MEMORY MANAGEMENT IN JAVASCRIPT

YOUR DATA
STORED ?

1.WHAT IS STACK ?

2.WHAT IS HEAP ?

3.WHAT IS RAM ?

PROUBLEM STATEMENT :

MEMORY OPTIMIZATION ?

WHAT IS MEMORY MANAGEMENT ?

Q. Your Phone → Any Application → WhatsApp, Instagram, Pubg etc. → Where are stored ?

Ans. Phone → Hard Disk or SSD Stored

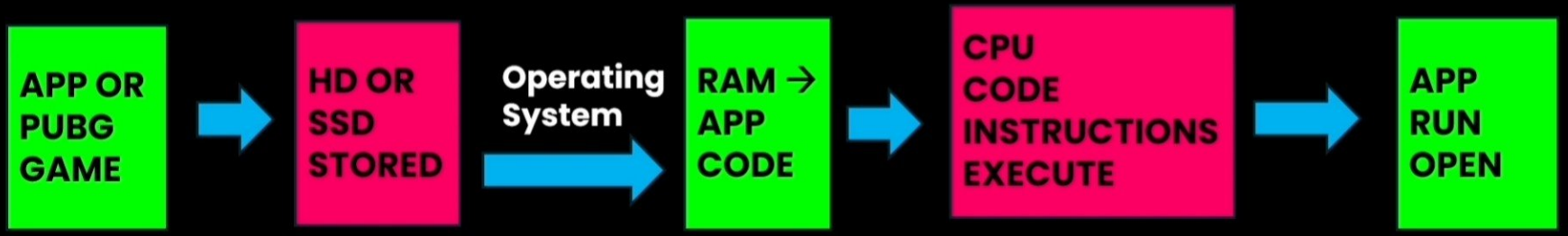
Note : Instagram, whatsapp, pubg → Application → At the end Code

PROCESS : RUN / OPEN → APP

Q. Your Phone → Pubg Game / Application → Run or Open ?

Ans. Pubg Game / Application → Code → Operating System → Ram(Random Access Memory) → Ram → App Code → Instructions → CPU → Execute → All Instructions → App Run / Open

Note : Code → HD / SSD → LOAD → RAM → RUN / EXECUTES → INSTRUCTIONS → CPU



WHY NOT D.T → SSD TO CPU ?

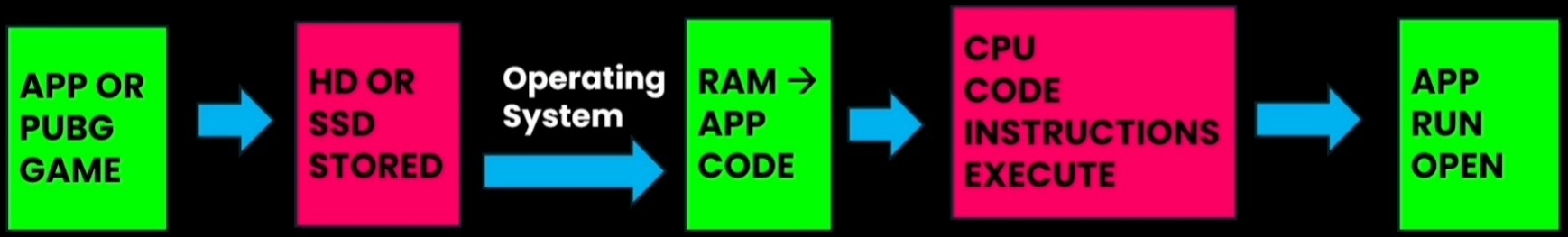
1. Hard Disk or SSD → Slow
2. HD / SSD TO CPU → Data Transfer Process → Slow
3. RAM To CPU → Data Transfer → Very Easy & Very Fast
4. Computer → HD / SSD → 1 TB, 2 TB ....
5. Computer → RAM → 4 GB, 8 GB, 16 GB, 32 GB ...
6. RAM → Create → Very Costly

WHY NOT D.T → SSD TO CPU ?

7. Hard Disk → Perment Storage → Your Phone → Switch Off → 5 Minutes → Switch On → Your Data Safe → Not Delete

8. But RAM → Tempory Storage → 5 Apps Like Pubg, Whatsapp, Insta, YT, Netflix Open → Phone Switch Off → 5 Minutes → Phone Switch On → Data / 5 Apps → Remove From Ram

Ex. Pubg Game 1 Hour → Play → But I Close Pubg Game → Ram Remove



PROUBLEM :

Q. Managing Data Without Unique Identifiers :

Assume → RAM → Size = 16 Byte

You → Write Some Code

let a = 10;

let b = 20;

Code Run → RAM → Code → Memory → Where ?

Few Minutes Later ..

a = 20;

b = 25;

Method : Ram → First Change

Note : 20 → TR Corner & BL Corner → Confusion → Which one → 25 ? → TR = 25 → Mistake

We → Uniquely Identify → Exact Location find → Method Use

Byte-Addressable Memory :

Assume → RAM → Size = 16 Byte

First We Make → 16 Byte Ram → Byte Addressable

B.A → Total = 16 Byte

Each & Every Byte → I will give 1 Address

Note : 1 Byte = 8 Bit

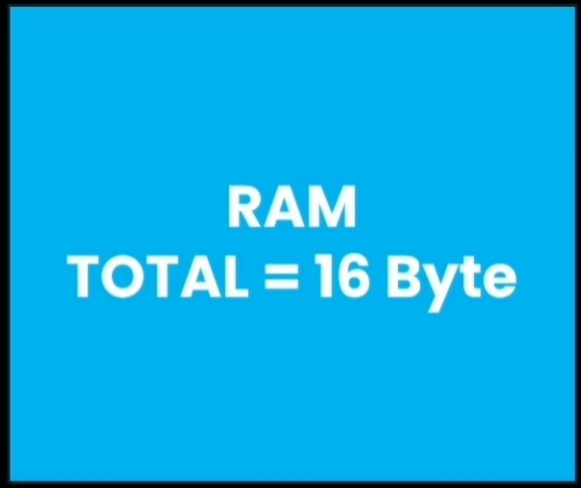


Chart Table

Unit	Symbol	Conversion	In Bytes
Byte	B	Base Unit	1 B
Kilobyte	KB	1KB = 1024 B	2^10 B
Megabyte	MB	1MB = 1024 KB	2^20 B
Gigabyte	GB	1 GB = 1024 MB	2^30 B
Terabyte	TB	1 TB = 1024 GB	2^40 B
Petabyte	PB	1 PB = 1024 TB	2^50 B
Exabyte	EB	1 EB = 1014 PB	2^60 B
Zettabyte	ZB	1 ZB = 1024 EB	2^70 B
Yottabyte	YB	1 YB = 1024 ZB	2^80 B

Byte Addressable Memory

Byte Addressable – Each and Every Byte → I will Give 1 Address → Uniquely Identify → 1 Byte = 8 Bit

Row & Column → Devide

Every Block → Size = 1 Byte → Ex. Block 0, 1, 2, 3, 4 ... = 1 Byte

Block 0	Block 1	Block 2	Block 3
Block 4	Block 5	Block 6	Block 7
Block 8	Block 9	Block 10	Block 11
Block 12	Block 13	Block 14	Block 15

Mapping Mechanism

Managing Data with Unique Identifiers :

We Have Ram → Size = 16 Byte

Assume → 1 Number = 1 Byte Of Memory = 8 Bit

Number → Binary Convert → At the end

Code :

a = 10;

b = 20;

a = 20;

b = 25;

Where are Stored This Data ?

Where are Stored This Data ?

Mapping Mechanism : a & b – Check → Address Present ?

Current Present Address : a = 10 → 2 Block & b = 20 → 8 Block

Future Update Address : a = 20 → 2 Block & b = 25 → 8 Block

RAM
16 Byte

0	1	2	3
		10	
20			

0	1	2	3
		20	
25			

Mapping Mechanism

Mapping Mechanism : Variable Corresponding → Address

Variable a = 10 → 2nd Block Address Map

Variable b = 20 → 8Th Block Address Map

Note : Data Stored → Variable Corresponding → Which Address Present

16 Byte Ram = We Want 16 Address

32 Byte Ram = We Want 32 Address

RAM 16 Byte

0	1	2	3
		10	
20			

Memory Requirement

16 Byte Ram = We Want 16 Address

32 Byte Ram = We Want 32 Address

16 Byte Address Stored = You need 4 Bit of Requirement

Because 4 Bit → 2^4 = 16 Unique Address Generate

EX. Total = 16 Unique Address / Different Address

0000

0001

0010

0011

...

1111

Behind The Scene

Q. Where are this Mapping Table Present ?

We Write Some Code :

a = 10; → User / Programmer → a ke andar 10 daaldo

But Computer Behind The Scene → Binary

a = 10; Replace → 0010 = 10; → 0010 ke andar 10 daaldo

0010 Address → 10 Stored

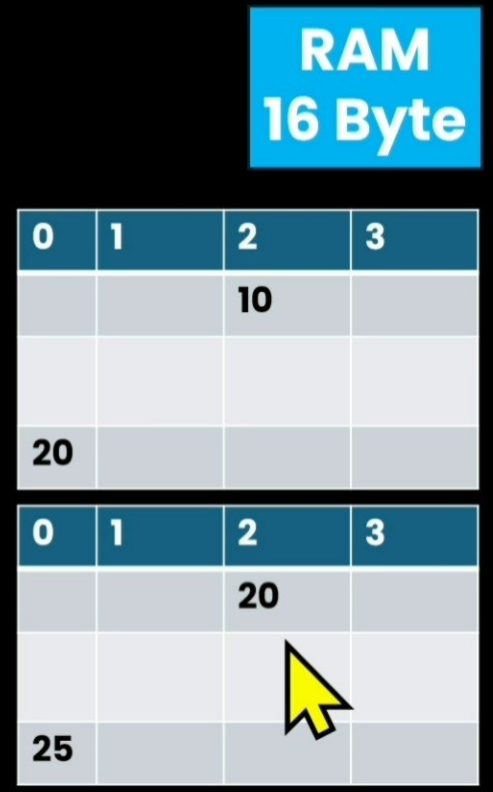
Similarly for b = 20; Replace → 1000 = 20; → 1000 ke andar 20 daaldo

1000 Address → 20 Stored

Note : a, b → Concepts remove ❌ → X Address = Y Stored ✅

0010 Address → 20 Stored

1000 Address → 25 Stored



Stack Vs Heap Memory :

RAM → 2 Portion → 1st = Stack, 2nd = Heap

STACK	HEAP

Difference BTW Stack & Heap :

Stack → Whenever You → Memory Allocation

Ex. I Want to store some data → 1st Data : Bottom 1 → Memory , 2nd Data : Bottom 2 → Memory ....

Stack Concept → Data : 1 After Another → Data Allocate → 1, 2, 3 ...

STACK	HEAP
c = 30	
b = 20	
a = 10	

Difference BTW Stack & Heap :

Note : Stack Size → Very Small → KB → Ex. 100KB TO 1 MB

Note : Heap Size → MB To GB → Increse Or Decrese → Ex. 200MB, 300MB, 400MB ....

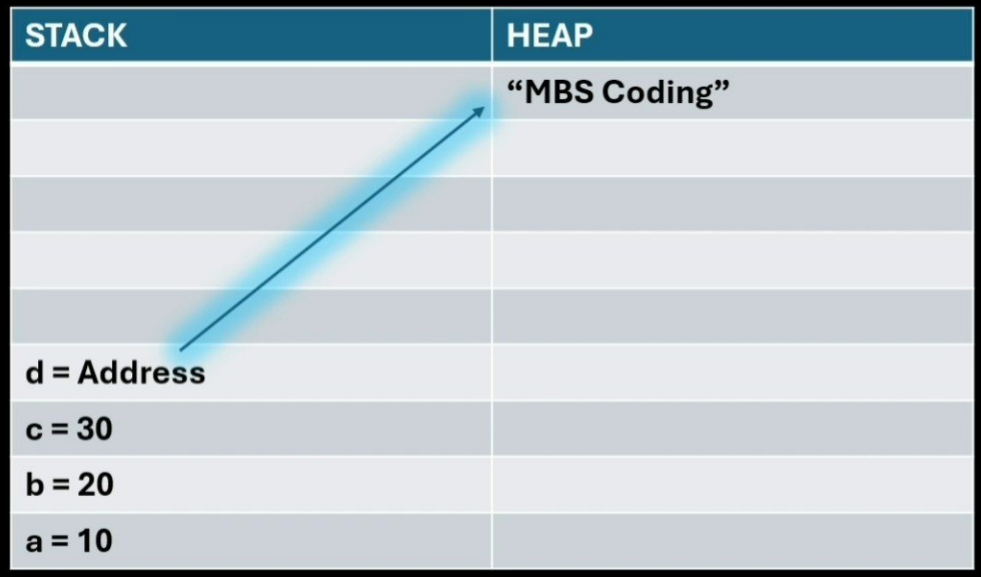
STACK	HEAP
c = 30	
b = 20	
a = 10	

Difference BTW Stack & Heap :

Heap Data Stored ?

