Arrays | Linked Lists

Lecture Note

Advanced Data Structures & Algorithms

Data Structures

Defining Data Structures

1. Data structures are systems for organizing and storing data.
2. Goal: Efficient access, manipulation, and processing of information.

Importance of Data Structures

1. Essential for structured data management.
2. Enable effective representation of information.

Common Datastructures

1. Arrays
   1. Ordered collections, fast element access.
   2. Limited flexibility for resizing.
2. Linked Lists
   1. Dynamic, flexible insertion/deletion.
   2. Consist of nodes with data and references.
3. Stacks
   1. LIFO (Last-In-First-Out) principle.
   2. Used for managing data with a focus on the last added item.

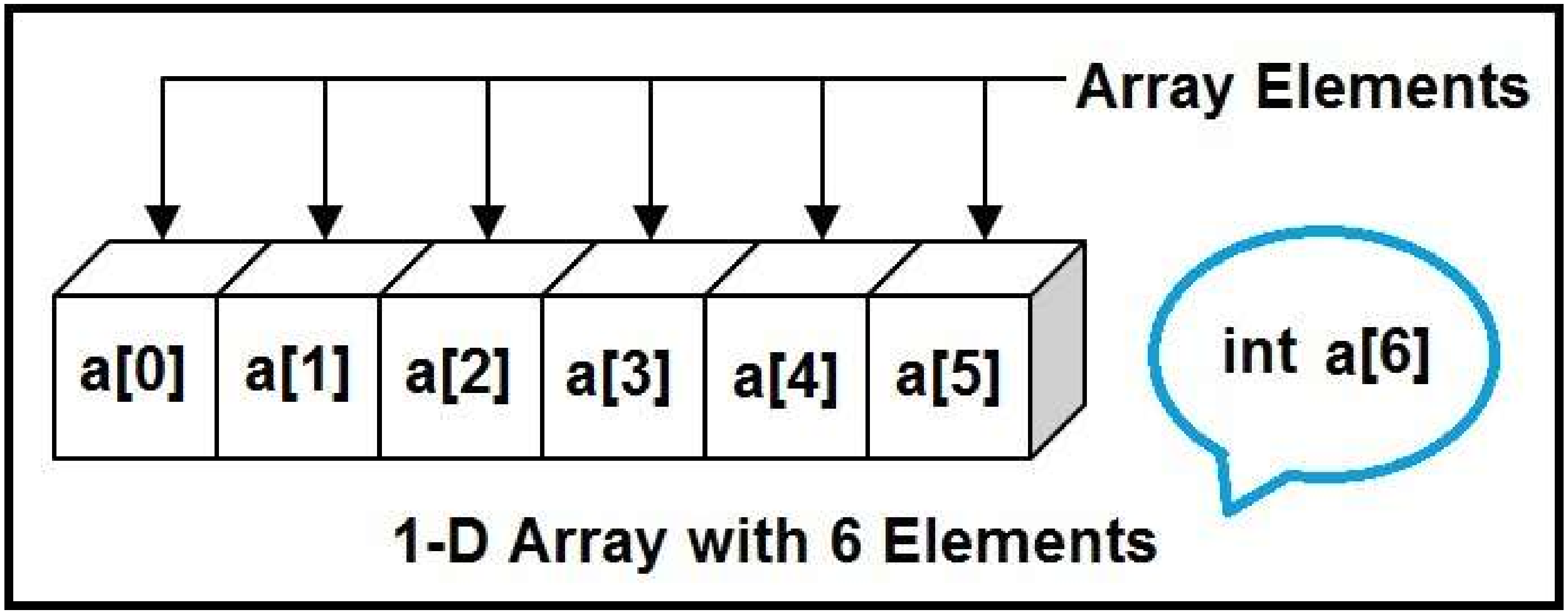
Common Datastructures

4. Queues
    1. FIFO (First-In-First-Out) principle.
    2. Used for managing data like a real-world queue.
5. Trees
    1. Hierarchical structures with root and child nodes.
    2. Binary trees, AVL trees, B-trees, etc.
6. Graphs
    1. Nodes and edges for complex relationships.
    2. Important for network analysis, routing, and more.

