UNIT-1

DATA STRUCTURES AND ALGORITHMS

1.1 PROBLEMS TO PROGRAMS

Half the battle is knowing what problem to solve. When initially approached, most problems have no simple, precise specification. In fact, certain problems, such as creating a "gourmet" recipe or preserving world peace, may be impossible to formulate in terms that admit of a computer solution. Even if we suspect our problem can be solved on a computer, there is usually considerable latitude in several problem parameters

Almost any branch of mathematics or science can be called into service to help model some problem domain. Problems essentially numerical in nature can be modeled by such common mathematical concepts as simultaneous linear equations (e.g., finding currents in electrical circuits, or finding stresses in frames made of connected beams) or differential equations (e.g., predicting population growth or the rate at which chemicals will react). Symbol and text processing problems can be modeled by character strings and formal grammars. Problems of this nature include compilation (the translation of programs written in a programming language into machine language) and information retrieval tasks such as recognizing particular words in lists of titles owned by a library.

Once we have a suitable mathematical model for our problem, we can attempt to find a solution in terms of that model. Our initial goal is to find a solution in the form of an algorithm**, which is 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.**

An integer assignment statement such as x := y + z is an example of an instruction that can be executed in a finite amount of effort. In an algorithm instructions can be executed any number of times, provided the instructions themselves indicate the repetition. However, we require that, no matter what the input values may be, an algorithm terminate after executing a finite number of instructions.

1.2 DATA STRUCTURE:

Data structure is basically a group of data elements that are put together under one name. It also defines a particular way of storing and organizing data in a computer, so that it can be used efficiently.

Data Structure is the structural representation of logical relationships between data or element. It represents the way of storing, organizing and retrieving data.

It is classified as

- **Linear data structure**
- **Non Linear data structure**

Examples:

Linear : Lists, Stacks, Queues etc.

Non Linear : Trees, graphs etc.

Normally all the data structures can be implemented using 2 methods

- Array implementation
- Linked List implementation

Applications of data structure:

- Compiler design
- Statistical analysis package
- Numerical Analysis
- Artificial Intelligence
- Operating System
- Data base management system
- Simulation
- Graphics

1.3 ABSTRACT DATA TYPE

Abstract data type is a set of operations of data. It also specifies the logical and mathematical model of the data type . ADT specification includes description of the data, list of operations that can be carried out on the data and instructions how to use these instructions. But ADT does not mention how the set of operations is implemented.

Examples for ADT: lists, sets and graphs along with their operations for ADT.

Examples for data type: Integers, reals and Booleans.

1.4 ALGORITHM

An ***algorithm*** is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.

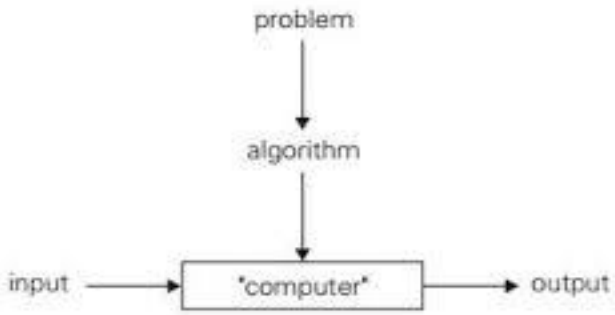


FIGURE 1.1 The notion of the algorithm.

It is a step by step procedure with the input to solve the problem in a finite amount of time to obtain the required output.

1.4.1 The notion of the algorithm illustrates some important points:

- The non-ambiguity requirement for each step of an algorithm cannot be compromised.
- The range of inputs for which an algorithm works has to be specified carefully.
- The same algorithm can be represented in several different ways.
- There may exist several algorithms for solving the same problem.
- Algorithms for the same problem can be based on very different ideas and can solve the problem with dramatically different speeds.

1.4.2 Characteristics of an algorithm:

- **Input:** Zero / more quantities are externally supplied.
- **Output:** At least one quantity is produced.
- **Definiteness:** Each instruction is clear and unambiguous.
- **Finiteness:** If the instructions of an algorithm is traced then for all cases the algorithm must terminates after a finite number of steps.
- **Efficiency:** Every instruction must be very basic and runs in short time.

1.4.3 Steps for writing an algorithm:

1. An algorithm is a procedure. It has two parts; the first part is **head** and the second part is **body**.
2. The Head section consists of keyword **Algorithm** and Name of the algorithm with **parameter list. E.g. Algorithm name1(p1, p2,…,p3)**
3. The head section also has the following:

   //ProblemDescription:

   //Input:

   //Output:
4. In the body of an algorithm various programming constructs like **if, for, while** and some statements like assignments are used.
5. The compound statements may be enclosed with **{** and **}** brackets. **if**, **for**, **while** can be closed by **endif**, **endfor**, **endwhile** respectively. Proper indention is must for block.
6. Comments are written using // at the beginning.
7. The **identifier** should begin by a letter and not by digit. It contains alpha numeric letters after first letter. No need to mention data types.
8. The left arrow **"←" used as assignment operator. E.g. v←10**
9. **Boolean** operators (TRUE, FALSE), **Logical** operators (AND, OR, NOT) and **Relational operators (<,<=, >, >=,=,** ≠, <>) are also used.
10. Input and Output can be done using **read** and **write**.
11. **Array[]**, **if then else condition**, **branch** and **loop** can be also used in algorithm.

Example:

The greatest common divisor(GCD) of two nonnegative integers *m* and *n* (not-both-zero), denoted gcd*(m, n)*, is defined as the largest integer that divides both *m* and *n* evenly, i.e., with a remainder of zero.

Euclid's algorithm is based on applying repeatedly the equality gcd*(m, n)* = gcd*(n, m* mod *n),* where *m* mod *n* is the remainder of the division of *m* by *n*, until *m* mod *n* is equal to 0. Since

gcd*(m, 0)* = *m*, the last value of *m* is also the greatest common divisor of the initial *m* and *n*. gcd*(*60, 24*)* can be computed as follows:gcd*(*60, 24*)* = gcd*(*24, 12*)* = gcd*(*12, 0*)* = 12.

Euclid's algorithm for computing gcd*(m, n)* in simple steps

Step 1 If *n* = 0, return the value of *m* as the answer and stop; otherwise, proceed to Step 2.

Step 2 Divide *m* by *n* and assign the value of the remainder to *r*.

Step 3 Assign the value of *n* to *m* and the value of *r* to *n*. Go to Step 1.

.

Euclid's algorithm for computing gcd*(m, n)* expressed in pseudocode

```
ALGORITHM Euclid_gcd(m, n)
//Computes gcd(m, n) by Euclid's algorithm
//Input: Two nonnegative, not-both-zero integers m and n
//Output: Greatest common divisor of m and n
while n ≠ 0 do
        r ←m mod n
        m←n
        n←r
return m
```

1.5 METHODS OF SPECIFYING AN ALGORITHM

There are three ways to specify an algorithm. They are**:**

a) Natural language
b) Pseudocode
c) Flowchart

a. Natural Language

It is very simple and easy to specify an algorithm using natural language. But many times specification of algorithm by using natural language is not clear and thereby we get brief specification.

Example: An algorithm to perform addition of two numbers

Step 1: Read the first number, say a.

Step 2: Read the first number, say b.

Step 3: Add the above two numbers and store the result in c.

Step 4: Display the result from c.

Such a specification creates difficulty while actually implementing it. Hence many programmers prefer to have specification of algorithm by means of Pseudocode.

b. Pseudocode

- Pseudocode is a mixture of a natural language and programming language constructs. Pseudocode is usually more precise than natural language.

For Assignment operation left arrow **"←"**, for comments two slashes **"//"**,**if** condition**, for, while** loops are used.

ALGORITHM *Sum(a,b)*

//Problem Description: This algorithm performs addition of two numbers

//Input: Two integers a and b

//Output: Addition of two integers

c←a+b

return c

This specification is more useful for implementation of any language.

c. **Flowchart**

In the earlier days of computing, the dominant method for specifying algorithms was a ***flowchart***, this representation technique has proved to be inconvenient.

Flowchart is a graphical representation of an algorithm. It is a a method of expressing an algorithm by a collection of connected geometric shapes containing descriptions of the algor**ithm's steps.**

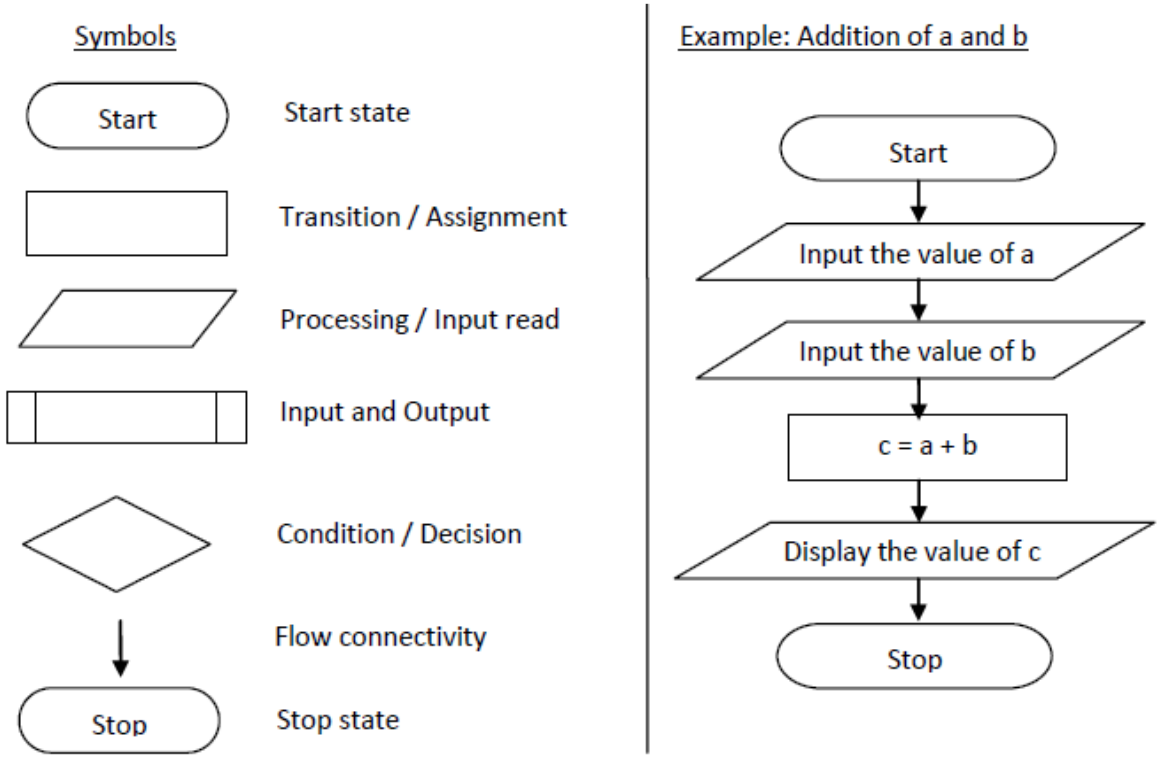


FIGURE 1.4 Flowchart symbols and Example for two integer addition.

1.6 RECURSION

A problem-solving technique in which problems are solved by reducing them to smaller problems of the same form.

In programming, recursion simply means that a function will call itself:

```
int sum() {
        sum();
        return 0;
 }
```

A recursive function always has to say when to stop repeating itself. There should always be two parts to a recursive function:

- the **recursive case** and
- the **base case**.

The recursive case is when the function calls itself.

Base case

- **Base case** in which the problem is simple enough to be solved directly without the need for any more calls to the same function
- The base case is when the function stops calling itself. This prevents infinite loops.

Recursive case

- The problem at hand is initially subdivided into simpler sub-parts.
- The function calls itself again, but this time with sub-parts of the problem obtained in the first step.
- The final result is obtained by combining the solutions of simpler sub-parts.

A recursive function is said to be **well-defined** if it has the following two properties:

- There must be a base criteria in which the function doesn't call itself.
- Every time the function is called itself, it must be closer to the base criteria.

Example:

The power() function: Write a recursive function that takes in a number (x) and an exponent (n) and returns the result of x^n

```
int power(int x,int exp)
{
  If(exp==0)
        return 1;
  else
        return X * power(X, exp-1)
}
```

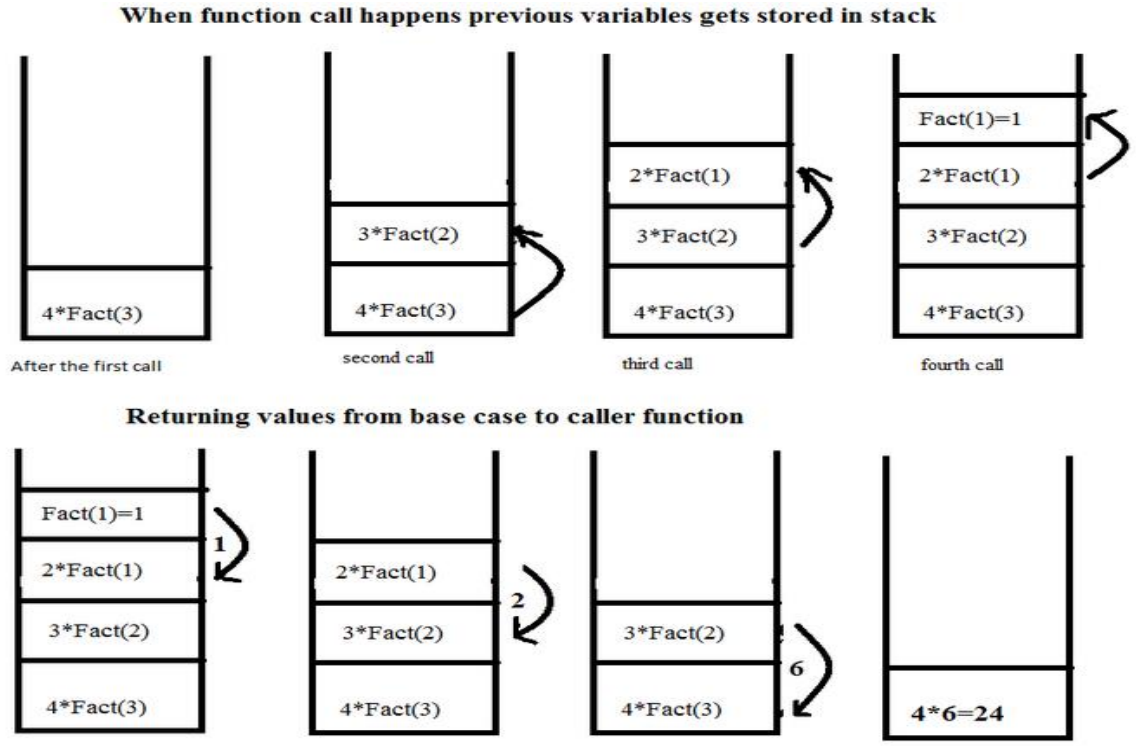
1.6.1 The Call Stack

Recursive functions use something called "the call stack." When a program calls a function, that function goes on top of the call stack. This similar to a stack of books. You add things one at a time. Then, when you are ready to take something off, you always take off the top item. Here is a recursive function to calculate the factorial of a number:

```
function fact(x) {
 if (x == 1) {
```

```
        return 1;
          } else {
              return x * fact(x-1);
              }
    }
```

Now let's see what happens if you call fact(3) The illustration bellow shows how the stack changes, line by line. The topmost box in the stack tells you what call to fact you're currently on.



1.6.2 Example : Towers of Hanoi

Consider the following puzzle

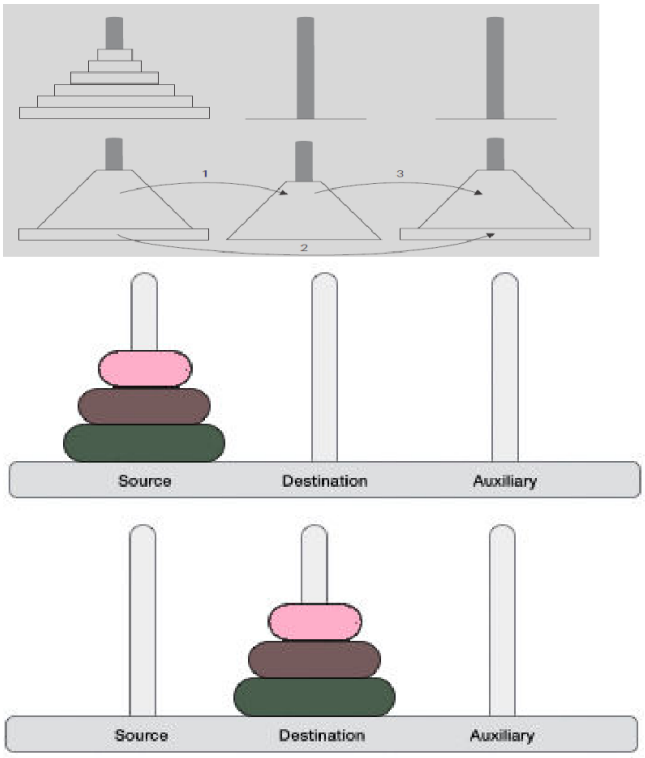
- There are 3 pegs (posts) a, b, c and n disks of different sizes
- Each disk has a hole in the middle so that it can fit on any peg
- At the beginning of the game, all n disks are on peg a, arranged such that the largest is on the bottom, and on top sit the progressively smaller disks, forming a tower
- Goal: find a set of moves to bring all disks on peg c in the same order, that is, largest on bottom, smallest on top

constraints

- the only allowed type of move is to grab one disk from the top of one peg and drop it on another peg

- a larger disk can never lie above a smaller disk, at any time

Consider educational workhorse of recursive algorithms: the ***Tower of Hanoi*** puzzle. We have n disks of different sizes that can slide onto any of three pegs. Consider A (source), B (auxiliary), and C (Destination). Initially, all the disks are on the first peg in order of size, the largest on the bottom and the smallest on top. The goal is to move all the disks to the third peg, using the second one as an auxiliary.



ALGORITHM TOH(n, A, C, B)

//Move disks from source to destination recursively //Input: *n disks and* 3 pegs A, B, and C

//Output: Disks moved to destination as in the source order.

if n=1

Move disk from A to C

else

Move top n-1 disks from A to B using C

TOH(n - 1, A, B, C)

Move top n-1 disks from B to C using A

TOH(n - 1, B, C, A)

Animation 3 disks : https://www.youtube.com/watch?v=5QuiCcZKyYU

Animation :https://www.youtube.com/watch?v=9nwXi9m31RI

Animation 5 disks : https://www.youtube.com/watch?v=BalWjeY2O9g

https://www.youtube.com/watch?v=0u7g9C0wSIA

C program

```
#include <stdio.h>
#include <conio.h>
void hanoi(char,char,char,int);
void main()
{
        int num;
        clrscr();
        printf("\nENTER NUMBER OF DISKS: ");
        scanf("%d",&num);
        printf("\nTOWER OF HANOI FOR %d NUMBER OF DISKS:\n", num);
        hanoi('A','B','C',num);
        getch();
}
void hanoi(char from,char to,char other,int n)
{
        if(n<=0)
                printf("\nILLEGAL NUMBER OF DISKS");
        if(n==1)
                printf("\nMOVE DISK FROM %c TO %c",from,other);
        if(n>1)
        {
                hanoi(from,other,to,n-1);
                hanoi(from,to,other,1);
                hanoi(to,from,other,n-1);
        }
}
```

1.7 COMPLEXITY

Complexity

Suppose M is an algorithm and suppose n is the size of the input data. The efficiency of M is measured in terms of time and space used by the algorithm. Time is measured by counting the number of operations and space is measured by counting the maximum amount of memory consumed by M.

The complexity of M is the function f(n) which gives running time and or space in terms of n. In complexity theory, we find complexity f(n) for three major cases as follows,

- **Worst case**: f(n) have the maximum value for any possible inputs.
- **Average case**: f(n) have the expected value.
- **Best case**: f(n) have the minimum possible value.

These ideas are illustrated by taking the example of the Linear search/ *Sequential Search* algorithm.

ALGORITHM SequentialSearch(A[0..n - 1], K)

//Searches for a given value in a given array by sequential search //Input: An array A[0..n - 1] and a search key K

//Output: The index of the first element in A that matches K or -1 if there are no matching elements

i ←0

while i < n **and** A[i] ≠ K **do**

    i ←i + 1

if i < n

    return i

else

    return -1

Clearly, the running time of this algorithm can be quite different for the same list size n.

In the worst case, there is no matching of elements or the first matching element can found at last on the list. In the best case, there is matching of elements at first on the list.

Worst-case efficiency

- The **worst-case efficiency** of an algorithm is its efficiency for the worst case input of size n.
- The algorithm runs the longest among all possible inputs of that size.
- For the input of size n, the running time is $C_{worst}(n) = n$.

Best case efficiency

- The **best-case efficiency** of an algorithm is its efficiency for the best case input of size n.
- The algorithm runs the fastest among all possible inputs of that size n.
- In sequential search, If we search a first element in list of size n. (i.e. first element equal to a search key), then the running time is $C_{best}(n) = 1$

Average case efficiency

- The Average case efficiency lies between best case and worst case.
- **To analyze the algorithm's average case efficiency, we must make some assumptions about** possible inputs of size n.
- The standard assumptions are that
- The probability of a successful search is equal to p ($0 \le p \le 1$) and
- The probability of the first match occurring in the ith position of the list is the same for every i.

$$C_{avg}(n) = [1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \cdots + i \cdot \frac{p}{n} + \cdots + n \cdot \frac{p}{n}] + n \cdot (1 - p)$$
$$= \frac{p}{n}[1 + 2 + \cdots + i + \cdots + n] + n(1 - p)$$
$$= \frac{p}{n}\frac{n(n+1)}{2} + n(1 - p) = \frac{p(n+1)}{2} + n(1 - p).$$

1.7.1 SPACE TIME TRADE OFF

Space-Time Tradeoff

- Most discussion on algorithm efficiency speaks to run-time, efficiency in CPU utilization. However, especially with cloud computing, memory utilization and data exchange volume must also be considered. This page gives a brief introduction to this topic.
- A **space-time** or **time-memory tradeoff** in computer science is a case where an algorithm or program trades increased space usage with decreased time. Here, **space** refers to the data

torage consumed in performing a given task (RAM, HDD, etc), and **time** refers to the time consumed in performing a given task (computation time or response time).

- The utility of a given space-time tradeoff is affected by related fixed and variable costs (of, e.g., CPU speed, storage space), and is subject to diminishing returns.

History

- Biological usage of time–memory tradeoffs can be seen in the earlier stages of animal behavior. Using stored knowledge or encoding stimuli reactions as "instincts" in the DNA avoids the need for "calculation" in time-critical situations. More specific to computers, look-up tables have been implemented since the very earliest operating systems.
- In 1980 Martin Hellman first proposed using a time–memory tradeoff for cryptanalysis.

Types of tradeoff

i. **Lookup tables vs. recalculation**

The most common situation is an algorithm involving a lookup table: an implementation can include the entire table, which reduces computing time, but increases the amount of memory needed, or it can compute table entries as needed, increasing computing time, but reducing memory requirements.

ii. **Compressed vs. uncompressed data**

A space–time tradeoff can be applied to the problem of data storage. If data is stored uncompressed, it takes more space but access takes less time than if the data were stored compressed (since compressing the data reduces the amount of space it takes, but it takes time to run the decompression algorithm). Depending on the particular instance of the problem, either way is practical. There are also rare instances where it is possible to directly work with compressed data, such as in the case of compressed bitmap indices, where it is faster to work with compression than without compression.

iii. **Re-rendering vs. stored images**

Storing only the SVG source of a vector image and rendering it as a bitmap image every time the page is requested would be trading time for space; more time used, but less space. Rendering the image when the page is changed and storing the rendered images would be trading space for time; more space used, but less time. This technique is more generally known as caching.

iv. **Smaller code vs. loop unrolling**

Larger code size can be traded for higher program speed when applying loop unrolling. This technique makes the code longer for each iteration of a loop, but saves the

computation time required for jumping back to the beginning of the loop at the end of each iteration.

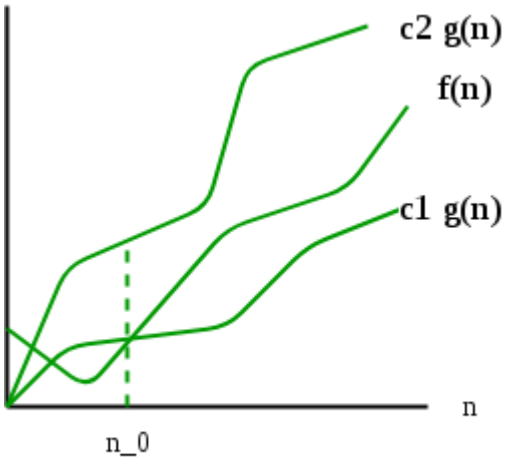
Other examples

- Algorithms that also make use of space-time tradeoffs include:
- Baby-step giant-step algorithm for calculating discrete logarithms
- Rainbow tables in cryptography, where the adversary is trying to do better than the exponential time required for a brute-force attack. Rainbow tables use partially precomputed values in the hash space of a cryptographic hash function to crack passwords in minutes instead of weeks. Decreasing the size of the rainbow table increases the time required to iterate over the hash space.
- The meet-in-the-middle attack uses a space–time tradeoff to find the cryptographic key in only $2n+12n+1$ encryptions (and $O(2n)O(2n)$ space) versus the expected $22n22n$ encryptions (but only $O(1)O(1)$ space) of the naive attack.
- Dynamic programming, where the time complexity of a problem can be reduced significantly by using more memory.

1.8 ASYMPTOTIC NOTATIONS

The main idea of asymptotic analysis is to have a measure of the efficiency of algorithms that don't depend on machine-specific constants and don't require algorithms to be implemented and time taken by programs to be compared. Asymptotic notations are mathematical tools to represent the time complexity of algorithms for asymptotic analysis. The following 3 asymptotic notations are mostly used to represent the time complexity of algorithms.

Θ Notation: The theta notation bounds a function from above and below, so it defines exact asymptotic behavior.



f(n) = theta(g(n))

A simple way to get the Theta notation of an expression is to drop low-order terms and ignore leading constants. For example, consider the following expression.

$3n^3 + 6n^2 + 6000 = \Theta(n^3)$

Dropping lower order terms is always fine because there will always be a number(n) after which $\Theta(n^3)$ has higher values than $\Theta(n^2)$ irrespective of the constants involved.

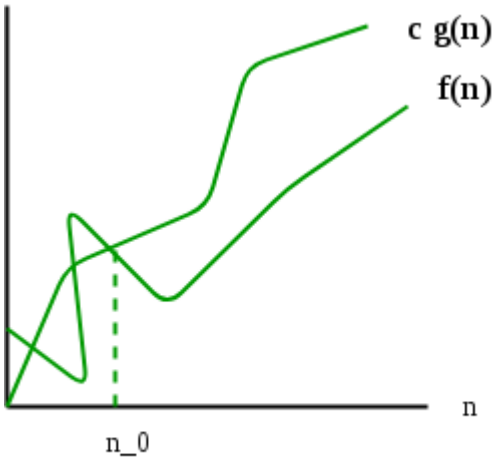
For a given function g(n), we denote $\Theta(g(n))$ is following set of functions.

$\Theta(g(n))$ = {f(n): there exist positive constants c1, c2 and n0 such

that 0 <= c1*g(n) <= f(n) <= c2*g(n) for all n >= n0}

The above definition means, if f(n) is theta of g(n), then the value f(n) is always between c1*g(n) and c2*g(n) for large values of n (n >= n0). The definition of theta also requires that f(n) must be non-negative for values of n greater than n0.

Big O Notation: The Big O notation defines an upper bound of an algorithm, it bounds a function only from above. For example, consider the case of Insertion Sort. It takes linear time in the best case and quadratic time in the worst case. We can safely say that the time complexity of Insertion sort is O(n^2). Note that O(n^2) also covers linear time.



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

If we use Θ notation to represent time complexity of Insertion sort, we have to use two statements for best and worst cases:

1. The worst-case time complexity of Insertion Sort is Θ(n^2).
2. The best case time complexity of Insertion Sort is Θ(n).

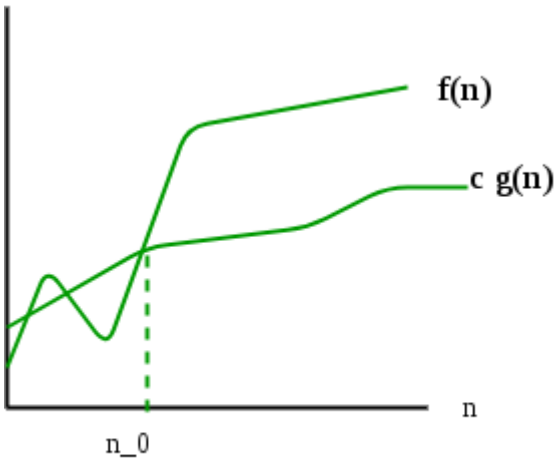
The Big O notation is useful when we only have an upper bound on the time complexity of an algorithm. Many times we easily find an upper bound by simply looking at the algorithm.

O(g(n)) = { f(n): there exist positive constants c and

n0 such that 0 <= f(n) <= c*g(n) for

all n >= n0}

Ω Notation: Just as Big O notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound.



f(n) = Omega(g(n))

Ω Notation can be useful when we have a lower bound on the time complexity of an

algorithm. As discussed in the previous post, the best case performance of an algorithm is generally not useful, the Omega notation is the least used notation among all three.
For a given function g(n), we denote by Ω(g(n)) the set of functions.

Ω (g(n)) = {f(n): there exist positive constants c and

n0 such that 0 <= c*g(n) <= f(n) for

all n >= n0}.

Let us consider the same Insertion sort example here. The time complexity of Insertion Sort can be written as Ω(n), but it is not very useful information about insertion sort, as we are generally interested in worst-case and sometimes in the average case.

UNIT II

Data Structure:

Data structure is basically a group of data elements that are put together under one name. It also defines a particular way of storing and organizing data in a computer, so that it can be used efficiently.

Data Structure is the structural representation of logical relationships between data or element. It represents the way of storing, organizing and retrieving data.

It is classified as

- **Linear data structure**
- **Non Linear data structure**

Examples:

Linear : Lists, Stacks, Queues etc.

Non Linear : Trees, graphs etc.

Normally all the data structures can be implemented using 2 methods

- Array implementation
- Linked List implementation

Applications of data structure:

- Compiler design
- Statistical analysis package
- Numerical Analysis
- Artificial Intelligence
- Operating System
- Data base management system
- Simulation
- Graphics

Abstract Data Type:

Abstract data type is a set of operations of data. It also specifies the logical and mathematical model of the data type . ADT specification includes description of the data, list of operations that can be carried out on the data and instructions how to use these instructions. But ADT does not mention how the set of operations is implemented.

Examples for ADT: lists, sets and graphs along with their operations for ADT.

Examples for data type: Integers, reals and Booleans.

List ADT:

A list is a collection of elements. It can be represented as A_1, A2, A3…...A_N and its size is N. A list with size 0 is called as an empty list.

For any list except the empty list, we say that A_{i+1} follows A_i (i<N) and A_{i-1} precedes A_i(i>1). The first element of the list is A_1 and the last element is A_N. the position of A_i in a list is i.

Some popular operations on the list are

- **Find** which returns the position of the first occurrence of an element or key.
- **Insert** which inserts an element in a given position.
- **Delete** which deletes an element from the list.

- **FindKth** which returns the element in K^{th} position.

Example: Consider 34,12,52,16,12 as a list , then

Find (52) returns 3

After **Insert (70,3)** the list is 34,12,70,52,16,12

After **Delete (52)** the list is 34,12,70,16,12

Implementation of list ADT:

There are 2 ways of implementation

- Using Arrays
- Using Linked list

List ADT –Array based Implementation:

A list can be implemented using an array. An array is a collection of similar data items or elements. In the array implementation maximum size of the array is required earlier. That is the maximum number of elements in the array is fixed and it can be modified only through the code.

