Data Structures and Algorithm

Reg # 02

Before Mids

4th Sem.

Rayyan Shabbir

BITF20M535

Multi Dimensional Array

* Multi-dimensional array, does not orignally exist in memory. It is just a perception, for the easiment of user.
* We map multi-dimensional array with the memory of 1-D array.
* A multi-dimensional array is an array having more than one dimensions.
* To refer elements in such array is more, then one subscripts / indices are needed
* The subscripts / indices are based on number of dimensions, e.g. to refer 2D array, two indices are required. Similarly, for 3D array, three indices are required and so on.

2-D ARRAY

* Two dimensional arrays are also termed as **Matrices** in which elements are ordered in number of **rows** and **columns**.

* An example of **m x n** matrix (2D array)

$$\begin{matrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{matrix}$$

* **int A[3][5];**
  A **2-D** Array of **3** rows and **5** columns

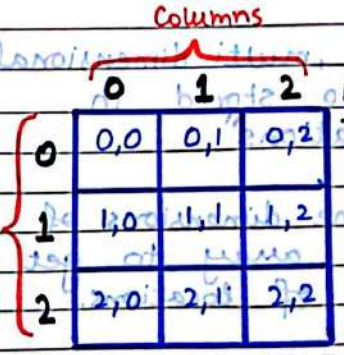
* A **2-D** Array is an array of **1-D** arrays.

* In the above example, each row (0,1,2) has **1-D** array of **5** elements.

* To refer an element of **2D** array, we have to specify the **row number** (first dimension) and **column number** (second dimension).

EXAMPLE OF 2-D ARRAY:

int A[3][3];



→ Logical perspective

It is for the easyment of user. In actual, 2D array does not exist. It only maps with 1-D array.

→ It will first find dimensions.

D_1 (Row) = 3

D_2 (Column) = 3

* Bcz 2D Array orignally does not exist, that why we find dimensions.

then

Size = Row **X** Column

= 3 x 3

Size = 9

Multi Dimensional Array Memory Representation :

* Like 1D arrays, multi dimensional anays are also stored in "Contagious Locations".
* Multiply all the dimensions of multi dimensional array to get required number of locations.
* In array int A[3][5], the total number of locations required are 3 x 5 = 15
* Similarly in a 3D array int[3][5][4] the required number of location will be 3 x 5 x 4 = 60.

2-D Array Memory Representation :

* To map Multi-dimensional array in 1-D array:

→ There are two conventions of storing any two dimensional array in the memory.

(1) **Row-Major Order:** The elements of the array are stored on a row by **row basis.**

(2) **Column-Major Order:** The elements of the array are stored on a column by **column basis.**

Row Major Order

* The elements of the array are stored on a row by row basis, i.e. all the elements of first row, then in the second row, and so on.

	0	1	2
0	(0,0)	(0,1)	(0,2)
1	(1,0)	(1,1)	(1,2)
2	(2,0)	(2,1)	(2,2)

Logical representation

It will first find Table-dimensions:

D_1 (Row) = 3

D_2 (Column) = 3

then,

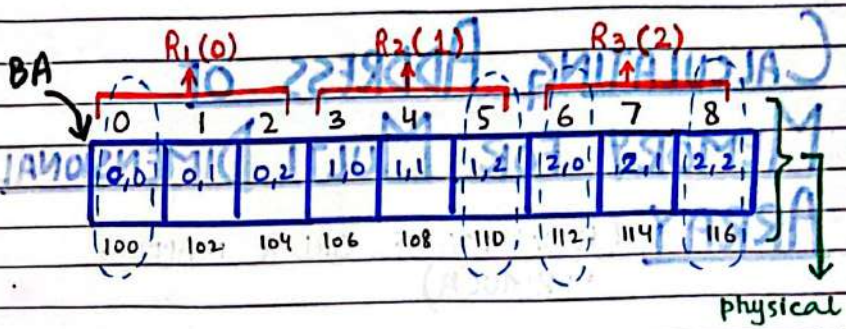
Size = Row × Columns

= 3 × 3 = 9

So,

Upper Bound = 8

Lower Bound = 0



NOTE:

* In row major order, it will place data in physical memory row wise/row by row.

So,

MD → 1D → Array

$$Address(A[i,j]) = BA + w(N(i-l_1) + (j-l_2))$$

CALCULATING ADDRESS OF MEMORY FOR MULTI DIMENSIONAL ARRAY (ROW MAJOR ORDER MAPPING FORMULA)

* We need to map logical indices to physical indices to access a particular element of the array.

* Suppose a 2D array has u_1 rows and u_2 columns, an element in the i^{th} row and j^{th} column can be obtained by the following formula:

GENERAL FORMULA

$D_1 = u_1$

$D_2 = u_2$

$A = A[u_1][u_2]$

$$Address(i,j) = BA + (i u_2 + j) \times ES$$

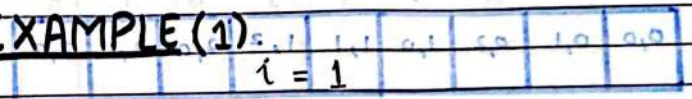
($i u_2$: $A[u_1][u_2]$ no. of col. to skip; $(i u_2 + j)$: index of 1D)

* To go to a specific row (i^{th}) we have to skip that specific number of columns (u_2).

* For this, simply multiply the total number of columns u_2 (second dimension) with i and add the desired column number j.

* This gives us the index of required element exist in the physical 1D array.

