PG-DAC SEPT-2021

ALGORITHMS & DATA STRUCTURES

SACHIN G. PAWAR

SUNBEAM INSTITUTE OF INFORMATION & TECHNOLOGIES

PUNE & KARAD

Data Structures: Introduction

Name of the Module : Algorithms & Data Structures Using Java.

Prerequisites: Knowledge of programming in C/C++/Java with Object Oriented Concepts.

Weightage : 100 Marks (Theory Exam : 40% + Lab Exam : 40% + Mini Project : 20%).

# Importance of the Module:

1. CDAC - Syllabus
2. To improve programming skills
3. Campus Placements
4. Applications in Industry work

Data Structures: Introduction

Q. Why there is a need of data structure?

- There is a need of data structure to achieve 3 things in programming:

1. **efficiency**
2. **abstraction**
3. **reusability**

Q. What is a Data Structure?

Data Structure is **a way to store data elements into the memory** (i.e. into the main memory) in **an organized manner** so that operations like **addition, deletion, traversal, searching, sorting** etc... can be performed on it efficiently.

Data Structures: Introduction

Two types of **Data Structures** are there:

1. Linear / Basic data structures : data elements gets stored / arranged into the memory in a **linear manner** (e.g. sequentially) and hence can be accessed linearly / sequentially.

- **Array**
- **Structure & Union**
- **Linked List**
- **Stack**
- **Queue**

2. Non-Linear / Advanced data structures : data elements gets stored / arranged into the memory in a **non-linear manner** (e.g. hierarchical manner) and hence can be accessed non-linearly.

- **Tree (Hierarchical manner)**
- **Binary Heap**
- **Graph**
- **Hash Table(Associative manner)**

Data Structures: Introduction

+ Array: It is a **basic / linear data structure** which is **a collection / list of logically related similar type of data elements** gets stored/arranged into the memory at **contiguos locations**.

+ Structure: It is a **basic / linear data structure** which is **a collection / list of logically related similar and disimmilar type of data elements** gets stored/arranged into the memory **collectively i.e. as a single entity/record.**

sizeof of the structure = sum of size of all its members.

+ Union: Union is same like structure, except, memory allocation i.e. size of union is the size of max size member defined in it and that memory gets shared among all its members for effective memory utilization (can be used in a special case only).

Data Structures: Introduction

Q. What is a Program?

- A Program is **a finite set of instructions written in any programming language** (either in a high level programming language like C, C++, Java, Python or in a low level programming language like assembly, machine etc...) given to the machine to do specific task.

Q. What is an Algorithm?

- An algorithm is **a finite set of instructions written in any human understandable language (like english)**, if followed, acomplishesh a given task.

- **Pseudocode :** It is a **special form of an algorithm**, which is a finite set of instructions written in any human understandable language (like english) **with some programming constraints**, if followed, acomplishesh a given task.

- An algorithm is a template whereas a program is an implementation of an algorithm.

Data Structures: Introduction

# Algorithm : to do sum of all array elements

Step-1: initially take value of sum is 0.

Step-2: scan an array seqentially from first element max till last element, and add each array element into the sum.

Step-3: return final sum.

# Pseudocode : to do sum of all array elements

```
Algorithm ArraySum(A, n){//whereas A is an array of size n
   sum=0;//initially sum is 0
   for( index = 1 ; index <= size ; index++ ) {
   sum += A[ index ];//add each array element into the sum
   }
   return sum;
}
```

Data Structures: Introduction

- There are two types of Algorithms OR there are two approaches to write an algorithm:

1. Iterative (non-recursive) Approach :

```
Algorithm ArraySum( A, n){//whereas A is an array of size n
   sum = 0;
   for( index = 1 ; index <= n ; index++ ){
         sum += A[ index ];
   }
      return sum;
}
```

```
e.g. iteration
for( exp1 ; exp2 ; exp3 ){
   statement/s
}
exp1 => initialization
exp2 => termination condition
exp3 => modification
```

Data Structures: Introduction

2. Recursive Approach:
While writing recursive algorithm -> We need to take care about 3 things
1. Initialization: at the time first time calling to recursive function
2. Base condition/Termination condition : at the begining of recursive function
3. Modification: while recursive function call

Example:

```
Algorithm RecArraySum( A, n, index )
{
   if( index == n )//base condition
         return 0;

   return ( A[ index ] + RecArraySum(A, n, index+1) );
}
```

Data Structures: Introduction

Recursion : it is a process in which we can give call to the function within itself.
function for which recursion is used => recursive function
- there are two types of recursive functions:
1. tail recursive function : recursive function in which recursive function call is the last executable statement.

```
void fun( int n ){
   if( n == 0 )
      return;

   printf("%4d", n);
   fun(n--);//rec function call
}
```

Data Structures: Introduction

2. non-tail recursive function : recursive function in which recursive function call is not the last executable statement

```
void fun( int n ){
    if( n == 0 )
        return;

    fun(n--);//rec function call
    printf(“%4d”, n);
}
```

Data Structures: Introduction

- An Algorithm is a solution of a given problem.

- **Algorithm = Solution**

- One problem may has many solutions. For example

Sorting : to arrange data elements in a collection/list of elements either in an ascending order or in descending order.

A1 : Selection Sort
A2 : Bubble Sort
A3 : Insertion Sort
A4 : Quick Sort
A5 : Merge Sort
etc...

- When one problem has many solutions/algorithms, in that case we need to select an efficient solution/algorithm, and to decide efficiency of an algorithm we need to do their analysis.

Data Structures: Introduction

- **Analysis of an algorithm** is a work of determining / calculating how much **time** i.e. computer time and **space** i.e. computer memory it needs to run to completion.

- There are two **measures** of an **analysis of an algorithms:**

1. Time Complexity of an algorithm is the amount of **time i.e. computer time** it needs to run to completion.

2. Space Complexity of an algorithm is the amount of **space i.e. computer memory** it needs to run to completion.

Data Structures: Introduction

# Space Complexity of an algorithm is the amount of space i.e. computer memory it needs to run to completion.
Space Complexity = Code Space + Data Space + Stack Space (applicable only for recursive algo)
Code Space = space required for an **instructions**
Data Space = space required for **simple variables, constants & instance variables.**
Stack Space = space required for **function activation records** (local vars, formal parameters, return address, old frame pointer etc...).

- Space Complexity has **two components:**
1. Fixed component : code space and **data space** (space required for simple vars & constants).
2. Variable component : data space for instance characteristics (i.e. space required for instance vars) and **stack space** (which is applicable only in recursive algorithms).

Data Structures: Introduction

# Calculation of Space complexity of non-recursive algorithm:

```
Algorithm ArraySum( A, n){//whereas A is an array of size n
   sum = 0;
   for( index = 1 ; index <= n ; index++ ){
   sum += A[ index ];
   }
   return sum;
}
```

Sp = Data Space + Instance charactristics
simple vars => formal params: A
local vars => sum, index
constants => 0 & 1
instance variable = n, input size of an array = **n units**
Data Space = 5 units (1 unit for simple var : A + 2 units for local vars : sum & index + 2 units for constants : 0 & 1) => **Data Space = 5 units**
Sp = (n + 5) units.

Data Structures: Introduction

S = C (Code Space) + Sp (Data Space)
S = C + (n+5)
S >= (n + 5) … (as C is constant, it can be neglected)
S >= O(n) => O(n)
Space required for an algo = O(n) => whereas n = input size array.

# Calculation of Space complexity of recursive algorithm:

```
Algorithm RecArraySum( A, n, index ){
   if( index == n )//base condition
            return 0;
   return ( A[ index ] + RecArraySum(A, n, index+1) );
}
```

Space Complexity = Code Space + Data Space + Stack Space (applicable only in recursive algorithms)
Code Space = space required for instructions
Data Space = space required for variables, constants & instance characteristics.
Stack Space = space required for FAR's.

Data Structures: Introduction

- When any function gets called, one entry gets created onto the stack for that function call, referred as **function activation record / stack frame,** it contains **formal params, local vars, return addr, old frame pointer etc...**

In our example of recursive algorithm:

3 units (for A, index & n) + 2 units (for constants 0 & 1) = total 5 **units** of memory is required per function call.

- for size of an array = **n,** algo gets called **(n+1) no. of times.**

Hence, total space required = **5 * (n+1)**

S = 5n + 5

=> S >= 5n

=> S >= 5n

=> S ~= 5n => O(n), wheras n = size of an array

Data Structures: Introduction

# Time Complexity:

Time Complexity = Compilation Time + Execution Time

Time complexity has two components :

1. Fixed component : compilation time

2. Variable component : execution time => it depends on instance characteristics of an algorithm.

Example :

```
Algorithm ArraySum( A, n){//whereas A is an array of size n
    sum = 0;
    for( index = 1 ; index <= n ; index++ ){
    sum += A[ index ];
    }

    return sum;
}
```

Data Structures: Introduction

- for size of an array = 5 => instruction/s inside for loop will execute 5 no. of times
- for size of an array = 10 => instruction/s inside for loop will execute 10 no. of times
- for size of an array = 20 => instruction/s inside for loop will execute 20 no. of times
- for size of an array = **n** => instruction/s inside for loop will execute **"n"** no. of times

# Scenario-1 :

Machine-1 : Pentium-4 : Algorithm : input size = 10
Machine-2 : Core i5 : Algorithm : input size = 10

# Scenario-2 :

Machine-1 : Core i5 : Algorithm : input size = 10 : system fully loaded with other processes
Machine-2 : Core i5 : Algorithm : input size = 10 : system not fully loaded with other processes.

- It is observed that, **execution time is not only depends on instance characteristics**, it also depends on **some external factors** like hardware on which algorithm is running as well as other conditions, and hence it is not a good practice to decide efficiency of an algo i.e. calculation of time complexity on the basis of an execution time and compilation time, hence to do analysis of an algorithms **asymptotic analysis** is preferred.

Data Structures: Introduction

# Asymptotic Analysis : It is a **mathematical way** to calculate time complexity and space complexity of an algorithm **without implementing it in any programming language.**

- In this type of analysis, analysis can be done on the basis of **basic operation** in that algorithm.

e.g. in searching & sorting algorithms **comparison** is the basic operation and hence analysis can be done on the basis of no. of comparisons, in addition of matrices algorithm **addition** is the basic operation and hence on the basis of addition operation analysis can be done.

"Best case time complexity" : if an algo takes **minimum** amount of time to run to completion then it is referred as best case time complexity.

"Worst case time complexity" : if an algo takes **maximum** amount of time to run to completion then it is referred as worst case time complexity.

"Average case time complexity" : if an algo takes **neither minimum nor maximum** amount of time to run to completion then it is referred as an average case time complexity.

Data Structures: Introduction

# Asympotic Notations:

1. Big Omega (Ω) : this notation is used to denote **best case time complexity** – also called as **asymptotic lower bound,** running time of an algorithm cannot be less than its asymptotic lower bound.

2. Big Oh (O) : this notation is used to denote **worst case time complexity** - also called as **asymptotic upper bound,** running time of an algorithm cannot be more than its asymptotic upper bound.

3. Big Theta (θ) : this notation is used to denote an **average case time complexity** - also called as **asymptotic tight bound,** running time of an algorithm cannot be less than its asymptotic lower bound and cannot be more than its asymptotic upper bound i.e. it is **tightly bounded.**

Data Structures: Searching Algorithms

1. Linear Search / Sequential Search:

# Algorithm :

Step-1 : Scan / Accept value of key element from the user which is to be search.

Step-2 : Start traversal of an array and compare value of the key element with each array element sequentially from first element either till match is found or max till last element, **if key is matches with any of array element then return true otherwise return false if key do not matches with any of array element.**

# Pseudocode:

```
Algorithm LinearSearch(A, size, key){
   for( int index = 1 ; index <= size ; index++ ){
   if( arr[ index ] ==  key )
   return true;
   }
   return false;
}
```

Data Structures: Searching Algorithms

Best Case: If key element is found at very first position in only 1 comparison then it is considered as a best case and running time of an algorithm in this case is **O(1)** => hence time complexity of linear search algorithm in base case = **Ω(1).**

Worst Case: If either key element is found at last position or key element does not exists, in this case maximum **n** no. of comparisons takes place, it is considered as a worst case and running time of an algorithm in this case is **O(n)** => hence time complexity of linear search algorithm in worst case = **O(n).**

Average Case: If key element is found at any in between position it is considered as an average case and running time of an algorithm in this case is **O(n/2) => O(n)** => hence time complexity = **θ(n).**

Data Structures: Searching Algorithms

2. Binary Search / Logarithmic Search / Half Interval Search :

- This algorithm follows **<u>divide-and-conquer</u>** approach.

- To apply binary search on an array **prerequisite is that array elements must be in a sorted manner.**

Step-1: Accept value of key element from the user which is to be search.

Step-2: In first iteration, find/calculate **mid position** by the formula **mid=(left+right)/2**, (by means of finding mid position big size array gets divided logically into 2 subarrays, left subarray and right subarray, **left subarray => [left to mid-1]** & **right subarray => [mid+1 to right]**.

Step-3 : Compare value of key element with an element which is at mid position, **if key matches in very first iteration in only one comparison then it is considered as a best case**, if key matches with mid pos element then return true otherwise if key do not matches then we have to go to next iteration, and in next iteration we go to search key either into the left subarray or into the right subarray.

