ECSE207L
DATA STRUCTURES

Dr. Tapas Badal
Dept. of CSE
Bennett University

COMMON DATA STRUCTURES

- ✓ List
  - ➢ Array
  - ➢ Linked List
- ✓ Record
- ✓ Stack
- ✓ Queue
- ✓ Tree
- ✓ Graph

DATA STRUCTURES:

- A data structure is a scheme for organizing data in the memory of a computer.
- The way in which the data is organized affects the performance of a program for different tasks.
- Computer Engineers decide which data structures to use based on the nature of the data and the *processes that need to be performed* on that data.

TYPES OF DATA STRUCTURES

1. Linear Vs Non-Linear
   - A data structure is said to be linear if its elements form a sequence like array or linked list
   - The data structure where there is no such sequence are called non-linear like tree and graph

2. Homogeneous Vs Non-Homogeneous
   - The data structure where we store similar type of data is called homogeneous data structure otherwise it is called non homogeneous.

LIST

Definition: List is a linear data structure, it contains data in a sequence.

Properties:

- Elements in the list are stored in sequence.
- Depending on the arrangement of data in memory, list data structure can be categorized into two:

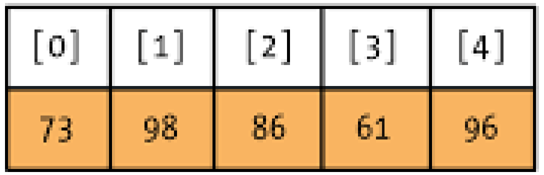
  - ✓ Array
  - ✓ Linked List

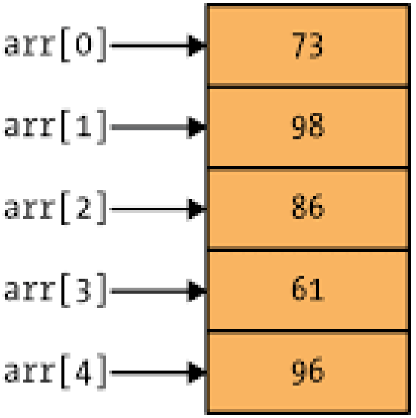
ARRAY

Definition: Array is a collection of homogeneous data elements stored in continuous memory locations.

Example:

Analogy: Books in a self.





Arrays are commonly used in computer programs to organize data so that a related set of values can be easily sorted or searched.

LINEAR STRUCTURE

ARRAY

Properties:

- The position of an element in the array is called index.
- The array elements can be accessed in sequential as well as in the random order with the help of index.

Example:

5	**3**	**1**	**7**	**4**	Array elements
0	1	2	3	4	Index of elements

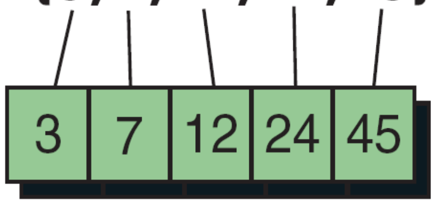
Operations on Array

- Traversal: Processing each element
- Search: Finding the location of the element with a given value
- Insertion: Adding a new element to the list
- Deletion: Deleting an element from the list
- Sort: Arranging the elements in some type of order
- Merge: Combining two lists into a single list

Initialization of the Array

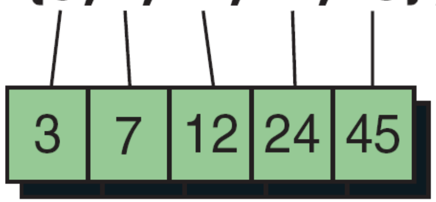
(a) Basic Initialization

```
int numbers[5] = {3,7,12,24,45};
```



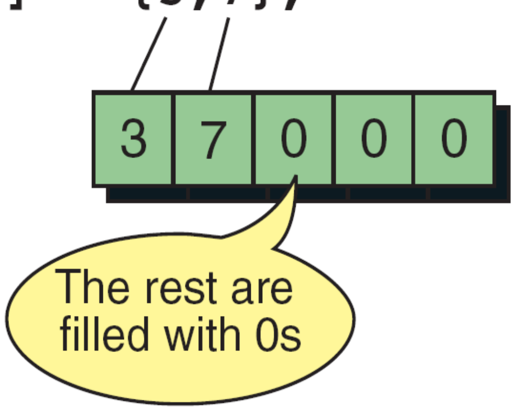
(b) Initialization without Size

```
int numbers[ ] = {3,7,12,24,45};
```



