Subject : Data Structures
Subject_code : CS-2201
Course : B.Tech.(III Sem.)

By
Poonam Saini
Department of Computer Science & Engineering
Sir Padampat Singhania University
Udaipur

Course Objectives

- To impart a thorough understanding of linear data structures such as stacks, queues and their applications.
- To impart a thorough understanding of non-linear data structures such as trees, graphs and their applications.
- To impart familiarity with various sorting, searching and hashing techniques and their performance comparison.

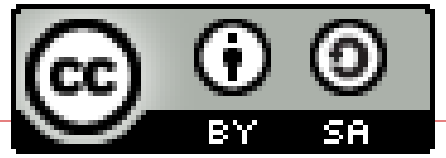
Course Outcomes

- Summarize different categories of data Structures
- Identify different parameters to analyze the performance of an algorithm.
- Explain the significance of dynamic memory management Techniques
- Design algorithms to perform operations with Linear and Nonlinear data structures
- Illustrate various technique to for searching, Sorting and hashing
- Choose appropriate data structures to solve real world problems efficiently.

Recommended Text/Reference Books

1. Data Structures Using C. Tenenbaum A. M., Langsam Y. & Augenstein M. J., Pearson.
2. Data Structures using C. Thareja R., Oxford.
3. Data Structures using C and C++. Shukla R. K., Wiley – India.
4. Data Structures: A Pseudocode Approach with C. Gilberg R. F. & Forouzan, 2nd Ed. CENGAGE Learning.
5. Introduction to Data Structure and Its Applications. Tremblay J. P. & Sorenson P. G.
6. C & Data Structures. Deshpande P. S. & Kakde O. G., DreamTech press.
7. Data Structure Using C. Balagurusamy, Tata McGraw-Hill.
8. Data Structures Using C. ISRD Group, 2nd Ed. Tata McGraw-Hill.
9. Data Structures, Adapted by: Pai G. Schaum‟s Outlines.

Module 01
Introduction
to
Data Structures



Introduction

- **Data**

  It is a value or a set of values of different types called data type like string, integer etc.

- **Structure**

  It is a way of organizing the information so that it becomes easy to use or it is a set of rules that hold the data together.

- **Data Structure**

  It is an aggregation of atomic and composite data types into a set with defined relationship.

Data Structure

- A computer is a machine that manipulates data.
- The prime aim of data structure includes:
  - To study how data is organized in a computer
  - How it is manipulated
  - How it is retrieved
  - How it can be utilized, resulting in more efficient programs.

What is a Data Structure ?

- In computer science, a data structure is a particular way of storing and organizing data in a computer so that it can be used efficiently.
- Data may be organized in many different ways, the logical or mathematical model of a particular organization of data in memory or on disk is called Data Structure.
- Algorithms are used for manipulation of data.

Need of a Data Structure

- To solve the complex requirements in efficient way.
- Provide fastest solution of human requirements.
- Provide efficient solution of complex problem.

A data structure helps in understanding: The relationship of one data element with the other and How the data should be organized within the memory?

Need of a Data Structure

- Therefore, a data structure helps you to
  - Analyze the data
  - Store that data
  - Organize that data in a logical or mathematical manner.

Data Representation

- The basic unit of a data representation in a computer is a bit which can be either 0 or 1.
- Basic data types of any language
  - Integer representation
  - Real No. representation
  - Character representation

Data Type

- It consists of two parts, a set of data and the operations that can be performed on the data. Data types facilitate the optimum use of memory as well as a defined way to interpret.

- **Atomic data type:** It is a set of atomic data with identical properties. It has a set of values and a set of operations that are to be performed on it. For eg.:

  Values: -∞, …, -2, -1, 0, 1, 2, ….., ∞

  Operations: *, +, -, %, /, ++, - -

Data Type

- **Composite data type**: It is the opposite of atomic data type. Composite data can be broken out into subfields that have meaning. For eg.: Enrollement No: 18CS00001, address etc.

Abstract Data Type

- It is defined as a mathematical model of the data objects that make up a data type as well as the functions that operate on these objects.
- An abstract data type is a data declaration packaged together with the operations that are meaningful for the data type.
- We know what a data type can do but how it is done is hidden i.e. known as the concept of abstraction.

Primitive Data Type

- These are the basic data types of any language that form the basic unit for the data structure defined by the user.
- It defines how the data will be internally represented in, stored and retrieved from the memory.
- For eg.: int, char, float

Difference between ADT, data Type and Data structure

- An ADT is the specification of the data type which specifies the logical and mathematical model of the data type.
- A data type is the implementation of an abstract data type.
- Data structure refers to the collection of computer variables that are connected in some specific manner.

