Data Structures and Algorithms

what is Data Structure?

It is a way to store and organize the data, so that it can be used efficiently.

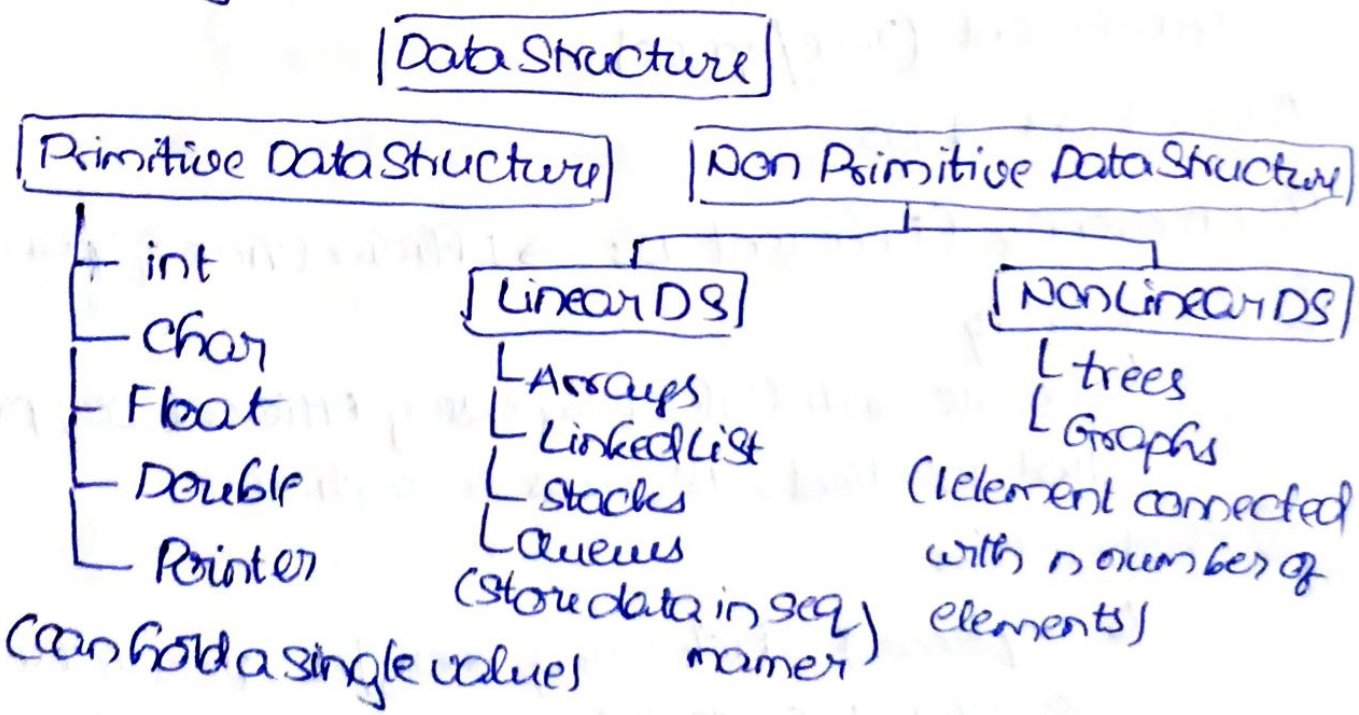
Examples

① Array ③ Structure ⑤ Stack ⑦ Graph
② pointer ④ LinkedList ⑥ Queue ⑧ Searching
⑨ Sorting

* DS is a set of algorithms that we can use in any PL to structure data in memory.

what is abstract data type?

* To keep the data structured in memory, abstract data type concept is been introduced, the ADT is bound by set of rules.



Data structures

↳ Static DS (Size allocated at compile time)
Maximum size is Fixed

↳ Dynamic DS (Size allocated at the runtime)
Maximum size is Flexible

Operations on DS

1. Searching
2. Sorting
3. Insertion
4. updation
5. Deletion

Necessity of DS

* Store the data efficiently in terms of time and space.
* we require some DS to implement ~~another~~ ~~DS~~ a particular ADT
* Ex: Stack (ADT) created/implemented using LL (DS)

ADT ⇒ Blueprint

DS ⇒ Implementation

* Selection of DS to implement ADT depends on user requirement (Time/space)

Advantages of DS

1. Efficiency (Efficient DS → Efficient time & space)
2. Reusability
   ↳ Create an Interface by using Efficient DS, provide that to client, he uses everytime
3. Abstraction
   ↳ Implement Stack using Array/LL, provide the interface to user, user don't have implementation details hence Abstraction

Main AIM: Store and retrieve as fast as possible

Basic Terminology

① Data: Elementry value or collection of values

② Data Item: Single unit of value

③ Group Items: Data items that have subordinate D.Items
Ex: Employee Name (First, Last, Middle Name)

④ Elementry Item: DataItem which are unable to Divide (EID)

⑤ Entity: Object that has a distinct identity (Person, Place)