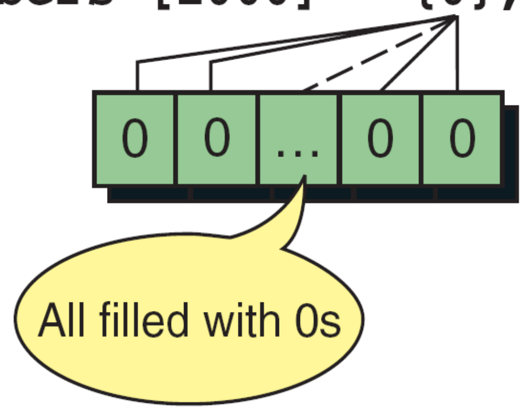
(c) Partial Initialization

```
int numbers[5] = {3,7};
```



(d) Initialization to All Zeros

```
int lotsOfNumbers [1000] = {0};
```



Address Calculation in Single (one) Dimension Array

Actual Address of the 1st element of the array is known as **Base Address (B)**
Here it is 1100

Memory space acquired by every element in the Array is called **Width (W)**
Here it is 4 bytes

Actual Address in the Memory	1100	1104	1108	1112	1116	1120
Elements	15	7	11	44	93	20
Address with respect to the Array (Subscript)	0	1	2	3	4	5

Lower Limit/Bound of Subscript **(LB)**

Traversing Algorithm

- Traversing operation means visit every element once. e.g. to print, etc.
- Example algorithm:

```
1. [Assign counter]
   K=LB   // LB = 0
2. Repeat step 2.1 and 2.2 while K <= UB   // If LB = 0
    2.1   [visit element]
    do PROCESS on LA[K]
        2.2        [add counter]
        K=K+1
3. end repeat step 2
4. exit
```

Insertion Algorithm

INSERT(LA, N, K, ITEM)
//LA is a linear array with N element
//K is integer positive where K < N and LB = 0
//Insert an element, ITEM in index K

1. [Assign counter]
   J = N – 1;   // LB = 0
2. Repeat step 2.1 and 2.2 while J >= K
   2.1 [shift to the right all elements from J]
       LA[J+1] = LA[J]
   2.2 [decrement counter]   J = J – 1
3. [Stop repeat step 2]
4. [Insert element]   LA[K] = ITEM
5. [Reset N]   N = N + 1
6. Exit

Deletion Algorithm

```
DELETE(LA, N, K, ITEM)
   1.  ITEM = LA[K]
   2.  Repeat for I = K to N–2    // If LB = 0
       2.1     [Shift element forward]
               LA[I] = LA[I+1]
   3.  [end of loop]
   4.  [Reset N in LA]
       N = N – 1
   5.  Exit
```

Address Calculation in single (one) Dimension Array

- Array of an element of an array say "A[I]" is calculated using the following formula:

- Address of A [I] = B + W * (I – LB)

- Where,
  - B = Base address
  - W = Storage Size of one element stored in the array (in byte)
  - I = Subscript of element whose address is to be found
  - LB = Lower limit / Lower Bound of subscript, if not specified assume 0 (zero)

2-D Array

- A 2-D array, A with *m X n* elements. In math application it is called *matrix.*

In business application – table.

Example:

Assume 25 students had taken 4 tests.

The marks are stored in 25 X 4 array locations:

	U0	U1	U2	U3
Stud 0	88	78	66	89
Stud 1	60	70	88	90
Stud 2	62	45	78	88
..	..	..	..	..
..	..	..	..	..
Stud 24	78	88	98	67

m

n

Declaration of 2D Array

- Multidimensional array declaration in Java:-

  int [][] StudentMarks = new int [25][4];

  StudentMarks[0][0] = 88;

  StudentMarks[0][1] = 78;…..

