# DATA STRUCTURE USING C (KCS 301)

**By:**
**Dr. Sunil Kumar**
**Professor, CSE Dept.**
**MIET, Meerut**

## Data Structure Using C (KCS301)

- **UNIT 1:**
  - Part 1: Introduction
  - Part 2: Arrays
  - Part 3: Linked List
- **UNIT 2:**
  - Part 1: Stacks
  - Part 2: Queues
- **UNIT 3:**
  - Part 1: Searching
  - Part 2: Sorting
- **UNIT 4:** Graphs
- **UNIT 5:** Trees

# Course Outcomes

After studying this course, students will be able to:

- Acquire knowledge of:
  - Various types of data structures, operations and algorithms.
  - Sorting and searching operations
- Analyse the performance of
  - Stack, Queue, Linked Lists, Trees, Graphs
  - Searching and Sorting techniques
- Implement all the applications of Data structures in a high-level language
- Design and apply appropriate data structures for solving computing problems.



# Books

**Text Book:**

- Seymour Lipschutz, "Data Structures" Schaum's Outline Series, Tata McGraw-hill Education (India) Pvt. Ltd.

**Reference Books:**

- A.K. Sharma, "Data Structure Using C", Pearson Education India.
- Aaron M. Tenenbaum, Yedidyah Langsam and Moshe J. Augenstein, "Data Structures Using C and C++", PHI Learning Private Limited, Delhi India
- Horowitz and Sahani, "Fundamentals of Data Structures", Galgotia Publications Pvt Ltd Delhi India.
- Thareja, "Data Structure Using C" Oxford Higher Education.

# Classification of Data Structures

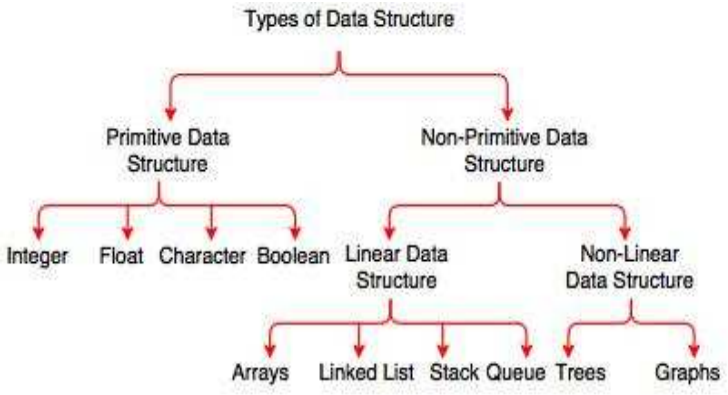



Fig. Types of Data Structure



# Primitive and Non-Primitive Data Types

- **PRIMITIVE DATATYPES:** The primitive data types are the basic data types that are available in most of the programming languages. The primitive data types are used to represent single values. The primitive data types are:
  - **Integer**
  - **Float**
  - **Double**
  - **Character**
  - **String**
  - **Boolean etc**
- **NON-PRIMITIVE DATATYPES**: The data types that are derived from primary data types are known as non-Primitive data types. These data types are used to store group of values. The non-primitive data types are:
  - **Arrays**
  - **Structure**
  - **Union**
  - **Linked list**
  - **Stacks**
  - **Queue etc**

# Types of Data Structure

- Based on the organizing method of data structure, data structures are divided into two types.
  - **Linear Data Structures:** If a data structure organizes the data in sequential order, then that data structure is called a Linear Data Structure.
  - **Examples are:**
    - Arrays
    - Linked List
    - Stacks
    - Queues
  - **Non - Linear Data Structures:** If a data structure organizes the data in random order, then that data structure is called as Non-Linear Data Structure.
  - **Examples are:**
    - Trees
    - Graphs
    - Dictionaries
    - Heaps



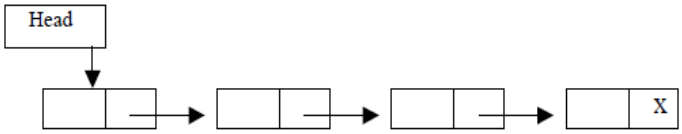
# Description of linear and non linear data structures

- **Arrays:** An array is a collection of homogeneous data elements described by a single name.
- Each element of an array is referenced by a subscripted variable or value, called subscript or index enclosed in parenthesis.
- If an element of an array is referenced by single subscript, then the array is known as one dimensional array or linear array and if two subscripts are required to reference an element, the array is known as two dimensional arrays and so on.

| A[0] | A[1] | A[2] | ..... | A[9] |
|---|---|---|---|---|

# Continue

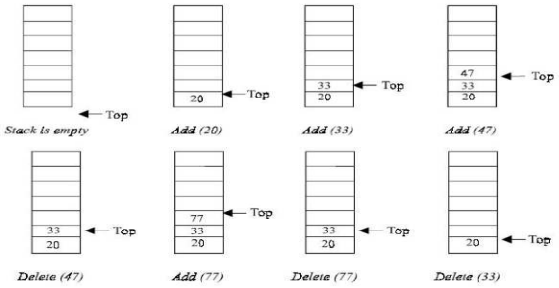
- **Linked List:** A linked list is an ordered set consisting of a varying number of elements to which insertion and deletion can be made.
- List can be implemented by using pointers.
- Each element is referred to as nodes; therefore a list can be defined as a collection of nodes as shown below :





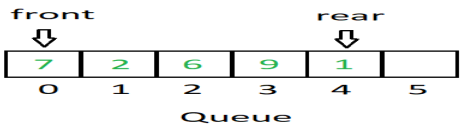

# Continue

- **Stacks:** It is an ordered collection of items into which new data items may be added / inserted and from which items may be deleted at only one end, called the top of the stack.
- As all the addition and deletion in a stack is done from the top of the stack, the last added element will be first removed from the stack.
- That is why the **stack is also called Last-in-First-out (LIFO)**.



# Continue

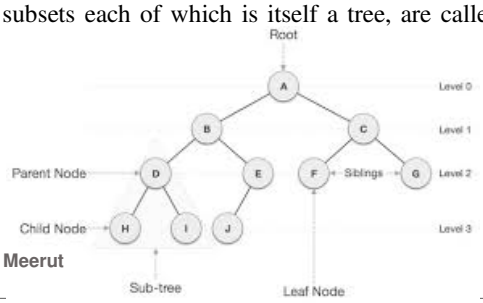
- **Queues:** A queue is logically a first in first out (**FIFO or first come first serve)** linear data structure.
- It is a homogeneous collection of elements in which new elements are added at one end called rear, and the existing elements are deleted from other end called front. The basic operations that can be performed on queue are:
  - 1. Insert (or add) an element to the queue (push)
  - 2. Delete (or remove) an element from a queue (pop).






# Continue

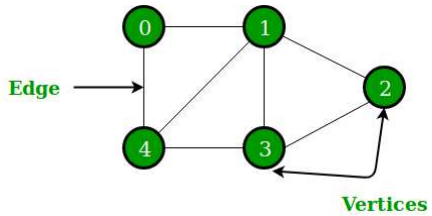
- **Trees:** Many real life problems can be represented and solved using trees.
- Trees are very flexible, versatile and powerful non-liner data structure.
- A tree is an ideal data structure for representing hierarchical data.
- A tree can be theoretically defined as a finite set of one or more data items (or nodes) such that:
- There is a special node called the root of the tree.
- Removing nodes (or data item) are partitioned into number of mutually exclusive (i.e., disjoined) subsets each of which is itself a tree, are called sub tree.



# Continue

- **Graphs:** A Graph is a non-linear data structure consisting of nodes and edges.
- The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph.
- More formally a Graph can be defined as,

  *"A Graph consists of a finite set of vertices(or nodes) and set of Edges which connect a pair of nodes."*




In the above Graph, the set of vertices V = {0,1,2,3,4} and the set of edges E = {01, 12, 23, 34, 04, 14, 13}.

# UNIT 1 - Part I: Introduction

## Table of Contents

# Introduction to Data Structures

- **Data Structure = Organised Data + Allowed Operations**

**"Data structure is a method of organizing a large amount of data more efficiently so that any operation on that data becomes easy."**

**In other words, "Data structure is the logical or mathematical model of a particular organization."**

## Selection of Data Structure

- There are many considerations to be taken into account when choosing the best data structure for a specific program
  - Size of data
  - Speed and manner data use
  - Data dynamics, as change and edit
  - Size of required storage
  - Fetch time of any information from data structure



## Need of Data Structure

- **Processor Speed:** To handle very large amount of data, high speed processing is required, but as the data is growing day by day to the billions of files per entity, processor may fail to deal with that much amount of data.
- **Data Search:** Consider an inventory size of 106 items in a store, If our application needs to search for a particular item, it needs to traverse 106 items every time, results in slowing down the search process.
- **Multiple Requests:** If thousands of users are searching the data simultaneously on a web server, then there are the chances that a very large server can be failed during that process.

# Characteristics of a Data Structure

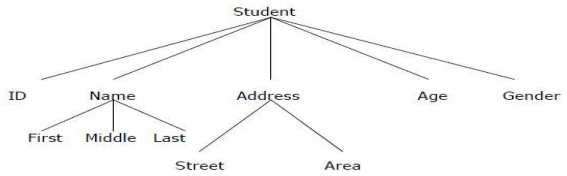
- **Correctness** – Data Structure implementation should implement its interface correctly.
- **Time Complexity** – Running time or execution time of operations of data structure must be as small as possible.
- **Space Complexity** – Memory usage of a data structure operation should be as little as possible.



Dr. Sunil Kumar, CSE Dept., MIET Meerut

## Basic Terminology: Elementary Data Organization

- **Data –** Data are values or set of values or collection of facts and figures.
- **Data Item –** Data item refers to single unit of values.
- **Group Items –** Data item that are divided into sub items are called as Group Items.
- For example, a student's name may be divided into three sub items – [first name, middle name and last name] but the ID of a student would normally be treated as a single item.




In the above example ( ID, Age, Gender, First, Middle, Last, Street, Area ) are elementary data items, whereas (Name, Address ) are group data items.



## Basic Terminology: Elementary Data Organization…

- **Data Types –** Data type is a classification identifying one of various types of data i.e. int, float, character etc.
- **Variable –** A variable is a quantity whose value can change.
- **Record –** Collection of related data items.
- **Entity –** An entity is something that has certain attributes or properties which may be assigned some values. Example:

| Attributes: | Name | Age | Gender | Social Society number |
|---|---|---|---|---|
| Values: | Hamza | 20 | M | 134-24-5533 |
| | Ali Rizwan | 23 | M | 234-9988775 |
| | Fatima | 20 | F | 345-7766443 |

- **Entity Set –** Collection of entities or set of similar entities.
- **Field –** Field is a single elementary unit of information representing an attribute of an entity.
- **File –** File is a collection of records in a given entity set.

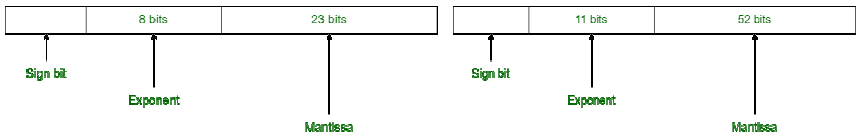
# Built-in Data Types in C

- Each variable in C language has an associated data type.
- Each data type requires different amounts of memory and has some specific operations which can be performed over it.



# Data Types Examples

- Examples of some very common data types used in C:
  - **Char:** The most basic data type in C. It stores a single character and requires a single byte of memory in almost all compilers.
  - **Int:** As the name suggests, an int variable is used to store an integer.
  - **Float:** It is used to store decimal numbers (numbers with floating point value) with single precision.
  - **Double:** It is used to store decimal numbers (numbers with floating point value) with double precision.



## Continue

- Different data types also have different ranges up to which they can store numbers.
- These ranges may vary from compiler to compiler.
- List of ranges along with the memory requirement and format specifiers on 32-bit gcc compiler has been shown onto the next slide.
- sizeof () operator is used to check the size of a variable.



| DATA TYPE | MEMORY (BYTES) | RANGE | FORMAT SPECIFIER |
|---|---|---|---|
| short int | 2 | -32,768 to 32,767 | %hd |
| unsigned short int | 2 | 0 to 65,535 | %hu |
| unsigned int | 4 | 0 to 4,294,967,295 | %u |
| int | 4 | -2,147,483,648 to 2,147,483,647 | %d |
| long int | 4 | -2,147,483,648 to 2,147,483,647 | %ld |
| unsigned long int | 4 | 0 to 4,294,967,295 | %lu |
| long long int | 8 | -(2^63) to (2^63)-1 | %lld |
| unsigned long long int | 8 | 0 to 18,446,744,073,709,551,615 | %llu |
| signed char | 1 | -128 to 127 | %c |
| unsigned char | 1 | 0 to 255 | %c |
| Float | 4 | 1.2E-38 to 3.4E+38 | %f |
| Double | 8 | 2.3E-308 to 1.7E+308 | %lf |
| long double | 12 | 3.4E-4932 to 1.1E+4932 | %Lf |

## Algorithm Definition

- An algorithm is a step by step procedure to solve a problem.
- In normal language, the algorithm is defined as a sequence of statements which are used to perform a task.



## Algorithm Definition…

- In computer science, an algorithm can be defined as follows...

**"An algorithm is a sequence of unambiguous instructions used for solving a problem, which can be implemented (as a program) on a computer."**

## Characteristics of an Algorithm

- **Input -** Every algorithm must take zero or more number of input values from external.
- **Output -** Every algorithm must produce an output as a result.
- **Definiteness -** Every statement / instruction in an algorithm must be clear and unambiguous (only one interpretation).
- **Finiteness -** For all different cases, the algorithm must produce a result within a finite number of steps.
- **Effectiveness -** Every instruction must be basic enough to be carried out and it also must be feasible.



## Example

- Let us consider the following problem for finding the largest value in a given list of values.
- **Problem Statement:** Find the largest number in the given list of numbers?
- **Input:** A list of positive integer numbers. (List must contain at least one number).
- **Output:** The largest number in the given list of positive integer numbers.

## Algorithm

- **Step 1:** Define a variable 'max' and initialize with '0'.
- **Step 2:** Compare first number (say 'x') in the list 'L' with 'max', if 'x' is larger than 'max', set 'max' to 'x'.
- **Step 3:** Repeat step 2 for all numbers in the list 'L'.
- **Step 4:** Display the value of 'max' as a result.



## Code using C Programming

```
int findMax(L)
{
int max = 0,i;
for(i=0; i < listSize; i++)
        {
        if(L[i] > max)
        max = L[i];
          }
        return max;
}
```

## Efficiency of an Algorithm

- A measure of the average execution time necessary for an **algorithm** to complete work on a set of data.
- **Algorithm efficiency** is characterized by its order.
- Typically a bubble sort **algorithm** will have **efficiency** in sorting N items proportional to and of the order of $N^2$, usually written $O(N^2)$.



## Time and Space Complexity

- ***Time complexity*** is a function describing the amount of time an algorithm takes in terms of the amount of input to the algorithm.
- **"Time"** can mean the ***number of memory accesses performed, the number of comparisons between integers, the number of times some inner loop is executed, or some other natural unit related to the amount of real-time the algorithm will take***.

## Time and Space Complexity…

- ***Space complexity*** is a function describing the amount of memory (space) an algorithm takes in terms of the amount of input to the algorithm.
- We often speak of ***"extra" memory needed, not counting the memory needed to store the input itself.***
- Space complexity is ***sometimes ignored because the space used is minimal and/or obvious, but sometimes it becomes as important an issue as time***.

