### **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.

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**

1. **Non-Primitive Data Structures** are those data structures derived from Primitive Data Structures.
2. These data structures can't be manipulated or operated directly by machine-level instructions.
3. The focus of these data structures is on forming a set of data elements that is either **homogeneous** (same data type) or **heterogeneous** (different data types).

* 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 structure**

-> Primititve

->Interger,float ,char,double,pointer

->Non- premititive

->liner ds

-> Static

->array

-> dynamic

->stack ,queue, linked list

-> non liner ds

->tree

->graph

Note for non premititve ds

[ The focus of these data structures is on forming a set of data elements that is either **homogeneous** (same data type) or **heterogeneous** (different data types). ]

**Arrays – [ Static Ds ]**

An **Array** is a data structure used to collect multiple data elements of the same data type into one variable. Instead of storing multiple values of the same data types in separate variable names, we could store all of them together into one variable. This statement doesn't imply that we will have to unite all the values of the same data type in any program into one array of that data type. But there will often be times when some specific variables of the same data types are all related to one another in a way appropriate for an array.

An Array is a list of elements where each element has a unique place in the list. The data elements of the array share the same variable name; however, each carries a different index number called a **subscript**. We can access any data element from the list with the help of its location in the list. Thus, the key feature of the arrays to understand is that the data is stored in **contiguous memory** locations, making it possible for the users to traverse through the data elements of the array using their respective indexes.

**Arrays can be classified into different types:**

**One-Dimensional Array:** An Array with only one row of data elements is known as a One-Dimensional Array. It is stored in ascending storage location.

**Two-Dimensional Array:** An Array consisting of multiple rows and columns of data elements is called a Two-Dimensional Array. It is also known as a Matrix.

**Multidimensional Array:** We can define Multidimensional Array as an Array of Arrays. Multidimensional Arrays are not bounded to two indices or two dimensions as they can include as many indices are per the need.

**Some Applications of Array:**

🡪We can store a list of data elements belonging to the same data type.

🡪Array acts as an auxiliary storage for other data structures.

🡪The array also helps store data elements of a binary tree of the fixed count.

🡪Array also acts as a storage of matrices.

**Linked Lists – [ Dynamic Ds ]**

A **Linked List** is another example of a linear data structure used to store a collection of data elements dynamically. Data elements in this data structure are represented by the Nodes, connected using links or pointers. Each node contains two fields, the information field consists of the actual data, and the pointer field consists of the address of the subsequent nodes in the list. The pointer of the last node of the linked list consists of a null pointer, as it points to nothing. Unlike the Arrays, the user can dynamically adjust the size of a Linked List as per the requirements.

**Linked Lists can be classified into different types:**

**Singly Linked List:** A Singly Linked List is the most common type of Linked List. Each node has data and a pointer field containing an address to the next node.

**Doubly Linked List:** A Doubly Linked List consists of an information field and two pointer fields. The information field contains the data. The first pointer field contains an address of the previous node, whereas another pointer field contains a reference to the next node. Thus, we can go in both directions (backward as well as forward).

**Circular Linked List:** The Circular Linked List is similar to the Singly Linked List. The only key difference is that the last node contains the address of the first node, forming a circular loop in the Circular Linked List.

**Some Applications of Linked Lists:**

1. The Linked Lists help us implement stacks, queues, binary trees, and graphs of predefined size.
2. We can also implement Operating System's function for dynamic memory management.
3. Linked Lists also allow polynomial implementation for mathematical operations.
4. We can use Circular Linked List to implement Operating Systems or application functions that Round Robin execution of tasks.
5. Circular Linked List is also helpful in a Slide Show where a user requires to go back to the first slide after the last slide is presented.
6. Doubly Linked List is utilized to implement forward and backward buttons in a browser to move forward and backward in the opened pages of a website.

**Stacks**

A **Stack** is a Linear Data Structure that follows the **LIFO** (Last In, First Out) principle that allows operations like insertion and deletion from one end of the Stack, i.e., Top. Stacks can be implemented with the help of contiguous memory, an Array, and non-contiguous memory, a Linked List. Real-life examples of Stacks are piles of books, a deck of cards, piles of money, and many more.

**Some Applications of Stacks:**

1. The Stack is used as a Temporary Storage Structure for recursive operations.
2. Stack is also utilized as Auxiliary Storage Structure for function calls, nested operations, and deferred/postponed functions.
3. We can manage function calls using Stacks.
4. Stacks are also utilized to evaluate the arithmetic expressions in different programming languages.
5. Stacks are also helpful in converting infix expressions to postfix expressions.
6. Stacks allow us to check the expression's syntax in the programming environment.
7. We can match parenthesis using Stacks.
8. Stacks can be used to reverse a String.
9. Stacks are helpful in solving problems based on backtracking.
10. We can use Stacks in depth-first search in graph and tree traversal.
11. Stacks are also used in Operating System functions.
12. Stacks are also used in UNDO and REDO functions in an edit.

**Queues**

A **Queue** is a linear data structure similar to a Stack with some limitations on the insertion and deletion of the elements. The insertion of an element in a Queue is done at one end, and the removal is done at another or opposite end. Thus, we can conclude that the Queue data structure follows FIFO (First In, First Out) principle to manipulate the data elements. Implementation of Queues can be done using Arrays, Linked Lists, or Stacks. Some real-life examples of Queues are a line at the ticket counter, an escalator, a car wash, and many more.

**Some Applications of Queues:**

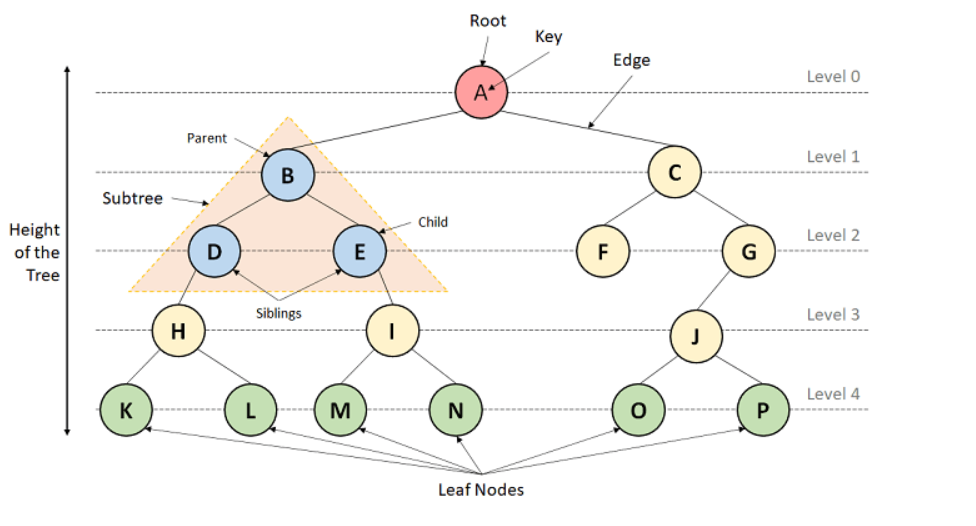
1. Queues are generally used in the breadth search operation in Graphs.
2. Queues are also used in Job Scheduler Operations of Operating Systems, like a keyboard buffer queue to store the keys pressed by users and a print buffer queue to store the documents printed by the printer.
3. Queues are responsible for CPU scheduling, Job scheduling, and Disk Scheduling.
4. Priority Queues are utilized in file-downloading operations in a browser.
5. Queues are also used to transfer data between peripheral devices and the CPU.
6. Queues are also responsible for handling interrupts generated by the User Applications for the CPU.

### **Non-Linear Data Structures**

**Trees**

A Tree is a Non-Linear Data Structure and a hierarchy containing a collection of nodes such that each node of the tree stores a value and a list of references to other nodes (the "children").

The Tree data structure is a specialized method to arrange and collect data in the computer to be utilized more effectively. It contains a central node, structural nodes, and sub-nodes connected via edges. We can also say that the tree data structure consists of roots, branches, and leaves connected.



**Trees can be classified into different types:**

1. **Binary Tree:** A Tree data structure where each parent node can have at most two children is termed a Binary Tree.
2. **Binary Search Tree:** A Binary Search Tree is a Tree data structure where we can easily maintain a sorted list of numbers.
3. **AVL Tree:** An AVL Tree is a self-balancing Binary Search Tree where each node maintains extra information known as a Balance Factor whose value is either -1, 0, or +1.
4. **B-Tree:** A B-Tree is a special type of self-balancing Binary Search Tree where each node consists of multiple keys and can have more than two children.

**Some Applications of Trees:**

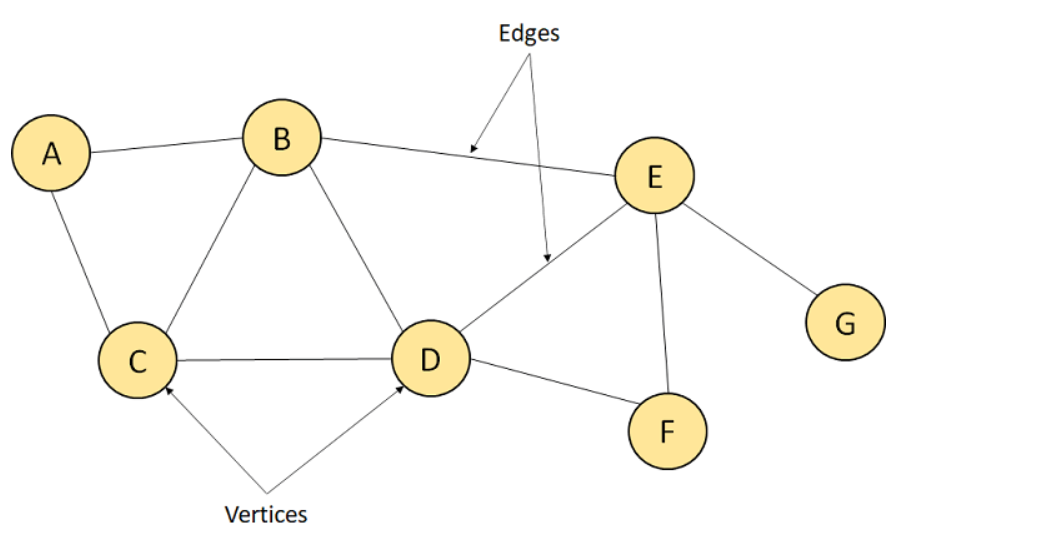
1. Trees implement hierarchical structures in computer systems like directories and file systems.
2. Trees are also used to implement the navigation structure of a website.
3. We can generate code like Huffman's code using Trees.
4. Trees are also helpful in decision-making in Gaming applications.
5. Trees are responsible for implementing priority queues for priority-based OS scheduling functions.
6. Trees are also responsible for parsing expressions and statements in the compilers of different programming languages.
7. We can use Trees to store data keys for indexing for Database Management System (DBMS).
8. Spanning Trees allows us to route decisions in Computer and Communications Networks.
9. Trees are also used in the path-finding algorithm implemented in Artificial Intelligence (AI), Robotics, and Video Games Applications.

**Graphs**

A Graph is another example of a Non-Linear Data Structure comprising a finite number of nodes or vertices and the edges connecting them. The Graphs are utilized to address problems of the real world in which it denotes the problem area as a network such as social networks, circuit networks, and telephone networks. For instance, the nodes or vertices of a Graph can represent a single user in a telephone network, while the edges represent the link between them via telephone.

**Some Applications of Graphs:**

1. Graphs help us represent routes and networks in transportation, travel, and communication applications.
2. Graphs are used to display routes in GPS.
3. Graphs also help us represent the interconnections in social networks and other network-based applications.
4. Graphs are utilized in mapping applications.
5. Graphs are responsible for the representation of user preference in e-commerce applications.
6. Graphs are also used in Utility networks in order to identify the problems posed to local or municipal corporations.
7. Graphs also help to manage the utilization and availability of resources in an organization.
8. Graphs are also used to make document link maps of the websites in order to display the connectivity between the pages through hyperlinks.
9. Graphs are also used in robotic motions and neural networks.



## **Some Applications of Data Structures**

1. Data Structures help in the organization of data in a computer's memory.
2. Data Structures also help in representing the information in databases.
3. Data Structures allows the implementation of algorithms to search through data (For example, search engine).
4. We can use the Data Structures to implement the algorithms to manipulate data (For example, word processors).
5. We can also implement the algorithms to analyse data using Data Structures (For example, data miners).
6. Data Structures support algorithms to generate the data (For example, a random number generator).
7. Data Structures also support algorithms to compress and decompress the data (For example, a zip utility).
8. We can also use Data Structures to implement algorithms to encrypt and decrypt the data (For example, a security system).
9. With the help of Data Structures, we can build software that can manage files and directories (For example, a file manager).
10. We can also develop software that can render graphics using Data Structures. (For example, a web browser or 3D rendering software).

## **Array:**

## **Basic operations**

* 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.

### Traversal operation

public class ArrayExample {

public static void main(String[] args) {

int[] Arr = {18, 30, 15, 70, 12};

System.out.println("Elements of the array are:");

for (int i = 0; i < 5; i++) {

System.out.printf("Arr[%d] = %d, ", i, Arr[i]);

}

}

}

### Insertion operation

public class ArrayInsertion {

public static void main(String[] args) {

int[] arr = {18, 30, 15, 70, 12};

int i, x, pos, n = 5;

System.out.println("Array elements before insertion");

for (i = 0; i < n; i++)

System.out.print(arr[i] + " ");

System.out.println();

x = 50; // element to be inserted

pos = 4;

n++;

for (i = n - 1; i >= pos; i--)

arr[i] = arr[i - 1];

arr[pos - 1] = x;

System.out.println("Array elements after insertion");

for (i = 0; i < n; i++)

System.out.print(arr[i] + " ");

System.out.println();

}

}

### Deletion operation

public class ArrayDeletion {

public static void main(String[] args) {

int[] arr = {18, 30, 15, 70, 12};

int i, pos, n = 5;

System.out.println("Array elements before deletion");

for (i = 0; i < n; i++)

System.out.print(arr[i] + " ");

System.out.println();

pos = 3; // position of element to be deleted

if (pos >= 1 && pos <= n) {

for (i = pos - 1; i < n - 1; i++)

arr[i] = arr[i + 1];

n--;

System.out.println("Array elements after deletion");

for (i = 0; i < n; i++)

System.out.print(arr[i] + " ");

System.out.println();

} else {

System.out.println("Invalid position to delete");

}

}

}

### Search operation

public class ArraySearch {

public static void main(String[] args) {

int[] arr = {18, 30, 15, 70, 12};

int searchElement = 70;

int n = arr.length;

boolean found = false;

int i;

System.out.println("Array elements:");

for (i = 0; i < n; i++)

System.out.print(arr[i] + " ");

System.out.println();

for (i = 0; i < n; i++) {

if (arr[i] == searchElement) {

found = true;

break;

}

}

if (found) {

System.out.println("Element " + searchElement + " found at index " + i);

} else {

System.out.println("Element " + searchElement + " not found in the array");

}

}

}

### Update operation

public class ArrayUpdate {

public static void main(String[] args) {

int[] arr = {18, 30, 15, 70, 12};

int updateElement = 50;

int pos = 2; // position to update

int n = arr.length;

System.out.println("Array elements before update:");

for (int i = 0; i < n; i++)

System.out.print(arr[i] + " ");

System.out.println();

if (pos >= 1 && pos <= n) {

arr[pos - 1] = updateElement;

System.out.println("Array elements after update:");

for (int i = 0; i < n; i++)

System.out.print(arr[i] + " ");

System.out.println();

} else {

System.out.println("Invalid position for update");

}

}

}

**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**

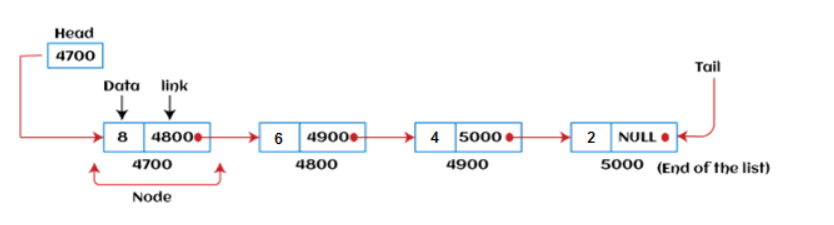
* Array provides the single name for the group of variables of the same type. Therefore, it is easy to remember the name of all the elements of an array.
* Traversing an array is a very simple process; we just need to increment the base address of the array in order to visit each element one by one.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

# Linked list

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. After array, linked list is the second most used data structure. In a linked list, every link contains a connection to another link.



### **Why use linked list over array?**

Linked list is a data structure that overcomes the limitations of arrays. Let's first see some of the limitations of arrays -

* The size of the array must be known in advance before using it in the program.
* Increasing the size of the array is a time taking process. It is almost impossible to expand the size of the array at run time.
* All the elements in the array need to be contiguously stored in the memory. Inserting an element in the array needs shifting of all its predecessors.

Linked list is useful because -

* It allocates the memory dynamically. All the nodes of the linked list are non-contiguously stored in the memory and linked together with the help of pointers.
* In linked list, size is no longer a problem since we do not need to define its size at the time of declaration. List grows as per the program's demand and limited to the available memory space.

### **Types of Linked list**

Linked list is classified into the following types -

* **Singly-linked list -** Singly linked list can be defined as the collection of an ordered set of elements. A node in the singly linked list consists 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.
* **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), and pointer to the previous node (previous pointer).
* **Circular singly linked list -** In a circular singly linked list, the last node of the list contains a pointer to the first node of the list. We can have circular singly linked list as well as circular doubly linked list.
* **Circular doubly linked list -** Circular doubly linked list is a more complex type of data structure in which a node contains pointers to its previous node as well as the next node. Circular doubly linked list doesn't contain NULL in any of the nodes. The last node of the list contains the address of the first node of the list. The first node of the list also contains the address of the last node in its previous pointer.

### **Advantages of Linked list**

* **Dynamic data structure -** The size of the linked list may vary according to the requirements. Linked list does not have a fixed size.
* **Insertion and deletion -** Unlike arrays, insertion, and deletion in linked list is easier. Array elements are stored in the consecutive location, whereas the elements in the linked list are stored at a random location. To insert or delete an element in an array, we have to shift the elements for creating the space. Whereas, in linked list, instead of shifting, we just have to update the address of the pointer of the node.
* **Memory efficient -** The size of a linked list can grow or shrink according to the requirements, so memory consumption in linked list is efficient.
* **Implementation -** We can implement both stacks and queues using linked list.

### **Disadvantages of Linked list**

* **Memory usage -** In linked list, node occupies more memory than array. Each node of the linked list occupies two types of variables, i.e., one is a simple variable, and another one is the pointer variable.
* **Traversal -** Traversal is not easy in the linked list. If we have to access an element in the linked list, we cannot access it randomly, while in case of array we can randomly access it by index. For example, if we want to access the 3rd node, then we need to traverse all the nodes before it. So, the time required to access a particular node is large.
* **Reverse traversing -** Backtracking or reverse traversing is difficult in a linked list. In a doubly-linked list, it is easier but requires more memory to store the back pointer.

### **Applications of Linked list**

The applications of the Linked list are given as follows -

* 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.

### **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.

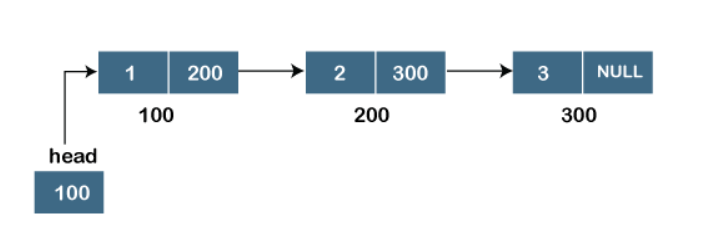
### **2. Space Complexity**

|  |  |
| --- | --- |
| **Operations** | **Space complexity** |
| **Insertion** | O(n) |
| **Deletion** | O(n) |
| **Search** | O(n) |

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

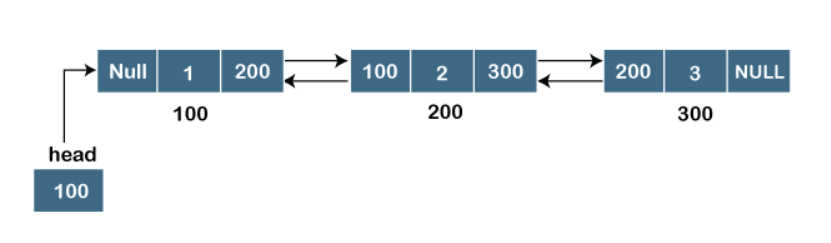
### **Singly Linked list**

It is the commonly used linked list in programs. If we are talking about the linked list, it means it is a singly linked list. The singly linked list is a data structure that contains two parts, i.e., one is the data part, and the other one is the address part, which contains the address of the next or the successor node. The address part in a node is also known as a **pointer**.



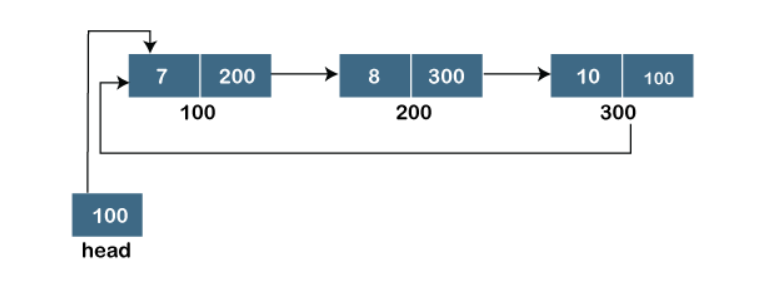
### **Doubly linked list**

As the name suggests, the doubly linked list contains two pointers. We can define the doubly linked list as a linear data structure with three parts: the data part and the other two address part. In other words, a doubly linked list is a list that has three parts in a single node, includes one data part, a pointer to its previous node, and a pointer to the next node.



### **Circular linked list**

A circular linked list is a variation of a singly linked list. The only difference between the **singly linked list** and a **circular linked** list is that the last node does not point to any node in a singly linked list, so its link part contains a NULL value. On the other hand, the circular linked list is a list in which the last node connects to the first node, so the link part of the last node holds the first node's address. The circular linked list has no starting and ending node. We can traverse in any direction, i.e., either backward or forward. The diagrammatic representation of the circular linked list is shown below:



### **Doubly Circular linked list**

The doubly circular linked list has the features of both the **circular linked list** and **doubly linked list**.

