**Linked List**

Linked list consists of nodes where each node contains a data field and a reference (link) to the next node in the list. Linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers.

**Linked List Representation**



**Features of Linked List**

1. Linked list consists of nodes; each node contains two fields – data field and link field. Data field contain the data item and link field contains the address of the next node.
2. The address of the first node is stored in the START or also known as START
3. The link field of the last node contains null to mark the end of the list

### Advantages of Linked List

1. **Dynamic Data Structure**

Linked list is a dynamic data structure so it can grow and shrink at runtime by allocating and deallocating memeory. So there is no need to give initial size of linked list.

1. **Insertion and Deletion**

Insertion and deletion of nodes are really easier. Unlike array here we don’t have to shift elements after insertion or deletion of an element. In linked list we just have to update the address present in next pointer of a node.

1. **No Memory Wastage**

As size of linked list can increase or decrease at run time so there is no memory wastage. In case of array there is lot of memory wastage, like if we declare an array of size 10 and store only 6 elements in it then space of 4 elements are wasted. There is no such problem in linked list as memory is allocated only when required.

1. **Implementation**

Data structures such as stack and queues can be easily implemented using linked list.

### Disadvantages of Linked List

1. **Memory Usage**

More memory is required to store elements in linked list as compared to array. Because in linked list each node contains a pointer and it requires extra memory for itself.

1. **Traversal**

Elements or nodes traversal is difficult in linked list. We can not randomly access any element as we do in array by index. For example if we want to access a node at position n then we have to traverse all the nodes before it. So, time required to access a node is large.

1. **Reverse Traversing**

In linked list reverse traversing is really difficult. In case of doubly linked list its easier but extra memory is required for back pointer hence wastage of memory.

**Types of Linked List**

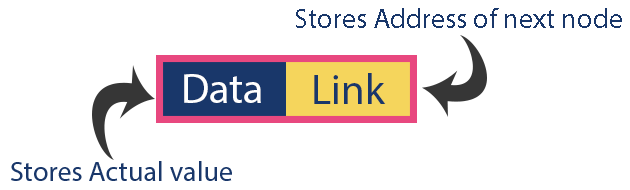
* **Singly Linked List** − Item navigation is forward only.
* **Doubly Linked List** − Items can be navigated forward and backward.
* **Circular Linked List** − Last item contains link of the first element as next and the first element has a link to the last element as previous.

**Operations of Linked List**

* **Insertion** − Adds an element at the beginning of the list, between two nodes and at the end of the node.
* **Deletion** − Deletes an element at the beginning of the list, between two nodes and at the end of the node.
* **Display** − Displays the complete list.
* **Search** − Searches an element using the given key.
* **Traverse** – Reads each element in the list.

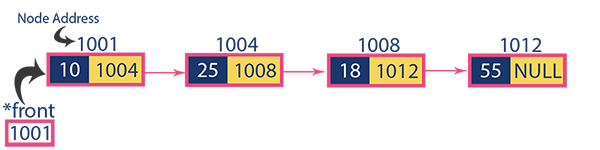
**Singly Linked List**

* Singly linked list is a sequence of elements in which every element has link to its next element in the sequence.
* In any singly linked list, the individual element is called as "Node". Every "Node" contains two fields, data and next. The data field is used to store actual value of that node and next field is used to store the address of the next node in the sequence.
* The graphical representation of a node in a singly linked list is as follows...



* In a singly linked list, the address of the first node is always stored in a reference node known as "front" (Sometimes it is also known as "START" or START).
* Always next part (reference part) of the last node must be NULL.

**Example of singly linked list**



**Operations of singly linked list**

In a single linked list we perform the following operations...

* Insertion
* Deletion
* Display
* **Node in singly linked list is represented as**

struct node

{ int info;

struct node \*link;

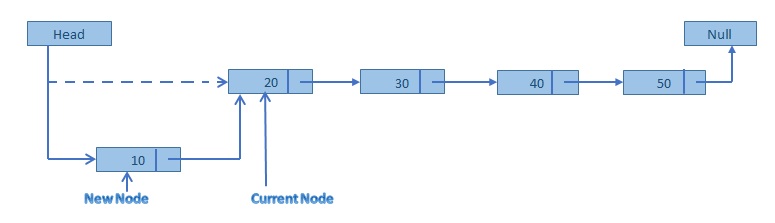
};

**Insertion**

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

* Inserting new node at Beginning of the list
* Inserting new node at End of the list
* Inserting new node 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.