# Asymptotic Notations

- Whenever we want to perform an analysis of an algorithm, we need to calculate the complexity of that algorithm.
- But when we calculate the complexity of an algorithm it does not provide the exact amount of resource required.
- So instead of taking the exact amount of resource, we represent that complexity in a general form (Notation) which produces the basic nature of that algorithm.

**"Asymptotic notation of an algorithm is a mathematical representation of its complexity."**



# Asymptotic Notations...

- Majorly, we use three types of asymptotic notations:
  - **Big - Oh (O)**
  - **Big - Omega (Ω)**
  - **Big - Theta (Θ)**

# Big - Oh Notation (O)

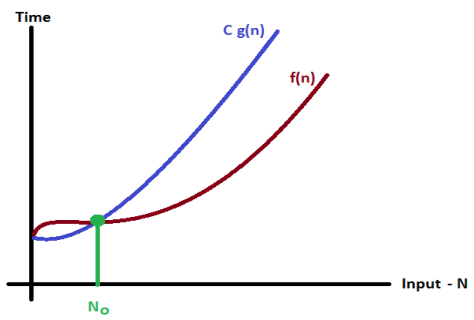
- **Big - Oh** notation is used to define the **upper bound** of an algorithm in terms of Time Complexity.
- That means **Big - Oh notation** always indicates the maximum time required by an algorithm for all input values.
- That means **Big - Oh** notation describes the **worst case** of an algorithm time complexity.
- **Big - Oh** Notation can be defined as follows:

**"Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term. If f(n) <= C g(n) for all n >= $n_0$, C > 0 and $n_0$ >= 1. Then we can represent f(n) as O(g(n))."**



# Big - Oh Notation (O)...

- Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis.
- In above graph after a particular input value $n_0$, always C g(n) is greater than f(n) which indicates the algorithm's **upper bound.**



# Example

- Consider the following f(n) and g(n).
  **f(n) = 3n + 2**
  **g(n) = n**
- If we want to represent **f(n)** as **O(g(n))** then it must satisfy **f(n) <= C g(n)** for all values of **C > 0** and **$n_0$>= 1**
  ⇒ f(n) <= C g(n)
  ⇒3n + 2 <= C n
- Above condition is always TRUE for all values of **C = 4** and **n >= 2**.
- By using **Big - Oh notation,** we can represent the time complexity as follows…
  **f(n)= 3n + 2 = O(n)**



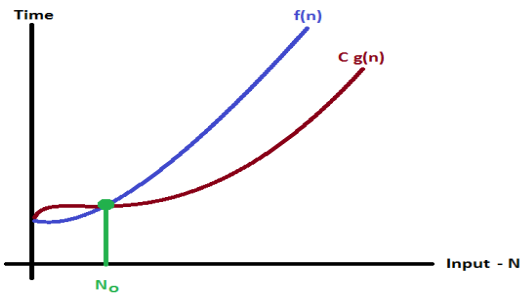
# Big - Omega Notation (Ω)

- Big - Omega notation is used to define the **lower bound** of an algorithm in terms of Time Complexity.
- That means Big-Omega notation always indicates the minimum time required by an algorithm for all input values.
- That means Big-Omega notation describes the **best case** of an algorithm time complexity.
- Big - Omega Notation can be defined as follows:

**"Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term. If f(n) >= C g(n) for all n >= n0, C > 0 and n0 >= 1. Then we can represent f(n) as Ω(g(n)). f(n) = Ω(g(n))"**

# Big - Omega Notation (Ω)...

- Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis
- In above graph after a particular input value $n_0$, always C g(n) is less than f(n) which indicates the algorithm's lower bound.






# Example

- Consider the following f(n) and g(n)…
  **f(n) = 3n + 2**
  **g(n) = n**
- If we want to represent **f(n)** as **Ω(g(n))** then it must satisfy **f(n) >= C g(n)** for all values of **C > 0** and $\mathbf{n_0 >= 1}$
  ⇒ f(n) >= C g(n)
  ⇒3n + 2 >= C n
- Above condition is always TRUE for all values of **C = 1** and **n >= 1**.
- By using Big-Omega notation we can represent the time complexity as follows…
  **f(n)= 3n + 2 = Ω(n)**

# Big - Theta Notation (Θ)

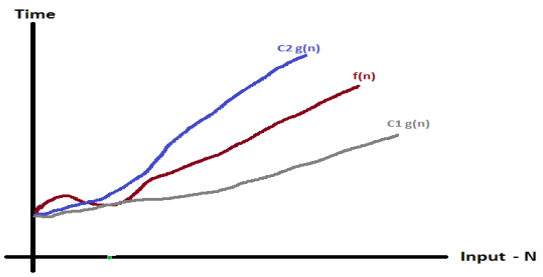
- Big - Theta notation is used to define the **average bound** of an algorithm in terms of Time Complexity.
- That means Big - Theta notation always indicates the average time required by an algorithm for all input values.
- That means Big - Theta notation describes the **average case** of an algorithm time complexity.
- Big - Theta Notation can be defined as follows:

**"Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term. If C1 g(n) <= f(n) <= C2 g(n) for all n >= n0, C1 > 0, C2 > 0 and n0 >= 1. Then we can represent f(n) as Θ(g(n)). f(n) = Θ(g(n))"**



# Big - Theta Notation (Θ)...

- Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis.
- In above graph after a particular input value $n_0$, always $C_1$ g(n) is less than f(n) and $C_2$ g(n) is greater than f(n) which indicates the algorithm's **average** bound.



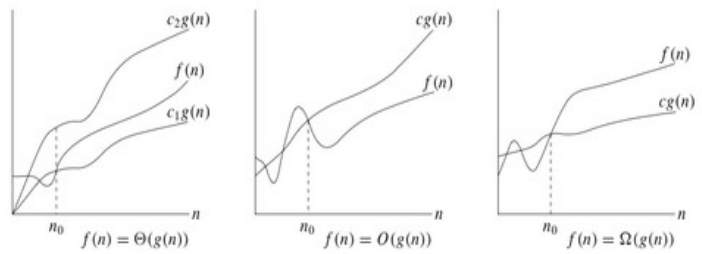
# Example

- Consider the following f(n) and g(n)…
  **f(n) = 3n + 2**
  **g(n) = n**
- If we want to represent **f(n)** as **Θ(g(n))** then it must satisfy $\mathbf{C_1\, g(n) <= f(n) <= C_2\, g(n)}$ for all values of $\mathbf{C_1 > 0, C_2 > 0}$ and $\mathbf{n_0 >= 1}$
  $\Rightarrow C_1\, g(n) <= f(n) <= C_2\, g(n)$
  $\Rightarrow C_1\, n <= 3n + 2 <= C_2\, n$
- Above condition is always TRUE for all values of $\mathbf{C_1 = 1, C_2 = 4}$ and **n >= 2**.
- By using Big - Theta notation we can represent the time complexity as follows…
  **f(n) =3n + 2 = Θ(n)**



# Intuition about the notations

| Notation | Intuition |
|---|---|
| O (Big-Oh) | $f(n) \le g(n)$ |
| Ω (Big-Omega) | $f(n) \ge g(n)$ |
| Θ (Theta) | $f(n) = g(n)$ |



# Common Asymptotic Notations

| | |
|---|---|
| **Constant** | **O(1)** |
| **Logarithmic** | O(**log n**) |
| **Linear** | O(**n**) |
| **n log n** | O(**n log n**) |
| **Quadratic** | O($n^2$) |
| **Cubic** | O($n^3$) |
| **Polynomial** | $n^{O(1)}$ |
| **Exponential** | $2^{O(n)}$ |



# Time-Space Trade-off

- The best algorithm to solve a given problem is one that requires less space in memory and takes less time to complete its execution.
- But in practice, it is not always possible to achieve both these objectives.
- As we know there may be more than one approach to solve a particular problem.
- One approach may take more space but takes less time to complete its execution while the other approach may take less space but takes more time to complete its execution.
- We may have to sacrifice one at the cost of the other.

# Time-Space Trade-off...

- If space is our constraint, then we have to choose a program that requires less space at the cost of more execution time.
- On the other hand, if time is our constraint then we have to choose a program that takes less time to complete its execution at the cost of more space.
- That is what we can say that there exists a time-space trade-off among algorithms.

# Abstract Data Types (ADT)

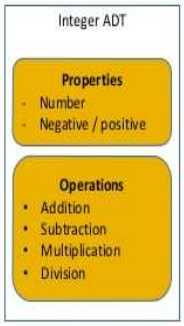
- **Model** of a data type
  - Properties of the data
  - Operations that can be performed on that data
- ***Definition:*** *Abstract data type* (ADT) is a **mathematical model** with a collection of operations defined on that model.




# Abstract Data Types (ADT)...

## Example of Abstract Data Type (ADT)

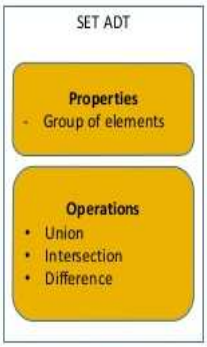
- Integer
  - ...., -4, -3, -2, -1, 0, 1, 2, 3, 4 ...



# Abstract Data Types (ADT)...

## One more Example of ADT

- SET
  - {2,4,6,8,10}
- Take two SET as input and return union
  - SET union(SET a, SET b)
- Take two SET as input and return intersection
  - SET intersection(SET a, SET b)
- Take two SET as input and return difference
  - SET difference(SET a, SET b)






# Abstract Data Types (ADT)...

- Some more examples of ADT are Stack, Queue, List etc.
- Let us see some operations of those mentioned ADT –
- **Stack –**
  - **isFull( )**, This is used to check whether stack is full or not
  - **isEmpty( )**, This is used to check whether stack is empty or not
  - **push(x)**, This is used to push x into the stack
  - **pop( )**, This is used to delete one element from top of the stack
  - **peek( )**, This is used to get the top most element of the stack
  - **size( )**, this function is used to get number of elements present into the stack

## Abstract Data Types (ADT)...

- **Queue –**
  - **isFull( )**, This is used to check whether queue is full or not
  - **isEmpty( )**, This is used to check whether queue is empty or not
  - **insert(x)**, This is used to add x into the queue at the rear end
  - **delete( )**, This is used to delete one element from the front end of the queue
  - **size( )**, this function is used to get number of elements present into the queue



## Abstract Data Types (ADT)...

- **List –**
  - **size( )**, this function is used to get number of elements present into the list
  - **insert(x)**, this function is used to insert one element into the list
  - **remove(x)**, this function is used to remove given element from the list
  - **get(i)**, this function is used to get element at position i
  - **replace(x, y)**, this function is used to replace x with y value

## Implementation of ADT

- Implementation of an ADT means writing a program in a programming language.
- Data structure deals with the implementation of various ADTs.



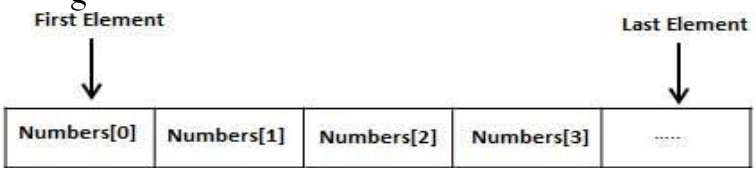
Dr. Sunil Kumar, CSE Dept., MIET Meerut

# UNIT 1 - Part II: Arrays

## Table of Contents

# Arrays

- **Definition:**
  - An array is a sequential collection of elements of same data type and stores data elements in a continuous memory location.
  - The elements of an array are accessed by using an index.
  - The index of an array of size N can range from 0 to N−1.
  - The lowest address corresponds to the first element and the highest address to the last element.






# Continue

- If array size is 5, then your index will range from 0 to 4 (5-1).
- Each element of an array can be accessed by using arr [index].
- Consider following array:
  - The size of this array is 5.
  - If you want to access 34, then you can access it by using arr[ 2 ] i.e. 34.

| arr | 5 | 23 | 34 | 9 | 28 |
|---|---|---|---|---|---|
| index | 0 | 1 | 2 | 3 | 4 |

# Array Declaration in C/C++

Declaring an array you must specify the following:

- **Size of the array:** This defines the number of elements that can be stored in the array.
- **Type of array:** This defines the type of each element i.e. number, character, or any other data type.

**data type** **variable** **name size**

**Example:** int arr[5];



# Array Initialization

- Array is initialized at the time of declaration.

  **The sample format if an array is initialized at the time of declaration is:**

  **Syntax:** type arr[size]={elements}

  **Example:** int arr[5]={4,12,7,15,9};

# Single Dimensional Array

- A one-dimensional array is also called a single dimensional array where the elements will be accessed in sequential order.
- This type of array will be accessed by the subscript of either a column or row index.

Syntax: ***data-type Array-name[size]***

Example: ***int x[10];***

***char name[20];***

***float f[5];***



# Address Calculation in 1D Array

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

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

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

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

## Continue

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

  **Address of A [ I ] = BA + W * ( I – LB )**

  Where,

  **BA** = Base address

  **W** = Storage Size of one element stored in the array (in byte)

  **I** = Subscript of element whose address is to be found

  **LB** = Lower limit / Lower Bound of subscript, if not specified assume 0 (zero)



## Example

- Given the base address of an array **B[1300…..1900]** as 1020 and size of each element is 2 bytes in the memory. Find the address of **B[1700]**.

  **Solution:**

  The given values are: B = 1020, LB = 1300, W = 2, I = 1700

  **Address of A [ I ] = B + W * ( I – LB )**

  = 1020 + 2 * (1700 – 1300)

  = 1020 + 2 * 400

  = 1020 + 800

  = 1820

# Multidimensional Arrays

- When the number of dimensions specified is more than one then it is called as a multi-dimensional array.

