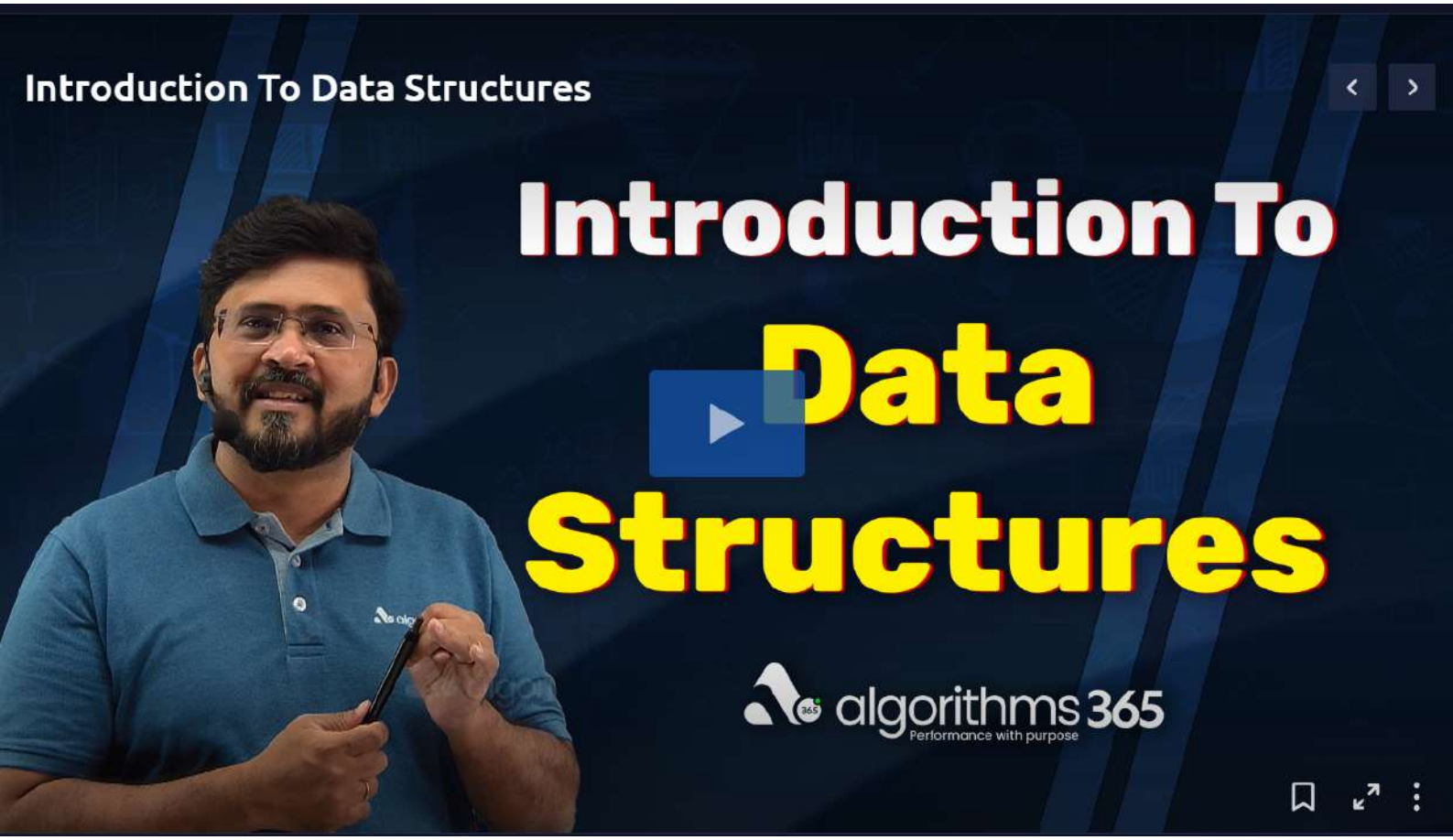

Introduction To Data Structures
Introduction To
Data
Structures
algorithms 365
Performance with purpose

Day 27 16-9-25 Tuesday

Introduction To Data Structures.

