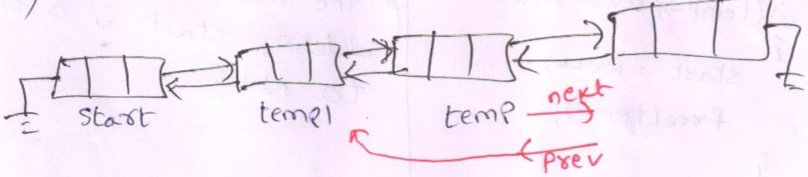
P.T.O



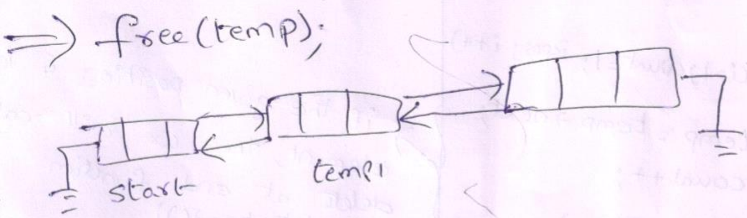
⟹ temp1 = temp→prev;



⟹ temp→next→prev = temp1;



⟹ free(temp);

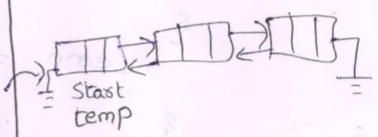


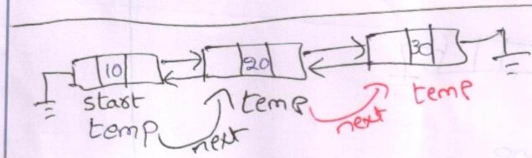
temp is deleted

Display from begin|head|start : DLL

```
display()
{
  temp = start;
  if (temp == NULL)
  {
     printf("empty list");
  }
  while (temp != NULL)
  {
     printf("%d", temp->data);
     temp = temp->next;
  }
}
```

if list is empty





```
while(temp != NULL)
{
  printf("%d", temp->data);   // 10
  temp = temp->next;          // 20
}                             // 30
```

After # 30;
temp reaches
NULL & loop
exits.

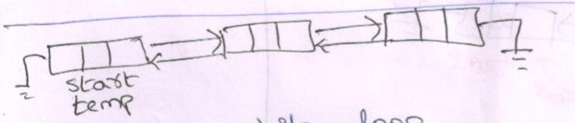
Display from last :- (Reverse of list)

```
void displayreverse( )
{
   temp = start;
   if(temp == NULL)
   {
     printf ("empty list");
   }
   while (temp->next != NULL)
   {
      temp = temp->next;
   }
   while(temp != NULL)
   {
      printf(" %d", temp->data);
      temp = temp->prev;
   }
}
```

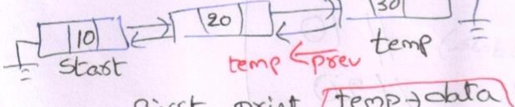
→ it executes if list is empty

→ temp reaches last node

print data from last node to start.



After first while loop



first print $\boxed{\begin{array}{l}temp \rightarrow data \\ temp = temp \rightarrow prev\end{array}}$ //30 //20

repeat until temp = NULL //10

Now it stops execution

```
//program to implement doubly linked list operations

#include<stdio.h>

#include<stdlib.h>

void insertatbegin();

void insertatend();

void insertatpos();

void display();

void displayreverse();

void deleteatbegin();

void deleteatend();

void deleteatpos();

int menu();

struct node

{

    int data;

    struct node *next;

    struct node *prev;

}*start=NULL,*newnode=NULL,*temp=NULL,*temp1=NULL;

int main()

{

  int ch;

  while(1)

  {

      ch=menu();

      switch(ch)
    {
       case 1: insertatbegin();
               break;
       case 2: display();break;
       case 3: exit(0);
       case 4: insertatend();break;
       case 5: insertatpos();break;
       case 6: deleteatbegin();break;
       case 7:deleteatend();break;
       case 8: deleteatpos();break;
       case 9: displayreverse();break;
       default : printf("enter valid choice");
    }
  }
}
int menu()
{

  int ch;
  printf("\n------------");
  printf("\n Doubly Linked List");
  printf("\n------------");

  printf("\n1.Insert at Begin\n2display\n3.Exit \n4.insert at end\n5.insertatpos\n");
  printf("\n6.deleteatbegin\n 7.deleteatend\n 8.deleteatpos");
```

```
    printf("\n\n-->Enter Your Choice:");
    scanf("%d",&ch);
    return ch;
}
void insertatbegin()
{
    newnode=(struct node*)malloc(sizeof(struct node));
    printf("\nEnter the Data in the Node:");
    scanf("%d",&newnode->data);
    newnode->next=NULL;
    newnode->prev=NULL;

    if(start==NULL)
    {
        //newnode->next==NULL;
        start=newnode;
    }
    else
    {
        newnode->next=start;
        start->prev = newnode;
        start=newnode;
    }

}
```

```
void insertatend()
{
    newnode=(struct node*)malloc(sizeof(struct node));
    printf("\nEnter the Data in the Node:");
    scanf("%d",&newnode->data);
    newnode->next=NULL;
    newnode->prev=NULL;
    if(start==NULL)
    {
        //newnode->prev=NULL;
        start=newnode;
    }
    else
    {

        temp=start;
        while(temp->next!=NULL)
        {
            temp=temp->next;
        }
        temp->next=newnode;
        newnode->prev=temp;
    }

}
```

```
void insertatpos()
{
   int pos,i;
    newnode=(struct node*)malloc(sizeof(struct node));
    printf("\nEnter the Data in the Node:");
    scanf("%d",&newnode->data);
    newnode->next=NULL;
    newnode->prev=NULL;
    printf("Enter the position : ");
    scanf("%d",&pos);
     if(start==NULL)
    {
       start=newnode;
    }
    else
    {
            temp = start;
             for(i=1;i<pos;i++)
             {
              temp = temp->next;
             }
            temp1=temp->prev;
            temp1->next=newnode;
            newnode->prev=temp1;
            newnode->next=temp;
            temp->prev=newnode;

        }

}



void deleteatbegin()
{
    if(start==NULL)
    {
        printf("list is empty deletion not possible");
    }
    else
    {
        temp=start;
        if(temp->prev==temp->next)
        {
            start=NULL;
            free(temp);
        }
        else
        {
            start=temp->next;
            start->prev=NULL;
            free(temp);

        }
    }
}
void deleteatend()
{

    if(start==NULL)
    {
        printf("list is empty deletion not possible");
    }
    else
    {

        temp=start;
        if(temp->prev==temp->next)
        {

            start=NULL;
            free(temp);
        }
        else
        {
            while(temp->next!=NULL)
            {
                temp=temp->next;

            }

           temp->prev->next=NULL;
           free(temp);
        }
    }
}
void deleteatpos()
{
    int pos,i,count;
    printf("\nEnter the Position of the element you would like to delete:");
    scanf("%d",&pos);
    temp=start;
    if(start==NULL)
    {
        printf("list is empty");
    }
    else
    {

        if(temp->prev==temp->next)
        {
```

```
    start=NULL;

    free(temp);

}

else

{

 for(i=1,count=1;i<pos;i++)

 {

  temp=temp->next;

  count++;

 }

 if(count==pos)

 {

    deleteatend();

 }

 else

 {

     temp1=temp->prev;

     temp1->next=temp->next;

     temp->next->prev=temp1;

     free(temp);

 }


}
```

```
    }


}
void display()
{


    temp=start;


    if(temp==NULL)
    {
        printf("empty list\n");
    }
    while(temp!=NULL)
    {
      printf("---->|%d|%d| ",temp->data,temp->next);
        temp=temp->next;
    }
}
void displayreverse()
{
    temp=start;
    if(temp==NULL)
    {
        printf("empty list\n");
    }
    while(temp->next!=NULL)
    {
        temp=temp->next;
    }
    while(temp!=NULL)
    {
        printf("\t%d",temp->data);
        temp=temp->prev;
    }
}
```

OUTPUT:

```


------------
 Doubly Linked List
------------
1.Insert at Begin
2display
3.Exit
4.insert at end
5.insertatpos


6.deleteatbegin
 7.deleteatend
 8.deleteatpos


-->Enter Your Choice:1


Enter the Data in the Node:10


------------
 Doubly Linked List
------------
1.Insert at Begin
2display
3.Exit
4.insert at end
5.insertatpos


6.deleteatbegin
 7.deleteatend
 8.deleteatpos


-->Enter Your Choice:1


Enter the Data in the Node:5


------------
 Doubly Linked List
```

```
------------
1.Insert at Begin
2display
3.Exit
4.insert at end
5.insertatpos

6.deleteatbegin
 7.deleteatend
 8.deleteatpos

-->Enter Your Choice:2
---->5 ---->10
------------
 Doubly Linked List
------------
1.Insert at Begin
2display
3.Exit
4.insert at end
5.insertatpos

6.deleteatbegin
 7.deleteatend
 8.deleteatpos
```

```
-->Enter Your Choice:4


Enter the Data in the Node:15



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

 Doubly Linked List

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

1.Insert at Begin

2display

3.Exit

4.insert at end

5.insertatpos


6.deleteatbegin

 7.deleteatend

 8.deleteatpos


-->Enter Your Choice:4


Enter the Data in the Node:20



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

 Doubly Linked List

------------
```

```
1.Insert at Begin

2display

3.Exit

4.insert at end

5.insertatpos


6.deleteatbegin

 7.deleteatend

 8.deleteatpos


-->Enter Your Choice:2

---->5 ---->10 ---->15 ---->20

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

 Doubly Linked List

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

1.Insert at Begin

2display

3.Exit

4.insert at end

5.insertatpos


6.deleteatbegin

 7.deleteatend

 8.deleteatpos
```

```
-->Enter Your Choice:5


Enter the Data in the Node:18

Enter the position : 3


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

 Doubly Linked List

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

1.Insert at Begin

2display

3.Exit

4.insert at end

5.insertatpos


6.deleteatbegin

 7.deleteatend

 8.deleteatpos


-->Enter Your Choice:2

---->5 ---->10 ---->18 ---->15 ---->20

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

 Doubly Linked List

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

1.Insert at Begin

2display
```

```
3.Exit

4.insert at end

5.insertatpos


6.deleteatbegin

 7.deleteatend

 8.deleteatpos


-->Enter Your Choice:8


Enter the Position of the element you would like to delete:4


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

 Doubly Linked List

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

1.Insert at Begin

2display

3.Exit

4.insert at end

5.insertatpos


6.deleteatbegin

 7.deleteatend

 8.deleteatpos
```

```
-->Enter Your Choice:2

---->5 ---->10 ---->18 ---->15

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

 Doubly Linked List

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

1.Insert at Begin

2display

3.Exit

4.insert at end

5.insertatpos


6.deleteatbegin

 7.deleteatend

 8.deleteatpos


-->Enter Your Choice:7


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

 Doubly Linked List

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

1.Insert at Begin

2display

3.Exit

4.insert at end

5.insertatpos
```

```
6.deleteatbegin

 7.deleteatend

 8.deleteatpos


-->Enter Your Choice:2

---->5 ---->10 ---->18

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

 Doubly Linked List

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

1.Insert at Begin

2display

3.Exit

4.insert at end

5.insertatpos


6.deleteatbegin

 7.deleteatend

 8.deleteatpos


-->Enter Your Choice:6


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

 Doubly Linked List

------------
```

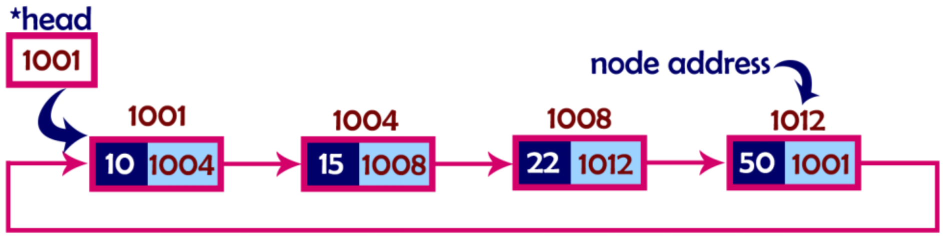
```
1.Insert at Begin

2display

3.Exit

4.insert at end

5.insertatpos


6.deleteatbegin

 7.deleteatend

 8.deleteatpos


-->Enter Your Choice:2

---->10 ---->18

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

 Doubly Linked List

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

1.Insert at Begin

2display

3.Exit

4.insert at end

5.insertatpos


6.deleteatbegin

 7.deleteatend

 8.deleteatpos
```

```
-->Enter Your Choice:3
```

Circular Linked List

- In single linked list, every node points to its next node in the sequence and the last node points NULL.
- But in circular linked list, every node points to its next node in the sequence but the last node points to the first node in the list.
- Circular linked list is a sequence of elements in which every element has link to its next element in the sequence and the last element has a link to the first element in the sequence.
- That means circular linked list is similar to the single linked list except that the last node points to the first node in the list.

Example



Operations:

In a circular linked list, we perform the following operations...

- Insertion
- Deletion
- Display

- INSERT
    - Insert at begin
    - Insert at end
    - Insert at position

- DELETE
    - Delete at begin
    - Delete at end
    - Delete at position
- DISPLAY
    - Display from starting node
    - Display from last node

Before we implement actual operations, first we need to setup empty list. First perform the following steps before implementing actual operations.

- Include all the header files which are used in the program.
- Declare all the user defined functions.
- Define a Node structure with two members data and next
- Define a Node pointer ‘start' and set it to NULL.
- Implement the main method by displaying operations menu and make suitable function calls in the main method to perform user selected operation.

Inserting At Beginning of the list:

We can use the following steps to insert a new node at beginning of the circular linked list...

- Create a newNode with given value.
- Check whether list is Empty (start == NULL)
- If it is Empty then, set start = newNode and newNode→next = start .
- If it is Not Empty then, define a Node pointer 'temp' and initialize with 'start'.
- Keep moving the 'temp' to its next node until it reaches to the last node (until 'temp → next == start').
- Set 'newNode → next =start', 'head = newNode' and 'temp → next = start'.

Inserting At End of the list:

- We can use the following steps to insert a new node at end of the circular linked list...
- Create a newNode with given value.
- Check whether list is Empty (start == NULL).
- If it is Empty then, set start = newNode and newNode → next = start.
- If it is Not Empty then, define a node pointer temp and initialize with head.
- Keep moving the temp to its next node until it reaches to the last node in the list (until temp → next == start).
- Set temp → next = newNode and newNode → next = start.

```
//circular linked list

#include<stdio.h>

#include<stdlib.h>



void display();

void insertAtBegin();
void insertAtEnd();
void insertAtPos();
void deleteAtBegin();
void deleteAtEnd();
void deleteAtPos();
int menu();
struct node
{
    int data;
    struct node *next;
}*start=NULL,*newnode=NULL,*temp=NULL,*temp1=NULL;
int main()
{
  int ch;
  while(1)
  {
      ch=menu();
      switch(ch)
      {
          case 1: display();break;
          case 2: insertAtBegin();break;
          case 3: insertAtEnd();break;
          case 4: insertAtPos();break;
          case 5: deleteAtBegin();break;
          case 6: deleteAtEnd();break;
          case 7: deleteAtPos();break;

          case 8: exit(0);

          default : printf("enter valid choice");

      }

  }

}

int menu()

{


    int ch;

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

    printf("\n CIRCULAR Single Linked List");

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


    printf("\n1display\n 2. insertAtBegin\n 3. insert at end\n4:insert at pos\n");

    printf("\n5:deleteFromFirst Position\n6:delete at end\n7:delete at pos\n8.exit\n");

    printf("\n\n-->Enter Your Choice:");

    scanf("%d",&ch);

    return ch;

}


void insertAtBegin()

{


    newnode=(struct node*)malloc(sizeof(struct node));
    printf("\nEnter Element:");
    scanf("%d",&newnode->data);
    newnode->next=newnode;
    if(start==NULL)
    {
      start=newnode;
   }
   else
   {
    temp = start;
    while(temp->next != start)
    {
        temp = temp->next;
    }
    newnode->next=start;
    temp->next=newnode;
    start = newnode;
   }
}

void insertAtEnd()
{

    newnode=(struct node*)malloc(sizeof(struct node));
    printf("\nEnter Element:");
    scanf("%d",&newnode->data);
    newnode->next=newnode;
    if(start==NULL)
    {
      start=newnode;
   }
   else
   {
     temp = start;
     while(temp->next != start)
     {
        temp = temp->next;
     }
     newnode->next=start;
     temp->next=newnode;
    }
}
void insertAtPos()
{
    int pos,i;
    newnode=(struct node*)malloc(sizeof(struct node));
    printf("\nEnter Element:");
    scanf("%d",&newnode->data);
    printf("\nenter the position");
    scanf("%d",&pos);
    newnode->next=newnode;
    temp=start;
      for(i=1;i<pos-1;i++)
      {
          temp=temp->next;

      }
        temp1=temp->next;
        temp->next=newnode;
        newnode->next=temp1;
}

void deleteAtBegin()
{
    temp=start;
    temp1=start;
    if(start==NULL)
    {
        printf("empty list");
    }
    else
    {
        if(start->next==start)
        {
            start=NULL;
            free(temp);
        }
        else
        {
            while(temp->next!=start)
            {
                temp=temp->next;
            }
            start=start->next;
            temp->next=start;
            free(temp1);
        }
    }
}
void deleteAtEnd()
{
    temp=start;

    if(start==NULL)
    {
        printf("empty list\n");

    }
    else
    {
        if(start->next==start)
        {
            start=NULL;
            free(temp);
        }
        else
        {
            while(temp->next!=start)
            {
                temp1=temp;
                temp=temp->next;
            }
            temp1->next=start;
            free(temp);
        }
    }
}
void deleteAtPos()
{
    int i,pos;
    printf("Enter the position");
    scanf("%d",&pos);
    temp=start;
    for(i=1;i<pos;i++)
    {
        temp1=temp;
        temp=temp->next;
    }
    temp1->next=temp->next;
    free(temp);
}
void display()
{
    temp =start;
    if(start==NULL)
    {
        printf("\n empty list\n");
    }
    else
    {
        while(temp->next!=start)
        {
            printf("  %d-",temp->data);
            temp=temp->next;
        }
        printf(" %d\n",temp->data);

    }
}
```

```
OUTPUT:


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

Single Linked List

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

1display

 2. insertAtBegin

 3. insert at end

4:insert at pos


5:deleteFromFirst Position

6:delete at end

7:delete at pos

8.exit




-->Enter Your Choice:2


Enter Element:22


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

Single Linked List

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

1display

 2. insertAtBegin
 3. insert at end
4:insert at pos

5:deleteFromFirst Position
6:delete at end
7:delete at pos
8.exit



-->Enter Your Choice:2

Enter Element:33

------------
Single Linked List
------------
1display
 2. insertAtBegin
 3. insert at end
4:insert at pos

5:deleteFromFirst Position
6:delete at end
7:delete at pos
8.exit
```

```
-->Enter Your Choice:2


Enter Element:44


------------
Single Linked List
------------
1display
 2. insertAtBegin
 3. insert at end
4:insert at pos


5:deleteFromFirst Position
6:delete at end
7:delete at pos
8.exit



-->Enter Your Choice:2


Enter Element:55


------------
```

```
Single Linked List

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

1display

 2. insertAtBegin

 3. insert at end

4:insert at pos


5:deleteFromFirst Position

6:delete at end

7:delete at pos

8.exit



-->Enter Your Choice:2


Enter Element:66


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

Single Linked List

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

1display

 2. insertAtBegin

 3. insert at end

4:insert at pos
```

```
5:deleteFromFirst Position

6:delete at end

7:delete at pos

8.exit



-->Enter Your Choice:1

  66-  55-  44-  33- 22


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

Single Linked List

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

1display

 2. insertAtBegin

 3. insert at end

4:insert at pos


5:deleteFromFirst Position

6:delete at end

7:delete at pos

8.exit



-->Enter Your Choice:7

Enter the position3
```

```
------------

Single Linked List

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

1display

 2. insertAtBegin

 3. insert at end

4:insert at pos


5:deleteFromFirst Position

6:delete at end

7:delete at pos

8.exit



-->Enter Your Choice:1

  66-  55-  33- 22


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

Single Linked List

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

1display

 2. insertAtBegin

 3. insert at end

4:insert at pos
```

```
5:deleteFromFirst Position
6:delete at end
7:delete at pos
8.exit


-->Enter Your Choice:10
enter valid choice
------------
Single Linked List
------------
1display
 2. insertAtBegin
 3. insert at end
4:insert at pos

5:deleteFromFirst Position
6:delete at end
7:delete at pos
8.exit


-->Enter Your Choice:
```

Sparse Matrix and its representations Using Arrays and Linked Lists

A matrix is a two-dimensional data object made of m rows and n columns, therefore having total m x n values. If most of the elements of the matrix have **0 value**, then it is called a sparse matrix.

Why to use Sparse Matrix instead of simple matrix ?

- **Storage:** There are lesser non-zero elements than zeros and thus lesser memory can be used to store only those elements.
- **Computing time:** Computing time can be saved by logically designing a data structure traversing only non-zero elements..

Example:

0 0 3 0 4

0 0 5 7 0

0 0 0 0 0

0 2 6 0 0

Representing a sparse matrix by a 2D array leads to wastage of lots of memory as zeroes in the matrix are of no use in most of the cases. So, instead of storing zeroes with non-zero elements, we only store non-zero elements. This means storing non-zero elements with **triples- (Row, Column, value).**

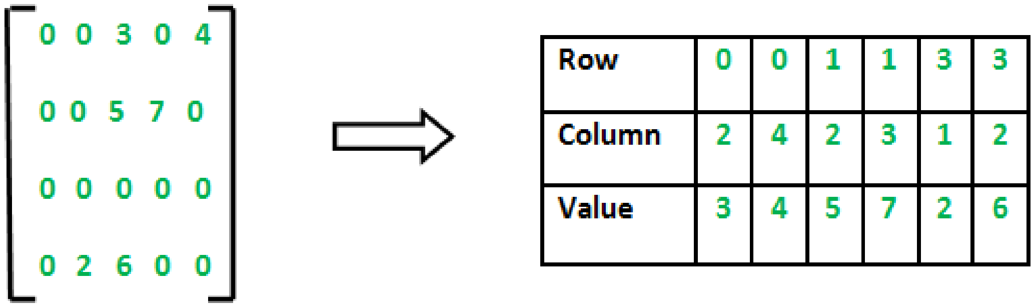
Sparse Matrix Representations can be done in many ways following are two common representations:

- Array representation
- Linked list representation

Method 1: Using Arrays

2D array is used to represent a sparse matrix in which there are three rows named as

- **Row:** Index of row, where non-zero element is located
- **Column:** Index of column, where non-zero element is located
- **Value:** Value of the non zero element located at index – (row,column)



$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{bmatrix} \Rightarrow$$

Row	0	0	1	1	3	3
Column	2	4	2	3	1	2
Value	3	4	5	7	2	6

```
// C++ program for Sparse Matrix Representation
// using Array
#include<stdio.h>

int main()
{
    // Assume 4x5 sparse matrix
    int sparseMatrix[4][5] =
    {
        {0 , 0 , 3 , 0 , 4 },
        {0 , 0 , 5 , 7 , 0 },
        {0 , 0 , 0 , 0 , 0 },
        {0 , 2 , 6 , 0 , 0 }
    };

    int size = 0;
    for (int i = 0; i < 4; i++)
        for (int j = 0; j < 5; j++)
            if (sparseMatrix[i][j] != 0)
                size++;

    // number of columns in compactMatrix (size) must be
    // equal to number of non - zero elements in
    // sparseMatrix
    int compactMatrix[3][size];
```

```
   // Making of new matrix

   int k = 0;

   for (int i = 0; i < 4; i++)

      for (int j = 0; j < 5; j++)

         if (sparseMatrix[i][j] != 0)

         {

            compactMatrix[0][k] = i;

            compactMatrix[1][k] = j;

            compactMatrix[2][k] = sparseMatrix[i][j];

            k++;

         }


   for (int i=0; i<3; i++)

   {

      for (int j=0; j<size; j++)

         printf("%d ", compactMatrix[i][j]);


      printf("\n");

   }

   return 0;

}
```

Output:

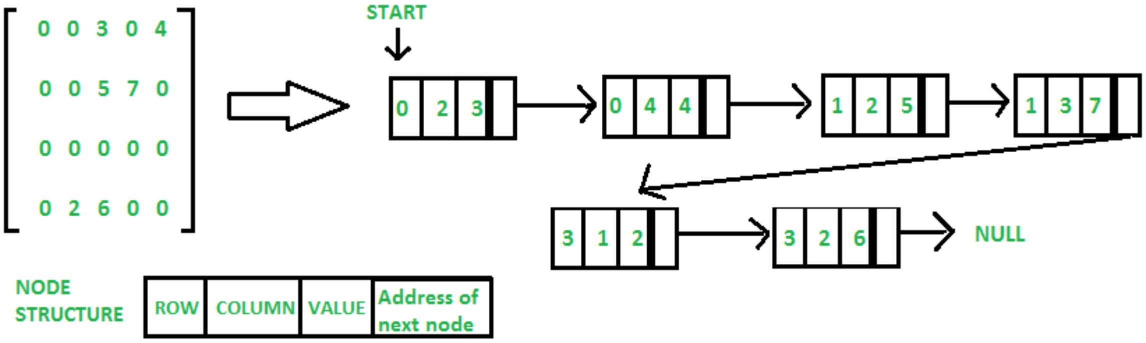
```
0 0 1 1 3 3
```

```
2 4 2 3 1 2

3 4 5 7 2 6
```

Method 2: Using Linked Lists

In linked list, each node has four fields. These four fields are defined as:

- **Row:** Index of row, where non-zero element is located
- **Column:** Index of column, where non-zero element is located
- **Value:** Value of the non zero element located at index – (row,column)
- **Next node:** Address of the next node



```
//C program for Sparse Matrix Representation

// using Linked Lists

#include<stdio.h>

#include<stdlib.h>


// Node to represent sparse matrix

struct Node

{
    int value;
    int row_position;
    int column_postion;
    struct Node *next;
};

// Function to create new node
void create_new_node(struct Node** start, int non_zero_element,
                     int row_index, int column_index )
{
    struct Node *temp, *r;
    temp = *start;
    if (temp == NULL)
    {
        // Create new node dynamically
        temp = (struct Node *) malloc (sizeof(struct Node));
        temp->value = non_zero_element;
        temp->row_position = row_index;
        temp->column_postion = column_index;
        temp->next = NULL;
        *start = temp;

    }
    else
    {
        while (temp->next != NULL)
            temp = temp->next;

        // Create new node dynamically
        r = (struct Node *) malloc (sizeof(struct Node));
        r->value = non_zero_element;
        r->row_position = row_index;
        r->column_postion = column_index;
        r->next = NULL;
        temp->next = r;

    }
}

// This function prints contents of linked list
// starting from start
void PrintList(struct Node* start)
{
    struct Node *temp, *r, *s;
    temp = r = s = start;
   printf("row_position: "); while(temp != NULL)
    {

        printf("%d ", temp->row_position);
        temp = temp->next;
    }
    printf("\n");

    printf("column_postion: ");
    while(r != NULL)
    {
        printf("%d ", r->column_postion);
        r = r->next;
    }
    printf("\n");
    printf("Value: ");
    while(s != NULL)
    {
        printf("%d ", s->value);
        s = s->next;
    }
    printf("\n");
}

// Driver of the program
int main()
{
   // Assume 4x5 sparse matrix
    int sparseMatric[4][5] =
    {
        {0 , 0 , 3 , 0 , 4 },
        {0 , 0 , 5 , 7 , 0 },
        {0 , 0 , 0 , 0 , 0 },
        {0 , 2 , 6 , 0 , 0 }
    };

    /* Start with the empty list */
    struct Node* start = NULL;

    for (int i = 0; i < 4; i++)
        for (int j = 0; j < 5; j++)

            // Pass only those values which are non - zero
            if (sparseMatric[i][j] != 0)
                create_new_node(&start, sparseMatric[i][j], i, j);

    PrintList(start);

    return 0;
}
```

Output:

```
Row position :0 0 1 1 3 3
Column position: 2 4 2 3 1 2
Value: 3 4 5 7 2 6
```

MODULE-III: Stacks and Queues [10 Periods]

A: Stacks: Basic stack operations, Representation of a stack using arrays and linked lists, Stack Applications - Reversing list, factorial calculation, postfix expression evaluation, infix-to-postfix conversion.

B:Queues: Basic queue operations, Representation of a queue using array and Linked list, Classification and implementation – Circular, Enqueue and Dequeue, Applications of Queues.

Stacks:

- A stack is a linear Data Structure in which a data item is inserted and deleted at one end.

- A stack is called LIFO (Last in First Out) or FILO (First in Last Out) structure because the data item that is inserted last into the stack is the first data item to be deleted from the stack.
- Stack uses the pointer called TOP – to organize elements.
- Basic operations on stack are:
  - PUSH()- to push/insert an element into the stack.
  - POP()- to delete an element from the stack.
  - DISPLAY()- to display the elements of the stack
- We can implement stack operations in two ways
  - Using arrays (static)
  - Using linked list (dynamic)
- 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 with example:

- Let us consider a stack with 5 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 stack. If such is the case, we will reach a stack **underflow** condition.

- When an element is added to a stack, the operation is performed by push().

When an element is taken off from the stack, the operation is performed by pop().

Implementation of stack using arrays:

Before implementing actual operations, first follow the below steps to create an empty stack.

- Include all the header files which are used in the program and define a constant 'SIZE' with specific value.
- Declare all the functions used in stack implementation.
- Create a one-dimensional array with fixed size (int stack[SIZE])
- Define a integer variable 'top' and initialize with '-1'. (int top = -1)

Inserting values into the stack: PUSH()

- In a stack, push() is a function used to insert an element into the stack.
- In a stack, the new element is always inserted at top position.
- Push function takes one integer value as parameter and inserts that value into the stack.
- We can use the following steps to push an element on to the stack.
    - Check whether stack is FULL. (top == SIZE-1)
    - If it is FULL, then display "Stack is FULL!!! Insertion is not possible!!! Overflow" and terminate the function.
    - If it is NOT FULL, then increment top value by one (top++) and set stack[top] to value (stack[top] = value).

```
void push(int value)
{
  if(top == SIZE-1)
    printf("\nStack is Full!!! OVERFLOW!!!");
  else
```

```
   {
    top++;
    stack[top] = value;
    printf("\nInsertion success!!!");
  }
}
```

Delete a value from the Stack- POP()

- In a stack, pop() is a function used to delete an element from the stack.
- In a stack, the element is always deleted from top position.
- Pop function does not take any value as parameter.
- We can use the following steps to pop an element from the stack.
  - Check whether stack is EMPTY. (top == -1)
  - If it is EMPTY, then display "Stack is EMPTY!!! Deletion is not possible!!!" and terminate the function.
  - If it is NOT EMPTY, then delete stack[top] and decrement top value by one (top--).

```
void pop()
{
  if(top == -1)
    printf("\nStack is Empty!!! UNDERFLOW!!!");
  else
  {
    printf("\nDeleted : %d", stack[top]);

    top--;

  }

}
```

display() - Displays the elements of a Stack

We can use the following steps to display the elements of a stack...

- Check whether stack is EMPTY. (top == -1)
- If it is EMPTY, then display "Stack is EMPTY!!!" and terminate the function.
- If it is NOT EMPTY, then define a variable 'i' and initialize with top.
  Display stack[i] value and decrement i value by one (i--).
- Repeat above step until i value becomes '0'.

```
void display()

{

  if(top == -1)

    printf("\nStack is Empty!!!");

  else

   {
    int i;

    printf("\nStack elements are:\n");

    for(i=top; i>=0; i--)

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

  }

}
```

```
// c program to implement static stack

#include<stdio.h>

#include<stdlib.h>

#define SIZE 10

void push(int);

void pop();

void display();

int stack[SIZE], top = -1;

void main()

{

  int value, choice;

  while(1)

  {

    printf("\n\n***** MENU *****\n");

    printf("1. Push\n2. Pop\n3. Display\n4. Exit");

    printf("\nEnter your choice: ");
    scanf("%d",&choice);
    switch(choice)
    {
     case 1: printf("Enter the value to be insert: ");
          scanf("%d",&value);
          push(value);
          break;
     case 2: pop();
          break;
     case 3: display();
          break;
     case 4: exit(0);
     default: printf("\nWrong selection!!! Try again!!!");
    }
  }
}



void push(int value)
{
  if(top == SIZE-1)
    printf("\nStack is Full!!! OVERFLOW!!!");
  else
   {
    top++;

    stack[top] = value;

    printf("\nInsertion success!!!");

  }

}

void pop()

{

  if(top == -1)

    printf("\nStack is Empty!!! UNDERFLOW!!!");

  else

  {

    printf("\nDeleted : %d", stack[top]);

    top--;

  }

}

void display()

{

  if(top == -1)

    printf("\nStack is Empty!!!");

  else

   {

    int i;

    printf("\nStack elements are:\n");

    for(i=top; i>=0; i--)

         printf("%d\n",stack[i]);
   }

}
```

OUTPUT:

```
***** MENU *****

1. Push

2. Pop

3. Display

4. Exit

Enter your choice: 3

Stack is Empty!!!

***** MENU *****

1. Push

2. Pop

3. Display

4. Exit

Enter your choice: 1

Enter the value to be insert: 10


Insertion success!!!


***** MENU *****

1. Push
```

```
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 20

Insertion success!!!

***** MENU *****
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 30

Insertion success!!!

***** MENU *****
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 3
```

```
Stack elements are:

30

20

10



***** MENU *****

1. Push

2. Pop

3. Display

4. Exit

Enter your choice: 2


Deleted : 30


***** MENU *****

1. Push

2. Pop

3. Display

4. Exit

Enter your choice: 3


Stack elements are:

20

10
```

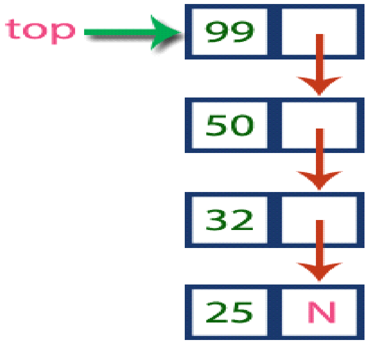
```
***** MENU *****
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 4


Process returned 0 (0x0)   execution time : 57.418 s
Press any key to continue.
```

Implementation of stack using linked list (dynamic stack)

- We can represent a stack as a 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.
- We can perform similar operations at one end of list using top pointer.



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.

OPERATIONS PUSH(),POP(),DISPLAY()

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

- Include all the header files which are used in the program. And declare all the user defined functions.
- Define a 'node' structure with two members data and next.
- Define a node pointer 'top' and set it to NULL.

push(value) - Inserting an element into the Stack

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

- Create a newNode with given value.
- Check whether stack is Empty (top == NULL)
- If it is Empty, then set newNode → next = NULL.
- If it is Not Empty, then set newNode → next = top.
- Finally, set top = newNode.

```
void push(int value)

{

  struct Node *newNode;

  newNode = (struct Node*)malloc(sizeof(struct Node));

  newNode->data = value;

  if(top == NULL)

  {

    newNode->next = NULL;

  }
  else

  {

    newNode->next = top;

  }


  top = newNode;

  printf("\nInsertion is Success!!!\n");

}
```

pop() - Deleting an Element from a Stack

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

- Check whether stack is Empty (top == NULL).
- If it is Empty, then display "Stack is Empty!!! Deletion is not possible!!!" and terminate the function
- If it is Not Empty, then define a Node pointer 'temp' and set it to 'top'.

- Then set 'top = top → next'.
- Finally, delete 'temp' (free(temp)).

```
void pop()
{
  if(top == NULL)
    printf("\nStack is Empty!!!\n");
  else
   {
    //struct Node *temp = top;
    temp=top;
    printf("\nDeleted element: %d", temp->data);
    top = temp->next;
    free(temp);
  }
}
```

display() - Displaying stack of elements

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

- Check whether stack is Empty (top == NULL).
- If it is Empty, then display 'Stack is Empty!!!'
- If it is Not Empty, then define a Node pointer 'temp' and initialize with top.
- Display 'temp → data --->' and move it to the next node. Repeat the same until temp reaches to the first node in the stack

  (temp → next != NULL).
- Finally! Display 'temp → data ---> NULL'

```
void display()
{
  if(top == NULL)
    printf("\nStack is Empty!!!\n");
  else
  {
    //struct Node *temp = top;
    temp=top;
    while(temp->next != NULL)
    {
      printf("%d--->",temp->data);
```

```
            temp = temp -> next;

          }

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

       }

    }
```

```
//stack using linked list

#include<stdio.h>

#include<stdlib.h>

struct Node

{

   int data;

   struct Node *next;

}*top = NULL,*temp=NULL;

void push(int);

void pop();

void display();
void main()

{

  int choice, value;

  printf("\n:: Stack using Linked List ::\n");

  while(1)

  {

    printf("\n****** MENU ******\n");

    printf("1. Push\n2. Pop\n3. Display\n4. Exit\n");

    printf("Enter your choice: ");

    scanf("%d",&choice);

    switch(choice)

    {

      case 1: printf("Enter the value to be insert: ");

              scanf("%d", &value);

              push(value);

              break;

      case 2: pop(); break;

      case 3: display(); break;

      case 4: exit(0);

      default: printf("\nWrong selection!!! Please try again!!!\n");

    }

  }

}

void push(int value)

{
  struct Node *newNode;
  newNode = (struct Node*)malloc(sizeof(struct Node));
  newNode->data = value;
  if(top == NULL)
  {
    newNode->next = NULL;
  }
  else
  {
    newNode->next = top;
  }
  top = newNode;
  printf("\nInsertion is Success!!!\n");
}
void pop()
{
  if(top == NULL)
    printf("\nStack is Empty!!!\n");
  else
   {
    //struct Node *temp = top;
    temp=top;
    printf("\nDeleted element: %d", temp->data);
    top = temp->next;
    free(temp);
  }
}
void display()
{
  if(top == NULL)
    printf("\nStack is Empty!!!\n");
  else
  {
    //struct Node *temp = top;
    temp=top;
    while(temp->next != NULL)
    {
      printf("%d--->",temp->data);
      temp = temp -> next;
    }
    printf("%d--->NULL",temp->data);
  }
}
```

OUTPUT:

```

:: Stack using Linked List ::


****** MENU ******
```

```
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 3

Stack is Empty!!!

****** MENU ******
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 12

Insertion is Success!!!

****** MENU ******
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 22
```

```
Insertion is Success!!!

****** MENU ******
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 3
22--->12--->NULL
****** MENU ******
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 2

Deleted element: 22
****** MENU ******
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 4
```

```
Process returned 0 (0x0)   execution time : 75.569 s

Press any key to continue.
```

Applications of STACK:

Application of Stack :

- Recursive Function.
- Expression Evaluation.
- Expression Conversion.
  - Infix to postfix
  - Infix to prefix
  - Postfix to infix
  - Postfix to prefix
  - Prefix to infix
  - Prefix to postfix
- Towers of hanoi.