**Two-Dimensional Arrays:**

**Syntax:** *data-type Array-name[row size][column size]*

- A two-dimensional array can be thought of a matrix with row and columns. For example if we declare a 2-D array as: int arr[3][4]; then this declaration means a 2-D arr of 3 rows and 4 column which can be represented as follows:

| | Col 1 | Col 2 | Col 3 | Col 4 |
|---|---|---|---|---|
| Row1 | Arr[0][0] | Arr[0][1] | Arr[0][2] | Arr[0][3] |
| Row2 | Arr[1][0] | Arr[1][1] | Arr[1][2] | Arr[1][3] |
| Row3 | Arr[2][0] | Arr[2][1] | Arr[2][2] | Arr[2][3] |



# Row Major Order

- Row Major Order is a method of representing multi-dimension array in sequential memory.
- In this method, elements of an array are **arranged sequentially row by row.**
- Thus elements of first row occupies first set of memory locations reserved for the array, elements of second row occupies the next set of memory and so on.
- Consider a Two Dimensional Array consist of N rows and M columns. It can be stored sequentially in memory row by row as shown below:

| | | | | |
|---|---|---|---|---|
| Row 0 | A[0,0] | A[0,1] | ..... | A[0,M-1] |
| Row 1 | A[1,0] | A[1,1] | ..... | A[1,M-1] |
| | ..... | | | |
| Row N-1 | A[N-1,0] | A[N-1,1] | ..... | A[N-1,M-1] |

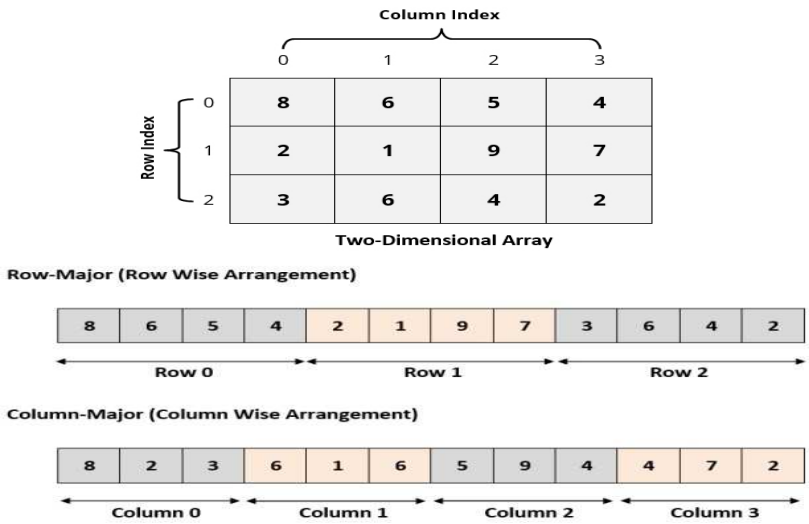
# Column Major Order

- Column Major Order is a method of representing multi dimension array in sequential memory.
- In this method elements of an array are arranged sequentially column by column.
- Thus elements of first column occupies first set of memory locations reserved for the array, elements of second column occupies the next set of memory and so on.
- Consider a Two Dimensional Array consist of N rows and M columns. It can be stored sequentially in memory column by column as shown below:

| | | | | |
|---|---|---|---|---|
| Column 0 | A[0,0] | A[1,0] | .. | A[N-1,0] |
| Column 1 | A[0,1] | A[1,1] | .. | A[N-1,1] |
| | ... | | | |
| Column N-1 | A[0,M-1] | A[1,M-1] | ... | A[N-1,M-1] |



# Example



## Row Major Order

- The Location of element A[i, j] can be obtained by evaluating expression:
  **LOC (A [i, j]) = Base Address (BA) + W [N (i-1) + (j-1)]**
  **Here,**
  **Base Address** is the address of first element in the array.
  **W** is the word size. It means number of bytes occupied by each element.
  **M** is number of rows in array.
  **N** is number of columns in array.



## Column Major Order

- The Location of element A[i, j] can be obtained by evaluating expression:
  **LOC (A [i, j]) = Base Address (BA) + W [M (j-1) + (i-1)]**
  **Here,**
  **Base Address** is the address of first element in the array.
  **W** is the word size. It means number of bytes occupied by each element.
  **M** is number of rows in array.
  **N** is number of columns in array.

## Example

- Given an array [1…5,1…7] of integers. Calculate address of element T[4,6] by using Row Major Order, where BA=900, W=1.

**Solution:**

Location ( A[ i, j ] ) =BA+ W*[ N x ( i - 1 ) + ( j - 1 )]

i = 4 , j = 6, M= 5 , N= 7, W=1

Location (T [4,6]) = BA + [(7 x (4-1) + (6-1)]

= 900+ [(7 x 3) +5]

= 900+ 21 + 5

= 926



## Continue

- Given an array [1…6,1…8] of integers. Calculate address element T[5,7] by using Column Major Order, where BA=300 and W=1.

**Solution:**

Location ( A[ i, j ] ) =Base Address + W*[ M x ( j - 1 ) ) + ( i - 1 )]

i = 5 , j = 7, M= 6 , N= 8, W=1

Location (T [5,7]) = BA + [6 x (7-1)) + (5-1)]

= 300+ [(6 x 6) +4]

= 300+ 36+4

= 340

## Continue

- **Important :** Usually number of rows and columns of a matrix are given ( like A[20][30] or A[40][60] ) but if it is given as **A[Lr- – – – – Ur, Lc- – – – – Uc]**
  {**Example: X [-15.....10, 15.....40]**} .

- In this case number of rows and columns are calculated by using the following methods:
  - Number of rows (**M**) will be calculated as = **(Ur – Lr) + 1**
  - Number of columns (**N**) will be calculated as = **(Uc – Lc) + 1**
  - And rest of the process will remain same as per requirement (Row Major Wise or Column Major Wise).

**Ur** = Upper limit of row/start row index of matrix, if not given assume 0 (zero)
**Lr** = Lower limit of row/start row index of matrix, if not given assume 0 (zero)
**Uc** = Upper limit of column/start column index of matrix, if not given assume 0 (zero)
**Lc** = Lower limit of column/start column index of matrix, if not given assume 0 (zero)



## Continue

**Row Major System:**

The address of a location in **Row Major System** is calculated by using the following formula:

**Address of A [ I ][ J ] = B + W * [ N * ( I – Lr ) + ( J – Lc ) ]**

**Column Major System:**

The address of a location in **Column Major System** is calculated by using the following formula:

**Address of A [ I ][ J ] = B + W * [( I – Lr ) + M * ( J – Lc )]**

Where,
**B** = Base address
**I** = Row subscript of element whose address is to be found
**J** = Column subscript of element whose address is to be found
**W** = Storage Size of one element stored in the array (in byte)
**Lr** = Lower limit of row/start row index of matrix, if not given assume 0 (zero)
**Lc** = Lower limit of column/start column index of matrix, if not given assume 0 (zero)
**M** = Number of row of the given matrix
**N** = Number of column of the given matrix

# Example

An array X [-15.....10, 15.....40] requires one byte of storage. If beginning location is 1500 determine the location of X [15][20].

**Solution:**

As you see here the number of rows and columns are not given in the question. So they are calculated as:

Number or rows say **M = (Ur – Lr) + 1** = [10 – (- 15)] +1 = 26
Number or columns say **N = (Uc – Lc) + 1** = [40 – 15)] +1 = 26

**(i) Column Major Order**

The given values are: B = 1500, W = 1 byte, I = 15, J = 20, Lr = -15, Lc = 15, M = 26

**Address of A[ I ][ J ] = B + W * [ ( I – Lr ) + M * ( J – Lc ) ]**

= 1500 + 1 * [(15 – (-15)) + 26 * (20 – 15)]

= 1500 + 1 * [30 + 26 * 5]

= 1500 + 1 *[160]

= 1660 **[Ans]**



# Continue

**(ii) Row Major Order**

The given values are: B = 1500, W = 1 byte, I = 15, J = 20, Lr = -15, Lc = 15, N = 26

**Address of A[ I ][ J ] = B + W * [ N * ( I – Lr ) + ( J – Lc ) ]**

= 1500 + 1* [26 * (15 – (-15))) + (20 – 15)]

= 1500 + 1 * [26 * 30 + 5]

= 1500 + 1 * [780 + 5]

= 1500 + 785

= 2285 **[Ans]**

## Example

- **Example:** Given an array, **arr[1.....10][1.....15]** with base value **100** and the size of each element is **1 Byte** in memory. Find the address of **arr[8][6]** with the help of row-major order?

- **Solution:**

**Address of A[ I ][ J ] = B + W * [ N * ( I – Lr ) + ( J – Lc ) ]**

Address of A[8][6] = 100 + 1 * ((8 – 1) * 15 + (6 – 1))
= 100 + 1 * ((7) * 15 + (5))
= 100 + 1 * (110)

Address of A[I][J] = 210



## Example

- **Example:** Given an array **arr[1.....10][1.....15]** with base value **100** and the size of each element is **1 Byte** in memory find the address of arr[8][6] with the help of column-major order.

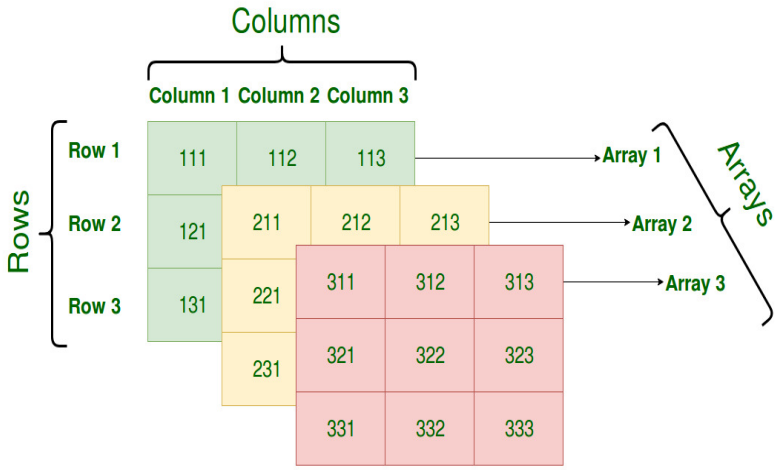
- **Solution:**

**Address of A[ I ][ J ] = B + W * [ ( I – Lr ) + M * ( J – Lc ) ]**

Address of A[8][6] = 100 + 1 * ((6 – 1) * 10 + (8 – 1))
= 100 + 1 * ((5) * 10 + (7))
= 100 + 1 * (57)

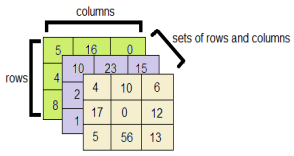
Address of A[I][J] = 157

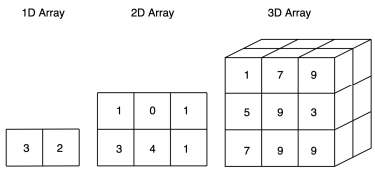
# Three Dimensional Array






# Three Dimensional Array




0th or the front set of numbers

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 4 | 10 | 6 | 17 | 0 | 12 | 5 | 56 | 13 |
| 1050 | 1052 | 1054 | 1056 | 1058 | 1060 | 1062 | 1064 | 1066 |

1st or middle set of numbers

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 10 | 23 | 15 | 2 | 5 | 9 | 1 | 16 | 20 |
| 1068 | 1070 | 1072 | 1074 | 1076 | 1078 | 1080 | 1082 | 1084 |

2nd or last set of numbers

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 5 | 16 | 0 | 4 | 35 | 19 | 8 | 13 | 2 |
| 1086 | 1088 | 1090 | 1092 | 1094 | 1096 | 1098 | 1100 | 1102 |

the memory addresses where data is stored

## Continue

- In three - dimensional array also address is calculated through two methods i.e; row-major order and column-major method.
- **To calculate address of element X[ i, j, k] using row-major order :**

  Location ( X[i, j, k] )=BA + W*[MN (i-1) + N (j-1) + (k-1)]
- **To calculate address of element X[ i, j, k] using column-major order:**

  Location ( X[i, j, k] )=BA + W*[MN (i-1) + (j-1) + M(k-1)]



## Example

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

**Solution:-** The dimensions of A are :

M=8 , N=5, R=7, i=5, j=3, k=6, W=1

**Rows - wise :**

Location (A[i, j, k]) = BA + W*[MN(i-1) + N(j-1) + (k-1)]
Location(A[5,3,6])= 900 + 1*[8x5(5-1) + 5(3-1) + (6-1)
= 900 + 40 x 4 +5 x 2 + 5
= 900 + 160 +10 +5
**= 1075**

**Columns - wise :**

Location (A[i, j, k]) = BA + W*[MN(i-1) + (j-1) + M(k-1)]
Location (A[5,3,6]) = 900 + 1*[8x5(5-1) + (3-1) + 8(6-1)]
= 900 + 40 x 4 +2 + 40
= 900 + 160 +2 +40
**= 1102**

# Applications of Arrays

- Arrays are used to Store List of values
- Arrays are used to Perform Matrix Operations
- Arrays are used to implement Search Algorithms
- Arrays are used to implement Sorting Algorithms
- Arrays are used to implement Data structures
- Arrays are also used to implement CPU Scheduling Algorithms

## Sparse Matrices

- In computer programming, a matrix can be defined with a 2-dimensional array.
- Any array with 'm' columns and 'n' rows represent a m X n matrix. There may be a situation in which a matrix contains more number of ZERO values than NON-ZERO values.
- Such matrix is known as sparse matrix.
- Sparse matrix is a matrix which contains very few non-zero elements.



## Sparse Matrices Representations

- A sparse matrix can be represented by using TWO representations, those are as follows…
  1. Triplet Representation (Array Representation)
  2. Linked Representation

## Triplet Representation

- In this representation, we consider only non-zero values along with their row and column index values.
- In this representation, the 0th row stores the total number of rows, total number of columns and the total number of non-zero values in the sparse matrix.



## Continue

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 9 & 0 \\ 0 & 8 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 2 & 0 & 0 & 0 \end{bmatrix}$$

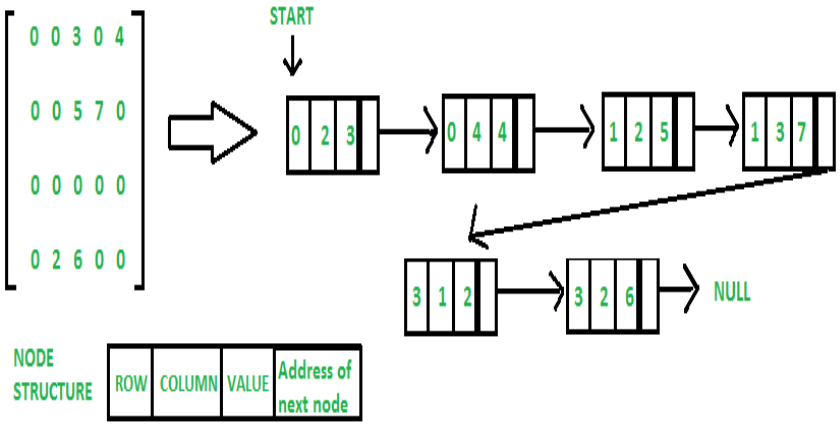
| Rows | Columns | Values |
|---|---|---|
| 5 | 6 | 6 |
| 0 | 4 | 9 |
| 1 | 1 | 8 |
| 2 | 0 | 4 |
| 2 | 3 | 2 |
| 3 | 5 | 5 |
| 4 | 2 | 2 |

# Linked List Representation

- In linked list representation, each node has four fields.
- These four fields are defined as:
  - **Row:** Index of row, where non-zero element is located
  - **Column:** Index of column, where non-zero element is located
  - **Value:** Value of the non zero element located at index – (row, column)
  - **Next node:** Address of the next node



# Continue



## Operations on Arrays

- Traversing
- Inserting
- Deleting
- Sorting
- Searching
- Merging



## Continue

- **Traversing** – Traversing refers to accessing each record exactly once so that certain items in the record may be processed (this accessing & processing is called visiting a record).
- **Inserting –** It refers to addition of new record in the given list of items.
- **Deleting –** It refers to removal of a record from the given list of items.
- **Sorting** – Arranging the records in some logical order (Ascending/Descending etc.) is called Sorting.
- **Searching –** Finding the location of record with a given key value or finding the locations of all records which satisfy one or more conditions.
- **Merging** – It refers to combining two different sorted files into a single sorted file.

## Traversing Algorithm

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

**Step 1:** [Initialize counter] Set K: = LB.

**Step 2:** Repeat Steps 3 and 4 while K<= UB

**Step 3:** [Visit element] Apply PROCESS to LA[K]

**Step 4:** [Increase counter] Set K:= K + 1

**Step 5:** Exit.



## Continue

- This algorithm traverses a Linear Array LA with lower bound LB and upper bound UB.

**Step 1:** Repeat for K= LB to UB:

Apply PROCESS to LA[K]

**Step 2:** Exit.

# Inserting Algorithm

- **INSERT (LA, N, K, ITEM)**
- Here LA is a linear array with N elements and K is a positive integer such as K<=N. This algorithm inserts an element ITEM into $K^{th}$ position in LA.