Since the maximum size should be known earlier, required high estimate, which wastes considerable space.

- An Array implementation allows **printlist** and **Find** to be carried out in **linear time**.
- **Findkth** takes **constant time**.
- **Insertion** and **Deletion** are expensive.

For Example inserting an element at position 0 (i.e) first positions requires shifting the entire elements one position to right.

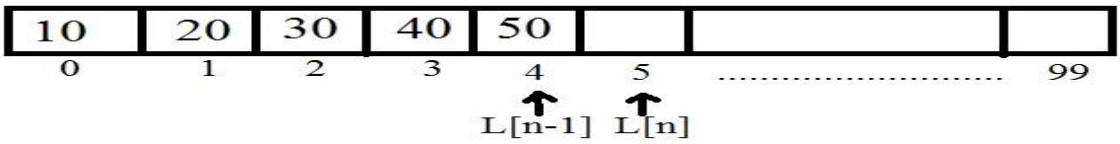
For deleting the first element requires shifting all the elements one position to the left.

Operations:

- Insertion -This inserts a given element X in to the List.
  - Insertion at First.
  - Insertion at any position.
  - Insertion at last.
- Deletion -> deletes a given element from the list
- Find - This returns the position of the first occurrence of an element or key.
- Findkth -returns the element in the k^{th} position.

Consider an array L[100] and insert 'n' elements into the array.

Assume n is 5 and elements are 10,20,30,40,50

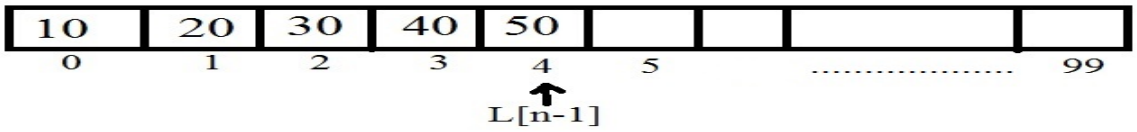


The elements are placed as above and the last element present at location L[n-1] (i.e) L[4] .

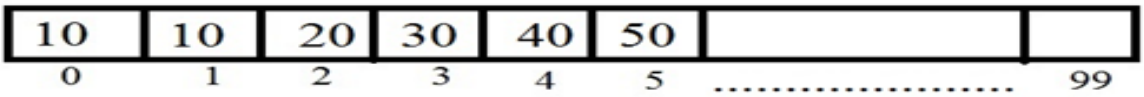
INSERTION

a) **Insertion at First :**

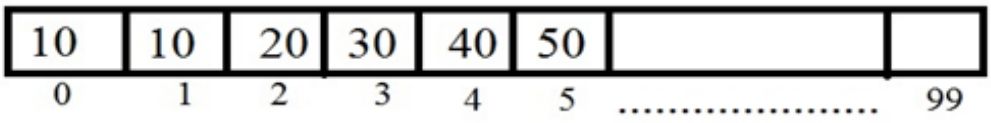
Consider a new element 'X' is to be inserted at the first position (i.e) at L[0]. For this ,shift all the elements to the right one bit position. Let us start shifting from the last location (i.e) L[n-1].



After shifting the array looks like as below.



Now inserts the new element in to first location that is L[0]=X; Assume X=100,Then



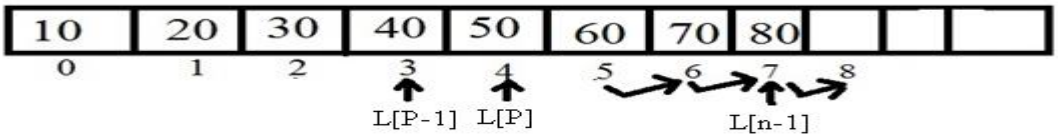
After insertion the no of elements in the list will be increased by one. Therefore n will be updated as n+1 .

Routine:

```
void InsertFirst(int L[],int x,int n)
{
     int n;
     for(i=n-1;i>=0;i--)  //shifting the
           L[i+1]=L[i];    //   elements
     L[0]=x;
    n=n+1;
}
```

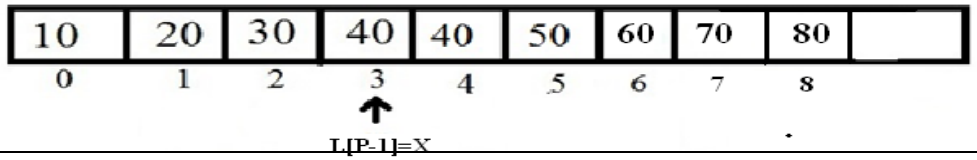
b) Inserting at any position :

Insert an element 'X' at position 'p' , the elements from the next position will be shifted to the right and then the new element will be inserted. Assume p=4 (i.e) 4th position. In array 4th is L[3] .

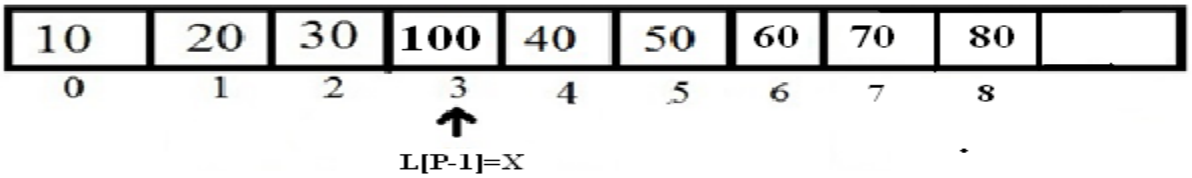


Shift all the elements from p-1th location upto L[n-1].

After shifting, the array looks like as below



Now X can be inserted at position L[p-1] i.e. L[p-1]=X and update the n value i.e. n+1 .Assume the value of X=100,Then the array is

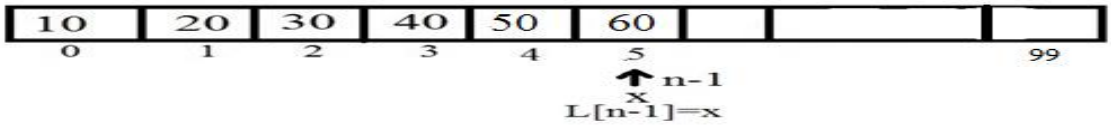


Routine :

```
void InsertAtAny(int L[] , int x , int p , int n)
{
   for(i=n-1;i>p-1;i--)
        L[i+1]=L[i];
   L[p-1]=x;
   n=n+1;
}
```

c) Insertion at Last :

To insert an element at last no shifting is required. Just assign the new element to the position L[n-1].

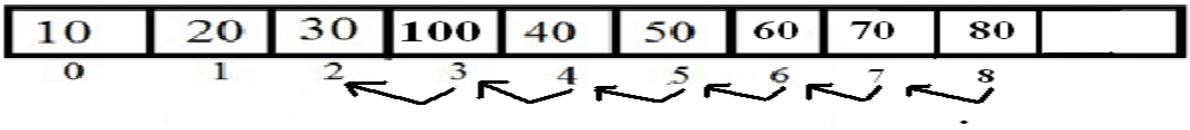


Routine:

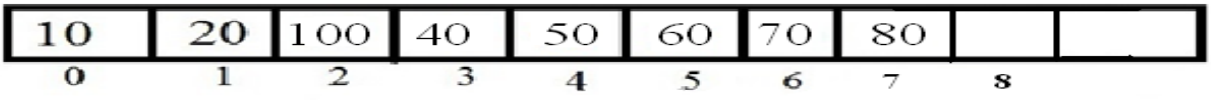
```
void InsertLast(int L[] , int x , int n )
{
   L[n-1]=x;
   n=n+1;
}
```

DELETION:

In deletion a given element will be removed from the list. For this initially the element is searched and the location of the element is identified, then the element is removed by shifting all the elements to the left from next location of the element to be deleted to the next location of the element to be deleted. Assume the element to be deleted 30,



After deletion,



Update the value of n (i.e) n-1.

```
void   Delete(int L[] , int x , int n)
{
   for(i=0;i<n;i++)
       if(L[i]==x)
            break;
  for(j=i+1;j<n;j++)
        L[j-1]=L[j];
  n=n-1;
}
```

FIND:

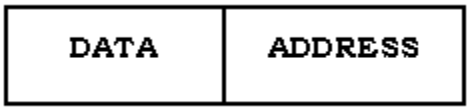
This operation checks whether the given element is present or not .

```
void Find(int L[] , int x , int n)
{
int flag=0;
for(i=0;i<n;i++)
        if(L[i]==x)
  flag=1;
if(flag==1)
  printf("The element is present");
else
  printf("The element is Not Present");

}
```

LINKED LIST:

Linked List is a list, which is implemented using structures and pointers. It can also be defined as group nodes or series of structures where each node has minimum 2 fields.

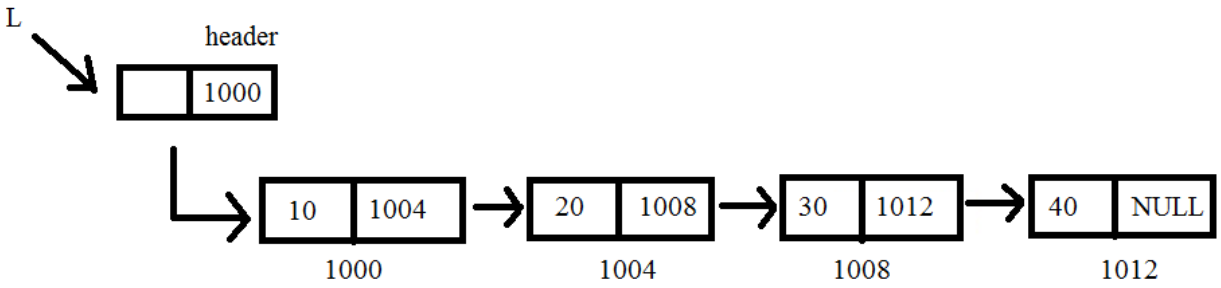
- **Data field :** Stores the actual information
- **Address field :** Points the next node



Node Declaration

```
struct Node
{
    ElementType Element;
    Position Next;
}
```

Example: Singly Linked list



The link field of the last node points to NULL which indicates end of linked list. In order to avoid the linear cost of insertion and deletion, the elements are not stored contiguously in linked list.

Operations

- Insertion
- Deletion
- Find
- FindPrevious
- DeleteList

Advantages:

- Linked lists allow dynamic memory management by allowing elements to be added or deleted at any time during program execution.
- Efficient utilization of memory space. Memory space is reserved only for the elements in a linked list.
- Easy to insert or delete elements in a linked list.

Disadvantages:

- Each element in the linked list requires more space when compared with array.
- Accessing an element is more difficult, Since to access an element it is mandatory to traverse all the preceding elements.

Types of Linked List:

3 types

- Singly Linked List
- Doubly Linked List
- Circularly Linked List

SINGLY LINKED LIST

Singly linked list consists of a series of structures or nodes. Each node contains only 2 fields.

Data field: Actual information

Address field: Address of next node

DATA	ADDRESS

NODE DECLARATION

```
struct Node
{
```

```
    ElementType Element;
    Position Next;
}
```

Type Declarations:

```
    struct Node;
    typedef struct Node *ptrToNode;
    typedef ptrToNode List;
    typedef ptrToNode Position;

    List MakeEmpty(List L);
    int IsEmpty(List L);
    int IsLast(Position P,List L);
    Position Find(ElementType x,List L);
    void Delete(ElementTypr x,List L) ;
    Position FindPrevious(ElementType x.List L);
    void Insert(ElementType x,List L,Position P);
    void DeleteList(List L);
```

OPERATIONS

- Insertion
- Deletion
- Find
- FindPrevious
- DeleteList

INSERTION

Insertion is the process of inserting node at the given position in the linked list. A new node can be inserted after node P. Here, P is the address of the node after which the new node is to be inserted.

For insertion, create a node first from Node structure and assign the value X to its data field.

Newnode = (struct Node *) malloc (sizeof(struct Node))

Newnode → Element = X

- Then perform the following

  Newnode →Next = P → Next

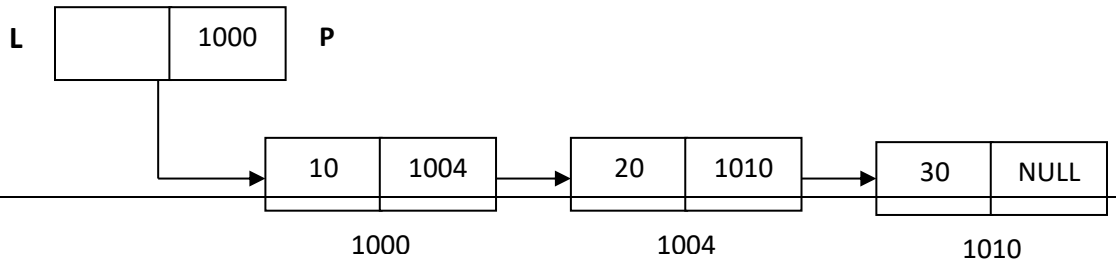
  P → Next= Newnode

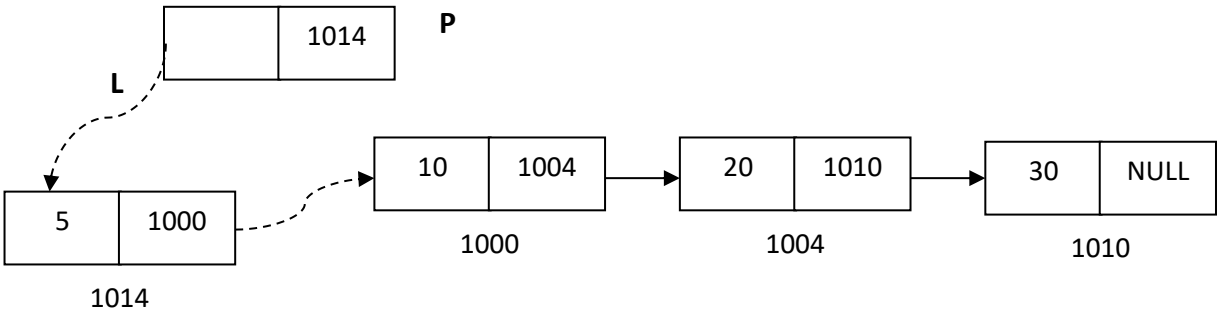
Insertion can be done at any position

a. Insertion at first
b. Insertion at any position
c. Insertion at last

a) INSERTION AT FIRST

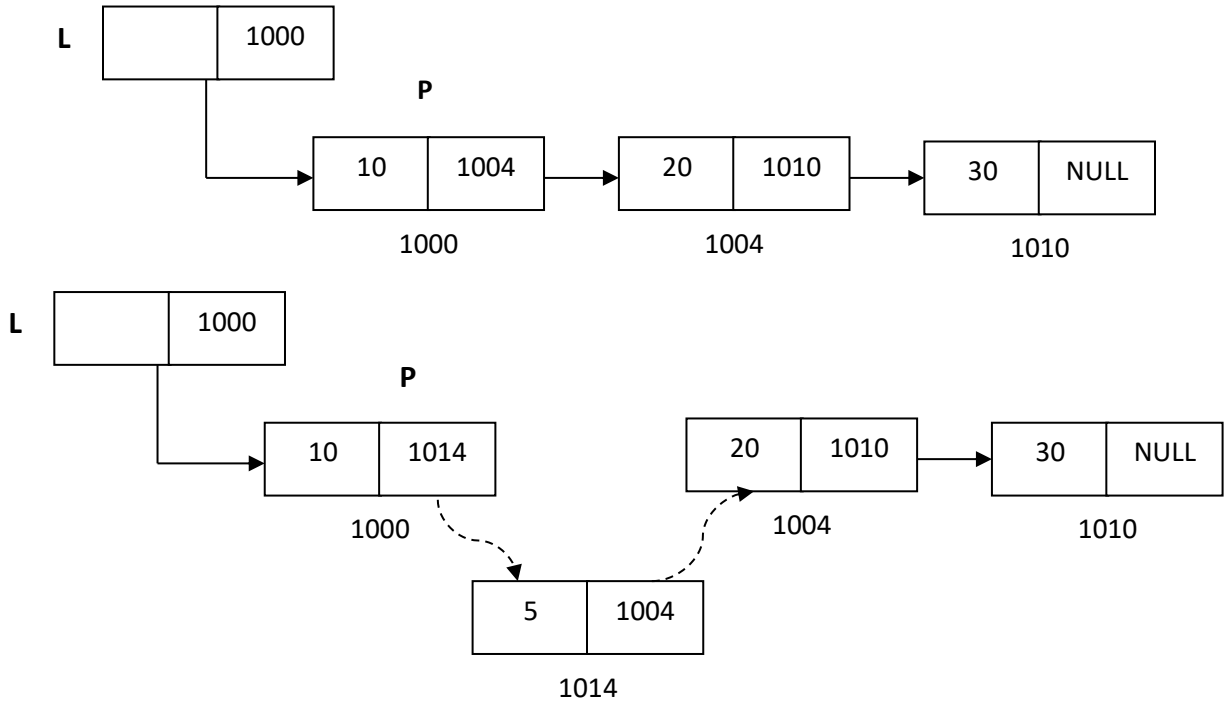
Insert(5,P)





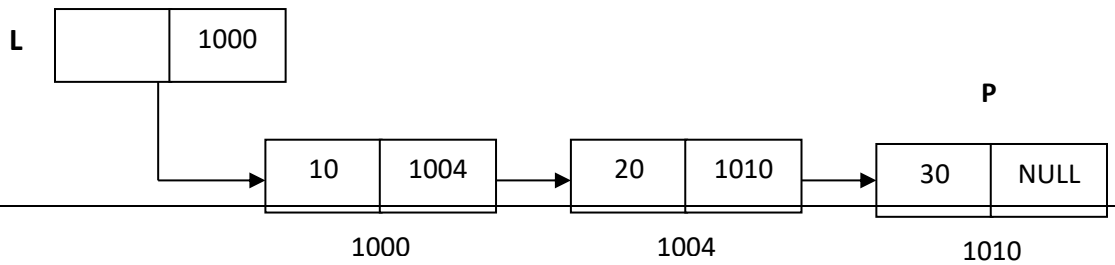
b) INSERTION AT MIDDLE

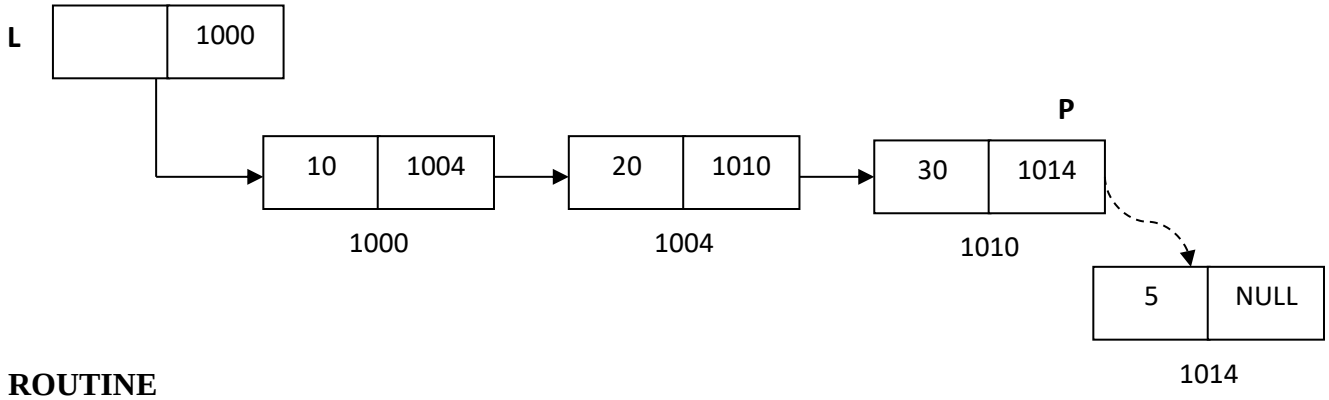
Insert(5,P)



c) INSERTION AT LAST

Insert(5,P)





ROUTINE

Routine to Insert at First	Routine to Insert at Last	Routine to insert at Any Position

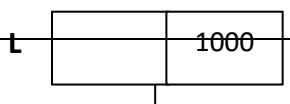
void **InsertFirst**(ElementType X, List L)
{
Position NewNode;
NewNode =malloc (sizeof (struct Node));
if(NewNode==NULL)
FatalError("Out of Space");
else
{
NewNode → Element = X;
NewNode → Next = L → Next;
L → Next=NewNode;
 	
}
}
	void **InsertLast**(ElementType X, List L)
	{
	Position NewNode,P;
	NewNode =malloc (sizeof (struct Node));
	if(NewNode ==NULL)
	FatalError("Out ofSpace");
	else
	{
	P = L → Next;
	while(Temp → Next != NULL)
	P = P → next;
	NewNode → Next=X;
	NewNode → Next=NULL;
	P → Next = NewNode;
	}
	}
		void **Insert** (ElementType X, List L, Position P)
		{
		Position NewNode;
		NewNode =malloc (sizeof (struct Node));
		if(NewNode==NULL)
		FatalError("Out of Space");
		else
		{
		NewNode → Element = x;
		NewNode ->Next = P → Next;
		P → Next = NewNode;
		}
		}

DELETION

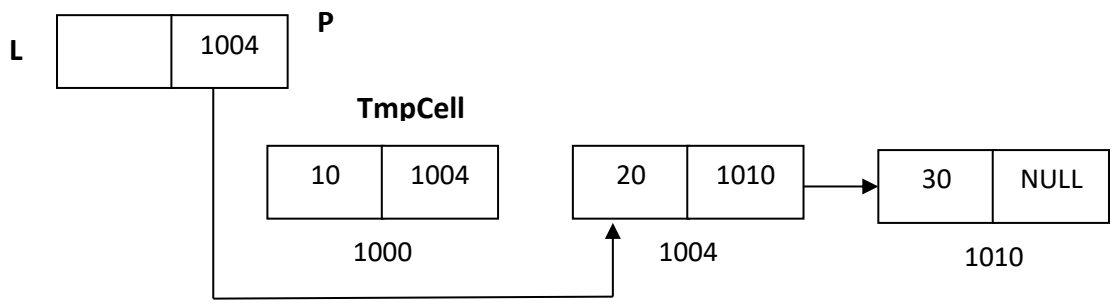
Deletion is the process of removing an element from the linked list. A node which is having the given element X will be removed from the list. For that, address of the previous node which contains X is needed. Address of the previous node of X is identified using the FindPrevious routine and then assign it to P.

- Assign the address of the node to be deleted to TmpCell.
  TmpCell=P→ Next;
- Then do the following
  P→ Next = TmpCell→Next
- After the above assignments the node having X will be removed from the list.
- Deallocate the memory of removed node using **free(TmpCell),**which is a predefined function in C.
- Element from any position can be deleted.
  - At first
  - At last
  - At any position

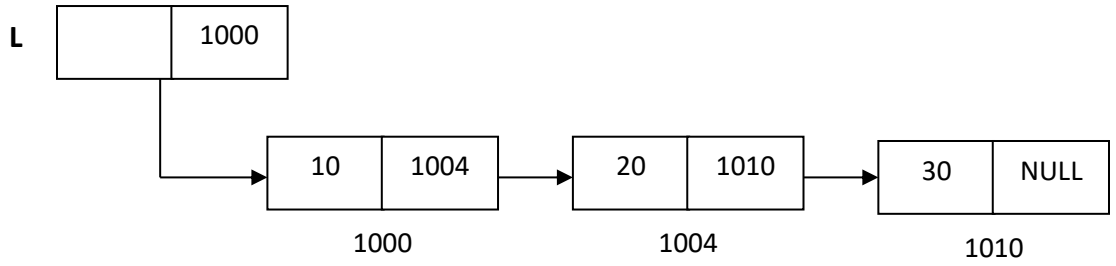
a) DELETION AT FIRST



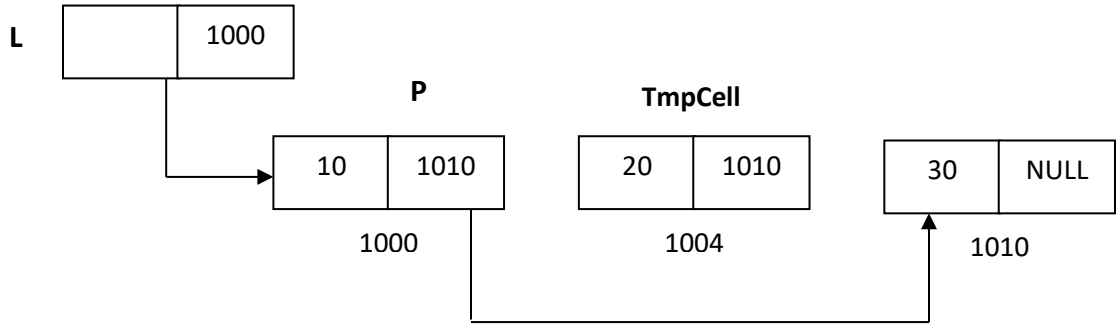
Delete(10)



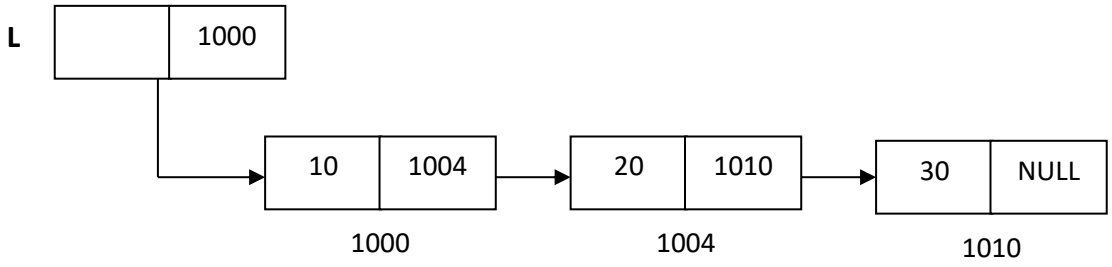
b) DELETION AT ANY POSITION



Delete(20)



c) DELETION AT LAST



Delete(30)

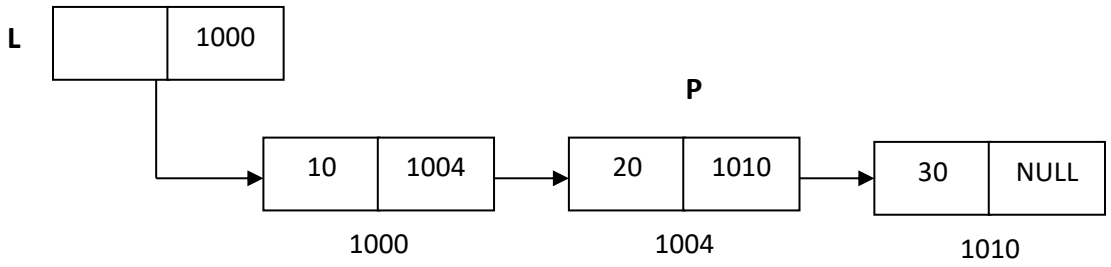
Routine to Delete from First	**Routine to Delete from Last**	Routine to **Delete from** Any Position

void deleteBeginning(List L)
{
Position TmpCell;
if(L → Next!=NULL)
{
TmpCell = L → Next;
L → next = TmpCell → Next;
free(TmpCell);
}
}
	void deleteEnd(List L)
	{
	Position P,TmpCell;
	P=L → **Next;**
	while(P → **Next** → **Next!=NULL**)
	P=P → **Next**;
	TmpCell = P → Next;
	P → **N**ext=NULL;
	free(TmpCell);
	}
		void Delete (ElementType X, List L)
		{
		Position P,TmpCell;
		P = FindPrevious(X,L);
		if(P → Next!=NULL)
		{
		TmpCell = P → Next;
		P → next = TmpCell->Next;
		free(TmpCell);
		}
		}

FIND

Find is the process of identifying the location of particular element in the linked list. This function will return the position of the given element.

Find(20)



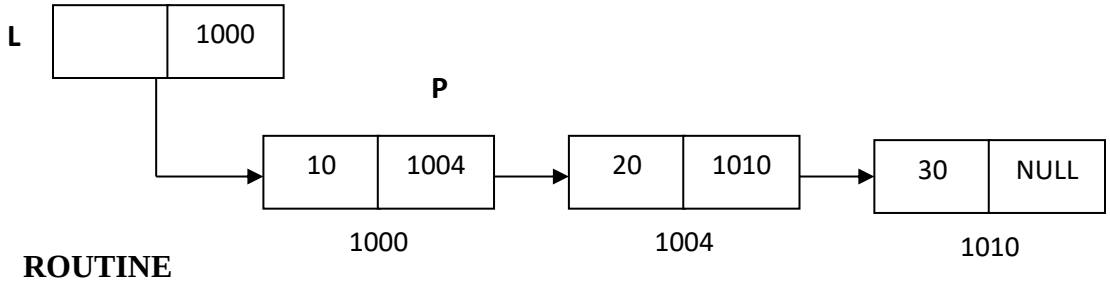
ROUTINE

```
Position Find ( ElementType X, List L )
{
    Position P;
    P = L → next;
    while( P != NULL && P → Element != x) )
        P = P → next;
     return P;
}
```

FIND PREVIOUS

Find is the process of identifying the previous location of particular element in the linked list. This function will return the previous position of the given element.

