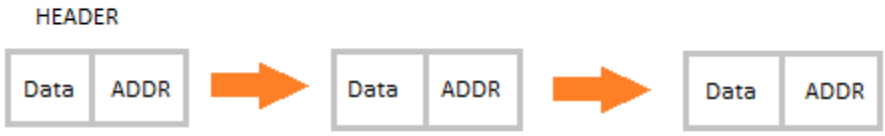
Introduction to Linked Lists

Linked List is a very commonly used linear data structure which consists of group of **nodes** in a sequence.

Each node holds its own **data** and the **address of the next node** hence forming a chain like structure.

Linked Lists are used to create trees and graphs.



Advantages of Linked Lists

- They are a dynamic in nature which allocates the memory when required.
- Insertion and deletion operations can be easily implemented.
- Stacks and queues can be easily executed.
- Linked List reduces the access time.

Disadvantages of Linked Lists

- The memory is wasted as pointers require extra memory for storage.
- No element can be accessed randomly; it has to access each node sequentially.
- Reverse Traversing is difficult in linked list.

Applications of Linked Lists

- Linked lists are used to implement stacks, queues, graphs, etc.
- Linked lists let you insert elements at the beginning and end of the list.
- In Linked Lists we don't need to know the size in advance.

Types of Linked Lists

There are 3 different implementations of Linked List available, they are:

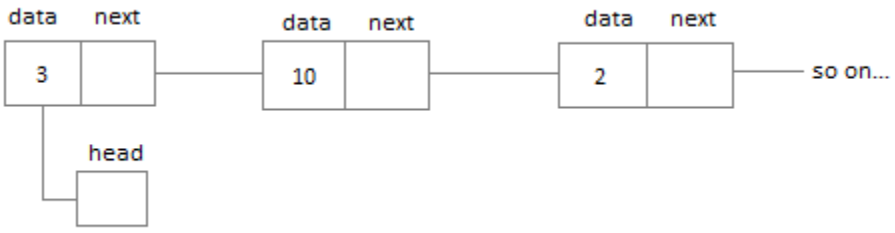
1. Singly Linked List
2. Doubly Linked List
3. Circular Linked List

Let's know more about them and how they are different from each other.

Singly Linked List

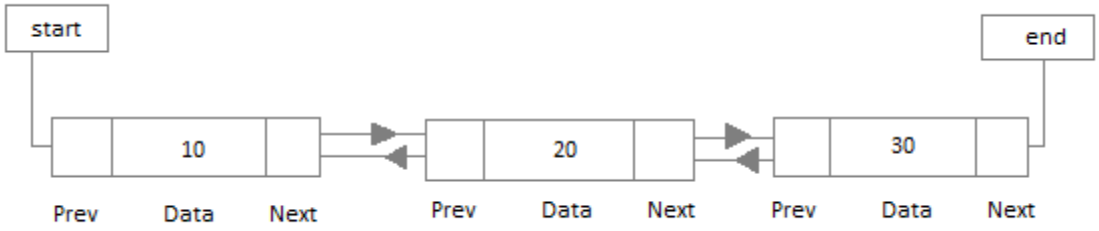
Singly linked lists contain nodes which have a **data** part as well as an **address part** i.e. `next`, which points to the next node in the sequence of nodes.

The operations we can perform on singly linked lists are **insertion**, **deletion** and **traversal**.



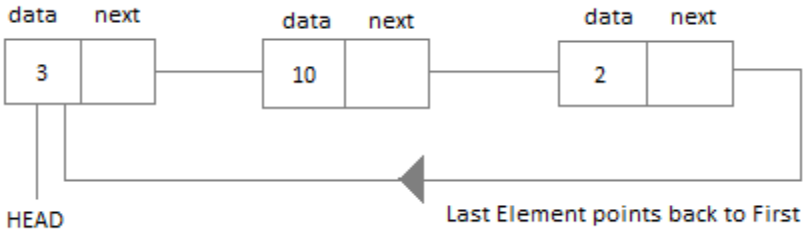
Doubly Linked List

In a doubly linked list, each node contains a **data** part and two addresses, one for the **previous** node and one for the **next** node.



