DON BOSCO COLLEGE
DEPARTMENT OF COMPUTER SCIENCE
STUDY MATERIAL
DATA STRUCTURES AND ALGORITHMS

B.Sc. COMPUTER SCIENCE SEMESTER

DATA STRUCTURES AND ALGORITHMS

UNIT -I

Algorithms (Analysis and design): Problem solving - Top-Down and Bottom- up approaches to algorithm design - Use of algorithms in problem solving - Design, Implementation, Verification of algorithm - Efficiency analysis of algorithms: Space, Time complexity, and Frequency count - Sample algorithms: Exchange the value of two variables - Summation of set of numbers - Decimal to Binary conversion - Sorting - Factorial - Fibonacci - Finding a largest Number in an array - Reverse the order of elements in array.

UNIT- II

Introduction: Definitions - concepts - Overview - Implementation of Data Structures. Arrays: Definition - Terminology - One dimensional array - Multi Dimensional Array. Stacks: Introduction - Definition - Representation of stacks - Operations on stacks - Applications of stack: Evaluation of Arithmetic Expression- Implementation of Recursion- Factorial Calculation

UNIT –III

Queues: Introduction - Definition - Representation of Queues -Various Queue Structures: Circular Queue - De-queue - Priority Queue - Applications of Queues: CPU Scheduling. Linked List: Definition - Single Linked List - Double Linked List - Circular Double Linked List - Applications: Sparse Matrix - Polynomial.

UNIT- IV

Trees: Terminologies - Definitions &Concepts - Representation of Binary tree - Operations on Binary Tree - Types of Binary Trees: Expression Tree - Binary Search Tree - Heap Tree - Red Black Tree. Graphs: Introduction - Graph terminologies - Representation of Graphs - Operations on Graphs - Applications of Graph: Shortest path problem - Minimum Spanning Tree: Kruskal and Prims Algorithm.

UNIT- V

Searching: Terminologies - Linear Search techniques with - Array, Linked List, and Ordered List - Binary search - Non Linear Search- Binary Tree Searching - Binary Search Tree Searching .Sorting: Terminologies - Sorting Techniques - Insertion Sort - Selection sort - Bubble sort - Quick sort - Merge sort.

TEXT BOOKS

1. Sathish Jain, Shashi Singh, "Data Structure Made Simple", 1st Edition, BPB Publications, New Delhi, 2006. 2. Debasis Samanta, "Classic Data Structures", 2nd Edition, PHI Learning, New Delhi, 2009. REFERENCE BOOKS 1. Aprita Gopal, "Magnifying Data Structures", 1st Edition, PHI Learning, New Delhi, 2010. 2. Chitra A &Rajan PT, "Data Structures", 2nd Edition, Vijay Nicole Publications, 2016

UNIT-I

ALGORITHMS (ANALYSIS AND DESIGN)

PROBLEM SOLVING

- To solve a problem using computer to write step by step solution first and write simple instructions for each operation.
- There might be a number of methods to solve the problem.

These steps are:

- Formulating the problem and deciding the data types to be entered.
- Identifying the steps of computation that are necessary for getting the solution.
- Identifying decision points.
- Finding the result and verifying the values.

Procedure for Problem Solving:

- Problem solving is a logical process of breaking down the problem into smaller parts each of which can be solved step by step to obtain the final solution.
- Step by step to obtain the final solution.

Six basic steps in solving a problem are:

- First spend some time in understanding the problem. This means try to formulate a problem correctly.
- Construct a list of variables that are needed to find the solution of the problem.
- Decide the layout for the output.
- Select a programming method best suited to solve the problem and then only carryout the coding using a suitable programming language.
- Test the program. Select test data so that each part of the program would be checked for correctness.
- Finally use data validation steps to guard against processing of wrongly inputted data.

1) Understanding the Problem:

- Read each statement of the problem slowly and carefully by understanding the keywords.
- Use paper and pencil to solve the problem manually for some test data.

Ex: Accept a value n and find the sum of first N even integers.

Solution:

- To take an input value, say a number 6 is given as the value of N.
- Next get the 6 integer values.
- The first 6 even integers are 2, 4, 6, 8, 10, and 12.
- The sum is 42.
- ✓ Hence the solution is to be made the sum of first 6 even integers.

2) Construction of the list of variables:

- The names chosen for the variables should be an aid to memory.
- The variables may be I, SUM and COUNT as given below:
  - Generate even integers 2, 4, 6…. (I)
  - Total the sum of even integers 2+4+6…. (SUM)
  - Count the number of even integers 1, 2, 3…. (COUNT)
- Finally 4 variables for this problem is

N	: To be entered by the user
I	: To generate even integers
COUNT	: To keep the number of even integer that has been summed.
SUM	: An accumulator that will hold the current total value of even integers.

3) Output design:

- The output report should be easily understandable by a reader.

Ex:

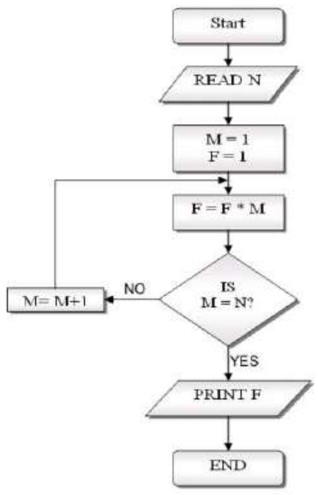
No of first even integer's	**Total sum**
6	42

- The output format should have the following characteristics

- Attractive
- Easy to read and
- Self-explanatory

4) Program Development:

- ➢ Draw a flowchart for the procedure for the steps 1, 2 & 3.
- ➢ Standard symbols should be used for drawing a flow chart, Then draw a flowchart for each part separately and combine them together using connectors.



5) Testing the program:

- ➢ Next run the program.
- ➢ This means by giving known values to the variables and by checking the result and thus comparing results with manually calculated values.

6) Validating the program:

- ➢ The user of your program may enter values, which are not expected by the program.
- ➢ Such values should be rejected by the procedure drawn by you, this known as validation of data.

TOP DOWN AND BOTTOM UP APPROACHES TO ALGORITHM DESIGN

1. Top- Down approach of problem solving:

- ➢ Top down design is the technique of **breaking down a problem into various sub tasks** needed to be performed.
- ➢ Each of these tasks is further broken down into separate subtasks and so on.
- ➢ The entire solution of the problem modules and joining them together will make the complete task of solving the complex problem.
- ➢ In top down design we initially describe the problem we are working with at the highest or most general level.
- ➢ All the operations at this level and individually break them down into simpler steps that begin to describe how to accomplish the tasks.
- ➢ Top-down process involves working from the most general form, down to the most specific form.

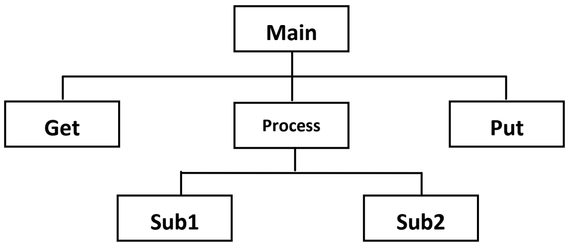


Fig: Top-Down modular design

Advantages of Top-Down approach:

- This approach allows analyst to remain **“on top of** “a problem and view the developing solution in the context.
- The solution always proceeds from the **highest level to the lowest level**.
- At each stage in the development the individual operation will be split up into a number of elementary steps.
- By dividing the problem into a number of sub problems, it is easier to share problem development.
- Bugs and debugging time grows quickly when the program is long.
- Top-down characteristic helps in faster completion of the solution of the complex problem.
- Increased comprehension of the problem.
- Unnecessary lower-level details are removed.
- Reduced debugging time.

2. Bottom-Up approach of problem Solving:

- A large and complex problem it may be difficult to solve.
- It may be easier to solve parts of the problem individually, taking the easier aspects first and thereby gaining the insight and experience to tackle the more difficult tasks, and finally join each of the solutions together to form the complete solution.

Disadvantages:

- There may be a lack of consistency among modules and thus more reprogramming may have to be carried.

USE OF ALGORITHMS IN PROBLEM SOLVING

- Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output.
- If the **algorithm is written in a language** that the computer can understand then such a set of instructions is called **a program**.

Rules:

- The starting point can be implemented.
- More than one starting point would create confusion about where to start, violating the ordering condition just stated.
- Accept to identify one or more of the steps in the algorithms as terminators.
- The characteristic of algorithms is that each individual operation must be both effective and well defined.
- Effective means some formal method must exist for carrying out that operation and also getting an answer.

Developing an Algorithm:

- The errors they discover will usually lead to insertions, deletions or modifications in the existing algorithm.

Characteristics of Algorithmic Language:

- **Unambiguous** − Algorithm should be clear and unambiguous. Each of its steps (Or phases), and their inputs/outputs should be clear and must lead to only one meaning.
- **Input** – An algorithm should have 0 or more well-defined inputs.
- **Output** – An algorithm should have 1 or more well-defined outputs, and should match the desired output.
- **Finiteness** − Algorithms must terminate after a finite number of steps.
- **Feasibility** − should be feasible with the available resources.
- **Independent** – An algorithm should have step-by-step directions, which should be independent of any programming code.

Ex:

To find the sum of the first K integers

```
Begin
        read k
        if k<=0
```

```
            then
                    Write "illegal values for k"
                    End
            else
                    set i to 1
                    set sum to 0
                    repeat k times
                            add i to sum
                            increment i by 1
                    end of the repeat loop
                    write "the sum of first ", k, "integer is ", sum
    End.
```

<u>DESIGN OF ALGORITHMS</u>

- An algorithm is set of simple instructions to be followed to solve a problem.
- Each of the algorithms will involve a particular data structure.
- A data structure including the type of data and the frequency with which various operations on data are applied.

1. How to design an algorithm?

- Algorithm design is a creative activity. Some common approaches for designing algorithms are

✓ **Greedy algorithm:**
  - The greedy algorithm works in steps.
  - In each step this algorithm selects the best available option until all options finish
    Ex: Shortest path algorithm

✓ **Divide and Conquer:**
  - When the method is applied it often leads to a large improvement in time complexity.
  - A big problem is divided into same type of smaller problems
    Ex: Quick sort

✓ **Non-recursive Algorithm:**
  - A set of instructions that perform a logical operation can be grouped together as a function.
  - If a function, which in turn invokes the calling function, then the technique is called as indirect recursion.

✓ **Randomized algorithm:**
  - We can use the feature of random number instead of a fixed number. It gives different results with different input data.

✓ **Back track algorithms:**
  - An algorithm technique to find solutions by trying one of several choices if the choice proves incorrect, computation back tracks or restarts at the point of choice and tries another choice.
  - A back tracking algorithm is to write a function or procedure which traverse the solution space.
    Ex: Game tree.

✓ **Modular programming approach:**
  - The importance of splitting a problem into a series of self contained modules that becomes important.
  - A module should not exceed about 100 or 50 lines and should preferable be short enough to be on a single page.
  - The analysis of an algorithm provides information that gives us a general idea of how long an algorithm will take for solving a problem.

<u>IMPLEMENTATION OF ALGORITHM</u>

- After spelling out completely and precisely the requirements for each tasks and sub-tasks, it is time to code them into our programming language once the specification at the top levels are complete and precise, we should code the subprograms at these levels and test them appropriately.

VERIFICATION OF ALGORITHM

- ➢ Verification of algorithm would consist of determining the quality of the output received.
- ➢ It is a process of measuring the performance of the program with any laid down standards. Algorithm verification should precede coding of the programmer.

EFFICIENCY ANALYSIS OF ALGORITHM

- ➢ An algorithm analysis provides information that gives us an idea of how long an algorithm will take for solving a problem.
- ➢ A number of algorithms might be able to solve a problem successfully yet the analysis of algorithm gives us the scientific reason to determine which algorithm should be chosen to solve the problem most efficiently.

(i) Space complexity:

- ➢ Space complexity of a program is the amount of main memory in a computer.
- ➢ It needs to run for completion.
- ➢ The space needed by a program is the sum of the following components:
  - ✓ Fixed part that includes space for the code, space for simple variable and fixed size component variables as well as, space for constant etc.
  - ✓ Variable part that consist of the space needed by component variable whose size is dependent on the particular problem instance being solved and the stack space used by recursive procedure.

(ii) Time complexity:

- ➢ Time complexity of a program is the amount of computer time it needed to run a program completion. The time complexity may give time for
  - ✓ **Best case**
  - ✓ **Worst case**
  - ✓ **Average case**

Best case:

- ➢ The algorithm searches the elements in first time itself.
- ➢ Best case takes shortest time it executes as it causes the algorithms to do the least amount of work.

Worst case:

- ➢ In worst case find the element of time at the end of the total time taken when searching elements fails.

Average case:

- ➢ Analyzing the average case behaviors of an algorithm is more complex than best case and worst case.
- ➢ As the volume of data increases, average case of algorithm behaves like worst case algorithm.

(iii) Frequency count:

- ➢ To make an analysis machine independent it can be assumed that every statement will take the same constant amount of time for it execution.
- ➢ The time complexity can also expressed to represent by the frequency counts. One such notation for frequency count is order notation (‘O’ notation)

‘O’ notation: **‘O’ notation** is used to **measure the performance of any algorithm**. Performance of an algorithm **depends upon the volume of input data.**

- O (1) to mean a computing time that us a constant.
- O (n) is called linear, when all the elements of the linear list will be traversed.
- O (log n) is when we divide linear list to half each time and traverse the middle element.
- O (n log n) is when we divide list half each time and traverse that half portion

SIMPLE ALGORITHMS

(i) Exchanging the value of two variables:

- Exchanging the value of two variables means interchanging their values.
- Two variables x and y.
- To swap or interchange the value of x and y.
- Original values of x and y are

Algorithm:

- Begin
- Get the values of variables x and y
- Assign the value of x to t.
- Assign the value of y to x. So x has original value of y now in place of the original value of x.
- Assign the value of t to y.
- Show the values of x any y
- Stop

ii) Reversing Digits of an Integer Number:

Reversing digits basically means changing the digits order backwards.

Input : 1 9 8 0

Output: 0 8 9 1

Algorithm:

- ✓ Begin
- ✓ Get positive integer number to be reversed.
- ✓ While the integer number being reversed is greater than 10 repeatedly do:
  - ✓ Extract the right most digit of the number to be reversed by remainder function ie) function mod( ).
  - ✓ Construct the reversed integer by writing the extracted digit to right hand side of the current reversed number
  - ✓ Write the integer to the RHS of the reversed number
- ✓ stop

iii) Factorial of a given number:

The product of all positive integers from 1 to n is called the 'factorial n' or 'n' factorial. It is denoted by n!

5! = 1*2*3*4*5 =120

n!= 1*2*3*4*5 ….. (n-2) *(n-1) *n

Where n is a + ve integer

Algorithm:

- ✓ Begin
- ✓ Get the number of which the factorial is to be calculated ie) n.
- ✓ Assign the value of i=1 and factorial =1
- ✓ Calculate factorial n=factorial (n-1) *i
- ✓ Increment the value of I by 1 ie) i=i+1
- ✓ Repeat steps 4 & 5 till step 4 has been executed with the value of i=n
- ✓ Write the value of factorial
- ✓ Stop.

v) Organize numbers in order (sorting):

Sorting of numbers means to arrange a given set of numbers in ascending or in descending order.

Input:	50	60	10	8	1	6	12
Output:	1	6	8	10	12	50	60

Algorithm:

1. Begin
2. Read numbers and store them in an array of n elements i.e.) size of array a is n with elements as a[1],a[2],a[3],……a[n].
3. Find the smallest element within unsorted array.
4. Exchange i.e.) swap smallest value of the array element and store it in 1st element of the unsorted part of array.
5. Repeat steps 3 and 4 until all the elements in the array are arranged in the ascending order

6. Stop.

RECURSION

- A recursion function is **a function that calls itself** to perform a specific operation.
- The process of a function calling itself is known as recursion.
- Indirect recursion occurs when one function calls another function that then calls the first function.

