DATA STRUCTURES-PART 1

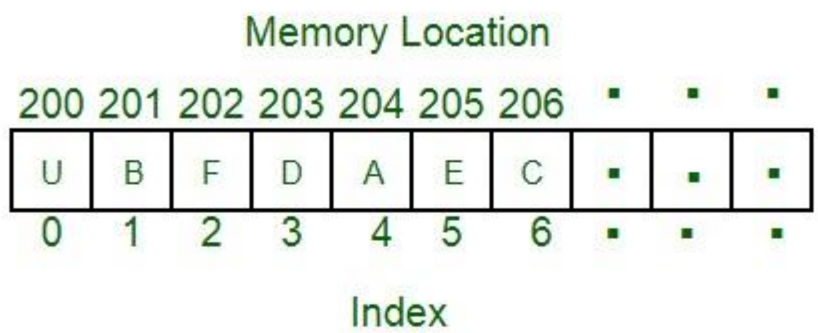
• Data Structures – abstract data types – time and space complexity (O, Ώ, θ) – practical complexities. Recursive algorithms. Randomized algorithms.

• Arrays – representation-address calculation, sparse matrix representation, polynomial and sparse polynomial representation.

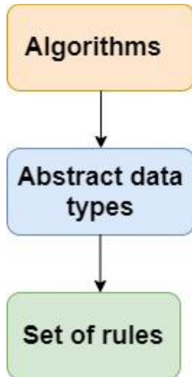
• Linked list – single, doubly, circular lists. Header and trailer nodes, basic operations on linked lists (insertion, deletion, merging, concentration, search), linked polynomial, sparse matrix representation using linked list.

DATA STRUCTURES

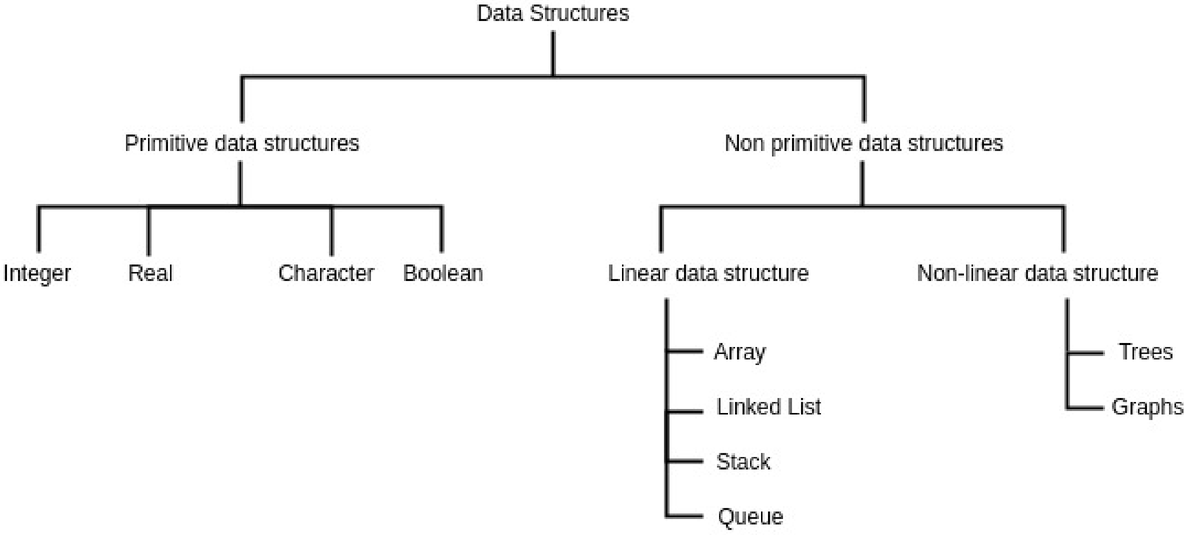
- A data structure is a particular way of organizing data in a computer so that it can be used effectively.
- For example, we can store a list of items having the same data-type using the array data structure.



- The data structure name indicates itself that organizing the data in memory.
- There are many ways of organizing the data in the memory as we have already seen one of the data structures, i.e., array in C language.
- Array is a collection of memory elements in which data is stored sequentially, i.e., one after another.
- In other words, we can say that array stores the elements in a continuous manner.
- This organization of data is done with the help of an array of data structures.
- There are also other ways to organize the data in memory.
- The data structure is not any programming language like C, C++, java, etc.
- It is a set of algorithms that we can use in any programming language to structure the data in the memory.
- To structure the data in memory, 'n' number of algorithms were proposed, and all these algorithms are known as Abstract data types.
- These abstract data types are the set of rules.



Types of Data Structures



- There are two types of data structures:
  - ✓ **Primitive data structure**
  - ✓ **Non-primitive data structure**

Primitive Data structure

- The primitive data structures are primitive data types.
- The int, char, float, double, and pointer are the primitive data structures that can hold a single value.

Non-Primitive Data structure

- The non-primitive data structure is divided into two types:
  - ✓ Linear data structure
  - ✓ Non-linear data structure

Linear Data Structure

- The arrangement of data in a sequential manner is known as a linear data structure.

- The data structures used for this purpose are Arrays, Linked list, Stacks, and Queues.
- In these data structures, one element is connected to only one another element in a linear form.
- When one element is connected to the 'n' number of elements known as a non-linear data structure.
- The best example is trees and graphs.
- In this case, the elements are arranged in a random manner.

➢ Data structures can also be classified as:

Static data structure:

- It is a type of data structure where the size is allocated at the compile time. Therefore, the maximum size is fixed.

Dynamic data structure:

- It is a type of data structure where the size is allocated at the run time. Therefore, the maximum size is flexible.

Major Operations

The major or the common operations that can be performed on the data structures are:

- ✓ **Searching:** We can search for any element in a data structure.
- ✓ **Sorting:** We can sort the elements of a data structure either in an ascending or descending order.
- ✓ **Insertion:** We can also insert the new element in a data structure.
- ✓ **Updation:** We can also update the element, i.e., we can replace the element with another element.
- ✓ **Deletion:** We can also perform the delete operation to remove the element from the data structure.

- A data structure is a way of organizing the data so that it can be used efficiently.
- Here, we have used the word efficiently, which in terms of both the space and time.
- For example, a stack is an ADT (Abstract data type) which uses either arrays or linked list data structure for the implementation.
- Therefore, we conclude that we require some data structure to implement a particular ADT.
- An ADT tells what is to be done and data structure tells how it is to be done.
- In other words, we can say that ADT gives us the blueprint while data structure provides the implementation part.
- As the different data structures can be implemented in a particular ADT, but the different implementations are compared for time and space.
- For example, the Stack ADT can be implemented by both Arrays and linked list.
- Suppose the array is providing time efficiency while the linked list is providing space efficiency, so the one which is the best suited for the current user's requirements will be selected.

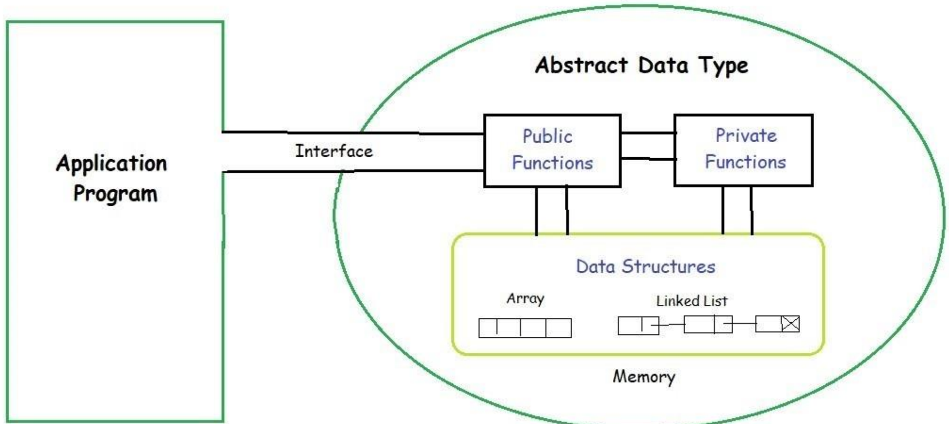
Advantages of Data structures

The following are the advantages of a data structure:

- **Efficiency:**
    - If the choice of a data structure for implementing a particular ADT is proper, it makes the program very efficient in terms of time and space.
- **Reusability:**
    - The data structure provides reusability means that multiple client programs can use the data structure.
- **Abstraction:**
    - The data structure specified by an ADT also provides the level of abstraction.
    - The client cannot see the internal working of the data structure, so it does not have to worry about the implementation part.
    - The client can only see the interface.

