



# STACKS & QUEUES

– PINAK PATEL

# OUTLINE

- ‡ Arrays: Definition, Memory organization, Operations on Arrays: Traversing, Insertion, Deletion
- ‡ Updating, Resizing., Stacks: Basic operations, Stack , Dstack and applications
- ‡ Queues : Operations of queues, Circular Queue, Priority Queue, Dequeue, Application of queues

# ARRAYS

- Imagine that we have 100 scores. We need to read them, process them and print them. We must also keep these 100 scores in memory for the duration of the program. We can define a hundred variables, each with a different name, as shown in Figure 1.

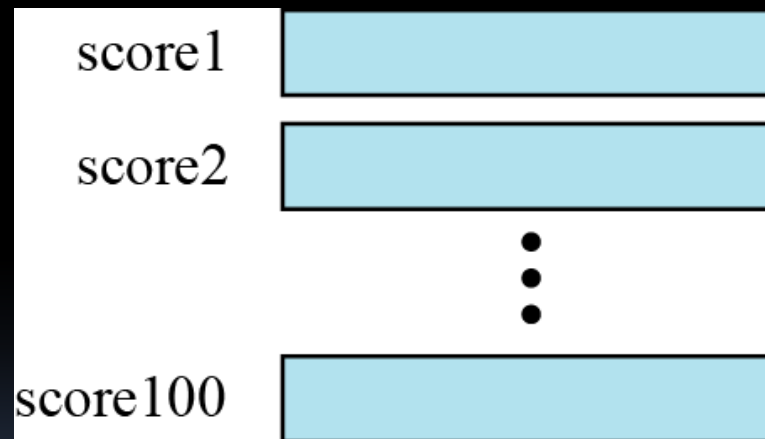


Fig.1 A hundred individual variables

- But having 100 different names creates other problems. We need 100 references to read them, 100 references to process them and 100 references to write them. Figure 2 shows a diagram that illustrates this problem.

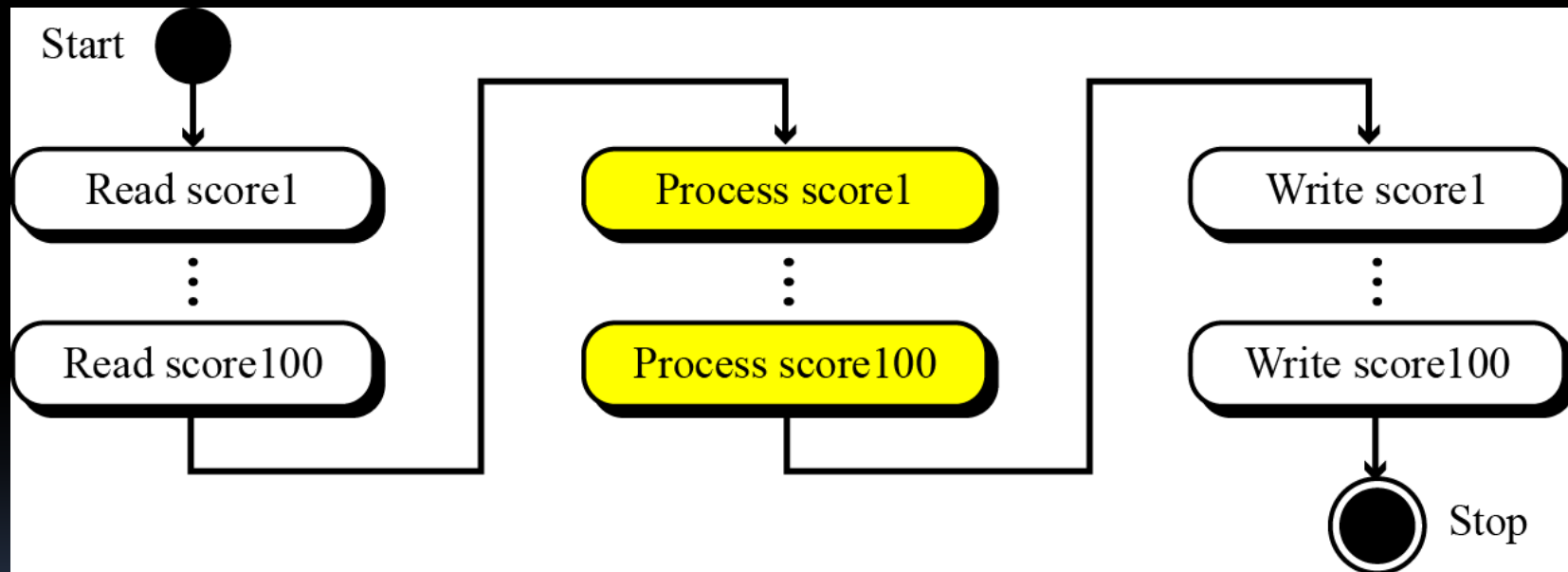


Fig. 2 Processing individual variables

- An array is a sequenced collection of elements, normally of the same data type, although some programming languages accept arrays in which elements are of different types. We can refer to the elements in the array as the first element, the second element and so forth, until we get to the last element.

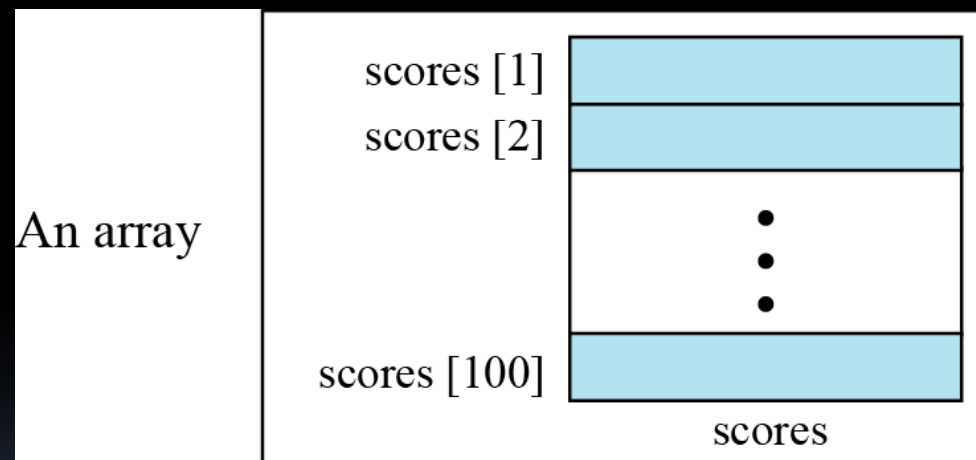


Fig. 3 Arrays with indexes

- We can use loops to read and write the elements in an array. We can also use loops to process elements. Now it does not matter if there are 100, 1000 or 10,000 elements to be processed—loops make it easy to handle them all. We can use an integer variable to control the loop and remain in the loop as long as the value of this variable is less than the total number of elements in the array (Figure 4).
- Note: We have used indexes that start from 1; some modern languages such as C, C++ and Java start indexes from 0.

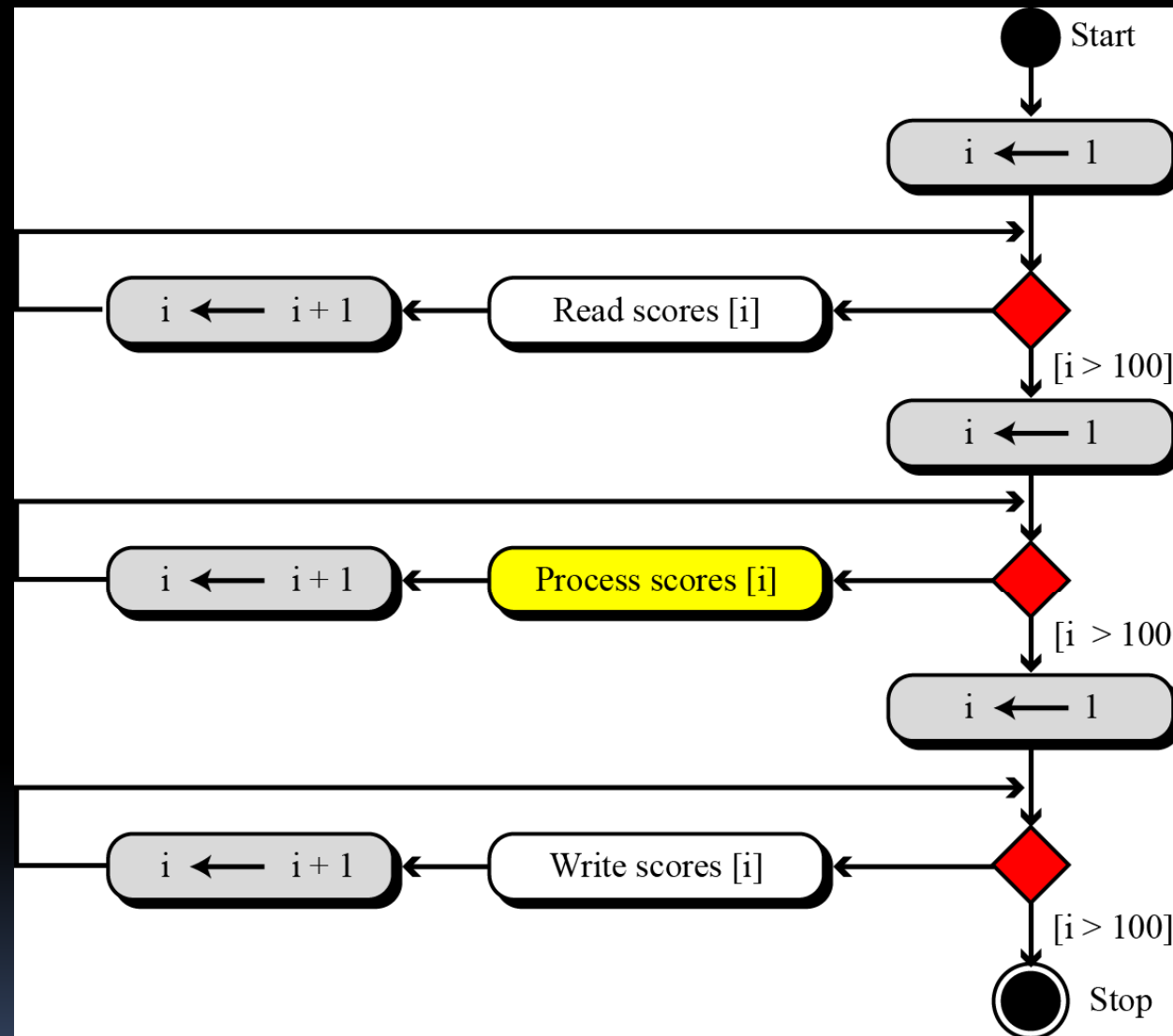



Fig. 4 Processing an array




Example: Compare the number of instructions needed to handle 100 individual elements in Figure 2 and the array with 100 in Figure 4. Assume that processing each score needs only one instruction.