Ex. d = "MBS Coding" → String → Heap Stored

Note : Stack → Heap Address Stored → Ex. d Address → Stack Stored → d point out → Heap Data MBS Coding



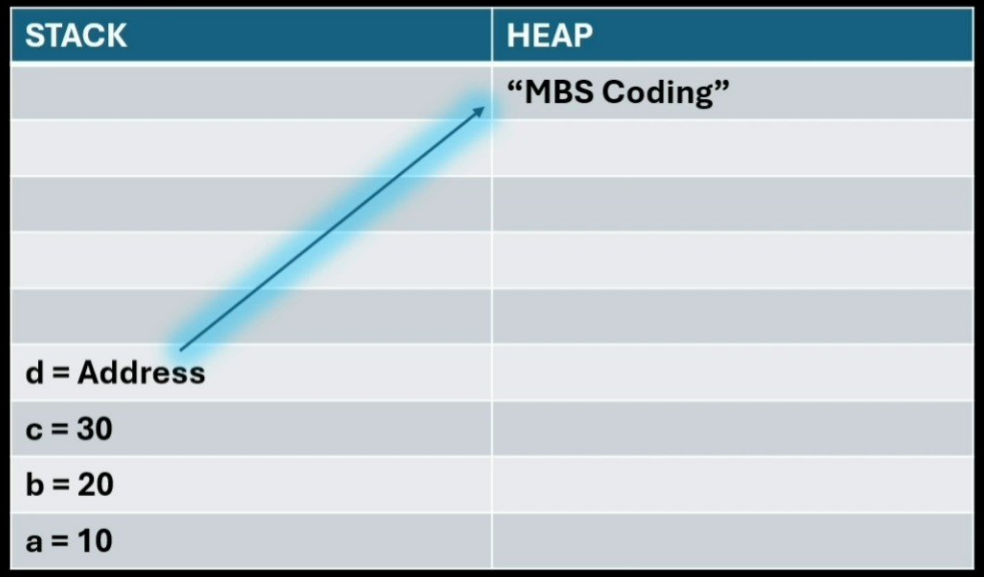
STACK	HEAP
	"MBS Coding"
d = Address	
c = 30	
b = 20	
a = 10	

Difference BTW Stack & Heap :

Note : Stack direct value → stored → Ex. 10, 20, 30 ..

Note : Stack → Address → Stored → Ex. 1000, 0001 ...

Data → Heap & Address → Stack ✅



STACK	HEAP
	"MBS Coding"
d = Address	
c = 30	
b = 20	
a = 10	

Difference BTW Stack & Heap :

Q. How are stored following data? Where are a,b,c,d Variables → Memory Present ?

a = 10;

b = 20;

c = 30 ;

d = "MBS Coding"

Ans. Location / Address → Binary → Stack 1 After Another

a = 10; Stored → Replace → 0000 Address / Location

b = 20; Stored → Replace → 0001 Address / Location

c = 30; Stored → Replace → 0010 Address / Location

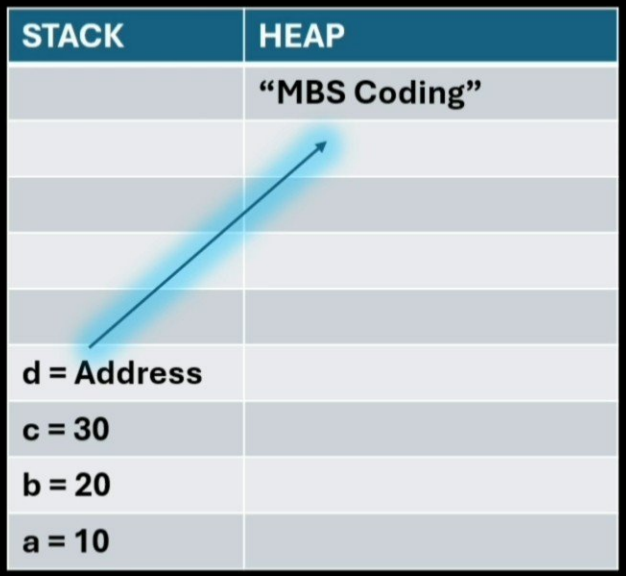
d = address → 1111 Address / Location

Note : d → Address ✅ = 1111 , Data ❌

What I will do → d = Address Follow Stack → Heap → Actual Data

Access → 2 Step Process – Stack Address, Follow Heap Data

Note : Heap Data → Those Addresses I Want to stored in STACK



Why Stack & Heap Exist :

STACK : EK KE BAAD EK DAAL DUNGA

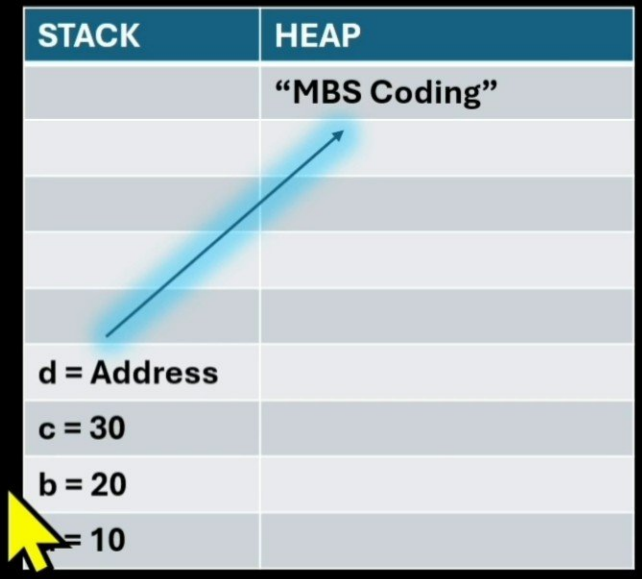
HEAP : MERI MERGI JAHAPE BHI SPACE KHALI DIKHA WAHAPE DAAL DUNGA

a = 10;

b = 20;

c = 30;

d = “MBS Coding”



Why Use Both Stack & Heap :

Ex. I want to stored some data

a = 10;

b = 20;

c = "SON";

Note :

1. Assume : 1 Number → 1 Byte of Memory
2. SON → 3 Character → Every Character → 1 Byte of Memory
3. SON Stored → 3 Byte Requirement

ASCII TABLE : SON → 3 Byte

S = 01010011 → Total 1 Byte

O = 01001111 → Total 1 Byte

N = 01001110 → Total 1 Byte

4. Stack → Each & Every → Block = 1 Byte Memory
5. Stack → Rule Defined → One After Another Data Stored

STACK	HEAP
N	
O	
S	
✕	
a = 10	

STACK	HEAP
N	
O	
S	
b = 20	
a = 10	

Why Use Both Stack & Heap :

Stack → Last In First Out → You Put Last Input → After That → Next Element Comes

Last Insertion → After That → Next Element Comes

Ex.

a = 10;

b = 20;

c = 30;

d = 40;

e = 90;

f = "SON";

Delete → b = 20

1 Place Shift Backward

STACK	HEAP
N.	
O.	
S.	
e = 90	
d = 40	
c = 30	
a = 10	

STACK	HEAP
N.	
O.	
S.	
e = 90	
d = 40	
c = 30	
❌	
a = 10	

STACK	HEAP
N.	
O.	
S.	
e = 90	
d = 40	
c = 30	
b = 20	
a = 10	

Why Use Both Stack & Heap :

Why this solution is not efficient ?

1. Data Shift → Backward → 1st, 2nd .....
2. It takes time
3. We want to do → All this things → Fast
4. Data Shift → Backward → 1 Data At a Time
5. Update → All
6. Process → Slow
7. Variable Address → Change
8. Variable 50 Times use → Address Change

Ex. console.log(b); → 1 Time Add. Change

...

...

50 Times Address Change

Stack → Proublem Hi Proublem

STACK	HEAP
N.	
O.	
S.	
e = 90	
d = 40	
c = 30	
a = 10	

STACK	HEAP
N.	
O.	
S.	
e = 90	
d = 40	
c = 30	
a = 10	

STACK	HEAP
N.	
O.	
S.	
e = 90	
d = 40	
c = 30	
b = 20	
a = 10	

Why Use Both Stack & Heap :

Similar Problem in HEAP

Ex.

let a = 10;

let b = 20;

let c = 30;

b = “Rahul”;

Note :

20 → R a h u l → Space Not Available

Space Find → R a h u l → Stored

b = 20 Remove → Last

a = 10 → 5th block → 0101 Address

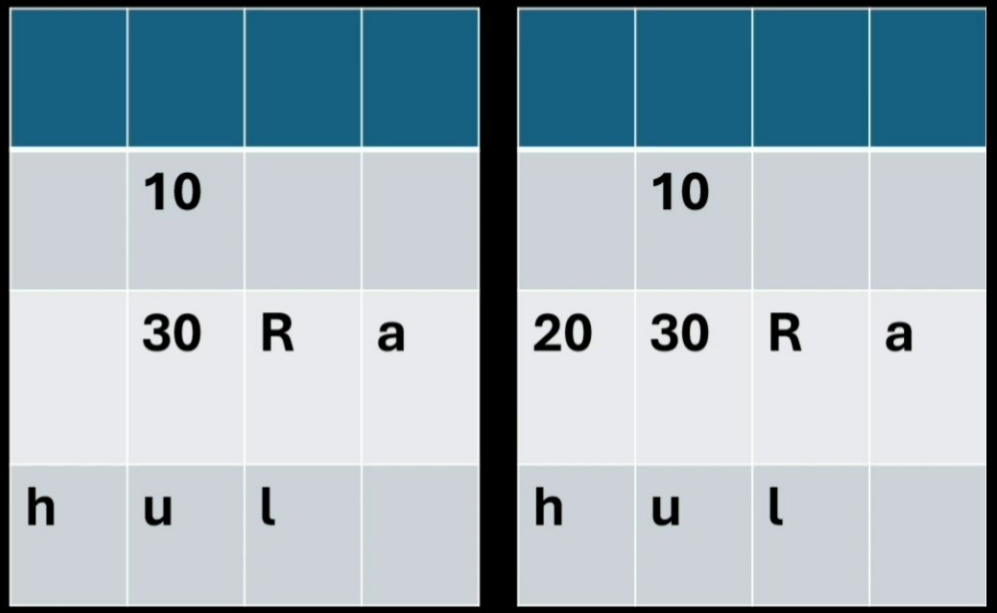
b = 20 → 8th block → 1000 Address

b code → Address Change → 10Th block R

Address = 1010

Proublem Hi Proublem

Heap Memory



Why Use Both Stack & Heap :

Proublem → Both → Stack & Heap

Location → Change → Address Change → Code Update

Fixed vs Dynamic Data Allocation :

Note : Fixed Size of Data → I will Put Those Data → STACK

Ex. Number

a = 10;

a = 50;

a = 100;

Fixed vs Dynamic Data Allocation :

Note : Those Data → Size → Not Fixed → Stack → Proublem

Ex. String → 1 Byte, 2 Byte, 3 Byte .....

