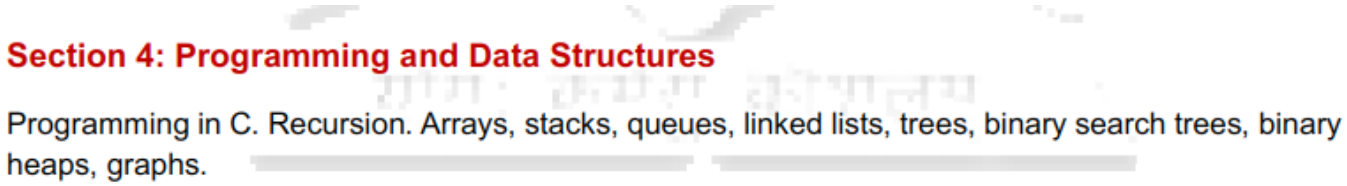
# Data Structure and C Programming Language

## Syllabus



**Section 4: Programming and Data Structures**

Programming in C. Recursion. Arrays, stacks, queues, linked lists, trees, binary search trees, binary heaps, graphs.

## Online Compiler for C language

- Link -> https://www.programiz.com/c-programming/online-compiler/

## Playlist

- Link -> https://unacademy.com/course/course-on-c-programming-data-structures-gate-24-25/07AHS79R

## Doubts and Solutions

> Check them out as well. They have answers to questions I wans't able to do or had problem solving.

- Link -> https://unacademy.com/goal/gate-cs-it/NVLIA/doubts-and-solutions
- Session 9 was Quiz 1

## Important Topics and Video to watch when doing revision

- Pointer videos -> * https://unacademy.com/class/pointer/D788WQUL

```
* https://unacademy.com/class/array/HG0OOZN2
* https://unacademy.com/class/array-with-pointers/56LKYSW6
* https://unacademy.com/class/structure-and-union/JA6PA1NQ
* https://unacademy.com/class/functions/660SO1X9
```

- Doubt Clearing(Many DPPs and Quiz question discussed) -> https://unacademy.com/class/doubt-clearing-session/SJZIOSWP

## Quiz and Practice Questions

- 1 -> https://unacademy.com/quiz/quiz-i/11XTRM5Y37
- 1(Solutions) -> https://unacademy.com/quiz/quiz-i/11XTRM5Y37/solutions/SP_WR9PER0GSENLRBNM5

# Revision Every week or after every chapter is complete.

- Revision 1 -> Till Switch Case
- Revision 2 -> Start from pointers.(Completed till NULL pointers)

# Watch tomorrow

- Link -> https://unacademy.com/class/introduction-to-data-structure/RFZ6RXXG
- Revison on C Language. (Completed)

# C programming.

# Introduction(1)

## Alphabets

^ -> Cap Symbol
& -> AND, ampersand

- -> Asterisk
  {} -> Curly braces
  ~ -> Tilde

| -> Pipe

\ -> Backward

/ -> Forward

# C Language Tokens and Statements(2)

- Keywords -> Pre-defined words and pre-defined usage.
- Identifiers -> user-generated words for naming things like variables.

## Operators

- Operand: 2+3; '2' and '3' are operands. '+' is the operator and more precisely a binary operator.

```
* -6: '6' is operand. '-' is operator and more precisely a unary operator.
* log x: 'x' is operand. 'log' is operator and more precisely a unary operator.
```

## Relational Operators

- = -> Assignment operator
- == -> Equals to operator
- != -> Not equals to.

## Left and Right Shift

### Left Shift

- a << b. 2 << 3.
- *Formula* -> a << b = a * 2^b.
- *Example* -> 2 << 3 = 2 * 2^3= 16(ans).

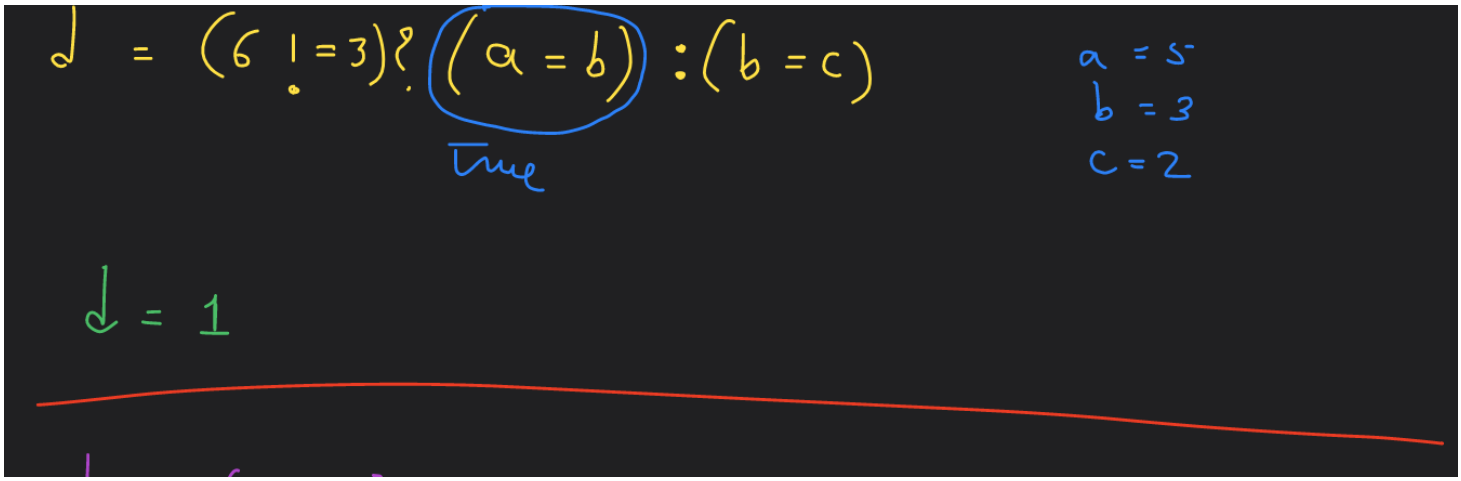
> One left shift means multiplied by 2(here), the multiplication is done 'b' times.

### Right Shift

- a >> b. 2 >> 3.
- *Formula* -> a >> b = (a/2^b) <- Floor Value(Integer part only).
- *Example* -> 14 >> 2 = 14/2^2 =14/4 = 3(ans)
- > One right shift is divide by 2(here), the division is done 'b' times.

# Batch Statements and loops(3)

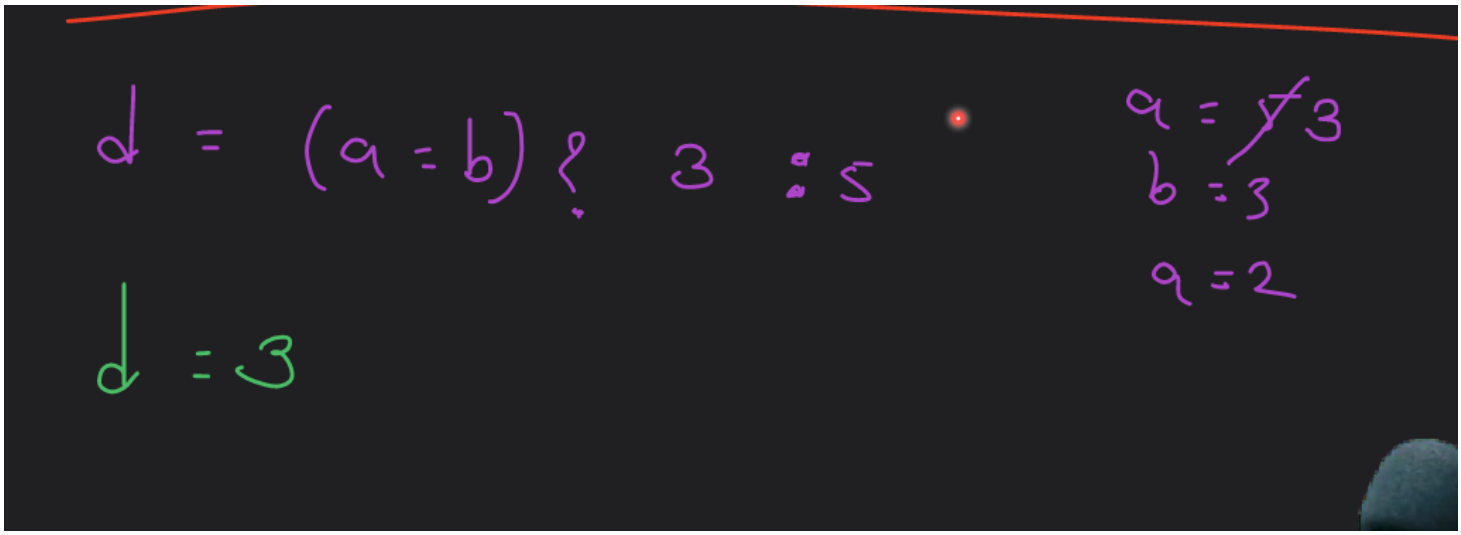
## Conditional Operator




- d= (6!=3)? (a=b) : (b=c) [a=5,b=3,c=2]

> 6!=3 -> True, then (a=b) which is an assig60nment operator. The assignment operation will take place but the assigned value will not be returned to *d* as the result of the *ternary* operator, instead the *result of the assignment* will be returned to *d* as the result of the *ternary* operator.
> The result is that the assignment was completed/successful/True. If some event happens then the result will be *True or 1*.

Hence **d=1** [Since integer value of True is 1.]




- d= (a=b)? 3:5 [a=5, b=3]

> (a=b) is assignment operation and not comparison operator, which is ==. As the assignment was successful and it happened and hence it will return *True* and the value of *a* is updated to *3*

> because of the assignment operator. which means *d=3 and a=3* (Ans).

## Data Types

> The default of Data Types in C programming language is *signed*.

## Header Files

> They are library functions which are pre-written. The functions which are written along with C language.

# DPP 1 (4)

- 1 -> A (Compiler)
- 2 -> B (Double)
- 3 -> C
- 4 -> B (3)
- 5 -> C (a+b =c)
- 6 -> B
- 7 -> A (if)
- 8 -> D (!=)
- 9 -> A (=)
- 10 -> A (&)

> Question 6, ['Associativity' comes into the picture when two or more operands have the same precedence. Here '/' and '*']

## printf() variants

- printf("%d", printf("program")); -> program7

> The word 'program' gets printed. *printf* counts the number of characters that it has printed. So printf counts the length of the characters. These count of the characters will be printed after the word 'program' which was already printed.

- For C programming there are at-least 60 different compilers for it where we can run c language code.

## Structure of C-programming language

> *main() function* -> library function. It is already given. It is to be written in every program. It is mandatory/necessary to be written in every program. Without main(), we cannot write a C-program which we want to execute because execution of a C-programs starts from the main() itself.

- **%u** -> It is used for **unsigned** values.
- **%x** -> It is used for getting values in **Hexa-decimal** format.

# DPP 2

- 1 -> D (Got it wrong, I checked B initially)
- 2 -> A
- 3 -> D
- 4 -> D (Got it wrong, I checked C initially)
- 5 -> C
- 6 -> B
- 7 -> A (d,100)
- 8 -> C (-1,0) (Got it wrong, I checked A initially)
- 9 -> E (Compilation Error) (Got it wrong, I checked C initially)
- 10 -> C (8) (Got it wrong, I checked D initially)
- 11 -> E (GATE EXAM9) (Got it wrong, I checked C initially)
- 12 -> B (GATE EXAM91) (Got it wrong, I checked E initially)
- 13 -> C (20,9,0) (Got it wrong, I checked B initially)

> '++' and '--' can only be used with variables. They cannot be used with constants as in 'Q9' from above.

- *%lu* ->Unsigned int or unsigned long

## Break and continue (5)

*Break and Continue* -> They are **loop controlling statements** and not branch controlling statements.

- *Break* -> It breaks the current loop.
- *Continue* -> It skips/stops the current iteration of the loop and goes to the next iteration.

# DPP 2.1

- 1 -> B (This is if0)
- 2 -> A (GATE 20232021)
- 3 -> F (You are in else)
- 4 -> C (23)
- 5 -> D (TwoThree)
- 6 -> D (Compilation Error) (Got it wrong, I checked C initially)
- 7 -> D (Default)
- 8 -> F (27)
- 9 -> F (I am WonderfulYaha Kyu?)
- 10 -> A (Let's Party)

# Switch Case

> In switch case we cannot use *float* and *double* type values. We can use *char*, *int*. As seen in Q6 from above.

## exit(0)

> Whenever this code/statement runs within a program then we will *come out or exit out* of the whole program *completely*.

# Pointers and Array (6) Very Important

- Links -> * https://unacademy.com/class/pointer/D788WQUL

```
* https://unacademy.com/class/array/HG0OOZN2
* https://unacademy.com/class/array-with-pointers/56LKYSW6
* https://unacademy.com/class/structure-and-union/JA6PA1NQ
* https://unacademy.com/class/functions/660SO1X9
```

> Important as pointer are confusing and many questions come from it. Revise it well. Also go through the notes and the PDFs as well.

- floor() and ceil() are found in *math.h* library. [#include<math.h>].
- floor() gives the *lower nearest* integer number.
- ceil() gives the *upper nearest* integer number.

# Pointer

> If we want to store the address of another variable into a particular variable then a normal variable cannot store the address of another variable. Hence we need a special variable for storing the address of another variable into a particular variable. That special variable is called as the *pointer* variable.
> When we want to store the address of anything into a variable, then we cannot normally define that variable. Normal variable cannot store addresses of anything(another variable). The system will give me an error.

- *Pointer* -> It is also a variable, but it doesn't store values like characters, floats, integers, double etc. It will only store addresses of others or other variables. It is a variable which is used to store address of other variables.
- int *ptr;
- int age=22;
- ptr = &age;
- printf("%d", ptr); -> *Address of the variable(age) ptr is pointing to.*
- printf("%d", *ptr); -> **Value in the variable(age)**
- printf("%d", &age); -> *Address of variable(age)*
- printf("%d\n", &ptr); -> *Address of pointer ptr.*
- *ptr=25;
- printf("%d", age); -> Value of age is updated from 22 to 25 as '*ptr' is pointing to the value of age.

> '*' gives the value stored at the memory address.
> '&' gives the address.

- int n;
- int* ptr;
- ptr =n;
  -> *Invalid* as 'ptr' is a pointer(stores memory address) and 'n' is a integer. Type mismatch error.
- *ptr=&n;
  -> *Invalid* as '*ptr' is integer(value stored at the location where pointer 'ptr' is pointing to) and '&n' is the address of 'n'.
- ptr=&n;
  -> *Valid* as 'ptr' and '&n' are both memory addresses.
- *ptr =n;
  -> *Valid* as '*ptr' and 'n' are both numbers. They represent the value stored in the memory location.

### Example

int a=5, b=2; # Normal variable
int *p; # Special variable called as pointer.(Declaration)
p= &a; # Pointer p storing the address of a.(Initialization)

int *q= &b; # Initialization and declaration in the same line.

> p is a pointer which points to a.

- char *cp; -> It is a pointer which will store address of a character type only.
- float *fp; -> It is a pointer which will store address of a float type only.
- int *ip; -> It is a pointer which will store address of a integer type only.

> All three(cp,fp,ip) are all storing address but whose address is mentioned in the datatype of the pointer.

- We are storing address then why do we need the type?

> It is needed because when we are fetching/accessing the value of *cp*, the system knows that we are getting a character type and not any other type.

> It is true for *fp* and *ip* as well.

- *Type of a pointer* -> It is saying how many bytes are to be read from the starting address. What type of values are stored from that given starting address.
- Size of Integer is same as the size of the pointer?

> Because pointer stores the address and address is in integer format.

- *Multiplication* and *Division* is not possible in pointers.

## Double Pointer

### Example

int **p;

> *p* is a pointer which hold's address of another pointer which is of integer type. (Pointer pointing to another pointer)

# Array

> Continuos allocation of memory of certain datatype. Homogeneous dataype.

## Example

- int A[5];

> It is compulsory/mandatory that it takes up 5 consecutive address in memory.
> The address of the very first element in the array is called as the base address of the array.

- int a[5]= {1,2,3};

> If we declared an array with certain size(say 5 here) and we do not completely fill or make the array full. We just go to a certain size which is smaller than the actual size of the array(say 3 here). Then the rest positions which do not have any values will be initialized with zero(0).

- int A[5];
- printf("%d", A[2]);

> If we tried to print A[2], then we will get an *unpredicted* value as we have not *initialized* the array.
> As we have not initialized the array, hence no values will be initialized and all of the values will become *unpredicted* values.
> If we tried to initialize even one(1) value in the array then rest of the values will be initialized with zero(0) like above.
> In C-programming language if we want an array values to be initialized with all zeroes, then we can do the below code.

- int A[size] = {0};

> The name of an array is a *constant pointer* which stores the base address of that array.
> If we print the name of the array like below, then we will get the base address of the array.

- printf("%d", A); -> gives base address of the array
- int A[5];
- A++;

> The 2nd line(A++) will give us an error when we try to run it because 'A' is the name of the array which means it is a constant pointer and it stores the base address of the array. Hence we cannot change/modify it.

# DPP 3

- 1 -> E
- 2 -> 7,7,8,147 [For n=6] 4,4,4, 48 [For n=3]
- 3 -> C (Got it wrong, I checked B initially)
- 4 -> 0 [When n=6. Any value of n, gives count=0]
- 5 -> 8,1,2 [When n=6]; 64 1 5 [When n=64]

# Array With Pointers (7)

- int *p =&A[0];
- int *p= A;

> These two *&A[0]* and *A* both give the same thing, which is the base address of array A.

- int A[5];
- printf("%u",A); -> 500 which is the base address of array 'A'.
- printf("%u",&A); -> gives address of the constant pointer 'A' which is different from the base address of array 'A'.
- A[0] -> *(500 + 0)

> Both are the same thing.

> Printing 'A' gives you the base address of the Array. *'&A' gives you the address of the constant pointer which is A*.
> Which means the array name and the constant pointer name are the same. 'A' here is the name of the array and the name of the constant pointer is 'A' as well.

- int x=6;
- int *p=&x;

> In normal pointers, the name of the pointer is different to the name of the variable it is pointing to. 'p' is the pointer here and 'x' is the variable to which 'p' is pointing.
> But in Array, both the constant pointer(which is the pointer) and the variable(the array 'A') where the pointer is pointing have the same name which is 'A' here.

- int A[5];
- printf("%u", &A); -> gives the address of the constant pointer 'A'.
- printf("%u", &A+1);

> *'&A+1' means that we are incrementing the address by the size of the entire array*. As '&A' gets initialized first due to precedence and gives '800' as the address of the constant pointer, plus the size of the entire array. We have '5' elements in the array and each element is '2bytes' as they are integer type. Hence size of array is 2*5=10bytes.

- '&A+1' -> 800 + 1* 2 *5 =800 +10=810.
- '&A+2' -> 800 + 2* 2 *5 =800 +20=820.
- '&A+3' -> 800 + 3* 2 *5 =800 +30=830.
- int a[5] = {1,2,3,4,5};
- printf("%d", *a + 3); -> 1 + 3 -> 4

> '*a' gives you the 0th element of the array. It gets executed first as it has higher precedence.

## Difference between regular array and 'int (*p)[5];':

- int (*p)[5]; -> It will create an integer array of size '5'. Each element of the array is an integer only. No name was given to the array. We created a pointer 'p' which will point to the starting address of the array. With this pointer we can access all of the elements within the array.

```
int A[5]= {1,2,3,4,5};
int (*p)[2] = A;
int *q=A;
printf("%d, %d\n", *p[0], *q); // 1, 1
q++;
printf("%d, %d\n", (*p)[0],*q); // 1, 2
p++,q++;
printf("%d, %d\n", *p[0],*q); // 3, 3
p++, q++;
printf("%d, %d\n", *p[0], *q); // 5, 4
// p++;
// printf("%d\n", *p[0]);
```

> 'int (*p)[5];' is a pointer to an array.

> When creating a regular array, the name of the regular array is constant. It means the name of the array and the name of the constant pointer is the same. We cannot increment the pointer as it is a constant pointer.

> In 'int (*p)[5];', we are naming the pointer ourselves. Hence we can increment the pointer as well(p++) and it will work fine. It is a pointer to an array.

- int *p[5]; or (int*) p[5]; -> This will create an array where all of the elements are pointers of type int rather than an element.

> 'int *p[5]; or (int*) p[5];' is an array of pointers.