Example: Compare the number of instructions needed to handle 100 individual elements in Figure 2 and the array with 100 in Figure 4. Assume that processing each score needs only one instruction.

- In the first case, we need 100 instructions to read, 100 instructions to write and 100 instructions to process. The total is 300 instructions.
- In the second case, we have three loops. In each loop we have two instructions, for a total of six instructions. However, we also need three instructions for initializing the index and three instructions to check the value of the index. In total, we have twelve instructions.



Example: In computer science, one of the big issues is the reusability of programs—for example, how much needs to be changed if the number of data items is changed. Assume we have written two programs to process the scores as shown in Figure. 2 and Figure. 4. If the number of scores changes from 100 to 1000, how many changes do we need to make in each program?

Example: In computer science, one of the big issues is the reusability of programs—for example, how much needs to be changed if the number of data items is changed. Assume we have written two programs to process the scores as shown in Figure. 2 and Figure. 4. If the number of scores changes from 100 to 1000, how many changes do we need to make in each program?

- In the first program we need to add  $3 \times 900 = 2700$  instructions. In the second program, we only need to change three conditions ( $I > 100$  to  $I > 1000$ ). We can actually modify the diagram in Figure 4 to reduce the number of changes to one.

# Array name versus element name

- In an array we have two types of identifiers: the name of the array and the name of each individual element. The name of the array is the name of the whole structure, while the name of an element allows us to refer to that element. In the array of Figure. 3, the name of the array is *scores* and name of each element is the name of the array followed by the index, for example, *scores*[1], *scores*[2], and so on. In this chapter, we mostly need the names of the elements, but in some languages, such as C, we also need to use the name of the array.

# Multi-dimensional arrays

- The arrays discussed so far are known as one-dimensional arrays because the data is organized linearly in only one direction. Many applications require that data be stored in more than one dimension. Figure. 5 shows a table, which is commonly called a two-dimensional array.

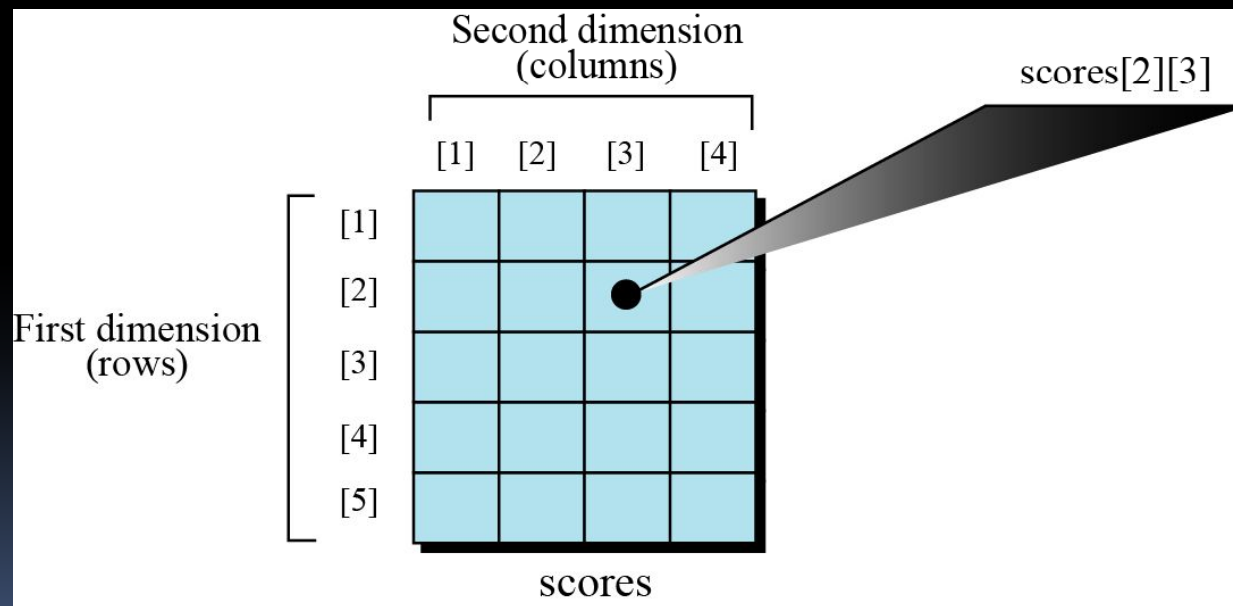


Fig. 5 A two-dimensional array

# Memory layout

- The indexes in a one-dimensional array directly define the relative positions of the element in actual memory. Figure. 6 shows a two-dimensional array and how it is stored in memory using row-major or column-major storage. Row-major storage is more common.

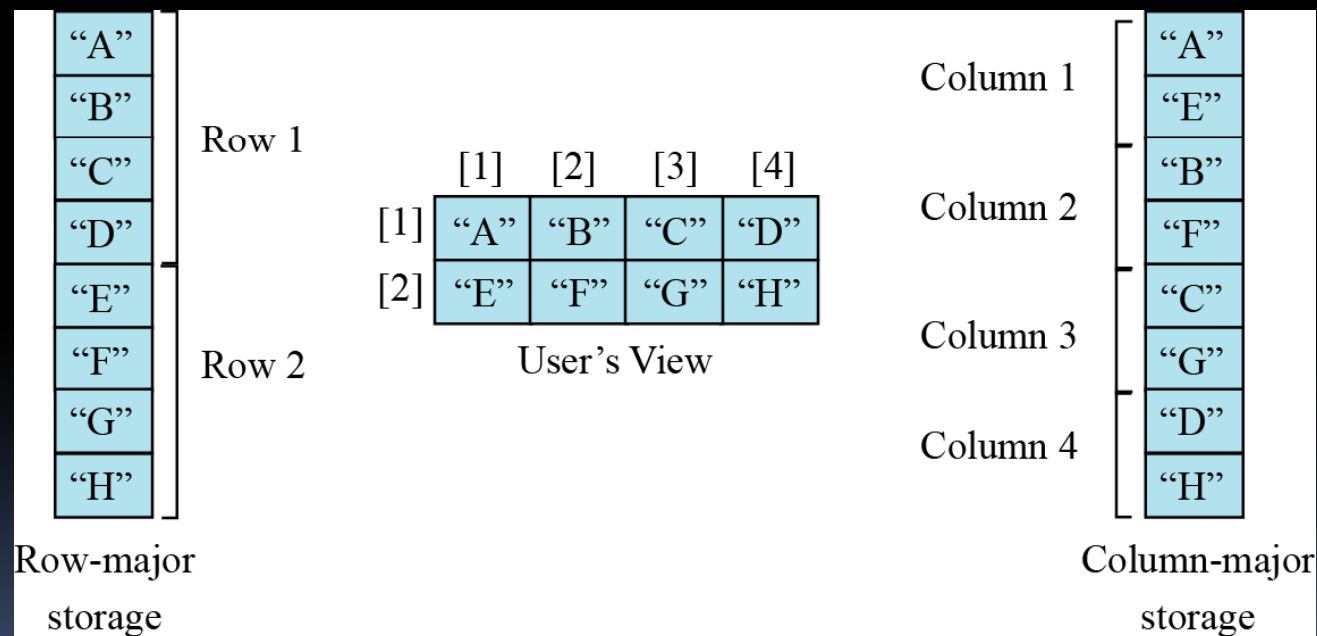


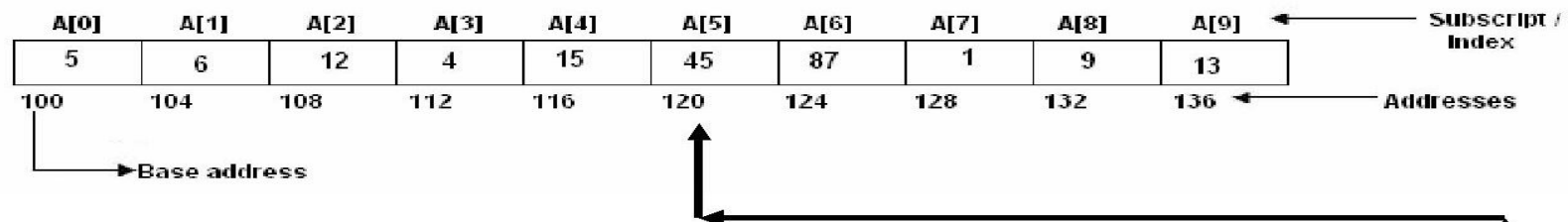
Fig. 6 Memory layout of arrays

Example: We have stored the two-dimensional array students in memory. The array is  $100 \times 4$  (100 rows and 4 columns). Show the address of the element students[5][3] assuming that the element student[1][1] is stored in the memory location with address 1000 and each element occupies only one memory location. The computer uses row-major storage.

- We can use the following formula to find the location of an element, assuming each element occupies one memory location.

$$y = x + \text{Cols} \times (i - 1) + (j - 1)$$

- If the first element occupies the location 1000, the target element occupies the location 1018.