⑥ Attribute: Characteristic of an Entity

Data structures => Allows us store & organise data

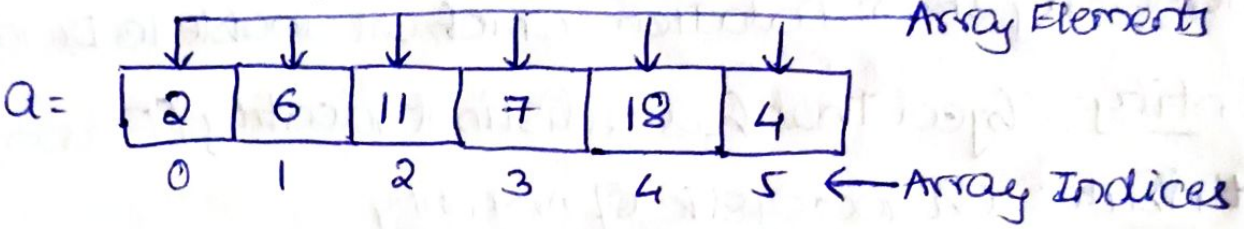
Algorithms => process the data meaningfully

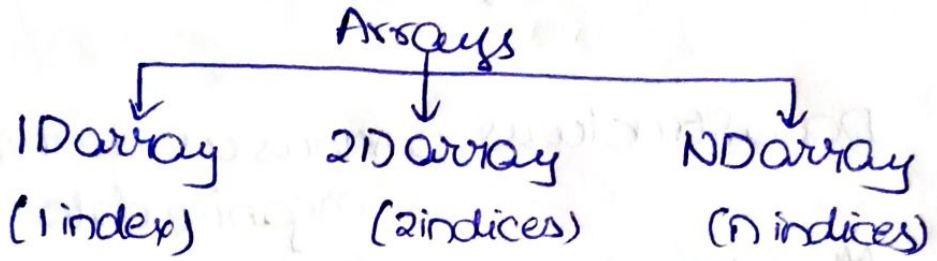
* Non Premitive Data Structure are data structures derived from Primitive DS
* NPDS forms set of data elements that is either Group of homogeneous / Hetrogeneous Dat structure
  - Same DT forms as grp
  - Diff DT forms as grp

Linear Data Structures

① Array

* Collect Multiple data elements of the same datatype into one variable.
* Data is stored in contiguous memory location, so retrieving randomly through index based on array variable is possible.





Applications

① Store list of data elements of same type
② use as auxilary storage for other data structs
③ Store data elements of Binary tree of Fixed count

② Linked List

* Singly linked list:
* Doubly Linked List
* Circular Linked List

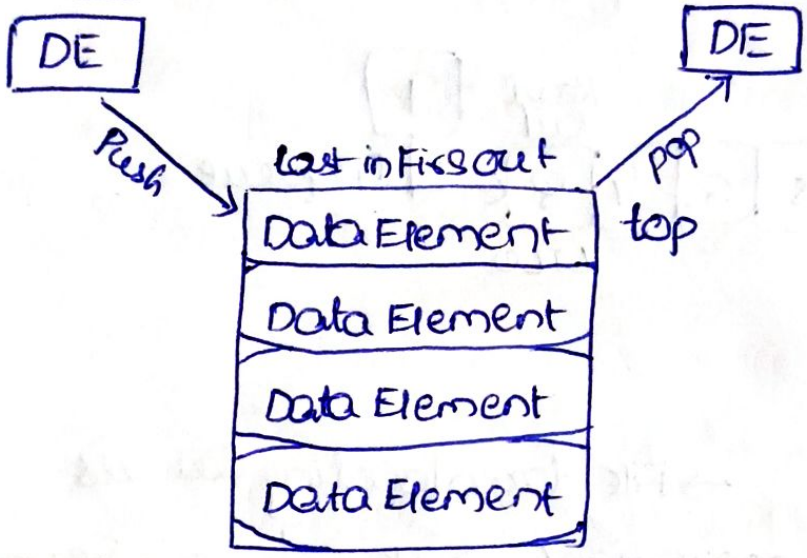
Applications of Linked List:

* Helps us implement stacks, queues, binary trees and graphs of predefined size.
* OS functionality Dynamic Mem Manag
* slideshow functionality of PPT
  First slide → last slide
* DLL → Browser Front & Back navigation

Stack

* Linear Data structure that follows LIFO
* Insertion & Deletion only from top end.
* Implemented using
  - Contiguous memory (Array)
  - Non Contiguous Memory (Linked List)
* Can access only stack's top at any time

operations:



Applications:

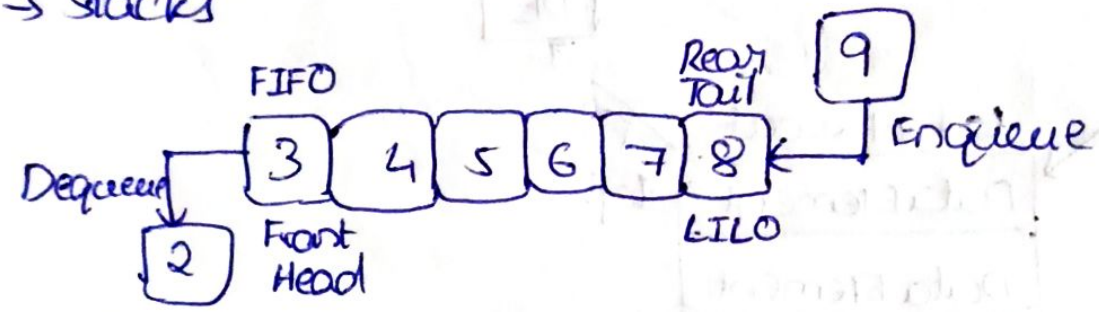
* Temporary storage for recursive operations, function calls nested operations
* Evaluate arithmetic Expressions
* Infix Exp → Postfix Exp
* Match paranthesis
* Reverse a string
* Backtracking
* DFS in graph & tree traversal
* undo/Redo-func

