DATA STRUCTURES AND APPLICATIONS

Module 1_1

Introduction

Dr. Pushpalatha K
Associate Professor
Sahyadri College of Engineering & Management

Elementary Data Organization

- **Data**
  - values or sets of values
- **Data item**
  - Single unit of values
  - Group item
    - Data item that can be subdivided into sub items
      - Eg: name can be divided into firstname, middlename and lastname
  - Elementary item
    - Data item that can not be sub divided into sub items
      - Eg: Passport number, bank account number
- **Collections of data are organized into a hierarchy of fields, records and files**

Elementary Data Organization

- **Entity**
  - Has attributes with values
  - Values may be numeric or non-numeric
    - Eg: Students

attributes

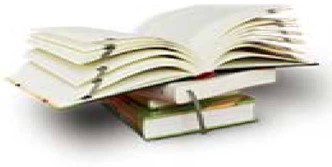
Name	Rollno	Sem	Branch
Aditya	1	5	CSE
Rama	15	4	ISE

Entity

Entity set

Values

- **Entity set**
  - Entities with similar attributes
  - Each attribute of an entity set has a range of values
- **Information**
  - Data with given attributes
  - Meaningful or processed data.



Elementary Data Organization

- **Field** is a single elementary unit of information representing an attribute of an entity
- **Record** is the collection of field values of a given entity
- **File** is the colJection of records of the entities in a given entity set.
- Each record in a file may contain many field items
  - The value in a certain field may uniquely determine the record in the file.
  - Such a field K is called a primary key
  - the values k1, k2,… of primary key are called keys or key values.
    - Eg: USN for student

Elementary Data Organization

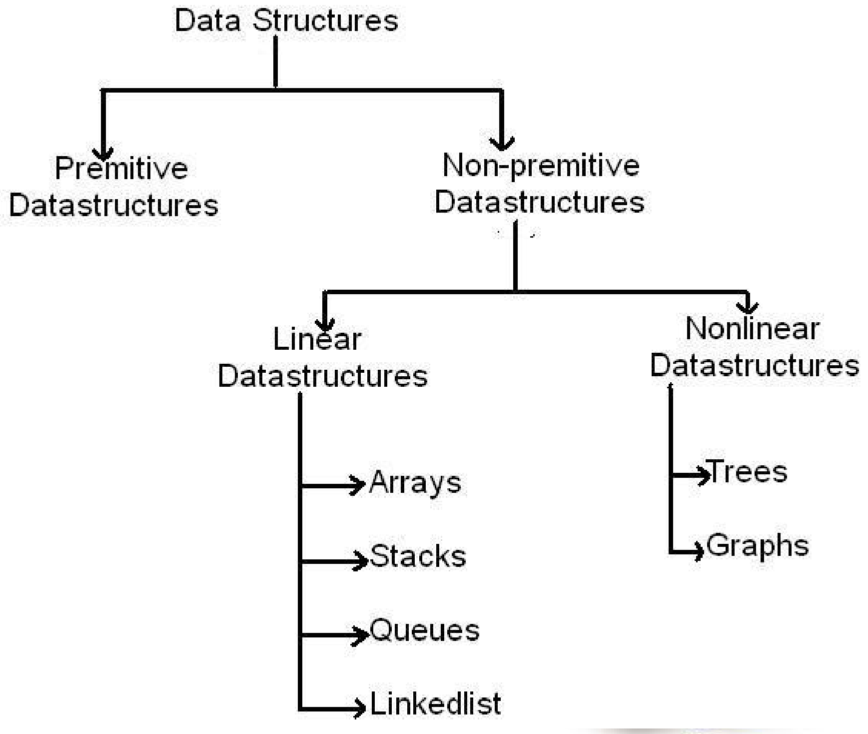
- A file can have fixed-length records or variable-length records.
  - In fixed length records, all the records contain the same data items with the same amount of space assigned to each data item.
  - In variable-length records, file records may contain different lengths
- Drawback of basic organization of data
  - May not maintain and process the data collections efficently
- Study of data structures requires
  - Logical or mathematical description of the structure
  - Implementation of the structure on a computer
  - Quantitative analysis of the structure which includes
    - determining the amount of memory needed to store the structure and
    - the time required to process the structure

DATA STRUCTURES

- The logical or mathematical model of a particular organization of data is called data structure.
- The choice of a data model depends on two factors
  - The structure must mirror the actual relationships of the data in the real world.
  - The structure should be simple enough to process the data effectively when necessary
- Classification of data structures
  - Primitive data structures
  - Non-primitive data structures

DATA STRUCTURES

- **Primitive data structure**
  - They are called simple data types which cannot be divided
    - Eg: basic data types such as integer, real. character and boolean
- **Non-primitive data structures**
  - Complicated data structures
  - Derived from primitive data structures.
    - Eg: Linked-lists, stacks. queues, trees and graphs

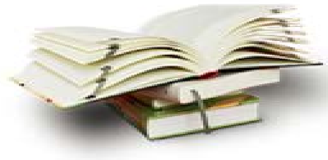


DATA STRUCTURES

- Based on the structure and arrangement of data, non-primitive data structures are classified into linear and non–linear data structures
- Linear data structure
  - The are linear and form a sequence.
  - The data is arranged in linear fashion although the data are stored in memory nonsequentially.
    - Eg: arrays. linked lists, stacks and queues
- Non-Linear data structure
  - Data is not arranged in sequence.
  - The insertion and deletion of data is not possible in lioear fashion.
    - Eg: Trees and graphs

Array

- **Linear (or one dimensional) array**
  - The simplest type of data structure
  - A list of a finite number n of similar data referenced respectively by a set of n consecutive numbers, usually 1, 2, 3 . . . . n.
  - For an array A, the elements of A are denoted by
    - subscript notation $a_1, a_2, a_3 \ldots . A_n$
    - or by the parenthesis notation A(1), A(2), A(3) . . . . . . A(n)
    - or by the bracket notation A[1], A[2], A[3] . . . . . . A[n]
  - the number K in A[K] is called a subscript
  - A[K] is called a subscripted variable.



- **Eg: A linear array A[8] consisting of numbers**

2	4	6	8	10
A[0]	A[1]	A[2]	A[3]	A[4]

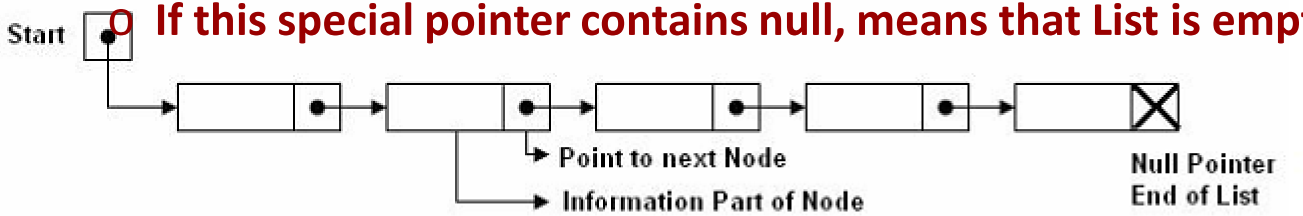
Linear/One Dimensional Array (1x5)

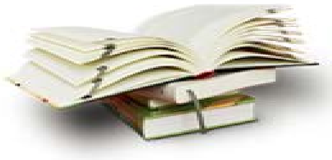
YEAR / DEPT	1	2	3	4
CSE	180	174	170	160
ISE	120	110	108	100

Two Dimensional Array (2x5)