EXAMPLE (1)



$i = 1$

$j = 2$

Find address of this 2D in 1D?

$$Address(index(1,2)) = BA + (1 \times 3 + 2) \times ES$$

(index of 1D; no of column which we won't to skip.)

$$Address(index(1,2)) = 100 + (3+2) \times 2$$
$$= 100 + 5 \times 2$$
$$= 100 + 10$$
$$= 110$$

EXAMPLE (2)

$i = 2$

$j = 0$

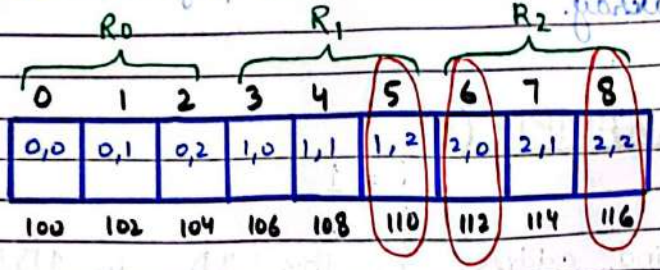
$Address(A[2][0]) = 100 + (2 \times 3 + 0) \times 2 = 112$

EXAMPLE (3)

$i = 2$

$j = 2$

Address $(A[2][2]) = 100 + (2 \times 3 + 2) \times 2 = 116$



CALCULATING ADDRESS OF MEMORY FOR 3-D ARRAY
(ROW MAJOR ORDER MAPPING FORMULA)

$A[u_1][u_2][u_3]$

* To access any arbitrary element i, j, K following mapping can be used

$$A(i,j,K) = BA + (iu_2u_3 + ju_3 + K) \times ES$$

By this, we get mapping of 3D in 1D (Address)

CALCULATING ADDRESS OF MEMORY FOR 4-D ARRAY
(ROW MAJOR ORDER MAPPING FORMULA)

$A[u_1][u_2][u_3][u_4]$

* To access any arbitrary element i, j, K, l following mapping formula can be used:

$$A(i,j,K,l) = BA + (iu_2u_3u_4 + ju_3u_4 + Ku_4 + \dots$$

GENERALIZED FORMULA (ROW MAJOR ORDER MAPPING FORMULA)

$A[u_1][u_2][u_3]\dots[u_{n-1}][u_n]$

★ To access any arbitrary element i, j, k, l, ..., z, following formula can be used:

Address $(A[i][j]\dots[y][z])$

$= BA + (iu_2u_3\dots u_n + ju_3\dots u_n + \dots + yu_n + 1z) \times ES$

Lecture #9

COLUMN MAJOR ORDER

→ The elements of the array are stored on a column by **column basis**.

	0	1	2
0	0,0	0,1	0,2
1	1,0	1,1	1,2
2	2,0	2,1	2,2

→ logical representation

D_1 (Row) = 3

D_2 (Column) = 3

then,

Size = Row × Column

= 3 × 3

= 9

So,

Upper Bound = 8

Lower Bound = 0

BA →

$C_1(0)$			$C_2(1)$			$C_3(2)$		
0	1	2	3	4	5	6	7	8
0,0	1,0	2,0	0,1	1,1	2,1	0,2	1,2	2,2
100	102	104	106	108	110	112	114	116

→ Physical

SPARSE MATRIX

* It is matrix having majority of elements **zero / NULL.**

FOR EXAMPLE:

We have a matrix having size 7 x 8

Row Column

Total Elements = 56 - 9

↳ useful elements (non zero)

0	0	0	0	0	0	0	1
0	0	2	0	0	3	0	0
0	0	0	0	0	0	0	0
0	0	0	4	0	0	0	0
5	0	0	0	0	6	0	0
0	0	0	0	7	0	0	0
0	0	8	0	0	0	9	0

SPARSE MATRIX; STORAGE AND SCANNING

* Matrices are stored using two dimensional arrays.
* Storage of zero / NULL elements is nothing but "wastage of memory."
* It will also increase the scanning time of non-zero elements, because we have to scan the whole **m x n** matrix each time.
* **For Example:**

If we have an integer matrix of **10 x 10** in which only **10** values are **non-zero.**

* So, in total we allocate **10 x 10 x 2 = 200 bytes** of space to store this integer matrix.
* To access these **10 non-zero** elements we have to make **100** scan hence "computationally expensive."

TO MAKE IT SIMPLE:

↳ We only store non-zero values.

* This will save the space as well as the computing time.

* **Data Structure Definition:**

"Best Arrangement" of data in computer's memory."

SPARSE REPRESENTATION TECHNIQUES:

* For this, two sparse Matrix Representation techniques are used:

(1) Triplet Representation (Array/3-Column representation)

(2) Linked List Representation

(1) TRIPLET REPRESENTATION:

* In this technique, a **2D** array is used having **3-columns** and **m+1 rows**, where **m** is total number of non-zero values and an extra row to store information about sparse matrix (Data about Sparse Matrix).

* Each row of **2D** array stores only non-zero values along with their row and column index values.

* The first row of **2D** array stores total number of rows, total number of columns and total number of non-zero values of sparse matrix.

SPARSE TRIPLET REPRESENTATION

★ consider a matrix of 5×5 containing 8 non zero values.

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 3 & 2 & 0 \\ 0 & 5 & 0 & 4 & 0 \\ 0 & 0 & 0 & 9 & 0 \\ 0 & 6 & 0 & 0 & 1 \end{bmatrix}_{5\times5}$$

If traverse this,

5×5 = 25 searches.

So,

→ First, we count no. of non-zero values.

Count of non-zero element = 8

So, we will make 2D Array having 3 columns.

