RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

Data Structure & Algorithms (DSA) : CCT203

Semester - III, B.Tech. CSE (Cyber Security)

UNIT I

GC Code: dyb4gii

Course Objectives

1. CO1: To impart to students the basic concepts of data structures and algorithms.
2. CO2: To familiarize students on different searching and sorting techniques.
3. CO3: To prepare students to use linear (stacks, queues, linked lists) and nonlinear (trees, graphs) data structures.
4. CO4: To enable students to devise algorithms for solving real-world problems.

Text Books & Reference Books

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

Text Books:

1. Ellis Horowitz, Sartaj Sahni & Susan Anderson-Freed, Fundamentals of Data Structures in C, Second Edition, Universities Press, 2008.
2. Mark Allen Weiss; Data Structures and Algorithm Analysis in C; Second Edition; Pearson Education; 2002.
3. G.A.V. Pai; Data Structures and Algorithms: Concepts, Techniques and Application; First Edition; McGraw Hill; 2008.

Reference Books:

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein; Introduction to Algorithms; Third Edition; PHI Learning; 2009.
2. Ellis Horowitz, Sartaj Sahni and Sanguthevar Rajasekaran; Fundamentals of Computer Algorithms; Second Edition; Universities Press; 2008.
3. A. K. Sharma; Data Structures using C, Second Edition, Pearson Education, 2013.

Course Outcomes

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

On completion of the course the student will be able to

1. Recognize different ADTs and their operations and specify their complexities.
2. Design and realize linear data structures (stacks, queues, linked lists) and analyze their computation complexity.
3. Devise different sorting (comparison based, divide-and-conquer, distributive, and tree-based) and searching (linear, binary) methods and analyze their time and space requirements.
4. Design traversal and path finding algorithms for Trees and Graphs.

UNIT I: Data Structures and Algorithms Basics

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

- **Introduction**: Basic terminologies, elementary data organizations, data structure operations; abstract data types (ADT) and their characteristics.
- **Algorithms**: definition, characteristics, analysis of an algorithm, asymptotic notations, time and space tradeoffs.
- **Array ADT**: definition, operations and representations – row-major and column-major.

Introduction to
Data Structures & Algorithm

Introduction

What is Data Structure?

- A data structure is basically a group of data elements that are put together under one name, and which defines a particular way of storing and organizing data in a computer so that it can be used efficiently.

What is the significance of Data Structure?

- Data structures are used in almost every program or software system.
- DS enable the programmers to manage huge amounts of data easily and efficiently
- The primary goal of a program or software is not to perform calculations or operations but to store and retrieve information as fast as possible.

What are the application of DS?

- Compiler design, Operating system, Statistical analysis package, DBMS
- Numerical analysis, Simulation, Artificial intelligence, Graphics

What are the common examples of DS?

- Arrays, linked lists, queues, stacks, binary trees, and hash tables

Introduction

What are the steps performed, when selecting a data structure to solve a problem?

- Analysis of the problem to determine the basic operations that must be supported. For example, basic operation may include inserting/deleting/searching a data item from the data structure.
- Quantify the resource constraints for each operation.
- Select the data structure that best meets these requirements.

Explain Elementary Data Structure Organization

- Data structures are building blocks of a program.
- The term **data** means a value or set of values. Ex: marks of students, name of an employee, etc.
- **A record** is a collection of data items. For example, the name, address, course, and marks obtained are individual data items. But all these data items can be grouped together to form a record.
- **A file** is a collection of related records. For example, if there are 60 students in a class, then there are 60 records of the students

Classification of Data Structures

Classification of DS

- DS are categorized into two classes: **primitive and non-primitive data structures.**

Primitive data structures:

- They are the fundamental data types which are supported by a programming language.
- Some basic data types are integer, real, character, and boolean.
- The terms ‘data type’, ‘basic data type’, and ‘primitive data type’ are often used interchangeably.

Non-primitive data structures:

- They are those data structures which are created using primitive data structures.
- Examples of such data structures include linked lists, stacks, trees, and graphs.
- Non-primitive data structures can further be classified into two categories: **linear and non-linear data structures.**

Classification of DS

Linear DS:

- Here, the elements of a data structure are **stored in a linear or sequential order**
- Examples include arrays, linked lists, stacks, and queues.
- Linear data structures can be represented in memory in two different ways.
- One way is to have to a **linear relationship between elements** by means of sequential memory locations.
- The other way is to have a linear relationship **between elements by means of links.**

Non Linear DS:

- The elements of a data structure are **not stored in a sequential order.**
- The relationship of adjacency is not maintained between elements of a **non-linear data structure.**
- Examples include trees and graphs.

Arrays

- An array is a collection of similar data elements.
- These data elements have the same data type.
- The elements of the array are stored in consecutive memory locations and are referenced by an
- index (also known as the subscript).
- In C, arrays are declared using the following syntax:

  type name[size];
- For example, int marks[10];
- In C, the array index starts from zero. (from 0 to 9)
- Fig: Memory representation of an array of 10 elements

1st element	2nd element	3rd element	4th element	5th element	6th element	7th element	8th element	9th element	10th element
marks[0]	marks[1]	marks[2]	marks[3]	marks[4]	marks[5]	marks[6]	marks[7]	marks[8]	marks[9]

Arrays

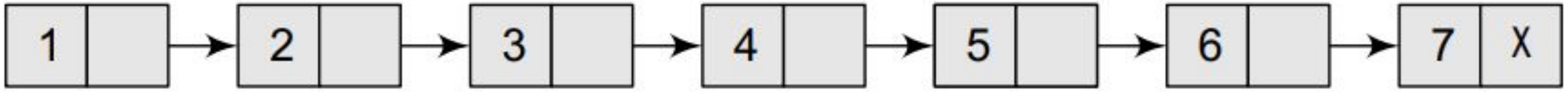
- Arrays are generally used when we want to **store large amount of similar type of data.**
- But they have the following **limitations:**
  - Arrays are of fixed size.
  - Data elements are stored in contiguous memory locations which may not be always available.
  - Insertion and deletion of elements can be problematic because of shifting of elements from their positions.

1st element	2nd element	3rd element	4th element	5th element	6th element	7th element	8th element	9th element	10th element
marks[0]	marks[1]	marks[2]	marks[3]	marks[4]	marks[5]	marks[6]	marks[7]	marks[8]	marks[9]

Linked Lists

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

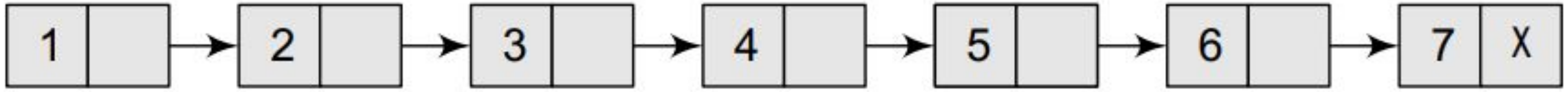
- A linked list is a very **flexible, dynamic data structure** in which elements (**called nodes**) form **a sequential list.**
- In contrast to static arrays, a programmer **need not worry about how many elements** will be stored in the linked list.
- This feature enables the programmers to **write robust programs** which require less maintenance.



Linked Lists

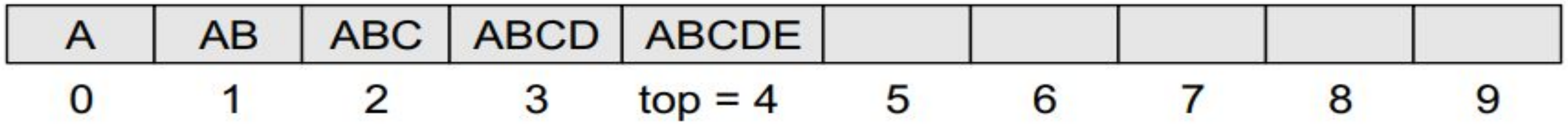
RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

- In a linked list, **each node is allocated space as it is added to the list.**
- Every node in the list **points to the next node** in the list.
- Therefore, in a linked list, every node contains the following **two types of data**:
  - The **value of the node** or any other **data** that corresponds to that node
  - A **pointer or link to the next node** in the list
- **Advantage:**
  Easier to insert or delete data elements
- **Disadvantage:**
  Slow search operation and requires more memory space



Stacks

- A stack is a linear data structure in which **insertion and deletion** of elements are done at only one end, which is known as **the top of the stack.**
- Stack is called a **last-in, first-out (LIFO) structure** because the last element which is added to the stack is the first element which is deleted from the stack.
- Stacks can be implemented using **arrays or linked lists.**
- Every stack has two variables: **variable "TOP" & variable "MAX".**
- **TOP** = **store the address of the topmost element** of the stack.
- **TOP** = It is this position from where the element will be added or deleted.
- **MAX** = It is used to **store the maximum number** of elements that the stack can store.
- If **top = NULL**, then it indicates that the **stack is empty**
- If **top = MAX–1**, then the **stack is full.**



Stacks

- TOP = 4, so insertions and deletions will be done at this position.
- Here, the stack can store a maximum of 10 elements (from 0–9).

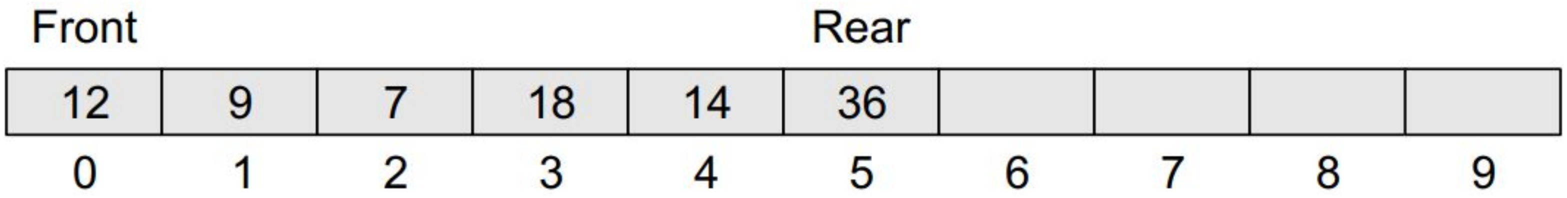
Basic operations: push, pop, and peep.

- The push operation adds an element to the top of the stack.
- The pop operation removes the element from the top of the stack.
- The peep operation returns the value of the topmost element of the stack (without deleting it).
- Before inserting an element in the stack, we must check for overflow conditions.
- An overflow occurs when we try to insert an element into a stack that is already full.
- Before deleting an element from the stack, we must check for underflow conditions.
- An underflow condition occurs when we try to delete an element from a stack that is already empty.