In this function temp refers to newNode.

**Function for inserting new node at Beginning of the list**

struct node \*beg(struct node \*start, int data)

{

struct node \*temp;

temp = (struct node \*) malloc(sizeof(struct node));

temp -> info = data;

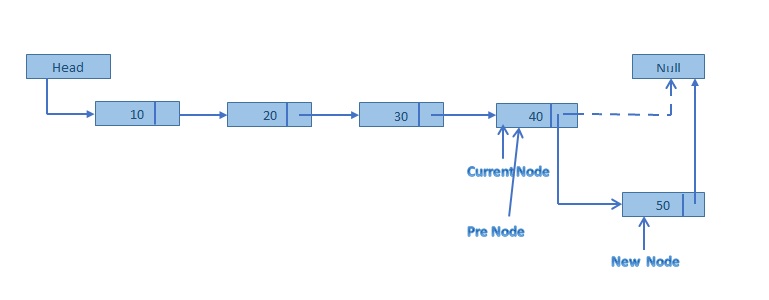
temp -> link = start;

start = temp;

return start;

}

**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 \*p and initialize with START.

Step 5: Keep moving ‘p’ to its next node until it reaches to the last node in the list (until p → next is equal to NULL).

Step 6: Set p → next = newNode.

In this function temp refers to newNode.

**Function for inserting new node at the end of the list**

struct node \*end(struct node \*start, int data)

{

struct node \*p, \*temp;

temp = (struct node \*) malloc(sizeof(struct node));

temp -> info = data;

p= start;

while( p -> link != NULL)

{

p= p -> link;

}

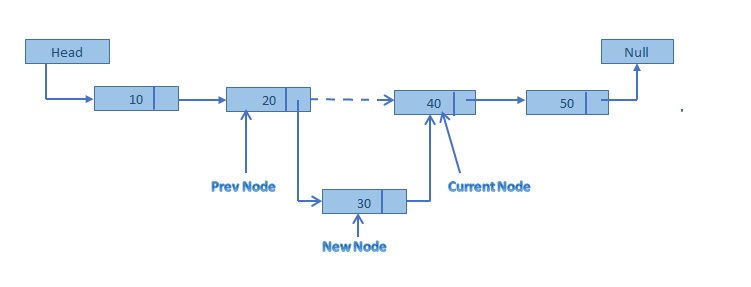
p -> link = temp;

temp -> link = NULL;

return start;

}

**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 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, define a node pointer ‘p’ and initialize with START.

Step 5: Keep moving ‘p’ to its next node until it reaches to the node after which we want to insert the newNode (until 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 ‘p’ is reached to last node or not. If it is reached to last node then display 'Given node is not found in the list!!! Insertion not possible!!!' and terminate the function. Otherwise move the temp to next node.

Step 7: Finally, Set 'newNode → next = p → next' and 'p → next = newNode'

In this function temp refers to newNode.

**Function for inserting new node at the specific position of the list**

struct node \*npostion(struct node \*start, int data, int position )

{

int i;

struct node \*temp, \*p;

temp = (struct node \*) malloc(sizeof(struct node));

temp -> info = data;

if(position ==1)

{

temp -> link = start;

start = temp;

return start;

}

p = start;

for(i=1; i< position -1 && p != NULL; i++)

{

p = p -> link;

}

if(p == NULL)

{

printf("There are less than %d elements \n", position);

}

else

{

temp -> link = p -> link;

p -> link = temp;

}

return start;

}

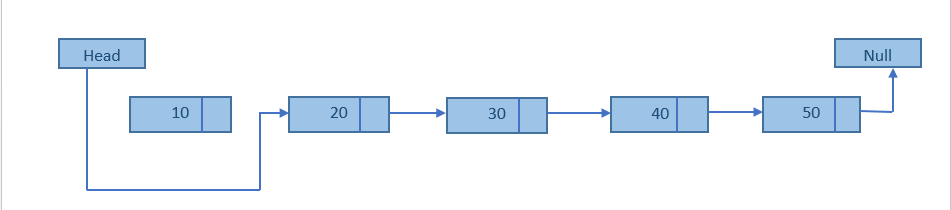
**Deletion**

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

* Deleting from Beginning of the list
* Deleting from End of the list
* 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.