Because String → Character → Less or More → Memory Less / More

a = “MBS”; → 3 Character → 3 Byte Memory

a = “MBSCODING”; → 9 Character → 9 Byte Memory

a = “Rahul”; → 5 Character → 5 Byte Memory

Simple Strategy : Stack & Heap

Q. Store Following Data

f = "Na"; → 6Th Block Address → 0110

Few Minutes Later : f = "Naya"; → 4 Byte Requirement → Space Find → Na → Remove

But Proublem → f Address Change → 0Th Block Start → 0000

If f Variable Code → 500 Times Present → 500 Places Change

STACK

30	50	40	90

HEAP

	10	N	a
	30	R	a
h	u	l	

HEAP

N	a	y	a
	10		
	30	R	a
h	u	l	

Simple Strategy : Stack & Heap

Variable f → Stack Stored

Na → Address → Stack Stored → 6th Block → 0110

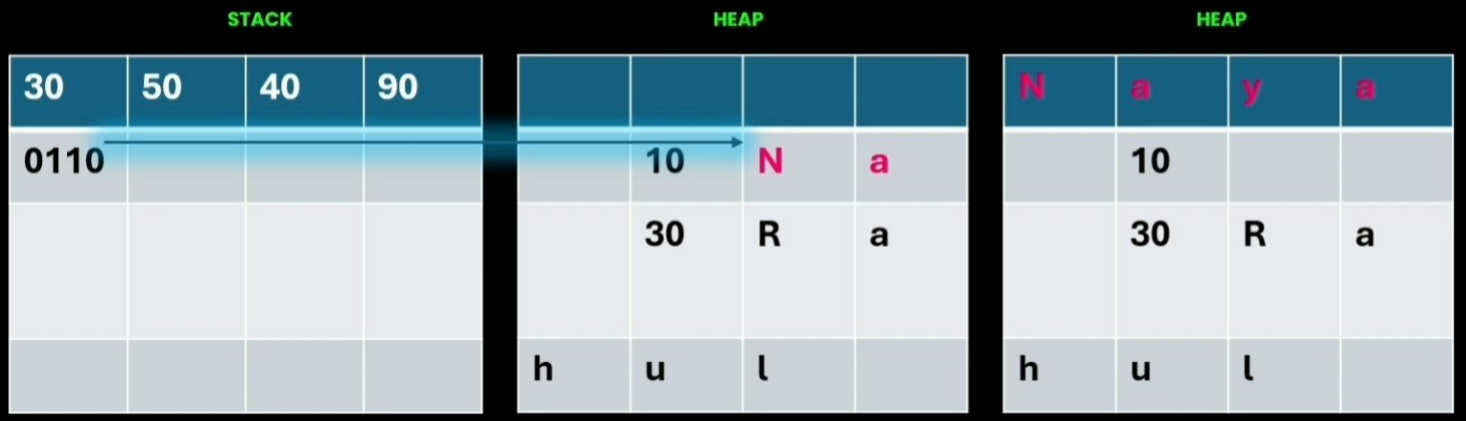
Stack 0110 Address (Memory Location) → Point Out → Na

Note : Address(0110) → Fixed Size → Stack 4TH Block Present

f = "Na" → 0110 = "Na" → If f want to access → Na

Step 1 : First Come Here 4Th Block Location Follow

Step 2 : Stack → Heap Data Na ✅



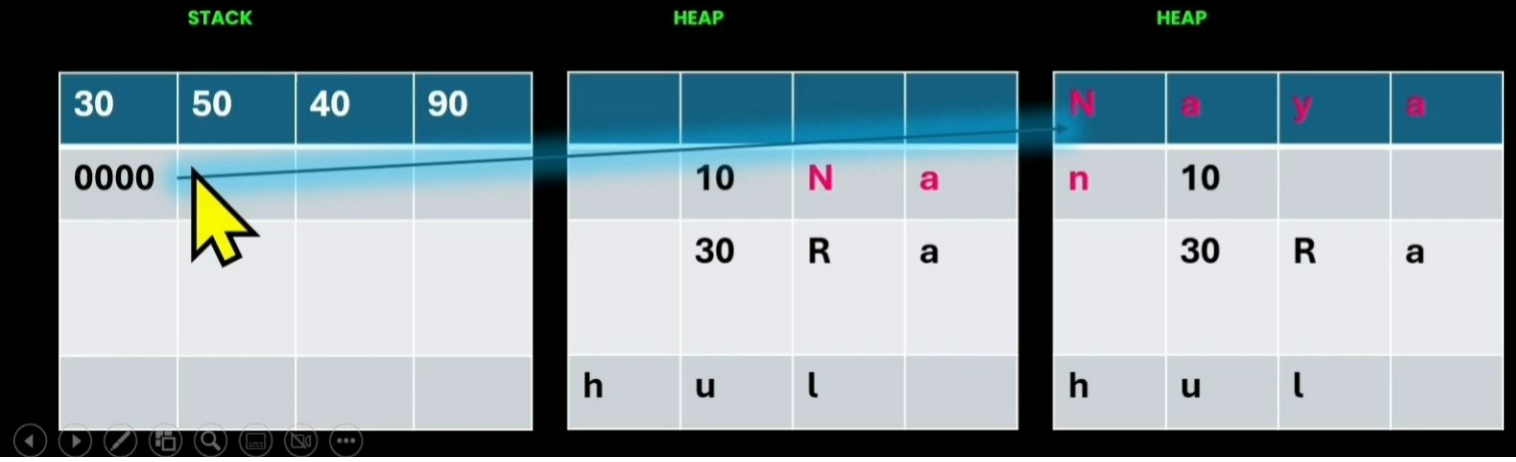
Simple Strategy : Stack & Heap

Now I want to do → f = “Na” To f = “Nayan”

Step 1 : Stack 0110 Address → Follow

Step 2 : Stack → Heap → Data = “Na” → Space Find → But 3 Byte Available → But I want → Nayan = 5 Byte Requirement → Starting 0 1 2 3 4 Block → Vecant Space ✅ → I Stored = Nayan → Na Remove

Note : I want to do only 1 Thing Change the address of 0110 to 0000 & 0000 Point out → N a y a n



Simple Strategy : Stack & Heap

Learning :

1. Size → Fixed → Ex. Number → Stack Stored → Why ? → Number Update → Number → Number Not Take Any Extra Space

2. String → Stack Not Stored → Why ? → Because String → Update → May be take Extra Space → It's Location may be change → If location change → Address also change

Note : Dynamic Data like string → Heap Stored & Static / Fixed Data like number → Stack Stored

Difference BTW Stack & Heap :

Rahul → Dynamic Data → Size Increase / Decrease

Dynamic Data Address → Stack Stored → 1111

Note : All Variables Ex. a,b,c,d → Memory → Stack ✅ → Number Only

Note : String Ex. Rahul → Dynamic Data → Heap Stored & Their Address → Stack Stored

Ex. a = 10 → 0000 = 10

Ex. b = 20 → 0001 = 20

Ex. c = 30 → 0010 = 30

Ex. d = "Rahul" → 1111 = "Rahul"

Dynamic Data in JavaScript :

Dynamic Data → Heap → Stored

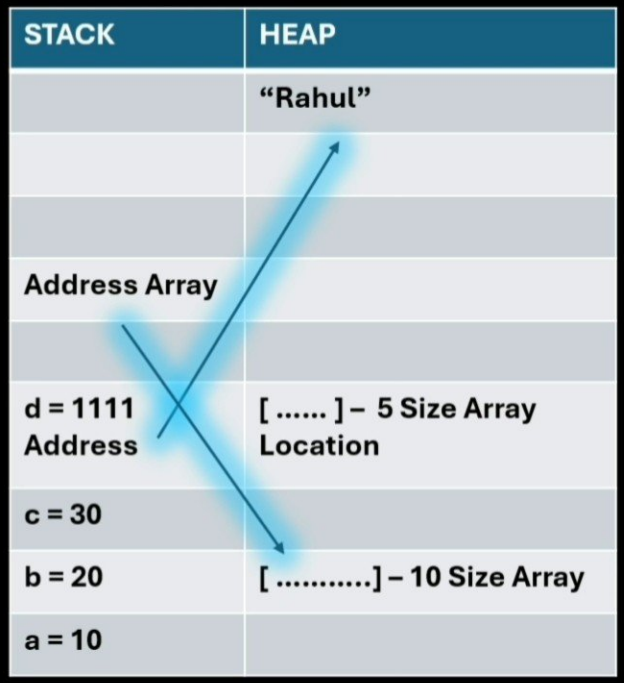
Dynamic Data Examples :

1. **String → Increase / Decrease → Data**
2. **Object → Increase / Decrease**
3. **Array → Element Push / Pop**

Ex. Initially → 5 Size Array → 10 Size Array

Heap → Space Find → Initally 5 Space → 10 ❌

Space Find Out → Down Space ✅ → Elements Stored ✅ & Their Address → Stack → Linked



JavaScript Memory Allocation :

1. Most of the time → You listened → Our System → 32 Bit OS / 64 Bit OS
2. 32 Bit → OS → Size Fixed = 4 Byte
3. 64 Bit → OS → Size Fixed = 8 Byte

Ex. 4 Byte Address Example

0000, 0001, 0010, 0011

4. System build → Ram → High

Address Size → 4 Byte = 32 Bit

2 ^ 32 Address Generate(Gigabyte) = Total 4 GB Ram

Easily Handle

Note : 4 GB → Ram → Byte Addressable ✅ →

Because 2 ^ 32 Different Diff. Byte → Address ✅

Similarly 8 Byte = 64 Bit

2 ^ 64 Address Generate = Greter Than 4 GB Ram

like 8,16,32 GB Ram Handle

JavaScript Memory Allocation :

Assume → 4 GB Ram

I Will Give → Each & Every Byte → Address ✅

Total = 2 ^ 32 Address Requirement

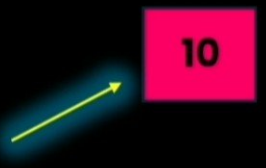
Note : 4 Byte Means → Address Size = 32 Bit ✅

Memory Allocation for Primitive Data Types :

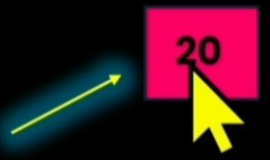
Q. Primitive Data → Memory Allocate ? → Stack or Heap ?

Code :

let a = 10;



a = 20;



Note : 10 → Data → Immutable – Not Change → Constant

Memory Allocation for Primitive Data Types :

Q. Primitive Data → Memory Allocate ? → Stack or Heap ?

Code :

let a = 10; → a replace → 0xAbc Address → Hex Dec Form

0xAbc = 10

let b = 30; → b replace → 0xAbd Address → Hex Form

0xAbd = 30

ADDRESS	STACK	HEAP
0xAbd	30	
0xAbc	10	

Memory Allocation for Primitive Data Types :

Q. Primitive Data → Memory Allocate ? → Stack or Heap ?

Now I want to do :

a = 50 → Go to → 0xAbc Address → 50 Banado

Note : JS Rule → Primitive Datatype → Immutable

10 → 50 ❌ → Immutable / Constant / Not Change

I have only 1 Choise :

New Memory Create → 50 → Address = 0xBca

a replace → 0xBca = 50

Proublem : a code 50 Times Present → 50 Times Address change

New Address put → Address = 0xBca

Efficient Solution → ❌

ADDRESS	STACK	HEAP
0xBca	50	
0xAbd	30	
0xAbc	10	

Memory Allocation for Primitive Data Types :

Q. Primitive Data → Memory Allocate ? → Stack or Heap ?

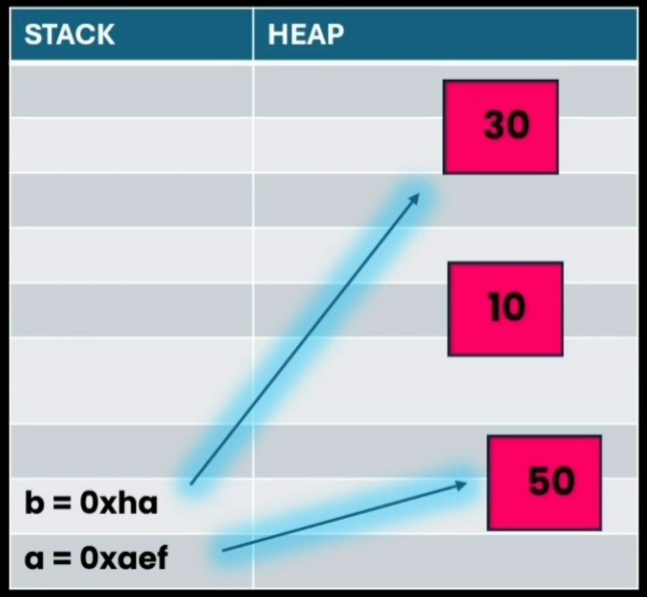
Best Solution :

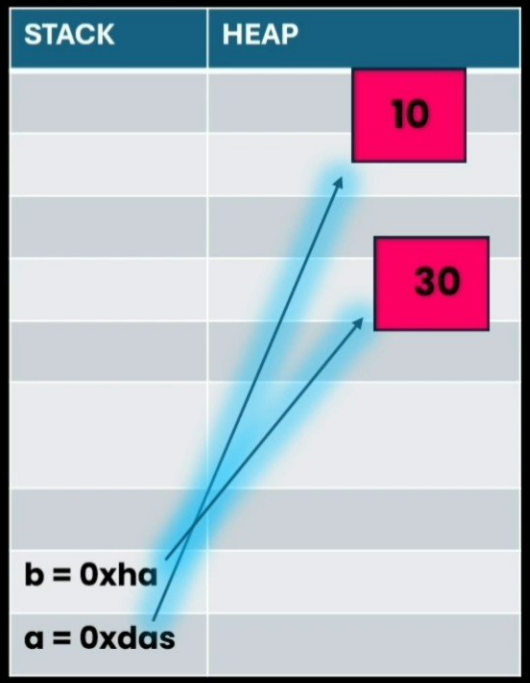
Primitive Data Type → Heap Sored → Their Address → Stack Store

let a = 10;

let b = 30;

a = 50;