Circular Linked List

In circular linked list the last node of the list holds the address of the first node hence forming a circular chain.



We will learn about all the 3 types of linked list, one by one, in the next tutorials. So click on **Next**button, let's learn more about linked lists.

Difference between Array and Linked List

Both Linked List and Array are used to store linear data of similar type, but an array consumes contiguous memory locations allocated at compile time, i.e. at the time of declaration of array, while for a linked list, memory is assigned as and when data is added to it, which means at runtime.

This is the basic and the most important difference between a linked list and an array. In the section below, we will discuss this in details along with highlighting other differences.

Linked List vs. Array

Array is a datatype which is widely implemented as a default type, in almost all the modern programming languages, and is used to store data of similar type.

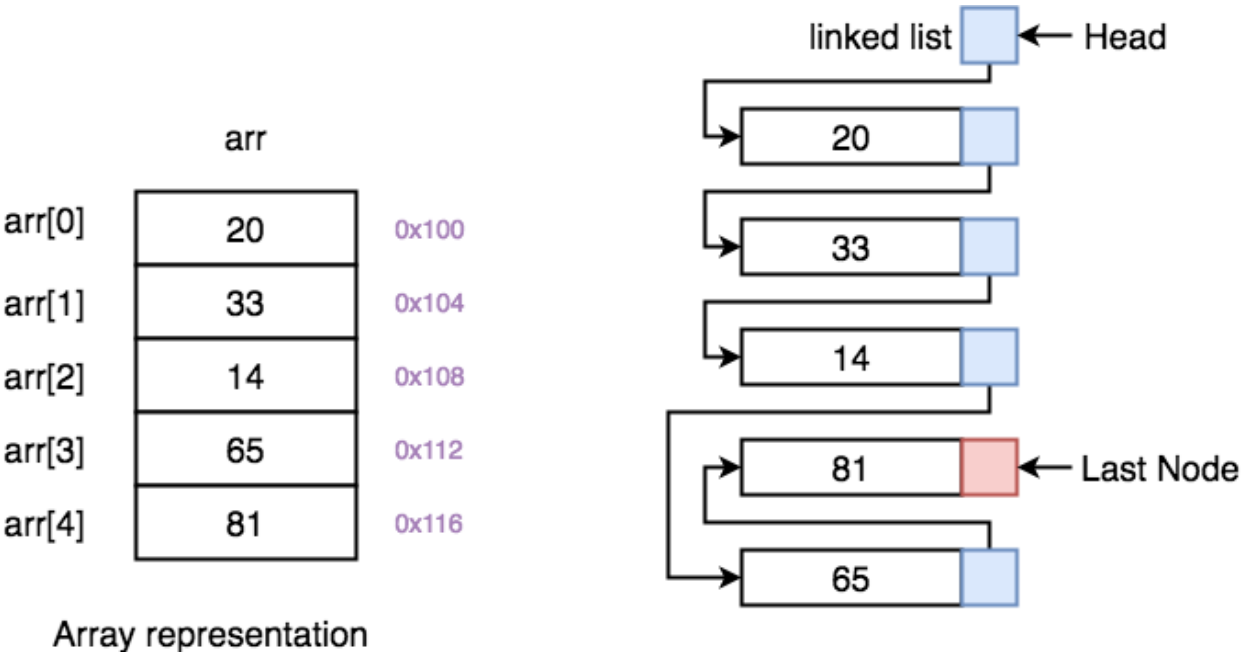
But there are many usecases, like the one where we don't know the quantity of data to be stored, for which advanced data structures are required, and one such data structure is **linked list**.

Let's understand how array is different from Linked list.

