PRECAT: OM19

Data Structure

SunBeam Institute of Information & Technology, Hinjwadi, Pune & Karad.

SACHIN PAWAR

PreCAT Scope : Data Structure

- In Section B, 7 Questions are reserved for this subject

and mostly all questions are pseudocode oriented (i.e. theory based/concepts).

- In this course the main focus is on **implementation of basic data structures** and **introduction to an advanced data structures** to build base required to learn and implement advanced data structures and algorithms in CDAC courses.

Data Structures: Introduction

+ Introduction

- Data structure
- Algorithm and analysis of an algorithm

+ Array

- Concept & definition
- **Searching Algorithms:**
  1. Linear Search (algorithm & implementation)
  2. Binary Search (algorithm & implementation)
- **Sorting Algorithms:**
  1. Selection Sort (algorithm & implementation)
  2. Bubble Sort (algorithm & implementation)
  3. Insertion Sort (algorithm & implementation)
  4. Quick Sort (algorithm)
  5. Merge sort (algorithm)

Data Structures: Introduction

+ Linked List

- Concept & definition
- Types of Linked List
- Operations on Linked List
- Differnce between an array and linked list

+ Stack

- Concept & definition
- Implementation of stack data structure
- Stack applications algorithms:

1. Conversion of infix expression into its equivalent prefix
2. Conversion of infix expression into its equivalent postfix
3. Conversion of prefix expression into its equivalent postfix
4. Postfix expression evalution

Data Structures: Introduction

+ Queue

- Concept & definition
- Types of queue
- Implementation of queue data structure
- Applications of queue

+ Introduction to an advanced data structure

- Tree
- Binary Heap
- Graph
- Hash Table

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 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 elements gets stored into the memory in a linear manner (e.g. sequentially) and hence can be accessed linearly/sequentially.

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

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

- Tree (Hierarchical)
- Graph
- Hash Table
- Binary Heap

Data Structures: Introduction

Array: It is a **basic/linear data structure** which is a collection/list of **logically related similar type of elements** in which data elements gets stored 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 elements** gets stored into the memory **collectively (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 (like C, C++, Java, Python, Assembly etc...) given to the machine to do specific task.

Q. What is an Algorithm?

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

- An algorithm is a finite set of instructions written in human understandable language (like english) **with some programming constraints**, if followed, acomplishesh a given task, such an algorithm also called as **pseudocode.**