A	AB	ABC	ABCD	ABCDE					
0	1	2	3	top = 4	5	6	7	8	9

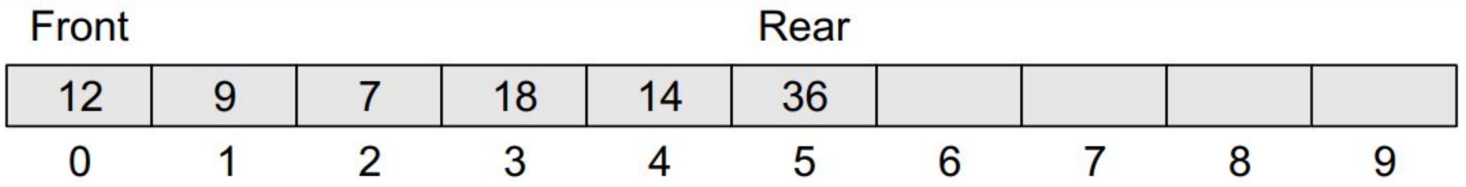
Queues

- It is a **first-in, first-out (FIFO)** data structure in which the element that is inserted first is the first one to be taken out.
- The elements in a queue are **added at one end called the rear** and **removed from the other end called the front.**
- Like stacks, queues can be implemented by using either arrays or linked lists.
- Every queue has **front and rear variables** that point to the position from where deletions and insertions can be done, respectively
- Here, front = 0 and rear = 5.
- If we want to add one more value to the list, say, value 45, then **rear = rear+1** and the value would be stored at the position pointed by the rear

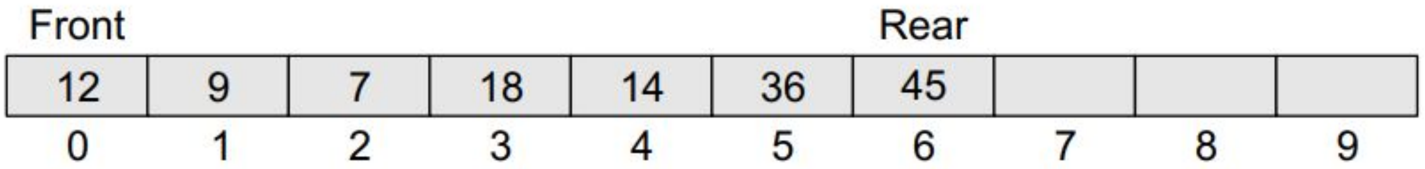


Queues

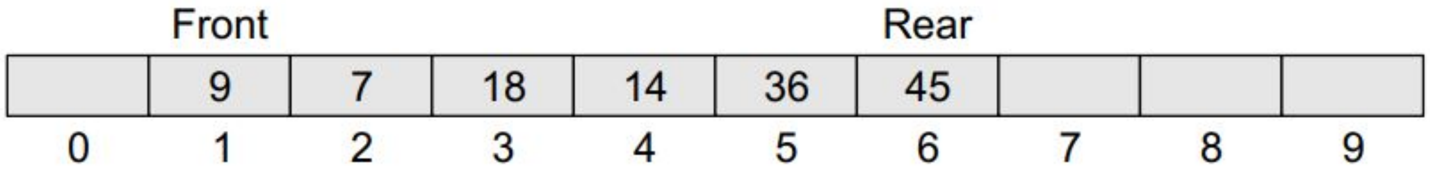
RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur



- If we want to **add one more value** to the list, say, value 45, then **rear = rear+1** and the value would be stored at the position pointed by the rear



- Now, if we want to delete an element from the queue, then **Front = Front +1** . Deletions are done only from this end of the queue



Queues

Before inserting an element in the queue, check for overflow conditions.

- **A queue is full when rear = MAX–1**,
  where MAX is the size of the queue, that is MAX specifies the maximum number of elements in the queue.
- **Note that we have written MAX–1 because the index starts from 0.**

Before deleting an element from the queue, check for underflow conditions.

- **If front = NULL** and **rear = NULL**, then there is no element in the queue.

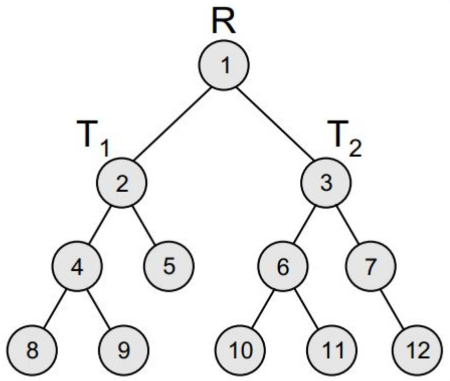
What is Overflow and Underflow conditions:

- An overflow occurs when we try to insert an element into a queue that is already full
- An underflow occurs when we try to delete an element from a queue that is already empty.

Trees

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

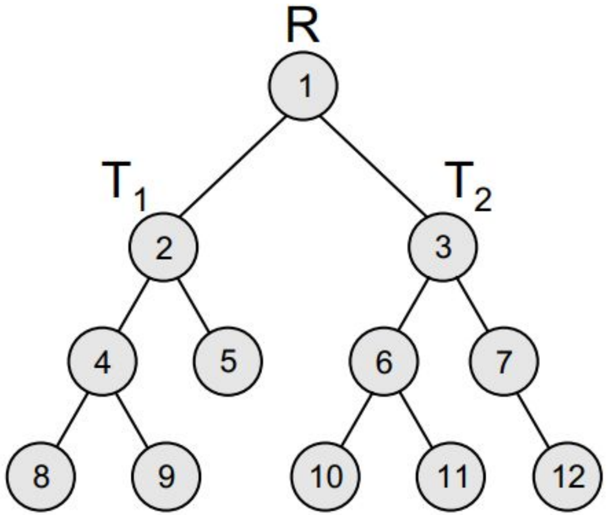
- A tree is a **non-linear data structure** which consists of a collection of nodes arranged in a **hierarchical** order.
- **One of the nodes is designated as the root node**, and the **remaining nodes** can be **partitioned into disjoint sets** such that **each set is a sub-tree of the root.**
- The simplest form of a tree is a binary tree.
- A binary tree consists of a **root node** and **left and right sub-trees**, where **binary trees.**
- Each node contains a **data element, a left pointer** which points to the left sub-tree, and a **right pointer** which points to the right sub-tree.
- **The root element is the topmost node** which is pointed by a 'root' pointer.
- **If root = NULL then the tree is empty.**



Trees

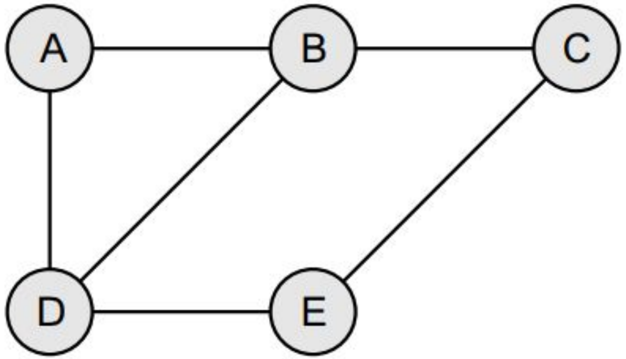
In fig:

- node 1 is the root node (R)
- node 2 is the left child of root node (T1)
- node 3 is the right child of the root node (T2)
- Note that the left sub-tree of the root node consists of: the nodes 2, 4, 5, 8, and 9.
- The right sub-tree of the root node consists of the nodes 3, 6, 7, 10, 11, and 12
- **Advantage:** Provides quick search, insert, and delete operations
- **Disadvantage:** Complicated deletion algorithm



Graphs

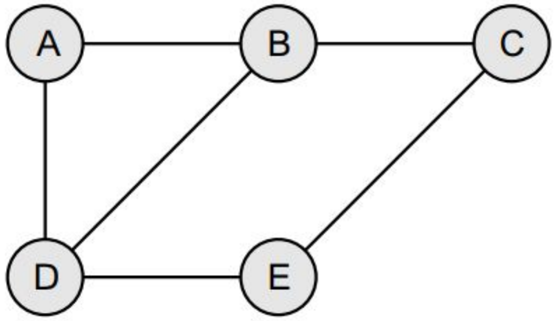
- It is a **non-linear data structure.**
- It is a collection of **vertices (also called nodes)** and **edges** that connect these vertices.
- A graph is often viewed as a **generalization of the tree** structure, where instead of a purely **parent-to-child relationship between tree nodes,** any kind of complex relationships between the nodes can exist.
- In a tree structure, **nodes can have any number of children but only one parent**, a graph on the other hand **relaxes all such kinds of restrictions.**



Graphs

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

- In **Graph**: *Node* and *Edges* can represent:
    - **Maps**: *City* and *Roads*
    - **CN**: *Workstations & Network Connections*.
- Standard graph operations:
    - Searching the graph
    - Finding the shortest path between the nodes of a graph.
- Unlike trees, graphs do not have any root node.
- Here, every node can be connected with every another node.
- When two nodes are connected via an edge, the two nodes are known as **neighbours.**
- For example, in Fig, node A has two neighbours: B and D.
- **Advantage:** Best models real-world situations
- **Disadvantage:** Some algorithms are slow and very complex



Operations of Data Structures

Operations on Data Structures

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

- **Traversing:** To access each data item exactly once.
  For ex, to print the all student names in a class.
- **Searching:** To find the location of one or more data items that satisfy the given constraint.
  For ex, to find the names of all the students who secured 100 marks in mathematics.
- **Inserting:** To add new data items to the given list of data items.
  For ex, to add the details of a new student who has recently joined the course.
- **Deleting:** To remove (delete) a particular data item from the given collection of data items.
  For ex, to delete the name of a student who has left the course.
- **Sorting:** Data items can be arranged in some order like ascending order or descending order.
  For ex, arranging the names of students in a class in an alphabetical order, etc.
- **Merging:** Lists of two sorted data items can be combined to form a single list of sorted data items.

Abstract Data Structures (ADT)

Abstract Data Type

- It is the way we look at a data structure, **focusing on what it does** and ignoring how it does.
- For example, **stacks** and **queues** are perfect examples of **an ADT**

What do we mean by **Abstract** and **Data Type**:

- **Data type** of a variable is the set of values that the variable can take. (int, char, float, & double)
- The word '**abstract**' in the context of data structures means considered apart from the detailed specifications or implementation.
- In C, an abstract data type can be **a structure** considered **without regard to its implementation**
- It can be thought of as a **'description'** of the data in the structure with a **list of operations** that can be performed on the data within that structure
- The end-user is not concerned about the details of how the methods carry out their tasks.
- They are only aware of the **methods that are available** to them and are only concerned about **calling those methods** and **getting the results**.
- They are not concerned about how they work.