ARRAY	**LINKED LIST**
Array is a collection of elements of similar data type.	Linked List is an ordered collection of elements of same type, which are connected to each other using pointers.
Array supports **Random Access**, which means elements can be accessed directly using their index, like `arr[0]` for 1st element, `arr[6]` for 7th element etc. Hence, accessing elements in an array is **fast** with a constant time complexity of `O(1)`.	Linked List supports **Sequential Access**, which means to access any element/node in a linked list, we have to sequentially traverse the complete linked list, upto that element. To access **nth** element of a linked list, time complexity is `O(n)`.
In an array, elements are stored in **contiguous memory location** or consecutive manner in the memory.	In a linked list, new elements can be stored anywhere in the memory. Address of the memory location allocated to the new element is stored in the previous node of linked list, hence formaing a link between the two nodes/elements.
In array, **Insertion and Deletion**operation takes more time, as the memory locations are consecutive and fixed.	In case of linked list, a new element is stored at the first free and available memory location, with only a single overhead step of storing the address of memory location in the previous node of linked list. Insertion and Deletion operations are **fast** in linked list.

Memory is allocated as soon as the array is declared, at **compile time**. It's also known as **Static Memory Allocation**.	Memory is allocated at **runtime**, as and when a new node is added. It's also known as **Dynamic Memory Allocation**.
In array, each element is independent and can be accessed using it's index value.	In case of a linked list, each node/element points to the next, previous, or maybe both nodes.
Array can **single dimensional**, **two dimensional** or **multidimensional**	Linked list can be **Linear(Singly)**, **Doubly** or **Circular**linked list.
Size of the array must be specified at time of array decalaration.	Size of a Linked list is variable. It grows at runtime, as more nodes are added to it.
Array gets memory allocated in the **Stack**section.	Whereas, linked list gets memory allocated in **Heap**section.

Below we have a pictorial representation showing how consecutive memory locations are allocated for array, while in case of linked list random memory locations are assigned to nodes, but each node is connected to its next node using **pointer**.



On the left, we have **Array** and on the right, we have **Linked List**.

Why we need pointers in Linked List? [Deep Dive]

In case of array, memory is allocated in contiguous manner, hence array elements get stored in consecutive memory locations. So when you have to access any array element, all we have to do is use the array index, for example `arr[4]` will directly access the 5th memory location, returning the data stored there.

But in case of linked list, data elements are allocated memory at runtime, hence the memory location can be anywhere. Therefore to be able to access every node of the linked list, address of every node is stored in the previous node, hence forming a link between every node.

We need this additional **pointer** because without it, the data stored at random memory locations will be lost. We need to store somewhere all the memory locations where elements are getting stored.

Yes, this requires an additional memory space with each node, which means an additional space of `O(n)` for every `n` node linked list.

Linear Linked List

The element can be inserted in linked list in 2 ways :

- Insertion at beginning of the list.
- Insertion at the end of the list.

We will also be adding some more useful methods like :

- Checking whether Linked List is empty or not.
- Searching any element in the Linked List
- Deleting a particular Node from the List

Before inserting the node in the list we will create a class **Node**. Like shown below :

```
class Node {

  public:

  int data;

  //pointer to the next node

  node* next;


  node() {

    data = 0;

    next = NULL;
```

```
  }

  node(int x) {
    data = x;
    next = NULL;
  }
}
```

We can also make the properties `data` and `next` as private, in that case we will need to add the getter and setter methods to access them. You can add the getters and setter like this :

```
int getData() {
  return data;
}

void setData(int x) {
  this.data = x;
}

node* getNext() {
  return next;
}

void setNext(node *n) {
  this.next = n;
}
```

Node class basically creates a node for the data which you enter to be included into Linked List. Once the node is created, we use various functions to fit in that node into the Linked List.