# 2D Array (8)

- int A[4][5];

> In C-language there is only one way to store 2D array elements which is *Row-wise or Row major ordering*.
> In general Data structure, there are two ways to store 2D array elements which are *Row-wise(Row major ordering)* and *Column-wise(Column major ordering)*.

- int A[4][5];
- printf("%d", A);
- printf("%d", &A[0][0]);

> Both 'A' and '&A[0][0]' gives the base address of the 2D array which is 'A'.

## Row-wise order

## Find the address of any element A[i][j].

- Initial Array -> A[4][5]
- Base Address -> 500
- Size of each element -> 2(Integer values)
- A[2][4] = Base address + size of each element * (i * (no. of columns) + j)

```
= 500 + 2 * (2*5 +4)
= 500 + 2*14
= 528
```

- int A[4][5];
- printf("%u", A) -> base address
- printf("%u", &A[0][0]) -> base address

'A+1', 'A+2', 'A+3'

> If we write 'A+0', then we are trying to access the name of the first(0th row) row of the 2D array. 'A+1', 'A+2', 'A+3' so no are trying to access the name of second, third, forth row of the 2D array respectively.

- 'A+0' -> trying to access the name of the *first(0th row)* row of the 2D array

- 'A+1' -> trying to access the name of the *second(1st row)* row of the 2D array
- 'A+2' -> trying to access the name of the *third(2nd row)* row of the 2D array
- 'A+3' -> trying to access the name of the *fourth(3rd row)* row of the 2D array
- 'A+i' -> trying to access the name of the *ith* row of the 2D array
- int A[4][5];
- printf("%d", &A[0]);
- printf("%d", A+0);

> 'A[0]' and 'A+0' gives us the same output which is the name of the 0th row of 2D array 'A'.

- printf("%d", &A[0] + 1);
- printf("%d", A[1]);

> '&A[0] + 1' and 'A[1]' will give the same output which is the name of 1st row of 2D array 'A'.

- printf("%d", *(&A[3][2]));
- printf("%d", A[3][2]);

> Both will give the same output which is the value/element at A[3][2].

## Initialize 2D array (9)

- int A[3][4] ={{1,2,3,4}, {5,6,7,8}, {9,10,11,12}};

> Anything that is missed will be initialized to zero or '0' like in 1D array.

# Runtime and Compile(Compilation) time errors:-

> **Runtime** -> Errors which occur during program execution(run-time) after successful compilation are called run-time errors. One of the most common run-time error is division by zero also known as Division error. These types of error are hard to find as the compiler doesn't point to the line at which the error occurs.

> **Compile** -> Errors that occur when you violate the rules of writing syntax are known as Compile-Time errors. This compiler error indicates something that must be fixed before the code can be compiled. All these errors are detected by the compiler and thus are known as compile-time errors.

# Structure (10)

> Collection of different datatype elements.

## Syntax

```c
    struct books
    {
        int id;
        float price;
        int quantity;
    };
```

- The size of the *struct books* will be 2 + 4 + 2=8bytes.

> 'structure' declaration should be outside main().
> Declaration of 'struct' or structure type means that the user has defined/declared a **new datatype**. It is a user defined datatype. It is using all of the already existing data-types.
> After we have declared the 'structure', then there will not be any memory allocation/declaration for the structure. It is for creating a prototype and anyone who want to use *struct books* can use it. It has a collection of three things which are *int id*, *float price* and *int quantity*. It is just a prototype that I have created and yet to use it. Hence memory will not be allocated for it as well.
> Memory will be allocated when we will use it to create variable of these type(struct type).

- Declaring a *structure* or 'struct' type -> Making a prototype which is user defined datatype. No memory is allocated to it.
- To use it we need to create variables of these structure type.

> 'struct' type can be present within a different 'struct' type. Same name struct cannot be within itself, it is not possible.

```c
struct student
{
    int rno;
    struct books b1;
};

struct student stud1;
stud1.b1.id;
```

```
struct ABC
{
    int x;
    char y;
    float z;
};

struct ABC a1;
struct ABC *ptr;
ptr= &a1;

a1.x=10;
a1.y='A';
a1.z=2.5;

printf("%d", ptr ->x); // 10
printf("%d", *ptr.x); //10
```

> 'struct ABC *ptr;' is a *struct ABC* type pointer which is pointing to 'a1'. 'ptr' is a pointer which will store address of structure variable(a1 here).
> 'ptr ->x' and '*ptr.x' are the same.

# Union

> The elements are stored in overlapping spaces.

## Syntax

```
union test
{
    char x;
    int y;
    float z;
};
```

> The size of the *union test* will be *4bytes* as the highest size of datatype is float which is 4bytes. Hence the size of the union is 4 bytes.
> We cannot store all of the values uniquely at a given time. We can assign one value and use it simultaneously. When we can replaced/changed the previous value with a new value then the previous value is not accessible anymore.

# Union and Structure difference

> In union, we are not getting separate space for each value. The largest size data type in the union is used as the size of the union. That size is used **repeatedly and overlapping and overwritten** to store the values of the union. Previous values are replaced/removed when newer values are entered and the previous values are not accessible anymore. All of the values are not uniquely and simultaneously present at the same time.
> In structure, we are getting separate space for each value. The size is the total sum of all the data-types present in the structure. All of the values are uniquely and simultaneously present at the same time.

# DPP 4.

- 1 -> 20 3050 60
- 2 ->
- 3 -> 632
- 4 -> 60,40
- 5 -> C (*(A+i)) (Got it wrong, I checked A initially, A+ i gives the address of the element. ( *(A+i) gives the value of the element.)
- 6 -> D (*((A+i) +j))
- 7 -> A,B,C (Got it wrong, I checked D initially thinking it asked for invalid declarations. D -> (int A[2][]={{1,2,3}, {4,5,6}}😉)

> In multi dimensional array, we have to mention the boundation except the first one. We can leave the first dimension and it will be filled by the system itself. We have to mention the rest of them and it is mandatory otherwise we will get an error.

# Functions (11)

> Able to use repeated code multiple times without writing it every time. Like *printf()* from *stdio.h* library.

## Syntax

```
float fun(int, int, float);

float fun(int x, int y, float z)
{
    float xyz= x*y*z;
    return xyz;
}
int x=10, y=20;
float z=23.5;
float result = fun(x,y,z);
printf("%d", result);
```

# Global and Local variable

- **Global** -> Declared outside the functions. Visible/accessible from anywhere or from any function. They are created in memory just at the start of the program and they are removed/deleted after the complete execution of the program. This is the lifetime of a global variable. Their lifetime is during the program running/execution or throughtout the program.
- **Local** -> Declared within a function. Visible/accessible from within the function itself only. When the function is called and the execution of the program is started then the local variables are created in memory and when the execution of the function is complete then the local variables are removed/deleted from the memory.

> Parameters of a function are local variables as well.

# Call by value and Call by address/reference

- *Call by value* -> We are just sending a copy of the value to the function and whatever changes are made to it are local to that function itself only. No connection is made between the value sent to the function and the final value in the main function.
- *Call by address/reference* -> We are sending the address of the variable to the function and the function has a pointer as a parameter to store that address and the connection is made. If we make any changes to the pointer's value then the value which the pointer is pointing too also get updated/changed.

main.c

Run

Output

```c
// Online C compiler to run C program online
#include <stdio.h>

void call_by_reference (char **x, char **y) {

char *str;
str=*x;
*x=*y;
*y=str;
}

void call_by_value (char *x, char *y) {

char *str;
str=x;
x=y;
y=str;
}


void main(){

char *p="Vishvadeep";
char *q="Gothi";
call_by_value(p,q);
printf("Call by Value -> %s, %s\n", p, q);
call_by_reference(&p,&q);
printf("Call by Reference -> %s, %s\n", p, q);
}
```

```
/tmp/ew38fPguBE.o
Call by Value -> Vishvadeep, Gothi
Call by Reference -> Gothi, Vishvadeep
```

C: > Users > Prince > C test.c > bar(char **, char **)

```c

#include <stdio.h>
void bar(char** x, char **y)
{
    char* str = *x;
    *x = *y;
    *y = str;
}
void foo(char* x, char *y)
{
    char* str = x;
    x = y;
    y = str;
}
int main() {
    // Write C code here
    char* p = "hello";
    char * q = "world";
    foo(p, q);
    printf("%s %s", p, q);
    bar(&p, &q);
    printf("\n%s %s", p, q);
    return 0;
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
PS C:\Users\Prince> gcc test.c
PS C:\Users\Prince> .\a
hello world
world hello
PS C:\Users\Prince>
```

It's worth noting that, if you want to swap the values that the pointers in the main function point to, you need to pass the address of the pointers, not the pointers themselves, and use dereferencing inside the function to change the values.

- Pointer -> It is a powerful tool yes but it is a destructive tool which can access memory. Hence it was removed partially from recent programming languages like JS, Python, Java.

# DPP 5

- 1 -> 96
- 2 -> error [Cannot write function inside the main()]
- 3 -> 11
- 4 -> 10 ['exit(0)', got it wrong , thought '1010' is the answer]
- 5 -> 1 [Finding HCF of two numbers]
- 6 -> 23 [ Changed the question]
- 7 -> 1

> *exit(0)* terminates/exits out of the whole program, not just the function but the whole program itself.

> In Q2, if function was outside of main(), then it will give *-10* as the output/result of the program.

# 12_Doubt_Solving_Session PDF questions

- 1 -> 321
- 2 -> 1213121
- 3 -> Error ("abc" is a string or more than one character. 'char *s' is a pointer of type character)
- 4 -> 7

# Recursion (12)

> Function calling itself. Good Recursive functions have a good condition as a base/terminating condition.

## 13_Recursion_Questions PDF

- 1 -> 2 [Q ->foo(513,2)]
- 2 ->

# DPP 6

- 1 -> D
- 2 -> 10230 [Got it wrong , thought '1048576' is the answer]
- 3 -> 4 [Got it wrong , thought '3' is the answer]

## 14_Storage_Classes PDF (13)

- fun(fun(5)) -> fun(5) + fun(7) = 7 + 17 = *24*(Ans)

> We are re-using the already calculated values(fun(0), fun(1)...fun(5)) from previous fun(5) but computer is doing it again from scratch. Hence first it is calculating *fun(5)* first as it is inside and then calculates *fun(7)* separately from scratch without re-using the values it got from fun(5). Hence we are adding *fun(5) + fun(7)* to get the count of *fun(fun(5))*.

- **IMPORTANT** look at all of the recursion question int *13_REcursive_Questions PDF* file.

### Question

- 1 -> 51 (Got it wrong, thought it was 5 initially)

# Static Variable (14)

> In recursion, **it is created for first call and used by all recursion calls as a single copy**. If a static variable is created within a recursive function then it will be **created/generated once only** when the function is called for the very first time unlike local variables which are created/generated for every recursive call of the function.
> **By default**, if we haven't initialized a *static* variable then it is initialized by *zero(0)*.
> In normal variables, if we haven't initialized the variable then it is initialized by some *garbage value*.

# Dynamic Memory Allocation (15)

> If any memory allocation assignment is decided on rumtime.
> On runtime we are deciding if we need more memory or not. Yes, we can get more memory during runtime and this is called as Dynamic Memory allocation. It happens during runtime only.

- int x; -> Pre-defined variables.

> The memory which is pre-defined when writing the program is called as normal allocation

- Two functions for Dynamic Memory Allocation:-
- **malloc()** -> memory allocation during runtime. For one element.

```
* Example -> malloc(size of memory in bytes)
```

- **calloc()** -> It can allocate memory for more than one element.

## Malloc (malloc())

> 'malloc()' only allocates a certain amount of memory location(2bytes for integer). It will not define the type of the memory wheather it is int, char, float etc. To get the type we need to do *type-casting* of the malloc().
> In some compilers, 'malloc()' return the address.

### Example

- int x= (int)malloc(sizeof(int));
- int *p = (int *)malloc(sizeof(int)); [Use for GATE]
- **Page 7** of *15_Constant_Macro* PDF is an important.

# Storage classes

1. auto
2. register
3. static
4. extern

## auto

> All of the variables created that are local variables without any specification are *auto* variables. By default all local variables are *auto* variables.

```
void main()
{
    int x;
    auto int x;
}
```

> If we don't mention the *auto* keyword, then also by default *auto* keyword is added to all of the local variables.

## Register

> Local variables and exactly like auto but storage is not in RAM(stack) but in the CPU register. Advantage of setting a variable as *register* variable is that the allocation will have in the CPU register instead of the stack. Whenever the cpu will use that variable, it will take it from the CPU register and inturn saving a lot of time as CPU is faster than the RAM(stack).
> If the CPU register is out of space then the newly created register variables will be placed into the stack itself.

```
register int i;
```

# blocks

> We can create blocks using curly braces or {}. It acts like a function even though it is not a function. It has it's own local variables. After the block execution is completed all of the local variables of the block are destroyed.

```
void main()
{
    int x;
    // block.
    {
        int y;
        y= 10*2;
        printf("%d",y);
    }
}
```

# Automatic variables

1. auto
2. register

> They are created automatically in the memory whenever a function or a block is called and they are removed automatically as well when the function or block has completed execution. Stored in stack. Scope within function or block itself.

# Global variables

> There is no dedicated/specific storage class for global variables but they are there.

> All of the variables(global, static variables) that are present in the data section, they are initialized with zero(0).

## static variables

> They can be local as well as global variables. Wheather a static is global or local it is created at the starting of the program execution. Both of their lifetime is throught the program execution. Scope depends if it is a global or a local variable. If global variable then the scope is global as well. If local variable then the scope is local as well.
> static variable is created for the first and the only time at the start of the program execution.

```
static int x; -> // Global static variable
void main()
{
    static int x; // Local static variable.
}
```

## extern variable

> It is similar to global variable but there is some differences. Lifetime is throughout the program. Scope is throughout the program(global) as well. It is initialized with zero(0) as well. It is located in the data section.

## Difference between static and global variable.

- Scope -> static can be a local variable as well but global cannot be a local variable. static variable's scope can be within a function or a block but the scope of a global variable is throughout the program.

# NULL pointer

> When a pointer is declared but not initialized then it has some *garbage value* initialized to it.
> If we want a pointer to not hold any address then we can initialize/set the pointer to *NULL*. This is called as NULL pointer.

```
int *p;
*p=NULL; // NULL pointer.
```

> This means the 'p' pointer is created but the address of pointer 'p' is neither pointeing to any valid address or any garbage address. It has a pointer(NULL) which shows that it has nothing.
> 'NULL' pointer denotes there is nothing at 'p' pointer.

# String

```
char ch[]= {'A','B','C','D', 'E'}; // Character Array. Size is 5.

char str[]="ABCDE";  // String. Size is 6. 'str' will be a constant pointer. We canr

char *p ="ABCDE"; // This is also a string. It removes the size issue from below. 'p

char ch[5]="ABCDE"; // Error as size should be 6.
```

> String has '6' size compared to character array which is of size '5' because the last position of string is reserved for '\0' which is a *NULL character*. *It represents the end of the string*.

# Literals and Constants

```
const float pie=3.14; //const type name=value;
```

> 'value' is mandatory to be given otherwise it will not be a constant. 'const' variable 'pie' cannot be changed, modified, cannot allocate new value, cannot increment or decrement the value. Nothing is allowed as it is a 'constant'.
> Once it is defined and the value is fixed, nothing can be changed.
> They can both be global as well as local.

## Macro

```
#define PIE 3.14   // macro
printf("%f", 2*PIE*r);
```

> Where ever we have mentioned 'PIE' in the program, that is going to be replaced with '3.14' value.

```
#include<stdio.h>
```

> All of the statements that are starting with '#' are called as preprocessor directive.
> All of these statements(starting with '#') are run before the compilation of the program starts.

# Quiz questions (1)

- int j =(x++, --y);

> No matter what is the value of 'x' and 'y' is, we will always take the 2nd or the last value i.e the value of 'y' here and store it in 'j' variable.

# Data Structure with C

## Daily Revision

- 1 -> Done still insertion in array (14 Jan 2023)
- 2 -> Start from insertion in array (15 Jan 2023)
- 3 -> Start from searching in array (16 Jan 2023)
- 4 -> Start from Deletion in linked list(17 Jan 2023) before starting new lectures.
- 5 -> Start from Header List.(17 Jan 2023).
- 6 -> Solve the DPP question as revision. (18 Jan 2023).
- 7 -> Start from Queue.(19 Jan 2023)
- 8 -> Start from Double Ended Queue(20 Jan 2023)

# Weekly Revision

- 1 -> Revise everything from 16 to 21 Jan 2023. Practice questions.

# Introduction to DS (17)

> It is not only the organization of data but the functions to access the data as well. It is sometimes referred to as **ADT or Abstract Data Type**.

- Data Structure -> Data Storage + operations.

## Types

- Linear -> Elements arranged in linear order. Eg -> Array, linked list, queue, stack
- Non-Linear -> Elements are arranged in non-linear fashion. Ed -> Graph, Heap, hashing, tree

## Analysis of Algorithms

> To find out among all of the potential solution to a particular program which one is better, best or worst among each other.

## Types

- **Space Complexity** -> Space(memory) needed by the program to run. The memory is calculated by the extra space taken by the algorithm to run itself.

> So memory or space of an algorithm is calculated excluding/without including the *input and output* of the program.

> In space complexity, we are not calculating the *bytes* of the program but rather the number of elements that we have stored in the program **separately not including the input and output variables/values**. The no. of elements stored separetely is called as the space complexity of a program.

- **Time Complexity** -> Time(Runtime) needed by the program to run.

# How to calculate time? (18)

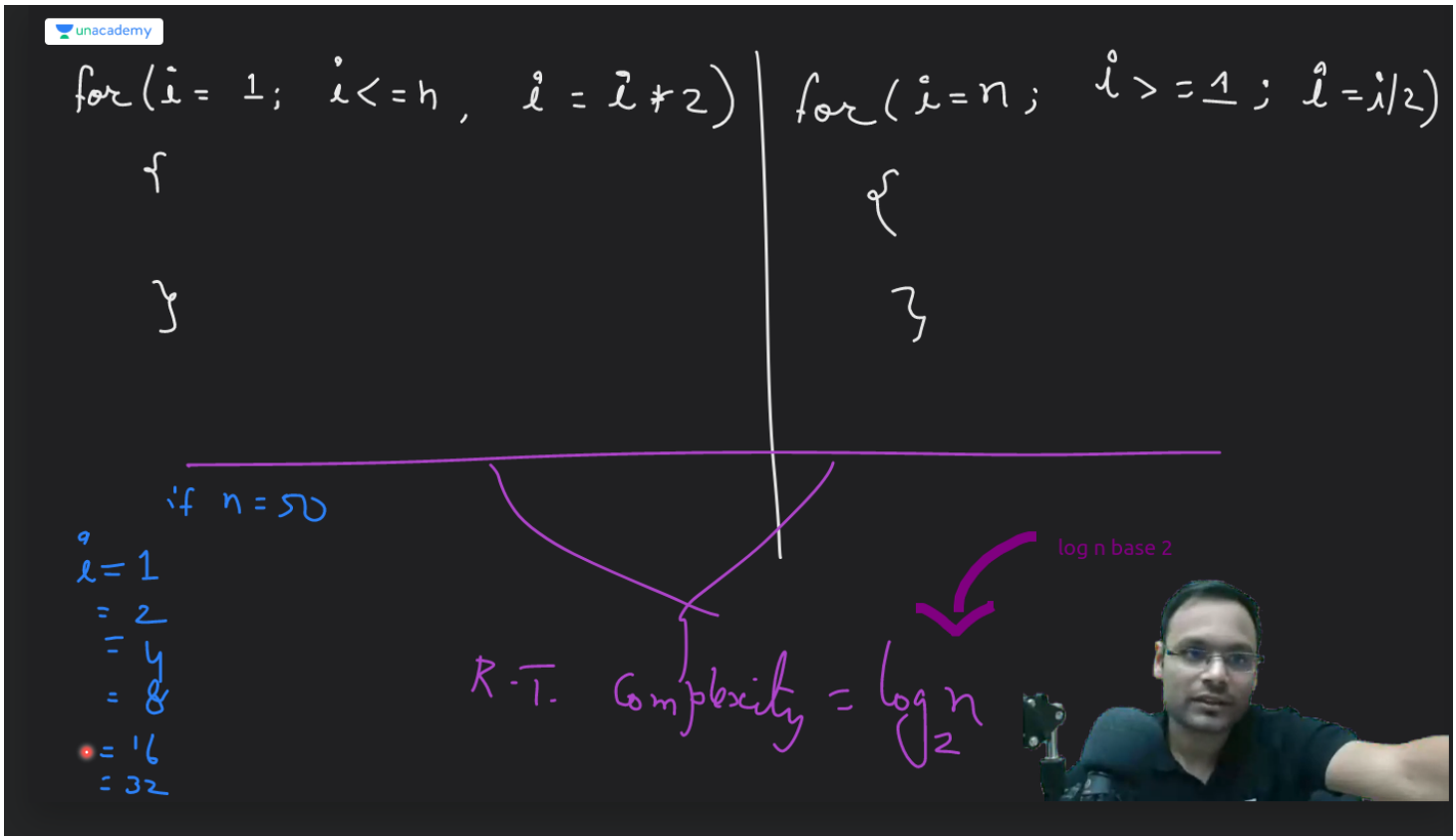
- On Wall Clock -> Implement the program/algorithm in any langauge and after implementation run it on CPU/system. Now calculate/measure the milliseconds/nanoseconds for which the program is running.
  > It is highly dependent on so many external factors. So no stability or no any consistency in results.
- Based on no. of operations or steps performed (Dry run) -> Manually run the program on paper. Calculate the total no. of steps and count it. Then the result will be consistent. It is based on the mathematical analysis of **rate of growth**.
- log(logn) < logn
- Time complexity is independent of the input and it is constant only. Then it is called as constant runtime complexity. It is represented by '1' and not the no. of steps of the algorithm.
- i=0, i<n

- i=1, i<=n

> They are the same thing.

