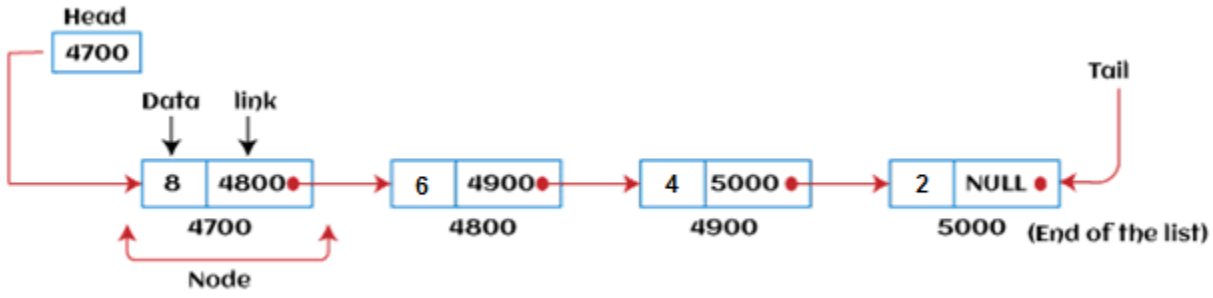
UNIT III

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.

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 -



Till now, we have been using array data structure to organize the group of elements that are to be stored individually in the memory. However, Array has several advantages and disadvantages that must be known to decide the data structure that will be used throughout the program.

How to declare a linked list?

It is simple to declare an array, as it is of single type, while the declaration of linked list is a bit more typical than array. Linked list contains two parts, and both are of different

types, i.e., one is the simple variable, while another is the pointer variable. We can declare the linked list by using the user-defined data type **structure.**

The declaration of linked list is given as follows -

```
1. struct node
2. {
3. int data;
4. struct node *next;
5. }
```

In the above declaration, we have defined a structure named as **node** that contains two variables, one is **data** that is of integer type, and another one is **next** that is a pointer which contains the address of next node.

Now, let's move towards the types of linked list.

Advantages of Linked list

The advantages of using the Linked list are given as follows -

- **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

The limitations of using the Linked list are given as follows -

- **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.

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.
- **Display -** It is performed to display the elements of the list.
- **Search -** It is performed to search an element from the list using the given key.

Types of Linked List

Before knowing about the types of a linked list, we should know what is ***linked list***. So, to know about the linked list, click on the link given below:

Types of Linked list

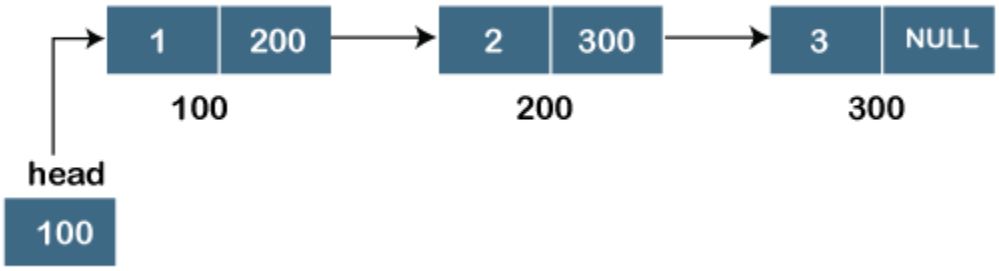
The following are the types of linked list:

- Singly Linked list
- Doubly Linked list
- Circular Linked list

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**.

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



We can observe in the above figure that there are three different nodes having address 100, 200 and 300 respectively. The first node contains the address of the next node, i.e., 200, the second node contains the address of the last node, i.e., 300, and the third node contains the NULL value in its address part as it does not point to any node. The pointer that holds the address of the initial node is known as a ***head pointer***.

The linked list, which is shown in the above diagram, is known as a singly linked list as it contains only a single link. In this list, only forward traversal is possible; we cannot traverse in the backward direction as it has only one link in the list.

Representation of the node in a singly linked list

```
1. struct node
2. {
3.    int data;
4.    struct node *next;
5. }
```

In the above representation, we have defined a user-defined structure named a **node** containing two members, the first one is data of integer type, and the other one is the pointer (next) of the node type.

What is a Singly Linked List?

A linked list is a linear data structure with several nodes that are linked to one another. Each node has two fields: the data field and the following node's address. So, in this article, we'll look at how to implement insertion at the start of a singly linked list in C++. Here is a template for a singly linked list Node that we create.

Syntax of Linked List

```
class Node

{

    int data;

    Node *next;

};
```

How to Insert at the Beginning of a Singly Linked List in C++

Assume you've been given a single linked list.The goal is to place a node at the start of the given linked list.

For example,
Input :

```
10 – 20 → 30 → 40 → 50

Insert '0' at the head or beginning of the given linked list.
```

Output :

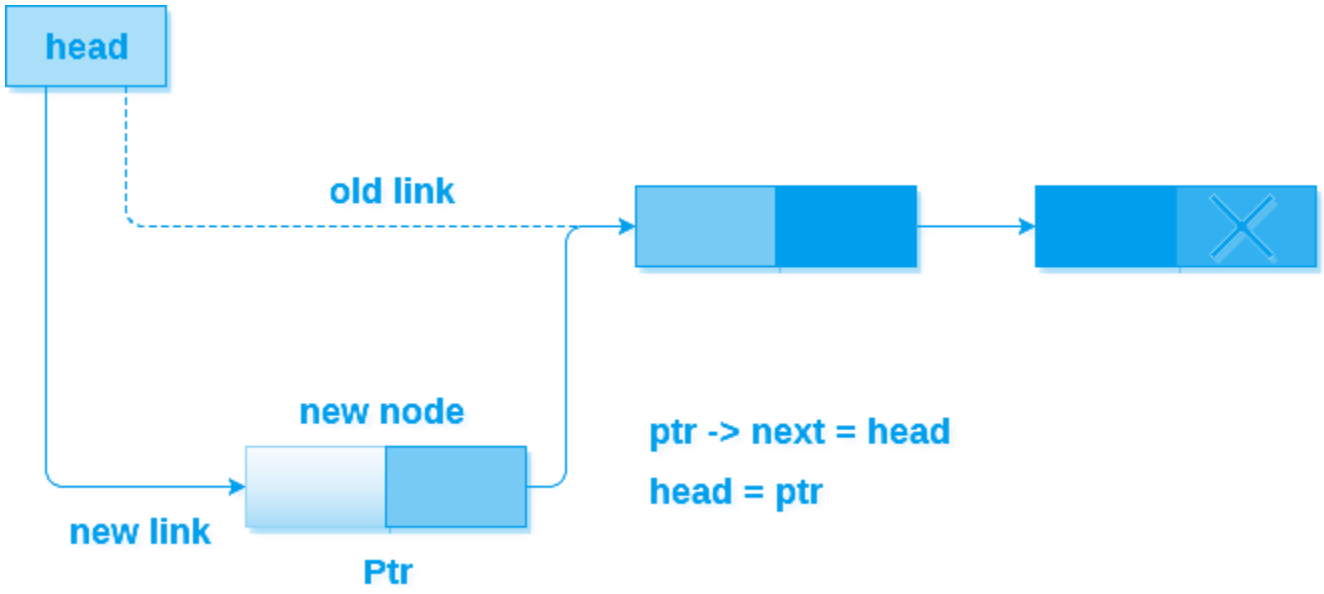
```
0 → 10 → 20 → 30 → 40→ 50
```

It will print the linked list after inserting the node at the beginning of it as
0 → 10 → 20 → 30 → 40→ 50

Algorithm for Insertion at the Beginning of a Singly Linked List

Below mentioned steps can be followed for insertion at the beginning of a singly linked list.

