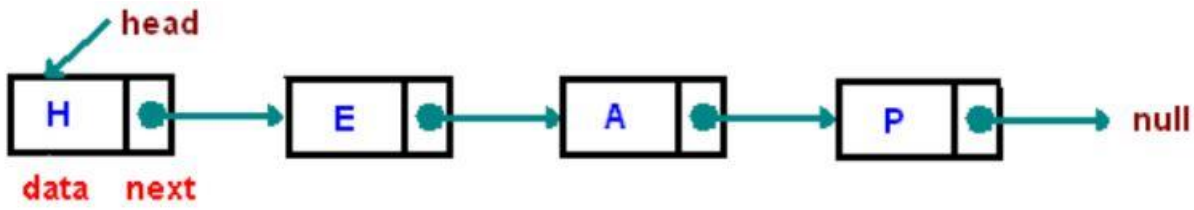
Linked Lists

Linked list is a linear data structure that consists of a sequence of elements where each element (usually called a node) comprises of two items - the data and a reference (link) to the next node. The last node has a reference to null. The entry point into a linked list is called the head (start) of the list. It should be noted that head is not a separate node, but the reference to the first node. If the list is empty then the start is a null reference. The list with no nodes –empty list or null list.



Note: The head is a pointer which points to the first node in the list. The implementation in the class, it has been discussed with the pointer's name as start. Head or start refers to the starting node. A linked list is a dynamic data structure. The number of nodes in a list is not fixed and can grow and shrink on demand. Any application which has to deal with an unknown number of objects will need to use a linked list.

Arrays –drawbacks

(1)The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. (static in nature)

(2) Inserting and a new element in an array of elements is expensive, because room has to be created for the new elements and to create room existing elements have to shifted. Same holds good for deletion also.

Advantages of linked list:

Efficient memory utilization: The memory of a linked list is not pre-allocated. Memory can be allocated whenever required. And it is de-allocated when it is no longer required.

Insertion and deletion operations are easier and efficient: Linked list provide flexibility in inserting a data item at a specified position and deletion of a data item from the given position.

Extensive manipulation:

We can perform any number of complex manipulations without any prior idea of the memory space available. (i.e. in stacks and queues we sometimes get overflow conditions. Here no such problem arises.)

Arbitrary memory locations

Here the memory locations need not be consecutive. They may be any arbitrary values. But even then the accessing of these items is easier as each data item contains within itself the address to the next data item. Therefore, the elements in the linked list are ordered not by their physical locations but by their logical links stored as part of the data with the node itself.

As they are dynamic data structures, they can grow and shrink during the execution of the program

Disadvantages of linked lists

 They have a tendency to use more memory due to pointers requiring extra storage space.

 Nodes in a linked list must be read in order from the beginning as linked lists are inherently sequentially accessed.(cannot be randomly accessed)

 Nodes are stored incontiguously, greatly increasing the time required to access individual elements within the list.

 Difficulties arise in linked lists when it comes to reverse traversing. For instance, singly linked lists are cumbersome to navigate backwards[1] and while doubly linked lists are somewhat easier to read, memory is wasted in allocating space for a back pointer.

Note : No particular data structure is the best. The choice of the data structure depends on the kind of application that needs to be implemented. While for some applications linked lists may be useful, for others, arrays may be useful.

Operations on linked lists

 Creation of a list

Creation operation is used to create a linked list. Once a linked list is created with one node, insertion operation can be used to add more elements in a node.

 Insertion of an element into a linked list

Insertion operation is used to insert a new node at any specified location in the linked list. A new node may be inserted.