**Step 1:** [Initialize counter] Set J: = N.
**Step 2:** Repeat Steps 3 and 4 while J>=K.
**Step 3:** Set LA[J+1]= LA [J]
**Step 4:** Set J=J-1
**Step 5:** Set LA[K]=ITEM
**Step 6:** Set N= N + 1
**Step 7:** Exit.



# Deleting Algorithm

- **DELETE(LA, N, K, ITEM)**
- Here LA is a linear array with N elements and K is a positive integer such that K<=N. This algorithm deletes the Kth element from LA.

**Step 1:** Set ITEM= LA[K]
**Step 2:** Apply PROCESS to LA[K]
**Step 3:** Repeat for J=K to N-1:
Set LA[J] = LA[J+1]
**Step 4:** Set N= N - 1
**Step 5:** Exit.

# Derivation of Index Formulae

**1D**

**Address calculation of 1D:**

**A[1]** **a[2]** **a[3 ].....** **A[l1]**

| | | | | | |
|---|---|---|---|---|---|
| | | | | | |

**Suppose address of a[1] in BA and word size W is 1.then**

Address of a[1]=BA

Address of a[2]=BA+1

Address of a[3]=BA+2

So Address of a[i]=BA+(i-1)

Address of a[i]=BA+W( i-1 )

**Address of a[i]=BA+W*$E_I$**

**Where $E_I$** is effective index of i.( **$E_I$=i - lower bound of array)**

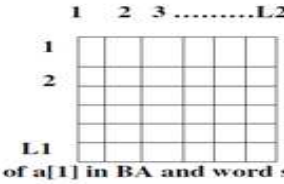


Dr. Sunil Kumar, CSE Dept., MIET Meerut

# Continue

**2D**

**Address calculation of 2 D (Row major order):**




**Suppose address of a[1] in BA and word size W is 1.then**

Address of a[1][1]=BA

Address of a[1][2]=BA+1

Address of a[1][3]=BA+2

So Address of a[1][L2]=BA+L2-1

Address of a[2][1]= BA+L2-1+1=BA+L2

Address of a[2][2]=BA+L2+1

Address of a[2][3]=BA+L2+2



# Continue

So Address of a[2][L2]=BA+L2+L2-1

So Address of a[3][1]=BA+L2+L2-1+1=BA+2*L2

So Address of a[i][1]=BA+( i-1)*L2

Address of a[i][2]=BA+( i-1)*L2+1

Address of a[i][3]=BA+( i-1)*L2+2

Address of a[i][j]=BA+( i-1)*L2+(j-1)

Address of a[i][j]=BA+W($E_1$*L2+$E_2$)

- Where $E_1$ and $E_2$ are effective index of i and j and l1 and l2 are length of row and column dimensions.($E_i$=i-lower bound)

# UNIT 1 - Part III: Linked List

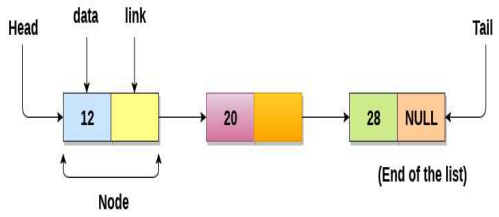
## Table of Contents

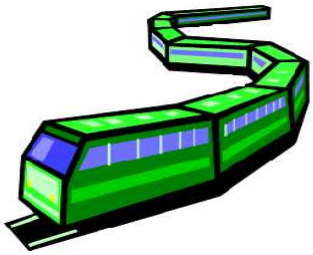


3

# Link List

- Linked List can be defined as ***collection of objects*** called **nodes** that are randomly stored in the memory.
- A node contains **two fields:**
  - Data stored at that particular address
  - Pointer which contains the address of the next node.



4

# Need of Linked List

- **What are the problems with Arrays?**
  - Size is fixed
  - Array items are stored contiguously
  - Insertion and deletion at particular position is complex
- **Why Linked List?**
  - Size is not fixed
  - Data can be stored at any place
  - Insertion and deletion at particular position is simple and faster

5

# Types of Linked List

- Singly Linked List


- Doubly Linked List


- Circularly Linked List


6

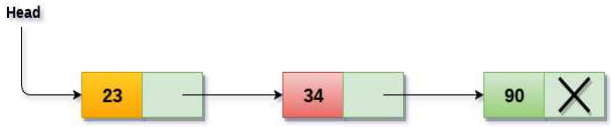
# Dynamic Memory Allocation

- Dynamic memory allocation
  - Obtain and release memory during execution
- **malloc**
  - Takes number of bytes to allocate
    - Use **sizeof** to determine the size of an object
  - Returns pointer of type **void***
    - A **void*** pointer may be assigned to any pointer
    - If no memory available, returns **NULL**
  - **new = malloc(sizeof(struct node));**
- **free**
  - Deallocates memory allocated by **malloc**
  - Takes a pointer as an argument
  - **free (new);**

7

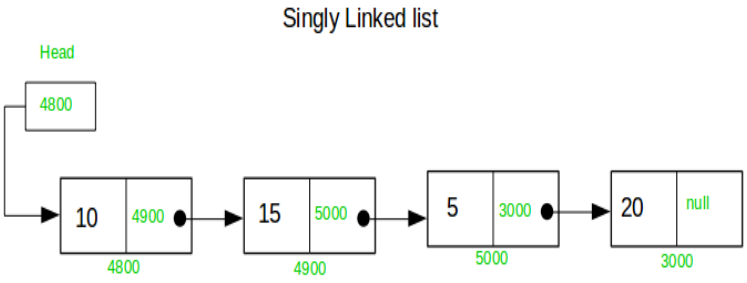
# Singly Linked List

- A node in the singly linked list consist of two parts: data part and link part.
- Data part of the node **stores actual information that is to be represented by the node** while the link part of the node stores the **address of its immediate successor**.
- Singly linked list can be traversed only in one direction.




8

# Continue




9

## Singly linked list operations

**Insertion:**

- **Insertion of a node at the beginning**
- **Insertion of a node at the ending**
- **Insertion of a node at any position in the list**

**Deletion:**

- **Deletion at beginning**
- **Deletion at the ending**
- **Deletion at any position**

**Display:**

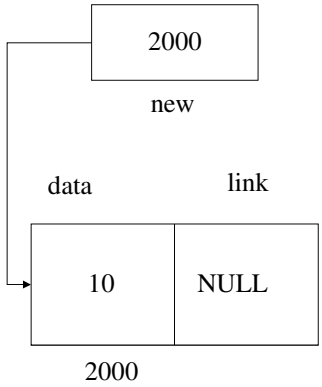
- **Displaying/Traversing the elements of a list**

10

# Singly Linked Lists

## Node Structure

```
struct node
{
    int data;
    struct node *link;
}*new, *ptr, *header;
```



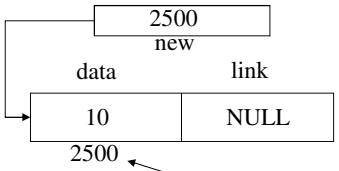

## Creating a node

```
new = malloc (sizeof(struct node));
new -> data = 10;
new -> link = NULL;
```

11

# Inserting a node at the beginning

## Create a node that is to be inserted

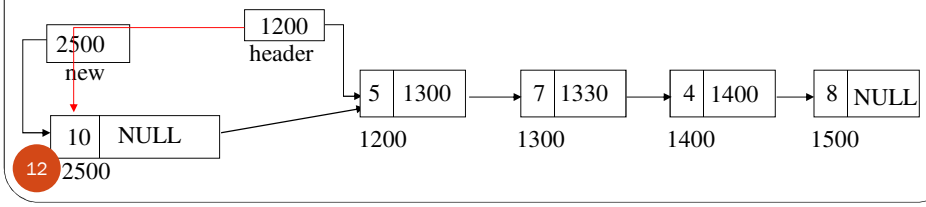



**Algorithm:**

1.Create a new node.
2.if (header = = NULL)
3. header = new;
4.else
5.{
6. new -> link = header;
7. header = new;
8.}

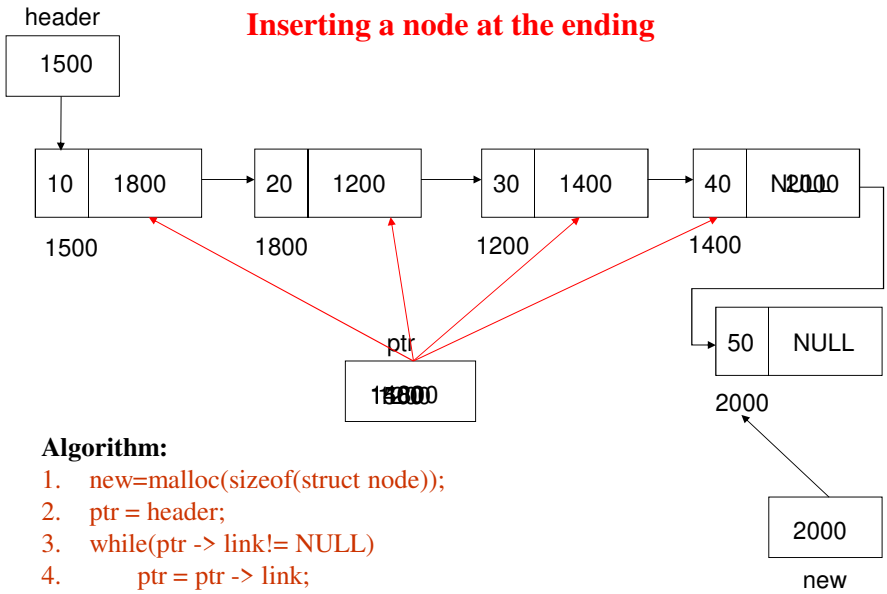
## If the list is empty


## If the list is not empty




12

## Inserting a node at the ending

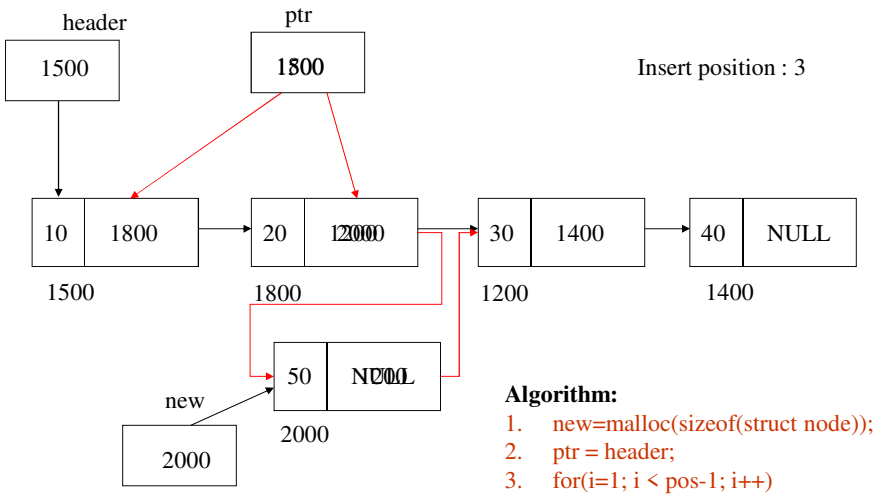



**Algorithm:**

```
new=malloc(sizeof(struct node));
ptr = header;
while(ptr -> link!= NULL)
        ptr = ptr -> link;
ptr -> link = new;
```

13

## Inserting a node at the given position

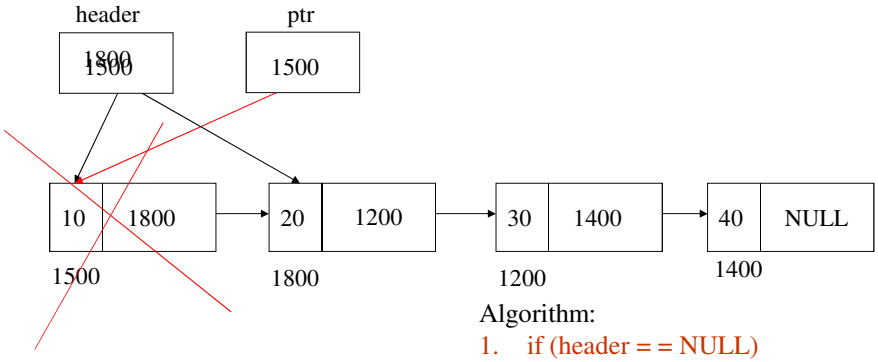



**Algorithm:**

```
new=malloc(sizeof(struct node));
ptr = header;
for(i=1; i < pos-1; i++)
        ptr = ptr -> link;
new -> link = ptr -> link;
ptr -> link = new;
```

14

## Deleting a node at the beginning

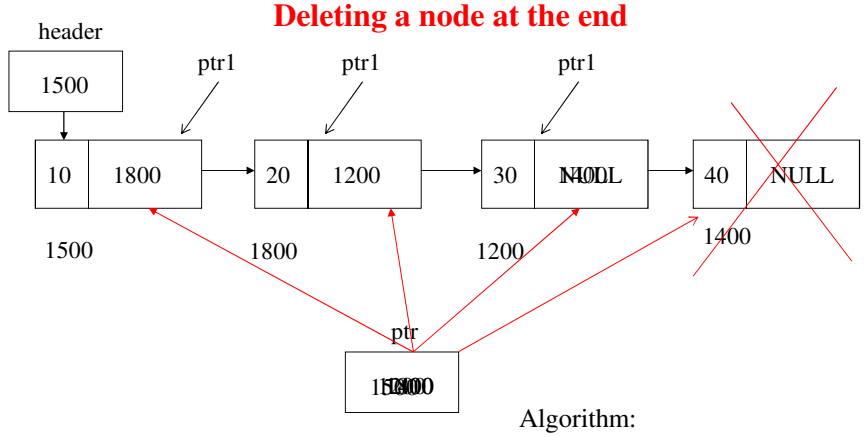



Algorithm:

```
if (header = = NULL)
        print  "List is Empty";
else
{
        ptr = header;
        header = header -> link;
        free(ptr);
}
```

15

## Deleting a node at the end

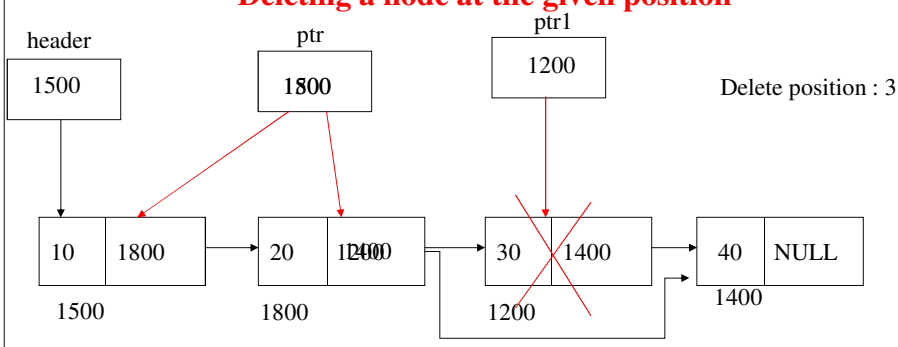



Algorithm:

1. ptr = header;
2. while(ptr -> link != NULL)
3. ptr1=ptr;
4. ptr = ptr -> link;
5. ptr1 -> link = NULL;
6. free(ptr);