Memory Allocation for Primitive Data Types :

Data → Premitive or Non Premitive → Heap Stored → Their Address → Stack Stored

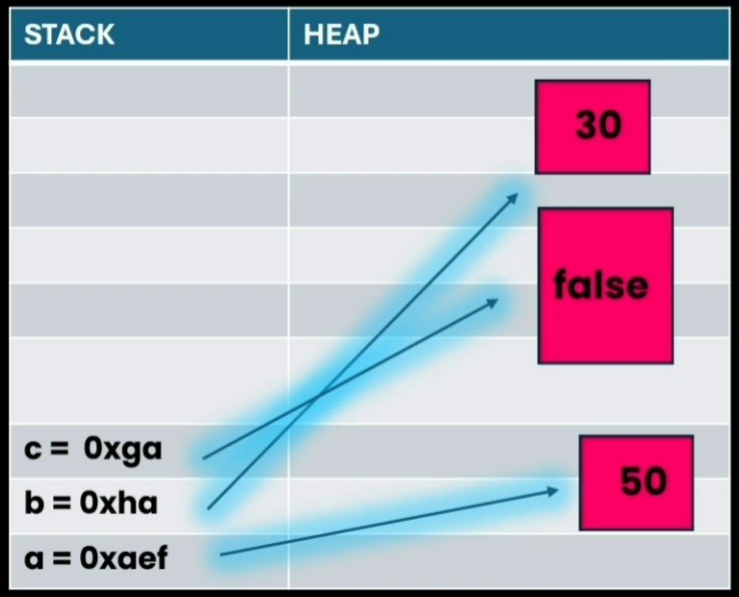
let a = 10;

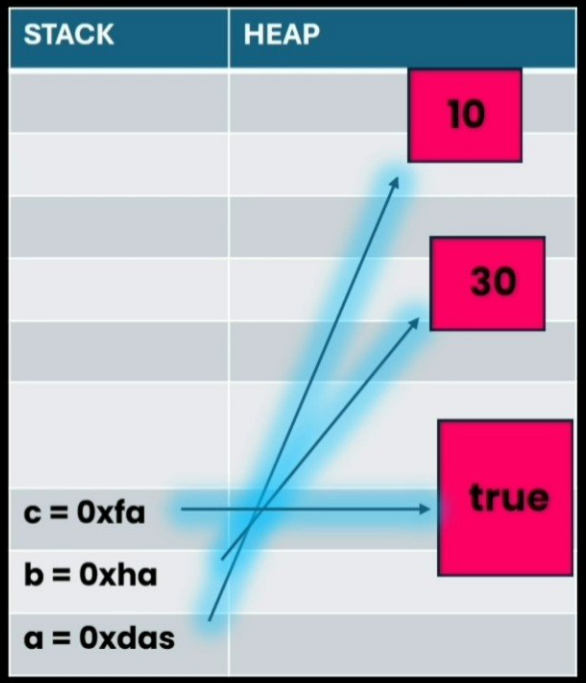
let b = 30;

a = 50;

let c = true;

c = false;

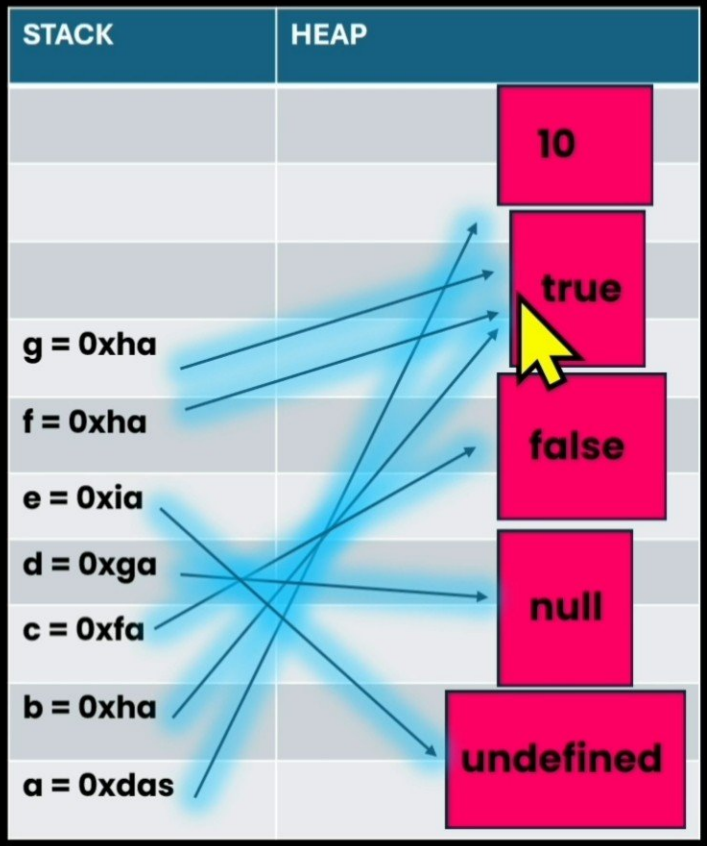




Memory Allocation for Primitive Data Types :

1. true, false, null, undefined → Program start → Memory Allocate ✅ → Fixed Location ✅ → Perticular Location ✅

```
let a = 10;
let b = true;
let c = false;
let d = null;
let e = undefined;
let f = true;
let g = true;
```



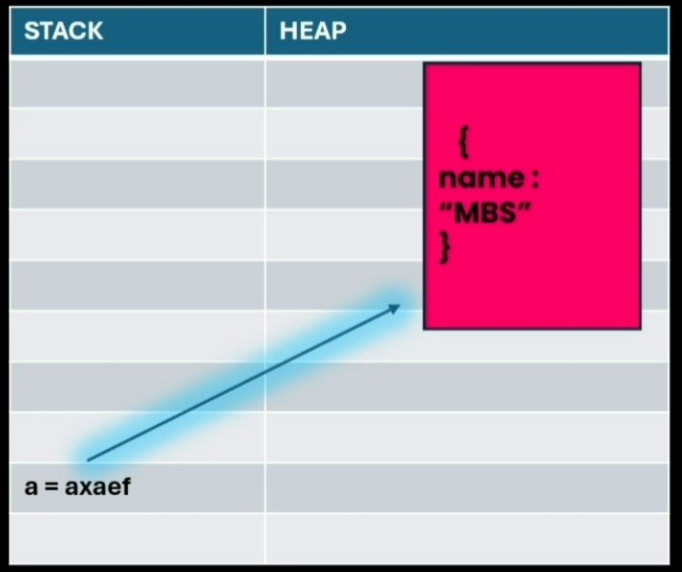
Memory Allocation for Primitive Data Types :

Object → Mutable → Data Change ✅

Memory Allocate → Heap → Space find ✅

```
let a = {
name : "MBS"
}

a = {
name : "Rahul",
age : 25,
category: "gen"
}
```



Garbage Collection : Cleaning Up Unused Memory

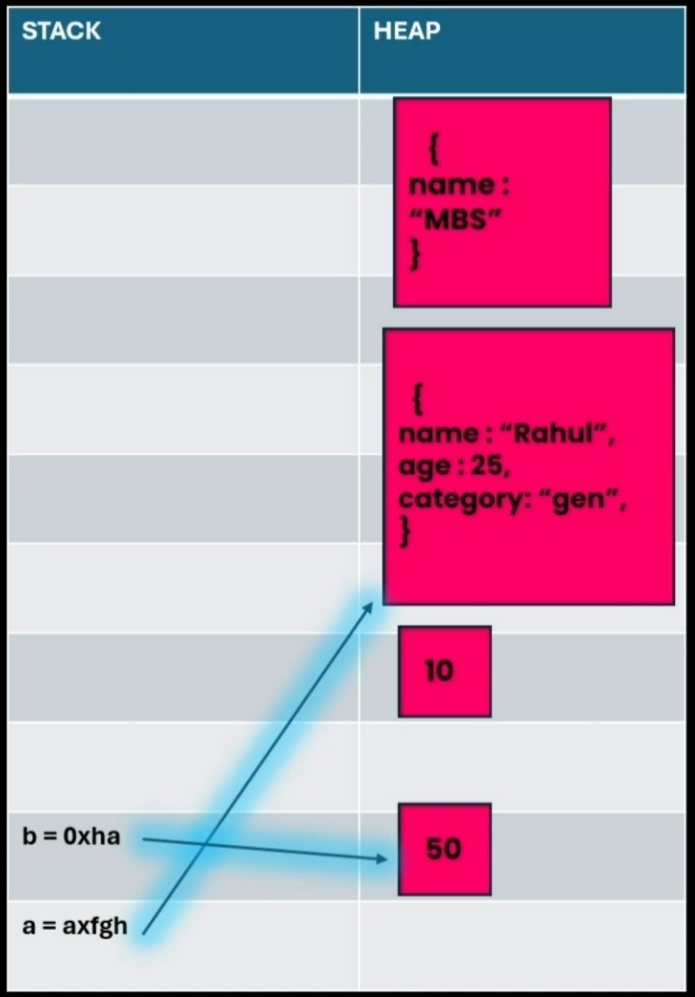
Garbage Collector → Heap Extra → Data delete → JS Handle ✅

But in CPP → Yout have to write → delete keyword

```
let a = {
name : "MBS"
}
a = {
name : "Rahul",
age : 25,
category: "gen"
}
let b = 10;
b = 50;
```

Note : 10 & { name : "MBS"} → No one point out

delete automatically → JS ✅ → Heap space available ✅



Garbage Collection : Cleaning Up Unused Memory

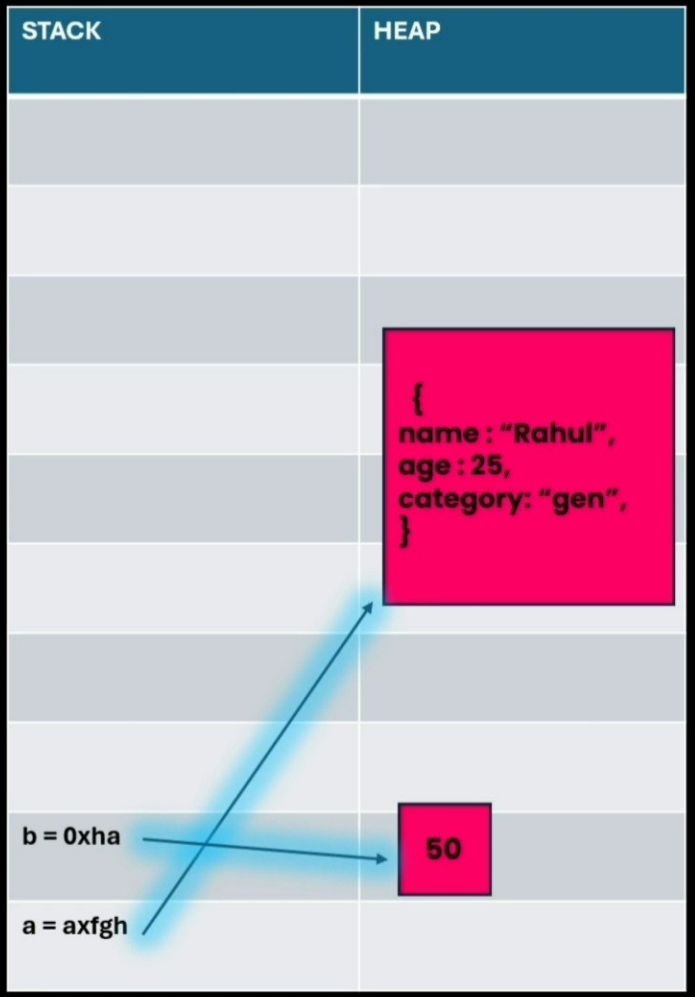
Garbage Collector → Heap Extra → Data delete → JS Handle ✅

System → Ram Limited → Extra Data Increase → Delete →

Memory Clean ✅

Space ❌ Program crash ✅

Ex. Recursion Code → Stop Condition Not Write → VS Code Crash



Loop Optimization :

We want Best Possible Solution

Ex. Loop → i = 0; to i = 100;

i = 0, 1, 2, 3, 4, 5, 6, 7, 8 .....