- for(i=1;i<=n;i*=2) or for(i=n;i>=1;i/=2)

> Runtime complexity is *log n base 2*




- for(i=1;i<=n;i*=3) or for(i=n;i>=1;i/=3)

> Runtime complexity is *log n base 3*

- int k;
- for(i=1;i<=n;i*=k) or for(i=n;i>=1;i/=k)

> Runtime complexity is *log n base k*

# Asymptotic Notations (19)

- Big O -> Provides Tighest upper-bound. Maximum complexity of an algorithm.
- Omega -> Provides Tighest lower-bound. Minimum complexity of an algorithm.
- Theta -> Provides exact bound. Max and min complexity of the algorithm is the same.

> Used for bounding complexities.

## Types of cases

- Best case -> Type of input for which, algorithm takes minimum time.
- Worst case -> Type of input for which, algorithm takes maximum time.
- Average case -> The input which is not best or worst.

> We cannot write omega(1) as the constant complexity because omega represents the lower-bound or the minimum complexity hence we don't know the maximum complexity.
> We can use O(1) or theta(1) as constant complexities. O(1) is the maximum complexity and nothing is smaller than that. Hence nothing can be smaller than that.

- O(1) and theta(1) are the same representation as highest bound is '1' then the lower bound will also be '1' as there is nothing lower than '1'.

# Array in DS

- int a[5];
- Lower Bound(LB) -> Starting index -> 0
- Upper Bound(UB) -> Last index -> 4 -> (size -1)
- name[LB:UB]
- A[0:4]
- B[2:6]
- C[-4:3]
- Find the size of the array

> size = UB-LB +1

## Relative Index

- Relative index(RI)= (i-LB)
- Relative index would help us to find the location of an element

## Location of an element

- Location of an element Loc of A[i] = Base address + size of an element in memory * relative index

```
= Base + w * (i-LB).
```

### Question

- A[-4:200]

- Base address -> 2500
- W -> 4 locations
- Location of A[17] = 2500 + 4*(17-(-4))

```
= 2500 + 4*21
= 2500 + 84
= 2584 (Ans)
```

- A[-6:13]
- W -> 8 locations
- 1 location = 1Byte
- Total no. of elements in array = UB-LB+1 = 13 -(-6)+1 = **20** elements(Ans)
- Size of memory required to store completed array(In Bytes) = 20 * 8 = **160 Bytes**(Ans)

# Why indexing in array in C language starts from 0?

> It saves the (i-LB) calculation. CPU doesn't have to do the substraction which helps in performance as well if indexing starts from '0'.
> Then the formula will be like below.

- Location of an element Loc of A[i] = Base address + size of an element in memory * i

```
= Base + w * i.
```

**Important**

> When CPU tries to access an array element, it has to first find the address of the element. To find the address he has to use the non-optimised or earlier formula(i-LB) one. If we keep LB=0, then the formula will be reduced to the above one(Base + w * i). In that case, CPU is saving one substraction every time it tries to access an element which in turns increases the performance of the CPU. Everytime we want to improve the performance of the CPU, hence the programming languages have adopted to keeping the starting index of an array to zero(0) for improving the performance.

# Quiz Questions (2) (20)

## Solution on the Quiz

Question 4

-0.66 MARKS

Your Time Taken: **2m 3s**

Avg Time Taken By Others: **3m 2s**

Attempt Accuracy: **31%**

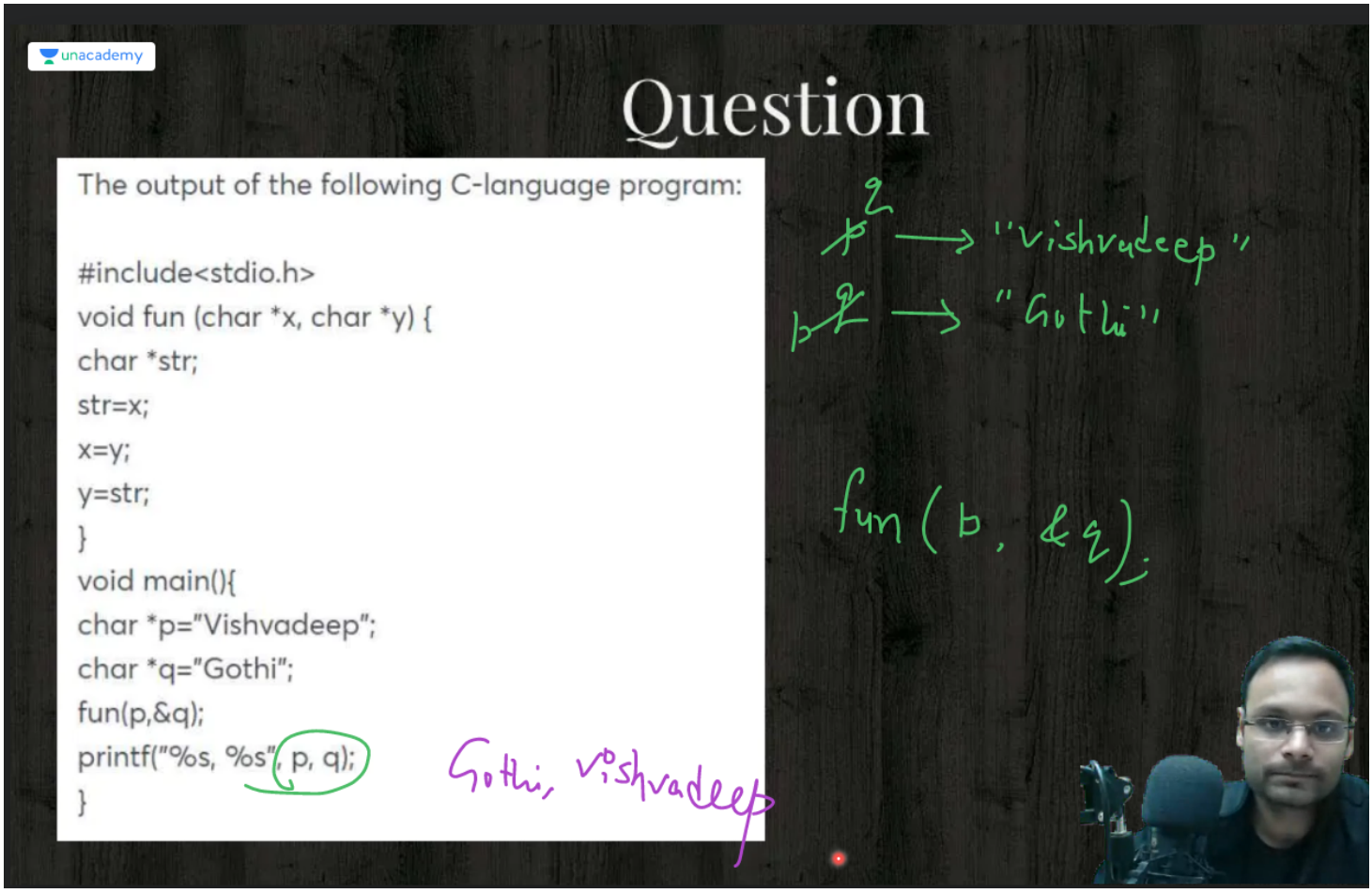
The output of the following C-language program:

```
#include<stdio.h>
void fun (char *x, char *y) {
char *str;
str=x;
x=y;
y=str;
}
void main(){
char *p="Vishvadeep";
char *q="Gothi";
fun(p,&q);
printf("%s, %s", p, q);
}
```

- Gothi, Vishvadeep — INCORRECT
- Vishvadeep, Vishvadeep
- Gothi, Gothi
- Vishvadeep, Gothi — CORRECT ANSWER

Solution View

## solution provided by teacher




> The solution provided by the quiz was wrong. Keep it in mind. Don't blindly accept answers. Check it yourself. I also ticked the correct answer in the quiz.
> Update, I ran the code on a compiler and it gave the answer as it was in the quiz solution. Better explaination on the below link.

- Link -> https://unacademy.com/goal/gate-cs-it/NVLIA/doubts-and-solutions/doubt/FE03159543

**Compiler**

main.c

```c
// Online C compiler to run C program online
#include <stdio.h>

void fun (char *x, char *y) {

char *str;
str=x;
x=y;
y=str;
printf("%s, %s\n", x, y);
}

void main(){

char *p="Vishvadeep";
char *q="Gothi";
fun(p,q);
printf("%s, %s", p, q);

}
```

Output

```
/tmp/ew38fPguBE.o
Gothi, Vishvadeep
Vishvadeep, Gothi
```

# Difference Between Call By Value and Call By Reference

main.c

```c
// Online C compiler to run C program online
#include <stdio.h>

void call_by_reference (char **x, char **y) {

char *str;
str=*x;
*x=*y;
*y=str;
}

void call_by_value (char *x, char *y) {

char *str;
str=x;
x=y;
y=str;
}


void main(){

char *p="Vishvadeep";
char *q="Gothi";
call_by_value(p,q);
printf("Call by Value -> %s, %s\n", p, q);
call_by_reference(&p,&q);
printf("Call by Reference -> %s, %s\n", p, q);
}
```

Output

```
/tmp/ew38fPguBE.o
Call by Value -> Vishvadeep, Gothi
Call by Reference -> Gothi, Vishvadeep
```

C: > Users > Prince > test.c > bar(char **, char **)

```c

#include <stdio.h>
void bar(char** x, char **y)
{
    char* str = *x;
    *x = *y;
    *y = str;
}
void foo(char* x, char *y)
{
    char* str = x;
    x = y;
    y = str;
}
int main() {
    // Write C code here
    char* p = "hello";
    char * q = "world";
    foo(p, q);
    printf("%s %s", p, q);
    bar(&p, &q);
    printf("\n%s %s", p, q);
    return 0;
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
PS C:\Users\Prince> gcc test.c
PS C:\Users\Prince> .\a
hello world
world hello
PS C:\Users\Prince>
```

It's worth noting that, if you want to swap the values that the pointers in the main function point to, you need to pass the address of the pointers, not the pointers themselves, and use dereferencing inside the function to change the values.

# Insertion in Array

- Always at the end
- Based on index
- Complexity of insertion always at the end
- Rumtime -> Theta(1)
- Space -> Theta(1)

## Insertion based on index

### Conditions

- Check for overflow condition. If not then check following conditons.
  - Index 'i' is between (LB) and (UB+1)
  - Check if we have to shift elements or not.
  - Empty index 'i' and input the new value 'item' in it.

### Algorithm

```
Insertion(A[], LB, UB, n, item, i);
// item -> new element to be inserted. i -> index where to be inserted. n -> count o
if(UB ==array_UB) //array_UB -> The size of the array
{
    printf("Overflow");
    return;
}
if(i <LB || i>(UB+1))
{
    printf("Invalid index");
    return;
}
for(k=UB;k>=i;k--)
{
    A[k+1]=A[k]; // Shifting to the right.
}
A[i]=item;
UB++;
n++;
```

- Complexity of insertion based on index
- Rumtime -> O(n)
- Space -> Theta(1)

# Deletion in Array

- Always last element
- Based on index

## Always last element

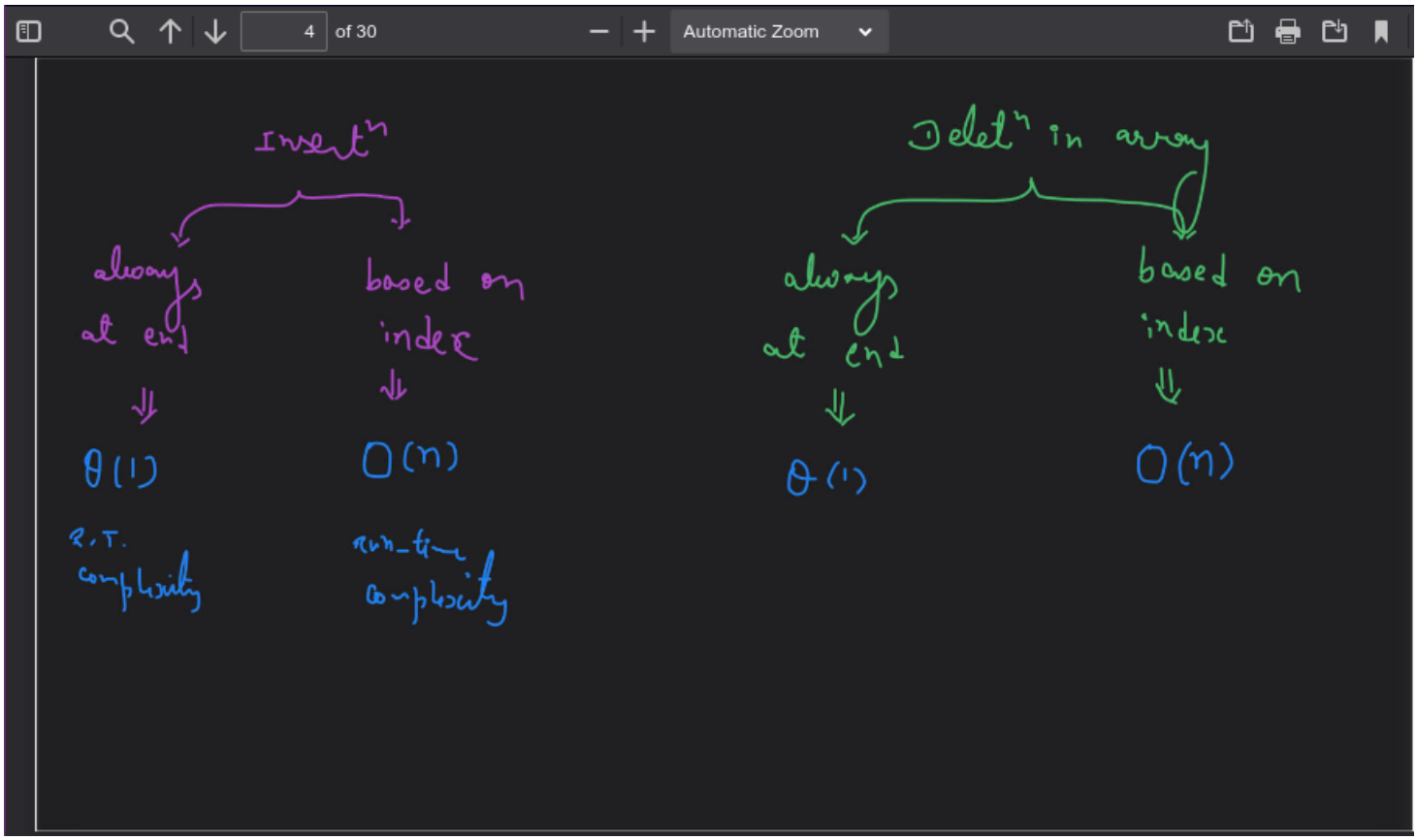
```
if(UB == -1)
{
    printf("Underflow");
    return;
}
UB--;
n--;
```

- Complexity of deletion always last element
- Rumtime -> Theta(1)
- Space -> Theta(1)

## Based on index

- index 'i' range -> LB <= i <= UB.
- Complexity of deletion based on index
- Rumtime -> O(n)
- Space -> Theta(1)

# Summary of complexities of insertion and deletion in Array




# Insertion, Deletion questions

- Insertion always at the start(LB) -> **Theta(n)**. Since we have to right shift all of the 'n' elements by 1 index.
- Deletion always at the start(LB) -> **Theta(n)**. Since we have to left shift all of the 'n' elements by 1 index.

# Finding min. element in Array. (21)

# Finding max. element in Array.

> Both algorithms and their complexities are in PDF(21_Finding_min_Max_In_Array).

- Runtime complexity -> Theta(n)
- Space complexity -> Theta(1)

# Tournament Method

> Even if the no. of players are odd or even in the tournament, the no. of comparison regardless is *(n-1)* comparisons, where *n* is the no. of players in the tournament.
> We need an array to store the results. Array size needed to store the results from the previous round, is **Ceil(n+1/2) value**. We are requiring more storage here. Algorithmns need additional space.
> As we need space to perform tournament method, hence it is not used that often because of the space limitation.

- Space Complexity -> Theta(n) or linear complexity. [Since we exactly need an array of size n/2 or (n+1)/2]
- Time complexity -> Theta(n).
- *Formula for finding comparisons required* -> n -1

## Finding both min and max using tournament method

- *Formula for finding comparisons required Method-1* -> n-1 +n-1 = *2n-2*
- *Formula for finding comparisons required Method-2* -> n/2 + n/2-1 + n/2-1 = (3n)/2 -2 = **1.5n -2**

> This is better than normal tournament method where we find min and max separately.

- When 'n' is odd:
- **Formula** -> Ceil((3n/2)) -2

# Finding second minimum

- *Formula for finding comparisons required Method-1* -> n-1 + n-2 = **2n -3**
- *Formula for finding comparisons required Method-2* -> n-1 + logn-1 = **n+logn-2**

# Questions before DPP(20_Doubt_Solving_Session)

- 1 -> B [Theta(logn)] [Got it wrong, thought Theta(n) and the options were wrong as well]
- 2 -> B [Theta(logn)] [Got it wrong, thought (O(n)) and the options were wrong as well]
- 3 -> B [Theta(logn)] [Got it wrong, thought (O(n)) and the options were wrong as well]

# DPP 1 (20_Doubt_Solving_Session)

- 1 -> [Had to write an algorithm].
- 2 -> 1032
- 3 -> Max value which is power of 2 and less than equal to n.

## Examples for question 3 in DPP 1

if n = 100, then position = 64(2^6)

if n = 200, then position = 128(2^7)

if n = 10, then position = 8(2^3)

# Find max power of 2 which is smaller than n

```
int x =Ceil(logn/log2); // Find the power
int y= pow(2,x); // Value
return y;
```

## DPP 2 (21_Finding_Min_Max_in_Array)

- 1 -> 4000 [100*40] [Faulty question]
- 2 -> [High level question]
- 3 -> 68
- 4 -> [Had to write an algorithm].
- 5 -> Theta(n) [Insertion at the beginning] [Asked in the Doubts, check there.]
- 6 -> 1032
- 7 -> Max value which is power of 2 and less than equal to n.

> The explaination of question 7 is in the above *DPP 1* questions, check above. Example is also given.

# Searching in Array (22) [Date -> 16 Jan 2023]

- Linear search
- Binary search

> In linear search, if an array has **duplicate elements**, then it will only return the **first** index/address/location of the element in the array.

- Runtime Complexities
- Best case -> Theta(1)
- Worst case -> Theta(n)
- Average case -> O(n)

## Find all index of an element using linear search

```
Linear_Search(A[],LB,UB,element)
{
    int index[UB];
    for(int i=LB;i<=UB;i++)
    {
        index[i]= LB-1;
    }
    int j=0;
    for(int i=LB;i<=UB;i++)
    {
        if(A[i] == element)
        {
            index[j]=i;
            j++;
        }
    }
    if(index[0] == -1)
    {
        printf("Element not found");
    }
    else
    {
        for(int i=0;i<UB;i++)
        {
            if(index[i]!=-1)
            {
                printf("%d\t", index[i]);
            }
        }
    }
}
```

- Runtime complexity
- Average case -> Theta(n)
- Best case -> Theta(n)
- worst case -> Theta(n)

> In linear search, whatever the formating of elements are in the array like duplicate, unique, any order/sort of elements, it will work fine.