16

## Deleting a node at the given position

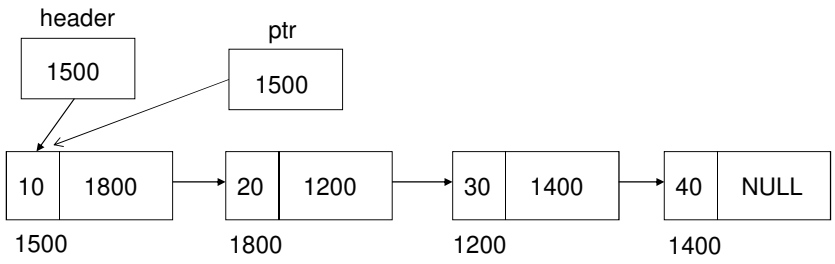



**Algorithm:**

1. ptr = header ;
2. for(i=1;i<pos-1;i++)
3. ptr = ptr -> link;
4. ptr1 = ptr -> link;
5. ptr -> link = ptr1-> link;
6. free(ptr1);

17

## Traversing the elements of a list




**Algorithm:**

```
1.  if(header = = NULL)
2.          print  "List is empty";
3.  else
4.          for (ptr = header ; ptr->link != NULL ; ptr = ptr -> link)
5.                  print "ptr->data";
```

18

```
/*Program to implement singly linked list*/
#include<stdio.h>          #include<malloc.h>          #include<conio.h>     #include<stdlib.h>
void traverse( );          void deletion( );          void insertion( );
int choice, i, pos, item;
struct node  {
        int data;
        struct node *link;
}*header,*ptr,*ptr1,*new;
void main( )   {
        header=NULL;
        ptr=header;
        printf("****Menu****\n");
        printf("\n 1.Insertion\n 2.Deletion\n 3.Traverse\n 4.Exit\n");
        while(1)    {
                printf("\nEnter your choice");
                scanf("%d",&choice);
                switch(choice)   {
                        case 1:     insertion( );              break;
                        case 2:     deletion( );        break;
                        case 3:     traverse( );        break;
                        case 4:     exit( );
                        default:    printf("\nWrong choice\n");
                }/*end of switch*/
        }/*end of while*/
}/*end of main*/
```

19

```
void insertion( )
    {
    new=malloc(sizeof(struct node));
    printf("\n Enter the item to be inserted\n");
    scanf("%d",&item);
    new->data=item;
    if(header = = NULL)
    {
        new->link=NULL;
        header=new;
    }/*end of if*/
    else
    {
        printf("\nEnter the place to insert the item\n");
        printf("1.Start\n 2.Middle\n 3.End\n");
        scanf("%d",&choice);
        if(choice = = 1)
        {
            new->link=header;
            header=new;
        }
```

20

```
if(choice = = 2)
{
    ptr=header;
    printf("Enter the position to place an item: ");
    scanf("%d",&pos);
    for(i=1;i<pos-1;i++)
        ptr=ptr->link;
    new->link=ptr->link;
    ptr->link=new;
 }
if(choice = = 3)
{
        ptr=header;
        while(ptr->link!=NULL)
            ptr=ptr->link;
        new->link=NULL;
        ptr->link=new;
}
}/*end of else*/
}/*end of insertion*/
```

21

```
void deletion( )
{
    ptr=header;
    if(header = = NULL)
    {
        printf("\nThe list is empty");
    }
    else
    {
        printf("\n1.Start \n2.Middle \n3.End");
        printf("\nEnter the place to delete the element from list");
        scanf("%d",&choice);
        if(choice = = 1)
        {
            printf("\nThe deleted item from the list is -> %d", ptr->data);
            header=header->link;
        }
```

22

```
        if(choice = = 2)
        {
            printf("\nEnter the position to delete the element from the list");
            scanf("%d",&pos);
            for(i=0;i<pos-1;i++)
            {
                ptr1=ptr;
                ptr=ptr->link;
            }
            printf("\nThe deleted element is ->%d", ptr->data);
            ptr1->link=ptr->link;
        }
        if(choice = = 3)
        {
            while(ptr->link!=NULL){
                ptr1=ptr;
                ptr=ptr->link;
            }//while
            printf("\nThe deleted element from the list is ->%d", ptr->data);
            ptr1->link=NULL;
        }
    }/*end of else*/
}/*end of deletion*/
```

23

```
void traverse( )
{
    if(header = = NULL)
        printf("List is empty\n");
    else
    {
        printf("\nThe elements in the list are");
        for(ptr=header;ptr->link!=NULL;ptr=ptr->link)
            printf("\n\tNode at %d is %d",++i,ptr->data);
    }
}/*end of traverse*/
```

24

## Doubly linked list

- Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence.
- Therefore, in a doubly linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer), pointer to the previous node (previous pointer).

25

# Continue

- In a singly linked list one can move from the header node to any node in one direction only (left-right).
- A doubly linked list is a two-way list because one can move in either direction. That is, either from left to right or from right to left.
- It maintains two links or pointer. Hence it is called as doubly linked list.

| PREV | DATA | NEXT |
|---|---|---|

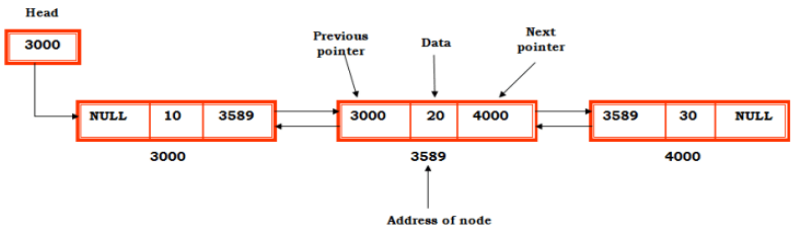
**Structure of the node**

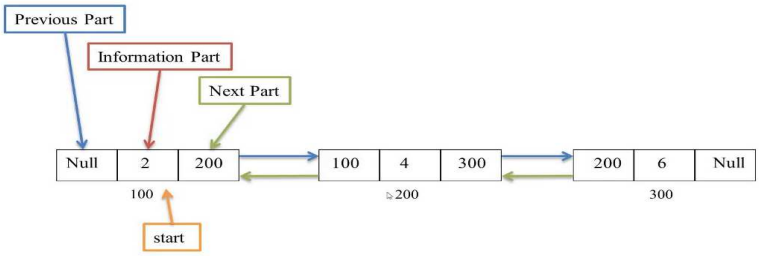
- Where, DATA field - stores the element or data, PREV- contains the address of its previous node, NEXT- contains the address of its next node.

**An example of a doubly linked list**


26

# Continue




27

## Doubly linked list operations

**Insertion:**

- **Insertion of a node at the beginning**
- **Insertion of a node at the ending**
- **Insertion of a node at any position in the list**

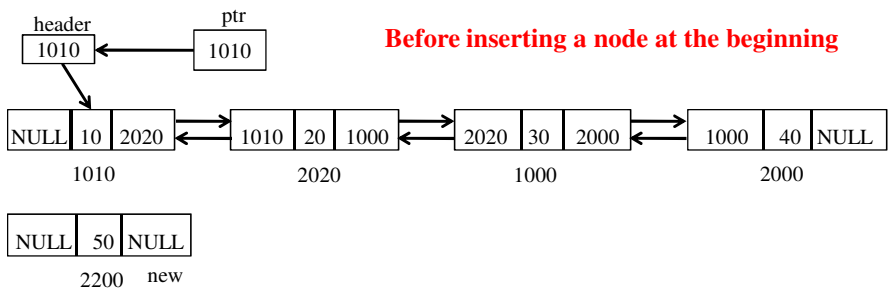
**Deletion:**

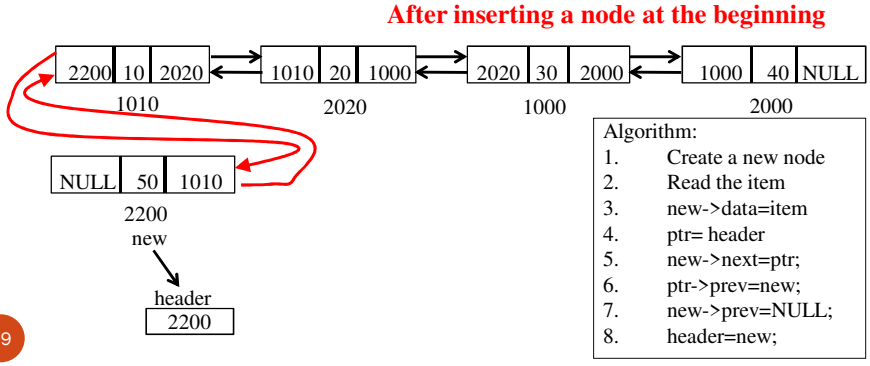
- **Deletion at beginning**
- **Deletion at the ending**
- **Deletion at any position**

**Display:**

- **Displaying/Traversing the elements of a list**

28

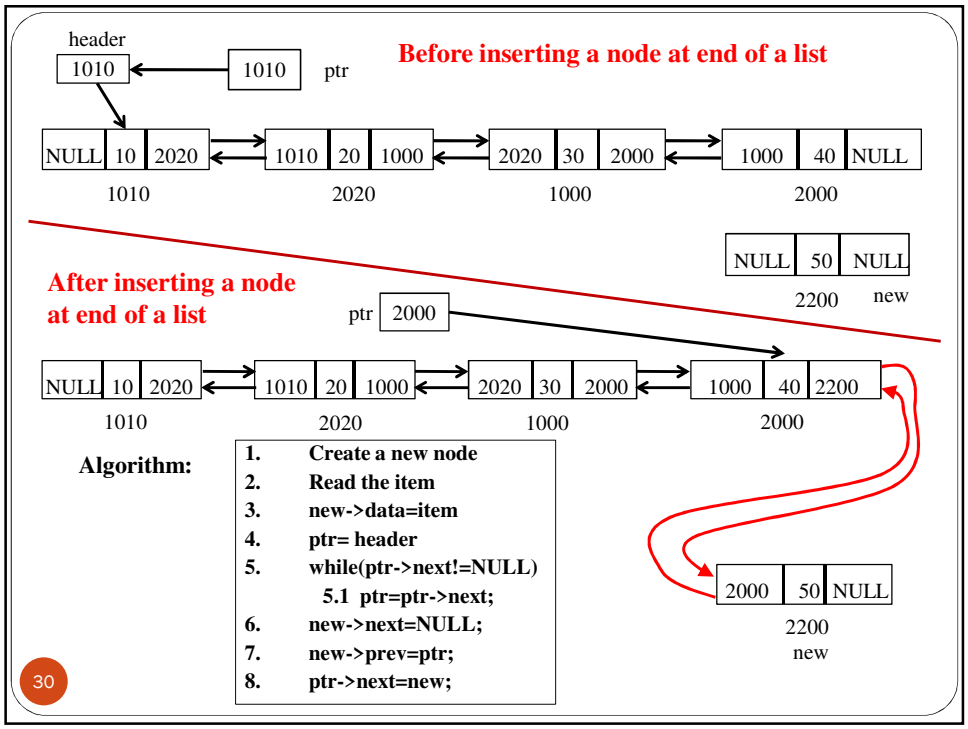


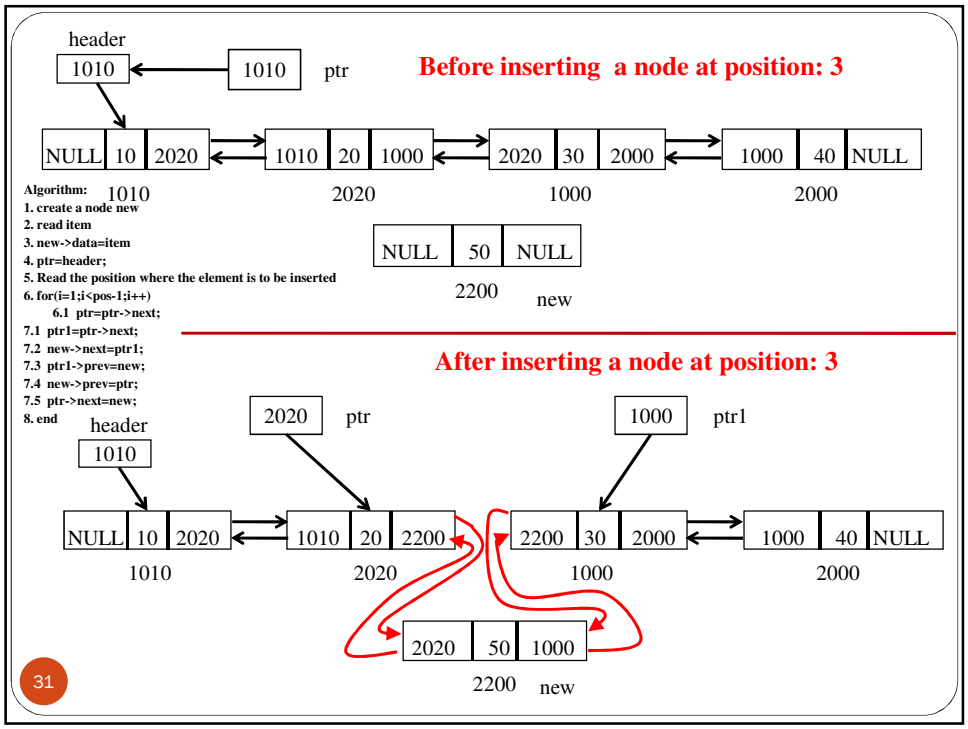



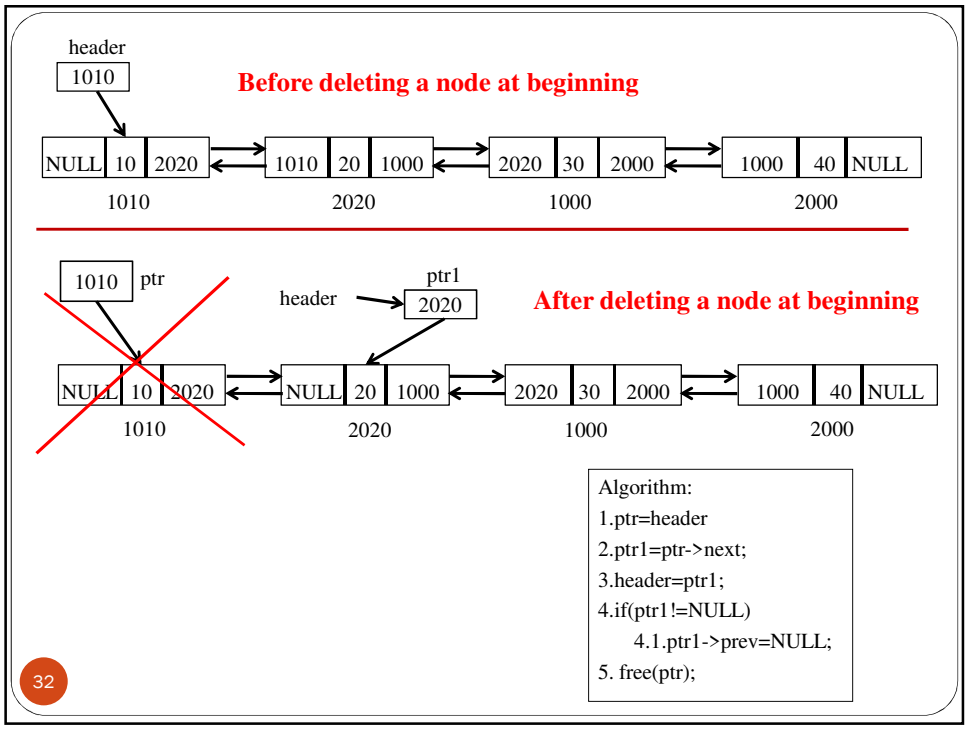

Algorithm:

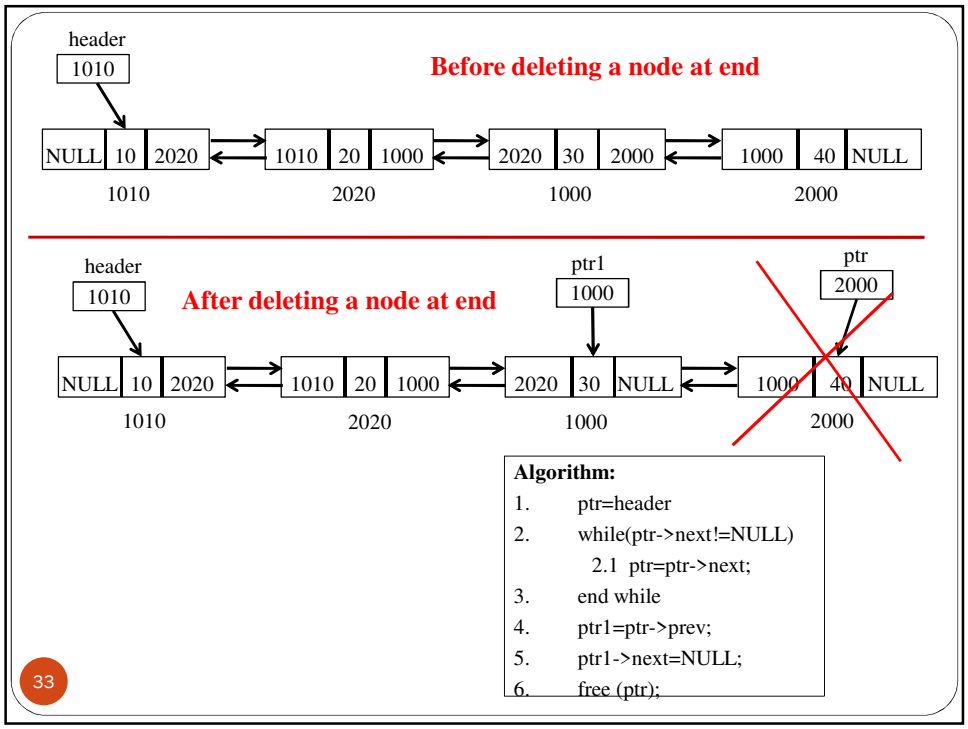
1. Create a new node
2. Read the item
3. new->data=item
4. ptr= header
5. new->next=ptr;
6. ptr->prev=new;
7. new->prev=NULL;
8. header=new;

29


header
1010
1010
ptr
Before inserting a node at end of a list
NULL 10 2020
1010 20 1000
2020 30 2000
1000 40 NULL
1010
2020
1000
2000
NULL 50 NULL
2200
new
After inserting a node at end of a list
ptr 2000
NULL 10 2020
1010 20 1000
2020 30 2000
1000 40 2200
1010
2020
1000
2000
2000 50 NULL
2200
new
Algorithm:
1. Create a new node
2. Read the item
3. new->data=item
4. ptr= header
5. while(ptr->next!=NULL)
5.1 ptr=ptr->next;
6. new->next=NULL;
7. new->prev=ptr;
8. ptr->next=new;
30


header
1010
1010
ptr
Before inserting a node at position: 3
NULL 10 2020
1010 20 1000
2020 30 2000
1000 40 NULL
1010
2020
1000
2000
Algorithm:
1. create a node new
2. read item
3. new->data=item
4. ptr=header;
5. Read the position where the element is to be inserted
6. for(i=1;i<pos-1;i++)
6.1 ptr=ptr->next;
7.1 ptr1=ptr->next;
7.2 new->next=ptr1;
7.3 ptr1->prev=new;
7.4 new->prev=ptr;
7.5 ptr->next=new;
8. end
NULL 50 NULL
2200
new
After inserting a node at position: 3
2020
ptr
1000
ptr1
header
1010
NULL 10 2020
1010 20 2200
2200 30 2000
1000 40 NULL
1010
2020
1000
2000
2020 50 1000
2200
new
31


header
1010
Before deleting a node at beginning
NULL 10 2020
1010 20 1000
2020 30 2000
1000 40 NULL
1010
2020
1000
2000
1010 ptr
header
ptr1
2020
After deleting a node at beginning
NULL 10 2020
NULL 20 1000
2020 30 2000
1000 40 NULL
1010
2020
1000
2000
Algorithm:
1.ptr=header
2.ptr1=ptr->next;
3.header=ptr1;
4.if(ptr1!=NULL)
4.1.ptr1->prev=NULL;
5. free(ptr);
32


header
1010
Before deleting a node at end
NULL 10 2020
1010 20 1000
2020 30 2000
1000 40 NULL
1010
2020
1000
2000
header
1010
After deleting a node at end
ptr1
1000
ptr
2000
NULL 10 2020
1010 20 1000
2020 30 NULL
1000 40 NULL
1010
2020
1000
2000
Algorithm:
1. ptr=header
2. while(ptr->next!=NULL)
2.1 ptr=ptr->next;
3. end while
4. ptr1=ptr->prev;
5. ptr1->next=NULL;
6. free (ptr);
33

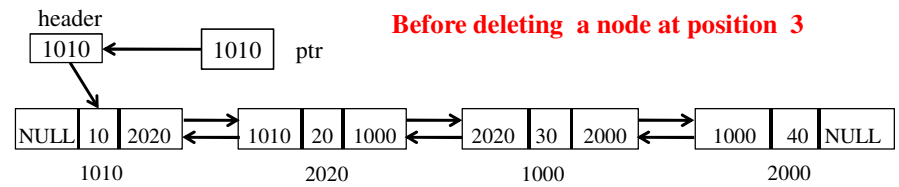
## Deletion at any position

**Algorithm:**

1. ptr=header;
2. while(ptr->next!=NULL)
   2.1 for(i=0;i<pos-1;i++)
      2.1.1 ptr=ptr->next;
   2.2 if(i = = pos-1)
      2.2.1 break;
3. end while
4. if(ptr = = header)
   //if the deleted item is first node
   4.1 ptr1=ptr->next;
   4.2 ptr1->prev=NULL;
   4.3 header=ptr1;
   4.4 end if
5. else
   5.1 ptr1=ptr->prev;
   5.2 ptr2=ptr->next;
   5.3 ptr1->next=ptr2;
   5.4 ptr2->prev=ptr1;
6. end else
7. end if

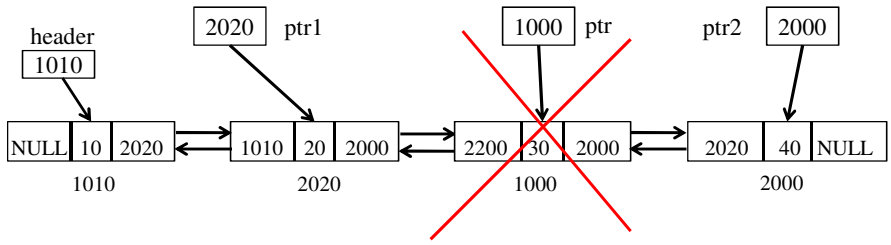


**Before deleting a node at position 3**

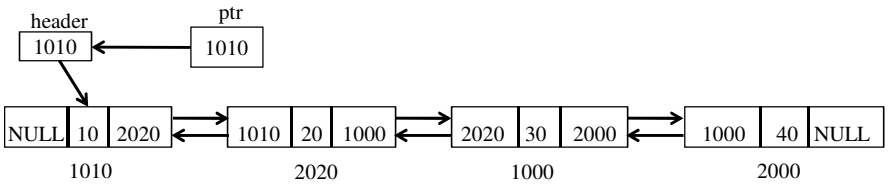



5.else
   5.1 ptr1=ptr->prev;
   5.2 ptr2=ptr->next;
   5.3 ptr1->next=ptr2;
   5.4 ptr2->prev=ptr1;

**After deleting a node at position 3**



# Traversing the elements of a list




**Forward Order : 10 20 30 40**

**Reverse Order : 40 30 20 10**

**Algorithm:**

1. ptr=header;
2. if(header = = NULL)
   2.1 printf("The list is empty\n");
3. else
   3.1 print "The elements in forward order: "
   3.2 while(ptr!=NULL)
      3.2.1 print "ptr->data";
      3.2.2 if(ptr->next = = NULL)
         3.2.2.1 break;
      3.2.3 ptr=ptr->next;
   3.3 print "The elements in reverse order: "
   3.4. while(ptr!=header)
      3.4.1 if(ptr->next = = NULL)
         3.4.1.1 print "ptr->data";
      3.4.2 else
         3.4.2.1 ptr=ptr->prev;
         3.4.2.2 print "ptr->data";
      3.4.3 .end else
4. end else



```
/*Program to implement operations of doubleylinked list*/
#include<stdio.h>        #include<conio.h>        #include<malloc.h>
void insertion();        void deletion();        void traverse();        int i,pos,item,choice;
struct node  {
        int data;
        struct node *next;
        struct node *prev;
}*new,*header,*ptr,*ptr1,*ptr2;
void main()  {
        header=NULL;
        printf(" ***** MENU ****");
        printf("\n1.Insertion \n2.Deletion \n3.Traverse \n4.Exit\n");
        while(1) {
                printf("\n\nEnter your choice: ");
                scanf("%d",&choice);
                switch(choice)  {
                        case 1:    insertion();        break;
                        case 2:    deletion();         break;
                        case 3:    traverse();         break;
                        case 4:    exit(0);
                        default: printf("\nWrong choice");
                }/* end of switch */
        }/* end of while */
}/* end of main */
```

```
void insertion()  {
        ptr=header;
        new=malloc(sizeof(struct node));
        printf("\nEnter the item to be inserted: ");
        scanf("%d",&item);
        new->data=item;
        if(header==NULL) {
                new->prev=NULL;
                new->next=NULL;
                header=new;
        }
        else  {
                printf("\nSelect the place:");
                printf("\n1.Start \n2.Middle \n3.End\n");
                scanf("%d",&choice);
                if(choice==1)  {
                        new->next=ptr;
                        ptr->prev=new;
                        new->prev=NULL;
                        header=new;
                }/* choice1 */
```

38

```
if(choice==2)
{
printf("\nEnter the position to place the new element: ");
                scanf("%d",&pos);
                for(i=1;i<pos-1;i++)
                        ptr=ptr->next;
                        if(ptr->next==NULL)
                        {
                                new->next=NULL;
                                new->prev=ptr;
                                ptr->next=new;
                        }
                        else
                        {
                                ptr1=ptr->next;
                                new->next=ptr1;
                                ptr1->prev=new;
                                new->prev=ptr;
                                ptr->next=new;
                        }
        }/* choice2 */
```

```
if(choice==3)
        {
                while(ptr->next!=NULL)
                        ptr=ptr->next;
                new->next=NULL;
                new->prev=ptr;
                ptr->next=new;
        }
    }/* end of else */
}/* end of insertion */
```

39

```
void deletion()
{
        ptr=header;
        if(header==NULL)
                printf("The list is empty\n");
        else
        {
                printf("\Select the place:");
                printf("\n1.Start \n2.Middle \n3.End\n");
                scanf("%d",&choice);
                if(choice==1)
                {
                        printf("\nThe deleted item is: %d",ptr->data);
                        ptr1=ptr->next;
                        header=ptr1;
                        if(ptr1!=NULL)
                                ptr1->prev=NULL;
                }/* choice1 */
```

40

```
if(choice==2)  {
        printf("\nEnter the position to delete the element: ");
                scanf("%d",&pos);
                while(ptr->next!=NULL)  {
                        for(i=0;i<pos-1;i++)
                                ptr=ptr->next;
                                if(i==pos-1)
                                        break;
                }//while
                printf("\n\nThe deleted node is: %d",ptr->data);
                if(ptr==header)//deleted item is starting node
                {
                        ptr1=ptr->next;
                        ptr1->prev=NULL;
                        header=ptr1;
                }//if
                else  {
                        ptr1=ptr->prev;
                        ptr2=ptr->next;
                        ptr1->next=ptr2;
                        ptr2->prev=ptr1;
                }
        }/* choice2 */
    * end of else */
```

```
if(choice==3)
{
        while(ptr->next!=NULL)
                ptr=ptr->next;
    printf("\n\nThe deleted node is: %d",ptr->data);
        ptr1=ptr->prev;
        ptr1->next=NULL;
  }/* choice3 */
}/*end of deletion */
```

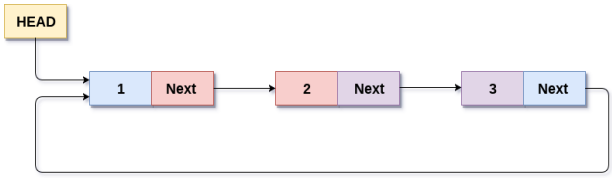
41

```
void traverse(){
    ptr=header;
    if(header==NULL)
        printf("The list is empty\n");
    else  {
        printf("\n\nThe elements in farword order: ");
        while(ptr!=NULL)  {
            printf("  %d",ptr->data);
            if(ptr->next==NULL) {
                break;
            }
            ptr=ptr->next;
        }/* end of while */
        printf("\n\nThe elements in reverse order: ");
        while(ptr!=header)  {
            if(ptr->next==NULL)
                printf("  %d",ptr->data);
            else
                printf("  %d",ptr->data);
            ptr=ptr->prev;
        }/* end of while */
        printf("  %d",ptr->data);
    }/* end of else */
}/* end of traverse() */
```

42

# Circular Singly Linked List

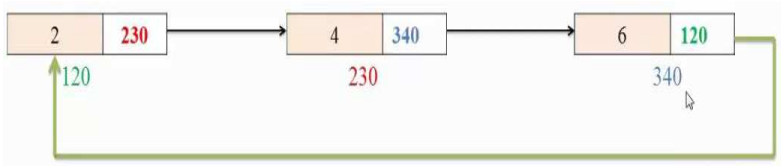
- In a circular Singly linked list, the last node of the list contains a pointer to the first node of the list.
- Circular linked list are mostly used in task maintenance in operating systems.
- We traverse a circular singly linked list until we reach the same node where we started.
- The circular singly liked list has no beginning and no ending.
- Node creation is same as singly link list.