## Representation of Linear Arrays in Memory

The memory of computer is simply a sequence of addressed locations as shown in Fig. Let **A** be a linear array stored in the memory of computer as given in above figure:

$\text{Add}(\mathbf{A}[k])$  = address of the  $\mathbf{A}[k]$  element of the array **A**

As the elements of the array **A** are stored in consecutive memory cells, the computer does not need to keep track of the address of every element of the array. It only needs to keep track of the address of the first element of the array which is denoted by:

$\text{Base}(\mathbf{A})$

It is called the base address of the **A**. Using this base address  $\text{Base}(\mathbf{A})$ , the computer calculates the address of any element of the array by using the following formula:

$$\text{Add}(\mathbf{A}[k]) = \text{Base}(\mathbf{A}) + w(k - \text{lower bound})$$

where  $w$  is the size of the data type of the array **A** and  $k$  is the index number.

Using the figure given on top of this page we are calculation the address of  $\mathbf{A}[5]$

$$\begin{aligned} \text{Add}(\mathbf{A}[5]) &= 100 + 4(5 - 0) \\ &= 120 \end{aligned}$$

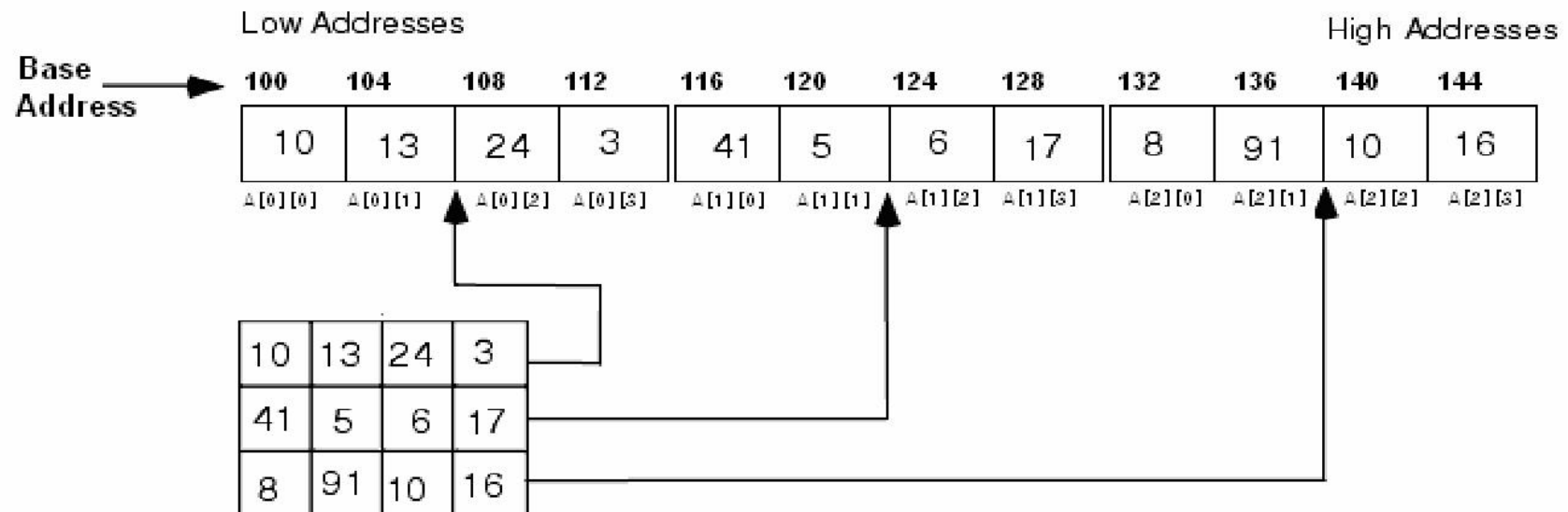
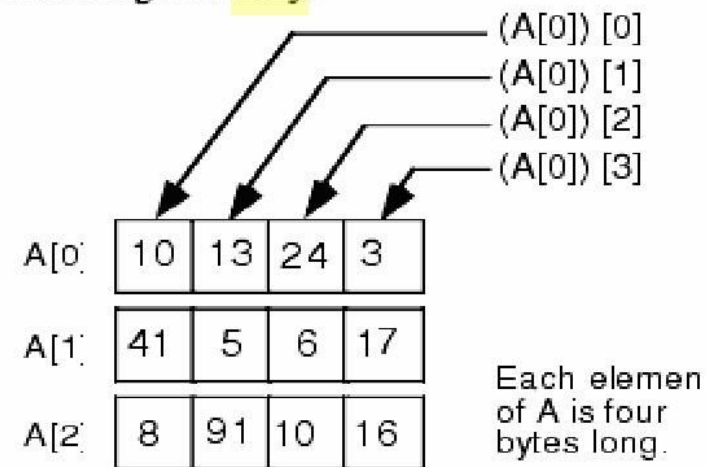


Now let us calculate the address of an element in the following 2-D array:

For Example:

```
int A[3][4] = {10,13,24,3,
               41,5,6,17,
               8,91,10,16};
```

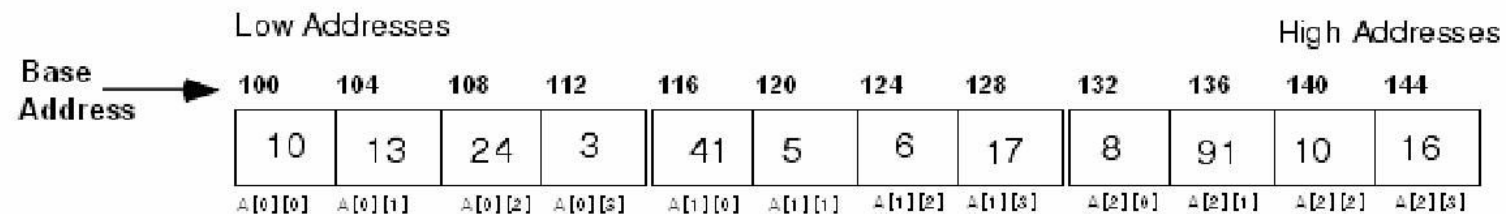
then how this 2-D array will be presented externally and in the memory of the computer using Row-Major-Order



```
int A[3][4] = {10,13,24,3, 41,5,6,17,8,91,10,16};
```

**Stored as Row-Major-Order**

Here  $M=3$  and  $N=4$



Formula to calculate/find the address of  $A[j][k]$ <sup>th</sup> element of a 2-D array of  $M \times N$  dimension is

$$A[j][k] = \text{base}(A) + W [ N (j - \text{Row\_LowBound}) + (k - \text{Col\_LowBound}) ]$$

where  $W$  = size of element

$N$  = number of columns

Let us assume that the base address of the array matrix is 100. Since  $W=4$  (as array is of integer type whose size is 4), therefore, according to the formula, address of  $(2, 3)$ <sup>th</sup> element in the array matrix will be

$$\begin{aligned} \text{LOC}(A[2][3]) &= 100 + 4 [ 4 (2 - 0) + (3 - 0) ] \\ &= 100 + 4 [ 8 + 3 ] \\ &= 100 + 4 [ 11 ] \\ &= 100 + 44 \\ &= 144 \end{aligned}$$

$$\begin{aligned} \text{LOC}(A[1][0]) &= 100 + 4 [ 4 (1 - 0) + (0 - 0) ] \\ &= 100 + 4 [ 4 + 0 ] \\ &= 100 + 4 [ 4 ] \\ &= 100 + 16 \\ &= 116 \end{aligned}$$

## Stored as Column-Major-Order

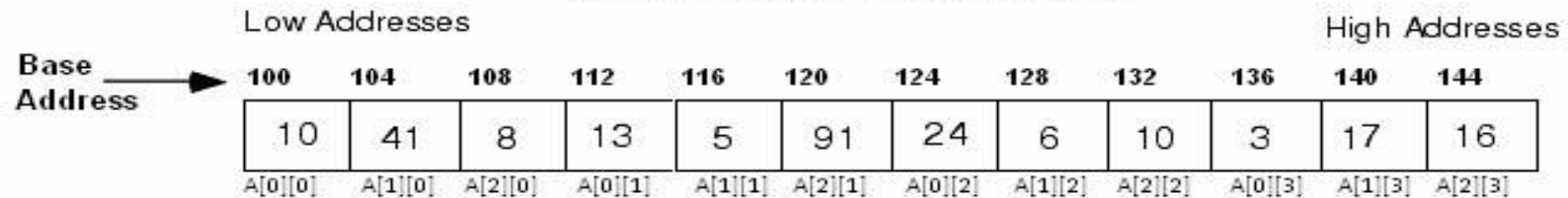
Similarly, for the **column major order** representation, let us consider the same matrix.

`int A[3][4] = {10,13,24,3, 41,5,6,17,8,91,10,16};`

Here  $M=3$  and  $N=4$

$$A(j, k) = \text{Base}(A) + W[M(k-1) + (j-1)]$$

10	13	24	3
41	5	6	17
8	91	10	16



Formula to calculate/find the address of  $A[j][k]$ <sup>th</sup> element of a 2-D array of  $M \times N$  dimension is

$$A[j][k] = \text{base}(A) + W[M(k - \text{Col\_LowBound}) + (j - \text{Row\_LowBound})]$$

where  $W$  = size of element

$M$  = number of Rows

Let us assume that the base address of the **array** matrix is 100. Since  $W=4$  (as **array** is of integer type whose size is 4), therefore, according to the **formula**, address of  $(2, 3)$ <sup>th</sup> element **in the array** matrix will be

$$\begin{aligned} \text{LOC}(A[2][1]) &= 100 + 4[3(1 - 0) + (2 - 0)] \\ &= 100 + 4[3 + 2] \\ &= 100 + 4[5] \\ &= 100 + 20 \\ &= 120 \end{aligned}$$

$$\begin{aligned} \text{LOC}(A[2][0]) &= 100 + 4[3(0 - 0) + (2 - 0)] \\ &= 100 + 4[0 + 2] \\ &= 100 + 4[2] \\ &= 100 + 8 \\ &= 108 \end{aligned}$$

# Operations on array

- **Traversing:** means to visit all the elements of the array in an operation is called traversing.
- **Insertion:** means to put values into an array
- **Deletion / Remove:** to delete a value from an array.
- **Sorting:** Re-arrangement of values in an array in a specific order (Ascending / Descending) is called sorting.
- **Searching:** The process of finding the location of a particular element in an array is called searching.