# Binary Search

> If array is sorted then only we can implement binary search.
> Binary search works only on sorted array.

- while(LB >= UB) -> Element not found condition.

## Conditions

1. Find mid
2. if(item == A[mid]) return mid
3. if(item < A[mid]) OR if(item > A[mid]) -> Which matches the condition
4. UB=mid -1 OR LB=mid+1
5. Calculate new mid
6. Go to step 2.

## Algorithm

```
Binary_search(A[],item, LB,UB)
{
    int low=LB,high=UB;
    // Alternate way of writing the algo. Both give the same result.
    // int mid=(low+high)/2;
    int mid;
    while(low<= high)
    {
        mid=(low+high)/2;
        if(A[mid] == item)
        {
            return mid;
        }
        else if(item < A[mid])
        {
            high=mid-1
        }
        else
        {
            low=mid+1;
        }
        // mid=(low+high)/2;
    }
    return LB-1;
}
```

- Runtime complexity -> O(logn)

# Finding first and last appearance of given array in sorted array

> As it is a sorted array, we can use *Binary Search* here. The runtime complexity of binary search is *logn*.

- Find first appearance -> log n
- Find last appearance -> log n
- Total comparisons -> logn + logn -> 2logn -> O(logn)
- Frequency = First appearance index - last appearance index.

# 22_Searching_in_Array Questions

- 1 -> B [Theta(logn)]
- 2 -> B [O(logn)]
- 3 -> B [O(logn)]

# Array (23)

- Only stores similar datatypes -> Basic need, not a disadvantage.

## Advantage

- Random or direct access using indexes

## Disadvantage

- Fixed size.
- Stored in consecutive location.

> To solve the disadvantages of array, we came up with linked list.

# Linked List

- **linear or sequential order maintained using pointers/links**.
- Linked list contains a node. Node has **two** parts.
  - Element part
  - link/address part which points to the next node.

> Last node contains 'NULL' at the link part of the node which marks the end of the linked list. Address of first node is stored in list/head/start pointer. This is used to access the whole linked list.

## Empty List condition

```
if(list == NULL) or if(!list)
```

## Single Node

```
if(list ->link == NULL) or if(!list ->link)
```

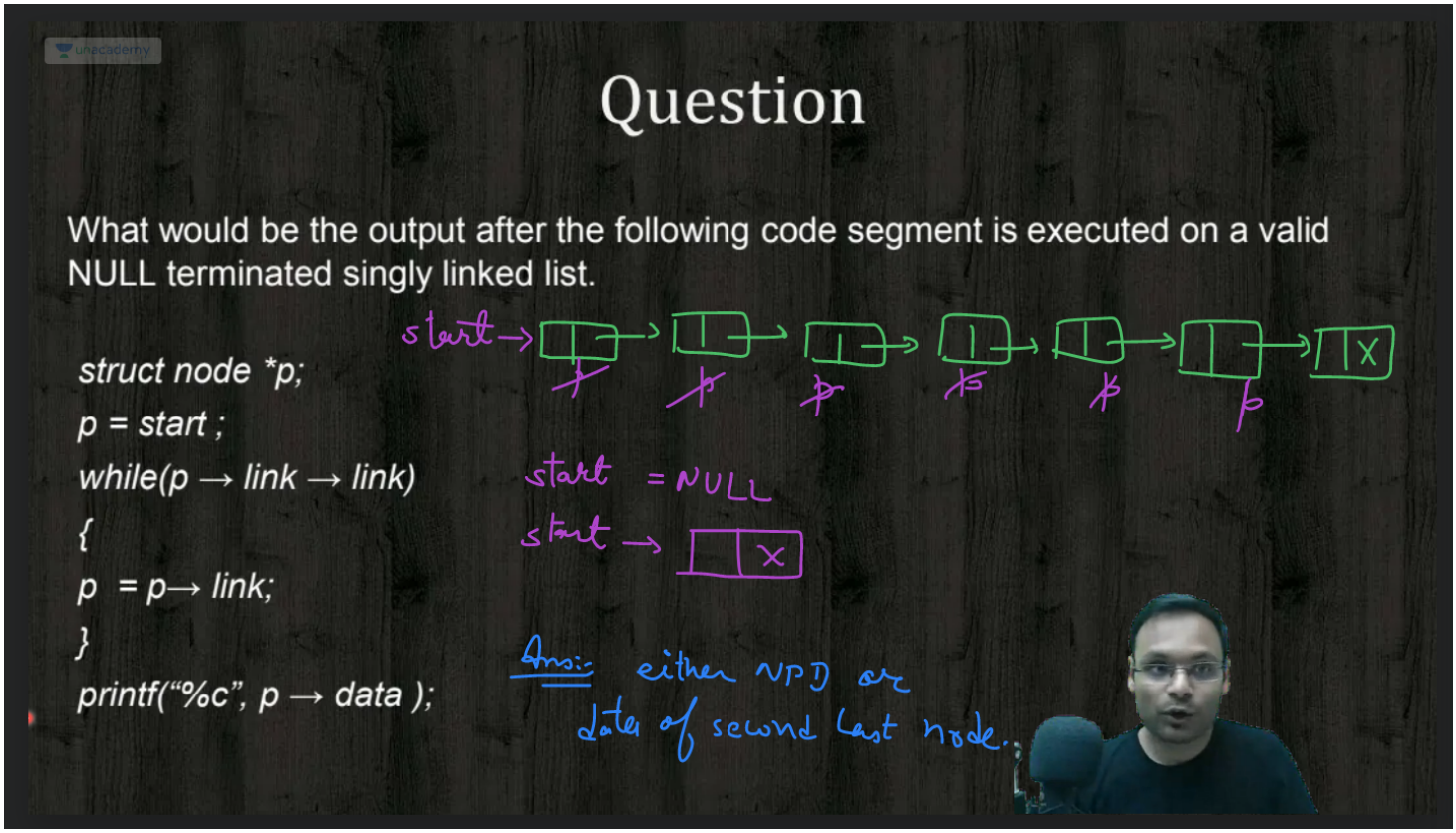
> Practice the linked list question in 23_Linked_list_basics PDF, Page 12 onwards.

# Question in 23_Linked_list_basics PDF

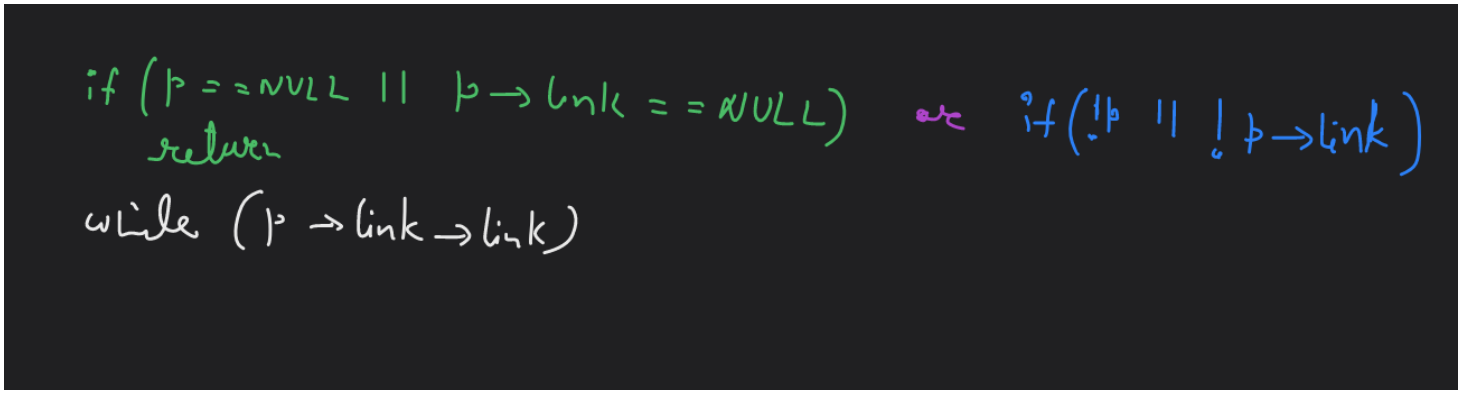
- 1 -> e [Either null pointer dereferencing error or data of 2nd last node get printed]
- 2 ->

> null pointer dereferencing errors or NPDs as the problem hasn't handled the base conditions like,

- start =NULL;
- Linked list has one element only. start points to only node only.



### To remove the NPDs errors




## DPP 3 (23_Linked_list_basics)

> These questions are in the next DPP as well, which is DPP 4 in *25_Linked_list_implementation* PDF file. Some extra questions are present as well. Check that out. I will complete all of them there.

## Doubt and questions (24)

> All of the previous DPP questions are solved. Check the above DPPs for solutions and questions.

## Null Pointer Dereferencing (25)

- NULL ->data
- NULL -> link
- if(p->link != NULL)

> Error.

## Traversing in Linked List.

- Runtime complexity -> **Theta(n).**

## No. of elements in Linked List

- Runtime complexity -> **Theta(n).**

# Sum of elements in linked list

- Runtime complexity -> **Theta(n).**

# Min element in linked list.

- Runtime complexity -> **Theta(n).**

# Address of last node in linked list

- Runtime complexity -> **Theta(n).**

# Valid NULL terminated linked_list

- start=NULL

> There is 'start' pointer so linked list exists but it is empty or has zero(0) nodes. As it has a 'start' pointer which means there exists a linked list which is empty or has zero(0) nodes, but a linked list exists never the less.

# Insertion in Linked List

> We can insert a node at any postion, beginning, end or at any given position of the linked list.

- Whenever a new node to be inserted:
  1. First create a new node dynamically using malloc()
  2. Insert the newly created node

## Creating a new node

```
struct node* n =(struct node*) malloc(sizeof(struct node));

if(n == NULL)
{
    printf("Out of memory");
}
```

> If **all of the memory** is used up, then **malloc()** will return **NULL**.
> If *n* == *NULL* is true then, the system is out of memory.

# Different insertion in linked list

- **Beginning** -> Runtime complexity = *Theta(1)*
- **After a given node** -> Runtime complexity = *Theta(1)*
- **At end** -> Runtime complexity = *Theta(n)*
- **At end when last node is given** -> Runtime complexity = *Theta(1)*
- **Before a given node** -> Runtime complexity = *O(n)*

# DPP 4 (25_Linked_list_implementation)

> Solved in *28_Doubt_Clearing_session* PDF file.

- 1 -> [Circular list not yet done]
- 2 -> 10 [count of the remaining nodes in the list. It is removing the duplicate elements from the list.]
- 3 -> D [Error or returns 1]
- 4 -> [Cross or axe] [Solved in the PDF file above]
- 5 -> [Cross or axe] [Solved in the PDF file above]
- 6 -> f
- 7 -> a [Got it wrong, thought 'e' was the answer]
- 8 -> error [NPD is the error]
- Answer of Question **4** from above:

```
x->next=x->next->next;
```

- Runtime complexity ->*Theta(1)*
- Answer of Question **5** from above:

```
p=start;
while(p->link->link!=x)
{
    p=p->link;
}
p->link=x;
```

- Runtime complexity ->*O(n)*

# Deletion in linked list (26)

- **Beginning** -> Runtime complexity = *Theta(1)*
- **of a given node** -> Runtime complexity = *O(n)*
- **at the end** -> Runtime complexity = *Theta(n)*
- **at the end when address of last node is given** -> Runtime complexity = *Theta(n)*

## Deletion at beginning

```
struct node* p=strat;
start= start -> link;
free(p);
```

> *free(p)*, the pointer 'p' was pointing to the node and the space take by that node would be set free. The node will neither exist in memory nor in the linked list.

- free(p) -> keyword.

## Deletion of a given node

```
struct node *p=start;
while(p->link != loc)
{
    p=p->link;
}
p->link=loc->link;
free(loc);
```

> Only valid way to delete a given node.

### More ways to delete a given node

> Interview question, **delete a given node but we cannot use an extra pointer or a new pointer, which will loop to one place before 'loc' position**. Below are the ways to tackle this problem/question asked in interviews.

### Invalid Method 1

```
start = loc ->link.
```

### Invalid Method 2

```
loc -> data = loc->link->data;
loc->link = loc->link->link;
```

> Both are invalid ways of deleting a given node. Normal way is the only valid way.
> Look at the *26_Types_of_linked_list* PDF file, the code and the explaination is there.

## Deletion at the end

- Runtime complexity -> Theta(n)

## Deletion at the end when last node address is given

```
struct node* p =start;
while(p->link->link)
{
    p=p->link;

}
p->link=NULL;
free(loc);
```

> Even if the address of last node is given, we still need to traverse the list and get to the 2nd last element of the list.

# Searching in linked list

> Returns address of the node where the element found in linked list.

## Linear search in LL

- Rumtime complexity -> O(n)

## Binary search in LL

> Linked List should be sorted.
> Binary search in linked list is not possible in *logn* runtime complexity. As we have to linearly traverse the list to go to the mid element of list. Hence to find mid element we have to do *n/2* comparisons every time, which is *O(n)* runtime complexity. As it is not possible to reach the mid element in *constant or O(1)* runtime. To reach mid element, we need linear or *O(n)* time

> complexity.
> Binary search in linked list can not be as efficient as it is on array because in linked list, reaching to mid node is not possible in constant time.

- If someone asks can we implement binary search in linked list?

> Yes we can implement binary search in linked list. But the runtime complexity will not be in *O(logn)* but will be in *O(n)*.

# Sets

- Union(U) -> Both sets are combined but common/duplicates are removed. Uniques are kept only.
- Intersection() -> Only the common values from both the sets are kept.
- Membership -> If an elements belongs to a given set or not.
- Cardinality -> To find the no. of elements in a given set.

> Good PYQ question from gate 2004, revise it a very important question about linked lists and sets in *26_Types_of_linked_list* PDF file.

# Reverse a linked list

```
struct node* p=start;
struct node* p=NULL;
struct node* r;
while(p)
{
    r=q;
    q=p;
    p=p->link;
    q->link=r;
}
start=q;
```

- Runtime complexity -> Theta(n) [Linear complexity]
- Space complexity -> Theta(1)

# Disadvantage of LL

1. The link part of last node is not utilized
2. The address of predecessor/previous node is not know

3. Stepping backward is not possible

- 1 is solved by circular linked list.
- 2 and 3 is solved by doubly linked list.

# Circular Linked list

```
struct node* p=start;
while(p->link)
{
    p=p->link;
}
p->link=start;
```

- Runtime complexity -> Theta(n)

# Header List (27) (17/01/2023)

> It contains some summary information. It is a special first node called as header node.
> Some types of summary information includes no. of nodes, sum of the elements.
> We add a node, which doesn't include an element but has information related to the summary of the linked list.

- Whenever an insertion happens then, (list->data)++;
- Whenever an deletion happens then, (list->data)--;

## Condition when header list is empty

## Types of header list

- Grounded list -> Last node has link as NULL.
- Circular list -> The last node's link is pointing to the starting or first node i.e the header node of the linked list.

# Traversal in header list.

## In grounded

- Runtime complexity -> Theta(n)

## In Circular

- Runtime complexity -> Theta(n)

# Doubly linked list

> Every node stores two addresses and a data. One pointer points to the next node and the other pointer points to the previous node.
> In some doubly linked lists, we maintain a pointer on the last node of the list for stepping backwards in the list.

- There could be two varients of doubly linked list

1. Only one pointer pointing to the first node of the list.
2. Two pointers, one pointing to the first node of the list and the other one pointing to the last node of the list.

# Insertion in doubly linked list.

## At benginning

- Runtime complexity -> **Theta(1)**

## After a given node.

```
n->next= loc->next;
n->prev=loc;
loc->next->prev=n;   // OR n->next->prev =n; Same thing
loc->next=n;
```

- Runtime complexity -> **Theta(1)**

> In linked list insertion, always update/add the links/pointers of the new node which is being inserted into the list first then change/modify/update the existing links/pointers in the list. Othewise we will loose access to the already existing links/pointers in the list. Sequence is important otherwise we will loose access to some nodes in the list.

## Before a given node.

```
n->prev=loc->prev;
n->next=loc;
loc->prev->next=n;
loc->prev=n;
```

- Runtime complexity -> **Theta(1)**

## At the end when address of last node is given.

```
n->next=NULL;
n->prev=last;  // 'last' is the pointer pointing to the last node of list.
last->next=n;
last=n;
```

- Runtime complexity -> **Theta(1)**

## At the end when address of last node is not given.

```
n->data=item;
p=first;
while(p->next)
{
    p=p->next;
}
n->next=NULL;
n->prev=p;
p->next=n;
// p=n;  Not needed as we are not maintaining the address of last node.
```

- Runtime complexity -> **Theta(n)**

# Deletion in Doubly linked list

## First node

```
first=first->next;  // 'first' is a pointer pointing to the first node of the list.
first->prev=NULL;
```

- Runtime complexity -> **Theta(1)**

## Given node

```
loc->prev->next=loc->next;
loc->next->prev= loc->prev;
// loc->next=NULL; Not needed they will be removed the list.
// loc->prev=NULL; Not needed they will be removed the list.
free(loc);
```

- Runtime complexity -> **Theta(1)**

## Last node, when address of last node given

```
p=last;
last=last->prev;
last->next=NULL;
free(p);
```

- Runtime complexity -> **Theta(1)**

## Last node, when address of last node is not given

```
p=start;
while(p->link->link)
{
    p=p->link;
}
free(p->next);
p->next=NULL;
```

- Runtime complexity -> **Theta(n)**

> All of the algorithms for intersion, deletion in doubly linked list have *Theta(1) or constant* space complexity. We have hardly used one(1) struct node* type pointer in them.

## For insertion

> When we have inserted a node in the **middle** wheather it is before or after a given node, we are **updating four(4) links**.
> For insertion at **beginning** we are **updating three(3) links**.
> For insertion at the **end** we are **updating three(3) links**.

## For deletion

> When we have deleted a node in the **middle** wheather it is before or after a given node, we are **updating two(2) links**.
> For deletion at **beginning or first node** we are **updating one(1) links**.
> For deletion at the **end or last node** we are **updating one(1) links**.

# Recursion in linked list

> Question 6 from *27_Types_of_linked_list_part_II* PDF file.

## Questions from *27_Types_of_linked_list_part_II* PDF file.

> Solved in *28_Doubt_Clearing_session* PDF file.

- 1-> Already done
- 2-> Omit for now. [Solved below. Interesting and tough question]
- 3-> dcba [Printing the elements of linked list in reverse order]
- 4-> B [Reversing the list]
- 5-> A [Got it wrong, thought 'B' is the answer]
- 6-> Returns the address of last node or NPD error.

> In question 4, whatever function we are executing, we can access the local variables of that function call only.
> All of these questions were good. Need to revise and know them to able to solve them in the future. Solve them in revision as well.

# Doubly Circular linked list (28)

> We can make any node as the 'first' pointer. No problem in that. Benefit of traversal in both the directions. We can travel from anywhere or any node and traverse back to that same node as well.

## Reverse doubly Circular linked list

# Linked list PYQS

> Question in *28_Doubt_Clearing_session* PDF file. At the end of the file. Solve them.

- 1-> C [Multiple pointers, 4 to be exact]
- 2-> Have to wring an algorithm
- 3-> B [Binary Search]
- 4-> D [I and III]
- 5-> B [Doubly linked list]
- 6->
- 7-> [Swapped i+1 and i+2 list data]
- 8-> Have to wring an algorithm
- 9-> D [n]
- 10-> B
- 11-> C [not possible with a single pointer]
- 12-> D [union, intersection]
- 13-> B [2,1,4,3,6,5,7]
- 14-> B
- 15-> B
- 16-> A
- 17-> B [Theta(1), Theta(n)]
- 18-> C [Theta(n^2)]
- 19-> C [error]

# Algorithm for question 2

```
struct node* p=head;
while(p->next!=x)
{
    p=p->next;
}
p->next=Y;
Y->next=X;
```

- Runtime complexity -> O(n)

# Algorithm for question 7

```
struct node* p=head;
n->next=p->next;
p->next=n;
```

> 'n' is the address of the new node to be inserted. 'p' is the pointer from the question.