1. Make a new node using the given data.
2. If the linked list's head is null, set the new node as the linked list's head and return.
3. Set the new node's next pointer to the current head of the linked list.
4. Set the linked list's head to point to the new node.
5. Return.



Pseudo Code

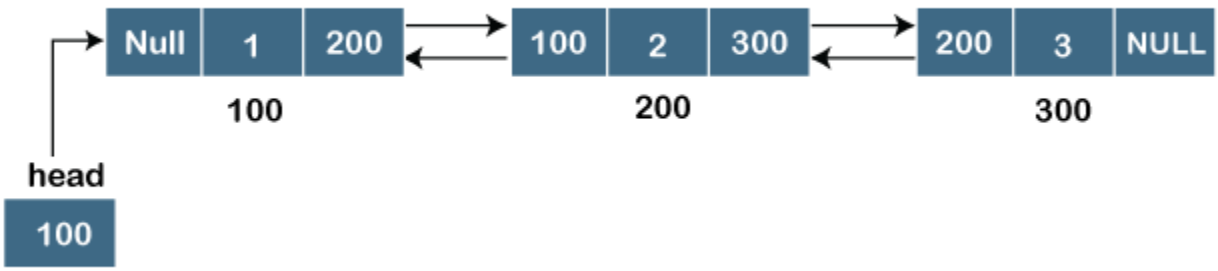
```
function insertAtBeginning(data):

    newNode = Node(data)

    if head is null:
```

```
        head = newNode

        return

    newNode.next = head

    head = newNode

    return
```

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.

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:



As we can observe in the above figure, the node in a doubly-linked list has two address parts; one part stores the ***address of the next*** while the other part of the node stores the ***previous node's address***. The initial node in the doubly linked list has the **NULL** value in the address part, which provides the address of the previous node.

Representation of the node in a doubly linked list

1. struct node

```
{
  int data;
  struct node *next;
 struct node *prev;
}
```

In the above representation, we have defined a user-defined structure named ***a node*** with three members, one is **data** of integer type, and the other two are the pointers, i.e., **next and prev** of the node type. The **next pointer** variable holds the address of the next node, and the **prev pointer** holds the address of the previous node. The type of both the pointers, i.e., **next and prev** is **struct node** as both the pointers are storing the address of the node of the ***struct node*** type.

A doubly linked list is a type of linked list in which each node consists of 3 components:

- ***prev -*** address of the previous node
- **data -** data item
- ***next -*** address of next node

Representation of Doubly Linked List

A doubly linked list is a data structure that consists of nodes, where each node contains a value and two pointers: one to the previous node and one to the next node. This allows for traversal in both directions.
Here is an algorithm for a doubly linked list:

1. Define a Node class with three properties: value, prev (pointer to the previous node), and next (arrow to the next node).
2. Create a DoublyLinkedList class with two properties: head (pointer to the first node) and tail (pointer to the last node).
3. Implement the following methods in the DoublyLinkedList class:
   - isEmpty(): Check if the list is empty by verifying if the head is null.
   - prepend(value): Add a node with the given value at the beginning of the list.

- append(value): Add a node with the given value at the end of the list.
- insertAfter(value, targetValue): Insert a node with the given value after the first occurrence of the target value in the list.
- remove(value): Remove the first occurrence of a node with the given value from the list.
- removeFirst(): Remove the first node from the list.
- removeLast(): Remove the last node from the list.
- printList(): Traverse the list from the head to the tail and print the values of all the nodes.

4. In each method, consider different cases such as adding or removing nodes at the beginning, end, or middle of the list, and update the pointers accordingly

Doubly Linked List in Data Structures Example

```
// Node of a doubly linked list
class Node {
public:
  int data;
  // Pointer to next node in DLL
  Node* next;
  // Pointer to previous node in DLL
  Node* prev;
};
```

In this doubly linked list program in data structure, we create a Node class for a doubly linked list (DLL) with features for data storage, references to the following and preceding nodes, and more.

Basic Operations on Doubly Linked List

Insertion Operation:

- **Insertion at the beginning:** To add a new node at the beginning of the list, you need to create a new node and update the references accordingly. If the list is empty, the new node will be both the head and tail of the list. Otherwise, you update the next reference of the new node to point to the current head, update the previous reference of the current head to point to the new node, and update the head reference to the new node.
- **Insertion at the end:** To add a new node at the end of the list, you create a new node and update the references accordingly. If the list is empty, the

new node will be both the head and tail of the list. Otherwise, you update the next reference of the current tail to point to the new node, update the previous reference of the new node to point to the current tail, and update the tail reference to the new node.

- **Insertion at a specific position:** To insert a new node at a specific position in the list, you need to traverse the list until you reach the desired position. Once you reach the position, you create a new node and update the references accordingly. You update the next reference of the new node to point to the node currently at that position, update the previous reference of the new node to point to the node before the position, update the next reference of the node before the position to point to the new node and update the previous reference of the node at the position to point to the new node.

Algorithm for insertion operation

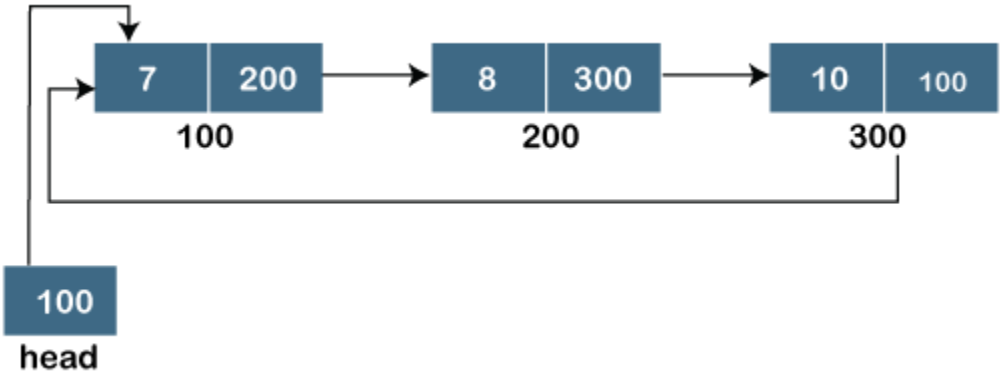
1. Create a new node with the given data.
2. If the doubly linked list is empty, set the head and tail pointers to point to the new node, and make the new node's prev and next pointers null.
3. If the doubly linked list is not empty, do the following:
   - Set the new node's next pointer to point to the current head of the list.
   - Set the new node's prev pointer to null.
   - Set the current head's prev pointer to point to the new node.
   - Set the head pointer to point to the new node.
4. The insertion operation is complete.