What are Data structures ?
Why we need DS ?
Why we study time and space complexity ?
Types
- Linear
- Non-Linear.

What are Data Structures ?

* Organizations of Data. * DS is centeral part

CRUD operation
- C create
- R Read
- U update
- D Delete

* Data Structures are specialized way of organizing, managing and storing data so that it can be accessed and modified efficiently

Why we need Data Structure ?

* To store the complex Data

* we need DS's because they allow us to olganize, store and manage data efficiently so that opeeations like searching, inseeting, deleting and updating can be peefomed quickly.

* They optimize the use of memory and processing power, making software systems scalable, reliable and efficient

classmate Date Page

Why we study time and space complexity?

To evaluate the efficiency of algorithms.

* Time Complexity: tells us how the execution time of an algorithm grows as the size of input increases

* Space Complexity: tells us how much memory the algorithm requires for execution

* Eg: when you open any application and if it takes more time to load the content then you will end up by going to another application

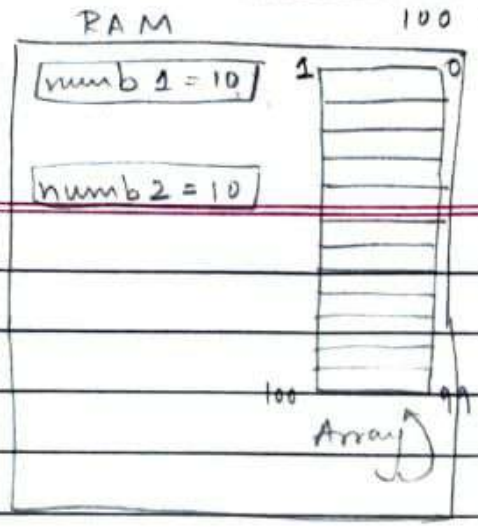
Types of Data Structures

Linear and Non Linear

Arrays
Linked list
Array list
Stack
Queue
Dictionary, Sets
~~Sets~~ Tree, Heap

Linear → Data arranged sequentially
- Array
- Linked List
- Stack
- Queue
- Dequeue / priority Queue

non-linear → Data not arranged sequentially instead, it forms a hierarchy (or) network
- Tree
- Graph

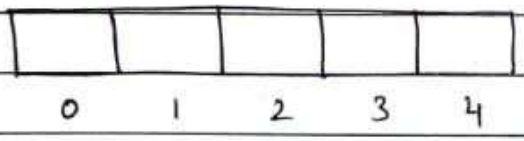


Tuesday - 16 - 09 - 2025 Day 27

Why learn Data Structures ?

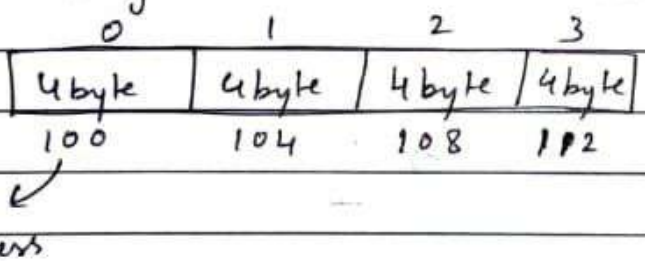
[DSA] : Data Structure & Algorithms

Arrays : Homogeneous type Data structure



* Same data type elements should be stored

Int = 4 byte



If a code takes more time to run → loss for Company
→ Loss for Customer

So learn how to write a optimized and good quality code

Array Support primitive Data type

classmate
Date
Page

Single linked list. :

Disadvantage of Array :