Data Structure and its characteristics

The logical and mathematical model of a particular organization of data is called a data structure. Or, it can be defined as a set of data elements that represent the operations such as insertion, deletion, modification and traversal of the values present in the data elements.

In other words, data structure can also be defined as the logical or mathematical model of a particular organization of data.

Data Structure and its characteristics

The main characteristics of a data structure are:

- It Contains **component data items** which may be atomic or another data structure
- It also contains a **set of operations** on one or more of the component items

Data Structure

- **It Defines rules** as to how components relate to each other and to the structure as a whole. The choice of a particular data structure depends on following consideration:

- It must be rich enough in structure to mirror actual relationships of data in real world for example the hierarchical relationship of the entities is best described by the — **tree** data structure.
- The structure should be simple enough that one can effectively process the data when necessary.

Types of Data structures

The various data structures are divided into following categories:

- **Linear data structure-**

A data structure whose elements form a sequence, and every element in structure has a unique **predecessor**

and a unique **successor**. Examples of linear data structure are:

- **Arrays**
- **Linked Lists**
- **Stacks**
- **Queues**

Types of Data structures

- **Non-Linear data structures-**

A data structure whose elements do not form a sequence. There is no unique predecessor or unique successor. Examples of non linear data structures are trees and graphs.

Linear Data Structures

☐ Arrays

- An array is a list of finite number of elements of same datatype. The individual elements of an array are accessed using an index or indices to the array. Depending on number of indices required to access an individual element of an array, array can be classified as:

• One-dimensional array or linear array that requires only one index to access an individual element of an array

Linear Data Structures

☐ Arrays

- Two dimensional array that requires two indices to access an individual element of array
- The arrays for which we need two or more indices are known as multidimensional array.

Linear Data Structures

☐ Linked List

- Linear collection of data elements called nodes
- Each node consists of two parts; data part and pointer or link part
- Nodes are connected by pointer links.
- The whole list is accessed via a pointer to the first node of the list
- Subsequent nodes are accessed via the link-pointer member of the current node

Linear Data Structures

☐ Linked List

•Link pointer in the last node is set to null to mark the list‘s end

– Use a linked list instead of an array when You have an unpredictable number of data elements (dynamic memory allocation possible)

• Your list needs to be sorted quickly

Linear Data Structures

Types of linked lists:

- Singly linked list
  - Begins with a pointer to the first node
  - Terminates with a null pointer
  - Only traversed in one direction
- Circular, singly linked List
  - Pointer in the last node points back to the first node

Linear Data Structures

- Doubly linked list

•Two —start pointers‖ – first element and last element

• Each node has a forward pointer and a backward pointer

• Allows traversals both forwards and backwards

Linear Data Structures

- Circular, doubly linked list

• Forward pointer of the last node points to the first node and backward pointer of the first node points to the last node

- Header Linked List

• Linked list contains a header node that contains information regarding complete linked list.

Linear Data Structures

☐ Stack

A stack, also called last-in-first-out (LIFO) system, is a linear list in which insertions (push operation) and deletions (pop operations) can take place only at one end, called the

top of stack

. Similar to a pile of dishes

Linear Data Structures

☐ Stack

- Bottom of stack indicated by a link member to **NULL.**
- Constrained version of a linked list

•The two operations on stack are:

push- Adds a new node to the top of the stack

pop – Removes a node from the top

- Stores the popped value
- Returns **true** if **pop** was successful

Linear Data Structures

❑ Queues

A queue, also called a First-in-First-out (FIFO) system, is a linear list in which insertions can take place at one end of the list, called the

Rear: of the list and deletions can take place only from other end , called the

Front: of the list.

- Similar to a supermarket checkout line
- Insert and remove operations

Non-Linear Data Structures

☐ Tree

A tree is a non-linear data structure that represents a hierarchical relationship between various elements. The top node of a tree is called the **root node** and each subsequent node is called the child node of the root. Each node can have one or more than one child nodes. A tree that can have any number of child nodes is called a general tree. If there is an maximum number N of successors for a node in a tree, then the tree is called an N-ary tree. In particular a binary (2-ary) tree is a tree in which each node has either 0, 1, or 2 successors.