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:

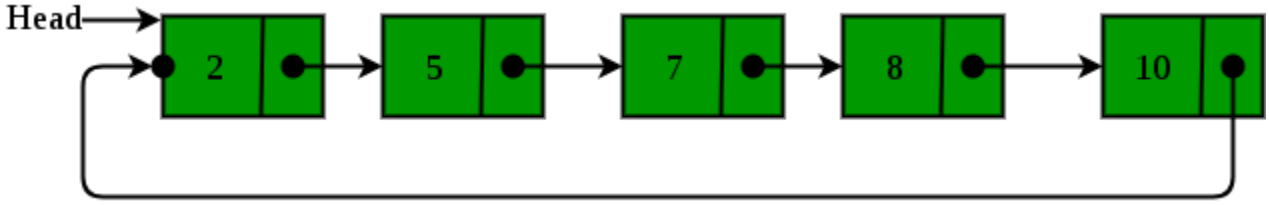
```
1. struct node
2. {
3.    int data;
4.    struct node *next;
5. }
```

A circular linked list is a sequence of elements in which each node has a link to the next node, and the last node is having a link to the first node. The representation of the circular linked list will be similar to the singly linked list, as shown below:



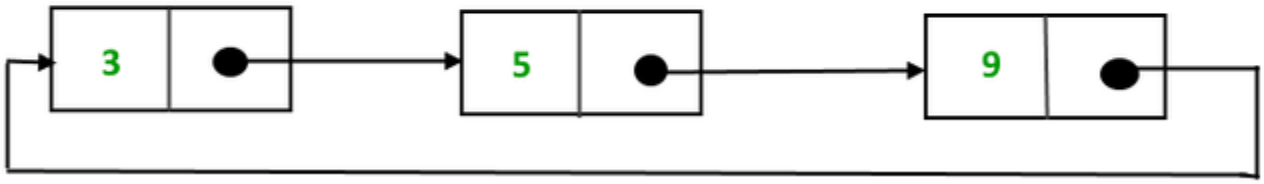
Circular linked list?

*The **circular linked list** is a linked list where all nodes are connected to form a circle. In a circular linked list, the first node and the last node are connected to each other which forms a circle. There is no NULL at the end.*



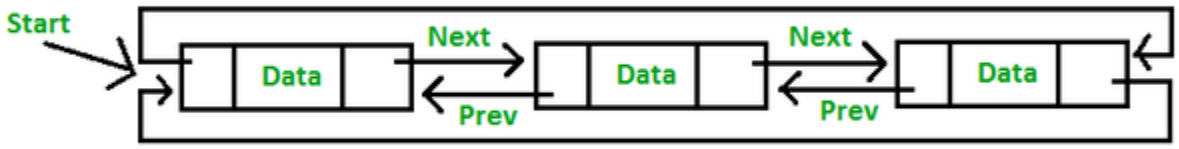
There are generally two types of circular linked lists:

- **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 traverse the circular singly linked list until we reach the same node where we started. The circular singly linked list has no beginning or end. No null value is present in the next part of any of the nodes.



Representation of Circular singly linked list

- **Circular Doubly linked list:** Circular Doubly Linked List has properties of both doubly linked list and circular linked list in which two consecutive elements are linked or connected by the previous and next pointer and the last node points to the first node by the next pointer and also the first node points to the last node by the previous pointer.



Representation of circular doubly linked list

Note: We will be using the singly circular linked list to represent the working of the circular linked list.

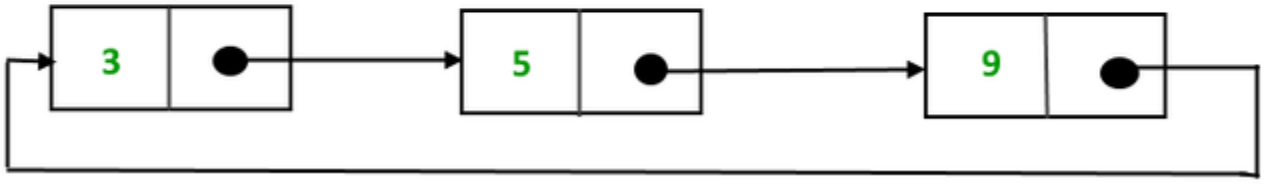
Representation of circular linked list:

Circular linked lists are similar to single Linked Lists with the exception of connecting the last node to the first node.

- Node representation of a Circular Linked List:

```
// Class Node, similar to the linked list

class Node{

    int value;



// Points to the next node.

    Node next;

}
```

Example of Circular singly linked list:



Example of circular linked list

The above Circular singly linked list can be represented as:

```
// Initialize the Nodes.

Node one = new Node(3);

Node two = new Node(5);

Node three = new Node(9);



// Connect nodes

one.next = two;

two.next = three;

three.next = one;
```

Explanation: In the above program one, two, and three are the node with values 3, 5, and 9 respectively which are connected in a circular manner as:

- **For Node One:** The Next pointer stores the address of Node two.
- **For Node Two:** The Next stores the address of Node three

- **For Node Three:** The Next points to node one.

Operations on the circular linked list:

We can do some operations on the circular linked list similar to the singly linked list which are:

1. Insertion
2. Deletion

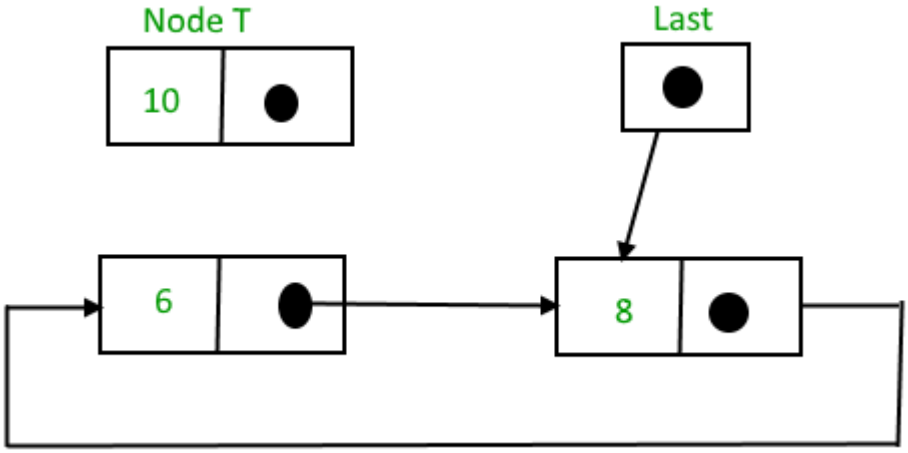
1. Insertion in the circular linked list:

A node can be added in three ways:

1. Insertion at the beginning of the list
2. Insertion at the end of the list
3. Insertion in between the nodes

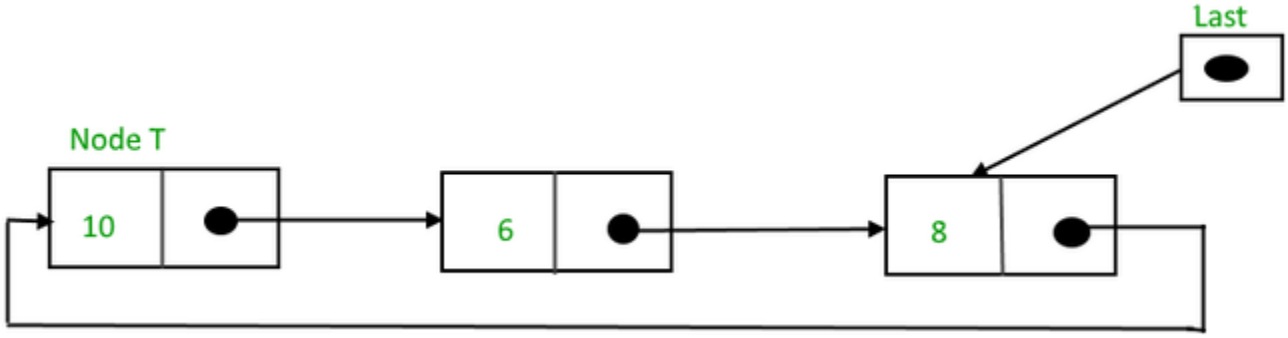
1) **Insertion at the beginning of the list:** To insert a node at the beginning of the list, follow these steps:

- Create a node, say T.
- Make T -> next = last -> next.
- last -> next = T.



Circular linked list before insertion

And then,

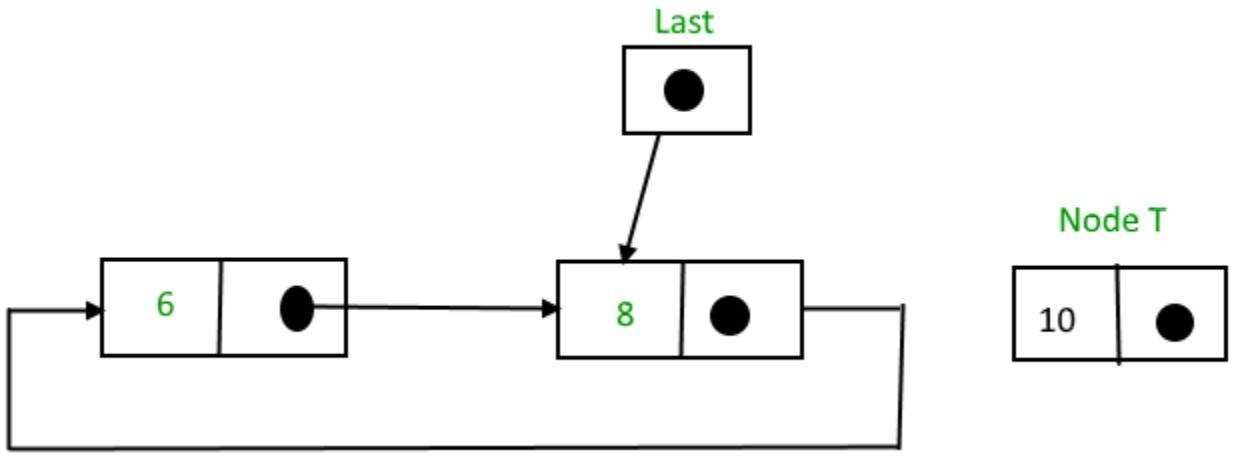


Circular linked list after insertion

2) **Insertion at the end of the list:** To insert a node at the end of the list, follow these steps:

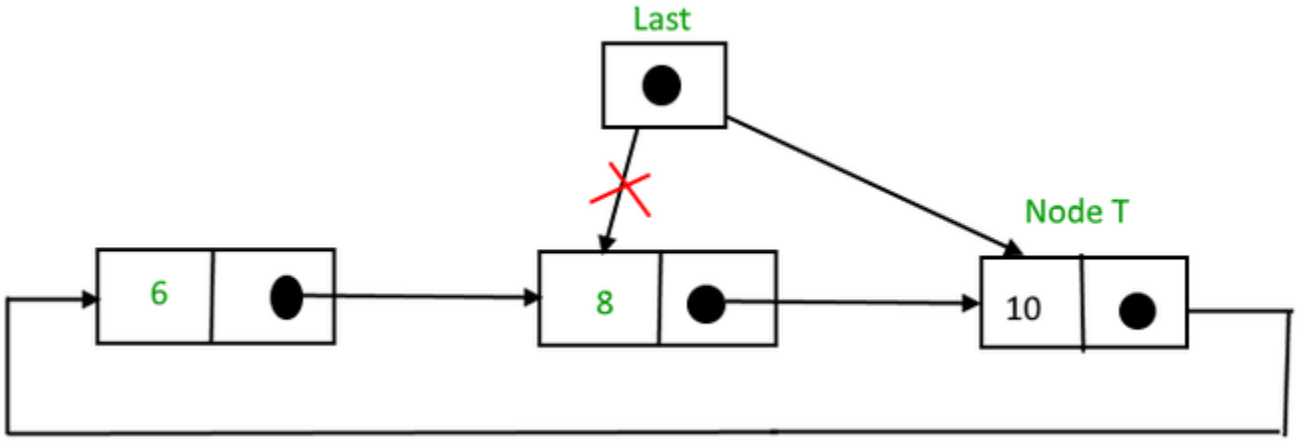
- Create a node, say T.
- Make T -> next = last -> next;
- last -> next = T.
- last = T.

Before insertion,



Circular linked list before insertion of node at the end

After insertion,



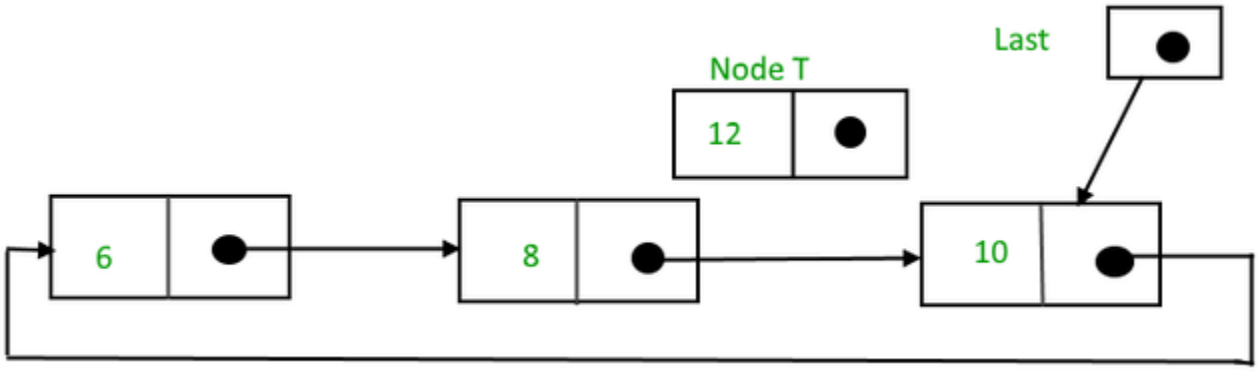
Circular linked list after insertion of node at the end

3) **Insertion in between the nodes:** To insert a node in between the two nodes, follow these steps:

- Create a node, say T.
- Search for the node after which T needs to be inserted, say that node is P.
- Make T -> next = P -> next;