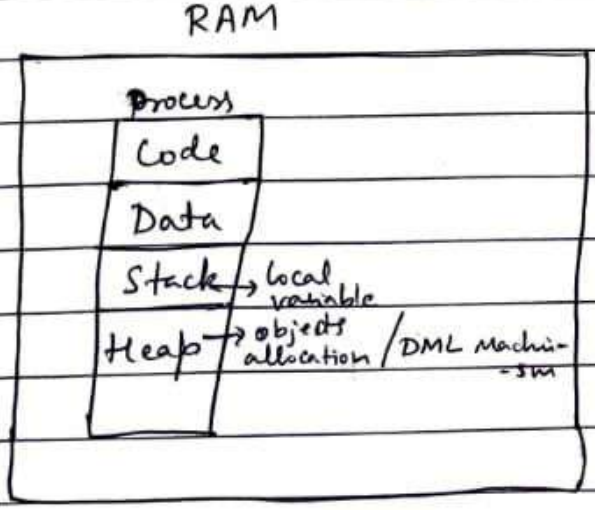
(1) * you cannot add the extra element to the Defined Size of array.
* If you try to add the element
EXCEPTION - ArrayOutOfBound.
Index

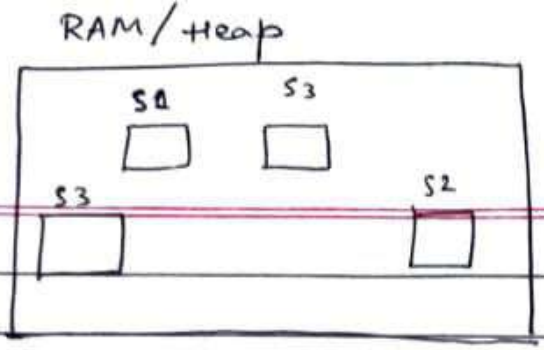
(2) [0 to 99] Size array But you will provide only ~~to~~ two to three value then the remaining memory is waste. That memory is not utilized by others.

(3) when you know the size of array and you are going to use that Array Completly at that time use Array.

*** pros and Cons of a perticular problem

* Single linked list used Dynamic memory allocation





Eg: IRCTC works on DMA.

No RAM

if no seats left

- out of Memory Exception

Allocate ↴

C/C++ - Malloc/calloc

free - to free the memory

Garbage Collector

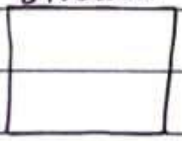
Garbage Collector 2 major problem

(1) * when it will be looking for the unused things to free the memory it is a long process and CPU mainly concentrate on that and your Application will get slow.

(2) until GC doesn't clear the waste memory your overall application will be slow.

But Comparitivly the problem like slow App launch and device hanging these two (above) problem's are OK to handle

How does Garbage collector get to know it should clean the unused object.



count

1, 2, 3 +

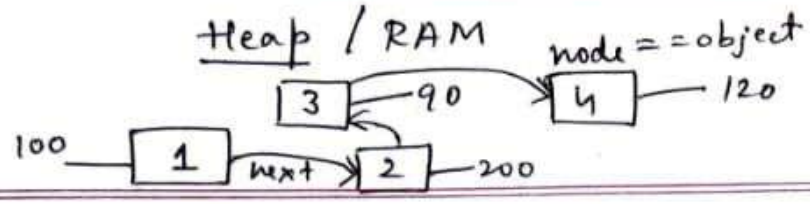
-1 2 3

at last it will become zero (0)

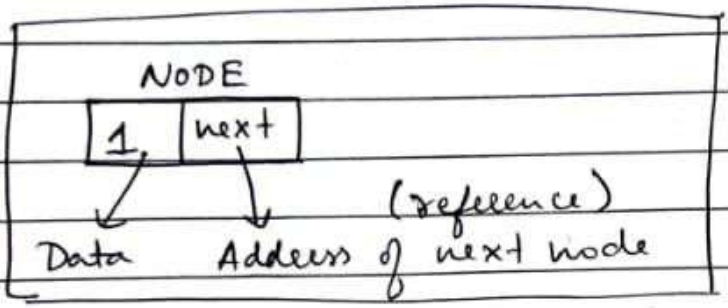
the zero mentioned object will get cleared by GC