Code :

```
for(let i = 0; i < 100; i++){
console.log(i);
}
```

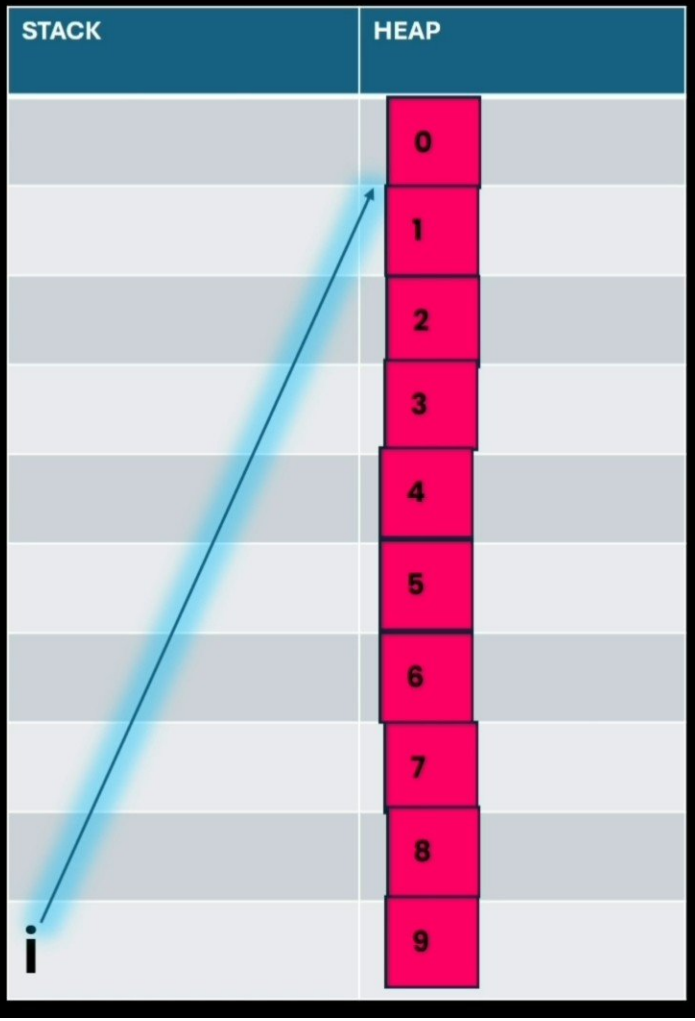
Loop Slow → 100 Times Loop Run → System Slow

Stack → i

1,2,3,4,5 ... Heap Store → New Memory Create → Immutable

Heap Location Free Avaiable ✅ → Heap Time → More

Stack → Less Time → 1 After Another



Loop Optimization :

We want Best Possible Solution

Ex. Loop → i = 0; to i = 100;

i = 0, 1, 2, 3, 4, 5, 6, 7, 8 .....

Code :

```
for(let i = 0; i < 100; i++){
console.log(i);
}
```

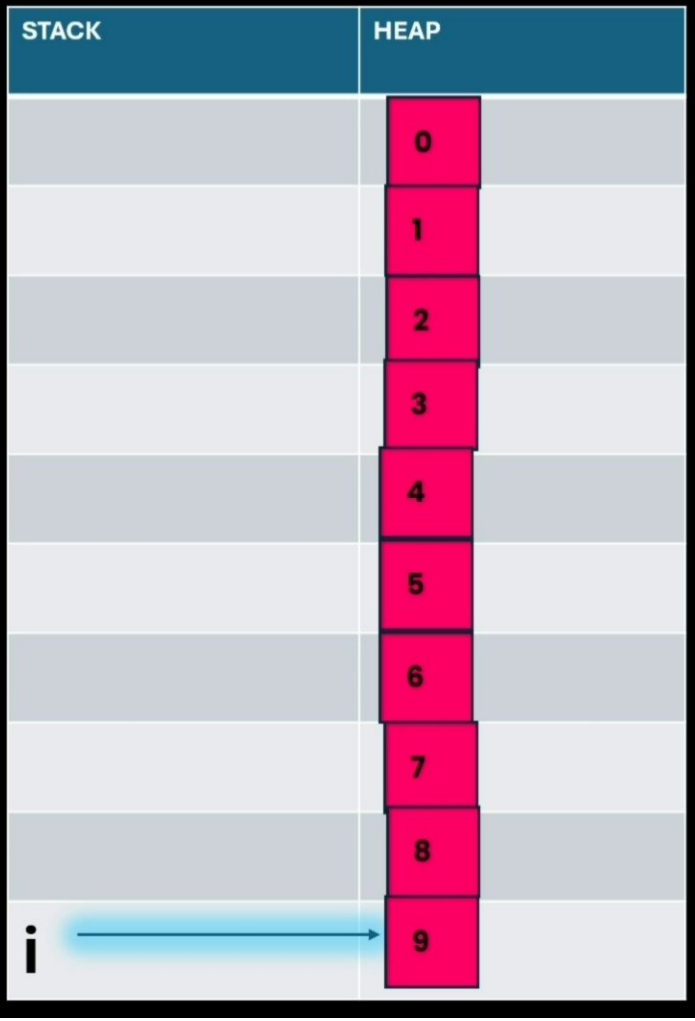
Loop Slow → 100 Times Loop Run → System Slow

Stack → i

1,2,3,4,5 ... Heap Store → New Memory Create → Immutable

Heap Location Free Avaiable ✅ → Heap Time → More

Stack → Less Time → 1 After Another



Loop Optimization :

We want Best Possible Solution

Memory Create → Process → Extra Time

Start → New Array Create → Big Array Size = 1 Lack

0	1	2	3	4	5	6	7	8	9

1000 1008 1016 1024 .....

Byte Addressable(Each & Every Byte)

→ 8 Byte of memory consume

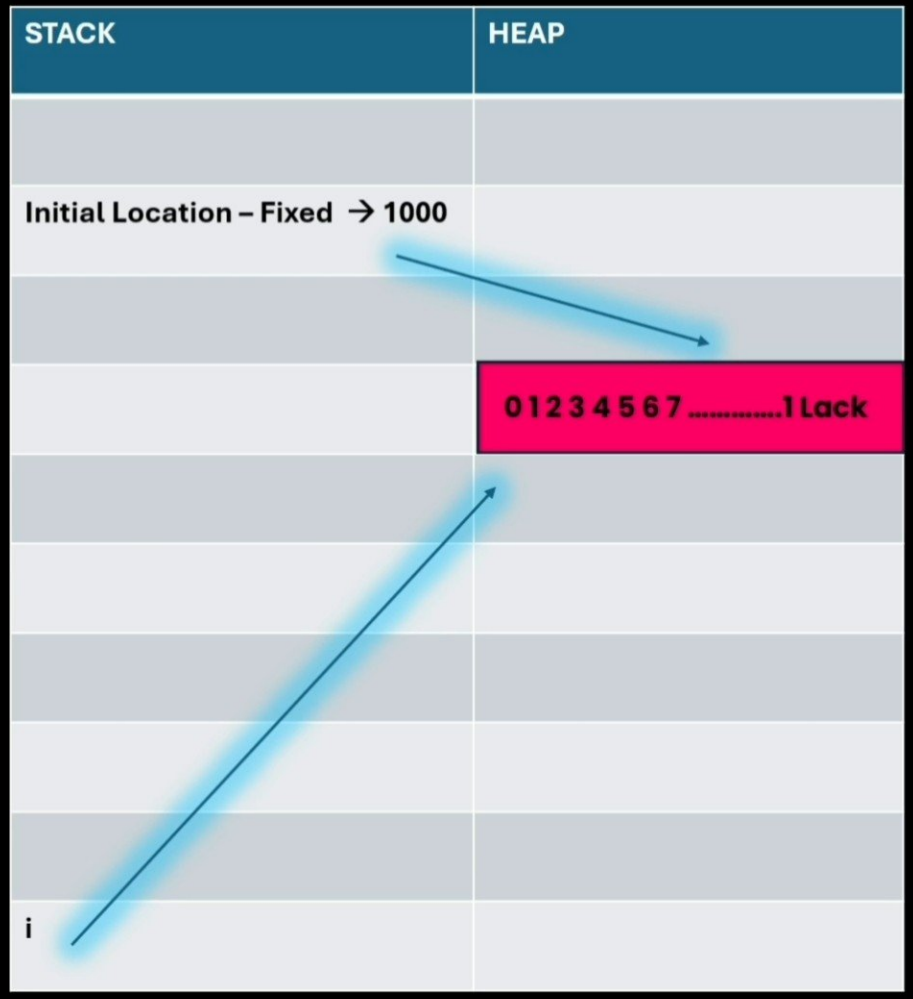
Number → 8 Byte of memory consume

Formula : Array Element → Access

Any Element Address = Base Address + Index * Size of Data

= 1000 + 3 * 8

= 1024 Location → Present ✅



Loop Optimization :

System Build → Base Address = 1000

Advance Array → Create ✅

Why ? → Memory Free Available → Find ❌

Initial Location = 1000 Fixed

0 Present = 1000 Location Present

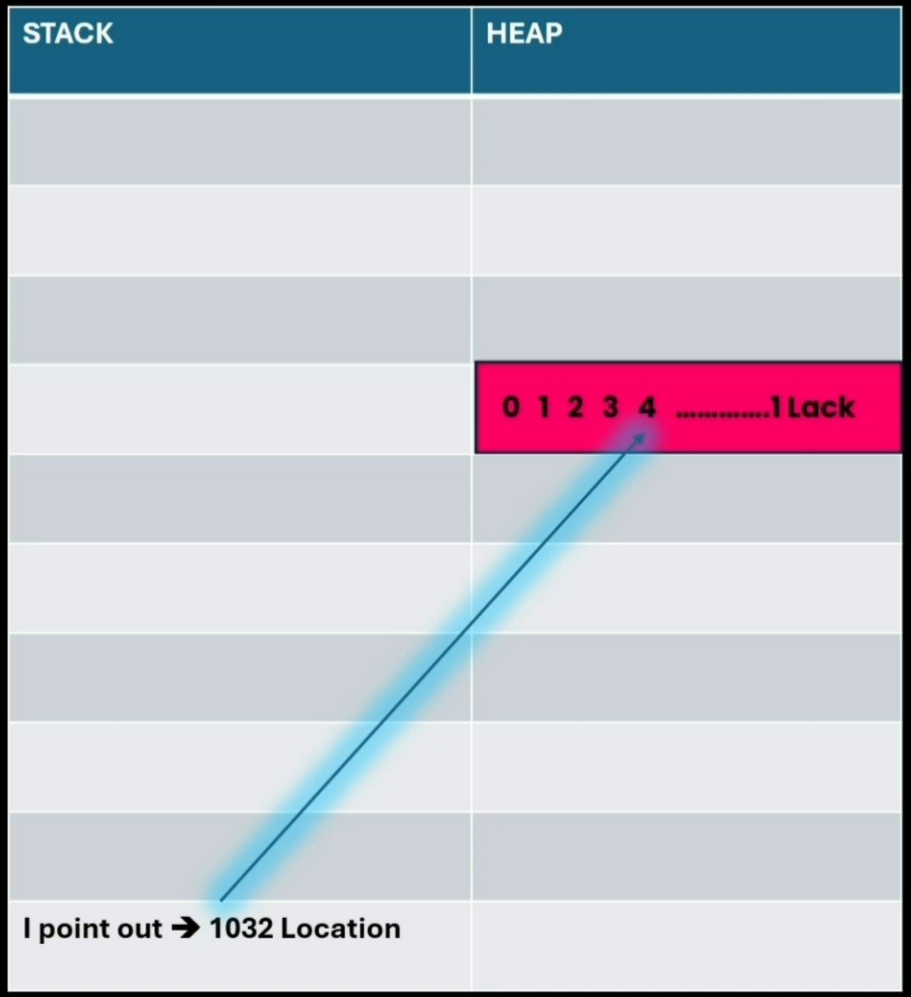
1 Present = 1008 Location Present

2 Present = 1016 Location Present

3 Present = 1024 Location Present

4 Present = 1032 Lcoation Present

I = 4 → Address Update → 1032 Point Out → 4



Loop Optimization :