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

Data Structures: Introduction

Example: An algorithm 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;
}
```

- In this algorithm, **traversal/scanning** operation is applied on an array. Initially sum is 0, each array element gets added into to the sum by traversing array sequentially from the first element till last element and final result is returned as an output.

Data Structures: Introduction

- **Analysis of an algorithm** is a work of determining 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 required for it to run to completion.

2. Space Complexity of an algorithm is the amount of space i.e. computer memory required for an algorithm to run to completion.

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 gets done on the basis of no. of comparisons, in addition of matrices algorithms addition is the basic operation and hence on the basis of addition operation.

Data Structures: Introduction

"Best case time complexity": if an algo takes min amount of time to complete its execution then it is referred as best case time complexity.

"Worst case time complexity": if an algo takes max amount of time to complete its execution then it is referred as worst case time complexity.

"Average case time complexity": if an algo takes neither min nor max amount of time to complete its execution then it is referred as an average case time complexity.

"Asympotic Notations":

1. Big Omega (Ω): this notation is used to denote best case time complexity – also called as **asymptotic lower bound**

2. Big Oh (O): this notation is used to denote worst case time complexity - also called as **asymptotic upper bound**

3. Big Theta (θ): this notation is used to denote an average case time complexity - also called as **asymptotic tight bound**

1. Linear Search/Sequential Search:

- In this algorithm, key element gets compared sequentially with each array element by traversing it from first element till either match is found or maximum till the last element.

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

Best Case: If key is found at very first position in only 1 no. of comparison then it is considered as a best case and running time of an algorithm in this case is O(1) => and hence time complexity = **Ω(1)**

Worst Case: If either key is found at last position or key does not exists, 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) => and hence time complexity = **O(n)**

Average Case: If key 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) => and hence time complexity = **θ(n)**

Data Structures: Searching Algorithms

2. Binary Search/Logarithmic Search:

- This algorithm follows **divide-and-conquer** approach.

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

Step-1 : accept key from the user

Step-2 : calculate mid position of an array by the formula, **mid = (left+right)/2** (by means of calculating mid position big size array has been divided logically into two subarrays, from **left to mid = left subarray** & **mid+1 to right = right subarray**)

Step-3 : compare the value of key with element which is at mid position. if key matches with element at mid position means key is found and return true.

Step-4 : if key do not matches then check, is the value of key is less than element which is at mid position, if yes then goto search key only into the left subarray by skipping whole right subarray otherwise (means of the value of key is greater than element which is at mid position) goto search key only into the right subarray by skipping whole left subarray.