## Traversing in Linear Array:

It means processing or visiting each element in the array exactly once;

Let '**A**' is an array stored in the computer's memory. If we want to display the contents of '**A**', it has to be traversed i.e. by accessing and processing each element of '**A**' exactly once.

- **Algorithm:** (Traverse a Linear Array) Here **LA** is a Linear array with lower boundary **LB** and upper boundary **UB**. This algorithm traverses **LA** applying an operation Process to each element of **LA**.

1. [Initialize counter.] Set  $K=LB$ .
2. Repeat Steps 3 and 4 while  $K \leq UB$ .
3. [Visit element.] Apply PROCESS to  $LA[K]$ .
4. [Increase counter.] Set  $k=K+1$ . [End of Step 2 loop.]
5. Exit.

# Application

- Thinking about the operations discussed in the previous section gives a clue to the application of arrays. If we have a list in which a lot of insertions and deletions are expected after the original list has been created, we should not use an array. An array is more suitable when the number of deletions and insertions is small, but a lot of searching and retrieval activities are expected.
- Note: An array is a suitable structure when a small number of insertions and deletions are required, but a lot of searching and retrieval is needed.

# RECORDS

- A record is a collection of related elements, possibly of different types, having a single name. Each element in a record is called a field.
- A field is the smallest element of named data that has meaning.
- A field has a type and exists in memory.
- Fields can be assigned values, which in turn can be accessed for selection or manipulation.
- A field differs from a variable primarily in that it is part of a record.

Figure. 7 contains two examples of records. The first example, fraction, has two fields, both of which are integers. The second example, student, has three fields made up of three different types.

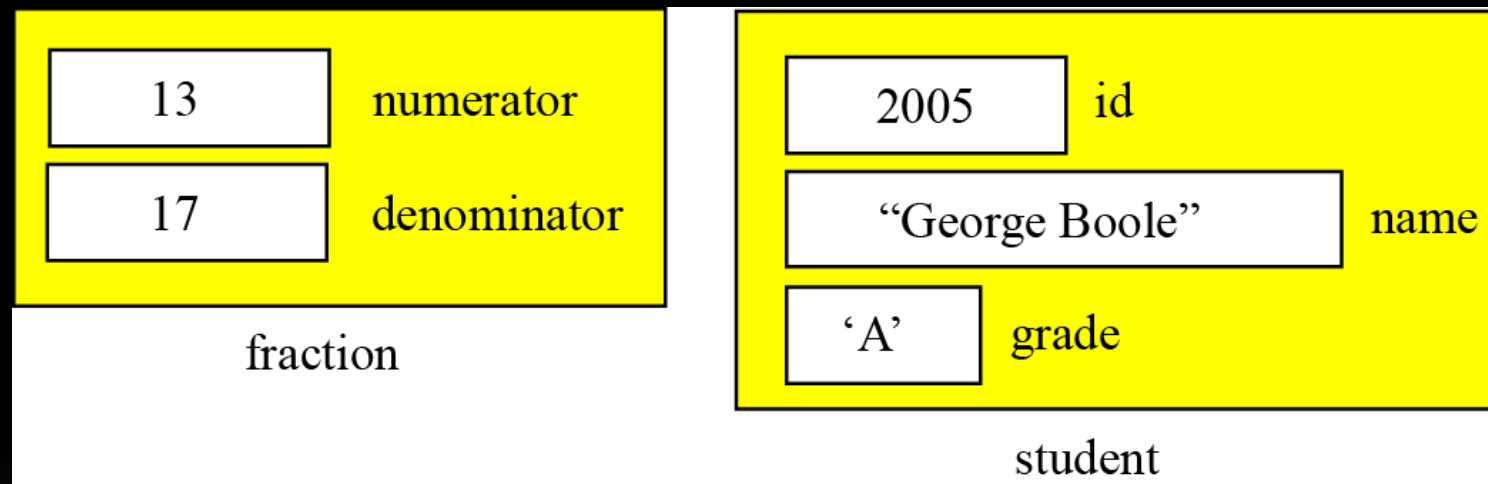


Fig. 7 Records



## Record name *versus* field name

- Just like in an array, we have two types of identifier in a record: the name of the record and the name of each individual field inside the record. The name of the record is the name of the whole structure, while the name of each field allows us to refer to that field. For example, in the student record of Figure. 7, the name of the record is student, the name of the fields are student.id, student.name and student. grade. Most programming languages use a *period* (.) to separate the name of the structure (record) from the name of its components (fields).

Example: The following shows how the value of fields in Figure. 7 are stored.

```
student.id ← 2005    student.name ← "G. Boole"    student.grade ← 'A'
```

# Comparison of records and arrays stored.

- We can conceptually compare an array with a record.
- This helps us to understand when we should use an array and when to use a record.
- An array defines a combination of elements, while a record defines the identifiable parts of an element.
- For example, an array can define a class of students (40 students), but a record defines different attributes of a student, such as id, name or grade.

# Array of records

- If we need to define a combination of elements and at the same time some attributes of each element, we can use an array of records. For example, in a class of 30 students, we can have an array of 30 records, each record representing a student.

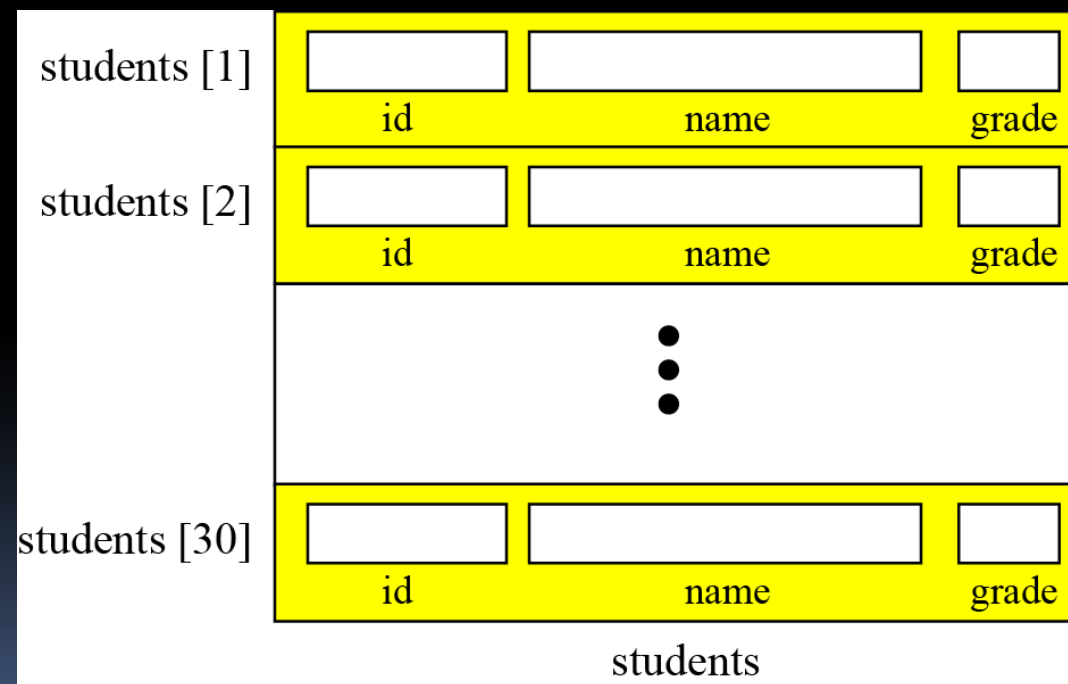


Fig. 8 Array of records

Example: The following shows how we access the fields of each record in the students array to store values in them.

<code>(students[1]).id ← 1001</code>	<code>(students[1]).name ← "J. Aron"</code>	<code>(students[1]).grade ← 'A'</code>
<code>(students[2]).id ← 2007</code>	<code>(students[2]).name ← "F. Bush"</code>	<code>(students[2]).grade ← 'F'</code>
...	...	...
<code>(students[30]).id ← 3012</code>	<code>(students[30]).name ← "M. Blair"</code>	<code>(students[1]).grade ← 'B'</code>

# STACKS

# Introduction to the Stack ADT

- It is an ordered group of homogeneous items or elements.
- Elements are added to and removed from the top of the stack (the most recently added items are at the top of the stack).
- The last element to be added is the first to be removed (LIFO: Last In, First Out).
- A stack is a list of elements in which an element may be inserted or deleted only at one end, called **TOP** of the stack.
- The elements are removed in reverse order of that in which they were inserted into the stack.

Cont..