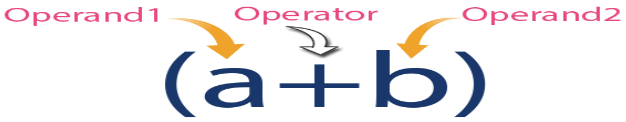
Expressions:

- An expression is a collection of operators and operands that represents a specific value.
- Operator is a symbol which performs a particular task like arithmetic operation or logical operation or conditional operation etc.,
- Operands are the values on which the operators can perform the task. Here operand can be a direct value or variable or address of memory location

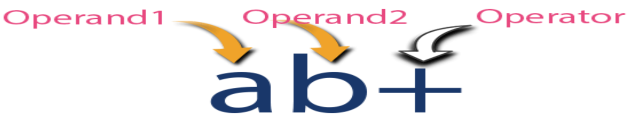
Expression types:

Based on the operator position, expressions are divided into THREE types. They are as follows.

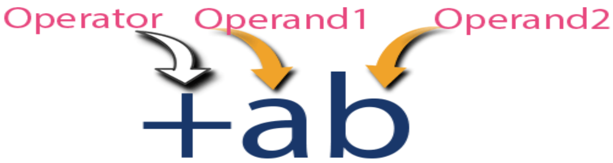
- **Infix Expression**
  - In infix expression, operator is used in between operands.
  - Syntax : operand1 operator operand2
  - Example



- **Postfix Expression**
  - In postfix expression, operator is used after operands. We can say that "Operator follows the Operands".
  - Syntax : operand1 operand2 operator
  - Example:



- **Prefix Expression**
  - In prefix expression, operator is used before operands. We can say that "Operands follows the Operator".
  - Syntax : operator operand1 operand2
  - Example:



Infix to postfix conversion using stack:

- Procedure to convert from infix expression to postfix expression is as follows:
- Scan the infix expression from left to right.
- If the scanned symbol is left parenthesis, push it onto the stack.
- If the scanned symbol is an operand, then place directly in the postfix expression (output).
- If the symbol scanned is a right parenthesis, then go on popping all the items from the stack and place them in the postfix expression till we get the matching left parenthesis.
- If the scanned symbol is an operator, then go on removing all the operators from the stack and place them in the postfix expression, if and only if the precedence of the operator which is on the top of the stack is greater than (or greater than or equal) to the precedence of the scanned operator and push the scanned operator onto the stack otherwise, push the scanned operator onto the stack.

Example-1

Reading Character	STACK	Postfix Expression
Initially	Stack is EMPTY	EMPTY
(	Push '(' top (	EMPTY
A	No operation Since 'A' is OPERAND top (	A
+	'+' has low priority than '(' so, PUSH '+' top + (	A
B	No operation Since 'B' is OPERAND top + (	A B
)	POP all elements till we reach '(' POP '+' POP '(' top	A B +
*	Stack is EMPTY & '*' is Operator PUSH '*' top *	A B +
(	PUSH '(' top (*	A B +
C	No operation Since 'C' is OPERAND top (*	A B + C
-	'-' has low priority than '(' so, PUSH '-' top - (*	A B + C
D	No operation Since 'D' is OPERAND top - (*	A B + C D
)	POP all elements till we reach '(' POP '-' POP '(' top *	A B + C D -
$	POP all elements till Stack becomes Empty	A B + C D - *

Example2:

Convert ((A – (B + C)) * D) ↑ (E + F) infix expression to postfix form:

SYMBOL	POSTFIX STRING	STACK	REMARKS
(		(	
(		((	
A	A	((	
-	A	((-	
(	A	((- (	
B	A B	((- (	
+	A B	((- (+	
C	A B C	((- (+	
)	A B C +	((-	
)	A B C + -	(	
*	A B C + -	(*	
D	A B C + - D	(*	
)	A B C + - D *		
↑	A B C + - D *	↑	
(	A B C + - D *	↑ (	
E	A B C + - D * E	↑ (	
+	A B C + - D * E	↑ (+	
F	A B C + - D * E F	↑ (+	

)	A B C + - D * E F +	↑	
End of string	A B C + - D * E F + ↑	The input is now empty. Pop the output symbols from the stack until it is empty.	

Example3

Convert a + b * c + (d * e + f) * g the infix expression into postfix form.

SYMBOL	POSTFIX STRING	STACK	REMARKS
a	a		
+	a	+	
b	a b	+	
*	a b	+ *	
c	a b c	+ *	
+	a b c * +	+	
(	a b c * +	+ (	
d	a b c * + d	+ (	

*	a b c * + d	+ (*	
e	a b c * + d e	+ (*	
+	a b c * + d e *	+ (+	
f	a b c * + d e * f	+ (+	
)	a b c * + d e * f +	+	
*	a b c * + d e * f +	+ *	
g	a b c * + d e * f + g	+ *	
End of string	a b c * + d e * f + g * +	The input is now empty. Pop the output symbols from the stack until it is empty.	

Example 3:

Convert the following infix expression A + B * C – D / E * H into its equivalent postfix expression.

SYMBOL	POSTFIX STRING	STACK	REMARKS
A	A		
+	A	+	
B	A B	+	
*	A B	+ *	
C	A B C	+ *	
-	A B C * +	-	
D	A B C * + D	-	
/	A B C * + D	- /	
E	A B C * + D E	- /	
*	A B C * + D E /	- *	
H	A B C * + D E / H	- *	
End of string	A B C * + D E / H * -	The input is now empty. Pop the output symbols from the stack until it is empty.	

Example 4:

Convert the following infix expression A+(B *C–(D/E↑F)*G)*H into its equivalent postfix expression.

SYMBOL	POSTFIX STRING	STACK	REMARKS
A	A		
+	A	+	
(	A	+ (	
B	A B	+ (	
*	A B	+ (*	
C	A B C	+ (*	
-	A B C *	+ (-	
(	A B C *	+ (- (	
D	A B C * D	+ (- (	
/	A B C * D	+ (- (/	
E	A B C * D E	+ (- (/	
↑	A B C * D E	+ (- (/ ↑	
F	A B C * D E F	+ (- (/ ↑	

)	A B C * D E F ↑ /	+ (-	
*	A B C * D E F ↑ /	+ (- *	
G	A B C * D E F ↑ / G	+ (- *	
)	A B C * D E F ↑ / G * -	+	
*	A B C * D E F ↑ / G * -	+ *	
H	A B C * D E F ↑ / G * - H	+ *	
End of string	A B C * D E F ↑ / G * - H * +	The input is now empty. Pop the output symbols from the stack until it is empty.	

```
//Program to convert infix expression to postfix by using stacks



#include<stdio.h>

#include<stdlib.h>

#include<math.h>

#include<string.h>
```

```
void expression_conversion();
int check_space_tabs(char character);
void push(int character);
int pop();
int priority(char character);
int isEmpty();

int top;
int stack[50];
char infix_expression[50], postfix_expression[50];

int main()
{
        top = -1;
        printf("\nEnter an Expression in Infix format:\t");
        scanf("%s", infix_expression);
        expression_conversion();
        printf("\nExpression in Postfix Format:\t%s\n",postfix_expression);
        return 0;
}

void expression_conversion()
{
        int count, temp = 0;
        char next;
```

```
    char character;
    for(count = 0; count < strlen(infix_expression); count++)
    {
        character = infix_expression[count];
        if(!check_space_tabs(character))
        {
            switch(character)
            {
                case '(': push(character);
                    break;
                case ')':
                    while((next = pop()) != '(')
                    {
                        postfix_expression[temp++] = next;
                    }
                    break;
                case '+':
                case '-':
                case '*':
                case '/':
                case '%':
                case '^':
                    while(!isEmpty()  &&  priority(stack[top])   >= priority(character))
                    postfix_expression[temp++] = pop();
```

```
                   push(character);
                     break;
                default:
                     postfix_expression[temp++] = character;
            }
        }
    }
    while(!isEmpty())
    {
        postfix_expression[temp++] = pop();
    }
    postfix_expression[temp] = '\0';
}

int priority(char character)
{
    switch(character)
    {
        case '(': return 0;
        case '+':
        case '-':
             return 1;
        case '*':
        case '/':
        case '%':
            return 2;
        case '^':
            return 3;
        default:
            return 0;
    }
}

void push(int character)
{
    if(top > 50)
    {
        printf("Stack Overflow\n");
        exit(1);
    }
    top = top + 1;
    stack[top] = character;
}

int check_space_tabs(char character)
{
    if(character == ' ' || character == '\t')
    {
        return 1;
    }
    else
    {
        return 0;
    }
}

int pop()
{
    if(isEmpty())
    {
        printf("Stack is Empty\n");
        exit(1);
    }
    return(stack[top--]);
}

int isEmpty()
{
    if(top == -1)
    {
        return 1;
    }
    else
    {
        return 0;
```

```
            }
}
```

OUTPUT:

```
Enter an Expression in Infix format:   A+(B*C-(D/E^F)*G)*H

Expression in Postfix Format:   ABC*DEF^/G*-H*+
```

Evaluation of postfix expression:

- The postfix expression is evaluated easily by the use of a stack.
- When a number is seen, it is pushed onto the stack;
- when an operator is seen, the operator is applied to the two numbers that are popped from the stack and the result is pushed onto the stack.
- When an expression is given in postfix notation, there is no need to know any precedence rules.

Example 1:

Evaluate the postfix expression: 6 5 2 3 + 8 * + 3 + *

SYMBOL	OPERAND 1	OPERAND 2	VALUE	STACK	REMARKS
6				6	
5				6, 5	
2				6, 5, 2	
3				6, 5, 2, 3	The first four symbols are placed on the stack.
+	2	3	5	6, 5, 5	Next a '+' is read, so 3 and 2 are popped from the stack and their sum 5, is pushed

8	2	3	5	6, 5, 5, 8	Next 8 is pushed
*	5	8	40	6, 5, 40	Now a ‘*’ is seen, so 8 and 5 are popped as 8 * 5 = 40 is pushed
+	5	40	45	6, 45	Next, a ‘+’ is seen, so 40 and 5 are popped and 40 + 5 = 45 is pushed
3	5	40	45	6, 45, 3	Now, 3 is pushed
+	45	3	48	6, 48	Next, ‘+’ pops 3 and 45 and pushes 45 + 3 = 48 is pushed
*	6	48	288	288	Finally, a ‘*’ is seen and 48 and 6 are popped, the result 6 * 48 = 288 is pushed

Example2

Infix Expression (5 + 3) * (8 - 2)

Postfix Expression 5 3 + 8 2 - *

Above Postfix Expression can be evaluated by using Stack Data Structure as follows...

Reading Symbol	Stack Operations		Evaluated Part of Expression
Initially	Stack is Empty		Nothing
5	push(5)	5	Nothing
3	push(3)	3 5	Nothing

+	value1 = pop()
	value2 = pop()
	result = value2 + value1
	push(result)
		8	value1 = pop(); // 3
			value2 = pop(); // 5
			result = 5 + 3; // 8
			Push(8)
			(5 + 3)

8	push(8)	8 8	(5 + 3)
2	push(2)	2 8 8	(5 + 3)

-	value1 = pop()
	value2 = pop()
	result = value2 - value1
	push(result)
		6
		8
			value1 = pop(); // 2
			value2 = pop(); // 8
			result = 8 - 2; // 6
			Push(6)
			(8 - 2)
			(5 + 3) , (8 - 2)

*	value1 = pop()
	value2 = pop()
	result = value2 * value1
	push(result)
		48	value1 = pop(); // 6
			value2 = pop(); // 8
			result = 8 * 6; // 48
			Push(48)
			(6 * 8)
			(5 + 3) * (8 - 2)

\$ End of Expression	result = pop()		Display (result) 48 As final result

Infix Expression (5 + 3) * (8 - 2) = 48

Postfix Expression 5 3 + 8 2 - * value is 48

Example 3:

Evaluate the following postfix expression: 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

SYMBOL	OPERAND 1	OPERAND 2	VALUE	STACK
6				6
2				6, 2
3				6, 2, 3
+	2	3	5	6, 5
-	6	5	1	1
3	6	5	1	1, 3
8	6	5	1	1, 3, 8
2	6	5	1	1, 3, 8, 2
/	8	2	4	1, 3, 4
+	3	4	7	1, 7
*	1	7	7	7
2	1	7	7	7, 2
↑	7	2	49	49
3	7	2	49	49, 3
+	49	3	52	52

```
//program to evaluate the given postfix expression


#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<string.h>
void push(int character);
int postfix_evaluation();
int pop();
int isEmpty();


int top;
int stack[50];
char postfix_expression[50];


int main()
{
        int evaluated_value;
        top = -1;
        printf("\nEnter an Expression in Postfix format:\t");
        scanf("%s", postfix_expression);
        printf("\nExpression in Postfix Format: \t%s\n", postfix_expression);
        evaluated_value = postfix_evaluation();
        printf("\nEvaluation of Postfix Expression: \t%d\n", evaluated_value);
        return 0;
}

int postfix_evaluation()
{
        int x, y, temp, value;
        int count;
        for(count = 0; count < strlen(postfix_expression); count++)
        {
                if(postfix_expression[count] <= '9' && postfix_expression[count] >= '0')
                {
                        push(postfix_expression[count] - '0');
                }
                else
                {
                        x = pop();
                        y = pop();
                        switch(postfix_expression[count])
```

```
            {
                case '+': temp = y + x;
                          break;
                case '-': temp = y - x;
                          break;
                case '*': temp = y * x;
                          break;
                case '/': temp = y / x;
                          break;
                case '%': temp = y % x;
                          break;
                case '^': temp = pow(y, x);
                          break;
                default:  printf("Invalid");
            }
            push(temp);
        }
    }
    value = pop();
    return value;
}

void push(int character)
{
    if(top > 50)
    {
        printf("Stack Overflow\n");
        exit(1);
    }
    top = top + 1;
    stack[top] = character;
}

int pop()
{
    if(isEmpty())
    {
        printf("Stack is Empty\n");
        exit(1);
    }
    return(stack[top--]);
}

int isEmpty()
{
    if(top == -1)
    {
        return 1;
    }
    else
```

```
          {
                  return 0;
          }
}
```

OUTPUT:

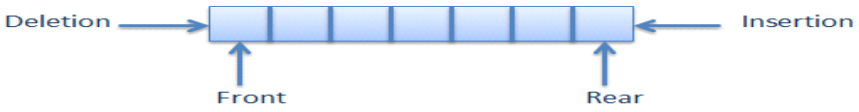
```
Enter an Expression in Postfix format:  6523+8*+3+*

Expression in Postfix Format:   6523+8*+3+*

Evaluation of Postfix Expression:       288
```

QUEUE

#	STACK	QUEUE
1	Objects are inserted and removed at the same end.	Objects are inserted and removed from different ends.
2	In stacks only one pointer is used. It points to the top of the stack.	In queues, two different pointers are used for front and rear ends.
3	In stacks, the last inserted object is first to come out.	In queues, the object inserted first is first deleted.
4	Stacks follow Last In First Out (LIFO) order.	Queues following First In First Out (FIFO) order.
5	Stack operations are called push and pop.	Queue operations are called enqueue and dequeue.
6	Stacks are visualized as vertical collections.	Queues are visualized as horizontal collections.
7	Collection of dinner plates at a wedding reception is an example of stack.	People standing in a line to board a bus is an example of queue.

- Queue is a linear data structure in which the insertion and deletion operations are performed at two different ends.

- In a queue data structure, adding and removing of elements are performed at two different positions.
- The insertion is performed at one end and deletion is performed at other end.
- In a queue data structure, the insertion operation is performed at a position which is known as '**rear**' and the deletion operation is performed at a position which is known as '**front'.**
- In queue data structure, the insertion and deletion operations are performed based on **FIFO (First In First Out) principle.**

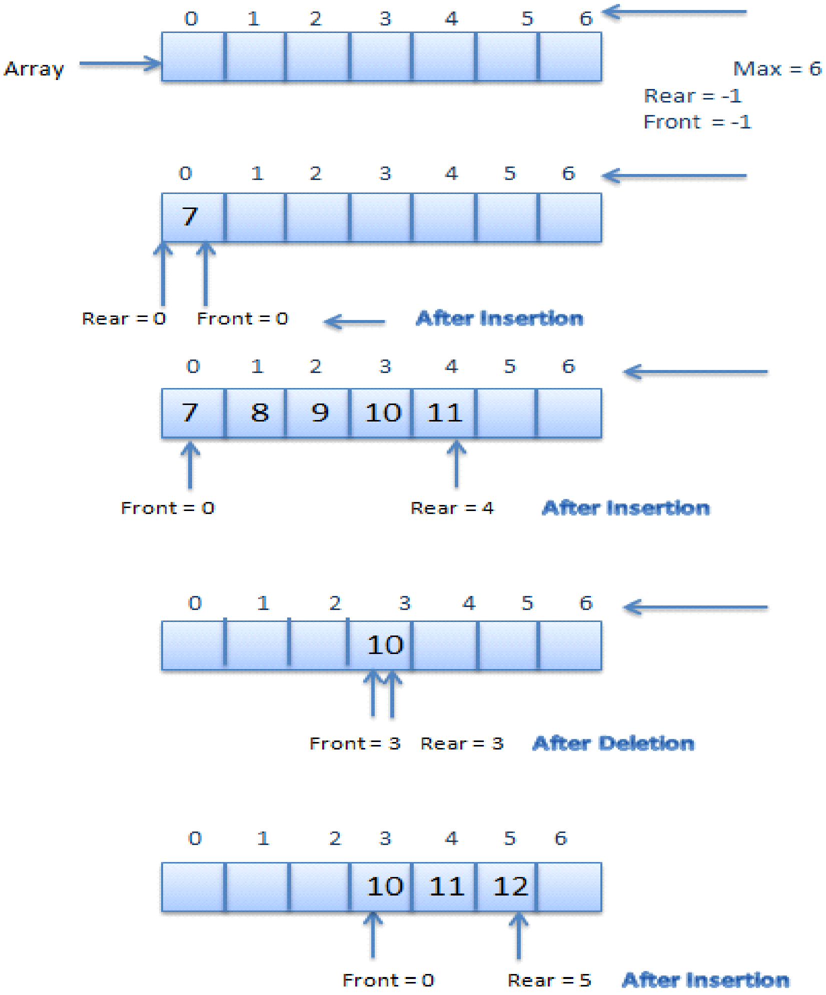


Operations on a queue:

The following operations are performed on a queue data structure...

- enQueue(value) - (To insert an element into the queue)
- deQueue() - (To delete an element from the queue)
- display() - (To display the elements of the queue)

Representation of queue with example:



Applications of queue:

- Print a file
- Queues are mostly used in operating systems.

- Waiting for a particular event to occur.
- Scheduling of processes.

Rear of Queue
Front of Queue
next...
PAY HERE

Implementation of queues:

We can implement queues in two ways.

- Arrays(static)
- Linked list(dynamic)

Queue implementation using arrays:

- 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
- Before we implement actual operations, first follow the below steps to create an empty queue.
    - Include all the header files which are used in the program and define a constant 'SIZE' with specific value.
    - Declare all the user defined functions which are used in queue implementation.
    - Create a one-dimensional array with above defined SIZE

      (int queue[SIZE])
    - Define two integer variables 'front' and 'rear' and initialize both with '-1'. (int front = -1, rear = -1)
    - Then implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on queue.

enQueue(value) - Inserting value into the queue:

- In a queue data structure, enQueue() is a function used to insert a new element into the queue.
- In a queue, the new element is always inserted at rear position.
- The enQueue() function takes one integer value as parameter and inserts that value into the queue.
- We can use the following steps to insert an element into the queue.
    - Check whether queue is FULL. (rear == SIZE-1)
    - If rear is FULL, then display "Queue is FULL!!! Insertion is not possible!!!" and terminate the function.
    - If it is NOT FULL, then increment rear value by one (rear++) and set queue[rear] = value.

```
void enQueue(int value){
  if(rear == SIZE-1)
    printf("\nQueue is Full!!! Insertion is not possible!!!");
  else{
    if(front == -1)
         front = 0;
    rear++;
    queue[rear] = value;
    printf("\nInsertion success!!!");
  }
}
```

deQueue() - Deleting a value from the Queue:

- In a queue data structure, deQueue() is a function used to delete an element from the queue.
- In a queue, the element is always deleted from front position.
- The deQueue() function does not take any value as parameter.
- We can use the following steps to delete an element from the queue...
- Check whether queue is EMPTY. (front == rear)
- If it is EMPTY, then display "Queue is EMPTY!!! Deletion is not possible!!!"
- If it is NOT EMPTY, then increment the front value by one (front ++). Then display queue[front] as deleted element. Then check whether both front and rear are equal (front == rear), if it TRUE, then set both front and rear to '-1' (front = rear = -1).

```
deQueue()

{

   if (front == - 1 || front > rear)

   {
       printf("Queue Underflow \n");

       return ;

   }

   else

   {

       printf("Element deleted from queue is : %d\n", queue_array[front]);

       front = front + 1;

   }

} /*End of deQueue() */
```

display() - Displays the elements of a Queue:

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

- Check whether queue is EMPTY. (front == rear)
- If it is EMPTY, then display "Queue is EMPTY!!!"
- If it is NOT EMPTY, then define an integer variable 'i' and set i = front
- Display 'queue[i]' value and increment 'i' value by one (i++).
- Repeat the same until 'i' value is equal to rear (i <= rear)

```
display()

{

    int i;

    if (front == - 1)

        printf("Queue is empty \n");

    else

    {

        printf("Queue is : \n");

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

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

        printf("\n");

    }

} /*End of display() */
```

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


#define MAX 5

int queue_array[MAX];

int rear = - 1;

int front = - 1;

main()

{

    int choice;

    while (1)

    {

        printf("1.Insert element to queue \n");

        printf("2.Delete element from queue \n");

        printf("3.Display all elements of queue \n");

        printf("4.Quit \n");

        printf("Enter your choice : ");

        scanf("%d", &choice);

        switch (choice)

        {

            case 1:  insert();   break;

            case 2:  delete();   break;

            case 3: display();   break;

            case 4: exit(1);

            default: printf("Wrong choice \n");

        } /*End of switch*/
    } /*End of while*/
} /*End of main()*/
insert()
{
    int add_item;
    if (rear == MAX - 1)
    printf("Queue Overflow \n");
    else
    {
        if (front == - 1)
        /*If queue is initially empty */
        front = 0;
        printf("Inset the element in queue : ");
        scanf("%d", &add_item);
        rear = rear + 1;
        queue_array[rear] = add_item;
    }
} /*End of insert()*/
```

```
delete()
{
    if (front == - 1 || front > rear)
    {
        printf("Queue Underflow \n");
        return ;
    }
    else
    {
        printf("Element deleted from queue is : %d\n", queue_array[front]);
        front = front + 1;
    }
} /*End of delete() */
display()
{
    int i;
    if (front == - 1)
        printf("Queue is empty \n");
    else
    {
        printf("Queue is : \n");
        for (i = front; i <= rear; i++)
            printf("%d ", queue_array[i]);
        printf("\n");

    }

} /*End of display() */mentation of queue using arrays
```

OUTPUT:

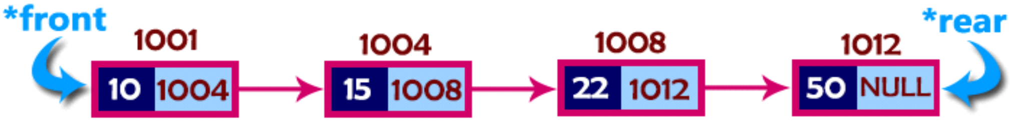
```
1.Insert element to queue

2.Delete element from queue

3.Display all elements of queue

4.Quit

Enter your choice : 3

Queue is empty

1.Insert element to queue

2.Delete element from queue

3.Display all elements of queue

4.Quit

Enter your choice : 1

Inset the element in queue : 20

1.Insert element to queue

2.Delete element from queue

3.Display all elements of queue

4.Quit

Enter your choice : 1

Inset the element in queue : 30

1.Insert element to queue
```

```
2.Delete element from queue

3.Display all elements of queue

4.Quit

Enter your choice : 1

Inset the element in queue : 40

1.Insert element to queue

2.Delete element from queue

3.Display all elements of queue

4.Quit

Enter your choice : 3

Queue is :

20 30 40

1.Insert element to queue

2.Delete element from queue

3.Display all elements of queue

4.Quit

Enter your choice : 2

Element deleted from queue is : 20

1.Insert element to queue

2.Delete element from queue

3.Display all elements of queue

4.Quit

Enter your choice : 2

Element deleted from queue is : 30

1.Insert element to queue
```

```
2.Delete element from queue

3.Display all elements of queue

4.Quit

Enter your choice : 2

Element deleted from queue is : 40

1.Insert element to queue

2.Delete element from queue

3.Display all elements of queue

4.Quit

Enter your choice : 2

Queue Underflow

1.Insert element to queue

2.Delete element from queue

3.Display all elements of queue

4.Quit

Enter your choice : 4
```

Linked List Implementation of Queue:

- 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'.**
- **Example:**



In above example, the last inserted node is 50 and it is pointed by 'rear' and the first inserted node is 10 and it is pointed by 'front'. The order of elements inserted is 10, 15, 22 and 50.

Operations:

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

- Include all the header files which are used in the program and declare all the user defined functions.
- Define a 'Node' structure with two fields data and next.
- Define two Node pointers 'front' and 'rear' and set both to NULL.
- 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

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

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

```
void insert(int value)
{
  struct Node *newNode;
  newNode = (struct Node*)malloc(sizeof(struct Node));
  newNode->data = value;
  newNode -> next = NULL;
  if(front == NULL)
    front = rear = newNode;
  else
  {
    rear -> next = newNode;
    rear = newNode;
  }
  printf("\nInsertion is Success!!!\n");
}
```

deQueue() - Deleting an Element from Queue:

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

- Check whether queue is Empty (front == NULL).
- If it is Empty, then display "Queue is Empty!!! "
- If it is Not Empty then, define a Node pointer 'temp' and set it to 'front'.
- Then set 'front = front → next' and delete 'temp' (free(temp)).

```
void delete()
{
  if(front == NULL)
    printf("\nQueue is Empty!!!\n");
  else
  {
    struct Node *temp = front;
    front = front -> next;
```

```
        printf("\nDeleted element: %d\n", temp->data);

        free(temp);

    }

}
```

display() - Displaying the elements of Queue

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

- Check whether queue is Empty (front == NULL).
- If it is Empty then, display 'Queue is Empty!!!' and terminate the function.
- If it is Not Empty then, define a Node pointer 'temp' and initialize with front.

- Display 'temp → data --->' and move it to the next node. Repeat the same until 'temp' reaches to 'rear' (temp → next != NULL).
- Finally! Display 'temp → data ---> NULL'.

```
void display()
{
  if(front == NULL)
    printf("\nQueue is Empty!!!\n");
  else
  {
    struct Node *temp = front;
    while(temp->next != NULL)
    {
      printf("%d--->",temp->data);
      temp = temp -> next;
    }
    printf("%d--->NULL\n",temp->data);
  }
}
```

```
// implementation of queue using linked list
#include<stdio.h>


struct Node
{
   int data;
   struct Node *next;
}*front = NULL,*rear = NULL;

void insert(int);
void delete();
void display();

void main()
{
   int choice, value;

   printf("\n:: Queue Implementation using Linked List ::\n");
   while(1)
   {
    printf("\n****** MENU ******\n");
    printf("1. Insert\n2. Delete\n3. Display\n4. Exit\n");
    printf("Enter your choice: ");
    scanf("%d",&choice);
    switch(choice){
        case 1: printf("Enter the value to be insert: ");
                scanf("%d", &value);
                insert(value);
                break;
        case 2: delete(); break;
        case 3: display(); break;
        case 4: exit(0);
        default: printf("\nWrong selection!!! Please try again!!!\n");
    }
  }
}
void insert(int value)
{
  struct Node *newNode;
  newNode = (struct Node*)malloc(sizeof(struct Node));
  newNode->data = value;
  newNode -> next = NULL;
  if(front == NULL)
    front = rear = newNode;
  else

  {

    rear -> next = newNode;

    rear = newNode;

  }

  printf("\nInsertion is Success!!!\n");

}

void delete()

{

  if(front == NULL)

    printf("\nQueue is Empty!!!\n");

  else

  {

    struct Node *temp = front;

    front = front -> next;

    printf("\nDeleted element: %d\n", temp->data);

    free(temp);

  }

}

void display()

{

  if(front == NULL)

    printf("\nQueue is Empty!!!\n");

  else

  {
    struct Node *temp = front;

    while(temp->next != NULL)

    {

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

      temp = temp -> next;

    }

    printf("%d--->NULL\n",temp->data);

  }

}
```

```
OUTPUT:


:: Queue Implementation using Linked List ::


****** MENU ******

1. Insert

2. Delete

3. Display

4. Exit

Enter your choice: 3


Queue is Empty!!!


****** MENU ******
```

```
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 10

Insertion is Success!!!

****** MENU ******
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 20

Insertion is Success!!!

****** MENU ******
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 3
```

```
10--->20--->NULL

****** MENU ******
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 2

Deleted element: 10

****** MENU ******
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 3
20--->NULL

****** MENU ******
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 4
```

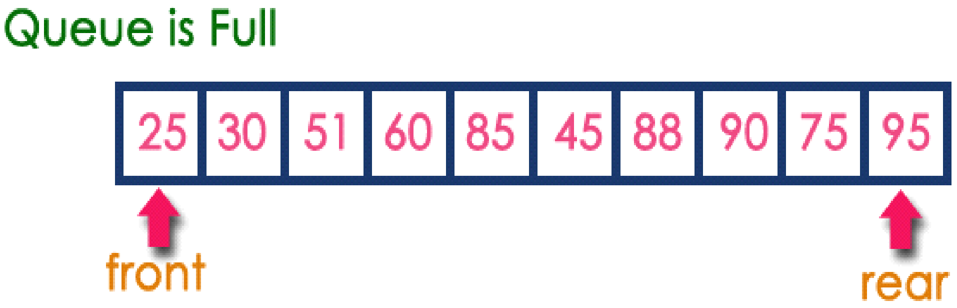
Types of Queues:

- Circular queue
- Double ended queue
- Priority queue

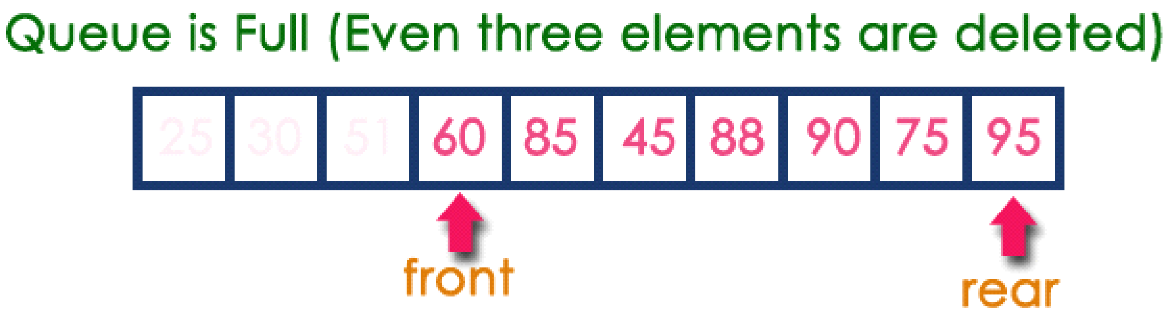
1.Circular queue:

- In a normal Queue Data Structure, we can insert elements until queue becomes full.
- But once if queue becomes full, we can not insert the next element until all the elements are deleted from the queue.
- For example consider the queue below...

  After inserting all the elements into the queue.



Now consider the following situation after deleting three elements from the queue...

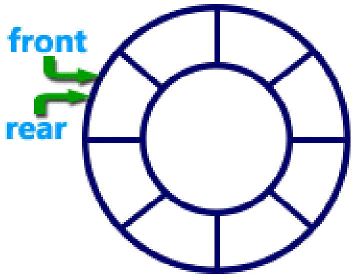


- This situation also says that Queue is Full and we can not insert the new element because, 'rear' is still at last position.
- In above situation, even though we have empty positions in the queue we can not make use of them to insert new element.
- This is the major problem in normal queue data structure.
- To overcome this problem we use circular queue data structure.

Circular Queue can be defined as….

Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle.

Graphical representation of a circular queue is as follows...



Representation of circular queue:

Let us consider a circular queue, which can hold maximum (MAX) of six elements. Initially the queue is empty.

F R

Queue Empty

MAX = 6

FRONT = REAR = 0

COUNT = 0

Circular Queue

Now, insert 11 to the circular queue. Then circular queue status will be:

R

FRONT = 0

4 REAR = (REAR + 1) % 6 = 1

COUNT = 1

Insert new elements 22, 33, 44 and 55 into the circular queue. The circular queue status is:

F

FRONT = 0

REAR = (REAR + 1) % 6 = 5

COUNT = 5

Now, delete an element. The element deleted is the element at the front of the circular queue. So, 11 is deleted. The circular queue status is as follows:

F

FRONT = (FRONT + 1) % 6 = 1

4 REAR = 5

COUNT = COUNT - 1 = 4

Again, delete an element. The element to be deleted is always pointed to by the FRONT pointer. So, 22 is deleted. The circular queue status is as follows:

FRONT = (FRONT + 1) % 6 = 2

4 REAR = 5

COUNT = COUNT - 1 = 3

F

Again, insert another element 66 to the circular queue. The status of the circular queue is:

^{4} FRONT = 2

REAR = (REAR + 1) % 6 = 0

COUNT = COUNT + 1 = 4

F

Now, insert new elements 77 and 88 into the circular queue. The circular queue status is:

^{4} FRONT = 2, REAR = 2

REAR = REAR % 6 = 2

COUNT = 6

R

Circular Queue

Now, if we insert an element to the circular queue, as COUNT = MAX we cannot add the element to circular queue. So, the circular queue is *full*.

```
# include <stdio.h>
# define MAX 6
int CQ[MAX];
int front = 0;
int rear = 0;
int count = 0;

void main()
{
  int ch;
  while(1)
  {
    ch = menu();
    switch(ch)
    {
        case 1: insertCQ();
                break;
        case 2: deleteCQ();
                break;
        case 3: displayCQ();
                break;
        case 4: return;
        default:
                printf("\n Invalid Choice ");
    }
 }
}


void insertCQ()
{
 int data;
 if(count ==MAX)
 {
  printf("\n Circular Queue is Full");
 }
 else
 {
  printf("\n Enter data: ");
  scanf("%d", &data);

  CQ[rear] = data;

  rear = (rear + 1) % MAX;

  count ++;

  printf("\n Data Inserted in the Circular Queue ");

 }

}





void deleteCQ()

{

 if(count ==0)

 {

  printf("\n\nCircular Queue is Empty..");

 }

 else

 {

  printf("\n Deleted element from Circular Queue is %d ", CQ[front]);

  front = (front + 1) % MAX;

  count --;

 }

}
```

```
void displayCQ()
{
 int i, j;
 if(count ==0)
 {
  printf("\n\n\t Circular Queue is Empty ");
 }
 else
 {
  printf("\n Elements in Circular Queue are: ");
  j = count;
  for(i = front; j!= 0; j--)
  {
    printf("%d\t", CQ[i]);
    i = (i + 1)%MAX;
  }
 }
}
int menu()
{
 int ch;
  printf("\n \t Circular Queue Operations using ARRAY..");
  printf("\n -----------**********-------------\n");
  printf("\n 1. Insert ");
  printf("\n 2. Delete ");
  printf("\n 3. Display");

  printf("\n 4. Quit ");

  printf("\n Enter Your Choice: ");

  scanf("%d", &ch);

  return ch;

}
```

OUTPUT:

```
        Circular Queue Operations using ARRAY..

 -----------**********-------------


 1. Insert

 2. Delete

 3. Display

 4. Quit

 Enter Your Choice: 3



         Circular Queue is Empty

         Circular Queue Operations using ARRAY..

 -----------**********-------------


 1. Insert

 2. Delete

 3. Display

 4. Quit
```

```
Enter Your Choice: 1

Enter data: 10

Data Inserted in the Circular Queue
        Circular Queue Operations using ARRAY..
-----------**********-------------

1. Insert
2. Delete
3. Display
4. Quit
Enter Your Choice: 1

Enter data: 20

Data Inserted in the Circular Queue
        Circular Queue Operations using ARRAY..
-----------**********-------------

1. Insert
2. Delete
3. Display
4. Quit
Enter Your Choice: 1
Enter data: 30


Data Inserted in the Circular Queue

        Circular Queue Operations using ARRAY..

-----------**********-------------


1. Insert

2. Delete

3. Display

4. Quit

Enter Your Choice: 3


Elements in Circular Queue are: 10     20      30

        Circular Queue Operations using ARRAY..

-----------**********-------------


1. Insert

2. Delete

3. Display

4. Quit

Enter Your Choice: 2


Deleted element from Circular Queue is 10

        Circular Queue Operations using ARRAY..
```

```
-----------**********-------------

1. Insert
2. Delete
3. Display
4. Quit
Enter Your Choice: 3

Elements in Circular Queue are: 20     30
        Circular Queue Operations using ARRAY..
-----------**********-------------

1. Insert
2. Delete
3. Display
4. Quit
Enter Your Choice: 1

Enter data: 40

Data Inserted in the Circular Queue
        Circular Queue Operations using ARRAY..
-----------**********-------------

1. Insert
```

```
2. Delete

3. Display

4. Quit

Enter Your Choice: 1


Enter data: 50


Data Inserted in the Circular Queue

        Circular Queue Operations using ARRAY..

-----------**********-------------


1. Insert

2. Delete

3. Display

4. Quit

Enter Your Choice: 1


Enter data: 60


Data Inserted in the Circular Queue

        Circular Queue Operations using ARRAY..