(a) At the beginning of the linked list

(b) At the end of the linked list

(c) At any specified position in between in a linked list

 Deletion of a node from the linked list

Deletion operation is used to delete an item (or node) from the linked list. A nodemay be deleted from the

(a) Beginning of a linked list

(b) End of a linked list

(c) Specified location of the linked list

 Traversing and displaying the elements in the list

Traversing is the process of going through all the nodes from one end to another end of a linked list. In a singly linked list we can visit from lptr to rptr, forward traversing,nodes only. But in doubly linked list forward and backward traversing is possible

 Counting the number of elements in the list

 Searching for an element in the list

 Merging two lists(Concatenating lists)

Merging is the process of appending the second list to the end of the first list. Consider a list A having n nodes and B with m nodes. Then the operation concatenation will place the 1st node of B in the (n+1)th node in A. After concatenation A will contain (n+m) nodes

Representation of Linked List in C

A linked list is represented by a pointer to the first node of the linked list. The first node is called head. If the linked list is empty, then value of head is NULL.
Each node in a list consists of at least two parts:
1) data

2) Pointer (Or Reference) to the next node

In C, we can represent a node using structures. Below is an example of a linked list node with an integer data.

```
// A linked list node

struct Node

{

int data;

struct Node *next;

};
```

First Simple Linked List in C Let us create a simple linked list with 3 nodes.

```
// A simple C program to introduce a linked list

#include<stdio.h>

#include<stdlib.h>

struct Node

{

int data;

struct Node *next;
```

```
};


// Program to create a simple linked list with 3 nodes
int main()
{
struct Node* head = NULL;
struct Node* second = NULL;
struct Node* third = NULL;
// allocate 3 nodes in the heap
head = (struct Node*)malloc(sizeof(struct Node));
second = (struct Node*)malloc(sizeof(struct Node));
third = (struct Node*)malloc(sizeof(struct Node));
head->data = 1; //assign data in first node
head->next = second; // Link first node with
// the second node
// assign data to second node
second->data = 2;
// Link second node with the third node
second->next = third;
return 0;
}
```

Searching for a node in doubly linked list C function

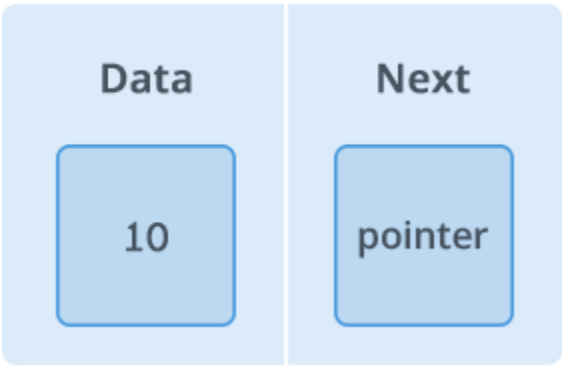
```
void search(int data) {
```

```
   int pos = 0;

   if(head==NULL) {
      printf("Linked List not initialized");
      return;
   }

   current = head;
   while(current!=NULL) {
      pos++;
      if(current->data == data) {
         printf("%d found at position %d\n", data, pos);
         return;
      }

      if(current->next != NULL)
         current = current->next;
      else
         break;
   }

   printf("%d does not exist in the list\n", data);
}
```

Singly Linked List

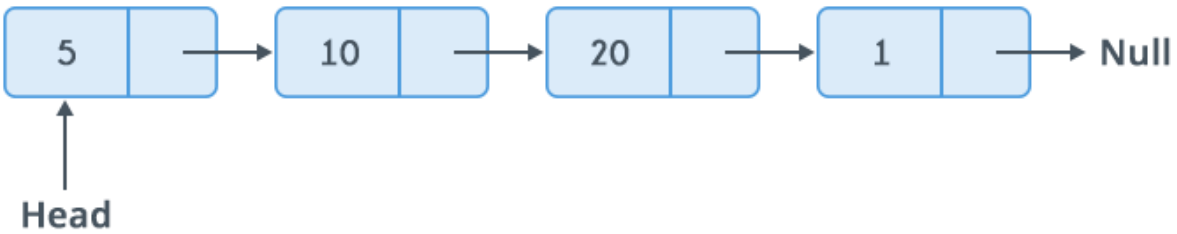
A **linked list** is a way to store a collection of elements. Like an array these can be character or integers. Each element in a linked list is stored in the form of a **node**.

Node:



A node is a collection of two sub-elements or parts. A **data** part that stores the element and a **next** part that stores the link to the next node.