Step-4 : Repeat Step-2 & Step-3 till either key is found or max till subarray is valid, **if subarray is not valid then key is not found in this case return false.**

Data Structures: Searching Algorithms

- As in each iteration 1 comparison takes place and search space is getting reduced by half.

n => n/2 => n/4 => n/8 ......

after iteration-1 => n/2 + 1 => **$T(n) = (n/2^1) + 1$**

after iteration-2 => n/4 + 2 => **$T(n) = (n/2^2) + 2$**

after iteration-3 => n/8 + 3 => **$T(n) = (n/2^3) + 3$**

Lets assume, after **k** iterations => **$T(n) = (n/2^k) + k$ ...... (equation-I)**

let us assume,

=> **$n = 2^k$**

=> $\log n = \log 2^k$ (by taking log on both sides)

=> $\log n = k \log 2$

=> $\log n = k$ (as $\log 2 \sim= 1$)

=> **$k = \log n$**

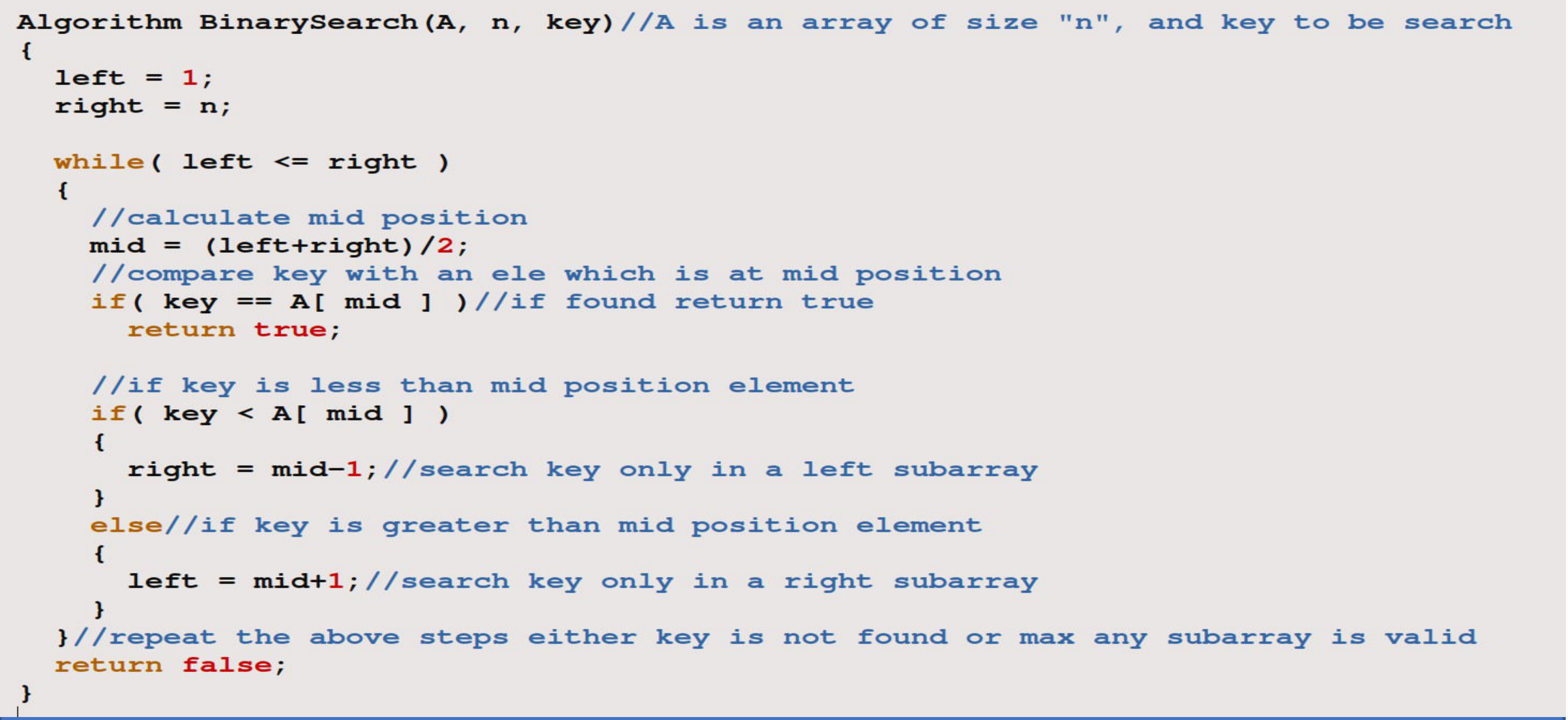
By substituting value of **$n = 2^k$ & $k = \log n$** in **equation-I,** we get

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

=> $T(n) = (2^k / 2^k) + \log n$

=> $T(n) = 1 + \log n$ => $T(n) = O(1 + \log n$ **) => $T(n) = O(\log n)$.**

Data Structures: Searching Algorithms



```
Algorithm BinarySearch(A, n, key)//A is an array of size "n", and key to be search
{
  left = 1;
  right = n;

  while( left <= right )
  {
    //calculate mid position
    mid = (left+right)/2;
    //compare key with an ele which is at mid position
    if( key == A[ mid ] )//if found return true
      return true;

    //if key is less than mid position element
    if( key < A[ mid ] )
    {
      right = mid-1;//search key only in a left subarray
    }
    else//if key is greater than mid position element
    {
      left = mid+1;//search key only in a right subarray
    }
  }//repeat the above steps either key is not found or max any subarray is valid
  return false;
}
```

Data Structures: Searching Algorithms

Best Case: if the key is found in very first iteration at mid position in only 1 comparison OR if key is found at root position it is considered as a best case and running time of an algorithm in this case is O(1) = **Ω(1).**

Worst Case: if either key is not found or key is found at leaf position it is considered as a worst case and running time of an algorithm in this case is O(log n) = **O(log n).**

Average Case: if key is found at non-leaf position it is considered as an average case and running time of an algorithm in this case is O(log n) = **θ(log n).**

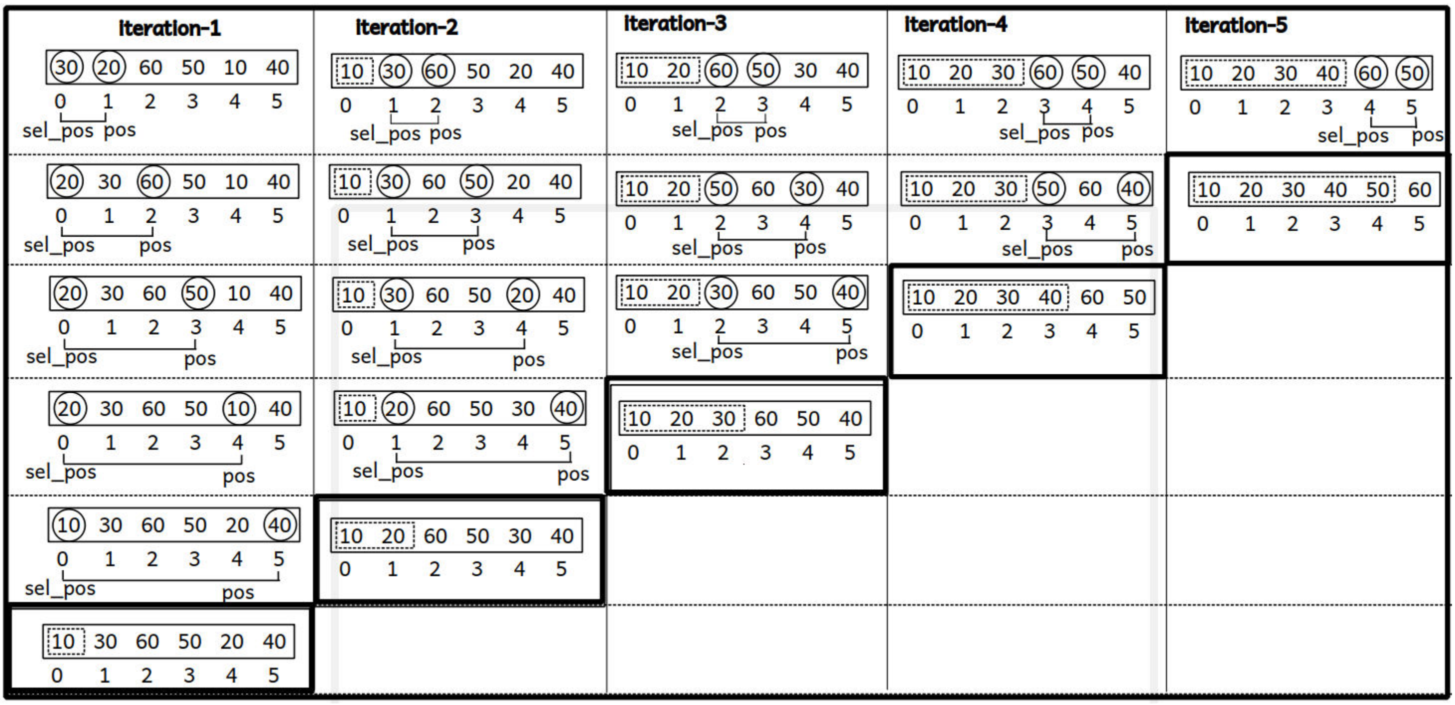
Data Structures: Sorting Algorithms

1. Selection Sort:

- In this algorithm, in first iteration, **first position gets selected** and **element which is at selected position gets compared with all its next position elements sequentially**, **if an element at selected position found greater than any other position element then swapping takes place** and in first iteration **smallest element** gets setteled at first position.

- In the second iteration, **second position gets selected and element which is at selected position gets compared with all its next position elements, if an element selected position found greater than any other position element then swapping takes place** and in second iteration **second smallest element** gets setteled at second position, and so on **in maximum (n-1) no. of iterations all array elements gets arranged in a sorted manner.**

Data Structures: Sorting Algorithms



Data Structures: Sorting Algorithms

Best Case : $\Omega(n^2)$

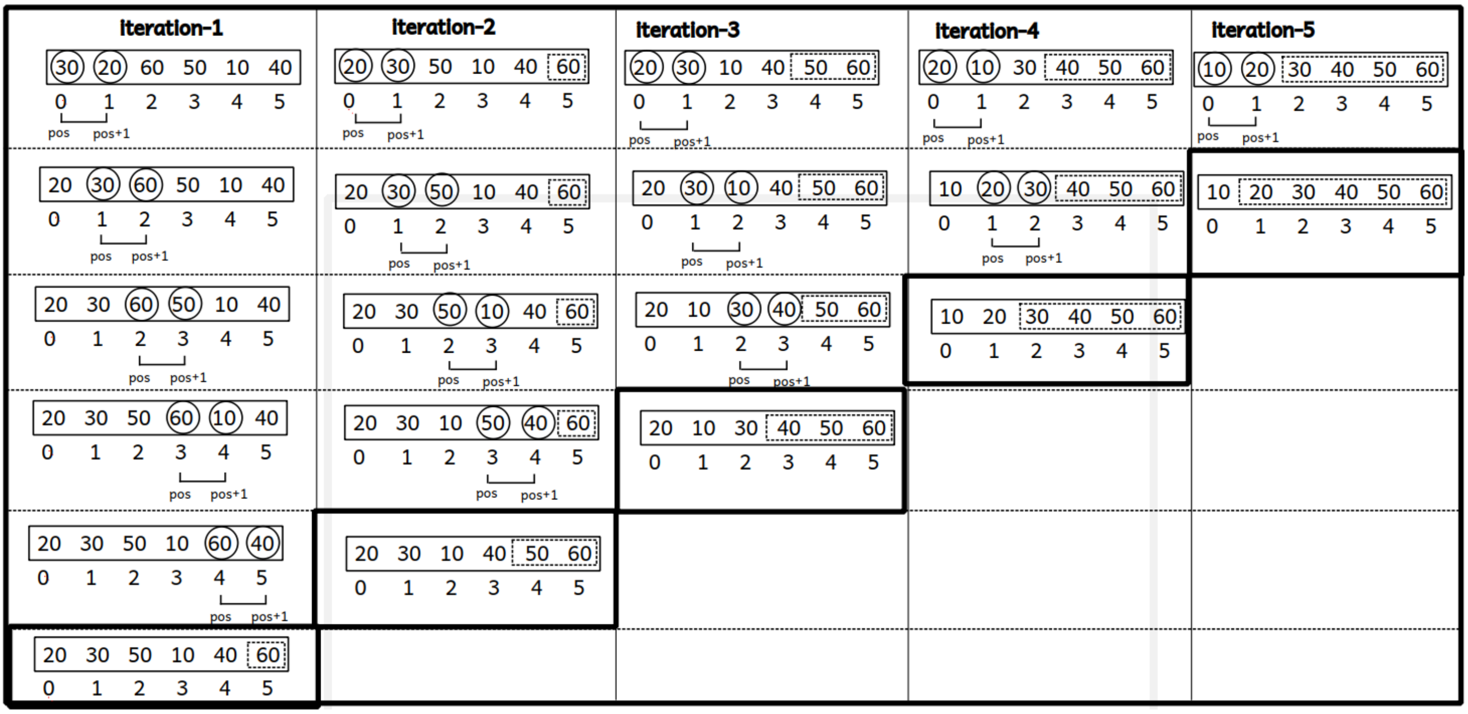
Worst Case : $O(n^2)$

Average Case : $\theta(n^2)$

2. Bubble Sort :

- In this algorithm, **in every iteration elements which are at two consecutive positions gets compared, if they are already in order then no need of swapping between them, but if they are not in order i.e. if prev position element is greater than its next position element then swapping takes place**, and by this logic in first iteration largest element gets setteled at last position, in second iteration second largest element gets setteled at second last position and so on, **in max (n-1) no. of iterations all elements gets arranged in a sorted manner.**

Data Structures: Sorting Algorithms



Data Structures: Sorting Algorithms

Best Case : **Ω(n) –** if array elements are already arranged in a sorted manner.

Worst Case : **O(n^2)**
Average Case : **θ(n^2)**

3. Insertion Sort:

- In this algorithm, in every iteration one element gets selected as a **key element** and key element gets inserted into an array at its appropriate position towards its left hand side elements in a such a way that elements which are at left side are arranged in a sorted manner, and so on, in max **(n-1)** no. of iterations all array elements gets arranged in a sorted manner.
- **This algorithm works efficiently for already sorted input sequence by design** and hence running time of an algorithm is O(n) and it is considered as a best case.

Data Structures: Sorting Algorithms

Best Case : $\Omega(n)$ – if array elements are already arranged in a sorted manner.

Worst Case : $O(n^2)$

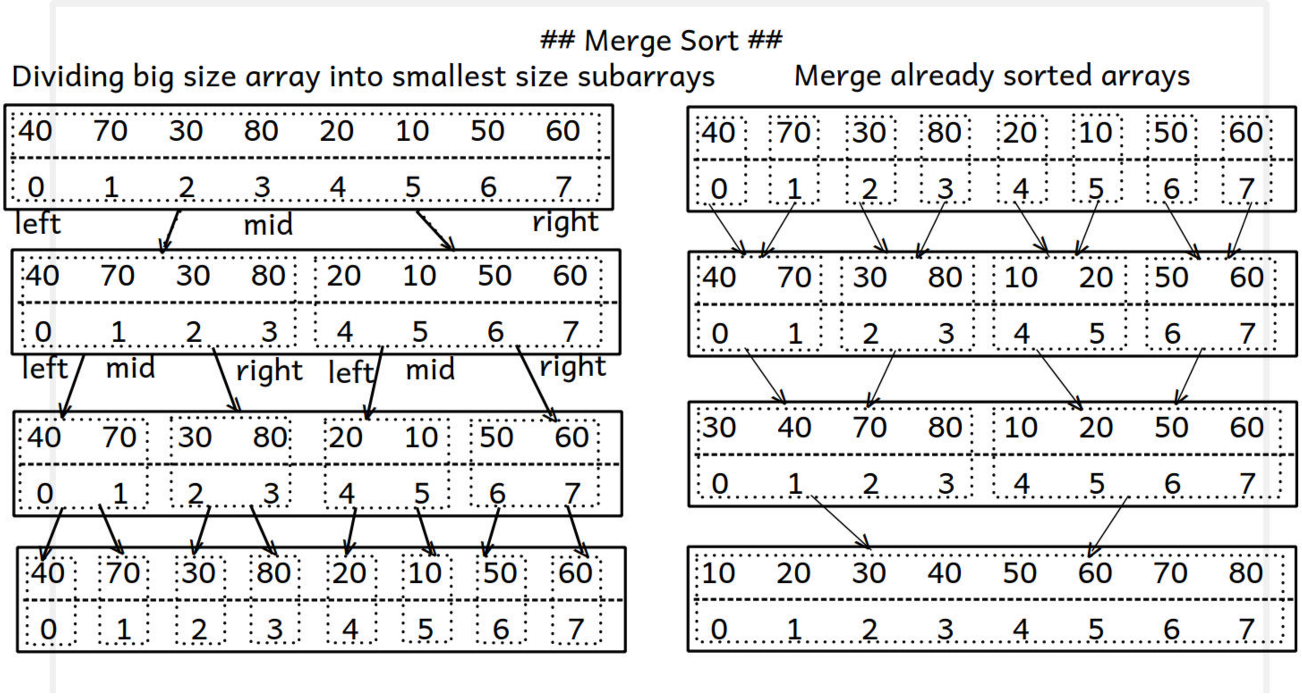
Average Case: $\theta(n^2)$

- Insertion sort algortihm is an efficient algorithm for smaller input size array.