Non-Linear Data Structures

- **Binary trees**
  - Binary tree can be empty without any node whereas a general tree cannot be empty.
  - All nodes contain two links
  - None, one, or both of which may be NULL
  - The root node is the first node in a tree.
  - Each link in the root node refers to a child
  - A node with no children is called a leaf node

Non-Linear Data Structures

- **Binary search tree**
  - A type of binary tree
  - Values in left subtree less than parent
  - Values in right subtree greater than parent
  - Facilitates duplicate elimination
  - Fast searches, maximum of log n comparisons

Non-Linear Data Structures

☐ Graph

A graph, G , is an ordered set (V,E) where V represent set of elements called nodes or vertices in graph terminology and E represent the edges between these elements. This data structure is used to represent relationship between pairs of elements which are not necessarily hierarchical in nature. Usually there is no distinguished ‘first' or `last' nodes. Graph may or may not have cycles

Non-Linear Data Structures

Types of Graphs

1. Connected Graph: A path exists between any two vertices of the graphs.

2. Complete Graph: Every node is connected with every other node in the graph.

3. Weighted Graph: Each edge is assigned a number.

4. Directed Graph: The direction of the edges indicate the path between vertices.

5. Undirected Graph: The direction of edges are not indicated.

Characteristics of Data Structures

Data Structure	Advantages	Disadvantages
Arrays	Quick insertion, very fast access if index is known	Slow search, Slow deletion, Fixed size
Ordered Array	Quicker search than unsorted array.	Slow insertion and deletion, Fixed size
Stack	Provides last-in first-out access	Slow access to other items
Queue	Provides first-in first-out access	Slow access to other items
Linked List	Quick insertion and quick deletion	Slow search
Binary Trees	Quick search, insertion and deletion if tree remains balance	Deletion algorithm is complex.
Hash Table	Very fast access if key known, Fast insertion.	Slow deletion, access slow if key not known, inefficient memory usage
Heap	Fast insertion ,deletion. Access to largest item	Slow access to other items
Graph	Models real world situation	Some algorithms are slow and access.

Operations on Data Structures

1. **Traversing:** Accessing each record exactly once so that certain items in the record may be processed.
2. **Searching:** Finding the location of the record with a given key value or finding the locations of all records which satisfy one or more conditions.
3. **Insertion:** Adding a new record to the structure.
4. **Deletion:** Removing a record from the structure.

Operations That are used in Special Situations on Data Structures

1. **Sorting:** Arranging the records in some logical order.
2. **Merging:** Combining the records in two different sorted files into a single sorted file.

Refinement Stages to solve a complex problem

- The best approach to solve a complex problem is to divide it into smaller parts such that each part becomes an independent module which is easy to manage.
- Different problems have different no. of refinement stages, but in general there are four levels of refinement processes.

Refinement Stages to solve a complex problem

1. **Conceptual or abstract Level**

At this level, we decide how the data is related to each other and what operations are needed.

2. **Algorithmic or Data structure level**

At this level, we decide, what kind of data structure will be required to solve the problem. For eg. Contiguous list for retrieving any element or stacks for evaluation of a prefix/ postfix notation.

Refinement Stages to solve a complex problem

3. Programming or Implementation level

At this level, we decide the details of how the data structures will be represented in the computer memory. For eg. We decide whether the linked lists will be implemented with pointers or with the cursors in an array.

4. Application level

This level settles all the details required for particular application such as names for variables or special requirements for the operations imposed by applications.

What is an algorithm?

- An algorithm is a finite set of instructions that takes some raw data as input and transforms it into refined data.

- An algorithm is a well-defined list of steps for solving computational problem.

Characteristics of an algorithm

- Initially an input is provided to an algorithm before it begins.
- The processing rules specified in the algorithm must be precise and unambiguous and lead to a specific action.
- Processing should be done in finite time.
- Repetition of steps should be finite.
- An algorithm must have one or more outputs.

Efficiency of an algorithm

- Efficiency of an algorithm can be analyzed by finding the memory space and running time required for an algorithm.
- An efficient algorithm takes less memory space and running time and produce correct results.

Different Approaches for Designing an Algorithm

☐ Top-Down Approach

It starts by identifying the major components of the system or program decomposing them into their low-level components and iterating until the desired level of module complexity is achieved.

☐ Bottom-up Approach

It starts with designing the most basic or primitive components and proceeds to higher-level components.

Comparison of both the Approaches

☐ In case of Top-Down Approach

- No emphasis is given on the identification of communication or on reusability of components.
- Little attention is paid to data.
- No information hiding.

☐ Bottom-up Approach

- More attention is paid to data
- More emphasis on communication or on reusability of components.
- It follows information hiding.

Analysis of Algorithms

Analysis of an algorithm requires two main considerations:

- Time Complexity
- Space Complexity

Complexity of Algorithm

- Efficiency or complexity of an algorithm is stated as a function relating the length to the number of steps (time complexity) or storage location (space complexity).

  f(n)

- In simple words complexity of an algorithm is the time and space it uses.

Time Complexity

- **Time Complexity**: It is the running time of the program as a function of size of input. While measuring the time complexity of an algorithm, we concentrate on developing only the frequency count for all key statements.