Linked List:



A linked list is formed when many such nodes are linked together to form a chain. Each node points to the next node present in the order. The first node is always used as a reference to traverse the list and is called **HEAD**. The last node points to **NULL**.

Declaring a Linked list :

In C language, a linked list can be implemented using structure and pointers .

```
struct LinkedList{
    int data;
    struct LinkedList *next;
 };
```

The above definition is used to create every node in the list. The **data** field stores the element and the **next** is a pointer to store the address of the next node.

Noticed something unusual with next?

In place of a data type, **struct LinkedList** is written before next. That's because its a **self-referencing pointer**. It means a pointer that points to whatever it is a part of. Here **next** is a part of a node and it will point to the next node.

Creating a Node:

Let's define a data type of struct LinkedListto make code cleaner.

```
typedef struct LinkedList *node; //Define node as pointer of data type
struct LinkedList

node createNode(){
    node temp; // declare a node
    temp = (node)malloc(sizeof(struct LinkedList)); // allocate memory
using malloc()
    temp->next = NULL;// make next point to NULL
    return temp;//return the new node
}
```

typedef is used to define a data type in C.

malloc() is used to dynamically allocate a single block of memory in C, it is available in the header file stdlib.h.

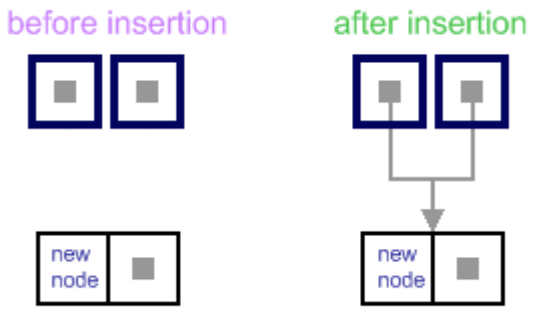
sizeof() is used to determine size in bytes of an element in C. Here it is used to determine size of each node and sent as a parameter to malloc.

Singly-linked list. Addition (insertion) operation.

Insertion into a singly-linked list has two special cases. It's insertion a new node before the head (to the very beginning of the list) and after the tail (to the very end of the list). In any other case, new node is inserted in the middle of the list and so, has a predecessor and successor in the list. There is a description of all these cases below.

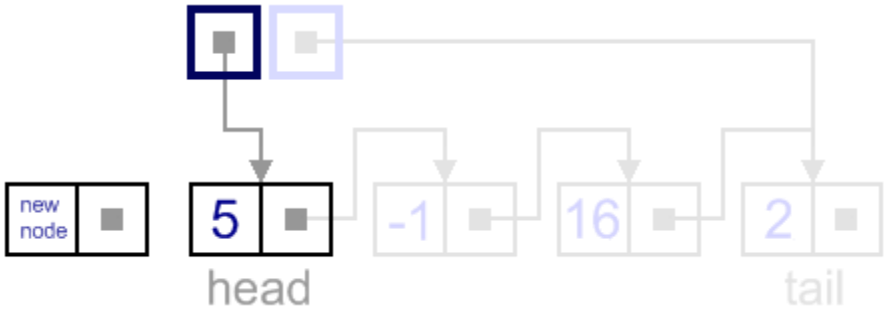
Empty list case

When list is empty, which is indicated by (head == NULL)condition, the insertion is quite simple. Algorithm sets both head and tail to point to the new node.



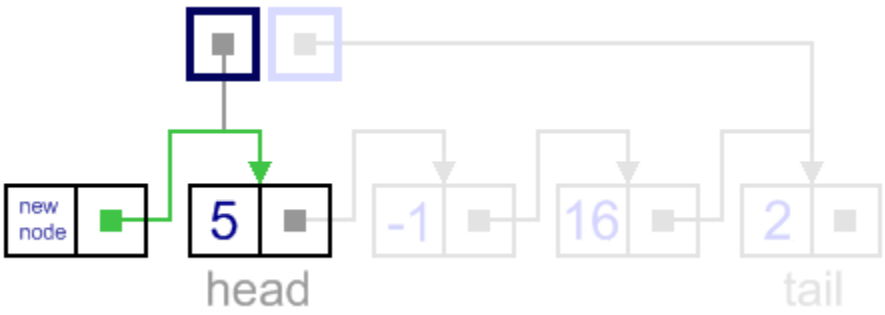
Add first

In this case, new node is inserted right before the current head node.