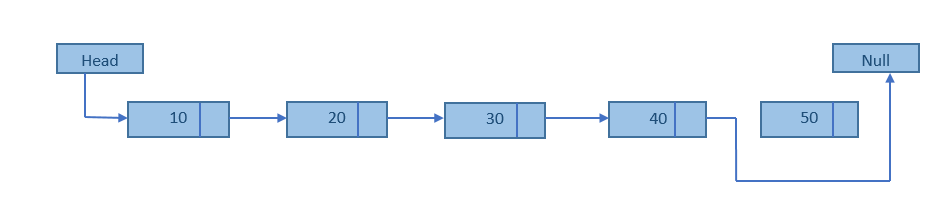
Step 4: Check whether list is having only one node (temp → next == NULL)

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.

**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 'temp1' and 'temp2' and initialize 'temp1' with START.

Step 4: Check whether list has only one Node (temp1 → next == NULL)

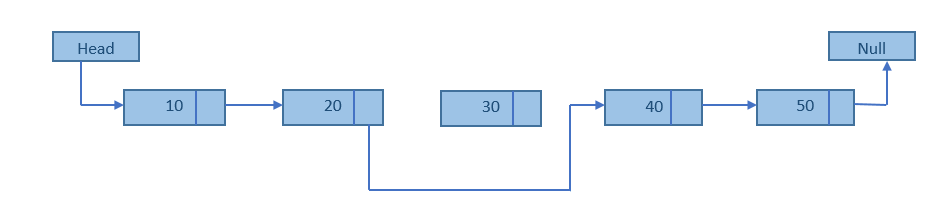
Step 5: If it is TRUE. Then, set START = NULL and delete temp1. And terminate the function. (Setting Empty list condition)

Step 6: If it is FALSE. Then, set 'temp2 = temp1 ' and move temp1 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 temp2 → next = NULL and delete temp1.

**Deleting a Specific Node from the list**

We can use the following steps to delete a specific node from 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 'temp1' and 'temp2' and initialize 'temp1' with START.

Step 4: Keep moving the temp1 until it reaches to the exact node to be deleted or to the last node. And every time set 'temp2 = temp1' before moving the 'temp1' to its next node.

Step 5: If it is reached to the last node then display 'Given node not found in the list! Deletion not possible!!!'. And terminate the function.

Step 6: If it is reached to the exact node which we want to delete, then check whether list is having only one node or not

Step 7: If list has only one node and that is the node to be deleted, then set START = NULL and delete temp1 (free (temp1)).

Step 8: If list contains multiple nodes, then check whether temp1 is the first node in the list (temp1 == START).

Step 9: If temp1 is the first node then move the START to the next node (START = START → next) and delete temp1.

Step 10: If temp1 is not first node then check whether it is last node in the list (temp1 → next == NULL).

Step 11: If temp1 is last node then set temp2 → next = NULL and delete temp1 (free(temp1)).

Step 12: If temp1 is not first node and not last node then set temp2 → next = temp1 → next and delete temp1 (free (temp1)).

**Function for deleting a node in the linked list (all three ways)**

struct node \*delete(struct node \*start,int data)

{

struct node \*temp, \*p;

if(start == NULL)

{ printf("empty list");

return start;

}

// deletion of only node

if(start -> info == data)

{

temp = sart;

start = NULL;

free(temp);

return start;

}

// deletion of first node

if(start -> info == data)

{ temp = start;

start = start -> link;

free(temp);

return start;

}

// deletion in between or at the end

p = start;

while( p ->link != NULL)

{

if(p -> link -> info == data)

{ temp = p -> link;

p -> link = temp -> link;

free(temp);

return start;

}

p = p ->link;

}

printf("element not found",data);

return start;

}

**Displaying a Single Linked List**

We can use the following steps to display the elements of a single linked list...



Step 1: Check whether list is Empty (START == NULL)

Step 2: If it is Empty then, display 'List is Empty!!!' and terminate the function.

Step 3: If it is Not Empty then, define a Node pointer 'temp' and initialize with START.

Step 4: Keep displaying temp → data with an arrow (--->) until temp reaches to the last node

Step 5: Finally display temp → data with arrow pointing to NULL (temp → data ---> NULL).