ABSTRACT DATA TYPE

- Abstract Data type (ADT) is a type (or class) for objects whose behavior is defined by a set of values and a set of operations.
- The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented.
- It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations.
- It is called "abstract" because it gives an implementation-independent view.
- The process of providing only the essentials and hiding the details is known as abstraction.



- The user of data type does not need to know how that data type is implemented, for example, we have been using Primitive values like int, float, char data types only with the knowledge that these data type can operate and be performed on without any idea of how they are implemented.
- So a user only needs to know what a data type can do, but not how it will be implemented.

- Think of ADT as a black box which hides the inner structure and design of the data type.

THREE ADTs

- ✓ **List ADT**
- ✓ **Stack ADT**
- ✓ **Queue ADT**

1. List ADT

- The data is generally stored in key sequence in a list which has a head structure consisting of count, pointers and address of compare function needed to compare the data in the list.
- The data node contains the pointer to a data structure and a self-referential pointer which points to the next node in the list.

- **The List ADT Functions is given below:**

- ✓ get() – Return an element from the list at any given position.
- ✓ insert() – Insert an element at any position of the list.
- ✓ remove() – Remove the first occurrence of any element from a non-empty list.
- ✓ removeAt() – Remove the element at a specified location from a non-empty list.
- ✓ replace() – Replace an element at any position by another element.
- ✓ size() – Return the number of elements in the list.
- ✓ isEmpty() – Return true if the list is empty, otherwise return false.
- ✓ isFull() – Return true if the list is full, otherwise return false.

2. Stack ADT

- In Stack ADT Implementation instead of data being stored in each node, the pointer to data is stored.
- The program allocates memory for the data and address is passed to the stack ADT.
- The head node and the data nodes are encapsulated in the ADT.
- The calling function can only see the pointer to the stack.
- The stack head structure also contains a pointer to top and count of number of entries currently in stack.

- ✓ push() – Insert an element at one end of the stack called top.
- ✓ pop() – Remove and return the element at the top of the stack, if it is not empty.
- ✓ peek() – Return the element at the top of the stack without removing it, if the stack is not empty.
- ✓ size() – Return the number of elements in the stack.
- ✓ isEmpty() – Return true if the stack is empty, otherwise return false.
- ✓ isFull() – Return true if the stack is full, otherwise return false.

3. Queue ADT

- The queue abstract data type (ADT) follows the basic design of the stack abstract data type.
- Each node contains a void pointer to the data and the link pointer to the next element in the queue.
- The program's responsibility is to allocate memory for storing the data.
  - ✓ enqueue() – Insert an element at the end of the queue.
  - ✓ dequeue() – Remove and return the first element of the queue, if the queue is not empty.
  - ✓ peek() – Return the element of the queue without removing it, if the queue is not empty.
  - ✓ size() – Return the number of elements in the queue.
  - ✓ isEmpty() – Return true if the queue is empty, otherwise return false.
  - ✓ isFull() – Return true if the queue is full, otherwise return false.

Features of ADT:

- ✓ **Abstraction**: The user does not need to know the implementation of the data structure.
- ✓ **Better Conceptualization**: ADT gives us a better conceptualization of the real world.
- ✓ **Robust**: The program is robust and has the ability to catch errors.

ALGORITHMS AND COMPLEXITIES

Algorithm

- An algorithm is a finite set of instructions, those if followed, accomplishes a particular task.
- It is not language specific; we can use any language and symbols to represent instructions.

The criteria of an algorithm

- **Input**: Zero or more inputs are externally supplied to the algorithm.
- **Output**: At least one output is produced by an algorithm.
- **Definiteness**: Each instruction is clear and unambiguous.
- **Finiteness**: In an algorithm, it will be terminated after a finite number of steps for all different cases.
- **Effectiveness**: Each instruction must be very basic, so the purpose of those instructions must be very clear to us.

Analysis of algorithms

- Algorithm analysis is an important part of computational complexities.
- The complexity theory provides the theoretical estimates for the resources needed by an algorithm to solve any computational task.

- Analysis of the algorithm is the process of analyzing the problem-solving capability of the algorithm in terms of the time and size required (the size of memory for storage while implementation).
- However, the main concern of the analysis of the algorithm is the required time or performance.

Complexities of an Algorithm

- The complexity of an algorithm computes the amount of time and spaces required by an algorithm for an input of size (n).
- The complexity of an algorithm can be divided into two types. The time complexity and the space complexity.

Time Complexity of an Algorithm

- The time complexity is defined as the process of determining a formula for total time required towards the execution of that algorithm.
- This calculation is totally independent of implementation and programming language.

Space Complexity of an Algorithm

- Space complexity is defining as the process of defining a formula for prediction of how much memory space is required for the successful execution of the algorithm.
- The memory space is generally considered as the primary memory.

ASYMPTOTIC NOTATION

- Asymptotic analysis of an algorithm refers to defining the mathematical boundation/framing of its run-time performance.
- Using asymptotic analysis, we can very well conclude the best case, average case, and worst case scenario of an algorithm.
- Asymptotic analysis is input bound i.e., if there's no input to the algorithm, it is concluded to work in a constant time.
- Other than the "input" all other factors are considered constant.
- Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation.
- For example, the running time of one operation is computed as f(n) and may be for another operation it is computed as $g(n^2)$.
- This means the first operation running time will increase linearly with the increase in n and the running time of the second operation will increase exponentially when n increases.
- Similarly, the running time of both operations will be nearly the same if n is significantly small.

Usually, the time required by an algorithm falls under three types −

✔ **Best Case** − Minimum time required for program execution.

✓ **Average Case** − Average time required for program execution.

✓ **Worst Case** − Maximum time required for program execution.

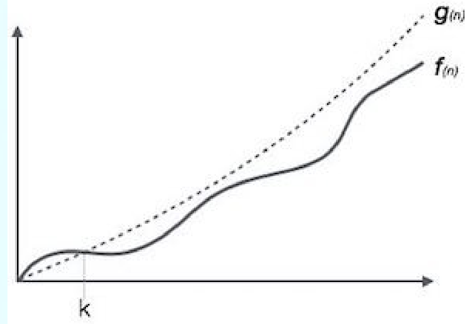
Asymptotic Notations

- Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

✓ **O Notation**
✓ **Ω Notation**
✓ **θ Notation**

✓ **Big Oh Notation, O**
- The notation O(n) is the formal way to express the upper bound of an algorithm's running time.

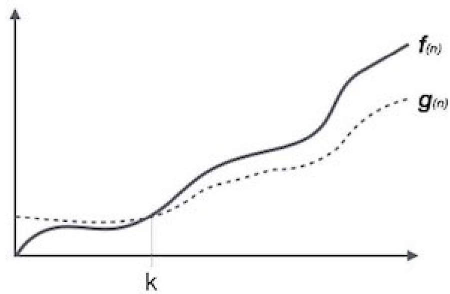
- It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.



- For example, for a function f(n)

O(f(n)) = { g(n) : there exists c > 0 and n0 such that f(n) ≤ c.g(n) for all n > n0. }

✓ **Omega Notation, Ω**
- The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time.
- It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.

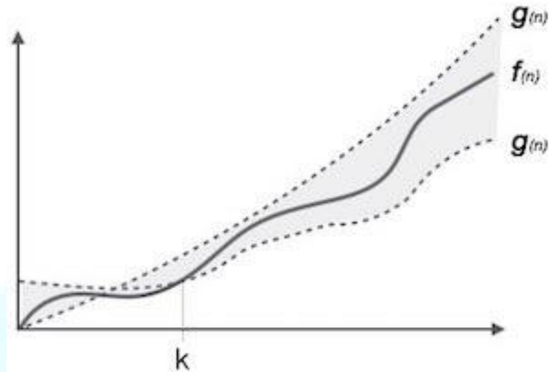


- For example, for a function f(n)

$\Omega(f(n)) \geq \{ g(n)$: there exists c > 0 and n0 such that $g(n) \leq c.f(n)$ for all n > n0. }

✓ **Theta Notation, θ**

- The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time.
- It is represented as follows –



$\theta(f(n)) = \{ g(n)$ if and only if $g(n) = O(f(n))$ and $g(n) = \Omega(f(n))$ for all n > n0. }

Common Asymptotic Notations

Following is a list of some 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	−	$2^{O(n)}$

RECURSIVE ALGORITHM

- A recursive algorithm calls itself with smaller input values and returns the result for the current input by carrying out basic operations on the returned value for the smaller input.
- Generally, if a problem can be solved by applying solutions to smaller versions of the same problem, and the smaller versions shrink to readily solvable instances, then the problem can be solved using a recursive algorithm.
- To build a recursive algorithm, you will break the given problem statement into two parts.
- **The first one is the base case, and the second one is the recursive step.**

Base Case:

- It is nothing more than the simplest instance of a problem, consisting of a condition that terminates the recursive function.
- This base case evaluates the result when a given condition is met.