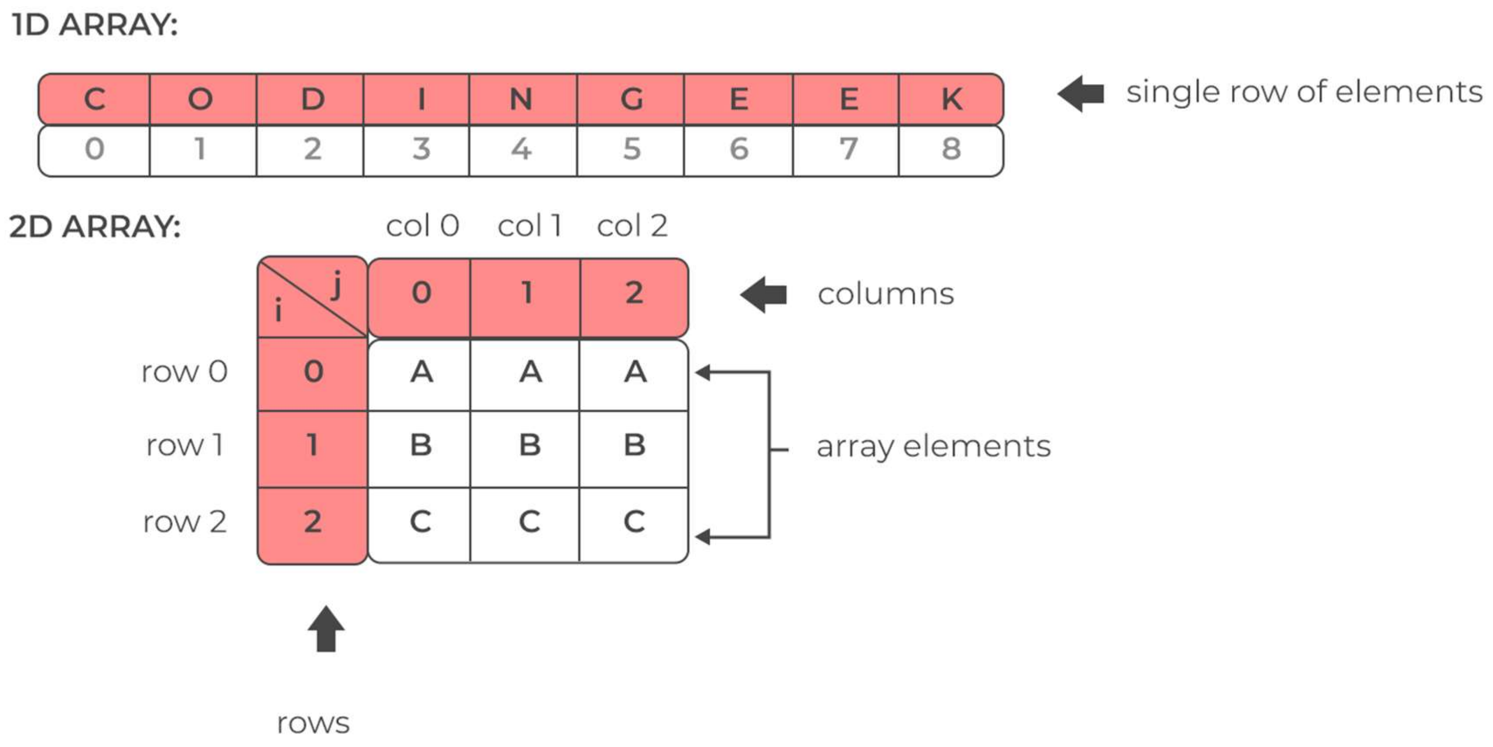
Common Datastructures

7. Hash Tables
    1. Keys mapped to values using hash functions.
    2. Fast data retrieval, indexing, and searching.
8. Heaps
    1. Priority queue implementations.
    2. Efficiently retrieve highest/lowest priority elements.
9. Sets and Maps
    1. Sets for unique elements, maps for key-value pairs.
    2. Efficient data retrieval by key.
10. Trie
    1. Stores sets of strings or associative arrays.
    2. Designed for efficient string-based operations.