-----------**********-------------


1. Insert

2. Delete
```

```
3. Display
4. Quit
Enter Your Choice: 3

Elements in Circular Queue are: 20    30    40    50    60
        Circular Queue Operations using ARRAY..
-----------**********-------------


1. Insert
2. Delete
3. Display
4. Quit
Enter Your Choice: 1

Enter data: 25

Data Inserted in the Circular Queue
        Circular Queue Operations using ARRAY..
-----------**********-------------
1. Insert
2. Delete
3. Display
4. Quit
Enter Your Choice: 3
```

```
Elements in Circular Queue are: 20	30	40	50	60	25


	Circular Queue Operations using ARRAY..
-----------**********-------------
1. Insert
2. Delete
3. Display
4. Quit
Enter Your Choice: 1
Circular Queue is Full
	Circular Queue Operations using ARRAY..
-----------**********-------------
1. Insert
2. Delete
3. Display
4. Quit
Enter Your Choice: 2


Deleted element from Circular Queue is 20
	Circular Queue Operations using ARRAY..
-----------**********-------------
1. Insert
2. Delete
3. Display
4. Quit
```

```
Enter Your Choice: 1

Enter data: 35

Data Inserted in the Circular Queue
        Circular Queue Operations using ARRAY..
-----------**********-------------

1. Insert
2. Delete
3. Display
4. Quit
Enter Your Choice: 3

Elements in Circular Queue are: 30     40      50      60      25      35

        Circular Queue Operations using ARRAY..
-----------**********-------------

1. Insert
2. Delete
3. Display
4. Quit
Enter Your Choice: 4
```

Double Ended Queue

Double ended queue is a more generalized form of queue data structure which allows insertion and removal of elements from both the ends, i.e , front and back.

Double Ended Queue

Implementation of Double ended Queue

Here we will implement a double ended queue using a circular array. It will have the following methods:

push_back : inserts element at back

push_front : inserts element at front

pop_back : removes last element

pop_front : removes first element

get_back : returns last element

get_front : returns first element

empty : returns true if queue is empty

full : returns true if queue is full

```
// Maximum size of array or Dequeue

#define SIZE 5

class Dequeue
{
    //front and rear to store the head and tail pointers
    int  *arr;
    int front, rear;
    public :

    Dequeue()
    {
            //Create the array
        arr = new int[SIZE];

        //Initialize front and rear with -1
        front = -1;
        rear = -1;
    }

    // Operations on Deque
    void  push_front(int );
    void  push_back(int );
    void  pop_front();
    void  pop_back();
    int  get_front();
    int  get_back();
    bool  full();
    bool  empty();
};
```

Insert Elements at Front

First we check if the queue is full. If its not full we insert an element at front end by following the given conditions :

If the queue is empty then intialize front and rear to 0. Both will point to the first element.

Double Ended Queue

Else we decrement front and insert the element. Since we are using circular array, we have to keep in mind that if front is equal to 0 then instead of decreasing it by 1 we make it equal to SIZE-1.

Double Ended Queue

```
void Dequeue :: push_front(int key)
{
    if(full())
    {
        cout << "OVERFLOW\n";
    }
    else
    {
            //If queue is empty
            if(front == -1)
                        front = rear = 0;

            //If front points to the first position element
        else if(front == 0)
            front = SIZE-1;

        else
            --front;

        arr[front] = key;
    }
}
```

Insert Elements at back

Again we check if the queue is full. If its not full we insert an element at back by following the given conditions:

If the queue is empty then intialize front and rear to 0. Both will point to the first element.

Else we increment rear and insert the element. Since we are using circular array, we have to keep in mind that if rear is equal to SIZE-1 then instead of increasing it by 1 we make it equal to 0.

Double Ended Queue

```
void Dequeue :: push_back(int key)
{
    if(full())
    {
        cout << "OVERFLOW\n";
    }
    else
    {
        //If queue is empty
               if(front == -1)
                          front = rear = 0;

               //If rear points to the last element
        else if(rear == SIZE-1)
            rear = 0;

        else
            ++rear;

        arr[rear] = key;
    }
}
```

Delete First Element

In order to do this, we first check if the queue is empty. If its not then delete the front element by following the given conditions :

If only one element is present we once again make front and rear equal to -1.

Else we increment front. But we have to keep in mind that if front is equal to SIZE-1 then instead of increasing it by 1 we make it equal to 0.

Double Ended Queue

```
void Dequeue :: pop_front()
{
    if(empty())
    {
        cout << "UNDERFLOW\n";
    }
    else
    {
            //If only one element is present
        if(front == rear)
            front = rear = -1;

        //If front points to the last element
        else if(front == SIZE-1)
            front = 0;

        else
            ++front;
    }
}
```

Delete Last Element

Inorder to do this, we again first check if the queue is empty. If its not then we delete the last element by following the given conditions :

If only one element is present we make front and rear equal to -1.

Else we decrement rear. But we have to keep in mind that if rear is equal to 0 then instead of decreasing it by 1 we make it equal to SIZE-1.

Double Ended Queue

```
void Dequeue :: pop_back()
{
    if(empty())
    {
        cout << "UNDERFLOW\n";
    }
    else
    {
            //If only one element is present
        if(front == rear)
            front = rear = -1;

        //If rear points to the first position element
        else if(rear == 0)
            rear = SIZE-1;

        else
            --rear;
    }
}
```

Check if Queue is empty

It can be simply checked by looking where front points to. If front is still intialized with -1, the queue is empty.

```
bool Dequeue :: empty()
```

```
{
    if(front == -1)
        return true;
    else
        return false;
}
```

Check if Queue is full

Since we are using circular array, we check for following conditions as shown in code to check if queue is full.

```
bool Dequeue :: full()
{
    if((front == 0 && rear == SIZE-1)  ||
        (front == rear + 1))
      return true;
    else
      return false;
}
```

Return First Element

If the queue is not empty then we simply return the value stored in the position which front points.

```
int Dequeue :: get_front()
{
    if(empty())
    {      cout << "f=" <<front << endl;
```

```
        cout << "UNDERFLOW\n";
        return -1;
    }
    else
    {
        return arr[front];
    }
}
```

Priority queue:

Priority Queue is more specialized data structure than Queue. Like ordinary queue, priority queue has same method but with a major difference. In Priority queue items are ordered by key value so that item with the lowest value of key is at front and item with the highest value of key is at rear or vice versa. So we're assigned priority to item based on its key value. Lower the value, higher the priority. Following are the principal methods of a Priority Queue.

Basic Operations

insert / enqueue − add an item to the rear of the queue.

remove / dequeue − remove an item from the front of the queue.

Priority Queue Representation

Queue

We're going to implement Queue using array in this article. There is few more operations supported by queue which are following.

Peek − get the element at front of the queue.

isFull ─ check if queue is full.

isEmpty ─ check if queue is empty.

Insert / Enqueue Operation

Whenever an element is inserted into queue, priority queue inserts the item according to its order. Here we're assuming that data with high value has low priority.

Insert Operation

```
void insert(int data){
   int i = 0;

   if(!isFull()){
      // if queue is empty, insert the data

      if(itemCount == 0){
         intArray[itemCount++] = data;
      }else{
         // start from the right end of the queue
         for(i = itemCount - 1; i >= 0; i-- ){
            // if data is larger, shift existing item to right end
            if(data > intArray[i]){
               intArray[i+1] = intArray[i];
            }else{
               break;
            }
         }
         // insert the data
         intArray[i+1] = data;
         itemCount++;
      }
   }
}
```

Remove / Dequeue Operation

Whenever an element is to be removed from queue, queue get the element using item count. Once element is removed. Item count is reduced by one.

Queue Remove Operation

```
int removeData(){
   return intArray[--itemCount];
}
```

Demo Program

PriorityQueueDemo.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>
#define MAX 6
```

```
int intArray[MAX];
int itemCount = 0;

int peek(){
   return intArray[itemCount - 1];
}

bool isEmpty(){
   return itemCount == 0;
}

bool isFull(){
   return itemCount == MAX;
}

int size(){
   return itemCount;
}

void insert(int data){
   int i = 0;

   if(!isFull()){
      // if queue is empty, insert the data
      if(itemCount == 0){
         intArray[itemCount++] = data;
      }else{
         // start from the right end of the queue

         for(i = itemCount - 1; i >= 0; i-- ){
            // if data is larger, shift existing item to right end
            if(data > intArray[i]){
               intArray[i+1] = intArray[i];
            }else{
               break;
            }
         }

         // insert the data
         intArray[i+1] = data;
         itemCount++;
      }
   }
}

int removeData(){
   return intArray[--itemCount];
}

int main() {
```

```
/* insert 5 items */
insert(3);
insert(5);
insert(9);
insert(1);
insert(12);

// ------------------
// index : 0  1 2 3 4
// ------------------
// queue : 12 9 5 3 1
insert(15);

// ---------------------
// index : 0  1 2 3 4  5
// ---------------------
// queue : 15 12 9 5 3 1

if(isFull()){
   printf("Queue is full!\n");
}

// remove one item
int num = removeData();
printf("Element removed: %d\n",num);
```

```
// ---------------------
// index : 0  1  2 3 4
// ---------------------
// queue : 15 12 9 5 3

// insert more items
insert(16);

// ----------------------
// index :  0  1 2 3 4  5
// ----------------------
// queue : 16 15 12 9 5 3

// As queue is full, elements will not be inserted.
insert(17);
insert(18);

// ----------------------
// index : 0   1  2 3 4 5
// ----------------------
// queue : 16 15 12 9 5 3
printf("Element at front: %d\n",peek());

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

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

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

   printf("Queue:  ");


   while(!isEmpty()){

      int n = removeData();

      printf("%d ",n);

   }

}
```

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

```
Queue is full!

Element removed: 1

Element at front: 3

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

index : 5 4 3 2 1 0

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

Queue: 3 5 9 12 15 16
```

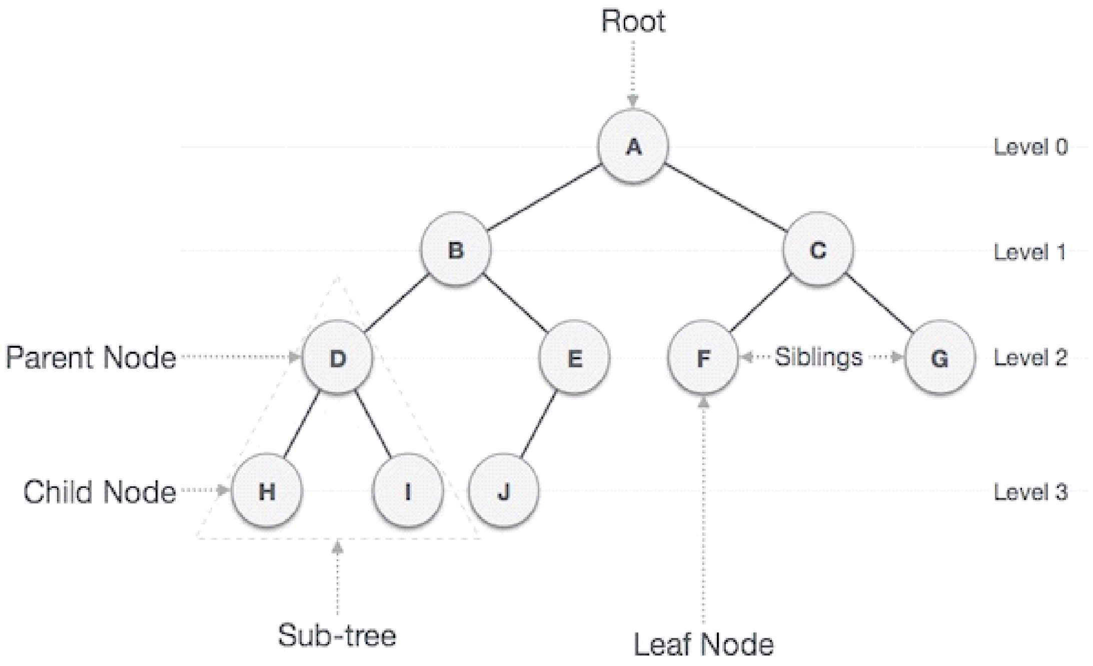
MODULE-IV:Trees and Graphs[10 Periods]

Trees: Basic concepts of Trees, Binary Tree: Properties, Representation of binary tree using array and linked lists, operations on a binary tree, binary tree traversals, creation of binary tree from in, pre and post-order traversals, Tree traversals using stack, Threaded binary tree.

Graphs: Basic concepts of Graphs, Representation of Graphs using Linked list and Adjacency matrix, Graph algorithms, Graph traversals- (BFS & DFS).

- A tree is hierarchical collection of nodes.

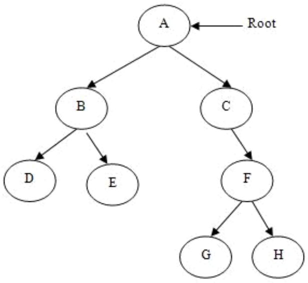
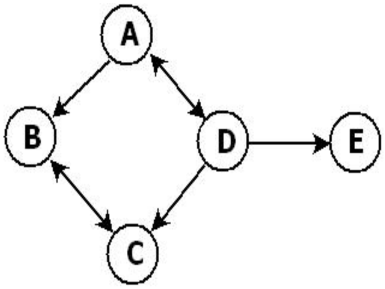
- One of the nodes, known as the root, is at the top of the hierarchy.
- Each node can have at most one link coming into it.
- The node where the link originates is called the parent node.
- The root node has no parent.
- The links leaving a node (any number of links are allowed) point to child nodes.
- Trees are recursive structures. Each child node is itself the root of a subtree.
- At the bottom of the tree are leaf nodes, which have no children.



-

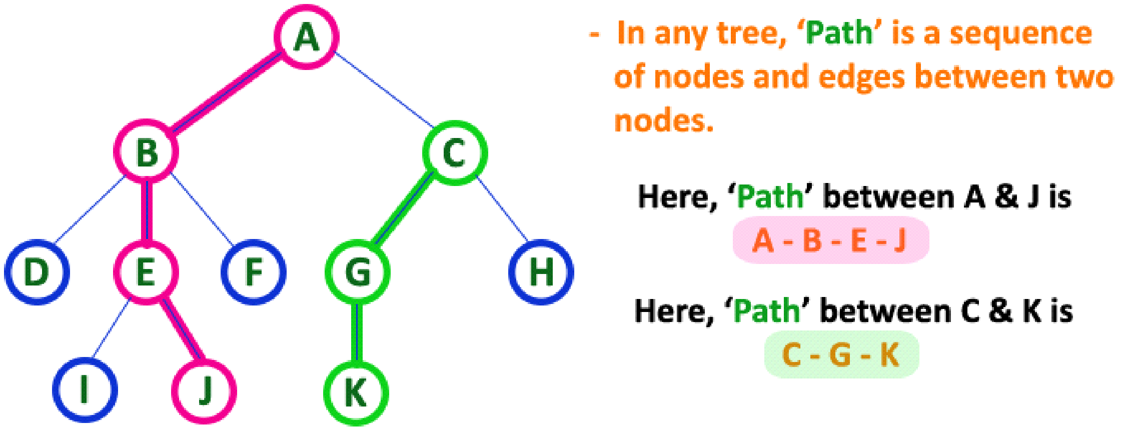
Difference between Trees and Graphs

	Trees	Graphs
Path	Tree is special form of graph i.e. **minimally connected graph** and having only one path between any two vertices.	In graph there can be more than one path i.e. graph can have uni-directional or bi-directional paths (edges) between nodes
Loops	Tree is a special case of graph having no **loops**, no **circuits** and no self-loops.	Graph can have loops, circuits as well as can have **self-loops**.
Root Node	In tree there is exactly one root node and every **child** have only one **parent**.	In graph there is no such concept of **root** node.
Parent Child relationship	In trees, there is parent child relationship so flow can be there with direction top to bottom or vice versa.	In Graph there is no such parent child relationship.
Complexity	Trees are less complex then graphs as having no cycles, no self-loops and still connected.	Graphs are more complex in compare to trees as it can have cycles, loops etc
Types of Traversal	Tree traversal is a kind of special case of traversal of graph. Tree is traversed in **Pre-Order**, **In-Order** and **Post-Order** (all three in DFS or in BFS algorithm)	Graph is traversed by **DFS: Depth First Search** and in **BFS : Breadth First Search algorithm**
Connection Rules	In trees, there are many rules / restrictions for making connections between nodes	In graphs no such rules/ restrictions are there for connecting the nodes through

	through edges.	edges.
DAG	Trees come in the category of **DAG : Directed Acyclic Graphs** is a kind of directed graph that have no cycles.	Graph can be **Cyclic or Acyclic**.
Different Types	Different types of trees are : **Binary Tree , Binary Search Tree, AVL tree, Heaps**.	There are mainly two types of Graphs : **Directed and Undirected graphs**.
Applications	Tree applications : sorting and searching like Tree Traversal & Binary Search.	Graph applications : Coloring of maps, in OR (**PERT & CPM**), algorithms, Graph coloring, job scheduling, etc.
No. of edges	Tree always has n-1 edges.	In Graph, no. of edges depend on the graph.
Model	Tree is a hierarchical model.	Graph is a network model.
Figure		

Following are the important terms with respect to tree.

Path − Path refers to the sequence of nodes along the edges of a tree.

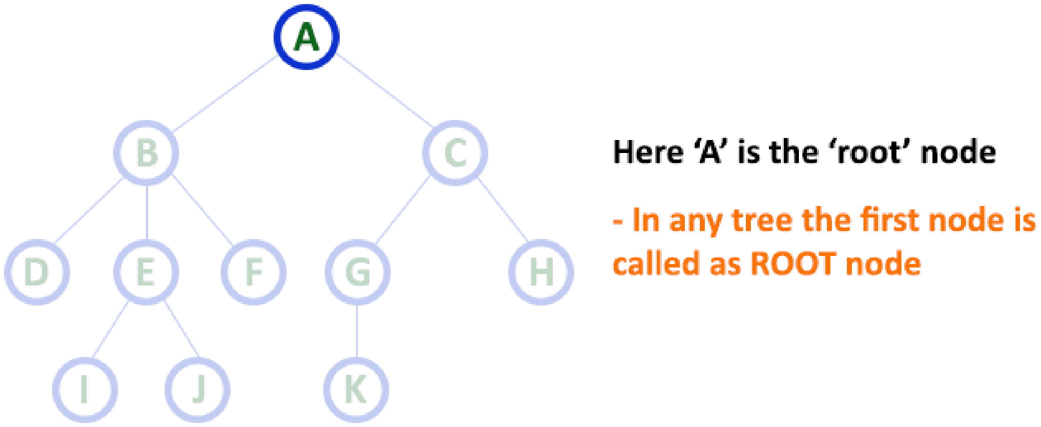


- In any tree, 'Path' is a sequence of nodes and edges between two nodes.

Here, 'Path' between A & J is
A - B - E - J

Here, 'Path' between C & K is
C - G - K

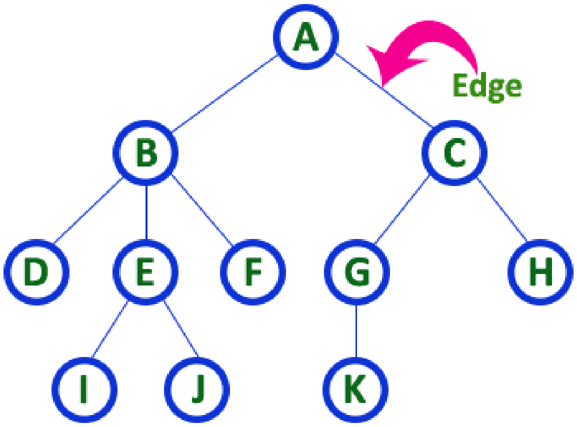
Root – The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.



Here 'A' is the 'root' node

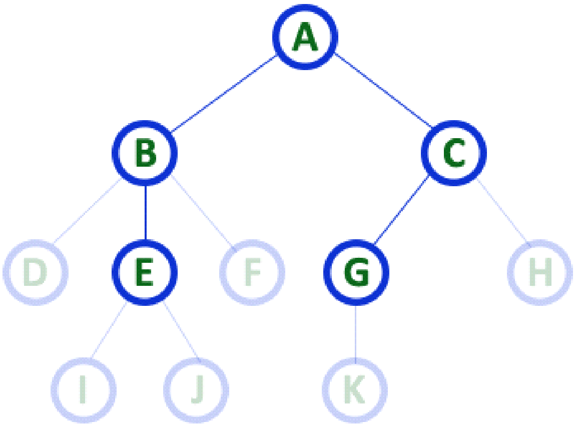
- In any tree the first node is called as ROOT node

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.

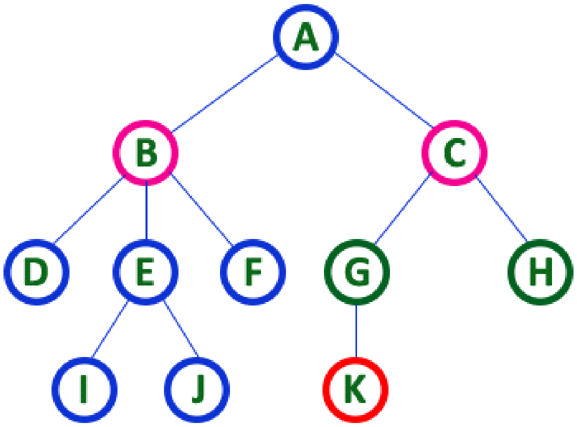
Parent − Any node except the root node has one edge upward to a node called parent.



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'

Child − The node below a given node connected by its edge downward is called its child node.



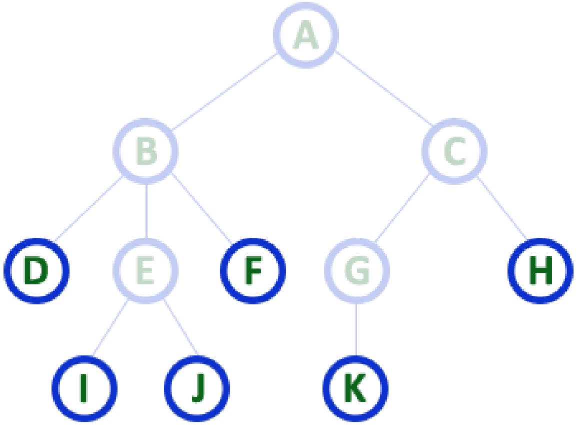
Here B & C are Children of A

Here G & H are Children of C

Here K is Child of G

- descendant of any node is called as CHILD Node

Leaf – The node which does not have any child node is called the leaf node.

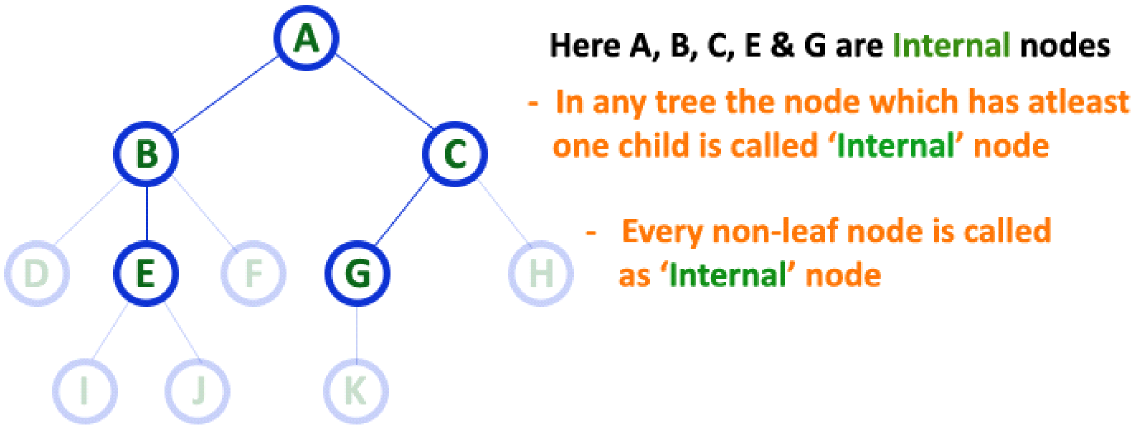


Here D, I, J, F, K & H are Leaf nodes

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

Internal 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.

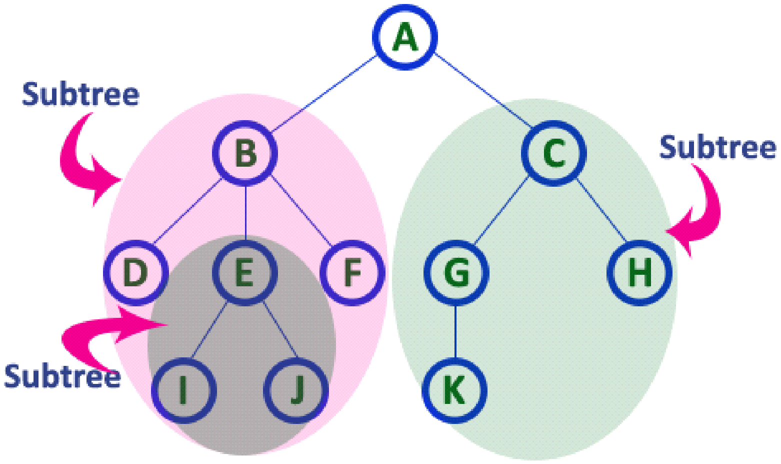


Here A, B, C, E & G are Internal nodes

- In any tree the node which has atleast one child is called 'Internal' node

- Every non-leaf node is called as 'Internal' node

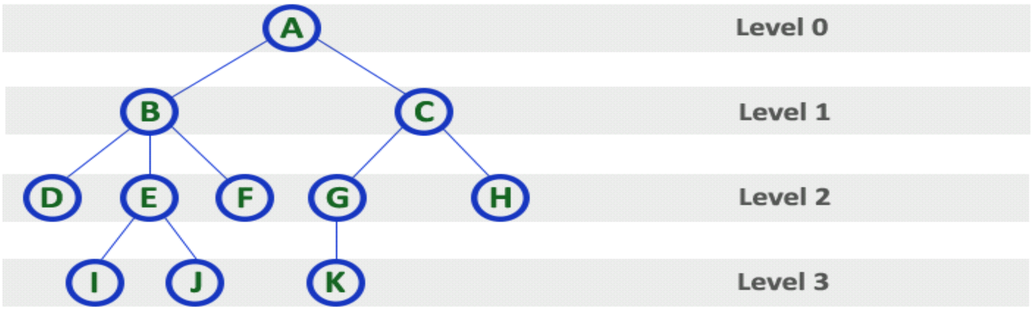
Subtree – Subtree represents the descendants of a node.



Visiting – Visiting refers to checking the value of a node when control is on the node.

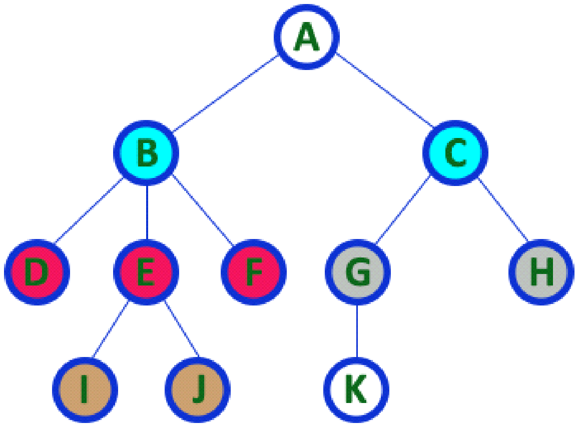
Traversing – Traversing means passing through nodes in a specific order.

Levels – Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.



keys – Key represents a value of a node based on which a search operation is to be carried out for a node.

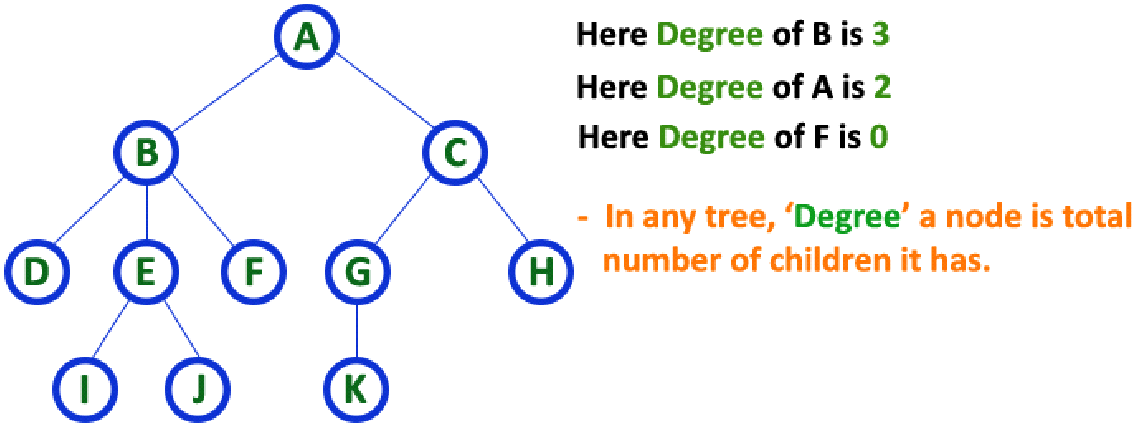
Siblings – In a tree data structure, nodes which belong to same Parent are called as SIBLINGS. In simple words, the nodes with same parent are called as Sibling nodes.



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'

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 'Degree of T.



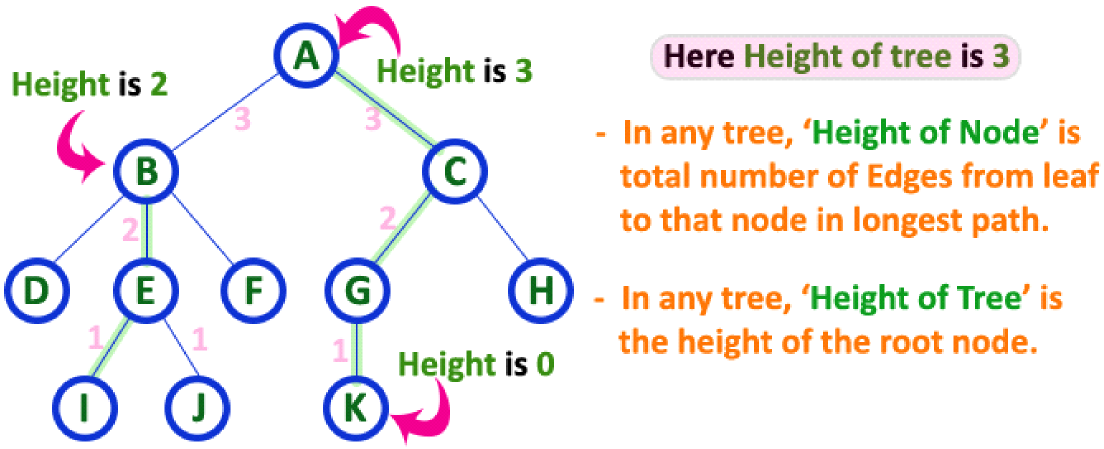
Here Degree of B is 3

Here Degree of A is 2

Here Degree of F is 0

- In any tree, ‘Degree’ a node is total number of children it has.

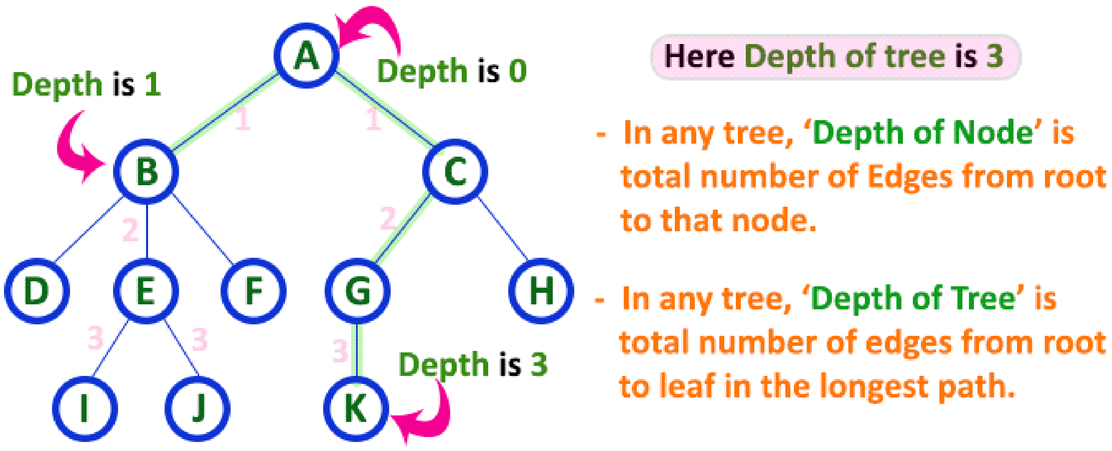
Height - In a tree data structure, the total number of egdes from leaf node to a particular node in the longest path is called as HEIGHT of that Node. In a tree, height of the root node is said to be height of the tree. In a tree, height of all leaf nodes is '0'.



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.

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

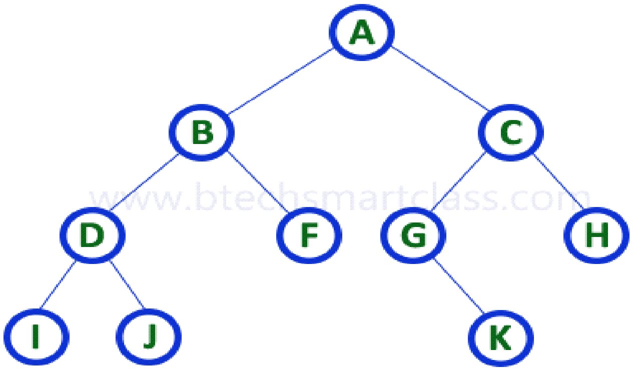
- In any tree, 'Depth of Node' is total number of Edges from root to that node.
- In any tree, 'Depth of Tree' is total number of edges from root to leaf in the longest path.

Binary Tree:

In general, tree nodes can have any number of children. In a binary tree, each node can have at most two children. A binary tree is either empty or consists of a node called the root together with two binary trees called the left subtree and the right subtree.

A tree in which every node can have a maximum of two children is called as Binary Tree.

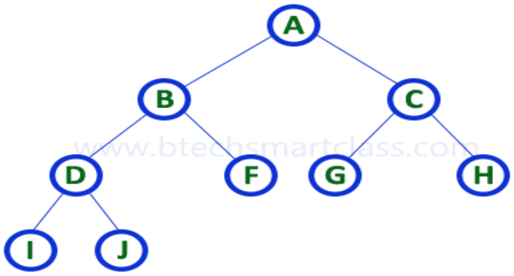
Example:



There are different types of binary trees and they are…

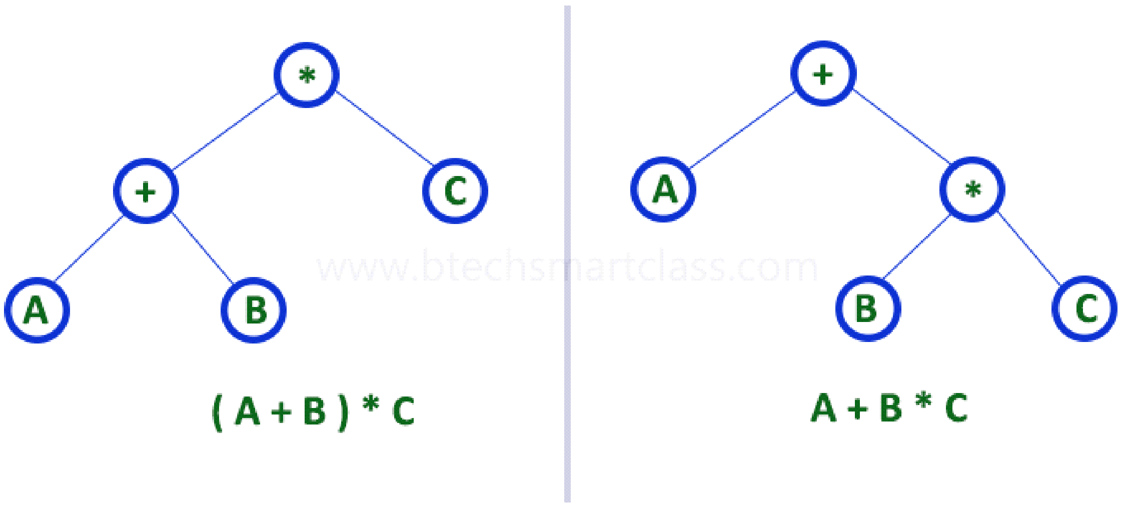
- **Strictly Binary tree:** A binary tree in which every node has either two or zero number of children is called Strictly Binary Tree. Strictly binary tree is also called as Full Binary Tree or Proper Binary Tree or 2-Tree.

Example:



Strictly binary tree data structure is used to represent mathematical expressions.

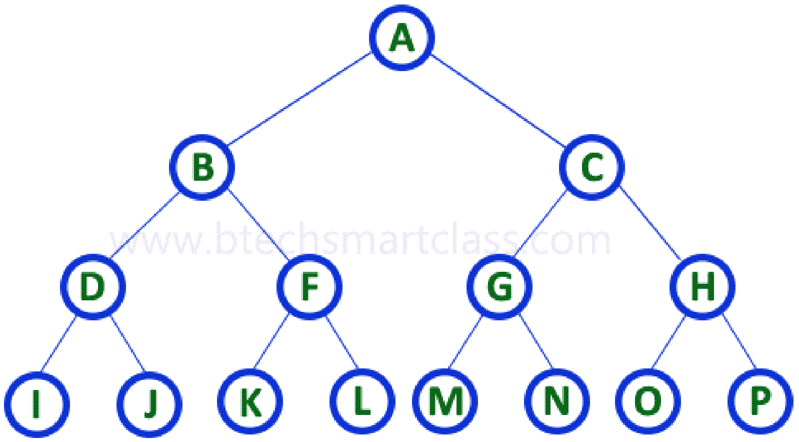
Example:



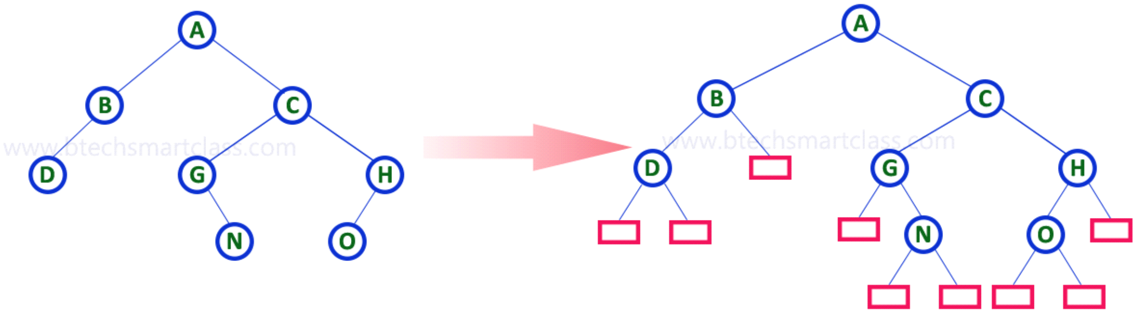
- **Complete Binary Tree:** In a binary tree, every node can have a maximum of two children. But in strictly binary tree, every node should have exactly two children or none and in complete binary tree all the nodes must have exactly two children and at every level of complete binary tree there must be 2level number of nodes.

  For example, at level 2 there must be 2^2 = 4 nodes and at level 3 there must be 2^3 = 8 nodes.

  Complete binary tree is also called as Perfect Binary Tree.



- **Extended Binary Tree:** A binary tree can be converted into Full Binary tree by adding dummy nodes to existing nodes wherever required.



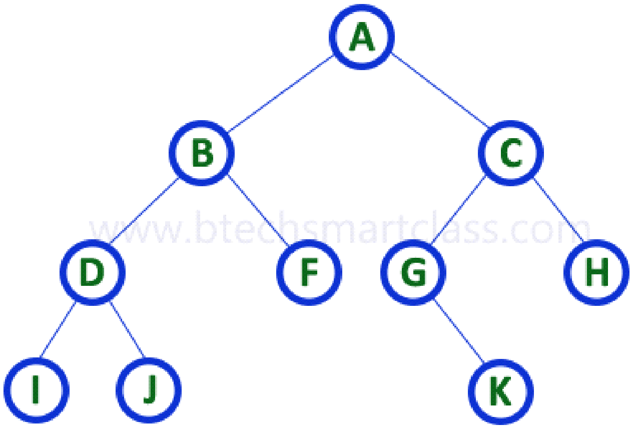
In above figure, a normal binary tree is converted into full binary tree by adding dummy nodes (In pink colour).