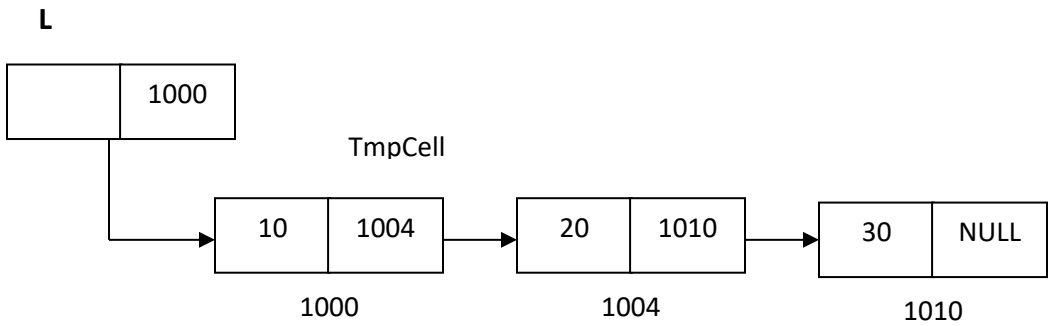
FindPrevious(20)

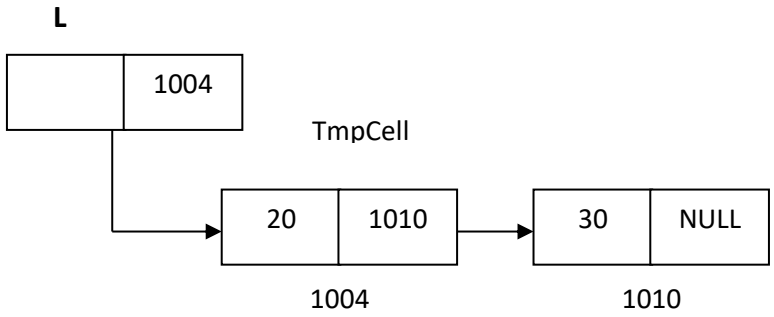


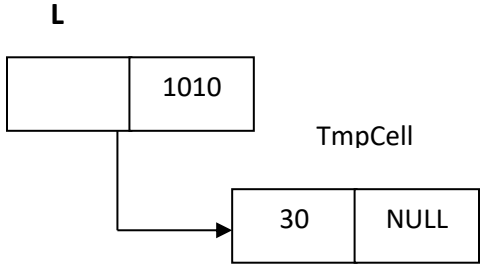
ROUTINE

```
Position FindPrevious ( ElementType X, List L )
{
        Position P;
        P = L;
        while( P → Next != NULL && P → Next → Element != x) )
                P = P → next;
        return P;
}
```

DELETINING ALL THE ELEMENTS IN THE LIST





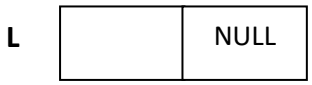


ROUTINE

```
void DeleteList( List L )
```

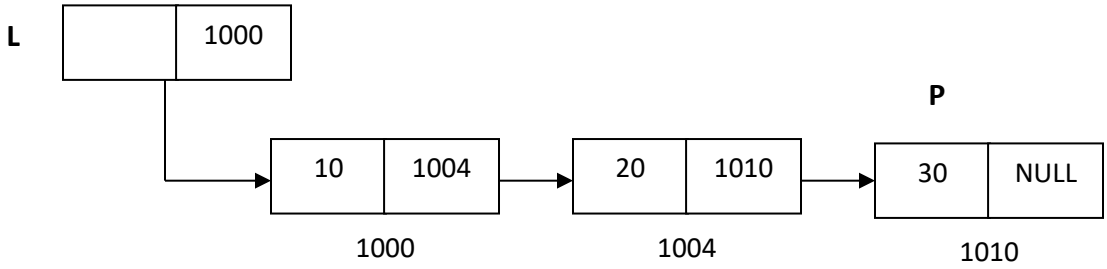
```
{
        position P, tmp;
        P = L → next; /* header assumed */
        L → next = NULL;
        while( p != NULL )
        {
                TmpCell = p → next;
                free( p );
                p = tmp;
        }
}
```

CHECKING WHETHER THE SINGLY LINKED LIST IS EMPTY




```
int  IsEmpty(List L)
{
        return L → Next==NULL;
}
```

CHECKING WHETHER THE GIVEN NODE P IS LAST



ROUTINE

```
int IsLast( Position P , List L )
{
        return( P → Next == NULL );
}
```

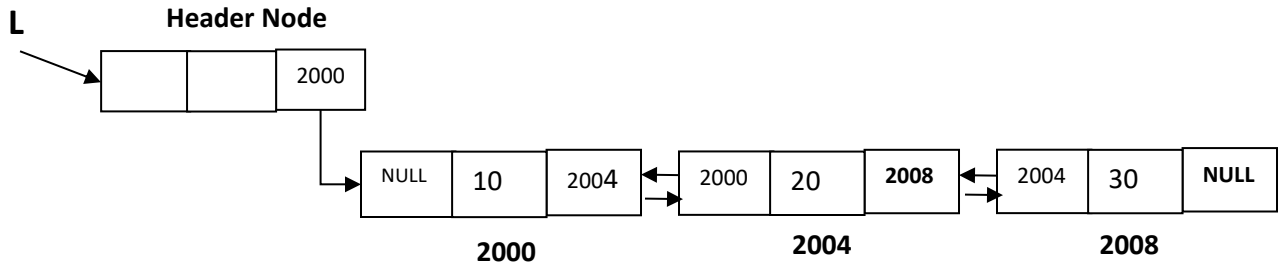
DOUBLY LINKED LIST

Doubly linked list is a list, which is implemented using structures and pointers. Each structure has three fields

- **Data :** Actual Information
- **Pointer to previous structure:** Points the previous node in the list.
- **Pointer to next structure:** Points the next node in the list.

It can be represented as

Example:



Advantages:

Singly linked list cannot be traversed in the backward direction. But it is convenient to traverse the doubly linked list backwards. To do this, an extra field to point the previous structure is added.

Disadvantages:

This requires an extra link to point the previous node which doubles the cost of insertions and deletions because there are more pointers to fix. But it simplifies deletion.

Operations:

The following operations can be performed on a doubly linked list

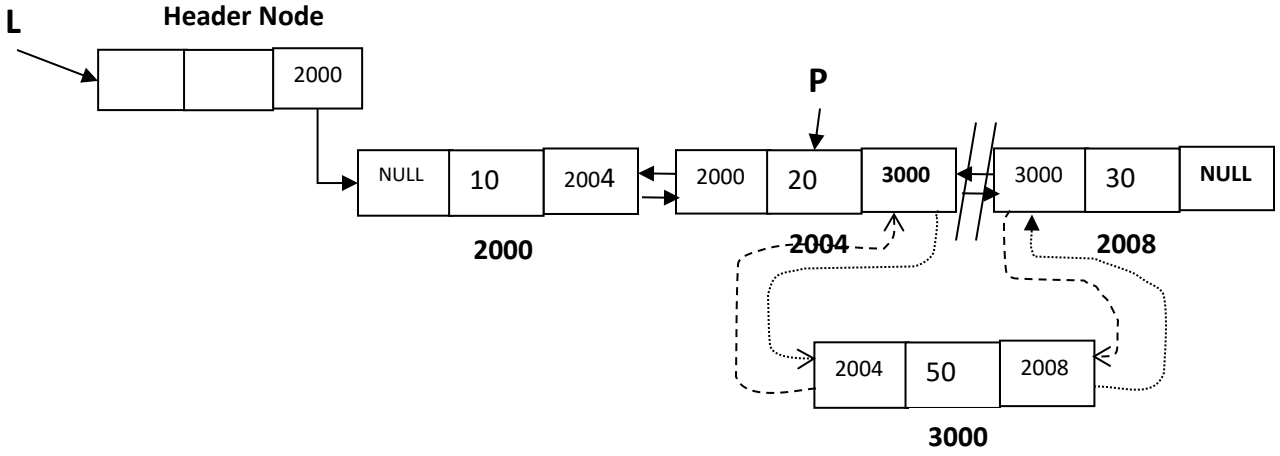
- **Insertion:** Inserts an element in a given position.
- **Deletion:** Deletes an element along with it node.
- **Find:** Finds a given element and returns the address of its node.

INSERTION:

- A new node can be inserted after node P.
- P is the address of the node after which the new node is to be inserted.
- For insertion create a new node in **TmpCell** and assign the ne value X to the node.

  TmpCell = (struct Node *)malloc(sizeof(struct Node))
- Then perform the following

$$\mathbf{TmpCell \rightarrow Next = P \rightarrow Next}$$
$$\mathbf{TmpCell \rightarrow Prev = P}$$
$$\mathbf{P \rightarrow Next = TmpCell}$$
$$\mathbf{P \rightarrow Next \rightarrow Prev = TmpCell}$$

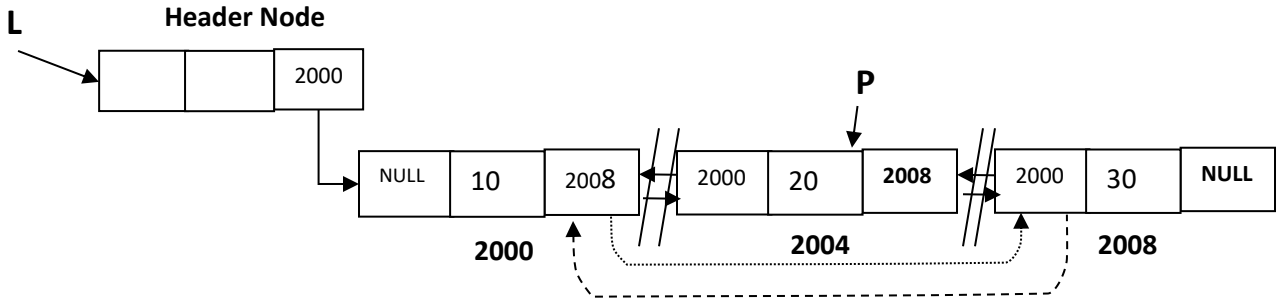


DELETION:

- A node which is having the given element X will be removed from the list.
- Find the address of the node to be deleted by traversing the list and assign the address to P.
- Then do the following

  $P \rightarrow$ **Prev** $\rightarrow$ **Next = P** $\rightarrow$ **Next**

  $P \rightarrow$ **Next** $\rightarrow$ **Prev=P** $\rightarrow$ **Next**
- After the above assignments the node P will be removed from the list.
- Deallocate the memory of removed node using **free(P),**which is a predefined function in C.



FIND:

- Finds a given element X from the list by traversing and returns the address of it.

Programming Details (Routines):

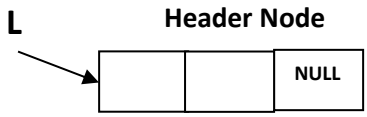
```
struct Node;
typedef struct Node *PtrToNode;
typedef PtrToNode DList;
typedef PtrToNode Position;

int IsEmpty(DList L);
int IsLast(Position P, DList L);
Position Find (Element Type X, DList L);
void Delete (Element Type X, DList L);
Position FindPrevious (Element Type X, DList L);
void Insert (Element Type X, DList L, Position P);
void DeleteList (DList L);

struct Node
{
        ElementType Element;
        Position Next;
        Position Prev;
};
```

Function to test whether the doubly linked list is empty

```
Int  IsEmpty(DList L)
{
        return L → Next==NULL;
}
```

```
if(L → Next==NULL)
        return 1;
else
        return 0;
```

This function checks whether the given list is empty or not. If it is empty it return 1 otherwise 0.

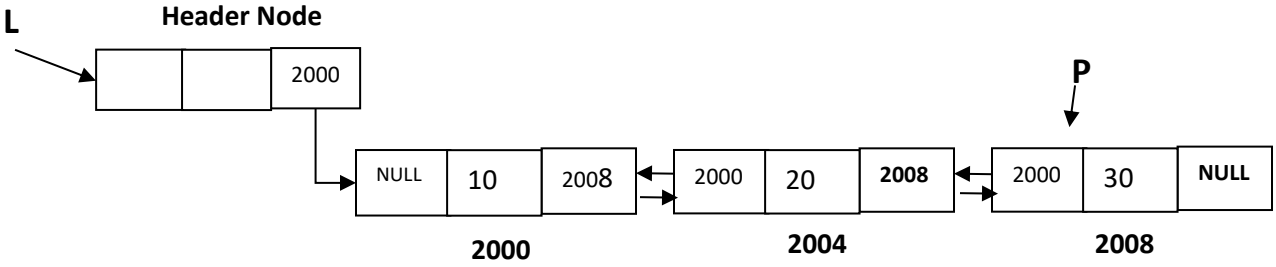


It is done by checking the Next pointer of the header. If it is NULL, then the list is empty

Function to test whether the given node is the last:

```
Int  IsLast(Position P, DList L)
{
        return P → Next==NULL;
}
```

```
if(P → Next==NULL)
        return 1;
else
        return 0;
```



Function for Find operation:

```
Position Find (ElementType X, DList L)
{
        Position P;
        P=L → Next;
        while(P!=NULL && P → Element!=X)
                P=P → Next;
        return P;
}
```

This function returns the address of the node in which the X present

Function for Deletion:

```
Position Delete (ElementType X, DList L)
{
   Position P;
   P=L → Next;
   while(P!=NULL && P → Element!=X)
         P=P → Next;
   P → Prev → Next=P → Next;
    P → Next → Prev=P → Prev;
    free(P)
          }
```

Function for Insertion:

```
void Insert ( ElementType X, List L, Position P )
{
Position NewNode;
NewNode =malloc (sizeof (struct Node));
if(NewNode==NULL)
        FatalError("Out of Space");
else
{
        NewNode  → Element = x;
        NewNode → Prev=P;
        NewNode → Next=P → Next;
        P->Next=NewNode;
        NewNode->Next->Prev=NewNode;
```

Function for DeleteList:

This function deletes all the elements of a list one by one

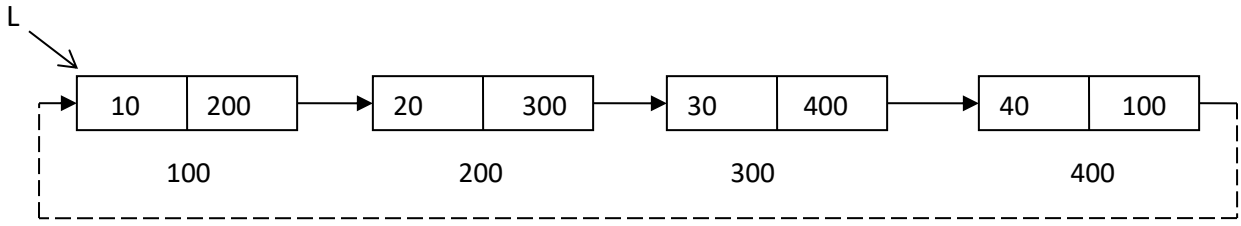
CIRCULAR LINKED LIST

Circular linked list a linked list, in which the next field of the last node points the first node. It can also be done with doubly linked list. It can be done with or without header node

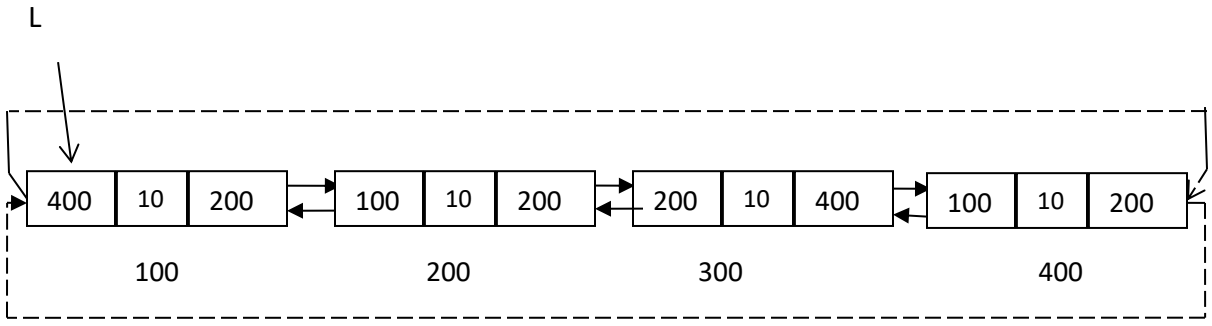
Circular linked lists can be used to traverse the same list again and again if needed. In a circular linked list there are two methods to know if a node is the first node or not.

- Either an external pointer, **List L**, points the first node or
- A header node is placed as the first node of the circular list.

A circularly singly linked list without header

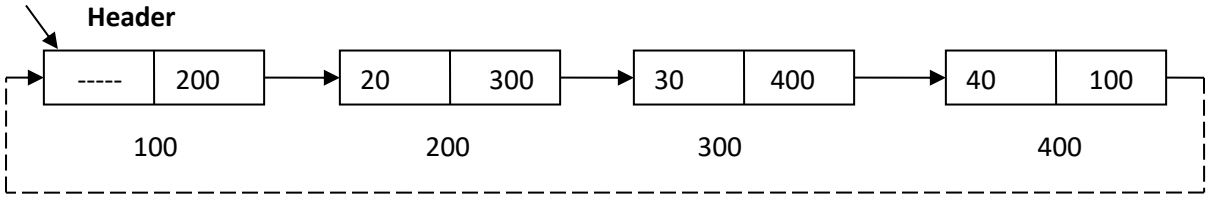


A double circularly linked list without header



A circularly singly linked list with header

L



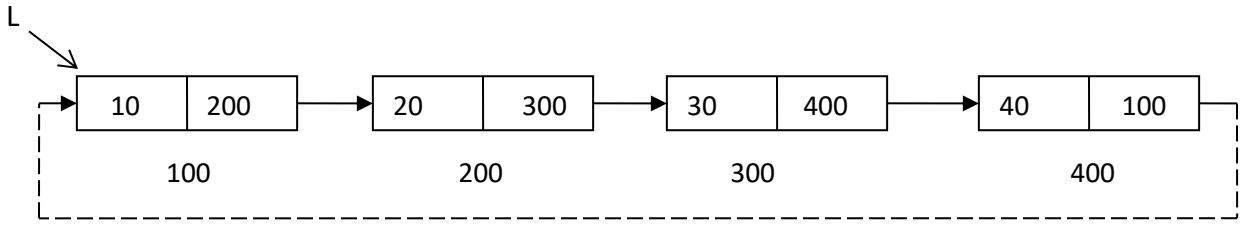
Various Operations

- Insertion
- Deletion
- Find
- Display all the elements

SINGLY CIRCULAR LINKED LIST:

In a normal singly linked list, for accessing any node of linked list, we start traversing from the first node. If we are at any node in the middle of the list, then it is not possible to access nodes that precede the given node. This problem can be solved by slightly altering the structure of singly linked list. In a singly linked list, next part (pointer to next node) is NULL, if we utilize this link to point to the first node then we can reach preceding nodes.

In circularly linked list the pointer of last node points the address of the first node. It allows quick access to the first and last node. It can be implemented using singly linked list or doubly linked list.



OPERATIONS

- Insertion
- Deletion
- Find
- FindPrevious
- DeleteList

Routine to Insert at First	Routine to Insert at Last	Routine to insert at Any Position

```
void InsertFirst( ElementType X, List L)
{
Position NewNode,P;
NewNode=malloc(sizeof(struct Node));
if(TmpCell==NULL)
        FatalError("Out of Space");
else
{
NewNode → Element = x;
NewNode ->Next = L;
L=NewNode;
while(Temp -> Next != L)
    P = P -> next;
P->Next=NewNode;
 }
}
```

```
void InsertLast( ElementType X, List L )
{
Position NewNode ,Temp;
NewNode=malloc (sizeof (struct Node));
if(NewNode ==NULL)
        FatalError("Out of Space");
 else
 {
   Temp = L;
  while(Temp -> Next != L)
     Temp = Temp -> next;
   Temp -> Next = NewNode ;
 NewNode -> next = L;
 }
}
```

```
void Insert ( ElementType X, List L, Position P )
{
Position NewNode;
NewNode =malloc (sizeof (struct Node));
if(NewNode ==NULL)
        FatalError("Out of Space");
else
{
NewNode → Element = x;
NewNode ->Next = P → Next;
P → Next = NewNode;
}
 }
```

Routine to Delete from First

```
void deleteBeginning(List L)
{
    Position TmpCell;
    if(L!=NULL)
    {
        TmpCell = L;
        L = TmpCell → Next;

         P=L;
         while(P → Next!=L)
             P=P → Next
         P->Next=TmpCell->Next;
         free( TmpCell);

    }
}
```

Routine to Delete from Last

```
void deleteEnd(List L)
{
 Position P,TmpCell;
 P=L;
 while(P → Next → Next!=L)
     P=P → Next;
 TmpCell = P → Next;
 P → Next = TmpCell → Next;
 free( TmpCell);
}
```

Routine to **Delete from** Any Position

```
void Delete ( ElementType X, List L )
{
        Position P,TmpCell;
        P = FindPrevious(X,L);
        if(P → Next!=L)
        {
          TmpCell = P → Next;
           P → next= TmpCell->Next;
          free( TmpCell);
        }
}
```

APPLICATIONS OF LIST OR LINKED LIST

1. Polynomial Addition
2. Radix Sort
3. Multi list
4. Buddy system
5. Dynamic memory allocation
6. Garbage Collection

POLYNOMIAL OPERATIONS

A polynomial p(x) is the expression in variable x which is in the form ($ax^n + bx^{n-1} + \ldots. + jx + k$), where a, b, c …., k fall in the category of real numbers and 'n' is non negative integer, which is called the degree of polynomial.

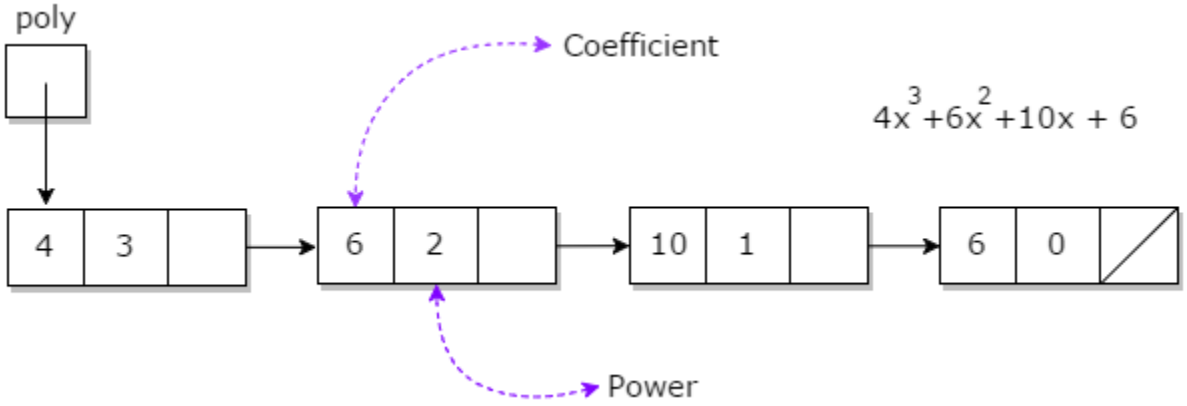
An essential characteristic of the polynomial is that each term in the polynomial expression consists of two parts:

- one is the coefficient
- other is the exponent

$10x^2 + 26x$, here 10 and 26 are coefficients and 2, 1 is its exponential value.

Points to keep in Mind while working with Polynomials:

- The sign of each coefficient and exponent is stored within the coefficient and the exponent itself
- Additional terms having equal exponent is possible one
- The storage allocation for each term in the polynomial must be done in ascending and descending order of their exponent



Polynomial can be represented in the various ways. These are:

- By the use of arrays
- By the use of Linked List

Addition:

```
void polyadd(struct Node *poly1, struct Node *poly2, struct Node *poly)
{
while(poly1 && poly2)
   {
      // If power of 1st polynomial is greater then 2nd, then store 1st as it is and move its pointer
      if(poly1->pow > poly2->pow)
      {


         poly->pow = poly1->pow;
         poly->coeff = poly1->coeff;
         poly1 = poly1->next;
```

```
        }

        // If power of 2nd polynomial is greater then 1st, then store 2nd as it is
        // and move its pointer
        else if(poly1->pow < poly2->pow)
        {
            poly->pow = poly2->pow;
            poly->coeff = poly2->coeff;
            poly2 = poly2->next;
        }

        // If power of both polynomial numbers is same then add their coefficients
        else
        {
            poly->pow = poly1->pow;
            poly->coeff = poly1->coeff+poly2->coeff;
            poly1 = poly1->next;
            poly2 = poly2->next;
        }

       // Dynamically create new node
        poly->next = (struct Node *)malloc(sizeof(struct Node));
        poly = poly->next;
        poly->next = NULL;
    }
while(poly1 || poly2)
    {
        if(poly1)
        {
            poly->pow = poly1->pow;
            poly->coeff = poly1->coeff;
            poly1 = poly1->next;
        }
        if(poly2)
        {
            poly->pow = poly2->pow;
            poly->coeff = poly2->coeff;
            poly2 = poly2->next;
        }
        poly->next = (struct Node *)malloc(sizeof(struct Node));
        poly = poly->next;
        poly->next = NULL;
    }
}
```

Multiplication:

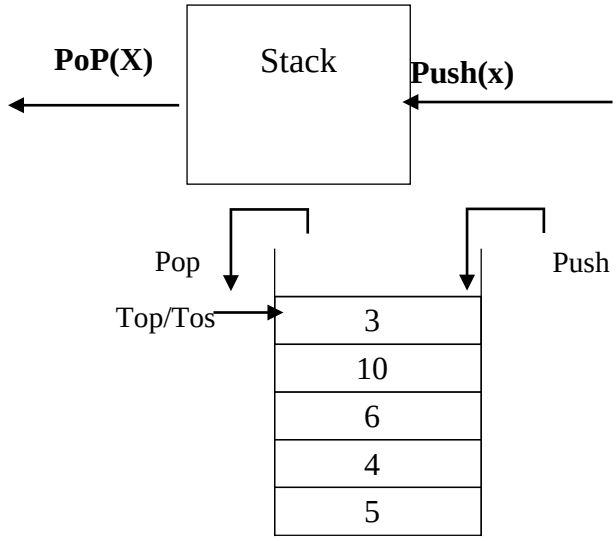
```
Node* multiply(Node* poly1, Node* poly2,
               Node* poly3)
{
```

```
    // Create two pointer and store the
    // address of 1st and 2nd polynomials
    Node *ptr1, *ptr2;
    ptr1 = poly1;
    ptr2 = poly2;
    while (ptr1 != NULL) {
       while (ptr2 != NULL) {
          int coeff, power;

          // Multiply the coefficient of both
          // polynomials and store it in coeff
          coeff = ptr1->coeff * ptr2->coeff;

          // Add the powerer of both polynomials
          // and store it in power
          power = ptr1->power + ptr2->power;

          // Invoke addnode function to create
          // a newnode by passing three parameters
          poly3 = addnode(poly3, coeff, power);

          // move the pointer of 2nd polynomial
          // two get its next term
          ptr2 = ptr2->next;
       }

             ptr2 = poly2;
        ptr1 = ptr1->next;
    }

       removeDuplicates(poly3);
    return poly3;
}
```

Stacks ADT:

- A stack is an ordered list in which all insertions and deletions are made at one end,called the top.
- Stack is a list with the restriction that insertions and deletions can be performed in only one position, namely the end of the list called Top.
- It follows **LIFO** approach. LIFO represents **"Last In First Out".** The basic operations
    - **Push**:Equivalent to insert.
    - **Pop :**Equivalent to delete. It deletes the most recently inserted element.

Stack Model:



The Basic Operations performed in the stack are:

- PUSH()
- POP()
- IsEmpty()
- IsFull()

EXCEPTION CONDITIONS IN STACK

Overflow:

This is the process of trying to insert an element when the stack is full.

Underflow:

This is the process of trying to delete an element when the stack is empty.

Stack can be implemented in following ways

- Array Implementation
- Linked List Implementation

Primitive operations on the stack

- To create a stack
- To insert an element on to the stack.
- To delete an element from the stack.
- To check which element is at the top of the stack.
- To check whether a stack is empty or not.
- To check whether the stack is full or not

Implementation of Stack:

There are two methods of implementing stack operations.

- Array implementation
- Linked List implementation

1.*Array Implementation of Stack*:

Global Declaration:

```
int *stack, top= -1,maxsize;
```

Function to Create Dynamic Stack:

Initially the Stack is declared as a pointer. Using create() function once the maximum size is known the stack array can be dynamically created.

```
void create()
{
      printf("Enter the size of Stack:");
      scanf("%d",&maxsize);
      Stack=(int)malloc(sizeof(int)*maxsize);
}
```

Function to Check IsFull or IsEmpty:

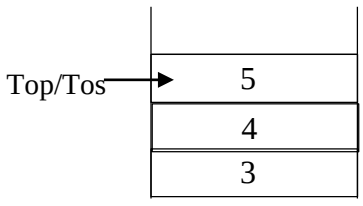
A Pop on an empty stack or a Push on a full stack will overflow the array bounds exception and cause a crash. To avoid this, the following functions are used.

```
int IsFull(int Stack[])
{
      return top==maxsize;
}
int IsEmpty(int Stack[])
{
      return top==-1;
}
```

PUSH Operation:

Inserting an element into the stack is called PUSH operation.

Example: Push 3,4,5 into the stack.



Function for Push Operation: Inserts an element into the stack.

```
void Push(int Stack[],int x)
{
      if(IsFull(Stack))
      printf("\nStack Overflow");
      else
      {
            top++;
            Stack[top]=x;
      }
}
```

Explanation: If top is maxsize, (Maxsize is the maximum size of the stack) it implies that the stack is full; no more elements can be added into the stack. Otherwise, increment top by 1. Store the data in stack[top].

Function for POP Operation:

Delete Operation is called POP operation in stack.

```
void Pop(int Stack[])
{
      if(IsEmpty(Stack))
            printf("\nStack underflow/empty");
      else
          top--;
}
```

Explanation:

If top=-1, it implies that the stack is empty; no element can be deleted from the stack.
Otherwise, decrement top by 1.

Function for Display Operation:

```
void display()
{
      int i;
      if(top==-1)
            printf("\n Stack is empty");
                  else
                  {
                        printf("\nThe elements in the stack are \n");
                        for(i=top;i>=0;i--)
                              printf("%d\t",Stack[i]);
                  }
               }
```

Explanation:

If top =-1, it implies that there are no elements in the stack; stack is empty.
Otherwise, display each element in the stack by formulating a loop; where i is initialized to zero, i is incremented till it reaches top.

Disadvantages of stack using array implementation:

- The size of the Stack is limited.
- If an element is to be inserted into the stack and if the array is fully utilized then the stack will overflow.

2. Linked List Implementation:

Code to Create a Stack:

```
struct Stack
{
      int Element;
      struct Stack *Next;
};
struct Stack *Ptr;
typedef struct Stack *STACK;
STACK create()
{
      STACK S;
      S=(STACK)malloc(sizeof(struct Stack));
      S->Next=NULL;
      return S;
}
```

The structure Stack is declared globally so that all the functions can access the structure. The pointer S->Next represents the address of top element of the Stack.

PUSH Operation:

```
void  Push(int x,STACK S)
{
      STACK TmpCell;
      TmpCell=(STACK)malloc(sizeof(struct Stack));
      TmpCell->Element=x;
      TmpCell->Next=S->Next;
      S->Next=TmpCell;
}
```

Explanation: This function creates a new node called TmpCell. The data is stored in the Element part of the TmpCell. The first element pointed by S-> Next is stored in TmpCell->Next. The address of TmpCell is stored in S->Next.

Deletion operation:

```
int IsEmpty(STACK S)
{
      return S==NULL;
}
void Pop(STACK S)
{
      if(IsEmpty(S))
            printf("\nStack is empty");
      else
            S->Next=S->Next->Next;
}
```

Explanation: First it checks whether Stack is Empty. If yes, it implies that the stack does not contain any nodes. Otherwise, Stack pointer S moved to Next pointer. It is only a logical deletion and not physical deletion.

Display operation

```
void display(STACK S)
{
      if(IsEmpty(S))
            printf("\nStack is empty.");
      else
      {
            printf("\nThe elements in the stack are \n");
            for(Ptr=S->Next;Ptr!=NULL;Ptr=Ptr->Next)
                  printf("%d\t",Ptr->Element);
      }
}
```

Explanation: This function displays all the elements in the element part of the nodes one after another.

Applications of Stack

1. Some of the applications which uses Stacks are,
2. Balancing Symbols
3. Evaluation of postfix expression
4. Conversion of infix to postfix expression
5. Function calls.

1. Balancing Symbols:

Compilers checks the program for syntax errors. During this process it checks for balancing of, parenthesis, braces and brackets. The number left parenthesis should be equal to the number of right parenthesis in the expression.

A stack is used to balance the parenthesis of the given expression .The simple algorithm uses a stack is as follows .

1. Make an empty stack. Read characters until end of file.
2. If the character is an opening symbol, push it onto stack.
3. If it is a closing symbol, then if the stack is empty, report an error. Otherwise, pop the stack.
4. If the symbol popped is not the corresponding opening symbol, then report an error.
5. At the end of file, if the stack is not empty, report an error.

2. Evaluation of Postfix Expressions:

Expression:

The collection of operators and operands are called expression.

- Operands: numbers or alphabets
- Operators: +, -, *, /
- Parenthesis: ( )
- Precedence:
- '(' and ')' have the highest precedence
- '*' and '/ 'have lower precedence than '(' and ')'
- '+' and '-' have lower precedence than '*' and '/'

Types of expression:

Infix: (operand 1) operator (operand 2)
Prefix: operator (operand 1) (operand 2)
Postfix: (operand 1) (operand 2) operator

The expressions in which the operators are placed at the end of operands on which they are going to operate is called **Postfix expression.**

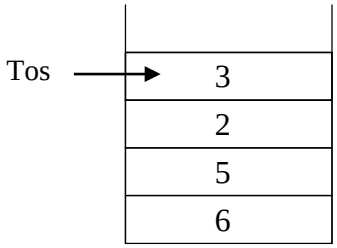
Algorithm to evaluate postfix expression:

1. When an operand is seen, it is pushed on to the stack.
2. When an operator is seen, the operator is applied to the two numbers that are popped from the top of the stack, and return the result pushed back on to the stack.