Linked List

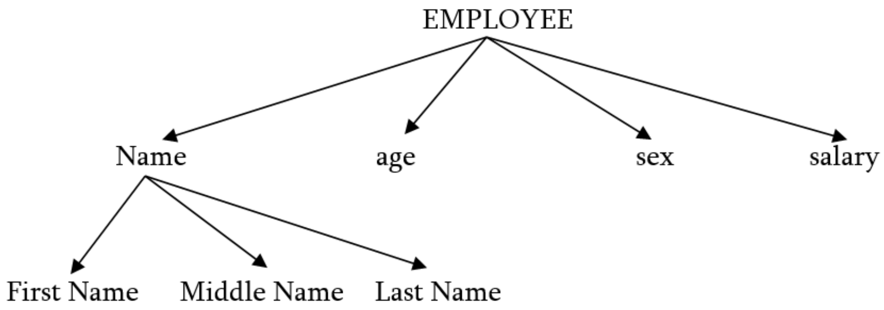
- **Linked list**
  - Is a linear collection of data elements, called nodes,
  - the linear order is given by means of pointers.
  - Each node is divided into two parts:
    - The first part contains the information of the element/node
    - The second part contains the address of the next node (link /next pointer field) in the list.
    - There is a special pointer Start/List contains the address of first node in the list.
      - If this special pointer contains null, means that List is empty.

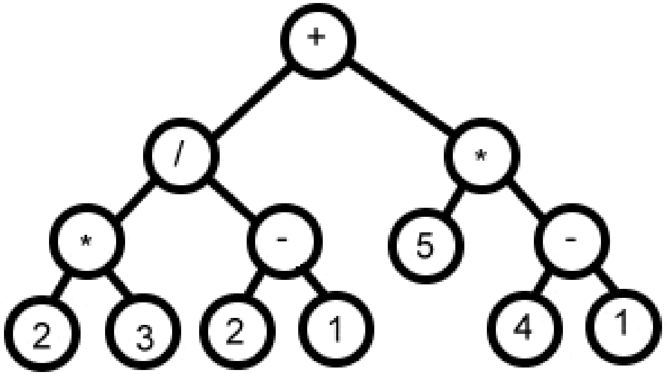




Trees

- **The data structure which defines the hierarchical relationship is called a rooted tree graph or tree.**

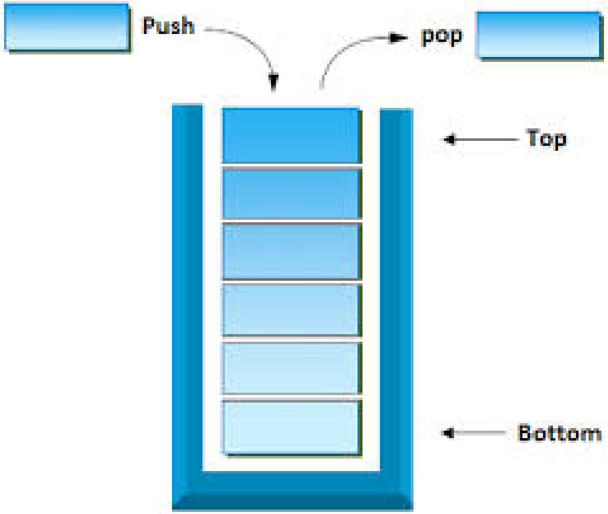




Expression tree for 2*3/(2-1)+5*(4-1)

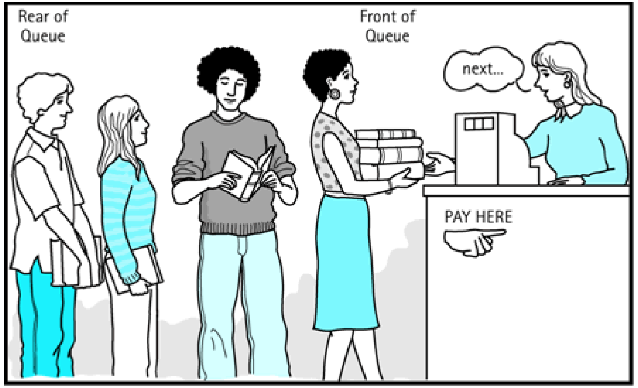
Stack

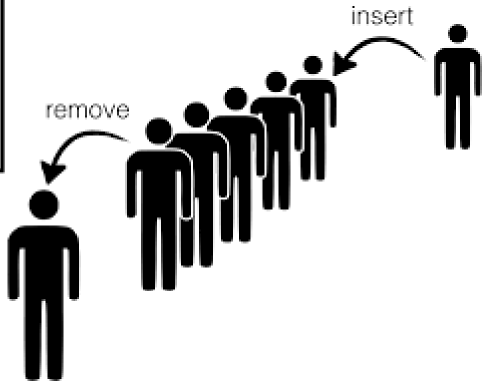
- A stack is also an ordered collection of elements like arrays
- It has a special feature that deletion and insertion of elements can be done only from one end called the top of the stack (TOP)
- It is also called as last in first out type of data structure (LIFO).

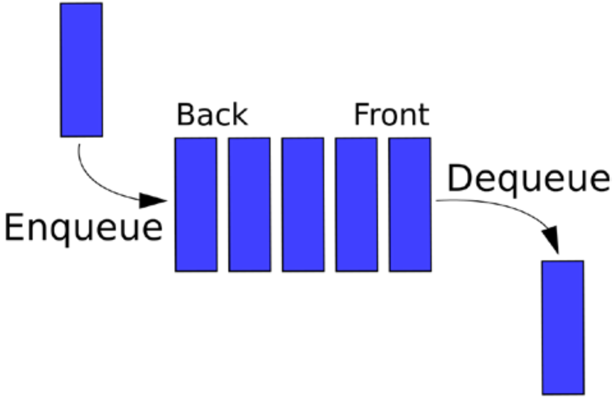


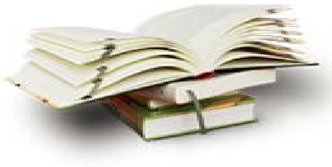
Queue

- Queue are first in first out type of data structure (i.e. FIFO)
- In a queue new elements are added to the queue from one end called REAR end and the element are always removed from other end called the FRONT end.