And,

Array for Meta-Data

Rows = count of non-zero element + 1
= 8 + 1 = 9

⇒ A[9][3]

ROW	COLUMN	VALUE
5	5	8
0	1	1
1	2	3
1	3	2
2	1	5
2	3	5
3	3	9
4	1	6
4	4	1

0 3 → Meta da
1 (Total rou
2 total colu
3 total non-z values)
4
5
6 BENEF
7 OF TH
8 For addi as well a substrac we want size, wh we will from this throw.

Representation in Memory

Sparse Matrix → Sparse Representation → 1-I

FOR TRAVERSING IN SPARSE MATRIX REPRESENTATION:

Now,

* As Triplet Representation have 3 columns, so there is no need to use nested loops, for column (second dimension) will always be 0, 1 and 2.

* We will only use one loop which will execute from '0' to m index.

* If we are searching for values (non-zero), then we will not need to go on '0' index. We will start traversing from index '1', because there is no value on '0' index, there is only meta-data.

* Now, by this Processing will be fast.

→ But if we want to copy data from sparse matrix to triplet representation, then we will require nested loops.

NOTE:

* We will not show user the sparse matrix.

* We will only show 3-column (Triplet) representation.

* As we have order (in Meta-data row) of sparse matrix, so we can regenerate the sparse matrix to show it to user whenever needed.

ADDITION OF TWO SPARSE MATRICES TRIPLET REPRESENTATION

A

Row	Col.	Value
5	5	8
0	1	1
2	3	3
3	3	2
2	1	5
2	3	4
3	3	9
4	1	6
4	4	1

B

Row	Col.	Value
5	5	6
0	2	9
0	3	4
1	2	5
2	0	8
3	1	2
3	3	7

C = A + B

CHECK POSIBILITY FOR ADDITION:

★ First, we will check if both sparse matrices are of same size by Meta data (data from row '0').
↳ number of rows must be equal to the number of columns of '0' index.

★ So, here the order of both matrices A and B is same, so the addition can be performed.

NOTE:

→ The size of resultant matrix A is unknown therefore we can take maximum size i-e. the total count of values of matrix A and B which is 5 + 6 = 11 + 1 for first row.

C

So, if true:

A

	ROW	COL.	VALUE
i→	5	50	8
i→	0	x	x
i→	1	2	3
i→	1	3	27
i→	2	1	5
i→	2	3	4
i→	3	3	9
i→	4	1	6
i→	4	4	1

B

	ROW	COL.	VALUE
j→	5	5	6
j→	0	00	9
j→	0	1	4
j→	1	24	5
j→	2	0	8
j→	3	1	2
j→	3	3	7

Resultant matrix, 'C' will create:

C

	ROW	COL.	VALUE
	5	5	11
K→	0	0	9
K→	0	1	5
K→	1	2	8
K→	1	3	2
K→	2	0	8
K→	2	1	5
K→	2	3	4
K→	3	1	2
K→	3	3	16
K→	4	1	6
K→	4	4	1
K→			
K→			
K→			

→ We will increment this count when element adds.

* We will keep adding elements in this matrix until the data of either A or B matrices finish.
* When data of one matrix finish, we copy data of other matrix.

→ here, memory is also wasting, but wastage is less.
→ To reduce wastage, we will copy the data in a new array whose size equal to non-zero

NOTE:

* Now, by this memory will not waste.
* In sparse matrix (3-column / Triplet) representation, the time complexity is O(1).

TRANSPOSE OF SPARSE MATRIX

→ Only replace columns by rows.

→ If after doing triplet represent we want take transpose of sparse matrix, then first we will create a new matrix and then will equal the number of columns of sparse matrix to the number of rows of new matrix, and vice versa.

BENEFIT OF THIS TRIPLET REPRESENTATION:

→ Lower Triangle
→ Diagonal
→ Upper Triangle

→ For all these type of matrices, we use sparse matrix.

→ Storage Operation on these type of matrices become easy.

Lecture #10

BEST, WORST, AVERAGE CASE ALGORITHM ANALYSIS

* The running time of an algorithm increases as the input size n increases or remains constant in case of constant running time.
* In that case, we perform Best, Average and Worst case analysis of an algorithm.
* These measures are performed asymptotically.
* For every algorithm, there are **THREE TIMES**:

1) BEST CASE:

* How much **minimum** time an algorithm takes to execute
* **Minimum** number of steps taken by an algorithm on any instance of size **n**.
* The best case is usually not estimated because it does not give us better measure about the performance of an algorithm.

(2) WORST CASE:

* How much **maximum** time an algorithm takes to execute.
* **Maximum** number of steps taken by an algorithm on any instances of size **n**.
* This gives us the better measure about the performance of an algorithm.
* We mostly interested in finding the worst case of an algorithm.
* The worst case is mostly performed in algorithm analysis.

(3) AVERAGE CASE:

* Average case of an algorithm is the function defined by the **average number** of steps taken on any instance of size n.

* We sum up all the possible cases time and divide with the number of cases.

Average Case = $\frac{\text{All possible cases Time}}{\text{Number of cases}}$

* The average case is also usually performed in algorithm analysis.

* Most of the time the average case is roughly equal to the worst case. (similar)

* Average case analysis may not be possible always.

* It is complex to determine average case, because we have to identify each case individually.

* It is difficult to find Average case of an algorithm.

➡ BEST, WORST AND AVERAGE CASE ANALYSIS WITH THE HELP OF SOME SORTING ALGORITHM:

* Suppose an algorithm sorts a list of n elements in **ascending** order.