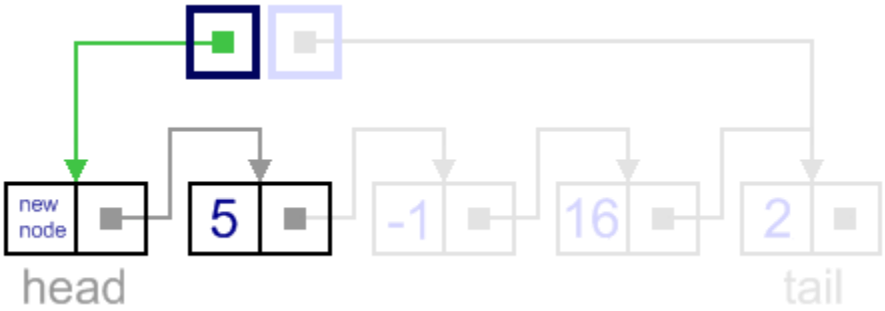


It can be done in two steps:

1. Update the next link of a new node, to point to the current head node.

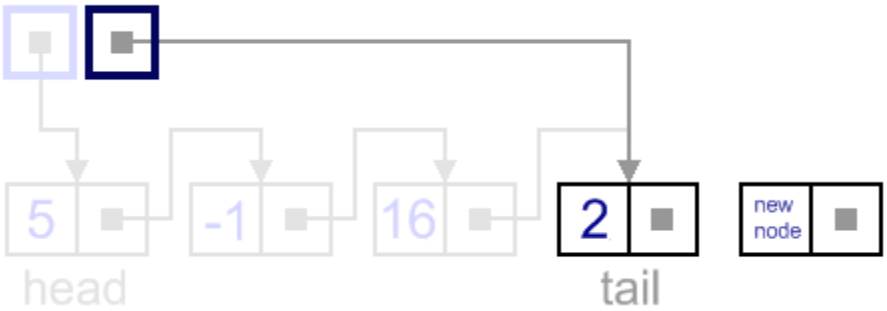


1.
2. Update head link to point to the new node.



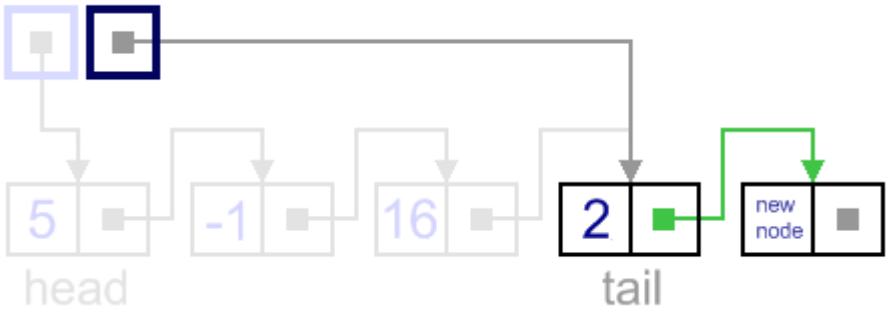
Add last

In this case, new node is inserted right after the current tail node.