**Function to display a linked list**

void disp(struct node \*start)

{

struct node \*p;

if(start == NULL)

{

printf("List is empty");

}

else

{ p = start;

printf("list \n");

while( p != NULL)

{ printf(" %d" , p -> info);

p = p -> link;

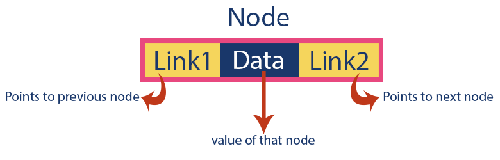
}

}

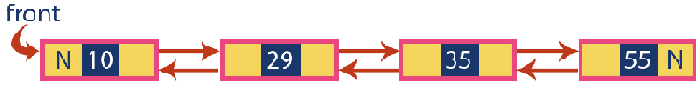
}

**Doubly Linked List**

* 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 – link 1(previous), data, and link2 (next).
* '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.
* In double linked list, the first node must be always pointed by head or START.
* Always the previous field of the first node must be NULL.
* Always the next field of the last node must be NULL.



**Example**



**Operations of singly linked list**

In a single linked list we perform the following operations...

* Insertion
* Deletion
* Display

**Insertion**

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

* Inserting new node at Beginning of the list
* Inserting new node at End of the list
* Inserting new node at Specific location in the list
* **Node in doubly linked list is represented as**

**A node is represented as**

struct node {

int data;

struct node \*next;

struct node \*prev;

}

**Inserting At Beginning of the list**

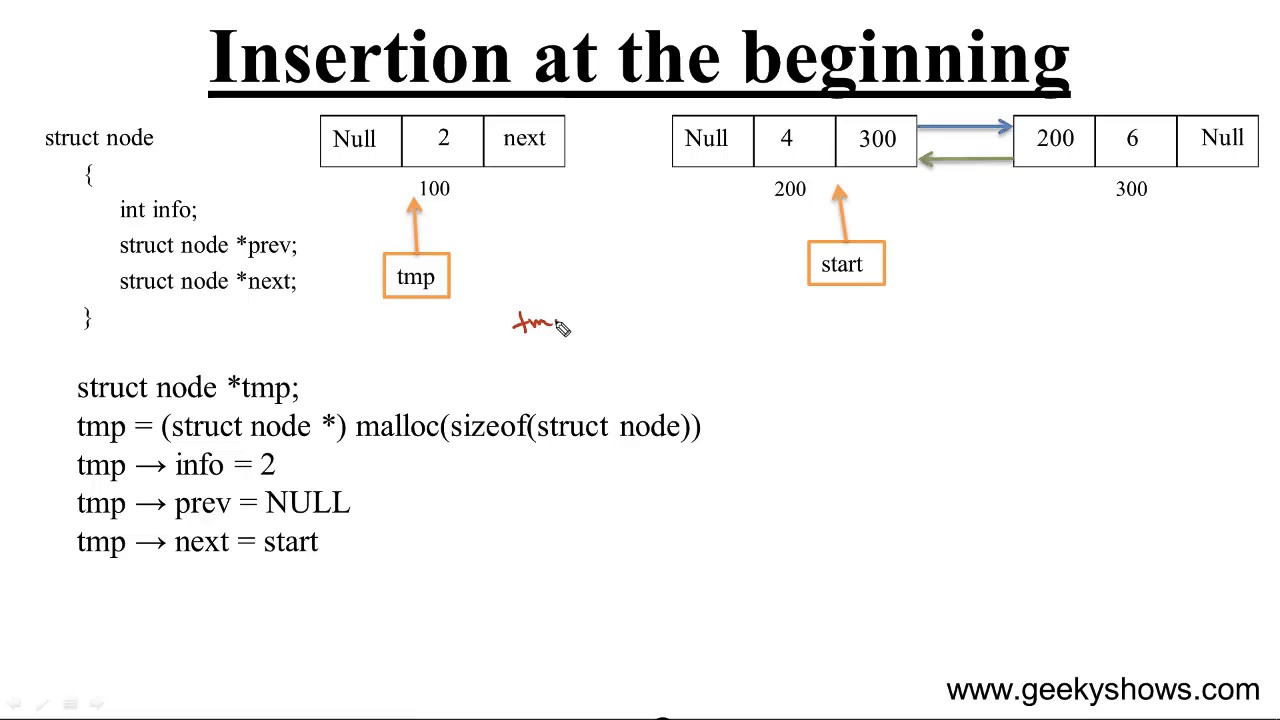
Step 1: Create a newNode with given value and newNode → previous as NULL.