- Runtime complexity -> Theta(1)

# Queue (29) (18/01/2023)

- FIFO -> First In First Out.

> It is a linear data structure, in which insertion is done from one end(Rear end) and deletion is done from other end(front end).

- Enqueue -> Insertion in queue
- Dequeue -> Deletion in queue.

# Implementation of queue using array

- Two implementation of queue in array:

1. **Linear queue** -> Insertion can be done on next index of 'Rear' linearly. Insertion can be done only after the index of 'Rear'. It cannot utilize the first/previous empty spaces in the array. Not better space utilization in linear queue.
2. **Circular queue** -> If array is full then the insertion is circularly taken to the starting of the array. Full utilization of the space happens here. No space is left empty.

- We need 2 index variables

1. **Front** -> Which stores index of front element of queue. The first element which was inserted into the queue, that element's index is stored there.
2. **Rear** -> Which stores index of rear element of queue. The last element which was inserted into the queue, that element's index is stored there.

# Insertion or enqueue in queue

## When queue is empty

```
front=read=-1;
```

> For insertion, rear index changes.
> For deletion, front index changes.

> For the first insertion in the queue, we set the front and rear variables to '0' as it is the first, second and the last element of the queue.

## When queue is full Overflow

```
if(front ==0 and rear==N-1) // Normal case.
{
    printf("Queue is full, overflow condition");
}
```

> Queue is full and no more insertion into the queue is possible.

- N -> Size of the array.

## Another overflow condition but for circular queue

```
front= (read+1)%N;  //N -> size of the array.
```

# Deletion or dequeue in queue

> We don't need to provide any element in dequeue() as we know dequeue or deletion in queue happens on the **first element** of the queue. Front element or the first inserted element in the queue will be deleted.

# Algorithm for insertion or enqueue in circular queue

```
Enqueue(item,queue[],n,front, rear); //n-> size of the array.
int queue[5];
if(front ==(rear+1)%n)
{
    printf("Overflow");
    return;
}
if(front==rear==-1) // Special condition, when inserting the first element in queue
{
    front=rear=0;
}
else
{
    rear= (rear+1)%n;  // Normal condition, after first insertion.
}
queue[rear]=item;
```

- Runtime complexity -> **Theta(1) or constant**
- Space complexity -> **Theta(1) or constant**

# Algorithm for deletion or dequeue in circular queue

```
if(front==rear==-1)
{
    printf("Underflow or queue empty");
    return;
}
if(front==rear) //special condition, when only one or last element left in queue.
{
    front=rear=-1;
}
else
{
    front=(front+1)%n;
}
```

> Sometimes they ask to store the deleted values in some place, as they will be used afterwards. Use the code below.

```
int deletedItem;
if(front==rear==-1)
{
    printf("Underflow or queue empty");
    return;
}
deletedItem= queue[Front]; //Storing the deleted item.
if(front==rear)
{
    front=rear=-1;
}
else
{
    front=(front+1)%n;
}
```

- Runtime complexity -> **Theta(1) or constant**
- Space complexity -> **Theta(1) or constant**

> Same for both even if we are storing the deleted item or not.

# Implementation of queue using linked list

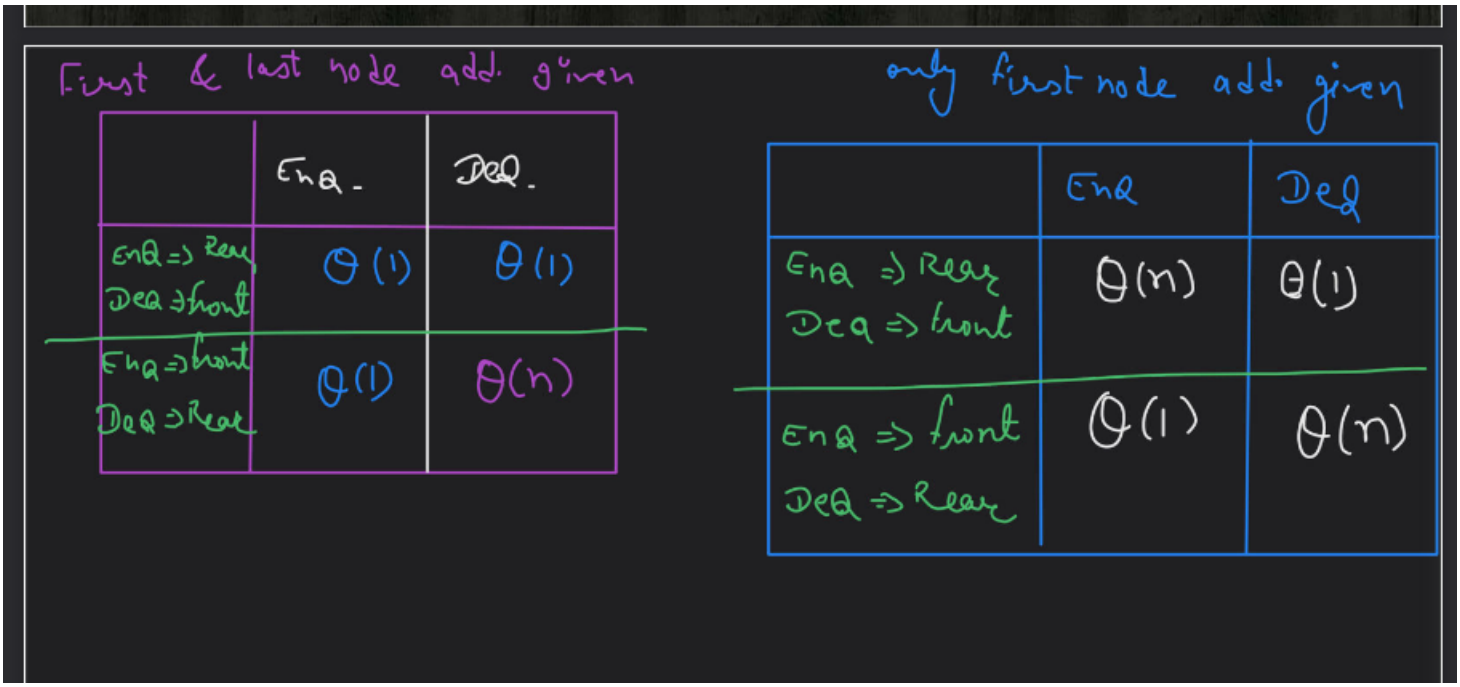
- **Enqueue** -> Insertion at the end -> **Theta(1)**
- **Dequeue** -> Deletion from the front -> **Theta(1)**

# Other Functions on queue (30) (19 Jan 2023)

- QueueFront()
- QueueRear()
- IsEmptyQueue()

> Some additional functions which can be given with queue.
> Default functions of queue are *Enqueue* and *Dequeue* only.

# Usefull runtime complexities of queue

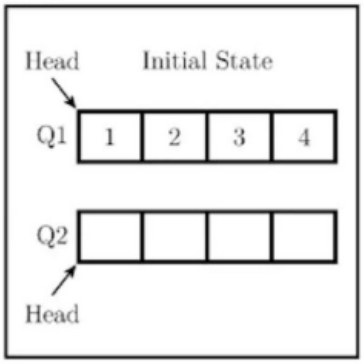


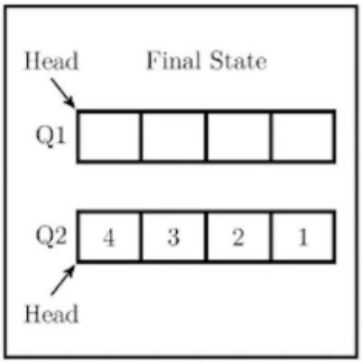

# MultiDequeue

> Many good questions have come in GATE PYQs. Check them out in revision. They are in *30_Queue_Types_of_Queue*.

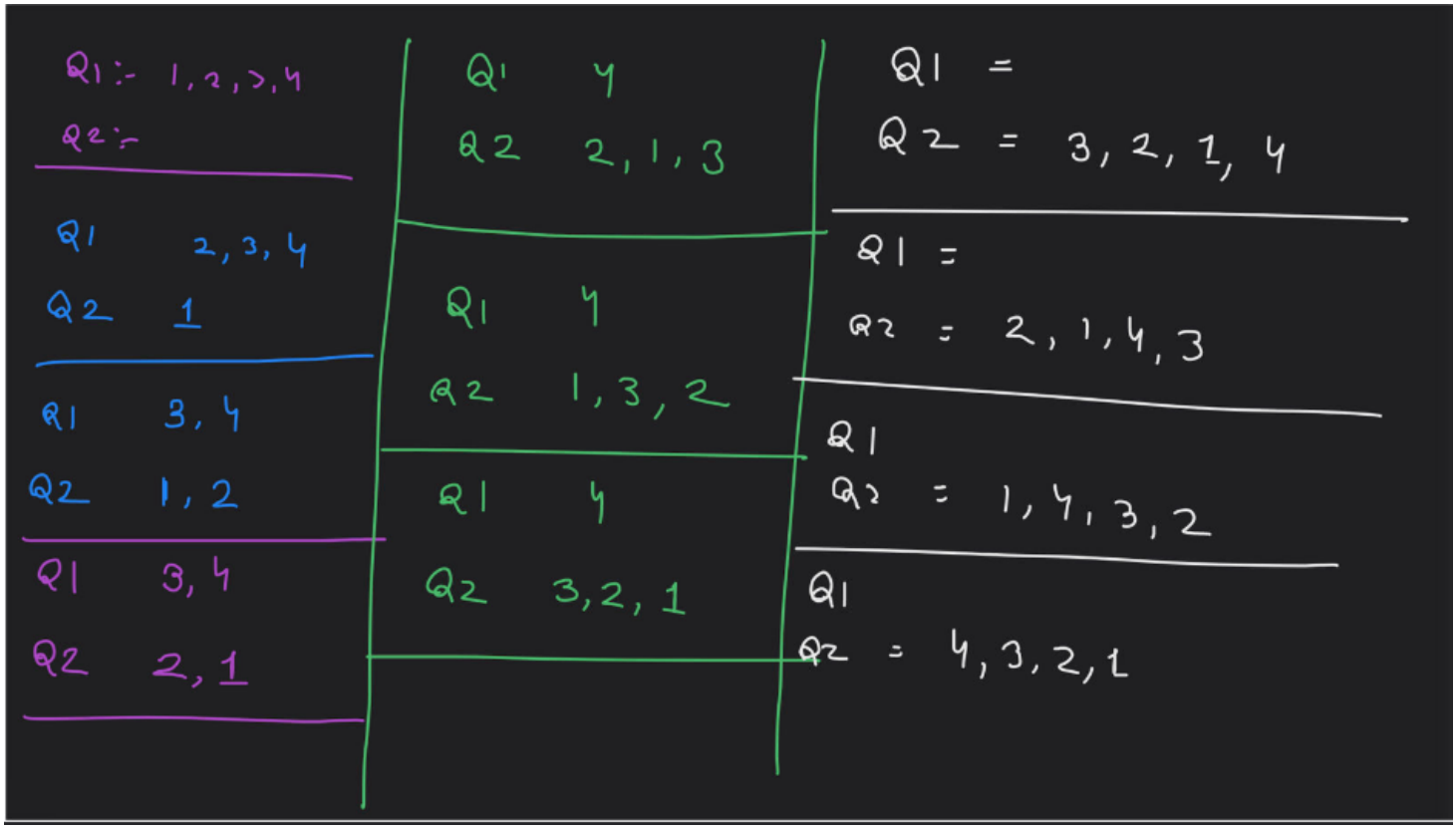
- One of the questions:

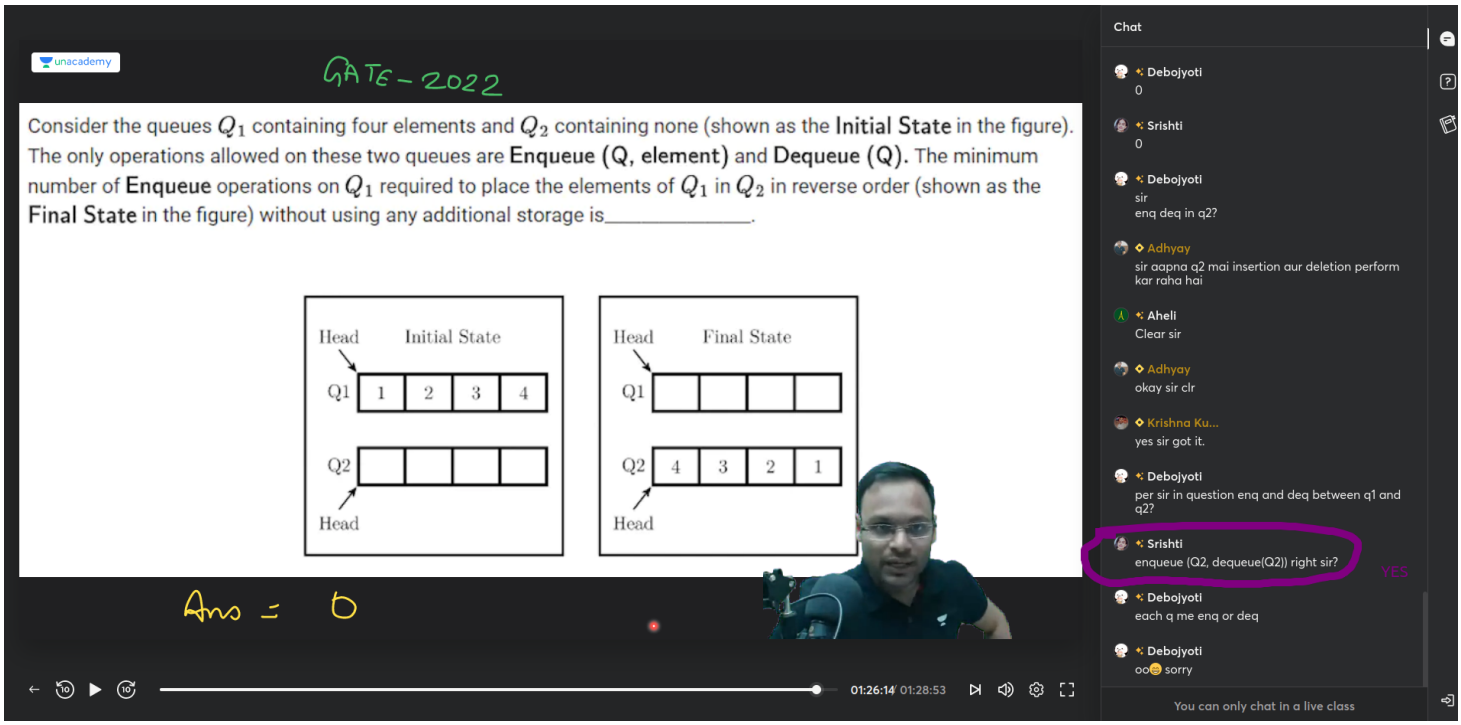
Consider the queues $Q_1$ containing four elements and $Q_2$ containing none (shown as the **Initial State** in the figure). The only operations allowed on these two queues are **Enqueue (Q, element)** and **Dequeue (Q)**. The minimum number of **Enqueue** operations on $Q_1$ required to place the elements of $Q_1$ in $Q_2$ in reverse order (shown as the **Final State** in the figure) without using any additional storage is_____.




Ans = b



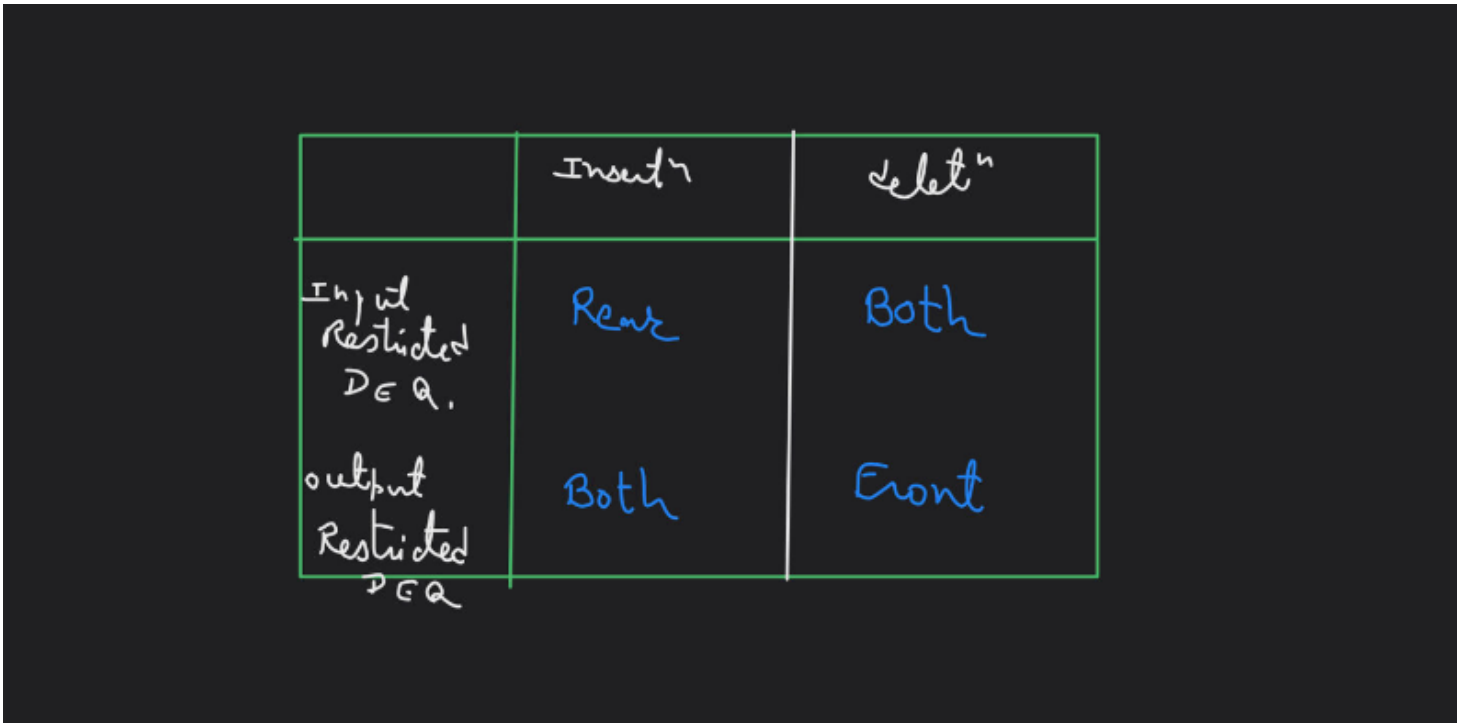

> The **reversing** that is happening in image 2. It is happening in queue2 or Q2, *where the elements are first being deleted or de-queued and then immediately inserted or enqueued into Q2*. Which is mentioned in image 3, **underlined with color**. This is how it is happening.

# Double Ended Queue (31)

> In this queue, insertion and deletion both can be performed from both ends(front and rear).

## Types

- Input Restricted Double ended queue
- Output Restricted Double ended queue




# Priority Queue

> In priority queue, the deletion in the queue doesn't happen to the first inserted element of the queue. It happens to the highest priority of the element.
> Each element of queue has associated priority and while deletion, the element with highest priority gets deleted.
> Delete element with maximum priority.
> If two elements have same priority then first inserted element gets deleted.

- Smallest no. highest priority
- Largest no. highest priority

## Priority queue implementation

> Implement standard queue using priority queue.

# Stack

- LIFO -> Last in fast out.

> It is a linear data structure in which insertion and deletion both are performed from the same end. The end where both operations are performed is called as the top of the stack.

- Insertion -> PUSH()
- Deletion -> POP()

# Implementing stack using array

- top -> Index which points to the last inserted element.

## PUSH() algorithm

```
PUSH(stack[],top,n,item); //n->size of the stack.

if(top==n-1)
{
    printf("Overflow, stack full");
    return;
}
top++;
stack[top]=item;
```

- Runtime complexity -> Theta(1)
- Space complexity -> Theta(1)

## POP() algorithm

```
POP(stack[],top,n); //n->size of the stack.

if(top==-1)
{
    printf("Underflow, stack empty");
    return;
}
item=stack[top];  // To store the deleted element for later use.
top--; // Deletion
```

> We have just removed the access of the element from the stack. It is not a part of stack now.