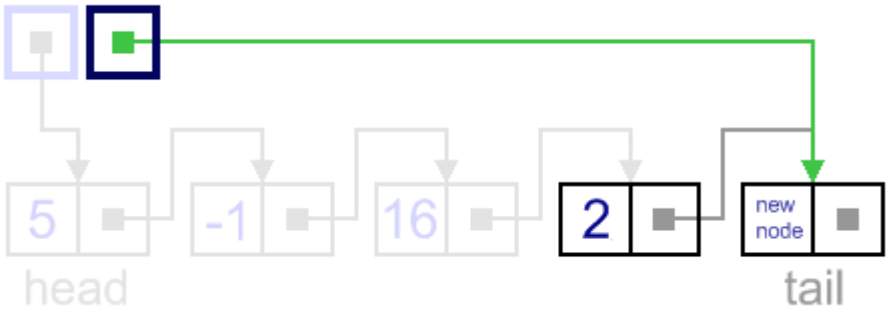


It can be done in two steps:

1. Update the next link of the current tail node, to point to the new node.

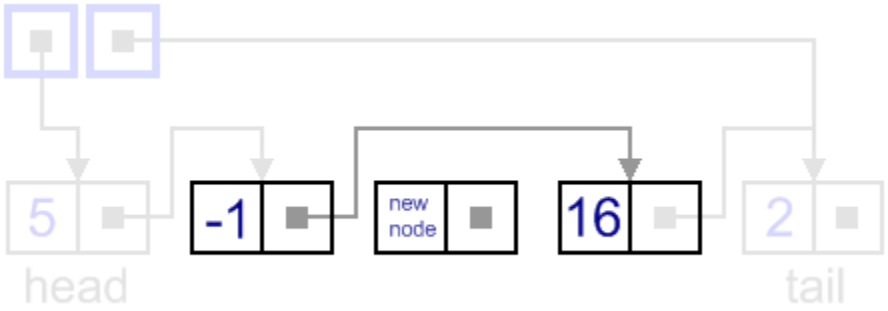


2. Update tail link to point to the new node.



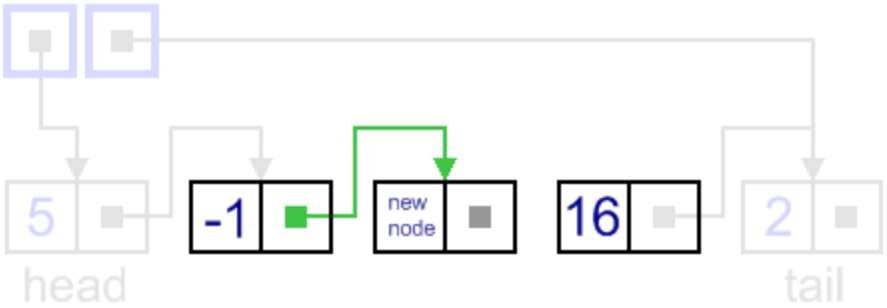
General case

In general case, new node is **always inserted between** two nodes, which are already in the list. Head and tail links are not updated in this case.

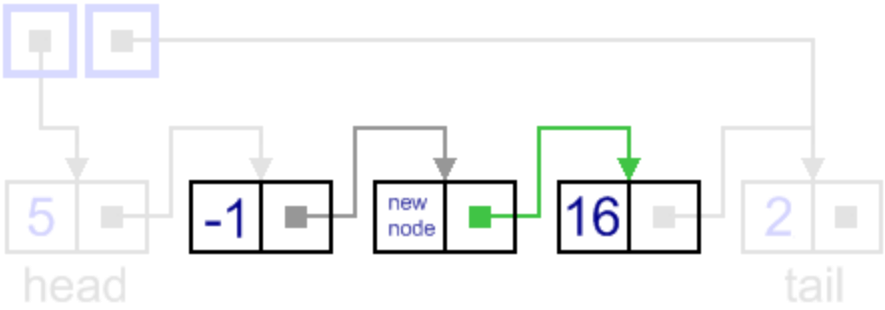


Such an insert can be done in two steps:

1. Update link of the "previous" node, to point to the new node.

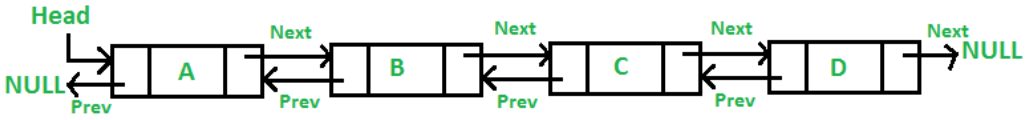


2. Update link of the new node, to point to the "next" node.



Doubly Linked List

A **D**oubly **L**inked **L**ist (DLL) contains an extra pointer, typically called *previous pointer*, together with next pointer and data which are there in singly linked list.