Step 2: Check whether list is Empty (head == NULL)

Step 3: If it is Empty then, assign NULL to newNode → next and newNode to head.

Step 4: If it is not Empty then, assign head to newNode → next and newNode to head.

newNode refers to temp



**Inserting At the end of the list**

Step 1: Create a newNode with given value and newNode → next as NULL.

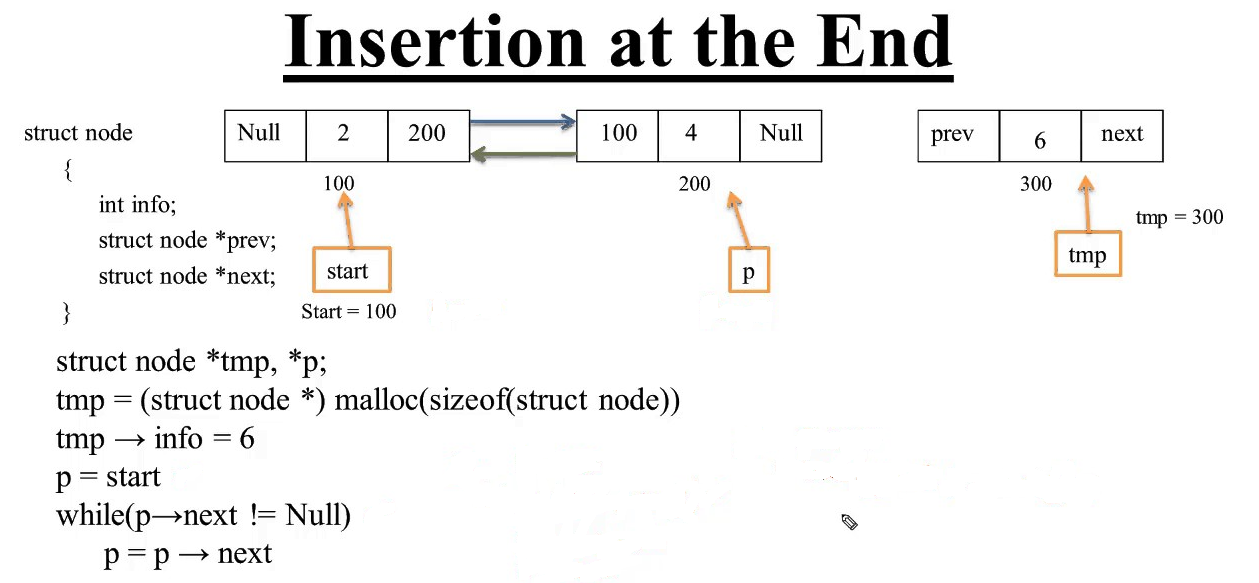
Step 2: Check whether list is Empty (head == NULL)