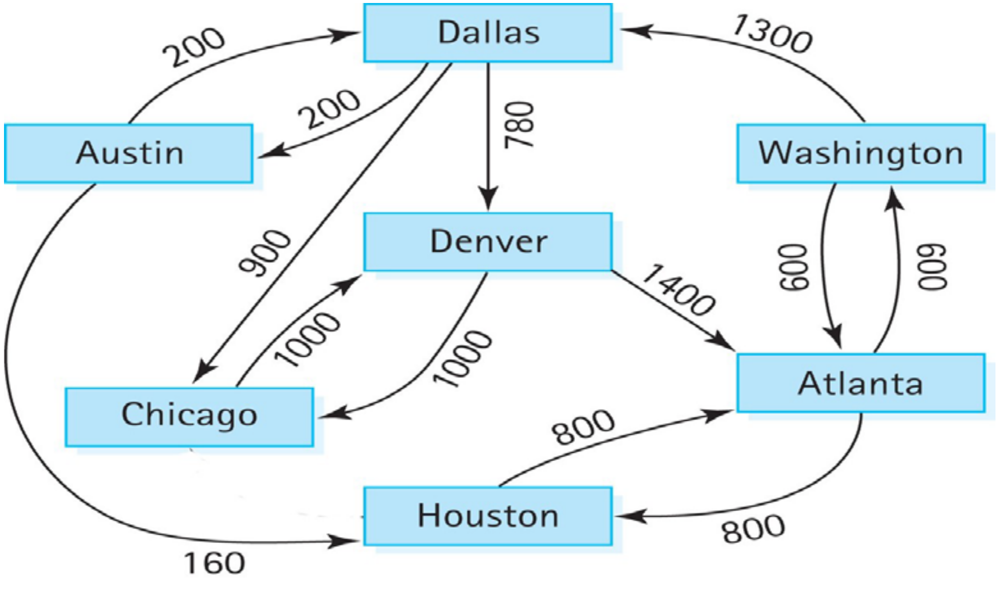


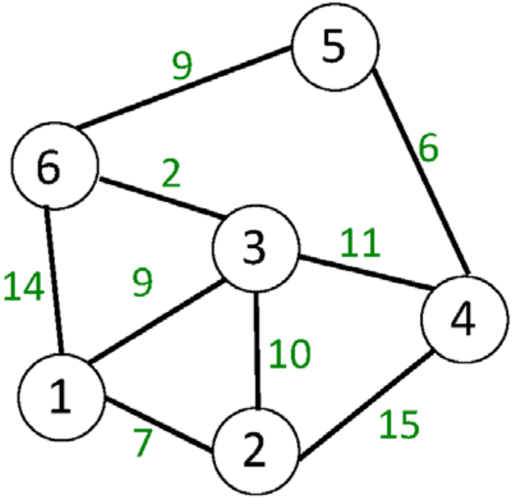




Graph

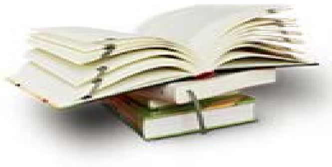
- Represnets the relationship between the data elements
- Contains nodes and edges



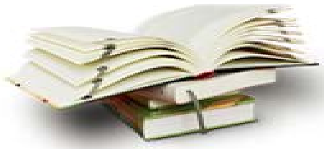


DATA STRUCTURE OPERATIONS

- **The data in data structures are processed using four types of operations**
  - **Traversing:**
    - **accessing each record/node exactly once so that certain items in the record may be processed.**
      - **This accessing and processing is sometimes called “visiting” the record**
  - **Searching:**
    - **Finding the location of the desired node with a given key value,**
    - **finding the locations of nodes which satisfy one or more conditions.**
  - **Inserting:**
    - **Adding a new node/record to the structure.**
  - **Deleting:**
    - **Removing a node/record from the structure.**

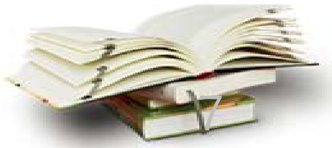


- **The following two operations, which are used in special situations**
  - **Sorting:**
    - **Arranging the records in some logical order**
  - **Merging**
    - **Combining the records in two different sorted files into a single sorted file**
- **Other supported operations are**
  - **Copying**
  - **concatenation**



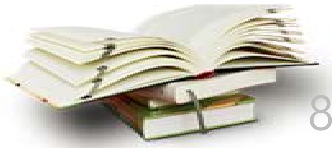
Linear Arrays

- A linear array is a list of a finite number of n homogeneous data elements (that is data elements of the same type) such that
  - The elements of the arrays are referenced respectively by an index set consisting of n consecutive numbers
  - The elements of the arrays are stored respectively in successive memory locations



Linear Arrays

- **The number n of elements is called the length or size of the array.**
  - The index set consists of the integer 1, 2, … n
  - Length or the number of data elements of the array can be obtained from the index set by
  - Length = UB – LB + 1
    - where UB is the largest index called the upper bound
    - LB is the smallest index called the lower bound of the arrays



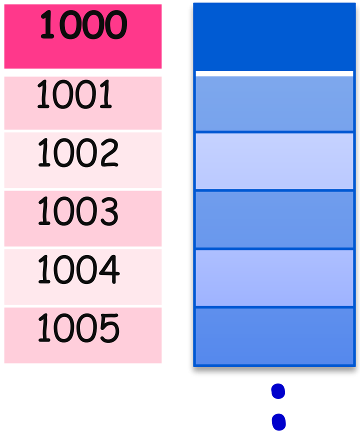
Linear Arrays

- Element of an array A may be denoted by
  - Subscript notation $A_1, A_2, , \dots , A_n$
  - Parenthesis notation A(1), A(2), …. , A(n)
  - Bracket notation A[1], A[2], ….. , A[n]

- The number K in A[K] is called **subscript** or an **index**
- A[K] is called a **subscripted variable**

Representation of Linear Array in Memory

- **Let LA be a linear array in the memory of the computer**
  - **LOC(LA[K]) = address of the element LA[K] of the array LA**
- **The element of LA are stored in the successive memory cells**
  - **Computer does not need to keep track of the address of every element of LA,**
  - **but need to track only the address of the first element of the array denoted by Base(LA) called the base address of LA**



Computer Memory

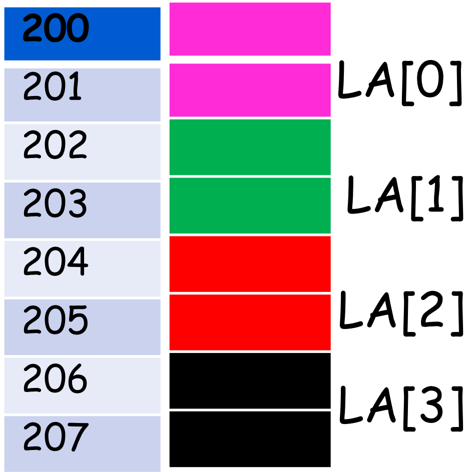
- $LOC(LA[K]) = Base(LA) + w(K - LB)$
  - where **w** is the number of words per memory cell of the array LA [w is the size of the data type]
- Eg: Find the address for LA[6]
  - Each element of the array occupy 1 byte
  - $LOC(LA[K]) = Base(LA)+w(K- LB)$
  - $LOC(LA[6]) = 200 + 1(6 - 0)$
    $= 206$

200		LA[0]
201		LA[1]
202		LA[2]
203		LA[3]
204		LA[4]
205		LA[5]
206		LA[6]
207		LA[7]
	:	

Example

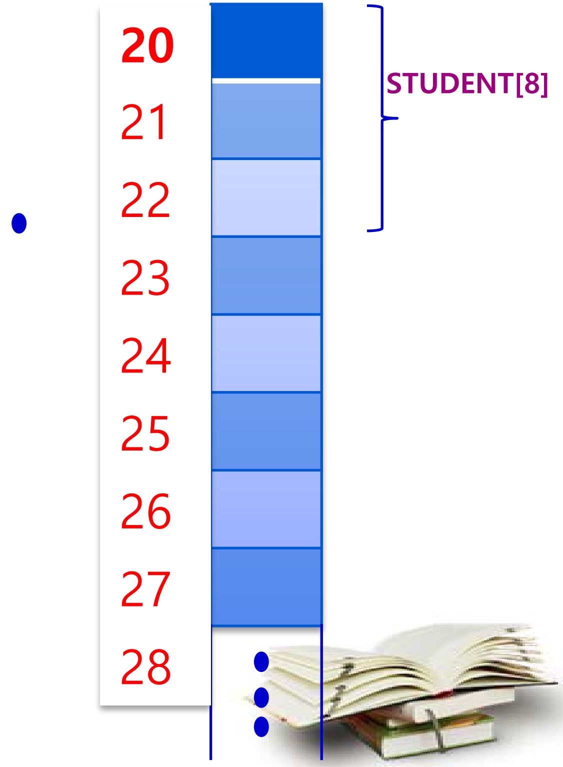
- Find the address for LA[15] Each element of the array occupy 2 bytes