Queues

* Linear Data Structure
* Insertion done at one end
* Deletion done at opposite end
* First in First out

Can be implemented by
→ Arrays
→ LinkedLists
→ Stacks

Real life Examples
ticket counter
Escalator
car wash



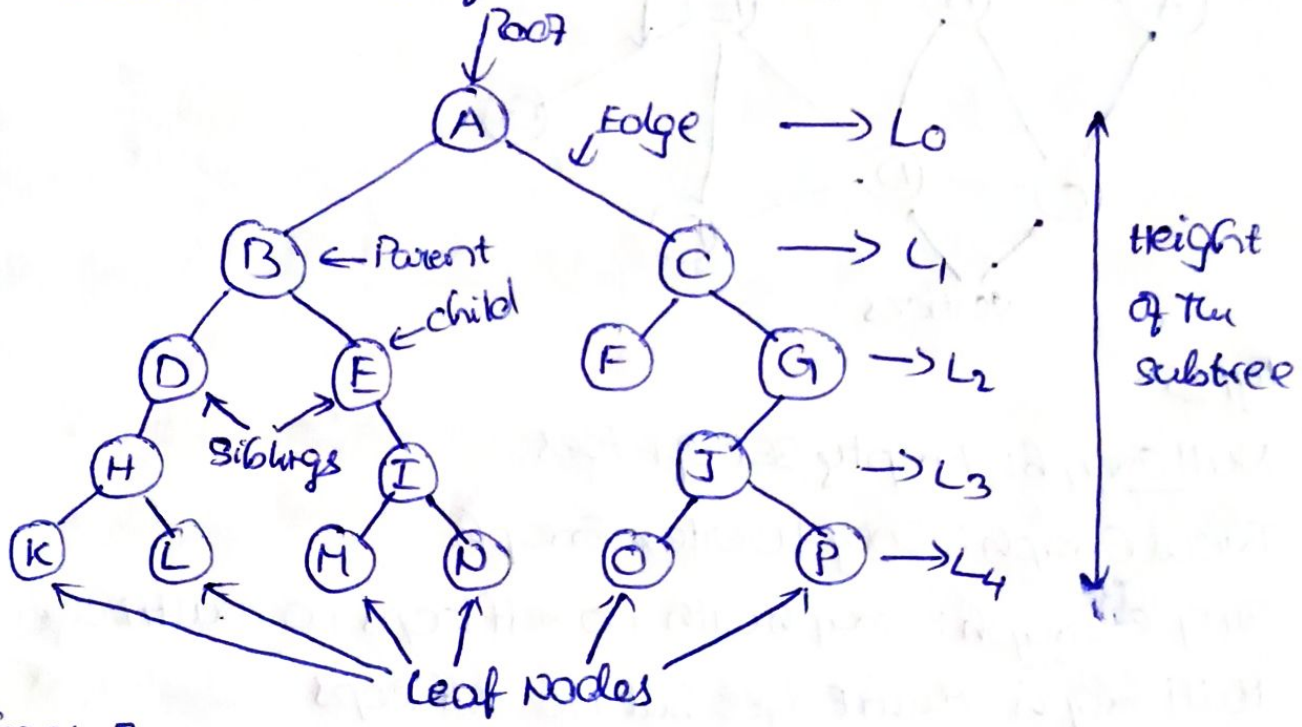
Applications
→ BFS in Graphs
→ File downloading Queues
→ Job scheduling operations
→ Handling interrupts generated by user apps
→ CPU/Job/Disk Scheduling

Non Linear Data Structure

* Data elements not arranged in sequential order.
* Insertion & Removal is not that easy, hierachial dependency between data items

Trees

* collection of Nodes such that each node of tree stores a value and list of references to other nodes (children)



Types of Tree:

Binary Tree: 1 Parent node → atmost 2 children

Binary Search Tree: can maintain sorted list of nums

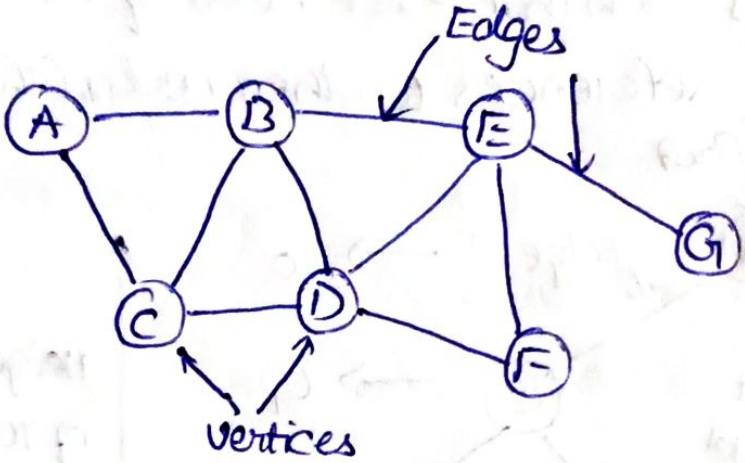
AVL Tree: Self Balancing Binary Search tree, Each node have Balance Factor (-1, 0, 1)

BTree: Similar to AVL Tree, Each node can have more than two children.