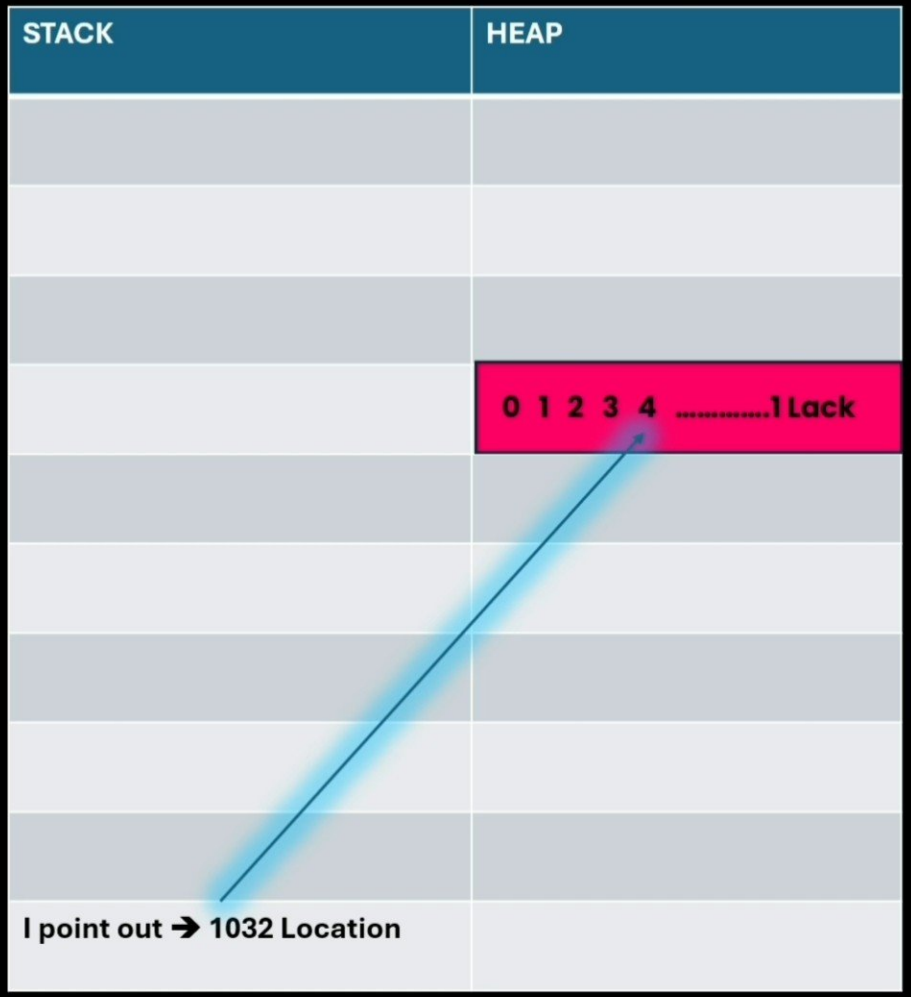
Proublem:

Very Big Array Create → 1 Lack Size

1 Element = 8 Byte Memory Consume

1 Lack Byte → Easily Vanished → In Seconds

Heap All Ready → 1 Lack Byte → Memory Consume ✅



Loop Optimization :

What we learn → Stack To Heap

Stack → 1040 Location → Read →Heap → Number = 5

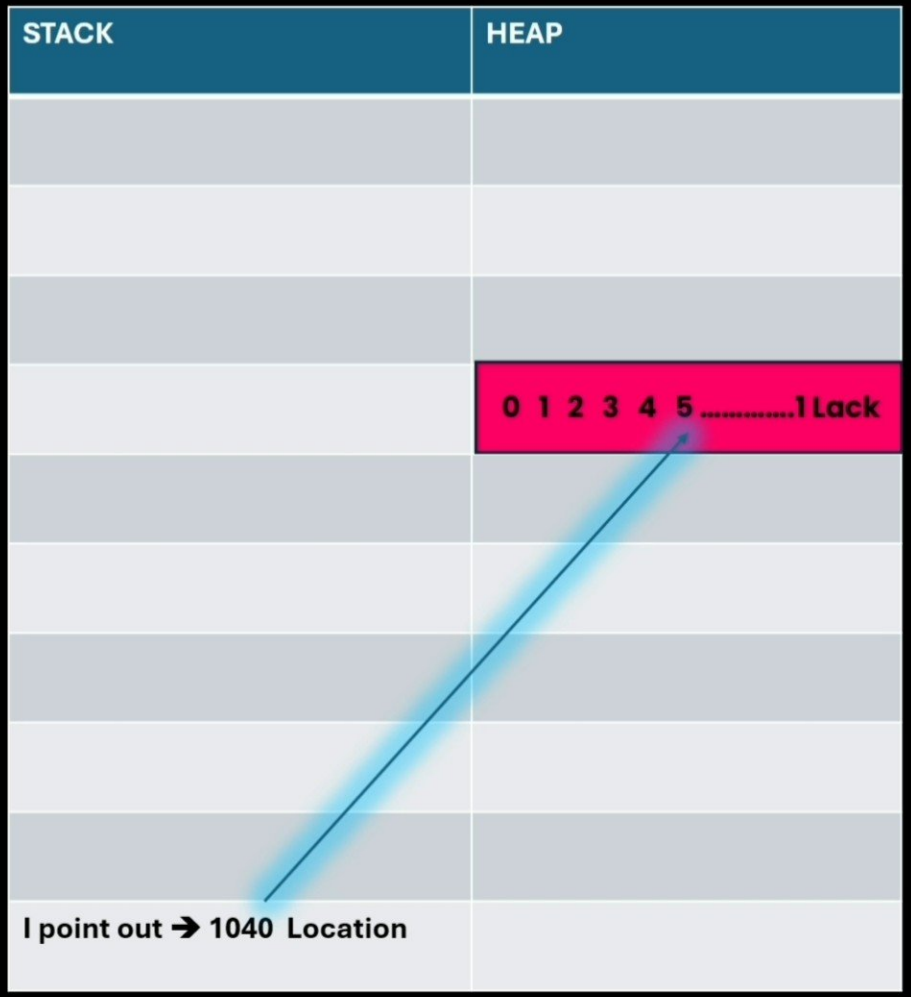
Solution : On the Go Read → Index / Number ✅

Note : Base Address → Fixed

Size Of Data Fixed

I Know Formula :

Element = Base Address + Index*Size Of Data



Loop Optimization :

What we learn → Stack To Heap

Stack → 1040 Location → Read →Heap → Number = 5

Solution : On the Go Read → Index / Number ✅

Note : Base Address → Fixed

Size Of Data Fixed

I Know Formula :

Element = Base Address + Index*Size Of Data

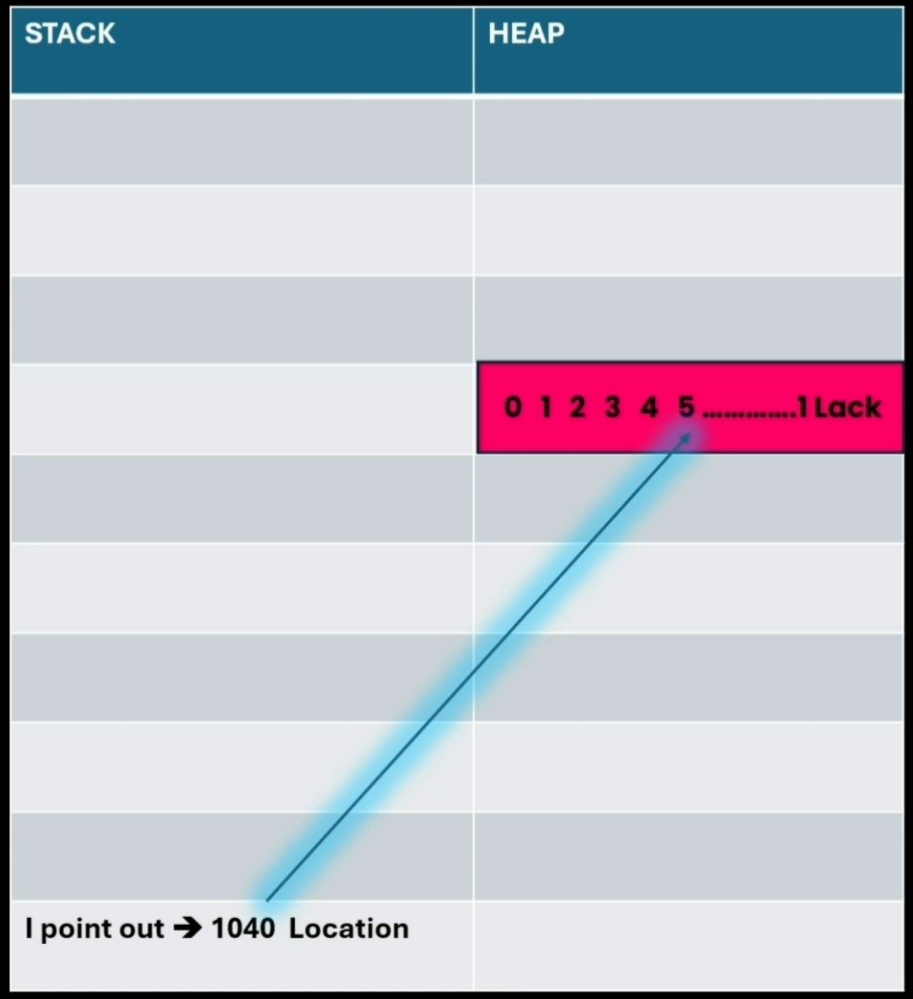
Index / Number = (Element – Base Address)/Size Of Data

Ex.

1000 + 5 * 8 = 1040 → Element 5Th Index Present

Or

(1040 – 1000) / 8 = 40 / 8 = 5



Encoding Data in Memory Address :

String → Rahul → Stored → Heap

X point Out → 2096 Location → Rahul

Now I want to calculate the value → Location ?

Now I don't use formula :

Let's assume :

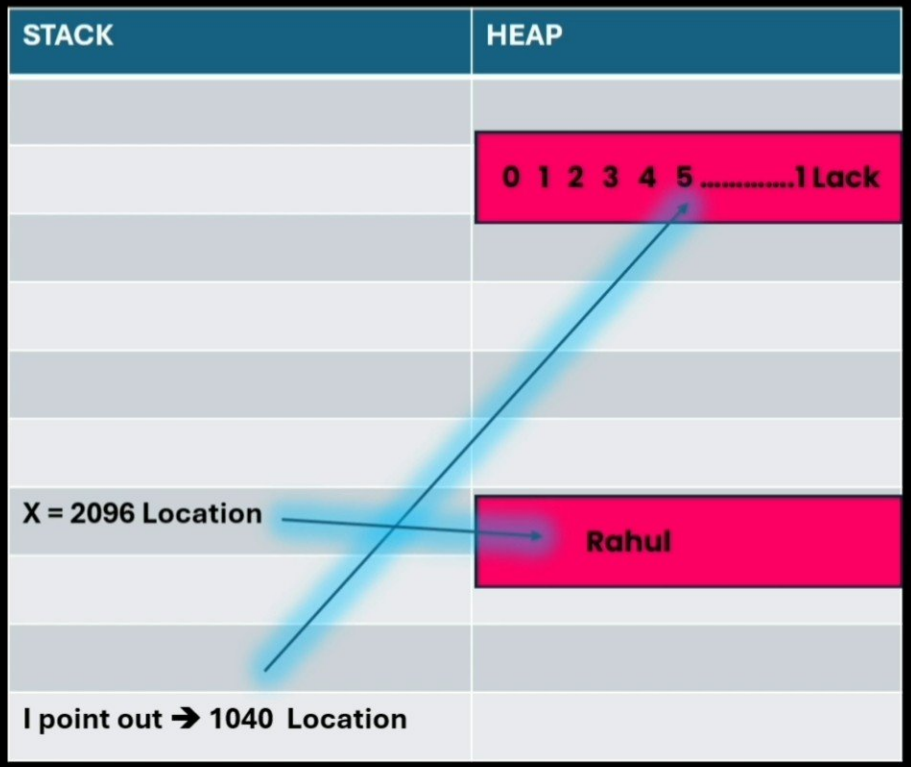
(2096 – 1000) / 8 = Y Ans.

But 2096 Location → Y ❌ → Rahul ✅

Array → Our Trick → Formula Work ✅

Object, String Present → Not Work ❌ → In this case

I want to go there → To read those data



Encoding Data in Memory Address :

String → Rahul → Stored → Heap

X point Out → 2096 Location → Rahul

Now I want to calculate the value → Location ?