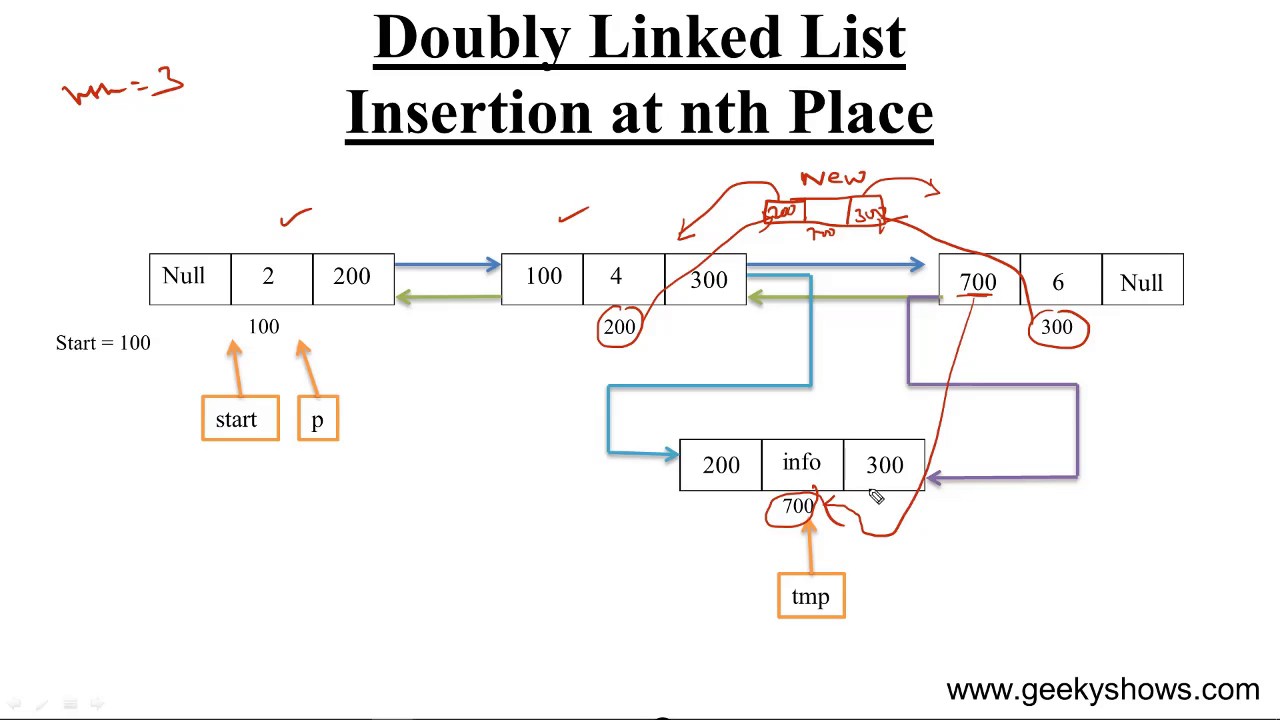
Step 3: If it is Empty, then assign NULL to newNode → previous and newNode to head.

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.





**Deletion**

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

* Deleting from Beginning of the list
* Deleting from End of the list
* Deleting a Specific Node

**Deleting from Beginning of the list**

Step 1: Check whether list is Empty (head == NULL)

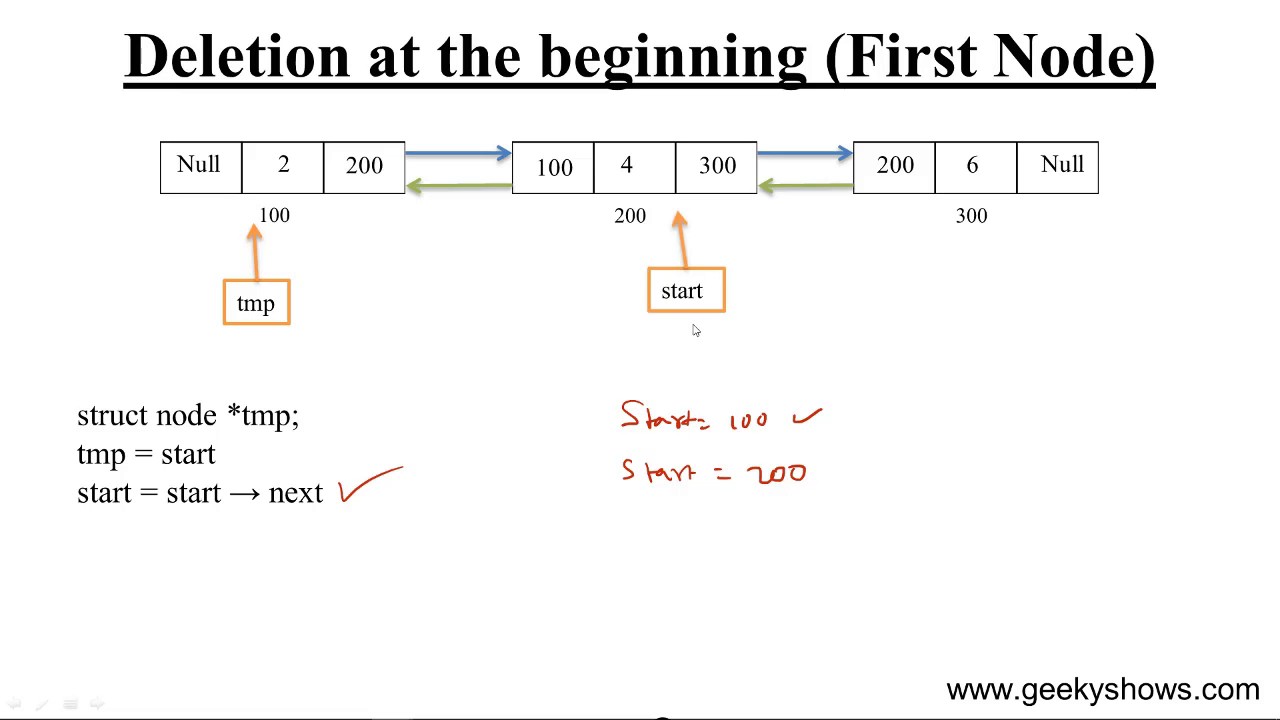
Step 2: If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.