Binary Tree Representations:

A binary tree data structure is represented using two methods. Those methods are as follows...

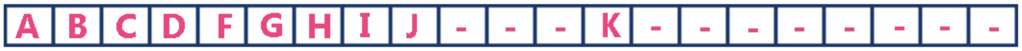
- Array Representation
- Linked List Representation

Consider the following binary tree...



1**. Array Representation**

In array representation of binary tree, we use a one-dimensional array (1-D Array) to represent a binary tree. Consider the above example of binary tree and it is represented as follows...



A	B	C	D	F	G	H	I	J	-	-	-	K	-	-	-	-	-	-	-	-

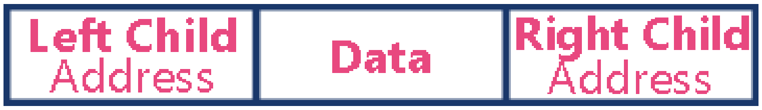
To represent a binary tree of depth 'n' using array representation, we need one dimensional array with a maximum size of 2^(n+1) - 1.

if the tree is a complete binary tree, this method wastes no space. In this compact arrangement, if a node has an index i, its children are found at indices **2i+1 and 2i+2**, while its parent (if any) is found at index floor((i-1)/2) (assuming the root of the tree stored in the array at an index zero).

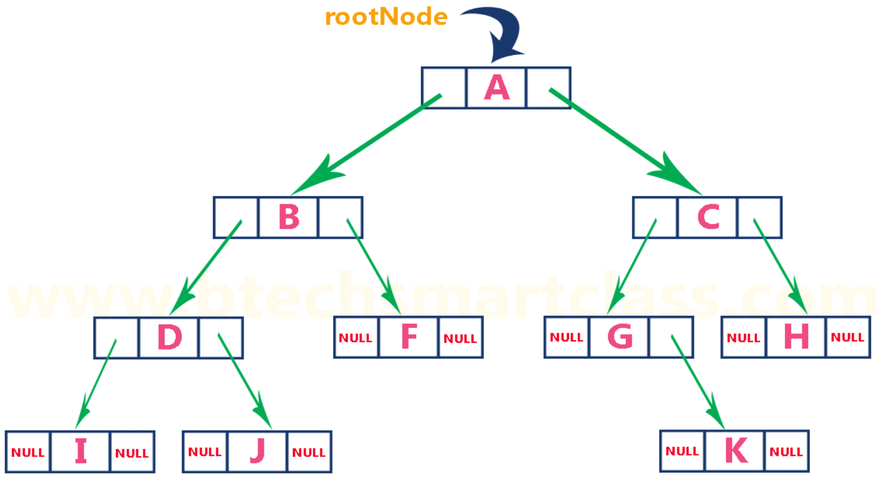
This method benefits from more compact storage and better locality of reference, particularly during a preorder traversal. However, it requires contiguous memory, expensive to grow and wastes space proportional to (2^h)- n for a tree of height h with n nodes.

2.Linked List Representation: We use double linked list to represent a binary tree. In a double linked list, every node consists of three fields. First field for storing left child address, second for storing actual data and third for storing right child address.

In this linked list representation, a node has the following structure...



The above example of binary tree represented using Linked list representation is shown as follows...



Binary Tree Traversals:

When we wanted to display a binary tree, we need to follow some order in which all the nodes of that binary tree must be displayed. In any binary tree displaying order of nodes depends on the traversal method.

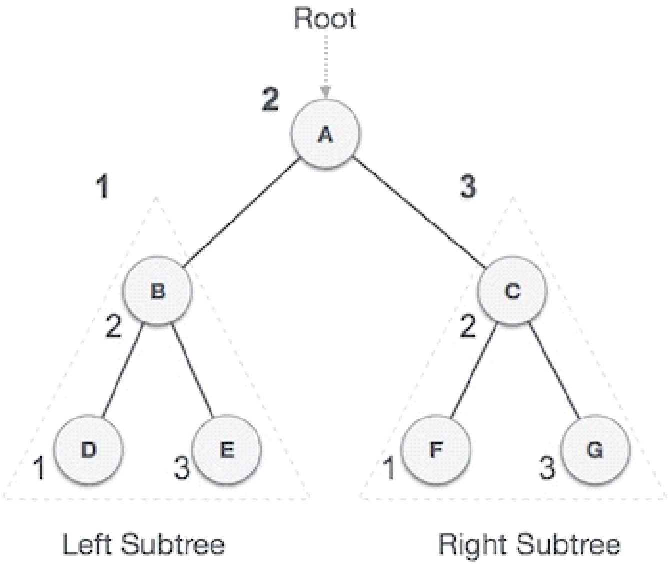
Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree –

- In-order Traversal

- Pre-order Traversal
- Post-order Traversal

In-order Traversal

- In this traversal method, the left subtree is visited first, then the root and later the right sub-tree.
- We should always remember that every node may represent a subtree itself.
- If a binary tree is traversed in-order, the output will produce sorted key values in an ascending order.



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

D → B → E → A → F → C → G

Algorithm

Until all nodes are traversed −

Step 1 − Recursively traverse left subtree.

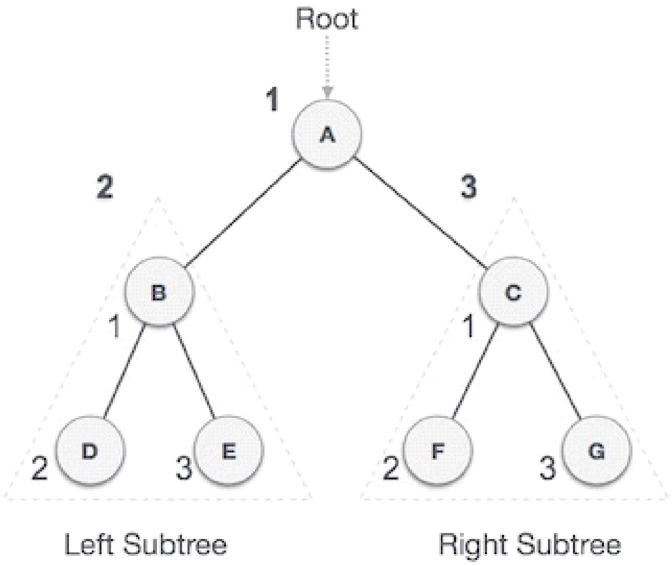
Step 2 − Visit root node.

Step 3 − Recursively traverse right subtree.

```
void inorder(node *root)
{
        if(root != NULL)
        {
                inorder(root->lchild);
                 print root -> data;
                inorder(root->rchild);
        }
}
```

PreOrder:

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



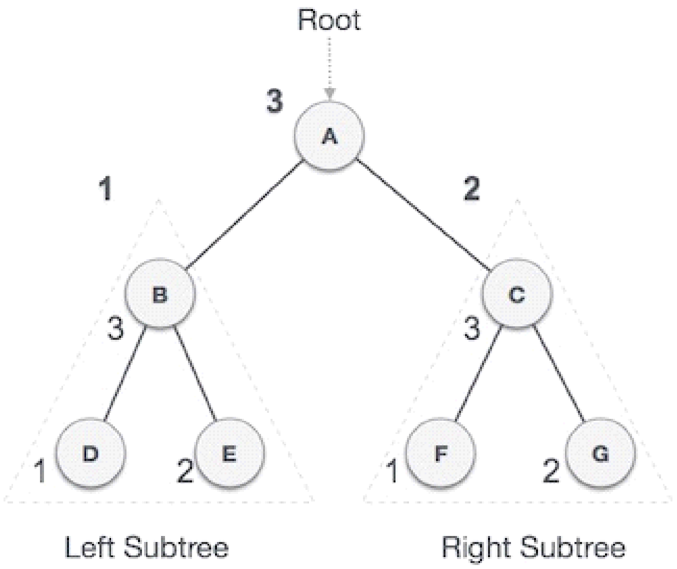
We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left subtree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be −

A → B → D → E → C → F → G

```
Algorithm PreOrder

Until all nodes are traversed −

Step 1 − Visit root node.

Step 2 − Recursively traverse left subtree.

Step 3 − Recursively traverse right subtree.

void preorder(node *root)

{

        if( root != NULL )

        {

                print root -> data;

             preorder (root -> lchild);

                preorder (root -> rchild);

        }

}
```

PostOrder:

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 post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be −

D → E → B → F → G → C → A

```
Algorithm PostOrder

Until all nodes are traversed −

Step 1 − Recursively traverse left subtree.

Step 2 − Recursively traverse right subtree.

Step 3 − Visit root node.

void postorder(node *root)

{

        if( root != NULL )

        {

                postorder (root -> lchild);
                postorder (root -> rchild);

                    print (root -> data);

          }

}
```

Example 1: Traverse the following binary tree in pre, post and inorder

- Preorder traversal yields:
  A, B, D, C, E, G, F, H, I

- Postorder traversal yields:
  D, B, G, E, H, I, F, C, A

- Inorder traversal yields:
  D, B, A, E, G, C, H, F, I

Example 2:

Traverse the following binary tree in pre, post and inoder

- Preorder traversal yields:
  P, F, B, H, G, S, R, Y, T, W, Z
- Postorder traversal yields:
  B, G, H, F, R, W, T, Z, Y, S, P
- Inorder traversal yields:
  B, F, G, H, P, R, S, T, W, Y, Z

Example 3

- Preorder traversal yields:
  2, 7, 2, 6, 5, 11, 5, 9, 4
- Postorder travarsal yields:
  2, 5, 11, 6, 7, 4, 9, 5, 2
- Inorder travarsal yields:
  2, 7, 5, 6, 11, 2, 5, 4, 9

Example 4:

			• Preorder traversal yields: A, B, D, G, K, H, L, M, C, E • Postorder travarsal yields: K, G, L, M, H, D, B, E, C, A • Inorder travarsal yields: K, G, D, L, H, M, B, A, E, C

Building Binary Tree from Traversal Pairs:

Sometimes it is required to construct a binary tree if its traversals are known. From a single traversal it is not possible to construct unique binary tree. However any of the two traversals are given then the corresponding tree can be drawn uniquely:

- Inorder and preorder
- Inorder and postorder

The basic principle for formulation is as follows:

If the preorder traversal is given, then the first node is the root node. If the postorder traversal is given then the last node is the root node. Once the root node is identified, all the nodes in the left sub-trees and right sub-trees of the root node can be identified using inorder.

Same technique can be applied repeatedly to form sub-trees.

It can be noted that, for the purpose mentioned, two traversal are essential out of which one should be inorder traversal and another preorder or postorder; alternatively, given preorder and postorder traversals, binary tree cannot be obtained uniquely.

Example 1:

Construct a binary tree from a given preorder and inorder sequence:

Preorder: A B D G C E H I F

Inorder: D G B A H E I C F

Solution:

- From Preorder sequence A B D G C E H I F, the root is**: A**
- From Inorder sequence D G B A H E I C F, we get the left and right sub trees:
- Left sub tree is: **D G B**
- Right sub tree is: **H E I C F**
- The Binary tree upto this point looks like:

To find the root, left and right sub trees for D G B:

- *From the preorder sequence* ***B*** *D G, the root of tree is: B*
- From the inorder sequence D G **B,** we can find that D and G are to the left of B.
- The Binary tree upto this point looks like:

To find the root, left and right sub trees for D G:

- From the preorder sequence D G, the root of the tree is: D
- From the inorder sequence D G, we can find that there is no left node to D and G is at the right of D.
- The Binary tree upto this point looks like:

To find the root, left and right sub trees for H E I C F:

- *From the preorder sequence* ***C*** E H I F*, the root of the left sub tree is: C*
- From the inorder sequence *H E I* ***C*** *F*, we can find that H E I are at the left of C and F is at the right of C.
- The Binary tree upto this point looks like:

To find the root, left and right sub trees for H E I:

- *From the preorder sequence* **E** H I*, the root of the tree is: E*
- From the inorder sequence *H* ***E*** *I*, we can find that H is at the left of E and I is at the right of E.
- The Binary tree upto this point looks like:

Example 2:

Construct a binary tree from a given postorder and inorder sequence:

Inorder: D G B A H E I C F
Postorder: G D B H I E F C A

Solution:

- *From Postorder sequence* G D B H I E F C **A**, *the root is: A*
- From Inorder sequence *D G B* **A** *H E I C F*, we get the left and right sub trees:
  - *Left sub tree is: D G B*
  - *Right sub tree is: H E I C F*
- The Binary tree upto this point looks like:
-

To find the root, left and right sub trees for D G B:

- *From the postorder sequence G D B, the root of tree is: B*
- From the inorder sequence *D G* **B,** we can find that D G are to the left of B and there is no right subtree for B.
- The Binary tree upto this point looks like:

To find the root, left and right sub trees for D G:

- From the postorder sequence G D, the root of the tree is: D
- From the inorder sequence D G, we can find that is no left subtree for D and G is to the right of D.
- The Binary tree upto this point looks like:

To find the root, left and right sub trees for H E I C F:

- *From the postorder sequence H I E F **C**, the root of the left sub tree is: C*
- From the inorder sequence H E I ***C*** F, we can find that H E I are to the left of C and F is the right subtree for C.
- The Binary tree upto this point looks like:

To find the root, left and right sub trees for H E I:

- *From the postorder sequence H I **E**, the root of the tree is: E*
- From the inorder sequence *H* **E** *I*, we can find that H is left subtree for E and I is to the right of E.
- The Binary tree upto this point looks like:

```
//Binary tree traversals
#include<stdio.h>
typedef struct node

{
    int data;
    struct node *left;
    struct node *right;
}node;
node *create()
{
    node *p;
    int x;
    printf("Enter data(-1 for no data)");
    scanf("%d",&x);
    if(x==-1)
        return NULL;
    p=(node*)malloc(sizeof(node));
    p->data=x;
    printf("Enter left child of %d:n",x);
    p->left=create();
    printf("Enter right child of %d:n",x);
    p->right=create();
    return p;
}
void preorder(node *t)              //address of root node is passed
{
    if(t!=NULL)
    {

      printf("%d",t->data);
      preorder(t->left);
//vi    sit the root
              //preorder traversal on left subtree
    preorder(t->right);         //preorder traversal om right subtree
    }
}
void inOrder(node *t)
{
    if(t!=NULL)
    {
            inOrder(t->left);
            printf("%d ", t->data);
            inOrder(t->right);
    }
}
void postOrder(node *t)
{
    if(t!=NULL)
    {
            inOrder(t->left);
            inOrder(t->right);
            printf("%d ", t->data);
    }
}
int main()
{
    node *root;
    root=create();
    printf("\nThe preorder traversal of tree is:\n");
    preorder(root);
    printf("\nThe inorder traversal of tree is:\n");
    inOrder(root);
    printf("\nThe postorder traversal of tree is:\n");
    postOrder(root);

    return 0;


}
```

OUTPUT:

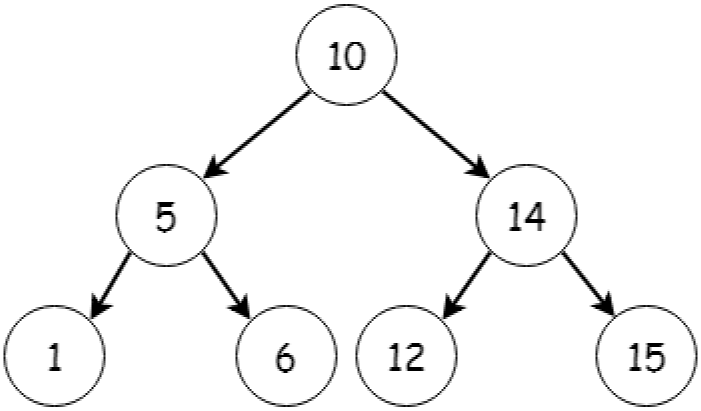
```
Enter data(-1 for no data)1
Enter left child of 1:nEnter data(-1 for no data)2
Enter left child of 2:nEnter data(-1 for no data)4
Enter left child of 4:nEnter data(-1 for no data)-1
Enter right child of 4:nEnter data(-1 for no data)-1
Enter right child of 2:nEnter data(-1 for no data)5
Enter left child of 5:nEnter data(-1 for no data)-1
Enter right child of 5:nEnter data(-1 for no data)-1
Enter right child of 1:nEnter data(-1 for no data)3
Enter left child of 3:nEnter data(-1 for no data)6
Enter left child of 6:nEnter data(-1 for no data)-1
Enter right child of 6:nEnter data(-1 for no data)-1
Enter right child of 3:nEnter data(-1 for no data)7
Enter left child of 7:nEnter data(-1 for no data)-1
Enter right child of 7:nEnter data(-1 for no data)-1

The preorder traversal of tree is:
1 2 4 5 3 6 7
The inorder traversal of tree is:
4 2 5 1 6 3 7
The postorder traversal of tree is:
```

4 2 5 6 3 7 1

Tree traversals using stack:

Iterative preorder traversal

In preorder traversal, root node is processed before left and right subtrees. For example, preorder traversal of below tree would be [10,5,1,6,14,12,15],



We already know how to implement preorder traversal in recursive way, let’s understand how to implement it in non-recursive way.

Iterative preorder traversal : Thoughts

If we look at recursive implementation, we see we process the root node as soon as we reach it and then start with left subtree before touching anything on right subtree.

Once left subtree is processed, control goes to first node in right subtree. To emulate this behavior in non-recursive way, it is best to use a stack. What and when push and pop will happen on the stack?
Start with pushing the root node to stack. Traversal continues till there at least one node onto stack.

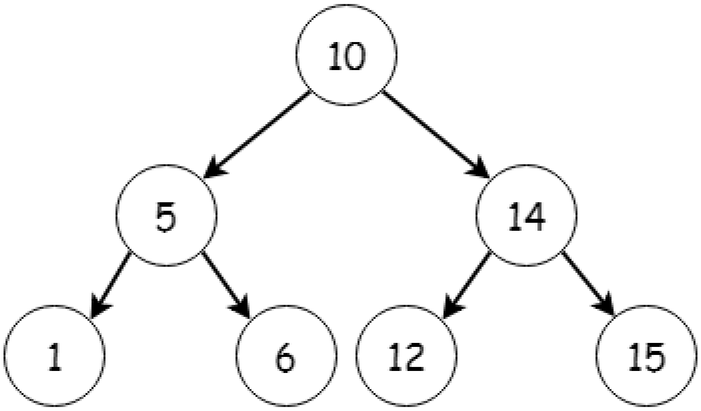
Pop the root node from stack,process it and push it’s right and left child on to stack. Why right child before left child? Because we want to process left subtree before right subtree. As at every node, we push it’s children onto stack, entire left subtree of node will be processed before right child is popped from the stack. Algorithm is very simple and is as follows.

1.

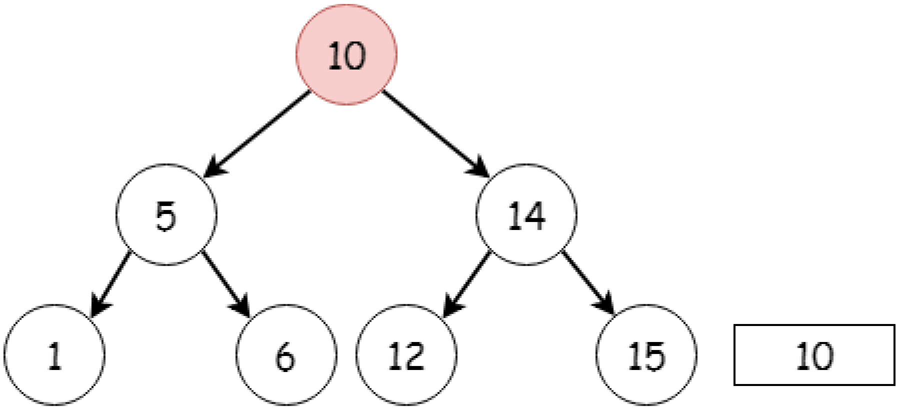
   1. Start with root node and push on to stack s
   2. While there stack is not empty
      1. Pop from stack `current` = s.pop() and process the node.
      2. Push `current.right` onto to stack.
      3. Push `current.left` onto to stack.

Iterative preorder traversal : example

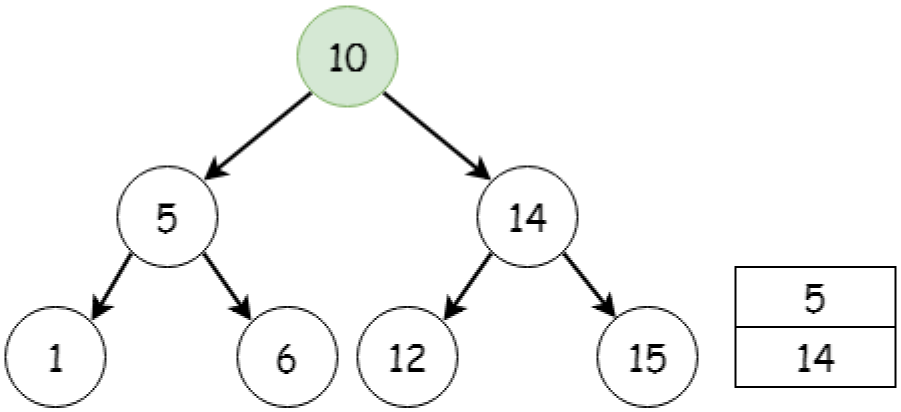
Let’s take and example and see how it works. Given below tree, do preorder traversal on it without recursion.



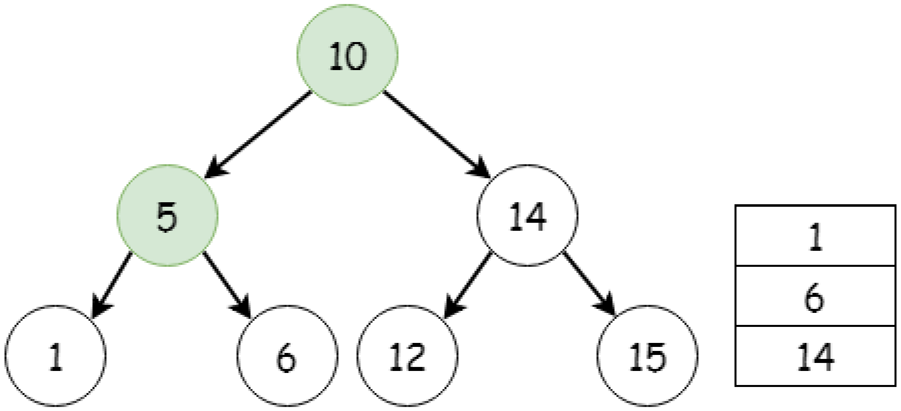
Let’s start from root node(10) and push it onto stack. current = node(10).



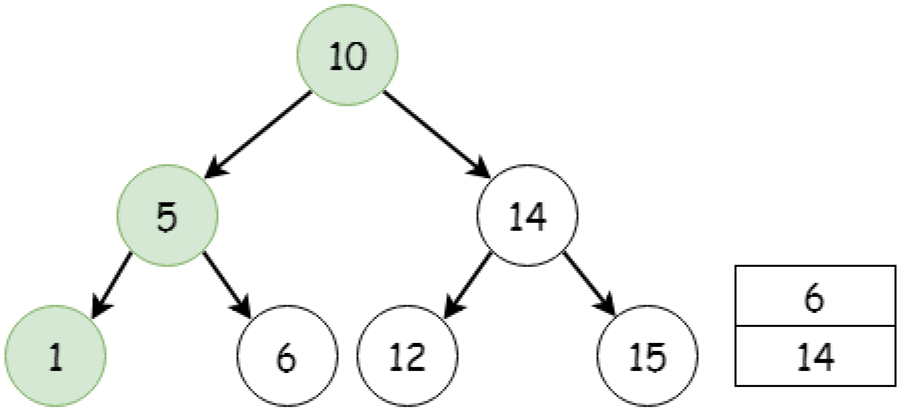
Here loop starts, which check if there is node onto stack. If yes, it pops that out. s.pop will return node(10), we will print it and push it’s right and left child onto stack. Preorder traversal till now : [10].



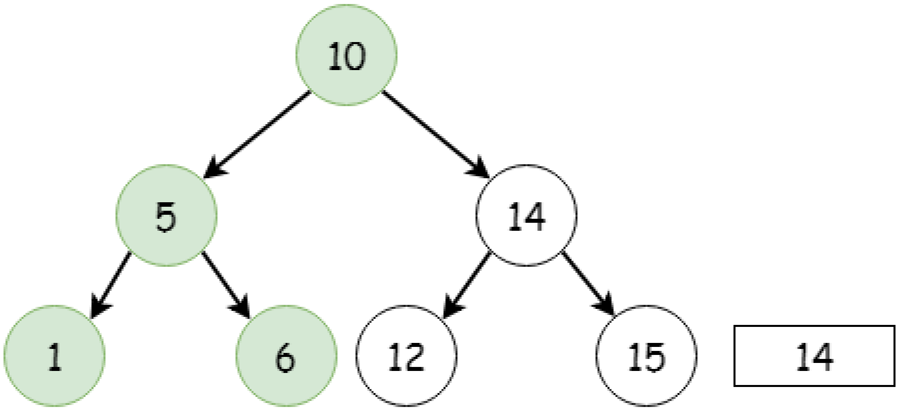
Since stack is not empty, pop from it.current= node(5). Print it and push it’s right and left child i.e node(6) and node(1) on stack.



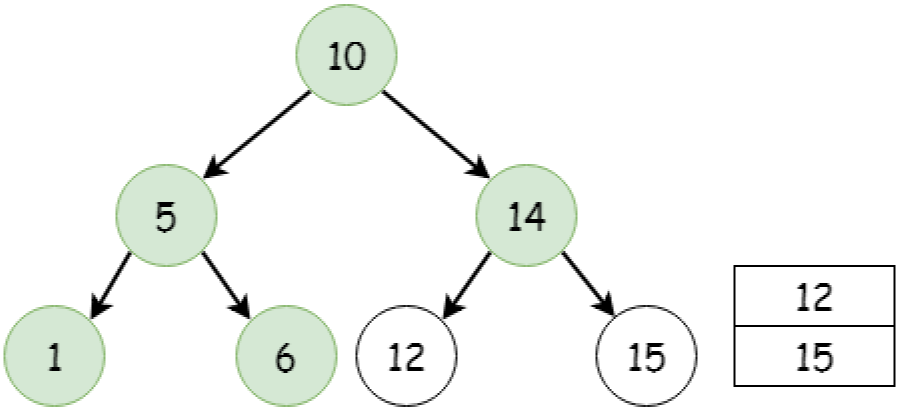
Again, stack is not empty, pop from stack. current = node(1). Print node. There is no right and left child for this node, so we will not push anything on the stack.



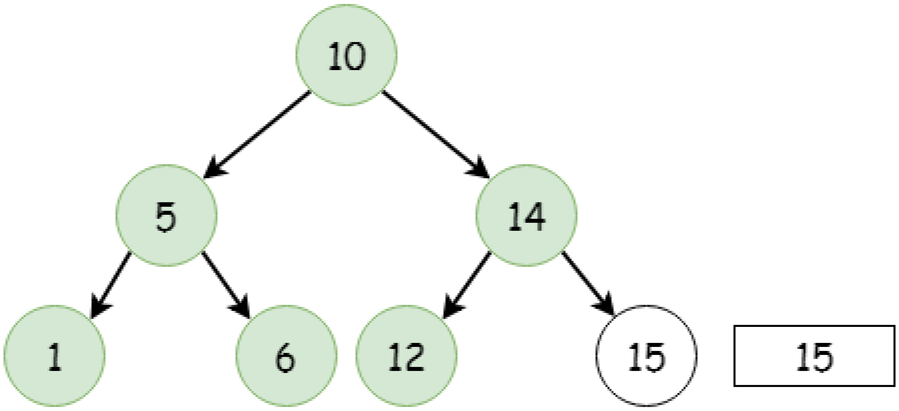
Stack is not empty yet, pop again. current= node(6). Print node. Similar to node(1), it also does not have right or left subtree, so nothing gets pushed onto stack.



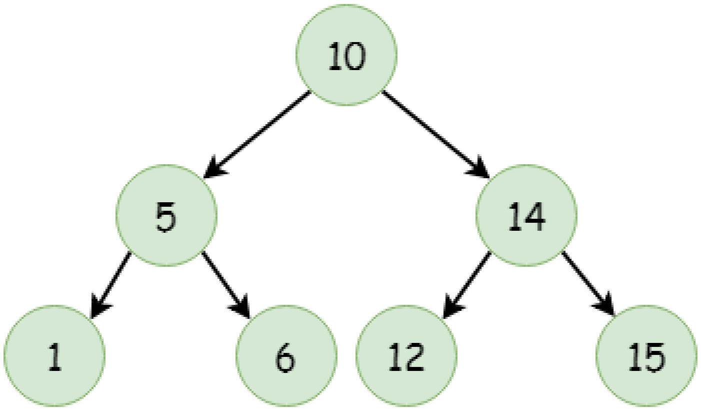
However, stack is not empty yet. Pop. Current = node(14). Print node, and as there are left and right children, push them onto stack as right child before left child.



Stack is not empty, so pop from stack, current = node(12). Print it, as there are no children of node(12), push nothing to stack.



Pop again from stack as it not empty. current = node(15). Print it. No children, so no need to push anything.

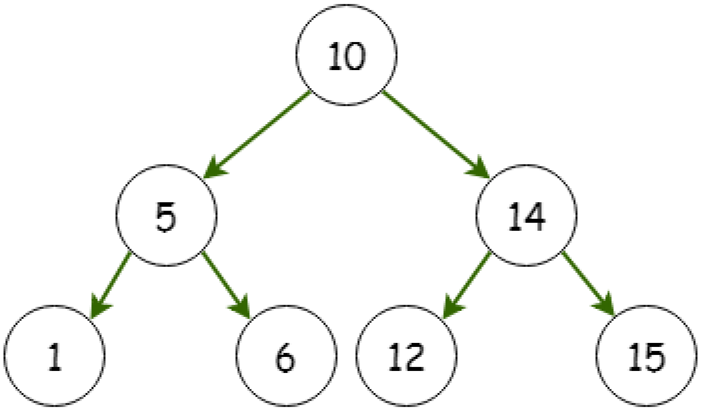


At this point, stack becomes empty and we have traversed all node of tree also.

Iterative Inorder traversal

One of the most common things we do on binary tree is traversal. In Binary search tree traversals we discussed different types of traversals like inorder, preorder and postorder traversals. We implemented those traversals in recursive way. In this post, let’s focus on iterative implementation of inorder traversal or iterative inorder traversal without recursion.

Before solution, what is inorder traversal of binary tree? In inorder traversal, visit left subtree, then root and at last right subtree. For example, for given tree, inorder traversal would be: [1,5,6,10,12,14,15]



Iterative inorder traversal without stack : Thoughts

As we go into discussion, one quick question : why recursive inorder implementation is not that great? We know that recursion uses implicitly stack to store return address and passed parameters. As recursion goes deep, there will be more return addresses and

parameters stored on stack, eventually filling up all the space system has for stack. This problem is known as stack overflow.
When binary tree is skewed, that is when every node has only one child, recursive implementation may lead to stack overflow, depending on the size of tree. In production systems, we usually do not know upfront size of data structures, it is advised to avoid recursive implementations.

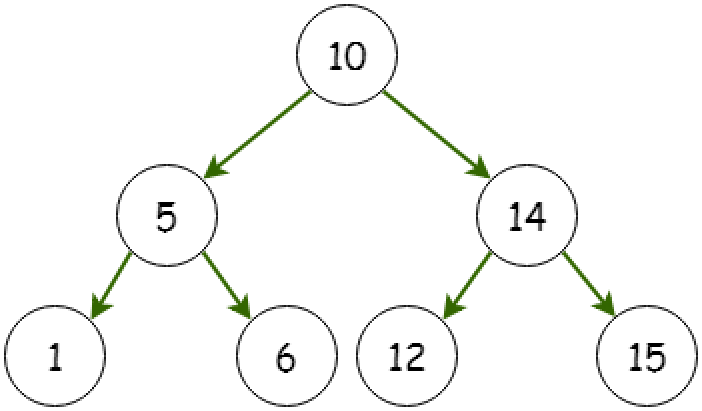
What are we essentially doing in recursive implementation? We check if node is null, then return. If not, we move down the left subtree. When there is nothing on left subtree, we move up to parent, and then go to right subtree.

All these steps are easy to translate in iterative way. One thing needs to be thought of is : how to go to parent node? In inorder traversal, the last node visited before current node is the parent node.
If we keep these nodes on some structure, where we can refer them back, things will be easy. As we refer the most recent node added to structure first (when finding parent of node, we have to just look at the last visited node), stack is great candidate for it which has last in first out property.

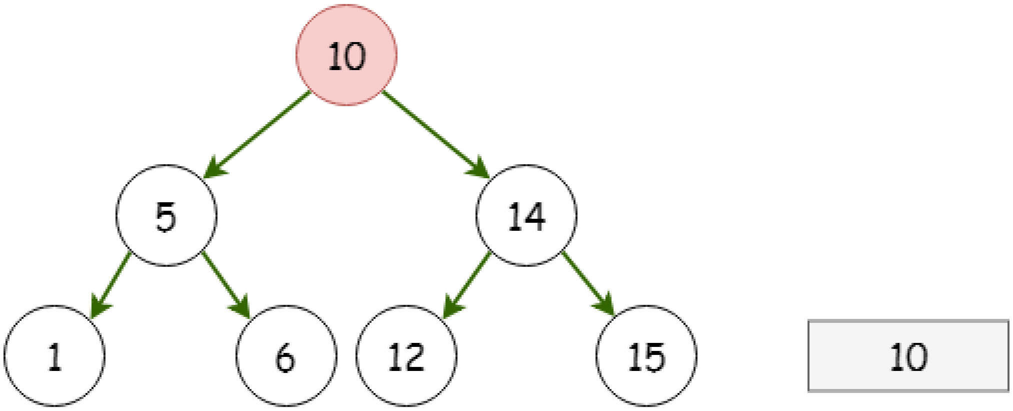
Iterative inorder traversal : algorithm

1. Start from the root, call it current .
2. If current is not NULL, push current on to stack.
3. Move to left child of current and go to step 2.
4. If `current` == NULL and !stack.empty(), `current` = s.pop.
5. Process current and set `current` = `current.right`, go to step 2.

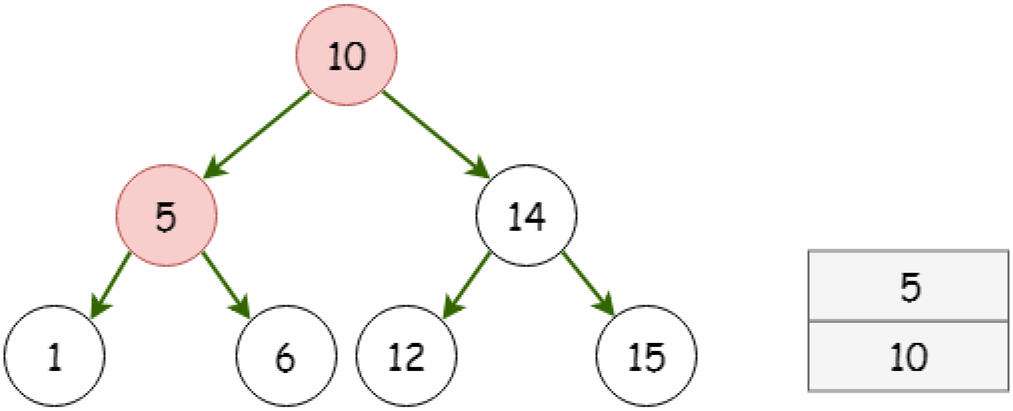
Let's take an example and see how this algorithm works.



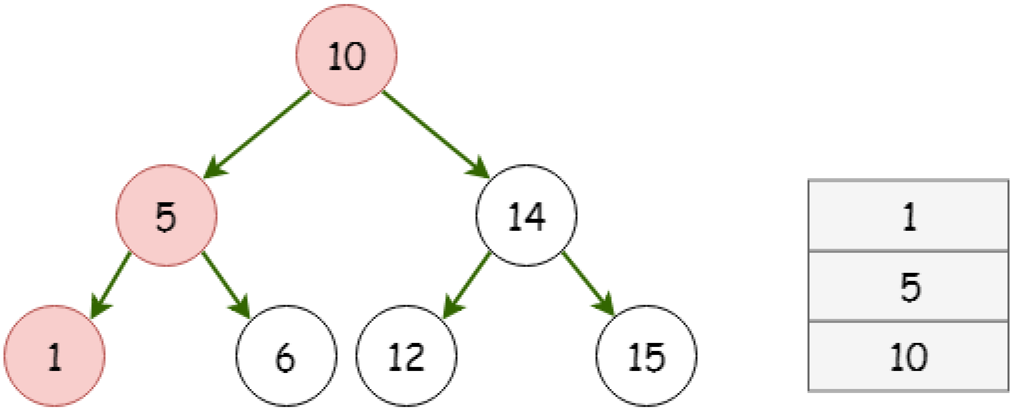
We start with node(10), current = node(10). Current node is not null, put it on stack.



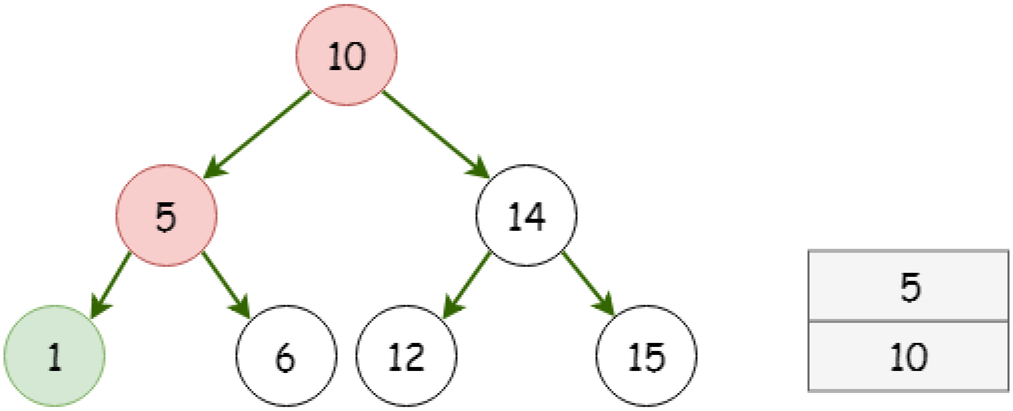
As there is left child of node(10), move current = current.left, so current = node(5), which is not null, put node on to stack.



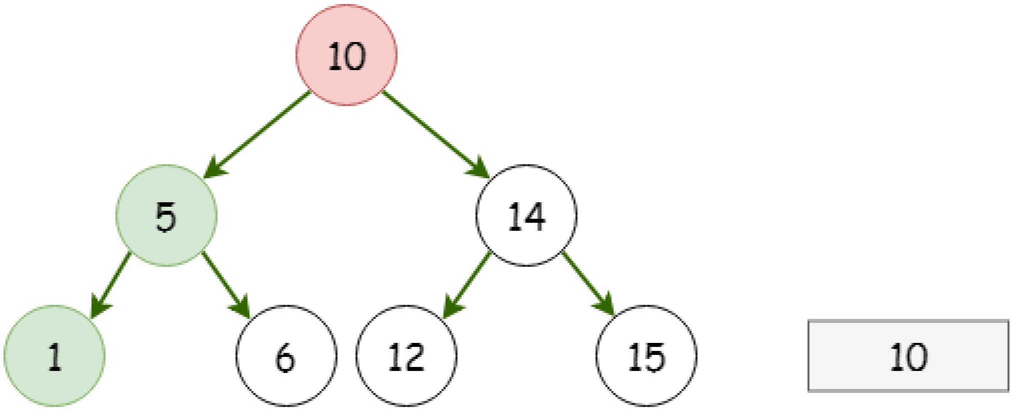
Again, move down to left child of node(5), current = current.left = node(1). Put the node on to stack.