Algorithm A: `a=a+1`

Frequency count of Algorithm A is 1

Time Complexity

Algorithm B: for x=1 to n step 1

a=a+1

Loop

Frequency count of Algorithm B is n

Algorithm C: for x=1 to n step 1

for y=1 to n step 1

a=a+1

Loop

Frequency count of Algorithm C is n^2

Time Complexity

If an algorithm performs f(n) basic operations where n is the size of the input then total running time will be

$$C\ f(n)$$

Where C is a constant that depends upon the algorithm.

Space Complexity

- It is the amount of Computer memory required during the program execution, as a function of input size.
- The space needed by the program is the sum of the following components:

- Fixed space requirements: It includes instruction space for simple variables, fixed size structured variables and constants.

Space Complexity

- Variable space requirement: It consists of space needed by structured variables whose size depends on particular instance of variables. It also includes the additional space required when the function uses recursion.

Big-O Notation

- It helps to determine the time as well as space complexity of the algorithm. We are not concerned with an exact measurement of an algorithm's efficiency but are concerned with its general order of magnitude.
- f(n) represents the number of statements executed for n elements of data. We are not concerned about the complete measure of efficiency but with only the factor that determines the magnitude.
- This factor is the big-O as an in "on the order of".

Big-O Notation

- It is expressed as O(n) i.e. on the order of n.
- The Big-O notation can be derived from f(n) using the following steps:

1. In each term, set the coefficient of the term to 1.
2. Keep the largest term in the function and discard the others.

Terms are ranked from lowest to highest as follows:

$\log_2 n$, n, $n\log_2 n$, n^2 , n^3 ,-----, n^k ,2^n ,n!

Big-O Notation

- For eg.: Calculate big-O notation for

  f(n) =n(n+1)/2

Sol. $f(n)=1/2\ n^2 + 1/2\ n$

$= n^2 + n$

$= n^2$

Therefore, big-O notation is stated as

$O(f(n)) = O(n^2)$

Asymptotic Notation

- Helps to compare algorithms.
- Suppose we are considering two algorithms, A and B, for solving a given problem. Furthermore, let us say that we have done a careful analysis of the running times of each of the algorithms and determined them to be Ta(n) and Tb(n),respectively, where n is a measure of the problem size. Then it should be a fairly simple matter to compare the two functions and to determine which algorithm is the best!

Types of Analysis

- Types of Analysis

- Worst case running time

– Average case running time

– Best case running time

Worst case Running Time

- The behavior of the algorithm with respect to the worst possible case of the input instance.
- The worst-case running time of an algorithm is an upper bound on the running time for any input. Knowing it gives us a guarantee that the item does not occur in data.
- There is no need to make an educated guess about the running time.

Average case Running Time

- The expected behavior when the input is randomly drawn from a given distribution.
- The average-case running time of an algorithm is an estimate of the running time for an "average" input.
- Computation of average-case running time entails“ knowing all possible input sequences, the probability distribution of occurrence of the sequences, and the running times for the individual sequences”.
- Often it is assumed that all inputs of a given size are equally likely

Best case Running time

- The behavior of the algorithm when input is in already in order.
- For example in sorting, if elements are already sorted for a specific algorithm.
- The best case running time rarely occurs in practice comparatively with the first and second case.

Time-Space Tradeoff

- In computer science, a **space-time** or **time-memory tradeoff** occurs.
- It is a way of solving a problem or calculation in less time by using more storage space (or memory), or by solving a problem in very little space by spending a long time.
- So if your problem is taking a long time but not much memory, a space-time tradeoff would let you use more memory and solve the problem more quickly.
- Or, if it could be solved very quickly but requires more memory than, you can try to spend more time solving the problem in the limited memory.

Time Complexity

```
int add(int num1, int num2)
 {
   int total = num1 + num2;
    return total;
};
```

Steps

1.Looking up num1
2.Looking up num2
3.Assigning the sum of the two numbers to the variable total
4.Returning total.

NOTE: Their time complexity is O(1) or constant time because the operations only happen once, and they do not depend on the size of the input as they run.

O(1 + 1 + 1 + 1) = O(4), which we will then simplify to O(1) as we strip our constants and identify our highest-order term.