- Runtime complexity -> Theta(1)
- Space complexity -> Theta(1)

## Another implementation of stack (Bottom of stack)

> Bottom index of stack is maintained. It is a constant value.

### Underflow

```
int bottom=-1;

POP(stack[],top,n); //n->size of the stack.

if(top==bottom)
{
    printf("Underflow, stack empty");
    return;
}
```

## Overflow

```
PUSH(stack[],top,n,item); //n->size of the stack. maxtop -> 'n-1' size of the array.
if(top==n-1) // or if(top==maxtop)
{
    printf("Overflow, stack full");
    return;
}
```

# Implementing stack using linked list

> When there is only **one pointer** in the list, which is 'top' and it is pointing to the first inserted element in the list or the starting node of the list.

- **PUSH()** -> Insertion from start. -> Run time complexity -> **Theta(1)**
- **POP()** -> Deletion from start. -> Run time complexity -> **Theta(1)**
- **PUSH()** -> Insertion from last. -> Run time complexity -> **Theta(n)**
- **POP()** -> Deletion from last. -> Run time complexity -> **Theta(n)**

> The list has **two pointers** *first* and *last*. *First* is pointing to the first node of the list and *last* is pointing to the last node of the list.

- **PUSH()** -> Insertion from last. -> Run time complexity -> **Theta(1)**

- **POP()** -> Deletion from last. -> Run time complexity -> **Theta(n)**

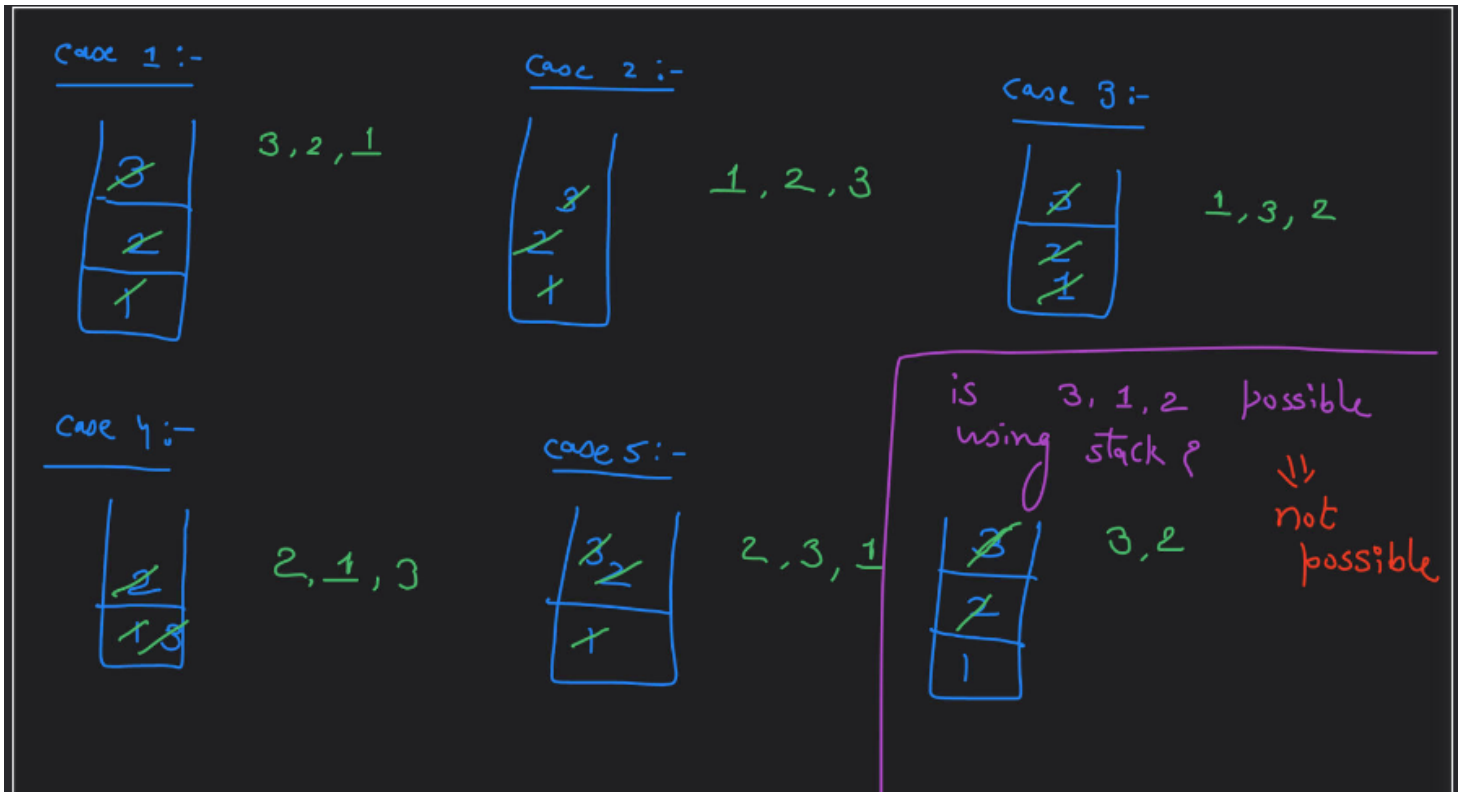
## Operations on stack

- stacktop(stack) -> Reads and returns top element of stack.
- IsEmpty(stack) -> Return **True** if stack is empty. Otherwise returns **false**.

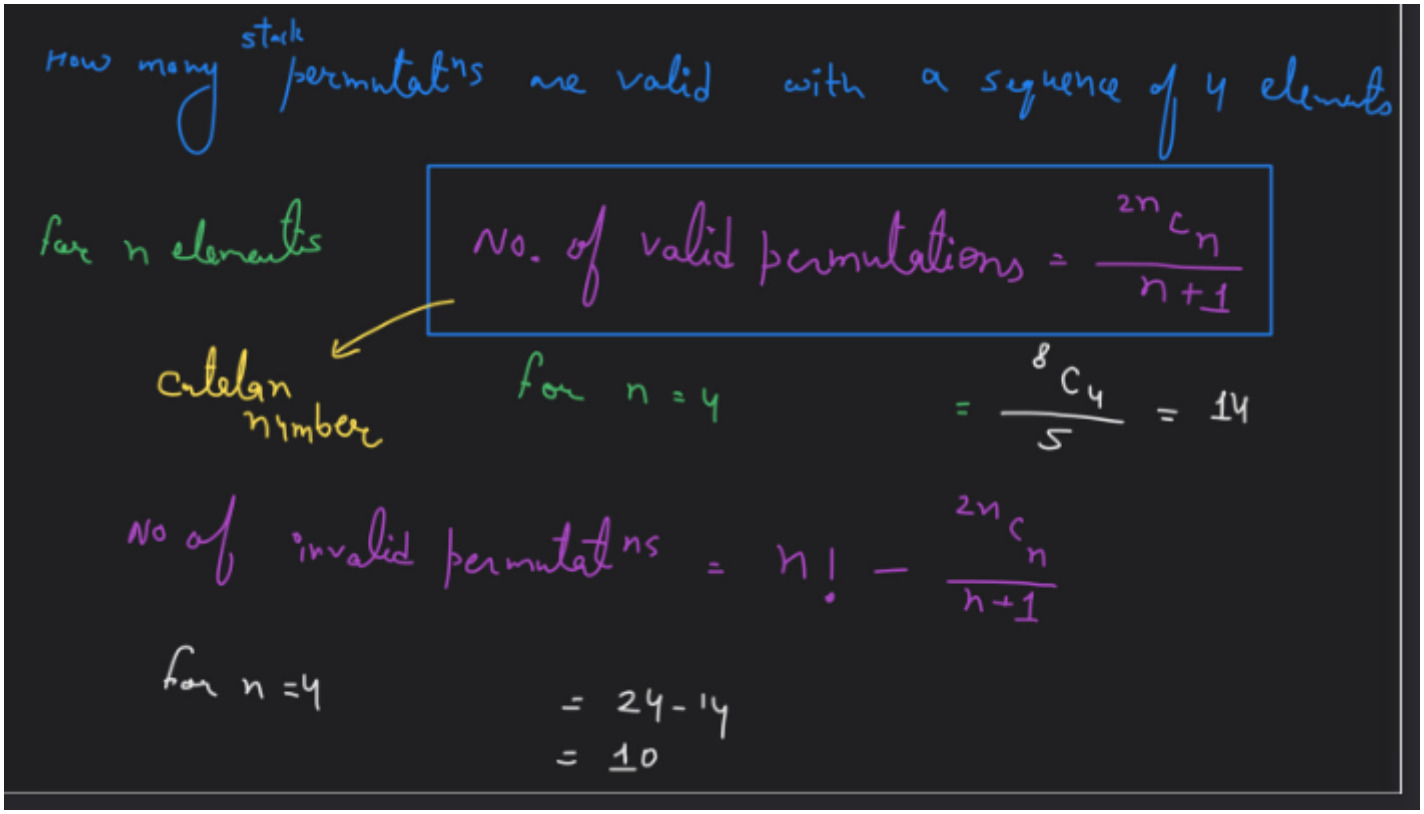
# Stack permutation

> The **different ways or different orders** in which we can *pop out elements* from the stack.

## Example



## Formula




# DPP 5 (*31_Basics_of_stack*) [20 Jan 2023]

## Questions before DPP

- 1-> D [strictly decreasing order]

## DPPs

> Solved in *32_Doubt_Clearing_Session*.

- 1-> A [ii,iii]
- 2-> A & C [1,2,4,5,3 3,5,4,2,1]
- 3-> C [3 & 1 instructions] [Discuss in a later class]
- 4-> Valid = 132, invalid=588

# Questions in *32_Doubt_Clearing_Session*

> Solved in *33_Stack_permutation_and_Advanced_Topics*.

- 1-> B [Two]

- 2-> A[n+m<=x<2n and 2m<=y<=n+m]
- 3-> D [top1=top2-1]
- 4-> * 1. if(top==n-1)

    * 2. if(top==bottom)

# DPP 6 (*32_Doubt_Clearing_Session*)

> Solved in *33_Stack_permutation_and_Advanced_Topics*.

- 1->
- 2-> B [3Y+2X]
- 3-> $16^{16[n}2]$

# Implementation of queue using stack

## Best Method

### Insertion or enqueue

```
Insertion(Q,item);

PUSH(s1,item);
```

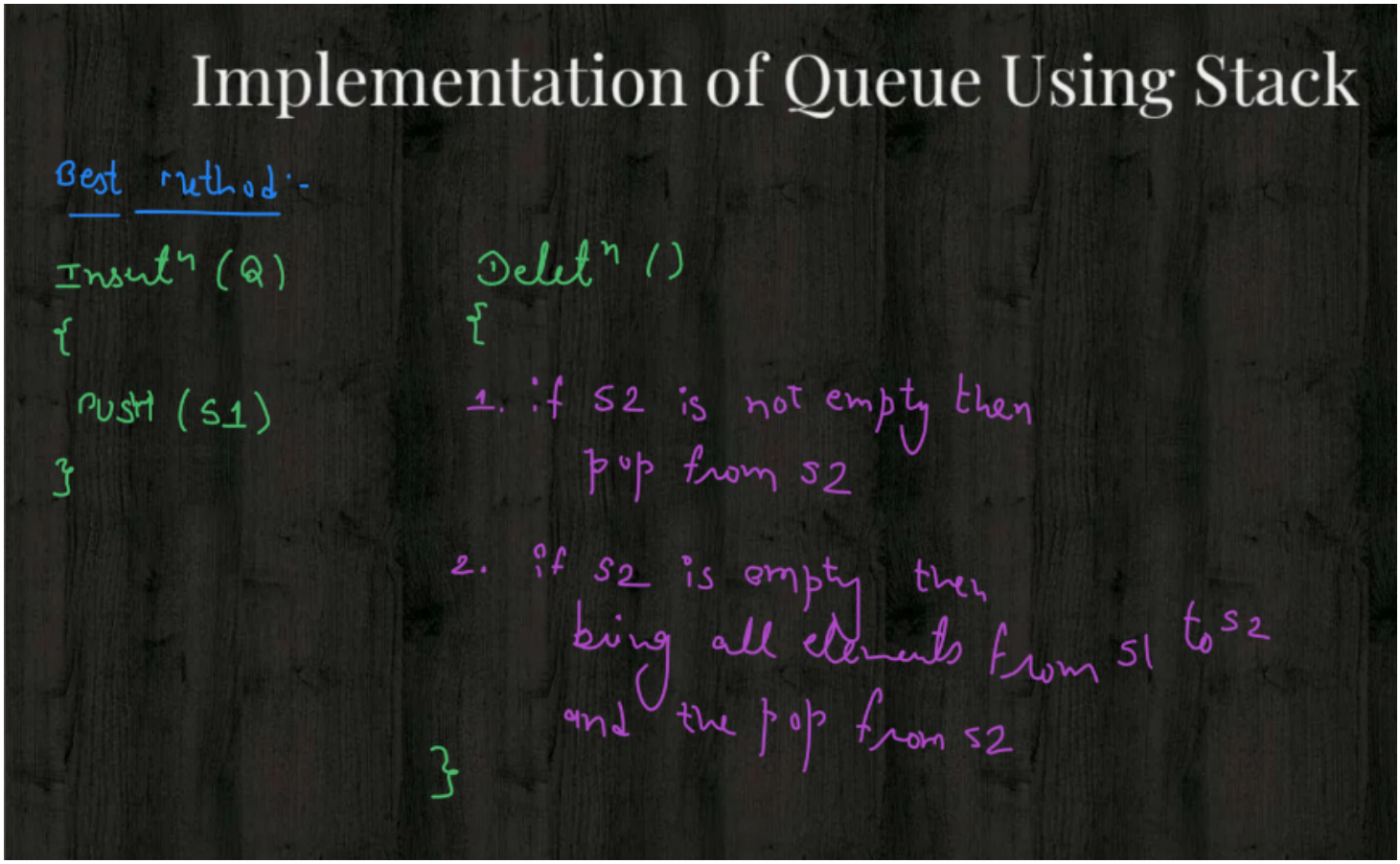
> In insertion, we are directly using PUSH(), to insert the element into s1 or first stack.

### Deletion or dequeue

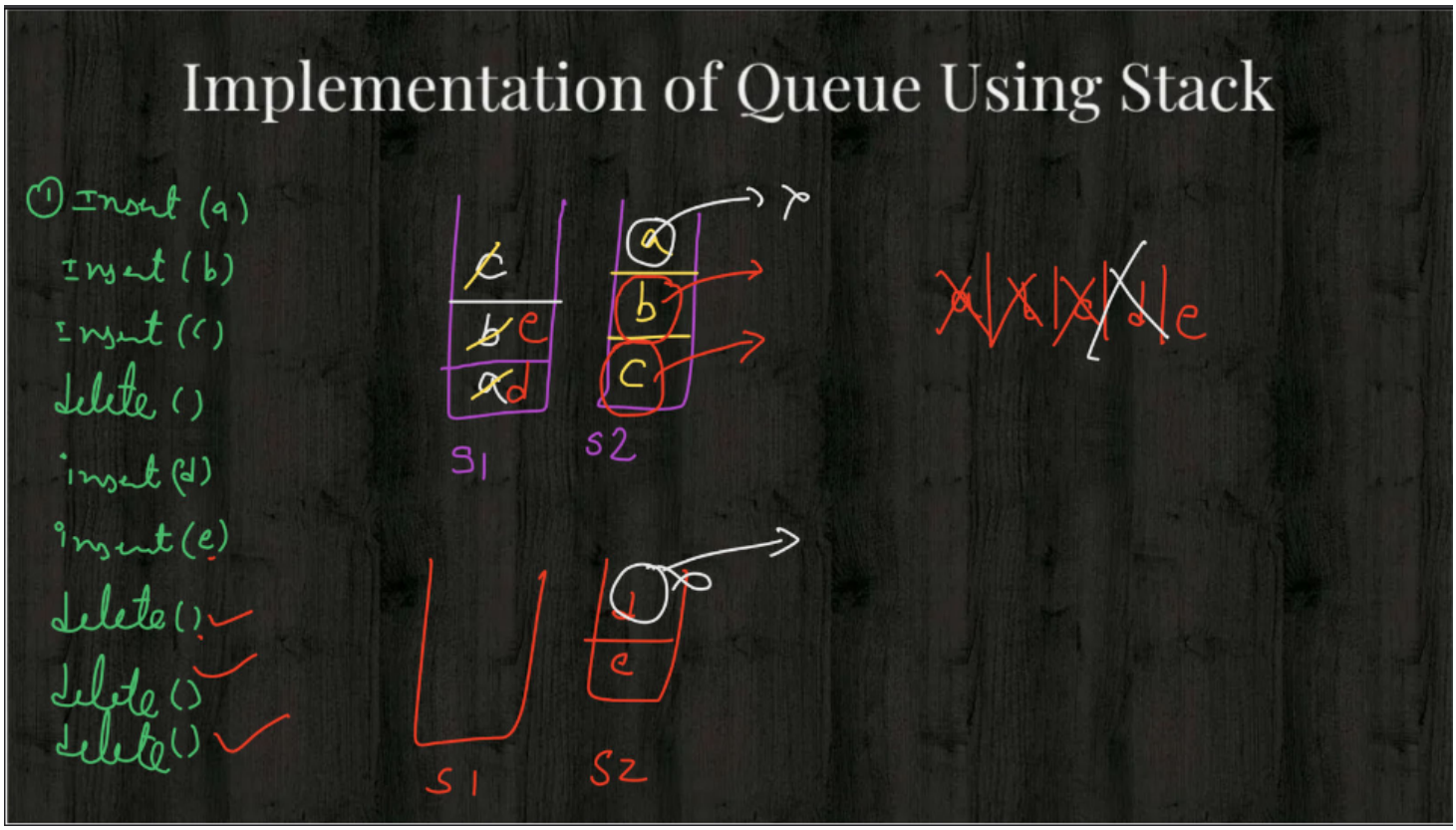
```
Deletion(Q);

if(!IsEmpty(s2))  // s2 is not empty
{
    POP(s2);   // Pop the top element from s2.
}
else
{
    PUSH(s2,s1);  // Pop the elements from s1 and then push them into s2.
    POP(s2);
}
```

> In deletion, first we are checking if s2 or second stack is empty or not.
> If it is empty then we are going to pop the remaining elements from s1 stack and push them in s2.
> Then we perform deletion but using POP() from s2.
> If s2 is not empty then we directly use POP() to pop elements from s2.