Graphs

* Finite nodes or vertices and the edges connecting them.

$G = (V, E)$ => set of vertices & Edges



Types

Null Graph: Empty set of Edges

Trivial Graph: Only 1 vertex Graph

Simple Graph: Graph with no self loops no multi Edges

Multi Graph: Multi Edges but no self loops

Pseudo Graph: self loops & Multi Edges

Non Directed Graph: non directed Edges

directed graph: directed edges

connected graph: atleast a single path b/w every pair of vertices

disconnected graph: atleast on pair of vertices doesn't have edge

Regular Graph: all vertices have same degree

Complete graph: all vertices have an edge b/w every pair of vertices

Cycle graph: Atleast 3 vertices and edges form a cycle

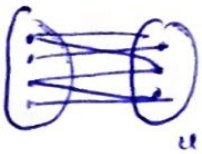
Cyclic graph: atleast one cycle exists

Acyclic graph: zero cycles

Finite / Infinite Graph: Finite / Infinite num of vertices / Edges

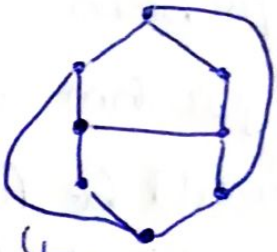
Bipartite Graph: vertices can be divided into 2 sets

SetA vertices can be connected to SB vertices



Planar graph: If we can "draw in a single plane" with "two edges intersecting to Each Other"

Euler graph: All vertices are even degrees



Two Edges Intersect to Each Other

Basic Operations of DS

1. Traversal
2. Search
3. Insertion
4. Deletion
5. Sorting
6. Merge
7. Selection
8. update
9. Splitting

Important points:

* All Data Structures are Examples of ADT
* If user want to store the data into memory, we provide array of LL, user don't know the implementation that is main idea of Abstract Datatype

Applications of DS

→ Rep info of DB

→ search through org data

→ Generate the Data

→ Encrypt & Decrypt the data

Algorithm

* set of rules required to perform calculations or some other problem solving operations Especially by computer.
  → Flow chart
  → Pseudo code

Why Algorithms

* Scalability:
  we need to scale down a real world big prob into small steps, which helps us to analyze the problem
* Algorithm says that each and every instruction should be followed in specific order to do a specific task.

Algorithms should consider these while creating are
→ Modularity (Break problems into small chunks)
→ Correctness (Generate precise output with precise inp)
→ Maintainability (designed in simple structured way)

Some algorithmic approaches

① Brute Force algorithm:

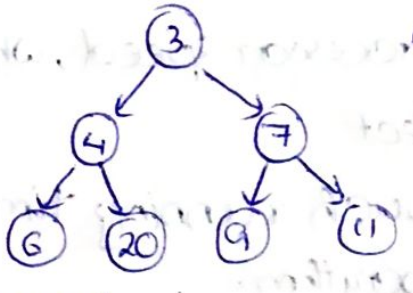
* searches all the possibilities to provide the required solution
  → Optimezed method: Take best solution out of all solutions
  → Sacrificing: stops at first solution, doesn't care about optimized or not

② Divide and conquer

→ Divide bigger problem into smaller and solve them and merge the output's to get result of solution

③ Greedy Algorithm

→ Problem solving approach of making the locally optimal choice at each stage with the hope of finding a globally optimum.
→ May not provide Globally optimized value
→ one of the case that would fail



Greedy Answer: 3+7+21 = 21
(local optimal)
Optimal answer: 3+4+20 = 27

When to use?

* Global optimum can be reached using local optimum
* optimal solution to a problem contains opt sol of subprob

Applications

* Activity selection prob
* Huffman coding
* Job sequencing
* Fractional knapsack
* Prim's Min spanning

④ Dynamic Programming
* Breaks problem into subproblems
* Stores results of subproblems using memorization
* Find the optimal solution out of these subprob
* Reuse the result of subproblems, to not execute them

⑤ Branch and bound algorithm:
* can be applied to only integer probles
* method of solving optimization problems by breaking them down into smaller sub problems and bounding function to eliminate subproblems that cannot contain the optimal solution.

6) Backtracking
* Solves the problem recursively and remove the solution if it doesn't satisfy constraints of problem.

Algorithmic Analysis

@ Priori analysis: Concider processor speed, which gave no implementation Effect

⑤ Posterior Analysis: How much running time and space taken by the algorithm.