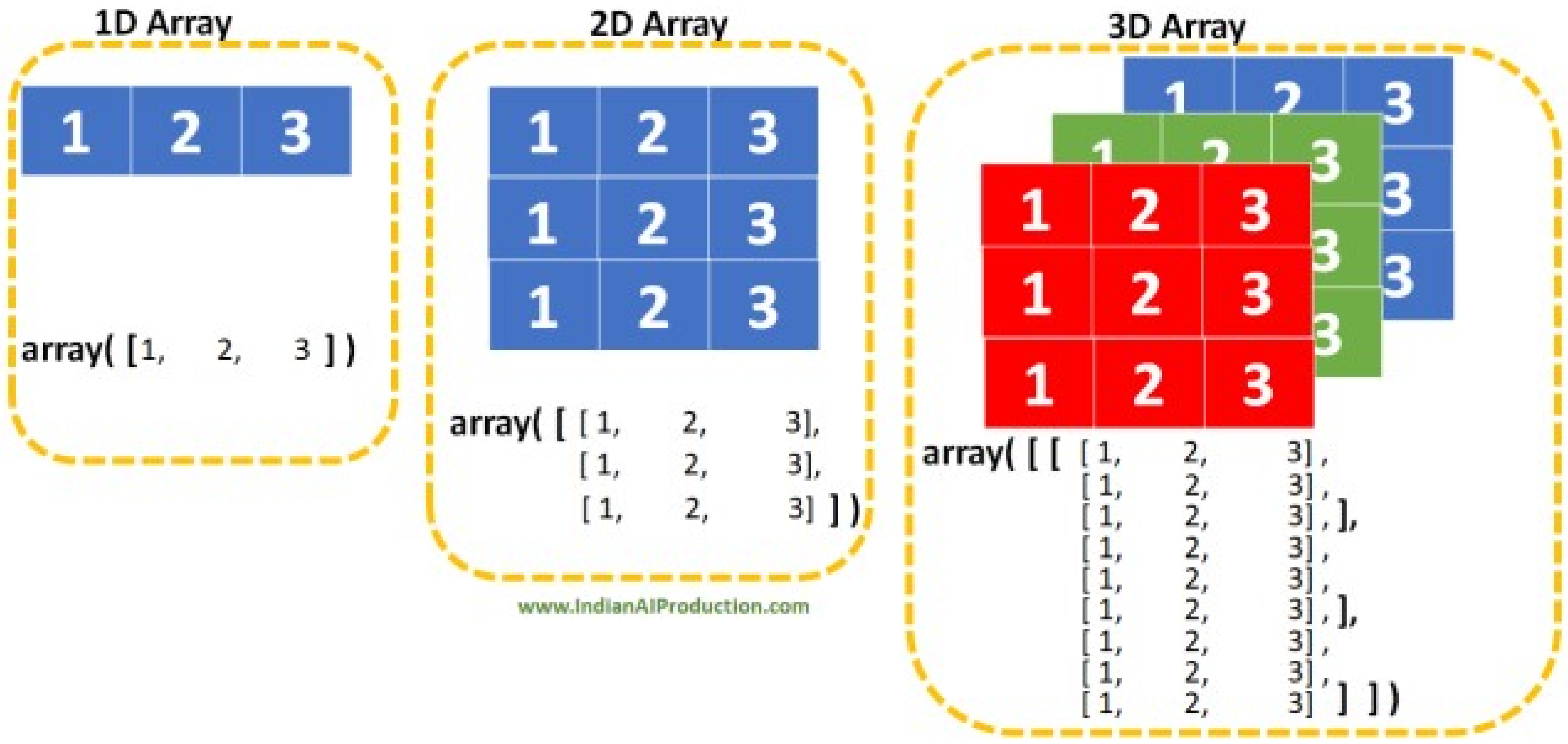
Arrays


Array Elements
a[0]
a[1]
a[2]
a[3]
a[4]
a[5]
int a[6]
1-D Array with 6 Elements

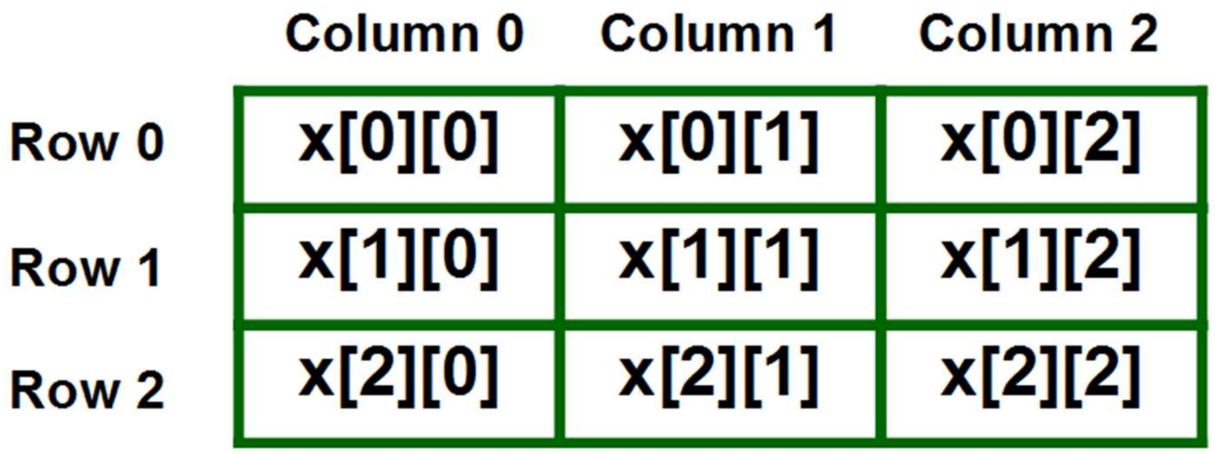
Arrays


1D ARRAY:
C O D I N G E E K
0 1 2 3 4 5 6 7 8
single row of elements
2D ARRAY:
col 0 col 1 col 2
i j
0 1 2
columns
row 0 0 A A A
row 1 1 B B B
row 2 2 C C C
array elements
rows

Multidimensional Arrays



Multidimensional Arrays



	Column 0	Column 1	Column 2
Row 0	x[0][0]	x[0][1]	x[0][2]
Row 1	x[1][0]	x[1][1]	x[1][2]
Row 2	x[2][0]	x[2][1]	x[2][2]

Representations of Arrays

- Row-Major Order
- Column-Major Order

Calculation of address of element of 1-D array

- To find the address of an element in an 1D array the following formula is used -

address of A[i] *= B + W * (i – LB)*

where,
A[lb,..., ub]

i *= index whose address to be found,*
B *= base address of the array,*
W *= storage size of one element store in any array(in byte),*
LB*= Lower Limit/Lower Bound of subscript(If not specified assume zero).*

Example

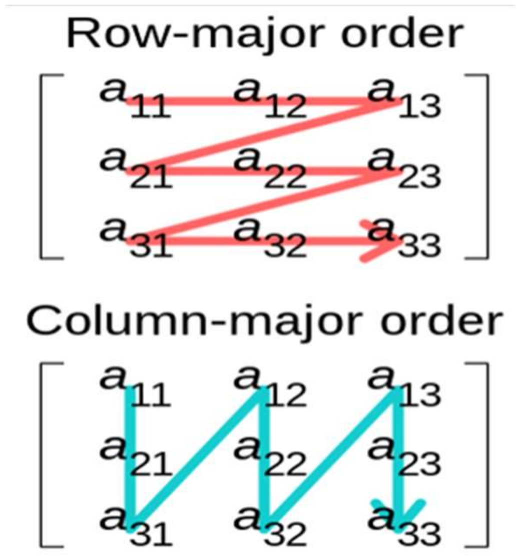
- Given the base address of an array **A[1300 .... 1900]** as **1020** and the size of each element is 2 bytes in the memory, find the address of **A[1700].**

*address of A[i] = B + W * (i – LB)*

```
address of A[1700] = 1020 + 2 * (1700 – 1300)
                   = 1020 + 2 * (400)
                   = 1020 + 800
address of A[1700] = 1820
```

Calculation of address of element of 2-D using row-major and column-major order

- To find the address of any element in a **2-Dimensional** array there are the following two ways-
  1. **Row Major Order**
  2. **Column Major Order**


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

Calculation of address of element of 2-D using row-major and column-major order

Row-Major Order

$$\text{address of } A[i][j] = B + W * ((i - LR) * N + (j - LC))$$

where,

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

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.

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

Calculation of address of element of 2-D using row-major and column-major order

Column-Major Order

$$address\ of\ A[i][j] = B + W * ((j - LC) * M + (i - LR))$$

where,

I = Row Subset of an element whose address to be found,
J = Column 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),
M = Number of rows given in the matrix.

Example

- Given an array **arr[1………10][1………15]** with a base value of **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.