# Implementation of Queue Using Stack




## Question on Implementation of queue using stack

# Question GATE-2006

An implementation of a queue $Q$, using two stacks $S1$ and $S2$, is given below:

```
void insert (Q, x) {
    push (S1, x);
}
void delete (Q) {
    if (stack-empty(S2)) then
        if (stack-empty(S1)) then {
            print("Q is empty");
            return;
        }
        else while (!(stack-empty(S1))){
            x=pop(S1);
            push(S2,x);
        }
    x=pop(S2);
}
```

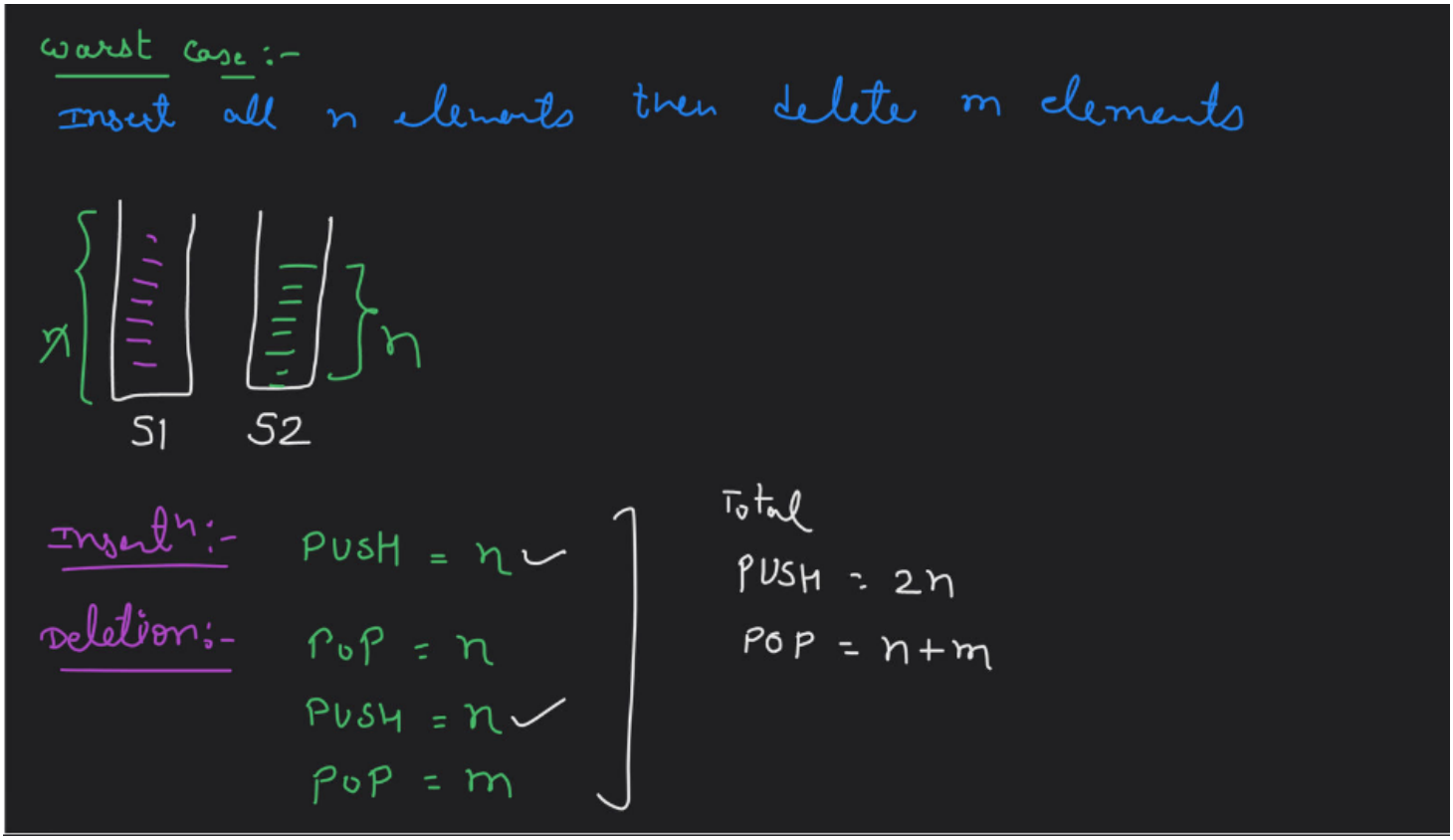
m => insert
n => delete

PUSH => x
POP => y

let $n$ *insert* and $m(\leq n)$ *delete* operations be performed in an arbitrary order on an empty queue $Q$. Let $x$ and $y$ be the number of *push* and *pop* operations performed respectively in the process. Which one of the following is true for all $m$ and $n$?

A. $n+m \leq x < 2n$ and $2m \leq y \leq n+m$
B. $n+m \leq x < 2n$ and $2m \leq y \leq 2n$
C. $2m \leq x < 2n$ and $2m \leq y \leq n+m$
D. $2m \leq x < 2n$ and $2m \leq y \leq 2n$

## Worst case



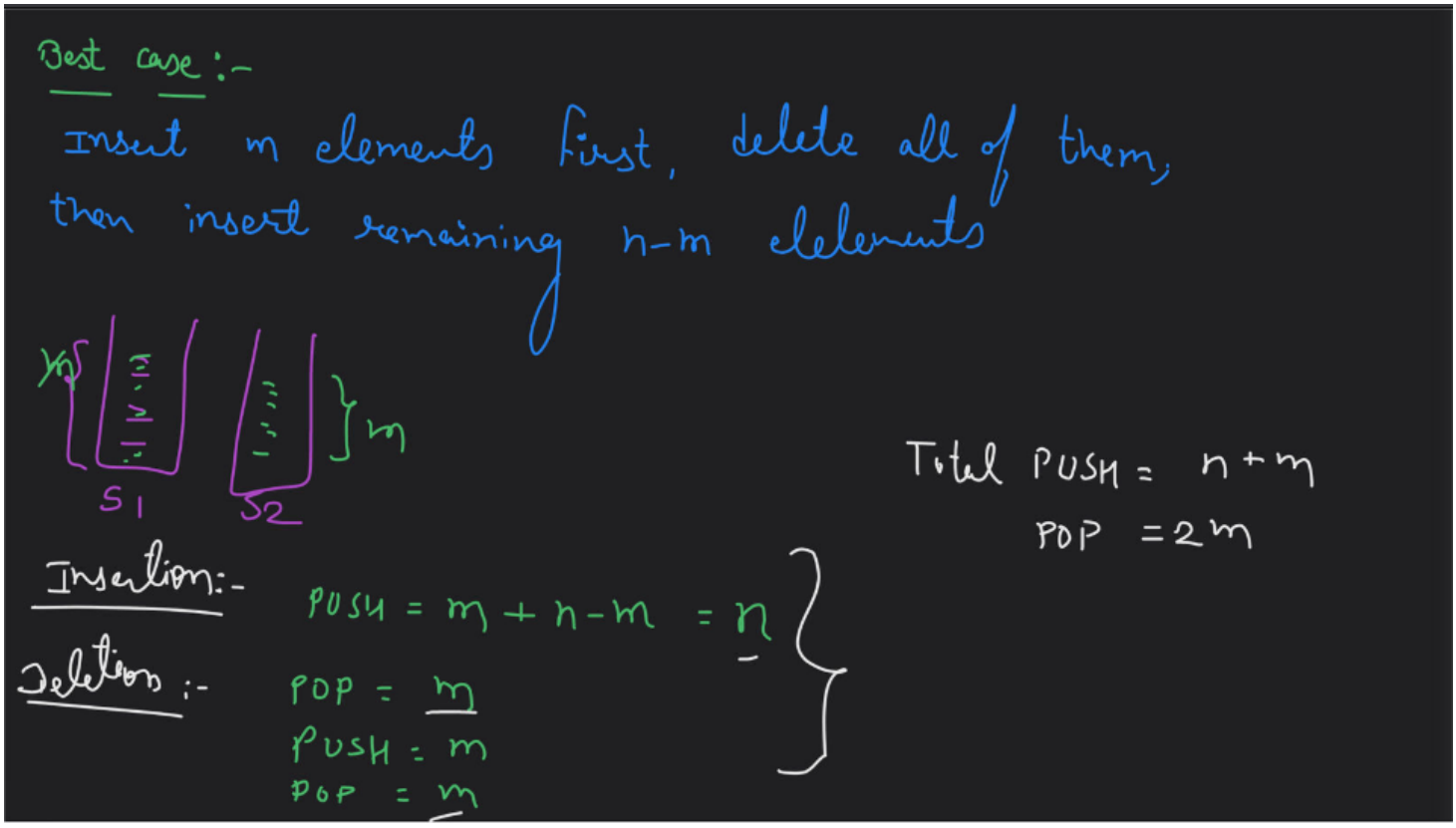

> We are inserting all of the 'n' elements in 's1' stack. After insertion into 's1' we are popping all of the elements from 's1' and then inserting/pushing them into 's2' stack. After that we are deleting only the 'm' elements out of the 'n' elements in 's2'.

- Total insertions -> n + n = 2n
- Total Deletions -> n + m = n+m

> In insertion, we are performing 'n' PUSH operations to insert/push the n elements into s1. After the popping from s1 we are doing n PUSH operations to insert/push the n elements into s2.

> In deletion, we are performing 'n' POP operations to delete/pop the elements from s1. We are performing 'm' POP operations to delete/pop the elements from s2.

## Best case




> We are firstly inserting only the 'm' elements into s1 stack. From s1 we pop the 'm' elements and then we push them into s2 stack. From s2 we pop/delete the 'm' elements. Then we insert the remaining/rest (n-m) elements.

- Total insertions -> m + m + (n-m) = n+m
- Total Deletions -> m+m=2m

> In insertion, we are performing 'm' PUSH operations to insert/push the m elements into s1. After the popping from s1 we are doing m PUSH operations to insert/push the m elements into s2. At the end after all of the deletion/pop in s2, we are inserting/pushing the remaining (n-m) elements into s1 stack.

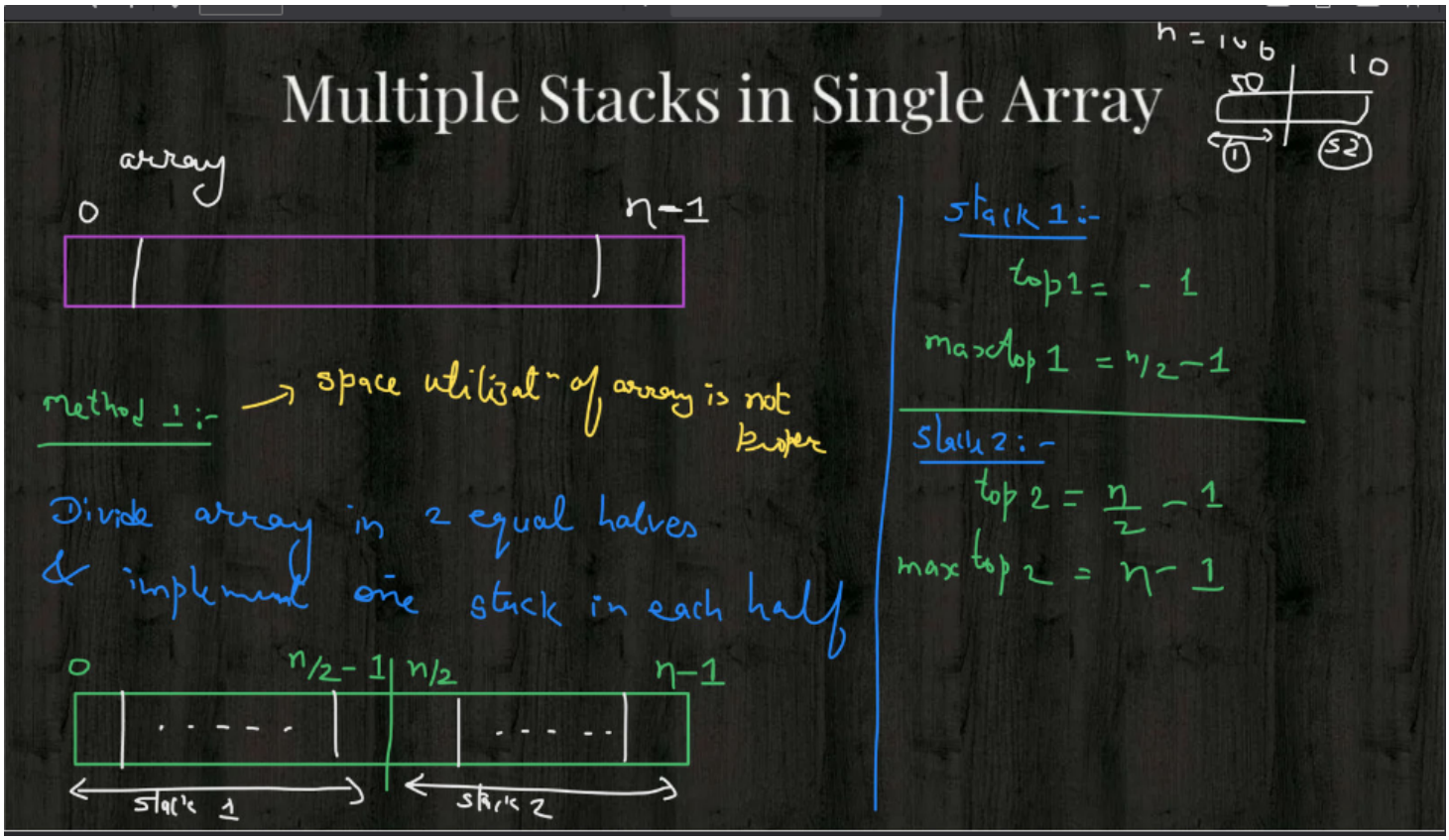
> In deletion, we are performing 'm' POP operations to delete/pop the elements from s1. We are performing 'm' POP operations to delete/pop the elements from s2.

## Conclusion

> We are saving a lot of PUSH() and POP() operations in *Best case* by just taking the 'm' elements instead of the whole 'n' elements. All of the PUSH() and POP() operations of remaining (n-m) elements are saved.

# Multiple stacks in single array.

## Method 1




> Divide the array into 2 equal halves. On one half, implement a stack and on another half, implement another stack.
> The space utilization is not the best as if one of the stacks is full but not the other, then we can't use the empty space in one of the stacks to push/insert more elements into it.

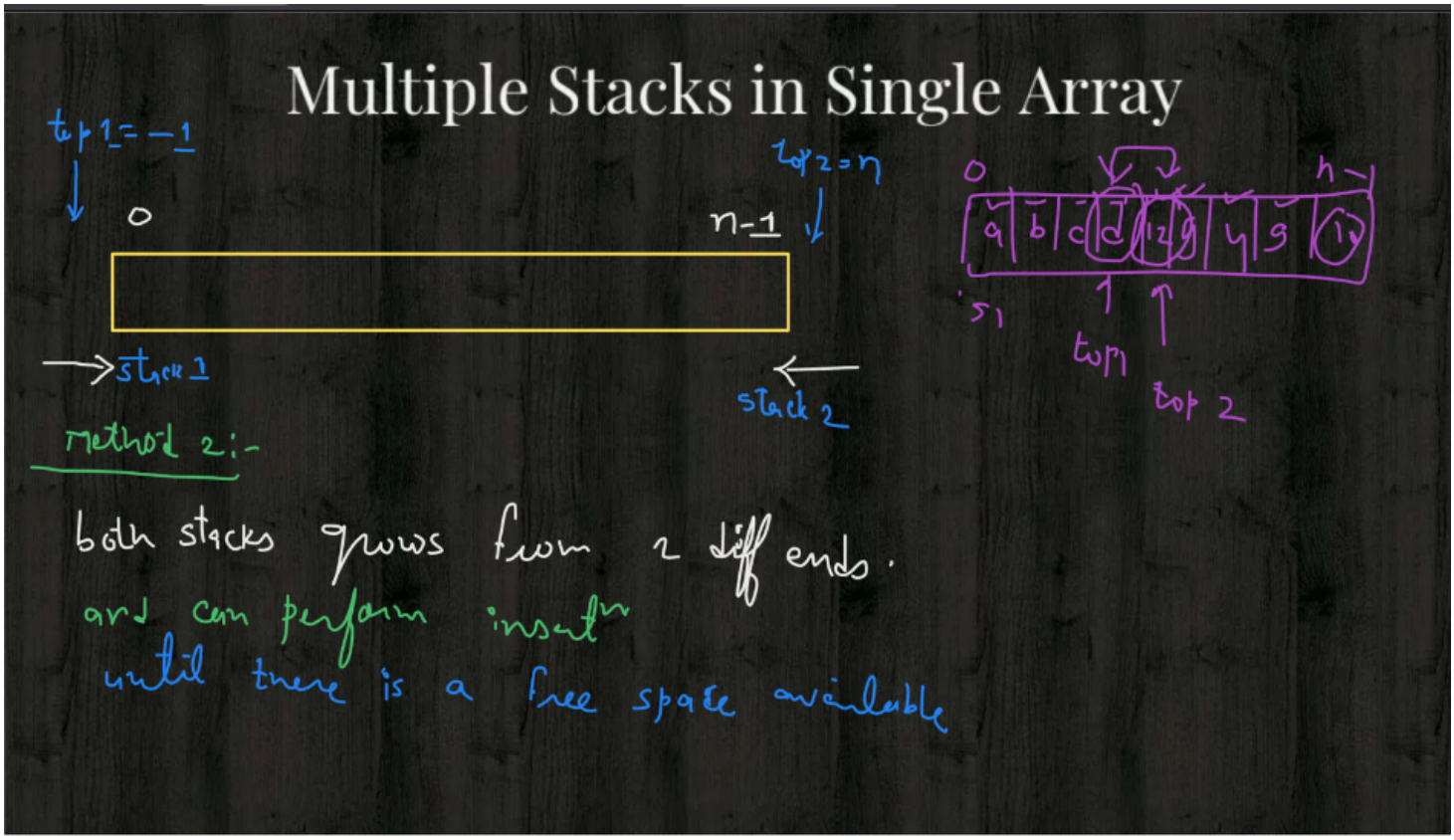
## stack one(1)

- n -> size of the array
- top1= -1 (initial position)
- maxtop1= n/2-1 (max position of stack 1, end of stack 1)

## stack two(2)

- top2= n/2-1 (initial position)
- maxtop2 = n-1 (max position of stack 2, end of stack 2)

## Method 2




> Both of the stacks start from the opposite ends of the arrays or two different ends of the array. Here, we can perform insertion as long as we have empty space or free space to store new elements in the array.
> Space utilization is optimized than *method 1*.

## stack one(1)

- n -> size of the array
- top1= -1 (initial position) // starts from '0' index of array.

> No max or end position.

## stack two(2)

- top2= n (initial position) //starts from 'n-1' index of array.

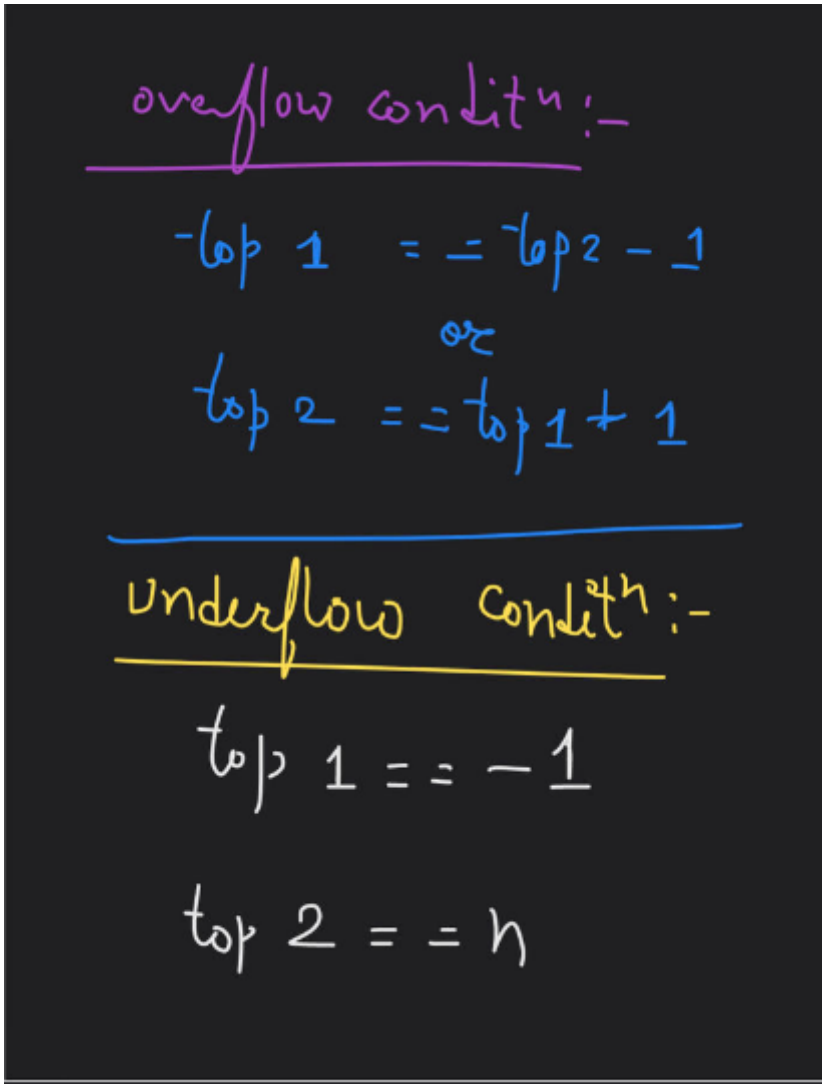
> No max or end position.