Big O Notation

Data Structure	Insert	Retrieve
Array	O(1)$O(1)$	O(1)$O(1)$
Linked List	At Head: O(1)$O(1)$ At Tail: O(n)$O(n)$	O(n)$O(n)$
Binary Search	O(n)$O(n)$	O(n)$O(n)$
Dynamic Array	O(1)$O(1)$	O(1)$O(1)$
Stack	O(1)$O(1)$	O(1)$O(1)$

Sorting Algorithm	Worst-case scenario	Average Case	Best-case scenario
Bubble Sort	O(n^2)$O(n2)$	O(n^2)$O(n2)$	O(n)$O(n)$
Insertion Sort	O(n^2)$O(n2)$	O(n^2)$O(n2)$	O(n)$O(n)$
Selection Sort	O(n^2)$O(n2)$	O(n^2)$O(n2)$	O(n^2)$O(n2)$
Quick Sort	O(n^2)$O(n2)$	O(n log n)$O(nlogn)$	O(n log n)$O(nlogn)$
Merge Sort	O(n log n)$O(nlogn)$	O(n log n)$O(nlogn)$	O(n log n)$O(nlogn)$
Heap sort	O(n log n)$O(nlogn)$	O(n log n)$O(nlogn)$	O(n log n)$O(nlogn)$

Arrays

Introduction

- An array is a finite collection of similar elements stored in adjacent memory locations.
- An array with n number of elements is referenced using an index that ranges from 0 to n-1.
- The lowest index of an array is called the lower bound(LR).
- The highest index of an array is called the upper bound(UP).
- The number of elements in an array is called its range.

  LB=0

  UB=n-1

Size of an array= UB-LB+1 = n – 1 – 0 + 1 =n

Array Declaration & Initialization in C

- int age[20]
- int age[5]=:8,10,5,15,20};
- char name[]=“SPSU”;
- char name[]={‘S’,’P’,’S’,’U’};

Linear Arrays

- All elements in A are written symbolically as, 1 .. n is the subscript.

  A1, A2, A3, ... , An

- i.e., FORTRAN, PL/1 and BASIC → A(1), A(2), ..., A(N)

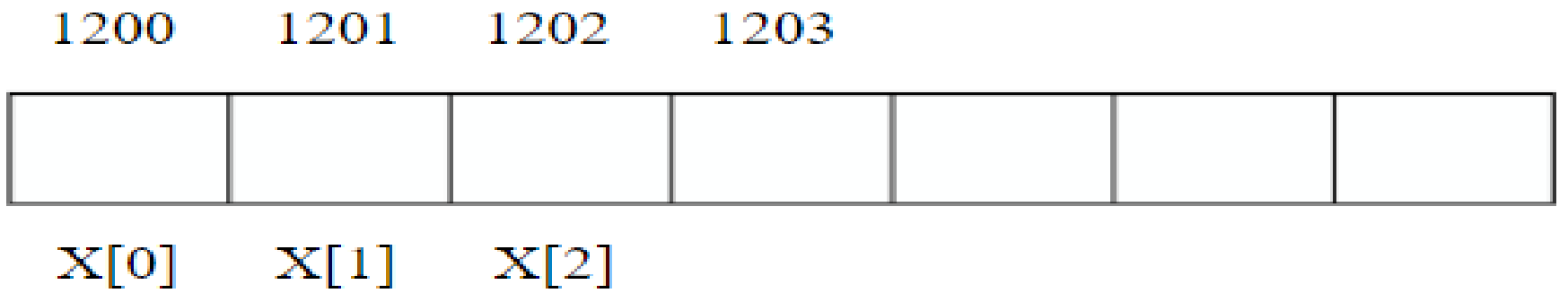
  Java → A[0], A[1], ..., A[N-1] , subscript starts from 0

  LB = 0, UB = N–1

Representation of Linear Arrays in Memory

- The process to determine the address in a memory:
  a) First address – base address.
  b) Relative address to base address through index function.

  Example: char X[100];
  Let *char* uses 1 location storage.
  If the base address is 1200 then the next element is in 1201.
  Index Function is written as:
  Loc (X[i]) = Loc(X[0]) + i , *i* is subscript and LB = 0



Representation of Linear Arrays in Memory

- In general, index function:
  Loc (X[i]) = Loc(X[LB]) + w*(i-LB);

  where w is length of memory location required.
  For real number: 4 byte, integer: 2 byte and character: 1 byte.

Example:
int a[5];
Given Base(a) = 100, LB=0, w=4
Calculate the address of the a[2]
Solution:
Loc(a[2]) = 100 + 4 * (2-0)
= 108

Representation of Linear Arrays in Memory

- Example:

  If LB = 5, Loc(X[LB]) = 1200, and w = 4, find Loc(X[8]) ?

  Loc(X[8])= Loc(X[5]) + 4*(8 – 5)

  = 1212

Array Operations

Array Traversal

Traversing operation means visit every element once.
e.g. to print, etc.

Traversing 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

Array Operations

- **Insertion**

Adding an Element in the:

- **Beginning**
- **Middle**
- **End**

Array Operations

Insertion

- Insert item at the back is easy if there is a space. Insert item in the middle requires the movement of all elements to the right as in Figure 1.