Now I don't use formula :

Let's assume :

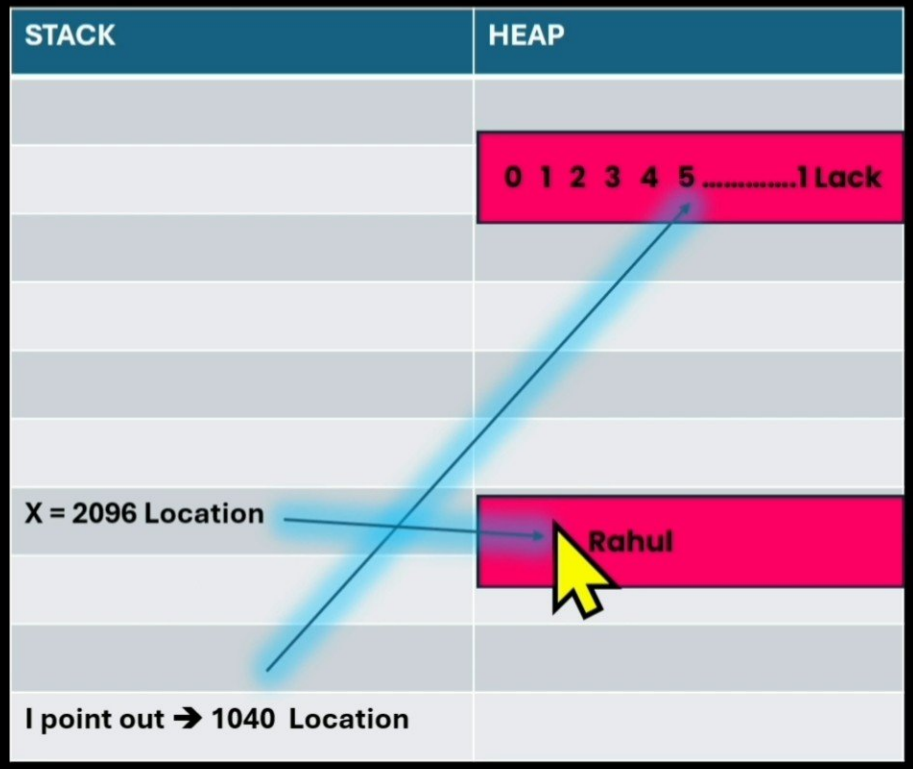
(2096 – 1000) / 8 = Y Ans.

But 2096 Location → Y ❌ → Rahul ✅

Array → Our Trick → Formula Work ✅

Object, String Present → Not Work ❌ → In this case

I want to go there → To read those data



Encoding Data in Memory Address :

32 Bit OS → Address Size = 32 Bit

32 Bit = | 31 Bit | 1 Bit |

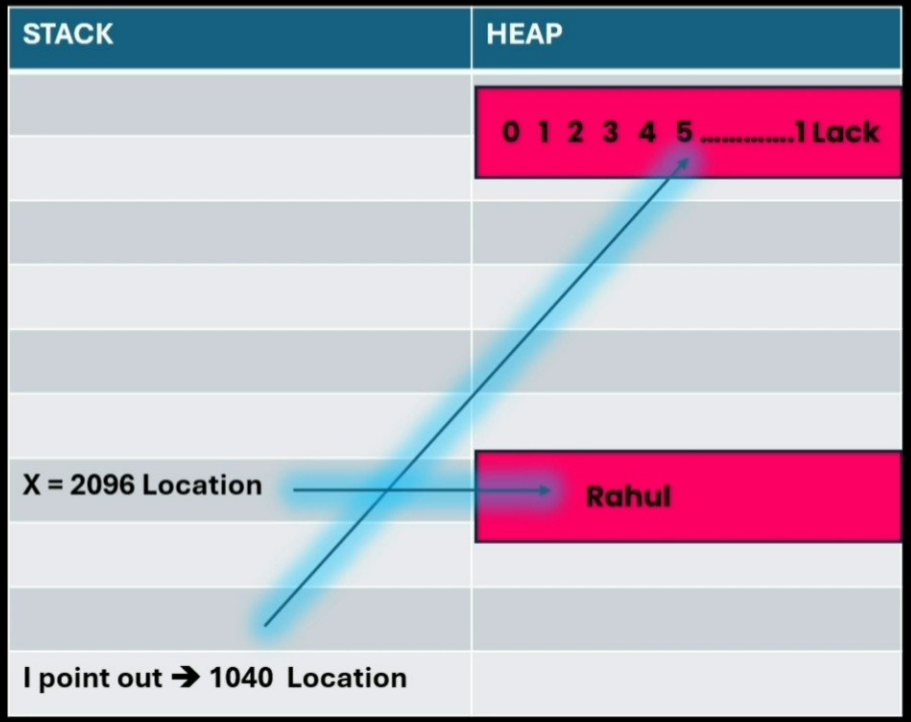
Note :

1. If your last Bit = 0 → I am sure → So I don't go there → It should Pointing Towards → Number

31 Bit — 0

2. If Your last Bit = 1 → You should go there → Because Actual Data Prsent There → So You Read Those Data 1st

31 Bit — 1



Encoding Data in Memory Address :

Now → No need to write → Those Big Array [.....]

Now Need of → [.....] → Waste Space ❌

Trick :

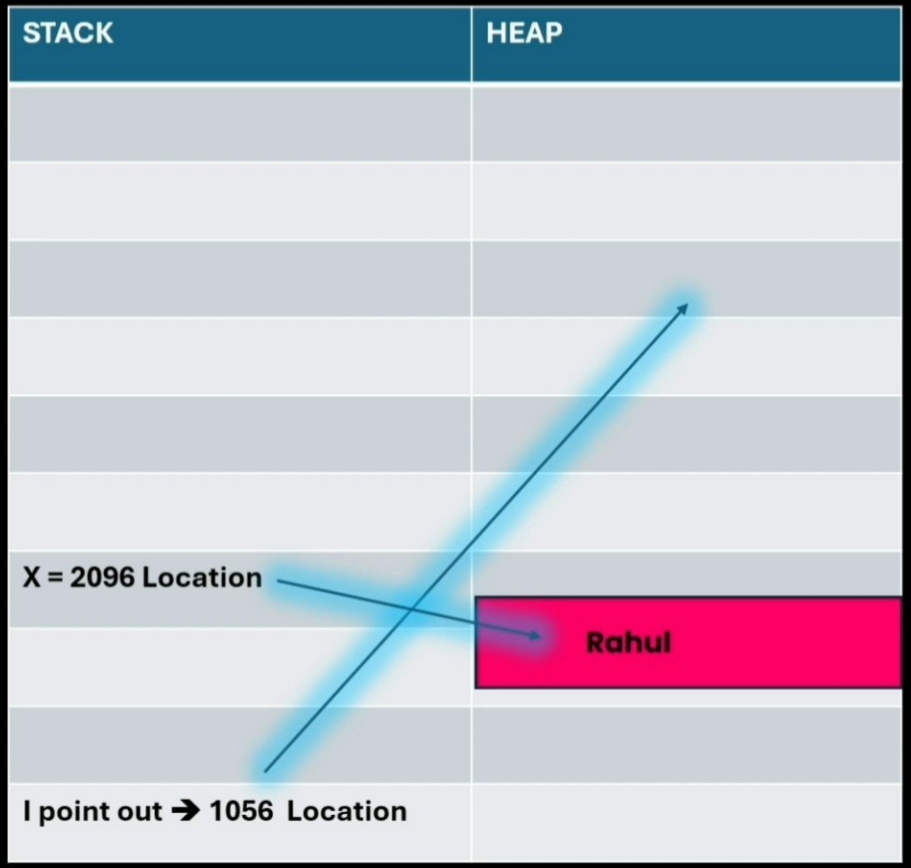
1. I read last bit of Address
2. If → Bit = 0 → Indicates → Number
3. If → Bit = 1 → Indicates → Actual Data Present

Go there & read those data

Ex. I = 1056 → Read Last Bit = 0 → Number Indicate

Increse → i++ Means I = I + 1

1056 + 8 = 1064



Encoding Data in Memory Address :

More Simplified Version :

No need of Base Address

Trick :

1. First 31 Bit → Actual Number Stored

Ex. 2 Stored → 31 Bit

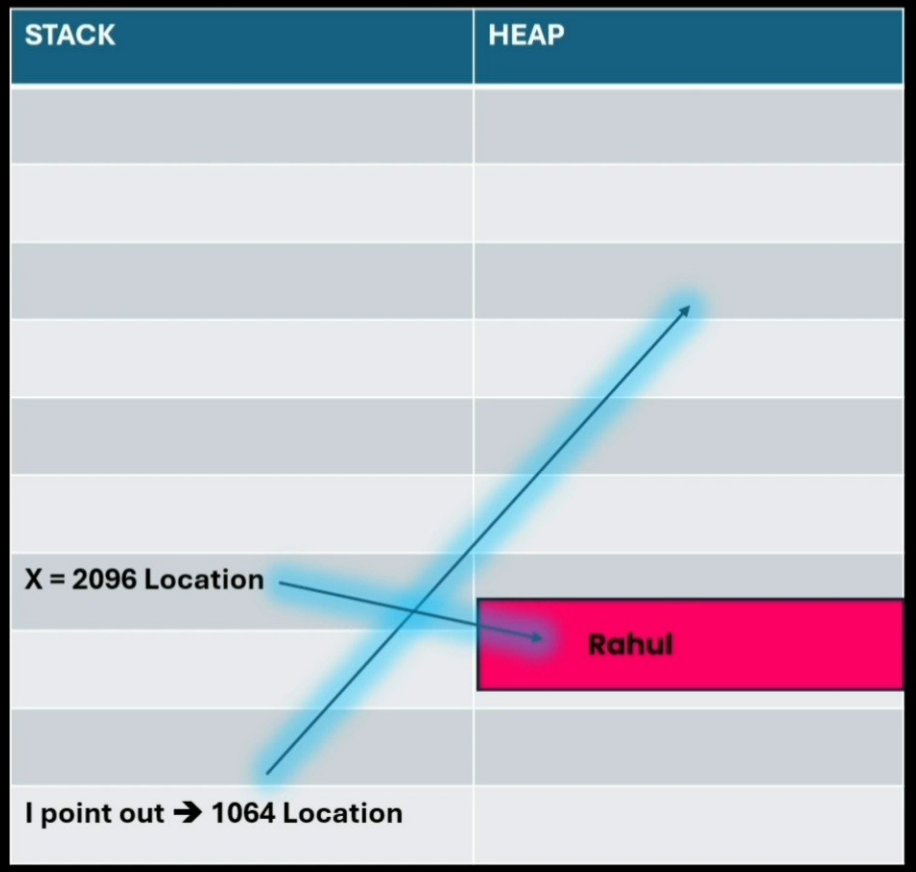
But 2 In Binary = 10

So Before 10 → 0000000000000

So first 29 Bit = 0, 2 Bit = 10, Last Bit = 0

2 In 31 Bit = 00000000000010 0

31 Bit 1 Bit



Encoding Data in Memory Address :

Ex. I = 0

00000000000000000000000000 0

First 31 Bit = 0 Last Bit = 0

Ex. I = 1

00000000000000000000000001 0

31 Bit Start → All 0, Last → 1 Last Bit = 0

Ex. I = 2

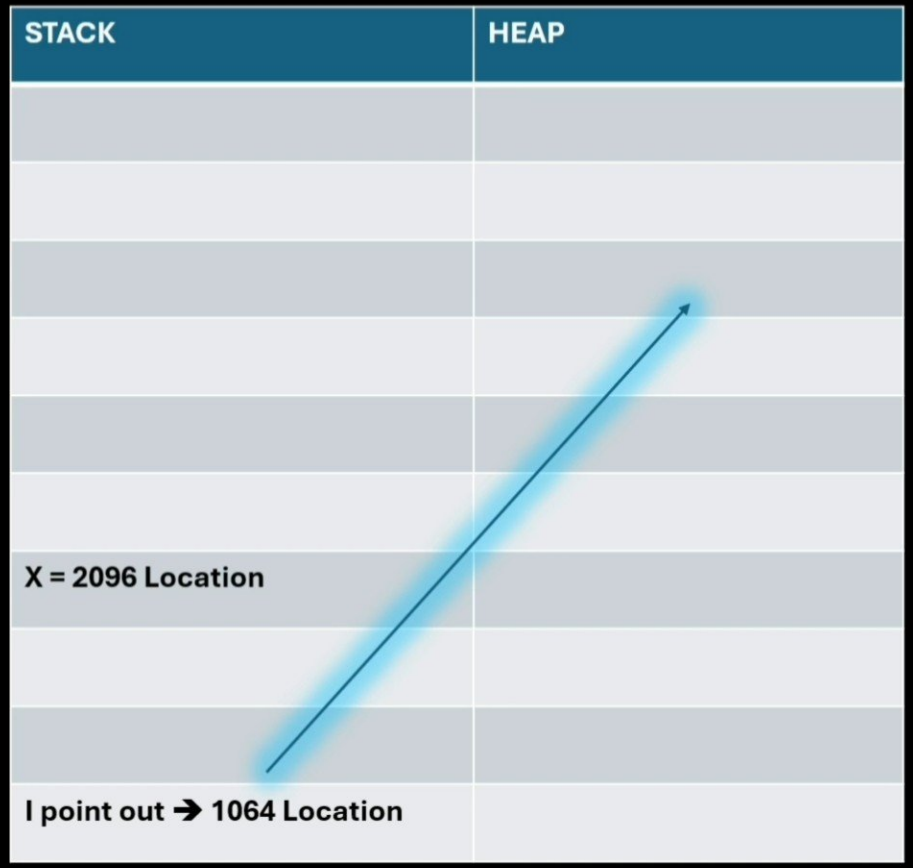
00000000000000000000000010 0

Ex. I = 3

00000000000000000000000011 0

Ex. I = 4

00000000000000000000000100 0



Encoding Data in Memory Address : (31 Bit)