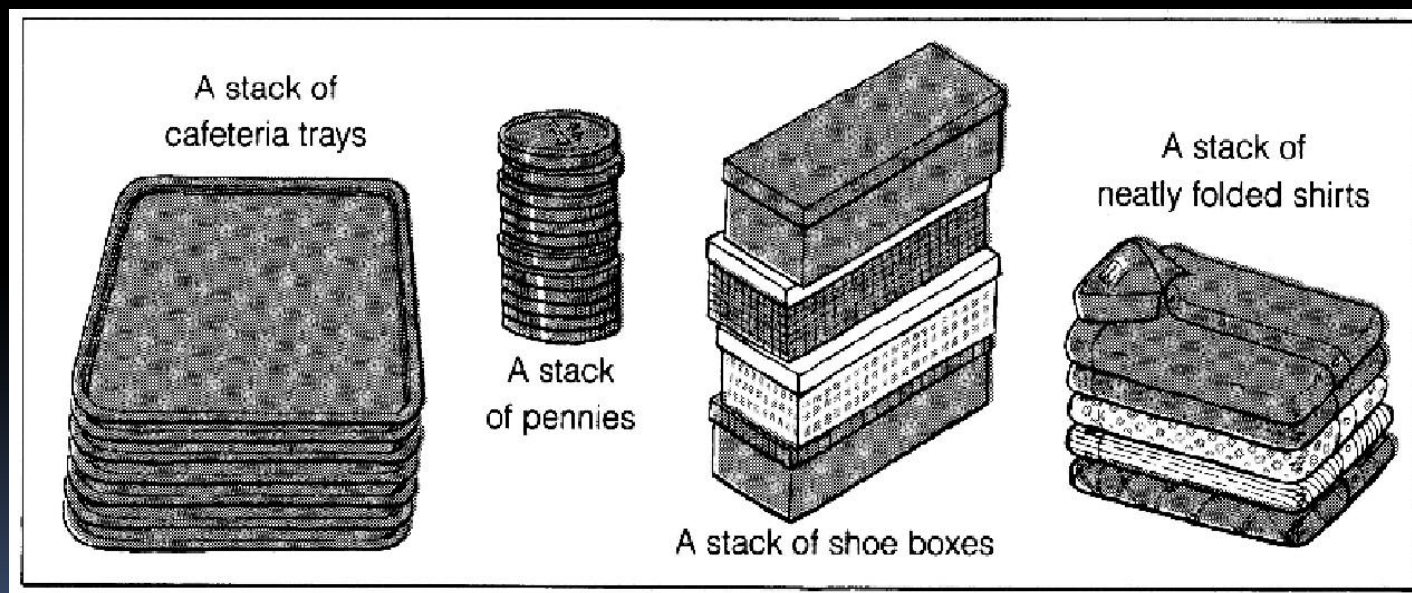
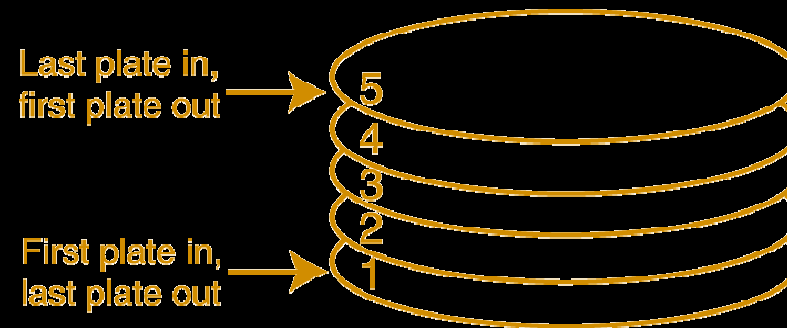


Fig. 9 Examples of Stack



# Static and Dynamic Stacks

- Static Stacks
  - Fixed size
  - Can be implemented with an array
- Dynamic Stacks
  - Grow in size as needed
  - Can be implemented with a linked list

# Stack Operations

- Push
  - causes a value to be stored in (pushed onto) the stack
- Pop
  - retrieves and removes a value from the stack

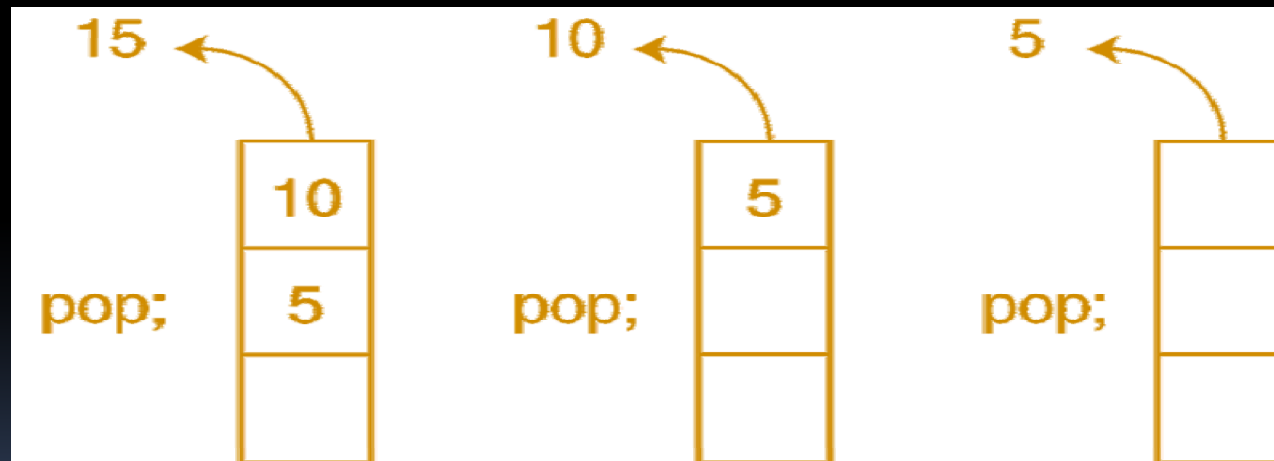
# The Push Operation

- The state of the stack after each of the push operations:

	5		10		15
push(5);		push(10);	5	push(15);	10
					5

# The Pop Operation

- Now, suppose we execute three consecutive pop operations on the same stack:



# Stack ADT Interface

- The main functions in the Stack ADT are (S is the stack)

<b>boolean isEmpty();</b>	// return true if empty
<b>boolean isFull(S);</b>	// return true if full
<b>void push(S, item);</b>	// insert <i>item</i> into stack
<b>void pop(S);</b>	// remove most recent item
<b>void clear(S);</b>	// remove all items from stack
<b>Item top(S);</b>	// retrieve most recent item
<b>Item topAndPop(S);</b>	// return & remove most recent item

# Algorithm for PUSH:

- **Algorithm:** PUSH(STACK, TOP, STACKSIZE, ITEM)
  1. [STACK already filled?] If  $TOP = STACKSIZE - 1$ , then: Print: OVERFLOW / Stack Full, and Return.
  2. Set  $TOP := TOP + 1$ . [Increase TOP by 1.]
  3. Set  $STACK[TOP] = ITEM$ . [Insert ITEM in new TOP position.]
  4. RETURN.

## Algorithm for POP:

- **Algorithm:** POP(STACK, TOP, ITEM)
  - This procedure deletes the top element of STACK and assigns it to the variable ITEM.
- 
1. [STACK has an item to be removed? Check for empty stack]  
If  $TOP = -1$ , then: Print: UNDERFLOW/ Stack is empty, and Return.
  2. Set  $ITEM = STACK[TOP]$ . [Assign TOP element to ITEM.]
  3. Set  $TOP = TOP - 1$ . [Decrease TOP by 1.]
  4. Return.

# Application of the Stack

- Computer systems use stacks during a program's execution to store function return addresses, local variables, etc.
- Some calculators use stacks for performing mathematical operations.
- line editing
- bracket matching
- postfix calculation
- function call stack



## Line Editing

- A line editor would place characters read into a buffer but may use a backspace symbol (denoted by ←) to do error correction.
- *Refined Task*
  - read in a line
  - correct the errors via backspace
  - print the corrected line in reverse

Input : `abc_defgh←2klpqr←←wxyz`


Corrected Input : `abc_defg2klpwxzy`

Reversed Output : `zyxwplk2gfed_cba`

## Cont...

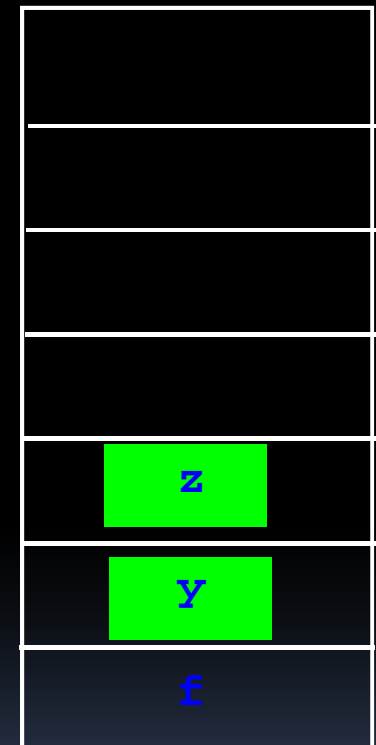
- Initialize a new stack
- For each character read:
  - if it is a backspace, *pop out last char entered*
  - if not a backspace, *push the char into stack*
- To print in reverse, pop out each char for output

Input : fgh←r←←yz



Corrected Input : fyz

Reversed Output : zyf



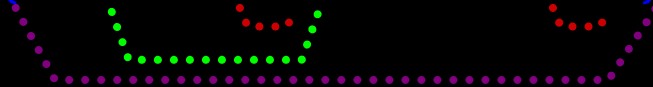
Stack

# Bracket Matching Problem

- Ensures that pairs of brackets are properly matched

An Example:

`{a, (b+f[4])*3, d+f[5]}`



Bad Examples:

`(..)..` // too many closing brackets

`(..(..)` // too many open brackets

`[..(..)]..` // mismatched brackets

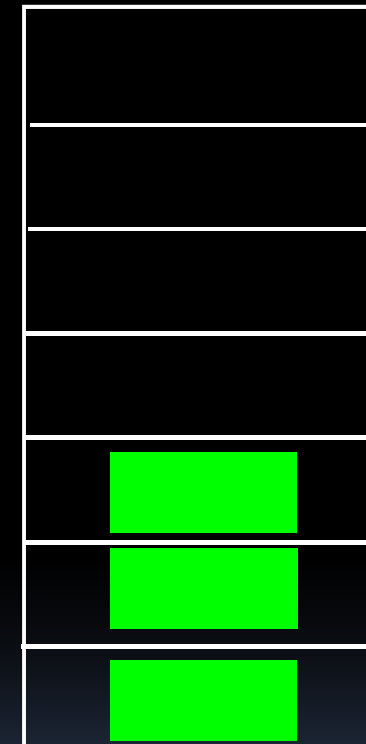


## Cont...

- Initialize the stack to empty
- For every char read
  - if open bracket then *push onto stack*
  - if close bracket, then
    - return & remove most recent item from *the stack*
    - if doesn't match then *flag error*
  - if non-bracket, *skip the char read*

### Example

{a, (b+f[4])\*3, d+f[5]}



*Stack*

## Application of the Stack (Arithmetic Expressions)

- Infix, Postfix & Prefix operations

Infix	Postfix	Prefix
$P+Q$	$PQ+$	$+PQ$
$P+Q-R$	$PQ+R-$	$-+PQR$
$(P+Q)*(R-S)$	$PQ+RS-*$	$*+PQ-RS$

Precedence	Binary Operations
Highest	Exponentiations (^)
Next Highest	Multiplication (*), Division (/) and Mod (%)
Lowest	Addition (+) and Subtraction (-)

Convert Q:  $A+(B * C-(D / E^F) * G) * H$  into postfix form showing stack status .