4. Merge Sort:

- This algorithm follows **divide-and-conquer** approach.
- In this algorithm, big size array is divided logically into smallest size (i.e. having size 1) subarrays, as if size of subarray is 1 it is sorted, after dividing array into sorted smallest size subarray's, subarrays gets merged into one array step by step in a sorted manner and finally all array elements gets arranged in a sorted manner.
- This algorithm works fine for **even** as well **odd** input size array.
- This algorithm takes extra space to sort array elements, and hence its space complexity is more.

Data Structures: Sorting Algorithms



Data Structures: Sorting Algorithms

Best Case : Ω(n log n)
Worst Case : O(n log n)
Average Case : θ(n log n)

5. Quick Sort:
- This algorithm follows **divide-and-conquer** approach.
- In this algorithm the basic logic is a **partitioning.**
- Partitioning: in parititioning, pivot element gets selected first (it may be either leftmost or rightmost or middle most element in an array), after selection of pivot element all the elements which are smaller than pivot gets arranged towards as its left as possible and elements which are greater than pivot gets arranged as its right as possible, and big size array is divided into two subarray's, so after first pass pivot element gets settled at its appropriate position, elements which are at left of pivot is referred as **left partition** and elements which are at its right referred as a **right partition.**

Data Structures: Sorting Algorithms

Best Case : Ω(n log n)
Worst Case : O(n^2) – worst case rarely occures
Average Case : θ(n log n)

- Quick sort algortihm is an efficient sorting algorithm for larger input size array.

Data Structures: Linked List

- Limitations of an array data structure:

1. Array is static, i.e. size of an array is fixed, its size cannot be either grow or shrink during runtime.

2. Addition and deletion operations on an array are not efficient as it takes O(n) time, and hence to overcome these two limitations of an Array data structure **Linked List** data structure has been designed.

Linked List: It is a basic/linear data structure, which is a collection/list of logically related similar type of elements in which, an address of first element in a collection/list is stored into a pointer variable referred as a head pointer and each element contains actual data and link to its next element i.e. an address of its next element (as well as an addr of its previous element).

- An element in a Linked List is also called as a **Node.**

- Four types of linked lists are there: **Singly Linear Linked List, Singly Circular Linked List, Doubly Linear Linked List and Doubly Circular Linked List.**

Data Structures: Linked List

- Basically we can perform **addition, deletion, traversal** etc... operations onto the linked list data structure.

- We can add and delete node into and from linked list by three ways:

add node into the linked list **at last position**, **at first position** and **at any specific position**, simillarly we can delete node from linked list which is **at first position**, **at last position** and **at any specific position.**

1. Singly Linear Linked List: It is a type of linked list in which

- head always contains an address of first element, if list is not empty.

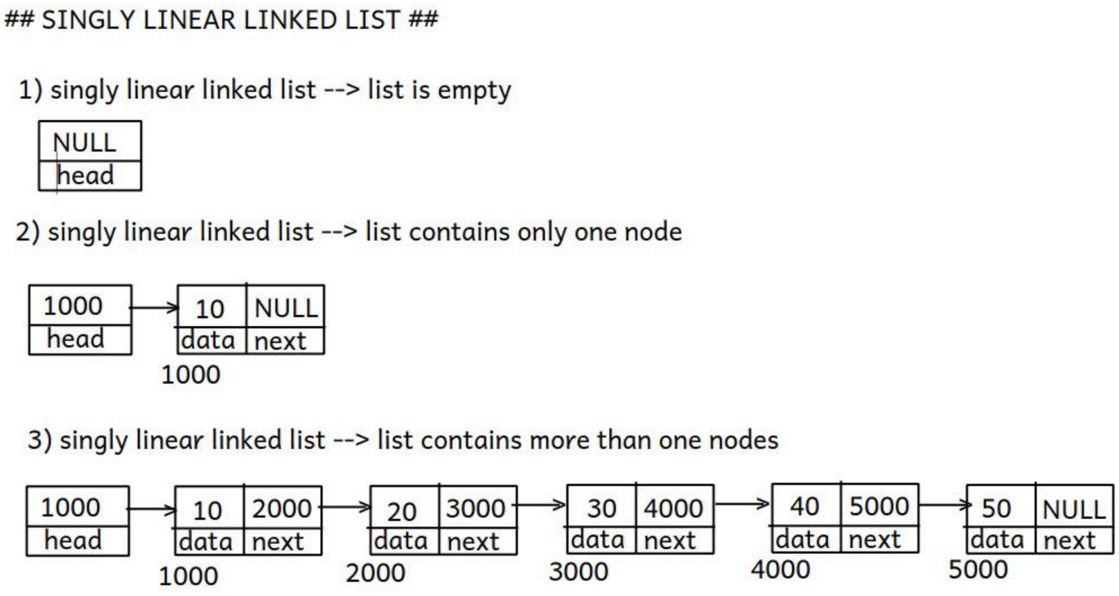
- each node has two parts:

i. data part : it contains actual data of any primitive/non-primitive type.

ii. pointer part (next) : it contains an address of its next element/node.

- last node points to NULL, i.e. next part of last node contains NULL.

Data Structures: Linked List



SINGLY LINEAR LINKED LIST

1) singly linear linked list --> list is empty

2) singly linear linked list --> list contains only one node

3) singly linear linked list --> list contains more than one nodes

Data Structures: Linked List

Limitations of Singly Linear Linked List:

- Add node at last position & delete node at last position operations are not efficient as it takes O(n) time.

- We can start traversal only from first node and can traverse the list only in a forward direction.

- Previous node of any node cannot be accesed from it.

- Any node cannot be revisited – to overcome this limitation Singly Circular Linked List has been designed.

2. Singly Circular Linked List: It is a type of linked list in which

- head always contains an address of first node, if list is not empty.

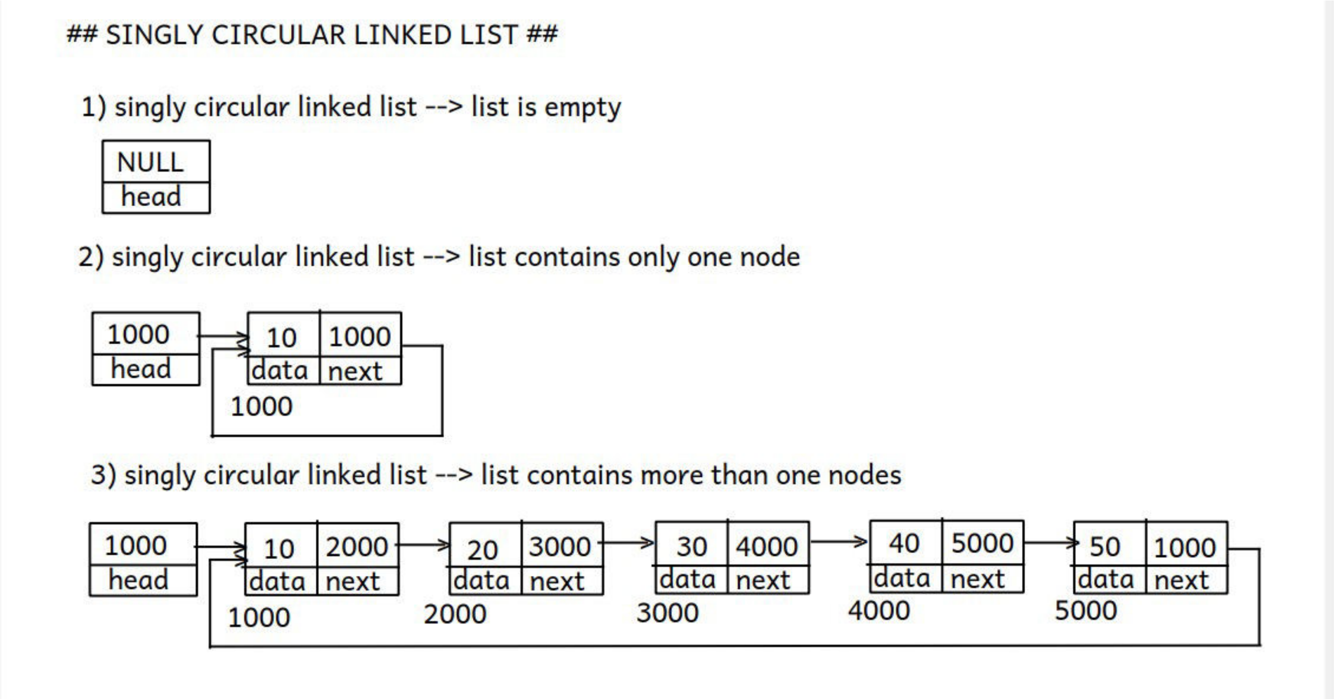
- each node has two parts:

i. data part : contains data of any primitive/non-primitive type.

ii. pointer part(next) : contains an address of its next node.

- last node points to first node, i.e. next part of last node contains an address of first node.

Data Structures: Linked List



SINGLY CIRCULAR LINKED LIST

1) singly circular linked list --> list is empty

2) singly circular linked list --> list contains only one node

3) singly circular linked list --> list contains more than one nodes

Data Structures: Linked List

Limitations of Singly Circular Linked List:

- Add last, delete last & add first, delete first operations are not efficient as it takes O(n) time.

- We can starts traversal only from first node and can traverse the SCLL only in a forward direction.

- **Previous node of any node cannot be accesed from it** – to overcome this limitation Doubly Linear Linked List has been designed.

3. Doubly Linear Linked List: It is a linked list in which

- head always contains an address of first element, if list is not empty.

- each node has three parts:

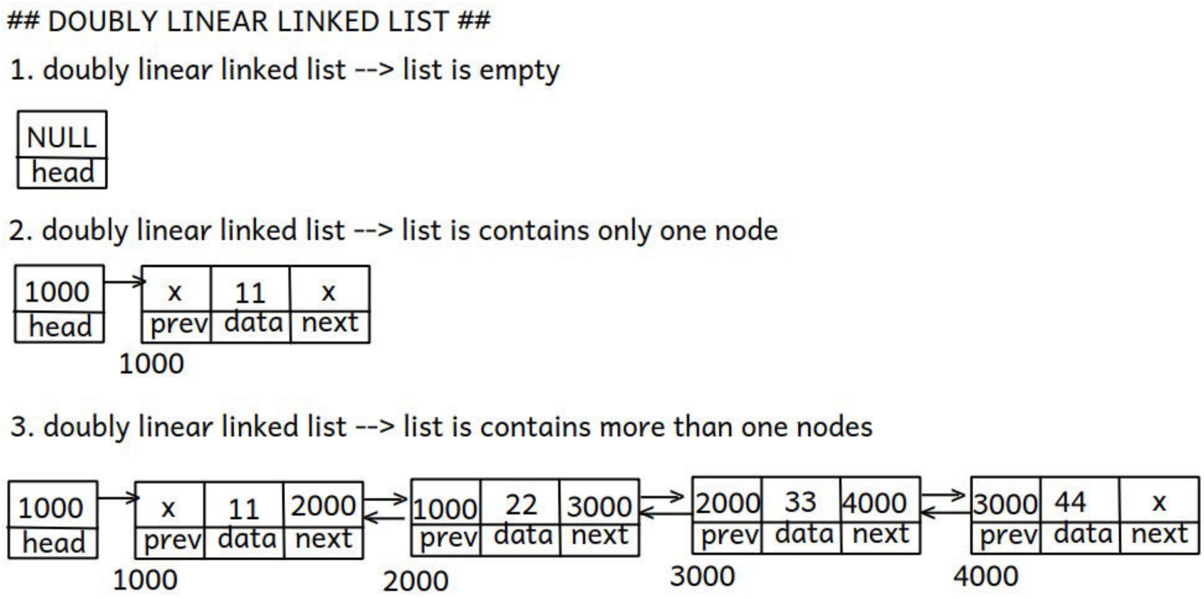
i. data part: contains data of any primitive/non-primitive type.

ii. pointer part(next): contains an address of its next element/node.

iii .pointer part(prev): contains an address of its previous element/node.

- next part of last node & prev part of first node points to NULL.

Data Structures: Linked List



DOUBLY LINEAR LINKED LIST

1. doubly linear linked list --> list is empty

2. doubly linear linked list --> list is contains only one node

3. doubly linear linked list --> list is contains more than one nodes

Data Structures: Linked List

Limitations of Doubly Linear Linked List:

- **Add last and delete last** operations are not efficient as it takes **O(n)** time.

- We can starts traversal only from first node, and hence to overcome these limitations **Doubly Circular Linked List** has been designed.

4. Doubly Circular Linked List: It is a linked list in which

- head always contains an address of first node, if list is not empty.

- each node has three parts:

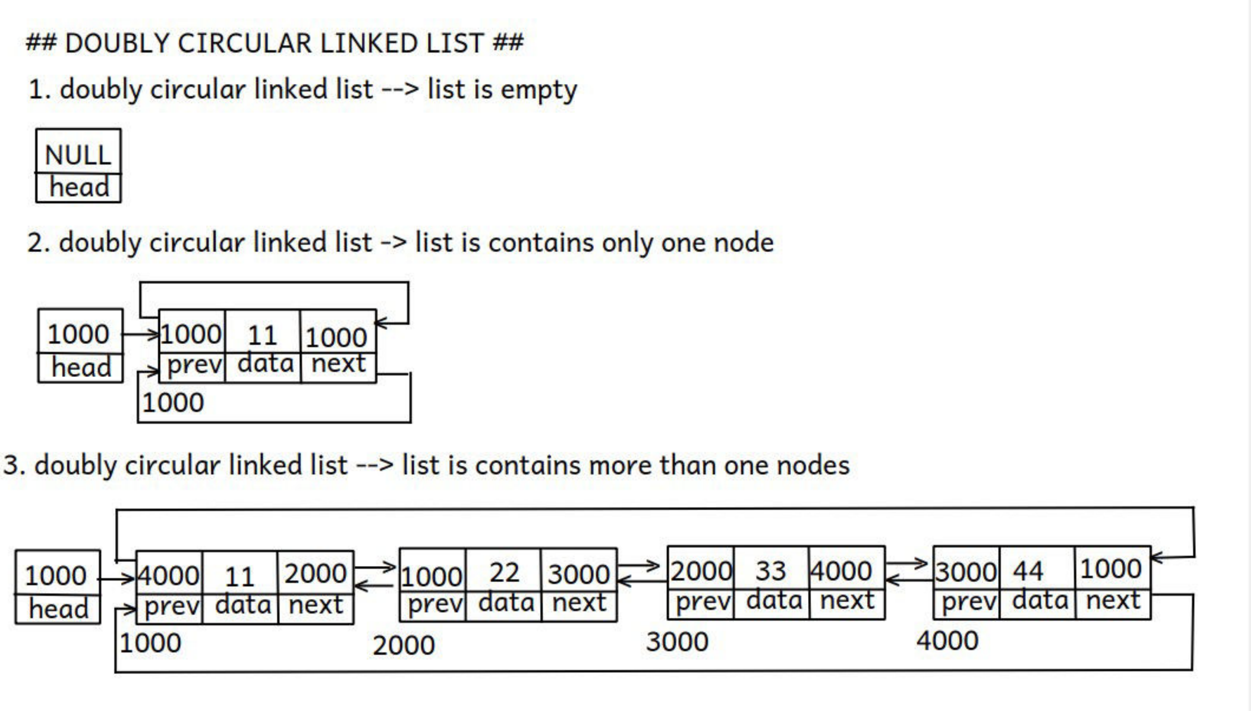
i. data part: contains data of any primitive/non-primitive type.

ii. pointer part(next): contains an address of its next element/node.

iii .pointer part(prev): contains an address of its previous element/node.