Ex: 1

Find the factorial values for the values 1 to 5

```
#include (stdio.h>
int factorial( int value)
{
if (value = = 1)
        return(1);
else
        return (value * factorial (value -1));
}
void main(void)
{
 Int I;
 For (i=1; i<=5; i++)
        Printf( " The factorial of %d is %d \n", i, factorial (i));
}
```

Ex: 2

Using recursion to complete the sum of integers from 1 to n

```
Int sum(n)
Int n;
{
If( n<=1)
    return n;
Else
    return (n+sum(n-1));
}
```

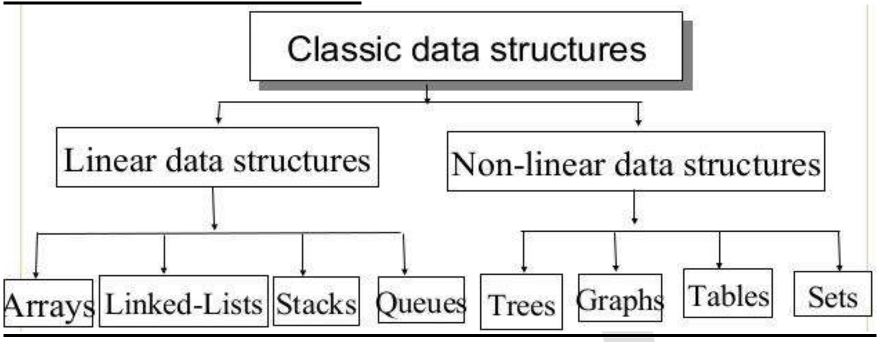
------------------------------------ END------------------------------------

UNIT-II

DATA STRUCTURES

- Data Structure is a way of collecting and organising data in such a way that we can perform operations on these data in an effective way.
- Data Structures are the programmatic way of storing data so that data can be used efficiently.

Overview of Data Structure:

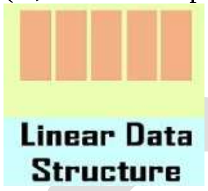


Categories of data structures:

i) Linear data structure
ii) Non linear data structure

Linear data structure:

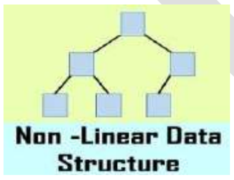
- In linear data structures processing of data items is possible in linear function (ie, data can be processed One by one).



- **Ex:** Array, Linked list, stack & queue.

Non-linear data structure:

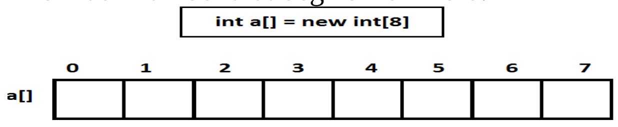
- The elements do not form a sequence.
- Insertion and deletion is not possible in a linear function.



- **Ex: Trees and graphs**

ARRAYS:

- An array is a collection of two or more adjacent memory locations.
- The memory location containing same type of data.
- These memory locations are called array elements which are given a particular symbolic name.
- Array elements are accessed using their index number.
- The index number that begins from zero.



Advantages:

- Insertion and deletion can be done randomly.

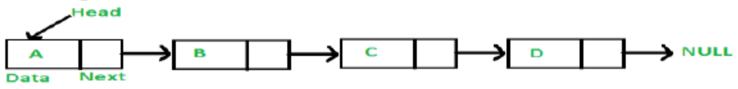
- At any location, it the start, somewhere at the middle or at the end.

Disadvantages:

- Insertion and deletion operation requires movement of large amount of data.

LINKED LIST

- A linked list is the chain of data items.
- Data items are connected by pointers.
- Each item contains a pointer.
- The pointer to the address of next item.



Types of linked lists are:

- ✓ Singly linked list
- ✓ Doubly linked list
- ✓ Circular linked list

✓ Singly linked list each node contains data and a single link which attaches it to the next node in the list.

✓ Doubly linked list each node contains data and two links one to the previous node and one to the next node.

✓ In circular linked list the last node, instead of pointing to NULL, points to the starting node.

Advantages:

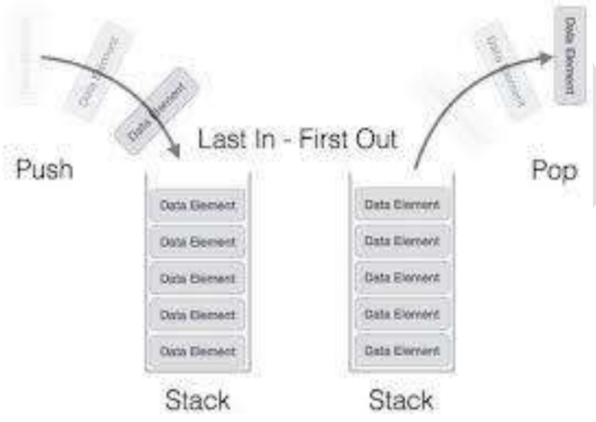
- Quick insertion
- Quick deletion
- Insertion and deletion operations in a linked list are much easier

Disadvantages:

- Slow search

STACK:

- A stack is a linear data structure.
- It is similar to an array.
- Elements are added or removed only from one end of the list.
- It is also known as **Last-In-First-Out (LIFO).**
- A Stack is implemented as an array or a linked list.



Advantages:

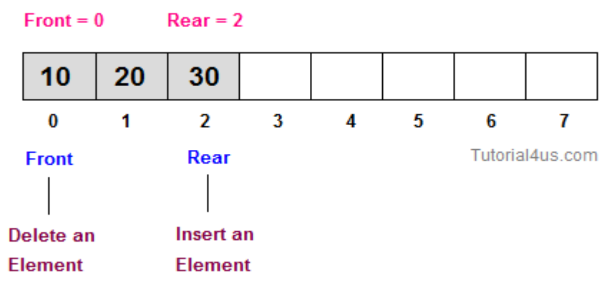
Provide last in first out access.

Disadvantages:

Slow access to other items.

4. QUEUES:

- A queue is a linear data structure.
- Entries can be added or removed at either end but not in the middle.
- Queue is also known as **First-in- First –Out (FIFO)**
- Queue is implemented as arrays or linked lists.



Advantages:

Provides first-in-first-out access

Disadvantages:

Slow access to other items.

Non-linear data structure:

TREE:

- ➢ Tree data structure is non-linear data structure.
- ➢ A tree is a data structure that represents hierarchical relationships between individual data items.

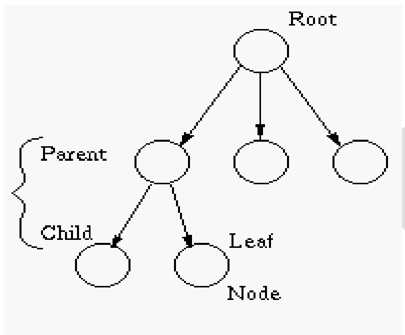


Figure: Tree data structure

Fig: A tree structure

- ➢ A tree consists of a **collection of nodes** which are connected by directed arcs.
- ➢ A tree contains a unique **first element known as the root**, which is at the top of the tree structure.
- ➢ A node with **no children is called a leaf node**.
- ➢ Tree is a very flexible and powerful data structure that can be used for a wide variety of applications.

Advantages:

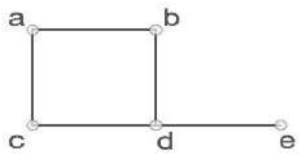
- ✓ Quick search, insertion and deletion if tree remains balance.
- ✓ Easy to understand from the graphical view.

Disadvantages:

- ✓ Needs unnecessary traversal in order to reach a particular node.

GRAPHS:

- ➢ A graph is a pictorial representation.
- ➢ Set of objects where some pairs of objects are connected by links.
- ➢ The interconnected objects are represented by points termed as vertices.
- ➢ The links that connect the vertices are called edges.
- ➢ Formally, a graph is a pair of sets (V, E).
- ➢ Where V is the set of vertices.
- ➢ E is the set of edges, connecting the pairs of vertices.
- ➢ Take a look at the following graph



Definition:

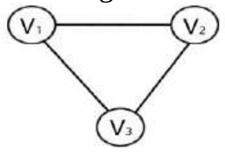
- ➢ A graph is defined as a set of nodes or vertices.
- ➢ Set of lines or edges or arcs that connect the two vertices.

Types:

1. Undirected graphs
2. Directed graphs.

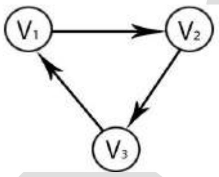
<u>Undirected Graph:</u>

- ➢ Edges are unordered pairs of vertices.



<u>Directed Graph:</u>

- ➢ A directed graph or digraph, the edges between nodes directionally oriented.
- ➢ There are directed edges from A to B, B to C, and C to D & D to E.
- ➢ The directed edges are also called an arc.



Advantages:

- ✓ Graphs are data structures which have wide ranging applications in real life, like analysis of electrical circuits, finding shortest routes, statistical analysis etc.
- ✓ A directed graph is a natural way of describing, representing and analyzing complex projects which consists of may interrelated activities.

<u>ARRAYS</u>

Define array:

- ✓ It is a linear data structure.
- ✓ An array is a finite, ordered and collection of homogeneous data elements.
- ✓ It contains only limited number of elements.
- ✓ All the elements are stored one by one in contiguous locations of computer memory in a linear ordered.
- ✓ All the elements of an array are of the same data type.

Example

Int A [4]; Dim A [4];

ARRAYS TERMINOLOGY:

<u>Size:</u>

- ➢ Number of elements in an array is called the size of the array.
- ➢ It is also called as **length or dimension.**

<u>Type:</u>

- ➢ Type of an away represents the kind of data type.

Ex: int, string

<u>Base:</u>

- ➢ Base of an array is the address of memory location where the first element in the array is located.

<u>Index:</u>

- ➢ All the elements in an array can be referenced by subscript like A_i or A[i], this subscript is known as index.
- ➢ Index is always as integer value.
- ➢ Every element is identified by a subscripted or indexed variable.

Range of index:

- Indices of array elements may charge from a lower bound (L) to an upper bound (U), which are called the boundaries of an array.

Ex:

Int A [100]: The range of index is from 0 to 99.
A: Array [-5…19] of integer: The points of the rage is -5, -4, -3…18, 19.
Here L is the lower bound.
Formula is Index (a_i) = L+i-1
If the range of index varies from L…..U then the size of the away can be calculated as Size (A) = U-L+1

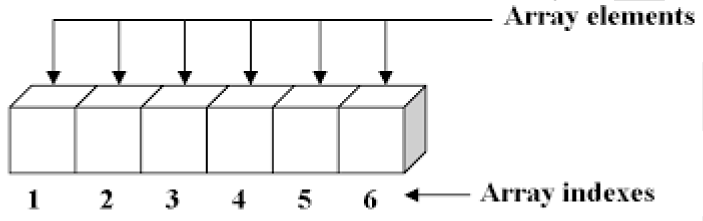
Word:

- It denotes the size of an element. In memory location computer can store an element of word size w.
- This word size varies from machine to machine such as 1 byte to 8 bytes.

ONE DIMENSIONAL ARRAY

Definition:

- Only one subscript /index is required to reference all the elements in an array then the array will be termed as one dimensional array.



One-dimensional array with six elements

Example:

- Declare a variable without using array,
  - int mark1;
  - int mark2;
- Declare a variable using array,
  - int mark [3];

mark1	mark2
X[0]	X[1]

MEMORY ALLOCATION FOR AN ARRAY

- The memory location where the first element can be stored is M.
- If each element requires one word then the location for any element say A[i] in the array can be obtained as

  Address (A[i] = M+ (i-1)

- An array can be written as A[L….U] , where L & U denote the lower and upper bounds for index.
- If it is stored starting from memory locations and for each element it requires w number of words.
- Then the address for a[i] will be

  Address (A[i]) = M+ (i-L) * W

- The formula is known as indexing formula which is used to map the logical presentation of an array to physical presentation.

OPERATIONS ON ARRAY

- Various operations can be performed on an array are

1. Traversing	3. Sorting	5. Searching	6. Insertion
2. Deletion	4. Merging		

1. Traversing:

- This operation is used visiting all elements in an array.

Algorithms	: Traverse-array ( )
Input	: An array A with elements
Output	: According to process ( )

Steps:

```
 i=L                                  // start from first location L
While i<=U do              // U upper bound
          Process (A[i])
          i=i+1                        // move to next position
End while
Stop
```

Here process ( ) is an procedure which when called for an element can perform an action

2. Sorting:

- This operation if performed on an array will sort it in a specified order.
- The algorithm is used to store the elements of an integer array in ascending order.

Input	: An array with integer data
Output	: An array with sorted element in an order according to ORDER ( )

Steps:

```
i= U
While i>= L do
          j=L        // start comparing from first
          While j< i do
                    If ORDER (A[j, A[j+1]) = FALSE     // if A[j] and A[j+1] are not in order
                    Swap (A[j], A [j+1])                        // Interchange the elements
                    End if
          j=j+1                                              // Go to next statement
          End while
          i=i-1
End while
Stop
```

- Here order ( ) is a procedure to test whether two elements are in order and SWAP ( ) is a procedure to interchange the elements between two consecutive locations.

3. Searching:

- This operation is applied to search an element of interest in an array.

Algorithm	: Search array (key)
Input	: Key is the element to be searched
Output	: Index of key in A or a message on failure

Steps:

```
I=L, found=0, location=0 // found=0 indicates search is not finished and unsuccessful
While (i<=U) and (found =0) do
          If compare (A[i], key) = true then
                    Found=1
                    Location =i
          Else
                    i=i+1
                    End if
End while
```

```
If found=0 then
        Print "search is unsuccessful, key is not in the array "
Else
        Print "search is successful: key is in the array at location ", location
End if
Return (location)
Stop
```

4. Insertion:

- This operation is used to insert an element into an array provided that the array is not full.

Algorithm: insert (key, location)
Input : key is the item; location is the index of the element where it is to be stored.
Output: array enriched with key

Steps:

```
    If A [U] # NULL then
            Print "Array is full, no insertion possible"
            Exit
    Else
            While i> location do
                    A [i+1] =A[i]
                    i = i-1
            End while
            A[location] =key
            U=U+1
End if
Stop
```

5. Deletion:

This operation is used to delete a particular element from an array. The element will be deleted by overwriting it with its subsequent element and this subsequent element then is to be deleted.

Algorithm: delete (key)
Input: key is the element to be deleted.
Output: slimed array without key
Steps:

```
i = search array (a, key)
if i = 0 then
        Print "key is not found, no deletion"
        Exit
Else
    While i< U do
    A[i] = A [i+1]
    i = i+1
    End while
End if
A [U] = NULL
U=U-1
Stop
```

6. Merging:

- Merging is an important operation when we need to compact the elements from two different arrays into a single array.

Algorithm: merge (A1, A2: A)
Input: Two arrays A1 [L1…U1], A2 [l2…U2]
Output: Result array A [L…U] , where L=L1 and U=U1+(U2-L2+1) when A1 is append after A2
Steps:

```
i1=L1, i2=L2;                       // initialization of variables
L=L1, U=U1+U2 –L2 +1                // initialization of lower and upper bounds of an array
i=L
```

```
Allocate memory for a[L…U]
While i1<U do                          // to copy array A1 into the first part of A
            A[i] = A1 [i1]
            i=i+1, i1=i1+1
End while
While i2<=U2 do                        // to copy the array A2 into last part of A
            A[i] = A2 [i2]
            i=i+1, i2=i2+1
End while
Stop
```

Application of Array:

- Every programming language include this data type as a built in data type.

Ex:

To store records of all students in a class, the record structure is given by students.

ROLLNO	MARK1	MARK2	MARK3	TOTAL	GRADE
(Alphanumeric)	(Numeric)	(Numeric)	(Numeric)	(Numeric)	(Character)

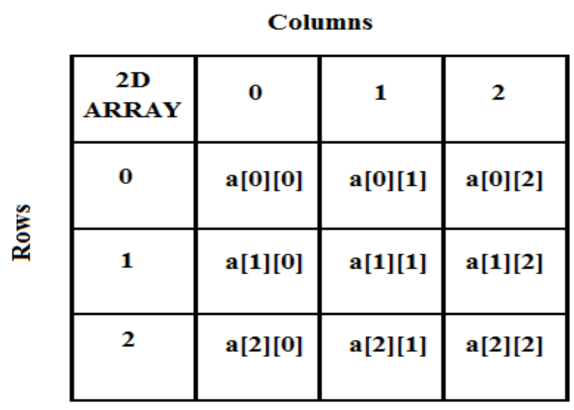
- If sequential storage of records is not an objection, then we can store the records by maintaining 6 arrays whose size is specified by the total number of students in the class.