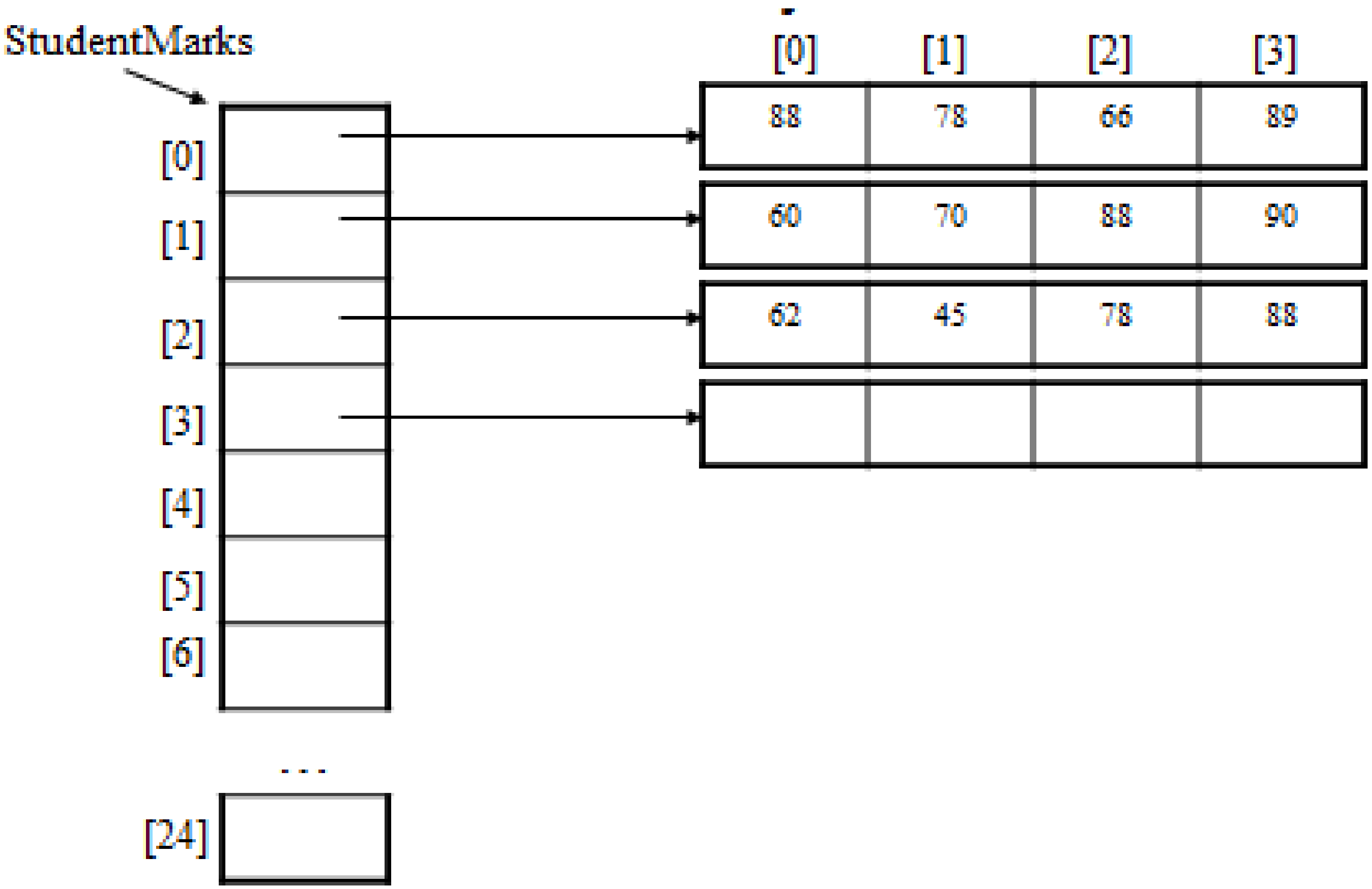
  OR

  int [][] StudentMarks = {{88, 78, 66, 89},
  {60, 70, 88, 90},…}

```
int disp[2][4] = {
    {10, 11, 12, 13},
    {14, 15, 16, 17}
};
```

```
int disp[2][4] = { 10, 11, 12, 13, 14, 15, 16, 17};
```