Linked List class

As we are following the complete OOPS methodology, hence we will create a separate class for **Linked List**, which will have all its methods. Following will be the Linked List class :

```
class LinkedList {
  public:
  node *head;
  //declaring the functions
```

```
  //function to add Node at front
  int addAtFront(node *n);
  //function to check whether Linked list is empty
  int isEmpty();
  //function to add Node at the End of list
  int addAtEnd(node *n);
  //function to search a value
  node* search(int k);
  //function to delete any Node
  node* deleteNode(int x);

  LinkedList() {
    head = NULL;
  }
}
```

Insertion at the Beginning

Steps to insert a Node at beginning :

1. The first Node is the Head for any Linked List.
2. When a new Linked List is instantiated, it just has the Head, which is Null.
3. Else, the Head holds the pointer to the first Node of the List.
4. When we want to add any Node at the front, we must make the head point to it.
5. And the Next pointer of the newly added Node, must point to the previous Head, whether it be NULL(in case of new List) or the pointer to the first Node of the List.
6. The previous Head Node is now the second Node of Linked List, because the new Node is added at the front.

```
int LinkedList :: addAtFront(node *n) {
  int i = 0;
  //making the next of the new Node point to Head
  n->next = head;
  //making the new Node as Head
```

```
  head = n;
  i++;
  //returning the position where Node is added
  return i;
}
```

Inserting at the End

Steps to insert a Node at the end :

1. If the Linked List is empty then we simply, add the new Node as the Head of the Linked List.
2. If the Linked List is not empty then we find the last node, and make it' next to the new Node, hence making the new node the last Node.

```
int LinkedList :: addAtEnd(node *n) {
  //If list is empty
  if(head == NULL) {
    //making the new Node as Head
    head = n;
    //making the next pointe of the new Node as Null
    n->next = NULL;
  }
  else {
    //getting the last node
    node *n2 = getLastNode();
    n2->next = n;
  }
}

node* LinkedList :: getLastNode() {
  //creating a pointer pointing to Head
  node* ptr = head;
  //Iterating over the list till the node whose Next pointer points to null
  //Return that node, because that will be the last node.
  while(ptr->next!=NULL) {
```

```
    //if Next is not Null, take the pointer one step forward
    ptr = ptr->next;
  }
  return ptr;
}
```

Searching for an Element in the List

In searhing we do not have to do much, we just need to traverse like we did while getting the last node, in this case we will also compare the **data** of the Node. If we get the Node with the same data, we will return it, otherwise we will make our pointer point the next Node, and so on.

```
node* LinkedList :: search(int x) {
  node *ptr = head;
  while(ptr != NULL && ptr->data != x) {
    //until we reach the end or we find a Node with data x, we keep moving
    ptr = ptr->next;
  }
  return ptr;
}
```

Deleting a Node from the List

Deleting a node can be done in many ways, like we first search the Node with **data** which we want to delete and then we delete it. In our approach, we will define a method which will take the **data** to be deleted as argument, will use the search method to locate it and will then remove the Node from the List.

To remove any Node from the list, we need to do the following :

- If the Node to be deleted is the first node, then simply set the Next pointer of the Head to point to the next element from the Node to be deleted.
- If the Node is in the middle somewhere, then find the Node before it, and make the Node before it point to the Node next to it.

```
node* LinkedList :: deleteNode(int x) {
  //searching the Node with data x
  node *n = search(x);
```

```
  node *ptr = head;
  if(ptr == n) {
    ptr->next = n->next;
    return n;
  }
  else {
    while(ptr->next != n) {
      ptr = ptr->next;
    }
    ptr->next = n->next;
    return n;
  }
}
```

Checking whether the List is empty or not

We just need to check whether the **Head** of the List is `NULL` or not.

```
int LinkedList :: isEmpty() {
  if(head == NULL) {
    return 1;
  }
  else { return 0; }
}
```

Now you know a lot about how to handle List, how to traverse it, how to search an element. You can yourself try to write new methods around the List.