ROLLNO MARK1 MARK2 MARK3 TOTAL GRADE

MULTIDIMENSIONAL ARRAYS

Two dimensional Arrays:

- Two dimensional arrays are the collection of homogeneous elements.
- It also called matrix.
- The elements are ordered in a number of rows and columns.



Columns

Rows

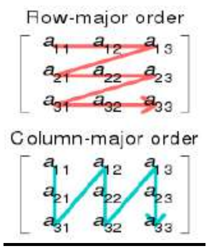
2D ARRAY	0	1	2
0	a[0][0]	a[0][1]	a[0][2]
1	a[1][0]	a[1][1]	a[1][2]
2	a[2][0]	a[2][1]	a[2][2]

Ex:

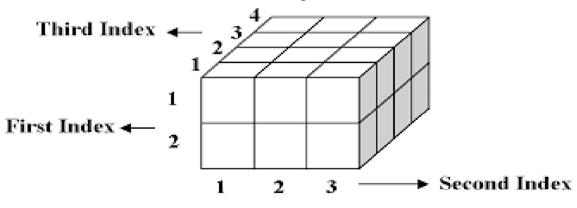
- An (m x n) matrix where m denotes the number of rows and n denotes the number of columns.
- The subscript of any arbitrary elements say (a_{ij}) represents the i^{th} row and j^{th} column.

Memory Representation of a Matrix:

- Matrix representation is a method used by a computer language.
- Used to store matrices of more than one dimension in memory.



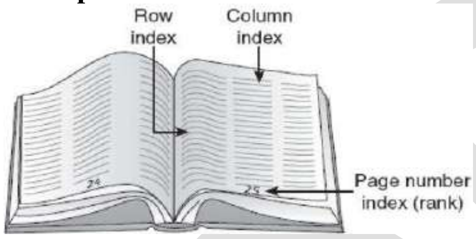
Three dimensional arrays:



Three-dimensional array with twenty four elements

- ➢ Three dimensional arrays have three indexes.
- ➢ First index refers to dimension.
- ➢ Second index refers to row.
- ➢ Third index refers to column.

Example:



Pointer Array

- ➢ Address of memory variable or array is known as pointer and an array containing pointers as its elements is known as pointer array.
- ➢ An array of pointers is an indexed set of variables in which the variables are pointers (a reference to a location in memory).

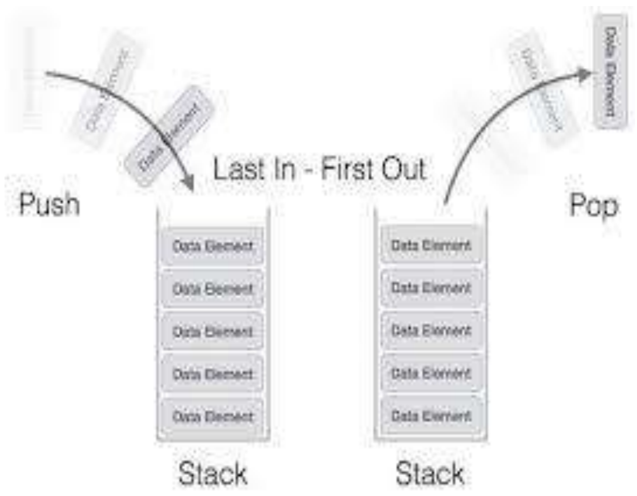
Ex:

To store the marks of all students of computer science & engineering dept for a year, there are six classes, say cs1,cs2…..cs6

<u>**STACKS**</u>

Introduction

- ➢ Stack is a linear data structure.
- ➢ Very much useful in various applications of computer science.
- ➢ Stack is a collection of elements in a Last-In-First-out fashion.
- ➢ Stack is also called as LIFO.



Ex:

- ✓ Shunting of trains in a rail yard.
- ✓ Shipment in a cargo.
- ✓ Order supply in a restaurant.
- ✓ Arrangements of books.

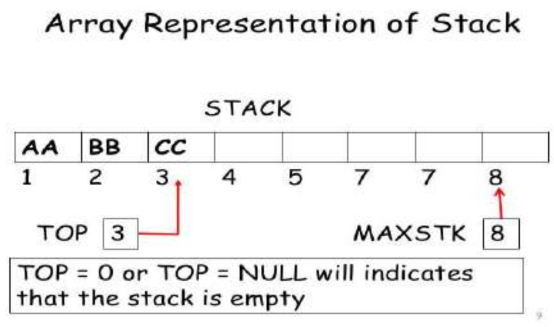
Definition

- ➢ A stack is an ordered collection of homogeneous data element.
- ➢ Where the insertion & deletion operation take place at one end only.
- ➢ Stack is also a linear data structure.
- ➢ The insertion and deletion operations in case of stack are specially termed as PUSH and POP.
- ➢ Where the operations are performed is known as TOP of the stack.
- ➢ An element in a stack is termed as ITEM.
- ➢ The maximum number of elements that a stack can accommodate is termed as SIZE.

REPRESENTATION OF STACK

- ➢ A stack representation in two ways
    1. Using one dimensional array.
    2. Single linked list

<u>Array representation of stacks:</u>


Array Representation of Stack
STACK
AA
BB
CC
1
2
3
4
5
7
7
8
TOP
3
MAXSTK
8
TOP = 0 or TOP = NULL will indicates that the stack is empty

- ➢ Allocate a memory block of sufficient size to accommodate the full capacity of the stack.
- ➢ Then from the first location of the memory block, items of the stack can be stored in sequence mode.
- ➢ Item denotes the i^{th} item in the stack, l and u denote the index range of array in use, and these values are 1 and size.
- ➢ Top is a pointer to point the position of array up to which it is filled with the items of stack.
- ➢ There are two statuses

EMPTY: TOP < l

FULL : TOP >= u+l-1

Linked list representation of stacks:

- The size of the stack may vary during program execution.
- Single linked list structure is sufficient to represent any stack.

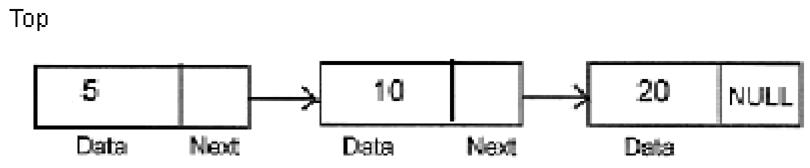


Fig.: 5.4: Linked list representation of a stack

OPERATIONS ON STACKS

- Basic operation required to manipulate stacks:
  - (i) **PUSH** – To insert n item into the stack
  - (ii) **POP** – To remove an item from a stack

(i) PUSH algorithm:

Algorithm PUSH_A (ITEM)

Steps:

```
If   TOP > = SIZE then
          Print "stack is full"
Else
          TOP= TOP+1;
          A [TOP] = ITEM
End if
Stop
```

- Array index varies from 1 to SIZE and TOP points the location of the current top-most item in the stack.

(ii) POP Algorithm:

Algorithm POP_A ( )

Steps:

```
If   TOP < 1 then
          Print "stack is empty"
Else
          ITEM=A [TOP]
          TOP=TOP-1
End if
Stop
```

- Next operation is to test the various states of a stack like whether it is full or empty, how many items are right now in it and read the current element at the top without removing it.

<u>Same operations can be defined for a Stack using Linked List:</u>

<u>(i) PUSH Algorithm:</u>

Algorithm: PUSH_L (ITEM)

Steps:

```
new= GETNODE(NODE)
new.DATA=NEW
new.LINK=TOP
TOP=new
STACK_HEAD.LINK=TOP
Stop
```

<u>(ii) POP Algorithm:</u>

Algorithm: POP_L ( )

Steps:

```
    If TOP = NULL
        print " stack is empty"
        Exit
    Else
        ptr =TOP.LINK
        ITEM= TOP.DATA
        STACK_HEAD.LINK=ptr
        TOP=ptr
    End if
    Stop
```

APPLICATIONS OF STACK

- ✓A classical application deals with evaluation of arithmetic expression; compiler uses a stack to translate input arithmetic expression into their corresponding object code.
- ✓Some machines are also known which use built-in-stack hardware called **"stack machine**".
- ✓Another important application of stack is during the execution of recursive program, some programming languages use stacks to run recursive programs.
- ✓**There are two scope rules known Static** scope rule and dynamic scope rule.
- ✓Implementation of such scope rule is possible using stack known as run time stack.

1. Evaluation of Arithmetic Expression:

- ➢ An arithmetic expression consists of operands and operators.
- ➢ Operands are variables or constant and operators are of various types like arithmetic unary and binary operators (+, -, Unary, *, /, ^, % ….) and relational operators (<, <=, >, >=, <>) and Boolean operators (AND, OR, NOT, XOR).
- ➢ A simple arithmetic expression is

    A+B
- ➢ The problem to evaluate this expression is the order of evaluation.
- ➢ There are two ways to fix it

Operators	**Precedence**	**Associativity**
(-) Unary, + (Unary) , NOT	6	-
^ (Exponentiation)	6	Right to Left
*, /	5	Left to Right
+, -	4	Left to Right
<, <=, + , >= , <>	3	Left to Right
AND	2	Left to Right
OR, XOR	1	Left to Right

Notations for arithmetic expressions:

- ➢ There are 3 notations to represent an arithmetic expression infix, postfix and prefix.
- ➢ The conventional way of writing an expression is called infix.

Ex:

A +B, C-D, E * F, G/H etc

Infix notation:

- ➢ The notation is

    < Operand> <operator> <operand>
- ➢ This is called **infix** because the operators come in between the operands.

    A +B

Prefix notation:

- ➢ The prefix notation on the other hand user the conventions.

    <Operator> < Operand> <operand>
- ➢ The operator comes before the operands.

    +AB
- ➢ This notation was introduced by polish mathematician Jan Lukasiewicz and hence also termed as polish notation.

Postfix notation:

- The last notation is called the postfix (or) suffix notation where operators is suffixed by operands.
  < Operand> <operand><operator>
  AB+
- This notation is just reverse to the polish notation; hence it is also alternatively termed as reverse polish notation.
- An expression given in infix notation can be easily converted into its equivalent prefix or postfix notation.

Rules for convert and infix expression into a postfix form:

- Assume the fully parenthesized version of the infix expression.
- Move all the operators so that they replace their corresponding right part of parentheses.
- Remove all parentheses.

Ex:
Arrows point from operators to its corresponding right parenthesis

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

Operators are moved to their respective right parenthesis.

Output:

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

Three notations for the given arithmetic expression:
Infix : (A * (B + (C / D)))
Prefix**:** (* (+ (/ C D) B) A)
Postfix: (A (B (C D /) +) *)

- ✓ In both the prefix and postfix equivalents of an infix expression, the variables are in the same relative positions.
- ✓ The expressions in prefix or postfix form are completely parenthesis free.
- ✓ The operators are rearranged according to the rules of precedence of operators.
- ✓ Out of three notations, postfix notation has certain advantages over other notations from the computational point of view.
- ✓ The main advantage is its evaluation.
- ✓ During the evaluation of an expression in postfix notation it no more requires to scan the expression from left to right several times, but exactly once.

Conversion of an infix expression to postfix expression:

- A simple arithmetic expression containing +, -, *, / and ^ operators only.
- The expression may be parenthesized or unparenthesized.
- First we have to append the symbol ')' as delimiter at the end of a given infix expression and initialize the stack with '('.

Symbol	**In stack priority**	**In coming priority values**
+ -	2	1
* /	4	3
^	5	6
Operand	8	7
(	0	9
)	-	0

- There are two priority values
  - ✓ **In stack priority**
  - ✓ **In coming priority values**
- A symbol will be pushed on to the sack if its incoming priority value is greater than the in-stack priority value of the top-most element.

- Similarly a symbol will be popped from the stack if its in-stack priority value is greater that the in-stock priority value of the top-most element.
- Similarly a symbol will be popped from the stack if its in-stack priority value is greater or equal to the incoming priority value of the in-coming element.

Function:
READ-SYMBOL ( ) - From a given infix expression this will read the next- symbol.
ISP(X) – Returns the in-stack priority value for a symbol x
ICP(X) – Returns the incoming priority value for a symbol x
OUTPUT(X)- Append the symbol x into the resultant expression.

Algorithm: INFIX-TO-POSTFIX (E)

```
Steps:
    TOP=0, PUSH (‘(‘)
    While (TOP > 0) do
            item = E.READ-SYMBOL( )
            x=POP ( )
            case: item=operand
                    PUSH(X)
                    OUTPUT (ITEM)
            case: item= ‘)’
                      While x # ‘(‘do
                              OUTPUT(X)
                              X=POP ( )
                      End while
            case: ISP(X) > = ICP(ITEM)
                      While (ISP(X) >= ICP (ITEM)) DO
                              OUTPUT(X)
                              X=POP ()
                      End while
                      PUSH(X)
                      PUSH (ITEM)
            case: ISP(X) < ICP(ITEM)
                      PUSH(X)
                      PUSH (ITEM)
            Otherwise: print “invalid expression”
End while
Stop
```

Ex:
Infix form (A+B) ^ C-(D*E)/F)
Symbol reading 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Read symbol	stack	output
Initial	(	
1	((	
2	((	A
3	((+	A
4	((+	AB
5	(	AB+

6	(^	AB+
7	(^	AB + C
8	(-	AB + C ^
9	(- (	AB + C ^
10	(- (	AB + C ^ D
11	(- (*	AB + C ^ D
12	(- (*	AB + C ^ DE
13	(-	AB + C ^ DE *
14	(- /	AB + C ^ DE *
15	(- /	AB + C ^ DE * F
16		AB + C ^ DE * F / -

Output: A B + C ^ DE * F / - (postfix form)

EVALUATION OF POSTFIX EXPRESSION:

- Algorithm EVAL_POSTFIX is to evaluate an arithmetic expression in postfix notation using a stack.

Steps:

```
Append special delimiter ‘# at the end of the expression
item = E.READ_SYMBOL( ) // Read the first push into the stack
While (item = “#”) do
    If (item = operand) then          // Operand is first push into the stack
    PUSH (item)
    Else
    op = item                         // The item is an operator
    y = POP ( )                       // The right most operand of the current operator
    x = POP ( )                       // The left most operand of the current operator
    t = x op y                        // Perform the operation with operator ‘op’ and
operand x,y
    PUSH (t)                          // Push the result into stack
    End If
    Item = E.READ_SYMBOL ( )          // Read the next item from E
End While
Value = POP ( )                                  // Get the value of the expression
Return (value)
Stop
```

Example:

Infix: A + (B * C) / D
Postfix: A B C * D / +
Input: A B C * D / + # with A = 2, B = 3, C = 4, and D = 6

ALGORITHM FOR CONVERTING A POSTFIX EXPRESSION:

Algorithm POSTFIX_TO_CODE (E)
Input: An arithmetic expression E in postfix notation
Output: Assembly code.
Steps:

```
Append delimiter ‘#’ at the end of the expression
item=E.READ_SYMBOL( )
i=1, TOP=0
While (item ≠ ‘#’) do
        case: item =operand
        PUSH (item)
        case: item=’+’
                X=POP ( )
                Y=POP ( )
                PRODUCE_CODE (Y, X,’ADD’, Ti)
                PUSH (Ti)
    case: item=’-’
                X=POP ( )
                Y=POP ( )
                PRODUCE_CODE (Y, X,’SUB’, Ti)
                PUSH (Ti)
    case: item=’*’
                X=POP ( )
                Y=POP ( )
                PRODUCE_CODE (Y, X,’MUL’, Ti)
                PUSH (Ti)
    case: item=’/’
                X=POP ( )
                Y=POP ( )
                PRODUCE_CODE (Y, X,’DIV’, Ti)
                PUSH (Ti)
    Otherwise
                print “Error in input”
                Exit
    item =E.READ-SYMBOL( )
    i=i+1
End while
stop
```

Ex:

Infix: (A+B) *C/D
Postfix: AB+C*D/

3. IMPLEMENTATION OF RECURSION:

- A recursion is termed as recursive if the procedure is defined by itself.
- Factorial of given integer n:

n! = n * (n -1) * (n-2) * …. * 3*2 *1 or n! =n * (n-1)

Algorithm: Factorial (n)
Steps:

```
fact =1
for (i=1 to n) do
        Fact=i*fact
end for
return (fact)
stop
```

- Here step 2 defines the interactive definition for the calculation of factorial

Algorithm: Factorial (N) // using recursion
Steps:

```
if (n=0) then
      1. Fact=1
else
      1. Fact= n* factorial (n-1)
endif
return(fact)
stop
```

- Step 2 recursively defines the factorial of an integer N. This is actually a direct translation of n! = n*(n-1)! In the form of Factorial (n) = n* factorial (n-1).