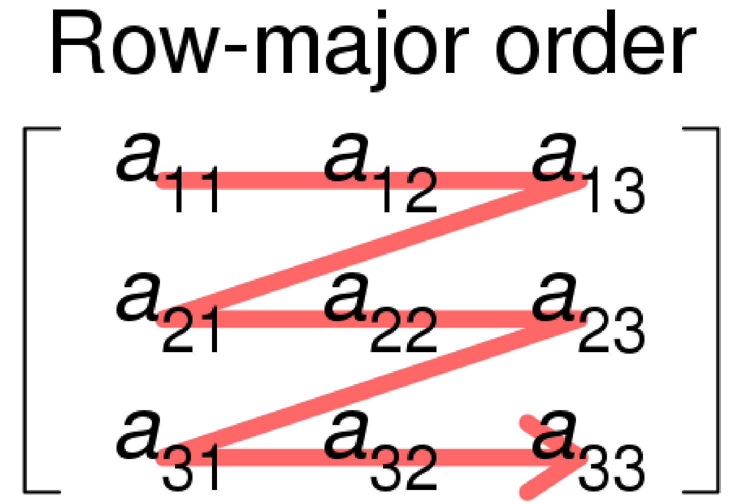
Visualization of 2-D Array



Implementation of 2D Array

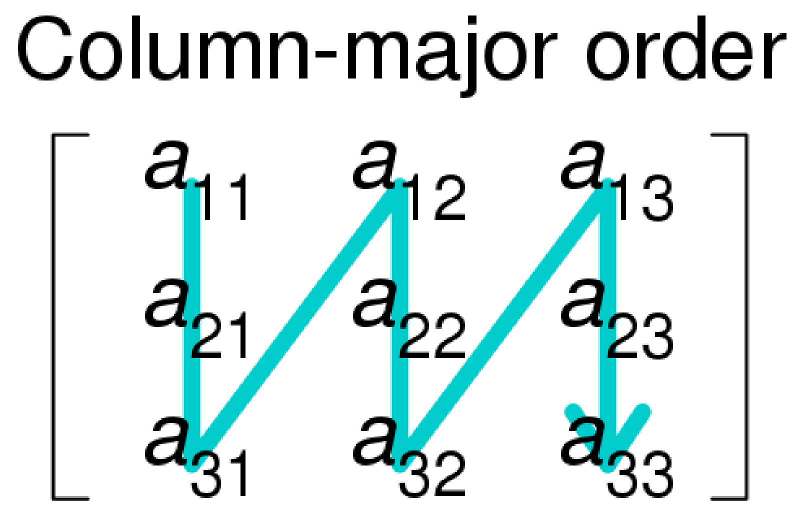
Row major order:

In this method elements of an array are arranged sequentially row by row.