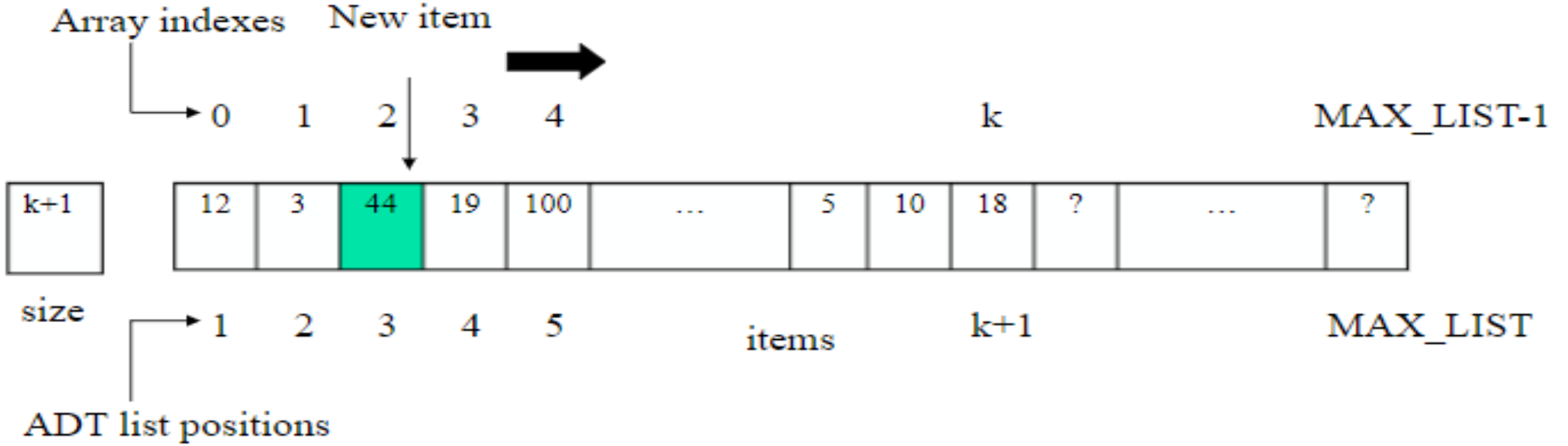


Figure 1: Shifting items for insertion at position 3

Array Operations

Insertion: Adding an Element in the End

Insert Ford at the End of Array

1	**Brown**
2	Davis
3	Johnson
4	Smith
5	Wagner
6	
7	
8	

1	**Brown**
2	Davis
3	Johnson
4	Smith
5	Wagner
6	Ford
7	
8	

Array Operations

Insertion: Adding an Element in the End

Insert Ford at the End of Array

1	**Brown**
2	Davis
3	Johnson
4	Smith
5	Wagner
6	
7	
8	

1	**Brown**
2	Davis
3	Johnson
4	Smith
5	Wagner
6	Ford
7	
8	

Array Operations

☐ Insertion Algorithm: INSERT (LA, N , K , ITEM)

LA is a linear array with N elements

K is a positive integers such that K ≤ N.

This algorithm insert an element ITEM into the K^{th} position in LA

1. [Initialize Counter]

   Set J := N

2. Repeat Steps 3 and 4 while J ≥ K

3. [Move the J^{th} element downward]

   Set LA[J + 1] := LA[J]

4. [Decrease Counter]

   Set J := J -1

5 [Insert Element]

   Set LA[K] := ITEM

6. [Reset N]

   Set N := N +1;

7. Exit

Array Operations

- **Deletion**

Deleting an Element from the:

- **Beginning**
- **Middle**
- **End**

Array Operations

Deletion: Deleting an Element from the end

Deletion of Wagner at the End of Array

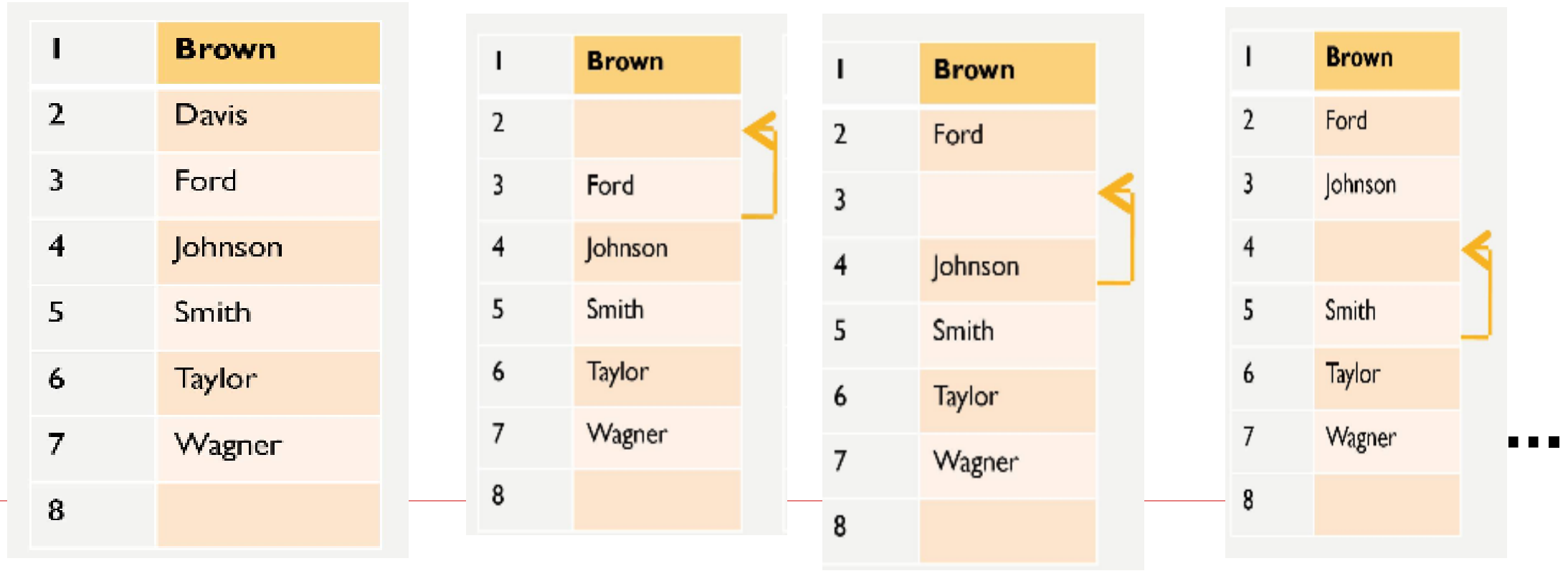
1	Brown
2	Davis
3	Ford
4	Johnson
5	Smith
6	Taylor
7	Wagner
8	

1	Brown
2	Davis
3	Ford
4	Johnson
5	Smith
6	Taylor
7	
8	

Array Operations

☐ Deletion: Deleting an Element from the middle

Deletion of Davis from the Array



Array Operations

☐ Deletion Algorithm: DELETE (LA, N , K , ITEM)

LA is a linear array with N elements

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]
3. [Reset the number N of elements]

   Set N := N - 1;
4. Exit

Two Dimensional Arrays

A Two-Dimensional **m x n** array **A** is a collection of **m . n** data elements such that each element is specified by a pair of integer (such as J, K) called subscript with property that

1 ≤ J ≤ m and **1 ≤ K ≤ n**

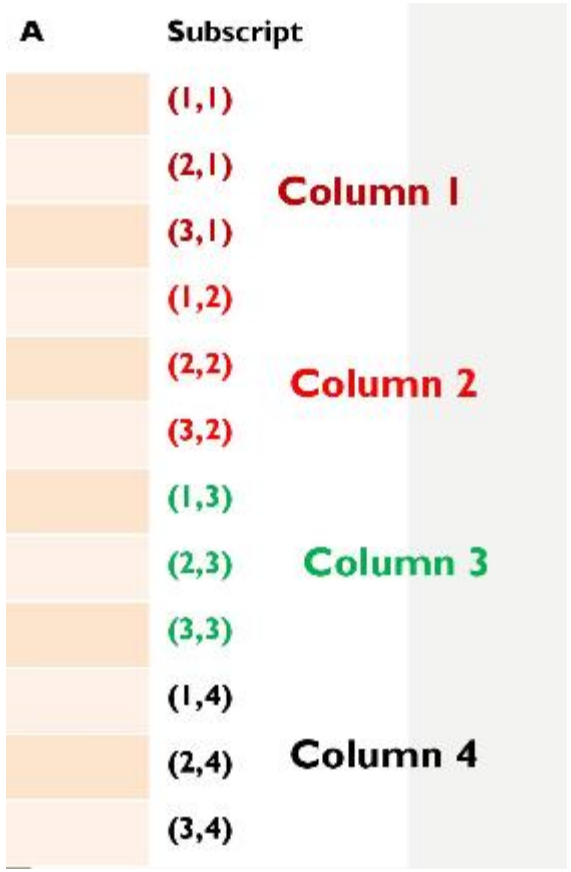
The element of **A** with first subscript **J** and second subscript **K** will be denoted by $A_{J,K}$ or **A[J][K]**