  – LOC(LA[K])=Base(LA)+w(K-LB)
  – LOC(LA[15]) = 200 + 2(15 – 0) = 230



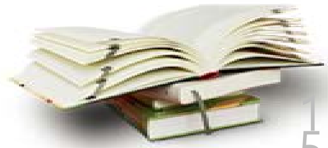
Example

- An array STUDENT stores the marks of students from rollno 8 to 33.
- Base(STUDENT)=20
- Let w=3
  - LOC(STUDENT[8])=20,
  - LOC(STUDENT[9])=23
  - LOC(STUDENT[10])=26
- Address of array element for rollno K=15 is
  - LOC(STUDENT[15])=Base(STUDENT)+w(15-LB)
  - =20+3(15-8)=41;



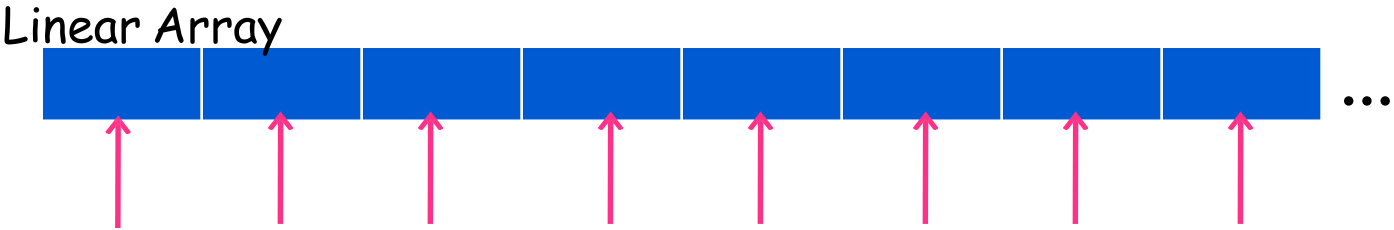
Representation of Linear Array in Memory

- Given any value of K, time to calculate LOC(LA[K]) is same
- Given any subscript K one can access and locate the content of LA[K] without scanning any other element of LA
- A collection A of data element is said to be indexed if any element of A called A_k can be located and processed in time that is independent of K



Traversing Linear Arrays

- **Traversing is accessing and processing each element of the data structure exactly once.**

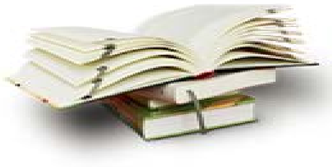


Algorithm: (Traversing a linear array)

- A is a linear array with lower bound LB and upper bound UB.
- The algorithm traverses A applying an operation PROCESS to each element of A.

1. Set I:= LB [Initialize counter]
2. Repeat steps 3 and 4 while I<= UB
3. Apply PROCESS to A. [Visit element.]
4. I:= I+1. [Increase counter.]

   [End of step 2 loop.]
5. Exit.



- **Alternate array traversal algorithm using repeat-for-loop**

1. Repeat for K = LB to UB:

   Apply PROCESS to LA[K]

   [End of Loop]

2. Exit

Inserting and Deleting

- **Insertion:**
  - Adding an element
    - Beginning
    - Middle
    - End
- **Deletion:**
  - Removing an element
    - Beginning
    - Middle
    - End

Insertion

1	**Aakash**
2	Samad
3	Abhishek
4	Adhiti
5	Akshay
6	
7	
8	

1	**Aakash**
2	Samad
3	Abhishek
4	Adhiti
5	Akshay
6	Akshita
7	
8	

Insert Akshita at the End of Array

Insertion

Insert Ajay as the 3rd Element of Array

1	**Aakash**	1	**Aakash**	1	**Aakash**	1	**Aakash**
2	Samad	2	Samad	2	Samad	2	Samad
3	Abhishek	3	Abhishek	3	Abhishek	3	Ajay
4	Adhiti	4	Adhiti	4		4	Abhishek
5	Akshay	5		5	Adhiti	5	Adhiti
6		6	Akshay	6	Akshay	6	Akshay
7		7		7		7	
8		8		8		8	

Insertion is not Possible without loss of data if the array is FULL

Insertion Algorithm

INSERT (LA, N , K , ITEM)

[LA is a linear array with N elements and K is a positive integers such that $K \le N$. This algorithm inserts an element ITEM into the K^{th} position in LA]

1. [Initialize Counter] Set J := N
2. Repeat Steps 3 and 4 while $J \ge K$
3. [Move the J^{th} element downward] Set LA[J + 1]:= LA[J]
4. [Decrease Counter] Set J := J -1
    1. [End of Step 2 loop]
5. [Insert Element] Set LA[K] := ITEM
6. [Reset N] Set N := N +1;
7. Exit

Deletion

1	**Aakash**	1	**Aakash**
2	Samad	2	Samad
3	Ajay	3	Ajay
4	Abhishek	4	Abhishek
5	Adhiti	5	Adhiti
6	Akshay	6	Akshay
7	Akshita	7	
8		8	

Deletion of Akshita at the End of Array

Deletion

1	**Aakash**	1	**Aakash**	1	**Aakash**	1	**Aakash**	1	**Aakash**
2	Samad	2		2	Ajay	2	Ajay	2	Ajay
3	Ajay	3	Ajay	3		3	Abhishek	3	Abhishek
4	Abhishek	4	Abhishek	4	Abhishek	4		4	Adhiti
5	Adhiti	5	Adhiti	5	Adhiti	5	Adhiti	5	Akshay
6	Akshay	6	Akshay	6	Akshay	6	Akshay	6	Akshita
7	Akshita	7	Akshita	7	Akshita	7	Akshita	7	
8		8		8		8		8	

Deletion of Samad from the Array

No data item can be deleted from an empty array

Deletion Algorithm

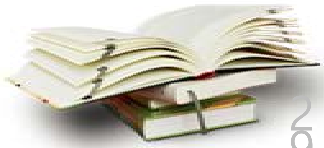
DELETE (LA, N , K , ITEM)

[LA is a linear array with N elements and K is a positive integers such that K ≤ N. This algorithm deletes K^{th} element from LA]

1. Set ITEM := LA[K]
2. Repeat for J = K to N-1:

   [Move the $J+1^{st}$ element upward] Set LA[J]:= LA[J+1]

   [End of loop]
3. [Reset the number N of elements] Set N := N - 1;
4. Exit

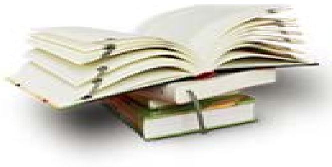


Sorting

- Sorting rearranges the elements of the array in increasing order or decreasing order
  - i.e. A[1]<A[2]<A[3]<…<A[n]
- Eg: if the elements of a list A are 6,2,7,9,3,12
  - After sorting the elements of A are 2,3,6,7,9,12 OR 12,9,7,6,3,2
- Bubble Sort
  - Very simple sorting algorithm

Bubble sort Algorithm explained

- Suppose the list of number A[1], [2], A[3], …, A[N] is in memory.
- Steps
  1. Compare A[1] and A[2], arrange them in the desired order so that A[1] < A[2]. Then Compare A[2] and A[3], arrange them in the desired order so that A[2] < A[3]. Continue until A[N-1] is compared with A[N], arrange them so that A[N-1] < A[N].
     - Involves n-1 comparisons
     - A[N] contain the largest element
  2. Repeat Step 1, Now stop after comparing and re-arranging A[N-2] and A[N-1].
     - Involves n-2 comparisons
     - A[N-1] contain the 2^{nd} largest element