Step-5 : repeat Step-2, Step-3 & Step-4 till key is found or max till the subarray is valid,

if subarray becomes invalid means key is not found and hence return false in this case.

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

- as in each iteration one 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 => $T(n) = (n/2^3) + 3$

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

let us assume,

=> $n = 2^k$

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

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

=> $\log n = k$

=> k = log n

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

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

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

Best Case: if the key is found in very first iteration at mid position in only 1 no. of comparison 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).**

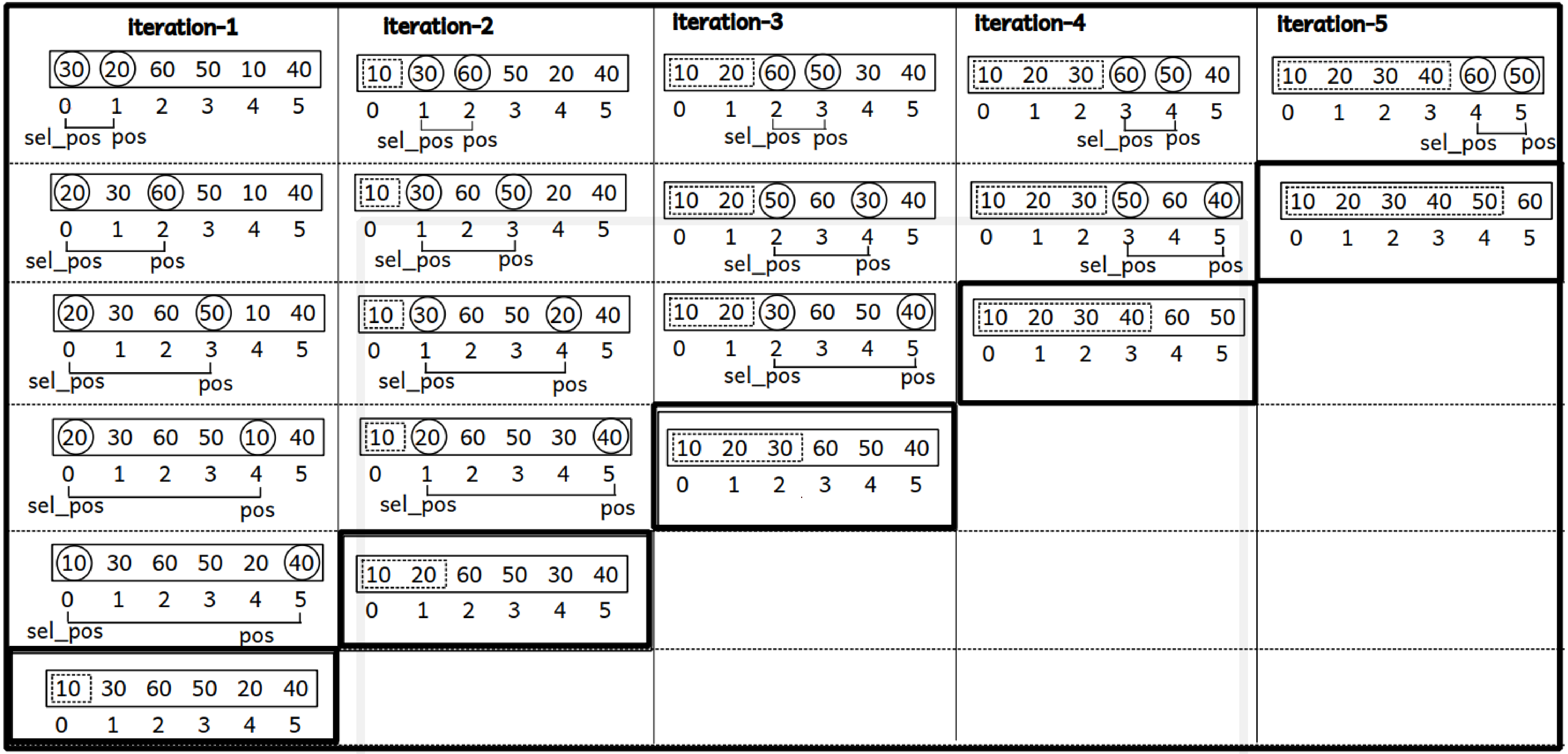
Average Case: if key is not found in the first iteration and it is found at non-leaf position it is considered as an average case and running time of an algorithm in this case is O(log n) = **θ(log n).**

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, if selected position element 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 selected position element 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

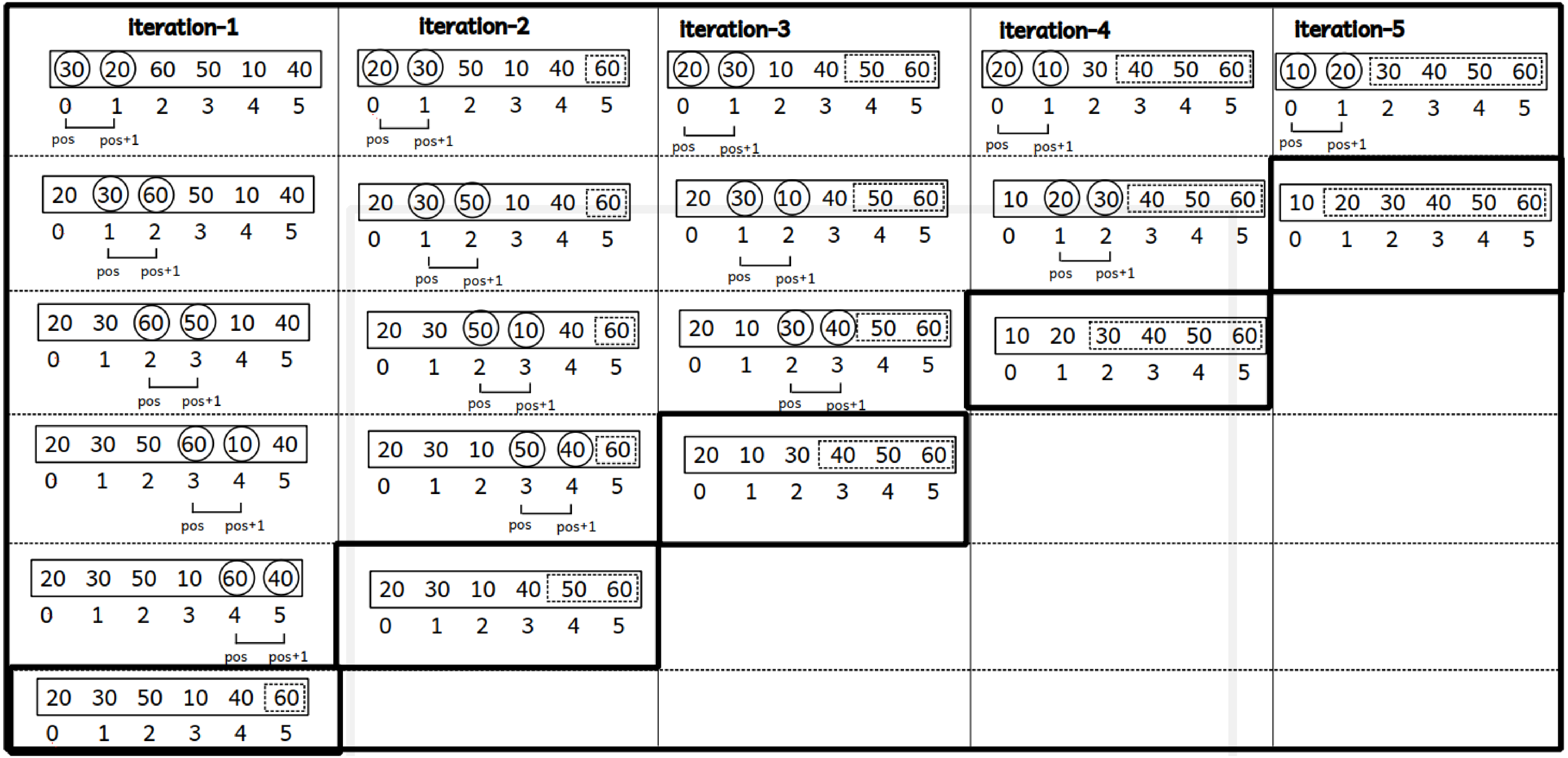


Best Case : $\Omega(n^2)$
Worst Case : $O(n^2)$
Average Case : $\theta(n^2)$

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 : $\theta(n^2)$

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