Recursive Step:

- It computes the result by making recursive calls to the same function, but with the inputs decreased in size or complexity.
- For example, consider this problem statement: Print sum of n natural numbers using recursion.
- This statement clarifies that we need to formulate a function that will calculate the summation of all natural numbers in the range 1 to n.
- Hence, mathematically you can represent the function as:

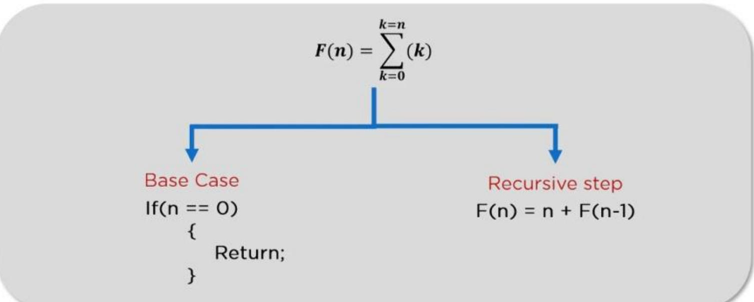
- F(n) = 1 + 2 + 3 + 4 + …………..+ (n-2) + (n-1) + n

- It can further be simplified as:

$$F(n) = \sum_{k=1}^{k=n} (k)$$

- You can breakdown this function into two parts as follows:

Breakdown of Problem Statement



Different Types of Recursion

- There are four different types of recursive algorithms.

1. Direct Recursion

- A function is called direct recursive if it calls itself in its function body repeatedly.
- To better understand this definition, look at the structure of a direct recursive program.

Example

```
int fun(int z)
{
```

```
fun(z-1);  //Recursive call

}
```

- In this program, you have a method named fun that calls itself again in its function body.
- Thus, you can say that it is direct recursive.

2. Indirect Recursion

- The recursion in which the function calls itself via another function is called indirect recursion.

Example

```
int fun1(int z)
{
  fun2(z-1);

}

int fun2(int y)
{
  fun1(y-2)

}
```

- In this example, you can see that the function fun1 explicitly calls fun2, which is invoking fun1 again.
- Hence, you can say that this is an example of indirect recursion.

3. Tailed Recursion

- A recursive function is said to be tail-recursive if the recursive call is the last execution done by the function.

Example

```
int fun(int z)

{

  printf("%d",z);

  fun(z-1);

  //Recursive call is last executed statement

}
```

- If you observe this program, you can see that the last line ADI will execute for method fun is a recursive call.
- And because of that, there is no need to remember any previous state of the program.

4. Non-Tailed Recursion

- A recursive function is said to be non-tail recursive if the recursion call is not the last thing done by the function.
- After returning back, there is something left to evaluate.

Example

```
int fun(int z)

{

  fun(z-1);

  printf("%d",z);

  //Recursive call is not the last executed statement

}
```

- In this function, you can observe that there is another operation after the recursive call.
- Hence the ADI will have to memorize the previous state inside this method block.
- That is why this program can be considered non-tail recursive.

RANDOMIZED ALGORITHM

- An algorithm that uses random numbers to decide what to do next anywhere in its logic is called Randomized Algorithm.
- For example, in Randomized Quick Sort, we use a random number to pick the next pivot (or we randomly shuffle the array).
- Typically, this randomness is used to reduce time complexity or space complexity in other standard algorithms.
- There are two main types of randomized algorithms:
  - ✓ **Las Vegas algorithms**
  - ✓ **Monte-Carlo algorithms**

✓ **Las Vegas algorithms**

- In Las Vegas algorithms, the algorithm may use the randomness to speed up the computation, but the algorithm must always return the correct answer to the input.

✓ **Monte-Carlo algorithms**

- Monte-Carlo algorithms do not have the former restriction, that is, they are allowed to give wrong return values. However, returning a wrong return value must have a small probability, otherwise that Monte-Carlo algorithm would not be of any use.

ARRAYS

- Arrays are defined as the collection of similar types of data items stored at contiguous memory locations.
- It is one of the simplest data structures where each data element can be randomly accessed by using its index number.

- In C programming, they are the derived data types that can store the primitive type of data such as int, char, double, float, etc.
- For example, if we want to store the marks of a student in 6 subjects, then we don't need to define a different variable for the marks in different subjects.
- Instead, we can define an array that can store the marks in each subject at the contiguous memory locations.

Properties of array

There are some of the properties of an array that are listed as follows -

- Each element in an array is of the same data type and carries the same size that is 4 bytes.
- Elements in the array are stored at contiguous memory locations from which the first element is stored at the smallest memory location.
- Elements of the array can be randomly accessed since we can calculate the address of each element of the array with the given base address and the size of the data element.

Representation of an array

- We can represent an array in various ways in different programming languages.
- As an illustration, let's see the declaration of array in C language -

Name
Elements
int array [10] = { 35, 33, 42, 10, 14, 19, 27, 44, 26, 31 }
Type
Size

- As per the above illustration, there are some of the following important points -

✓ Index starts with 0.
✓ The array's length is 10, which means we can store 10 elements.
✓ Each element in the array can be accessed via its index.

- **Arrays are useful because -**

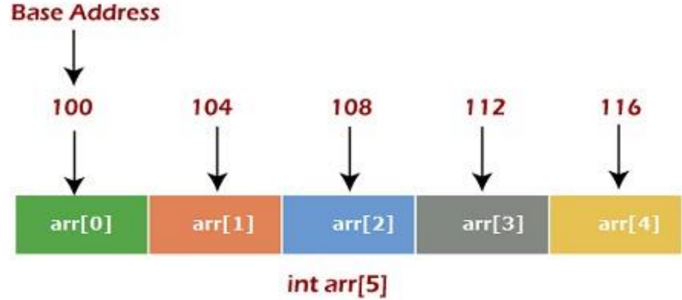
- Sorting and searching a value in an array is easier.
- Arrays are best to process multiple values quickly and easily.
- Arrays are good for storing multiple values in a single variable
  - In computer programming, most cases require storing a large number of data of a similar type.
  -To store such an amount of data, we need to define a large number of variables.
  -It would be very difficult to remember the names of all the variables while writing the programs.
  - Instead of naming all the variables with a different name, it is better to define an array and store all the elements into it.

Memory allocation of an array

- As stated above, all the data elements of an array are stored at contiguous locations in the main memory.
- The name of the array represents the base address or the address of the first element in the main memory.
- Each element of the array is represented by proper indexing.

- We can define the indexing of an array in the below ways -

✓ **0 (zero-based indexing)**: The first element of the array will be arr[0].
✓ **1 (one-based indexing)**: The first element of the array will be arr[1].
✓ **n (n - based indexing)**: The first element of the array can reside at any random index number.



- In the above image, we have shown the memory allocation of an array arr of size 5.
- The array follows a 0-based indexing approach.
- The base address of the array is 100 bytes.
- It is the address of arr[0].
- Here, the size of the data type used is 4 bytes; therefore, each element will take 4 bytes in the memory.

Access an element from the array

- We required the information given below to access any random element from the array -

✓ Base Address of the array.
✓ Size of an element in bytes.
✓ Type of indexing, array follows.

The formula to calculate the address to access an array element -

Byte address of element A[i] = base address + size * (i - first index)

Basic operations

- The basic operations supported in the array -

✓ **Traversal** - This operation is used to print the elements of the array.
✓ **Insertion** - It is used to add an element at a particular index.
✓ **Deletion** - It is used to delete an element from a particular index.
✓ **Search** - It is used to search an element using the given index or by the value.

- ✓ **Update** - It updates an element at a particular index.

Complexity of Array operations
Time and space complexity of various array operations are described in the following table.

Time complexity

Operation	Average Case	Worst Case
Access	O(1)	O(1)
Search	O(n)	O(n)
Insertion	O(n)	O(n)
Deletion	O(n)	O(n)

Space Complexity

In array, space complexity for worst case is O(n).

Advantages of Array

- ✓ Array provides the single name for the group of variables of the same type. Therefore, it is easy to remember the name of all the elements of an array.
- ✓ Traversing an array is a very simple process; we just need to increment the base address of the array in order to visit each element one by one.
- ✓ Any element in the array can be directly accessed by using the index.

Disadvantages of Array

- ✓ Array is homogenous. It means that the elements with similar data type can be stored in it.
- ✓ In array, there is static memory allocation that is size of an array cannot be altered.
- ✓ There will be wastage of memory if we store less number of elements than the declared size.

TYPES OF ARRAY

- Three types of array:

  - ✓ **One-dimensional Array**
  - ✓ **Two-dimensional Array**
  - ✓ **Three-dimensional Array**

- Two dimensional and three dimensional arrays are also called multi-dimensional arrays.
- In types of arrays, multi-dimensional arrays also include arrays with four and higher dimensions.