Algorithmic Complexity

① Time Complexity:
* amount of time req to complete the execution
* Denoted by Big(O) notation.
* number of steps it may take to complete execution

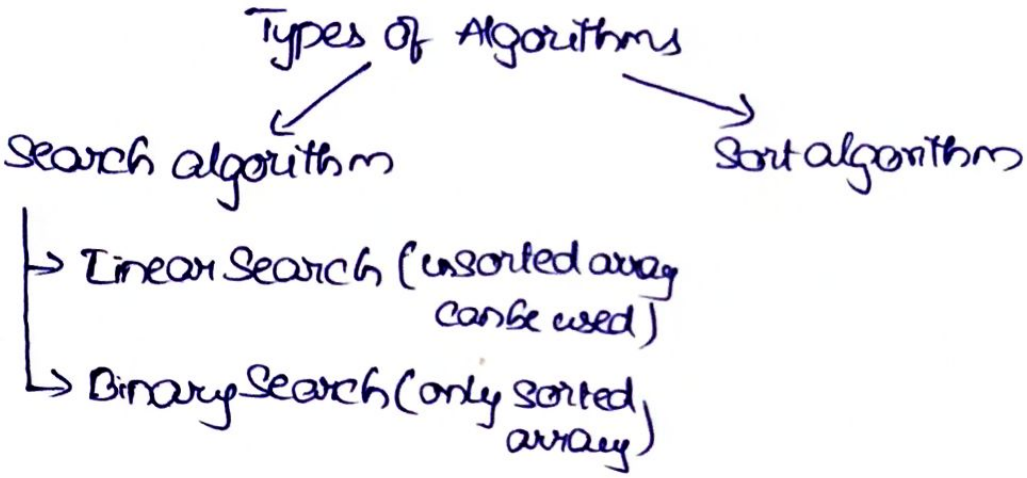
```
sum=0
for i in 1 to n:
    sum=sum+i;      } O(n)
return sum;         → O(1)
```

② Space Complexity:

* Amount of space required to solve a problem and produce an output.
* Expressed in Big O notation.
* space is required by store program instructions, constant values, variable values, function calls, jumping statements etc.

Auxilary space: Extra space required by algorithm, excluding the input size.

Space complexity: Auxilary space + Input size



Asymptotic Analysis

we choose the best algorithm, based on time complexity

Real time example:

If we need algorithm to add a element in front then possible solution could be by array, Linkelist But the best solution is through linked list because in array's we need to push all elements to right to add an element to the beginning array. But for linked list we need to add a node in front and tag the address with the second node.

* using Asymptotic analysis, we can conclude the average case, bestcase and worst case scenario of an algorithm.

Example: $f(n) = 5n^2 + 6n + 12$

For small values of $n \Rightarrow 12$ is best

For large values of n (worstcase) $= 5n^2$ is best

So T.C should be taken based on worst case

$n^3 \Rightarrow$ More Time complexity

$n^2 \Rightarrow$ comparitively less time complexity

If we have an algo which solves in n^2 & n^3 then we must go for n^2 as less time-C than n^3

Time Complexity

① worst case

② Average case

③ Best case

Asymptotic notations used for calculating T.C

* Bigoh Notation (O)
  - → provides the order of growth of the function
  - → provides an upper bound on a function
  - → It Ensures that program that we written will be less or equal complexity of our upper bound.
  - → provides worst time complexity
* Omega Notation (Ω)
  - → provides best case scenario
  - → opposite action to Bigoh Notation
  - → provides lower bound of an algorithm
  - → provides faster time that an algorithm can run.
* Theta Notation (Θ)
  - → provides average case scenarios.
  - → provides realistic time complexity of an algorithm

Common Asymnotations

Constant	O(1)	⇒ Adding to the front of a linked list
O(logN)	log	⇒ Finding an entry in a sorted array
O(N)	linear	⇒ Finding an entry in an unsorted array
O(NlogN)	nlogn	⇒ Sorting n items by 'divide & conquer'
$O(N^2)$	quadratic	⇒ Shortest path b/w 2 nodes in graph
$O(N^3)$	cubic	⇒ Simultaneous linear equations
$O(2^N)$	exponential	⇒ Tower of Hanoi problem

Pointer

* Pointer is used to points the address of the value stored anywhere in computer memory
* obtaining such value using pointer is called dereferencing the pointer :

Mainly useful in

1. Lookup tables
2. Traversing String
3. control tables
4. Tree Structures

Pointer can be used in ways

1) Pointer arthematic
   ⇒ ++, --, +, -
2) Array of Pointers
3) pointer to pointer
4) passing pointer to function
5) Return pointer from func inc

a → [10] → value
2000 → address

b → [ ]
3000

b = &a → [ ] → [10]
(b points a) 3000 → 2000

Pointer

```
int a=5
int *b;
b=&a;
```

Pointer to Pointer

```
int a=5;
int *b;
int **c;
b = &a;
c = &b;
```

Structure

* Structure is a composite data type that defines a grouped list of variables that are placed under one name in block of memory.
* Allows different variables to be accessed by using a single pointer to structure.

```
struct structure_name {
    datatype member_1;
    datatype member2;
    :
    datatype member;
};
```

```
struct structur_nm e1, e2, e3;
```

Advantages

* Can hold variables of different datatypes.
* reuse the data layout across programs.
* used to implement LL, stacks, queues, trees, graphs etc.

Arrays

* collection of similar types of data items stored in contiguous memory locations.
* randomly accessable by using its index number.
* Elements are of same datatype & same size
* First element stored at smallest Memory location, second element is calculated based on given base address and size of data element.

```
int array[10] = {35, 33, 42, 10, 4, 19, 27}
↑     ↑    ↑                    ↓
Type  Name Size                 Elements
```

* Index starts with 0
* Sorting and searching a value in array is easier, fast to process values quickly, good for storing multiple values in a single variable

*Types of indexing (zero based, one based, n based)

Byte address of element

A[i] = base address + size * (i - first index)

Insert an element at particular index

```
x=50; //value
pos=4; //index
n++;
for(i=n-1; i>=pos; i--)
    arr[i]=arr[i-1];
arr[pos-1]=x;
```

Time and space complexity of various array operations

Time Complexity:

	Average Case	worst case
Access	O(1)	O(1)
Search	O(n)	O(n)
Insertion	O(n)	O(n)
Deletion	O(n)	O(n)

Space Complexity

worst case O(n)

Advantages

* single name for group of variables of same type.
* Easy traversing and accessing through index.