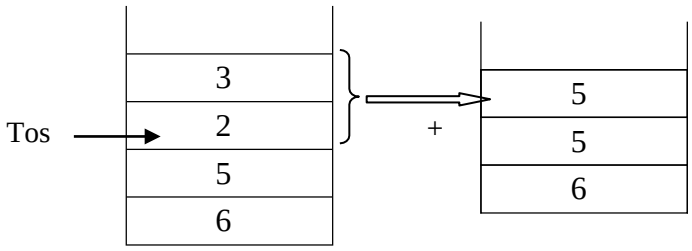
e.g : 6 5 2 3 + 8 * + 3 + *

Consider TOS as a stack pointer.

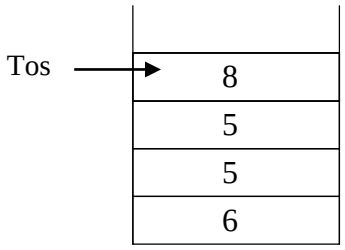
i. First four numbers are operand. So push it on to the stack.



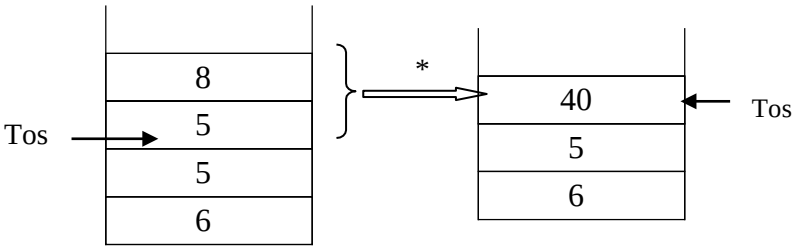
ii. Next symbol is a operator , '+'. So, pop the most recent two operands from the stack and do the operation. i.e., (3+2) = 5 and return the result(5) back to stack.



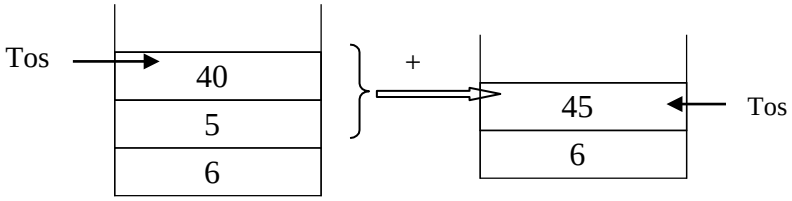
iii. Push the operand ‘8’ to stack.



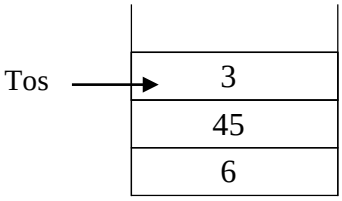
iv. Next symbol is a operator, ‘*’ so pop two elements from the stack and apply the operation i.e., 8*5=40 then push the result 40 again to the stack.



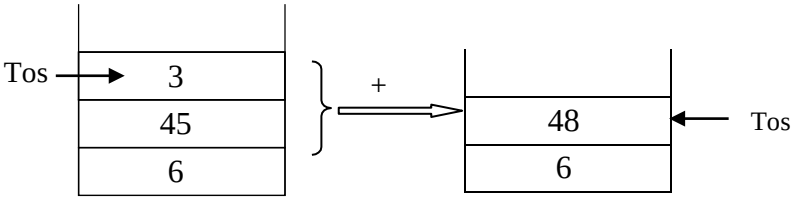
v. Next symbol is a operator, ‘+’ => (40+5 = 45).



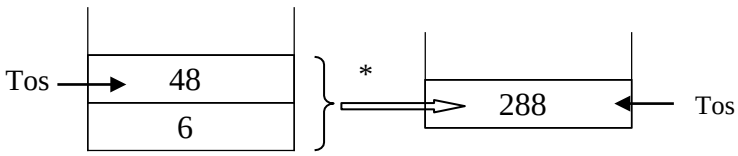
vi. Next symbol is a operand push it on to the stack.(i.e. 3)



vii. Next symbol is a operator, ‘+’ => (3+45 =48)



viii. Next symbol is a operator, ‘*’ => (48 * 6 = 288)



After Evaluation, The Compiler will result 288 as output.

3. Infix to postfix Conversion

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

Algorithm :

(1) Initialize a stack to empty.

(2) Read a symbol

(2.1) If it is an operand, place onto the output.

(2.2) If it is an operator

(a) If it is a right parenthesis, then pop the stack, write symbols, until a left parenthesis is encountered.

Remark: The '(' is popped, but not written as output.

(b) If it is any other symbol, such as '+', '*', '(', pop entries from the stack until an entry of lower priority is found, or '(' is found. When the popping is done, push the operator onto the stack.

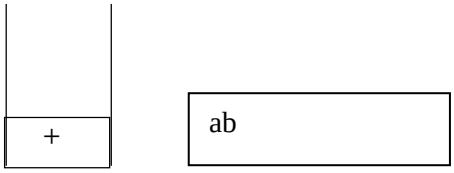
Remark: If '(' is found, don't pop it.

(3) Go to step (2).

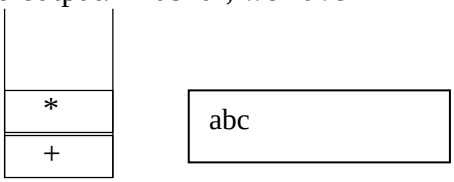
(4) If read the end of input, pop the stack until it is empty, writing symbols onto the output.

Input : a + b*c +(d*e+f)*g

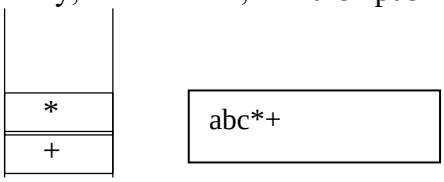
➢ First, the symbol 'a' is read, so it is passed through to the output. Then '+' is read and pushed onto the stack. Next b is read and passed through to the output.



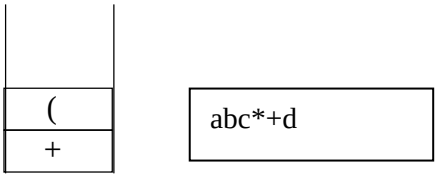
➢ Next a '*' is read. The top entry on the operator stack has lower precedence than '*', so nothing is output and '*' is put on the stack. Next, c is read and output. Thus far, we have



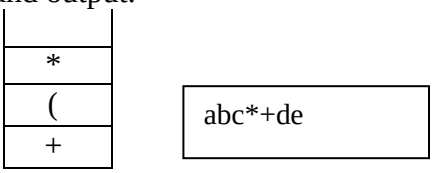
➢ The next symbol is a '+'. Checking the stack, we find that we will pop a '*' and place it on the output; pop the other '+', which is not of lower but equal priority, on the stack; and then push the '+'.



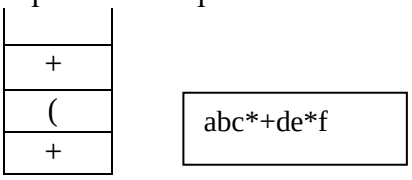
➢ The next symbol read is a '(', which, being of highest precedence, is placed on the Stack Then d is read and output.



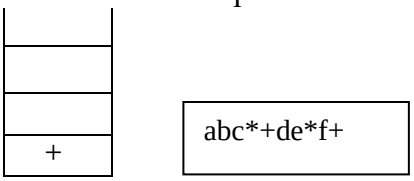
➢ Next by reading a '*', the open parentheses do not get removed except when a closed parenthesis is being processed, there is no output. Next, e is read and output.



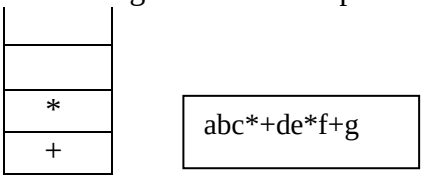
➢ The next symbol read is a '+'. We pop and output and then push '+'. Thenread and output f.



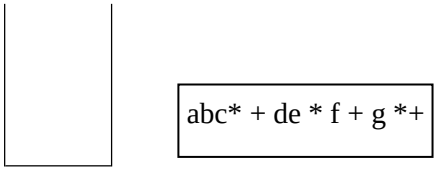
➢ Now we read a ')', so the stack is emptied back to the' We output a '+'



➢ Now read a '*' next; it is pushed onto the stack. Then g is read and output.



➢ The input is now empty, so we pop and output symbols from the stack until itempty.



4. Function Calls:

- The algorithm to check balanced symbols suggests a way to implement function calls.

- The problem here is that when a call is made to a new function, all the variables local to the calling routine need to be saved by the system, since otherwise the new function will overwrite the calling routine's variables.

- Furthermore, the current location in the routine must be saved so that the new function knows where to go after it is done. When there is a function call, all the important information that needs to be saved such as register values and the return address is saved "on a piece of paper" in an abstract way and put at the top of a pile.
- Then the control is transferred to the new function, which is free to replace the registers with its valued. If it makes other function calls, it follows the same procedure.

- When the function wants to return, it looks at the "paper" at the top of the pile and restores all the registers. It then makes the return jump.

- Clearly, all of this work can be done using a stack, and that is exactly what happens in virtually every programming language that implements recursion. The information saved is called either an **activation record** or **stack frame**.

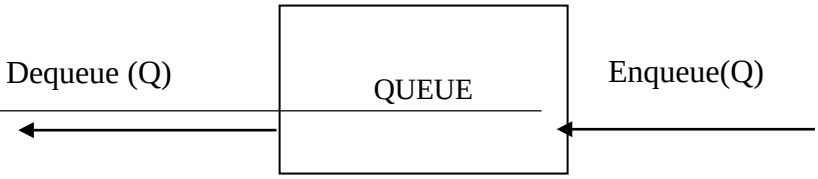
Queue ADT:

Queue is an ordered collection of data items. It delete item at front of the queue. It inserts item at rear of the queue. It has FIFO structure i.e. "**First In First Out".**

3	**5**	**2**	**7**	**6**

front **rear**

Queue Model:



The basic operations are,

- **Enqueue :**which inserts an element at the end of the list called **rear end**.
- **Dequeue:**Which deletes an element at the other end (front) of the list called **front end.**

Types of Queue:

- Simple Queue
- Circular Queue
- Double Ended Queue
- Priority Queue

Implementation of Simple Queue(Queue):

Like stack, queue can also be implemented in two methods.

- Using Array
- Using Linked list

1. Array Implementation of Queue(Simple Queue):

Implementation of Operations:

Global Declaration:

```
int *Queue, Front=0,Rear=-1,maxsize,i;
```

Explanation: The initial values of Front is zero and Rear is -1.

Global Declaration:

```
int *Queue, Rear=-1,front=-1,maxsize;
```

Code to Create Dynamic Queue:

Initially the Queue is declared as a pointer. Using create() function, once the maximum size is known the queue array can be dynamically created.

```
void create()
{
      printf("Enter the size of Queue:");
      scanf("%d",&maxsize);
      Queue=(int)malloc(sizeof(int)*maxsize);
}
```

Code to Check IsFull or IsEmpty:

A Dequeue on an empty queue or a Enqueue on a full queue will overflow the array bounds exception and cause a crash. To avoid this, the following functions are used.

```
int IsFull(int Queue[])
{
      Rear+1==maxsize;
}
int IsEmpty(int Queue[])
{
      return Rear==-1;
}
```

Enqueue(Insert) Operation : To insert element at Rear side.

```
void Enqueue(int Queue[],int x)
{
      if(IsFull(Queue))
            printf("\nQueue Overflow");
      else
      {     Rear++;
            Queue[Rear]=x;
      }
}
```

Explanation:

- The element to be inserted is passed to the function as argument x.
- If Rear reaches the maximum size of the Queue, it implies Queue is full.
- Otherwise rear is incremented and data is stored in Queue[rear].

Dequeue(Delete) Operation: To Dequeue or delete element at front.

```
void Dequeue(int Queue[])
{
      if(IsEmpty(Queue))
            printf("\nQueue underflow/empty");
      else
      front=front+1;
}
```

Explanation:

- If front is -1, it implies that there are no elements in the queue. Hence, Queue is empty.
- Otherwise, delete queue[front] and then shifts all the remaining elements into one position forward.
- Decreases Rear by 1.

Display Operation:

```
void display()
{
      int i;
      if(Rear==-1)
            printf("\n Queue is empty");
      else
      {
            printf("\nThe elements in the stack are \n");
            for(i=0;i<=Rear;i++)
                  printf("%d\t",Queue[i]);
      }
}
```

Explanation:

- If rear = -1, it implies that there are no elements in the queue and it is empty.
- Otherwise, display the elements one by one, from Front to Rear.

Disadvantages of Queue using array implementation:

- The size of the Queue is limited.
- If an element is deleted from the queue, all the remaining elements to be shifted by 1 position forward. This operation takes more cost.
- This problem can be overcome by Circular Queue.

Linked List implementation of Simple Queue:

```
struct Queue
{
      intdata;
      struct Queue *next;
};
struct Queue *queue,*front=NULL,*rear=NULL;
typedefstruct Queue *QUEUE;
QUEUEcreate()
{
      structqueue *temp;
      temp=(structqueue*)malloc(sizeof(structqueue));
      return temp;
}
```

The structure is declared gloabally so that all the functions can access the structure. Front and rearare initially made to NULL. It implies that there is no Queue created.

Enqueue Operation:

```
void  enqueue(int x)
{     int data;
      temp=create();
      temp->data=x;
      temp->next=NULL;
      if(front==NULL)
            Front=rear=temp;
      else
      {
            Rear->next=temp;
            rear=rear->next;
      }
}
```

Explanation: This function creates a new node called temp. The data is stored in the data part of the temp. If front==NULL, it implies that the queue is not created. The node temp becomes front and rear. The next contains NULL. Otherwise, insert the new node at rear->next and make temp as rear.

Dequeue (Deletion) operation:

```
void dequeue()
{
        if(front==NULL)
                printf("\nQueue is empty");
        else
                Front=front->next;
}
```

Explanation: This function deletes the topmost node in the stack. First it checks whether top is NULL. If yes, it implies that the stack does not contain any nodes. Otherwise, top is moved to top->next. It is only a logical deletion and not physical deletion

Display operation

```
void display()
{
        structqueue *ptr=NULL;
        if(front==NULL)
                printf("\nQueue is empty.");
        else
        {
                printf("\nThe elements in the Queue are \n");
                for(ptr=front;ptr!=NULL;ptr=ptr->next)
                        printf("%d\t",ptr->data);
        }
}
```

Explanation: This function displays all the elements in the data part of the nodes one after another.

Circular Queue:

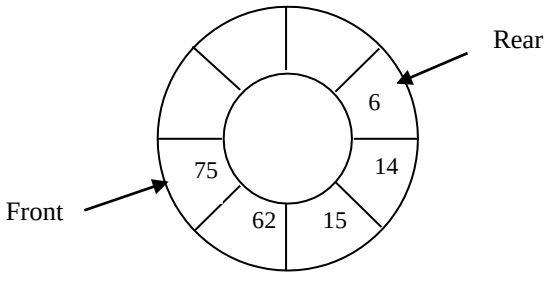
The drawbacks of simple queues are, time consuming and there is a chance to declare queue is full even if there is a empty space at front.

These can be overcome by circular queue.

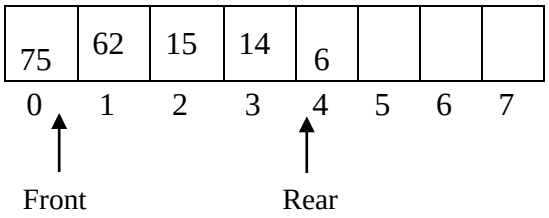
Circular queue is a wrap around queue, after insertion of last position of the queue, if there is an empty space at first, the insertion pointer will insert at first. ie., it won't show queue full message.

It is possible to insert new elements into the circular queue if the array slots at the beginning of the queue is empty.

Pictorial representation of Circular Queue:



Logical representation of Circular Queue:



Circular queue operations are,

- Enqueue(Insertion)
- Dequeue(Deletion)
- Display

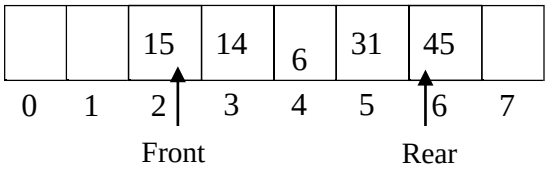
Global declarations:

```
int CQueue[100],front=-1,rear=-1,count=0,maxsize=99;
```

Function to Enqueue at Rear:

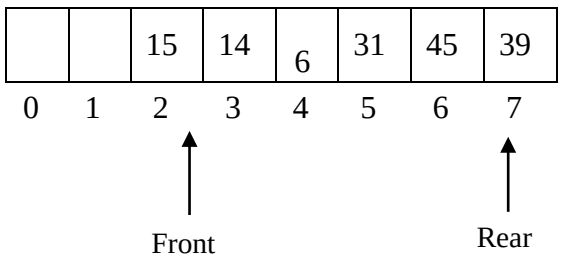
```
void enqueue_rear(int x)
{
      If(count==maxsize)
            Printf("Queue is full");
      Else
      {
            Rear=(rear+1)%maxsize;
            CQueue[rear]=x;
            Count++;
      }
}
```

Example:
Maxsize=8
Initial status of the queue is,



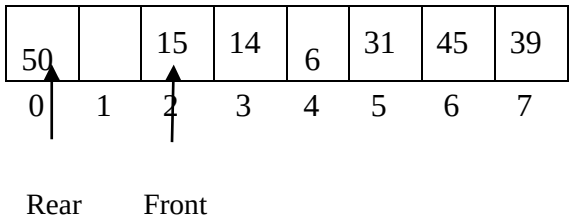
Enqueue 39.

Front=2, rear=(6+1)%8=>*7*



Enqueue 50

Front=2, rear=(7+1)%8=>0

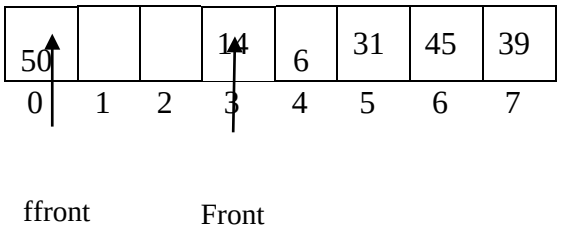


Function to dequeue at front:

```
void dequeue(int x)
{
        If(count==0)
                Printf("Queue is empty");
        Else
        {
                Front=(front+1)%maxsize;
                Count--;

        }
}
```

Example:
Dequeue,

Front=(front+1)%maxsize;



Function to display Queue elements:

```
Void display()
{
        int i;
        for(i=1;i<=count;i++)
        {
                printf(" %d ",CQueue[j]);
                j=(j+1)%maxsize;
        }
}
```

Double Ended Queue.

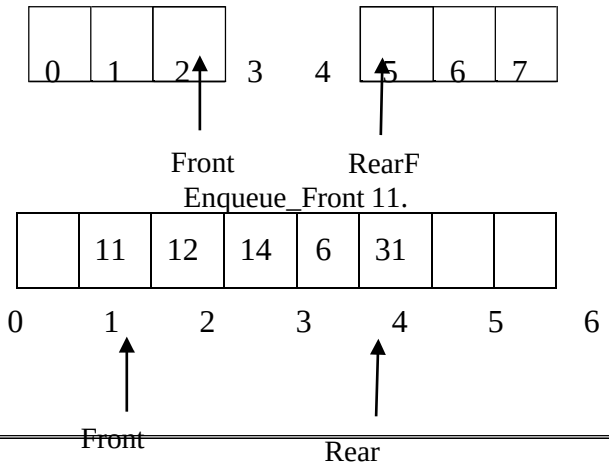
It is another type of queue and this is also called as Deque. A Deque is a special type of data structure in which insertions and deletions can be performed in both the ends. ie., at the front end and the rear end of the queue. There are 4 operations, 1. Insertion at front, 2. Insertion at rear, 3. Deletion at front, 4. Deletion at rear.

Function to enqueue at front:

```
void enqueue_front(int x)
{       if(front==0)
                Printf("Insertion at front is not possible.");
        Else
        {


                Front=front-1;

                Deque[front]=x;
                Count++;
        }
```

7

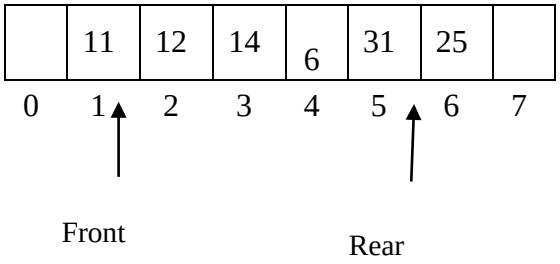
Example:



Function to enqueue at rear:

```
void enqueue_rear(int x)
{       if(rear==maxsize)
                printf("Insertion at rear is not possible:");
        else
        {

                Rear=rear+1;
                Deque[rear]=x;
                Count++;

        }
}
```

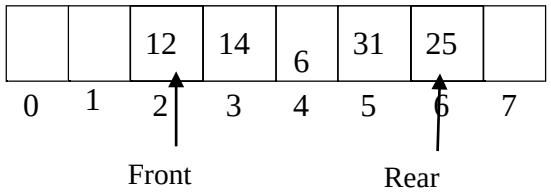
Enqueue_Rear 25



Function to dequeue at front:

```
Void dequeue_front()
{
        if(front>rear)
                Printf("Queue is empty");
        else
        {       Front=front+1;
                Count--;

        }
}
```

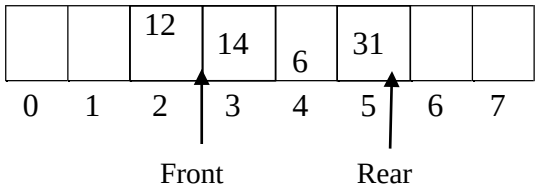
Dequeue_Front



Function to dequeue at rear:

```
Void dequeue_rear()
{
        if(front>rear)
                Printf("Queue is empty");
        Else
        {       Rear=rear-1;
                Count--;
                        }
}
```

Dequeue_Rear



Applications of Queues.

Print jobs

When jobs are submitted to a printer, they are arranged in the order they arrive. Then jobs sent to a line printer are placed on a queue.

Computer networks

There are many network setup of personal computers in which the disk is attached to one machine called file server. Users on other machines are given access to files on a first come first served basis where the data structure is queue.

OS Operating system performs various tasks namely memory management, CPU management etc. The operating system follows a technique called **First Come Fist Serve (FCFS)** to allocate CPU for the waiting processes.

Real-life waiting lines:

i. Calls to large companies are generally placed on a Queue when all the operators are busy.

ii. In large Universities, where resources are limited, students must sign a waiting list if all terminals are occupied.

Mathematics:

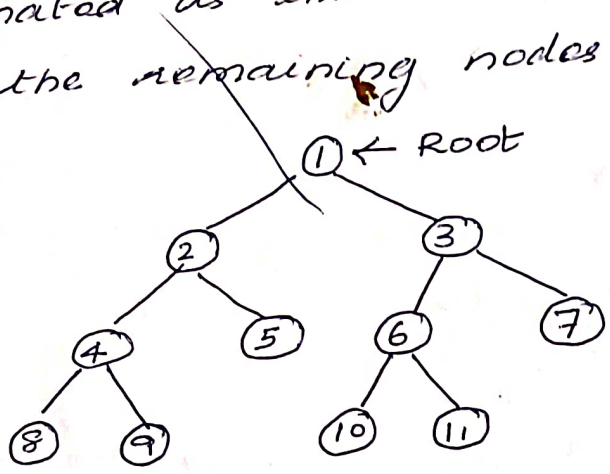
There is a branch of Mathematics called Queuing Theory, which deals with computing, probabilistically, how long users expect to wait on a line.

NON-LINEAR DATA STRUCTURES

Tree ADT – tree traversals – Binary Tree ADT Expression trees – applications of trees Binary search Tree ADT – Threaded Binary Tree AVL trees – B-tree – B+ tree – Heap – Applications of heap

Tree ADT:

A tree is recursively defined as a set of one or more nodes where one node is designated as the root of the tree and all the remaining nodes can form subtrees



Root Node: The root node R is the topmost node in the tree. If R = NULL, then it means the tree is empty.

Leaf Node: A node that has no children is called leaf node or terminal node

Eg: 8, 9, 5, 10, 11, 7.

Degree: Degree of a node is equal to the number of children that a node has. The degree of leaf node is zero

Eg: Degree (6) = 2

Siblings: All the nodes that share same parent are called siblings.

Eg: (8,9) (10,11)

Path: A sequence of consecutive edges

Eg: Path (1,8) = 1,2,4 and 8

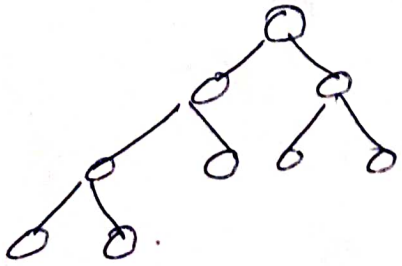
Depth: The depth of a node N is the length of the path from the root to the node N.

Eg: Depth (5) = 3
Depth (9) = 3

Depth of a root node is zero.

Level: Every node in the tree is assigned a level number in such a way that root node is at level 0, children of the root node are at level 1

Complete binary tree: A complete binary tree is a binary tree, in which all the levels are completely filled from left except possibly the last.



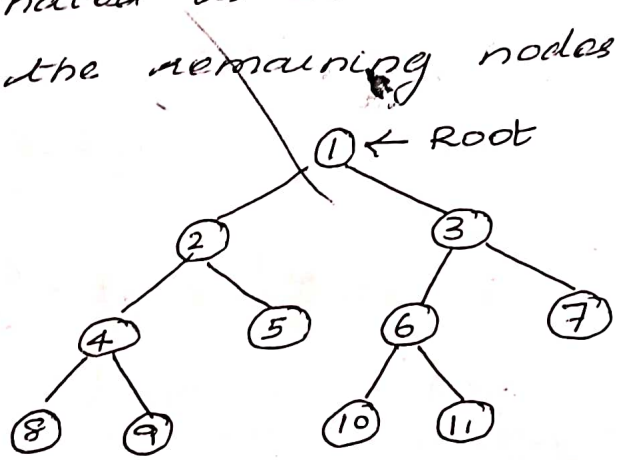
Height: The height of a node N is the length of the path from the node 'N' to root.

NON LINEAR DATA STRUCTURES

Tree ADT – tree traversals – Binary Tree ADT Expression trees – applications of trees Binary search Tree ADT – Threaded Binary Tree AVL trees – B–tree – B+ tree – Heap – Application of heap

Tree ADT:

A tree is recursively defined as a set of one or more nodes where one node is designated as the root of the tree and all the remaining nodes can form subtrees



Root Node: The root node R is the topmost node in the tree. If R = NULL, then it means the tree is empty.

Leaf Node: A node that has no children is called leaf node or terminal node

Eg: 8, 9, 5, 10, 11, 7.

Degree: Degree of a node is equal to the number of children that a node has. The degree of leaf node is zero

Eg: Degree (6) = 2

Siblings: All the nodes that share same parent are called siblings

Eg: (8,9) (10,11)

Path: A sequence of consecutive edges

Eg: Path (1,8) = 1,2,4 and 8

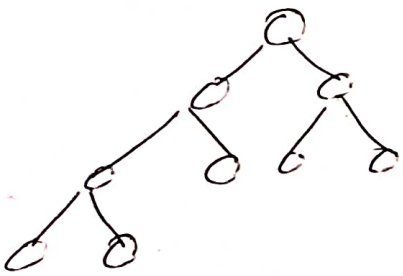
Depth: The depth of a node N is the length of the path from the root to the node N

Eg: Depth (5) = 3
Depth (9) = 3

Depth of a root node is zero.

Level: Every node in the tree is assigned a level number in such a way that root node is at level 0, children of the root node are at level 1

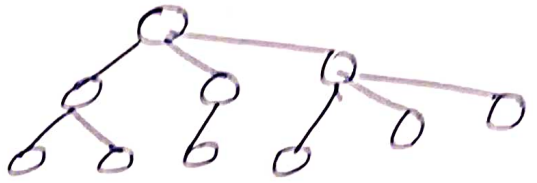
Complete binary tree: A complete binary tree is a binary tree, which all the levels are completely filled from left except possibly the last



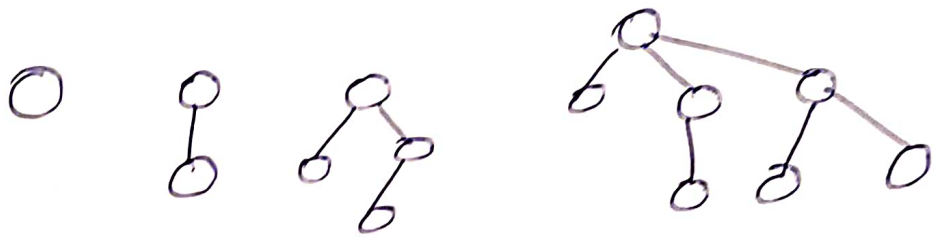
Height: The height of a node N is the length of the path from the node 'N' to root.

Types of trees:

- General trees: General trees store elements hierarchically. A node in a general tree may have zero or more subtrees.

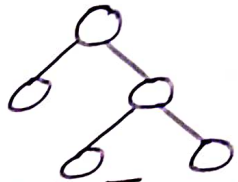


- Forests: A forest is a disjoint union of trees. A forest is also defined as an ordered set of zero or more general trees.

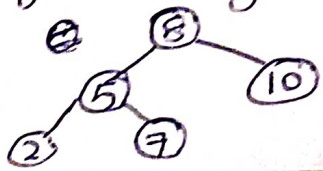


A forest.

- Binary Trees: Binary tree is a tree with not more than 2 children. Every node can have 0, 1, or at most 2 children.



- Binary search Tree (BST): Binary search Tree is a binary tree with a condition that left child is smaller than the root and right is greater than the ro



Tournament trees: In a tournament tree, each external node represents a player and each internal node represents the winner of the match played between the players represented by its children.

TREE TRAVERSALS:

- Traversal is the process of visiting each node in the tree exactly once in a systematic way.
- Types traversals
  - ⇒ Inorder traversal
  - ⇒ Preorder traversal.
  - ⇒ Post order traversal.
  - ⇒ Level-order Traversal

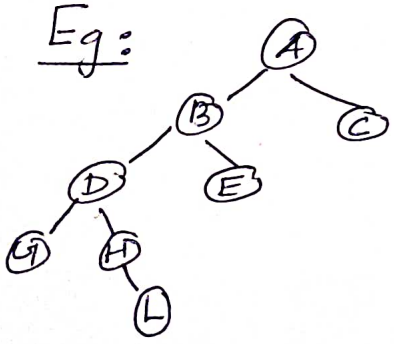
Inorder traversal:

To traverse a non-empty binary tree in in-order the following operations are performed recursively at each node

Algorithm:

1. Traverse the left subtree
2. Visit the root node
3. Traversing the right subtree

Eg:



Inorder traversal:

GDHLBEAC

In-order traversal is also called as symmetric traversal. In-order algorithm is also known as the LNR traversal algorithm (Left-Node-Right).

Routine:

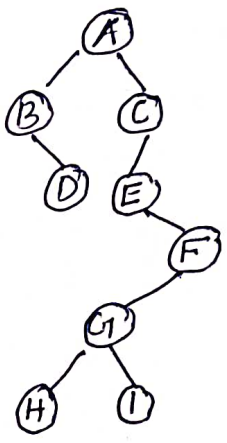
```
void    Inorder (Tree T)
 {
    if(T != NULL)
      {
        Inorder (T→Left);
        Write T→Data;
        Inorder (T→Right);
      }
 }
```

Preorder Traversal:

To traverse a non empty binary tree in pre-order, the following operations are performed recursively at each node.