Following are advantages/disadvantages of doubly linked list over singly linked list.

Advantages over singly linked list

1) A DLL can be traversed in both forward and backward direction.

2) The delete operation in DLL is more efficient if pointer to the node to be deleted is given.

3) We can quickly insert a new node before a given node.

In singly linked list, to delete a node, pointer to the previous node is needed. To get this previous node, sometimes the list is traversed. In DLL, we can get the previous node using previous pointer.

Disadvantages over singly linked list

1) Every node of DLL Require extra space for an previous pointer. It is possible to implement DLL with single pointer though (See this and this).

2) All operations require an extra pointer previous to be maintained. For example, in insertion, we need to modify previous pointers together with next pointers. For example in following functions for insertions at different positions, we need 1 or 2 extra steps to set previous pointer.

Insertion

A node can be added in four ways

1) At the front of the DLL

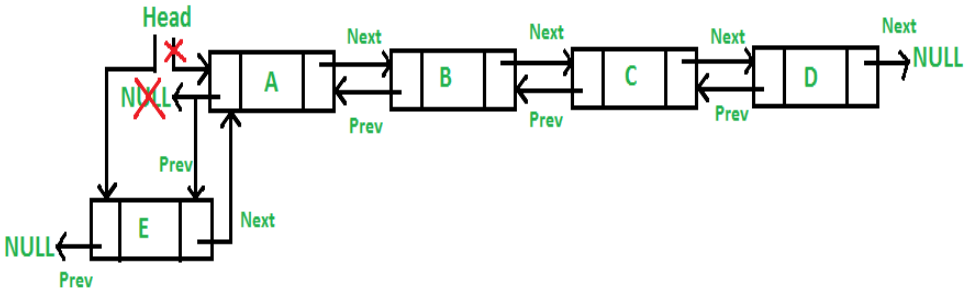
2) After a given node.

3) At the end of the DLL

4) Before a given node.

1) Add a node at the front: (A 5 steps process)

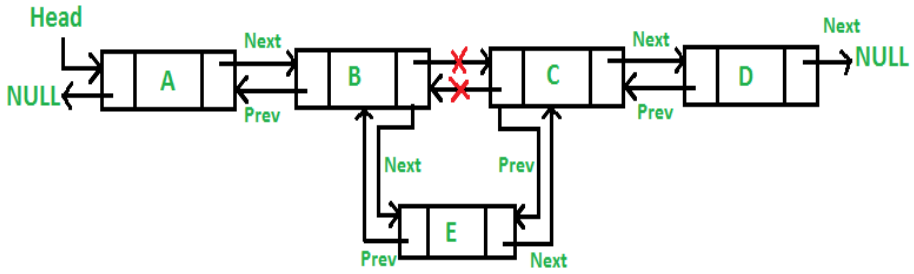
The new node is always added before the head of the given Linked List. And newly added node becomes the new head of DLL. For example if the given Linked List is 10152025 and we add an item 5 at the front, then the Linked List becomes 510152025. Let us call the function that adds at the front of the list is push(). The push() must receive a pointer to the head pointer, because push must change the head pointer to point to the new node (See this)



Four steps of the above five steps are same as the 4 steps used for inserting at the front in singly linked list. The only extra step is to change previous of head.

2) Add a node after a given node.: (A 7 steps process)

We are given pointer to a node as prev_node, and the new node is inserted after the given node.