Advantage of using ADTs

- In the real world, **programs evolve as a result of new requirements or constraints**.
- Hence, **a modification to a program commonly requires a change** in one or more of its data structures.
- For example, if you want to add a new field to a student's record to keep track of more information about each student, then it will be better to replace an array with a linked structure to improve the program's efficiency.
- In such a scenario, **rewriting every procedure** that uses the changed structure is **not** desirable.
- Therefore, a better alternative is to separate the use of a data structure from the details of its implementation. This is the principle underlying the use of **abstract data types**

Algorithm

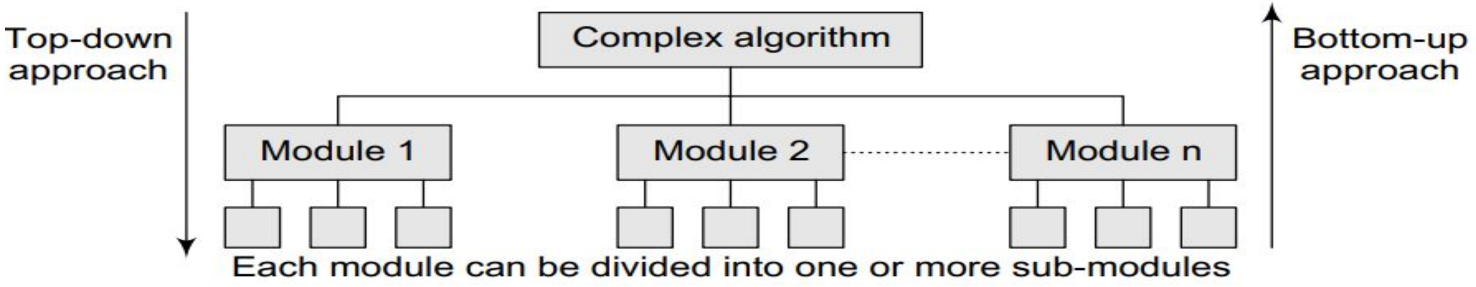
Algorithm

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

- Definition: A formally defined procedure for performing some calculation
- An algorithm provides a blueprint to write a program to solve a particular problem.
- It is considered to be an effective procedure for solving a problem in finite number of steps.
- A well-defined algorithm always provides an answer and is guaranteed to terminate
- Algorithms are mainly used to achieve software reuse

Different Approaches to designing an algorithm

- A complex algorithm is often divided into smaller units called **modules.**
- This process of dividing an algorithm into modules is called **modularization.**
- The **key advantages** of modularization are as follows:
    - It makes the **complex algorithm simpler** to design and implement.
    - Each module can be designed **independently.**
    - While designing one module, the details of other modules can be ignored, thereby enhancing clarity in design which in turn simplifies implementation, debugging, testing, documenting, and maintenance of the overall algorithm.



Different Approaches to designing an algorithm

- There are two main approaches to design an algorithm: **top-down** and **bottom-up**

Top-Down Approach:

- A top-down design approach starts by **dividing the complex algorithm** into one or more modules.
- These modules can further be **decomposed into one or more sub-modules**, and this process of decomposition is **iterated** until the desired level of module complexity is achieved.
- Top-down design method is a form of **stepwise refinement** where we begin with the topmost module and incrementally add modules that it calls.
- Therefore, in a top-down approach, we start from an abstract design and then at each step, this design is refined into more concrete levels until a level is reached that requires no further refinement.

Different Approaches to designing an algorithm

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

- There are two main approaches to design an algorithm: **top-down** and **bottom-up**

Bottom-up approach:

- A bottom-up approach is just the **reverse of top-down** approach.
- In the bottom-up design, we start with designing the **most basic** or **concrete** modules and then proceed towards designing higher level modules.
- The higher level modules are implemented by using the operations performed by lower level modules. Thus, in this approach **sub-modules are grouped together** to form a higher level module.
- All the higher level modules are clubbed together to form even higher level modules.
- This process is repeated until the design of the complete algorithm is obtained.

Top-down vs bottom-up approach

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

Top-down Approach

- It follows a **stepwise refinement** by decomposing the algorithm into modules
- Easy for: generation of test cases, implementation of code, and debugging.
- Problems: sub-modules are analysed in isolation without focusing on their communication with other modules
- reusability of components is less
- It does not allow information hiding

Bottom-up Approach

- It defines a module and then groups together several modules to form a new higher level module.
- It allows information hiding

Design of complex algorithms must not be constrained to proceed according to a fixed pattern but should be a **blend of top-down and bottom-up approaches**.

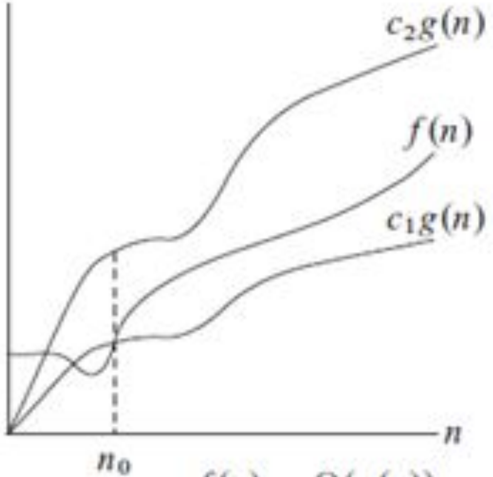
Different ways for analysis of an algorithm

- **Best case Running Time**:
  - Fastest time to complete, with optimal inputs chosen.
  - For example, the best case for a sorting algorithm would be data that's already sorted.
  - Best Case : The minimum possible value of f(n) is called the best case.
- **Worst case Running Time**:
  - Slowest time to complete, with pessimal inputs chosen.
  - For example, the worst case for a sorting algorithm might be data that's sorted in reverse order (but it depends on the particular algorithm).
  - Worst Case : The maximum value of f(n) for any key possible input.
- **Average case Running Time**:
  - Run the algorithm many times, using many different inputs of size n that come from some distribution that generates these inputs, compute the total running time and divide by the number of trials
  - Average Case : The expected value of f(n).

Why Asymptotic notations?

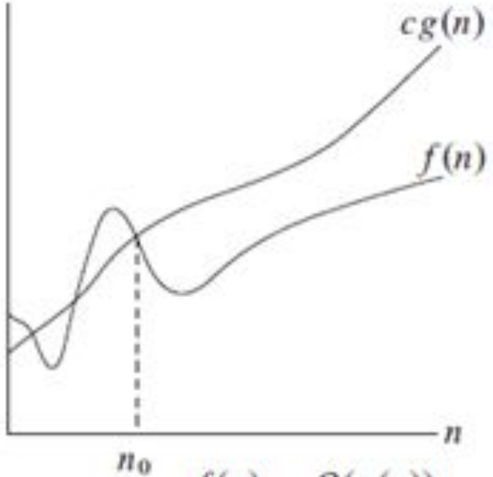
- The efficiency of an algorithm depends on **the amount of time, storage and other resources** required to execute the algorithm.
- **The efficiency is measured with the help of asymptotic notations.**
- An algorithm may not have the same performance for different types of inputs.
- With the increase in the input size, the performance will change.
- For example: x^2, when the value of $x= 10$ and $x=10^{10}$
- The **study of change in performance of the algorithm** with the **change in the order of the input size** is defined as asymptotic analysis.

Asymptotic notations



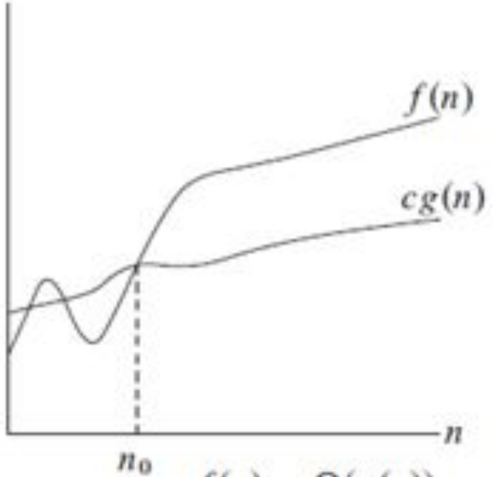
$f(n) = \Theta(g(n))$

(a)



$f(n) = O(g(n))$

(b)



$f(n) = \Omega(g(n))$

(c)

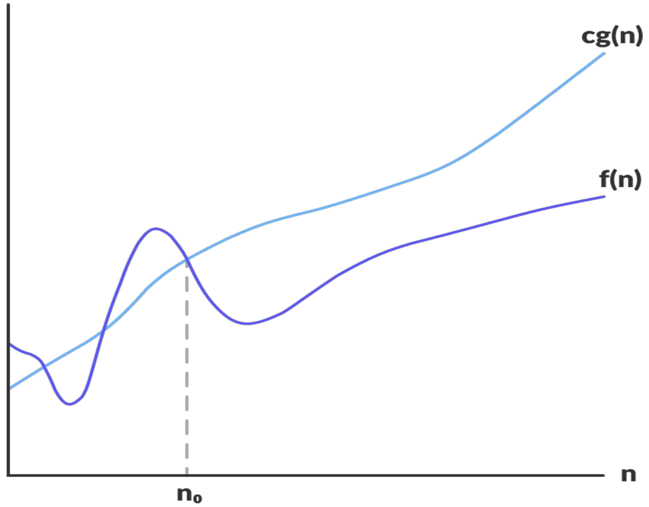
Asymptotic notations

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

- Asymptotic notations are the mathematical notations used to describe the **running time** of an algorithm when the input tends towards a particular value or a limiting value.
- For example: In bubble sort, if input array is already sorted, the time taken by the algorithm is linear i.e. the best case.
- But, when the input array is in reverse condition, the algorithm takes the maximum time (quadratic) to sort the elements i.e. the worst case.
- When the input array is neither sorted nor in reverse order, then it takes average time. These durations are denoted using asymptotic notations.
- There are mainly three asymptotic notations:
    - Big-O notation
    - Omega notation
    - Theta notation

Asymptotic notations

- There are mainly three asymptotic notations:
  - Big-O notation: Big-O notation represents the **upper bound** of the running time of an algorithm. Thus, it gives the **worst-case** complexity of an algorithm.



f(n) = O(g(n))

- $f(n) = O\ g(n)$
- $f(n) <= c.g(n)$, where $c>0$, $n_0 >=0$
- Ex: $f(n) = 2n^2 + n$

  $2n^2 + n = O(\ ??\ )$

  $2n^2 + n <= c.\ g\ (n)$

  $2n^2 + n <= 3(n^2)$

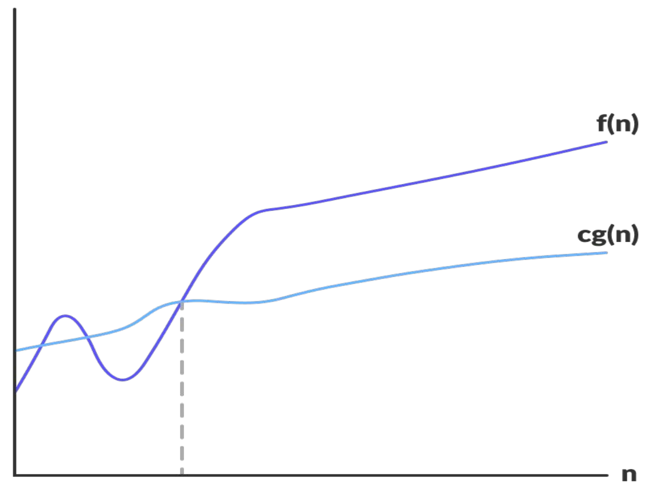
  $n <= n^2$

  $n >= 1$ & $c= 3$
- Put n=5, $2(5^2) + 5 <= 3\ (5^2)$

  55 <= 75, hence, the conduction holds true.