1. **One-dimensional Array**

- In a one-dimensional array the elements are stored in contiguous memory locations where each element is accessed by using a single index value.
- It is a linear data structure storing all the elements in sequence.

0	1	2	3	4	5	6	7	8	9
2	42	71	9	21	33	41	82	2	11

One Dimensional Array

- The elements are stored in memory in continuation and the variable declared as an array is actually a pointer to the address of first element of the array.
- This address is called the base address.
- The address of any other element can be calculated with the following formula

ADDRESS (ARRAY [K]) =BASEADDRESS (ARRAY) +WORDLENGTH*(LOWERBOUND-K)

Where

- ADDRESS -memory location of the Kth element of the array (To be calculated)
- ARRAY – name of the ARRAY
- WORDLENGTH – number of bytes required to store one element depending upon its data type like a character value needs 1 byte and an integer value needs 2 bytes.
- LOWERBOUND – index of the first element of the array.
- BASEADDRESS– Address of first element of the array

Address of the memory	Value	indexes of array elements
1001, 1002	20	1
1003, 1004	50	2
1005, 1006	102	3
1007, 1008	600	4
1009, 1010	2	5
1011, 1012	34	6
1013, 1014	500	7
1015, 1016	100	8

- In this example a snap shot of the memory displays how an array of size 8 is stored in memory.
- To calculate memory address of a given element say 6th element of the array you will put the values in the previous formula

K=6

WORDLENGTH=2 (integer data)

LOWERBOUND =1 (index of first element of the array)

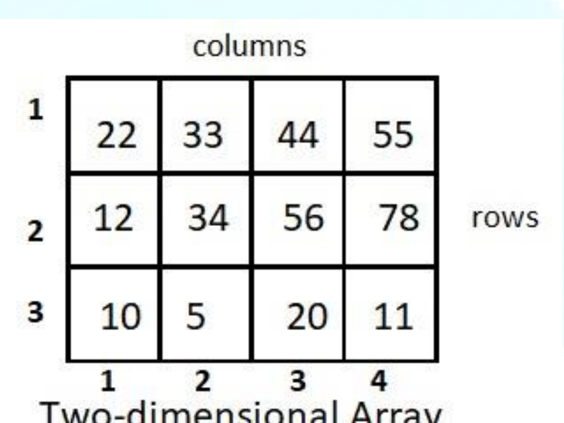
BASEADDRESS = 1001

Putting these values in formula

ADDRESS (ARRAY [6]) = 1001+ 2*(6-1) =1011 this is the address of memory location where 6th element (34) is stored as visible in the figure above

2. Two-dimensional Array

- In types of arrays, a two dimensional array is a tabular representation of data where elements are stored in rows and columns.
- A two dimensional array is actually a collection of M X N elements which has M rows and N columns.
- To access any element in a two-dimensional array two subscripts are required for defining position of an element in a specific row and column.
- The first index is for row number and second is for column index.
- In the example shown the first index values row=2 column=3 is used to access element 56.



Two dimensional arrays are stored in memory in two representations

- ✓ **Row Major Representation**
- ✓ **Column Major Representation**

Row Major Representation

- In the row major representation the storage of array elements takes place row wise.
- All elements of first row of the array are first stored in sequence followed by second row and then third, fourth and so on.
- The 2-dimensional array given in previous examples is stored in row-major representation in the figure below.

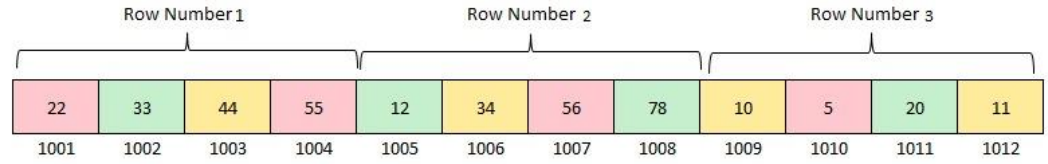
- To find the Address of any element located at I^{th} row and J^{th} column is calculated by using the formula

ADDRESS (ARRAY [I, J]) = BASEADDRESS (ARRAY) + WORDLENGTH (N (I-1) + (J-1))

Where

- ADDRESS – memory location of the element at I^{th} row and J^{th} column of the array (To be calculated)
- ARRAY – name of the ARRAY
- WORDLENGTH – number of bytes required to store one element depending upon its data type like a character values needs 1 byte and an integer value needs 2 bytes.
- N – Number of columns of the array.
- BASEADDRESS – Address of first element of the 2-D array

- In this example to calculate memory address of a given element (44) says in 1st row and 3rd column you will put the values in the formula.



I=1, J=3

N=4

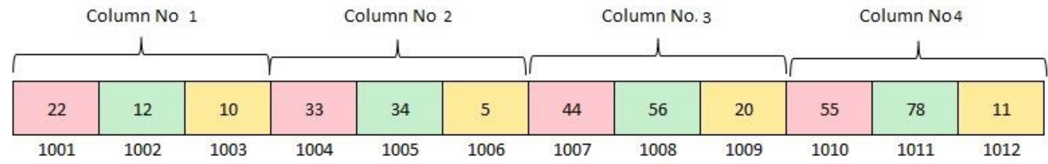
WORDLENGTH=1 (assuming only one byte is required to store these small ints)

BASEADDRESS = 1001

ADDRESS (ARRAY (6)) = 1001+ 1*(4*(1-1) + (3-1)) =1003 this is the address of memory location where 44 is stored as visible in the previous figure

Column Major Representation

- In the column major representation the storage of array elements takes place column wise.
- All elements of first column of the array are first stored in sequence followed by second column and then third, fourth and so on.
- The 2-dimensional array given in previous examples is stored in column-major representation in the figure below.



- To find the Address of any element located at Ith row and jth column is calculated by using the formula

ADDRESS (ARRAY [K]) = BASEADDRESS (ARRAY) + WORDLENGTH (M (J-1) + (I-1))

Where

- ADDRESS – memory location of the Ith and Jth element of the array (To be calculated)
- ARRAY – name of the ARRAY
- WORDLENGTH – number of bytes required to store one element depending upon its data type like a character values needs 1 byte and an integer value needs 2 bytes.
- M – Number of rows of the array.
- BASEADDRESS -Address of first element of the array

- In this example to calculate memory address of a given element (44) says in 1st row and 3rd column you will put the values in the formula.

  I=1, J=3

  M=3

  WORDLENGTH=1 (assuming only one byte is required to store these small ints)

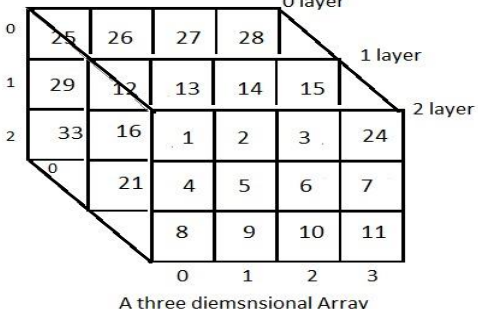
  BASEADDRESS = 1001

  ADDRESS(ARRAY(6))= 1001+ 1*(3*(3-1) + (1-1))

  =1007 this is the address of memory location where 44 is stored as visible in the previous figure

3. **Three-dimensional Array**

- In types of arrays, a three-dimensional array is an extension to the two dimensional array with addition of depth.
- It can be seen as a cube that has rows, columns and depth as third dimension.
- To access any element in a three-dimensional array three subscripts are required for position of element in a specific row, column and depth.
- The first index is for depth (dimension or layer), second is for row index and third is for column.
- In the example shown the index values (2,0,3) is used to access element 24.



A three diemsnsional Array

- To find the address of any element in 3-Dimensional arrays there are the following two ways-

- ✓ **Row Major Order**
- ✓ **Column Major Order**

Row Major Order:

- To find the address of the element using row-major order, use the following formula:

Address of[i][j][k] = B + W * {[(I – LR) * N + (J – LC)]* R + [K – LB]}

Where,

- I = Row Subset of an element whose address to be found,
- J = Column Subset of an element whose address to be found,
- K = Block Subset of an element whose address to be found,
- B = Base address,
- W = Storage size of one element store in any array(in byte),
- LR = Lower Limit of row/start row index of matrix(If not given assume it as zero),
- LC = Lower Limit of column/start column index of matrix(If not given assume it as zero),
- LB = Lower Limit of blocks in matrix,
- N = Number of column given in the matrix
- R = Number of blocks given in the matrix.