4. FACTORIAL CALCULATION:

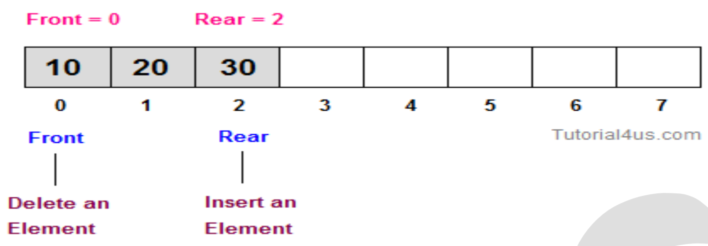
Algorithm: FACTORIAL_STACK(N)

```
val =N, top=0, addr=step 10
PUSH (val, addr)
val=val-1, addr=step 7
if (val=0) then
        1. Fact=1
        2. Go to step8
else
        PUSH (val, addr)
        go to step3
Endif
fact=val*fact
val=POP-PARAM( ), addr=pop_addr( )
Go to addr
Return (fact)
```

***************END OF THE UNIT II ***********

UNIT – III
QUEUE
What is Queue?

- Queue is a simple but very powerful data structure.
- o solve numerous computer applications.
- It is Linear Data Structure.
- Insert element at one end called Rear.
- Delete element at another end called Front.



Example:

1. Queuing in front of a counter
2. Traffic control at a turning point
3. Process synchronization in multi-user environment
4. Resource sharing in a computer centre.

Definition:

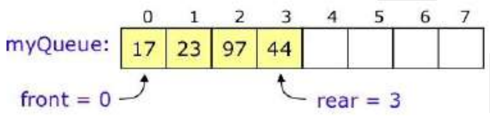
- Queue is an ordered collection of homogeneous data elements.
- It permits insertion of new element at one end.
- Deletion of an element at the other end.
- The **deletion (DEQUEUE**) of an element take place is called **front**.
- The **insertion (ENQUEUE**) of a new element can take place is called **rear**.
- An element which enter first into the queue is the first element to delete from queue so it is called **First In First out (FIFO)**

<u>Representation of Queue:</u>

- There are two ways to represent a queue in memory:
    1. Using an array
    2. Using linked list

1. Representation of queue using Array:

- A queue is first in, first out (FIFO) data structure.
- Inserting at one end (the rear).
- Deleting from the other (the front).



<u>Algorithm Enqueue (Item)</u>
Steps:

```
1. If (REAR=N) then
      1. Print "Queue is full"
      2. Exit
2. Else
      1. If( REAR=0) and (FRONT=0)
           1. FRONT=1
      2. Endif
      3. REAR=REAR+1
      4. Q[REAR]=ITEM
3. Endif
4. Stop
```

Algorithms DEQUEUE ( )

Step:

```
1. If (Front=0) then
    1. Print " Queue is empty"
    2. Exit
2. Else
    1. ITEM=Q[FRONT]
    2. IF (FRONT=REAR)
            1. REAR=0
            2. FRONT=0
    3. ELSE
            1. FRONT=FRONT+1
    4. ENDIF
3. End if
4. Stop
```

2. Representation of queue using Linked list:

- Queue is a structure containing two pointers:
    1. Front
    2. Rear
- Front: point to the head of the node.
- Rear: point to the end of the list (last node).
- Enque operates upon the rear pointer.
- Deque operates upon the front pointer.
- Empty queue is represented by NULL front & rear pointers.

Various Queue Structure:

1. Circular Queue
2. Priority Queue
3. Dequeue

Circular Queue:

- In a circular queue, all nodes are treated as circular. Last node is connected back to the first node.
- Circular queue is also called as **Ring Buffer.**
- Circular queue contains a collection of data which allows insertion of data at the end of the queue and deletion of data at the beginning of the queue.

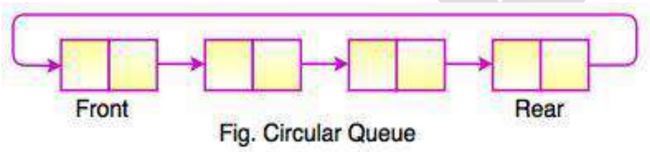


Fig. Circular Queue

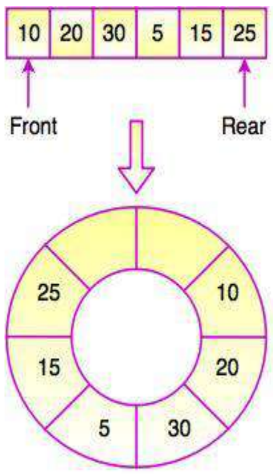


Fig. Circular Queue

Operation on Circular Queue:

1. Insertion
2. Deletion

Insertion Algorithm

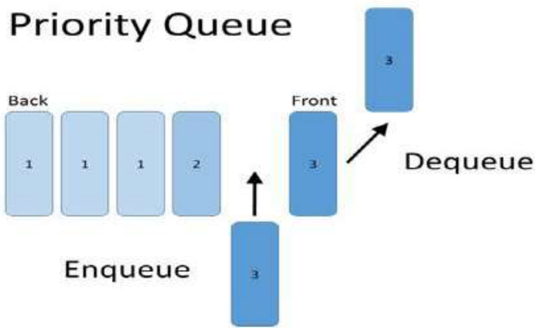
```
[Reset pointer position]
If (R= =N)
{
        R←1
}
Else
{
        R←R+1
}
[Overflow]
If (F= = R)
{
        "Queue is Empty"
}
Else
{
Q[rear]←X
}
[Is pointer]
If (F= =0)
{
F←1
}
```

Deletion Algorithm

```
[Underflow]
If (F= =0)
{
        "Queue is Empty"
}
Else
{
          X←Q[rear]
}
[Is the Front Pointer]
If (F= = R)
{
        "Queue is Empty"
}
[Reset the Front]
If (F= =N)
{
        "Queue is Empty"
}
```

Priority Queues

- Insert & delete operation based on the priority.
- If the elements in the queue are same priority, then the element deletes which is inserting first into the queue.



Types of priority Queue:

1. Ascending priority queue.
2. Descending priority queue.

1. Ascending priority queue:
   - Elements can be inserted in any order.
   - Elements deleted which is smallest element in the queue.

2. Descending priority queue:
   - Elements can be inserted in any order.
   - Elements deleted which is largest element in the queue.

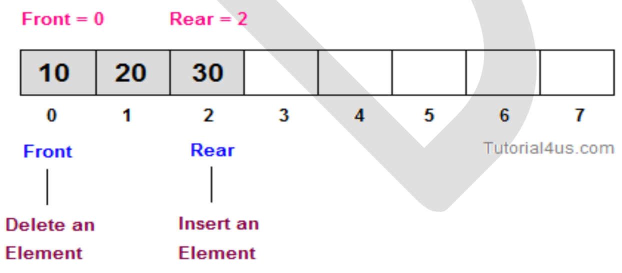
Operation of priority queue:

1. Insertion – insert item into queue.
2. Deletion – delete item from queue.
3. Display – display item in the queue.

Dequeue:

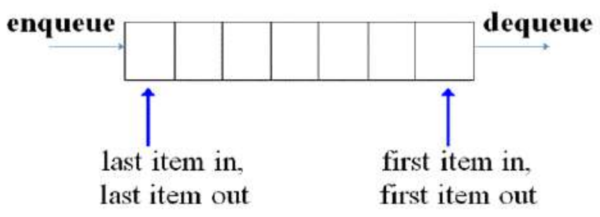
- It is used as a stack.
- In queue all operations take place at one end of the queue.

Example:



- In enqueue, add an item to the back of the queue.
- In dequeue, delete item from front of the queue.

Example:



Application Of Queues

1. Simulation:

- Simulation is a modeling of a real life problem (or) it is the method of a real life situation in the form of computer program.
- The main o**bjectives of the simulation** to study the real life situation under the control of various parameters which affects the real problem, and is a reach interest of system analysts or operation research scientists.
- Any process or situation that is to be simulated is called a system.
- A system is a collection of interconnected objects which accepts zero or more inputs and produces at least one output.
  - A system is discrete if the input/output parameters are of discrete values.
    Ex: Ticket reservation
  - A system is stochastic is based on the randomness.
    Ex: Ticket counter.

3. CPU scheduling in Multiprogramming Environment:

- A Single CPU has to serve more than one program simultaneously.
- Multiprogramming environments the CPU are categorized into three groups:
- Interrupts to be serviced.
- Verify of devices and terminals are connected with the CPU and they may interrupt at any moment to get a service from it.
- Interactive users to be services.
- These are mainly student's programs in various terminals under execution.
- Batch jobs to be services.

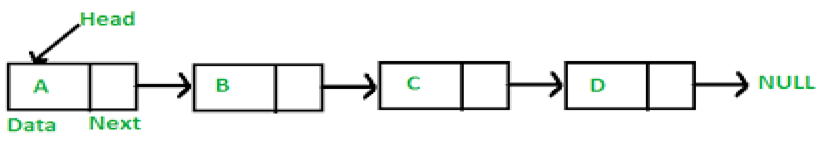
3. Round-Robin Algorithm:

- Round-Robin (RR) algorithm, is a scheduling algorithm, and is designed for time sharing systems.
- Suppose there are n processes P1, P2….Pn required to be served by the CPU.
- Different processes require different execution time.
- P1 comes first than P2 comes in general.
- RR algorithm, first decides a small unit of time, called a time quantum/slice.
- A time quantum is generally from 10 to 100 milliseconds.
- CPU starts with P1. P1 gets CPU for instant of time; afterwards CPU switches to P2 and so on.
- When CPU reaches the end of time quantum of Pn it returns to P1 and the same process will be repeated.
- Implementation of RR scheduling algorithm circular queue is the best choice, because the process when it completes its execution required to be deleted from the queue and it is not necessarily from the front of the queue rather from any position.

LINKED LISTS

What is linked list?

- Linked list are linear data structure.
- Each node in the linked list is connected with its previous node which is a pointer to the node.
- The nodes in the linked list can be added and deleted from the list.
- The node have two field,
  1. Data
  2. Address



Single Linked List

- A single linked list each node contains only one link which points the subsequent node in the list.

(i) Representation of a Linked List in Memory:

- There are 2 ways to represent a linked list in a memory.
    1. Static representation using array
    2. Dynamic representation free pool of storage.

Static representation:

- Static representation of a single linked list maintains two arrays:
- One array for data and other for links.
- Two parallel arrays of equal size are allocated which should be sufficient to store the entire linked list.

Dynamic Representation:

- The efficient way of representing a linked list is using free pool of storage.
- There is a memory bank, and a memory manager.
- During the creation of linked list, whenever a node is required the request is placed to the memory manager, memory manager will then search the memory bank for the block requested and if found grants a desired block to the caller.

Types of Linked List

- Following are the various types of linked list.
    1. Simple Linked List − Item navigation is forward only.
    2. Doubly Linked List − Items can be navigated forward and backward.
    3. Circular Linked List − Last item contains link of the first element as next and the first element has a link to the last element as previous.

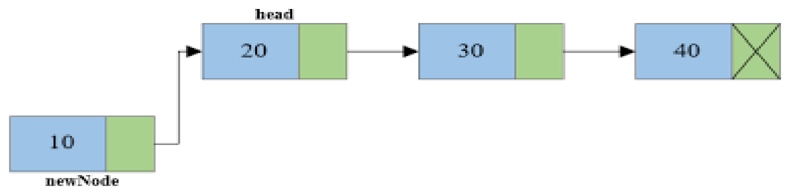
Operations on a single linked list:

1. Insertion of a node in the before head node.
2. Deletion of a node in the before head node.
3. Insertion of a node in the middle of the list.
4. Deletion of a node in the middle of the list.
5. Insertion of a node after the last node.
6. Deletion of a node after the last node

1. Insertion of a node in the before head node:

Algorithm:

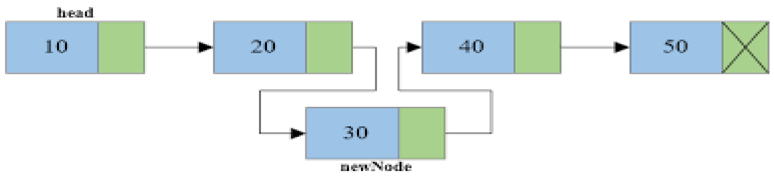
```
New_node → next = head;
Head = next_node;
```



2. Insertion of a node in the middle of the list.

Algorithm:

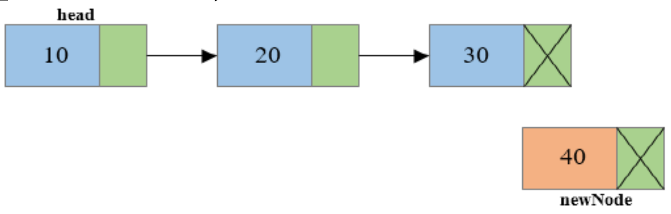
```
While (p1 = Insert position)
{
P =p→next;
}
Store_next = p→next;
P→ next = next_node;
new_node→next= store_next;
```



3. Insertion of a node after the last node.

Algorithm:

```
While (p1 = next! =null)
{
P =p→next;
}
P→ next = next_node;
new_node→next= null;
```



4. Deletion of a node in the before head node.

Algorithm

```
node* delete (node *head, char d)
{
node *p,*q;
q=head;
p=head→next;
if (q→data = ==d)
{
head = p;
delete (q);
}
else
{
While (p→data!=d)
{
P=p→next;
Q=q→next;
}}
```

5. Deletion of a node in the middle and last of the list.

Algorithm

```
If (p→next = = null)
{
q→next = null;
delete (p);
}
Else
{
q→next = p→next;
delete (p);
}}
```

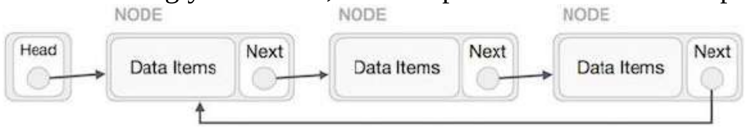
Circular Linked List

- Circular Linked List is a variation of Linked list in which the first element points to the last element and the last element points to the first element.

- Both Singly Linked List and Doubly Linked List can be made into a circular linked list.

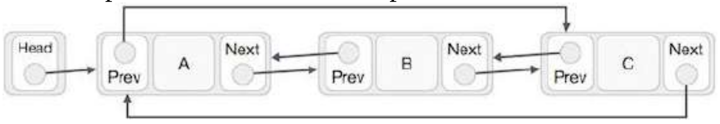
Singly Linked List as Circular

- In singly linked list, the next pointer of the last node points to the first node.



Doubly Linked List as Circular

- In doubly linked list, the next pointer of the last node points to the first node and the previous pointer of the first node points to the last node making the circular in both directions.



- As per the above illustration, following are the important points to be considered.
- The last link's next points to the first link of the list in both cases of singly as well as doubly linked list.
- The first link's previous points to the last of the list in case of doubly linked list.

Basic Operations

- Following are the important operations supported by a circular list.
  1. Insert − Inserts an element at the start of the list.
  2. Delete − Deletes an element from the start of the list.
  3. Display − Displays the list.

Insertion Operation

- Following code demonstrates the insertion operation in a circular linked list based on single linked list.

Example

```
//insert link at the first location
void insertFirst(int key, int data) {
 struct node *link = (struct node*) malloc(sizeof(struct node));
   link->key = key;
   link->data= data;
            if (isEmpty()) {
      head = link;
      head->next = head;
   } else {
         link->next = head;
          head = link;
   }
}
```

Deletion Operation

Following code demonstrates the deletion operation in a circular linked list based on single linked list.

```
//delete first item
struct node * deleteFirst() {
   //save reference to first link
   struct node *tempLink = head;

   if(head->next == head) {
      head = NULL;
      return tempLink;
   }

   //mark next to first link as first
   head = head->next;
   //return the deleted link
   return tempLink;
}
```

Display List Operation

Following code demonstrates the display list operation in a circular linked list.