Asymptotic notations

- There are mainly three asymptotic notations:
  - Omega notation: Omega notation represents the **lower bound** of the running time of an algorithm. Thus, it provides the **best case** complexity of an algorithm.



$f(n) = \Omega(g(n))$

- $f(n) = \mathbf{\Omega}\ g(n)$
- $f(n) >= c.g(n)$, where $c>0$, $n_0 >=0$
- Ex: $f(n) = 2n^2 + n$

  $2n^2 + n = \mathbf{\Omega}\ (\ ??\ )$

  $2n^2 + n >= c.\ g\ (n)$

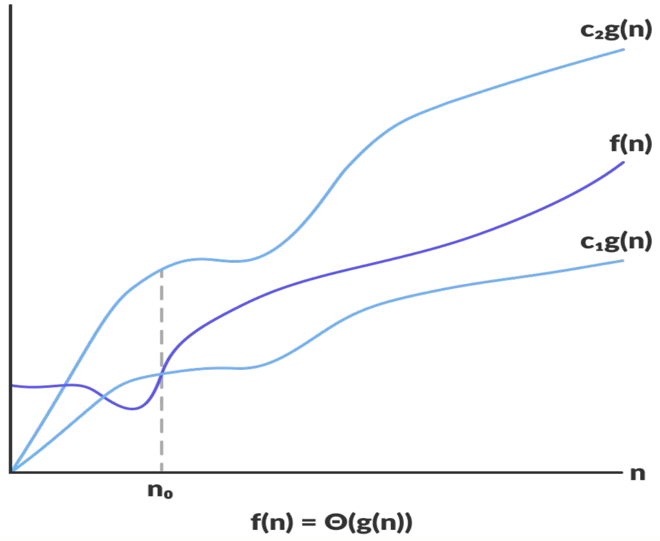
  $2n^2 + n >= 2(n^2)$

  $n >= 0\ \&\ c= 2$
- Put $n=5$, $2(5^2) + 5 >= (5^2)$

  $55 >= 50$, hence, the conduction holds true.

Asymptotic notations

- There are mainly three asymptotic notations:
  - Theta notation: Theta notation encloses the function from above and below. Since it represents **the upper and the lower bound** of the running time of an algorithm, it is used for analyzing the **average-case** complexity of an algorithm.



- f(n) = ϴ g(n)
- c1.g(n) <= f(n) <= c2 .g(n), where c1, c2>0, n_0 >=0
- Ex: f(n) = $2n^2 + n$

  c1.g(n) <= f(n) <= c2 .g(n)

  (2) $n^2 <= 2n^2 + n <=$ (3) n^2

- Put n=5, (2) $n^2 <= 2n^2 + n <=$ (3) n^2

  (2) $5^2 <= 2*5^2 + 5 <=$ (3) 5^2

  50 <= 55 <= 75, hence, the conduction holds true.

Complexity of an Algorithm

- The complexity of an algorithm M is the function f(n) which gives the running time and/or storage space requirement of the algorithm in terms of the size ‘n’ of the input data.
- Mostly, the storage space required by an algorithm is simply a multiple of the data size ‘n’.
- Complexity shall refer to the running time of the algorithm.

Time and Space Complexity

Time and Space Complexity

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

- Analysing an algorithm means determining the amount of resources such as **time** and **memory**

- **Time complexity of an algorithm:**
  It is basically the running time of a program as a function of the input size.
  In other words, the number of machine instructions which a program executes is called its time complexity.
- **Space complexity of an algorithm:**
  It is the amount of computer memory that is required during the program execution as a function of the input size.

- However, **running time** requirements are **more critical** than memory requirements.
- Hence, we will concentrate on the running time efficiency of algorithms.

Worst-case, Average-case, Best-case, and Amortized Time Complexity

Worst-case running time

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

- This denotes the behaviour of an 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.
- Therefore, having the knowledge of worst-case running time gives us an assurance that the algorithm will never go beyond this time limit.

Average-case running time

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

- The average-case running time of an algorithm is an estimate of the running time for an ‘average’ input.
- It specifies the expected behaviour of the algorithm when the input is randomly drawn from a given distribution.
- Average-case running time assumes that all inputs of a given size are equally likely.

Best-case running time

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

- The term ‘best-case performance’ is used to analyse an algorithm under optimal conditions.
- For example, the best case for a simple linear search on an array occurs when the desired element is the first in the list.
- However, while developing and choosing an algorithm to solve a problem, we hardly base our decision on the best-case performance.
- It is always recommended to improve the average performance and the worst-case performance of an algorithm

Amortized running time

- Amortized running time refers to the time required to perform a sequence of (related) operations averaged over all the operations performed.
- Amortized analysis guarantees the average performance of each operation in the worst case.

Time–Space Trade-off

Time Space Trade Off

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

- 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.
- It is a case where an algorithm or program trades increased space usage with decreased time.
- Here, space refers to the data storage consumed in performing a given task (RAM, HDD, etc), and time refers to the time consumed in performing a given task (computation time or response time)
- Need of Time Space Tradeoff:
  - Less time by using more memory
  - By solving a problem in very little space by spending a long time.
- Example involving the concept of Time Space Tradeoff:
  - If data is stored uncompressed,it takes more space but less time.
  - Storing only the source and rendering it as an image everytime the page is requested would be trading time for space. More time used but less space.

Expressing Time and Space Complexity

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

- The time and space complexity can be expressed using a **function f(n)** where n is the input size for a given instance of the problem being solved.
- Expressing the complexity is required when
  - We want to **predict the rate of growth of complexity** as the **input size** of the problem **increases.**
  - There are **multiple algorithms** that find a solution to a given problem and we need to find the algorithm that is most efficient.
- The most widely used notation to express this function f(n) is the **Big O notation.**
- It provides the upper bound for the complexity

Algorithm Efficiency

Expressing Time and Space Complexity

- If a **function is linear** (without any loops or recursions), the efficiency of that algorithm or the running time of that algorithm can be given as **the number of instructions** it contains.
- However, if an algorithm **contains loops**, then the efficiency of that algorithm may vary depending on the **number of loops** and the **running time of each loop** in the algorithm.

Such as:

- Linear Loops.
- Logarithmic Loops
- Nested Loops
  - Linear logarithmic loop
  - Quadratic loop
  - Dependent quadratic loop

Linear Loops

- To calculate the efficiency of an algorithm that has a single loop, we need to first determine the number of times the statements in the loop will be executed.
- This is because the number of iterations is directly proportional to the loop factor.
- Greater the loop factor, more is the number of iterations.
- For example, consider the loop given below:

```
for(i=0;i<100;i++)
        statement block;
```

- Here, 100 is the loop factor.
- We have already said that efficiency is directly proportional to the number of iterations. Hence, the general formula in the case of linear loops may be given as **$f(n) = n$**

Linear Loops

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

- Consider the loop given below:

```
for(i=0;i<100;i+=2)
        statement block;
```

- Here, the number of iterations is half the number of the loop factor.
- So, here the efficiency can be given as **f(n) = n/2**

Logarithmic Loops

- In logarithmic loops, the loop-controlling variable is **either multiplied or divided** during each iteration of the loop.
- For ex,

```
for(i=1;i<1000;i*=2)
        statement block;
```

```
for(i=1000;i>=1;i/=2)
        statement block
```

- Consider the first for loop in which the loop-controlling variable **i is multiplied by 2.**
- The loop will be executed only 10 times and not 1000 times because in each iteration the value of i doubles. Now, consider the second loop in which the loop-controlling variable **i is divided by 2.** In this case also, the loop will be executed 10 times
- **So, the number of iterations is a function of the number by which the loop-controlling variable is divided or multiplied (Here, this value is 2)**
- That is, when **n = 1000**, the number of iterations can be given by **log 1000** which is approximately equal to 10.
- **Efficiency of Log Loops => f(n) = log n**

Nested Loops

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

- Loops that contain loops are known as nested loops
- In order to analyse nested loops, we need to determine the number of iterations each loop completes.
- The total is then obtained as the product of the number of iterations in the inner loop and the number of iterations in the outer loop
- In this case, we analyse the efficiency of the algorithm based on whether it is a **linear logarithmic, quadratic, or dependent quadratic nested loop.**

Nested Loops: Linear logarithmic loop

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

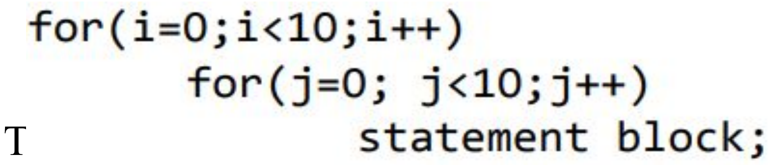
- Consider the following code in which the loop-controlling variable of the inner loop is multiplied after each iteration.

```
for(i=0;i<10;i++)
          for(j=1; j<10;j*=2)
                     statement block;
```

- The number of iterations in the inner loop is log 10.
- This inner loop is controlled by an outer loop which iterates 10 times.
- Therefore, according to the formula, the number of iterations for this code can be given as 10 log 10
- The efficiency of such loops: **f(n) = n log n.**

Nested Loops: Quadratic loop

- In a quadratic loop, the number of iterations in the inner loop is equal to the number of iterations in the outer loop.
- Consider the following code in which the outer loop executes 10 times and for each iteration of the outer loop, the inner loop also executes 10 times.



```
for(i=0;i<10;i++)
        for(j=0; j<10;j++)
                statement block;
```

- T
- The efficiency of such loops: **$f(n) = n^2$**

Nested Loops: Dependent quadratic loop

- In a dependent quadratic loop, the number of iterations in the inner loop is dependent on the outer loop.
- Consider the code given below:

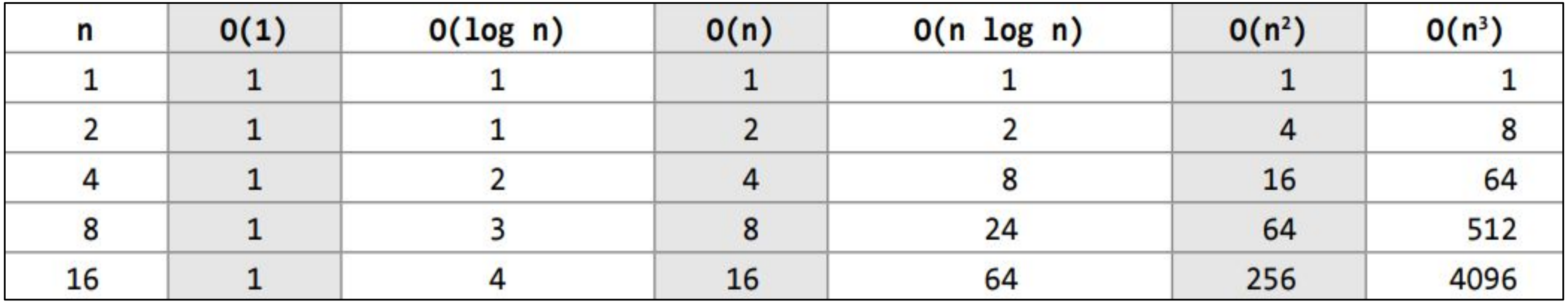
```
for(i=0;i<10;i++)
        for(j=0; j<=i;j++)
                statement block;
```

- In this code, the inner loop will execute just once in the first iteration, twice in the second iteration, thrice in the third iteration, so on and so forth. In this way, the number of iterations can be calculated as 1 + 2 + 3 + ... + 9 + 10 = 55
- If we calculate the average of this loop (55/10 = 5.5), we will observe that it is equal to the number of iterations in the outer loop (10) plus 1 divided by 2.
- The efficiency of such loops: **f(n) = n (n + 1)/2**

Categories of Algorithms

According to the **Big O notation**, we have **five** different categories of algorithms:

- **Constant time** algorithm: running time complexity given as **O(1)**
- **Linear time** algorithm: running time complexity given as **O(n)**
- **Logarithmic time** algorithm: running time complexity given as **O(log n)**
- **Polynomial time** algorithm: running time complexity given as **$O(n^k)$** where k > 1
- **Exponential time** algorithm: running time complexity given as **$O(2^n)$**



n	O(1)	O(log n)	O(n)	O(n log n)	$O(n^2)$	$O(n^3)$
1	1	1	1	1	1	1
2	1	1	2	2	4	8
4	1	2	4	8	16	64
8	1	3	8	24	64	512
16	1	4	16	64	256	4096

Ex:Show that $4n^2 = O(n^3)$.

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

1. By definition, we have $\mathbf{0 \leq h(n) \leq cg(n)}$
2. Substituting $\mathbf{4n^2}$ as $\mathbf{h(n)}$ and $\mathbf{n^3}$ as $\mathbf{g(n)}$, we get
   $\mathbf{0 \leq 4n^2 \leq cn^3}$
3. Dividing by $\mathbf{n^3}$
   $\mathbf{0/n^3 \leq 4n^2/n^3 \leq cn^3/n^3}$ => $\mathbf{0 \leq 4/n \leq c}$
4. Now to determine the value of c, we see that
   $\mathbf{4/n}$ is maximum when $\mathbf{n=1}$. Therefore, $\mathbf{c=4}$.
5. To determine the value of $\mathbf{n_0}$,
   $\mathbf{0 \leq 4/n_0 \leq 4}$ => $\mathbf{0 \leq 4/4 \leq n_0}$ => $\mathbf{0 \leq 1 \leq n_0}$
6. This means $\mathbf{n_0 = 1}$.
7. Therefore, $\mathbf{0 \leq 4n^2 \leq 4n^3,\ \forall\ n \geq n_0 = 1}$

DIY: Examples

1. Show that $n = O(n \log n)$
2. Show that $10n^3 + 20n \neq O(n^2)$.

Show that n = O(n log n)

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

Solution By definition, we have

$0 \le h(n) \le cg(n)$

Substituting n as $h(n)$ and $n \log n$ as $g(n)$, we get

$0 \le n \le c\, n \log n$

Dividing by $n \log n$, we get

$0/n \log n \le n/n \log n \le c\, n \log n / n \log n$

$0 \le 1/\log n \le c$

We know that $1/\log n \longrightarrow 0$ as $n \longrightarrow \infty$

To determine the value of c, it is clearly evident that $1/\log n$ is greatest when $n=2$. Therefore,

$0 \le 1/\log 2 \le c = 1$. Hence $c = 1$.

To determine the value of $n_{0,}$ we can write

$0 \le 1/\log n_0 \le 1$

$0 \le 1 \le \log n_0$

Now, $\log n_0 = 1$, when $n_0 = 2$.

Hence, $0 \le n \le cn \log n$ when $c = 1$ and $\forall\ n \ge n_0 = 2$.

Show that $10n^3 + 20n \neq O(n^2)$

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

Solution By definition, we have

$$0 \leq h(n) \leq cg(n)$$

Substituting $10n^3 + 20n$ as $h(n)$ and n^2 as $g(n)$, we get

$$0 \leq 10n^3 + 20n \leq cn^2$$

Dividing by n^2

$$0/n^2 \leq 10n^3/n^2 + 20n/n^2 \leq cn^2/n^2$$

$$0 \leq 10n + 20/n \leq c$$

$$0 \leq (10n^2 + 20)/n \leq c$$

Hence, $10n^3 + 20n \neq 0^2(n^2)$

ARRAY ADT

Array

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

- An array is a collection of similar data elements.
- These data elements have the same data type.
- The elements of the array are stored in consecutive memory locations and are referenced by an index (also known as the subscript).
- The subscript is an ordinal number which is used to identify an element of the array

DECLARATION OF ARRAYS:

- An array must be declared before being used.
- Declaring an array means specifying the following:
- **Data type**—the kind of values it can store, for example, int, char, float, double.
- **Name**—to identify the array.
- **Size**—the maximum number of values that the array can hold.
- Arrays are declared using the syntax: type name[size];

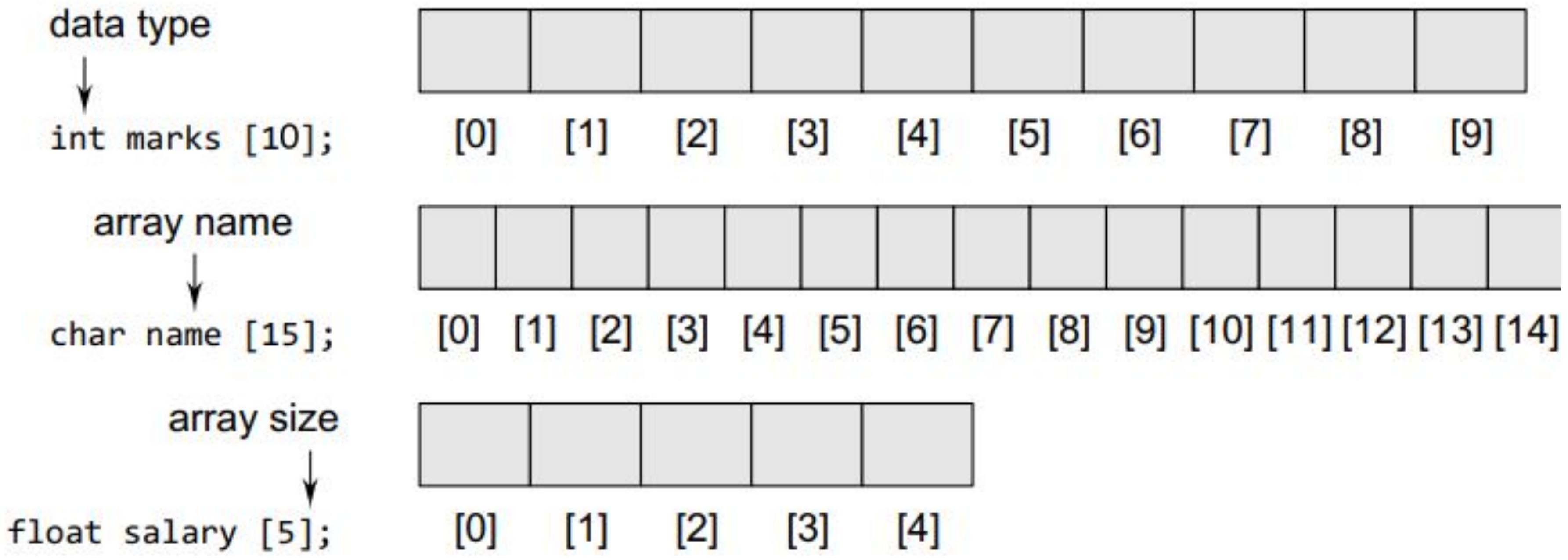
Memory representation of an array of 10 elements

- For example, if we write, int marks[10];
  then the statement declares marks to be an array containing 10 elements.
- In C, the array index starts from zero.
- The first element will be stored in marks[0], second element in marks[1], and so on.
- Therefore, the last element, that is the 10th element, will be stored in marks[9].
- Note that 0, 1, 2, 3 written within square brackets are the subscripts.

1st element	2nd element	3rd element	4th element	5th element	6th element	7th element	8th element	9th element	10th element
marks[0]	marks[1]	marks[2]	marks[3]	marks[4]	marks[5]	marks[6]	marks[7]	marks[8]	marks[9]

Declaring arrays of different data types and sizes

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

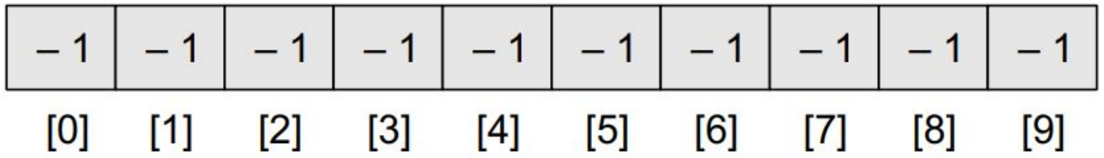


ACCESSING THE ELEMENTS OF AN ARRAY

- For example, write a C program to initialize each element of the array to –1.
  Consider marks[10]

```
int i, marks[10];
for(i=0;i<10;i++)
        marks[i] = -1;
```

- Array marks after executing the code:



Array Address Calculation

Calculating the Address of Array Elements

- **Address of data element,**
  A[k] = BA(A) + w(i – lower_bound)
- Here,
  A is the array,
  i is the index of the element of which we have to calculate the address,
  BA is the base address of the array A, and
  w is the size of one element in memory,
  for example, size of int is 2.

