DS-Unit-1

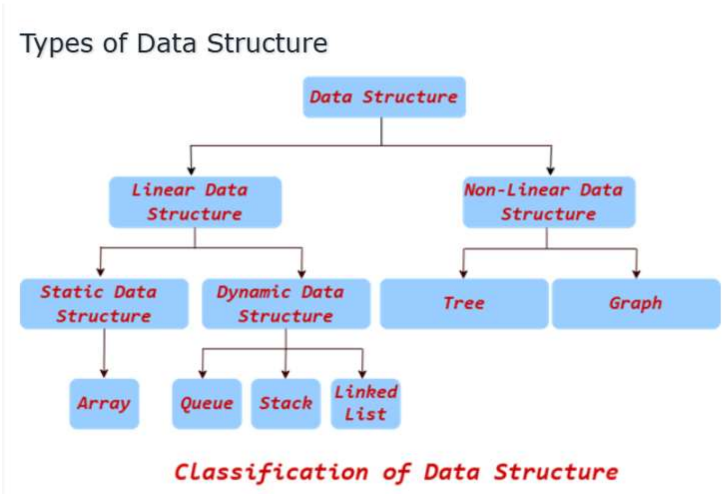
22 May 2025 08:46

The **data structure** name indicates that it organizes the data in memory.
Data structures are ways to organize, store, and manage data efficiently.

An **algorithm** is a set of rules that must be followed when solving a particular problem. In short, it is a step-by-step procedure to solve the problem. These steps describe a set of instructions to be executed in a certain order to get the expected output.

Key Features of Data Structure

Efficient Data Storage
Memory Utilization
Data Manipulation
Data Organization
Ease of Access
Flexibility
Optimization
Dynamic Size



Screen clipping taken: 22-05-2025 08:54

Data structures can be mainly categorized into the following:
1. Linear Data Structure
2. Non-Linear Data Structure

Linear Data Structure
A type of data structure where data elements are arranged linearly or sequentially, where each element is attached to its next as well as previous adjacent elements, is called a linear data structure.

Note that the order of elements is also preserved in linear data structures. It means that the first element added will be the first one to be accessed or removed. Linear data structures can have fixed or dynamic sizes.

Example of Linear Data Structure
Array, Stack, Queue, Linked List, etc., are examples of linear data structures.

The linear data structure can be further categorized into the following:
1. Static Data Structure
2. Dynamic Data Structure

Static Data Structure
A type of data structure that has a fixed size of memory is called a static data structure. Accessing elements is easier in a static data structure. An Array is an example of a static data structure.

Dynamic Data Structure
A type of data structure where the size is not fixed is known as a dynamic data structure. The size can be updated randomly during runtime, which is considered efficient concerning the space (memory) complexity of the code. Queue, Stack, Linked List, and Tree are examples of dynamic data structures.

Non-linear Data Structure
A type of data structure where the placement of data elements is not done linearly or sequentially is called a non-linear data structure. Traversing all the elements in a single run is not possible in a non-linear data structure.

Example of Non-linear Data Structure
Graphs, Trees, Heap, and HashTable are examples of non-linear data structures.

Basic Operations of Data Structure
In data structures, basic operations are essential for manipulating and managing data efficiently. Here are the following common operations that can be used across various data structures.

1. **Insertion**: Adding a new element to the data structure.
2. **Deletion**: Removing an element from the data structure.
3. **Traversal**: Accessing each element of the data structure to perform some operation.
4. **Searching**: Finding an element in the data structure.
5. **Sorting**: Arranging elements in a specific order (ascending or descending).
6. **Updation**: Modifying an existing element in the data structure.
7. **Merge:** Combining two or more sorted data structures into a single sorted data structure.
8. **Access:** Accessing or retrieving the value of an element based on its position (index) or key.
9. **Reversing:** Reversing the order of elements in the data structure.