If you are still figuring out, how to call all these methods, then below is how your `main()` method will look like. As we have followed OOP standards, we will create the objects of **LinkedList** class to initialize our List and then we will create objects of **Node** class whenever we want to add any new node to the List.

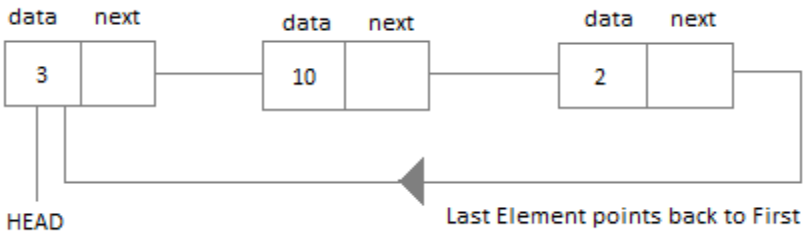
```
int main() {
  LinkedList L;
  //We will ask value from user, read the value and add the value to our Node
  int x;
  cout << "Please enter an integer value : ";
  cin >> x;
```

```
  Node *n1;
  //Creating a new node with data as x
  n1 = new Node(x);
  //Adding the node to the list
  L.addAtFront(n1);
}
```

Similarly you can call any of the functions of the LinkedList class, add as many Nodes you want to your List.

Circular Linked List

Circular Linked List is little more complicated linked data structure. In the circular linked list we can insert elements anywhere in the list whereas in the array we cannot insert element anywhere in the list because it is in the contiguous memory. In the circular linked list the previous element stores the address of the next element and the last element stores the address of the starting element. The elements points to each other in a circular way which forms a circular chain. The circular linked list has a dynamic size which means the memory can be allocated when it is required.



Application of Circular Linked List

- The real life application where the circular linked list is used is our Personal Computers, where multiple applications are running. All the running applications are kept in a circular linked list and the OS gives a fixed time slot to all for running. The Operating System keeps on iterating over the linked list until all the applications are completed.
- Another example can be Multiplayer games. All the Players are kept in a Circular Linked List and the pointer keeps on moving forward as a player's chance ends.
- Circular Linked List can also be used to create Circular Queue. In a Queue we have to keep two pointers, FRONT and REAR in memory all the time, where as in Circular Linked List, only one pointer is required.

Implementing Circular Linked List

Implementing a circular linked list is very easy and almost similar to linear linked list implementation, with the only difference being that, in circular linked list the last **Node** will have it's **next** point to the **Head** of the List. In Linear linked list the last Node simply holds NULL in it's next pointer.

So this will be oue Node class, as we have already studied in the lesson, it will be used to form the List.

```
class Node {
  public:
  int data;
  //pointer to the next node
  node* next;

  node() {
    data = 0;
    next = NULL;
  }

  node(int x) {
    data = x;
    next = NULL;
  }
}
```

Circular Linked List

Circular Linked List class will be almost same as the Linked List class that we studied in the previous lesson, with a few difference in the implementation of class methods.

```
class CircularLinkedList {
  public:
  node *head;
  //declaring the functions

  //function to add Node at front
  int addAtFront(node *n);
```

```
  //function to check whether Linked list is empty
  int isEmpty();
  //function to add Node at the End of list
  int addAtEnd(node *n);
  //function to search a value
  node* search(int k);
  //function to delete any Node
  node* deleteNode(int x);

  CircularLinkedList() {
    head = NULL;
  }
}
```

Insertion at the Beginning

Steps to insert a Node at beginning :

1. The first Node is the Head for any Linked List.
2. When a new Linked List is instantiated, it just has the Head, which is Null.
3. Else, the Head holds the pointer to the fisrt Node of the List.
4. When we want to add any Node at the front, we must make the head point to it.
5. And the Next pointer of the newly added Node, must point to the previous Head, whether it be NULL(in case of new List) or the pointer to the first Node of the List.
6. The previous Head Node is now the second Node of Linked List, because the new Node is added at the front.

```
int CircularLinkedList :: addAtFront(node *n) {
  int i = 0;
  /* If the list is empty */
  if(head == NULL) {
    n->next = head;
    //making the new Node as Head
    head = n;
    i++;
```

```
  }
  else {
    n->next = head;
    //get the Last Node and make its next point to new Node
    Node* last = getLastNode();
    last->next = n;
    //also make the head point to the new first Node
    head = n;
    i++;
  }
  //returning the position where Node is added
  return i;
}
```