Disadvantages

* Memory allocation is static, the size of array cannot be altered.
* Memory inefficient usage if we not use whole statically allocated

2D arrays

Declare 2D array: int arr[max_rows][max_columns]

Value can be accessed by: arr[i][j]

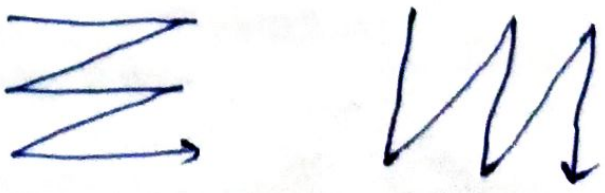
we must declare the size of the 2Dimensional array while we do initialization. This is not same while the case of single dimensional array.

int arr[2][2] = {0,1,2,3}

Mapping 2D array to 1D array:

row idx \ col idx	0	1	2
0	(0,0)	(0,1)	(0,2)
1	(1,0)	(1,1)	(1,2)
2	(2,0)	(2,1)	(2,2)

→ col idx ① Row Major ordering ② col major ordering



Locate address of 2D array : row/column Major

int A[3][4]

datatype size => 2

A	0	1	2	3
0	100	102	104	106
1	108	110	112	114
2	116	118	120	122

3×4 (m×n)

Row major order:

Addr(A[2][1]) = [Lo + [i×n + j] × w] (indices starts from 0)

Lo => Base address

Addr(A[2][1]) = [100 + [2×4 + 1] × 2]

Addr(A[2][1]) = 100 + 18 = 118

Addr(A[i][j]) = Lo + [(i-1)×n + (j-1)] w (index starts with 1)

Column Major order

Add[A[i][j]] = Lo + [j×m + i] × w (index starts with 0)

Add[A[1][2]] = 100 + [2×3 + 1] × 2

= 100 + 14

Addr(A[1][2]) = 114

If we want locate an element of a 2D array, then we must either use row major order / column major order

Linked Lists

* series of connected nodes
* nodes that are randomly stored in memory

* Last node of the list contains pointer to null
* Every link contains a connection to another link.

Why linkid lists over array?

* Array the size should be prefixed, expand the size of array at runtime is not possible, Elements should be stored in contiguous memory Inserting an element shift all its predecessors.
* LL allocates memory dynamically, elements can be stored non contiguously and linked together with help of pointer. memory allocated at runtime.

Declare the structure of Node

```
class Node:
    def __init__(self, dataval=None):
        self.dataval=dataval
        self.nextval = None
```

we use classes to create a custom datatype similar to what we use for structure in c programming

Types

1. Singly linked list
2. Doubly linked list
3. Circular singly linked list
4. Circular doubly linked list

Advantages

* Dynamic data structure
* Ins & Del is easier
* Memory efficient
  - Memory used is dynamic and flexible
* Implements other DS like Stacks & Queues

Disadvantages

* Memory usage more than array (pointer size is xtra than array)
* Traversal and random accessing is complicated
  - Difficult, if we want to access last element, we need to traverse the whole LL. no possible of random accessing.
* Reverse traversing is needed for Backtracking but occupies more

Applications

* polynomial representation & sparse matrix repr
* Implement
  - → Stack → Queue → Tree
  - → Adjancy list → Dynamic Mem alloc

Time Complexity

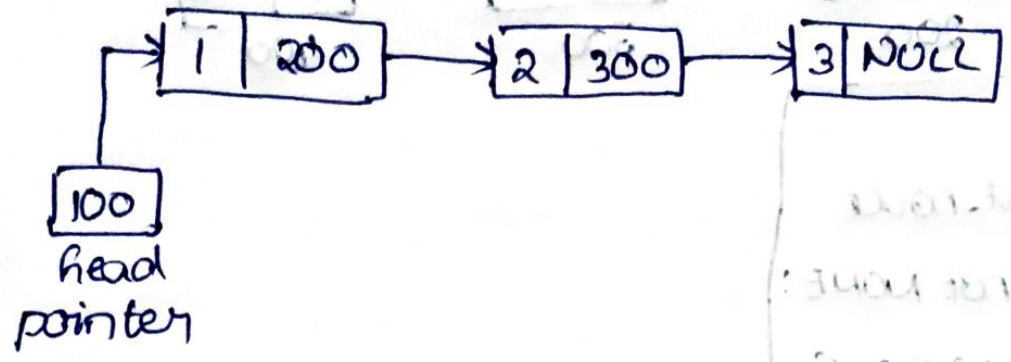
	Avg case	worst case
Insertion	O(1)	O(1)
Deletion	O(1)	O(1)
Search	O(n)	O(n)