Now add “)” at the end of expression,

$A+(B * C-(D / E^F) * G) * H)$  and also Push a “(” on Stack.

Symbol Scanned	Stack	Expression Y
	(	
A	(	A
+	(+	A
(	(+(	A
B	(+(	AB
*	(+(*	AB
C	(+(*	ABC
-	(+(-	ABC*
(	(+(-(	ABC*
D	(+(-(	ABC*D
/	(+(-(/	ABC*D
E	(+(-(/	ABC*DE
^	(+(-(/^	ABC*DE
F	(+(-(/^	ABC*DEF
)	(+(-	ABC*DEF^/
*	(+(-*	ABC*DEF^/
G	(+(-*	ABC*DEF^/G
)	(+	ABC*DEF^/G*-
*	(+*	ABC*DEF^/G*-
H	(+*	ABC*DEF^/G*-H
)	empty	ABC*DEF^/G*-H*+

## Transforming Infix Expression into Postfix Expression:

The following algorithm transform the infix expression **Q** into its equivalent postfix expression **P**. It uses a stack to temporary hold the operators and left parenthesis. The postfix expression will be constructed from left to right using operands from **Q** and operators popped from **STACK**.

- **Algorithm: Infix\_to\_PostFix(Q, P)**
- Suppose **Q** is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression **P**.
- 
- 1. Push "(" onto **STACK**, and add ")" to the end of **Q**.
- 2. Scan **Q** from left to right and repeat Steps 3 to 6 for each element of **Q** until the **STACK** is empty:
- 3. If an operand is encountered, add it to **P**.
- 4. If a left parenthesis is encountered, push it onto **STACK**.
- 5. If an operator  $\odot$  is encountered, then:
  - 1. Repeatedly pop from **STACK** and add to **P** each operator (on the top of **STACK**) which has the same or higher precedence/priority than  $\odot$
  - 2. Add  $\odot$  to **STACK**.[End of If structure.]
- 6. If a right parenthesis is encountered, then:
  - 1. Repeatedly pop from **STACK** and add to **P** each operator (on the top of **STACK**) until a left parenthesis is encountered.
  - 2. Remove the left parenthesis. [Do not add the left parenthesis to **P**.][End of If structure.]
- [End of Step 2 loop.]
- 7. Exit.

**For example:**

Following is an infix arithmetic expression  $(5 + 2) * 3 - 8 / 4$

And its postfix is:  $5\ 2\ +\ 3\ *\ 8\ 4\ /\ -$

Now add "\$" at the end of expression as a sentinel.

Scanned Elements	Stack	Action to do
5	5	Pushed on stack
2	5, 2	Pushed on stack
+	7	Remove the two top elements and calculate $5 + 2$ and push the result on stack
3	7, 3	Pushed on Stack
*	21	Remove the two top elements and calculate $7 * 3$ and push the result on stack
8	21, 8	Pushed on Stack
4	21, 8, 4	Pushed on Stack
/	21, 2	Remove the two top elements and calculate $8 / 2$ and push the result on stack
-	19	Remove the two top elements and calculate $21 - 2$ and push the result on stack
\$	19	<b>Sentinel \$ encouter , Result is on top of the STACK</b>



## Evaluation of Postfix Expression:

If **P** is an arithmetic expression written in postfix notation. This algorithm uses **STACK** to hold operands, and evaluate **P**.

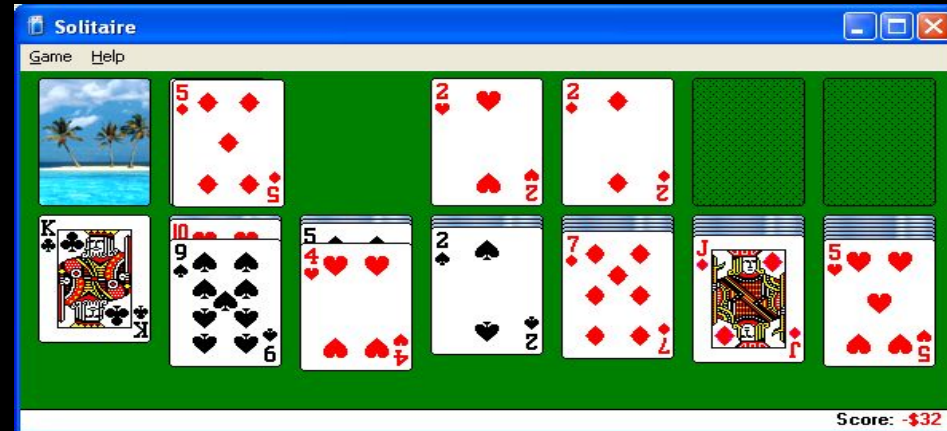
- **Algorithm:** This algorithm finds the **VALUE** of **P** written in postfix notation.
- 1. Add a Dollar Sign "\$" at the end of **P**. [This acts as sentinel.]
- 2. Scan **P** from left to right and repeat Steps 3 and 4 for each element of **P** until the sentinel "\$" is encountered.
- 3. If an operand is encountered, put it on **STACK**.
- 4. If an operator © is encountered, then:
  - 1. Remove the two top elements of **STACK**, where **A** is the top element and **B** is the next-to—top-element.
  - 2. Evaluate **B © A**.
  - 3. Place the result of (b) back on **STACK**.[End of If structure.]  
[End of Step 2 loop.]
- 5. Set **VALUE** equal to the top element on **STACK**.
- 6. Exit.

# Algorithm for Playing Solitaire

*WHILE (deck not empty)*

*Pop the deckStack*

*Check for Aces*



*While (There are playStacks to check)*

*If (can place card)*

*Push card onto playStack*

*Else*

*push card onto usedStack*

# QUEUES

# QUEUES

- A queue is a linear list in which data can only be inserted at one end, called the rear, and deleted from the other end, called the front.
- These restrictions ensure that the data is processed through the queue in the order in which it is received.
- In other words, a queue is a first in, first out (FIFO) structure.

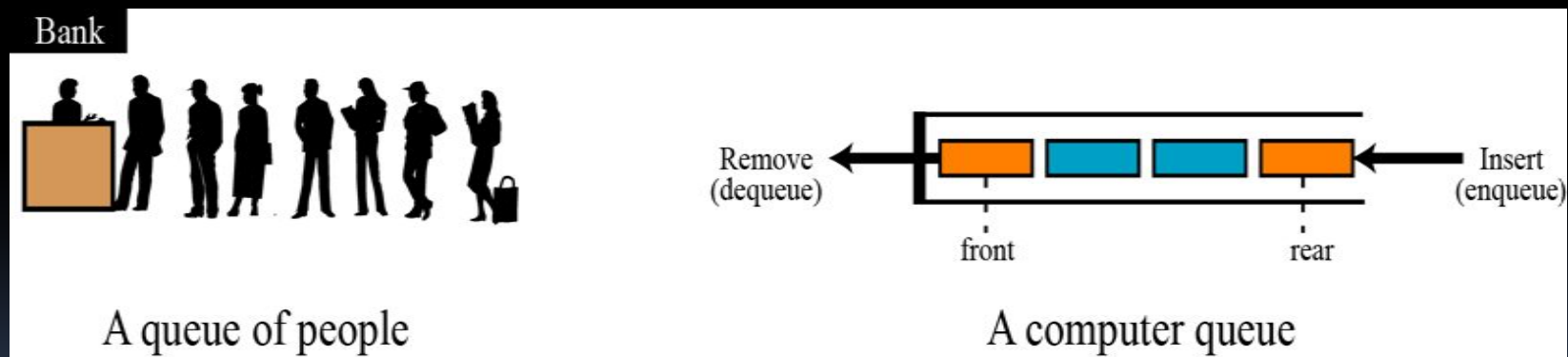


Fig. 10 Examples of Queue

# Static and Dynamic Queues

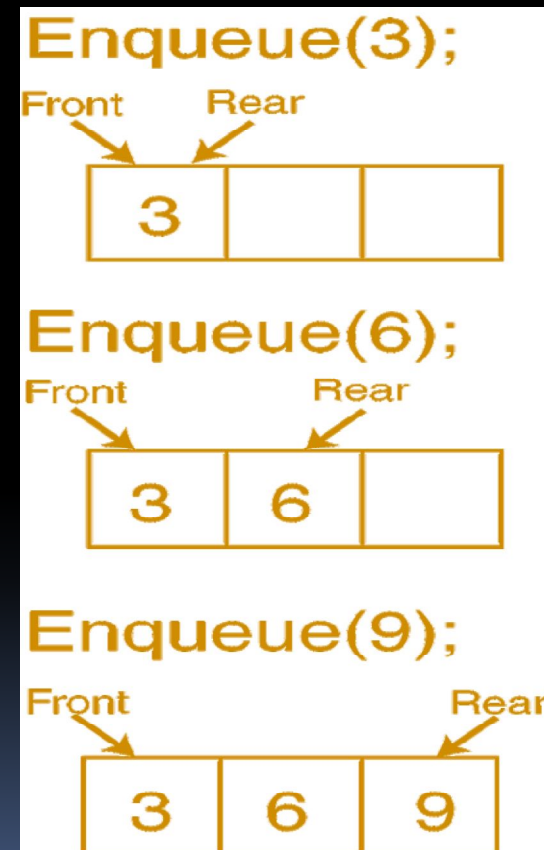
- Implementation:
  - static: fixed size, implemented as array
  - dynamic: variable size, implemented as linked list
- Applications:
  - An electronic mailbox is a queue
    - The ordering is chronological (by arrival time)
  - A waiting line in a store, at a service counter, on a one-lane road
  - Equal-priority processes waiting to run on a processor in a computer system
  - people in line at the theatre box office
  - print requests sent by users to a network printer