* **BEST CASE:**
  When the input list is already sorted.

* **WORST CASE:**
  When the input list is reverse sorted.

* **AVERAGE CASE:**
  When the input list is partially sorted. (have to consider all the cases).

Asymptotic Notations for Best, Worst and Average Cases:

Note:

* Any asymptotic notation **Big-OH (O)**, **Big-Omega (Ω)** or **Big Theeta (θ)** can be used to represent best, worst and average cases of an algorithm.

* **For example,** if an algorithm performs $n+1$ steps in best case, we can represent it as $O(n)$, $\Omega(n)$ or $\theta(n)$.

* **For example,** if an algorithm performs $3n^2+5n+1$ steps in worst case, we can represent it as $O(n^2)$, $\Omega(n^2)$ or $\theta(n^2)$

* In the same way, average case can be represented with any of the asymptotic notations.

Searching Algorithms

(1) Linear Search:

* Linear search is very simple search algorithm.
* It sequentially checks each element of the list until a match found or the whole list has been searched.
* A value (key) is given, we will check/compare to find it in the array.
  - if value found → return 'index'
  - if value not found → return '-1'

PROGRAM (OF LINEAR SEARCH)

return index ↑ | list ↑ | size ↑ | search ↑

```
int LinearSearch(int a[], int n, int key)
{
    for(int i=0; i<n; i++){          // we use for loop because no of iterations are known.
        if(a[i]==key)
            return i;
    }
    return -1;  // if element does not exist.
}
```

	B.C	W.C
for(int i=0; i<n; i++){	1	n+1
if(a[i]==key)	1	n
return i;	1	1
	T(n) = 3	T(n) = 2n+2

i.e: i = 0 ... n-1

→ Now, we will find the Best, Worst and Average case of Linear Search:

BEST CASE:

* Best case of linear search exist when the desired **key** to be searched is present at the first index of array

* If value exists at index = 0

0	1	2	3	4	5	6
8	5	11	43	29	83	4

* We measure best case mostly (not) in **big-O**. (but it is our choice; we can also use Big-Omega and Big-Theeta)

$T(n) = 3$ → It is efficient algorithm as it takes constant time.

$O(1)$ ↔ bcz we only compare at '1' time.

* We can also use $O(1)$, $\Omega(1)$ or $\Theta(1)$ for Best case.

Worst Case:

* Worst case of linear search exist when the desired **key** to be searched is present at the last index of array (if value exist at end index)
* Worst case of linear search exist when the value does not exist.

$T(n) = 2n + 2$

$O(n)$ ⟷ Linear growth; it is not a good growth.

* It means linear search has $O(n)$ worst case time complexity.

Average Case:

* Average case does not mean picking up middle value and taking its time complexity.
* We individually check time complexity of every case one by one in average case.
* Identify all the case times

no. of comparisons

- When the key is present at first '1' index.
  $T(1) = 1$
- When the key is present at second '2' index.
  $T(2) = 2$
- When the key is present at third '3' index.
  $T(3) = 3$
- When the key is present at four '4' index.
  $T(4) = 4$

⋮

- When the key is present at Last 'n' index.
  $T(n) = n$

SORTING

* Sorting refers to ordering elements of data in increasing or decreasing fashion, according to some relationship among the data items.
* Sorting greatly improves the efficiency of searching.
* Sorting of large quantities of data is really desireable to improve efficiency.

SORTING ALGORITHMS

1) Bubble Sort
2) Selection Sort
3) Insertion Sort
4) Merge Sort
5) Quick Sort
6) Heap Sort
7) Counting Sort
8) Bucket Sort
9) Radix Sort
10) Shell Sort
11) Bitonic Sort
12) Cocktail Sort

① BUBBLE SORT:

* Bubble Sort is a simple sorting algorithm.
* It picks '2' (two) adjacent elements and compare them
  - If they are not in the intended order then it "swaps their position".
* Then next comparision.
  - In the end (completion of inner loop '1' time), one (1) element will be placed to its correct position.
* It repeats the above process untill the last element.

EXAMPLE: (SORT DATA IN ASCENDING ORDER) → 5, 4, 3, 2, 1

0	1	2	3	4	
(5)	(4)	3	2	1	
4	(5)	(3)	2	1	
4	3	(5)	(2)	1	
4	3	2	(5)	(1)	
4	3	2	1	5	

- After one iteration an element (1) reached to its position.
- Here we do 'n-1' comparisons.

↳ Repeat the above process for remaining 'n-1' elements.

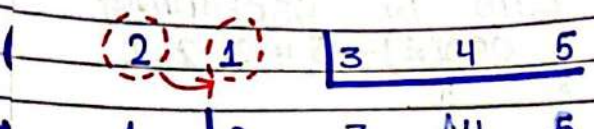
(4)	(3)	2	1	5
3	(4)	(2)	1	5
3	2	(4)	(1)	5
3	2	1	4	5

- '2' elements placed to their position after iterations

↳ Repeating.

(3)	(2)	1	4	5
2	(3)	(1)	4	5
2	1	3	4	5

- '3' elements sorted (placed at their right position.



• 4 elements sorted.

• As we do n-1 comparisions, which means last element will be automatically sorted.

1 2 3 4 5 ⇒ * means all elements sorted.