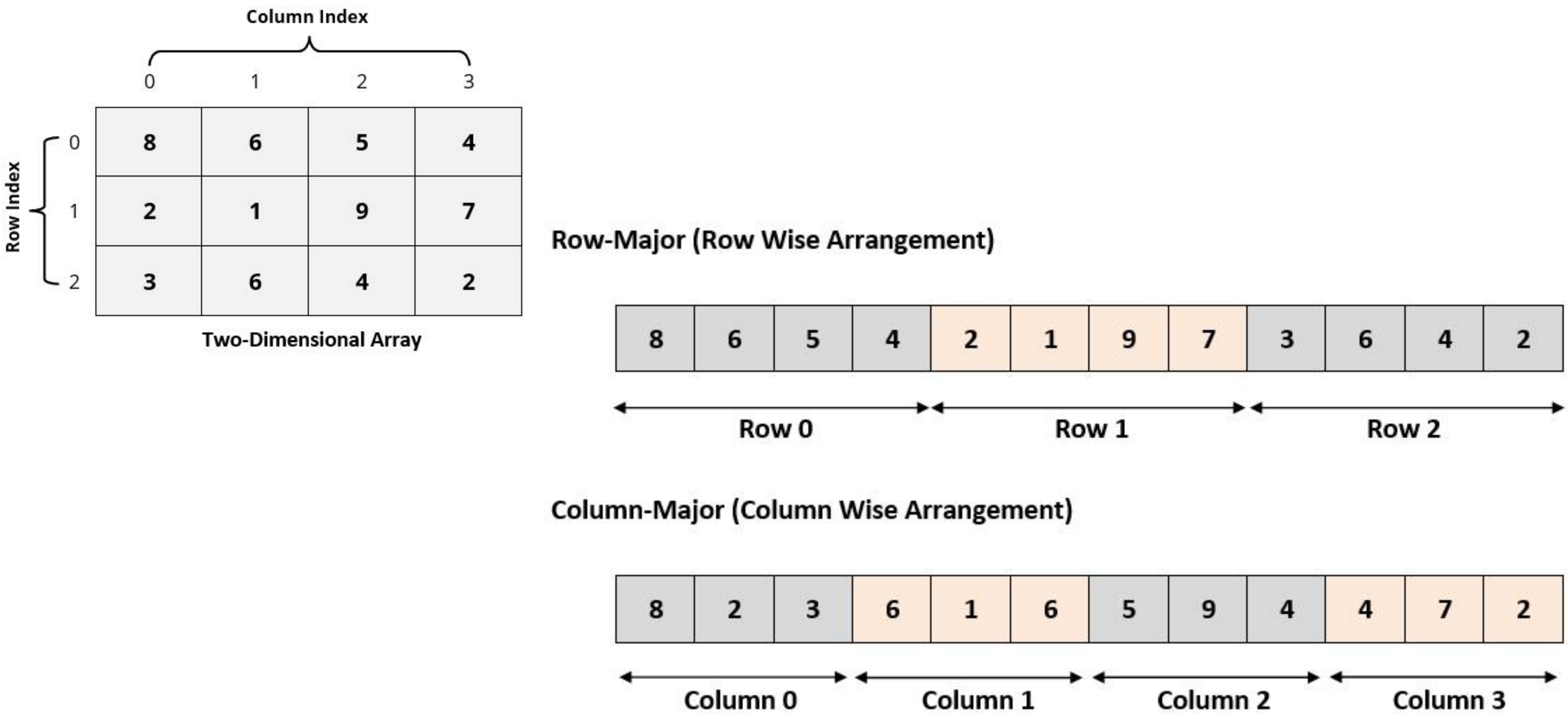


Column major order:

In this method elements of an array are arranged sequentially column by column.



Implementation of 2D Array



Memory Address Calculation

1. Row Major Order for a[m][n] or a[0…m-1][0…n-1]

Address of a[I, J] element =

$$\mathbf{B + w (N_c (I - L_r) + (J - L_c))}$$

2. Column Major Order- a[m][n] or a[0…m-1][0…n-1]

Address of a[I, J] element =

$$\mathbf{B + w (N_r (J - L_c) + (I - L_r))}$$

B = Base Address,
I = subscript (row),
J = subscript (column),
N_c = No. of column,
N_r = No. of rows,
L_r = row lower bound(0) ,
L_c = column lower bound (0),
U_c=column upper bound(n-1) ,
U_r =row upper bound (m-1),
w = element size.

Memory Address Calculation

1. Row Major Order for a[m][n] or a[0…m-1][0…n-1]

Address of a[I, J] element =

$$\mathbf{B + w\,(\,N_c\,(\,I - L_r\,) + (\,J - L_c\,))}$$

2. Column Major Order- a[m][n] or a[0…m-1][0…n-1]

Address of a[I, J] element =

$$\mathbf{B + w\,(\,N_r\,(\,J - L_c\,) + (\,I - L_r\,))}$$

Suppose we want to calculate the address of element A [1, 2].