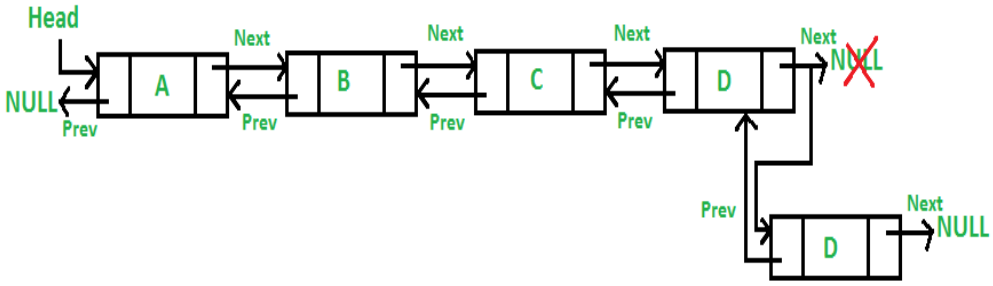


Five of the above steps step process are same as the 5 steps used for inserting after a given node in singly linked list. The two extra steps are needed to change previous pointer of new node and previous pointer of new node's next node.

3) Add a node at the end: (7 steps process)

The new node is always added after the last node of the given Linked List. For example if the given DLL is 510152025 and we add an item 30 at the end, then the DLL becomes 51015202530.

Since a Linked List is typically represented by the head of it, we have to traverse the list till end and then change the next of last node to new node.



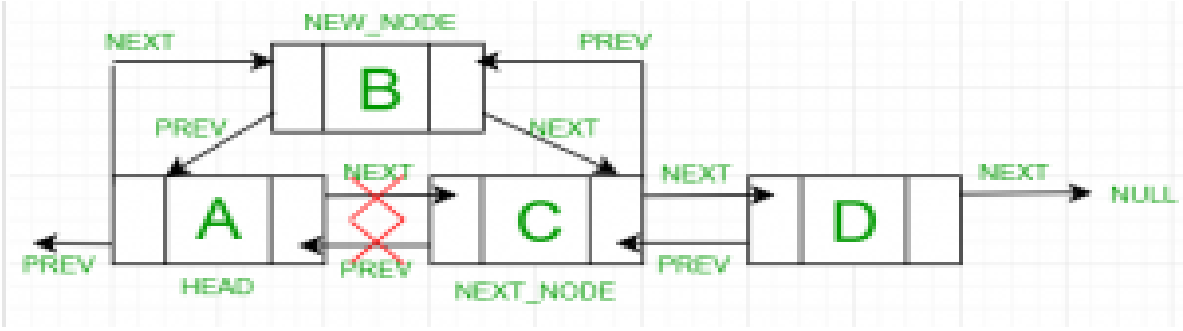
Six of the above 7 steps are same as the 6 steps used for inserting after a given node in singly linked list. The one extra step is needed to change previous pointer of new node.

4) Add a node before a given node:

Steps

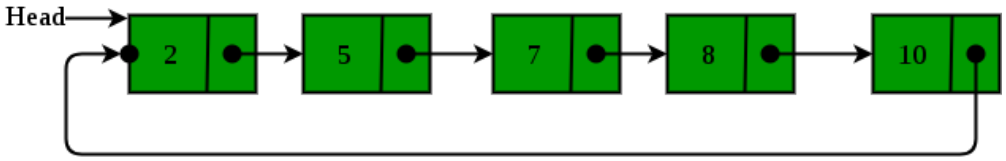
Let the pointer to this given node be next_node and the data of the new node to be added as new_data.

1. Check if the next_node is NULL or not. If it's NULL, return from the function because any new node can not be added before a NULL
2. Allocate memory for the new node, let it be called new_node
3. Set new_node->data = new_data
4. Set the previous pointer of this new_node as the previous node of the next_node, new_node->prev = next_node->prev
5. Set the previous pointer of the next_node as the new_node, next_node->prev = new_node
6. Set the next pointer of this new_node as the next_node, new_node->next = next_node;
7. If the previous node of the new_node is not NULL, then set the next pointer of this previous node as new_node, new_node->prev->next = new_node



Circular Linked List

Circular linked list *is a linked list where all nodes are connected to form a circle. There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly circular linked list.*



Advantages of Circular Linked Lists:
1) Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.

2) Useful for implementation of queue. Unlike this implementation, we don't need to maintain two pointers for front and rear if we use circular linked list. We can maintain a pointer to the last inserted node and front can always be obtained as next of last.