Worst Case : O(n^2)

Average Case: θ(n^2)

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

4. Merge Sort:

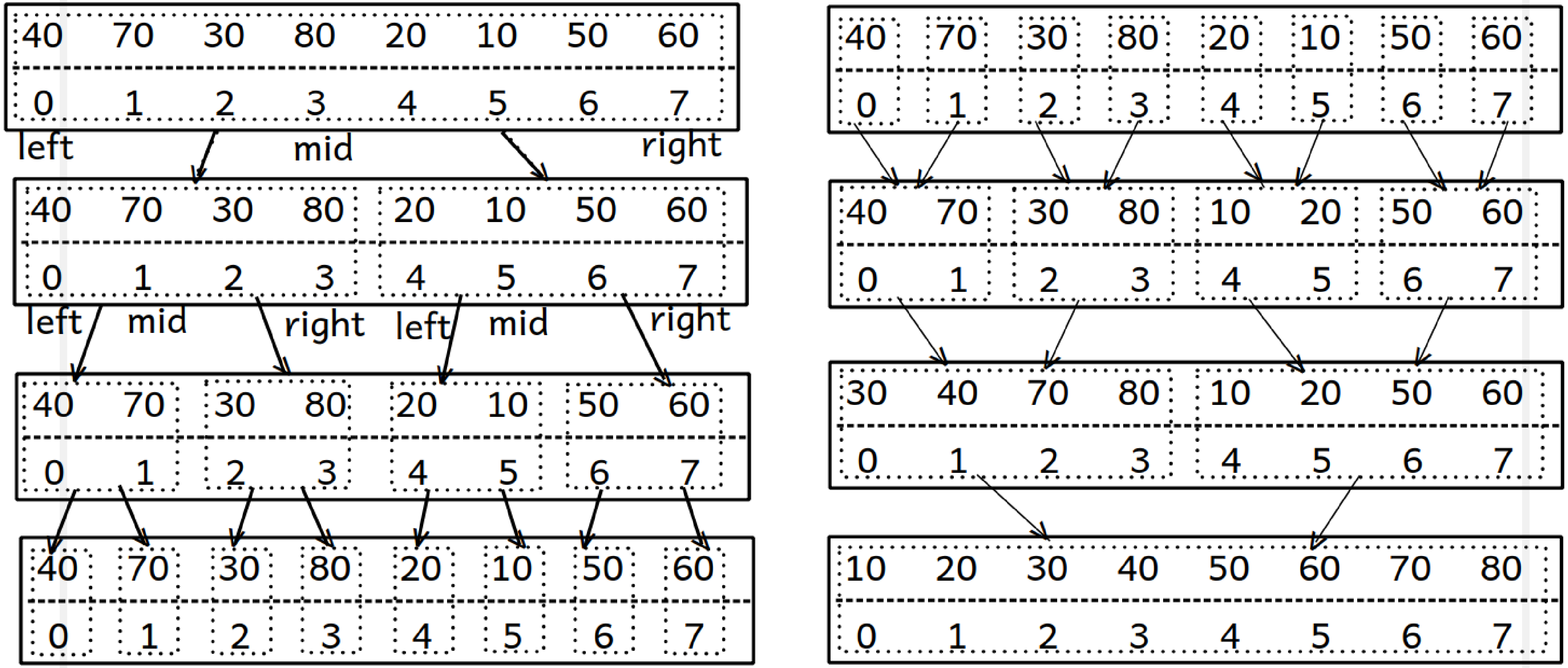
- This algorithm follows **divide-and-conquer** approach.
- In this algorithm, big size array is divided logically into smallest size (i.e. having size 1) subarrays, as if size of subarray is 1 it is sorted, after dividing array into sorted smallest size subarray's, subarrays gets merged into one array 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

Merge Sort

Dividing big size array into smallest size subarrays

Merge already sorted arrays



Data Structures: Sorting Algorithms

Best Case : $\Omega(n \log n)$
Worst Case : $O(n \log n)$
Average Case : $\theta(n \log n)$

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 : $\Omega(n \log n)$

Worst Case : $O(n^2)$ – worst case rarely occures

Average Case : $\theta(n \log n)$

- 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 **Linked List** data structure has been designed.

Linked List: It 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.

- each element contains data and an address of its next (as well as 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 on linked list data structure.

- We can add and delete node by three ways: we can 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, last position and at any specific position.

1. Singly 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 two parts:

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

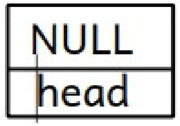
ii. pointer part(next): contains an address of its next element/node.

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

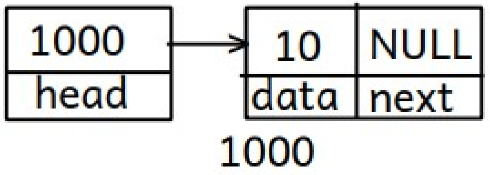
Data Structures: Linked List