2000	A[0][0]	Row 0
2002	A[0][1]	
2004	A[0][2]	
2006	A[0][3]	
2008	A[1][0]	Row 1
2010	A[1][1]	
2012	A[1][2]	
2014	A[1][3]	

Memory Address Calculation

1. Row Major Order for a[m][n] or a[0…m-1][0…n-1]

Address of a[I, J] element =

$$\mathbf{B + w (N_c (I – L_r) + (J – L_c))}$$

2. Column Major Order- a[m][n] or a[0…m-1][0…n-1]

Address of a[I, J] element =

$$\mathbf{B + w (N_r (J – L_c) + (I – L_r))}$$

- ✓ An array S[10][15] is stored in the memory with each element requiring 4 bytes of storage. If the base address of S is 1000, determine the location of S[8][9] when the array is S stored by
- ✓ (i) Row major (ii) Column major.

ANSWER

- ✓ Let us assume that the Base index number is [0][0].
- ✓ Number of Rows = R = 10
- ✓ Number of Columns = C = 15
  Size of data = W = 4
- Base address = B = S[0][0] = 1000 Location of S[8][9] = X

Memory Address Calculation

1. Row Major Order for a[m][n] or a[0…m-1][0…n-1]

Address of a[I, J] element =

$$B + w (N_c (I - L_r) + (J - L_c))$$

2. Column Major Order- a[m][n] or a[0…m-1][0…n-1]

Address of a[I, J] element =

$$B + w (N_r (J - L_c) + (I - L_r))$$

(i) When S is stored by Row Major

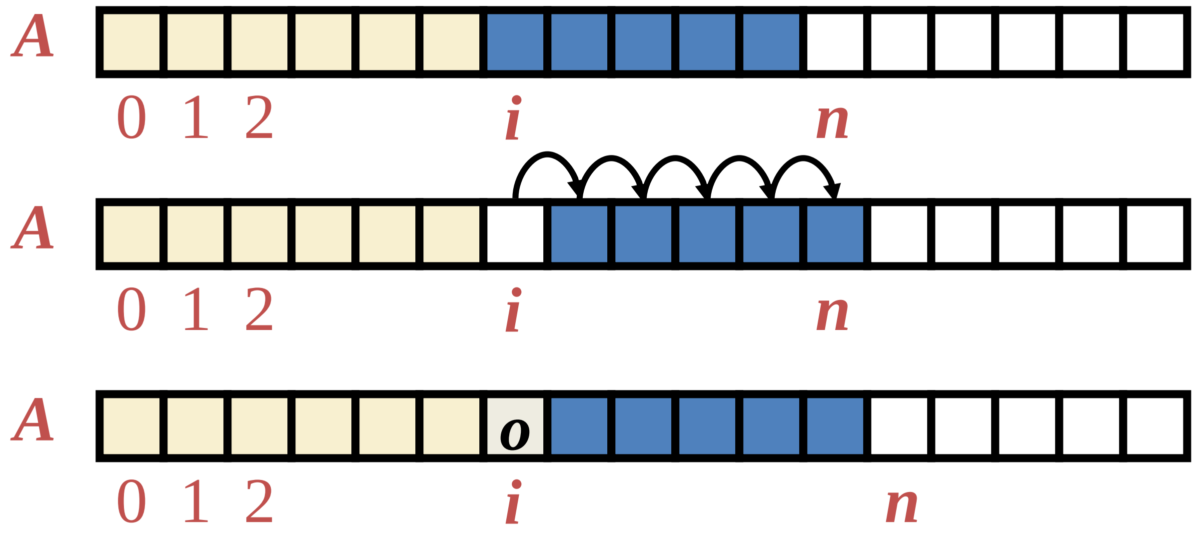
$$\begin{aligned} \mathbf{X} &= \mathbf{B + W * [8 * C + 9]} \\ &= 1000 + 4 * [8 * 15 + 9] \\ &= 1000 + 4 * [120 + 9] \\ &= 1000 + 4 * 129 \\ &= 1516 \end{aligned}$$

(ii) When S is stored by Column Major

$$\begin{aligned} \mathbf{X} &= \mathbf{B + W * [8 + 9 * R]} \\ &= 1000 + 4 * [8 + 9 * 10] \\ &= 1000 + 4 * [8 + 90] \\ &= 1000 + 4 * 98 \\ &= 1000 + 392 \\ &= 1392 \end{aligned}$$