- **next part of last node contains an address of first node** & **prev part of first node contains an address of last node.**

Data Structures: Linked List



DOUBLY CIRCULAR LINKED LIST

1. doubly circular linked list --> list is empty

2. doubly circular linked list -> list is contains only one node

3. doubly circular linked list --> list is contains more than one nodes

Data Structures: Linked List

Advantages of Doubly Circular Linked List:

- DCLL can be traverse in forward as well as in a backward direction.

- **Add last, add first, delete last & delete first** operations are efficient as it takes **O(1)** time and are convenient as well.

- Traversal can be start either from first node (i.e. from head) or from last node (from head.prev) in O(1) mtime.

- Any node can be revisited.

- Previous node of any node can be accessed from it

# Array v/s Linked List => Data Structure:

- Array is **static** data structure whereas linked list is **dynamic** data structure.

- Array elements can be accessed by using **random access method** which is **efficient** than **sequential access method** used to access linked list elements.

- **Addition & Deletion operations are efficient** on linked list than on an array.

- Array elements gets stored into the **stack section**, whereas linked list elements gets stored into **heap section.**

- In a linked list **extra space is required to maintain link between elements**, whereas in an array to maintained link between elements is the job of the **compiler**.
- searching operation is faster on an array than on linked list as on linked list we cannot apply binary search.

Data Structures: Linked List

# Applications of linked list in computer science –

- To implementation of stacks and queues
- To implement advanced data structures like tree, hash table, graph
- Dynamic memory allocation : We use linked list of free blocks.
- Maintaining directory of names
- Performing arithmetic operations on long integers
- Manipulation of polynomials by storing constants in the node of linked list
- representing sparse matrices

# Applications of linked list in real world :

- Image viewer : Previous and next images are linked, hence can be accessed by next and previous button.
- Previous and next page in web browser – We can access previous and next url searched in web browser by pressing back and next button since, they are linked as linked list.
- Music Player – Songs in music player are linked to previous and next song. you can play songs either from starting or ending of the list.

Data Structures: Stack

Stack: It is a collection/list of logically related similar type elements into which data elements can be added as well as deleted from only one end referred **top** end.

- In this collection/list, element which was inserted last only can be deleted first, so this list works in **last in first out/first in last out** manner, and hence it is also called as **LIFO list**/FILO list.

- We can perform basic three operations on stack in **O(1)** time: **Push, Pop & Peek.**

1. Push : **to insert/add an element onto the stack at top position**

step1: check stack is not full

step2: increment the value of top by 1

step3: insert an element onto the stack at top position.

2. Pop : to delete/remove an element from the stack which is at top position

step1: check stack is not empty

step2: decrement the value of top by 1.

Data Structures: Stack

3. Peek : to get the value of an element which is at top position without push & pop.

step1: check stack is not empty

step2: return the value of an element which is at top position

Stack Empty : top == -1

Stack Full : top == SIZE-1

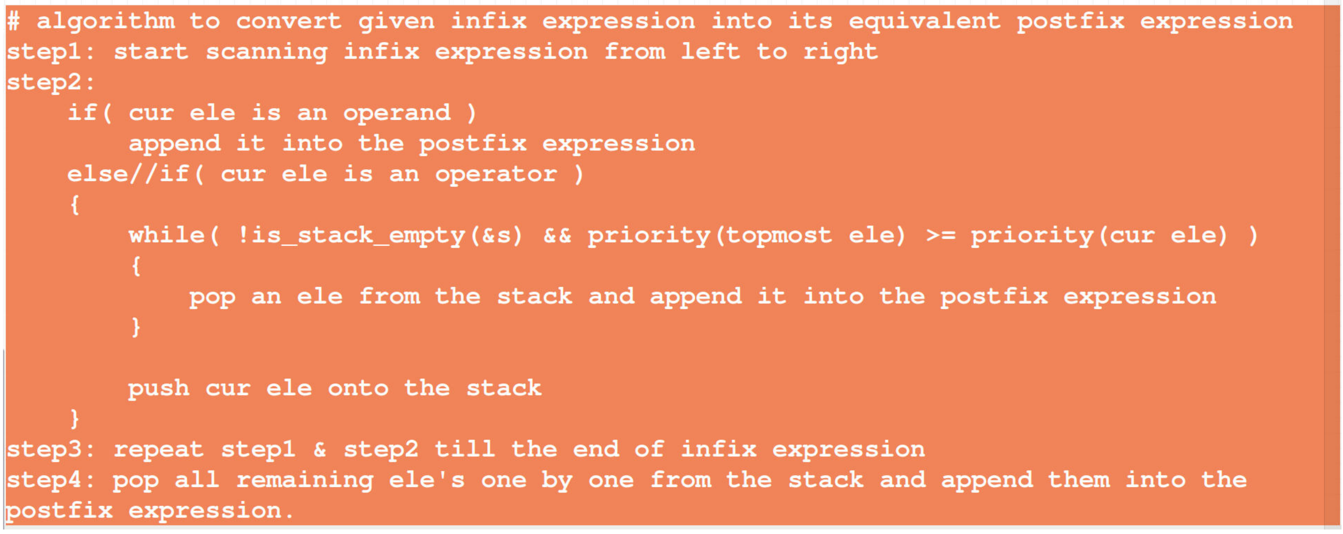
# Applications of Stack:

- Stack is used by an OS to control of flow of an execution of program.
- In recursion internally an OS uses a stack.
- undo & redo functionalities of an OS are implemented by using stack.
- Stack is used to implement advanced data structure algorithms like **DFS: Depth First Search** traversal in tree & graph.
- Stack is used in an algorithms to covert given infix expression into its equivalent postfix and prefix, and for postfix expression evaluation.

Data Structures: Stack

- Algorithm to convert given infix expression into its equivalent postfix expression:

Initially we have, an Infix expression, an empty Postfix expression & empty Stack.



```
# algorithm to convert given infix expression into its equivalent postfix expression
step1: start scanning infix expression from left to right
step2:
    if( cur ele is an operand )
        append it into the postfix expression
    else//if( cur ele is an operator )
    {
        while( !is_stack_empty(&s) && priority(topmost ele) >= priority(cur ele) )
        {
            pop an ele from the stack and append it into the postfix expression
        }

        push cur ele onto the stack
    }
step3: repeat step1 & step2 till the end of infix expression
step4: pop all remaining ele's one by one from the stack and append them into the
postfix expression.
```

Data Structures: Stack

- Algorithm to convert given infix expression into its equivalent prefix expression:

Initially we have, an Infix expression, an empty Prefix expression & empty Stack.

```
# algorithm to convert given infix expression into its equivalent prefix:
step1: start scanning infix expression from right to left
step2:
    if( cur ele is an operand )
        append it into the prefix expression
    else//if( cur ele is an operator )
    {
        while( !is_stack_empty(&s) && priority(topmost ele) > priority(cur ele) )
        {
            pop an ele from the stack and append it into the prefix expression
        }

        push cur ele onto the stack
    }
step3: repeat step1 & step2 till the end of infix expression
step4: pop all remaining ele's one by one from the stack and append them into the
prefix expression.
step5: reverse prefix expression - equivalent prefix expression.
```

Data Structures: Queue

Queue: It is a collection/list of logically related similar type of elements into which elements can be added from one end referred as **rear** end, whereas elements can be deleted from another end referred as a **front** end.

- In this list, element which was inserted first can be deleted first, so this list works in **first in first out** manner, hence this list is also called as **FIFO list/LILO list.**

- Two basic operations can be performed on queue in O(1) time.

1. Enqueue: to insert/push/add an element into the queue from rear end.

2. Dequeue: to delete/remove/pop an element from the queue which is at front end.

- There are different types of queue:

1. Linear Queue (works in a fifo manner)

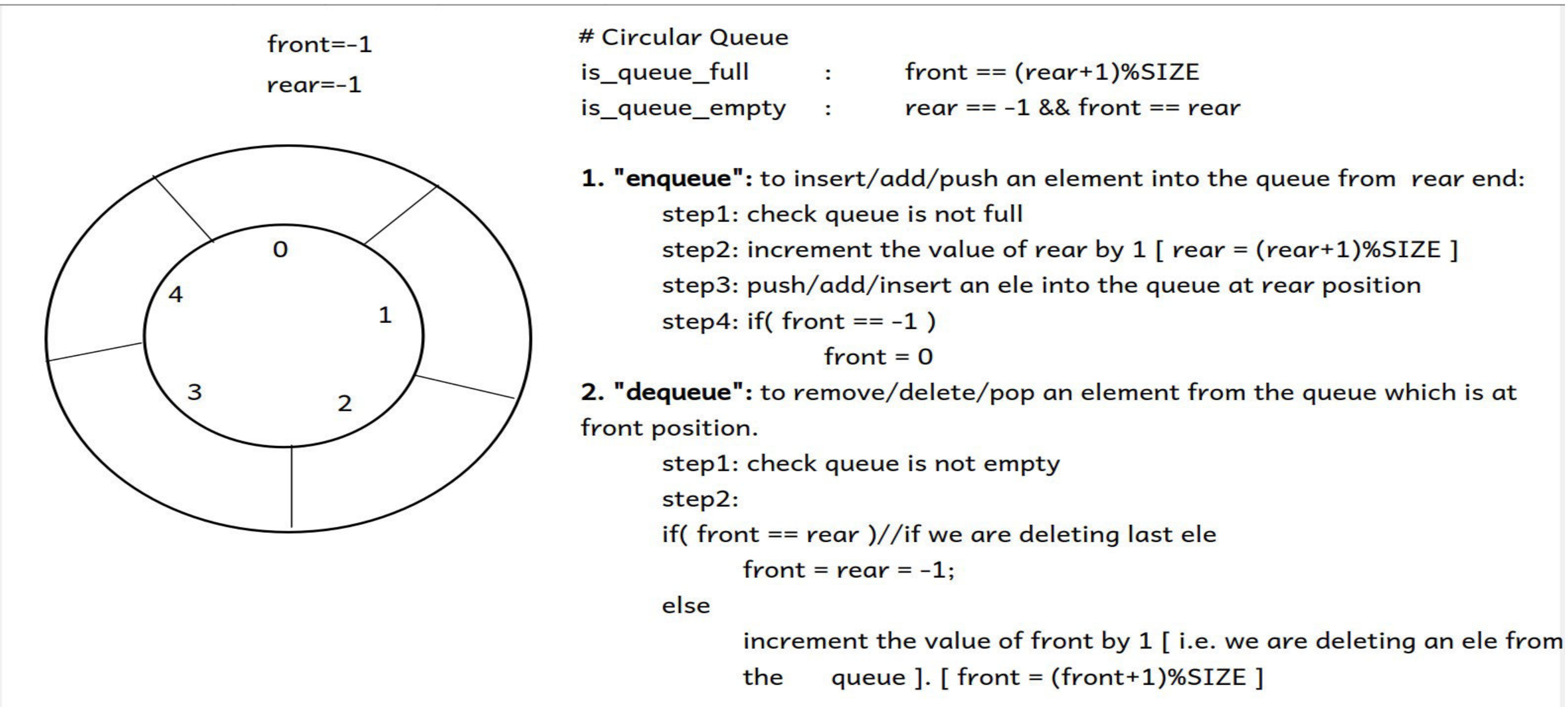
2. Circular Queue (works in a fifo manner)

3. Priority Queue: it is a type of queue in which elements can be inserted from rear end randomly (i.e. without checking priority), whereas an element which is having highest priority can only be deleted first.

- Priority queue can be implemented by using linked list, whereas it can be implemented efficiently by using **binary heap.**

4. Double Ended Queue (deque) : it is a type of queue in which elements can added as well as deleted from both the ends.

Data Structures: Queue



front=-1

rear=-1

```
# Circular Queue
is_queue_full      :      front == (rear+1)%SIZE
is_queue_empty     :      rear == -1 && front == rear

1. "enqueue": to insert/add/push an element into the queue from  rear end:
        step1: check queue is not full
        step2: increment the value of rear by 1 [ rear = (rear+1)%SIZE ]
        step3: push/add/insert an ele into the queue at rear position
        step4: if( front == -1 )
                    front = 0
2. "dequeue": to remove/delete/pop an element from the queue which is at
front position.
        step1: check queue is not empty
        step2:
        if( front == rear )//if we are deleting last ele
                front = rear = -1;
        else
                increment the value of front by 1 [ i.e. we are deleting an ele from
                the      queue ]. [ front = (front+1)%SIZE ]
```

Data Structures: Queue

Applications of Queue:

- Queue is used to implement OS data structures like **job queue, ready queue, message queue, waiting queue** etc...

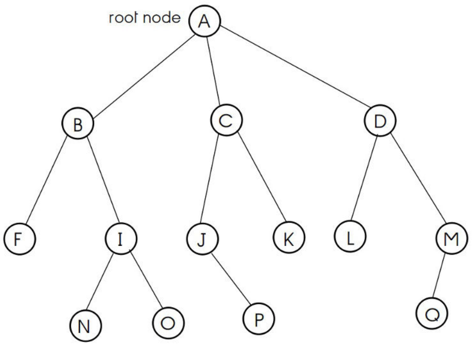
- Queue is used to implement OS algorithms like **FCFS CPU Scheduling, Priority CPU Scheduling, FIFO Page Replacement** etc...

- Queue is used to implenent an advanced data structure algorithms like **BFS: Breadth First Search** Traversal in tree and graph.

- Queue is used in any application/program in which list/collection of elements should works in a **first in first out manner or whereaver it should works according to priority.**

Data Structures: Tree

Tree: It is a **non-linear** / **advanced data structure** which is a **collection of finite no. of logically related similar type of data elements** in which, there is a first specially designated element referred as a **root element**, and remaining all elements are connected to it in a **hierarchical manner**, follows **parent-child relationship**.