Algorithm:

1. Visit the root node
2. Traverse the left sub-tree
3. Traverse the right sub tree



A B D C E F G H I
(pre order)

Preorder algorithm is also known as the NLR traversal algorithm (Node - Left - Right). Preorder traversal algorithms are used to extract a pr notation from an expression tree.

Routine

```
void Preorder ( Tree T)
{
    if (T != NULL)
    { Write Node T→Data;
      Preorder (T→Left);
      Preorder ( T→Right);
    }
}.
```

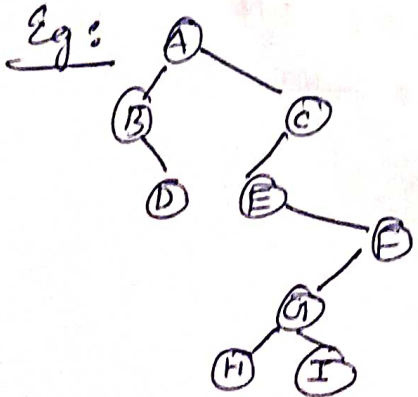
Post order Traversal:

To traverse a non-empty binary tree in post order, the following operations are performed recursively at each node.

Algorithm:

1. Traverse the left sub-tree
2. Traverse the right subtree
3. Visit the root node.

Eg:



D B H I G F E C A

Post-order algorithm is also known as LRN traversal algorithm (Left-Right-Node) Post order traversals are used to extract postfix notation from an expression tree.

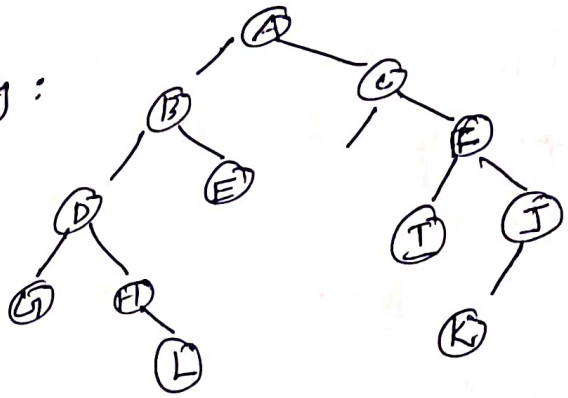
Routine:

```
Void Postorder (Tree T)
{
  if (T != NULL)
    {
      Postorder (T->Left);
      PostOrder (T->Right);
      Write T->Data;
    }
}.
```

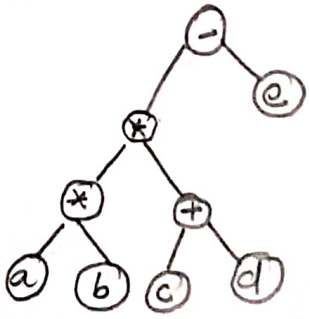
Level order Traversal:

In level order traversal, all the nodes at same level are accessed before going to the next level. This algorithm is also called as the breadth-first travel algorithm.

Eg:

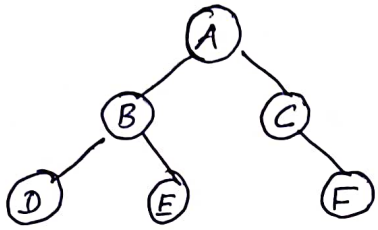


A B C D E F G H I J K L
(Traversal)



BINARY TREE ADT:

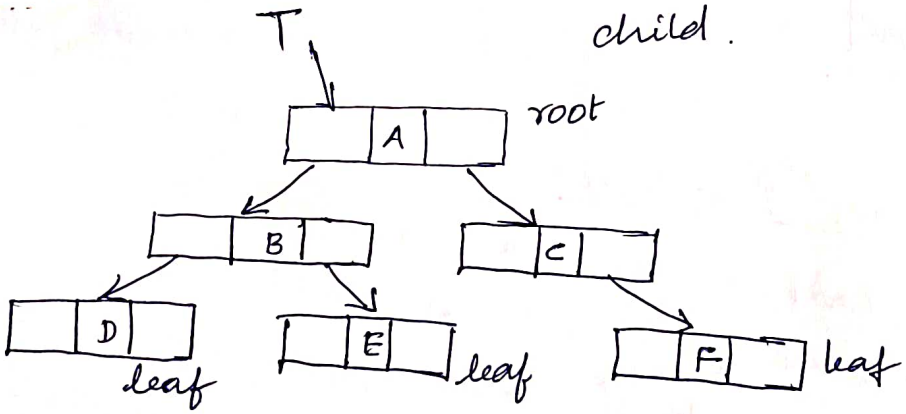
Binary tree is a tree in which no node can have more than 2 children. Every node can have either 0, 1 or 2 children.



Eg: Binary Tree

A binary tree of height h h
at least 'h' nodes and at
$2^h - 1$ nodes.
A binary tree of 'n' nodes
exactly n-1 edges
The height of a binary t
with 'n' nodes is at least l
and atmost n.

It is also defined as collection of nodes and top most node is root node. Every node contains data element, a left pointer which points to the child, and a right po which points to the r child.



- Root node is pointed by a pointer 'T'.
- Every node in a tree is connected by a direc edge from exactly one other node ie parent
- A node can have 0, 1 or 2 children.
- Nodes with no children are called leave external nodes.
- Nodes with same parent are called siblings

Node declaration:

```
struct Node
{
    ElementType Element;
    Tree        Left;
    Tree        Right;
};
```

- The depth of a node 'x' is the number of edges from root to the node 'x'.
- The height of a node 'x' is the number of edges from node 'x' to the deepest leaf.
- Height of a tree is the height of the root.
- Complete binary tree is a binary tree which is completely filled from left to right except the last level.

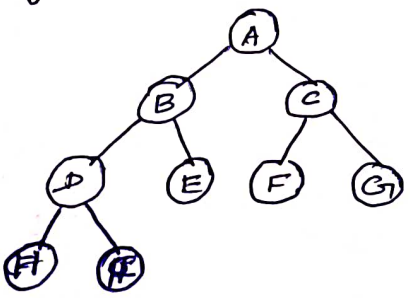


Fig: Complete binary tree.

Full binary tree is a tree in which every node other than the leaves has 2 children.

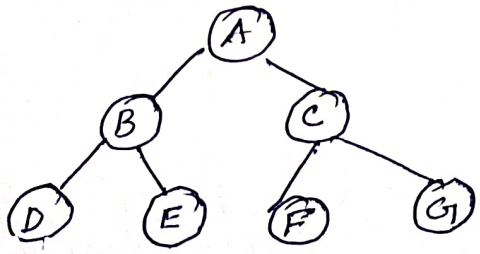


Fig: Full binary tree

Representation of Binary Trees:

- Linked list representation.
- Sequential or Array representation.

Linked list representation:

In this, every node will have three parts

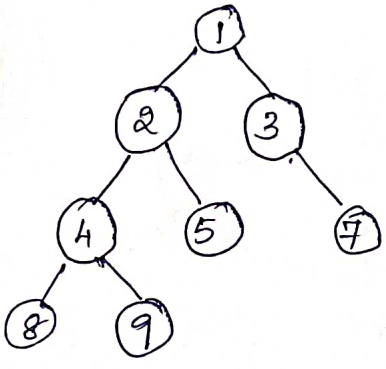
Data → Data to be stored.

Left → Pointer to the left sub tree

Right → Pointer to the right subtree.

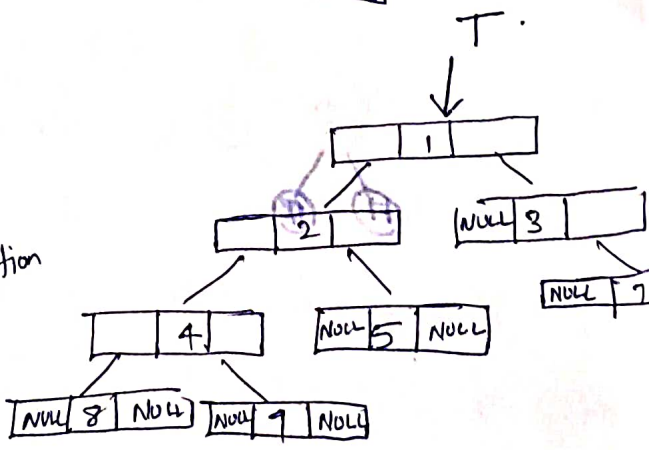
Node declaration:

```
struct Node
{
    ElementType Element;
    struct Node *Left;
    struct Node *Right;
};
```



A Binary Tree

linked list representation



Linked List represen

Root node is pointed by a pointer T.

If 'T' is NULL, then tree is empty.

Sequential representation / Array representation

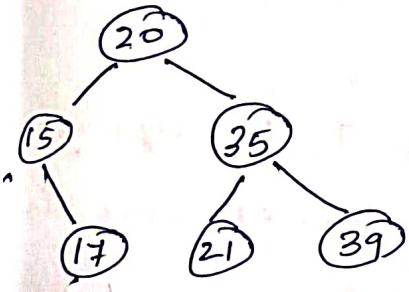
A single dimensional array is used for sequential representation. This is the simplest technique for memory representation.

- Root of the tree is stored in the 1st location.
- Left subtree of x

  Left(x) = 2(i), i → position of x

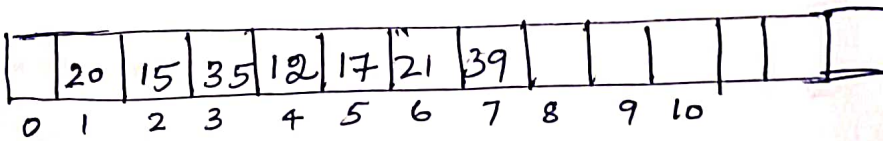
- Right subtree of x

  Right(x) = 2i + 1,



Binary Tree

Array / sequential representation



x = 20, i = 1

Left(x) = Left(20) = 2i = 2(1) = 2

Right(x) = Right(20) = 2i + 1 = 3

x = 15, i = 2

Left(15) = 2i = 2 * 2 = 4

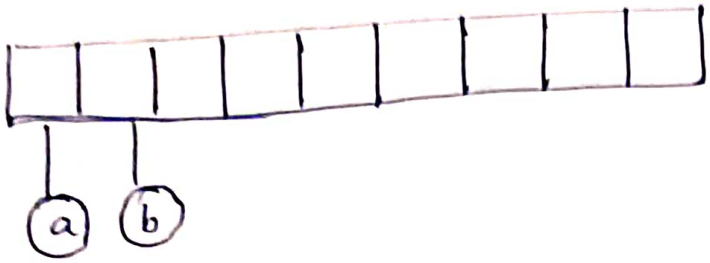
Right(15) = 2i + 1 = 4 + 1 = 5

x = 35, i = 3

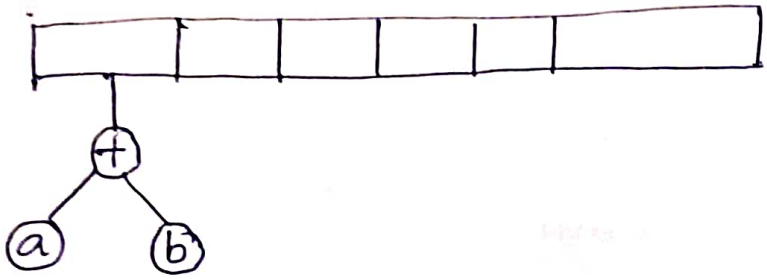
Left(35) = 2 * 3 = 6

Right(35) = 2 * 3 + 1 = 7

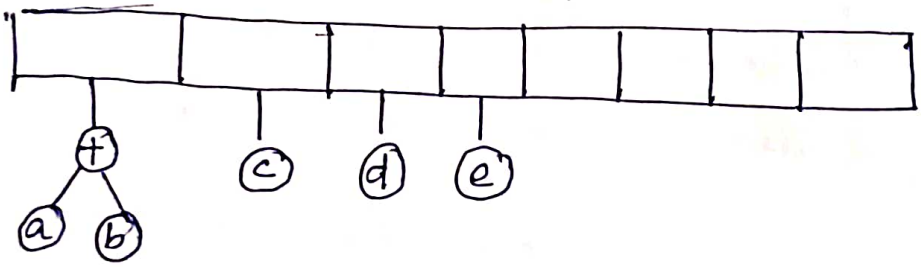
3. Next input is b, an operand, create a new node



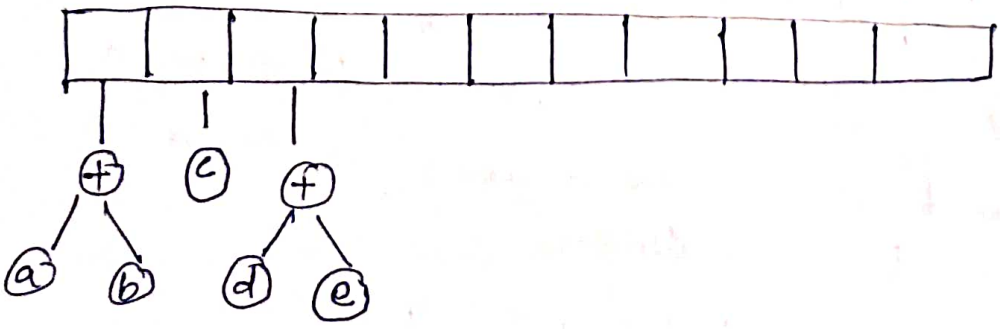
4. Next input is +, an operator, pop top 2 and create a new tree



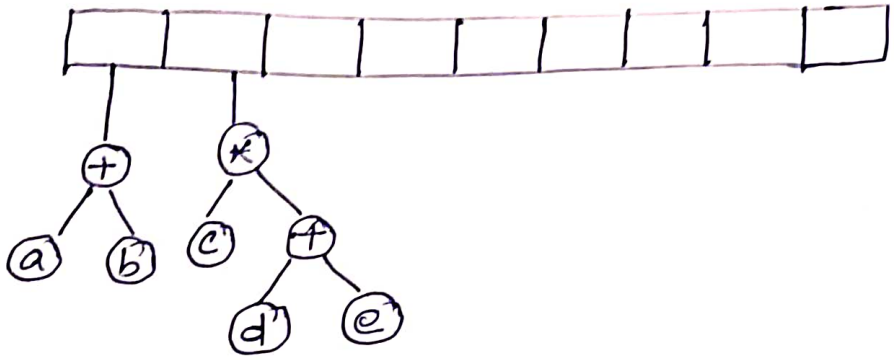
5. Next 3 inputs are operands ie b, d, e, create new nodes and push it



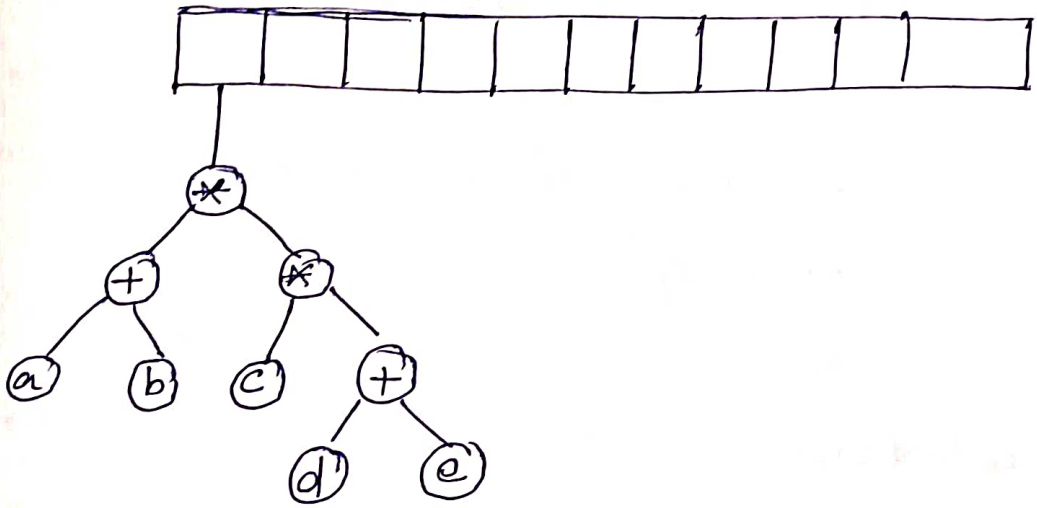
6. Next input is +, pop top 2 and creat a new tree



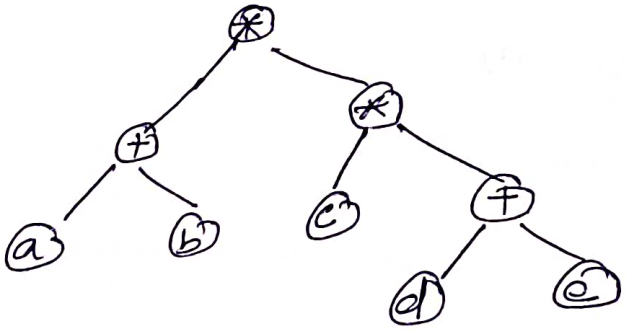
7. Next input is an operator '*', pop 2, and create new tree.



8. Next input is an operator *, pop 2 elements, and create new subtree.



Final expression tree is



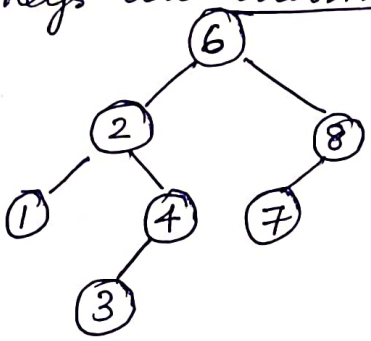
Applications of trees:

- Trees are used to store simple and complex data. Simple means an integer and character values. Complex data means a structure or a reco
- Trees are often used for implementing oth types of data structures like hash tables, sets and maps.
- A self-balancing tree, Red-black tree is used in kernal scheduling, to preempt massively multiprocessor computer operating system use
- B-trees are prominently used to store tre structures on disc. They are used to in a large number of records.
- B-trees are also used for secondary in in databases, where the index facilit a select operation to answer some ran criteria.
- Trees are an important data structu used for compiler construction.
- Trees are also used in database design.
- Trees are used in file system directories
- Trees are also widely used for information storage and retrieval in symbol tables.

BINARY SEARCH TREE ADT: (BST)

Binary search Tree is a binary tree with a condition that the key values of left subtree is smaller than the root node and the key values of right subtree is greater than the root node.

- Thus, every node is assigned with a key value
- All the keys are distinct.



Binary search Tree

The root node is 6. The left subtree of root node consists of nodes 1, 2, 3, and 4. The right sub tree consists of 7, and 8.

For the left subtree root node is 2, The left has 1 and right has 3 and 4.

Thus recursively it applies to all the nodes.

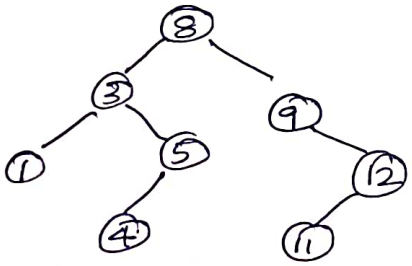
- Since the nodes in the binary search tree are ordered, the time needed to search an element in the tree is reduced.
- Whenever we search for an element, do not need to traverse the entire tree
- Binary search trees are widely used in dictionary problems.

Operations on BST:

1. Insertion
2. Deletion
3. Find
4. Find Min
5. Fin Max.

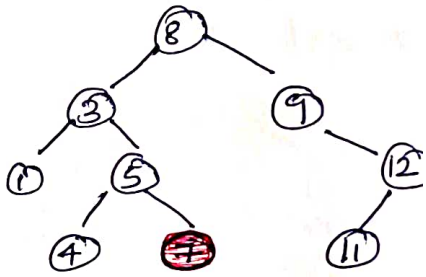
(i) Insertion:

We start at the root and recursively go d the tree for appropriate location and inserts node. If the element to be inserted is alree in the tree, no need to insert.



Before insertion.

Insert 7



After insertion.

Node declaration:

```
Struct Node
{
    ElementType Element;
    Struct Node * Left;
    Struct Node * Right;
};
```

Routine for insertion:

```
SearchTree  Insert ( ElementType X , SearchTree T )
{
   if ( T == NULL)
      {
        T = malloc (sizeof (struct Node));
        if (T==NULL)
            FatalError ("out of space");
        else
            {
              T→Element = X;
              T→ Left  = T→Right = NULL;
            }
      }
   else
      {
        if ( X< T→Element )
              T→Left = Insert ( X, T→Left);
        else  if ( X > T→Element )
                    T→Right = Insert ( X, T→Right);
              else
                    return T;
      }
```

Deletion:

Unlike insertion, deletion ~~case~~ has different ceases

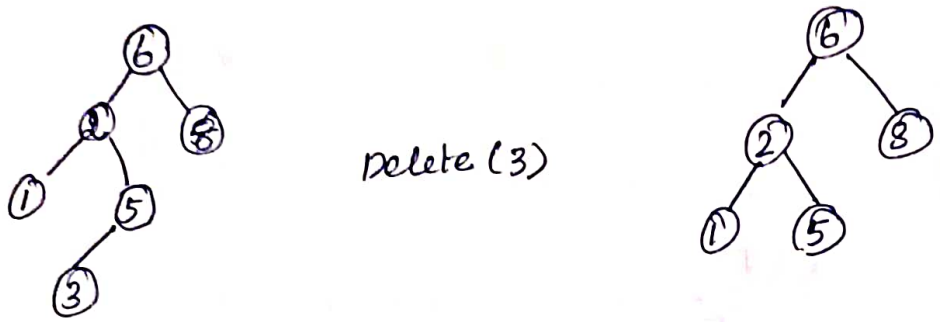
case 1: Deleting a leaf node

case 2: Deleting a node with one child

case 3: Deleting a node with 2 children.

Case 1 : Deleting a leaf node

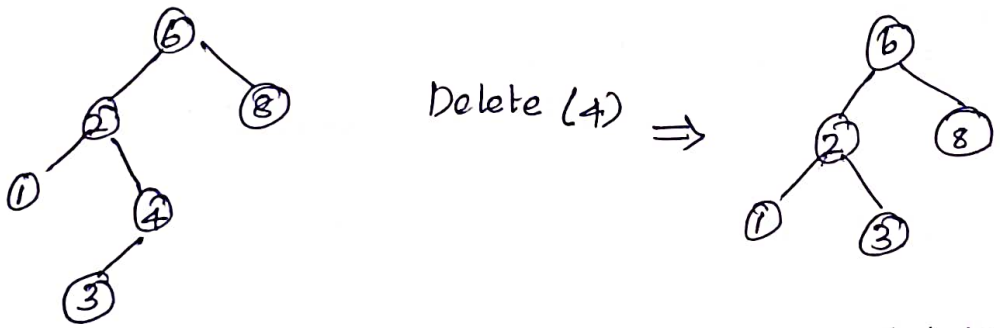
If a node is a leaf, it can be deleted immediately.



Before Deletion. After deletion.

Case 2 : Deleting a node with one child.

If the node has one child, the node can be de after its parent adjusts a pointer to bypass node

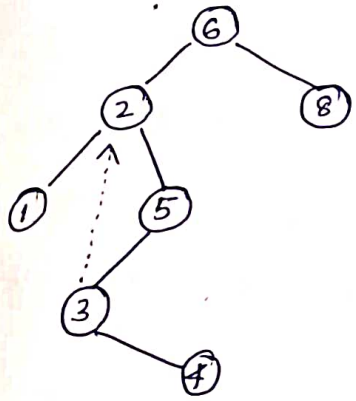


After deletion.

Before deletion.

case 3: Deleting a node with 2 children:

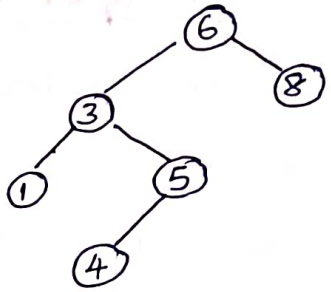
Replace the node to be deleted with the smallest data in the right subtree and delete the node.



Before Deletion

Delete (2)

Find the min at the right subtree i.e 3 replace (2) with (3)



After deletion.

Routine for deletion:

```
SearchTree Delete (ElementType X, SearchTree T)
{
  Position TmpCell;
  if (T==NULL)
      Error ("Element not found");
  else
     if (X< T->Element)
          T->Left = Delete (X, T->Left)
  else if (X > T->Element)
          T->Right = Delete ( X, T->Right)
       else if ( T->Left && T->Right) // Two children
            {
              Tmpcell= Find Min (T->Right);
              T->Element = TmpCell -> Element ;
              T->Right = Delete ( T->Element ,T->Right
            }
       else
            {
              TmpCell =T;
              if ( T->Left ==NULL)
                    T= T->Right;
              else if (T->Right == NULL)
  return T;} free ( TmpCell); T->Left;
```

(iii) Find operation

Find operation start comparing with root node and goes towards down until it finds the element. Once it found, returns the address of the node.

Routine:

```
Position Find ( ElementType X, SearchTree T)
{
if (T==NULL)
      return NULL;
if ( X< T→Element )
      return Find ( X,T→Left);
else if ( X>T→Element )
      return Find ( X,T→Right);
else
      return T;
}
```

(iv) Find Min operation:

Returns the minimum element in the tree

```
Position FindMin ( SearchTree T)
{
  if ( T==NULL)
      return NULL;
  else if ( T→Left ==NULL)
      return T;
  else return FindMin ( T→Left);
}
```

(v) Find Max Operation

Returns the maxi element in the

```
Position FindMax (SearchTree
{
  if (T==NULL)
      return NULL;
  else if (T→Right==N
      return T;
  else
      return FindMax
}
```

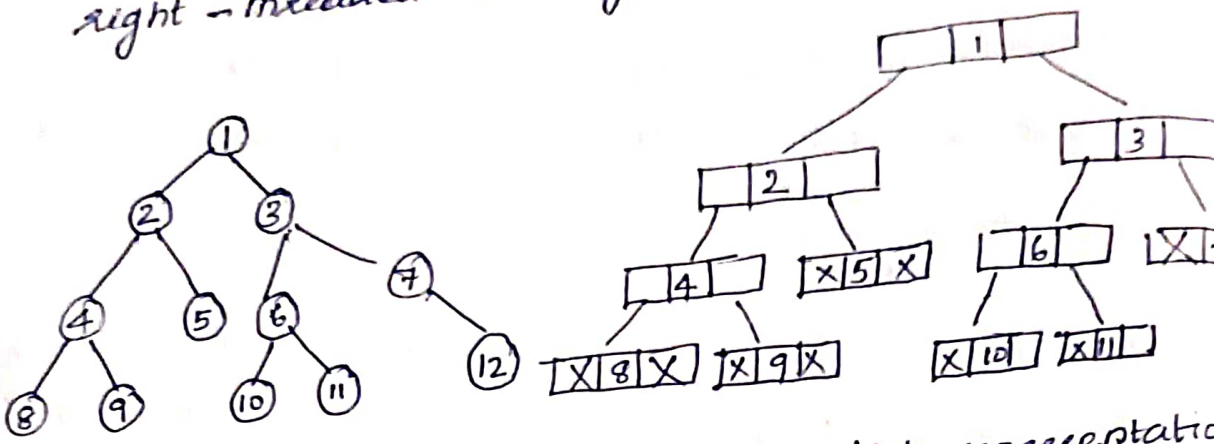
THREADED BINARY TREES:

- A threaded binary tree is a binary tree, with pointers in the leaves pointing the predecessor instead of being NULL.
- In linked list representation of a binary tree, left and right pointers will be NULL.. This space is wasted. To efficiently use this space, NULL entries are replaced with some values.
- These pointers are called as threads and binary tree containing threads are called threaded binary trees.
- There are 2 ways of threading
  - One way threading
  - Two way threading.

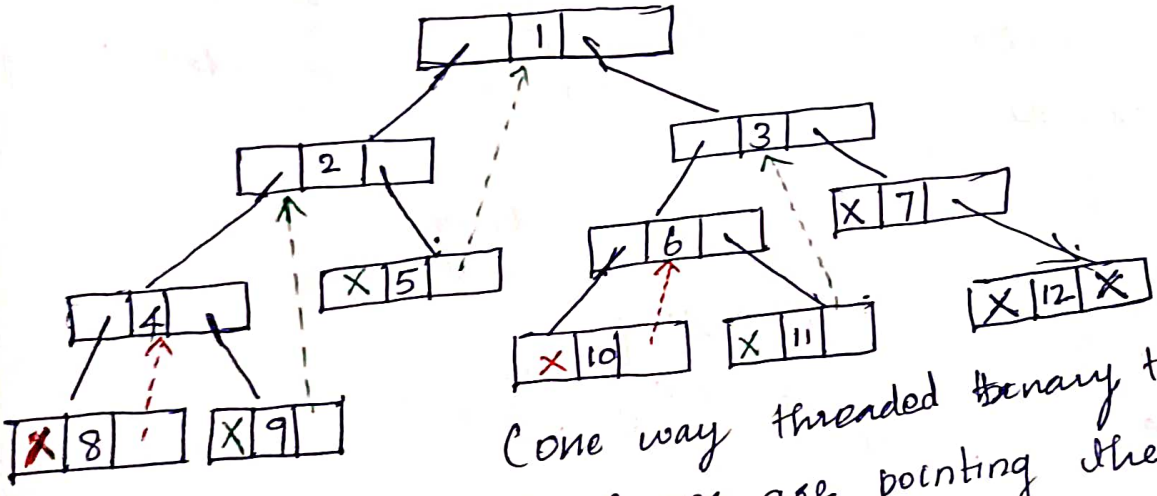
One way threading:

- A thread will appear in the right field or left field of the node
- A one way threaded tree is also called a single threaded tree.
- If the thread appears in the left field, then the left field will be made to point to the in-order predecessor of the node
- Such a one way threaded tree is called a left-threaded binary tree.
- If the thread appears in the right field, then it will point to the in-order successor

- Such a one~~ne~~ way threaded tree is called a right-threaded binary tree.



Linked list representatio
(without threading



(One way threaded ~~B~~inary trees)

- Right pointers ~~of~~ leaves are pointing the inor successor
- Left pointers are made as NULL
- Inorder successors are calculated from inorder traversal of the tree.