Algorithm Complexity
The performance of the algorithm can be measured in two factors:

Time complexity
The time complexity of an algorithm is the amount of time required to complete its execution. The Big O notation denotes the time complexity of an algorithm. Here, big O notation is the asymptotic notation to represent the time complexity. The time complexity is mainly calculated by counting the number of steps to finish the execution.

Space Complexity
An algorithm's space complexity is the amount of space required to solve a problem and produce an output. Similar to the time complexity, space complexity is also expressed in big O notation.

For an algorithm, the space is required for the following purposes:

1. To store program instructions
2. To store constant values
3. To store variable values
4. To track the function calls, jumping statements, etc.

Auxiliary Space
The extra space required by the algorithm, excluding the input size, is known as auxiliary space. The space complexity considers both the spaces, i.e., auxiliary space and the space used by the input. Therefore, we can calculate the space complexity by using the following formula:

Worst case: It defines the input for which the algorithm takes a huge time.
Average case: It takes average time for the program execution.
Best case: It defines the input for which the algorithm takes the lowest time

Asymptotic Notations
The commonly used asymptotic notations used for calculating the running time complexity of an algorithm is given below:

- Big oh Notation (?)
- Omega Notation (Ω)

- Theta Notation (θ)

Big oh Notation (O)

- Big O notation is an asymptotic notation that measures the performance of an algorithm by simply providing the order of growth of the function.
- This notation provides an upper bound on a function which ensures that the function never grows faster than the upper bound. So, it gives the least upper bound on a function so that the function never grows faster than this upper bound.

It is the formal way to express the upper boundary of an algorithm running time. It measures the worst case of time complexity or the algorithm's longest amount of time to complete its operation.

Omega Notation (Ω)

- It basically describes the best-case scenario which is opposite to the big o notation.
- It is the formal way to represent the lower bound of an algorithm's running time. It measures the best amount of time an algorithm can possibly take to complete or the best-case time complexity.
- It determines what is the fastest time that an algorithm can run.

Theta Notation (θ)

- The theta notation mainly describes the average case scenarios.
- It represents the realistic time complexity of an algorithm. Every time, an algorithm does not perform worst or best, in real-world problems, algorithms mainly fluctuate between the worst-case and best-case, and this gives us the average case of the algorithm.
- Big theta is mainly used when the value of worst-case and the best-case is same.
- It is the formal way to express both the upper bound and lower bound of an algorithm running time.

What is abstract data type?

An abstract data type is an abstraction of a data structure that provides only the interface to which the data structure must adhere. The interface does not give any specific details about something should be implemented or in what programming language.

we can say that abstract data types are the entities that are definitions of data and operations but do not have implementation details

Abstract data type model

Before knowing about the abstract data type model, we should know about abstraction and encapsulation.
Abstraction: It is a technique of hiding the internal details from the user and only showing the necessary details to the user.
Encapsulation: It is a technique of combining the data and the member function in a single unit is known as encapsulation.

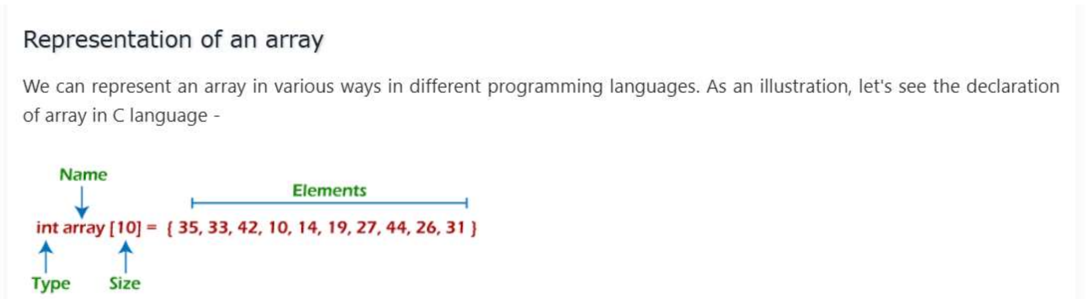
==

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.