Tree: Data Structure

Data Structures: Tree

- **siblings/brothers:** child nodes of same parent are called as siblings.
- **ancestors:** all the nodes which are in the path from root node to that node.
- **descedents:** all the nodes which can be accessible from that node.
- **degree of a node** = no. of child nodes having that node
- **degree of a tree** = max degree of any node in a given tree
- **leaf node/external node/terminal node:** node which is not having any child node OR node having degree 0.
- **non-leaf node/internal node/non-terminal node:** node which is having any no. of child node/s OR node having non-zero degree.
- **level of a node** = level of its parent node + 1
- **level of a tree** = max level of any node in a given tree (by assuming level of root node is at level 0).
- **depth of a tree** = max level of any node in a given tree.
- as tree data structure can grow upto any level and any node can have any number of child nodes, operations on it becomes unefficient, so restrictions can be applied on it to achieve efficiency and hence there are diefferent types of tree.

Data Structures: Tree

- **Binary tree:** it is a tree in which each node can have max 2 number of child nodes, i.e. each node can have either 0 OR 1 OR 2 number of child nodes.

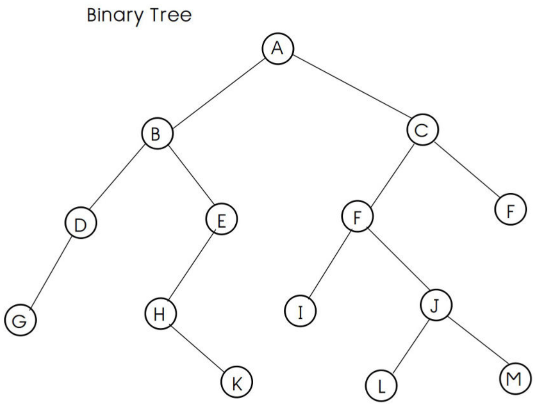
OR

Binary tree: it is a set of finite number of elements having three subsets:

1. root element

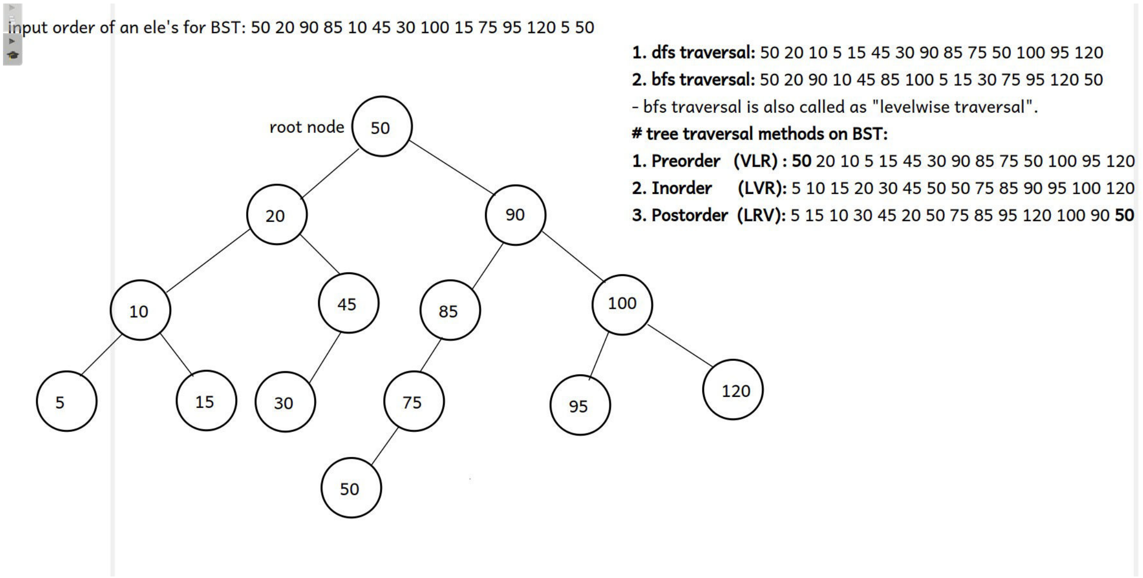
2. left subtree (may be empty)

3. right subtree (may be empty)



Data Structures: Tree

- **Binary Search Tree(BST):** it is a **binary tree** in which left child is always smaller than its parent and right child is always greater than or equal to its parent.


input order of an ele's for BST: 50 20 90 85 10 45 30 100 15 75 95 120 5 50

1. dfs traversal: 50 20 10 5 15 45 30 90 85 75 50 100 95 120

2. bfs traversal: 50 20 90 10 45 85 100 5 15 30 75 95 120 50

- bfs traversal is also called as "levelwise traversal".

# tree traversal methods on BST:

1. Preorder (VLR) : 50 20 10 5 15 45 30 90 85 75 50 100 95 120

2. Inorder (LVR): 5 10 15 20 30 45 50 50 75 85 90 95 100 120

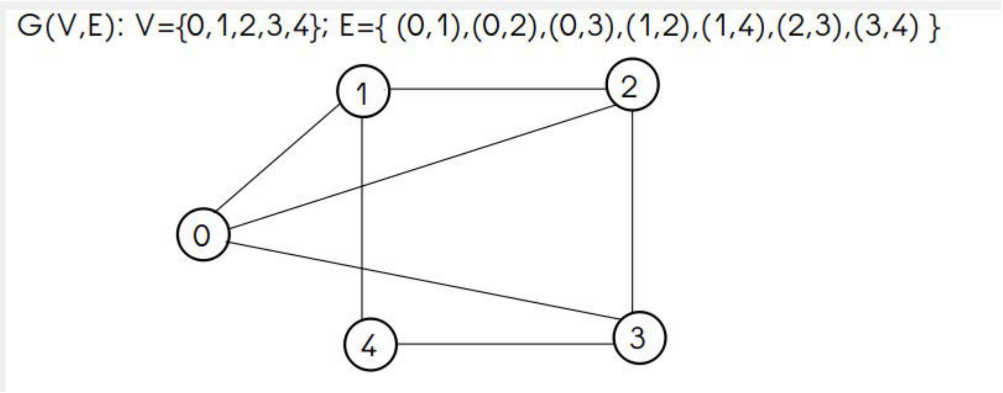
3. Postorder (LRV): 5 15 10 30 45 20 50 75 85 95 120 100 90 **50**

Data Structures: Graph

Graph: It is **non-linear**, **advanced** data structure, which is a collection of logically related similar and disimmilar type of elements which contains:

- set of finite no. of elements referred as a **vertices**, also called as **nodes**, and

- set of finite no. of ordered/unordered pairs of vertices referred as an **edges**, also called as an **arcs**, whereas it may carries weight/cost/value (cost/weight/value may be -ve).

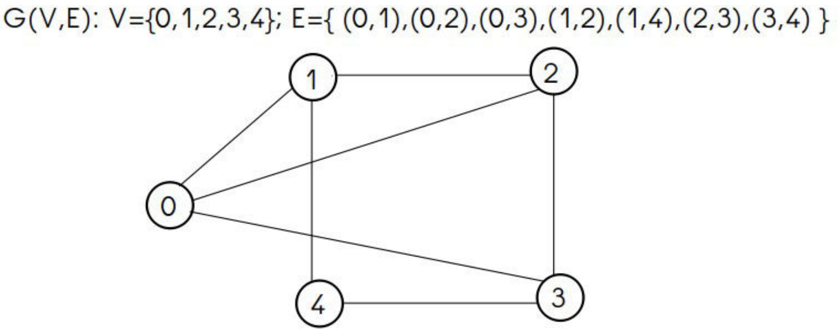


G(V,E): V={0,1,2,3,4}; E={ (0,1),(0,2),(0,3),(1,2),(1,4),(2,3),(3,4) }

Data Structures: Graph

Graph: Graph is a **non-linear / an advanced data structure**, defined as set of vertices and edges.

- **Vertices** (or **Nodes**) holds the data.
- **Edges** (or **Arcs**) represents relation between vertices.
  - Edges may have direction and/or value assigned to them called as **weight or cost.**
- **Applications of Graph:**
  - Electronic circuits
  - Social media apps
  - Communication network
  - Road network
  - Flight/Train/Bus services
  - Bio-logical & Chemical experiments
  - Deep Learning (Neural network, Tensor flow)
  - Graph databases (Neo4j)



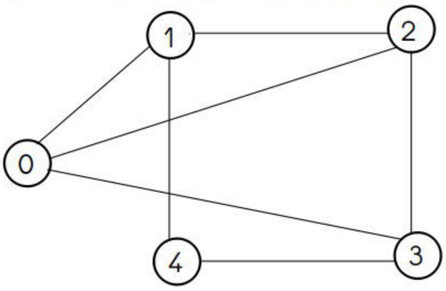
G(V,E): V={0,1,2,3,4}; E={ (0,1),(0,2),(0,3),(1,2),(1,4),(2,3),(3,4) }

Data Structures: Graph

Graph: Graph is a **non-linear / an advanced data structure**, defined as set of vertices and edges.

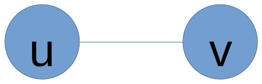
- **Vertices** (or **Nodes**) holds the data.
- **Edges** (or **Arcs**) represents relation between vertices.
  - Edges may have direction and/or value assigned to them called as **weight or cost.**
- **Applications of Graph:**
  - Electronic circuits
  - Social media apps
  - Communication network
  - Road network
  - Flight/Train/Bus services
  - Bio-logical & Chemical experiments
  - Deep Learning (Neural network, Tensor flow)
  - Graph databases (Neo4j)

G(V,E): V={0,1,2,3,4}; E={ (0,1),(0,2),(0,3),(1,2),(1,4),(2,3),(3,4) }

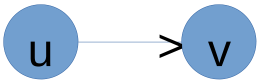


Data Structures: Graph

- If there exists a direct edge between two vertices then those vertices are referred as an **adjacent vertices** otherwise **non-adjacent**.



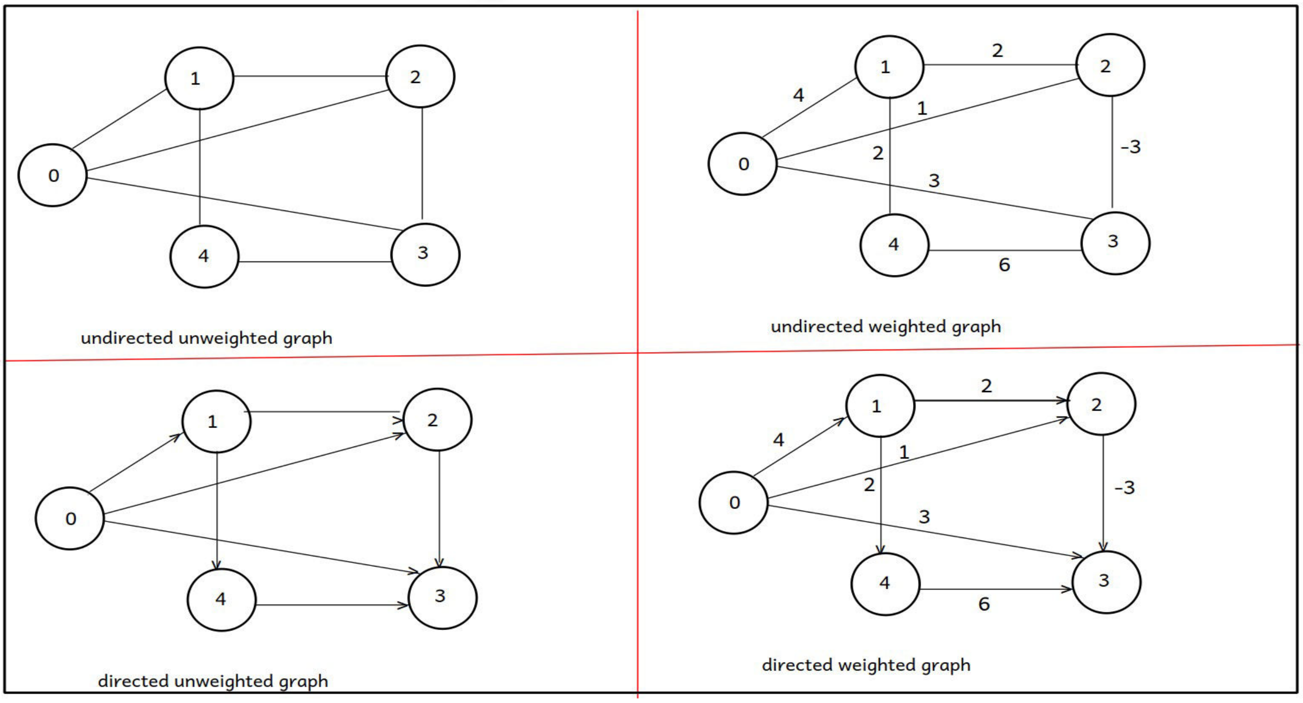
(u, v) == (v, u)



(u, v) != (v, u)

- If we can represent any edge either (u,v) OR (v,u) then it is referred as **unordered pair of vertices i.e. undirected edge.**
- **e.g. (u,v) == (v,u) => unordered pair of vertices => undirected edge => graph which contanis undirected edges referred as undirected graph.**
- If we cannot represent any edge either (u,v) OR (v,u) then it is referred as an **unordered pair of vertices** i.e. directed edge.
- **<u, v> != <v, u> => ordered pair of vertices => directed edge -> graph which contains set of directed edges referred as directed graph (di-graph).**

Data Structures: Graph

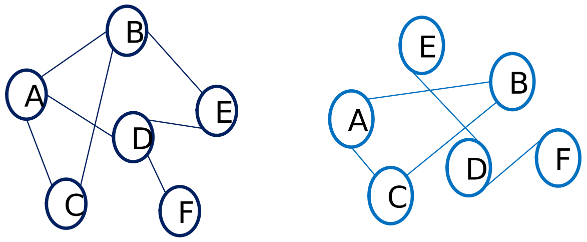

1
2
0
4
3
undirected unweighted graph
2
1
2
4
1
-3
0
2
3
4
3
6
undirected weighted graph
1
2
0
4
3
directed unweighted graph
2
1
2
4
1
-3
0
2
3
4
3
6
directed weighted graph

Data Structures: Graph

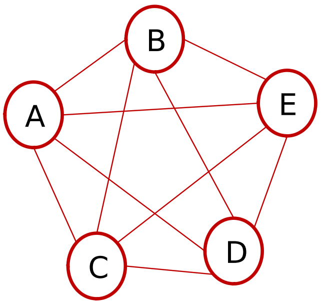
- **Path:** Path is set of edges connecting two vertices.
- **Cycle:** in a given graph, if in any path starting vertex and end vertex are same, such a path is called as a cycle.
- **Loop:** if there is an edge from any vertex to that vertex itself, such edge is called as a loop. Loop is the smallest cycle.
- **Connected Vertices:** if there exists a direct/indirect path between two vertices then those two vertices are referred as a connected vertices otherwise not-connected.
  - Adjacent vertices are always connected but vice-versa is not true.