⇒ CODE (ALGORITHM) OF BUBBLE SORT:

```
void bubbleSort(int A[], int n)
{
  for(int i=0; i<(n-1); i++)
  {
    for(int j=0; j<(n-i-1); j++)
    {
      if(A[j] > A[j+1])
        swap(A[j], A[j+1]);
    }
  }
}
```

- We do 'n-1' repetations bcz last element will automatically sorted.
- keep count of sorted elements. (How much element sorted)
- Bcz we don't to compare to element (already sorted elements) So, run the inner loop till 'n-i-1'
- If we do 'n-1' it not make = differ
- compares the adjacent element.
- It sorts data in ascending order.
- If we want to sort data in descending order A[j] < A[j+1]
- swaps the elements

BUBBLE SORT TIME COMPLEXITY:

↳ The key comparision (if (A[j] > A[j+1])) statement is always going to execute the

- Data is "already" Sorted → BEST CASE
- Data is "partially" Sorted → AVERAGE CASE
- Data is "Reverse"-i-Sorted → WORST CASE

* We only see "if (A[j] > A[j+1])" statement.

$$= 1 + 2 + 3 + 4 + \dots + n$$

$$= \frac{n(n+1)}{2}$$

- **TIME COMPLEXITY:**

→ WORST CASE : $O(n^2)$
→ BEST CASE : $O(n^2)$
→ AVERAGE CASE : $O(n^2)$

} → bcz condition will always check either it is best, worst or average case.

- **SPACE COMPLEXITY:** $O(1)$

* It does not use any extra space to complete its sorting process.

* OPTIMIZATION OF BUBBLE SORT:

→ The bubble Sort can be optimized by inbroduction a **'flag'** variable to monitor whether elements are getting swapped inside the inner loop or not?

→ If **'no swap'** occured, it means the array is already sorted and we can simply exit the program.

→ By this optimization the best case time complexity of bubble Sort becomes **O(n)**, because the outer loop will executes itself only once (constant time)

→ BUBBLE SORT OPTIMIZED CODE:

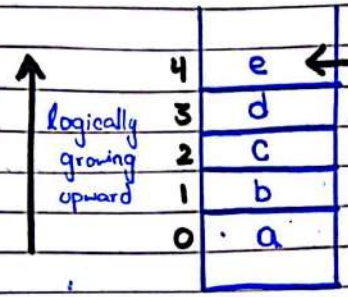
```
void bubbleSort (int A[], int n) {

    bool flag = false; // To monitor swap of ele
    for(int i=0; i<n-1; i++) {
        for(int j=0; j<n-i-1; j++) {
            if (A[j] > A[j+1]) {
                swap(A[j], A[j+1]);
                flag = true; // If swap occu
            }
        }
        if (!flag) // If no swap occured, array
            break; // already sorted; exit the
                   outer loop.
    }
}
```

Lecture #13

STACK

* Stack is a data structure.
* In stack operations are performed in "LAST IN FIRST OUT (LIFO)" OR "FIRST IN - LAST OUT (FILO)"
* The elements in the stack are added and removed from same end.

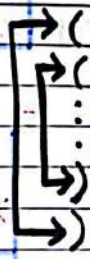
EXAMPLE:



* It's like a Bucket.
* Last thing is place at top/first.
* It means system is processing the top element.

APPLICATIONS/USES OF STACK:

* Back Tracking.
* To reverse a string.
* "UNDO" mechanism in text editors.
* File/Folder explorer having back and forward functionality.
* Internet browsers with back and for
* Stack also use to check "plandrome" behaviour of string:
  - It simply place string one by one in stack and then extract each character of str from stack in reverse order.
* Also use for syntax checking by compiler. example: ((...))



* Thus, stack is used for syntax analyzer, error checker, reversing the string, converti decimal to binary.

NOTE:

* No compiler can be made without stack.

SYSTEM STACK

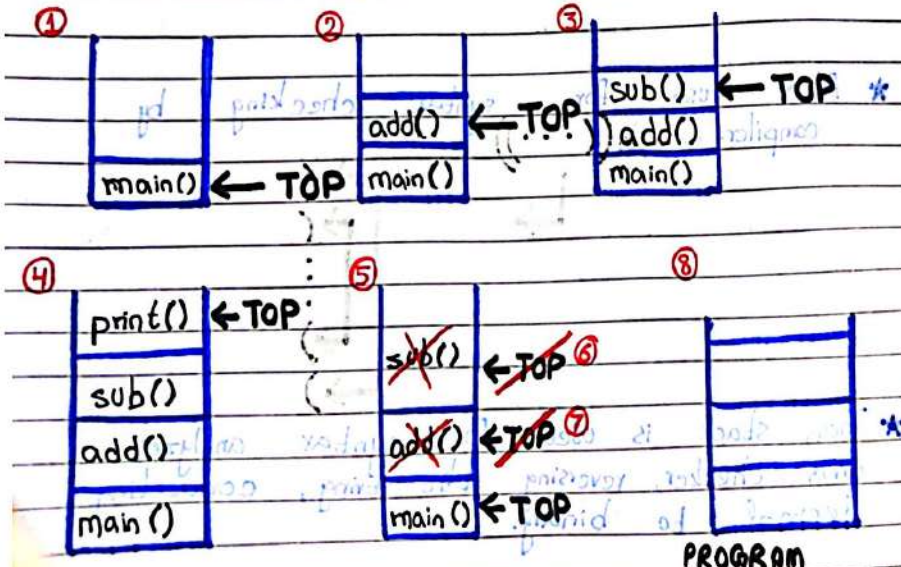
* The system stack is used to "keep" track of the function calls.

Example:- wait → wait → wait

```
main(){        add(){        sub(){        print(){
  add();         sub();        print();      stmt;
}
```

We want "main()" to not exit.