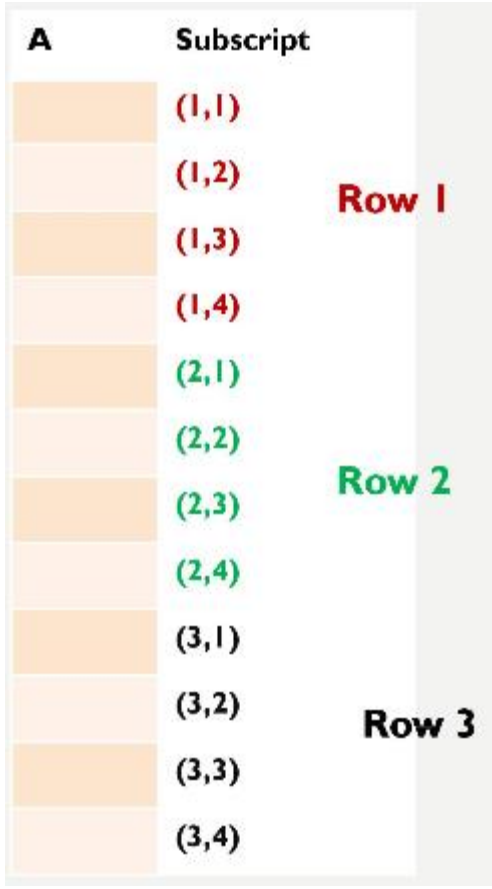
Two Dimensional Arrays

A[0][0]	A[0][1]	A[0][2]	A[0][3]
A[1][0]	A[1][1]	A[1][2]	A[1][3]
A[2][0]	A[2][1]	A[2][2]	A[2][3]

- Let **A** be a two-dimensional array **m x n**
- The array **A** will be represented in the memory by a block of **m x n** sequential memory location
- Programming language will store array **A** either
  - **Column by Column** (Called Column-Major Order) Ex: Fortran, MATLAB
  - **Row by Row** (Called Row-Major Order) Ex: C, C++ , Java

Two Dimensional Arrays: Row and Column Major Order





Column-Major Order

Row-Major Order

Two Dimensional Arrays: Calculating the Address of the Element of a 2-D Array

LOC(LA[K]) = Base(LA) + w(K - I)

- **LOC(A[J,K]) of A[m,n]**

Column-Major Order

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

Row-Major Order

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

w→ word Size
m→ No. of rows
n→ No. of Columns

Two Dimensional Arrays: Calculating the Address of the Element of a 2-D Array Example

- Consider a 25 x 4 array A. Suppose the Base(A) = 200 and w =4. Suppose the programming store 2D array using row-major. Compute LOC(A[12,3])

Row-Major Order

$$LOC(A[J,K]) = Base(A) + w[n(J-1) + (K-1)]$$

$$LOC(A[12,3]) = 200 + 4[4(12-1) + (3 -1)]$$

$$= 384$$

Searching

Searching

- Searching refers to the operation of finding the location of any item in the array.
- The search is said to be successful if the item is found;
- Otherwise, it is unsuccessful.

Linear Search

- Compare the given item i.e., to be searched with each element of the array one by one.
- This method traverses the data sequentially and is called linear or sequential search.

Algorithm for Linear Search: Linear(A,N,ITEM,LOC)

A→ A Linear array of size N
N→ Size of the array A
ITEM→ item to be searched
LOC→ Location of the element
Found→ Boolean variable

Step 1: [Initialization]
Found:= FALSE, K:=0
Step 2: [Read the value i.e., to be searched]
Read ITEM
Step 3: [Comparing each element with the ITEM]
Repeat while K<N
If A[K] = ITEM
FOUND:= TRUE
goto STEP 4
else
K:= K+1

Step 4: [Checking for the value of FOUND]
If Found:= TRUE
Write “Record Found”
else
Write “Record Not Found”
Step 5: [Finished]
Exit

Algorithm for Binary Search: Binary(A,LB,UB,ITEM,LOC)

A→ A is a sorted array of size N
LB→ Lower Bound
UB→ Upper Bound
ITEM→ item to be searched
LOC→ Location of the element

Step 1: [Initialization]
Set BEG:=LB, END:=UB,
MID:=INT(BEG+END)/2
Step 2: Repeat Step 3 and 4
while BEG<=END and A[MID] ≠ ITEM
Step 3: If ITEM < A[MID] then
Set END:=MID-1
else
Set BEG:=MID+1
[End of If structure]
Step 4: Set MID:=INT((BEG+END)/2)
[End of Step 2 Loop]
Step 5: If A[MID]=ITEM then
Set LOC:=MID
else
Set LOC:=NULL
[End of If structure]
Step 5: [Finished]
Exit