- Each element in an array is of the same data type and carries the same size of 4 bytes.
- Elements in the array are stored at contiguous memory locations, and 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 data element's size.
- Fixed Size - The size of an array is fixed and cannot be changed after it is created. You decide how many elements it can hold during the declaration.

1. Index-based Access
2. Efficient Access
3. Insertion/Deletion Costly
4. Memory Wastage
5. Simple to Use
6. Cache Friendly


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 -

```
int array [10] = { 35, 33, 42, 10, 14, 19, 27, 44, 26, 31 }
```

Screen clipping taken: 22-05-2025 11:23

Basic operations

Now, let's discuss 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)**.

Multidimensional Array in Data Structure

A multidimensional array can be defined as an array of arrays.
The multidimensional array is organized as matrices which can be represented as the collection of rows and columns. It can be 2D, 3D and so on

1. `int arr[max_rows][max_columns];`

Due to the fact that the elements of 2D arrays can be random accessed. Similar to one dimensional arrays, we can access the individual cells in a 2D array by using the indices of the cells. There are two indices attached to a particular cell, one is its row number while the other is its column number.

2. `int arr[2][2] = {0,1,2,3};`
   The number of elements that can be present in a 2D array will always be equal to (**number of rows * number of columns**).

Sparse Matrix:

What is a matrix?

A matrix can be defined as a two-dimensional array having 'm' rows and 'n' columns. A matrix with m rows and n columns is called m n matrix. It is a set of numbers that are arranged in the horizontal or vertical lines of entries.

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

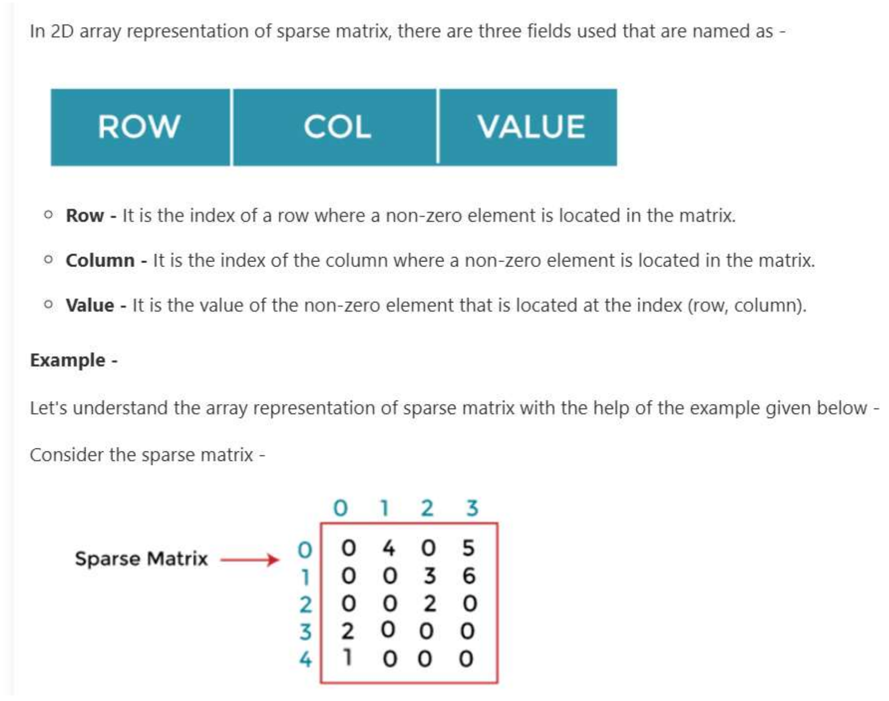
Storage - We know that a sparse matrix contains lesser non-zero elements than zero, so less memory can be used to store elements. It evaluates only the non-zero elements.

Computing time: In the case of searching in sparse matrix, we need to traverse only the non-zero elements rather than traversing all the sparse matrix elements. It saves computing time by logically designing a data structure traversing non-zero elements.

Representation of sparse matrix

Now, let's see the representation of the sparse matrix. The non-zero elements in the sparse matrix can be stored using triplets that are rows, columns, and values. There are two ways to represent the sparse matrix that are listed as follows -

- Array representation
- Linked list representation



In 2D array representation of sparse matrix, there are three fields used that are named as -

ROW	COL	VALUE

- **Row -** It is the index of a row where a non-zero element is located in the matrix.
- **Column -** It is the index of the column where a non-zero element is located in the matrix.
- **Value -** It is the value of the non-zero element that is located at the 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 -

Sparse Matrix $\longrightarrow$

$$\begin{array}{c|cccc|} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 4 & 0 & 5 \\ 1 & 0 & 0 & 3 & 6 \\ 2 & 0 & 0 & 2 & 0 \\ 3 & 2 & 0 & 0 & 0 \\ 4 & 1 & 0 & 0 & 0 \\ \hline \end{array}$$

Screen clipping taken: 22-05-2025 13:28

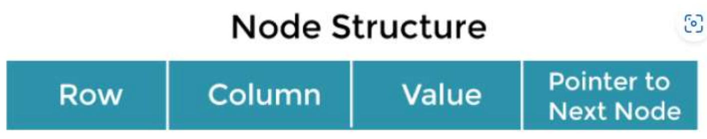
Linked List representation of the sparse matrix

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 -

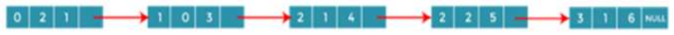


Screen clipping taken: 22-05-2025 13:30

Sparse Matrix $\longrightarrow$
$$\begin{array}{c|cccc|} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 0 & 1 & 0 \\ 1 & 3 & 0 & 0 & 0 \\ 2 & 0 & 4 & 5 & 0 \\ 3 & 0 & 6 & 0 & 0 \\ \hline \end{array}$$

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.

===

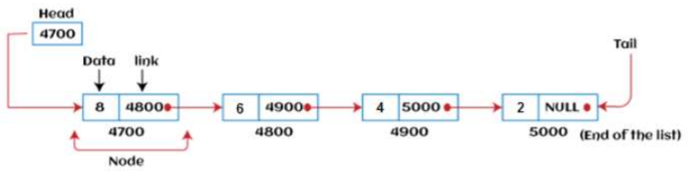
Linked List in Data Structure

21 Apr 2025 | 6 min read

Linked list is a linear data structure that includes a series of connected nodes.
Linked list can be defined as the nodes that are randomly stored in the memory.
A node in the linked list contains two parts, i.e., first is the data part and second is the address part.
The last node of the list contains a pointer to the null.

Representation of a Linked List

Linked list can be represented as the connection of nodes in which each node points to the next node of the list. The representation of the linked list is shown below -



Screen clipping taken: 22-05-2025 13:50

Types of Linked List:
1. Single linked list
2. Doubly linked list
3. Circular singly linked list
4. Circular doubly linked list

Advantages:
Dynamic data structure
Insertion and deletion
Memory efficient
Implementation

Disadvantage:
Memory usage
Traversal
Reverse Traversing

Applications of Linked list:

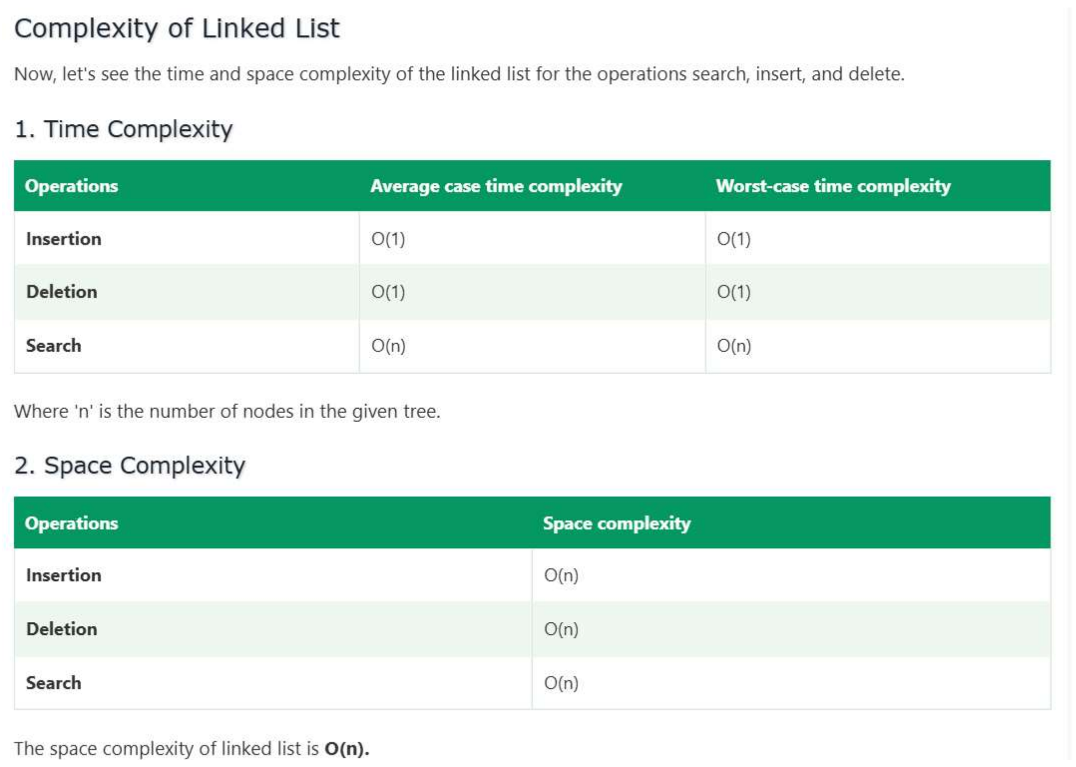
- With the help of a linked list, the polynomials can be represented as well as we can perform the operations on the polynomial.
- A linked list can be used to represent the sparse matrix.
- The various operations like student's details, employee's details, or product details can be implemented using the linked list as the linked list uses the structure data type that can hold different data types.

- Using linked list, we can implement stack, queue, tree, and other various data structures.
- The graph is a collection of edges and vertices, and the graph can be represented as an adjacency matrix and adjacency list. If we want to represent the graph as an adjacency matrix, then it can be implemented as an array. If we want to represent the graph as an adjacency list, then it can be implemented as a linked list.
- A linked list can be used to implement dynamic memory allocation. The dynamic memory allocation is the memory allocation done at the run-time.

Operations performed on Linked List

The basic operations that are supported by a list are mentioned as follows -

- **Insertion :** This operation is performed to add an element into the list.
- **Deletion :** It is performed to delete an operation from the list.
- **Traversal :** It is performed to traverse the elements of the list.
- **Search :** It is performed to search an element from the list using the given key.



Complexity of Linked List

Now, let's see the time and space complexity of the linked list for the operations search, insert, and delete.

1. Time Complexity

Operations	Average case time complexity	Worst-case time complexity
Insertion	O(1)	O(1)
Deletion	O(1)	O(1)
Search	O(n)	O(n)

Where 'n' is the number of nodes in the given tree.

2. Space Complexity

Operations	Space complexity
Insertion	O(n)
Deletion	O(n)
Search	O(n)

The space complexity of linked list is **O(n).**

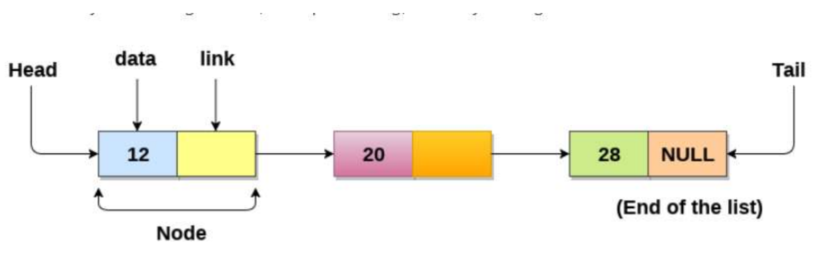
Screen clipping taken: 22-05-2025 14:01

==

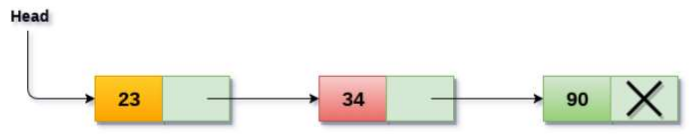
Singly linked list or One way chain

Singly linked list can be defined as the collection of ordered set of elements.
The number of elements may vary according to need of the program.
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.

One way chain or singly linked list can be traversed only in one direction. In other words,
we can say that each node contains only next pointer, therefore we can not traverse the list in the reverse direction.



Screen clipping taken: 22-05-2025 14:27



Screen clipping taken: 22-05-2025 14:27

Operations on Singly Linked List

There are various operations which can be performed on singly linked list. A list of all such operations is given below.

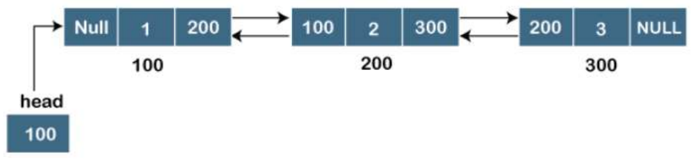
Insertion:
  at beginning,
  at end of the list,
  after specified node
Deletion:
  at beginning,
  at end of the list,
  after specified node
Traversing
Searching

==

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). A sample node in a doubly linked list is shown in the figure.

Suppose we have three nodes, and the address of these nodes are 100, 200 and 300, respectively.
The representation of these nodes in a doubly-linked list is shown below:



Operations on doubly linked list

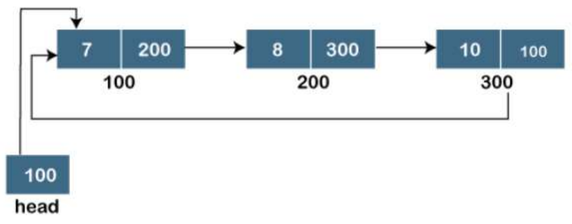
Insertion:
  at beginning,
  at end of the list,
  after specified node
Deletion:
  at beginning,
  at end of the list,
  after specified node
Traversing
Searching

==

Circular Linked List

In a circular Singly linked list, the last node of the list contains a pointer to the first node of the list.
We can have circular singly linked list as well as circular doubly linked list.
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.
There is no null value present in the next part of any of the nodes.
We can traverse in any direction, i.e., either backward or forward
The following image shows a circular singly linked list.



Operations on Circular Singly linked list:

Insertion ar beginning
Insertion at the end
Deletion at beginning
Deletion at the end
Searching
Traversing

===

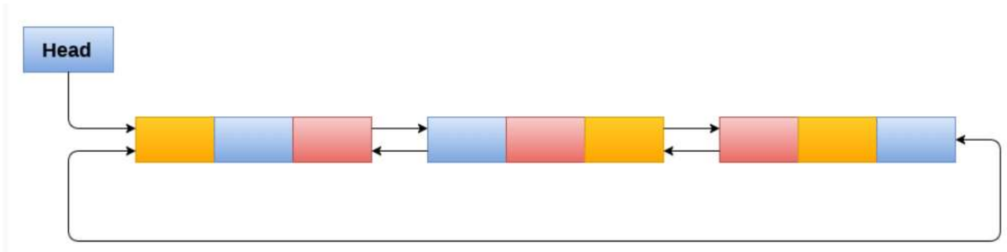
Circular Doubly Linked List

Circular doubly linked list is a more complexed type of data structure.
In which a node contain pointers to its previous node as well as the next node.
Circular doubly linked list doesn't contain NULL in any of the node.
The last node of the list contains the address of the first node of the list.
The first node of the list also contain address of the last node in its previous pointer.
A circular doubly linked list is shown in the following figure.

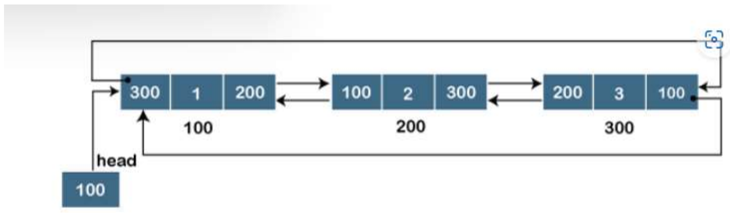
Due to the fact that a circular doubly linked list contains three parts in its structure therefore, it demands more space per node and more expensive basic operations.

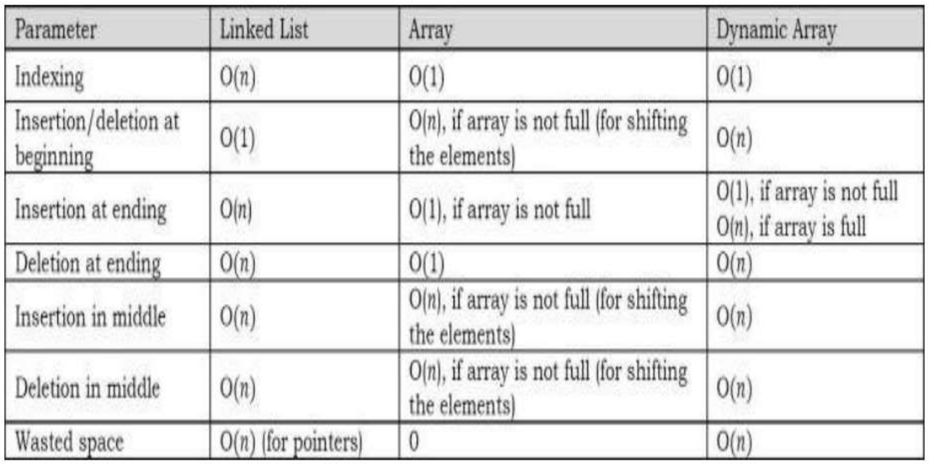
1	Insertion at beginning	Adding a node in circular doubly linked list at the beginning.
2	Insertion at end	Adding a node in circular doubly linked list at the end.
3	Deletion at beginning	Removing a node in circular doubly linked list from beginning.
4	Deletion at end	Removing a node in circular doubly linked list at the end.

Traversing and searching in circular doubly linked list is similar to that in the circular singly linked list.



Circular Doubly Linked List





Parameter	Linked List	Array	Dynamic Array
Indexing	$O(n)$	$O(1)$	$O(1)$
Insertion/deletion at beginning	$O(1)$	$O(n)$, if array is not full (for shifting the elements)	$O(n)$
Insertion at ending	$O(n)$	$O(1)$, if array is not full	$O(1)$, if array is not full $O(n)$, if array is full
Deletion at ending	$O(n)$	$O(1)$	$O(n)$
Insertion in middle	$O(n)$	$O(n)$, if array is not full (for shifting the elements)	$O(n)$
Deletion in middle	$O(n)$	$O(n)$, if array is not full (for shifting the elements)	$O(n)$
Wasted space	$O(n)$ (for pointers)	0	$O(n)$

Screen clipping taken: 22-05-2025 15:22