# Queue Operations

- Rear (tail): position where elements are added
- Front (head): position from which elements are removed
- Enqueue : add an element to the rear of the queue
- Dequeue : remove an element from the front of a queue

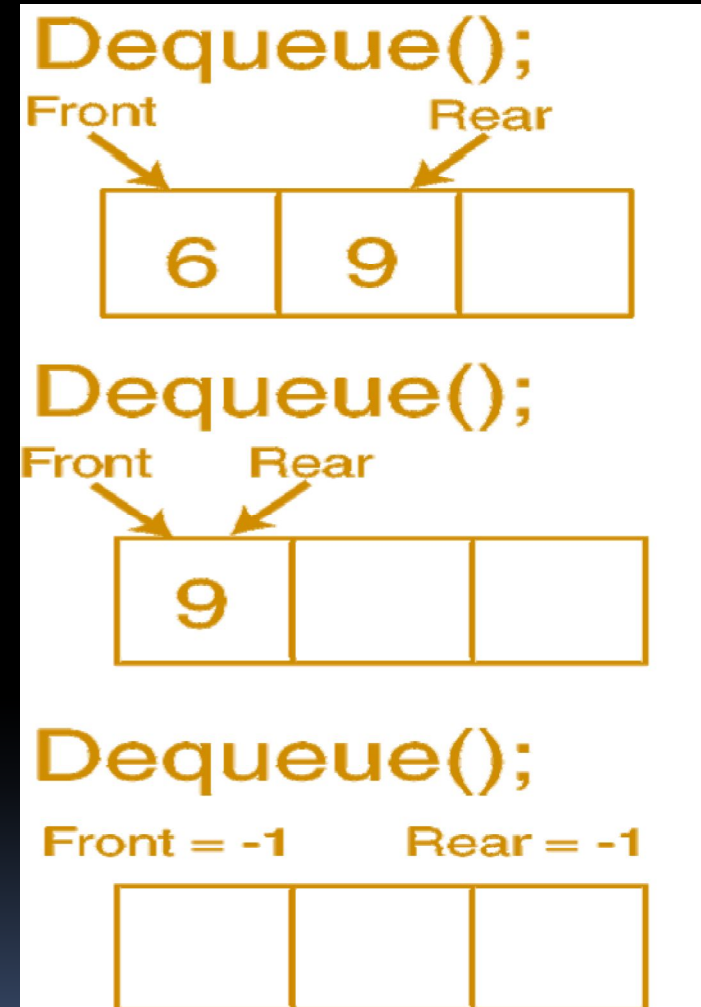
## Cont..

- Suppose we have an empty static integer queue that is capable of holding a maximum of three values. With that queue we execute the following enqueue operations.
- Enqueue(3);
- Enqueue(6);
- Enqueue(9);
- Given figure illustrates the state of the queue after each of the enqueue operations.



## Cont..

- Now let's see how dequeue operations are performed. Given figure illustrates the state of the queue after each of three consecutive dequeue operations.





# Array Implementation Issues

- In the preceding example, Front never moves.
- Whenever **dequeue** is called, all remaining queue entries move up one position. This takes time.
- Alternate approach:
  - Circular array: **front** and **rear** both move when items are added and removed. Both can 'wrap around' from the end of the array to the front if warranted.
- **enqueue** moves **rear** to the right as it fills positions in the array
- **dequeue** moves **front** to the right as it empties positions in the array
- When **enqueue** gets to the end, it wraps around to the beginning to use those positions that have been emptied
- When **dequeue** gets to the end, it wraps around to the beginning use those positions that have been filled

Enqueue wraps around by executing

```
rear=(rear + 1)% QSIZE;
```

Dequeue wraps around by executing

```
front=(front + 1)% QSIZE;
```

Following Figure shows that how a queue may be maintained by a circular array with  $\text{MAXSIZE} = 6$  (Six memory locations). The queue will be empty only when  $\text{Count} = 0$  or ( $\text{Front} = \text{Rear}$  but not null) and an element is deleted. For this reason, -1 (null) is assigned to Front and Rear.

(a) Initially QUEUE is Empty

- Front = -1
- Rear = -1
- Count = 0

0	1	2	3	4	5

(b) A, B, C are Enqueued / Inserted

- Front = 0
- Rear = 2
- Count = 3

A	B	C			
0	1	2	3	4	5

(c) A is Deleted / Dequeue

- Front = 1
- Rear = 2
- Count = 2

	B	C			
0	1	2	3	4	5

## Cont..

(d) D, E, F are Enqueued / Inserted

- Front = 1
- Rear = 5
- Count = 5

	B	C	D	E	F
0	1	2	3	4	5

(e) B and C are Deleted / Dequeue

- Front = 3
- Rear = 5
- Count = 3

			D	E	F
0	1	2	3	4	5

(f) G is Enqueued / Inserted

- Front = 3
- Rear = 0
- Count = 4

G			D	E	F
0	1	2	3	4	5

(g) D and E are Deleted / Dequeue

- Front = 5
- Rear = 0
- Count = 2

G					F
0	1	2	3	4	5

## Cont..

(h) H and I are Enqueued / Inserted

- Front = 5
- Rear = 2
- Count = 4

G	H	I			F
0	1	2	3	4	5

(i) F is Deleted / Dequeue

- Front = 0
- Rear = 2
- Count = 3

G	H	I			
0	1	2	3	4	5

(j) G and H are Deleted / Dequeue

- Front = 2
- Rear = 2
- Count = 1

		I			
0	1	2	3	4	5

(k) I is Deleted. Queue is *Empty*

- Front = -1
- Rear = -1
- Count = 0

0	1	2	3	4	5

# Insertion in Queue:

Algorithm: ENQUEUE(Queue, MAXSIZE, FRONT, REAR, COUNT, ITEM)

- This algorithm inserts an element ITEM into a circular queue.

## 1. [Queue already filled?]

- If COUNT = MAXSIZE then: [COUNT is number of values in the Queue]
  - Write: OVERFLOW, and Return.

## 2. [Find new value of REAR.]

- If COUNT = 0, then: [Queue initially empty.]
  - Set FRONT = 0 and REAR = 0
- Else: if REAR = MAXSIZE - 1, then:
  - Set REAR = 0
- Else:
  - Set REAR = REAR + 1.
- [End of If Structure.]

3. Set Queue[REAR] = ITEM. [This insert new element.]

4. COUNT = COUNT + 1 [Increment to Counter.]

5. Return.

## Deletion in Queue:

**Algorithm: DEQUEUE(Queue, MAXSIZE, FRONT, REAR, COUNT, ITEM)**

- This procedure deletes an element from a queue and assigns it to the variable ITEM.

### **1. [Queue already empty?]**

- If COUNT = 0, then: Write: UNDERFLOW, and Return.

### **2. Set ITEM = Queue[FRONT].**

### **3. Set COUNT = COUNT - 1**

### **4. [Find new value of FRONT.]**

- If COUNT = 0, then: [There was one element and has been deleted]
  - Set FRONT = -1, and REAR = -1.
- Else if FRONT = MAXSIZE, then: [Circular, so set Front = 0]
  - Set FRONT = 0
- Else:
  - Set FRONT := FRONT + 1.
- [End of If structure.]

### **5. Return ITEM**

# Priority Queue

- Suppose you have few assignments from different courses, which assignment will you want to work on first?

Course	Priority	Due day
DBMS	2	August 3
CG	4	August 9
C++	1	July 31
DSA	3	August 6
JAVA	5	August 15

- You set your priority based on due days. Due days are called keys

## Priority Queue Cont..

- It is collection of elements where elements are stored according to the their priority levels.
- Inserting and removing of elements from queue is decided by the priority of the elements.
- The two fundamental methods of a priority queue P:
  - `insertItem(k,e)`: Insert an element e with key k into P.
  - `removeMin()`: Return & remove from P an element with the smallest key.
- When you want to determine the priority for your assignments, you need a value for each assignment, that you can compare with each other.
- **key**: An object that is assigned to an element as a specific attribute for that element, which can be used to identify, rank, or weight that element.



# Rules to maintain a Priority Queue

- The elements with the higher priority will be processed before any element of lower priority.
- If there are elements with the same priority, then the element added first in the queue would get processed.