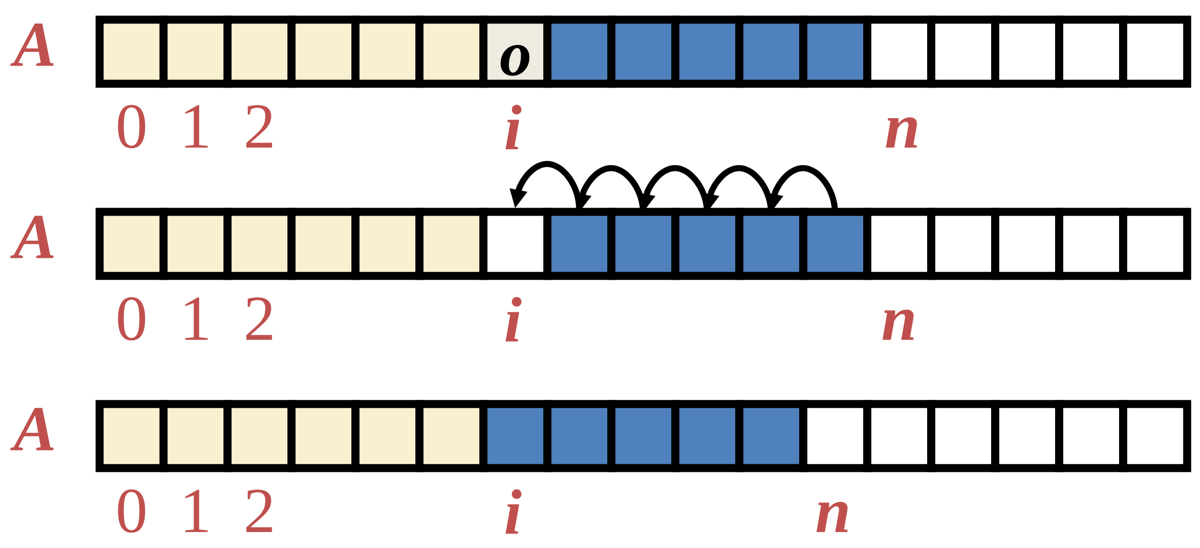
Insertion

- In an operation ***add***$(\boldsymbol{i}, \boldsymbol{o})$, we need to make room for the new element by shifting forward the $\boldsymbol{n} - \boldsymbol{i}$ elements $\boldsymbol{A}[\boldsymbol{i}], \ldots, \boldsymbol{A}[\boldsymbol{n} - 1]$
- In the worst case ($\boldsymbol{i} = 0$), this takes $\boldsymbol{O}(\boldsymbol{n})$ time



Element Removal

- In an operation ***remove***(i), we need to fill the hole left by the removed element by shifting backward the $\boldsymbol{n}$ - $\boldsymbol{i}$ - 1 elements $\boldsymbol{A}[\boldsymbol{i} + 1]$, …, $\boldsymbol{A}[\boldsymbol{n} - 1]$
- In the worst case ($\boldsymbol{i} = 0$), this takes $\boldsymbol{O}(\boldsymbol{n})$ time



Performance

- In an array based implementation of a dynamic list:
  - The space used by the data structure is $O(n)$
  - Indexing the element at I takes $O(1)$ time
  - ***add*** and ***remove*** run in $O(n)$ time in worst case
- In an ***add*** operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one…

Growable Array-based Array List

- In an add(o) operation (without an index), we could always add at the end.
- When the array is full, we replace the array with a larger one.
- How large should the new array be?
  - Incremental strategy: increase the size by a constant ***c***
  - Doubling strategy: double the size

Algorithm ***add***(*o*)
  if $t = S.length - 1$ **then**
    $A \leftarrow$ **new array of size …**
    for $i \leftarrow 0$ **to** $n-1$ **do**
      $A[i] \leftarrow S[i]$
    $S \leftarrow A$
  $n \leftarrow n + 1$
  $S[n-1] \leftarrow o$

Comparison of the Strategies

- We compare the incremental strategy and the doubling strategy by analyzing the total time ***T***(***n***) needed to perform a series of ***n*** add(o) operations.
- We assume that we start with an empty stack represented by an array of size 1.
- We call amortized time of an add operation the average time taken by an add over the series of operations, i.e., ***T***(***n***)/***n*.**