43

Circular Singly Linked List

# Continue




44

## Circular Singly linked list operations

**Insertion:**

- **Insertion of a node at the beginning**
- **Insertion of a node at the ending**
- **Insertion of a node at any position in the list**

**Deletion:**

- **Deletion at beginning**
- **Deletion at the ending**
- **Deletion at any position**

**Display:**

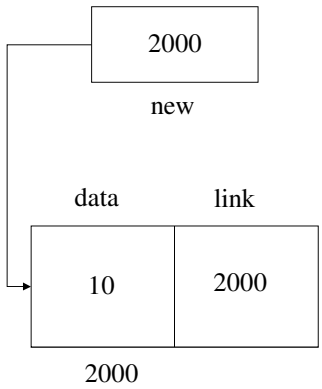
- **Displaying/Traversing the elements of a list**

45

# Circular Singly Linked Lists

## Node Structure

```
struct node
{
    int data;
    struct node *link;
}*new, *ptr, *header;
```



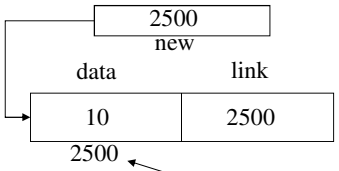

## Creating a node

```
new = malloc (sizeof(struct node));
new -> data = 10;
new -> link = new;
```

46

# Inserting a node at the beginning

## Create a node that is to be inserted

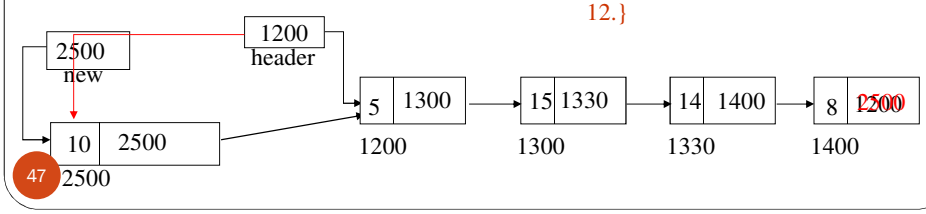



## If the list is empty


**Algorithm:**

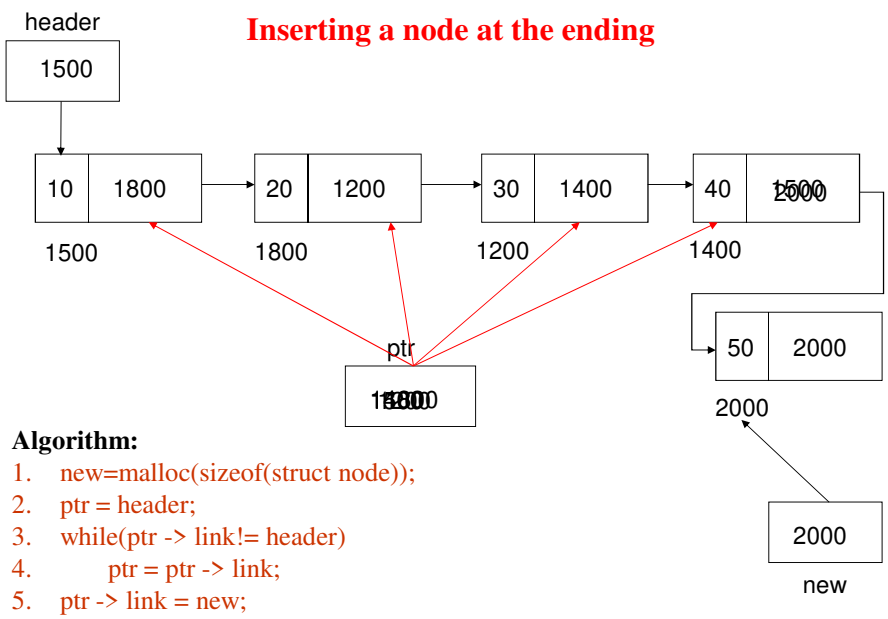
1.Create a new node.
2. if (header = = NULL)
3. header = new;
4. new -> link = new;
5. while(ptr -> link!= header)
6. ptr = ptr -> link;
7. ptr -> link = new;
8.else
9.{
10. new -> link = header;
11. header = new;
12.}

## If the list is not empty




47

## Inserting a node at the ending

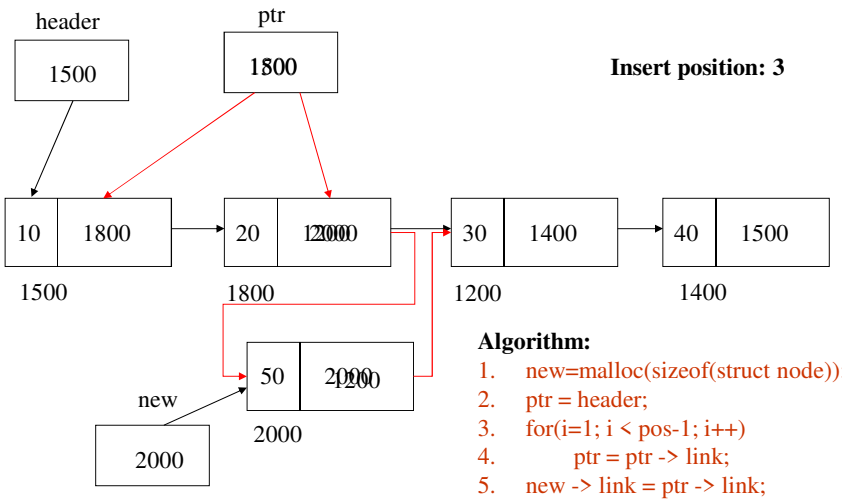



**Algorithm:**

1. new=malloc(sizeof(struct node));
2. ptr = header;
3. while(ptr -> link!= header)
4. ptr = ptr -> link;
5. ptr -> link = new;
6. new->link = header;

48

## Inserting a node at the given position

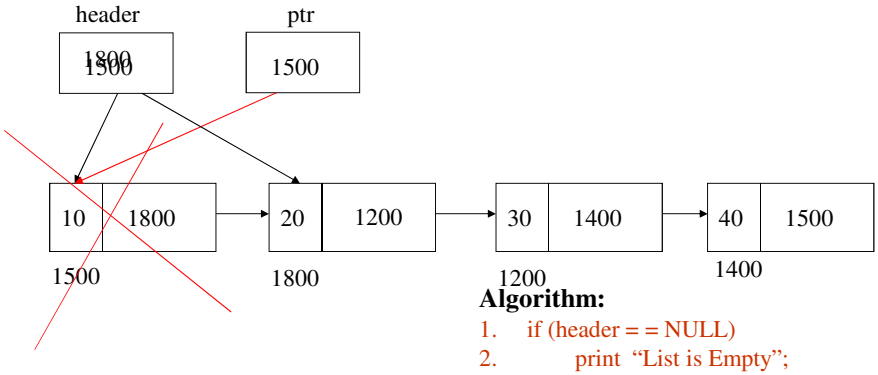



**Algorithm:**

1. new=malloc(sizeof(struct node));
2. ptr = header;
3. for(i=1; i < pos-1; i++)
4. ptr = ptr -> link;
5. new -> link = ptr -> link;
6. ptr -> link = new;

49

## Deleting a node at the beginning

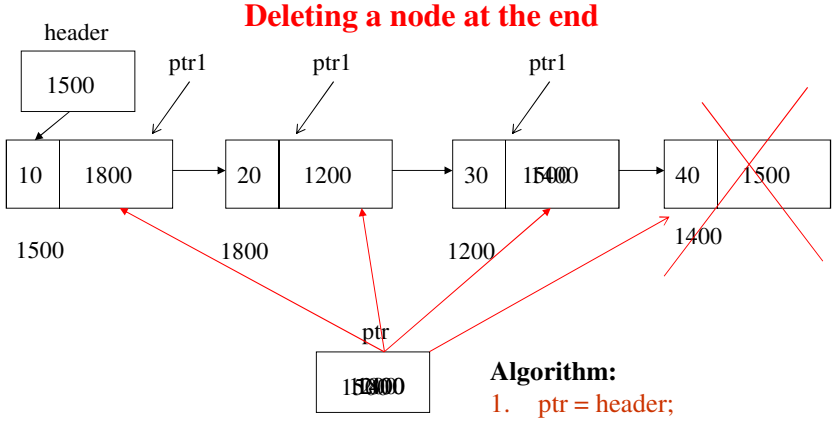



**Algorithm:**

```
if (header = = NULL)
        print  "List is Empty";
else
{
    while (ptr->link!=header)
            ptr= ptr->link;
ptr->link=header->link;
        ptr = header;
        header = header -> link;
        free(ptr);
}
```

50

## Deleting a node at the end

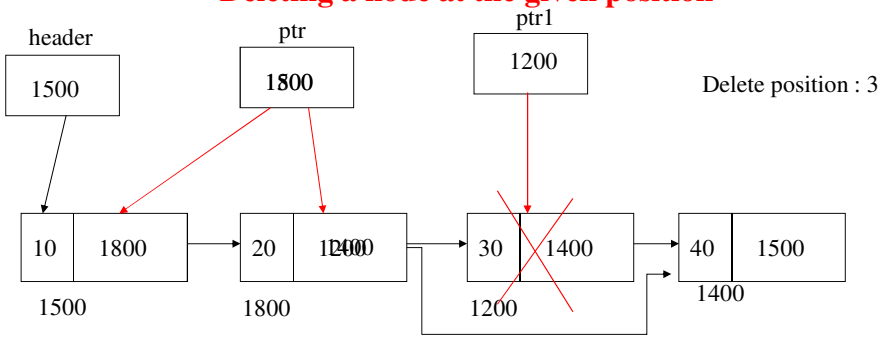



**Algorithm:**

```
ptr = header;
while(ptr -> link != header)
        ptr1=ptr;
        ptr = ptr -> link;
ptr1 -> link = header;
free(ptr);
```

51

## Deleting a node at the given position

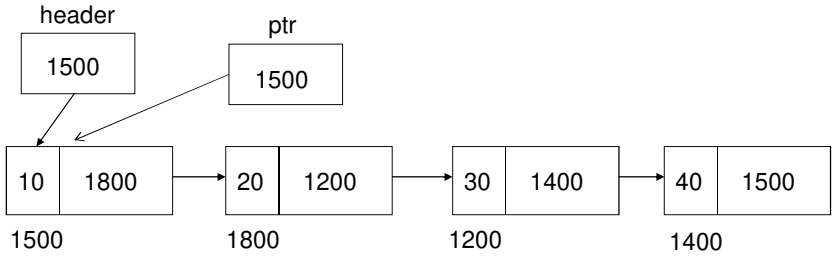



**Algorithm:**

1. ptr = header ;
2. for(i=1;i<pos-1;i++)
3. ptr = ptr -> link;
4. ptr1 = ptr -> link;
5. ptr -> link = ptr1-> link;
6. free(ptr1);

52

## Traversing the elements of a list




**Algorithm:**

1. if(header = = NULL)
2. print "List is empty";
3. else
4. for (ptr = header ; ptr->link != header ; ptr = ptr -> link)
5. print "ptr->data";

53

# Polynomial Representation

## Algebraic Expression

- If we start with variables such as $x$, $y$, and $z$ and some real numbers, and combine them by using addition, subtraction, multiplication, division, powers, and roots, then we obtain an algebraic expression.
- Some examples are:

$$2x^2 - 3x + 4 \qquad \sqrt{x} + 10 \qquad \frac{y - 2z}{y^2 + 4}$$

65

## Monomial, Binomial, & Trinomial

- A monomial is an expression of the form $ax^k$—where $a$ is a real number and $k$ is a nonnegative integer.
  - A binomial is a sum of two monomials.

    Example: $3x + 4$
  - A trinomial is a sum of three monomials.

    Example: $3x^2 + 3y^2 + 2$

66

## Polynomial

- In general, a sum of monomials is called a polynomial.
- For example, the first expression listed below is a polynomial, but the other two are not.

| | | |
|---|---|---|
| $2x^2 - 3x + 4$ | $\sqrt{x} + 10$ | $\dfrac{y - 2z}{y^2 + 4}$ |

67

## Polynomial—Definition

- A polynomial in the variable $x$ is an expression of the form:

$$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

where:

- $a_0, a_1, \ldots, a_n$ are real numbers.
- $n$ is a nonnegative integer.

68

## Degree

- Note that the degree of a polynomial is the highest power of the variable that appears in the polynomial.

| Polynomial | Type | Terms | Degree |
|---|---|---|---|
| $2x^2 - 3x + 4$ | trinomial | $2x^2, -3x, 4$ | 2 |
| $x^8 + 5x$ | binomial | $x^8, 5x$ | 8 |
| $8 - x + x^2 - \frac{1}{2}x^3$ | four terms | $-\frac{1}{2}x^3, x^2, -x, 8$ | 3 |
| $5x + 1$ | binomial | $5x, 1$ | 1 |
| $9x^5$ | monomial | $9x^5$ | 5 |
| $6$ | monomial | $6$ | 0 |

69

# Adding and Subtracting Polynomials

## Combining Algebraic Expressions

- We add and subtract polynomials by using the properties of real numbers.

71

## Adding Algebraic Expressions

- The idea is to combine like terms—**terms with the same variables raised to the same powers**—using the ***Distributive Property.***
  - For instance,

$$5x^7 + 3x^7 = (5 + 3)x^7$$
$$= 8x^7$$

72

## Subtracting Polynomials

- In subtracting polynomials, we have to remember that:
  - If a minus sign precedes an expression in parentheses, the sign of every term within the parentheses is changed when we remove the parentheses:

$$\boldsymbol{-(b + c) = -b - c}$$

  - This is simply a case of the Distributive Property, $a(b + c) = ab + ac$, with $a = -1$.

73

## E.g. 1—Adding and Subtracting Polynomials

(a) Find the sum
$(x^3 - 6x^2 + 2x + 4) + (x^3 + 5x^2 - 7x)$.

(b) Find the difference
$(x^3 - 6x^2 + 2x + 4) - (x^3 + 5x^2 - 7x)$.

74

Example (a)

## E.g. 1—Adding Polynomials

- $(x^3 - 6x^2 + 2x + 4) + (x^3 + 5x^2 - 7x)$

$= (x^3 + x^3) + (-6x^2 + 5x^2) + (2x - 7x) + 4$ (Group like terms)

$= 2x^3 - x^2 - 5x + 4$ (Combine like terms)

75

Example (b)

## E.g. 1—Subtracting Polynomials

- $(x^3 - 6x^2 + 2x + 4) - (x^3 + 5x^2 - 7x)$

  $= x^3 - 6x^2 + 2x + 4 - x^3 - 5x^2 + 7x$ (Distributive property)

  $= (x^3 - x^3) + (-6x^2 - 5x^2) + (2x + 7x) + 4$ (Group like terms)

  $= -11x^2 + 9x + 4$ (Combine like terms)

76

# Multiplying Algebraic Expressions

## Multiplying Polynomials

- To find the product of polynomials or other algebraic expressions, we need to **use the Distributive Property repeatedly.**

78

## Multiplying Polynomials

- In particular, using it three times on the product of two binomials, we get:

$$\begin{aligned}(a+b)(c+d) &= a(c+d)+b(c+d)\\ &= ac+ad+bc+bd\end{aligned}$$

- This says that **we multiply the two factors by multiplying each term in one factor by each term in the other factor and adding these products**.

79

## FOIL

- Schematically, we have:

$$(a + b)(c + d) = \underset{\substack{\uparrow \\ \text{F}}}{ac} + \underset{\substack{\uparrow \\ \text{O}}}{ad} + \underset{\substack{\uparrow \\ \text{I}}}{bc} + \underset{\substack{\uparrow \\ \text{L}}}{bd}$$

  - The acronym **FOIL** helps us to remember that the product of two binomials is the sum of the products of the **first terms, the outer terms, the inner terms, and the last terms**.

80

## Multiplying Polynomials

- In general, we can multiply two algebraic expressions by using:
  - The Distributive Property.
  - The Laws of Exponents.

| **Law of Exponents** For $a \neq 0, b \neq 0$ | |
|---|---|
| Product | $a^x \times a^y = a^{x+y}$ |
| Quotient | $a^x \div a^y = a^{x-y}$ |
| Power | $(a^x)^y = a^{xy}$ |
| Zero Exponent | $a^0 = 1$ |
| Negative Exponent | $a^{-x} = \frac{1}{a^x}$ |

81

## E.g. 2—Multiplying Binomials Using FOIL

$(2x + 1)(3x - 5)$

$= 6x^2 - 10x + 3x - 5$ (Distributive Property)

$= 6x^2 - 7x - 5$ (Combine like terms)

82

## Multiplying Trinomials and Polynomials

- When we multiply trinomials and other polynomials with more terms:
  - We use the Distributive Property.
  - The next example illustrates both methods.