Inorder traversal: 8 4 9 2 5 1 10 6 11

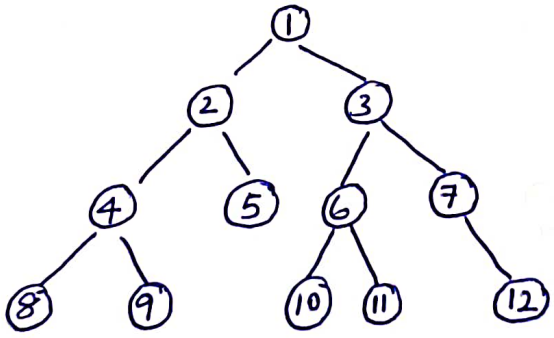
- Element followed by the leaves ~~are~~ is success

inorder successor of 8 is 4, 9 is 2, 1
11 is 3,

12 has no successor

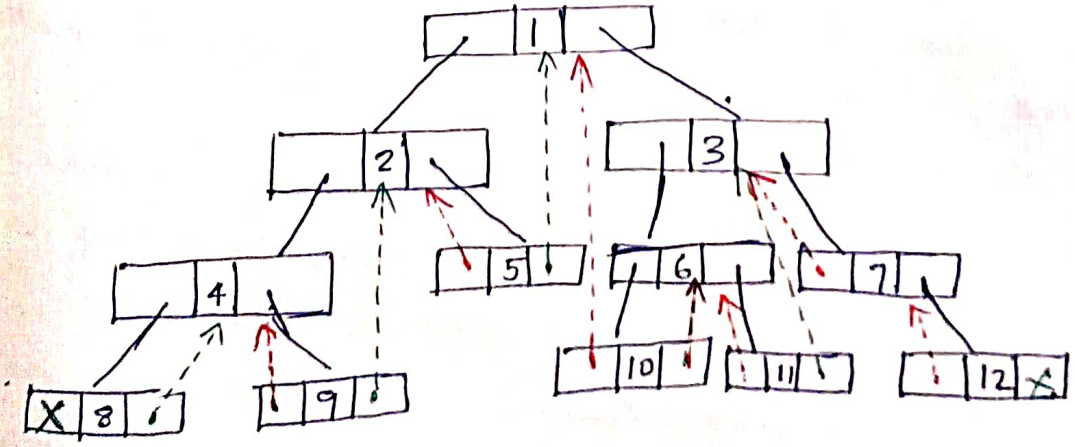
Two way threading:

- In a two way threaded binary tree, threads will appear in both the left and right field of the node
- It is also called as double threaded tree.
- Left field will point the inorder predecessor of the node
- Right field will point to its successor
- It is also called as fully threaded binary tree



Binary tree

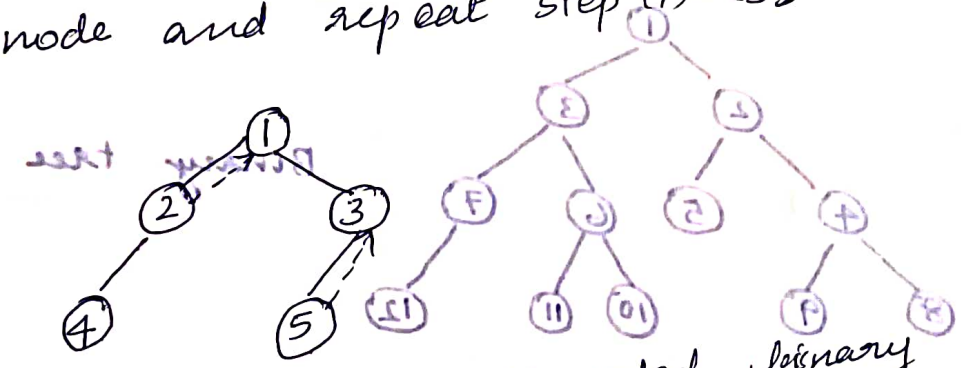
Inorder traversal : 8 4 9 2 5 1 10 6 11 3 7 12



Traversing a threaded Binary Tree:

Algorithm:

1. For every node, if there is a left subtree, mark it as visited, if not visited
2. Then consider the visited node as the current node and check for its left, and if there is no left and right ~~unvisited~~, print the current node.
3. Print the root node and check for its right, if no right follow the threaded link and go back to the previously visited node and repeat step (1)-(3)



Let us consider the threaded binary tree in the above fig and traverse it using the algorithm

1. Node 1 has a left child ie 2 which has not been visited, so add 2 in the list of unvisited, make as the current node.
2. Node 2 has a left child ie 4 which has not been visited. so add 4 in the list of visited nodes and make it as current node.
3. Node 4 does not have any left or right child, so print 4 and check for its thread link. It has a threaded link to node 2, so make 2 the current. Thus repeat for all nodes

Advantages of Threaded Binary tree:

1. It enables linear traversal of elements in the tree.
2. Linear traversal eliminates the use of stacks which in turn consume a lot of memory space and computer time.
3. It enables to find the parent of a given element without explicit use of parent pointers.
4. Since nodes contain pointers to inorder predecessor and successor, the threaded tree enables forward and backward traversal.

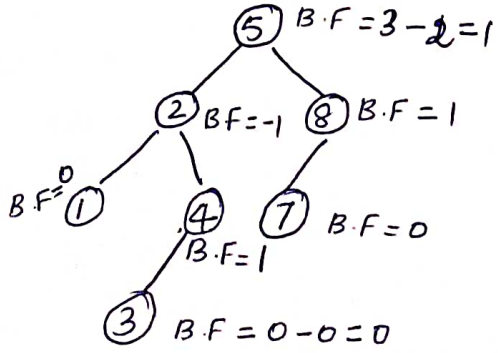
AVL TREES: (Adelson Velsky Landis)

An AVL tree is a binary search tree in which every node has a balance factor of 0, -1, or 1

> Balance factor = Height of left subtree − Height of right subtree

A node with any other balance factor is considered to be unbalanced and requires rebalancing of tree.

- If the balance factor of a node is 1, then it means that the left subtree of the tree is one level higher than the right subtree is therefore called as left-heavy tree
- If the balance factor of a node is 0, then it means that the height of the left subtree is equal to the height of the right subtree.
- If the balance factor of a node is -1, then it means that the left subtree of the tree is one level lower than the of the right subtree. Such a tree is therefore called as right heavy tree



An AVL tree.

- AVL tree should be maintained balanced.
- During insertion or deletion violation may occurs. To rebalance the tree rotation is performed on tree.

ROTATION:

- Rotation is the process of changing the positions of the node, inorder to rebalance the tree.
- Violation may occur in following cases
  1. An insertion into the left subtree of the left child of α
  2. An insertion into the right subtree of the left child of α.
  3. An insertion into the left subtree of the right child of α.
  4. An insertion into the right subtree of the right child of α.

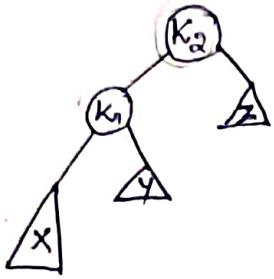
$\alpha \rightarrow$ Violated Node

Types of rotations.

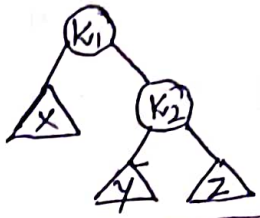
- Single rotation.
  - → Left Left rotation. (LL - Rotation)
  - → Right Right rotation (RR - Rotation)
- Double Rotation
  - → Left – Right Rotation (LR - Rotation)
  - → Right - Left Rotation (RL - Rotation)

LL Rotation:

When the node is inserted at the left subtree of left child of violated node α, LL rotation is applied

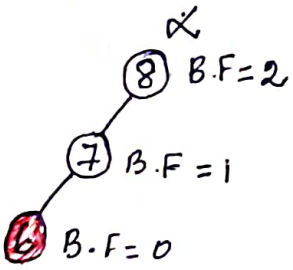


LL Rotation $\Rightarrow$

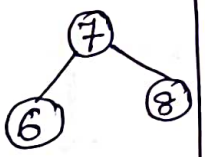



```
Position SingleRotateWithLeft
                        Position
{
  Position K1;
  K1 = K2→Left;
  K1→Right = K2;
  K2→Left = K1→Right;
K2→Height = Max(Height (
                    Height (K2
K1→Height = Max(Height (K1→
                    K2→Left
  return K1;
}.
```

Eg:1:

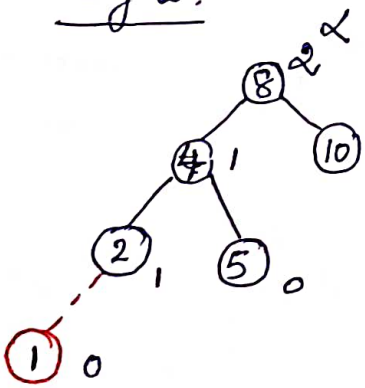


LL(8) $\Longrightarrow$

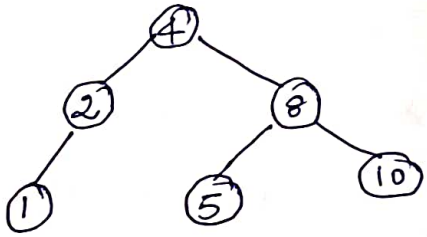


After Inserting (6) violation occurs at (8), which is α, perform LL-rotation on (8)

Eg 2:



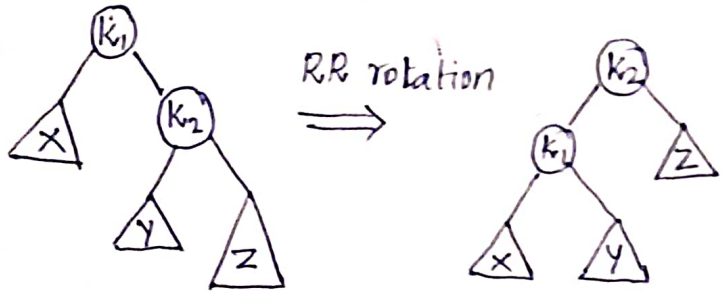
LL(8) $\Longrightarrow$



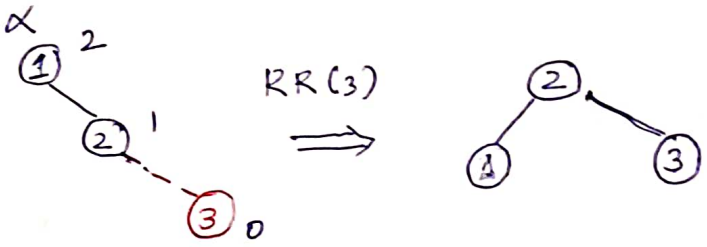
Inserting (1), (2) is violated.

RR Rotation:

When the node is inserted at the right subtree of right child of violated node α RR rotation is applied.

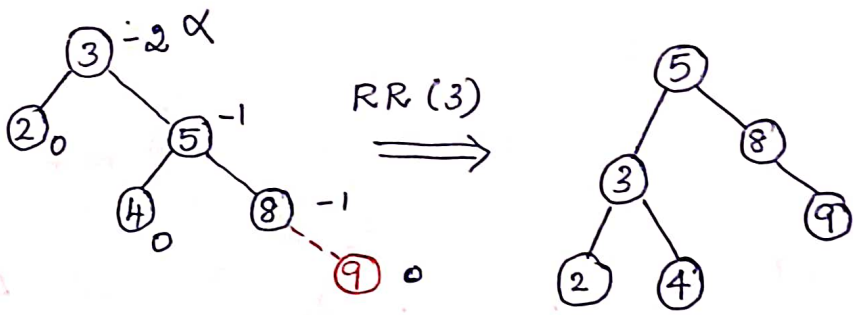


Eg:



Inserting (3)

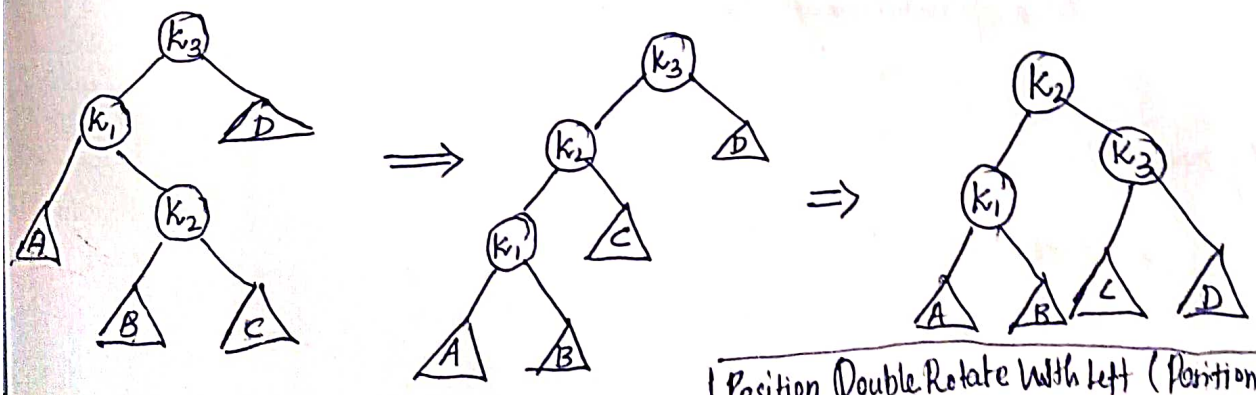
Eg 2:



Inserting (9)

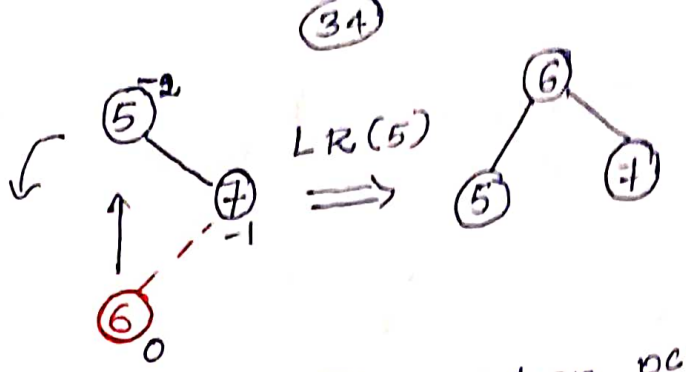
LR Double Rotation:

- When a node is inserted at the right subtree of left child of violated node α, LR rotation is applied.




```
Position DoubleRotate WithLeft ( Position K3)
{
  K3->Left = SingleRotateWithRight (K3->Left);
  return SingleRotateWithLeft(K3);
}
```

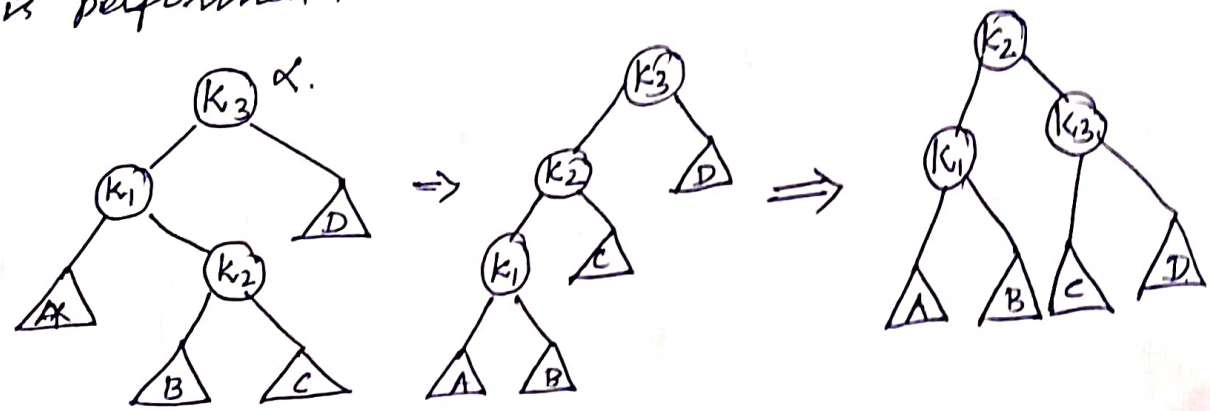
Eg:



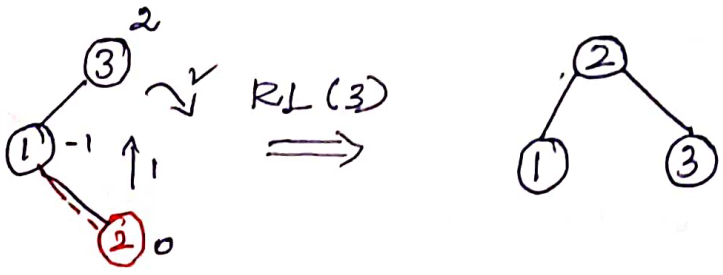
After inserting (6), violation occcurs at 5.

RL Rotation:

When a node is inserted at the left subtree of right child of violated node α, RL rotation is performed.



Eg 1:



After Inserting (2)

Operations:

- Insertion operation.
- Deletion operation.
- FindMin. operation
- FindMax. operation
- Find operation

Insertion:

Insertion is performed similar as binary search tree. New node is compared with the root node, if key value of new node is smaller than root node, then inserted at the left and if greater than root, then inserted at the right.

After every insertion balance condition is checked for every node, if violation occurs rotation is applied.

Routine:

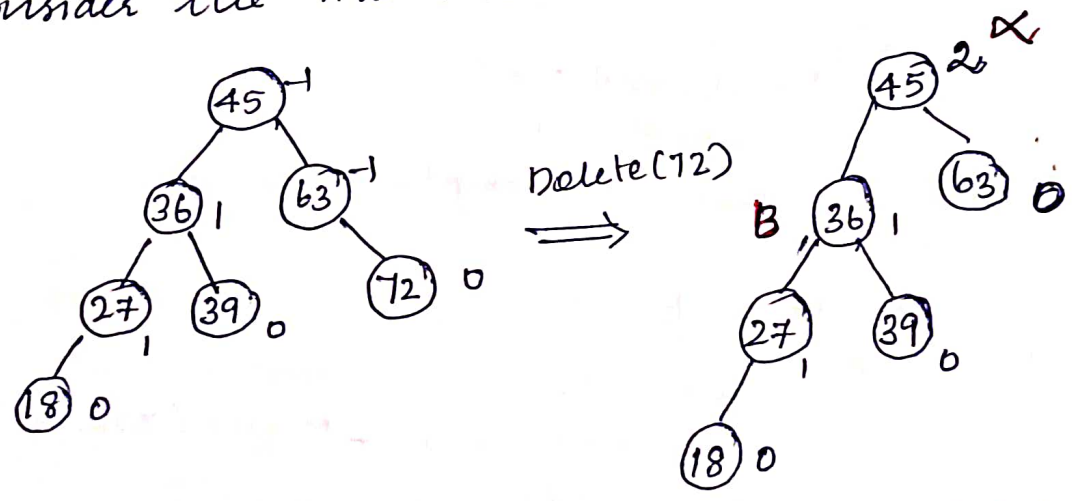
```
AVLTree Insert ( ElementType X, AVLTree T)
{
  if (T == NULL)
    {
      T= malloc(sizeof( struct AVLNode);
      if (T==NULL)
          FatalError ("Out of space");
      else {
          T→Element = X;
          T→ Height = 0;
          T→ Left  =T→Right = NULL;
        }
  else if ( X< T→ Element )
        {
          T→ Left = Insert ( X, T→Left );
          if ( Height (T→Left) - Height ( T→Right ) == 2 )
              if (X < T→Left → Element )
                    T= SingleRotate WithLeft (T);
              else
                    T= DoubleRotate WithLeft (T);
        }
```

```
else if ( X > T -> Element )
{
    T -> Right = Insert ( X, T -> Right );
    if ( Height (T -> Right - Height (T -> Left ) == 2 )
        if ( X > T -> Right -> Element ).
            T = SingleRotatewith Right (T);
        else
            T = DoubleRotatewith Right (T);
}
T -> Height = Max ( Height (T -> Left), Height (T -> Right)
return T;
}
```

Deletion:

Deletion of a node in an AVL tree is similar to that of binary search trees. But after every deletion balance condition is checked to rebalance the tree..

Consider the AVL tree and delete 72 from it

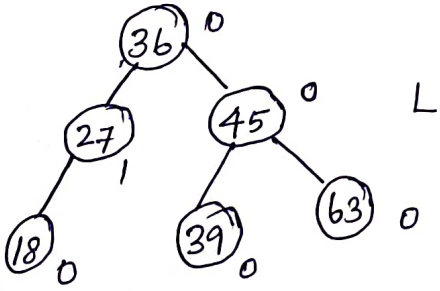


After deletion, node 45 is violated

There are 2 cases,

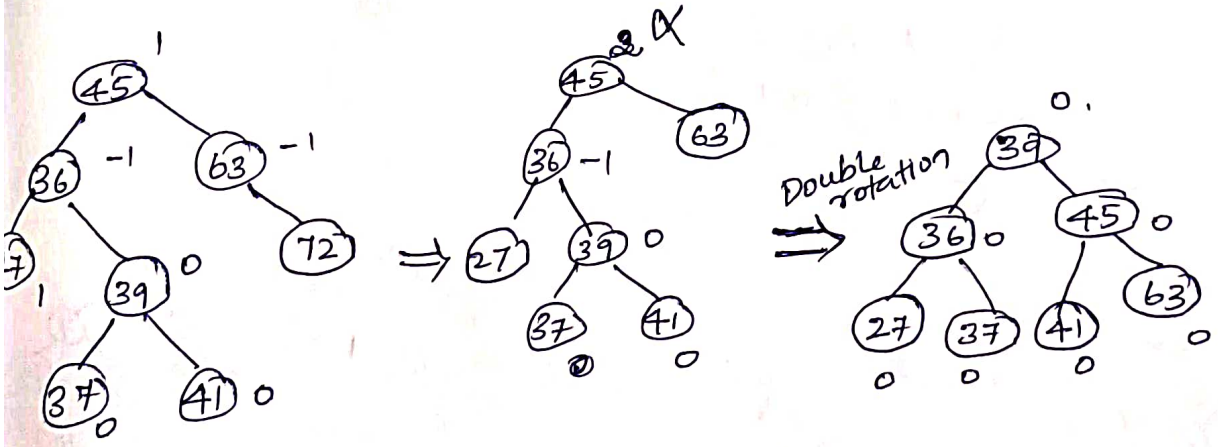
- Let B be the left or right of violated node α. If B is 0 or 1 then single rotation is applied.
- If B is -1, then double rotation is performed.

In the above example, B has 1 hence single rotat is performed.



LL rotation is applied on violated node (45)

Consider the below tree and delete 72



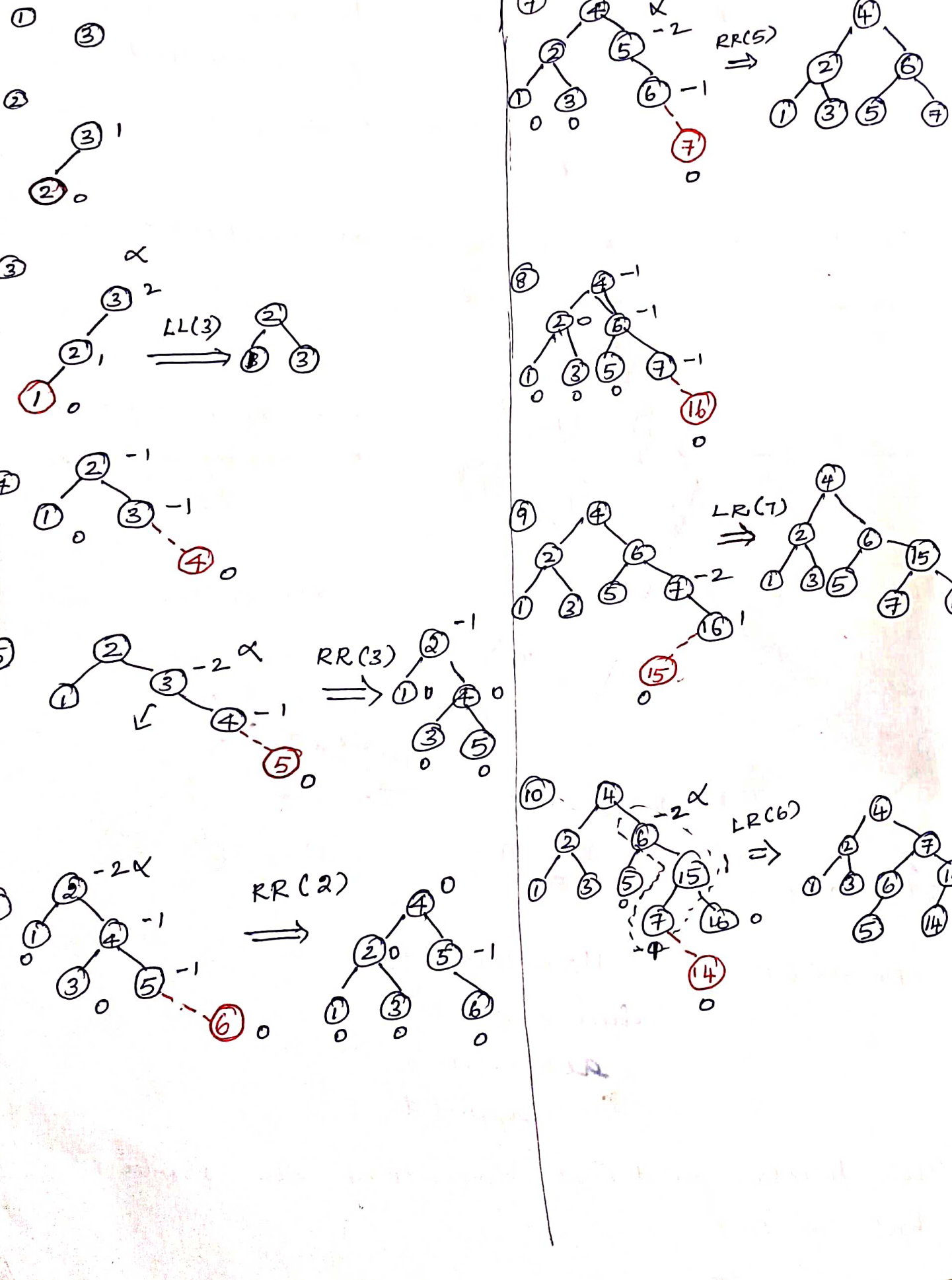
Delete (72)

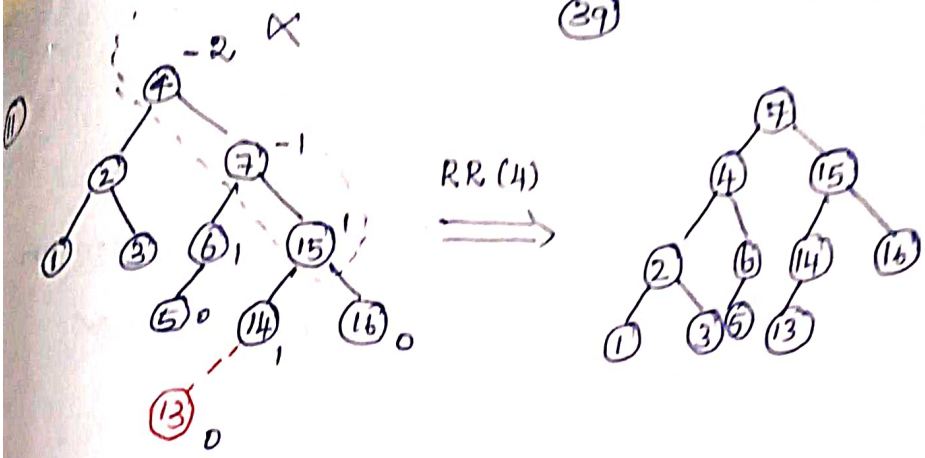
Here B is -1, hence double rotation is applied.

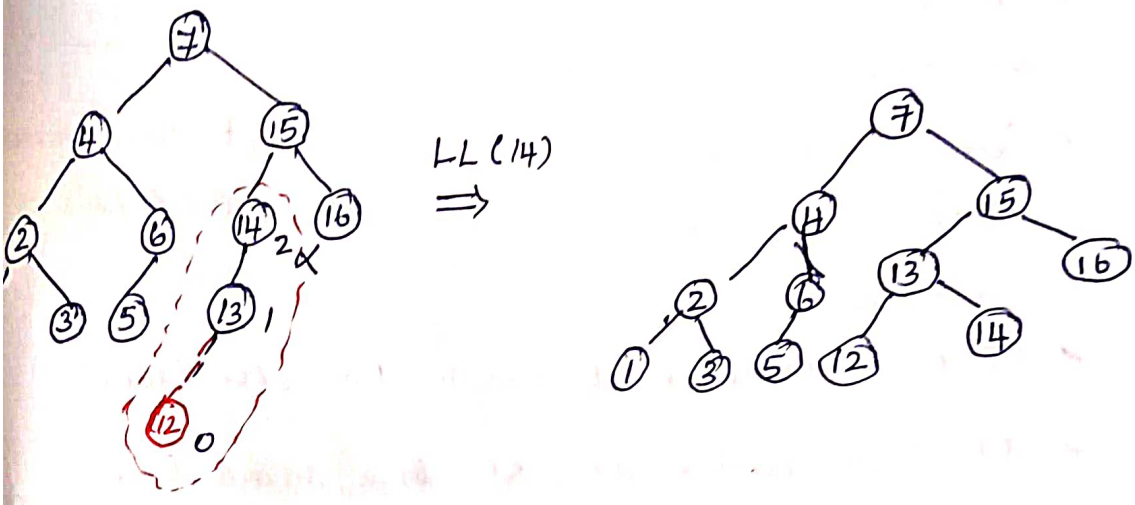
FindMin, FindMax and Find operations are similar to that of BST.

Construction of an AVL tree:

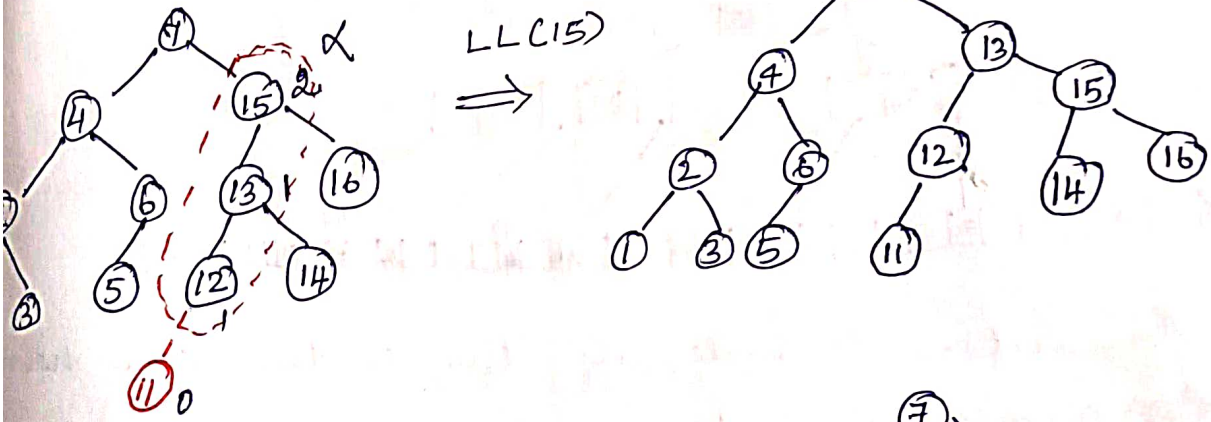
3, 2, 1, 4, 5, 6, 7, 16, 15, 14, 13, 12, 11, 10.

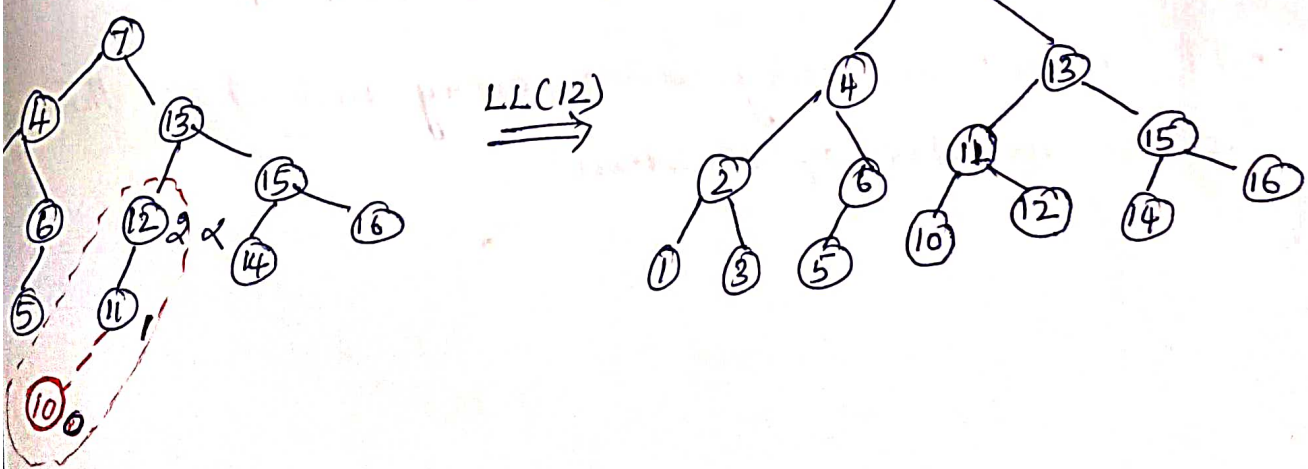






Insert 11





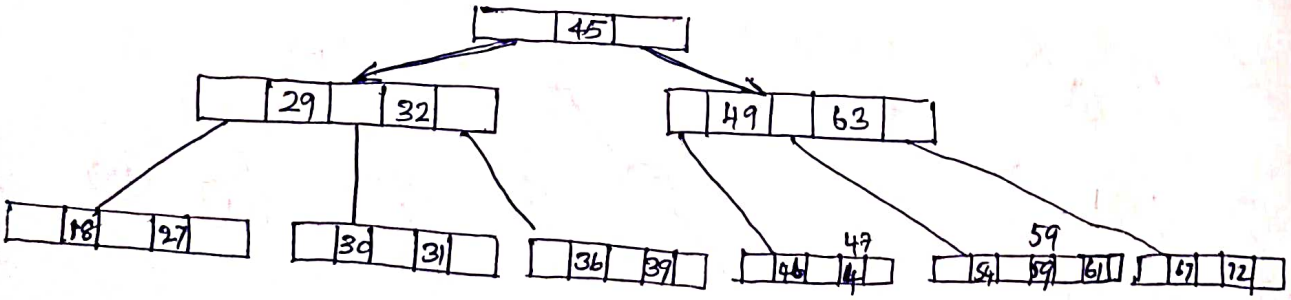
B-TREES

- B-tree is a specialized M-way search tree.
- A B-tree of order m can have a maximum of m-1 keys and m pointers to its sub tree

Properties:

* Every node in the B tree has at most 'm' children. (maximum)
* Every node in the B tree except the root node and leaf node has atleast m/2 children.
* Root node has at least two children
* All leaf nodes are at the same level.