Ex: Calculating the Address of Array Elements

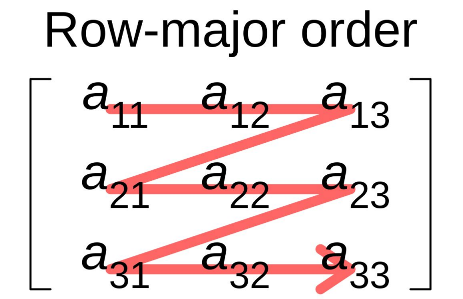
- Given an array int marks[]={99,67,78,56,88,90,34,85}, calculate the address of marks[4] if the base address = 1000.

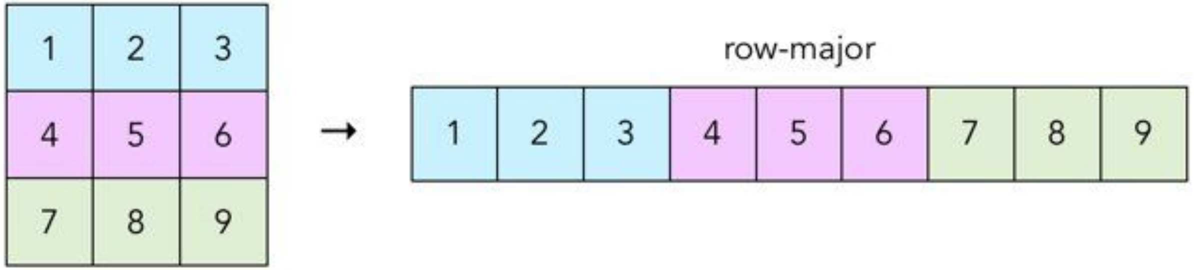
99	67	78	56	**88**	90	34	85
marks[0]	marks[1]	marks[2]	marks[3]	**marks[4]**	marks[5]	marks[6]	marks[7]
1000	1002	1004	1006	**1008**	1010	1012	1014

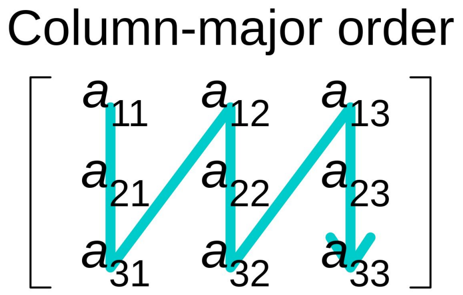
- We know that storing an integer value requires 2 bytes, therefore, its size is 2 bytes.
- Formula: **A[k] = BA(A) + w(i – lower_bound)**
- marks[4] = 1000 + 2(4 – 0) = 1000 + 2(4) = 1008

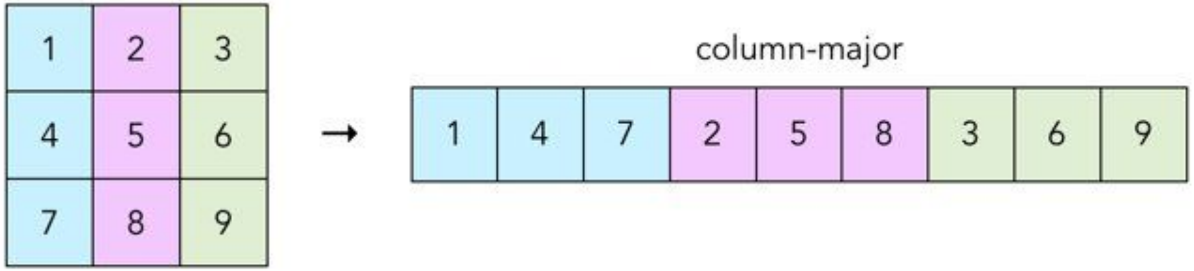
Array ADT: Representations

- Row-major: All the elements are stored in the memory row-wise
- Column-major: All the elements are stored in the memory column-wise


Row-major order
a_{11} a_{12} a_{13}
a_{21} a_{22} a_{23}
a_{31} a_{32} a_{33}




Column-major order
a_{11} a_{12} a_{13}
a_{21} a_{22} a_{23}
a_{31} a_{32} a_{33}



Array ADT: Address of the element

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

- **In 1D Array:**
  - Address of A [i] = Base Address + i * w = **l_0 + i * w**

Calculating the Length of an Array

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

- The length of an array is given by the number of elements stored in it.
- The general formula to calculate the length of an array is

  Length = upper_bound – lower_bound + 1

  where, upper_bound is the index of the last element

  lower_bound is the index of the first element in the array.
- Ex: Let Age[5] be an array of integers such that

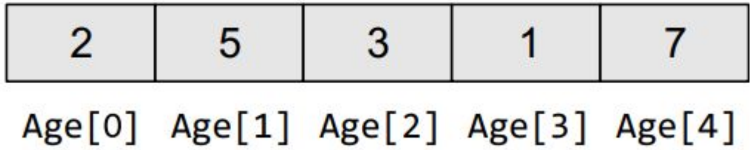
  Age[0] = 2, Age[1] = 5, Age[2] = 3, Age[3] = 1, Age[4] = 7

  Show the memory representation of the array and calculate its length.
- The memory representation of the array Age[5] is given as below:

  Length = upper_bound – lower_bound + 1

  Here, lower_bound = 0, upper_bound = 4.

  Therefore, length = 4 – 0 + 1 = 5



Array ADT: Address of the element: Examples

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

Case 1: When the array is zero-indexed (starting from 0 to MAX-1, Length = MAX)

Given the base address of an array A[19, 22, 23, 24, 56] as 10 and the size of each element is 2 bytes in the memory, find the address of A[3].

Answer:

- Given: Base address B = 10 (**l_0**) (A_0)
- Storage size of one element store in any array w = 2 Byte (**w**)
- Subset of element whose address to be found i= 3
- Formula used: Address of A [i] = Base Address + i * w = **l_0 + i * w**
- Address of A[3] = 10 * (3*2) = 16

Array ADT: Address of the element: Examples

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

Case 2: When the array is non-indexed (starting from a to b, example: A[1:10])

The following details are available about an array RESULT.

Find the address of RESULT[17].

Base address : 520, Index range : 1:20, Array type : Real, Size of the memory location : 4 bytes.

Answer:

- Given: Base address B = 520 (l_0) (A_0)
- Lower Bound, i.e, **x =1** and Upper Bound = 20. (Length = 20)
- Storage size of one element store in any array w = 4 Byte (**w**)
- Subset of element whose address to be found i= 17
- Formula used: Address of RESULT [i] = Base Address + (i-x)* w = l_0 **+ (i -x)* w**
- Address of RESULT[17] = 520 + **(i - lower bound of array)** * 4)
  = 520 + ((17- 1) * 4))
  = 584

Array ADT: Address of the element

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

- **In 2D array of size MxN (M rows, N cols) :**
  - **Row-Major Order** (Lexicographic Order) A[i,j] = BA + size [(i - LBR)*N + (j - LBC)]
  - **Col-Major Order** (Co-lexicographic Order) A[i,j] = BA + size [(i - LBR) + (j - LBC)*M]
  - Find A[2,3] in RMO and CMO. (144 & 156)

A[4,4]=[0:3, 0:3]	c0	c1	c2	c3
r0	a (0,0)	b(0,1)	c(0,2)	d(0,3)
r1	e(1,0)	f(1,1)	g(1,2)	h(1,3)
r2	i(2,0)	j(2,1)	k(2,2)	l(2,3)
r3	m(3,0)	n(3,1)	o(3,2)	p(3,3)

Add	100	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60
RMO	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p
CMO	a	e	i	m	b	f	j	n	c	g	k	o	d	h	l	p

Example:

- **Row-Major Order** (Lexicographic Order) A[i,j] = BA + size [(i - LBR)*N + (j - LBC)]
- **Col-Major Order** (Co-lexicographic Order) A[i,j] = BA + size [(i - LBR) + (j - LBC)*M]
- **Find A[2,3] in RMO and CMO. (144 & 156)**

  BA = 100, size = 4, MxN = [4,4] => arr[0:3, 0:3] (LBR = 0, LBC = 0)
- RMO = BA + size [(i - LBR)*N + (j - LBC)] = 100 + 4 [(2-0)*4 + (3-0)] = 100 + 4 (8+3) = 144
- CMO = BA + size [(i - LBR) + (j - LBC)*M] = 100 + 4 [(2-0)+(3-0)*4] = 100 + 4 (2+12) = 156

- Find A [3,1] in RMO and CMO ?

Example: 2D array address calculations

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

Consider, Arr [25,4] is an array with base value 200 & w=4 for this array. Calculate the address of Arr[12,3] using row-major order and using column-major order.

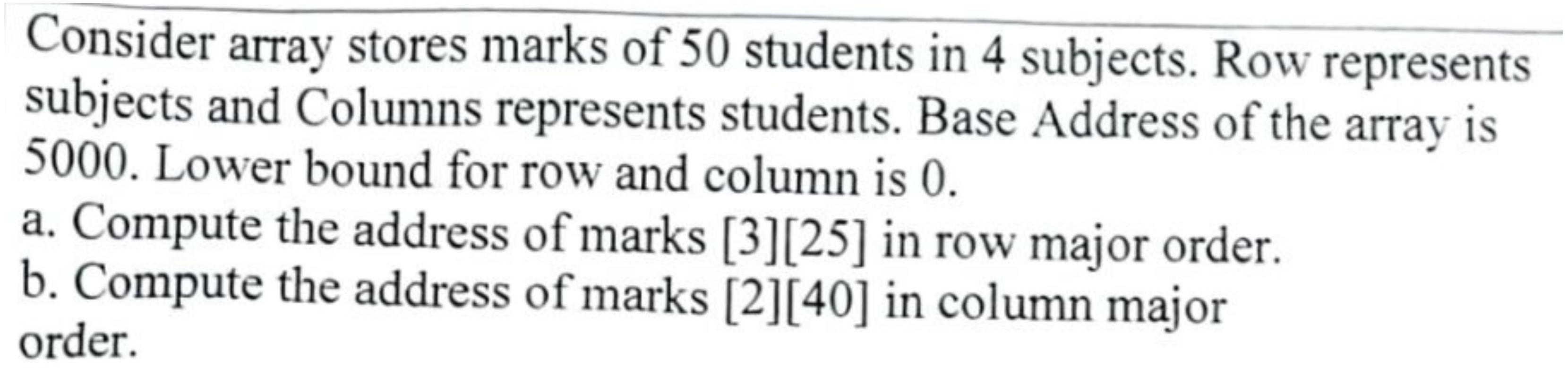
- **In 2D array of size MxN (M rows, N cols) :**
  - **Row-Major Order** (Lexicographic Order) A[i,j] = BA + size [(i - LBR)*N + (j - LBC)]
  - **Col-Major Order** (Co-lexicographic Order) A[i,j] = BA + size [(i - LBR) + (j - LBC)*M]
  - Given, BA = 200, size = 4, arr[0:24, 0:3], LBR=0, LBC=0, M=25, N=4

RMO = Arr[12,3] = 200 + 4 [(12-0)*4 + (3-0)] = 200 + 4 (48+3) = 404

CMO = Arr[12,3] = 200 + 4 [(12-0) + (3-0)*25] = 200 + 4 (12+75) =548

Example: 2D array address calculations

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

- An array X [-15……….10, 15……………40] requires one byte of storage. If beginning location is 1500 determine the location of X [15][20]. (2285,1660)
- Consider a zero-indexed array A with dimensions 5 x 6 x 7 that stores 4-byte integers is located at an address 6400. Calculate the address of A (4, 5, 6) using lexicographic order (row-major order).
- 

Consider array stores marks of 50 students in 4 subjects. Row represents subjects and Columns represents students. Base Address of the array is 5000. Lower bound for row and column is 0.
a. Compute the address of marks [3][25] in row major order.
b. Compute the address of marks [2][40] in column major order.