Space Complexity

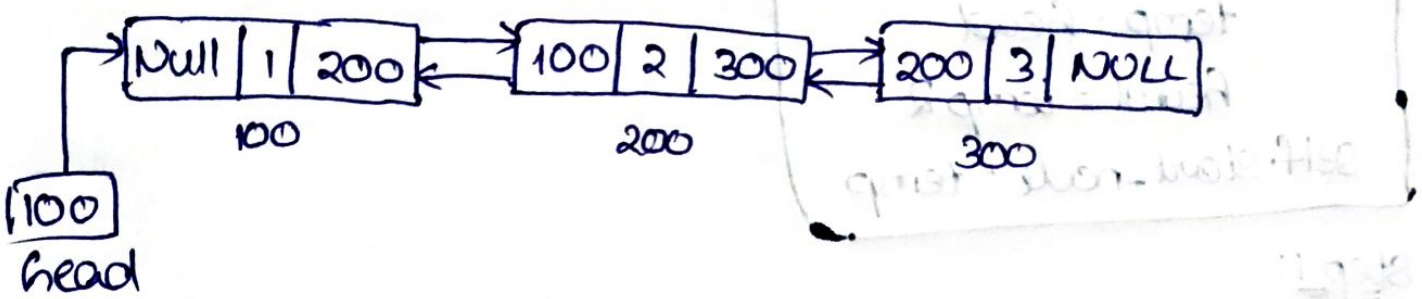
* Space Complexity of Linked List is O(n)

Types of Linked Lists

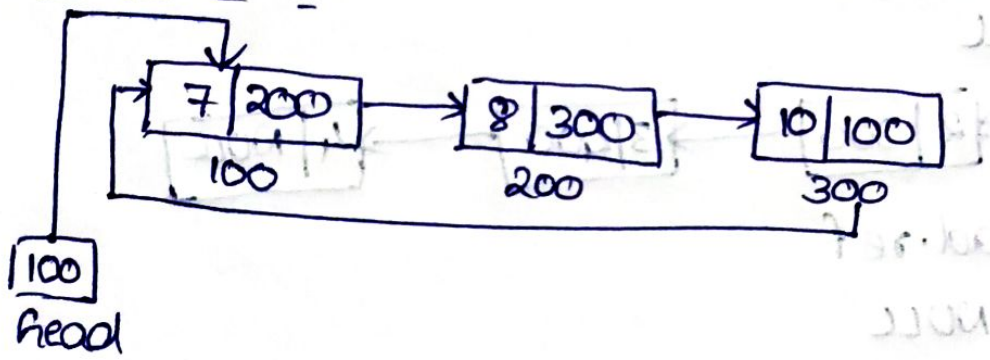
Singly Linked List (Single Link)



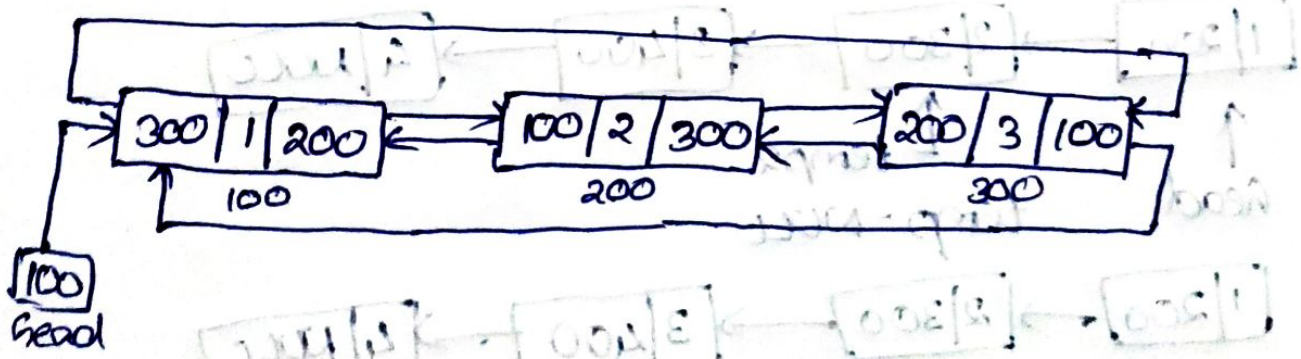
Doubly Linked List



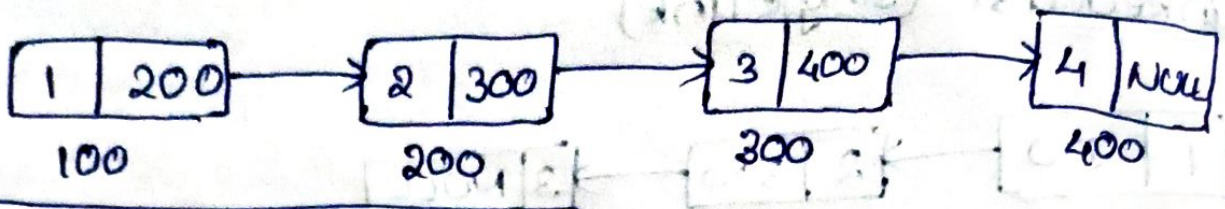
Circular Linked List



Doubly Circular Linked List



Reverse Linked List




```
temp = NULL
head = self.start_node
while head is not NONE:
        temp2 = head.ref
        head.ref = temp
        temp = head
        head = temp2
self.start_node = temp
```

step 1:

head = start_node

temp = NULL

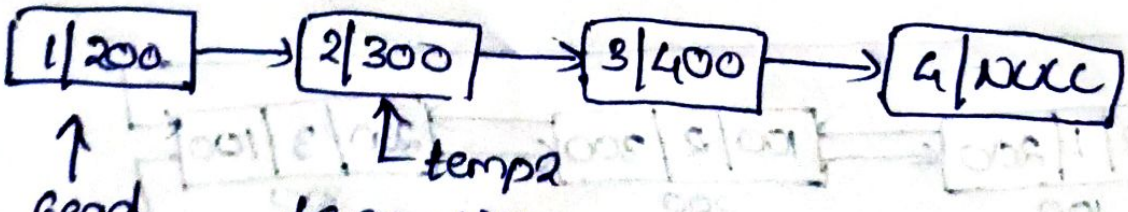
temp2 = head.ref

head.ref = NULL

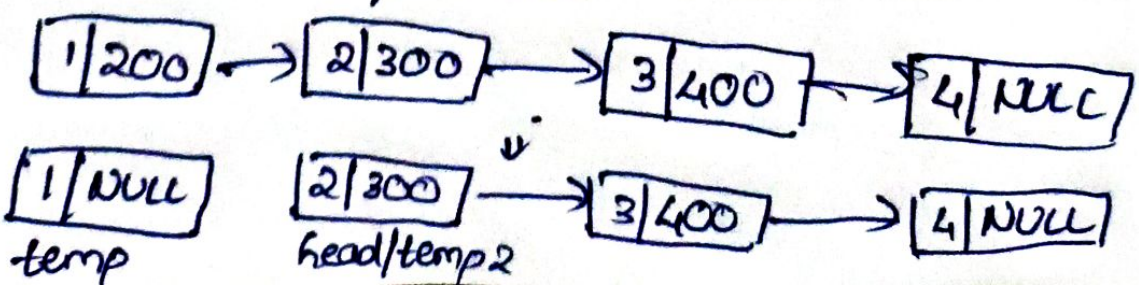
temp = head

head = temp2

step2



step3:



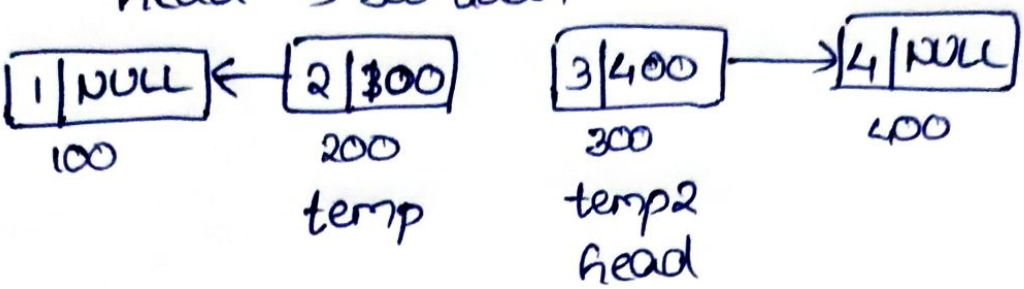
step4:

temp2 => 300 addr

head.ref => 100 addr

temp => 200 addr

head => 300 addr



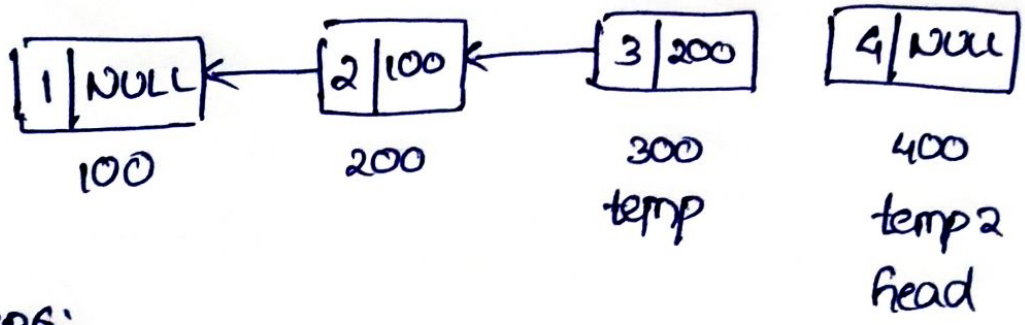
step5:

temp2 = head.ref

head.ref = temp

temp = head

head = temp2



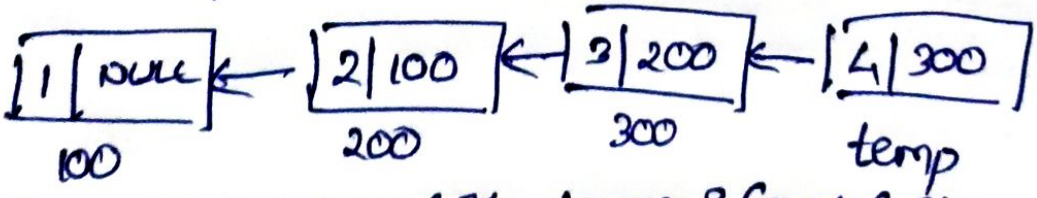
step6:

temp2 = head.ref

head.ref = temp

temp = head

head = temp2



At the end of the loop both temp2 & head both points to same node