SINGLY LINEAR LINKED LIST

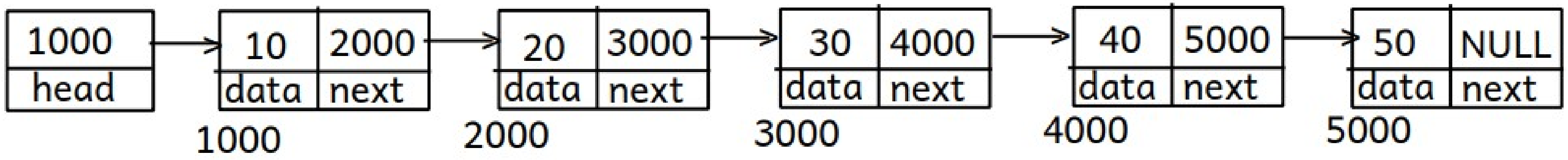
1) singly linear linked list --> list is empty



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 starts traversal only from first node and can traverse the SLLL 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 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: contains data of any primitive/non-primitive type.

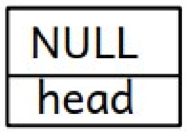
ii. pointer part(next): contains an address of its next element/node.

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

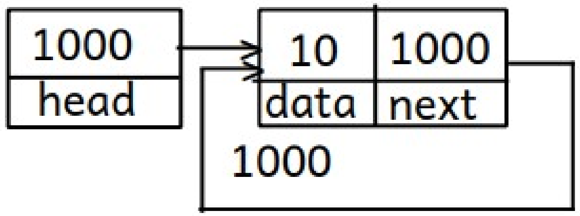
Data Structures: Linked List

SINGLY CIRCULAR LINKED LIST

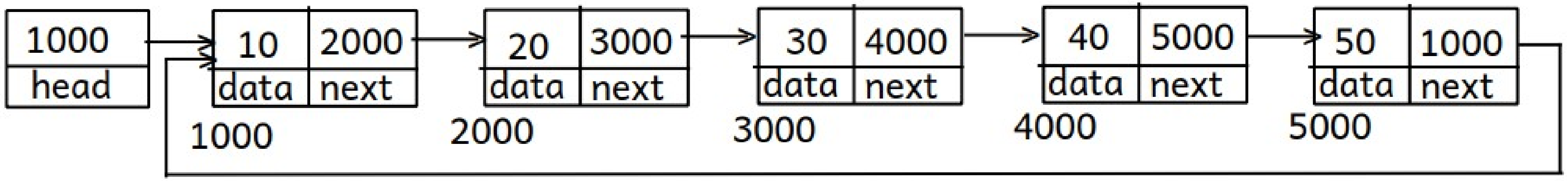
1) singly circular linked list --> list is empty



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.

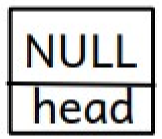
iii .pointer part(prev): contains an address of its previous element/node.

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

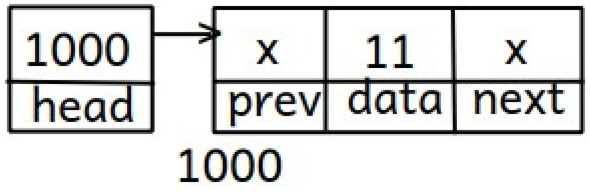
Data Structures: Linked List

DOUBLY LINEAR LINKED LIST

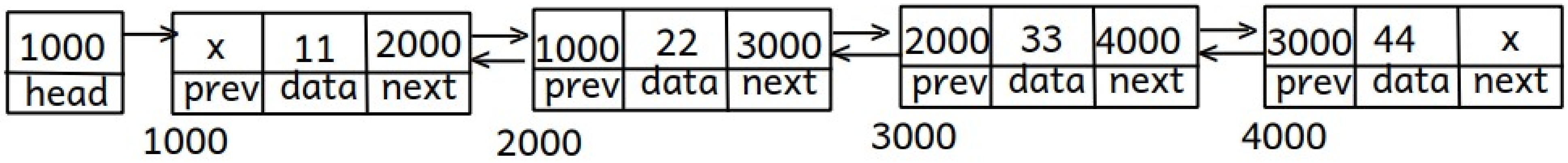
1. doubly linear linked list --> list is empty



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.

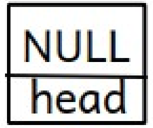
iii .pointer part(prev): contains an address of its previous element/node.

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

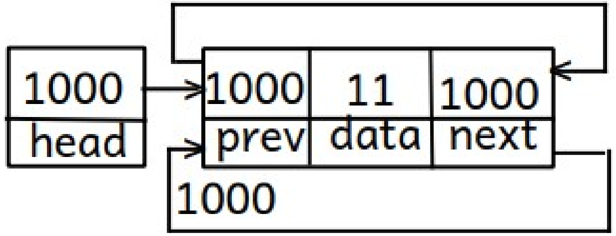
Data Structures: Linked List

DOUBLY CIRCULAR LINKED LIST

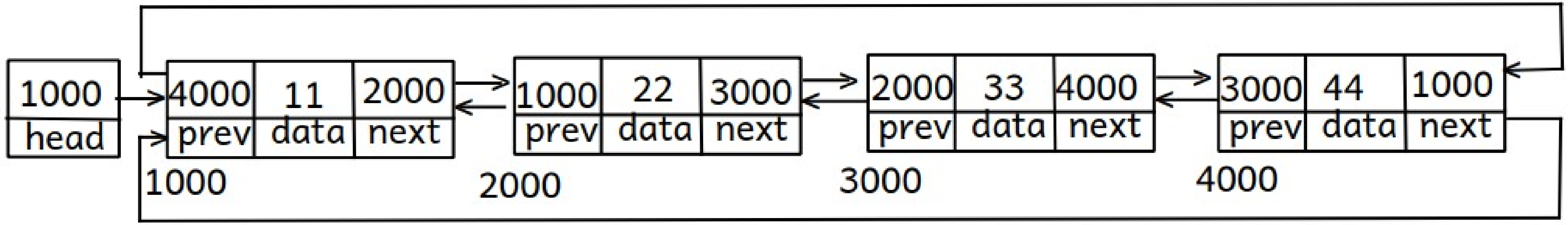
1. doubly circular linked list --> list is empty



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 or from last node.
- Any node can be revisited.
- Previous node of any node can be accessed from it