# Priority Queue Insertion

- PQInsertion (M, Item)
  1. Find the Row M
  2. [Reset the Rear Pointer]
    - If  $\text{Rear}[M] = N-1$  then
      - $\text{Rear}[M] = 0$
    - Else
      - $\text{Rear}[M] = \text{Rear}[M] + 1$
  3. [Overflow]
    - If  $\text{Front}[M] = \text{Rear}[M]$  then
      - Write ("This Priority Queue is full")
    - Return
  4. [Insert Element]
    - $Q[M][\text{Rear}[M]] = \text{Item}$
  5. [Is Front Pointer Properly Set]
    - If  $\text{Front}[M] = -1$  then
      - $\text{Front}[M] = 0$
    - Return
  6. Exit

# Priority Queue Deletion

- PQDelete (K, Item)
  1. Initialize  $K = 0$
  2. while (Front[K] = -1)
    - $K = K + 1$
    - [To find the first non empty queue]
  3. [Delete Element]
    - $Item = Q[K][Front[K]]$
  4. [Queue Empty]
    - If Front[K] = N-1 then
      - Front[K] = 0
    - Else
      - Front[K] = Front[K] + 1
    - Return Item
  5. Exit

# Priority Queue Insertion using array

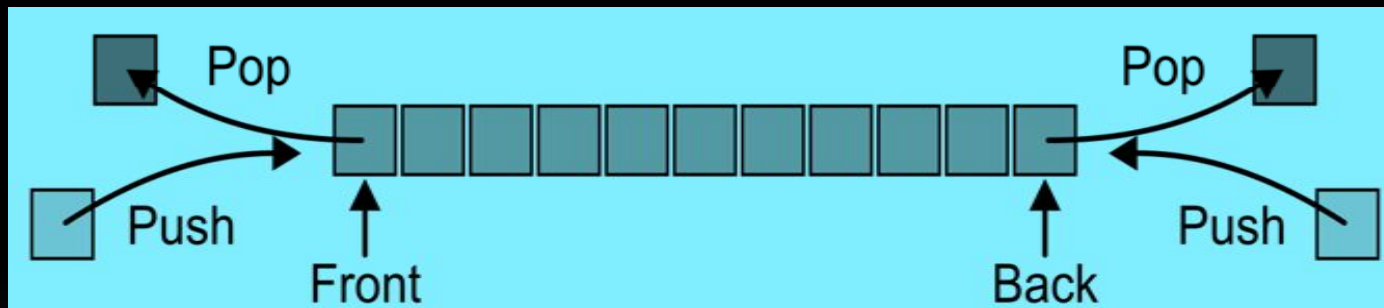
```
▪ int insert (int que[size], int rear, int front)
{
    int item,j;
    printf("Enter the element\n");
    scanf("%d",&item);
    if(front== -1)
        front++;
    j=rear;
    while(j>0 && item<que[j])
    {
        que[j+1]=que[j];
        j--;
    }
    que[j+1]=item;
    rear=rear+1;
    return rear;
}
```

## Priority Queue Deletion using array

```
■ int delete(int que[size], int front)
{
    int item;
    item=que[front];
    printf("The item deleted is %d",item);
    front++;
    return front;
}
```

# Double-Ended Queue

- A Deque or deck is a double-ended queue.
- Allows elements to be added or removed on either the ends.



# Types of DEQUE

- Input restricted Deque
  - Elements can be inserted only at one end.
  - Elements can be removed from both the ends.
- Output restricted Deque
  - Elements can be removed only at one end.
  - Elements can be inserted from both the ends.

# Deque as Stack and Queue

## As STACK

- When insertion and deletion is made at the same side.

## As Queue

- When items are inserted at one end and removed at the other end.



# Operations in DEQUE

- Insert element at back (rear).
- Insert element at front.
- Remove element at front.
- Remove element at back (rear).

## ALGORITHMS FOR INSERTING AN ELEMENT

- Let Q be the array of MAX elements.
- Front (or left) and rear (or right) are two array index (pointers), where the addition and deletion of elements occurred.
- Let DATA be the element to be inserted.
- Before inserting any element to the queue left and right pointer will point to the  $-1$ .

# INSERT AN ELEMENT AT THE RIGHT SIDE OF THE DE-QUEUE

1. Input the DATA to be inserted
2. If  $((\text{left} == 0 \ \&\& \ \text{right} == \text{MAX}-1) \ || \ (\text{left} == \text{right} + 1))$ 
  - ▣ (a) *Display "Queue Overflow"*
  - ▣ (b) *Exit*
3. If  $(\text{left} == -1)$ 
  - ▣ (a)  $\text{left} = 0$
  - ▣ (b)  $\text{right} = 0$Else
  - ▣ (a) *if*  $(\text{right} == \text{MAX}-1)$ 
    - $\text{right} = 0$
  - ▣ (b) *else*
    - $\text{right} = \text{right}+1$
4.  $\text{Q}[\text{right}] = \text{DATA}$
5. Exit

# INSERT AN ELEMENT AT THE LEFT SIDE OF THE DE-QUEUE

1. Input the DATA to be inserted
2. If  $((\text{left} == 0 \ \&\& \ \text{right} == \text{MAX}-1) \ || \ (\text{left} == \text{right}+1))$ 
  - (a) *Display "Queue Overflow"*
  - (b) *Exit*
3. If  $(\text{left} == -1)$ 
  - (a) *Left = 0*
  - (b) *Right = 0*

Else

  - (a) *if  $(\text{left} == 0)$* 
    - *$\text{left} = \text{MAX} - 1$*
  - (b) *else*
    - *$\text{left} = \text{left} - 1$*
4.  $\text{Q}[\text{left}] = \text{DATA}$
5. Exit

# ALGORITHMS FOR DELETING AN ELEMENT

- Let Q be the array of MAX elements.
- Front (or left) and rear (or right) are two array index (pointers), where the addition and deletion of elements occurred.
- DATA will contain the element just deleted.

## DELETE AN ELEMENT FROM THE RIGHT SIDE OF THE DE-QUEUE

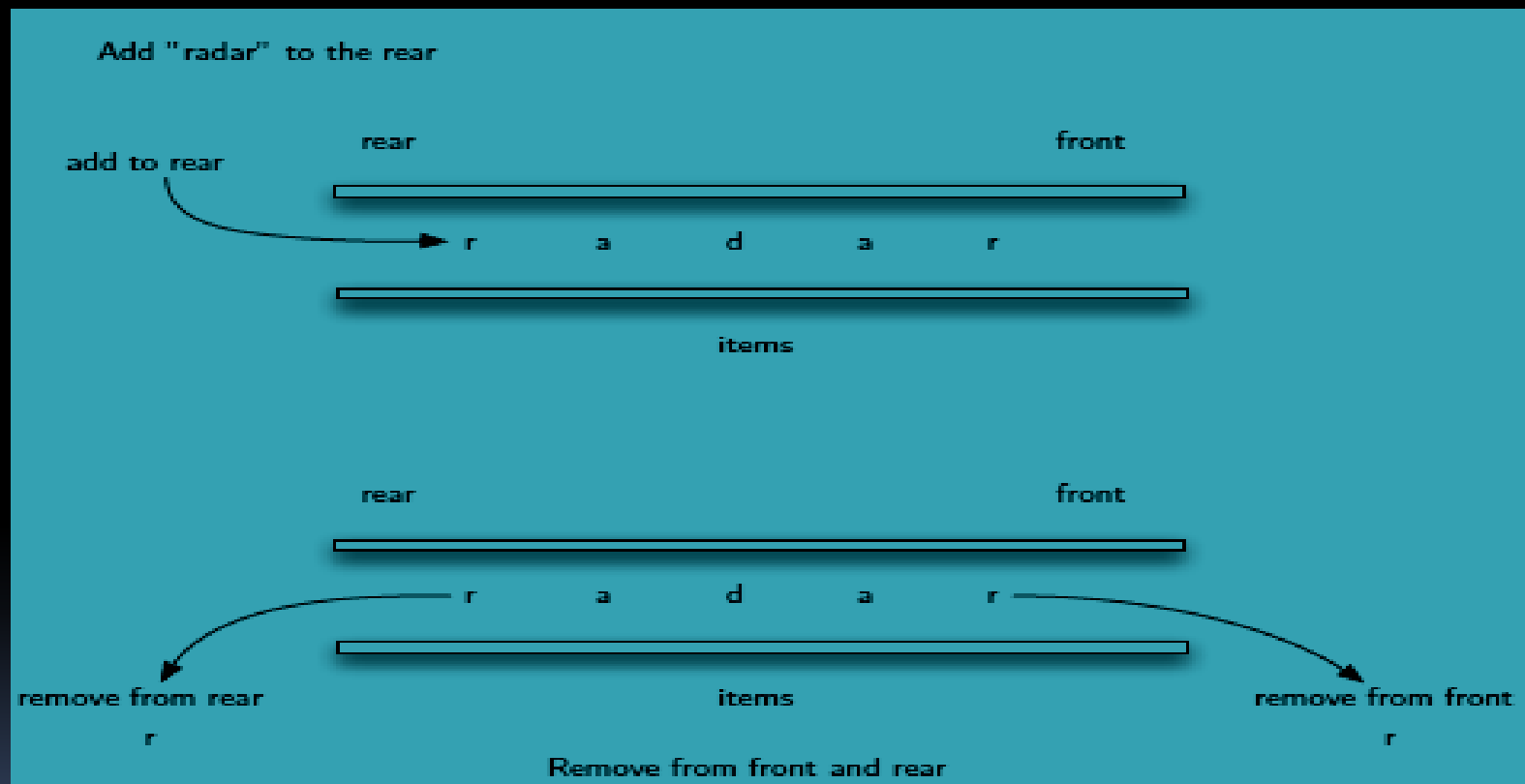
1. If ( $left == -1$ )
  - (a) *Display "Queue Underflow"*
  - (b) *Exit*
2.  $DATA = Q[right]$
3. If ( $left == right$ )
  - (a)  $left = -1$
  - (b)  $right = -1$Else
  - (a) *if*( $right == 0$ )
    - $right = MAX-1$
  - (b) *else*
    - $right = right-1$
4. Exit

## DELETE AN ELEMENT FROM THE LEFT SIDE OF THE DE-QUEUE

1. If ( $left == -1$ )
  - (a) *Display "Queue Underflow"*
  - (b) *Exit*
2.  $DATA = Q [left]$
3. If( $left == right$ )
  - (a)  $left = -1$
  - (b)  $right = -1$Else
  - (a) *if* ( $left == MAX-1$ )
    - $left = 0$
  - (b) *Else*
    - $left = left + 1$
4. Exit

# APPLICATIONS OF DEQUE

## 1. Palindrome-checker



## 2. Undo-Redo operations in Software applications.



**Thank You !**

**Questions ?**