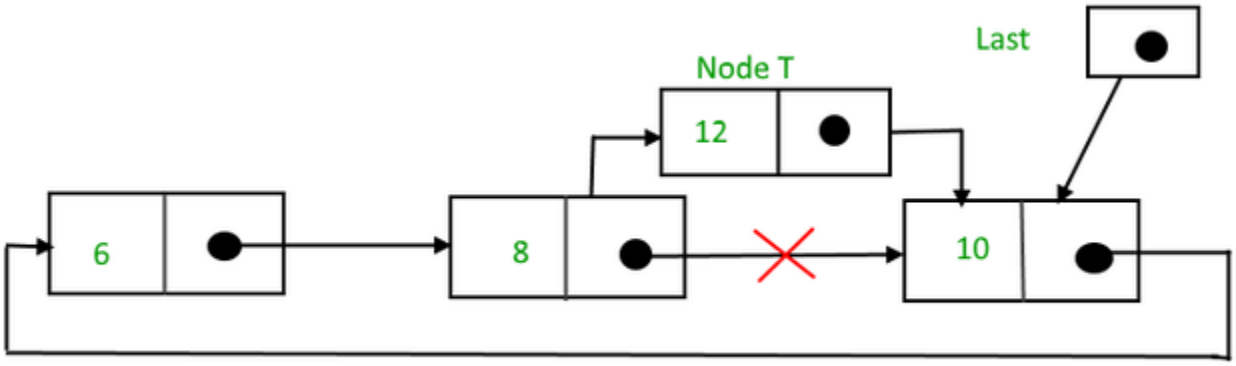
- P -> next = T.

Suppose 12 needs to be inserted after the node has the value 10,



Circular linked list before insertion

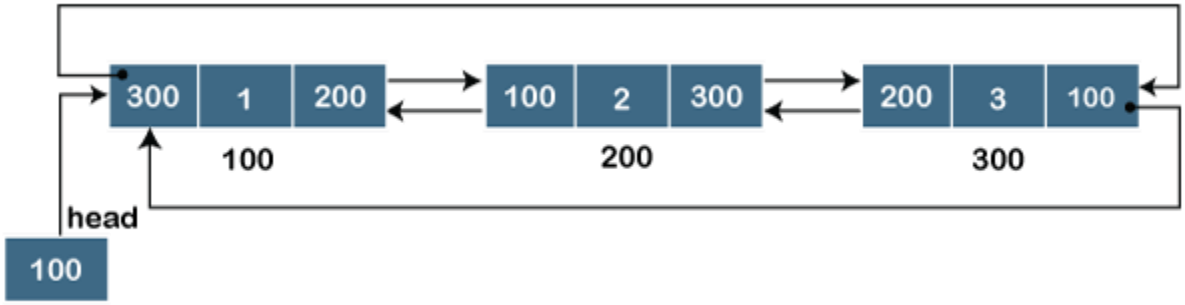
After searching and insertion,



Circular linked list after insertion

Doubly Circular linked list

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



The above figure shows the representation of the doubly circular linked list in which the last node is attached to the first node and thus creates a circle. It is a doubly linked list also because each node holds the address of the previous node also. The main difference between the doubly linked list and doubly circular linked list is that the doubly circular linked list does not contain the NULL value in the previous field of the node. As the doubly circular linked contains three parts, i.e., two address parts and one data part so its representation is similar to the doubly linked list.

```
1. struct node
2. {
3.   int data;
4.   struct node *next;
5.  struct node *prev;
6. }
```

v

Array vs Linked List

Array and **Linked list** are the two ways of organizing the data in the memory. Before understanding the differences between the **Array** and the **Linked List**, we first look **at an array** and **a linked list**.

What is an array?

An array is a data structure that contains the elements of the same type. A data structure is a way of organizing the data; an array is a data structure because it sequentially organizes the data. An array is a big chunk of memory in which memory is divided into small-small blocks, and each block is capable of storing some value.

Suppose we have created an array that consists of 10 values, then each block will store the value of an integer type. If we try to store the value in an array of different types, then it is not a correct array and will throw a compile-time error.

Declaration of array

An array can be declared as:

```
data_type name of the array[no of elements]
```

To declare an array, we first need to specify the type of the array and then the array's name. Inside the square brackets, we need to specify the number of elements that our array should contain.

Let's understand through an example.

```
1. int a[5];
```

In the above case, we have declared an array of 5 elements with '**a**' name of an **integer** data type.

What is Linked list?

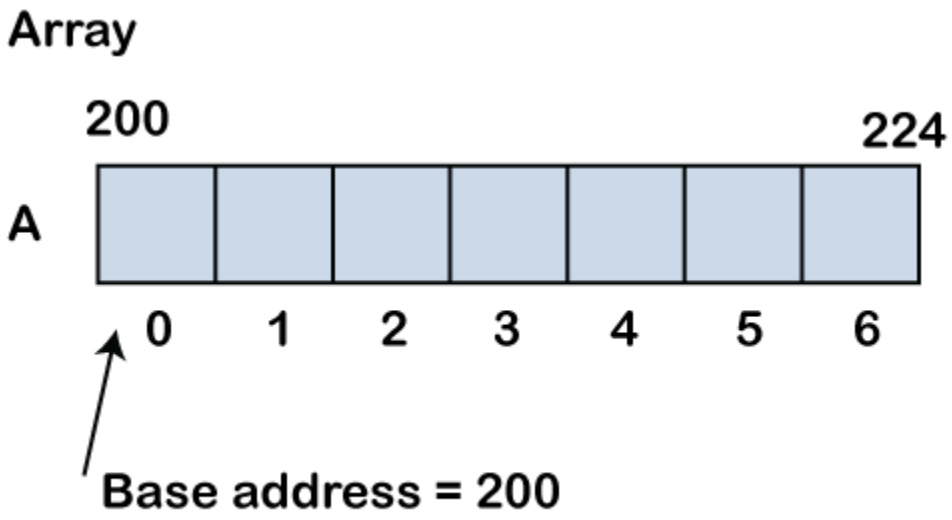
A linked list is the collection of nodes that are randomly stored. Each node consists of two fields, i.e., **data** and **link**. Here, data is the value stored at that particular node, and the link is the pointer that holds the address of the next node.

Differences between Array and Linked list

We cannot say which data structure is better, i.e., array or linked list. There can be a possibility that one data structure is better for one kind of requirement, while the other data structure is better for another kind of requirement. There are various factors like what are the frequent operations performed on the data structure or the size of the data, and other factors also on which basis the data structure is selected. Now we will see some differences between the array and the linked list based on some parameters.

1. Cost of accessing an element

In case of an array, irrespective of the size of an array, an array takes a constant time for accessing an element. In an array, the elements are stored in a contiguous manner, so if we know the base address of the element, then we can easily get the address of any element in an array. We need to perform a simple calculation to obtain the address of any element in an array. So, accessing the element in an array is **O(1)** in terms of time complexity.



In the linked list, the elements are not stored in a contiguous manner. It consists of multiple blocks, and each block is represented as a node. Each node has two fields, i.e.,

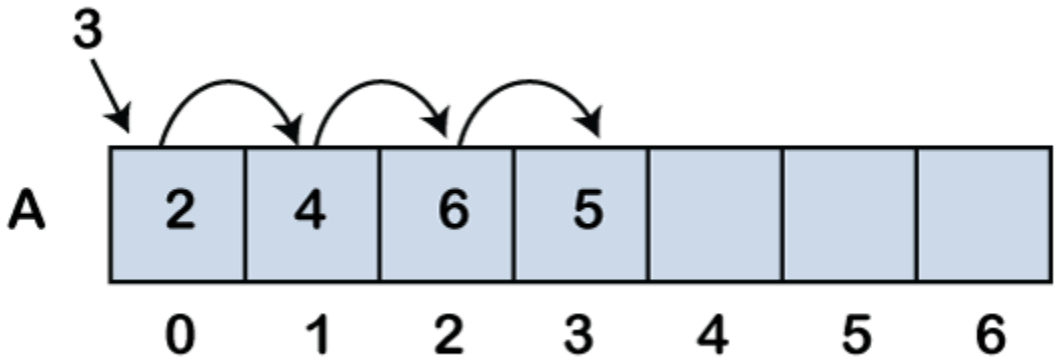
one is for the data field, and another one stores the address of the next node. To find any node in the linked list, we first need to determine the first node known as the head node. If we have to find the second node in the list, then we need to traverse from the first node, and in the worst case, to find the last node, we will be traversing all the nodes. The average case for accessing the element is O(n).

We conclude that the cost of accessing an element in array is less than the linked list. Therefore, if we have any requirement for accessing the elements, then array is a better choice.

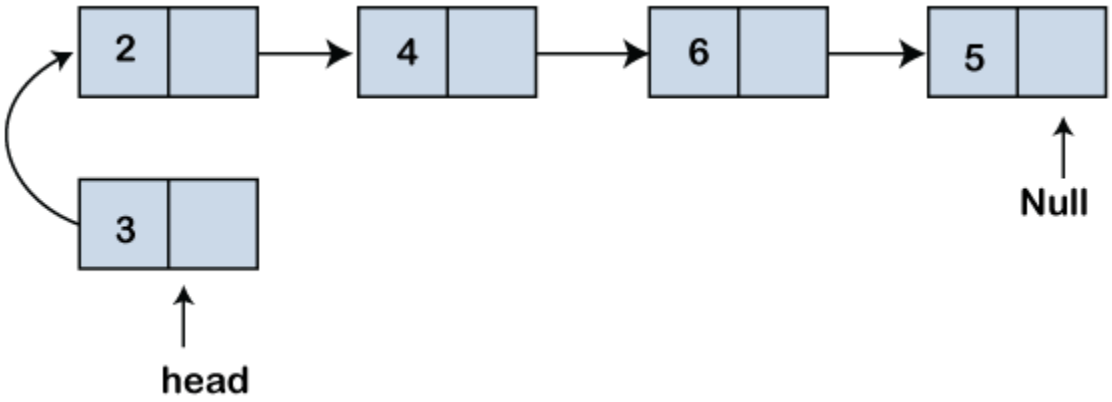
2. Cost of inserting an element

There can be three scenarios in the insertion:

- **Inserting the element at the beginning:** To insert the new element at the beginning, we first need to shift the element towards the right to create a space in the first position. So, the time complexity will be proportional to the size of the list. If n is the size of the array, the time complexity would be O(n).



In the case of a linked list, to insert an element at the starting of the linked list, we will create a new node, and the address of the first node is added to the new node. In this way, the new node becomes the first node. So, the time complexity is not proportional to the size of the list. The time complexity would be constant, i.e., O(1).



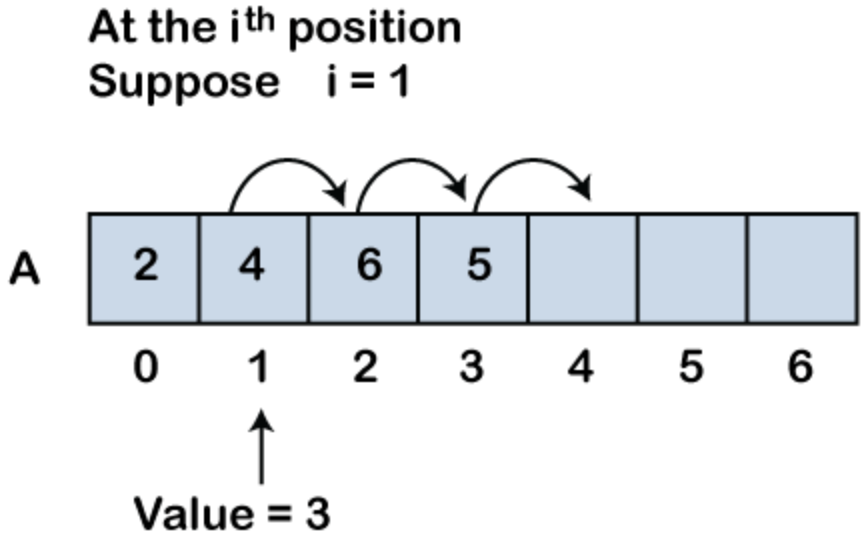
- **Inserting an element at the end**

If the array is not full, then we can directly add the new element through the index. In this case, the time complexity would be constant, i.e., O(1). If the array is full, we first need to copy the array into another array and add a new element. In this case, the time complexity would be O(n).

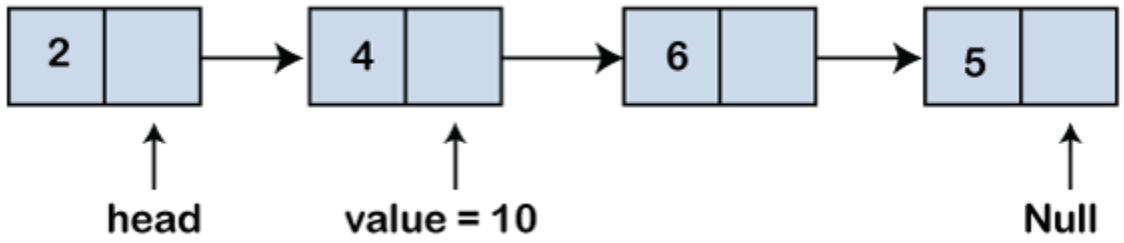
To insert an element at the end of the linked list, we have to traverse the whole list. If the linked list consists of n elements, then the time complexity would be O(n).

- **Inserting an element at the mid**

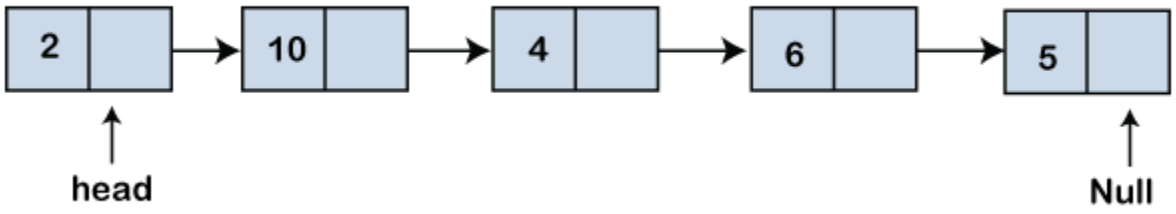
Suppose we want to insert the element at the i^{th} position of the array; we need to shift the n/2 elements towards the right. Therefore, the time complexity is proportional to the number of the elements. The time complexity would be O(n) for the average case.



In the case of linked list, we have to traverse to that position where we have to insert the new element. Even though, we do not have to perform any kind of shifting, but we have to traverse to n/2 position. The time taken is proportional to the n number of elements, and the time complexity for the average case would be O(n).

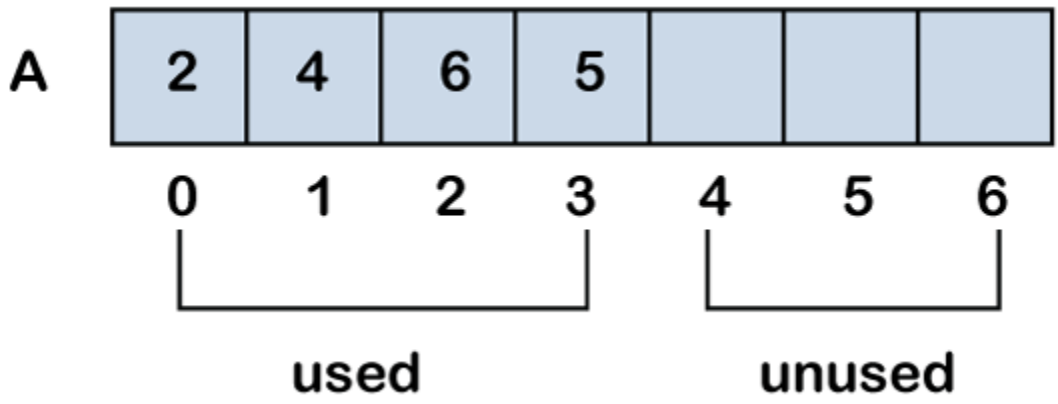


The resultant linked list is:



3. Memory requirements

As the elements in an array store in one contiguous block of memory, so array is of fixed size. Suppose we have an array of size 7, and the array consists of 4 elements then the rest of the space is unused. The memory occupied by the 7 elements:



Memory space = 7*4 = 28 bytes

Where 7 is the number of elements in an array and 4 is the number of bytes of an integer type.

In case of linked list, there is no unused memory but the extra memory is occupied by the pointer variables. If the data is of integer type, then total memory occupied by one node is 8 bytes, i.e., 4 bytes for data and 4 bytes for pointer variable. If the linked list consists of 4 elements, then the memory space occupied by the linked list would be:

Memory space = 8*4 = 32 bytes

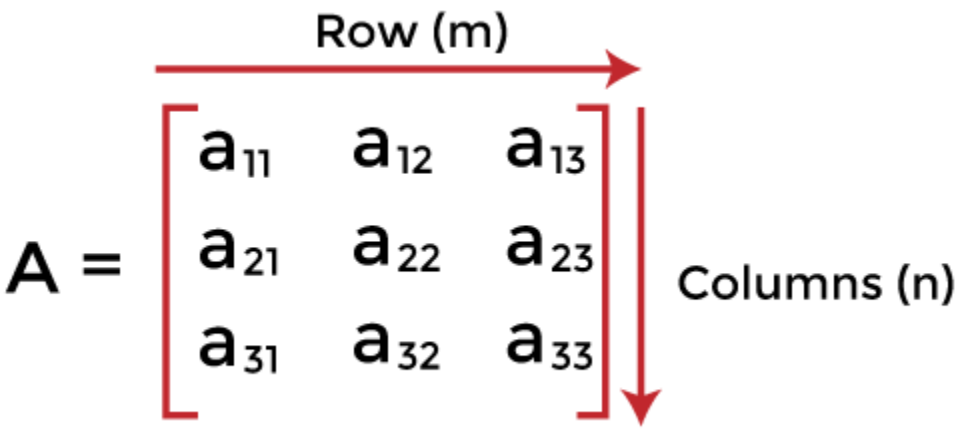
The linked list would be a better choice if the data part is larger in size. Suppose the data is of 16 bytes. The memory space occupied by the array would be 16*7=112 bytes while the linked list occupies 20*4=80, here we have specified 20 bytes as 16 bytes for the size of the data plus 4 bytes for the pointer variable. If we are choosing the larger size of data, then the linked list would consume a less memory; otherwise, it depends on the factors that we are adopting to determine the size.

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.

For example -



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.

Now, the question arises: we can also use the simple matrix to store the elements, then why is the sparse matrix required?

Why is a sparse matrix required if we can use the simple matrix to store elements?

There are the following benefits of using the sparse matrix -

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

Array representation of the sparse matrix

Representing a sparse matrix by a 2D array leads to the wastage of lots of memory. This is because zeroes in the matrix are of no use, so storing zeroes with non-zero elements is wastage of memory. To avoid such wastage, we can store only non-zero elements. If we store only non-zero elements, it reduces the traversal time and the storage space.

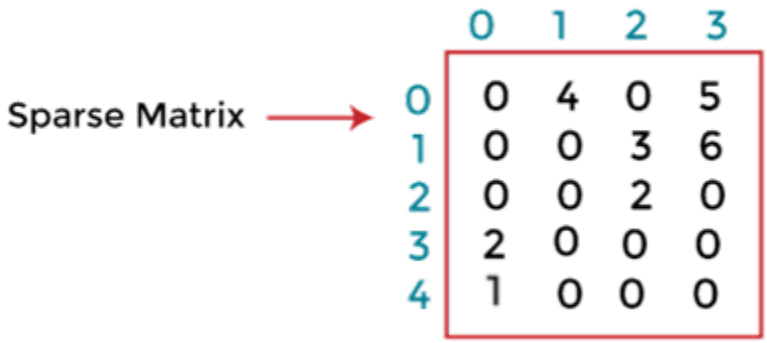
In 2D array representation of sparse matrix, there are three fields used that are named as -

- **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 -



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. So, what's the benefit of using 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.

Linked List representation of the sparse matrix

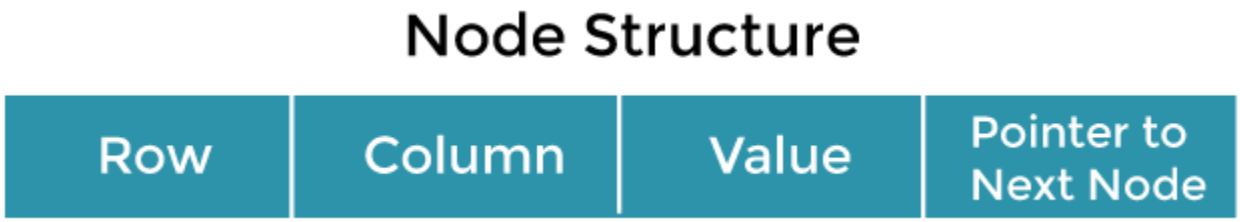
In a linked list representation, the linked list data structure is used to represent the sparse matrix. The advantage of using a linked list to represent the sparse matrix is that the complexity of inserting or deleting a node in a linked list is lesser than the array.

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.

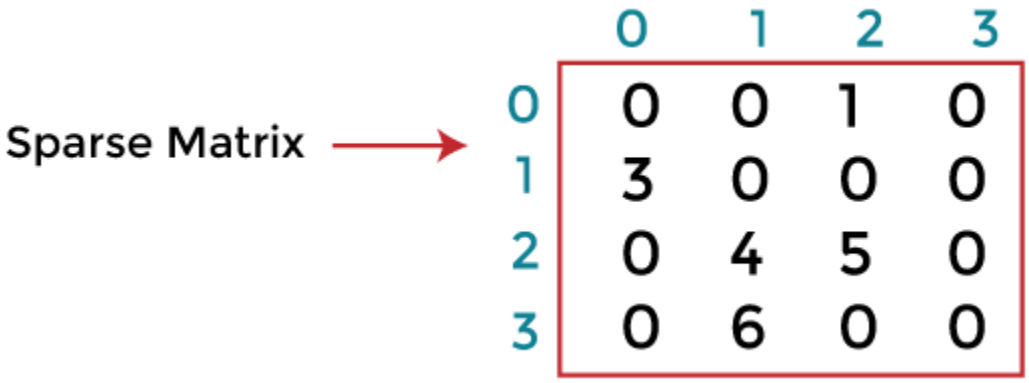
The node structure of the linked list representation of the sparse matrix is shown in the below image -



Example -

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

Consider the sparse matrix -



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 0^{th} row, the second field contains 2, which means 2^{nd} 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 0^{th} row-2^{nd} column in the given sparse matrix. In a similar manner, all of the nodes represent the non-zero elements of the sparse matrix.

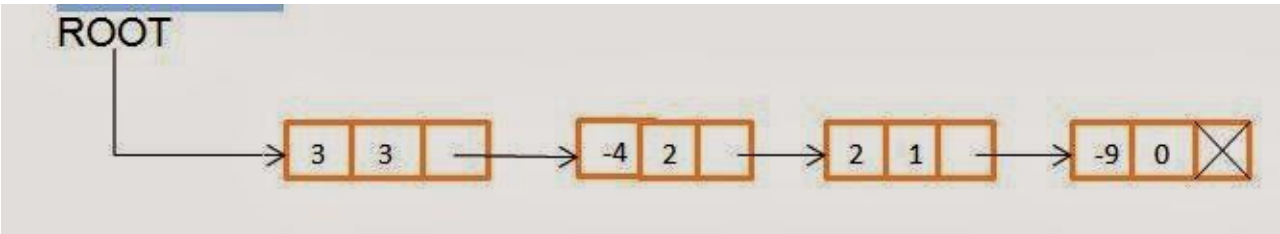
Polynomial representation using Linked List

The linked list can be used to represent a polynomial of any degree. Simply the information field is changed according to the number of variables used in the polynomial. If a single variable is used in the polynomial the information field of the node contains two parts: one for coefficient of variable and the other for degree of variable. Let us consider an example to

represent a polynomial using linked list as follows:

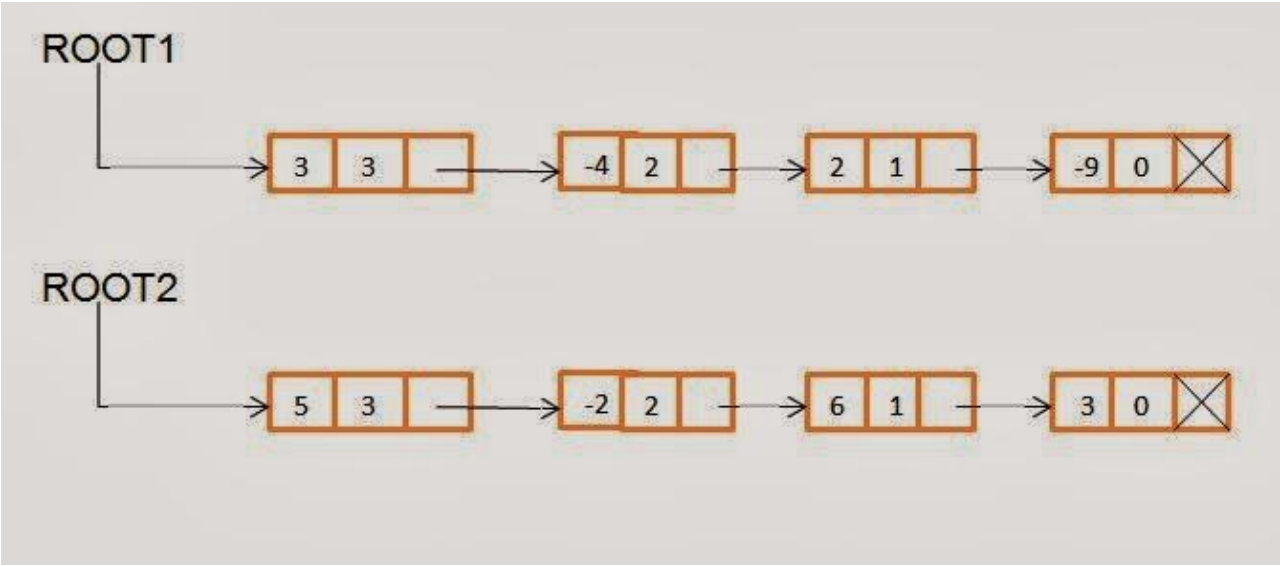
Polynomial: $3x^3-4x^2+2x-9$

Linked List:

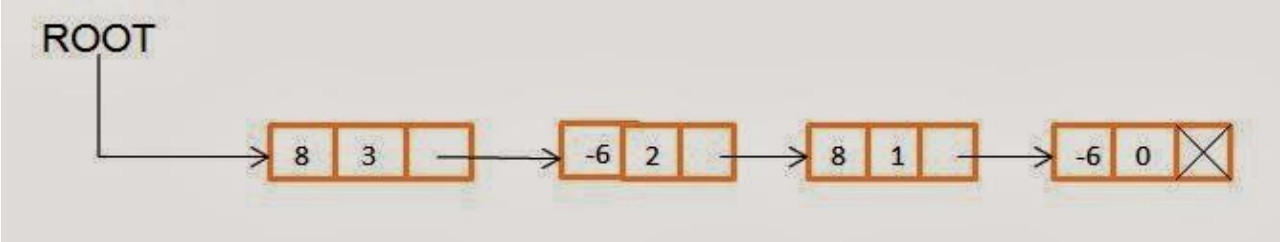


In the above linked list, the external pointer ‘ROOT’ point to the first node of the linked list. The first node of the linked list contains the information about the variable with the highest degree. The first node points to the next node with next lowest degree of the variable.

Representation of a polynomial using the linked list is beneficial when the operations on the polynomial like addition and subtractions are performed. The resulting polynomial can also be traversed very easily to display the polynomial.



The above two linked lists represent the polynomials, $3x^3-4x^2+2x-9$ and $5x^3-2x^2+6x+3$ respectively. If both the polynomials are added then the resulting linked will be:



The linked list pointer ROOT gives the representation for polynomial, $8x^3-6x^2+8x-6$.