↳ For this purpose there is a system stack in backend.

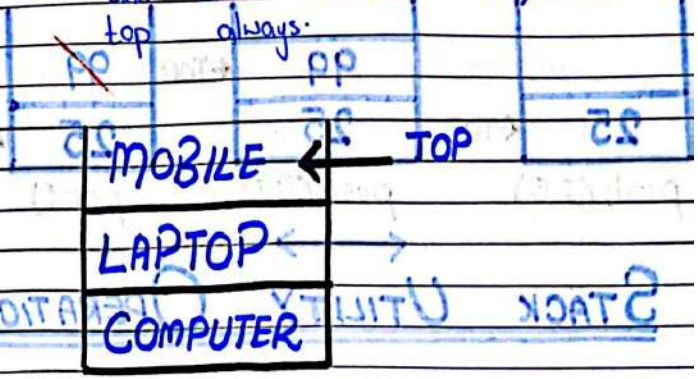


* Thus, by this our "main()" exits in last in our program.

↳ STACK ENDS:

* STack has only "ONE END".

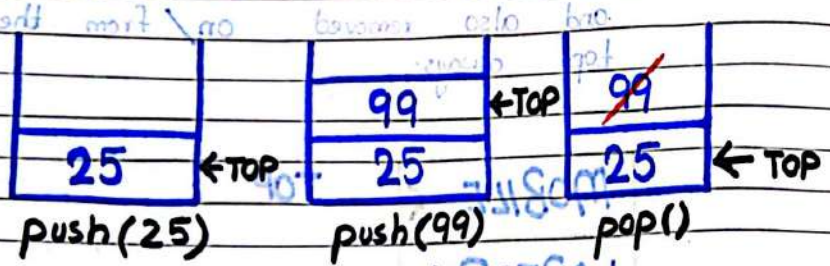
→ **TOP:** It holds the top of the stack. The items are inserted and also removed on/from the top always.



★ STACK OPERATIONS

↳ Two operations are performed with stack.

- **PUSH**: This operation is used to insert items on top.
- **POP**: This operation is used to remove items from top.



★ STACK UTILITY OPERATIONS

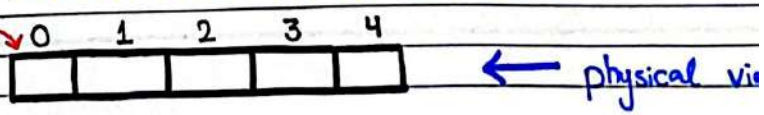
isEmpty: To verify either the stack is empty or not? This operation is usually performed before/prior to "POP".

isFULL: To verify either the stack is full or not? This operation is usually performed before/prior to "PUSH".

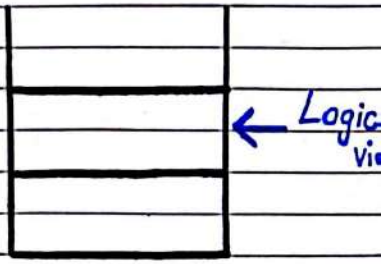
IMPLEMENTATION OF STACK:

- Stack can be implemented using two ways:
  - (1) Array Base Stack.
  - (2) Linked List.

① ARRAY BASE STACK:



FOR STACK:



- We need 'n' max size of stack.
- We need a variable (pointer) which points to "TOP" (i.e most recent (TOP) element.)
  - ★ "Top" will always at 'n-1' index.
  - ★ At the beginning "TOP" would have '-1' to indicate an empty stack.

NOTE:

↳ The **TOP** incremented each time a new item is **PUSHED** (inserted).

* **TOP = TOP + 1**

↳ The **TOP** decremented each time an item is **POPPED** (removed).

* **TOP = TOP - 1**

STACK IMPLEMENTATION PERSPECTIVE: (CODE)

```
const int n = 5;
int arr[5];
int top;
top = -1;
```

• At start we place "TOP" '-1' bc: at start array (stack) is empty. As, in C++ indices start from

* **BEFORE REMOVING (EXCTRACTING) AN ELEMENT WE CHECK:**

• **ISEMPTY()**

```
bool isEmpty() {
    return top == -1;
}
```

→ return "TRUE" if s empty.

* **BEFORE INSERTING AN ELEMENT WE C**

• **ISFULL()**

```
bool isFull() {
    return top == n-1;
}
```

→ return "TRUE" stack is

↳ i.e (max - 1)

NOTE:

↳ The **TOP** incremented each time a new item is **PUSHED** (inserted).

* **TOP = TOP + 1**

↳ The **TOP** decremented each time an item is **POPPED** (removed).

* **TOP = TOP - 1**

STACK IMPLEMENTATION PERSPECTIVE: (CODE)

```
Const int n = 5;
int arr[5];
int top;
top = -1;
```

• At start we place "TOP" '-1' baz at start array (stack) is empty. As, in C++ indices start from '0'

* **BEFORE REMOVING (EXCTRACTING) AN ELEMENT WE CHECK:**

• **ISEMPTY()**

```
bool isEmpty() {
    return top == -1;
}
```

→ return "TRUE" if stack empty.

* **BEFORE INSERTING AN ELEMENT WE CHECK**

• **ISFULL()**

```
bool isFull() {
    return top == n-1;
}
```

→ return "TRUE" if stack is Full

↳ i.e (max - 1)

NOTE:

- IF STACK IS EMPTY → STACK UNDERFLOW
- IF STACK IS FULL → STACK OVERFLOW

★ To inserts an element in stack we use operation: → "PUSH"

★ To remove / extract an element from stack we use operation: → "POP"

PUSH OPERATION::

- It inserts an element in stack.