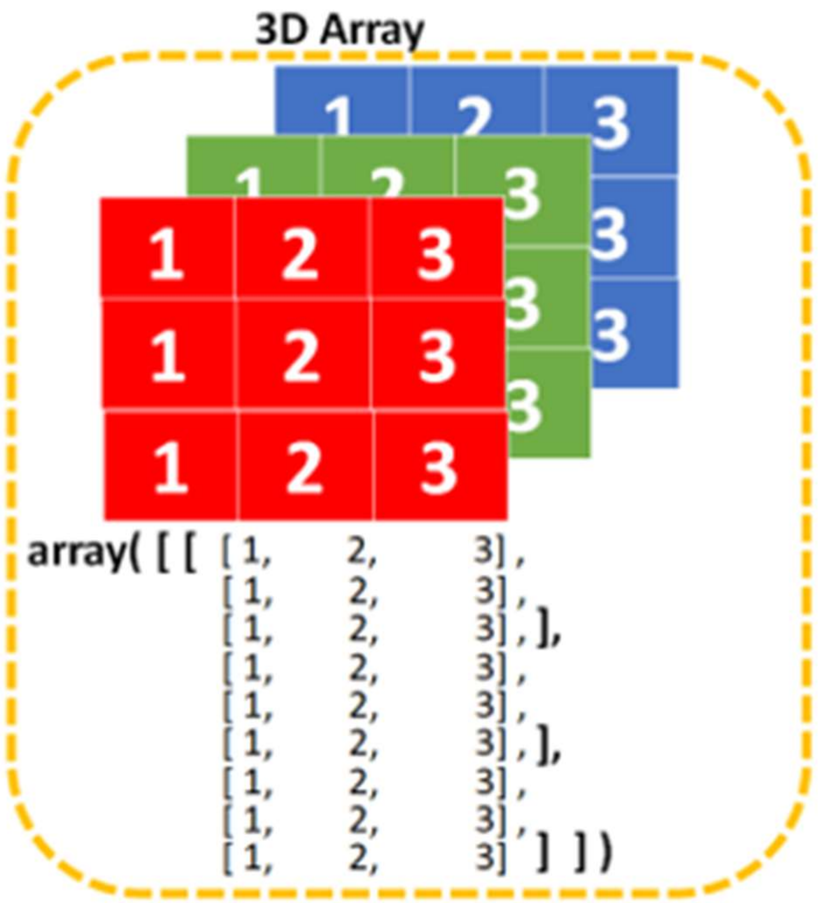
address of A[I][J] = B + W * ((J – LC) * M + (I – LR))

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

Calculation of address of element of 3-D using row-major and column-major order

- **3-Dimensional** array is a collection of 2-Dimensional arrays. It is specified by using three subscripts:
  - Block size
  - Row size
  - Column size

A[row][col][block]



```
array( [ [ [1,   2,   3] ,
           [1,   2,   3] ,
           [1,   2,   3] , ],
           [1,   2,   3] ,
           [1,   2,   3] ,
           [1,   2,   3] , ],
           [1,   2,   3] ,
           [1,   2,   3] ,
           [1,   2,   3]  ] ] )
```

Calculation of address of element of 3-D using row-major and column-major order

Row-Major Order

address of A[i][j][k] = B + W *(M * N(i-x) + N *(j-y) + (k-z))

Where,

B = Base Address (start address)
W = Weight (storage size of one element stored in the array)
M = Row (total number of rows)
N = Column (total number of columns)
P = Width (total number of cells depth-wise)
x = Lower Bound of Row
y = Lower Bound of Column
z = Lower Bound of Width

Example

- Given an array, **arr[1:9, -4:1, 5:10]** with a base value of **400** and the size of each element is **2 Bytes** in memory find the address of element **arr[5][-1][8]** with the help of row-major order?

Example

- **Given:**
  I = 5, J = -1, K = 8
  Base address B = 400
  Storage size of one element store in any array(in Byte) W = 2
  Lower Limit of row/start row index of matrix x = 1
  Lower Limit of column/start column index of matrix y = -4
  Lower Limit of blocks in matrix z = 5
  M(row) = Upper Bound – Lower Bound + 1 = 9 – 1 + 1 = 9
  N(Column)= Upper Bound – Lower Bound + 1 = 1 – (-4) + 1 = 6
- **Formula used:**
  Address of[I][J][K] =B + W (M * N(i-x) + N *(j-y) + (k-z))
- **Solution:**
  Address of arr[5][-1][8] = 400 + 2 * {[9 * 6 * (5 – 1)] + 6 * [(-1 + 4)]} + [8 – 5]
  = 400 + 2 * (9*6*4)+(6*3)+3
  = 400 + 2 * (237)
  = 874

Calculation of address of element of 3-D using row-major and column-major order

Column-Major Order

Address of A[i][j][k]= B + W(M * N(i – x) + M *(k – z) + (j – y))

B = Base Address (start address)
W = Weight (storage size of one element stored in the array)
M = Row (total number of rows)
N = Column (total number of columns)
P = Width (total number of cells depth-wise)
x = Lower Bound of Row
y = Lower Bound of Column
z = Lower Bound of Width

Example

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

Example

- **Given:**
  I = 3, J = 3, K = 3
  Base address B = 400
  Storage size of one element store in any array(in Byte) W = 4
  Lower Limit of row/start row index of matrix x = 1
  Lower Limit of column/start column index of matrix y = -5
  Lower Limit of blocks in matrix z = -10
  M (row)= Upper Bound – Lower Bound + 1 = 8-1+1 = 8
  N (column)= Upper Bound – Lower Bound + 1 = 5 +5 + 1 = 11
- **Formula used:**
  Address of arr[i][j][k] = B + W(M * N(i – x) + M * (j-y) + (k – z))
- **Solution:**
  Address of arr[3][3][3] = 400 + 4 * ((8*11*(3-1)+8*(3-(-5)+(3-(-10)))
  = 400 + 4 * ((88*2 + 8*8+13)
  = 400 + 4 * (253)
  = 400 + 1012
  = 1412

What is a sparse matrix?

- Sparse matrices are those matrices that have the majority of their elements equal to zero.
- In other words, the sparse matrix can be defined as the matrix that has a greater number of zero elements than the non-zero elements.