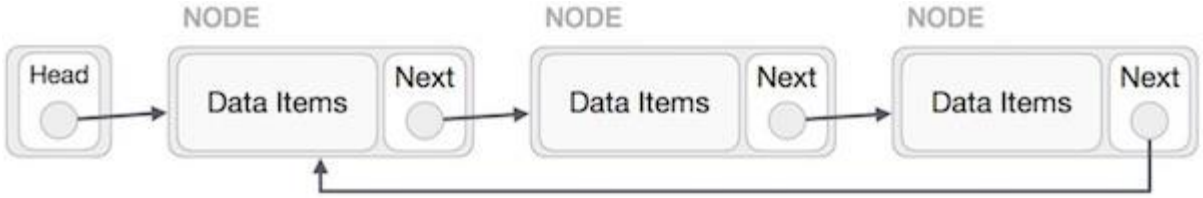
3) Circular lists are useful in applications to repeatedly go around the list. For example, when multiple applications are running on a PC, it is common for the operating system to put the

running applications on a list and then to cycle through them, giving each of them a slice of time to execute, and then making them wait while the CPU is given to another application. It is convenient for the operating system to use a circular list so that when it reaches the end of the list it can cycle around to the front of the list.

4) Circular Doubly Linked Lists are used for implementation of advanced data structures like Fibonacci Heap.

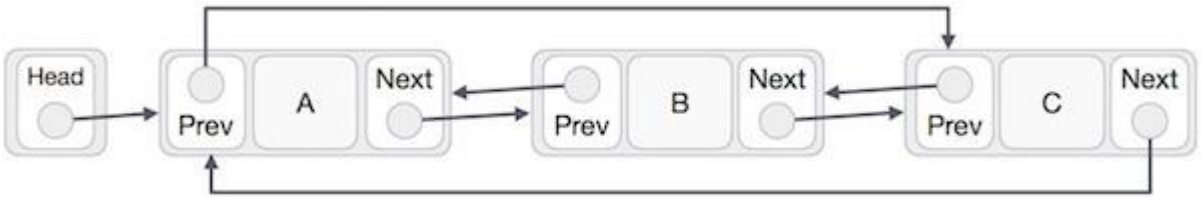
Singly Linked List as Circular

In singly linked list, the next pointer of the last node points to the first node.



Doubly Linked List as Circular

In doubly linked list, the next pointer of the last node points to the first node and the previous pointer of the first node points to the last node making the circular in both directions.



As per the above illustration, following are the important points to be considered.

- The last link's next points to the first link of the list in both cases of singly as well as doubly linked list.
- The first link's previous points to the last of the list in case of doubly linked list.

Basic Operations

Following are the important operations supported by a circular list.

- **insert** − Inserts an element at the start of the list.
- **delete** − Deletes an element from the start of the list.
- **display** − Displays the list.

Insertion Operation

Following code demonstrates the insertion operation in a circular linked list based on single linked list.

Example

```
//insert link at the first location
void insertFirst(int key, int data) {
   //create a link
   struct node *link = (struct node*) malloc(sizeof(struct node));
   link->key = key;
   link->data= data;

   if (isEmpty()) {
      head = link;
      head->next = head;
   } else {
      //point it to old first node
      link->next = head;

      //point first to new first node
      head = link;
   }
}
```

Deletion Operation

Following code demonstrates the deletion operation in a circular linked list based on single linked list.

```
//delete first item
struct node * deleteFirst() {
   //save reference to first link
   struct node *tempLink = head;

   if(head->next == head) {
      head = NULL;
      return tempLink;
```

```
   }

   //mark next to first link as first
   head = head->next;

   //return the deleted link
   return tempLink;
}
```

Display List Operation

Following code demonstrates the display list operation in a circular linked list.

```
//display the list
void printList() {
   struct node *ptr = head;
   printf("\n[ ");

   //start from the beginning
   if(head != NULL) {
      while(ptr->next != ptr) {
         printf("(%d,%d) ",ptr->key,ptr->data);
         ptr = ptr->next;
      }
   }

   printf(" ]");
}
```