Graph

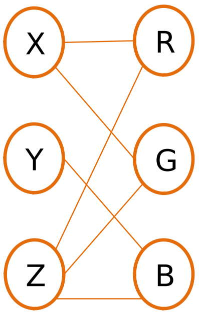
- **Connected graph:**
  - From each vertex some path exists for every other vertex.
  - Can traverse the entire graph starting from any vertex.



- **Complete graph:**
  - Each vertex of a graph is adjacent to every other vertex.
  - Un-directed graph: Number of edges = v (v-1) / 2
  - Directed graph: Number of edges = v (v-1)

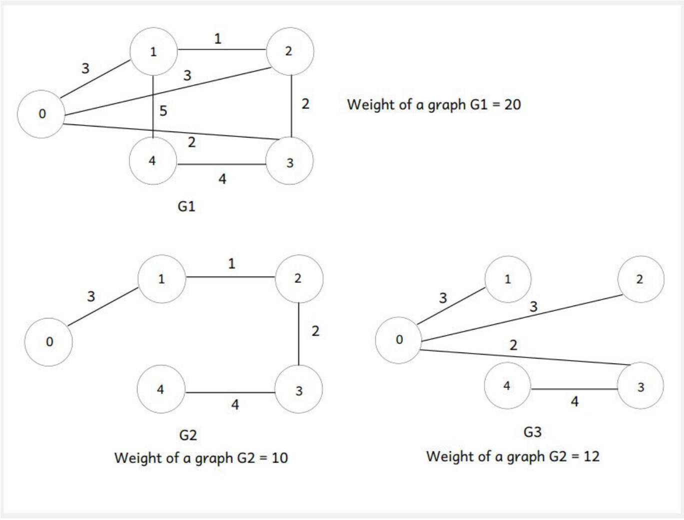


- **Bi-partite graph (bigraph):**
  - Vertices can be divided in two disjoint and indepedent sets.
  - Vertices in first set are connected to vertices in second set.
  - Vertices in a set are not directly connected (not adjacent) to each other.



Data Structures: Graph

Spanning Tree:


0
1
2
3
4
3
1
3
5
2
2
4
Weight of a graph G1 = 20
G1
G2
Weight of a graph G2 = 10
G3
Weight of a graph G2 = 12

- **Weight of a graph =** sum of weights of all its edges.
- **Spanning Tree:**
  - - Connected subgraph of a graph.
  - - Includes all V vertices and V-1 edges.
  - - Do not contain cycle.
  - - A graph may have multiple spanning trees.
- **Minimum Spanning Tree:** Spanning tree of a given graph having minimum weight.
  - - Used to minimize resources/cost.
  - **MST Algorithms**:
    - **Prim’s Algorithm => O(E log V)**
    - **Kruskal’s Algorithm => O(E log V)**

Data Structures: Graph

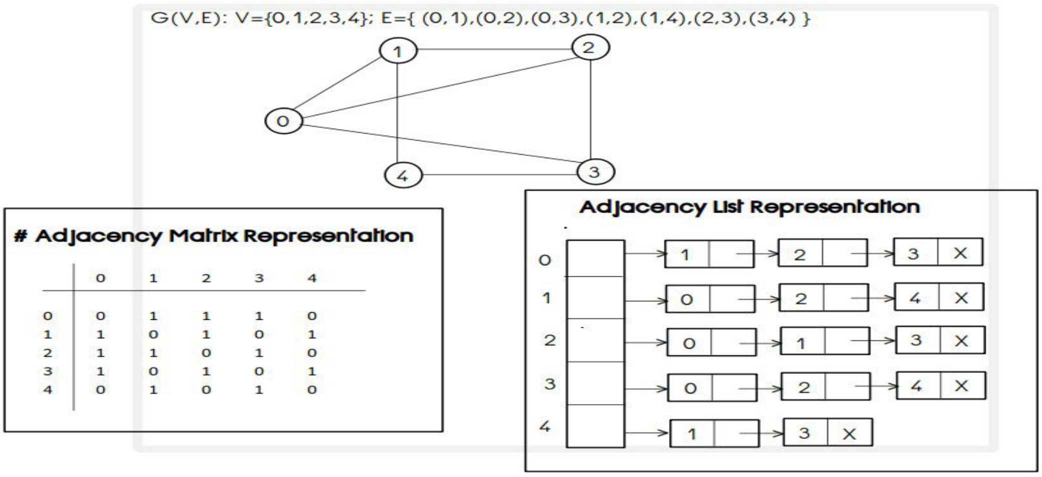
- **Graph Traversal Algorithms:**
  - Used to traverse all vertices in the graph.
  - DFS Traversal (using Stack) and BFS Traversal (by using Queue)
- **Shortest Path Algorithms:**
  - Single source SPT algorithm used to find minimum distance from the given source vertex to all other vertices.
  - **Dijsktra's Algorithm** (Doesn't work for -ve weight edges) => **O(V log V)**.
  - **Bellman Ford Algorithm** => **O(VE)**.
- **All pair Shortest Path Algorithm:**
  - To find minimum distance from each vertex to all other vertices.
  - **Floyd Warshall Algorithm => O(V^3)**
  - **Johnson's Algorithm => O(V^2 log V + VE)**

Data Structures: Graph

There are two graph representation methods:

1. Adjacency Matrix Representation (2-D Array)

2. Adjacency List Representation (Array of Linked Lists)



G(V,E): V={0,1,2,3,4}; E={ (0,1),(0,2),(0,3),(1,2),(1,4),(2,3),(3,4) }

Adjacency Matrix Representation

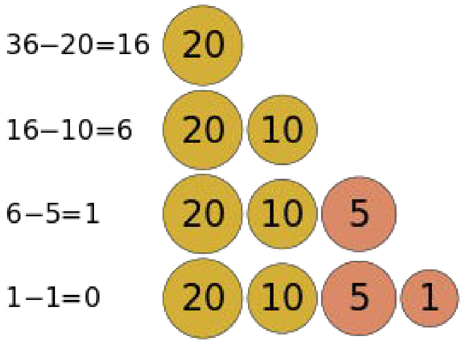
	0	1	2	3	4
0	0	1	1	1	0
1	1	0	1	0	1
2	1	1	0	1	0
3	1	0	1	0	1
4	0	1	0	1	0

Adjacency List Representation

0 → 1 → 2 → 3 X
1 → 0 → 2 → 4 X
2 → 0 → 1 → 3 X
3 → 0 → 2 → 4 X
4 → 1 → 3 X

Problem solving technique: Greedy approach

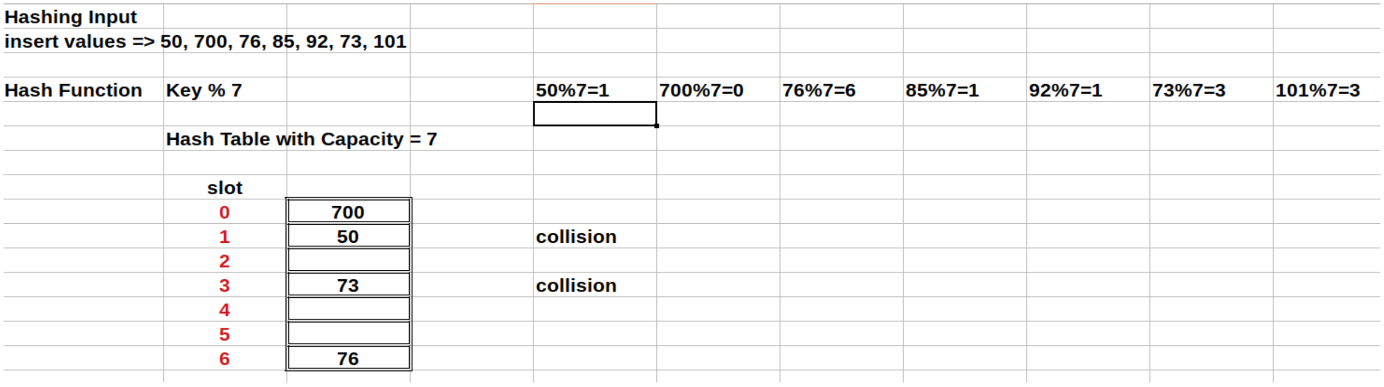
- A greedy algorithm is any algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage with the intent of finding a global optimum.
- We can make choice that seems best at the moment and then solve the sub-problems that arise later.
- The choice made by a greedy algorithm may depend on choices made so far, but not on future choices or all the solutions to the sub-problem.
- It iteratively makes one greedy choice after another, reducing each given problem into a smaller one.
- A greedy algorithm never reconsiders its choices.
- A greedy strategy may not always produce an optimal solution.



- Greedy algorithm decides minimum number of coins to give while making change.

Data Structures: Hash Table

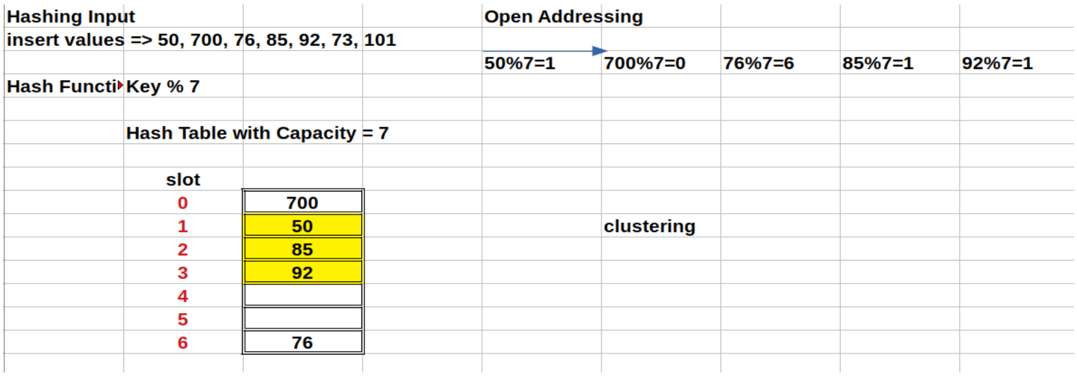
- **HashTable:** Hash table is an **associative** data structure in which data is stored in **key-value pairs** so that for the given key value can be searched in minimal possible time. Ideal time complexity is **O(1)**. Internally it is an array (table) where values can accessed by the index (slot) calculated from the key.
- **Hash Function:** It is mathematical function of the key that yields slot of the hash table where key-value is stored. Simplest example is: **f(k) = k % size.**
- **Collision:** There is possibility that two keys result in same slot, this is called collision and must be handled using some **collision handling technique.** It can be handled by either Open addressing or Chaining.



Hashing Input										
insert values => 50, 700, 76, 85, 92, 73, 101										
Hash Function	Key % 7			50%7=1	700%7=0	76%7=6	85%7=1	92%7=1	73%7=3	101%7=3
	Hash Table with Capacity = 7									
	slot									
	0	700								
	1	50		collision						
	2									
	3	73		collision						
	4									
	5									
	6	76								

Data Structures: Hash Table

- **Open Addressing:**
  - All key-value pairs are stored in the hash table itself.
  - If key (to find) is not matching with the key in the slot calculated by hash function, it is probed in next possible slot using one of the following.
    - **Linear Probing:** In linear probing, if collision occurs next free slot will be searched/probed linearly.
    - **Quadratic Probing:** In quadratic probing, if collision occurs next free slot will be searched/probed quadratically.
- **Double Hashing:** In double hashing, if collision occurs next free slot will be searched/probed by using another hash function, so two hash functions can be use to find next/probe next free slot.



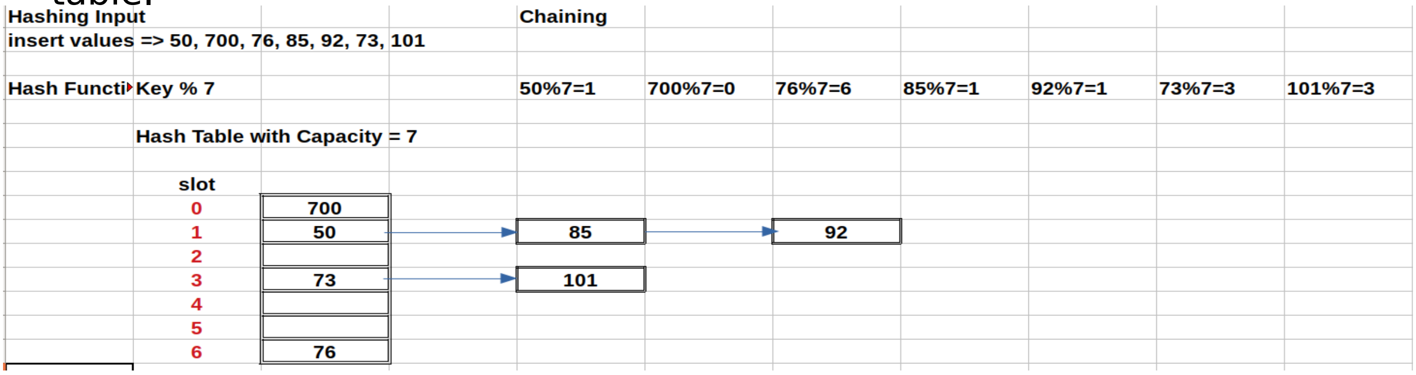
Hashing Input				Open Addressing				
insert values => 50, 700, 76, 85, 92, 73, 101								
				50%7=1	700%7=0	76%7=6	85%7=1	92%7=1
Hash Functi	Key % 7							
	Hash Table with Capacity = 7							
	slot							
	0	700						
	1	50			clustering			
	2	85						
	3	92						
	4							
	5							
	6	76						

Data Structures: Hash Table

- **Load Factor = n / m**
  - n = Number of key-value pairs to be inserted in the hash table
  - m = Number of slots in the hash table
  - If n < m, then load factor < 1
  - If n = m, then load factor = 1
  - If n > m, then load factor > 1
- **Limitations of Open Addressing**
  - Open addressing requires more computation.
  - Cannot be used if load factor is greater than 1 (i.e. number of pairs are more than number of slots in the table).

Data Structures: Hash Table

- **Chaining:**
  - Another collision handling technique.
  - Each slot of hash table holds a collection of key-values for which hash value of keys are same.
  - This collection in each slot is also referred as **bucket.**
  - Chaining is simple to implement, but requires additional memory outside the table.



Hashing Input				Chaining						
insert values => 50, 700, 76, 85, 92, 73, 101										
Hash Functi	Key % 7			50%7=1	700%7=0	76%7=6	85%7=1	92%7=1	73%7=3	101%7=3
	Hash Table with Capacity = 7									
	slot									
	0	700								
	1	50		85		92				
	2									
	3	73		101						
	4									
	5									
	6	76								

# Module Name : Algorithms & Data Structures Using Java.

===

# DAY-01:

# Introduction to data structures?

Q. Why there is a need of data structures?

- if we want to store marks of 100 students

```
int m1, m2, m3, m4, ....., m100;//400 bytes if
sizeof(int) = 4 bytes

int marks[ 100 ];//400 bytes - if sizeof(int) = 4 bytes
```

- we want to store rollno, marks & name

```
rollno   : int
marks    : float
name     : char [] / String / String

struct student
{
    int rollno;
    char name[ 32 ];
    float marks;
};

struct student s1;

class Employee
{
    //data members
    int empid;
    String empName;
    float salary;
    //member functions/methods
};

Employee e1;
Employee e2;
```

=> to learn data structures is not learn any programming language, it is a programming concept i.e. it is nothing but to learn algorithms, and algorithms learned in data structures can be implemented by using any programming language.