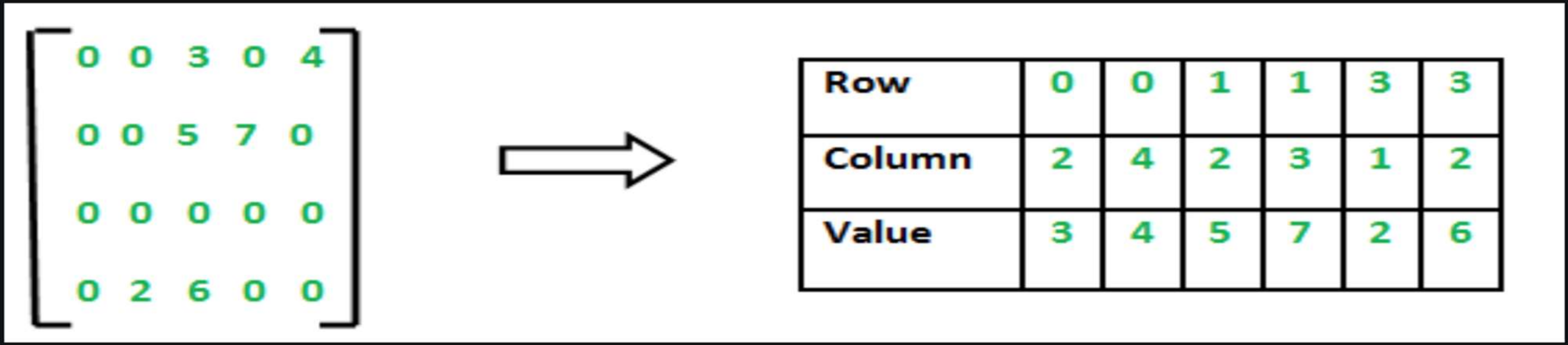
Sparse Matrix Representations can be done in many ways following are two common representations:

1. Array representation
2. Linked list representation

Method 1: 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)



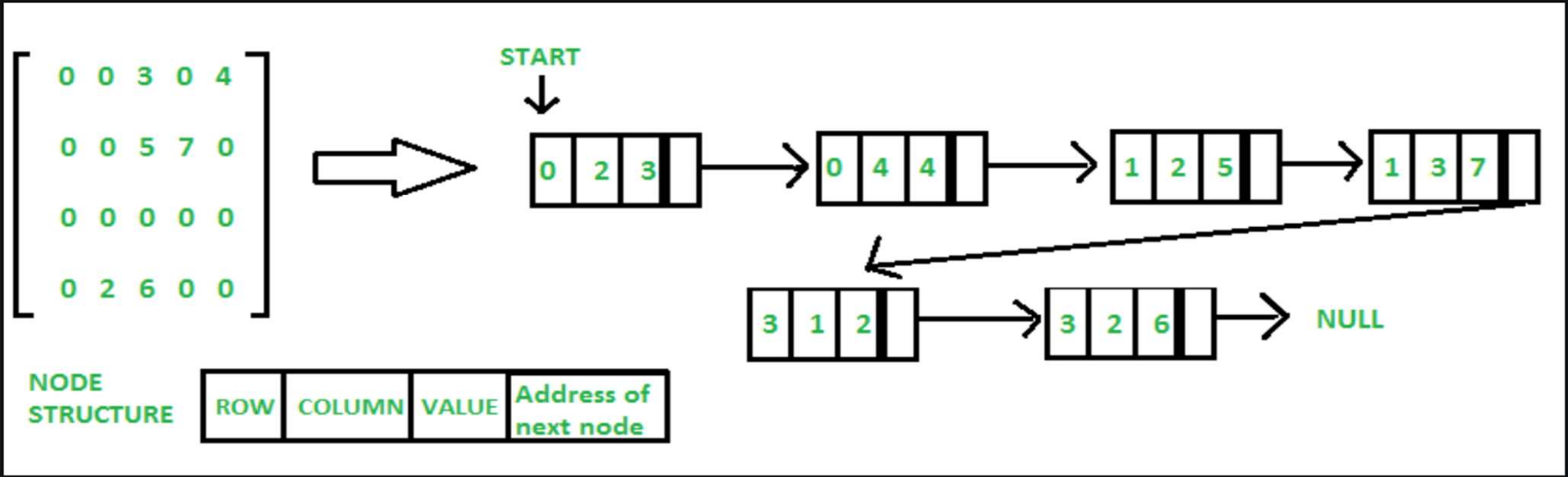
$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{bmatrix} \Rightarrow$$