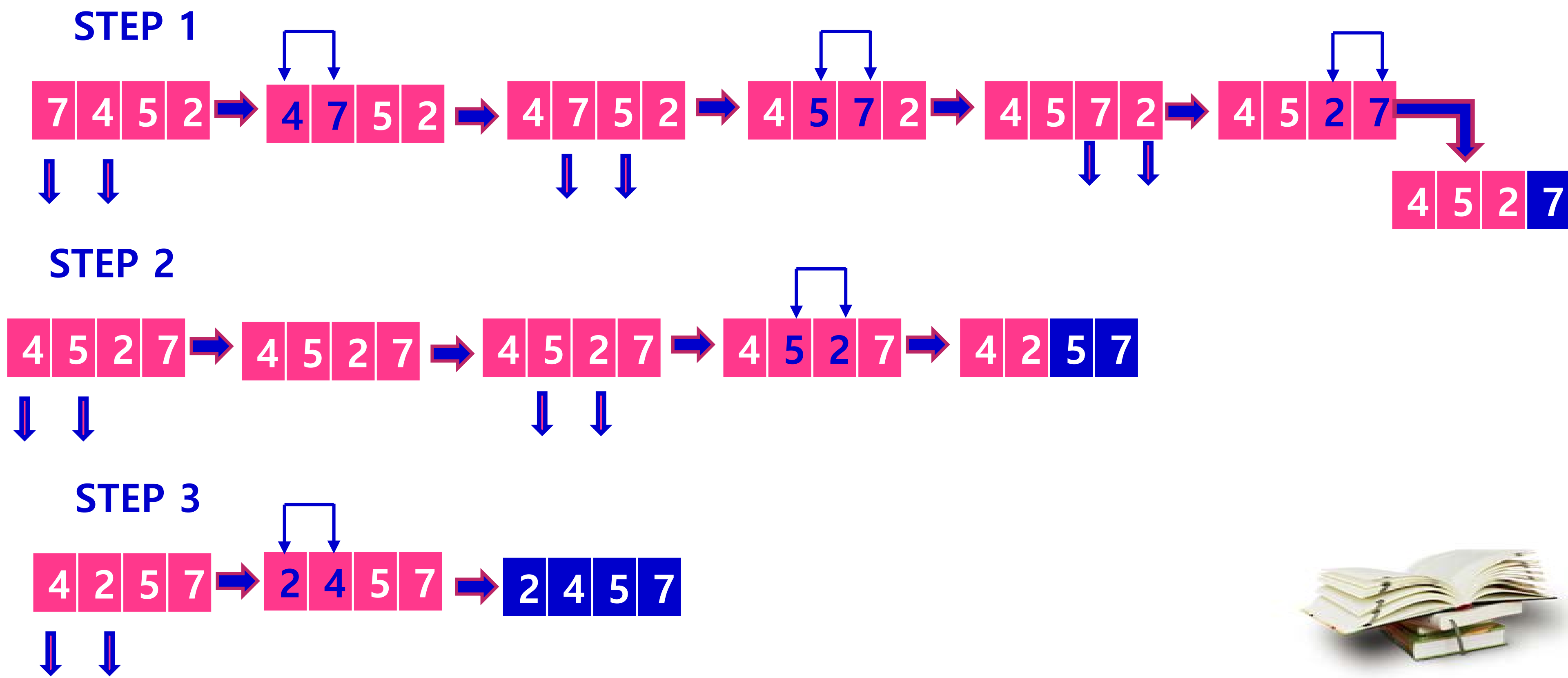


3. Repeat Step 3, Now stop after comparing and re-arranging A[N-3] and A[N-2].
4. . --------------------
5. .------------------------
6. .-----------------------

N-1. Compare A[1] and A[2] and arrange them in sorted order so that A[1] < A[2].

- After N-1 steps the list will be sorted in increasing order
- The process of sequentially traversing through all or part of a list is called pass
  - Requires n-1 passes for n number of elements

BUBBLE SORT


STEP 1
7 4 5 2
4 7 5 2
4 7 5 2
4 5 7 2
4 5 7 2
4 5 2 7
4 5 2 7
STEP 2
4 5 2 7
4 5 2 7
4 5 2 7
4 5 2 7
4 2 5 7
STEP 3
4 2 5 7
2 4 5 7
2 4 5 7

(Bubble sort) BUBBLE (DATA, N)

DATA is an array with N elements. This algorithm sorts the elements in DATA.

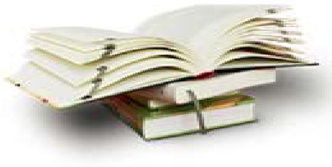
1. Repeat Steps 2 and 3 for K = 1 to N – 1.
2. Set PTR := 1. [Initialize pass pointer PTR.]
3. Repeat while PTR <= (N – K): [Executes pass.]
    a) If DATA[PTR] > DATA[PTR + 1], then:
        Interchange DATA[PTR] and DATA[PTR+1]
        [End of If structure.]
    b) Set PTR := PTR + 1.
        [End of inner loop.]
        [End of Step 1 outer loop.]
4. Exit

Complexity of Bubble sort algorithm

- The time for a sorting algorithm is measured in terms of the number of comparisons
- In bubble sort n-1 comparisons made in first pass, n-2 comparisons in second pass and so on
- The number of comparisons for bubble sort is

$$f(n) = (n-1) + (n-2) + \ldots + 2 + 1 = \frac{n(n-1)}{2} = \frac{n^2}{2} + O(n) = O(n^2)$$

- The time required to execute the bubble sort algorithm is propotional to n^2, where n is the number of input items



SELECTION SORT ALGORITHM

- The Selection sort algorithm is based on the idea of finding the minimum or maximum element in an unsorted array and then putting it in its correct position in a sorted array.
- Steps
  - First it finds the smallest element in the array.
  - Exchange that smallest element with the element at the first position.
  - Then find the second smallest element and exchange that element with the element at the second position.
  - This process continues until the complete array is sorted.

- Suppose A is an array which consists of n elements namely A[l], A[2], . . . , A[N]. The selection sort algorithm will works as follows.

  Pass 1: Find the location LOC of the smallest element in the list A[l], A[2], . . . , A[N] and put it in the first position.

  - Interchange A[LOC] and A[1].
  - Now, A[1] is sorted.

  Pass 2: Find the location of the second smallest element in the list A[2], A[3],. . ., A[N] and put it in the second position.

  - Interchange A[LOC] and A[2].
  - Now, A[1] and A[2] is sorted. Hence, A[l] ≤ A[2].

  Pass 3: Find the location of the third smallest element in the list A[3],A[4], . . . ,A[N] and put it in the third position.

  - Interchange A[LOC] and A[3].
  - Now, A[1], A[2] and A[3] is sorted. Hence, A[l] ≤ A[2] ≤ A[3].

  …. . . . .

Pass N-1. Find the location of the smallest element in the list A[A–1] and A[N].

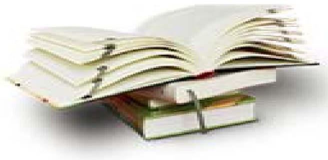
- Interchange A[LOC] and A[N–1] & put into the second last position.
- Now, A[1], A[2], …..,A[N] is sorted. Hence, A[l] ≤ … ≤ A[N–1] ≤A[N].

- Thus A is sorted after N-1 passes
- Algorithm requires a variable MIN to hold the current smallest value.
  - Initially set MIN:=A[K] and LOC=K
    - where K=starting location at each pass

Selection Sort

Pass	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
K=1 LOC=4	77	33	44	11	88	22	66	55
K=2 LOC=6	11	33	44	77	88	22	66	55
K=3 LOC=6	11	22	44	77	88	33	66	55
K=4 LOC=6	11	22	33	77	88	44	66	55
K=5 LOC=8	11	22	33	44	88	77	66	55
K=6 LOC=7	11	22	33	44	55	77	66	88
K=7 LOC=4	11	22	33	44	55	66	77	88

Sorted	11	22	33	44	55	66	77	88



MIN(A,K,N,LOC)

This procedure finds the location LOC of the smallest element among A[K],A[K+1],…,A[N]

1. Set MIN=A[K] and LOC=K [Initialize pointers]
2. Repeat for J=K+1,K+2…….N

   If MIN>A[J], then Set MIN=A[J] and LOC=J.