Array ADT: 3D Array Address Calculations

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

In any 3D array, the array has three dimensions: **1) Blocks (Depth) 2) Rows 3) Cols**

Consider a non-zero indexed array such as arr[1:10, 2:15, 3:14], then total number of
a) Block = (upper limit - lower limit + 1) = 10 -1 + 1 = 10 Blocks/ Depth
b) Rows = (upper limit - lower limit + 1) = 15 - 2 + 1 = 14 Rows (The value of M)
c) Columns = (upper limit - lower limit + 1) = 14 - 3 + 1 =12 Columns (The value of N)

Consider a zero indexed array such as arr[0:15, 0:12, 0:18], then total number of
a) Blocks = (upper limit - lower limit + 1) = 15-0+1 = 16 Blocks/ Depth
b) Rows = (upper limit - lower limit + 1) = 12-0+1 = 13 Rows (The value of M)
c) Columns = (upper limit - lower limit + 1) = 18-0+1 =19 Columns (The value of N)

Array ADT: 3D Array Address Calculations

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

Formula for Row-Major: (Lexicographical Order)

Arr[i,j,k] = BA + size [M * N (i - LBR) + (j - LBC) + N (k - LBD)]

Formula for Column-Major: (Co-lexicographical Order)

Arr[i,j,k] = BA + size [(i - LBR) + M * (j - LBC) + M * N (k - LBD)]

Where,

i,j,k = dimensions of the array to be calculated
(such as find the address of arr[5][-1][8], then i = 5, j = -1, k = 8)

w (size) = memory size of the element

M = Total number of rows

N = Total number of Columns

LBD, LBR, LBC = lower bound limit of row, column, depth
(such as A[1:9, -5:2, 9:12],
then the lower bound limits are LBD = 1, LBR = -5, LBC = 9

Array ADT: 3D Array Address Calculations

Formula for Row-Major: (Lexicographical Order)

Arr[i,j,k] = BA + size [M * N (i - LBR) + (j - LBC) + N (k - LBD)]

Formula for Column-Major: (Co-lexicographical Order)

Arr[i,j,k] = BA + size [(i - LBR) + M * (j - LBC) + M * N (k - LBD)]

Ex: Given an array arr[1:8, -5:5, -10:5] with a base value of 400 and the size of each element is 4 Bytes in memory find the address of element arr[3][3][3] with the help of row-major order & column-major order?

RMO of arr[3][3][3] = 400 + 4 {11*16(3+5) + (3+10) + 16(3-1)} = 400 + 4 (1408+13+32) = 6212

CMO of arr[3][3][3] = 400 + 4* {(3+5) + 11 (3+10) + (11*16)(3-1)} = 400 + 4 (8+143+352) = 2412

Example: 3D array address calculations

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

- Suppose a 3–dimensional array A is defined as below—A[2:8, –6:3, -4:8]. Size of each element in A is 8 and base address of A is 1800: (1) Compute the address of the element stored at A [7, –3, -2] using RMO form (2) Compute the Base Address if the address of the element stored at A [0, 0, 0] using CMO form is 5560.

Given: Depth (D) = 8-2+1 = 7 Rows (M) = 3+6+1=10 Cols(N) = 8+4+1 = 13
BA = 1800 size = 8 LBD=2 LBR=-6 LBC=-4

1) RMO of A[7,-3,-2] = BA + size {M*N (i-LBR)+(j-LBC)+N*(k-LBD)}
= 1800 + 8 {10*13 (7+6) + (-3+4) + 13*(-2-2) }
= 14912

2) CMO of A[0,0,0] = BA + size {(i-LBR)+M*(j-LBC)+M*N (k-LBD)}
5560 = BA + 8{(0+6)+10*(0+4)+10*13(0-2)}
5560 = BA + 8 {6 + 40 -260 }
5560 = BA - 1712
BA = 7272

Example: 3D array address calculations

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

- Suppose a 3 – dimensional array A is defined as below —
- A[2 : 6, –3 : 2, 4 : 7]. Size of each element in A is 2 and the base address of A is 3200 :
  1) Compute the address of the element stored at A [4, -1, 5] using RMO form.
  2) Compute the address of the element stored at A [5, 1, 6] using CMO form.

Given: Depth (D) = 6-2+1=5 Rows (M)= 2+3+1= 6 Cols(N) = 7-4+1=4
BA = 3200 size = 2 LBD=2 LBR=-3 LBC=4

1) RMO of A[4, -1, 5]= BA + size {M*N (i-LBR)+(j-LBC)+N*(k-LBD)}
= 3200 + 2 {6*4 (4-2) + (-1-4) + 4*(5-2) }
= 3200 + 2 {48-5+12}= 3310

2) CMO of A[5,1,6] = BA + size {(i-LBR)+M*(j-LBC)+M*N (k-LBD)}
= 3200 + 2{(5+3)+6*(1-4)+6*4(6-2)}
= 3200 + 2 {8 -18 + 96} = 3372

Array ADT: 3D Array Address Calculations

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

Example: Given an array [1..8, 1..5, 1..7] of integers and memory size of 2. Calculate address of element A[5,3,6], by using rows and columns methods, if BA=900?

RMO of A[5,3,6] = 1254 CMO of A[5,3,6] = 1278

OPERATIONS ON ARRAYS

Operations on Array

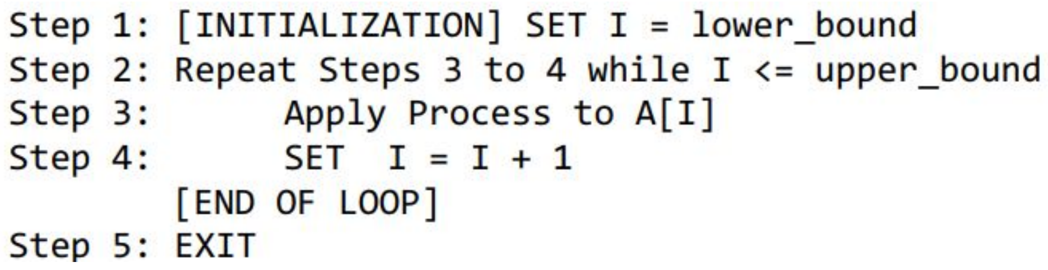
RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

The array operations include:

- Traversing an array
- Inserting an element in an array
- Searching an element in an array
- Deleting an element from an array
- Merging two arrays
- Sorting an array in ascending or descending order

Traversing an Array

- Traversing an array means accessing each and every element of the array for a specific purpose.
- Traversing the data elements of an array A can include printing every element, counting the
- total number of elements, or performing any process on these elements.
- Since, array is a linear data structure (because all its elements form a sequence), traversing its elements is very simple and straightforward



```
Step 1: [INITIALIZATION] SET I = lower_bound
Step 2: Repeat Steps 3 to 4 while I <= upper_bound
Step 3:      Apply Process to A[I]
Step 4:      SET  I = I + 1
        [END OF LOOP]
Step 5: EXIT
```

Traversing an Array

- In Step 1, we initialize the index to the lower bound of the array.
- In Step 2, a while loop is executed.
- Step 3 processes the individual array element as specified by the array name and index value.
- Step 4 increments the index value so that the next array element could be processed. The while loop in Step 2 is executed until all the elements in the array are processed, i.e., until I is less than or equal to the upper bound of the array

```
Step 1: [INITIALIZATION] SET I = lower_bound
Step 2: Repeat Steps 3 to 4 while I <= upper_bound
Step 3:         Apply Process to A[I]
Step 4:         SET  I = I + 1
        [END OF LOOP]
Step 5: EXIT
```

Practice Exercises

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

- Write a program to read and display n numbers using an array
- Write a program to find the mean of n numbers using arrays.
- Write a program to print the position of the smallest number of n numbers using arrays
- Write a program to find the second largest of n numbers using an array
- Write a program to enter n number of digits. Form a number using these digits
- Write a program to find whether the array of integers contains a duplicate number.

Question 1 Code

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

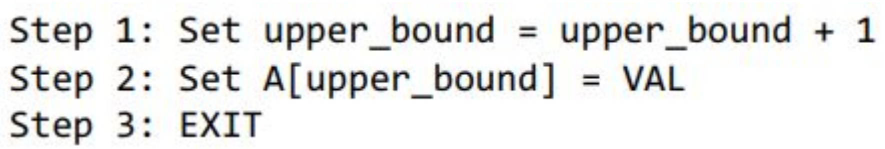
```
#include <stdio.h>
#include <conio.h>
void main() {
int i, n, arr[20];
clrscr();
printf("\n Enter the number of elements in the array : ");
scanf("%d", &n);
for(i=0;i<n;i++) {
printf("\n arr[%d] = ", i);
scanf("%d",&arr[i]);
}
printf("\n The array elements are ");
for(i=0;i<n;i++)
printf("\t %d", arr[i]);
}
```

Output

```
Enter the number of elements in the array : 5
arr[0] = 1
arr[1] = 2
arr[2] = 3
arr[3] = 4
arr[4] = 5
The array elements are 1 2 3 4 5
```

Inserting an Element in an Array

- If an element has to be inserted at the end of an existing array, then the task of insertion is quite simple.
- We just have to add 1 to the upper_ bound and assign the value



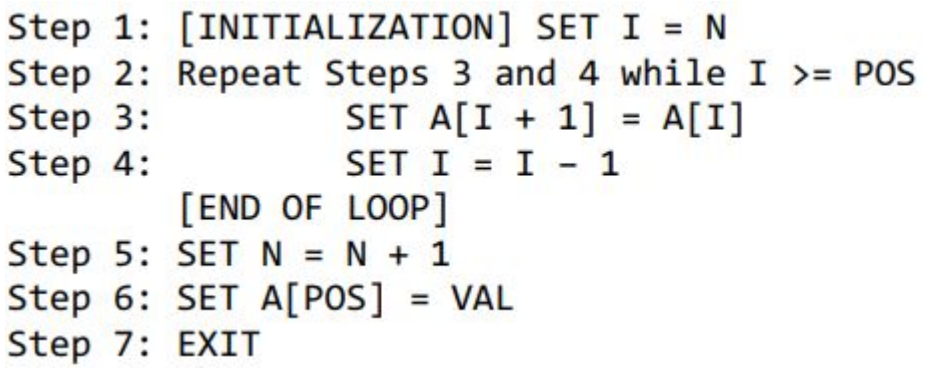
```
Step 1: Set upper_bound = upper_bound + 1
Step 2: Set A[upper_bound] = VAL
Step 3: EXIT
```

Practice Example:

- Data[] is an array that is declared as int Data[20]; and contains the following values:
  Data[] = {12, 23, 34, 45, 56, 67, 78, 89, 90, 100};
  (a) Calculate the length of the array.
  (b) Find the upper_bound and lower_bound.
  (c) Show the memory representation of the array.
  (d) If a new data element with the value 75 has to be inserted, find its position.
  (e) Insert a new data element 75 and show the memory representation after the insertion.