Step 3: If it is not Empty then, define a Node pointer 'temp' and initialize with head.

Step 4: Check whether list is having only one node (temp → previous is equal to temp → next)

Step 5: If it is TRUE, then set head to NULL and delete temp (Setting Empty list conditions)

Step 6: If it is FALSE, then assign temp → next to head, NULL to head → previous and delete temp.



**Deleting a node at the end of the list**

Step 1: Check whether list is Empty (head == NULL)

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

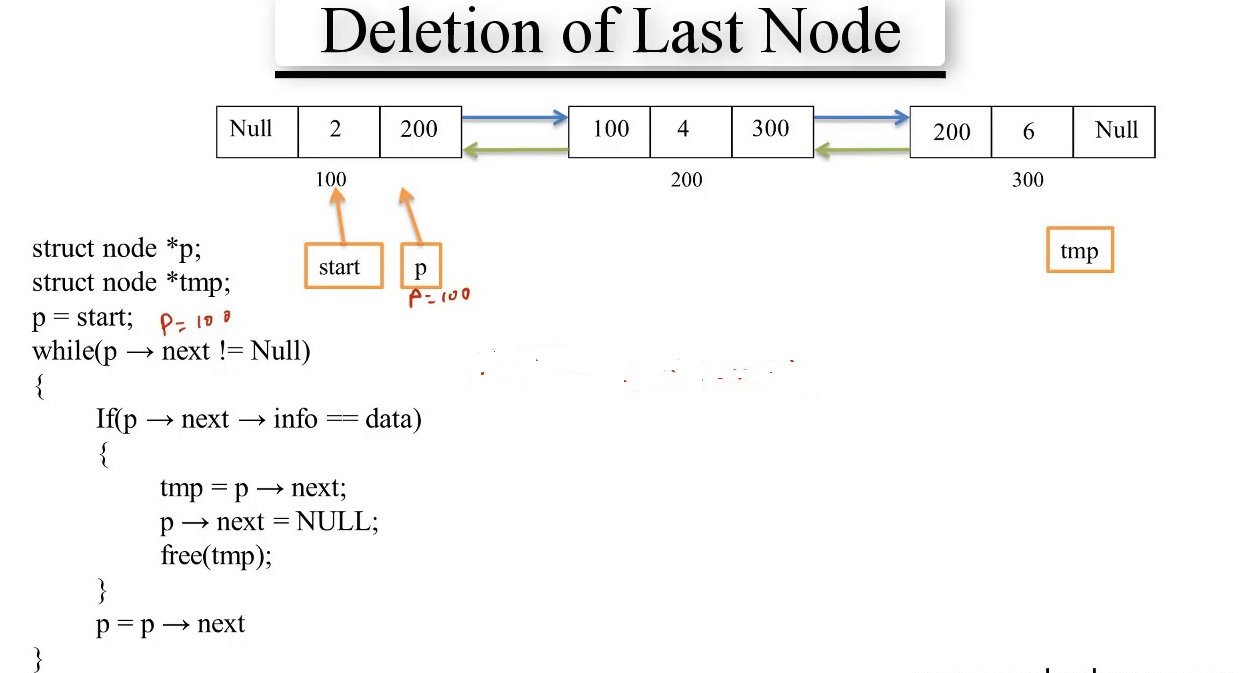
Step 3: If it is not Empty then, define a Node pointer 'temp' and initialize with head.

Step 4: Check whether list has only one Node (temp → previous and temp → next both are NULL)

Step 5: If it is TRUE, then assign NULL to head and delete temp. And terminate from the function. (Setting Empty list condition)

Step 6: If it is FALSE, then keep moving temp until it reaches to the last node in the list. (until temp → next is equal to NULL)

Step 7: Assign NULL to temp → previous → next and delete temp.