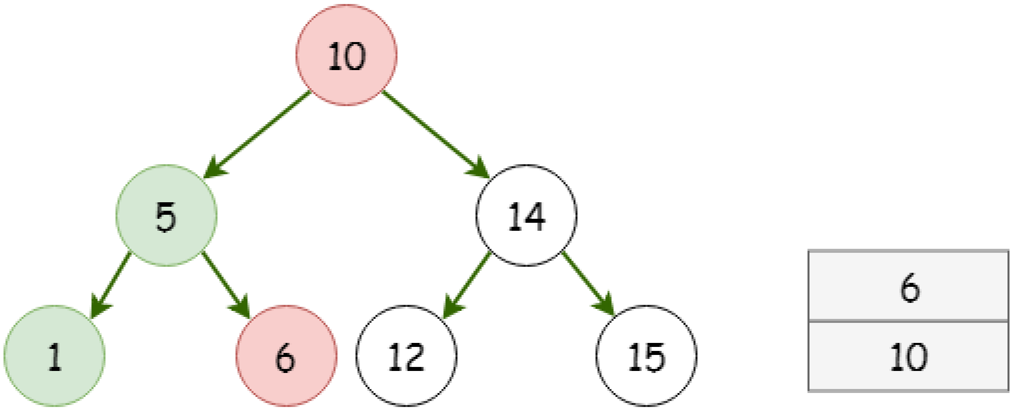
Again move down to left child, which in this case it is null. What to do now? As stack is not empty, pop last node added to it. current = node(1). Process node(1). Traversal = [1]



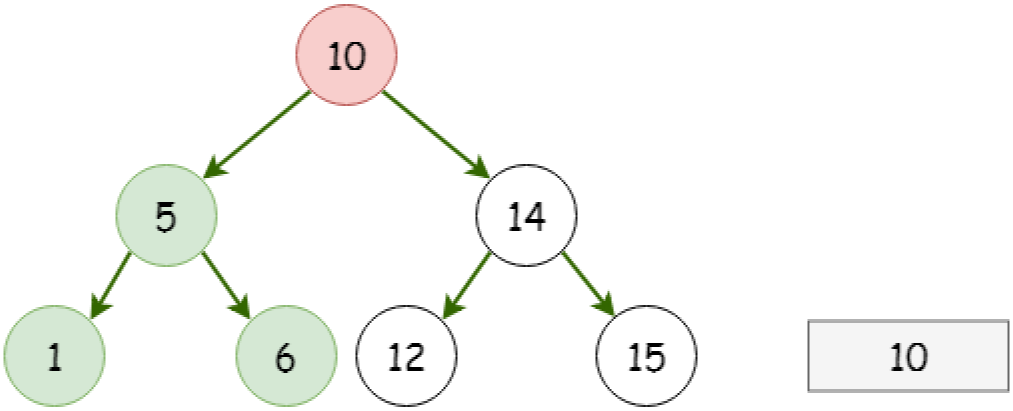
Move to right child of node(1), which is null, in that case pop from the stack and process the node, current = node(5). Traversal = [1,5]



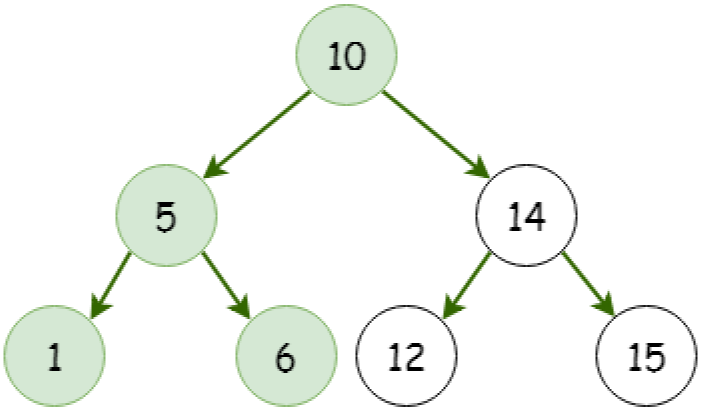
Move to the right child of node(5) i.e. node(6). Push on to the stack.



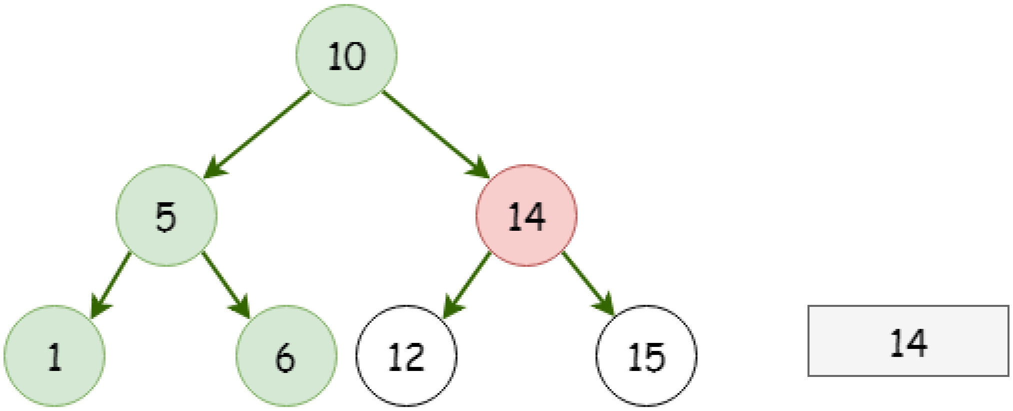
Move down to left subtree, which is null, so pop from stack. current = node(6), process it. Traversal = [1,5,6]



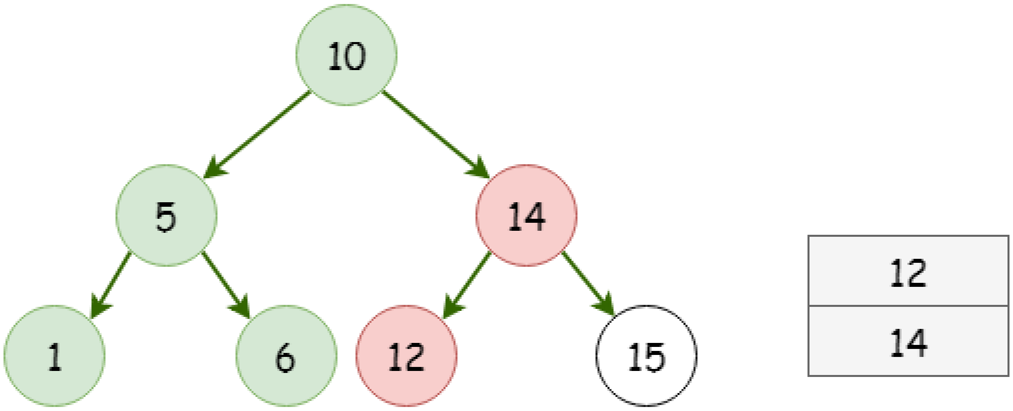
Move to right child of node(6), which is null, so pop from stack current = node(10). Process the node. Traversal = [1,5, 6,10]



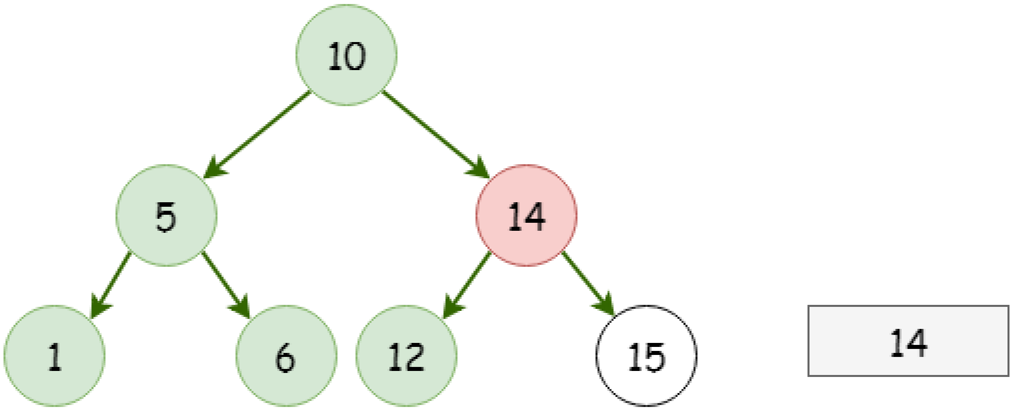
Get right child of node(10), which is node(14), current = node(14), as current is not null, put it on to stack.



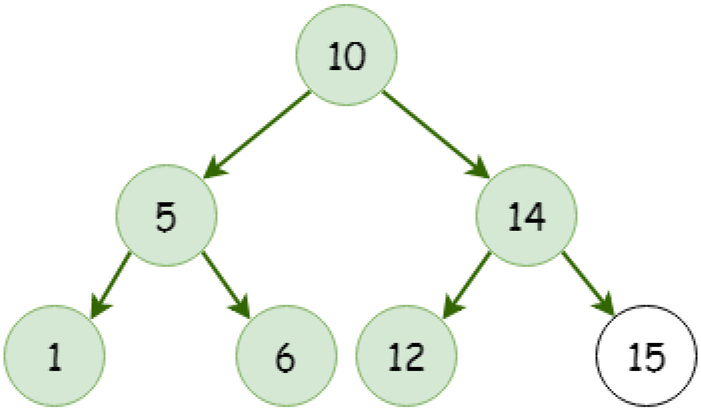
Again move to left child of current node (14), which is node(12). current = node(12) which is not null, put it onto stack.



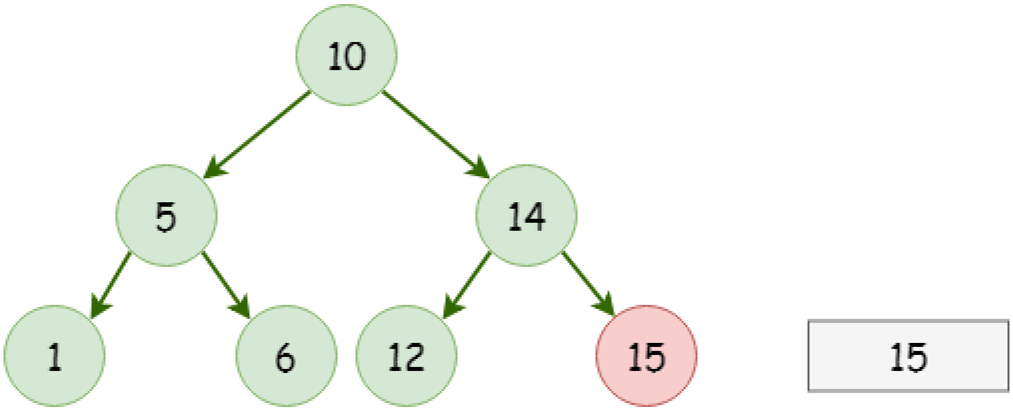
Get left child of current node, which is null. So pop from stack, current = node(12). Process it. Traversal = [1,5,6,10,12]



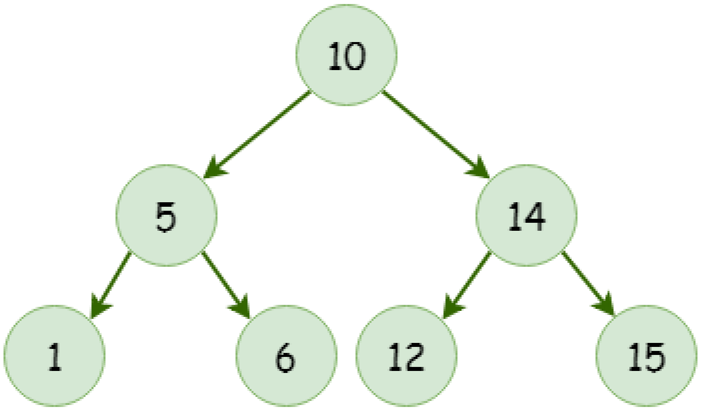
Current node = current.right, i.e null, so pop out of stack. current = node(14). Process node(14). Traversal = [1,5,6,10,12,14]



Again current = current.right which is node(15). Put it back on to stack.

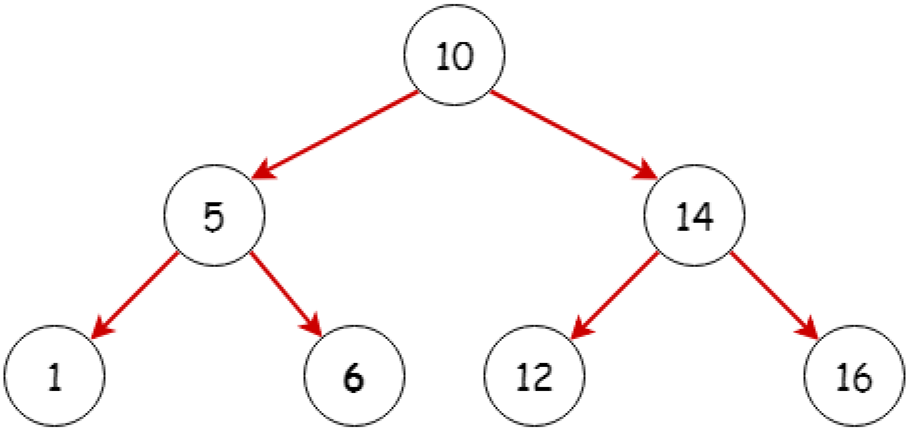


Left child of node(15) is null, so we pop from stack. current = node(15). Process node(15). Fetch right child of current node which is again null and this time even stack is already empty. So stop processing and everything is done. Traversal = [1,5,6,10,12,14,15]



Iterative postorder traversal

Iterative postorder traversal of binary tree which is most complex of all traversals. A traversal where left and right subtrees are visited before root is processed. For example, post order traversal of below tree would be : [1,6,5,12,16,14,10]



Iterative postorder traversal : Thoughts

Let’s look at the recursive implementation of postorder.

```
private void postOrder(Node root){
    if(root == null) return;

    postOrder(root.left);
    postOrder(root.right);
    System.out.println(root.value);

}
```

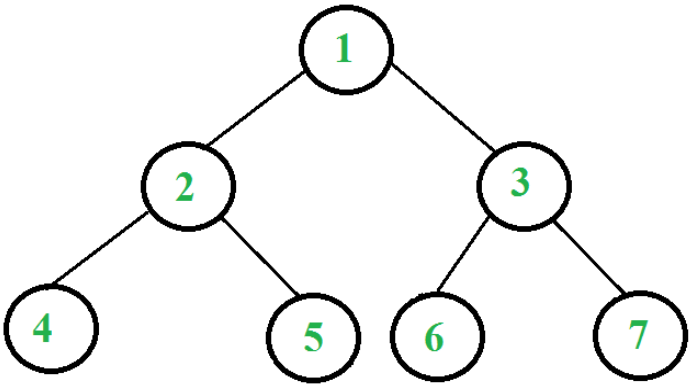
As we are going into left subtree and then directly to right subtree, without visiting root node. Can you find the similarity of structure between preorder and postorder implementation? Can we reverse the entire preorder traversal to get post order traversal? Reverse preorder will give us right child, left child and then root node, however order expected is left child, right child and root child.
Do you remember we pushed left and right node onto stack in order where right child went before left. How about reversing that?

There is one more problem with just reversing the preorder. In preorder, a node was processed as soon as popped from stack, like root node will be the first node to be processed. However, in postorder, root node is processed last. So, we actually need the order of processing too be reversed. What better than using a stack to store the reverse order of root nodes to processed.
All in all, we will be using two stacks, one to store left and right child, second to store processing order of nodes.

1. Create two stacks s an out and push root node onto s
2. While stack s is not empty
   1. op from stack s, `current` = s.pop

2. Put `current` onto stack out.
3. Put left and right child of `current` on to stack s

3. Pop everything from out stack and process it.
    4. Let us consider the following tree



5. Following are the steps to print postorder traversal of the above tree using one stack.

```
1. Right child of 1 exists.
   Push 3 to stack. Push 1 to stack. Move to left child.
        Stack: 3, 1

2. Right child of 2 exists.
   Push 5 to stack. Push 2 to stack. Move to left child.
        Stack: 3, 1, 5, 2

3. Right child of 4 doesn't exist. '
   Push 4 to stack. Move to left child.
        Stack: 3, 1, 5, 2, 4

4. Current node is NULL.
   Pop 4 from stack. Right child of 4 doesn't exist.
   Print 4. Set current node to NULL.
        Stack: 3, 1, 5, 2

5. Current node is NULL.
    Pop 2 from stack. Since right child of 2 equals stack top element,
    pop 5 from stack. Now push 2 to stack.
    Move current node to right child of 2 i.e. 5
        Stack: 3, 1, 2

6. Right child of 5 doesn't exist. Push 5 to stack. Move to left child.
        Stack: 3, 1, 2, 5

7. Current node is NULL. Pop 5 from stack. Right child of 5 doesn't exist.
```

```
   Print 5. Set current node to NULL.
        Stack: 3, 1, 2

8. Current node is NULL. Pop 2 from stack.
   Right child of 2 is not equal to stack top element.
   Print 2. Set current node to NULL.
        Stack: 3, 1

9. Current node is NULL. Pop 1 from stack.
   Since right child of 1 equals stack top element, pop 3 from stack.
   Now push 1 to stack. Move current node to right child of 1 i.e. 3
        Stack: 1

10. Repeat the same as above steps and Print 6, 7 and 3.
    Pop 1 and Print 1.
```

Introduction to Graphs:

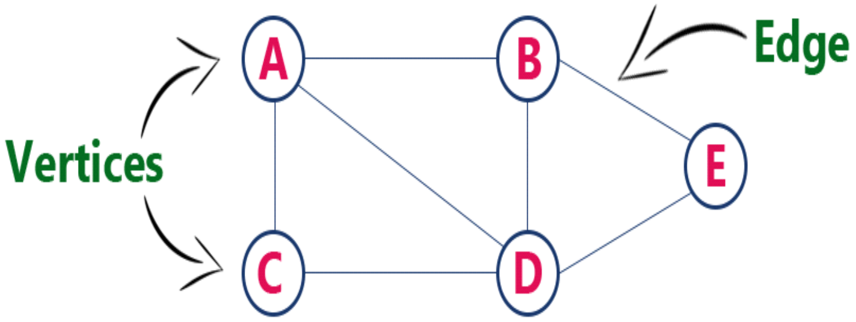
- Graph is a collection of nodes and edges which connects nodes in the graph.
- Generally, a graph G is represented as G = (V , E), where V is set of vertices and E is set of edges.

Example:

The following is a graph with 5 vertices and 6 edges.

This graph G can be defined as G = (V , E)

Where V = {A,B,C,D,E} and E = {(A,B),(A,C)(A,D),(B,D),(C,D),(B,E),(E,D)}.



Graph Terminology:

We use the following terms in graph data structure.

Vertex

- A individual data element of a graph is called as Vertex.
- Vertex is also known as node.
- In above example graph, A, B, C, D & E are known as vertices.

Edge

- An edge is a connecting link between two vertices.
- Edge is also known as Arc.
- An edge is represented as (startingVertex, endingVertex).
- For example, in above graph, the link between vertices A and B is represented as (A,B).
- In above example graph, there are 7 edges (i.e., (A,B), (A,C), (A,D), (B,D), (B,E), (C,D), (D,E)).

Edges are three types:

- **Undirected Edge** - An undirected egde is a bidirectional edge. If there is a undirected edge between vertices A and B then edge (A , B) is equal to edge (B , A).
- **Directed Edge** - A directed egde is a unidirectional edge. If there is a directed edge between vertices A and B then edge (A , B) is not equal to edge (B , A).
- **Weighted Edge** - A weighted egde is an edge with cost on it.

Undirected Graph:

A graph with only undirected edges is said to be undirected graph.

Directed Graph:

A graph with only directed edges is said to be directed graph.

Mixed Graph:

A graph with undirected and directed edges is said to be mixed graph.

End vertices or Endpoints:

The two vertices joined by an edge are called the end vertices (or endpoints) of the edge.

Origin:

If an edge is directed, its first endpoint is said to be origin of it.

Destination:

If an edge is directed, its first endpoint is said to be origin of it and the other endpoint is said to be the destination of the edge.

Adjacent:

If there is an edge between vertices A and B then both A and B are said to be adjacent. In other words, Two vertices A and B are said to be adjacent if there is an edge whose end vertices are A and B.

Incident:

An edge is said to be incident on a vertex if the vertex is one of the endpoints of that edge.

Outgoing Edge:

A directed edge is said to be outgoing edge on its orign vertex.

Incoming Edge:

A directed edge is said to be incoming edge on its destination vertex.

Degree:

Total number of edges connected to a vertex is said to be degree of that vertex.

Indegree:

Total number of incoming edges connected to a vertex is said to be indegree of that vertex.

Outdegree:

Total number of outgoing edges connected to a vertex is said to be outdegree of that vertex.

Parallel edges or Multiple edges:

If there are two undirected edges to have the same end vertices, and for two directed edges to have the same origin and the same destination. Such edges are called parallel edges or multiple edges.

Self-loop:

An edge (undirected or directed) is a self-loop if its two endpoints coincide.

Simple Graph:

A graph is said to be simple if there are no parallel and self-loop edges.

Path:

A path is a sequence of alternating vertices and edges that starts at a vertex and ends at a vertex such that each edge is incident to its predecessor and successor vertex.

Representation of Graphs:

There are two ways of representing digraphs. They are:

- Adjacency matrix.
- Adjacency List.
- Incidence matrix.

Adjacency Matrix:

In this representation, the adjacency matrix of a graph G is a two dimensional n x n matrix, say A = (ai,j), where

a i, j = **1 if there is an edge from vi to v j**

= 0 otherwise

The matrix is symmetric in case of undirected graph, while it may be asymmetric if the graph is directed.

This matrix is also called as Boolean matrix or bit matrix.

In this representation, graph can be represented using a matrix of size total number of vertices by total number of vertices.

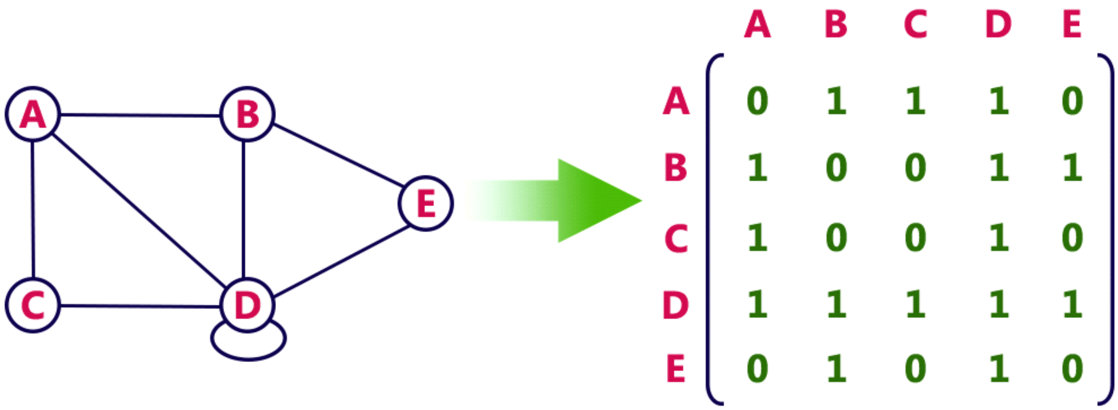
That means if a graph with 4 vertices can be represented using a matrix of 4X4 class.

In this matrix, rows and columns both represents vertices.

This matrix is filled with either 1 or 0.

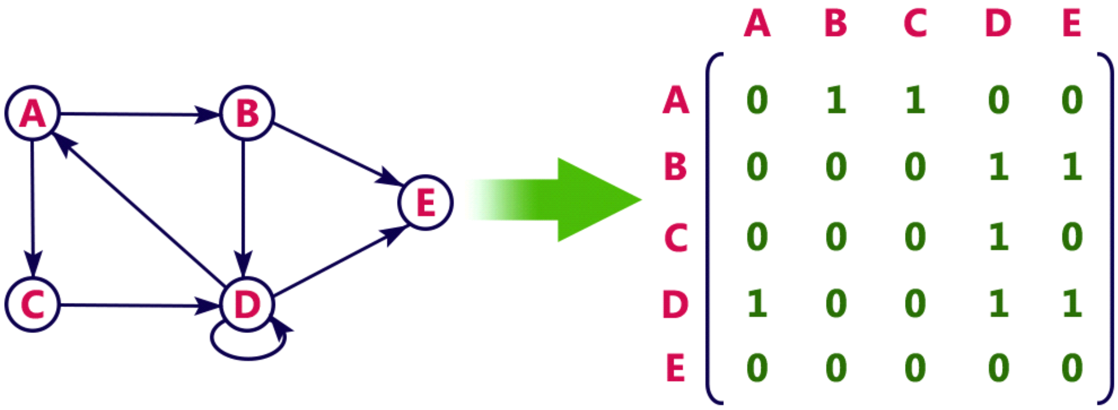
Here, 1 represents there is an edge from row vertex to column vertex and 0 represents there is no edge from row vertex to column vertex.

For example, consider the following undirected graph representation...



$$\begin{array}{c|ccccc}
 & A & B & C & D & E \\
\hline
A & 0 & 1 & 1 & 1 & 0 \\
B & 1 & 0 & 0 & 1 & 1 \\
C & 1 & 0 & 0 & 1 & 0 \\
D & 1 & 1 & 1 & 1 & 1 \\
E & 0 & 1 & 0 & 1 & 0
\end{array}$$

Directed graph representation...



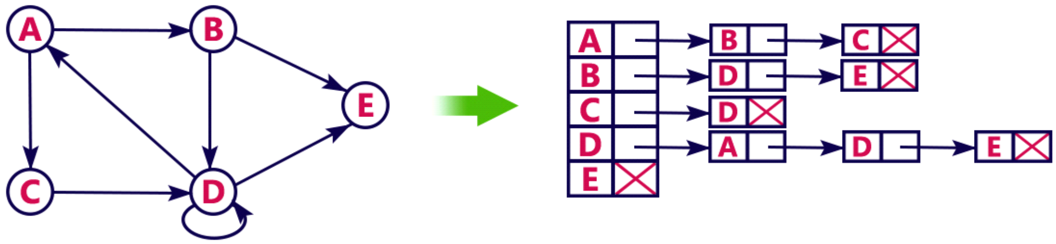
$$\begin{array}{c|ccccc}
 & A & B & C & D & E \\
\hline
A & 0 & 1 & 1 & 0 & 0 \\
B & 0 & 0 & 0 & 1 & 1 \\
C & 0 & 0 & 0 & 1 & 0 \\
D & 1 & 0 & 0 & 1 & 1 \\
E & 0 & 0 & 0 & 0 & 0
\end{array}$$

In case of weighted graph, the entries are weights of the edges between the vertices. The adjacency matrix for a weighted graph is called as cost adjacency matrix.

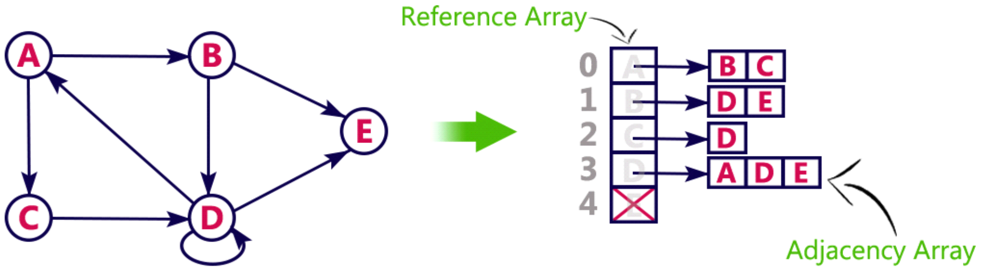
<u>Adjacency List</u>:

In this representation, every vertex of graph contains list of its adjacent vertices.

For example, consider the following directed graph representation implemented using linked list...



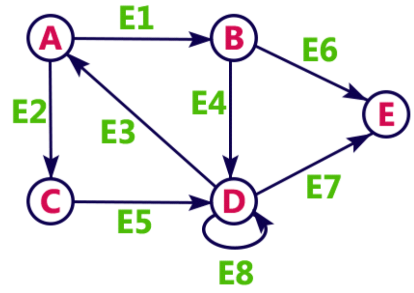
This representation can also be implemented using array as follows.



<u>Incidence Matrix:</u>

- In this representation, graph can be represented using a matrix of size total number of vertices by total number of edges.
- That means if a graph with 4 vertices and 6 edges can be represented using a matrix of 4X6 class.
- In this matrix, rows represents vertices and columns represents edges.
- This matrix is filled with either 0 or 1 or -1.
- Here, 0 represents row edge is not connected to column vertex
- 1 represents row edge is connected as outgoing edge to column vertex
- -1 represents row edge is connected as incoming edge to column vertex.

For example, consider the following directed graph representation...



$$\begin{array}{cc} & \begin{array}{cccccccc} E1 & E2 & E3 & E4 & E5 & E6 & E7 & E8 \end{array} \\ \begin{array}{c} A \\ B \\ C \\ D \\ E \end{array} & \left(\begin{array}{cccccccc} 1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & -1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & -1 & -1 & 0 \end{array}\right) \end{array}$$

	a b		c	d	e	f	g	h	i	j	k	l
A	1	0	0	0	0	0	1	0	0	0	0	0
B	1	1	1	0	0	0	0	0	0	0	0	0
C	0	1	0	1	0	0	1	1	0	0	1	0
D	0	0	1	1	1	1	0	0	0	0	0	0
E	0	0	0	0	1	0	0	1	1	1	0	0
F	0	0	0	0	0	0	0	0	0	1	1	1
G	0	0	0	0	0	1	0	0	1	0	0	1

(b)

Undirected Graph

```
// Write a C program to represent graphs in adjacency matrix

#include<stdio.h>

void creategraph(int G[][20],int n)

{

        int sourse,dest,i,max_edges,type,j;

        printf("\nenter graph type 1 for Undirected graph and 2 for Directed graph");

        scanf("%d",&type);

        if(type==1)

         max_edges= n*(n-1)/2;

        else

         max_edges=n*(n-1);

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

```
    {
        printf("\nenter edge %d  (enter -1 and -1 to terminate)",i);
        scanf("%d%d",&sourse,&dest);
        if(sourse==-1 || dest==-1)
          break;
        if(sourse<0 || sourse>=n || dest<0 || dest>=n)
        {
            printf("\nInvalid edge");
            i--;
        }
        else
        {
          G[sourse][dest]=1;
          if(type==1)
           G[dest][sourse]=1;
       }
    }
}
void display(int G[][20],int n)
{
    int i,j;
    printf("\nADJACENCY MATRIX\n");
    for(i=0;i<n;i++)
    {
        for(j=0;j<n;j++)
```

```
                 printf("\t%d",G[i][j]);

                printf("\n");

        }

}

main()

{

        int n, G[20][20],i,j;

        printf("enter number of vertices");

   scanf("%d",&n);

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

        {

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

                 G[i][j]=0;

   }

        creategraph(G,n);

        display(G,n);

}
```

OUTPUT:

```
enter number of vertices4


enter graph type 1 for Undirected graph and 2 for Directed graph1


enter edge 1  (enter -1 and -1 to terminate)0 1
enter edge 2  (enter -1 and -1 to terminate)0 2