   [End of loop]
3. Exit

Selection Sort Algorithm

SELECTION(A,N)

This algorithm sorts the array A with N elements

1. Repeat Steps 2 and 3 for K=1,2,…,N-1:
2. Call MIN(A,K,N,LOC)
3. Interchange A[K] AND A[LOC]

   Set Temp:=A[K], A[K]:=A[LOC] and A[LOC]:=TEMP

   [End of Step 1 loop]
4. Exit

Complexity of the Selection sort algorithm

- The number f(n) of comparisons in the selection sort algorithm is independent of the original order of the elements
- MIN(A,K,N,LOC) requires n-K comparisons
- There are n-1 comparisons in pass 1, n-2 comparisons in pass 2 and so on.

$$f(n) = (n-1) + (n-2) + \cdots + 2 + 1 = \frac{n(n-1)}{2} = O(n^2)$$

- The number of interchanges and assignments depend on the original order of the elements of the array
  - Worst case :$O(n^2)$

SEARCHING

- Let DATA be a collection of data elements in memory
- ITEM is a data element
- Searching
  - Operation of finding the location LOC of ITEM in array DATA, or printing some message that ITEM does not appear in array.
- The search is said to be successful if ITEM does appear in DATA and unsuccessful otherwise.
- Many different searching algorithms
  - The algorithm is selected based on how the elements of DATA are organized
- The complexity of searching algo is measured in terms of number f(n) of comparison required to find ITEM in DATA, where DATA contains n elements.

Linear Search

- Let DATA is a linear array with n elements.
- Search ITEM in DATA.
- First test whether DATA[1]=ITEM,
  - Then check whether DATA[2]=ITEM and so on.
- The method which traverses DATA sequentially to locate item is called linear search or sequential search.

- **Method**
  - Insert **ITEM** to be searched at **DATA[n+1]** location, ie, at the last position of the array.
  - **LOC=N+1**
    - **LOC** denotes the location where **ITEM** first occurs in **DATA**.
    - Indicates the search was unsuccessful
- **The purpose of this initial assignment is to avoid checking that the end of the array is reached or not.**

Linear Search Algorithm

LINEAR (DATA, N, ITEM, LOC)

Here DATA is a linear array with N elements, and ITEM is a given item of information. This algorithm finds the location LOC of ITEM in DATA, or sets LOC:= 0 if the search is unsuccessful.

1. **[Insert ITEM at the end of DATA] Set DATA [N + 1]: = ITEM.**
2. **[Initialize counter] Set LOC: = 1**
3. **[Search for ITEM.]**

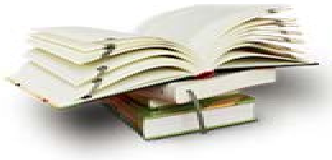
   Repeat while DATA [LOC] ≠ ITEM:

   Set LOC:= LOC + 1.

   [End of loop.]
4. **[Successful?] If LOC = N + 1, then: Set LOC:= 0**
5. **Exit**

COMPLEXITY OF LINEAR SEARCH

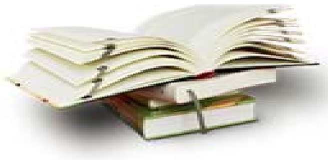
- Complexity is measured by the number f(n) of comparisons required to find ITEM in DATA where DATA contains n elements.
- Worst Case:
  - Worst case occurs the entire array DATA has been searched for ITEM, i.e., when ITEM does not appear in DATA.
  - The algorithm requires $f(n)=n+1$ comparisons.
  - the running time is proportional to n.
- Average Case:
  - Uses the probablitic notation
  - The average number of comparisons required to find the location of ITEM is approximately equal to half the number of elements in the array.



- p_k is the probability that ITEM appears in DATA[K]
- q is the probability that ITEM does not appears in DATA
  - Then $p_1 + p_2 + p_3 + \ldots + p_n + q = 1$
- The algorithm uses K comparisons when DATA[K]= ITEM
- The average number of comparisons is given by
  - $f(n) = 1.p_1 + 2.p_2 + 3.p_3 + \ldots + n.p_n + (n+1).q$
  - Let q is very smal i.e. $q \approx 0$, the probalitiy of ITEM appearing in each element of DATA is equal and $p_i = 1/n$, then

$$f(n) = 1.\frac{1}{n} + 2.\frac{1}{n} + \ldots + n.\frac{1}{n} + (n+1).0 = (1+2+\ldots+n).\frac{1}{n}$$

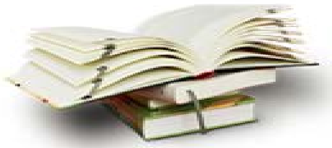
$$= \frac{n(n+1)}{2}.\frac{1}{n} = \frac{n+1}{2}$$



BINARY SEARCH

- Binary search is an efficient searching algorithm when data in array are ordered.
- It can be used to find the location LOC of a given ITEM of information in DATA
  - Eg: telephone directory
- Working of Algorithm
  - During each stage the search is reduced to a segment of DATA.
  - Let the array DATA is DATA[BEG], DATA[BEG+1], DATA[BEG+2],....., DATA[END].
    - BEG and an END are the beginning and end of array segment
  - Compare the ITEM with the middle element DATA[MID] of the segment,
    - where MID = int((BEG+END)/2).
    - int is used to get an integer value of MID.

- If DATA[MID]= ITEM, then search is successful, and LOC:=MID.
- Otherwise search is narrowed down to a new segment, which is obtained as:
  - If ITEM< DATA[MID], then ITEM appears in the left half of the segment: DATA[BEG], DATA[BEG+1],…, DATA[MID-1]
    - Reset END as END= MID-1, and begin search again.
  - If ITEM> DATA[MID], then ITEM appears in the right half of the segment: DATA[MID+1], DATA[MID+2],…, DATA[END]
    - Reset BEG as BEG= MID+1, and begin search again.
- Initially BEG=LB and END=UB.
- If ITEM is not in DATA, then
  - END<BEG and LOC:=null;



Binary Search Algorithm

BINARY (DATA, LB, UB, ITEM, LOC)

Here DATA is a sorted array with lower bound LB and upper bound UB, and ITEM is a given item of information. The variables BEG, END and MID denote, the beginning, end and middle locations of a segment of elements of DATA. This algorithm finds the location LOC of ITEM in DATA or sets LOC=NULL

1. [Initialize segment variables.]

   Set BEG: = LB, END := UB and MID = INT((BEG + END)/2).

2. Repeat Steps 3 and 4 while BEG ≤ END and DATA [MID] ≠ ITEM

Binary Search Algorithm

3. If ITEM < DATA [MID], then:

   Set END := MID - 1.

   Else:

   Set BEG := MID + 1.

   [End of If structure]

4. Set MID := INT((BEG + END)/2).

   [End of Step 2 loop.]

5. If DATA[MID] = ITEM, then:

   Set LOC := MID.

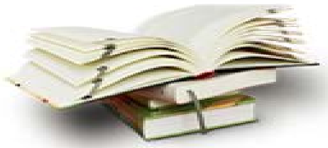
   Else:

   Set LOC := NULL.

   [End of If structure.]