Column Major Order:

- To find the address of the element using column-major order, use the following formula-

Address of[i][j][k] = B + W * {[(I – LR) + (J – LC) * M]* R + [K – LB]}

Where

- I = Row Subset of an element whose address to be found,
- J = Column Subset of an element whose address to be found,
- K = Block Subset of an element whose address to be found
- B = Base address,
- W = Storage size of one element store in any array(in byte),
- LR = Lower Limit of row/start row index of matrix(If not given assume it as zero),
- LC = Lower Limit of column/start column index of matrix(If not given assume it as zero),
- LB = Lower Limit of blocks in matrix,
- M = Number of rows given in the matrix,
- R = Number of blocks given in the matrix.

SPARSE MATRIX REPRESENTATION

- A matrix is a two-dimensional data object made of m rows and n columns, therefore having total m x n values.
- If most of the elements of the matrix have 0 value, then it is called a sparse matrix.

Why to use Sparse Matrix instead of simple matrix?

- **Storage:** There are lesser non-zero elements than zeros and thus lesser memory can be used to store only those elements.
- **Computing time:** Computing time can be saved by logically designing a data structure traversing only non-zero elements..

  Example:

  0 0 3 0 4
  0 0 5 7 0
  0 0 0 0 0
  0 2 6 0 0
- Representing a sparse matrix by a 2D array leads to wastage of lots of memory as zeroes in the matrix are of no use in most of the cases.
- So, instead of storing zeroes with non-zero elements, we only store non-zero elements.
- This means storing non-zero elements with triples- (Row, Column, value).

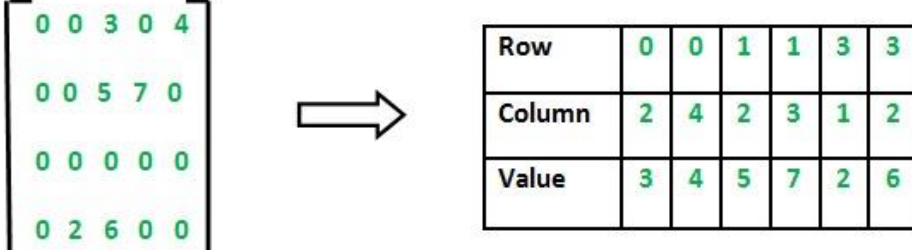
- Sparse Matrix Representations can be done in many ways following are two common representations:

✓ **Array representation**
✓ **Linked list representation**

Using Arrays:

- 2D array is used to represent a sparse matrix in which there are three rows named as

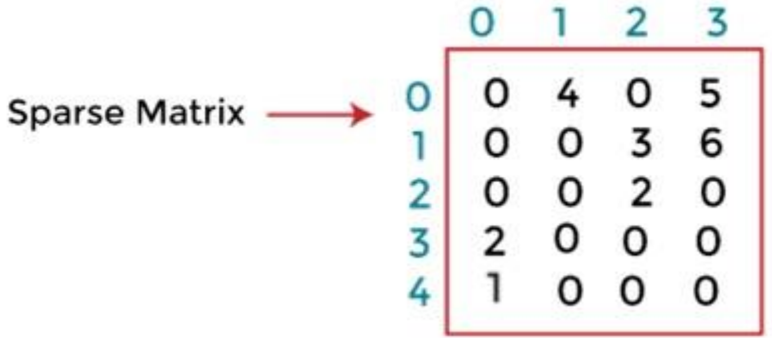
✓ **Row**: Index of row, where non-zero element is located
✓ **Column**: Index of column, where non-zero element is located
✓ **Value**: Value of the non zero element located at index – (row, column)



Example -

- Let's understand the array representation of sparse matrix with the help of the example given below -

- Consider the sparse matrix -



- In the above figure, we can observe a 5x4 sparse matrix containing 7 non-zero elements and 13 zero elements.
- The above matrix occupies 5x4 = 20 memory space.
- Increasing the size of matrix will increase the wastage space.
- The tabular representation of the above matrix is given below -

Table Structure

Row	Column	Value
0	1	4
0	3	5
1	2	3
1	3	6
2	2	2
3	0	2
4	0	1
5	4	7

- In the above structure, first column represents the rows, the second column represents the columns, and the third column represents the non-zero value.
- The first row of the table represents the triplets.
- The first triplet represents that the value 4 is stored at 0th row and 1st column.
- Similarly, the second triplet represents that the value 5 is stored at the 0th row and 3rd column.
- In a similar manner, all triplets represent the stored location of the non-zero elements in the matrix.
- The size of the table depends upon the total number of non-zero elements in the given sparse matrix.
- Above table occupies 8x3 = 24 memory space which is more than the space occupied by the sparse matrix.
- Consider the case if the matrix is 8*8 and there are only 8 non-zero elements in the matrix, then the space occupied by the sparse matrix would be 8*8 = 64, whereas the space occupied by the table represented using triplets would be 8*3 = 24.

POLYNOMIAL REPRESENTATION

- Polynomials and Sparse Matrix are two important applications of arrays and linked lists.
- A polynomial is composed of different terms where each of them holds a coefficient and an exponent.

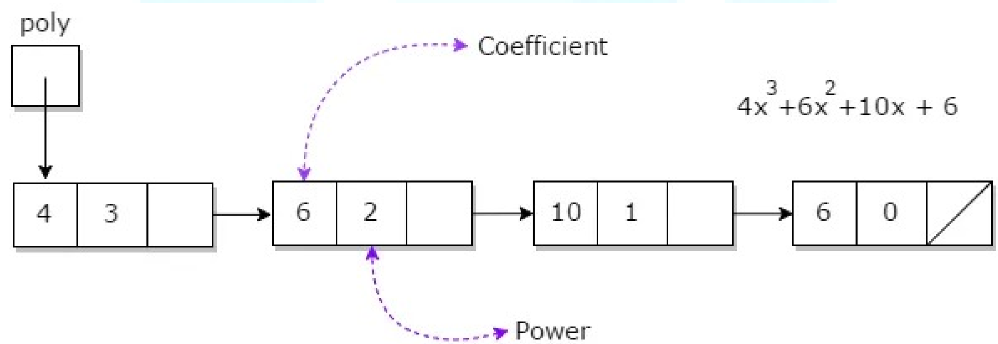
POLYNOMIAL

- A polynomial p(x) is the expression in variable x which is in the form ($ax^n + bx^{n-1} + \ldots + jx + k$), where a, b, c …., k fall in the category of real numbers and 'n' is non negative integer, which is called the degree of polynomial.
- An essential characteristic of the polynomial is that each term in the polynomial expression consists of two parts:

  ✓ **One is the coefficient**
  ✓ **Other is the exponent**
- $10x^2 + 26x$, here 10 and 26 are coefficients and 2, 1 is its exponential value.

Points to keep in Mind while working with Polynomials:

- The sign of each coefficient and exponent is stored within the coefficient and the exponent itself
- Additional terms having equal exponent is possible one
- The storage allocation for each term in the polynomial must be done in ascending and descending order of their exponent



REPRESENTATION OF POLYNOMIAL

- Polynomial can be represented in the various ways. These are:

- **By the use of arrays**
- **By the use of Linked List**
- There may arise some situation where you need to evaluate many polynomial expressions and perform basic arithmetic operations like addition and subtraction with those numbers.
- For this, you will have to get a way to represent those polynomials.
- The simple way is to represent a polynomial with degree 'n' and store the coefficient of n+1 terms of the polynomial in the array.
- So every array element will consist of two values:

- ✓ **Coefficient**
- ✓ **Exponent**

- A polynomial can be represented using the C++ code

```
#include <iostream>
#include <iomanip.h>
using namespace std;

struct poly {
   int coeff;
   int pow_val;
   poly* next;
};

class add {
   poly *poly1, *poly2, *poly3;

public:
   add() { poly1 = poly2 = poly3 = NULL; }
   void addpoly();
   void display();
};

void add::addpoly()
{
   int i, p;
   poly *newl = NULL, *end = NULL;
   cout << "Enter highest power for x\n"; cin >> p;
   //Read first poly
   cout << "\nFirst Polynomial\n"; for (i = p; i >= 0; i--) {
      newl = new poly;
      newl->pow_val = p;
      cout << "Enter Co-efficient for degree" << i << ":: "; cin >> newl->coeff;
      newl->next = NULL;
      if (poly1 == NULL)
         poly1 = newl;
      else
         end->next = newl;
      end = newl;
   }

   //Read Second poly
   cout << "\n\nSecond Polynomial\n"; end = NULL; for (i = p; i >= 0; i--) {
      newl = new poly;
      newl->pow_val = p;
      cout << "Enter Co-efficient for degree" << i << ":: "; cin >> newl->coeff;
      newl->next = NULL;
      if (poly2 == NULL)
         poly2 = newl;
```

```
        else
            end->next = newl;
        end = newl;
    }

    //Addition Logic
    poly *p1 = poly1, *p2 = poly2;
    end = NULL;
    while (p1 != NULL && p2 != NULL) {
        if (p1->pow_val == p2->pow_val) {
            newl = new poly;
            newl->pow_val = p--;
            newl->coeff = p1->coeff + p2->coeff;
            newl->next = NULL;
            if (poly3 == NULL)
                poly3 = newl;
            else
                end->next = newl;
            end = newl;
        }
        p1 = p1->next;
        p2 = p2->next;
    }
}

void add::display()
{
    poly* t = poly3;
    cout << "\n\nAnswer after addition is : ";
    while (t != NULL) {
        cout.setf(ios::showpos);
        cout << t->coeff;
        cout.unsetf(ios::showpos);
        cout << "X" << t->pow_val;
        t = t->next;
    }
}
int main()
{
    add obj;
    obj.addpoly();
    obj.display();
}
```