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

Method 2: Using Linked Lists

In linked list, 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

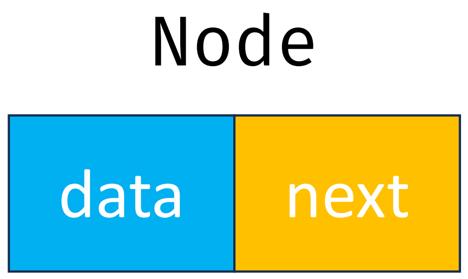


Abstract Data Types

- Linked List
- Stacks
- Queues
- Trees
- Graphs
- Hash Tables

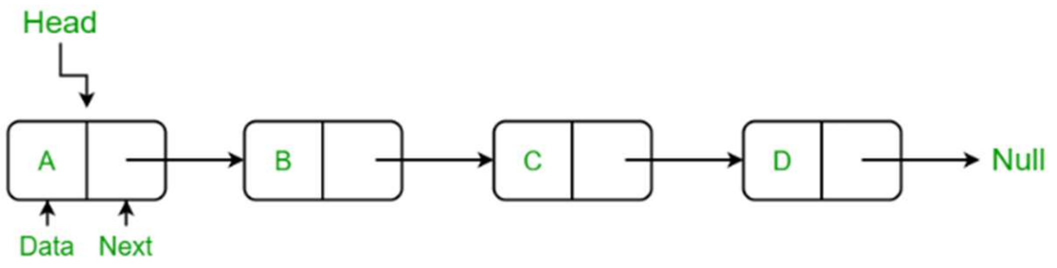
LinkedList

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

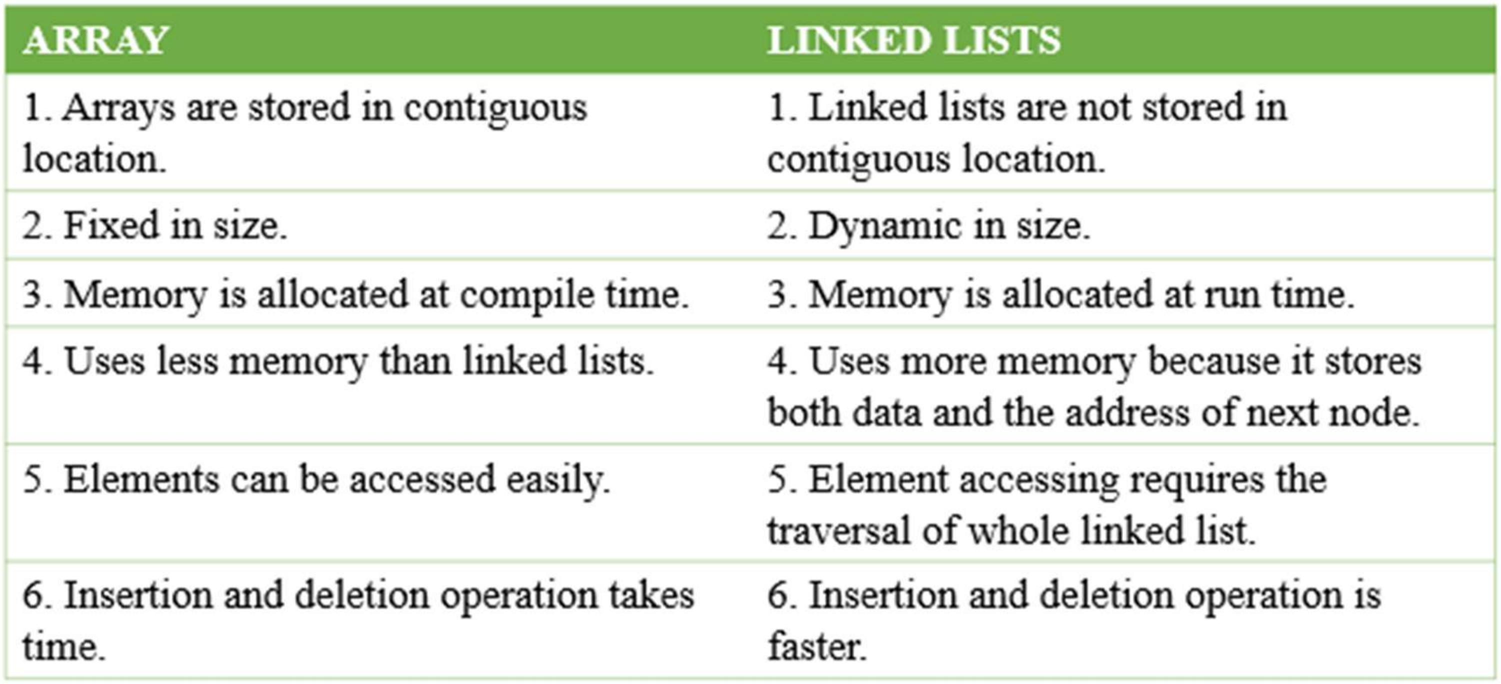


Linked List

- Linked List is a linear data structure.
- Elements are not stored at a contiguous memory location.
- Elements are linked using pointers.
- They include a series of connected nodes. Each node stores the data and the address of the next node.



Linked List



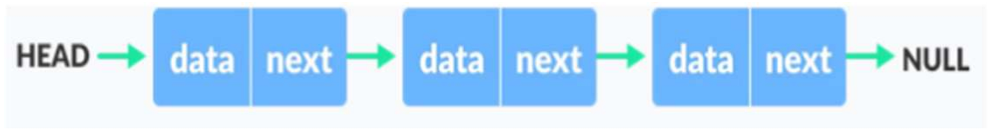
ARRAY	LINKED LISTS
1. Arrays are stored in contiguous location.	1. Linked lists are not stored in contiguous location.
2. Fixed in size.	2. Dynamic in size.
3. Memory is allocated at compile time.	3. Memory is allocated at run time.
4. Uses less memory than linked lists.	4. Uses more memory because it stores both data and the address of next node.
5. Elements can be accessed easily.	5. Element accessing requires the traversal of whole linked list.
6. Insertion and deletion operation takes time.	6. Insertion and deletion operation is faster.

Types of Linked Lists

- Singly Linked List
- Doubly Linked List
- Circular Linked List
- Circular Doubly Linked List