Loop Fast → ✅

Address / Data in Stack → Read → Value ✅

No need to go there in Heap

No need to → Memory Allocate / De Allocate

3 Bit = 2 ^ 3 = 8 Different Types of Numbers

Denotes

4 +Ve Number = 0 To 3

4 –Ve Number = -1 to -4

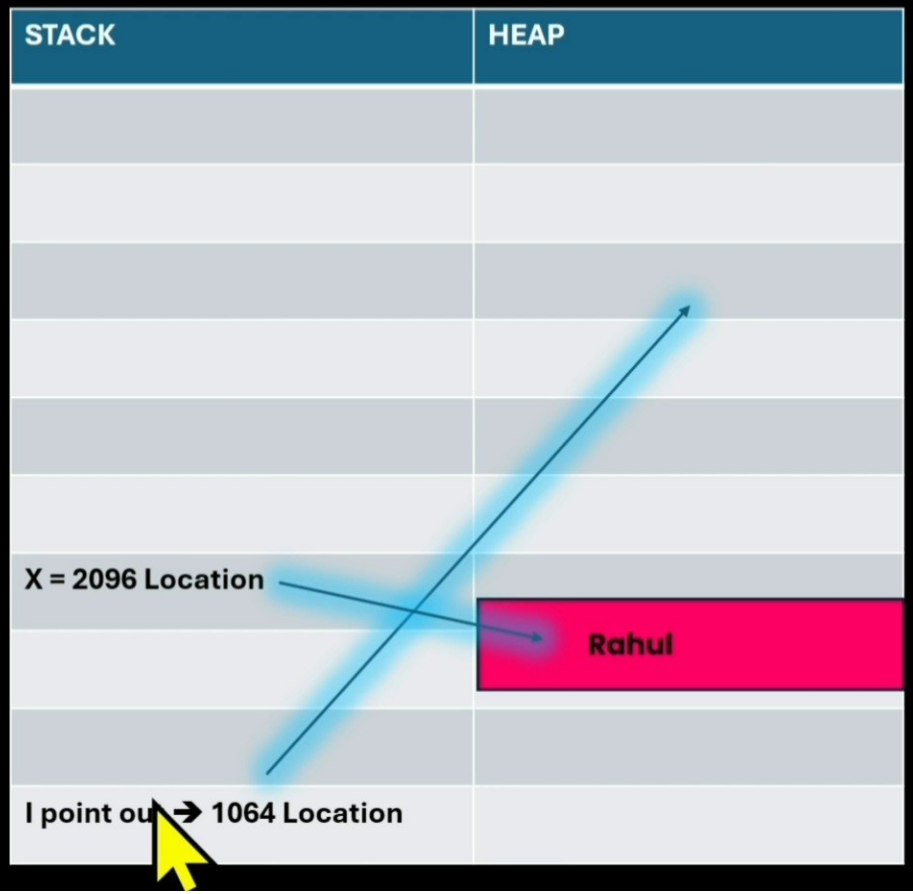
31 Bit = 2 ^ 31 Different Numbers Generate / Denotes / Represent

Half +Ve = 2 ^ 31 / 2 = 2 ^ 30 Different Numbers

Half –Ve

Highest Number = +(2 ^ 30 – 1)

Lowest Number = (- 2 ^ 30)



Encoding Data in Memory Address :

Which one +Ve or −Ve → Representation (31 Bit)

Trick:

Starting = 0 → +Ve

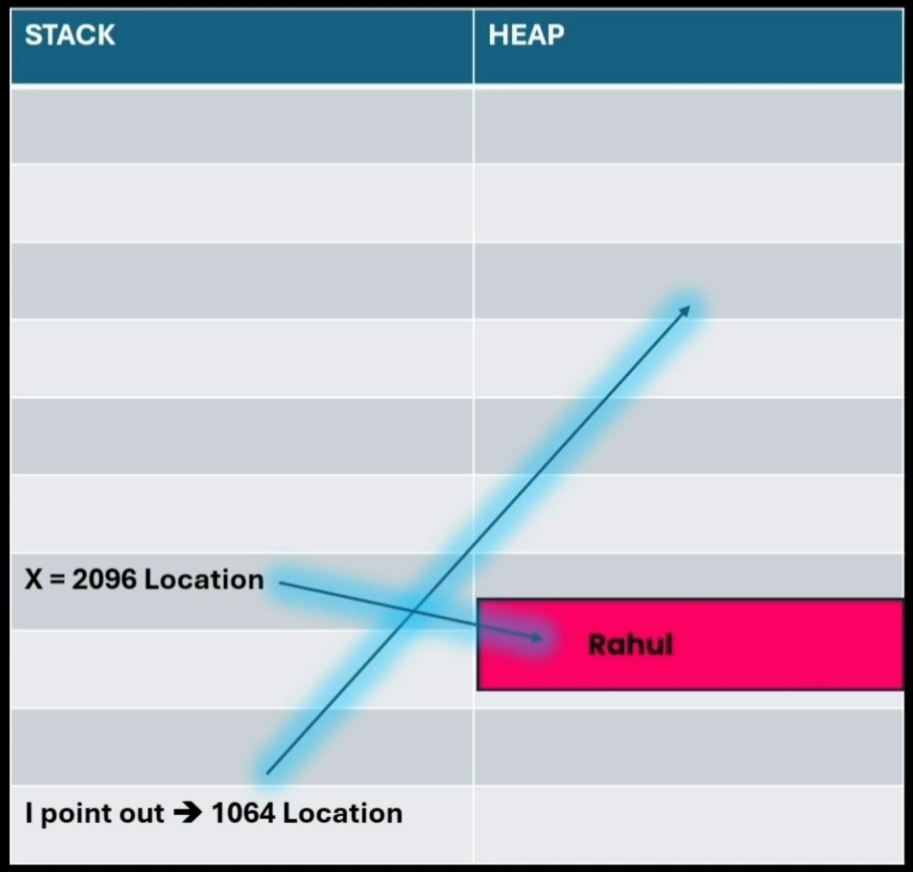
Starting = 1 → -Ve

0000 → +ve

0001 → +ve

1000 → -ve

1001 → -ve



Encoding Data in Memory Address :

Number Represent → Real → 8 Byte

Here We Have 31 Bit

Note :

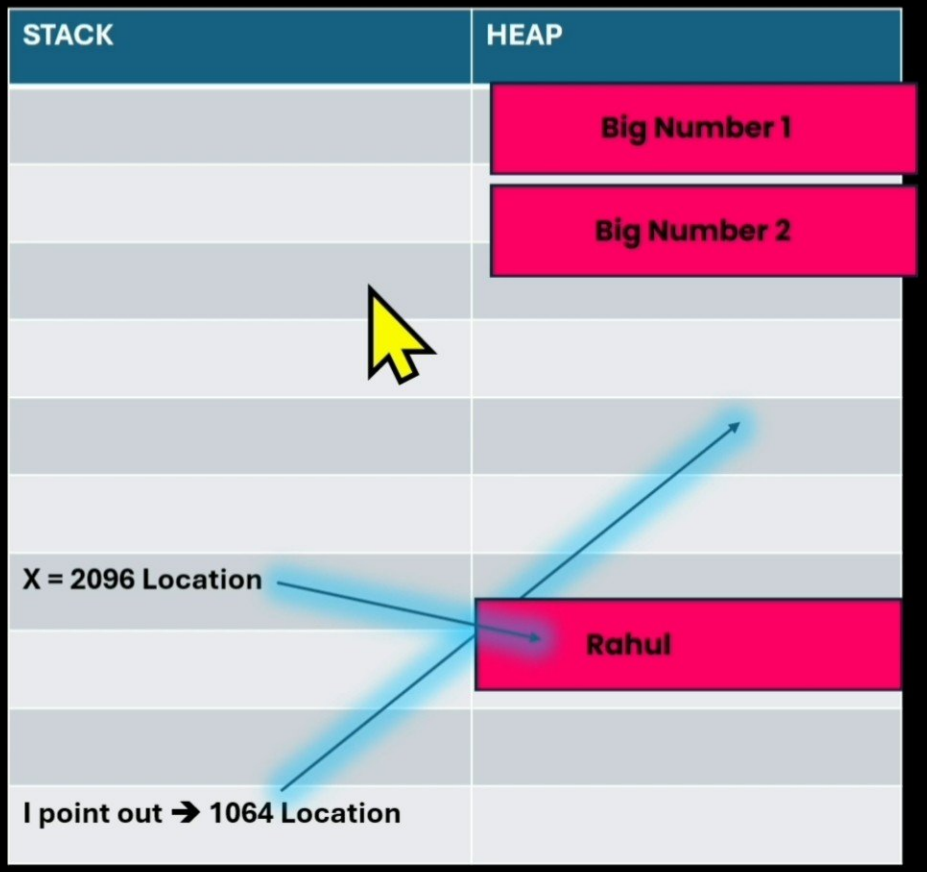
1. Heap → Memory Allocate → Similar Like Other
- Big Numbers like greter than 31 Bit Not Stored in 31 Bit
- Floating point number Not Stored in 31 Bit

33 Bit, 34 Bit, Floating Point Number ❌

Note :

Range : +(2^30 – 1) to (-2^30)

As a address → stored



Encoding Data in Memory Address :

Number Represent → Real → 8 Byte

Here We Have 31 Bit

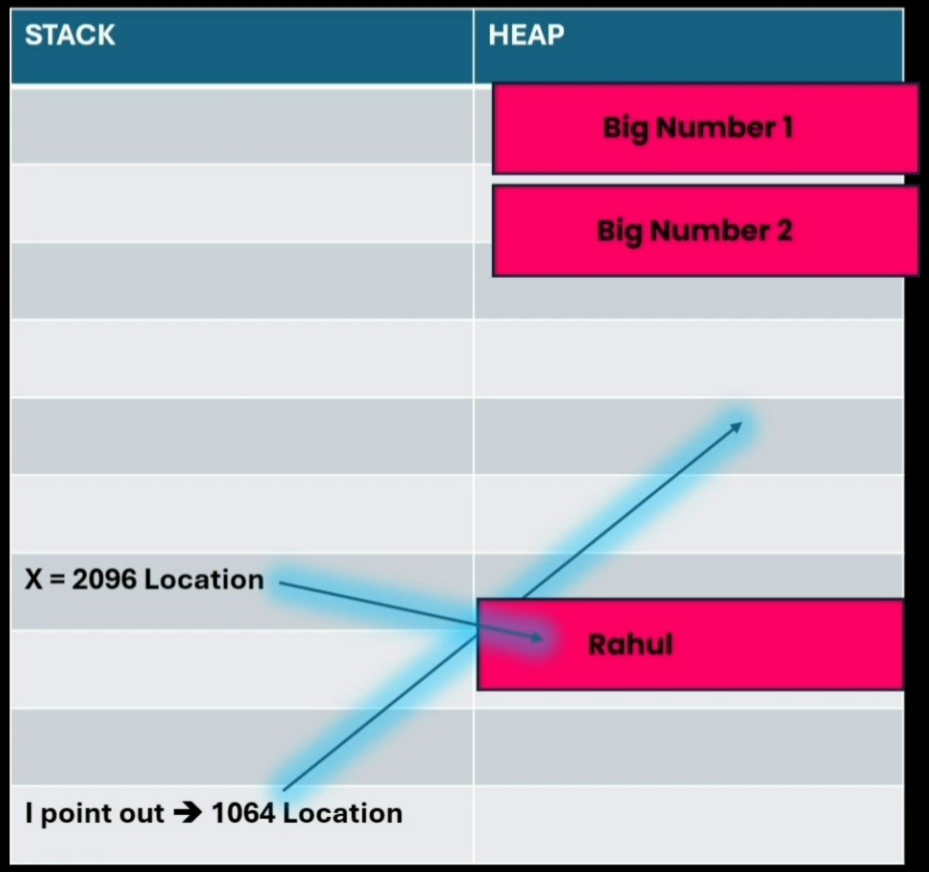
Note :

Range : +(2^30 – 1) to (-2^30)

As a address → stored

Loop → Not For Big Number

2 ^ 30 → Very big number → may be cr



Encoding Data in Memory Address :

VIMP Note :

1. **Small Integers → 31 Bit ✅ → Stack Stored In The Form of Address**

2. **Heap number or Big Number →**
   String
   Undefined
   All Other than small integer → Heap ✅ → Memory Allocate

Object, String → Last bit = 1 → Heap →Data Form

Number → Last bit = 0 → Number → Stack → Add. Form