OUTPUT:

```
Q:\example\example.exe
Enter highest power for x
2

First Polynomial
Enter Co-efficient for degree2::  1
Enter Co-efficient for degree1::  2
Enter Co-efficient for degree0::  3

Second Polynomial
Enter Co-efficient for degree2::  3
Enter Co-efficient for degree1::  2
Enter Co-efficient for degree0::  1

Answer after addition is : +4X2+4X1+4X0_
```

LINKED LIST

- A linked list is a sequence of data structures, which are connected together via links.
- Linked List is a sequence of links which contains items.
- Each link contains a connection to another link.
- Linked list is the second most-used data structure after array.
- **Following are the important terms to understand the concept of Linked List.**

✓ **Link** − Each link of a linked list can store a data called an element.

✓ **Next** − Each link of a linked list contains a link to the next link called Next.

✓ **LinkedList** − A Linked List contains the connection link to the first link called First.

Linked List Representation

- Linked list can be visualized as a chain of nodes, where every node points to the next node.

- As per the above illustration, following are the important points to be considered.

✓ Linked List contains a link element called first.
✓ Each link carries a data field(s) and a link field called next.
✓ Each link is linked with its next link using its next link.
✓ Last link carries a link as null to mark the end of the list.

Types of Linked List

- Following are the various types of linked list.

✔ **Simple Linked List** − Item navigation is forward only.

✔ **Doubly Linked List** − Items can be navigated forward and backward.

✔ **Circular Linked List** − Last item contains link of the first element as next and the first element has a link to the last element as previous.

Basic Operations

- Following are the basic operations supported by a list.

✔ **Insertion** − Adds an element at the beginning of the list.

✔ **Deletion** − Deletes an element at the beginning of the list.

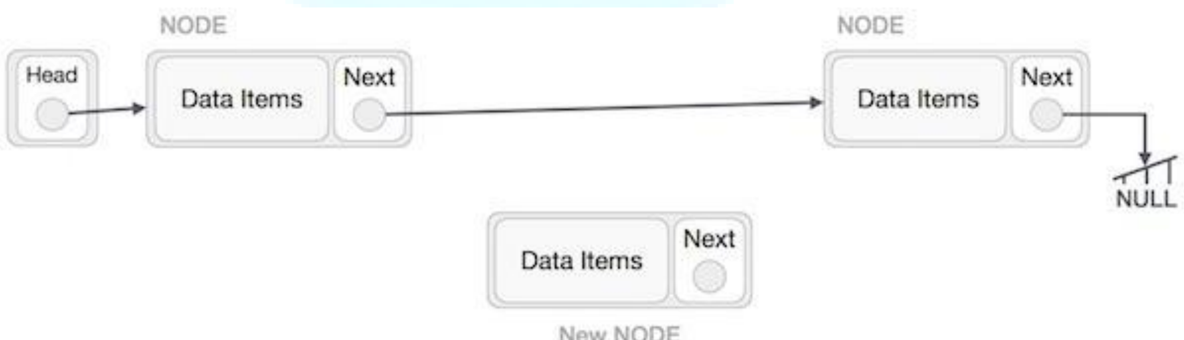
✔ **Display** − Displays the complete list.

✔ **Search** − Searches an element using the given key.

✔ **Delete** − Deletes an element using the given key.

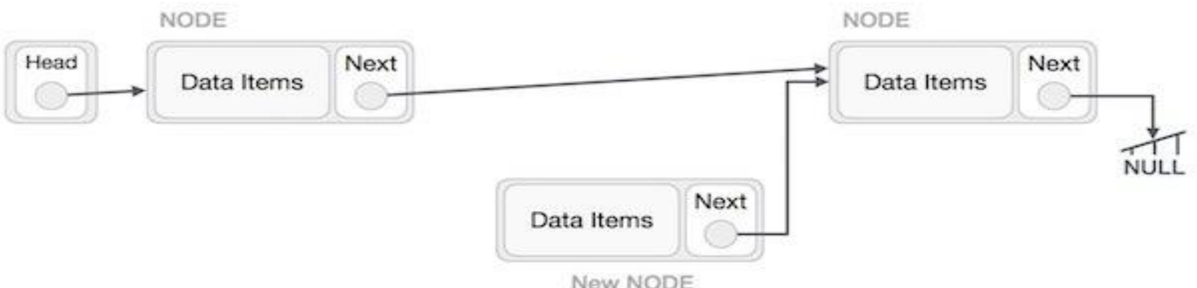
Insertion Operation

- Adding a new node in linked list is a more than one step activity.
- We shall learn this with diagrams here.
- First, create a node using the same structure and find the location where it has to be inserted.



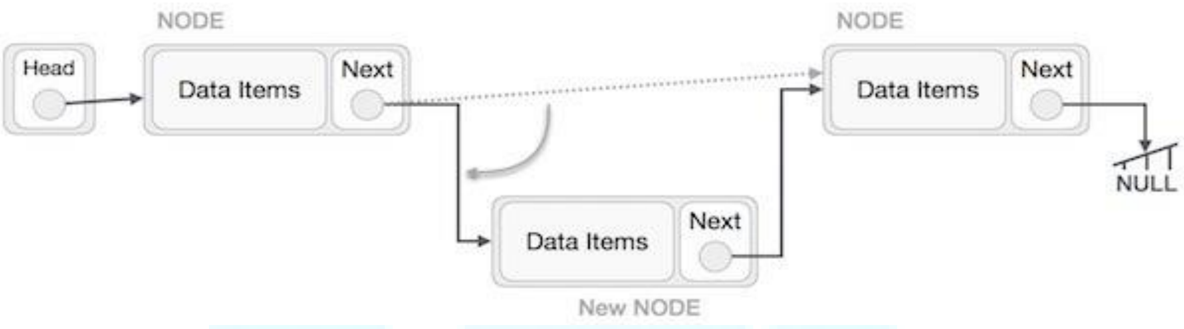
- Imagine that we are inserting a node B (NewNode), between A (LeftNode) and C (RightNode).
- Then point B.next to C −

NewNode.next −> RightNode;

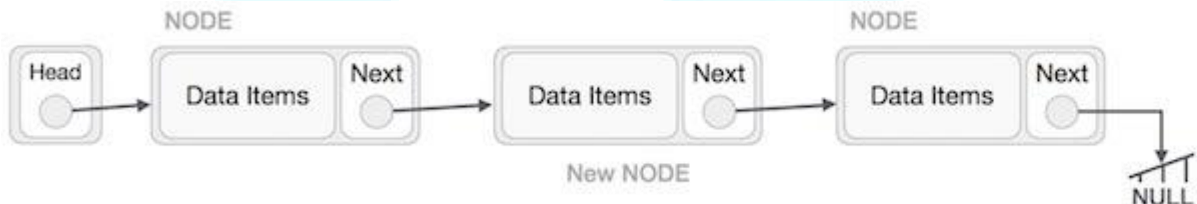


- Now, the next node at the left should point to the new node.

  LeftNode.next −> NewNode;



- This will put the new node in the middle of the two. The new list should look like this −



- Similar steps should be taken if the node is being inserted at the beginning of the list.
- While inserting it at the end, the second last node of the list should point to the new node and the new node will point to NULL.

Deletion Operation

- Deletion is also a more than one step process.
- First, locate the target node to be removed, by using searching algorithms.

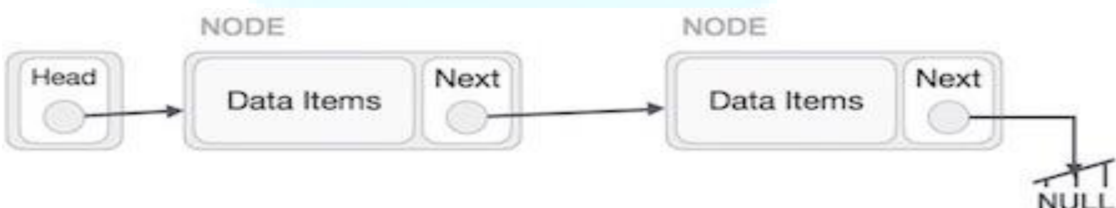
- The left (previous) node of the target node now should point to the next node of the target node –

LeftNode.next −> TargetNode.next;

- This will remove the link that was pointing to the target node.
- Now, using the following code, we will remove what the target node is pointing at.