# algorithm to do sum of array elements => end user (human being)
step-1: initially take sum as 0.
step-2: traverse an array and add each array element into sum sequentially
from first element max till last element.
step-3: return final sum.

# pseudocode => programmer user

```
Algorithm ArraySum(A, n)//whereas A is an array of size "n"
{
    sum = 0;
    for( index = 1 ; index <= n ; index++ ){
        sum += A[ index ];
    }
    return sum;
}
```

# program => compiler => machine

```
int arraySum(int [] arr, int size){
    int sum = 0;
    for( int index = 0 ; index < size ; index++ ){
        sum += arr[ index ];
    }
    return sum;
}
```

Bank Manager => Algorithm => Project Manager => Software Architect
=> Pseudocode => Developers => Program => Machine

Problem : **"Searching"** => to search / to find an element (can be referred as a key element) into a collection/list of elements.

1. Linear Search
2. Binary Search

Problem : **"Sorting"** => to arrange data elements in a collection / list of elements either in an ascending order or in a descending order.

1. Selection Sort
2. Bubble Sort
3. Insertion Sort
4. Merge Sort
5. Quick Sort

etc....

- when one problem has many solutions we need to go for an efficient solution.

City-1:
City-2:

multiple paths exists => optimum path
distance, condition, traffic situation ....

- to traverse an array => to visit each array element sequentially from first element max till last element.

- there are two types of algorithms:

1. iterative approach (non-recursive)
2. recursive approach

- recursion
- recursive function
- recursive function call
- tail-recursive function
- non-tail recursive function

```
Class Employee
{
    int empid;
    String name;
    float salary;
};
```

- object e1 is an instance of Employee class

```
Employee e1(1, "sachin", 1111.11);
Employee e2(2, "nilesh", 2222.22);
Employee e3(3, "amit", 3333.33);
```

# Space Complexity:

for size of an array = 5 => index = 0 to 5 => only 1 mem copy of index = 1 unit
for size of an array = 10 => index = 0 to 10 => only 1 mem copy of index = 1 unit

.

.

for size of an array = n => index = 0 to n => only 1 mem copy of index = 1 unit

for any input size array we require only 1 memory copy of index var =>
simple var

+ sum:

for size of an array = 5 => sum => only 1 mem copy of sum = 1 unit
for size of an array = 10 => sum => only 1 mem copy of sum = 1 unit

.

.

for any input size array we require only 1 memory copy of sum var => simple var

+ n = input size of an array -> instance characteristics of an algo

for size of an array = 5 => if n = 5 => 5 memory copies required to store 5 ele's => 5 units

for size of an array = 10 => if n = 10 => 10 memory copies required to store 10 ele's => 10 units for size of an array = 20 => if n = 20 => 20 memory copies required to store 20 ele's => 20 units

for size of an array = 100 => if n = 100 => 100 memory copies required to store 100 ele's => 100 units

size

- for any input size array no. of instructions inside an algo remains fixed i.e. space required for instructions i.e. code space for any size array will going to remains fixed or constant.

```
int sum( int n1, int n2 )//n1 & n2 are formal params
{
    int res;//local var
    res = n1 + n2;
    return res;

}
```

- When any function completes its excution control goes back into its calling function as an addr of next instruction to be executed in its calling function gets stored into FAR of that function as a **"return addr"**.

FAR contains:
local vars
formal params
return addr => addr of next instruction to be executed in its calling function
old frame pointer => an addr of its prev stack frame/FAR.
etc...

Linear Search:

```
for( index = 1 ; index <= n ; index++ ){
    if( key  == arr[ index ] )
        return true;
}

return false;
```

- In Linear Search best case "if key found at very first position"

for size of an array = 10, no. of comparisons = 1
for size of an array = 20, no. of comparisons = 1
for size of an array = 50, no. of comparisons = 1
.
.

for any input size array => no. of comparisons = 1
Running Time => O(1)

- In Linear Search worst case "if either key found at last first position or key does not exists"

for size of an array = 10, no. of comparisons = 10
for size of an array = 20, no. of comparisons = 20
for size of an array = 50, no. of comparisons = 50
.
.

for size of an array = n, no. of comparisons = n
Running Time => O(n)

Lab Work => Implement Linear Search => by non-rec as well rec way

# DAY-02:

- linear search
- binary search
- comparison between searching algo
- sorting algorithms:
basic sorting algo's : selection, bubble & insertion

assumption-1:
if running time of an algo is having any additive / substractive / divisive / multiplicative constant then it can be neglected.

e.g.
O(n + 3) => O(n)
O(n - 2) => O(n)
O(n / 5) => O(n)
O(6 * n) => O(n)

# Binary Search:

by means of calculating mid position big size array gets divided logically into two subarray's:
left subarray => left to mid-1
right subarray => mid+1 to right

for left subarray => value of left remains same, whereas value of right = mid-1

for right subarray => value of right remains same, whereas value of left = mid+1

best case occurs in binary search if key is found in very first iteration in only 1 comparison.
if size of an array = 10, no. Of comparisons = 1
if size of an array = 20, no. Of comparisons = 1
.
.
for any input size array, no. Of comparisons = 1
=> running time => O(1)

- in this algo, **in every iteration 1 comparison takes place and search space gets divided by half** i.e. array gets divided logically into two subarray’s and in next iteration we will search key either into left subarray or into right subarray.

# Substitution method to calculate time complexity of binary search:

Size of an array = 1000
1000 = n
500 = n/2
250 = n/4
125 = n/8
.
.
.

n/2 / 2 => n / 4
n/4 / 2 => n / 8
.
.

for iteration-1 input size of an array => n
after iteration-1: n/2 + 1 => $n / 2^1 + 1$
after iteration-2: n/4 + 2 => $n / 2^2 + 2$
after iteration-3: n/8 + 3 => $n / 2^3 + 3$
.
.
.
after iteration-k: $n / 2^k + k$ .... eq-I

T(n) = $n / 2^k + k$ .... eq-I

lets assume, n = 2^k
n = 2^k
log n = log 2^k .... **[by taking log on both sides]**
log n = k log 2

log n = k .... **[as log 2 ~= 1]**
k = log n

put value of n = 2^k and k = log n in eq-I, we get
T(n) = n / 2^k + k
=> T(n) = 2^k / 2^k + log n
=> T(n) = 1 + log n
=> T(n) = O(1 + log n)
=> T(n) = O(log n + 1)
=> T(n) = O(log n).

1. Selection Sort:

total no. of comparisons = (n-1)+(n-2)+(n-3) + ...+ 1
=> n(n-1) / 2
hence
=> T(n) = O(n(n-1) / 2)
=> T(n) = O((n^2 - n) / 2)
=> T(n) = O(n^2 - n)
=> **T(n) = O(n^2)**

assumption:
if running time of an algo is having a ploynomial then in its time complexity only leading term will be considered.
e.g.
O(n^3 + n^2 + 5) => O(n^3).

assumption:
if an algo contains a nested loops and no. Of iterations of outer loop and inner loop dont know in advanced then running time of such algo will be whatever time required for statements which are inside inner loop.

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

    for( j = 0 ; j < n ; j++ ){
        statement/s  => n*n no. Of times => O(n^2) times
    }

}
```

+ features of sorting algorithms:

1. inplace => if a sorting algo do not takes extra space (i.e. space required other than actual data ele’s and constant space) to sort data elements in a collection/list of elements.

2. adaptive => if a sorting algorithm works efficiently for already sorted input sequence then it is referred as an adaptive.

3. stable => if in a sorting algorithm, relative order of two elements having same key value remains same even after sorting then such sorting algorithm is referred as stable.

Input array => 10 40 20 30 10’ 50

After Sorting:
Output => 10 10’ 20 30 40 50 => stable

Output => 10’ 10 20 30 40 50 => not stable

# Design & Analysis of an Algorithm By Coreman

2. Bubble Sort:

DAY-03:

3. Bubble Sort:

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

=> n(n-1) / 2

hence

=> T(n) = O(n(n-1) / 2)

=> T(n) = O((n^2 - n) / 2)

=> T(n) = O(n^2 - n) .... [as 2 is a divisive constant it can be negected]

=> **T(n) = O(n^2)** ....

for it=0 => pos=0,1,2,3,4

for it=1 => pos=0,1,2,3

for it=2 => pos=0,1,2

for it=3 => pos=0,1

```
for( pos = 0 ; pos < arr.length-1-it ; pos++ )
```

Best Case : if array elements are already sorted

flag = false

iteration-0:

10 20 30 40 50 60

10 20 30 40 50 60

10 20 30 40 50 60

10 20 30 40 50 60

10 20 30 40 50 60

if all pairs are in order => array is already sorted => no need of swapping => no need to goto next iteration

in best case only 1 iteration takes places and in iteration total (n-1) no. Of comparisons takes place
T(n) = O(n - 1)
T(n) = O(n).

3. Insertion Sort:

```
for( i = 1 ; i < SIZE ; i++ ){//for loop for iterations
     key = arr[ i ];
     j = i-1;

     /* if index is valid && compare value of key with an
     ele at that index *
     while( j >= 0 && key < arr[ j ] ){
          //shift ele towards its right hand side by 1 pos
          arr[ j+1 ] = arr[ j ];
          j--;//goto prev ele
     }

     //insert key at its appropriate pos
     arr[ j+1 ] = key;
}

Best Case:
Iteration-1:
10 20 30 40 50 60

10 20 30 40 50 60

no. of comparisons = 1

Iteration-2:

10 20 30 40 50 60

10 20 30 40 50 60

no. of comparisons = 1

Iteration-3:

10 20 30 40 50 60

10 20 30 40 50 60

no. of comparisons = 1

Iteration-4:
10 20 30 40 50 50

10 20 30 40 50 50

no. of comparisons = 1

Iteration-5:
10 20 30 40 50 50

10 20 30 40 50 50

no. of comparisons = 1
```

in best case total (n-1) no. of iterations are required and in each iteration only 1 comparison takes place
total no. Of comparisons = 1 * (n-1) => n-1
T(n) = O(n - 1)
T(n) = O(n) => Ω(n).

+ Linked List:

Q. Why Linked List ?

+ Limitations of an array data structure:
1. in an array we can collect/combine logically related only similar type of data element => to overcome this limitation **structure** data structure has been designed.

2. array is **static** i.e. size of an array is fixed , its size cannot be either grow or shrink during runtime.
int arr[100];

3. **addition & deletion** on an array are not efficient as it takes **O(n)** time.
- while adding ele into an array we need to shift elements towards its right hand side by one-one pos till depends on size of an array, whereas while deleting an ele from an array we need to shift elements towards its left hand side by one-one pos till depends on size of an array

- to overcome 2nd & 3rd limitations of an array data structure **linked list** data structure has been designed.

- linked list must be dynamic
- addition & deletion operations on linked list must be perfomed efficiently i.e. expected in O(1) time.

Q. What is a Linked List ?

Linked List is a basic/linear data structure, which is a collection/list of logically related similar type of data elements in which an addr(ref) of first element in it always kept into a pointer/ref referred as head, and each element contains actual data and an addr/link of its next element (as well as link of its prev element) in it.

- Element in a Linked List is also called as a node.
- Basically there are 2 types of linked list:

1. singly linked list : it is a type of linked list in which each node in it contains link to its next node only i.e. in each node no. of links = 1

- there are 2 subtypes of sll:

i. singly linear linked list
ii. singly circular linked list

2. doubly linked list : it is a type of linked list in which each node in it contains link to its next node as well as link to its prev node i.e. in each node no. of links = 2

- there are 2 subtypes of dll:

i. doubly linear linked list
ii. doubly circular linked list

i. singly linear linked list =>

- on linked list we can perform basic 2 operations

1. addition : to add/insert node into the linked list

2. deletion : to delete/remove node from the linked list

1. addition : to add/insert node into the linked list:

- we can add node into the linked list by 3 ways:

i. add node into the linked list at last position

ii. add node into the linked list at first position

iii. add node into the linked list at speficif position (in between position).

i. add node into the linked list at last position:

- we can add as many as we want number of nodes into the **slll** at last position in **O(n) time.**

```
Best Case    : Ω(1) - if list is empty
Worst Case   : O(n)
Average Case : Θ(n)
```

- to traverse a linked list => to visit each node in a linked list sequentially from first node max till last node.

- we can start traversal from first node and we get an addr/ref of first node always from head

ii. add node into the linked list at first position:

- we can add as many as we want number of nodes into the **slll** at first position in **O(1) time.**

```
Best Case    : Ω(1) - if list is empty
Worst Case   : O(1)
Average Case : Θ(1)
```

iii. add node into the linked list at specific position:
- we can add as many as we want number of nodes into the **slll** at first position in **O(n) time.**

```
Best Case    : Ω(1) - if pos = 1
Worst Case   : O(n) - if pos is last position
Average Case : Θ(n)
```

- In a Linked List Programming remember one rule:
make before break => always create new links (links which are associative with newly created node) first and then only break old links.

Lab Work : Convert SLLL program into a menu driven program.

DAY-04:

there are 2 types of programming langauges:

1. procedure oriented programming language

e.g. C : procedure => function

logic of a program gets divided into functions

maintainability =>

2. object oriented programming language

e.g. C++, Java :

logic of a program gets divided into classes

- we can delete node from linked list by 3 ways:

1. delete node at last position

2. delete node at first position
3. delete node at specific position

searchAndDelete():

- priority queue can be implemented using linked list

searchAndDelete()

- BST => deleteNode()

addition();

addLast()

addFirst()

addAtPos()

- whatever basic operations (i.e. addition & deletion) we applied on slll, all operations can be applied onto the scll as it is, execpt we need to maintained / take care about next part of last node always.

i. add node at last position
Lab Work:
ii. add node at first position
iii. add node at specific position
iv. delete node at first position
v. delete node at last position
vi. delete node at specific position

DS DAY-05:

+ Operations on slll:
  - delete node at first position
  - delete node at last position
  - delete node at specific position
  - search and delete : linear search
  - to display linked list in reverse order by using recursion
  - to reverse the linked list
  - limitations of slll
  - operations on scll:

+ limitations of scll:
- in scll, addLast(), addFirst(), deleteLast() & deleteFirst() operations are not efficient as it takes O(n) time.
- we can traverse scll only in a forward direction
- prev node of any node cannot be accessed from it

- to overcome limitations of singly linked list (slll & scll) doubly linked list has been designed.

"doubly linear linked list": it is a type of linked list in which,
- head always contains an addr of first node if list is not empty
- each node has 3 parts:
  i. data part : contains actual data of any primtive/non-primitive type
  ii. next part (ref) : contains reference of its next node
  iii. prev part (ref): contains reference of its prev node
- prev part of first node and next part of last node points to null.

- all operations that we performed on slll, can be applied as it is on dlll, only we need to maintained forward link as well as backward link of each node.

+ limitations of dlll:
- in dlll, addLast() and deleteLast() operations are not efficient as it takes O(n) time.
- we can start traversal only from first node in O(1) time.

- to overcome limitations of dlll, dcll linked list has been designed.

"doubly circular linked list": it is a type of linked list in which,
- head always contains an addr of first node if list is not empty
- each node has 3 parts:
 i. data part : contains actual data of any primtive/non-primitive type
 ii. next part (ref) : contains reference of its next node
 iii. prev part (ref): contains reference of its prev node
- prev part of first node points to last node and next part of last node points to first node.

Collection LinkedList => DCLL - generics