enter edge 3  (enter -1 and -1 to terminate)2 3


enter edge 4  (enter -1 and -1 to terminate)1 3


enter edge 5  (enter -1 and -1 to terminate)-1 -1


ADJACENCY MATRIX
      0     1     1     0
      1     0     0     1
      1     0     0     1
      0     1     1     0
```

Graph Traversals:

- Graph traversal is technique used for searching a vertex in a graph.
- The graph traversal is also used to decide the order of vertices to be visit in the search process.
- A graph traversal finds the edges to be used in the search process without creating loops that means using graph traversal we visit all vertices of graph without getting into looping path.

There are two graph traversal techniques and they are as follows...

- DFS (Depth First Search)
- BFS (Breadth First Search)
- DFS (Depth First Search)

1. **DFS (Depth First Search):**

- DFS traversal of a graph, produces a spanning tree as final result.
- Spanning Tree is a graph without any loops.
- We use Stack data structure with maximum size of total number of vertices in the graph to implement DFS traversal of a graph.

We use the following steps to implement DFS traversal...

Step 1: Define a Stack of size total number of vertices in the graph.

Step 2: Select any vertex as starting point for traversal. Visit that vertex and push it on to the Stack.

Step 3: Visit any one of the adjacent vertex of the vertex which is at top of the stack which is not visited and push it on to the stack.

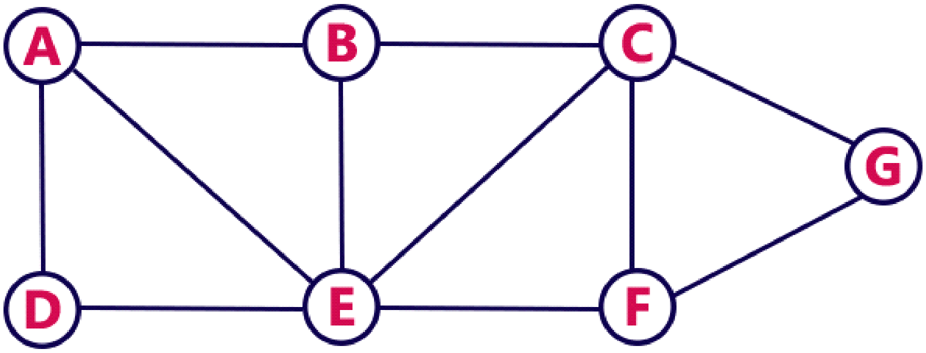
Step 4: Repeat step 3 until there are no new vertex to be visit from the vertex on top of the stack.

Step 5: When there is no new vertex to be visit then use back tracking and pop one vertex from the stack.

Step 6: Repeat steps 3, 4 and 5 until stack becomes Empty.

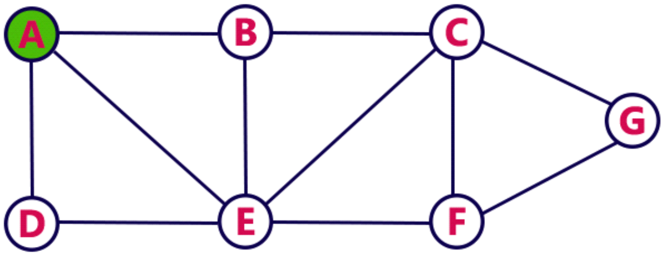
Step 7: When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

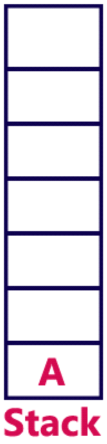
Example:



Step 1:

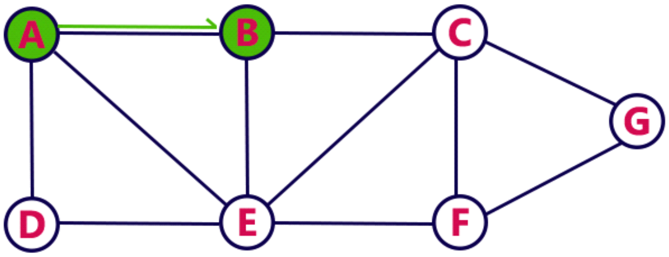
- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.

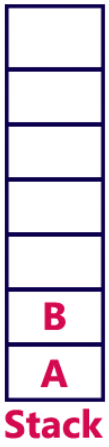




Step 2:

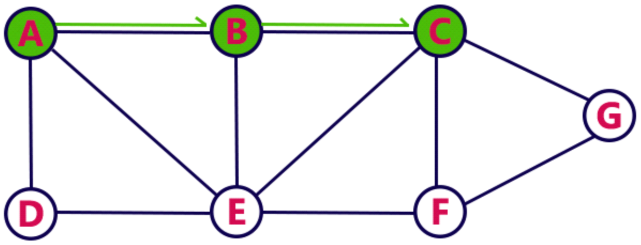
- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.

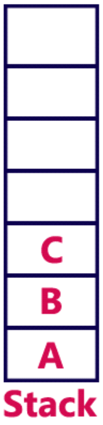




Step 3:

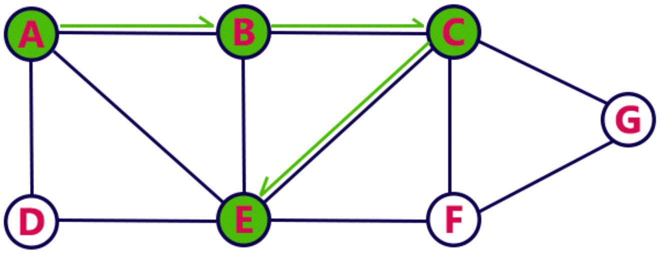
- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.

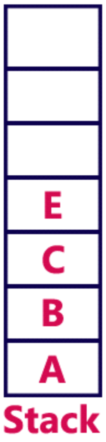




Step 4:

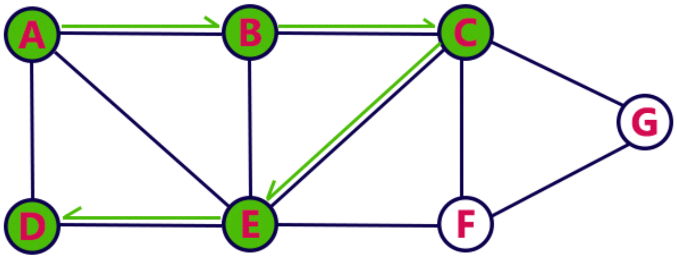
- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack

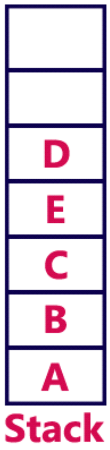




Step 5:

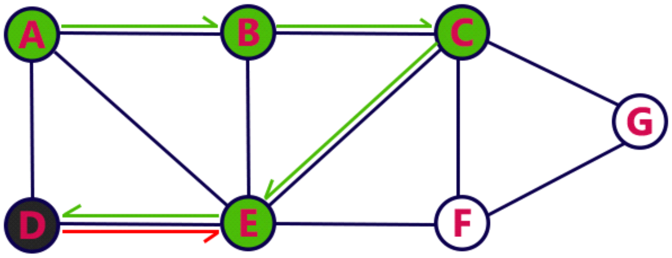
- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack





Step 6:

- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.



E

C

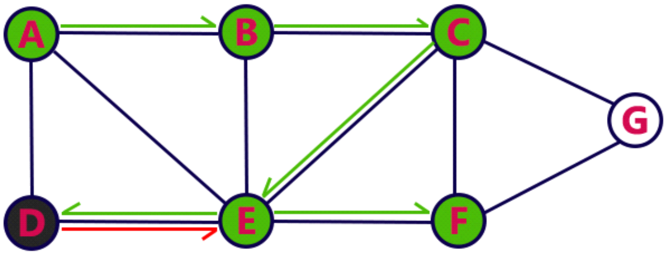
B

A

Stack

Step 7:

- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.



F

E

C

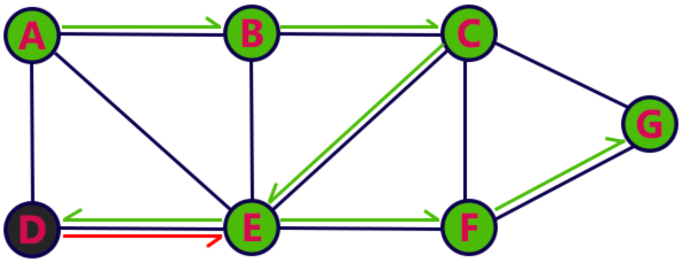
B

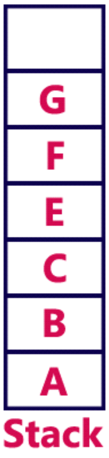
A

Stack

Step 8:

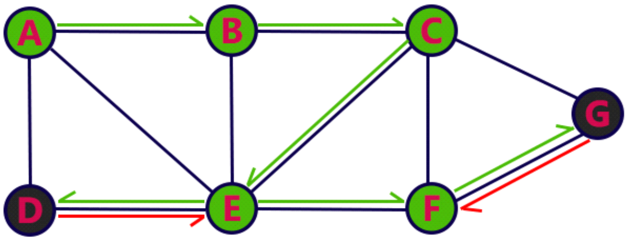
- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.

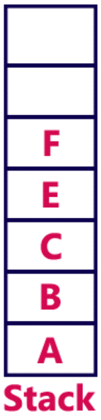




Step 9:

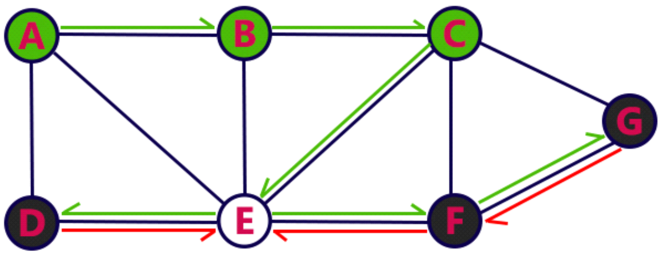
- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.

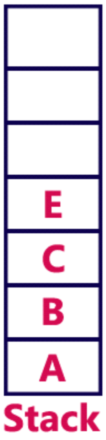




Step 10:

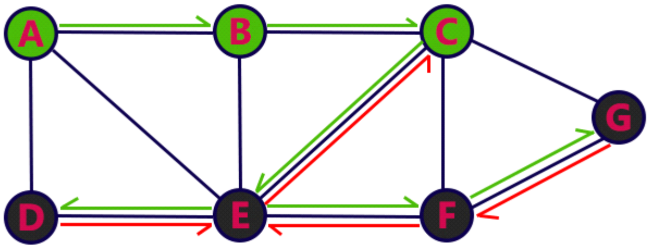
- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.

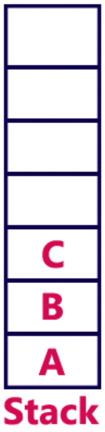




Step 11:

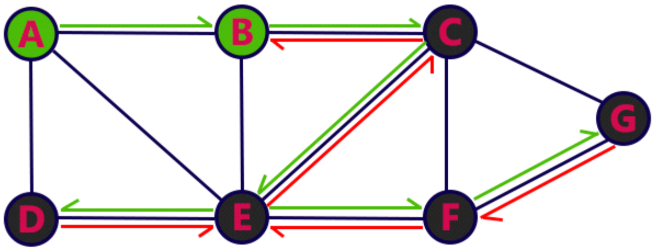
- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.

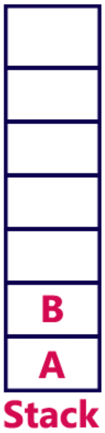




Step 12:

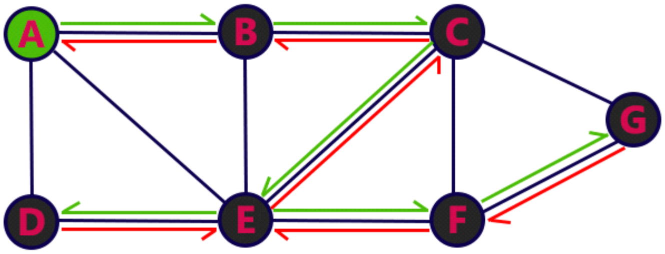
- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.

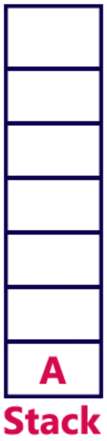




Step 13:

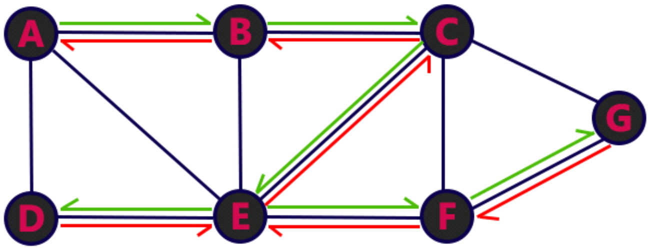
- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.

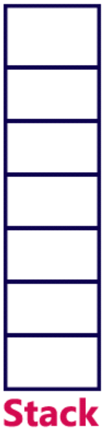




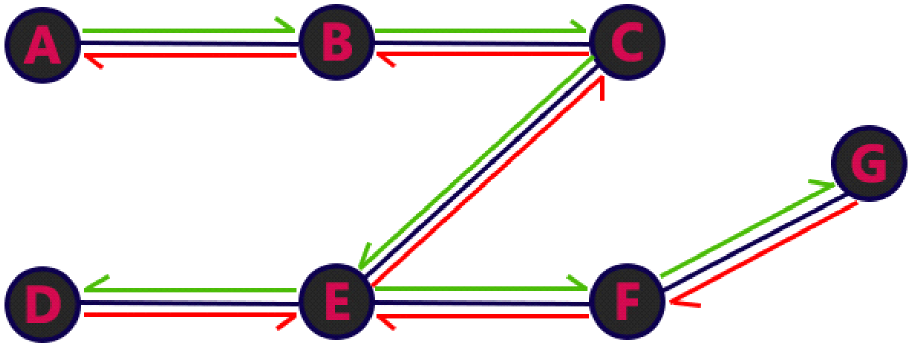
Step 14:

- There is no new vertiex to be visited from A. So use back track.
- Pop A from the Stack.





- Stack became Empty. So stop DFS Treversal.
- Final result of DFS traversal is following spanning tree.



BFS (Breadth First Search):

BFS traversal of a graph, produces a spanning tree as final result.

Spanning Tree is a graph without any loops.

We use Queue data structure with maximum size of total number of vertices in the graph to implement BFS traversal of a graph.

We use the following steps to implement BFS traversal...

Step 1: Define a Queue of size total number of vertices in the graph.

Step 2: Select any vertex as starting point for traversal. Visit that vertex and insert it into the Queue.

Step 3: Visit all the adjacent vertices of the verex which is at front of the Queue which is not visited and insert them into the Queue.

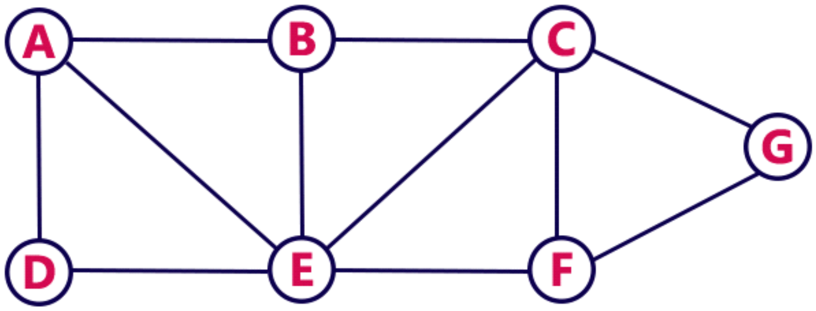
Step 4: When there is no new vertex to be visit from the vertex at front of the Queue then delete that vertex from the Queue.

Step 5: Repeat step 3 and 4 until queue becomes empty.

Step 6: When queue becomes Empty, then produce final spanning tree by removing unused edges from the graph

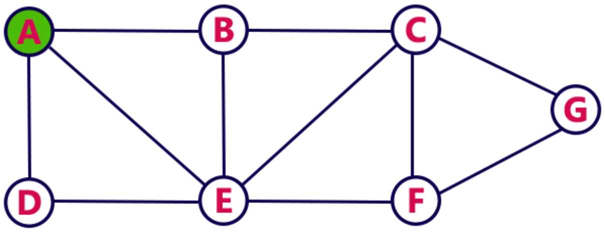
Example:

Consider the following example graph to perform BFS traversal



Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.

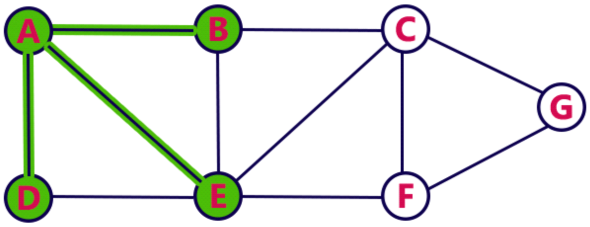


Queue

A						

Step 2:

- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..

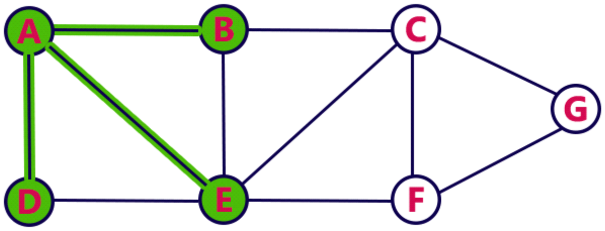


Queue

	D	E	B			

Step 3:

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.

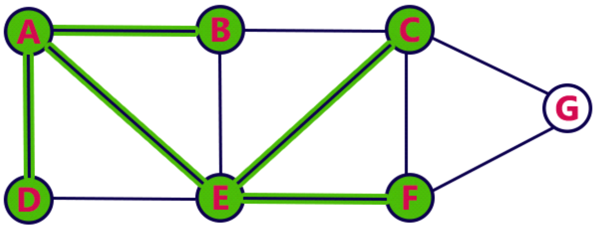


Queue

		E	B			

Step 4:

- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.

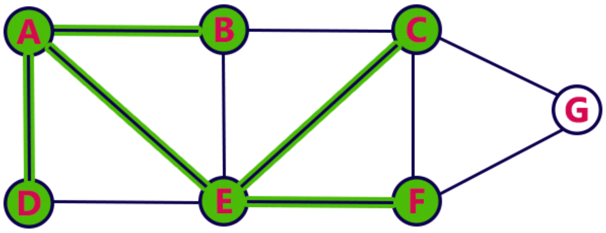


Queue

			B	C	F	

Step 5:

- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.

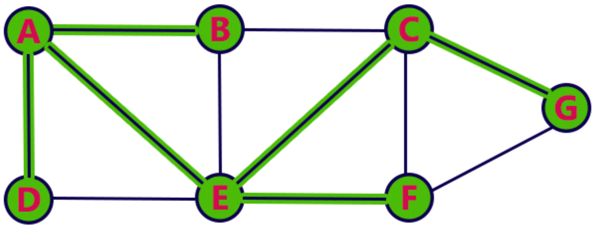


Queue

				C	F	

Step 6:

- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.

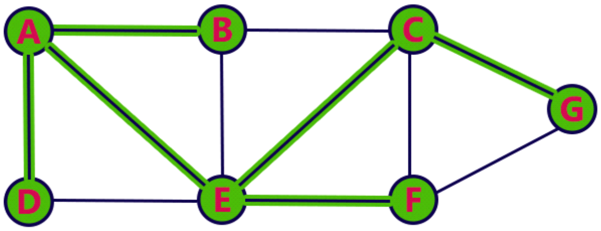


Queue

					F	G

Step 7:

- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.

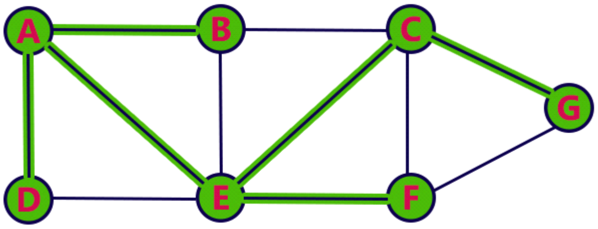


Queue

						G

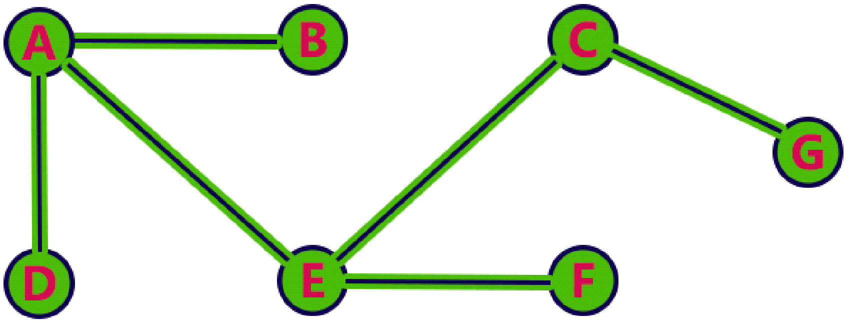
Step 8:

- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



Queue

- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...



Example 2:

Traverse the graph shown below in breadth first order, depth first order and construct the breadth first and depth first spanning trees.

Node	***Adjacency List***
A	F, B, C, G
B	A
C	A, G
D	E, F
E	G, D, F
F	A, E, D
G	A, L, E, H, J, C
H	G, I
I	H
J	G, L, K, M
K	J
L	G, J, M
M	L, J

If the depth first traversal is initiated from vertex A, then the vertices of graph G are visited in the order: ***A F E G L J K M H I C D B***. The depth first spanning tree is shown in the figure given below:

If the breadth first traversal is initiated from vertex A, then the vertices of graph G are visited in the order: **A** ***F B C G E D L H J M I K***. The breadth first spanning tree is shown in the figure given below:

APPLICATIONS OF GRAPHS:

- **Social network graphs:** to tweet or not to tweet. Graphs that represent who knows whom, who communicates with whom, who influences whom or other relationships in social structures. An example is the twitter graph of who follows whom. These can be used to determine how information flows, how topics become hot, how communities develop, or even who might be a good match for who, or is that whom.
- **Transportation networks:** In road networks vertices are intersections and edges are the road segments between them, and for public transportation networks vertices are stops and edges are the links between them. Such networks are used by many map programs such as **Google maps, Bing maps** and now Apple IOS 6 maps (well perhaps without the

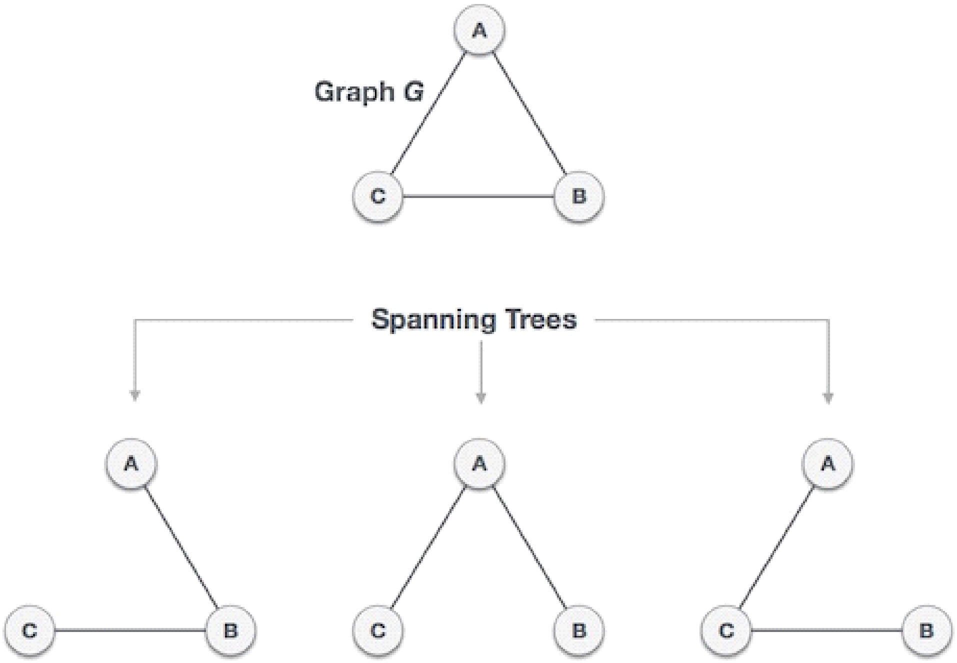
public transport) to find the best routes between locations. They are also used for studying traffic patterns, traffic light timings, and many aspects of transportation.

- **Utility graphs:** The power grid, the Internet, and the water network are all examples of graphs where vertices represent connection points, and edges the wires or pipes between them. Analyzing properties of these graphs is very important in understanding the reliability of such utilities under failure or attack, or in minimizing the costs to build infrastructure that matches required demands.
- **Document link graphs:** The best known example is the link graph of the web, where each web page is a vertex, and each hyperlink a directed edge. Link graphs are used, for example, to analyze relevance of web pages, the best sources of information, and good link sites.
- **Semantic networks:** Vertices represent words or concepts and edges represent the relationships among the words or concepts. These have been used in various models of how humans organize their knowledge, and how machines might simulate such an organization.
- **Graphs in compilers:** Graphs are used extensively in compilers. They can be used for type inference, for so called data flow analysis, register allocation and many other purposes. They are also used in specialized compilers, such as query optimization in database languages.
- **Dependence graphs:** Graphs can be used to represent dependences or precedences among items. Such graphs are often used in large projects in laying out what components rely on other components and used to minimize the total time or cost to completion while abiding by the dependences.

Spanning trees:

A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected.

By this definition, we can draw a conclusion that every connected and undirected Graph G has at least one spanning tree. A disconnected graph does not have any spanning tree, as it cannot be spanned to all its vertices.



We found three spanning trees off one complete graph.

A complete undirected graph can have maximum n^(n-2)number of spanning trees, where n is the number of nodes.

In the above addressed example, 3^(3−2) = 3 spanning trees are possible.

Properties of Spanning Tree:

- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G, have the same number of edges and vertices.
- The spanning tree does not have any cycle (loops).
- Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is minimally connected.
- Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is maximally acyclic
- Spanning tree has n-1 edges, where n is the number of nodes

Minimum Spanning Tree:

Minimum spanning tree (MST) is a spanning tree with the smallest possible weight.

Example:

A graph G:	Three (of many possible) spanning trees from graph G:

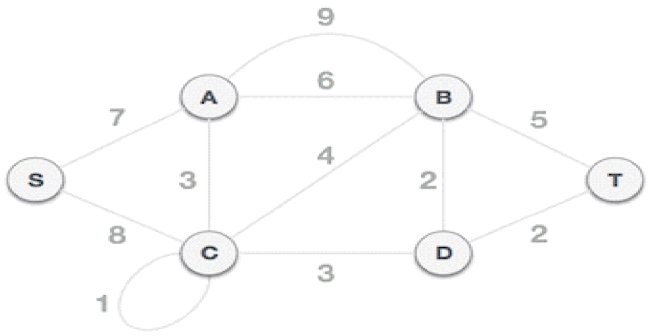
Minimum spanning tree, can be constructed using any of the following two algorithms:

- Kruskal’s algorithm and
- Prim’s algorithm.

1.Kruskal’s algorithm:

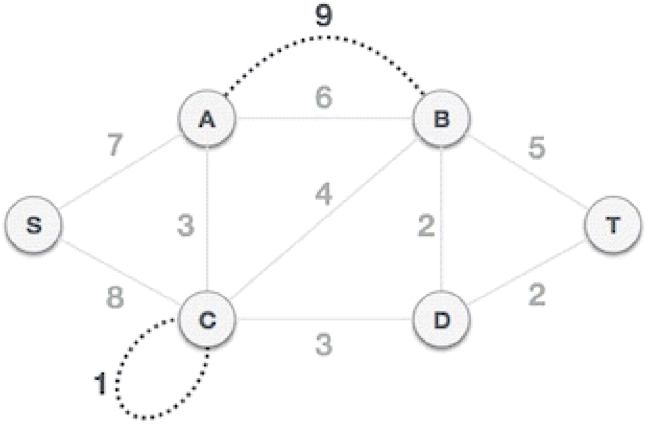
Kruskal's algorithm to find the minimum cost spanning tree uses the greedy approach.

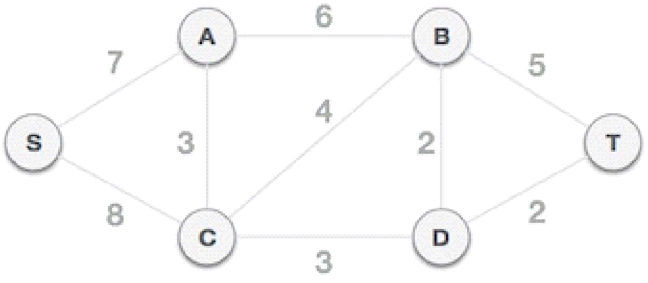
To understand Kruskal's algorithm let us consider the following example



Step 1 - Remove all loops and Parallel Edges

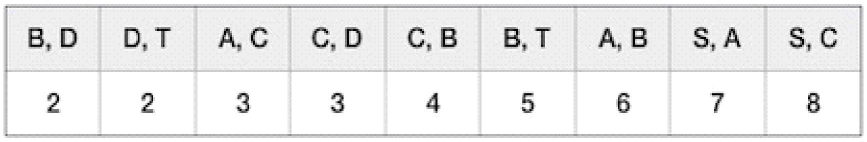
Remove all loops and parallel edges from the given graph.





Step 2 - Arrange all edges in their increasing order of weight

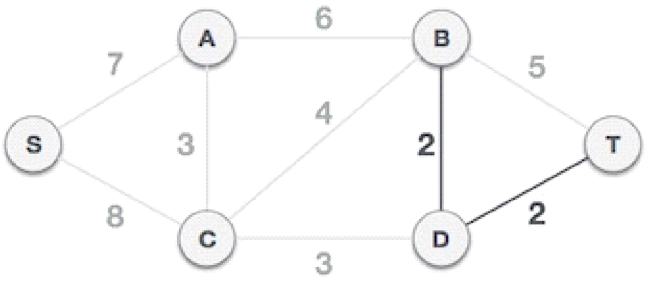
The next step is to create a set of edges and weight, and arrange them in an ascending order of weightage (cost).



B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8

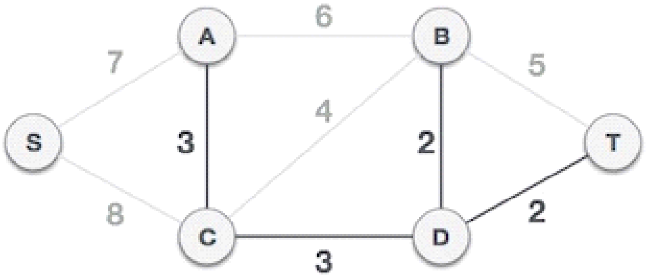
Step 3 - Add the edge which has the least weightage

- Now we start adding edges to the graph beginning from the one which has the least weight.
- Throughout, we shall keep checking that the spanning properties remain intact.
- In case, by adding one edge, the spanning tree property does not hold then we shall consider not to include the edge in the graph.

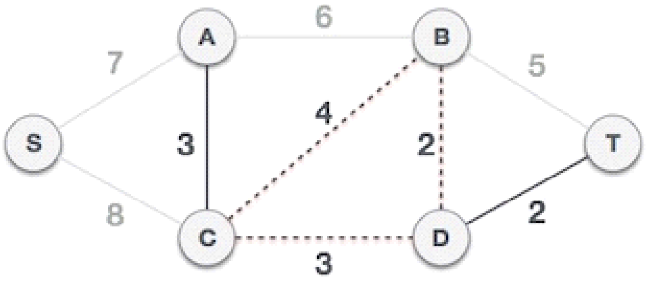


The least cost is 2 and edges involved are B,D and D,T. We add them. Adding them does not violate spanning tree properties, so we continue to our next edge selection.

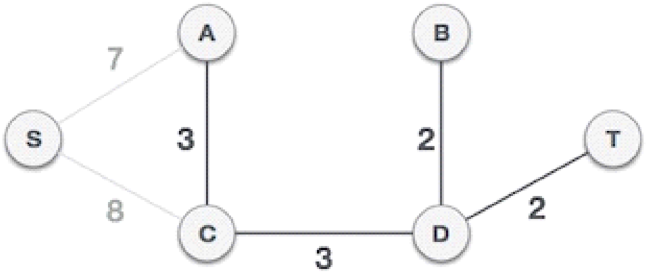
Next cost is 3, and associated edges are A,C and C,D. We add them again –



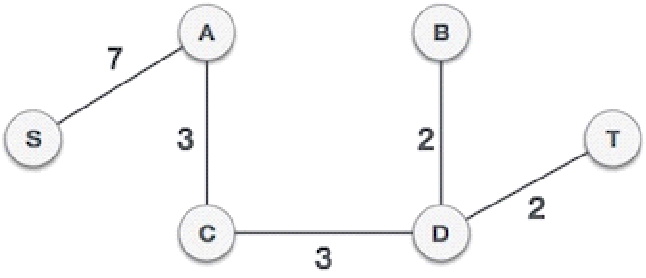
Next cost in the table is 4, and we observe that adding it will create a circuit in the graph. –



We observe that edges with cost 5 and 6 also create circuits. We ignore them and move on.



Now we are left with only one node to be added. Between the two least cost edges available 7 and 8, we shall add the edge with cost 7.



By adding edge S,A we have included all the nodes of the graph and we now have minimum cost spanning tree.

Kruskal's Algorithm for minimal spanning tree is as follows:

- Make the tree T empty.
- Repeat the steps 3, 4 and 5 as long as T contains less than n - 1 edges and E is not empty otherwise, proceed to step 6.
- Choose an edge (v, w) from E of lowest cost. 4. Delete (v, w) from E.
- If (v, w) does not create a cycle in T

  then Add (v, w) to T

  else discard (v, w)
- If T contains fewer than n - 1 edges then print no spanning tree.

Example 2:

Construct the minimal spanning tree for the graph shown below:

Arrange all the edges in the increasing order of their costs:

Cost	***10***	***15***	***20***	***25***	***30***	***35***	***40***	***45***	***50***	***55***
Edge	(1, 2)	(3, 6)	(4, 6)	(2, 6)	(1, 4)	(3, 5)	(2, 5)	(1, 5)	(2, 3)	(5, 6)

The stages in Kruskal's algorithm for minimal spanning tree is as follows:

EDGE	**COST**	**STAGES IN KRUSKAL'S ALGORITHM**	**REMARKS**
(1, 2)	10		The edge between vertices 1 and 2 is the first edge selected. It is included in the spanning tree.

(3, 6)	15		Next, the edge between vertices 3 and 6 is selected and included in the tree.
(4, 6)	20		The edge between vertices 4 and 6 is next included in the tree.
(2, 6)	25		The edge between vertices 2 and 6 is considered next and included in the tree.
(1, 4)	30	Reject	The edge between the vertices 1 and 4 is discarded as its inclusion creates a cycle.
(3, 5)	35		Finally, the edge between vertices 3 and 5 is considered and included in the tree built. This completes the tree. The cost of the minimal spanning tree is 105.

Example 3:

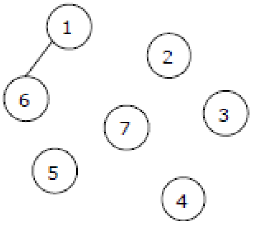
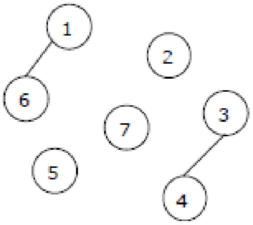
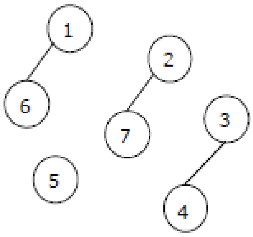
Construct the minimal spanning tree for the graph shown below:

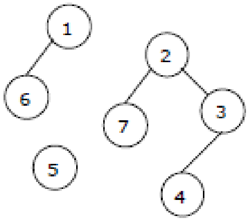
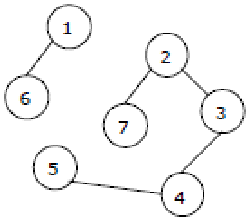
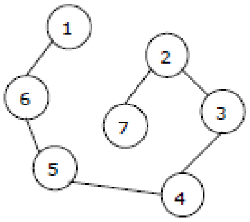
Solution:

Arrange all the edges in the increasing order of their costs:

Cost	***10***	***12***	***14***	***16***	***18***	***22***	***24***	***25***	***28***
Edge	(1, 6)	(3, 4)	(2, 7)	(2, 3)	(4, 7)	(4, 5)	(5, 7)	(5, 6)	(1, 2)

The stages in Kruskal's algorithm for minimal spanning tree is as follows:

EDGE	COST	STAGES IN KRUSKAL'S ALGORITHM	REMARKS
(1, 6)	10		The edge between vertices 1 and 6 is the first edge selected. It is included in the spanning tree.
(3, 4)	12		Next, the edge between vertices 3 and 4 is selected and included in the tree.
(2, 7)	14		The edge between vertices 2 and 7 is next included in the tree.

(2, 3)	16		The edge between vertices 2 and 3 is next included in the tree.
(4, 7)	18	Reject	The edge between the vertices 4 and 7 is discarded as its inclusion creates a cycle.
(4, 5)	22		The edge between vertices 4 and 7 is considered next and included in the tree.
(5, 7)	24	Reject	The edge between the vertices 5 and 7 is discarded as its inclusion creates a cycle.
(5, 6)	25		Finally, the edge between vertices 5 and 6 is considered and included in the tree built. This completes the tree. The cost of the minimal spanning tree is 99.

2 PRIM’s Algorithm:

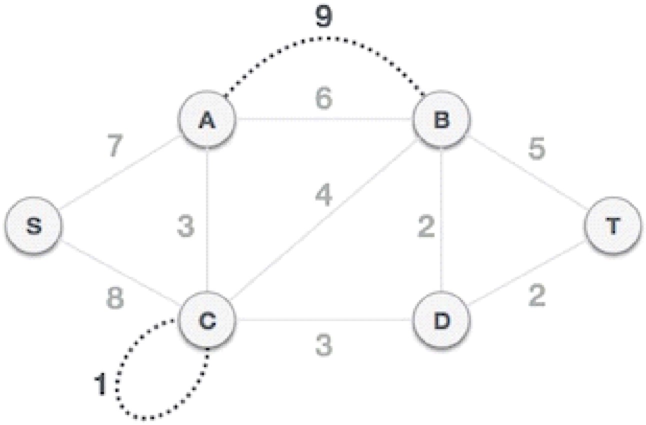
A given graph can have many spanning trees. From these many spanning trees, we have to select a cheapest one. This tree is called as minimal cost spanning tree.

Minimal cost spanning tree is a connected undirected graph G in which each edge is labeled with a number (edge labels may signify lengths, weights other than costs). Minimal cost spanning tree is a spanning tree for which the sum of the edge labels is as small as possible

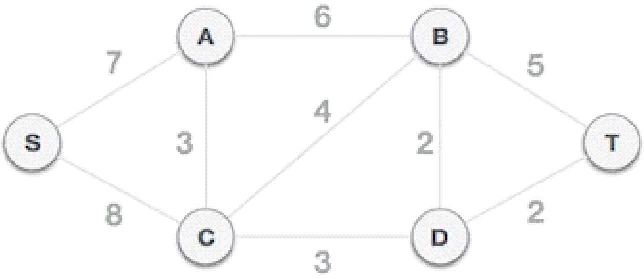
The slight modification of the spanning tree algorithm yields a very simple algorithm for finding an MST. In the spanning tree algorithm, any vertex not in the tree but connected to it by an edge can be added. To find a Minimal cost spanning tree, we must be selective - we must always add a new vertex for which the cost of the new edge is as small as possible.

This simple modified algorithm of spanning tree is called prim's algorithm for finding an Minimal cost spanning tree. Prim's algorithm is an example of a greedy algorithm.

Step 1 - Remove all loops and parallel edges



Remove all loops and parallel edges from the given graph. In case of parallel edges, keep the one which has the least cost associated and remove all others.

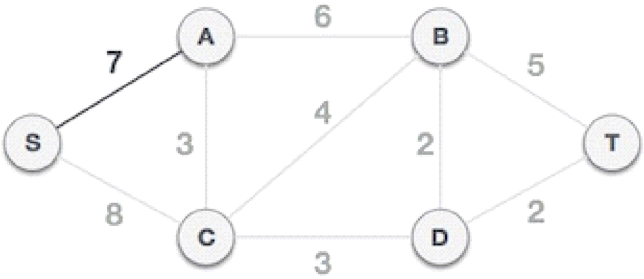


Step 2 - Choose any arbitrary node as root node

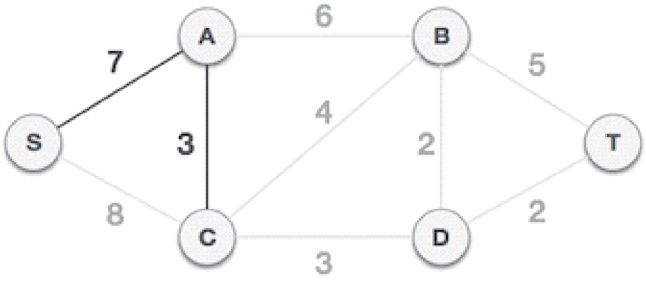
In this case, we choose **S** node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node. One may wonder why any video can be a root node. So the answer is, in the spanning tree all the nodes of a graph are included and because it is connected then there must be at least one edge, which will join it to the rest of the tree.

Step 3 - Check outgoing edges and select the one with less cost

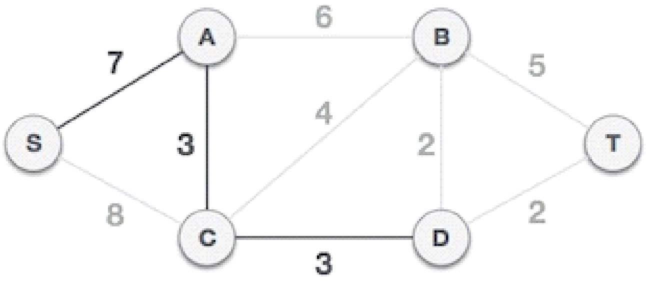
After choosing the root node **S**, we see that S,A and S,C are two edges with weight 7 and 8, respectively. We choose the edge S,A as it is lesser than the other.



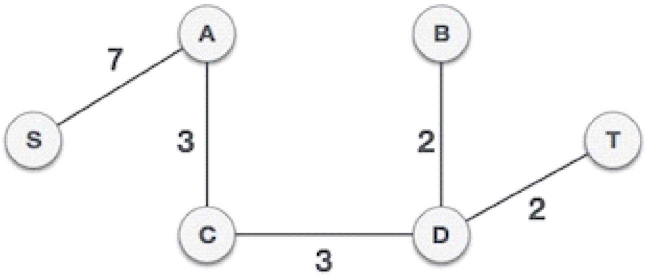
Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.



After this step, S-7-A-3-C tree is formed. Now we'll again treat it as a node and will check all the edges again. However, we will choose only the least cost edge. In this case, C-3-D is the new edge, which is less than other edges' cost 8, 6, 4, etc.



After adding node **D** to the spanning tree, we now have two edges going out of it having the same cost, i.e. D-2-T and D-2-B. Thus, we can add either one. But the next step will again yield edge 2 as the least cost. Hence, we are showing a spanning tree with both edges included.



We may find that the output spanning tree of the same graph using two different algorithms is same.

EXAMPLE:

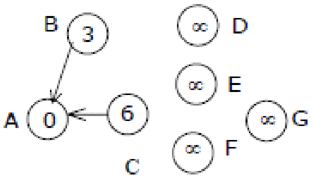
Use Prim’s Algorithm to find a minimal spanning tree for the graph shown below starting with vertex A.

Solution:

The cost adjacency matrix is $\begin{pmatrix} 0 & 3 & 6 & \infty & \infty & \infty & \infty \\ 3 & 0 & 2 & 4 & \infty & \infty & \infty \\ 6 & 2 & 0 & 1 & 4 & 2 & \infty \\ \infty & 4 & 1 & 0 & 2 & \infty & 4 \\ \infty & \infty & 4 & 2 & 0 & 2 & 1 \\ \infty & \infty & 2 & \infty & 2 & 0 & 1 \\ \infty & \infty & \infty & 4 & 1 & 1 & 0 \end{pmatrix}$

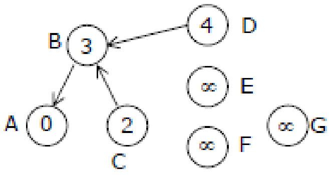
The stepwise progress of the prim's algorithm is as follows:

Step 1:



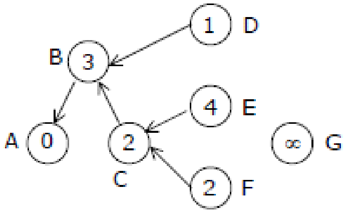
Vertex	A	B	C	D	E	F	G
Status	0	1	1	1	1	1	1
Dist.	0	3	6	∞	∞	∞	∞
Next	*	A	A	A	A	A	A

Step 2:



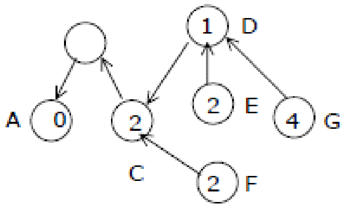
Vertex	A	B	C	D	E	F	G
Status	0	0	1	1	1	1	1
Dist.	0	3	2	4	∞	∞	∞
Next	*	A	B	B	A	A	A

Step 3:



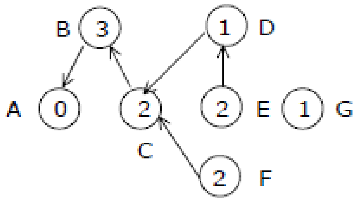
Vertex	A	B	C	D	E	F	G
Status	0	0	0	1	1	1	1
Dist.	0	3	2	1	4	2	∞
Next	*	A	B	C	C	C	A

Step 4:



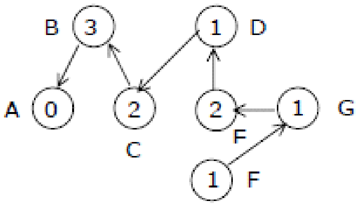
Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	1	1	1
Dist.	0	3	2	1	2	2	4
Next	*	A	B	C	D	C	D

Step 5:



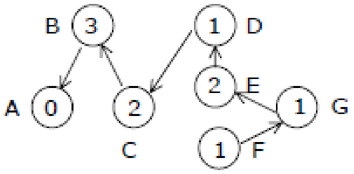
Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	1	0	1
Dist.	0	3	2	1	2	2	1
Next	*	A	B	C	D	C	E

Step 6:



Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	0	1	0
Dist.	0	3	2	1	2	1	1
Next	*	A	B	C	D	G	E

Step 7:



Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	0	0	0
Dist.	0	3	2	1	2	1	1
Next	*	A	B	C	D	G	E