Incremental Strategy Analysis

- We replace the array $\boldsymbol{k} = \boldsymbol{n}/\boldsymbol{c}$ times
- The total time $\boldsymbol{T}(\boldsymbol{n})$ of a series of $\boldsymbol{n}$ add operations is proportional to
  - $\boldsymbol{n} + \boldsymbol{c} + 2\boldsymbol{c} + 3\boldsymbol{c} + 4\boldsymbol{c} + \ldots + \boldsymbol{kc} =$
  - $\boldsymbol{n} + \boldsymbol{c}(1 + 2 + 3 + \ldots + \boldsymbol{k}) =$
  - $\boldsymbol{n} + \boldsymbol{ck}(\boldsymbol{k} + 1)/2$
- Since $\boldsymbol{c}$ is a constant, $\boldsymbol{T}(\boldsymbol{n})$ is $\boldsymbol{O}(\boldsymbol{n} + \boldsymbol{k}^2)$, i.e., $\boldsymbol{O}(\boldsymbol{n}^2)$
- The amortized time of an add operation is $\boldsymbol{O}(\boldsymbol{n})$

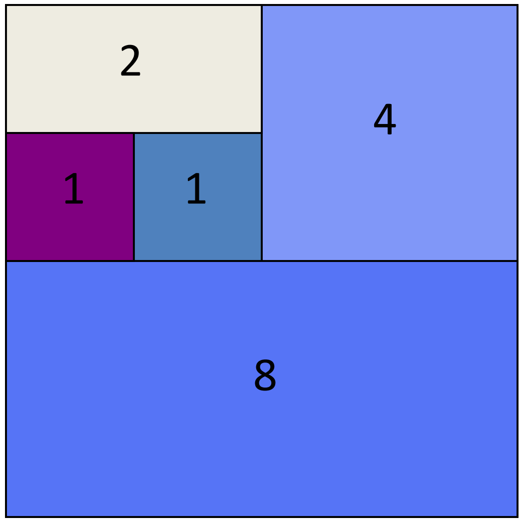
Doubling Strategy Analysis

- We replace the array $\boldsymbol{k} = \log_2 \boldsymbol{n}$ times
- The total time $\boldsymbol{T}(\boldsymbol{n})$ of a series of $\boldsymbol{n}$ add operations is proportional to

$$\boldsymbol{n} + 1 + 2 + 4 + 8 + \ldots + 2^{\boldsymbol{k}} =$$
$$\boldsymbol{n} + 2^{\boldsymbol{k}+1} - 1 =$$
$$3\boldsymbol{n} - 1$$

- $\boldsymbol{T}(\boldsymbol{n})$ is $\boldsymbol{O}(\boldsymbol{n})$
- The amortized time of an add operation is $\boldsymbol{O}(1)$

geometric series



Python Implementation

```
import ctypes                                      # provides low-level arrays

class DynamicArray:
  """A dynamic array class akin to a simplified Python list."""

  def __init__(self):
    """Create an empty array."""
    self._n = 0                                    # count actual elements
    self._capacity = 1                             # default array capacity
    self._A = self._make_array(self._capacity)     # low-level array

  def __len__(self):
    """Return number of elements stored in the array."""
    return self._n

  def __getitem__(self, k):
    """Return element at index k."""
    if not 0 <= k < self._n:
      raise IndexError('invalid index')
    return self._A[k]                              # retrieve from array

  def append(self, obj):
    """Add object to end of the array."""
    if self._n == self._capacity:                  # not enough room
      self._resize(2 * self._capacity)             # so double capacity
    self._A[self._n] = obj
    self._n += 1

  def _resize(self, c):                            # nonpublic utitity
    """Resize internal array to capacity c."""
    B = self._make_array(c)                        # new (bigger) array
    for k in range(self._n):                       # for each existing value
      B[k] = self._A[k]
    self._A = B                                    # use the bigger array
    self._capacity = c

  def _make_array(self, c):                        # nonpublic utitity
     """Return new array with capacity c."""
     return (c * ctypes.py_object)( )              # see ctypes documentation
```

THANKYOU

@csebennett cse_bennett