```
class Employee{
     //data members
     //methods
}

class LinkedList{

     static class Node{
          Employee data;//data part of a node is of type
Employee class object
          Node next;
          Node prev;
     }
```

```
    .....
}
```

+ applications of linked list:

1. linked list is used to imnplement basic data structures like stack, queue, priority queue, deque etc...
2. linked list is used to imnplement advanced data structures like tree, graph & hash table.
3. linked list is used in an OS to implement kernel data structures lik job queue, ready queue, kernel linked list, iNode list (linked list of FCB's) etc....
4. undo & redo functionalities of an OS

etc...

+ difference between array & linked list:

1. array is "static", whereas linked list "dynamic"
2. addition and deletion operations on an array are not efficient as it takes O(n) time, whereas addition & deletion operations on linked list can be performed efficiently in O(1) time and are convenient as well.
3. array elements can be accessed by using "random access method" which is efficient than "sequential access method" used in linked list.
4. searching operation can be performed on an array efficiently as we can apply binary search on it, whereas on linked list we can perform only linear search.
5. to store "n" no. of elements in an array it takes less space than to store "n" no. of elements in a linked list as we have maintained link between element in it explicitly which takes extra space.
6. array ele's gets stored into memory in data section/stack section, whereas linked list elements gets stored into the memory in the heap section.
7. as array elements gets stored into the memory at contiguos locations, to maintain link between array elements is the job compiler, whereas to maintained link between elements is the job of programmer, we have to explicitly maintained link between nodes in a list.

+ "Stack": it is a basic/linear data structure, which is a collection/list of logically related similar type of data elements in which elements can be added as well as deleted from only one end referred as top end.

- in this list, element which was inserted last can only be deleted first, so this list works in "last in first out" / "first in last out" manner, hence stack is also called as
LIFO list/FILO list.

- We can perform basic 3 operations onto the Stack in O(1) time:
1. Push : to add/insert an element onto the stack from top end
2. Pop : to delete/remove an element from the stack which is at top end
3. Peek : to get the value of an element which is at top end (without Push/Pop of an element).

- Stack can be implemented by 2 ways:
1. static implementation of stack (by using an array) => static stack
2. dynamic implementation of stack (by using linked list) => dynamic stack

"adaptor": as stack can be static as well as dynamic, as it adopts the feature of data structure by using which we implement it.

1. static implementation of stack (by using an array) => static stack:

```
    int arr[ 5 ];
    int top;


    arr : int [] - non-primitive data type
    top : int - primitive data type
```

1. Push : to add/insert an element onto the stack from top end:

step-1: check stack is not full (if stack is not full then only element can be pushed onto the stack from top end).

step-2: increment the value of top by 1

step-3: insert an element onto the stack at top end

2. Pop : to delete/remove an element from the stack which is at top end

step-1: check stack is not empty (if stack is not empty then only element can be pop from the stack which is at top end).

step-2: decrement the value of by 1 [by decrementing value of top by 1 we are achieving deletion of an element from the stack].

3. Peek : to get the value of an element which is at top end (without Push/Pop of an element).

step-1: check stack is not empty (if stack is not empty then only element can be peeked from the stack which is at top end).

step-2: return/get the value of an element at top end [without increment/decrement top].

# DAY-06:
- Operations SCLL, DLLL, DCLL
- Stack: concept & definition
- We can perform basic 3 operations on Stack in O(1) time: Push, Pop & Peek
- Stack can be implemented by 2 ways:

1. Static Stack (by using an array)
2. Dynamic Stack (by using linked list-dcll)

- there is no stack full condition in a dynamic stack
- if list is empty => stack is empty

Stack works in LIFO manner:
Push => addLast() - O(1)
Pop => deleteLast() - O(1)
Peek => get the data part of last node

if(head == null) => list is empty => stack is empty

head => 44 33 22 11

OR

Push => addFirst() - O(1)
Pop => deleteFirst() - O(1)
Peek => get the data part of first node

Lab Work => to implement dynamic stack

- stack is also used in / to implement expression conversion algorithms and evaluation algorithms.

What is an expression ?
Combination of operands and operators
- there are 3 types of expression

1. infix expression   :   a+b
2. prefix expression   :   +ab
3. postfix expression :   ab+

infix expression => a*b/c*d+e-f*g+h

Lab Work : Implement algo to convert given parenthesized infix expression into its equivalent prefix expression.

Lab Work : Implement an algo for postfix evaluation (if an expression contains multi-digit operands).
- vector of strings to store infix & postfix expression
- stack of integers, while pushing an operand of type string we need to convert into its equivalent int.

+ Queue : it is a **linear/basic data structure** which is **collection/list of logically related similar type of data elements** in which elements can be added into it from one end referred as **rear end** and elements can be deleted from it which is at **front end.**

- in this collection/list, **element which was inserted first can be deleted first,** so this list works in **first in first out manner** or **last in last out manner,** hence queue is also called as **fifo list / lilo list.**
- On Queue data structure basic 2 operations can be performed in **O(1)** time:

1. Enqueue => insert/add an element into the queue from rear end.

2. Dequeue => delete/remove an element from the queue which is at front end.

- there are 4 types of queue:

1. linear queue (fifo)

2. circular queue (fifo)

3. priority queue => it is a type of queue in which elements can be added into it from rear end randomly (i.e. without checking priority), whereas element which is having highest priority can only be deleted first.

4. double ended queue (deque) => it is a type of queue in which elements can added as well as deleted from both the ends.
- on deque we can perform basic 4 operations in **O(1)** time:

i. push_back()   => addLast()
ii. push_front() => addFirst()
iii. pop_back() => deleteLast()
iv. pop_front() => deleteFirst()

- deque can be implemented by using **DCLL.**
- further there are 2 types of deque:

1. input restricted deque => in this type of deque elements can be added into it only from one end, whereas elements can be deleted from both the ends.

2. output restricted deque => in this type of deque elements can be added into it from both the ends, whereas elements can be deleted only from one end.

1. Enqueue => insert/add an element into the queue from rear end.

```
step-1: check queue is not full (if queue is not full
then only we can insert an element    into it).
step-2: increment the value of rear by 1
step-3: insert an element into the queue from rear end
step-4: if( front == -1 )
            front = 0
```

2. Dequeue => delete/remove an element from the queue which is at front end.

```
step-1: check queue is not empty (if queue is not empty
then only we can delete an element from it).
step-2: increment the value of front by 1 [ by means of
incrementing value of front by 1 we are achieving
deletion of an element from queue ].
```

DAY-07:

- queue can be implemented by 2 ways:

1. static implementation of queue (by using an array)
2. dynamic implementation of queue (by using an linked list).

1. static implementation of queue (by using an array):

```
int arr[ 5 ];
int front;
int rear;
```

```
arr   : int []
front : int
rear  : int
```

Circular Queue:
rear = 4, front = 0
rear = 0, front = 1
rear = 1, front = 2

in a cir quueue => if front is at next pos of rear => queue full
front == (rear + 1)%SIZE

for rear = 0, front = 1 => front is at next pos of rear
=> cir q is full
=> front == (rear + 1)%SIZE
=> 1 == (0+1)%5
=> 1 == 1%5
=> 1 == 1 => LHS == RHS => cir q is full

for rear = 1, front = 2 => front is at next pos of rear
=> cir q is full
=> front == (rear + 1)%SIZE
=> 2 == (1+1)%5
=> 2 == 2%5
=> 2 == 2 => LHS == RHS => cir q is full

```
for rear = 2, front = 3 => front is at next pos of rear
=> cir q is full
=> front == (rear + 1)%SIZE
=> 3 == (2+1)%5
=> 3 == 3%5
=> 3 == 3 => LHS == RHS => cir q is full

for rear = 3, front = 4 => front is at next pos of rear
=> cir q is full
=> front == (rear + 1)%SIZE
=> 4 == (3+1)%5
=> 4 == 4%5
=> 4 == 4 => LHS == RHS => cir q is full



for rear = 4, front = 0 => front is at next pos of rear
=> cir q is full
=> front == (rear + 1)%SIZE
=> 0 == (4+1)%5
=> 0 == 5%5
=> 0 == 0 => LHS == RHS => cir q is full


rear++;
rear = rear + 1;

rear = ( rear + 1 ) % SIZE

for rear=0 => rear=(rear+1)%SIZE = (0+1)%5 = 1%5 = 1
for rear=1 => rear=(rear+1)%SIZE = (1+1)%5 = 2%5 = 2
for rear=2 => rear=(rear+1)%SIZE = (2+1)%5 = 3%5 = 3
for rear=3 => rear=(rear+1)%SIZE = (3+1)%5 = 4%5 = 4
for rear=4 => rear=(rear+1)%SIZE = (4+1)%5 = 5%5 = 0
```

2. dynamic implementation of queue (by using an linked list : DCLL).

- if list is empty => queue is empty
- there is no queue full condition for dynamic queue

Enqueue => addLast()
Dequeue => deleteFirst()

OR
head => 44

Enqueue => addFirst()
Dequeue => deleteLast()

Lab Work:
1. implement dynamic queue
2. implement priority queue by using dcll

Priority Queue by using Linked List:
searchAndDelete()

```
class Node{
     int data;
     Node next;
     Node prev;
     int priorityValue;
}
```

# Basic Data Structures: comfortable
# Advanced Data Structures:

- tree (can be implemented by using an array as well as linked list).
- binary heap (array implementation of a tree)
- graph (array & linked list)
- hash table (array & linked list)
- merge sort & quick sort

+ Tree:
+ tree terminologies:

root node
parent node/father
child node/son
grand parent/grand father
grand child/grand son
ancestors => all the nodes which are in the path from root node to that node

- further restrictions can be applied on binary tree to mainly to achieve addition, deletion and searching operations effciently expected in O(log n) time => binary search tree can be formed.

	Addition	Deletion	Searching
Array	O(n)	O(n)	O(log n)
Linked List	O(1)	O(1)	O(n)
BST	O(log n)	O(log n)	O(log n)

- binary search tree => it is a binary tree in which left child is always smaller than its parent, and right child is always greater than or equal to its parent.

While adding node into the BST => We need to first find/search its appropriate position in a BST and after that we can add node at that position.

- there are basic two tree traversal methods:
1. bfs (breadth first search) traversal / levelwise traversal:
- traversal always starts from root node
- and nodes in a bst gets visited levelwise from left to right.

2. dfs (depth first search) traveral
- under dfs traversal further there are 3 ways by which tree can be traversed:
i. Preorder (V L R) :
- start traversal always from root node
- first visit cur node, then visit its left subtree and we can visit right subtree of any node only after either visitng its whole left subtree or left subtree is empty.
- in this traversal, root node always gets visited first, and this property remains recursively true for each subtree.

ii. Inorder (L V R) :
- start traversal always from root node
- in this traversal, we can visit any node only after visiting its whole left subtree or its left subtree is empty.
- in this traversal, all the nodes gets visited always in a an ascending sorted order.

iii. Postorder (L R V):
- start traversal always from root node
- in this traversal, we can visit any node only after visiting its whole left subtree as well as right subtree or its left subtree and right subtree are empty.
- in this traversal, root node always gets visited last, and this property remains recursively true for each subtree.

Lab Work : Implement nonRecPostOrder() algorithm/method.

Lab Work : implement recusrive addNode() function for BST.

# DAY-08:
- queue : linear queue & circular queue
- tree : concept & definition
- tree terminologies
- addNode into the BST

inorder successor
inorder predecessor

- In a BST, inorder successor of any node usually exists/found at left end in its right subtree if it is is having right subtree.

- tree traversal algorithms:

preorder, postorder & inorder by using rec & non-rec way

dfs traversal & bfs traversal

searching on BST

deletion of a node in a BST.

calculation of height of BST.

Todays:

quick sort

m   ort

tree concepts: complete binary tree, avl tree, balanced BST, balace factor, threaded binary treem multi-way tree, b-tree & b+ tree, binary heap

```
# partitioning
i=left;
j=right;
pivot = arr[ left ];

/* shift ele's which are smaller than pivot towards left
hand side, and ele's which are greater than pivot shift
towards right side.*/

while( i < j ){

    while( i <= right && arr[ i ] <= pivot )
        i++;//goto the next ele

    while( arr[ j ] > pivot )
        j--;//goto the prev ele

    //if i & j have not crossed then swap them
    swap(arr[ i ], arr[ j ]);
}
//swap pivot ele with jth pos ele
swap( arr[ left ], arr[ j ] );
```

- in quick sort, worst case may occurs either array ele’s are already sorted or present exactly in reverse order, this condition rarely occurs.

[60 50 40 30 20 10]
pass-1: [50 40 30 20 10] 60 [RP]
pass-2: [40 30 20 10] 50 [RP]
pass-3: [30 20 10] 40 [RP]
pass-4: [20 10] 30 [RP]
pass-5: [10] 20 [RP]

n * n => n^2

[10 20 30 40 50 60]
pass=1: pivot = 10
[LP] 10 [20 30 40 50 60]

[20 30 40 50 60]
pass=2: pivot = 20
[LP] 20 [30 40 50 60]

pass-3: pivot = 30
[30 40 50 60]
[LP] 30 [40 50 60]

pass-4: pivot
[40 50 60]

n^2

if pivot element gets selected as mid pos ele, then there are very rare chances to occurs worst case.

[60 50 40 30 20 10]
pass-1: [10 20 30] 40 [60 50]
pass-2: [10] 20 [30]
pass-3: [50] 60 [RP]

log n + log n
2 (log n) => O(n log n)

merge sort on linked list : to merge two already sorted linked lists into a third list in a sorted manner

l1 => head =>

l2 => head =>

l3 => head => 10 -> 15 -> 20 -> 25 -> 30 -> 35 -> 40 -> 45 -> 50

google map app:
information about 1000’s of cities and info between paths of those cities can be kept.

City

```
class City{
     String cityName;
     String cityCode;
     String pinCode;
     String state;
     ......
}
```

City class objects => vetices
Pune <====> Mumbai

Information of Path between Citie => Weight

```
class Path{
     String toCity;
     String fromCity;
     float distanceInKM;
     .....
}
```

# DS_DAY-11:

+ Graph Traversal Algorithms:
1. dfs (depth first search) traversal - stack
2. bfs (breadth first search) traversal - queue

- graph can be a tree but tree cannot be a graph.
- subgraph of a graph which can be formed by means of removing one or more edges from it in such a way that it should remains connected and do not contains a cycle. Such a subgraph of a graph is referred as spanning tree of a given graph.

Hash table: