UNIT -III

Single Linked List

The disadvantages of arrays are:

• The size of the array is fixed. Most often this size is specified at compile time. This makes the programmers to allocate arrays, which seems "large enough" than required.

•Inserting new elements at the front is potentially expensive because existing elements need to be shifted over to make room.

• Deleting an element from an array is not possible. Linked lists have their own strengths and weaknesses, but they happen to be strong where arrays are weak. Generally array's allocates the memory for all its elements in one block whereas linked lists use

• An entirely different strategy. Linked lists allocate memory for each element separately and only when necessary.

Advantages of linked lists:

Linked lists have many advantages. Some of the very important advantages are:

1. Linked lists are dynamic data structures. i.e., they can grow or shrink during the execution of a program.

2. Linked lists have efficient memory utilization. Here, memory is not preallocated. Memory is allocated whenever it is required and it is de-allocated (removed) when it is no longer needed.

3. Insertion and Deletions are easier and efficient. Linked lists provide flexibility in inserting a data item at a specified position and deletion of the data item from the given position.

4. Many complex applications can be easily carried out with linked lists.

Disadvantages of linked lists:

1. It consumes more space because every node requires a additional pointer to store address of the next node.

2. Searching a particular element in list is difficult and also time consuming.

Types of Linked Lists:

Basically we can put linked lists into the following four items:

1. Single Linked List.

2. Double Linked List.

3. Circular Linked List.

4. Circular Double Linked List.

Comparison between array and linked list:

ARRAY	LINKED LIST
Size of an array is fixed	Size of a list is not fixed
Memory is allocated from stack	Memory is allocated from heap
It is necessary to specify the number of elements during declaration (i.e., during compile time).	It is not necessary to specify the number of elements during declaration (i.e., memory is allocated during run time).
It occupies less memory than a linked list for the same number of elements.	It occupies more memory.
Inserting new elements at the front is potentially expensive because existing elements need to be shifted over to make room.	Inserting a new element at any position can be carried out easily.
Deleting an element from an array is not possible.	Deleting an element is possible.

Trade offs between linked lists and arrays:

FEATURE	ARRAYS	LINKED LISTS
Sequential access	efficient	efficient
Random access	efficient	inefficient
Resigning	inefficient	efficient
Element rearranging	inefficient	efficient
Overhead per elements	none	1 or 2 links

What is Linked List?

When we want to work with unknown number of data values, we use a linked list data structure to organize that data. Linked list is a linear data structure that contains sequence of elements such that each element links to its next element in the sequence. Each element in a linked list is called as "Node".

What is Single Linked List?

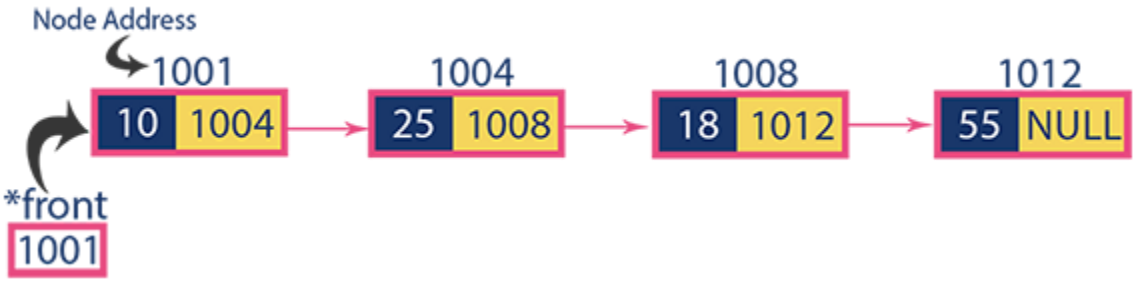
Simply a list is a sequence of data, and linked list is a sequence of data linked with each other. The formal definition of a single linked list is as follows...

Single linked list is a sequence of elements in which every element has link to its next element in the sequence.

In any single linked list, the individual element is called as "Node". Every "Node" contains two fields, data field and next field. The data field is used to store actual value of the node and next field is used to store the address of next node in the sequence.

The graphical representation of node in a single linked list is as follows...

Example



Operations on Single Linked List

The following operations are performed on a Single Linked List

- Insertion
- Deletion
- Display

Before we implement actual operations, first we need to setup empty list. First perform the following steps before implementing actual operations.

- **Step 1 -** Include all the **header files** which are used in the program.
- **Step 2 -** Declare all the **user defined functions**.
- **Step 3 -** Define a **Node** structure with two members **data** and **next**
- **Step 4 -** Define a Node pointer **'start'** and set it to **NULL**.
- **Step 5 -** Implement the main method by displaying operations menu and make suitable function calls in the main method to perform user selected operation.

Insertion

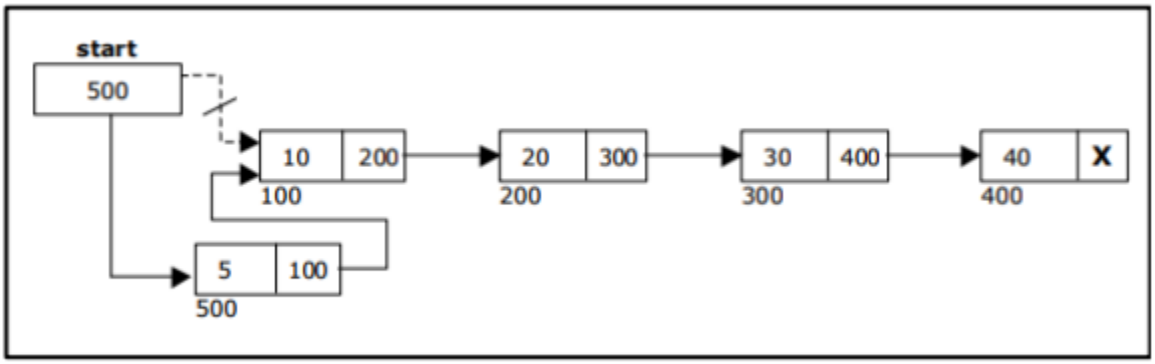
In a single linked list, the insertion operation can be performed in three ways. They are as follows...

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list

Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the single linked list...

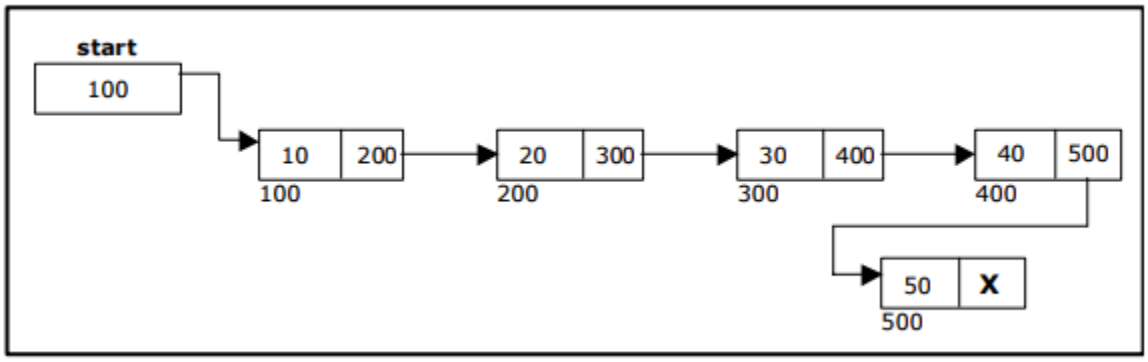
- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**start** = = **NULL**)
- **Step 3 -** If it is **Empty** then, set **newNode→next** = **NULL** and **start** = **newNode**.
- **Step 4 -** If it is **Not Empty** then, set **newNode→next** = **start** and **start**= **newNode**.



Inserting At End of the list

We can use the following steps to insert a new node at end of the single linked list...

- **Step 1 -** Create a **newNode** with given value and **newNode** → **next** as **NULL**.
- **Step 2 -** Check whether list is **Empty** (**start** = = **NULL**).
- **Step 3 -** If it is **Empty** then, set **start** = **newNode**.
- **Step 4 -** If it is **Not Empty** then, define a node pointer **temp** and initialize with **start**.
- **Step 5 -** Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp** → **next** is not equal to **NULL**).
- **Step 6 -** Set **temp** → **next** = **newNode**.



Inserting at middle of the list

Step 1 - Create a newNode with given value

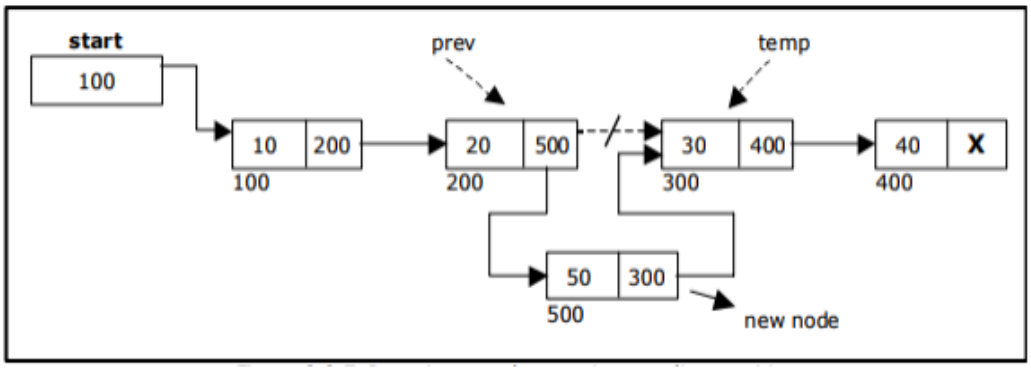
Step 2 - Check whether list is **Empty** (**start** == **NULL**)

Step 3 - If it is **Empty** then, set **newNode** → **next** = **NULL** and **start** = **newNode**.

Step 4 - If it is **Not Empty** then, define a node pointer **temp** and initialize with **start**.

Step 5 - Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the newNode

Step 6-Finally, Set p->next=newnode, newnode->next=temp



Deletion

In a single linked list, the deletion operation can be performed in three ways. They are as follows...

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the single linked list...

Step 1 - Check whether list is **Empty** (**start** == **NULL**)

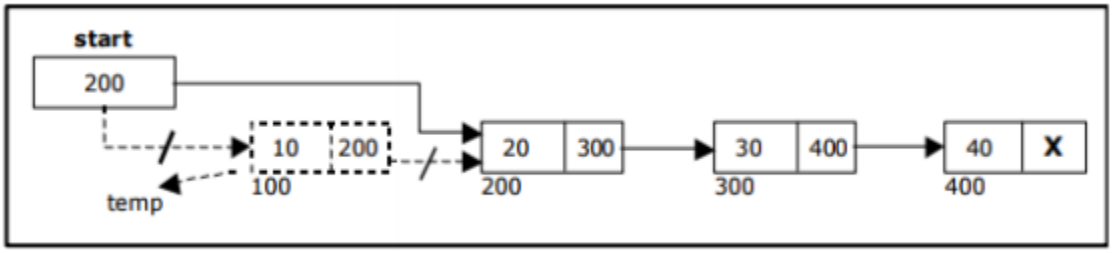
Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.

Step 3 - If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **start**.

Step 4 - Check whether list is having only one node (**temp → next** == **NULL**)

Step 5 - If it is **TRUE** then set **start**= **NULL** and delete **temp** (Setting **Empty** list conditions)

Step 6 - If it is **FALSE** then set **start**= **temp → next**, and delete **temp(free(temp))**.



Deleting from End of the list

We can use the following steps to delete a node from end of the single linked list...

Step 1 - Check whether list is **Empty** (**start** == **NULL**)

Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.

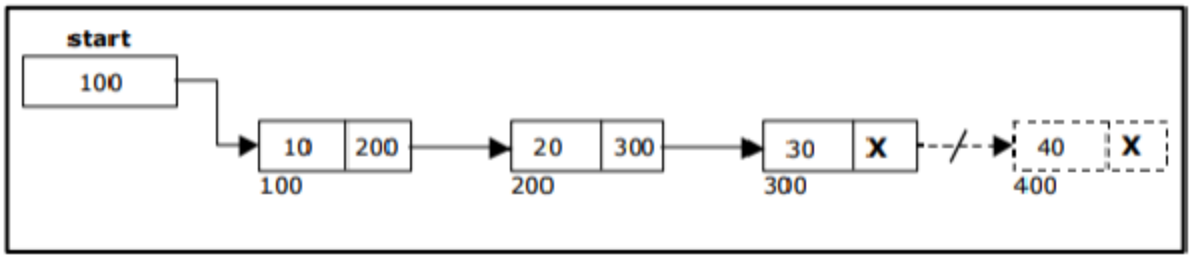
Step 3 - If it is **Not Empty** then, define two Node pointers **'temp'** and **'temp1'** and initialize **'temp'** with **start**.

Step 4 - Check whether list has only one Node (**temp → next** == **NULL**)

Step 5 - If it is **TRUE**. Then, set **start** = **NULL** and delete **temp**. And terminate the function. (Setting **Empty** list condition)

Step 6 - If it is **FALSE**. Then, set '**temp1 = temp** ' and move **temp** to its next node. Repeat the same until it reaches to the last node in the list. (until **temp1 → next** == **NULL**)

Step 7 - Finally, Set **temp1 → next** = **NULL** and delete **temp(free(temp))**.



Deleting a Specific Node from the list

Step 1 - Create a newNode with given value

Step 2 - Check whether list is **Empty** (**start** == **NULL**)

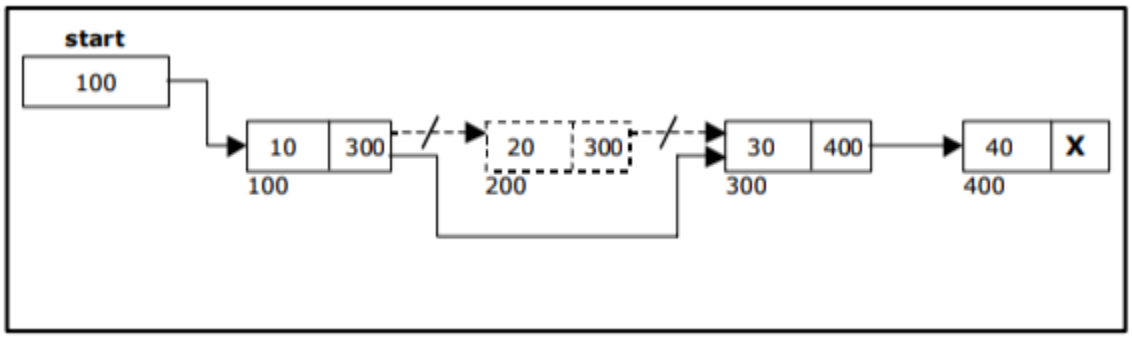
Step 3 - If it is **Empty** then, set **newNode** → **next** = **NULL** and **start** = **newNode**.

Step 4 - If it is **Not Empty** then, define a node pointer **temp** and initialize with **start**.

Step 5 - Keep moving the **temp** to its next node until it reaches to the node after which we want to delete the Node

Step 6-Finally, Set p->next=temp->next

Step 7-delete temp(free(temp)).



Traversing

- Assign the address of start pointer to a temp pointer.
- Display the information from the data field of each node.
- The function traverse () is used for traversing and displaying the information stored in the list from left to right.

Applications of Linked Lists

Polynomials and Sparse Matrix are two important applications of arrays and linked lists.

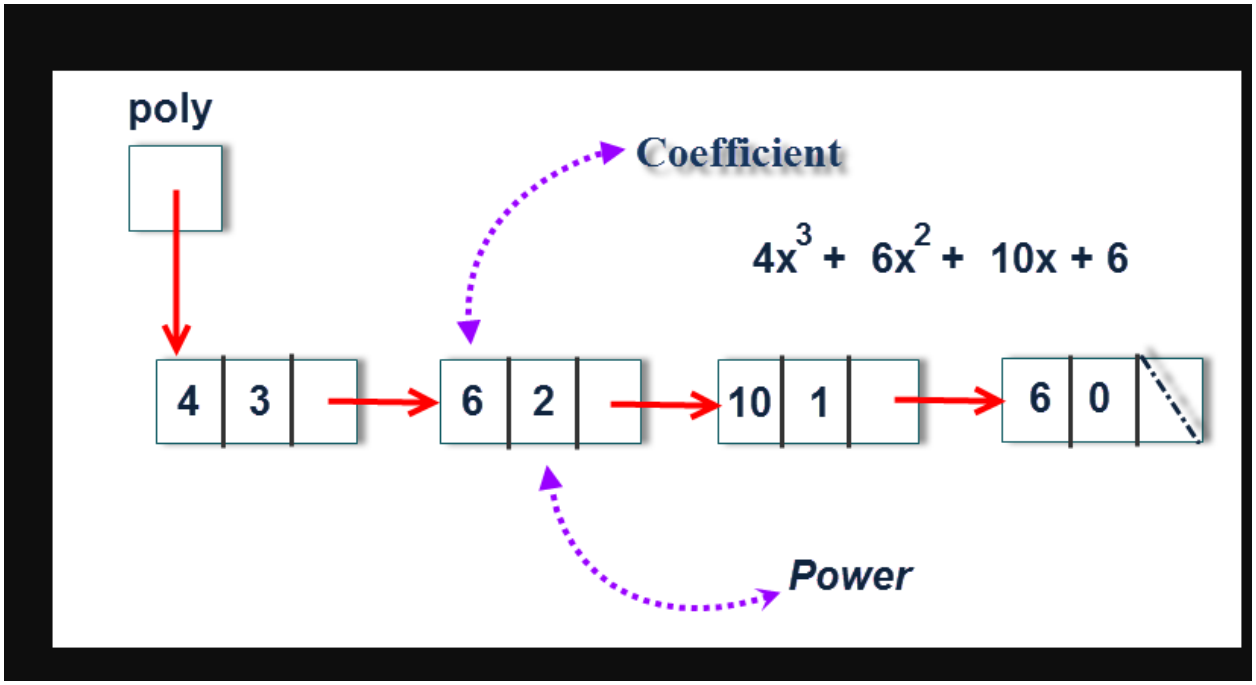
A polynomial is composed of different terms where each of them holds a coefficient and an exponent.

Polynomial Representation:

A polynomial may also be represented using a linked list. A structure may be defined such that it contains two parts- one is the coefficient and second is the corresponding exponent. The structure definition may be given as shown below:

```
struct polynomial
{
int coefficient;
int exponent;
struct polynomial *next;
};
```

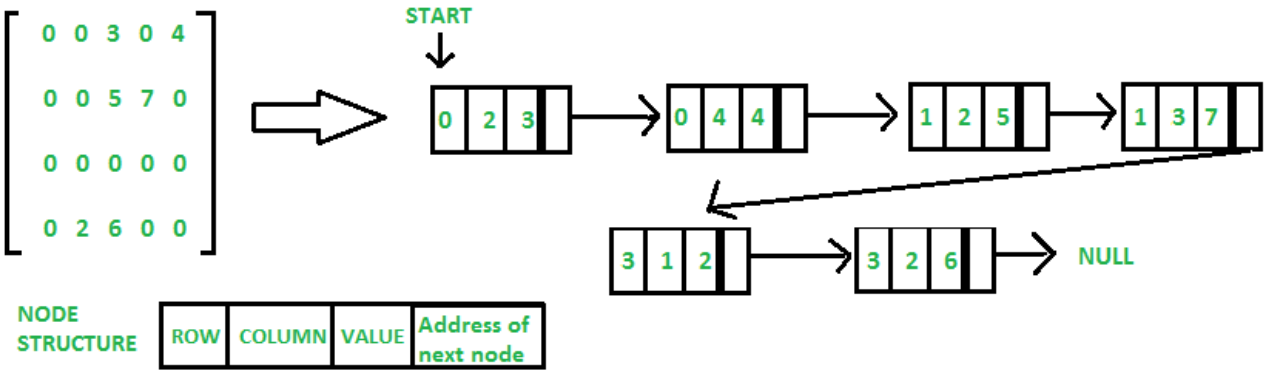
Thus the above polynomial may be represented using linked list as shown below:



Sparse Matrix Representation

In linked list, each node has four fields. These four fields are defined as:

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



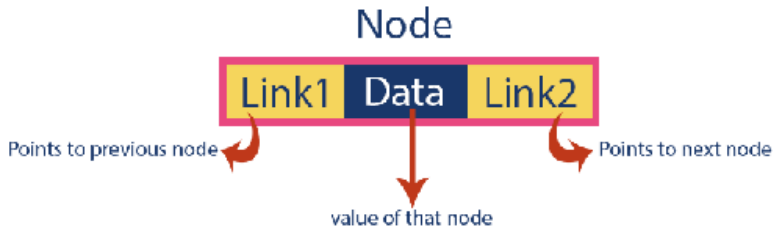
Double Linked List

What is Double Linked List?

In a single linked list, every node has link to its next node in the sequence. So, we can traverse from one node to other node only in one direction and we cannot traverse back. We can solve this kind of problem by using double linked list. Double linked list can be defined as follows...

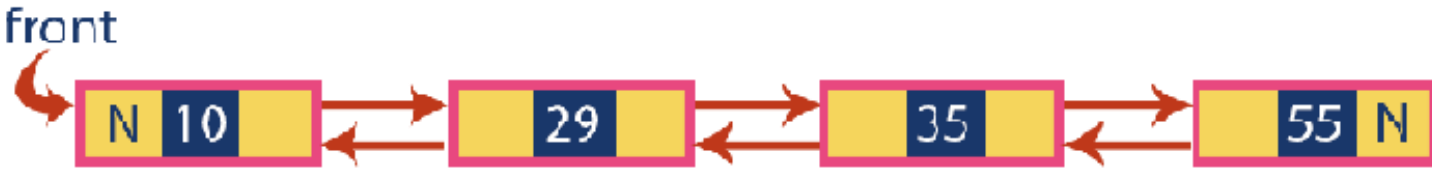
Double linked list is a sequence of elements in which every element has links to its previous element and next element in the sequence.

In double linked list, every node has link to its previous node and next node. So, we can traverse forward by using next field and can traverse backward by using previous field. Every node in a double linked list contains three fields and they are shown in the following figure...



Here, '**link1**' field is used to store the address of the previous node in the sequence, '**link2**' field is used to store the address of the next node in the sequence and '**data**' field is used to store the actual value of that node.

Example



Operations on Double Linked List

In a double linked list, we perform the following operations...

1. Insertion
2. Deletion
3. Display

Insertion

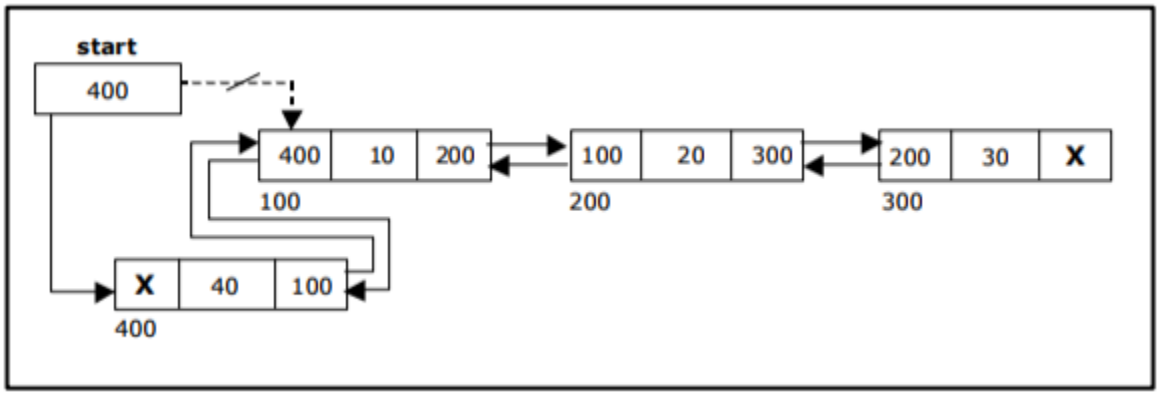
In a double linked list, the insertion operation can be performed in three ways as follows...

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list

Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the double linked list...

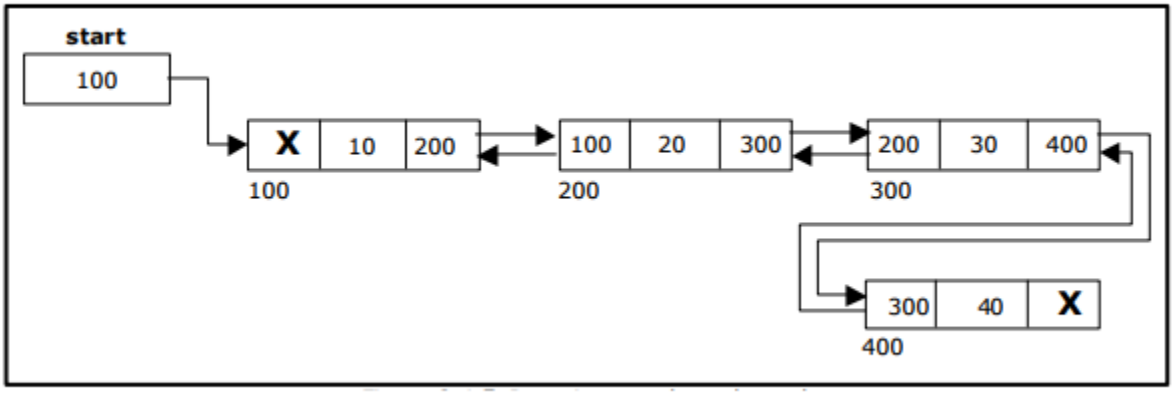
- **Step 1 -** Create a **newNode** with given value and **newNode → previous** as **NULL**.
- **Step 2 -** Check whether list is **Empty** (**start** = = **NULL**)
- **Step 3 -** If it is **Empty** then, assign **NULL** to **newNode → next** and **newNode** to **start**.
- **Step 4 -** If it is **not Empty** then, assign **start** to **newNode → next** and **newNode** to **start**.



Inserting At End of the list

We can use the following steps to insert a new node at end of the double linked list...

- **Step 1 -** Create a **newNode** with given value and **newNode → next** as **NULL**.
- **Step 2 -** Check whether list is **Empty** (**start** = = **NULL**)
- **Step3-** If it is **Empty**, then assign **NULL** to **newNode→previous** and **newNode** to **head**.
- **Step 4 -** If it is **not Empty**, then, define a node pointer **temp** and initialize with **head**.
- **Step 5 -** Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp → next** is equal to **NULL**).
- **Step 6 -** Assign **newNode** to **temp → next** and **temp** to **newNode → previous**.



Inserting At Specific location in the list (After a Node)

We can use the following steps to insert a new node after a node in the double linked list...

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3 -** If it is **Empty** then, assign **NULL** to both **newNode → previous**& **newNode → next** and set **newNode** to **head**.
- **Step 4 -** If it is **not Empty** then, define two node pointers **temp1** & **temp2** and initialize **temp1** with **head**.
- **Step 5 -** Keep moving the **temp1** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1 → data** is equal to **location**, here location is the node value after which we want to insert the newNode).
- **Step 6 -** Every time check whether **temp1** is reached to the last node. If it is reached to the last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp1** to next node.
- **Step 7 -** Assign **temp1 → next** to **temp2**, **newNode** to **temp1 → next**, **temp1** to **newNode → previous**, **temp2** to **newNode → next** and **newNode** to **temp2 → previous**.