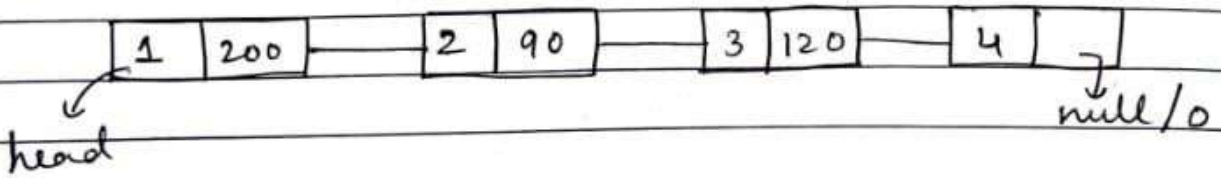
Memory Leak

Manged code → Java which contains Garbage Collector



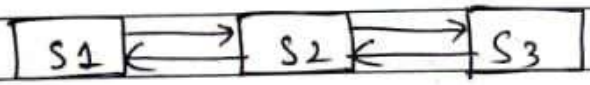
* each node is an object
* each node has an unique address





* No Reverse in Single linked list

Doubly linked list : reverse is Allowed



Operations

SLL, DLL, CLL, Stack, Queue

1. Create
2. Add / Insert
3. Delete
4. Search
5. print
6. Update

IT → How to Manage Information & Transfer the Information & How to process that Information

SLL

Create

node	
data	next
10	null

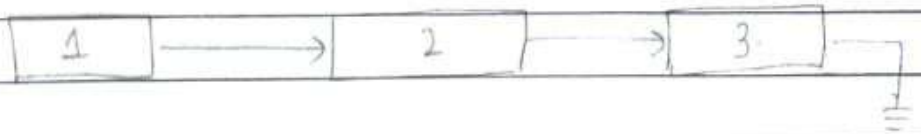
→ object

Heap

RAM

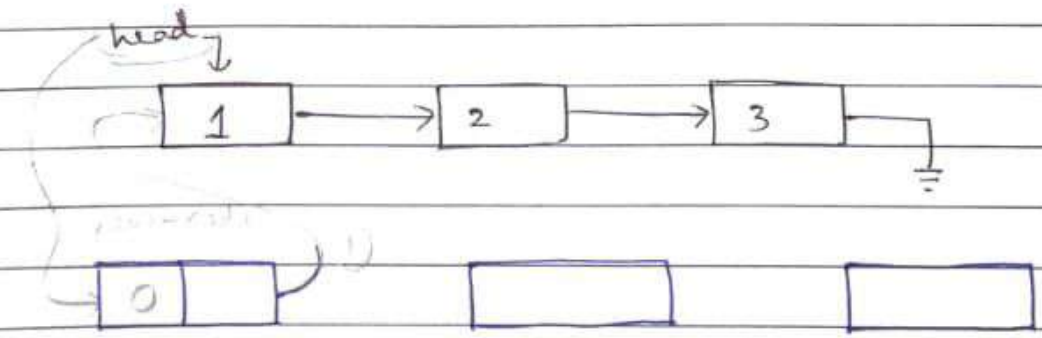
Age

operations



head

Insert → at the beginning
→ at the middle / any position
→ at the end

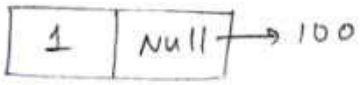



```
Node {
    int data
    Node next
}

Node new_node = new Node(1);
```

data → [1 | null] → next

new_node . next = head ;



head = new_node ;

Empty : head which points to NULL

Insert at end

```
currentNode = head
while (currentNode != Null) {
    currentNode = currentNode . next
}
```