Array v/s Linked List:

- 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 linked list elements which can be accessed by **sequential access** method.
- 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 maintain link between elements is the job of **compiler**.

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 algorithm 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.

- 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.

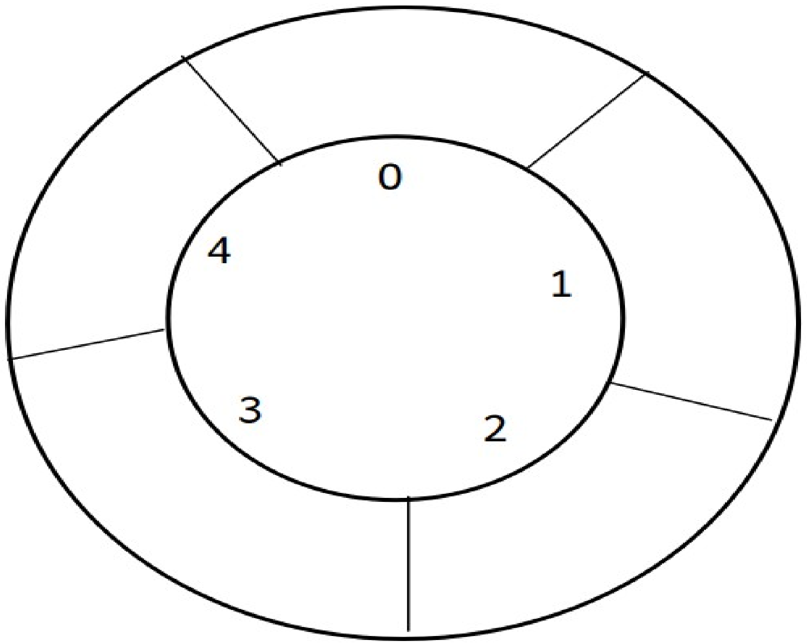
- Priority queue can be implemented by using linked list, whereas it can be implemented efficiently by using **binary heap.**

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:

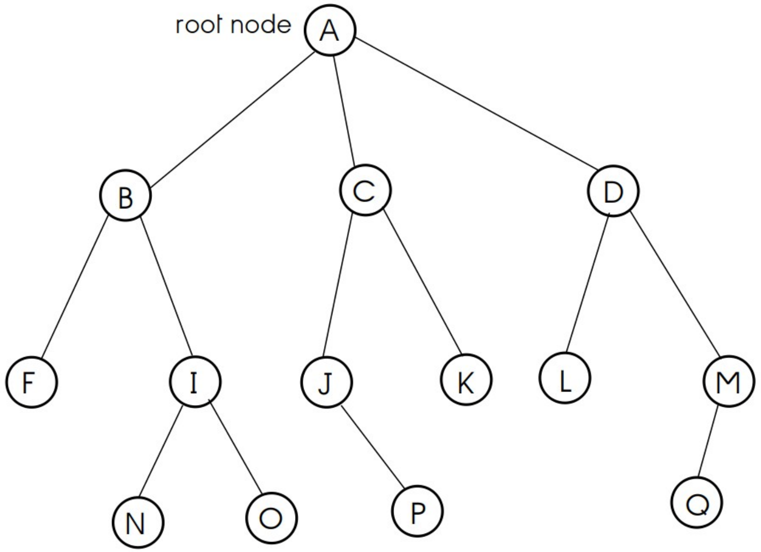
- Queue is used to implement OS data structures like **job queue, ready queue, message queue, waiting queue** etc...

- 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.

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 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 dynamic in nature, it can grow upto any level and any node can have any number of child nodes, but due this 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.

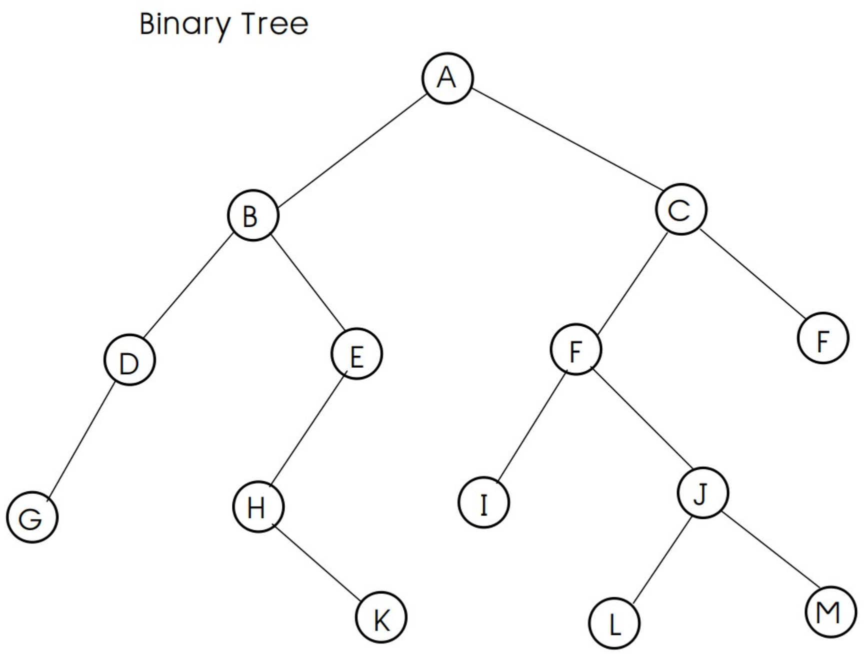
OR

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

1. root element

2. left subtree (may be empty)

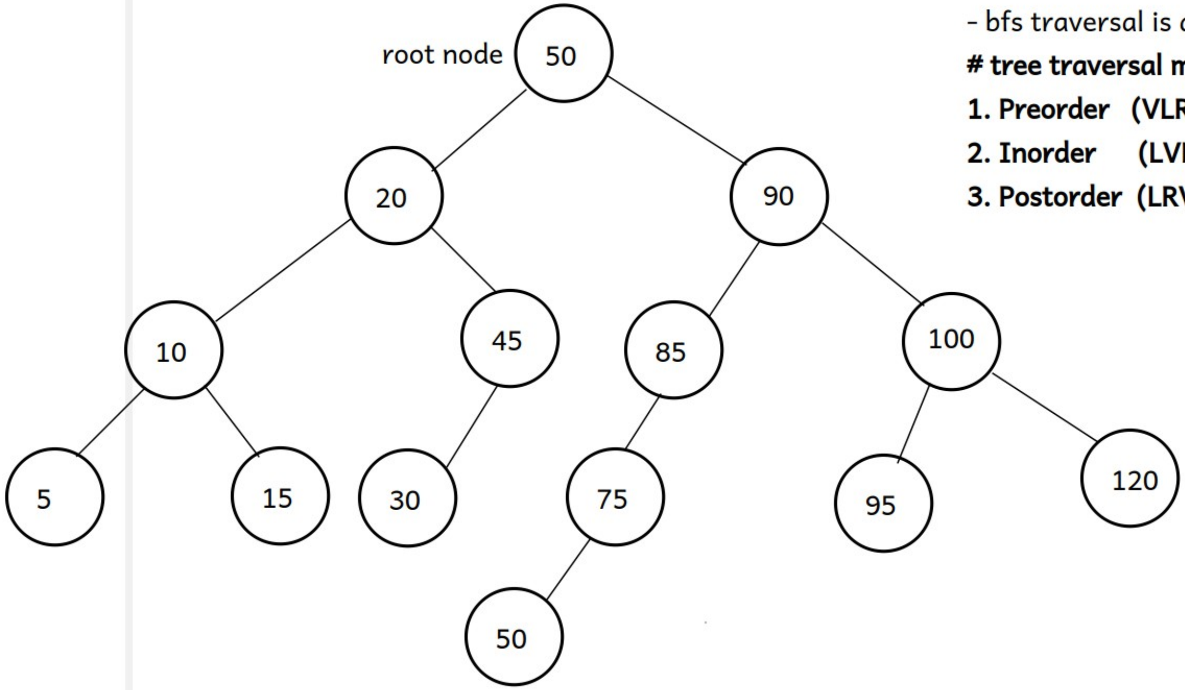
3. right subtree (may be empty)



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: Tree

- **Height of a node** = max(ht. of its left subtree, ht.of its right subtree) + 1.

- **Maximum height of a BST = n**, n=no. of elements/nodes in it

- Minimum height of a BST = log n, n=no. of elements/nodes in it