83

Solution 1

## E.g. 3—Multiplying Polynomials

Using the Distributive Property

$(2x + 3)(x^2 - 5x + 4)$

$= 2x(x^2 - 5x + 4) + 3(x^2 - 5x + 4)$ (Distributive Property)

$= (2x^3 - 10x^2 + 8x) + (3x^2 - 15x + 12)$ (Distributive Property)

$= 2x^3 - 7x^2 - 7x + 12$ (Combine like terms)

84

Solution 2

## E.g. 3—Multiplying Polynomials

Using Table Form

| | |
|---|---|
| $x^2 - 5x + 4$ | (First factor) |
| $2x + 3$ | (Second factor) |
| $3x^2 - 15x + 12$ | (Multiply first factor by 3) |
| $2x^3 - 10x^2 + 8x$ | (Multiply first factor by $2x$) |
| $2x^3 - 7x^2 - 7x + 12$ | (Add like terms) |

85

# Special Product Formulas

## Special Product Formulas

- Certain types of products occur so frequently that you should memorize them.
  - You can verify the following formulas by performing the multiplications.

**Special Product Formulas**

If $A$ and $B$ are any real numbers or algebraic expressions, then

| | |
|---|---|
| **1.** $(A + B)(A - B) = A^2 - B^2$ | Sum and product of same terms |
| **2.** $(A + B)^2 = A^2 + 2AB + B^2$ | Square of a sum |
| **3.** $(A - B)^2 = A^2 - 2AB + B^2$ | Square of a difference |
| **4.** $(A + B)^3 = A^3 + 3A^2B + 3AB^2 + B^3$ | Cube of a sum |
| **5.** $(A - B)^3 = A^3 - 3A^2B + 3AB^2 - B^3$ | Cube of a difference |

87

## Principle of Substitution

- The key idea in using these formulas (or any other formula in algebra) is the **Principle of Substitution**:
  - We may substitute any algebraic expression for any letter in a formula.

88

## Principle of Substitution

- For example, to find $(x^2 + y^3)^2$, we use Product Formula 2—substituting $x^2$ for $A$ and $y^3$ for $B$—to get:

$$(x^2 + y^3)^2 = (x^2)^2 + 2(x^2)(y^3) + (y^3)^2$$
$$= x^4 + 2x^2y^3 + y^6$$

89

## E.g. 4—Using the Special Product Formulas

Use the Special Product Formulas to find:

(a) $(3x + 5)^2$

(b) $(x^2 - 2)^3$

90

Example (a)

## E.g. 4—Special Product Formulas

Substituting $A = 3x$ and $B = 5$ in Product Formula 2, we get:

$$\begin{aligned}(3x + 5)^2 &= (3x)^2 + 2(3x)(5) + 5^2 \\ &= 9x^2 + 30x + 25\end{aligned}$$

91

Example (b)

## E.g. 4—Special Product Formulas

- Substituting $A = x^2$ and $B = 2$ in Product Formula 5, we get:

$$(x^2 - 2)^3 = (x^2)^3 - 3(x^2)^2(2) + 3(x)^2(2)^2 - 2^3$$
$$= x^6 - 6x^4 + 12x^2 - 8$$

92

## E.g. 5—Using the Special Product Formulas

- Use the Special Product Formulas to find:

(a) $(2x - \sqrt{y})(2x + \sqrt{y})$

(b) $(x + y - 1)(x + y + 1)$

93

Example (a)

## E.g. 5—Special Product Formulas

- Substituting $A = 2x$ and $B = \sqrt{y}$ in Product Formula 1, we get:

$$(2x - \sqrt{y})(2x + \sqrt{y}) = (2x)^2 - (\sqrt{y})^2$$
$$= 4x^2 - y$$

94

Example (b)

## E.g. 5—Special Product Formulas

If we group $x + y$ together and think of this as one algebraic expression, we can use Product Formula 1 with $A = x + y$ and $B = 1$.

$$(x + y - 1)(x + y + 1) = [(x + y) - 1][(x + y) + 1]$$
$$= (x + y)^2 - 1^2$$
$$= x^2 + 2xy + y^2 - 1$$

95

## Polynomials in Several Variables

- A **polynomial in two variables**, $x$ and $y$, contains the sum of one or more monomials in the form $\boldsymbol{ax^ny^m}$.
- The constant, $\boldsymbol{a}$, is the **coefficient**.
- The exponents, $\boldsymbol{n}$ **and** $\boldsymbol{m}$, represent whole numbers.
- The d**egree** of the monomial $\boldsymbol{ax^ny^m}$ **is** $\boldsymbol{n + m}$.
- The **degree of a polynomial in two variables** is the highest degree of all its terms.

96

## Example: Adding Polynomials in Two Variables

**Addition:**

$$(x^3 - 4x^2y + 5xy^2 - y^3) + (x^3 + 6x^2y + y^3)$$

$$= (x^3 + x^3) + (-4x^2y + 6x^2y) + 5xy^2 + (-y^3 + y^3)$$

$$= 2x^3 + 2x^2y + 5xy^2$$

97

## Example: Subtracting Polynomials in Two Variables

- Subtract: $(x^3 - 4x^2y + 5xy^2 - y^3) - (x^3 - 6x^2y + y^3)$

$$(x^3 - 4x^2y + 5xy^2 - y^3) - (x^3 - 6x^2y + y^3)$$

$$= (x^3 - 4x^2y + 5xy^2 - y^3) + (-x^3 + 6x^2y - y^3)$$

$$= (x^3 - x^3) + (-4x^2y + 6x^2y) + 5xy^2 + (-y^3 - y^3)$$

$$= 2x^2y + 5xy^2 - 2y^3$$

98

## Example: Multiplying Polynomials in Two Variables

**Multiply:** $(7x - 6y)(3x - y)$

Each of the factors is a binomial, so we can apply the FOIL method for this multiplication.

F $\longrightarrow$ $7x \cdot 3x = 21x^2$

O $\longrightarrow$ $7x \cdot -y = -7xy$

I $\longrightarrow$ $-6y \cdot 3x = -18xy$

L $\longrightarrow$ $-6y \cdot -y = 6y^2$

$$(7x - 6y)(3x - y) = 21x^2 - 7xy - 18xy + 6y^2$$
$$= 21x^2 - 25xy + 6y^2$$

99

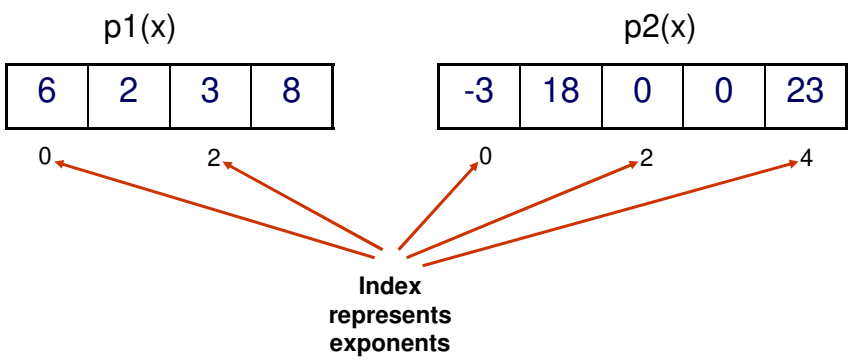
## Polynomial (continued)

- How to implement this?
- There are different ways of implementing the polynomial:
  - Array (not recommended)
  - Linked List (preferred and recommended)

100

## Polynomial (continued)

- Array Implementation:
- $p1(x) = 8x^3 + 3x^2 + 2x + 6$
- $p2(x) = 23x^4 + 18x - 3$




101

## Polynomial (continued)

- This is why arrays aren't good to represent polynomials:

- $p3(x) = 16x^{21} - 3x^5 + 2x + 6$

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 6 | 2 | 0 | 0 | -3 | 0 | ..... | 0 | 16 |

**WASTE OF SPACE!**

102

## Polynomial (continued)

- Advantages of using an Array:
  - only good for non-sparse polynomials.
  - ease of storage and retrieval.

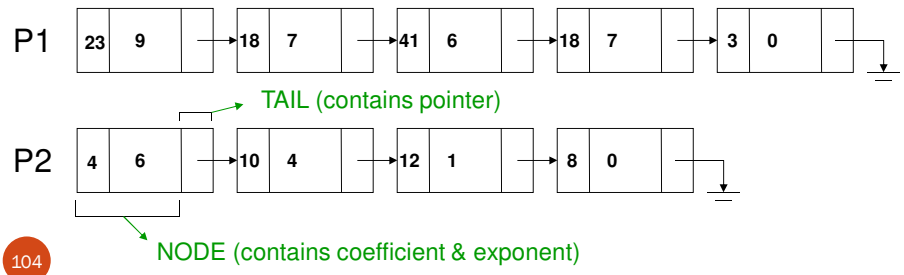
- Disadvantages of using an Array:
  - have to allocate array size ahead of time.
  - huge array size required for sparse polynomials.
  - Waste of space and runtime.

103

## Polynomial (continued)

- Linked list Implementation:

- $p1(x) = 23x^9 + 18x^7 + 41x^6 + 163x^4 + 3$
- $p2(x) = 4x^6 + 10x^4 + 12x + 8$




104

## Polynomial (continued)

- Advantages of using a Linked list:
  - save space (don't have to worry about sparse polynomials) and easy to maintain
  - don't need to allocate list size and can declare nodes (terms) only as needed

- Disadvantages of using a Linked list :
  - can't go backwards through the list
  - can't jump to the beginning of the list from the end.

105

# Polynomials

$$A(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x^1 + a_0 x^0$$

## Representation

```
struct polynode
{
    int coef;
    int exp;
    struct polynode * next;
};
```

| coef | exp | next |
|---|---|---|

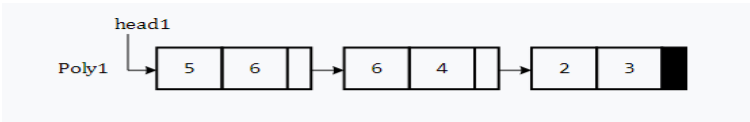
106

# Addition of two polynomials

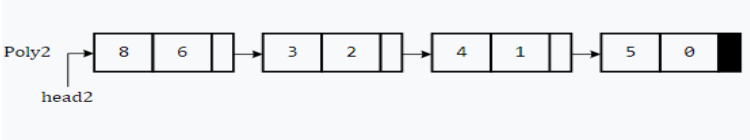
- Let there be two polynomials which we are required to add together:

$$5x^6+6x^4+2x^3$$

$$8x^6+3x^2+4x+5$$




Polynomial 1 represented with a Linked List




Polynomial 2 represented with a Linked List

107

# Addition of two polynomials

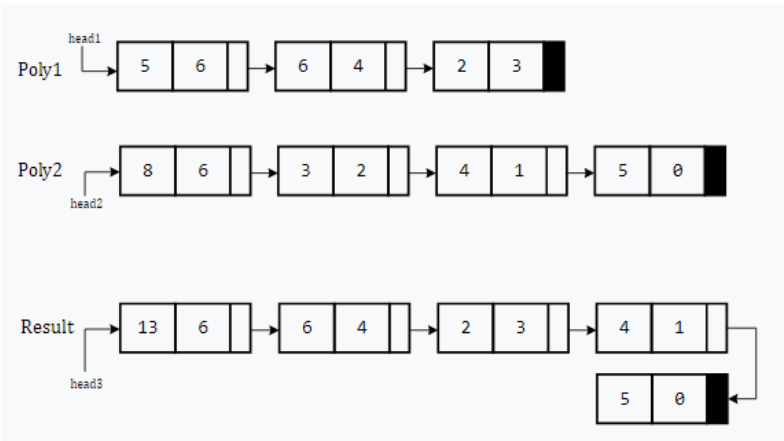
- **if(ptr1->expo == ptr2->expo)**
  //Add the coefficients and insert the newly created node in the resultant linked list and make ptr1 and ptr2 point to the next nodes.

- **if (ptr1->expo > ptr2->expo)**
  //Insert the node pointed by ptr1 in the resultant linked list and make ptr1 point to the next node.

- **if (ptr1->expo < ptr2->expo)**
  //Insert the node pointed by ptr2 in the resultant linked list and make ptr2 point to the next node.

108

# Addition of two polynomials




109

## Multiplication of two polynomials

- Consider the following polynomials:

**4x^3 + 3x^2 + 1**

**5x^3 + 7x + 5**

- Our target here is to multiply both the polynomials with each other, i.e., ***each term of the polynomial 1 must be multiplied with each term of the polynomials 2***.

110

## Multiplication of two polynomials

- Thus the resultant polynomials so obtained must look like:

$$(4 \times 5)x^{3+3} + (4 \times 7)x^{3+1} + (4 \times 5)x^{3+0} + (3 \times 5)x^{2+3} + (3 \times 7)x^{2+1}$$
$$+(3 \times 5)x^{2+0} + (1 \times 5)x^{0+3} + (1 \times 7)x^{0+1} + (1 \times 5)x^{0+0}$$

- As it can be noticed, each term gets multiplied with every other term, where the coefficients get multiplied and the exponents get added together.

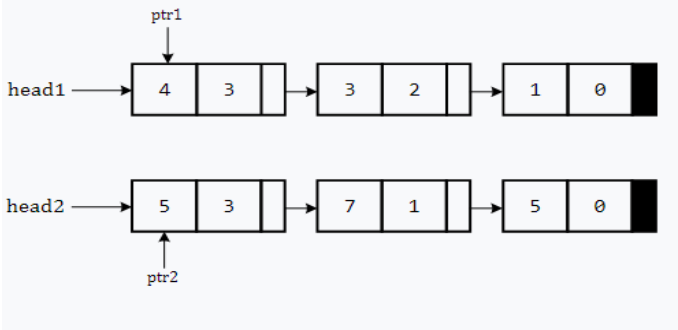
- The simplified final solution OR the resultant polynomial is=

$$20x^6 + 28x^4 + 20x^3 + 15x^5 + 21x^3 + 15x^2 + 5x^3 + 7x + 5$$

111

# Multiplication of two polynomials

- The representation of the polynomial equations in the form of linked lists:




Representation of the Polynomials using Linked Lists

112

# Multiplication of two polynomials

- **Process to multiply these polynomials:**
  - Here, we have two pointers head1 and head2 pointing to the first node of their respective linked lists which are representing the polynomials expressions.
  - The next task is to traverse both the lists and simply multiply the terms. For traversal, we need two pointers ptr1 and ptr2. We cannot use head1 or head2 for the same purpose as then we would lose the reference of both the linked lists in the process.

113

# Multiplication of two polynomials

- Here, we also need a nested loop as each term of the first polynomial must be multiplied with every term of the second polynomial.
- Consider below:

```
while(ptr1 != NULL)
{
        while(ptr2 != NULL)
        {
        ...
        }
}
```

114

# Multiplication of two polynomials

```
{
res1 = ptr1->coeff * ptr2->coeff;
res2 = ptr1->expo + ptr2->expo;
head3 =insert(head3, res1, res2);
ptr2 = ptr2->link;
}
ptr1 = ptr1->link;
}
```

- res1 holds the value for the product of the coefficients.
- res2 holds the value for the addition of the exponents.

115