## Underflow

- if(top1==-1 && top2==n)

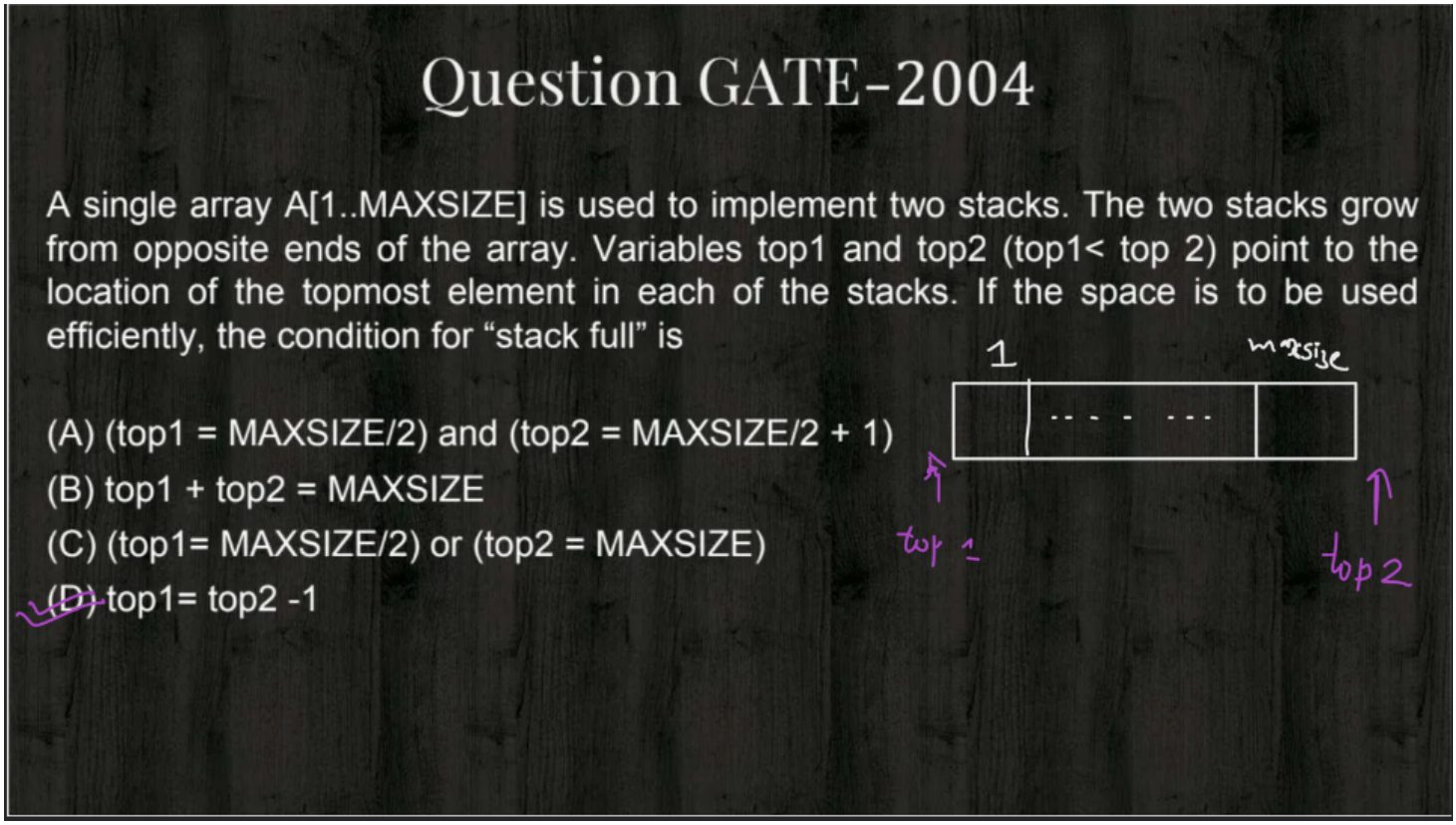
## Overflow

- if(top2 == top1+1)
- OR
- if(top2-1 == top1)

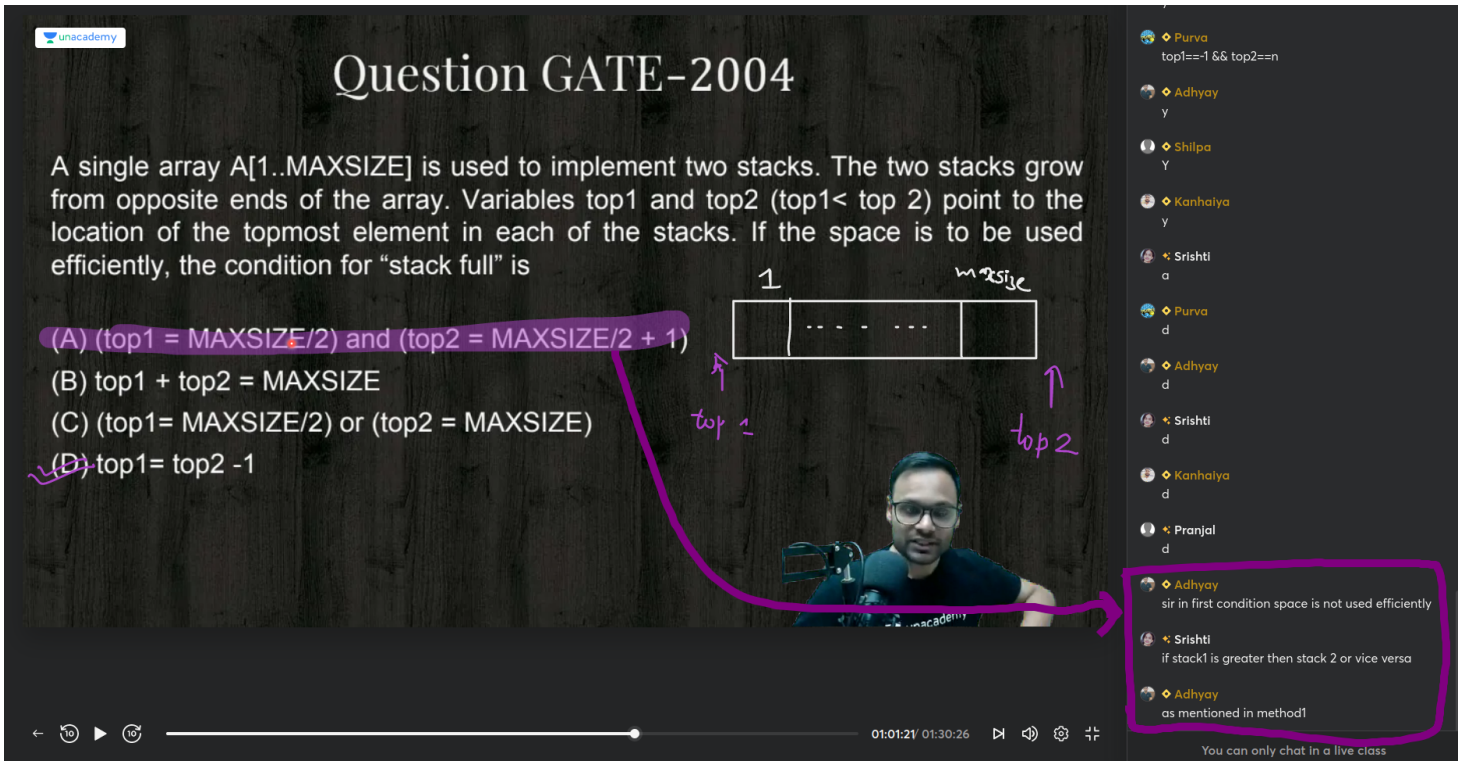


# Question on implementation of two stacks in an array

## Question

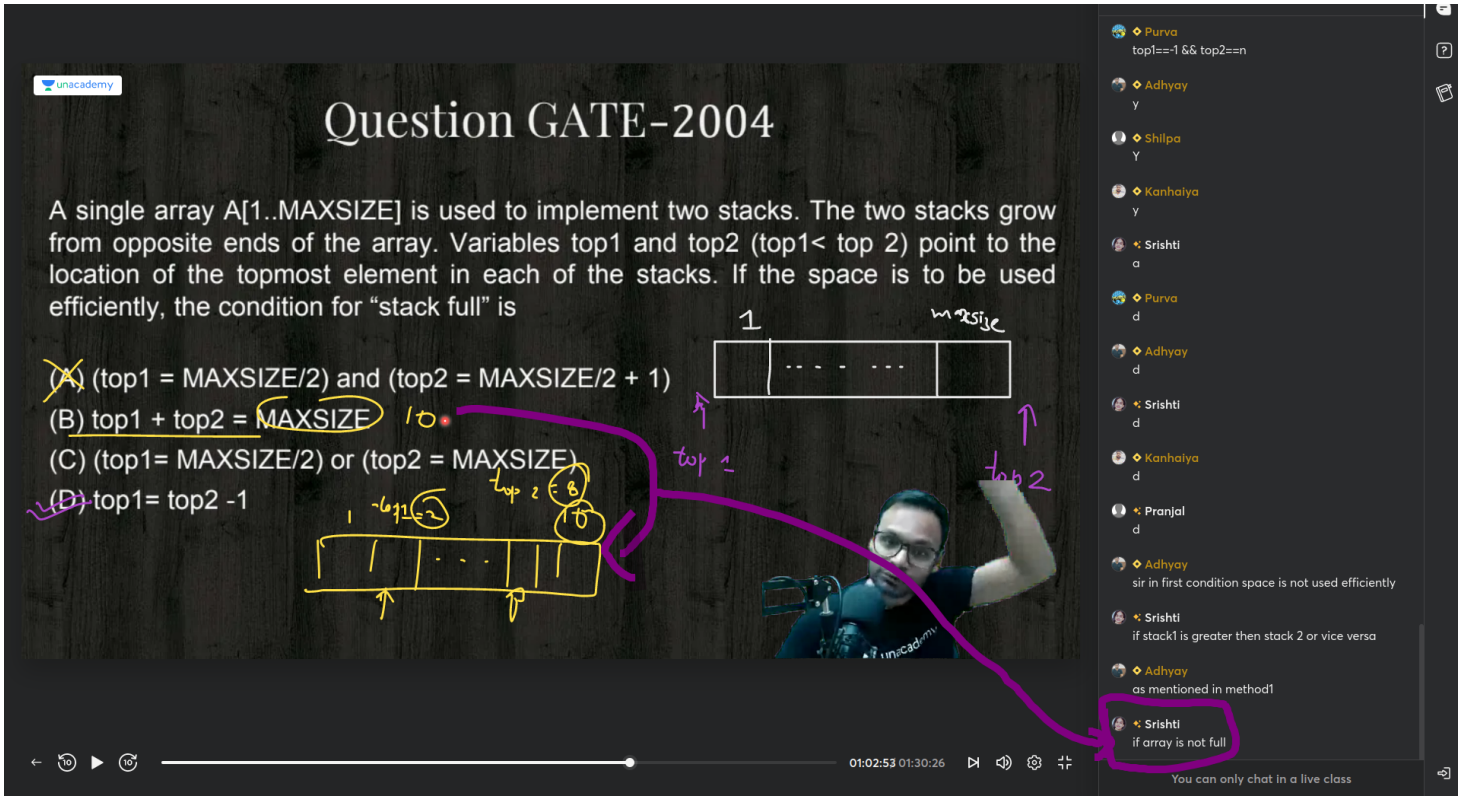



## Option 1



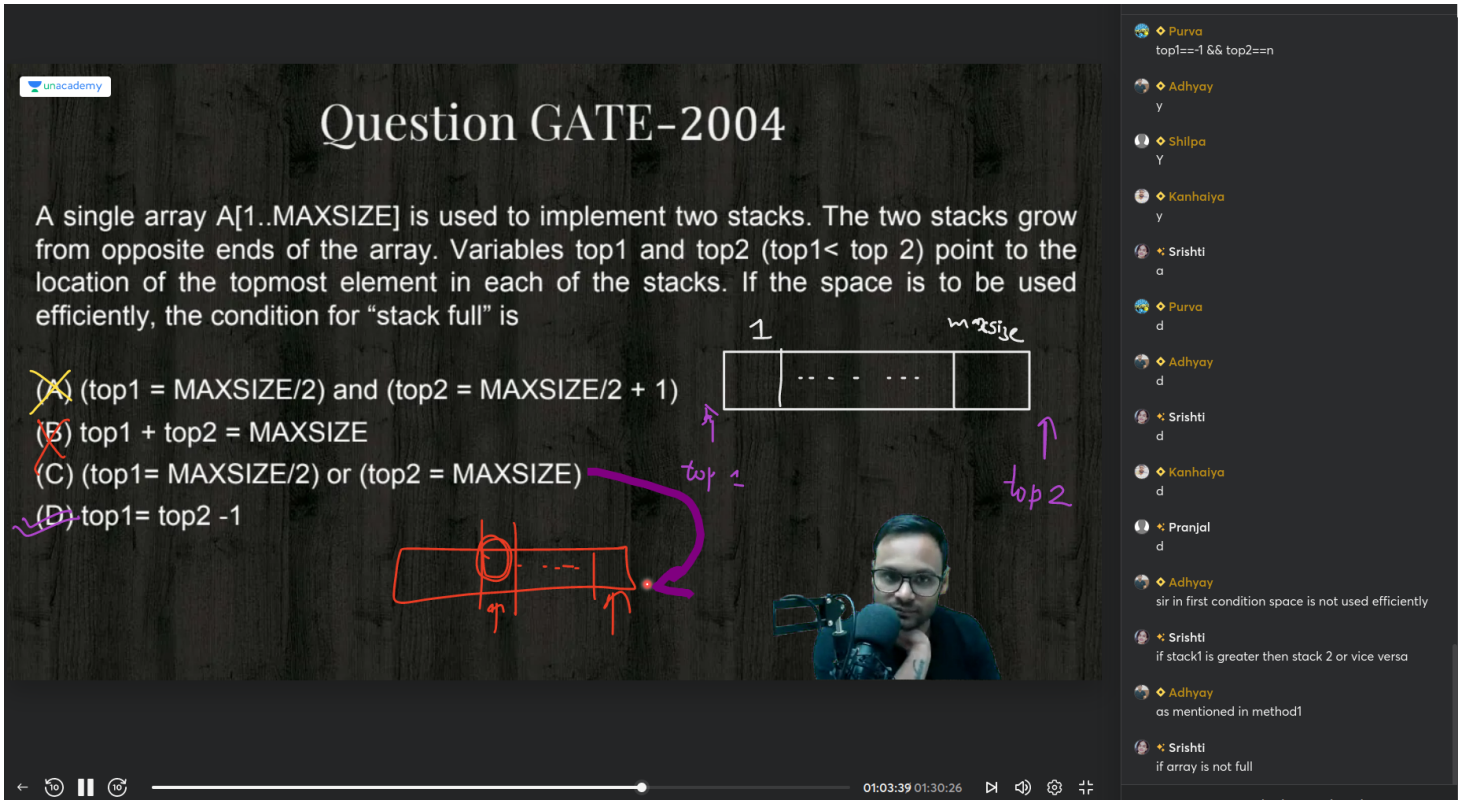
> Here, we are limiting the size of the stacks in the array. Space optimization will be hampered. If the stacks are not distributed equally i.e one of the stacks is larger/smaller thant the other.

## Option 2




> Even if the stacks are not full or have the overflow condition. Then also this option we condition to *True* in that case as well, which is wrong. Even if have free space to store elements, this condition will say *True* for the array full condition.
> **top1+top2=maxsize** -> Even though stack is not full, it will give *True* to the stack full condition.

## Option 3




> If one of the stack is empty and has free space to store more elements, this will return *True* for the array full condition.

# Implement more than 2 stacks in an array

# Scratch Board/Dry runs.

a=1 2 3 4
b=1 2 3 4 2 3 4 3

1 2 3
2 3
3

i=5 4 3 2 1
j=5 4 3 2 1 0 4 3 2 1 0 1

5555
444

33
2

i=20
j=20 21 GATE 20232021

i=1
j=3 0 || 3==4

m=10
i=1 2 4 8 16
count=0 1 2 3 4

m=20
i=1 2 4 8 16 32
count=0 1 2 3 4 5

i=5
j=1 2 4 8
count=0 1 0 -1 -2
n=5

i=1 2 4 8
j=1
count=0 1 2 3 2
n=5

8 1 2

i=0
j=0 1 2 3 4 5
k=1 2 3 4 5
count=0 1 2 3 4 5 6 7 8 9 10 15 20 25 30
n=5

n=5
m=7
k=3
t=10

i=3
j=7

x=12
y=96

a=20

a=1 1
b=511 255
z=1 1

256 511/2 =255

n=5
j=1 2 4 6
count=0 1 2 3 2 1 0
i=5 2 1 0

count= 0 (Ans)

n= 64
i=1 2 4 8 16 32 64
j=1 1 1 1 1 1
count=0 1 2 3 4 5 6 5

64 1 5

n=6
i=1 2 4 8
j=1 1 1
count=0 1 2 3 2

8,1,2

i=1 2 4 8 16 32 64 128 256 512 1024
count=0 1 2 3 4 5 6 7 8 9 10

x=18 12 6
y=24 6

6(Ans)

10, 20 -> 10 [HCF]
1,2

a=20 21 22 23

23(Ans)

x=3

Head(3) -> Head(2)[2] -> Head(1)[1] -> Head(0)[0]
Head(2) -> 1 -> Head(0)[0]
1213121

X(5) -> X(4) + X(2) + 1= 5+1+1= 7
X(4) -> X(3) + X(1) + 1= 3+1+1= 5
X(3) -> X(2) + X(0) + 1 = 1+1+1= 3
X(2) -> 1
X(1) -> 1
X(0) -> 1

fun(4)[4] -> fun(3)[3] -> fun(2)[2] -> fun(1)[1] -> fun(0)[]

1234

fun(95)

fun(95) -> fun(fun(106)) -> fun(96) -> fun(fun(107)) -> fun(97) -> fun(fun(108)) -> fun(98) ->
fun(fun(109)) -> fun(99) -> fun(fun(110)) -> fun(100) -> fun(fun(111)) -> fun(101) -> *91*(Ans)

fun(5)

fun(5) -> (fun(4) + fun(2) + 1) -> fun(4) -> (fun(3)+ fun(1)+ 1) -> fun(3) -> (fun(2) + fun(0) + 1) -> fun(2)
[1] -> fun(1)[1] -> fun(0)[1]

fun(3) -> fun(2) +fun(0)+1 = 1+1+1 =3
fun(4) -> fun(3)+ fun(1)+ 1 = 3+1+1 =5
fun(5) -> fun(4) + fun(2) + 1 = 5+1+1 =*7*(Ans)

fun(fun(5))

fun(6) -> fun(5) + fun(3) + 1 = 7+3+1=11
fun(7) -> fun(6) + fun(4) + 1 = 11+5+1=*17*(Ans)

f(5)

static int r=5
f(5)[18] ->f(3)+2 ->f(3)[16] -> f(2)+r ->f(2)[11] -> f(1)+r -> f(1)[6] -> f(0)+r -> f(0)[1]

f(5) -> *18*(Ans)

fun(5)

x=1
k=1
fun(5) ->

4+2*4*+*2*4+2
4+8+8+2
22

int x=10
int *p=&x
int y

- *Individual cases*

y= *p--; -> 10 [x=9, p=498]
y= --*p; -> 9 [x=9, *p=9, p=1100]
y= (*p)--; -> 10 [x=9, *p=9]
y= --(*p); -> 9 [x=9, *p=9]

A= {1,2,2,3,3,4,4,4,4,4,6,7,8,9,9}
k=6

B= {1,2,2,3,3,3,3,3,4,6,7,8,9,9}
k=3

n=5
C= {3,4,2,1,5}

A[-6:15]
base address -> 1000
W -> 4

A[2] -> 1000 + 4*(2 -(-6)) -> 1000 + 32 -> 1032

1 to n

1 to 10

1,2,3,4,5,6,7,8,9,10
2,4,6,8,10 (n/2)
1,2,3,4,5
2,4
1,2
2 (n/2 -1)

1 to 5
1,2,3,4,5
2,4 (n/2)
1,2
2 (n/2 -1)

n/2 -1

A = {1,4,5,7,9,0}
k=2

A= {7,9,0,1,4,5}

n=64
n + logn -2
64 + 6-2
64+4
68

a -> b -> c -> d -> e -> f
a -> b -> c -> d -> C -> f
start -> a
p -> c
**d**(Ans)

a -> b -> c -> d -> e -> f
a -> b -> e -> c -> d
start -> a
p -> c
**c**(Ans)

a -> b -> c -> d -> e -> f

a -> b -> c -> d -> e -> f -> a

a ->f ->a b ->c ->d ->e ->f

p -> d

**f**(Ans)

a -> b -> c -> d -> e -> f

start -> a

p -> a b c d e

p -> link -> link = c d e f

3,5,5,5,7,8,9,9,9,9,9,12,15,18,19,23

3,5,5,7,8,9,9,9,9,9,12,15,18,19,23

3,5,,7,8,9,9,9,9,9,12,15,18,19,23

head -> 3

p = 5 7

current -> 3 5