- BST with minimum height i.e. log n for given input is referred as **Balanced BST, as operations like addition, deletion and searching can be performed in O(log n) time.**

- BST is said to be balanced, only if all nodes in it are balanced.

- Node is said to be balanced, only if its balance factor is either -1 or 0 or +1.

- Balance Factor of a node = ht. of its left subtree – ht. of its right subtree

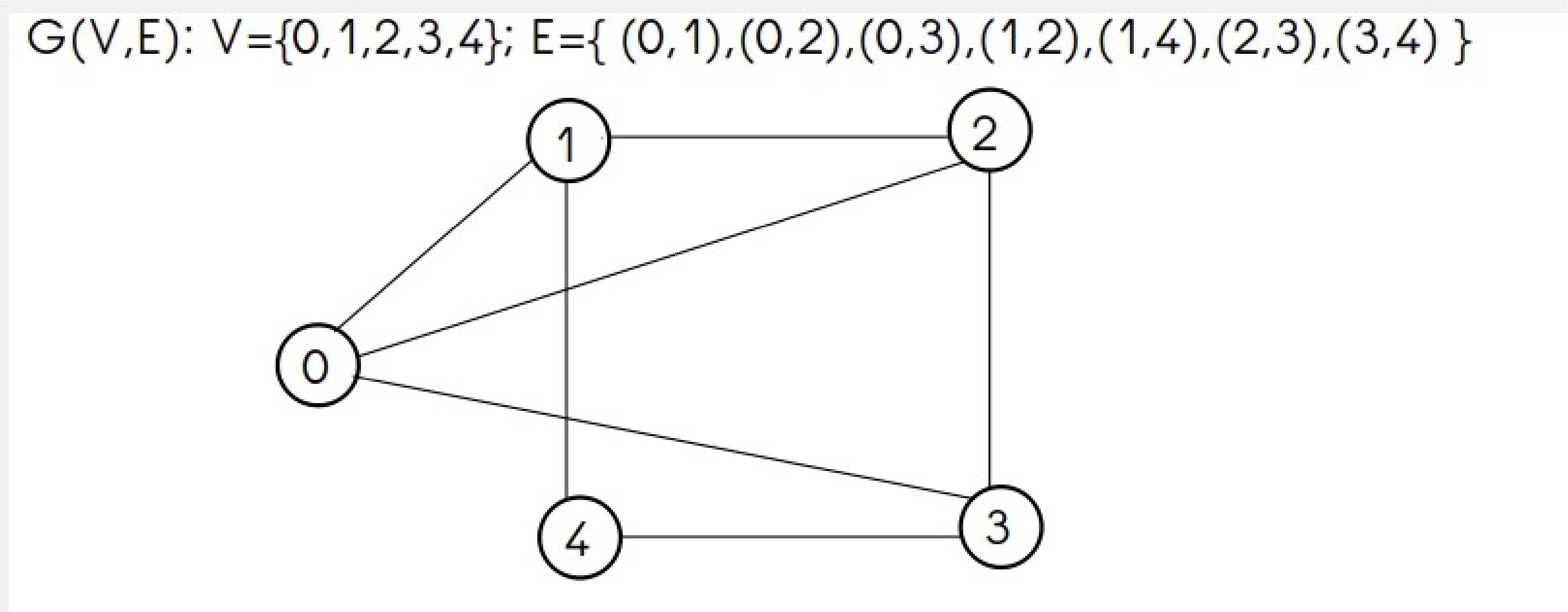
- Self Balanced BST – BST in which while adding and deleting node itself it makes sure that BST remains balanced, this cocept was designed by **Adelson Velsinki** & **Lendis** and hence Self Balanced BST is also called as **AVL Tree.**

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).



Data Structures: Graph

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

- 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.**

(u,v) == (v,u) --> unordered pair of vertices -> undirected edge -> undirected graph

- if we cannot represent any edge either (u,v) OR (v,u) then it is referred as **unordered pair of vertices** i.e. directed edge.

(u,v) != (v,u) --> ordered pair of vertices -> directed edge -> directed graph (di-graph).

- complete graph: if all the vertices are adjacent to remaining all vertices in a given graph.

- connected vertices: if path exists between two vertices then those two vertices are referred as a connected vertices otherwise not-connected.

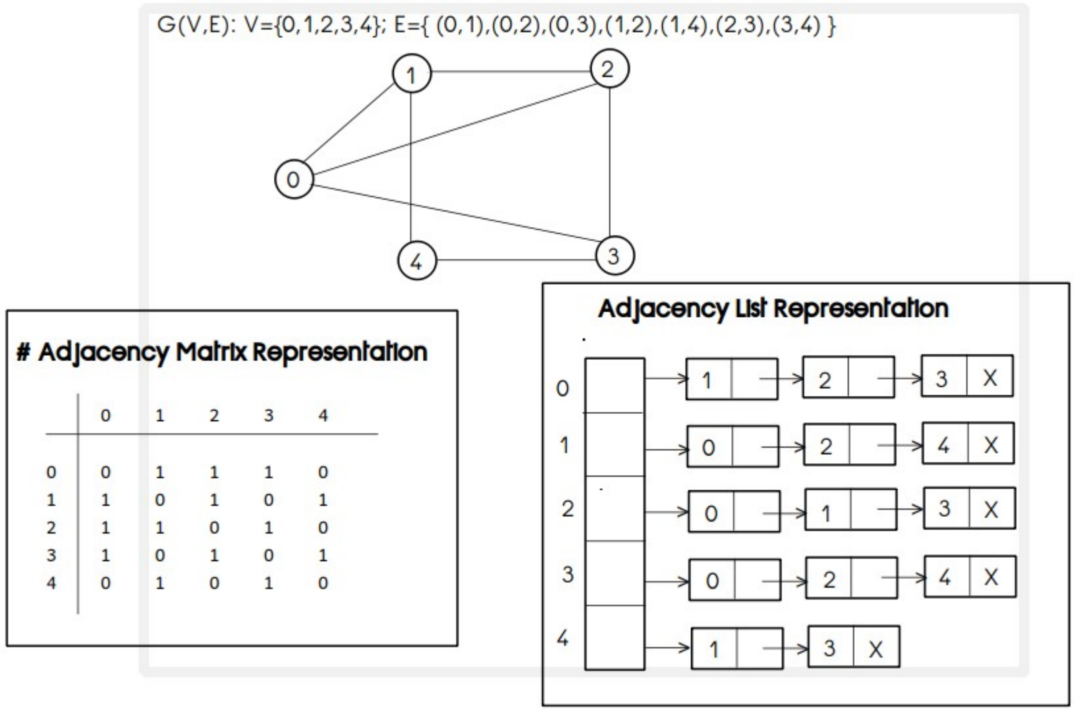
- connected graph: if any vertex is connected to remaining all vertices in a given graph

Data Structures: Graph

- There are two graph representation methods:

1. Adjacency Matrix Representation (2-D Array)

2. Adjacency List Representation (Array of Linked Lists)



	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