Eg: B tree of order 4



- An internal node in the B tree can have n number of children, where $0 \leq n \leq m$.
- It is not necessary that every node has the same number of children.

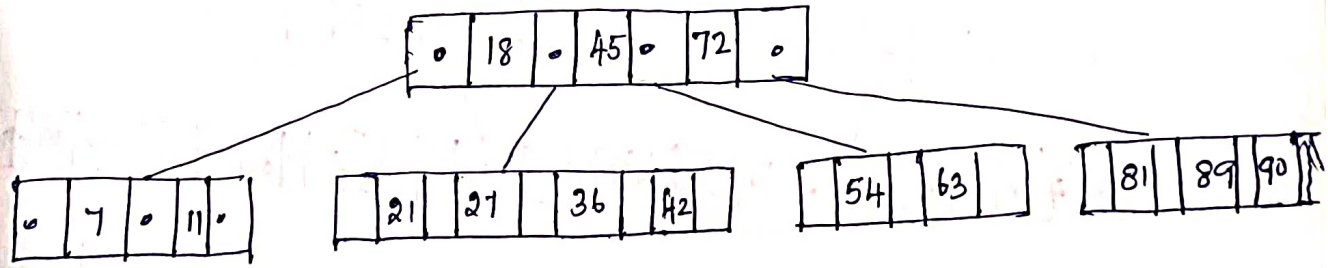
while performing insertion and deletion operations in a B-tree, the number of child nodes may change In order to maintain a minimum number of children, the internal nodes may be joined or plit.

Insertion:

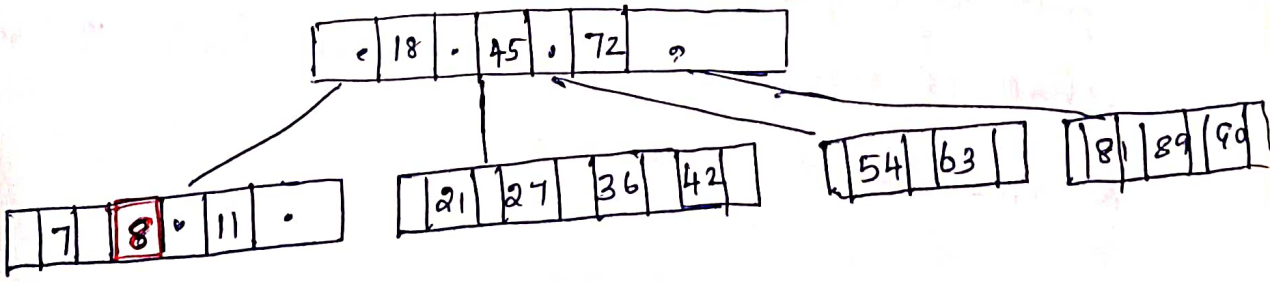
All the insertions are done at the leaf node

Cases:

case 1: If the leaf node is not full, ie it contains less than m−1 key values then insert the element in the node keeping the node's elements ordered.

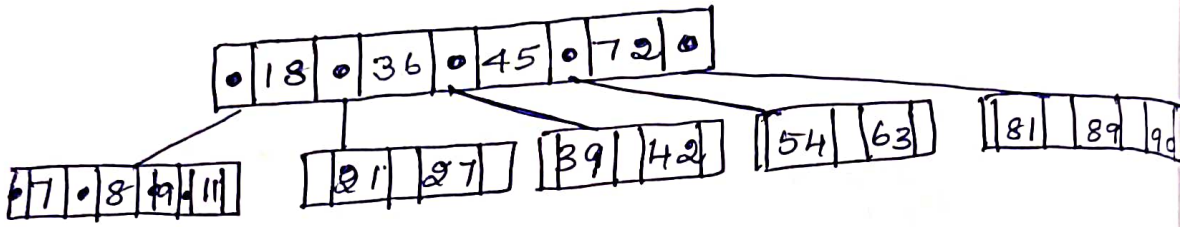


Insert 8



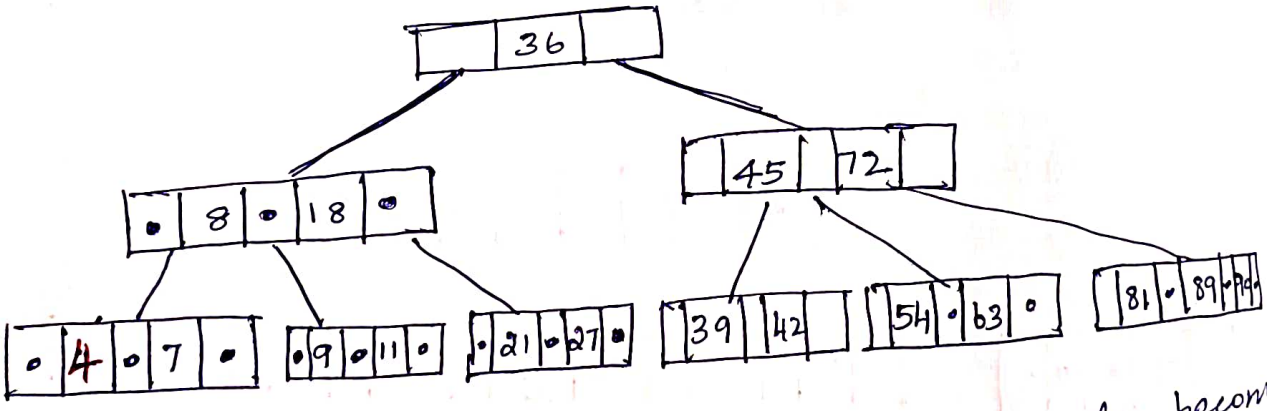
case 2: If the leaf node is full and root is not full, split the node into two nodes and move the median to its root.

Insert 39 in the above tree



case 3: If the leaf node full and root is also full.

Insert 4 in the above tree



When 4 is inserted, the leaf node becom full and pushed the median to root, since that is also full, it was split into 2 and a new node is created

Deletion:

Cases:

Case 1: If the leaf node contains more than the minimum number of key values, then delete the value

Case 2: If the leaf node does not contain $m/2$ elements, then fill the node by taking an element from the left or right sibling.

(a) If the left sibling has more than the minimum number of key values, push its largest key to its parent's node and pull down the intervening element from the parent node to the leaf node where the key is deleted.

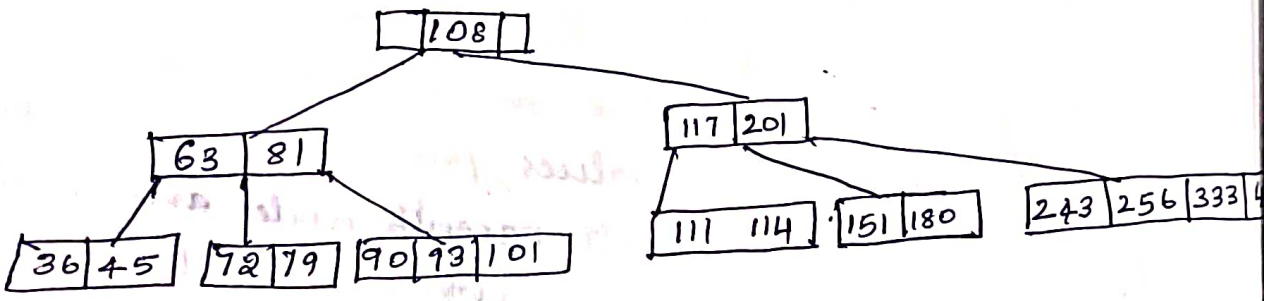
(b) If the right sibling has more than the minimum number of key values, push its smallest key into its parent node and pull down the intervening element from the parent node to the leaf node where the key is deleted.

Case 3: If both left and right siblings contain minimum, then create a new leaf node by combining the two leaf nodes and intervening element of parent node.

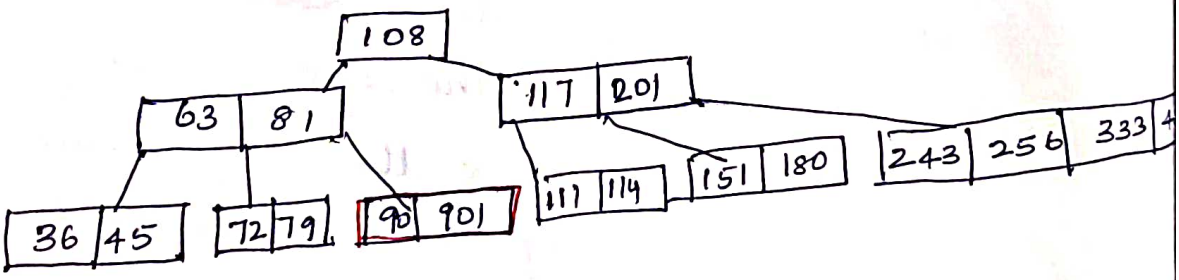
If pulling the intervening element from the parent node leaves it with less than minimum number of keys in the node, then propagate the process upwards.

Case 4: To delete an internal node, promote the successor or predecessor of the key to be deleted.

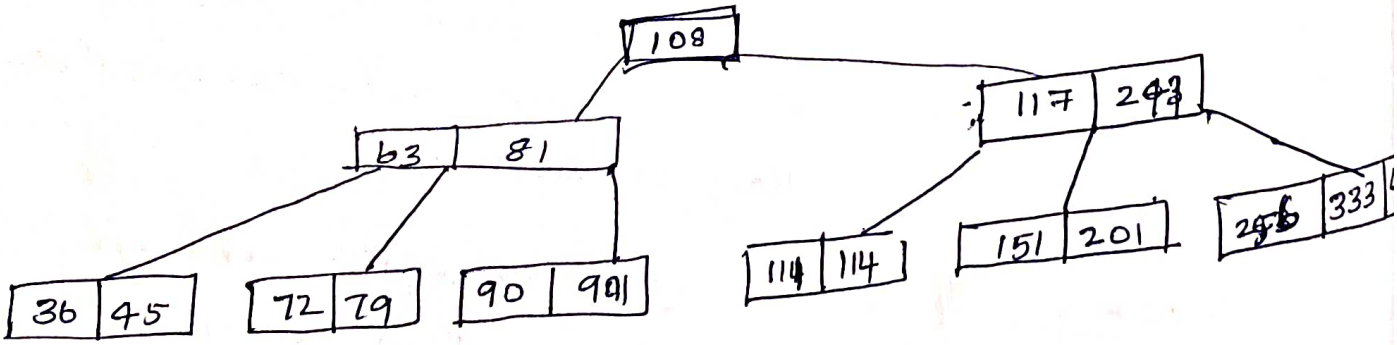
~~Case 1:~~ Example for case 1:



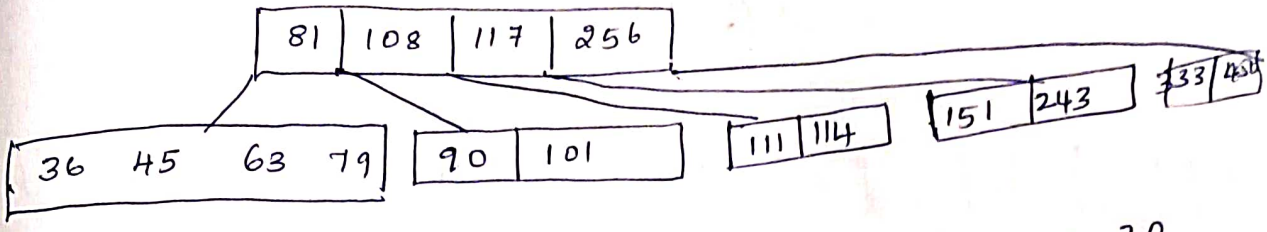
Delete 93



Example for case 2: Delete 180

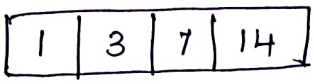


Example for case 3: Delete 72

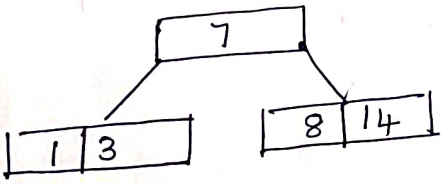


Construct a B-tree: 3, 14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20
26, 4, 16, 18, 24, 25 & 19

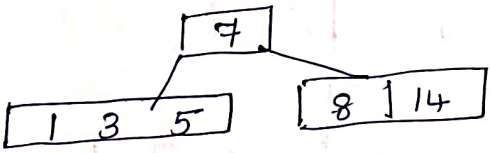
Order of 5



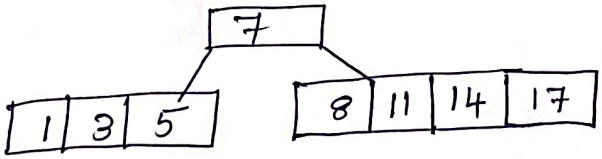
Insert 8



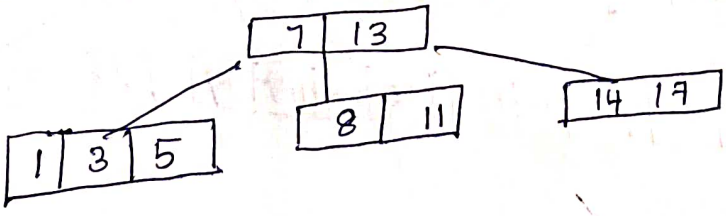
Insert 5



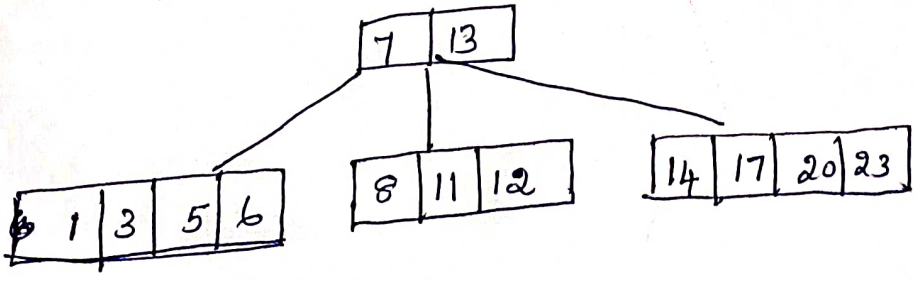
Insert 11, 17



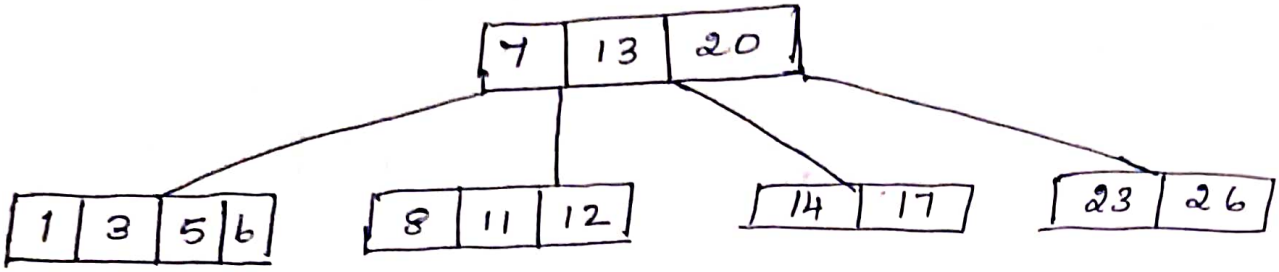
Insert 13, split right into 2



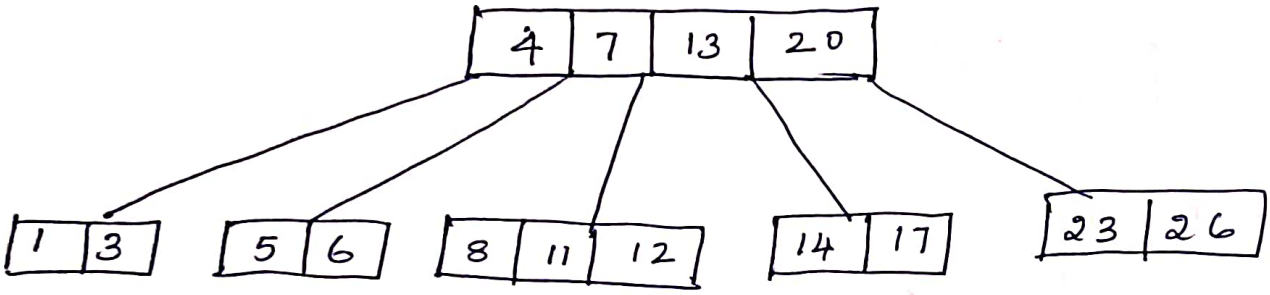
Insert 6, 23, 12, 20



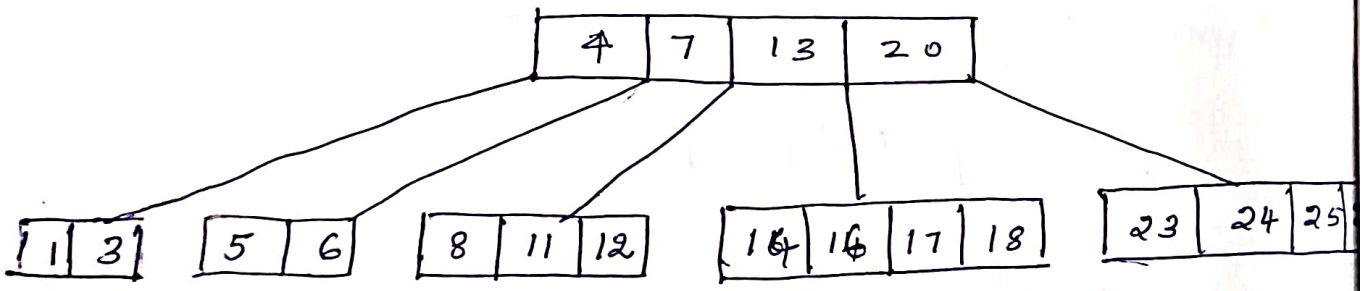
7. Insert 26



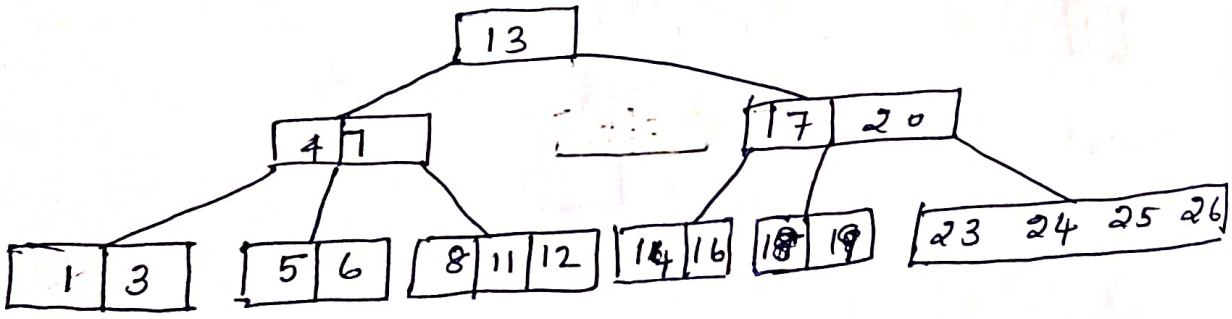
8. Insert 4



9. Insert 16, 18, 24, 25



10. Insert 19

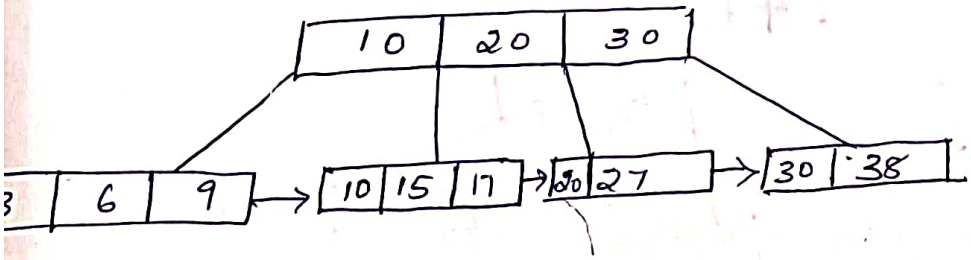


- It can be used to perform a wide range of queries easily as leaves are linked to nodes at the upper level.
- Height of a tree is less and balanced
- Supports both random and sequential access to records.
- Keys are used for indexing.

Insertion:

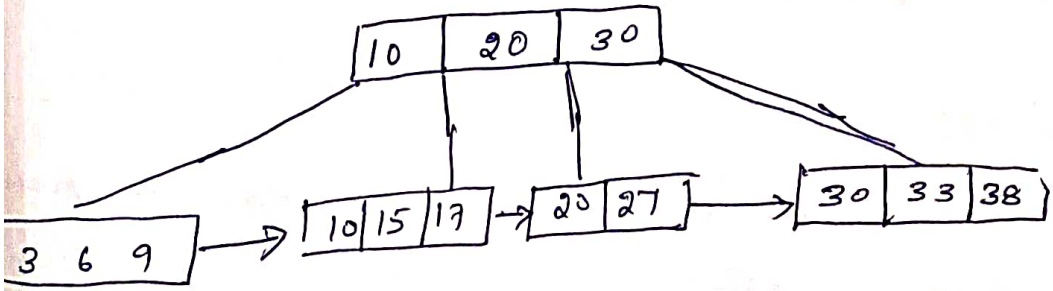
Algorithm:

1. Insert a new node in leaf
2. If the leaf node overflows, split the node and copy the middle element to next index node.
3. If the index node overflows, split that node and move the middle element to next index page.



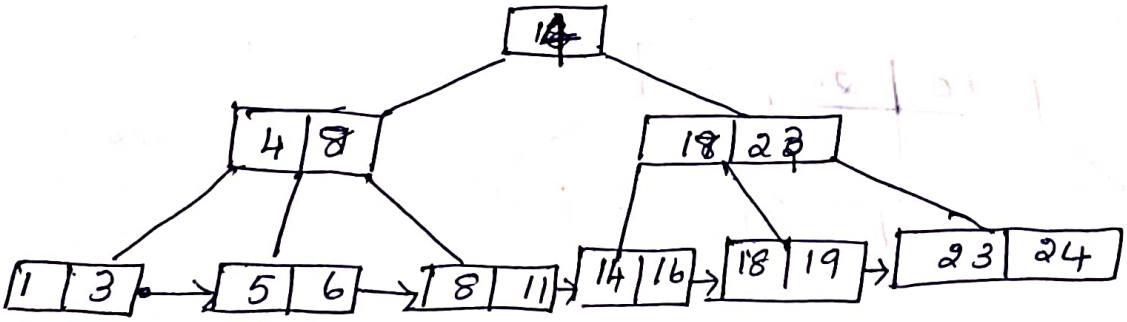
B+ tree of order 4

Insert (33)



B+ TREES:

- A B+ tree is a variant of B tree which stores sorted data.
- B+ tree can store both keys and records in its interior nodes. But B+ tree stores all the records at the leaf level of the tree only keys are stored in the interior nodes.
- Leaves are linked to one another in a linked list.
- B+ trees are used to store large amounts of data that cannot be stored in the main memo Leaf nodes are stored on the secondary storage and internal nodes are stored in the main memory.
- B+ trees stores data only in the leaf nodes and all other nodes store index values.



Eg: B+ tree of order 3.

Advantages:

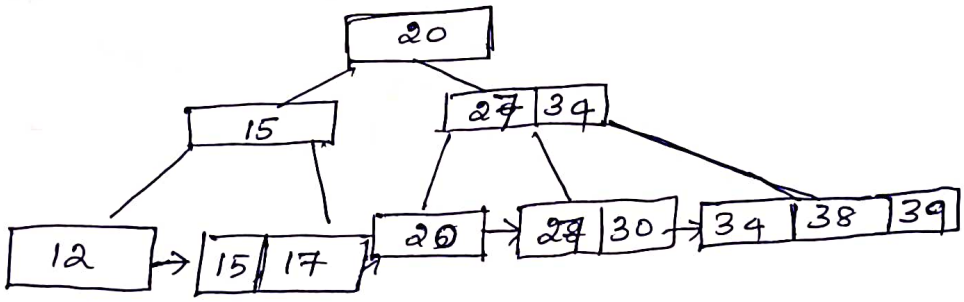
- Records can be fetched in equal num of disk accesses

Deletion:

- Deletion is always done from a leaf node
- If deleting a data element leaves that node empty, then the neighbaing nodes are are examined and merged.

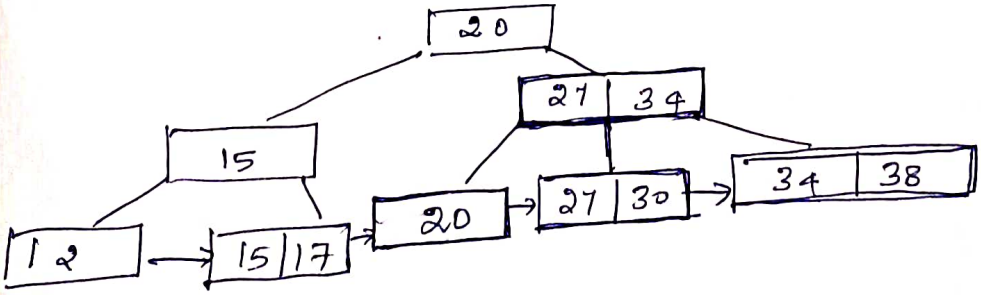
---

1. Delete the key and data from leaves
2. If the leaf node underflows, merge that node with the sibling and delete the key in bbtwen them.
3. If the index node underflows, merge that node with sibling and move down the key in between them.



B+ tree

Delete (39)

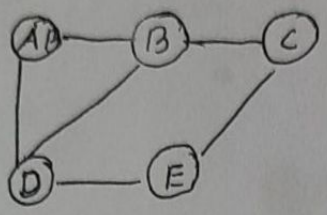


UNIT IV

(1) 57 copies

GRAPHS:

A graph 'G' is defined as an ordered set (V, E) where V(G) represents the set of vertices and E(G) represents the edges that connect these vertices.



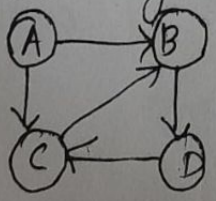
undirected graph.

$V(G) = \{A, B, C, D, E\}$

$E(G) = \{(A,B), (B,C), (A,D), (B,D), (D,E), (C,E)\}$

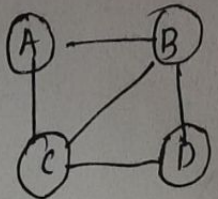
Terminologies:

Directed graph: In a directed graph, all the edges have direction associated with it. It is also called as digraph.



Directed graph.

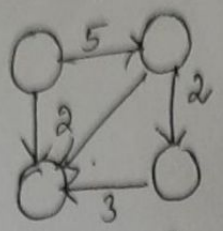
Undirected graph: In an undirected graph, edges do not have any direction associated with them.



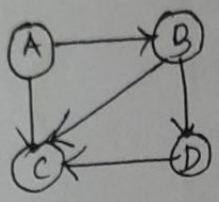
(29)

Weighted graph: All the edges in a graph have a weight or cost associated with it.

Eg:



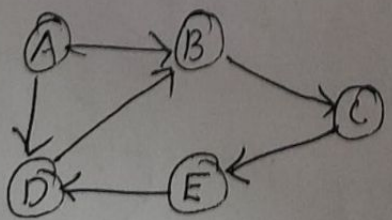
Unweighted graph: Edges of a graph will not have any cost or weight.



Path: A path in a graph is a sequence of vertices from source 'u' to destination 'v'. It is written as $P = \{v_0, v_1, v_2 \ldots v_n\}$.

Length of a path: Number of edges on a path.

Degree of a node: Degree of a node u, is the total number of incoming and outgoing edges.



Path (A, D) = A, B, C, E, D.

Length = 5

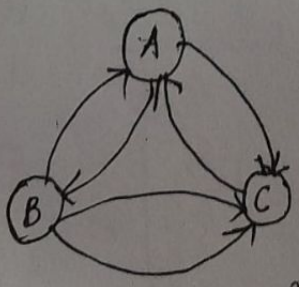
Degree (B) = 3.

outdegree: Number of outgoing edges of a node.
Indegree: Number of incoming edges of a node.

Connected graph: An undirected graph is connected if there is a path from every vertex to every other vertex. A directed graph with this property is called strongly connected graph. If not it is weakly connected graph.

Complete graph: A complete graph is a graph in which there is an edge between every pair of vertices.

Strongly connected digraph: A digraph is said to be strongly connected if and only if there exist a path between every pair of nodes in G. That is, if there is a path from node u to v, then there must be a path from node v to u.



Cycle: A path, in which the first and the last vertices are the same.

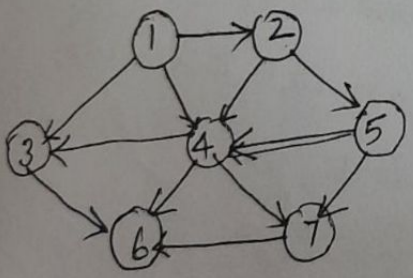
REPRESENTATION OF GRAPHS

2 ways.

* Adjacency matrix representation.
* Adjacency list representation.

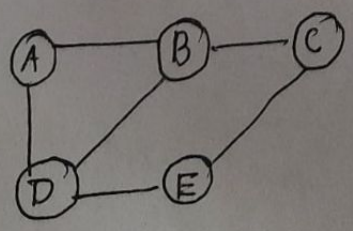
Adjacency matrix representation.

In an adjacency matrix, the rows and columns are labelled by graph vertices. An entry a_{ij} in the adjacency matrix will contain 1, if vertices V_i and V_j are adjacent to each other. If the nodes are not adjacent a_{ij} will be set to zero. Since an adjacency matrix contains only 0s and 1s it is called a bit matrix or a Boolean matrix



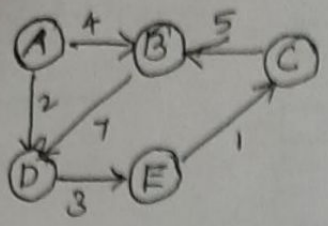
Directed Graph.

	1	2	3	4	5	6	7
1	0	1	1	1	0	0	0
2	0	0	0	1	1	0	0
3	0	0	0	0	0	1	0
4	0	0	1	0	0	1	1
5	0	0	0	1	0	0	1
6	0	0	0	0	0	0	0
7	0	0	0	0	0	1	0



undirected graph.