*To call push operation from main():

```
s.push (value);
```

obj ↑ — We passed value as parameter bcz we want to add this element in stack

```
void push (int value)
{
    if(top != n-1)
    {
```

- Check if stack is FULL?
- **OPTIMIZATION:** As, it have only one check. So, we simply check it h

→ "BEST ONE"

OR

```
if( ! isFull() )
{
```

- We can also use function call here but as function call takes time. So, we avoid it.

```
    a[++top] = value;
```

- First incremented top a then place value at loca (next to previous top

```
}
else{
    cerr << "Stack Overflow";
}
}
```

- It is a stream which deals wit "ERROR".
- It display "ERROR" message.

POP OPERATION ::

- It remove an element from stack.
- We can also return the last/top element which will be going to remove by making its return type int.

```
void pop ()        // No parameter is passed.
{
  * First check if stack not underflow.

  if (top == -1)
      cerr << "Stack Underflow";
  else
      top --;      // Only deleted the last/top element.
}
```

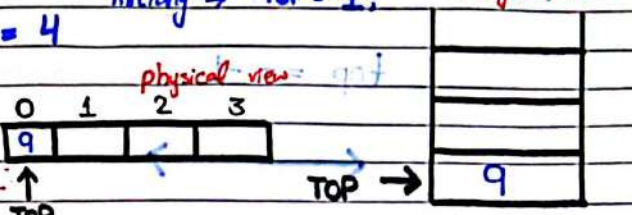
NOTE: "Cerr"

* "Cerr" is a stream which deals with errors. (It display error message).
* "cerr" does not Buffer (temporary store) data before displaying.
* We cannot read "cerr" from memory bcz it does not store in memory.
* "cout" does Buffer (temporary store) before displaying it.

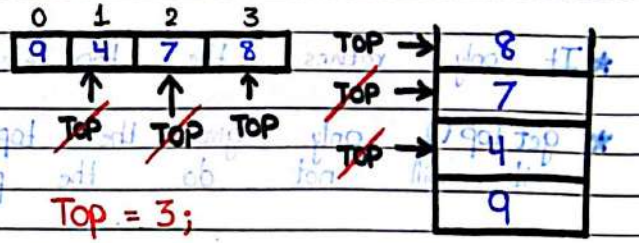
STACK EXAMPLE :

initially → TOP = -1;

maxSize = 4



* push(9)

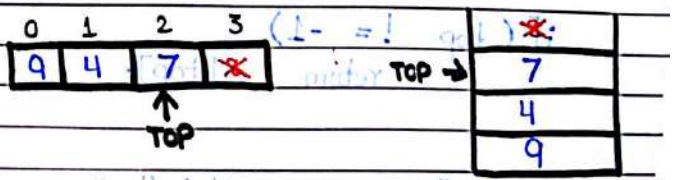


* push(4)
* push(7)
* push(8)

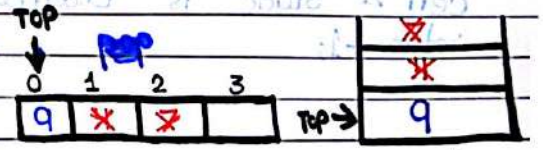
Top = 3;

* push(7)
  - STACK OVERFLOW (Stack is Full)

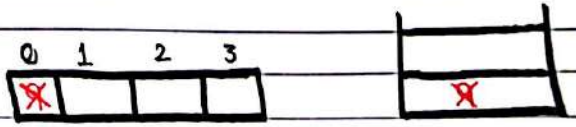
top == size - 1;



* pop()



* pop(), pop()



* pop()

* pop()
  • STACK UNDERFLOW (Stack is Empty)

top == -1

TO GET ELEMENT OF STACK

* It only returns the top element.
* getTop() it will only not give the top value do the pop.

```
int getTop()
{
```

* First check: before getting "Top" if stack not empty.

```
if ( top != -1)
    return a[top];

cerr << "Stack is Underflow";
return -1;
```

SHOW STRUCTURE (To print all data of Stack)

* It only displays data of stack. (not increment or decrement "top").

TO UPDATE

* We pass key value or index value to check where we want to do updation.

STACK TIME COMPLEXITY:

Complexity of:

* isEmpty → $\Theta(1)$
* isFull → $\Theta(1)$
* PUSH → $\Theta(1)$
* POP → $\Theta(1)$
* getTop → $\Theta(1)$

* Thus, stack is an efficient "data structure."

Lecture # 14

QUEUE

* Queue is a data structure.
* In queue, operations are performed in:
  - "FIRST IN FIRST OUT (FIFO)"
  - "FIRST COME FIRST SERVED (FCFS)"
* In Operating System, queue is mostly used.
* The elements in the queue are added from one end and removed from other end.

APPLICATIONS:

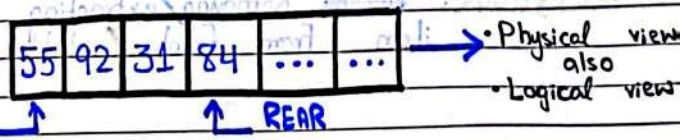
* Processing of customer requests in bank.
* Waiting on hold at call centre for tech sprt.
* Processing schedulling in Operating System.
* Serving requests on a single shared resource,
  - Like printer to print document.
  - shopping centre counters.

NOTE:

* Queus are also priotirized (you set priority and then work according to it).

QUEUE ENDS:

* Queue has two ends.
  - **REAR:** Items are inserted at rear.
  - **FRONT:** Items are removed from front. (where elements served).