6. Exit.

Note: When item doesnot appear in DATA, the algorithm arrives at a position where BEG=END=MID



Complexity of Binary Serach Algorithm

- The complexity is measured by the number $f(n)$ of comparisons to locate ITEM in DATA where DATA contains n elements.
- Here each comparison reduces the sample size in half.
- Hence we require at most $f(n)$ comparisons to locate ITEM where
  - $2^{f(n)} > n$ OR $f(n) = [\log_2 n + 1]$
- worst case:
  - the running time is approximately equal to $\log_2 n$.
- Average case:
  - the running time is equal to that of worst case.
- Limitations:
  - Requires array to be sorted.
  - Sorting array is quite expensive when new elements are inserted and deleted frequently.

MULTIDIMENSIONAL ARRAY

- **Linear array are known as one-dimensional arrays**
  - Each element is referenced by a single subscript
- **Multidimensional arrays**
  - Referenced by more than one subscript
  - Eg: two dimensional arrays, three-dimensional arrays
- **Two dimensional array**
  - A two-dimensional $m \times n$ array A is a collection of $m \cdot n$ data elements such that each element is specified by a pair of integers (such as J, K), called subscripts, with the property that
    - $1 \le J \le m$ and $1 \le K \le n$

- The element of A with first subscript j and second subscript k will be denoted by
  - $A_{J,K}$ or A[J, K]
- Two-dimensional arrays are called matrices in mathematics and tables in business applications.
- Standard way of drawing a two-dimensional m x n array A
  - the elements of A form a rectangular array with m rows and n columns
  - the element A[J, K] appears in row J and column K.

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Two-dimensional 3x4 Array

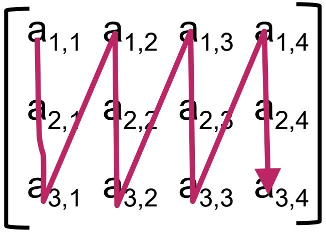
Representation of Two-Dimensional Arrays in Memory

- Let A be a two-dimensional m x n array
- the array will be represented in memory by a block of m . n sequential memory locations.
- The programming language will store the array A
  - column by column, is called column-major order, or
  - row by row, in row-major order

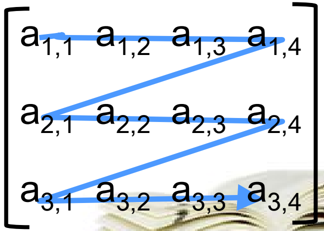
(1,1)	
(2,1)	Column1
(3,1)	
(1,2)	
(2,2)	Column2
(3,2)	
(1,3)	
(2,3)	Column3
(3,3)	
(1,4)	
(2,4)	Column4
(3,4)	

(1,1)	
(1,2)	Row1
(1,3)	
(1,4)	
(2,1)	Row2
(2,2)	
(2,3)	
(2,4)	
(3,1)	Row3
(3,2)	
(3,3)	
(3,4)	

Column-major order



Row-major order



- The computer uses the formula to find the address of LA[K] in time independent of K.
- For linear array
  - LOC (LA[K]) = Base(LA) + w(K - LB)
    - W is number of words per memory cell
    - LB is the lower bound of the index set of LA
- For two-dimensional array
  - The computer keeps track of Base(A) i.e the address of the first element A[1, 1] of A
  - Computes the address LOC(A[J, K]) of A[J,K]

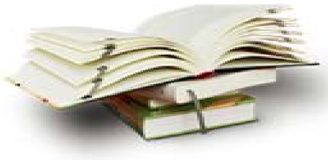
Column-major order

LOC(A[J, K]) = *Base(A)* + w[M(K - LB) + (J - LB)]

Row-major order

LOC(A[J, K]) = *Base(A)* + w[N(J - LB) + (K - LB)]

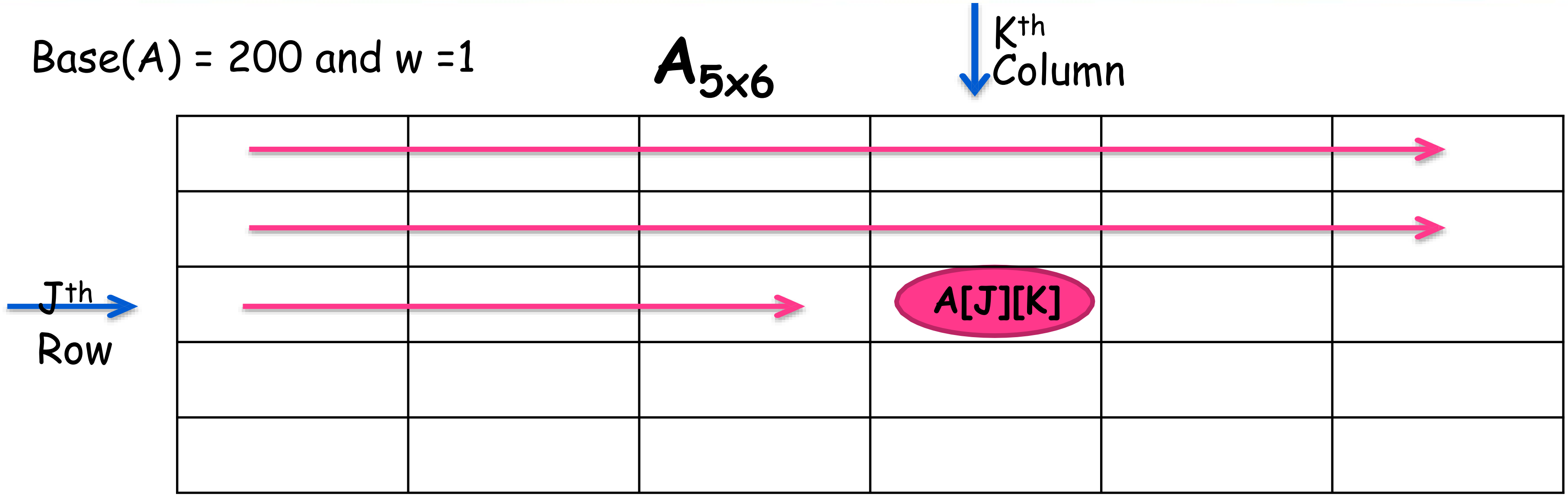
- W is number of words per memory cell
- LB is the lower bound of the index set of LA
  - Generally LB=1



2D Array (Row Major)

Base(A) = 200 and w =1

A_{5x6}



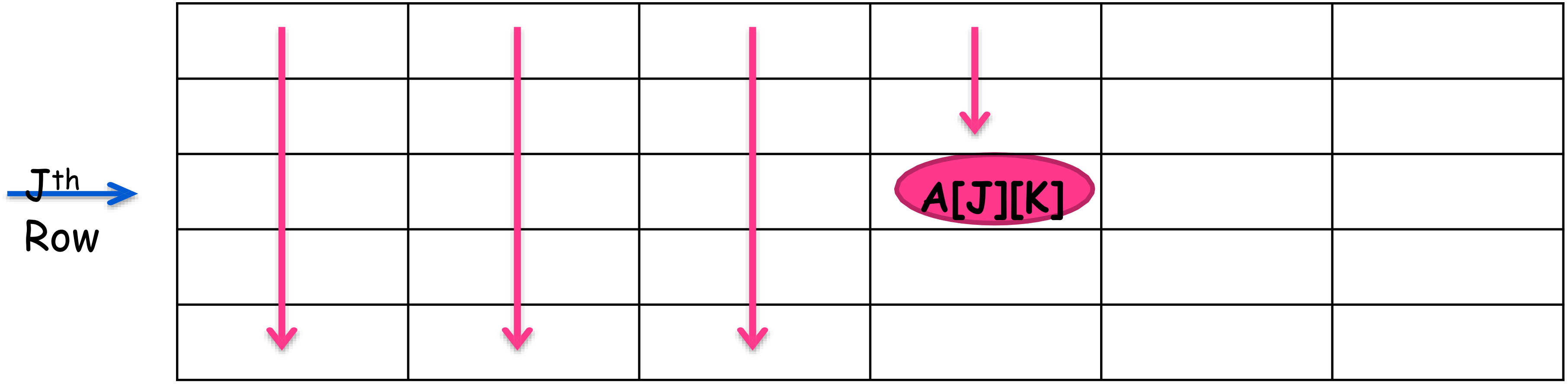
LOC(A[J,K]) = Base(A) + w[n(J-LB) + (K-LB)]