	A	B	C	D	E
A	0	1	0	1	0
B	1	0	1	1	0
C	0	1	0	0	1
D	1	1	0	0	1
E	0	0	1	1	0



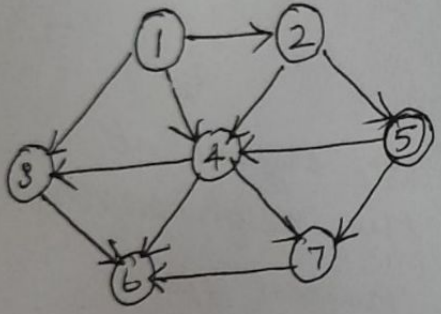
Weighted Graph

	A	B	C	D	E
A	0	4	0	2	0
B	0	0	0	7	0
C	0	5	0	0	0
D	0	0	0	0	7
E	0	0	1	0	0

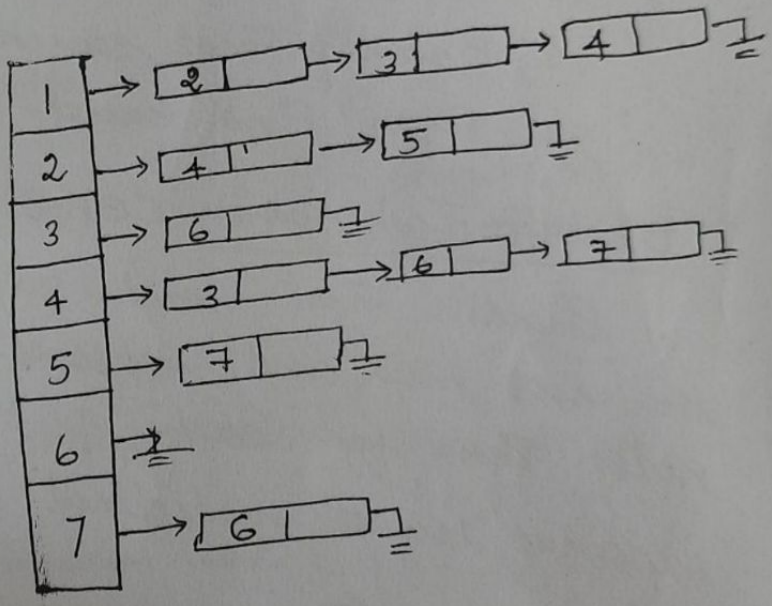
(5)

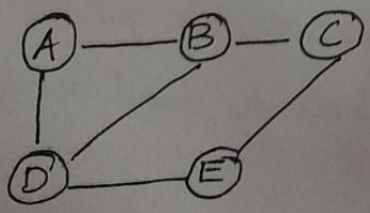
Adjacency list representation:

An adjacency list is another way in which graphs can be represented in the computer memory. This ~~case~~ structure consists of a list of all nodes in G. Thus every node is linked to its own list that contains the names of all other nodes that are adjacent to it.

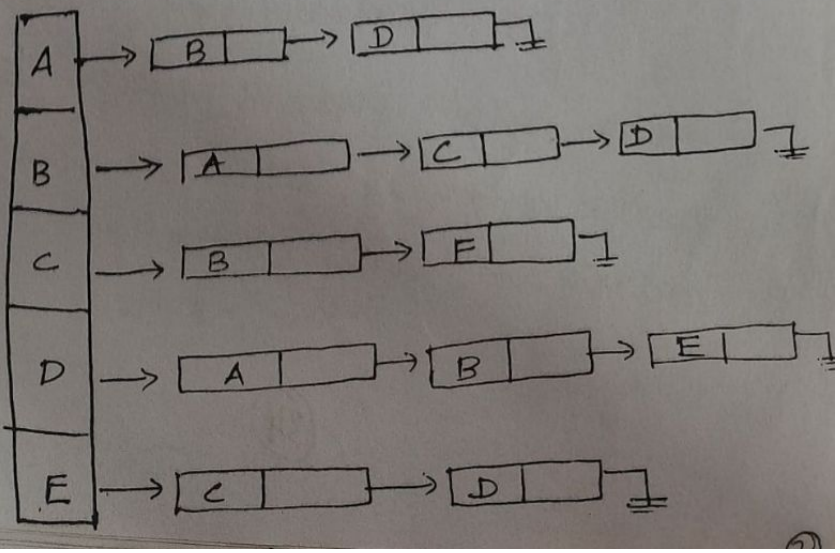


Directed Graph

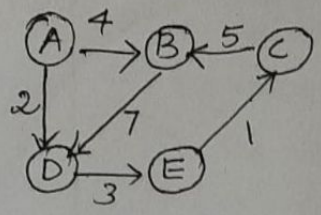




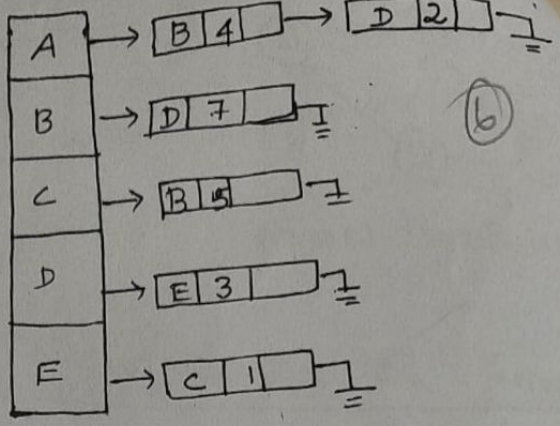
undirected graph



(33)



Weighted Graph.



(6)

GRAPH TRAVERSAL

Graph traversal is the process of examining the nodes and edges of the graph. There are Two standard methods of graph traversal.

1. Breadth First search (BFS)
2. Depth First search (DFS)

1. Breadth First Search (BFS).

Breadth First search algorithm begins at the root node and explores all the neighbouring nodes. Then for each of those nearest nodes, explores their unexplored neighbouring nodes. and so on.

BFS traverses a graph in a breadth-wards motion and uses a queue to remember to get the next vertex to start a search when a dead end occurs in any iteration.

(34)

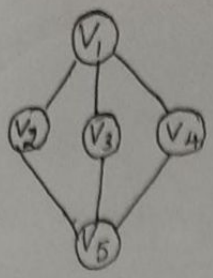
Algorithm:

1. Create an empty queue
2. Start with any node 'v' and enqueue it in the queue. and mark it as visited.
3. Dequeue a node from the queue and enqueue its adjacents to the queue and Mark them as visited. if not visited.
4. Repeat step 3 until the queue is empty.

Routine:

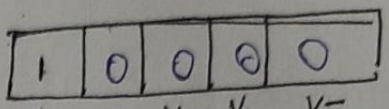
```
void BFS (Vertex v)
{
   Initialize (Q);                    v → source vertex
   visited [u] = 1;                   u → adjacent vertex
   Enqueue (u, Q);
   while ( ! IsEmpty (Q))
    {
      u = Dequeue (Q);
      print u;
      for all vertices v adjacent to u do
         if (visited [v] == 0) then
          {
             Enqueue (v, Q);
             visited [v] = 1;
          }
    }
}
```

Eg:

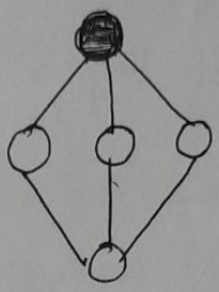


Iteration 1: Enqueue (V_1), mark it as visited.

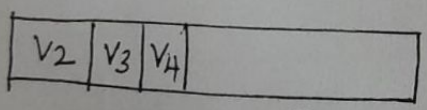
Queue Q



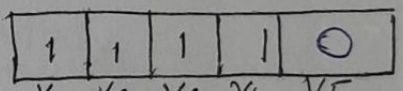
Visited.



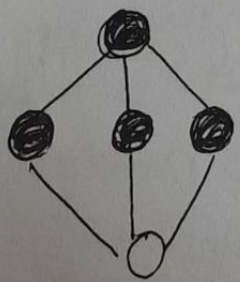
Iteration 2: Enqueue (V_2), Enqueue (V_3), Enqueue (V_4) and dequeue (V_1) and print it



Queue Q

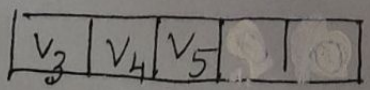


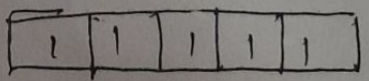
Visited.

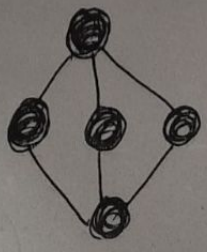


V_1

Iteration 3: Enqueue (V_5) and dequeue (V_2) and display.







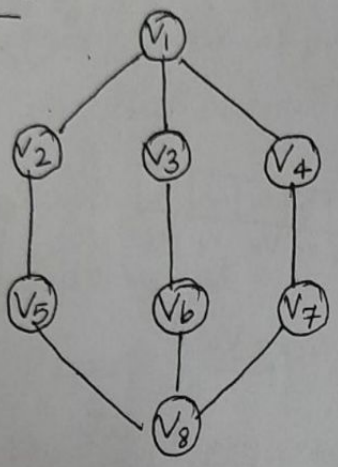
$V_1 \rightarrow V_2$

36

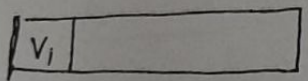
Iteration 4 : since all the nodes are visited dequeue the ~~elements~~ nodes and print all the nodes.

BFS : $V_1 \rightarrow V_2 \rightarrow V_3 \rightarrow V_4 \rightarrow V_5$

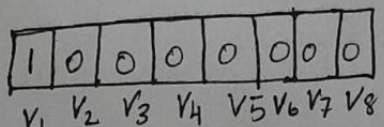
Eg 2 :



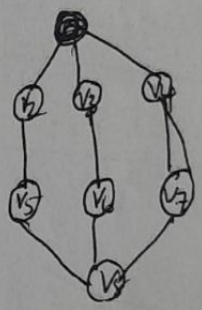
Iteration 1 :



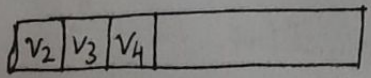
Queue



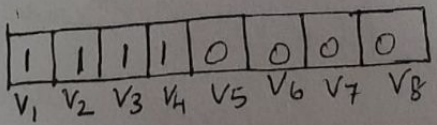
visited.

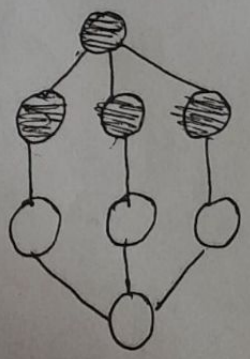


Iteration 2 : (V_1 dequeued, V_2, V_3, V_4 enqueued).



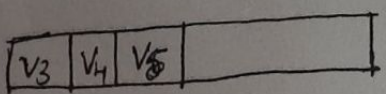
Queue

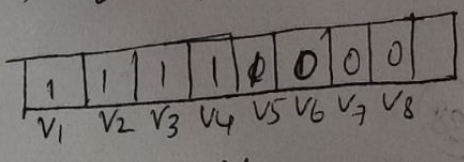


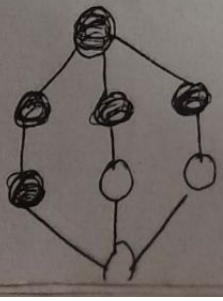


print : V_1

Iteration 3 : V_2 dequeued, ~~V_5~~ enqueued.



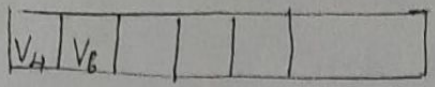




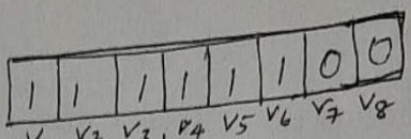
print : $V_1 \rightarrow V_2$

(37)

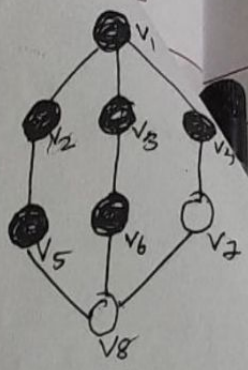
Iteration 4: V_3 dequeued, V_6 enqueued.



Queue

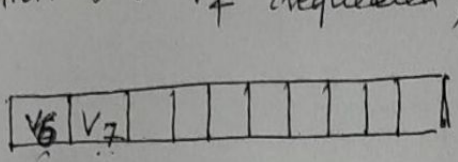


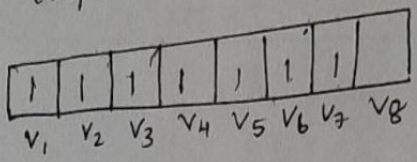
visited

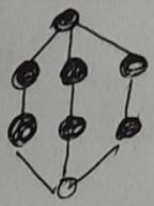


print : $V_1 \rightarrow V_2 \rightarrow V_3$

Iteration 5: V_4 dequeued, V_7 enqueued.

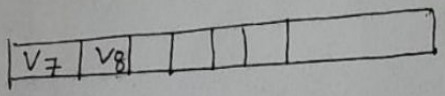


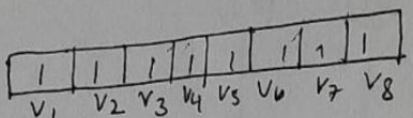


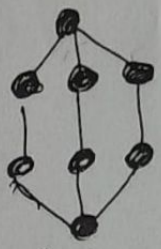


Print : $V_1 \rightarrow V_2 \rightarrow V_3 \rightarrow V_4$

Iteration 6: V_6 dequeued, V_8 enqueued.







print : $V_1 \rightarrow V_2 \rightarrow V_3 \rightarrow V_4 \rightarrow V_6$

Iteration 7:

Dequeue all the nodes and print it

BFS : $V_1 \rightarrow V_2 \rightarrow V_3 \rightarrow V_4 \rightarrow V_6 \rightarrow V_7 \rightarrow V_8$

2. Depth First search: (DFS).

DFS starts with a node and visits one of its adjacent and further moves one by one until it fonds a dead end. then it come back to previous ~~node~~ node and repeats the same until visiting all the nodes.

DFS traverse a graph in a depth wand motion and uses a stack to remember to get the next vertex to re-start the search when a dead end occurs.

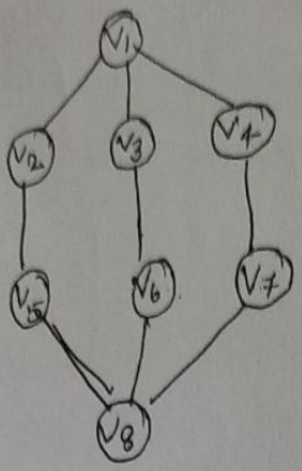
Algorithm:

1. Choose any one of the node as starting node and push it in the stack, ~~and~~ Display the same in the output and mark it visited.
2. Push the adjacent nodes in the stack and display it.
3. Repeat step 2 until finds a dead end.
4. If no adjacent node, perform pop operation. Again repeat step 2.

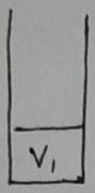
Routine:

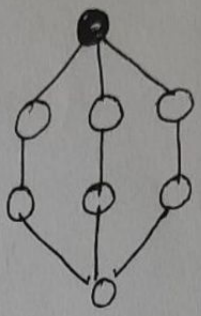
```
void DFS (Vertex V)
{
    visited[v] = True;
    for each w adjacent to v
        if (!visited[w])
            DFS(w);
}
```

Eg:



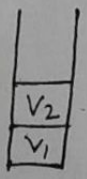
Iteration 1:

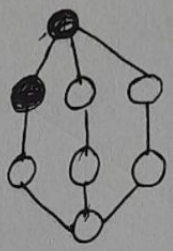




V_1

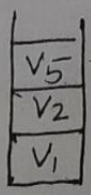
Iteration 2:

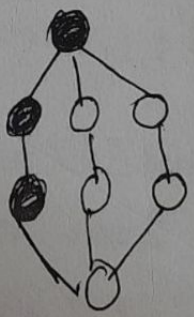




$V_1 \rightarrow V_2$

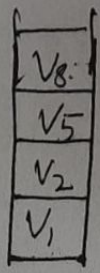
Iteration 3:

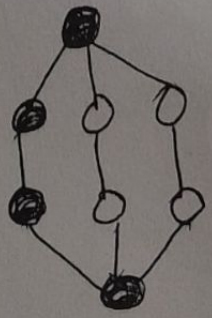




$V_1 \rightarrow V_2 \rightarrow V_5$

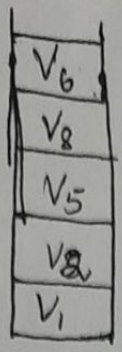
Iteration 4:

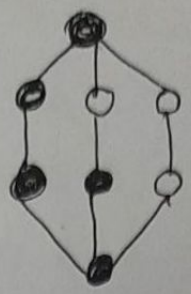




$V_1 \rightarrow V_2 \rightarrow V_5 \rightarrow V_8$

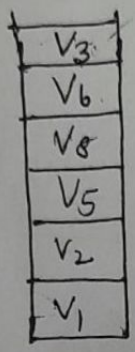
Iteration 5

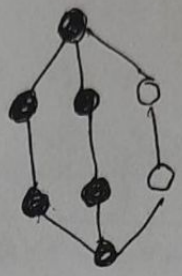




$V_1 \rightarrow V_2 \rightarrow V_5 \rightarrow V_8 \rightarrow V_6$

Iteration 6

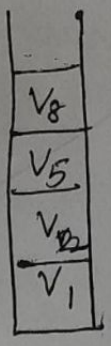




$V_1 \rightarrow V_2 \rightarrow V_5 \rightarrow V_8 \rightarrow V_6 \rightarrow V_3$

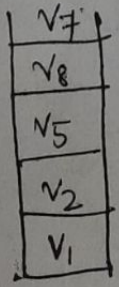
Iteration 7:

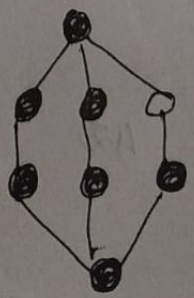
Reached dead end hence pop out V_3, and check whether V_6 has any unvisited adjacent. Since it does not has any unvisited adjacent, it is also poped out.



Iteration 8:

Since V_8 has an unvisited adjacent ie V_7 it is pushed.

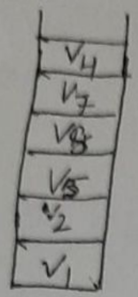


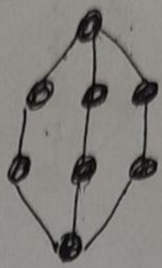


$V_1 \rightarrow V_2 \rightarrow V_5 \rightarrow V_8 \rightarrow V_6 \rightarrow V_3 \rightarrow V_7$

Iteration 9:

since v_4 has adjacent, it is pushed:





$v_1 \to v_2 \to v_5 \to v_8 \to v_6 \to v_3 \to v_7 \to v_4$.

∴ since all the nodes are visited stop the process and pop out all the nodes

DFS : $v_1 \to v_2 \to v_5 \to v_8 \to v_6 \to v_3 \to v_7 \to v_4$.

TOPOLOGICAL SORTING:

Topological sorting is the process of ordering the nodes of a directed acyclic graph in a linear manner. That is, if there is a path from v_i to v_j means v_j appears after v_i

Ordering is only possible for directed acyclic graph and not possible for the graph with cycle.

42 / Kulkarni Yesbee Sree Krishna

TOPOLOGICAL SORTING

- Topological sorting is the process of ordering the vertices in directed acyclic graph such that if there is a path from v_i to v_j, then v_j appears after v_i.
- Topological ordering is not possible if the graph has a cycle.
- Ordering is not necessarily unique; any legal ordering can be done.
- Directed Acyclic Graph (DAG) is a directed graph without cycles.

Algorithm:

step 1: Calculate the indegree of all the vertices

step 2: Identify the vertices whose indegree is zero and insert into a queue.

step 3: Dequeue an element from the queue (i.e a vertex from the queue) and remove all the edges connected to it.

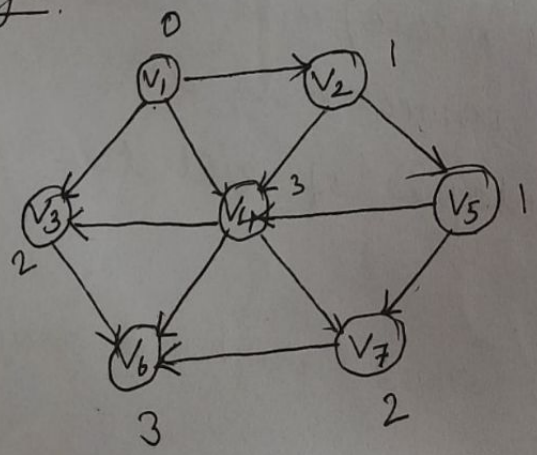
step 4: Update the indegree of all the vertices

step 5: Repeat step 2 to step 3 until the graph is empty.

Routine:

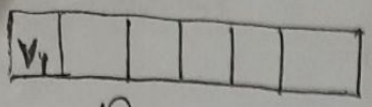
```
void Topsort ( Graph G)
{
  int counter;
  Vertex V, W;
  for(counter=0; counter < NumVertex ; counter++)
  {
    V=FindNewVertexofDegreeZero();
    if (V== NotAVertex)
    {
      Error ("Graph has a cycle");
      break;
    }
    TopNum[V]= counter ;
    for each w adjacent to V
      Indegree[W]--;
  }
}
```

Eg:



V_1	0
V_2	1
V_3	2
V_4	3
V_5	1
V_6	3
V_7	2

Step 1: Enqueue V_1 (since its indegree is 0)



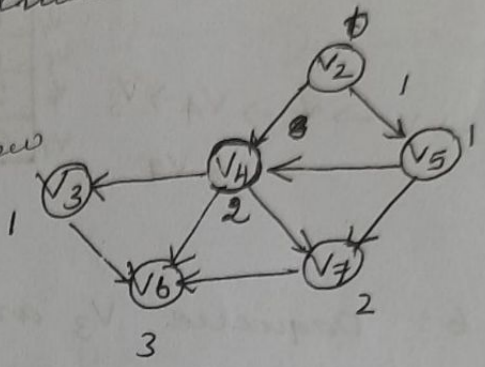
Queue.

V_1

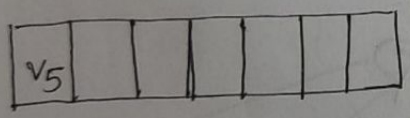
Step 2: Dequeue V_1 from queue and enqueue V_2

V_1

V_1	0
V_2	0 ← New
V_3	1
V_4	2
V_5	1
V_6	2
V_7	2

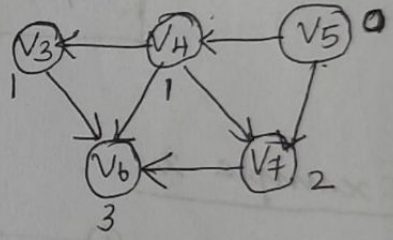


Step 3: Dequeue V_2 from queue and enqueue V_5

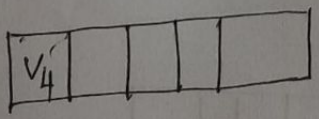


$V_1 \rightarrow V_2$

V_1	0
V_2	0
V_3	1
V_4	1
V_5	0 ← New.
V_6	3
V_7	2

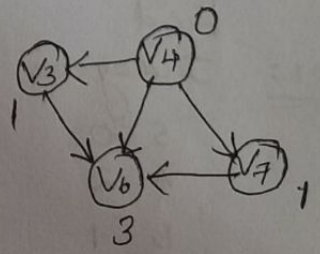


Step 4: Dequeue V_5 from queue and ~~deq~~ enqueue V_4

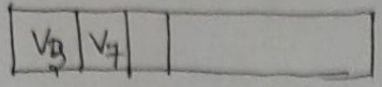


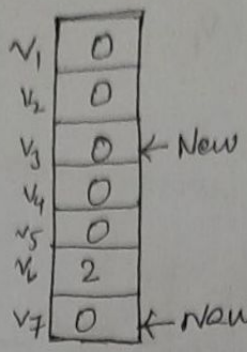
$V_1 \rightarrow V_2 \rightarrow V_5$

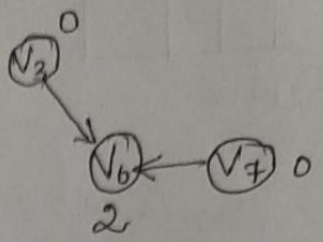
V_1	0
V_2	0
V_3	1
V_4	0 ← New
V_5	0
V_6	3
V_7	1



Step 5: Dequeue V_4, Enqueue V_3 & V_7







$V_1 \rightarrow V_2 \rightarrow V_5 \rightarrow V_4$

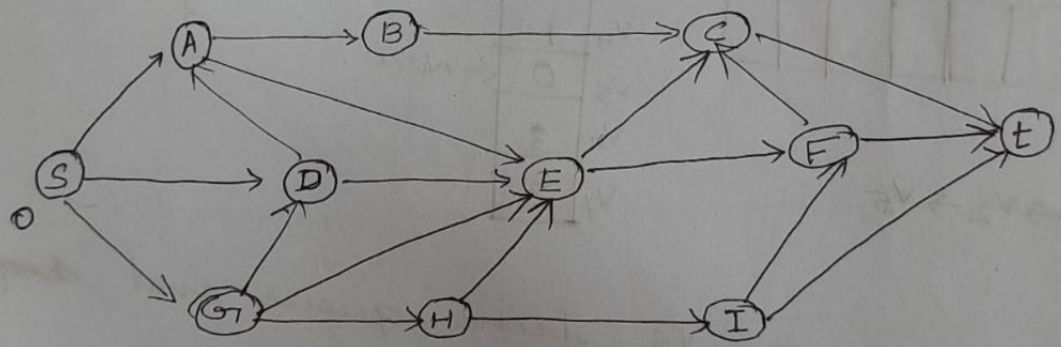
Step 6: Dequeue V_3 and V_7.

$V_1 \rightarrow V_2 \rightarrow V_5 \rightarrow V_4 \rightarrow V_3 \rightarrow V_7$

Topological sorting is

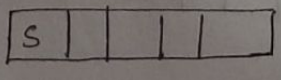
$V_1 \rightarrow V_2 \rightarrow V_5 \rightarrow V_4 \rightarrow V_3 \rightarrow V_7$

Example 2:

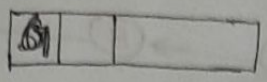


Step 1: Find Indegree of all vertices. and enqueue S.

S	0
A	2
B	1
C	3
D	2
E	4
F	2
G1	1
H	1
I	1
t	3

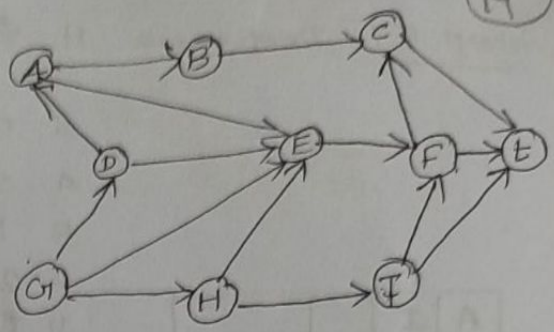


Step 2: Dequeue S and enqueue G1.

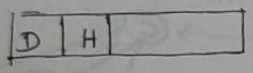


S

S	0
A	1
B	1
C	2
D	1
E	4
F	2
G	0 ← New
H	1
I	1
t	3

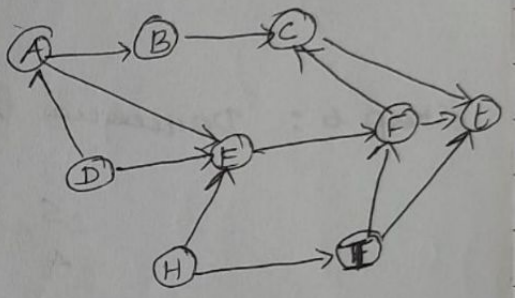


Step 3: Dequeue G1, Enqueue D & H

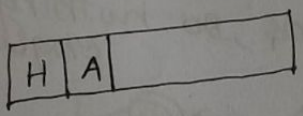


S → G1

S	0
A	1
B	1
C	2
D	0 ← New
E	3
F	2
G	0
H	0 ← New
I	1
t	3

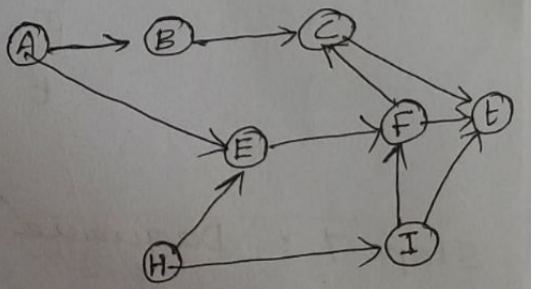


Step 4: Dequeue D, Enqueue A

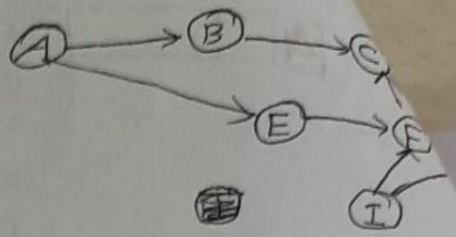


S → G1 → D

S	0
A	0 ← New
B	1
C	2
D	0
E	2
F	2
G	0
H	0
I	1
t	3



Step 5: Dequeue H, Enqueue I

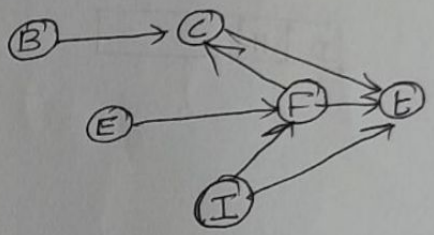


S	0
A	0
B	1
C	2
D	0
E	1
F	2
G	0
H	0 ~~Head~~
I	0 ← New
t	3

A	I		

S → G → D → H

Step 6: Dequeue A, Enqueue B & E

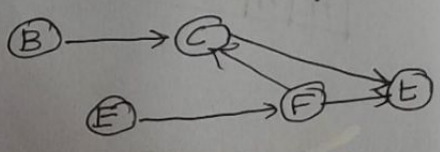


S	0
A	0
B	0 ← New
C	2
D	0
E	0 ← New
F	2
G	0
H	0
I	0
t	3

I	B	E	

S → G → D → H → A

Step 7: Dequeue I, no new zeros, so nothing enqueued

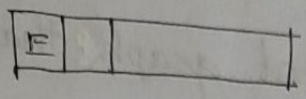


B	0
C	2
E	0
F	1
~~G~~	
t	2

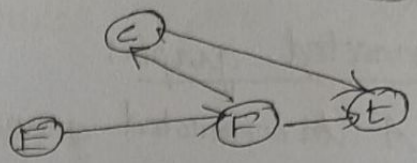
B	E		

S → G → D → H → A → I

step 8: Dequeue B

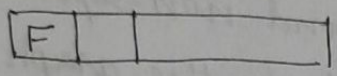


C 1
E 0
F 1
t 2

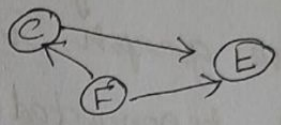


S → G → D → H → A → I → B

step 9: Dequeue E, Enqueue F

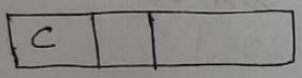


C 1
F 0 ← New
t 2



S → G → D → H → A → I → B → F

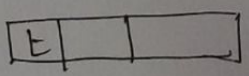
Step 10: Dequeue F, enqueue C



C 0
t 1

S → G → D → H → A → I → B → E → F

Step 11: Dequeue C, and enque t



t 0

S → G → D → H → A → I → B → E → F → C → t