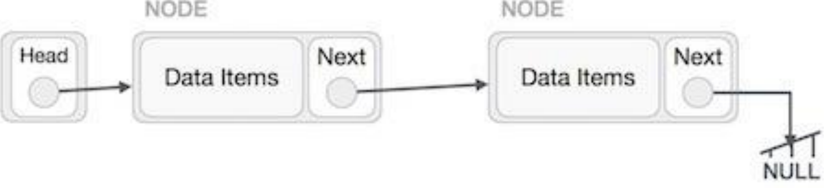
TargetNode.next −> NULL;

- We need to use the deleted node.
- We can keep that in memory otherwise we can simply deallocate memory and wipe off the target node completely.

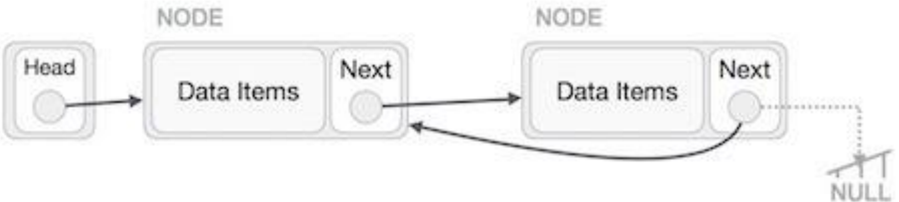


Reverse Operation

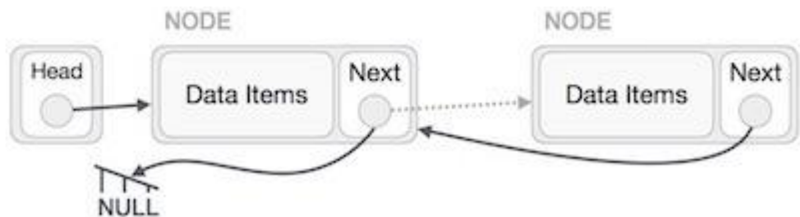
- This operation is a thorough one.
- We need to make the last node to be pointed by the head node and reverse the whole linked list.



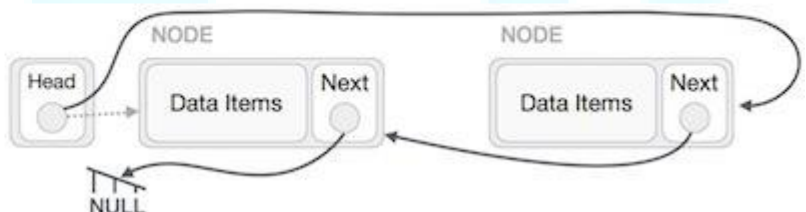
- First, we traverse to the end of the list. It should be pointing to NULL.
- Now, we shall make it point to its previous node −



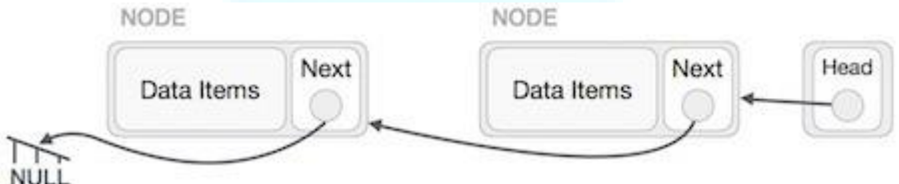
- We have to make sure that the last node is not the last node.
- So we'll have some temp node, which looks like the head node pointing to the last node.
- Now, we shall make all left side nodes point to their previous nodes one by one.



- Except the node (first node) pointed by the head node, all nodes should point to their predecessor, making them their new successor.
- The first node will point to NULL.



- We'll make the head node point to the new first node by using the temp node.



DOUBLY LINKED LIST

- Doubly Linked List is a variation of Linked list in which navigation is possible in both ways, either forward and backward easily as compared to Single Linked List.
- Following are the important terms to understand the concept of doubly linked list.

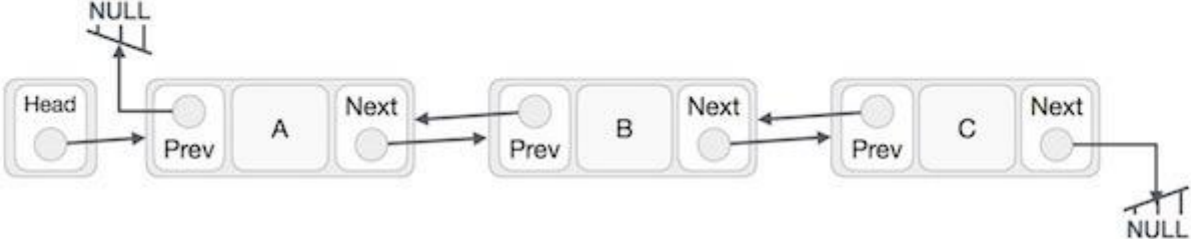
✓ **Link** − Each link of a linked list can store a data called an element.

✓ **Next** − Each link of a linked list contains a link to the next link called Next.

- ✓ **Prev** − Each link of a linked list contains a link to the previous link called Prev.

- ✓ **LinkedList** − A Linked List contains the connection link to the first link called First and to the last link called Last.

Doubly Linked List Representation



As per the above illustration, following are the important points to be considered.

- Doubly Linked List contains a link element called first and last.
- Each link carries a data field(s) and two link fields called next and prev.
- Each link is linked with its next link using its next link.
- Each link is linked with its previous link using its previous link.
- The last link carries a link as null to mark the end of the list.

Basic Operations

- Following are the basic operations supported by a list.

- ✓ **Insertion** − Adds an element at the beginning of the list.
- ✓ **Deletion** − Deletes an element at the beginning of the list.
- ✓ **Insert Last** − Adds an element at the end of the list.
- ✓ **Delete Last** − Deletes an element from the end of the list.
- ✓ **Insert After** − Adds an element after an item of the list.
- ✓ **Delete** − Deletes an element from the list using the key.
- ✓ **Display forward** − Displays the complete list in a forward manner.
- ✓ **Display backward** − Displays the complete list in a backward manner.

Insertion Operation

- Following code demonstrates the insertion operation at the beginning of a doubly linked list.

Example

```
//insert link at the first location
void insertFirst(int key, int data) {

  //create a link
  struct node link = (struct node) malloc(sizeof(struct node));
  link->key = key;
  link->data = data;

  if(isEmpty()) {
    //make it the last link
```

```
    last = link;
  } else {
    //update first prev link
    head->prev = link;
  }

  //point it to old first link
  link->next = head;

  //point first to new first link
  head = link;
}
```

Deletion Operation

- Following code demonstrates the deletion operation at the beginning of a doubly linked list.

Example

```
//delete first item
struct node* deleteFirst() {

  //save reference to first link
  struct node *tempLink = head;

  //if only one link
  if(head->next == NULL) {
    last = NULL;
  } else {
    head->next->prev = NULL;
  }

  head = head->next;

  //return the deleted link
  return tempLink;
}
```

Insertion at the End of an Operation

- Following code demonstrates the insertion operation at the last position of a doubly linked list.

Example

```
//insert link at the last location
void insertLast(int key, int data) {

  //create a link
  struct node link = (struct node) malloc(sizeof(struct node));
  link->key = key;
  link->data = data;
```

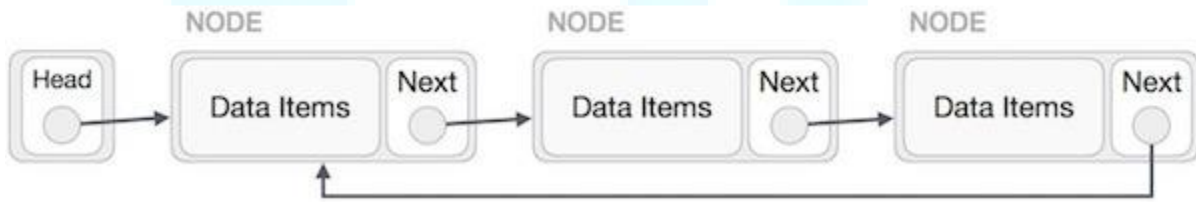
```
  if(isEmpty()) {
    //make it the last link
    last = link;
  } else {
    //make link a new last link
    last->next = link;

    //mark old last node as prev of new link
    link->prev = last;
  }

  //point last to new last node
  last = link;
}
```

CIRCULAR LINKED LIST

- Circular Linked List is a variation of Linked list in which the first element points to the last element and the last element points to the first element.
- Both Singly Linked List and Doubly Linked List can be made into a circular linked list.