A[J][K]=200+1[6(3-1)+(4-1)]=200+15=215

2D Array (Column Major)

Base(A) = 200 and w =1 A_{5x6}

K^{th} Column



LOC(A[J,K]) = Base(A) + w[m(K-LB) + (J-LB)]

A[J][K]=200+1[5(4-1)+(3-1)]=200+17=217

General Multidimensional Arrays

- An n-dimensional m_1 x m_2 x …. x m_n array B is a collection of $m_1.m_2…m_n$ data elements
- Each element is specified by a list of n integers indices – such as K_1, K_2, …., K_n
  - These are called subscripts with the property that
    - $1 \le K_1 \le m_1$, $1 \le K_2 \le m_2$, …. $1 \le K_n \le m_n$
- The Element B with subscript K_1, K_2, …,K_n will be denoted by
  - $B_{K_1,K_2,…,K_n}$ or $B[K_1,K_2,….,K_n]$
- The array will be stored in memory as sequence of memory locations
  - The programming language will store the array B either in row-major order or in column-major order.

- Let C be a n-dimensional array
- The index set for each dimension of C consists of the consecutive integers from the lower bound to the upper bound of the dimension.
  - General multidimensional array mayhave lower bound <1
- Length L_i of dimension i of C is the number of elements in the index set
  - $L_i = UB_i – LB_i + 1$
- For a given subscript K_i, the effective index E_i of K_i is the number of indices preceding K_i in the index set
  - $E_i = K_i – LB_i$

- **Address LOC(C[$K_1, K_2, \ldots, K_n$]) of an arbitrary element of C can be obtained as**
  - **Column-Major Order**

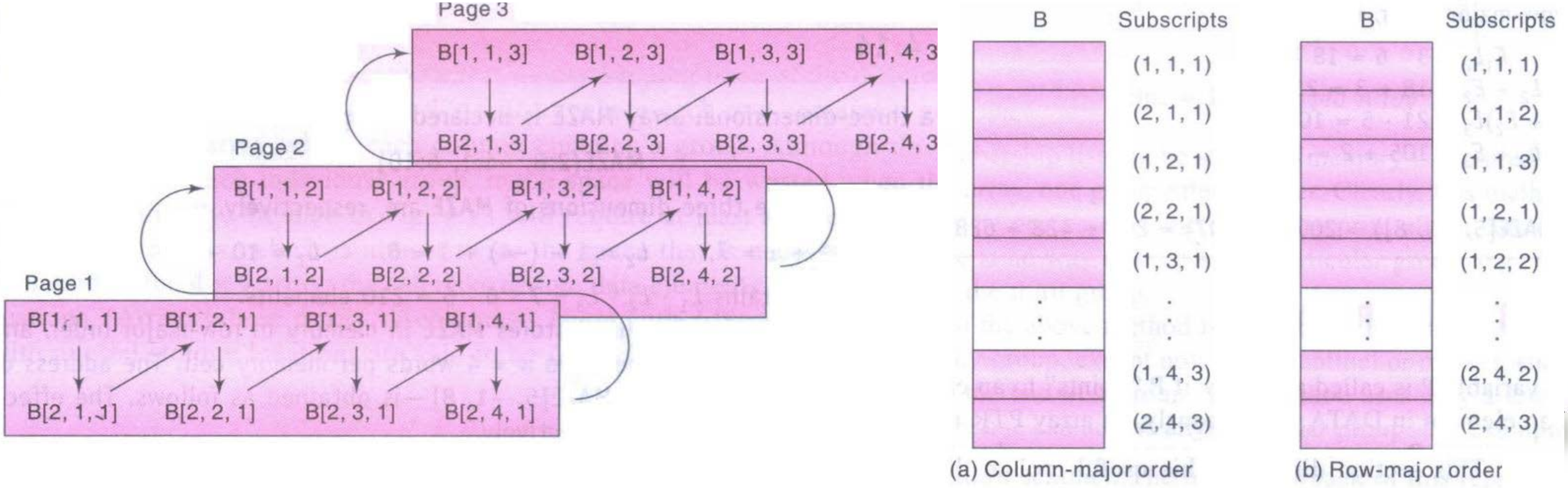
$$\text{Base}(C) + w[(((\ldots(E_N L_{N-1} + E_{N-1})L_{N-2}) + \ldots + E_3)L_2 + E_2)L_1 + E_1]$$

  - **Row-Major Order**

$$\text{Base}(C) + w[(\ldots((E_1 L_2 + E_2)L_3 + E_3)L_4 + \ldots + E_{N-1})L_N + E_N]$$

Column major order

- **Eg: Array B[2x4x3]**
  - Contains 24 elements
  - Displayed in 3 layers known as pages
    - Each layer contains 2x4 array of elements with same 3rd subscript
    - Subscripts of 3-dimesional array are row,column, and page



Eg: MAZE(2:8, -4:1, 6:10), Calculate the address of MAZE[5,-1,8]

– Given: Base(MAZE) = 200, w = 2,

– MAZE is stored in Row-Major order

$L_i = UB_i - LB_i + 1$

$E_i = K_i - LB_i$

➢ $L_1 = 8-2+1 = 7$, $L_2 = 1-(-4)+1 = 6$, $L_3 = 5$

– MAZE contains 7.6.5=210 elements

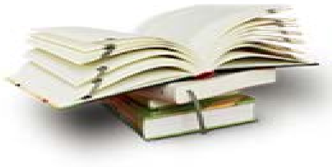
➢ $E_1 = 5 - 2 = 3$, $E_2 = -1-(-4) = 3$, $E_3 = 2$

$LOC(C[K_1, K_2, \ldots, K_n]) = Base(C) + w[(\ldots((E_1L_2 + E_2)L_3 + E_3)L_4 + \ldots + E_{N-1})L_N + E_N]$

$MAZE[5,-1,8] = Base(C) + w[(E_1L_2 + E_2)L_3 + E_3]$

$= 200+2[(3.6+3).5+2] = 200+2[(21).5+2]$

$= 200+2[107] = 200+214 = 414$



- When MAZE is stored in Column-Major order
- $L_1 = 8-2+1 = 7$, $L_2 = 1-(-4)+1 = 6$, $L_3 = 5$
  - MAZE contains 7.6.5=210 elements
- $E_1 = 5 - 2 = 3$, $E_2 = -1-(-4) = 3$, $E_3 = 2$

$$LOC(C[K_1, K_2, \ldots, K_n]) = Base(C) + w[(((\ldots(E_N L_{N-1} + E_{N-1})L_{N-2}) + \ldots + E_3)L_2 + E_2)L_1 + E_1]$$

$$MAZE[5,-1,8] = Base(C) + w[(E_3 L_2 + E_2)L_1 + E_1]$$

$$= 200 + 2[(2.6+3).7+3] = 200 + 2[(15).7+3]$$

$$= 200 + 2[108] = 200 + 216 = 416$$