Insertion at the End

Steps to insert a Node at the end :

1. If the Linked List is empty then we simply, add the new Node as the Head of the Linked List.
2. If the Linked List is not empty then we find the last node, and make it' next to the new Node, and make the next of the Newly added Node point to the Head of the List.

```
int CircularLinkedList :: addAtEnd(node *n) {
  //If list is empty
  if(head == NULL) {
    //making the new Node as Head
    head = n;
    //making the next pointer of the new Node as Null
    n->next = NULL;
  }
  else {
    //getting the last node
    node *last = getLastNode();
    last->next = n;
    //making the next pointer of new node point to head
```

```
    n->next = head;
  }
}
```

Searching for an Element in the List

In searhing we do not have to do much, we just need to traverse like we did while getting the last node, in this case we will also compare the **data** of the Node. If we get the Node with the same data, we will return it, otherwise we will make our pointer point the next Node, and so on.

```
node* CircularLinkedList :: search(int x) {
  node *ptr = head;
  while(ptr != NULL && ptr->data != x) {
    //until we reach the end or we find a Node with data x, we keep moving
    ptr = ptr->next;
  }
  return ptr;
}
```

Deleting a Node from the List

Deleting a node can be done in many ways, like we first search the Node with **data** which we want to delete and then we delete it. In our approach, we will define a method which will take the **data** to be deleted as argument, will use the search method to locate it and will then remove the Node from the List.

To remove any Node from the list, we need to do the following :

- If the Node to be deleted is the first node, then simply set the Next pointer of the Head to point to the next element from the Node to be deleted. And update the next pointer of the Last Node as well.
- If the Node is in the middle somewhere, then find the Node before it, and make the Node before it point to the Node next to it.
- If the Node is at the end, then remove it and make the new last node point to the head.

```
node* CircularLinkedList :: deleteNode(int x) {
  //searching the Node with data x
```

```
  node *n = search(x);
  node *ptr = head;
  if(ptr == NULL) {
    cout << "List is empty";
    return NULL;
  }
  else if(ptr == n) {
    ptr->next = n->next;
    return n;
  }
  else {
    while(ptr->next != n) {
      ptr = ptr->next;
    }
    ptr->next = n->next;
    return n;
  }
}
```

Header Linked List

Header Linked List

Definition:

A header linked list is linked list that always contain a special note at the front of the list, this special node is called *header node*. It does not contain actual data item included in the list but usually contains some useful information about the entire linked list.

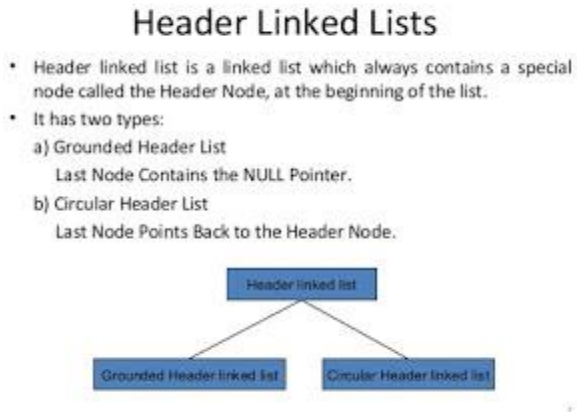
Header Linked List

Header linear linked list

Such as

- · Total number of nodes in the list .
- · pointer to the last node in the list or to the last node accessed the header node in the list is never deleted that always point to the first node in the list.