QUEUE UTILITY OPERATIONS:

* **isEmpty:**
  To verify either the queue is empty or not?
  This operation is usually performed before/prior to "dequeue."

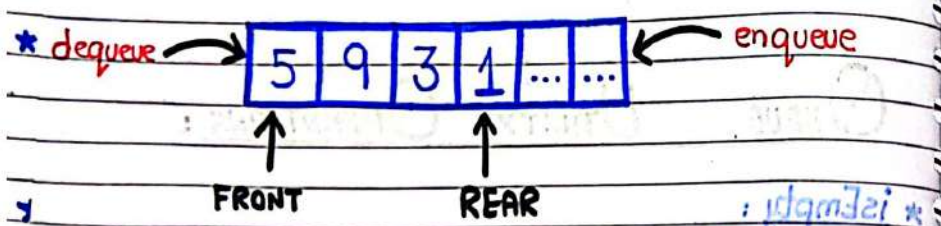
* **isFull:**
  To verify either the queue is full or not?
  This operation is usually performed before/prior to "enqueue."

Queue Operations:

Two operations are performed with queus:

* **enQueue:** Adding/inserting an item at rear. (relate with : rear)

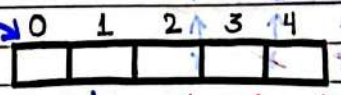
* **deQueue:** ~~Adding~~ Removing/extracting an item from front. (relate with front)



Queue Implementation Perspective:

* Queues can be implemented using arrays.

```
Const int n=5;   } max size of queue.
int a[n];
```



↳ physical also logical view

* The **rear** and **front** each holds the index of array 'a' for **enqueue** and **dequeue** respectively.

* Initially, both **front** and **rear** are '-1' to indicate an empty queue.

```
int front = -1;
int rear = -1;
```

LOGICALLY WRONG CONCEPT:

* The "rear" incremented each time a new item is "enqueued" (inserted).

```
rear = rear + 1
```

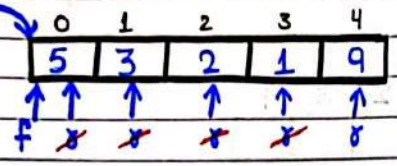
* The "front" incremented each time an item is dequeued (removed).

```
front = front + 1
```

EXAMPLE:

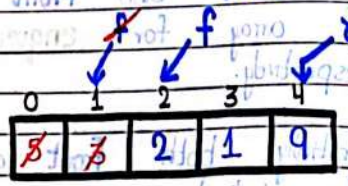
* For 'FULL' if we do → r = size - 1;

To enQueue: 5, 3, 2, 1, 9

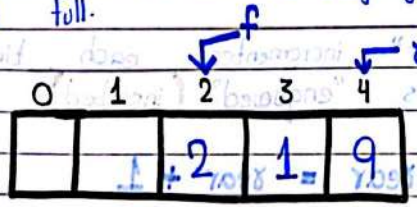


If we deQueue after filling the queue

- deQueue()
- deQueue()



* As, we did r = size - 1;
Now, if we do
enqueue(8)
then it would give an error bcz it will say queue is "FULL" but originally queue is not full.



- The rear reached to "last index".
Is the queue becomes full?

→ No, it is not.

CIRCULAR QUEUE:

* To deal with the previous problem in which locations are still vacant but the "rear" reached to "last index".

* We implement queue in "CIRCULAR FASHION."
Queues are normally dealt in circular fashion.

* The "Last Position" is (logically) connected back to the "First Position" to make a circle.

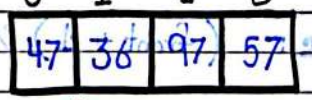
TO GO BACK (ROTATE) WE HAVE A OPERATOR

* MOD %: The remainder operator (%) can help us to rotate among the indices of array.
property of %:- It gives remainder by remaining in the range of denominator.

* If we take remainder of elements with number 'n' the result will always be amongst one of 0, 1, 2, 3, ... n-1.

LIKE: %5 result → (0-4)

- 19 % 5 = 4
- 37 % 5 = 2
- 100 % 5 = 0
- 923 % 5 = 3

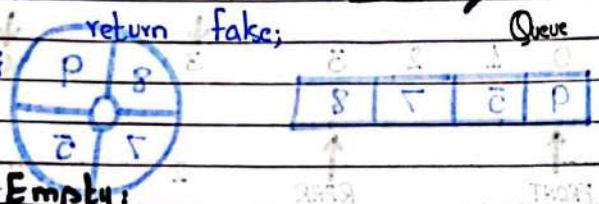


CODE IMPLEMENTATION OF CIRCULAR QUEUE :

- **isFULL:**

```
bool isFull ()
{
    if((rear +1) % n == front)
        return true;
    return fake;
}
```

- It return "TRUE" if Queue is Full.



- **isEmpty:**

```
bool isEmpty ()
{
    return ( rear == -1 && front == -1);
```

OR

```
    if ( rear == -1 && front == -1)
        return true;
    return false;
}
```

- It return "TRUE" if Queue is Empty.

- **enQueue:**

```
void enQueue (int value) {
    if ((rear + 1) % n != front)        } if queue is not FULL.
    {
```

OR

```
    if (!isFull ())
    {
        rear = (rear+1)% n;
        a[rear] = value;
        if (front == -1)                } For case 1:
            front = rear;               } if queue is empty then we also move front one place forward i.e From '-1' to '0'.
    } else {
        cerr << "Queue is Full";
    }
}
```

special case ←