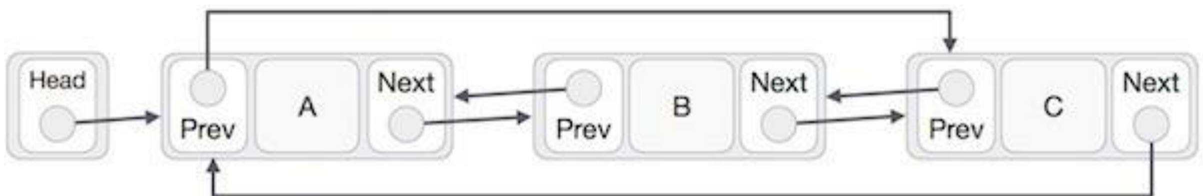
Singly Linked List as Circular

- In singly linked list, the next pointer of the last node points to the first node.



Doubly Linked List as Circular

- In doubly linked list, the next pointer of the last node points to the first node and the previous pointer of the first node points to the last node making the circular in both directions.



- As per the above illustration, following are the important points to be considered.
- The last link's next points to the first link of the list in both cases of singly as well as doubly linked list.
- The first link's previous points to the last of the list in case of doubly linked list.

Basic Operations

- Following are the important operations supported by a circular list.

✔ **insert** − Inserts an element at the start of the list.

✔ **delete** − Deletes an element from the start of the list.

✔ **display** − Displays the list.

Insertion Operation

- Following code demonstrates the insertion operation in a circular linked list based on single linked list.

Example

```
insertFirst(data):
Begin
  create a new node
  node -> data := data
  if the list is empty, then
    head := node
    next of node = head
  else
    temp := head
    while next of temp is not head, do
    temp := next of temp
    done
    next of node := head
    next of temp := node
    head := node
  end if
End
```

Deletion Operation

- Following code demonstrates the deletion operation in a circular linked list based on single linked list.

```
deleteFirst():
Begin
  if head is null, then
    it is Underflow and return
  else if next of head = head, then
    head := null
    deallocate head
  else
    ptr := head
    while next of ptr is not head, do
      ptr := next of ptr
    next of ptr = next of head
    deallocate head
    head := next of ptr
```

```
  end if
End
```

Display List Operation

- Following code demonstrates the display list operation in a circular linked list.

```
display():
Begin
  if head is null, then
    Nothing to print and return
  else
    ptr := head
    while next of ptr is not head, do
      display data of ptr
      ptr := next of ptr
    display data of ptr
  end if
End
```

Linked List representation of the sparse matrix

- In a linked list representation, the linked list data structure is used to represent the sparse matrix.
- The advantage of using a linked list to represent the sparse matrix is that the complexity of inserting or deleting a node in a linked list is lesser than the array.
- Unlike the array representation, a node in the linked list representation consists of four fields.
- The four fields of the linked list are given as follows -
    - **Row** - It represents the index of the row where the non-zero element is located.
    - **Column** - It represents the index of the column where the non-zero element is located.
    - **Value** - It is the value of the non-zero element that is located at the index (row, column).
    - **Next node** - It stores the address of the next node.
- The node structure of the linked list representation of the sparse matrix is shown in the below image -

EXAMPLE

- Let's understand the linked list representation of sparse matrix with the help of the example given below -

- Consider the sparse matrix -

Sparse Matrix →

	0	1	2	3
0	0	0	1	0
1	3	0	0	0
2	0	4	5	0
3	0	6	0	0

- In the above figure, we can observe a 4x4 sparse matrix containing 5 non-zero elements and 11 zero elements.
- Above matrix occupies 4x4 = 16 memory space. Increasing the size of matrix will increase the wastage space.
- The linked list representation of the above matrix is given below -

- In the above figure, the sparse matrix is represented in the linked list form.
- In the node, the first field represents the index of the row, the second field represents the index of the column, the third field represents the value, and the fourth field contains the address of the next node.
- In the above figure, the first field of the first node of the linked list contains 0, which means 0th row, the second field contains 2, which means 2nd column, and the third field contains 1 that is the non-zero element.
- So, the first node represents that element 1 is stored at the 0th row-2nd column in the given sparse matrix.
- In a similar manner, all of the nodes represent the non-zero elements of the sparse matrix.

REPRESENTATION OF POLYNOMIAL LINKED LIST

- A polynomial can be thought of as an ordered list of non zero terms. Each non zero term is a two-tuple which holds two pieces of information:
    - **The exponent part**
    - **The coefficient part**

```
#include <iostream>
using namespace std;

class polyll {
private:
   struct polynode {
```

ENTRI

```
        float coeff;
        int exp;
        polynode* link;
    } * p;

public:
    polyll();
    void poly_append(float c, int e);
    void display_poly();
    void poly_add(polyll& l1, polyll& l2);
    ~polyll();
};
polyll::polyll()
{
    p = NULL;
}
void polyll::poly_append(float c, int e)
{
    polynode* temp = p;
    if (temp == NULL) {
        temp = new polynode;
        p = temp;
    }
    else {
        while (temp->link != NULL)
            temp = temp->link;
        temp->link = new polynode;
        temp = temp->link;
    }
    temp->coeff = c;
    temp->exp = e;
    temp->link = NULL;
}
void polyll::display_poly()
{
    polynode* temp = p;
    int f = 0;

    cout << endl; while (temp != NULL) { if (f != 0) { if (temp->coeff > 0)
                cout << " + ";
            else
                cout << " "; } if (temp->exp != 0)
            cout << temp->coeff << "x^" << temp->exp;
        else
            cout << temp->coeff;
        temp = temp->link;
        f = 1;
    }
}
void polyll::poly_add(polyll& l1, polyll& l2)
```

```
{
   polynode* z;
   if (l1.p == NULL && l2.p == NULL)
      return;
   polynode *temp1, *temp2;
   temp1 = l1.p;
   temp2 = l2.p;
   while (temp1 != NULL && temp2 != NULL) {
      if (p == NULL) {
         p = new polynode;
         z = p;
      }
      else {
         z->link = new polynode;
         z = z->link;
      }
      if (temp1->exp < temp2->exp) {
         z->coeff = temp2->coeff;
         z->exp = temp2->exp;
         temp2 = temp2->link;
      }
      else {
         if (temp1->exp > temp2->exp) {
            z->coeff = temp1->coeff;
            z->exp = temp1->exp;
            temp1 = temp1->link;
         }
         else {
            if (temp1->exp == temp2->exp) {
               z->coeff = temp1->coeff + temp2->coeff;
               z->exp = temp1->exp;
               temp1 = temp1->link;
               temp2 = temp2->link;
            }
         }
      }
   }
   while (temp1 != NULL) {
      if (p == NULL) {
         p = new polynode;
         z = p;
      }
      else {
         z->link = new polynode;
         z = z->link;
      }
      z->coeff = temp1->coeff;
      z->exp = temp1->exp;
      temp1 = temp1->link;
   }
```

```
    while (temp2 != NULL) {
        if (p == NULL) {
            p = new polynode;
            z = p;
        }
        else {
            z->link = new polynode;
            z = z->link;
        }
        z->coeff = temp2->coeff;
        z->exp = temp2->exp;
        temp2 = temp2->link;
    }
    z->link = NULL;
}
polyll::~polyll()
{
    polynode* q;
    while (p != NULL) {
        q = p->link;
        delete p;
        p = q;
    }
}
int main()
{
    polyll p1;
    p1.poly_append(1.4, 5);
    p1.poly_append(1.5, 4);
    p1.poly_append(1.7, 2);
    p1.poly_append(1.8, 1);
    p1.poly_append(1.9, 0);
    cout << "\nFirst polynomial:";
    p1.display_poly();
    polyll p2;
    p2.poly_append(1.5, 6);
    p2.poly_append(2.5, 5);
    p2.poly_append(-3.5, 4);
    p2.poly_append(4.5, 3);
    p2.poly_append(6.5, 1);
    cout << "\nSecond polynomial:";
    p2.display_poly();
    polyll p3;
    p3.poly_add(p1, p2);
    cout << "\nResultant polynomial: ";
    p3.display_poly();
    getch();
}
```

Output:

Q:\example\example.exe

```
First polynomial:
1.4x^5 + 1.5x^4 + 1.7x^2 + 1.8x^1 + 1.9
Second polynomial:
1.5x^6 + 2.5x^5 -3.5x^4 + 4.5x^3 + 6.5x^1
Resultant polynomial:
1.5x^6 + 3.9x^5 -2x^4 + 4.5x^3 + 1.7x^2 + 8.3x^1 + 1.9
```