Widely used types of header linked list are:



1.A *Grounded Header* list is the header list where the last node contains the null pointer . grounded comes from the fact that many text use the electrical ground symbol to indicate the null pointer.

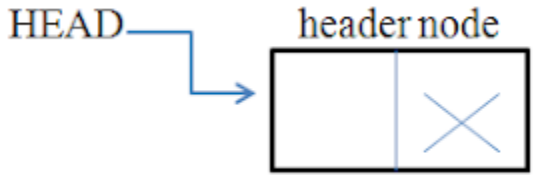
- · Location Of first node is written as:

  PTRàLINK(HEAD)

  Where as in singly linked list we write it as PTRàHEAD

- · Condition to check empty header linked list:

  LINK(HEAD)=NULL



2. *circular header list* is a header list where the last more points back to the header node .the term node by itself normally refer to an ordinary node not the header node, when used with header list. Thus the first node in header is the node following the header node and the list.

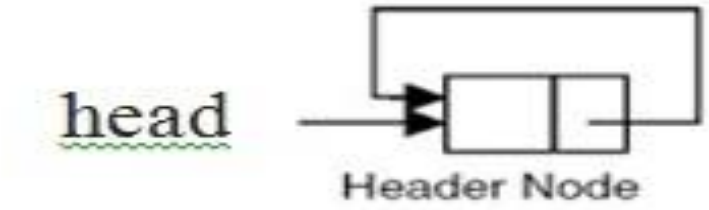
location of the first node is:

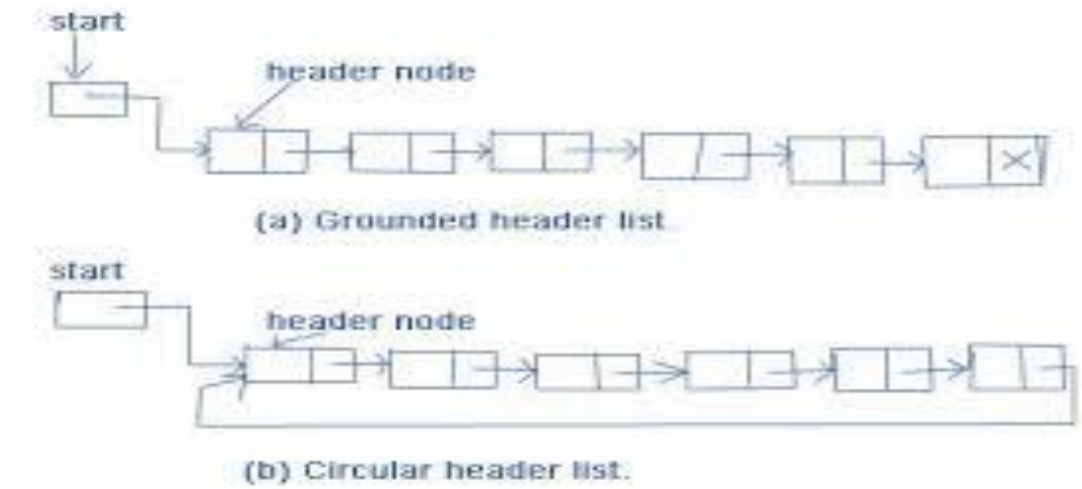
- · Location Of first node is written as:

  PTRàLINK(HEAD)

- · Condition to check empty Circular header linked list:

LINK(HEAD)=HEAD





Advantage: the main advantage of using head and node in a linked list if that you can avoid the special testing case while inserting and deleting nodes from the linked list.

- The special cases inserting or deleting a node either at or from the beginning of the list or from the empty list. In single linked list special cases were handed separately.
- As header linked list always contains at least one node insertion and deletion to take place only after the head and the special cases of inserting for deletion at the front will never occur to perform the operation on a header linked list.
- While travelling in order to point to the first we have to use the statement similarly by inserting or deleting a node from sorted header linked list we need to maintain pointer.