Singly Linked List

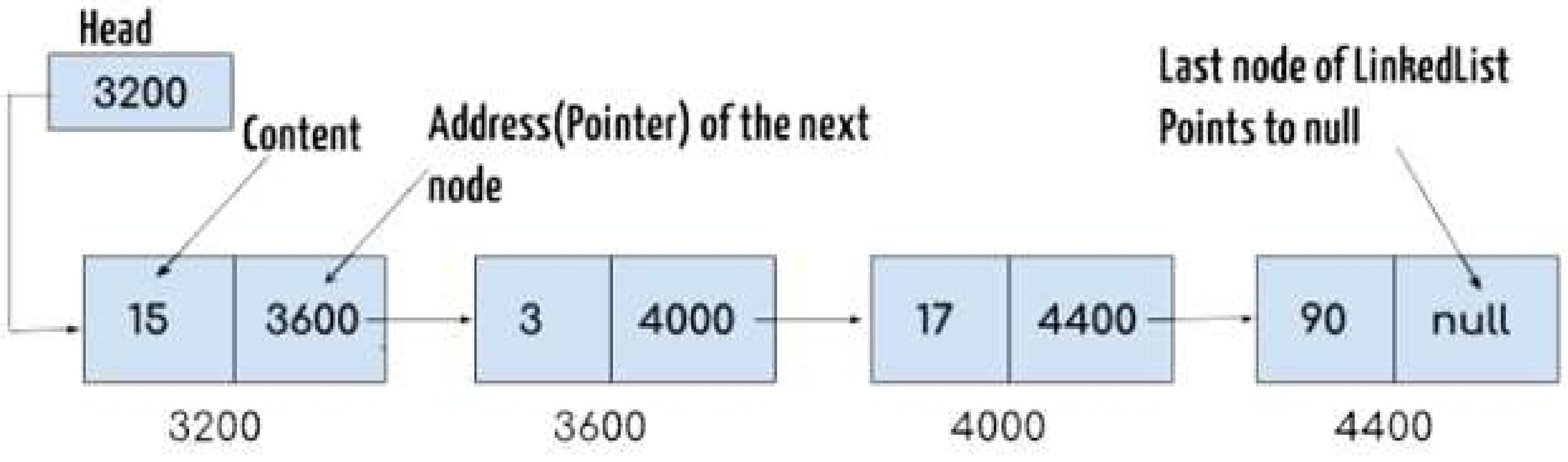
- Nodes are linked using pointers as shown below:



Traversal is unidirectional i.e. from the head node to the last node.

Singly Linked List

- Nodes are linked using pointers as shown below:

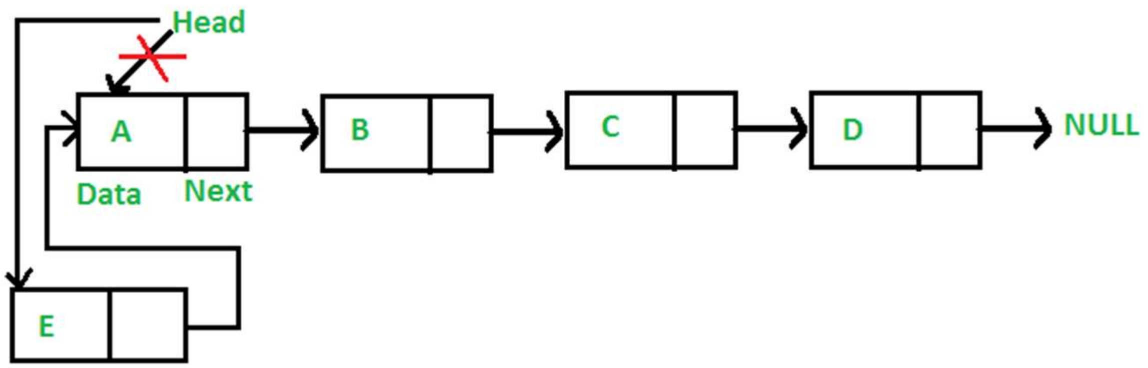


Head
3200
Content
Address(Pointer) of the next node
Last node of LinkedList Points to null
15
3600
3
4000
17
4400
90
null
3200
3600
4000
4400

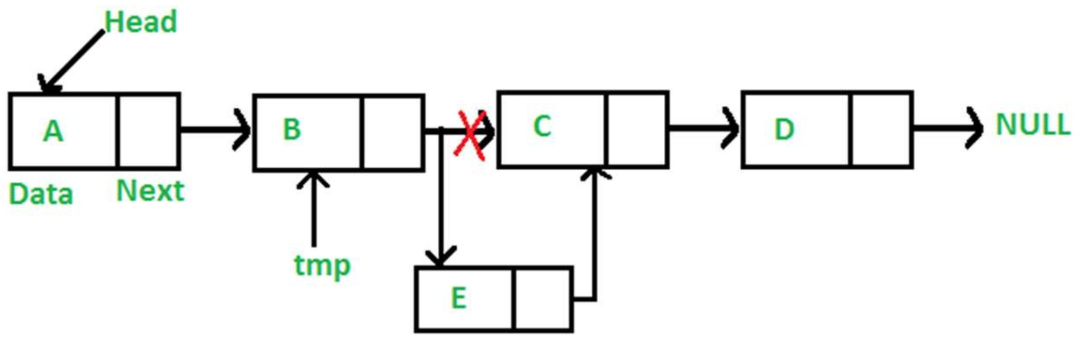
Singly Linked List

- Basic Operations
    - Insertion
    - Deletion
    - Insertion at position K
    - Deletion at position K
    - Insertion at the end
    - Deletion at the end