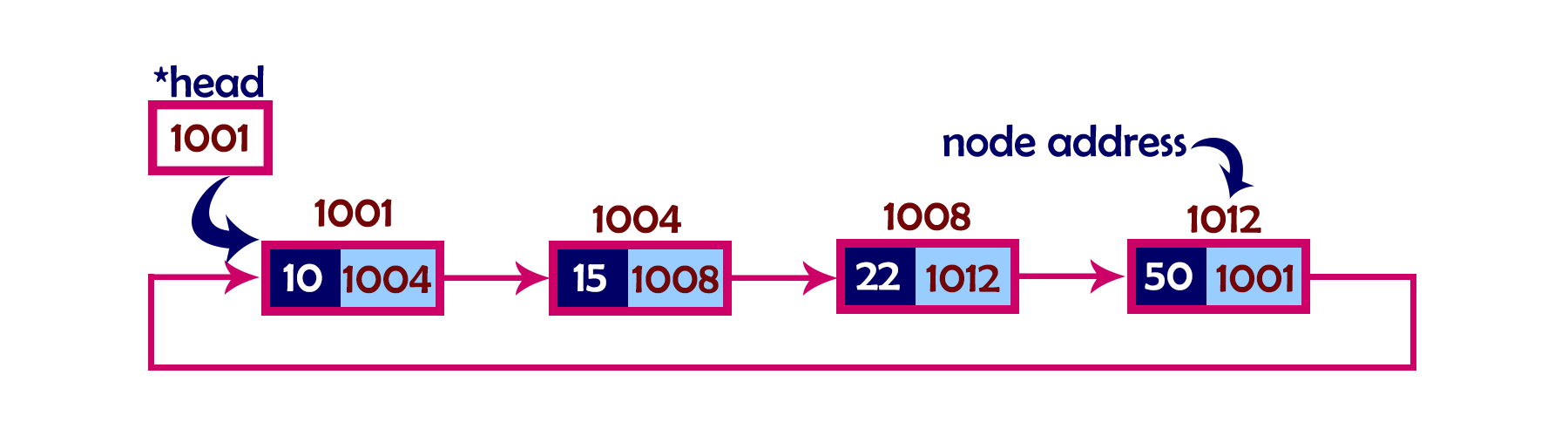


**Circular Linked list**

Circular linked list is a sequence of elements in which every element has link to its next element in the sequence and the last element has a link to the first element in the sequence.

That means circular linked list is similar to the single linked list except that the last node points to the first node in the list.

**Example**



A circular linked list can be either singly linked or doubly linked.

* for singly linked list, next pointer of last item points to the first item
* In doubly linked list, prev pointer of first item points to last item as well.

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

* Insertion
* Deletion
* Display

**Inserting At Beginning of the list**

We can use the following steps to insert a new node at beginning of the circular 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, set head = newNode and newNode→next = 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 (until 'temp → next == head').

Step 6: Set 'newNode → next =head', 'head = newNode' and 'temp → next = head'.

**Inserting At End of the list**

We can use the following steps to insert a new node at end of the circular 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, set head = newNode and newNode → next = 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 == head).

Step 6: Set temp → next = newNode and newNode → next = head.

**Deleting from Beginning of the list**

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

Step 1: Check whether list is Empty (head == 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 'temp1' and 'temp2' and initialize both 'temp1' and 'temp2' with head.

Step 4: Check whether list is having only one node (temp1 → next == head)

Step 5: If it is TRUE then set head = NULL and delete temp1 (Setting Empty list conditions)

Step 6: If it is FALSE move the temp1 until it reaches to the last node. (until temp1 → next == head )

Step 7: Then set head = temp2 → next, temp1 → next = head and delete temp2.

**Deleting from End of the list**

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

Step 1: Check whether list is Empty (head == 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 'temp1' and 'temp2' and initialize 'temp1' with head.

Step 4: Check whether list has only one Node (temp1 → next == head)

Step 5: If it is TRUE. Then, set head = NULL and delete temp1. And terminate from the function. (Setting Empty list condition)

Step 6: If it is FALSE. Then, set 'temp2 = temp1 ' and move temp1 to its next node. Repeat the same until temp1 reaches to the last node in the list. (until temp1 → next == head)

Step 7: Set temp2 → next = head and delete temp1.