Insert an Element at any position of an Array

- The algorithm INSERT will be declared as INSERT (A, N, POS, VAL). The arguments are
  (a) A, the array in which the element has to be inserted
  (b) N, the number of elements in the array
  (c) POS, the position at which the element has to be inserted
  (d) VAL, the value that has to be inserted



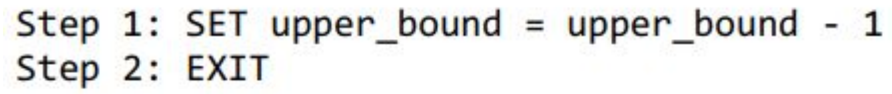
```
Step 1: [INITIALIZATION] SET I = N
Step 2: Repeat Steps 3 and 4 while I >= POS
Step 3:         SET A[I + 1] = A[I]
Step 4:         SET I = I - 1
        [END OF LOOP]
Step 5: SET N = N + 1
Step 6: SET A[POS] = VAL
Step 7: EXIT
```

Practice Example:

- Write a program to insert a number at a given location in an array.
- Write a program to insert a number in an array that is already sorted in ascending order.

Deleting an Element from an Array

- Deleting an element from an array means removing a data element from an already existing array.
- If the element has to be deleted from the end of the existing array, then the task of deletion is quite simple. We just have to subtract 1 from the upper_bound



```
Step 1: SET upper_bound = upper_bound - 1
Step 2: EXIT
```

Deleting an Element from the middle of an array

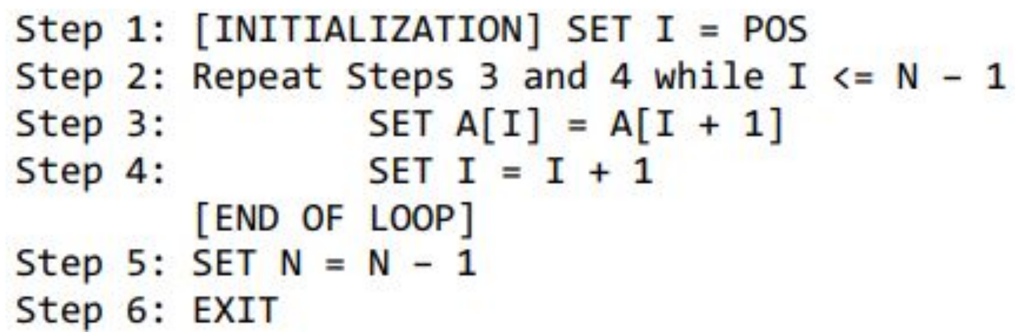
RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

- The algorithm DELETE will be declared as DELETE(A, N, POS).
- The arguments are:

  (a) A, the array from which the element has to be deleted

  (b) N, the number of elements in the array

  (c) POS, the position from which the element has to be deleted



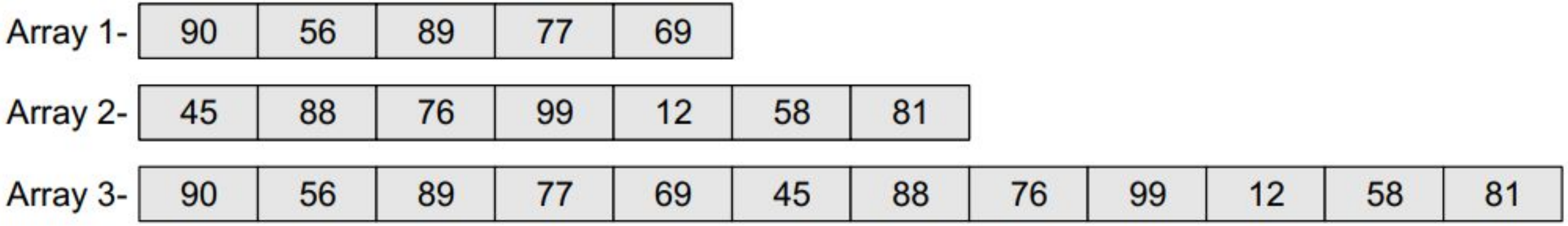
```
Step 1: [INITIALIZATION] SET I = POS
Step 2: Repeat Steps 3 and 4 while I <= N - 1
Step 3:         SET A[I] = A[I + 1]
Step 4:         SET I = I + 1
        [END OF LOOP]
Step 5: SET N = N - 1
Step 6: EXIT
```

Practice Example:

- Write a program to delete a number from a given location in an array
- Write a program to delete a number from an array that is already sorted in ascending order

Merging Two Arrays

- Merging two arrays in a third array means first copying the contents of the first array into the third array and then copying the contents of the second array into the third array. Hence, the merged array contains the contents of the first array followed by the contents of the second array.
- If the arrays are unsorted, then merging the arrays is very simple, as one just needs to copy the contents of one array into another.
- But merging is not a trivial task when the two arrays are sorted and the merged array also needs to be sorted.
- Let us first discuss the merge operation on unsorted arrays

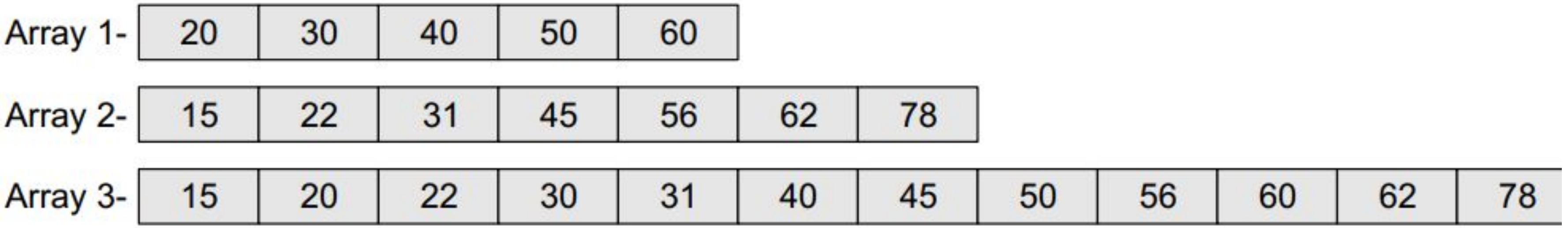


Array 1-	90	56	89	77	69							
Array 2-	45	88	76	99	12	58	81					
Array 3-	90	56	89	77	69	45	88	76	99	12	58	81

Merging Two Arrays

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

- If we have two sorted arrays and the resultant merged array also needs to be a sorted one, then the task of merging the arrays becomes a little difficult.
- Here, compare the 1st element of array1 with the 1st element of array2, and then put the smaller element in the merged array.
- Since 20 > 15, we put 15 as the first element in the merged array.
- We then compare the 2nd element of the second array with the 1st element of the first array. Since 20 < 22, now 20 is stored as the second element of the merged array.
- Next, the 2nd element of the first array is compared with the 2nd element of the second array. Since 30 > 22, we store 22 as the third element of the merged array.



Array 1-	20	30	40	50	60							
Array 2-	15	22	31	45	56	62	78					
Array 3-	15	20	22	30	31	40	45	50	56	60	62	78

Merging Two Arrays

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

- Now, we will compare the 2nd element of the first array with the 3rd element of the second array.
- Because 30 < 31, we store 30 as the 4th element of the merged array.
- This procedure will be repeated until elements of both the arrays are placed in the right location in the merged array.
- **NOW: Write a program to merge two sorted arrays.**

Array 1-	20	30	40	50	60							
Array 2-	15	22	31	45	56	62	78					
Array 3-	15	20	22	30	31	40	45	50	56	60	62	78

ARRAYS OF STRUCTURES

Arrays of Structures

- **Scenario 1:** In a class, we do not have just one student. But there may be at least 30 students. So, the same definition of the structure can be used for all the 30 students. This would be possible when we make an array of structures. An array of structures is declared in the same way as we declare an array of a built-in data type
- **Scenario 2:** Another example where an array of structures is desirable is in case of an organization. An organization has a number of employees. So, defining a separate structure for every employee is not a viable solution. So, here we can have a common structure definition for all the employees. This can again be done by declaring an array of structure employee

Arrays of Structures: Syntax

- The general syntax for declaring an array of structures:

```
struct struct_name
{
data_type member_name1;
data_type member_name2;
data_type member_name3;
........................
};
struct struct_name struct_var[index];
```

Arrays of Structures: Syntax

- Consider the given structure definition.

```
struct student{
        int r_no;
        char name[20];
        char course[20];
        float fees;
};
```

- A student array can be declared by writing, struct student stud[30];
- Now, to assign values to the ith student of the class, we will write

```
stud[i].r_no = 09;
stud[i].name = "RASHI";
stud[i].course = "MCA";
stud[i].fees = 60000;
```

Arrays of Structures: Syntax

RCOEM
Shri Ramdeobaba College of Engineering and Management, Nagpur

- In order to initialize the array of structure variables at the time of declaration, we can write as follows:

```
struct student stud[3] = {
                    {01,  "Aman", "BCA", 45000},
                    {02,  "Aryan", "BCA", 60000},
                    {03,  "John", "BCA", 45000}
                };
```

- **DIY: Write a program to read and display the information of all the students in a class. Then edit the details of the ith student and redisplay the entire information.**

End of Unit I