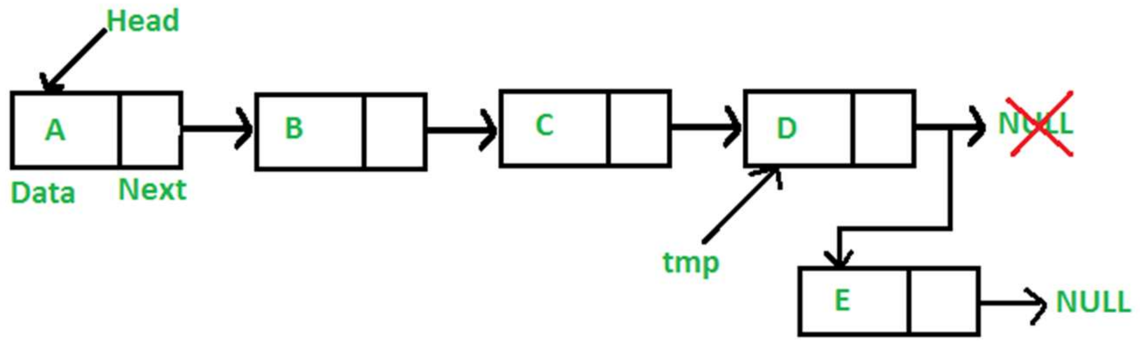
Insertion at the beginning

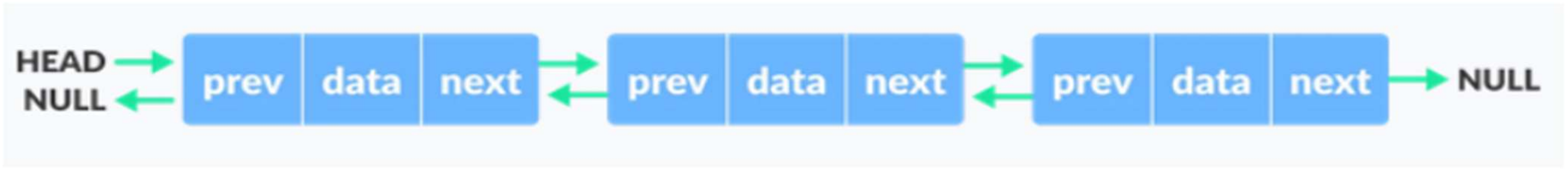


Insertion after a given node or position K



Insertion at the end





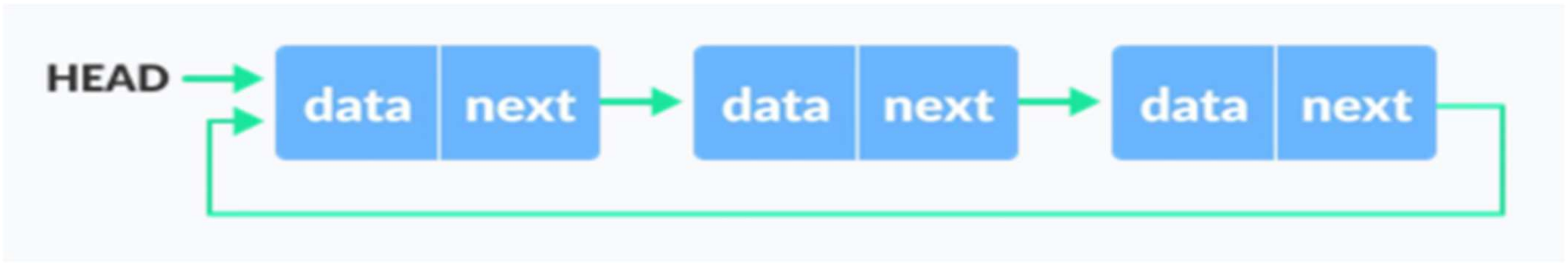
Doubly Linked List

- We add a pointer to the previous node.
- Can traverse in either direction: forward or backward

Traversal can be done in both ways, and hence it requires an extra pointer.

Circular Linked List

- Linked List (Singly, Doubly) whose last node is linked to the first node, forming a circular loop.



Linked Lists are most commonly used for:

- Linked Lists are mostly used because of their effective insertion and deletion.
- Insertion and deletion in the linked list are very effective and take less time complexity as compared to the array data structure.
- This data structure is simple and can be also used to implement a stack, queues, and other abstract data structures.

Applications of Linked Lists:

- Linked Lists are used to implement stacks and queues.
- It is used for the various representations of trees and graphs.
- It is used in dynamic memory allocation(linked list of free blocks).
- It is used for representing sparse matrices.
- It is used for the manipulation of polynomials.

Applications of Linked Lists:

- It is also used for performing arithmetic operations on long integers.
- It is used for finding paths in networks.
- In operating systems, they can be used in Memory management, process scheduling and file system.
- Linked lists can be used to improve the performance of algorithms that need to frequently insert or delete items from large collections of data.
- Implementing algorithms such as the LRU cache, which uses a linked list to keep track of the most recently used items in a cache.

Advantages of Linked Lists:

- Linked lists are a popular data structure in computer science, and have a number of advantages over other data structures, such as arrays. Some of the key advantages of linked lists are:
- Dynamic size: Linked lists do not have a fixed size, so you can add or remove elements as needed, without having to worry about the size of the list. This makes linked lists a great choice when you need to work with a collection of items whose size can change dynamically.
- Efficient Insertion and Deletion: Inserting or deleting elements in a linked list is fast and efficient, as you only need to modify the reference of the next node, which is an O(1) operation.

Advantages of Linked Lists:

- Memory Efficiency: Linked lists use only as much memory as they need, so they are more efficient with memory compared to arrays, which have a fixed size and can waste memory if not all elements are used.
- Easy to Implement: Linked lists are relatively simple to implement and understand compared to other data structures like trees and graphs.
- Flexibility: Linked lists can be used to implement various abstract data types, such as stacks, queues, and associative arrays.
- Easy to navigate: Linked lists can be easily traversed, making it easier to find specific elements or perform operations on the

Disadvantages of Linked Lists:

- Linked lists are a popular data structure in computer science, but like any other data structure, they have certain disadvantages as well. Some of the key disadvantages of linked lists are:
- Slow Access Time: Accessing elements in a linked list can be slow, as you need to traverse the linked list to find the element you are looking for, which is an O(n) operation. This makes linked lists a poor choice for situations where you need to access elements quickly.

Disadvantages of Linked Lists:

- Pointers: Linked lists use pointers to reference the next node, which can make them more complex to understand and use compared to arrays. This complexity can make linked lists more difficult to debug and maintain.
- Higher overhead: Linked lists have a higher overhead compared to arrays, as each node in a linked list requires extra memory to store the reference to the next node.

Disadvantages of Linked Lists:

- Cache Inefficiency: Linked lists are cache-inefficient because the memory is not contiguous. This means that when you traverse a linked list, you are not likely to get the data you need in the cache, leading to cache misses and slow performance.
- Extra memory required: Linked lists require an extra pointer for each node, which takes up extra memory. This can be a problem when you are working with large data sets, as the extra memory required for the pointers can quickly add up.