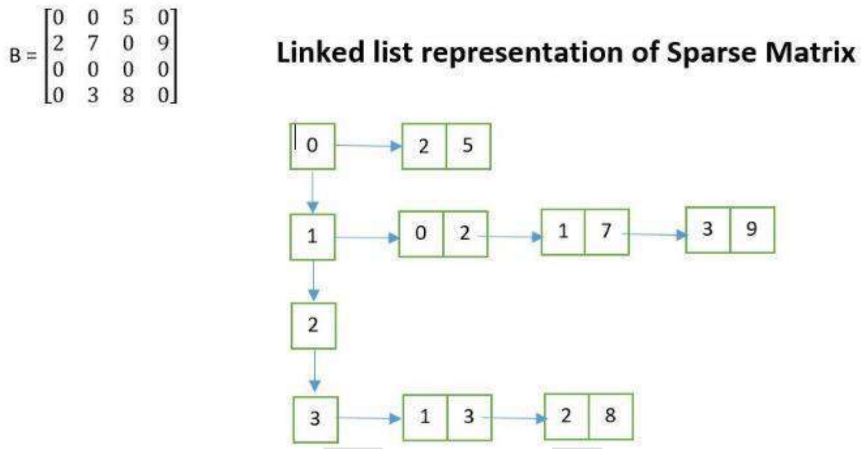
```
//display the list
void printList() {
   struct node *ptr = head;
   printf("\n[ ");

   //start from the beginning
   if(head != NULL) {
      while(ptr->next != ptr) {
         printf("(%d,%d) ",ptr->key,ptr->data);
         ptr = ptr->next;
      }
   }

   printf(" ]");
}
```

Application of Linked Lists

1. Sparse matrix manipulation:



$$B = \begin{bmatrix} 0 & 0 & 5 & 0 \\ 2 & 7 & 0 & 9 \\ 0 & 0 & 0 & 0 \\ 0 & 3 & 8 & 0 \end{bmatrix}$$

- The fields I and j store the row & column for a matrix element.
- Data field stores the matrix element at the i th row and the j th column. ie) a_{ij}.
- The ROWLINK points to the next node in the same row and COLLINK points the next node in the same column.
- All the nodes particularly in a row (column) are circular linked with each other each row (column) contains a header node.
- A sparse matrix of order m*n, we have to maintain m headers for all rows and n headers for all columns, plus one extra node use of which can be evident.

2. Polynomial

- A polynomial is a mathematical expression consisting of a sum of terms.
- Each term including a variable or variables raised to a power and multiplied by a coefficient.
- An essential characteristic of the polynomial is that each term in the polynomial expression consists of two parts:
    1. one is the coefficient
    2. other is the exponent

Example:

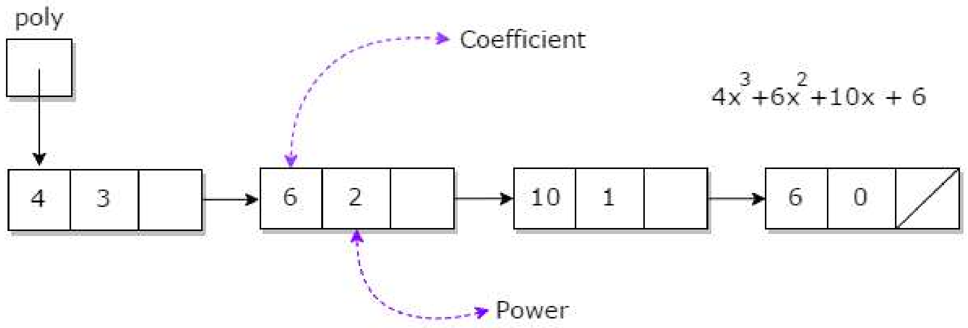
$10x^2 + 26x$

10 & 26 → Coefficients

2 & 1 → Exponential value.

<u>Points to keep in Mind while working with Polynomials:</u>

- 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



Representation of polynomial using linked list:

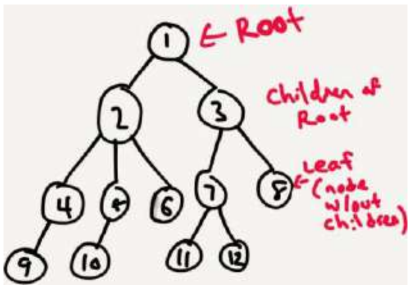
- A polynomial can be thought of as an ordered list of non zero terms. Each non zero term is a two-tuple which holds two pieces of information:
  1. The exponent part
  2. The coefficient part

UNIT-IV
TREES

Introduction about Trees:

A tree is a very flexible and powerful data structure that can be used for a wide variety of applications.

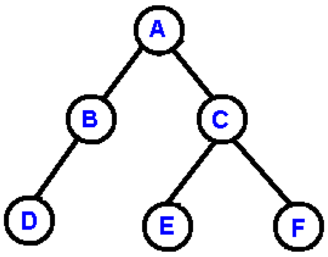
➢ Each tree node contains a name for data and one or more pointers to the other tree nodes.



Basic Terminology:

✓ **Node**: Each element presents in a binary tree is called a node of that tree.
✓ **Parent:** Parent of a node is the immediate predecessor of a node
✓ **Root:** The element represent the base node of the tree is called the root of the tree.
✓ **Child:** If the immediate predecessor of a node is the parent of the node then all immediate successors of a node are known as child.
✓ **Link:** This is a pointer to a node in a tree.
✓ **Left and Right sub trees:** Apart from the root, the other two sub sets of binary trees are binary trees. They are called the left and right sub trees of the original tree.
✓ **Leaf node**: A node that does not have any sons.
✓ **Degree:** The number of sub trees of a node.
✓ **Interior or non-interior nodes**: Nodes that have degree > 0.
✓ **Parent and child:** The roots of the sub trees of a node are called the children of that node.
✓ **Siblings:** Children's of the same parent are said to be siblings.
✓ **Ancestors**: The ancestors of anode are all the nodes along the path from the root of that node.
✓ **Level:** The level of anode is defined by initially letting the root be at level1. If a node is at level p, then its children are at level P+1.
✓ **Height or depth**: The height of a tree is defined to be the maximum level of any node in the tree.
✓ **Forest:** A forest is a set of n>=0 disjoint trees.
✓ **Path length of a node**: The number of edges needed to reach specified node form the root is called its path length.
✓ **Internal path**: The sum of path length of all the nodes in the tree.

Simple Binary tree



➢ A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties −

1. The left sub-tree of a node has a key less than or equal to its parent node's key.
2. The right sub-tree of a node has a key greater than to its parent node's key.

Definition and Concepts

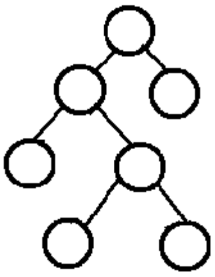
Binary trees:

➢ A binary tree consists of a **finite set of elements** that can be partitioned into three distinct subsets called the **root, the left and right sub tree.**

- ➢ If there are no elements in the binary tree it is called an empty binary tree.

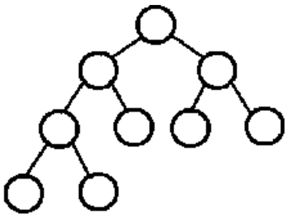
Full Binary Tree:

- ➢ All nodes have two children.
- ➢ Each sub-tree has same length of path.



Complete binary tree:

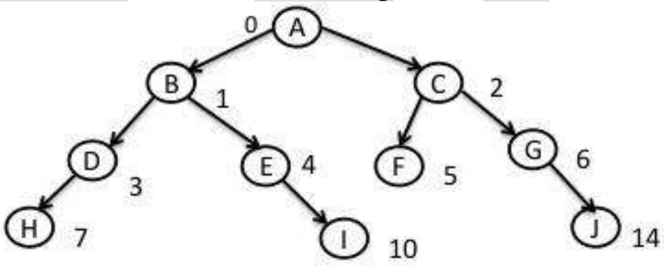
- ➢ All nodes have two children.
- ➢ Each sub-tree has different length of path.

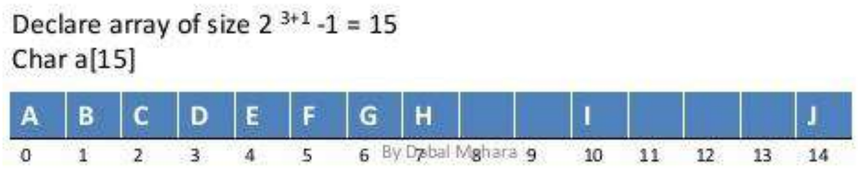


<u>**REPRESENTATION OF A BINARY TREE**</u>

Array representation

- ➢ An array is used to store the nodes of the binary tree.
- ➢ Nodes stored in the array area access sequentially.
- ➢ Root is at index 0, and then left child and right child are stored.





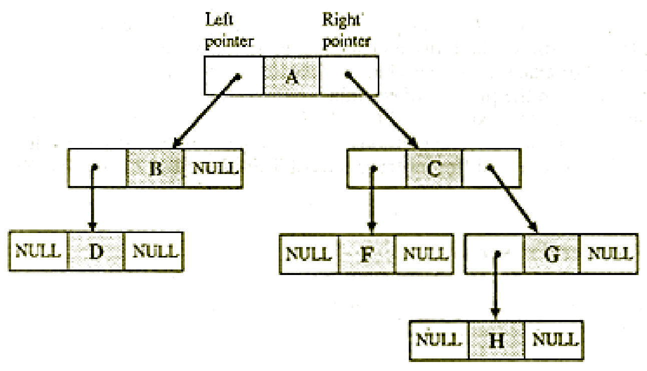
Declare array of size 2 $^{3+1}$ -1 = 15

Char a[15]

A	B	C	D	E	F	G	H			I				J
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

<u>Linked representation of binary trees:</u>

- ➢ In linked list every element is represented as nodes.
- ➢ A node consists of three fields such as,
  1. Left child
  2. Information node
  3. Right child



\

- ➢ The left child point to the left child of parent node.
- ➢ The right child point to the right child of parent node.
- ➢ The information holds the information of every node.

OPERATIONS ON A BINARY TREE

- ✓ **Insertion**: To include a node into an existing binary tree.
- ✓ **Deletion:** To delete a node from a non-empty binary tree.
- ✓ **Traversal:** To visit all the nodes in a binary tree.
- ✓ **Merge:** To merge two binary trees into a larger one.

Insertion Algorithm	**Deletion Algorithm**		
Algorithm for inserting a node:	**The node containing the data has no children**	**The node containing the data has one children**	**The node containing the data has two children**

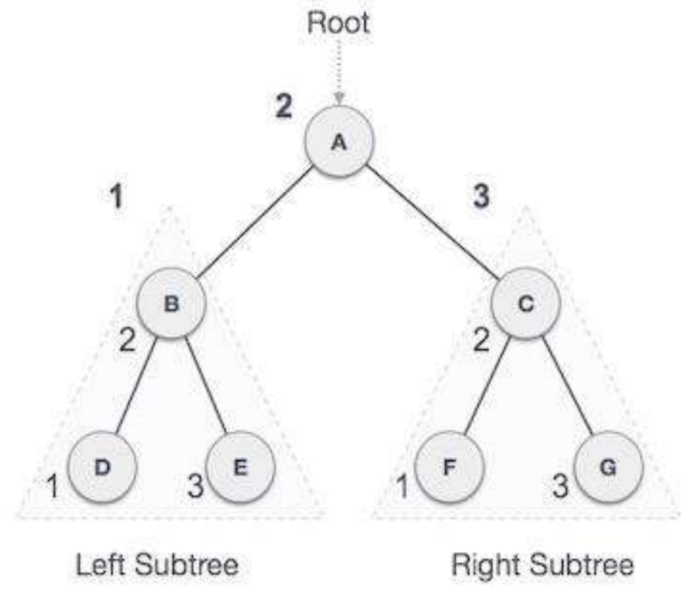
Struct node* curr;
Curr=root;
While(curr)
{
Parent=cur;
If(t→data > curr→data)
{
Curr=curr→right;
}
Else
{
Curr=curr→left;
}
}
If(t→data > parent→data)
{
Parent→right=t;
}
Else
{
Parent→left=t;
}
}
	If(curr==parent→left
	)
	{
	Parent→left=null;
	}
	
	Else
	{
	Parent→right=null;
	}
	Free(curr);
	}
		If(curr→left!=null)
		{
		If(curr==parent→left)
		{
		Parent→left=cur→left;
		}
		Curr→left=null;
		Free(curr);
		}
			If(curr→left!=null
			&&
			cur→right!=null)
			{
			Int t;
			T=curr→right;
			If(t→right!=null
			&& t→left==null)
			{
			Curr→data=t→data;
			Curr→right=t→righ
			t;
			T→right=null;
			Free(t);
			}

TRAVERSAL OF A BINARY TREE

- Traversal is a process to visit all the nodes of a tree and may print their values.
- That is, we cannot randomly access a node in a tree.
- There are three ways which we use to traverse a tree −
  1. In-order Traversal
  2. Pre-order Traversal
  3. Post-order Traversal

In-order Traversal

- In this traversal method, the left subtree is visited first, then the root and later the right sub-tree.
- We should always remember that every node may represent a subtree itself.
- If a binary tree is traversed in-order, the output will produce sorted key values in an ascending order.



D → B → E → A → F → C → G

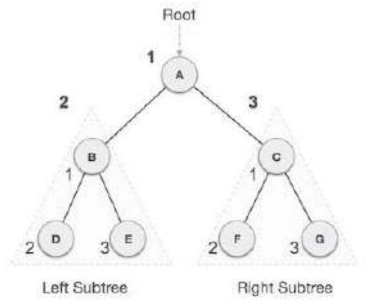
- We start from A, and following in-order traversal, we move to its left subtree B.
- B is also traversed in-order. The process goes on until all the nodes are visited.
- The output of inorder traversal of this tree will be −

Algorithm

```
Until all nodes are traversed −
Step 1 − Recursively traverse left subtree.
Step 2 − Visit root node.
Step 3 − Recursively traverse right subtree.
```

Pre-order Traversal

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



A → B → D → E → C → F → G

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

Algorithm

Until all nodes are traversed −

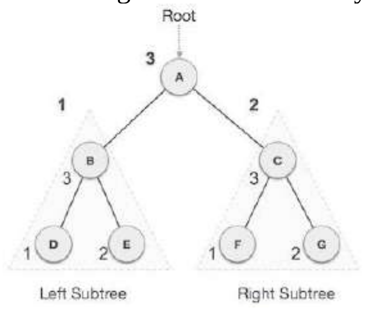
Step 1 − Visit root node.

Step 2 − Recursively traverse left subtree.

Step 3 − Recursively traverse right subtree.

Post-order Traversal

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



D → E → B → F → G → C → A

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

Algorithm

Until all nodes are traversed −

Step 1 − Recursively traverse left subtree.

Step 2 − Recursively traverse right subtree.

Step 3 − Visit root node.

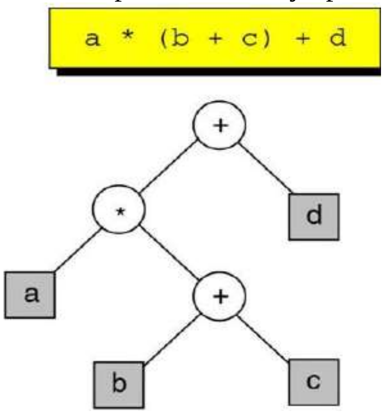
TYPES OF BINARY TREES

There are several types of binary trees

1. Expression tree
2. Binary search tree
3. Heap tree
4. Threaded binary tree
5. Huffman binary tree
6. Height balanced tree (AVL tree)
7. Decision tree

1. Expression tree:

- An expression tree is a binary tree which stores an arithmetic expression.
- The leaves of an expression tree are operand, such as constants or variable names and all internal nodes are the operators.
- Expression tree is always a binary tree because an arithmetic expression contains either binary operator or unary operator.



Operations on the expression tree:

- There are two operations are possible on any expression tress:

1. Traversing
2. Evaluating

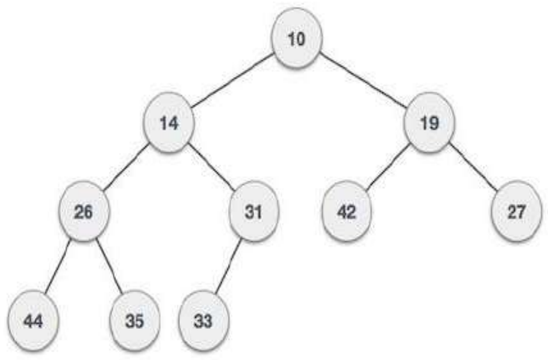
- Traversal operations are same as binary tree such as inorder, preorder and postorder.
- Evaluating of the expression is to evaluate the expression for which the tree is found.

3. HEAP TREES

- Heap is a complete binary tree it will be termed as heap tree, if it satisfies the

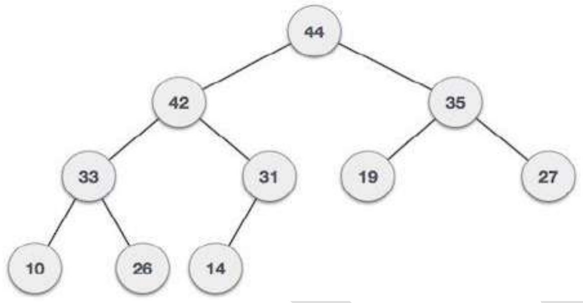
Min-Heap:

- Where the value of the root node is less than or equal to either of its children.



Max-Heap:

- Where the value of the root node is greater than or equal to either of its children.



Representation of a heap tree:

- A heap tree can be represented using linked structure.

Operations on heap tree:

1. *Insert into a heap tree*
2. *Deletion of a node*
3. *Merging two heap trees*

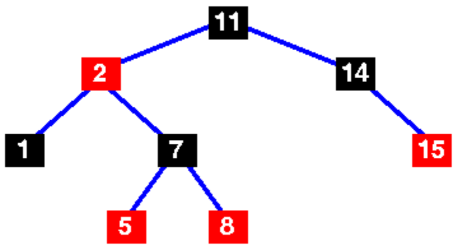
1. Insert into a heap tree

- This operation is used to insert a node into an existing heap tree satisfy the properties if heap tree.
  Step 1 − Create a new node at the end of heap.
  Step 2 − Assign new value to the node.
  Step 3 − Compare the value of this child node with its parent.
  Step 4 − If value of parent is less than child, then swap them.
  Step 5 − Repeat step 3 & 4 until Heap property holds.

RED-BLACK TREE

A red-black tree is a binary search tree which has the following *red-black properties*:

1. Every node is either red or black.
2. Every leaf (NULL) is black.
3. If a node is red, then both its children are black.
4. Every simple path from a node to a descendant leaf contains the same number of black nodes.



2. Deletion of a node from a heap tree:

- Any node can be deleted from a heap tree. Deleting the root node has some special importance.
  Step 1 − Remove root node.
  Step 2 − Move the last element of last level to root.
  Step 3 − Compare the value of this child node with its parent.
  Step 4 − If value of parent is less than child, then swap them.
  Step 5 − Repeat step 3 & 4 until Heap property holds.

3. Merging two heap tree

The two heap trees H1 and H2. Merging the tree H2 with H1 means to include all the nodes from H2 to H1. H2 may be min heap or max heap and the resultant tree will be min heap if H1 is min heap else it will be max heap.
This operation consists of two steps:

1. Delete the root node say x from H2.
2. Insert the node x into H1 satisfying the property of H1.

------------------------------End----------------------------

GRAPHS
Introduction:

- Graph is another non-linear data structure.
- It is hierarchical relationship between parent and children's.

Application:

- Airlines
- Source-destination network
- Konigsberg's bridges.
- Flowchart of a program

Graph Terminology:

Graph:

- A graph consist of two sets
  (i) A set V called set of all vertices (or node)
  (ii) A set E called set of all edges (or arcs)m
  Set of vertices = { 1,2,3}
  Set of edges={ (1,2),(1,3)}

Digraph:

- A digraph is also called a directed graph. If a graph G, such that G=<V,E>, where V is the set of all vertices and E is the set of ordered pair of elements from V.
- Here G2 is a Digraph where
  V = {v1, v2, v3, v4}
  E = {(v1, v2), (v1, v3), (v2, v3), (v3, v4), (v4, v1)}

Weighted graph:

- A graph is termed as weighted graph if all the edges in it are labeled with some weight.
- Ex: G3 and G4 are two weighted graphs.

Adjacent vertices:

- A vertex v_i is adjacent to another vertex say v_j if there is an edge from v_i to v_j.
- Ex: Graph G11, v2 is adjacent to v3 and v4.

Self loop:

- If there is an edge whose starting and end vertices are same, that is (vi,vj) is an edge then it is called a self loop.
- Ex: GraphG5

Parallel edges:

- If there are more than one edges between the same pair of vertices, then they are known as the parallel edge.
- Ex: Graph G5.

Multi graph:

- A graph which has either self loop or parallel edges or both is called multi graph.
- Ex: Graph G5.

Simple graph (digraph):

- A graph if it does not have any self loop or parallel edges is called a simple graph.
- Ex: graph G5.

Complete graph:

- A graph G is said to be complete if each vertex vi adjacent to every other vertex vj in G
- Ex: Grapg G6 and G9.

Acyclic graph:

- If there is a path containing one or more edges which starts from a vertex vi and terminates into the same vertex then the path is known as a cycle.
- Ex: graph G4 and G7.

Isolated vertex:

- A vertex is isolated if there is no edge connected from any other vertex to the vertex.
- Ex: Graph G8.

Degree of vertex:

- The number of edges connected with vertex v_i called the degree of vertex v_i and is denoted by degree (v_i).
- In digraph there are two degrees: indegree and ourdegree.
- **Indegree** of v_i denoted as indegree(v_i) = number of edges incident into $v_{i.}$
- **Outdegree**(vi) = number of edges emanating from vi.
- Ex: In graph G4indegree(v_1)=2 outdegree(v_1)=1
  indegree(v_2)=2 outdegree(v_1)=0

Pendent vertex:

- A vertex v_i is pendent if its indegree(v_i)=1 and outdegree (v_i)=0
- Ex: G8 is a pendent vertex.

Connected graph:

- In a graph G two vertices v_I and v_j are said to be connected if there is a path in G from v_i to v_j.
- A graph is said to be connected if for every pair of distinct vertices v_i, v_j in G there is a path.
- Ex: Graph G1, G3 and G6.

<u>REPRESENTATION OF GRAPHS</u>

- A graph can be represented in many ways
  1. Set representation
  2. Linked representation
  3. Sequential (matrix) representation

(i) Set representation:

This is one of the straightforward methods of representing a graph. In this method two sets are maintained (i) V is the set of vertices
(ii) E is the set of edges.

Graph G1

V(G1)= { v1,v2,v3,v4,v5,v6,v7}
E(G1)= { (v1,v2), (v1,v3), (v2,v4), (v2,v5),(v3,v6),(v3,v7)}

Graph G2

V(G2)= { v1,v2,v3,v4,v5,v6,v7}
E(G2)= { (v1,v2), (v1,v3), (v2,v4), (v2,v5),(v3,v4),(v3,v6), (v4,v7),(v5,v7),(v6,v7)}

Graph G3

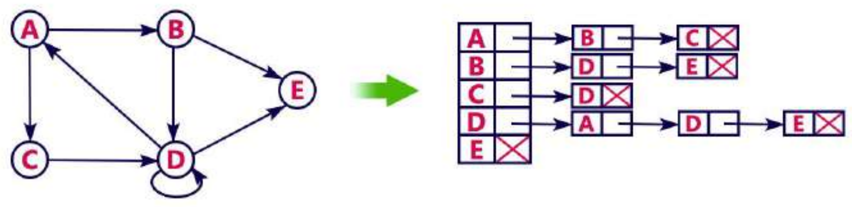
V(G3)= { A,B,C,D,E}
E(G3)= { (A,B), (A,C), (C,B), (C,A),(D,A),(D,B),(D,C),(D,E),(E,B)}

Graph G4

V(G4)= { A,B,C,D}
E(G4)= { (3,A,C), (5,B,A), (1,B,C), (7,B,D),(2,C,A),(4,C,D),(6,D,B),(8,D,C)}

(ii) Linked representation:

- Linked representation is another space-saving way of graph representation.
- Node structure is,

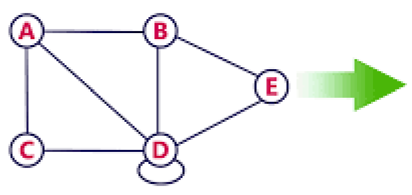


- Linked representation of graphs, the number of lists depends on the number of vertices in the graph.
- The header node in each list maintains a list of all adjacent vertices of anode for which header node is meant.

iii. Matrix Representation

- The adjacency matrix is represents in 2D array of size n*n matrix in which n is the number of vertices.

Ex: A[i][j]=1 means that the adjacency matrix has one edge.
A[i][j]=0 means that the adjacency matrix has no edges.



$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 1 & 1 & 0 \\ B & 1 & 0 & 0 & 1 & 1 \\ C & 1 & 0 & 0 & 1 & 0 \\ D & 1 & 1 & 1 & 1 & 1 \\ E & 0 & 1 & 0 & 1 & 0 \end{array}$$

Adjacency matrix for undirected graph:

- The adjacency matrix for an undirected graph is symmetric.
- The adjacency matrix for a directed graph need not be symmetric.
- The space needed to represent a graph using its adjacency matrix is n^2 locations.

<u>OPERATIONS ON GRAPHS</u>

<u>Insertion</u>

- To insert a vertex and hence establishing connectivity with other vertices in the existing graph
- To insert an edge between vertices in the graph.

<u>Deletion</u>

- To delete a vertex from the graph.
- To delete an edge from the graph.

<u>Merging</u>

- To merge two graph G1 and G2 into a single graph.

<u>Traversal</u>

- To visit all the vertices in the graph.

Operations on Linked List Representation of Graphs:

- In this representation using two representations.

(i) An array of vertices having two fields: LABEL – label for the vertices, LINK-the pointer to the linked list.

(ii) A linked list to maintain the list of all adjacent vertices for any vertex vi for which it is meant.

(iii) A node structure has two fields other than the filed LINK.

(iv) The first field WEIGHT is to store the weight of the edge and the second field LABLE to store the vertex's label

<u>Insertion</u>

- Insertion procedure differs for undirected graph and directed graph.
- To insert of a vertex into an undirected graph, if V_x is inserted and V_i be its adjacent vertex V_i has to be incorporated in the adjacency list of V_x as well as has to be incorporated in the adjacency list of V_i.
- If it's a digraph and if there is a path from V_x to V_i we add a node for V_i into the adjacency list of V_x, if there is an edge from V_i to V_x add a node for V_x in the adjacency list of V_i.

1) Algorithm INSERT_VERTEX_LL_UG(V_x, X)

Vx is the new vertex that has to be inserted into a graph.

Steps:

```
1. N=N+1, $V_x$=N
2. For i=1 to l do
    1. Let j=X[i]
    2. If (j>=N) then
            1. Print "No vertex labeled X[i] exist; edge from $V_x$ into the list of vertices."
    3. Else
            1. INSERT_SL-END(UGptr[N],x[i])
            2. INSERT_SL-END(UGptr[j],Vx)
    4. Endif
3. Endfor
4. Stop
```

2) Algorithm INSERT_VERTEX_LL_DG(V_x, X,Y)
Vx is the new vertex that has to be inserted into a graph.
Steps:

```
1. N=N+1, $V_x$=N
2. For i=1 to m do
       1. Let j=X[i]
       2. If (j>=N) then
           1. Print "No vertex labeled X[i] exist; edge from $V_x$ to X[i] is not established"
       3. Else
           1. INSERT_SL-END (DGptr[N],x[i])
     4. Endif
3. Endfor
4. For i=1 to n do
       1. Let j=Y[i]
       2. If (j>=N) then
           1. Print "No vertex labeled Y[i] exist; edge from $V_x$ to X[i] is not established"
       3. Else
           1. INSERT_SL-END (DGptr[j],Vx)
     4. Endif
5. Endfor
6. stop
```

3) Insert the edge between two vertices in Undirected graph:
Algorithm: INSERT_EDGE_LL_UG(Vi,Vj)

Insert the edge to be inserted between vertices Vi and Vj.

Steps:

```
1. Let N=number of vertices in the graph
2. If(Vi>N) or (Vj>N) then
       1. Print" Edge is not possible between Vi and Vj"
3. Else
       1. INSERT_SL_END (UGptr[Vi],Vj"
       2. INSERT_SL_END (UGptr[Vj],Vi"
4. Endif
5. Stop
```

4)Insert the edge between two vertices in directed graph:
Algorithm: INSERT_EDGE_DG (Vi,Vj)

Insert the edge to be inserted between vertices Vi and Vj.

Steps:

```
1. Let N=number of vertices in the graph
2. If(Vi>N) or (Vj>N) then
       1. Print" Edge is not possible between Vi and Vj"
3. Else
       1. INSERT_SL_END (UGptr[Vi],Vj")
4. Endif
5. Stop
```

Deletion:

1) Delete the vertex:

Algorithm Delete_Vertex_LL_UG (Vx)

Step:

```
1.  If (N=0) then
        1.  Print" Graph is empty : No deletion"
        2.  Exit
2.  Endif
3.  ptr=UGptr[Vx] . LINK
4.  While(ptr ≠ NULL) do
        1.  j=ptr.LABEL
        2.  DELETE_SL_ANY(UGptr[j],Vx)
        3.  DELETE_SL_ANY(UGptr[Vx], j)
        4.  ptr=UGptr[Vx].LINK
5.  Endwhile
6.  UGptr[Vx].LABEL=NULL
7.  UGptr[Vx].LINK=NULL
8.  RETURN_NODE(ptr)
9.  N=N-1
10. Stop
```

2) To delete the edge between vertices Vi and Vj

Algorithm: Delete_Edge_LL_UG(Vi,Vj)

```
1.  let N=number of vertices in the graph
2.  if(Vi>N ) or ( Vj >N) then
    1. Print "Vertex does not exist: Error in edge removal"
3.  Else
        1.  DELETE_SL_ANY(UGptr[Vi],Vj)
        2.  DELETE_SL_ANY(UGptr[Vj],Vi)
4.  Endif
5.  Stop
```

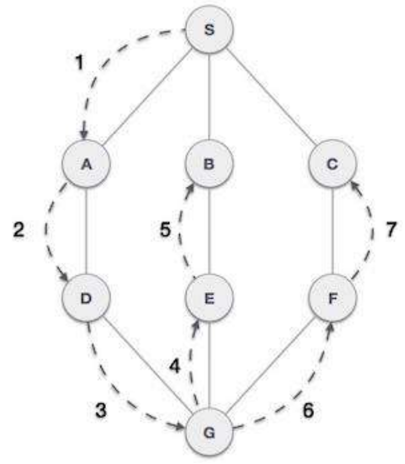
GRAPH TRAVESAL

In the traversal of a binary tree there are two ways as follows.

- **Depth First search**
- **Breadth first search**

Depth First Search:

- Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

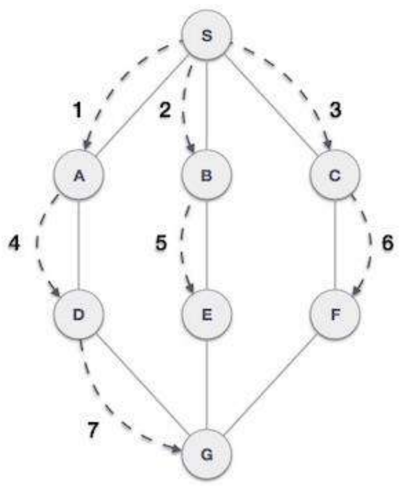


- As in the example given above, DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to C.

- It employs the following rules.
  1. Rule 1 − Visit the adjacent unvisited vertex.
  2. Mark it as visited.
  3. Display it.
  4. Push it in a stack.
  5. Rule 2 − If no adjacent vertex is found, pop up a vertex from the stack.
  6. It will pop up all the vertices from the stack, which do not have adjacent vertices.
  7. Rule 3 − Repeat Rule 1 and Rule 2 until the stack is empty.

Breadth First Search:

- Breadth First Search (BFS) algorithm traverses a graph in a breadthward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



- As in the example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs the following rules.

  1. Rule 1 − Visit the adjacent unvisited vertex.

  2. Mark it as visited.

  3. Display it.

  4. Insert it in a queue.

  5. Rule 2 − If no adjacent vertex is found, remove the first vertex from the queue.

  6. Rule 3 − Repeat Rule 1 and Rule 2 until the queue is empty.

<u>APPLICATION OF GRAPH STRUCTURES</u>

- Graph is an important data structure whose extensive applications are known in almost all application areas.
- Conversion of a particular problem into this general graph theoretic problem will rest on the reader.
  1. Shortest path problem
  2. Topological sorting of a graph
  3. Spanning trees

SHORTEST PATH PROBLEM

- The shortest path problem is about finding a path between 2 vertices in a graph such that the total sum of the edges weights is minimum.
- To find shortest problem, we have three algorithm,
    1. Floyd & Warshall's Algorithms
    2. Dijkstra's Algorithm

<u>Warshall's Algorithms:</u>

- This is a classical algorithm by which we can determine whether there is a path from any vertex Vi to another vertex Vj either directly or through one or more intermediate vertices.

Steps:

```
1.  For i=0 to N do
    1. For j=1 to N do
            1. P[i][j]= Gptr[i][j]
    2. Endfor
2.  End for
3.  For k=1 to N do
        1.  For i=1 to N do
            1. For j=1 to N do
                    1. P[i][j]= P[i][j] v (P[i][k] ^ P[k][j])
            2. Endfor
        2. Endfor
4.  End for
5.  Return(p)
6.  Stop
```

<u>Floyd's Algorithm</u>**:**

- The basic structure of the Floyd's algorithm is same as Warshall's algorithm.

Steps:

```
1. For i=1 to N do
    1. For j=1 to N do
            1.  If (Gptr[i][i] =0) then
                    1. Q[i][j]=∞
                    2. PATHS[i][j]=NULL
            2. Else
                    1. Q[i][j]=Gptr[i][j]
                    2. P=COMBINE (i,j)
                    3. PATHS[i][j]=P
            3. Endif
    2. Endfor
2. Endfor
3. For k=1 to N do
        1. For i=1 to N do
            1. For j=1 to N do
                    1. Q[i][j]= MIN(Q[i][j], Q[i][k]+ Q[k][j])
                    2. If (Q[i][k]+ Q[k][j]< Q[i][j]) then
                            1. p1= PATHS[i][k]
                            2. p2= PATHS[k][j]
                            3. PATHS[i][j]=COMBINE(p1,p2)
                    3. Endif
```

```
                2. Endfor
            2. Endfor
    4. End for
    5. Return (Q, PATHS)
    6. Stop
```

Dijkstra's Algorithm:

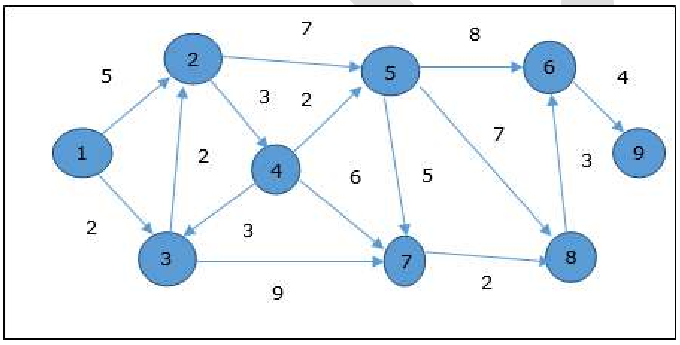
- Dijkstra's algorithm solves the single-source shortest-paths problem on a directed weighted graph G = (V, E), where all the edges are non-negative.

Example

- Let us consider vertex 1 and 9 as the start and destination vertex respectively.
- Initially, all the vertices except the start vertex are marked by ∞ and the start vertex is marked by 0.

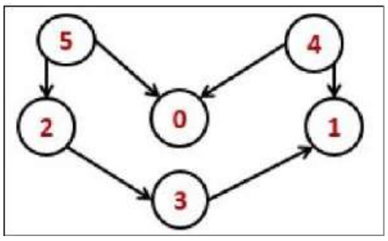
Vertex	Initial	Step1 V1	Step2 V3	Step3 V2	Step4 V4	Step5 V5	Step6 V7	Step7 V8	Step8 V6
1	0	0	0	0	0	0	0	0	0
2	∞	5	4	4	4	4	4	4	4
3	∞	2	2	2	2	2	2	2	2
4	∞	∞	∞	7	7	7	7	7	7
5	∞	∞	∞	11	9	9	9	9	9
6	∞	∞	∞	∞	∞	17	17	16	16
7	∞	∞	11	11	11	11	11	11	11
8	∞	∞	∞	∞	∞	16	13	13	13
9	∞	∞	∞	∞	∞	∞	∞	∞	20

- Hence, the minimum distance of vertex 9 from vertex 1 is 20. And the path is
  1→ 3→ 7→ 8→ 6→ 9



Topological Sorting:

- Topological sorting is an ordering of vertices of a graph, such that if there is a path from u to v in the graph then u appears before v in the ordering.
- A topological ordering is not possible if the graph has a cycle, since for two vertices u and v on the cycle, u precedes v and v precedes u.
- A simple algorithm to find a topological ordering is to find out any vertex with in degree zero, that is vertex without any predecessor.

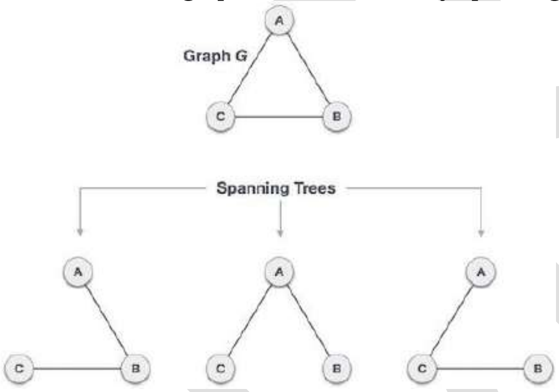


Algorithm:

```
Begin
   Initially mark all nodes as unvisited
   For all nodes v of the graph, do
      If v is not visited, then
         TopoSort (i, visited, stack)
   Done
   Pop and print all elements from the stack
End.
```

MINIMUM SPANNING TREE

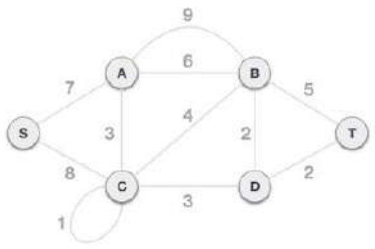
- A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges.
- Hence, a spanning tree does not have cycles and it cannot be disconnected.
- A disconnected graph does not have any spanning tree.



- There are two methods are efficient:
    1. Kruskal's algorithm
    2. Prim's Algorithm

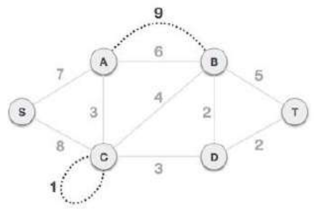
Kruskal's Algorithm:

- Kruskal's algorithm to find the minimum cost spanning tree uses the greedy approach.
- This algorithm treats the graph as a forest and every node it has as an individual tree.
- A tree connects to another only and only if, it has the least cost among all available options and does not violate MST properties.
- To understand Kruskal's algorithm let us consider the following example −



Step 1 - Remove all loops and Parallel Edges

- Remove all loops and parallel edges from the given graph.



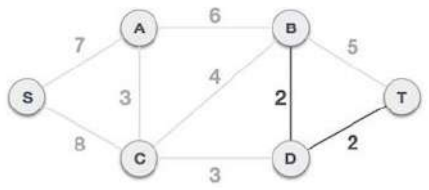
Step 2 - Arrange all edges in their increasing order of weight

- The next step is to create a set of edges and weight, and arrange them in an ascending order of weightage (cost).

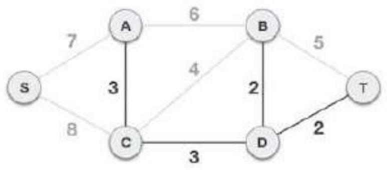
B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8

Step 3 - Add the edge which has the least weightage

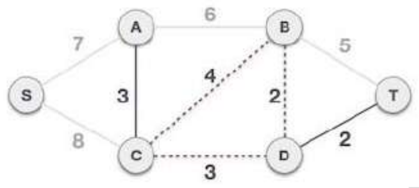
- Now we start adding edges to the graph beginning from the one which has the least weight.
- In case, by adding one edge, the spanning tree property does not hold then we shall consider not including the edge in the graph.



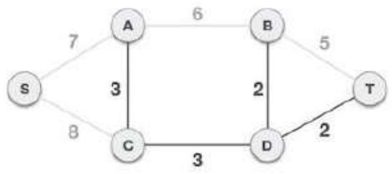
- The least cost is 2 and edges involved are B, D and D,T.
- We add them. Adding them does not violate spanning tree properties, so we continue to our next edge selection.
- Next cost is 3, and associated edges are A,C and C,D. We add them again −



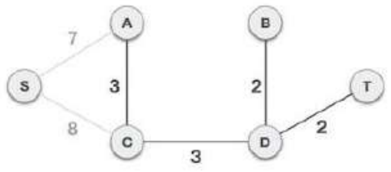
- Next cost in the table is 4, and we observe that adding it will create a circuit in the graph. −



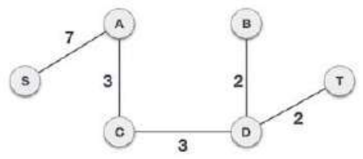
- We ignore it. In the process we shall ignore/avoid all edges that create a circuit.



- We observe that edges with cost 5 and 6 also create circuits.
- We ignore them and move on.



- Now we are left with only one node to be added.
- Between the two least cost edges available 7 and 8, we shall add the edge with cost 7.

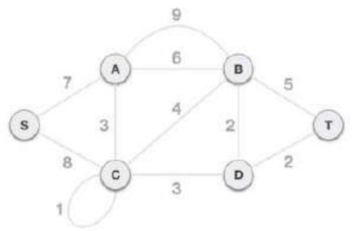


- By adding edge S, A we have included all the nodes of the graph and we now have minimum cost spanning tree.

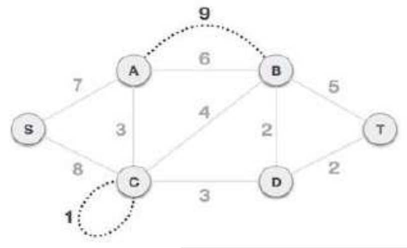
Prim’s Algorithm:

- Prim's algorithm to find minimum cost spanning tree.
- Prim's algorithm, treats the nodes as a single tree and keeps on adding new nodes to the spanning tree from the given graph.

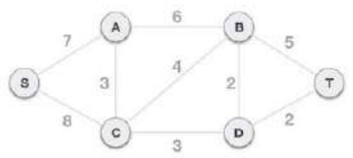
Example −



Step 1 - Remove all loops and parallel edges

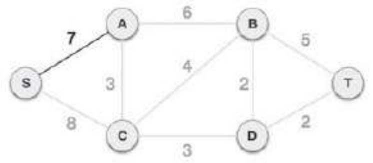


- Remove all loops and parallel edges from the given graph.
- In case of parallel edges, keep the one which has the least cost associated and remove all others.

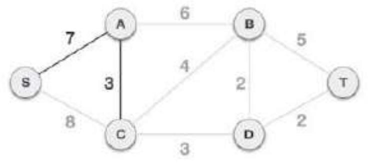


Step 3 - Check outgoing edges and select the one with less cost

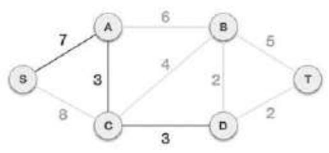
- After choosing the root node S, we see that S, A and S,C are two edges with weight 7 and 8, respectively. We choose the edge S, A as it is lesser than the other.



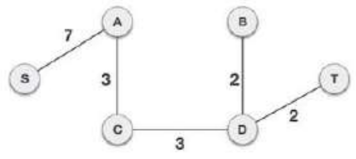
- Now, the tree S-7-A is treated as one node and we check for all edges going out from it
- We select the one which has the lowest cost and include it in the tree.



- After this step, S-7-A-3-C tree is formed. Now we'll again treat it as a node and will check all the edges again.
- However, we will choose only the least cost edge.
- In this case, C-3-D is the new edge, which is less than other edges' cost 8, 6, 4, etc.



- After adding node D to the spanning tree, we now have two edges going out of it having the same cost, i.e. D-2-T and D-2-B.
- Thus, we can add either one.
- But the next step will again yield edge 2 as the least cost.
- Hence, we are showing a spanning tree with both edges included.



---END---------------------------------------

UNIT – V
SEARCHING

What is searching?

- Searching is the process of finding a given value position in a list of values.
- It decides whether a search key is present in the data or not.
- It is the algorithmic process of finding a particular item in a collection of items.
- It can be done on internal data structure or on external data structure.
- There are two types of search,
    1. Linear search
    2. Non-linear search

Searching Techniques

- To search an element in a given array, it can be done in following ways.
    1. Sequential Search/Linear Search.
    2. Binary Search

1. Sequential Search/Linear Search

- Linear search is a very simple search algorithm.
- In this type of search, a sequential search is made over all items one by one.
- Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.

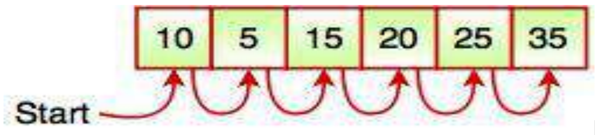
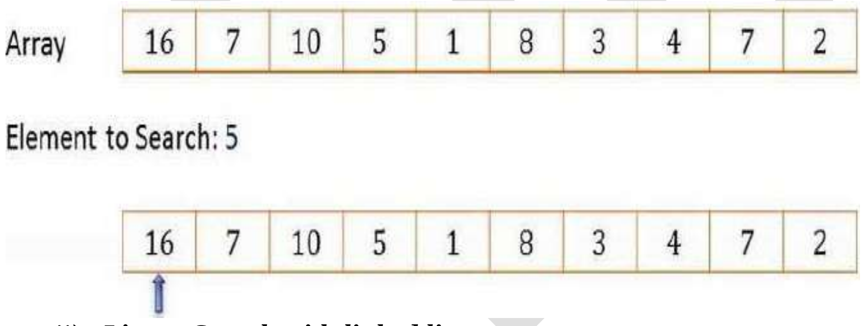


Fig. Sequential Search

i) Linear Search with Array:

- Start the search from the first element and continue till you get the element (or reach end of the array).

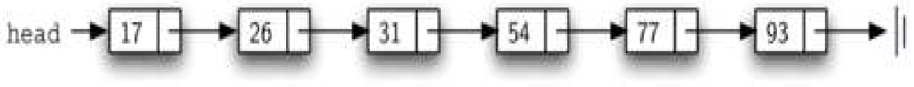


ii) ***Linear Search with linked list****:*

- Our sequential search function for linked lists will take two arguments,
    1. A pointer to the first element in the list.
    2. The value for which we are searching.
- The function will return a pointer to the list structure containing the correct data, or will return NULL if the value wasn't found.

iii) Linear Search with ordered list:

- In order to implement the ordered list, we must remember that the relative positions of the items are based on some underlying characteristic.
- The ordered list of integers given above (17, 26, 31, 54, 77, and 93) can be represented by a linked structure as shown below.
- Again, the node and link structure is ideal for representing the relative positioning of the items.



2. Binary Search/Non Linear Search

- ➢ Binary Search is used for searching an element in a sorted array.
- ➢ It is a fast search algorithm with run-time complexity of O(log n).
- ➢ Binary search works on the principle of divide and conquer.
- ➢ This searching technique looks for a particular element by comparing the middle most element of the collection.
- ➢ It is useful when there are large numbers of elements in an array.

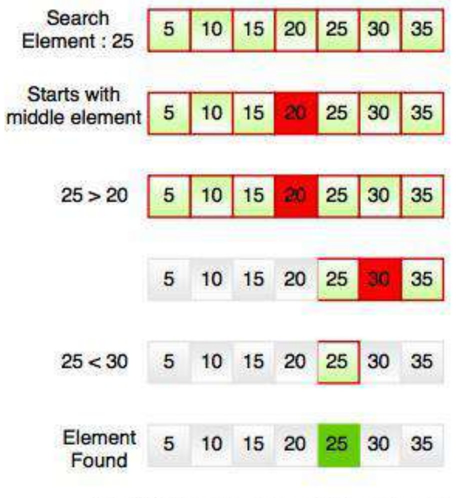


Fig. Working Structure of Binary Search

SORTING

What is sorting?

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

Telephone Directory:

- ➢ The telephone directory stores the telephone numbers of people sorted by their names, so that the names can be searched easily.

Dictionary:

- ➢ The dictionary stores words in an alphabetical order so that searching of any word becomes easy.

TERMINOLOGY:

What are Internal sorting?

- ➢ When all data that needs to be sorted cannot be placed in-memory at a time, the sorting is called external sorting.

What is External Sortings?

- ➢ External Sorting is used for massive amount of data.
- ➢ Merge Sort and its variations are typically used for external sorting.
- ➢ Some external storage like hard-disk, CD, etc is used for external storage. When all data is placed in-memory, then sorting is called internal sorting.

SORTING TECHNIQUES:

In-place Sorting

- ➢ Algorithms may require some extra space for comparison and temporary storage of few data elements.
- ➢ These algorithms do not require any extra space and sorting is said to happen in-place.

Example: Bubble sort is an example of in-place sorting.

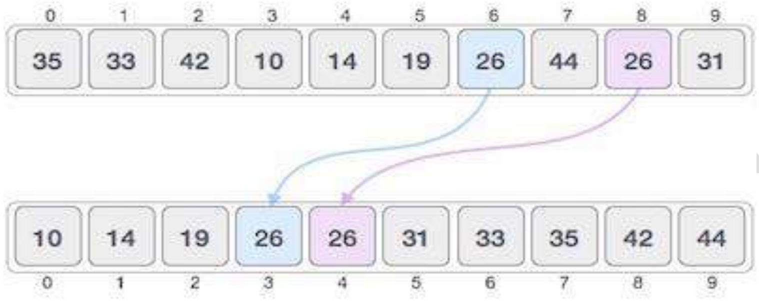
Not-in-place Sorting

- Algorithms requires space which is more than or equal to the elements being sorted.
- Sorting which uses equal or more space is called not-in-place sorting.

Example: Merge-sort is an example of not-in-place sorting.

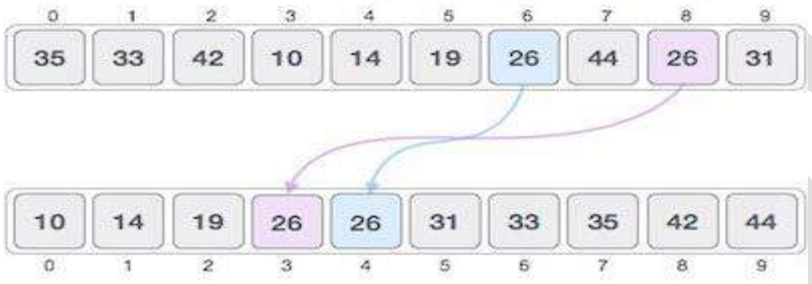
Stable Sorting:

- If a sorting algorithm, after sorting the contents, does not change the sequence of similar content in which they appear, it is called stable sorting.



Not Stable Sorting:

- If a sorting algorithm, after sorting the contents, changes the sequence of similar content in which they appear, it is called unstable sorting.



Adaptive Sorting Algorithm

- A sorting algorithm is said to be adaptive, if it takes advantage of already 'sorted' elements in the list that is to be sorted.
- That is, while sorting if the source list has some element already sorted, adaptive algorithms will take this into account and will try not to re-order them.

Non-Adaptive Sorting Algorithm

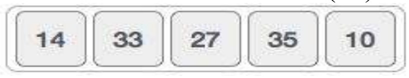
- A non-adaptive algorithm is one which does not take into account the elements which are already sorted.
- They try to force every single element to be re-ordered to confirm their sortedness.

BUBBLE SORT

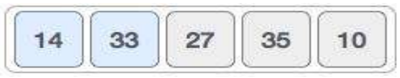
- Bubble sort is a simple sorting algorithm.
- This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order.
- This algorithm is not suitable for large data sets as its average and worst case complexities are of O (n2) where n is the number of items.

How Bubble Sort Works?

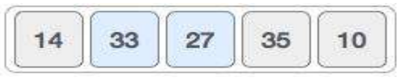
- We take an unsorted array for our example.
- Bubble sort takes O(n2) time so we're keeping it short and precise.



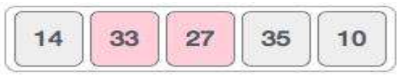
- Bubble sort starts with very first two elements, comparing them to check which one is greater.



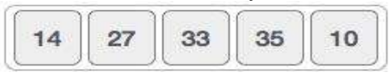
- In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.



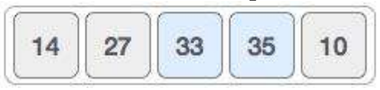
- We find that 27 is smaller than 33 and these two values must be swapped.



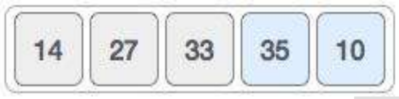
- The new array should look like this −



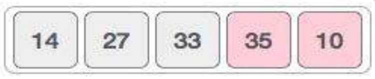
- Next we compare 33 and 35. We find that both are in already sorted positions.



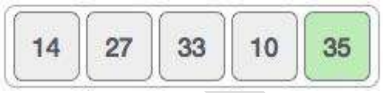
- Then we move to the next two values, 35 and 10.



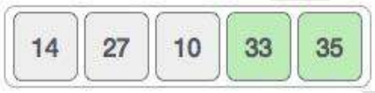
- We know then that 10 is smaller 35. Hence they are not sorted.



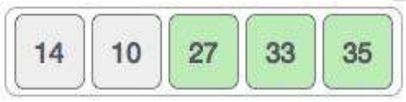
- We swap these values.
- We find that we have reached the end of the array.
- After one iteration, the array should look like this −



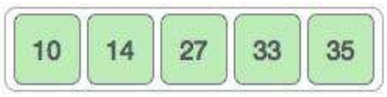
- To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this −



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



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



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

Algorithm

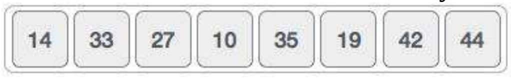
```
Begin BubbleSort (list)
  for all elements of list
    If list[i] > list [i+1]
      Swap (list[i], list [i+1])
    End if
  End for
    Return list
  End Bubble Sort
```

INSERTION SORT

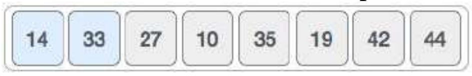
- This is an in-place comparison-based sorting algorithm.
- Here, a sub-list is maintained which is always sorted.
- For example, the lower part of an array is maintained to be sorted.
- An element which is to be 'inserted in this sorted sub-list, has to find its appropriate place and then it has to be inserted there.
- Hence the name, insertion sorts.
- The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array).
- This algorithm is not suitable for large data sets as its average and worst case complexity are of O(n2), where n is the number of items.

How Insertion Sort Works?

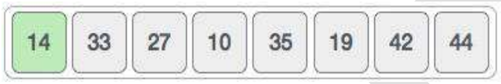
- We take an unsorted array for our example.



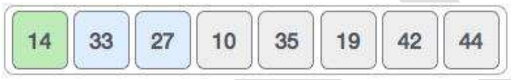
- Insertion sort compares the first two elements.



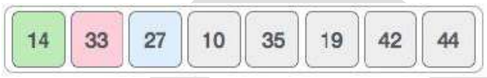
- It finds that both 14 and 33 are already in ascending order.
- For now, 14 is in sorted sub-list.



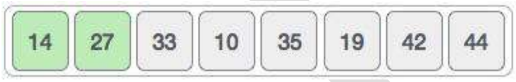
- Insertion sort moves ahead and compares 33 with 27.



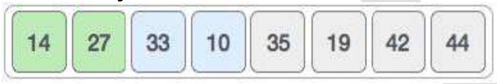
- And finds that 33 is not in the correct position.



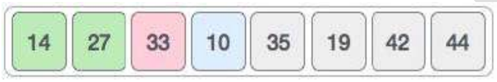
- It swaps 33 with 27. It also checks with all the elements of sorted sub-list.
- Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14.
- Hence, the sorted sub-list remains sorted after swapping.



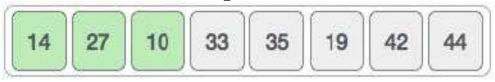
- By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.



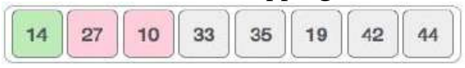
- These values are not in a sorted order.



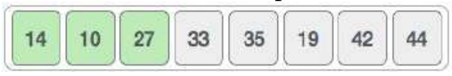
- So we swap them.



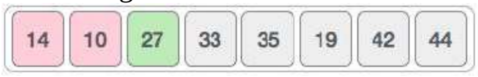
- However, swapping makes 27 and 10 unsorted.



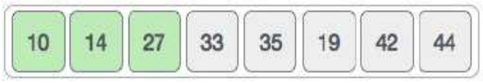
- Hence, we swap them too.



- Again we find 14 and 10 in an unsorted order.



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



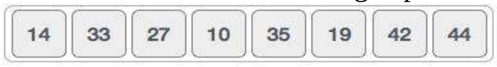
- This process goes on until all the unsorted values are covered in a sorted sub-list.
- Now we shall see some programming aspects of insertion sort.

SELECTION SORT

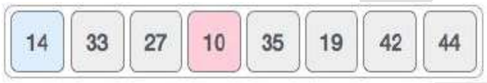
- Selection sort is a simple sorting algorithm.
- This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end.
- Initially, the sorted part is empty and the unsorted part is the entire list.
- The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array.
- This process continues moving unsorted array boundary by one element to the right.
- This algorithm is not suitable for large data sets as its average and worst case complexities are of O (n2), where n is the number of items.

How Selection Sort Works?

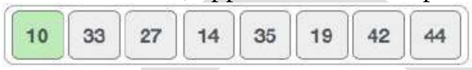
- Consider the following depicted array as an example.



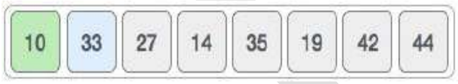
- For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.



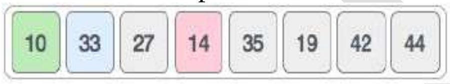
- So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.



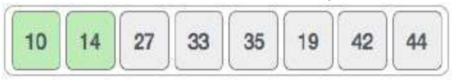
- For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.



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



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



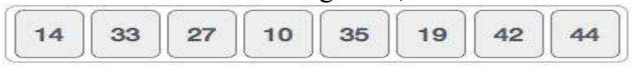
- The same process is applied to the rest of the items in the array.
- Following is a pictorial depiction of the entire sorting process –

MERGE SORT

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

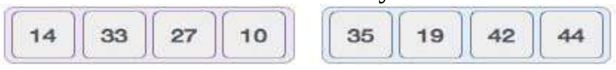
How Merge Sort Works?

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

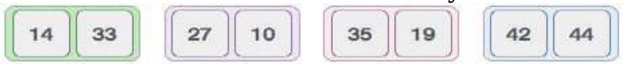


- We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved.

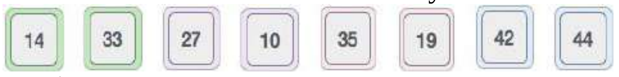
- We see here that an array of 8 items is divided into two arrays of size 4.



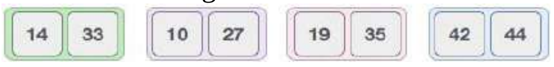
- This does not change the sequence of appearance of items in the original.
- Now we divide these two arrays into halves.



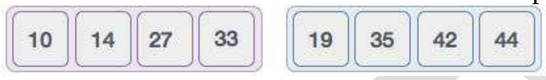
- We further divide these arrays and we achieve atomic value which can no more be divided.



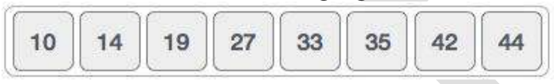
- Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.
- We first compare the element for each list and then combine them into another list in a sorted manner.
- We see that 14 and 33 are in sorted positions.
- We compare 27 and 10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



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



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



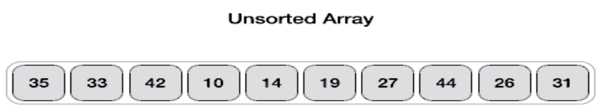
- Now we should learn some programming aspects of merge sorting.

QUICK SORT

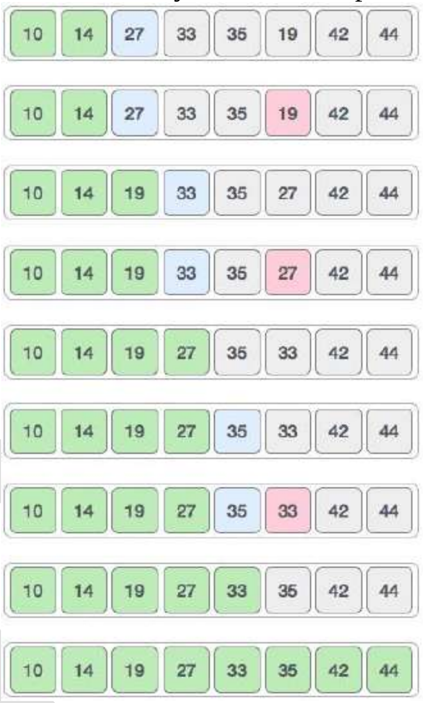
- Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays.
- A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.
- Quick sort partitions an array and then calls itself recursively twice to sort the two resulting subarrays.

Partition in Quick Sort

- Following animated representation explains how to find the pivot value in an array.



➢ The pivot value divides the list into two parts. And recursively, we find the pivot for each sub-lists until all lists contains only one element.



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

HEAP SORT

➢ Heap sort is a comparison based sorting algorithm.
➢ It is a special tree-based data structure.
➢ Heap sort is similar to selection sort. The only difference is, it finds largest element and places it at the end.
➢ This sort is not a stable sort. It requires a constant space for sorting a list.
➢ It is very fast and widely used for sorting.
➢ It has following two properties:
   1. Shape Property
   2. Heap Property

Shape Property

➢ Shape property represents all the nodes or levels of the tree are fully filled.
➢ Heap data structure is a complete binary tree.

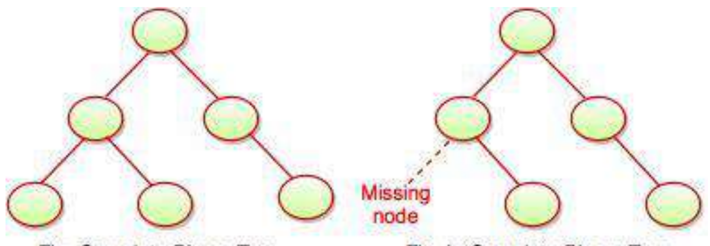


Fig. Complete-Binary Tree   Fig. In-Complete-Binary Tree

Heap property

➢ Heap property is a binary tree with special characteristics.
➢ It can be classified into two types,
   1. Max-Heap.
   2. Min Heap.

<u>Max Heap</u>:

➢ If the parent nodes are greater than their child nodes, it is called a Max-Heap.

<u>Min Heap:</u>

➢ If the parent nodes are smaller than their child nodes, it is called a Min-Heap

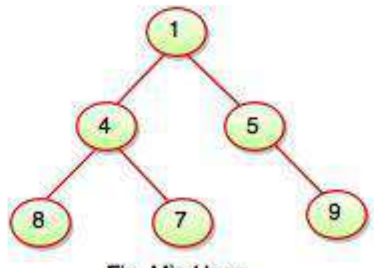


Fig. Min Heap

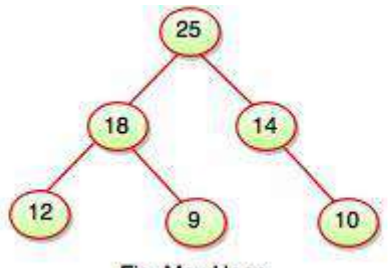


Fig. Max Heap