```
// C program to implement BFS
#include<stdio.h>
#include<stdlib.h>
```

```
#define MAX 20
#define initial 0
#define waiting 1
#define visited 2
int Q[MAX],front=-1,rear=-1;
int status[20];

int isempty()
{
        if(front==-1)
         return 1;
        else
         return 0;
}
void insertion(int el)
{
        if(front==-1)
         front=0;
        rear++;
        Q[rear]=el;
}
int deletion()
{
        int el;
        if(isempty())
          exit(1);
        el=Q[front];
        front++;
        if(front==rear+1)
          front=rear=-1;
        return el;
}
void BFS(int G[][20],int v,int n)
{
        int i;
        insertion(v);
        status[v]=waiting;
        while(!isempty())
        {
           v=deletion();
           status[v]=visited;
           printf("\t%d",v);
          for(i=0;i<n;i++)
          {
                 if(G[v][i]==1 && status[i]==initial)
                 {
                   insertion(i);
                   status[i]=waiting;
           }
          }
   }
}

void BFS_Traversal(int G[][20],int n)
{
        int v,i;
        for(i=0;i<n;i++)
         status[i]=initial;
        printf("enter starting vertex");
        scanf("%d",&v);
        BFS(G,v,n);
        for(i=0;i<n;i++)
        {
                if(status[i]==initial)
                 BFS(G,i,n);
        }
}
void creategraph(int G[][20],int n)
{
        int sourse,dest,i,max_edges,type,j;
        printf("\nenter graph type 1 for Undirected graph and 2 for Directed graph");
        scanf("%d",&type);
        if(type==1)
         max_edges= n*(n-1)/2;
        else
         max_edges=n*(n-1);
        for(i=1;i<=max_edges;i++)
        {
                printf("\nenter edge %d  (enter -1 and -1 to terminate)",i);
                scanf("%d%d",&sourse,&dest);
                if(sourse==-1 || dest==-1)
                  break;
                if(sourse<0 || sourse>=n || dest<0 || dest>=n)
                {
```

```
            printf("\nInvalid edge");
            i--;
        }
        else
        {
          G[sourse][dest]=1;
          if(type==1)
           G[dest][sourse]=1;
      }
    }
}
void display(int G[][20],int n)
{
    int i,j;
    printf("\nADJACENSY MATRIX\n");
    for(i=0;i<n;i++)
    {
        for(j=0;j<n;j++)
         printf("\t%d",G[i][j]);
        printf("\n");
    }
}
main()
{
    int n, G[20][20],i,j;
    printf("enter number of vertices");
  scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        for(j=0;j<n;j++)
         G[i][j]=0;
  }
    creategraph(G,n);
    display(G,n);
    BFS_Traversal(G,n);
}
```

OUTPUT BFS:

```
enter number of vertices6

enter graph type 1 for Undirected graph and 2 for Directed graph1
enter edge 1  (enter -1 and -1 to terminate)0 1

enter edge 2  (enter -1 and -1 to terminate)0 2

enter edge 3  (enter -1 and -1 to terminate)1 2

enter edge 4  (enter -1 and -1 to terminate)1 3

enter edge 5  (enter -1 and -1 to terminate)2 4

enter edge 6  (enter -1 and -1 to terminate)3 4

enter edge 7  (enter -1 and -1 to terminate)3 5

enter edge 8  (enter -1 and -1 to terminate)4 5

enter edge 9  (enter -1 and -1 to terminate)-1 -1

ADJACENSY MATRIX
        0       1       1       0       0       0
        1       0       1       1       0       0
        1       1       0       0       1       0
        0       1       0       0       1       1
        0       0       1       1       0       1
        0       0       0       1       1       0
enter starting vertex0
        0       1       2       3       4       5
```

```
// c program for DFS

#include<stdio.h>
#include<stdlib.h>
#define MAX 20
#define initial 0
#define visited 1
int S[MAX],top=-1;
int status[20];

int isempty()
{
        if(top==-1)
         return 1;
```

```
            else
             return 0;
}
void push(int el)
{
            S[++top]=el;
}
int pop()
{
            if(isempty())
              exit(1);
            return S[top--];
}
void DFS(int G[][20],int v,int n)
{
            int i;
            push(v);
            while(!isempty())
            {
               v=pop();
               if(status[v]==initial)
               {
                  status[v]=visited;
                  printf("\t%d",v);
           }
               for(i=n-1;i>=0;i--)
               {
                         if(G[v][i]==1 && status[i]==initial)
                            push(i);
               }
    }
}

void DFS_Traversal(int G[][20],int n)
{
            int v,i;
            for(i=0;i<n;i++)
              status[i]=initial;
            printf("enter starting vertex");
            scanf("%d",&v);
            DFS(G,v,n);
            for(i=0;i<n;i++)
```

```
        {
                if(status[i]==initial)
                 DFS(G,i,n);
        }
}
void creategraph(int G[][20],int n)
{
        int sourse,dest,i,max_edges,type,j;
        printf("\nenter graph type 1 for Undirected graph and 2 for Directed graph");
        scanf("%d",&type);
        if(type==1)
         max_edges= n*(n-1)/2;
        else
         max_edges=n*(n-1);
        for(i=1;i<=max_edges;i++)
        {
                printf("\nenter edge %d  (enter -1 and -1 to terminate)",i);
                scanf("%d%d",&sourse,&dest);
                if(sourse==-1 || dest==-1)
                  break;
                if(sourse<0 || sourse>=n || dest<0 || dest>=n)
                {
                        printf("\nInvalid edge");
                        i--;
                }
                else
                {
                  G[sourse][dest]=1;
                  if(type==1)
                   G[dest][sourse]=1;
            }
        }
}
void display(int G[][20],int n)
{
        int i,j;
        printf("\nADJACENSY MATRIX\n");
        for(i=0;i<n;i++)
        {
                for(j=0;j<n;j++)
                 printf("\t%d",G[i][j]);
                printf("\n");
        }
}
main()
{
        int n, G[20][20],i,j;
        printf("enter number of vertices");
   scanf("%d",&n);
        for(i=0;i<n;i++)
        {
                for(j=0;j<n;j++)
                 G[i][j]=0;
   }
        creategraph(G,n);
        display(G,n);
        DFS_Traversal(G,n);
}
```

OUTPUT:

```
enter number of vertices6

enter graph type 1 for Undirected graph and 2 for Directed graph1

enter edge 1  (enter -1 and -1 to terminate)0 1

enter edge 2  (enter -1 and -1 to terminate)0 2

enter edge 3  (enter -1 and -1 to terminate)1 2

enter edge 4  (enter -1 and -1 to terminate)1 3

enter edge 5  (enter -1 and -1 to terminate)3 4

enter edge 6  (enter -1 and -1 to terminate)2 4

enter edge 7  (enter -1 and -1 to terminate)3 5

enter edge 8  (enter -1 and -1 to terminate)3 4

enter edge 9  (enter -1 and -1 to terminate)4 5

enter edge 10  (enter -1 and -1 to terminate)-1 -1
```

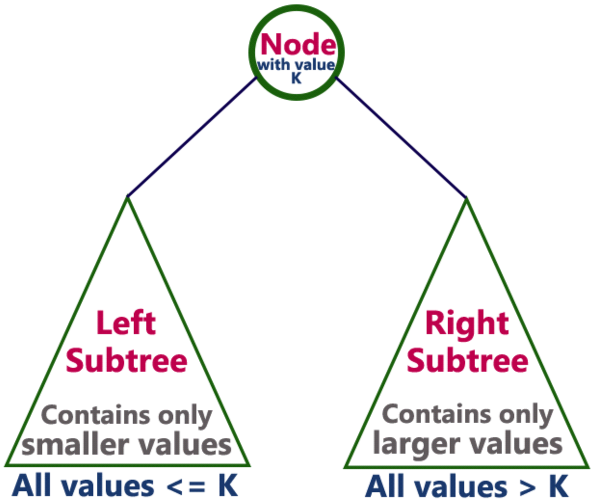
```
ADJACENSY MATRIX
        0       1       1       0       0       0
        1       0       1       1       0       0
        1       1       0       0       1       0
        0       1       0       0       1       1
        0       0       1       1       0       1
        0       0       0       1       1       0
enter starting vertex0
        0       1       2       4       3       5
```

MODULE-V: Search Trees [09 Periods]

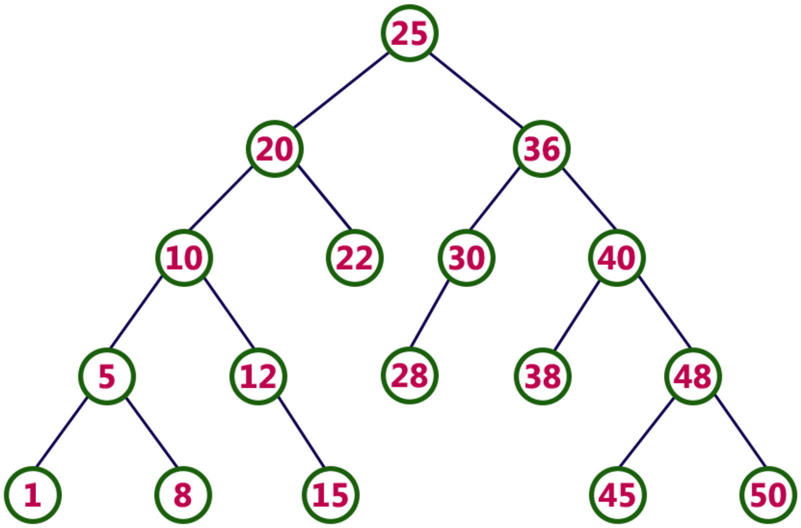
Binary Search Trees and AVL Trees: Binary Search Tree, Definition, Operations - Searching, Insertion and Deletion, AVL Trees (Elementary treatment-only Definitions and Examples).B-Trees and Red-Black Tree: B-Trees, Red-Black and Splay Trees (Elementary treatment-only Definitions and Examples), Comparison of Search Trees.

Binary Search Tree

Binary Search Tree is a binary tree in which every node contains only smaller values in its left subtree and only larger values in its right subtree.



Example:



Operations on Binary Search tree:

The following operations are performed on a binary earch tree...

- Search
- Insertion
- Deletion

SEARCH

In a binary search tree, the search operation is performed with O(log n) time complexity. The search operation is performed as follows...

Step 1: Read the search element from the user

Step 2: Compare, the search element with the value of root node in the tree.

Step 3: If both are matching, then display "Given node found!!!" and terminate the function

Step 4: If both are not matching, then check whether search element is smaller or larger than that node value.

Step 5: If search element is smaller, then continue the search process in left subtree.

Step 6: If search element is larger, then continue the search process in right subtree.

Step 7: Repeat the same until we found exact element or we completed with a leaf node

Step 8: If we reach to the node with search value, then display "Element is found" and terminate the function.

Step 9: If we reach to a leaf node and it is also not matching, then display "Element not found" and terminate the function.

Insertion Operation in BST

In a binary search tree, the insertion operation is performed with O(log n) time complexity. In binary search tree, new node is always inserted as a leaf node. The insertion operation is performed as follows...

Step 1: Create a newNode with given value and set its left and right to NULL.

Step 2: Check whether tree is Empty.

Step 3: If the tree is Empty, then set set root to newNode.

Step 4: If the tree is Not Empty, then check whether value of newNode is smaller or larger than the node (here it is root node).

Step 5: If newNode is smaller than or equal to the node, then move to its left child. If newNode is larger than the node, then move to its right child.

Step 6: Repeat the above step until we reach to a leaf node (e.i., reach to NULL).

Step 7: After reaching a leaf node, then insert the newNode as left child if newNode is smaller or equal to that leaf else insert it as right child.

Deletion Operation in BST

In a binary search tree, the deletion operation is performed with O(log n) time complexity. Deleting a node from Binary search tree has follwing three cases...

Case 1: Deleting a Leaf node (A node with no children)

Case 2: Deleting a node with one child

Case 3: Deleting a node with two children

Case 1: Deleting a leaf node

We use the following steps to delete a leaf node from BST...

Step 1: Find the node to be deleted using search operation

Step 2: Delete the node using free function (If it is a leaf) and terminate the function.

Case 2: Deleting a node with one child

We use the following steps to delete a node with one child from BST...

Step 1: Find the node to be deleted using search operation

Step 2: If it has only one child, then create a link between its parent and child nodes.

Step 3: Delete the node using free function and terminate the function.

Case 3: Deleting a node with two children

We use the following steps to delete a node with two children from BST...

Step 1: Find the node to be deleted using search operation

Step 2: If it has two children, then find the largest node in its left subtree (OR) the smallest node in its right subtree.

Step 3: Swap both deleting node and node which found in above step.

Step 4: Then, check whether deleting node came to case 1 or case 2 else goto steps 2

Step 5: If it comes to case 1, then delete using case 1 logic.

Step 6: If it comes to case 2, then delete using case 2 logic.

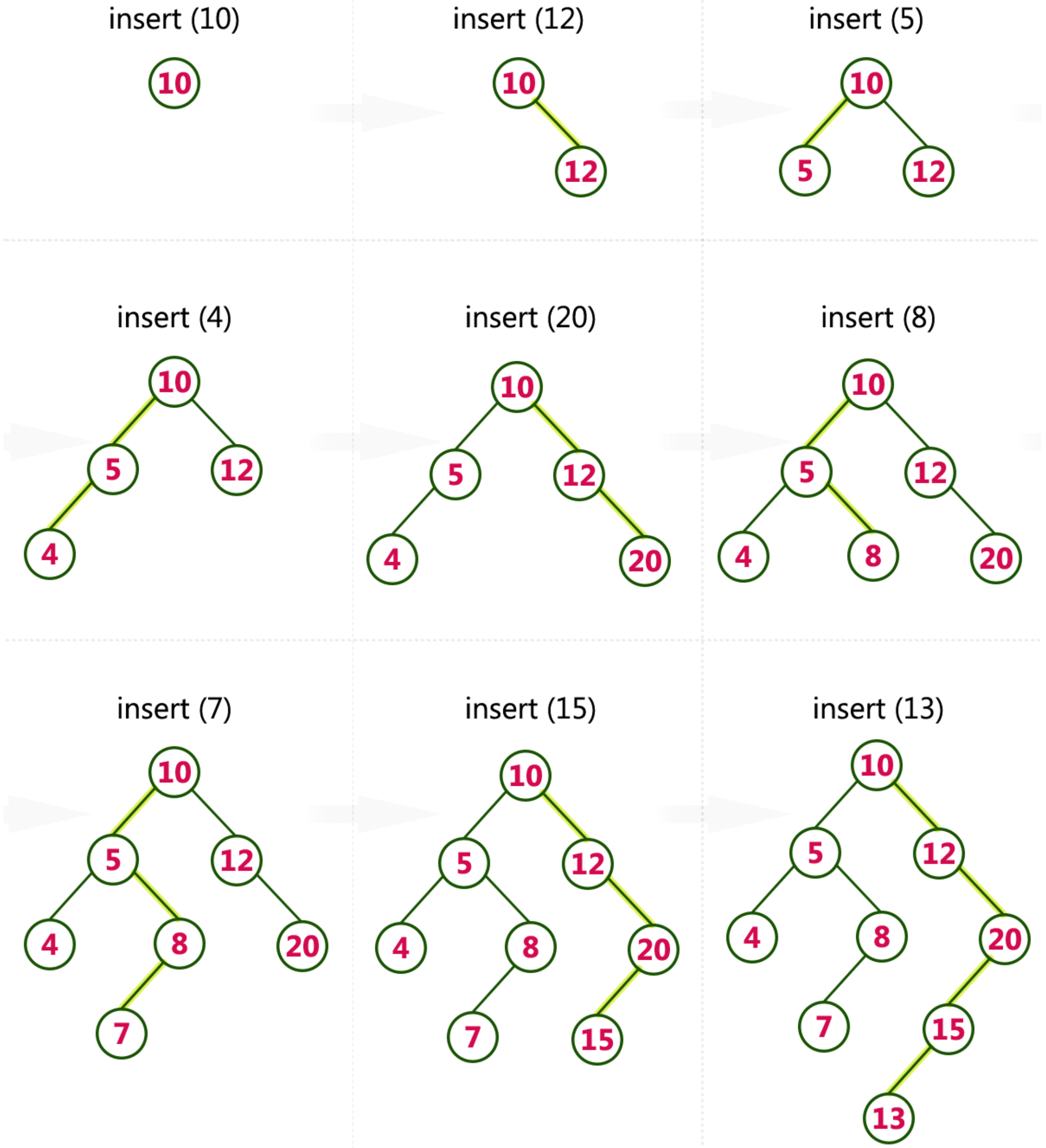
Step 7: Repeat the same process until node is deleted from the tree.

Example

Construct a Binary Search Tree by inserting the following sequence of numbers...

10,12,5,4,20,8,7,15 and 13

Above elements are inserted into a Binary Search Tree as follows...


insert (10)
10
insert (12)
10
12
insert (5)
10
5
12
insert (4)
10
5
12
4
insert (20)
10
5
12
4
20
insert (8)
10
5
12
4
8
20
insert (7)
10
5
12
4
8
20
7
insert (15)
10
5
12
4
8
20
7
15
insert (13)
10
5
12
4
8
20
7
15
13

```
// bst program
#include<stdio.h>
struct node
{
 int data;
 struct node *left,*right;
};
struct node* insert(struct node *p, int el)
{
 if(p==NULL)
 {
  p=(struct node*) malloc(sizeof(struct node));
  p->data=el;
  p->left=p->right=NULL;
 }
 else if(el<p->data)
  p->left=insert(p->left,el);
 else if(el>p->data)
  p->right=insert(p->right,el);
 return p;
}
int search(struct node *p,int el)
{
 if(p==NULL)
  return 0;
 if(el<p->data)
  search(p->left,el);
 else if(el>p->data)
  search(p->right,el);
 else
  return 1;
}

main()
{
 struct node *root=NULL;
 int i,n,el,f,key;
 printf("enter number of elements");
 scanf("%d",&n);
 printf("\nenter elements");
 for(i=1;i<=n;i++)

 {

  scanf("%d",&el);

  root=insert(root,el);

 }

 printf("enter element to search");

 scanf("%d",&key);

 f=search(root,key);

 if(f==1)

  printf("element found");

 else

  printf("element not found");

}
```

OUTPUT:

```
enter number of elements9



enter elements10 12 5 4 20 8 7 15 13

enter element to search7

element found
```

AVL TREE(Adelson, Velski & Landis)

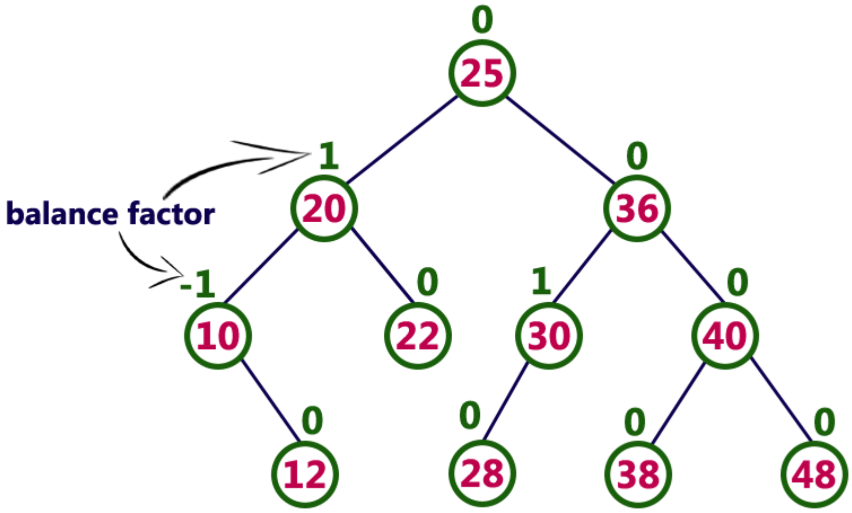
- An AVL tree is a balanced binary search tree. In an AVL tree, balance factor of every node is either -1, 0 or +1.
- AVL tree is a self balanced binary search tree. That means, an AVL tree is also a binary search tree but it is a balanced tree.
- A binary tree is said to be balanced, if the difference between the hieghts of left and right subtrees of every node in the tree is either -1, 0 or +1.
- In other words, a binary tree is said to be balanced if for every node, height of its children differ by at most one.
- In an AVL tree, every node maintains extra information known as balance factor.
- The AVL tree was introduced in the year of 1962 by G.M. Adelson-Velsky and E.M.

Landis.

- **Balance factor** of a node is the difference between the heights of left and right subtrees of that node.
- The balance factor of a node is calculated either height of left subtree - height of right subtree (OR) height of right subtree - height of left subtree.
- In the following explanation, we are calculating as follows...

Balance factor = heightOfLeftSubtree - heightOfRightSubtree

Example:

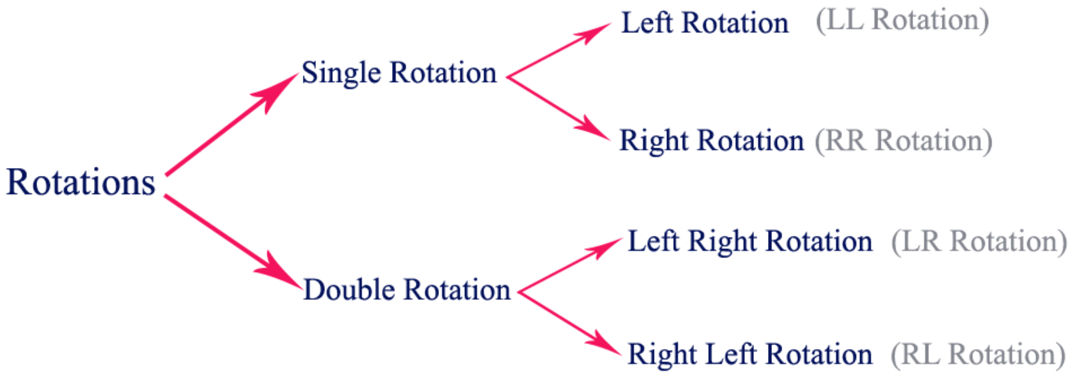


AVL Tree Rotations:

- In AVL tree, after performing every operation like insertion and deletion we need to check the balance factor of every node in the tree.
- If every node satisfies the balance factor condition then we conclude the operation otherwise we must make it balanced.
- We use rotation operations to make the tree balanced whenever the tree is becoming imbalanced due to any operation.

- Rotation operations are used to make a tree balanced.
- Rotation is the process of moving the nodes to either left or right to make tree balanced.

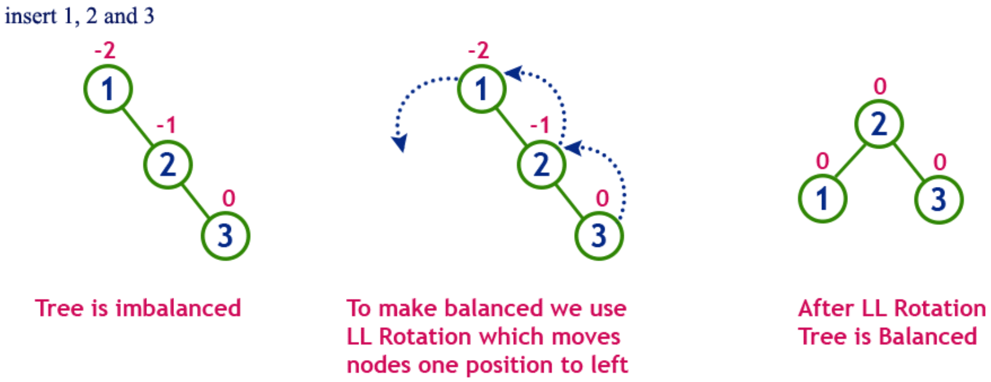
There are four rotations and they are classified into two types.


Rotations
Single Rotation
Left Rotation (LL Rotation)
Right Rotation (RR Rotation)
Double Rotation
Left Right Rotation (LR Rotation)
Right Left Rotation (RL Rotation)

<u>Single Left Rotation (LL Rotation)</u>

In LL Rotation every node moves one position to left from the current position.

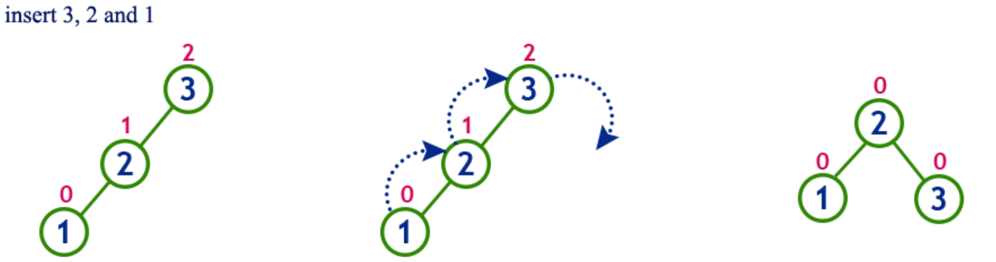
To understand LL Rotation, let us consider following insertion operations into an AVL Tree...


insert 1, 2 and 3
-2
1
-1
2
0
3
Tree is imbalanced
-2
1
-1
2
0
3
To make balanced we use LL Rotation which moves nodes one position to left
0
2
0
1
0
3
After LL Rotation Tree is Balanced

Single Right Rotation (RR Rotation):

In RR Rotation every node moves one position to right from the current position.

To understand RR Rotation, let us consider following insertion operations into an AVL Tree...



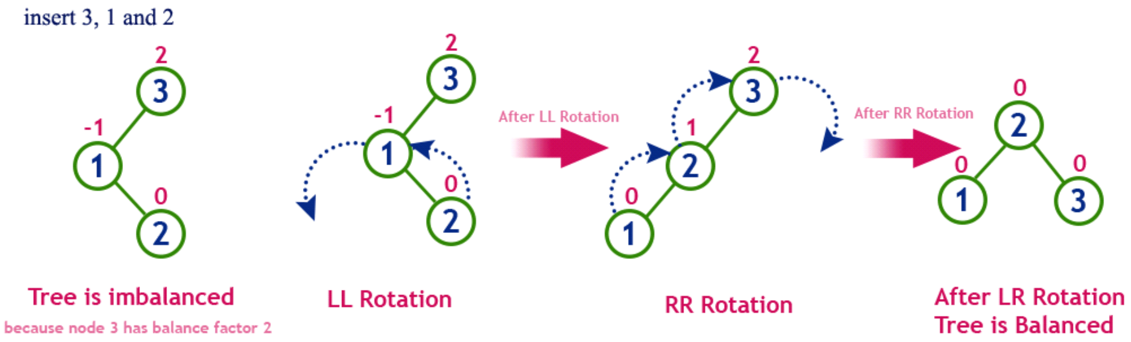
Tree is imbalanced
because node 3 has balance factor 2

To make balanced we use RR Rotation which moves nodes one position to right

After RR Rotation Tree is Balanced

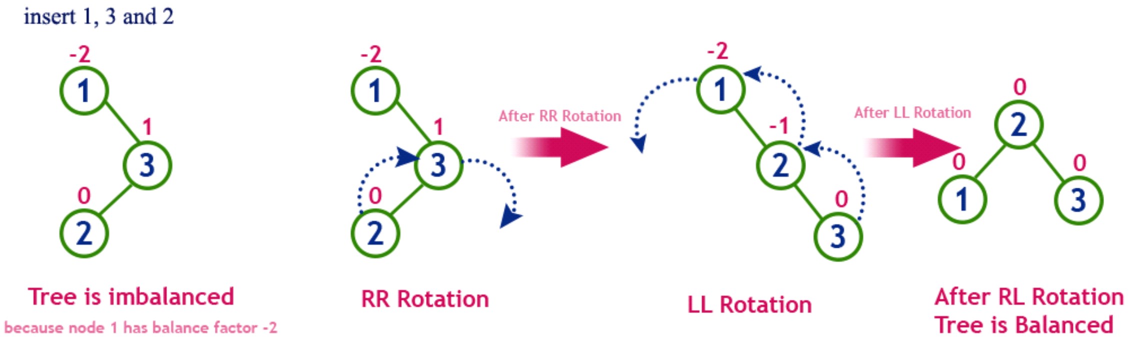
Left Right Rotation (LR Rotation):

The LR Rotation is combination of single left rotation followed by single right rotation. In LR Roration, first every node moves one position to left then one position to right from the current position. To understand LR Rotation, let us consider following insertion operations into an AVL Tree...



Right Left Rotation (RL Rotation)

The RL Rotation is combination of single right rotation followed by single left rotation. In RL Roration, first every node moves one position to right then one position to left from the current position. To understand RL Rotation, let us consider following insertion operations into an AVL Tree...



Operations on an AVL Tree

The following operations are performed on an AVL tree...

- Search
- Insertion
- Deletion

Search Operation in AVL Tree

In an AVL tree, the search operation is performed with O(log n) time complexity. The search operation is performed similar to Binary search tree search operation. We use the following steps to search an element in AVL tree...

Step 1: Read the search element from the user

Step 2: Compare, the search element with the value of root node in the tree.

Step 3: If both are matching, then display "Given node found!!!" and terminate the function

Step 4: If both are not matching, then check whether search element is smaller or larger than that node value.

Step 5: If search element is smaller, then continue the search process in left subtree.

Step 6: If search element is larger, then continue the search process in right subtree.

Step 7: Repeat the same until we found exact element or we completed with a leaf node

Step 8: If we reach to the node with search value, then display "Element is found" and terminate the function.

Step 9: If we reach to a leaf node and it is also not matching, then display "Element not found" and terminate the function.

Insertion Operation in AVL Tree

In an AVL tree, the insertion operation is performed with O(log n) time complexity. In AVL Tree, new node is always inserted as a leaf node. The insertion operation is performed as follows...

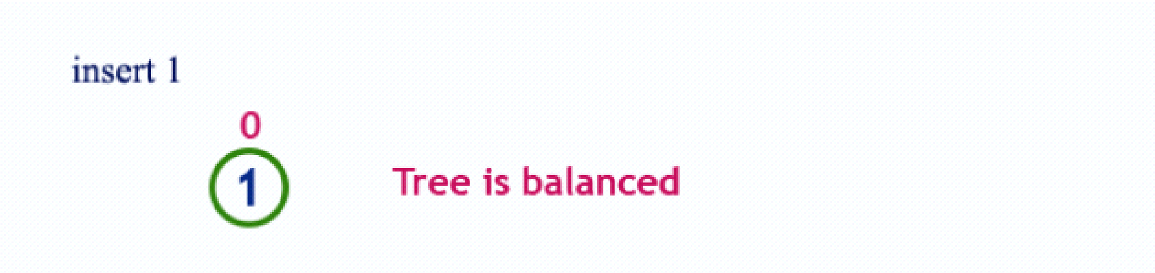
Step 1: Insert the new element into the tree using Binary Search Tree insertion logic.

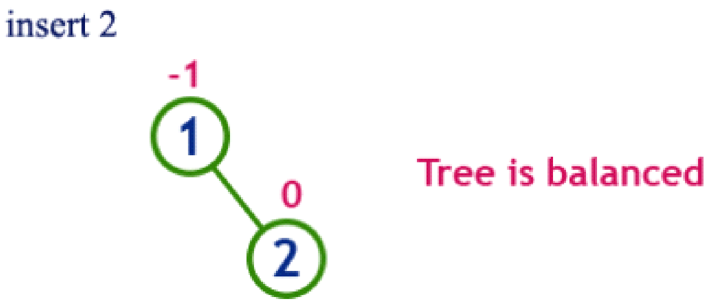
Step 2: After insertion, check the Balance Factor of every node.

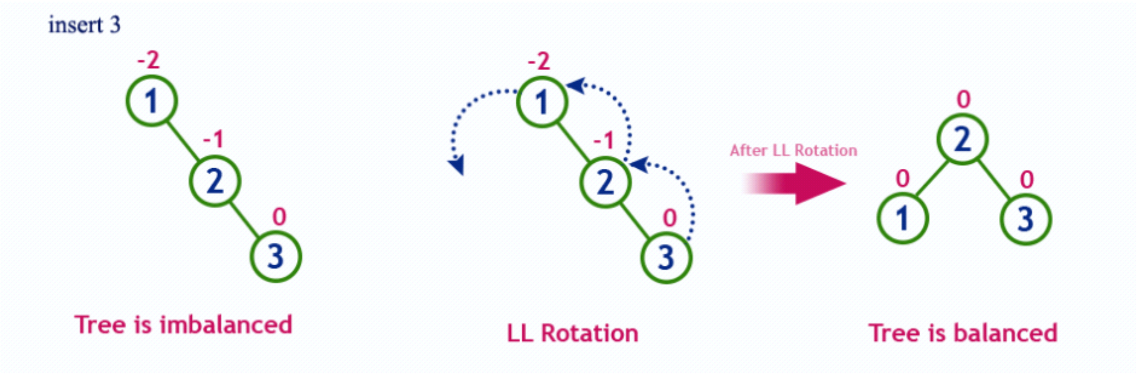
Step 3: If the Balance Factor of every node is 0 or 1 or -1 then go for next operation.

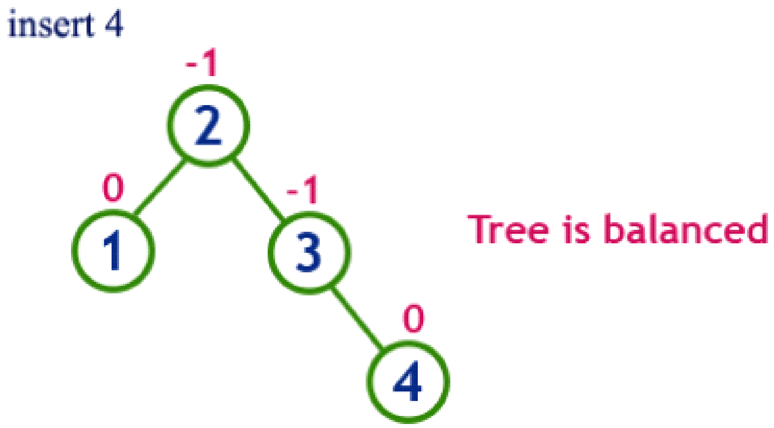
Step 4: If the Balance Factor of any node is other than 0 or 1 or -1 then tree is said to be imbalanced. Then perform the suitable Rotation to make it balanced. And go for next operation.

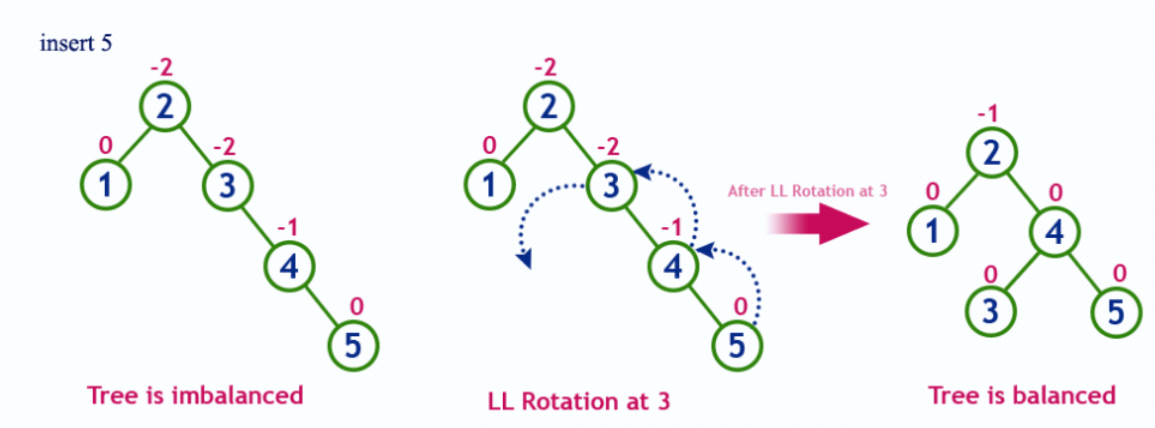
Example: Construct an AVL Tree by inserting numbers from 1 to 8.


insert 1
0
1
Tree is balanced

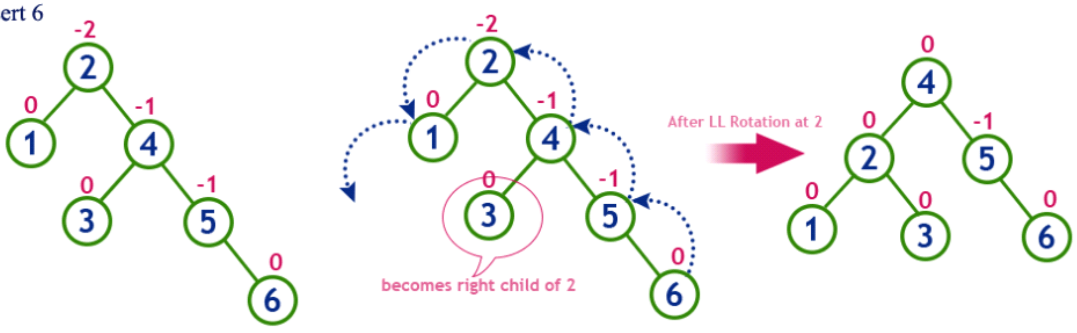

insert 2
-1
1
0
2
Tree is balanced


insert 3
-2
1
-1
2
0
3
Tree is imbalanced
-2
1
-1
2
0
3
LL Rotation
After LL Rotation
0
2
0
1
0
3
Tree is balanced


insert 4
-1
2
0
1
-1
3
0
4
Tree is balanced


insert 5
-2
2
0
1
-2
3
-1
4
0
5
Tree is imbalanced
-2
2
0
1
-2
3
-1
4
0
5
LL Rotation at 3
After LL Rotation at 3
-1
2
0
1
0
4
0
3
0
5
Tree is balanced

insert 6

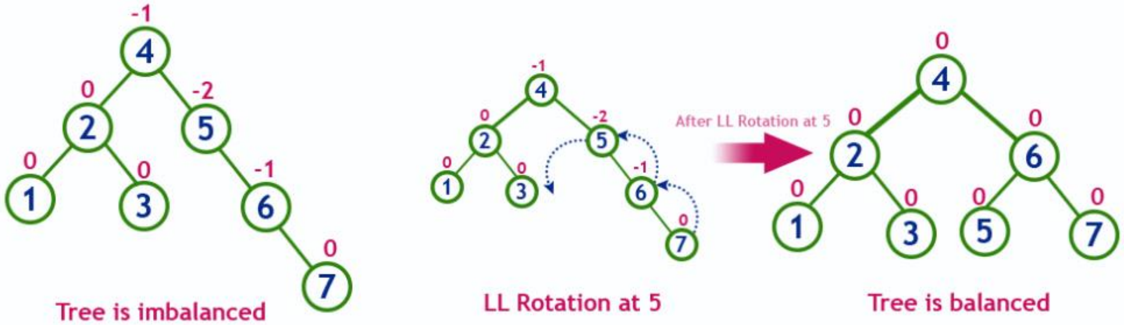


Tree is imbalanced

LL Rotation at 2

Tree is balanced

insert 7

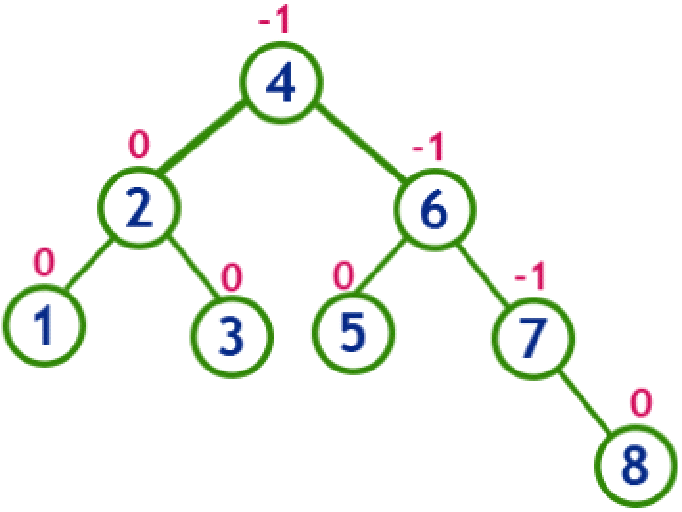


Tree is imbalanced

LL Rotation at 5

Tree is balanced

insert 8



Deletion Operation in AVL Tree

In an AVL Tree, the deletion operation is similar to deletion operation in BST. But after every deletion operation we need to check with the Balance Factor condition. If the tree is balanced after deletion, then go for next operation otherwise perform the suitable rotation to make the tree Balanced.

B Tree

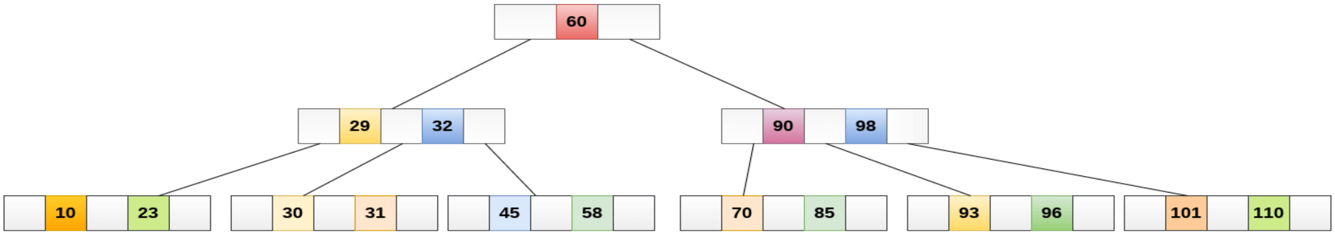
B Tree is a specialized m-way tree that can be widely used for disk access. A B-Tree of order m can have at most m-1 keys and m children. One of the main reason of using B tree is its capability to store large number of keys in a single node and large key values by keeping the height of the tree relatively small.

A B tree of order m contains all the properties of an M way tree. In addition, it contains the following properties.

1. Every node in a B-Tree contains at most m children.
2. Every node in a B-Tree except the root node and the leaf node contain at least m/2 children.
3. The root nodes must have at least 2 nodes.
4. All leaf nodes must be at the same level.

It is not necessary that, all the nodes contain the same number of children but, each node must have m/2 number of nodes.

A B tree of order 4 is shown in the following image.



While performing some operations on B Tree, any property of B Tree may violate such as number of minimum children a node can have. To maintain the properties of B Tree, the tree may split or join.

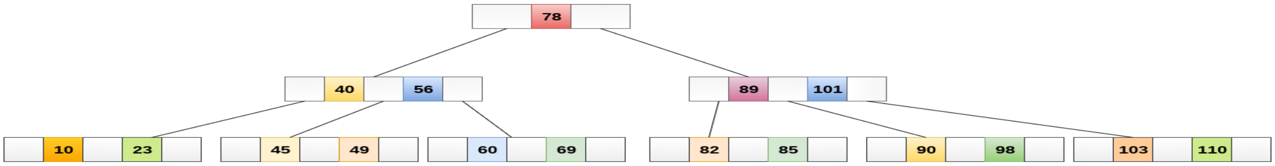
Operations

Searching :

Searching in B Trees is similar to that in Binary search tree. For example, if we search for an item 49 in the following B Tree. The process will something like following :

1. Compare item 49 with root node 78. since 49 < 78 hence, move to its left sub-tree.
2. Since, 40<49<56, traverse right sub-tree of 40.
3. 49>45, move to right. Compare 49.
4. match found, return.

Searching in a B tree depends upon the height of the tree. The search algorithm takes O(log n) time to search any element in a B tree.



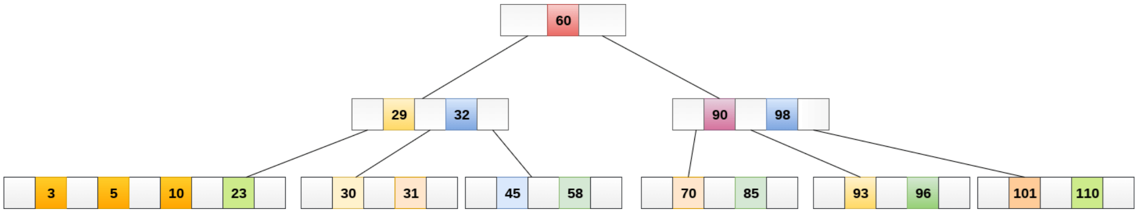
Inserting

Insertions are done at the leaf node level. The following algorithm needs to be followed in order to insert an item into B Tree.

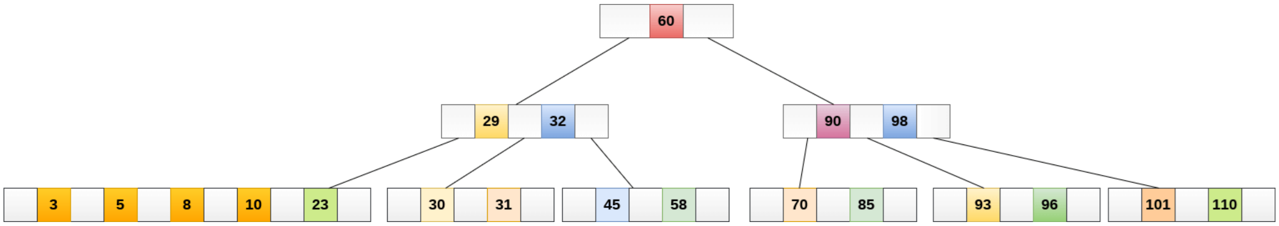
1. Traverse the B Tree in order to find the appropriate leaf node at which the node can be inserted.
2. If the leaf node contain less than m-1 keys then insert the element in the increasing order.
3. Else, if the leaf node contains m-1 keys, then follow the following steps.
   - Insert the new element in the increasing order of elements.
   - Split the node into the two nodes at the median.
   - Push the median element upto its parent node.
   - If the parent node also contain m-1 number of keys, then split it too by following the same steps.

Example:

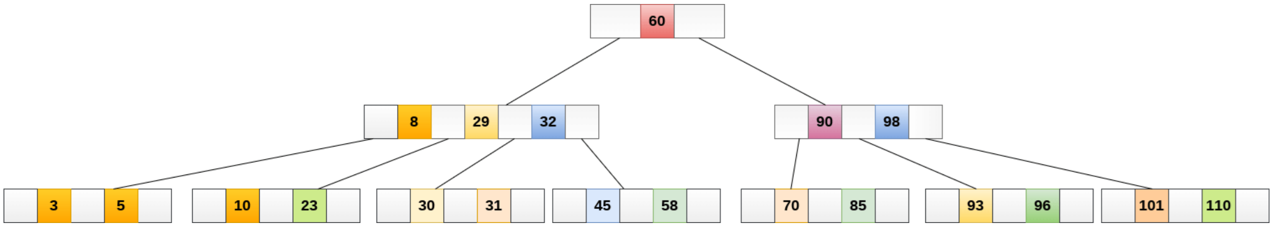
Insert the node 8 into the B Tree of order 5 shown in the following image.



8 will be inserted to the right of 5, therefore insert 8.



The node, now contain 5 keys which is greater than (5 -1 = 4) keys. Therefore split the node from the median i.e. 8 and push it up to its parent node shown as follows.



Deletion

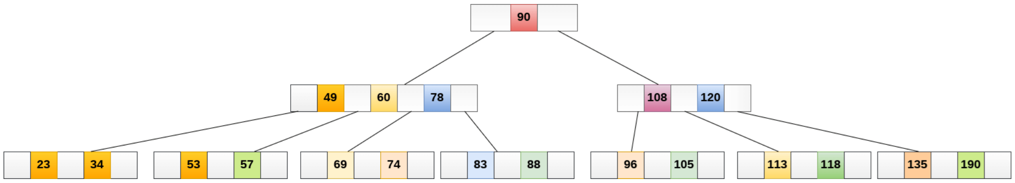
Deletion is also performed at the leaf nodes. The node which is to be deleted can either be a leaf node or an internal node. Following algorithm needs to be followed in order to delete a node from a B tree.

1. Locate the leaf node.
2. If there are more than m/2 keys in the leaf node then delete the desired key from the node.
3. If the leaf node doesn't contain m/2 keys then complete the keys by taking the element from eight or left sibling.
    - If the left sibling contains more than m/2 elements then push its largest element up to its parent and move the intervening element down to the node where the key is deleted.
    - If the right sibling contains more than m/2 elements then push its smallest element up to the parent and move intervening element down to the node where the key is deleted.
4. If neither of the sibling contain more than m/2 elements then create a new leaf node by joining two leaf nodes and the intervening element of the parent node.
5. If parent is left with less than m/2 nodes then, apply the above process on the parent too.

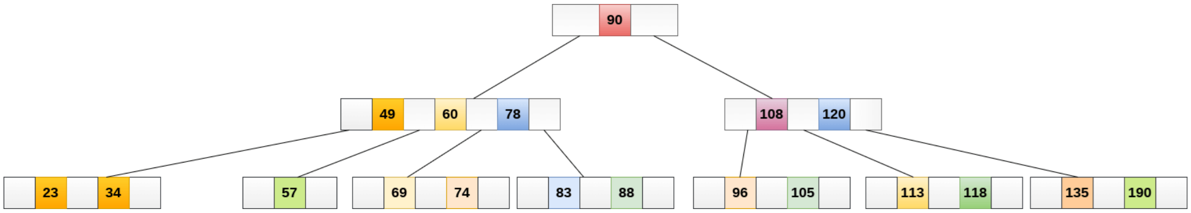
If the the node which is to be deleted is an internal node, then replace the node with its in-order successor or predecessor. Since, successor or predecessor will always be on the leaf node hence, the process will be similar as the node is being deleted from the leaf node.

Example 1

Delete the node 53 from the B Tree of order 5 shown in the following figure.

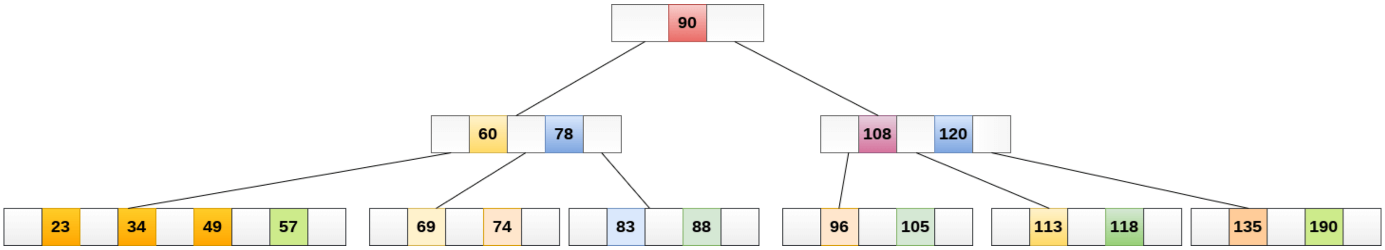


53 is present in the right child of element 49. Delete it.



Now, 57 is the only element which is left in the node, the minimum number of elements that must be present in a B tree of order 5, is 2. it is less than that, the elements in its left and right sub-tree are also not sufficient therefore, merge it with the left sibling and intervening element of parent i.e. 49.

The final B tree is shown as follows.



Application of B tree

B tree is used to index the data and provides fast access to the actual data stored on the disks since, the access to value stored in a large database that is stored on a disk is a very time consuming process.

Searching an un-indexed and unsorted database containing n key values needs O(n) running time in worst case. However, if we use B Tree to index this database, it will be searched in O(log n) time in worst case.

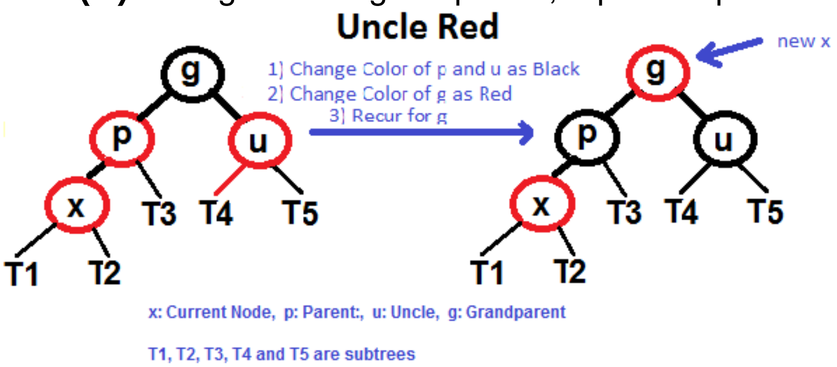
Red Black trees:

In AVL tree insertion, we used rotation as a tool to do balancing after insertion caused imbalance. In Red-Black tree, we use two tools to do balancing.
1) Recoloring
2) Rotation

We try recoloring first, if recoloring doesn't work, then we go for rotation. Following is detailed algorithm. The algorithms has mainly two cases depending upon the color of uncle. If uncle is red, we do recoloring. If uncle is black, we do rotations and/or recoloring.

Color of a NULL node is considered as BLACK.

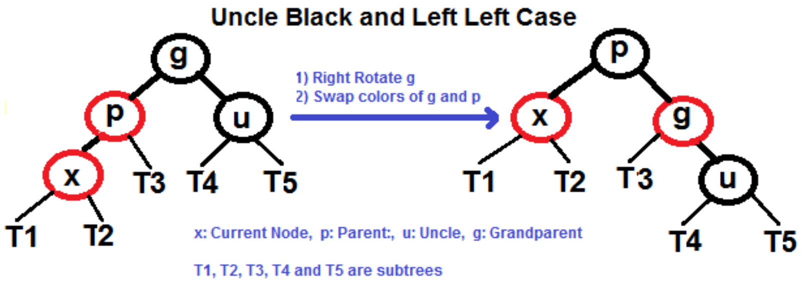
Let x be the newly inserted node.
1) Perform standard BST insertion and make the color of newly inserted nodes as RED.
2) If x is root, change color of x as BLACK (Black height of complete tree increases by 1).
3) Do following if color of x's parent is not BLACK or x is not root.
….**a) If x's uncle is RED** (Grand parent must have been black from property 4)
……..**(i)** Change color of parent and uncle as BLACK.
……..**(ii)** color of grand parent as RED.
……..**(iii)** Change x = x's grandparent, repeat steps 2 and 3 for new x.


Uncle Red
1) Change Color of p and u as Black
2) Change Color of g as Red
3) Recur for g
new x
g
p
u
x
T1
T2
T3
T4
T5
g
p
u
x
T1
T2
T3
T4
T5
x: Current Node, p: Parent:, u: Uncle, g: Grandparent
T1, T2, T3, T4 and T5 are subtrees

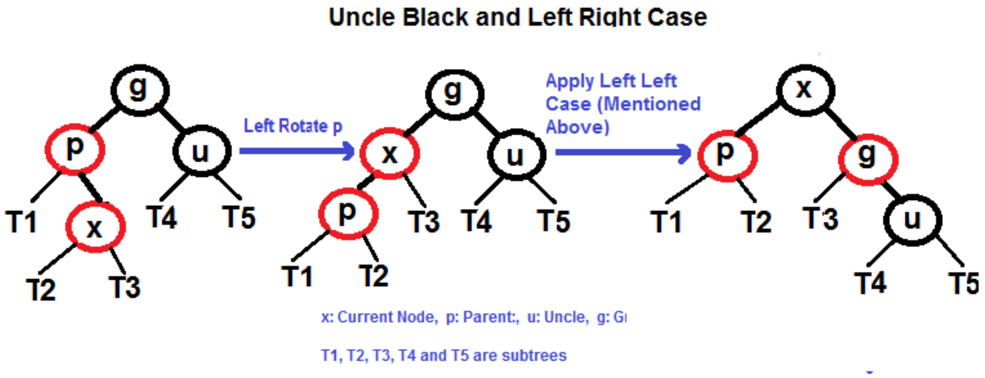
….**b) If x's uncle is BLACK**, then there can be four configurations for x, x's parent (**p**) and x's grandparent (**g**) (This is similar to AVL Tree)
……..**i)** Left Left Case (p is left child of g and x is left child of p)
……..**ii)** Left Right Case (p is left child of g and x is right child of p)
……..**iii)** Right Right Case (Mirror of case a)
……..**iv)** Right Left Case (Mirror of case c)
Following are operations to be performed in four subcases when uncle is BLACK.

All four cases when Uncle is BLACK

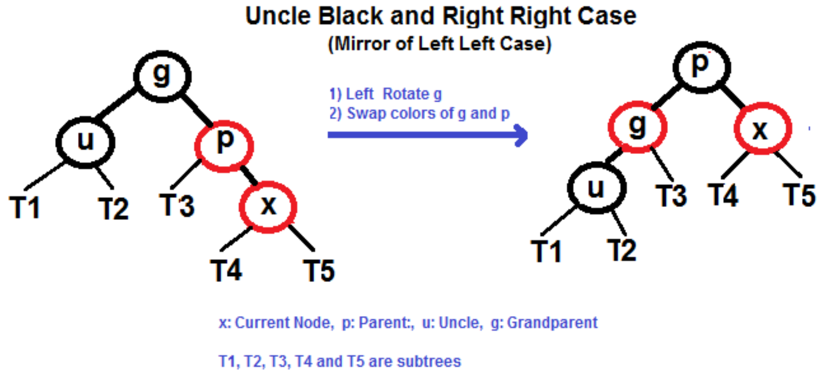
Left Left Case (See g, p and x)


Uncle Black and Left Left Case
g
p
u
x
T1
T2
T3
T4
T5
1) Right Rotate g
2) Swap colors of g and p
p
x
g
T1
T2
T3
u
T4
T5
x: Current Node, p: Parent:, u: Uncle, g: Grandparent
T1, T2, T3, T4 and T5 are subtrees

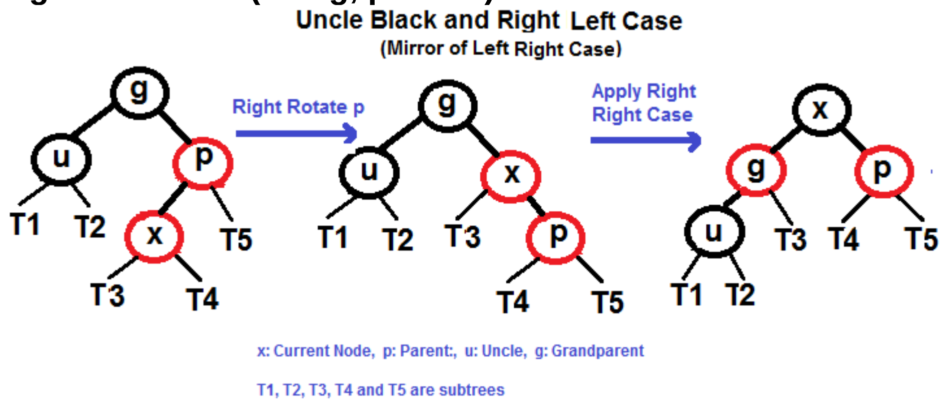
Left Right Case (See g, p and x)


Uncle Black and Left Right Case
g
p
u
T1
x
T4
T5
T2
T3
Left Rotate p
g
x
u
p
T3
T4
T5
T1
T2
Apply Left Left
Case (Mentioned
Above)
x
p
g
T1
T2
T3
u
T4
T5
x: Current Node, p: Parent:, u: Uncle, g: Gı
T1, T2, T3, T4 and T5 are subtrees

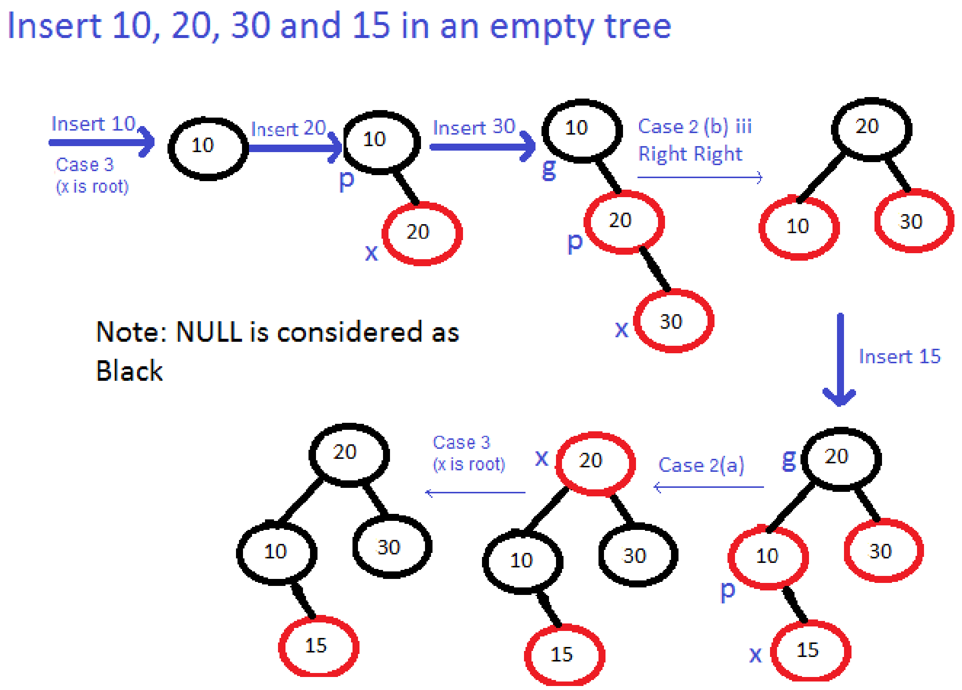
Right Right Case (See g, p and x)


Uncle Black and Right Right Case
(Mirror of Left Left Case)
g
u
p
T1
T2
T3
x
T4
T5
1) Left Rotate g
2) Swap colors of g and p
p
g
x
u
T3
T4
T5
T1
T2
x: Current Node, p: Parent:, u: Uncle, g: Grandparent
T1, T2, T3, T4 and T5 are subtrees

Right Left Case (See g, p and x)


Uncle Black and Right Left Case
(Mirror of Left Right Case)
g
u
p
T1
T2
x
T5
T3
T4
Right Rotate p
g
u
x
T1
T2
T3
p
T4
T5
Apply Right
Right Case
x
g
p
u
T3
T4
T5
T1
T2
x: Current Node, p: Parent:, u: Uncle, g: Grandparent
T1, T2, T3, T4 and T5 are subtrees

Examples of Insertion



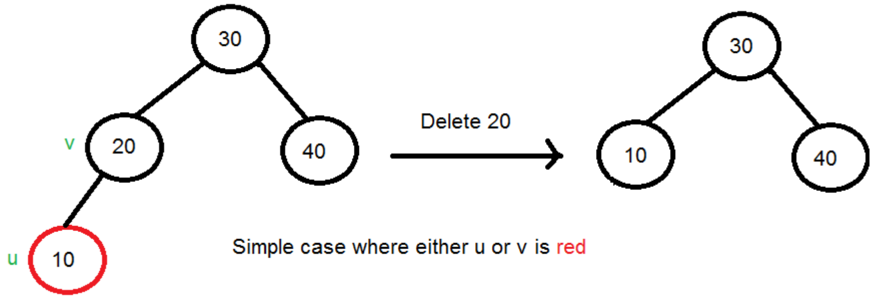
Deletion is fairly complex process. To understand deletion, notion of double black is used. When a black node is deleted and replaced by a black child, the child is marked as ***double black***. The main task now becomes to convert this double black to single black.

Deletion Steps

Following are detailed steps for deletion.

1) Perform standard BST delete. When we perform standard delete operation in BST, we always end up deleting a node which is either leaf or has only one child (For an internal node, we copy the successor and then recursively call delete for successor, successor is always a leaf node or a node with one child). So we only need to handle cases where a node is leaf or has one child. Let v be the node to be deleted and u be the child that replaces v (Note that u is NULL when v is a leaf and color of NULL is considered as Black).

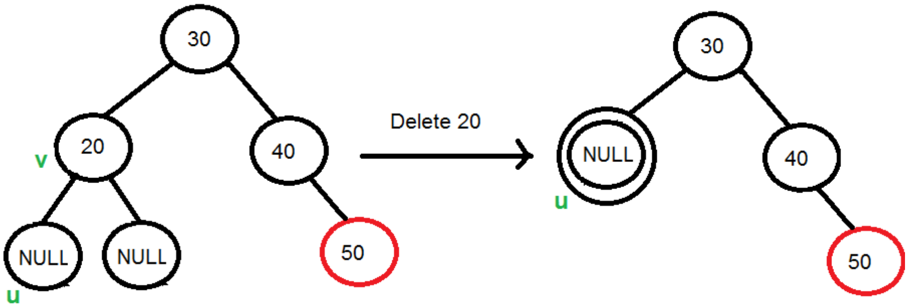
2) Simple Case: If either u or v is red, we mark the replaced child as black (No change in black height). Note that both u and v cannot be red as v is parent of u and two consecutive reds are not allowed in red-black tree.



Simple case where either u or v is red

3) If Both u and v are Black.

3.1) Color u as double black. Now our task reduces to convert this double black to single black. Note that If v is leaf, then u is NULL and color of NULL is considered as black. So the deletion of a black leaf also causes a double black.



When 20 is deleted, it is replaced by a NULL, so the NULL becomes double black.

Note that deletion is not done yet, this double black must become single black

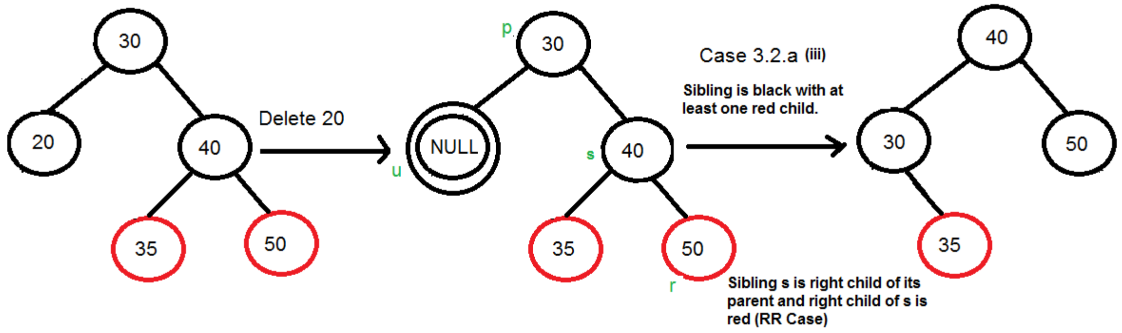
3.2) Do following while the current node u is double black and it is not root. Let sibling of node be **s**.

….**(a): If sibling s is black and at least one of sibling's children is red**, perform rotation(s). Let the red child of s be **r**. This case can be divided in four subcases depending upon positions of s and r.

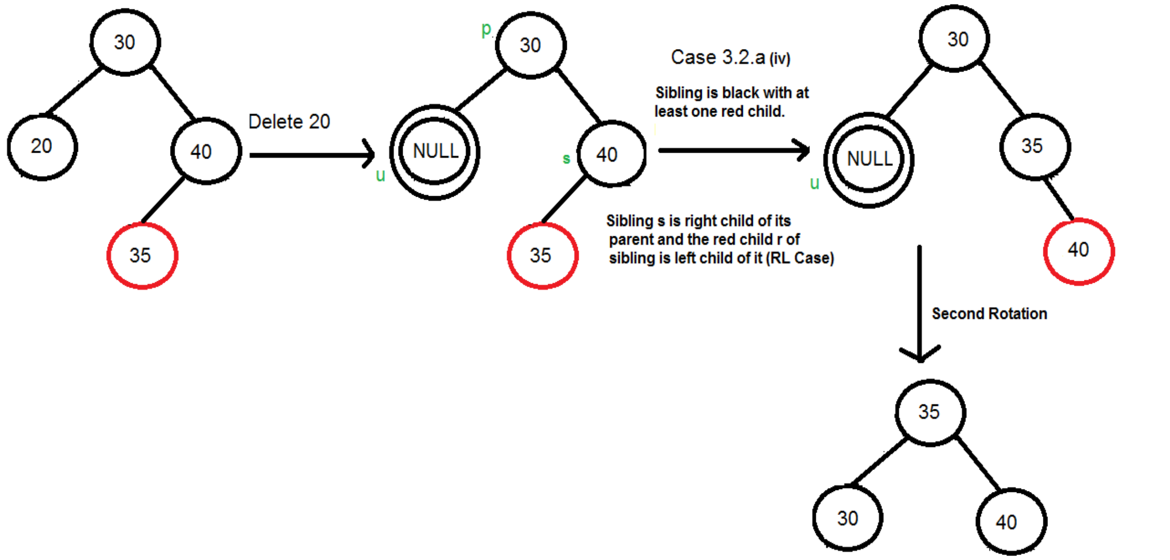
.**(i)** Left Left Case (s is left child of its parent and r is left child of s or both children of s are red). This is mirror of right right case shown in below diagram.

…………..**(ii)** Left Right Case (s is left child of its parent and r is right child). This is mirror of right left case shown in below diagram.

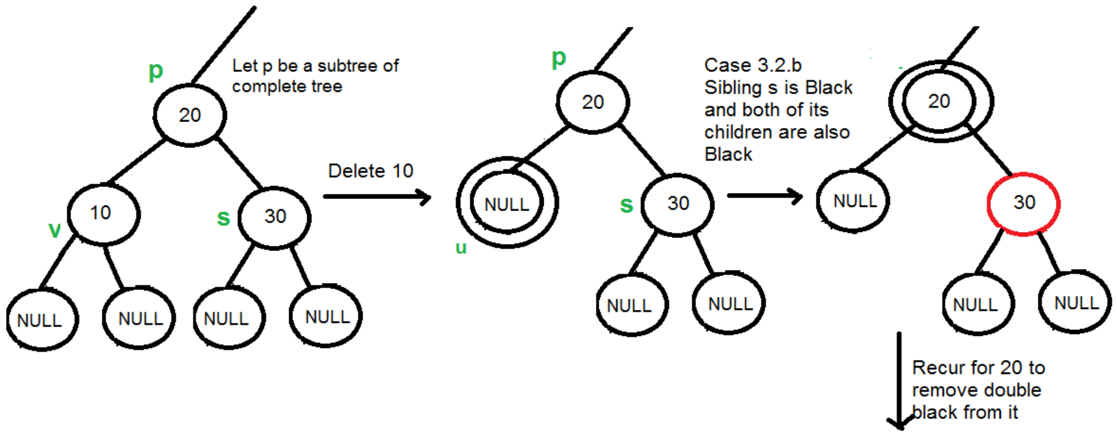
…………..**(iii)** Right Right Case (s is right child of its parent and r is right child of s or both children of s are red)


30
20
40
35
50
Delete 20
p
30
NULL
u
s
40
35
50
r
Case 3.2.a (iii)
Sibling is black with at least one red child.
40
30
50
35
Sibling s is right child of its parent and right child of s is red (RR Case)

…………..**(iv)** Right Left Case (s is right child of its parent and r is left child of s)


30
20
40
35
Delete 20
p
30
NULL
u
s
40
35
Case 3.2.a (iv)
Sibling is black with at least one red child.
Sibling s is right child of its parent and the red child r of sibling is left child of it (RL Case)
30
NULL
u
35
40
Second Rotation
35
30
40

…..**(b): If sibling is black and its both children are black**, perform recoloring, and recur for the parent if parent is black.

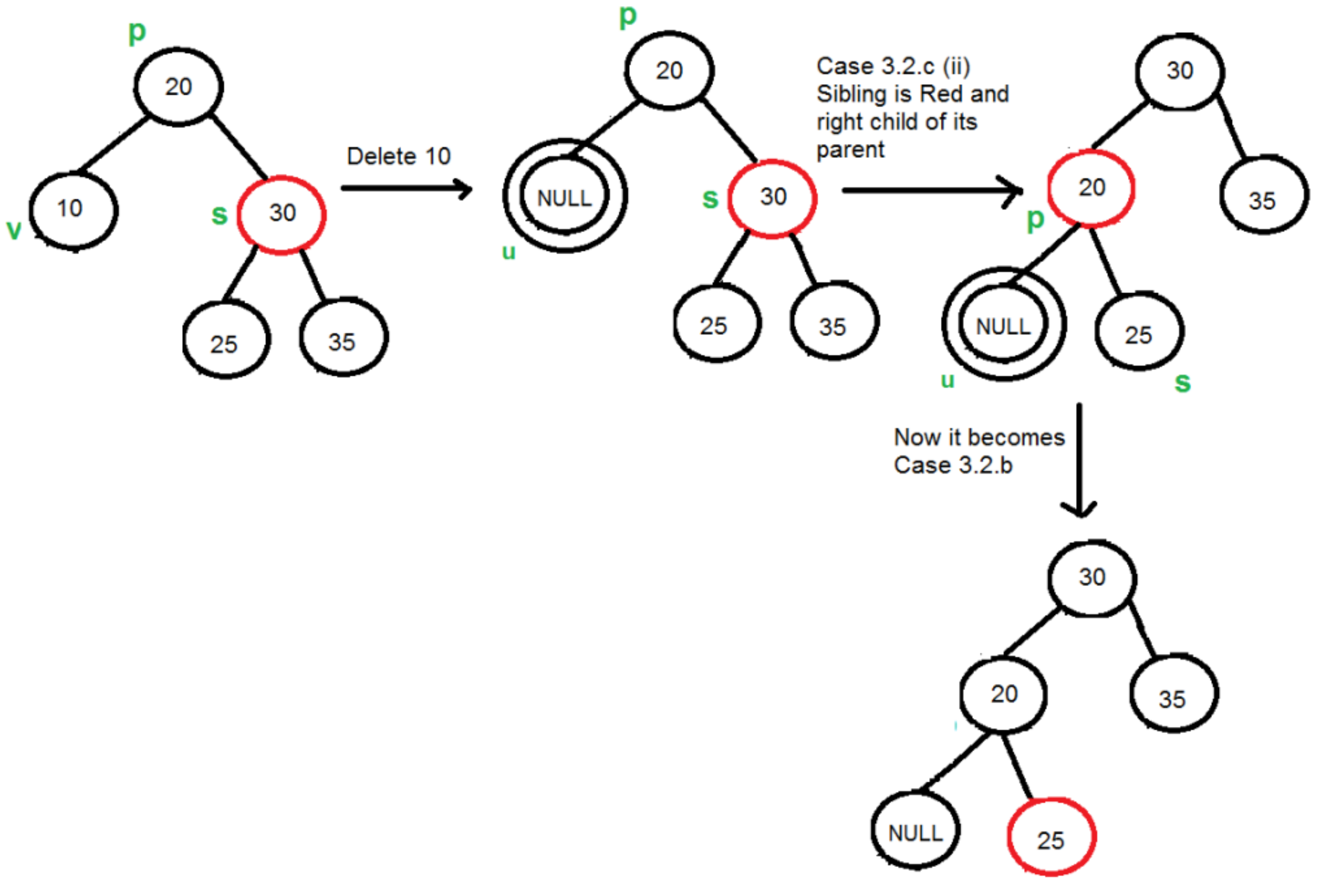

p
Let p be a subtree of complete tree
20
10
v
s
30
NULL
NULL
NULL
NULL
Delete 10
p
20
NULL
u
s
30
NULL
NULL
Case 3.2.b
Sibling s is Black and both of its children are also Black
20
NULL
30
NULL
NULL
Recur for 20 to remove double black from it

In this case, if parent was red, then we didn't need to recur for prent, we can simply make it black (red + double black = single black)

…..**(c): If sibling is red**, perform a rotation to move old sibling up, recolor the old sibling and parent. The new sibling is always black (See the below diagram). This mainly converts the tree to black sibling case (by rotation) and leads to case (a) or (b). This case can be divided in two subcases.

…………..**(i)** Left Case (s is left child of its parent). This is mirror of right right case shown in below diagram. We right rotate the parent p.

………….**(iii)** Right Case (s is right child of its parent). We left rotate the parent p.



3.3) If u is root, make it single black and return (Black height of complete tree reduces by 1).

Comparison with AVL Tree

The AVL trees are more balanced compared to Red Black Trees, but they may cause more rotations during insertion and deletion. So if your application involves many frequent insertions and deletions, then Red Black trees should be preferred. And if the insertions and deletions are less frequent and search is more frequent operation, then AVL tree should be preferred over Red Black Tree.

How does a Red-Black Tree ensure balance?

A simple example to understand balancing is, a chain of 3 nodes is not possible in red black tree. We can try any combination of colors and see all of them violate Red-Black tree property.

A chain of 3 nodes is nodes is not possible in Red-Black Trees.

Following are **NOT** Red-Black Trees

```
     30          30            30
    / \         /  \          /  \
  20  NIL      20   NIL      20   NIL
  / \         / \           /  \
 10  NIL        10  NIL        10  NIL
Violates       Violates      Violates
Property 4.     Property 4     Property 3
```

Following are different possible Red-Black Trees with above 3 keys

```
      20                    20
     /  \                  /  \
   10    30              10    30
  / \  / \              / \   / \
 NIL  NIL NIL NIL       NIL  NIL NIL NIL
```

From the above examples, we get some idea how Red-Black trees ensure balance. Following is an important fact about balancing in Red-Black Trees.

Black Height of a Red-Black Tree :

Black height is number of black nodes on a path from a node to a leaf. Leaf nodes are also counted black nodes. From above properties 3 and 4, we can derive, **a node of height h has black-height >= h/2**.

Every Red Black Tree with n nodes has height <= $2Log_2(n+1)$

This can be proved using following facts:

1) For a general Binary Tree, let **k** be the minimum number of nodes on all root to NULL paths, then $n >= 2^k – 1$ (Ex. If k is 3, then n is atleast 7). This expression can also be written as $k <= 2Log_2(n+1)$

2) From property 4 of Red-Black trees and above claim, we can say in a Red-Black Tree with n nodes, there is a root to leaf path with at-most $Log_2(n+1)$ black nodes.

3) From property 3 of Red-Black trees, we can claim that the number black nodes in a Red-Black tree is at least ⌊ n/2 ⌋ where n is total number of nodes.From above 2 points, we can conclude the fact that Red Black Tree with **n** nodes has height <= $2Log_2(n+1)$